### Decision-Procedure Based Theorem Provers

**Tactic-Based Theorem Proving**

Inferring Loop Invariants

CS 294-B
Lecture 12

---

### Combining Satisfiability Procedures

- Consider a set of literals $F$.
  - Containing symbols from two theories $T_1$ and $T_2$.
- We split $F$ into two sets of literals:
  - $F_1$ containing only literals in theory $T_1$.
  - $F_2$ containing only literals in theory $T_2$.
- We name all subexpressions:
  $p(f, E)$ is split into $f_1(E) = n \land p(n)$.
- We have: $\text{unsat}(F_1 \land F_2)$ iff $\text{unsat}(F)$.
  - $\text{unsat}(F) \lor \text{unsat}(F) = \text{unsat}(F)$.
  - But the converse is not true.
- So we cannot compute $\text{unsat}(F)$ with a trivial combination of the sat procedures for $T_1$ and $T_2$.

---

### Combining Satisfiability Procedures. Example

- Consider equality and arithmetic:
  
  \[
  f(f(x) - f(y)) = f(z) \
  x \leq y \
  y + z \leq x \
  0 \leq z 
  \]

  \[
  \text{false} \quad f(f(x) - f(y)) = f(z) 
  \]

---

### Nelson-Oppen Method (I)

1. Represent all conjuncts in the same DAG:
   \[
   f(f(x) - f(y)) = f(z) \land x \geq x \land x \geq y \land z \geq 0 
   \]
Nelson-Oppen Method (2)

2. Run each sat, procedure
   - Require it to report all contradictions (as usual)
   - Also require it to report all equalities between nodes

   ![Diagram]

Prof. Nicole CS 294-8 Lecture 12 7

Nelson-Oppen Method (3)

3. Broadcast all discovered equalities and re-run sat, procedures
   - Until no more equalities are discovered or a contradiction arises

   ![Diagram]

Prof. Nicole CS 294-8 Lecture 12 8

What Theories Can be Combined?

- Only theories without common interpreted symbols
  - But OK if one theory takes the symbol interpreted
- Only certain theories can be combined
  - Consider \( \leq \) and Equality
  - Consider \( 1 \leq x \leq 2 \land a = 1 \land b = 2 \land f(x) = f(a) \land f(x) = f(b) \)
  - No equalities and no contradictions are discovered
  - Yet, unsatisfiable
- A theory is non-convex when a set of literals entails a disjunction of equalities without entailing any single equality

Prof. Nicole CS 294-8 Lecture 12 9

Handling Non-Convex Theories

- Many theories are non-convex
  - Consider the theory of memory and pointers
  - It is not-convex:
    \[
    \text{true} \rightarrow A \rightarrow x : \text{set}(\text{up}(M, A, v), x) = \text{set}(M, x)
    \]
    (neither of the disjuncts is entailed individually)
  - For such theories it can be the case that
    - No contradiction is discovered
    - No single equality is discovered
    - But a disjunction of equalities is discovered
  - We need to propagate disjunction of equalities

Prof. Nicole CS 294-8 Lecture 12 10

Propagating Disjunction of Equalities

- To propagate disjunctions we perform a case split:
  - If a disjunction \( E_1 \lor \ldots \lor E_n \) is discovered:

    ```
    Save the current state of the prover
    for i = 1 to n {
        broadcast \( E_i \)
        if no contradiction arises then return "satisfiable"
        restore the saved prover state
    }
    return "unsatisfiable"
    ```

Prof. Nicole CS 294-8 Lecture 12 11

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.
      \[
      \text{up}(\text{up}(M, l_1, x), \ldots , l_n, x) =
      \text{up}(\text{up}(M, l_1, x), \ldots , l_n, x) \land
      \text{set}(M, l_1) = x \land \ldots \land \text{set}(M, l_n) = x
      \]
    - entails \( V_{l_1} \land \ldots \land V_{l_n} \)
      (conjunction of length \( n \) entails \( n^2 \) disjuncts)

