Representing higher-order logic proofs in HOL

J. VON WRIGHT

Åbo Akademi University, 20520 Turku, Finland

We describe an embedding of higher order logic in the HOL theorem proving system. Types, terms, sequents and inferences are represented as new types in the logic of the HOL system, and notions of proof and provability are defined. Using this formalisation, it is possible to reason about the correctness of derived rules of inference and about the relations between different notions of proofs. The formalisation is also intended to make it possible to reason about programs that handle proofs as their data (e.g., proof checkers).

Downloaded from https://academic.oup.com/comjnl/article/38/2/171/369014 by guest on 11 April 2024

1. INTRODUCTION

This paper describes a formalisation of higher order logic proof theory within the logic of the HOL theorem proving system. The aim is to be able to reason about the proofs that the HOL system produces. This can be useful in a number of ways. It gives a basis for reasoning about programs that handle proofs. One specific kind of program that we have in mind is a proof checker: a program that takes a purported HOL proof as input and checks that it actually is a proof. Furthermore, our theory can be used in formal reasoning about the HOL system itself. For example, the HOL system has implemented a number of non-primitive inference rules as basic rules (for efficiency reasons). Using our formalisation, it is possible to verify the soundness of such rules. Our formalisation also permits us to define different notions of proof (e.g., tree-structured proofs and linear proofs) and study how they are related.

Overview of the paper

Our aim is to formalise (in HOL) the logic of the HOL theorem prover. We define two new types, representing HOL types and terms. We formalise 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 also define a type of sequents and a type of (primitive) inferences. Using these notions, we define what it means for a term to be provable, given a list of axioms. We then define a notion of proof and show that the notions of provability and proof agree. Finally, we define the notion of derived inference and show how one can reason about derived rules of inference.

The aim of our work is to be able to reason about proofs, not to generate them. Thus, we need only be able to recognise a correct inference, once the result is given. This means that we do not have to capture HOL's intricate (and under-specified) procedures for variable renaming used in some inference rules. Our formalisation permits arbitrary renaming schemes, and the one used by HOL is a special instance.

A *theory* in HOL is characterised by a type structure, a set of constants and a set of axioms. We represent a type structure by a list of pairs `(op,n)`, 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 possibly polymorphic generic type of the constant `const`. The axioms are represented by a list of sequents. The theorems that characterise constants are also considered to be axioms. Sequents, in turn, are formed from pairs `(as,tm)`, where `as` is a set of terms (the assumptions) and `tm` is a term (the conclusion).

For every concept that we have formalised, we have also written a *proof function*. For example, when we define a new constant `foo` by a defining theorem

```
⊢ foo x = E
```

we also provide an ML function `Rfoo` (ML is the Meta Language of the HOL theorem prover) which, given a term `t` as argument returns the theorem

```
⊢ foo t = ...
```

where the right hand side is canonical (i.e., it cannot be simplified further using definitional theorems). Essentially, these proof functions do rewriting, but in an efficient way, compared with the built-in rewriting rules of the system.

The theory described in this paper comes as a contribution with HOL88 version 2.02. A more detailed description of the theory, together with listings, can be found in [9]. A port for the SML-based version of the HOL system (known as HOL90) also exists.

2. THE HOL SYSTEM AND ITS LOGIC

The HOL system is an interactive theorem prover for higher order logic. Below we give a brief description of the system and the logic that it is based on. For a more detailed description, we refer to [5, 6].

The logic of the HOL system is a polymorphic version of higher order logic, based on the Simple Typed

Lambda Calculus [4]. Essentially, it extends first-order logic by permitting lambda expressions that denote functions. It also permits higher order functions and quantification over arbitrary types. Every term in the logic has a type. The logic has a facility which permits the user to add new types and type operators. Polymorphic types are supported through the use of type variables. A constant specification facility lets the user extend the logic by introducing new constants. The basic HOL theory contains among other things the atomic types `bool` (truth values) and `num` (natural numbers), function types and a number of constants (e.g., implication $\Rightarrow$ and polymorphic equality $=$), together with axioms and theorems that characterise the constants.

