Technical Report

UCAM-CL-TR-103
ISSN 1476-2986

Number 103

HOL

A proof generating system for higher-order logic

Mike Gordon

January 1987

ISSN 1476-2986

January 20, 1987

HOL

A Proof Generating System for Higher-Order Logic

Mike Gordon

Computer Laboratory

Corn Exchange Street

Cambridge CB2 3QG

Abstract

HOL is a version of Robin Milner's LCF theorem proving system for higher-order logic. It is currently being used to investigate

- how various levels of hardware behaviour can be rigorously modelled and
- how the resulting behavioral representations can be the basis for verification by mechanized formal proof.

This paper starts with a tutorial introduction to the meta-language ML. The version of higher-order logic implemented in the HOL system is then described. This is followed by an introduction to goal-directed proof with *tactics* and *tacticals* Finally, there is a little example showing the system in action. This example illustrates how HOL can be used for hardware verification.

Contents

1 Introduction to HOL **3**

2 Introduction to ML **3**

3 Syntax of the HOL logic **10**

3.1 HOL Types 10

3.2 HOL Terms 12

3.3 Quotation 13

3.4 Antiquotation 16

4 Sequents, Theorems and Proof **17**

4.1 Axioms and definitions 18

4.2 Type definitions 18

5 Theories **20**

5.1 Abstract Definition of a Theory 20

5.2 ML Functions for Manipulating Theories 21

5.3 The Theory bool 23

5.4 Primitive Rules of Inference of the HOL Logic 24

5.4.1 Assumption introduction 24

5.4.2 Reflexivity 25

5.4.3 Beta-conversion 25

5.4.4 Substitution 25

5.4.5 Abstraction 26

5.4.6 Type instantiation 26

5.4.7 Discharging an assumption 26

5.4.8 Modus Ponens 27

5.5 The Theory prod 27

5.6 The Theory sum 28

5.7 The Theory ind 29

5.8 The Theory num 29

5.9 The Theory prim_rec 30

5.10 The Theory arithmetic 32

5.11 The Theory list 33

5.12 The Theory tok 33

6 Derived Inference Rules **34**

7 Goal Directed Proof: Tactics and Tacticals **35**

7.1 Tactics 35

7.2 Using Tactics to Prove Theorems 37

7.3 Tacticals 38

7.3.1 `ORELSE : tactic -> tactic -> tactic` 38

7.3.2 `THEN : tactic -> tactic -> tactic` 38

7.3.3 `THENL : tactic -> tactic list -> tactic` 38

7.3.4 `REPEAT : tactic -> tactic` 38

7.3.5 `EVERY_ASSUM : (thm -> tactic) -> tactic` 39

7.3.6 `FIRST_ASSUM : (thm -> tactic) -> tactic` 39

7.4 Tactics Built into HOL 39

7.4.1 `ACCEPT_TAC : thm_tactic` 39

7.4.2 `DISJ_CASES_TAC : thm_tactic` 39

7.4.3 `ASM_CASES_TAC : term -> tactic` 40

7.4.4 `COND_CASES_TAC : tactic` 40

7.4.5 `REWRITE_TAC : thm list -> tactic` 40

7.4.6 `ASSUME_TAC : thm_tactic` 41

7.4.7 `CONJ_TAC : tactic` 41

7.4.8 `DISCH_TAC : tactic` 42

7.4.9 `GEN_TAC : tactic` 42

7.4.10 `IMP_RES_TAC : tactic` 42

7.4.11 `STRIP_TAC : tactic` 43

7.4.12 `SUBST_TAC : thm list -> thm` 43

7.4.13 `ALL_TAC : tactic` 43

7.4.14 `NO_TAC : tactic` 44

8 An Example of HOL in Action **44**

9 Conclusion **54**

1 Introduction to HOL

Higher-order logic is a promising language for specifying all aspects of hardware behaviour [8], [1]. It was originally developed as a foundation for mathematics [2]. Its use for hardware specification and verification was first advocated by Keith Hanna [9].

The approach to mechanising logic described in this paper is due to Robin Milner [6]. He originally developed the approach for a system called LCF designed for reasoning about higher-order recursively defined functions. The HOL system is a direct descendant of this work. The original LCF system was implemented at Edinburgh and is called "Edinburgh LCF". The code for it was ported from Stanford Lisp to Franz Lisp by Gerard Huet at INRIA and formed the basis for a French research project called "Formel". Huet's Franz Lisp version of LCF was further developed at Cambridge by Larry Paulson and eventually became known as "Cambridge LCF" [18]. The HOL system is implemented on top of Cambridge LCF and consequently many (good and bad) features of LCF are found in it. In particular, the axiomatization of higher-order logic used is not the classical one due to Church, but an equivalent formulation strongly influenced by LCF.

To make this paper self-contained, a brief introduction to ML (the LCF metalanguage) is included. The version of ML described here (and included as part of the HOL system) is not Standard ML [16] but the version of ML documented in the ML Handbook [4].

The acronym "HOL" refers to both the computer system and the version of higher-order logic that it supports; the former will be called the "HOL system" and the latter the "HOL logic".

Because this paper is about the HOL system, the machine-readable syntax for the logic will be presented. For example, instead of the conventional logical symbols $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$ and λ, we use the following strings of ASCII characters `/\`, `\/`, `~`, `!`, `?` and `\` respectively. In various other papers on HOL (*e.g.* [8], [1]) conventional notation, rather than the ASCII notation, is employed. The two notations are, of course, equivalent.

2 Introduction to ML

This section is a very brief introduction to the metalanguage ML. The aim is just to give a feel for what it is like to interact with ML and to introduce some features of the HOL system (*e.g.* the representation in ML of the types and terms of the

HOL logic[1]).

ML is an interactive programming language like Lisp. At top level one can evaluate expressions and perform declarations. The former results in the expression's value and type being printed, the latter in a value being bound to a name. The boxes below contain a little session with the system. The interactions in these boxes should be understood as occurring in sequence. For example, variable bindings made in earlier boxes are assumed to persist to later ones. To enter the HOL system one types `hol` to Unix[2]; the HOL system then prints a sign-on message and puts one into ML. The ML prompt is `#`, so lines beginning with `#` are typed by the user and other lines are the system's response.

```
% hol


_  _        __       _
|__|       |  |      |
|  | IGHER |__| RDER |__ OGIC
===============================
(Built on Sept  31)

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

The ML expression `1.[2;3;4;5]` has the form e_1 *op* e_2 where e_1 is the expression `1` (whose value is the integer 1), e_2 is the expression `[2;3;4;5]` (whose value is a list of four integers) and *op* is the infixed operator '`.`' which is like Lisp's *cons* function. Other list processing functions include `hd` (*car* in Lisp), `tl` (*cdr* in Lisp) and `null` (*null* in Lisp). The double semicolon '`;;`' terminates a top-level phrase. The system's response is shown on the line not starting with a prompt. It consists of the value of the expression followed, after a colon, by its type. The ML typechecker infers the type of expressions using methods invented by Robin Milner [15]. The type `int list` is the type of 'lists of integers'; `list` is a unary type operator. The type system of ML is very similar to the type system of the HOL logic which is explained in Section 3.1

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

[1]Types and terms are explained in detail in Section 3.2.

[2]The Unix prompt is `%`.

```
let l = it;;
l = [1; 2; 3; 4; 5] : int list

#tl l;;
[2; 3; 4; 5] : int list

#hd it;;
2 : int

#tl(tl(tl(tl(tl l))));;
[] : int list
```

Following standard λ-calculus usage, the application of a function f to an argument x can be written without brackets as fx (although the more conventional $f(x)$ is also allowed). The expression $fx_1x_2\cdots x_n$ abbreviates $(\cdots((fx_1)x_2)\cdots)x_n$ (*i.e.* function application associates to the left).

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

```
#let l1 = [1;2;3] and l2 = [`a`;`b`;`c`];;
l1 = [1; 2; 3] : int list
l2 = [`a`; `b`; `c`] : string list
```

ML expressions like `` `a` ``, `` `b` ``, `` `foo` `` *etc.* are *strings* and have type `string`. Any sequence of ASCII characters can be written between the quotes. The function `words` splits a single string into a list of single character strings, using space as separator.

```
#words`a b c`;;
[`a`; `b`; `c`] : string list
```

An expression of the form `(`e_1`,`e_2`)` evaluates to a pair of the values of e_1 and e_2. If e_1 has type σ_1 and e_2 has type σ_2 then `(`e_1`,`e_2`)` has type σ_1`#`σ_2. A tuple `(`e_1`,`...`,`e_n`)` is equivalent to `(`e_1`,(`e_2`,`...`,`e_n`))` (*i.e.* '`,`' is right associative). The brackets around pairs and tuples are optional; the system doesn't print them. The first and second components of a pair can be extracted with the ML functions `fst` and `snd` respectively.

```
#(1,true,`abc`);;
1,true,`abc` : (int # bool # string)

#snd it;;
true,`abc` : (bool # string)


#fst it;;
true : bool
```

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

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

Functions are defined with declarations of the form `let` $f\ v_1\ \ldots\ v_n = e$ where each v_i is either a variable or a pattern build out of variables [3]. Recursive functions are declared with `letrec` instead of `let`.

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

```
#letrec zip(l1,l2) =
#if null l1 or null l2
# then []
# else (hd l1,hd l2).zip(tl l1,tl l2);;
zip = - : ((* list # ** list) -> (* # **) list)

#zip([1;2;3],[`a`;`b`;`c`]);;
[1,`a`; 2,`b`; 3,`c`] : (int # string) list
```

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

```
#let curried_zip l1 l2 = zip(l1,l2);;
curried_zip = - : (* list -> ** list -> (* # **) list)

#let zip_num = curried_zip [0;1;2;3;4;5;6;7;8;9];;
zip_num = - : (* list -> (int # *) list)

#zip_num [`a`;`b`;`c`];;
[0,`a`; 1,`b`; 2,`c`] : (int # string) list
```

Curried function are useful because they can be 'partially applied' as illustrated above by the partial application of `curried_zip` to `[0;1;2;3;4;5;6;7;8;9]` which results in the function `zip_num`.

The evaluation of an expression either *succeeds* or *fails*. In the former case, the evaluation returns a value; in the latter case the evaluation is aborted and a failure string (usually the name of the function that caused the failure) is passed to whatever invoked the evaluation. This context can either propagate the failure (this is the default) or it can *trap* it. These two possibilities are illustrated below. A failure trap is an expression of the form $e_1?e_2$. An expression of this form is

[3]The ML Handbook [4] gives exact details.

evaluated by first evaluating e_1. If the evaluation succeeds (*i.e.* doesn't fail) then the value of $e_1?e_2$ is the value of e_1. If the evaluation of e_1 fails, then the value of $e_1?e_2$ is obtained by evaluating e_2.

```
hd[];;
evaluation failed     hd

#hd[] ? `hd applied to empty list`;;
`hd applied to empty list` : string
```

Terms of the HOL logic are represented in ML by a type called `term`. For example, the expression `"x /\ y ==> z"` evaluates in ML to a term representing $x \wedge y \supset z$. Anything between a pair of quotes is parsed as a logical term. The quotation mechanism is described in Section 3.3. Terms can be manipulated by various built-in ML functions. For example, `dest_imp` splits an implication into a pair of terms consisting of the antecedant and consequent, and `dest_conj` splits a conjunction into its conjuncts.

```
#"x /\ y ==> z";;
"x /\ y ==> z" : term

#dest_imp it;;
"x /\ y","z" : (term # term)

#dest_conj(fst it);;
"x","y" : (term # term)
```

Terms of the HOL logic are quite similar to ML expressions and this can at first be confusing. Indeed, terms of the logic have types similar to ML expressions. For example, `"(1,2)"` is an ML expression with ML type `term`. The HOL type of this term is `num#num`. By contrast, the ML expression `("1","2")` has type `term#term`. The types of HOL terms form an ML type called `type`. Expressions of the form `": ··· "` evaluate to logical types. The built-in function `type_of` has ML type `term->type` and returns the logical type of a term.

