Title: HOL in HOL

Ref: DS/FMU/RBJ/150 *Issue:* 0.1 *Date:* 1989-10-05

Status: Draft *Type:*

Author:

Name	*Location*	*Signature*	*Date*
Roger Bishop Jones			

Authorisation for Issue:

Name	*Function*	*Signature*	*Date*

Abstract:

Distribution:

0 Document control

0.1 Contents list

0.2 Document cross references

0.3 Changes history

First version.

0.4 Changes forecast

Under development, highly incomplete, totally volatile.

0.5 Abbreviations and notation

HOL Higher Order Logic

1 INTRODUCTION

1.1

1.2 Introducing the new theory

We first introduce a new theory with Zermelo Fraenkel (pf137) as a parent. Definitions theorems and axioms from various ancestors are loaded and bound to ML variable names.

SML

```
new_theory‘ppf150‘;;
map new_parent [‘string‘; ‘infra‘];;
map loadf [‘infra‘];;
lemmas_on := false;;
```

2 INTRODUCING THE NEW TYPE HOLSYN

2.1 The type definition

SML

```
new_type 0 ‘TYPE‘;;
new_constant (‘mk_vartype‘,":string → TYPE");;
new_constant (‘mk_type‘,":string × TYPE list → TYPE");;
let TYPE_axiom = new_axiom(‘TYPE_axiom‘,"
        ∀(f0:string → ∗) (f1: string × TYPE list → ∗ list → ∗)•
       ∃ (fn:TYPE → ∗)•
        (∀s• fn(mk_vartype s) = f0 s) ∧
        (∀s tl• fn(mk_type (s,tl)) = f1 (s, tl) (map fn tl))
");;
```

SML

```
new_type 0 ‘FTERM‘;;
new_constant (‘mk_fvar‘,":string × TYPE → FTERM");;
new_constant (‘mk_fconst‘,":string × TYPE → FTERM");;
new_constant (‘mk_fcomb‘,":FTERM × FTERM  → FTERM");;
new_constant (‘mk_fabs‘,":string × TYPE × FTERM  → FTERM");;

let FTERM_axiom = new_axiom(‘FTERM_axiom‘,"
        ∀
        (f0:string × TYPE → ∗)
        (f1:string × TYPE → ∗)
        (f2: FTERM × FTERM → ∗ × ∗ → ∗)
        (f3: string × TYPE × FTERM → ∗ → ∗)•
       ∃ (fn:FTERM → ∗)•
        (∀s ty• fn (mk_fvar (s, ty))
                        = f0 (s, ty))
        ∧  (∀s ty•      fn(mk_fconst (s, ty))
                        = f1 (s, ty))
        ∧  (∀f a•       fn(mk_fcomb (f, a))
                        = f2 (f, a) (fn f, fn a))
        ∧  (∀v ty b•    fn(mk_fabs (v, ty, b))
                        = f3 (v, ty, b) (fn b))
");;
```

HOL Constant

```
        dest_fvar: FTERM → string × TYPE
├──────────────────────────────────────────
                dest_fvar o mk_fvar = I
```

HOL Constant

$$dest\_fconst: FTERM \to string \times TYPE$$

$$dest\_fconst \ o \ mk\_fconst = I$$

HOL Constant

$$dest\_fcomb: FTERM \to FTERM \times FTERM$$

$$dest\_fcomb \ o \ mk\_fcomb = I$$

HOL Constant

$$dest\_fabs: FTERM \to string \times TYPE \times FTERM$$

$$dest\_fabs \ o \ mk\_fabs = I$$

3 WELL FORMED TERMS

HOL Constant

$$consistent\_vartypes: (string \times TYPE \to bool) \to bool$$

$$consistent\_vartypes \ sts = \forall(s,t)(s',t')\bullet \ sts \ (s,t) \wedge sts \ (s',t') \wedge (s = s') \Rightarrow (t = t')$$

HOL Constant

$$freevars\_fset: FTERM \to (string \times TYPE \to bool)$$

$$\forall(s : string) \ (t : TYPE) \ (f : FTERM) \ (a : FTERM) \ (v : string) \ (b : FTERM) \bullet$$
$$(freevars\_fset \ (mk\_fvar(s,t)) = \in x\bullet \ x = (s,t))$$
$$\wedge \quad (freevars\_fset \ (mk\_fconst(s,t)) = \in x\bullet \ F)$$
$$\wedge \quad (freevars\_fset \ (mk\_fcomb(f, \ a)) = (freevars\_fset \ f) \cap (freevars\_fset \ a))$$
$$\wedge \quad (freevars\_fset \ (mk\_fabs(v, \ t, \ b)) = (freevars\_fset \ b) \ \delta \in x\bullet \ x = (v,t))$$

