Technical Report

UCAM-CL-TR-323
ISSN 1476-2986

Number 323

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Representing higher-order logic proofs in HOL

J. von Wright

January 1994

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

ISSN 1476-2986

Representing higher-order logic proofs in HOL

J. von Wright*

January 18, 1994

1 Introduction

When using a theorem prover based on classical logic, such as HOL [2], we are generally interested in the facts that are proved (the theorems) than in the way in which they were proved (the proofs). However, we may not trust the prover completely, and so be interested in checking the correctness of the proof. Since machine-generated proofs are generally very long, checking by hand is out of the question; we need a computer program, a *proof checker*. However, we also need to trust the proof checker. Preferrably, we would want the correctness of proof checker to be verified formally. One way of doing this is by specifying it in a mechanised logic (such as that of the HOL system) and then doing a correctness proof in this logic. This process may seem circular, but it is acceptable, provided that we have a theory of proofs embedded in the logic.

This paper describes an attempt to formalise the notion of HOL proofs within HOL. The aim is to be able to verify (inside HOL) that what is claimed to be a proof is really a proof.

We have defined two new types, Type and Pterm, which represent HOL types and HOL terms (in fact, HOL-terms are only represented by those terms of type Pterm for which the predicate `Pwell_typed` holds). Furthermore, we have formalised a number of proof-theoretic concepts that are needed in the discussion of proofs, such as the concept of a variable being free in a term, a term having a certain type, two terms being alpha-equivalent etc.

We have also defined a type of sequents and a type of inferences. Proofs are defined as lists of correct inferences. The results are stored in a number of theories:

`proofaux`	Auxiliary results about lists and sets
`Type`	Formalisation of HOL types
`Pterm`	Formalisation of HOL terms
`inference`	Formalisation of sequents and inferences
`proof`	Proofs and provability
`derived`	Derived rules of inference

The aim of our formalisation is to be able to check proofs, not to generate them. This means that we do not necessarily have to copy the HOL inference rules, as functions which

*Åbo Akademi University, Turku, Finland. E-mail: jwright@aton.abo.fi

return theorems, inside HOL. Instead, it is enough that we are able to recognise a correct inference, once the result is given. This means that we do not have to capture HOL's intricate (and ill documented) procedures for variable renaming used in the two primitive inference rules INST_TYPE and SUBST. Our formalisation permits arbitrary renaming schemes, and the one used by HOL as a special instance.

A *theory* in HOL is characterised by a type structure, a set of defined constants and a set of axioms. In our formalisation, a type structure is represented by a list of pairs (op,n) of type string#num, where n is the arity of the type operator op.

The constants of a theory are represented by a list of pairs (const,ty) where ty is the generic type (which can be polymorphic) of the constant const.

Finally, the axioms of a theory are represented by a list of sequents. Sequents, in turn, are pairs (as,tm), where as is a set of terms (the assumptions) and tm is a term (the conclusion). The equations that define constants are also considered to be axioms.

For every concept we have formalised, we have also written a proof function. For example, if we have defined a new constant foo, e.g., by a defining theorem

```
⊢ foo x y = E
```

then there is also an ML function Rfoo which can be called in the following way:

```
#Rfoo ["e1";"e2"];;
⊢ foo e1 e2 = ...
```

where the right hand side is canonical (i.e., it cannot be further simplified using definitional theorems). Essentially, these proof functions do rewriting, but in an efficient way, compared to the REWRITE_RULE function.

Notation We assume that the reader is familiar with the HOL theorem prover and its syntax. We mainly use the syntax of HOL, nut we use ordinary logical symbols, rather than the ASCII character combinations used by HOL. When referring to HOL objects, we use `typewriter font`. Similarly, we show examples of interaction with the HOL system in typewriter font.

2 Types

The HOL logic has four different kinds of types: type constants, type variables, function types and n-ary type operators. To make the definition simple, we consider type constants and the function type to be special cases of type operators. The definitions are stored in the theory Type. Parts of this theory are documented in Appendix B.

2.1 Defining types

We define types as a recursive type with the following syntax:

```
Type = Tyvar string
     | Tyop string (Type)list
```

The HOL define-type package for automatic definitions of recursive types does not permit this syntax, so we have made this definition "by hand". To distinguish these "HOL-as-object-logic-types" from the HOL types we will from now on call them Types.

The proof function RType_eq takes a two-Type list and checks whether the Types are equal. For example, the following dialogue proves that bool→bool and bool are distinct types.

```
#RType_eq ["Tyop `fun`[Tyop `bool`[];Tyop `bool`[]]";"Tyop `bool`[]"];;
⊢ (Tyop `fun`[Tyop `bool`[];Tyop `bool`[]] = Tyop `bool`[]) = F
```

The type structure of the current theory is represented by a list of pairs of type :string#num. For example, the simplest possible theory (referring only to booleans) has the following type structure list:

```
[(`bool`,0);(`fun`,2)]
```

The type structure list is used when we check whether a Pterm is well-typed.

2.2 Destructor functions for types

In order to facilitate function definitions over Type, we have developed some infrastructure for making recursive function definitions over Type. Using this, we have defined "destructor functions" which look into Types. Thus Is_Tyvar ty holds if ty is of the form Tyvar s, while Is_Tyop ty holds if ty is of the form Tyop s ts.

We also often want to extract the components of a Type, given that its structure is known. For example, we define Tyvar_nam by

```
⊢def (∀s. Tyvar_nam(Tyvar s) = s) ∧
     (∀s ts. Tyvar_nam(Tyop s ts) = (εy. T))
```

i.e., Tyvar_nam ty returns the name of a type variable ty, and an arbitrary string if ty is not a type variable. In fact, the second conjunct in the definition of Tyvar_nam is not needed, we could just specify Tyvar_nam to satisfy

```
⊢ ∀s. Tyvar_nam (Tyvar s) = s
```

However, the definition we have makes our proof functions (in this case RTyvar_nam) more uniform. Similarly, we define Tyop_nam and Tyop_tyl which return the name and argument list of a type operator type, respectively.

2.3 Other functions over Types

When reasoning about inferences and proofs we need functions that check for occurrences of type variables in types and for type instantiation. The function Type_OK is defined recursively over Types using the infrastructure for function definitions mentioned above. Evaluating

```
#let Type_OK_DEF = new_Type_rec_definition(`Type_OK_DEF`,
#  "(Type_OK Typl (Tyvar s) = T) ∧
#   (Type_OK Typl (Tyop s ts) =
#      mem1 s Typl ∧ (LENGTH ts = corr s Typl ) ∧ EVERY(Type_OK Typl )ts)");;
```

yields the definitional theorem Type_OK_DEF:

```
⊢ (∀tyl s. Type_OK Typl (Tyvar s) = T) ∧
  (∀tyl s ts. Type_OK Typl (Tyop s ts) =
      mem1 s Typl ∧ (LENGTH ts = corr1 s Typl) ∧ EVERY(Type_OK Typl)ts)
```

Here mem1 s l holds if s is the first component in some pair in the list l and corr1 s l is the corresponding second component (these are all defined in the proofaux theory, see Appendix A). Essentially, the theorem says that a Type is OK if it is a type variable or it is composed from OK types by a permitted type operator.

Similarly, we can define other functions on Types. Type_occurs a ty is defined to hold if the type variable occurs anywhere in the type ty.

The function Type_replace is defined so that Type_replace tyl ty is the result of replacing type variables in the Type ty according to tyl, where each element of tyl is a pair (ty,s) of type Type#string.

The constant Type_compat is defined so that Type_compat ty ty' holds when ty is compatible with ty', in the sense that the structure of ty is can be mapped onto the structure of ty'.

The function Type_compat does not allow us to tell whether a type instantiation is correct. For example, we must be able to detect that bool→num is not a correct instantiation of the polymorphic type *→*, even though these two types are compatible. For this, we have defined Type_instl so that Type_instl ty ty' returns the list of type instantiations used in going from ty from ty'. This list can then be checked for consistency, using the function nocontr from the proofaux theory (see Appendix A).

3 Terms

We formalise terms using the same syntax as HOL uses. Thus a term can be a constant, a variable, an application or an abstraction. Variable names are represented by strings. The type Pterm permits terms that are not well-typed. Well-typing is enforced by a predicate Pwell_typed. Definitions and theorems from the Pterm theory can be found in Appendix C.

3.1 Basic definitions

We represent terms by a recursive type with the following syntax:

```
Pterm = Const string Type
      | Var string#Type
      | App Pterm Pterm
      | Lam string#Type Pterm
```

We will call these objects Pterms, to distinguish them from HOL terms.

