HOL Formalised:
Language and Overview

R.D. Arthan
Lemma 1 Ltd.
rda@lemma-one.com

25 October 1993
Revised 27 April 2014

Abstract

This document is the first in a suite of documents which give a formal specification of HOL. It acts as an overview to the formal treatment and includes the detailed treatment of the HOL language.

The overview of the specification discusses the theoretical background and some of the decisions which have been taken in approaching the specification task. It also describes briefly the ProofPower-HOL specification facilities which are used.

The description of the HOL language defines the syntax of types, terms sequents and theories. Some supporting functions, such as a function to carry out type instantiation, are also defined.

An index to the formal material is provided at the end of the document.

Reference: DS/FMU/IED/SPC001; issue *Revision : 2.12*

1 DOCUMENT CONTROL

1.1 Contents list

1.2 Document cross references

[1] Elliot Mendelson. *Introduction to Mathematical Logic.* Wadworth and Brook/Cole, third edition, 1987.

[2] DS/FMU/IED/SPC001. *HOL Formalised: Language and Overview.* R.D. Arthan, Lemma 1 Ltd., `http://www.lemma-one.com`.

[3] DS/FMU/IED/SPC002. *HOL Formalised: Semantics.* R.D. Arthan, Lemma 1 Ltd., `http://www.lemma-one.com`.

[4] DS/FMU/IED/SPC003. *HOL Formalised: Deductive System.* R.D. Arthan, Lemma 1 Ltd., `http://www.lemma-one.com`.

[5] DS/FMU/IED/SPC004. *HOL Formalised: Proof Development System.* R.D. Arthan, Lemma 1 Ltd., `http://www.lemma-one.com`.

[6] *The HOL System: Description.* SRI International, 4 December 1989.

2 GENERAL

2.1 Scope

This document is part of a formal specification of the HOL logic. The formal specification is a formal treatment of the description of the HOL logic and proof development system given in chapters 9 and 10 of [6].

This document contains a brief overview of the specification and also defines the syntax of the HOL language as used throughout the specification.

3 OVERVIEW OF THE SPECIFICATION

3.1 Theoretical Background

It may be helpful to discuss some generalities about the definition of logics, in order to set in context the specific constructions we will make to specify the HOL deductive system. Readers who know what to expect are invited to skip this section. If we apply Occam's Razor fairly viciously to the sort of definition found in, e.g., Mendelson's textbook on logic [1], one finds that a deductive system[1] is given by a set, S, whose elements we will call sentences in this section, and a subset I of $\bigcup_{n=1}^{\infty} S^n$. One says that $x \in S$ is *directly derivable* from $X \subseteq S$ if for some n, $(X^{n-1} \times \{x\}) \cap I \neq \varnothing$. One then says that $x \in S$ is *derivable* from $X \subseteq S$, if for some sequence $x_1, x_2, \ldots, x_k$ of elements of S, $x_k = x$ and, for each i, x_i is either in X or is directly derivable from $\{x_1, x_2, \ldots, x_{i-1}\}$. One says that $x \in S$ is a *theorem* if it is derivable from $\varnothing$.

In practice, S is usually defined by a decidable "well-formedness" predicate on the free algebra, $F(\Omega)$, over some signature, Ω, and I is given as the union of a set of decidable n-ary relations (the rules of inference).

The above ideas, while of theoretical value, are not sufficient for a practical proof development system like HOL, since, in such a system, the user can introduce new constructs into the language S by modifying the signature Ω, and can assert that certain sentences in the extended language, $S(\Omega)$, are axioms. For example, when a new constant is defined in HOL, the language is extended to include the new constant and an axiom that the constant is equal to the value given in its definition is asserted.

Let us assume that the well-formedness predicates and inference rules are defined so as to apply to sentences over any signature the user can define. This may be achieved by restricting the signatures to be subsignatures of a signature Σ_{max}. A predicate over $F(\Sigma_{max})$ then restricts to a predicate over $F(\Omega)$ for any subsignature, Ω, of Σ_{max}, and, similarly, any set of rules of inference over $L(\Sigma_{max})$ restricts to a set of rules of inference over $S(\Sigma_{max})$. Let us assume that a well-formedness predicate and a set of rules of inference have been defined for some signature Σ_{max}.

Let us define a *theory* to be a pair (Ω, X), where Ω is a subsignature of Σ_{max} as above, and $X \subseteq S(\Omega)$. X is the set of axioms of the theory. A *theorem* in a theory, (Ω, X), is a sentence in $S(\Omega)$ which is derivable from X (with respect to the rules of inference restricted to $S(\Omega)$). Thus the axioms, X, act as additional unary rules of inference. Theories form a partially ordered set with respect to inclusion. We will actually use *extension*: the relation inverse to inclusion.

