HOL Formalised: Deductive System

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

25 October 1993
Revised 17 October 2002

Abstract

This is part of a suite of documents giving a formal specification of the HOL logic. It defines the primitive inference rules, including conservative extension mechanisms. Related notions such as derivability are also defined.

The treatment of the HOL deductive system formally defined here is closely based on the semi-formal treatment in the documentation for the Cambridge HOL system.

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

Reference: DS/FMU/IED/SPC003; issue 1.15

1 DOCUMENT CONTROL

1.1 Contents list

1.2 Document cross references

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

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

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

2 GENERAL

2.1 Scope

This document specifies the HOL deductive system. Some high level aspects of the implementation of the proof development system are also discussed. It is part of a suite of documents specifying the HOL logic, an overview of which may be found in [1].

2.2 Introduction

In [1] a brief theoretical discussion of the definition of deductive systems is given. In this document we fill in the details for HOL.

The first task is to define the rules of inference. HOL has five rules of inference: *ABS*, *DISCH*, *INST_TYPE*, *MP*, *SUBST* (defined in section 4 below) and three axiom schemata: *ASSUME*, *BETA_CONV* and *REFL* (defined in section 5). We follow [3] in treating the axiom schemata just like unary rules of inference. Such rules are a convenient home for infinite families of axioms that we wish to have in every theory.

With the rules of inference in hand, we define derivability in section 6. We then define the type of theorems of HOL as those pairs (s, T) where T is a theory and s is a sequent in the language of T derivable from the axioms of T.

Section 9, defines the type of all theorems and specifies the notions of consistency and conservative extension.

Mechanisms for extending theories by making definitions are of great practical importance, particularly those which preserve consistency. Section 10 discusses the means by which theories may be extended in the HOL system. Of particular importance are certain mechanisms for introducing new constants and types.

In section 11 we define the individual axioms of the HOL logic. The resulting theory is of special interest, as are what we call its definitional extensions, which we define in section 11.3: they are all consistent and have a common standard set-theoretic model; their theorems comprise what are normally taken to be the theorems of HOL by those who shun axiomatic extensions.

3 PREAMBLE

We introduce the new theory. Its parent is the theory *spc001* which contains definitions concerned with the HOL language.

SML

```
open_theory"spc001";
new_theory"spc003";
```

4 THE RULES OF INFERENCE

In this section we treat the syntax manipulating functions required to define the various rules of inference. We consider each inference rule in turn. In the HOL system the inference rules are functions which take theorems (and other things) as arguments and return theorems. Since we cannot define the type of theorems until we have defined the inference rules we define the rules as functions taking sequents (and other things) as arguments and returning sequents.

4.1 Free Variables

freevars_list returns the free variables of a term listed in order of first appearance (from left to right in the usual concrete syntax).

HOL Constant

$$\textbf{freevars\_list}: TERM \rightarrow ((STRING \times TYPE)LIST)$$

$$\begin{array}{l}
\forall s : STRING;\ ty : TYPE;\ tm\ f\ a\ vty\ b : TERM \bullet \\
freevars_list\ (mk_var(s,\ ty)) = [(s,\ ty)] \\
\land \\
freevars_list\ (mk_const(s,\ ty)) = [] \\
\land \\
(has_mk_comb(f,\ a)\ tm \Rightarrow freevars_list\ tm = freevars_list\ f \frown freevars_list\ a) \\
\land \\
((has_mk_abs(vty,\ b)\ tm \land mk_var(s,\ ty) = vty) \Rightarrow \\
\quad\quad freevars_list\ tm = freevars_list\ b \upharpoonright \sim\{(s,\ ty)\})
\end{array}$$

freevars_set returns the set of free variables of a term. We use it in cases where the order of appearance of the free variables in the term is immaterial.

HOL Constant

$$\textbf{freevars\_set}: TERM \rightarrow (STRING \times TYPE)\ SET$$

$$\forall tm : TERM \bullet freevars\_set\ tm = Elems(freevars\_list\ tm)$$

4.2 Object Language Constructs

To define the rules of inference we need to form certain object language types and terms. We have already defined the function space type constructor. The other definitions needed are given in this section.

We need to form instances of the polymorphic constant "=":

HOL Constant

$$\textbf{Equality} : TYPE \rightarrow TERM$$

$$\forall\ ty \bullet Equality\ ty = mk\_const(\texttt{"="},\ Fun\ ty\ (Fun\ ty\ Bool))$$

The following is our analogue of the derived constructor function for equations in the HOL system.

HOL Constant

$$\textbf{has\_mk\_eq} : (TERM \times TERM) \to TERM \to BOOL$$

$$\forall\ lhs\ rhs\ tm \bullet has\_mk\_eq(lhs,\ rhs)\ tm \Leftrightarrow$$
$$\exists\ tm2 \bullet$$
$$\quad has\_mk\_comb(Equality(type\_of\_term\ lhs),\ lhs)\ tm2$$
$$\wedge \quad has\_mk\_comb(tm2,\ rhs)\ tm$$

We also need to form implications. The following functions are analogous to those treating equality above.

HOL Constant

$$\textbf{Implication} : TERM$$

$$Implication = mk\_const(\text{"}\Rightarrow\text{"},\ Fun\ Bool\ (Fun\ Bool\ Bool))$$

HOL Constant

$$\textbf{has\_mk\_imp} : (TERM \times TERM) \to TERM \to BOOL$$

$$\forall\ lhs\ rhs\ tm \bullet has\_mk\_imp(lhs,\ rhs)\ tm \Leftrightarrow$$
$$\exists\ tm2 \bullet$$
$$\quad has\_mk\_comb(Implication,\ lhs)\ tm2$$
$$\wedge \quad has\_mk\_comb(tm2,\ rhs)\ tm$$

4.3 Substitution of Equals

In this section we define the inference rule *SUBST*.

In essence, *SUBST* says that given a theorem whose conclusion is an equation, $\mathcal{A} = \mathcal{B}$, where $\mathcal{A}$ and $\mathcal{B}$ are arbitrary terms of the same type, and given any other theorem with conclusion $\mathcal{C}$, we may obtain a new theorem by substituting $\mathcal{B}$ for any subterm of $\mathcal{C}$ which is identical with $\mathcal{A}$. This is subject to the proviso that no variable capture problems arise, i.e. no free variables of $\mathcal{B}$ should become bound in the conclusion of the new theorem. (The assumption set of the consequent theorem is the union of the assumption sets of the antecedent theorems.)

The inference rule is, in fact, slightly more general. It allows one to use a whole set of theorems whose conclusions are equations to perform (simultaneous) substitutions for many subterms of $\mathcal{C}$. Moreover, it is implemented as a functional relation, effectively by renaming any bound variables of $\mathcal{C}$ which would give rise to the capture problem.

The inference rule is parametrised by a template term and a set of some of its free variables, one for each equation. The actual statement of the rule is, essentially, that, if the result of substituting the left hand sides of the equations for the corresponding variables in the template term is equal to $\mathcal{C}$ (*modulo* renaming bound variables), then we may infer the result of substituting the right hand sides of the equations for the corresponding template variables in the template term (providing we rename bound variables to avoid the capture problem).

The notions we must formalise are therefore: (i) substituting terms for free variables in a term according to a given mapping of variables to terms renaming bound variables as necessary to avoid variable capture; (ii) testing equivalence of terms *modulo* renaming of bound variables (aka. α-conversion).

4.3.1 Substitution

We will need to choose new names for variables. More precisely, given a variable and a set of same we will wish to rename the variable, when necessary, to ensure that the result does not lie in the set. In practice in an implementation we would insist that the new name be derived from the old one in a specified way.

HOL Constant

$$\textbf{variant} : ((STRING \times TYPE)\ SET) \to (STRING \times TYPE) \to STRING$$

$$\begin{array}{l} \forall\ vs\ v\ ty \bullet \\ \quad if\ \neg(v,\ ty) \in vs \\ \quad then\ variant\ vs\ (v,\ ty) = v \\ \quad else\ \neg(variant\ vs\ (v,\ ty),\ ty) \in vs \end{array}$$

Now we can define *subst*. Given a function R associating free variables with terms, $subst\ R\ t1$ is the term resulting from replacing every free variable $mk\_var(s,t)$ in $t1$ by $R(mk\_var(s,t))$ with bound variables renamed as necessary to avoid capture. Variables which are not to be changed correspond to pairs (s,t) with $R(s,t) = mk\_var(s,t)$.

Note R here is intended to respect types, in the sense that $\forall sty \bullet type\_of\_term(R(s,ty)) = ty$, but this is not checked here (since it is convenient for *subst* to be a total function). This property should be checked whenever *subst* is used.

The only difficult case in *subst* is when the second argument is an abstraction. In this case we calculate the variables which must not get captured (this is the value *new_frees* below) and use *variant* to give an alternative name for the bound variable if necessary. We then perform the substitution on the body using a function, *RR*, which is R modified to send the old bound variable to the new one.

