Implementing a Simple Theorem Prover

Specification and Verification with Higher-Order Logic

Arnd Poetzsch-Heffter
(Slides by Jens Brandt)

Software Technology Group
Fachbereich Informatik
Technische Universität Kaiserslautern

Sommersemester 2008

Outline

Overview

Theorem Prover

- theorem prover implements a proof system
- used for proof checking and automated theorem proving

Goals

- build your own theorem prover for propositional logic
- understanding the fundamental structure of a theorem prover

Outline

Data Types

Data Types of a Theorem Prover

- formulas, terms and types
- axioms and theorems
- deduction rules
- proofs

Formulas, Terms and Types

Propositional Logic

- each term is a formula
- each term has the type $\mathbb{B}$

Data Type Definition

datatype Term =
Variable **of** string |
Constant **of** bool |
Negation **of** Term |
Conjunction **of** Term $*$ Term |
Disjunction **of** Term $*$ Term |
Implication **of** Term $*$ Term;

Syntactical Operations on Terms

Determining the Topmost Operator

```
fun isVar (Variable x) = true
  | isVar _ = false;
fun isConst (Constant b) = true
  | isConst _ = false;
fun isNeg (Negation t1) = true
  | isNeg _ = false;
fun isCon (Conjunction (t1,t2)) = true
  | isCon _ = false;
fun isDis (Disjunction (t1,t2)) = true
  | isDis _ = false;
fun isImp (Implication (t1,t2)) = true
  | isImp _ = false;
```

Syntactical Operations on Terms

Composition

- combine several subterms with an operator to a new one

Composition of Terms

```
fun mkVar s1 = Variable s1;
fun mkConst b1 = Constant b1;
fun mkNeg t1 = Negation t1;
fun mkCon (t1,t2) = Conjunction(t1,t2 );
fun mkDis (t1,t2) = Disjunction(t1 , t2 );
fun mkImp (t1,t2) = Implication(t1 , t2 );
```

Syntactical Operations on Terms

Decomposition

- decompose a term

Decomposition of Terms

```
exception SyntaxError;

fun destNeg (Negation t1) = t1
  | destNeg _ = raise SyntaxError ;
fun destCon (Conjunction (t1,t2)) = (t1 ,t2)
  | destCon _ = raise SyntaxError ;
fun destDis (Disjunction (t1 ,t2)) = (t1 ,t2)
  | destDis _ = raise SyntaxError ;
fun destImp (Implication (t1 ,t2)) = (t1 ,t2)
  | destImp _ = raise SyntaxError ;
```

Term Examples

Example (Terms)

- $t_1 = a \wedge b \vee \neg c$;
- $t_2 = \text{true} \wedge (x \wedge y) \vee \neg z$;
- $t_3 = \neg((a \vee b) \wedge \neg c)$

```
val t1 = Disjunction(Conjunction(Variable "a", Variable "b"),
                     Negation(Variable "c") );
val t2 = Disjunction(
           Conjunction( Constant true,
                        Conjunction (Variable "x", Variable "y" )),
                          Negation(Variable "z" ));
val t3 = Negation(Conjunction(
             Disjunction(Variable "a", Variable "b"),
             Negation(Variable "c" )));
```

Theorems

Data Type Definition

```
datatype Theorem =
    Theorem of Term list * Term;
```

Syntactical Operations

```
fun assumptions (Theorem (assums,concl)) = assums;
fun conclusion (Theorem (assums,concl)) = concl;
fun mkTheorem(assums,concl) = Theorem(assums,concl);
fun destTheorem (Theorem (assums,concl)) = (assums,concl);
```

Rules

Data Type Definition

datatype Rule =
    Rule **of** Theorem list $*$ Theorem;

Application of Rules

Application of Rules

- form a new theorem from several other theorems

Application (Version 1)

```
exception DeductionError;

fun applyRule rule thms =
    let
        val Rule (prem,concl) = rule
    in
        if prem=thms then concl else raise DeductionError end;
```

Application of Rules

Application of Rules

- premises and given theorems do not need to be identical
- premises only need to be in the given theorems

Application (Version 2)

```
fun mem x [] = false
  | mem x (h::t) = (x=h) orelse (contains t x);
fun sublist [] l2 = true
  | sublist (h1::t1) l2 = (contains l2 h1) andalso (sublist t1 l2);
fun applyRule rule thms =
      let
          val Rule (prem,concl) = rule
      in
          if sublist prem thms then concl else raise DeductionError end;
```

Application of Rules

Example (Rule Application)

```
val axiom1 = Theorem( [], (Variable "a" ));
val axiom2 = Theorem( [], Implication((Variable "a"),( Variable "b" )));
val axiom3 = Theorem( [], Implication((Variable "b"),( Variable "c" )));

val modusPonens =
    Rule(
        [Theorem( [], Implication((Variable "a"),( Variable "b")) ),
         Theorem( [], (Variable "a") )]
    ,
        Theorem( [], (Variable "b") )
    );
```