(The signature part of a theory can in many cases of interest be omitted. For example, treatments of first-order logic commonly offer an infinite supply of constant letters and predicate letters for use in

[1] Mendelson calls it a *formal theory.* The term *formal system* and others are also used.

constructing sentences. This corresponds to insisting that each signature Ω is equal to Σ_{max} in the above formulation. The more general treatment discussed here seems more appropriate to HOL.)

The rules of inference over $S(\Sigma_{max})$ induce rules of inference on the sentences in the language, $L(T)$, of a given theory T. The theorems of the theory (Ω, X) are then precisely the sentences in $S(\Omega)$ which are derivable from X using the induced inference rules.

A theory is *consistent* if not every sentence in its language is a theorem. Of particular interest in a practical proof development system are mechanisms for extending a theory which preserve consistency. A theory T_1 is a *conservative extension* of a theory T if T_1 extends T and all sentences in $L(T)$ which are theorems in T_1 are also theorems in T. Clearly conservative extensions preserve consistency.

A *semantics* for a theory (Ω, X) gives meaning to the sentences of the language $S(\Omega)$ by assigning values to them. This is most readily done by selecting on the basis of intuitive or theoretical considerations, some Ω-algebra, V and considering the restriction to $S(\Omega)$ of the mapping from $F(\Omega)$ given by the universal property of a free algebra. Such a mapping is called an *interpretation* of the theory. If V has sufficient structure for us to view the sentences of $S(\Omega)$ as propositions, we may use an interpretation to reason about the rules of inference and the axioms X. In particular, given a *a model* — an interpretation which maps the axioms to true propositions — we may ask whether the inference rules are *valid*, i.e. whether they preserve truth.

3.2 Structure of the Specification

The previous section identifies three main topics we have to consider for HOL: its language, its deductive system and its semantics. We also wish to specify, at an abstract level, the critical properties of a program purporting to support the development of proofs in the logic. We devote a document to each of these four topics [2, 3, 4, 5]. Each document in the specification contributes an HOL theory. The theories are briefly described in the following table:

Name	Parents	Description
spc001	fin_set[2]	This contains our definition of the HOL language. The main definitions are types *TYPE*, *TERM* and *THEORY* representing the types, terms and theories described in [6].
spc002	spc001	This specifies the semantics of the HOL language. The main definitions are of a predicate *is_set_theory* which specifies the sorts of universe in which the semantics can be given and a predicate *is_model* which specifies what it means to be a model of a *THEORY* in some such universe.
spc003	spc001	This specifies the HOL deductive system. That is to say it defines the notion of derivability (with respect to a formalisation of the primitive inference rules of HOL as described in [6]).
spc004	spc002 spc003	This gives an abstract model of an HOL proof development system and gives semantic and syntactic formulations of the critical properties of such a system.

[2] This is a library theory making available operations on sets, lists, strings etc.

3.3 Approach

Initially we had hoped to present something which could specify both the deductive system (i.e. the formal theory in the sense of a mathematical structure with sentences, inference rules etc.) and the system (i.e. the program which enable one to calculate theorems). However, in defining the deductive system we frequently found that attempts to make the specification "constructive" tended to obscure some issues. We have consequently erred on the side of abstraction in most cases. For example, many of the functions we need are partial functions: we represent these as binary relations, rather than approximate them with total functions. This approach was felt to lead to a clearer specification than would be obtained by using approximating total functions together with checks on the arguments supplied in each application.

In formalising the system we have, on occasion, felt that certain changes would be desirable for one reason or another. We have resisted all such urges — what is presented here is meant to be a rigorous formulation of the logic as described in [6]. Where [6] has proved a little too loose for our purposes (e.g. in the details of type instantiation), we have tried to follow the spirit of the HOL system.

There are occasionally differences in terminology between our usage and [6]. We have attempted to indicate these as they arise. This is most evident in the semantics since our treatment is in HOL rather than ZF set theory as used in [6].

3.4 Notation

The documents which make up the specification are *literate scripts* containing a mixture of narrative text and input for the ProofPower system. The theory listings in the documents are obtained by loading the HOL input from the documents into the HOL system and the listing the theories produced.

The specification makes much use of the Z-like specification features ProofPower provides. These features are briefly explained here.

Constants are intoduced using constant definition boxes which have the form:

```
SML
│(*

HOL Constant
│        c1 : ty1,
│        c2 : ty2,
│        ...
├────────────────────────────────────
│        P

SML
│*)
```

The intention of this is to introduce new constants, *c1*, *c2*, ..., satisfying the property P, using *new_specification*, and, if the consistency proposition required by *new_specification* can be proved automatically by one of a range of heuristics, then the effect is exactly that. If the consistency proposition cannot be proved automatically the constants are still introduced but with a defining property which is consistent and which is equivalent to P if P is consistent. A metalanguage function “*specification*”, analagous to “*definition*” may be used to extract the defining property from the theory database.

