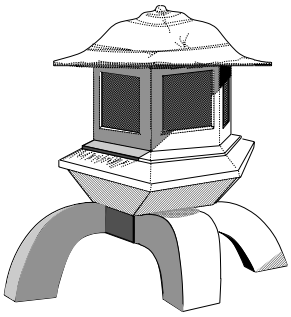
[For HOL Kananaskis-13] August 20, 2019

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 [4].) 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 [3], 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 [10] 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.

In Memory of Mike Gordon

As documented in the academic literature, in material available from his archived webpages at the University of Cambridge Computer Laboratory, and in these manuals, Mike Gordon was HOL's primary creator and developer. Mike not only created a significant piece of software, inspiring this and many other projects since, but also built a world-leading research group in the Computer Laboratory. This research environment was wonderfully productive for many of the system's authors, and we all owe Mike an enormous debt for both the original work on HOL, and for the way he and his group supported our own development as researchers and HOL hackers.

Mike Gordon, 1948–2017

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 1.2); the quotation, parsing, and printing of logical types and terms (Section 1.3); the representation of theorems (Section 1.5); the representation of theories (Section 1.8); the fundamental HOL theory `bool` (Section 5.2.1); the primitive rules of inference (Section 1.6); and the methods for extending theories (throughout Section 1.8 and also later in Section 7.3). 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 Types

The allowed types depend on which type constants have been declared in the current theory. See Section 1.8 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 warn-

ing 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 $\sigma_1 \to \sigma_2$ 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="min", Args=[]} evaluates to an ML value of type hol_type 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="min", 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

1
```
> mk_thy_type{Tyop="fun", Thy="min",
              Args=[mk_vartype "'a",
                    mk_thy_type{Tyop="bool", Thy="min", Args=[]}]}
val it = ":α -> bool": hol_type
```

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.

1.2 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 5.2.1.2.

At any time, the terms that may be constructed depends on which constants have been declared in the current theory. See Section 1.8 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 =` σ`}` 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 1.8).

```
mk_comb : (term * term) -> term
```

`mk_comb(`t_1`,`t_2`)` evaluates to a term representing the combination t_1 t_2. It fails if:

(i) the type of t_1 does not have the form σ'`->`σ;

(ii) the type of t_1 has the form σ'`->`σ, but the type of t_2 is not equal to σ'.

```
mk_abs : (term * term) -> term
```

`mk_abs(`x`,`t`)` evaluates to a term representing the abstraction $\lambda x.\ t$; it fails if x is not a variable.

There are four primitive destructor functions on terms:

```
dest_var       : term -> (string * 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.

1.3 Quotations, Parsing and Printing

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",                                    2
              Args = [mk_thy_type{Tyop = "bool", Thy = "min", Args = []},
                      mk_thy_type{Tyop = "bool", Thy = "min", Args = []}]}
val it = ":bool -> bool": hol_type
```

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

```
> val numty = mk_thy_type{Tyop="num",Thy="num",Args=[]}                    1
val numty = ":num": hol_type
> 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}))));
val it = "λx. x + 1": term
> val parsed = ``\x. x + 1``;
val parsed = "λx. x + 1": term
```

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 $\alpha \to (ind \to bool)$ would be printed out as below:

```
> let                                                                    2
    val ind_ty = mk_thy_type{Tyop="ind", Thy="min", Args=[]}
  in
    alpha --> (ind --> bool)
  end;
val it = ":α -> ind -> bool": hol_type
```

Similarly, terms are printed in the form "*term*", as in the session above printing the term "$\lambda x.\ x + 1$". The leading colon is used to distinguish a type quotation from a term quotation: the former have the form ": ⋯ " and the latter have the form " ⋯ ".

1.3.1 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 1.3.1.1 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 1.3.1.1 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.

1.3.1.1 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 7.1.3. The dollar-sign (`$`) can also be used to form symbolic identifiers, but only in tokens where it is the only symbol. Thus, `$`, `$$`, and `$$$` will all lex as identifiers.

This restriction arises because of the other uses to which the dollar sign is put:

- The dollar can be used as an escaping mechanism to remove special syntactic treatment of other identifiers. Thus, `$+` and `$if` are effectively special forms of the tokens `+` and `if` respectively.
- Finally, the dollar can also be used as a namespace separator character, giving unambiguous "long form" identifiers. For example, the token `bool$COND` is an unambiguous way of writing the `COND` constant from theory segment `bool`.

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 5.3.3.

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 descendant 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 `*)`).

1.3.1.2 Literals

There are two classes of literal in HOL's term syntax: numbers and strings (which latter also includes a treatment of character literals). String literals are a convenient way to write large terms of type `char list`; numerals are a convenient way to write values of type `num`. In addition, both string and numeric literals can be injected into other types, so that, for example, it is possible to write `23` and have the system see it as a rational number. For more on these syntaxes, see Section 5.3.3 for numerals, and Section 5.4.4 for strings.

1.3.2 Unicode *vs* ASCII

As the definition of `parsed` in the session above suggests, there are ASCII alternatives to the Unicode syntax. To repeat that definition:

```
> val parsed = ``\x. x + 1``;                                    3
val parsed = “λx. x + 1”: term
```

The backslash can be used instead of λ, and the ‘‘ symbol can be used instead of both the “ and ” symbols. Similarly, the logical connectives have both ASCII and Unicode forms, and either can be used (even within the same term), except that the begin- and end-delimiters must be of the same type. Usually the system prefers to print its output with the Unicode symbols. Thus:

4
```
> val t1 = ``!x y. x < y ==> ?z. x + z = y``;
val t1 = “∀x y. x < y ⇒ ∃z. x + z = y”: term

> val t2 = “∀x. ∃y. x < y ∧ y < x + 1”;
val t2 = “∀x. ∃y. x < y ∧ y < x + 1”: term

> val t3 = ``!x. ∃y. x < y /\ y < x + 1``;
val t3 = “∀x. ∃y. x < y ∧ y < x + 1”: term
```

Section 7.1 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.

1.3.3 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)``;
Exception-
Type inference failure: unable to infer a type for the application of

$= (x :num)

on line 1, characters 3-13

which has type

:num -> bool

to

(x :num) = (1 :num)

on line 1, characters 8-12

which has type

:bool

unification failure message: Attempt to unify different type operators: num$num and min$bool
HOL_ERR
  {message =
   "on line 1, characters 8-12:\n\nType inference failure: unable to infer a type for the appli
   origin_function = "type-analysis", origin_structure = "Preterm"} raised
```

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),
         numSyntax.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.

1.3.4 Parentheses and Precedence

As with programming languages, the grammar governing the parsing of terms and types includes a notion of precedence. As with standard mathematics, for example, if we write `2 + 3 * 6`, we expect this to denote the abstract syntax tree that groups the `3` and `6` together, giving a value of 20 for the term. Again as is usual, to adjust parses in the face of precedence one can use parentheses:

3

```
> EVAL “2 + 3 * 6”;
val it = ⊢ 2 + 3 * 6 = 20: thm

> EVAL “(2 + 3) * 6”;
val it = ⊢ (2 + 3) * 6 = 30: thm
```

Function application can be seen as an invisible high-precedence (or “tightly binding”) infix operator so that `f x + 6` is an addition term, with the application of `f` to `x` being added to `6`. This makes HOL syntax more like functional programming: one typically doesn't bother to write `f(x)` because `f x` suffices. Of course, sometimes arguments do need parentheses. For example, `f(x + 6)`.

Finally, drawing inspiration from the Haskell programming language, HOL also supports the dollar-sign as a *low* precedence function application symbol. In this way, one has an option that can result in needing to write fewer parentheses:

4

```
> EVAL “FACT $ SUC $ 2 + 3”;
val it = ⊢ FACT (SUC (2 + 3)) = 720: thm
```

Note that, as above, the pretty-printer will always print terms with the “invisible” function application symbol and parentheses as necessary, even if they were input with dollar-signs.

1.3.5 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`, `ind`, and `bool`.

```
> type_grammar();                                                          5
val it =
   Rules:
     (50)   TY  ::=  TY -> TY [fun] (R-associative)
     (60)   TY  ::=  TY + TY [sum] (R-associative)
     (70)   TY  ::=  TY # TY [prod] (R-associative)
            TY  ::=  bool | (TY, TY) fun | ind | TY itself | TY list | num |
                     one | TY option | (TY, TY) prod | TY recspace | TY set |
                     (TY, TY) sum | unit
            TY  ::=  TY[TY] (array type)
   Type abbreviations:
     bool              = min$bool
     (α, β) fun        = (α, β) min$fun
     ind               = min$ind
     α itself          = α bool$itself
     α list            = α list$list
     num               = num$num
     one               = one$one                  [not printed]
     α option          = α option$option
     (α, β) prod       = (α, β) pair$prod
     α recspace        = α ind_type$recspace
     α set             = (α, min$bool) min$fun   [not printed]
     (α, β) sum        = (α, β) sum$sum
     unit              = one$one                : type_grammar.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 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.

1.3.6 Namespace control

In order to provide convenience, the parser deals with overloading and ambiguity. Overloading of numeric literals is discussed in Section 5.3.3.1, 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.

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

```
> 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()`

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.

1.4 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*`$`*op*	`mk_thy_type{Thy="`*thy*`", Tyop="`*op*`",Args=[]}`
Function type	`:`σ_1`->`σ_2	σ_1`-->`σ_2
Compound type	`:(`σ_1`,...,`σ_n`)`*op*	`mk_type("`*op*`",[`σ_1`,...,`σ_n`])`
	`:(`σ_1`,...,`σ_n`)`*thy*`$`*op*	`mk_thy_type{Thy="`*thy*`", Tyop="`*op*`",Args=[`σ_1`,...,`σ_n`]}`

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	t_1 t_2	`mk_comb(`t_1`,`t_2`)`
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`,[`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`)`

1.5 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.[1] 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[2] 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)
```

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

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

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

```
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 1.6.)

It was mentioned in *LOGIC* that the deductive system of HOL includes four axioms.[3] 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 1.8.1 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):

2
```
> BOOL_CASES_AX;
val it = ⊢ ∀t. (t ⟺ T) ∨ (t ⟺ F): thm
```

Note the special print format, with the ⊢ notation used to indicate ML type `thm` status; as well as the absence of HOL quotation marks in the ML context.

The session below illustrates the use of the destructor functions:

1
```
> hyp BOOL_CASES_AX;
val it = []: term list

> concl BOOL_CASES_AX;
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`.

[3] 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
```

1.6 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.

1.6.1 Assumption introduction

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

$$\frac{}{t \;\texttt{|-}\; t}$$

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

1.6.2 Reflexivity

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

$$\frac{}{\texttt{|-}\; t = t}$$

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

1.6.3 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 ``(\x.t1)t2``` evaluates to the theorem `|-` $(\backslash x.t_1)t_2$ `=` $t_1[t_2/x]$. Failure occurs if the argument to `BETA_CONV` is not a β-redex (*i.e.* is not of the form $(\backslash x.t_1)t_2$.

1.6.4 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 occurs 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 $1 \leq i \leq n$.

1.6.5 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 occurs if x is not a variable, or x occurs free in any assumption in Γ.

1.6.6 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 $\sigma_1, \ldots, \sigma_n$ for the type variables $\alpha_1, \ldots, \alpha_n$ 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}
```

1.6.7 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`.

1.6.8 Modus Ponens

```
MP : thm -> thm -> thm
```

$$\frac{\Gamma_1 \ |- \ t_1 \ ==> \ t_2 \qquad \Gamma_2 \ |- \ t_1}{\Gamma_1 \cup \Gamma_2 \ |- \ t_2}$$

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.

1.7 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[4] 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.
```

1.8 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.

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

1.8.1 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 respond with a list of the argument's ancestors in the theory hierarchy.

```
> ancestry "-";                                                          2
val it =
   ["ConseqConv", "quantHeuristics", "patternMatches", "ind_type", "while",
    "one", "sum", "option", "pair", "combin", "sat", "normalForms",
    "relation", "min", "bool", "marker", "num", "prim_rec", "arithmetic",
    "numeral", "basicSize", "numpair", "pred_set", "list", "rich_list",
    "indexedLists"]: string list
```

Creating a theory segment New theory segment is 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 occurs 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. In addition, various tools can be primed to pay particular attention to saved theorems through the use of special attributes. Such attributes are indicated by appending the list of the attribute names to the *name*. Thus, to indicate the `simp` attribute (for which, see discussion of the stateful simpset in Section 7.5.2.5), one can write

```
save_thm("name[simp]", th)
```

Multiple attributes can be listed between the square brackets, separated by commas.

Another attribute, `local`, can be used to create theorems that will *not* be exported to disk, but which are important locally. Such local theorems can have other attributes attached to them, which will have their effect within the given session/script-file.

Equivalently, a special `Theorem` syntax is available for use in script files. One can write

```
Theorem name = th
```

to achieve the same effect as an SML declaration

```
val name = save_thm("name", th);
```

Attributes can also be added; for example:

```
Theorem name[simp] = th
```

If one wishes to prove a goal with a tactic (see Chapter 4), and store the resulting theorem, the combination of these actions can be achieved with the `store_thm` function:

```
store_thm : (string * term * tactic) -> thm
```

A call to `store_thm(`*name*`,`*t*`,`*tac*`)` results in the application of *tac* to goal *t*. If the tactic is successful, the resulting theorem is saved under the name *name*, as before. Also as before, theorem attributes can be added to the name.

Finally, again for use in script files only, there is a `Theorem` syntax to replace `store_thm`, reducing the need to write the same name twice, and giving a cleaner appearance. One can write

```
Theorem name[attr1,attr2,...]:
  ...goal statement...
Proof
  ...tactic...
QED
```

The goal statement in this form is not an arbitrary term value, but must use the surface syntax used by the system parser (see Sections 1.3 and 7.1). For example:

```
Theorem IMP_CLAUSE[simp]:
  !p. (p ==> p) <=> T
Proof
  rpt strip_tac
QED
```

Note further that the `Proof` and `QED` keywords must occur in the leftmost column of the script file so that the parser can know when the term and tactic arguments terminate.

One last special form is available to abbreviate the `Theorem-local` combination (with either the = or : following). Instead of writing `Theorem foo[local,...]`, one can write `Triviality foo[...]`.

The choice to use `Theorem` (or `Triviality`) syntax, or to use `store_thm` or `save_thm` directly is a matter of users' aesthetic preference. But recall: if writing library code to prove and store theorems, the underlying `store_thm` must be used, as the special treatment of the `Theorem` keyword is only available in script files.

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 8.3), which maintains dependencies among collections of HOL theory segments.

1.8.2 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`.

1.8.3 Functions for creating definitional extensions

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

1.8.3.1 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 $v_1, \ldots, v_n$ 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 $v_1, \ldots, v_n$ (this is equivalent to requiring $\lambda v_1\ \cdots\ v_n.\ t$ to be a closed term);

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

1.8.3.2 Constant specifications

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

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

such that:

(i) $c_1, \ldots, c_n$ are distinct names;

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

(iii) $tyvars(\lambda x_{1\sigma_1} \cdots x_{n\sigma_n}.\ t_{bool})\ \subseteq\ tyvars(\sigma_i)$ for $1 \leq i \leq n$;

(iv) $\exists x_{1\sigma_1}\ \cdots\ x_{n\sigma_n}.\ t\ \in\ \mathsf{Theorems}_{\mathcal{T}}$.

The following ML function is used to make constant specifications in the HOL system.

```
new_specification : string * string list * thm -> thm
```

Evaluating:

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

simultaneously introduces new constants named $c_1, \ldots, c_n$ 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) $c_1, \ldots, c_n$ are not distinct variables;

(iv) the type of some c_i does not contain all the type variables which occur in the term `\`$x_1\ \cdots\ x_n.\ \ t[x_1, \ldots, x_n]$.

1.8.3.3 Type definitions

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

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

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

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

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 1.6, 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.

1
```
> 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, t1 ⇒ t2] ⊢ t2: thm
```

More briefly, one could evaluate the following, and 'count' the invocations of functions representing primitive inferences. Here, the `Count.apply` function counts the number of primitive inferences performed after the function is applied to the argument. In the first invocation, this means that only the *modus ponens* step is counted. We create an artificial function to see the count of all 3 in the second interaction:

2
```
> Count.apply (MP (ASSUME ``t1 ==> t2``)) (ASSUME ``t1:bool``);
runtime: 0.00002s,    gctime: 0.00000s,     systime: 0.00000s.
Axioms: 0, Defs: 0, Disk: 0, Orcl: 0, Prims: 1; Total: 1
val it =  [t1, t1 ⇒ t2] ⊢ t2: thm

> fun f () = MP (ASSUME ``t1 ==> t2``) (ASSUME ``t1:bool``);
val f = fn: unit -> thm
> Count.apply f ();
runtime: 0.00009s,    gctime: 0.00000s,     systime: 0.00001s.
Axioms: 0, Defs: 0, Disk: 0, Orcl: 0, Prims: 3; Total: 3
val it =  [t1, t1 ⇒ t2] ⊢ t2: thm
```

Each of the three inference steps of the abstract proof corresponds to the application of an ML function in the performance of the proof in HOL; and each of the ML functions corresponds to a primitive inference of the deductive system.

It is worth emphasising that, in either case, every primitive inference in the proof chain is made, in the sense that for each inference, the corresponding ML function is evaluated. That is, HOL permits no short-cut around the necessity of performing complete proofs. The short-cut provided by derived inference rules (as implemented in ML) is around the necessity of *specifying* every step; something that would be impossible for a proof of any

length. It can be seen from this that the derived rule, and its representation as an ML function, is essential to the HOL methodology; theorem proving would be otherwise impossible.

There are, of course, an infinite number of proofs, of the 'form' shown in the example, that can be conducted in HOL: one for every pair of ``:bool``-typed terms. Moreover, every time a theorem of the form

$$t_1 \Rightarrow t_2,\ t_1 \vdash t_2$$

is required, its proof must be constructed anew. To capture the general pattern of inference, an ML function can be written to implement an inference rule as a derivation from the primitive inferences. Abstractly, a *derived inference rule* is a rule that can be justified on the basis of the primitive inference rules (and/or the axioms). In the present case, the rule required 'undischarges' assumptions. It is specified for HOL by

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

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

```
> fun UNDISCH th = MP th (ASSUME(fst(dest_imp(concl th))));                    3
val UNDISCH = fn: thm -> thm
```

This provides a function that maps a theorem to a theorem; that is, performs proofs in HOL. The following session illustrates the use of the derived rule, on a consequence of the axiom IMP_ANTISYM_AX. (The inferences are counted.) Assume that the printing of theorems has been adjusted as above and th is bound as shown below:

```
> th;                                                                          1
val it =  [] ⊢ (t1 ⇒ t2) ⇒ (t2 ⇒ t1) ⇒ (t1 ⟺ t2): thm

> Count.apply UNDISCH th;
runtime: 0.00000s,    gctime: 0.00000s,     systime: 0.00000s.
Axioms: 0, Defs: 0, Disk: 0, Orcl: 0, Prims: 2; Total: 2
val it =  [t1 ⇒ t2] ⊢ (t2 ⇒ t1) ⇒ (t1 ⟺ t2): thm

> Count.apply UNDISCH it;
runtime: 0.00001s,    gctime: 0.00000s,     systime: 0.00001s.
Axioms: 0, Defs: 0, Disk: 0, Orcl: 0, Prims: 2; Total: 2
val it =  [t1 ⇒ t2, t2 ⇒ t1] ⊢ t1 ⟺ t2: thm
```

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. The implementation is partly described in Chapter 3.

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.

```
> Count.apply (REWRITE_RULE [arithmeticTheory.GREATER_DEF])                    1
    (ASSUME ``SUC 4 > 0 <=> (SUC 3 > 0 <=> (SUC 2 > 0 <=>
                (SUC 1 > 0 <=> SUC 0 > 0)))``);
runtime: 0.00004s,    gctime: 0.00000s,     systime: 0.00000s.
Axioms: 0, Defs: 0, Disk: 0, Orcl: 0, Prims: 14; Total: 14
val it =

   [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))):
   thm
```

Notice that rewriting equations can be extracted from universally quantified theorems. To construct the proof step-wise, with all of the instantiations, substitutions, 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.

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 5.2.1), the eight primitive inferences of the HOL logic, and inferences defined earlier in the sequence.

[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.

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 family of theorems. No single theorem can be expressed in the HOL logic that is equivalent to BETA_CONV. (See Chapter 3 for further discussion of this point.) (UNDISCH is not a rule of this sort, as it can, in fact, be expressed as a theorem.)

For each derivation given below, there is an ML function definition in the HOL system that implements the derived rule as a procedure in ML. The actual implementation in the HOL system differs in some cases from the derivations given here, since the system code has been optimised for improved performance.

In addition, for reasons that are mostly historical, not all the inferences that are derived in terms of the abstract logic are actually derived in the current version of the HOL system. That is, there are currently a number of rules that are installed in the system on an 'axiomatic' basis, all of which should be derived by explicit inference. These rules' 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.

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]

2.3.1 Adding an assumption

```
ADD_ASSUM : term -> thm -> thm
```

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

[2]This stands for 'conversion', as explained in Chapter 3.

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 : thm
```

$$\vdash \top$$

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 ∀]
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 (∀-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 ∀]
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.[3]

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]

[3] 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
```

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

- 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
```

$$\frac{\Gamma_1 \vdash \exists x.\, t[x] \qquad\qquad \Gamma_2,\ t[v] \vdash t'}{\Gamma_1 \cup \Gamma_2 \vdash t'}$$

- 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
```

$$\frac{\Gamma \vdash t_1 \wedge t_2}{\Gamma \vdash t_1 \qquad \Gamma \vdash t_2}$$

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 ∨-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 ∨-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 ¬]
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

Conversions

A *conversion* in HOL is a rule that maps a term to a theorem expressing the equality of that term to some other term. An example is the rule for β-conversion:

$$(\backslash x.t_1)t_2 \quad \mapsto \quad \texttt{|-}\ (\backslash x.t_1)t_2 = t_1[t_2/x]$$

Theorems of this sort are used in HOL in a variety of contexts, to justify the replacement of particular terms by semantically equivalent terms.

The ML type of conversions is `conv`:

```
type conv = term -> thm
```

For example, `BETA_CONV` is an ML function of type `conv` (*i.e.* a conversion) that expresses β-conversion in HOL. It produces the appropriate equational theorem on β-redexes and fails elsewhere.

2

```
> BETA_CONV;
val it = fn: term -> thm

> BETA_CONV ``(\x. (\y. (\z. x + y + z)3)2) 1``;
val it = ⊢ (λx. (λy. (λz. x + y + z) 3) 2) 1 = (λy. (λz. 1 + y + z) 3) 2: thm

> BETA_CONV ``(\y. (\z. 1 + (y + z))3) 2``;
val it = ⊢ (λy. (λz. 1 + (y + z)) 3) 2 = (λz. 1 + (2 + z)) 3: thm

> BETA_CONV ``(\z. 1 + (2 + z)) 3``;
val it = ⊢ (λz. 1 + (2 + z)) 3 = 1 + (2 + 3): thm

> BETA_CONV ``1 + (2 + 3)``;
Exception- HOL_ERR
  {message = "not a beta-redex", origin_function = "BETA_CONV",
   origin_structure = "Thm"} raised
```

The basic conversions, as well as a number of those commonly used, are provided in HOL. There are also groups of application-specific conversions to be found in some of the libraries. (Of those provided, some are derived and some, like `BETA_CONV` are taken as axiomatic[1].) In addition, HOL provides a collection of ML functions enabling users

[1] A list of the axiomatic rules was supplied in Section 1.6.

to define new conversions (as well as new rules and tactics) as functions of existing ones. Some of these are described in Sections 3.2 and 3.3. The notion of conversions is inherited from Cambridge LCF; the underlying principles are described in [11, 12].

Conversions such as BETA_CONVrepresent infinite families of equations[2]. They are particularly useful in cases in which it is impossible to state, within the logic, a single axiom or theorem instantiable to every equation in a family.[3] Instead, an ML procedure returns the instance of the desired theorem for any given term. This is also the reason that quite a few of the other rules in HOL are not stated instead as axioms or theorems. As rules, conversions are distinguished with an ML type abbreviation simply because there are relatively many of them with the same type, and because they return equational theorems that lend themselves directly to term rewriting.[4] In many HOL applications, the main use of conversions is to produce these equational theorems. A few examples of conversions are illustrated below.

2

```
> NOT_FORALL_CONV ``~!x. (f:'a->'a) x = g x``;
val it = ⊢ ¬(∀x. f x = g x) ⟺ ∃x. f x ≠ g x: thm

> CONTRAPOS_CONV ``(!x. f x = g x) ==> ((f:'a->'a) = g)``;
val it = ⊢ (∀x. f x = g x) ⇒ f = g ⟺ f ≠ g ⇒ ¬∀x. f x = g x: thm

> SELECT_CONV ``(@f:'a->'a. f x = g x)x = g x``;
val it = ⊢ (@f. f x = g x) x = g x ⟺ ∃f. f x = g x: thm

> EXISTS_UNIQUE_CONV ``?!z. (f:'a->'a) z = g z``;
val it =
   ⊢ (∃!z. f z = g z) ⟺
     (∃z. f z = g z) ∧ ∀z z'. f z = g z ∧ f z' = g z' ⇒ z = z': thm
```

An example of an application specific conversion is numLib's num_CONV:

[2]This was also mentioned in Section 1.6.

[3]In the case of β-conversion specifically, it is the substitution of one term for another in a context that is inexpressible; but in general, there is a variety of reasons that arise.

[4]In fact, some ML functions have names with the suffix '_CONV' but do not have the type conv; SUBST_CONV, for example, has type (thm # term) list -> term -> conv. Those that eventually produce conversion are thought of as 'conversion schemas'.

2

```
> numLib.num_CONV ``2``;
val it = ⊢ 2 = SUC 1: thm

> numLib.num_CONV ``1``;
val it = ⊢ 1 = SUC 0: thm

> numLib.num_CONV ``0`` handle e => Raise e;
Exception-
Exception raised at Num_conv.num_CONV:
Term either not a numeral or zero
HOL_ERR
  {message = "Term either not a numeral or zero", origin_function =
   "num_CONV", origin_structure = "Num_conv"} raised
```

Another application of conversions, related to the first, is in the implementation of the existing rewriting tools, REWRITE_CONV (Section 3.5), REWRITE_RULE (Section 2.2) and REWRITE_TAC (Chapter 4), which are central to theorem proving in HOL. This use is explained in Section 3.5, both as an example and because users may have occasion to construct rewriting tools of their own design, by similar methods. The next section introduces the conversion-building tools in general.

3.1 Indicating Unchangedness

All conversions may raise the special exception Conv.UNCHANGED on an input term t, as a "short-hand" instead of returning the reflexive theorem |- t = t. This is done for efficiency reasons. All of the connectives described below in Section 3.2 handle this exception appropriately. The standard function QCONV is provided to automatically handle this exception in contexts where it would be inappropriate (typically where a conversion is being called to provide a theorem directly). QCONV's implementation is

```
fun QCONV c t = c t handle UNCHANGED => REFL t
```

3.2 Conversion Combining Operators

A term u is said to *reduce* to a term v by a conversion c if there exists a finite sequence of terms $t_1, t_2, \ldots, t_n$ such that:

(i) $u = t_1$ and $v = t_n$;

(ii) $c\ t_i$ evaluates to the theorem |- $t_i = t_{i+1}$ for $1 \leq i < n$;

(iii) The evaluation of $c\ t_n$ fails.

The first session of this chapter illustrates the reduction of the term

```
(\x. (\y. (\z. x + y + z)3)2)1
```

to `1 + (2 + 3)` by the conversion `BETA_CONV`, in a reduction sequence of length four:

```
(\x. (\y. (\z. x + (y + z))3)2)1
(\y. (\z. 1 + (y + z))3)2
(\z. 1 + (2 + z))3
1 + (2 + 3)
```

That is, `BETA_CONV` applies to each term of the sequence, except the fourth and last, to give a theorem equating that term to the next term. Therefore, each term of the sequence, from the second on, can be extracted from the theorem for the previous term; namely, it is the right hand side of the conclusion. The whole reduction can therefore be accomplished by repeated application of `BETA_CONV` to the terms of the sequence as they are generated.

To transform `BETA_CONV` to achieve this effect, two operators on conversions are introduced. The first one, infixed, is `THENC`, which sequences conversions.

```
op THENC : conv -> conv -> conv
```

If $c_1\ t_1$ evaluates to Γ_1 `|-` t_1=t_2 and $c_2\ t_2$ evaluates to Γ_2 `|-` t_2=t_3, then (`(`c_1 `THENC` c_2`)` t_1 evaluates to $\Gamma_1 \cup \Gamma_2$ `|-` t_1=t_3. If the evaluation of $c_1\ t_1$ or the evaluation of $c_2\ t_2$ fails, then so does the evaluation of c_1 `THENC` c_2. `THENC` is justified by the transitivity of equality.

The second, also infixed, is `ORELSEC`; this applies a second conversion if the application of the first one fails.

```
op ORELSEC : conv -> conv -> conv
```

(`(`c_1 `ORELSEC` c_2`)` t evaluates to $c_1\ t$ if that evaluation succeeds, and to $c_2\ t$ otherwise. (The failure to evaluate is detected via the `ML` failure construct.)

The functions `THENC` and `ORELSEC` are used to define the desired operator, `REPEATC`, which successively applies a conversion until it fails:

```
REPEATC : conv -> conv
```

`REPEATC` c is intuitively equivalent to:

```
(c THENC c THENC ... THENC c THENC ...) ORELSEC ALL_CONV
```

It is defined recursively:[5]

[5]Note that because ML is a call-by-value language, the extra argument t is needed in the definition of `REPEATC`; without it the definition would loop. There is a similar problem with the tactical `REPEAT`; see Chapter 4.

```
fun REPEATC c t = ((c THENC (REPEATC c)) ORELSEC ALL_CONV) t
```

The current example term can thus be completely reduced by use of BETA_CONV transformed by the REPEATC operator:

```
> REPEATC BETA_CONV;                                                    2
val it = fn: conv

> REPEATC BETA_CONV ``(\x. (\y. (\z. x + y + z)3)2)1``;
val it = ⊢ (λx. (λy. (λz. x + y + z) 3) 2) 1 = 1 + 2 + 3: thm
```

BETA_CONV applies to terms of a certain top level form only, namely to β-redexes, and fails on terms of any other form. In addition, no number of repetitions of BETA_CONV will β-reduce *arbitrary* β-redexes embedded in terms. For example, the term shown below fails even at the top level because it is not a β-redex:

```
> BETA_CONV ``(((\x.(\y.(\z. x + y + z))) 1) 2) 3``;                    2
Exception- HOL_ERR
  {message = "not a beta-redex", origin_function = "BETA_CONV",
   origin_structure = "Thm"} raised

> is_abs ``(((\x.(\y.(\z. x + y + z))) 1) 2)``;
val it = false: bool
```

The β-redex (\w.w)3 is not affected in the third input of the session shown below, because of its position in the structure of the whole term. This is so even though the whole term is reduced, and the subterm at top level could be reduced:

```
> BETA_CONV ``(\z. x + y + z) 3``;                                      2
val it = ⊢ (λz. x + y + z) 3 = x + y + 3: thm

> BETA_CONV ``(\w. w) 3``;
val it = ⊢ (λw. w) 3 = 3: thm

> REPEATC BETA_CONV ``(\z. x + y + z) ((\w. w) 3)``;
val it = ⊢ (λz. x + y + z) ((λw. w) 3) = x + y + (λw. w) 3: thm
```

To produce, from a conversion c, a conversion that applies c to every subterm of a term, the function DEPTH_CONV can be applied to c:

```
DEPTH_CONV : conv -> conv
```

DEPTH_CONV c is a conversion

$$t \quad \mapsto \quad \vdash t = t'$$

where t' is obtained from t by replacing every subterm u by v if u reduces to v by c. (Subterms for which $c\ u$ fails are left unchanged.) The definition leaves open the search strategy; in fact, DEPTH_CONV c traverses a term[6] 'bottom up', once, and left-to-right, repeatedly applying c to each subterm until no longer applicable. This helps with the two problems thus far:

```
> DEPTH_CONV BETA_CONV ``(((\x. (\y. (\z. x + y + z))) 1) 2) 3``;                2
val it = ⊢ (λx y z. x + y + z) 1 2 3 = 1 + 2 + 3: thm

> DEPTH_CONV BETA_CONV ``(\z. x + y + z) ((\w. w) 3)``;
val it = ⊢ (λz. x + y + z) ((λw. w) 3) = x + y + 3: thm
```

It may happen, however, that the result of such a conversion still contains subterms that could themselves be reduced at top level. For example:

```
> val t = ``(\f. \x. f x) (\n. n + 1)``;                                         2
val t = “(λf x. f x) (λn. n + 1)”: term

> DEPTH_CONV BETA_CONV t;
val it = ⊢ (λf x. f x) (λn. n + 1) = (λx. (λn. n + 1) x): thm
```

The function TOP_DEPTH_CONV does more searching and reduction than DEPTH_CONV: it replaces every subterm u by v' if u reduces to v by c and v *recursively* reduces to v' by TOP_DEPTH_CONV c.[7]

```
   TOP_DEPTH_CONV : conv -> conv
```

Thus:

```
> TOP_DEPTH_CONV BETA_CONV t;                                                    3
val it = ⊢ (λf x. f x) (λn. n + 1) = (λx. x + 1): thm
```

Finally, the simpler function ONCE_DEPTH_CONV is provided:

```
   ONCE_DEPTH_CONV : conv -> conv
```

ONCE_DEPTH_CONV c t applies c once to the first term (and only the first term) on which it succeeds in a top-down traversal:

```
> ONCE_DEPTH_CONV BETA_CONV t;                                                   4
val it = ⊢ (λf x. f x) (λn. n + 1) = (λx. (λn. n + 1) x): thm

> ONCE_DEPTH_CONV BETA_CONV ``(\x. (\n. n + 1) x)``;
val it = ⊢ (λx. (λn. n + 1) x) = (λx. x + 1): thm
```

[6] That is, it traverses the abstract parse tree of the term.

[7] Readers interested in characterizing the search strategy of TOP_DEPTH_CONV should study the ML definitions near the end of this section.

The equational theorems returned by conversions are not always useful in equational form. To make the results more useful for theorem proving, a conversion can be converted to a rule or a tactic, using the functions CONV_RULE or CONV_TAC, respectively.

```
CONV_RULE : conv -> thm -> thm
CONV_TAC  : conv -> tactic
```

CONV_RULE *c* (|- *t*) returns |- *t′*, where *c t* evaluates to the equation |- *t*=*t′*. CONV_TAC *c* is a tactic that converts the conclusion of a goal using *c*. CONV_RULE is defined by:

```
fun CONV_RULE c th = EQ_MP (c (concl th)) th
```

(The validation of CONV_TAC also uses EQ_MP[8].) For example, the built-in rule BETA_RULE reduces some of the β-redex subterms of a term.

```
BETA_RULE : thm -> thm
```

It is defined by:

```
val BETA_RULE = CONV_RULE(DEPTH_CONV BETA_CONV)
```

The search invoked by BETA_RULE is adequate for some purposes but not others; for example, the first use shown below but not the second:

5

```
> BETA_RULE (ASSUME ``(((\x. (\y. (\z. x + y + z))) 1) 2) 3 < 10``);
val it =  [.] ⊢ 1 + 2 + 3 < 10: thm

> val th = ASSUME ``NEXT = ^t``;
val th =  [.] ⊢ NEXT = (λf x. f x) (λn. n + 1): thm

> BETA_RULE th;
val it =  [.] ⊢ NEXT = (λx. (λn. n + 1) x): thm

> BETA_RULE (BETA_RULE th);
val it =  [.] ⊢ NEXT = (λx. x + 1): thm
```

A more powerful β-reduction rule that used the second search strategy could be defined as shown below (this is not built into HOL).

6

```
> val TOP_BETA_RULE = CONV_RULE (TOP_DEPTH_CONV BETA_CONV);
val TOP_BETA_RULE = fn: thm -> thm

> TOP_BETA_RULE th;
val it =  [.] ⊢ NEXT = (λx. x + 1): thm
```

TOP_DEPTH_CONV is the traversal strategy used by the HOL rewriting tools described in Section 3.5.

[8] For EQ_MP, see 2.3.7.

3.3 Writing Compound Conversions

There are several other conversion operators in HOL, which, together with THENC, ORELSEC and REPEATC are available for building more complex conversions, as well as rules, tactics, and so on. These are described below; several are good illustrations themselves of how functions are built using conversions. The section culminates with the explanation of how DEPTH_CONV, TOP_DEPTH_CONV, and ONCE_DEPTH_CONV are built.

The conversion NO_CONV is an identity for ORELSEC, useful in building functions.

```
NO_CONV : conv
```

NO_CONV t always fails.

The function FIRST_CONV returns $c\ t$ for the first conversion c, in a list of conversions, for which the evaluation of $c\ t$ succeeds.

```
FIRST_CONV : conv list -> conv
```

FIRST_CONV [c_1; ... ;c_n] is equivalent, intuitively, to:

c_1 ORELSEC c_2 ORELSEC ... ORELSEC c_n

It is defined by:

```
fun FIRST_CONV [] tm = NO_CONV tm
  | FIRST_CONV (c :: rst) tm =
      c tm handle HOL_ERR _ => FIRST_CONV rst tm
```

The conversion ALL_CONV is an identity for THENC, useful in building functions.

```
ALL_CONV : conv
```

ALL_CONV t evaluates to |- t=t. It is defined as being identical to REFL.

The function EVERY_CONV applies a list of conversions in sequence.

```
EVERY_CONV : conv list -> conv
```

EVERY_CONV [c_1; ... ;c_n] is equivalent, intuitively, to:

c_1 THENC c_2 THENC ... THENC c_n

It is defined by:

```
fun EVERY_CONV cl t =
    List.foldr (op THENC) ALL_CONV cl t
      handle HOL_ERR _ => raise ERR "EVERY_CONV" ""
```

The operator CHANGED_CONV converts one conversion to another that fails on arguments that it cannot change.

```
CHANGED_CONV : conv -> conv
```

If c t evaluates to |- t=t', then CHANGED_CONV c t also evaluates to |- t=t', unless t and t' are the same (up to α-conversion), in which case it fails.

The operator TRY_CONV maps one conversion to another that always succeeds, by replacing failures with the identity conversion.

```
TRY_CONV : conv
```

If c t evaluates to |- t=t', then TRY_CONV c t also evaluates to |- t=t'. If c t fails, then TRY_CONV c t evaluates to |- t=t. TRY_CONV is implemented by:

```
fun TRY_CONV c = c ORELSEC ALL_CONV
```

It is used in the implementation of TOP_DEPTH_CONV (given later).

There are a number of operators for applying conversions to the immediate subterms of a term. These use the ML functions:

```
MK_COMB : thm * thm -> thm
MK_ABS  : thm -> thm
```

MK_COMB and MK_ABS implement the following derived rules:

$$\frac{\Gamma_1 \ \texttt{|-}\ u_1{=}v_1 \qquad \Gamma_2 \ \texttt{|-}\ u_2{=}v_2}{\Gamma_1 \cup \Gamma_2 \ \texttt{|-}\ u_1\ u_2{=}v_1\ v_2}\ \texttt{MK\_COMB}$$

$$\frac{\Gamma \ \texttt{|-}\ !x.u{=}v}{\Gamma \ \texttt{|-}\ (\backslash x.u) = (\backslash x.v)}\ \texttt{MK\_ABS}$$

The function SUB_CONV applies a conversion to the immediate subterms of a term.

```
SUB_CONV : conv
```

In particular:

- SUB_CONV c "x" = |- x=x;
- SUB_CONV c "u v" = |- u v=u' v', if c u = |- u=u' and c v = |- v=v';
- SUB_CONV c "\x.u" = |- ($\backslash x.u$) = ($\backslash x.u'$), if c u = |- u=u'.

SUB_CONV is implemented in terms of MK_COMB and MK_ABS:

```
fun SUB_CONV c t =
    if is_comb t then
       (let val (rator, rand) = dest_comb t in
        MK_COMB (c rator, c rand) end)
    if is_abs t then
       (let val (bv, body) = dest_abs t;
            val bodyth = c body in
        MK_ABS (GEN bv bodyth) end)
             else (ALL_CONV t)
```

SUB_CONV, too, is used in the definitions of DEPTH_CONV and TOP_DEPTH_CONV.

Three other useful conversion operators, also for applying conversions to the immediate subterms of a term, are as follows:

```
RATOR_CONV : conv -> conv
RAND_CONV  : conv -> conv
ABS_CONV   : conv -> conv
```

RATOR_CONV c converts the operator of an application using c; RAND_CONV c converts the operand of an application; and ABS_CONV c converts the body of an abstraction. Combinations of these are useful for applying conversions to particular subterms. These are implemented by:

```
fun RATOR_CONV c t =
  let val (rator, rand) = dest_comb t
  in
    MK_COMB (c rator, REFL rand)
  end

fun ABS_CONV c t =
  let val (bv, body) = dest_abs t;
      val bodyth = c body
  in
    MK_ABS (GEN bv bodyth)
end
```

The following is an example session illustrating these immediate subterm conversions (recalling that the expression t_1+t_2 actually parses as + t_1 t_2).

