A *brief* introduction to Higher Order Logic and the HOL proof assistant

Monica Nesi

Course on Formal Methods 2010-2011

Laurea Magistrale in Informatica

Facoltà di Scienze MM. FF. NN

Università degli Studi di L'Aquila

This note is taken from Appendix A of the technical report "Formalising Process Calculi in Higher Order Logic" by Monica Nesi, T.R. No. 411, Computer Laboratory, University of Cambridge, January 1997.

Chapter 1

Introduction

Higher order logic and the general purpose theorem prover HOL [6] can be used to develop interactive proof environments for supporting reasoning about various theories and their applications. Typically, the aim is to build a proof environment based on theorem proving which: (i) is logically sound, (ii) allows meta-reasoning to be carried out, (iii) is interactive but allows automation whenever possible, (iv) can be used in a practical and effective way.

Higher order logic is a good formalism for mechanising other mathematical languages because it is both powerful and general enough to allow sound and practical formulations. Several logics have been mechanised in higher order logic [5] and the theorem proving system HOL is used in these mechanisations. The HOL system was originally developed by Gordon [4] for reasoning about hardware systems, but in the following years the range of its applications has widened considerably. The HOL proof assistant is now used for mechanised theorem proving in many areas, including design and verification of critical and real-time systems, program refinement, program correctness, compiler verification and concurrency.

The HOL logic is a version of (classical) higher order logic based on Church's simple theory of types [1]. The interface to the logic is the functional programming language ML [2]. The approach to theorem proving in HOL is based on the LCF methodology [9] for interactive and secure theorem proving by mechanising the logic in a strongly typed language like ML. Theorems are represented by abstract data types and the user can interact with the theorem prover by ML procedures which operate on such data types. Theorem proving tools are ML functions and user-defined ML programs can only perform valid logical inferences. Typically, a purely definitional approach is taken when using the HOL logic. This means that new entities are only introduced by means of (primitive and derived) definition mechanisms which allow one to extend the logic in a sound way. Propositions and theorems are then derived from definitions and/or previously proved theorems by formal proof. This guarantees that inconsistencies are not introduced into the logic.

The HOL system can be used in two ways: (I) for proving theorems directly in

the HOL logic whenever higher order logic is a suitable specification language, e.g. in hardware verification [3]; (II) as a theorem proving environment to support reasoning about other formalisms.

According to the former approach, a formal system is mechanised in HOL by translating its syntactic objects into appropriate denotations in higher order logic. In this way, the embedded system inherits a certain amount of syntactic infrastructure from the HOL logic, such as λ-abstraction and β-reduction. This approach allows one to build an environment suitable for reasoning about applications of the embedded system, but not for meta-reasoning about the embedded system itself. In fact, meta-theorems about the formalised system cannot be proved in the HOL logic.

The latter approach allows for both reasoning about applications and meta-reasoning about the mechanised system. The formalism to be represented in HOL is not translated into terms of higher order logic, but is encoded as a type in the logic. These types are objects within the logic which can be referred to, properties about them can be expressed and meta-theorems about the embedded language can be proved.

Chapter 2

The HOL System

This section contains a brief introduction to the HOL proof assistant. The aim is to present some basic notions of higher order logic, definition mechanisms and theorem proving infrastructure. A detailed description of the HOL system can be found in [6]. The version of the system under consideration is the original one, referred to as HOL88, where the so-called 'Classic ML' (an early version of ML derived from LCF [2]) is adopted as the meta-language. In the following the term 'HOL' will be used to denote both the version of higher order logic implemented in the HOL system and the theorem proving environment.

2.1 The Meta-Language ML

ML is an interactive programming language. At top level one can evaluate expressions and perform declarations. The result of evaluating an expression is its value and type being printed; making a declaration results in a value being bound to a name.

In what follows, HOL sessions will be displayed through boxes in sequence, each box representing some interaction steps with the HOL system via the ML language. The ML prompt is `#`, thus lines beginning with `#` show text typed by the user, which is always terminated by a double semicolon '`;;`'. The other lines in the boxes show the system's response.

```
#‘hello!‘;;
‘hello!‘ : string
```

The ML expression `‘hello!‘` is a string, i.e. a sequence of characters enclosed between string quotes. Its type is `string`.

