Towards self-verification of HOL Light

John Harrison
Intel Corporation

IJCAR 2006, Seattle

August 18, 2006

Who checks the checker?

Formalization in a proof checker is often used to ensure correctness of proofs.

- Pure mathematics — better than traditional social process
- Formal verification — often the only practical option

Why should we believe that these proofs are more reliable than human proofs?

What if the underlying logic is inconsistent or the proof checker is faulty?

Who cares?

The robust view:

- Bugs in theorem provers do happen, but are unlikely to produce apparent “proofs” of real results.
- Even the flakiest theorem provers are far more reliable than most human hand proofs.
- Problems in specification and modelling are more likely.
- Nothing is ever 100% certain, and a foundational death spiral adds little value.

We care

The hawkish view:

- There has been at least one false “proof” of a real result.
- It’s unsatisfactory that we urge formality on others while developing provers so casually.
- It should be beyond reasonable doubt that we do or don’t have a formal proof.
- A quest for perfection is worthy, even if the goal is unattainable.

Prover architecture

The reliability of a theorem prover increases dramatically if its correctness depends only on a small amount of code.

- de Bruijn approach — generate proofs that can be certified by a simple, separate checker.
- LCF approach — reduce all rules to sequences of primitive inferences implemented by a small logical kernel.

The checker or kernel can be much simpler than the prover as a whole.

But it is still non-trivial . . .

HOL Light

HOL Light is an extreme case of the LCF approach. The entire critical core is 430 lines of code:

- 10 rather simple primitive inference rules
- 2 conservative definitional extension principles
- 3 mathematical axioms (infinity, extensionality, choice)

Everything, even arithmetic on numbers, is done by reduction to the primitive basis.

Still...

HOL Light does contain subtle code, e.g.

- Variable renaming in substitution and type instantiation
- Treatment of polymorphic types in definitions

It would still be nice to verify the core . . .

One fell swoop

We can imagine problems at several levels:

- The underlying logic is unsound or even inconsistent
- The formal definitions of the inference rules are incorrect
- The implementing code contains bugs

To eliminate all of these:

> Formalize the intended set-theoretic semantics of the logic and prove that the code implements inference rules that are sound w.r.t. this semantics.

HOL in HOL

We chose to verify HOL Light using . . . HOL Light. On the positive side:

- It's a capable theorem prover
- We know it well

However there are two apparent problems . . .

Logical objections

Taken too literally, our goal is impossible:

- Tarski: you cannot formalize the semantics of HOL in itself
- Gödel: you cannot prove the consistency of HOL in itself, unless it is in fact *inconsistent*

Actually we aim to prove two slightly different statements:

- HOL $\vdash Con(\mathsf{HOL} - \{\infty\})$
- HOL $+ I \vdash Con(\mathsf{HOL})$.

Practical objections

What can we deduce from a HOL Light proof that HOL Light contains no bugs?

Apparently, not much. But:

- Seems an improbable coincidence that a bug should help to verify its own absence.
- We can log the proof, thanks to Steven Obua, and check it in Isabelle/HOL or another different system.

Related work

- Jockum von Wright and others have formalized the notion of a HOL proof inside HOL (1994)
- Bill McCune and Olga Shumsky have verified an ACL2 proof checker for first-order logic for use with Otter (2000)
- Tom Ridge has verified a simple tableau prover for first-order logic in Isabelle/HOL (2005)

We go beyond these in:

- Set-theoretic model for a full mathematical framework.
- Verification of reasonable model of the prover's code.

Set-theoretic universe

We need a universe of sets containing models for all the types built up by '$\rightarrow$' from '`bool`' and '`ind`'.

$$|\texttt{ind}| < |I|$$
$$\forall S.\ |S| < |I| \Rightarrow |\wp(S)| < |I|$$

If we jetisson the axiom of infinity, we can prove the existence of such a set in plain HOL.

If we want to prove the consistency of full HOL, we add the analogous statement as an axiom.

The proofs are identical in all other respects.

Syntax of HOL

We map the OCaml definitions of the core logical notions into HOL (derived) type definitions.

```
type term = Var of string * hol_type
          | Const of string * hol_type
          | Comb of term * term
          | Abs of term * term
```

We slightly mangle the syntax of abstractions and stick to the primtive constants for now:

```
let term_INDUCT,term_RECURSION = define_type
 "term = Var string type
       | Equal type | Select type
       | Comb term term
       | Abs string type term";;
```

May need welltypedness hypotheses enforced in OCaml by abstract type.

Syntactic notions

Many OCaml syntax functions are mapped naively into HOL functions (they always terminate and never generate exceptions).

```
let rec vfree_in v tm =
  match tm with
    Abs(bv,bod) -> v <> bv & vfree_in v bod
  | Comb(s,t) -> vfree_in v s or vfree_in v t
  | _ -> tm = v
```

The function maps almost directly into HOL:

```
let VFREE_IN = define
  `(VFREE_IN v (Var x ty) <=> (Var x ty = v)) /\
   (VFREE_IN v (Equal ty) <=> (Equal ty = v)) /\
   (VFREE_IN v (Select ty) <=> (Select ty = v)) /\
   (VFREE_IN v (Comb s t) <=> VFREE_IN v s \/ VFREE_IN v t) /\
   (VFREE_IN v (Abs x ty t) <=> ~(Var x ty = v) /\ VFREE_IN v t)`;;
```

Type instantiation (1)