```
> val t = ``(\x.x + 1) m + (\x. x + 2) n``;                                  2
val t = “(λx. x + 1) m + (λx. x + 2) n”: term

> RAND_CONV BETA_CONV t;
val it = ⊢ (λx. x + 1) m + (λx. x + 2) n = (λx. x + 1) m + (n + 2): thm

> RATOR_CONV (RAND_CONV BETA_CONV) t;
val it = ⊢ (λx. x + 1) m + (λx. x + 2) n = m + 1 + (λx. x + 2) n: thm
```

Finally, the definitions of DEPTH_CONV and TOP_DEPTH_CONV are given below.

```
fun DEPTH_CONV c t =
 (SUB_CONV (DEPTH_CONV c) THENC (REPEATC c)) t

fun TOP_DEPTH_CONV c t =
 (REPEATC c THENC
  (TRY_CONV
    (CHANGED_CONV (SUB_CONV (TOP_DEPTH_CONV c)) THENC
     TRY_CONV (c THENC TOP_DEPTH_CONV c)))) t

fun ONCE_DEPTH_CONV c t =
 (c ORELSEC (SUB_CONV (ONCE_DEPTH_CONV c))) t
```

Note that the extra argument t is needed to stop these definitions looping (because ML is a call-by-value language). Note also that the actual definition of ONCE_DEPTH_CONV used in the system has been optimised to use failure to avoid rebuilding unchanged subterms.

3.4 Built-in Conversions

Many conversions are predefined in HOL; only those likely to be of general interest are listed here.

3.4.1 Generalized beta-reduction

The conversion:

```
pairLib.PAIRED_BETA_CONV : conv
```

does generalized beta-conversion of tupled lambda abstractions applied to tuples.

Given the term:

```
‘‘(\(x1, ... ,xn).t) (t1, ... ,tn)‘‘
```

PAIRED_BETA_CONV proves that:

```
|- (\(x1, ... ,xn). t[x1,...,xn]) (t1, ... ,tn)  =  t[t1, ... ,tn]
```

The conversion works for arbitrarily nested tuples. For example:

```
> PAIRED_BETA_CONV ``(\((a,b),(c,d)). [a;b;c;d]) ((1,2),(3,4))``;                2
val it = ⊢ (λ((a,b),c,d). [a; b; c; d]) ((1,2),3,4) = [1; 2; 3; 4]: thm
```

3.4.2 Arithmetical conversions

The conversion:

```
reduceLib.ADD_CONV : conv
```

does addition by formal proof. If n and m are numerals then ADD_CONV "n + m" returns the theorem |- n + m = s, where s is the numeral denoting the sum of n and m. For example:

2
```
> ADD_CONV ``1 + 2``;
val it = ⊢ 1 + 2 = 3: thm

> ADD_CONV ``0 + 1000``;
val it = ⊢ 0 + 1000 = 1000: thm

> ADD_CONV ``101 + 102``;
val it = ⊢ 101 + 102 = 203: thm
```

For more general arithmetic, the conversion REDUCE_CONV handles all of the operators in the theory of natural number arithmetic (see Section 5.3.1):

3
```
> REDUCE_CONV ``2 * 3``;
val it = ⊢ 2 * 3 = 6: thm
> REDUCE_CONV ``2 ** 3 + 101 MOD 6``;
val it = ⊢ 2^3 + 101 MOD 6 = 13: thm
```

3.4.3 List processing conversions

There are two useful built-in conversions for lists:

```
LENGTH_CONV : conv
list_EQ_CONV: conv
```

LENGTH_CONV computes the length of a list. A call to:

LENGTH_CONV ‘‘LENGTH[t_1;...;t_n]‘‘

generates the theorem:

|- LENGTH [t_1;...;t_n] = n

The other conversion, list_EQ_CONV, proves or disproves the equality of two lists, given a conversion for deciding the equality of elements. A call to:

list_EQ_CONV *conv* "[u_1;...;u_n] = [v_1;...;v_m]"

returns: `|- ([`u_1`;...;`u_n`] = [`v_1`;...;`v_m`]) = F` if:

(i) `~(`n`=`m`)` or

(ii) *conv* proves `|- (`u_i `= `v_i`) = F` for any $1 \leq i \leq m$.

`|- ([`u_1`;...;`u_n`] = [`v_1`;...;`v_m`]) = T` is returned if:

(i) `(`n`=`m`)` and u_i is syntactically identical to v_i for $1 \leq i \leq m$, or

(ii) `(`n`=`m`)` and *conv* proves `|- (`u_i`=`v_i`)=T` for $1 \leq i \leq n$.

3.4.4 Skolemization

Two conversions are provided for a higher-order version of Skolemization (using existentially quantified function variables rather than first-order Skolem constants).

The conversion

```
X_SKOLEM_CONV : term -> conv
```

takes a variable parameter, f say, and proves:

`|- (!`x_1 ... x_n`. ?`y`. `$t[x_1,\ldots,x_n,y]$ `= (?`f`. !`x_1 ... x_n`. `$t[x_1,\ldots,x_n,f\ x_1 \ldots\ x_n]$

for any input term `!`x_1 ... x_n`. ?`y`.` $t[x_1,\ldots,x_n,y]$. Note that when $n = 0$, this is equivalent to alpha-conversion:

`|- (?`y`. `$t[y]$`) = (?`f`. `$t[f]$`)`

and that the conversion fails if there is already a free variable f of the appropriate type in the input term. For example:

```
> X_SKOLEM_CONV “f:num->’a” “!n:num. ?x:*. x = (f n)”;                    4
Exception- <<HOL message: inventing new type variable names: ’a, ’b>>
HOL_ERR
  {message = "`f` free in the input term", origin_function =
   "X_SKOLEM_CONV", origin_structure = "Conv"} raised
```

will fail. The conversion `SKOLEM_CONV` is like `X_SKOLEM_CONV`, except that it uses a primed variant of the name of the existentially quantified variable as the name of the skolem function it introduces. For example:

```
> SKOLEM_CONV “!x. ?y. P x y”;                                            5
<<HOL message: inventing new type variable names: ’a, ’b>>
val it = ⊢ (∀x. ∃y. P x y) ⟺ ∃y. ∀x. P x (y x): thm
```

3.4.5 Quantifier movement conversions

A complete and systematically-named set of conversions for moving quantifiers inwards and outwards through the logical connectives ~, /\, \/, and ==> is provided. The naming scheme is based on the following atoms:

```
<quant> := FORALL | EXISTS
<conn>  := NOT | AND | OR | IMP
[dir]   := LEFT | RIGHT          (optional)
```

The conversions for moving quantifiers inwards are called:

```
<quant>_<conn>_CONV
```

where the quantifier <*quant*> is to be moved inwards through <*conn*>.

The conversions for moving quantifiers outwards are called:

```
[dir]_<conn>_<quant>_CONV
```

where <*quant*> is to be moved outwards through <*conn*>, and the optional [*dir*] identifies which operand (left or right) contains the quantifier. The complete set is:

```
NOT_FORALL_CONV    |- ~(!x.P) = ?x.~P
NOT_EXISTS_CONV    |- ~(?x.P) = !x.~P
EXISTS_NOT_CONV    |- (?x.~P) = ~!x.P
FORALL_NOT_CONV    |- (!x.~P) = ~?x.P

FORALL_AND_CONV         |- (!x. P /\ Q) = (!x.P) /\ (!x.Q)
AND_FORALL_CONV         |- (!x.P) /\ (!x.Q) = (!x. P /\ Q)
LEFT_AND_FORALL_CONV    |- (!x.P) /\ Q = (!x'. P[x'/x] /\ Q)
RIGHT_AND_FORALL_CONV   |- P /\ (!x.Q) = (!x'. P /\ Q[x'/x])

EXISTS_OR_CONV          |- (?x. P \/ Q) = (?x.P) \/ (?x.Q)
OR_EXISTS_CONV          |- (?x.P) \/ (?x.Q) = (?x. P \/ Q)
LEFT_OR_EXISTS_CONV     |- (?x.P) \/ Q = (?x'. P[x'/x] \/ Q)
RIGHT_OR_EXISTS_CONV    |- P \/ (?x.Q) = (?x'. P \/ Q[x'/x])

FORALL_OR_CONV
  |- (!x.P \/ Q) = P \/ !x.Q          [x not free in P]
  |- (!x.P \/ Q) = (!x.P) \/ Q        [x not free in Q]
  |- (!x.P \/ Q) = (!x.P) \/ (!x.Q)   [x not free in P or Q]

OR_FORALL_CONV
  |- (!x.P) \/ (!x.Q) = (!x.P \/ Q)   [x not free in P or Q]

LEFT_OR_FORALL_CONV    |- (!x.P) \/ Q = !x'. P[x'/x] \/ Q
RIGHT_OR_FORALL_CONV   |- P \/ (!x.Q)  = !x'. P \/ Q[x'/x]
```

```
EXISTS_AND_CONV
  |- (?x.P /\ Q) = P /\ ?x.Q          [x not free in P]
  |- (?x.P /\ Q) = (?x.P) /\ Q        [x not free in Q]
  |- (?x.P /\ Q) = (?x.P) /\ (?x.Q)   [x not free in P or Q]

AND_EXISTS_CONV
  |- (?x.P) /\ (?x.Q) = (?x.P /\ Q)   [x not free in P or Q]

LEFT_AND_EXISTS_CONV    |- (?x.P) /\ Q = ?x'. P[x'/x] /\ Q
RIGHT_AND_EXISTS_CONV   |- P /\ (?x.Q)  = ?x'. P /\ Q[x'/x]

FORALL_IMP_CONV
  |- (!x.P ==> Q) = P ==> !x.Q          [x not free in P]
  |- (!x.P ==> Q) = (?x.P) ==> Q        [x not free in Q]
  |- (!x.P ==> Q) = (?x.P) ==> (!x.Q)   [x not free in P or Q]

LEFT_IMP_FORALL_CONV   |- (!x.P) ==> Q = !x'. P[x/'x] ==> Q
RIGHT_IMP_FORALL_CONV  |- P ==> (!x.Q) = !x'. P ==> Q[x'/x]

EXISTS_IMP_CONV
  |- (?x.P ==> Q) = P ==> ?x.Q          [x not free in P]
  |- (?x.P ==> Q) = (!x.P) ==> Q        [x not free in Q]
  |- (?x.P ==> Q) = (!x.P) ==> (?x.Q)   [x not free in P or Q]

LEFT_IMP_EXISTS_CONV   |- (?x.P) ==> Q = !x'. P[x/'x] ==> Q
RIGHT_IMP_EXISTS_CONV  |- P ==> (?x.Q) = ?x'. P ==> Q[x'/x]
```

3.5 Rewriting Tools

The rewriting tool REWRITE_RULE was introduced in Chapter 2. There are also rewriting conversion like REWRITE_CONV. All of the various rewriting tools provided in HOL are implemented by use of conversions. Certain new tools could also be built in a similar way.

The rewriting primitive in HOL is REWR_CONV:

```
REWR_CONV : thm -> conv
```

REWR_CONV (Γ |- u=v) t evaluates to a theorem Γ |- t=t' if t is an instance (by type and/or variable instantiation) of u and t' is the corresponding instance of v. The first argument to REWR_CONV can be quantified. Below is an illustration.

```
> REWR_CONV ADD1 ``SUC 0``;                                          2
val it = ⊢ SUC 0 = 0 + 1: thm
```

All subterms of t can be rewritten according to an equation th using

```
DEPTH_CONV(REWR_CONV th)
```

as shown below. The function "PRE" is the usual predecessor function.

```
> DEPTH_CONV (REWR_CONV ADD1) ``SUC (SUC 0) = PRE (SUC 2)``;                3
val it = ⊢ SUC (SUC 0) = PRE (SUC 2) ⟺ 0 + 1 + 1 = PRE (2 + 1): thm
```

In itself, this is not a very useful rewriting tool, but a collection of others have been developed for use in HOL. All of the rewriting tools are, in fact, logically derived, and are based on conversions similar to DEPTH_CONV. They have been optimized in various ways, so their implementation is in some cases rather complex and is not given here. The conversions, rules and tactics for rewriting all take a list of theorems to be used as rewrites. The theorems in the list need not be in simple equational form (*e.g.* a conjunction of equations is permissible); but are converted to equational form automatically (and internally). (For example, a conjunction of equations is split into its constituent conjuncts.) There are also a number of standard equations (representing common tautologies) held in the ML variable basic_rewrites, and these are used by some of the rewriting tools. All the built-in rewriting tools are listed below, for reference, beginning with the rules. (All are fully described in *REFERENCE*.)

The prefix 'PURE_' indicates that the built-in equations in basic_rewrites are not used, (*i.e.* only those given explicitly are used). The prefix 'ONCE_' indicates that the tool makes only one rewriting pass through the expression (this is useful to avoid divergence). It is based on ONCE_DEPTH_CONV, while the other tools traverse using TOP_DEPTH_CONV.

The rewriting converions are:

```
REWRITE_CONV                : thm list -> conv
PURE_REWRITE_CONV           : thm list -> conv
ONCE_REWRITE_CONV           : thm list -> conv
PURE_ONCE_REWRITE_CONV      : thm list -> conv
```

The basic rewriting rules are:

```
REWRITE_RULE                      : thm list -> thm -> thm
PURE_REWRITE_RULE                 : thm list -> thm -> thm
ONCE_REWRITE_RULE                 : thm list -> thm -> thm
PURE_ONCE_REWRITE_RULE            : thm list -> thm -> thm
```

The prefix 'ASM_' indicates that the rule rewrites using the assumptions of the theorem as rewrites.

```
ASM_REWRITE_RULE                  : thm list -> thm -> thm
PURE_ASM_REWRITE_RULE             : thm list -> thm -> thm
ONCE_ASM_REWRITE_RULE             : thm list -> thm -> thm
PURE_ONCE_ASM_REWRITE_RULE        : thm list -> thm -> thm
```

The prefix 'FILTER_' indicates that the rule only rewrites with those assumptions of the theorem satisfying the predicate supplied.

```
FILTER_ASM_REWRITE_RULE           : (term -> bool) -> thm list -> thm -> thm
FILTER_PURE_ASM_REWRITE_RULE      : (term -> bool) -> thm list -> thm -> thm
FILTER_ONCE_ASM_REWRITE_RULE      : (term -> bool) -> thm list -> thm -> thm
FILTER_PURE_ONCE_ASM_REWRITE_RULE : (term -> bool) -> thm list -> thm -> thm
```

Tactics are introduced in Chapter 4, but are listed here for reference. The tactics corresponding to the above rules are the following:

```
REWRITE_TAC                       : thm list -> tactic
PURE_REWRITE_TAC                  : thm list -> tactic
ONCE_REWRITE_TAC                  : thm list -> tactic
PURE_ONCE_REWRITE_TAC             : thm list -> tactic
```

The prefix 'ASM_' indicates that the tactic rewrites using the assumptions of the goal as rewrites.

```
ASM_REWRITE_TAC                   : thm list -> tactic
PURE_ASM_REWRITE_TAC              : thm list -> tactic
ONCE_ASM_REWRITE_TAC              : thm list -> tactic
PURE_ONCE_ASM_REWRITE_TAC         : thm list -> tactic
```

The prefix 'FILTER_' indicates that the tactic only rewrites with those assumptions of the goal satisfying the predicate supplied.

```
FILTER_ASM_REWRITE_TAC            : (term -> bool) -> thm list -> tactic
FILTER_PURE_ASM_REWRITE_TAC       : (term -> bool) -> thm list -> tactic
FILTER_ONCE_ASM_REWRITE_TAC       : (term -> bool) -> thm list -> tactic
FILTER_PURE_ONCE_ASM_REWRITE_TAC  : (term -> bool) -> thm list -> tactic
```

Chapter 4

Goal Directed Proof: Tactics and Tacticals

There are three primary devices that together make theorem proving practical in HOL. All three originate with Milner for Edinburgh LCF. The first is the theory as a record of (among other things) facts already proved and thence available as lemmas without having to be re-proved. The second, the subject of Chapter 2, is the derived rule of inference as a meta-language procedure that implements a broad pattern of inference, but that also, at each application, generates every primitive step of the proof. The third device is the tactic as a means of organizing the construction of proofs; and the use of tacticals for composing tactics.

Even with recourse to derived inference rules, it is still surprisingly awkward to work forward, to find a chain of theorems that culminates in a desired theorem. This is in part because chains have no structure, while 'proof efforts' do. For instance, if within one sequence, two chains of steps are to be combined in the end by conjunction, then one chain must follow or be interspersed with the other in the overall sequence. It can also be difficult to direct the proof toward its object when starting from only hypotheses (if any), lemmas (if any), axioms, and theorems following from no hypotheses (*e.g.*, by `ASSUME` or `REFL`). Likewise, it can be equally difficult to reconstruct the plan of the proof effort after the fact, from the linear sequence of theorems; the sequence is unhelpful as documentation.

The idea of goal directed proof is a simple one, well known in artificial intelligence: to organize the search as a tree, and to reverse the process and *begin* with the objective. The goal is then decomposed, successively if necessary, into what one hopes are more tractable subgoals, each decomposition accompanied by a plan for translating the solution of subgoals into a solution of the goal. The choice of decomposition is an explicit way of expressing a proof 'strategy'.

Thus, for example, instead of the linear sequencing of two branches of the proof of the conjunction, each branch starting from scratch, the proof task is organized as a tree search, starting with a conjunctive goal and decomposing it into the two conjunct subgoals (undertaken in optional order), with the intention of conjoining the two solutions when and if found. The proof itself, as a sequence of steps, is the same however it is found; the difference is in the search, and in the preservation, if required, of the

structured proof plan.

The representation of this idea in LCF was Milner's inspiration; the idea is similarly central to theorem proving in HOL. Although subgoaling theorem provers had already been built at the time, Milner's particular contribution was in formalizing the method for translating subgoals solutions to solutions of goals.

4.1 Tactics, Goals and Validations

A *tactic* is an ML function that when applied to a *goal* reduces it to (i) a list[1] of (sub)goals, along with (ii) a *validation* function mapping a list of theorems to a theorem. The idea is that this function justifies the decomposition of the goal, and so it is also known as a *justification*. A goal is an ML value whose type is isomorphic to, but distinct from, the ML abstract type `thm` of theorems. That is, a goal is a list of terms (*assumptions*) paired with a term. These two components correspond, respectively, to the list of hypotheses and the conclusion of a theorem. The list of assumptions is a working record of facts that may be used in decomposing the goal.

The relation of theorems to goals is achievement: a theorem achieves a goal if the conclusion of the theorem is equal to the term part of the goal (up to α-conversion), and if each hypothesis of the theorem is equal (up to α-conversion, again) to some assumption of the goal. This definition assures that the theorem purporting to satisfy a goal does not depend on assumptions beyond the working assumptions of the goal.

A tactic is said to *solve* a goal if it reduces the goal to the empty set of subgoals. This depends, obviously, on there being at least one tactic that maps a goal to the empty subgoal list. The simplest tactic that does this is one that can recognize when a goal is achieved by an axiom or an existing theorem; in HOL, the function `ACCEPT_TAC` does this. `ACCEPT_TAC` takes a theorem *th* and produces a tactic that maps a value of type `thm` to the empty list of subgoals. It justifies this 'decomposition' by a validation function that maps the empty list of theorems to the theorem *th*. The use of this technical device, or other such tactics, ends the decomposition of subgoals, and allows the proof to be built up.

Unlike theorems, goals need not be defined as an abstract type; they are transparent and can be constructed freely. Thus, an ML type abbreviation is introduced for goals. The operations on goals are therefore just the ordinary pair selectors and constructor. Likewise, type abbreviations are introduced for validations and tactics. Conceptually, the following abbreviations are made in HOL:

```
goal       = term list * term
tactic     = goal -> goal list * validation
validation = thm list -> thm
```

[1] The ordering is necessary for selecting a tree search strategy.

It does not follow, of course, from the type `tactic` that a particular tactic is well-behaved. For example, suppose that $T\ g$ `=` `(`$[g_1,\ldots,g_n]$`,`v`)`, and that the subgoals g_1 , … , g_n have been solved. That means that some theorems th_1 , … , th_n have been proved such that each th_i $(1 \leq i \leq n)$ achieves the goal g_i. The validation v is intended to be a function that when applied to the list $[th_1,\ldots,th_n]$, succeeds in returning a theorem, th, achieving the original goal g; but, of course, it might sometimes not succeed. If v succeeds for every list of achieving theorems, then the tactic T is said to be *valid*. This does not guarantee, however, that the subgoals are solvable in the first place. If, in addition to being valid, a tactic always produces solvable subgoals from a solvable goal, it is called *strongly valid*.

Tactics can be perfectly useful without being strongly valid, or without even being valid; in fact, some of the most basic theorem proving strategies, expressed as tactics, are invalid or not strongly valid.[2] An invalid tactic cannot result in the proof of false theorems; theorems in HOL are always the result of performing a proof in the basic logic, whether the proof is found by goal directed search or forward search.[3] However, an invalid tactic may produce an unintended theorem—one that does not achieve the original goal. The typical case is when a theorem purporting to achieve a goal actually depends on hypotheses that extend beyond the assumptions of the goal. The inconvenience to the HOL user in this case is that the problem may be not immediately obvious; the default print format of theorems has hypotheses abbreviated as dots. Invalidity may also be the result of the failure of the proof function, in the ML sense of failure, when applied to a list of theorems (if, for example, the function were defined incorrectly); but again, no false theorems can result. Likewise, a tactic that is not strongly valid cannot result in a false theorem; the worst outcome of applying such a tactic is the production of unsolvable subgoals.

Tactics are specified using the following notation:

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

For example, the tactic for decomposing conjunctions into two conjunct subgoals is called `conj_tac`.[4] It is described by:

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

[2]The subgoal package, discussed in Section 7.2, prevents the use of invalid tactics when they are liable to result in unexpected theorem results, but the HOL system used directly allows it.

[3]'Invalid' is perhaps a misleading term, since there is nothing logically amiss in the use of invalid tactics or the theorems produced thereby; but the term has stuck over time.

[4]The tactic is also available under the name `CONJ_TAC`.

This indicates that `conj_tac` reduces a goal of the form $(\Gamma, t_1 /\backslash t_2)$ to subgoals (Γ, t_1) and (Γ, t_2). The fact that the assumptions of the original goal are propagated unchanged to the two subgoals is indicated by the absence of assumptions in the notation. The notation gives no indication of the proof function.

Another example is `numLib.INDUCT_TAC`, a low-level tactic for performing mathematical induction on the natural numbers:

$$\frac{\forall n.\ t[n]}{t[0] \qquad \{t[n]\}\ t[\mathtt{SUC}\ n]}$$

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

Tactics fail (in the ML sense) if they are applied to inappropriate goals. For example, `conj_tac` will fail if it is applied to a goal whose conclusion is not a conjunction. Some tactics never fail; for example `all_tac`

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

is the identity tactic; it reduces a goal (Γ, t) to the single subgoal (Γ, t)—*i.e.*, it has no effect. The `all_tac` tactic is useful for writing compound tactics, as discussed later (see Section 4.3).

In just the way that the derived rule `REWRITE_RULE` can be used in forward proof (Section 2.2), the corresponding tactic `rewrite_tac` can be used in goal-directed proof. Given a goal and a list of equational theorems, `rewrite_tac` transforms the term component of the goal by applying the equations as left-to-right rewrites, recursively and to all depths, until no more changes can be made. Unless not required, the function includes as rewrites the same standard set of pre-proved tautologies that `REWRITE_RULE` uses. By use of the tautologies, some subgoals can be solved internally by rewriting, and in that case, an empty list of subgoals is returned. The transformation of the goal is justified in each case by the appropriate chain of inferences. Rewriting is a simple implementation of an extremely powerful idea. The more sophisticated simplification functions (*e.g.*, `simp`; see Section 7.5) often do a very large share of the work in goal directed proof searches.

A simple example from list theory (Section 5.4.1) illustrates the use of tactics. A conjunctive goal is declared, and `conj_tac` applied to it:

```
> val gl0 = ([]:term list,``(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])``);
val gl0 = ([], “HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]”): term list * term

> val (gl1,p1) = conj_tac gl0;
val gl1 = [([], “HD [1; 2; 3] = 1”), ([], “TL [1; 2; 3] = [2; 3]”)]:
   goal list
val p1 = fn: validation
```
1

The subgoals are each rewritten, using the definitions of HD and TL:

```
> open listTheory;  ...output elided...

> HD;
val it = ⊢ ∀h t. HD (h::t) = h: thm

> TL;
val it = ⊢ ∀h t. TL (h::t) = t: thm

> val (gl1_1,p1_1) = rewrite_tac[HD,TL](hd gl1);
val gl1_1 = []: goal list
val p1_1 = fn: validation

> val (gl1_2,p1_2) = rewrite_tac[HD,TL](hd(tl gl1));
val gl1_2 = []: goal list
val p1_2 = fn: validation
```
2

Both of the two subgoals are now solved, so the decomposition is complete and the proof can be built up in stages. First the theorems achieving the subgoals are proved, then from those, the theorem achieving the original goal:

```
> val th1 = p1_1[];
val th1 = ⊢ HD [1; 2; 3] = 1: thm

> val th2 = p1_2[];
val th2 = ⊢ TL [1; 2; 3] = [2; 3]: thm

> p1[th1,th2];
val it = ⊢ HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]: thm
```
3

Although only the theorems achieving the subgoals are ‘seen’ here, the proof functions of the three tactic applications together perform the entire chain of inferences leading to the theorem achieving the goal. The same proof could be constructed by forward search, starting from the definitions of HD and TL, but not nearly as easily.

The HOL system provides a very large collection of pre-defined tactics that includes conj_tac, INDUCT_TAC, all_tac and rewrite_tac. The pre-defined tactics are adequate for many applications. In addition, there are two means of defining new tactics. Since a tactic is an ML function, the user can define a new tactic directly in ML. Definitions of this

sort use ML functions to construct the term part of the subgoals from the term part of the original goal (if any transformation is required); and they specify the justification, which expects a list of theorems achieving the subgoals and returns the theorem achieving (one hopes) the goal. The proof of the theorem is encoded in the definition of the justification function; that is, the means for deriving the desired theorem from the theorems given. This typically involves references to axioms and primitive and defined inference rules, and is usually the more difficult part of the project.

A simple example of a tactic written in ML is afforded by `conj_tac`, whose definition in HOL is as follows:

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

This shows how the subgoals are constructed, and how the proof function is specified in terms of the derived rule `CONJ` (Section 2.3.23).

The second method is to compose existing tactics by the use of ML functions called *tacticals*. The tacticals provided in HOL are listed in Section 4.3. For example, two existing tactics can be sequenced by use of the tactical >>[5]: if T_1 and T_2 are tactics, then the ML expression T_1 >> T_2 evaluates to a tactic that first applies T_1 to a goal and then applies T_2 to each subgoal produced by T_1. The tactical >> is an infixed ML function. Complex and powerful tactics can be constructed in this way; and new tacticals can also be defined, although this is unusual.

The example from earlier is continued, to illustrate the use of the tactical >>:

```
> val (gl2,p2) = (conj_tac >> rewrite_tac [HD, TL]) gl0;                    4
val gl2 = []: goal list
val p2 = fn: thm list -> thm

> p2 [];
val it = ⊢ HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]: thm
```

The single tactic `conj_tac >> rewrite_tac[HD;TL]` solves the goal in one single application. The chain of inference computed, however, is exactly the same as in the interactive proof; only the search is different.

In general, the second method is both easier and more reliable. It is easier because it does not involve writing ML procedures (usually rather complicated procedures); and more reliable because the composed tactics are valid when the constituent tactics are valid, as a consequence of the way the tacticals are defined. Tactics written directly in ML may fail in a variety of ways, and although, as usual, they cannot cause false theorems

[5]The >> function can also be written as `THEN`.

to appear, the failures can be difficult to understand and trace. On the other hand, there are some proof strategies that cannot be implemented as compositions of existing tactics, and these have to be implemented directly in ML. Certain sorts of inductions are an example of this; as well as tactics to support some personal styles of proof.

Either sort of tactic can be difficult to apply by hand, as shown in the examples above. There can be a lot of book-keeping required to support such an activity. For this reason, most interactive theorem-proving uses the subgoal or goalstack package described in Section 7.2.

4.2 Some Tactics Built into HOL

This section contains a selection of the more commonly used tactics in the HOL system. (See *REFERENCE* for the complete list, with fuller explanations.)

It should be recalled that the ML type `thm_tactic` abbreviates `thm->tactic`, and the type `conv` abbreviates `term->thm`.

4.2.1 Specialization

```
gen_tac : tactic
```

Summary: Specializes a universally quantified theorem to an arbitrary value. Also available under the name `GEN_TAC`.

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

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

Use: Solving universally quantified goals. `gen_tac` is often the first step of a goal directed proof. `strip_tac` (see below) applies `gen_tac` to universally quantified goals.

4.2.2 Conjunction

```
conj_tac : tactic
```

Summary: Splits a goal $t_1 \wedge t_2$ into two subgoals, t_1 and t_2.

$$\frac{t_1 \ /\backslash \ t_2}{t_1 \quad t_2}$$

Use: Solving conjunctive goals. `conj_tac` is also invoked by `strip_tac` (see below).

4.2.3 Combined simple decompositions

```
strip_tac : tactic
```

Summary: Breaks a goal apart. The tactic `strip_tac` removes one outer connective from the goal, using `conj_tac`, `gen_tac`, and other tactics. If the goal has the form $t_1 \wedge \cdots \wedge t_n$ `==>` t then `strip_tac` makes each t_i into a separate assumption.

Use: Useful for splitting a goal up into manageable pieces. Often the best thing to do first is `rpt strip_tac`, where `rpt` is the tactical that repeatedly applies a tactic until it fails (see Section 4.3.7).

4.2.4 Case analysis

```
Cases_on : term quotation -> tactic
```

Summary: `Cases_on` q, where q is a quotation denoting a term of an algebraic type (*e.g.*, booleans, lists, natural numbers, types defined by the user with `Datatype`), does case analysis on that term. Let the relevant type have m constructors C_1 to C_m, where C_i has n_i arguments. If the term is a variable, then the variable is substituted out of the goal entirely:

$$\frac{t}{t[q := \mathsf{C}_1\, a_{11} \cdots a_{1n_1}] \quad \cdots \quad t[q := \mathsf{C}_m\, a_{m1} \cdots a_{mn_m}]}$$

If the term denoted by q is not simply a variable (*e.g.*, `Cases_on‘n + 1‘`), then fresh assumptions are introduced:

$$\frac{t}{\{q := \mathsf{C}_1\, a_{11} \cdots a_{1n_1}\}t \quad \cdots \quad \{q := \mathsf{C}_m\, a_{m1} \cdots a_{mn_m}\}t}$$

Use: Case analysis.

4.2.5 Rewriting

```
rewrite_tac : thm list -> tactic
```

Summary: `rewrite_tac`$[th_1,\ldots,th_n]$ transforms the term part of a goal by rewriting it with the given theorems th_1, ... , th_n, and the set of pre-proved standard tautologies. Also written `REWRITE_TAC`.

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

where t' is obtained from t as described.

Use: Advancing goals by using definitions and previously proved theorems.

Other rewriting tactics (based on `rewrite_tac`) are:

1. `asm_rewrite_tac` adds the assumptions of the goal to the list of theorems used for rewriting. Also written `ASM_REWRITE_TAC`.
2. `PURE_ASM_REWRITE_TAC` is like `asm_rewrite_tac`, but it doesn't use any built-in rewrites.
3. `PURE_REWRITE_TAC` uses neither the assumptions nor the built-in rewrites.
4. `FILTER_ASM_REWRITE_TAC` p $[th_1,\ldots,th_n]$ simplifies the goal by rewriting it with the explicitly given theorems th_1 , ... , th_n , together with those assumptions of the goal which satisfy the predicate p and also the standard rewrites.

See also the more powerful simplification tactics described in Section 7.5.

4.2.6 Resolution by Modus Ponens

```
imp_res_tac : thm -> tactic
```

Summary: `imp_res_tac` th does a limited amount of automated theorem proving in the form of forward inference; it 'resolves' the theorem th with the assumptions of the goal and adds any new results to the assumptions. The specification for `imp_res_tac` is:

$$\frac{\{t_1,\ldots,t_m\}t}{\{t_1,\ldots,t_m,u_1,\ldots,u_n\}t}$$

where $u_1, \ldots, u_n$ are derived by 'resolving' the theorem th with the existing assumptions $t_1, \ldots, t_m$. Resolution in HOL is not classical resolution, but just Modus Ponens with one-way pattern matching (not unification) and term and type instantiation. The general case is where th is of the canonical form

|- !$x_1 \ldots x_p . v_1$ ==> v_2 ==> ... ==> v_q ==> v

`imp_res_tac` th then tries to specialize $x_1, \ldots, x_p$ in succession so that $v_1, \ldots, v_q$ match members of $\{t_1, \ldots, t_m\}$. Each time a match is found for some antecedent v_i, for i successively equal to 1, 2, ..., q, a term and type instantiation is made and the rule of Modus Ponens is applied. If all the antecedents v_i (for $1 \leq i \leq q$) can be dismissed in this way, then the appropriate instance of v is added to the assumptions. Otherwise, if only some initial sequence $v_1, \ldots, v_k$ (for some k where $1 < k < q$) of the assumptions can be dismissed, then the remaining implication:

|- v_{k+1} ==> ... ==> v_q ==> v

is added to the assumptions.

For a more detailed description of resolution and `imp_res_tac`, see *REFERENCE*. (See also the Cambridge LCF Manual [12].)

Use: Deriving new results from a previously proved implicative theorem, in combination with the current assumptions, so that subsequent tactics can use these new results.

4.2.7 Identity

```
all_tac : tactic
```

Summary: The identity tactic for the tactical >> (see the example in Section 4.1, and Section 4.3.4). Useful for writing tactics.

Use:
1. Writing tacticals (see description of `rpt` in Section 4.3).
2. With `THENL` (see Section 4.3.5); for example, if tactic T produces two subgoals and T_1 is to be applied to the first while nothing is to be done to the second, then T `THENL` [T_1, `all_tac`] is the tactic required.

4.2.8 Splitting logical equivalences

```
eq_tac : tactic
```

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

$$\frac{u \iff v}{u \Rightarrow v \qquad v \Rightarrow u}$$

Use: Proving logical equivalences, *i.e.* goals of the form "$u \iff v$" where u and v are boolean terms. Note that the same end can often be achieved by rewriting with the theorem EQ_IMP_THM, which states

$$\vdash (u \iff v) \iff (u \Rightarrow v) \wedge (v \Rightarrow u)$$

4.2.9 Solving existential goals

```
EXISTS_TAC : term -> tactic
```

Summary: EXISTS_TAC u reduces an existential goal $\exists x.\ t[x]$ to the subgoal $t[u]$. (In ASCII form, the existential is printed as a question mark.)

$$\frac{?x\,.\,t[x]}{t[u]}$$

Use: Proving existential goals.

4.3 Tacticals

A *tactical* is not represented by a single ML type, but is in general an ML function that returns a tactic (or tactics) as result. Tacticals may take parameters, and this is reflected in the variety of ML types that the built-in tacticals have. Tacticals are used for building compound tactics. Some important tacticals in the HOL system are listed below. For a complete list of the tacticals in HOL see *REFERENCE*.

4.3.1 Alternation

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

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

```
(T1 ORELSE T2) g = T1 g handle HOL_ERR _ => T2 g
```

4.3.2 First success

```
FIRST : tactic list -> tactic
```

The tactical FIRST applies the first tactic, in a list of tactics, that succeeds.

FIRST $[T_1;T_2;\ldots;T_n]$ = T_1 ORELSE T_2 ORELSE ... ORELSE T_n

4.3.3 Change detection

```
CHANGED_TAC : tactic -> tactic
```

CHANGED_TAC T g fails if the subgoals produced by T are just $[g]$; otherwise it is equivalent to T g. It is defined by the following, where

```
set_eq: 'a list -> 'a list -> bool
```

tests whether two lists denote the same set (*i.e.* contain the same elements).

```
fun CHANGED_TAC tac g =
 let val (gl,p) = tac g
 in
   if set_eq gl [g] then raise ERR "CHANGED_TAC" "no change"
   else (gl,p)
 end
```

4.3.4 Sequencing

```
THEN : tactic -> tactic -> tactic
>>   : tactic -> tactic -> tactic
```

The tactical >> is an ML infix. Its alias THEN is also an infix. If T_1 and T_2 are tactics, then the ML expression T_1 >> T_2 evaluates to a tactic which first applies T_1 and then applies T_2 to each subgoal produced by T_1.

Both >> and THEN associate to the left, which can lead to some counter-intuitive behaviours when they combine with other sequencing operators. However, chains of tactics connected with >> behave as one might expect. In particular, if T_1 produces n sub-goals, and T_2 produces varying numbers of sub-goals when applied to each of those, then the expression T_1 >> T_2 >> T_3 will apply T_3 to *all* of the leaf sub-goals generated by the sequencing of T_1 and T_2.

4.3.5 Selective sequencing

```
THENL : tactic -> tactic list -> tactic
>|    : tactic -> tactic list -> tactic
```

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

4.3.6 Successive application

```
EVERY : tactic list -> tactic
```

The tactical `EVERY` applies a list of tactics one after the other.

`EVERY` $[T_1,T_2,\ldots,T_n]$ = T_1 `>>` T_2 `>>` ... `>>` T_n

4.3.7 Repetition

```
REPEAT : tactic -> tactic
rpt    : tactic -> tactic
```

If T is a tactic then `rpt` T is a tactic that repeatedly applies T until it fails. It is defined in ML by:

```
fun REPEAT T g = ((T THEN REPEAT T) ORELSE ALL_TAC) g
```

(The extra argument `g` is needed because ML does not use lazy evaluation.)

4.4 Tactics for Manipulating Assumptions

There are in general two kinds of tactics in HOL: those that transform the conclusion of a goal without affecting the assumptions, and those that do (also or only) affect the assumptions. The various tactics that rewrite are typical of the first class; those that do 'resolution' belong to the second. Often, many of the steps of a proof in HOL are carried out 'behind the scenes' on the assumptions, by tactics of the second sort. A tactic that in some way changes the assumptions must also have a justification that 'knows how' to restore the corresponding hypotheses of the theorem achieving the subgoal. All of this is explicit, and can be examined by a user moving about the subgoal-proof tree.[6] Using these tactics in the most straightforward way, the assumptions at any point in a

[6]The current subgoal package makes this difficult, but the point still holds.

goal-directed proof, *i.e.* at any node in the subgoal tree, form an unordered record of every assumption made, but not yet dismissed, up to that point.

In practice, the straightforward use of assumption-changing tactics, with the tools currently provided in HOL, presents at least two difficulties. The first is that assumption sets can grow to an unwieldy size, the number and/or length of terms making them difficult to read. In addition, forward-search tactics such as resolution often add at least some assumptions that are never subsequently used, and these have to be carried along with the useful assumptions; the straightforward method provides no ready way of intercepting their arrival. Likewise, there is no straightforward way of discarding assumptions after they have been used and are merely adding to the clutter. Although perhaps against the straightforward spirit, this is a perfectly valid strategy, and requires no more than a way of denoting the specific assumptions to be discarded. That, however, raises the more general problem of denoting assumptions in the first place. Assumptions are also denoted so that they can be manipulated: given as parameters, combined to draw inferences, etc. The only straightforward way to denote them in the existing system is to supply their quoted text. Though adequate, this method may result in bulky ML expressions; and it may take some effort to present the text correctly (with necessary type information, *etc.*).

As always in HOL, there are quite a few ways around the various difficulties. One approach, of course, is the one intended in the original design of Edinburgh LCF, and advocates the rationale for providing a full programming language, ML, rather than a simple proof command set: that is for the user to implement new tactics in ML. For example, resolution tactics can be adapted by the user to add new assumptions more selectively; and case analysis tactics to make direct replacements without adding case assumptions. This, again, is adequate, but can involve the user in extensive amounts of programming, and in debugging exercises for which there is no system support.

Short of implementing new tactics, two other standard approaches are reflected in the current system. Both were originally developed for Cambridge LCF [11, 12]; both reflect fresh views of the assumptions; and both rely on tacticals that transform tactics. The two approaches are partly but not completely complementary.

The first approach, described in this section, implicitly regards the assumption set, already represented as a list, as a stack, with a *pop* operation, so that the assumption at the top of the stack can be (i) discarded and (ii) denoted without explicit quotation. (The corresponding *push* adds new assumptions at the head of the list.) The stack can be generalized to an array to allow for access to arbitrary assumptions.

The other approach, described in Section 4.4.2, gives a way of intercepting and manipulating results without them necessarily being added as assumptions in the first place. The two approaches can be combined in HOL interactions.

4.4.1 Theorem continuations with popping

The first proof style, that of popping assumptions from the assumption 'stack', is illustrated using its main tool: the tactical POP_ASSUM.

