The HOL System
DESCRIPTION

University of Cambridge DSTO SRI International

Preface

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

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

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

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

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

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

The HOL system is designed to support interactive theorem proving in higher order logic (hence the acronym 'HOL'). To this end, the formal logic is interfaced to a general purpose programming language (ML, for meta-language) in which terms and theorems of the logic can be denoted, proof strategies expressed and applied, and logical theories developed. 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 [**?**]. 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 [**?**].) However, the logic does not restrict applications to hardware; HOL has been applied to many other areas.

This document presents the HOL logic, and it explains the means by which meta-language functions can be used to generate proofs in the logic.

The approach to mechanizing formal proof used in HOL is due to Robin Milner [**?**], 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 [**?**] by Konrad Slind at the University of Calgary. It has been further developed through the 1990's. HOL98 is the latest version of HOL, and is also implemented in SML; it features a number of novelties compared to its predecessors. HOL98 is intended to serve as a stable platform for a number of research projects and technology transfer activities that are in progress at Cambridge, and elsewhere, at the time of writing. It is also the supported version of the system for the international HOL community. The main differences between the various versions and releases of HOL are described in Appendix **??**.

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

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.

Later editions

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

The third edition of all three volumes represents a wide-ranging and still incomplete revision of material written for HOL88 so that it applies to the hol98 system a decade later. The third edition has been prepared by Konrad Slind and Michael Norrish.

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

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

Contents

Part I

The HOL Logic

Chapter 1

Syntax and Semantics

1.1 Introduction

This chapter describes the syntax and set-theoretic semantics of the logic supported by the HOL system, which is a variant of Church's simple theory of types [**?**] and will henceforth be called the HOL logic, or just HOL. The meta-language for this description will be English, enhanced with various mathematical notations and conventions. The object language of this description is the HOL logic. Note that there is a 'meta-language', in a different sense, associated with the HOL logic, namely the programming language ML. This is the language used to manipulate the HOL logic by users of the system. It is hoped that because of context, no confusion results from these two uses of the word 'meta-language'. When ML is the object of study (as in [**?**]), ML is the object language under consideration—and English is again the meta-language!

The HOL syntax contains syntactic categories of types and terms whose elements are intended to denote respectively certain sets and elements of sets. This set theoretic interpretation will be developed along side the description of the HOL syntax, and in the next chapter the HOL proof system will be shown to be sound for reasoning about properties of the set theoretic model.[1] This model is given in terms of a fixed set of sets $\mathcal{U}$, which will be called the *universe* and which is assumed to have the following properties.

Inhab Each element of $\mathcal{U}$ is a non-empty set.

Sub If $X \in \mathcal{U}$ and $\emptyset \neq Y \subseteq X$, then $Y \in \mathcal{U}$.

Prod If $X \in \mathcal{U}$ and $Y \in \mathcal{U}$, then $X \times Y \in \mathcal{U}$. The set $X \times Y$ is the cartesian product, consisting of ordered pairs (x, y) with $x \in X$ and $y \in Y$, with the usual set-theoretic coding of ordered pairs, viz. $(x, y) = \{\{x\}, \{x, y\}\}$.

Pow If $X \in \mathcal{U}$, then the powerset $P(X) = \{Y : Y \subseteq X\}$ is also an element of $\mathcal{U}$.

Infty $\mathcal{U}$ contains a distinguished infinite set I.

[1] There are other, 'non-standard' models of HOL, which will not concern us here.

Choice There is a distinguished element $\mathrm{ch} \in \prod_{X \in \mathcal{U}} X$. The elements of the product $\prod_{X \in \mathcal{U}} X$ are (dependently typed) functions: thus for all $X \in \mathcal{U}$, X is non-empty by **Inhab** and $\mathrm{ch}(X) \in X$ witnesses this.

There are some consequences of these assumptions which will be needed. In set theory functions are identified with their graphs, which are certain sets of ordered pairs. Thus the set $X \to Y$ of all functions from a set X to a set Y is a subset of $P(X \times Y)$; and it is a non-empty set when Y is non-empty. So **Sub**, **Prod** and **Pow** together imply that $\mathcal{U}$ also satisfies

Fun If $X \in \mathcal{U}$ and $Y \in \mathcal{U}$, then $X \to Y \in \mathcal{U}$.

By iterating **Prod**, one has that the cartesian product of any finite, non-zero number of sets in $\mathcal{U}$ is again in $\mathcal{U}$. $\mathcal{U}$ also contains the cartesian product of no sets, which is to say that it contains a one-element set (by virtue of **Sub** applied to any set in $\mathcal{U}$—**Infty** guarantees there is one); for definiteness, a particular one-element set will be singled out.

Unit $\mathcal{U}$ contains a distinguished one-element set $1 = \{0\}$.

Similarly, because of **Sub** and **Infty**, $\mathcal{U}$ contains two-element sets, one of which will be singled out.

Bool $\mathcal{U}$ contains a distinguished two-element set $2 = \{0, 1\}$.

The above assumptions on $\mathcal{U}$ are weaker than those imposed on a universe of sets by the axioms of Zermelo-Fraenkel set theory with the Axiom of Choice (ZFC), principally because $\mathcal{U}$ is not required to satisfy any form of the Axiom of Replacement. Indeed, it is possible to prove the existence of a set $\mathcal{U}$ with the above properties from the axioms of ZFC. (For example one could take $\mathcal{U}$ to consist of all non-empty sets in the von Neumann cumulative hierarchy formed before stage $\omega + \omega$.) Thus, as with many other pieces of mathematics, it is possible in principal to give a completely formal version within ZFC set theory of the semantics of the HOL logic to be given below.

1.2 Types

The types of the HOL logic are expressions that denote sets (in the universe $\mathcal{U}$). Following tradition, σ, possibly decorated with subscripts or primes, is used to range over arbitrary types.

There are four kinds of types in the HOL logic. These can be described informally by the following BNF grammar, in which α ranges over type variables, *c* ranges over atomic types and *op* ranges over type operators.

$$\sigma \quad ::= \quad \underset{\substack{\uparrow \\ \text{type variables}}}{\alpha} \quad | \quad \underset{\substack{\uparrow \\ \text{atomic types}}}{c} \quad | \quad \underbrace{(\sigma_1,\ldots,\sigma_n)op}_{\text{compound types}} \quad | \quad \underbrace{\sigma_1\to\sigma_2}_{\substack{\text{function types} \\ \text{(domain } \sigma_1\text{, range } \sigma_2\text{)}}}$$

In more detail, the four kinds of types are as follows.

1. **Type variables:** these stand for arbitrary sets in the universe. In Church's original formulation of simple type theory, type variables are part of the meta-language and are used to range over object language types. Proofs containing type variables were understood as proof schemes (i.e. families of proofs). To support such proof schemes *within* the HOL logic, type variables have been added to the object language type system.[2]

2. **Atomic types:** these denote fixed sets in the universe. Each theory determines a particular collection of atomic types. For example, the standard atomic types *bool* and *ind* denote, respectively, the distinguished two-element set 2 and the distinguished infinite set I.

3. **Compound types:** These have the form $(\sigma_1,\ldots,\sigma_n)op$, where σ_1, …, σ_n are the argument types and op is a *type operator* of arity n. Type operators denote operations for constructing sets. The type $(\sigma_1,\ldots,\sigma_n)op$ denotes the set resulting from applying the operation denoted by op to the sets denoted by σ_1, …, σ_n. For example, *list* is a type operator with arity 1. It denotes the operation of forming all finite lists of elements from a given set. Another example is the type operator *prod* of arity 2 which denotes the cartesian product operation. The type $(\sigma_1,\sigma_2)prod$ is written as $\sigma_1 \times \sigma_2$.

4. **Function types:** If σ_1 and σ_2 are types, then $\sigma_1\to\sigma_2$ is the function type with *domain* σ_1 and *range* σ_2. It denotes the set of all (total) functions from the set denoted by its domain to the set denoted by its range. (In the literature $\sigma_1\to\sigma_2$ is written without the arrow and backwards—i.e. as $\sigma_2\sigma_1$.) Note that syntactically $\to$ is simply a distinguished type operator of arity 2 written with infix notation. It is singled out in the definition of HOL types because it will always denote the same operation in any model of a HOL theory—in contrast to the other type operators which may be interpreted differently in different models. (See Section 1.2.2.)

It turns out to be convenient to identify atomic types with compound types constructed with 0-ary type operators. For example, the atomic type *bool* of truth-values can be regarded as being an abbreviation for $()bool$. This identification will be made

[2] This technique was invented by Robin Milner for the object logic PPλ of his LCF system.

in the technical details that follow, but in the informal presentation atomic types will continue to be distinguished from compound types, and $()c$ will still be written as c.

1.2.1 Type structures

The term 'type constant' is used to cover both atomic types and type operators. It is assumed that an infinite set $\mathsf{TyNames}$ of the *names of type constants* is given. The greek letter ν is used to range over arbitrary members of $\mathsf{TyNames}$, *c* will continue to be used to range over the names of atomic types (i.e. 0-ary type constants), and *op* is used to range over the names of type operators (i.e. n-ary type constants, where $n > 0$).

It is assumed that an infinite set TyVars of *type variables* is given. Greek letters $\alpha, \beta, \ldots$, possibly with subscripts or primes, are used to range over Tyvars. The sets $\mathsf{TyNames}$ and TyVars are assumed disjoint.

A *type structure* is a set Ω of type constants. A *type constant* is a pair (ν, n) where $\nu \in \mathsf{TyNames}$ is the name of the constant and n is its arity. Thus $\Omega \subseteq \mathsf{TyNames} \times \mathsf{N}$ (where N is the set of natural numbers). It is assumed that no two distinct type constants have the same name, i.e. whenever $(\nu, n_1) \in \Omega$ and $(\nu, n_2) \in \Omega$, then $n_1 = n_2$.

The set Types_Ω of types over a structure Ω can now be defined as the smallest set such that:

- $\mathsf{TyVars} \subseteq \mathsf{Types}_\Omega$.
- If $(\nu, 0) \in \Omega$ then $()\nu \in \mathsf{Types}_\Omega$.
- If $(\nu, n) \in \Omega$ and $\sigma_i \in \mathsf{Types}_\Omega$ for $1 \leq i \leq n$, then $(\sigma_1, \ \ldots \ , \sigma_n)\nu \in \mathsf{Types}_\Omega$.
- If $\sigma_1 \in \mathsf{Types}_\Omega$ and $\sigma_2 \in \mathsf{Types}_\Omega$ then $\sigma_1 \to \sigma_2 \in \mathsf{Types}_\Omega$.

The type operator $\to$ is assumed to associate to the right, so that

$$\sigma_1 \to \sigma_2 \to \ldots \to \sigma_n \to \sigma$$

abbreviates

$$\sigma_1 \to (\sigma_2 \to \ldots \to (\sigma_n \to \sigma) \ldots)$$

The notation $tyvars(\sigma)$ is used to denote the set of type variables occurring in σ.

1.2.2 Semantics of types

A *model* M of a type structure Ω is specified by giving for each type constant (ν, n) an n-ary function

$$M(\nu) : \mathcal{U}^n \longrightarrow \mathcal{U}$$

Thus given sets $X_1, \ldots, X_n$ in the universe $\mathcal{U}$, $M(\nu)(X_1, \ldots, X_n)$ is also a set in the universe. In case $n = 0$, this amounts to specifying an element $M(\nu) \in \mathcal{U}$ for the atomic type ν.

Types containing no type variables are called *monomorphic*, whereas those that do contain type variables are called *polymorphic*. What is the meaning of a polymorphic type? One can only say what set a polymorphic type denotes once one has instantiated its type variables to particular sets. So its overall meaning is not a single set, but is rather a set-valued function, $\mathcal{U}^n \longrightarrow \mathcal{U}$, assigning a set for each particular assignment of sets to the relevant type variables. The arity n corresponds to the number of type variables involved. It is convenient in this connection to be able to consider a type variable to be involved in the semantics of a type σ whether or not it actually occurs in σ, leading to the notion of a type-in-context.

A *type context*, αs, is simply a finite (possibly empty) list of *distinct* type variables $\alpha_1, \ldots, \alpha_n$. A *type-in-context* is a pair, written $\alpha s.\sigma$, where αs is a type context, σ is a type (over some given type structure) and all the type variables occurring in σ appear somewhere in the list αs. The list αs may also contain type variables which do not occur in σ.

For each σ there are minimal contexts αs for which $\alpha s.\sigma$ is a type-in-context, which only differ by the order in which the type variables of σ are listed in αs. In order to select one such context, let us assume that TyVars comes with a fixed total order and define the *canonical* context of the type σ to consist of exactly the type variables it contains, listed in order.[3]

Let M be a model of a type structure Ω. For each type-in-context $\alpha s.\sigma$ over Ω, define a function

$$[\![\alpha s.\sigma]\!]_M : \mathcal{U}^n \longrightarrow \mathcal{U}$$

(where n is the length of the context) by induction on the structure of σ as follows.

- If σ is a type variable, it must be α_i for some unique $i = 1, \ldots, n$ and then $[\![\alpha s.\sigma]\!]_M$ is the ith projection function, which sends $(X_1, \ldots, X_n) \in \mathcal{U}^n$ to $X_i \in \mathcal{U}$.

- If σ is a function type $\sigma_1 \to \sigma_2$, then $[\![\alpha s.\sigma]\!]_M$ sends $Xs \in \mathcal{U}^n$ to the set of all functions from $[\![\alpha s.\sigma_1]\!]_M(Xs)$ to $[\![\alpha s.\sigma_2]\!]_M(Xs)$. (This makes use of the property **Fun** of $\mathcal{U}$.)

- If σ is a compound type $(\sigma_1, \ldots, \sigma_m)\nu$, then $[\![\alpha s.\sigma]\!]_M$ sends Xs to $M(\nu)(S_1, \ldots, S_m)$ where each S_j is $[\![\alpha s.\sigma_j]\!]_M(Xs)$.

One can now define the meaning of a type σ in a model M to be the function

$$[\![\sigma]\!]_M : \mathcal{U}^n \longrightarrow \mathcal{U}$$

[3] It is possible to work with unordered contexts, specified by finite sets rather than lists, but we choose not to do that since it mildly complicates the definition of the semantics to be given below.

given by $[\![\alpha s.\sigma]\!]_M$, where αs is the canonical context of σ. If σ is monomorphic, then $n = 0$ and $[\![\sigma]\!]_M$ can be identified with the element $[\![\sigma]\!]_M()$ of $\mathcal{U}$. When the particular model M is clear from the context, $[\![\_]\!]_M$ will be written $[\![\_]\!]$.

To summarize, given a model in $\mathcal{U}$ of a type structure Ω, the semantics interprets monomorphic types over Ω as sets in $\mathcal{U}$ and more generally, interprets polymorphic types involving n type variables as n-ary functions $\mathcal{U}^n \longrightarrow \mathcal{U}$ on the universe. Function types are interpreted by full function sets.

Examples Suppose that Ω contains a type constant $(b, 0)$ and that the model M assigns the set 2 to b. Then:

1. $[\![b{\to}b{\to}b]\!] = 2{\to}2{\to}2 \in \mathcal{U}$.

2. $[\![(\alpha{\to}b){\to}\alpha]\!] : \mathcal{U} \longrightarrow \mathcal{U}$ is the function sending $X \in \mathcal{U}$ to $(X{\to}2){\to}X \in \mathcal{U}$.

3. $[\![\alpha, \beta.(\alpha{\to}b){\to}\alpha]\!] : \mathcal{U}^2 \longrightarrow \mathcal{U}$ is the function sending $(X, Y) \in \mathcal{U}^2$ to $(X{\to}2){\to}X \in \mathcal{U}$.

Remark A more traditional approach to the semantics would involve giving meanings to types in the presence of 'environments' assigning sets in $\mathcal{U}$ to all type variables. The use of types-in-contexts is almost the same as using partial environments with finite domains—it is just that the context ties down the admissible domain to a particular finite (ordered) set of type variables. At the level of types there is not much to choose between the two approaches. However for the syntax and semantics of terms to be given below, where there is a dependency both on type variables and on individual variables, the approach used here seems best.

1.2.3 Instances and substitution

If σ and $\tau_1, \ldots, \tau_n$ are types over a type structure Ω,

$$\sigma[\tau_1, \ldots, \tau_p/\beta_1, \ldots, \beta_p]$$

will denote the type resulting from the simultaneous substitution for each $i = 1, \ldots, p$ of τ_i for the type variable β_i in σ. The resulting type is called an *instance* of σ. The following lemma about instances will be useful later; it is proved by induction on the structure of σ.

Lemma 1 *Suppose that σ is a type containing distinct type variables $\beta_1, \ldots, \beta_p$ and that $\sigma' = \sigma[\tau_1, \ldots, \tau_n/\beta_1, \ldots, \beta_p]$ is an instance of σ. Then the types $\tau_1, \ldots, \tau_p$ are uniquely determined by σ and σ'.*

We also need to know how the semantics of types behaves with respect to substitution:

Lemma 2 *Given types-in-context $\beta s.\sigma$ and $\alpha s.\tau_i$ ($i = 1, \ldots, p$, where p is the length of βs), let σ' be the instance $\sigma[\tau s/\beta s]$. Then $\alpha s.\sigma'$ is also a type-in-context and its meaning in any model M is related to that of $\beta s.\sigma$ as follows. For all $Xs \in \mathcal{U}^n$ (where n is the length of αs)*

$$[\![\alpha s.\sigma']\!](Xs) = [\![\beta s.\sigma]\!]([\![\alpha s.\tau_1]\!](Xs), \ldots, [\![\alpha s.\tau_p]\!](Xs))$$

Once again, the lemma can be proved by induction on the structure of σ.

1.3 Terms

The terms of the HOL logic are expressions that denote elements of the sets denoted by types. The meta-variable t is used to range over arbitrary terms, possibly decorated with subscripts or primes.

There are four kinds of terms in the HOL logic. These can be described approximately by the following BNF grammar, in which x ranges over variables and c ranges over constants.

$$t \quad ::= \quad \underset{\text{variables}}{x} \quad | \quad \underset{\text{constants}}{\mathsf{c}} \quad | \quad \underbrace{t\ t'}_{\substack{\text{function applications}\\ \text{(function } t\text{, argument } t')}} \quad | \quad \underbrace{\lambda x.\ t}_{\lambda\text{-abstractions}}$$

Informally, a λ-term $\lambda x.\ t$ denotes a function $v \mapsto t[v/x]$, where $t[v/x]$ denotes the result of substituting v for x in t. An application $t\ t'$ denotes the result of applying the function denoted by t to the value denoted by t'. This will be made more precise below.

The BNF grammar just given omits mention of types. In fact, each term in the HOL logic is associated with a unique type. The notation t_σ is traditionally used to range over terms of type σ. A more accurate grammar of terms is:

$$t_\sigma \quad ::= \quad x_\sigma \quad | \quad \mathsf{c}_\sigma \quad | \quad (t_{\sigma' \to \sigma}\ t'_{\sigma'})_\sigma \quad | \quad (\lambda x_{\sigma_1}.\ t_{\sigma_2})_{\sigma_1 \to \sigma_2}$$

In fact, just as the definition of types was relative to a particular type structure Ω, the formal definition of terms is relative to a given collection of typed constants over Ω. Assume that an infinite set Names of names is given. A *constant* over Ω is a pair (c, σ), where $\mathsf{c} \in \mathsf{Names}$ and $\sigma \in \mathsf{Types}_\Omega$. A *signature* over Ω is just a set Σ_Ω of such constants.

The set $\mathsf{Terms}_{\Sigma_\Omega}$ of terms over Σ_Ω is defined to be the smallest set closed under the following rules of formation:

1. **Constants:** If $(\mathsf{c}, \sigma) \in \Sigma_\Omega$ and $\sigma' \in \mathsf{Types}_\Omega$ is an instance of σ, then $(\mathsf{c}, \sigma') \in \mathsf{Terms}_{\Sigma_\Omega}$. Terms formed in this way are called *constants* and are written $\mathsf{c}_{\sigma'}$.

2. **Variables:** If $x \in \mathsf{Names}$ and $\sigma \in \mathsf{Types}_\Omega$, then $\mathtt{var}\ x_\sigma \in \mathsf{Terms}_{\Sigma_\Omega}$. Terms formed in this way are called *variables*. The marker var is purely a device to distinguish variables from constants with the same name. A variable $\mathtt{var}\ x_\sigma$ will usually be written as x_σ, if it is clear from the context that x is a variable rather than a constant.

3. **Function applications:** If $t_{\sigma' \to \sigma} \in \mathsf{Terms}_{\Sigma_\Omega}$ and $t'_{\sigma'} \in \mathsf{Terms}_{\Sigma_\Omega}$, then $(t_{\sigma' \to \sigma}\ t'_{\sigma'})_\sigma \in \mathsf{Terms}_{\Sigma_\Omega}$. (Terms formed in this way are sometimes called *combinations*.)

4. **λ-Abstractions:** If $\mathtt{var}\ x_{\sigma_1} \in \mathsf{Terms}_{\Sigma_\Omega}$ and $t_{\sigma_2} \in \mathsf{Terms}_{\Sigma_\Omega}$, then $(\lambda x_{\sigma_1}.\ t_{\sigma_2})_{\sigma_1 \to \sigma_2} \in \mathsf{Terms}_{\Sigma_\Omega}$.

Note that it is possible for constants and variables to have the same name. It is also possible for different variables to have the same name, if they have different types.

The type subscript on a term may be omitted if it is clear from the structure of the term or the context in which it occurs what its type must be.

Function application is assumed to associate to the left, so that $t\ t_1\ t_2\ \ldots\ t_n$ abbreviates $(\ \ldots\ ((t\ t_1)\ t_2)\ \ldots\ t_n)$.

The notation $\lambda x_1\ x_2\ \cdots\ x_n.\ t$ abbreviates $\lambda x_1.\ (\lambda x_2.\ \cdots\ (\lambda x_n.\ t)\ \cdots)$.

A term is called polymorphic if it contains a type variable. Otherwise it is called monomorphic. Note that a term t_σ may be polymorphic even though σ is monomorphic—for example, $(f_{\alpha \to b}\ x_\alpha)_b$, where b is an atomic type. The expression $tyvars(t_\sigma)$ denotes the set of type variables occurring in t_σ.

An occurrence of a variable x_σ is called *bound* if it occurs within the scope of a textually enclosing λx_σ, otherwise the occurrence is called *free*. Note that λx_σ does not bind $x_{\sigma'}$ if $\sigma \neq \sigma'$. A term in which all occurrences of variables are bound is called *closed*.

1.3.1 Terms-in-context

A *context* $\alpha s, xs$ consists of a type context αs together with a list $xs = x_1, \ldots, x_m$ of distinct variables whose types only contain type variables from the list αs.

The condition that xs contains *distinct* variables needs some comment. Since a variable is specified by both a name and a type, it is permitted for xs to contain repeated names, so long as different types are attached to the names. This aspect of the syntax means that one has to proceed with caution when defining the meaning of type variable instantiation, since instantiation may cause variables to become equal 'accidentally': see Section 1.3.3.

A *term-in-context* $\alpha s, xs.t$ consists of a context together with a term t satisfying the following conditions.

- αs contains any type variable that occurs in xs and t.

- xs contains any variable that occurs freely in t.
- xs does not contain any variable that occurs bound in t.

The context αs,xs may contain (type) variables which do not appear in t. Note that the combination of the second and third conditions implies that a variable cannot have both free and bound occurrences in t. For an arbitrary term, there is always an α-equivalent term which satisfies this condition, obtained by renaming the bound variables as necessary.[4] In the semantics of terms to be given below we will restrict attention to such terms. Then the meaning of an arbitrary term is taken to be the meaning of some α-variant of it having no variable both free and bound. (The semantics will equate α-variants, so it does not matter which is chosen.) Evidently for such a term there is a minimal context αs,xs, unique up to the order in which variables are listed, for which αs,$xs.t$ is a term-in-context. As for type variables, we will assume given a fixed total order on variables. Then the unique minimal context with variables listed in order will be called the *canonical* context of the term t.

1.3.2 Semantics of terms

Let Σ_Ω be a signature over a type structure Ω (see Section 1.3). A *model* M of Σ_Ω is specified by a model of the type structure plus for each constant $(\mathsf{c}, \sigma) \in \Sigma_\Omega$ an element

$$M(\mathsf{c}, \sigma) \in \prod_{Xs \in \mathcal{U}^n} [\![\sigma]\!]_M(Xs)$$

of the indicated cartesian product, where n is the number of type variables occurring in σ. In other words $M(\mathsf{c}, \sigma)$ is a (dependently typed) function assigning to each $Xs \in \mathcal{U}^n$ an element of $[\![\sigma]\!]_M(Xs)$. In the case that $n = 0$ (so that σ is monomorphic), $[\![\sigma]\!]_M$ was identified with a set in $\mathcal{U}$ and then $M(c, \sigma)$ can be identified with an element of that set.

The meaning of HOL terms in such a model will now be described. The semantics interprets closed terms involving no type variables as elements of sets in $\mathcal{U}$ (the particular set involved being derived from the type of the term as in Section 1.2.2). More generally, if the closed term involves n type variables then it is interpreted as an element of a product $\prod_{Xs \in \mathcal{U}^n} Y(Xs)$, where the function $Y : \mathcal{U}^n \longrightarrow \mathcal{U}$ is derived from the type of the term (in a type context derived from the term). Thus the meaning of the term is a (dependently typed) function which, when applied to any meanings chosen for the type variables in the term, yields a meaning for the term as an element of a set in $\mathcal{U}$. On the other hand, if the term involves m free variables but no type variables, then it is interpreted as a function $Y_1 \times \cdots \times Y_m \to Y$ where the sets $Y_1, \ldots, Y_m$ in $\mathcal{U}$ are the interpretations of the types of the free variables in the term and the set $Y \in \mathcal{U}$ is

[4]Recall that two terms are said to be α-equivalent if they differ only in the names of their bound variables.

the interpretation of the type of the term; thus the meaning of the term is a function which, when applied to any meanings chosen for the free variables in the term, yields a meaning for the term. Finally, the most general case is of a term involving n type variables and m free variables: it is interpreted as an element of a product

$$\prod_{Xs \in \mathcal{U}^n} Y_1(Xs) \times \cdots \times Y_m(Xs) \to Y(Xs)$$

where the functions $Y_1, \ldots, Y_m, Y : \mathcal{U}^n \longrightarrow \mathcal{U}$ are determined by the types of the free variables and the type of the term (in a type context derived from the term).

More precisely, given a term-in-context $\alpha s, xs.t$ over Σ_Ω suppose

- t has type τ
- $xs = x_1, \ldots, x_m$ and each x_j has type σ_j
- $\alpha s = \alpha_1, \ldots, \alpha_n$.

Then since $\alpha s, xs.t$ is a term-in-context, $\alpha s.\tau$ and $\alpha s.\sigma_j$ are types-in-context, and hence give rise to functions $[\![\alpha s.\tau]\!]_M$ and $[\![\alpha s.\sigma_j]\!]_M$ from $\mathcal{U}^n$ to $\mathcal{U}$ as in section 1.2.2. The meaning of $\alpha s, xs.t$ in the model M will be given by an element

$$[\![\alpha s, xs.t]\!]_M \in \prod_{Xs \in \mathcal{U}^n} \left(\prod_{j=1}^{m} [\![\alpha s.\sigma_j]\!]_M(Xs) \right) \to [\![\alpha s.\tau]\!]_M(Xs).$$

In other words, given

$$\begin{aligned} Xs &= (X_1, \ldots, X_n) \in \mathcal{U}^n \\ ys &= (y_1, \ldots, y_m) \in [\![\alpha s.\sigma_1]\!]_M(Xs) \times \cdots \times [\![\alpha s.\sigma_m]\!]_M(Xs) \end{aligned}$$

one gets an element $[\![\alpha s, xs.t]\!]_M(Xs)(ys)$ of $[\![\alpha s.\tau]\!]_M(Xs)$. The definition of $[\![\alpha s, xs.t]\!]_M$ proceeds by induction on the structure of the term t, as follows. (As before, the subscript M will be dropped from the semantic brackets $[\![\_]\!]$ when the particular model involved is clear from the context.)

- If t is a variable, it must be x_j for some unique $j = 1, \ldots, m$, so $\tau = \sigma_j$ and then $[\![\alpha s, xs.t]\!](Xs)(ys)$ is defined to be y_j.
- Suppose t is a constant $\mathsf{c}_{\sigma'}$, where $(\mathsf{c}, \sigma) \in \Sigma_\Omega$ and σ' is an instance of σ. Then by Lemma 1 of 1.2.3, $\sigma' = \sigma[\tau_1, \ldots, \tau_p/\beta_1, \ldots, \beta_p]$ for uniquely determined types $\tau_1, \ldots, \tau_p$ (where $\beta_1, \ldots, \beta_p$ are the type variables occurring in σ). Then define $[\![\alpha s, xs.t]\!](Xs)(ys)$ to be $M(\mathsf{c}, \sigma)([\![\alpha s.\tau_1]\!](Xs), \ldots, [\![\alpha s.\tau_p]\!](Xs))$, which is an element of $[\![\alpha s.\tau]\!](Xs)$ by Lemma 2 of 1.2.3 (since τ is σ').

- Suppose t is a function application term $(t_1\ t_2)$ where t_1 is of type $\tau' \to \tau$ and t_2 is of type τ'. Then $f = [\![\alpha s, xs.t_1]\!](Xs)(ys)$, being an element of $[\![\alpha s.\tau' \to \tau]\!](Xs)$, is a function from the set $[\![\alpha s.\tau']\!](Xs)$ to the set $[\![\alpha s.\tau]\!](Xs)$ which one can apply to the element $y = [\![\alpha s, xs.t_2]\!](Xs)(ys)$. Define $[\![\alpha s, xs.t]\!](Xs)(ys)$ to be $f(y)$.

- Suppose t is the abstraction term $\lambda x.t_2$where x is of type τ_1 and t_2 of type τ_2. Thus $\tau = \tau_1 \to \tau_2$ and $[\![\alpha s.\tau]\!](Xs)$ is the function set $[\![\alpha s.\tau_1]\!](Xs) \to [\![\alpha s.\tau_2]\!](Xs)$. Define $[\![\alpha s, xs.t]\!](Xs)(ys)$ to be the element of this set which is the function sending $y \in [\![\alpha s.\tau_1]\!](Xs)$ to $[\![\alpha s, xs, x.t_2]\!](Xs)(ys, y)$. (Note that since $\alpha s, xs.t$ is a term-in-context, by convention the bound variable x does not occur in xs and thus $\alpha s, xs, x.t_2$ is also a term-in-context.)

Now define the meaning of a term t_τ in a model M to be the dependently typed function

$$[\![t_\tau]\!] \in \prod_{Xs \in \mathcal{U}^n} \left(\prod_{j=1}^{m} [\![\alpha s.\sigma_j]\!](Xs) \right) \to [\![\alpha s.\tau]\!](Xs)$$

given by $[\![\alpha s, xs.t_\tau]\!]$, where $\alpha s, xs$ is the canonical context of t_τ. So n is the number of type variables in t_τ, αs is a list of those type variables, m is the number of ordinary variables occurring freely in t_τ (assumed to be distinct from the bound variables of t_τ) and the σ_j are the types of those variables. (It is important to note that the list αs, which is part of the canonical context of t, may be strictly bigger than the canonical type contexts of σ_j or τ. So it would not make sense to write just $[\![\sigma_j]\!]$ or $[\![\tau]\!]$ in the above definition.)

If t_τ is a closed term, then $m = 0$ and for each $Xs \in \mathcal{U}^n$ one can identify $[\![t_\tau]\!]$ with the element $[\![t_\tau]\!](Xs)() \in [\![\alpha s.\tau]\!](Xs)$. So for closed terms one gets

$$[\![t_\tau]\!] \in \prod_{Xs \in \mathcal{U}^n} [\![\alpha s.\tau]\!](Xs)$$

where αs is the list of type variables occurring in t_τ and n is the length of that list. If moreover, no type variables occur in t_τ, then $n = 0$ and $[\![t_\tau]\!]$ can be identified with the element $[\![t_\tau]\!]()$ of the set $[\![\tau]\!] \in \mathcal{U}$.

The semantics of terms appears somewhat complicated because of the possible dependency of a term upon both type variables and ordinary variables. Examples of how the definition of the semantics works in practice can be found in Section 2.4.2, where the meaning of several terms denoting logical constants is given.

1.3.3 Substitution

Since terms may involve both type variables and ordinary variables, there are two different operations of substitution on terms which have to be considered—substitution of types for type variables and substitution of terms for variables.

Substituting types for type variables in terms

Suppose t is a term, with canonical context αs,xs say, where $\alpha s = \alpha_1, \ldots, \alpha_n$, $xs = x_1, \ldots, x_m$ and where for $j = 1, \ldots, m$ the type of the variable x_j is σ_j. If $\alpha s'.\tau_i$ ($i = 1, \ldots, n$) are types-in-context, then substituting the types τ_i for the type variables α_i in the list xs, one obtains a new list of variables xs'. Thus the jth entry of xs' has type $\sigma'_j = \sigma_j[\tau s/\alpha s]$. Only substitutions with the following property will be considered.

> In instantiating the type variables αs with the types τs, no two distinct variables in the list xs become equal in the list xs'.[5]

This condition ensures that $\alpha s', xs'$ really is a context. Then one obtains a new term-in-context $\alpha s'$,$xs'.t'$ by substituting the types $\tau s = \tau_1, \ldots, \tau_n$ for the type variables αs in t (with suitable renaming of bound occurrences of variables to make them distinct from the variables in xs'). The notation

$$t[\tau s/\alpha s]$$

is used for the term t'.

Lemma 3 *The meaning of $\alpha s'$,$xs'.t'$ in a model is related to that of t as follows. For all $Xs' \in \mathcal{U}^{n'}$ (where n' is the length of $\alpha s'$)*

$$[\![\alpha s',xs'.t']\!](Xs') = [\![t]\!]([\![\alpha s'.\tau_1]\!](Xs'), \ldots, [\![\alpha s'.\tau_n]\!](Xs')).$$

Lemma 2 in 1.2.3 is needed to see that both sides of the above equation are elements of the same set of functions. The validity of the equation is proved by induction on the structure of the term t.

Substituting terms for variables in terms

Suppose t is a term, with canonical context αs,xs say, where $\alpha s = \alpha_1, \ldots, \alpha_n$, $xs = x_1, \ldots, x_m$ and where for $j = 1, \ldots, m$ the type of the variable x_j is σ_j. If one has terms-in-context αs,$xs'.t_j$ for $j = 1, \ldots, m$ with t_j of the same type as x_j, say σ_j, then one obtains a new term-in-context αs,$xs'.t''$ by substituting the terms $ts = t_1, \ldots, t_m$ for the variables xs in t (with suitable renaming of bound occurrences of variables to prevent the free variables of the t_j becoming bound after substitution). The notation

$$t[ts/xs]$$

is used for the term t''.

[5] Such an identification of variables could occur if the variables had the same name component and their types became equal on instantiation.

Lemma 4 *The meaning of $\alpha s, xs'.t''$ in a model is related to that of t as follows. For all $Xs \in \mathcal{U}^n$ and all $ys' \in [\![\alpha s.\sigma'_1]\!] \times \cdots \times [\![\alpha s.\sigma'_{m'}]\!]$ (where σ'_j is the type of x'_j)*

$$[\![\alpha s, xs'.t'']\!](Xs)(ys') = [\![t]\!](Xs)([\![\alpha s, xs'.t_1]\!](Xs)(ys'), \ldots, [\![\alpha s, xs'.t_m]\!](Xs)(ys'))$$

Once again, this result is proved by induction on the structure of the term t.

1.4 Standard notions

Up to now the syntax of types and terms has been very general. To represent the standard formulas of logic it is necessary to impose some specific structure. In particular, every type structure must contain an atomic type *bool* which is intended to denote the distinguished two-element set $2 \in \mathcal{U}$, regarded as a set of truth-values. Logical formulas are then identified with terms of type *bool*. In addition, various logical constants are assumed to be in all signatures. These requirements are formalized by defining the notion of a standard signature.

1.4.1 Standard type structures

A type structure Ω is *standard* if it contains the atomic types *bool* (of booleans or truth-values) and *ind* (of individuals). (In the literature, the symbol o is often used instead of *bool* and ι instead of *ind*.)

A model M of Ω is *standard* if $M(bool)$ and $M(ind)$ are respectively the distinguished sets 2 and I in the universe $\mathcal{U}$.

It will be assumed from now on that type structures and their models are standard.

1.4.2 Standard signatures

A signature Σ_Ω is *standard* if it contains the following three primitive constants:

$$\Rightarrow_{bool \to bool \to bool}$$

$$=_{\alpha \to \alpha \to bool}$$

$$\varepsilon_{(\alpha \to bool) \to \alpha}$$

The intended interpretation of these constants is that $\Rightarrow$ denotes implication, $=_{\sigma \to \sigma \to bool}$ denotes equality on the set denoted by σ, and $\varepsilon_{(\sigma \to bool) \to \sigma}$ denotes a choice function on the set denoted by σ. More precisely, a model M of Σ_Ω will be called *standard* if

- $M(\Rightarrow, bool \to bool \to bool) \in (2 \to 2 \to 2)$ is the standard implication function, sending $b, b' \in 2$ to

$$(b \Rightarrow b') = \begin{cases} 0 & \text{if } b = 1 \text{ and } b' = 0 \\ 1 & \text{otherwise} \end{cases}$$

- $M(=, \alpha \to \alpha \to bool) \in \prod_{X \in \mathcal{U}} .X \to X \to 2$ is the function assigning to each $X \in \mathcal{U}$ the equality test function, sending $x, x' \in X$ to

$$(x =_X x') = \begin{cases} 1 & \text{if } x = x' \\ 0 & \text{otherwise} \end{cases}$$

- $M(\varepsilon, (\alpha \to bool) \to \alpha) \in \prod_{X \in \mathcal{U}} .(X \to 2) \to X$ is the function assigning to each $X \in \mathcal{U}$ the choice function sending $f \in (X \to 2)$ to

$$\mathrm{ch}_X(f) = \begin{cases} \mathrm{ch}(f^{-1}\{1\}) & \text{if } f^{-1}\{1\} \neq \emptyset \\ \mathrm{ch}(X) & \text{otherwise} \end{cases}$$

  where $f^{-1}\{1\} = \{x \in X : f(x) = 1\}$. (Note that $f^{-1}\{1\}$ is in $\mathcal{U}$ when it is non-empty, by the property **Sub** of the universe $\mathcal{U}$ given in Section 1.1. The function ch is given by property **Choice**.)