Some use is made of an experimental metalanguage function *type_spec* which is an analogue of the constant definition box for defining types and deferring any proof obligations. It is supported by object language constants $\simeq$ and *Of*. A definition such as:

```
type_spec {rep_fun="rep_three", def_tm =
        ⌜THREE ≃ mk_three Of (λi:ℕ• i < 3)⌝
};
```

introduces a new type *THREE*, with representation function *rep_three* and abstraction function *mk_three*, which is in one-to-one correspondence with the set of natural numbers less than *3*.

In cases where the proof obligations for introducing a new type are proved, an experimental metalanguage function called *abs_rep_spec* is used as a convenient way of introducing abstraction and representation functions for the new type. This takes as parameters a metalanguage labelled record including components which name the type and the two functions to be introduced.

Other significant differences from Cambridge HOL are: object language terms are quoted using Strachey brackets, “⌜” and “⌝”; type variables are distinguished using a prime rather than an asterisk, e.g. *′a* rather than $*$; and type abbreviations with arguments are supported (any type variables in the definition of the the abbreviation become arguments).

Largely for historical reasons, the object language described in this formal treatment uses the Cambridge HOL conventions in those places where it is necessary to give concrete syntax.

4 PREAMBLE

We introduce the new theory. Its parent is a library theory containing various definitions we need.

SML

```
open_theory"fin_set";
new_theory"spc001";
push_pc"hol";
```

5 THE SYNTAX OF TYPES AND TERMS

We now embark on defining the language of HOL. The treatment will follow the lines discussed in section 3.1 above. However, since we are only interested in a particular language we do not do any general universal algebra. Thus, apart from a minor complication dealt with in section 6 below, defining our version of $F(\Sigma_{max})$ and $L(\Sigma_{max})$ amounts to specifying the language of HOL types and terms.

The language is defined informally in [6] by a grammar essentially the same as the following (in which the terminal symbols, *tyvar*, *tyop* etc., stand for names of various sorts of objects).

BNF

```
        type    =       tyvar                                   (* Type Variable *)
                |       `(`, type, {`,`, type}, `)`, tyop;              (* Compound Type *)

        term    =       var, `:`, type                          (* Variable *)
                |       con, `:`, type                          (* Constant *)
                |       term, term                              (* Application *)
                |       `λ`, var, `:`, type, `•`, term;                 (* λ−abstraction *)
```

Here the atomic types and function types of [6] are subsumed by the compound types (an atomic type being a compound type with no parameters and a function type being one with exactly two parameters and with the type operator '→').

The type and term languages are subject to well-formedness rules of two sorts: context-sensitive rules governing conformance of the type of a constant or the arity of a type with a definition of the constant or type contained in a theory; and the "local" rule that the operator of a combination be of an appropriate type to apply to its operand. To avoid a mutual recursion between the types *TYPE*, *TERM* and *THEORY* which we are going to define, we will not impose the context-sensitive rules as part of the definitions of *TYPE* and *TERM*. Instead, when we define the type *THEORY*, we insist that any types or terms which appear in a theory satisfy appropriate conditions. In the following subsections we therefore only consider the local well-typing rule.

If machinery were available to define the recursive types we need automatically, we would probably use it (to define the free algebra of types and a free algebra which would have the type of terms as a subset). Unfortunately, the type *TYPE* involves a recursion through the *list* type constructor and this is not currently supported by T. Melham's system for defining recursive types (and no suitable analogue is currently available for ProofPower). Consequently we work here with an explicit concrete representation of types and terms using strings.

5.1 Names

We could, in priniciple, take the names which appear in types and terms from some arbitrary type. However the extra generality would add complexity and does not seem to offer any benefit over the natural representation of names as strings.

It is, however, technically convenient to allow arbitrary strings to be used as names (since this lets us formulate and use the constructor functions for types and terms in a natural way). To enable this we use an encoding of names in the concrete representation which allows an arbitrary string to be viewed as a name. To do this we use an escape character to protect any occurrences of the characters which act as delimiters in the concrete representation of types or terms.