The constants of the current theory are represented by a list. A constant always has a generic type which is given in this list. When the constant occurs in a term, it has an actual type which must be an instance of the generic type (this requirement is enforced by the rules of well-typedness). For example, the equality constant on the booleans is represented as the Pterm

```
Const `=`(Tyop `fun`[Tyop `bool`[];Tyop `fun`[Tyop `bool` [];Tyop `bool` []]])
```

and this term is well-typed if the list of constants contains the pair

```
(`=`, Tyop `fun`[Tyvar `*`;Tyop `fun`[Tyvar `*`;Tyop `bool`[]]])
```

as a member.

A simple logic might have the following the list of constants:

```
[`T`,Tyop `bool`[] ;
 `F`,Tyop `bool`[] ;
 `=`,Tyop `fun` [Tyvar `*`;Tyop `fun` [Tyvar `*`;Tyop `bool` []]]
 `⇒`,Tyop `fun` [Tyop `bool` [];Tyop `fun`[Tyop `bool` [];Tyop `bool`
[]]]]
```

i.e., truth, falsity, equality and implication.

The function `RPterm_eq` takes a two-Pterm list and checks whether the Pterms are equal.

3.2 Destructor functions on Pterms

Exactly as for Types, we have defined a number of destructors functions for Pterms. For example, `Is_Const t` holds if the Pterm `t` is a `Const` and `App_ty t` gives the type argument of the Pterm `t`, provided that it is an `App`. For details, see Appendix C.

3.3 Well-typedness

Every Pterm has a unique Type. The function `Ptype_of` is defined to compute the Type of a Pterm. The corresponding proof function is `RPtype_of`:

```
#RPtype_of ["App (Var(`f`,Tyop `fun`[Tyop `bool`[];Tyop `bool`[]]))
#                (Var(`x`,Tyop `bool`[]))"];;
⊢ Ptype_of (App (Var(`f`,Tyop `fun`[Tyop `bool`[];Tyop `bool`[]]))
                (Var(`x`,Tyop `bool`[])))
   = Tyop `bool`[]
```

Our syntax permits terms which are ill-typed, in the sense that they do not correspond to the any terms of a current HOL theory. A term is well-typed if it satisfies two requirements. First, the types of the subterms of applications and abstractions must match. Second, the constants occuring in the term must have types which are correct instantiations of their generic types. The function `Pwell_typed` checks these conditions. Well-typedness restrictions will not be considered further until we formalise the notion of a correct inference.

3.4 A simple pretty-printer

Our Pterms quickly become very large and ugly. Even a simple HOL-term like

```
λx. x ⇒ (x = y)
```

becomes the massive Pterm

```
Lam(`x`,Tyop `bool`[])
   (App(App(Const `⇒ `
            (Tyop`fun`[Tyop`bool`[];Tyop`fun`[Tyop`bool`[];Tyop`bool`[]]]))
         (Var(`x`,Tyop `bool`[])))
      (App(App(Const `=`
               (Tyop`fun`[Tyop`bool`[];Tyop`fun`[Tyop`bool`[];Tyop`bool`[]]]))
            (Var(`x`,Tyop `bool`[])))
         (Var(`y`,Tyop `bool`[])))))
```

which is difficult both to write and read. To simplify things, we have an ML function `tm_trans` which translates a HOL-term to the corresponding Pterm:

```
#tm_trans "λ(x:bool).x";;
"Lam (`x`,Tyop `bool`[])
     (Var(`x`,Tyop `bool`[]))"
```

and a function `tm_back` which makes the opposite translation

```
#tm_back "Lam (`x`,Tyop `bool`[])
#             (Var(`x`,Tyop `bool`[]))";;
"λx. x" : term
```

These functions will be used for entering and pretty-printing terms later on.

3.5 Free and bound variables

The notion of free and bound variables are defined in the obvious way. For example, we define `Pfree` so that `Pfree x t` holds if the variable `x` occurs free in the Pterm `t`. Similarly, we define the functions `Pbound` and `Poccurs`.

The proof functions for these constants are `RPfree`, `RPbound` and `RPoccurs`. For example, we have

```
#RPoccurs["(`x`,Tyop `bool`[])";tm_trans "λ(x:bool).x"];;
⊢ Poccurs (`x`,Tyop `bool`[])
          (Lam (`x`,Tyop `bool`[])
               (Var(`x`,Tyop `bool`[]))) = T
```

which says that the boolean variable `x` occurs free in the term `λx.x`. We also have versions of these constants that work on collections of variables and Pterms, for example the following:

`Plnotfree xl t`	holds if no variable in the list `xl` is `Pfree` in `t`
`Pallnotfree x ts`	holds if the variable `x` is `Pfree` in none of the Pterms in the set `ts`
`Plallnotfree xl ts`	holds if no variable in the list `xl` is `Pfree` in any of the Pterms in the set `ts`

For sets, we use the HOL `Finite-sets` library.

We also define `Plnotbound` and `Plnotoccurs` similarly. As usual, each proof function has the name of the corresponding constant, prefixed with `R`.

3.6 Occurrences of type variables in terms

`Pty_snotoccurs a ts` holds if the type variable `a` occurs in none of the Pterms in the set `ts`. Similarly, `Plty_snotoccurs al tl` holds if no type variable in the list `al` occurs in any of the Pterms in the set `ts`. For each of these constants, we also have a corresponding proof function.

3.7 Alpha-renaming

Alpha-renaming and substitution of a term for a variable are closely bound together. We have defined a function `Palreplace` so that `Palreplace t' tvl t` holds if `t'` is the result of substituting according to the list `tvl` and alpha-renaming. The list `tvl` consists of pairs `(t,a)` of type `Pterm#(string#Type)`, indicating what terms should be substituted for what variables.

The corresponding proof function is `RPalreplace`:

```
#RPalreplace ["Var(`y`,Tyop `bool`[])";
#             "[Var(`y`,Tyop`bool`[]),`x`,Tyop `bool`[]]";
#             "Var(`x`,Tyop `bool`[])"];;
⊢ Palreplace
     (Var(`y`,Tyop `bool`[]))
     [Var(`y`,Tyop `bool`[]),`x`,Tyop `bool`[]]
     (Var(`x`,Tyop `bool`[])) = T
```

which tells us that substituting the variable `y` for `x` in the term `x` yields the term `y`.

In order to appreciate larger examples and tests, we have a pretty-printer for printing theorems, similar to `tm_back` described earlier. This prettyprinter is a function `th_back`. It prints Pterms using `tm_back` and functions using dummy-functions that we have added. For example, we have defined a dummy constant `Xalreplace` which corresponds to `Palreplace`.

Evaluating

```
#RPalreplace [tm_trans "λz.z ⇒ x";
#             "[Var(`x`,Tyop`bool`[]),`y`,Tyop `bool`[]]";
#             tm_trans "λx.x ⇒ y"];;
```

yields a massive theorem, stating that this substitution is in fact correct. However, if we apply `th_back` to this theorem, we get it in a form which is easier to read:

```
#th_back it;;
⊢ Xalreplace(λz. z ⇒ x)[x,`y`,Tyop `bool`[]](λx. x ⇒ y) = T
```

Now we can define alpha-equivalence using an empty substitution; `Palpha t' t` holds if `t'` and `t` are alpha-equivalent.

$\vdash_{def}$ `∀t' t. Palpha t' t = Palreplace t'[] t`

The following example shows that our corresponding proof function `RPalpha` also detects incorrect alpha-renamings:

```
#th_back(RPalpha[tm_trans "λy y.y ⇒ y" ; tm_trans "λx y.y ⇒ x"]);;
⊢ Xalpha(λy y. y ⇒ y)(λx y. y ⇒ x) = F
```

i.e., the terms `λy y.y ⇒ y` and `λx y.y ⇒ x` are not alpha-equivalent.

3.8 Multiple substitutions and beta-reduction

We can now also formalise HOL's notion of a substitution, as it occurs in the inference rule SUBST. Assume that ttvl is a list of triples of type Pterm#Pterm#(string#Type). For each triple (tm',tm,d) in this list, tm' is a Pterm that is to replace tm and d is a dummy variable used to indicate the positions where this substitution is to be made. Then Psubst t' ttvl td t holds if t is the result of substituting some tm-terms for d-dummies in the term t and if t' is the result of substituting tm'-terms for d-dummies. Both substitutions are done according to ttvl, and they may involve alpha-renaming.

The corresponding proof function is RPalpha and it can recognise both correct and incorrect substitutions.