It will be assumed from now on that signatures and their models are standard.

Remark This particular choice of primitive constants is arbitrary. The standard collection of logical constants includes T ('true'), F ('false'), $\Rightarrow$ ('implies'), $\wedge$ ('and'), $\vee$ ('or'), $\neg$ ('not'), $\forall$ ('for all'), $\exists$ ('there exists'), $=$ ('equals'), ι ('the'), and ε ('a'). This set is redundant, since it can be defined (in a sense explained in Section 2.5.1) from various subsets. In practice, it is necessary to work with the full set of logical constants, and the particular subset taken as primitive is not important. The interested reader can explore this topic further by reading Andrews' book [**?**] and the references it contains.

Terms of type *bool* are called *formulas*.

The following notational abbreviations are used:

Notation	Meaning
$t_\sigma = t'_\sigma$	$=_{\sigma \to \sigma \to bool}\ t_\sigma\ t'_\sigma$
$t \Rightarrow t'$	$\Rightarrow_{bool \to bool \to bool}\ t_{bool}\ t'_{bool}$
$\varepsilon x_\sigma.\ t$	$\varepsilon_{(\sigma \to bool) \to \sigma}(\lambda x_\sigma.\ t)$

These notations are special cases of general abbreviatory conventions supported by the HOL system. The first two are infixes and the third is a binder (see Section 3.5.1).

Chapter 2

Theories

2.1 Introduction

The result, if any, of a session with the HOL system is an object called a *theory*. This object is closely related to what a logician would call a theory, but there are some differences arising from the needs of mechanical proof. A HOL theory, like a logician's theory, contains sets of types, constants, definitions and axioms. In addition, however, a HOL theory, at any point in time, contains an explicit list of theorems that have already been proved from the axioms and definitions. Logicians have no need to distinguish theorems actually proved from those merely provable; hence they do not normally consider sets of proven theorems as part of a theory; rather, they take the theorems of a theory to be the (often infinite) set of all consequences of the axioms and definitions. A related difference between logicians' theories and HOL theories is that for logicians, theories are static objects, but in HOL they can be thought of as potentially extendable. For example, the HOL system provides tools for adding to theories and combining theories. A typical interaction with HOL consists in combining some existing theories, making some definitions, proving some theorems and then saving the new results.

The purpose of the HOL system is to provide tools to enable well-formed theories to be constructed. The HOL logic is typed: each theory specifies a signature of type and individual constants; these then determine the sets of types and terms as in the previous chapter. All the theorems of such theories are logical consequences of the definitions and axioms of the theory. The HOL system ensures that only well-formed theories can be constructed by allowing theorems to be created only by *formal proof*. Explicating this involves defining what it means to be a theorem, which leads to the description of the proof system of HOL, to be given below. It is shown to be *sound* for the set theoretic semantics of HOL described in the previous chapter. This means that a theorem is satisfied by a model if it has a formal proof from axioms which are themselves satisfied by the model. Since a logical contradiction is not satisfied by any model, this guarantees in particular that a theory possessing a model is necessarily consistent, i.e. a logical contradiction cannot be formally proved from its axioms.

This chapter also describes the various mechanisms by which HOL theories can be extended to new theories. Each mechanism is shown to preserve the property of possessing a model. Thus theories built up from the initial HOL theory (which does possess a model) using these mechanisms are guaranteed to be consistent.

2.2 Sequents

The HOL logic is phrased in terms of hypothetical assertions called *sequents*. Fixing a (standard) signature Σ_Ω, a sequent is a pair (Γ, t) where Γ is a finite set of formulas over Σ_Ω and t is a single formula over Σ_Ω.[1] The set of formulas Γ forming the first component of a sequent is called its set of *assumptions* and the term t forming the second component is called its *conclusion*. When it is not ambiguous to do so, a sequent $(\{\}, t)$ is written as just t.

Intuitively, a model M of Σ_Ω *satisfies* a sequent (Γ, t) if any interpretation of relevant free variables as elements of M making the formulas in Γ true, also makes the formula t true. To make this more precise, suppose $\Gamma = \{t_1, \ldots, t_p\}$ and let $\alpha s, xs$ be a context containing all the type variables and all the free variables occurring in the formulas $t, t_1, \ldots, t_p$. Suppose that αs has length n, that $xs = x_1, \ldots, x_m$ and that the type of x_j is σ_j. Since formulas are terms of type *bool*, the semantics of terms defined in the previous chapter gives rise to elements $[\![\alpha s,xs.t]\!]_M$ and $[\![\alpha s,xs.t_k]\!]_M$ $(k = 1, \ldots, p)$ in

$$\prod_{Xs \in \mathcal{U}^n} \left(\prod_{j=1}^{m} [\![\alpha s.\sigma_j]\!]_M(Xs) \right) \to 2$$

Say that the model M *satisfies* the sequent (Γ, t) and write

$$\Gamma \models_M t$$

if for all $Xs \in \mathcal{U}^n$ and all $ys \in [\![\alpha s.\sigma_1]\!]_M(Xs) \times \cdots \times [\![\alpha s.\sigma_m]\!]_M(Xs)$ with

$$[\![\alpha s,xs.t_k]\!]_M(Xs)(ys) = 1$$

for all $k = 1, \ldots, p$, it is also the case that

$$[\![\alpha s,xs.t]\!]_M(Xs)(ys) = 1.$$

(Recall that 2 is the set $\{0, 1\}$.)

In the case $p = 0$, the satisfaction of $(\{\}, t)$ by M will be written $\models_M t$. Thus $\models_M t$ means that the dependently typed function

$$[\![t]\!]_M \in \prod_{Xs \in \mathcal{U}^n} \left(\prod_{j=1}^{m} [\![\alpha s.\sigma_j]\!]_M(Xs) \right) \to 2$$

is constant with value $1 \in 2$.

[1] Note that the type subscript is omitted from terms when it is clear from the context that they are formulas, i.e. have type *bool*.

2.3 Logic

A deductive system $\mathcal{D}$ is a set of pairs $(L, (\Gamma, t))$ where L is a (possibly empty) list of sequents and (Γ, t) is a sequent.

A sequent (Γ, t) follows from a set of sequents Δ by a deductive system $\mathcal{D}$ if and only if there exist sequents (Γ_1, t_1), ... , (Γ_n, t_n) such that:

1. $(\Gamma, t) = (\Gamma_n, t_n)$, and
2. for all i such that $1 \leq i \leq n$
   (a) either $(\Gamma_i, t_i) \in \Delta$ or
   (b) $(L_i, (\Gamma_i, t_i)) \in \mathcal{D}$ for some list L_i of members of $\Delta \cup \{(\Gamma_1, t_1), \ldots, (\Gamma_{i-1}, t_{i-1})\}$
   .

The sequence $(\Gamma_1, t_1), \cdots, (\Gamma_n, t_n)$ is called a *proof* of (Γ, t) from Δ with respect to $\mathcal{D}$.

Note that if (Γ, t) follows from Δ, then (Γ, t) also follows from any Δ' such that $\Delta \subseteq \Delta'$. This property is called *monotonicity*.

The notation $t_1, \ldots, t_n \vdash_{\mathcal{D},\Delta} t$ means that the sequent $(\{t_1, \ldots, t_n\},\ t)$ follows from Δ by $\mathcal{D}$. If either $\mathcal{D}$ or Δ is clear from the context then it may be omitted. In the case that there are no hypotheses (i.e. $n = 0$), just $\vdash t$ is written.

In practice, a particular deductive system is usually specified by a number of (schematic) *rules of inference*, which take the form

$$\frac{\Gamma_1 \ \vdash \ t_1 \qquad \cdots \qquad \Gamma_n \ \vdash \ t_n}{\Gamma \ \vdash \ t}$$

The sequents above the line are called the *hypotheses* of the rule and the sequent below the line is called its *conclusion*. Such a rule is schematic because it may contain metavariables standing for arbitrary terms of the appropriate types. Instantiating these metavariables with actual terms, one gets a list of sequents above the line and a single sequent below the line which together constitute a particular element of the deductive system. The instantiations allowed for a particular rule may be restricted by imposing a *side condition* on the rule.

2.3.1 The HOL deductive system

The deductive system of the HOL logic is specified by eight rules of inference, given below. The first three rules have no hypotheses; their conclusions can always be deduced. The identifiers in square brackets are the names of the ML functions in the HOL system that implement the corresponding inference rules (See Section 3.9). Any side conditions restricting the scope of a rule are given immediately below it.

Assumption introduction [ASSUME]

$$\overline{t \vdash t}$$

Reflexivity [REFL]

$$\overline{\vdash t = t}$$

Beta-conversion [BETA_CONV]

$$\overline{\vdash (\lambda x.\ t_1)t_2 = t_1[t_2/x]}$$

- Where $t_1[t_2/x]$ is the result of substituting t_2 for x in t_1, with suitable renaming of variables to prevent free variables in t_2 becoming bound after substitution.

Substitution [SUBST]

$$\frac{\Gamma_1 \vdash t_1 = t_1' \qquad \cdots \qquad \Gamma_n \vdash t_n = t_n' \qquad\qquad \Gamma \vdash t[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t_1', \ldots, t_n']}$$

- Where $t[t_1, \ldots, t_n]$ denotes a term t with some free occurrences of subterms t_1, $\ldots$, t_n singled out and $t[t_1', \ldots, t_n']$ denotes the result of replacing each selected occurrence of t_i by t_i' (for $1 \leq i \leq n$), with suitable renaming of variables to prevent free variables in t_i' becoming bound after substitution.

Abstraction [ABS]

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.\ t_1) = (\lambda x.\ t_2)}$$

- Provided x is not free in Γ.

Type instantiation [INST_TYPE]

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

- Where $t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]$ is the result of substituting, in parallel, the types σ_1, $\ldots$, σ_n for type variables α_1, $\ldots$, α_n in t, with the restrictions:

  (i) none of the type variables α_1, $\ldots$, α_n occur in Γ;

  (ii) no distinct variables in t become identified after the instantiation.[2]

[2]The ML function implementing INST_TYPE in the HOL system fails if side condition (i) is violated, but instead of failing if (ii) is violated, it automatically renames any variable whose type is instantiated if the variable is preceded in t by a different variable with the same name.

Discharging an assumption [DISCH]

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

- Where $\Gamma - \{t_1\}$ is the set subtraction of $\{t_1\}$ from Γ.

Modus Ponens [MP]

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

In addition to these eight rules, there are also five *axioms* which could have been regarded as rules of inference without hypotheses. This is not done, however, since it is most natural to state the axioms using some defined logical constants and the principle of constant definition has not yet been described. The axioms are given in Section 2.4.3 and the definitions of the extra logical constants they involve are given in Section 2.4.2.

The particular set of rules and axioms chosen to axiomatize the HOL logic is rather arbitrary. It is partly based on the rules that were used in the LCF logic PPλ, since HOL was implemented by modifying the LCF system. In particular, the substitution rule SUBST is exactly the same as the corresponding rule in LCF; the code implementing this was written by Robin Milner and is highly optimized. Because substitution is such a pervasive activity in proof, it was felt to be important that the system primitive be as fast as possible. From a logical point of view it would be better to have a simpler substitution primitive, such as 'Rule R' of Andrews' logic $\mathcal{Q}_0$, and then to derive more complex rules from it.

2.3.2 Soundness theorem

The rules of the the HOL deductive system are sound for the notion of satisfaction defined in Section 2.2: for any instance of the rules of inference, if a (standard) model satisfies the hypotheses of the rule it also satisfies the conclusion.

Proof The verification of the soundness of the rules is straightforward. The properties of the semantics with respect to substitution given by Lemmas 3 and 4 in Section 1.3.3 are needed for rules BETA_CONV, SUBST and INST_TYPE.[3] The fact that $=$ and $\Rightarrow$ are interpreted standardly (as in Section 1.4.2) is needed for rules REFL, BETA_CONV, SUBST, ABS, DISCH and MP.

[3] Note in particular that the second restriction on INST_TYPE enables the result on the semantics of substituting types for type variables in terms to be applied.

2.4 HOL Theories

A HOL *theory* $\mathcal{T}$ is a 4-tuple:

$$\mathcal{T} \quad = \quad \langle \mathsf{Struc}_{\mathcal{T}}, \mathsf{Sig}_{\mathcal{T}}, \mathsf{Axioms}_{\mathcal{T}}, \mathsf{Theorems}_{\mathcal{T}} \rangle$$

where

(i) $\mathsf{Struc}_{\mathcal{T}}$ is a type structure called the type structure of $\mathcal{T}$;

(ii) $\mathsf{Sig}_{\mathcal{T}}$ is a signature over $\mathsf{Struc}_{\mathcal{T}}$ called the signature of $\mathcal{T}$;

(iii) $\mathsf{Axioms}_{\mathcal{T}}$ is a set of sequents over $\mathsf{Sig}_{\mathcal{T}}$ called the axioms of $\mathcal{T}$;

(iv) $\mathsf{Theorems}_{\mathcal{T}}$ is a set of sequents over $\mathsf{Sig}_{\mathcal{T}}$ called the theorems of $\mathcal{T}$, with the property that every member follows from $\mathsf{Axioms}_{\mathcal{T}}$ by the HOL deductive system.

The sets $\mathsf{Types}_{\mathcal{T}}$ and $\mathsf{Terms}_{\mathcal{T}}$ of types and terms of a theory $\mathcal{T}$ are, respectively, the sets of types and terms constructable from the type structure and signature of $\mathcal{T}$, i.e.:

$$\begin{aligned} \mathsf{Types}_{\mathcal{T}} &= \mathsf{Types}_{\mathsf{Struc}_{\mathcal{T}}} \\ \mathsf{Terms}_{\mathcal{T}} &= \mathsf{Terms}_{\mathsf{Sig}_{\mathcal{T}}} \end{aligned}$$

A model of a theory $\mathcal{T}$ is specified by giving a (standard) model M of the underlying signature of the theory with the property that M satisfies all the sequents which are axioms of $\mathcal{T}$. Because of the Soundness Theorem 2.3.2, it follows that M also satisfies any sequents in the set of given theorems, $\mathsf{Theorems}_{\mathcal{T}}$.

2.4.1 The theory `MIN`

The *minimal theory* `MIN` is defined by:

$$\texttt{MIN} = \langle \{(bool, 0),\ (ind, 0)\},\ \{\Rightarrow_{bool\to bool\to bool}, =_{\alpha\to\alpha\to bool}, \varepsilon_{(\alpha\to bool)\to\alpha}\},\ \{\},\ \{\}\rangle$$

Since the theory `MIN` has a signature consisting only of standard items and has no axioms, it possesses a unique standard model, which will be denoted *Min*.

Although the theory `MIN` contains only the minimal standard syntax, by exploiting the higher order constructs of HOL one can construct a rather rich collection of terms over it. The following theory introduces names for some of these terms that denote useful logical operations in the model *Min*.

2.4.2 The theory LOG

The theory LOG has the same type structure as MIN. Its signature contains the constants in MIN and the following constants:

T_{bool}

$\forall_{(\alpha \to bool) \to bool}$

$\exists_{(\alpha \to bool) \to bool}$

F_{bool}

$\neg_{bool \to bool}$

$\wedge_{bool \to bool \to bool}$

$\vee_{bool \to bool \to bool}$

$\mathsf{One\_One}_{(\alpha \to \beta) \to bool}$

$\mathsf{Onto}_{(\alpha \to \beta) \to bool}$

$\mathsf{Type\_Definition}_{(\alpha \to bool) \to (\beta \to \alpha) \to bool}$

The following special notation is used in connection with these constants:

Notation	Meaning
$\forall x_\sigma .\ t$	$\forall(\lambda x_\sigma .\ t)$
$\forall x_1\ x_2\ \cdots\ x_n .\ t$	$\forall x_1 .\ (\forall x_2 .\ \cdots\ (\forall x_n .\ t)\ \cdots)$
$\exists x_\sigma .\ t$	$\exists(\lambda x_\sigma .\ t)$
$\exists x_1\ x_2\ \cdots\ x_n .\ t$	$\exists x_1 .\ (\exists x_2 .\ \cdots\ (\exists x_n .\ t)\ \cdots)$
$t_1\ \wedge\ t_2$	$\wedge\ t_1\ t_2$
$t_1\ \vee\ t_2$	$\vee\ t_1\ t_2$

The axioms of the theory LOG consist of the following sequents:

$$
\begin{array}{ll}
\vdash & \mathsf{T} = ((\lambda x_{bool} .\ x) = (\lambda x_{bool} .\ x)) \\
\vdash & \forall = \lambda P_{\alpha \to bool} .\ \ P = (\lambda x .\ \mathsf{T}) \\
\vdash & \exists = \lambda P_{\alpha \to bool} .\ \ P(\varepsilon\ P) \\
\vdash & \mathsf{F} = \forall b_{bool} .\ \ b \\
\vdash & \neg = \lambda b .\ \ b \Rightarrow \mathsf{F} \\
\vdash & \wedge = \lambda b_1\ b_2 .\ \forall b .\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b \\
\vdash & \vee = \lambda b_1\ b_2 .\ \forall b .\ (b_1 \Rightarrow b) \Rightarrow ((b_2 \Rightarrow b) \Rightarrow b) \\
\vdash & \mathsf{One\_One} = \lambda f_{\alpha \to \beta} .\ \forall x_1\ x_2 .\ (f\ x_1 = f\ x_2) \Rightarrow (x_1 = x_2) \\
\vdash & \mathsf{Onto} = \lambda f_{\alpha \to \beta} .\ \forall y .\ \exists x .\ y = f\ x \\
\vdash & \mathsf{Type\_Definition} = \ \lambda P_{\alpha \to bool}\ rep_{\beta \to \alpha} . \mathsf{One\_One}\ rep\ \ \wedge \\
 & \qquad\qquad\qquad\qquad (\forall x .\ P\ x\ =\ (\exists y .\ x = rep\ y))
\end{array}
$$

Finally, as for the theory `MIN`, the set $\mathsf{Theorems}_{\mathtt{LOG}}$ is taken to be empty.

Note that the axioms of the theory `LOG` are essentially *definitions* of the new constants of `LOG` as terms in the original theory `MIN`. (The mechanism for making such extensions of theories by definitions of new constants will be set out in general in Section 2.5.1.) The first seven axioms define the logical constants for truth, universal quantification, existential quantification, falsity, negation, conjunction and disjunction. Although these definitions may be obscure to some readers, they are in fact standard definitions of these logical constants in terms of implication, equality and choice within higher order logic. The next two axioms define the properties of a function being one-one and onto; they will be used to express the axiom of infinity (see Section 2.4.3), amongst other things. The last axiom defines a constant used for type definitions (see Section 2.5.4).

The unique standard model *Min* of `MIN` gives rise to a unique standard model of `LOG`. This is because, given the semantics of terms set out in Section 1.3.2, to satisfy the above equations one is forced to interpret the new constants in the following way:

- $[\![\mathsf{T}_{bool}]\!] = 1 \in 2$
- $[\![\forall_{(\alpha\to bool)\to bool}]\!] \in \prod_{X\in\mathcal{U}}(X\to 2)\to 2$ sends $X \in \mathcal{U}$ and $f \in X\to 2$ to

$$[\![\forall]\!](X)(f) = \begin{cases} 1 & \text{if } f^{-1}\{1\} = X \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\exists_{(\alpha\to bool)\to bool}]\!] \in \prod_{X\in\mathcal{U}}(X\to 2)\to 2$ sends $X \in \mathcal{U}$ and $f \in X\to 2$ to

$$[\![\exists]\!](X)(f) = \begin{cases} 1 & \text{if } f^{-1}\{1\} \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\mathsf{F}_{bool}]\!] = 0 \in 2$
- $[\![\neg_{bool\to bool}]\!] \in 2\to 2$ sends $b \in 2$ to

$$[\![\neg]\!](b) = \begin{cases} 1 & \text{if } b = 0 \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\wedge_{bool\to bool\to bool}]\!] \in 2\to 2\to 2$ sends $b, b' \in 2$ to

$$[\![\wedge]\!](b)(b') = \begin{cases} 1 & \text{if } b = 1 = b' \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\vee_{bool\to bool\to bool}]\!] \in 2\to 2\to 2$ sends $b, b' \in 2$ to

$$[\![\vee]\!](b)(b') = \begin{cases} 0 & \text{if } b = 0 = b' \\ 1 & \text{otherwise} \end{cases}$$

- $[\![\mathsf{One\_One}_{(\alpha\to\beta)\to bool}]\!] \in \prod_{(X,Y)\in\mathcal{U}^2}(X\to Y)\to 2$ sends $(X,Y)\in\mathcal{U}^2$ and $f\in(X\to Y)$ to

$$[\![\mathsf{One\_One}]\!](X,Y)(f) = \begin{cases} 0 & \text{if } f(x) = f(x') \text{ for some } x \neq x' \text{ in } X \\ 1 & \text{otherwise} \end{cases}$$

- $[\![\mathsf{Onto}_{(\alpha\to\beta)\to bool}]\!] \in \prod_{(X,Y)\in\mathcal{U}^2}(X\to Y)\to 2$ sends $(X,Y)\in\mathcal{U}^2$ and $f\in(X\to Y)$ to

$$[\![\mathsf{Onto}]\!](X,Y)(f) = \begin{cases} 1 & \text{if } \{f(x) : x\in X\} = Y \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\mathsf{Type\_Definition}_{(\alpha\to bool)\to(\beta\to\alpha)\to bool}]\!] \in \prod_{(X,Y)\in\mathcal{U}^2}(X\to 2)\to(Y\to X)\to 2$ sends $(X,Y)\in\mathcal{U}^2$, $f\in(X\to 2)$ and $g\in(Y\to X)$ to

$$[\![\mathsf{Type\_Definition}]\!](X,Y)(f)(g) = \begin{cases} 1 & \text{if } [\![\mathsf{One\_One}]\!](Y,X)(g) = 1 \\ & \text{and } f^{-1}\{1\} = \{g(y) : y\in Y\} \\ 0 & \text{otherwise.} \end{cases}$$

Since these definitions were obtained by applying the semantics of terms to the left hand sides of the equations which form the axioms of `LOG`, these axioms are satisfied and one obtains a model of the theory `LOG`.

2.4.3 The theory `INIT`

The theory `INIT` is obtained by adding the following five axioms to the theory `LOG`.

`BOOL_CASES_AX`	$\vdash \forall b.\ (b = \mathsf{T}) \vee (b = \mathsf{F})$
`IMP_ANTISYM_AX`	$\vdash \forall b_1\ b_2.\ (b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_1 = b_2)$
`ETA_AX`	$\vdash \forall f_{\alpha\to\beta}.\ (\lambda x.\ f\ x) = f$
`SELECT_AX`	$\vdash \forall P_{\alpha\to bool}\ x.\ P\ x \Rightarrow P(\varepsilon\ P)$
`INFINITY_AX`	$\vdash \exists f_{ind\to ind}.\ \mathsf{One\_One}\ f\ \wedge\ \neg(\mathsf{Onto}\ f)$

The unique standard model of `LOG` satisfies these five axioms and hence is the unique standard model of the theory `INIT`. (For axiom `SELECT_AX` one needs to use the definition of $[\![\varepsilon]\!]$ given in Section 1.4.2; for axiom `INFINITY_AX` one needs the fact that $[\![ind]\!] = \mathrm{I}$ is an infinite set.)

The theory `INIT` is the initial theory of the HOL logic. A theory which extends `INIT` will be called a *standard theory*.

2.4.4 Consistency

A (standard) theory is *consistent* if it is not the case that every sequent over its signature can be derived from the theory's axioms using the HOL logic, or equivalently, if the particular sequent $\vdash \mathsf{F}$ cannot be so derived.

The existence of a (standard) model of a theory is sufficient to establish its consistency. For by the Soundness Theorem 2.3.2, any sequent that can be derived from the theory's axioms will be satisfied by the model, whereas the sequent $\vdash \mathsf{F}$ is never satisfied in any standard model. So in particular, the initial theory `INIT` is consistent.

However, it is possible for a theory to be consistent but not to possess a standard model. This is because the notion of a *standard* model is quite restrictive—in particular there is no choice how to interpret the integers and their arithmetic in such a model. The famous incompleteness theorem of Gödel ensures that there are sequents which are satisfied in all standard models (i.e. which are 'true'), but which are not provable in the HOL logic.

2.5 Extensions of theories

A theory $\mathcal{T}'$ is said to be an *extension* of a theory $\mathcal{T}$ if:

(i) $\mathsf{Struc}_{\mathcal{T}} \subseteq \mathsf{Struc}_{\mathcal{T}'}$.

(ii) $\mathsf{Sig}_{\mathcal{T}} \subseteq \mathsf{Sig}_{\mathcal{T}'}$.

(iii) $\mathsf{Axioms}_{\mathcal{T}} \subseteq \mathsf{Axioms}_{\mathcal{T}'}$.

(iv) $\mathsf{Theorems}_{\mathcal{T}} \subseteq \mathsf{Theorems}_{\mathcal{T}'}$.

In this case, any model M' of the larger theory $\mathcal{T}'$ can be restricted to a model of the smaller theory $\mathcal{T}$ in the following way. First, M' gives rise to a model of the structure and signature of $\mathcal{T}$ simply by forgetting the values of M' at constants not in $\mathsf{Struc}_{\mathcal{T}}$ or $\mathsf{Sig}_{\mathcal{T}}$. Denoting this model by M, one has for all $\sigma \in \mathsf{Types}_{\mathcal{T}}$, $t \in \mathsf{Terms}_{\mathcal{T}}$ and for all suitable contexts that

$$
\begin{aligned}
[\![\alpha s.\sigma]\!]_M &= [\![\alpha s.\sigma]\!]_{M'} \\
[\![\alpha s,xs.t]\!]_M &= [\![\alpha s,xs.t]\!]_{M'}.
\end{aligned}
$$

Consequently if (Γ, t) is a sequent over $\mathsf{Sig}_{\mathcal{T}}$ (and hence also over $\mathsf{Sig}_{\mathcal{T}'}$), then $\Gamma \models_M t$ if and only if $\Gamma \models_{M'} t$. Since $\mathsf{Axioms}_{\mathcal{T}} \subseteq \mathsf{Axioms}_{\mathcal{T}'}$ and M' is a model of $\mathcal{T}'$, it follows that M is a model of $\mathcal{T}$. M will be called the *restriction* of the model M' of the theory $\mathcal{T}'$ to the subtheory $\mathcal{T}$.

There are two main mechanisms for making extensions of theories in HOL:

- Extension by a constant specification (see Section 2.5.2).
- Extension by a type specification (see Section 2.5.5).[4]

The first mechanism allows 'loose specifications' of constants as in the **Z** notation [**?**]; the latter allows new types and type-operators to be introduced. As special cases (when the thing being specified is uniquely determined) one also has:

- Extension by a constant definition (see Section 2.5.1).
- Extension by a type definition (see Section 2.5.4).

These mechanisms are described in the following sections. They all produce *definitional extensions* in the sense that they extend a theory by adding new constants and types which are defined in terms of properties of existing ones. Their key property is that the extended theory possesses a (standard) model if the original theory does. So a series of these extensions starting from the theory `INIT` is guaranteed to result in a theory with a standard model, and hence in a consistent theory. It is also possible to extend theories simply by adding new uninterpreted constants and types. This preserves consistency, but is unlikely to be useful without additional axioms. However, when adding arbitrary new axioms, there is no guarantee that consistency is preserved. The advantages of postulation over definition have been likened by Bertrand Russell to the advantages of theft over honest toil.[5] As it is all too easy to introduce inconsistent axiomatizations, users of the HOL system are strongly advised to resist the temptation to add axioms, but to toil through definitional theories honestly.

2.5.1 Extension by constant definition

A *constant definition* over a signature Σ_Ω is a formula of the form $\mathsf{c}_\sigma = t_\sigma$, such that:

(i) c is not the name of any constant in Σ_Ω;

(ii) t_σ a closed term in $\mathsf{Terms}_{\Sigma_\Omega}$.

(iii) all the type variables occurring in t_σ also occur in σ

Given a theory $\mathcal{T}$ and such a constant definition over $\mathsf{Sig}_\mathcal{T}$, then the *definitional extension* of $\mathcal{T}$ by $\mathsf{c}_\sigma = t_\sigma$ is the theory $\mathcal{T}{+_{def}}\langle \mathsf{c}_\sigma = t_\sigma\rangle$ defined by:

$$\begin{aligned}\mathcal{T}{+_{def}}\langle \mathsf{c}_\sigma = t_\sigma\rangle \;=\; \langle\; & \mathsf{Struc}_\mathcal{T},\ \mathsf{Sig}_\mathcal{T} \cup \{(\mathsf{c},\sigma)\},\\ & \mathsf{Axioms}_\mathcal{T} \cup \{\mathsf{c}_\sigma = t_\sigma\},\ \mathsf{Theorems}_\mathcal{T}\rangle\end{aligned}$$

[4]This theory extension mechanism is not implemented in Version 2.0 of the HOL system.

[5]See page 71 of Russell's book *Introduction to Mathematical Philosophy*.

Note that the mechanism of extension by constant definition has already been used implicitly in forming the theory `LOG` from the theory `MIN` in Section 2.4.2. Thus with the notation of this section one has

$$
\begin{aligned}
\texttt{LOG} \;=\; \texttt{MIN} & +_{def}\langle \mathsf{T} \;=\; ((\lambda x_{bool}.\, x) = (\lambda x_{bool}.\, x))\rangle \\
& +_{def}\langle \forall \;=\; \lambda P_{\alpha\to bool}.\;\; P = (\lambda x.\, \mathsf{T})\rangle \\
& +_{def}\langle \exists \;=\; \lambda P_{\alpha\to bool}.\;\; P(\varepsilon\, P)\rangle \\
& +_{def}\langle \mathsf{F} \;=\; \forall b_{bool}.\;\; b\rangle \\
& +_{def}\langle \neg \;=\; \lambda b.\;\; b \Rightarrow \mathsf{F}\rangle \\
& +_{def}\langle \wedge \;=\; \lambda b_1\; b_2.\, \forall b.\, (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b\rangle \\
& +_{def}\langle \vee \;=\; \lambda b_1\; b_2.\, \forall b.\, (b_1 \Rightarrow b) \Rightarrow ((b_2 \Rightarrow b) \Rightarrow b)\rangle \\
& +_{def}\langle \mathsf{One\_One} \;=\; \lambda f_{\alpha\to\beta}.\, \forall x_1\; x_2.\, (f\; x_1 = f\; x_2) \Rightarrow (x_1 = x_2)\rangle \\
& +_{def}\langle \mathsf{Onto} \;=\; \lambda f_{\alpha\to\beta}.\, \forall y.\, \exists x.\, y = f\; x\rangle \\
& +_{def}\langle \mathsf{Type\_Definition} \;=\; \lambda P_{\alpha\to bool}\; rep_{\beta\to\alpha}. \\
& \qquad\qquad \mathsf{One\_One}\; rep \;\wedge \\
& \qquad\qquad (\forall x.\, P\; x \;=\; (\exists y.\, x = rep\; y))\rangle
\end{aligned}
$$

If $\mathcal{T}$ possesses a standard model then so does the extension $\mathcal{T}+_{def}\langle \mathsf{c}_\sigma = t_\sigma\rangle$. This will be proved as a corollary of the corresponding result in Section 2.5.2 by showing that extension by constant definition is in fact a special case of extension by constant specification. (This reduction requires that one is dealing with *standard* theories in the sense of section 2.4.3, since although existential quantification is not needed for constant definitions, it is needed to state the mechanism of constant specification.)

Remark Condition (iii) in the definition of what constitutes a correct constant definition is an important restriction without which consistency could not be guaranteed. To see this, consider the term $\exists f_{\alpha\to\alpha}.\, \mathsf{One\_One}\; f \;\wedge\; \neg(\mathsf{Onto}\; f)$, which expresses the proposition that (the set of elements denoted by the) type α is infinite. The term contains the type variable α, whereas the type of the term, $bool$, does not. Thus by (iii)

$$\mathsf{c}_{bool} = \exists f_{\alpha\to\alpha}.\, \mathsf{One\_One}\; f \;\wedge\; \neg(\mathsf{Onto}\; f)$$

is not allowed as a constant definition. The problem is that the meaning of the right hand side of the definition varies with α, whereas the meaning of the constant on the left hand side is fixed, since it does not contain α. Indeed, if we were allowed to extend the consistent theory `INIT` by this definition, the result would be an inconsistent theory. For instantiating α to ind in the right hand side results in a term that is provable from the axioms of `INIT`, and hence $\mathsf{c}_{bool} = \mathsf{T}$ is provable in the extended theory. But equally, instantiating α to $bool$ makes the negation of the right hand side provable from the axioms of `INIT`, and hence $\mathsf{c}_{bool} = \mathsf{F}$ is also provable in the extended theory. Combining these theorems, one has that $\mathsf{T} = \mathsf{F}$, i.e. F is provable in the extended theory.

2.5.2 Extension by constant specification

Constant specifications introduce constants (or sets of constants) that satisfy arbitrary given (consistent) properties. For example, a theory could be extended by a constant specification to have two new constants b_1 and b_2 of type *bool* such that $\neg(\mathsf{b}_1 = \mathsf{b}_2)$. This specification does not uniquely define b_1 and b_2, since it is satisfied by either $\mathsf{b}_1 = \mathsf{T}$ and $\mathsf{b}_2 = \mathsf{F}$, or $\mathsf{b}_1 = \mathsf{F}$ and $\mathsf{b}_2 = \mathsf{T}$. To ensure that such specifications are consistent, they can only be made if it has already been proved that the properties which the new constants are to have are consistent. This rules out, for example, introducing three boolean constants b_1, b_2 and b_3 such that $\mathsf{b}_1 \neq \mathsf{b}_2$, $\mathsf{b}_1 \neq \mathsf{b}_3$ and $\mathsf{b}_2 \neq \mathsf{b}_3$.

Suppose $\exists x_1 \cdots x_n.\ t$ is a formula, with $x_1, \ldots, x_n$ distinct variables. If $\vdash\ \exists x_1 \cdots x_n.\ t$, then a constant specification allows new constants $\mathsf{c}_1, \ldots, \mathsf{c}_n$ to be introduced satisfying:

$$\vdash\ t[\mathsf{c}_1, \cdots, \mathsf{c}_n / x_1, \cdots, x_n]$$

where $t[\mathsf{c}_1, \cdots, \mathsf{c}_n / x_1, \cdots, x_n]$ denotes the result of simultaneously substituting $\mathsf{c}_1, \ldots, \mathsf{c}_n$ for $x_1, \ldots, x_n$ respectively. Of course the type of each constant c_i must be the same as the type of the corresponding variable x_i. To ensure that this extension mechanism preserves the property of possessing a model, a further more technical requirement is imposed on these types: they must each contain all the type variables occurring in t. This condition is discussed further in Section 2.5.3 below.

Formally, a *constant specification* for a theory $\mathcal{T}$ is given by

Data

$$\langle(\mathsf{c}_1, \ldots, \mathsf{c}_n), \lambda x_{1\sigma_1}, \ldots, x_{n\sigma_n}.\ t_{bool}\rangle$$

Conditions

(i) $\mathsf{c}_1, \ldots, \mathsf{c}_n$ are distinct names that are not the names of any constants in $\mathsf{Sig}_{\mathcal{T}}$.

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

(iii) $tyvars(t_{bool})\ =\ tyvars(\sigma_i)$ for $1 \leq i \leq n$.

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

The extension of a standard theory $\mathcal{T}$ by such a constant specification is denoted by

$$\mathcal{T} +_{spec} \langle(\mathsf{c}_1, \ldots, \mathsf{c}_n), \lambda x_{1\sigma_1}, \ldots, x_{n\sigma_n}.\ t_{bool}\rangle$$

and is defined to be the theory:

$$\begin{array}{l}\langle\mathsf{Struc}_{\mathcal{T}},\\ \ \mathsf{Sig}_{\mathcal{T}} \cup \{\mathsf{c}_{1\sigma_1}, \ldots, \mathsf{c}_{n\sigma_n}\},\\ \ \mathsf{Axioms}_{\mathcal{T}} \cup \{t[\mathsf{c}_1, \ldots, \mathsf{c}_n / x_1, \ldots, x_n]\},\\ \ \mathsf{Theorems}_{\mathcal{T}}\rangle\end{array}$$

Proposition *The theory* $\mathcal{T}+_{spec}\langle(\mathsf{c}_1,\ldots,\mathsf{c}_n),\lambda x_{1\sigma_1},\ldots,x_{n\sigma_n}.\ t_{bool}\rangle$ *has a standard model if the theory* $\mathcal{T}$ *does.*

Proof Suppose M is a standard model of $\mathcal{T}$. Let $\alpha s = \alpha_1,\ldots,\alpha_m$ be the list of distinct type variables occurring in the formula t. Then $\alpha s,xs.t$ is a term-in-context, where $xs = x_1,\ldots,x_n$. (Change any bound variables in t to make them distinct from xs if necessary.) Interpreting this term-in-context in the model M yields

$$[\![\alpha s,xs.t]\!]_M \in \prod_{Xs\in\mathcal{U}^m}\left(\prod_{i=1}^{n}[\![\alpha s.\sigma_i]\!]_M(Xs)\right)\to 2$$

Now $\exists xs.\ t$ is in $\mathsf{Theorems}_{\mathcal{T}}$ and hence by the Soundness Theorem 2.3.2 this sequent is satisfied by M. Using the semantics of $\exists$ given in Section 2.4.2, this means that for all $Xs\in\mathcal{U}^m$ the set

$$S(Xs) = \{ys \in [\![\alpha s.\sigma_1]\!]_M(Xs)\times\cdots\times[\![\alpha s.\sigma_n]\!]_M(Xs)\ :\ [\![\alpha s,xs.t]\!]_M(Xs)(ys) = 1\}$$

is non-empty. Since it is also a subset of a finite product of sets in $\mathcal{U}$, it follows that it is an element of $\mathcal{U}$ (using properties **Sub** and **Prod** of the universe). So one can apply the global choice function $\mathrm{ch}\in\prod_{X\in\mathcal{U}}X$ to select a specific element

$$(s_1(Xs),\ldots,s_n(Xs)) = \mathrm{ch}(S(Xs)) \in \prod_{i=1}^{n}[\![\alpha s.\sigma_i]\!]_M(Xs)$$

at which $[\![\alpha s,xs.t]\!]_M(Xs)$ takes the value 1. Extend M to a model M' of the signature of $\mathcal{T}+_{spec}\langle(\mathsf{c}_1,\ldots,\mathsf{c}_n),\lambda x_{1\sigma_1},\ldots,x_{n\sigma_n}.\ t_{bool}\rangle$ by defining its value at each new constant (c_i,σ_i) to be

$$M'(\mathsf{c}_i,\sigma_i) = s_i \in \prod_{Xs\in\mathcal{U}^m}[\![\sigma_i]\!]_M(Xs).$$

Note that the Condition (iii) in the definition of a constant specification ensures that αs is the canonical context of each type σ_i, so that $[\![\sigma_i]\!] = [\![\alpha s.\sigma_i]\!]$ and thus s_i is indeed an element of the above product.