We use ‘$’ as the escape character (in fact any character other than ‘(’, ‘)’, ‘,’, ‘:’, ‘λ’ or ‘•’ would do). The encoding is then given by the following function:

HOL Constant

```
│        encode : STRING → STRING
├──────────────────────────────────────────
│∀ch : CHAR; s : STRING •
│               encode "" = ""
│        ∧      encode (Cons ch s) =
│               if     ch ∈ {‘$‘; ‘(‘; ‘)‘; ‘,‘; ‘:‘; ‘λ‘; ‘•‘}
│               then   Cons ‘$‘ (Cons ch (encode s))
│               else   Cons ch (encode s)
```

The range of the function *encodeencode* will comprise the strings produced by the following grammar:

BNF

```
│        name   =      ""
│               |      (char − ("$" | "(" | ")" | "," | ":" | "λ" | "•")), name
│               |      "$", char, name;
```

5.2 Types

Our concrete representations for types are the strings which satisfy a predicate, *is_type*, defined below. This is satisfied only by the strings produced by the following grammar:

BNF

```
│        type   =      name
│               |      "(‘, [type, {‘,‘, type}], ‘)‘, name;
```

The following utility function is used to construct the argument lists of compound types.

HOL Constant

```
│        comma_list : STRING LIST → STRING
├──────────────────────────────────────────
│        comma_list [] = ""
│∧       ∀x t •        comma_list (Cons x t)
│               =      if     t = []
│                      then   x
│                      else   x @ (Cons ‘,‘ (comma_list t))
```

The operations on strings which will represent the constructor functions of the type *TYPE* are the following:

HOL Constant

```
│       mk_var_type_rep: STRING → STRING
│
├──────────────────────────────────────
│       ∀s•mk_var_type_rep s = encode s
```

HOL Constant

```
│       mk_type_rep: STRING × STRING LIST  → STRING
├──────────────────────────────────────
│       ∀s tl•mk_type_rep(s, tl) = "(" @ comma_list tl @ ")" @ encode s
```

We may now define *is_type* as the smallest set which is closed under the constructors *type_rep* and *mk_type_rep*.

HOL Constant

```
│       is_type : STRING SET
├──────────────────────────────────────
│       is_type = ⋂{X:STRING SET |
│       (∀s• mk_var_type_rep s ∈ X)
│∧      (∀pars tycon•
│               Elems pars ⊆ X
│       ⇒       mk_type_rep(tycon, pars) ∈ X)}
```

We prove that *is_type* is non-empty and use the result to define a new type, *TYPE*.

SML

```
│val thm1 = save_thm("thm1", (
│       set_goal([], ⌜∃ty•ty ∈ is_type⌝);
│       a(∃_tac⌜encode s⌝);
│       a(rewrite_tac (map get_spec[⌜is_type⌝, ⌜⋂⌝, ⌜mk_var_type_rep⌝]));
│       a(REPEAT strip_tac THEN asm_rewrite_tac[]);
│       pop_thm()
│));
```

The definition of the new type follows the usual pattern:

SML

```
│val type_def = new_type_defn (["TYPE"], "TYPE", [],
│       (tac_proof( ([], ⌜∃ty•(λty•ty ∈ is_type) ty⌝),
│               rewrite_tac[thm1])));
│val abs_type_rep_type_def = abs_rep_spec
│       {type_def_name = "TYPE",
│        abs_fun = "abs_type",
│        rep_fun = "rep_type",
│        def_conv = Value (rewrite_conv[])};
```

The constructor functions for the new type are:

HOL Constant

mk_var_type: $STRING \to TYPE$

$\forall s \bullet mk\_var\_type\ s = abs\_type\ (mk\_var\_type\_rep\ s)$

. . . and:

HOL Constant

mk_type: $STRING \times TYPE\ LIST\ \to TYPE$

$\forall s\ tl \bullet mk\_type(s,\ tl) = abs\_type(mk\_type\_rep(s,\ Map\ rep\_type\ tl))$

We will also need a destructor function for types:

HOL Constant

dest_type: $TYPE \to STRING \times (TYPE\ LIST)$

$\forall s\ tyl \bullet dest\_type(mk\_type(s,\ tyl)) = (s,\ tyl)$

. . . and the constant type "*:bool*":

HOL Constant

Bool : $TYPE$

$Bool = mk\_type(\texttt{"}BOOL\texttt{"},\ [])$

. . . and a function to extract the set of type variables of a type:

HOL Constant

type_ty_vars: $TYPE \to STRING\ SET$

$(\forall x\ s \bullet\ x \in type\_ty\_vars\ (mk\_var\_type\ s) \Leftrightarrow x = s)$

$\land \quad (\forall x\ s\ tyl \bullet$

$\qquad x \in type\_ty\_vars\ (mk\_type\ (s,\ tyl))$

$\Leftrightarrow \quad \exists ty \bullet\ ty \in Elems\ tyl \land x \in type\_ty\_vars\ ty)$