```
POP_ASSUM : (thm -> tactic) -> tactic
pop_assum : (thm -> tactic) -> tactic
```

Given a function f : thm -> tactic, the tactic pop_assum f applies f to the (assumed) first assumption of a goal (*i.e.* to the top element of the assumption stack) and then applies the tactic created thereby to the original goal minus its top assumption:

pop_assum f ([t_1;...;t_n],t) = f (ASSUME t_1) ([t_2;...;t_n],t)

ML functions such as f, with type thm -> tactic, abbreviated to thm_tactic, are called theorem continuations, suggesting the fact that they take theorems and then continue the proof.[7] The use of pop_assum can be illustrated by applying it to a particular tactic, namely DISCH_TAC.

```
DISCH_TAC : tactic
```

On a goal whose conclusion is an implication $u \Rightarrow v$, DISCH_TAC reflects the natural strategy of attempting to prove v under the assumption u, the discharged antecedent. For example, suppose it were required to prove that $(n = 0) \Rightarrow (n \times n = n)$:

```
> g `(n = 0) ==> (n * n = n)`;                                    1
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        n = 0 ⇒ n * n = n

> e DISCH_TAC;
OK..
1 subgoal:
val it =

    0.  n = 0
   ------------------------------------
        n * n = n
```

Application of DISCH_TAC to the goal produces one subgoal, as shown, with the added assumption. To engage the assumption as a simple substitution, the tactic SUBST1_TAC is useful (see *REFERENCE* for details).

[7] There is a superficial analogy with continuations in denotational semantics.

```
   SUBST1_TAC : thm_tactic
```

SUBST1_TAC expects a theorem with an equational conclusion, and substitutes accordingly, into the conclusion of the goal. At this point in the session, the tactical POP_ASSUM is applied to SUBST1_TAC to form a new tactic. The new tactic is applied to the current subgoal.

2
```
> p();
val it =

    0.  n = 0
   ------------------------------------
        n * n = n

> e(pop_assum SUBST1_TAC);;
OK..
1 subgoal:
val it =

   0 * 0 = 0
```

The result, as shown, is that the assumption is used as a substitution rule and then discarded. The one subgoal therefore has no assumptions on its stack. The two tactics used thus far could be combined into one using the tactical >>:

1
```
> g `(n = 0) ==> (n * n = n)`;
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        n = 0 ⇒ n * n = n

> e(DISCH_TAC >> pop_assum SUBST1_TAC);;
OK..
1 subgoal:
val it =

   0 * 0 = 0
```

The goal can now be solved by simplification:of arithmetic:

2
```
> e(simp[]);
<<HOL message: Initialising SRW simpset ... done>>
OK..

Goal proved.
⊢ 0 * 0 = 0
val it =
   Initial goal proved.
   ⊢ n = 0 ⇒ n * n = n: proof
```

A single tactic can, of course, be written to solve the goal:[8]

```
> restart();  ...output elided...                                    1

> e(DISCH_TAC >> pop_assum SUBST1_TAC >> simp[]);;
OK..
val it =
   Initial goal proved.
   ⊢ n = 0 ⇒ n * n = n: proof
```

This example illustrates how the tactical `pop_assum` provides access to the top of the assumption 'stack' (a capability that is useful, obviously, only when the most recently pushed assumption is the very one required). To accomplish this access in the straightforward way would require some more awkward construct, with explicit assumptions:

```
> g `(n = 0) ==> (n * n = n)`;                                       1
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        n = 0 ⇒ n * n = n

> e DISCH_TAC;
OK..
1 subgoal:
val it =

    0.  n = 0
   ------------------------------------
        n * n = n

> e(SUBST1_TAC(ASSUME ``n = 0``));
OK..
1 subgoal:
val it =

    0.  n = 0
   ------------------------------------
        0 * 0 = 0
```

In contrast to the above, the popping example also illustrates the convenient disappearance of an assumption no longer required, by removing it from the stack at the moment when it is accessed and used. This is valid because any theorem that achieves the subgoal will still achieve the original goal. Discarding assumptions is a separate issue from accessing them; there could, if one liked, be another tactical that produced a similar tactic on a theorem continuation to `pop_assum` but which did not pop the stack.

[8] Indeed, the tactic `simp[]` solves the entire goal from the outset.

Finally, `pop_assum` f induces case splits where f does. To prove $(n = 0 \vee n = 1) \Rightarrow (n \times n = n)$, the function `DISJ_CASES_TAC` can be used. The tactic

```
DISJ_CASES_TAC |- p \/ q
```

splits a goal into two subgoals that have p and q, respectively, as new assumptions.

Simply using `DISCH_TAC` does not cause the disjunction to split when it becomes an assumption.

1

```
> g `((n = 0) \/ (n = 1)) ==> (n * n = n)`;
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        n = 0 ∨ n = 1 ⇒ n * n = n

> e DISCH_TAC;
OK..
1 subgoal:
val it =

    0.  n = 0 ∨ n = 1
   ------------------------------------
        n * n = n
```

So, we can combine `pop_assum` and `DISJ_CASES_TAC` to apply the latter to the theorem corresponding to the disjunctive assumption:

2

```
> restart();  ...output elided...

> e(DISCH_TAC >> pop_assum DISJ_CASES_TAC);
OK..
2 subgoals:
val it =

    0.  n = 1
   ------------------------------------
        n * n = n

    0.  n = 0
   ------------------------------------
        n * n = n
```

Indeed, we can then combine this with our earlier use of `SUBST1_TAC` to have both branches make progress at once:

```
> restart();  ...output elided...                                    3

> e(DISCH_TAC >> pop_assum DISJ_CASES_TAC >> pop_assum SUBST1_TAC);
OK..
2 subgoals:
val it =

   1 * 1 = 1

   0 * 0 = 0
```

As noted earlier, `pop_assum` is useful when an assumption is required that is still at the top of the stack, as in the examples. However, it is often necessary to access assumptions made at arbitrary previous times, in order to give them as parameters, combine them, etc.

Two other useful tacticals that can be used to manipulate the assumption list are `first_assum` and `qpat_assum`. The first has an easy characterisation:

```
first_assum f ([t1, ... ,tn], t) =
 (f(ASSUME t1) ORELSE ... ORELSE f(ASSUME tn)) ([t1, ... ,tn], t)
```

The `first_x_assum` tactical is similar to `first_assum`, but in addition to assuming one of the goal's assumptions as a theorem and passing this to the function f, the goal-state that $f\ (t_i \vdash t_i)$ acts upon has had t_i removed from the assumption list. It is a "popping" version of `first_assum`.

The `qpat_assum` tactical takes a pattern-quotation as its first argument. The second `thm_tactic` parameter is then passed the first assumption that matches this pattern.

Thus:

```
qpat_assum pat f ([t1, ... ,tp, ... ,tn], t) = f (ASSUME tp) ([t1, ..., tn], t)
```

where t_p is the first assumption that matches the pattern `pat`.

Just as with `first_assum`, there is a "popping" version called `qpat_x_assum` that removes the matching assumption from the list. Thus:

```
qpat_x_assum pat f ([t1, ... ,tp, ... ,tn], t) =
  f (ASSUME tp) ([t1, ... ,tp-1,tp+1, ... ,tn], t)
```

where t_p is the first assumption that matches the pattern `pat`.

While `first_assum` (and `first_x_assum`) try to apply their theorem-tactic to every assumption, eventually using the first that succeeds, `qpat_assum` (and `qpat_x_assum`) only apply their theorem-tactic to one assumption (the first that matches).

4.4.2 Theorem continuations without popping

The idea of the second approach is suggested by the way the array-style tacticals supply a list of theorems (the assumed assumptions) to a function. These tacticals use the function to infer new results from the list of theorems, and then to do something with the results. In some cases, *e.g.* the last example, the assumptions need never have been made in the first place, which suggests a different use of tacticals. The original example for `pop_assum` illustrates this: namely, to show that $(n = 0) \Rightarrow (n \times n = n)$. Here, instead of discharging the antecedent by applying `DISCH_TAC` to the goal, which adds the antecedent as an assumption and returns the consequent as the conclusion, and *then* supplying the (assumed) added assumption to the theorem continuation `SUBST1_TAC` and discarding it at the same time, a tactical called `disch_then` is applied to `SUBST1_TAC` directly. `disch_then` transforms `SUBST1_TAC` into a new tactic: one that applies `SUBST1_TAC` directly to the (assumed) antecedent, and the resulting tactic to a subgoal with no new assumptions and the consequent as its conclusion:

1
```
> disch_then;
val it = fn: thm_tactic -> tactic

> disch_then SUBST1_TAC;
val it = fn: tactic

> g `(n = 0) ==> (n * n = n)`;
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        n = 0 ⇒ n * n = n

> e(disch_then SUBST1_TAC);
OK..
1 subgoal:
val it =

   0 * 0 = 0
```

This gives the same result as the stack method, but more directly, with a more compact ML expression, and with the attractive feature that the term $n = 0$ is never an assumption, even for an interval of one step. This technique is often used at the moment when results are available; as above, where the result produced by discharging the antecedent can be immediately passed to substitution. If the result were only needed later, it *would* have to be held as an assumption. However, results can be manipulated when they are available, and their results either held as assumptions or used immediately. For example, to prove $(0 = n) \Rightarrow (n \times n = n)$, the result $n = 0$ could be reversed immediately:

```
> g `(0 = n) ==> (n * n = n)`;                                    1
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        0 = n ⇒ n * n = n
> e(disch_then(SUBST1_TAC o SYM));
OK..
1 subgoal:
val it =

   0 * 0 = 0
```

The justification of `disch_then SUBST1_TAC` is easily constructed from the justification of `DISCH_TAC` composed with the justification of `SUBST1_TAC`. The term $n = 0$ is assumed, to yield the theorem that is passed to the theorem continuation `SUBST1_TAC`, and it is accordingly discharged during the construction of the actual proof; but the assumption happens only internally to the tactic `disch_then SUBST1_TAC`, and not as a step in the tactical proof. In other words, the subgoal tree here has one node fewer than before, when an explicit step (`DISCH_TAC`) reflected the assumption.

On the goal with the disjunctive antecedent, this method again provides a compact tactic:

```
> g `((n = 0) \/ (n = 1)) ==> (n * n = n)`;                       1
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        n = 0 ∨ n = 1 ⇒ n * n = n

> e(disch_then(DISJ_CASES_THEN SUBST1_TAC));
OK..
2 subgoals:
val it =

   1 * 1 = 1

   0 * 0 = 0
```

This avoids the repeated popping and pushing of the stack solution, and likewise, gives a shorter ML expression. Both give a shorter expression than the direct method, which is:

```
DISCH_TAC
 >> DISJ_CASES_TAC(ASSUME ``(n = 0) \/ (n = 1)``)
 >| [SUBST1_TAC(ASSUME ``n = 0``);
     SUBST1_TAC(ASSUME ``n = 1``)]
```

To summarize, there are so far at least five ways to solve a goal (and these are often combined in one interaction): directly, using the stack view of the assumptions, using the array view with or without discarding assumptions, and using a tactical to intercept an assumption step. All of the following work on the goal $(n = 0) \Rightarrow (n \times n = n)$:

```
DISCH_TAC
 >> SUBST1_TAC(ASSUME ``n = 0``)
 >> simp[]

DISCH_TAC
 >> pop_assum SUBST1_TAC
 >> simp[]

disch_then SUBST1_TAC
 >> simp[]
```

Furthermore, all induce the same sequence of inferences leading to the desired theorem; internally, no inference steps are saved by the economies in the ML text or the subgoal tree. In this sense, the choice is entirely one of style and taste; of how to organize the decomposition into subgoals. The first expression illustrates the verbosity of denoting assumptions by text (the goal with the disjunctive antecedent gave a clearer example); but also the intelligibility of the resulting expression, which, of course, is all that is saved of the interaction, aside from the final theorem. The last expression illustrates both the elegance and the inscrutibility of using functions to manipulate intermediate results directly, rather than as assumptions. The middle expression shows how results can be used as assumptions (discarded when redundant, if desired); and how assumptions can be denoted without recourse to their text. It is a strength of the LCF approach to theorem proving that many different proof styles are supported, (all in a secure way) and indeed, can be studied in their own right.

HOL provides several other theorem continuation functions analogous to `disch_then` and `DISJ_CASES_THEN`. (Their names always end with '`_THEN`', '`_THENL` or '`_THEN2`'.) Some of these do convenient inferences for the user. For example:

```
CHOOSE_THEN : thm_tactical
```

Where `thm_tactical` abbreviates `thm_tactic -> tactic`. `CHOOSE_THEN` f $(\vdash \exists x.\, t[x])$ is a tactic that, given a goal, generates the subgoal obtained by applying f to $(t[x]\,|\!-t[x])$. The intuition is that if `|-` $\exists x.\, t[x]$ holds then `|-` $t[x]$ holds for some value of x (as long as the variable x is not free elsewhere in the theorem or current goal). This gives an easy way of using existentially quantified theorems, something that is otherwise awkward.

The new method has other applications as well, including as an implementation technique. For example, taking `DISJ_CASES_THEN` as basic, `DISJ_CASES_TAC` can be defined by:

```
val DISJ_CASES_TAC = DISJ_CASES_THEN ASSUME_TAC
```

Similarly, the method is useful for modifying existing tactics (*e.g.* resolution tactics) without having to re-program them in ML. This avoids the danger of introducing tactics whose justifications may fail, a particularly difficult problem to track down; it is also much easier than starting from scratch.

The main theorem continuation functions in the system are:

```
ANTE_RES_THEN
CHOOSE_THEN       X_CHOOSE_THEN
CONJUNCTS_THEN    CONJUNCTS_THEN2
DISJ_CASES_THEN   DISJ_CASES_THEN2   DISJ_CASES_THENL
DISCH_THEN
IMP_RES_THEN
RES_THEN
STRIP_THM_THEN
STRIP_GOAL_THEN
```

See *REFERENCE* for full details.

Chapter 5

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`.

5.1 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 [7], 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`.

5.2 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.

5.2.1 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 1.6, 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 1.6 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 5.2.3.2.

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.

5.2.1.1 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.

5.2.1.2 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 1.8), 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	`$=` t_1 t_2	`mk_eq(`t_1`,`t_2`)`
∀-quantification	`!`x`.`t	`$!(\`x`.`t`)`	`mk_forall(`x`,`t`)`
∃-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 5.2.1. The constants `\/`, `/\`, `==>` and `=` are examples of *infixes* and represent ∨, ∧, ⇒ 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 ∀, ∃ 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.*

5.2.1.3 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.

5.2.2 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 g. f ∘ g = (λx. f (g x))
APP_DEF
  ⊢ ∀x f. (x :> f) = f x
UPDATE_def
  ⊢ ∀a b. (a =+ b) = (λf c. if a = c then b else f c)
K_DEF
  ⊢ K = (λx y. x)
S_DEF
  ⊢ S = (λf g x. f x (g x))
I_DEF
  ⊢ I = S K K
W_DEF
  ⊢ W = (λf x. f x x)
C_DEF
  ⊢ combin$C = (λf x y. f y x)
```

The following elementary properties are proved in the theory `combin`:

```
o_THM
  ⊢ ∀f g x. (f ∘ g) x = f (g x)
o_ASSOC
  ⊢ ∀f g h. f ∘ g ∘ h = (f ∘ g) ∘ h

UPDATE_EQ
  ⊢ ∀f a b c. f⦇a ↦ c; a ↦ b⦈ = f⦇a ↦ c⦈
UPDATE_COMMUTES
  ⊢ ∀f a b c d. a ≠ b ⇒ f⦇a ↦ c; b ↦ d⦈ = f⦇b ↦ d; a ↦ c⦈
APPLY_UPDATE_THM
  ⊢ ∀f a b c. f⦇a ↦ b⦈ c = if a = c then b else f c

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. combin$C f x y = f y x
```

The above illustrates that there are two ways of writing function update terms. As per the definition above (`UPDATE_def`), the infix `=+` takes a key k and a value v, and returns a higher-order function, which when in turn is passed a function f, returns a version of that function that has been updated to return v when applied to k, and is otherwise the same as f. The same effect can be achieved with the "substitution style" syntax: $f(\!|k \mapsto v|\!)$. There is an ASCII form of this notation as well:

```
> ``(k2 =+ v2) ((k1 =+ v1) f)``;                                        1
<<HOL message: inventing new type variable names: 'a, 'b>>
val it = "f(|k2 ↦ v2; k1 ↦ v1|)": term
> ``f (| k |-> v |)``;
<<HOL message: inventing new type variable names: 'a, 'b>>
val it = "f(|k ↦ v|)": term
```

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 7.1.2.9). 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`. It is also printed this way, as can be seen above.

5.2.3 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 $(t_1,(t_2,\ldots,t_n))$.

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

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)
```

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
```

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

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 5.2.3.1.

Wellfounded relations on pairs Wellfoundedness, defined in Section 5.3.1.4, 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)
```

5.2.3.1 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 $\backslash(x_1, x_2).t$ to UNCURRY($\backslash x_1\ x_2.t$). The transformation is done recursively so that, for example,

$\backslash(x_1, x_2, x_3).t$

is converted to

```
UNCURRY (\x1. UNCURRY(\x2 x3.t))
```

More generally, the quotation parser repeatedly applies the transformation:

```
\(v1,v2).t   ⇝   UNCURRY(\v1.\v2.t)
```

until no more variable structures remain. For example:

```
\(x,y).t                      ⇝ UNCURRY(\x y.t)
\(x1,x2,...,xn).t             ⇝ UNCURRY(\x1.\(x2,...,xn).t)
\((x1,...,xn),y1,...,ym).t    ⇝ UNCURRY(\(x1,...,xn).\(y1,...,ym).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:

2

```
> ``\(x,y).x+y``;
val it = “λ(x,y). x + y”: term

> val p = Term `(x:num,y:num)`;
val p = “(x,y)”: term

> Lib.try Term `\^p.x+y` handle _ => T

Exception raised at Term.dest_var:
not a var
val it = “T”: term
```

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))`.

5.2.3.2 `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); a = 20*20; 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` $\ldots$ `and` v_n=t_n `in` t	$\rightsquigarrow$ LET(...(LET(LET(\$v_1\ldots v_n.t$)$t_1$)$t_2$)...)$t_n$

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

```
> Term `let x = 1; y = 2; in x+y`;                                    3
val it = “let x = 1 ; y = 2 in x + y”: term

> dest_comb it;
val it = (“LET (λx. (let y = 2 in x + y))”, “1”): term * term

> Term `let (x,y) = (1,2) in x+y`;
val it = “let (x,y) = (1,2) in x + y”: term

> dest_comb it;
val it = (“LET (λ(x,y). x + y)”, “(1,2)”): term * term
```

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

5.2.4 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 ‘‘:σ_1+σ_2‘‘ into ‘‘:(σ_1,σ_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.

[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 [9].

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) ∧ ∀y. h (INR y) = g y
```

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
```

The sum type operator can be seen as functorial over its arguments and so has a "map" function, SUM_MAP, with definition and results showing its functoriality:

```
SUM_MAP_def
  ⊢ (∀f g a. SUM_MAP f g (INL a) = INL (f a)) ∧
     ∀f g b. SUM_MAP f g (INR b) = INR (g b)
SUM_MAP_I
  ⊢ SUM_MAP I I = I
SUM_MAP_o
  ⊢ SUM_MAP f g ∘ SUM_MAP h k = SUM_MAP (f ∘ h) (g ∘ k)
```

5.2.5 The one-element type

The theory one defines the type one which contains one element. The type is also abbreviated as unit, which is the name of the analogous type in ML, and this is the type's preferred printing form. The constant one denotes this one element, but, again by analogy with ML, the preferred parsing and printing form for this constant is ().[5] The

[5] When using the parenthesis-version, the one value's syntax consists of two parenthesis tokens, so that one can write the value with white-space between the parentheses if desired.

pre-proved theorems in the theory one are:

```
one_axiom
  ⊢ ∀(f :α -> unit) (g :α -> unit). f = g
one
  ⊢ ∀(v :unit). v = ()
one_Axiom
  ⊢ ∀(e :α). ∃!(fn :unit -> α). fn () = 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.

5.2.5.1 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 5.3.8 below).

5.2.6 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. ∃fn. fn NONE = e ∧ ∀x. fn (SOME x) = f x
```

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
  ⊢ (∀v f. option_CASE NONE v f = v) ∧
     ∀x v f. option_CASE (SOME x) v f = f x
```

Another useful function maps a function over an option:

```
OPTION_MAP_DEF
  ⊢ (∀f x. OPTION_MAP f (SOME x) = SOME (f x)) ∧
     ∀f. OPTION_MAP f NONE = NONE
```

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

```
THE_DEF
  ⊢ ∀x. THE (SOME x) = x
```

5.3 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.

5.3.1 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.

5.3.1.1 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.

5.3.1.2 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.[6] 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
  ⊢ ∀e f. ∃fn. fn 0 = e ∧ ∀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.

[6] 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_DEF
  ⊢ ∀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[7] and contains `m` but not `n`.

5.3.1.3 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

$$\begin{array}{l}\texttt{fun 0}\quad\ \texttt{x}_1\ \ldots\ \texttt{x}_i = f_1[\texttt{x}_1, \ldots, \texttt{x}_i] \\ \texttt{fun (SUC n)}\ \texttt{x}_1\ \ldots\ \texttt{x}_i = f_2[\texttt{fun n}\ t_1\ \ldots\ t_i, \texttt{n}, \texttt{x}_1, \ldots, \texttt{x}_i]\end{array}$$

where all the free variables in the terms $t_1, \ldots, t_i$ are contained in $\{\texttt{n}, \texttt{x}_1, \ldots, \texttt{x}_i\}$, is logically equivalent to:

$$\begin{array}{l}\texttt{fun 0}\quad\ = \texttt{\textbackslash x}_1\ \ldots\ \texttt{x}_i . f_1[\texttt{x}_1, \ldots, \texttt{x}_i] \\ \texttt{fun (SUC n)} = \texttt{\textbackslash x}_1\ \ldots\ \texttt{x}_i . f_2[\texttt{fun n}\ t_1\ \ldots\ t_i, \texttt{n}, \texttt{x}_1, \ldots, \texttt{x}_i] \\ \qquad\qquad = (\texttt{\textbackslash f n x}_1\ \ldots\ \texttt{x}_i . f_2[\texttt{f}\ t_1\ \ldots\ t_i, \texttt{n}, \texttt{x}_1, \ldots, \texttt{x}_i])\ \texttt{(fun n) n}\end{array}$$

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

$$\texttt{x=\textbackslash x}_1\ \ldots\ \texttt{x}_i . f_1[\texttt{x}_1, \ldots, \texttt{x}_i] \quad \text{and} \quad \texttt{f=\textbackslash f n x}_1\ \ldots\ \texttt{x}_i . f_2[\texttt{f}\ t_1\ \ldots\ t_i, \texttt{n}, \texttt{x}_1, \ldots, \texttt{x}_i])$$

yields the existence theorem shown below:

$$\begin{array}{l}\texttt{|- ?fn. fn 0}\quad\ = \texttt{\textbackslash x}_1\ \ldots\ \texttt{x}_i . f_1[\texttt{x}_1, \ldots, \texttt{x}_i]\ \texttt{/\textbackslash} \\ \qquad \texttt{fn (SUC n) = (\textbackslash f n x}_1\ \ldots\ \texttt{x}_i . f_2[\texttt{f}\ t_1\ \ldots\ t_i, \texttt{n}, \texttt{x}_1, \ldots, \texttt{x}_i])\ \texttt{(fn n) n}\end{array}$$

This theorem allows a constant `fun` to be introduced (via the definitional mechanism of constant specifications—see Section 1.8.3.2) 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:

[7] 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 7.3.1) 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.

5.3.1.4 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
  ⊢ ∀R. Wellfounded R ⟺ ¬∃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 (λx 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)
```

5.3.2 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 ** 0 = 1) ∧ ∀m n. m ** SUC n = m * m ** 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). In addition, the parser special-cases superscript 2 and 3 notations, so that x^2 is actually the same term as x EXP 2, and x^3 is the same term as x EXP 3.

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 âĽě 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 5.6 for discussion of the LEAST operator, which returns the least number satisfying a predicate.

5.3.2.1 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

5.3.3 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 quickly:

```
> Count.apply reduceLib.REDUCE_CONV ``2 EXP ((1023 + 14) DIV 9)``;                1
runtime: 0.00183s,    gctime: 0.00000s,     systime: 0.00017s.
Axioms: 0, Defs: 0, Disk: 0, Orcl: 0, Prims: 4202; Total: 4202
val it = ⊢ 2 ** ((1023 + 14) DIV 9) = 415383748682786210282439706337607 68:
   thm
```

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.*

```
> numSyntax.mk_numeral                                                            2
    (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

```
> ``0xAA``;                                                              3
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:

```
> open numLib;  ...output elided...                                      4
> 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.

5.3.3.1 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.

```
> load "integerTheory";                                                  5
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):

```
> type_of ``2n``;                                                    6
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 5.3.4 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*.

5.3.4 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_mul	*	600	left
/		600	left
%		650	left
int_exp	**	700	right
int_of_num	&	900	prefix
int_neg	~	900	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.

```
> Term `1i = &(1n + 0n)`;                                                    1
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
```

5.3.5 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_minv			
rat_mul	*	600	left
rat_div	/	600	left
rat_ainv	~	900	prefix
rat_of_num	&	900	prefix

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.

5.3.6 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 [5]. 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_mul`	`*`	600	left
`real_div`	`/`	600	left
`pow`		700	right
`real_of_num`	`&`	900	prefix
`real_neg`	`~`	900	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.

5.3.7 Probability theory

A foundational construction of probability theory developed by Joe Hurd [6]. 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.

5.3.8 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.[8]

5.3.8.1 Finite Cartesian products

The HOL theory `fcp` introduces an infix type operator `**`, which is used to represent finite Cartesian products.[9] 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

[8] 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.

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

```
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`.

5.3.8.2 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`.

5.3.8.3 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]→*bool*[α]→*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.[10]

Some basic operations The function `w2n:bool[α]→num` gives the natural number value of a word. If $x \in \mathbf{T}^{\{0,1,\ldots,n-1\}}$ 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]→num`. The function `n2w:num→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]→bool[β]` provides word-to-word conversion (casting):

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

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.*

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

The function `sw2sw:bool[α]→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.

[10] 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.

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 expected 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[11] 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)`.

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,

[11] 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_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`:$(num \to bool \to bool) \to bool\,[\alpha] \to bool\,[\alpha]$ 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 5.1 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.

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)
```

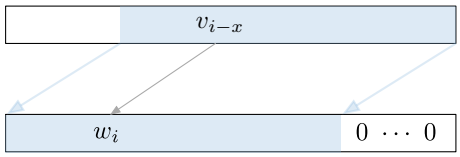
The + on the left-hand side is word addition and on the right it is natural number addition.

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".

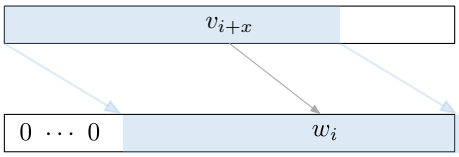
Constants The word theory also defines a few word constants:

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`	$2^{l-1} - 1$	$01 \cdots 11$

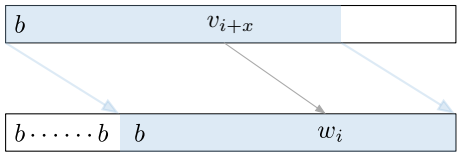
List of bit vector operations A list of operations is provided in the table below.



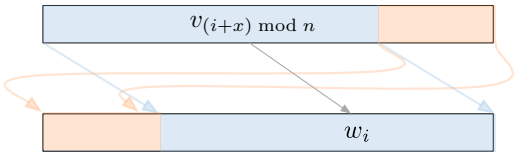
(a) Logical shift left: `w = v << x`.



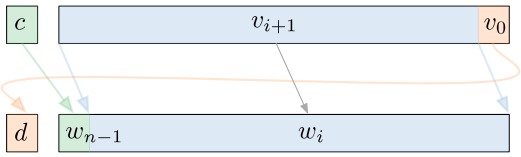
(b) Logical shift right: `w = v >>> x`.



(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 5.1: Shift operations.

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

continued on next page

continued from previous page

Operation	Symbol	Type	Description
word_modify		$(num \rightarrow bool \rightarrow bool) \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Apply a function to each bit
word_join		$bool[\alpha] \rightarrow bool[\beta] \rightarrow bool[\alpha + \beta]$	Join words
word_concat	@@	$bool[\alpha] \rightarrow bool[\beta] \rightarrow bool[\gamma]$	Concatenate words
concat_word_list		$bool[\alpha]\ list \rightarrow bool[\beta]$	Concatenate list of words
word_replicate		$num \rightarrow bool[\alpha] \rightarrow bool[\beta]$	Replicate word
word_or	||	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Bitwise disjunction
word_xor	??	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Bitwise exclusive-or
word_and	&&	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Bitwise conjunction
word_nor	~||	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Bitwise NOR
word_xnor	~??	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Bitwise XNOR
word_nand	~&&	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Bitwise NAND
word_reduce		$(bool \rightarrow bool \rightarrow bool) \rightarrow bool[\alpha] \rightarrow bool[1]$	Word reduction
reduce_or		$bool[\alpha] \rightarrow bool[1]$	Disjunction reduction
reduce_xor		$bool[\alpha] \rightarrow bool[1]$	Exclusive-or reduction
reduce_and		$bool[\alpha] \rightarrow bool[1]$	Conjunction reduction
reduce_nor		$bool[\alpha] \rightarrow bool[1]$	NOR reduction
reduce_xnor		$bool[\alpha] \rightarrow bool[1]$	XNOR reduction
reduce_nand		$bool[\alpha] \rightarrow bool[1]$	NAND reduction
word_1comp	~	$bool[\alpha] \rightarrow bool[\alpha]$	One's complement
word_2comp	-	$bool[\alpha] \rightarrow bool[\alpha]$	Two's complement
word_add	+	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Addition
word_sub	-	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Subtraction
word_mul	*	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Multiplication
word_div	//	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Division (unsigned)
word_sdiv	/	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Division (signed)
word_mod		$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool[\alpha]$	Modulus
word_log2		$bool[\alpha] \rightarrow bool[\alpha]$	Logarithm base-2
word_lsl	<<	$bool[\alpha] \rightarrow num \rightarrow bool[\alpha]$	Logical shift left
word_lsr	>>>	$bool[\alpha] \rightarrow num \rightarrow bool[\alpha]$	Logical shift right
word_asr	>>	$bool[\alpha] \rightarrow num \rightarrow bool[\alpha]$	Arithmetic shift right
word_ror	#>>	$bool[\alpha] \rightarrow num \rightarrow bool[\alpha]$	Rotate right
word_rol	#<<	$bool[\alpha] \rightarrow num \rightarrow bool[\alpha]$	Rotate left
word_rrx		$bool\#bool[\alpha] \rightarrow bool\#bool[\alpha]$	Rotate right extended by 1 place
word_lt	<	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool$	Signed "less than"
word_le	<=	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool$	Signed "less than or equal"
word_gt	>	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool$	Signed "greater than"
word_ge	>=	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool$	Signed "greater than or equal"
word_lo	<+	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool$	Unsigned "less than"
word_ls	<=+	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool$	Unsigned "less than or equal"
word_hi	>+	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool$	Unsigned "greater than"
word_hs	>=+	$bool[\alpha] \rightarrow bool[\alpha] \rightarrow bool$	Unsigned "greater than or equal"

continued on next page

continued from previous page			
Operation	Symbol	Type	Description

5.4 Sequences

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

5.4.1 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
  ⊢ ∀f0 f1. ∃fn. fn [] = f0 ∧ ∀a0 a1. fn (a0::a1) = f1 a0 a1 (fn a1)
list_INDUCT
  ⊢ ∀P. P [] ∧ (∀t. P t ⇒ ∀h. P (h::t)) ⇒ ∀l. P l
list_CASES
  ⊢ ∀l. l = [] ∨ ∃h t. l = h::t
CONS_11
  ⊢ ∀a0 a1 a0' a1'. a0::a1 = a0'::a1' ⇔ a0 = a0' ∧ a1 = a1'
NOT_NIL_CONS
  ⊢ ∀a1 a0. [] ≠ a0::a1
NOT_CONS_NIL
  ⊢ ∀a1 a0. a0::a1 ≠ []
```

The theorem `list_Axiom` shown above is analogous to the primitive recursion theorem on the natural numbers discussed above in Section 5.3.1.3. 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_1 e_2 ($\lambda h\ t.\ e_3$) where the constant `list_CASE` is defined as follows:

```
list_case_def
  ⊢ (∀v f. list_CASE [] v f = v) ∧
     ∀a0 a1 v f. list_CASE (a0::a1) v f = f a0 a1
```

List membership Membership in a list, written using the `MEM` syntax, is characterised 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`. The special case where a single element is appended to the end of a list (the "opposite" of `CONS`, which adds elements to the front of a list), is implemented by `SNOC`.

```
APPEND
  ⊢ (∀l. [] ++ l = l) ∧ ∀l1 l2 h. h::l1 ++ l2 = h::(l1 ++ l2)
FLAT
  ⊢ FLAT [] = [] ∧ ∀h t. FLAT (h::t) = h ++ FLAT t
SNOC
  ⊢ (∀x. SNOC x [] = [x]) ∧ ∀x x' l. SNOC x (x'::l) = x'::SNOC x l
```

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.

The GENLIST constant can be used to generate a list of a particular size, where the value of each element is independently determined by reference to a function that takes natural numbers (the set $\{0 \ldots n-1\}$) to element values:

```
GENLIST
  ⊢ (∀f. GENLIST f 0 = []) ∧
     ∀f n. GENLIST f (SUC n) = SNOC (f n) (GENLIST f n)
EL_GENLIST
  ⊢ ∀f n x. x < n ⇒ EL x (GENLIST f n) = f x
```

Working with SNOC, and thus the definition above, can occasionally be awkward, so a characterisation of GENLIST's SUC clause in terms of CONS can also be useful:

```
GENLIST_CONS
  ⊢ GENLIST f (SUC n) = f 0::GENLIST (f ∘ SUC) n
```

For more on the "indexed" treatment of lists, see Section 5.4.1.2 below.

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_DEF
  ⊢ (∀t2 t1 h2 h1 f. MAP2 f (h1::t1) (h2::t2) = f h1 h2::MAP2 f t1 t2) ∧
     (∀y f. MAP2 f [] y = []) ∧ ∀v5 v4 f. MAP2 f (v4::v5) [] = []
```

If passed lists of unequal length, MAP2 returns a list of length equal to that of the shorter list.

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.

Relations over lists A binary relation on elements can be "lifted" to a relation on lists of such elements with the LIST_REL constant:

```
LIST_REL_def
  ⊢ (LIST_REL R [] [] ⟺ T) ∧ (LIST_REL R (a::as) [] ⟺ F) ∧
     (LIST_REL R [] (b::bs) ⟺ F) ∧
     (LIST_REL R (a::as) (b::bs) ⟺ R a b ∧ LIST_REL R as bs)
```

This can be viewed as an application of EVERY:

```
LIST_REL_EVERY_ZIP
  ⊢ ∀R l1 l2.
         LIST_REL R l1 l2 ⟺
         LENGTH l1 = LENGTH l2 ∧ EVERY (UNCURRY R) (ZIP (l1,l2))
```

Acknowledging this view, the system overloads the name EVERY2 to map to the same constant.

```
> “EVERY2 (λm n. EVEN (m + n)) [1;2;3] [3;4;5]”;
val it = “LIST_REL (λm n. EVEN (m + n)) [1; 2; 3] [3; 4; 5]”: term
```

Some theorems in `listTheory` have names that reflect this.

Equally, `LIST_REL` can be seen as a test that checks the relation at all relevant indices:

```
LIST_REL_EL_EQN
  ⊢ ∀R l1 l2.
        LIST_REL R l1 l2 ⟺
        LENGTH l1 = LENGTH l2 ∧ ∀n. n < LENGTH l1 ⇒ R (EL n l1) (EL n l2)
```

Finally, there is a natural induction principle for this constant (as *per* Section 6.6.1, the tactic `Induct_on ‘LIST_REL‘` applies it):

```
LIST_REL_strongind
  ⊢ ∀R LIST_REL'.
        LIST_REL' [] [] ∧
        (∀h1 h2 t1 t2.
             R h1 h2 ∧ LIST_REL R t1 t2 ∧ LIST_REL' t1 t2 ⇒
             LIST_REL' (h1::t1) (h2::t2)) ⇒
        ∀a0 a1. LIST_REL R a0 a1 ⇒ LIST_REL' a0 a1
```

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))
```

Removal of duplicates The `nub` function removes all duplicate entries from a list.

```
nub_def
  ⊢ nub [] = [] ∧ ∀x l. nub (x::l) = if MEM x l then nub l else x::nub l
```

Conversion to sets Lists can be converted to sets with the LIST_TO_SET constant, which is overloaded to the prettier name set. The definition is made by primitive recursion in listTheory:

```
> listTheory.LIST_TO_SET;
val it = ⊢ set [] = ∅ ∧ set (h::t) = h INSERT set t: thm
```

Note that MEM is an overloaded form of syntax such that MEM x l is actually a pretty-printing of the underlying term x ∈ set 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. As with MAP2, the result of zipping lists of unequal lengths is a list whose length is that of the shorter argument. The inverse operation, UNZIP, translates a list of pairs into a pair of lists.

```
ZIP_def
  ⊢ (∀l2. ZIP ([],l2) = []) ∧ (∀l1. ZIP (l1,[]) = []) ∧
     ∀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 symbols <<= (ASCII) and ≼ (U+227C) 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`.

5.4.1.1 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. The `mergesort` theory defines Merge sort and shows that it is a stable 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
  ⊢ (∀R. SORTED R [] ⟺ T) ∧ (∀x R. SORTED R [x] ⟺ T) ∧
     ∀y x rst R. 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
  ⊢ (∀ord. QSORT ord [] = []) ∧
     ∀t ord h.
         QSORT ord (h::t) =
         (let
            (l1,l2) = PARTITION (λy. ord y h) t
          in
            QSORT ord l1 ++ [h] ++ QSORT ord l2)
QSORT_SORTS
  ⊢ ∀R. transitive R ∧ total R ⇒ SORTS QSORT R
```

5.4.1.2 Indexed lists

As mentioned earlier, lists can be indexed with the constant EL, viewing lists as partial functions from natural numbers (starting at 0!) into the element type. The definition is given by primitive recursion over the index argument, in theorem EL:

```
EL
  ⊢ (∀l. EL 0 l = HD l) ∧ ∀l n. EL (SUC n) l = EL n (TL l)
```

Because of the use of HD and TL, the value of EL $n\ \ell$ is unspecified when $n \geq$ LENGTH ℓ. Subsequently, many theorems involving EL have preconditions to preclude this possibility.

For example, these theorems describing the relationship between EL, MAP and MEM:

```
EL_MAP
  ⊢ ∀n l. n < LENGTH l ⇒ ∀f. EL n (MAP f l) = f (EL n l)
MEM_EL
  ⊢ ∀l x. MEM x l ⟺ ∃n. n < LENGTH l ∧ x = EL n l
```

It is occasionally useful to be able to update lists at particular positions, viewing them as similar to programming language arrays. The relevant constant is LUPDATE, where the term LUPDATE $e\ n\ \ell$ has the same value as list ℓ, except that the n-th element of the list is equal to e. The definition is given in three clauses:

```
LUPDATE_def
  ⊢ (∀e n. LUPDATE e n [] = []) ∧ (∀e x l. LUPDATE e 0 (x::l) = e::l) ∧
     ∀e n x l. LUPDATE e (SUC n) (x::l) = x::LUPDATE e n l
```

The definition implies that attempting to update a list at an index beyond the end of the list returns the input list unchanged.