HOL Constant

$$\textbf{subst} : ((STRING \times TYPE) \to TERM) \to TERM \to TERM$$

$$\begin{array}{l} \forall\ R :(STRING \times TYPE) \to TERM;\ tm : TERM; \\ s : STRING;\ ty : TYPE;\ vty : TERM; \\ f : TERM;\ a : TERM;\ b : TERM \\ \bullet \\ subst\ R\ (mk\_var(s,\ ty)) = R(s,ty) \\ \wedge \\ subst\ R\ (mk\_const(s,\ ty)) = mk\_const(s,\ ty) \\ \wedge \\ (has\_mk\_comb(f,\ a)\ tm \Rightarrow \\ (subst\ R\ tm = \epsilon t \bullet has\_mk\_comb(subst\ R\ f,\ subst\ R\ a)t)) \\ \wedge \\ ((has\_mk\_abs(vty,\ b)\ tm \wedge mk\_var(s,\ ty) = vty) \Rightarrow \\ (subst\ R\ tm = \\ \quad let\ new\_frees = \bigcup(Graph\ (freevars\_set\ o\ R)\ Image \\ \qquad\qquad\qquad (freevars\_set\ b \setminus \{(s,\ ty)\})) \\ \quad in\ \ let\ s' = variant\ new\_frees\ (s,\ ty) \\ \quad in\ let\ RR\ x = if\ x = (s,\ ty)\ then\ mk\_var\ (s',\ ty)\ else\ R\ x \end{array}$$

in

$\epsilon t \bullet$

has_mk_abs

$(mk\_var(s', ty), subst\ RR\ b)t$

$))$

The special case of substitution where we simply wish to rename a variable is needed in the definition of our α-conversion test and elsewhere. The following function *rename* is used for this purpose. $rename(v, ty) w\ e$ is the result of changing the name in every free occurrence of the variable with name v, and type ty, in the term e, to w, renaming any bound variables as necessary.

HOL Constant

rename : $(STRING \times TYPE) \to STRING \to TERM \to TERM$

$\forall\ v : STRING;\ ty : TYPE;\ w: STRING$

$\bullet$

$rename\ (v, ty)\ w =$

$subst\ (\lambda x \bullet if\ x = (v, ty)\ then\ mk\_var(w, ty)\ else\ mk\_var\ x)$

4.3.2 α-conversion

Our α-conversion test is as follows:

HOL Constant

aconv : $TERM \to TERM \to BOOL$

$\forall t1\ t2 : TERM \bullet$

$aconv\ t1\ t2 \Leftrightarrow$

$(t1 = t2)$

$\lor\ (\exists t1f\ t1a\ t2f\ t2a \bullet$

$has\_mk\_comb(t1f, t1a)t1$

$\land\ has\_mk\_comb(t2f, t2a)t2$

$\land\ aconv\ t1f\ t2f \land aconv\ t1a\ t2a)$

$\lor\ (\exists v1\ v2\ ty\ v1ty\ v2ty\ b1\ b2 \bullet$

$has\_mk\_abs(v1ty, b1)t1 \quad \land \quad has\_mk\_abs(v2ty, b2)t2$

$\land\ mk\_var(v1, ty) = v1ty \quad \land \quad mk\_var(v2, ty) = v2ty$

$\land\ aconv\ b1\ (rename\ (v2, ty)\ v1\ b2)$

$\land\ ((v1 = v2) \lor (\neg(v1, ty) \in freevars\_set\ b2)))$

4.3.3 The Inference Rule *SUBST*

We can now define the inference rule. Its first argument gives the correspondence between the template variables and equation theorems. We could take this argument to behave as $REFL\_axiom$ $o\ mk\_var$ on variables which are not template variables . Note that, to allow implementation as a partial function, we test up to α-convertibility on the first sequent argument only. Note also that the way that the first argument to *subst* is constructed by dismantling equations ensures that it respects types.

HOL Constant

$$\textbf{SUBST\_rule} : ((STRING \times TYPE) \to SEQ) \to TERM \to SEQ \to SEQ \to BOOL$$

$$\begin{array}{l}
\forall\ eqs\ tm\ old_asms\ old_conc\ new_asms\ \ new_conc\bullet \\
SUBST_rule\ eqs\ tm\ (old_asms,\ old_conc)\ (new_asms,\ new_conc) \Leftrightarrow \\
(\forall v\ ty\ \bullet \\
\quad \exists lhs\ rhs\bullet \\
\quad has_mk_eq(lhs,\ rhs)(concl(eqs(v,\ ty)))\ \wedge \\
\quad (type_of_term\ lhs\ =\ ty)) \\
\wedge \\
(aconv\ old_conc\ (subst(\lambda(v,ty)\bullet\epsilon lhs\bullet\exists rhs\bullet has_mk_eq(lhs,\ rhs)(concl(eqs(v,ty))))tm)) \\
\wedge \\
(new_conc\ =\ subst\ (\lambda(v,ty)\bullet\epsilon rhs\bullet\exists lhs\bullet has_mk_eq(lhs,\ rhs)(concl(eqs(v,ty))))tm) \\
\wedge \\
(new_asms\ =\ old_asms\ \cup \bigcup\ \{asms\ \mid\ \exists vty\bullet asms\ =\ (hyp\ \ (eqs\ \ vty))\})
\end{array}$$

4.4 Abstraction: ABS

Again *ABS* is a partial function which we specify as a relation:

HOL Constant

$$\textbf{ABS\_rule} : (STRING \times TYPE) \to SEQ \to SEQ \to BOOL$$

$$\begin{array}{l}
\forall\ vty\ old_asms\ old_conc\ new_asms\ new_conc\ \bullet \\
ABS_rule\ vty\ (old_asms,\ old_conc)\ (new_asms,\ new_conc) \Leftrightarrow \\
(\exists\ old_lhs\ old_rhs\ new_lhs\ new_rhs\ v\bullet \\
\quad has_mk_eq(old_lhs,\ old_rhs)old_conc\ \wedge \\
\quad has_mk_eq(new_lhs,\ new_rhs)new_conc\ \wedge \\
\quad mk_var\ vty\ =\ v\ \wedge \\
\quad has_mk_abs(v,\ old_lhs)\ new_lhs\ \wedge \\
\quad has_mk_abs(v,\ old_rhs)\ new_rhs) \\
\wedge \\
(\neg vty\ \in\ \bigcup(Graph\ freevars_set\ Image\ old_asms)) \\
\wedge \\
(new_asms\ =\ old_asms)
\end{array}$$

4.5 Type Instantiation

The ability to prove and use general (polymorphic) theorems is one of the great strengths of the HOL system. The feature in the inference system which gives this strength is the inference rule *INST_TYPE* which allows us to instantiate the type variables in the conclusion of a polymorphic theorem.

In essence, the inference rule says that, given a theorem with conclusion, $\mathcal{A}$, say, we may infer the theorem which has the same assumption set and whose conclusion results from instantiating every type in $\mathcal{A}$ according to a given mapping of type variables to types. This is subject to two provisos: (i) no type variable may be changed which appears in the assumption set for the theorem; (ii) no two variables in the assumptions or conclusion of the antecedent theorem, which are different, by virtue of their type, should become identified in the consequent theorem as a result of the transformation.

The first proviso is, we believe, only enforced to preserve a convention of natural deduction systems, whereby inference rules involve only simple set operations on the assumption sets. It would seem to be quite in order for the first proviso to be dropped provided we insisted that the type instantiation be applied to every term in the sequent (we have, of course, not done this).

The second proviso cannot be avoided. Consider for example: $\lambda(x : **)\bullet\lambda(x : *)\bullet(x : **)$. If the types in this were instantiated according to $\{: ** \mapsto : *, : * \mapsto : *\}$, then from:

$$\vdash \forall(\mathbf{y} : **)(\mathbf{z} : *)\bullet(\lambda(\mathbf{x} : **)\bullet\lambda(\mathbf{x} : *)\bullet(\mathbf{x} : **))\mathbf{yz} = \mathbf{y}$$

we could infer that:

$$\vdash \forall(\mathbf{y} : *)(\mathbf{z} : *)\bullet(\lambda(\mathbf{x} : *)\bullet\lambda(\mathbf{x} : *)\bullet(\mathbf{x} : *))\mathbf{yz} = \mathbf{y}$$

whence, by β-conversions:

$$\vdash \forall(\mathbf{y} : *)(\mathbf{z} : *)\bullet\mathbf{z} = \mathbf{y}.$$

This leads to a contradiction whenever $: *$ is instantiated to a type with more than one inhabitant.

To permit an implementation which is convenient to use, the inference rule is actually formulated without the second proviso. Instead, variables (both free and bound, in general) in the conclusion of the consequent theorem, which would violate the rule are renamed to avoid the problem. It is valid to rename free variables in these circumstances, given the first proviso, since the variables in question cannot occur free in the assumption set. Note that it would be invalid to rename free variables in $\mathcal{A}$ which are not changed by the type instantiation (since these may appear free in the assumption set).