5.3 Terms

The representation type for the well-formed terms will be $string \times TYPE$. The *string* component gives the concrete representation of the term according to the following grammar:

BNF

$term \quad = \quad `V`,\ name,\ `{:}`,\ type$

$\qquad\qquad | \quad `C`,\ name,\ `{:}`,\ type$

$\qquad\qquad | \quad `(`,\ term,\ `)(`,\ term,\ `)`$

$\qquad\qquad | \quad `\lambda V`,\ name,\ `{:}`,\ type,\ `\bullet`,\ term;$

The *TYPE* component gives the type of the term. This representation is analogous to the terms subscripted with their types one finds in [6]. (Note that the types which appear in the *string* components are not redundant. Without them it would not in general be possible to recover the types of the constituents of a combination, so that the constructor, *mk_comb*, for combinations would not be injective.)

The constructor functions for the type of terms will be represented by the following operations on strings.

HOL Constant

$$\textbf{mk\_var\_rep} : STRING \times TYPE \to STRING$$

$$\forall s\ ty \bullet mk\_var\_rep\ (s,\ ty) = \texttt{"}V\texttt{"} \mathbin{@} encode\ s \mathbin{@} \texttt{":"} \mathbin{@} rep\_type\ ty$$

HOL Constant

$$\textbf{mk\_const\_rep} : STRING \times TYPE \to STRING$$

$$\forall s\ ty \bullet mk\_const\_rep\ (s,\ ty) = \texttt{"}C\texttt{"} \mathbin{@} encode\ s \mathbin{@} \texttt{":"} \mathbin{@} rep\_type\ ty$$

HOL Constant

$$\textbf{mk\_comb\_rep} : STRING \times STRING \to STRING$$

$$\forall tm1\ tm2 \bullet mk\_comb\_rep\ (tm1,\ tm2) = \texttt{"("} \mathbin{@} tm1 \mathbin{@} \texttt{")("} \mathbin{@} tm2 \mathbin{@} \texttt{")"}$$

HOL Constant

$$\textbf{mk\_abs\_rep} : STRING \times TYPE \times STRING \to STRING$$

$$\forall s\ ty\ tm \bullet mk\_abs\_rep\ (s,\ ty,\ tm) = \texttt{"}\lambda V\texttt{"} \mathbin{@} s \mathbin{@} \texttt{":"} \mathbin{@} rep\_type\ ty \mathbin{@} \texttt{"}\bullet\texttt{"} \mathbin{@} tm$$

The following utility for forming function types is useful:

HOL Constant

$$\textbf{Fun} : TYPE \to TYPE \to TYPE$$

$$\forall\ ty1\ ty2 \bullet Fun\ ty1\ ty2 = mk\_type(\texttt{"}\to\texttt{"},\ [ty1;\ ty2])$$

The following predicate picks out the well-formed terms, by imposing the appropriate typing rules.

HOL Constant

$$\textbf{is\_wf\_term} : (STRING \times TYPE)\ SET$$

$$\begin{array}{ll}
& is\_wf\_term = \bigcap\{X{:}(STRING \times TYPE)\ SET \mid \\
& (\forall s\ ty \bullet (mk\_var\_rep(s,\ ty),\ ty) \in X) \\
\wedge & (\forall s\ ty \bullet (mk\_const\_rep(s,\ ty),\ ty) \in X) \\
\wedge & (\forall f\ a\ tya\ ty \bullet ((f,\ Fun\ tya\ ty) \in X \wedge (a,\ tya) \in X) \Rightarrow (mk\_comb\_rep(f,\ a),\ ty) \in X) \\
\wedge & (\forall s\ b\ tys\ tyb \bullet\ (b,\ tyb) \in X \Rightarrow (mk\_abs\_rep(s,\ tys,\ b),\ Fun\ tys\ tyb) \in X)\}
\end{array}$$

We prove that well-formed terms exist according to the above condition using a variable as a witness:

SML

```
val thm2 = save_thm("thm2", (
        set_goal([], ⌜∃tm•tm ∈ is_wf_term⌝);
        a(∃_tac ⌜(mk_var_rep(s, ty), ty)⌝);
        a(rewrite_tac (map get_spec[⌜is_wf_term⌝, ⌜⋂⌝]));
        a(REPEAT strip_tac THEN asm_rewrite_tac[]);
        pop_thm()
));
```

The definition of the new type follows the usual pattern:

SML

```
val term_def = new_type_defn (["TERM"], "TERM", [],
        (tac_proof( ([], ⌜∃tm•(λtm•tm ∈ is_wf_term) tm⌝),
                rewrite_tac[thm2])));
val abs_term_rep_term_def = abs_rep_spec
        {type_def_name = "TERM",
         abs_fun = "abs_term",
         rep_fun = "rep_term",
         def_conv = Value (rewrite_conv[])};
```

