HOL Done Right

John Harrison
University of Cambridge Computer Laboratory
New Museums Site
Pembroke Street
Cambridge
CB2 3QG
England
jrh@cl.cam.ac.uk

21st August 1995

Abstract

In our opinion, history and compatibility considerations have rendered existing HOL implementations rather messy and badly organized. We describe how, building on joint work with Konrad Slind, we have produced a re-engineered HOL. Various experiments have been tried on this 'toy' version, and we will report the results.

1 Introduction

We assume that the reader is already familiar with HOL (Gordon and Melham 1993). The system has been evolving for about ten years now, and the user community has grown steadily. This means that it's not easy to make incompatible changes without upsetting people. The changeover from `HOL88` to `hol90` is already sufficiently traumatic that some pains were taken to make the systems as closely compatible as possible. However in the process some opportunities have been lost. Harrison and Slind (1994) have developed a cut-down and rationalized version of HOL, and both authors have continued to use it as a platform for experimentation with new ideas. Slind has added type quantifiers as suggested by Melham (1992), while the present author has kept the existing logic, but modified some of the system features.

Some of our experiments have ended in failure, but we are pleased with the results of some others. In this papers we give a quick overview of the most substantial changes which we consider (at present) to be successful. These ideas, and even their first implementation, are not all due to the present author, and some were already reported in our paper with Slind.

2 Implementation Language

The system has been implemented in CAML Light. This was partly motivated by our visceral dislike of SML syntax, and partly by our hope of making a system small enough to run even on small machines (at the time we began, the only SML compiler we had access to was Standard ML of New Jersey, which is notoriously memory-hungry). The system described here has been run, albeit slowly, on an Apple Macintosh portable (thanks to Malcolm Newey). The CAML system is remarkably economical, at least as regards code size. Heap size is still a problem, and we believe hash-consing should be investigated, but the situation is much better than for SML/NJ (which copies the whole heap).

The CAML Light system provides an excellent programming environment. The language is mostly simple, without the complexities of equality types or the SML module system. Excellent libraries are provided for interfacing with the system and even custom C code. A debugger and profiler are available. Automatic invocation of user printers at the top level is supported, and the library of box primitives makes it easy to write reasonable printers without too much trouble. The language lacks a special treatment of quotation (SML/NJ includes some well-thought-out facilities due to Slind), but it was easy enough to fit a crude filter on the front of the read-eval-print loop, and invoke it automatically when reading files. CAML is rather slow compared to SML/NJ, since it's a bytecode interpreter rather than a compiler, but it was fast enough for our purposes; in fact the system is much faster than `HOL88`, even though it does more work in primitive rules (see below). The fact that CAML builds on DEC Alphas (and seems to run about 3 times as fast as on Sparcs) is some compensation.

3 General organization

In existing HOL implementations, there is an ugly and dangerous melding together of the interface with the logical core. For example, in `hol90`, the abstract type of HOL types is encumbered with extra fields which are only used in type checking. Apart from being bad software engineering (typechecking is purely a user interface issue) this could actually be dangerous, since a bug in the typechecker might lead to ill-typed terms being constructed. Another example: all the functions which handle definitions take extra arguments to indicate parse status. Again, parsing status should be quite separate from logical concerns, and has little to do with whether something is a constant or a variable.

In our system, there is a rigorous separation of the logical core from interface modules. The interface consists of the parser, printer, typechecker and subgoal package. The typechecker uses a quite separate type of 'pretypes' as well as 'preterms' and all translations to real types and terms are mediated by the simple abstract type constructors which check correctness at each application. This is not a serious efficiency bottleneck, since one can normally find the type of a term by diving down the rator of each combination; generally after just a couple of steps one hits a constant or variable whose type is saved. (HOL88 stores types at all subnodes of a term, but for the reason we have just outlined, we believe this to be a waste of time.)

4 Programming

A few parts of the implementation have been rationalized. In particular we provide generic iterators for destructors, by analogy with `itlist` and friends for constructors.

```
let rec splitlist dest x =
  try let l,r = dest x in
      let ls,res = splitlist dest r in
      (l::ls,res)
  with _ -> ([],x);;

let rev_splitlist dest =
  let rec rsplist ls x =
    try let l,r = dest x in
```

```
      rsplist (r::ls) l
    with _ -> (x,ls) in
  fun x -> rsplist [] x;;

let striplist dest =
  let rec strip x acc =
    try let l,r = dest x in
        strip l (strip r acc)
    with _ -> x::acc in
  fun x -> strip x [];;
```

This gives a unified treatment of term destructors:

```
let conjuncts = striplist dest_conj;;
let disjuncts = striplist dest_disj;;

let strip_comb = rev_splitlist dest_comb;;
let strip_abs = splitlist dest_abs;;
let strip_forall = splitlist dest_forall;;
let strip_exists = splitlist dest_exists;;
let strip_select = splitlist dest_select;;
```

and in some cases, even theorem destructors:

```
let CONJUNCTS = striplist CONJ_PAIR;;
```

Note that this is much more efficient (w.r.t. consing) than the old HOL implementation of `CONJUNCTS` which used appends.

