Proving the Completeness Theorem within Isabelle/HOL

James Margetson

22 September 2004

Abstract

This is a report about formalising a maths proof with the theorem prover Isabelle/HOL. The proof was for the completeness theorem of first order logic. The informal proof used symmetry arguments (duality) and also gave the cut elimination theorem as a corollary. Both these aspects were preserved formally.

The paper outlines this formalisation. It notes that parts of the proof can be viewed as a correctness proof for a naive proof procedure. Some speculative comments about the possibility of a proven reflection principle are made in the conclusions.

Contents

1 Introduction

Here, the Completeness theorem is formalised within Isabelle/HOL, [5]. The proof follows the first 5 pages of Wainer and Wallen's chapter in Aczel *et al.* [1]. This gives a concise presentation of FOL, a proof of completeness and obtains the cut elimination theorem as a corollary. Their presentation of formulas allow symmetry arguments to be used in proofs. This symmetry is preserved in the formal proofs. Their principal inference rules proceed by breaking down the structure of the sequents. This is closely related to a simple tableaux proof procedure. In §5 I note the connection between completeness and the correctness of the proof procedure.

Prior work. There have been several similar formal proofs in the area of meta mathematics.

Shankar [10] proved the incompleteness theorem within the Boyer-Moore theorem prover. The incompleteness theorem is Gödel's famous proof. He estimates that this was about 18 months work.

Persson has formalised a completeness proof for intuitionistic FOL within ALF, [9]. He interprets the FOL formulas in formal topology, where the excluded middle law need not hold. Here FOL formulas are interpreted in the enclosing HOL logic. For variables, he used both a naive naming representation and de Bruijn indices. He comments that his soundness proof is simpler with de Bruijn indices.

In [6], Paulson proves completeness for propositional logic in Isabelle/ZF to demonstrate the inductive definition package. This was an adaption of Tobias Nipkow's proof in Isabelle/HOL. Their representation for inference rules and deductions is also used here.

Raffalli has formalised an abstract completeness proof in the proof assistant AF2, available as part of the AF2 distribution.

More closely related work includes recent papers by John Harrison[**?**] and Anna Mikhajlova[**?**]. References to appear here soon!

Isabelle. Isabelle is a tactical theorem prover. It supports severals logics, but mainly HOL and ZF set theory in practice. The logic used here was HOL. However set theory might have been more natural for formalising this proof.

The Isabelle/HOL syntax is different to some of the other HOL provers. Selected comments about the syntax occur in footnotes where needed.

The Isabelle system has many tools available to help automate proofs. `blast_tac`,[7], in particular was very useful. As was the inductive definition package.

Outline. The representation of FOL in HOL is described in §2. Models and valuations are defined in §3. Inference rules and deductions are defined in §4. These give the two notions of truth. The completeness theorem and its relation to cut-elimination and the implicit proof procedure is described in §5. The main theorems are stated, interpreted and proofs outlined briefly. Some properties of trees were required, §6. Details about the proof procedure are given in §7. Observations and comments on further work are made in the remaining sections.

2 Formulas and Sequents

Formulas, F, have the following grammar:

$$\begin{array}{rclcl} F & ::= & P_i(v_1,\ldots,v_{n_i}) & \mid & \overline{P}_i(v_1,\ldots,v_{n_i}) \\ & \mid & F \wedge F & \mid & F \vee F \\ & \mid & \forall x.F & \mid & \exists x.F \end{array}$$

where P_i are predicates and v_i are variables. Predicates occur in complementary pairs and are countable, as are the variables. The *atomic* formula are predicates applied to variables, subject to arity restrictions. These are extended as shown, so formulas are in negation normal form. The other connectives, e.g. $\neg$ and $\Longrightarrow$, can be defined. The structural symmetry visible above is emphasised and used in proofs.