We can now define a function which assigns to any term its type:

HOL Constant

```
        type_of_term : TERM → TYPE
├──────────────────────────────────────────
        ∀ tm •  type_of_term tm = Snd(rep_term tm)
```

The constructor functions for the type *TERM*, namely *mk_var*, *mk_const*, *mk_comb* and *mk_abs*, could be defined as composites of *mk_cand_var* etc. and the abstraction and representation functions for *TERM*. Unfortunately the resulting functions *mk_comb* and *mk_abs* are not total functions[3]. Attempts to use an approximating total function turn out to lead to difficulties when we wish to define functions on terms by cases. Thus we must use relations to represent these constructors. Implementations exploit the fact that the relations corresponds to a partial function.

In our informal discussions below we will often use the name *mk_comb* and *mk_abs* to refer to these relations viewed as partial functions (i.e. with applicative notation).

The names chosen for the relations are intended to be suggestive of phrases like: *('x:num', "1") has mk_abs "λx:num•1".*

HOL Constant

```
        mk_var : (STRING × TYPE) → TERM
├──────────────────────────────────────────
        ∀ s ty • mk_var (s, ty) = abs_term (mk_var_rep(s, ty), ty)
```

[3] *mk_abs* could be reparameterised to be total quite simply, but we prefer to follow the treatment of [6]. *mk_comb*, however, is of necessity partial.

HOL Constant

$$\mathbf{mk\_const} : (\mathit{STRING} \times \mathit{TYPE}) \to \mathit{TERM}$$

$$\forall\ s\ ty \bullet\ mk\_const\ (s,\ ty) = abs\_term\ (mk\_const\_rep(s,\ ty),\ ty)$$

HOL Constant

$$\mathbf{has\_mk\_comb} : (\mathit{TERM} \times \mathit{TERM}) \to \mathit{TERM} \to \mathit{BOOL}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall\ f\ a\ tm \bullet} \\
\multicolumn{2}{l}{has\_mk\_comb\ (f,\ a)\ tm \Leftrightarrow} \\
\exists ty \bullet & rep\_term\ tm = (mk\_comb\_rep(Fst(rep\_term\ f),\ Fst(rep\_term\ a)),\ ty) \\
\wedge & type\_of\_term\ f = Fun\ (type\_of\_term\ a)\ ty
\end{array}$$

HOL Constant

$$\mathbf{has\_mk\_abs} : (\mathit{TERM} \times \mathit{TERM}) \to \mathit{TERM} \to \mathit{BOOL}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall v\ b\ tm \bullet\ has\_mk\_abs\ (v,\ b)\ tm \Leftrightarrow} \\
\multicolumn{2}{l}{(\exists\ s\ tys \bullet\ mk\_var(s,\ tys) = v} \\
\wedge & rep\_term\ tm = \\
 & (mk\_abs\_rep(s,\ tys,\ Fst(rep\_term\ b)),\ Fun\ tys\ (type\_of\_term\ b)))
\end{array}$$

5.4 Instantiation of Types

When we define the type of HOL theories we will need the following function to formulate some of context-sensitive conditions that we will wish to impose.

HOL Constant

$$\mathbf{inst\_type} : (\mathit{STRING} \to \mathit{TYPE}) \to \mathit{TYPE} \to \mathit{TYPE}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall (f\!: STRING \to TYPE) \bullet} \\
 & (\forall s \bullet\ \ inst\_type\ f\ (mk\_var\_type\ s) = f\ s) \\
\wedge & (\forall s\ tl \bullet inst\_type\ f\ (mk\_type(s,\ tl)) = \\
 & \qquad mk\_type(s,\ Map\ (inst\_type\ f)\ tl))
\end{array}$$

6 SYNTAX OF SEQUENTS

The minor complication mentioned in the previous section is that HOL is defined as a sequent calculus. It is the sequents which make up our $L(\Sigma_{max})$.

A sequent is simply a set of assumptions and a conclusion. Assumptions and conclusion alike are just terms. The following definition allows infinite assumption sets, since they are easier for us to define. However the axioms with which we shall work all have finite sets of assumptions and the inference rules will preserve this property. Another pleasant property of sequents is for their constituent terms to have type "*:bool*". This property, too, holds of our axioms and is preserved by our inference rules and when we define theories we insist that the sequents in them have it.

SML

```
declare_type_abbrev("SEQ", [], ⌜:(TERM SET) × TERM⌝);
```

The following functions on sequents are useful for reasons of clarity. Their names are as in the HOL system.

