Formal Verification Methods
3: Theorem Proving

John Harrison
Intel Corporation

Marktoberdorf 2003

Fri 1st August 2003 (11:25 – 12:10)

Summary

- Need for general theorem proving
- Set theory vs. higher-order logic
- Herbrand-based approaches
- Unification
- Decidable problems
- Interactive proof
- LCF
- Proof Style

Need for general theorem proving

Propositional and temporal logic are useful tools for specification and verification, especially in the hardware domain.

However, sometimes we need more general mathematics, e.g. infinite sets, real numbers etc. Consider verifying:

- A floating-point sin function.
- The new AKS polynomial-time primality test.

We need non-trivial number theory, algebra and analysis.

In the case of sin, we need basic real analysis just to say what it's *supposed* to do.

Set theory vs. higher-order logic

Two standard systems give a good general framework for mathematics and computer science:

- First-order set theory (first-order logic with set axioms)
- Higher-order logic (a.k.a. type theory)

For typical applications, it doesn't much matter which is used.

First-order set theory is better-known among mathematicians as the 'standard' foundation for mathematics.

More theorem provers support higher-order logic, since it's slightly simpler and easier to mechanize.

Provers for set and type theory

Popular theorem provers include:

- Coq (type theory)
- EVES (set theory)
- HOL (type theory)
- Mizar (set theory)
- PVS (type theory)

Some provers (e.g. Isabelle) are generic and can support both.

Others (e.g. ACL2) adopt more restrictive logical systems that are easier to automate.

Many (e.g. Otter) support pure first-order logic, but can in principle be used with set-theoretic axioms.

First-order automation

Validity in pure first-order logic is semi-decidable: there is a program that will verify that a formula is valid, but it may loop indefinitely if it is not.

We can reduce the problem to propositional logic using the so-called *Herbrand theorem*:

> Let $\forall x_1, \ldots, x_n.\ P[x_1, \ldots, x_n]$ be a first order formula with only the indicated universal quantifiers (i.e. the body $P[x_1, \ldots, x_n]$ is quantifier-free). Then the formula is satisfiable iff the infinite set of 'ground instances' $P[t_1^i, \ldots, t_n^i]$ that arise by replacing the variables by arbitrary variable-free terms made up from functions and constants in the original formula is *propositionally* satisfiable.

together with Skolemization to eliminate existential quantifiers.

Example

Suppose we want to prove the ‘drinker’s principle’

$$\exists x.\, \forall y.\, D(x) \Rightarrow D(y)$$

Negate the formula, and prove negation unsatisfiable:

$$\neg(\exists x.\, \forall y.\, D(x) \Rightarrow D(y))$$

Convert to prenex normal form: $\forall x.\, \exists y.\, D(x) \wedge \neg D(y)$

Skolemize: $\forall x.\, D(x) \wedge \neg D(f(x))$

Enumerate set of ground instances, first $D(c) \wedge \neg D(f(c))$ is not unsatisfiable, but the next is:

$$(D(c) \wedge \neg D(f(c))) \wedge (D(f(c)) \wedge \neg D(f(f(c))))$$

Unification

The first automated theorem provers actually used that approach.

It was to test the propositional formulas resulting from the set of ground-instances that the Davis-Putnam method was developed.

However, more efficient than enumerating ground instances is to use *unification* to choose instantiations intelligently.

Many theorem-proving algorithms based on unification exist:

- Tableaux
- Resolution
- Model elimination
- Connection method
- ...

Decidable problems

Although first order validity is undecidable, there are special cases where it is decidable, e.g.

- AE formulas: no function symbols, universal quantifiers before existentials in prenex form.
- Monadic formulas: no function symbols, only unary predicates

These are not particularly useful in practice, though they can be used to automate syllogistic reasoning.

If all M are P, and all S are M, then all S are P

can be expressed as the monadic formula:

$$(\forall x.\, M(x) \Rightarrow P(x)) \wedge (\forall x.\, S(x) \Rightarrow M(x)) \Rightarrow (\forall x.\, S(x) \Rightarrow P(x))$$

Decidable theories

More useful in practical applications are cases not of *pure* validity, but validity in special models, or consequence from useful axioms, e.g.

- Presburger arithmetic: arithmetic equations and inequalities with addition but *not multiplication*, interpreted over $\mathbb{Z}$.
  $\forall x\ y.\ x < y \Rightarrow 2x + 1 < 2y$
- Tarski arithmetic: arithmetic equations and inequalities with addition and multiplication, interpreted over $\mathbb{R}$.
  $\forall x_1\ x_2\ y_1\ y_2.\ (x_1 \cdot y_1 + x_2 \cdot y_2)^2 \leq (x_1^2 + x_2^2) \cdot (y_1^2 + y_2^2)$

However, arithmetic with multiplication over $\mathbb{Z}$ is not even semidecidable, by Gödel's theorem.