The basic characterisation of the link between EL and LUPDATE is

```
EL_LUPDATE
  ⊢ ∀ys x i k.
         EL i (LUPDATE x k ys) = if i = k ∧ k < LENGTH ys then x else EL i ys
```

The `indexedLists` theory The `indexedLists` theory defines a number of extra constants that are "aware" of lists as indexed values. Some of these constants are:

```
delN      : num -> 'a list -> 'a list
findi     : 'a -> 'a list -> num
LIST_RELi : (num -> 'a -> 'b -> bool) -> 'a list -> 'b list -> bool
MAPi      : (num -> 'a -> 'b) -> 'a list -> 'b list
```

The `findi` constant is such that `findi` $e\ \ell$ returns the first index of element e within list ℓ, or a number equal to ℓ's length, if e is not present. The definition is by recursion over the structure of the input list:

```
findi_def
  ⊢ (∀x. findi x [] = 0) ∧
     ∀x h t. findi x (h::t) = if x = h then 0 else 1 + findi x t
```

The `delN` constant is used to remove the n-th element from a list. It is also defined by recursion over the structure of the input list:

```
delN_def
  ⊢ (∀i. delN i [] = []) ∧
     ∀i h t. delN i (h::t) = if i = 0 then t else h::delN (i − 1) t
```

The higher-order `MAPi` function exemplifies another set of constants within the `indexedLists` theory: its function parameter, which works on elements of the list argument is given access to the index of the list element as well as its value. A simple example use might be to generate a numbered version of a list, using the term `MAPi` ($\lambda i\ e.\ (i, e)$). If this term were applied to the list `[a;c;d]` the resulting value would be `[(0,a);(1,c);(2,d)]`.

An example theorem about `MAPi` relates it to `MEM`:

```
MEM_MAPi
  ⊢ ∀x f l. MEM x (MAPi f l) ⟺ ∃n. n < LENGTH l ∧ x = f n (EL n l)
```

5.4.2 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.
        ∃g.
            (∀x. LHD (g x) = OPTION_MAP SND (f x)) ∧
            ∀x. LTL (g x) = OPTION_MAP (g ∘ FST) (f x)
```

An equivalent form of the above is

```
llist_Axiom_1
  ⊢ ∀f. ∃g. ∀x. g x = case f x of NONE => [||] | SOME (a,b) => b:::g a
```

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
  ⊢ (∀f. LMAP f [||] = [||]) ∧ ∀f h t. LMAP f (h:::t) = f h:::LMAP f t
LFINITE_THM
  ⊢ (LFINITE [||] ⟺ T) ∧ ∀h t. LFINITE (h:::t) ⟺ LFINITE t
LNTH_THM
  ⊢ (∀n. LNTH n [||] = NONE) ∧ (∀h t. LNTH 0 (h:::t) = SOME h) ∧
     ∀n h t. LNTH (SUC n) (h:::t) = LNTH n t
LTAKE_THM
  ⊢ (∀l. LTAKE 0 l = SOME []) ∧ (∀n. LTAKE (SUC n) [||] = NONE) ∧
     ∀n h t. LTAKE (SUC n) (h:::t) = OPTION_MAP (CONS h) (LTAKE n t)
LDROP_THM
  ⊢ (∀ll. LDROP 0 ll = SOME ll) ∧ (∀n. LDROP (SUC n) [||] = NONE) ∧
     ∀n h t. LDROP (SUC n) (h:::t) = LDROP n t
LFILTER_THM
  ⊢ (∀P. LFILTER P [||] = [||]) ∧
     ∀P h t. LFILTER P (h:::t) = if P h then 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 [||] x = x) ∧ ∀h t x. LAPPEND (h:::t) x = h:::LAPPEND t x
LFLATTEN_THM
  ⊢ LFLATTEN [||] = [||] ∧ (∀tl. LFLATTEN ([||]:::t) = LFLATTEN t) ∧
     ∀h t tl. LFLATTEN ((h:::t):::tl) = h:::LFLATTEN (t:::tl)
```

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

```
fromList_def
  ⊢ fromList [] = [||] ∧ ∀h t. fromList (h::t) = h:::fromList t
toList_THM
  ⊢ toList [||] = SOME [] ∧
     ∀h t. toList (h:::t) = OPTION_MAP (CONS h) (toList t)
```

Note that `toList ll = NONE` when `ll` is infinite.

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. ll1 = ll2 ⟺ ∀n. LTAKE n ll1 = LTAKE n ll2
```

This theorem is subsequently used to derive the bisimulation principle:

```
LLIST_BISIMULATION
  ⊢ ∀ll1 ll2.
        ll1 = ll2 ⟺
        ∃R.
            R ll1 ll2 ∧
            ∀ll3 ll4.
                R ll3 ll4 ⇒
                ll3 = [||] ∧ ll4 = [||] ∨
                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.

5.4.3 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
  ⊢ (∀x. first (stopped_at x) = x) ∧ ∀x r p. first (pcons x r 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
  ⊢ (∀x. last (stopped_at x) = x) ∧ ∀x r p. last (pcons x r 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
  ⊢ (∀x. finite (stopped_at x) ⟺ T) ∧
     ∀x r p. finite (pcons x r 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
  ⊢ (∀x lab p2. pconcat (stopped_at x) lab p2 = pcons x lab p2) ∧
     ∀x r p 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
  ⊢ ∀R.
        (∀x. okpath R (stopped_at x)) ∧
        ∀x r p. 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
  ⊢ ∀R.
        (∀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)) ⇒
        ∀sigma. okpath R sigma ∧ finite sigma ⇒ P sigma
```

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
```

5.4.4 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
```
> load "stringTheory";
val it = (): unit
> val t = ``f #"c" #"\n"``;
<<HOL message: inventing new type variable names: 'a>>
val t = “f #"c" #"\n"”: term

> dest_comb ``#"\t"``;
val it = (“CHR”, “9”): term * term
```

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 "$c_1 c_2 \ldots c_n$" to the compound term

$$\texttt{STRING}\ c_1\ (\texttt{STRING}\ c_2\ \ldots\ (\texttt{STRING}\ c_{n-1}\ (\texttt{STRING}\ c_n\ \texttt{EMPTYSTRING}))\ \ldots)$$

Of course, one could also write

2
```
> ``[#"a"; #"b"]``;
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.

3
```
> val t = ``"foo bar\n\001"``;
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 7.1.3.)

As with numerals, string literals can be injected into other types, where it might make sense to have string literals appear to inhabit types in addition to the core system's `string` type. Such literals can be written with different delimiters to make it clear that such an injection has occurred. For more on this facility, see *REFERENCE*'s description of the `add_strliteral_form` function.

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 can be written with the surface syntax

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

Such an expression is actually a case-expression over the underlying list, and so the underlying constant is that for lists.

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))
```

5.5 Collections

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

5.5.1 Sets (pred_set)

An extensive development of set theory is available in the theory pred_set. Sets are represented by functions of the type $\alpha \to$ bool, *i.e.*, they are so-called characteristic functions. One can use the type abbreviation α set instead of $\alpha \to$ 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 = (λx 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 ∈ P ⟺ P x
```

Two sets are equal if they have the same elements.

```
EXTENSION
  ⊢ ∀s t. s = t ⟺ ∀x. x ∈ s ⟺ x ∈ t
```

The negation of set-membership is not a separate constant, but is available as a convenient overload (in both ASCII and Unicode forms). Thus, instead of writing ~(e IN s), one can instead write e NOTIN s or e ∉ s.

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 ∅ (U+2205). The universal set, UNIV, on a type is the characteristic function that is always true for elements of that type.

```
EMPTY_DEF
  ⊢ ∅ = (λx. F)
UNIV_DEF
  ⊢ 𝕌(:α) = (λx. 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 𝕌(:'a) means the same. The Unicode symbol for 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 8.2) "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 7.1.2.7), and overloaded patterns (see Section 7.1.2.3) 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 ∈ s}
UNION_DEF
  ⊢ ∀s t. s ∪ t = {x | x ∈ s ∨ x ∈ t}
INTER_DEF
  ⊢ ∀s t. s ∩ t = {x | x ∈ s ∧ x ∈ 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 ∈ P ∧ x ∈ s}
BIGINTER
  ⊢ ∀P. BIGINTER P = {x | ∀s. s ∈ P ⇒ x ∈ 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 ⊆ t ⟺ ∀x. x ∈ s ⇒ x ∈ t
PSUBSET_DEF
  ⊢ ∀s t. s ⊂ t ⟺ s ⊆ t ∧ s ≠ t
POW_DEF
  ⊢ ∀set. POW set = {s | s ⊆ 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 ∈ s ∧ x ∉ t}
DELETE_DEF
  ⊢ ∀s x. s DELETE x = s DIFF {x}
COMPL_DEF
  ⊢ ∀P. COMPL P = 𝕌(:α) 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 ∈ s}
```

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

```
INJ_DEF
  ⊢ ∀f s t.
        INJ f s t ⟺
        (∀x. x ∈ s ⇒ f x ∈ t) ∧ ∀x y. x ∈ s ∧ y ∈ s ⇒ f x = f y ⇒ x = y
SURJ_DEF
  ⊢ ∀f s t.
        SURJ f s t ⟺
        (∀x. x ∈ s ⇒ f x ∈ t) ∧ ∀x. x ∈ t ⇒ ∃y. y ∈ 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 ∉ 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 ∈ 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 ⊂ 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 × Q = {p | FST p ∈ P ∧ SND p ∈ Q}
```

Cardinality and cross product are related by the following theorem:

```
CARD_CROSS
  ⊢ ∀P Q. FINITE P ∧ FINITE Q ⇒ CARD (P × 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. Σ f s = ITSET (λe acc. f e + acc) s 0
SUM_IMAGE_THM
  ⊢ ∀f.
        Σ f ∅ = 0 ∧
        ∀e s. FINITE s ⇒ Σ f (e INSERT s) = f e + Σ 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 HOL's formulation, the tree is characterised by making various assumptions about the finite-ness or otherwise of elements reachable using a relation `R`. Then the following version of Koenig's Lemma is stated and proved:

```
KoenigsLemma
  ⊢ ∀R.
        (∀x. FINITE {y | R x y}) ⇒
        ∀x. INFINITE {y | R* x y} ⇒ ∃f. f 0 = x ∧ ∀n. R (f n) (f (SUC n))
```

5.5.1.1 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 $\{t \mid p\}$ is the set of all ts such that p. In HOL, such syntax parses to: `GSPEC(\(`$x_1, \ldots, x_n$`).(`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 ∈ 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 5.2.3.1. 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 `1`. (If the same trace is set to `2`, then the unambiguous notation will never be used.)

5.5.2 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))
```

5.5.3 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
  ⊢ ∀R. transitive R ⟺ ∀x y z. R x y ∧ R y z ⇒ R x z
reflexive_def
  ⊢ ∀R. reflexive R ⟺ ∀x. R x x
irreflexive_def
  ⊢ ∀R. irreflexive R ⟺ ∀x. ¬R x x
symmetric_def
  ⊢ ∀R. symmetric R ⟺ ∀x y. R x y ⟺ R y x
antisymmetric_def
  ⊢ ∀R. antisymmetric R ⟺ ∀x y. R x y ∧ R y x ⇒ x = y
equivalence_def
  ⊢ ∀R. equivalence R ⟺ reflexive R ∧ symmetric R ∧ transitive R
trichotomous
  ⊢ ∀R. trichotomous R ⟺ ∀a b. R a b ∨ R b a ∨ a = b
total_def
  ⊢ ∀R. 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`, or $\emptyset_r$), relation composition (`O`, or $\circ_r$, infix), inversion (`inv`, or $\_^{\mathsf{T}}$ (suffix superscript 'T')), domain (`RDOM`), and range (`RRANGE`).

```
EMPTY_REL_DEF
  ⊢ ∀x y. ∅ᵣ x y ⟺ F
O_DEF
  ⊢ ∀R1 R2 x z. (R1 ∘ᵣ R2) x z ⟺ ∃y. R2 x y ∧ R1 y z
inv_DEF
  ⊢ ∀R x y. Rᵀ x y ⟺ R y x
RDOM_DEF
  ⊢ ∀R x. RDOM R x ⟺ ∃y. R x y
RRANGE
  ⊢ ∀R y. RRANGE R y ⟺ ∃x. R x y
```

Set operations lifted to work on relations include subset (RSUBSET, or $\subseteq_r$, infix), union (RUNION, or $\cup_r$, infix), intersection (RINTER, or $\cap_r$, infix), complement (RCOMPL), and universe (RUNIV, or $\mathbb{U}_r$).

```
RSUBSET
  ⊢ ∀R1 R2. R1 ⊆ᵣ R2 ⟺ ∀x y. R1 x y ⇒ R2 x y
RUNION
  ⊢ ∀R1 R2 x y. (R1 ∪ᵣ R2) x y ⟺ R1 x y ∨ R2 x y
RINTER
  ⊢ ∀R1 R2 x y. (R1 ∩ᵣ R2) x y ⟺ R1 x y ∧ R2 x y
RCOMPL
  ⊢ ∀R x y. RCOMPL R x y ⟺ ¬R x y
RUNIV
  ⊢ ∀x y. 𝕌ᵣ x y ⟺ T
```

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

```
PreOrder
  ⊢ ∀R. PreOrder R ⟺ reflexive R ∧ transitive R
Order
  ⊢ ∀Z. Order Z ⟺ antisymmetric Z ∧ transitive Z
WeakOrder
  ⊢ ∀Z. WeakOrder Z ⟺ reflexive Z ∧ antisymmetric Z ∧ transitive Z
StrongOrder
  ⊢ ∀Z. StrongOrder Z ⟺ irreflexive Z ∧ transitive Z
LinearOrder
  ⊢ ∀R. LinearOrder R ⟺ Order R ∧ trichotomous R
WeakLinearOrder
  ⊢ ∀R. WeakLinearOrder R ⟺ WeakOrder R ∧ trichotomous R
StrongLinearOrder
  ⊢ ∀R. 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. The ASCII syntax for the transitive closure R^+ is meant to suggest

the prettier R^+. Similarly, `R^*` is meant to suggest R^*. Indeed, with Unicode enabled, transitive closure will print with a superscript +, and `RTC` will print as a superscript asterisk.

From the underlying definitions, one can recover the initial rules:

```
TC_RULES
  ⊢ ∀R. (∀x y. R x y ⇒ R⁺ x y) ∧ ∀x y z. R⁺ x y ∧ R⁺ y z ⇒ R⁺ x z
RTC_RULES
  ⊢ ∀R. (∀x. R* x x) ∧ ∀x y z. R x y ∧ R* y z ⇒ R* x z
RTC_RULES_RIGHT1
  ⊢ ∀R. (∀x. R* x x) ∧ ∀x y z. R* x y ∧ R y z ⇒ R* x z
```

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

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

```
TC_CASES1
  ⊢ R⁺ x z ⟺ R x z ∨ ∃y. R x y ∧ R⁺ y z
TC_CASES2
  ⊢ R⁺ x z ⟺ R x z ∨ ∃y. R⁺ x y ∧ R y z

RTC_CASES1
  ⊢ ∀R x y. R* x y ⟺ x = y ∨ ∃u. R x u ∧ R* u y
RTC_CASES2
  ⊢ ∀R x y. R* x y ⟺ x = y ∨ ∃u. R* x u ∧ R u y
RTC_CASES_RTC_TWICE
  ⊢ ∀R x y. R* x y ⟺ ∃u. R* x u ∧ 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. 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. 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 ∧ R^+ x y ∧ R^+ y z ⇒ P x z) ⇒
        ∀u v. R^+ u v ⇒ P u v
RTC_STRONG_INDUCT
  ⊢ ∀R P.
        (∀x. P x x) ∧ (∀x y z. R x y ∧ R^* y z ∧ P y z ⇒ P x z) ⇒
        ∀x y. 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. When applied to an argument, as in EQC R, EQC is written with the suffix ^=. Note how the suffix binds more tightly than function application, so that in EQC_DEF, RC really is applied to the transitive closure of the symmetric closure of R.

```
RC_DEF
  ⊢ ∀R x y. RC R x y ⟺ x = y ∨ R x y
SC_DEF
  ⊢ ∀R x y. SC R x y ⟺ R x y ∨ R y x
EQC_DEF
  ⊢ ∀R. R^= = RC (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
  ⊢ ∀R a. 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 = (λy. 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 6.5. 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 = (λx 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 R⁺
WF_EMPTY_REL
  ⊢ WF ∅ᵣ
```

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
  ⊢ ∀R. diamond R ⟺ ∀x y z. R x y ∧ R x z ⇒ ∃u. R y u ∧ R z u
CR_def
  ⊢ ∀R. CR R ⟺ diamond R*
WCR_def
  ⊢ ∀R. WCR R ⟺ ∀x y z. R x y ∧ R x z ⇒ ∃u. R* y u ∧ R* z u
SN_def
  ⊢ ∀R. SN R ⟺ WF R^T
```

From those, Newman's Lemma is proved.

```
Newmans_lemma
  ⊢ ∀R. WCR R ∧ SN R ⇒ CR R
```

5.5.4 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`, *i.e.*, 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 ∈ FDOM f ⇒ f ' x = g ' x
fmap_INDUCT
  ⊢ ∀P. P FEMPTY ∧ (∀f. P f ⇒ ∀x y. x ∉ FDOM f ⇒ P (f |+ (x,y))) ⇒ ∀f. P f
FCARD_DEF
  ⊢ ∀fm. FCARD fm = CARD (FDOM fm)
FCARD_FUPDATE
  ⊢ ∀fm a b.
        FCARD (fm |+ (a,b)) = if a ∈ 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
  ⊢ $|++ = FOLDL $|+
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
  ⊢ ∀f. FRANGE f = {y | ∃x. x ∈ 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 ∩ r ∧
        ∀x. DRESTRICT f r ' x = if x ∈ FDOM f ∩ r then f ' x else FEMPTY ' x
RRESTRICT_DEF
  ⊢ ∀f r.
        FDOM (RRESTRICT f r) = {x | x ∈ FDOM f ∧ f ' x ∈ r} ∧
        ∀x.
            RRESTRICT f r ' x =
            if x ∈ FDOM f ∧ f ' x ∈ r then f ' x else FEMPTY ' x
DRESTRICT_FUPDATE
  ⊢ ∀f r x y.
        DRESTRICT (f |+ (x,y)) r =
        if x ∈ r then DRESTRICT f r |+ (x,y) else DRESTRICT f r
RRESTRICT_FUPDATE
  ⊢ ∀f r x y.
        RRESTRICT (f |+ (x,y)) r =
        if y ∈ 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. 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) 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 (f ⊎ g) = FDOM f ∪ FDOM g ∧
        ∀x. (f ⊎ g) ' x = if x ∈ FDOM f then f ' x else g ' x
SUBMAP_DEF
  ⊢ ∀f g. f ⊑ g ⟺ ∀x. x ∈ FDOM f ⇒ x ∈ 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 (FLOOKUP) 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).

```
FLOOKUP_DEF
  ⊢ ∀f x. FLOOKUP f x = if x ∈ FDOM f then SOME (f ' x) else NONE

FUN_FMAP_DEF
  ⊢ ∀f P.
         FINITE P ⇒
         FDOM (FUN_FMAP f P) = P ∧ ∀x. x ∈ 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 ∩ {x | g ' x ∈ FDOM f} ∧
         ∀x. x ∈ 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 ∈ FDOM (f o_f g) ⇒ (f o_f g) ' x = f (g ' x)
f_o_DEF
  ⊢ ∀f g. f f_o g = f f_o_f FUN_FMAP g {x | g x ∈ FDOM f}
```

5.6 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 $\lambda x.\ \mathsf{T}$, 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, [12] which uses the expressive power of HOL to surprising effect. Consider the following total and non-recursive function:

[12] The original idea is due to J Moore, who suggested it for use in ACL2.

```
\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.

5.7 Monads

HOL's simple type system means that it is impossible to define a general type of monad in the way that is possible in programming languages such as Haskell. Nonetheless, a number of the types predefined in HOL, such as options and lists, can indeed be seen as monads, and it is useful to be able to write functions over those types that leverage this view. Equally, it is useful to be able to declare monads of one's own that can use the same syntactic facilities.

Monads are defined by their "unit" and "bind" functions, and these can be composed in expressive ways. In particular, HOL supports a syntax inspired by Haskell's do notation, wherein it is possible to write such functions as

4

```
> val mapM_def = Define‘
    (mapM f [] = return []) ∧
    (mapM f (x::xs) = do
       e  <- f x;
       es <- mapM f xs;
       return (e::es);
     od)’;
<<HOL message: inventing new type variable names: ’a, ’b>>
Definition has been stored under "mapM_def"
val mapM_def =
   ⊢ (∀f. mapM f [] = [[]]) ∧
     ∀f x xs. mapM f (x::xs) = do e <- f x; es <- mapM f xs; [e::es] od: thm

> type_of “mapM”;
<<HOL message: inventing new type variable names: ’a, ’b>>
val it = “:(α -> β list) -> α list -> β list list”: hol_type
```

Again, because HOL does not have a sufficiently expressive type system, though the notation is generic, the function is fixed to a particular monad instance. In this case, the monad instance is that of lists. We can use the `mapM` function to implement what one might term a cross-product operation on lists:

5

```
> EVAL “mapM I [[1;2;3]; [a;b]; [x;y;z]]”;
val it =
   ⊢ mapM I [[1; 2; 3]; [a; b]; [x; y; z]] =
     [[1; a; x]; [1; a; y]; [1; a; z]; [1; b; x]; [1; b; y]; [1; b; z];
      [2; a; x]; [2; a; y]; [2; a; z]; [2; b; x]; [2; b; y]; [2; b; z];
      [3; a; x]; [3; a; y]; [3; a; z]; [3; b; x]; [3; b; y]; [3; b; z]]: thm
```

The general abstract syntax is described by the following grammar

$$
\begin{array}{rcl}
M & ::= & e_{\alpha\,\mathtt{M}} \;\mid\; \mathtt{do}\ binds\ \mathtt{od} \;\mid\; \mathtt{return}\ e_{\alpha} \\
binds & ::= & bind\ ;^{?} \;\mid\; bind\ ;\ binds \\
bind & ::= & M \;\mid\; vs\ \mathtt{<-}\ M
\end{array}
$$

where e_τ is a HOL expression required to be of type τ, and vs is a single variable, or a tuple of variables (*e.g.*, `(x,y,z)`). If a given M has type $:\alpha$ `M` for a monad instance `M`, then writing a binding such as v `<-` M will require variable v to have type α. This variable is then bound in later bindings within the same `do-od` block.

The special monad syntax has a straightforward translation into underlying HOL terms. The `do-od` delimiters have no semantic effect; they can be viewed as a special form of parenthesis that identifies where the binding syntax is going to be used.[13] Subsequently, the translation from vs `<-`M_1`;` M_2 is to `monad_bind` M_1 ($\lambda vs.\ M_2$), making it clear that vs can be used (and is bound) in M_2. If there is no variable-arrow on the first binding, then M_1`;` M_2 translates to something equivalent to `monad_bind` M_1 (`K` M_2), where `K` is the K-combinator from `combinTheory` (see Section 5.2.2). Finally, the `return` keyword is an overloading for the monad instance's unit function (which will have type $\alpha \to \alpha$ `M`).

5.7.1 Declaring monads

Monad instances have to be declared if the system is to support their parsing and pretty-printing. The function responsible is `declare_monad` in the `monadsyntax` module. (There is also an accompanying `temp_declare_monad` function that achieves the same thing dynamically but doesn't cause the declaration to persist for the benefit of other later descendant theories.)

```
type monadinfo =
   {bind: term,
     choice: term option,
     fail: term option,
     guard: term option, ignorebind: term option, unit: term}
val declare_monad = fn: string * monadsyntax.monadinfo -> unit
```

The terms for the `bind` and `unit` fields are the terms implementing the corresponding monad functions. For example, the bind function for the option function is `OPTION_BIND`, characterised by its defining theorem:

```
OPTION_BIND_def
  ⊢ (∀f. OPTION_BIND NONE f = NONE) ∧ ∀x f. OPTION_BIND (SOME x) f = f x
```

The unit function is just the `SOME` constructor for the type.

All of the other fields in the `monadinfo` record can be left unspecified. The `ignorebind` field is used to encode the situation where the user writes

```
   do m1; m2 od
```

[13] Indeed, writing `do` M `od` for a single binding form M will see the system print back M on its own.

meaning that though m1 may return a value, the remainder of the function does not use that value. As above, this can be handled through the use of the K combinator, which is what is done if the ignorebind field is set to NONE. However, if desired, one can specify a specific term to be used in this situation. For example, the system's encoding of the option monad uses a separate constant with exactly the definition one would expect:

```
OPTION_IGNORE_BIND_def
  ⊢ ∀m1 m2. OPTION_IGNORE_BIND m1 m2 = OPTION_BIND m1 (K m2)
```

The three remaining fields (guard, fail and choice) are relevant for monads with errors or failure modes. The underlying property is that if m_1 is an error value, then monad_bind m_1 f = m_1, meaning that attempting to sequence computations after an error just causes the error to be the result, without any use of the f value. In this way, one might see m_1 as a computation that has thrown some sort of exception.

If one specifies a fail term in declaring a monad, an overload is set up from the string fail to that term. This is flexible enough to allow parameterised errors: make the term be the function that takes a parameter and returns a monad error value. There is no monad-specific support required for this concept: the overload is sufficient.

When provided, the guard and choice terms are similarly established as overloads so that monadic code across different monads will look similar. If one views fail as throwing an exception, then the choice notation allows for catching an exception and trying another computation. The overload is to the infix ++ syntax, meaning that one can write M_1 ++ M_2 to represent the "choice" between M_1 and M_2. (Note that in the list monad, ++ is APPEND. This explains the choice of notation, though in the list monad, viewing "choice" as throwing and catching exceptions is actually harder to motivate.)

Finally, the guard term (if given) overloads to the term assert, which is expected to be defined such that

assert b = if b then return () else fail

A typical use of assert might be in a function such as

```
do
   list <- some_monad;
   assert(list <> []);
   return (HD list + 1)
od
```

where the assert ensures that the subsequent call to HD makes sense.

5.7.2 Enabling monad syntax

There are two steps to being able to use monadic syntax. Both steps *persist*, meaning that their effects are preserved for the benefit of descendant theories. Because of this

persistence, the function calls implementing these two steps should only be used in *x*`Script.sml` files. As with other parsing and pretty-printing functions, there are `temp_` versions of the functions. These do not cause persistence and can safely be used in other `.sml` files (such as library implementations).

The first step is to enable the generic monad syntax, by calling

```
monadsyntax.enable_monadsyntax : unit -> unit
```

After this, one can write `do...od` blocks, even though without any specific instances enabled the output will be unhelpful (the `monad_bind` printed in the session below is actually a variable):

6
```
> monadsyntax.enable_monadsyntax();
val it = (): unit
> “do x <- M1; M2 od”;
<<HOL message: inventing new type variable names: ’a, ’b, ’c, ’d>>
val it = “monad_bind M1 (λx. M2)”: term
```

One can see which monads have been declared with a call to `all_monads`:

7
```
> monadsyntax.all_monads()
val it =
   [("list",
     {bind = “LIST_BIND”, choice = SOME (“APPEND”), fail = SOME (“[]”),
      guard = SOME (“LIST_GUARD”), ignorebind = SOME (“LIST_IGNORE_BIND”),
      unit = “λx. [x]”}),
    ("option",
     {bind = “OPTION_BIND”, choice = SOME (“OPTION_CHOICE”), fail =
      SOME (“NONE”), guard = SOME (“OPTION_GUARD”), ignorebind =
      SOME (“OPTION_IGNORE_BIND”), unit = “SOME”})]:
   (string * monadsyntax.monadinfo) list
```

Particular monads can be enabled with calls to `enable_monad`. The most recently enabled is preferred when the context makes the choice ambiguous.

8
```
> List.app monadsyntax.enable_monad ["list", "option"];
val it = (): unit

> val t = “do x <- M; return (x + 1) od”;
<<HOL message: more than one resolution of overloading was possible>>
val t = “do x <- M; SOME (x + 1) od”: term
> type_of t;
val it = “:num option”: hol_type

> val t’ = “do x <- MAP f l; return (x + 1); od”;
<<HOL message: inventing new type variable names: ’a>>
val t’ = “do x <- MAP f l; [x + 1] od”: term
> type_of t’;
val it = “:num list”: hol_type
```

Thanks to the persistence features of these API points, loading fresh theories may cause more monads to be declared and/or enabled:

```
> load "errorStateMonadTheory";                                                    9
val it = (): unit
> monadsyntax.all_monads();
val it =
   [("errorState",
     {bind = “BIND”, choice = SOME (“ES_CHOICE”), fail = SOME (“ES_FAIL”),
      guard = SOME (“ES_GUARD”), ignorebind = SOME (“IGNORE_BIND”), unit =
      “UNIT”}),
    ("list",
     {bind = “monad_bind”, choice = SOME (“$++”), fail = SOME (“[]”), guard =
      SOME (“assert”), ignorebind = SOME (“monad_unitbind”), unit =
      “λx. [x]”}),
    ("option",
     {bind = “monad_bind”, choice = SOME (“$++”), fail = SOME (“NONE”),
      guard = SOME (“assert”), ignorebind = SOME (“monad_unitbind”), unit =
      “SOME”})]: (string * monadsyntax.monadinfo) list
```

Everything that has been enabled can in turn be disabled, with calls drawn from:

```
disable_monad            : string -> unit
temp_disable_monad       : string -> unit

disable_monadsyntax      : unit -> unit
temp_disable_monadsyntax : unit -> unit
```

5.7.3 Some built-in monad theories

See Figure 5.2 for the bind definitions for a number of different monads that are present in the core HOL set of theories.

5.8 Further Theories

Other theories of interest in HOL are listed and briefly described in Figure 5.3.

```
errorStateMonadTheory.BIND_DEF
  ⊢ ∀g f s0.
        errorStateMonad$BIND g f s0 =
        case g s0 of NONE => NONE | SOME (b,s) => f b s

listTheory.LIST_BIND_THM
  ⊢ LIST_BIND [] f = [] ∧
     LIST_BIND (h::t) f = APPEND (f h) (LIST_BIND t f)

optionTheory.OPTION_BIND_def
  ⊢ (∀f. OPTION_BIND NONE f = NONE) ∧
     ∀x f. OPTION_BIND (SOME x) f = f x

readerMonadTheory.BIND_def
  ⊢ ∀M f s. readerMonad$BIND M f s = f (M s) s

state_transformerTheory.BIND_DEF
  ⊢ ∀g f. state_transformer$BIND g f = UNCURRY f ∘ g
```

Figure 5.2: Monad bind definitions

`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 5.3: A selection of HOL theories

Chapter 6

Advanced Definition Principles

6.1 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 `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 `Datatype` accepts is found in Table 6.1.[1]

`Datatype` ‘[*binding* ;]* *binding*‘

binding	::=	*ident* = *constructor-spec*
	|	*ident* = *record-spec*
constructor-spec	::=	[*clause* |]* *clause*
clause	::=	*ident ty-spec**
ty-spec	::=	(*hol_type*) | *atomic-type*
record-spec	::=	<| [*ident* : *hol_type* ;]* *ident* : *hol_type* ;? |>

Table 6.1: Datatype Declaration. An *atomic-type* is a single token that denotes a *hol_type* (*e.g.*, `num`, `real`, or `'a`).

HOL maintains an underlying database of datatype facts called the `TypeBase`. This database is used to support various high-level proof tools (see Section 7.3), and is augmented whenever a `Datatype` declaration is made. When a datatype is defined by `Datatype`, the following information is derived and stored in the database.

- initiality theorem for the type

[1] HOL also supports another syntax for datatype definition through the `Hol_datatype` entrypoint. For more details on this syntax, see `Hol_datatype`'s entry in *REFERENCE*.

- 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
```

could be declared in HOL with a call to the `Datatype` function:

```
Datatype `btree = Leaf 'a | Node btree 'b btree`
```

Also: good practice in script files is to make sure that everything is an ML declaration, so the above should really appear as

```
val _ = Datatype `btree = Leaf 'a | Node btree 'b btree`
```

To reduce the verbiage of the above, there is a special syntax (akin to `Theorem` and `Definition`), allowing one to instead write

```
Datatype: btree = Leaf 'a | Node btree 'b btree
End
```

As with the other forms, the keywords here must be in column 1.

Note that in all forms, 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

```
Datatype `sum = C1 'left | C2 'right`;
```

and then writes `('a,'b)sum`, will the `'a` value be under the `C1` or `C2` constructor? The system chooses to make the arguments corresponding to variables appear in the order given by the dictionary ordering of the names of the variables occurring in the definition. Thus, in the example given, the `'a` of `('a,'b)sum` will be the argument to the `C1` constructor because `'left` comes before `'right` in the standard (ASCII) dictionary ordering.

6.1.1 Further examples

In the following, we shall give an overview of the kinds of types that may be defined by `Datatype`.

To start, enumerated types can be defined as in the following example:

```
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:

```
Datatype`foo = N num | B bool | Fn ('a -> 'b) | Pr ('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.

```
Datatype `tree = Leaf 'a | Node (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:

```
Datatype `lambda = Var string
                 | Const 'a
                 | Comb lambda lambda
                 | Abs lambda lambda`
```

The syntax for 'de Bruijn' terms is roughly similar:

```
Datatype `dB = Var string
             | Const 'a
             | Bound num
             | Comb dB dB
             | Abs 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.

```
Datatype `ntree = Node 'a (ntree list)`
```

A type of 'first order terms' can be declared as follows:

```
Datatype `term = Var string | Fnapp (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.

```
Datatype
  `atexp = var_exp string
         | let_exp dec exp ;

     exp = aexp    atexp
         | app_exp exp atexp
         | fn_exp  match ;

   match = match  rule
         | matchl rule match ;

    rule = rule pat exp ;

     dec = val_dec   valbind
         | local_dec dec dec
         | seq_dec   dec dec ;

 valbind = bind  pat exp
         | bindl pat exp valbind
         | rec_bind valbind ;

     pat = wild_pat
         | var_pat string`
```

Simple record types may be introduced using the <| ... |> notation.

```
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.

```
Datatype
  `file = Text string | Dir directory
     ;
   directory = <| owner : string ;
                  files : (string # file) list |>`
```

6.1.2 Type definitions that fail

Now we address some types that cannot be declared with Datatype. In some cases they cannot exist in HOL at all; in others, the type can be built in the HOL logic, but 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.

```
Datatype ‘foo = A foo‘
```

So called ‘nested types’, which are occasionally quite useful, cannot at present be built with Datatype:

```
Datatype ‘btree = Leaf ’a | Node ((’a # ’a) btree)‘
```

Types may not recurse on either side of function arrows. Recursion on the right is consistent (see the theory inftree), but Datatype is not capable of defining algebraic types that require it. Thus, examples such as the following will fail:

```
Datatype ‘flist = Nil | Cons ’a (’b -> flist)‘
```

Recursion on the left must fail for cardinality reasons. For example, HOL does not allow the following attempt to model the untyped lambda calculus (note the -> in the clause for the Abs constructor):

```
Datatype ‘lambda = Var string
                 | Const ’a
                 | Comb lambda lambda
                 | Abs (lambda -> lambda)‘
```

6.1.3 Theorems arising from a datatype definition

The consequences of an invocation of 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

```
Datatype `tree = Leaf num | Node tree tree`
```

results in the definitions

```
tree_case_def =
  |- (!a f f1 a. tree_CASE (Leaf a) f f1 = f a) /\
     !f f1 a0 a1. tree_CASE (Node a0 a1) f f1 = 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 case constant (here `tree_CASE`) allows pretty case expressions; see Section 6.4 below. 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))
        ==>
      (tree_CASE M f f1 = tree_CASE M' f' f1')
```

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.

6.2 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 `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:

```
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 7.5).

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 $field_i\ (field_j\texttt{\_fupd}\ f\ r)$. If $i = j$, then the RHS is $f\,(field_i(r))$, if not, it is $(field_i\ r)$. This theorem is installed in the `TypeBase`.

`_component_equality` A theorem stating that $(r_1 = r_2) \equiv \bigwedge_i (field_i(r_1) = field_i(r_2))$.

`_fupdfupds` A thereom stating that $field_i$`_fupd` f ($field_i$`_fupd` g r) = $field_i$`_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 $i < j$,

$$field_j\texttt{\_fupd}\ f\ (field_i\texttt{\_fupd}\ g\ r) = field_i\texttt{\_fupd}\ g\ (field_j\texttt{\_fupd}\ f\ r)$$

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 100, 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.

Better yet, users should define their own system of hierarchically nested sub-records if their records are getting too large.

The pretty-printing of big records can be controlled with the `pp_bigrecs` trace-flag.

6.3 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

$$|- \forall x\ y.\ R\ x\ y\ \Leftrightarrow\ (R\ x = R\ y),$$

or 2) a theorem that R is a nonempty partial equivalence relation, (see [HOQ] §5) of the form

$$|- (\exists x.\ R\ x\ x)\ \wedge\ (\forall x\ y.\ R\ x\ y\ \Leftrightarrow\ R\ x\ x \wedge R\ y\ y \wedge (R\ x = R\ y)).$$

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};
```

6.4 Case Expressions

Case constructs are an important feature of functional programming languages such as Standard ML. They provide a very compact and convenient notation for multi-way selection among the values of several expressions. HOL provides such a feature in the form of case expressions. Case expressions can simplify the expression of complicated branches between different cases or combinations of cases.

Pattern matching and case expressions are not directly supported by higher order logic. Thus, some effort is needed to support them in HOL. There are two implementations of case expressions in HOL. The parser supports case expressions, which it compiles into decision trees based on if-then-else expressions and case-constants as introduced by datatype definitions. The pretty-printer presents these complicated decision trees as nicely readable case expressions again. In addition, there is `patternMatchesLib`, which provides a formalisation of case expressions based on Hilbert's choice operator. Following the name of the main function, these are called PMATCH case expressions.

The benefit of decision-tree case expressions is that the whole semantic complexity is handled by the parser and pretty-printer. The resulting terms are simple. However, one needs to trust the complicated, lengthy case expression code in the parser and pretty-printer. Moreover, the internal structure might differ considerably from the input and is hard to predict. In some cases there can be a serious blow-up in size.

PMATCH case expressions are simple to parse and pretty-print. They support more features than decision-tree ones. There are guards, unbound variables in patterns and a wider variety of supported patterns in general. There is no size-blowup or surprising internal structure. Therefore, code generated from PMATCH case expressions tends to be better. However, this comes at the price of a much heavier machinery to reason about PMATCH case expressions.

6.4.1 Decision-tree case expressions

By default, HOL uses decision-tree case expressions. Let the non-terminal *term* stand for any HOL term and the non-terminal *cpat* represent any constructor pattern, i.e. any HOL term containing only literals, datatype constructors and variables. Then, the syntax of decision-tree case expressions is given by

term ::= `case` *term* `of` `|`$^{?}$ *cpat* => *term* (`|` *cpat* => *term*)*

The choice in the rule 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 to the syntax used by SML. If PMATCH case expressions have been enabled (see below), then decision-tree case expressions can still be generated by using the `dtcase` keyword instead of `case`.

Case expressions consider their list of pattern expressions in sequence to see if they match the test expression. Matching means that there is an assignment for the bound variables of that pattern expressions such that it equals the test expression. The first pattern which successfully matches causes its associated result expression to be evaluated with the matching variable assignment. The resulting value is yielded as the value of the entire case expression. If no pattern expression matches, the result of the case expression is `ARB`. Since decision-tree case expressions support only constructor patterns, it is guaranteed that if a pattern expression matches, the resulting variable assignment is uniquely determined.

A simple example of a case expression is

```
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. Internally though, it is compiled to nested “if-then-else” statements.

In addition to literals as patterns, as above, patterns may be constructor expressions. Many standard HOL types have constructors, including `num`, `list`, and `option`. A simple example using constructor patterns is (notice the optional bar in front of the first case).

```
case spouse(employee) of
  | NONE   => "single"
  | SOME s => "married to " ++ name_of s
```

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 6.5. In

```
> “case n of                                                              10
      0 => "none"
    | 1 => "one"
    | 2 => "two"
    | SUC m => "many"”;
Exception- HOL_ERR
  {message = "case expression mixes literals with non-literals.",
   origin_function = "mk_case", origin_structure = "Pmatch"} raised
```

Figure 6.1: Decision-tree case expression parsing going wrong

addition, case expressions may contain literals as patterns, either singly or as elements of deeply nested patterns.