5 Parser

The parser allows the user to specify operator precedence and associativity (as in `hol90`, but not, alas, `HOL88` — assumptions that multiplication binds more strongly than addition account for a lot of beginners' errors in the author's experience). It also provides a class of prefixes, which associative to the left. In previous implementations, negation was hacked specially, but now it is just an instance of a prefix operator.

The parser is largely independent of the variable/constant status of terms; the recognition of constants only happens at the end, as part of typechecking. So variables can be given special parse status in just the same way as constants, useful for general theories of order relations, groups etc.

As an illustration of how the definition of infix constants is simplified, compare the definition of addition in our system:

```
parse_as_infix("+",(15,"right"));;

let ADD = new_recursive_definition num_Axiom
 `(!n. 0 + n = n) /\
  (!m n. (SUC m) + n = SUC(m + n))`;;
```

with the corresponding `HOL88` definition:

```
let ADD = new_recursive_definition true num_Axiom `ADD`
        "($+ 0 n = n) /\
         ($+ (SUC m) n = SUC($+ m n))";;
```

and the `hol90` definition:

```
val ADD = new_recursive_definition
   {name = "ADD",
    fixity = Infix 500,
    rec_axiom = num_Axiom,
    def = --`($+ 0 n = n) /\
             ($+ (SUC m) n = SUC($+ m n))`--};
```

Clearly it's much better. (Though part of the greater tidiness is because of the absence of theory files.)

6 Prelogic

Following Slind, we make `mk_const` take a type instantiation for the generic type rather than a type to produce. This simplifies the critical core by taking out matching (admittedly type matching isn't very complicated, but every little helps). Surprisingly, we have seldom found it inconvenient to use this version; although we provided a duplicate of the usual HOL `mk_const`, we found ourselves using the primitive one in almost all situations.

HOL's rewriting was optimized by Boulton (1993), based on ideas of Roger Fleming, to avoid rebuilding unchanged subterms by propagating an 'unchanged' exception. We extend this to the basic term operations of substitution (`subst`) and type instantiation (`inst`). We do not have any firm performance statistics, but intuitively we believe that this policy saves us a lot of consing, and hence garbage collection time. Of course, it also slightly increases the complexity of the critical code.

We have alternative implementations of both name-carrying and de Bruijn terms. The advantages of de Bruijn terms for giving clean, reliable definitions of operations like substitution are well-known. However they also have their drawbacks, particularly for a system like HOL where the abstract constructors for HOL terms present a 'name carrying' interface, and where it is common practice for users to break apart terms inside derived rules. Every time we break apart an abstraction, we need to descend throughout the subterm, replacing bound variables by free variables, a consing load which could be appalling in large abstraction-rich terms. This made us settle on name-carrying syntax as the main choice. We have grasped the nettle and implemented the troublesome operations `subst` and `inst`. We would like to perform some kind of informal verification to see that they are correct, but for the moment we just hope! Incidentally, we have used a special function `vsubst` which only substitutes for variables in the core. Besides being simpler, it avoids some repeated checks; in existing implementations we check whether the term in question is equal to the term to be substituted for before we even check whether it's a variable. All the core operations (and most non-core operations, `COND_CASES_TAC` being one of the few exceptions) can use this form.