The deductive system of HOL is a sequent encoding of a natural deduction system, with eight basic inference rules. These are

- `ASSUME` which asserts that $t \vdash t$ is always a theorem,
- `REFL` which asserts that equality is reflexive,
- `BETA_CONV`, the rule of beta conversion,
- `SUBST`, a rule for multiple substitution of equals for equals in theorems,
- `ABS`, abstraction; from $\Gamma \vdash t = t'$ infer $\Gamma \vdash (\lambda x.\ t) = (\lambda x.\ t')$ if x does not occur free in Γ,
- `INST_TYPE`, which permits type variables in a theorem to be instantiated, provided that the type variable being instantiated does not occur in the assumptions and that the instantiation does not identify two differently typed variables with the same name,
- `DISCH`, the rule for discharging an assumption, and
- `MP`, the rule of Modus Ponens inference.

A theorem is a sequent $\Gamma \vdash t$ which has been proved using these inference rules. Theorems are a secure type in the metalanguage ML.

The inference rules are ML functions which, given proper arguments, return theorems. It is possible to add derived rules of inference. Such rules are ML functions which specify how the basic inference rules should be combined to perform a derived inference. This means that derived rules do not extend the logic. For efficiency reasons, a number of additional rules have been made primitive in the actual implementation of the HOL system, even though they could be derived from the eight basic rules.

The user interacts with the system through an ML interface. By evaluating ML expressions, the user can create new theories, make definitions, store new theorems, etc. An important feature of the HOL system is the amount of existing infrastructure for defining new concepts and for proving theorems. Theorems can be proved by forward proof, since inference rules are ML functions which return theorems. The HOL system also supports backward proof through tactics. A number of libraries exist, with pre-proved theorems and derived inference rules that the user can load and use within the theory being developed. In this paper, we make use of existing libraries for strings and sets.

The implementation of HOL departs slightly from the specification of the logic (both are described in [5]). In such situations, we must decide which to model. The only major difference is that the implementation of the inference rule of type instantiation (`INST_TYPE`) permits names of free variables to be changed. Here our formalisation follows the implementation rather than the specification. The reason for this design choice is that we want to be able to reason about proofs as they are recorded by the HOL system.

Another difference is that the specification collects assumptions of sequents in sets while the implementation uses lists. Here, we have chosen the more abstract representation, i.e., sets.

Notation

The HOL system has a simple interface which uses ASCII character combinations for logical symbols. In this paper we mainly use the syntax of HOL, but we use ordinary logical symbols, for readability. The truth values are written as `T` and `F`. When referring to HOL objects and interaction with the system, we use `typewriter` font. The reader should note that terms of the HOL logic are enclosed in double quotes while strings are enclosed in single quotes. Lists are written in square brackets with semicolon as separator (e.g., `[T;T;F]`), while pairs are written in parentheses with comma as separator (e.g., `(T,1)`). Furthermore, `#` is the system prompt and `;;` the input terminator symbol.

3. REPRESENTING TYPES

The type system of the HOL logic has type variables and types constructed by applying n-ary type operators to type arguments. Type constants are nullary type operators. Function types are constructed using a binary type operator $\rightarrow$ (written infix).

Thus we have represented types by a new type in the HOL logic with the following syntax:

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

To distinguish these "HOL-as-object-logic-types" from the HOL types we will from now on call them Types.

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

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

The HOL type `bool` is then represented by `Tyop 'bool' []` while the function type `bool→bool` is represented by

```
Tyop 'fun' [Tyop 'bool' [];Tyop 'bool' []]
```


Downloaded from https://academic.oup.com/comjnl/article/38/2/171/369014 by guest on 11 April 2024

3.1. Functions for types

We have developed some infrastructure (i.e., some ML functions) for making recursive function definitions over Type. As an example, the function Type_OK is defined as follows:

```
#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=corr1 s Typl) ∧
#    EVERY (Type_OK Typl) ts)"
# );;
```

