Forward vs. Backward Theorem Proving Tactics

Forward vs. Backward Theorem Proving

- The state of a prover can be expressed as:
  $H_1 \wedge \ldots \wedge H_n \Rightarrow^? G$
  - Given the hypotheses H_i try to derive goal G
  - Written also as $[\, H_1, \ldots, H_n] \Rightarrow^? G$
- A forward theorem prover derives new hypotheses, in hope of deriving G
  - If $H_1 \wedge \ldots \wedge H_n \Rightarrow H$ then move to state $[\, H_1, \ldots, H_n, H \,] \Rightarrow^? G$
  - Success state: $[\, H_1, \ldots, G, \ldots H_n \,] \Rightarrow^? G$
- A forward theorem prover uses heuristics to reach G
  - Or it can exhaustively derive everything that is derivable !

Forward Chaining

- Consider a theory with proof rule
  $\forall x.\ A_1 \wedge \ldots \wedge A_m \Rightarrow C$

Use this rule for forward chaining

1. in state $[\, H_1, \ldots, H_n \,] \Rightarrow^? G$
2. Find a substitution Φ
3. such that for all i = 1, …, m exists j. $\Phi(A_i) = H_j$
4. Then move to state $[\, H_1, \ldots, H_n, \Phi(C) \,] \Rightarrow^? G$

Example of Forward Chaining

- Consider the axiom
  $\forall x.\ a(x) \wedge b(x, y) \Rightarrow c(x)$
- and state
  $\ldots, a(t), \ldots, b(t, t'), \ldots \Rightarrow^? G$
- move to state
  $\ldots, a(t), \ldots, b(t, t'), \ldots, c(t) \Rightarrow^? G$
- In general a rule $\forall x.\ A_1 \wedge \ldots, \wedge A_m \Rightarrow C$ works for forward chaining if $Var(C) \subseteq \cup_{i=1,m} Var(A_i)$

Backward Theorem Proving

- A backward theorem prover derives new subgoals from the goal
  - The current state is $[\, H_1, \ldots, H_n] \Rightarrow^? G$
  - If $H_1 \wedge \ldots \wedge H_n \wedge G_1 \wedge \ldots \wedge G_n \Rightarrow G$ (G_i are subgoals)
  - Produce "n" new states (all must lead to success):
    $[\, H_1, \ldots, H_n \,] \Rightarrow^? G_i$
- Prolog works like this

Backward Chaining

- Consider a theory with proof rule
  $\forall x.\ A_1 \wedge \ldots \wedge A_m \Rightarrow C$

Use this rule for backward chaining

1. in state $[\, H_1, \ldots, H_n \,] \Rightarrow^? G$
2. Find a substitution Φ
3. such that $\Phi(C) = G$
4. for all i = 1, …, m
   - Solve the state $[\, H_1, \ldots, H_n \,] \Rightarrow^? \Phi(A_i)$

Example of Backward Chaining

- Consider the axiom

  $\forall x.\ a(x) \wedge b(x) \Rightarrow c(x, y)$
- In state

  $[\ \ldots\ ] \Rightarrow^? c(t, t')$
- move to states

  $[\ \ldots\ ] \Rightarrow^? a(t)$ and $[\ \ldots\ ] \Rightarrow^? b(t)$

- In general a rule $\forall x.\ A_1 \wedge \ldots \wedge A_m \Rightarrow C$ works for forward chaining if $\cup_{i=1,m} Var(A_i) \subseteq Var(C)$

Programming Theorem Provers

- Backward theorem provers most often use heuristics
- If it useful to be able to program the heuristics

- Such programs are called <u>tactics</u> and tactic-based provers have this capability
  - E.g. the Edinburgh LCF was a tactic based prover whose programming language was called the Meta-Language (ML)

- A tactic examines the state and either:
  - Announces that it is not applicable in the current state, or
  - Modifies the proving state

Programming Theorem Provers. Tactics.