Type instantiation may generate exceptions (trapped internally in recursions).

```
let rec inst env tyin tm =
  match tm with
    Var(n,ty)   -> let ty' = type_subst tyin ty in
                   let tm' = if ty' == ty then tm else Var(n,ty') in
                   if rev_assocd tm' env tm = tm then tm'
                   else raise (Clash tm')
  | Const(c,ty) -> let ty' = type_subst tyin ty in
                   if ty' == ty then tm else Const(c,ty')
  | Comb(f,x)   -> let f' = inst env tyin f and x' = inst env tyin x in
                   if f' == f & x' == x then tm else Comb(f',x')
  | Abs(y,t)    -> let y' = inst [] tyin y in
                   let env' = (y,y')::env in
                   try let t' = inst env' tyin t in
                       if y' == y & t' == t then tm else Abs(y',t')
                   with (Clash(w') as ex) ->
                   if w' <> y' then raise ex else
                   let ifrees = map (inst [] tyin) (frees t) in
                   let y'' = variant ifrees y' in
                   let z = Var(fst(dest_var y''),snd(dest_var y)) in
                   inst env tyin (Abs(z,vsubst[z,y] t))
```

Type instantiation (2)

Formalized inside HOL using a sum type to model exceptions.

```
(INST_CORE env tyin (Var x ty) =
   let tm = Var x ty
   and tm' = Var x (TYPE_SUBST tyin ty) in
  if REV_ASSOCD tm' env tm = tm then Result tm' else Clash tm') /\
(INST_CORE env tyin (Equal ty) = Result(Equal(TYPE_SUBST tyin ty))) /\
(INST_CORE env tyin (Select ty) = Result(Select(TYPE_SUBST tyin ty))) /\
(INST_CORE env tyin (Comb s t) =
     let sres = INST_CORE env tyin s in
     if IS_CLASH sres then sres else
     let tres = INST_CORE env tyin t in
     if IS_CLASH tres then tres else
     let s' = RESULT sres and t' = RESULT tres in
     Result (Comb s' t')) /\
(INST_CORE env tyin (Abs x ty t) =
     let ty' = TYPE_SUBST tyin ty in
     let env' = CONS (Var x ty,Var x ty') env in
     let tres = INST_CORE env' tyin t in
     if IS_RESULT tres then Result(Abs x ty' (RESULT tres)) else
     let w = CLASH tres in
     if ~(w = Var x ty') then tres else
     let x' = VARIANT (RESULT(INST_CORE [] tyin t)) x ty' in
     INST_CORE env tyin (Abs x' ty (VSUBST [Var x' ty,Var x ty] t)))
```

Note that the ‘pointer eq’ optimizations have vanished!

The deductive system

This is the inductive definition of the entire deductive system.

```
|- (welltyped t ==> [] |- t === t) /\
   (asl1 |- l === m1 /\ asl2 |- m2 === r /\ ACONV m1 m2
    ==> TERM_UNION asl1 asl2 |- l === r) /\
   (asl1 |- l1 === r1 /\ asl2 |- l2 === r2 /\ welltyped(Comb l1 l2)
    ==> TERM_UNION asl1 asl2 |- Comb l1 l2 === Comb r1 r2) /\
   (~(EX (VFREE_IN (Var x ty)) asl) /\ asl |- l === r
    ==> asl |- (Abs x ty l) === (Abs x ty r)) /\
   (welltyped t ==> [] |- Comb (Abs x ty t) (Var x ty) === t) /\
   (p has_type Bool ==> [p] |- p) /\
   (asl1 |- p === q /\ asl2 |- p' /\ ACONV p p'
    ==> TERM_UNION asl1 asl2 |- q) /\
   (asl1 |- c1 /\ asl2 |- c2
    ==> TERM_UNION (FILTER((~) o ACONV c2) asl1)
                   (FILTER((~) o ACONV c1) asl2)
           |- c1 === c2) /\
   (asl |- p ==> MAP (INST tyin) asl |- INST tyin p) /\
   ((!s s'. MEM (s',s) ilist ==> ?x ty. (s = Var x ty) /\ s' has_type ty) /\
       asl |- p ==> MAP (VSUBST ilist) asl |- VSUBST ilist p)
```

The semantics

Semantics of terms is defined w.r.t. valuations of polymorphic type variables and term variables.

Here is the theorem that alpha-equivalent terms have the same semantics:

```
|- type_valuation tau /\ term_valuation tau sigma /\
   welltyped s /\ welltyped t /\ ACONV s t
   ==> (semantics sigma tau s = semantics sigma tau t)
```

The proofs are a bit messy but essentially routine. Definition of semantic entailment:

```
|- asms |= p <=> ALL (\a. a has_type Bool) (CONS p asms) /\
                 !sigma tau. type_valuation tau /\
                             term_valuation tau sigma /\
                              ALL (\a. semantics sigma tau a = true) asms
                              ==> (semantics sigma tau p = true)
```

Correctness proof

We can prove various individual inference steps correct, e.g. abstracting both sides of an equation:

```
|- ~(EX (VFREE_IN (Var x ty)) asl) /\ asl |= l === r
   ==> asl |= (Abs x ty l) === (Abs x ty r)
```

and so get our grand final theorems:

```
|- asl |- p ==> asl |= p
```

and

```
|- ?p. p has_type Bool /\ ~([] |- p)
```

To do

- Include arbitrary signatures
- Prove conservativity of definitional extension
- Use more realistic model of OCaml
- Verify extensions such as Melham-style type quantifiers