For this input, the HOL system returns the definitional theorem Type_OK_DEF:

```
⊢ (∀Typl s. Type_OK Typl (Tyvar s) = T) ∧
  (∀Typl 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 of some pair in the list `l` and `corr1 s l` is the corresponding second component (these are defined in a separate theory containing useful definitions and theorems, mainly about lists). 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 (the list `Typl` models the type structure).

Similarly, we have defined other functions on `Type`s. For example, `Type_occurs a ty` is defined to hold if the type variable `a` occurs anywhere in the type `ty`. The function `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'`. This function 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 a separate function.

4. REPRESENTING TERMS

A HOL term can be a constant, a variable, an application or an abstraction. Thus terms are represented by a new type with the following syntax:

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

We call these objects `Pterms`, to distinguish them from the HOL terms that they represent. Variable names are represented by strings (as implemented in the `string` library of the HOL system). The reader should note that we compose a lambda abstraction from a pair of type `string#Type` and a `Pterm`, whereas in the term syntax of the HOL system, lambda abstraction is composed from two terms. Our syntax makes the checking of well-formedness easier.

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, its actual type must be an instance of the generic type. A simple logic might have the following 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.

Equality on booleans is represented by the `Pterm`

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

Note that the `Type` of this `Pterm` is an instance of the `Type` of equality in the above list, with `Tyop `bool` []` replacing `Tyvar `*``.

4.1. Well-typedness

Every `Pterm` has a unique `Type`, computed by the function `Ptype_of`. This function simply returns the top-level type of the term. This implies that our syntax permits terms which are ill-typed, in the sense that they do not correspond to any (well-typed) HOL terms. A term is well-typed if it satisfies two requirements. First, the constants occurring in the term must have types which are correct instantiations of their generic types. Second, the types of the two subterms in an application must match. The function `Pwell_typed` checks these conditions.

At this point, we could have introduced a new type which represents well-typed terms of the HOL logic. However, since proof checking involves checking both correctness of inferences and well-formedness of terms, we want to permit ill-formed (ill-typed) terms to appear in purported proofs. Thus we would not gain anything by having a separate type representing well-typed terms.

4.2. A function for compressing terms

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

```
λx. x ⇒ (x = y)
```


Downloaded from https://academic.oup.com/comjnl/article/38/2/171/369014 by guest on 11 April 2024

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 into the corresponding Pterm:

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

and a function tm_back which does the opposite translation

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

These functions are used for entering and displaying terms that are used in simple examples.

4.3. 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.

We also have versions of these constants that work on collections of variables and Pterms. For example, Plallnotfree xl ts holds if no variable in the list xl is Pfree in any of the Pterms in the set ts.

4.4. Alpha-renaming

Alpha-renaming and substitution of a term for a variable are closely related. We have defined Palreplace so that Palreplace t' tvl t holds if t' is the result of substituting in t 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 definition of Palreplace is shown in the Appendix.

In order to appreciate larger examples and tests, we have a compressing function th_back for theorems, similar to tm_back described earlier. It uses tm_back to print subterms of theorems.

The proof function or Palreplace is called RPalreplace and it takes a list of arguments (one argument for each argument of 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 (that is, $\lambda z.\ z \Rightarrow x$ is a correct result when substituting x for y in $\lambda x.\ x \Rightarrow y$). However, if we apply th_back to this theorem, we are shown the theorem in the following form

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

which is much easier to read. Note that x here is a compressed notation for Var(`x`,Tyop `bool` []), while `y` is not compressed, i.e., it is in fact a one-character string. The modified turnstile symbol (]-) indicates that we do not see an actual theorem, but a compressed version.

We define alpha-equivalence using an empty substitution:

$\vdash_{def}$ ∀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"]);;
]- Palpha (λy y. y ⇒ y) (λx y. y ⇒ x) = F
```

i.e., the terms λyy. y⇒y and λxy. y⇒x are not alpha-equivalent. In fact, all those of our proof functions that check for properties can detect both instances and non-instances in this way.

4.5. Multiple substitutions

Using Palreplace we have formalised HOL's notion of a substitution, as it occurs in the inference rule SUBST. Assume that ttvl is a list of triples each having 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 tm-terms for d-dummies in the term td and if t' is the result of substituting tm'-terms for d-dummies in td. Both substitutions are done according to ttvl, and they may involve alpha-renaming.

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

Downloaded from https://academic.oup.com/comjnl/article/38/2/171/369014 by guest on 21 April 2024

4.6. Type instantiation

Type instantiation, as implemented by the inference rule INST_TYPE in HOL, is quite tricky to check. 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 (in this respect we follow the implementation rather than the specification of the HOL logic, see the discussion in Section 1.).

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

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`. `Ptyinst` is defined using `Palreplace` and a number of other auxiliary functions (some of these are described in Section 3.1.).