Formalising these notions is a little tricky. We present here a highly unconstructive specification, reminiscent of α-conversion. The notion to be formalised is the predicate on pairs of terms which says whether one is a type instance of another according to a given mapping of type variables to types and with respect to a set of variables with which clashes must not occur (this will be the set of free variables of the assumptions in practice).

It is entertaining and instructive to consider algorithms meeting these specifications.

4.5.1 Instantiation of Terms

Instantiation of terms is a little tricky. The following two functions should be viewed as local to the function *inst*. *inst_loc1* is very similar to an α-convertibility test. Indeed *aconv* could have been defined as *inst_loc1 I*. The first *TERM* argument of *inst_loc1* and *inst_loc2* gives the terms whose types are being instantiated (i.e. it is the "more polymorphic" term).

inst_loc1 checks that one term, *tm2*, is a type instance of *tm1*, according to a mapping from type variable names to types given by *tysubs*, under the assumption that the free variable names agree, i.e. that the first occurrence of each variable which may need renaming will be its binding occurrence in a $\lambda - abstraction$.

HOL Constant

```
inst_loc1 : (STRING → TYPE) → TERM → TERM → BOOL
─────────────────────────────
∀
tysubs : STRING → TYPE;
tm1 tm2 : TERM•
inst_loc1 tysubs tm1 tm2 ⇔
        (∃s ty1 ty2 mk_X•
                ((mk_X = mk_var) ∨ (mk_X = mk_const))
        ∧       mk_X(s, ty1) = tm1 ∧ mk_X(s, ty2) = tm2
        ∧       (ty2 = inst_type tysubs ty1))
∨       (∃tm1f tm1a tm2f tm2a•
                has_mk_comb(tm1f, tm1a)tm1 ∧ has_mk_comb(tm2f, tm2a)tm2
        ∧       inst_loc1 tysubs tm1f tm2f ∧ inst_loc1 tysubs tm1a tm2a)
∨       (∃v1 v2 ty1 ty2 b1 b2 v1ty1 v2ty2 •
                mk_var(v1, ty1) = v1ty1 ∧ has_mk_abs(v1ty1, b1)tm1
        ∧       mk_var(v2, ty2) = v2ty2 ∧ has_mk_abs(v2ty2, b2)tm2
        ∧       inst_loc1 tysubs (rename (v1, ty1) v2 b1) b2
        ∧       (ty2 = inst_type tysubs ty1)
        ∧       ¬(∃ ty3 v2ty3 •
                        mk_var(v2, ty3) = v2ty3
                ∧       ((v2, ty3) ∈ freevars_set b1)
                ∧       (ty2 = inst_type tysubs ty3)
                ∧       (¬v2ty3 = v1ty1) ))
```

inst_loc2 uses *inst_loc1* to check that a term *tm2* is a type instance of the result of renaming free variables of a term *tm2* according to a mapping given by a list of pairs. It also checks that the type of the second variable in each pair in the list is a type instance of the type of the first variable in the pair, and that the second variable in each pair is not in the set, *avoid*, unless both names and types agree for that pair. In the application of *inst_loc2* in *inst* the list of pairs is obtained by combining the free variable lists of the two terms side by side. The set *avoid* is a set of variables (coming from the assumptions of a sequent) whose free occurrences must not change as a result of the type instantiation.

HOL Constant

```
inst_loc2 : ((STRING × TYPE) SET) →
        (STRING → TYPE) →
        (((STRING × TYPE) × (STRING × TYPE)) LIST) →
        TERM → TERM → BOOL
─────────────────────────────
∀avoid : (STRING × TYPE) SET;
tysubs :STRING → TYPE;
v1 : STRING; ty1 : TYPE;
v2 : STRING; ty2 : TYPE;
rest : ((STRING × TYPE) × (STRING × TYPE)) LIST;
tm1 tm2 : TERM•
```

$(inst\_loc2\ avoid\ tysubs\ []\ tm1\ tm2 \Leftrightarrow$
$\quad inst\_loc1\ tysubs\ tm1\ tm2)$
$\wedge$
$(inst\_loc2\ avoid\ tysubs\ (Cons\ ((v1,\ ty1),(v2,\ ty2))\ rest)\ tm1\ tm2 \Leftrightarrow$
$\quad (((v2,\ ty2) \in avoid) \Rightarrow ((v1,\ ty1) = (v2,\ ty2)))$
$\wedge \quad (ty2 = inst\_type\ tysubs\ ty1)$
$\wedge \quad inst\_loc2\ avoid\ tysubs\ rest$
$\quad (rename\ (v1,\ ty1)\ v2\ tm1)\ tm2)$

With the above preliminaries we can now define *inst*. Note that the condition that the free variable lists of the two terms have the same length is required to ensure that *inst_loc2* examines each free variable of each term.

HOL Constant

$\mathbf{inst} : ((STRING \times TYPE)\ SET) \rightarrow$
$\quad (STRING \rightarrow TYPE) \rightarrow TERM \rightarrow TERM$

$\forall avoid : (STRING \times TYPE)\ SET;$
$tysubs : STRING \rightarrow TYPE;\ tm1 : TERM \bullet$
$let\ tm2 = inst\ avoid\ tysubs\ tm1$
$in\ let\ fl1 = freevars\_list\ tm1$
$in\ let\ fl2 = freevars\_list\ tm2$
in
$\quad ((Length\ fl1 = Length\ fl2)$
$\wedge \quad inst\_loc2\ avoid\ tysubs\ (Combine\ fl1\ fl2)\ tm1\ tm2)$

4.5.2 The Inference Rule *INST_TYPE*

Given *inst*, we need a few simple auxiliaries before we can define the inference rule *INST_TYPE*.

We need to detect the type variables in a term. We use some auxiliary functions to do this: *type_tyvars* detects the type variables in a type.

HOL Constant

$\mathbf{type\_tyvars} : TYPE \rightarrow (STRING\ SET)$

$\quad (\forall s \bullet\ type\_tyvars\ (mk\_var\_type\ s) = \{s\})$
$\wedge \quad (\forall s\ tl \bullet\ type\_tyvars\ (mk\_type(s,\ tl)) =$
$\quad\quad \bigcup\ (Elems\ (Map\ type\_tyvars\ tl)))$

term_types detects the types in a term.

HOL Constant

$$\textbf{term\_types} : TERM \to (TYPE\ SET)$$

$$\begin{array}{l}
\forall\ tm : TERM;\ s\!: STRING;\ ty : TYPE;\\
f : TERM;\ a : TERM;\ v\!: TERM;\ b\!: TERM \bullet\\
term_types\ (mk_var(s,\ ty)) = \{ty\}\\
\land\\
term_types\ (mk_const(s,\ ty)) = \{ty\}\\
\land\\
(has_mk_comb(f,\ a)\ tm \Rightarrow (term_types\ tm = term_types\ f \cup term_types\ a))\\
\land\\
(has_mk_abs(v,\ b)\ tm \Rightarrow (term_types\ tm = term_types\ v \cup term_types\ b))
\end{array}$$

term_tyvars detects all the type variables in a term using the previous two functions.

HOL Constant

$$\textbf{term\_tyvars} : TERM \to (STRING\ SET)$$

$$\forall tm \bullet term\_tyvars\ tm = \bigcup(Graph\ type\_tyvars\ Image\ (term\_types\ tm))$$

INST_TYPE_rule is now readily defined:

HOL Constant

$$\textbf{INST\_TYPE\_rule} : (STRING \to TYPE) \to SEQ \to SEQ \to BOOL$$

$$\begin{array}{l}
\forall\ tysubs\ old_asms\ old_conc\ new_seq \bullet\\
INST_TYPE_rule\ tysubs\ (old_asms,\ old_conc)\ new_seq \Leftrightarrow\\
(\forall\ tyv \bullet\\
\qquad (tyv \in \bigcup\ (Graph\ term_tyvars\ Image\ old_asms)) \Rightarrow\\
\qquad (tysubs\ tyv = mk_var_type\ tyv))\\
\land\\
let\ asms_frees = \bigcup\ (Graph\ freevars_set\ Image\ old_asms)\\
in\\
\qquad new_seq = (old_asms,\ inst\ asms_frees\ tysubs\ old_conc)
\end{array}$$

4.6 Discharging an Assumption: DISCH

DISCH is, in essence, the usual rule of natural deduction which allows one to infer from a proof of $\mathcal{B}$ on the assumption $\mathcal{A}$, that $\mathcal{A}\Rightarrow\mathcal{B}$ on no assumption. The actual rule is suitably generalised to cover sequents and their assumption sets. It is not required that $\mathcal{A}$ be in the assumption set, and the logic would probably not be complete otherwise.

HOL Constant

$$\textbf{DISCH\_rule} : TERM \to SEQ \to SEQ \to BOOL$$

$$\forall\ tm\ old\_asms\ old\_conc\ new\_seq\ \bullet$$
$$DISCH\_rule\ tm\ (old\_asms,\ old\_conc)\ new\_seq \Leftrightarrow$$
$$(type\_of\_term\ tm = Bool) \wedge$$
$$(new\_seq = ((old\_asms \setminus \{tm\}),\ \epsilon t \bullet has\_mk\_imp(tm,\ old\_conc)t))$$

4.7 Modus Ponens: MP

This is the usual rule: from $\mathcal{A} \Rightarrow \mathcal{B}$ and $\mathcal{A}$, infer $\mathcal{B}$. This generalises to sequents by taking the union of the assumption sets.