Since t is a term of the subtheory $\mathcal{T}$ of $\mathcal{T}+_{spec}\langle(\mathsf{c}_1,\ldots,\mathsf{c}_n),\lambda x_{1\sigma_1},\ldots,x_{n\sigma_n}.\ t_{bool}\rangle$, as remarked at the beginning of Section 2.5, one has that $[\![\alpha s,xs.t]\!]_{M'} = [\![\alpha s,xs.t]\!]_M$. Hence by definition of the s_i, for all $Xs\in\mathcal{U}^m$

$$[\![\alpha s,xs.t]\!]_{M'}(Xs)(s_1(Xs),\ldots,s_n(Xs)) = 1$$

Then using Lemma 4 in Section 1.3.3 on the semantics of substitution together with the definition of $[\![\mathsf{c}_i]\!]_{M'}$, one finally obtains that for all $Xs\in\mathcal{U}^m$

$$[\![t[\mathsf{c}_1,\ldots,\mathsf{c}_n/x_1,\ldots,x_n]]\!]_{M'}(Xs) = 1$$

or in other words that M' satisfies $t[\mathsf{c}_1,\ldots,\mathsf{c}_n/x_1,\ldots,x_n]$. Hence M' is a model of $\mathcal{T}+_{spec}\langle(\mathsf{c}_1,\ldots,\mathsf{c}_n),\lambda x_{1\sigma_1},\ldots,x_{n\sigma_n}.\ t_{bool}\rangle$, as required.

The constants which are asserted to exist in a constant specification are not necessarily uniquely determined. Correspondingly, there may be many different models of $\mathcal{T}+_{spec}\langle(\mathsf{c}_1,\ldots,\mathsf{c}_n),\lambda x_{1\sigma_1},\ldots,x_{n\sigma_n}.\ t_{bool}\rangle$ whose restriction to $\mathcal{T}$ is M; the above construction produces such a model in a uniform manner by making use of the global choice function on the universe.

Extension by a constant definition, $\mathsf{c}_\sigma = t_\sigma$, is a special case of extension by constant specification. For let t' be the formula $x_\sigma = t_\sigma$, where x_σ is a variable not occurring in t_σ. Then clearly $\vdash\ \exists x_\sigma.\ t'$ and one can apply the method of constant specification to obtain the theory

$$\mathcal{T}+_{spec}\langle\mathsf{c},\lambda x_\sigma.\ t'\rangle$$

But since $t'[\mathsf{c}_\sigma/x_\sigma]$ is just $\mathsf{c}_\sigma = t_\sigma$, this extension yields exactly $\mathcal{T}+_{def}\langle\mathsf{c}_\sigma = t_\sigma\rangle$. So as a corollary of the Proposition, one has that for each standard model M of $\mathcal{T}$, there is a standard model M' of $\mathcal{T}+_{def}\langle\mathsf{c}_\sigma = t_\sigma\rangle$ whose restriction to $\mathcal{T}$ is M. In contrast with the case of constant specifications, M' is uniquely determined by M and the constant definition.

2.5.3 Remarks about constants in HOL

Note how Condition (iii) in the definition of a constant specification was needed in the proof that the extension mechanism preserves the property of possessing a standard model. Its role is to ensure that the introduced constants have, via their types, the same dependency on type variables as does the formula loosely specifying them. The situation is the same as that discussed in the Remark in Section 2.5.1. In a sense, what is causing the problem in the example given in that Remark is not so much the method of extension by introducing constants, but rather the syntax of HOL which does not allow constants to depend explicitly on type variables (in the way that type operators can). Thus in the example one would like to introduce a 'polymorphic' constant $\mathsf{c}_{bool}(\alpha)$ explicitly depending upon α, and define it to be $\exists f_{\alpha\to\alpha}.\ \mathsf{One\_One}\ f\ \wedge\ \neg(\mathsf{Onto}\ f)$. Then in the extended theory one could derive $\mathsf{c}_{bool}(ind) = \mathsf{T}$ and $\mathsf{c}_{bool}(bool) = \mathsf{F}$, but now no contradiction results since $\mathsf{c}_{bool}(ind)$ and $\mathsf{c}_{bool}(bool)$ are different.

In the current version of HOL, constants are (name,type)-pairs. One can envision a slight extension of the HOL syntax with 'polymorphic' constants, specified by pairs $(\mathsf{c},\alpha s.\sigma)$ where now $\alpha s.\sigma$ is a type-in-context and the list αs may well contain extra type variables not occurring in σ. Such a pair would give rise to the particular constant term $\mathsf{c}_\sigma(\alpha s)$, and more generally to constant terms $\mathsf{c}_{\sigma'}(\tau s)$ obtained from this one by instantiating the type variables α_i with types τ_i (so σ' is the instance of σ obtained by substituting τs for αs). This new syntax of polymorphic constants is comparable to the existing syntax of compound types (see section 1.2): an n-ary type operator op gives rise to a compound type $(\alpha_1,\ldots,\alpha_n)op$ depending upon n type variables. Similarly, the above syntax of polymorphic constants records how they depend upon type variables (as well as which generic type the constant has).

However, explicitly recording dependency of constants on type variables makes for a rather cumbersome syntax which in practice one would like to avoid where possible. It is possible to avoid it if the type context αs in $(\mathsf{c}, \alpha s.\sigma)$ is actually the *canonical* context of σ, i.e. contains exactly the type variables of σ. For then one can apply Lemma 1 of Section 1.2.3 to deduce that the polymorphic constant $\mathsf{c}_{\sigma'}(\tau s)$ can be abbreviated to the ordinary constant $\mathsf{c}_{\sigma'}$ without ambiguity—the missing information τs can be reconstructed from σ' and the information about the constant c given in the signature. From this perspective, the rather technical side Conditions (iii) in Sections 2.5.1 and 2.5.2 become rather less mysterious: they precisely ensure that in introducing new constants one is always dealing just with canonical contexts, and so can use ordinary constants rather than polymorphic ones without ambiguity. In this way one avoids complicating the existing syntax at the expense of restricting somewhat the applicability of these theory extension mechanisms.

2.5.4 Extension by type definition

Every (monomorphic) type σ in the initial theory `INIT` determines a set $[\![\sigma]\!]$ in the universe $\mathcal{U}$. However, there are many more sets in $\mathcal{U}$ than there are types in `INIT`. In particular, whilst $\mathcal{U}$ is closed under the operation of taking a non-empty subset of $[\![\sigma]\!]$, there is no corresponding mechanism for forming a 'subtype' of σ. Instead, subsets are denoted indirectly via characteristic functions, whereby a closed term p of type $\sigma{\to}bool$ determines the subset $\{x \in [\![\sigma]\!] : [\![p]\!](x) = 1\}$ (which is a set in the universe provided it is non-empty). However, it is useful to have a mechanism for introducing new types which are subtypes of existing ones. Such types are defined in HOL by introducing a new type constant and asserting an axiom that characterizes it as denoting a set in bijection (i.e. one-to-one correspondence) with a non-empty subset of an existing type (called the *representing type*). For example, the type `num` is defined to be equal to a countable subset of the type `ind`, which is guaranteed to exist by the axiom `INFINITY_AX` (see Section 2.4.3).

As well as defining types, it is also convenient to be able to define type operators. An example would be a type operator *inj* which mapped a set to the set of one-to-one (i.e. injective) functions on it. The subset of $\sigma{\to}\sigma$ representing $(\sigma)inj$ would be defined by the predicate `One_One`. Another example would be a binary cartesian product type operator *prod*. This is defined by choosing a representing type containing two type variables, say $\sigma[\alpha_1;\alpha_2]$, such that for any types σ_1 and σ_2, a subset of $\sigma[\sigma_1;\sigma_2]$ represents the cartesian product of σ_1 and σ_2. The details of such a definition are given in Section 4.3.

Types in HOL must denote non-empty sets. Thus it is only consistent to define a new type isomorphic to a subset specified by a predicate p, if there is at least one thing for which p holds, i.e. $\vdash\ \exists x.\ p\ x$. For example, it would be inconsistent to define a binary type operator *iso* such that $(\sigma_1, \sigma_2)iso$ denoted the set of one-to-one functions from σ_1 *onto* σ_2 because for some values of σ_1 and σ_2 the set would be empty; for example $(ind, bool)iso$ would denote the empty set. To avoid this, a precondition of defining a new type is that the representing subset is non-empty.

To summarize, a new type is defined by:

1. Specifying an existing type.
2. Specifying a subset of this type.
3. Proving that this subset is non-empty.
4. Specifying that the new type is isomorphic to this subset.

In more detail, defining a new type $(\alpha_1, \ldots, \alpha_n)op$ consists in:

1. Specifying a type-in-context, $\alpha_1, \ldots, \alpha_n.\sigma$ say. The type σ is called the *representing type*, and the type $(\alpha_1, \ldots, \alpha_n)op$ is intended to be isomorphic to a subset of σ.
2. Specifying a closed term-in-context, $\alpha_1, \ldots, \alpha_n, .p$ say, of type $\sigma{\rightarrow}bool$. The term p is called the *characteristic function*. This defines the subset of σ to which $(\alpha_1, \ldots, \alpha_n)op$ is to be isomorphic.[6]
3. Proving $\vdash\ \exists x_\sigma.\ p\ x$.
4. Asserting an axiom saying that $(\alpha_1, \ldots, \alpha_n)op$ is isomorphic to the subset of σ selected by p.

To make this formal, the theory `LOG` provides the polymorphic constant Type_Definition defined in Section 2.4.2. The formula $\exists f_{(\alpha_1,\ldots,\alpha_n)op\rightarrow\sigma}.$ Type_Definition $p\ f$ asserts that there exists a one-to-one map f from $(\alpha_1, \ldots, \alpha_n)op$ onto the subset of elements of σ for which p is true. Hence, the axiom that characterizes $(\alpha_1, \ldots, \alpha_n)op$ is:

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

Defining a new type $(\alpha_1, \ldots, \alpha_n)op$ in a theory $\mathcal{T}$ thus consists of introducing op as a new n-ary type operator and the above axiom as a new axiom. Formally, a *type definition* for a theory $\mathcal{T}$ is given by

Data

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

Conditions

[6]The reason for restricting p to be closed, i.e. to have no free variables, is that otherwise for consistency the defined type operator would have to *depend* upon (i.e. be a function of) those variables. Such dependent types are not (yet!) a part of the HOL system.

(i) (op, n) is not the name of a type constant in $\mathsf{Struc}_{\mathcal{T}}$.

(ii) $\alpha_1, \ldots, \alpha_n.\sigma$ is a type-in-context with $\sigma \in \mathsf{Types}_{\mathcal{T}}$.

(iii) $p_{\sigma \to bool}$ is a closed term in $\mathsf{Terms}_{\mathcal{T}}$ whose type variables occur in $\alpha_1, \ldots, \alpha_n$.

(iv) $\exists x_\sigma.\ p\ x \ \in\ \mathsf{Theorems}_{\mathcal{T}}$.

The extension of a standard theory $\mathcal{T}$ by a such a type definition is denoted by

$$\mathcal{T} +_{tydef} \langle (\alpha_1, \ldots, \alpha_n)op, \sigma, p \rangle$$

and defined to be the theory

$$\begin{aligned}&\langle \mathsf{Struc}_{\mathcal{T}} \cup \{(op, n)\},\\ &\ \mathsf{Sig}_{\mathcal{T}},\\ &\ \mathsf{Axioms}_{\mathcal{T}} \cup \{\exists f_{(\alpha_1,\ldots,\alpha_n)op \to \sigma}.\ \mathsf{Type\_Definition}\ p\ f\},\\ &\ \mathsf{Theorems}_{\mathcal{T}} \rangle\end{aligned}$$

Proposition *The theory $\mathcal{T} +_{tydef} \langle (\alpha_1, \ldots, \alpha_n)op, \sigma, p \rangle$ has a standard model if the theory $\mathcal{T}$ does.*

Instead of giving a direct proof of this result, it will be deduced as a corollary of the corresponding proposition in the next section.

2.5.5 Extension by type specification[7]

The type definition mechanism allows one to introduce new types by giving a concrete representation of the type as a 'subtype' of an existing type. One might instead wish to introduce a new type satisfying some property without having to give an explicit representation for the type. For example, one might want to extend `INIT` with an atomic type one satisfying $\vdash\ \forall f_{\alpha \to one}\ g_{\alpha \to one}.\ f = g$ without choosing a specific type in `INIT` and saying that one is in bijection with a one-element subset of it. (The idea being that the choice of representing type is irrelevant to the properties of one that can be expressed in HOL.) The mechanism described in this section provides one way of achieving this while at the same time preserving the all-important property of possessing a standard model and hence maintaining consistency.

Each closed formula q involving a single type variable α can be thought of as specifying a property $q[\tau/\alpha]$ of types τ. Its interpretation in a model is of the form

$$[\![\alpha, .q]\!] \in \prod_{X \in \mathcal{U}} [\![\alpha.bool]\!](X) \ =\ \prod_{X \in \mathcal{U}} 2 \ =\ \mathcal{U} \to 2$$

[7]This theory extension mechanism is not implemented in Version 2.0 of the HOL system. It was proposed by T. Melham and refines a suggestion from R. Jones and R. Arthan.

which is a characteristic function on the universe, determining a subset $\{X \in \mathcal{U} : [\![\alpha, .q]\!](X) = 1\}$ consisting of those sets in the universe for which the property q holds. The most general way of ensuring the consistency of introducing a new atomic type ν satisfying $q[\nu/\alpha]$ would be to prove '$\exists\alpha.\ q$'. However, such a formula with quantification over types is not[8] a part of the HOL logic and one must proceed indirectly—replacing the formula by (a logically weaker) one that can be expressed formally with HOL syntax. The formula used is

$$(\exists f_{\alpha\to\sigma}.\ \mathsf{Type\_Definition}\ p\ f)\ \Rightarrow\ q$$

where σ is a type, $p_{\sigma\to bool}$ is a closed term and neither involve the type variable α. This formula says 'q holds of any type which is in bijection with the subtype of σ determined by p'. If this formula is provable and if the subtype is non-empty, i.e. if

$$\exists x_\sigma.\ p\ x$$

is provable, then it is consistent to introduce an extension with a new atomic type ν satisfying $q[\nu/\alpha]$.

In giving the formal definition of this extension mechanism, two refinements will be made. Firstly, σ is allowed to be polymorphic and hence a new type constant of appropriate arity is introduced, rather than just an atomic type. Secondly, the above existential formulas are permitted to be proved (in the theory to be extended) from some hypotheses.[9] Thus a *type specification* for a theory $\mathcal{T}$ is given by

Data

$$\langle(\alpha_1,\dots,\alpha_n)op, \sigma, p, \alpha, \Gamma, q\rangle$$

Conditions

(i) (op, n) is a type constant that is not in $\mathsf{Struc}_\mathcal{T}$.

(ii) $\alpha_1,\dots,\alpha_n.\sigma$ is a type-in-context with $\sigma \in \mathsf{Types}_\mathcal{T}$.

(iii) $p_{\sigma\to bool}$ is a closed term in $\mathsf{Terms}_\mathcal{T}$ whose type variables occur in $\alpha s = \alpha_1,\dots,\alpha_n$.

(iv) α is a type variable distinct from those in αs.

(v) Γ is a list of closed formulas in $\mathsf{Terms}_\mathcal{T}$ not involving the type variable α.

(vi) q is a closed formula in $\mathsf{Terms}_\mathcal{T}$.

[8]yet!

[9]This refinement increases the applicability of the extension mechanism without increasing its expressive power. A similar refinement could have be made to the other theory extension mechanisms.

(vii) The sequents

$$(\Gamma \quad , \quad \exists x_\sigma .\ p\ x)$$
$$(\Gamma \quad , \quad (\exists f_{\alpha\to\sigma}.\ \mathsf{Type\_Definition}\ p\ f) \ \Rightarrow\ q)$$

are in $\mathsf{Theorems}_{\mathcal{T}}$.

The extension of a standard theory $\mathcal{T}$ by such a type specification is denoted

$$\mathcal{T} +_{tyspec} \langle (\alpha_1, \ldots, \alpha_n)op, \sigma, p, \alpha, \Gamma, q\rangle$$

and is defined to be the theory

$$\begin{array}{l}\langle \mathsf{Struc}_{\mathcal{T}} \cup \{(op, n)\}, \\ \ \mathsf{Sig}_{\mathcal{T}}, \\ \ \mathsf{Axioms}_{\mathcal{T}} \cup \{(\Gamma, q[(\alpha_1, \ldots, \alpha_n)op/\alpha])\}, \\ \ \mathsf{Theorems}_{\mathcal{T}}\rangle\end{array}$$

Example To carry out the extension of `INIT` mentioned at the start of this section, one forms

$$\mathtt{INIT} +_{tyspec} \langle ()one, bool, p, \alpha, \emptyset, q\rangle$$

where p is the term $\lambda b_{bool}.\ b$ and q is the formula $\forall f_{\beta\to\alpha}\ g_{\beta\to\alpha}.\ f = g$. Thus the result is a theory extending `INIT` with a new type constant one satisfying the axiom $\forall f_{\beta\to one}\ g_{\beta\to one}.\ f = g$.

To verify that this is a correct application of the extension mechanism, one has to check Conditions (i) to (vii) above. Only the last one is non-trivial: it imposes the obligation of proving two sequents from the axioms of `INIT`. The first sequent says that p defines an inhabited subset of $bool$, which is certainly the case since T witnesses this fact. The second sequent says in effect that any type α that is in bijection with the subset of $bool$ defined by p has the property that there is at most one function to it from any given type β; the proof of this from the axioms of `INIT` is left as an exercise.

Proposition *The theory $\mathcal{T} +_{tyspec} \langle (\alpha_1, \ldots, \alpha_n)op, \sigma, p, \alpha, \Gamma, q\rangle$ has a standard model if the theory $\mathcal{T}$ does.*

Proof Write αs for $\alpha_1, \ldots, \alpha_n$, and suppose that $\alpha s' = \alpha'_1, \ldots, \alpha'_m$ is the list of type variables occurring in Γ and q, but not already in the list $\alpha s, \alpha$.

Suppose M is a standard model of $\mathcal{T}$. Since $\alpha s, .p$ is a term-in-context of type $\sigma\to bool$, interpreting it in M yields

$$[\![\alpha s, .p]\!]_M \in \prod_{Xs\in\mathcal{U}^n} [\![\alpha s.\sigma\to bool]\!]_M(Xs) = \prod_{Xs\in\mathcal{U}^n} [\![\alpha s.\sigma]\!]_M(Xs)\to 2.$$

There is no loss of generality in assuming that Γ consists of a single formula γ. (Just replace Γ by the conjunction of the formulas it contains, with the convention that this conjunction is $\top$ if Γ is empty.) By assumption on $\alpha s'$ and by Condition (iv), $\alpha s, \alpha s', .\gamma$ is a term-in-context. Interpreting it in M yields

$$[\![\alpha s, \alpha s'.\gamma]\!]_M \in \prod_{(Xs,Xs')\in\mathcal{U}^{n+m}} [\![\alpha s, \alpha s'.bool]\!]_M(Xs, Xs') = \mathcal{U}^{n+m}\to 2$$

Now $(\gamma, \exists x_\sigma.\ p\ x)$ is in $\mathsf{Theorems}_\mathcal{T}$ and hence by the Soundness Theorem 2.3.2 this sequent is satisfied by M. Using the semantics of $\exists$ given in Section 2.4.2 and the definition of satisfaction of a sequent from Section 2.2, this means that for all $(Xs, Xs') \in \mathcal{U}^{n+m}$ if $[\![\alpha s, \alpha s'.\gamma]\!]_M(Xs, Xs') = 1$, then the set

$$\{y \in [\![\alpha s.\sigma]\!]_M\ :\ [\![\alpha s, .p]\!](Xs)(y) = 1\}$$

is non-empty. (This uses the fact that p does not involve the type variables $\alpha s'$, so that by Lemma 4 in Section 1.3.3 $[\![\alpha s, \alpha s'.p]\!]_M(Xs, Xs') = [\![\alpha s, .p]\!]_M(Xs)$.) Since it is also a subset of a set in $\mathcal{U}$, it follows by property **Sub** of the universe that this set is an element of $\mathcal{U}$. So defining

$$S(Xs) = \begin{cases} \{y \in [\![\alpha s.\sigma]\!]_M\ :\ [\![\alpha s, .p]\!](Xs)(y) = 1\} & \text{if } [\![\alpha s, .\gamma]\!]_M(Xs, Xs') = 1, \text{ some } Xs' \\ 1 & \text{otherwise} \end{cases}$$

one has that S is a function $\mathcal{U}^n\to\mathcal{U}$. Extend M to a model of the signature of $\mathcal{T}'$ by defining its value at the new n-ary type constant op to be this function S. Note that the values of σ, p, γ and q in M' are the same as in M, since these expressions do not involve the new type constant op.

For each $Xs \in \mathcal{U}^n$ define i_{Xs} to be the inclusion function for the subset $S(Xs) \subseteq [\![\alpha s.\sigma]\!]_M$ if $[\![\alpha s, \alpha s'.\gamma]\!]_M(Xs, Xs') = 1$ for some Xs', and otherwise to be the function $1\to[\![\alpha s.\sigma]\!]_M$ sending $0 \in 1$ to $\mathrm{ch}([\![\alpha s.\sigma]\!]_M)$. Then $i_{Xs} \in (S(Xs)\to[\![\alpha s.\sigma]\!]_{M'}(Xs))$ because $[\![\alpha s.\sigma]\!]_{M'} = [\![\alpha s.\sigma]\!]_M$. Using the semantics of $\mathsf{Type\_Definition}$ given in Section 2.4.2, one has that for any $(Xs, Xs') \in \mathcal{U}^{n+m}$, if $[\![\alpha s, \alpha s'.\gamma]\!]_{M'}(Xs, Xs') = 1$ then

$$[\![\mathsf{Type\_Definition}]\!]_{M'}([\![\alpha s.\sigma]\!]_{M'}, S(Xs))([\![\alpha s, .p]\!]_{M'})(i_{Xs}) = 1.$$

Thus M' satisfies the sequent

$$(\gamma\ ,\ \exists f_{(\alpha s)op\to\sigma}.\ \mathsf{Type\_Definition}\ p\ f).$$

But since the sequent $(\gamma, (\exists f_{\alpha\to\sigma}.\ \mathsf{Type\_Definition}\ p\ f)\ \Rightarrow\ q)$ is in $\mathsf{Theorems}_\mathcal{T}$, it is satisfied by the model M and hence also by the model M' (since the sequent does not involve the new type constant op). Instantiating α to $(\alpha s)op$ in this sequent (which is permissible since by Condition (iv) α does not occur in γ), one thus has that M' satisfies the sequent

$$(\gamma\ ,\ (\exists f_{(\alpha s)op\to\sigma}.\ \mathsf{Type\_Definition}\ p\ f) \Rightarrow q[(\alpha s)op/\alpha]).$$

Applying Modus Ponens, one concludes that M' satisfies $(\gamma\ ,\ q[(\alpha s)op/\alpha])$ and therefore M' is a model of $\mathcal{T}'$, as required.

An extension by type definition is in fact a special case of extension by type specification. To see this, suppose $\langle(\alpha_1,\ldots,\alpha_n)op,\ \sigma,\ p_{\sigma\to bool}\rangle$ is a type definition for a theory $\mathcal{T}$. Choosing a type variable α different from $\alpha_1,\ldots,\alpha_n$, let q denote the formula

$$\exists f_{\alpha\to\sigma}.\ \mathsf{Type\_Definition}\ p\ f$$

Then $\langle(\alpha_1,\ldots,\alpha_n)op,\sigma,p,\alpha,\emptyset,q\rangle$ satisfies all the conditions necessary to be a type specification for $\mathcal{T}$. Since $q[(\alpha_1,\ldots,\alpha_n)op/\alpha]$ is just $\exists f_{(\alpha_1,\ldots,\alpha_n)op\to\sigma}.\ \mathsf{Type\_Definition}\ p\ f$, one has that

$$\mathcal{T}+_{tydef}\langle(\alpha_1,\ldots,\alpha_n)op,\sigma,p\rangle = \mathcal{T}+_{tyspec}\langle(\alpha_1,\ldots,\alpha_n)op,\sigma,p,\alpha,\emptyset,q\rangle$$

Thus the Proposition in Section 2.5.4 is a special case of the above Proposition.

In an extension by type specification, the property q which is asserted of the newly introduced type constant need not determine the type constant uniquely (even up to bijection). Correspondingly there may be many different standard models of the extended theory whose restriction to $\mathcal{T}$ is a given model M. By contrast, a type definition determines the new type constant uniquely up to bijection, and any two models of the extended theory which restrict to the same model of the original theory will be isomorphic.

Part II

The HOL System

Chapter 3

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 3.3); the quotation, printing and parsing of logical terms (Section 3.4); the representation of theorems (Section 3.6); the representation of theories (Section 3.7); some useful HOL theories (Sections **??** and 3.9); the methods for extending theories (throughout Section **??** and in Section 5.7); and the ML system functions concerning the logic (Section 6.3). 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 ML types `hol_type` and `term` to represent types and terms of the HOL logic, as defined in Sections 1.2 and 1.3, respectively. It also provides primitive ML functions for creating and manipulating values of these types. The key idea of the HOL system, due to Robin Milner, and discussed in this chapter, is that theorems are represented as an abstract ML type whose only pre-defined values are axioms, and whose only operations are rules of inference. This means that the only way to construct theorems in HOL is to apply rules of inference to axioms or existing theorems; hence the consistency of the logic is preserved.

The purpose of the meta-language ML is to provide a programming environment in which to build theorem proving tools to assist in the construction of proofs. When the HOL system is built, a range of useful theorems 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.

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 3.1.1 below). The use of non-identifiers as variable names is discouraged except in special circumstances (for example, when writing derived rules that generate variables with names that are guaranteed to be different from 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 3.1.1.3 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.

3.1.1 Identifiers

A HOL identifier can be of two forms:

(i) A finite sequence of alphanumerics starting with a letter.

(ii) A *symbolic* identifier, i.e., a finite sequence formed by any combination of the following characters:

```
# ? + * / \ = < > & % @ ! , : ; _ | ~ -
```

A letter is a member of the list:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

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

Alphanumerics are letters or digits or underscores (`_`) or primes (`'`). A digit is one of `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`. A *number* is a string of one or more digits.

3.1.1.1 Separators

The separators used by the HOL lexical analyser are (with ascii codes in brackets):

space (32), carriage return (13), line feed (10), tab (^I, 9), form feed (^L, 12)

3.1.1.2 Special identifiers

The following valid identifiers should not be used as the name of a variable or a constant.

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

3.1.1.3 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
```

3.2 Types

The allowed types depend on which type constants have been declared in the current theory. See Section 3.7 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_type    : (string * hol_type list) -> hol_type
```

The function `mk_vartype` constructs a type variable with a given name; it fails if the name is not an allowable type variable name (i.e. not a ' followed by an alphanumeric).

The function `mk_type` constructs a compound type from a string representing the name of the type operator and a list of types representing the arguments to the operator. Function types $\sigma_1 \to \sigma_2$ of the logic are represented in ML as though they were compound types (σ_1, σ_2)`fun` (in Section 1.2, however, function types were not regarded as compound types).

The evaluation of `mk_type("`$name$`", [`σ_1`,`$\cdots$`,`σ_n`])` fails if

(i) $name$ is not a type operator of the current theory;

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

For example, `mk_type("bool",[])` evaluates to an ML value of type term representing the type *bool* and `mk_type("fun", [mk_type("ind",[]), mk_type("bool",[])])` evaluates to a value representing $ind \to bool$. (These types are introduced in Section **??**).

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

```
dest_vartype : hol_type -> string
dest_type    : hol_type -> (string * hol_type list)
```

The function `dest_vartype` extracts the name of a type variable. The function `dest_type` destructs a compound type into the name of the type operator and a list of the argument types; `dest_vartype` and `dest_type` are thus the inverses of `mk_vartype` and `mk_type`, respectively. The destructors fail on arguments of the wrong form.

Types are printed in the form `` `: ⋯ ` `` using the quotation syntax described in Section 3.4. For example, the ML value of type `hol_type` representing $ind \to (ind \to bool)$ would be printed as `` `:ind -> ind -> bool` ``.

3.3 Terms

The four primitive kinds of terms of the logic are described in Section 1.3. The ML functions for manipulating these are described in this section. There are also various derived terms that are described in Section 3.5.1.

The allowed terms depend on which constants have been declared in the current theory. See Section 3.7 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_const : (string * hol_type) -> term
```

`mk_const(`c`,`σ`)` evaluates to a term representing the constant with name c and type σ; it fails if:

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

(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 3.7).

```
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_const : term -> (string * hol_type)
dest_comb  : term -> (term * term)
dest_abs   : term -> (term * term)
```

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

The function

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

returns the type of a term. It could be defined (recursively) in terms of the destructors but is predefined for convenience.

Terms are printed in the form ` ‘ ··· ‘ ` using the quotation syntax described in Section 3.4. For example, the term representing

$$\forall x\ y.\ x < y \Rightarrow \exists z.\ x + z = y$$

would be printed as:

```
‘!x y.  x < y ==> ?z.  x + z = y‘
```

Note that a colon is used to distinguish type quotation from term quotation; the former have the form ` ‘: ··· ‘ ` and the latter have the form ` ‘ ··· ‘ `.

3.4 Quotation

HOL types and terms can be input to the system in two ways: by using constructor functions, or by using *quotation*. The former allows some terms to be built which cannot be constructed using quotation. For example, a term containing two variables with the same name but different types, e.g. the term $x_{bool} = (x_{num} = 1)$, can be built only by using constructors.

It would be tedious, however, to always have to input types and terms using the constructor functions. The HOL system, adapting the approach taken in LCF, , has special quotation parsers for HOL types and terms (named `Type` and `Term`, respectively) which enables types and terms to be input using a fairly standard syntax. The HOL printer also outputs types and terms using this syntax.

For example, the ML expression

```
Type ‘:bool -> bool‘
```

denotes exactly the same value (of ML type `type`) as

```
mk_type("fun",[mk_type("bool",[]), mk_type("bool",[])])
```

and

```
Term ‘\x.x+1‘
```

can be used instead of[1]

[1] In order to be processed successfully, the latter quotation (which features a numeral) requires the theory of arithmetic to have already been loaded. This can be accomplished by `load "arithmeticTheory"`.

```
mk_abs
 (mk_var("x",mk_type("num",[])),
  mk_comb
  (mk_comb
   (mk_const
    ("+",
     mk_type("fun",[mk_type("num",[]),
                    mk_type("fun",[mk_type("num",[]),
                                   mk_type("num",[])])]))),
    mk_var("x", mk_type("num",[]))),
   mk_numeral (Arbnum.fromString "1")))
```

It should be noted that there is no explicit type information in `\x.x+1`. The HOL type checker knows that `1` has type `num` and `+` has type `num->(num->num)`. From this information it can infer that both occurrences of `x` in `\x.x+1` could have type `num`. This is not the only possible type assignment; for example, the first occurrence of `x` could have type `bool` and the second one have type `num`. In that case there would be two *different* variables with name `x`, namely $\mathtt{x}_{\mathtt{bool}}$ and $\mathtt{x}_{\mathtt{num}}$, the second of which is free. In fact, as mentioned, the only way to construct a term with this second type assignment is by using constructors, since the type checker uses the heuristic that all variables in a term with the same name have the same type. This is illustrated in the following session.

```
- Term `x = (x = 1)`;                                                   1
Type inference failure: unable to infer a type for the application of

$= (x :num)

which has type

:num -> bool

to

(x :num) = (1 :num)

which has type

:bool

unification failure message: unify failed

- mk_eq
   (mk_var("x",mk_type("bool",[])),
    mk_eq
     (mk_var("x",mk_type("num",[])),
      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, `` Term `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.

The type inference algorithm used for the HOL logic is almost identical to that used for ML. For example, the ML expression `fn x => x` will be ascribed ML type `'a -> 'a`, and the HOL term constructed by `` Term `\x.x` `` will get an analogous type, as shown in the session below. This session also shows that a HOL term has both an ML type (namely `hol_type`) and a HOL type (`: 'a -> 'a` in this case).

```
- Term `\x. x`;                                                          2
<<HOL message: inventing new type variable names: 'a.>>
> val it = `\x. x` : term

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

For terms of polymorphic type, i.e., terms whose types have type variables, the type checker will invent names for the type variables (as in the above session). This is further shown in the following session (in which we first tell the HOL printer to output type information):

```
- show_types := true;                                                    3
> val it = () : unit

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

In this example, `x` is unconstrained in the term `f x`, since it appears only as an argument. The system assigns it the type variable `'a`. On the other hand, `f` is a function, since it is applied to `x`. Thus `f` has a function type, the domain of which is `'a`; moreover, since the result of the application is also unconstrained, the range of the function type is chosen to be the next type variable different from `'a`, i.e., `'b`.

Allowing the system to invent type variables introduces a degree of non-determinism that may not be suitable for some applications. In such cases, explicit type constraints should be used. The system can be prevented from inventing type variables by setting the flag `Globals.guessing_tyvars` to `false`.

3.4.1 Overloading

A limited amount of overloading resolution is performed by the quotation parser for terms. For example, the tilde symbol (~) denotes boolean negation in the initial theory of HOL and it also denotes the additive inverse in the `integer` and `real` theories. If we load the `integer` theory and enter an ambiguous term featuring `~`, the system will inform us that overloading resolution is being performed.

```
- load "integerTheory";                                                    1
> val it = () : unit

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

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

A priority mechanism is used to resolve multiple possible choices. In the example, `~` could be consistently chosen to have type `:bool -> bool` or `:int -> int`, and the mechanism has chosen the former. For finer control, explicit type constraints may be used. In the following session, the `~~x` in the first quotation has type `:bool`, while in the second, a type constraint ensures that `~~x` has type `:int`.

```
- show_types := true;                                                      2
> val it = () : unit

- Term `~(x = ~~x)`;
<<HOL message: more than one resolution of overloading was possible.>>
> val it = `~((x :bool) = ~~x)` : term

- Term `~(x:int = ~~x)`;
> val it = `~((x :int) = ~~x)` : term
```

Note that the symbol `~` stands for two different constants in the second quotation; its first occurrence is boolean negation, while the other two occurrences are the additive inverse operation for integers. For more information on how to set up and use overloading, consult *REFERENCE*.

3.4.2 Antiquotation

Within a quotation, expressions of the form `^`(t) (where t is an ML expression of type `term` or `type`) are called *antiquotations*. An antiquotation `^`(t) evaluates to the ML value of t. For example, `` Term `x \/ ^(mk_conj(Term`y:bool`, Term`z:bool`))` `` evaluates to the same term as `` Term `x \/ (y /\ z)` ``. The most common use of antiquotation is when the term t is just an ML variable x. In this case `^`(x) can be abbreviated by `^`x.

The following session illustrates antiquotation.