Formulas are represented as follows:

```
datatype vbl = X nat
datatype predicate = Predicate nat
datatype sign = Pos | Neg

datatype formula =
    FAtom sign predicate (vbl list)
  | FConj sign formula formula
  | FAll  sign formula
```

There are only three `formula` constructors. The structural symmetry has been made explicit with the `sign` argument. This gives concise definitions for purely structural functions like `freeVarsF`. Symmetry can be used in proofs by considering `formula` but not `sign` cases. A de Bruijn variable representation is used [2]. The quantifiers implicitly bind the least variable. This is why `FAll` does not identify the bound variable. Informally, predicates had an associated arity. This restriction can be dropped, so atoms are signed predicates on variable lists.

Initially, I had used a named-variable representation but a proof linking substitution and evaluation was a problem[1]. The de Bruijn representation greatly simplified the proofs and definitions.

The following functions are defined:

[1]Substitution needed to rename bound variables to avoid capture, so a single substitution became two substitutions. One solution was to accumulate a list of substitutions, but this complicated the proof.

```
freeVarsF :: formula => vbl set
subF      :: (vbl => vbl) => formula => formula
instanceF :: vbl => formula => formula
freshVar  :: vbl set => vbl
```

`subF` extends a variable mapping over a `formula`, respecting de Bruijn representation. Specifically, when extending a variable mapping, θ, inside a quantifier the body variables are mapped with a lifted version of θ. This fixes the least variable and maps successor variables from v^+ to $\theta(v)^+$. `subF` is used to define `instanceF u B` which gives $B(u)$ for some quantifier body $B(x_0)$. Calling `freshVar A` yields a variable not in `A`, provided `A` is finite.

Informally, sequents are finite multisets of formulas, which represent disjunctions. Sequents are represented as formula lists. Functions like `freeVarsS` are defined extending the corresponding formula functions. Sequents, like formulas, have only finitely many free variables. This property ensures there are always fresh variables available. Some mechanism to permute sequents is required. This is done via an explicit inference rule, which is described in §4.

3 Models and valuations

A formula is *valid* if it is true under all interpretations. The FOL formulas are interpreted in HOL. An interpretation assigns objects to variables, object-relations to predicates and restricts quantifiers to range of the object set. Restricting quantifiers is sometimes called relativisation. Here interpretations are represented by models and their assignments.

A model is a non empty set of objects and a relation on objects for each predicate. A new type is defined with functions:

```
objects      :: model => object set
evalP        :: model => predicate => object list => bool
modelAssigns :: model => (vbl => object) set
```

Each model has assignment functions. These are the possible assignment of objects to the variables. The objects must be drawn from the model's objects, not just the object type. This range restriction is only required when proving soundness of the $\exists$ rule. It is surprising how many proofs can still be done with 'incorrect' definitions.

Given a model and an associated assignment, each formula can be evaluated by extending the predicate evaluation structurally. Using duality,[2] the definition of `evalF` does not distinguish the symmetric cases[3].

```
sign  :: sign => bool => bool
evalF :: model => (vbl => object) => formula => bool
```

[2]Specifically, $A \vee B = \neg(\neg A \wedge \neg B)$ and $\exists x.B = \neg\forall x.\neg B$.

[3]`!x:A. P x` is bounded quantification, `x:A` is set membership.

```
evalF M phi (FAtom z P vs)  = sign z (evalP M P (map phi vs))
evalF M phi (FConj z A0 A1) =
    sign z (sign z (evalF M phi A0) & sign z (evalF M phi A1))
evalF M phi (FAll  z body) =
    sign z (!x: (objects M). sign z (evalF M (vblcase x phi) body))
```

The function `sign z` is either negation or identity depending on `z`. The term `vblcase x phi` defines a function by case analysis on variables, it could be written as $(\lambda\ 0.\ x \mid v^+.\ \phi(v))$. There is an equation linking evaluation and substitution which states that the variable mapping, θ, can be combined with the variable assignment, ϕ.

```
phi   :: vbl => object
theta :: vbl => vbl

evalF M phi (subF theta A) = evalF M (phi o theta) A
```

The proof of the equation does not split the symmetric cases. In an evaluation, only the assignments of the free variables matter.

```
equalOn (freeVarsF A) phi psi ==> evalF M phi A = evalF M psi A
```

Some properties of `equalOn` were proven to support some automated proofs.

Recall that considering all models and their assignments considers all interpretations. A sequent, Γ, is valid, `validS` Γ, if under all models and their assignments it evaluates to true.