A declaration `let` x `=` e evaluates the expression e and binds the resulting value to the name x. The value of the last expression evaluated is always recorded in the variable `it` (note that declarations do not affect the value of `it`):

```
#let x = 3 * 4;;
x = 12 : int

#it;;
‘hello!‘ : string
```

The general form of declaration `let` x_1 `=` e_1 `and` $\cdots$ `and` x_n `=` e_n results in binding the value of each expression e_i to the name x_i. A declaration d can be made local to the evaluation of an expression e by evaluating the expression d `in` e.

```
#let x = 3 and y = 4 in x * y;;
12 : int

#it;;
12 : int
```

A declaration `let` $f\ x$ `=` e defines a function f with formal parameter x and body e.

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

#f 4;;
16 : int
```

Functions are printed as a dash followed by their type, because a function as such is not printable. The application of a function f to an argument x can be written as $f\,x$, even though the notation $f(x)$ is also allowed. Functions can also be curried and partially applied.

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

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

#g 4;;
12 : int
```

Functions can be written as λ-abstractions. The expression `\`x`.`e evaluates to a function with formal parameter x and body e. The declaration `let` f `=` `\`x`.`e is thus equivalent to `let` $f\ x$ `=` e.

```
#\x. x * x;;
- : (int -> int)

#it 4;;
16 : int
```

The ML type checker is able to infer the type of expressions if there is enough information.

```
#[3; 4];;
[3; 4] : int list

#tl it;;
[4] : int list
```

The ML expression `[3; 4]` is a list of integers and its type is `int list`, where `list` is a unary type constructor. The function `tl` takes a non-empty list and returns the tail of that list.

ML types can be *polymorphic*, i.e. they can contain type variables (denoted by `*`, `**`, etc. in HOL notation). The type `list` is indeed polymorphic and so are functions operating on lists, e.g. `tl: * list -> * list`. In the above example, the type variable `*` is instantiated to type `int`, thus getting the particular type `int list`.

Besides `list`, type constructors include `#` for pairs (product type) and `+` for the disjoint union of types. These are both binary type constructors, `* # **` and `* + **`, and are provided with primitive functions for performing typical operations on such types. Among others are the ML functions `fst: (* # **) -> *` and `snd: (* # **) -> **` for extracting the first and second components of a pair, and `inl: * -> (* + **)` and `outl: (* + **) -> *` for injecting and projecting the left summand of a disjoint union.

2.2 Higher Order Logic

The formulation of higher order logic in HOL is based on an extension of Church's simple theory of types [1]. The standard predicate calculus is extended in the HOL logic by allowing variables to range over functions and predicates, and arguments of functions can themselves be functions (hence higher order). Moreover, functions can be written as λ-abstractions and terms of the HOL logic can be polymorphic (with type variables ranged over by $\alpha, \beta, \gamma, \ldots$ represented in HOL by `*`, `**`, ... as mentioned above).

Terms of the HOL logic (object-language terms) are represented in ML by an abstract type called `term` and they are distinguished from ML expressions by enclosing them in double quotes. Terms can be manipulated through various built-in ML

functions. For example, given the expression `x \/ y`, which in ML evaluates to a term representing the disjunction x∨y, the function `dest_disj: term -> (term # term)` splits the disjunction into the pair of its two disjuncts:

```
#"x \/ y";;
"x \/ y" : term

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

The types of terms of the HOL logic are similar to those of ML expressions. The ML type called `type` represents the types of HOL terms, which are expressions of the form `": ··· "`. The built-in function `type_of` returns the logical type of a term.

```
#(3,4);;
(3, 4) : (int # int)

#"(3,4)";;
"3,4" : term

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

There are four kinds of terms: variables, constants, function applications and λ-abstractions. A λ-term $\backslash x.t$ denotes a function $v \mapsto t[v/x]$, where $t[v/x]$ denotes the result of substituting v for x in the term t.

```
#"\x. x * x";;
"\x. x * x" : term

#type_of it;;
":num -> num" : type
```

The type checking algorithm tries to infer the type of HOL terms using the types of constants and operators that occur in the same quotation. Above, the type of the multiplication operator `":num -> (num -> num)"` is used to determine the type of the function denoted by the λ-term. If there is not enough type information, a type checking error results and the user has to provide some more type information explicitly.