HOL Constant

$$\textbf{MP\_rule} : SEQ \to SEQ \to SEQ \to BOOL$$

$$\forall\ imp\_asms\ imp\_conc\ ant\_asms\ ant\_conc\ new\_asms\ new\_conc\ \bullet$$
$$MP\_rule\ (imp\_asms,\ imp\_conc)\ (ant\_asms,\ ant\_conc)\ (new\_asms,\ new\_conc) \Leftrightarrow$$
$$(has\_mk\_imp(ant\_conc,\ new\_conc)imp\_conc) \wedge$$
$$(new\_asms = imp\_asms \cup ant\_asms)$$

5 THE AXIOM SCHEMATA

5.1 The Axiom Schema ASSUME

ASSUME allows us to infer for any boolean term $\mathcal{A}$, that $\mathcal{A}$ holds on the assumptions $\{\mathcal{A}\}$. This is straightforward to formalise. We must check that the term being assumed is of the right type.

HOL Constant

$$\textbf{ASSUME\_axiom} : TERM \to SEQ \to BOOL$$

$$\forall\ tm\ seq\ \bullet\ ASSUME\_axiom\ tm\ seq \Leftrightarrow$$
$$(type\_of\_term\ tm = Bool) \wedge$$
$$(seq = (\{tm\},\ tm))$$

5.2 The Axiom Schema REFL

REFL says that for any term $\mathcal{A}$, we may infer that $\mathcal{A} = \mathcal{A}$ without assumptions.

HOL Constant

$$\textbf{REFL\_axiom} : TERM \to SEQ$$

$$\forall\ tm\ \bullet\ REFL\_axiom\ tm = (\{\},\ \epsilon t \bullet has\_mk\_eq(tm,\ tm)t)$$

5.3 The Axiom Schema BETA_CONV

BETA_CONV says that, without any assumptions, any β-redex is equal to its β-reduction. This is straightforward to define, given the apparatus we used to define *SUBST*. Note that the way we construct the first argument to *subst* by dismantling a combination ensures that it respects types.

HOL Constant

$$\textbf{BETA\_CONV\_axiom} : TERM \to SEQ \to BOOL$$

$$
\begin{array}{l}
\forall\ tm\ new\_seq\bullet \\
BETA\_CONV\_axiom\ tm\ new\_seq \Leftrightarrow \\
\exists\ v\ ty\ vty\ b\ abs\ a\ \bullet \\
mk\_var(v,\ ty) = vty\ \wedge \\
has\_mk\_abs(vty,\ b)abs\ \wedge \\
has\_mk\_comb(abs,\ a)tm\ \wedge \\
(new\_seq = \\
let\ subs\!: ((STRING \times TYPE) \to TERM) = \\
\quad (\lambda(vx,\ tyx)\bullet if\ \ vx = v \wedge tyx = ty\ then\ a\ else\ mk\_var(vx,\ tyx)) \\
in \\
\quad (\{\},\ (\epsilon t\bullet has\_mk\_eq(tm,\ subst\ subs\ b)t)))
\end{array}
$$

6 DERIVABILITY

In this section we will define derivability. This is a relation between sets of sequents and sequents. As usual, we first define direct derivability. We include instances of the axiom schemata as valid direct derivations from no premisses. This is merely for convenience, we could equally well include all instances of the axiom schemata as axioms in every theory when theories are defined.

HOL Constant

$$\textbf{directly\_derivable\_from} : SEQ \to (SEQ\ SET) \to BOOL$$

$$
\begin{array}{l}
\forall\ seq\ seqs\ \bullet \\
directly\_derivable\_from\ seq\ seqs \Leftrightarrow \\
(\exists\ eqs\ tm\ old\_seq\ \bullet \\
Ran\ (Graph\ eqs) \subseteq seqs \wedge old\_seq \in seqs \wedge SUBST\_rule\ eqs\ tm\ old\_seq\ seq) \\
\vee \\
(\exists\ vty\ old\_seq\ \bullet\ old\_seq \in seqs \wedge ABS\_rule\ vty\ old\_seq\ seq) \\
\vee \\
(\exists\ tysubs\ old\_seq\ \bullet\ old\_seq \in seqs \wedge INST\_TYPE\_rule\ tysubs\ old\_seq\ seq) \\
\vee \\
(\exists\ tm\ old\_seq\ \bullet\ old\_seq \in seqs \wedge DISCH\_rule\ tm\ old\_seq\ seq) \\
\vee \\
(\exists\ imp\_seq\ ant\_seq\ \bullet \\
imp\_seq \in seqs \wedge ant\_seq \in seqs \wedge MP\_rule\ imp\_seq\ ant\_seq\ seq) \\
\vee \\
(\exists\ tm\ \bullet\ ASSUME\_axiom\ tm\ seq)
\end{array}
$$

$$\begin{array}{l} \lor \\ (\exists\ tm \bullet seq = REFL\_axiom\ tm) \\ \lor \\ (\exists\ tm \bullet BETA\_CONV\_axiom\ tm\ seq) \end{array}$$

Proofs will just be lists of sequents. Any non-empty list is a valid proof (of the sequent at its head) on the premisses given by those elements of the list which are not directly derivable from elements later in the list. There is little point in making the relevant type definition for a syntactic class of proofs in this sense, since they contain so little information. We simply define the function which extracts the set of premisses.

HOL Constant

$$\textbf{premisses} : (SEQ\ LIST) \to (SEQ\ SET)$$

$$\begin{array}{l} \forall\ seq\ rest \bullet \\ premisses\ [] = \{\} \\ \land \\ premisses\ (Cons\ seq\ rest) = \\ if\ directly\_derivable\_from\ seq\ (Elems\ rest) \\ then\ premisses\ rest \\ else\ \{seq\} \cup premisses\ rest \end{array}$$

HOL Constant

$$\textbf{derivable\_from} : SEQ \to (SEQ\ SET) \to BOOL$$

$$\begin{array}{l} \forall\ seq\ seqs \bullet \\ derivable\_from\ seq\ seqs = \\ \exists\ seql \bullet premisses\ (Cons\ seq\ seql) \subseteq seqs \end{array}$$

7 NORMAL THEORIES

In [1] a type *THEORY* is defined to represent the idea of a theory comprising signatures governing the formation of types and terms and a set of axioms. However the type *THEORY* is too general for our present purposes, since we have formulated rules of inference on the assumption that the nullary type ":*bool*" and the constants "=" and "⇒" are available. In this section we define a predicate *normal_theory* which selects the theories in which the inference rules are intended to be valid. (The normal theories correspond to those whose type structures and signatures are standard in the terminology of [3]. Unfortunately the term *standard theory* is used for a stronger notion in [3].)

7.1 Object Language Constructs

To define the type of all well-formed HOL theories we need two further object language constructs: the choice function "ϵ" and the type of individuals": *ind*". These are required since we will follow [3] in insisting on the presence of the equality, implication and choice functions in each theory. It is noteworthy however that neither the rules of inference nor the standard conservative extension mechanisms require choice or the individuals; they are only used in the axioms given in section 11.

HOL Constant

Star : *TYPE*

Star = *mk_var_type* "∗"

HOL Constant

Choice : *TERM*

Choice = *mk_const*(("ϵ", *Fun* (*Fun Star Bool*) *Star*))

HOL Constant

Ind : *TYPE*

Ind = *mk_type*("*ind*", [])

7.2 Normal Thoeries

We now wish to define the predicate *normal_theory*. It is natural to say that the normal theories are those which extend the minimal normal theory which contains only ":*bool*", "=" etc. Thus we must define this minimal normal theory and also the notion of extension of theories.

MIN is the minimal normal theory. It is represented by the triple *MIN_REP*:

HOL Constant

MIN_REP : *TY_ENV* × *CON_ENV* × *SEQS*

```
MIN_REP = (
        {("bool", 0); ("→", 2 ); ("ind", 0 )},
        {("=", Fun Star (Fun Star Bool));
         ("⇒", Fun Bool (Fun Bool Bool));
         ("ϵ", Fun (Fun Star Bool) Star)},
        {}
)
```

HOL Constant

MIN : *THEORY*

MIN = *abs_theory MIN_REP*

Extension for objects of type *THEORY* is the following binary relation:

SML

```
declare_infix(200, "extends");
```

HOL Constant

$\$\mathbf{extends} : \mathit{THEORY} \to \mathit{THEORY} \to \mathit{BOOL}$

$$\forall\ thy1\ thy2 \bullet$$
$$thy1\ extends\ thy2 \Leftrightarrow$$
$$(types\ thy2 \subseteq types\ thy1) \land$$
$$(constants\ thy2 \subseteq constants\ thy1) \land$$
$$(axioms\ thy2 \subseteq axioms\ thy1)$$

The normal theories are those which extend the minimal theory *MIN*. Note that we do not exclude inconsistent theories here. (This corresponds to the possibility of introducing inconsistent axioms in the HOL system).