```
- load "arithmeticTheory";                                          1
> val it = () : unit

- val y = Term `x+1`;
> val y = `x + 1` : term

val z = Term `y = ^y`;
> val z = `y = x + 1` : term

- Term `!x:num.?y:num.^z`;
> val it = `!x. ?y. y = x + 1` : term
```

Types may be antiquoted as well:

```
- val pred = Type `:'a -> bool`;                                    1
> val pred = `:'a -> bool` : hol_type

- Type`:^pred -> bool`;
> val it = `:('a -> bool) -> bool` : hol_type
```

One requirement of the system is that antiquoting a type into a term quotation requires the use of `ty_antiq`. For example,

```
- Term `!P:^pred. P x ==> Q x`;                                     2

! Toplevel input:
! Term `!P:^pred. P x ==> Q x`;
!           ^^^^
! Type clash: expression of type
!   hol_type
! cannot have type
!   term

- Term `!P:^(ty_antiq pred). P x ==> Q x`;
> val it = `!P. P x ==> Q x` : term
```

3.5 Ways to construct types and terms

The table below shows ML expressions for various kinds of type quotations. The expressions in the same row are equivalent.

Types		
Kind of type	*ML quotation*	*Constructor expression*
Type variable	`:` $'alphanum$	`mk_vartype("`$'alphanum$`")`
Type constant	$:op$	`mk_type("`op`",[])`
Function type	$:\sigma_1$ `->` σ_2	`mk_type("fun", [`σ_1 `,` σ_2`])`
Compound type	`:(`σ_1`, ... ,` σ_n`)`op	`mk_type("`op`", [`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`)`
Combination	$t_1\ t_2$	`mk_comb(`t_1`,` t_2`)`
Abstraction	`\`x`.`t	`mk_abs(`x`,` t`)`

3.5.1 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 3.7) (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 constant name might have. This is illustrated in the table below, in which the terms in the column headed '*ML quotation*' are translated by the quotation parser to the corresponding terms in the column headed '*Primitive term*'. Conversely, the terms in the latter column are always printed in the form shown in the former one. The ML constructor expressions in the rightmost column evaluate to the same values (of type `term`) as the other quotations in the same row.

Non-primitive terms

Kind of term	*ML quotation*	*Primitive term*	*Constructor expression*
Negation	~t	$~ t	mk_neg(t)
Disjunction	t_1\/t_2	$\/ t_1 t_2	mk_disj(t_1,t_2)
Conjunction	t_1/\t_2	$/\ t_1 t_2	mk_conj(t_1,t_2)
Implication	t_1==>t_2	$==> t_1 t_2	mk_imp(t_1,t_2)
Equality	t_1=t_2	$= t_1 t_2	mk_eq(t_1,t_2)
∀-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	(t=>t_1|t_2)	COND t t_1 t_2	mk_cond(t,t_1,t_2)
let-expression	let x=t_1 in t_2	LET(\\x.t_2)t_1	mk_let(\\x.t_1,t_2)

There are constructors, destructors and indicators for all the obvious constructs. (Indicators, e.g. is_neg, return truth values indicating whether or not a term belongs to the syntax class in question.) In addition to the constructors listed in the table there are constructors, 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 Sections **??** and 4.3.2, respectively. The constants \/, /\, ==> and = are examples of *infixes* and represent $\vee$, $\wedge$, $\Rightarrow$ and equality, respectively. If c is declared to be an infix, then the HOL parser will translate $t_1\ c\ t_2$ to \$$c$ t_1 t_2.

The constants !, ? and @ are examples of *binders* and represent $\forall$, $\exists$ and ε, respectively. If c is declared to be a binder, then the HOL parser will translate $c\,x\,.\,t$ to the combination \$$c$(\\$x$.$t$) (i.e. the application of the constant c to the representation of the abstraction $\lambda x.\ t$).

In addition to the kinds of terms in the tables above, the parser also supports the following syntactic abbreviations.

Syntactic abbreviations

Abbreviated term	*Meaning*	*Constructor expression*
$t\ t_1 \cdots t_n$	$(\cdots(t\ t_1)\cdots t_n)$	list_mk_comb(t,[t_1, ... ,t_n])
\\$x_1 \cdots x_n$.$t$	\\$x_1$. ··· \\$x_n$.$t$	list_mk_abs([x_1, ... ,x_n],t)
!$x_1 \cdots x_n$.t	!x_1. ··· !x_n.t	list_mk_forall([x_1, ... ,x_n],t)
?$x_1 \cdots x_n$.t	?x_1. ··· ?x_n.t	list_mk_exists([x_1, ... ,x_n],t)

There are also constructors list_mk_conj, list_mk_disj, list_mk_imp and list_mk_pair for conjunctions, disjunctions, implications and tuples respectively. The corresponding destructor functions are called `strip_comb`, etc.,

3.6 Theorems

In Chapter 1, the notion of deduction was introduced in terms of *sequents*, where a sequent is a pair whose second component is a formula being asserted (a conclusion), and whose first component is a set of formulas (hypotheses). Based on this was the notion of a *deductive system*: a set of pairs, whose second component is a sequent, and whose first component is a sequent list[2]. The concept of a sequent *following from* a set of sequents via a deductive system was then defined: a sequent follows from a set of sequents if the sequent is the last element of some chain of sequents, each of whose elements is either in the set, or itself follows from the set along with earlier elements of the chain, via the deductive system.

A notation for 'follows from' was then introduced. That a sequent $(\{t_1, \ldots, t_n\},\ t)$ follows from a set of sequents Δ, via a deductive system $\mathcal{D}$, is denoted by: $t_1, \ldots, t_n \vdash_{\mathcal{D},\Delta} t$. (It was noted that where either $\mathcal{D}$ or Δ were clear by context, their mention could be omitted; and where the set of hypotheses was empty, its mention could be omitted.)

A sequent that follows from the empty set of sequents via a deductive system is called a *theorem* of that deductive system. That is, a theorem is the last element of a *proof* (in the sense of Chapter 1) from the empty set of sequents. When a pair $(L, (\Gamma, t))$ belongs to a deductive system, and the list L is empty, then the sequent (Γ, t) is called an *axiom*. Any pair $(L, (\Gamma, t))$ belonging to a deductive system is called a *primitive inference* of the system, with hypotheses[3] L and conclusion (Γ, t).

A formula in the abstract is represented concretely in HOL by a term whose HOL type is `":bool"`. Therefore, a term of type `":bool"` is used to represent a member of the set of hypotheses of a sequent; and likewise to represent the conclusion of a sequent. Sets in this context are represented by lists, so the set of hypotheses of a sequent is represented by a list of `` `:bool` ``-typed terms.

A theorem in the abstract is represented concretely in the HOL system by a value with the ML abstract type `thm`. The type `thm` has a primitive destructor function

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

which returns a pair consisting of the hypothesis list and the conclusion, respectively, of a theorem. From this, two destructor functions are derived

```
hyp   : thm -> term list
concl : thm -> term
```

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

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

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 first in Section 3.9, and further in Chapter8.)

It was mentioned in Chapter 1 that the deductive system of HOL includes five axioms[4]. In that Chapter, the axioms were presented in abstract form. The concrete representation of the axioms in HOL is given in Section **??**. To anticipate, the axiom `BOOL_CASES_AX` mentioned in Chapter 1 is printed in HOL as follows (where `T` and `F` are the HOL logic's constants representing truth and falsity, respectively):

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

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

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

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

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

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

In addition to the print conventions mentioned above, the printing of theorems prints hypotheses as periods (i.e. full stops or dots). The flag `show_assums` prints theorems with hypotheses shown in full. These points are illustrated with a theorem inferred, for example purposes, from another axiom mentioned in Chapter 1: `SELECT_AX`.

```
- val th = UNDISCH (SPEC_ALL SELECT_AX);                         1
> val th =  [.] |- P ($@ P) : thm

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

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

[4]This is a simplification: the axioms are an extension of the basic logic. See Sections **??** and **??**.

3.7 Theories

In Chapter 1 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 $\mathcal{T}$.

Theories are structured hierarchically to represent sequences of extensions called *segments* of an initial theory (see Section 2.5) called `min`. A theory segment is not really a logical concept, but rather a concept of the representation of 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.

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).

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
```

On startup, the current theory segment of HOL is named `scratch`, which is an empty theory, having the theory `bool` as its sole parent. This is a very simple logical setting; for example, common types such as numbers and pairs are not present. Typically, a user would begin by loading whatever specific logical context is required.

3.7.1 Primitive ML functions for creating theories

The ML functions for creating theories and manipulating are listed below.

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

One creates a new theory segment by a call to `new_theory`. 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.

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 are now held to be *out-of-date*, and will 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.

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.

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

Executing `new_type` n `"`*op*`"` makes *op* a new n-ary type operator in the current theory. Failure if:

(i) there already exists a type operator named op in an ancestor theory segment.

(ii) *op* is not an allowed name for a type.

```
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. Failure if:

(i) there already exists a constant named c in an ancestor theory segment.

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

Executing `new_infix("`ix`",`σ`)` declares ix to be a new constant with generic type σ and infix status. Failure if:

(i) there already exists a constant named ix in an ancestor theory segment;

(ii) σ not of the form σ_1`->`σ_2`->`σ_3.

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

Executing `new_binder("`b`",`σ`)` declares b to be a new constant with generic type σ and binder status. Failure if:

(i) there already exists a constant named b an ancestor theory segment;

(ii) σ not of the form `(`σ_1`->`σ_2`)->`σ_3.

```
new_axiom : (string * term) -> thm
```

Executing `new_axiom("`$name$`",`t`)` declares the sequent `(`$\{\}$`,`t`)` to be an axiom of the current theory with name $name$. Failure if:

(i) t contains out-of-date constants or types.

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.

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`, which maintains dependencies among collections of HOL theory segments.

3.7.2 Functions for creating definitional extensions

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

3.7.2.1 Constant definitions

In Section 2.5.1 a constant definition over a signature Σ_Ω is defined to be an equation, i.e. a formula of the form $c_\sigma = t_\sigma$, such that:

(i) c is not the name of any constant in Σ_Ω;

(ii) t_σ is a closed term in $\mathsf{Terms}_{\Sigma_\Omega}$;

(iii) all the type variables occurring in t_σ occur in σ.

In HOL, definitions can be slightly more general than this, in that an equation:

$$c\, v_1 \ \cdots \ v_n \ = \ t$$

is allowed to be a definition where v_1, $\ldots$, 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 Section 2.5.1 if (i), (ii) and (iii) hold.

The following ML function creates a new definition in the current theory.

```
new_definition : (string * term) -> thm
```

Evaluating `new_definition("`$name$`", ` $c\ v_1\ \cdots\ v_n\ =\ t$`)`, where c is not already a constant, declares the sequent `({},`$\lambda v_1\ \cdots\ v_n.\ t$`)` to be a constant definition of the current theory. The name associated with the definition in this theory is $name$. Failure if:

(i) c is already a constant in an ancestor current theory;

(ii) t contains free variables that are not in any of the variable structures v_1, ..., v_n (this is equivalent to requiring $\lambda v_1\ \cdots\ v_n.\ t$ to be a closed term);

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

3.7.2.2 Constant specifications

In Section 2.5.2 a constant specification for a theory $\mathcal{T}$ is defined to be a pair:

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

such that:

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

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

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

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

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

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

Evaluating:

```
new_specification
 "name"
 [flag1,"c1", ... , "flagn","cn"]
 |- ?x1 ··· xn. t[x1, ... ,xn]
```

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

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

If $flag_i$ is `constant` then c_i is declared an ordinary constant, if it is `infixl` n then c_i is declared a left associative infix with binding strength n, if it is `infixr` n then c_i is declared a right associative infix with binding strength n, and if it is `binder` then `c`$_i$ is declared a binder. This theorem is stored, with name $name$, as a definition in the current theory segment. A call to `new_specification` fails if:

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

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

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

(iv) some c_i is already a constant in an ancestor theory;

(v) some c_i is not an allowed name for a constant;

(vi) some $flag_i$ is not either `constant`, `infix` or `binder`;

(vii) the type of c_i is not suitable for a constant with the syntactic status specified by $flag_i$;

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

3.7.2.3 Type definitions

In Section 2.5.4 it is explained that defining a new type $(\alpha_1,\ldots,\alpha_n)op$ in a theory $\mathcal{T}$ consists of introducing op as a new n-ary type operator and

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

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 * term * thm) -> thm
```

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

`new_type_definition("`*op*`", `t`, |- ?`x`.` $t\ \ x$`)`

results in *op* being declared as a new n-ary type operator characterized by the definitional axiom:

```
|- ?rep. TYPE_DEFINITION t rep
```

which is stored as a definition with the automatically generated name *op*`_TY_DEF`.. The constant `TYPE_DEFINITION` is defined in the theory `bool` by:

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

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

(i) op is already the name of a type or type operator in an ancestor theory;

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

3.7.2.4 Defining bijections

The result of a type definition using `new_type_definition` is a theorem which asserts only the *existence* of a bijection from the type it defines to the corresponding subset of an existing type. To introduce constants that in fact denote such a bijection and its inverse, the following ML function is provided:

```
define_new_type_bijections : string -> string -> string -> thm -> thm
```

This function takes three string arguments and a theorem argument. The theorem argument must be a definitional axiom of the form returned by `new_type_definition`. The first string argument is the name under which the constant definition (a constant specification, in fact) made by `define_new_type_bijections` will be stored in the current theory segment, and the second and third string arguments are user-specified names for the two constants that are to be defined. These constants are defined so as to denote mutually inverse bijections between the defined type, whose definition is given by the supplied theorem, and the representing type of this defined type.

Evaluating:

```
define_new_type_bijections "name" "abs" "rep"
        |- ?rep:newty->ty. TYPE_DEFINITION P rep
```

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

```
|- (!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) either abs or rep is already the name of a constant in an ancestor theory;

(ii) th is not a theorem of the form returned by `new_type_definition`.

3.7.2.5 Properties of type bijections

The following ML functions are provided for proving that the bijections introduced by `define_new_type_isomorphisms` are injective (one-to-one) and surjective (onto):

```
prove_rep_fn_one_one : thm -> thm
prove_rep_fn_onto    : thm -> thm
prove_abs_fn_one_one : thm -> thm
prove_abs_fn_onto    : thm -> thm
```

The theorem argument to each of these functions must be a theorem of the form returned by `define_new_type_bijections`:

`|- (!a.` abs`(`rep `a) = a) /\ (!r.` P `r = (`rep`(`abs `r) = r))`

If th is a theorem of this form, then evaluating `prove_rep_fn_one_one` th proves that the function rep is one-to-one, and returns the theorem:

`|- !a a'. (`rep `a =` rep `a') = (a = a')`

Likewise, `prove_rep_fn_onto` th proves that rep is onto the set of values that satisfy P:

`|- !r.` P `r = (?a. r =` rep `a)`

Evaluating `prove_abs_fn_one_one` th proves that abs is one-to-one for values that satisfy P, and returns the theorem:

`|- !r r'.` P `r ==>` P `r' ==> ((`abs `r =` abs `r') = (r = r'))`

And evaluating `prove_abs_fn_onto` th proves that abs is onto, returning the theorem:

`|- !a. ?r. (a =` abs `r) /\` P `r`

All four functions will fail if applied to any theorem that does not have the form of a theorem returned by `define_new_type_bijections`. None of these functions saves anything in the current theory.

3.7.3 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
infixes     : string -> term list
binders     : string -> term list
axioms      : string -> (string * thm) list
definitions : string -> (string * thm) list
theorems    : string -> (string * thm) 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; in the case of `axioms`, `definitions` or `theorems` a list of string-theorem pairs is returned, where the string is the name of the theorem representing the axiom, definition or theorem that was supplied by the user. Note that constant specifications and type definitions are both retrieved using the function `definitions`.

Individual axioms, definitions and theorems can be read from the current 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.

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

3.8 The theory `min`

The theory `min` declares the type constant `bool` of booleans, the binary type operator `fun` of functions, and the type constant `ind` of individuals. Building on this, three primitive constants are declared in the theory `min`: equality, implication, and a choice operator.

Equality (`$= : 'a -> 'a -> bool`) parses as an infix with low binding precedence (100).

Implication (`$==> : bool -> bool -> bool`) parses as a right-associative infix with binding precedence 200.

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 [**?**], Leisenring's book on Hilbert's ε-symbol [**?**], or Andrews' textbook on type theory [**?**].

3.9 Primitive rules of inference of the HOL Logic

The primitive rules of inference of the logic were described abstractly in Section 2.3.1. The descriptions relied on meta-variables t, t_1, t_2, and so on. In the HOL logic, infinite families of primitive inferences are grouped together and thought of as single primitive inference schemes. Each family contains all the concrete instances of one particular inference 'pattern'. These can be produced, in abstract form, by instantiating the meta-variables in Section 2.3.1 to concrete terms.

In HOL, primitive inference schemes are represented by ML functions that return theorems as values. That is, for particular HOL terms, the ML functions return the instance of the theorem at those terms. The ML functions are part of the ML abstract type `thm`: although `thm` has no primitive constructors, it has (eight) operations which return theorems as values: `ASSUME`, `REFL`, `BETA_CONV`, `SUBST`, `ABS`, `INST_TYPE`, `DISCH` and `MP`.

The ML functions that implement the primitive inference schemes in the HOL system are described below. The same notation is used here as in Section 2.3.1: hypotheses above a horizontal line and conclusion beneath. The machine-readable ASCII notation is used for the logical constants.

3.9.1 Assumption introduction

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

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

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

3.9.2 Reflexivity

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

$$\overline{\texttt{|-}\ t\ \texttt{=}\ t}$$

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

3.9.3 Beta-conversion

```
BETA_CONV : term -> thm
```

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

- where $t_1[t_2/x]$ denotes the result of substituting t_2 for x in t_1, with suitable renaming of variables to prevent free variables in t_2 becoming bound after substitution. The substitution $t_1[t_2/x]$ is always defined.

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

3.9.4 Substitution

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

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

- where $t[t_1,\ldots,t_n]$ denotes a term t with some free occurrences of the terms t_1, ..., t_n singled out and $t[t'_1,\ldots,t'_n]$ denotes the result of simultaneously replacing each such occurrences of t_i by t'_i (for $1 \leq i \leq n$), with suitable renaming of variables to prevent free variables in t'_i becoming bound after substitution.

The first argument to `SUBST` is a list `[(|-`t_1`=`t'_1`,` x_1`); ... ;(|-`t_n`=` t'_n`,` x_n`)]`. The second argument is a template term $t[x_1,\ldots,x_n]$ in which occurrences of the variable x_i (where $1 \leq i \leq n$) are used to mark the places where substitutions with `|-` t_i`=`t'_i are to be done. Thus

`SUBST [(|-`t_1`=`t'_1`,` x_1`);...;(|-`t_n`=` t'_n`,` x_n`)]` $t[x_1,\ldots,x_n]$ Γ `|-` $t[t_1,\ldots,t_n]$

returns Γ `|-` $t[t'_1,\ldots,t'_n]$. Failure if:

(i) any of the arguments are of the wrong form;

(ii) the type of x_i is not equal to the type of t_i for some $1 \leq i \leq n$.

3.9.5 Abstraction

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

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

- where x is not free in Γ.

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

3.9.6 Type instantiation

```
INST_TYPE : (type*type) list -> thm -> thm
```

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

- $t[\sigma_1,\;\ldots\;,\sigma_n/\alpha_1,\;\ldots\;,\alpha_n]$ denotes the result of substituting (in parallel) the types $\sigma_1, \ldots, \sigma_n$ for the type variables $\alpha_1, \ldots, \alpha_n$ in t, with the restriction that none of $\alpha_1, \ldots, \alpha_n$ occur in Γ.

`INST_TYPE[(`σ_1`,`α_1`);...;(`σ_n`,`α_n`)]` th returns the result of instantiating each occurrence of α_i in the theorem th to σ_i (for $1 \leq i \leq n$). Failure if:

(i) arguments of the wrong form (e.g. an α_i is not a type variable);

(ii) α_i (for $1 \leq i \leq n$) occurs in any assumption in Γ.

3.9.7 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`.

3.9.8 Modus Ponens

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

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

`MP` takes two theorems (in the order shown above) and returns the result of applying Modus Ponens; it fails if the arguments are not of the right form.

3.10 Oracles

hol98 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, hol98 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 : tag -> term list * term -> thm
```

which directly creates the requested theorem and attaches the given tag to it. Tags may be created with

```
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.

```
- Globals.show_tags := true;                                          1
> val it = () : unit

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

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

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

and prettyprinted by

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

3.11 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 contains the five axioms for higher order logic. These axioms, together with the rules of inference described in Section 3.9, constitute the core of the HOL logic. Because of the way the HOL system evolved from LCF,[6] the particular axiomatization of higher order logic it uses differs from the classical axiomatization due to Church [?]. 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 use 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)))
```

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

[6]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.

There are five axioms in the theory `bool`; the first four are the following:

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

IMP_ANTISYM_AX  |- !t1 t2. (t1 ==> t2) ==> (t2 ==> t1) ==> (t1 = t2)

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

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

The fifth 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 four other axioms of the theory `bool`, the rules of inference in Section 3.9 and the Axiom of Infinity are, together, sufficient for developing all of standard mathematics. Thus, in principle, the user of the HOL system should never need to make a non-definitional theory. In practice, it is often very tempting to take the risk of introducing new axioms because deriving them from definitions can be tedious—proving that 'axioms' follow from definitions amounts to proving their consistency.

The 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))

ARB_DEF      |- ARB    = @x. T
```

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 4.3.

The constant `COND` is used in representing 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 syntax t_1 `=>` t_2 `|` t_3 is also permitted. The system alway prints out conditionals in the "if t_1 then t_2 else t_3" form.

The polymorphic constant ARB is used to denote a fixed but arbitrary element in a type, which is occasionally useful when attempting to deal with the issue of partiality.

A large number of theorems involving the logical constants are pre-proved in the theory bool. The following are only a selection.

```
BOTH_EXISTS_AND_THM  |- !P Q. (?x. P /\ Q) = (?x. P) /\ ?x. Q

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

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

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

COND_ABS        |- !b f g. (\x. (if b then f x else g x)) = if b then f else g

COND_EXPAND     |- !b t1 t2. (if b then t1 else t2) = (~b \/ t1) /\ (b \/ t2)

COND_ID         |- !b t. (if b then t else t) = t

COND_RAND       |- !f b x y. f (if b then x else y) = if b then f x else f y

COND_RATOR      |- !b f g x. (if b then f else g) x = if b then f x else g x

DE_MORGAN_THM        |- !A B. (~(A /\ B) = ~A \/ ~B) /\ (~(A \/ B) = ~A /\ ~B)

ETA_THM              |- !M. (\x. M x) =  M

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

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

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

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

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

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

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

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

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

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

RIGHT_AND_OVER_OR    |- !A B C. (B \/ C) /\ A = B /\ A \/ C /\ A

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

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

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

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

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

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

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

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

SELECT_REFL          |- !x. (@y. y = x) = x

SELECT_UNIQUE        |- !P x. (!y. P y = y = x) ==> ($@ P = x)
```

Chapter 4

Commonly-used Theories

A useful subset of the collection of theories distributed with the HOL system is listed in Table **??**.

In the rest of this section, each of these theories is briefly described. A complete list of all the definitions and theorems in each theory is not given here; the sections that follow provide only an overview of the contents of each theory. For a complete list of all the built-in axioms, definitions and theorems in HOL, see *REFERENCE*.

minTheory	the origin theory
boolTheory	definitions of logical operators and basic axioms
combinTheory	combinators
pairTheory	theory of pairs
sumTheory	disjoint sums
relationTheory	transitive closure and wellfoundedness
numTheory	Peano's axioms derived from the axiom of infinity
prim_recTheory,	the primitive recursion theorem
arithmeticTheory	Peano arithmetic development
numeralTheory	numerals
integerTheory	integers
setTheory	sets as a separate type (includes finite sets)
pred_setTheory	sets as predicates (includes finite sets)
bagTheory	bags (also known as *multisets*)
listTheory	lists
rich_listTheory	extended theory of lists
optionTheory	the `option` type
finite_mapTheory	finite functions
ltreeTheory	polymorphic finitely branching trees
restr_binderTheory	definitions of binder restrictions
res_quanTheory	restricted quantifier support
asciiTheory	ascii
stringTheory	strings
wordTheory	(*plus several others*) theory of bitstrings
realTheory	(*plus several others*) real numbers and analysis

Table 4.1: Commonly-used Theories

4.1 Combinators and the theory `combin`

The theory `combin` contains the definitions of function composition (infixed `o`) and the combinators `S`, `K` and `I`.

```
o_DEF |- !f g. f o g = (\x. f(g x))

K_DEF |- K = (\x y. x)

S_DEF |- S = (\f g x. f x(g x))

I_DEF |- I = S K K
```

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

```
o_THM |- !f g x. (f o g)x = f(g x)

o_ASSOC |- !f g h. f o (g o h) = (f o g) o h

K_THM |- !x y. K x y = x

S_THM |- !f g x. S f g x = f x (g x)

I_THM |- !x. I x = x

I_o_ID |- !f. (I o f = f) /\ (f o I = f)
```

Having the symbols `o`, `S`, `K` and `I` as built-in constants is sometimes inconvenient because they are often wanted as mnemonic names for variables (e.g. `S` to range over sets and `o` to range over outputs). Variables (though not constants) with these names can be used in the current system if `o`, `S`, `K` and `I` are first hidden (see Section 6.4).

4.2 The theory `relation`

Mathematical relations can be represented in HOL by the type `:'a -> 'b -> bool`. The theory `relation` is intended to suppport this view of relations, but does not as yet provide a well-rounded collection of definitions; indeed, it is common to treat relations directly. For example, $R_1 \subseteq R_2$ can be phrased as `!x y. R1 x y ==> R2 x y`. The theory `relation` currently provides definitions and theorems about the transitive closure of a relation and for wellfounded relations.

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

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

Wellfoundedness is used to justify the principle of wellfounded induction and also a general recursion theorem. The statement of the recursion theorem requires that the notion of a function restriction be defined as well.

```
WF_INDUCTION_THM
|- !R:'a->'a->bool.
       WF R
         ==> !P. (!x. (!y. R y x ==> P y) ==> P x)
         ==> !x. P x

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

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

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

The theorems `WF_INDUCTION_THM` and `WFREC_COROLLARY` are used to automate recursive definitions. A few basic combinators for wellfounded relations are also provided in this theory.

```
Empty_def      |- !x y. Empty x y = F

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

WF_Empty       |- WF Empty

WF_SUBSET      |- !R P. WF R /\ (!x y. P x y ==> R x y) ==> WF P

WF_TC          |- !R. WF R ==> WF (TC R)

WF_inv_image   |- !R f. WF R ==> WF (inv_image R f)
```

4.3 Pairs and the type prod

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 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 σ_1#σ_2. It is usual, but not necessary, to write pairs within brackets: (t_1,t_2). The comma symbol associates to the right, so that (t_1,t_2,...,t_n) means (t_1,(t_2,...,t_n)).

Cartesian products are defined by representing a pair (t_1,t_2) by the function

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

The representing type of σ_1#σ_2 is thus σ_1->σ_2->`bool`. To define pairs this way, the constants `MK_PAIR` and `IS_PAIR` are first defined.

```
MK_PAIR_DEF   |- !x y. MK_PAIR x y = (\a b. (a = x) /\ (b = y))

IS_PAIR_DEF   |- !p. IS_PAIR p = (?x y. p = MK_PAIR x y)
```

From these two definitions it is easy to prove that:

```
|- ?p:'a->'b->bool. IS_PAIR p
```

since `|- IS_PAIR(MK_PAIR x y)` follows easily from the definition of `IS_PAIR`. The existence theorem shown above is called `PAIR_EXISTS`. Given this theorem, the type operator `prod` is defined by evaluating:

```
new_type_definition(`prod`, "IS_PAIR:(*->**->bool)->bool", PAIR_EXISTS)
```

which results in the definitional axiom `prod_TY_DEF` shown below being asserted in the theory `bool`.

```
prod_TY_DEF  |- ?rep. TYPE_DEFINITION IS_PAIR rep
```

Next, a new constant `REP_prod` is defined, which maps a pair to its representation as a function:

```
REP_prod    |- REP_prod =
                 (@rep : 'a # 'b -> 'a -> 'b -> bool.
                   (!p' p''. (rep p' = rep p'') ==> (p' = p'')) /\
                   (!p. IS_PAIR p = (?p'. p = rep p')))
```

The infix constructor ',' and the selectors `FST:'a#'b->'a` and `SND:'a#'b->'b` are then defined by the equations shown below.

```
COMMA_DEF  |- !x y. x,y = (@p. REP_prod p = MK_PAIR x y)

FST_DEF    |- !p. FST p = (@x. ?y. MK_PAIR x y = REP_prod p)

SND_DEF    |- !p. SND p = (@y. ?x. MK_PAIR x y = REP_prod p)
```

The following standard theorems about pairs follow easily from these definitions and the axiom `prod_TY_DEF`.

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

FST      |- !x y. FST(x,y) = x

SND      |- !x y. SND(x,y) = y

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

4.3.1 Paired abstractions

The quotation parser will convert[1] `\(`x_1`,`x_2`).`t to `UNCURRY(\`x_1 x_2`.`t`)`, where the constant `UNCURRY` is defined by:

```
UNCURRY f (x,y)  =  f x y
```

The transformation is done recursively so that, for example,

`\(`x_1`,`x_2`,`x_3`).`t

is converted to

`UNCURRY \`x_1`.UNCURRY(\`x_2`,`x_3`.`t`))`

More generally, the quotation parser repeatedly applies the transformation:

`\(`v_1`,`v_2`).`t $\leadsto$ `UNCURRY(\`v_1`.\`v_2`.`t`)`

until no more variable structures remain. For example:

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

[1] Only when the theory of pairs is loaded.

Note that a variable structure like `(x,y)` in `\(x,y).x+y` is not a subterm of the abstraction in which it occurs; it disappears on parsing. This can lead to unexpected errors (accompanied by obscure error messages). For example:

```
- Term `\(x,y).x+y`;                                                    2
> val it = `\(x,y). x + y` : term

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

- Lib.try Term `\^p.x+y`;

Exception raised at Term.dest_var:
not a var
! Uncaught exception:
! HOL_ERR <poly>
```

If b is a binder, then b`(`x_1`,`x_2`).`t is parsed as b`(\(`x_1`,`x_2`).`t`)`, and hence transformed as above. For example, `!(x,y).x>y` parses to `$!(UNCURRY(\x.\y.$> x y))` (where `>` is an infixed constant of the theory `arithmetic` meaning 'is greater than').

Applications of paired abstraction to tuples can be β-reduced using `PAIRED_BETA_CONV` (see Section 9.3.1).

4.3.2 `let`-terms

The quotation parser accepts `let`-terms superficially similar to those in ML. For example, the following terms are allowed:

```
let x = 1 and y = 2 in x+y

let f(x,y) = (x*x)+(y*y) and a = 20*20 and b = 50*49 in f(a,b)
```

`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(`$\ldots$`(LET(LET(\`$v_1 \ldots v_n$`.`t`)`t_1`)`t_2`)`$\ldots$`)`t_n

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

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

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

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

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

The reader is recommended to convince himself or herself that the translations of `let`-terms represent the intuitive meaning suggested by the surface syntax.

`let`-terms can be simplified with `let_CONV` – see Section 9.3.4.

4.4 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.[2] The HOL parser converts "$:\sigma_1+\sigma_2$" into (σ_1,σ_2)`sum`, and the printer inverts this.

The standard operations on sums are:

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

These are all defined as constants in the theory `sum`. The constants `INL` and `INR` inject into the left and right summands, respectively. The constants `ISL` and `ISR` test for membership of the left and right summands, respectively. The constants `OUTL` and `OUTR` project from a sum to the left and right summands, respectively.

The following two theorems, which are minor variants of each other, are pre-proved in the built-in theory `sum`. Each one, on its own, provides a complete and abstract characterization of the disjoint sum type.

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

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

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

Also provided as built-in, are the following theorems having to do with the discriminator functions `ISL` and `ISR`:

```
ISL        |- (!x. ISL(INL x)) /\ (!y. ~ISL(INR y))

ISR        |- (!x. ISR(INR x)) /\ (!y. ~ISR(INL y))

ISL_OR_ISR |- !x. ISL x \/ ISR x
```

The `sum` theory also provides the following built-in theorems:

```
OUTL       |- !x. OUTL(INL x) = x

OUTR       |- !x. OUTR(INR x) = x

INL        |- !x. ISL x ==> (INL(OUTL x) = x)

INR        |- !x. ISR x ==> (INR(OUTR x) = x)
```

which describe the projection functions `OUTL` and `OUTR`.

4.5 The theory `one`

The theory `one` defines the type `one` which contains one element. The constant `one` is specified to denote this element. The pre-proved theorems in the theory `one` are:

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

These three theorems are equivalent characterizations of the type with only one value. The theory `one` is typically used in constructing more elaborate types.

4.6 Natural numbers

The natural numbers are developed in a series of theories. First, the type of numbers is defined from the Axiom of Infinity, and Peano's axioms are derived. Then the primitive recursion theorem is proved. Based on that, a large theory treating the standard arithmetic operations is developed. Lastly, a theory of numerals is provided.

4.6.1 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.

Uses of the theorem INDUCTION are supported by the tactic numLib.INDUCT_TAC (see the documentation in *REFERENCE* for details).

4.6.2 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.[3] For any x and f the *primitive recursion theorem* tells us that there is a unique function fn such that:

$$\texttt{fn 0} = x)\ \texttt{/\\}\ (!n.\texttt{fn(SUC}\,n) = f\ (\texttt{fn}\,n)\ n)$$

The primitive recursion theorem follows from Peano's axioms. When the HOL system is built, the following theorem is proved and stored in the theory prim_rec:

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

The theorem states the validity of primitive recursive definitions on the natural numbers: for any x and f there exists a corresponding total function fn which satisfies the primitive recursive definition whose form is determined by x and f.

[3] 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.

4.6.2.1 Primitive recursive definitions

The primitive recursion theorem can be used to justify any definition of a function on the natural numbers by primitive recursion. For example, a primitive recursive definition in higher order logic of the form

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

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:

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

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 (ignoring the uniqueness of `fn`) the existence theorem shown below:

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

This theorem allows a constant `fun` to be introduced (via the definitional mechanism of constant specifications—see Section 3.7.2.2) to denote the recursive function that satisfies the two equations in the body of the theorem. Introducing a constant `fun` to name the function asserted to exist by the theorem shown above, and simplifying using β-reduction, yields the following theorem:

$$
\begin{array}{ll}
\texttt{|- fun 0} & \texttt{= \textbackslash x}_1 \ \ldots \ \texttt{x}_i\,.\,f_1[\texttt{x}_1,\ldots,\texttt{x}_i]\ \texttt{/\textbackslash} \\
\texttt{\ \ \ fun (SUC n)} & \texttt{= \textbackslash x}_1 \ \ldots \ \texttt{x}_i\,.\,f_2[\texttt{fun n}\ t_1 \ \ldots \ t_i,\ \texttt{n},\ \texttt{x}_1,\ldots,\texttt{x}_i]
\end{array}
$$

It follows immediately from this theorem that the constant `fun` satisfies the primitive recursive defining equations given by the theorem shown below:

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

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 : thm -> string -> term -> thm
```

In fact, `new_recursive_definition` handles primitive recursive definitions over a range of types, not just the natural numbers. For details, see Section 5.7.2, or the *REFERENCE* documentation.

4.6.2.2 The less-than relation

The less-than relation ‘<’ is most naturally defined by primitive recursion. However, it is needed for the proof of the primitive recursion theorem, so it must be defined before definition by primitive recursion is available. The theory `prim_rec` therefore contains the following non-recursive definition of `<`:

```
LESS  |- !m n. m < n = (?P. (!n. P(SUC n) ==> P n) /\ P m /\ ~P n)
```

This definition says that `m < n` if there exists a set (with characteristic function `P`) that is downward closed[4] and contains `m` but not `n`.

4.6.2.3 Consequences of primitive recursion

Once the primitive recursion theorem is available, other useful theorems can be proved. The theory `prim_rec` supplies the Axiom of Dependent Choice, which is a theorem in HOL because it follows from `SELECT_AX`:

```
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 theorem `DC` is useful when one wishes to build a function having a certain property from a relational characterization. For example, an alternate characterization of wellfoundedness is the absence of infinite decreasing R chains. By use of `DC`, this can be proved to be equal to the the notion of wellfoundedness (namely, that every set has an R-minimal element) defined in the theory `relation`.

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

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

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

The theory `prim_rec` also provides theorems asserting the wellfoundedness of the predecessor relation and the less-than relation, as well as the wellfoundedness of measure functions.

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

WF_LESS     |- WF $<

measure_def |- measure = inv_image $<

WF_measure  |- !m. WF (measure m)
```

4.6.3 The theory `arithmetic`

The built-in theory `arithmetic` contains primitive recursive definitions of following standard arithmetic operators.

```
ADD            |- (!n. 0 + n = n) /\
                  (!m n. (SUC m) + n = SUC(m + n))

SUB            |- (!m. 0 - m = 0) /\
                  (!m n. (SUC m) - n = (m < n => 0 | SUC(m - n)))

MULT           |- (!n. 0 * n = 0) /\
                  (!m n. (SUC m) * n = (m * n) + n)

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

It also contains the following non-recursive definitions.

```
GREATER        |- !m n. m > n = n < m

LESS_OR_EQ     |- !m n. m <= n = m < n \/ (m = n)

GREATER_OR_EQ  |- !m n. m >= n = m > n \/ (m = n)

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

An *ad hoc* but useful collection of over a hundred elementary theorems of arithmetic are pre-proved when HOL is built and stored in the theory `arithmetic`. Each theorem will be autoloaded when its name is first mentioned during any HOL session. For a complete list of available theorems, see *REFERENCE*.

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

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

4.6.4 The theory `numeral`

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

NUMERAL_BIT1    !x. NUMERAL_BIT1 n = n + (n + SUC 0)

NUMERAL_BIT2    !x. NUMERAL_BIT2 n = n + (n + SUC(SUC 0))

ALT_ZERO        ALT_ZERO = 0
```

For example, the numeral 5 is represented by the term

```
NUMERAL(NUMERAL_BIT1(NUMERAL_BIT2(ALT_ZERO)))
```

but 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. Numerals may of course be built using `mk_comb`, and taken apart with `dest_comb`. A more convenient interface to this functionality is provided by `mk_numeral`, `dest_numeral`, and `is_numeral`. These functions make use of an ML structure `Arbnum` (written by Michael Norrish) which implements arbitrary precision numbers.

```
- mk_numeral (Arbnum.fromString "3432432423423423234");                     1
> val it = `3432432423423423234` : term

- dest_numeral it;
> val it = <num> : Arbnum.num

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

- mk_numeral it;
> val it = `6864864846846846468` : term
```

Numerals are related to numbers via the derived inference rule `num_CONV`, found in the `numLib` library.

```
   numLib.num_CONV : term -> thm
```

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

```
- load "numLib"; open numLib;                                               1

- num_CONV (Term `1`);
> val it = |- 1 = SUC 0 : thm

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

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

The `num_CONV` function works purely by inference, using the definitions provided above.[5]

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. 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):

[5]In previous versions of HOL, `num_CONV` would not prove its result, which was not in keeping with the LCF approach, and which moreover made `num_CONV` dependent on the underlying implementation of numbers. This made `num_CONV` incomplete on ML systems without arbitrary-sized numbers.

```
- load "integerTheory";                                              2
> val it = () : unit

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

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

- Term`2n`;
> val it = `2` : term

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

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

A numeric literal 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 4.7 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_info` entry in *REFERENCE*.

4.7 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	right
`int_le`	`<=`	450	right
`int_gt`	`>`	450	right
`int_lt`	`<`	450	right
`int_add`	`+`	500	left
`int_sub`	`-`	500	left
`int_neg`	`~`	900	trueprefix
`int_mul`	`*`	600	left
`/`		600	left
`%`		650	left
`int_exp`	`**`	700	right
`int_of_num`	`&`		prefix

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

```
Term ‘1i = &(1n + 0n)‘;                                                    3
> val it = ‘1 = & (1 + 0)‘ : term

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

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

4.8 Real numbers and analysis

There is an extensive collection of theories that make up the development of real numbers and analysis in HOL, due to John Harrison [**?**]. We will only give an 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	right
`real_lte`	`<=`	450	right
`real_gt`	`>`	450	right
`real_lt`	`<`	450	right
`real_add`	`+`	500	left
`real_sub`	`-`	500	left
`real_neg`	`~`	900	trueprefix
`real_mul`	`*`	600	left
`real_div`	`/`	600	left
`pow`		700	right
`real_of_num`	`&`		prefix

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

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

netsTheory Moore-Smith covergence nets, and special cases like sequences.

seqTheory Sequences and series of real numbers.

limTheory Limits, continuity and differentiation.

powserTheory Power series.

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

A separate development that depends only on `realTheory` is a theory of polynomials, found in `polyTheory`. A standard collection of operations on polynomials, and theorems about them, are also derived.

4.9 The theory `list`

The theory `list` introduces the unary type operator `list` by a type definition.[6] The standard list processing functions are then defined on this type:

```
NIL  : 'a list
CONS : 'a -> 'a list -> 'a list
HD   : 'a list -> 'a
TL   : 'a list -> 'a list
NULL : 'a list -> bool
```

[6]For details of the definition, see [**?**, **?**].

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`;...;`t_n`]` into `CONS` t_1 `(CONS` $t_2 \cdots$ `(CONS` t_n `NIL)` $\cdots$ `)`. The HOL printer reverses these transformations.

The functions `NIL` and `CONS` are defined in terms of the representing type of lists. From their definitions, the following fundamental theorems about lists are proved and stored in the theory `list`.