Beta-reduction is much easier to formalise. Our definition states that Pbeta t' x t1 t2 holds when t' is the result of beta reducing (λx. t1)t2. For details, see Appendix B.

3.9 Type instantiation

Type instantiation is quite tricky to check. There are two reasons for this. First, it is necessary to check that the type instantiation has not identified two variables that were previously distinct. Second, the type instantiation rule permits free variables to be renamed.

Checking a renaming of a free variable is more complicated that checking a renaming of a bound variable, because bound variables are always "announced" (in the left subtree of the abstraction), but a free variables can occur in two widely separated subtrees, without being announced in the same way.

Thus, we have been forced to define a number of auxiliary functions before defining the Ptyinst function. Assume that tyl is a list of pairs of type Type#string, indicating what types are to be substituted for what type variables. Furthermore assume that as is a set of Pterms (they represent the assumption of the theorem that is to be type-instantiated). Then Ptyinst as t' tyl t holds if t' is the result (after renaming) of replacing type variables in t according to tyl and if no variables that are type instantiated occur free in as.

4 Inferences

The theory of inferences has the Pterm theory as its ancestor. A number of definitions and theorems from this theory can be found in Appendix D.

4.1 Sequents

We represent sequents by a new concrete type with a very simple syntax:

```
Pseq (Pterm)set Pterm
```

where the first argument to Pseq is the set of assumptions and the second argument is the conclusion. The corresponding destructor functions are Pseq_assum and Pseq_concl.

4.2 Basic inferences

An inference step consists of a *conclusion* (result sequent) that is "below the line" and a list of *hypotheses* (argument sequents) that are "above the line".

HOL inference rules are functions which in addition to the hypotheses may require some information in the form of a term in order to compute the conclusion. For example, the rule of abstraction (ABS) in the logic is

$$\frac{\Gamma \;\vdash\; \mathtt{t} \;=\; t'}{\Gamma \;\vdash\; (\lambda x.\, t) \;=\; (\lambda x.\, t)}$$

(with the side condition that x must not be free in Γ). As an inference rule in the HOL system, ABS is a function which takes a term (representing the variable x) and a theorem (the hypothesis) as arguments and returns a theorem (the conclusion). Inferences are also dependent on the type structure Typl, the list of constants Conl and the list of axioms Axil of the current theory.

In our framework, inferences need only be checked. This means that our formalisation of an inference rule always has an additional (first) argument, which is the conclusion of the inference. For each inference rule, we define a function which returns a boolean value: T for a correct inference and F for an incorrect one.

4.2.1 The ASSUME rule

The ASSUME rule is modelled by the function PASSUME:

```
⊢def ∀Typl Conl as t tm. PASSUME Typl Conl (Pseq as t) tm =
    Pwell_typed Typl Conl tm ∧ Pboolean tm ∧ (t = tm) ∧ (as = {tm})
```

where Pboolean tm is defined to mean that the Pterm tm has boolean Type.

Notice that this is the point where we require well-typedness; the way we build up the inference rule checks ensures that only well-typed sequents can occur in a result sequent. Notice that well-typedness is not enough, we must also require that the term is boolean, i.e., that its Type is Tyop 'bool'[]. Because we must check for well-typedness, we must have the type structure Typl and the constant list Conl as explicit arguments to ASSUME.

The proof function RPASSUME is now used to prove the correctness of an ASSUME inference:

```
#RPASSUME[Typl;Conl;
#         "Pseq {Var('x',Tyop 'bool'[])} (Var('x',Tyop 'bool'[]))";
#         "Var('x',Tyop 'bool'[])"];;
⊢ PASSUME (...) (...)
           (Pseq {Var('x',Tyop 'bool'[])} (Var('x',Tyop 'bool'[])))
           (Var('x',Tyop'bool'[])) = T
```

where Typl and Conl stand for a suitable type structure and a suitable list of constants (we have replaced them by dots in the printout). Our pretty-printing function removes the Typl and Conl arguments to make the theorem more readable (we assume that the theory in question is known to the user).

```
#th_back RPASSUME[Typl;Conl;
#                 "Pseq {Var(`x`,Tyop `bool`[])} (Var(`x`,Tyop `bool`[]))";
#                 "Var(`x`,Tyop `bool`[])"];;
⊢ XASSUME (Pseq {x} x) x = T
```

The resulting theorem states that the sequent $\{x\} \vdash x$ is the correct result of the inference ASSUME x.

4.2.2 The REFL rule

The REFL inference is modelled by PREFL. Thus

```
PREFL Typl Conl (Pseq as t) tm
```

holds if the assumption set as is empty and t represents the term t̂m=tm. In addition to this tm must be well-typed.

4.2.3 The BETA_CONV rule

For the BETA_CONV inference, we have defined PBETA_CONV so that

```
PBETA_CONV Typl Conl (Pseq as t) tm
```

holds if the assumption set as is empty and tm is a beta-redex which reduces to t. Furthermore, we require that t is well-typed and boolean.

4.2.4 The SUBST rule

The SUBST rule is modelled by PSUBST. It is defined so that

```
PSUBST Typl Conl (Pseq as t) thdl td th
```

holds if the sequent Pseq as t is the result of performing a multiple substitution in theorem th according to the list thdl of pairs (theorem,dummy), where td is a term with dummies indicating the places where substitutions are to be made.

PSUBST also checks the dummy term td for well-typedness. No other checks are necessary. This is because th and all the theorems in thdl must be well-typed, since they are the conclusions of previous inferences.

4.2.5 The ABS rule

The function PABS models the ABS inference. Thus

```
PABS Typl Conl (Pseq as t) tm th
```

holds if t is the result of doing an abstraction of the term tm (which must be a variable with a permitted type) on both sides of the conclusion of th which must be an equality). Furthermore, the variable tm must not occur free in the assumption set as.

4.2.6 The INST_TYPE rule

For the INST_TYPE inference, we have defined PINST_TYPE so that

```
PINST_TYPE Typl (Pseq as t) tyl th
```

holds if t is the result of instantiating types in the conclusion of th according to tyl and if as is the same set as the assumptions in th. Furthermore, we require that the type variables that are being substituted for do not occur in as.

4.2.7 The DISCH rule

The function PDISCH models the DISCH inference. Thus

```
PDISCH Typl Conl (Pseq as t) tm th
```

holds if Pseq as t is the result of discharging the term tm in the theorem th. The term argument tm must be well-typed and boolean.

4.2.8 The MP rule

Finally, The function PMP models the MP inference. Thus

```
PMP (Pseq as t) th1 th2
```

holds if Pseq as t is the result of a Modus Ponens inference on th1 and th2.

4.3 Inferences

The arguments of the basic inference rules are not uniform. For example, the MP rule takes two theorems as arguments, while the REFL rule takes a single term. To be able to reason about inferences in a uniform way, we define a new type :Inference with the following syntax:

```
Inference = AX_inf Psequent
          | AS_inf Psequent Pterm
          | RE_inf Psequent Pterm
          | BE_inf Psequent Pterm
          | SU_inf Psequent (Psequent#string#Type)list Pterm Psequent
          | AB_inf Psequent Pterm Psequent
          | IN_inf Psequent (Type#string)list Psequent
          | DI_inf Psequent Pterm Psequent
          | MP_inf Psequent Psequent Psequent
```

The first production rule corresponds to an inference by axiom, each of the other rules corresponds to a basic inference rule.

The first argument of an inference constructor is always the conclusion of the inference. The remaining arguments represent the hypotheses and other arguments. The destructor function Inf_concl picks out the conclusion from an object of type inference while the destructor Inf_hyps picks out the list of hypotheses (for definitions, see Appendix D). Note that in some cases (e.g., AX_inf) the list of assumptionsis empty, but for SU_inf it can be of arbitrary length.

An inference is correct if it satisfies the defining properties of the inference rule. For example, an object built with DI_inf is a correct inference if its arguments satisfy the predicate PDISCH, i.e. if it represents an application of the HOL inference rule DISCH. The function OK_inf is defined to represent this notion of correct inference. Thus OK_inf i holds if and only if i represents a correct inference, according to the basic inference rules of the HOL logic.