The implementation of substitution for name-carrying syntax in a way which is both sound and efficient is itself an interesting research problem. There are plenty of choices over whether to trap name-capture errors at enclosing binders, or carry an environment of binders down and check for clashes at the time of substitution. All sorts of refinements are possible. Our implementation was motivated by the desire to make the common cases where variable capture doesn't occur (in fact variable clashes in `inst` *never* happen in the whole HOL core!) as fast as possible, while keeping the whole thing sound. We have a separate substitution function which is used recursively until we come to a lambda – this accounts for about half of substitutions and is much faster. With some trepidation, we make our code public:

```
let vsubst =
  let rec vsubst theta tm =
    match tm with
      Var(v)  -> (try snd(rev_assoc tm theta) with _ -> raise Unchanged)
    | Const(_) -> raise Unchanged
    | Comb(l,r) -> qcomb (fun (x,y) -> Comb(x,y)) (vsubst theta) (l,r)
    | Abs(_) -> fst(vasubst theta tm)
  and vasubst theta tm =
    match tm with
```

```
        Var(v)  -> (try snd(rev_assoc tm theta),[tm] with _ -> raise Unchanged)
      | Const(_) -> raise Unchanged
      | Comb(l,r) -> (try let l',vs = vasubst theta l in
                          try let r',vt = vasubst theta r in
                              Comb(l',r'),union vs vt
                          with Unchanged -> Comb(l',r),vs
                      with Unchanged ->
                          let r',vt = vasubst theta r in Comb(l,r'),vt)
      | Abs(v,bod) -> let theta' = filter (prefix not o prefix=v o snd) theta in

                      if theta' = [] then raise Unchanged else
                      let bod',vs = vasubst theta' bod in
                      let tms = map
                        (eval o fst o C rev_assoc theta') vs in
                      if exists (mem v) tms then
                        let fvs = itlist union tms (subtract (frees bod) vs) in
                        let v' = variant fvs v in
                        let bod',vars' = vasubst
                          (((eager [v'],v'),v)::theta') bod in
                        Abs(v',bod'),subtract vars' [v]
                      else
                        Abs(v,bod'),vs in
    fun theta ->
      if theta = [] then (fun tm -> tm) else
      let atheta = map
        (fun (t,x) -> if is_var x & type_of t = type_of x
                      then (lazy frees t,t),x
                      else failwith "vsubst: Bad substitution list") theta in
      qtry(vsubst atheta);;

  let inst =
    let rec inst env tyin tm =
      match tm with
        Var(n,ty)   -> let tm' = Var(n,type_subst tyin ty) in
                       if try not rev_assoc tm' env = tm with _ -> false
                       then raise (Clash tm') else tm'
      | Const(c,ty) -> Const(c,type_subst tyin ty)
      | Comb(s,t)   -> qcomb (fun (t1,t2) -> Comb(t1,t2)) (inst env tyin) (s,t)

      | Abs(y,t)    -> (let y' = qtry (inst [] tyin) y in
                        let env' = (y,y')::env in
                        if y = y' then Abs(y,inst env' tyin t) else
                        try Abs(y',qtry (inst env' tyin) t)
                        with (Clash(w') as ex) ->
                        if not 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''),type_of y) in
                        inst env tyin (Abs(z,vsubst[z,y] t))) in
    fun tyin -> qtry (inst [] tyin);;
```

Note that `qcomb` just applies a function to arguments, but propagates an 'unchanged' exception if applicable to both arguments. And `qtry` just traps out the exception completely. Laziness is used to avoid calculating free variable lists more than once, or at all where they are not needed. Bugs in the above code, or suggested improvements, will be gratefully (though not gracefully) received.

We have changed the notion of generalized binding. For example, $\lambda(x, y).\, P[x, y]$ is a syntactic sugar (modulo a tagging constant called `GABS` for the prettyprinter) $\varepsilon f.\ \forall x\ y.\ f(x, y) = P[x, y]$. This generalizes to arbitrary varstructs, which need not be based on disjoint, injective constructors (though otherwise the analog of beta reduction needs to be justified by hand). It also simplifies the parser since pairing does not have to be treated specially. In HOL, after all, pairing is just another binary operator[1]. Finally, it generalizes in an obvious way to case expressions where there

[1] As in Classic ML — pity SML forgot this!

are several alternative matches.

7 Choice of Primitives

The existing HOL systems are notionally based on the primitive inference rules ASSUME, REFL, BETA_CONV, SUBST, ABS, INST_TYPE, DISCH and MP and the axioms BOOL_CASES_AX, IMP_ANTISYM_AX, ETA_AX, SELECT_AX and INFINITY_AX.

However in fact a large number of additional rules of inference are hardwired in: ADD_ASSUM, AP_TERM, AP_THM, CCONTR, CHOOSE, CONJ, CONJUNCT1, CONJUNCT2, CONTR, DISJ1, DISJ2, DISJ_CASES, EQT_INTRO, EQ_IMP_RULE, EQ_MP, ETA_CONV, EXISTS, EXT, GEN, IMP_ANTISYM_RULE, IMP_TRANS, INST, MK_EXISTS, NOT_ELIM, NOT_INTRO, SPEC, SUBS, SUBST_CONV, SUBS_OCCS, SYM and TRANS.

This situation is lamentable; one of the selling points of HOL is that all inference is reduced to a few low-level primitives. Admittedly these pseudo-derived rules are not very complicated, but they allow plenty of scope for making errors. For example, in CHOOSE, the precise sideconditions on free variables require some care, whereas they automatically appear from the proper derivation. In the source, proper derived versions are given commented out, and the author has been told that they sometimes perform differently (w.r.t. bound variable renaming for example).

Why has this situation arisen? In some cases, because a proper derivation is simply too inefficient in practice (for example, SPEC). But in most cases the inefficiency has little impact on overall system performance, since the rules are very seldom used and the derivations are not too slow. The pseudo-derivations are probably a historical relic from the time when even this small difference in performance was significant.