```
list_Axiom    |- !x f. ?!fn.(fn[] = x) /\ (!h t. fn(h::t) = f(fn t)h t)

list_INDUCT   |- !P. P[] /\ (!t. P t ==> (!h. P(h::t))) ==> (!l. P l)

list_CASES    |- !l. (l = []) \/ (?t h. l = h::t)

CONS_11       |- !h t h' t'. (h::t = h'::t') = (h = h') /\ (t = t')

NOT_NIL_CONS  |- !h t. ~([] = h::t)

NOT_CONS_NIL  |- !h t. ~(h::t = [])
```

The theorem `list_Axiom` shown above is analogous to the primitive recursion theorem on the natural numbers discussed above in Section 4.6.2.1. 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. For example, the HOL system defines a length function, `LENGTH`, on lists by the primitive recursive definition on lists shown below:

```
new_recursive_definition Prefix list_Axiom "LENGTH"
   (Term `(LENGTH NIL = 0) /\
          (!h t. LENGTH (h::t) = SUC (LENGTH t))`)
```

When this ML expression is evaluated, HOL uses `list_Axiom` to prove existence of a function that satisfies the given primitive recursive definition, introduces a constant to name this function using a constant specification, and stores the resulting theorem:

```
LENGTH   |- (LENGTH [] = 0) /\ (!h t. LENGTH(h::t) = SUC(LENGTH t))
```

in the current theory (in this case, the theory `list`).

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

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

HD   |- !h t. HD(h::t) = h

TL   |- !h t. TL(h::t) = t
```

The following primitive recursive definitions of functions on lists are also made in the theory `list`:

```
SUM        |- (SUM [] = 0) /\ (!h t. SUM(h::t) = h + SUM t)

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

FLAT       |- (FLAT[] = []) /\ (!h t. FLAT(h::t) = APPEND h (FLAT t))

LENGTH     |- (LENGTH [] = 0) /\ (!h t. LENGTH (h::t) = SUC(LENGTH t))

MAP        |- (!f. MAP f [] = []) /\
              (!f h t. MAP f (h::t) = f h::MAP f t)

EL         |- (!l. EL 0 l = HD l) /\ (!l n. EL (SUC n)l = EL n (TL l))

EVERY_DEF  |- (!P. EVERY P [] = T) /\
              (!P h t. EVERY P (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)

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))

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

For a complete list of available theorems in `listTheory`, see *REFERENCE*. The theory `list` is relatively compact, largely because of how the HOL system has evolved. A more extensive theory of lists can be found in `rich_listTheory`.

4.10 Trees

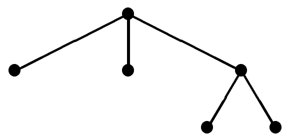
The theories `tree` and `ltree` contain the definitions of two structurally-isomorphic types of finitely-branching ordered trees. The types defined in these theories are used by Tom Melham's type definition package (see Section 5.7) to construct representations for arbitrary concrete recursive types. The following is a summary of the main theorems which are available in the theories `tree` and `ltree`, and which may be of use in certain specialized applications. For full details of the logical basis for these two theories, see [**?**].

4.10.1 The theory `tree`

In the theory `tree`, a type `tree` is defined to denote the set of all ordered trees whose nodes can branch any (finite) number of times. A constructor function

```
node : tree list -> tree
```

is then defined in the theory `tree`. This function can be used to construct any tree-structured value of type `tree`. The expression `"node []"` denotes the tree consisting of a single leaf node with no subtrees. If *tl*`:(tree)list` is a non-empty list of trees, then the term `"node` *tl*`"` denotes the tree whose immediate subtrees are the trees in the list *tl*. Using `node`, it is possible to construct a tree of any shape. For example, the tree



is denoted by the term `"node[node[]; node[]; node[node[]; node[]]"`.

The next two theorems follow from the formal definition of `node` and are stored in the theory `tree`:

```
node_11      |- !tl1 tl2. (node tl1 = node tl2) = (tl1 = tl2)
tree_Induct  |- !P. (!tl. EVERY P tl ==> P(node tl)) ==> (!t. P t)
```

These two theorems are analogous to the Peano postulates for the natural numbers, and are used to prove the following abstract characterization of the defined type `tree`.

```
tree_Axiom  |- !f. ?! fn. !tl. fn(node tl) = f(MAP fn tl)tl
```

This theorem states the validity of general 'primitive recursive' definitions of functions over finitely-branching ordered trees.

4.10.2 The theory `ltree`

In the theory `ltree`, a type of *labelled* trees (called `'a ltree`) is defined. Labelled trees have the same sort of structure as values of the defined type `tree` discussed above. The only difference is that a tree of type `'a ltree` has a value or 'label' of type `'a` associated with each of its nodes. A constructor

```
Node : 'a -> 'a ltree list -> 'a ltree
```

is defined in the theory `ltree`. The function `Node` constructs labelled trees by mapping a label of type `'a` and a list of labelled subtrees to a labelled tree of type `'a ltree`. The following theorems about labelled trees are pre-proved and stored in the theory `ltree`.

```
Node_11       |- !v1 v2 trl1 trl2.
                 (Node v1 trl1 = Node v2 trl2) = (v1 = v2) /\ (trl1 = trl2)
ltree_Induct  |- !P. (!t. EVERY P t ==> (!h. P(Node h t))) ==> (!l. P l)
ltree_Axiom   |- !f. ?! fn. !v tl. fn(Node v tl) = f(MAP fn tl)v tl
```

These theorems are analogous to their counterparts in the theory `tree` discussed above. The theorems `Node_11` and `ltree_Induct` amount to a Peano-type characterization of labelled trees, and the theorem `ltree_Axiom` is a primitive recursion theorem for labelled trees.

Chapter 5

Commonly-used Libraries

5.1 A simple proof manager

The *goal stack* provides a simple interface to tactic-based proof. When one uses tactics to decompose a proof, many intermediate states arise; the goalstack takes care of the necessary bookeeping. 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 operations so that the focus can be changed.

5.1.1 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 the goal to have any assumptions.

Calling `set_goal`, or `g`, adds a new proof attempt to the existing ones, *i.e.*, rather than overwriting the current proof attempt, the new attempt is stacked on top.

5.1.2 Applying a tactic to a goal

```
expandf : tactic -> goalstack
expand  : tactic -> goalstack
e       : tactic -> goalstack
```

How does one actually do a goalstack proof then? In most cases, the application of tactics to the current goal is done with the function `expand`. In the rare case that one wants to apply an *invalid* tactic, then `expandf` is used. (For an explanation of invalid tactics, see Chapter 24 of Gordon & Melham.) The abbreviation `e` may also be used to expand a tactic.

5.1.3 Undo

```
b          : unit -> goalstack
drop       : unit -> proofs
dropn      : int  -> proofs
backup     : unit -> goalstack
restart    : unit -> goalstack
set_backup : int  -> unit
```

Often (we are tempted to say *usually*!) one takes a wrong path in doing a proof, or makes a mistake when setting a goal. To undo a step in the goalstack, the function `backup` and its abbreviation `b` are used. This will restore the goalstack to its previous state.

To directly back up all the way to the original goal, the function `restart` may be used. Obviously, it is also important to get rid of proof attempts that are wrong; for that there is `drop`, which gets rid of the current proof attempt, and `dropn`, which eliminates the top n proof attempts.

Each proof attempt has its own *undo-list* of previous states. The undo-list for each attempt is of fixed size (initially 12). If you wish to set this value for the current proof attempt, the function `set_backup` can be used. If the size of the backup list is set to be smaller than it currently is, the undo list will be immediately truncated. You can not undo a "proofs-level" operation, such as `set_goal` or `drop`.

5.1.4 Viewing the state of the proof manager

```
p            : unit -> goalstack
status       : unit -> proofs
top_goal     : unit -> goal
top_goals    : unit -> goal list
initial_goal : unit -> goal
top_thm      : unit -> thm
```

To view the state of the proof manager at any time, the functions `p` and `status` can be used. The former only shows the top subgoals in the current goalstack, while the second gives a summary of every proof attempt.

To get the top goal or goals of a proof attempt, use `top_goal` and `top_goals`. To get the original goal of a proof attempt, use `initial_goal`.

Once a theorem has been proved, the goalstack that was used to derive it still exists (including its undo-list): its main job now is to hold the theorem. This theorem can be retrieved with `top_thm`.

5.1.5 Switch focus to a different subgoal or proof attempt

```
r             : int -> goalstack
R             : int -> proofs
rotate        : int -> goalstack
rotate_proofs : int -> proofs
```

Often we want to switch our attention to a different goal in the current proof, or a different proof. The functions that do this are `rotate` and `rotate_proofs`, respectively. The abbreviations `r` and `R` are simpler to type in.

5.2 The boss library

The library `bossLib` marshalls 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; heigh-level interactive proof operations, and composition of automated reasoners. Loading `bossLib` commits one to working in a context that already supplies the theories of booleans, pairs, sums, the option type, arithmetic, and lists.

5.2.1 Datatype definition

There are several useful consequences of an object logic datatype definition: structural induction, rewrite rules for constructors, etc. However, these have not traditionally been automatically derived at the invocation of the definition package: the user would have to build the required theorems by explicitly invoking various proof procedures. To remedy this, `bossLib` offers the `Hol_datatype` function. This function allows for the definition of mutually recursive types, nested recursive types and record types. The syntax of declarations that `Hol_datatype` accepts is found in Table 5.1.

There is an underlying database of datatype facts that supports the activities of `bossLib`. This database already contains the relevant entries for the types `bool`, `prod`, `num`, `option`, and `list`. When a datatype is defined by `Hol_datatype`, the following information is derived and stored in the database.

- initiality theorem for the type
- injectivity of the constructors

```
Hol_datatype `[binding ;]* binding`

         binding  ::=  ident = constructor-spec
                    |  ident = record-spec

constructor-spec  ::=  [clause |]* clause

          clause  ::=  ident
                    |  ident of [hol_type =>]* hol_type

     record-spec  ::=  <| [ident : hol_type ;]* ident : hol_type |>
```

Table 5.1: Datatype Declaration

- 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

5.2.2 Support for high-level proof steps

The following functions use information in the database to ease the application of hol98's underlying functionality:

```
type_rws     : string -> 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 by name in the underlying 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 pre-existing rewrite rules in the database are already integrated into the simplification sets provided by `bossLib`; however rewrite rules arising from an invocation of `Hol_datatype`, or which come from a user-defined theory, will have to be manually added into the simpsets used by the simplifier.

- 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.

Two supplementary entrypoints 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 [1] stronger induction hypothesis than ordinary mathematical induction: to wit, when inducting on n, one is allowed to assume the property holds for *all* m smaller than n. Formally: $\forall P.\ (\forall x.\ (\forall y.\ y < x \supset P\,y) \supset P\,x) \supset \forall x.\ P\,x$. This allows the inductive hypothesis to be used more than once, and also allows instantiating the inductive hypothesis to other than the predecessor.

measureInduct_on takes a quotation, and breaks it apart to find a term and a measure function with which to induct. For example, if one wanted to induct on the length of a list `L`, the invocation `measureInduct_on ‘LENGTH L‘` would be be appropriate.

[1] Complete induction and ordinary mathematical induction are each derivable from the other.

5.2.3 Function definition

```
Define   : term quotation -> thm
xDefine  : string -> term quotation -> thm
Hol_defn : string -> term quotation -> Defn.defn
```

The Define function is a general-purpose function definition mechanism. The xDefine function is identical to Define except that it takes an explicit name to use when storing the definition in the current theory. Define accepts the following syntax:

1. Non-recursive definition, varstructs allowed on lhs.

```
Define ‘f w (x, y, z) = x + y / w + z‘;
```

2. Primitive recursive (or non-recursive) over known datatype.

```
Define
    ‘(fold b f [] = b) /\
     (fold b f (h::t) = f h (fold b f t))‘;
```

3. Non-recursive definition, over complex patterns:

```
Define
    ‘(g (0,x,y,z) = 1)
  /\ (g (w,0,y,z) = 2)
  /\ (g (w,x,0,z) = 3)
  /\ (g (w,x,y,0) = 4)‘;
```

4. Recursions (not mutual or nested) that aren’t handled by 2.

```
Define ‘(flatten  []            = [])
   /\   (flatten ([]::rst)      = flatten rst)
   /\   (flatten ((h::t)::rst) = h::flatten(t::rst))‘;
```

5. Nested recursions.

```
Define ‘N x = if x>100 then x-10 else N(N(x+11))‘;
```

6. Mutual recursion.

```
xDefine "even_odd"
   `(even 0 = T)
/\  (even (SUC n) = odd n)
/\  (odd 0 = F)
/\  (odd (SUC n) = even n)`;
```

7. Schematic definitions (mutual and nested recursive schemata are accepted).

```
Define `While s = if B s then While (C s) else s`;
```

For complex recursions, `Define` attempts to find a measure under which recursive calls become smaller (and to prove that they do indeed become smaller). Currently, it examines the domain type of the function being defined and synthesizes a "size" measure. Then it does some basic simplifications and then attempts to automatically prove the termination constraints. If this termination proof fails, then the definition attempt fails. If the termination proof succeeds, an induction theorem for the function is also automatically derived and stored in the current theory.
Example. Invoking

```
  Define
    `(gcd 0 y = y)
/\   (gcd (SUC x) 0 = SUC x)
/\   (gcd (SUC x) (SUC y) =
         if y <= x then gcd (x-y)   (SUC y)
                   else gcd (SUC x) (y-x))`;
```

proves all termination conditions and stores the theorem

```
|- (gcd 0 y = y)           /\
   (gcd (SUC x) 0 = SUC x) /\
   (gcd (SUC x) (SUC y) =
        if y <= x then gcd (x - y) (SUC y)
                  else gcd (SUC x) (y - x))
```

in the current theory under the name `"gcd_def"` and also stores the theorem

```
!P. (!y. P 0 y)       /\
    (!x. P (SUC x) 0) /\
    (!x y. (~(y <= x) ==> P (SUC x) (y - x)) /\
           (y <= x  ==> P (x - y) (SUC y))
            ==> P (SUC x) (SUC y))
      ==>
        !v v1. P v v1.
```

in the current theory under the name "gcd_ind" before returning the requested recursion equations.

Recall that, if the termination proof fails, an invocation of Define (or xDefine) fails. In such situations, the ML function Hol_defn should be used.

```
Hol_defn     : string -> term quotation -> Defn.defn
WF_REL_TAC   : Defn.defn -> term quotation -> tactic
```

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.

Example. An invocation of Define on the following equations for Quicksort will currently fail, since the termination proof is beyond the capabilities of our naive termination prover. Instead, we make an application of Hol_defn:

```
val qsort_def =
  Hol_defn "qsort"
    `(qsort r [] = []) /\
     (qsort r (h::t) =
         APPEND (qsort r (FILTER (\x. r x h) t))
           (h :: qsort r (FILTER (\x. ~(r x h)) t)))`;
```

which returns a defn, but does not try to prove termination. Although it is possible to directly work with elements of type defn, it is more convenient to invoke 'Defn.tgoal qsort_def', which sets up a termination proof in a goalstack. The goal is just to get the unrestricted recursion equations and induction theorem.

```
Defn.tgoal qsort_def;

> val it =
>    Proof manager status: 1 proof.
>    1. Incomplete:
>         Initial goal:
>         ((qsort r [] = []) /\
>          (qsort r (h::t) =
>           APPEND (qsort r (FILTER (\x. r x h) t))
>             (h::qsort r (FILTER (\x. ~r x h) t)))) /\
>         !P.
```

```
>           (!r. P r []) /\
>           (!r h t. P r (FILTER (\x. r x h) t) /\
>                    P r (FILTER (\x. ~r x h) t) ==> P r (h::t))
>           ==> !v v1. P v v1
```

How to proceed? The function `WF_REL_TAC` now shows its utility. When given a `defn` and a quotation denoting a termination relation for the function, `WF_REL_TAC` initiates the termination proof. For our example, we obtain two subgoals both of which are easy to prove.

```
    - e (WF_REL_TAC qsort_def `measure (LENGTH o SND)`
>   OK..
>
>   2 subgoals:
>   val it =
>      !t h r. LENGTH (FILTER (\x. r x h) t) < LENGTH (h::t)
>
>
>      !t h r. LENGTH (FILTER (\x. ~r x h) t) < LENGTH (h::t)
```

Both goals are provable; once the proof is completed, we can encapsulate it with `Defn.tprove`, which takes a `defn`, builds a termination goal from it, applies the given tactic, and, if the initial goal is proved, returns a pair comprising the requested equations and the induction theorem.

```
   val (qsort_eqns,qsort_ind) =
      Defn.tprove
       (qsort_def,
        WF_REL_TAC qsort_def `measure (LENGTH o SND)`
        THEN ...);

   > val qsort_eqns =
   >  |- (qsort r [] = []) /\
   >     (qsort r (h::t) =
   >        APPEND (qsort r (FILTER (\x. r x h) t))
   >           (h::qsort r (FILTER (\x. ~r x h) t)))  : thm

   >  val qsort_ind =
   >   |- !P.
   >      (!r. P r []) /\
   >      (!r h t. P r (FILTER (\x. r x h) t) /\
   >               P r (FILTER (\x. ~r x h) t) ==> P r (h::t))
   >      ==> !v v1. P v v1
```

5.2.4 Automated reasoners

`bossLib` brings together the most powerful reasoners in hol98 and tries to make it easy to compose them in a simple way. We take our basic reasoners from `mesonLib`, `simpLib`, and `decisionLib`, but the point of `bossLib` is to provide a layer of abstraction so the user has to know only a few entrypoints.[2]

```
PROVE      : thm list -> term quotation -> thm
PROVE_TAC  : thm list -> tactic

DECIDE     : term quotation -> thm
DECIDE_TAC : tactic
```

The inference rule `PROVE` (and the corresponding tactic `PROVE_TAC`) takes a list of theorems and a quotation, and attempts to prove the term using a first order reasoner. 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
&&       : simpset * thm list -> simpset  (* infix *)
std_ss   : simpset
arith_ss : simpset
list_ss  : simpset
```

The rewriting tactic `RW_TAC` works by first adding the given theorems into the given `simpset`; then it simplifies the goal as much as possible; then it performs case splits on any conditional expressions in the goal; then it repeatedly (1) eliminates all hypotheses of the form $v = M$ or $M = v$ where v is a variable not occurring in M, (2) breaks down any equations between constructor terms occurring anywhere in the goal. The infix combinator `&&` is used to build a new simpset from a given simpset and a list of theorems.

Simplification sets for its native datatypes are provided by `bossLib`. In general, these are extended versions of those found in `simpLib`. 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`.

```
STP_TAC  : simpset -> tactic -> tactic
ZAP_TAC  : simpset -> thm list -> tactic
```

[2] In the mid 1980's Graham Birtwistle advocated such an approach, calling it 'Ten Tactic HOL'.

The compound reasoners of bossLib take a basic approach: they simplify the goal as much as possible with RW_TAC and then a 'finishing' tactic is applied. The primitive entrypoint for this is STP_TAC. Currently, the most powerful reasoner is ZAP_TAC, which features a finishing tactic that first tries a tautology checking tactic; if that fails, DECIDE_TAC is called; if that fails, PROVE_TAC is called with the second argument. Although this general approach (simplify as much as possible, then apply automated reasoners in sequence) is crude, we have found that it allows one to make good progress in a high percentage of proof situations.

```
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'[3]

SPOSE_NOT_THEN 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.

Note. When the library bossLib is loaded, the infix parsing status of && and "by" must be re-asserted by the user.

5.3 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 (cross-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 hol98 are similar to C's struct types and to Pascal's records. However, the current HOL implementation doesn't allow the equivalent of variant records, nor for records to be recursive.

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

[3]Proofs in the Mizar system are readable documents, unlike almost all tactic-based proofs.

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.

5.3.1 Defining a record type

Defining a record type is achieved with the function `Hol_datatype`, as previously discussed. For example, to create a record type called `person` with boolean, string and number fields called `employed`, `name` and `age`, one would enter:

```
val _ = Hol_datatype `person = <| employed : bool ; age : num ;
                                   name : string
                                |>`;
```

The order in which the fields are entered is not significant. As well as defining the type (called `person`), the datatype definition function also defines three other sets of constants. These are the field access functions, update 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 update functions are given the names "⟨*record-type*⟩_⟨*field*⟩`_update`" for each field in the type. They take a value of the type of the field in question and a record value to be modified. They return a new record value that is otherwise the same as the old value but with the specified field having the new value. They can be written with the keyword `with` and the `:=` operator:

```
``bob with employed := T``
```

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" |>``
```

Finally, the second sort of update functions, the so-called "functional" updates have names of the form "⟨*record-type*⟩_⟨*field*⟩`_fupd`". Rather than specifying a new value for the record, these functions take a function as their first parameter, which will be 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. The functional update functions allow more concision when writing updates on a record that depend on the field's old value.

The special syntax for writing these updates is to again use the `with` keyword, but to use the infix `updated_by` rather than `:=`. Thus

```
``bob with employed updated_by $~``
```

is a record value with the opposite boolean value in the `employed` field as held by `bob`.

5.3.2 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" |>``
```

5.3.3 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. Most of these are returned in a big record; all are stored using `save_thm` so that they can be recovered.

The record returned has the following fields:

`type_axiom` The type axiom for the record type, as returned by the standard datatype definition package.

`accessor_fns` The definitions of the accessor functions. This theorem should be included in rewrites used for this type.

`update_fns` The definitions of the update functions. This theorem should be included in rewrites used for this type.

`cases_thm` The usual cases theorem for a type, stating that for all record values, there exist component values making it up.

`fn_upd_thm` The definitions of the functional update functions. This theorem should be included in rewrites used for this type.

`acc_upd_thm` A theorem stating simpler forms for expressions of the form *field*$_i$ (*field*$_j$`_update` $v\ r$). If $i = j$, then the RHS is v, if not, it is (*field*$_i$ r). This theorem should be included in rewrites used for this type.

`upd_acc_thm` A theorem stating that *field*$_i$`_update` (*field*$_i$ r) $r = r$ for all of the fields defined in the type. This theorem should be included in rewrites used for this type.

`upd_upd_thm` A thereom stating that *field*$_i$`_update` v_1 (*field*$_i$`_update` v_2 r) = *field*$_i$`_update` v_1 r. This theorem should be included in rewrites used for this type.

`upd_canon_thm` 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. This theorem should be included in rewrites used for this type.

`cons_11_thm` The standard result stating the type constructor is injective. This theorem should be included in rewrites used for this type.

`create_term` This last component of the record returned is not a theorem, but rather an ML function. It is identical to the `create_term_fn` already defined in `RecordType`, but is pre-applied to the relevant arguments, so that it is of the type string-value list to term.

5.4 The `meson` library

5.5 The `simp` library

5.6 The `num` library

5.7 The type definition package

> *All of this section of the documentation is out of date. Users of hol98 should use `Hol_datatype` to define types, and `Define` or `xDefine` to define functions over them. The functions described here do exist in hol98 but generally with different types. This section was written for HOL88 and has not been updated since.*

In the HOL system, new types and type operators can be introduced using the consistency-preserving definitional mechanism of type definitions (see Sections 2.5.4 and 3.7.2.3). The ML rule for introducing a new type is:

```
new_type_definition : (string # term # thm) -> thm
```

This rule allows axioms of a restricted form to be added to the primitive basis of the logic. These axioms are analogous to definitional axioms for new constants: they define new types in terms of other type expressions already present in the logic. Like the rule `new_definition` for making constant definitions, the rule `new_type_definition` for type definitions ensures that adding a new syntactic entity (in this case, a type or type operator) is a conservative extension of the logic.

The basic idea behind `new_type_definition` is that a type definition is made by adding an axiom to the logic which asserts that the set of values denoted by a new type is isomorphic to an appropriate subset of the values denoted by a type expression already present in the logic. A definitional axiom of this form merely states that a new type is isomorphic to a particular subset of an existing type. From such type definition axioms, it is usual to prove theorems that characterize newly-defined types more abstractly. The idea is to prove a collection of theorems that state the essential properties of a new type without reference to how it is defined. These theorems then constitute a derived 'abstract axiomatization' of the new type, and once they have been proved they become the basis for all further reasoning about it.

With this approach, introducing a new type (or type operator) in HOL involves two distinct steps:

1. Finding an appropriate representation for the new type, and making a type definition using `new_type_definition` based on this representation.

2. Using the axiomatic definition of the new type and the properties of its representation to prove a set of theorems that abstractly characterizes it.

Defining a new type using this approach can be hard work. But a set of tools is provided in the system which—for a certain class of commonly-used *concrete recursive types*—automatically carries out all the formal proofs necessary to define these types and derive abstract characterizations from their definitions. This section provides a user-level overview of these tools. Details of the formal proofs carried out by these tools are discussed in [?].

5.7.1 Defining types

The main ML function in the HOL type definition package is

```
define_type : string -> string -> thm
```

This function can be used to define any concrete recursive type in the HOL system. These are types whose values are generated by a set of *constructors* (i.e. functions) which yield concrete representations for these values. Examples include types which denote finite sets of atomic values (enumerated types), types which denote sets of structured values (record types) or finite disjoint unions of structured values (variant records), and types which denote sets of recursive data structures (recursive types).

The two inputs to `define_type` are both strings. The first string is a name under which the results of making the type definition will be stored in the current theory segment. The second is a user-supplied informal[4] specification of the concrete recursive type to be defined. This type specification is written in a notation (explained below) which resembles a data type declaration in functional programming languages like Standard ML [?]. It simply states the names of the new type's constructors and the logical types of their arguments. The output is a theorem which abstractly characterizes the properties of the desired recursive type—i.e. a derived 'abstract axiomatization' of the type.

5.7.1.1 Input syntax

The type specification given as input to `define_type` must be an ML string (of ML type `string`) of the form:

$$`\mathit{op} = C_1\, ty_1^1 \, \ldots \, ty_1^{k_1} \quad | \ \cdots \ | \ C_m\, ty_m^1 \, \ldots \, ty_m^{k_m}`$$

where each ty_i^j is either a type expression already defined as a type in the current theory (this type expression must not contain *op*) or is the name *op* itself. A string of this form describes an n-ary type operator *op*, where n is the number of distinct type variables in the types ty_i^j on the right hand side of the equation. If $n = 0$ then *op* is a type constant; otherwise *op* is an n-ary type operator. The concrete type described has m distinct constructors $C_1, \ldots, C_m$ where $m \geq 1$. Each constructor C_i takes k_i arguments, where $k_i \geq 0$; and the types of these arguments are given by the type expressions ty_i^j for $1 \leq j \leq k_i$. If one or more of the type expressions ty_i^j is the type *op* itself, then

[4]In this context, *informal* means not in the language of higher order logic.

the equation specifies a *recursive* type. In any specification of a recursive type, at least one constructor must be non-recursive—i.e. all its arguments must have types which already exist in the current theory.

The input parser for `define_type` treats type expressions exactly as the HOL quotation parser does, with precedences among the various built-in type operators in force.

5.7.1.2 The type specified

The logical type described by an input string of the form shown above is intended to denote the set of all values which can be finitely generated using the constructors $C_1, \ldots, C_m$, where each constructor is one-to-one and any two different constructors yield different values. Every value of this type will be denoted by some term of the form:

$$C_i\ x_i^1\ \ldots\ x_i^{k_i}$$

where x_i^j is a term of type ty_i^j for $1 \leq j \leq k_i$. In addition, any two terms:

$$C_i\ x_i^1\ \ldots\ x_i^{k_i} \qquad \text{and} \qquad C_j\ x_j^1\ \ldots\ x_j^{k_j}$$

denote equal values exactly when their constructors are the same (i.e. $i = j$) and these constructors are applied to equal arguments (i.e. $x_i^n = x_j^n$ for $1 \leq n \leq k_i$).

5.7.1.3 The output

For any type specification in the form of an equation of the kind discussed above, executing:

$$\texttt{define\_type}\ \text{`}name\text{`}\ \text{`}\mathit{op}\ \texttt{=}\ C_1\ ty_1^1\ \ldots\ ty_1^{k_1}\ \ \texttt{|}\ \cdots\ \texttt{|}\ C_m\ ty_m^1\ \ldots\ ty_m^{k_m}\text{`}$$

will make a formal definition for a type (or type operator) *op* in the current theory segment, make appropriate definitions for constants $C_1, C_2, \ldots, C_m$, and automatically prove a theorem which provides an abstract characterization of the newly-defined type *op*. This theorem, which is stored in the current theory segment under the name $name$ and also returned by `define_type`, has the form shown below:

$$\begin{array}{l}
\texttt{|-}\ \texttt{!f}_1 \cdots \texttt{f}_m\texttt{.}\ \texttt{?!fn:}\mathit{op}\texttt{->*.} \\
\qquad \texttt{!x}_1^1 \cdots \texttt{x}_1^{k_1}\texttt{.}\ \texttt{fn(}C_1\ \texttt{x}_1^1 \ldots \texttt{x}_1^{k_1}\texttt{)} = \texttt{f}_1\ \texttt{(fn x}_1^1\texttt{)} \ldots \texttt{(fn x}_1^{k_1}\texttt{)}\ \texttt{x}_1^1 \ldots \texttt{x}_1^{k_1} \\
\qquad \qquad \vdots \\
\qquad \texttt{!x}_m^1 \cdots \texttt{x}_1^{k_m}\texttt{.}\ \texttt{fn(}C_m\ \texttt{x}_m^1 \ldots \texttt{x}_m^{k_m}\texttt{)} = \texttt{f}_m\ \texttt{(fn x}_m^1\texttt{)} \ldots \texttt{(fn x}_m^{k_m}\texttt{)}\ \texttt{x}_m^1 \ldots \texttt{x}_m^{k_m}
\end{array}$$

where the right hand sides of the equations include recursive applications '`fn` x_i^j' only for variables x_i^j of type *op*. (See the examples given below.) A theorem of this form asserts the unique existence of primitive recursive functions defined by cases on the constructors $C_1, C_2, \ldots, C_m$. This is a slight extension of the *initiality* property by which structures of this kind are characterized in the 'initial algebra' approach to specifying abstract data types [**?**]. This property provides an abstract characterization of the type *op* which is both succinct and complete, in the sense that it completely determines the structure of the values of *op* up to isomorphism.

The call to `define_type` shown above fails if:

(i) not in draft mode;

(ii) *op* is already the name of a type constant or type operator in the current theory;

(iii) any one of $C_1, \ldots, C_m$ is already the name of a constant in the current theory.

(iv) either *op* or any one $C_1, \ldots, C_m$ is not a legal identifier. Identifiers must start with a letter (as defined by `is_letter`) and contain only alphanumeric characters (as defined by `is_alphanum`)

(v) `ABS_`op or `REP_`op are already constants in the current theory;

(vi) there is already an axiom, definition, constant specification or type definition stored under either the name op`_TY_DEF` or the name op`_ISO_DEF` in the current theory segment.

(vii) there is already a theorem stored under the name `'`$name$`'` in the current theory segment.

(viii) the input type specification does not conform to the syntax described above.

5.7.1.4 Examples

The session that follows illustrates the use of `define_type` in defining a variety of simple concrete types. It is assumed that the session begins with the user in draft mode.

The first definition is simple, the definition of a type `three` with exactly three distinct values: `ONE`, `TWO`, and `THREE`.

```
#let three_Axiom = define_type `three_Axiom` `three = ONE | TWO | THREE`;;    1
three_Axiom =
|- !e0 e1 e2. ?! fn. (fn ONE = e0) /\ (fn TWO = e1) /\ (fn THREE = e2)
```

The theorem returned by `define_type` provides a complete and abstract characterization of a defined logical type `three` which denotes a set of exactly three elements. This characterization takes the form of a degenerate 'primitive recursion' theorem for the concrete type `three`. Since `three` is an enumerated type with no recursive constructors, the theorem returned by `define_type` simply states that any function defined by cases on the three constants `ONE`, `TWO`, and `THREE` exists and is uniquely defined.

It follows immediately from this theorem that the type constant `three` denotes a set containing exactly three values: the fact that the function `fn` always exists implies that the constants `ONE`, `TWO`, and `THREE` denote distinct values of type `three`, and the fact that `fn` is uniquely determined by its values for `ONE`, `TWO`, and `THREE` implies that these constants denote the only values of type `three`.

The next call to `define_type` defines a 'record type' `rec`, values of which are records with three boolean fields (essentially 3-tuples):

```
#let rec_Axiom = define_type `rec_Axiom` `rec = REC bool bool bool`;;     2
rec_Axiom = |- !f. ?! fn. !b0 b1 b2. fn(REC b0 b1 b2) = f b0 b1 b2
```

Here, the resulting theorem states that a function `fn` on record values of type `rec` can be uniquely defined in terms of a function `f` of the three components of the record.

A more interesting *recursive* example is the type of natural numbers , which can be defined using `define_type` as follows:

```
#let nat_Axiom = define_type `nat_Axiom` `nat = Z | Suc nat`;;     3
nat_Axiom = |- !e f. ?! fn. (fn Z = e) /\ (!n. fn(Suc n) = f(fn n)n)
```

Here, the input string describes a type `nat` with two constructors: `Z`, which stands for zero; and `Suc`, which is the successor function on natural numbers. (The names `Z`, and `Suc` are used here because `0` and `SUC` are already constants in the built-in HOL theory `num`.) The output theorem is just the primitive recursion theorem[5] for the natural numbers; it states that any primitive recursive definition on the natural numbers (i.e. on values of type `nat`) uniquely defines a total function.

A recursive type of labelled binary trees, where labels of type `*` appear only on leaf nodes, can likewise be defined using `define_type`. The input states that a binary tree is either a leaf node (`LEAF`) labelled by a value of type `*` or an internal node `NODE` with two binary trees as subtrees:

```
#let btree_Axiom =                                                    4
#    define_type `btree_Axiom` `btree = LEAF * | NODE btree btree`;;
btree_Axiom =
|- !f0 f1.
    ?! fn.
     (!x. fn(LEAF x) = f0 x) /\
     (!b1 b2. fn(NODE b1 b2) = f1(fn b1)(fn b2)b1 b2)
```

[5]See Section 4.6.2 for a discussion of the primitive recursion theorem.

The result returned by the call to `define_type` is, in this case, an abstract characterization for a defined type `(*)btree`, in the form of a 'primitive recursion theorem' for the required type of labelled binary trees.

Any simple concrete recursive type can be defined automatically from a user-supplied equation using `define_type` in exactly the same way.

5.7.2 Defining recursive functions

An important property of the characterizing theorems for concrete types shown in the examples given above is that they provide a formal means for defining recursive functions on those types. When a concrete recursive type *op* is characterized by a theorem of the kind returned by `define_type` (see Section 5.7.1.3) this theorem can be used to prove the existence of any *primitive recursive* function on *op* and to define constants which denote such functions.

This is illustrated for a particular example by the method of defining primitive recursive functions on the natural numbers discussed in Section 4.6.2.1. In that section, an ML function `new_prim_rec_definition` was described which automates the logical inferences necessary to derive particular primitive recursive definitions on the built-in defined type `num` of natural numbers. The basis of this function is the primitive recursion theorem

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

which is pre-proved and stored in the built-in theory `prim_rec` (see Section 4.6.2). The ML function `new_prim_rec_definition` uses `num_Axiom` to automate the justification of any user-supplied primitive recursive definition on the natural numbers.

The type definition package provides a similar function for defining primitive recursive functions on arbitrary concrete recursive types.[6] The ML function

```
new_recursive_definition : bool -> thm -> string -> term -> thm
```

automates the inferences necessary to justify any given primitive recursive definition on a concrete recursive type of the kind definable by `define_type`. It takes four arguments. The first is a boolean flag which indicates if the function to be defined will be an infix or not. The second is the primitive recursion theorem for the concrete type in question (i.e. a theorem obtained from `define_type`). The third argument is a name under which the resulting definition will be saved in the current theory segment. The fourth argument is a term giving the desired primitive recursive definition. The value returned by `new_recursive_definition` is a theorem which states the primitive recursive definition

[6] In fact, `new_prim_rec_definition` is defined in ML using the more general tools provided by the type definition package.

requested by the user. This theorem is derived by formal proof from an instance of the general primitive recursion theorem given as the second argument.

If the ML variable *op*`_Axiom` is bound to a theorem of the form returned by `define_type`, then evaluating:

```
new_recursive_definition
  `flag` op_Axiom `name` "primitive recursive definition on op"
```

automatically proves the existence of the primitive recursive function supplied as the fourth argument, and then declares a new constant in the current theory with this definition as its specification. This constant specification is returned as a theorem and is saved in the current theory segment under the name $name$. If $flag$ is `true`, the constant is given infix status. Failure occurs if:

(i) HOL cannot prove there is a function satisfying the defining equations supplied by the user (i.e. the term supplied to `new_recursive_definition` is not a well-formed primitive recursive definition on values of type *op*);

(ii) any other condition for making a constant specification is violated (see the failure conditions for `new_specification` in Section 3.7.2.2).

Curried functions defined using `new_recursive_definition` can be recursive on any one of their arguments. Furthermore, defining equations need not be given for all the constructors of the concrete type in question. See the examples given in the next section, or the examples of functions defined on `num` given in Section 4.6.2.1 for more details.

The ML function

```
prove_rec_fn_exists : thm -> term -> thm
```

is a version of `new_recursive_definition` which proves only that the required function exists; it does not make a constant specification. The first argument is a theorem of the form returned by `define_type`, and the second is a user-supplied primitive recursive function definition. The theorem which is returned asserts the existence of the recursively-defined function in question (if it is primitive recursive over the type characterized by the theorem given as the first argument).

5.7.2.1 More examples

Continuing the example session started above in Section 5.7.1.4, the following interactions with the system show how the ML function `new_recursive_definition` can be used to define functions on concrete types, which have themselves been defined using `define_type`.