```
#"(1,2)";;
"1,2" : term

#type_of it;;
":num # num" : type

#("1","2");;
"1","2" : (term # term)

#type_of(fst it);;
":num" : type
```

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

```
#"(1,T)";;
"1,T" : term

#type_of it;;
":num # bool" : type
```

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

```
#let x = mk_var(`x`,":bool")
#and y = mk_var(`y`,":bool")
#and z = mk_var(`z`,":bool")
x = "x" : term
y = "y" : term
z = "z" : term
```

The constructors `mk_conj` and `mk_imp` construct conjunctions and implications respectively.

```
#let t = mk_imp(mk_conj(x,y),z);;
t = "x /\ y ==> z" : term
```

Theorems are represented in HOL by values of type `thm`. The only way to create theorems is by *proof*. For a logician, a formal proof is a sequence each of whose elements is either an *axiom* or follows from earlier members of the sequence by a *rule of inference*. In HOL (following LCF) a proof is a sequence in just that sense. There are five axioms of the HOL logic and eight primitive inference rules. The axioms are bound to ML names. For example, the Law of Excluded Middle is bound to the ML name `BOOL_CASES_AX`:

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

Theorems are printed with a preceding turnstile `|-` as illustrated above; `!` is the universal quantifier ∀. Rules of inference are ML functions that return values of type **thm**. For example, the rule of *specialization* (or ∀-elimination) is an ML function, `SPEC`, which takes as arguments a term "a" and a theorem `|-` `!`x.$t[x]$ and returns the theorem `|-` $t[a]$, the result of substituting a for x in $t[x]$.

```
#SPEC;;
- : (term -> thm -> thm)

#SPEC "1=2" BOOL_CASES_AX;;
|- ((1 = 2) = T) \/ ((1 = 2) = F)
```

A proof in the HOL system is constructed by repeatedly applying inference rules to axioms or to previously proved theorems.

Since proofs may consist of many tens of thousands of steps, it is necessary to provide tools to make proof construction easier for the user. The proof generating tools in the HOL system are just those of LCF, and are described in Section 7.1.

Theorems have *assumptions* and a *conclusion*. The inference rule `ASSUME` generates theorems of the form t `|-` t. The ML printer prints each assumption as a dot. The function `dest_thm` decomposes a theorem into a pair consisting of list of assumptions and the conclusion. The ML type goal abbreviates `(term)list#term`, this is motivated in Section 7.1.

```
#let th1 = ASSUME "t1==>t2";;
th1 = . |- t1 ==> t2

#dest_thm th1;;
["t1 ==> t2"],"t1 ==> t2" : goal
```

The inference rule `UNDISCH` moves the antecedant of an implication to the assumptions.

```
#let th2 = UNDISCH th1;;
th2 = .. |- t2

#dest_thm th2;;
["t1 ==> t2"; "t1"],"t2" : goal
```

The rule `DISCH` takes a term and a theorem and 'discharges' the given term from the assumptions of the theorem. The functions `hyp` and `concl` select the list of hypotheses and the conclusion of a theorem, respectively. They just return the components of the pair computed by `dest_thm`.

```
#DISCH "t1==>t2" (DISCH "t1" th2);;
|- (t1 ==> t2) ==> t1 ==> t2

#let th3 = DISCH "t1==>t2" (DISCH "t1" th2);;
th3 = |- (t1 ==> t2) ==> t1 ==> t2

#hyp th3;;
[] : term list

#concl th3;;
"(t1 ==> t2) ==> t1 ==> t2" : term
```

In the sections that follow we systematically describe HOL. We then conclude with another session illustrating the various theorem-proving tools in action.

3 Syntax of the HOL logic

In this section, the structure of HOL terms and types is explained. The particular sets of types, constants and axioms that are available to a user of the HOL system is specified by the *theory* he or she is working in. Each theory $\mathcal{T}$ specifies sets $\mathsf{Tyops}_{\mathcal{T}}$ and $\mathsf{Consts}_{\mathcal{T}}$ of type operators and constants. These sets then determine the sets $\mathsf{Types}_{\mathcal{T}}$ and $\mathsf{Terms}_{\mathcal{T}}$ of types and terms. How to set up theories is described in Section 5.

We start by giving fairly abstract definitions of the sets $\mathsf{Types}_{\mathcal{T}}$ and $\mathsf{Terms}_{\mathcal{T}}$. Various constructor and destructor functions are then described. Finally, we explain the quotation mechanism built-in to the ML parser. This mechanism includes a type inference algorithm that enables the user to input terms without having to write an excessive amount of explicit type information.

In what follows, a *name* is a sequence of letters, digits or primes (') beginning with a letter. The set of names is denoted by Names. For example, `x`, `fred23`, `fred23'` and `fred23''` are all members of Names.

3.1 HOL Types

Object language types are expressions that denote sets. Following tradition, we use σ, σ', σ'', ... , σ_1, σ_2, *etc.* to stand for arbitrary types. HOL terms will be defined in such a way that all well-formed terms are type-consistent. Before describing the terms of the HOL logic we must first specify its types.

There are four kinds of object language types. These can be described informally as follows.

1. **Type variables:** These are sequences of asterisks, optionally followed by sequences of digits. The set of type variables is denote by Tyvars; for example, `*`, `**`, `***`, `*1`, `***24` are all members of Tyvars. Type variables provide the system with a limited amount of *polymorphism* (this is explained later). Small Greek letters, possibly with subscripts or primes, are used to stand for type variables.

2. **Type constants:** These are names like `bool`, `num` and `tok`. They denote fixed sets of values.

3. **Function types:** If σ_1 and σ_2 are types, then σ_1`->`σ_2 is the function type with *domain* σ_1 and *range* σ_2; it denotes the set of functions from the set denoted by its domain to the set denoted by its range.

4. **Compound types:** These have the form $(\sigma_1, \ldots, \sigma_n)op$, where the types $\sigma_1, \ldots, \sigma_n$ are the argument types and *op* is a *type operator* of arity n. Type operators must be names; they 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 $\sigma_1, \ldots, \sigma_n$. 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.

Although these four kinds of types are logically distinct, the HOL system (following LCF) represents constant types as compound types built with 0-ary type operators and function types as compound types built with a 2-ary type operator called `fun`. Thus the constant type `num` is represented as the compound type `()num` and σ_1`->`σ_2 is represented as (σ_1,σ_2)`fun`.

In general, compound types must be written in the postfixed form described above, but there are two exceptions (in addition to `->`).

- *Cartesian product* types of the form (σ_1,σ_2)`prod` can be written as σ_1`#`σ_2.

- *Disjoint union* types of the form (σ_1,σ_2)`sum` can be written as σ_1`+`σ_2.

Products and unions are not primitive, but because they are so useful, the HOL parser treats them specially. These types are explained in Section 5.5 and Section 5.6 respectively.

It is useful to describe the set of types of a theory a bit more formally. Each theory $\mathcal{T}$ determines a set $\mathsf{Tyops}_{\mathcal{T}}$ of type operators. A type operator is a pair

(op, n) where op is the name of the operator and n is its arity. The set $\mathsf{Types}_{\mathcal{T}}$ of types of the theory $\mathcal{T}$ is the smallest set such that:

- $\mathsf{Tyvars} \subseteq \mathsf{Types}_{\mathcal{T}}$.
- If $\sigma_1 \in \mathsf{Types}_{\mathcal{T}}$ and $\sigma_2 \in \mathsf{Types}_{\mathcal{T}}$ then σ_1`->`$\sigma_2 \in \mathsf{Types}_{\mathcal{T}}$.
- If $\sigma_i \in \mathsf{Types}_{\mathcal{T}}$ (for all i between 1 and n) and $(op, n) \in \mathsf{Tyops}_{\mathcal{T}}$ then `(`σ_1`, ... ,`σ_n`)`$op \in \mathsf{Types}_{\mathcal{T}}$.

Types containing type variables are called *polymorphic*; others are *monomorphic*.

An *instance* σ' of a type σ is obtained by replacing all occurrences of a type variable in σ by a type. The only instance of a monomorphic type is itself. Note that $\mathsf{Types}_{\mathcal{T}}$ is closed under the operation of taking instances.

3.2 HOL Terms

A theory $\mathcal{T}$ specifies a set $\mathsf{Consts}_{\mathcal{T}}$ of *constants*. Each constant is a pair (c, σ) where c is a name and σ is a type of $\mathcal{T}$ (*i.e.* $\sigma \in \mathsf{Types}_{\mathcal{T}}$); σ is called the *generic type* of c.

Distinct constants of a theory cannot have the same name; (*i.e.* if (c, σ) and (c, σ') are both members of $\mathsf{Consts}_{\mathcal{T}}$ then $\sigma = \sigma'$).

The set $\mathsf{Terms}_{\mathcal{T}}$ of *terms* of the theory $\mathcal{T}$ is the smallest set such that:

1. If x is a name which is not the name of a constant in $\mathcal{T}$ (*i.e.* $\mathsf{Consts}_{\mathcal{T}}$ does not contain a pair whose first component is x) and $\sigma \in \mathsf{Types}_{\mathcal{T}}$, then $(x, \sigma) \in \mathsf{Terms}_{\mathcal{T}}$. Terms formed in this way are called *variables.*

2. If $(c, \sigma) \in \mathsf{Consts}_{\mathcal{T}}$ and $\sigma' \in \mathsf{Types}_{\mathcal{T}}$ is an instance of σ, then $(c, \sigma') \in \mathsf{Terms}_{\mathcal{T}}$. Terms formed in this way are called *constants.*

3. If $(t, \sigma'$`->`$\sigma) \in \mathsf{Terms}_{\mathcal{T}}$ and $(t', \sigma') \in \mathsf{Terms}_{\mathcal{T}}$ then $(comb, t_1, t_2, \sigma) \in \mathsf{Terms}_{\mathcal{T}}$. Terms formed in this way are called *combinations* or *function applications.*

4. If $(x, \sigma) \in \mathsf{Terms}_{\mathcal{T}}$ (where x is a name that is not the name of a constant in $\mathcal{T}$) and $(t, \sigma') \in \mathsf{Terms}_{\mathcal{T}}$ and σ`->`$\sigma' \in \mathsf{Types}_{\mathcal{T}}$ then $(abs, x, t, \sigma$`->`$\sigma') \in \mathsf{Terms}_{\mathcal{T}}$. Terms formed in this way are called *abstractions* or λ-terms (\ is HOL's ASCII approximation to λ).

Members of the sets $\mathsf{Types}_{\mathcal{T}}$ and $\mathsf{Terms}_{\mathcal{T}}$ correspond closely to the internal representations of ML values of type `type` and `term` in the HOL system. The following constructor functions (listed below with their ML types) can be used to input types and terms; they are explained in Section 3.3 below.