HOL Constant

```
type_of_term: FTERM → TYPE → bool
```

```
∀(s : string) (t : TYPE) (ty: TYPE)
 (f : FTERM) (a : FTERM) (v : string)
 (b : FTERM) •
(type_of_term (mk_fvar(s,t)) ty = (t = ty))
∧ (type_of_term (mk_fconst(s,t)) ty = (t = ty))
∧ (type_of_term (mk_fcomb(f, a)) ty =
        ∃(t1 : TYPE)•
                (type_of_term f (mk_type(‘→‘, [t1; ty])))
        ∧       (type_of_term a t1)
        ∧       ((consistent_vartypes o freevars_fset) (mk_fcomb(f, a))))
∧ (type_of_term (mk_fabs(v, t, b)) ty =
        ∃(t1 : TYPE)•
                (type_of_term b t1)
        ∧       (mk_type(‘→‘, [t; t1]) = ty)
        ∧       (consistent_vartypes ((freevars_fset b) ∩ (∈x• x = (v,t)))))
```

For comparison, we give a constructive version of the above (omitting the checks for type consistency of free variables). The relation *type_of_term* is actually a partial function. We use the following gadgets to define partial functions.

SML

```
new_type_abbrev(‘FAIL‘, ":one");;
```

HOL Constant

```
FAIL : FAIL
```

```
FAIL = one
```

We need the *dest_type* function.

HOL Constant

```
dest_type : TYPE → (string × TYPE list) + FAIL
```

```
(dest_type(mk_vartype s) = INR FAIL)
∧ (dest_type(mk_type(s, tl)) = INL (s, tl))
```

HOL Constant

```
        ctype_of_term: FTERM → TYPE + FAIL
├──────────────────────────────────────────────────────
        ∀(s : string) (t : TYPE) (ty: TYPE)
         (f : FTERM) (a : FTERM) (v : FTERM)
         (b : FTERM) •
        (ctype_of_term (mk_fvar(s,t)) = INL t)
∧       (ctype_of_term (mk_fconst(s,t)) = INL t)
∧       (ctype_of_term (mk_fcomb(f, a)) =
                let tf = ctype_of_term f in
              let ta = ctype_of_term a in
                if ((ISL tf) ∧ (ISL ta))
                then (
                let s_tl = dest_type (OUTL tf) in
                        if ((ISL s_tl)
                        ∧ (FST (OUTL s_tl) = '→')
                        ∧ (length (SND (OUTL s_tl)) = 2)
                        ∧ (HD (SND (OUTL s_tl)) = (OUTL ta)))
                        then (INL (HD (TL (SND (OUTL s_tl)))))
                        else (INR FAIL))
                else (INR FAIL))
∧       (ctype_of_term (mk_abs(v, t, b)) =
                let tb =  (ctype_of_term b) in
                if (ISL tb)
                then (INL (mk_type('→', [t; OUTL tb])))
                else (INR FAIL))
```

HOL Constant

```
        is_wf_term : FTERM → bool
├──────────────────────────────────────────────────────
        ∀term:FTERM•
        is_wf_term term = ∃type:TYPE• type_of_term term type
```

SML

```
sim_type_def 'TERM' ("x:FTERM, is_wf_term x");;
let TERM_AXIOM = new_type_definition('TERM',
        "p_or_choice is_wf_term",
        TAC_PROOF(([],
        "∃x:FTERM• (p_or_choice is_wf_term) x"),
        REWRITE_TAC[EXISTS_p_or_choice]));;
let EQ_IMP thm = DISCH_ALL (EQ_MP thm (ASSUME ((lhs o concl) thm)));;
let new_type_lemmas type_axiom =
        let [a1;a2;a3;a4;a5;a6] = prove_new_type_lemmas type_axiom
in      LIST_CONJ [a1;a5;a3; GEN_ALL (EQ_IMP (SPEC_ALL a6));a2;a4];;
let TERM_LEMMAS = save_thm('TERM_LEMMAS',new_type_lemmas TERM_AXIOM);;
```