Decision-tree 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 6.1, 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`. The session in Figure 6.1 demonstrates the way in which such an improper mixture is misinterpreted. 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 first example of Figure 6.2, where the function `parents` returns a pair containing the person's father and/or mother, where each is represented by `NONE` if deceased. This shows the nesting of option patterns within a pair pattern, and also the use of a wildcard `_` to match the cases not given.

Since decision-tree case expressions are compiled internally to a decision tree, the result of this compilation might look quite different from the input. Rows might be reordered or modified, there might be several new rows generated, and new variables or the `ARB` constant may also be introduced to properly represent the case expression.

```
case parents(john) of
   (NONE,NONE) => "orphan"
 | _ => "not an orphan"
```

```
case a of
    (1, y, z) => y + z
  | (x, 2, z) => x - z
  | (x, y, 3) => x * y
```

Figure 6.2: Two examples of case expressions

Moreover, the exact result depends on complicated heuristics to decide which case-split to perform next in the decision tree. For non-trivial case expressions the result can be hard to predict.

For example, consider the second case expression in Figure 6.2. Using the default heuristic, this compiles to a reasonably small decision tree that is pretty-printed well. The exact combination of equality tests with if-then-else and the case constant for products that this term produces can be seen in the session in Figure 6.3. However other heuristics result in more complicated terms as shown in the session in Figure 6.4.

```
> set_trace "pp_cases" 0;  ...output elided...                    11
> “case a of
      (1,y,z) => y + z
    | (x,2,z) => x - z
    | (x,y,3) => x * y”;
<<HOL message: mk_functional:
  pattern completion has added 1 clause to the original specification.>>
val it =
   “pair_CASE a
      (λv v1.
           pair_CASE v1
             (λy z.
                  literal_case
                    (λx.
                         if x = 1 then y + z
                         else
                           literal_case
                             (λy'.
                                  if y' = 2 then x − z
                                  else
                                    literal_case
                                      (λv10. if v10 = 3 then x * y' else ARB)
                                      z) y) v))”: term
```

Figure 6.3: The decision-tree structure underneath a case expression

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 6.5 on the definition of recursive functions.

```
> Pmatch.set_classic_heuristic ();  ...output elided...                    12
> “case a of
      (1,y,z) => y + z
    | (x,2,z) => x - z
    | (x,y,3) => x * y”;
<<HOL message: mk_functional:
  pattern completion has added 5 clauses to the original specification.>>
val it =
   “case a of
      (1,2,3) => 2 + 3
    | (1,2,v10) => 2 + v10
    | (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’,v24) => ARB”: term
> PmatchHeuristics.set_heuristic PmatchHeuristics.pheu_last_col;
...output elided...
> “case a of
      (1,y,z) => y + z
    | (x,2,z) => x - z
    | (x,y,3) => x * y”;
<<HOL message: mk_functional:
  pattern completion has added 2 clauses to the original specification.>>
val it =
   “case a of
      (1,y,z) => y + z
    | (x,2,3) => x − 3
    | (x,y’,3) => x * y’
    | (x,2,z’) => x − z’
    | (x,v13,z’) => ARB”: term
> Pmatch.set_default_heuristic ();  ...output elided...
> “case a of
      (1,y,z) => y + z
    | (x,2,z) => x - z
    | (x,y,3) => x * y”;
<<HOL message: mk_functional:
  pattern completion has added 1 clause to the original specification.>>
val it =
   “case a of
      (1,y,z) => y + z
    | (x,2,z) => x − z
    | (x,y’,3) => x * y’
    | (x,y’,v10) => ARB”: term
```

Figure 6.4: Effect of different pattern match heuristics

6.4.2 PMATCH case expressions

The library `patternMatchesLib` supports another form of case expression. After executing `patternMatchesLib.ENABLE_PMATCH_CASES()`, the case expression syntax introduced above is parsed into PMATCH-based terms.[2] If we turn off the library's pretty-printing, we can see how the example expression we used earlier is rendered:

13
```
> patternMatchesLib.ENABLE_PMATCH_CASES();  ...output elided...
> set_trace "use pmatch_pp" 0;  ...output elided...
> “case a of
     (1, y, z) => y + z
   | (x, 2, z) => x - z
   | (x, y, 3) => x * y”;
val it =
   “PMATCH a
      [PMATCH_ROW (λ(y,z). (1,y,z)) (λ(y,z). T) (λ(y,z). y + z);
       PMATCH_ROW (λ(x,z). (x,2,z)) (λ(x,z). T) (λ(x,z). x − z);
       PMATCH_ROW (λ(x,y). (x,y,3)) (λ(x,y). T) (λ(x,y). x * y)]”: term
```

One can see that in contrast to decision-tree case expressions the internal representation of PMATCH case expressions is very close to the input. No fancy parsing is required to turn the input into the internal representation, and conversely, printing (when enabled) can easily produce the syntax that the user chose as input.

PMATCH case expressions are more expressive than decision-tree based ones. The syntax allowed for decision-tree case expressions is a subset of the syntax of PMATCH ones. In addition, PMATCH case expressions support guards. Moreover, arbitrary terms instead of just ones using literals and constructors can be used as patterns. One has full control over the variables bound by patterns. Bound variables can even be used more than once in a pattern.

$$
\begin{array}{rcl}
\mathit{term} & ::= & \texttt{case}\ \mathit{term}\ \texttt{of}\ \texttt{|}^{?}\ \mathit{clause}\ (\texttt{|}\ \mathit{clause})^{*} \\
\mathit{clause} & ::= & \mathit{vardecl}^{?}\ \mathit{term}\ (\texttt{when}\ \mathit{term})^{?}\ \texttt{=>}\ \mathit{term} \\
\mathit{vardecl} & ::= & (\mathit{vars}^{?})\ \texttt{.|} \quad \texttt{|} \quad \mathit{vars}\ \texttt{.|} \\
\mathit{vars} & ::= & \mathit{var} \quad \texttt{|} \quad \mathit{var}\ \texttt{,}\ \mathit{vars}
\end{array}
$$

If a `when`-guard is omitted, it defaults to true (`T`). Omitting the declaration of variables bound by a pattern means that all variables used in the pattern are bound (as with the decision-tree syntax). Variables whose names start with an underscore are always bound, no matter whether they appear in the list of bound variables. This is convenient for using wildcard notations.

[2] After parsing PMATCH case expressions has been enabled, the keyword `dtcase` is used for parsing and pretty-printing decision-tree case expressions.

These extensions of the basic case expression syntax allow the expression of some interesting concepts using case expressions. One can for example express division with reminder using PMATCH case expressions:

```
!n c.
  0 < c ==>
  ((case n of (q,r) .| q * c + r when r < c => (q,r)) =
   (n DIV c,n MOD c))
```

Notice that

- the guard is used to make the match unique;
- listing the bound variables of the pattern `q * c + r` explicitly allows `c` to not be bound by the pattern; and
- the pattern (of form $qc+r$) would not be supported by decision-tree case expressions because it is not a constructor pattern

Another interesting option is to use bound variables multiple times. The following case expression states for example that a list `l` starts with two copies of the same element:

```
case l of | x::x::_ => T | _ => F
```

The price for this increased expressive power and the simpler, more trustworthy parsing and pretty-printing is that the semantic complexity now needs to be dealt with inside the logic. The `patternMatchesLib` library provides the necessary machinery in the form or conversions, rules, simpsets, and other technology. For example, `patternMatchesLib` enhances `bossLib`'s standard simpsets with the conversions to deal with PMATCH case expressions. Therefore, in practice PMATCH case expressions are as convenient to use as decision-tree based case expressions. In fact, the explicit case expression structure provided by PMATCH case expressions allows simplifications not easily possible for decision-tree case expressions. The standard tools can only (partially) evaluate decision-tree case expressions. This evaluation usually destroys the case expression view. This is illustrated by the example shown in Figure 6.5, which adds a row that is subsumed by a later one to our running example.

There are many more tools for working with PMATCH case expressions. There are, for example, tools for removing multiple variable binding or guards. One can translate between PMATCH and decision-tree case expressions. There are tools for proving exhaustiveness of PMATCH case expressions or for removing redundant rows. For more details, please consult the `patternMatchesLib` documentation in Section 7.8.

14

```
> SIMP_CONV (srw_ss()) []
    “case (x,y,z) of
       (1,y,z) => y + z
     | (x,2,4) => x - 4  (* subsumed by next row *)
     | (x,2,z) => x - z
     | (x,y,3) => x * y”;
val it =
   ⊢ (case (x,y,z) of
       (1,y,z) => y + z
     | (x,2,4) => x − 4
     | (x,2,z) => x − z
     | (x,y,3) => x * y) =
     case (x,y,z) of (1,y,z) => y + z | (x,2,z) => x − z | (x,y,3) => x * y:
   thm
> SIMP_CONV (srw_ss()) []
   “dtcase (x,y,z) of
      (1,y,z) => y + z
    | (x,2,4) => x - 4
    | (x,2,z) => x - z
    | (x,y,3) => x * y”;
<<HOL message: mk_functional:
  pattern completion has added 2 clauses to the original specification.>>
val it =
   ⊢ (dtcase (x,y,z) of
        (1,y,z) => y + z
      | (x,2,4) => x − 4
      | (x,2,3) => x − 3
      | (x,2,z') => x − z'
      | (x,y',3) => x * y'
      | (x,y',v14) => ARB) =
     if x = 1 then y + z
     else if y = 2 then
       if z = 4 then x − 4 else if z = 3 then x − 3 else x − z
     else if z = 3 then x * y
     else ARB: thm
```

Figure 6.5: Simplification of case expressions

6.5 Recursive Functions

HOL provides a function definition mechanism based on the wellfounded recursion theorem proved in `relationTheory`, discussed in Section 5.5.3. With the `Definition` syntax, users provide a high-level, possibly recursive, specification of a function, and HOL attempts to define the function in the logic. This technology may be used to define abbreviations, recursive functions, and mutually recursive functions. An induction theorem may be generated as a by-product of this activity. This induction theorem follows the recursion structure of the function, and may be useful when proving properties of the function. The definition technology 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. Once a termination argument is found, it can be provided as part of the final `Definition` syntax. The technology underlying `Definition` and `Hol_defn` is explained in detail in Slind [13].

The basic form of a `Definition` (without a termination argument) is

```
Definition defname:
  fn-spec
End
```

Note that this is *not* valid ML syntax. Instead, HOL is using lexical pre-processing to transform the above into something more complicated underneath. To make this reasonable, there are constraints introduced on the syntax above: both keywords (`Definition` and `End`) must appear in column 1 of the input (hard against the left margin in one's input source file). For example

```
> Definition last0_def:                                              15
    (last0 [] = 0) /\
    (last0 [x] = x) /\
    (last0 (h::t) = last0 t)
  End
Equations stored under "last0_def".
Induction stored under "last0_ind".
val last0_def =
   ⊢ last0 [] = 0 ∧ (∀x. last0 [x] = x) ∧
     ∀v3 v2 h. last0 (h::v2::v3) = last0 (v2::v3): thm
```

The *fn-spec* above is quoted syntax representing a conjunction of equations. The specified function(s) may be phrased using ML-style pattern-matching. The *fn-spec* should conform with the grammar in Table 6.2.

Pattern expansion In general, `Definition` attempts to derive exactly the specified conjunction of equations. However, the rich syntax of patterns allows some ambiguity. For example, the input

fn-spec	::=	*eqn*
	|	(*eqn*) ∧ *fn-spec*
eqn	::=	*alphanumeric pat* ... *pat* = *term*
pat	::=	*variable*
	|	*wildcard*
	|	*cname*
	|	($cname_n\ pat_1 \ldots pat_n$)
cname	::=	*alphanumeric* | *symbolic*
wildcard	::=	_
	|	_*wildcard*

Table 6.2: Syntax of Function Declaration

```
Definition f_def:
  (f 0 _ = 1) /\
  (f _ 0 = 2)
End
```

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, `Definition` 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, `Definition` 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 `Definition` 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 `Definition` fails. If it succeeds, then `Definition` 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 The name given in the use of the `Definition` syntax is used both to store the definition in the current theory and is also bound to the same theorem within the ML environment. If there is an associated induction theorem, its name is derived as follows:

- If the principal name ends with `_def` or with `_DEF`, then the induction theorem is the same as the principal name with the suffix replaced by `_ind` or `_IND` respectively.
- Otherwise, the induction theorem is the principal name with `_ind` added as a suffix.
- Finally, the user may override these choices by adding an `induction_thm=`*name* "attribute" to the principal name.

6.5.1 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.

```
> Definition fact_def:
    fact x = if x = 0 then 1 else x * fact(x-1)
  End
Equations stored under "fact_def".
Induction stored under "fact_ind".
val fact_def = ⊢ ∀x. 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.

```
> fact_ind; (* DB.fetch "-" "fact_ind" would also work *)
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 use the `induction=` "attribute" to require a non-standard name for the generated induction theorem.

```
> Definition flatten[induction=flat_ind]:
     (flatten  []           = [])
  /\ (flatten ([]::rst)     = flatten rst)
  /\ (flatten ((h::t)::rst) = h::flatten(t::rst))
  End
<<HOL message: inventing new type variable names: 'a>>
Equations stored under "flatten".
Induction stored under "flat_ind".
val flatten =
   ⊢ flatten [] = [] ∧ (∀rst. flatten ([]::rst) = flatten rst) ∧
     ∀t rst h. flatten ((h::t)::rst) = h::flatten (t::rst): thm
> flat_ind;
val it =
   ⊢ ∀P.
         P [] ∧ (∀rst. P rst ⇒ P ([]::rst)) ∧
         (∀h t rst. P (t::rst) ⇒ P ((h::t)::rst)) ⇒
         ∀v. P v: thm
```

Next we define a curried recursive function, which uses wildcard expansion and pattern-matching pre-processing.

```
> Definition min_def: (min (SUC x) (SUC y) = min x y + 1)
                 /\   (min  ____    ____   = 0)
  End
<<HOL message: mk_functional:
  pattern completion has added 1 clause to the original specification.>>
<<HOL warning: GrammarDeltas.revise_data:
  Grammar-deltas:
    overload_on("min_def_tupled")
  invalidated by DelConstant(scratch$min_def_tupled)>>
Equations stored under "min_def".
Induction stored under "min_ind".
val min_def =
   ⊢ (∀y x. min (SUC x) (SUC y) = min x y + 1) ∧ (∀v1. min 0 v1 = 0) ∧
     ∀v3. min (SUC v3) 0 = 0: thm
```

Next we make a primitive recursive definition. Note that no induction theorem is generated in this case.

```
> Definition filter:
    (filter P [] = [])
    /\    (filter P (h::t) = if P h then h::filter P t
                             else filter P t)
  End
<<HOL message: inventing new type variable names: 'a>>
Definition has been stored under "filter"
val filter =
   ⊢ (∀P. filter P [] = []) ∧
     ∀P h t. filter P (h::t) = if P h then h::filter P t else filter P t: thm
```

`Definition` 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:

```
> Datatype`prop = VAR 'a | NOT prop | AND prop prop
                | OR  prop prop';
<<HOL message: Defined type: "prop">>
val it = (): unit
```

Then two mutually recursive functions `nnfpos` and `nnfneg` are defined:

```
> Definition nnfpos_def:
     (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))
  End
<<HOL message: inventing new type variable names: 'a>>
Equations stored under "nnfpos_def".
Induction stored under "nnfpos_ind".
val nnfpos_def =
   ⊢ (∀x. nnfpos (VAR x) = VAR x) ∧ (∀p. nnfpos (NOT p) = nnfneg p) ∧
     (∀q p. nnfpos (AND p q) = AND (nnfpos p) (nnfpos q)) ∧
     (∀q p. nnfpos (OR p q) = OR (nnfpos p) (nnfpos q)) ∧
     (∀x. nnfneg (VAR x) = NOT (VAR x)) ∧ (∀p. nnfneg (NOT p) = nnfpos p) ∧
     (∀q p. nnfneg (AND p q) = OR (nnfneg p) (nnfneg q)) ∧
     ∀q p. nnfneg (OR p q) = AND (nnfneg p) (nnfneg q): thm
```

`Definition` may also be used to define non-recursive functions:

```
> Definition f: f x (y,z) = (x + 1 = y DIV z)
  End
Definition has been stored under "f"
val f = ⊢ ∀x y z. f x (y,z) ⟺ x + 1 = y DIV z: thm
```

Finally, `Definition` 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. It may also return theorems that do not look quite like what was originally input. For more on this, see the discussion of case expressions (into which the clauses in a `Definition` are converted internally) in Section 6.4.

6.5.2 When termination is not automatically proved

If the termination proof for a prospective definition fails, the invocation of the definitional machinery fails. In such situations, the user must supply a termination argument explicitly. The primitive machinery for beginning this operation is the ML function `Hol_defn`:

```
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_defn =
    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))`;
<<HOL message: inventing new type variable names: 'a>>
val qsort_defn =
   HOL function definition (recursive)

   Equation(s) :
    [...] ⊢ qsort ord [] = []
    [...]
   ⊢ qsort ord (h::t) =
     qsort ord (FILTER (λx. ord x h) t) ++ [h] ++
     qsort ord (FILTER (λx. ¬ord x h) t)

   Induction :
    [...]
   ⊢ ∀P.
         (∀ord. P ord []) ∧
         (∀ord h t.
              P ord (FILTER (λx. ¬ord x h) t) ∧
              P ord (FILTER (λx. ord x h) t) ⇒
              P ord (h::t)) ⇒
         ∀v v1. P v v1

   Termination conditions :
     0. ∀t h ord. R (ord,FILTER (λx. ¬ord x h) t) (ord,h::t)
     1. ∀t h ord. R (ord,FILTER (λx. ord x h) t) (ord,h::t)
     2. WF R: defn
```

which returns a value of type `defn`, but does not try to prove termination.

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.

2
```
> Defn.tgoal qsort_defn;
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        ∃R.
            WF R ∧ (∀t h ord. R (ord,FILTER (λx. ¬ord x h) t) (ord,h::t)) ∧
            ∀t h ord. R (ord,FILTER (λx. ord x h) t) (ord,h::t)
```

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.

```
> e (WF_REL_TAC `measure (LENGTH o SND)`);                                    3
OK..
1 subgoal:
val it =

   (∀t h ord. LENGTH (FILTER (λx. ¬ord x h) t) < LENGTH (h::t)) ∧
   ∀t h ord. LENGTH (FILTER (λx. 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/tactic into the Termination section of a Definition. As long as the tactic does indeed prove the termination conditions, the recursion equations and induction theorem are stored in the current theory segment before the recursion equations are returned:

```
> Definition qsort_def:                                                       4
    (qsort ord [] = []) /\
    (qsort ord (h::t) =
         qsort ord (FILTER (\x. ord x h) t) ++ [h] ++
         qsort ord (FILTER (\x. ~(ord x h)) t))
  Termination
    WF_REL_TAC `measure (LENGTH o SND)` THEN cheat
  End
<<HOL message: inventing new type variable names: 'a>>
Equations stored under "qsort_def".
Induction stored under "qsort_ind".
val qsort_def =
   ⊢ (∀ord. qsort ord [] = []) ∧
     ∀t ord h.
         qsort ord (h::t) =
         qsort ord (FILTER (λx. ord x h) t) ++ [h] ++
         qsort ord (FILTER (λx. ¬ord x h) t): thm
```

As we have not specified a special name for it, the custom induction theorem for our function can be obtained under the name qsort_ind, or by using fetch, which returns named elements in the specified theory.[3]

[3] In a call to fetch, the first argument denotes a theory; the current theory may be specified by "-".

```
> DB.fetch "-" "qsort_ind";                                                    5
val it =
   ⊢ ∀P.
         (∀ord. P ord []) ∧
         (∀ord h t.
              P ord (FILTER (λx. ¬ord x h) t) ∧
              P ord (FILTER (λx. ord x h) t) ⇒
              P ord (h::t)) ⇒
         ∀v v1. P v v1: thm
```

The induction theorem produced by `Definition` can be applied by `recInduct`. See Section 7.3 for details.

6.5.2.1 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:

1

```
> val gcd_defn =
     Hol_defn "gcd"
        `(gcd (0,n) = n) /\
         (gcd (m,n) = gcd (n MOD m, m))`;
val gcd_defn =
   HOL function definition (recursive)

   Equation(s) :
    [..] ⊢ gcd (0,n) = n
    [..] ⊢ gcd (SUC v2,n) = gcd (n MOD SUC v2,SUC v2)

   Induction :
    [..]
   ⊢ ∀P.
         (∀n. P (0,n)) ∧ (∀v2 n. P (n MOD SUC v2,SUC v2) ⇒ P (SUC v2,n)) ⇒
         ∀v v1. P (v,v1)

   Termination conditions :
     0. ∀v2 n. R (n MOD SUC v2,SUC v2) (SUC v2,n)
     1. WF R: defn

> Defn.tgoal gcd_defn;
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        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 `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

```
prim_recTheory.measure_thm
  ⊢ ∀f x y. 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..
val it =
   Initial goal proved.
   ⊢ (gcd (0,n) = n ∧ gcd (SUC v2,n) = gcd (n MOD SUC v2,SUC v2)) ∧
     ∀P.
         (∀n. P (0,n)) ∧ (∀v2 n. P (n MOD SUC v2,SUC v2) ⇒ P (SUC v2,n)) ⇒
         ∀v v1. P (v,v1): proof
```

The built-in reasoning then suffices to prove the remaining goal.

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
```
> Datatype `btree = Leaf | Brh btree btree`;  ...output elided...

> 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)))`;
val Unbal_defn =
   HOL function definition (recursive)

   Equation(s) :
    [...] ⊢ Unbal Leaf = Leaf
    [...] ⊢ Unbal (Brh Leaf bt) = Brh Leaf (Unbal bt)
    [...] ⊢ Unbal (Brh (Brh bt1 bt2) bt) = Unbal (Brh bt1 (Brh bt2 bt))

   Induction :
    [...]
   ⊢ ∀P.
         P Leaf ∧ (∀bt. P bt ⇒ P (Brh Leaf bt)) ∧
         (∀bt1 bt2 bt. P (Brh bt1 (Brh bt2 bt)) ⇒ P (Brh (Brh bt1 bt2) bt)) ⇒
         ∀v. P v

   Termination conditions :
     0. ∀bt bt2 bt1. R (Brh bt1 (Brh bt2 bt)) (Brh (Brh bt1 bt2) bt)
     1. ∀bt. R bt (Brh Leaf bt)
     2. WF R: defn
```

The termination conditions above can be turned into an interactive goal with `Defn.tgoal`:

2
```
> Defn.tgoal Unbal_defn;
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        ∃R.
            WF R ∧
            (∀bt bt2 bt1. R (Brh bt1 (Brh bt2 bt)) (Brh (Brh bt1 bt2) bt)) ∧
            ∀bt. R bt (Brh Leaf 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

```
Definition Weight_def:                                                    3
   (Weight (Leaf) = 0) /\
   (Weight (Brh x y) = (2 * Weight x) + (Weight y) + 1)
End
```

Now we can invoke WF_REL_TAC:

```
> e (WF_REL_TAC `measure Weight`);                                        4
OK..
1 subgoal:
val it =

   (∀bt bt2 bt1.
        Weight (Brh bt1 (Brh bt2 bt)) < Weight (Brh (Brh bt1 bt2) bt)) ∧
   ∀bt. Weight bt < Weight (Brh Leaf bt)
```

Both conjuncts of this goal 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))`;  ...output elided...

> Definition Match_def:  Match pat str = match pat str pat str
  End
<<HOL message: inventing new type variable names: 'a>>
Definition has been stored under "Match_def"
val Match_def = ⊢ ∀pat str. Match pat str ⟺ match pat str pat str: thm
```

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:

```
pairTheory.LEX_DEF
  ⊢ ∀R1 R2. R1 LEX R2 = (λ(s,t) (u,v). R1 s u ∨ s = u ∧ R2 t v)
```

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:

```
relationTheory.inv_image_def
  ⊢ ∀R f. inv_image R f = (λx 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;  ...output elided...

> e (WF_REL_TAC `inv_image($< LEX $<) (\(w,x,y,z). (LENGTH z,LENGTH x))`);
OK..
1 subgoal:
val it =

   (∀ss s b a.
        a ≠ b ∧ ¬NULL s ⇒
        LENGTH (TL s) < LENGTH s ∨
        LENGTH (TL s) = LENGTH s ∧ LENGTH (TL s) < LENGTH (b::ss)) ∧
   ∀ss a. LENGTH ss < LENGTH (a::ss)
```

The first conjunct needs a case-split on s before it is proved by rewriting, and the second is also easy to prove by rewriting.

6.5.2.2 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 descendant 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
```
> Definition SIGMA_def:
   (SIGMA f [] = 0) /\
   (SIGMA f (h::t) = f h + SIGMA f t)
  End  ...output elided...
```

We then use `SIGMA` in the definition of a function for summing the results of a function in a arbitrarily (finitely) branching tree.

2
```
  > Datatype `ltree = Node 'a (ltree list)`;  ...output elided...

  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:

3
```
<<HOL message: inventing new type variable names: 'a>>
val it =
   HOL function definition (recursive)

   Equation(s) :
    [..] ⊢ ltree_sigma f (Node v tl) = f v + SIGMA (λa. 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. ∀tl v f a. R (f,a) (f,Node v tl)
     1. WF R: 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 `ltree`s. 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, or by using a 'defncong' attribute on a theorem when it is saved,the termination conditions extracted for the definition are now provable, since `a` is a proper subterm of `Node v tl`.

4
```
> val _ = DefnBase.add_cong SIGMA_CONG;
> Defn.Hol_defn "ltree_sigma"
   `ltree_sigma f (Node v tl) = f v + SIGMA (ltree_sigma f) tl`;
<<HOL message: inventing new type variable names: 'a>>
val it =
   HOL function definition (recursive)

   Equation(s) :
    [..] ⊢ ltree_sigma f (Node v tl) = f v + SIGMA (λa. ltree_sigma f a) tl

   Induction :
    [..]
   ⊢ ∀P. (∀f v tl. (∀a. MEM a tl ⇒ P f a) ⇒ P f (Node v tl)) ⇒ ∀v v1. P v v1

   Termination conditions :
     0. ∀v f tl a. MEM a tl ⇒ R (f,a) (f,Node v tl)
     1. WF R: defn
```

6.5.3 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`, which can also be targeted by using the `schematic` attribute with the `Definition` syntax. 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
```
> Definition linRec_def[schematic]:
    linRec (x:'a) = if d(x) then e(x) else f(linRec(g x))
  End
<<HOL message: inventing new type variable names: 'b>>
<<HOL message: Definition is schematic in the following variables:
    "d", "e", "f", "g">>
<<HOL message: Unable to add linRec_def to global compset>>
Equations stored under "linRec_def".
Induction stored under "linRec_ind".
val linRec_def =
    [..]
   ⊢ ∀x f e. linRec d e f g x = if d x then e x else f (linRec d e f g (g x)):
   thm
```

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 linRec_def;
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.

6.6 Inductive Relations

Inductive definitions are made with the special `Inductive` syntax form, which wraps the underlying function `Hol_reln`, which is in turn found in the `bossLib` structure. The resulting definitions and theorems are handled with functions defined in the library `IndDefLib`. The `Inductive` syntax and `Hol_reln` function take a term quotation as input and attempt to define the relations there specified. The input term quotation must parse to a term that conforms to the following grammar:

$$
\begin{array}{rcl}
\langle inputFormat\rangle & ::= & \langle clause\rangle \ \texttt{/\backslash}\ \langle inputFormat\rangle \ \mid \ \langle clause\rangle \\
\langle clause\rangle & ::= & (\texttt{!}x_1 \ldots x_n.\ \langle hypothesis\rangle \ \texttt{==>}\ \langle conclusion\rangle) \\
& \mid & (\texttt{!}x_1 \ldots x_n.\ \langle conclusion\rangle) \\
\langle conclusion\rangle & ::= & \langle con\rangle\ sv_1\ sv_2 \ldots \\
\langle hypothesis\rangle & ::= & \text{any term} \\
\langle con\rangle & ::= & \text{a new relation constant}
\end{array}
$$

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 this definitional principle returns three important theorems *rules*, *ind* and *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 (if using `Hol_reln`), or as provided by the user, when using the `Inductive` syntax.
- *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 6.5).

Alternatively, the `Inductive` syntax can be used, requiring the user to specify the stem, but saving on verbosity: instead of writing

```
val (foo_rules,foo_ind,foo_cases) = Hol_reln‘
  ...
‘;
```

one writes

```
Inductive foo:
  ...
End
```

where, as with other special syntaxes, the keywords (`Inductive` and `End`) have to be in the leftmost column of the source file.

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 an inductive definition is made. 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 5.4.1), 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:

1
```
> Inductive EVEN_ODD:
    EVEN 0 /\
    (!n. ODD n ==> EVEN (n + 1)) /\
    (!n. EVEN n ==> ODD (n + 1))
  End
val EVEN_ODD_cases =
   ⊢ (∀a0. EVEN a0 ⟺ a0 = 0 ∨ ∃n. a0 = n + 1 ∧ ODD n) ∧
     ∀a1. ODD a1 ⟺ ∃n. a1 = n + 1 ∧ EVEN n: thm
val EVEN_ODD_ind =
   ⊢ ∀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: thm
val EVEN_ODD_rules =
   ⊢ EVEN 0 ∧ (∀n. ODD n ⇒ EVEN (n + 1)) ∧ ∀n. EVEN n ⇒ ODD (n + 1): thm
```

The next example shows how to inductively define the reflexive and transitive closure of relation R, which we write as `rtc`. 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.

```
> Inductive rtc:                                                    2
    (!x. rtc R x x) /\
    (!x z. (?y. R x y /\ rtc R y z) ==> rtc R x z)
  End
<<HOL message: inventing new type variable names: 'a>>
<<HOL message: Treating "R" as schematic variable>>
val rtc_cases = ⊢ ∀R a0 a1. rtc R a0 a1 ⟺ a1 = a0 ∨ ∃y. R a0 y ∧ rtc R y a1:
   thm
val rtc_ind =
   ⊢ ∀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: thm
val rtc_rules =
   ⊢ ∀R. (∀x. rtc R x x) ∧ ∀x z. (∃y. R x y ∧ rtc R y z) ⇒ rtc R x z: thm
```

Inductive definitions 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.

6.6.1 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, which will actually use the automatically generated "strong" induction principle.[4] (This tactic is also used to perform structural inductions over algebraic data types; see Section 7.3.) When performing a rule induction, the quotation passed to `Induct_on` should be of the constant, perhaps also applied to arguments. Indeed, if there are multiple instances of an `R`-term in the goal, then quoting arguments allows selection of the correct term to induct on. Thus, one can write invocations such as

```
Induct_on `R (f x) y`
```

[4]To get an approximation of the `Induct_on` call, one might write something like `HO_MATCH_MP_TAC R_strongind`.

Chapter 7

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.

7.1 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 prettyprinting 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.

7.1.1 Parsing types

The language of types is a simple one. An abstract grammar for the language is presented in Figure 7.1. The actual grammar (with concrete values for the infix symbols and type operators) can be inspected using the function `type_grammar`.

τ	::=	$\tau \odot \tau$ | *vtype* | *tyop* | (*tylist*) *tyop* | τ *tyop* | (τ) | τ[τ]
$\odot$	::=	-> | # | + | ⋯
vtype	::=	'a | 'b | 'c | ⋯
tylist	::=	τ | τ , *tylist*
tyop	::=	bool | list | num | fun | ⋯

Figure 7.1: An abstract grammar for HOL types (τ). Infixes ($\odot$) always bind more weakly than type operators (*tyop*) (and type-subscripting (τ[τ])), so that $\tau_1 \odot \tau_2$ *tyop* is always parsed as $\tau_1 \odot (\tau_2$ *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 5.5.1, 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 if the `type_abbrev_pp` entry-point is used; otherwise the abbreviation is for input only. For more information, see `type_abbrev`'s entry in *REFERENCE*.

There are special syntactic forms available for both type abbreviation entry-points. Instead of

```
val _ = type_abbrev("set", ``:'a -> bool``)
```

one can write

```
Type set = ``:'a -> bool``
```

and if an underlying call to `type_abbrev_pp` is desired, the `[pp]` "attribute" should be added to the name, thus:

```
Type set[pp] = ``:'a -> bool``
```

7.1.2 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 1.8.3.1 and 1.8.3.2.) 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.

7.1.2.1 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 1.3.1.

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 7.1.2.7 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 7.1.3); `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 7.1.2.4). At this stage of the translation, there is no distinction made between 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")
```

[1]The types in `Absyn` constraints are not full HOL types, but values from another intermediate type, `Pretype`.

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 7.1.2.3 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 users do not have to provide types for all of the identifiers they type. (See Section 7.1.2.5 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.

7.1.2.2 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 a provide a pure ASCII syntax, they can do so by defining that pure ASCII syntax first. Having done this, they can create a Unicode version of the syntax with the function `Unicode.unicode_version`. Then, either Unicode or ASCII characters can be used to input the syntax, and, while the trace variable `"PP.avoid_unicode"` is 1, the ASCII syntax will be used for printing. If the trace is set to 0, then again, both syntaxes can be used to *write* the terms, but the pretty-printer will prefer the Unicode syntax when the terms are printed by the system.

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.)

7.1.2.3 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 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.

[2] The matching done is first-order; contrast the higher-order matching done in the simplifier.

7.1.2.4 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
```

7.1.2.5 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.

7.1.2.6 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.

1
```
> load "integerTheory";
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`.

2
```
> show_types := true;
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.

7.1.2.7 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*)

[3]When `pairTheory` has been loaded.

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 $(f\ x)\ y$

but

`~` $x\ y$ is parsed as `~` $(x\ y)$

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 5.5.1) 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 suffixes `^+`, `^*` and `^=` corresponding to the transitive, the reflexive and transitive, and the "equivalence" closure used in `relationTheory` (Section 5.5.3). 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`

[4] There are tighter infix operators: the dot in field selection causes $f\ x.fld$ to parse as $f \cdot (x.fld)$.

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.

7.1.2.8 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]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.

7.1.2.9 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* 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 7.1.2.1. 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.

7.1.2.10 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

```
> arithmeticTheory.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
> arithmeticTheory.ADD_CLAUSES;
val it = ⊢ ... + ... = m ∧ ... = ... ∧ ... ∧ ...: thm
```

7.1.3 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 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` 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 = “:α -> 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:

3
```
> ``f ^` x ``;
<<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
```

Finally, if a single caret is followed by a “symbol” character, then the caret and symbol are passed through to HOL unchanged. Thus the following example illustrates two different ways of writing the same thing (in the first input, two carets become one):

4
```
> ``f ^^+ x``;
<<HOL message: inventing new type variable names: 'a>>
val it = “f+ x”: term

> ``f ^+ x``;
<<HOL message: inventing new type variable names: 'a>>
val it = “f+ x”: term
```

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 = “:α -> bool”: hol_type

> ``:^pred -> bool``;
val it = “:(α -> 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,

3
```
> ``!P:^pred. P x ==> Q x``;
Exception- Type error in function application.
   Function: Parse.Term : term frag list -> term
   Argument:
      [
         QUOTE " (*#loc 1 4*)!P:",
         ANTIQUOTE (pred),
         QUOTE " (*#loc 1 12*). P x ==> Q x"
         ] : hol_type frag list
   Reason:
      Can't unify term (*Created from opaque signature*) with
         hol_type (*Created from opaque signature*)
         (Different type constructors)
Fail "Static Errors" raised

> ``!P:^(ty_antiq pred). P x ==> Q x``;
val it = "∀P. P x ⇒ Q x": term
```

7.1.4 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.

7.2 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.

7.2.1 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.

7.2.2 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.

7.2.3 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`.

7.2.4 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`.

7.2.5 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.

7.3 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.

7.3.1 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

[6]Complete induction and ordinary mathematical induction are each derivable from the other.

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 appropriate.

`recInduct` takes a induction theorem generated by `Define` or `Hol_defn` and applies it to the current goal.

7.3.2 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 7.4.1 below. The `METIS` system is described in Section 7.4.2. 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

[7] In the mid 1980's Graham Birtwistle advocated such an approach, calling it 'Ten Tactic HOL'.

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.

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 7.5 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.

7.4 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

[8]Proofs in the Mizar system are readable documents, unlike most tactic-based proofs.

not transform a goal into a different (and more helpful) form.

7.4.1 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.

7.4.2 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.

7.5 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:

1
```
> SIMP_CONV bool_ss [] ``p /\ T \/ ~(q /\ r)``;
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:

2
```
> SIMP_CONV bool_ss [] ``?x. (\y. P (f y)) x /\ (x = z)``;
<<HOL message: inventing new type variable names: 'a, 'b>>
val it = ⊢ (∃x. (λy. 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:

```
> open arithmeticTheory;  ...output elided...                    3
> SIMP_CONV bossLib.arith_ss [LEFT_ADD_DISTRIB] ``p * (n + 1)``;
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.

7.5.1 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`.

7.5.1.1 `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`.

7.5.1.2 `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:

```
OK..                                                           4
1 subgoal:
val it =

    0.  x = 3
   ------------------------------------
        P x

> e (ASM_SIMP_TAC bool_ss []);
OK..
1 subgoal:
val it =

    0.  x = 3
   ------------------------------------
        P 3
```

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`.

7.5.1.3 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:

5

```
OK..
1 subgoal:
val it =

    0.  f x < 10
    1.  x = 4
   ------------------------------------
        4 + x < 10

> e (FULL_SIMP_TAC bool_ss []);
OK..
1 subgoal:
val it =

    0.  f 4 < 10
    1.  x = 4
   ------------------------------------
        4 + 4 < 10
```

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 demonstrates more interesting behaviour:

```
OK..                                                                    6
1 subgoal:
val it =

    0.  x ≤ 4
   ------------------------------------
        f x + 1 < 10

> e (FULL_SIMP_TAC bool_ss [arithmeticTheory.LESS_OR_EQ]);
OK..
2 subgoals:
val it =

    0.  x = 4
   ------------------------------------
        f 4 + 1 < 10

    0.  x < 4
   ------------------------------------
        f x + 1 < 10
```

In this example, the goal was rewritten with the theorem stating

$$\vdash x \leq y \iff x < y \vee x = y$$

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.

7.5.1.4 `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.

7.5.1.5 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 7.5.2.5). 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 7.5.3.

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.

7.5.1.6 Abbreviated "Power" tactics

HOL's bossLib module (part of the standard interactive environment) comes with a number of powerful simplification tactics with short lower-case names. These do not require or allow for the specification of a particular simpset because they use srw_ss() combined with the arithmetic decision procedure for the natural numbers, as well as a simpset fragment that eliminates let-terms. All of the tactics take a list of rewrite theorems as their one argument. These tactics are:

simp	Corresponds to ASM_SIMP_TAC
rw	Corresponds to SRW_TAC[]
fs	Corresponds to FULL_SIMP_TAC
rfs	Corresponds to REV_FULL_SIMP_TAC

Thus, the following is possible:

```
> g `x < 3 /\ P x ==> x < 20 DIV 2`;  ...output elided...
> e (simp[]);
<<HOL message: Initialising SRW simpset ... done>>
OK..
val it =
   Initial goal proved.
   ⊢ x < 3 ∧ P x ⇒ x < 20 DIV 2: proof
```

In addition, two similar tactics allow special-purpose simplification of propositional structure:

- `csimp` simplifies with `srw_ss()` and the `CONJ_ss` simpset fragment. The latter allows for conjuncts to be used as assumptions when rewriting other conjuncts. Thus, if simplifying $c_1 \wedge c_2$, c_2 will be assumed as a possible source of rewrites while c_1 is simplified to c_1'. Then c_1' will be assumed while c_2 is simplified.
- `dsimp` simplifies with `srw_ss()` and the `DNF_ss` simpset fragment. The latter normalises to disjunctive normal form and also moves quantifiers so as to maximise the chance that they can be eliminated over equalities. When this tactic works well, it can provide a "pure" simplification analogue of the repeated stripping and variable elimination done by `rw`.

7.5.2 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.

7.5.2.1 `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 7.5.5; the generation of rewrite rules from theorems is described in Section 7.5.4.

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

$$\exists x.\, P(x) \wedge \ldots (x = e) \cdots \wedge Q(x)$$

into

$$P(e) \wedge \cdots \wedge Q(e)$$

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 7.5.4 below, but can be illustrated by some examples (the first also demonstrates unwinding under a universal quantifier):

```
> SIMP_CONV bool_ss [] ``!x. (x = 3) /\ P x ==> Q x /\ P 3``;                    8
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``;
<<HOL message: inventing new type variable names: 'a>>
val it = ⊢ (if x ≠ 3 then P x else Q x) = if x ≠ 3 then P x else Q 3: thm
```

7.5.2.2 `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`.