```
mk_vartype : string -> type
mk_type    : (string # type list) -> type

mk_var     : (string # type) -> term
mk_const   : (string # type) -> term
mk_comb    : (term # term) -> term
mk_abs     : (term # term) -> term
```

The HOL logic consists of the build-in theories `bool` and `ind` (see Sections 5.3 and 5.7)) together with eight rules of inference. In the theory `bool` the type `bool` is introduced (we often name theories after the type which they introduce). Terms of type `bool` are called *formulas*. There are two constant formulas: `T` and `F`; these represent *true* and *false* respectively. It follows from the Law of Excluded Middle (an axiom of HOL) that any formula is either equivalant to `T` or to `F`.

3.3 Quotation

It would be extremely tedious to always have to input types and terms using the constructor functions. The HOL system has a special parser and type-checker that enable them to be input using a fairly standard syntax. The HOL printer also outputs types and terms using this syntax.

For example, the ML expression `":bool->bool"` denotes exactly the same value (of ML type `type`) as

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

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

```
mk_abs
 (mk_var(`x`,mk_type(`num`,[])),
  mk_comb
  (mk_comb
   (mk_const
    (`+`,
     mk_type(`fun`,[mk_type(`num`,[]);
                    mk_type(`fun`,[mk_type(`num`,[]);
                                   mk_type(`num`,[])])])),
    mk_var(`x`, mk_type(`num`,[]))),
   mk_const(`1`, mk_type(`num`,[]))))
```

Notice that there is no explicit type information in `"\x.x+1"`. The HOL type-checker knows that (`1`, `num`) $\in$ $\mathsf{Consts}_{\mathtt{num}}$ and (`+`, `num->(num->num)`) $\in$ $\mathsf{Consts}_{\mathtt{num}}$, *i.e.* that the constants `1` and `+` have type `num` and `num->(num->num)`, respectively. From the types of `1` and `+` the type-checker infers that both occurrences of `x` in

"\x.x+1" could have type num. This is not the only type assignment possible; one could, for example, make the first occurrence of x have type bool and the second one have type num. In that case there would be two *different* variables with name x, namely (x, bool) and (x, num), the second of which is free. In fact, in HOL, the only way to construct a term with this second type assignment would be by using constructors, since the type-checker uses the heuristic that variables with the same name have the same type. The type-checker was designed by Robin Milner. It uses the heuristics like the one above to infer a sensible type for all variables occurring in a term. It uses the types of any constants in making this inference. If there are not enough clues, then the system will complain with the message

```
evaluation failed    types indeterminate in quotation
```

To give the system a hint, one can explicitly indicate types by following any subterm by a colon and then a type. For example, "f(x:num):bool" will typecheck with f and x getting types num->bool and num respectively. If there are polymorphic constants in a term, there must be enough type information to uniquely identify a type instance for each such constant.

The type-checking algorithm used for the HOL logic differs from that used for ML. For example, the ML expression \x.x will get ML type *->*, but the HOL term "\x.x" will fail to type-check and will result in the message shown above. To get this term to type-check one must indicate explicity the type of the variable x by writing, for example, "\x:*.x". This treatment of types is inherited from LCF.

The table below shows ML expressions for various kinds of type quotations. The expressions in adjacent columns are equivalent.

Types

Kind of type	ML *quotation*	*Constructor expression*
Type variable	":$*\cdots$"	mk_vartype(\`$*\cdots$\`)
Type constant	":*op*"	mk_type(\`*op*\`,[])
Function type	":σ_1->σ_2"	mk_type(\`fun\`, [":σ_1";":σ_2"])
Compound type	":(σ_1, ... , σ_n)*op*"	mk_type(\`*op*\`, [":σ_1"; ... ;":σ_n"])

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	"$\backslash x.t$"	mk_abs("x","t")

The HOL quotation mechanism can translate various standard logical notations into primitive terms. For example, if `+` has been declared an infix (as explained in Section 5), then `"x+1"` is translated to `"$+ 1 2"`. The escape character `$` 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	"$t_1\backslash/t_2$"	"\$\/ t_1 t_2"	mk_disj("t_1","t_2")
Conjunction	"$t_1/\backslash t_2$"	"\$/\ 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	"!$x.t$"	"\$!($\backslash x.t$)"	mk_forall("x","t")
∃-quantification	"?$x.t$"	"\$?($\backslash x.t$)"	mk_exists("x","t")
ε-term	"@$x.t$"	"\$@($\backslash x.t$)"	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($\backslash x.t_2$)t_1"	mk_let("x","t_1","t_2")

The constants `COND` and `LET` are explained in Section 5.3. The constants `\/`, `/\`, `==>` and `=` are examples of *infixes*. 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* (see Section 5.3 also). If c is declared to be a binder, then the HOL parser will translate "$c\ x.t$" to the combination "\$$c$(\\$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.\ \cdots$ \\$x_n.t$"	`list_mk_abs(`["x_1"; ... ;"x_n"],"t"`)`
"!$x_1 \cdots x_n.t$"	"!$x_1.\ \cdots$!$x_n.t$"	`list_mk_forall(`["x_1"; ... ;"x_n"],"t"`)`
"?$x_1 \cdots x_n.t$"	"?$x_1.\ \cdots$?$x_n.t$"	`list_mk_exists(`["x_1"; ... ;"x_n"],"t"`)`

The parser will also convert "\(x_1,x_2).t" to "UNCURRY(\\$x_1\ x_2.t$)" (the constant UNCURRY is described in Section 5.5). This 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, if b is a binder, then "b(x_1,x_2).t" is parsed as "\$b(\(x_1,x_2).t)". For example, "!(x,y).x>y" parses to "\$!(UNCURRY(\x.\y.\$> x y))" (> is an infixed constant of the theory num meaning "is greater than").

3.4 Antiquotation

Within a quotation, expressions of the form ^(t) (where t is an ML expression of type term or type) are called *antiquotations*. An antiquotation ^(t) evaluates to the ML value of t. For example, "x \/ ^(mk_conj("y:bool","z:bool"))" evaluates to the same term as "x \/ (y /\ z)". The most common occurrence 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 antiquotion.

```
#let y = "x+1";;
y = "x + 1" : term

#let z = "y = ^y";;
z = "y = x + 1" : term

#"!x.?y.^z";;
"!x. ?y. y = x + 1" : term
```

4 Sequents, Theorems and Proof

The HOL system supports proof by *natural deduction*. Assertions in the HOL logic are not individual formulas (*i.e.* boolean terms), but are *sequents* of the form (Γ, t), where Γ is a set of formulas called the *assumptions* and t a formula called the *conclusion*. A sequent(Γ, t) asserts that if all the formulas in Γ are true, then so is t. Using sequents, one can design a deductive system in which proofs are more 'natural' than in other (*e.g.* Hilbert-style) systems.

Sequents are represented in ML by pairs with ML type `(term list)#term`; the first component represents the assumptions and the second component the conclusion. For example, `(["x>y";"y>z"],"x>z")` is an ML expression representing the sequent with assumptions `"x>y"` and `"y>z"` and conclusion `"x>z"`.

A *theorem* is a sequent that has a *proof*. More precisely, a theorem is a sequent that is either an *axiom*, or follows from other theorems by a *rule of inference*.

To guarantee that the only way to get theorems is by proof, the HOL system (following LCF) distinguishes sequents from theorems by having a separate ML type `thm`[4]. There are five initial theorems corresponding to the five axioms of the HOL logic. The only way to generate other objects with ML type `thm` is by using one of the eight primitive inference rules of the logic (see Section 5.4). ML's type discipline ensures that only inference rules are applicable to theorems [5].

Interesting theorems require large numbers of applications of the primitive inference rules for their proof. For example, a recent verification of the design of a simple microprocessor took over a million inference steps [3]. The HOL system provides tools to help the user generate such proofs. One such tool is a *derived inference rule*. This is an ML function which invokes sequences of applications of the eight primitive rules. There are about a hundred derived inference rules predefined in the system. A brief discussion of derived rules can be found in Section 6. Proofs can be generated in a goal-oriented style using *tactics*; these were invented (for LCF) by Robin Milner and are explained in Section 7.1. The goal-directed proof style starts with a formula (representing the goal one wants to prove) and then reduces this to sub-goals, sub-sub-goals *etc.* until the problem is decomposed into trivial goals (*e.g.* instances of axioms). This is in contrast to *forward proof* in which one works forward from the axioms, using the rules of inference, until the desired theorem is deduced.

[4]In ML jargon, `thm` is an abstract type whose representing type is `(term list)#term`.

4.1 Axioms and definitions

Each theory $\mathcal{T}$ has a set $\mathsf{Axioms}_{\mathcal{T}} \subseteq \mathsf{Terms}_{\mathcal{T}}$ of terms of type `bool`, called axioms. These axioms usually specify the meaning of the constants of $\mathcal{T}$; however, the HOL system enables the user to assert *any* formula as an axiom.

A *definition* is an axiom of the form c=t, where c is a constant and t is a closed term[5]. A constant c is *defined in* $\mathcal{T}$ if there is exactly one axiom in $\mathcal{T}$ containing c and that axiom is a definition. If all the axioms of $\mathcal{T}$ are definitions, then $\mathcal{T}$ is a *definitional* theory. Definitional theories have the important property that they cannot introduce any new inconsistency that wasn't already present; they are what logicians call *conservative extensions*.

Axioms of the form $f\ x_1 \ldots x_n$ = t, where the free variables of t are included among x_1, ... , x_n, are also regarded as definitions. This is because the formula $f\ x_1 \ldots x_n$ = t is equivalent to the ordinary definition f = \\$x_1 \ldots x_n$.$t$. Definitions are also allowed to have the form f(x_1,...,x_n) = t.

Another kind of conservative extension is a *type definition*.

4.2 Type definitions

There does not seem to be any standard notion of type definition in the literature on higher-order logic. The mechanism described in this section was suggested to me by Mike Fourman.

Recalling that a constant type denotes a set and an n-ary type operator denotes an operation for combining n sets to form a set, the intuitive idea behind type definitions in the HOL logic is as follows:

- A constant type is defined by specifying it to be in one-to-one correspondence with a non-empty subset of an existing type.

- An n-ary type operator *op* is defined by specifying (α_1,...,α_n)*op* to be in one-to-one correspondence with a non-empty subset of an existing type constructed from α_1, ... , α_n.

All types in the HOL logic must denote non-empty sets. This is because the term `@x:`σ`.T` denotes a member of the set denoted by σ (for arbitrary σ). The binder `@` is explained in Section 5.3.

The HOL system has a single mechanism for defining types. The idea behind this mechanism is to define an n-ary type operator[6] *op* by specifying the set denoted

[5]A term is *closed* if it contains no free variables.

[6]Constant types are defined by taking $n = 0$.

by $(\alpha_1,\ldots,\alpha_n)op$ for all denotations of the type variables $\alpha_1, \ldots, \alpha_n$. This set is specified by giving a (polymorphic) term denoting its characteristic function[7].

This idea can perhaps be made clearer with a concrete example. Suppose we want to define a binary type operator `iso` such that the type denoted by `":(*,**)iso"` is the subset of the set denoted by `":*->**"` consisting of those functions which have an inverse. The characteristic function of the set of functions with inverses is denoted by the term `"\f:*->**.?g.!x. g(f x) = x"`. To ensure that only non-empty sets are used to define types, the HOL system requires the user to supply a theorem asserting non-emptiness when making a type definition. If t_{op} is the term representing the characteristic function defining *op*, then the user is required to supply a theorem of the form `|- ?`$x.t_{op}\ x$. In the case of our example, *op* is `iso` and $t_{\mathtt{iso}}$ is `"\f:*->**.?g.!x. g(f x) = x"`, so the required theorem is:

```
|- ?f. (\f:*->**.?g.!x. g(f x) = x) f
```

The ML function

```
new_type_definition : (string # term # thm) -> thm
```

is used to make a new type definition. Suppose *op* is a name, t_{op} is a term whose object language type has the form σ`->bool` (where σ contains n distinct type variables $\alpha_1, \ldots, \alpha_n$) and *th* is the theorem `|- ?`$x.t_{op}\ x$. Then

`new_type_definition(`op, t_{op}, th`)`

declares *op* to be a new n-ary type-operator such that $(\alpha_1,\ldots,\alpha_n)op$ denotes the set with characteristic function t_{op}. This set is a subset of the set denoted by σ. The theorem *th* ensures that this subset is non-empty for all values of the type variables $\alpha_1, \ldots, \alpha_n$. The type σ is called the *representing type* of *op*. The result of the type definition is an axiom asserting the existence of a one-to-one function from $(\alpha_1,\ldots,\alpha_n)op$ onto the subset of σ characterized by t_{op}.

Suppose we have proved `|- ?f. (\f:*->**.?g.!x. g(f x) = x) f` and bound this theorem to the ML name `th`. Then the new type `iso` is defined by executing the ML expression

```
new_type_definition(`iso`, "(\f:*->**.?g.!x. g(f x) = x)", th)
```

The HOL system then automatically declares a new 2-ary type operator `iso`, a new constant `REP_iso` (whose name is generated by prefixing the name of the type operator being defined with the string `` `REP_` ``), and finally, HOL automatically generates the axiom

[7]The *characteristic function* of a set is a boolean-valued function that maps an element to *true* if and only if the element is a member of the set.