3.1 The `object` type

The models draw their objects from a type, `object`. This type must contain, at least, a countable subset of objects: $obj_0, obj_1, \ldots$. The existence of this type and an injective function are asserted.

```
types  object
consts obj :: nat => object
rules  inj_obj "inj obj"
```

These are the only axioms introduced in the proof.

Initially I defined models to be polymorphic in the object type. This may have allowed the counter-model, described in §7, to be built directly on the variables. However, it was not possible to state the completeness theorem in HOL. Stating validity required quantifying over all models which required quantifying over all type instances (but not at the outside level). This is not a first class operation in HOL.

$$\frac{}{P_i(v_1,\ldots,v_{n_i}),\overline{P}_i(v_1,\ldots,v_{n_i})}\ \text{(Axioms)}$$

$$\frac{A_0, A_1, \Gamma}{A_0 \vee A_1, \Gamma}\ (\vee) \qquad \frac{A_0, \Gamma \quad A_1, \Delta}{A_0 \wedge A_1, \Gamma, \Delta}\ (\wedge)$$

$$\frac{A(x'), \Gamma}{\exists x.A(x), \Gamma}\ (\exists) \qquad \frac{A(x'), \Gamma}{\forall x.A(x), \Gamma}\ (\forall)\ \text{x' fresh}$$

$$\frac{A, A, \Gamma}{A, \Gamma}\ \text{(C)} \qquad \frac{\Gamma}{A, \Gamma}\ \text{(W)}$$

$$\frac{C, \Gamma \quad \neg C, \Gamma}{\Gamma, \Delta}\ \text{(Cut)}$$

Figure 1: The PC inference rules

4 Rules and deductions

The inference rules for FOL are given in Figure 1. These describe the Predicate Calculus, `PC`. Without the Cut rule, they describe `CutFreePC`.

An inference rule is represented as a pair: a conclusion and set of premises. Each FOL inference rule is represented as the set of all its instances, as in [6]. For example[4]:

```
Alls :: rule set
Alls == { z. ? A x Gamma . z = (FAll Pos A#Gamma,{instanceF x A#Gamma})
                              & x ~: freeVarsS (FAll Pos A#Gamma) }
```

The other FOL rules are defined similarly along with an additional rule to permute sequents. The rule sets `PC` and `CutFreePC` are defined to be the appropriate unions. These rule sets describe every single step inference that can occur.

Deductions are the set of sequents inferable from some rules. They are defined inductively, parameterised on a given rule set:

```
[| (conc, prems) : rules; prems : Pow (deductions rules) |]
==> conc : deductions rules
```

Axioms, having no premises, are the base case.

Isabelle's inductive definition package derives the induction rules automatically. The assumption that `prems` is a subset of `deductions rules` is stated using the power set, not the subset, operator. This form is required by the inductive definition package. These different representations of the same thing are not seen as equivalent.

[4]`?` is the existential quantifier, `#` is the list cons operator, `x ~: A` means `x` not in `A`.

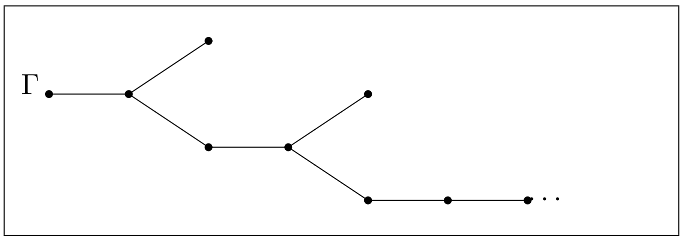

Figure 2: The search tree from Gamma

In Wainer and Wallen's chapter [1] several subsystems of `PC` are mentioned, characterised by subsets of the `PC` rules, e.g. `CutFreePC` and `MPC`. Parameterising deductions allows these to be represented in a uniform way. Proofs can be shared, e.g. soundness of deductions follows from soundness of the individual rule sets.

Recall that sequents are represented as lists, not multisets. The rule permuting sequents makes this multiset property explicit. This is also done by Smullyan [11] with a rule to exchange adjacent formulas. A similar rule would allow sequents to be considered as sets.