Prof. Nicole CS 294-8 Lecture 12 12
Forward vs. Backward Theorem Proving

- The state of a prover can be expressed as:
  \[ H_1 \land \ldots \land H_n \Rightarrow G \]
  - Given the hypotheses \( H_1 \) try to derive goal \( G \)

- A forward theorem prover derives new hypotheses, in hope of deriving \( G \):
  - If \( H_1 \land \ldots \land H_k \Rightarrow H \) then
    move to state \( H_1 \land \ldots \land H_k \land H \Rightarrow G \)
  - Success state: \( H_1 \land \ldots \land H_k \Rightarrow G \)

- A forward theorem prover uses heuristics to reach \( G \):
  - Or it can exhaustively derive everything that is derivable!

Forward Theorem Proving

- Nelson-Oppen is a forward theorem prover:
  - The state is \( L_1 \land \ldots \land L_n \Rightarrow \) false
  - If \( L_1 \land \ldots \land L_n \land \neg L \Rightarrow E \) (an equality) then
    - New state is \( L_1 \land \ldots \land L_n \land \neg L \land E \Rightarrow \) false (add the equality)
  - Success state is \( L_1 \land \ldots \land L_n \land \neg L \land \neg L \Rightarrow \) false

- Nelson-Oppen provers exhaustively produce all derivable facts hoping to encounter the goal

  - Case splitting can be explained this way too:
    - If \( L_1 \land \ldots \land L_n \land \neg L \Rightarrow E \lor E' \) (a disjunction of equalities) then
      - Two new states are produced (both must lead to success)
        - \( L_1 \land \ldots \land L_n \land \neg L \land E \Rightarrow \) false
        - \( L_1 \land \ldots \land L_n \land \neg L \land E' \Rightarrow \) false

Backward Theorem Proving

- A backward theorem prover derives new subgoals from the goal:
  - The current state is \( H_1 \land \ldots \land H_k \Rightarrow G \)
  - If \( H_1 \land \ldots \land H_k \land G \land L \Rightarrow \) \( G \) (\( G \) are subgoals)
  - Produce \( n \) new states (all must lead to success)
    \[ H_1 \land \ldots \land H_k \Rightarrow G \]

  - Similar to case splitting in Nelson-Oppen:
    - Consider a non-convex theory
      \[ H_1 \land \ldots \land H_k \land E \land E' \Rightarrow \] is same as
      \[ H_1 \land \ldots \land H_k \land E \land \neg E' \Rightarrow \] (thus we have reduced the goal "false" to subgoals - \( E \land \neg E' \))

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.

- Consider an axiom: \( \forall x, \phi(x) \Rightarrow \psi(x) \)
  - Like the clause \( b(x) \wedge \phi(x) \) in Prolog

- This could be turned into a tactic
  - Given tactic for each clause: \( \phi_1, \ldots, \phi_n \)
  - Prolog tactic
    - \( \text{Prolog} \overset{\rho}{\Rightarrow} \text{REPEAT} \left( \phi_1, \text{ORELSE} \phi_2, \ldots, \text{ORELSE} \phi_n \right) \)

- Nelson-Oppen can also be programmed this way
  - The result is not as efficient as a special-purpose implementation

- This is a very powerful mechanism for semi-automatic theorem proving
  - Used in: Isabelle, HOL, and many others

Programming Theorem Provers. Tactics.