```
|- ONE_ONE REP_iso /\
   (!f. (\f. ?g. !x. g(f x) = x)f = (?f'. f = REP_iso f'))
```

This axiom has the form

$$\texttt{|- ONE\_ONE REP\_}op\texttt{ /\textbackslash{} !f.}t_{op}\texttt{ f = (?f'. f = REP\_}op\texttt{ f')}$$

where the function `ONE_ONE` is defined by

```
|- ONE_ONE f = (!x1 x2. (f x1 = f x2) ==> (x1 = x2))
```

The constant `ONE_ONE` is introduced in the theory `bool` (see Section 5.3) and so is built into the HOL system.

5 Theories

The set of types, constants and axioms that are available in HOL depends on the *theory* in which one is working (see Section 3). Theories form a hierarchy in which a theory $\mathcal{T}_1$ can be a *parent* of another theory $\mathcal{T}_2$, meaning that the types, constants and axioms of $\mathcal{T}_1$ will be available in $\mathcal{T}_2$. If $\mathcal{T}_1$ is a parent of $\mathcal{T}_2$, then $\mathcal{T}_2$ is an *immediate descendant* of $\mathcal{T}_1$. $\mathcal{T}_2$ is a *descendant* of $\mathcal{T}_1$ if the two theories are in the transitive closure of the immediate descendant relation. In this case we also say that $\mathcal{T}_1$ is an *ancestor* of $\mathcal{T}_2$. When first entering the HOL system one enters a theory called `HOL`; this is a descendant of several theories including `bool` and theories of numbers, lists and pairs. In the subsections below, we describe the ML functions provided for manipulating theories and the various theories built in to the HOL system. First we describe a bit more abstractly exactly what a theory is.

5.1 Abstract Definition of a Theory

A theory $\mathcal{T}$ is characterized by a 4-tuple

$$(\mathsf{Parents}_{\mathcal{T}},\ \mathsf{Tyops}_{\mathcal{T}},\ \mathsf{Consts}_{\mathcal{T}},\ \mathsf{Axioms}_{\mathcal{T}})$$

where:

- $\mathsf{Parents}_{\mathcal{T}}$ is a (finite) set of all the parent theories of $\mathcal{T}$.
- $\mathsf{Tyops}_{\mathcal{T}}$ is a set of type operators (see Section 4.2).
- $\mathsf{Consts}_{\mathcal{T}}$ is a set of constants (see Section 3.2).
- $\mathsf{Axioms}_{\mathcal{T}}$ is a set of theorems which are the axioms of $\mathcal{T}$ (see Section 4.1).

5.2 ML Functions for Manipulating Theories

In the HOL system, the four components of a theory are stored on disk in files with names of the form *name*.th, where *name* is the name given to the theory (with new_theory) when it is created. Various additional pieces of information are stored in the *name*.th files, including the parsing status of the contants (*i.e.* whether they are infixes or binders), which axioms are definitions (see Section 4.1) and the theorems that have been proved and saved by the user.

The ML functions for creating theories are listed below.

```
new_theory              : string -> void
new_parent              : string -> void
new_type                : int -> string -> void
new_constant            : (string # type) -> void
new_infix               : (string # type) -> void
new_binder              : (string # type) -> void
new_axiom               : (string # term) -> thm
new_definition          : (string # term) -> thm
new_infix_definition    : (string # term) -> thm
new_binder_definition   : (string # term) -> thm
new_type_definition     : (string # term # thm) -> thm
```

The effect of these functions should be reasonably clear from their names and types. The first argument of new_type is the arity of the type operator being declared; the second argument is its name. The arguments of 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 various functions for setting up theories are illustrated in the example session in Section 8.

A theory with no descendants can be extended by adding new parents, types, constants, axioms and definitions. Theories that are already the parents of other theories cannot be extended because it would be messy (though not impossible) to implement the necessary checks to ensure that added types, constants *etc.* did not invalidate declarations in the descendant theories. When one is creating a new theory or extending an existing theory one is said to be in *draft mode*. When one is working in a completed theory, one is said to be in *working mode*; in this mode the functions with prefix "new_" listed above are not available. In draft mode there is the danger that one is not prevented from asserting inconsistent axioms such as |- T=F. See the discussion of definitional axioms in Section 4.1.

The functions for entering an already existing theory in either draft mode or working mode are, respectively:

```
extend_theory : string -> void
load_theory   : string -> void
```

To finish a session and write all new declarations to the theory file there is the function:

```
close_theory : void -> void
```

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

Once a theorem has been proved, it can be saved with the function

```
save_thm : string # thm -> thm
```

Evaluating `save_thm(`Th1`,th)` will save the theorem `th` with name `Th1` on the current theory. Individual axioms, definitions and theorems can be read (from ancestor theories) using the following ML functions:

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

Theories can be printed using the function `print_theory`, which takes a theory name (a value of type `string`) and then prints out the named theory in a readable format.

In the remaining subsections of this section we describe all the theories built into the HOL system[8]. The theory `HOL` has all of these as parents.

[8]The actual theory structure in the HOL system is very slightly different from that described in this paper. Specifically, the types and definitions for pairs are included in the theory `bool` (rather than being in a separate descendant theory called `prod`). It is hoped that future versions of the system will correct this anomaly.

5.3 The Theory bool

The theory bool introduces the type bool and contains four of the five axioms for higher-order logic (the fifth axiom is in the theory ind). These axioms, together with the rules of inference described in Section 5.4, constitute the core of the HOL logic. Because of the way the HOL system evolved from LCF[9], the particular axiomatization of higher-order logic it uses is superficially different from the classical axiomatization due to Church [2]. The biggest difference is that in Church's formulation, type variables are in the meta-language, whereas in the HOL logic they are part of the object language.

There are three primitive constants in the theory bool: = (equality, an infix), ==> (implication, an infix) and @ (choice, a binder). 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 Leisenring's book [12] and Church's original paper [2].

The logical constants T (truth), F (falsity), ~ (negation), /\ (conjunction), \/ (disjunction), ! (universal quantification) and ? (existential quantification) 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:*. x)=(\x. x))

FORALL_DEF     |- $!  = \P:*->bool. P=(\x. T)

EXISTS_DEF     |- $?  = \P:*->bool. P($@ P)

AND_DEF        |- $/\ = \t1 t2. !t. (t1 ==> t2 ==> t) ==> t

OR_DEF         |- $\/ = \t1 t2. !t. (t1 ==> t) ==> (t2 ==> t) ==> t

F_DEF          |- F   = !t. t

NOT_DEF        |- $~  = \t. t ==> F
```

There are four axioms in the theory bool:

[9]In order to simplify the porting of the LCF theorem-proving tools to the HOL system, the HOL logic was made as like PPLAMBDA (the logic built-in to LCF) as possible.

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

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

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

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

The fifth and last axiom of the HOL logic is the Axiom of Infinity; this is in the theory `ind` described in Section 5.7. The theory `bool` 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))

ONE_ONE_DEF  |- ONE_ONE f = (!x1 x2. (f x1 = f x2) ==> (x1 = x2))

ONTO_DEF     |- ONTO f    = (!y. ?x. y = f x)
```

The constant `LET` is used in representing terms containing local variable bindings (*i.e.* `let`-terms, as in Section 3.3). The constant `COND` is used in representing conditionals. The constant `ONE_ONE` is used in type definitions (see Section 3.3). The constant `ONTO` is used in stating the Axiom of Infinity (see Section 5.7).

5.4 Primitive Rules of Inference of the HOL Logic

There are eight primitive rules of inference of the HOL logic. We will specify these using standard natural deduction notation. The metavariables t, t_1, t_2, *etc.* stand for arbitrary terms. The theorems above the horizontal line are called the *hypotheses* of the rule and the theorem below the line is called the *result*. Each rule asserts that its result can be deduced from its hypotheses, provided any restrictions (listed after the bullets) hold.

The first three rules have no hypotheses; their results can always be deduced. The square brackets contain the ML names of the rules followed by their ML types.

5.4.1 Assumption introduction

[`ASSUME : term -> thm`]

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

`ASSUME "`t`"` evaluates to `|-` t.

5.4.2 Reflexivity

[`REFL : term -> thm`]

$$\frac{}{\vdash t = t}$$

`REFL` "t" evaluates to $\vdash t = t$.

5.4.3 Beta-conversion

[`BETA_CONV : term -> thm`]

$$\frac{}{\vdash (\backslash x.t_1)t_2 = t_1[t_2/x]}$$

- where $t_1[t_2/x]$ denotes the result of substituting t_2 for x in t_1, with the restriction that no free variables in t_2 become bound after substitution into t_1.

`BETA_CONV` "$(\backslash x.t_1)t_2$" evaluates to the theorem $\vdash (\backslash x.t_1)t_2 = t_1[t_2/x]$.

5.4.4 Substitution

[`SUBST : (thm#term)list -> term -> thm -> thm`]

$$\frac{\Gamma_1 \vdash t_1{=}t'_1 \quad \cdots \quad \Gamma_n \vdash t_n{=}t'_n \quad \Gamma \vdash t[t_1,\ldots,t_n/x_1,\ldots,x_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t'_1,\ldots,t'_n]}$$

- where $t[t_1,\ldots,t_n]$ denotes a term t with some free occurrences of the terms t_1, ... , t_n singled out and $t[t'_1,\ldots,t'_n]$ denotes the result of simultaneously replacing each occurrences of t_i by t'_i (for $1 \leq i \leq n$), with the restriction that the context $t[\ ]$ must not bind any variable occurring free in either t_i or t'_i (for $1 \leq i \leq n$).

- $\cup$ denotes set union.

The first argument to `SUBST` is a list $[(\vdash t_1{=}t'_1,\ x_1);\ldots(\vdash t_n{=}t'_n,\ x_n)]$. 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 $\vdash t_i{=}t'_i$ are to be done.

5.4.5 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.

If `th` is an ML name for the theorem Γ `|-` t_1=t_2, then `ABS "`x`" th` returns the theorem Γ `|-` $(\backslash x.t_1)$ `=` $(\backslash x.t_2)$.

5.4.6 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 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 $1 \leq i \leq n$).

5.4.7 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.

If `th` is the theorem Γ `|-` t_2, then the ML expression `DISCH "`t_1`" th` evaluates to the theorem $\Gamma-\{t_1\}$ `|-` t_1 `==>` t_2. Note that if t_1 is not a member of Γ, then $\Gamma-\{t_1\} = \Gamma$.

5.4.8 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.

5.5 The Theory prod

The binary type operator prod denotes the Cartesian product operator (see Section 3.1). Values of type (σ_1,σ_2)prod are ordered pairs whose first component has type σ_1 and whose second σ_2. The HOL parser recognises σ_1#σ_2 as an abbreviation for (σ_1,σ_2)prod.

Cartesian products can be defined by representing a pair (t_1,t_2) as the function \x y.(x=t_1)/\(y=t_2). The representing type of σ_1#σ_2 is thus σ_1->σ_2->bool. To define pairs this way, we first evaluate the ML expressions:

```
new_definition
 (`MK_PAIR_DEF`, "MK_PAIR(x:*)(y:**) = \a b.(a=x)/\(b=y)")

new_definition
 (`IS_PAIR_DEF`, "IS_PAIR p = ?x:*.?y:**. p = MK_PAIR x y")
```

We then prove that:

```
|- ?p:*->**->bool. IS_PAIR p
```

which is easily done (since |- IS_PAIR(MK_PAIR x y) is easily proved). This theorem is called PAIR_EXISTS. Next we define the type operator prod by evaluating

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

This results in the constant REP_prod being declared and the following axiom (called DEF_prod) being asserted.

```
|- ONE_ONE REP_prod /\ (!p. IS_PAIR p = (?p'. p = REP_prod p'))
```

The infix constructor "," and the selectors FST:*#**->* and SND:*#**->** can then be defined by:

```
new_infix_definition
 ('COMMA_DEF', "$, (x:*) (y:**) = @p. REP_prod p = MK_PAIR x y")

new_definition
 ('FST_DEF', "FST(p:(*,**)prod) = @x.?y. MK_PAIR x y = REP_prod p")

new_definition
 ('SND_DEF', "SND(p:(*,**)prod) = @y.?x. MK_PAIR x y = REP_prod p")
```

From these definitions and the axiom DEF_prod, the following theorems are proved and stored in the theory prod.

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

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

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

In addition to the constants just described, the theory prod also contains the definitions of CURRY and UNCURRY.

```
|- CURRY f x y     = f(x,y)

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

These constants are used for representing generalized the abstractions of the form \\(x_1,...,x_n).t described in Section 3.3.