We undertook an analysis of what would be a suitable set of small, simple primitives in terms of which the others could be implemented efficiently. We decided to adopt the following: REFL, SYM, TRANS, BETA_CONV, ABS, MK_COMB, ASSUME, DISCH, MP, EQ_MP, IMP_ANTISYM_RULE, INST_TYPE, INST.

Why these choices? They are all simple. Except for INST_TYPE, INST and BETA_CONV, they are almost trivial; these three involve the substitution and type instantiation primitives. We decided to jettison SUBST in favour of a few more, but simpler, rules, because its specification seems rather complicated for a primitive. In any case, rewriting is based on the congruence rules which we adopt as primitive, so SUBST hardly has a role except as an economical way of reaching these other properties. We made IMP_ANTISYM_RULE a rule rather than an axiom because it seems to belong naturally with the others. The primitives group naturally into equivalence and congruence properties of equality, lambda calculus conversion, basic properties of implication and its relationship to equality, and instantiation.

They are also sufficient to derive other rules efficiently. Because we have INST, most other rules can be implemented based on a proforma theorem which can be instantiated with efficient. The set of primitive chosen is of course redundant; we know INST is derivable, but not efficiently, and REFL is a trivial consequence of TRANS and any other equation (such as the result of BETA_CONV). By the way, since TRANS works up to alpha-equivalence, alpha conversion is also easily derivable from it, whereas a direct proof is too inefficient.

We now have axioms ETA_AX, SELECT_AX and INFINITY_AX. However we introduce them slowly, because much of the logic can be developed without them; we also do not introduce the selector itself till much later. Actually, we derive BOOL_CASES_AX from SELECT_AX, but we also get a long way intuitionistically. This necessitates a change in the definition of the existential quantifier, from:

```
let EXISTS_DEF = new_definition
 `$? = \P:A->bool. P($@ P)`;;
```

to

```
let EXISTS_DEF = new_definition
 `$? = \P:A->bool. !Q. (!x. P x ==> Q) ==> Q`;;
```

This definition, while opaque at first sight, is the natural infinitary analog of the existing HOL definition of disjunction:

```
let OR_DEF = new_definition
 `$\/ = \t1 t2. !t. (t1 ==> t) ==> (t2 ==> t) ==> t`;;
```

It is also intuitionistically admissible, as are all the other definitions of the logical constants. In fact, they are exactly the definitions given by Prawitz (1965). Note that Henkin (1963) showed how all the logical constants could be derived classically from just a higher order function calculus. If we were really interested in parsimony, every logical operation could be defined, and the turnstile regarded as a logical operator.

As primitive principles of definition and type definition, we use only the core constant of equality, and so avoid any dependence on defined notions like the existential quantifier. The primitive rule of term definition takes a theorem $\vdash P(t)$ for a closed term t and introduces a new constant symbol c and an axiom $\vdash P(c)$. The equational form of definition is trivially derivable using `REFL` and `SYM`, while constant specification also becomes available once we add the selector. Actually, the author favours taking equational definitions as basic and deriving specifications from them; the objection that this makes equalities between constants introduced by the same predicate provable can be evaded simply by throwing away the primitive definition.

The primitive rule of type definition takes an explicit witness for the nonemptiness of the representing predicate, and returns the type bijections directly as unquantified universal theorems. Actually, this is a bit inconsistent with our efforts to get along intuitionistically, since the existence of a total function out of the whole representing type into the newly defined type is a weakly nonconstructive principle. But the intuitionist core is mainly an aesthetic exercise, so this was not considered important.

8 Logical Development

Our changes to the logical organization of the system was motivated by two principles:

- To get as far as convenient without introducing strong axioms; in particular we get a long way without Choice, Excluded Middle or Extensionality. This is largely an intellectual exercise, though one could imagine in the future that someone might want to use HOL constructively.
- To make productive proof techniques (rewriting, tactics) available as early as possible. This is because building up low-level infrastructure is tedious without such techniques.

Accordingly we structured the basic logical theories in the following order:

1. Equality reasoning, including all the conversion combining operators like depth conversions. This is independent of the logic proper, and it's *very* useful to have it available early.

2. Boolean theory: all the basic definitions of logical constants and their introduction and elimination rules (all properly derived, mostly using proforma theorems). All the reasoning is intuitionistic and non-extensional. (Perhaps the name 'Boolean' is no longer appropriate!)

3. A few more sophisticated derived rules, mostly those which use matching, e.g. rewriting and `MATCH_MP_TAC`. A simple form of second-order matching is used, to allow rewriting with schematic theorems. For example, all the quantifier-movement conversions in HOL are now implemented simply as rewrite rules. The term nets used to speed up matching have been appropriately modified, so this makes using these conversions automatically much faster. We also include an intuitionistic tautology prover, based on exhaustive search in Kleene's G3 sequent calculus.