- State = Formula list × Formula
  - A set of hypotheses and a goal
- A tactic given a state has three possible outcomes
  - Success: proves the goal
  - Change: makes some changes to the state
  - Fail: cannot prove the goal, or make changes to the state

- Write the tactic in continuation-passing style

  Tactic = State $\rightarrow$ (proof $\rightarrow \alpha$) $\rightarrow$ (State $\rightarrow \alpha$) $\rightarrow$ (unit $\rightarrow \alpha$) $\rightarrow \alpha$

Congruence Closure as a Tactic

- Example: a congruence-closure based tactic

```
cc (h, false) s c f =
    if contradiction detected then
        s proof_of_false
    else
      let e1, …, en new equalities in the congruence closure of h
      c (h ∪ {e1, …, en}, false)
    else (* no new equalities *)
      f ()
```

  - A forward chaining tactic (also called a <u>rewriting</u> step)

Programming Theorem Provers. Tactics.

- Consider an axiom: $\forall x.\ a(x) \Rightarrow b(x)$
  - Like the clause b(x) :- a(x) in Prolog

- This could be turned into a tactic

```
clause (h, g) s c f =
        if unif(g, b) = φ then
            c (h, φ(a))
        else
            f ()
```

  - A backward chaining tactic

Programming Theorem Provers. Tacticals.

- Tactics can be composed using <u>tacticals</u>

Examples:

- THEN : tactic $\rightarrow$ tactic $\rightarrow$ tactic

  THEN t_1 t_2 = $\lambda h.\lambda s.\lambda c.\lambda f.\ t_1\ h\ s\ (\lambda h'.\ t_2\ h'\ s\ c\ f)\ f$
- ORELSE : tactic $\rightarrow$ tactic $\rightarrow$ tactic

  ORELSE t_1 t_2 = $\lambda h.\lambda s.\lambda c.\lambda f.\ t_1\ h\ s\ c\ (\lambda h'.t_2\ h'\ s\ c\ f)$
- BOTH: tactic $\rightarrow$ tactic $\rightarrow$ tactic
  - BOTH t_1 t_2 = $\lambda h.\lambda s.\lambda c.\lambda f.\ t_1\ h\ (\lambda\_.\ t_2\ h\ s\ c\ f)\ c\ f$

- REPEAT : tactic $\rightarrow$ tactic

  REPEAT t = THEN t (REPEAT t)

Programming Theorem Provers. Tacticals

- Prolog is just one possible tactic:
  - Given backwards tactics for each clause: $c_1, \ldots, c_n$
  - Prolog : tactic
    Prolog = REPEAT (c_1 ORLESE c_2 ORELSE … ORELSE c_n)
  - clauses themselves can invoke Prolog on the subgoals
- This is a very powerful mechanism for semi-automatic theorem proving
  - Used in: Isabelle, HOL, Coq, and many others

Adding Tactical Support to Nelson-Oppen

Recall Nelson-Oppen

- The state consists of a set of literals, goal is false
  $[L_1, \ldots, L_n] \Rightarrow^? \text{false}$
- Nelson-Oppen is a forward theorem prover:
  - The state is $[L_1, \ldots, L_n] \Rightarrow^? \text{false}$
  - If $L_1 \wedge \ldots \wedge L_n \Rightarrow E$ (an equality) then
  - New state is $[L_1 , \ldots , L_n , E] \Rightarrow^? \text{false}$ (add the equality)
  - Success state is $[L_1 , \ldots , \underline{L} , \ldots , \underline{\neg L} , \ldots L_n] \Rightarrow^? \text{false}$
- Nelson-Oppen provers exhaustively produce all derivable facts hoping to encounter the goal

Nelson-Oppen as a Tactical

- Assume that each sat. proc. is a tactic
  sat_i: tactic
  sat_i (h, false) s c f either
  - Calls s with a proof of false (proved the goal), or
  - Extends the set of literals with equalities, and calls c, or
  - Calls f
- Nelson-Oppen : keep extending the set of literals until Contradiction, or no more equalities are possible

no (satlist : tactic list) : tactic =
    REPEAT (ORELSE_LIST satlist)