5.6 The Theory sum

A type (σ_1,σ_2)sum (which may be abbreviated as σ_1+σ_2) denotes the disjoint union of types σ_1 and σ_2. The type operator sum can be defined just as prod was, but we omit the details here[10]. From the user's point of view, all that is needed are the functions

```
INL  : * -> *+**
INR  : ** -> *+**
OUTL : *+** -> *
OUTR : *+** -> **
ISL  : *+** -> bool
ISR  : *+** -> bool
```

INL and INR are the *injections* for inserting elements into the sum; OUTL and OUTR are the corresponding *projections* out of the sum. The predicates ISL and ISR test whether an element of a sum is in the left or right summand respectively. The following theorems are pre-proved in the system.

[10]The definition of disjoint unions in the HOL system is due to Tom Melham.

```
ISL          |- (!e. ISL(INL e)) /\ (!e. ~ISL(INR e))
ISR          |- (!e. ISR(INR e)) /\ (!e. ~ISR(INL e))
ISL_OR_ISR   |- !s. ISL s \/ ISR s
OUTL         |- !e. OUTL(INL e) = e
OUTR         |- !e. OUTR(INR e) = e
INL          |- !s. ISL s ==> (INL(OUTL s) = s)
INR          |- !s. ISR s ==> (INR(OUTR s) = s)
```

These theorems follow from the definitions.

5.7 The Theory ind

The theory ind introduces the type ind of *individuals* and the *Axiom of Infinity*. This axiom states that the set denoted by ind is infinite. The four axioms of the theory bool, the rules of inference in Section 5.4 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 giving a consistency proof of them.

The Axiom of Infinity is called INFINITY_AX; it states:

```
|- ?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.

5.8 The Theory num

The type num of natural numbers can be defined as equivalent to a countable subset of ind. Peano's axioms can then be proved as theorems. However, this has not yet been done in the HOL system. The type num is declared as a new primitive type and the constants 0 and SUC are taken as primitive constants with types num and num->num respectively. Peano's axioms are then asserted using new_axiom. The resulting theorems are:

```
NOT_SUC      |- !n. ~(SUC n = 0)

INV_SUC      |- !m n. (SUC m = SUC n) ==> (m = n)

INDUCTION    |- !P. P 0 /\ (!n. P n ==> P(SUC n)) ==> (!n. P n)
```

In higher-order logic, Peano's axioms are sufficient for developing number theory because addition and multiplication can be defined. In first order logic these must be taken as primitive. Note also that INDUCTION could not be stated as a single axiom in first-order logic because predicates (*e.g.* P) cannot be quantified.

5.9 The Theory prim_rec

In classical logic, unlike domain theory logics such as PPLAMBDA, 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. For any x and f the *Primitive Recursion Theorem* tells us that there is a unique function fun such that:

```
(fun 0 = x) /\
(!m. fun(SUC m) = f(fun m)m)
```

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:

```
|- !x f. ?fun.
    (fun 0 = x) /\
    (!m. fun(SUC m) = f(fun m)m)
```

From this it follows that there exists a function PRIM_REC such that:

```
|- !m x f.
    (PRIM_REC x f 0  = x) /\
    (PRIM_REC x f (SUC m) = f(PRIM_REC x f m)m)
```

The function PRIM_REC can be used to justify any primitive recursive definition. In higher-order logic a recursion of the form

```
fun 0 x1...xn        = f1(x1,...,xn)

fun (SUC m) x1...xn = f2(fun m x1...xn, m, x1,...,xn)
```

is equivalent to:

```
fun 0        = \x1...xn. f1(x1,...,xn)

fun (SUC m) = \x1...xn. f2(fun m x1...xn, m, x1,...,xn)
             = (\f m x1...xn. f2(f x1...xn, m, x1,...,xn))(fun m)m
```

which is equivalent to the non-recursive definition:

```
fun = PRIM_REC
        (\x1...xn. f1(x1,...,xn))
        (\f m x1...xn. f2(f x1...xn, m, x1,...,xn))
```

For example, we can define addition and multiplication by:

```
|- + = PRIM_REC(\n. n)(\f m n. SUC(f n))

|- * = PRIM_REC(\n. 0)(\f m n. (f n) + n)
```

To automate the use of the Primitive Recursion Theorem, HOL provides two functions:

```
new_prim_rec_definition       : (string # term) -> thm
new_infix_prim_rec_definition : (string # term) -> thm
```

Evaluating

```
new_prim_rec_definition
 (`fun`,
  "(!x1...xn. fun 0 x1...xn = t1[x1,...,xn]) /\
   (!m x1...xn.
     fun (SUC m) x1...xn = t2[fun m x1...xn, m, x1,...,xn])")
```

automatically makes the (non-recursive) definition:

```
new_definition
 (`fun_DEF`,
  "fun = PRIM_REC
          (\x1...xn. t1[x1,...,xn])
          (\fun m x1...xn. t2[fun x1...xn, m, x1,...,xn])")
```

and then proves the theorem:

```
|-  fun 0       x1...xn = t1[x1,...,xn] /\
    fun (SUC m) x1...xn = t2[fun m x1...xn, m, x1,...,xn]
```

which is saved as the theorem with name `fun`.

The ML function new_infix_prim_rec_definition declares an infixed function by primitive recursion. For example, here is the definition of + used by the system:

```
new_infix_prim_rec_definition
 (`ADD`,
  "($+ 0 n = n) /\
   ($+ (SUC m) n = SUC($+ m n))")
```

The $'s are there to indicate that + is being declared an infix. Evaluating this ML expression will create a definition of the following form in the current theory:

```
ADD_DEF   |- $+ = PRIM_REC(\n. n)(\f m n. SUC(f n))
```

The **$** is now necessary since the call to `new_infix_prim_rec_definition` declares + as an infix. It also automatically proves the following theorem:

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

which is saved in the current theory with name `ADD`.

We might hope to define the less-than relation "<" by primitive recursion, but it is needed for the proof of the Primitive Recursion Theorem, so we must somehow define it before we have primitive recursion available. A definition that works is:

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

Intuitively, this says that `m<n` if there exists a set (with characteristic function `P`) that is downward closed[11] and contains `m` but not `n`.

5.10 The Theory `arithmetic`

The theory `arithmetic` contains primitive recursive definitions of standard arithmetic operators. For example:

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

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

|- (0 * n = 0) /\ ((SUC m) * n = (m * n) + n)
```

It also contains various non-recursive definitions.

```
|- m > n   = n < m

|- m <= n  = m < n \/ (m = n)

|- m >= n  = m > n \/ (m = n)

|- m DIV n = (@x. (n * x) <= m /\ m < ((n * x) + n))

|- m REM n = m - (n * (m DIV n))
```

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

An *ad hoc* collection of elementary arithmetic theorems are pre-proved in the theory `arithmetic`. For example:

```
|- !m n. n < m ==> (?p. p + n = m)

|- !m n p. m < n /\ n < p ==> m < p

|- !m n. ~(m < n /\ n < m)

|- !m n. m * n = n * m

|- !m n p. (m + n) * p = (m * p) + (n * p)
```

At the time of writing there are about seventy such pre-proved theorems.

5.11 The Theory `list`

The theory `list` introduces the unary type operator `list` and the list processing functions:

```
NIL  : (*)list
CONS : * -> (*)list -> (*)list
HD   : (*)list -> *
TL   : (*)list -> (*)list
NULL : (*)list -> bool
```

These satisfy the usual axioms:

```
|- NULL NIL /\ !(x:*) l. ~(NULL(CONS x l))

|- !(x:*) l. HD(CONS x l) = x

|- !(x:*) l. TL(CONS x l) = l

|- !l:(*)list. CONS(HD l)(TL l) = l
```

These theorems could be proved from suitable definitions (which we don't give); the rule of structural induction can also be derived from these definitions.

The HOL parser allows one to write `[]` instead of `NIL` and `[`t_1`;`t_2`;`$\ldots$`;`t_n`]` instead of `CONS` t_1 `(CONS` $t_2 \cdots$ `(CONS` t_n `NIL)` $\cdots$ `)`.

5.12 The Theory `tok`

The theory `tok` (for token) introduces the types `char` and `tok` to represent characters and strings of characters respectively. The type `char` could be defined equivalent to a suitable subset of `num` (for example, representing ASCII codes) and the type `tok` could then be defined as equivalent to `(char)list`. In fact, characters and tokens are currently axiomatized rather than defined (but we plan to eventually change this). The HOL parser converts any expression of the form `'string'` into a constant of type `tok` (where *string* can be any string of characters not containing the quote `'`).

There are two constants in the theory `tok`:

```
EXPLODE : tok -> (char)list
IMPLODE : (char)list -> tok
```

`EXPLODE` converts a token to the list of characters it contains; `IMPLODE` is its inverse.

6 Derived Inference Rules

Derived inference rules are rules that can be justified on the basis of the axioms and primitive inference rules. For example, consider the following rule for 'undischarging' assumptions:

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

This is valid because if we have a theorem Γ `|-` t_1`==>`t_2 then we can derive the theorem $\Gamma \cup \{t_1\}$ `|-` t_2 by Modus Ponens from the theorem t_1 `|-` t_1 (which follows from the primitive rule `ASSUME`).

Derived inference rules enable proofs to be done using bigger and more natural steps. They can be defined in ML simply as functions that call the primitive inference rules. For example, the undischarging rule just described can be implemented by:

```
let UNDISCH th = MP th (ASSUME(fst(dest_imp(concl th))))
```

Each application of `UNDISCH` will invoke an application of `ASSUME` followed by an application of `MP`. Some of the predefined derived rules in HOL can invoke many thousands of primitive inference steps.

The HOL system has all the standard introduction and elimination rules of Predicate Calculus predefined as derived inference rules. It is these, rather than the primitive rules, that one normally uses in practice. In addition, there are some special rules that do a limited amount of automatic theorem-proving. The most generally useful examples of these are a collection of *rewriting* rules developed by Larry Paulson [17]. Rewriting rules use equations of the form `|-` t_1=t_2 to repeatedly replace subterms matching t_1 by the corresponding instances of t_2. The Cambridge LCF Manual [18] documents the rewriting rules in the HOL system, so we do not describe them here.

7 Goal Directed Proof: Tactics and Tacticals

A *tactic* is an ML function which is applied to a goal to reduce it to subgoals. A *tactical* is a (higher-order) ML function for combining tactics to build new tactics[12].

For example, 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 to a goal and then applies T_2 to each subgoal produced by T_1. The tactical `THEN` is an infixed ML function.

The tactics and tacticals in the HOL system are derived from those in the Cambridge LCF system [18] (which evolved from the ones in Edinburgh LCF [6]).

7.1 Tactics

It simplifies the description of tactics if we use various *type abbreviations*. A type abbreviation is just a name given to a type . A type and its abbreviation can be used interchangeably. The system prints types using any abbreviations that are in force. A type abbreviation is introduced by executing a declaration of the form

```
lettype name = type
```

For example:

```
lettype goal = term list # term
```

The following ML type abbreviations are also used in connection with tactics (they will be explained later).

```
proof       = thm list -> thm
subgoals    = goal list # proof
tactic      = goal -> subgoals
thm_tactic  = thm -> tactic
conv        = term -> thm
```

If T is a tactic and g is a goal, then applying T to g (*i.e.* evaluating the ML expression $T\ g$) will result in an object of ML type `subgoals`, *i.e.* a pair whose first component is a list of goals and whose second component has ML type `proof`.

Suppose $T\ g$ = ([g_1;...;g_n],p). The idea is that g_1 , ... , g_n are subgoals and p is a 'justification' of the reduction of goal g to subgoals g_1 , ... , g_n. Suppose further that we have solved the subgoals g_1 , ... , g_n. This would mean that we had somehow proved theorems th_1 , ... , th_n such that each th_i $(1 \leq i \leq n)$ 'achieves' the goal g_i. The justification p (produced by applying T to g) is an ML function which when applied to the list [th_1;...;th_n] returns a theorem, th, which

[12]The terms "tactic" and "tactical" are due to Robin Milner, who invented the concepts.