5. SEQUENTS AND INFERENCES

We represent sequents by a new concrete type with a single constructor `Pseq`. Its syntax is the following:

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

(`set` is a unary type operator for set formation, provided by the `finite_sets` library of HOL). 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`.

5.1. Inferences in the HOL system

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

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

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

(with the side condition that x must not occur 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).

5.2. Inferences as a new type

We represent inferences as syntactic objects of a new type. This type has nine constructors; one for inference by hypothesis and one for each primitive inference rule of the HOL logic (the logic has eight primitive inference rules). The syntax is

```
Inference
  = AXIOM_inf Psequent
  | ASSUME_inf Psequent Pterm
  | REFL_inf Psequent Pterm
  | BETA_inf Psequent Pterm
  | SUBST_inf Psequent (Psequent#string#Type)list
              Pterm Psequent
  | ABS_inf Psequent Pterm Psequent
  | INST_inf Psequent (Type#string)list Psequent
  | DISCH_inf Psequent Pterm Psequent
  | MP_inf Psequent Psequent Psequent
```

Here `AXIOM_inf` represents an inference by hypothesis (by axiom), while the remaining cases each correspond to a primitive inference rule (`BETA_inf` for `BETA_CONV` and `INST_inf` for `INST_TYPE`). The first argument of each constructor is the conclusion of the inference. The remaining arguments represent hypotheses and other arguments.

5.3. Checking inferences

The function `OK_inf` is defined to represent the notion of correct inference. Thus `OK_inf i` holds if and only if `i` represents a correct inference, according to the primitive 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 compressing functions, we check a simple inference:

```
#th_back
# (ROK_Inf[Typl;Conl;Axil;
#  "BETA_inf
#     (Pseq {} ^(tm_trans "(λ(x:bool).x)y = y"))
#     ^(tm_trans "(λ(x:bool).x)y")"]
# );;
]- OK_Inf
     (BETA_inf (Pseq {} ((λx. x)y = y))
               ((λx. x)y) )
```

(`^` is a "back-quote" which allows ML expressions to be evaluated inside HOL terms). 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"
```

5.4. Primitive inferences

We shall now show how the nine different kinds of inferences are checked. For each inference rule, we define

Downloaded from https://academic.oup.com/comjnl/article/38/2/171/369014 by guest on 11 April 2024

a function which returns a boolean value: T for a correct inference and F for an incorrect one. The correctness check is local, in the sense that it checks whether the result of an inference is valid under the assumption that the hypotheses (argument sequents) are valid. These functions are used by the function OK_Inf described above (the definition of OK_Inf is shown in the Appendix).

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 where well-typedness is enforced. The check ensures that the conclusion sequent Pseq as t is well-typed. To make this check, we must have the type structure Typl and the constant list Conl as explicit arguments to ASSUME.

In a similar way, the REFL and BETA_CONV inferences are modelled by PREFL and PBETA_CONV. Thus

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

holds if the assumption set as is empty, t represents the term tm=tm, and tm is well-typed. Similarly,

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

holds if the assumption set as is empty, tm is a beta-redex which reduces in a one-step beta reduction to t, and t is well-typed and boolean.

The SUBST rule is modelled by PSUBST;

```
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.

The function PABS models the ABS inference. Thus

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

holds if t is the result of abstracting 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.

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.

Finally,

```
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, and

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

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

6. 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 notion of correct inference, i.e., on the predicate OK_Inf defined in the Appendix.

6.1. Provability

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

We have defined the predicate Provable using the basic ideas from the HOL package for inductive definitions [5]. The inductive nature of provability is captured in the following theorem:

```
⊢ ∀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
```

In fact, Provable is defined to be the smallest relation satisfying this theorem. In the above theorem, Inf_concl is a function which returns the result sequent of an inference (the first Psequent argument in the syntax of inferences above) while Inf_hyps returns the list of hypotheses (the remaining Psequent arguments).

Note that the base case and the inductive case are handled together. The base case occurs when the list Inf_hyps i is empty. The list of hypotheses can be arbitrarily long in a SUBST-inference; for all other inferences it has length zero, one or two. We have also proved the induction theorem (rule induction) for the Provable predicate.

6.2. Proofs

By a proof we mean a sequence of correct inferences where each inference has the property that all its hypotheses appear as conclusions of some inference earlier in the proof.

This is captured in the following definition of Is_proof:

Downloaded from https://academic.oup.com/comjnl/article/38/2/171/339014 by guest on 11 April 2024

```
⊢ (∀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)
```

where `lmem l1 l2` holds if every element of list `l1` is also an element of `l2`.

The corresponding proof function is `RIs_proof`, which is in fact a proof checker. The following is an example of a (compressed) theorem produced using this proof function.

