Lesson 1

SAT & SMT solvers

Static Program Analysis and Constraint Solving

Master's Degree in Formal Methods in Computer Science

Manuel Montenegro (montenegro@fdi.ucm.es)

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Outline

The challenge of satisfiability

Solving SAT

Solving SMT

Equality logic and uninterpreted functions

Array theory

More quantifier-free theories

Logic with quantifiers

The challenge of satisfiability

The challenge: satisfiability problem

[SAT] Given a formula φ in propositional logic:
Is φ satisfiable?

[SAT-FOL] Given a formula φ in first-order logic:
Is φ satisfiable?

We want to study decision procedures for these two problems.

Which are satisfiable?

- $p \wedge q \Rightarrow \neg q$.
    -
- $(p \Rightarrow q) \wedge p \wedge \neg q$
    -
- $\forall x.\ p(x) \wedge \neg p(x)$
    -
- $\forall x.\ \exists y.\ q(x, y)$
    -

Why is satisfiability important?

Many decision problems in logic can be expressed in terms of satisfiability:

- **Logical validity**. Given a formula φ. Is φ valid?

$$\varphi \text{ valid} \quad \Longleftrightarrow \quad \neg\varphi \text{ unsatisfiable}$$

- **Logical consequence**. Given formulas $\varphi_1, \ldots, \varphi_n$, and φ. Does φ follow from $\varphi_1, \ldots, \varphi_n$?

$$\varphi_1, \ldots, \varphi_n \vDash \varphi \quad \Longleftrightarrow \quad \varphi_1 \wedge \ldots \wedge \varphi_n \wedge \neg\varphi \text{ unsatisfiable}$$

Why is satisfiability important?

Many applications. For example, program verification:

```
method sqrt(x: nat) returns (r: nat)
  ensures $r^2 \leq x < (r + 1)^2$
{
    r := 0;
    while $((r + 1)^2 \leq x)$
      invariant $r^2 \leq x$
    {
        r := r + 1;
    }
}
```

A program verifier generates first-order formulas (verification conditions) whose validity implies program soundness.

Why is it important?

Verification conditions for `sqrt`:

- Loop invariant holds on entry
  $\forall r.\, \forall x.\, r = 0 \Rightarrow r^2 \leq x$
- Invariant is preserved by loop body
  $\forall r.\, \forall x.\, \forall r'.\, r^2 \leq x \wedge (r+1)^2 \leq x \wedge r' = r + 1 \Rightarrow r'^2 \leq x$
- Postcondition follows when exiting the loop
  $\forall r.\, \forall x.\, r^2 \leq x \wedge \neg((r+1)^2 \leq x) \Rightarrow r^2 \leq x < (r+1)^2$

Why is satisfiability important?

Another application: static analysis of programs.

- **Nullity analysis**

  NullPointerException?

  ```
  x = y;
  if (y != null) { x . doSomething(); }
  ```

  Prove: $\forall x.\, \forall y.\, x = y \land y \neq null \Rightarrow x \neq null$

- **Termination analysis**

  Termination measure for `sqrt`: $x - r$

  - Is it nonnegative?

    $\forall x.\, \forall r.\, r^2 \leq x \Rightarrow x - r \geq 0$
  - Is it strictly decreasing?

    $\forall x.\, \forall r.\, \forall r'.\, r^2 \leq x \land (r+1)^2 \leq x \land r' = r + 1 \Rightarrow x - r' < x - r$

- **Array bounds checking**, etc.

Why is satisfiability important?

More applications:

- Graph coloring *(see exercises)*.
- Termination of Term Rewriting Systems.
- Hardware verification.
- etc.

J. Marques-Silva
Practical Applications of Boolean Satisfiability
9th International Workshop on Discrete Event Systems (WODES 2008)

Why is satisfiability a challenge?

- Satisfiability in propositional logic (SAT) is NP-complete
  - 😟 No known algorithm running in polynomial time.
  - 😩 ...and probably it does not exist (unless $P = NP$).
  - 😊 But, still, we can decide satisfiability in a reasonable amount of time (algorithms DPLL / CDCL).
- Satisfiability in first-order logic (SAT-FOL) is undecidable
  - 😟 There is no algorithm at all that works in any case.
  - 😫 ...and there will never be!
  - 😊 But we can tackle an easier challenge which is useful in many cases: Satisfiability Modulo Theories (SMT).

So, what is SMT?

- A **theory** T is defined by:
  - A fixed signature Σ.
  - A set of axioms (closed Σ-formulas).
- A formula φ is **T-satisfiable** if there exists an structure satisfying φ and the axioms of T.
- A formula φ is **T-valid** if every structure satisfying the axioms of T also satisfies φ.

Instead studying a decision procedure for [SAT-FOL], we want to address the following decision problem:

[SMT] Given a formula φ in a theory T: Is φ T-satisfiable?