HOL Constant

$$freevars\_set : TERM \to (string \times TYPE \to bool)$$

$$freevars\_set = freevars\_fset \ o \ REP\_TERM$$

HOL Constant

$$mk\_var : string \times TYPE \to TERM$$

$$mk\_var = ABS\_TERM \ o \ mk\_fvar$$

HOL Constant

$$mk\_const : string \times TYPE \to TERM$$

$$mk\_const = ABS\_TERM \ o \ mk\_fconst$$

HOL Constant

$$mk\_comb : TERM \times TERM \to TERM$$

$$mk\_comb = (ABS\_TERM \ o \ mk\_fcomb) \ o \in(x,y)\bullet \ (REP\_TERM \ x, \ REP\_TERM \ y)$$

HOL Constant

$$dest\_var: TERM \to string \times TYPE$$

$$dest\_var \ o \ mk\_var = I$$

HOL Constant

$$mk\_abs : TERM \times TERM \to TERM$$

$$mk\_abs = (ABS\_TERM \ o \ mk\_fabs) \ o \in(x,y)\bullet \ (\ FST(dest\_var \ x), \ SND(dest\_var \ x), \ REP\_TERM \ y)$$

HOL Constant

$$dest\_const: TERM \to string \times TYPE$$

$$dest\_const \ o \ mk\_const = I$$

HOL Constant

$$dest\_comb: TERM \to TERM \times TERM$$

$$dest\_comb \ o \ mk\_comb = I$$

HOL Constant

$$dest\_abs: TERM \to TERM \times TERM$$

$$dest\_abs \ o \ mk\_abs = I$$

4 FREE VARIABLES

HOL Constant

$type\_tyvars : TYPE \to (string \to bool)$

$(\forall s{:}string\bullet\ type\_tyvars\ (mk\_vartype\ s) = \in x\bullet x{=}s)$

$\wedge\quad (\forall s\ tl\bullet\ type\_tyvars\ (mk\_type(s,\ tl)) =$
$\ominus \in x\bullet\ x \in (map\ term\_tyvars\ tl))$

HOL Constant

$term\_types : TERM \to (TYPE \to bool)$

$(\forall s\ t\bullet\ term\_types\ (mk\_var\ (s,t)) = \in x\bullet x{=}t)$

$\wedge\quad (\forall s\ t\bullet\ term\_types\ (mk\_const\ (s,t)) = \in x\bullet x{=}t)$

$\wedge\quad (\forall f\ a\bullet\ term\_types\ (mk\_comb\ (f,a)) =$
$(term\_types\ f) \cap (term\_types\ a))$

$\wedge\quad (\forall v\ b\bullet\ term\_types\ (mk\_abs(v,b)) =$
$(term\_types\ v) \cap (term\_types\ b))$

HOL Constant

$image : (* \to **) \to (* \to bool) \to (** \to bool)$

$\forall function\ set\bullet\ image\ function\ set =$
$\in x{:}**\bullet\ \exists y{:}*\bullet\ (x = f\ y) \wedge y \in set$

HOL Constant

$types\_tyvars : (TYPE \to bool) \to (string \to bool)$

$\forall types\bullet\ types\_tyvars\ types =$
$\ominus ((image{:}\ (TYPE \to (string \to bool)) \to (TYPE \to bool) \to ((string \to bool) \to bool))$
$type\_tvars\ types)$

HOL Constant

$term\_tyvars : TERM \to (string \to bool)$

$term\_tyvars = types\_tyvars\ o\ term\_types$

HOL Constant

$terms\_tyvars : (TERM \to bool) \to (string \to bool)$

$\forall\ terms{:}(TERM \to bool)\bullet$
$terms\_tyvars\ terms =$
$\ominus ((image{:}(TERM \to (string \to bool)) \to (TERM \to bool) \to ((string \to bool) \to bool))$
$term\_tvars\ terms)$

HOL Constant

$$l\_t\_s : (* \; list) \to (* \to bool)$$

$$l\_t\_s \; s \; e = e \in s$$

5 TYPE INSTANTIATION

5.1 Of Types

HOL Constant

$$inst\_typef : (string \to TYPE) \to TYPE \to TYPE$$

$$\begin{array}{ll} & \forall (f{:}\ string \to TYPE) \bullet \\ & \quad (\forall s \bullet \; inst\_typef \; f \; (mk\_vartype \; s) = f \; s) \\ \wedge & \quad (\forall s \; tl \bullet inst\_typef \; f \; (mk\_type(s, \; tl)) = \\ & \qquad mk\_type(s, \; map \; (inst\_typef \; f) \; tl)) \end{array}$$

5.2 Of Terms

The problem here is to rename variables in such a way as to prevent identification of previously distinct variables and prevent seperation of previously identical ones. .LP This is definitely not trivial, and I havn't yet figured out how to do it.