4. Tools for associative-commutative operators, mainly elaborations of `AC_CONV` in the existing HOL systems. These are also useful to have around early, since they deal with some tedious theorems automatically.

5. Tactics and tacticals. We still don't have the classical tactics `BOOL_CASES_TAC`, `ASM_CASES_TAC` and `COND_CASES_TAC`, but all the others are supported, and derived easily.

6. Additional theorems, mostly about quantifier movement and various degenerate cases of the quantifiers and logical operators. The derivations are much easier given the availability of tactics and higher order rewriting.

7. Inductive definitions: general monotone inductive definitions are automated, and most of the monotonicity goals are dealt with automatically. Inductive definitions help in the definition of various theories, especially the natural numbers and free recursive types. And the package uses no advanced logic or theories: we still haven't assumed Choice, Excluded Middle, Extensionality or Infinity!

8. Now at last we throw in the axiom of extensionality (i.e. `ETA_AX`), and the Hilbert Choice operator and its characterizing axiom. Excluded Middle is derived from this, and we implement the few additional new facilities which we can now derive: constant specification, the classical rules, tactics, and theorems, and a classical tautology checker (which works by double negating and calling the intuitionistic one!) We could do a lot more of the theory development intuitionistically, but it would be increasingly tortuous and piecemeal, and our main interests are not constructive.

9. We introduce the Axiom of Infinity and carve out the natural numbers from the postulated Dedekind-infinite set using the inductive definitions package. We also prove the recursion theorem for $\mathbb{N}$ using an inductive definition; this is much shorter than the previous development.

10. We add various other useful theories; pairing, a neat little theory about well-founded relations, and other odds and ends like function composition. We also derive `new_recursive_definition` to automate recursive definitions given the recursion theorem for a concrete recursive type (as yet we only have the natural numbers, of course).

11. We develop natural number arithmetic. This is done much more rationally than before, including the systematic use of primitive recursive definitions not only for theoperators like addition, but also for the predicates. The proofs are mostly short, and all of them are intuitionistically valid except one or two forms of wellfoundedness. We include here a new scheme for numerals.

12. We develop a theory which supports general free recursive types; thanks to inductive definitions we do not need any elaborate trees, but we simply build a large enough set and carve out what we want inductively. Then we write a function to automate the procedure starting from the proforma theorems that this theory provides.

13. Then we have more theories, such as the reals, but now our system is much like the existing implementations.

Now we will look at some of these things in more detail.

9 Higher order matching

In existing HOL implementations, the scope of rewriting, matching modus ponens etc. is rather restricted because only first order matching is done. This means that even quite simple schematic theorems need to be instantiated manually — a very tedious task. For example, if we want to use the theorem:

```
SKOLEM_THM = |- !P. (!x. ?y. P x y) = (?y. !x. P x (y x))
```

to rewrite the term:

```
!n. ?m. m EXP 2 <= n /\ n < (SUC m) EXP 2
```

then simple rewriting won't work; one first needs to instantiate the theorem with:

```
P = (\n m. m EXP 2 <= n /\ n < (SUC m) EXP 2)
```

then beta-reduce it, and only then rewrite with it. As a consequence, lots of quantifier movement operations need to be implemented with specialized conversions instead of just being represented as rewrite rules. There arise quite a lot of other theorems where this kind of schematic matching is enormously convenient. Recently, the author proved the following:

```
BOUNDS_DIVIDED = |- !P. (?B. !n. P n <= B) = (?A B. !n. n * P n <= A * n + B)
```

which is a very useful rewrite rule in some arithmetic goals.

Our implementation of higher order matching is simple, and does not have large coverage. This is partly because of our ignorance and laziness, but the method we have chosen has these properties: it is completely deterministic, it is quick, it has been easily integrated with the term nets used to speed up matching, and it works in most cases one wants (certainly all those exemplified above). Essentially, it will perform a higher order match against $P\ x_1\ \ldots\ x_n$ if and only if P is a free variable (or only bound by outer universal quantifiers), while all the x_i are bound variables in the whole term. This means that there are unambiguous binding instances to identify each x_i, so the process is deterministic and fairly quick.