```
> SIMP_CONV std_ss [] ``FST (x,y) + OUTR (INR z)``;                              9
<<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``;
<<HOL message: inventing new type variable names: 'a, 'b>>
val it = ⊢ (case SOME x of NONE => P | SOME y => f y) = 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.

```
> SIMP_CONV std_ss [] ``P (0 <= x) /\ Q (y + x - y)``;                           10
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 + 7² − 31 DIV 3 = 177: thm
```

7.5.2.3 `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 7.7 below.

These two facets of the `arith_ss` simpset are demonstrated here:

```
> SIMP_CONV arith_ss [] ``x < 3 /\ P x ==> x < 20 DIV 2``;                       11
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``;                    12
Exception- UNCHANGED raised
```

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 $2x + y < z$.

7.5.2.4 `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]``;             13
val it = ⊢ MAP (λx. 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 (λx. 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 7.6 below); `computeLib` is more efficient, but less general than the simplifier. For example:

```
> EVAL ``FILTER (\x. x < 4) [1;2;y + 4]``;                        14
val it =
   ⊢ FILTER (λx. x < 4) [1; 2; y + 4] =
     1::2::if y + 4 < 4 then [y + 4] else []: thm
```

7.5.2.5 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:

15
```
> Datatype `tree = Leaf | Node num tree tree`;
<<HOL message: Defined type: "tree">>
val it = (): unit

> SIMP_CONV (srw_ss()) [] ``Node x Leaf Leaf = Node 3 t1 t2``;
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 ∈ {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.

16
```
> 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
```

Alternatively, the user may also flag theorems directly when using store_thm, save_thm, or the Theorem and Definition syntaxes by appending the simp attribute to the name of the theorem. Thus:

```
Theorem useful_rwt[simp]:                                              17
  ...term...
Proof ...tactic...
QED
```

is a way of avoiding having to write a call to `export_rewrites`. Equally, the example above could be written:

```
> Definition tsize_def[simp]:                                          18
    (tsize Leaf = 0) /\
    (tsize (Node n t1 t2) = n + tsize t1 + tsize t2)
  End  ...output elided...
```

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.

7.5.3 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 7.5.4.3.)

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 7.1. 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 7.1: 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
```

7.5.3.1 Removing rewrites and conversions from simpsets

The -* (infix) function can be used to remove elements from simpsets. This can be done to temporarily affect the simplifier when it is applied to a particular goal. For example:

19
```
> SIMP_CONV (srw_ss()) [] “x ++ (y ++ z)”;
<<HOL message: inventing new type variable names: 'a>>
val it = ⊢ x ++ (y ++ z) = x ++ y ++ z: thm

> SIMP_CONV (srw_ss() -* ["APPEND_ASSOC"]) [] “x ++ (y ++ z)”;
Exception- <<HOL message: inventing new type variable names: 'a>>
UNCHANGED raised
```

The second argument to -* is a list of strings, naming rewrite theorems or conversions. The names to use are visible if simpset values are printed out in the interactive session. The example below demonstrates removing the beta-conversion:

20
```
> SIMP_CONV (bool_ss -* ["BETA_CONV"]) [] “(\x. x + 3) 10”;
Exception- UNCHANGED raised
```

Further, because a theorem like AND_CLAUSES

```
AND_CLAUSES
  ⊢ ∀t.
        (T ∧ t ⇔ t) ∧ (t ∧ T ⇔ t) ∧ (F ∧ t ⇔ F) ∧
        (t ∧ F ⇔ F) ∧ (t ∧ t ⇔ t)
```

has multiple conjuncts, one theorem can generate multiple different rewrites. Specific sub-rewrites can be removed from a simpset without affecting others derived from the same original theorem by appending numbers to the theorem name:

```
> SIMP_CONV (bool_ss -* ["AND_CLAUSES"]) [] “(T ∧ p) ∧ (q ∧ T)”;                21
Exception- UNCHANGED raised

> SIMP_CONV (bool_ss -* ["AND_CLAUSES.1"]) [] “(T ∧ p) ∧ (q ∧ T)”;
val it = ⊢ (T ∧ p) ∧ q ∧ T ⟺ (T ∧ p) ∧ q: thm
```

If using a “power tactic” such as `simp`, there is no simpset value visible to modify with `-*`. Instead, one must use a special theorem form (see Section 7.5.5.4 below), `Excl` to exclude a rewrite. For example, sometimes the associativity of list-append can be annoying (here it masks the rewrite defining list-append):

```
> g `f (x ++ (h::t ++ y)) = f (x ++ h::(t ++ y))`;  ...output elided...        22
> e (simp[]);
OK..
1 subgoal:
val it =

   f (x ++ h::t ++ y) = f (x ++ h::(t ++ y))
```

We can prevent the application of this normalising rewrite with the `Excl` form:

```
> b();  ...output elided...                                                    23
> e (simp[Excl "APPEND_ASSOC"]);
OK..
val it =
   Initial goal proved.
   ⊢ f (x ++ (h::t ++ y)) = f (x ++ h::(t ++ y)): proof
```

7.5.4 Rewriting with the simplifier

Rewriting is the simplifier's “core operation”. This section describes the action of rewriting in more detail.

7.5.4.1 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 7.5.4.4 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 7.5.5.1 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 7.5.5.3). If the rewriting phase or any of the decision procedures altered the term, return to step 1. Otherwise, finish.

7.5.4.2 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.)

7.5.4.3 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 7.5.2.1). 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 7.5.5.4 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

$$\vdash \ (\forall n\ x.\ 0 < n \Rightarrow ((x + n)\,\mathsf{MOD}\,n = x\,\mathsf{MOD}\,n)) \ \wedge \\ (\forall n\ x.\ 0 < n \Rightarrow ((n + x)\,\mathsf{MOD}\,n = x\,\mathsf{MOD}\,n))$$

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.

7.5.4.4 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.

7.5.5 Advanced features

This section describes some of the simplifier's advanced features.

7.5.5.1 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 7.5.4.4, this rule will match all possible abstraction terms.

These simple examples have not yet demonstrated the use of *ctxt* conditions on sub-equations. 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

$$\begin{array}{l}\vdash\ (P = P') \Rightarrow (P' \Rightarrow (x = x')) \Rightarrow (\neg P' \Rightarrow (y = y'))\ \Rightarrow \\ \quad (\mathsf{if}\ P\ \mathsf{then}\ x\ \mathsf{else}\ y = \mathsf{if}\ P'\ \mathsf{then}\ x'\ \mathsf{else}\ y')\end{array}$$

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 7.5.4.3.

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):

$$\begin{array}{lll}\vdash & \mathsf{if}\ \top\ \mathsf{then}\ x\ \mathsf{else}\ y & = \quad x \\ \vdash & \mathsf{if}\ \bot\ \mathsf{then}\ x\ \mathsf{else}\ y & = \quad y\end{array}$$

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.

7.5.5.2 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 { name = NONE,                                                    24
convs = [], rewrs = [], congs = [],
filter = NONE, ac = [(DISJ_ASSOC, DISJ_COMM)],
dprocs = [] }
```

The pair of provided theorems must state

$$\begin{aligned} x \oplus y &= y \oplus x \\ x \oplus (y \oplus z) &= (x \oplus y) \oplus z \end{aligned}$$

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``;              25
<<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``;   26
<<HOL message: inventing new type variable names: 'a>>
val it = ⊢ p ∧ q ∨ r ∨ P z ⇔ r ∨ P z ∨ p ∧ q: thm
```

See Section 7.5.5.4 for more on special rewrite forms.

7.5.5.3 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 7.5.4.4, 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 is lifted for decision procedures (see below), but for conversions 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,
                    conv : 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, a more general call-back to the simplifier, 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 `conv` argument is call-back to the simplifier, which given a term t returns a theorem of the form $\vdash t = t'$ or fails. In contrast, the `solver` either returns the theorem $\vdash t$ or fails. 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.

7.5.5.4 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.

Another pair of special forms allow the user to *require* that certain rewrites are applied. Both forms check the count of instances of rewrite-redexes appearing in the goal that results after simplification has happened. If the requirement is not satisfied, the relevant tactic fails. In this context, a rewrite redex is the LHS of a theorem being used as a rewrite, so that, for example, the redex of the theorem $\vdash x + 0 = x$ is $x + 0$. The `Req0` form checks that the number of redexes of the corresponding rewrite is zero in the resulting goal. For unconditional rewrites, such a requirement is usually redundant, but this form can be useful when rewrites are conditional and the simplifier may have failed to discharge side-conditions. For example:

27
```
> val th = arithmeticTheory.ZERO_MOD;
val th = ⊢ ∀n. 0 < n ⇒ 0 MOD n = 0: thm
> simp[Req0 th] ([], ``0 MOD z``);
Exception- HOL_ERR
  {message = "LHS of |- !n. 0 < n ==> 0 MOD n = 0 remains in goal",
   origin_function = "REQUIRE0_TAC", origin_structure = "Ho_Rewrite"} raised

> simp[Req0 th] ([], ``0 MOD (z + 1)``)
   (* succeeds because arithmetic d.p. knows z + 1 is nonzero *);
val it = ([([], "0")], fn): goal list * validation
```

The `ReqD` modifier requires that the redex count should have decreased. This is implicitly a check on the original goal as well: it must have a non-zero count of redexes itself.

Both `Req0` and `ReqD` can be combined with `Once` and `Ntimes`.

Simplifying at particular sub-terms We have already seen (Section 7.5.5.1 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.

7.5.5.5 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.

7.5.5.6 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 5.5.3).

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.

7.6 Efficient Applicative Order Reduction—`computeLib`

Section 6.1 and Section 6.5 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 7.5. 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";
val it = (): unit

> 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]``;
<<HOL message: inventing new type variable names: 'a>>
val it = ⊢ REVERSE [u; v; w; x; y; z] = [z; y; x; w; v; u]: thm

> EVAL ``REVERSE alist``;
<<HOL message: inventing new type variable names: 'a>>
val it = ⊢ REVERSE alist = REV alist []: thm
```

7.6.1 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)`;
Equations stored under "fact_def".
Induction stored under "fact_ind".
val it = ⊢ ∀n. 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";
val fact = “fact”: term
> computeLib.monitoring := SOME (same_const fact);
val it = (): unit

> 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 5.3.3. 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 HOL's top-level definition facilities (*i.e.*, the `Define` function and the `Definition` syntax) automatically add definitions 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.

Occasionally, one does *not* want a definition automatically added to the global compset. The easiest way to achieve this is to use the `nocompute` "pseudo-attribute":

```
> Definition f_def[nocompute]: f x = x + 10                                  6
  End
Definition has been stored under "f_def"
val f_def = ⊢ ∀x. f x = x + 10: thm
> EVAL ``f 6``;
val it = ⊢ f 6 = f 6: thm
```

7.7 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 5.3.6 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.

7.8 Pattern Matches Library—patternMatchesLib

HOL supports two different types of case expressions: decision tree based and PMATCH case expressions. These are presented in Section 6.4. In subsection 6.4.2, the basic usage of PMATCH case expressions is discussed. Some concepts presented there briefly are discussed here in detail. Moreover, advanced features are discussed here.

7.8.1 Simplification

The most important tool to deal with PMATCH case expressions is the conversion PMATCH_SIMP_CONV or the corresponding PMATCH_SIMP_ss, which is part of bossLib.std_ss. It combines the following methods of simplifying and (partially) evaluating PMATCH case expressions. A subset of these methods that skips normalisations and potentially expensive searches for redundant and subsumed rows is available as PMATCH_FAST_SIMP_CONV.

7.8.1.1 Normalisation

Many simplifications rely on the variables of a pattern being named consistently and no extra, unused pattern variables being present. The conversion PMATCH_CLEANUP_PVARS_

CONV removes unused pattern variables and ensures that the names of variables used by the pattern, the guard and the right-hand-side of a row coincide.

7
```
> PMATCH_CLEANUP_PVARS_CONV ``PMATCH (x:('a # 'b) option) [
    PMATCH_ROW (\x:'a. NONE) (\x. T) (\x. 5);
    PMATCH_ROW (\ (x,(y:'c)). SOME (x,z)) (\ (x,y). T) (\ (x,y). 8);
    PMATCH_ROW (\ (x,z). SOME (x,z)) (\_. T) (\ (a,y). 8)]``
val it =
   ⊢ PMATCH x
       [PMATCH_ROW (λx. NONE) (λx. T) (λx. 5);
        PMATCH_ROW (λ(x,y). SOME (x,z)) (λ(x,y). T) (λ(x,y). 8);
        PMATCH_ROW (λ(x,z). SOME (x,z)) (λ_. T) (λ(a,y). 8)] =
     case x of NONE => 5 | x .| SOME (x,z) => 8 | SOME (x,z) => 8: thm
```

Similarly, many PMATCH tools rely on each pattern of a case expression having the same number of columns. This normal form is enforced by PMATCH_EXPAND_COLS_CONV.

8
```
> PMATCH_EXPAND_COLS_CONV ``case (x,y,z) of
   (0,y,T) => y
  | xyz when ~ SND (SND xyz) => 2
  | (x,yz) => x``
val it =
   ⊢ (case (x,y,z) of
       (0,y,T) => y | xyz when ¬SND (SND xyz) => 2 | (x,yz) => x) =
     case (x,y,z) of
       (0,y,T) => y
     | (xyz_0,xyz_1,xyz_2) when ¬SND (SND (xyz_0,xyz_1,xyz_2)) => 2
     | (x,yz_0,yz_1) => x: thm
```

Finally, the conversion PMATCH_INTRO_WILDCARDS_CONV renames unused pattern variables such that they start with an underscore. As a result, they are printed as a wildcard pattern, making case expressions more readable. It also renames used variables that start with an underscore. This is rarely needed, though.

9
```
> PMATCH_INTRO_WILDCARDS_CONV ``case (x,y,z) of
   (_x, y, z) => _x + y
  | (x, y, z) when z => x``
val it =
   ⊢ (case (x,y,z) of (_,y,z) => _ + y | (x,y,z) when z => x) =
     case (x,y,z) of (v0,y,_) => v0 + y | (x,_,z) when z => x: thm
```

A combination of these conversions for normalising PMATCH case expressions is available as PMATCH_NORMALISE_CONV.

7.8.1.2 (Partial) evaluation

The function PMATCH_CLEANUP_CONV checks each row of a PMATCH case expression and determines whether it matches the tested expression. There are three possible outcomes of such a check: a proof that the row matches, a proof that the row does not match or that it could not be decided whether the row matches. Rows that are proved to not match are removed. Similarly, all rows after the first matching row are redundant and are removed. If the first remaining row is known to match, the whole case expression is evaluated.

The proof of whether a row matches is attempted using some default proof methods. In particular information about datatype constructors is automatically used from TypeBase and constrFamiliesLib (see Sec. 7.8.6.2). If used *via* PMATCH_SIMP_ss, a callback to the simplifier is used. The conversion PMATCH_CLEANUP_CONV_GEN is a generalised version of the partial evaluation conversion that allows manually providing additional simpset fragments to the used proof method.

In the following example, the first row is removed, because it does not match. The second line is kept, since depending on the value of y it might or might not match. Since the third line matches in any case, the fourth one is deleted.

10
```
> PMATCH_CLEANUP_CONV ``case (SOME (x:num),y) of
    (NONE, y) => 1
  | (x, 0) => 2
  | (SOME x, y) => 3
  | (x, y) => 4``
val it =
   ⊢ (case (SOME x,y) of
       (NONE,y) => 1 | (x,0) => 2 | (SOME x,y) => 3 | (x,y) => 4) =
     case (SOME x,y) of (x,0) => 2 | (SOME x,y) => 3: thm
```

If the first row remaining matches, the case expression is evaluated:

11
```
> PMATCH_CLEANUP_CONV ``case (SOME x, y) of
    (NONE, y) => 1
  | (SOME x, y) => x+y
  | (x, y) => 4``
val it =
   ⊢ (case (SOME x,y) of (NONE,y) => 1 | (SOME x,y) => x + y | (x,y) => 4) =
     x + y: thm
```

Similarly, if no row matches, the whole case expression is evaluated.

12
```
> PMATCH_CLEANUP_CONV ``case (SOME (x:num), y:num) of (NONE, y) => 1``
val it = ⊢ (case (SOME x,y) of (NONE,y) => 1) = PMATCH_INCOMPLETE: thm
```

7.8.1.3 Simplifying columns

Before, we saw how rows can be removed. PMATCH_SIMP_COLS_CONV allows removing a column of a PMATCH case expression. If for all rows a certain column matches the input value for this column, the column can be removed. This situation usually arises after removing certain rows from a case expression via partial evaluation.

13
```
> PMATCH_SIMP_COLS_CONV ``case (SOME x,y) of
  | (SOME x, 1) => x+y
  | (x, y) => 4``
val it =
   ⊢ (case (SOME x,y) of (SOME x,1) => x + y | (x,y) => 4) =
     case y of 1 => x + y | y => 4: thm
```

Similarly, a column is partially evaluated if all rows contain either a variable, a wildcard or a term of the same constructor in this column.

14
```
> PMATCH_SIMP_COLS_CONV ``case (SOME x,y) of
  | (SOME x, 1) => SOME (x+y)
  | (SOME 2, 2) => NONE
  | (x, y) => x``
val it =
   ⊢ (case (SOME x,y) of
       (SOME x,1) => SOME (x + y) | (SOME 2,2) => NONE | (x,y) => x) =
     case (x,y) of
       (x,1) => SOME (x + y) | (2,2) => NONE | (x_0,y) => SOME x_0: thm
```

7.8.1.4 Removing redundant rows

The simplifications above easily lead to case expressions that contain multiple similar rows. The conversion PMATCH_REMOVE_FAST_REDUNDANT_CONV is intended to cleanup such rows. A row is called redundant if each value that matches it also matches an earlier row. Redundant rows will never matter and can therefore safely be removed. Thus the conversion PMATCH_REMOVE_FAST_REDUNDANT_CONV checks whether a pattern of a row is an instance of a pattern of an earlier row. This simple, fast heuristic is sufficient to detect most instances of redundant rows occurring during simplification. In the following example, the rows with right-hand-side 2, 4 and 5 are redundant. However, this simple heuristic cannot detect that row 5 is redundant. A more advanced method for removing redundant rows, which is slower but for example able to detect that row 5 is redundant, is discussed in Section 7.8.7.

15
```
> PMATCH_REMOVE_FAST_REDUNDANT_CONV ``case xy of
  | (SOME x, y) => 1 | (SOME 2, 3) => 2
  | (NONE, y) => 3 | (NONE, y) => 4
  | (x, 5) => 5``
val it =
   ⊢ (case xy of
       (SOME x,y) => 1
     | (SOME 2,3) => 2
     | (NONE,y) => 3
     | (NONE,y) => 4
     | (x,5) => 5) =
     case xy of (SOME x,y) => 1 | (NONE,y) => 3 | (x,5) => 5: thm
```

7.8.1.5 Removing subsumed rows

Redundant rows are rows that are not needed, because they are shadowed by an earlier row. Similarly, subsumed rows are rows that can be dropped, because in case they match a later row matches as well and evaluates to the same value. It is trickier to check for subsumed rows, because one needs to check that no row between the subsuming row and the possibly subsumed row matches, and because the right hand sides of the rows need to be considered as well. The function PMATCH_REMOVE_FAST_SUBSUMED_CONV removes subsumed rows that can be detected quickly.

If no row matches, a PMATCH case expression evaluates to ARB. Therefore, a row with right-hand-side of ARB is considered to be subsumed if no further row matches. This is not always what users expect or want. For example, the user might not want to see an exhaustive pattern match turn into a non-exhaustive one. Thus PMATCH_REMOVE_FAST_-SUBSUMED_CONV takes an additional boolean argument ra, which allows one to configure whether such rows are removed.

16
```
> PMATCH_REMOVE_FAST_SUBSUMED_CONV true ``case xy of
  | (SOME 2, _) => 2 | (NONE, 3) => 1
  | (SOME x, _) => x | (NONE, y) => y
  | (x, 5) => ARB``
val it =
   ⊢ (case xy of
       (SOME 2,_) => 2
     | (NONE,3) => 1
     | (SOME x,_) => x
     | (NONE,y) => y
     | (x,5) => ARB) =
     case xy of (NONE,3) => 1 | (SOME x,_) => x | (NONE,y) => y: thm
```

17
```
> PMATCH_REMOVE_FAST_SUBSUMED_CONV false ``case xy of
  | (SOME 2, _) => 2 | (NONE, 3) => 1
  | (SOME x, _) => x | (NONE, y) => y
  | (x, 5) => ARB``
val it =
   ⊢ (case xy of
        (SOME 2,_) => 2
      | (NONE,3) => 1
      | (SOME x,_) => x
      | (NONE,y) => y
      | (x,5) => ARB) =
      case xy of
        (NONE,3) => 1 | (SOME x,_) => x | (NONE,y) => y | (x,5) => ARB: thm
```

The PMATCH_SIMP_CONV conversion keeps such rows.

18
```
> PMATCH_SIMP_CONV ``case xy of
  | (SOME 2, _) => 2 | (NONE, 3) => 1
  | (SOME x, _) => x | (NONE, y) => y
  | (x, 5) => ARB``
val it =
   ⊢ (case xy of
        (SOME 2,_) => 2
      | (NONE,3) => 1
      | (SOME x,_) => x
      | (NONE,y) => y
      | (x,5) => ARB) =
      case xy of
        (NONE,3) => 1 | (SOME x,_) => x | (NONE,y) => y | (_,5) => ARB: thm
```

7.8.2 Support for computeLib

The conversion PMATCH_CLEANUP_CONV (see Sec. 7.8.1.2) is added to the internal database of computeLib. This allows the efficient evaluation of ground terms that contain PMATCH case expressions.

19
```
> EVAL ``case (SOME 3, SOME 4) of
  | (SOME x, SOME y) => SOME (x + y)
  | (_, _) => NONE``
val it =
   ⊢ (case (SOME 3,SOME 4) of (SOME x,SOME y) => SOME (x + y) | (_,_) => NONE) =
     SOME 7: thm
```

20
```
> EVAL ``case (NONE, SOME 4) of
  | (SOME x, SOME y) => SOME (x + y)
  | (_, _) => NONE``
val it =
   ⊢ (case (NONE,SOME 4) of (SOME x,SOME y) => SOME (x + y) | (_,_) => NONE) =
     NONE: thm
```

7.8.3 Removing extra features

PMATCH case expressions support features that are not usually supported by programming languages. One can use the same pattern variable multiple times in a pattern and use variables not bound by a pattern. Moreover, there is support for guards.

Sometimes, it is desirable to remove such features from a PMATCH case expression. A typical example is that they need to be removed before code-extraction.

7.8.3.1 Normalising pattern variables

The function PMATCH_REMOVE_DOUBLE_BIND_CONV and the corresponding simpset fragment PMATCH_REMOVE_DOUBLE_BIND_ss remove variables bound multiple times by a pattern as well as variables not bound by the pattern. This is easily achievable by introducing extra variables into the pattern and constraining their value by adding extra conditions to the guard.

21
```
> PMATCH_REMOVE_DOUBLE_BIND_CONV ``case xy of
  | (x, x) when x > 0 => x + x
  | x.| (x, y) => x
  | (x, _) => SUC x``
val it =
   ⊢ (case xy of (x,x) when x > 0 => x + x | x .| (x,y) => x | (x,_) => SUC x) =
     case xy of
       (x,x') when x' = x ∧ x > 0 => x + x
     | (x,y') when y' = y => x
     | (x,_) => SUC x: thm
```

7.8.3.2 Removing guards

Guards can be removed by introducing an if-then-else expression on the right-hand-side. The else-part of this if-then-else expression needs to continue the case-split with the rows occurring after the row whose guard is removed. Usually this case expression can be simplified significantly, since we know that the input matches the pattern of the row, whose guard is removed. Therefore, the conversion PMATCH_REMOVE_GUARDS_CONV as well as the corresponding PMATCH_REMOVE_GUARDS_ss internally call PMATCH_SIMP_CONV.

22
```
> PMATCH_REMOVE_GUARDS_CONV ``case (x, y) of
  | (x, 2) when EVEN x => x + x
  | (SUC x, y) when ODD x => y + x + SUC x
  | (SUC x, 1) => x
  | (x, _) => x+3``
val it =
   ⊢ (case (x,y) of
        (x,2) when EVEN x => x + x
      | (SUC x,y) when ODD x => y + x + SUC x
      | (SUC x,1) => x
      | (x,_) => x + 3) =
     case (x,y) of
       (x,2) =>
            if EVEN x then x + x
            else case x of SUC x when ODD x => 2 + x + SUC x | x => x + 3
     | (SUC x,y) =>
            if ODD x then y + x + SUC x
            else case y of 1 => x | _ => SUC x + 3
     | (x,_) => x + 3: thm
```

23
```
> PMATCH_REMOVE_GUARDS_CONV ``case (x, y) of
  | (x, 0) when EVEN x => (SOME x, T)
  | (x, 0) => (SOME x, F)
  | (0, _) => (NONE, T)
  | (_, _) => (NONE, F)``
val it =
   ⊢ (case (x,y) of
        (x,0) when EVEN x => (SOME x,T)
      | (x,0) => (SOME x,F)
      | (0,_) => (NONE,T)
      | (_,_) => (NONE,F)) =
     case (x,y) of
       (x,0) => if EVEN x then (SOME x,T) else (SOME x,F)
     | (0,_) => (NONE,T)
     | (_,_) => (NONE,F): thm
```

24
```
> SIMP_CONV (std_ss ++ PMATCH_REMOVE_GUARDS_ss) [] ``case x of
  | _ when x < 5 => 0
  | _ when x < 10 => 1
  | _ => 2``
val it =
   ⊢ (case x of _ when x < 5 => 0 | _ when x < 10 => 1 | _ => 2) =
     if x < 5 then 0 else if x < 10 then 1 else 2: thm
```

7.8.4 Lifting case expressions

HOL provides powerful tools for rewriting. Probably the most commonly used way of using case expressions in HOL is at top-level for defining recursive functions. Special

support in Define turns multiple top-level equations into a decision tree case expressions, uses this case expression for defining a function and then derives top-level equations similar to the input ones. Since compilation to decision trees is used, the issues discussed in Sec. 6.4 are present. It is sometimes hard to predict, which equations will be generated. There might be a blow-up in the number of equations. Moreover, equations cannot overlap and are therefore often unnecessarily complicated.

As an example consider the following definition of a zipping functions for lists.

25
```
> val MYZIP_def = Define `
  (MYZIP [] _ = []) /\
  (MYZIP _ [] = []) /\
  (MYZIP (x::xs) (y::ys) = (x,y) :: (MYZIP xs ys))`  ...output elided...

val MYZIP_def =
   ⊢ (∀v0. MYZIP [] v0 = []) ∧ (∀v4 v3. MYZIP (v3::v4) [] = []) ∧
     ∀ys y xs x. MYZIP (x::xs) (y::ys) = (x,y)::MYZIP xs ys: thm
```

26
```
> val MYZIP2_def = Pmatch.with_classic_heuristic Define `
  (MYZIP2 [] _ = []) /\
  (MYZIP2 _ [] = []) /\
  (MYZIP2 (x::xs) (y::ys) = (x,y) :: (MYZIP2 xs ys))`  ...output elided...

val MYZIP2_def =
   ⊢ MYZIP2 [] [] = [] ∧ (∀v8 v7. MYZIP2 [] (v7::v8) = []) ∧
     (∀v4 v3. MYZIP2 (v3::v4) [] = []) ∧
     ∀ys y xs x. MYZIP2 (x::xs) (y::ys) = (x,y)::MYZIP2 xs ys: thm
```

We can use PMATCH case expressions to fight these issues. There is, however, no special support for PMATCH case expressions built into Define. Instead, one needs to define a function with a PMATCH case expression on the right-hand-side. Using the rule PMATCH_TO_TOP_RULE then produces the desired (conditional) equations.

27
```
> val MYZIP3_def = Define `
  MYZIP3 xl yl = (case (xl, yl) of
  | ([], _) => []
  | (_, []) => []
  | (x::xs, y::ys) => (x,y) :: (MYZIP3 xs ys))`  ...output elided...

> val MYZIP3_EQS = PMATCH_TO_TOP_RULE MYZIP3_def
val MYZIP3_EQS =
   ⊢ (∀yl. MYZIP3 [] yl = []) ∧ (∀xl. MYZIP3 xl [] = []) ∧
     ∀x xs y ys. MYZIP3 (x::xs) (y::ys) = (x,y)::MYZIP3 xs ys: thm
```

Similarly, the resulting induction theorems are more predictable and contain fewer cases. However, the structure tends not to be as nice.

28
```
val MYZIP_ind =
   ⊢ ∀P.
         (∀v0. P [] v0) ∧ (∀v3 v4. P (v3::v4) []) ∧
         (∀x xs y ys. P xs ys ⇒ P (x::xs) (y::ys)) ⇒
         ∀v v1. P v v1: thm
val MYZIP2_ind =
   ⊢ ∀P.
         P [] [] ∧ (∀v7 v8. P [] (v7::v8)) ∧ (∀v3 v4. P (v3::v4) []) ∧
         (∀x xs y ys. P xs ys ⇒ P (x::xs) (y::ys)) ⇒
         ∀v v1. P v v1: thm
val MYZIP3_ind =
   ⊢ ∀P.
         (∀xl yl.
              (∀x xs y ys. (xl,yl) = (x::xs,y::ys) ∧ T ⇒ P xs ys) ⇒ P xl yl) ⇒
         ∀v v1. P v v1: thm
```

For the zipping examples the resulting equations are particularly nice. In general, conditional equations need to be generated. The preconditions state that no previous row matched or that the result of such a matching row coincides with the result of the current row.

29
```
> val MYZIP4_def = Define `
  MYZIP4 xl yl = (case (xl, yl) of
  | ([], []) => (NONE, [])
  | ([], _) => (SOME T, [])
  | (_, []) => (SOME F, [])
  | (x::xs, y::ys) => (dtcase (MYZIP4 xs ys) of
     | (r, l) => (r, (x,y)::l)))`  ...output elided...

> val MYZIP4_EQS = PMATCH_TO_TOP_RULE MYZIP4_def
val MYZIP4_EQS =
   ⊢ MYZIP4 [] [] = (NONE,[]) ∧ (∀yl. [] ≠ yl ⇒ MYZIP4 [] yl = (SOME T,[])) ∧
     (∀xl. [] ≠ xl ⇒ MYZIP4 xl [] = (SOME F,[])) ∧
     ∀x xs y ys.
         MYZIP4 (x::xs) (y::ys) =
         dtcase MYZIP4 xs ys of (r,l) => (r,(x,y)::l): thm
```

The lifting functionality is also available via PMATCH_LIFT_BOOL_ss and PMATCH_LIFT_BOOL_CONV, which lift a PMATCH case expression to the next highest boolean level and expands it there. Since trying to prove exhaustiveness (see Sec. 7.8.9) might be slow, there is flag for turning it on and off explicitly. Moreover, notice that PMATCH_LIFT_BOOL_CONV always tries to lift to the top-level. Therefore, it should usually be combined with something like DEPTH_CONV.

30
```
> DEPTH_CONV (PMATCH_LIFT_BOOL_CONV true) ``
  P /\ (f (case x of [] => 0 | x::xs => x) = 5) /\ Q``
val it =
   ⊢ P ∧ f (case x of [] => 0 | x::xs => x) = 5 ∧ Q ⟺
     P ∧ ((x = [] ⇒ f 0 = 5) ∧ ∀x' xs. x = x'::xs ⇒ f x' = 5) ∧ Q: thm
```

31
```
> DEPTH_CONV (PMATCH_LIFT_BOOL_CONV false) ``
  P /\ (f (case x of [] => 0 | x::xs => x) = 5) /\ Q``
val it =
   ⊢ P ∧ f (case x of [] => 0 | x::xs => x) = 5 ∧ Q ⟺
     P ∧
     ((x = [] ⇒ f 0 = 5) ∧ (∀x' xs. x = x'::xs ⇒ f x' = 5) ∧
      (¬PMATCH_IS_EXHAUSTIVE x
         [PMATCH_ROW (λ_. []) (λ_. T) (λ_. 0);
          PMATCH_ROW (λ(x,xs). x::xs) (λ(x,xs). T) (λ(x,xs). x)] ⇒
       f ARB = 5)) ∧ Q: thm
```

7.8.5 Translating PMATCH and decision tree case expressions

As discussed in Sec. 6.4, there are benefits to both PMATCH and decision tree based case expressions. Therefore, there are tools for translating between both representations.

The function `pmatch2case` uses the pattern compilation algorithm implemented in HOL's parser to generate decision tree case expressions. This is done outside the logic without any formal justification. However, a brute force method that repeatedly performs case splits and evaluates is sufficient for proving equivalence. This leads to `PMATCH_ELIM_CONV`.

Only PMATCH case expressions that fall into the subset supported by decision tree ones can be translated. This means that no guards can be used and that all patterns need to be constructor patterns.

32
```
> PMATCH_ELIM_CONV
  ``case (xy:(num option # num list)) of (NONE, x::xs) => 0``
<<HOL message: mk_functional:
  pattern completion has added 2 clauses to the original specification.>>
val it =
   ⊢ (case xy of (NONE,x::xs) => 0) =
     dtcase xy of (v,[]) => ARB | (NONE,x::xs) => 0 | (SOME v5,x::xs) => ARB:
   thm
```

An approach similar to the one implemented in HOL's pretty printer allows the translation of decision tree case expressions to equivalent PMATCH expressions. The underlying function is `case2pmatch do_opt`, where the `do_opt` flag determines whether certain non-trivial optimisations are attempted. The corresponding conversions are named `PMATCH_INTRO_CONV` and `PMATCH_INTRO_CONV_NO_OPTIMISE`.

33
```
> PMATCH_INTRO_CONV
  ``dtcase (xy:(num option # num list)) of (NONE, x::xs) => 0``
<<HOL message: mk_functional:
  pattern completion has added 2 clauses to the original specification.>>
val it =
   ⊢ (dtcase xy of (v,[]) => ARB | (NONE,x::xs) => 0 | (SOME v5,x::xs) => ARB) =
     case xy of (NONE,_::_) => 0: thm
```

```
> PMATCH_INTRO_CONV_NO_OPTIMISE                                        34
  ``dtcase (xy:(num option # num list)) of (NONE, x::xs) => 0``
<<HOL message: mk_functional:
  pattern completion has added 2 clauses to the original specification.>>
val it =
   ⊢ (dtcase xy of (v,[]) => ARB | (NONE,x::xs) => 0 | (SOME v5,x::xs) => ARB) =
     case xy of (v,[]) => ARB | (NONE,x::xs) => 0 | (SOME v5,x::xs) => ARB:
   thm
```

7.8.6 Pattern Compilation

The pmatch2case function allows PMATCH case expressions to be compiled into decision tree case expressions. It is fast and the result is usually pretty good. However, it relies on the pattern compilation implementation of HOL's parser. This has several drawbacks. The most significant one is that it is an all-or-nothing approach. Either the compilation succeeds and we get an equivalent decision tree case expression (without proof) or it fails and one has nothing. It is not easily possible to get partial results or use the information obtained during pattern compilation to prove exhaustiveness or find a set of missing patterns. With simplification of PMATCH case expressions (see Sec. 7.8.1) in place, it is straightforward to implement pattern compilation. One performs a case-split on one variable occurring in the input of the case expression, simplifies and iterates. This is implemented as PMATCH_CASE_SPLIT_CONV. Note that PMATCH_CASE_SPLIT_CONV has no support for guards or pattern variables bound multiple times. These features need to be removed (see Sec. 7.8.3) before calling it.

```
> PMATCH_CASE_SPLIT_CONV                                               35
  ``case l of (SOME x, SOME y) => SOME (x+y) | (_, _) => NONE``
val it =
   ⊢ (case l of (SOME x,SOME y) => SOME (x + y) | (_,_) => NONE) =
     dtcase l of
       (NONE,v') => NONE
     | (SOME x',NONE) => NONE
     | (SOME x',SOME x'') => SOME (x' + x''): thm
```

The trick is to choose which case split to apply next. This decision is taken by two mechanisms: a *column heuristic* picks the column to perform a case split on and the available case-splits are maintained by the *constructor family library*.

7.8.6.1 Column Heuristic

The most important decision during pattern compilation is which column, *i.e.*, which input variable to perform a case split on next. Different decisions lead to different decision trees, which can differ significantly in size and time needed to evaluate. It is not trivial to find a good column to split on. Currently, mainly heuristics presented by

Maranget [8] are implemented. In HOL a column heuristic is a ML function of type `column_heuristic`. Given a list of columns such a heuristic returns the number of the column to perform a split on. There are very simple heuristics like always picking the first or last column, but also sophisticated ones like qba (see [8]). Users can easily implement additional heuristics should the need arise. Figure 7.2 shows the effects of using different heuristics.

7.8.6.2 Constructor Family Library

Once a column has been chosen, a case split needs to be performed. This requires getting information about an appropriate case split function. Moreover, even for selecting the column some heuristics need information. It might for example be desirable to know to how many cases splitting on a column would lead.

Essentially, one needs to lookup the constructors of a datatype together with its case-constant. Moreover, theorems about injectivity and pairwise distinctiveness of the constructors as well as some theorems about the case-constant are needed. All this information can be found in `TypeBase` (see Sec. 6.1). The pattern compilation algorithm in the parser uses `TypeBase`. However, `patternMatchesLib` has two demands not met by `TypeBase`. For each type, it should be possible to store multiple sets of constructors. Moreover, sometimes the case split should not be stored statically but—given a column—be computed dynamically. These demands lead to the implementation of `constrFamiliesLib`.

The constructor family library `contrFamiliesLib` is a library for collecting information about constructors and case-splits. At its core is the concept of a *constructor family*. A constructor family is a list of functions together with a case-split functions. The functions should all be injective and pairwise distinct. The case-split function should provide a case-analysis that corresponds with the functions. Constructor families can be exhaustive or inexhaustive. For inexhaustive ones, the case-split function has to provide an extra *otherwise*-case.

The constructors of datatypes together with the case constant form constructor families. For example, the constructors `[]` and `CONS` with `list_CASE` form an exhaustive constructor family for lists. The information for the constructor families corresponding to the datatype constructors is automatically extracted from `TypeBase` and available *via* `constrFamiliesLib`. However, there might be other interesting constructor families. For example, `[]` and `SNOC` together with an appropriate case split function form another exhaustive constructor family for lists.

First, we need to define a case-split function for `[]` and `SNOC`.

36

```
> val t = ‘‘...‘‘

> PMATCH_CASE_SPLIT_CONV_HEU colHeu_first_col t
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
val it =
   ⊢ (case (x,y,z) of
        (_,F,T) => 1 | (F,T,_) => 2 | (_,_,F) => 3 | (_,_,T) => 4) =
     dtcase (x,y,z) of
       (T,T,T) => 4
     | (T,T,F) => 3
     | (T,F,T) => 1
     | (T,F,F) => 3
     | (F,T,v'3') => 2
     | (F,F,T) => 1
     | (F,F,F) => 3: thm

> PMATCH_CASE_SPLIT_CONV_HEU colHeu_last_col t
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
val it =
   ⊢ (case (x,y,z) of
        (_,F,T) => 1 | (F,T,_) => 2 | (_,_,F) => 3 | (_,_,T) => 4) =
     dtcase (x,y,z) of
       (T,T,T) => 4
     | (F,T,T) => 2
     | (v,F,T) => 1
     | (T,T,F) => 3
     | (F,T,F) => 2
     | (v,F,F) => 3: thm

> PMATCH_CASE_SPLIT_CONV_HEU colHeu_default t
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
val it =
   ⊢ (case (x,y,z) of
        (_,F,T) => 1 | (F,T,_) => 2 | (_,_,F) => 3 | (_,_,T) => 4) =
     dtcase (x,y,z) of
       (T,T,T) => 4
     | (T,T,F) => 3
     | (F,T,v'3') => 2
     | (v,F,T) => 1
     | (v,F,F) => 3: thm
```

Figure 7.2: Effect of different column heuristics

```
> val list_REVCASE_def = Define `                                        37
    list_REVCASE l c_nil c_snoc =
      (if l = [] then c_nil else (c_snoc (LAST l) (BUTLAST l)))`
...output elided...
```

Next, we define an exhaustive list of constructors. This is the list of functions combined with names for the arguments of each constructor.

```
> open constrFamiliesLib  ...output elided...                            38
> val cl = make_constructorList true [
     (``[]:'a list``, []),
     (``SNOC: 'a -> 'a list -> 'a list``,  ["x", "xs"])]
val cl = ?: constructorList
```

The function `mk_constructorFamily` is then used to create a constructor family. This requires proving the discussed properties. In order to develop the necessary tactic, `set_constructorFamily` can be used.