HOL Constant

$$ran : (* \times ** \to bool) \to (** \to bool)$$

$$ran \; f \; e = \exists x \bullet f \; (x, \; e)$$

HOL Constant

$$\begin{array}{ll} & inst\_termf : \\ & \quad (string \to TYPE) \\ & \quad (* \; instantiations \; required \; *) \\ \to & \quad (\quad (string \times TYPE \to bool) \\ & \quad (* \; free \; variables \; to \; avoid \; clashing \; with \; *) \\ & \quad \times \quad ((string \times TYPE) \times (string \times TYPE) \to bool) \\ & \quad (* \; substitutions \; already \; done \; and \; still \; relevant \; *) \\ & \quad \times \quad TERM) \\ \to & \quad (\quad (string \times TYPE \to bool) \\ & \quad \times \quad ((string \times TYPE) \times (string \times TYPE) \to bool) \\ & \quad \times \quad TERM) \end{array}$$

$$\begin{array}{ll} \forall & (tsubs : string \to TYPE) \\ & (fvars : string \times TYPE \to bool) \\ & (vsubs : (string \times TYPE) \times (string \times TYPE) \to bool) \bullet \\ & (\forall s \; t \bullet \\ & \quad inst\_termf \; tsubs \; (fvars, \; vsubs, \; (mk\_var \; (s,t))) \end{array}$$

```
|                   =
|                   (∃v'• vsubs ((s,t), v')) =>
|                           (fvars,
|                           vsubs,
|                           mk_var v')
|                   | (inst_typef tsubs t = t) =>
|                           ((s,t) ∈ (ran vsubs) =>
|                             (let    nn = s (* rename it *)
|                             in              fvars,
|                                             (vsubs ∩ (l_t_s [(s,t),(nn,t)])),
|                                             mk_var(nn,t))
|                           | fvars ∩ (l_t_s [s,t]),
|                             vsubs,
|                             mk_var(s,t))
|                   |       let nn =
|                                   ((s, inst_typef tsubs t) ∈ (fvars ∩ (ran vsubs))
|                                   => s (* change to new name for variable *)
|                                   | s)
|                           in      (fvars,
|                                   vsubs ∩ (l_t_s[(s,t), (nn,inst_typef tsubs t)]),
|                                   mk_var(nn,inst_typef tsubs t)
|                                   )
|           )
| ∧         (∀s t•
|                   inst_termf tsubs (fvars, vsubs, (mk_const (s,t)))
|                   =       (fvars,
|                           vsubs,
|                           mk_const(s, inst_typef tsubs t)))
| ∧         (∀f a•
|                   inst_termf tsubs (fvars, vsubs, (mk_comb (f,a)))
|                   =       let lr = inst_termf tsubs (fvars, vsubs, f)
|                           in
|                           let rr = inst_termf tsubs (FST lr, FST(SND lr), a)
|                           in
|                           (FST rr,
|                           FST(SND rr),
|                           mk_comb(SND(SND lr),SND(SND rr))))
| ∧         (∀v b•
|                   inst_termf tsubs (fvars, vsubs, (mk_abs (v,b)))
|                   =       let lr = inst_termf tsubs (fvars, vsubs, v)
|                           in
|                           let rr = inst_termf tsubs (FST lr, FST(SND lr), v)
|                           in
|                           (FST rr δ ((FST lr) δ fvars),
|                           FST(SND rr),
|                           mk_abs(SND(SND lr),SND(SND rr))))
```

6 SUBSTITUTION

We will give a rather unconstructive formulation of the notion of substitution. First we need to define renaming: *rename (v, ty) w e* is the result of changing all instances of the variable with name v and type ty in the term e to have name w.