The proof function for OK_inf is ROK_inf, and it identifies both correct and incorrect inferences. Using the pretty-printing facilities, we check a simple inference:

```
#th_back (ROK_Inf[Typl;Conl;Axil;
#    "BE_inf (Pseq {} ^(tm_trans "(λ(x:bool).x)y = y"))
#        ^(tm_trans "(λ(x:bool).x)y")"]);;
⊢ XOK_Inf (BE_Xinf (Xseq {} ((λx. x)y = y)) ((λx. x)y))
```

This tells us that the theorem $\vdash (\lambda x.\ x)y = y$ is the result of the following application of the BETA_CONV inference rule:

```
#BETA_CONV "(λx. x)y"
```

Or, to put it another way, it shows that the following inference is correct:

$$\frac{}{\{\} \quad \vdash \quad (\lambda x.\ x)y \ = \ y}$$

5 Proofs and provability

In this section, we consider the notions of provability and proofs. These two concepts are closely related, but we define them independently of each other. Both depend on the underlying notions of inference, i.e., on the predicate OK_Inf defined over the type of inferences. Selected parts of the proof theory are listed in Appendix E.

5.1 Provability

Provability is an inductive concept. A sequent is provable (within a given theory) if it is an axiom or if it can be inferred from provable sequents by a correct application of an inference rule.

We have defined the predicate Provable on sequents using the basic ideas from the HOL package for inductive definitions. However, our inductive relation is too complex to be handled by this package. This is because the SUBST rule infers a new sequent from a list of old sequents, rather than from a fixed number of old sequents. Thus the definitions and related proofs have been done "by hand".

The inductive nature of provability is captured in the following theorem, which describes the predicate Provable:

```
⊢ ∀Typl Conl Axil i s.
    (OK_Inf Typl Conl Axil i ∧ (s = Inf_concl i)) ∧
    EVERY (Provable Typl Conl Axil) (Inf_hyps i)
    ⇒ Provable Typl Conl Axil s
```

Note that we have a base case and an inductive case together here. The base case occurs when the list `Inf_hyps i` is empty. Note also that the determinants of the current theory (the type structure `Typl`, the constant list `Conl` and the axiom list `Axil`) are present as arguments to `Provable`.

We have also proved an induction theorem (rule induction) for the `Provable` predicate. This can be found in Appendix E.

5.2 Proofs

By a proof we mean a sequence of correct inferences where each inference has the following property: all the hypotheses of the inference must appear as conclusions of some inference appearing earlier on in the proof.

This is captured in the predicate `Is_proof`:

```
⊢ (∀Typl Conl Axil. Is_proof Typl Conl Axil[] = T) ∧
   (∀Typl Conl Axil i P. Is_proof Typl Conl Axil (CONS i P) =
     OK_Inf Typl Conl Axil i ∧
     lmem (Inf_hyps i) (MAP Inf_concl P) ∧
     Is_proof Typl Conl Axil P)
```

The corresponding proof function is called `RIs_proof`. This proof function is in fact a program that proves the correctness (or incorrectness) of a proposed HOL proof, i.e., a proof checker. The following simple example shows how it works together with the pretty-printing facility:

```
#th_back(RIs_proof [Typl;Conl;Axil;
#  "[MP_inf (Pseq {^(tm_trans "(y:bool)=y")} ^(tm_trans "(x:bool)=x"))
#           (Pseq {} ^(tm_trans "((y:bool)=y) ⇒ ((x:bool)=x)"))
#           (Pseq {^(tm_trans "(y:bool)=y")} ^(tm_trans "(y:bool)=y"))
#   ;AS_inf (Pseq {^(tm_trans "(y:bool)=y")} ^(tm_trans "(y:bool)=y"))
#           ^(tm_trans "(y:bool)=y")
#   ;DI_inf (Pseq {} ^(tm_trans "((y:bool)=y) ⇒ ((x:bool)=x)"))
#           ^(tm_trans "(y:bool)=y")
#           (Pseq {} ^(tm_trans "(x:bool)=x"))
#   ;RE_inf (Pseq {} ^(tm_trans "(x:bool)=x"))
#           (Var(`x`,Tyop`bool`[]))
#   ]"] );;
⊢ Is_xproof
   [MP_Xinf (Xseq {y = y} (x = x))
            (Xseq {} ((y = y) ⇒ (x = x)))
            (Xseq {y = y} (y = y));
    AS_Xinf (Xseq {y = y} (y = y)) (y = y);
    DI_Xinf (Xseq {} ((y = y) ⇒ (x = x))) (y = y) (Xseq {} (x = x));
    RE_Xinf (Xseq {} (x = x)) x] =
    T
```

This took 1 minute to prove on a Sparcstation ELC with plenty of memory. It encodes to the following proof:

1.	$\vdash\ x = x$	by REFL
2.	$\vdash\ y = y \Rightarrow (x = x)$	by DISCH, 1
3.	$\{y = y\}\ \vdash\ y = y$	by ASSUME,
4.	$\{y = y\}\ \vdash\ x = x$	by MP, 2,3

i.e., an example of adding an assumption to a theorem.

5.3 Relating proofs and provability

Proofs and provability are obviously related: a sequent should be provable if and only if there is a proof of it. We have proved that this in fact the case (this can be seen as a check that our definitions are reasonable):

```
⊢ Provable Typl Conl Axil s =
    (∃i P. Is_proof Typl Conl Axil(CONS i P) ∧ (s = Inf_concl i))
```

The proof of this theorem rests on the fact that appending two proofs yields a new proofs. Given proofs of all the hypotheses of an inference, this fact allows us to construct a proof of the conclusion by appending all the given proofs and adding the given inference.

5.4 Reasoning about proofs

There is, of course, no way to prove that our definition of a proof actually captures the HOL notion of a proof. However, we can reason about proofs and check that they satisfy some minimal requirements. As an example of this, we have proved that proofs can only yield sequents where the hypotheses and the conclusions are well-typed and boolean:

```
⊢ ∀P. Is_proof Typl Conl Axil P ∧ Is_standard(Typl,Conl,Axil) ⇒
        EVERY Pseq_boolean(MAP Inf_concl P) ∧
        EVERY (Pseq_well_typed Typl Conl) (MAP Inf_concl P)
```

where `Is_standard` holds for the triple `(Typl,Conl,Axil)` if the type structure contains booleans and function types, the constant list contains polymorphic equality and implication and the axiom list contains only well-typed boolean sequents.

The main result needed to prove the above theorem is that the functions `Palpha`, `Psubst` and `Ptyinst` function on `Pterms` preserve well-typedness (see Appendix C).

6 Derived inferences

In real proofs, we often use derived inference rules, rather than the primitive inference rules of a logic. These do not extend the logic, but they are convenient, as they make proofs shorter. The HOL system has a number of derived inference rules hard-wired into the system. This means that every HOL-proof consists of inferences belonging to a set of some thirty inference rules, rather than the eight primitive rules of the logic. In this section we show how derived inference rules can be defined within our framework.

6.1 Definition of derived inference

In order to make derived inference rules uniform, we let them have three arguments. The first argument is the name of the rule, the second argument is the conclusion and the third argument is a list of hypotheses. We can now define the notion of a derived inference rule using the `Provable` predicate: we have a derived inference of a (conclusion) sequent `s` from a list of (hypothesis) sequents `sl` if s can be proved when `sl` is added to the list of axioms, under the assumption that all terms occurring in `sl` are boolean and well-typed:

```
⊢def ∀Typl Conl Axil name s sl. Dinf Typl Conl Axil name s sl =
    (EVERY Pseq_boolean  sl ∧ EVERY (Pseq_well_typed Typl Conl) sl
      ⇒  Provable Typl Conl (APPEND sl Axil) s)
```

Using this definition, we can for example formalise the `ADD_ASSUM` rule

$$\frac{\Gamma \quad \vdash \quad \mathtt{t}}{\Gamma, t' \quad \vdash \quad \mathtt{t}}$$

This rule is encoded in the following theorem, which we have proved:

```
⊢ ∀Typl Conl Axil G t' t. Pwell_typed Typl Conl t' ∧ Pboolean t'
     ⇒ Dinf Typl Conl Axil 'ADD_ASSUM' (Pseq (t' INSERT G) t) [Pseq G t]
```

Note that the derived rules added in this fashion correspond to the traditional notion of an inference rule: they relate hypotheses and conclusion without additional arguments. However, this means that they do not have the same structure as the corresponding rules that the HOL system uses.

We can define a new notion of proof `Is_Dproof`, where derived inferences are permitted. It is then possible (but not trivial) to show that `Is_proof` and `Is_Dproof` are equally strong, in the sense that whenever there is a `Dproof` of a sequent, there is also a `proof` of it. For details, see Appendix F. This is quite reasonable, since both notions of proof are directly related to the notion of provability.