The ML language is used to manipulate terms of the HOL logic. In particular, ML is used to prove that certain terms are *theorems*. Theorems of the logic are also represented by abstract data types, and the user can work on them only through ML functions. The ML type for theorems is `thm`. A theorem is represented by a

finite set of terms called *assumptions* and a term called *conclusion*. Given a set of assumptions Γ and a conclusion t, $\Gamma \vdash t$ (or simply $\vdash t$, if Γ is empty) denotes the corresponding theorem. In order to introduce theorems into the logic, they must either be postulated as axioms or derived from existing theorems and definitions by formal proof.

A *theory* is a collection of logical types, type operators, constants, definitions, axioms and theorems. Theories enable a hierarchical organisation of facts, i.e. if facts from other theories need to be used, the relevant theories must be declared as *parents*. There are several built-in theories in HOL. Examples are the theories `bool`, `prim_rec` and `sets`.

The theory `bool` contains the definitions of various constants and type operators, such as the usual logical constants of the predicate calculus `T`, `F`, `~`, `/\`, `\/`, `==>`, `=`, `!`, `?` and `?!` to represent true T, false F, negation $\neg$, conjunction $\wedge$, disjunction $\vee$, implication $\supset$, equality $=$, universal quantification $\forall$, existential quantification $\exists$ and unique existential quantification $\exists!$, respectively. Hilbert's choice operator ε (written `@` in HOL notation) is another primitive constant defined in the theory `bool`. If P has type $\alpha \to \mathit{bool}$, then the term $\varepsilon x.\, P\, x$ denotes some element of the set whose characteristic function is P. If the set is empty, then $\varepsilon x.\, P\, x$ denotes an arbitrary element of the set denoted by α. This implies that all logical types must denote non-empty sets, since for any type α, the term $\varepsilon x : \alpha.\, \mathsf{T}$ represents an element of the set denoted by α. The product type $\alpha \times \beta$ for ordered pairs is also defined in the theory `bool` together with its constants, such as $\mathsf{Fst} : \alpha \times \beta \to \alpha$ and $\mathsf{Snd} : \alpha \times \beta \to \beta$ for selecting the first and second components of pairs, and the theorems describing such operations.

Constants, definitions and theorems concerning primitive recursive functions are given in the theory `prim_rec`. The following is the basic theorem `num_Axiom`

$$\vdash \forall e\ f.\ \exists!\ \mathit{fn}.\ (\mathit{fn}\ 0 = e)\ \wedge\ (\forall n.\ \mathit{fn}\ (n+1) = f\ (\mathit{fn}\ n)\ n)$$

stating the validity of primitive recursive definitions on the natural numbers. This means that for any e and f there exists a unique total function *fn* which satisfies the primitive recursive definition whose form is determined by e and f.

The theory `sets` contains constants, definitions and theorems about finite and infinite sets. The basis is the polymorphic type $(\alpha)\mathit{set}$ which is just an object-language abbreviation for the type $\alpha \to \mathit{bool}$. In fact, a set is represented by its characteristic function and the elements of a set $s : (\alpha)\mathit{set}$ are just those values of type α for which the corresponding predicate is true. Generalised set specifications are also supported, that is for any expression $E[x]$ and predicate $P[x]$, $\{E[x] : P[x]\}$ (represented in HOL by `{E[x] | P[x]}`) is the set of all values $E[x]$ for which $P[x]$ holds. The theory `sets` provides a wide collection of theorems about the various operations on sets and their properties.

Rules of definition are included in the HOL logic for extending theories in a purely definitional way. This is done by defining new constants and types in terms of properties of existing ones, thus extending Church's formulation. The rules of definition are briefly presented in the following sections.

2.3 Primitive Rules of Definition

The primitive basis of the HOL logic includes three rules of definition for extending the logic in a sound way.

The rule of *constant definition* allows one to introduce a new constant $\mathtt{c}$ as an object-language abbreviation for a closed (no free variables) term t. This is achieved by defining an equational axiom $\vdash \mathtt{c} = t$. Among the various properties that the constant $\mathtt{c}$ must satisfy is the condition that $\mathtt{c}$ may not occur in the term t. Thus, recursive definitions cannot be introduced into the logic using the rule of constant definition. The functions `new_definition` and `new_infix_definition` are some of the ML functions implementing the rule of constant definition.