Application of Rules

Example (Rule Application)

```
val thm1 = applyRule modusPonens [axiom1,axiom2];
val thm2 = applyRule modusPonens [thm1,axiom3];
```

Problem

- axioms and rules should work for arbitrary variables
- axiom scheme, rule scheme
- definition of substitution and unification needed

Outline

Support Functions

Support Functions

```
fun insert x l = if mem x l then l else x :: l;

fun assoc [] a = NONE
  | assoc ((x,y):: t) a = if (x=a) then SOME y else assoc t a;

fun occurs v (w as Variable _) = (v=w)
  | occurs v (Constant b) = false
  | occurs v (Negation t) = occurs v t
  | occurs v (Conjunction (t1,t2)) = occurs v t1 orelse occurs v t2
  | occurs v (Disjunction (t1,t2)) = occurs v t1 orelse occurs v t2
  | occurs v (Implication (t1,t2)) = occurs v t1 orelse occurs v t2;
```

Substitution

Substitution

```
fun subst theta (v as Variable _) =
                (case assoc theta v of NONE => v | SOME w => w)
| subst theta (Constant b) = Constant b
| subst theta (Negation t) = Negation(subst theta t)
| subst theta (Conjunction (t1,t2)) =
                Conjunction(subst theta t1, subst theta t2)
| subst theta (Disjunction (t1,t2)) =
                Disjunction(subst theta t1, subst theta t2)
| subst theta (Implication (t1,t2)) =
                Implication(subst theta t1, subst theta t2);
```

Substitution

Example (Substitution)

```
val theta1 = [(Variable "a",Variable "b"),(Variable "b",Constant true)];
```

Unification

Definition (Matching)

A term matches another if the latter can be obtained by instantiating the former.

$$\text{matches}(M, N) \Leftrightarrow \exists\theta.\text{subst}(\theta, M) = N$$

Definition (Unifier, Unifiability)

A substitution is a *unifier* of two terms, if it makes them equal.

$$\text{unifier}(\theta, M, N) \Leftrightarrow \text{subst}(\theta, M) = \text{subst}(\theta, N)$$

Two terms are unifiable if they have a unifier.

$$\text{unifiable}(M, N) \Leftrightarrow \exists\theta.\text{unifier}(\theta, M, N)$$

Unification Algorithm

General Idea

- traverse two terms in exactly the same way
- eliminating as much common structure as possible
- things actually happen when a variable is encountered (in either term)
- when a variable is encoutered, make a binding with the corresponding subterm in the other term, and substitute through
- important: making a binding (x, M) where x occurs in M must be disallowed since the resulting substitution will not be a unifier

Unification Algorithm

Unification

```
exception UnificationException;

fun unify1 [] [] theta = theta
  | unify1 ((v as Variable _)::L) (M::R) theta =
      if v=M then unify1 L R theta
      else if occurs v M then raise UnificationException
      else unify1 (map (subst [(v,M)]) L)
                  (map (subst [(v,M)]) R)
                  (combineSubst [(v,M)] theta)
    | unify1 L1 (L2 as (Variable _::_)) theta = unify1 L2 L1 theta
    ...
```

Unification Algorithm

Unification

```
    ...
  | unifyl (Negation tl :: L) (Negation tr :: R) theta =
                unifyl (tl :: L) (tr :: R) theta
  | unifyl (Conjunction (tl1, tl2) :: L) (Conjunction (tr1, tr2) :: R) theta =
                unifyl (tl1 :: tl2 :: L) (tr1 :: tr2 :: R) theta
  | unifyl (Disjunction (tl1, tl2) :: L) (Disjunction (tr1, tr2) :: R) theta =
                unifyl (tl1 :: tl2 :: L) (tr1 :: tr2 :: R) theta
  | unifyl (Implication (tl1, tl2) :: L) (Implication (tr1, tr2) :: R) theta =
                unifyl (tl1 :: tl2 :: L) (tr1 :: tr2 :: R) theta
  | unifyl _ _ _ = raise UnificationException;

fun unify M N = unifyl [M] [N] [];
```

Combining Substitutions

Combining Substitutions

```
fun combineSubst theta sigma =
  let val (dsigma,rsigma) = ListPair.unzip sigma
      val sigma1 = ListPair.zip(dsigma,(map (subst theta) rsigma))
      val sigma2 = List. filter (op <>) sigma1
      val theta1 = List. filter (fn (a,_) => not (mem a dsigma)) theta
  in
     sigma2 @ theta1
  end;
```

Outline

Summary

Theorem-Prover Kernel

- terms
- theorems
- rules
- substitution
- unification

Outlook

- theorem-prover architecture
- user interface
- real-world example