HOL Constant

$$rename : (string \times TYPE) \to string \to FTERM \to FTERM$$

$$\begin{array}{l}
\forall \\
(v : string)\ (ty : TYPE)\ (w\text{: } string) \\
(vv : string)\ (tty : TYPE)\ (cc : string) \\
(ff : FTERM)\ (aa : FTERM)\ (bb : FTERM) \\
\bullet \\
(rename\ (v,\ ty)\ w\ (mk_fvar(vv,\ tty)) = \\
\quad ((v = vv) \wedge (ty = tty)) => \\
\quad\quad mk_fvar(w,\ ty) \mid mk_fvar(vv,\ tty)) \\
\wedge \\
(rename\ (v,\ ty)\ w\ (mk_fconst(cc,\ tty)) = \\
\quad mk_fconst(cc,\ tty)) \\
\wedge \\
(rename\ (v,\ ty)\ w\ (mk_fcomb(ff,\ aa)) = \\
\quad mk_fcomb(rename\ (v,\ ty)\ w\ ff,\ rename\ (v,\ ty)\ w\ aa)) \\
\wedge \\
(rename\ (v,\ ty)\ w\ (mk_fabs(vv,\ tty,\ bb)) = \\
\quad ((v = vv) \wedge (ty = tty)) => \\
\quad\quad mk_fabs(vv,\ tty,\ bb) \mid mk_fabs(vv,\ tty,\ rename\ (v,\ ty)\ w\ bb))
\end{array}$$

We will need to test when two terms are α-convertible:

HOL Constant

$$aconv : FTERM \to FTERM \to bool$$

$$\begin{array}{lll}
\forall(t1 : FTERM)\ (t2 : FTERM)\bullet & & \\
aconv\ t1\ t2 = & & \\
 & (t1 = t2) & \\
\vee & (\exists t1f\ t1a\ t2f\ t2a\bullet & \\
 & \quad (t1 = mk_fcomb(t1f,\ t1a)) & \\
 & \wedge \quad (t2 = mk_fcomb(t2f,\ t2a)) & \\
 & \wedge \quad (aconv\ t1f\ t2f \wedge aconv\ t1a\ t2a)) & \\
\vee & (\exists v1\ v2\ ty\ b1\ b2\bullet & \\
 & \quad (t1 = mk_fabs(v1,\ ty,\ b1)) & \\
 & \wedge \quad (t2 = mk_fabs(v2,\ ty,\ b2)) & \\
 & \wedge \quad aconv\ t1\ (mk_fabs(v1,\ ty,\ rename\ (v2,\ ty)\ v1\ b2))) &
\end{array}$$

We will need to choose new variable names:

HOL Constant

```
new_var : TYPE → (string × TYPE → bool) → string
```

```
∀ ty vs • (new_var ty vs, ty) Σ vs
```

Now we can define *subst*. *subst F t1* gives the term resulting from replacing every free variable in *t1* by its "image under *F*" with bound variables renamed as necessary to avoid capture. Variables which are not to be changed correspond to pairs (s, t) with $F(s, t) = mk\_fvar(s, t)$.

HOL Constant

```
subst : (string × TYPE → FTERM) → FTERM → FTERM
```

```
∀
(v : string) (ty : TYPE) (c : string)
(f : FTERM) (a : FTERM) (b : FTERM)
•
(subst R (mk_fvar(v, ty)) = R (v,ty))
∧
(subst R (mk_fconst(c, ty)) = mk_fconst(c, ty))
∧
(subst R (mk_fcomb(f, a)) =
        mk_fcomb(subst R f, subst R a))
∧
(subst R (mk_fabs(v, ty, b)) =
        let RR = ∈x• (x = (v, ty)) => (mk_fvar x) | R x
        in      if( ∃w•
                        (w ■ (v, ty))
                ∧       ((v, ty) ∈ freevars_fset (RR w))
                ∧       (w ∈ freevars_fset b)
                )then(  let new_frees =
∈(image (freevars_fset o RR) (freevars_fset b))
                        in let v' = new_var ty new_frees
                        in let RRR = ∈x• (x = (v', ty)) => (mk_fvar x) | RR x
                        in
mk_fabs(v', ty, subst RRR (rename (v, ty) v' b))
                )else(
mk_fabs(v, ty, subst RR b)))
```

The inference rule `SUBST` may readily be defined in terms of the above function.