Given a theorem of the form $\vdash \exists x_1 \ldots x_n.P[x_1, \ldots, x_n]$, the rule of *constant specification* (implemented by the ML function `new_specification`) allows one to give a name to existing values $x_1, \ldots, x_n$ for which $P[x_1, \ldots, x_n]$ holds. This is obtained by introducing new constants $\mathtt{c_1}, \ldots, \mathtt{c_n}$ and deriving the theorem $\vdash P[\mathtt{c_1}, \ldots, \mathtt{c_n}]$.

The rule of *type definition* (provided by the ML function `new_type_definition`) allows one to define a new type constant or type operator. Given a type α, let $P : \alpha \to \mathit{bool}$ be the characteristic function of some (non-empty) subset of the set denoted by α. From the theorem $\vdash \exists x : \alpha.\, P\, x$, the rule of type definition derives the existence of a bijection from the elements of a new type β to the subset of elements of type α that satisfy P, as asserted by the theorem

$$\vdash \exists f : \beta \to \alpha.\, (\forall x\, y.\, (f x = f y) \supset (x = y)) \wedge (\forall x.\, P x = (\exists y.\, x = f y))$$

This theorem introduces the new type β to name the non-empty subset of elements of type α that satisfy the predicate P. Functions to denote the above bijection and its inverse can be defined. A *representation* function $\mathsf{REP}_\beta : \beta \to \alpha$ maps a value of the new type β into the value of type α which represents it. The *abstraction* function $\mathsf{ABS}_\beta : \alpha \to \beta$ maps a representation of type α to its abstract value of type β. ABS_β is the left inverse of REP_β and, for those elements of type α that satisfy P, REP_β is the left inverse of ABS_β:

$$\vdash \forall a.\, \mathsf{ABS}_\beta\, (\mathsf{REP}_\beta\, a) = a$$
$$\vdash \forall r.\, P\, r = (\mathsf{REP}_\beta\, (\mathsf{ABS}_\beta\, r) = r)$$

These mappings allow one to define operations on the values of the new type β in terms of operations on values of the representing type α.

2.4 Derived Rules of Definition

The primitive rules of definition are of very restricted forms and this means that all other kinds of definitions must be derived from the primitive ones by formal proof. This can sometimes lead to rather complex formalisations. However, several derived rules of definition have been mechanised in HOL and are supported in a fully automatic way. These rules include recursive concrete type definitions, primitive recursive function definitions over these types and certain forms of inductive definition, all developed by Melham [7, 8].

The derived rule of *recursive type definition* (`define_type` in ML) allows one to define arbitrary concrete recursive types in terms of their constructors [7]. The input to this definition mechanism is a specification of the syntax of the operators written in terms of existing types and recursive calls to the type being defined:

$$(\alpha_1, \ldots, \alpha_n)rty \ ::= \ \mathsf{C}_1 \, ty_1^1 \ldots ty_1^{k_1} \ \mid \ \ldots \ \mid \ \mathsf{C}_m \, ty_m^1 \ldots ty_m^{k_m}$$

where $\mathsf{C}_1, \ldots, \mathsf{C}_m$ ($m \geq 1$) are distinct constructors, each taking k_i arguments ($k_i \geq 0$), and each ty_i^j is either the recursive type rty or an existing logical type (not containing rty). If one or more of the ty_i^j is rty, then the type specification defines a recursive type. Non-recursive types defined through this type definition mechanism are just special cases. If the type being defined is recursive, at least one constructor must be non-recursive, i.e. the type of all its arguments may not be rty. The type $(\alpha_1, \ldots, \alpha_n)rty$ is polymorphic in the type variables $\alpha_1, \ldots, \alpha_n$ if $n \geq 1$; if $n = 0$ then rty is a type constant.