7 Conclusion

We have defined in the logic of HOL a theory which captures the notions of types, terms and inferences that is used in the HOL logic. Within this theory we defined the notions of provability and of proof and proved them to be related in the desired way: a boolean term is provable if and only if there exists a proof of it.

Together with the HOL theory, we have developed ML functions for proving each property introduced. These function are in fact a proof checker, i.e., a program which takes a purported proof as input and determines whether it is a proof or not. This proof checker is extremely slow, since it computes the result by performing a proof inside HOL. It is our hope that the theory of proofs can also be used as a basis for verifying more efficient proof checkers for higher order logic. Work on such a proof checker is under way.

HOL is a fully expansive theorem prover, which means that when proving theorems, it reduces derived rules of inference to sequences of basic inferences. This makes proofs longer and more time-consuming. Since our theory of proofs includes a method for proving the correctness of derived rules of inference, we have provided a formal basis for a faster HOL, where derived rules of inference can be added to the core of the system, once they have been proved correct. This idea was suggested for the HOL system by Slind [3].

The theory reported in this paper is for the HOL88 version of HOL. However, we have also ported the theory (but not the proof functions) to the Standard ML version HOL90.

Related work, but in a completely different framework, is reported in [1], where the type-checker of the Calculus of Constructions is implemented in the logic of Nqthm (the Boyer-Moore system).

References

[1] Robert S. Boyer and Gilles Dowek. Towards checking proof-checkers. In Herman Geuvers, editor, *Workshop on types for Proofs and Programs*, pages 51–70, 1993.

[2] M.J.C. Gordon. Mechanizing programming logics in higher-order logic. In G. Birtwistle and P.A. Subrahmanyam (ed.), *Current Trends in Hardware Verification and Theorem Proving.* Springer-Verlag, 1989.

[3] K. Slind. Adding new rules to an LCF-style logic implementation. In *Proc. 1992 International Workshop on Higher Order Logic Theorem Proving and its Applications*, Leuwen, Belgium, September 1992.

Appendix A: the `proofaux` theory

This appendix shows selected definitions from the `proofaux` theory. This theory defined a number of functions for handling lists and sets, needed in the handling of proofs.

```
Definitions --
  EVERY2_DEF
    |- (!P yl. EVERY2 P[]yl = T) /\
       (!P x xl yl.
         EVERY2 P(CONS x xl)yl = P x(HD yl) /\ EVERY2 P xl(TL yl))
  LAPPEND_DEF
    |- (LAPPEND[] = []) /\
       (!h t. LAPPEND(CONS h t) = APPEND h(LAPPEND t))
  LUNION_DEF
    |- (LUNION[] = {}) /\ (!h t. LUNION(CONS h t) = h UNION (LUNION t))
  lor_DEF |- (lor[] = F) /\ (!h t. lor(CONS h t) = h \/ lor t)
  corr1_DEF
    |- (!x. corr1 x[] = (@y. T)) /\
       (!x h t. corr1 x(CONS h t) = ((x = FST h) => SND h | corr1 x t))
  corr2_DEF
    |- (!x. corr2 x[] = (@y. T)) /\
       (!x h t. corr2 x(CONS h t) = ((x = SND h) => FST h | corr2 x t))
  lmem_DEF
    |- (!l. lmem[]l = T) /\
       (!x xl l. lmem(CONS x xl)l = mem x l /\ lmem xl l)
  mem_DEF
    |- (!x. mem x[] = F) /\
       (!x h t. mem x(CONS h t) = (x = h) \/ mem x t)
  mem1_DEF
    |- (!x. mem1 x[] = F) /\
       (!x h t. mem1 x(CONS h t) = (x = FST h) \/ mem1 x t)
  mem2_DEF
    |- (!x. mem2 x[] = F) /\
       (!x h t. mem2 x(CONS h t) = (x = SND h) \/ mem2 x t)
  nocontr_DEF
    |- (nocontr[] = T) /\
       (!xy xyl. nocontr(CONS xy xyl) =
         (~mem2(SND xy)xyl \/ (corr2(SND xy)xyl = FST xy)) /\ nocontr xyl)

Theorems --
  SEVERY_DEF
    |- (!P. SEVERY P{} = T) /\
       (!P x s. SEVERY P(x INSERT s) = P x /\ SEVERY P s)
```

Appendix B: the Type theory

This appendix shows selected theorems from the Type theory. These theorems characterise a number of functions defined over Types.

```
Types --  ":Type"

Theorems --
  Type_Axiom
    |- !f1 f2. ?! fn. (!s. fn(Tyvar s) = f1 s) /\
                      (!s ts. fn(Tyop s ts) = f2 s(MAP fn ts)ts)
% destructor functions %
  Is_Tyvar_DEF
    |- (!s. Is_Tyvar(Tyvar s) = T) /\ (!s ts. Is_Tyvar(Tyop s ts) = F)
  Is_Tyop_DEF
    |- (!s. Is_Tyop(Tyvar s) = F) /\ (!s ts. Is_Tyop(Tyop s ts) = T)
  Tyvar_nam_DEF
    |- (!s. Tyvar_nam(Tyvar s) = s) /\
       (!s ts. Tyvar_nam(Tyop s ts) = (@y. T))
  Tyop_nam_DEF
    |- (!s. Tyop_nam(Tyvar s) = (@y. T)) /\
       (!s ts. Tyop_nam(Tyop s ts) = s)
  Tyop_tyl_DEF
    |- (!s. Tyop_tyl(Tyvar s) = (@y. T)) /\
       (!s ts. Tyop_tyl(Tyop s ts) = ts)
% other functions %
  Type_OK_DEF
    |- (!tyl s. Type_OK tyl(Tyvar s) = T) /\
       (!tyl s ts. Type_OK tyl(Tyop s ts) =
         mem1 s tyl /\ (LENGTH ts = corr1 s tyl) /\ EVERY(Type_OK tyl)ts)
  Type_compat_DEF
    |- (!s ty. Type_compat ty(Tyvar s) = T) /\
       (!s ts ty. Type_compat ty(Tyop s ts) =
         Is_Tyop ty /\
         (Tyop_nam ty = s) /\
         (LENGTH(Tyop_tyl ty) = LENGTH ts) /\
         EVERY2 Type_compat(Tyop_tyl ty)ts)
  Type_occurs_DEF
    |- (!s' s. Type_occurs s'(Tyvar s) = (s = s')) /\
       (!s' s ts. Type_occurs s'(Tyop s ts) = lor(MAP(Type_occurs s')ts))
  Type_replace_DEF
    |- (!l s. Type_replace l(Tyvar s) = (mem2 s l => corr2 s l | Tyvar s)) /\
       (!l s ts. Type_replace l(Tyop s ts) = Tyop s(MAP(Type_replace l)ts))

Theorems --
  Type_instl_thm
    |- !s ts ty. Type_compat ty(Tyop s ts) ==>
        (Type_instl ty(Tyop s ts) = LAPPEND(MAP2 Type_instl(Tyop_tyl ty)ts))
```