Nelson-Oppen and Non-Convex Theories

- Recall, in a non-convex theory:
  - No contradiction is discovered
  - No single equality is discovered
  - But a disjunction of equalities is discovered
- Many theories are non-convex
  - Theory of sel/upd
    $\text{true} \Rightarrow x = y \vee \text{sel}(\text{upd}(m,x,v),y) = \text{sel}(m,y)$
- How do we handle such theories with Nelson-Oppen ?

Nelson-Oppen and Non-Convex Theories

- Consider the state $[L_1, \ldots, L_n] \Rightarrow^? \text{false}$
- and $L_1 \wedge \ldots \wedge L_n \Rightarrow E_1 \vee E_2$
- We add to the state of Nelson-Oppen disjunctions of equalities, not just literals
  - Most sat. proc. work as before (ignore disjunctions)
  - Non-convex sat. proc. add disjunctions
- We have a new module Case that processes disjunctions
  - After there is nothing else to do

The Case Analysis Tactic

- Define the Case tactic

Case (h, false) s c f =
  if no disjunctions in h then f ()
  elseif $L \in h$ and $\neg L \vee L' \in h$ then c ($h \cup \{ L' \}$, false)
  else pick $L \vee L' \in h$;
    c ($h \cup \{ L \}$, false) ;
    c ($h \cup \{ L' \}$, false)

- Case splitting for Nelson-Oppen is useful
  - for non-convex theories
  - for adding backwards chaining sat. procs.

Recall: Nelson-Oppen with Proof Generation

NO: (pair of L:f and pf L) list $\rightarrow$ pf false

NO F =
  match asat(F), bsat(F) with
   | Contra d, _ -> d
   | _, Contra d -> d
   | Eq (x =y, d), _ -> NO (F $\cup$ { (x = y, d) })
   | _, Eq (x = y, d) -> NO (F $\cup$ { (x = y, d) })
   | Sat, Sat -> raise NoProof

- With the following properties:
  - If asat F = Contra d, then d : pf false
  - If asat F = Eq (x = y, d) , then d : pf (eq x y)

Propagating Disjunctions of Equalities

- To propagate disjunctions we perform a case split:
- If a disjunction of equalities $E_1 \vee E_2$ is discovered:
  - Must try to derive a contradiction for each E_i assumption !

NO F =
  match asat(F), bsat(F) with
   ...
   | Disj ($x_1 = y_1 \vee x_2 = y_2$, d) $\rightarrow$
     let $d_1 = \lambda h_1$: pf (eq x_1 y_1). NO (F $\cup$ { ($x_1 = y_1$, h_1) })
     let $d_2 = \lambda h_2$: pf (eq x_2 y_2). NO (F $\cup$ { ($x_2 = y_2$, h_2) })
     ore d d_1 d_2
 with ore: Π A,B,C:f. pf (or A B) $\rightarrow$
     (pf A $\rightarrow$ pf C) $\rightarrow$ (pf B $\rightarrow$ pf C) $\rightarrow$ pf C

Handling Non-Convex Theories

- Case splitting is expensive
  - Must backtrack (performance --)
  - Must implement all satisfiability procedures in incremental fashion (simplicity --)

- In some cases the splitting can be prohibitive:
  - Take pointers for example.

upd(upd(…(upd(m, i_1, x), …, i_{n-1}, x), i_n, x) =
  upd(…(upd(m, j_1, x), …, j_{n-1}, x) $\wedge$
sel(m, i_1) $\neq$ x $\wedge$ … $\wedge$ sel(m, i_n) $\neq$ x
entails $\bigvee_{j \neq k} i_j \neq i_k$
(a conjunction of length n entails n^2 disjuncts)