The above type specification denotes the set of all expressions which can be finitely generated using the constructors $\mathsf{C}_1, \ldots, \mathsf{C}_m$, which are distinct and one-to-one. Given such a type specification, the rule of recursive type definition performs all the formal inference necessary to define the type in higher order logic and derives an abstract characterisation of the type $(\alpha_1, \ldots, \alpha_n)rty$ in a fully automatic way. This characterisation is a theorem of higher order logic asserting that there exists a unique function which satisfies the primitive recursion defined by the type specification for $(\alpha_1, \ldots, \alpha_n)rty$. The recursive type definition package also provides functions which automatically prove that the type constructors are distinct and one-to-one and derive theorems for structural induction and case analysis.

The derived mechanism of *primitive recursive function definition* (provided by the ML function `new_recursive_definition`) automates existence proofs for primitive recursive functions defined over concrete recursive types. The system proves the existence of a total function satisfying the recursive defining equations, and then a constant specification introduces a new constant to denote such a total function.

The derived rule for *inductive definitions* (`new_inductive_definition` in ML) allows one to define relations which are inductively defined by a set of rules [8]. Let

R be an n-place relation defined through a set of rules of the form

$$\frac{R(t_1^1,\ldots,t_n^1) \quad \ldots \quad R(t_1^k,\ldots,t_n^k)}{R(t_1,\ldots,t_n)} \qquad c_1 \ldots c_r$$

where the terms $R(t_1^i,\ldots,t_n^i)$ for $1 \leq i \leq k$ are the premisses of the rule, $c_1 \ldots c_r$ are the side conditions (not involving the relation R being defined) and $R(t_1,\ldots,t_n)$ is the conclusion of the rule. The relation R is closed under the above rule if the conclusion is true whenever the premisses and the side conditions are true. The relation R is inductively defined by a set of such rules if R is the least relation closed under all the rules.

In the inductive definition package, any such relation is simply defined as the intersection of all relations closed under a given set of rules. The system automatically proves that the resulting relation is itself closed under the set of rules and is the least such relation. The theorems resulting from this definition mechanism constitute a complete characterisation of the properties of the newly-defined relation. They include a list of theorems (one for each rule) which assert that the relation satisfies those rules, and a theorem which states a principle of *rule induction* for the relation. Given the above relation R, rule induction allows one to prove that every element in R has a property P, i.e. $R(x_1,\ldots,x_n)$ implies $P(x_1,\ldots,x_n)$, by simply proving that the relation $\{(x_1,\ldots,x_n) \mid P(x_1,\ldots,x_n)\}$ is closed under the rules that define R. This is due to the fact that R is the least relation closed under the same rules. The theorem of rule induction allows proofs by induction to be performed over the structure of the derivations defined by the set of rules. Furthermore, the inductive definition package provides a theorem for performing exhaustive case analysis over the inductively defined relation.

2.5 Proofs in HOL

Theories are extended in a sound way by deriving new theorems by formal proof. To prove a theorem in a theory, one must apply a sequence of proof steps to either axioms or previously proved theorems using ML programs called *inference rules* (forward proof). The core of the deductive system in HOL is made up of eight primitive inference rules, from which all other rules are derived. Among the primitive inference rules is β-conversion, implemented by the ML function `BETA_CONV`, which reduces a β-redex $(\lambda x.\, u)\, v$ to the term $u[v/x]$ by returning the theorem $\vdash (\lambda x.\, u)\, v = u[v/x]$ (by renaming variables where necessary to avoid free variable capture).

```
#BETA_CONV "(\x y. x + y)y";;
|- (\x y. x + y)y = (\y'. y + y')
```

In fact, *conversions* are ML functions that map a term t to a theorem $\vdash t = u$ expressing the equality of t with some other term u. Inference rules typically transform

theorems into other theorems, e.g. the rule BETA_RULE which, given a theorem $\Gamma \vdash t$, returns the theorem resulting from reducing all the β-redexes, at any depth, in the conclusion t.

```
#let thm = ASSUME "f = ((\x y. x + y) y)";;
thm = f = (\x y. x + y)y |- f = (\x y. x + y)y

#BETA_RULE thm;;
f = (\x y. x + y)y |- f = (\y'. y + y')
```

BETA_RULE is defined in terms of BETA_CONV using the ML functions DEPTH_CONV, that applies a conversion repeatedly to all subterms in a bottom-up order, and CONV_RULE, which transforms a conversion into an inference rule.