```
]- Is_Proof
   [MP_inf (Pseq {y = y} (x = x))
           (Pseq {} ((y = y) ⇒ (x = x)))
           (Pseq {y = y} (y = y));
    ASSUME_inf (Pseq {y = y} (y = y))
           (y = y);
    DISCH_inf (Pseq {} ((y = y) ⇒ (x = x)))
           (y=y)
           (Pseq {} (x=x));
    REFL_inf (Pseq {} (x = x))
           x]
  = T
```

This theorem states that the following is a correct 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

This is an example of adding an assumption to a theorem.

6.3. 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 proof. 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.

6.4. 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 shown 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(Typl,Conl,Axil)` holds if the type structure `Typl` contains at least booleans and function types, the constant list `Conl` contains at least implication and polymorphic equality and the axiom list `Axil` contains only well-typed boolean sequents.

In one respect, the above theorem is very important; it shows that the well-typedness checks in the functions that are used when checking an inference (described in Section 5.) are sufficient to guarantee that all conclusions that appear in a proof are well-typed. However, as the above theorem shows, this requires that all the theorems that are assumed as axioms are well-typed.

7. DERIVED INFERENCES

In real proofs, we often use derived rules of inference, rather than the primitive inference rules of a logic. Derived rules 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. Thus derived rules are an essential feature at the core of the HOL system.

7.1. Definition of derived inference

In order to make derived inference rules uniform, we let them have two arguments: the conclusion, and a list of hypotheses. We have a derived inference (`Dinf`) of a sequent `s` from a list of sequents `sl` if `s` can be proved when `sl` is added to the list of axioms:

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

7.2. Verifying the correctness of a derived rule of inference

As an example, we formalise the rule for adding an assumption to a theorem (the `ADD_ASSUM` rule of the HOL system). In traditional notation, this rule is expressed as follows:

Downloaded from https://academic.oup.com/comjnl/article/38/2/171/369014 by guest on 11 April 2024

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

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
           (Pseq (t′ INSERT G) t) [Pseq G t]
```

The proof of this theorem is in fact a verification of the correctness of the derived inference rule ADD_ASSUM.

Note that derived rules added in this fashion relate hypotheses and conclusion without additional arguments. In the HOL system, the added assumption t' is an argument to the inference rule ADD_ASSUM. However, different derived rules require different numbers of additional arguments of different types, and it is not possible to define Dinf in a way which would permit arbitrary additional arguments.

7.3. Proofs with derived inferences

We have also defined a new notion of proof, Is_Dproof, where derived inferences are permitted. We have proved (in HOL) 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, and vice versa. This is quite reasonable, since both notions of proof are directly related to the notion of provability.

Proofs with derived inferences cannot be checked with a function similar to Is_proof. This is because proving that a purported derived inference step is incorrect requires proving that no sequence of inferences could yield the conclusion in question, and this is much more complicated than proving that a proposed primitive inference is incorrect. The set of primitive inference rules is fixed by the syntax of the type Inference, but the set of derived inference rules can be extended freely.

8. CONCLUSION

We have defined in the logic of HOL a theory which captures the notions of types, terms and inferences that are 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 introduce.

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 (the example shown in Section 6.2. took 1 minute to run on a Sparcstation ELC with plenty of memory). 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 [11], and we believe that the methodology described in [10] can be used to verify a proof checker.

HOL is a fully expansive theorem prover, which means that when proving theorems, it reduces derived rules of inference to sequences of basic inferences. 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 [8].

It seems that there is generally a growing interest in using theorem proving system in the "introspective" way that we have described here. Similar ideas in a different framework are reported in [3], where a type checker for the Calculus of Constructions is implemented in the logic of Nqthm (the Boyer-Moore system). Related work on using proof-checkers to check metatheory is reported in [2] and [7], as well as in [1].