Given the characterizing theorem `btree_Axiom` for the type of labelled binary trees defined in Section 5.7.1.4, a recursive function `Leaves`, which computes the number of leaf nodes in a binary tree, can be defined recursively in HOL as shown below:

```
#let Leaves =                                                          5
#    new_recursive_definition false btree_Axiom `Leaves`
#      "(Leaves (LEAF (x:*)) = 1) /\
#       (Leaves (NODE t1 t2) = (Leaves t1) + (Leaves t2))";;
Leaves =
|- (!x. Leaves(LEAF x) = 1) /\
   (!t1 t2. Leaves(NODE t1 t2) = (Leaves t1) + (Leaves t2))
```

The result of the call to `new_recursive_definition` is a theorem which states that the constant `Leaves` satisfies the primitive-recursive defining equations supplied by the user. This theorem is derived automatically from an instance of the general primitive recursion theorem for binary trees (`btree_Axiom`) and an appropriate constant specification for the constant `Leaves`.

The function defined using `new_recursive_definition` need not, in fact, be recursive. Here is the definition of a predicate `IsLeaf`, which is true of binary trees which are leaves, but is false of the internal nodes in a binary tree:

```
#let IsLeaf =                                                          6
#    new_recursive_definition false btree_Axiom `IsLeaf`
#      "(IsLeaf (NODE t1 t2) = F) /\ (IsLeaf (LEAF (x:*)) = T)";;
IsLeaf = |- (!t1 t2. IsLeaf(NODE t1 t2) = F) /\ (!x. IsLeaf(LEAF x) = T)
```

Note that two equations defining a (recursive or non-recursive) function on binary trees by cases can be given in either order. Here, the `NODE` case is given first, and the `LEAF` case second. The reverse order was used in the above definition of `Leaves`.

The ML function `new_recursive_definition` also allows the user to partially specify the value of a function defined on a concrete type, by allowing defining equations for some of the constructors to be omitted. Here, for example, is the definition of a function `Label` which extracts the label from a leaf node. The value of `Label` applied to an internal node is left unspecified:

```
#let Label =                                                           7
#    new_recursive_definition false btree_Axiom `Label`
#      "Label (LEAF (x:*)) = x";;
Label = |- !x. Label(LEAF x) = x
```

Curried functions can also be defined, and the recursion can be on any argument. The next definition defines an infix (curried) function << which expresses the idea that one tree is a proper subtree of another.

```
#let Subtree =                                                         8
#    new_recursive_definition true btree_Axiom `Subtree`
#      "(<< (t:(*)btree) (LEAF (x:*)) = F) /\
#       (<< t (NODE t1 t2) = ((t=t1) \/ (t=t2) \/ (<< t t1) \/ (<< t t2)))";;
Subtree =
|- (!t x. t << (LEAF x) = F) /\
   (!t t1 t2.
     t << (NODE t1 t2) = (t = t1) \/ (t = t2) \/ t << t1 \/ t << t2)
```

Note that the first argument to the ML function is `true` (to indicate that the function being defined is to have infix status) and that the constant << is an infix after the definition has been made. Furthermore, the function << is recursive on its second argument.

Finally, the function `new_recursive_definition` can also be used to define functions by cases on enumerated types. For example, a predicate `One`, which is true of only the value `ONE` of the three-valued type `three` defined above in Section 5.7.1.4, can be defined as follows:

```
#let One = new_recursive_definition false three_Axiom `One`                9
#            "(One ONE = T) /\ (One TWO = F) /\ (One THREE = F)";;
One = |- (One ONE = T) /\ (One TWO = F) /\ (One THREE = F)
```

The existence only of any function definable using `new_recursive_definition` can be proved using `prove_rec_fn_exists`. For example:

```
#close_theory();;                                                          10
() : void

#let exists = prove_rec_fn_exists three_Axiom
#             "(f ONE = T) /\ (f TWO = F) /\ (f THREE = F)";;
exists = |- ?f. (f ONE = T) /\ (f TWO = F) /\ (f THREE = F)
```

The resulting theorem simply states the existence of the required function. Here, a constant is not defined, and the user need not be in draft mode.

5.7.3 Structural induction

For any concrete recursive type definable using the HOL type definition package there is a structural induction theorem which states the validity of proof by induction on the structure of the type's values. The ML function

```
prove_induction_thm : thm -> thm
```

can be used to derive a structural induction theorem for any concrete recursive type defined using `define_type`. If the ML variable *op*`_Axiom` is bound to a theorem of the form returned by `define_type`, then executing

`prove_induction_thm` *op*`_Axiom`

will prove and return a structural induction theorem for the concrete type *op*. The 'induction' theorem is degenerate in the case of non-recursive types (see the examples given below). Failure occurs, or an unpredictable output theorem is returned, if the input theorem does not have the form of a theorem returned by `define_type`.

5.7.3.1 Examples

A structural induction theorem on the type of binary trees defined in the session beginning on Section 5.7.1.4 can be proved by:

```
#let btree_Induct = prove_induction_thm btree_Axiom;;                    11
btree_Induct =
|- !P.
    (!x. P(LEAF x)) /\ (!b1 b2. P b1 /\ P b2 ==> P(NODE b1 b2)) ==>
    (!b. P b)
```

The output theorem states that a predicate `P` is true of all binary trees if it is true of all labelled leaf nodes, and whenever it is true of two binary trees `b1` and `b2` it is also true of the binary tree `NODE b1 b2`, in which `b1` and `b2` occur as immediate left and right subtrees.

For non-recursive types, the induction theorem returned by `prove_induction_thm` is degenerate: there are no 'step' cases in the induction. For the two types `three` and `rec` defined in the preceding interactions of this session, the induction theorems are:

```
#let three_Induct = prove_induction_thm three_Axiom;;                    12
three_Induct = |- !P. P ONE /\ P TWO /\ P THREE ==> (!t. P t)

#let rec_Induct = prove_induction_thm rec_Axiom;;
rec_Induct = |- !P. (!b0 b1 b2. P(REC b0 b1 b2)) ==> (!r. P r)
```

Here, induction simply reduces to the consideration of cases, one for each of the constructors for the concrete type involved.

5.7.4 Structural induction tactics

This section has been included here for reference because it relates chiefly to the type definition package, but it involves concepts not defined until later, in Chapter 10. Tactics, goals and subgoals are defined in Section 10.1; and theorem continuations, in Section 10.5. `MAP_EVERY` is defined in Section 10.5.1. `ASSUME_TAC` is defined in Section 10.3. `MP_TAC` and `INDUCT_TAC` can be found in *REFERENCE*.

The ML function

```
   INDUCT_THEN : thm -> (thm -> tactic) -> tactic
```

can be used to generate a structural induction tactic for any concrete types definable using `define_type`. The first argument is an induction theorem of the form returned by the function `prove_induction_thm` discussed in the previous section. The second argument is a theorem continuation (see Chapter 10) that determines what is to be done with the induction hypotheses when the resulting tactic is applied to a goal.

If th is an induction theorem for a concrete type *op* with m constructors $C_1, \ldots, C_m$ (i.e. a theorem of the kind returned by `prove_induction_thm`) and F is a theorem continuation, then the tactic `INDUCT_THEN` th F will reduce a goal `(`Γ`,"!`x`:`op`.`$t[x]$`")` to the collection of m induction subgoals generated by:

$$
\begin{array}{l}
\texttt{MAP\_EVERY}\ F\ \texttt{[}th_1^1\texttt{;}\ \ldots\texttt{;}\ th_1^{k_1}\texttt{]}\quad \texttt{(}\Gamma\texttt{, "}t[C_1\, x_1^1\ \ldots\ x_1^{k_1}]\texttt{"),}\\
\qquad\qquad\vdots\\
\texttt{MAP\_EVERY}\ F\ \texttt{[}th_m^1\texttt{;}\ \ldots\texttt{;}\ th_m^{k_m}\texttt{]}\ \texttt{(}\Gamma\texttt{, "}t[C_m\, x_m^1\ \ldots\ x_m^{k_m}]\texttt{")}
\end{array}
$$

where th_i^j is a theorem of the form `|-` $t[x_i^j]$ asserting the truth of $t[x_i^j]$ for the jth recursive argument (for non-recursive arguments, there will be no th_i^j in the list) of the ith constructor C_i (for $1 \leq i \leq m$).

The most common use of `INDUCT_THEN` is in conjunction with the theorem continuation `ASSUME_TAC`. For example, the built-in induction tactic `INDUCT_TAC` for mathematical induction on the natural numbers is defined in ML by:

```
let INDUCT_TAC = INDUCT_THEN INDUCTION ASSUME_TAC
```

This built-in tactic reduces a goal `(`Γ`,"!`$n.t[n]$`")` to a basis subgoal `(`Γ`,"`$t[0]$`")` and a step subgoal `(`$\Gamma \cup \{$`"`$t[n]$`"`$\}$`,"`$t[$`SUC` $n]$`")`. The extra assumption `"`$t[n]$`"` (i.e. the induction hypothesis) is added to the assumptions Γ by `ASSUME_TAC`.

By contrast, the induction tactic `INDUCT_MP_TAC` (which is not built-in) defined by:

```
let INDUCT_MP_TAC = INDUCT_THEN INDUCTION MP_TAC
```

reduces a goal `(`Γ`,"!`$n.t[n]$`")` to a basis subgoal `(`Γ`,"`$t[0]$`")` and an induction step subgoal `(`Γ`, "`$t[n]$ `==>` $t[$`SUC` $n]$`")`. Here, the theorem continuation `MP_TAC` makes the induction hypothesis an antecedent of the step subgoal, rather than an assumption.

As this example illustrates, the theorem continuation F in an induction tactic

`INDUCT_THEN` th F

generated using an induction theorem th can be thought of as a function which determines what is to be done with the induction hypotheses corresponding to the recursive arguments of constructors in the step cases of a proof by structural induction. When F is `ASSUME_TAC`, the induction hypotheses become assumptions in the subgoals generated; and when F is `MP_TAC`, the induction hypotheses become the antecedents of implicative subgoals. Other theorem continuations (for which, see Chapter 10 and *REFERENCE*) can also be used.

5.7.5 Other tools

The function

```
prove_constructors_one_one : thm -> thm
```

proves that the constructors of a concrete type which take arguments are one-to-one. The argument to `prove_constructors_one_one` is a theorem of the form returned by `define_type`.

The function

```
prove_constructors_distinct : thm -> thm
```

proves that the constructors of a concrete type yield distinct values. The argument to `prove_constructors_distinct` is again a theorem of the form returned by `define_type`.

The function

```
prove_cases_thm : thm -> thm
```

proves a cases theorem for any concrete type. Such a theorem states that every value can be constructed using one of the type's constructors. This property follows more easily (and therefore is faster to prove) from induction than from primitive recursion, so the function `prove_cases_thm` takes as an argument an induction theorem of the kind returned by `prove_induction_thm`.

These auxiliary tools work for any concrete type definable using `define_type`.

5.7.5.1 Examples

The following interactions with the system show the proof that the constructor `LEAF` for the type `(*)btree` is one-one, and also that the constructor `REC` for the type `rec` is one-to-one.

```
#let LEAF_one_one = prove_constructors_one_one btree_Axiom;;        13
LEAF_one_one =
|- (!x x'. (LEAF x = LEAF x') = (x = x')) /\
   (!b1 b2 b1' b2'.
     (NODE b1 b2 = NODE b1' b2') = (b1 = b1') /\ (b2 = b2'))

#let REC_one_one = prove_constructors_one_one rec_Axiom;;
REC_one_one =
|- !b0 b1 b2 b0' b1' b2'.
    (REC b0 b1 b2 = REC b0' b1' b2') =
    (b0 = b0') /\ (b1 = b1') /\ (b2 = b2')
```

The function `prove_constructors_one_one` fails when the concrete type involved has no constructors that take arguments. For example:

```
#let th = prove_constructors_one_one three_Axiom;;                    14
evaluation failed     prove_constructors_one_one: invalid input theorem
```

The function `prove_constructors_distinct` returns the theorem stating that the constructors of a concrete type yield pair-wise distinct values. For example:

```
#let NOT_LEAF_NODE = prove_constructors_distinct btree_Axiom;;        15
NOT_LEAF_NODE = |- !x b1 b2. ~(LEAF x = NODE b1 b2)

#let three_distinct = prove_constructors_distinct three_Axiom;;
three_distinct = |- ~(ONE = TWO) /\ ~(ONE = THREE) /\ ~(TWO = THREE)
```

Cases theorems are proved from structural induction theorems. For the binary tree example considered in the present session, here is the cases theorem:

```
#let btree_cases = prove_cases_thm btree_Induct;;                     16
btree_cases = |- !b. (?x. b = LEAF x) \/ (?b1 b2. b = NODE b1 b2)
```

Note that the structural induction theorem for binary trees, `btree_Induct`, is used.

Chapter 6

Miscellaneous Features

This section describes some of the features that exist for managing the interface to the HOL system.

- A help system.
- A theorem database.
- A datatype database.
- A tool for dependency maintenance in large developments.
- Flags for controlling the parsing and printing of terms.
- A function for adjusting the maximum depth to which terms and theorems are printed by the pretty printer (the default is 500).
- Functions for counting the number of primitive inferences done in an evaluation, and timing it.
- A version of the system which allows the implicit invocation of the parsers for HOL types and terms.

6.1 Help

There are several kinds of help available in hol98, all accessible through the same incantation:

```
help <string>;
```

The kinds of help available are:

MoscowML help. 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 hol98.

HOL help. This is the on-line help from Hol88 and Hol90, and 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 HOL90 or HOL88, or even be missing!

HOL structure information. For most structures in the hol98 source, one can get a listing of the entrypoints found in the accompanying signature. This is helpful for locating functions and is automatically derived from the system sources, so it is alway up-to-date.

Theory facts. These are automatically derived from theory files, so they are always up-to-date. The signature of each theory is available (since theories are represented by structures in hol98). Also, each axiom, definition, and theorem in the theory can be accessed by name in the help system; the theorem itself is given.

Therefore the following example queries can be made:

`help "installPP"`	Moscow ML help
`help "hol"`	hol98 overview
`help "aconv"`	on-line HOL help
`help "Tactic"`	HOL source structure information
`help "boolTheory"`	theory structure signature
`help "list_Axiom"`	theory structure signature and theorem statement

6.2 `Holmake`—a tool for maintaining HOL formalizations

The purpose of `Holmake`[1] is to maintain dependencies in a hol98 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 dependency information,e.g., a Makefile. `Holmake` can be very convenient to use, but there are some conventions and restrictions on it that must be followed, which we will describe in the sequel.

`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,

[1] `Holmake` was first written by Ken Larsen and then extended by Michael Norrish.

the user builds up a theory by interacting with HOL, perhaps over many sessions. In 'system rebuild' mode, a component that others depend on has been altered, so all modules dependent on it have to be brought up to date. System rebuild mode is simpler so we deal with it first.

6.2.1 System Rebuild

A system rebuild happens when an existing theory has been improved in some way (augmented with a new theorem, a change to a definition, etc.), or perhaps some support ML code has been modified or added to the formalization under development. The user needs to find and recompile just those modules affected by the change. This is what an invocation of `Holmake` does, by identifying the out-of-date modules and re-compiling and re-executing them.

6.2.2 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, i.e., the text file resulting from session n is "`use`"d to start session $n + 1$; after that, theory construction resumes.

Once the user finishes the perhaps long and arduous task of constructing a theory, the user should

1. make the script separately compilable;
2. invoke `Holmake`. This will (a) compile and execute the script file; and (b) compile the resulting theory file. After this, the theory file is available for use.

6.2.3 Making the script separately compilable

First, the invocation

```
val _ = export_theory();
```

should be added at the end of the file. When the script is finally executed, this call writes the theory to disk.

Second, we address a crucial environmental issue: if a theory script has been constructed using `<holdir>/bin/hol`, then it has been developed in an environment where some commonly used structures, e.g., `Tactic`, have already been loaded and opened for the user's convenience. When we wish to apply `Holmake` to a script developed in this way, we have to take some extra steps to ensure that the compilation environment also provides these structures. In the common case, this is simple; one must only add, at the head of the theory script, the following "boilerplate":

```
open HolKernel Parse basicHol90Lib;
infix THEN THENL THENC ORELSE ORELSEC THEN_TCL ORELSE_TCL ## |->;
infixr -->;
```

This will duplicate the starting environment that one obtains with `<holdir>/bin/hol` and `<holdir>/bin/hol.unquote`.

Now the script should be separately compilable. Invoke `Holmake` to check; MoscowML will flag any unaccounted-for identifiers it finds. The user has to resolve these, either by using the 'dot' notation to locate the identifier for the compiler, or by `open`ing the relevant module. This "compile/resolve-identifier" loop should continue until `Holmake` succeeds in compiling the module.

The following notes may be of some further help.

1. The filenames of theory scripts must follow the following convention: a HOL theory script for theory "x" should be named `xScript.sml`. When `export_theory` is called during an invocation of `Holmake`, the files `xTheory.sig` and `xTheory.sml` will be generated and then compiled.

2. In the MoscowML batch compiler, modules are not allowed to have unbound top-level expressions. Hence, something like the following is not allowed:

   ```
   new_theory "ted";
   ```

   To make Moscow ML happy, one must instead write something like

   ```
   val _ = new_theory "ted";
   ```

3. In the interactive system, one has to explicitly `load` modules; on the other hand, the batch compiler will load modules automatically. For example, in order to execute `open Foo` (or refer to values in `Foo`) in the interactive system, one must first have executed `load "Foo"`. Contrarily, the batch compiler will reject files having occurrences of `load`, since `load` is only defined for the interactive system.

4. Take care not to have the string "Theory" embedded in the name of any of your files. hol98 generates files containing this string, and when it cleans up after itself, it removes such files using a regular expression. This will also remove other files with names containing "Theory". For example, if, in your development directory,

you had a file of ML code named `MyTheory.sml` and you also were managing a hol98 development there with `Holmake`, then MyTheory.sml would get deleted if `Holmake clean` was invoked.

5. We can see that some users may not wish to use (some of) the support provided by `basicHol90Lib`, since it is becoming dated. In that case, the same general principle set out above will apply: the user must ensure that the compilation environment for a theory script is the same as the interactive environment it was developed in.

6.2.4 Summary

A complete theory construction is 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.

6.2.5 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. If one is developing a system over more than one directory, one should write a master Makefile (or shell script) that invokes `Holmake` in the subsidiary directories, in the correct order, i.e.., such that there never is an out-of-date dependence leading outside of the current directory. This should always be achievable, simply by ordering the directories in the order that one would have to "`use`" files in them.

6.2.6 `Holmake`'s command-line arguments

Like `make`, `Holmake` takes command-line arguments corresponding to the targets that the user desires to build. If there are none, then `Holmake` will attempt to build all ML modules and HOL theories it can detect in the current directory. In addition, there are three special targets that can be used:

`clean` Removes all compiled files.

`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, the user can directly affect the workings of `Holmake` with the following command-line options:

`-I <directory>` Look in specified directory for additional MoscowML object files, including other HOL theories. This option can be repeated, with multiple `-I`'s to allow for multiple directories to be referenced.

`-d <file>` Ignore the given file and don't try to build it. The file may be rebuilt anyway if other files you have specified depend on it. This is useful to stop Holmake from attempting to compile files that are interactive scripts (include use of `load` or `use`, for example).

`-f <theory>` Toggles whether or not a theory should be built in "fast" mode. Fast building causes tactic proofs (invocations of `prove` and `store_thm`) to automatically succeed. This lack of soundness is marked by the `fast_proof` oracle tag. This tag will appear on all theorems proved in this way and all subsequent theorems that depend on such theorems. `Holmake`'s default is not to build in fast mode.

`--fast` Makes `Holmake`'s default be to build in fast mode (see above).

`--help` **or** `-h` Prints out a useful option summary and exits.

`--holdir <directory>` Associate this build with the given HOL directory, rather than the one this version of `Holmake` was configured to use by default.

`--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 MoscowML `make` system. However, it will still attempt to do "HOL things" with files whose names end in `Script` and `Theory`.

`--qof` Standing for "quit on failure", if a tactic fails to prove a theorem, quit the build. The default is to use `mk_thm` to assert that the failed goal is true so that the build can continue and other theorems proved.

`--rebuild_deps` **or** `-r` Forces `Holmake` to always rebuild the dependency information, whether or not it thinks it needs to.

`--version` **or** `-v` Show some brief version information. As of this writing, `Holmake` is at version 2.1.1.

`Holmake` should never exit with the MoscowML message “Uncaught exception”. Such behaviour is a bug, please report it!

6.3 Flags for the HOL logic

The subset of flags that control aspects of HOL relating to the logic is summarized in the table below.

Settable system flags		
Flag	*Function*	*Default value*
`timing`	Print number of theorems proved	`false`
`show_types`	Prints types in quotations	`false`
`theory_pp`	Pretty printing of theory files	`false`
`type_error`	Verbose type checking errors in quotations	`true`
`interface_print`	Causes inverse of interface map to be used when printing	`true`

6.4 Hiding constants

The following function can be used to hide the constant status of a name from the quotation parser.

```
hide_constant : string -> void
```

Evaluating `hide_constant` ‘x‘ makes the quotation parser treat x as a variable (lexical rules permitting), even if x is the name of a constant in the current theory (constants and variables can have the same name). This is useful if one wants to use variables with the same names as previously declared (or built-in) constants (e.g. `o`, `I`, `S` etc.). The name x is still a constant for the constructors, theories, etc; `hide_constant` affects only parsing.

Hiding a constant and then attempting to declare it as a new constant will fail (as it must, if the system is to remain sound).

The function

```
unhide_constant : string -> void
```

undoes the hiding; it fails if its argument is not a previously hidden constant.

The function:

```
is_hidden : string -> bool
```

tests whether a string is the name of a hidden constant.

6.5 Adjusting the pretty-print depth

The following ML function can be used to adjust the maximum depth of printing.

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

The default print depth is 500. Evaluating `max_print_depth` n sets the maximum to n and returns the previous value of the maximum. Subterms nested more deeply than the maximum print depth are printed as `&`. For example:

```
#ADD_CLAUSES;;                                                      1
Theorem ADD_CLAUSES autoloaded from theory `arithmetic`.
ADD_CLAUSES =
|- (0 + m = m) /\
   (m + 0 = m) /\
   ((SUC m) + n = SUC(m + n)) /\
   (m + (SUC n) = SUC(m + n))

|- (0 + m = m) /\
   (m + 0 = m) /\
   ((SUC m) + n = SUC(m + n)) /\
   (m + (SUC n) = SUC(m + n))

#max_print_depth 7;;
500 : int

#ADD_CLAUSES;;
|- (& + & = m) /\ (& + & = m) /\ ((& + & = &(&) /\ (& + (& = &(&)

#max_print_depth 5;;
7 : int

#ADD_CLAUSES;;
|- (& /\ (& /\ (& /\ (&

#max_print_depth 3;;
5 : int

#ADD_CLAUSES;;
|- &
```

6.6 Timing and counting theorems

Whenever HOL performs a primitive inference (or accepts an axiom or definition) a counter is incremented. The value of this counter is returned by the function:

```
thm_count : void -> int
```

This counter can be reset with the function:

```
set_thm_count : int -> int
```

The previous value of the counter is returned.

The following function is used to switch ML into a mode in which the number of primitive inferences done during each top-level interaction is shown. Run-time and garbage collection time are also shown.

```
timer : bool -> bool
```

Executing `timer true` causes the number of primitive inferences and timings to be printed; `timer false` switches the printing off. The previous setting is returned. Executing `timer` b is equivalent to setting the flag `timing` to the value b.

6.7 Quotation preprocessing

A person usually works with hol98 by interacting with the ML top level loop in order to build formalizations and perform proofs. In this setting, the user often needs to enter expressions of the HOL logic to ML, and interpret the resulting responses. Since the ML representations of the types, terms, and theorems of the HOL logic are quite unreadable in their 'raw' form, so-called *prettyprinters* for HOL logic expressions are automatically invoked by the ML top level when printing output.

Similarly, types and terms often have to be constructed by the user, e.g., in order to make definitions, state goals to prove, provide existential witnesses, etc. Since it would be unbearable to make a type or term of any size 'by hand', the system comes equipped with parsers for type and term expressions. The parser for types is called `Type`, and the parser for terms is called `Term`. These parsers take *quotations*. A quotation `‘...‘` is much like an SML string, except that it can span several lines without requiring awkward backslashes, as an ML string would.[2]

For added convenience, the HOL system distribution supplies a version of hol98 that features a *combined parser* that accepts both types and terms. Enclosing some object language concrete syntax between occurrences of `‘‘` will result in the correct parser being invoked. For example

[2]Quotations were a feature in the original LCF system. See the MoscowML User's Manual for more information.

```
``x /\ y /\ z ==> ?p. p``
```

will parse as a term while

```
``:'a -> ('b -> 'h) -> bool``
```

parses as an HOL type. Note that the concrete syntax given in the quotation for a type needs to provide a hint: the type parser will only be called if the first character after the leading `` `` `` is a colon (`:`).

Knowledgable ML programmers will notice that the idiom `` ``... `` `` is not ML-typable; for that reason, it is implemented as a pre-processor to ML, thanks to work by Richard Boulton. Users who wish to use the pre-processor should invoke `<hol-dir>/bin/hol.unquote`. `Holmake` will accept source files having occurrences of `` `` ``.

Part III

Theorem Proving with HOL

Chapter 7

Syntax

The HOL logic is a classical higher-order predicate calculus. Its syntax enjoys two main differences from the syntax of standard first order logic.[1] First, there is no distinction in HOL between terms and formulas: HOL has only terms. Second, each term has a type: types are used in order to build well-formed terms. There are two ways to construct types and terms in HOL: by use of a parser, or by use of the programmer's interface. In this chapter, we will focus on the concrete syntax accepted by the parsers, leaving the programmer's interface for Chapter **??**.

7.1 Types

A HOL type can be a variable, a constant, or a compound type, which is a constant of arity n applied to a list of n types.

hol_type	::=	*'ident*	(type variable)
	|	`bool`	(type of truth values)
	|	`ind`	(type of individuals)
	|	*hol_type* `->` *hol_type*	(function arrow)
	|	*hol_type ident hol_type*	(binary compound type)
	|	*ident*	(nullary type constant)
	|	*hol_type ident*	(unary compound type)
	|	$(\mathit{hol\_type}_1, \ldots, \mathit{hol\_type}_n)\mathit{ident}$	(compound type)

Type constants are also known as type operators. They must be alphanumeric. Type variables are alphanumerics written with a leading prime ('). In `hol98`, the type constants `bool`, `fun`, and `ind` are primitive. The introduction of new type constants is described in Chapter **??**. `bool` is the two element type of truth values. The binary operator `fun` is used to denote function types; it can be written with an infix arrow. The nullary type constant `ind` denotes an infinite set of individuals; it is used for a few highly technical developments in the system and can be ignored by beginners. Thus

```
'a -> 'b
(bool -> 'a) -> ind
```

[1]We assume the reader is familiar with first order logic.

are both well-formed types. The function arrow is ”right associative”, which means that ambiguous uses of the arrow in types are resolved by adding parentheses in a right-to-left sweep: thus the type expression

```
ind -> ind -> ind -> ind
```

is identical to

```
ind -> (ind -> (ind -> ind)).
```

The product (`#`) and sum (`+`) are other infix type operators, also right associative; however, they are not loaded by default in hol98. How to load in useful logical context is dealt with in Chapter **??**.

7.2 Terms

Ultimately, a HOL term can only be a variable, a constant, an application, or a lambda term.

term	::=	*ident*	(variable or constant)
	|	*term term*	(combination)
	|	*ident*. *term*	(lambda abstraction)

In the system, the usual logical operators have already been defined, including truth (`T`), falsity (`F`), negation (`~`), equality (`=`), conjunction (`/\`), disjunction (`\/`), implication (`==>`), universal (`!`) and existential (`?`) quantification, and an indefinite description operator (`@`). As well, the basis includes conditional, lambda, and ‘let’ expressions.

Thus the set of terms available is, in general, an extension of the following grammar:

<table>
<tr><td>term</td><td>::=</td><td>term : hol_type</td><td>(type constraint)</td></tr>
<tr><td></td><td>|</td><td>term term</td><td>(application)</td></tr>
<tr><td></td><td>|</td><td><code>~</code>term</td><td>(negation)</td></tr>
<tr><td></td><td>|</td><td>term = term</td><td>(equality)</td></tr>
<tr><td></td><td>|</td><td>term <code>==></code> term</td><td>(implication)</td></tr>
<tr><td></td><td>|</td><td>term <code>\/</code> term</td><td>(disjunction)</td></tr>
<tr><td></td><td>|</td><td>term <code>/\</code> term</td><td>(conjunction)</td></tr>
<tr><td></td><td>|</td><td><code>if</code> term <code>then</code> term <code>else</code> term</td><td>(conditional)</td></tr>
<tr><td></td><td>|</td><td>$\backslash ident_1 \ldots ident_n.\ term$</td><td>(lambda abstraction)</td></tr>
<tr><td></td><td>|</td><td><code>!</code>$ident_1 \ldots ident_n.\ term$</td><td>(forall)</td></tr>
<tr><td></td><td>|</td><td><code>?</code>$ident_1 \ldots ident_n.\ term$</td><td>(exists)</td></tr>
<tr><td></td><td>|</td><td><code>@</code>$ident_1 \ldots ident_n.\ term$</td><td>(choose)</td></tr>
<tr><td></td><td>|</td><td><code>?!</code>$ident_1 \ldots ident_n.\ term$</td><td>(exists-unique)</td></tr>
<tr><td></td><td>|</td><td><code>let</code> ident = term
[<code>and</code> ident = term]* <code>in</code> term</td><td>(let expression)</td></tr>
<tr><td></td><td>|</td><td><code>T</code></td><td>(truth)</td></tr>
<tr><td></td><td>|</td><td><code>F</code></td><td>(falsity)</td></tr>
<tr><td></td><td>|</td><td>ident</td><td>(constant or variable)</td></tr>
<tr><td></td><td>|</td><td>(term)</td><td>(parenthesized term)</td></tr>
</table>

Some examples may be found in Table 7.1. Term application can be iterated. Application is left associative so that *term term term* ... *term* is equivalent in the eyes of the parser to $(\ldots((term\ term)\ term)\ldots)\ term$.

The lexical structure for term identifiers is much like that for ML: identifiers can be alphanumeric or symbolic. Variables must be alphanumeric. A symbolic identifier is any concatenation of the characters in the following list:

```
#?+*/\\=<>&%@!,:;_|~-
```

with the exception of the keywords `\\`, `;`, `=>`, `|`, and `:` (colon). Any alphanumeric can be a constant except the keywords `let`, `in`, `and`, and `of`.

<table>
<tr><td align="right"><code>x = T</code></td><td>x is equal to true.</td></tr>
<tr><td align="right"><code>!x. Person x ==> Mortal x</code></td><td>All persons are mortal.</td></tr>
<tr><td align="right"><code>!x y z. (x ==> y) /\ (y ==> z) ==> x ==> z</code></td><td>Implication is transitive.</td></tr>
<tr><td align="right"><code>!x. P x ==> Q x</code></td><td>P is a subset of Q</td></tr>
<tr><td align="right"><code>S = \f g x. f x (g x)</code></td><td>Definition of a famous combinator.</td></tr>
</table>

Table 7.1: Concrete Syntax Examples

7.2.1 Constants

The HOL grammar gets extended when a new constant is introduced. The introduction of new constants will be discussed in section **??**. In order to provide some notational flexibility, constants come in various flavours or *fixities*: besides being an ordinary constant (with a fixity of Prefix), constants can also be *binders*, *true prefixes*[2], *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.

7.2.1.1 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.
```

7.2.1.2 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

[2]The use of the term "true prefix" is forced upon us by the history of the system, which reserved the classification "prefix" for terms without any special syntactic features.

[3]When `pairTheory` has been loaded.

```
((X /\ Y) ==> (C \/ D)), ((P = E), Q).
```

An expression

term `<infix>` *term*

is internally represented as

((`<infix>` *term*) *term*)

.

7.2.1.3 True prefixes

Where infixes appear between their arguments, true prefixes appear before theirs. This might initially appear to be the same thing as happens with normal function application (is f in $f(x)$ not acting as a prefix?), but in fact, it is useful to allow for prefixes to have binding power less than that associated with 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 true prefix, if only to allow one to write

$negop\ negop\ 3$

(whatever *negop* happened to be) without needing extra parentheses.

7.2.1.4 Suffixes

Suffixes appear after their arguments. There are no suffixes introduced into the standard theories available in HOL, but users are always able to introduce their own if they choose. Suffixes are associated with a precedence just as infixes and true prefixes are. If `p` is a true 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

7.2.2 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`. Similarly, `T:bool` performs a (vacuous) constraint of the constant `T` to `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 (though, again redundant) 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.

7.2.2.1 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.

7.2.2.2 Type inference

Consider the term `x = T`. Each term (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 hol98. Thus, hol98 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 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.

7.2.3 Expanded term grammar

There is some further syntax that is specially treated by the parser. The theory of pairs introduces the infix pairing operator `(,)` as well as the corresponding infix product `(#)` type operator. The theory of sets introduces notation for the empty set `{}` (or `EMPTY`), membership (the infix `IN`) insertion (the infix `INSERT`), set comprehension, enumerated sets, and many other defined constants. The theory of lists introduces the constants `NIL` (the surface syntax `[]` can be used) and `CONS`, as well as notation for enumerated lists. The theories of (Peano) numbers and strings introduce the constructors `0`, `SUC`, `""`, and `STRING`, as well as literals for numbers and strings. If the theory of restricted quantifiers is present, syntax is provided for constraining bound variables by predicates.

Thus, if the theories of pairs, sets, numbers, strings, lists, and restricted quantifiers are loaded, the HOL grammar is an extension of that in Table 7.2.

In the table, the varstruct (*vstr*) construct is used. A varstruct is (apparently) an arbitrarily nested tuple of variables, where each variable only occurs once. The translation of varstructs into the internal abstract syntax trees is complex, so we avoid the

<table>
<tr><td>term</td><td>::=</td><td>term : hol_type</td><td>(type constraint)</td></tr>
<tr><td></td><td>|</td><td>term term</td><td>(application)</td></tr>
<tr><td></td><td>|</td><td><code>CONS</code> term term</td><td>(list builder)</td></tr>
<tr><td></td><td>|</td><td><code>INSERT</code> term term</td><td>(set builder)</td></tr>
<tr><td></td><td>|</td><td><code>SUC</code> term</td><td>(successor)</td></tr>
<tr><td></td><td>|</td><td><code>~</code>term</td><td>(negation)</td></tr>
<tr><td></td><td>|</td><td>term = term</td><td>(equality)</td></tr>
<tr><td></td><td>|</td><td>term <code>==></code> term</td><td>(implication)</td></tr>
<tr><td></td><td>|</td><td>term <code>\/</code> term</td><td>(disjunction)</td></tr>
<tr><td></td><td>|</td><td>term <code>/\</code> term</td><td>(conjunction)</td></tr>
<tr><td></td><td>|</td><td>term <code><</code> term</td><td>(less-than)</td></tr>
<tr><td></td><td>|</td><td>term <code>+</code> term</td><td>(addition)</td></tr>
<tr><td></td><td>|</td><td>term <code>*</code> term</td><td>(multiplication)</td></tr>
<tr><td></td><td>|</td><td>term <code>-</code> term</td><td>(subtraction)</td></tr>
<tr><td></td><td>|</td><td>term <code>=></code> term | term</td><td>(conditional)</td></tr>
<tr><td></td><td>|</td><td>\$vstr_1 \ldots vstr_n$. term</td><td>(lambda abstraction)</td></tr>
<tr><td></td><td>|</td><td><code>!</code>$vstr_1 \ldots vstr_n$. term</td><td>(forall)</td></tr>
<tr><td></td><td>|</td><td><code>?</code>$vstr_1 \ldots vstr_n$. term</td><td>(exists)</td></tr>
<tr><td></td><td>|</td><td><code>@</code>$vstr_1 \ldots vstr_n$. term</td><td>(choose)</td></tr>
<tr><td></td><td>|</td><td><code>?!</code>$vstr_1 \ldots vstr_n$. term</td><td>(exists-unique)</td></tr>
<tr><td></td><td>|</td><td><code>let</code> vstr = term
[<code>and</code> vstr = term]* <code>in</code> term</td><td>
(let expression)</td></tr>
<tr><td></td><td>|</td><td><code>T</code></td><td>(truth)</td></tr>
<tr><td></td><td>|</td><td><code>F</code></td><td>(falsity)</td></tr>
<tr><td></td><td>|</td><td><code>[]</code></td><td>(empty list)</td></tr>
<tr><td></td><td>|</td><td><code>{}</code></td><td>(empty set)</td></tr>
<tr><td></td><td>|</td><td><code>(</code>term<code>,</code>term<code>)</code></td><td>(pair)</td></tr>
<tr><td></td><td>|</td><td>ident</td><td>(constant or variable)</td></tr>
<tr><td></td><td>|</td><td>numeral</td><td>(numeric literal)</td></tr>
<tr><td></td><td>|</td><td><code>"</code>charseq<code>"</code></td><td>(string literal)</td></tr>
<tr><td></td><td>|</td><td><code>(</code>term<code>)</code></td><td>(parenthesized term)</td></tr>
<tr><td></td><td>|</td><td><code>[</code>term<code>;</code> ... <code>;</code>term<code>]</code></td><td>(enumerated list)</td></tr>
<tr><td></td><td>|</td><td><code>{</code>term<code>;</code> ... <code>;</code>term<code>}</code></td><td>(enumerated set)</td></tr>
<tr><td></td><td>|</td><td><code>{</code>term <code>|</code> term<code>}</code></td><td>(set comprehension)</td></tr>
</table>

Table 7.2: Expanded Term Grammar

explanation (for this draft).

$$\begin{array}{lcl} \mathit{vstr} & ::= & \mathit{ident} : \mathit{hol\_type} \\ & | & \mathit{ident} \\ & | & \mathit{vstr}\,,\mathit{vstr} \\ & | & (\mathit{vstr}) \\ & | & (\mathit{vstr}::\mathit{term}) \end{array}$$

The `::` syntax is used with restricted quantifiers to allow arbitrary predicates to restrict binding variables. Further to the above, the default grammar also 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.

Also, in the term grammar a *charseq* is just a finite sequence of characters.

7.3 Changes from older versions

This section of the manual documents the (extensive) changes made to the parsing of HOL terms and types in the Taupo release 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. This difference can be seen in the following expression which parses successfully in the old version:

  $\texttt{let}\ x\ =\ p \Rightarrow q \mid r\ \texttt{in}\ Q$

  In Taupo releases and later, this expression will not parse because the conditional expression binds to the left more weakly than the equality binds to the right, and the parser ends up believing that the binding between the `let` and the `in` is not an equality after all, as it should be.

- Old style conditional expressions in the right half of set comprehensions have to be parenthesised to avoid confusing the parser. Thus

  $\{\ x \mid p \Rightarrow q \mid r\ \}$ must be written $\{\ x \mid (p \Rightarrow q \mid r)\ \}$

  Better yet, `if-then-else` syntax could be used for the conditional expression.

- 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.

7.3.1 Error messages

When complete this subsection will document all of the possibly confusing error messages that the new parser and lexing code might generate.

7.3.2 Parser tricks and magic

Here we describe how to achieve some useful effects with the new "Taupo" parser in hol98 available in releases from Taupo-1 onwards.

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 `TruePrefix`. 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 = TruePrefix 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 is in `src/bool/boolScript.sml`.

Mix-fix syntax for term substitution: Here we want 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 `TruePrefix`, but the choice of the square brackets (`[` and `]`) as delimiters would conflict with the concrete syntax for list literals if we did this. Given that list literals are effectively of the `CloseFix` class, we need to make our new syntax the same. 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.[4] 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))};
```

[4]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

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`. First we should allow for overloading on the new name at the exact type of the old. Thus:

```
val _ = allow_for_overloading_on ("MEM",
                                  Type`:'a -> 'a list -> bool`);
```

The next step is 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.