'achieves' the original goal g. Thus p is a function for converting a solution of the subgoals to a solution of the original goal. If p does this successfully, then the tactic T is called *valid*. Invalid tactics cannot result in the proof of invalid theorems; the worst they can do is result in insolvable goals or unintended theorems being proved. If T were invalid and were used to reduce goal g to subgoals g_1 , ... , g_n, then one might prove theorems th_1 , ... , th_n achieving g_1 , ... , g_n, only to find that these theorems are useless because p[th_1;...;th_n] doesn't achieve g (*i.e.* it fails, or else it achieves some other goal).

A theorem *achieves* a goal if the assumptions of the theorem are included in the assumptions of the goal *and* if the conclusion of the theorems is equal (up to renaming of bound variables) to the conclusion of the goal. More precisely, we say a theorem t_1, ... , t_m |- t achieves a goal ([u_1;...;u_n],u) if and only if {t_1,...,t_m} is a subset of {u_1,...,u_n} and t is equal to u (up to renaming of bound variables). For example, the goal (["x=y";"y=z";"z=w"],"x=z") is achieved by the theorem x=y, y=z |- x=z (the assumption "z=w" is not needed).

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

Tactics are specified using the following notation:

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

For example, a tactic called CONJ_TAC is described by

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

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

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

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

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

Tactics generally 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`

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

is the 'identity tactic'; it reduces a goal `(`Γ`,`t`)` to the single subgoal `(`Γ`,`t`)` — *i.e.* it has no effect. `ALL_TAC` is useful for writing complex tactic using tacticals (*e.g.* see the definition of `REPEAT` in Section 7.3).

7.2 Using Tactics to Prove Theorems

Suppose one has a goal g to solve. If g is simple one might be able to think up a tactic, T say, which reduces it to the empty list of subgoals. If this is the case then executing

```
let gl,p = T g
```

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

```
let th = p[]
```

After proving a theorem one usually wants to store it in the current theory. To do this one chooses an unused name, `Thm1` say, and then executes:

```
save_thm(`Thm1`, th)
```

This saves the theorem *th* with name `Thm1`. If the current theory is called $\mathcal{T}$ then to retrieve *th* (in a later session, in $\mathcal{T}$ or its descendants) one does:

```
let th = theorem `𝒯` `Thm1`
```

The ML function `theorem` has type `string->string->thm`. Its first argument should be the name of a theory and its second argument the name of a theorem on that theory. The named theorem is returned.

To simplify the use of tactics there is a standard function called `prove_thm`:

```
prove_thm : (string # term # tactic) -> thm
```

`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.

7.3 Tacticals

A *tactical* is an ML function that returns a tactic (or tactics) as result. Tacticals may take various parameters; this is reflected in the various ML types that the built-in tacticals have. Some important tactics in the HOL system are listed below (these are all in LCF also).

7.3.1 `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 first tries T_1 and then if T_1 fails it tries T_2. It is defined in ML by

```
let (T1 ORELSE T2) g =  T1 g ? T2 g
```

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

The tactical `THEN` is an ML infix. If T_1 and T_2 are tactics, then the ML expression T_1 `THEN` T_2 evaluates to a tactic which first applies T_1 and then applies T_2 to all the subgoals produced by T_1. Its definition in ML is tricky and not given here.

7.3.3 `THENL : tactic -> tactic list -> tactic`

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

7.3.4 `REPEAT : tactic -> tactic`

If T is a tactic then `REPEAT` T is a tactic which repeatedly applies T until it fails. The ML code defining `REPEAT` illustrates the elegance of programming with higher-order functions.

```
letrec REPEAT (T:tactic) = (T THEN REPEAT T) ORELSE ALL_TAC
```

7.3.5 `EVERY_ASSUM : (thm -> tactic) -> tactic`

Applying `EVERY_ASSUM` f to a goal `([`t_1`;`...`;`t_n`],`t`)` is equivalent to applying the tactic:

f`(ASSUME` t_1`) THEN ... THEN` f`(ASSUME` t_n`)`

7.3.6 FIRST_ASSUM : (thm -> tactic) -> tactic

Applying FIRST_ASSUM f to a goal ([t_1;...;t_n],t) is equivalent to applying the tactic:

f(ASSUME t_1) ORELSE ... ORELSE f(ASSUME t_n)

7.4 Tactics Built into HOL

We list below some of the tactics built into the HOL system, including those that are used in the example proof in Section 8.

Recall that the ML type thm_tactic abbreviates theorem->tactic, and the type conv[13] abbreviates term->thm.

7.4.1 ACCEPT_TAC : thm_tactic

- **Summary:** ACCEPT_TAC *th* is a tactic that solves any goal that is achieved by *th*.

- **Use:** Interfacing forward and backward 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 achieving these goals by forward proof. The tactic

  T THENL[ACCEPT_TAC th_1;...;ACCEPT_TAC th_n]

  would then solve g.

7.4.2 DISJ_CASES_TAC : thm_tactic

- **Summary:** DISJ_CASES_TAC |- u\/v splits a goal into two cases: one with "u" as an assumption and the other with "v" as an assumption.

$$\frac{t}{\{u\}t \quad \{v\}t}$$

- **Uses:** Case analysis. The tactic ASM_CASES_TAC (see below) 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.

[13]The type conv comes from Larry Paulson's theory of conversions [17].

7.4.3 `ASM_CASES_TAC : term -> tactic`

- **Summary:** `ASM_CASES_TAC` "u" does case analysis on the boolean term "u".

$$\frac{t}{\{u\}t \qquad \{\char`~u\}t}$$

- **Uses:** Case analysis.

7.4.4 `COND_CASES_TAC : tactic`

- **Summary:** Does a case split on the condition of a conditional term.

$$\frac{t[p\texttt{=>}u\,|\,v]}{\{p\texttt{=T}\}t[u] \qquad \{p\texttt{=F}\}t[v]}$$

  `COND_CASES_TAC` searches for a conditional term and then does cases on its if-part. It fails if the context $t[]$ captures any free variables in the if-part p.

- **Uses:** Most useful when there is only one conditional term in the goal (otherwise it is difficult to predict which condition will be chosen for the case split). Successive case analysis on all conditions can be done using `REPEAT COND_CASES_TAC`.

7.4.5 `REWRITE_TAC : thm list -> tactic`

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

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

  where t' is obtained from t by rewriting with

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

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

- **Other rewriting tactics** (based on REWRITE_TAC):
  1. ASM_REWRITE_TAC adds the assumptions of the goal to the list of theorems used for rewriting.
  2. 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 built-in rewrites in the ML variable basic_rewrites.
  3. PURE_ASM_REWRITE_TAC is like ASM_REWRITE_TAC, but it doesn't use any built-in rewrites.
  4. PURE_REWRITE_TAC uses neither the assumptions nor the built-in rewrites

7.4.6 ASSUME_TAC : thm_tactic

- **Summary:** ASSUME_TAC |-u adds u as an assumption.

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

- **Uses:** Enriching the assumptions of a goal with previously proved theorems.

7.4.7 CONJ_TAC : tactic

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

$$\frac{t_1 \;/\backslash\; t_2}{t_1 \qquad t_2}$$

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

7.4.8 DISCH_TAC : tactic

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

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

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

7.4.9 GEN_TAC : tactic

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

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

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

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

7.4.10 IMP_RES_TAC : tactic

- **Summary:** IMP_RES_TAC *th* 'resolves' (see below) *th* with the assumptions of the goal and then adds the results to the assumptions.

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

where u_1, ... , u_n are derived by 'resolving' *th* with t_1, ... , t_m. Resolution in HOL is not classical resolution, but just Modus Ponens with a bit of one-way pattern matching (not unification). The usual case is where *th* is of the form

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

IMP_RES_TAC *th* then tries to specialize x_1, ... , x_p so that v_1, ... , v_q match members of $\{t_1,\ldots,t_m\}$. If such a match is found then the appropriate instance of v is added to the assumptions, together with all appropriate instances of v_i==>...v_n==>v $(2 \leq i \leq n)$. IMP_RES_TAC can also be given a conjunction of implications in which case it will do 'resolution' with each of the conjuncts. In fact, it applies a canonicalization rule to its argument to split it into a list of theorems. Each theorem produced by this canonicalization process is resolved with the assumptions. Full details can be found in the Cambridge LCF Manual [18].

- **Uses:** Deriving new assumptions from existing ones and previously proved theorems so that subsequent tactics (*e.g.* `ASM_REWRITE_TAC`) have more to work with.

7.4.11 `STRIP_TAC : tactic`

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

7.4.12 `SUBST_TAC : thm list -> thm`

- **Summary:** `SUBST_TAC[|-`u_1`=`v_1`;`$\ldots$`;|-`u_n`=`v_n`]` converts a goal of the form $t[u_1,\ldots,u_n]$ to a subgoal of the form $t[v_1,\ldots,v_n]$.
- **Uses:** To make replacements for terms in situation in which `REWRITE_TAC` is too general or would loop.

7.4.13 `ALL_TAC : tactic`

- **Summary:** Identity tactic for the tactical `THEN` (see end of Section 7.1).
- **Uses:**
    1. Writing tacticals (see description of `REPEAT` in Section 7.3).
    2. With `THENL`; for example, if tactic T produces two subgoals and we want to apply T_1 to the first one but to do nothing to the second, then the tactic to use is T `THENL[`T_1`;ALL_TAC]`.

7.4.14 `NO_TAC : tactic`

- **Summary:** Tactic that always fails.
- **Uses:** Writing tacticals (see the example in Section 8).

8 An Example of HOL in Action

The example in this section has been chosen to give a flavour of what it is like to use the HOL system. Although the theorems proved are simple, the way we prove them illustrates the kind of intricate 'proof engineering' that is typical. The proofs below could be done more elegantly, but presenting them that way would defeat our purpose of illustrating various features of HOL. We have tried to use a small example to give the reader a feel for what it is like to do a big one. Readers who are not interested in hardware verification should be able to learn something about the HOL system even if they do not wish to penetrate the details of the parity-checking example we use.

As in Section 2, the boxed interactions below should be understood as occurring in sequence. These interactions comprise the specification and verification of a device that computes the parity of a sequence of bits. More specifically, we verify the implementation of a device with an input `in` and an output `out` with the specification that the *n*th output on `out` is `T` if and only if there have been an even number of `T`'s input on `in`. We shall construct a new theory called `Parity` in which we specify and verify the device. The first thing we do is start up the HOL system and then enter draft mode for this theory.

```
% hol


-  -         --        -
|__|        |  |       |
|  | IGHER  |__| RDER  |__ OGIC
===============================
(Built on Sept  31)

#new_theory`Parity`;;
() : void
```

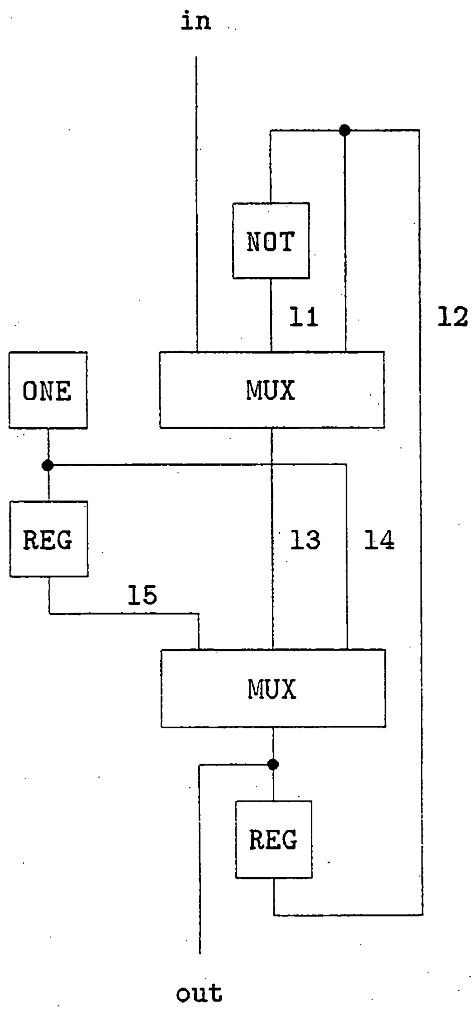
To specify the device we define a primitive recursive function `PARITY` such that `PARITY n f` is true if the number of `T`'s in the sequence `f 0`, ... , `f n` is even.

```
#let PARITY_DEF =
# new_prim_rec_definition
#  (`PARITY_DEF`,
#   "(PARITY 0 f = T) /\
#    (PARITY(SUC n)f = (f(SUC n) => ~(PARITY n f) | PARITY n f))");;
PARITY_DEF =
|- (PARITY 0 f = T) /\
   (PARITY(SUC n)f = (f(SUC n) => ~PARITY n f | PARITY n f))
```

The effect of `new_prim_rec_definition` is to store the definition of the constant `PARITY` on the theory `Parity` and to bind the defining theorem to the ML variable `PARITY`. The specification of the device can now be given as:

```
!t. out t = PARITY t in
```

The schematic diagram below shows the design of a device that is intended to implement this specification.