```
> set_constructorFamily (cl, ‘‘list_REVCASE‘‘)                           39
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        (∀x xs x' xs'. SNOC x xs = SNOC x' xs' ⟺ x = x' ∧ xs = xs') ∧
        ((∀x xs. [] ≠ SNOC x xs) ∧ ∀x xs. SNOC x xs ≠ []) ∧
        (∀ff x. ff x = list_REVCASE x (ff []) (λx' xs. ff (SNOC x' xs))) ∧
        (∀x' f1 f2 x f1' f2'.
             x' = x ⇒
             (x = [] ⇒ f1 = f1') ⇒
             (∀x' xs. x = SNOC x' xs ⇒ f2 x' xs = f2' x' xs) ⇒
             list_REVCASE x' f1 f2 = list_REVCASE x f1' f2') ∧
        ∀x. x = [] ∨ ∃x' xs. x = SNOC x' xs

> val cf = mk_constructorFamily (cl, ‘‘list_REVCASE‘‘, ... some tactic ...)
val cf = ?: constructorFamily
```

Finally, we can register this newly defined constructor family.

```
> val _ = pmatch_compile_db_register_constrFam cf                        40
```

Now this new family is available for pattern compilation. Notice, that the old constructors for lists are still present.

41
```
> PMATCH_CASE_SPLIT_CONV ``case ll of
    (SNOC x xs, []) => x
  | ([], x::xs) => x
  | (_, _) => 0``
metis: r[+0+5]+0+0+0+1+0+2#
val it =
   ⊢ (case ll of (SNOC x xs,[]) => x | ([],x::xs) => x | (_,_) => 0) =
     dtcase ll of
       (v,v') =>
         list_REVCASE v (dtcase v' of [] => 0 | h::t => h)
           (λx' xs. dtcase v' of [] => x' | h'::t' => 0): thm
```

Inexhaustive constructor families are often handy as well. Consider the example of red-black-trees defined as follows:

42
```
> val _ = Datatype `
  tree = Empty
       | Red tree 'a tree
       | Black tree 'a tree`;  ...output elided...
```

A lot of functions (*e.g.*, balancing) treat black nodes and leaves the same. However, when compiling corresponding case expressions to decision trees, 3 cases instead of the required 2 are produced. Defining an inexhaustive constructor family for just the RED constructor solves this issue (see Fig. 7.3).

7.8.6.3 Compiling to nchotomy theorems

Compiling to decision tree based case expressions is sometimes handy. However, computing the patterns corresponding to this decision tree is even more useful, since this set of patterns has very interesting properties. It is exhaustive and for each input pattern each pattern in this set is either a subpattern of the input pattern or distinct. There are no partial overlaps. Even better, whether an output pattern is a subpattern of an input pattern is checkable *via* simple first order matching.

Let's look at an example. First we compile a case expression to a decision tree.

44
```
> PMATCH_CASE_SPLIT_CONV ``case xy of
  | (SOME x, SOME y) => x + y
  | (_, SOME 0) => 0``
metis: r[+0+5]+0+0+0+1+0+2#
val it =
   ⊢ (case xy of (SOME x,SOME y) => x + y | (_,SOME 0) => 0) =
     dtcase xy of
       (v,NONE) => ARB
     | (NONE,SOME 0) => 0
     | (NONE,SOME (SUC n)) => ARB
     | (SOME x'',SOME x') => x'' + x': thm
```

43

```
> val tree_red_CASE_def = Define `
   tree_red_CASE tr f_red f_else =
   tree_CASE tr (f_else Empty) f_red
     (\t1 n t2. f_else (Black t1 n t2))`  ...output elided...
> val cl = make_constructorList false [(``Red``, ["t1", "n", "t2"])]
...output elided...
> val cf = mk_constructorFamily (cl, ‘‘tree_red_CASE‘‘, ... some tactic ...)
> val _ = pmatch_compile_db_register_constrFam cf  ...output elided...

> PMATCH_CASE_SPLIT_CONV ``case (t:'a tree) of
    | Red _ _ _ => T
    | _ => F``
val it =
   ⊢ (case t of Red _ _ _ => T | _ => F) ⟺
     tree_red_CASE t (λt1 n t2. T) (λx. F): thm

> PMATCH_CASE_SPLIT_CONV ``case (t:'a tree) of
    | Black _ _ _ => T
    | _ => F``
val it =
   ⊢ (case t of Black _ _ _ => T | _ => F) ⟺
     dtcase t of Empty => F | Red t a t0 => F | Black t' a' t0' => T: thm
```

Figure 7.3: Example inexhaustive constructor family

We end up with 4 rows in the pretty-printed form of the decision tree case expression. These 4 output patterns have the desired properties. They are exhaustive and for example `(NONE, SOME 0)` is a subpattern of `(_, SOME 0)`, but distinct from `(SOME x, SOME y)`. The `nchotomy_of_pats` function compiles the list of input patterns to an nchotomy-theorem containing exactly these 4 patterns.

45
```
> nchotomy_of_pats [``\(x,y). (SOME (x:num), SOME (y:num))``,
                    ``\(xo:num option). (xo, SOME 0)``]
val it =
   ⊢ ∀x.
         (∃v0. x = (v0,NONE)) ∨ x = (NONE,SOME 0) ∨
         (∃v4. x = (NONE,SOME (SUC v4))) ∨ (∃v3. x = (SOME v3,SOME 0)) ∨
         ∃v3 v5. x = (SOME v3,SOME (SUC v5)): thm
```

Such nchotomy theorems are very useful for finding missing patterns, detecting redundant rows and proving exhaustiveness. Essentially, one just removes one input pattern after the other by applying first order matching. The patterns that remain are not covered by the input.

7.8.7 Removing Redundant Rows

Using pattern compilation, it is straightforward to implement advanced redundancy checks. The conversion `PMATCH_REMOVE_REDUNDANT_CONV` and the corresponding simpset fragment `PMATCH_REMOVE_REDUNDANT_ss` are able to remove row 5 of the example already discussed in Sec. 7.8.1.4.

46
```
> PMATCH_REMOVE_REDUNDANT_CONV ``case xy of
  | (SOME x, y) => 1 | (SOME 2, 3) => 2
  | (NONE, y) => 3 | (NONE, y) => 4
  | (x, 5) => 5``
metis: r[+0+5]+0+0+1+1+2+0+1+0+0#
metis: r[+0+7]+0+0+0+0+1+1+2+2+2+2+2+0+0+1+0+1+0+0+0+1+1+0+2#
val it =
   ⊢ (case xy of
        (SOME x,y) => 1
      | (SOME 2,3) => 2
      | (NONE,y) => 3
      | (NONE,y) => 4
      | (x,5) => 5) =
      case xy of (SOME x,y) => 1 | (NONE,y) => 3: thm
```

If the redundancy of a row depends not only on patterns, but also guards, the automated method often fails. Figure 7.4 shows an example, where the information that each natural number is either even or odd is needed to show that a row is redundant. In such situations, it is often beneficial to combine the automated redundancy removal technique with manual reasoning (as in the figure).

```
> val t = ``case x of _ when EVEN x => 0 | _ when ODD x => 1 | _ => 2``   47
...output elided...
> PMATCH_REMOVE_REDUNDANT_CONV t
Exception- UNCHANGED raised

> val info = COMPUTE_REDUNDANT_ROWS_INFO_OF_PMATCH t
val info =
   ⊢ IS_REDUNDANT_ROWS_INFO x
       [PMATCH_ROW (λ_0. _0) (λ_0. EVEN x) (λ_0. 0);
        PMATCH_ROW (λ_0. _0) (λ_0. ODD x) (λ_0. 1);
        PMATCH_ROW (λ_0. _0) (λ_0. T) (λ_0. 2)] F
       [¬∃_0. x = _0 ∧ EVEN x;
        (∃v0. x = v0 ∧ ¬EVEN x) ⇒ ¬∃_0. x = _0 ∧ ODD x;
        (∃v0. x = v0 ∧ ¬EVEN x ∧ ¬ODD x) ⇒ ¬∃_0. x = _0]: thm

> IS_REDUNDANT_ROWS_INFO_SHOW_ROW_IS_REDUNDANT_set_goal info 2
val it =
   Proof manager status: 2 proofs.
   2. Incomplete goalstack:
        Initial goal:
        (∀x xs x' xs'. SNOC x xs = SNOC x' xs' ⟺ x = x' ∧ xs = xs') ∧
        ((∀x xs. [] ≠ SNOC x xs) ∧ ∀x xs. SNOC x xs ≠ []) ∧
        (∀ff x. ff x = list_REVCASE x (ff []) (λx' xs. ff (SNOC x' xs))) ∧
        (∀x' f1 f2 x f1' f2'.
             x' = x ⇒
             (x = [] ⇒ f1 = f1') ⇒
             (∀x' xs. x = SNOC x' xs ⇒ f2 x' xs = f2' x' xs) ⇒
             list_REVCASE x' f1 f2 = list_REVCASE x f1' f2') ∧
        ∀x. x = [] ∨ ∃x' xs. x = SNOC x' xs

   1. Incomplete goalstack:
        Initial goal:
        (∃v0. x = v0 ∧ ¬EVEN x ∧ ¬ODD x) ⇒ ¬∃_0. x = _0

> val info' = IS_REDUNDANT_ROWS_INFO_SHOW_ROW_IS_REDUNDANT info 2 ...
val info' =
   ⊢ IS_REDUNDANT_ROWS_INFO x
       [PMATCH_ROW (λ_0. _0) (λ_0. EVEN x) (λ_0. 0);
        PMATCH_ROW (λ_0. _0) (λ_0. ODD x) (λ_0. 1);
        PMATCH_ROW (λ_0. _0) (λ_0. T) (λ_0. 2)] F
       [¬∃_0. x = _0 ∧ EVEN x;
        (∃v0. x = v0 ∧ ¬EVEN x) ⇒ ¬∃_0. x = _0 ∧ ODD x; T]: thm

> val thm = IS_REDUNDANT_ROWS_INFO_TO_PMATCH_EQ_THM info'
val thm =
   ⊢ (case x of _ when EVEN x => 0 | _ when ODD x => 1 | _ => 2) =
     case x of _ when EVEN x => 0 | _ when ODD x => 1: thm
```

Figure 7.4: Manual reasoning about redundant rows

7.8.8 Pattern Match Completion

The techniques used for computing redundant rows implicitly compute a set of missing patterns. The conversion PMATCH_COMPLETE_CONV and simpset fragment PMATCH_COMPLETE_ss use this implicitly computed information to extend case expressions with ARB rows and thereby produce exhaustive PMATCH case expressions. A flag determines whether these newly introduced rows should use guards.

48
```
> PMATCH_COMPLETE_CONV true ``case (xy : (num option # num option)) of
      (SOME x, NONE) when x > 0 => 0 | (NONE, _) => 1``;
val it =
   ⊢ (case xy of (SOME x,NONE) when x > 0 => 0 | (NONE,_) => 1) =
     case xy of
       (SOME x,NONE) when x > 0 => 0
     | (NONE,_) => 1
     | (SOME v2,NONE) when ¬(v2 > 0) => ARB
     | (SOME v2,SOME v3) => ARB: thm
```

49
```
> PMATCH_COMPLETE_CONV false ``case (xy : (num option # num option)) of
      (SOME x, NONE) when x > 0 => 0 | (NONE, _) => 1``;
val it =
   ⊢ (case xy of (SOME x,NONE) when x > 0 => 0 | (NONE,_) => 1) =
     case xy of
       (SOME x,NONE) when x > 0 => 0
     | (NONE,_) => 1
     | (SOME v2,NONE) => ARB
     | (SOME v2,SOME v3) => ARB: thm
```

7.8.9 Exhaustiveness Checks

Similarly, exhaustiveness can be derived *via* pattern compilation.

50
```
> PMATCH_IS_EXHAUSTIVE_COMPILE_CHECK
    ``case (xy : (num option # num option)) of
      (SOME _, _) => 0 | (_, NONE) => 1 | (NONE, SOME _) => 2``
val it =
   ⊢ PMATCH_IS_EXHAUSTIVE xy
       [PMATCH_ROW (λ(_0,_1). (SOME _0,_1)) (λ(_0,_1). T) (λ(_0,_1). 0);
        PMATCH_ROW (λ_0. (_0,NONE)) (λ_0. T) (λ_0. 1);
        PMATCH_ROW (λ_0. (NONE,SOME _0)) (λ_0. T) (λ_0. 2)] ⟺ T: thm
```

Often, the exhaustiveness can be proved much faster by just searching a matching row.

51
```
> PMATCH_IS_EXHAUSTIVE_FAST_CHECK ``case (x:num option, y:num) of
      (SOME _, _) => 0 | (_, _) => 1``
val it =
   ⊢ PMATCH_IS_EXHAUSTIVE (x,y)
       [PMATCH_ROW (λ(_0,_1). (SOME _0,_1)) (λ(_0,_1). T) (λ(_0,_1). 0);
        PMATCH_ROW (λ(_0,_1). (_0,_1)) (λ(_0,_1). T) (λ(_0,_1). 1)] ⟺ T: thm

> PMATCH_IS_EXHAUSTIVE_FAST_CHECK ``case (xy : (num option # num option)) of
     (SOME _, _) => 0 | (_, NONE) => 1 | (NONE, SOME _) => 2``
Exception- UNCHANGED raised
```

Both methods are combined to from PMATCH_IS_EXHAUSTIVE_CHECK.

Another interface to the pattern compilation engine is provided by SHOW_NCHOTOMY_CONSEQ_CONV. Exhaustiveness is this time expressed in the form of an nchotomy theorem. Missing cases are automatically added.

52
```
> SHOW_NCHOTOMY_CONSEQ_CONV
    ``!x:'a list. (x = []) \/ (?e. x = [e]) \/ (?e1 e2 l. x = e1::e2::l)``
val it = ⊢ ∀x. T ⇒ x = [] ∨ (∃e. x = [e]) ∨ ∃e1 e2 l. x = e1::e2::l: thm
```

53
```
> SHOW_NCHOTOMY_CONSEQ_CONV
    ``!x:'a list. (x = []) \/ (?e1 e2 l. x = e1::e2::l)``
val it = ⊢ ∀x. ¬(∃v1. x = [v1]) ⇒ x = [] ∨ ∃e1 e2 l. x = e1::e2::l: thm
```

7.8.10 Code Extraction

There is support for PMATCH case expressions in EmitML. However, not all case expressions are supported. Supported case expressions may only contain constructor patterns and each pattern variable needs to be used exactly once. Moreover, when extracting to SML, no guards are allowed.

To check whether a case expression can be exported, the function analyse_pmatch can be used. The flag of this function indicates whether an exhaustiveness proof should be attempted.

54
```
> val info = analyse_pmatch false
  ``case l of [] => 1 | [x] when (x > 2) => 2 | _ => 3``
val info =
   {pmi_exhaustiveness_cond = NONE, pmi_has_double_bound_pat_vars = [],
    pmi_has_free_pat_vars = [], pmi_has_guards = [1], pmi_has_lambda_in_pat =
    [], pmi_has_non_contr_in_pat = [], pmi_has_unused_pat_vars = [],
    pmi_ill_formed_rows = [], pmi_is_well_formed = true}: pmatch_info

> val sml_ok = is_sml_pmatch info
val sml_ok = false: bool
> val ocaml_ok = is_ocaml_pmatch info
val ocaml_ok = true: bool
```

7.9 Bit Vector Library—wordsLib

The library wordsLib provides tool support for bit-vectors, this includes facilities for: evaluation, parsing, pretty-printing and simplification.

7.9.1 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.

```
> load "wordsLib";                                                    1
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.

7.9.2 Parsing and pretty-printing

Words can be parsed in binary, decimal and hexadecimal. For example:

```
> ``0b111010w : word8``;                                              2
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 = 43690w: 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.term_to_string ``32w``;  5
<<HOL message: inventing new type variable names: 'a>>
val it = "32w": string
```

The choices are as follows: 0 (default) – small numbers decimal, large numbers hexadecimal; 1 – binary; 2 – octal; 3 – decimal; and 4 – hexadecimal.

7.9.2.1 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`:

```
> Lib.try Parse.Type `:word15` handle _ => bool;                     6

Exception raised at Parse.type parser:
on line 1, characters 22-27:
word15 not a known type operator
val it = “:bool”: hol_type

> wordsLib.mk_word_size 15;
val it = (): unit

> ``:word15``;
val it = “:word15”: hol_type
```

7.9.2.2 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:

7

```
> type_of ``a + b``;
<<HOL message: more than one resolution of overloading was possible>>
<<HOL message: inventing new type variable names: 'a>>
val it = ":α word": 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:

8

```
> load "intLib";  ...output elided...

> 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>>
val it = ":α word": hol_type
```

Of course, type annotations could have been added to avoid this problem entirely.

7.9.2.3 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.

```
> wordsLib.guessing_word_lengths:=true;                                    9
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: α <- 4>>
<<HOL message: assigning word length: β <- 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.

7.9.3 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:

```
> fun conv ss = SIMP_CONV (pure_ss++ss) [];                                    10
val conv = fn: ssfrag -> conv
```

The following session demonstrates SIZES_ss:

```
> conv wordsLib.SIZES_ss ``dimindex(:12)``;                                    11
val it = ⊢ dimindex (:12) = 12: thm

> conv wordsLib.SIZES_ss ``FINITE univ(:32)``;
val it = ⊢ FINITE 𝕌(:32) ⟺ T: thm
```

The fragment BIT_ss converts BIT into membership test over a set of (high) bit positions:

```
> conv wordsLib.BIT_ss ``BIT 3 5``;                                            12
val it = ⊢ BIT 3 5 ⟺ F: thm

> conv wordsLib.BIT_ss ``BIT i 123``;
val it = ⊢ BIT i 123 ⟺ i ∈ {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.

7.9.3.1 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:

```
> load "blastLib";  ...output elided...                                             18
> blastLib.BBLAST_PROVE ``a + 2w <+ 4w <=> a <+ 2w \/ 13w <+ a :word4``;
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.

[9] This approach enables counter-examples to be given when a goal's negation is satisfiable.

7.10 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 "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 elided...

> 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.

```
> SAT_PROVE ``~((a ==> b) ==> (a=b))``                                    2
  handle HolSatLib.SAT_cex th => 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 §7.10.1 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.

7.10.1 `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.

7.10.2 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/HOL-Theorem-Prover/HOL`.

7.10.3 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 §7.10.2), 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

[10] `http://www.satlib.org/Benchmarks/SAT/satformat.ps`

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 `zchaff.cnf` containing a DIMACS-formatted problem, we do:

3

```
> open satConfig;  ...output elided...

> val c = base_config |> set_infile "zchaff.cnf"
                      |> set_solver SatSolvers.zchaff;
val c = ?: sat_config

> GEN_SAT c;
Exception- SAT_cex
  [oracles: DISK_THM] [axioms: ] []
  ⊢ v1 ∧ v5 ∧ v4 ∧ v3 ⇒
    ¬((v1 ∨ ¬v5 ∨ v4) ∧ (¬v1 ∨ v5 ∨ v3 ∨ v4) ∧ (¬v3 ∨ ¬v4)) raised
```

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.

7.10.4 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.

7.11 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 7.10), 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.

7.11.1 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.

7.11.2 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.

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.

```
- load "HolQbfLib";                                                    1
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
```

Figure 7.5: `HolQbfLib` in action.

`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).

7.11.3 Wishlist

The following features have not been implemented yet. Please submit additional feature requests (or code contributions) via `http://github.com/HOL-Theorem-Prover/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`).

7.12 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 7.10), 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).

7.12.1 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).

[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.

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 (`<`, `<=`, `>`, `>=`, 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).

7.12.2 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.

7.12.3 Wishlist

The following features have not been implemented yet. Please submit additional feature requests (or code contributions) via http://github.com/HOL-Theorem-Prover/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

[12] Later versions of Z3 than 2.19 are available for Mac OS X directly, but not supported by HOL.

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 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.

7.13 The `Quantifier Heuristics` library

7.13.1 Motivation

Often interactive proofs can be simplified by instantiating quantifiers. The `Unwind` library, which is part of the simplifier, allows instantiations of "trivial" quantifiers:

$$\forall x_1 \ \ldots x_i \ldots x_n.\ P_1 \wedge \ldots \wedge x_i = c \wedge \ldots \wedge P_n \Rightarrow Q$$

and

$$\exists x_1 \ \ldots x_i \ldots x_n.\ P_1 \wedge \ldots \wedge x_i = c \wedge \ldots \wedge P_n$$

can be simplified by instantiating x_i with c. Because unwind-conversions are part of `bool_ss`, they are used with nearly every call of the simplifier and often simplify proofs considerably. However, the `Unwind` library can only handle these common cases. If the term structure is only slightly more complicated, it fails. For example, $\exists x.\ P(x) \Rightarrow (x = 2) \wedge Q(x)$ cannot be tackled.

There is also the `Satisfy` library, which uses unification to show existentially quantified formulas. It can handle problems like $\exists x.\ P_1(x, c_1) \ \wedge \ldots P_n(x, c_n)$ if given theorems of the form $\forall x\ c.\ P_i(x, c)$. This is often handy, but still rather limited.

The quantifier heuristics library (`quantHeuristicsLib`) provides more power and flexibility. A few simple examples of what it can do are shown in Table 7.2. Besides the power demonstrated by these examples, the library is highly flexible as well. At its core, there is a modular, syntax driven search for instantiation. This search consists

of a collection of interleaved heuristics. Users can easily configure existing heuristics and add own ones. Thereby, it is easy to teach the library about new predicates, logical connectives or datatypes.

Problem	**Result**
basic examples	
$\exists x.\ x = 2 \wedge P(x)$	$P(2)$
$\forall x.\ x = 2 \Rightarrow P(x)$	$P(2)$
solutions and counterexamples	
$\exists x.\ x = 2$	*true*
$\forall x.\ x = 2$	*false*
complicated nestings of standard operators	
$\exists x_1. \forall x_2.\ (x_1 = 2) \wedge P(x_1, x_2)$	$\forall x_2.\ P(2, x_2)$
$\exists x_1, x_2.\ P_1(x_2) \Rightarrow (x_1 = 2) \wedge P(x_1, x_2)$	$\exists x_2.\ P_1(x_2) \Rightarrow P(2, x_2)$
$\exists x.\ ((x = 2) \vee (2 = x)) \wedge P(x)$	$P(2)$
exploiting unification	
$\exists x.\ (f(8+2) = f(x+2)) \wedge P(f(10))$	$P(f(10))$
$\exists x.\ (f(8+2) = f(x+2)) \wedge P(f(x+2))$	$P(f(8+2))$
$\exists x.\ (f(8+2) = f(x+2)) \wedge P(f(x))$	- (no instantiation found)
partial instantiation for datatypes	
$\forall p.\ c = \mathsf{FST}(p) \Rightarrow P(p)$	$\forall p_2.\ P(c, p_2)$
$\forall x.\ \mathsf{IS\_NONE}(x) \vee P(x)$	$\forall x'.\ P(\mathsf{SOME}(x'))$
$\forall l.\ l \neq [\,] \Rightarrow P(l)$	$\forall hd, tl.\ P(hd :: tl)$
context	
$P_1(c) \Rightarrow \exists x.\ P_1(x) \vee P_2(x)$	*true*
$P_1(c) \Rightarrow \forall x.\ \neg P_1(x) \wedge P_2(x)$	$\neg P_1(c)$
$(\forall x.\ P_1(x) \Rightarrow (x = 2)) \Rightarrow (\forall x.\ P_1(x) \Rightarrow P_2(x))$	$(\forall x.\ P_1(x) \Rightarrow (x = 2)) \Rightarrow (P_1(2) \Rightarrow P_2(2))$
$\big((\forall x.\ P_1(x) \Rightarrow P_2(x)) \wedge P_1(2)\big) \Rightarrow \exists x.\ P_2(x)$	*true*

Table 7.2: Examples

7.13.2 User Interface

The quantifier heuristics library can be found in the sub-directory `src/quantHeuristics`. The entry point to the framework is the library `quantHeuristicsLib`.

7.13.2.1 Conversions

Usually the library is used for converting a term containing quantifiers to an equivalent one. For this, the following high level entry points exists:

```
QUANT_INSTANTIATE_CONV     :  quant_param list -> conv
QUANT_INST_ss              :  quant_param list -> ssfrag
QUANT_INSTANTIATE_TAC      :  quant_param list -> tactic
ASM_QUANT_INSTANTIATE_TAC  :  quant_param list -> tactic
```

All these functions get a list of *quantifier heuristic parameters* as arguments. These parameters essentially configure, which heuristics are used during the guess-search.

If an empty list is provided, the tools know about the standard Boolean combinators, equations and context. `std_qp` adds support for common datatypes like pairs or lists. Quantifier heuristic parameters are explained in more detail in Section 7.13.4.

So, some simple usage of the quantifier heuristic library looks like:

1
```
- QUANT_INSTANTIATE_CONV [] ``?x. (!z. Q z /\ (x=7)) /\ P x``;
> val it = |- (?x. (!z. Q z /\ (x = 7)) /\ P x) <=> (!z. Q z) /\ P 7: thm

- QUANT_INSTANTIATE_CONV [std_qp] ``!x. IS_SOME x ==> P x``
> val it = |- (!x. IS_SOME x ==> P x) <=> !x_x'. P (SOME x_x'): thm
```

Usually, the quantifier heuristics library is used together with the simplifier using `QUANT_INST_ss`. Besides interleaving simplification and quantifier instantiation, this has the benefit of being able to use context information collected by the simplifier:

2
```
- QUANT_INSTANTIATE_CONV [] ``P m ==> ?n. P n``
Exception- UNCHANGED raised

- SIMP_CONV (bool_ss ++ QUANT_INST_ss []) [] ``P m ==> ?n. P n``
> val it = |- P m ==> (?n. P n) <=> T: thm
```

It's usually best to use `QUANT_INST_ss` together with e. g. `SIMP_TAC` when using the library with tactics. However, if free variables of the goal should be instantiated, then `ASM_QUANT_INSTANTIATE_TAC` should be used:

3
```
P x
------------------------------------
  IS_SOME x
  : proof

- e (ASM_QUANT_INSTANTIATE_TAC [std_qp])
> P (SOME x_x') : proof
```

There is also `QUANT_INSTANTIATE_TAC`. This tactic does not instantiate free variables. Neither does it take assumptions into consideration. It is just a shortcut for using `QUANT_INSTANTIATE_CONV` as a tactic.

7.13.2.2 Unjustified Guesses

Most heuristics justify the guesses they produce and therefore allow to prove equivalences of e. g. the form $\exists x.\ P(x) \Leftrightarrow P(i)$. However, the implementation also supports unjustified guesses, which may be bogus. Let's consider e. g. the formula $\exists x.\ P(x) \Rightarrow (x = 2) \wedge Q(x)$. Because nothing is known about P and Q, we can't find a safe instantiation for x here. However, 2 looks tempting and is probably sensible in many situations. (Counterexample: $P(2)$, $\neg Q(2)$ and $\neg P(3)$ hold)

implication_concl_qp is a quantifier parameter that looks for valid guesses in the conclusion of an implication. Then, it assumes without justification that these guesses are probably sensible for the whole implication as well. Because these guesses might be wrong, one can either use implications or expansion theorems like $\exists x.\ P(x) \iff (\forall x.\ x\neg c \Rightarrow \neg P(x)) \Rightarrow P(c)$.

4
```
- QUANT_INSTANTIATE_CONV [implication_concl_qp]
     ‘‘?x. P x ==> (x = 2) /\ Q x‘‘
Exception- UNCHANGED raised

- QUANT_INSTANTIATE_CONSEQ_CONV [implication_concl_qp]
     CONSEQ_CONV_STRENGTHEN_direction
     ‘‘?x. P x ==> (x = 2) /\ Q x‘‘
> val it =
   |- (P 2 ==> Q 2) ==> ?x. P x ==> (x = 2) /\ Q x: thm

- EXPAND_QUANT_INSTANTIATE_CONV [implication_concl_qp]
    ‘‘?x. P x ==> (x = 2) /\ Q x‘‘
> val it = |- (?x. P x ==> (x = 2) /\ Q x) <=>
              (!x. x <> 2 ==> ~(P x ==> (x = 2) /\ Q 2)) ==> P 2 ==> Q 2

- SIMP_CONV (std_ss++EXPAND_QUANT_INST_ss [implication_concl_qp]) []
    ‘‘?x. P x ==> (x = 2) /\ Q x‘‘
> val it =
   |- (?x. P x ==> (x = 2) /\ Q x) <=>
      (!x. x <> 2 ==> P x) ==> P 2 ==> Q 2: thm
```

The following entry points should be used to exploit unjustified guesses:

```
QUANT_INSTANTIATE_CONSEQ_CONV  :  quant_param list -> directed_conseq_conv
EXPAND_QUANT_INSTANTIATE_CONV  :  quant_param list -> conv
EXPAND_QUANT_INST_ss           :  quant_param list -> ssfrag
QUANT_INSTANTIATE_CONSEQ_TAC   :  quant_param list -> tactic
```

7.13.2.3 Explicit Instantiations

A special (degenerated) use of the framework, is turning guess search off completely and providing instantiations explicitly. The tactic QUANT_TAC allows this. This means that it allows to partially instantiate quantifiers at subpositions with explicitly given terms. As such, it can be seen as a generalisation of EXISTS_TAC.

5
```
- val it = !x. (!z. P x z) ==> ?a b.    Q a        b z : proof

> e( QUANT_INST_TAC [("z", ‘0‘, []), ("a", ‘SUC a’‘, [‘a’‘])] )
- val it = !x. (    P x 0) ==> ?  b a'. Q (SUC a') b z : proof
```

This tactic is implemented using unjustified guesses. It normally produces implications, which is fine when used as a tactic. There is also a conversion called `INST_QUANT_CONV` with the same functionality. For a conversion, implications are problematic. Therefore, the simplifier and Metis are used to prove the validity of the explicitly given instantiations. This succeeds only for simple examples.

7.13.3 Simple Quantifier Heuristics

The full quantifier heuristics described above are powerful and very flexible. However, they are sometimes slow. The unwind library[13] on the other hand is limited, but fast. The simple version of the quantifier heuristics fills the gap in the middle. They just search for gap guesses without any free variables. Moreover, slow operations like recombining or automatically looking up datatype information is omitted. As a result, the conversion `SIMPLE_QUANT_INSTANTIATE_CONV` (and corresponding `SIMPLE_QUANT_INST_ss`) is nearly as fast as the corresponding unwind conversions. However, it supports more complicated syntax. Moreover, there is support for quantifiers, pairs, list and much more.

7.13.4 Quantifier Heuristic Parameters

Quantifier heuristic parameters play a similar role for the quantifier instantiation library as simpsets do for the simplifier. They contain theorems, ML code and general configuration parameters that allow to configure guess-search. There are predefined parameters that handle common constructs and the user can define own parameters.

7.13.4.1 Quantifier Heuristic Parameters for Common Datatypes

There are `option_qp`, `list_qp`, `num_qp` and `sum_qp` for option types, lists, natural numbers and sum types respectively. Some examples are displayed in the following table:

$$\begin{array}{rcl}
\forall x.\ \texttt{IS_SOME}(x) \Rightarrow P(x) & \Longleftrightarrow & \forall x'.\ P(\texttt{SOME}(x')) \\
\forall x.\ \texttt{IS_NONE}(x) & \Longleftrightarrow & \mathit{false} \\
\forall l.\ l \neq [\,] \Rightarrow P(l) & \Longleftrightarrow & \forall h, l'.\ P(h :: l') \\
\forall x.\ x = c + 3 & \Longleftrightarrow & \mathit{false} \\
\forall x.\ x \neq 0 \Rightarrow P(x) & \Longleftrightarrow & \forall x'.\ P(\texttt{SUC}(x'))
\end{array}$$

7.13.4.2 Quantifier Heuristic Parameters for Tuples

For tuples the situation is peculiar, because each quantifier over a variable of a product type can be instantiated. The challenge is to decide which quantifiers should be instantiated and which new variable names to use for the components of the pair. There is

[13] see `src/simp/src/Unwind.sml`

a quantifier heuristic parameter called `pair_default_qp`. It first looks for subterms of the form $(\lambda(x_1, \ldots, x_n).\ \ldots)\ x$. If such a term is found x is instantiated with $(x_1, \ldots, x_n)$. Otherwise, subterms of the form $\texttt{FST}(x)$ and $\texttt{SND}(x)$ are searched. If such a term is found, x is instantiated as well. This parameter therefore allows simplifications like:

$$\begin{array}{rcl} \forall p.\ (x = \texttt{SND}(p)) \Rightarrow P(p) & \Longleftrightarrow & \forall p_1.\ P(p_1, x) \\ \exists p.\ (\lambda(p_a, p_b, p_c).P(p_a, p_b, p_c))\ p & \Longleftrightarrow & \exists p_a, p_b, p_c.\ P(p_a, p_b, p_c) \end{array}$$

`pair_default_qp` is implemented in terms of the more general quantifier heuristic parameter `pair_qp`, which allows the user to provide a list of ML functions. These functions get the variable and the term. If they return a tuple of variables, these variables are used for the instantiation, otherwise the next function in the list is called or - if there is no function left - the variable is not instantiated. In the example of $\exists p.\ (\lambda(p_a, p_b, p_c).P(p_a, p_b, p_c))\ p$ these functions are given the variable p and the term $(\lambda(p_a, p_b, p_c).P(p_a, p_b, p_c))\ p$ and return $\texttt{SOME}(p_a, p_b, p_c)$. This simple ML-interface gives the user full control over what quantifier over product types to expand and how to name the new variables.

7.13.4.3 Quantifier Heuristic Parameter for Records

Records are similar to pairs, because they can always be instantiated. Here, it is interesting that the necessary monochotomy lemma comes from HOL 4's `Type_Base` library. This means that `record_qp` is stateful. If a new record type is defined, the automatically proven monochotomy lemma is then automatically used by `record_qp`. In contrast to the pair parameter, the one for records gets only one function instead of a list of functions to decide which variables to instantiate. However, this function is simpler, because it just needs to return true or false. The names of the new variables are constructed from the field-names of the record. The quantifier heuristic parameter `default_record_qp` expands all records.

7.13.4.4 Stateful Quantifier Heuristic Parameters

The parameter for records is stateful, as it uses knowledge from `Type_Base`. Such information is not only useful for records but for general datatypes. The quantifier heuristic parameter `TypeBase_qp` uses automatically proven theorems about new datatypes to exploit mono- and dichotomies. Moreover, there is also a stateful `pure_stateful_qp` that allows the user to explicitly add other parameters to it. `stateful_qp` is a combination of `pure_stateful_qp` and `TypeBase_qp`.

7.13.4.5 Standard Quantifier Heuristic Parameter

The standard quantifier heuristic parameter `std_qp` combines the parameters for lists, options, natural numbers, the default one for pairs and the default one for records.

7.13.5 User defined Quantifier Heuristic Parameters

The user is also able to define own parameters. There is `empty_qp`, which does not contain any information. Several parameters can be combined using `combine_qps`. Together with the basic types of user defined parameters that are explained below, these functions provide an interface for user defined quantifier heuristic parameters.

7.13.5.1 Rewrites / Conversions

A very powerful, yet simple technique for teaching the guess search about new constructs are rewrite rules. For example, the standard rules for equations and basic logical operations cannot generate guesses for the predicate `IS_SOME`. By rewriting `IS_SOME(x)` to `?x'. x = SOME(x')`, however, these rules fire.

`option_qp` uses this rewrite to implement support for `IS_SOME`. Similarly support for predicates like `NULL` is implemented using rewrites. Even adding rewrites like $\mathsf{append}(l_1, l_2) = [\,] \Longleftrightarrow (l_1 = [\,] \wedge l_2 = [\,])$ for list-append turned out to be beneficial in practice.

`rewrite_qp` allows to provide rewrites in the form of rewrite theorems. For the example of `IS_SOME` this looks like:

6
```
> val thm = QUANT_INSTANTIATE_CONV [] ``!x. IS_SOME x ==> P x``
Exception- UNCHANGED raised

> val IS_SOME_EXISTS = prove (``IS_SOME x = (?x'. x = SOME x')``,
   Cases_on `x` THEN SIMP_TAC std_ss []);
val IS_SOME_EXISTS = |- IS_SOME x <=> ?x'. x = SOME x': thm

> val thm = QUANT_INSTANTIATE_CONV [rewrite_qp[IS_SOME_EXISTS]]
    ``!x. IS_SOME x ==> P x``
val thm = |- (!x. IS_SOME x ==> P x) <=>
             !x'. IS_SOME (SOME x') ==> P (SOME x'): thm
```

To clean up the result after instantiation, theorems used to rewrite the result after instantiation can be provided via `final_rewrite_qp`.

7
```
> val thm = QUANT_INSTANTIATE_CONV [rewrite_qp[IS_SOME_EXISTS],
                                    final_rewrite_qp[option_CLAUSES]]
      ``!x. IS_SOME x ==> P x``
val thm = |- (!x. IS_SOME x ==> P x) <=> !x'. P (SOME x'): thm
```

If rewrites are not enough, `conv_qp` can be used to add conversions:

8
```
- val thm = QUANT_INSTANTIATE_CONV [] ``?x. (\y. y = 2) x``
Exception- UNCHANGED raised

- val thm = QUANT_INSTANTIATE_CONV [convs_qp[BETA_CONV]] ``?x. (\y. y = 2) x``
> val thm = |- (?x. (\y. y = 2) x) <=> T: thm
```

7.13.5.2 Strengthening / Weakening

In rare cases, equivalences that can be used for rewrites are unavailable. There might be just implications that can be used for strengthening or weakening. The function `imp_qp` might be used to provide such implication.

9

```
- val thm = QUANT_INSTANTIATE_CONV [list_qp] ‘‘!l. 0 < LENGTH l ==> P l‘‘
Exception- UNCHANGED raised

- val LENGTH_LESS_IMP = prove (‘‘!l n. n < LENGTH l ==> l <> []‘‘,
    Cases_on ‘l‘ THEN SIMP_TAC list_ss []);
> val LENGTH_LESS_IMP = |- !l n. n < LENGTH l ==> l <> []: thm

- val thm = QUANT_INSTANTIATE_CONV [imp_qp[LENGTH_LESS_IMP], list_qp]
    ‘‘!l. 0 < LENGTH l ==> P l‘‘
> val thm =
   |- (!l. 0 < LENGTH l ==> P l) <=>
      !l_t l_h. 0 < LENGTH (l_h::l_t) ==> P (l_h::l_t): thm

- val thm = SIMP_CONV (list_ss ++
              QUANT_INST_ss [imp_qp[LENGTH_LESS_IMP], list_qp]) []
              ‘‘!l. SUC (SUC n) < LENGTH l ==> P l‘‘
> val thm =
   |- (!l. SUC (SUC n) < LENGTH l ==> P l) <=>
      !l_h l_t_h l_t_t_t l_t_t_h. n < SUC (LENGTH l_t_t_t) ==>
                                  P (l_h::l_t_h::l_t_t_h::l_t_t_t): thm
```

7.13.5.3 Filtering

Sometimes, one might want to avoid to instantiate certain quantifiers. The function `filter_qp` allows to add ML-functions that filter the handled quantifiers. These functions are given a variable x and a term $P(x)$. The tool only tries to instantiate x in $P(x)$, if all filter functions return *true*.

10

```
- val thm = QUANT_INSTANTIATE_CONV []
     ‘‘?x y z. (x = 1) /\ (y = 2) /\ (z = 3) /\ P (x, y, z)‘‘
> val thm = |- (?x y z. (x = 1) /\ (y = 2) /\ (z = 3) /\ P (x,y,z)) <=>
               P (1,2,3): thm

- val thm = QUANT_INSTANTIATE_CONV
     [filter_qp [fn v => fn t => (v = ‘‘y:num‘‘)]]
     ‘‘?x y z. (x = 1) /\ (y = 2) /\ (z = 3) /\ P (x, y, z)‘‘
> val thm = |- (?x y z. (x = 1) /\ (y = 2) /\ (z = 3) /\ P (x,y,z)) <=>
                ?x   z. (x = 1) /\            (z = 3) /\ P (x,2,z): thm
```

7.13.5.4 Satisfying and Contradicting Instantiations

As the satisfy library demonstrates, it is often useful to use unification and explicitly given theorems to find instantiations. In addition to satisfying instantiations, the quantifier heuristics framework is also able to use contradicting ones. The theorems used for finding instantiations usually come from the context. However, `instantiation_qp` allows to add additional ones:

11
```
> val thm = SIMP_CONV (std_ss++QUANT_INST_ss[]) []
    ``P n ==> ?m:num. n <= m /\ P m``
Exception- UNCHANGED raised