The `deductions` operator is monotonic, so `deductions CutFreePC <= deductions PC` since `CutFreePC <= PC`.

5 Completeness

The theorems leading to the completeness theorem are summarised in the following diagram.

$$\begin{tikzcd}
\texttt{Gamma : deductions PC} \arrow[r, "soundness"] & \texttt{validS Gamma} \arrow[d, "search - complete"'] \\
\texttt{Gamma : deductions CutFreePC} \arrow[u, "mono"] & \texttt{proofTree (search Gamma)} \arrow[l, "search - sound"']
\end{tikzcd}$$

The completeness theorem states that deducible and valid sequents coincide. The cut elimination theorem states that deductions of `PC` are also deductions of `CutFreePC`. These results can be seen in the above diagram, by following the implications.

Before commenting on the theorems the `search Gamma` term is explained: `search Gamma` represents the tree searched by the (implicit) tableaux prover.

The nodes are sequents. It is unfolded from Γ using a sub-node function. This function is justifiable using the inference rules. If all branches terminate on axioms then the tree corresponds to a derivation tree, and is called a *proof-tree*. A branch may not terminate. The implied proof procedure is: search the tree to check whether all branches terminate on axioms. The procedure is not a practical one compared with real provers, for example [3]. One key difference is the way existential witnesses are found. Here all variables are considered in turn. A more practical approach introduces 'free-variables' and tries to solve them through unification.

Some comments about the implications follow:

soundness This is proved by induction over deductions. The soundness of each rule set can be proved separately, and the results combined.

search-sound This says that if proof-search for Γ terminates and says yes, then Γ was a deduction. This is a soundness result for the proof-search. The proof uses a leaf-up induction principle for bounded trees, described in §6.2. This propagates the `deduction CutFreePC` property from the leaves to the root.

search-complete This says that for valid Γ the proof-search terminates and says yes, which is a completeness result for the proof-search. The contrapositive form is proven, in the diagram: negate the statements and reverse the arrows. This uses the non proof-tree to construct a counter model in which Γ is false. It requires a branch following the non proof-tree property, and also some fairness properties of the `subs` function. This is the most involved of the four theorems and is discussed in §7.

6 Trees

A representation of infinite trees is needed, to represent the attempted proof-search. The development is partly abstracted away from the completeness proof in the sense that it is parameterised by the sub-node function. However the representation of trees, as node sets, is not hidden.

Here, a tree is defined by a root node and a sub-node function. It is generated by unfolding the tree a level at a time, adding sub-nodes. An example tree is shown in Figure 3. The nodes are stratified into levels, equal sub-nodes are identified within a level. Infinite branching occurs if there are infinite sub-nodes. A terminal node occurs when their are no sub-nodes.

Each node is determined by its level node and its annotation. A tree is represented by a `(nat * 'a) set` which is defined inductively:

```
inductive "tree subs gamma"
  intrs
    tree0 "(0,gamma) : tree subs gamma"

    tree1 "[| (n,delta) : tree subs gamma; sigma : subs(delta) |]
           ==> (Suc n,sigma) : tree subs gamma"
```

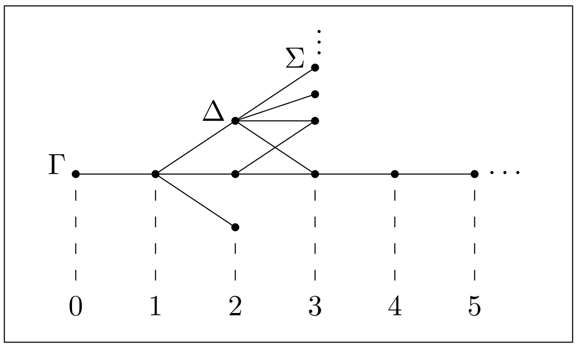

Figure 3: Trees

Some terminology was borrowed from [12]. A *fan* is a tree where each node has only finitely many sub-nodes. A *terminal* node is one with no sub-nodes. A *branch* is a function enumerating nodes, starting at the root, strictly following the sub-node function and repeating only on terminals. If all branches eventually terminate then the tree is said to be *bounded*.