Intuitively, this works by storing the parity of the sequence input so far in the lower of the two registers. Each time `T` is input at `in`, this stored value is complemented. Registers are assumed to 'power up' in a state in which they are storing `F`. The second register (connected to `ONE`) initially outputs `F` and then outputs `T` forever. Its role is just to ensure that the device works during the first cycle by connecting the output `out` to the device `ONE` via the lower multiplexer. For all subsequent cycles out is connected to 13 and so either carries the stored parity value (if the current input is F) or the complement of this value (if the current input is T).

The devices making up this schematic will be modelled with predicates [8]. For example, the predicate ONE is true of a signal[14] out if for all times t, the value of out is T.

```
#let ONE_DEF =
# new_definition
#  (`ONE_DEF`, "ONE(out:num->bool) = !t. out t = T");;
ONE_DEF = |- ONE out = (!t. out t = T)
```

Note that "ONE_DEF" is used both as an ML variable and as the name of the definition in the theory Parity.

The binary predicate NOT is true of a pair of signals (in,out) if the value of out is always the negation of the value of in. We thus model inverters as having no delay. This is appropriate for a register-transfer level model, but would be wrong at a lower level.

```
#let NOT_DEF =
# new_definition
#  (`NOT_DEF`, "NOT(in,out:num->bool) = !t. out t = ~(in t)");;
NOT_DEF = |- NOT(in,out) = (!t. out t = ~in t)
```

The final combinational device we need is a multiplexer. This is a 'hardware conditional'; the input sw selects which of the other two inputs are to be connected to the output out.

```
#let MUX_DEF =
# new_definition
#  (`MUX_DEF`,
#   "MUX(sw,in1,in2,out:num->bool) =
#     !t. out t = (sw t => in1 t | in2 t)");;
MUX_DEF = |- MUX(sw,in1,in2,out) = (!t. out t = (sw t => in1 t | in2 t))
```

The remaining devices in the schematic are registers. These are unit-delay elements; the values output at time t+1 are the values input at the preceding time t, except at time 0 when the register outputs F[15] .

[14]Signals are modelled as functions from numbers (representing times) to booleans.

[15]Time 0 represents when the device is switched on.

```
#let REG_DEF =
# new_definition
# (`REG_DEF`, "REG(in,out:num->bool) =
#              !t. out t = ((t=0) => F | in(t-1))");;
REG_DEF = |- REG(in,out) = (!t. out t = ((t = 0) => F | in(t - 1)))
```

The schematic diagram above can be represented as a predicate by conjoining the relations holding between the various signals and then existentially quantifying the internal lines. This techniques is explained elsewhere (*e.g.* see [8], [1]).

```
#let PARITY_IMP_DEF =
# new_definition
#  (`PARITY_IMP_DEF`,
#   "PARITY_IMP(in,out) =
#    ?l1 l2 l3 l4 l5.
#     NOT(l2,l1) /\ MUX(in,l1,l2,l3) /\ REG(out,l2) /\
#     ONE l4     /\ REG(l4,l5)       /\ MUX(l5,l3,l4,out)");;
PARITY_IMP_DEF =
|- PARITY_IMP(in,out) =
   (?l1 l2 l3 l4 l5.
     NOT(l2,l1) /\
     MUX(in,l1,l2,l3) /\
     REG(out,l2) /\
     ONE l4 /\
     REG(l4,l5) /\
     MUX(l5,l3,l4,out))
```

What we shall prove is

```
|- !in out. PARITY_IMP(in,out) ==> (!t. out t = PARITY t in)
```

This states that *if* in and out are related as in the schematic diagram (*i.e.* as in the definition of PARITY_IMP), *then* the pair of signals (in,out) satisfies the specification.

To assist in the proof, it is convenient to make the following auxiliary definition.

```
#let PARITY_BODY_DEF =
# new_definition
#  (`PARITY_BODY_DEF`,
#   "PARITY_BODY(in,out,l1,l2,l3,l4,l5) =
#     NOT(l2,l1) /\ MUX(in,l1,l2,l3) /\ REG(out,l2) /\
#     ONE l4     /\ REG(l4,l5)       /\ MUX(l5,l3,l4,out)");;
PARITY_BODY_DEF =
|- PARITY_BODY(in,out,l1,l2,l3,l4,l5) =
   NOT(l2,l1) /\
   MUX(in,l1,l2,l3) /\
   REG(out,l2) /\
   ONE l4 /\
   REG(l4,l5) /\
   MUX(l5,l3,l4,out)
```

We shall start by proving the following lemma:

```
|- !in out.
     PARITY_BODY(in,out,l1,l2,l3,l4,l5) ==>
      (out 0 = T) /\ !t. out(SUC t) = (in(SUC t) => ~(out t) | out t)
```

We will prove this interactively using HOL's subgoal package[16]. We start the proof by putting the goal we want to prove on a goal stack using the function set_goal which takes a goal as argument.

```
#set_goal
# ([], "!in out.
#          PARITY_BODY(in,out,l1,l2,l3,l4,l5) ==>
#          (out 0 = T) /\
#          !t. out(SUC t) = (in(SUC t) => ~(out t) | out t)");;
"!in out.
  PARITY_BODY(in,out,l1,l2,l3,l4,l5) ==>
  (out 0 = T) /\ (!t. out(SUC t) = (in(SUC t) => ~out t | out t))"
```

The subgoal package prints out the goal on the top of the goal stack. We now expand this top goal by rewriting with the definitions PARITY_BODY_DEF, ONE_DEF, NOT_DEF, MUX_DEF and REG_DEF. The ML function expand takes a tactic and applies it to the top goal; the resulting subgoals are pushed on to the goal stack. The message "OK.." is printed out just before the tactic is applied. Only one subgoal is produced by the rewriting tactics (see Section 7.4.5).

```
#expand
# (PURE_REWRITE_TAC[PARITY_BODY_DEF;ONE_DEF;NOT_DEF;MUX_DEF;REG_DEF]);;
OK..
"!in out.
  (!t. l1 t = ~l2 t) /\
  (!t. l3 t = (in t => l1 t | l2 t)) /\
  (!t. l2 t = ((t = 0) => F | out(t - 1))) /\
  (!t. l4 t = T) /\
  (!t. l5 t = ((t = 0) => F | l4(t - 1))) /\
  (!t. out t = (l5 t => l3 t | l4 t)) ==>
  (out 0 = T) /\ (!t. out(SUC t) = (in(SUC t) => ~out t | out t))"
```

We proceed by first removing the quantifiers (with GEN_TAC), then moving the conjuncts of the antecedant into the assumptions (using STRIP_TAC) and finally, splitting the remaining conjunction into its two conjuncts (with CONJ_TAC). We can do all this in one step using a compound tactic built with the tacticals REPEAT and THEN.

[16]The subgoal package is part of Cambridge LCF, but not Edinburgh LCF. It was implemented by Larry Paulson and provides proof building tools similar to those found in Stanford LCF [14]. We do not describe the subgoal package in this paper, but hope that the simple uses of it that we make will be clear.

```
#expand(REPEAT GEN_TAC THEN STRIP_TAC THEN CONJ_TAC);;
OK..
2 subgoals
"!t. out(SUC t) = (in(SUC t) => ~out t | out t)"
    [ "!t. l1 t = ~l2 t" ]
    [ "!t. l3 t = (in t => l1 t | l2 t)" ]
    [ "!t. l2 t = ((t = 0) => F | out(t - 1))" ]
    [ "!t. l4 t = T" ]
    [ "!t. l5 t = ((t = 0) => F | l4(t - 1))" ]
    [ "!t. out t = (l5 t => l3 t | l4 t)" ]

"out 0 = T"
    [ "!t. l1 t = ~l2 t" ]
    [ "!t. l3 t = (in t => l1 t | l2 t)" ]
    [ "!t. l2 t = ((t = 0) => F | out(t - 1))" ]
    [ "!t. l4 t = T" ]
    [ "!t. l5 t = ((t = 0) => F | l4(t - 1))" ]
    [ "!t. out t = (l5 t => l3 t | l4 t)" ]
```

The assumptions of the two subgoals are shown in square brackets. The last goal printed is the one on the top of the stack. We would like to expand the top goal (*i.e.* the second one listed above) by rewriting with the assumptions. However, if we do this the system will go into an infinite loop because the assumptions of this goal are mutually recursive. To prevent this, we must be more delicate and only rewrite with a non-looping subset of the assumptions.

To enable the assumptions corresponding to particular lines to be selected for rewriting, we define an ML function `lines` such that `lines` \`$l_1 \ldots l_n$\` t is true if t has the form `"!t.` l_i `t = ···"` for some l_i in the set specified by the string \`$l_1 \ldots l_n$\`. The functions `words` and `rator` used below are explained in the examples in Section 2.

```
#let lines tok t =
# (let x = fst(dest_var(rator(lhs(snd(dest_forall t)))))
#  in
#  mem x (words tok)) ? false;;
lines = - : (string -> term -> bool)
```

The tactic `FILTER_ASM_REWRITE_TAC(lines`\``out l1 l3 l4 l5`\``)[]` rewrites with only those assumptions that involve `out`, `l1`, `l3`, `l4` and `l5` (see Section 7.4.5).

```
#expand(FILTER_ASM_REWRITE_TAC(lines`out l1 l3 l4 l5`)[]);;
OK..
goal proved
..... |- out 0 = T

Previous subproof:
"!t. out(SUC t) = (in(SUC t) => ~out t | out t)"
    [ "!t. l1 t = ~l2 t" ]
    [ "!t. l3 t = (in t => l1 t | l2 t)" ]
    [ "!t. l2 t = ((t = 0) => F | out(t - 1))" ]
    [ "!t. l4 t = T" ]
    [ "!t. l5 t = ((t = 0) => F | l4(t - 1))" ]
    [ "!t. out t = (l5 t => l3 t | l4 t)" ]
```

The first of the two subgoals is proved, so the system backs up to the second (and last) subgoal. We start working on this by first stripping off the quantifiers using GEN_TAC.

```
#expand GEN_TAC;;
OK..
"out(SUC t) = (in(SUC t) => ~out t | out t)"
    [ "!t. l1 t = ~l2 t" ]
    [ "!t. l3 t = (in t => l1 t | l2 t)" ]
    [ "!t. l2 t = ((t = 0) => F | out(t - 1))" ]
    [ "!t. l4 t = T" ]
    [ "!t. l5 t = ((t = 0) => F | l4(t - 1))" ]
    [ "!t. out t = (l5 t => l3 t | l4 t)" ]
```

Inspecting this goal we see that it will be solved if we expand `out(SUC t)` using the assumption `!t. out t = (l5 t => l3 t | l4 t)`. If we just rewrite with this assumption, then all the subterms of the form `out t` will also be expanded. To prevent this we are forced to use the messy and *ad hoc* tactic shown below.

```
FIRST_ASSUM
 (\th. if lines`out`(concl th)
        then SUBST_TAC[SPEC "SUC t" th]
        else NO_TAC)
```

FIRST_ASSUM is explained in Section 7.3.6. The function it is applied to converts a theorem of the form `|- !t. out t = ··· t ···` to a tactic that replaces `out(SUC t)` by `··· SUC t ···`; it maps all other theorems to NO_TAC, a tactic that always fails.

```
#expand
# (FIRST_ASSUM
#   (\th. if lines`out`(concl th)
#          then SUBST_TAC[SPEC "SUC t" th]
#          else NO_TAC));;
OK..
"(l5(SUC t) => l3(SUC t) | l4(SUC t)) = (in(SUC t) => ~out t | out t)"
    [ "!t. l1 t = ~l2 t" ]
    [ "!t. l3 t = (in t => l1 t | l2 t)" ]
    [ "!t. l2 t = ((t = 0) => F | out(t - 1))" ]
    [ "!t. l4 t = T" ]
    [ "!t. l5 t = ((t = 0) => F | l4(t - 1))" ]
    [ "!t. out t = (l5 t => l3 t | l4 t)" ]
```

The fact that we had to resort to this messy use of `FIRST_ASSUM` illustrates both the strengths and weaknesses of the HOL system. Trivial deductions sometimes require elaborate tactics, but on the other hand one never reaches an impasse. HOL experts can prove arbitrarily complicated theorems if they are willing to use sufficient ingenuity. Furthermore, the type discipline ensures that no matter how complicated and *ad hoc* are the tactics, it is impossible to prove an invalid theorem.