Appendix C: the Pterm theory

This appendix shows selected definitions and theorems from the Pterm theory.

```
Types --  ":Pterm"

Definitions --
% destructor functions: a few examples %
  Is_App_DEF
    |- (!s ty. Is_App(Const s ty) = F) /\
       (!x. Is_App(Var x) = F) /\
       (!t1 t2. Is_App(App t1 t2) = T) /\
       (!x t. Is_App(Lam x t) = F)
  Const_ty_DEF
    |- (!s ty. Const_ty(Const s ty) = ty) /\
       (!x. Const_ty(Var x) = (@y. T)) /\
       (!t1 t2. Const_ty(App t1 t2) = (@y. T)) /\
       (!x t. Const_ty(Lam x t) = (@y. T))
  Var_var_DEF
    |- (!s ty. Var_var(Const s ty) = (@y. T)) /\
       (!x. Var_var(Var x) = x) /\
       (!t1 t2. Var_var(App t1 t2) = (@y. T)) /\
       (!x t. Var_var(Lam x t) = (@y. T))
  Lam_bod_DEF
    |- (!s ty. Lam_bod(Const s ty) = (@y. T)) /\
       (!x. Lam_bod(Var x) = (@y. T)) /\
       (!t1 t2. Lam_bod(App t1 t2) = (@y. T)) /\
       (!x t. Lam_bod(Lam x t) = t)
% well-typedness %
  Ptype_of_DEF
    |- (!s ty. Ptype_of(Const s ty) = ty) /\
       (!x. Ptype_of(Var x) = SND x) /\
       (!t1 t2. Ptype_of(App t1 t2) = HD(TL(Tyop_tyl(Ptype_of t1)))) /\
       (!x t. Ptype_of(Lam x t) = Tyop `fun` [SND x;Ptype_of t])
  Pboolean_DEF  |- !t. Pboolean t = (Ptype_of t = Tyop `bool`[])
  Pwell_typed_DEF
    |- (!Typl Conl s ty. Pwell_typed Typl Conl(Const s ty) =
         mem1 s Conl /\
         Type_OK Typl ty /\
         Type_compat ty(corr1 s Conl) /\
         nocontr(Type_instl ty(corr1 s Conl))) /\
       (!Typl Conl x. Pwell_typed Typl Conl(Var x) = Type_OK Typl(SND x)) /\
       (!Typl Conl t1 t2. Pwell_typed Typl Conl(App t1 t2) =
         Pwell_typed Typl Conl t1 /\ Pwell_typed Typl Conl t2 /\
         Is_Tyop(Ptype_of t1) /\ (Tyop_nam(Ptype_of t1) = `fun`) /\
         (LENGTH(Tyop_tyl(Ptype_of t1)) = 2) /\
         (HD(Tyop_tyl(Ptype_of t1)) = Ptype_of t2)) /\
       (!Typl Conl x t. Pwell_typed Typl Conl(Lam x t) =
         Pwell_typed Typl Conl t /\ Type_OK Typl(SND x))
% Free and bound variables in a term %
  Pfree_DEF
    |- (!x s ty. Pfree x(Const s ty) = F) /\
       (!x y. Pfree x(Var y) = (y = x)) /\
       (!x t1 t2. Pfree x(App t1 t2) = Pfree x t1 \/ Pfree x t2) /\
       (!x y t. Pfree x(Lam y t) = ~(y = x) /\ Pfree x t)
```

```
  Pbound_DEF
    |- (!x s ty. Pbound x(Const s ty) = F) /\
       (!x y. Pbound x(Var y) = F) /\
       (!x t1 t2. Pbound x(App t1 t2) = Pbound x t1 \/ Pbound x t2) /\
       (!x y t. Pbound x(Lam y t) = (y = x) \/ Pbound x t)
  Poccurs_DEF  |- !x t. Poccurs x t = Pfree x t \/ Pbound x t
  Plnotfree_DEF
    |- (!t. Plnotfree[]t = T) /\
       (!x xl t. Plnotfree(CONS x xl)t = ~Pfree x t /\ Plnotfree xl t)
  Plnotbound_DEF
    |- (!t. Plnotbound[]t = T) /\
       (!x xl t.
         Plnotbound(CONS x xl)t = ~Pbound x t /\ Plnotbound xl t)
  Plnotoccurs_DEF
    |- !xl t. Plnotoccurs xl t = Plnotfree xl t /\ Plnotbound xl t
  Pallnotfree_SPEC
    |- !x tms. Pallnotfree x tms = (!t. t IN tms ==> ~Pfree x t)
  Plallnotfree_SPEC
    |- !xl tms. Plallnotfree xl tms = (!t. t IN tms ==> Plnotfree xl t)
% substituting and renaming variables %
  Palreplace1_DEF
    |- (!t' vvl tvl s ty.
         Palreplace1 t' vvl tvl(Const s ty) = (t' = Const s ty)) /\
       (!t' vvl tvl x. Palreplace1 t' vvl tvl(Var x) =
         ((Is_Var t' /\ mem1(Var_var t')vvl) =>
          (x = corr1(Var_var t')vvl) |
          (~mem1 x vvl /\ (mem2 x tvl => (t' = corr2 x tvl) | (t' = Var x))))) /\
       (!t' vvl tvl t1 t2. Palreplace1 t' vvl tvl(App t1 t2) =
         Is_App t' /\ Palreplace1(App_fun t')vvl tvl t1 /\
         Palreplace1(App_arg t')vvl tvl t2) /\
       (!t' vvl tvl x t1. Palreplace1 t' vvl tvl(Lam x t1) =
         Is_Lam t' /\ (SND(Lam_var t') = SND x) /\
         Palreplace1(Lam_bod t')(CONS(Lam_var t',x)vvl)tvl t1)
  Palreplace_DEF
    |- !t' tvl t. Palreplace t' tvl t = Palreplace1 t'[]tvl t
  Palpha_DEF  |- !t' t. Palpha t' t = Palreplace t'[]t
  Psubst_triples_DEF
    |- (Psubst_triples[] = T) /\
       (!ttv ttvl. Psubst_triples(CONS ttv ttvl) =
         (Ptype_of(FST ttv) = Ptype_of(FST(SND ttv))) /\
         (Ptype_of(FST ttv) = SND(SND(SND ttv))) /\
         Psubst_triples ttvl)
  list13_DEF
    |- (list13[] = []) /\
       (!xyz xyzl. list13(CONS xyz xyzl) = CONS(FST xyz,SND(SND xyz))(list13 xyzl))
  Psubst_DEF
    |- !t' ttvl td t. Psubst Typl Conl t' ttvl td t =
        Psubst_triples ttvl /\ Pwell_typed Typl Conl td /\
        Plnotoccurs(MAP(SND o SND)ttvl)t /\
        Palreplace t(MAP SND ttvl)td /\
        Palreplace t'(list13 ttvl)td
  Pbeta_DEF  |- !t' x t1 t2. Pbeta t' x t1 t2 = Palreplace t'[t2,x]t1


% types in terms, type instantiation %
```

```
Pty_occurs_DEF
  |- (!a s ty. Pty_occurs a(Const s ty) = Type_occurs a ty) /\
     (!a x. Pty_occurs a(Var x) = Type_occurs a(SND x)) /\
     (!a t1 t2.
       Pty_occurs a(App t1 t2) = Pty_occurs a t1 \/ Pty_occurs a t2) /\
     (!a x t1.
       Pty_occurs a(Lam x t1) =
       Type_occurs a(SND x) \/ Pty_occurs a t1)
Pty_snotoccurs_SPEC
  |- !a tms. Pty_snotoccurs a tms = (!t. t IN tms ==> ~Pty_occurs a t)
Plty_snotoccurs_DEF
  |- (!tms. Plty_snotoccurs[]tms = T) /\
     (!h t tml. Plty_snotoccurs(CONS h t)tml =
       Pty_snotoccurs h tml /\ Plty_snotoccurs t tml)
Pnewfree1_DEF
  |- (!t bl s ty. Pnewfree1 t bl(Const s ty) = []) /\
     (!t bl x. Pnewfree1 t bl(Var x) =
       ((mem x bl \/ (FST(Var_var t) = FST x)) => [] | [Var_var t,x])) /\
     (!t bl t1 t2. Pnewfree1 t bl(App t1 t2) =
       APPEND(Pnewfree1(App_fun t)bl t1)(Pnewfree1(App_arg t)bl t2)) /\
     (!t bl x t1. Pnewfree1 t bl(Lam x t1) = Pnewfree1(Lam_bod t)(CONS x bl)t1)
Pnewfree_DEF  |- !t' t. Pnewfree t' t = Pnewfree1 t'[]t
Ptyinst1_DEF
  |- (!t bvl fvl tyl s ty. Ptyinst1 t bvl fvl tyl(Const s ty) =
       (t = Const s(Type_replace tyl ty))) /\
     (!t bvl fvl tyl x. Ptyinst1 t bvl fvl tyl(Var x) =
       (mem1(Var_var t)bvl =>
        (Is_Var t /\ (x = corr1(Var_var t)bvl) /\
         (SND(Var_var t) = Type_replace tyl(SND x))) |
        (mem2 x fvl =>
         (t = Var(FST(corr2 x fvl),Type_replace tyl(SND x))) |
         (t = Var(FST x,Type_replace tyl(SND x)))))) /\
     (!t bvl fvl tyl t1 t2. Ptyinst1 t bvl fvl tyl(App t1 t2) =
       Is_App t /\  Ptyinst1(App_fun t)bvl fvl tyl t1 /\
       Ptyinst1(App_arg t)bvl fvl tyl t2) /\
     (!t bvl fvl tyl x t1. Ptyinst1 t bvl fvl tyl(Lam x t1) =
       Is_Lam t /\ ~mem1(Lam_var t)fvl /\
     (SND(Lam_var t) = Type_replace tyl (SND x)) /\
       Ptyinst1(Lam_bod t)(CONS(Lam_var t,x)bvl)fvl tyl t1)
Ptyinst_DEF
  |- !as t' tyl t. Ptyinst as t' tyl t =
      Ptyinst1 t'[](Pnewfree t' t)tyl t /\
      Plallnotfree(MAP SND(Pnewfree t' t))as

Theorems --
% the type characterisation theorem %
 Pterm
  |- !f0 f1 f2 f3. ?! fn.
       (!s T'. fn(Const s T') = f0 s T') /\
       (!p. fn(Var p) = f1 p) /\
       (!P1 P2. fn(App P1 P2) = f2(fn P1)(fn P2)P1 P2) /\
       (!p P. fn(Lam p P) = f3(fn P)p P)

% characterisations of functions defined over sets %
 Pallnotfree_DEF
```

```
    |- (!x. Pallnotfree x{} = T) /\
       (!x tm tms. Pallnotfree x(tm INSERT tms) = ~Pfree x tm /\ Pallnotfree x tms)
  Plallnotfree_DEF
    |- (!xl. Plallnotfree xl{} = T) /\
       (!xl tm tms. Plallnotfree xl(tm INSERT tms) =
         Plnotfree xl tm /\ Plallnotfree xl tms)
  Pty_snotoccurs_DEF
    |- (!s. Pty_snotoccurs s{} = T) /\
       (!s tm tms. Pty_snotoccurs s(tm INSERT tms) =
         ~Pty_occurs s tm /\ Pty_snotoccurs s tms)
% theorems which show how well-typedness is preserved %
  Palrep_wty
    |- !t t' tvl.
        Pwell_typed Typl Conl t /\
        Palreplace t' tvl t /\
        EVERY(Pwell_typed Typl Conl)(MAP FST tvl) /\
        EVERY(\(t,v). Ptype_of t = SND v)tvl ==>
        Pwell_typed Typl Conl t' /\ (Ptype_of t' = Ptype_of t)
  Palpha_wty
    |- !t t'.
        Pwell_typed Typl Conl t /\ Palpha t' t ==>
        Pwell_typed Typl Conl t' /\ (Ptype_of t' = Ptype_of t)
  Pbeta_wty
    |- !t' x t1 t2.
        Pwell_typed Typl Conl t1 /\
        Pwell_typed Typl Conl t2 /\
        (Ptype_of t2 = SND x) /\
        Pbeta t' x t1 t2 ==>
        Pwell_typed Typl Conl t' /\ (Ptype_of t' = Ptype_of t1)
  Psubst_wty
    |- !t' ttvl td t.
        Pwell_typed Typl Conl td /\
        EVERY(Pwell_typed Typl Conl)(MAP FST ttvl) /\
        EVERY(Pwell_typed Typl Conl)(MAP(FST o SND)ttvl) /\
        Psubst Typl Conl t' ttvl td t ==>
        Pwell_typed Typl Conl t' /\ (Ptype_of t' = Ptype_of t)
  Ptyinst_wty
    |- !as t t' tyl.
        Ptyinst as t' tyl t /\ Pwell_typed Typl Conl t /\
        EVERY(Type_OK Typl)(MAP FST tyl) ==>
        Pwell_typed Typl Conl t' /\ (Ptype_of t' = Type_replace tyl(Ptype_of t))
```