HOL Constant

$\mathbf{is\_normal\_theory} : \mathit{THEORY\ SET}$

$$\forall thy \bullet thy \in is\_normal\_theory = thy\ extends\ MIN$$

8 THEOREMS

We can, at last, define the type of all HOL theorems. A theorem will consist of a sequent and a theory. The type is the subtype of the type of all such pairs in which the sequent is well-formed with respect to the type and constant environments of the theory and in which the sequent may be derived from the axioms of the theory.

HOL Constant

$\mathbf{is\_thm} : (\mathit{SEQ} \times \mathit{THEORY})\ \mathit{SET}$

$$\forall seq\ thy \bullet$$
$$(seq,\ thy) \in is\_thm \Leftrightarrow$$
$$thy \in is\_normal\_theory$$
$$\land$$
$$seq \in sequents\ thy$$
$$\land$$
$$derivable\_from\ seq\ (axioms\ thy)$$

Note that if (seq, thy) is a theorem in this sense, the derivation of *seq* from the axioms of *thy* may involve sequents which are not well-formed with respect to *thy* (i.e. which contain type operators or constants which are not in *thy*). This is allowed since it simplifies the definition of derivability and makes no difference to the set of theorems in a given theory (this is essentially the fact that the extension mechanisms *new_type* and *new_constant* are conservative).

Proving that $\exists thm \bullet thm \in is\_thm$ involves rather more work than has been involved in previous type definitions. (A witness is easy to supply, e.g. $(REFL\_axiom\,(mk\_var(`x, Star)), MIN)$ would do. However, to show that it is a witness we need to compute *sequents MIN* and to do this we must show that *MIN_REP* is indeed the representative of a theory and checking the conditions on the two environments is rather long-winded). For the time being we therefore defer this proof task and use *type_spec* to define the type, *THM*, of theorems.

SML

```
type_spec {rep_fun="rep_thm", def_tm = ⌜
        THM ≃ mk_thm Of is_thm
⌝
};
```

The components of a theorem are extracted using the following functions:

HOL Constant

```
        thm_seq : THM → SEQ
├──────────────────────────────
        ∀ thm •
        thm_seq thm = Fst(rep_thm thm)
```

HOL Constant

```
        thm_thy : THM → THEORY
├──────────────────────────────
        ∀ thm •
        thm_thy thm = Snd(rep_thm thm)
```

9 CONSISTENCY AND CONSERVATIVE EXTENSION

A theory is consistent if not every sequent which is well-formed in it can be derived from the axioms:

HOL Constant

```
        consistent_theory : THEORY SET
├──────────────────────────────
        ∀ thy •
        thy ∈ consistent_theory ⇔
        ∃ seq •
        (seq ∈ sequents thy)
        ∧
        ¬(derivable_from seq (axioms thy))
```

An extension of a theory is conservative if no sequent of the smaller theory is provable in the larger but not in the smaller.

SML

```
declare_infix(200, "conservatively_extends");
```

HOL Constant

$\$$**conservatively_extends** : $THEORY \to THEORY \to BOOL$

$\forall$ *thy1 thy2*$\bullet$
thy1 conservatively_extends thy2 $\Leftrightarrow$
(*thy1 extends thy2*) $\land$
($\forall$ *seq* $\bullet$
(*seq* $\in$ *sequents thy2*) $\Rightarrow$
(*derivable_from seq* (*axioms thy1*)) $\Rightarrow$
(*derivable_from seq* (*axioms thy2*)))

10 DEFINITIONAL EXTENSIONS

10.1 Object Language Constructs

A theory *LOG* in which more of the standard logical apparatus is available will be needed to define some of the definitional extension mechanisms. For example, *new_type_definition* works with a theorem whose conclusion must be an existentially quantified term of a particular form. To define *LOG* we need some more object language types and terms and these are defined in this section. (It is convenient to leave the definition of *LOG* itself until we have defined *new_definition*.)

The formulation of the various logical connectives follows the HOL manual, [3].

It is helpful now to have the following term constructor functions. Note that we are now using total functions to approximate partial ones; we must, therefore, be careful only to apply them to appropriate arguments.

HOL Constant

mk_comb : $(TERM \times TERM) \to TERM$

---

mk_comb = $\$\epsilon$ *o has_mk_comb*

HOL Constant

mk_abs : $(TERM \times TERM) \to TERM$

---

mk_abs = $\$\epsilon$ *o has_mk_abs*

HOL Constant

mk_eq : $(TERM \times TERM) \to TERM$

---

mk_eq = $\$\epsilon$ *o has_mk_eq*

HOL Constant

mk_imp : $(TERM \times TERM) \to TERM$

---

mk_imp = $\$\epsilon$ *o has_mk_imp*

We can now define the object language constructs needed. (These could be defined via our explicit representations of types and terms using strings. This has not been done since the explicit concrete syntax used is very hard to read.)

10.1.1 Truth

The constant $T : bool$ is defined by the following equation:

$$\mathbf{T} = ((\lambda(\mathbf{x} : \mathbf{bool})\bullet\mathbf{x}) = (\lambda(\mathbf{x} : \mathbf{bool})\bullet\mathbf{x}))$$

HOL Constant

Truth : *TERM*

Truth = *mk_const*("*T*", *Bool*)

HOL Constant

Truth_def : *TERM*

Truth_def =
let x = *mk_var*("*x*", *Bool*)
in
mk_eq(*mk_abs*(*x*, *x*), *mk_abs*(*x*, *x*))

10.1.2 Universal Quantification

The constant $\forall : (*\to bool)\to bool$ is defined by the following equation:

$$\$\forall = (\lambda(\mathbf{P} : *\to\mathbf{bool})\bullet\mathbf{P} = (\lambda(\mathbf{x} : *)\bullet\mathbf{T})$$

HOL Constant

Forall : *TYPE* → *TERM*

∀ *ty*•*Forall ty* = *mk_const*("∀", *Fun* (*Fun ty Bool*) *Bool*)

HOL Constant

Forall_def : *TERM*

Forall_def =
let P = *mk_var*("*P*", *Fun Star Bool*)
in let x = *mk_var*("*x*", *Star*)
in
mk_abs(*P*, *mk_eq*(*P*, *mk_abs*(*x*, *Truth*)))

HOL Constant

mk_forall : (*TERM* × *TERM*) → *TERM*

∀ *tm1 tm2*•*mk_forall*(*tm1*, *tm2*) =
mk_comb(*Forall* (*type_of_term tm1*), *mk_abs*(*tm1*, *tm2*))

10.1.3 Existential Quantification

The constant $\exists : (*\to bool)\to bool$ is defined by the following equation, which defines $\exists$ in terms of the choice function $\epsilon : (*\to bool)\to *$:

$$\$\exists = \lambda(\mathbf{P} : *\to\mathbf{bool})\bullet\mathbf{P}(\epsilon\mathbf{P})$$

(This may be a little perplexing at first sight. In the intended interpretations, given a predicate $P : *\to bool$, if there is some $x : *$ for which P is true (i.e. for which $Px = T$), then ϵP is such an x. I.e. taking as known the intuitive notion of "whether or not something with a given property exists", ϵ chooses something with a given property if such a thing exists. The above definition can be viewed as taking as known the informal notion of "choosing something with a given property" and defining $\exists$ to determine whether or not something with a given property exists by attempting to choose something with the given property and checking whether the attempt succeeded.)

HOL Constant

Exists : *TYPE* → *TERM*

∀ *ty* • *Exists ty* = *mk_const*("∃", *Fun* (*Fun ty Bool*) *Bool*)

HOL Constant

Exists_def : *TERM*

Exists_def =
let P = *mk_var*("*P*", *Fun Star Bool*)
in let PchoiceP = *mk_comb*(*P*,*mk_comb*(*Choice*, *P*))
in
mk_abs(*P*, *PchoiceP*)

HOL Constant

has_mk_exists : (*TERM* × *TERM*) → *TERM* → *BOOL*

∀ *tm1 tm2 tm3* •
has_mk_exists(*tm1*, *tm2*) *tm3* =
has_mk_comb(*Exists* (*type_of_term tm1*), *mk_abs*(*tm1*, *tm2*))*tm3*

HOL Constant

mk_exists : (*TERM* × *TERM*) → *TERM*

∀ *tm1 tm2*•*mk_exists*(*tm1*, *tm2*) =
mk_comb(*Exists* (*type_of_term tm1*), *mk_abs*(*tm1*, *tm2*))

10.1.4 Falsity

The constant $F : bool$ is defined by the following equation:

$$\mathbf{F} = \forall(\mathbf{x} : \mathbf{bool})\bullet\mathbf{x}$$

(Again this may seem perplexing. The type *bool* is intended to contain the truth values. The above definition says that false is the truth value of the proposition that every truth value is true!)