Fragments of a logic

- A theory only restricts the signature Σ.
- It does not restrict the set of logical symbols.
- If we want to restrict the latter, we speak about a fragment of the logic.
- For example:
    - Quantifier-free fragment: No $\forall$ or $\exists$ are allowed.
    - 2-CNF: Formulas in CNF with only two literals per clause.
    - Conjunctive fragment: The only connective allowed is $\wedge$.

Unless otherwise stated, we will address satisfiability of quantifier-free fragments of theories.

Example: equality logic

It is the quantifier-free fragment of the following theory:

- Signature $\Sigma = \{=\}$.
- Axioms:
  - Reflexivity: $\forall x.\ x = x$.
  - Symmetry: $\forall x.\ \forall y.\ x = y \Rightarrow y = x$.
  - Transitivity: $\forall x.\ \forall y.\ \forall z.\ x = y \land y = z \Rightarrow x = z$.

Equality logic is decidable.

Is the following formula satisfiable in equality logic?

$$x = y \land y = z \land z = w \land x \neq w$$

Example: equality logic with uninterpreted functions

- Signature $\Sigma = \{=\} \cup \{\underbrace{f_1, \ldots, f_n}_{\text{Function symbols}}\} \cup \{\underbrace{P_1, \ldots, P_m}_{\text{Predicate symbols}}\}$.
    - Each f_i has n_i parameters.
    - Each P_i has m_i parameters.
- Axioms:
    - Those of equality logic, plus:
    - For each f_i ($i \in \{1..n\}$):

$$\forall \overline{x_i}, \overline{y_i}.\, x_1 = y_1 \wedge \cdots \wedge x_{n_i} = y_{n_i} \Rightarrow f_i(x_1, \ldots, x_{n_i}) = f_i(y_1, \ldots, y_{n_i})$$

    - For each P_i ($i \in \{1..m\}$):

$$\forall \overline{x_i}, \overline{y_i}.\, x_1 = y_1 \wedge \cdots \wedge x_{m_i} = y_{m_i} \Rightarrow P_i(x_1, \ldots, x_{m_i}) \Leftrightarrow P_i(y_1, \ldots, y_{m_i})$$

This logic is decidable.

Is the following formula satisfiable in EUF theory?

Given $\Sigma = \{=, get\}$:

$$a = b \land get(a, i) = k \land \neg(get(b, i) = k)$$

Example: linear integer arithmetic

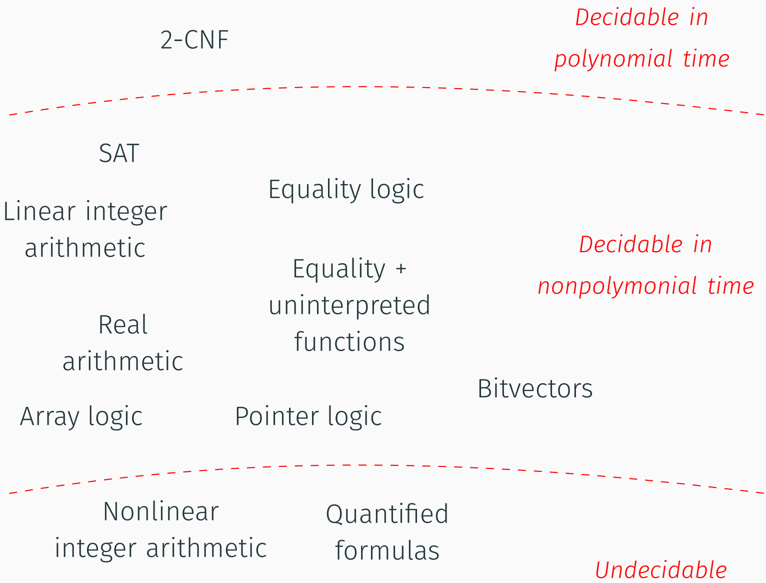
- Signature $\Sigma = \{=, +, \leq, 0, 1\}$
- Axioms:
  - Equality axioms: reflexivity, symmetry, transitivity.
  - Order axioms: reflexivity, antisymmetry, transitivity, etc.
  - Addition axioms: $x + 0 = x$, $x + y = y + x$ etc.

Example

The following formula is satisfiable in linear arithmetic:

$$y \leq 2 - 3x \wedge x \geq 0 \wedge y \geq 0$$

Some first-order theories



2-CNF

Decidable in polynomial time

SAT

Equality logic

Linear integer arithmetic

Decidable in nonpolymonial time

Equality + uninterpreted functions

Real arithmetic

Bitvectors

Array logic

Pointer logic

Nonlinear integer arithmetic

Quantified formulas

Undecidable

Solving SAT

Solving SAT

[SAT] Given a formula in propositional logic. Is it satisfiable?

- Davis-Putnam-Logemann-Loveland (DPLL) algorithm.
  - Classic algorithm, dating back from 1962.
  - Basis of several SMT solvers.

- Conflict-Driven Clause Learning (CDCL) algorithm. ⟵ This lesson
  - Inspired by DPLL.
  - Adds ”learning” capabilities and nonchronological backtracking.