HOL Constant

```
    concl : SEQ → TERM
──────────────────────────
    concl = Snd
```

HOL Constant

```
    hyp : SEQ → (TERM SET)
──────────────────────────
    hyp = Fst
```

7 THEORIES

In this section we define a type *THEORY* whose elements are what we shall think of as the well-formed HOL theories. In our case, the signature part of a theory amounts to two "environments", one giving the arity of the type constructors in the theory and the other giving the types of the constants[4].

The following type abbreviations help us to formalise the context-sensitive aspects of the well-formedness of terms, which we have avoided until now. Once this is done we can define the type of all well-formed HOL theories.

SML

```
declare_type_abbrev("TY_ENV", [], ⌜:STRING ↔ ℕ⌝);

declare_type_abbrev("CON_ENV", [], ⌜:STRING ↔ TYPE⌝);

declare_type_abbrev("SEQS", [], ⌜:SEQ SET⌝);
```

We can now define the well-formedness of types and terms with respect to a type environment. We assume that the names for type variables and type constructors are in distinct lexical classes, and so all we check is the arity of constructors. (HOL implementations may impose additional lexical constraints on the names.)

HOL Constant

```
    wf_type : TY_ENV → TYPE SET
──────────────────────────
    ∀ tyenv •
    wf_type tyenv = ⋂{tyset|
            (∀s • mk_var_type s ∈ tyset)
            ∧
            ∀s tyl •        s ↦ Length tyl ∈ tyenv ∧ (∀ t • t ∈ Elems tyl ⇒ t ∈ tyset)
            ⇒       mk_type(s, tyl) ∈ tyset}
```

[4] These correspond to the *type structures* and *signatures* respectively in [6].

For terms we place no restrictions on the names of variables. (The HOL system tries to prevent constant names being used as variable names but does not always succeed, e.g, if the constant is declared after a theorem using a variable with the same name has been saved on a theory). The polymorphic nature of constants in HOL becomes apparent here in that we may instantiate type variables appearing in the constant environment.

HOL Constant

$$\mathbf{wf\_term}: \mathit{TY\_ENV} \rightarrow \mathit{CON\_ENV} \rightarrow \mathit{TERM\ SET}$$

$$\begin{array}{l}
\forall\ \mathit{tyenv}\ \mathit{conenv} \bullet \\
\mathit{wf\_term}\ \mathit{tyenv}\ \mathit{conenv} = \bigcap\{\mathit{tmset} | \\
\qquad (\forall s\ \mathit{ty} \bullet \mathit{ty} \in \mathit{wf\_type}\ \mathit{tyenv} \Rightarrow \mathit{mk\_var}(s,\ \mathit{ty}) \in \mathit{tmset}) \\
\wedge \\
(\forall s\ \mathit{ty} \bullet (\mathit{ty} \in \mathit{wf\_type}\ \mathit{tyenv} \wedge \exists \mathit{ty}'\ \mathit{tysubs} \bullet\ s \mapsto \mathit{ty}' \in \mathit{conenv} \wedge \mathit{inst\_type}\ \mathit{tysubs}\ \mathit{ty}' = \mathit{ty}) \\
\qquad \Rightarrow \mathit{mk\_const}(s,\ \mathit{ty}) \in \mathit{tmset}) \\
\wedge \\
(\forall f\ a\ \mathit{tm} \bullet (\mathit{has\_mk\_comb}(f,\ a)\ \mathit{tm} \wedge f \in \mathit{tmset} \wedge a \in \mathit{tmset}) \Rightarrow \mathit{tm} \in \mathit{tmset}) \\
\wedge \\
(\forall v\ b\ \mathit{tm} \bullet (\mathit{has\_mk\_abs}(v,\ b)\ \mathit{tm} \wedge v \in \mathit{tmset} \wedge b \in \mathit{tmset}) \Rightarrow \mathit{tm} \in \mathit{tmset})\}
\end{array}$$

The well-formedness of terms extends straightforwardly to sequents and to sets thereof. We impose an additional constraint for sequents: they must be made up from terms of type "$:\mathit{bool}$".