HOL Constant

Falsity : *TERM*

Falsity = *mk_const*("*F*", *Bool*)

HOL Constant

Falsity_def : *TERM*

Falsity_def =
let x = *mk_var*("*x*", *Bool*)
in
mk_forall(*x*, *x*)

10.1.5 Negation

The constant ¬ : *bool*→*bool* is defined by the following equation:

$$\$\neg = \lambda(\mathbf{b} : \mathbf{bool})\bullet\mathbf{b}\Rightarrow\mathbf{F}$$

HOL Constant

Negation : *TERM*

Negation = *mk_const*("¬", *Fun Bool Bool*)

HOL Constant

Negation_def : *TERM*

Negation_def =
let b = *mk_var*("*b*", *Bool*)
in
mk_abs(*b*, *mk_imp*(*b*, *Falsity*))

10.1.6 Conjunction

The constant ∧ : *bool*→*bool*→*bool* is defined by the following equation:

$$\$\wedge = \lambda\mathbf{b1}\bullet\lambda\mathbf{b2}\bullet\forall\mathbf{b}\bullet(\mathbf{b1}\Rightarrow(\mathbf{b2}\Rightarrow\mathbf{b}))\Rightarrow\mathbf{b}$$

(I assume, but do not know, that the above formulation has some practical advantage in the present context over the more obvious definition in terms of ¬ and ⇒.)

The name of the constant is a slash, /, followed by a backslash, \. The backslash character must be escaped by another backslash character within an HOL string.

HOL Constant

Conjunction : *TERM*

$Conjunction = mk\_const(\texttt{"/\\\\"},\ Fun\ Bool\ (Fun\ Bool\ Bool))$

HOL Constant

Conjunction_def : *TERM*

$Conjunction\_def\ =$
$let\ b = mk\_var(\texttt{"}b\texttt{"},\ Bool)$
$in\ let\ b1 = mk\_var(\texttt{"}b1\texttt{"},\ Bool)$
$in\ let\ b2 = mk\_var(\texttt{"}b2\texttt{"},\ Bool)$
in
$mk\_abs(b1,\ mk\_abs(b2,\ mk\_forall(b,\ mk\_imp(mk\_imp(b1,\ mk\_imp(b2,\ b)),\ b))))$

A derived constructor function for conjunctions is useful.

HOL Constant

mk_conj : $(TERM \times TERM) \to TERM$

$\forall\ tm1\ tm2 \bullet$
$mk\_conj(tm1,\ tm2) = mk\_comb(mk\_comb(Conjunction,\ tm1),tm2)$

10.1.7 Disjunction

The constant $\vee$: *bool*→*bool*→*bool* is defined by the following equation:

$$\$\vee = \lambda \mathbf{b1} \bullet \lambda \mathbf{b2} \bullet \forall \mathbf{b} \bullet ((\mathbf{b1} \Rightarrow \mathbf{b}) \Rightarrow (\mathbf{b2} \Rightarrow \mathbf{b})) \Rightarrow \mathbf{b}$$

(As for conjunction I assume this has some advantage over a definition from the propositional calculus.)

The name of the constant is a backslash, \, followed by a slash, /. The backslash character must be escaped by another backslash character within an HOL string.

HOL Constant

Disjunction : *TERM*

$Disjunction = mk\_const(\texttt{"\\\\/"},\ Fun\ Bool\ (Fun\ Bool\ Bool))$

HOL Constant

Disjunction_def : *TERM*

Disjunction_def =
let b = *mk_var*("*b*", *Bool*)
in let b1 = *mk_var*("*b1*", *Bool*)
in let b2 = *mk_var*("*b2*", *Bool*)
in
mk_abs(*b1*, *mk_abs*(*b2*, *mk_forall*(*b*, *mk_imp*(*mk_imp*(*b1*, *b*),
mk_imp(*mk_imp*(*b2*, *b*), *b*)))))

A derived constructor function for disjunctions is useful later.

HOL Constant

mk_disj : (*TERM* × *TERM*) → *TERM*

∀ *tm1 tm2*•
mk_disj(*tm1*, *tm2*) = *mk_comb*(*mk_comb*(*Disjunction*, *tm1*),*tm2*)

10.1.8 ONE_ONE

The definition of *Type_Definition* below requires the notion of a one-to-one function. The constant *ONE_ONE* is defined by the following equation:

$$\mathbf{ONE\_ONE} = \lambda(\mathbf{f} : * \to **)\bullet\forall(\mathbf{x}1 : *)\bullet\forall(\mathbf{x}2 : *)\bullet(\mathbf{f}\,\mathbf{x}1 = \mathbf{f}\,\mathbf{x}2)\Rightarrow(\mathbf{x}1 = \mathbf{x}2)$$

HOL Constant

StarStar : *TYPE*

StarStar = *mk_var_type* "∗∗"

HOL Constant

One_One : *TERM*

One_One = *mk_const*("*ONE_ONE*", *Fun*(*Fun Star StarStar*)*Bool*)

HOL Constant

One_One_def : *TERM*

One_One_def =
let f = *mk_var*("*f*",*Fun Star StarStar*)
in let x1 = *mk_var*("*x1*",*Star*)
in let x2 = *mk_var*("*x2*",*Star*) *in*
mk_abs(*f*, *mk_forall*(*x1*, *mk_forall*(*x2*,
mk_imp(*mk_eq*(*mk_comb*(*f*, *x1*), *mk_comb*(*f*, *x2*)),
mk_eq(*x1*, *x2*)))))

10.1.9 ONTO

The axiom of infinity requires the notion of an onto function. The constant *ONTO* is defined by the following equation:

$$\mathbf{ONTO} = \lambda(\mathbf{f} : * \rightarrow **)\bullet\forall(\mathbf{y} : **)\bullet\exists(\mathbf{x} : *)\bullet\mathbf{y} = \mathbf{f}\,\mathbf{x}$$

HOL Constant

ONTO : *TERM*

ONTO = *mk_const*("*ONTO*", *Fun*(*Fun Star StarStar*)*Bool*)

The name is all upper case to avoid conflict with the actual constant *Onto* used in the metalanguage system.

HOL Constant

ONTO_def : *TERM*

ONTO_def =
let f = *mk_var*("*f*",*Fun Star StarStar*)
in let x = *mk_var*("*x*",*Star*)
in let y = *mk_var*("*y*",*StarStar*) *in*
mk_abs(*f*, *mk_forall*(*y*, *mk_exists*(*x*, *mk_eq*(*y*, *mk_comb*(*f*, *x*)))))

10.1.10 Type_Definition

Type_Definition may be new to some readers. It is a term asserting that a function represents one type as a subtype of another. It is used in defining *new_type_definition*. It has type $(** \rightarrow bool) \rightarrow (* \rightarrow **) \rightarrow bool$ and is defined by the following equation:

Type_Definition = λ(*P*:∗∗→*bool*)•(*rep*:∗→∗∗)•*ONE_ONE rep*
∧ ∀(*x*:∗∗)•*P x* = ∃(*y*:∗)•*x* = *rep y*

It is useful later to have a version of *Type_Definition* parameterised over the types involved.

HOL Constant

Type_Definition : *TYPE* → *TYPE* → *TERM*

∀ *ty1 ty2* •
Type_Definition ty1 ty2 =
mk_const("*Type_Definition*", (*Fun* (*Fun ty2 Bool*) (*Fun*(*Fun ty1 ty2*)*Bool*)))

HOL Constant

Type_Definition_def : *TERM*

```
Type_Definition_def =
let P = mk_var("P",Fun StarStar Bool)
in let rep = mk_var("rep",Fun Star StarStar)
in let x = mk_var("x",StarStar)
in let y = mk_var("y",Star) in
mk_abs(P, mk_abs(rep,
        mk_conj(mk_comb(One_One, rep),
                mk_forall(x, mk_eq(mk_comb(P, x), mk_exists(y,
                        mk_eq(x, mk_comb(rep, y))))))))
```

10.2 *new_type* and *new_constant*

The first two definitional extension mechanisms, *new_type* and *new_constant* are conservative, but not very powerful.

new_type is used to declare a name to be used as a type constructor. No axioms about the type are introduced so that only instances of polymorphic functions may be applied to it. The only constraint is that the name should not be a type constructor in the theory to be extended.

To see, syntactically, that *new_type* is conservative observe that, given a proof in which the new type does not appear in the conclusion, distinct applications of the new type operator could be replaced by distinct type variables not used elsewhere in the proof. The result would be a proof in the unextended theory with the same conclusion as the original proof.

HOL Constant

new_type : $\mathbb{N} \rightarrow STRING \rightarrow THEORY \rightarrow THEORY \rightarrow BOOL$

$\forall$ *arity name thy1 thy2* $\bullet$
new_type arity name thy1 thy2 $\Leftrightarrow$
$\neg$ *name* $\in$ *Dom*(*types thy1*) $\wedge$
types thy2 = *types thy1* $\cup$ {(*name*, *arity*)} $\wedge$
constants thy2 = *constants thy1* $\wedge$
axioms thy2 = *axioms thy1*

new_constant is used to declare a name to be used as a constant of a given type. No axioms about the constant are introduced so that it behaves as a value which we cannot determine. The only constraint is that the name should not be a constant in the theory to be extended and that the type of the constant should be well-formed.