The term nets have been modified to linearize a term by stripping each combination and starting from the head rator. If this is a variable, then now the whole term is treated as a variable (i.e. as a possible match for anything). Thanks to term nets, there are not many bogus matching attempts, so applying higher order rewrites using depth conversions can actually be dramatically faster than using the corresponding conversion. Note, by the way, that even beta-conversion can be implemented as a higher order rewrite rule, and hence conveniently thrown into a bunch of rewrites instead of being called separately.

```
BETA_THM = |- !f y. (\x. f x) y = f y
```

But note that rewrites with the following theorem go into an infinite loop at any beta-redex because of higher order matching!

```
ETA_AX = |- !t. (\x. t x) = t
```

The author still regularly commits this blunder, so it might be worth adding a hack to look for these tricky cases and force only a first order match. The higher order matching applies pervasively, not just in rewriting, but in other situations like `MATCH_MP_TAC`. At first it was possible to switch it off by setting a global flag, but since we never used it, the option was removed (which we may one day come to regret). Accidental second order rewrites are seldom a problem, and always picking a first order rewrite first has some unfortunate consequences. For example, induction tactics now use `MATCH_MP_TAC` with the induction theorem. If a first order match is picked first, then `INDUCT_TAC` on the goal:

```
!n. n + 1 >= n
```

instantiates the induction predicate to produce the following goal:

```
n + 1 >= 0 /\ (!x. n + 1 >= x ==> (SUC n) + 1 >= x)
```

which obviously isn't what's wanted.

The most significant defect of the current system is that bound variable names are just taken from the rewriting theorem. So when rewriting with `SKOLEM_THM` above, one actually gets:

```
?y. !x. y(x) EXP 2 <= x /\ x < (SUC y(x)) EXP 2
```

It's not hard to think of ad-hoc solutions, but the author has yet to come up with a simple, elegant scheme for dealing with this problem. Note that even comparing the bound variable names on both sides of the equation can be tricky, since in `SKOLEM_THM`, the variable `y` has a different type on one side.

10 Inductive definitions

Inductive definitions are very useful; once one knows about them, they can be seen in all sorts of situations. As we have already remarked, we use them to define the natural numbers and other free inductive types, and to prove the recursion theorem for such types by inductively defining the graph of the recursive function. This approach extends to the wellfounded recursion theorem.

We have implemented quite a general package, which supports mutual recursion, additional schematic arguments, and arbitrary monotone hypotheses (with an automatic proof of most monotonicity goals). It produces theorems giving closure rules, induction and an equational cases theorem. For more details of the implementation, see Harrison (1995). Let us just remark that the complete development uses only some very simple logical principles; it's entirely done without Extensionality, Choice or Excluded Middle.

11 Proving the Law of Excluded Middle

It's a well-known fact that the Law Of Excluded Middle is derivable from some forms of the Axiom of Choice. Tom Melham once raised the question of whether such a derivation could be done in HOL. Thanks to the system reorganization, we (i) can be sure that we aren't unwittingly appealing to LEM in the proof, and (ii) can take advantage of tactics to make the proof easier.

The idea of the proof, from Beeson (1984), is as follows. Let t be any proposition. Consider the sets:

$$A = \{x \mid (x = \bot) \vee (x = \top) \wedge t\}$$

$$B = \{x \mid (x = \bot) \wedge t \vee (x = \top)\}$$

Obviously, since $\bot \in A$ and $\top \in B$, we have

$$\forall s.\ s \in \{A, B\} \Rightarrow \exists y.\ y \in s$$

Now using the choice operator to define $f = \lambda s.\ \varepsilon y.\ y \in s$, the characterizing axiom gives us:

$$\forall s.\ s \in \{A, B\} \Rightarrow f(s) \in s$$

(Note that the selector immediately allows the weaker nonconstructive principle of permuting the existential quantifier and implication — in a constructive logic, selectors need to be partial functions to avoid this.) So:

$$(f(A) = \bot) \vee (f(A) = \top) \wedge t$$

and

$$(f(B) = \bot) \wedge t \vee (f(B) = \top)$$

Now consider the four cases which these theorems give us. Three of them yield t immediately, and the remaining case, $f(A) = \bot$ and $f(B) = \top$, tells us that $A \neq B$ (since f is a *function*). But since $t \Rightarrow (A = B)$, we must have $\neg t$.

```
let EXCLUDED_MIDDLE = prove
 (`!t. t \/ ~t`,
  GEN_TAC THEN SUBGOAL_THEN
   `(((@x. (x = F) \/ (x = T) /\ t) = F) \/
     ((@x. (x = F) \/ (x = T) /\ t) = T) /\ t) /\
    (((@x. (x = T) \/ (x = F) /\ t) = T) \/
     ((@x. (x = T) \/ (x = F) /\ t) = F) /\ t)`
  MP_TAC THENL
   [CONJ_TAC THEN CONV_TAC SELECT_CONV THENL
     [EXISTS_TAC `F`; EXISTS_TAC `T`] THEN
    DISJ1_TAC THEN REFL_TAC; ALL_TAC] THEN
  DISCH_THEN STRIP_ASSUME_TAC THEN
  TRY(DISJ1_TAC THEN FIRST_ASSUM ACCEPT_TAC) THEN
  MP_TAC(ITAUT `~(T = F)`) THEN
  POP_ASSUM_LIST(PURE_ONCE_REWRITE_TAC o map SYM) THEN
  DISCH_THEN(prefix THEN DISJ2_TAC o MP_TAC) THEN
  MATCH_MP_TAC(ITAUT `(a ==> b) ==> ~b ==> ~a`) THEN
  DISCH_THEN(SUBST1_TAC o EQT_INTRO) THEN
  GEN_REWRITE_TAC (RAND_CONV o ONCE_DEPTH_CONV)
   [ITAUT `a \/ (b /\ T) = b \/ (a /\ T)`] THEN
  REFL_TAC);;
```

