Forward vs. Backward Theorem Proving

Tactics

Forward vs. Backward Theorem Proving

- The state of a prover can be expressed as:
  \[ H_1 \land \ldots \land H_n \Rightarrow \neg G \]
  - Given the hypotheses, try to derive goal \( G \)
  - Written also as \( [H_1, \ldots, H_n] \Rightarrow \neg G \)
- A forward theorem prover derives new hypotheses, in hope of deriving \( G \)
  - If \( H_1 \land \ldots \land H_n \Rightarrow H \) then
    move to state \( [H_1, \ldots, H_n, H] \Rightarrow \neg G \)
- 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
  \[ \forall x. A_1 \land \ldots \land A_m \Rightarrow C \]

Use this rule for forward chaining

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

Example of Forward Chaining

- Consider the axiom
  \[ \forall x. a(x) \land b(x, y) \Rightarrow c(x) \]

- and state
  \[ \ldots, a(t), \ldots, b(t, t'), \ldots \Rightarrow \neg G \]
- move to state
  \[ \ldots, a(t), \ldots, b(t, t'), \ldots, c(t) \Rightarrow \neg G \]
- In general a rule \( \forall x. A_1 \land \ldots \land A_m \Rightarrow C \) works for forward chaining if \( \text{Var}(C) \subseteq \cup_{i=1,m} \text{Var}(A_i) \)

Backward Theorem Proving

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

Backward Chaining

- Consider a theory with proof rule
  \[ \forall x. A_1 \land \ldots \land A_m \Rightarrow C \]

Use this rule for backward chaining

1. in state \( [H_1, \ldots, H_n] \Rightarrow \neg G \)
2. Find a substitution \( \Phi \)
3. such that \( \Phi(C) \Rightarrow G \)
4. for all \( i = 1, \ldots, m \)
  - Solve the state \( [H_1, \ldots, H_n] \Rightarrow \neg \Phi(A_i) \)
Example of Backward Chaining

- Consider the axiom $\forall x. a(x) \land b(x) \Rightarrow c(x, y)$
- In state $[\ldots] \Rightarrow q(t, 1)$
- move to states $[\ldots] \Rightarrow q(t)$ and $[\ldots] \Rightarrow b(t)$
- In general a rule $\forall x. A_1 \land \ldots \land A_n \Rightarrow C$ works for forward chaining if $\cup_{i \leq n} \text{Var}(A_i) \subseteq \text{Var}(C)$

Programming Theorem Provers

- Backward theorem provers most often use heuristics
- If it useful to be able to program the heuristics
- Such programs are called tactics 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
  $\text{Tactic} = \text{State} \rightarrow (\text{proof} \rightarrow a) \rightarrow (\text{State} \rightarrow a) \rightarrow (\text{unit} \rightarrow a) \rightarrow a$

Congruence Closure as a Tactic

- Example: a congruence-closure based tactic
  $cc(h, false) s c f =$
    if contradiction detected then
      $a$ proof_of_false
    else
      let $e_1, \ldots, e_n$ new equalities in the congruence closure of $h$
      $c(h \cup \{e_1, \ldots, e_n\}, false)$
    else (* no new equalities *)
      $f()$
  - A forward chaining tactic (also called a rewriting 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
  $\text{clause} = \text{h} = g \Rightarrow \text{if unif}(g, b) = \text{then}$
  $c(h, s())$
  $\text{else}$
  $f()$
  - A backward chaining tactic

Programming Theorem Provers. Tacticals.

- Tactics can be composed using tacticals
  Examples:
  - THEN : tactic → tactic → tactic
    THEN $t_1, t_2 = \lambda h. s \cdot f. t_1 , h \cdot s \cdot f \cdot c \cdot f$
  - ORELSE : tactic → tactic
    ORELSE $t_1, t_2 = \lambda h. s \cdot f. t_1 , h \cdot c \cdot f \cdot c$
  - BOTH : tactic → tactic
    BOTH $t_1, t_2 = \lambda h. s \cdot f. t_1 , h \cdot t_2 , h \cdot s \cdot f \cdot c$
  - REPEAT : tactic → tactic
    REPEAT $t = \text{THEN} (\text{REPEAT} t)$
Programming Theorem Provers. Tactics

- Prolog is just one possible tactic:
  - Given backwards tactics for each clause: c₁, ..., cₖ
  - Prolog : tactic
  Prolog = REPEAT (c₁ ORELSE c₂ ORELSE ... ORELSE cₖ)
  - 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
  \[ L₀ \land \cdots \land Lₙ \Rightarrow \text{false} \]

- Nelson-Oppen is a forward theorem prover:
  - The state is \( \{ L₀, \ldots, Lₙ \} \Rightarrow \text{false} \)
  - If \( L₀ \land \cdots \land Lₙ \Rightarrow E \) (an equality) then
  - New state is \( \{ L₀, \ldots, Lₙ, E \} \Rightarrow \text{false} \) (add the equality)
  - Success state is \( \{ L₀, \ldots, Lₙ \Rightarrow \text{false} \) 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: tactic
  sat \( \circ \) tactic
  \[ \text{no \( (\text{satlist} \circ \text{tactic list}) \)} : \text{tactic} = \text{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 \text{sol/qid}
    \[ \text{true} = x \land y \Rightarrow \text{sol}(\text{upd}(m,x),y) = \text{sol}(m,y) \]

- How do we handle such theories with Nelson-Oppen ?

Nelson-Oppen and Non-Convex Theories

- Consider the state \( \{ L₀, \ldots, Lₙ \} \Rightarrow \text{false} \)
  and \( L₀ \land \cdots \land Lₙ \Rightarrow E₁ \lor E₂ \)

- 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 \text{Case} that processes disjunctions
  - After there is nothing else to do
The Case Analysis Tactic

- Define the Case tactic
  Case (h, false) ≜ c f =
  if no disjunctions in h then f()
  elseif L ∈ h and L ∨ L' ∈ h then c (h ∪ (L'), false)
  else pick L ∨ L' ∈ h:
  c (h ∪ (L), false)
  c (h ∪ (L'), false)

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

Recall: Nelson-Oppen with Proof Generation

NO: (pair of L-f and pf L) list → pf false

NO F =
match asat(F), basat(F) with
| Contra d ∶ d → d
| Contra d ∶ d → d
| Eq (x, y; d) ∶ NO (F ∪ {(x = y; d)})
| Eq (x, y; d) ∶ NO (F ∪ {(x = y; d)})
| Sat, Sat → raise NaProof

- 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 E1 ∨ E2 is discovered:
    - Must try to derive a contradiction for each Ei assumption!
    NO F =
    match asat(F), basat(F) with
    | Disj (x1 = y1 ∨ x2 = y2; d) →
    | let d1 = λh; pf (eq x1, y1). NO (F ∪ {(x1 = y1, h)})
    | let d2 = λh; pf (eq x2, y2). NO (F ∪ {(x2 = y2, h)})
    | otherwise →
    | (pf A → pf C) → (pf B → pf C) → 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(m, i, x),…, i, x),…, i, x)
    upd(…,(upd(m, i, x),…, i, x) ∧
    sel(m, i) x ∧… ∧ sel(m, i) x ∧
    entails \bigvee_{1 \leq k \leq i} i_k \vdash
    (a conjunction of length n entails n disjuncts)