HOL Constant

new_constant : $STRING \rightarrow TYPE \rightarrow THEORY \rightarrow THEORY \rightarrow BOOL$

$\forall$ *name type thy1 thy2* $\bullet$
new_constant name type thy1 thy2 $\Leftrightarrow$
$\neg$ *name* $\in$ *Dom*(*constants thy1*) $\wedge$

$type \in wf\_type\ (types\ thy1) \wedge$
$constants\ thy2\ = constants\ thy1 \cup \{(name,\ type)\} \wedge$
$types\ thy2 = types\ thy1 \wedge$
$axioms\ thy2 = axioms\ thy1$

Again it is easy to see syntactically that this is conservative. Simply replace distinct instances of the new constant in a proof by distinct variables not used elsewhere in the proof to obtain a proof in the unextended theory.

10.3 *new_axiom*

new_axiom is both powerful and dangerous! It allows a sequent with no hypotheses and a given conclusion to be taken as an axiom. The only constraint is that the sequent be well-formed with respect to the environments of the theory being extended.

It is convenient, for technical reasons, in [2] to have the more general operation of adding a set of new axioms. We therefore define *new_axiom* in terms of the more general *new_axioms*.

HOL Constant

new_axioms : $(TERM\ SET) \rightarrow THEORY \rightarrow THEORY \rightarrow BOOL$

$\forall\ tms\ thy1\ thy2 \bullet$
$new\_axioms\ tms\ thy1\ thy2 =$
$let\ seqs = \{(x,\ tm) \mid x = \{\} \wedge tm \in tms\}$
in
$seqs \subseteq sequents\ thy1 \wedge$
$types\ thy2 = types\ thy1 \wedge$
$constants\ thy2 = constants\ thy1 \wedge$
$axioms\ thy2 = axioms\ thy1 \cup seqs$

HOL Constant

new_axiom : $TERM \rightarrow THEORY \rightarrow THEORY \rightarrow BOOL$

$\forall\ tm\ thy1\ thy2 \bullet$
$new\_axiom\ tm\ thy1\ thy2 = new\_axioms\ \{tm\}\ thy1\ thy2$

10.4 *new_definition*

new_definition is useful and conservative. It allows the simultaneous introduction of a new constant and an axiom asserting that the new constant is equal to a given term. The constraints imposed are *(a)* the name must satisfy the check made in *new_constant*, *(b)* the term must be closed and *(c)* the term must contain no bound variables whose types contain type variables which do not appear in the type of the new constant. Condition *(c)* ensures that different type instances of the term result in different instances of the constant; this avoids a possible inconsistency (see [2] for an example which arises in the course of this specification).

HOL Constant

new_definition : $STRING \to TERM \to THEORY \to THEORY \to BOOL$

$\forall$ *name tm thy1 thy2* •
new_definition name tm thy1 thy2 $\Leftrightarrow$
let ty = *type_of_term tm*
in
$\exists$ *thy1a* •
new_constant name ty thy1 thy1a $\wedge$
freevars_set tm = {} $\wedge$
term_tyvars tm $\subseteq$ *type_tyvars ty* $\wedge$
new_axiom (*mk_eq*(*mk_const*(*name*, *ty*), *tm*)) *thy1a thy2*

10.5 *new_specification*

new_specification allows the simultaneous introduction of a set of new constants satisfying a given predicate provided that a theorem asserting the existence of some set of values satisfying the constants is given. An axiom asserting the predicate for the new constants is introduced. Like *new_definition*, *new_specification* is useful and conservative.

The constraints imposed are analogous to those imposed in *new_definition*: *(a)* the constant names must be pairwise distinct and different from any constant name in the theory being extended, *(b)* the predicate must have no free variables apart from those corresponding to the new constants, *(c)* any type variable contained in a bound variable of the predicate must appear as a type variable of each of the new constants. Also, of course, the theorem must have the right form.

Since we now need to work with existential quantifiers it is necessary to introduce the theory *LOG*. We impose the restriction that *new_specification* may only be used to extend theories which extend *LOG*.

HOL Constant

LOG : *THEORY*

$\exists$ *thy1 thy2 thy3 thy4 thy5 thy6 thy7 thy8 thy9*•
let Name = λ*con*•ϵ*s*•$\exists$*ty*•*mk_const*(*s*, *ty*) = *con*
in
(*new_definition* (*Name Truth*) *Truth_def MIN thy1*
$\wedge$ *new_definition* (*Name* (*Forall Star*)) *Forall_def thy1 thy2*
$\wedge$ *new_definition* (*Name* (*Exists Star*)) *Exists_def thy2 thy3*
$\wedge$ *new_definition* (*Name Falsity*) *Falsity_def thy3 thy4*
$\wedge$ *new_definition* (*Name Negation*) *Negation_def thy4 thy5*
$\wedge$ *new_definition* (*Name Conjunction*) *Conjunction_def thy5 thy6*
$\wedge$ *new_definition* (*Name Disjunction*) *Disjunction_def thy6 thy7*
$\wedge$ *new_definition* (*Name One_One*) *One_One_def thy7 thy8*
$\wedge$ *new_definition* (*Name ONTO*) *ONTO_def thy8 thy9*
$\wedge$ *new_definition* (*Name* (*Type_Definition Star StarStar*)) *Type_Definition_def thy9 LOG*)

To define *new_specification* we need the relation *has_list_mk_exists*, and the relation *new_constants* which is like *new_constant* but handles a set of new constants.

HOL Constant

$$\textbf{has\_list\_mk\_exists} : (TERM\ LIST) \to TERM \to TERM \to BOOL$$

$$(\forall tm1\ tm2 \bullet has\_list\_mk\_exists\ []\ tm1\ tm2 \Leftrightarrow tm1 = tm2)$$
$$\land$$
$$(\forall\ v\ rest\ tm1\ tm2\ \bullet$$
$$has\_list\_mk\_exists\ (Cons\ v\ rest)\ tm1\ tm2 \Leftrightarrow$$
$$\exists\ rem\ \bullet\ has\_mk\_exists(v,\ rem)\ tm2\ \land$$
$$\quad has\_list\_mk\_exists\ rest\ rem\ tm1)$$

HOL Constant

$$\textbf{new\_constants} : ((STRING \times TYPE)\ SET) \to THEORY \to THEORY \to BOOL$$

$$\forall\ cons\ thy1\ thy2\ \bullet$$
$$new\_constants\ cons\ thy1\ thy2 \Leftrightarrow$$
$$Dom\ cons \cap Dom\ (constants\ thy1) = \{\}\ \land$$
$$Ran\ cons \subseteq wf\_type(types\ thy1)\ \land$$
$$constants\ thy2 = constants\ thy1 \cup cons\ \land$$
$$types\ thy2 = types\ thy1\ \land$$
$$axioms\ thy2 = axioms\ thy1$$

We can now define *new_specification*.

HOL Constant

$$\textbf{new\_specification} : ((STRING \times (STRING \times TYPE))\ LIST) \to$$
$$TERM \to THM \to THEORY \to THEORY \to BOOL$$

$$\forall\ pairs\ tm\ thm\ thy1\ thy2\ \bullet$$
$$new\_specification\ pairs\ tm\ thm\ thy1\ thy2 =$$
$$let\ conl = Fst(Split\ pairs)$$
$$in\ let\ varl = Map\ mk\_var\ (Snd(Split\ pairs))$$
$$in\ let\ tyl = Map\ Snd\ (Snd(Split\ pairs))$$
$$in\ let\ subs = \lambda(s,\ ty)\ \bullet$$
$$\quad if \quad \exists\ c\ \bullet\ (c,\ (s,\ ty)) \in Elems\ pairs$$
$$\quad then \quad mk\_const((\epsilon c \bullet (c,\ (s,\ ty)) \in Elems\ pairs),\ ty)$$
$$\quad else \quad mk\_var(s,\ ty)$$
$$in\ let\ axiom = subst\ subs\ tm$$
$$in\ (\exists\ conc \bullet$$
$$has\_list\_mk\_exists\ varl\ tm\ conc$$
$$\land\ thy1\ extends\ LOG$$
$$\land\ (freevars\_set\ conc = \{\})$$
$$\land\ conl \in Distinct$$

$\wedge$ *varl* $\in$ *Distinct*
$\wedge$ *thm_seq thm* = ({}, *conc*)
$\wedge$ *thy1 extends thm_thy thm*
$\wedge$ ($\forall$ *ty*• *ty* $\in$ *Elems tyl* $\Rightarrow$ *term_tyvars conc* $\subseteq$ *type_tyvars ty*)
$\wedge$ ($\exists$ *thy1a* •
new_constants (*Elems* (*Combine conl tyl*)) *thy1 thy1a* $\wedge$
new_axiom axiom thy1a thy2))

10.6 *new_type_definition*

new_type_definition allows the introduction of a new type in one-to-one correspondence with the subset of an existing type satisfying a given predicate, given a theorem asserting that the subset is not empty. A new axiom asserting the existence of a representation function for the new type is introduced. Like *new_definition*, *new_type_definition* is useful and conservative.