12 Arithmetic

Our theory of arithmetic is completely new, though quite a lot of the theorems are similar. We have decided to adopt a more rational naming convention (the existing theorem is almost a case study in bad naming). This leads to some considerable incompatibility with previous versions of the system (though a compatibility module would be easy enough to rig up). Furthermore, we have changed some of the definitions, so that everything, including relations, is defined in a uniform recursive style:

```
let PRE = new_recursive_definition num_Axiom
 `(PRE 0 = 0) /\
  (!n. PRE (SUC n) = n)`;;

let ADD = new_recursive_definition num_Axiom
 `(!n. 0 + n = n) /\
  (!m n. (SUC m) + n = SUC(m + n))`;;

let MULT = new_recursive_definition num_Axiom
 `(!n. 0 * n = 0) /\
  (!m n. (SUC m) * n = (m * n) + n)`;;

let EXP = new_recursive_definition num_Axiom
 `(!m. m EXP 0 = 1) /\
  (!m n. m EXP (SUC n) = m * (m EXP n))`;;

let LE = new_recursive_definition num_Axiom
 `(!m. (m <= 0) = (m = 0)) /\
  (!m n. (m <= SUC n) = (m = SUC n) \/ (m <= n))`;;

let LT = new_recursive_definition num_Axiom
 `(!m. (m < 0) = F) /\
  (!m n. (m < SUC n) = (m = n) \/ (m < n))`;;

let GE = new_definition
  `m >= n = n <= m`;;

let GT = new_definition
  `m > n = n < m`;;

let EVEN = new_recursive_definition num_Axiom
  `(EVEN 0 = T) /\
   (!n. EVEN (SUC n) = ~(EVEN n))`;;

let ODD = new_recursive_definition num_Axiom
  `(ODD 0 = F) /\
   (!n. ODD (SUC n) = ~(ODD n))`;;

let SUB = new_recursive_definition num_Axiom
 `(!m. m - 0 = m) /\
  (!m n. m - (SUC n) = PRE(m - n))`;;

let FACT = new_recursive_definition num_Axiom
  `(FACT 0 = 1) /\
   (!n. FACT (SUC n) = (SUC n) * FACT(n))`;;
```

All the proofs, with the exception of one or two versions of wellfoundedness, are intuitionistically and extensionally admissible. This remark has been verified by setting up the real number axioms without the classical stuff loaded in. (Note that in this framework, the uniqueness part of the recursion theorem is not exactly true, though the introduction of recursive functions is still intuitionistically admissible.) In practice avoiding Excluded Middle is almost trivial because induction and cases are still true, so all the familiar theorems like the totality of the ordering still hold. Most theorems can be proved mechanically starting with induction.

13 Numerals

An embarrassment in the existing HOL systems is the status of numerals. They are implemented as an infinite family of constants, with the defining axiom schema $\mathbf{n} + \mathbf{1} = SUC\ \mathbf{n}$ produced by `mk_thm`. We can object to this on grounds of logical correctness, as it builds in a dependency on the native bignum package (`hol90` simply gives up if one tries to go beyond machine arithmetic). Furthermore, it is exceedingly slow to do explicit numerical calculations involving large numbers. We have solved this in our system along the lines of Tim Leonard's numeral library for `HOL88`. Right after defining addition in the arithmetic theory, we make the following two definitions:

```
let BIT0 = new_definition
 `BIT0 n = n + n`;;

let BIT1 = new_definition
 `BIT1 n = SUC(n + n)`;;
```

These allow numbers to be written in binary (least significant bit first), e.g. 13 is $BIT1(BIT0(BIT1(BIT1(0))))$. The parser and printer automatically handle this transformation (of course this is still prone to bugs, but at least it's separated from the logical core). The remaining system is built using this notion of numeral; note that 0 is still a constant, SUC still exists, and `num_CONV` is easily implemented as a derived rule for compatibility.