Appendix D: the inference theory

This appendix shows selected parts from the `inference` theory.

```
Types --
  ":Psequent"      ":Inference"

Definitions --
  Pseq_assum_DEF  |- !as t. Pseq_assum(Pseq as t) = as
  Pseq_concl_DEF  |- !as t. Pseq_concl(Pseq as t) = t
  Pseq_boolean_DEF
    |- !as t. Pseq_boolean(Pseq as t) = SEVERY Pboolean as /\ Pboolean t
  Pseq_well_typed_DEF
    |- !Typl Conl as t. Pseq_well_typed Typl Conl(Pseq as t) =
         SEVERY(Pwell_typed Typl Conl)as /\ Pwell_typed Typl Conl t
% abbreviations for certain terms %
  PEQ_DEF
    |- !t1 t2. PEQ t1 t2 =
          App (App (Const `=` (Tyop `fun` [Ptype_of t1;
                                              Tyop `fun`[Ptype_of t1;Tyop `bool`[]]]))
                   t1)
              t2
  PIMP_DEF
    |- !t1 t2. PIMP t1 t2 =
          App (App (Const `==>` (Tyop `fun`[Tyop `bool`[];
                                              Tyop `fun`[Tyop `bool`[];Tyop `bool`[]]]))
                   t1)
              t2
  Is_EQtm_DEF
    |- !t. Is_EQtm t =
         Is_App t /\ Is_App(App_fun t) /\
         (App_fun(App_fun t) =
          Const `=` (Tyop `fun`[Ptype_of(App_arg t);
                                  Tyop `fun`[Ptype_of(App_arg t);Tyop `bool`[]]]))
% requirements for the eight primitive inferences %
  PASSUME_DEF
    |- !Typl Conl as t tm. PASSUME Typl Conl(Pseq as t)tm =
         Pwell_typed Typl Conl tm /\
         Pboolean tm /\ (t = tm) /\ (as = {tm})
  PREFL_DEF
    |- !Typl Conl as t tm. PREFL Typl Conl(Pseq as t)tm =
         Pwell_typed Typl Conl tm /\ (as = {}) /\ (t = PEQ tm tm)
  PBETA_CONV_DEF
    |- !Typl Conl as t tm. PBETA_CONV Typl Conl(Pseq as t)tm =
         Pwell_typed Typl Conl tm /\ (as = {}) /\
         Is_App tm /\ Is_Lam(App_fun tm) /\
         (t = PEQ tm(App_arg t)) /\
         Pbeta (App_arg t) (Lam_var(App_fun tm))
               (Lam_bod(App_fun tm)) (App_arg tm)
 Psubst_newlist_DEF
    |- (Psubst_newlist[] = []) /\
       (!h t. Psubst_newlist(CONS h t) =
         CONS (App_arg(Pseq_concl(FST h)),
                App_arg(App_fun(Pseq_concl(FST h))),SND h)
               (Psubst_newlist t))
```

```
  PSUBST_DEF
    |- !Typl Conl as t thdl td th. PSUBST Typl Conl (Pseq as t)thdl td th =
         Pwell_typed td /\
         EVERY Is_EQtm(MAP Pseq_concl(MAP FST thdl)) /\
         Psubst Typl Conl t(Psubst_newlist thdl)td(Pseq_concl th) /\
         (as = (Pseq_assum th) UNION (LUNION(MAP Pseq_assum(MAP FST thdl))))
  PABS_DEF
    |- !Typl Conl as t tm th. PABS Typl Conl(Pseq as t)tm th =
         Pwell_typed Typl Conl tm /\ Is_EQtm(Pseq_concl th) /\
         Is_Var tm /\ Type_OK Typl(SND(Var_var tm)) /\
         (t = PEQ (Lam (Var_var tm) (App_arg(App_fun(Pseq_concl th))))
                  (Lam (Var_var tm) (App_arg(Pseq_concl th))) /\
         (as = Pseq_assum th) /\ Pallnotfree(Var_var tm)as
  PINST_TYPE_DEF
    |- !Typl as t tyl th. PINST_TYPE Typl(Pseq as t)tyl th =
         EVERY(Type_OK Typl)(MAP FST tyl) /\
         Ptyinst as t tyl(Pseq_concl th) /\
         Plty_snotoccurs(MAP SND tyl)as /\
         (as = Pseq_assum th)
  PDISCH_DEF
    |- !Typl Conl as t tm th. PDISCH Typl Conl (Pseq as t) tm th =
         Pwell_typed Typl Conl tm /\ Pboolean tm /\
         (t = PIMP tm(Pseq_concl th)) /\ (as = (Pseq_assum th) DELETE tm)
  PMP_DEF
    |- !as t th1 th2. PMP(Pseq as t)th1 th2 =
         (Pseq_concl th1 = PIMP(Pseq_concl th2)t) /\
         (as = (Pseq_assum th1) UNION (Pseq_assum th2))
% functions over inferences %
  Inf_concl_DEF
    |- (!s. Inf_concl(AX_inf s) = s) /\
       (!s t. Inf_concl(AS_inf s t) = s) /\
       (!s t. Inf_concl(RE_inf s t) = s) /\
       (!s t. Inf_concl(BE_inf s t) = s) /\
       (!s tdl t s1. Inf_concl(SU_inf s tdl t s1) = s) /\
       (!s t s1. Inf_concl(AB_inf s t s1) = s) /\
       (!s tyl s1. Inf_concl(IN_inf s tyl s1) = s) /\
       (!s t s1. Inf_concl(DI_inf s t s1) = s) /\
       (!s s1 s2. Inf_concl(MP_inf s s1 s2) = s)
  Inf_hyps_DEF
    |- (!s. Inf_hyps(AX_inf s) = []) /\
       (!s t. Inf_hyps(AS_inf s t) = []) /\
       (!s t. Inf_hyps(RE_inf s t) = []) /\
       (!s t. Inf_hyps(BE_inf s t) = []) /\
       (!s sdl t s1. Inf_hyps(SU_inf s sdl t s1) = CONS s1(MAP FST sdl)) /\
       (!s t s1. Inf_hyps(AB_inf s t s1) = [s1]) /\
       (!s tyl s1. Inf_hyps(IN_inf s tyl s1) = [s1]) /\
       (!s t s1. Inf_hyps(DI_inf s t s1) = [s1]) /\
       (!s s1 s2. Inf_hyps(MP_inf s s1 s2) = [s1;s2])
```

```
 OK_Inf_DEF
   |- (!Typl Conl Axil s. OK_Inf Typl Conl Axil(AX_inf s) = mem s Axil) /\
      (!Typl Conl Axil s t.
        OK_Inf Typl Conl Axil(AS_inf s t) = PASSUME Typl Conl s t) /\
      (!Typl Conl Axil s t.
        OK_Inf Typl Conl Axil(RE_inf s t) = PREFL Typl Conl s t) /\
      (!Typl Conl Axil s t.
        OK_Inf Typl Conl Axil(BE_inf s t) = PBETA_CONV Typl Conl s t) /\
      (!Typl Conl Axil s tdl t s1.
        OK_Inf Typl Conl Axil(SU_inf s tdl t s1) = PSUBST Typl Conl s tdl t s1) /\
      (!Typl Conl Axil s t s1.
        OK_Inf Typl Conl Axil(AB_inf s t s1) = PABS Typl Conl s t s1) /\
      (!Typl Conl Axil s tyl s1.
        OK_Inf Typl Conl Axil(IN_inf s tyl s1) = PINST_TYPE Typl s tyl s1) /\
      (!Typl Conl Axil s t s1.
        OK_Inf Typl Conl Axil(DI_inf s t s1) = PDISCH Typl Conl s t s1) /\
      (!Typl Conl Axil s s1 s2.
        OK_Inf Typl Conl Axil(MP_inf s s1 s2) = PMP s s1 s2)

Theorems --
 Psequent  |- !f. ?! fn. !s P. fn(Pseq s P) = f s P
 Inference
   |- !f0 f1 f2 f3 f4 f5 f6 f7 f8.
       ?! fn.
        (!P. fn(AX_inf P) = f0 P) /\
        (!P0 P1. fn(AS_inf P0 P1) = f1 P0 P1) /\
        (!P0 P1. fn(RE_inf P0 P1) = f2 P0 P1) /\
        (!P0 P1. fn(BE_inf P0 P1) = f3 P0 P1) /\
        (!P0 l P1 P2. fn(SU_inf P0 l P1 P2) = f4 P0 l P1 P2) /\
        (!P0 P1 P2. fn(AB_inf P0 P1 P2) = f5 P0 P1 P2) /\
        (!P0 l P1. fn(IN_inf P0 l P1) = f6 P0 l P1) /\
        (!P0 P1 P2. fn(DI_inf P0 P1 P2) = f7 P0 P1 P2) /\
        (!P0 P1 P2. fn(MP_inf P0 P1 P2) = f8 P0 P1 P2)
```