Using definitions to capture properties concisely seemed to help. They are a form of abstraction. The term `branch subs Gamma f` has abstracted the relation between its arguments. It steps cleanly through proofs providing its properties when needed. Well related definitions may lead to shorter proofs. A proof using rules relating definitions may be shorter than one done after unfolded definitions, due to sharing of the derived-rule proofs.

An equation relates a tree with its sub-trees[5]:

```
tree subs gamma = insert (0,gamma) (UN delta:subs gamma .
                                        incLevel `` tree subs delta)
```

It is proved by induction and is used to expand `tree subs gamma` in some proofs, avoiding the need for an explicit induction. This followed the paper proofs.

Although called a tree, the structure here is more like the closure of the subs relation. The levels help describe when the tree is bounded.

6.1 Branches through trees

The completeness proof required constructing a branch following the non proof-tree property through a tree. More generally, inherited properties are considered, where

P inherited iff (P holds on a parent iff P holds on all children).

Other examples of inherited properties include:

[5]`UN x:A. B x` is the indexed union of a family of sets. `` f `` A `` is the image of the set `A` mapped by the function `f`.

- being bounded, (fans only).
- being founded on Q, meaning all terminal nodes satisfy Q.
- having a finite number of nodes, (fans only).

If such a P fails for a parent node then it must fail for some child. This allows a branch to be constructed which follows the failing property. The following theorem abstracts the path-construction function using exists.

```
[| inherited subs P; fans subs; ~P(tree subs gamma) |]
==> ? f . branch subs gamma f & (! n . ~P(tree subs (f n)))
```

This is the result required by the completeness proof, but it requires showing that the proof-tree property is inherited.

For fans, a parent tree can be constructed by moving the sub-trees up a level, taking the finite union (pairwise) and inserting the (non terminal) parent node. If a property is invariant over these structural operations then it is inherited on fans. The examples given above meet these conditions. A proof-tree is is a bounded tree which is founded on axioms. It is inherited because the conjunction of inherited properties is inherited[6].

```
(inherited subs P & inherited subs Q)
==> inherited subs (%x. P x & Q x)
```

Another application, taking P to be `finite`, gives a version of König's lemma:

```
[| fans subs; ~ finite (tree subs Gamma) |]
==> ? f . infBranch subs Gamma f
```

which states that a finitely branching tree with infinitely many nodes has an infinite branch.

6.2 Leaf-up induction

For bounded trees their depth strictly decreases on subtrees. This measure gives a terminal to root induction principle:

```
[| fans subs;
   bounded (tree subs Gamma);
   !delta. (! sigma:subs delta. P sigma) --> P delta |]
==> P Gamma
```

The fans assumption could be eliminated.

This is used to prove the search-sound result, which states that if Γ gives a proof-tree then it is a `CutFreePC` deduction. The property P is taken to be “is a `CutFreePC` deduction”. Recall that proof-trees are bounded trees which

[6] `%x. P x` is lambda abstraction.

are founded on axioms. Terminal nodes are axioms, which are `CutFreePC` deductions. For non-terminal nodes, if the sub-nodes are `CutFreePC` deductions then the parent node is also. So `subs` propagates P one level, as required. The induction rule propagates P from the leaves to the root, Γ.

Separating trees from the completeness proof made this inductive argument clearer.

7 Proof-search completeness

This section relates to the search-complete theorem. It is required to construct a counter-model falsifying a sequent, Γ, from a failed proof-tree. There is a non proof-tree branch through the tree. The (positive)-predicates are evaluated to false if they are on this branch. Under this evaluation, all formulas on the branch evaluate to false.

The `subs` function is described in §7.1. A fairness property of non proof-tree branches is described in §7.2. The theorems which justify that the evaluation falsifies branch formulas are given in §7.3.