Arithmetic with these numerals is much, much faster. Five-digit numbers can be multiplied by proof in a few seconds. What's more, most arithmetic operations can be implemented as a set of rewrite rules! Throwing these into a rewrite will automatically do numeral arithmetic. Here are some examples.

```
ARITH_ADD =
|- (0 + 0 = 0) /\
   (!n. 0 + BIT0 n = BIT0 n) /\
   (!n. 0 + BIT1 n = BIT1 n) /\
   (!n. BIT0 n + 0 = BIT0 n) /\
   (!n. BIT1 n + 0 = BIT1 n) /\
   (!m n. BIT0 m + BIT0 n = BIT0 (m + n)) /\
   (!m n. BIT0 m + BIT1 n = BIT1 (m + n)) /\
   (!m n. BIT1 m + BIT0 n = BIT1 (m + n)) /\
   (!m n. BIT1 m + BIT1 n = BIT0 (SUC (m + n)))

ARITH_LE =
|- (0 <= 0 = T) /\
   (!n. BIT0 n <= 0 = (n = 0)) /\
   (!n. BIT1 n <= 0 = F) /\
   (!n. 0 <= BIT0 n = T) /\
   (!n. 0 <= BIT1 n = T) /\
   (!m n. BIT0 m <= BIT0 n = m <= n) /\
   (!m n. BIT0 m <= BIT1 n = m <= n) /\
   (!m n. BIT1 m <= BIT0 n = m < n) /\
   (!m n. BIT1 m <= BIT1 n = m <= n)

ARITH_SUB =
|- (0 - 0 = 0) /\
   (!n. 0 - BIT0 n = 0) /\
   (!n. 0 - BIT1 n = 0) /\
   (!n. BIT0 n - 0 = BIT0 n) /\
   (!n. BIT1 n - 0 = BIT1 n) /\
   (!m n. BIT0 m - BIT0 n = BIT0 (m - n)) /\
   (!m n. BIT0 m - BIT1 n = PRE (BIT0 (m - n))) /\
   (!m n. BIT1 m - BIT0 n = (n <= m => BIT1 (m - n) | 0)) /\
   (!m n. BIT1 m - BIT1 n = BIT0 (m - n))
```

Evidently, rewriting with `ARITH_SUB` is a bit inefficient, since the same condition is tested repeatedly. However this isn't too bad in practice, and if one really wants high performance arithmetic, special conversions along the lines of the existing `reduce` library could be written (for example, the numeral library uses asymptotically faster optimizations for multiplication, a tweak added by the present author). Alternatively, one could set up a separate type of numerals (though this would be less trivial than the above two definitions). This would mean, since every numeral would have the prescribed form of a list of zeros and ones, one could prove general ‘metatheorems’ and so justify the use of some optimized rewrites. For example, one could have an auxiliary subtraction function, valid only when the result is positive, and invoke this after just one comparison of the input numerals. The problem of providing a good set of rewrites for division and remainder has not yet been solved. It seems quite hard without building in non-confluence, but we believe it’s probably possible. Any ideas?

14 The future

This version of HOL is steadily becoming the preferred platform for our own research. This isn’t practical for most HOL users, since we don’t have any of the libraries (except the reals, which nobody else uses anyway), and we don’t have theory files, still less autoloading. The system is meant mainly as a vehicle for research, and a testbed for ideas to put into `hol90`. However it has a certain life of its own (compare PVS and EHDM!) It seems a good basis for learning about LCF-style theorem proving; a good survey article could be based on a fairly low-level walk through the ML code, since it’s relatively clean, though in places pretty incomprehensible. We believe that some of the changes, in particular numerals and higher order matching, should be considered a *sine qua non* for future mainstream HOL versions.

References

Beeson, M. J. (1984) *Foundations of constructive mathematics: metamathematical studies*, Volume 3 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag.

Boulton, R. J. (1993) Efficiency in a fully-expansive theorem prover. Technical Report 337, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK. Author’s PhD thesis.

Gordon, M. J. C. and Melham, T. F. (1993) *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press.

Harrison, J. (1995) Inductive definitions: automation and application. In Windley, P. J., Schubert, T., and Alves-Foss, J. (eds.), *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop*, Volume ??? of *Lecture Notes in Computer Science*, Aspen Grove, Utah, pp. ?–? Springer-Verlag. To appear.

Harrison, J. and Slind, K. (1994) A reference version of HOL. Presented in poster session of 1994 HOL Users Meeting and only published in participants’ supplementary proceedings. Available on the Web from `http://www.dcs.glasgow.ac.uk/~hug94/sproc.html`.

Henkin, L. (1963) A theory of propositional types. *Fundamenta Mathematicae*, **52**, 323–344.

Melham, T. F. (1992) The HOL logic extended with quantification over type variables. In Claesen, L. J. M. and Gordon, M. J. C. (eds.), *Proceedings of the IFIP TC10/WG10.2 International Workshop on Higher Order Logic Theorem Proving and its Applications*, Volume A-20 of *IFIP Transactions A: Computer Science and Technology*, IMEC, Leuven, Belgium, pp. 3–18. North-Holland.

Prawitz, D. (1965) *Natural deduction; a proof-theoretical study*, Volume 3 of *Stockholm Studies in Philosophy.* Almqvist and Wiksells.