- Tactics can be composed using tactics

Examples:

- \( \text{REPEAT} \): tactic \( \Rightarrow \) tactic
  - \( \text{REPEAT} t_1 \Rightarrow \text{REPEAT} t_2 \Rightarrow \text{REPEAT} t_3 \)
  - \( \text{ORELSE} \): tactic \( \Rightarrow \) tactic
  - \( \text{ORELSE} t_1 \Rightarrow \text{ORELSE} t_2 \Rightarrow \text{ORELSE} t_3 \)

Techniques for Inferring Loop Invariants

Inferring Loop Invariants

- Traditional program verification has several elements:
  - Function specifications and loop invariants
  - Verification condition generation
  - Theorem proving

- Requiring specifications from the programmer is often acceptable
- Requiring loop invariants is not acceptable
  - Same for specifications of local functions

Inferring Loop Invariants

- A set of cutpoints is a set of program points:
  - There is at least one cutpoint on each circular path in CFG
  - There is a cutpoint at the start of the program
  - There is a cutpoint before the return

- Consider that our function uses \( x \):
  - We associate with each cutpoint an assertion \( I(x) \)

- If \( x \) is a path from cutpoint \( k \) to \( j \) then:
  - \( r'(x) : \mathbb{Z}^n \rightarrow \mathbb{Z}^n \) expresses the effect of path \( x \) on the values of \( x \) at \( j \) as a function of these at \( k \)
  - \( P(x) \) is a path predicate that is true exactly of these values of \( x \) at \( k \) that will enable the path \( x \)
**Cutpoints Example.**

- \( P_0 = \text{true} \)
  - \( r_0 = \{ A \rightarrow A, K = 0, L = \text{len}(A), S = 0, \mu = \mu \} \)

- \( P_2 = \{ A \rightarrow A, K = K + 1, \mu = \mu \} \)
- \( r_2 = S - S + \text{sel}(A, K), \mu = \mu \)

- Easily obtained through sym eval.

---

**Equational Definition of Invariants**

- A set of assertions is a set of invariants if:
  - The assertion for the start cutpoint is the precondition
  - The assertion for the end cutpoint is the postcondition
  - For each path from to we have:
    \( \forall x, \forall y, \forall \delta, \forall \mu \) \( I_x(A) \land P_y(A) \Rightarrow I_y(A) \)

- Now we have to solve a system of constraints with the unknowns \( I_0, I_1, I_2 \)
  - \( I_0 \) and \( I_2 \) are known
  - We will consider the simpler case of a single loop
  - Otherwise we might want to try solving the inner/last loop first

---

**Invariants Example.**

1. \( I_0 = I_0(A) \)
   - The invariant \( I_0 \) is established initially

2. \( I_0 \land K = L \Rightarrow I_1(A) \)
   - The invariant \( I_0 \) is preserved in the loop

3. \( I_0 \land K = L \Rightarrow I_1(A) \)
   - The invariant \( I_0 \) is strong enough (i.e. useful)

---

**The Lattice of Invariants**

- Weak predicates satisfy the condition 1
  - Are satisfied initially

- Strong predicates satisfy condition 3
  - Are useful

- A few predicates satisfy condition 2
  - Are invariant
  - Form a lattice

---

**Finding the Invariant**

- Which of the potential invariants should we try to find?
  - We prefer to work backwards
    - Essentially proving only what is needed to satisfy \( I_0 \)
    - Forward is also possible but sometimes wasteful since we have to prove everything that holds at any point

---

**Finding the Invariant**

- Thus we do not know the "precondition" of the loop
  - The weakest invariant that is strong enough has most chances of holding initially
  - This is the one that we'll try to find
    - And then check that it is weak enough
**Induction Iteration Method**

- Equation 3 gives a predicate weaker than any invariant:
  \[ I_1 \land K \geq L \Rightarrow I_1(r_1(x)) \]
  \[ W_2 \land K \geq L \Rightarrow I_2(r_2(x)) \]
- Equation 2 suggests an iterative computation of the invariant \( I_k \):
  \[ I_k \Leftarrow (K \geq L) \Rightarrow I_k(r_k(x)) \]