Nor is arithmetic over $\mathbb{Q}$ (Julia Robinson), nor just solvability of equations over $\mathbb{Z}$ (Matiyasevich). Equations over $\mathbb{Q}$ unknown.

Interactive theorem proving

In practice, most interesting problems can't be automated completely:

- They don't fall in a practical decidable subset
- Pure first order proof search is not a feasible approach

In practice, we need an interactive arrangement, where the user and machine work together.

The user can delegate simple subtasks to pure first order proof search or one of the decidable subsets.

However, at the high level, the user must guide the prover.

In order to provide custom automation, the prover should be *programmable* — without compromising logical soundness.

LCF

One successful solution was pioneered in Edinburgh LCF ('Logic of Computable Functions').

The same 'LCF approach' has been used for many other theorem provers.

- 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.

Gives a good combination of extensibility and reliability.

Now used in Coq, HOL, Isabelle and several other systems.

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. 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);;
```

While this process is tedious at the beginning, we can quickly reach the stage of automatic derived rules that

- Prove propositional tautologies
- Perform Knuth-Bendix completion
- Prove first order formulas by standard proof search and translation

Real LCF-style theorem provers like HOL have many powerful derived rules.

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.

Procedural proof example

```
REPEAT GEN_TAC THEN REWRITE_TAC[contl; LIM; REAL_SUB_RZERO] THEN
BETA_TAC THEN DISCH_TAC THEN X_GEN_TAC "e:real" THEN
DISCH_TAC THEN
FIRST_ASSUM(UNDISCH_TAC o assert is_conj o concl) THEN
DISCH_THEN(CONJUNCTS_THEN MP_TAC) THEN
DISCH_THEN(\th. FIRST_ASSUM(MP_TAC o MATCH_MP th)) THEN
DISCH_THEN(X_CHOOSE_THEN "d:real" STRIP_ASSUME_TAC) THEN
DISCH_THEN(MP_TAC o SPEC "d:real") THEN ASM_REWRITE_TAC[] THEN
DISCH_THEN(X_CHOOSE_THEN "c:real" STRIP_ASSUME_TAC) THEN
EXISTS_TAC "c:real" THEN ASM_REWRITE_TAC[] THEN
X_GEN_TAC "h:real" THEN DISCH_THEN(ANTE_RES_THEN MP_TAC) THEN
ASM_CASES_TAC "&0 < abs(f(x + h) - f(x))" THENL
[UNDISCH_TAC "&0 < abs(f(x + h) - f(x))" THEN
 DISCH_THEN(\th. DISCH_THEN(MP_TAC o CONJ th)) THEN
 DISCH_THEN(ANTE_RES_THEN MP_TAC) THEN
 REWRITE_TAC[REAL_SUB_ADD2];
 UNDISCH_TAC "~(&0 < abs(f(x + h) - f(x)))" THEN
 REWRITE_TAC[GSYM ABS_NZ; REAL_SUB_0] THEN
 DISCH_THEN SUBST1_TAC THEN
 ASM_REWRITE_TAC[REAL_SUB_REFL; ABS_0]]);;
```

Declarative proof example

```
let f be A->A;
assume L:antecedent;
antisymmetry: (!x y. x <= y /\ y <= x ==> (x = y)) by L;
transitivity: (!x y z. x <= y /\ y <= z ==> x <= z) by L;
monotonicity: (!x y. x <= y ==> f x <= f y) by L;
least_upper_bound:
    (!X. ?s:A. (!x. x IN X ==> s <= x) /\
               (!s'. (!x. x IN X ==> s' <= x) ==> s' <= s)) by L;
set Y_def: Y = {b | f b <= b};
Y_thm: !b. b IN Y = f b <= b by Y_def,IN_ELIM_THM,BETA_THM;
consider a such that
    lub: (!x. x IN Y ==> a <= x) /\
         (!a'. (!x. x IN Y ==> a' <= x) ==> a' <= a)
    by least_upper_bound;
take a;
now let b be A;
    assume b_in_Y: b IN Y;
    then L0: f b <= b by Y_thm;
    a <= b by b_in_Y, lub;
    so f a <= f b by monotonicity;
    hence f a <= b by L0, transitivity;
    end;
so Part1: f(a) <= a by lub;
so f(f(a)) <= f(a) by monotonicity;
so f(a) IN Y by Y_thm;
so a <= f(a) by lub;
hence thesis by Part1, antisymmetry;
```

Summary

- We need general theorem proving for some applications; it can be based on first order set theory or higher-order logic.
- We can semi-automate pure first order logic via Herbrand's theorem, usually with unification.
- Some interesting classes of problems, especially in arithmetic, are decidable.
- In practice, we need a combination of interaction and automation for difficult proofs.
- LCF gives a good way of realizing a combination of soundness and extensibility.
- Different proof styles may be preferable, and they can be supported on top of an LCF-style core.