7.1 The `subs` function

The function `subs` tries to construct a derivation tree following the structure of Γ. It *considers* the leading non atomic formula and attempts a `CutFreePC` justifiable proof step[7]:

$$\begin{array}{lcll}
atoms, A_0 \wedge A_1, \Gamma & \rightarrow & atoms, A_i, \Gamma & \text{for each } i \\
atoms, A_0 \vee A_1, \Gamma & \rightarrow & atoms, A_0, A_1, \Gamma & \\
atoms, \forall A, \Gamma & \rightarrow & atoms, A(x), \Gamma & x \text{ fresh} \\
atoms, (\exists A)^i, \Gamma & \rightarrow & atoms, A(x_i), \Gamma, (\exists A)^{i^+} &
\end{array}$$

The tree terminates as soon as complementary atoms, derived axioms, occur. It forks when considering conjunctions. A fresh variable can always be found when considering $\forall A$. The interesting case is $\exists A$, where the formula is retained. It introduces the i^{th} witness instance and puts $\exists A$ to the back of the queue. This ensures that all formulas are considered in turn, a fairness condition.

A separate representation of sequents is used,

```
types pseq  = "(atom list * (nat * formula) list)"
```

which separates atoms and formulas waiting to be considered. Each formula is annotated with a witness index, although these are only used in the exists case.

[7]Recall that sequents represent disjunctions of their formula.

7.2 A fairness property

On a branch following the non proof-tree property every sequent formula is eventually considered by `subs`. The following predicates are defined:

```
contains  f (i,A) :: nat => bool
considers f (i,A) :: nat => bool

EV :: (nat => bool) => bool
```

to describe i-formulas at a given point on the branch. `EV pred` says that `pred` is eventually true at some point. The fairness condition amounts to a proof that

```
[| branch subs gamma f; !n . ~ proofTree (tree subs (f n));
   EV (contains f iA) |]
==> EV (considers f iA)
```

This required a termination argument. The `barrier` for `(i,A)` are the leading i-formulas before it. Whilst `(i,A)` is contained but unconsidered, the measure

$$\sum_{B \in \texttt{barrier}} 3^{|B|}$$

strictly decreases. The proof involved properties of functions `takeWhile` and `sumList`. It also required solving some basic inequalities. These were proved using `blast_tac`.

7.3 Failing branch properties

A counter-model is constructed from the branch. Positive atoms are evaluated to false if they are on the branch. It is required to show that under this evaluation all formulas on the branch are false.

For `f` a branch following non proof-trees, the following are proven:

```
~ occurs (i,FAtom Pos p vs) | ~ occurs (j,FAtom Neg p vs)
occurs (i,FConj Pos A0 A1) ==> occurs (0,A0) | occurs (0,A1)
occurs (i,FConj Neg A0 A1) ==> occurs (0,A0) & occurs (0,A1)
occurs (i,FAll Pos body)   ==> ? v . occurs (0,instanceF v body)
occurs (i,FAll Neg body)   ==> ! v . occurs (0,instanceF v body)
```

where `occurs (i,A)` is short for `EV (contains f (i,A))`. The theorems justify that formula on this path can be falsified. The $\exists$ one required case analysis on previously considered formula and some temporal reasoning along the branch.

If a representation of temporal operators had been developed, then this temporal reasoning could have been clearer. These proofs require properties about `subs`, for example: relating what is contained, considered and introduced over one step.

8 Observations

- The de Bruijn indices representation for bound variables greatly simplified the related formula proofs. With this representation α-equivalent terms are identical.
- Definitions, representing terminology, were found to be useful. Proofs via derived-rules seemed shorter than those which unfolded definitions.
- The paper proofs used local definitions and theorems, but this was not done formally. The lack of context led to explicit parameters in definitions and common assumptions in lemmas. This was also noted by Paulson and Grąbczewski, [8].
- There were several places where definitions could be abstracted, for example there are many definitions of `freshVar` that satisfy the required intro-rule. Another example was the function which constructed a failing branch. Its definition was fully abstracted using exists. This followed the paper proofs, which showed there was a function, then took it to be `f`.
- The development of trees was separated out from the main proof. This made the proofs clearer, the generalised terms were smaller and it was easier to see simplifications, e.g. leaf-up induction became apparent. The additional parameterisation was not a problem. The temporal reasoning along the branch could also have been separated out.