Chapter 8

Derived Inference Rules

The notion of *proof* was defined in the abstract in Chapter 1: a proof of a sequent (Γ, t) from a set of sequents Δ (with respect to a deductive system $\mathcal{D}$) was defined to be a chain of sequents culminating in (Γ, t), such that every element of the chain either belongs to Δ or else follows from Δ and earlier elements of the chain by deduction. The notion of a *theorem* was also defined in Chapter 1: 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 3.9, where the eight families of primitive inferences making up the deductive system were specified by diagrams. It was explained that these families of inferences are represented in HOL via ML functions, and that theorems are represented by an ML abstract type called `thm`. The eight ML functions corresponding to the inferences are operations of the type `thm`, and each of the eight returns a value of type `thm`. It was explained that the type `thm` has primitive destructors, but no primitive constructor; and that in that way, the logic is protected against the computation of theorems except by functions representing primitive inferences, or compositions of these.

Finally, the primitive HOL logic was supplemented by three primitive constants and five axioms, to form the basic logic[1]. The primitive inferences, together with the primitive constants, the five axioms, and a collection of definitions, give a starting point for constructing proofs, and hence computing theorems. However, proving even the simplest theorems from this minimal basis costs considerable effort. The basis does not immediately provide the transitivity of equality, for example, or a means of universal quantification; both of these themselves have to be derived.

8.1 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]This corresponds to the HOL theory `BASIC-HOL`; see Section **??**

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`.

```
#top_print print_all_thm;;                                              1
- : (thm -> void)

#let th1 = ASSUME "t1 ==> t2";;
th1 = t1 ==> t2 |- t1 ==> t2

#let th2 = ASSUME "t1:bool";;
th2 = t1 |- t1

#MP th1 th2;;
t1 ==> t2, t1 |- t2
```

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

```
#set_flag(`timing`, true);;                                             2
false : bool
Run time: 0.0s

#MP(ASSUME "t1 ==> t2")(ASSUME "t1:bool");;
t1 ==> t2, t1 |- t2
Run time: 0.0s
Intermediate theorems generated: 3
```

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

It is worth emphasising that, in either case, every primitive inference in the proof chain is made, in the sense that for each inference, the corresponding ML function is evaluated. That is, HOL permits no short-cut around the necessity of performing complete proofs. The short-cut provided by derived inference rules (as implemented in ML) is around the necessity of *specifying* every step; something that would be impossible for a proof of any length. It can be seen from this that the derived rule, and its representation as an ML function, is essential to the HOL methodology; theorem proving would be otherwise impossible.

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

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

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

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

This general rule is valid because from a HOL theorem of the form Γ `|-` t_1`==>`t_2, the theorem $\Gamma \cup \{t_1\}$ `|-` t_2 can be derived as for the specific instance above. The rule can be implemented in ML as a function (`UNDISCH`, say) that calls the appropriate sequence of primitive inferences. The ML definition of `UNDISCH` is simply

```
#let UNDISCH th = MP th (ASSUME(fst(dest_imp(concl th))));;                3
UNDISCH = - : (thm -> thm)
```

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

```
#th;;                                                                      1
|- (t1 ==> t2) ==> (t2 ==> t1) ==> (t1 = t2)
Run time: 0.0s

#set_flag(`timing`,true);;
true : bool
Run time: 0.0s

#UNDISCH th;;
t1 ==> t2 |- (t2 ==> t1) ==> (t1 = t2)
Run time: 0.1s
Intermediate theorems generated: 2

#UNDISCH it;;
t1 ==> t2, t2 ==> t1 |- t1 = t2
Run time: 0.0s
Intermediate theorems generated: 2
```

Each successful application of `UNDISCH` to a theorem invokes an application of `ASSUME`, followed by an application of `MP`; `UNDISCH` constructs the 2-step proof for any given theorem (of appropriate form). As can be seen, it relies on the class of ML functions that access HOL syntax: in particular, `concl` to produce the conclusion of the theorem, `dest_imp` to separate the implication, and the selector `fst` to choose the antecedent.

This particular example is very simple, but a derived inference rule can perform proofs of arbitrary length. It can also make use of previously defined rules. In this way, the normal inference patterns can be developed much more quickly and easily; transitivity, generalization, and so on, support the familiar patterns of inference.

A number of derived inference rules are pre-defined when the HOL system is entered (of which `UNDISCH` is one of the first). In Section 8.3, 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*.

8.2 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 8.3.13 and 8.3.11) 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 9.

In practice, the derived rule `REWRITE_RULE` plays a central role in proofs, because it takes over a very large number of inferences which may happen in a complex and unpredictable order. It is unlike any other primitive or pre-defined rule, first because of the number of inferences it generates[2]; and second because its outcome is often unexpected. Its power is increased by the fact that any existing equational theorem can be supplied as a ‘rewrite rule’, including a standard HOL set of pre-proved tautologies; and these rewrite rules can interact with each other in the rewriting process to transform the original theorem.

The application of `REWRITE_RULE`, in the session below, illustrates that replacements are made at all levels of the structure of a term. The example is numerical; the infixes `"$>"` and `"$<"` are the usual ‘greater than’ and ‘less than’ relations, respectively, and `"SUC"`, the usual successor function. Use is made of the pre-existing definition of `"$>"`: `GREATER` (see *REFERENCE*). The timing facility is used again, for interest, and the printing of theorems is adjusted as above.

1

```
#top_print print_all_thm;;
- : (thm -> void)

#set_flag(‘timing‘,true);;
false : bool
Run time: 0.0s

#REWRITE_RULE
 [GREATER]
 (ASSUME "SUC 4 > 0 = (SUC 3 > 0 = (SUC 2 > 0 = (SUC 1 > 0 = SUC 0 > 0)))");;
##Definition GREATER autoloaded from theory ‘arithmetic‘.
GREATER = |- !m n. m > n = n < m
Run time: 1.5s
Intermediate theorems generated: 1

(SUC 4) > 0 =
((SUC 3) > 0 = ((SUC 2) > 0 = ((SUC 1) > 0 = (SUC 0) > 0)))
|- 0 < (SUC 4) =
   (0 < (SUC 3) = (0 < (SUC 2) = (0 < (SUC 1) = 0 < (SUC 0))))
Run time: 0.3s
Intermediate theorems generated: 23
```

Notice that rewriting equations can be extracted from universally quantified theorems. To construct the proof step-wise, with all of the instantiations, substitutions, and uses of transitivity, etc., would be a lengthy process. The rewriting rules make it easy, and do so whilst still generating the entire chain of inferences.

[2]The number of inferences performed by this rule is generally ‘inflated’; i.e. is generally greater than the length of the proof itself, if the proof could be ‘seen’. This is because, in the current implementation, some inference is done during the search phase that is not necessarily in support of successful replacements.

8.3 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 **??**), the eight primitive inferences of the HOL logic, and inferences defined earlier in the sequence.

Theorems, in accordance with the definition given at the beginning of this chapter, are treated as rules without hypotheses; thus the derivation of a theorem resembles the derivation of a rule except in not having hypotheses. (The derivation of `TRUTH`, Section 8.3.9, 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 9 for further discussion of this point.) (`UNDISCH` is not a rule of this sort, as it can, in fact, be expressed as a theorem.)

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

In addition, for reasons that are mostly historical, not all the inferences that are derived in terms of the abstract logic are actually derived in the current version of the HOL system. That is, there are currently about forty rules that are installed in the system on an 'axiomatic' basis, all of which should be derived by explicit inference. Although the current status of these rules is not satisfactory, and it is planned, as a high priority, to derive them properly in a future version, their current status does not actually compromise the consistency of the logic. In effect, the existing HOL system has a deductive system more comprehensive than the one presented abstractly, but the model outlined in Chapter 2 would easily extend to cover it.

For reference, in HOL Version 2.0 the following rules that should be derived are not derived, but (for efficiency) are implemented as primitives. The list includes some conversions and conversion-valued functions (conversions are discussed in Chapter 9).

`ADD_ASSUM`	`CONTR`	`IMP_ANTISYM_RULE`
`ALPHA`	`DEF_EXISTS_RULE`	`IMP_TRANS`
`AP_TERM`	`DISJ_CASES`	`INST`
`AP_THM`	`DISJ1`	`MK_ABS`
`SUBS`	`DISJ2`	`MK_COMB`
`SUBS_OCCS`	`EQ_IMP_RULE`	`MK_EXISTS`
`CCONTR`	`EQ_MP`	`NOT_ELIM`
`CHOOSE`	`EQT_INTRO`	`NOT_INTRO`
`CONJ`	`ETA_CONV`	`num_CONV`
`EXISTS`	`SPEC`	`TRANS`
`EXT`	`SUBST_CONV`	`CONJUNCT1`
`GEN`	`SYM`	`CONJUNCT2`

The derivations that follow consist of sequences of numbered steps each of which

1. is an axiom, or
2. is a hypothesis of the rule being derived, or
3. follows from preceding steps by a rule of inference (either primitive or previously derived).

Note that the abbreviation `conv` is used for the ML type `term -> thm`.[3]

8.3.1 Adding an assumption

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

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

1. $t' \vdash t'$ [`ASSUME`]
2. $\Gamma \vdash t$ [Hypothesis]
3. $\Gamma \vdash t' \Rightarrow t$ [`DISCH` 2]
4. $\Gamma,\ t' \vdash t$ [`MP` 3,1]

[3] This stands for 'conversion', as explained in Chapter 9.

8.3.2 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]

8.3.3 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]

8.3.4 Transitivity of equality

```
TRANS : thm -> thm -> thm
```

$$\frac{\Gamma_1 \ \vdash \ t_1 = t_2 \qquad\qquad \Gamma_2 \ \vdash \ t_2 = t_3}{\Gamma_1 \cup \Gamma_2 \ \vdash \ t_1 = t_3}$$

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]

8.3.5 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]

8.3.6 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]

8.3.7 Modus Ponens for equality

```
EQ_MP : thm -> thm -> thm
```

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

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]

8.3.8 Implication from equality

```
EQ_IMP_RULE : thm -> (thm # thm)
```

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_1 \Rightarrow t_2 \qquad\qquad \Gamma \vdash t_2 \Rightarrow t_1}$$

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]

8.3.9 $\top$-Introduction

```
TRUTH
```

$$\vdash \top$$

1. $\vdash \top = ((\lambda x.\ x) = (\lambda x.\ x))$ [Definition of $\top$]
2. $\vdash ((\lambda x.\ x) = (\lambda x.\ x)) = \top$ [SYM 1]
3. $\vdash (\lambda x.\ x) = (\lambda x.\ x)$ [REFL]
4. $\vdash \top$ [EQ_MP 2,3]

8.3.10 Equality-with-$\top$ elimination

```
EQT_ELIM : thm -> thm
```

$$\frac{\Gamma \vdash t = \top}{\Gamma \vdash t}$$

1. $\Gamma \vdash t = \top$ [Hypothesis]
2. $\Gamma \vdash \top = t$ [`SYM` 1]
3. $\vdash \top$ [`TRUTH`]
4. $\Gamma \vdash t$ [`EQ_MP` 2,3]

8.3.11 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. $\vdash \forall = (\lambda P.\ P = (\lambda x.\ \top))$ [`INST_TYPE` applied to the definition of ∀]
2. $\Gamma \vdash \forall(\lambda x.\ t)$ [Hypothesis]
3. $\Gamma \vdash (\lambda P.\ P = (\lambda x.\ \top))(\lambda x.\ t)$ [`SUBST` 1,2]
4. $\vdash (\lambda P.\ P = (\lambda x.\ \top))(\lambda x.\ t) = ((\lambda x.\ t) = (\lambda x.\ \top))$ [`BETA_CONV`]
5. $\Gamma \vdash (\lambda x.\ t) = (\lambda x.\ \top)$ [`EQ_MP` 4,3]
6. $\Gamma \vdash (\lambda x.\ t)\ t' = (\lambda x.\ \top)\ t'$ [`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. $\Gamma \vdash t[t'/x] = (\lambda x.\ \top)\ t'$ [`TRANS` 8,6]
10. $\vdash (\lambda x.\ \top)\ t' = \top$ [`BETA_CONV`]
11. $\Gamma \vdash t[t'/x] = \top$ [`TRANS` 9,10]
12. $\Gamma \vdash t[t'/x]$ [`EQT_ELIM` 11]

8.3.12 Equality-with-⊤ introduction

```
EQT_INTRO : thm -> thm
```

$$\frac{\Gamma \vdash t}{\Gamma \vdash t = \top}$$

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]

8.3.13 Generalization ($\forall$-introduction)

```
GEN : term -> thm -> thm
```

$$\frac{\Gamma\ \vdash\ t}{\Gamma\ \vdash\ \forall x.\ t}$$

- Where x is not free in Γ.

1. $\Gamma\ \vdash\ t$ [Hypothesis]
2. $\Gamma\ \vdash\ t = \mathsf{T}$ [EQT_INTRO 1]
3. $\Gamma\ \vdash\ (\lambda x.\ t) = (\lambda x.\ \mathsf{T})$ [ABS 2]
4. $\vdash\ \forall(\lambda x.\ t) = \forall(\lambda x.\ t)$ [REFL]
5. $\vdash\ \forall = (\lambda P.\ P = (\lambda x.\ \mathsf{T}))$ [INST_TYPE applied to the definition of $\forall$]
6. $\vdash\ \forall(\lambda x.\ t) = (\lambda P.\ P = (\lambda x.\ \mathsf{T}))(\lambda x.\ t)$ [SUBST 5,4]
7. $\vdash\ (\lambda P.\ P = (\lambda x.\ \mathsf{T}))(\lambda x.\ t) = ((\lambda x.\ t) = (\lambda x.\ \mathsf{T}))$ [BETA_CONV]
8. $\vdash\ \forall(\lambda x.\ t) = ((\lambda x.\ t) = (\lambda x.\ \mathsf{T}))$ [TRANS 6,7]
9. $\vdash\ ((\lambda x.\ t) = (\lambda x.\ \mathsf{T})) = \forall(\lambda x.\ \mathsf{T})$ [SYM 8]
10. $\Gamma\ \vdash\ \forall(\lambda x.\ t)$ [EQ_MP 9,3]

8.3.14 Simple α-conversion

```
SIMPLE_ALPHA
```

$$\vdash\ (\lambda x_1.\ t\ x_1) = (\lambda x_2.\ t\ x_2)$$

- Where neither x_1 nor x_2 occurs free in t.[4]

1. $\vdash\ (\lambda x_1.\ t\ x_1)\ x = t\ x$ [BETA_CONV]
2. $\vdash\ (\lambda x_2.\ t\ x_2)\ x = t\ x$ [BETA_CONV]
3. $\vdash\ t\ x = (\lambda x_2.\ t\ x_2)\ x$ [SYM 2]
4. $\vdash\ (\lambda x_1.\ t\ x_1)\ x = (\lambda x_2.\ t\ x_2)\ x$ [TRANS 1,3]
5. $\vdash\ (\lambda x.\ (\lambda x_1.\ t\ x_1)\ x) = (\lambda x.\ (\lambda x_2.\ t\ x_2)\ x)$ [ABS 4]
6. $\vdash\ \forall f.\ (\lambda x.\ f\ x) = f$ [Appropriately type-instantiated axiom]
7. $\vdash\ (\lambda x.\ (\lambda x_1.\ t\ x_1)x) = \lambda x_1.\ t\ x_1$ [SPEC 6]
8. $\vdash\ (\lambda x.\ (\lambda x_2.\ t\ x_2)x) = \lambda x_2.\ t\ x_2$ [SPEC 6]
9. $\vdash\ (\lambda x_1.\ t\ x_1) = (\lambda x.\ (\lambda x_1.\ t\ x_1)x)$ [SYM 7]
10. $\vdash\ (\lambda x_1.\ t\ x_1) = (\lambda x.\ (\lambda x_2.\ t\ x_2)x)$ [TRANS 9,5]
11. $\vdash\ (\lambda x_1.\ t\ x_1) = (\lambda x_2.\ t\ x_2)$ [TRANS 10,8]

8.3.15 η-conversion

```
ETA_CONV : conv
```

$$\vdash\ (\lambda x'.\ t\ x') = t$$

- Where x' does not occur free in t (we use x' rather than just x to motivate the use of SIMPLE_ALPHA in the derivation below).

1. $\vdash\ \forall f.\ (\lambda x.\ f\ x) = f$ [Appropriately type-instantiated axiom]
2. $\vdash\ (\lambda x.\ t\ x) = t$ [SPEC 1]
3. $\vdash\ (\lambda x'.\ t\ x') = (\lambda x.\ t\ x)$ [SIMPLE_ALPHA]
4. $\vdash\ (\lambda x'.\ t\ x') = t$ [TRANS 3,2]

[4] SIMPLE_ALPHA is included here because it is used in a subsequent derivation, but it is not actually in the HOL system, as it is subsumed by other functions.

8.3.16 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]

8.3.17 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]

8.3.18 ε-elimination

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

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

- Where v occurs nowhere except in the assumption $t_1\ v$ of the second hypothesis.

1. $\Gamma_2,\ t_1\ v \ \vdash \ t$ [Hypothesis]
2. $\Gamma_2 \ \vdash \ t_1\ v \Rightarrow t$ [DISCH 1]
3. $\Gamma_2 \ \vdash \ \forall v.\ t_1\ v \Rightarrow t$ [GEN 2]
4. $\Gamma_2 \ \vdash \ t_1(\varepsilon\ t_1) \Rightarrow t$ [SPEC 3]
5. $\Gamma_1 \ \vdash \ t_1(\varepsilon\ t_1)$ [Hypothesis]
6. $\Gamma_1 \cup \Gamma_2 \ \vdash \ t$ [MP 4,5]

8.3.19 $\exists$-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]

8.3.20 $\exists$-elimination

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

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

- Where $t[v]$ denotes a term t with some free occurrences of the variable v singled out, and $t[x]$ denotes the result of replacing these occurrences of v by x, subject to the restriction that x doesn't become bound after substitution.

1. $\vdash \ \exists = \lambda P.\ P(\varepsilon\ P)$ [INST_TYPE applied to the definition of $\exists$]
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]

8.3.21 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]

8.3.22 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]

8.3.23 $\wedge$-introduction

```
CONJ : thm -> thm -> thm
```

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

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]

8.3.24 $\wedge$-elimination

```
CONJUNCT1 : thm -> thm, CONJUNCT2 : thm -> thm
```

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

1. $\vdash \wedge = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b$ [Definition of $\wedge$]
2. $\vdash t_1 \wedge t_2 = \forall b.\ (t_1 \Rightarrow (t_2 \Rightarrow b)) \Rightarrow b$ [RIGHT_LIST_BETA 1]
3. $\Gamma \vdash t_1 \wedge t_2$ [Hypothesis]
4. $\Gamma \vdash \forall b.\ (t_1 \Rightarrow (t_2 \Rightarrow b)) \Rightarrow b$ [EQ_MP 2,3]
5. $\Gamma \vdash (t_1 \Rightarrow (t_2 \Rightarrow t_1)) \Rightarrow t_1$ [SPEC 4]
6. $t_1 \vdash t_1$ [ASSUME]
7. $t_1 \vdash t_2 \Rightarrow t_1$ [DISCH 6]
8. $\vdash t_1 \Rightarrow (t_2 \Rightarrow t_1)$ [DISCH 7]
9. $\Gamma \vdash t_1$ [MP 5,8]
10. $\Gamma \vdash (t_1 \Rightarrow (t_2 \Rightarrow t_2)) \Rightarrow t_2$ [SPEC 4]
11. $t_2 \vdash t_2$ [ASSUME]
12. $\vdash t_2 \Rightarrow t_2$ [DISCH 11]
13. $\vdash t_1 \Rightarrow (t_2 \Rightarrow t_2)$ [DISCH 12]
14. $\Gamma \vdash t_2$ [MP 10,13]
15. $\Gamma \vdash t_1$ and $\Gamma \vdash t_2$ [9,14]

8.3.25 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]

8.3.26 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]

8.3.27 ∨-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]

8.3.28 Classical contradiction rule

```
CCONTR : term -> thm -> thm
```

$$\frac{\Gamma,\ \neg t \ \vdash \ \mathsf{F}}{\Gamma \ \vdash \ t}$$

1. $\vdash \ \neg = \lambda b.\ b \Rightarrow \mathsf{F}$ [Definition of $\neg$]
2. $\vdash \ \neg t = t \Rightarrow \mathsf{F}$ [`RIGHT_LIST_BETA` 1]
3. $\Gamma,\ \neg t \ \vdash \ \mathsf{F}$ [Hypothesis]
4. $\Gamma \ \vdash \ \neg t \Rightarrow \mathsf{F}$ [`DISCH` 3]
5. $\Gamma \ \vdash \ (t \Rightarrow \mathsf{F}) \Rightarrow \mathsf{F}$ [`SUBST` 2,4]
6. $t = \mathsf{F} \ \vdash \ t = \mathsf{F}$ [`ASSUME`]
7. $\Gamma,\ t = \mathsf{F} \ \vdash \ (\mathsf{F} \Rightarrow \mathsf{F}) \Rightarrow \mathsf{F}$ [`SUBST` 6,5]
8. $\mathsf{F} \ \vdash \ \mathsf{F}$ [`ASSUME`]
9. $\vdash \ \mathsf{F} \Rightarrow \mathsf{F}$ [`DISCH` 8]
10. $\Gamma,\ t = \mathsf{F} \ \vdash \ \mathsf{F}$ [`MP` 7,9]
11. $\vdash \ \mathsf{F} = \forall b.\ b$ [Definition of F]
12. $\Gamma,\ t = \mathsf{F} \ \vdash \ \forall b.\ b$ [`SUBST` 11,10]
13. $\Gamma,\ t = \mathsf{F} \ \vdash \ t$ [`SPEC` 12]
14. $\vdash \ \forall b.\ (b = \mathsf{T}) \ \vee \ (b = \mathsf{F})$ [Axiom]
15. $\vdash \ (t = \mathsf{T}) \ \vee \ (t = \mathsf{F})$ [`SPEC` 14]
16. $t = \mathsf{T} \ \vdash \ t = \mathsf{T}$ [`ASSUME`]
17. $t = \mathsf{T} \ \vdash \ t$ [`EQT_ELIM` 16]
18. $\Gamma \ \vdash \ t$ [`DISJ_CASES` 15,17,13]

Chapter 9

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`:

```
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.

```
#BETA_CONV;;                                                              1
- : conv

#BETA_CONV "(\x. (\y. (\z. x + y + z)3)2) 1";;
|- (\x. (\y. (\z. x + (y + z))3)2)1 = (\y. (\z. 1 + (y + z))3)2

#BETA_CONV "(\y. (\z. 1 + (y + z))3) 2";;
|- (\y. (\z. 1 + (y + z))3)2 = (\z. 1 + (2 + z))3

#BETA_CONV "(\z. 1 + (2 + z)) 3";;
|- (\z. 1 + (2 + z))3 = 1 + (2 + 3)

#BETA_CONV "1 + (2 + 3)";;
evaluation failed     BETA_CONV
```

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

[1]A list of the axiomatic rules was supplied in Section 8.3.

ones. Some of these are described in Sections 9.1 and 9.2. The notion of conversions is inherited from Cambridge LCF; the underlying principles are described in [**?**, **?**].

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

```
#NOT_FORALL_CONV "~!x. (f:*->*) x = g x";;                          1
|- ~(!x. f x = g x) = (?x. ~(f x = g x))

#CONTRAPOS_CONV "(!x. f x = g x) ==> ((f:*->*) = g)";;
|- (!x. f x = g x) ==> (f = g) = ~(f = g) ==> ~(!x. f x = g x)

#SELECT_CONV "(@f:*->*. f x = g x)x = g x";;
|- ((@f. f x = g x)x = g x) = (?f. f x = g x)

#EXISTS_UNIQUE_CONV "?!z. (f:*->*) z = g z";;
|- (?! z. f z = g z) =
   (?z. f z = g z) /\ (!z z'. (f z = g z) /\ (f z' = g z') ==> (z = z'))
```

An example of an application specific conversion is `num_CONV`:

```
#num_CONV "2";;                                                     1
|- 2 = SUC 1

#num_CONV "1";;
|- 1 = SUC 0

#num_CONV "0";;
evaluation failed     num_CONV: argument less than 1
```

Another application of conversions, related to the first, is in the implementation of the existing rewriting tools, `REWRITE_CONV` (Section 9.4), `REWRITE_RULE` (Section 8.2) and

[2]This was also mentioned in Section 8.3.

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

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

REWRITE_TAC (Chapter 10), which are central to theorem proving in HOL. This use is explained in Section 9.4, both as an example and because users may have occasion to construct rewriting tools of their own design, by similar methods. The next section introduces the conversion-building tools in general.

9.1 Conversion combining operators

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

```
$THENC : conv -> conv -> conv
```

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

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

```
$ORELSEC : conv -> conv -> conv
```

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

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

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

`REPEATC` c is intuitively equivalent to:

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

It is defined recursively:[5]

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

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

```
#REPEATC BETA_CONV;;                                                    1
- : conv

#REPEATC BETA_CONV "(\x. (\y. (\z. x + y + z)3)2)1";;
|- (\x. (\y. (\z. x + (y + z))3)2)1 = 1 + (2 + 3)
```

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

```
#BETA_CONV "(((\x.(\y.(\z. x + y + z))) 1) 2) 3";;                      1
evaluation failed     BETA_CONV

#is_abs "(((\x.(\y.(\z. x + y + z))) 1) 2)";;
false : bool
```

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

[5] Note that because ML is a call-by-value language, the extra argument t is needed in the definition of `REPEATC`; without it the definition would loop. There is a similar problem with the tactical `REPEAT`; see Chapter 10.

```
#BETA_CONV "(\z. x + y + z)3";;                                    1
|- (\z. x + (y + z))3 = x + (y + 3)

#BETA_CONV "(\w.w)3";;
|- (\w. w)3 = 3

#REPEATC BETA_CONV "(\z. x + y + z)((\w.w)3)";;
|- (\z. x + (y + z))((\w. w)3) = x + (y + ((\w. w)3))
```

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

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

`DEPTH_CONV` c is a conversion

$t \quad \mapsto \quad$ `|-` 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";;       1
|- (\x y z. x + (y + z))1 2 3 = 1 + (2 + 3)

#DEPTH_CONV BETA_CONV "(\z. x + y + z)((\w.w)3)";;
|- (\z. x + (y + z))((\w. w)3) = x + (y + 3)
```

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

```
#let t = "(\f.\x.f x)(\n.n+1)";;                                   1
t = "(\f x. f x)(\n. n + 1)" : term

#DEPTH_CONV BETA_CONV t;;
|- (\f x. f x)(\n. n + 1) = (\x. (\n. n + 1)x)
```

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

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

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

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

Thus:

```
#TOP_DEPTH_CONV BETA_CONV t;;                                  2
|- (\f x. f x)(\n. n + 1) = (\x. x + 1)
```

Finally, the simpler function `ONCE_DEPTH_CONV` is provided:

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

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

```
#ONCE_DEPTH_CONV BETA_CONV t;;                                 3
|- (\f x. f x)(\n. n + 1) = (\x. (\n. n + 1)x)

#ONCE_DEPTH_CONV BETA_CONV "(\x. (\n. n + 1)x)";;
|- (\x. (\n. n + 1)x) = (\x. x + 1)
```

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

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

`CONV_RULE` c (`|-` t) returns `|-` t', where c t evaluates to the equation `|-` t=t'. `CONV_TAC` c is a tactic that converts the conclusion of a goal using c. `CONV_RULE` is defined by:

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

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

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

It is defined by:

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

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

[8]For `EQ_MP`, see 8.3.7.

```
#BETA_RULE(ASSUME "(((\x.(\y.(\z. x + y + z))) 1) 2) 3 < 10");;           4
. |- (1 + (2 + 3)) < 10

#let th = ASSUME "NEXT = ^t";;
th = . |- NEXT = (\f x. f x)(\n. n + 1)

#BETA_RULE th;;
. |- NEXT = (\x. (\n. n + 1)x)

#BETA_RULE(BETA_RULE th);;
. |- NEXT = (\x. x + 1)
```

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

```
#let TOP_BETA_RULE = CONV_RULE(TOP_DEPTH_CONV BETA_CONV);;                 5
TOP_BETA_RULE = - : (thm -> thm)

#TOP_BETA_RULE th;;
. |- NEXT = (\x. x + 1)
```

TOP_DEPTH_CONV is the traversal strategy used by the HOL rewriting tools described in Section 9.4.

9.2 Writing compound conversions

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

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

```
NO_CONV : conv
```

NO_CONV t always fails.

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

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

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

c_1 ORELSEC c_2 ORELSEC . . . ORELSEC c_n

It is defined by:

```
let FIRST_CONV cl t =
    itlist $ORELSEC cl NO_CONV t ? failwith `FIRST_CONV`;;
```

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

```
ALL_CONV : conv
```

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

The function EVERY_CONV applies a list of conversions in sequence.

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

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

c_1 THENC c_2 THENC ... THENC c_n

It is defined by:

```
let EVERY_CONV cl t =
    itlist $THENC cl ALL_CONV t ? failwith `EVERY_CONV`
```

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

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

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

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

```
TRY_CONV : conv
```

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

```
let TRY_CONV c = c ORELSEC ALL_CONV
```

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

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

```
MK_COMB : thm # thm -> thm
MK_ABS  : thm -> thm
```

`MK_COMB` and `MK_ABS` implement the following derived rules:

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

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

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

```
SUB_CONV : conv
```

In particular:

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

`SUB_CONV` is implemented in terms of `MK_COMB` and `MK_ABS`:

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

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

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

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

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

```
let RATOR_CONV c t =
 (let rator,rand = dest_comb t in
  MK_COMB (c rator, REFL rand)) ? failwith `RATOR_CONV`

let ABS_CONV c t =
 (let bv,body = dest_abs t in
  let bodyth = c body in
  MK_ABS (GEN bv bodyth)) ? failwith `ABS_CONV`
```

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

```
#let t = "(\x.x+1)m + (\x.x+2)n";;                                      1
t = "((\x. x + 1)m) + ((\x. x + 2)n)" : term

#RAND_CONV BETA_CONV t;;
|- ((\x. x + 1)m) + ((\x. x + 2)n) = ((\x. x + 1)m) + (n + 2)

#RATOR_CONV (RAND_CONV BETA_CONV) t;;
|- ((\x. x + 1)m) + ((\x. x + 2)n) = (m + 1) + ((\x. x + 2)n)
```

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

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

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

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

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

9.3 Built in conversions

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

9.3.1 Generalized beta-reduction

The conversion:

```
PAIRED_BETA_CONV : conv
```

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

Given the term:

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

`PAIRED_BETA_CONV` proves that:

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

The conversion works for arbitrarily nested tuples. For example:

```
#PAIRED_BETA_CONV "(\((a,b),(c,d)). [a;b;c;d]) ((1,2),(3,4))";;
|- (\((a,b),c,d). [a;b;c;d])((1,2),3,4) = [1;2;3;4]
```

9.3.2 Arithmetical conversions

The conversion:

```
ADD_CONV : conv
```

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

```
#ADD_CONV "1 + 2";;
|- 1 + 2 = 3

#ADD_CONV "0 + 1000";;
|- 0 + 1000 = 1000

#ADD_CONV "101 + 102";;
|- 101 + 102 = 203
```

The next conversion decides the equality of natural number constants.

```
num_EQ_CONV : conv
```

If n and m are terms constructed from numeral constants and the successor function `SUC`, then: `num_EQ_CONV` "n=m" returns:

```
|- (n=m) = T   if n and m represent the same number
|- (n=m) = F   if n and m represent different numbers
```

In addition, `num_EQ_CONV` "t = t" returns: `|-` (t=t) `= T`

9.3.3 List processing conversions

There are two useful built-in conversions for lists:

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

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

LENGTH_CONV "LENGTH[t_1;...;t_n]"

generates the theorem:

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

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

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

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

(i) ~(n=m) or

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

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

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

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

9.3.4 Simplifying `let`-terms

A conversion for reducing `let`-terms is now provided.

```
let_CONV : conv
```

Given a term:

"let v_1 = t_1 and $\cdots$ and v_n = t_n in $t[v_1, \ldots, v_n]$"

`let_CONV` proves that:

|- let v_1 = t_1 and $\cdots$ and v_n = t_n in $t[v_1, \ldots, v_n]$ = $t[t_1, \ldots, t_n]$

The v_i's can take any one of the following forms:

(i) Variables: `x` etc.

(ii) Tuples: `(x,y)`, `(a,b,c)`, `((a,b),(c,d))` etc.

(iii) Applications: `f (x,y) z`, `f x` etc.

Variables are just substituted for. With tuples, the substitution is done component-wise, and function applications are effectively rewritten in the body of the `let`-term.

```
#let_CONV "let x = 1 in x+y";;                                                1
|- (let x = 1 in x + y) = 1 + y

#let_CONV "let (x,y) = (1,2) in x+y";;
|- (let (x,y) = 1,2 in x + y) = 1 + 2

#let_CONV "let f x = 1 and f y = 2 in (f 10) + (f 20)";;
|- (let f x = 1 and f y = 2 in (f 10) + (f 20)) = 2 + 2

#let_CONV "let f x = x + 1 and g x = x + 2 in f(g(f(g 0)))";;
|- (let f x = x + 1 and g x = x + 2 in f(g(f(g 0)))) =
(((0 + 2) + 1) + 2) + 1

#CONV_RULE(DEPTH_CONV ADD_CONV)it;;
|- (let f x = x + 1 and g x = x + 2 in f(g(f(g 0)))) = 6

#let_CONV "let f x y = x+y in f 1";;  % NB: partial application %
|- (let f x y = x + y in f 1) = (\y. 1 + y)
```

9.3.5 Skolemization

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

The conversion

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

takes a variable parameter, f say, and proves:

$$\texttt{|- (!}x_1\ \ldots\ x_n\texttt{. ?}y\texttt{. }t[x_1,\ldots,x_n,y]\texttt{)} \;=\; \texttt{(?}f\texttt{. !}x_1\ \ldots\ x_n\texttt{. }t[x_1,\ldots,x_n,f\ x_1\ldots\ x_n]$$

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

$$\texttt{|- (?}y\texttt{. }t[y]\texttt{) = (?}f\texttt{. }t[f]\texttt{)}$$

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

```
X_SKOLEM_CONV "f:num->*" "!n:num. ?x:*. x = (f n)"
```

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

```
SKOLEM_CONV "!x. ?y. P x y"
```

proves that:

```
|- ?y. !x. P x (y x)
```

9.3.6 Quantifier movement conversions

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

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

The conversions for moving quantifiers inwards are called:

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

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

The conversions for moving quantifiers outwards are called:

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

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

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

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

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


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


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


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


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


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


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


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


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


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


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

9.4 Rewriting tools

The rewriting tool `REWRITE_RULE` was introduced in Chapter 8. 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";;                                                    1
Theorem ADD1 autoloaded from theory `arithmetic`.
ADD1 = |- !m. SUC m = m + 1

|- SUC 0 = 0 + 1
```

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

`DEPTH_CONV(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)";;                     2
|- (SUC(SUC 0) = PRE(SUC 2)) = ((0 + 1) + 1 = PRE(2 + 1))
```

In itself, this is not a very useful rewriting tool, but a collection of others have been developed for use in HOL. All of the rewriting 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 10, 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 10

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 8, 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.

10.1 Tactics, goals and justifications

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

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

A justification is (rather confusingly) called a *proof* in HOL following the LCF usage; it is, as mentioned, an ML function from a theorem list to a theorem. The ML 'proof' function corresponds to a proof in the logical sense (of a sequence of theorems depending on inference rules) only in that it must evaluate the ML function corresponding to each inference rule on which the sequence depends in order to compute its `thm`-valued result. ('Justification', or 'validation', as is sometimes used, are less confusing terms for the ML function in question.) The proof function, or justification, returned by a tactic is intended to map the list of theorems respectively achieving the subgoals to the theorem achieving the original goal; it justifies the decomposition into subgoals.

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

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

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.[2]. The operations on goals are therefore just the ordinary pair selectors and constructor. Likewise, type abbreviations are introduced for justifications (proofs) and tactics. Conceptually, the following abbreviations are made in HOL:

```
goal       = term list # term
tactic     = goal -> goal list # proof
proof      = thm list -> thm
```

In fact, the type `goal list # proof` is abbreviated in ML to `subgoals`, and the abbreviation of `tactic` made indirectly through it. Thus, if T is a tactic and g is a goal, then applying T to g (i.e. evaluating the ML expression $T\ g$) results in an ML value of type `subgoals`, i.e. a pair whose first component is a list of goals and whose second component has ML type `proof`. (The word 'tactic' is occasionally used loosely to mean a tactic-valued function.)

