The LCF approach to proof (Deduction panel)

John Harrison
Intel Corporation

Grand Challenge Verification Workshop 2005

Mon 21st February 2005 (15:30 – 16:30)

Summary

- The LCF approach to proof
- Explicit example
- Derived decision procedures
- Proof styles

LCF

A methodology for making a prover extensible by ordinary users, yet reliable.

Idea due to Milner in Edinburgh LCF project, now used in many other sytems like Coq, HOL, Isabelle and Nuprl.

- Implement in a strongly-typed functional programming language (usually a variant of ML)
- Make `thm` (‘theorem’) an abstract data type with only simple primitive inference rules
- Make the implementation language available for arbitrary extensions.

LCF kernel for first order logic (1)

Define type of first order formulas:

```
type term = Var of string | Fn of string * term list;;

type formula = False
             | True
             | Atom of string * term list
             | Not of formula
             | And of formula * formula
             | Or of formula * formula
             | Imp of formula * formula
             | Iff of formula * formula
             | Forall of string * formula
             | Exists of string * formula;;
```

LCF kernel for first order logic (2)

Define some useful helper functions:

```
let mk_eq s t = Atom("=",[s;t]);;

let rec occurs_in s t =
  s = t or
  match t with
    Var y -> false
  | Fn(f,args) -> exists (occurs_in s) args;;

let rec free_in t fm =
  match fm with
    False -> false
  | True -> false
  | Atom(p,args) -> exists (occurs_in t) args
  | Not(p) -> free_in t p
  | And(p,q) -> free_in t p or free_in t q
  | Or(p,q) -> free_in t p or free_in t q
  | Imp(p,q) -> free_in t p or free_in t q
  | Iff(p,q) -> free_in t p or free_in t q
  | Forall(y,p) -> not (occurs_in (Var y) t) & free_in t p
  | Exists(y,p) -> not (occurs_in (Var y) t) & free_in t p;;
```

LCF kernel for first order logic (3)

```
module type Proofsystem =
   sig type thm
       val axiom_addimp : formula -> formula -> thm
       val axiom_distribimp :
            formula -> formula -> formula -> thm
       val axiom_doubleneg : formula -> thm
       val axiom_allimp : string -> formula -> formula -> thm
       val axiom_impall : string -> formula -> thm
       val axiom_existseq : string -> term -> thm
       val axiom_eqrefl : term -> thm
       val axiom_funcong : string -> term list -> term list -> thm
       val axiom_predcong : string -> term list -> term list -> thm
       val axiom_iffimp1 : formula -> formula -> thm
       val axiom_iffimp2 : formula -> formula -> thm
       val axiom_impiff : formula -> formula -> thm
       val axiom_true : thm
       val axiom_not : formula -> thm
       val axiom_or : formula -> formula -> thm
       val axiom_and : formula -> formula -> thm
       val axiom_exists : string -> formula -> thm
       val modusponens : thm -> thm -> thm
       val gen : string -> thm -> thm
       val concl : thm -> formula
   end;;
```

LCF kernel for first order logic (4)

```
module Proven : Proofsystem =
  struct type thm = formula
         let axiom_addimp p q = Imp(p,Imp(q,p))
         let axiom_distribimp p q r = Imp(Imp(p,Imp(q,r)),Imp(Imp(p,q),Imp(p,r)))
         let axiom_doubleneg p = Imp(Imp(Imp(p,False),False),p)
         let axiom_allimp x p q = Imp(Forall(x,Imp(p,q)),Imp(Forall(x,p),Forall(x,q)))
         let axiom_impall x p =
           if not (free_in (Var x) p) then Imp(p,Forall(x,p)) else failwith "axiom_impall"
         let axiom_existseq x t =
           if not (occurs_in (Var x) t) then Exists(x,mk_eq (Var x) t) else failwith "axiom_existseq"
         let axiom_eqrefl t = mk_eq t t
         let axiom_funcong f lefts rights =
            fold_right2 (fun s t p -> Imp(mk_eq s t,p)) lefts rights (mk_eq (Fn(f,lefts)) (Fn(f,rights)))
         let axiom_predcong p lefts rights =
            fold_right2 (fun s t p -> Imp(mk_eq s t,p)) lefts rights (Imp(Atom(p,lefts),Atom(p,rights)))
         let axiom_iffimp1 p q = Imp(Iff(p,q),Imp(p,q))
         let axiom_iffimp2 p q = Imp(Iff(p,q),Imp(q,p))
         let axiom_impiff p q = Imp(Imp(p,q),Imp(Imp(q,p),Iff(p,q)))
         let axiom_true = Iff(True,Imp(False,False))
         let axiom_not p = Iff(Not p,Imp(p,False))
         let axiom_or p q = Iff(Or(p,q),Not(And(Not(p),Not(q))))
         let axiom_and p q = Iff(And(p,q),Imp(Imp(p,Imp(q,False)),False))
         let axiom_exists x p = Iff(Exists(x,p),Not(Forall(x,Not p)))
         let modusponens pq p =
           match pq with Imp(p',q) when p = p' -> q | _ -> failwith "modusponens"
         let gen x p = Forall(x,p)
         let concl c = c
  end;;
```

Derived rules

The primitive rules are very simple. Using them 'manually' is very tedious.

But using the LCF technique we can build up a set of derived rules. The following derives $p \Rightarrow p$:

```
let imp_refl p = modusponens (modusponens (axiom_distribimp p (Imp(p,p)) p)
                                          (axiom_addimp p (Imp(p,p))))
                             (axiom_addimp p p);;
```

This can be just the start of a tower of more and more powerful derived rules.

Derived decision procedures

How to realize conventional decision procedures as combinations of primitive inferences?

- Quite often, can “naively” translate algorithms from doing ad-hoc term manipulation to producing theorems (e.g. rewriting, Knuth-Bendix completion).
- Other times, the main computational cost is proof search, but there is a certificate that can be separately checked (e.g. conventional first-order proof search, refutations using Gröbner bases).

For tackling the remaining difficult problems, we can consider more exotic techniques like *reflection*.

Proof styles

Directly invoking the primitive or derived rules tends to give proofs that are *procedural*.

A *declarative* style (*what* is to be proved, not *how*) can be nicer:

- Easier to write and understand independent of the prover
- Easier to modify
- Less tied to the details of the prover, hence more portable

Mizar pioneered the declarative style of proof.

Recently, several other declarative proof languages have been developed, as well as declarative shells round existing systems like HOL and Isabelle.

Finding the right style is an interesting research topic.