Inspecting the goal above, we see that the next step is to unwind the equations for lines `l1`, `l3`, `l4` and `l5` and then, when this is done, unwind with the equation for line `l2`.

```
#expand(FILTER_ASM_REWRITE_TAC(lines`l1 l3 l4 l5`)[]
#       THEN FILTER_ASM_REWRITE_TAC(lines`l2`)[]);;
OK..
"(((SUC t = 0) => F | T) =>
  (in(SUC t) =>
   ~((SUC t = 0) => F | out((SUC t) - 1)) |
   ((SUC t = 0) => F | out((SUC t) - 1))) |
  T) =
 (in(SUC t) => ~out t | out t)"
    [ "!t. l1 t = ~l2 t" ]
    [ "!t. l3 t = (in t => l1 t | l2 t)" ]
    [ "!t. l2 t = ((t = 0) => F | out(t - 1))" ]
    [ "!t. l4 t = T" ]
    [ "!t. l5 t = ((t = 0) => F | l4(t - 1))" ]
    [ "!t. out t = (l5 t => l3 t | l4 t)" ]
```

This goal can now be solved by rewriting with two standard theorems:

```
NOT_SUC      |- !n. ~(SUC n = 0)
SUC_SUB1     |- !m. (SUC m) - 1 = m
```

```
#expand(REWRITE_TAC[NOT_SUC;SUC_SUB1]);;
OK..
goal proved
|- (((SUC t = 0) => F | T) =>
     (in(SUC t) =>
      ~((SUC t = 0) => F | out((SUC t) - 1)) |
      ((SUC t = 0) => F | out((SUC t) - 1))) |
     T) =
    (in(SUC t) => ~out t | out t)
..... |- (l5(SUC t) => l3(SUC t) | l4(SUC t)) =
          (in(SUC t) => ~out t | out t)
...... |- out(SUC t) = (in(SUC t) => ~out t | out t)
...... |- !t. out(SUC t) = (in(SUC t) => ~out t | out t)
|- !in out.
     (!t. l1 t = ~l2 t) /\
     (!t. l3 t = (in t => l1 t | l2 t)) /\
     (!t. l2 t = ((t = 0) => F | out(t - 1))) /\
     (!t. l4 t = T) /\
     (!t. l5 t = ((t = 0) => F | l4(t - 1))) /\
     (!t. out t = (l5 t => l3 t | l4 t)) ==>
     (out 0 = T) /\ (!t. out(SUC t) = (in(SUC t) => ~out t | out t))
|- !in out.
     PARITY_BODY(in,out,l1,l2,l3,l4,l5) ==>
     (out 0 = T) /\ (!t. out(SUC t) = (in(SUC t) => ~out t | out t))

Previous subproof:
goal proved
```

The goal is proved, *i.e.* the empty list of subgoals is produced. The system now applies the justification functions (see Section 7.1) produced by the tactics to the lists of theorems achieving the subgoals (starting with the empty list). These theorems are printed out in the order they are generated (note that assumptions of theorems are printed as dots).

We name the theorem just proved PARITY_LEMMA and save it in the current theory.

```
#save_top_thm`PARITY_LEMMA`;;
|- !in out.
     PARITY_BODY(in,out,l1,l2,l3,l4,l5) ==>
     (out 0 = T) /\ (!t. out(SUC t) = (in(SUC t) => ~out t | out t))

#let PARITY_LEMMA = it;;
PARITY_LEMMA =
|- !in out.
     PARITY_BODY(in,out,l1,l2,l3,l4,l5) ==>
     (out 0 = T) /\ (!t. out(SUC t) = (in(SUC t) => ~out t | out t))
```

We could have proved PARITY_LEMMA in one step with a single compound tactic. To illustrate this we set up the goal again.

```
#set_goal
# ([], "!in out.
#         PARITY_BODY(in,out,l1,l2,l3,l4,l5) ==>
#         (out 0 = T) /\
#         !t. out(SUC t) = (in(SUC t) => ~(out t) | out t)");;
"!in out.
  PARITY_BODY(in,out,l1,l2,l3,l4,l5) ==>
  (out 0 = T) /\ (!t. out(SUC t) = (in(SUC t) => ~out t | out t))"
```

We then expand with a single tactic corresponding to the sequence of tactics that we used interactively.

```
#expand
  (PURE_REWRITE_TAC[PARITY_BODY_DEF;ONE_DEF;NOT_DEF;MUX_DEF;REG_DEF]
    THEN REPEAT GEN_TAC
    THEN STRIP_TAC
    THEN CONJ_TAC
    THENL
     [FILTER_ASM_REWRITE_TAC(lines`out l1 l3 l4 l5`)[];GEN_TAC]
    THEN FIRST_ASSUM
          (\th. if lines`out`(concl th)
                 then SUBST_TAC[SPEC "SUC t" th]
                 else NO_TAC)
    THEN FILTER_ASM_REWRITE_TAC(lines`l1 l3 l4 l5`)[]
    THEN FILTER_ASM_REWRITE_TAC(lines`l2`)[]
    THEN REWRITE_TAC[NOT_SUC;SUC_SUB1]);;
OK..
goal proved
|- !in out.
    PARITY_BODY(in,out,l1,l2,l3,l4,l5) ==>
    (out 0 = T) /\ (!t. out(SUC t) = (in(SUC t) => ~out t | out t))
```

Armed with PARITY_LEMMA, we can now move quickly towards the final theorem. First we prove:

```
  |- !in out.
      PARITY_BODY(in,out,l1,l2,l3,l4,l5) ==> (!t. out t = PARITY t in)
```

We will do this proof in one step using the ML function prove_thm described in Section 7.2.

```
#let PARITY_THM =
# prove_thm
#  (`PARITY_THM`,
#   "!in out.
#     PARITY_BODY(in,out,l1,l2,l3,l4,l5) ==> (!t. out t = PARITY t in)",
#   REPEAT GEN_TAC
#    THEN STRIP_TAC
#    THEN INDUCT_TAC
#    THEN IMP_RES_TAC PARITY_LEMMA
#    THEN ASM_REWRITE_TAC[PARITY]);;
PARITY_THM =
|- !in out.
    PARITY_BODY(in,out,l1,l2,l3,l4,l5) ==> (!t. out t = PARITY t in)
```

This proof consists of first removing the quantifiers (with GEN_TAC), then moving the antecedant of the implication to the assumptions (with STRIP_TAC), then doing mathematical induction on the term "t", then resolving with PARITY_LEMMA and finally rewriting with the assumptions and the theorem PARITY.

We can use PARITY_THM to prove the correctness of the device. We strip off the quantifiers, then rewrite with PARITY_IMP_DEF and SYM(PARITY_BODY_DEF)[17], then we apply STRIP_TAC followed by resolution with PARITY_THM and finally we wrewrite with the assumptions.

```
#let PARITY_CORRECT =
# prove_thm
#  (`PARITY_CORRECT`,
#   "!in out. PARITY_IMP(in,out) ==> (!t. out t = PARITY t in)",
#   REPEAT GEN_TAC
#    THEN REWRITE_TAC[PARITY_IMP_DEF;SYM(PARITY_BODY_DEF)]
#    THEN REPEAT STRIP_TAC
#    THEN IMP_RES_TAC PARITY_THM
#    THEN ASM_REWRITE_TAC[]);;
PARITY_CORRECT =
|- !in out. PARITY_IMP(in,out) ==> (!t. out t = PARITY t in)

#close_theory();;
() : void
```

This completes the proof of the parity checking device.

9 Conclusion

The example in the preceding section, though simple, is representative of larger scale proofs using the HOL system. For descriptions of such proofs see Avra Cohn's

[17]SYM is a derived rule that reverses an equation.

paper on the verification of the major state machine of the Viper microprocessor [3], Jeff Joyce's paper on the verification of a simple microcoded computer [11], John Herbert's proof of the ECL chip of the Cambridge Fast Ring [10] and Tom Melham's proof of a simple local area network [13]. All of these examples show that HOL can be used to prove non-trivial digital systems correct.

Current research at Cambridge is looking at new ways of modelling hardware in higher-order logic (*e.g.* the representation of low-level circuit behaviour); ways of expressing and relating behavioural abstractions; and methods of executing formal specification (for animation and simulation).

In addition, we are trying to make the HOL technology available to the industrial community. This involves reimplementing the research prototye system to make it acceptably efficient and rugged. It also involves preparing documentation and tutorial material.

References

[1] A. J. Camilleri, T. F. Melham and M. J. C. Gordon, *Hardware Verification Using Higher-Order Logic*, University of Cambridge Computer Laboratory, Technical Report No. 91, 1986.

[2] A. Church, *A Formulation of the Simple Theory of Types*, Journal of Symbolic Logic 5, 1940.

[3] A. J. Cohn, *A Proof of Correctness of the Viper Microprocessor: The First Level*, Proceedings of the Calgary Hardware Verification Workshop, Calgary, Canada, 12-16 January 1987.

[4] G. Cousineau, G. Huet and L. Paulson, *The* ML *Handbook*, INRIA, 1986.

[5] M. Gordon, R. Milner L., Morris, M. Newey and C. Wadsworth, *A Metalanhuage for Interactive proof in LCF*, Fifth ACM SIGACT-SIGPLAN Conference on *Principles of Programming Languages*, Tucson, Arizona, 1978.

[6] M. Gordon, R. Milner and C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, Lecture Notes in Computer Science, Springer-Verlag, 1979.

[7] M. Gordon, *HOL: A Machine Oriented Formulation of Higher-Order Logic*, University of Cambridge Computer Laboratory, Technical Report No. 68, 1985.

[8] M. Gordon, *Why Higher-order Logic is a Good Formalism for Specifying and Verifying Hardware.* In: *Formal Aspect of VLSI Design*, edited by G. Milne and P. A. Subrahmanyam, North-Holland, 1986.

[9] F. K. Hanna and N. Daeche, *Specification and Verification Using Higher-Order Logic.* In: *Formal Aspect of VLSI Design*, edited by G. Milne and P. A. Subrahmanyam, North-Holland, 1986.

[10] J. Herbert, Ph.D. Thesis, University of Cambridge, to appear 1987.

[11] J. J. Joyce, G. Birtwistle and M. Gordon, *Verification and Implementation of a Microprocessor*, To appear in: Proceedings of the Calgary Hardware Verification Workshop, Calgary, Canada, 12-16 January 1987, Also to appear as University of Cambridge Computer Laboratory, Technical Report No. 100, 1986.

[12] A. Leisenring, *Mathematical Logic and Hilbert's ϵ-Symbol*, Macdonald & Co. Ltd., London, 1969.

[13] T. Melham, Ph.D. Thesis, University of Cambridge, to appear.

[14] R. Milner, *Implementation and Application of Scott's Logic for Computable Functions*, Proceedings of the ACM Conference on *Proving Assertions about Programs*, SIGPLAN notices 7,1, 1972.

[15] R. Milner, *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences, 17, 1978.

[16] R. Milner, *A Proposal for Standard ML*, Proceedings of the *1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, 1984.

[17] L. Paulson, *A Higher-Order Implementation of Rewriting*, Science of Computer Programming 3, 119-149, 1983.

[18] L. Paulson, *Interactive Theorem Proving with Cambridge LCF*, Cambridge University Press, To Appear 1987.