The LCF Approach to Theorem Proving

John Harrison

Intel Corporation

- Ideas and historical context
- Key ideas of LCF
- Equational logic example
- More about HOL Light
- Programming example

Key ideas

Despite decades of steady progress in automated theorem proving, there are still difficulties in tackling many real-world problems in mathematics and verification. There's a need to:

- Organize and use large body of knowledge
- Make use both of pure logical deduction and special decision methods
- Have a reasonable level of assurance that proofs are indeed correct

LCF is an approach to the design of theorem proving programs that attempts to satisfy these needs.

Historical context

Most early theorem provers were fully automatic, even though there were several different approaches:

- Pure logical deduction (Gilmore, Wang, Prawitz)
- Special decision methods (Davis)
- Human-oriented AI style approaches (Newell-Simon, Gelerntner)

Interactive theorem proving

The idea of a more ‘interactive’ approach was already anticipated by pioneers, e.g. Wang (1960):

> [...] the writer believes that perhaps machines may more quickly become of practical use in mathematical research, not by proving new theorems, but by formalizing and checking outlines of proofs, say, from textbooks to detailed formalizations more rigorous that *Principia* [Mathematica], from technical papers to textbooks, or from abstracts to technical papers.

Early interactive systems

Some 'interactive' systems were really batch-oriented proof checkers, e.g.

- AUTOMATH (de Bruijn et al.)
- Mizar (Trybulec et al.)
- Stanford LCF (Milner)

Others like the SAM series (Guard et al.) were truly interactive:

> Semi-automated mathematics [...] seeks to combine automatic logic routines [with] human intervention in the form of control and guidance.

Edinburgh LCF

The LCF approach started in the mid/late 1970s with Edinburgh LCF (Milner et al.)

The name LCF comes from the logic implemented in that project, namely Scott's "Logic of Computable Functions"

However, the key LCF approach is applicable to any logic. There are now various LCF descendants, e.g.

- Nuprl (Constable et al., Martin-Löf type theory)
- HOL (Gordon et al., classical higher order logic)
- Coq (Coquand, Huet et al., the Calculus of Constructions)

Basic LCF ideas

The key ideas behind the LCF approach are as follows:

- Have a special abstract type `thm` of ‘theorems’
- Make the constructors of the abstract type the inference rules of the logical system
- Implement the system in a strongly-typed high-level language

The ML family of programming languages (CAML, Objective CAML, Standard ML) are all descended from the programming language designed in the LCF project.

The advantages of LCF

The abstract type supported by strong typing in the implementation language enforces logical correctness: everything of type `thm` has really been proved. Yet proofs do not need to be explicitly generated, still less stored.

The embedding in a full programming language allows the user to implement more sophisticated derived rules that decompose to the primitives, without compromising soundness. Thus, proof can be conducted at a much higher level than in a simple proof checker.

Fully-expansive decision procedures

How can we code sophisticated derived rules that decompose to primitives? At first sight this might seem hopelessly inefficient.

- Represent inference steps as object-level theorems
- Separate search from inference

Many useful decision procedures can be coded in this manner, without unacceptable slowness.

For example, HOL has linear arithmetic, tautology checking and model elimination.

Other things, like explicit arithmetic with very large numbers, or the BDD-based fixpoint calculations in model checking, seem more challenging.

LCF-style prover for equational logic

We start with an abstract type signature:

```
module type Birkhoff =
   sig type thm
       val axiom : formula -> thm
       val inst : (string, term) func -> thm -> thm
       val refl : term -> thm
       val sym : thm -> thm
       val trans : thm -> thm -> thm
       val cong : string -> thm list -> thm
       val dest_thm : thm -> formula list * formula
   end;;
```

This identifies the basic inference rules of equational logic as the only type constructors.