> val thm = SIMP_CONV (std_ss++
               QUANT_INST_ss[instantiation_qp[LESS_EQ_REFL]]) []
               ``P n ==> ?m:num. n <= m /\ P m``
> val thm = |- P n ==> ?m:num. n <= m /\ P m = T : thm
```

7.13.5.5 Di- and Monochotomies

Dichotomies can be exploited for guess search. `distinct_qp` provides an interface to add theorems of the form $\forall x.\ c_1(x) \neq c_2(x)$. `cases_qp` expects theorems of the form $\forall x.\ (x = \exists fv.c_1(fv)) \vee \ldots \vee (x = \exists fv.c_n(fv))$. However, only theorems for $n = 2$ and $n = 1$ are used. All other cases are currently ignored.

7.13.5.6 Oracle Guesses

Sometimes, the user does not want to justify guesses. The tactic `QUANT_TAC` is implemented using oracle guesses for example. A simple interface to oracle guesses is provided by `oracle_qp`. It expects a ML function that given a variable and a term returns a pair of an instantiation and the free variables in this instantiation.

As an example, lets define a parameter that states that every list is non-empty:

```
val dummy_list_qp = oracle_qp (fn v => fn t =>
  let
     val (v_name, v_list_ty) = dest_var v;
     val v_ty = listSyntax.dest_list_type v_list_ty;

     val x = mk_var (v_name ^ "_hd", v_ty);
     val xs = mk_var (v_name ^ "_tl", v_list_ty);
     val x_xs = listSyntax.mk_cons (x, xs)
  in
     SOME (x_xs, [x, xs])
  end)
```

Notice, that an option type is returned and that the function is allowed to throw HOL_ERR exceptions. With this definition, we get

```
- NORE_QUANT_INSTANTIATE_CONSEQ_CONV [dummy_list_qp]                    12
    CONSEQ_CONV_STRENGTHEN_direction ‘‘?x:’a list y:’b. P (x, y)‘‘
> val it = ?y x_hd x_tl. P (x_hd::x_tl,y)) ==> ?x y. P (x,y) : thm
```

7.13.5.7 Lifting Theorems

The function inference_qp enables the user to provide theorems that allow lifting guesses over user defined connectives. As writing these lifting theorems requires deep knowledge about guesses, it is not discussed here. Please have a look at the detailed documentation of the quantifier heuristics library as well as its sources. You might also want to contact Thomas Tuerk (tt291@cl.cam.ac.uk).

7.13.5.8 User defined Quantifier Heuristics

At the lowest level, the tool searches guesses using ML-functions called *quantifier heuristics*. Slightly simplified, such a quantifier heuristic gets a variable and a term and returns a set of guesses for this variable and term. Heuristics allow full flexibility. However, to write your own heuristics a lot of knowledge about the ML-datastructures and auxiliary functions is required. Therefore, no details are discussed here. Please have a look at the source code and contact Thomas Tuerk (tt291@cl.cam.ac.uk), if you have questions. heuristics_qp and top_heuristics_qp provide interfaces to add user defined heuristics to a quantifier heuristics parameter.

7.14 Tree-Structured Finite Sets and Finite Maps

[The current source files are wotScript.sml, totoScript.sml, totoTacs.sml and .sig, enumeralScript.sml, enumTacs.sml and .sig, fmapalScript.sml, fmapalTacs.sml and .sig, tcScript.sml, and tcTacs.sml and .sig. F. L. Morris, flmorris@syr.edu, October 2013]

For any type ty that has been equipped with a total order and a conversion for evaluating it, new terms of type ty set are provided which embody minimum-depth binary search trees. The primary objective has been to supply an IN_CONV for such terms with running time logarithmic in the cost of a single order comparison together with additional set operations which have reasonable running times.

Similarly, for ty as above and any type ty', new terms of type ty |-> ty' embody binary search trees, and enable a logarithmic-time FAPPLY_CONV and various other operations on finite maps.

Total orders: the type `'a toto` The use of binary search trees naturally requires that a total order be supplied for whatever type `ty` of elements [arguments] the sets [finite maps] are to have. Rather than the relation type `ty -> ty -> bool`, it is found computationally advantageous to use the type `ty -> ty -> cpn`, where `cpn` is the HOL datatype of three elements `LESS`, `EQUAL`, `GREATER`. A polymorphic defined type, `'a toto`, has been created isomorphic to the class of functions `: 'a -> 'a -> cpn` satisfying a predicate `totoTheory.TotOrd` which axiomatizes total order-hood. The representation function for the type is called `apto` (for "apply total order"). Specifically what is needed in order to use `ty` as an element [argument] type is (the name of) an element of type `ty toto`, say `tyto`, and a conversion, say `tyto_CONV`, that will reduce terms of the form `apto tyto x y` to one of `LESS`, `EQUAL`, `GREATER`.

Provided in `totoTheory` are orders `numto`, `intto`, `charto`, `stringto`, and `qk_numto` (the last is an unnatural order on type *num* that should in principle be quicker to compute on `NUMERAL` terms than the usual order) with corresponding conversions `numto_CONV`, etc. Also, for lexicographic order on pairs, there are the object language function `lextoto : 'a toto -> 'b toto -> ('a#'b)toto` and the function `lextoto_CONV : conv -> conv -> conv`; if `cva` and `cvb` are conversions for evaluating terms that start `apto toa ...` and `apto tob ...` respectively, `lextoto_CONV cva cvb` is a conversion for evaluating terms starting `apto (toa lextoto tob) ...` . Similarly, there are `listoto : 'a toto -> 'a list toto` and `listoto_CONV : conv -> conv`. Inspection of `listoto` and `listoto_CONV`, possibly also of `qk_numto` and `qk_numto_CONV`, should make it feasible to define orders directly as `toto`'s and corresponding conversions as needed for other HOL datatypes.

Additionally, given any linear order `R : ty -> ty -> bool`, if one supplies the theorem and definition

```
lin_ord_thm:  |- LinearOrder $R
toto_of_dfn:  |- cmp = toto_of_LinearOrder $R
```

and two conversions, say `eq_conv` for reducing equations of ground terms (of `R`'s argument type) to `T` or to `F`, and `lo_conv` for reducing terms `t R t'` to `T` or to `F`, then

```
toto_CONV lin_ord_thm toto_of_dfn eq_conv lo_conv
```

is a corresponding conversion for evaluating terms of the form `apto cmp c c'`.

Interpretation of binary trees as sets; `IN_CONV` The datatype

```
bt = nt | node of 'a bt => 'a => 'a bt
```

is defined with the objective of forming terms `ENUMERAL cmp b`, where `cmp` is a `ty toto` and `b` is a `ty bt`. These should justify the pair of theorems

$\vdash \forall cmp\ y.\ y \in \texttt{ENUMERAL}\ cmp\ \texttt{nt} \iff \texttt{F}$

and

```
⊢ ∀cmp x l y r.
    x ∈ ENUMERAL cmp (node l y r) ⟺
    case apto cmp x y of
      LESS ⇒ x ∈ ENUMERAL cmp l
    | EQUAL ⇒ T
    | GREATER ⇒ x ∈ ENUMERAL cmp r
```

To make these theorems come out true requires the following definition of ENUMERAL:

```
⊢ (∀cmp. ENUMERAL cmp nt = { }) ∧
  ∀cmp l x r.
    ENUMERAL cmp (node l x r) =
    {y | y ∈ ENUMERAL cmp l ∧ (apto cmp y x = LESS)} ∪ {x} ∪
    {z | z ∈ ENUMERAL cmp r ∧ (apto cmp x z = LESS)} .
```

Invoking IN_CONV keyconv ``x IN ENUMERAL cmp b``, where keyconv is a conversion for evaluating applications of cmp, will convert the term to whichever truth value the definition of ENUMERAL compels; that is, to T if and only if top-down tree search discovers x in b. Operations that create ENUMERAL terms, discussed below, will ensure that b is a well-formed binary search tree of minimal depth. (The new IN_CONV, if it is given an equality-deciding conversion and a set built with INSERT rather than a toto-evaluating conversion and an ENUMERAL set, will revert to pred_setLib.IN_CONV.)

Translating between set representations HOL offers two notations for explicit finite sets: the display notation {x1; ... ; xn}, which is short for x1 INSERT ... INSERT xn INSERT {}, and notation with an explicit list: set [x1; ... ; xn]. We provide here conversions back and forth between these:

```
DISPLAY_TO_set_CONV: conv
set_TO_DISPLAY_CONV: conv
```

and to create ENUMERAL sets from either:

```
set_TO_ENUMERAL_CONV: conv -> term -> conv
DISPLAY_TO_ENUMERAL_CONV: conv -> term -> conv
```

(these demanding a toto order on the element type and a conversion for evaluating it), and to recover either from an ENUMERAL:

```
ENUMERAL_TO_set_CONV: conv -> conv
ENUMERAL_TO_DISPLAY_CONV: conv -> conv
```

requiring only the order-evaluating conversion. Additionally,

```
TO_set_CONV: conv -> conv
```

will normalize any of the three forms to the `set [ ... ]` form. (`NO_CONV` will suffice as the conversion argument if it is not an `ENUMERAL` that is to be normalized.) The conversions from the `ENUMERAL` form have to execute, for a well-formed n-element binary search tree, $n-1$ comparisons to verify that all the tree elements belong to the represented set; they will also succeed, in conformance with the definition of `ENUMERAL`, for ill-formed trees, if such are ever created, at the cost of between $2n$ and $3n$ comparisons.

Creation of an `ENUMERAL` form from either of the others entails sorting, performed by an $n \log n$ list-merging algorithm, followed by a detour through another datatype, `'a bl`, which could as well have been `('a # 'a bt) option list`, where the list element of index k, if present, consists of a full binary tree of $2^k - 1$ set elements and one more, which may be thought of as a root of which the full tree is the right subtree. A sorted linear list is copied into a `bl` by successive "`BL_CONS`" operations imitating the incrementation of a binary counter; when the copy is complete, it is collapsed into a single `bt` of which the biggest constituent full `bt` is indeed the right subtree. The upshot is that going from an ordered linear list to the `ENUMERAL` form is a linear-time operation, and that the `bt` in any `ENUMERAL` set has a unique shape for its size: that of a minimal-depth tree with all right subtrees full as one proceeds down the left spine.

Binary operations on sets The functions

```
UNION_CONV: conv -> conv
INTER_CONV: conv -> conv
SET_DIFF_CONV: conv -> conv
```

will work out applications of `UNION`, `INTER`, `DIFF` respectively to two `ENUMERAL` sets, given a conversion to evaluate the relevant order. `UNION_CONV` will revert to `pred_setLib.UNION_CONV` if that is what fits its arguments.

These operations work by list merging, hence with a linear number of comparisons. The strategy is to convert each input, say `ENUMERAL cmp b`, to a theorem,

```
|- OWL cmp (ENUMERAL cmp b) l
```

asserting that it could have been created with `set_TO_ENUMERAL_CONV` from a certain list, which moreover is in strict ascending order:

```
OWL
  |- !cmp s l. OWL cmp s l <=> (s = set l) /\ OL cmp l
OL
  |- (!cmp. OL cmp [] <=> T) /\
     !cmp a l. OL cmp (a::l) <=>
               OL cmp l /\ !p. MEM p l ==> (apto cmp a p = LESS) .
```

Underlying conversions `OWL_UNION`, `OWL_INTER`, `OWL_DIFF`, each of type `conv -> thm -> thm -> thm`, combine two such theorems to produce a third; the result, known to be

ordered, is retransformed into an ENUMERAL term without further comparisons, as in the post-sorting steps of set_TO_ENUMERAL_CONV.

Explicit translations

```
OWL_TO_ENUMERAL: thm -> thm
ENUMERAL_TO_OWL: conv -> term -> thm
set_TO_OWL: conv -> term -> term -> thm
```

between ENUMERAL terms and OWL theorems permit OWL_UNION, etc. to be invoked directly. (set_TO_OWL allows to create an OWL theorem from either a set [...] term or a { ... } term without first making an ENUMERAL.)

In addition, there is

```
SET_EXPR_CONV: conv -> conv
```

which, given a conversion to evaluate the order cmp, will work out the value of any set expression built up with UNION, INTER, and DIFF from ENUMERAL terms, avoiding intermediate translations.

Interpretation of binary trees as finite maps; FAPPLY_CONV The treatment of finite maps parallels that of sets: the new terms denoting maps of type ty |-> ty' are terms FMAPAL cmp b, where cmp is a ty toto and b is a (ty#ty') bt. The theorems supporting tree search are

```
⊢ ∀cmp x. FMAPAL cmp nt ' x = FEMPTY ' x
```

and

```
⊢ ∀cmp x l a b r.
    FMAPAL cmp (node l (a,b) r) ' x =
    case apto cmp x a of
      LESS ⇒ FMAPAL cmp l ' x
    | EQUAL ⇒ b
    | GREATER ⇒ FMAPAL cmp r ' x ,
```

and the definition of FMAPAL is

```
⊢ (∀cmp. FMAPAL cmp nt = FEMPTY) ∧
  ∀x v r l cmp.
    FMAPAL cmp (node l (x,v) r) =
    DRESTRICT (FMAPAL cmp l) {y | apto cmp y x = LESS} FUNION
    FEMPTY |+ (x,v) FUNION
    DRESTRICT (FMAPAL cmp r) {z | apto cmp x z = LESS} .
```

An invocation FAPPLY_CONV keyconv ``(FMAPAL cmp b) ' x``, where keyconv is a conversion to evaluate applications of cmp, will yield the value paired with x if x is in the domain of FMAPAL cmp b, or ``FEMPTY ' x`` if x is not to be found. If FAPPLY_CONV is

instead given an equality-deciding conversion and a term `fmap [ ... ] ' x`, it will produce the first value paired with `x` in the list if any, ``` ``FEMPTY ' x`` ``` if none.

Translating between finite map representations A compact list representation for finite maps is defined:

```
fmap
  |- !l. fmap l = FEMPTY |++ REVERSE l
```

The point of the reversal is that `l` is now treated as an association list—in case of duplicated arguments, the pair nearest the front of the list will take precedence. Translating between `fmap [ ... ]` and `FMAPAL cmp ...` terms for finite maps we have

```
fmap_TO_FMAPAL_CONV: conv -> term -> conv
FMAPAL_TO_fmap_CONV: conv -> conv
```

the first of which demands the name of a `toto`-evaluating function as well as a conversion for computing its value.

Given a term `FUN_FMAP f (set [ ... ])` and a conversion `f_conv` for working out applications of f,

```
FUN_fmap_CONV: conv -> conv
```

will convert the term to the form `fmap [ ... ]`, and

```
FUN_FMAPAL_CONV: conv -> term -> conv -> conv,
```

which expects the conversion and term arguments to `fmap_TO_FMAPAL_CONV` followed by the conversion argument for `FUN_fmap_CONV`, will apply the latter and then the former to yield a `FMAPAL` term.

Binary operations involving finite maps Parallel to the treatment of sets, a theorem representation of a finite map `FMAPAL cmp b`, namely

```
ORWL cmp (FMAPAL cmp b) l,
```

asserting that it corresponds to a certain ordered list, is used for operations involving merging:

```
ORWL
  |- !cmp f l. ORWL cmp f l <=> (f = fmap l) /\ ORL cmp l
ORL
  |- (!cmp. ORL cmp [] <=> T) /\
     !l cmp b a. ORL cmp ((a,b)::l) <=>
           ORL cmp l /\ !p q. MEM (p,q) l ==> (apto cmp a p = LESS) .
```

Functions

```
FMAPAL_TO_ORWL: conv -> term -> thm
ORWL_TO_FMAPAL: thm -> thm
```

translate between FMAPAL terms and ORWL theorems, and the latter can be produced directly from `fmap [ ... ]` terms by

```
fmap_TO_ORWL: conv -> term -> term -> thm
```

which might find use, independent of finite maps, as a list sorting routine.

The only binary operation on FMAPAL terms is

```
FUNION_CONV: conv -> conv
```

which will convert a term ‘‘`FUNION cmp (FMAPAL cmp b) (FMAPAL cmp b')`‘‘ to a FMAPAL term denoting a map defined on the union of the two domains, with the first argument map taking precedence where these overlap. But there are also two forms of domain restriction:

```
DRESTRICT f s
DRESTRICT f (COMP s)
```

for `f` a FMAPAL term and `s` an ENUMERAL term with the same order as `f`. A single conversion

```
DRESTRICT_CONV: conv -> conv
```

will work out either of these forms to a FMAPAL result.

Like UNION_CONV, INTER_CONV, SET_DIFF_CONV, both FUNION_CONV and DRESTRICT_-CONV entail two preliminary computations of FMAPAL_TO_ORWL or ENUMERAL_TO_OWL, a list-merging working part, one of

```
ORWL_FUNION: conv -> thm -> thm -> thm
ORWL_DRESTRICT: conv -> thm -> thm -> thm
ORWL_DRESTRICT_COMPL: conv -> thm -> thm -> thm,
```

and a final use of ORWL_TO_FMAPAL. As with sets, the working parts may be used directly. Alternatively, translations between terms and theorems may be held to a minimum by the use of

```
FMAP_EXPR_CONV: conv -> conv
```

on any expression built up with FUNION, DRESTRICT, and COMP from compatible FMAPAL and ENUMERAL terms.

Other operations on finite maps The following operations on FMAPAL terms have no need to translate to the ORWL theorem form; two of them are unconcerned with the total ordering.

The conversion

```
FDOM_CONV: conv
```

will reduce any term `FDOM (FMAPAL ... )` to the isomorphic `ENUMERAL ...` , also any term `FDOM (fmap [ ... ])` to `set [ ... ]`.

The conversion-valued functon

```
IN_FDOM_CONV: conv -> conv,
```

given a conversion to evaluate the applications of `cmp`, will reduce any term `x IN FDOM (FMAPAL cmp b)` to a truth value, or if given an equality-deciding conversion, it will reduce a term `x IN FDOM (fmap [ ... ])` to a truth value.

The conversion-valued function

```
o_f_CONV: conv -> conv,
```

given a conversion for working out applications of the function `f`, will reduce a term `f o_f (FMAPAL ... )` to an isomorphic `FMAPAL` term, alternatively a term `f o_f fmap [ ... ]` to an isomorphic `fmap` term.

Similarly,

```
FUPDATE_CONV: conv -> conv
```

expects either a `cmp`-evaluating conversion and a term `FMAPAL cmp b |+ (x, y)` or an equality-deciding conversion and a term `fmap [ ... ] |+ (x, y)`, and if `x` is already in the domain of the finite map, will produce an isomorphic structure in which the value paired with `x` has been replaced by `y`. If `x` is not in the domain of the finite map, an error is reported.

It may be noted that `FUN_FMAPAL_CONV`, `FAPPLY_CONV`, and `FUPDATE_CONV` combine to provide a functional array facility with logarithmic number of index comparisons for both reading and writing.

An application: transitive closure The idea of Warshall's algorithm for transitive closure of a relation, to build up an approximation to the answer by repeatedly allowing one fresh element as an intermediate stop in building a path between any two elements, is captured by the definition

$$\begin{aligned}&\texttt{subTC}\ R\ s\ x\ y \iff \\ &\quad R\ x\ y \vee \exists a\ b.\ (R\ \texttt{^|^}\ s)^*\ a\ b \wedge a \in s \wedge b \in s \wedge R\ x\ a \wedge R\ b\ y\end{aligned}$$

where $(R\ \texttt{^|^}\ s)^*$ denotes the reflexive transitive closure of the relation R restricted fore and aft to the set s, and by the theorem (note that it is a set equation, as the Curried relations have been given only one argument)

$$\begin{aligned}&\vdash \forall R\ s\ x\ a.\ \texttt{subTC}\ R\ (x\ \texttt{INSERT}\ s)\ a = \\ &\quad \textbf{if}\ x \in \texttt{subTC}\ R\ s\ a\ \textbf{then}\ \texttt{subTC}\ R\ s\ a \cup \texttt{subTC}\ R\ s\ x\ \textbf{else}\ \texttt{subTC}\ R\ s\ a\ .\end{aligned}$$

A representation of finite relations by set-valued finite maps is defined by

```
FMAP_TO_RELN (f:'a |-> 'a set) x = if x IN FDOM f then f ' x else {}.
```

The conversion-valued function

```
TC_CONV: conv -> conv,
```

given a conversion for evaluating `cmp`, will transform a term

```
(FMAP_TO_RELN (FMAPAL cmp ... ))^+,
```

where the `bt` "..." stores (element, `ENUMERAL`) pairs, to the form `FMAP_TO_RELN (FMAPAL cmp ..... )`, or given an equality-deciding conversion, will turn

```
(FMAP_TO_RELN (fmap [ ... ]))^+,
```

where now "..." is a list of elements paired with { ...} sets, into `FMAP_TO_RELN (fmap [ ..... ])`. It is because `TC_CONV` uses only the operations `TO_set_CONV`, `IN_CONV`, `UNION_CONV`, `FDOM_CONV`, and `o_f_CONV` that it is insensitive to which pair of representations is chosen for finite maps and sets.

As a convenience in preparing the tree-structured representation of a finite relation, the conversion

```
ENUF_CONV: conv -> term -> conv,
```

given a conversion for evaluating a `toto` and its name, will convert a term `fmap [ ... ]` whose list members pair elements with either { ...} sets or `set [ ... ]` sets into a `FMAPAL` term with `ENUMERAL` values.

Chapter 8

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.

8.1 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 always 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. As such theorems are pretty-printed into the corresponding `Theory.sig` file, the help system will find both the declaration in the signature (*e.g.*, `val nm :thm`), and the entry for that theorem in the comment-block.

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

8.2 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).

8.3 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.

8.3.1 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.

8.3.2 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.

8.3.3 Source conventions for script and SML files

Script and theory files The file that generates the HOL theory *my*`Theory` must be called *my*`Script.sml`. After the theory has been successfully generated, it can be `open`-ed at the head of other developments:

```
open myTheory
```

and it can be loaded interactively:

```
load "myTheory";
```

The file `myScript.sml` should begin with the standard boilerplate:

```
open HolKernel Parse boolLib bossLib

val _ = new_theory "my"
```

This "boilerplate" ensures that the standard tactics and SML commands will be in the namespace when the script file is compiled. Interactively, these modules have already been loaded and `open`-ed, so what can be typed directly at `hol` cannot necessarily be included as-is in a script file. In addition, if `myTheory` depends on other HOL theories, this ancestry should also be recorded in the script file. The easiest way to achieve this is simply to `open` the relevant theories. Conventionally, the `open` declarations for such theories appear just before the call to `new_theory`. For example:

```
open HolKernel Parse boolLib bossLib

open myfirstAncestorTheory OtherAncestorTheory

val _ = new_theory "my"
```

Interactively, these may well be the names of theories that have been explicitly loaded into the context with the `load` function. 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.

In addition, simply referring to a theory's theorems using the 'dot-notation' will make that theory an ancestor. For example,

```
Theorem mytheorem:
  ...
Proof
  simp[ThirdAncestorTheory.important_lemma] ...
QED
```

will record a dependency on `ThirdAncestoryTheory`, making it just as much an ancestor as the theories that have been explicitly `open`-ed elsewhere.

Finally, all script files should also end with the invocation:

```
val _ = export_theory()
```

When the script is finally executed, this call writes the theory to disk.

The calls to `new_theory` and `export_theory` must bracket a sequence of SML declarations. A declaration will typically be a `val`-binding, but might also be a function definition (*via* `fun`), an `open`, or even a `structure` declaration. Declarations are *not* expressions. This means that script files should *not* include bare calls to HOL functions like `Datatype`. Instead, declarations such as the following need to be used:

```
val _ = Datatype`tree = Lf | Nd num tree tree`
```

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 generally should) omit the bracketing

```
structure myScript = struct
...
end
```

lines, but the contents of the file are still interpreted as if belonging to a structure.

Finally, take care not to have the string "`Theory`" appear at the end of 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.sml`" or "`Theory.sig`". 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.

Other SML code When developing HOL libraries, one should again attempt to follow Moscow ML's conventions. Most importantly, file names should match `signature` and `structure` names. If this can be done, the automatic dependency analysis done by `Holmake` will work "out of the box". A signature for module `foo` should always appear in file `foo.sig`, and should have the form

```
signature foo =
sig
  ...
end
```

The accompanying implementation of `foo` should appear in file `foo.sml`, and should have the form

```
structure foo :> foo =
struct
...
end
```

As with theory files, the contents of a `structure` must be a sequence of declarations only. Neither sort of file should have any other declarations within it (before or after the `signature` or `structure`).

Deviations from this general pattern are possible, but life is much simpler if such deviations can be avoided. The HOL distribution[1] contains some examples of trickier situations where the guidelines need to be ignored. Ignoring the guidelines will generally result in the need for quite involved `Holmakefiles` (see Section 8.3.7 below).

8.3.4 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,[2] one can develop a theory with a script file that is always separately compilable.

[1] See, for example, the kernel implementation in `src/0`.

[2] There are editor modes for `emacs` and `vim`.

8.3.5 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.[3]

8.3.6 Holmake's command-line arguments

Like make, Holmake takes command-line arguments corresponding to the targets that the user desires to build. As a special case of this, targets ending with the suffix Theory are treated as an instruction to build the theory files that lie behind a HOL theory. (If successful such a build will enable other script-files to refer to that theory, and for interactive sessions to issue load "xTheory"; commands.) If there are no command-line targets, then Holmake will look for a Holmakefile in the current directory. If there is none, or if that file specifies no targets, then Holmake will attempt to build all ML modules and HOL theories it can detect in the current directory. If there is a target in the Holmakefile, then Holmake will try to build the first such target (only).

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 8.3.7 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:

-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.

[3] See *Recursive Make Considered Harmful* by Peter Miller for why this is not ideal.

`--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 8.3.7 below for more on this.

`-I <directory>` Look in specified directory for additional 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.

`--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 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,--noqof` Where q-o-f stands for "quit on failure". By default, if a tactic fails to prove a theorem, the running script exits with a failure. Depending on the presence or absence of the `-k` flag, this failure to build a theory may cause `Holmake` to also exit (with a failure). With the `--noqof` option, `Holmake` will cause the running script to use `mk_thm` to assert the failed goal, allowing the build to continue and other theorems to be 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.

`-r` Forces `Holmake` to behave recursively, overriding the `--no_prereqs` option, and also causing `Holmake` to clean recursively in the "includes" directories (which is not done otherwise).

`--rebuild_deps` 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 error messages such as "Uncaught exception". Such behaviour is a bug, please report it!

8.3.7 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.[4]

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.

[4]It is possible to give a variable a value of pure whitespace by writing

```
NOTHING =
ONE_SPACE = $(NOTHING)␣$(NOTHING)
```

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 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.

In addition, it is possible to indicate that the built-in process of generating theory files from script files generates side products. This is done by writing a command-less rule of the form

```
target : *thyScript.sml
```

where an asterisk character precedes the name of the script file. This indicates that the action of executing the code in thyScript.sml will not only generate the usual thyTheory.sig and thyTheory.sml files, but also the file target. If Holmake is asked to build any of these three files, and any is absent or out of date with respect to thyScript.sml (or any other dependency), then the code in thyScript.sml will be run.

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.

Special targets Some target names for rules are handled specially by Holmake:

- Dependencies associated with the target name .PHONY are taken to be list of other targets in the make-file that are not actually the name of files to be built. For

example, targets naming conceptual collections of files such as `all` should be marked as "phony". If a target is phony, then its dependencies will be built even if a file of that name exists and is newer than the dependencies.

- The special way that command-line arguments `clean`, `cleanAll` and `cleanDeps` are handled means that targets of those names will not work. In order to extend cleaning behaviour, use the `EXTRA_CLEANS` variable (see below).

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`.

`$(which arg)` is replaced by the full path to an executable and readable occurrence of a file called `arg` within a directory in the list of directories in the `PATH` environment variable. For example `$(which cat)` will usually expand to `/bin/cat` on Unix-like systems. If there is no occurrence of `arg` in any directory in `PATH`, this function call expands to the empty string.

`$(wildcard pattern)` expands the shell "glob" pattern (*e.g.*, `*Script.sml`) into the list of matching filenames. If the pattern doesn't match any files, then the function returns `pattern` unchanged.

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 <sigobj>` 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.

By default, directories specified in the `INCLUDES` and `PRE_INCLUDES` directory are also built by `Holmake` before it attempts to build in the current directory. If the `-r` ("force recursion") command-line flag is used, these directories are also "clean"-ed when a cleaning target is given to `Holmake`.

The `CLINE_OPTIONS` variable is used for the specification of command-line switches that are presumably usually appropriate for calls to `Holmake` in the containing directory. The options present in `CLINE_OPTIONS` are used to build a "base environment" of switches;

this base environment is then overridden by whatever was actually passed on the command-line. For example, a useful CLINE_OPTIONS line[5] might be

```
CLINE_OPTIONS = -j1 --noqof
```

Under Poly/ML, the similar POLY_CLINE_OPTIONS variable can be used to pass run-time options to the Poly/ML executable that is run during theory construction.

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.)

DEBUG_FLAG This variable is replaced by "--dbg" if that flag was passed to Holmake, or the empty string if not.

DEFAULT_TARGETS This variable expands to a list of the targets in the current directory that Holmake would build if there was no target in the Holmakefile, and no target was specified on the command-line. Thus, if one wishes to continue to have all these defaults built alongside an additional target, an appropriate idiom to use at the head of the file would be

```
all: $(DEFAULT_TARGETS) mytarget1 mytarget2
.PHONY: all
```

followed by rules for building the new target(s).

HOLDIR The root of the HOL installation.

HOLHEAP Under Poly/ML, this variable expands to the name of the heap that should be used to build this directory (to be used instead of the heap that underlies the hol executable). See Section 8.4 below for more on using custom heaps with Poly/ML.

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).

[5]Note that a --noqof option in a makefile might be overridden from the command-line with the otherwise useless seeming --qof option. In addition, the --no_hmakefile command-line option will stop the makefile from being consulted at all.

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.

KERNELID The kernel option that was passed to HOL's build command, stripped of its leading hyphens. This will typically be stdknl (the standard kernel) but may take on other values if other custom kernels are being used.

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.

Under Poly/ML, commands involving the variable MOSMLC are interpreted "appropriately". If the behaviour is not as desired, we recommend using ifdef POLY (see below)

to write rules that pertain only to HOL under Poly/ML. We strongly discourage the use of MOSMLYAC and MOSMLLEX, even when running HOL under Moscow ML.

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.

8.4 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.

8.4.1 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
```

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 8.1. 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`.

8.4.2 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

```
ifdef POLY
HOLHEAP = realheap
OBJNAMES = polyTheory transcTheory
DEPS = $(patsubst %,$(dprot $(SIGOBJ)/%),$(OBJNAMES))

$(HOLHEAP): $(DEPS)
        $(protect $(HOLDIR)/bin/buildheap) -o $@ $(OBJNAMES)
endif
```

Figure 8.1: 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.

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 when using the `HOLHEAP` variable, heaps are required to be built before everything else in a directory, and that such heaps embody theories or ML sources that are *ancestral* to the directory in which the heap occurs. Thus, if one wanted to

package up a heap embodying the standard theories for the real numbers, and to do it in `src/real` (which feels natural), this heap could be built using the method described here, but could only be referred to as a `HOLHEAP` in the directories that used it, *not* in `src/real`'s `Holmakefile`. Subsequently, developments in other directories could use this heap by specifying

```
$(HOLDIR)/src/real/realheap
```

as the value for their `HOLHEAP` variables.

8.5 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
```

8.6 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 8.6.3 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.

8.6.1 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.[6] 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:

`\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

[6] The output is a mixture of typewriter font and math-mode characters embedded in a `\texttt` block within an `\mbox`.

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 `Datatype`) will be printed. Datatype theorems with these names are automatically generated when `Datatype` is run. If the trace `EmitTeX: print datatypes compactly` is set to 1 (see the `tr` option below) the description is printed in a more compact form. Also, if the type is a collection of nullary constants (a type consisting of only "enumerated constants"), then it will always be printed compactly. When not compact, all of a type's constructors will appear on the same line, or each will be on a separate line.

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 $\vdash \mathtt{F} \wedge t \iff \mathtt{F}$.

`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, and a special form of equality is printed for the top-level ("defining") equality in each clause. 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 =def= 1
FACT (SUC n) =def= SUC n * FACT n
```

If the special equality is not desired, the option `nodefsym` can be used to turn this off. The special equality symbol can also be redefined by changing the LaTeX definition of the macro `\HOLTokenDefEquality`.

`depth=`*n* Causes printing to be done with a maximum print depth of *n*; see Section 7.1.2.10.

`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 5.2.2), 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.

`m`*space*`,nomath` The `m` option makes HOL material be typeset in "math-mode". In particular, the output of the pretty-printer will be modified so that newline characters are replaced by `\\` commands. This then requires that the surrounding LaTeX environment be array-like, so that the `\\` command will have the desired effect.

In addition, because raw spaces have minimal effect in math-mode (something like `f␣x` will be typeset as fx), math-mode munging also replaces spaces with math-mode macros. By default, the command `\;\;` is used, but if the `m` option is followed by some characters, each is interpreted as a single-letter macro name, with each macro concatenated together to provide the space command that will be used.

For example, if the option is `m;`, then the spacing command will be `\;`. If the option is `m;!`, then the spacing command will be `\;\!`. The comma character cannot be used because it conflicts with parsing the list of options, but one can use `c` instead, so that the option `mc` will make the spacing command be `\,`.

The `m` option can be installed globally with the `-m` command-line option. If this option is enabled globally, it can be cancelled on a case-by-case basis by using the `nomath` option. The `nomath` option also takes precedence over any `m` options that might occur.

See also the discussion about math-mode munging in Section 8.6.2 below.

`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 7.1.2.1. 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). For controlling how the turnstile is printed when this option is not present, see the paragraph on Overrides in Section 8.6.4.

`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).

For adding a name to the rule, see the `rulename` option below.

`rulename=`*name* (Only has an effect with `rule` or `stackedrule`.) Adds *name* as the optional argument to the `\infer` command when typesetting the rule. The name is wrapped with `\HOLRuleName`, which by default is the same as `\textsf`. For ease of parsing options, *name* should not contain braces, brackets, or commas. (A name including such special characters could be typeset by renewing the `\HOLRuleName` command.)

`showtypes`*n* (For use with `\HOLthm` and `\HOLtm`.) Causes the term or theorem to be printed with the `types` trace set to level *n*. The *n* is optional and defaults to 1 if omitted (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

$$\frac{\begin{array}{c} p \vee q \\ p \Rightarrow r \\ q \Rightarrow r \end{array}}{r}$$

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.

For adding a name to the rule, see the `rulename` option.

`tr'`*tracename*`'=`*n* This option allows the temporary setting of the provided trace to the integer value *n*. For example, one can set `pp_unambiguous_comprehensions` to 1 to ensure that set comprehensions are printed with bound variables explicitly identified. See Section 5.5.1.1 for more on set comprehensions, and Section 8.2 for more on traces.

`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 8.6.4 below).

`-`*name* This option causes the printing of the term or theorem to be done with respect to a grammar that has all overloading for *name* removed. When used with `\HOLty`, prints the type with all type abbreviations for *name* removed. For example, the command `\HOLtm[-+]{x + y}` will print as

`arithmetic$+` *x y*

because the underlying constant will no longer map to the string `"+"` and, in the absence of any other mappings for it, will be printed as a fully qualified name.

If the theory of integers is loaded, then the command `\HOLtm[-+]{x + y:int}` will print as

`int_add` *x y*

because the mapping from the integer addition constant to `"+"` is removed, but the mapping to `"int_add"` remains, allowing that form to be what is printed.

The `-` option can be useful when complicated notation involving overloads is first introduced in a document.

`>>` **and** `>>~` Indents the argument. These options only make 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.

The version with the tilde character (`~`) does not add indentation to the first line of output, but adds the specified amount (again 2, if no number is provided) to subsequent lines. This allows one to achieve suitable alignment when other non-HOL text has been put onto the same line. For example,

```
AND_CLAUSES \HOLthm[width=46,>>~12]{bool.AND_CLAUSES}
TRUTH       \HOLthm[>>~12]{bool.TRUTH}
MAP         \HOLthm[>>~12,width=50]{list.MAP}
```

ensures correct vertical alignment when extra lines are printed, as they will be with the printing of `bool.AND_CLAUSES` and `list.MAP`.

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.

$s//t$ (For use with `\HOLtm`, `\HOLthm`, and `\HOLty`) Causes LaTeX string s to be substituted for token t. This allows one-off manipulation of the override map (see Section 8.6.4 below). The difference between this operation and the "normal substitution" done with a single slash (as above) is that it happens as the HOL entity is printed, whereas normal substitution happens before pretty-printing is done. If printing depends on particular variable name choices, the "last minute" manipulations possible with this form of substitution may be preferable. The width of the LaTeX string is taken to be the width of the original token t.

8.6.2 Math-mode munging

There are a few steps needed to make math-mode munging a relatively painless affair. First, there are two LaTeX macros from `holtexbasic.sty` that should probably be overridden:

`\HOLConst` By default this will print names in typewriter font. In math mode, this will probably look better in sans serif, suggesting

```
\renewcommand{\HOLConst}[1]{\textsf{#1}}
```

Depending on personal taste, the `\HOLKeyword` macro might be redefined similarly. This macro is used for keywords such as `if`.

`\HOLinline` This macro, used to wrap standard `\HOLtm` arguments, puts text into typewriter font. One possibility for its redefinition would be

```
\renewcommand{\HOLinline}[1]{\ensuremath{#1}}
```

Note that if the term being typeset causes the pretty-printer to break over multiple lines, LaTeX will complain because of the appearance of `\\` commands. If necessary, this can be avoided on a case-by-case basis by setting the `width` option to a larger than normal width.

When using math-mode munging, one also has to be aware of how larger pieces of text will appear. In non-math-mode munging, material is usually put into `alltt` environments. The recommended alternative for math-mode is to use the `\HOLmath` environment:

```
article text

\begin{HOLmath}
\HOLthm{bool.AND_CLAUSES}
\end{HOLmath}
```

This uses a standard `array` environment within a `displaymath`.

Occasionally, one will want to arrange blocks of HOL material within a larger math context. The `HOLarray` environment is a simple alias for a single-column left-aligned array that one can use in these situations.

8.6.3 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.

Under Poly/ML, the `mkmunge.exe` executable also takes an optional `-b` option that can be used to specify a heap (see Section 8.4) to use as a base. Doing so allows for the incorporation of many theories at once, and will be more efficient than loading the heap's theories separately on top of the default HOL heap. The use of a base heap argument to `mkmunge.exe` doesn't affect the efficiency of the resulting munging tool.

8.6.4 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 a number of 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 8.6.6 below).

By overriding the special token `$Turnstile$`, one can control the printing of the turnstile produced by `\HOLthm`. The default setup is roughly equivalent to overriding `$Turnstile$` to `\HOLTokenTurnstile{}` followed by a space, giving a total width of 3. Overriding the turnstile in this way may will probably be necessary in math-mode printing, where the turnstile character is typically of the same width as 5 `\;` invocations. Providing the correct width is important in order to get lines past the first to line up with the left edge of the mathematical text rather than the turnstile.

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.

Setting Math-mode Spacing If one expects to include all of the various `\HOL` commands in LaTeX math contexts (as described above), then the `-m` option both sets the default width for math-mode spaces, and also enables math-mode typesetting by default.

The specification of spacing is with a string of characters, as already described. Note that if the command-line option includes any semi-colons or exclamation marks (*e.g.*, `-mc;`), then they need to be quoted to prevent the shell from getting confused. If the `-m` option appears without any additional characters, the default math-mode spacing will be `\;\;`.

8.6.5 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 formatting 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 formatting options for these new identifiers as well as the old ones. HOL definition files can be used for defining and setting the formatting 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 formatting 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*`. For example, the id `arithmetic.LESS_EQ*` represents all theorems 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 its HOL definition. For theorems it is of the form `theory.theorem`.

`options` formatting options for the munger as described in section 8.6.1. Please use the Holindex commands for typesetting 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 its 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 formatting 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.

8.6.6 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] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics Series. Academic Press, 1986.

[2] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[3] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[4] 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.

[5] John Harrison. *Theorem-proving with the Real Numbers*. CPHC/BCS Distinguished Dissertations. Springer, 1998.

[6] Joe Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge, 2002.

[7] A. C. Leisenring. *Mathematical Logic and Hilbert's ϵ-Symbol*. University Mathematical Series. Macdonald & Co. Ltd., London, 1969.

[8] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 35–46, New York, NY, USA, 2008. ACM.

[9] 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.

[10] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[11] L. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[12] L. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[13] Konrad Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Technical University of Munich, 1999.

Index