---

**Induction Iteration Method**

- Define a family of predicates:
  \[ W_0 = K \geq L \Rightarrow \exists_0(r_0(x)) \]
  \[ W_j = W_{j-1} \land K \geq L \Rightarrow \exists_j(r_j(x)) \]
- Properties of \( W_j \):
  - \( W_j \Rightarrow W_{j+1} \Rightarrow \ldots \Rightarrow W_0 \) (they form a strengthening chain)
  - \( I_k \Rightarrow W_j \) (they are weaker than any invariant)
  - If \( W_0 \Rightarrow W_j \) then
    - \( W_j \) is an invariant (satisfies both equations 2 and 3)
    - \( W_j \Rightarrow K \geq L \Rightarrow W_j(r_j(x)) \)
  - \( W_j \) is the weakest invariant

(recall domain theory, predicates form a domain, and we use the fixed point theorem to obtain least solutions to recursive equations)

---

**Induction Iteration.**

- The sequence of \( W_j \) approaches the weakest invariant from above:
- The predicate \( W_j \) can quickly become very large
  - Checking \( W_j \Rightarrow W_j \) becomes harder and harder
- This is not guaranteed to terminate

---

**Induction Iteration, Strengthening.**

- We can try to strengthen the inductive invariant
- Instead of:
  \[ W_j = W_{j-1} \land K \geq L \Rightarrow W_j(r_j(x)) \]
  we compute:
  \[ W_j = \text{strengthen}(W_{j-1} \land K \geq L \Rightarrow W_j(r_j(x))) \]
  where \( \text{strengthen}(P) \Rightarrow P \)
- We still have \( W_j \Rightarrow W_0 \), and we stop when \( W_{j+1} \Rightarrow W_j \)
  - The result is still an invariant that satisfies 2 and 3
**Strengthening Heuristics**

- One goal of strengthening is simplification:
  - Drop disjunctions: $P_1 \lor P_2 \rightarrow P_1$
  - Drop implications: $P_1 \Rightarrow P_2 \rightarrow P_1$

- A good idea is to try to eliminate variables changed in the loop body:
  - If $W_i$ does not depend on variables changed by $r$, (e.g. $K, S$)
  - $W_{i+1} = W_i \land K \land L \Rightarrow W_i \land (K \land S)$
  - $W_i \land K \land L \Rightarrow W_i$
  - Now $W_i \Rightarrow W_{i+1}$, and we are done!

---

**One Strengthening Heuristic for Integers**

- Rewrite $W_i$ in conjunctive normal form
  - $W_i = K \geq 0 \land K \leq \text{len}(A) \land (K \land L \Rightarrow K \geq 0 \land K \leq \text{len}(A))$
  - $W_i = K \geq 0 \land K \leq \text{len}(A)$
- Take each disjunction containing arithmetic literals:
  - Negate it and obtain a conjunction of arithmetic literals
  - $K \land L \land K \land L \geq \text{len}(A)$
  - Weaken the result by eliminating a variable (preferably a loop-carried variable)
    - E.g., add the literal: $L \land \text{len}(A)$
      - Negate the result and get another disjunction:
        - $L \leq \text{len}(A)$
        - $W_i = K \geq 0 \land K \leq \text{len}(A) \land L \leq \text{len}(A)$ (check that $W_i \Rightarrow W_{i+1}$)

---

**Theorem Proving, Conclusions.**

- Theorem proving strengths
  - Very expressive
- Theorem proving weaknesses
  - Too ambitious
- A great toolbox for software analysis
  - Symbolic evaluation
  - Decision procedures
- Related to program analysis
  - Abstract interpretation on the lattice of predicates

---

**Induction Iteration. Strengthening**

- We are still in the "strong-enough" area
- We are making bigger steps
- And we might over-shoot then weakest invariant
- We might also fail to find any invariant
- But we do so quickly

---

**Induction Iteration**

- We showed a way to compute invariants algorithmically
  - Similar to fixed-point computation in domains
  - Similar to abstract interpretation on the lattice of predicates
- Then we discussed heuristics that improve the termination properties
  - Similar to widening in abstract interpretation

---

**Prof. Nicolas CS 204-8 Lecture 12**