7 SEQUENTS

SML

```
new_type_abbrev(‘SEQ‘, ":(TERM) list × TERM");;
```

8 PROOFS

SML

```
new_type 0 `FPROOF`;;
new_constant (`mk_fAXp`,":SEQ → FPROOF");;
new_constant (`mk_fASSUMEp`,":TERM → FPROOF");;
new_constant (`mk_fREFLp`,":TERM → FPROOF");;
new_constant (`mk_fBETA_CONVp`,":TERM → FPROOF");;
new_constant (`mk_fSUBSTp`,
        ":(FPROOF × TERM)list → TERM → FPROOF → FPROOF");;
new_constant (`mk_fABSp`,":TERM → FPROOF → FPROOF");;
new_constant (`mk_fINST_TYPEp`,
        ":(TYPE × TYPE)list → FPROOF → FPROOF");;
new_constant (`mk_fDISCHp`,":TERM → FPROOF → FPROOF");;
new_constant (`mk_fMPp`,":FPROOF → FPROOF → FPROOF");;

let FPROOF_axiom = new_axiom(`FPROOF_axiom`,"
        ∀
        (fAXf:SEQ → ∗)
        (fASSUMEf:TERM → ∗)
        (fREFLf:TERM → ∗)
        (fBETA_CONVf:TERM → ∗)
        (fSUBSTf:(FPROOF × TERM)list → TERM → FPROOF → (∗)list → ∗ → ∗)
        (fABSf:TERM → FPROOF → ∗ → ∗)
        (fINST_TYPEf:(TYPE × TYPE)list → FPROOF → ∗ → ∗)
        (fDISCHf:TERM → FPROOF → ∗ → ∗)
        (fMPf:FPROOF → FPROOF → ∗ → ∗ → ∗)•
       ∃ (fn:FPROOF → ∗)•
        (∀seq•  fn (mk_fAXp seq)
                        = fAXf s)
        ∧  (∀term•      fn(mk_fASSUMEp term)
                        = fASSUMEf term)
        ∧  (∀term•      fn(mk_fREFLp term)
                        = fREFLf term)
        ∧  (∀term•      fn(mk_fBETA_CONVp term)
                        = fBETA_CONVf term)
        ∧  (∀fptl term fproof•
                fn(mk_fSUBSTp fptl term fproof)
                        = fSUBSTf fptl term fproof
                                (map (fn o FST) fptl) (fn fproof))
        ∧  (∀term fproof•
                fn(mk_fABSp term fproof)
                        = fABSf term fproof (fn fproof))
        ∧  (∀ttl fproof•
                fn(mk_fINST_TYPEp ttl fproof)
                        = fINST_TYPEf ttl fproof (fn fproof))
        ∧  (∀term fproof•
```

```
              fn(mk_fDISCHp term fproof)
                      = fDISCHf term fproof (fn fproof))
       ∧ (∀fproof1 fproof2•
              fn(mk_fMPp fproof1 fproof2)
                      = fMPf fproof1 fproof2 (fn fproof1) (fn fproof2))
");;
```

9 THEORIES

We have already defined the type environment part of a theory. A theory also has both a constant environment and a bunch of axioms:

SML

```
new_type_abbrev(‘TY_ENV‘, ":string → num → bool");;

new_type_abbrev(‘CON_ENV‘, ":string → TYPE → bool");;

new_type_abbrev(‘SEQS‘, ":SEQ → bool");;
```

Z

```
‘THEORY“T‘
name       :string,
parents    :string → bool,
ty_env     :TY_ENV,
con_env    :CON_ENV,
extensions :SEQS,
axioms     :SEQS,
theorems   :SEQS
----------------------------------
T
```

SML

```
new_type_abbrev(‘FTHEORIES‘, ":THEORY list");;
```

10 THE THEORY

11 SEMANTIC DOMAINS

1. Mono-types are sets (objects of type SET).
2. A type variable assignment is a map from strings to Mono-types (SETS).
3. A type constant assignment is a map from (string X SET list) to SET.
4. Monomorphic individuals are elements of SET.
5. Polmorphic values are maps from type variable assignments to Monomorphic values.
6. A variable assignment is a map from string to polymorphic value.
7. A constant assignment is a map from string to polymorphic value.

12 ISSUES

12.1 Changes to the Language

I am strongly tempted to make some adjustments to the language. The strongest temptation I have is to outlaw overloading of variable names. Some but not all of the above specs are written on that assumption. I am also tempted to have term formation rules which are independent of the theory environment (i.e. allow constants to be used before declaration). Also to allow overloading of constants. (rules for conservative extension permitting sets of constant definitions provided the types used are pairwise non-unifiable).

12.2 Formalisation

I would like to avoid having FAILure results on all functions which are not certain to succeed, but am not sure whether they can be avoided. The reason is to keep the specification simple. Where a function is normally used in circumstances when its arguments are known to be satisfactory, I think it is better to use a version which does not have failure codes in it. To make this possible as consistently as possible I think functions with failure results should be used only where absolutely necessary. This corresponds to programming in ML by checking with "is_?" before doing "dest_?" instead of going ahead regardless and then trapping the exception. Trapping exceptions is easier in ML but more complicated in HOL.