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

Tactics can be perfectly useful without being strongly valid, or without even being valid; in fact, some of the most basic theorem proving strategies, expressed as tactics, are invalid or not strongly valid.[3] An invalid tactic cannot result in the proof of false theorems; theorems in HOL are always the result of performing a proof in the basic logic, whether the proof is found by goal directed search or forward search.[4] However, an invalid tactic may produce an unintended theorem—one that does not achieve the original goal. The typical case is when a theorem purporting to achieve a goal actually depends on hypotheses that extend beyond the assumptions of the goal. The inconvenience to the HOL user in this case is that the problem may be not immediately obvious;

[2] However, if goals were an abstract type, the print abbreviation could be avoided where not intended.

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

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

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`. It is described by:

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

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

Another example is `INDUCT_TAC`, the tactic for performing mathematical induction on the natural numbers:

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

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

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

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

is the identity tactic; it reduces a goal (Γ, t) to the single subgoal (Γ, t)—i.e. it has no effect. `ALL_TAC` is useful for writing compound tactics, as discussed later (see Section 10.4).

In just the way that the derived rule REWRITE_RULE is central to forward proof (Section 8.2), the corresponding function REWRITE_TAC is central to goal directed proof. Given a goal and a list of equational theorems, REWRITE_TAC transforms the term component of the goal by applying the equations as left-to-right rewrites, recursively and to all depths, until no more changes can be made. Unless not required, the function includes as rewrites the same standard set of pre-proved tautologies that REWRITE_RULE uses. By use of the tautologies, some subgoals can be solved internally by rewriting, and in that case, an empty list of subgoals is returned. The transformation of the goal is justified in each case by the appropriate chain of inferences. Rewriting often does a large share of the work in goal directed proof searches.

A simple example from list theory (Section 4.9) illustrates the use of tactics. A conjunctive goal is declared, and CONJ_TAC applied to it:

```
#let g = ([]:term list),"(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])";;     1
g = ([], "(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])") : goal

#let gl1,p1 = CONJ_TAC g;;
gl1 = [([], "HD[1;2;3] = 1"); ([], "TL[1;2;3] = [2;3]")] : goal list
p1 = - : proof
```

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

```
#HD;;                                                                  2
Definition HD autoloaded from theory `list`.
HD = |- !h t. HD(CONS h t) = h

|- !h t. HD(CONS h t) = h

#TL;;
Definition TL autoloaded from theory `list`.
TL = |- !h t. TL(CONS h t) = t

|- !h t. TL(CONS h t) = t

#let gl1_1,p1_1 = REWRITE_TAC[HD;TL](hd gl1);;
gl1_1 = [] : goal list
p1_1 = - : proof

#let gl1_2,p1_2 = REWRITE_TAC[HD;TL](hd(tl gl1));;
gl1_2 = [] : goal list
p1_2 = - : proof
```

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

```
#let th1 = p1_1[];;                                                          3
th1 = |- HD[1;2;3] = 1

#let th2 = p1_2[];;
th2 = |- TL[1;2;3] = [2;3]

#p1[th1;th2];;
|- (HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])
```

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

The HOL system provides a collection of pre-defined tactics (and `tactic`-valued functions) that includes `CONJ_TAC`, `INDUCT_TAC`, `ALL_TAC` and `REWRITE_TAC`. The pre-defined tactics are adequate for many applications. In addition, there are two means of defining new tactics. Since a tactic is an ML function, the user can define a new tactic directly in ML. Definitions of this sort use ML functions to construct the term part of the subgoals from the term part of the original goal (if any transformation is required); and they specify the justification, which expects a list of theorems achieving the subgoals and returns the theorem achieving (one hopes) the goal. The proof of the theorem is encoded in the definition of the justification function; that is, the means for deriving the desired theorem from the theorems given. This typically involves references to axioms and primitive and defined inference rules, and is usually the more difficult part of the project.

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

```
let CONJ_TAC : tactic (asl,w) =
(let l,r = dest_conj w in
           [(asl,l);(asl,r)],(\[th1;th2].CONJ th1 th2)
) ? failwith `CONJ_TAC`;;
```

This shows how the subgoals are constructed, and how the proof function is specified in terms of the derived rule `CONJ` (Section 8.3.23).

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 10.4. For example, two existing tactics can be sequenced by use of the tactical `THEN`: if T_1 and T_2 are tactics, then the ML expression T_1 `THEN` T_2 evaluates to a tactic that first applies T_1 to a goal and then applies T_2 to each subgoal produced by T_1. The tactical `THEN` is an infixed ML function. Complex and powerful tactics can be constructed in this way; and new tacticals can also be defined, although this is unusual.

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

```
#let gl2,p2 = (CONJ_TAC THEN REWRITE_TAC[HD;TL])g;;                    4
gl2 = [] : goal list
p2 = - : proof

#p2[];;
|- (HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])
```

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

In general, the second method is both easier and more reliable. It is easier because it does not involve writing ML procedures (usually rather complicated procedures); and more reliable because the composed tactics are valid when the constituent tactics are valid, as a consequence of the way the tacticals are defined. Tactics written directly in ML may fail in a variety of ways, and although, as usual, they cannot cause false theorems to appear, the failures can be difficult to understand and trace.[5] On the other hand, there are some proof strategies that cannot be implemented as compositions of existing tactics, and these have to be implemented directly in ML. Certain sorts of inductions are an example of this; as well as tactics to support some personal styles of proof.

10.1.1 Details of proving theorems

When a theorem is proved that the user wishes to preserve for future use, it can be stored in the current theory by using the function `save_thm` (see Section 3.7.1).

To simplify the use of tactics there are three standard functions

```
   TAC_PROOF : (goal # tactic) -> thm
   prove_thm : (string # term # tactic) -> thm
   PROVE     : (term # tactic) -> thm
```

`TAC_PROOF` takes a goal and a tactic, and applies the tactic to the goal; the goal can have assumptions. Executing `prove_thm(‘foo‘,`t`,`T`)` proves the goal `([],`t`)` (i.e. the goal with no assumptions and conclusion t) using tactic T and saves the resulting theorem with name `foo` on the current theory. Executing `PROVE(`t`,`T`)` proves the goal `([],`t`)` using T and returns the result without saving it. In all cases the evaluation fails if T does not solve the goal `([],`t`)`.

In short, HOL provides a very general framework in which proof strategies can be designed, implemented, applied and tested. Tactics range from the very simple to the very advanced; in theory, a conventional automatic theorem prover could be expressed as a tactic or group of tactics. In contrast, some users never have need to go beyond

[5] A possible extension to HOL would be a ‘debugging environment’ for this class of tactic.

the built in tactics of the system. The vital support that HOL provides in all cases is the assurance that only theorems of the deductive system can be represented as theorems of the HOL system—security is always preserved.

10.2 The subgoal package

It was mentioned earlier that goal directed proof is a way of organizing the construction of a proof as a tree search. For any tactic and goal, the tactic implicitly determines a tree of subgoals for that goal: each node is a subgoal, and each edge is a tactic. Associated with each node in a successful proof effort is also an achieving theorem—the theorem that achieves that subgoal. That is, the tree is traversed in two phases: from the root (the original goal) to the final layer of subgoals; and from the theorems achieving the final subgoals back to the theorem achieving the goal at the root. The first phase is the decomposition phase, in which goals are reduced to subgoals (and eventually to trivial subgoals). The second phase is the computation of the proof, through each primitive step, culminating in the desired theorem. The tree, however, is not explicitly represented in the HOL system, so each proof effort requires some amount of book-keeping: application of tactics to goals, naming of subgoals and proof functions, application of the appropriate proof functions to theorem lists, naming of theorems, and so on.

When conducting a proof that involves many subgoals and tactics, it is difficult to keep track of this book-keeping. While it is actually feasible for the user to take responsibility, even in large proofs, it is tedious and error-prone. Therefore HOL provides a package traversing the tree of subgoals once through, stacking the subgoals and proof functions, and applying the proof functions automatically, when appropriate to do so. This package was originally implemented for Cambridge LCF by Paulson.

The subgoal package implements a simple framework for interactive proof, and this is adequate for most users in most applications. The tree is traversed depth first. The current goal can be expanded into subgoals and a proof function by supplying it with a tactic; the subgoals are pushed onto a goal stack and the justifications onto a proof stack. Subgoals at the same depth in the tree can be considered in any order by rotating through them, but one otherwise has to work through the tree depth first. When a tactic solves a subgoal (i.e. returns an empty subgoal list), then the package computes a part of the proof, and presents the user with the next subgoal.

For many users, the subgoal package is the primary interface to HOL for proving theorems. As mentioned, it is very convenient to be relieved of all the book-keeping labour. However, there is some cost in that the subgoal-proof tree cannot be inspected; it only exists ephemerally for the user during an interaction, and can only be viewed at the current top subgoal of the stack. Achieving-theorems are only displayed at the moment they are proved; there is no naming or preserving of subgoals or justifications. If there is any reason to view other subgoals, this can only be accomplished by undoing segments of the proof effort (backing up). Likewise, intermediate achieving-theorems cannot be inspected after they have been displayed at proof time. One situation in which it is necessary to have the tree available is in the debugging of tactics written by the user directly in ML. It is planned in future versions of HOL to implement a more sophisticated subgoal management package.

Finally, the application of certain tactics to certain goals generates a failure in ML where the tactic is invalid; this does not happen when using HOL directly.

The example from earlier is continued below. In the session below, the conjunction proof is generated again, but using the subgoal package, in which a goal is 'set' using the function `set_goal` and 'expanded' using the function `expand`. The side effects of these functions on the subgoal package's stacks can be inferred.

5

```
#set_goal([],"(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])");;
"(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])"

() : void

#expand CONJ_TAC;;
OK..
2 subgoals
"TL[1;2;3] = [2;3]"

"HD[1;2;3] = 1"

() : void

#expand(REWRITE_TAC[HD;TL]);;
OK..
goal proved
|- HD[1;2;3] = 1

Previous subproof:
"TL[1;2;3] = [2;3]"

() : void

#expand(REWRITE_TAC[HD;TL]);;
OK..
goal proved
|- TL[1;2;3] = [2;3]
|- (HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])
```

The following functions are available for interacting with the subgoal package. The function

```
set_goal: goal -> void
```

initializes the subgoal package with a new goal.

The function

```
expand : tactic -> void
```

applies a tactic to the top goal on the stack, then pushes the resulting subgoals onto the goal stack and prints them. If there are no resulting subgoals (i.e. if the current goal was just solved), then the appropriate proof function is applied to the empty list of theorems and the resulting theorems are printed.

The function

```
backup : void -> void
```

allows backing up from last state-change. The assignable variable `backup_limit`, initially set to `12`, determines the maximum number of proof states saved on the backup list. The function `backup` can be repeated until the list is exhausted; backing up discards the current state irretrievably.

The function

```
rotate : int -> void
```

rotates the order of subgoals on the stack. Calling `rotate` on n rotates by n steps the set of subgoals on top of the stack. This enables subgoals at a given depth in the subgoal tree to be considered in any order. However, subgoals deeper in the stack cannot be worked on, nor can subgoals higher in the tree.

The function

```
top_goal : void -> goal
```

returns the top goal on the stack.

The function

```
top_thm : * -> thm
```

returns the theorem on top of the theorem stack. It is used to access the result of an interactive proof session with the subgoal package.

The function

```
save_top_thm : string -> thm
```

saves the top theorem on the goal stack in the current theory, and also returns it as a value. (It is generally used only to save the final theorem, rather than the intermediate theorems in the proof search.)

The function

```
get_state : void -> goalstack
```

returns the current proof state, which can then be assigned to a variable for additional backup.

The function

```
set_state : goalstack -> void
```

restores the proof state to that saved earlier using `get_state`.

The function

```
print_state : int -> void
```

applied to n, prints n levels of the goal stack.

The following abbreviations are pre-declared for use in the subgoal package:

```
let g t = set_goal([],t)
and e   = expand
and p   = print_state
and b   = backup
and r   = rotate;;
```

The flag `print_all_subgoals` affects all operations where the subgoal stack is printed. If the flag is `true`, the entire subgoal stack is printed. If the flag is `false`, only the top subgoal on the stack is printed. If only the current subgoal is to be printed, the subgoal package will report the number of subgoals remaining before displaying the subgoal on the top of the goal stack. The default value of this flag is `true`.

10.3 Some tactics built into HOL

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

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

10.3.1 Acceptance of a theorem

```
ACCEPT_TAC : thm_tactic
```

- **Summary:** `ACCEPT_TAC` th is a tactic that solves any goal that is achieved by th.

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

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

  would then solve g, where `THENL` is the tactical that applies the respective elements of the tactic list to the subgoals produced by `T` (see Section 10.4.5).

10.3.2 Adding an assumption

```
ASSUME_TAC : thm_tactic
```

- **Summary:** `ASSUME_TAC |-`u adds u as an assumption.

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

- **Use:** Enriching the assumptions of a goal with definitions or previously proved theorems.

10.3.3 Specialization

```
GEN_TAC : tactic
```

- **Summary:** Specializes a universally quantified theorem to an arbitrary value.

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

  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.

10.3.4 Conjunction

```
CONJ_TAC : tactic
```

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

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

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

10.3.5 Discharging an assumption

```
DISCH_TAC : tactic
```

- **Summary:** Moves the antecedant of an implicative goal into the assumptions, leaving the consequent as the term component.

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

- **Use:** Solving goals of the form u `==>` v by assuming u and then solving v under the assumption. `STRIP_TAC` (see below) invokes `DISCH_TAC` on implicative goals.

10.3.6 Combined simple decompositions

```
STRIP_TAC : tactic
```

- **Summary:** Breaks a goal apart. `STRIP_TAC` removes one outer connective from the goal, using `CONJ_TAC`, `DISCH_TAC`, `GEN_TAC`, and other tactics. If the goal has the form t_1`/\`$\cdots$`/\`t_n `==>` t then `DISCH_TAC` makes each t_i into a separate assumption.

- **Use:** Useful for splitting a goal up into manageable pieces. Often the best thing to do first is `REPEAT STRIP_TAC`, where `REPEAT` is the tactical that repeatedly applies a tactic until it fails (see Section 10.4.7).

10.3.7 Substitution

```
SUBST_TAC : thm list -> tactic
```

- **Summary:** `SUBST_TAC[|-`u_1`=`v_1`;`$\ldots$`;|-`u_n`=`v_n`]` changes each sub-term $t[u_1, \ldots, u_n]$ of the goal to $t[v_1, \ldots, v_n]$ by substitution.

- **Use:** Useful in situations where `REWRITE_TAC` does too much, or would loop.

10.3.8 Case analysis on a boolean term

```
ASM_CASES_TAC : term -> tactic
```

- **Summary:** `ASM_CASES_TAC` u , where u is a boolean-valued term, does case analysis on u.

$$\frac{t}{\overline{\{u\}t \quad \{\tilde{}u\}t}}$$

- **Use:** Case analysis.

10.3.9 Case analysis on a disjunction

```
DISJ_CASES_TAC : thm_tactic
```

- **Summary:** `DISJ_CASES_TAC |-` u `\/` v splits a goal into two cases: one with u as an assumption and the other with v as an assumption.

$$\frac{t}{\overline{\{u\}t \quad \{v\}t}}$$

- **Use:** Case analysis. The tactic `ASM_CASES_TAC` is defined in ML by

```
let ASM_CASES_TAC t = DISJ_CASES_TAC(SPEC t EXCLUDED_MIDDLE)
```

  where `EXCLUDED_MIDDLE` is the theorem `|- !t. t \/ ~t`.

10.3.10 Rewriting

```
REWRITE_TAC : thm list -> tactic
```

- **Summary:** `REWRITE_TAC[`th_1`;...;`th_n`]` transforms the term part of a goal by rewriting it with the given theorems th_1, ..., th_n, and the set of pre-proved standard tautologies.

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

  where t' is obtained from t as described.

- **Use:** Advancing goals by using definitions and previously proved theorems (lemmas).

- **Some other rewriting tactics** (based on `REWRITE_TAC`) are:

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

10.3.11 Resolution by Modus Ponens

```
IMP_RES_TAC : thm -> tactic
```

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

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

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

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

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

`|-` v_{k+1} `==>` $\ldots$ `==>` v_q `==>` v

is added to the assumptions.

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

- **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.

10.3.12 Identity

```
ALL_TAC : tactic
```

- **Summary:** The identity tactic for the tactical `THEN` (see Section 10.1). Useful for writing tactics.
- **Use:**
  1. Writing tacticals (see description of `REPEAT` in Section 10.4).
  2. With `THENL` (see Section 10.4.5); for example, if tactic T produces two subgoals T_1 is to be applied to the first while nothing is to be done to the second, then T `THENL[`T_1`;ALL_TAC]` is the tactic required.

10.3.13 Null

```
NO_TAC : tactic
```

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

10.3.14 Splitting logical equivalences

```
EQ_TAC : tactic
```

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

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

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

10.3.15 Solving existential goals

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

- **Summary:** `EXISTS_TAC "`u`"` reduces an existential goal `!`x`.` $t[x]$ to the subgoal $t[u]$.

$$\frac{\texttt{!}x\,.\,t[x]}{\overline{t[u]}}$$

- **Use:** Proving existential goals.

- **Comment:** `EXISTS_TAC` is a crude way of solving existential goals, but it is the only built-in tactic for this purpose. A more powerful approach uses Prolog-style 'logic variables' (i.e. meta-variables) that can be progressively refined towards the eventual witness. Implementing this requires goals to contain an environment giving the binding of logic variables to terms. Details (in the context of LCF) are given in a paper by Stefan Sokołowski [**?**].

10.4 Tacticals

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

10.4.1 Alternation

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

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

```
(T1 ORELSE T2) g =  T1 g ? T2 g
```

10.4.2 First success

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

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

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

10.4.3 Change detection

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

`CHANGED_TAC` T g fails if the subgoals produced by T are just `[`g`]`; otherwise it is equivalent to T g. It is defined by the following, where `set_equal : * list -> * list -> bool` tests whether two lists denote the same set (i.e. contain the same elements).

```
letrec CHANGED_TAC tac g =
 let gl,p = tac g in
 if set_equal gl [g] then fail else (gl,p)
```

10.4.4 Sequencing

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

The tactical `THEN` is an ML infix. If T_1 and T_2 are tactics, then the ML expression T_1 `THEN` T_2 evaluates to a tactic which first applies T_1 and then applies T_2 to each subgoal produced by T_1. Its definition in ML is complex (and due to Milner) but worth understanding as an exercise in ML. It is an ML curried infix.

```
let ((T1:tactic) THEN (T2:tactic)) g =
 let gl,p = T1 g
 in
 let gll,pl = split(map T2 gl)
 in
 (flat gll, (p o mapshape(map length gll)pl));;
```

Here are the definitions of the ML functions `map`, `split`, `o`, `length`, `flat` and `mapshape`:

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

$\texttt{map}\ f\ \texttt{[}x_1\texttt{;}\ldots\texttt{;}x_n\texttt{]} \ \ \texttt{=}\ \ \texttt{[}f\ x_1\texttt{;}\ldots\texttt{;}f\ x_n\texttt{]}$

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

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

`$o : ((* -> **) # (*** -> *)) -> *** -> **$` (an infix)

$\texttt{(}f\ \texttt{o}\ g\texttt{)}\ x \ \ \texttt{=}\ \ f\texttt{(}g\ x\texttt{)}$

`length : * list -> int`

$\texttt{length[}x_1\texttt{;}\ldots\texttt{;}x_n\texttt{]} \ \ \texttt{=}\ \ \texttt{n}$

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

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

`mapshape : int list -> (* list -> **) list -> * list -> ** list`

$$\begin{array}{l}\texttt{mapshape}\\ \quad\texttt{[}m_1\texttt{;}\ldots\texttt{;}m_n\texttt{]}\\ \quad\texttt{[}f_1\texttt{;}\ldots\texttt{;}f_n\texttt{]}\\ \quad\texttt{[}x_{11}\texttt{;}\ldots\texttt{;}x_{1m_1}\texttt{;}x_{21}\texttt{;}\ldots\texttt{;}x_{2m_2}\texttt{;}\ \ldots\ \texttt{;}x_{n1}\texttt{;}\ldots\texttt{;}x_{nm_n}\texttt{]}\ \texttt{=}\\ \texttt{[}f_1\texttt{[}x_{11}\texttt{;}\ldots\texttt{;}x_{1m_1}\texttt{];}f_2\texttt{[}x_{21}\texttt{;}\ldots\texttt{;}x_{2m_2}\texttt{];}\ \ldots\ \texttt{;}f_n\texttt{[}x_{n1}\texttt{;}\ldots\texttt{;}x_{nm_n}\texttt{]]}\end{array}$$

Suppose $T_1\ g\ \texttt{=}\ \texttt{(}gl\texttt{,}p\texttt{)}$ where $gl\texttt{=[}g_1\texttt{;}\ldots\texttt{;}g_n\texttt{]}$. Suppose also that for i between 1 and n it is the case that $T_2\ g_i\ \texttt{=}\ \texttt{([}g_{i1}\texttt{;}\ldots\texttt{;}g_{im_i}\texttt{],}p_i\texttt{)}$. Then $\texttt{split(map}\ T_2\ gl\texttt{)}$ will evaluate to the pair $\texttt{(}gll\texttt{,}pl\texttt{)}$ of a subgoal list and a proof function, where

$$gll\ \texttt{=}\ \texttt{[[}g_{11}\texttt{;}\ldots\texttt{;}g_{1m_1}\texttt{];[}g_{21}\texttt{;}\ldots\texttt{;}g_{2m_2}\texttt{];}\ \ldots\ \texttt{;[}g_{n1}\texttt{;}\ldots\texttt{;}g_{nm_n}\texttt{]]}$$

and $pl\ \texttt{=}\ \texttt{[}p_1\texttt{;}\ldots\texttt{;}p_n\texttt{]}$. Note that

$$\texttt{map length}\ gll\ \texttt{=}\ \texttt{[}m_1\texttt{;}\ldots\texttt{;}m_n\texttt{]}$$

and that

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

Suppose now that, for i between 1 and n, the theorems th_{i1}, ..., th_{im_i} achieve the goals g_{i1}, ..., g_{im_i}, respectively. It will follow that if T_2 is valid then for i between 1 and n the result of applying p_i to the list of theorems $\texttt{[}th_{i1}\texttt{;}\ldots\texttt{;}th_{im_i}\texttt{]}$ will be a theorem, th_i say, which achieves g_i. Now if T_1 is valid then $p\texttt{[}th_1\texttt{;}\ldots\texttt{;}th_n\texttt{]}$ will evaluate to a theorem, th say, that achieves the goal g. Thus

$$\begin{array}{l}
p \\
\texttt{(mapshape} \\
\ \texttt{(map length}\ gll\texttt{)} \\
\ pl \\
\ \texttt{[}th_{11}\texttt{;}\ldots\texttt{;}th_{1m_1}\texttt{;}th_{21}\texttt{;}\ldots\texttt{;}th_{2m_2}\texttt{;}\ \ldots\ \texttt{;}th_{n1}\texttt{;}\ldots\texttt{;}th_{nm_n}\texttt{])}\ \texttt{=} \\
\\
p\texttt{([}p_1\texttt{[}th_{11}\texttt{;}\ldots\texttt{;}th_{1m_1}\texttt{];}p_2\texttt{[}th_{21}\texttt{;}\ldots\texttt{;}th_{2m_2}\texttt{];}\ \ldots\ \texttt{;}p_n\texttt{[}th_{n1}\texttt{;}\ldots\texttt{;}th_{nm_n}\texttt{]])}\ \texttt{=} \\
\\
p\texttt{([}th_1\texttt{;}\ldots\texttt{;}th_n\texttt{])}\ \texttt{=} \\
\\
th
\end{array}$$

This shows that p `o mapshape(map length` gll`)`pl is a function that, when applied to a list of theorems respectively achieving `flat` gll, returns a theorem (namely th) that achieves g.

10.4.5 Selective sequencing

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

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

Here is the definition of `THENL`:

```
let ((T:tactic) THENL (Tl:tactic list)) g =
 let gl,p = T g
 in
 let gll,pl = (split(map (\(T,g). T g) Tgl)
               where Tgl = combine(Tl,gl) ? failwith `THENL`)
 in
 (flat gll, (p o mapshape(map length gll)pl))
```

The understanding of this procedure is left as an exercise!

10.4.6 Successive application

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

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

EVERY [T_1;T_2;...;T_n] = T_1 THEN T_2 THEN ... THEN T_n

10.4.7 Repetition

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

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

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

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

10.5 Tactics for manipulating assumptions

There are in general two kinds of tactics in HOL: those that transform the conclusion of a goal without affecting the assumptions, and those that do (also or only) affect the assumptions. The various tactics that rewrite are typical of the first class; those that do 'resolution' belong to the second. Often, many of the steps of a proof in HOL are carried out 'behind the scenes' on the assumptions, by tactics of the second sort. A tactic that in some way changes the assumptions must also have a justification that 'knows how' to restore the corresponding hypotheses of the theorem achieving the subgoal. All of this is explicit, and can be examined by a user moving about the subgoal-proof tree.[6] Using these tactics in the most straightforward way, the assumptions at any point in a goal-directed proof, i.e. at any node in the subgoal tree, form an unordered record of every assumption made, but not yet dismissed, up to that point.

In practice, the straightforward use of assumption-changing tactics, with the tools currently provided in HOL, presents at least two difficulties. The first is that assumption sets can grow to an unwieldy size, the number and/or length of terms making them difficult to read. In addition, forward-search tactics such as resolution often add at least some assumptions that are never subsequently used, and these have to be carried along with the useful assumptions; the straightforward method provides no ready way of intercepting their arrival. Likewise, there is no straightforward way of discarding assumptions after they have been used and are merely adding to the clutter. Although perhaps

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

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 [**?**, **?**]; 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 10.5.2, gives a way of intercepting and manipulating results without them necessarily being added as assumptions in the first place. The two approaches can be combined in HOL interactions.

10.5.1 Theorem continuations with popping

The first proof style, that of popping assumptions from the assumption 'stack', is illustrated using its main tool: the tactical `POP_ASSUM`.[7]

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

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

[7] The type of `POP_ASSUM` is actually more general than the type shown here. The present format is used simply for readability.

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

ML functions such as f, with type `thm -> tactic`, abbreviated to `thm_tactic`, are called theorem continuations, suggesting the fact that they take theorems and then continue the proof.[8] The use of `POP_ASSUM` can be illustrated by applying it to a particular tactic, namely `DISCH_TAC` (Section 10.3.5).

```
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 $(n = 0) \Rightarrow (n \times n = n)$:

1

```
#g "(n = 0) ==> (n * n = n)";;
"(n = 0) ==> (n * n = n)"

() : void

#e DISCH_TAC;;
OK..
"n * n = n"
    [ "n = 0" ]
```

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

```
SUBST1_TAC : thm_tactic
```

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

2

```
#top_goal();;
(["n = 0"], "n * n = n") : goal

#e(POP_ASSUM SUBST1_TAC);;
OK..
"0 * 0 = 0"
```

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

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

```
#g "(n = 0) ==> (n * n = n)";;                                        1
"(n = 0) ==> (n * n = 0)"

() : void

#e(DISCH_TAC THEN POP_ASSUM SUBST1_TAC);;
OK..
"0 * 0 = 0"
```

The goal can now be solved by rewriting with a fact of arithmetic:

```
#e(REWRITE_TAC[MULT_CLAUSES]);;                                       2
Theorem MULT_CLAUSES autoloaded from theory `arithmetic`.
MULT_CLAUSES =
|- !m n.
    (0 * m = 0) /\
    (m * 0 = 0) /\
    (1 * m = m) /\
    (m * 1 = m) /\
    ((SUC m) * n = (m * n) + n) /\
    (m * (SUC n) = m + (m * n))

OK..
goal proved
|- 0 * 0 = 0
|- (n = 0) ==> (n * n = n)
```

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

```
#g "(n = 0) ==> (n * n = n)";;                                        1
"(n = 0) ==> (n * n = n)"

() : void

#e(DISCH_TAC THEN POP_ASSUM SUBST1_TAC THEN REWRITE_TAC[MULT_CLAUSES]);;
Theorem MULT_CLAUSES autoloaded from theory `arithmetic`.
MULT_CLAUSES =
|- !m n.
    (0 * m = 0) /\
    (m * 0 = 0) /\
    (1 * m = m) /\
    (m * 1 = m) /\
    ((SUC m) * n = (m * n) + n) /\
    (m * (SUC n) = m + (m * n))

OK..
goal proved
|- (n = 0) ==> (n * n = n)
```

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

```
#g "(n = 0) ==> (n * n = n)";;                                          1
"(n = 0) ==> (n * n = n)"

() : void

#e(DISCH_TAC);;
OK..
"n * n = n"
    [ "n = 0" ]

() : void

#e(SUBST1_TAC(ASSUME "n = 0"));;
OK..
"0 * 0 = 0"
    [ "n = 0" ]
```

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

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

DISJ_CASES_TAC |- p \/ q

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

```
#g "((n = 0) \/ (n = 1)) ==> (n * n = n)";;                             1
"(n = 0) \/ (n = 1) ==> (n * n = n)"

() : void

#e DISCH_TAC;;
OK..
"n * n = n"
    [ "(n = 0) \/ (n = 1)" ]

() : void

#backup();;
"(n = 0) \/ (n = 1) ==> (n * n = n)"
```

```
#e(DISCH_TAC THEN POP_ASSUM DISJ_CASES_TAC);;                                2
OK..
2 subgoals
"n * n = n"
    [ "n = 1" ]

"n * n = n"
    [ "n = 0" ]

() : void

#backup();;
"(n = 0) \/ (n = 1) ==> (n * n = n)"

() : void

#e(DISCH_TAC THEN POP_ASSUM DISJ_CASES_TAC THEN POP_ASSUM SUBST1_TAC);;
OK..
2 subgoals
"1 * 1 = 1"

"0 * 0 = 0"
```

As noted earlier, `POP_ASSUM` is useful when an assumption is required that is still at the top of the stack, as in the examples. However, it is often necessary to access assumptions made at arbitrary previous times, in order to give them as parameters, combine them, etc. The stack approach can be extended to such cases by re-conceiving the stack as an array, and by use of the tactical `ASSUM_LIST`:

```
ASSUM_LIST : (thm list -> tactic ) -> tactic
```

where

`ASSUM_LIST` f `([`t_1`;...;`t_n`],`t`)` = f`([ASSUME` t_1`;...;ASSUME` t_n`])`

That is, given a function f, `ASSUM_LIST` f forms a new tactic by applying f to the list of (assumed) assumptions of a goal, then applies the resulting tactic to the goal. For example, a tactic of the form `ASSUM_LIST (\thl.` f `(el` i `thl))` applies the function f to the ith assumption of a goal to produce a new tactic, then applies the new tactic to the goal. Again, `ASSUM_LIST REWRITE_TAC` is a tactic that engages all of the current assumptions as rewrite rules. In this way, the array approach enables arbitrary assumptions to be accessed; and in particular, specific assumptions to be accessed by location using the function `el`.

To illustrate the use of `ASSUM_LIST`, suppose it were required to prove something different: that $(\forall m.\ m + n = m) \Rightarrow (n \times n = n)$. Suppose also that the arithmetic fact

ADD_INV_0 is already known: namely, that $\forall m\ n.\ (m + n = m) \Rightarrow (n = 0)$. After discharging the assumption, the conclusion of the theorem ADD_INV_0 is imported as an assumption, occupying first place in the array.

```
#g "(!m. m + n = m) ==> (n * n = n)";;                              1
"(!m. m + n = m) ==> (n * n = n)"

() : void

#e(DISCH_TAC);;
OK..
"n * n = n"
    [ "!m. m + n = m" ]

() : void

#e(ASSUME_TAC ADD_INV_0);;
Theorem ADD_INV_0 autoloaded from theory `arithmetic`.
ADD_INV_0 = |- !m n. (m + n = m) ==> (n = 0)

OK..
"n * n = n"
    [ "!m. m + n = m" ]
    [ "!m n. (m + n = m) ==> (n = 0)" ]
```

The problem is now to combine the two assumptions to produce the obvious conclusion. That requires denoting them, for which ASSUM_LIST provides the means. Finally, ASSUME_TAC places the conclusion of the new result in the assumptions. (The ML function `el: int -> * list -> *` is used here to select a numbered element of a list.)

```
#e(ASSUM_LIST(\thl. ASSUME_TAC                                      2
                       (MP (SPECL ["m:num";"n:num"] (el 1 thl))
                           (SPEC "m:num"(el 2 thl)))));;
##OK..
"n * n = n"
    [ "!m. m + n = m" ]
    [ "!m n. (m + n = m) ==> (n = 0)" ]
    [ "n = 0" ]
```

The goal can now be solved as in the previous example.

To access the two particular assumptions in the straightforward way would again require quoting their text. To access all of them (to pass to REWRITE_TAC, for instance) would require quoting all of them.

ASSUM_LIST addresses the issue of accessing assumptions, but not the issue of discarding them. A related function generalizes POP_ASSUM to discard them as well:

```
POP_ASSUM_LIST : (thm list -> tactic ) -> tactic
```

`POP_ASSUM_LIST` resembles `ASSUM_LIST` except in removing all of the old assumptions of the subgoal, the way that `POP_ASSUM` removes the most recent. (Thus `POP_ASSUM` is no more than a special case of `POP_ASSUM_LIST` that selects the first element of those supplied and re-assumes the others.)

`POP_ASSUM_LIST` f `([`t_1`; ... ;`t_n`],`t`) =` f `[ASSUME` t_1`; ... ;ASSUME` t_n`] ([],t)`

This is used when the existing assumptions have served their purpose and can be discarded, as in the current example:

```
#backup();;                                                          3
"n * n = n"
    [ "!m. m + n = m" ]
    [ "!m n. (m + n = m) ==> (n = 0)" ]

() : void

#e(POP_ASSUM_LIST(\thl. ASSUME_TAC
                           (MP (SPECL ["m:num";"n:num"] (el 1 thl))
                               (SPEC "m:num"(el 2 thl)))));;
##OK..
"n * n = n"
    [ "n = 0" ]
```

This leaves only the one assumption vital to solving the goal, as before. In some contexts, the new result is required as an assumption, but here it can be used immediately:

```
#backup();;                                                          4
"n * n = n"
    [ "!m. m + n = m" ]
    [ "!m n. (m + n = m) ==> (n = 0)" ]

() : void

#e(POP_ASSUM_LIST(\thl. SUBST1_TAC
                           (MP (SPECL ["m:num";"n:num"] (el 1 thl))
                               (SPEC "m:num"(el 2 thl)))));;
##OK..
"0 * 0 = 0"
```

`POP_ASSUM_LIST` can, of course, take any function of appropriate type, but is in fact often used in conjunction with the element-selecting functions. Function composition occasionally allows a more compact expression to be written.

The array view (of which the stack view is a special case) gives a way in which unnecessary assumptions can be dropped, and assumptions can be accessed, individually if necessary, using tacticals. Although this approach can be effective, as illustrated, it does tend to rely on the ordering of the representation of the assumption set. (That is, `POP_ASSUM` necessarily does, while the other two provide the temptation!) A minor drawback of this reliance is that tactics are then sensitive to changes that alter the order or composition of the assumptions; for example, changes in the implementation of HOL, modifications of existing tactics, and so on. However, that sensitivity is not so serious in any one incarnation of HOL; there is a logical viewpoint that regards the assumptions (sequents) as ordered anyway. A more serious problem is that order-sensitive tactics are meaningful only during interactive sessions; to reconstruct the assumptions from the ML text and the original goal alone is generally difficult, and more so when assumptions are denoted by location. This means that (i) the resulting tactics cannot easily be generalized for use in other contexts, and (ii) the ML text does not supply useful documentation of the solution of the goal. Also, as shown in the last example, it it slightly unsatisfactory to push and subsequently pop assumptions, especially in immediate succession, where this could be avoided.

Two other tacticals that can be used to manipulate the assumption list are `FIRST_ASSUM` and `EVERY_ASSUM`. These are characterized by:

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

EVERY_ASSUM f ([t1; ... ;tn], t)  =
 (f(ASSUME t1) THEN ... THEN  f(ASSUME tn)) ([t1; ... ;tn], t)
```

10.5.2 Theorem continuations without popping

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

```
#DISCH_THEN;;                                                          1
- : (thm_tactic -> tactic)

#DISCH_THEN SUBST1_TAC;;
- : tactic

#g "(n = 0) ==> (n * n = n)";;
"(n = 0) ==> (n * n = n)"

() : void

#e(DISCH_THEN SUBST1_TAC);;
OK..
"0 * 0 = 0"
```

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

```
#g "(0 = n) ==> (n * n = n)";;                                         1
"(0 = n) ==> (n * n = n)"

() : void

#e(DISCH_THEN(SUBST1_TAC o SYM));;
OK..
"0 * 0 = 0"
```

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

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

```
#g "((n = 0) \/ (n = 1)) ==> (n * n = n)";;                          1
"(n = 0) \/ (n = 1) ==> (n * n = n)"

() : void

#e(DISCH_THEN(DISJ_CASES_THEN SUBST1_TAC));;
OK..
2 subgoals
"1 * 1 = 1"

"0 * 0 = 0"
```

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

```
DISCH_TAC
 THEN DISJ_CASES_TAC(ASSUME "(n = 0) \/ (n = 1)")
 THENL[SUBST1_TAC(ASSUME "n = 0");
       SUBST1_TAC(ASSUME "n = 1")]
```

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

```
DISCH_TAC
 THEN SUBST1_TAC(ASSUME "n = 0")
 THEN REWRITE_TAC[MULT_CLAUSES]

DISCH_TAC
 THEN POP_ASSUM SUBST1_TAC
 THEN REWRITE_TAC[MULT_CLAUSES]

DISCH_TAC
 THEN ASSUM_LIST (SUBST1_TAC o el 1)
 THEN REWRITE_TAC[MULT_CLAUSES]

DISCH_TAC
 THEN POP_ASSUM_LIST (SUBST1_TAC o el 1)
 THEN REWRITE_TAC[MULT_CLAUSES]

DISCH_THEN SUBST1_TAC
 THEN REWRITE_TAC[MULT_CLAUSES]
```

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

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

```
CHOOSE_THEN : thm_tactical
```

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

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

```
let DISJ_CASES_TAC = DISJ_CASES_THEN ASSUME_TAC
```

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

The main theorem continuation functions in the system are:

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

See *REFERENCE* for full details. For `INDUCT_THEN`,
see Section 5.7.4 and *REFERENCE*.