HOL Constant

$$\mathbf{wf\_seq}: \mathit{TY\_ENV} \rightarrow \mathit{CON\_ENV} \rightarrow \mathit{SEQ\ SET}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall\ \mathit{seq}\ \mathit{tyenv}\ \mathit{conenv} \bullet} \\
\multicolumn{2}{l}{\mathit{seq} \in \mathit{wf\_seq}\ \mathit{tyenv}\ \mathit{conenv} \Leftrightarrow} \\
\mathit{let} & \mathit{ok} = \{\mathit{tm} \mid \mathit{tm} \in \mathit{wf\_term}\ \mathit{tyenv}\ \mathit{conenv} \wedge \mathit{type\_of\_term}\ \mathit{tm} = \mathit{Bool}\} \\
\mathit{in} & \mathit{concl}\ \mathit{seq} \in \mathit{ok} \wedge \forall \mathit{tm} \bullet\ \mathit{tm} \in \mathit{hyp}\ \mathit{seq} \Rightarrow \mathit{tm} \in \mathit{ok}
\end{array}$$

The *SEQS* component of a theory is well formed if it is a subset of the set of sequents which are well-formed with respect to the type and constant environments:

HOL Constant

$$\mathbf{wf\_seqs}: \mathit{TY\_ENV} \rightarrow \mathit{CON\_ENV} \rightarrow \mathit{SEQS\ SET}$$

$$\begin{array}{l}
\forall \mathit{tyenv}\ \mathit{conenv} \bullet \\
\mathit{wf\_seqs}\ \mathit{tyenv}\ \mathit{conenv} = \mathbb{P}\ (\mathit{wf\_seq}\ \mathit{tyenv}\ \mathit{conenv})
\end{array}$$

For the constant environments, we insist that the type associated with each name be well-formed and that at most one type is associated with each name (i.e. the environment must be a functional relation). Overloaded constant names could, in principle, be allowed, as an extension to the system. This function would then be modified to impose some weaker condition.

HOL Constant

```
│ wf_con_env: TY_ENV → CON_ENV SET
├──────────────────────────────────
│ ∀ conenv tyenv•
│           conenv ∈ wf_con_env tyenv
│ ⇔         conenv ∈ Functional
│           ∧     ∀ con ty• con ↦ ty ∈ conenv ⇒ ty ∈ wf_type tyenv
```

We insist that at most one arity be associated with each name in a well-formed type environment (i.e. that the environment is a functional relation) :

HOL Constant

```
│ wf_ty_env: TY_ENV SET
├──────────────────────────────────
│ wf_ty_env = Functional
```

We will consider a triple consisting of a type environment, a constant environment and a set of sequents to be a well-formed theory if each constituent is well-formed with respect to its predecessors:

HOL Constant

```
│ is_theory: (TY_ENV × CON_ENV × SEQS) SET
├──────────────────────────────────
│ ∀ty_env con_env axioms•
│           (ty_env, con_env, axioms) ∈ is_theory ⇔
│           ty_env ∈ wf_ty_env ∧
│           con_env ∈ wf_con_env ty_env ∧
│           axioms ∈ wf_seqs ty_env con_env
```

Note that a theory can contain infinitely many types, constants, or axioms. This possibility occurs in practice, at least for constants and axioms. The theory ℕ of natural numbers is an example, since it contains an axiom defining the decimal representation of each positive number.

SML

```
│val thm3 = save_thm("thm3", (
│       set_goal([], ⌜∃thy•thy ∈ is_theory⌝);
│       a(∃_tac ⌜({}:TY_ENV, {}:CON_ENV, {}:SEQS)⌝);
│       a(rewrite_tac (map get_spec[⌜is_theory⌝, ⌜wf_ty_env⌝,
│               ⌜wf_con_env⌝, ⌜wf_seqs⌝,
│               ⌜Dom⌝, ⌜$At⌝, ⌜$⇸⌝, ⌜$↔⌝, ⌜Functional⌝]));
│       pop_thm()
│));
```

SML

```
val theory_def = new_type_defn (["THEORY"], "THEORY", [],
        (tac_proof( ([], ⌜∃thy•(λthy•thy ∈ is_theory) thy⌝),
                rewrite_tac[thm3])));
val abs_theory_rep_theory_def = abs_rep_spec
        {type_def_name = "THEORY",
         abs_fun = "abs_theory",
         rep_fun = "rep_theory",
         def_conv = Value (rewrite_conv[])};
```

We will use the following functions to extract the components of theories:

HOL Constant

```
        axioms : THEORY → SEQS
──────────────────────────────────────
        ∀ thy • axioms thy = Snd(Snd(rep_theory thy))
```

HOL Constant

```
        types : THEORY → TY_ENV
──────────────────────────────────────
        ∀ thy • types thy = Fst(rep_theory thy)
```

HOL Constant

```
        constants : THEORY → CON_ENV
──────────────────────────────────────
        ∀ thy • constants thy = Fst(Snd(rep_theory thy))
```

The following function which returns the set of sequents which are in the language associated with a theory is also useful:

HOL Constant

```
        sequents : THEORY → SEQS
──────────────────────────────────────
        ∀ seq thy •
                seq ∈ sequents thy ⇔
                seq ∈ wf_seq (types thy) (constants thy)
```

8 INDEX OF DEFINED TERMS