  One reason for formalised proofs to blow-up, may be because they draw on results from another area. Informally, these results are assumed, but formally that area needs to be developed. The informal proof relied on trees and temporal reasoning implicitly.
- Isabelle provides automation. `blast_tac`,[7], a generic tableaux prover, was very useful. Solving trivial orderings on naturals seemed to be harder than it should have been though. This might be simpler now that decision procedures are available in Isabelle.

9 Further work

Proofs The formal proof scripts developed here are lengthy, not very readable and are tied to Isabelle/HOL. These are all areas for improvement. Proof scripts based on stating and connecting properties, rather than tactic sequences might address some of these issues.

Abstraction and parameterisation The benefits of scoping and module facilities in programming is well known. They seem related to the following aspects of proofs:

1. hiding definitions once derived rules and properties are available.
2. hiding arbitrary definitions, e.g. `freshVar`, which provide witnesses to justify abstraction.
3. parameterising proofs, maybe to support several instances. Separating out the development of trees is an example of this. One benefit is the possibility of reuse. Another is having a signature to work to. It seems easier to explore changes when fewer following proofs are broken.

Computation There are applications for efficient computation within theorem provers. Computation may be useful for concrete arithmetic or for translations, e.g. compiling programs between abstract machines. One of the design goals of ACL2 was to support efficient computation, and this was exploited in a floating point verification proof.

Reflection The evaluation function allows the FOL formula in HOL to be internalised, meaning they can be replaced by the evaluation of their corresponding representations. The completeness theorem links evaluations and deductions. So the representations can be transformed if justified by valid deductions. Through the evaluation function again new FOL formula in HOL can be obtained. This form of reflection is discussed in the metafunctions paper in [4].

In principle it seems possible to write proof procedures whose correctness is proven. The procedures would be coded as functions in the logic. To be practical they would require efficient computation.

The completeness theorem might help if reflection is also required. In this case set theory would be more suitable than HOL. There the reflected FOL formula would be all absolute FOL formula, ones whose quantifiers can be bounded. Whereas in HOL, here, they would only be a few specific formula about one type.

One problem might be handling equality. Another might be the use of derived theorems to justify inferences. And another might be coping with defined binding operators, like indexed union.

These comments are very speculative. However, they were suggested by noticing that the implicit tableaux prover could, in principle, be executed on the representations of FOL formula.

Acknowledgements Paul Taylor's Commutative Diagrams in TEX macros were used. EPSRC grants GR/K57381 'Mechanizing Temporal Reasoning' and GR/K77051 'Authentication Logics' funded some of the equipment used.

References

[1] Aczel, P., Simmons, H., Wainer, S. S., *Proof Theory: A Selection of Papers from the Leeds Proof Theory Programme*, Cambridge University Press, 1992

[2] Barendregt, H. P., *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, 1984

[3] Beckert, B., Posegga, J., leanT^AP: Lean tableau-based deduction, *Journal of Automated Reasoning* **15**, 3 (1995), 339–358

[4] Boyer, R., Moore, J., *The correctness problem in computer science*, Academic Press, 1981

[5] Paulson, L. C., Isabelle's object-logics, Tech. Rep. 286, Computer Laboratory, University of Cambridge, 1993

[6] Paulson, L. C., Set theory for verification: II. Induction and recursion, *Journal of Automated Reasoning* **15**, 2 (1995), 167–215

[7] Paulson, L. C., A generic tableau prover and its integration with Isabelle, Tech. Rep. 441, Computer Laboratory, University of Cambridge, Jan. 1998

[8] Paulson, L. C., Grąbczewski, K., Mechanizing set theory: Cardinal arithmetic and the axiom of choice, *Journal of Automated Reasoning* **17**, 3 (Dec. 1996), 291–323

[9] Persson, H., Constructive completeness of intuitionistic predicate logic: A formalisation in type theory

[10] Shankar, N., *Metamathematics, Machines, and Gödel's Proof*, vol. 38 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, 1994

[11] Smullyan, R. M., *First-Order Logic*, second corrected ed., Dover Publications, New York, 1995, First published 1968 by Springer-Verlag

[12] van Dalen, D., Doets, H. C., de Swart, H., *Sets: Naive, Axiomatic and Applied*, Pergamon Press, Oxford, 1978