Appendix E: the proof theory

This appendix shows selected parts from the proof theory.

```
Definitions --
  Is_proof_DEF
    |- (!Typl Conl Axil. Is_proof Typl Conl Axil[] = T) /\
       (!Typl Conl Axil i P.
         Is_proof Typl Conl Axil(CONS i P) =
         OK_Inf Typl Conl Axil i /\ lmem(Inf_hyps i)(MAP Inf_concl P) /\
         Is_proof Typl Conl Axil P)
  Is_standard_DEF
    |- !Typl Conl Axil.
        Is_standard(Typl,Conl,Axil) =
        EVERY(Pseq_well_typed Typl Conl)Axil /\ EVERY Pseq_boolean Axil /\
        mem1 `fun` Typl /\ (corr1 `fun` Typl = 2) /\
        mem1 `bool` Typl /\ (corr1 `bool` Typl = 0) /\
        mem1 `==>` Conl /\
        (corr1 `==>` Conl = Tyop`fun`[Tyop`bool`[];Tyop`fun`[Tyop`bool`[];Tyop`bool`[]]])
/\
        mem1 `=` Conl /\
        (corr1 `=` Conl = Tyop `fun`[Tyvar `*`;Tyop `fun`[Tyvar `*`;Tyop `bool`[]]]) /\
        mem1 `@` Conl /\
        (corr1 `@` Conl = Tyop `fun`[Tyop `fun`[Tyvar `*`;Tyop `bool`[]];Tyvar `*`])

Theorems --
  Provable_rules
    |- (!Typl Conl Axil i s.
         (OK_Inf Typl Conl Axil i /\ (s = Inf_concl i)) /\
         EVERY(Provable Typl Conl Axil)(Inf_hyps i) ==>
         Provable Typl Conl Axil s)
  Provable_induct
    |- !R'. (!Typl Conl Axil i s. (OK_Inf Typl Conl Axil i /\
              (s = Inf_concl i)) /\ EVERY(R' Typl Conl Axil)(Inf_hyps i) ==>
              R' Typl Conl Axil s) ==>
            (!Typl Conl Axil s.
              Provable Typl Conl Axil s ==> R' Typl Conl Axil s)
  Provable_cases
    |- Provable Typl Conl Axil s =
       (?i. OK_Inf Typl Conl Axil i /\ (s = Inf_concl i) /\
            EVERY(Provable Typl Conl Axil)(Inf_hyps i))
  Proof_APPEND
    |- !P1 P2. Is_proof Typl Conl Axil P1 /\ Is_proof Typl Conl Axil P2
               ==> Is_proof Typl Conl Axil(APPEND P1 P2)
  Provable_thm
    |- Provable Typl Conl Axil s =
       (?i P. Is_proof Typl Conl Axil(CONS i P) /\ (s = Inf_concl i))
  Inf_wty
    |- !i. Is_standard(Typl,Conl,Axil) /\ OK_Inf Typl Conl Axil i /\
           EVERY Pseq_boolean(Inf_hyps i) /\
           EVERY(Pseq_well_typed Typl Conl)(Inf_hyps i) ==>
           Pseq_boolean(Inf_concl i) /\ Pseq_well_typed Typl Conl(Inf_concl i)
  Proof_wty
    |- !P. Is_proof Typl Conl Axil P /\ Is_standard(Typl,Conl,Axil) ==>
           EVERY Pseq_boolean(MAP Inf_concl P) /\
```

```
EVERY(Pseq_well_typed Typl Conl)(MAP Inf_concl P)
```

Appendix F: the derived theory

This appendix shows selected parts from the derived theory.

```
Definitions --
  Dinf_DEF
    |- !Typl Conl Axil name s sl.
        Dinf Typl Conl Axil name s sl =
        EVERY Pseq_boolean sl /\
        EVERY(Pwell_typed Typl Conl) sl ==>
        Provable Typl Conl(APPEND sl Axil)s

Theorems --
  Is_Dproof_DEF
    |- (Is_Dproof Typl Conl Axil[] = T) /\
       (Is_Dproof Typl Conl Axil(CONS(n,s,sl)P) =
        Is_Dproof Typl Conl Axil P /\
        lmem sl(MAP(FST o SND)P) /\
        Dinf Typl Conl Axil n s sl)
  Dproof_Provable
    |- !P.
        Is_Dproof Typl Conl Axil P ==>
        EVERY(Provable Typl Conl Axil)(MAP(FST o SND)P)
  DADD_ASSUM
    |- !Typl Conl Axil G t' t.
        Pwell_typed Typl Conl t' /\ Pboolean t' ==>
        Dinf Typl Conl Axil `ADD_ASSUM`(Pseq(t' INSERT G)t)[Pseq G t]
  DUNDISCH
    |- !Typl Conl Axil G t1 t2.
        Dinf Typl Conl Axil `UNDISCH` (Pseq(t1 INSERT G)t2)
         [Pseq G (App (App (Const `==>`(Tyop `fun`[Tyop `bool`[];
                                                  Tyop `fun`[Tyop `bool`[];
                                                             Tyop `bool`[]]]))
                        t1)
                   t2)]
```