ACKNOWLEDGMENTS

I am grateful to Mike Gordon for many helpful discussions, and Tom Melham for showing how to define inductive relations in HOL. I also want to thank members of the Automated Reasoning Group in Cambridge and the anonymous referees for their comments on previous versions of this paper. Financial support from the Science and Engineering Research Council in Great Britain and the Research Institute of the Åbo Akademi Foundation in Finland is gratefully acknowledged.

REFERENCES

[1] S. Allen, R. Constable, D. Howe and W. Aitken. The semantics of reflected proof. In *Proc. 5th Annual Symposium on Logic in Computer Science*, pp. 95–107, Los Alamitos, USA, 1990. IEEE Computer Society Press.

[2] T. Altenkirch. A formalization of the strong normalisation proof for system F in LEGO. In *Typed Lambda Calculus and Applications*, Lecture Notes in Computer Science 664, pages 13–28, 1993.

[3] R. S. Boyer and G.Dowek. Towards checking proof-checkers. In *Workshop on Types for Proofs and Programs (Types '93)*, 1993.

[4] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68,1940.

[5] M. Gordon and T. Melham. *Introduction to HOL*. Cambridge University Press, New York, 1993.

[6] M.J.C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam (ed.), *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1988.

[7] J. McKinna and R. Pollack. Pure type systems formalized. In Herman Geuvers, editor, *Typed Lambda Calculus and Applications*, Lecture Notes in Computer Science 664, pages 289–305, 1993.

Downloaded from https://academic.oup.com/comjnl/article/38/2/171/369014 by guest on 11 April 2024

[8] K. Slind. Adding new rules to an LCF-style logic implementation. In M.J.C. Gordon L.J.M. Claesen, editor, *Higher Order Logic Theorem Proving and its Applications*, pp. 549–560, Leuwen, Belgium, September 1992. North-Holland.

[9] J. von Wright. Representing higher-order logic proofs in HOL. Techn. Rep. 323, Computer Lab, University of Cambridge, 1994.

[10] J. von Wright. Verifying modular programs in HOL. Techn. Rep. 324, Computer Lab, University of Cambridge, 1994.

[11] W. Wong. Recording HOL-proofs. Techn. Rep. 306, Computer Lab, University of Cambridge, 1993.

A SAMPLE DEFINITIONS

This appendix shows some definitions that are part of the theory of proofs. For the complete list of definitions, we refer to 9.

The inference checker `OK_Inf` is defined as follows:

```
OK_Inf_DEF =
⊢ (∀Typl Conl Axil s.
    OK_Inf Typl Conl Axil (AXIOM_inf s)
    = mem s Axil) ∧
  (∀Typl Conl Axil s t.
    OK_Inf Typl Conl Axil (ASSUME_inf s t)
    = PASSUME Typl Conl s t) ∧
  (∀Typl Conl Axil s t.
    OK_Inf Typl Conl Axil (REFL_inf s t)
    = PREFL Typl Conl s t) ∧
  (∀Typl Conl Axil s t.
    OK_Inf Typl Conl Axil (BETA_inf s t) =
    PBETA_CONV Typl Conl s t) ∧
  (∀Typl Conl Axil s tdl t s1.
    OK_Inf Typl Conl Axil
          (SUBST_inf s tdl t s1)
    = PSUBST Typl Conl s tdl t s1) ∧
  (∀Typl Conl Axil s t s1.
    OK_Inf Typl Conl Axil (ABS_inf s t s1)
    = PABS Typl Conl s t s1) ∧
  (∀Typl Conl Axil s tyl s1.
    OK_Inf Typl Conl Axil (INST_inf's tyl s1)
    = PINST_TYPE Typl s tyl s1) ∧
  (∀Typl Conl Axil s t s1.
    OK_Inf Typl Conl Axil (DISCH_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)
```

The constant `Palreplace` is defined using an auxiliary constant `Palreplace1` which has an additional argument (a list which contains bound variables encountered so far). The definition also uses other functions that we have defined. The functions `mem2` and `corr2` are similar to `mem1` and `corr1` (see Section 3.1.). `Is_var`, `Is_App` and `Is_Lam` check term construction while `Var_var`, `App_fun`, `App_arg`, `Lam_var` and `Lam_bod` are term destructors. Furthermore, `FST` and `SND` are projections on pairs and `b→t|t'` is HOL syntax for conditional expressions.

```
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
```


Downloaded from https://academic.oup.com/comjnl/article/38/2/171/369014 by guest on 11 April 2024