Among the derived inference rules is the basic rewriting rule REWRITE_RULE which, given a list of equational theorems (namely theorems whose conclusion is of the form $t = u$) and a theorem *thm*, replaces any subterm in *thm* that matches the left-hand side of any of the theorems in the list by the corresponding instance of the right-hand side.

```
#MULT_0;;
|- !m. m * 0 = 0

#MULT_ASSOC;;
|- !m n p. m * (n * p) = (m * n) * p

#REWRITE_RULE [MULT_0] (SPECL ["1"; "0"; "0"] MULT_ASSOC);;
|- T
```

Note how the theorem MULT_ASSOC for associativity of multiplication has been 'specialised' using the inference rule SPECL. Given a list of terms and a theorem, SPECL instantiates (some of) the universally quantified variables in the conclusion of the theorem (by renaming variables where necessary to avoid free variable capture). When specialising only one variable, the rule SPEC can also be used.

Some built-in basic tautologies are also implicitly used by REWRITE_RULE. The search for subterms to be replaced is performed top-down and recursively, until no more replacements can be done. This may lead to unwanted and/or unnecessary reductions or even non-termination of the rewriting process. Other versions of the rewriting rule, such as ONCE_REWRITE_RULE that rewrites subterms once, and substitution rules, such as SUBST that replaces selected subterms, may be used instead.

```
#ONCE_REWRITE_RULE [MULT_0] (SPECL ["1"; "0"; "0"] MULT_ASSOC);;
|- 1 * 0 = 0
```

Besides forward proofs, the HOL system supports another way of carrying out a proof, called goal directed proof or backward proof. The idea is to do the proof starting from the desired result (*goal*) and manipulating it until it is reduced to a subgoal which is obviously true. ML functions that reduce goals to subgoals are called *tactics* and were developed by Milner. The HOL system provides a *subgoal package* due to Paulson [9], which implements a simple framework for interactive proofs. A goal given by an assumption list Γ and a term t, written Γ `?-` t (if Γ is empty, the goal is written `?-` t), can be set by invoking either the function `set_goal` or the function `g` (an abbreviation of `set_goal` whenever Γ is empty), which initialises the subgoal package with a new goal.

As an example, let us prove the above theorem `MULT_0`:

```
#set_goal([],"!m. (m * 0 = 0)");;
"!m. m * 0 = 0"
```

The current goal can be expanded using the function `expand` (written `e` for short) which applies a tactic to the top goal on the stack and pushes the resulting subgoals onto the goal stack. By applying mathematical induction with the built-in tactic `INDUCT_TAC`, two subgoals corresponding to the basis case (the subgoal at the bottom of the subgoal list) and to the induction step are generated:

```
#e INDUCT_TAC;;
OK..
2 subgoals
"(SUC m) * 0 = 0"
    [ "m * 0 = 0" ]

"0 * 0 = 0"
```

Tactics corresponding to conversions and inference rules are defined in HOL. For example, rewriting tactics such as `REWRITE_TAC` and `ASM_REWRITE_TAC`, which adds the assumptions of the goal to the given list of theorems, are fundamental in goal directed proofs. The basis case is solved by rewriting with the definition of multiplication:

```
#MULT;;
|- (!n. 0 * n = 0) /\ (!m n. (SUC m) * n = (m * n) + n)

#e (REWRITE_TAC [MULT]);;
OK..
goal proved
|- 0 * 0 = 0

Previous subproof:
"(SUC m) * 0 = 0"
    [ "m * 0 = 0" ]
```

When a tactic solves a subgoal, the package computes a part of the proof and presents the user with the next subgoal. The definition of multiplication is used once more to transform the induction subgoal:

```
#e (REWRITE_TAC [MULT]);;
OK..
"(m * 0) + 0 = 0"
  1  ["m * 0 = 0" ]
```

The zero summand can be deleted by applying properties of addition (given in HOL by the theorem ADD_CLAUSES):

```
#e (REWRITE_TAC [ADD_CLAUSES]);;
OK..
"m * 0 = 0"
  1  ["m * 0 = 0" ]
```