Implementation of primitive rules

The following is the core's implementation:

```
module Proven : Birkhoff =
  struct
    type thm = formula list * formula
    let axiom p =
      match p with
        Atom("=",[s;t]) -> ([p],p)
      | _ -> failwith "axiom: not an equation"
    let inst i (asm,p) = (asm,formsubst i p)
    let refl t = ([],Atom("=",[t;t]))
    let sym (asm,Atom("=",[s;t])) =
          (asm,Atom("=",[t;s]))
    let trans (asm1,Atom("=",[s;t]))
              (asm2,Atom("=",[t';u])) =
      if t' = t then (union asm1 asm2,Atom("=",[s;u]))
      else failwith "trans: theorems don't match up"
    let cong f ths =
      let asms,eqs =
        unzip(map (fun (asm,Atom("=",[s;t]))
             -> asm,(s,t)) ths) in
      let ls,rs = unzip eqs in
      (unions asms,Atom("=",[Fn(f,ls);Fn(f,rs)]))
    let dest_thm th = th
  end;;
```

A simple derived rule

We can implement repeated rewriting at depth:

```
let conclusion th = snd(dest_thm th);;

let rewrite1 eq t =
  match conclusion eq with
    Atom("=",[l;r]) -> inst (term_match l t) eq
  | _ -> failwith "rewrite1";;

let thenc conv1 conv2 t =
  let th1 = conv1 t in
  let th2 = conv2 (rhs(conclusion th1)) in
  trans th1 th2;;

let rec depth fn tm =
  try (thenc fn (depth fn)) tm with Failure _ ->
  match tm with
    Var x -> refl tm
  | Fn(f,args) ->
      let th = cong f (map (depth fn) args) in
      if rhs(conclusion th) = tm then th
      else trans th (depth fn (rhs(conclusion th)));;
```

Similarly, Knuth-Bendix completion etc...

More about HOL Light

HOL Light is a member of the family of provers descended from Mike Gordon's HOL system.

An LCF-style implementation of classical higher-order logic with object-logic polymorphism.

HOL Light is written in CAML Light and is intended to be a cleaner, more rational implementation.

The system includes a number of derived rules for automated theorem proving of various kinds and a reasonable body of pre-proved mathematics.

Used at Intel to formally verify floating-point algorithms.

HOL Light primitive rules (1)

$$\frac{}{\vdash t = t}\ \texttt{REFL}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \texttt{TRANS}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \texttt{MK\_COMB}$$

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x.\, s) = (\lambda x.\, t)}\ \texttt{ABS}$$

$$\frac{}{\vdash (\lambda x.\, t)x = t}\ \texttt{BETA}$$

HOL Light primitive rules (2)

$$\frac{}{\{p\} \vdash p} \; \texttt{ASSUME}$$

$$\frac{\Gamma \vdash p = q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q} \; \texttt{EQ\_MP}$$

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p = q} \; \texttt{DEDUCT\_ANTISYM\_RULE}$$

$$\frac{\Gamma[x_1, \ldots, x_n] \vdash p[x_1, \ldots, x_n]}{\Gamma[t_1, \ldots, t_n] \vdash p[t_1, \ldots, t_n]} \; \texttt{INST}$$

$$\frac{\Gamma[\alpha_1, \ldots, \alpha_n] \vdash p[\alpha_1, \ldots, \alpha_n]}{\Gamma[\gamma_1, \ldots, \gamma_n] \vdash p[\gamma_1, \ldots, \gamma_n]} \; \texttt{INST\_TYPE}$$

Example of programming

In verifying certain floating-point square root algorithms for the Intel Itanium^{TM} processor, we have the following situation:

- We can analytically verify correctness for the vast majority of input numbers
- Certain 'difficult cases' are hard to deal with using the same methods
- The difficult cases can be enumerated as the solution of certain diophantine equations.

Specifically, the equations are all of the form $2^p m = k^2 + d$ for small integers d and fixed p, giving the floating-point mantissa m.

Solving the equations

It's quite easy to program HOL to enumerate all the solutions of such diophantine equations, returning a disjunctive theorem of the form:

$$(2^p m = k^2 + d) \Longrightarrow (m = n_1) \vee \ldots \vee (m = n_i)$$

The procedure simply uses even-odd reasoning and recursion on the power of two (effectively so-called 'Hensel lifting'). For example, if

$$2^{25} m = k^2 - 7$$

then we know k must be odd; we can write $k = 2k' + 1$ and get the derived equation:

$$2^{24} m = 2k'^2 + 2k' - 3$$

And so on by recursion. We can then explicitly verify the algorithm on the solutions, all by proof.

Conclusions

- The LCF approach to theorem proving is a promising way of organizing an interactive theorem prover to combine soundness and programmability.
- The basic approach is applicable to any logic and there are currently several such provers including HOL for classical higher order logic.
- The programmability is often important in verification applications; it would be very difficult to perform these proofs in other kinds of prover.