For simplicity, we have made the list of type variable names to be used as the parameters of the type being defined, a parameter to *new_type*. The constraints imposed are *(a)* that the list of type parameter names contain no repeats, *(b)* the theorem must have the right form and *(c)* all type variables contained in the predicate must be contained in the list of type parameters names. Condition *(c)* ensures that different type instances of the new axiom involve different type instances of the new type.

HOL Constant

new_type_definition :
STRING $\rightarrow$ (*STRING LIST*) $\rightarrow$ *THM* $\rightarrow$ *THEORY* $\rightarrow$ *THEORY* $\rightarrow$ *BOOL*

$\forall$ *name typars thm thy1 thy2* •
new_type_definition name typars thm thy1 thy2 $\Leftrightarrow$
$\exists$ *p xty x ty px thy1a axiom* •
let newty = *mk_type*(*name*, *Map mk_var_type typars*)
in let f = *mk_var*("*f*", *Fun newty ty*)
in *thy1 extends LOG*
$\wedge$ *hyp* (*thm_seq thm*) = {}
$\wedge$ *has_mk_exists* (*xty*, *px*) (*concl* (*thm_seq thm*))
$\wedge$ *mk_var* (*x*, *ty*) = *xty*
$\wedge$ *has_mk_comb* (*p*, *xty*) *px*
$\wedge$ *freevars_set p* = {}
$\wedge$ *term_tyvars p* $\subseteq$ *Elems typars*
$\wedge$ *typars* $\in$ *Distinct*
$\wedge$ *has_mk_exists*(*f*, *mk_comb*(*mk_comb*(*Type_Definition newty ty*, *p*), *f*)) *axiom*
$\wedge$ *new_type* (# *typars*) *name thy1 thy1a*
$\wedge$ *new_axiom axiom thy1a thy2*

11 THE THEORY INIT

By extending the theory *LOG* with five axioms we will arrive at the theory *INIT*. In a typical HOL proof development system all theories will be extensions of this theory.

11.1 The Axioms

11.1.1 BOOL_CASES_AX

This is the law of the excluded middle:

$BOOL\_CASES\backslash\_AX \vdash \forall(b{:}bool)\bullet(b = T) \lor (b = F)$

HOL Constant

BOOL_CASES_AX : *TERM*

BOOL_CASES_AX =
let b = mk_var("b", Bool)
in mk_forall(b, mk_disj(mk_eq(b, Truth), mk_eq(b, Falsity)))

11.1.2 IMP_ANTISYM_AX

This says that implication is an antisymmetric relation:

$IMP\_ANTISYM\_AX \vdash \forall(b1{:}bool)\bullet\forall(b2{:}bool)\bullet(b1 \Rightarrow b2) \Rightarrow (b2 \Rightarrow b1) \Rightarrow (b1{=}b2)$

HOL Constant

IMP_ANTISYM_AX : *TERM*

IMP_ANTISYM_AX =
let b1 = mk_var("b1", Bool)
in let b2 = mk_var("b2", Bool)
in mk_forall(b1, mk_forall(b2,
mk_imp(mk_imp(mk_imp(b1, b2), mk_imp(b2, b1)), mk_eq(b1, b2))))

11.1.3 ETA_AX

This says that an η-redex is equal to its η-reduction.

$ETA\_AX \vdash \forall(f{:}* \to **)\bullet(\lambda(x{:}*)\bullet f\ x) = f$

HOL Constant

ETA_AX : *TERM*

ETA_AX =
let f = mk_var("f1", Fun Star StarStar)
in let x = mk_var("x", Star)
in mk_forall(f, mk_eq(mk_abs(x, mk_comb(f, x)), f))

11.1.4 SELECT_AX

This is the defining property of the choice function ϵ.

$SELECT\_AX \vdash \forall(P{:}*\to bool)\bullet\forall(x{:}*)\bullet P\ x \;\Rightarrow\; P(\epsilon\ P)$

HOL Constant

SELECT_AX : *TERM*

SELECT_AX =
let P = *mk_var*("*P*", *Fun Star Bool*)
in let x = *mk_var*("*x*", *Star*)
in mk_forall(*P*,*mk_forall*(*x*,
mk_imp(*mk_comb*(*P*, *x*), *mk_comb*(*P*, *mk_comb*(*Choice*, *P*)))))

11.1.5 INFINITY_AX

This is the axiom of infinity. It asserts that the type *ind* is in one-to-one correspondence with a proper subset of itself:

$INFINITY\_AX \vdash \exists(f{:}ind\to ind)\bullet ONE\_ONE\ f \;\wedge\; \neg ONTO\ f$

HOL Constant

INFINITY_AX : *TERM*

INFINITY_AX =
let f = *mk_var*("*f*", *Fun Ind Ind*)
in mk_conj(*mk_comb*(*One_One*, *f*), *mk_comb*(*Negation*, *mk_comb*(*ONTO*, *f*)))

11.2 The Theory

HOL Constant

INIT : *THEORY*

∃ *thy1 thy2 thy3 thy4 thy5 thy6* •
new_axiom BOOL_CASES_AX LOG thy1
∧ *new_axiom IMP_ANTISYM_AX thy1 thy2*
∧ *new_axiom ETA_AX thy2 thy3*
∧ *new_axiom SELECT_AX thy4 thy5*
∧ *new_type 0* (*Fst*(*dest_type Ind*)) *thy5 thy6*
∧ *new_axiom INFINITY_AX thy6 INIT*

11.3 DEFINITIONAL EXTENSIONS

We will say that a theory *thy1* is a *definitional* extension of a theory *thy2* if one may go from *thy2* to *thy1* by some sequence of extensions by the functions *new_type*, *new_constant*, *new_definition*, *new_specification* and *new_type_definition*. It is stressed that definitional extensions in this sense comprise significantly more than just extension by adjoining a defining equation for a new constant.

HOL Constant

$$\textbf{definitional\_extension} : \mathit{THEORY} \to \mathit{THEORY\ SET}$$

$$\begin{array}{l}
\forall \mathit{thy} \bullet \mathit{definitional_extension}\ \mathit{thy} = \bigcap \{ \mathit{thyset}\ | \\
\qquad \mathit{thy} \in \mathit{thyset} \\
\wedge\ (\quad \forall \mathit{thy1}\ \mathit{thy2}\ \mathit{arity}\ \mathit{name} \bullet \\
\qquad \mathit{thy1} \in \mathit{thyset}\ \wedge \\
\qquad \mathit{new_type}\ \mathit{arity}\ \mathit{name}\ \mathit{thy1}\ \mathit{thy2} \Rightarrow \mathit{thy2} \in \mathit{thyset} \\
)\ \wedge\ (\\
\forall \mathit{thy1}\ \mathit{thy2}\ \mathit{name}\ \mathit{type} \bullet \\
\qquad \mathit{thy1} \in \mathit{thyset}\ \wedge \\
\qquad \mathit{new_constant}\ \mathit{name}\ \mathit{type}\ \mathit{thy1}\ \mathit{thy2} \Rightarrow \mathit{thy2} \in \mathit{thyset} \\
)\ \wedge\ (\\
\forall \mathit{thy1}\ \mathit{thy2}\ \mathit{name}\ \mathit{tm} \bullet \\
\qquad \mathit{thy1} \in \mathit{thyset}\ \wedge \\
\qquad \mathit{new_definition}\ \mathit{name}\ \mathit{tm}\ \mathit{thy1}\ \mathit{thy2} \Rightarrow \mathit{thy2} \in \mathit{thyset} \\
)\ \wedge\ (\\
\forall \mathit{thy1}\ \mathit{thy2}\ \mathit{pairs}\ \mathit{tm}\ \mathit{thm} \bullet \\
\qquad \mathit{thy1} \in \mathit{thyset}\ \wedge \\
\qquad \mathit{new_specification}\ \mathit{pairs}\ \mathit{tm}\ \mathit{thm}\ \mathit{thy1}\ \mathit{thy2} \Rightarrow \mathit{thy2} \in \mathit{thyset} \\
)\ \wedge\ (\\
\forall \mathit{thy1}\ \mathit{thy2}\ \mathit{name}\ \mathit{typars}\ \mathit{thm} \bullet \\
\qquad \mathit{thy1} \in \mathit{thyset}\ \wedge \\
\qquad \mathit{new_type_definition}\ \mathit{name}\ \mathit{typars}\ \mathit{thm}\ \mathit{thy1}\ \mathit{thy2} \Rightarrow \mathit{thy2} \in \mathit{thyset} \\
)\}
\end{array}$$

Of particular importance are theories which may be obtained from *INIT* by definitional extension. These theories are of interest since, we assert, they form a sound formalism in which much of the practical machine-checked proof work one might wish to do can be carried out.

12 INDEX OF DEFINED TERMS