The induction subgoal is thus reduced to the inductive hypothesis and simply solved by rewriting with it:

```
#e (ASM_REWRITE_TAC[]);;
OK..
goal proved
. |- m * 0 = 0
. |- (m * 0) + 0 = 0
. |- (SUC m) * 0 = 0
|- !m. m * 0 = 0

Previous subproof:
goal proved
() : void
```

Conversions can be mapped into tactics using the ML function `CONV_TAC`, such as the tactic `BETA_TAC` (defined in terms of `BETA_CONV`) for applying β-conversion to the conclusion of a goal.

```
#g "(\x. x + 1)1 = SUC 1";;
"(\x. x + 1)1 = SUC 1"

#e BETA_TAC;;
OK..
"1 + 1 = SUC 1"
```

Tactics can be composed using other ML functions called *tacticals*. For example, tactics can be sequenced by means of the (infixed) tactical `THEN`: given tactics T_1 and T_2, the ML expression T_1 `THEN` T_2 evaluates to a tactic that first applies T_1 and then applies T_2 to each subgoal produced by T_1. Another sequencing tactical is `THENL`: given a tactic T that generates n subgoals and a tactic list $[T_1; \ldots; T_n]$, then the tactic T `THENL` $[T_1, \ldots, T_n]$ first applies T and then applies T_i to the ith subgoal produced by T.

When a theorem is proved, it can be stored in the current theory using several standard functions. Among the others, `TAC_PROOF` takes a goal and a tactic, and applies the tactic to the goal in an attempt to prove it; or one can use the function `prove_thm` which takes a string s, a boolean term t and a tactic *tac*, and attempts to prove the goal `?-` t by applying *tac*. If it succeeds, the resulting theorem is saved under the name s in the current theory. For example, the above goal about multiplication is proved by the following tactic and stored under the name `MULT_0`:

```
#let MULT_0 =
#  prove_thm
#   (`MULT_0`,
#    "!m. (m * 0 = 0)",
#    INDUCT_TAC THEN ASM_REWRITE_TAC [MULT;ADD_CLAUSES]);;
MULT_0 = |- !m. m * 0 = 0
```

A mixed approach of backward and forward proofs is also possible and can sometimes be convenient. When developing a tactic in a goal directed proof, it may be useful to derive new theorems from the assumptions and/or other theorems in a forward manner, and then use these new facts to manipulate the goal.

Bibliography

[1] Church, A., 'A Formulation of the Simple Theory of Types', *Journal of Symbolic Logic* **5**, 1940, pp. 56–68.

[2] Cousineau, G., G. Huet, and L. Paulson, 'The ML Handbook', INRIA, 1986.

[3] Gordon, M., 'Why Higher-order Logic is a Good Formalism for Specifying and Verifying Hardware', in *Formal Aspects of VLSI Design*, G. Milne and P. A. Subrahmanyam (eds.), North-Holland, 1986.

[4] Gordon, M. J. C., 'HOL—A Proof Generating System for Higher-Order Logic', in *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. Subrahmanyam (eds.), Kluwer Academic Publishers, Boston, 1988, pp. 73–128.

[5] Gordon, M. J. C., 'Mechanizing Programming Logics in Higher Order Logic', in *Current Trends in Hardware Verification and Automated Theorem Proving*, G. Birtwistle and P. Subrahmanyam (eds.), Springer-Verlag, 1989, pp. 387–439.

[6] Gordon, M. J. C. and T. F. Melham, '*Introduction to* HOL*: a theorem proving environment for higher order logic*', Cambridge University Press, 1993.

[7] Melham, T. F., 'Automating Recursive Type Definitions in Higher Order Logic', in *Current Trends in Hardware Verification and Automated Theorem Proving*, G. Birtwistle and P. Subrahmanyam (eds.), Springer-Verlag, 1989, pp. 341–386.

[8] Melham, T. F., 'A Package for Inductive Relation Definitions in HOL', in Proceedings of the *1991 International Workshop on the HOL Theorem Proving System and its Applications*, Archer, M., J. J. Joyce, K. N. Levitt, and P. J. Windley (eds.), IEEE Computer Society Press, 1992, pp. 350–357.

[9] Paulson, L. C., *Logic and Computation—Interactive Proof with Cambridge* LCF, Cambridge Tracts in Theoretical Computer Science **2**, Cambridge University Press, 1987.