CDCL: ASSUMPTIONS

We assume that the input formula is in Conjunctive Normal Form (CNF).

Example

$$\begin{array}{rl} & (\neg p_1 \vee \neg p_2) \\ \wedge & (\neg p_1 \vee p_3) \\ \wedge & (\neg p_4 \vee \neg p_3 \vee \neg p_5) \\ \wedge & (p_2 \vee p_5 \vee p_6) \\ \wedge & (p_5 \vee \neg p_7) \\ \wedge & (\neg p_6 \vee p_7) \\ \wedge & (\neg p_5 \vee \neg p_8 \vee \neg p_3) \end{array}$$

Each member of the conjunction is a clause.

CDCL: ASSUMPTIONS

We assume that the input formula is in Conjunctive Normal Form (CNF).

Example

$$\begin{aligned}
c_1 &: \quad (\neg p_1 \vee \neg p_2) \\
c_2 &: \quad (\neg p_1 \vee p_3) \\
c_3 &: \quad (\neg p_4 \vee \neg p_3 \vee \neg p_5) \\
c_4 &: \quad (p_2 \vee p_5 \vee p_6) \\
c_5 &: \quad (p_5 \vee \neg p_7) \\
c_6 &: \quad (\neg p_6 \vee p_7) \\
c_7 &: \quad (\neg p_5 \vee \neg p_8 \vee \neg p_3)
\end{aligned}$$

Each member of the conjunction is a clause.

CDCL: BASIC DEFINITIONS

- Idea: incremental construction of partial assignments.
- A **partial assignment** (PA) is a partial mapping from propositional symbols to values in $\{true, false\}$.
- Given a PA, a literal can be **satisfied**, **not satisfied**, or **unassigned**.

Example

Assume $\alpha = [p_1 \mapsto true, p_2 \mapsto false, p_4 \mapsto true]$.

- Satisfied: p_1, $\neg p_2$, p_4.
- Not satisfied: $\neg p_1$, p_2, $\neg p_4$.
- Unassigned: p_5, $\neg p_3$.

CDCL: BASIC DEFINITIONS

Given a PA:

- A clause is satisfied if at least one of its literals is satisfied.

Example

Given $\alpha = [p_1 \mapsto \mathit{true}, p_2 \mapsto \mathit{false}, p_4 \mapsto \mathit{true}]$:

$$p_2 \vee p_1 \vee \neg p_3 \vee \neg p_4$$ satisfied

Will satisfiability be affected if we add new bindings to the PA?

CDCL: BASIC DEFINITIONS

Given a PA:

- A clause is conflicting if all its literals are not satisfied.

Example

Given $\alpha = [p_1 \mapsto \mathit{true}, p_2 \mapsto \mathit{false}, p_4 \mapsto \mathit{true}]$:

$$\neg p_1 \vee \neg p_4 \qquad \text{conflicting}$$

Can we extend α so that the clause becomes satisfied?

CDCL: BASIC DEFINITIONS

Given a PA:

- A clause is unit if **one** of its literals is unassigned and the rest is not satisfied.

Example

Given $\alpha = [p_1 \mapsto \mathit{true}, p_2 \mapsto \mathit{false}, p_4 \mapsto \mathit{true}]$:

$$\neg p_1 \vee \neg p_5 \vee p_2 \vee \neg p_4 \qquad \text{unit}$$

Why do we want to distinguish unit clauses from the other kind of clauses?

CDCL: BASIC DEFINITIONS

Given a PA:

- A clause is **unresolved** if is not satisfied, conflicting, or unit.

Example

Given $\alpha = [p_1 \mapsto \mathit{true}, p_2 \mapsto \mathit{false}, p_4 \mapsto \mathit{true}]$:

$$p_2 \vee p_5 \vee \neg p_8 \vee \neg p_1$$

CDCL: our goal

We want to find a total assignment satisfying all clauses:

Example

$$\text{Given } \alpha = \begin{bmatrix} p_1 \mapsto \mathit{true}, & p_3 \mapsto \mathit{true}, & p_5 \mapsto \mathit{true}, & p_7 \mapsto \mathit{true}, \\ p_2 \mapsto \mathit{false}, & p_4 \mapsto \mathit{false}, & p_6 \mapsto \mathit{true}, & p_8 \mapsto \mathit{false} \end{bmatrix}$$

c_1 :	$(\neg p_1 \vee \neg p_2)$	satisfied
c_2 :	$(\neg p_1 \vee p_3)$	satisfied
c_3 :	$(\neg p_4 \vee \neg p_3 \vee \neg p_5)$	satisfied
c_4 :	$(p_2 \vee p_5 \vee p_6)$	satisfied
c_5 :	$(p_5 \vee \neg p_7)$	satisfied
c_6 :	$(\neg p_6 \vee p_7)$	satisfied
c_7 :	$(\neg p_5 \vee \neg p_8 \vee \neg p_3)$	satisfied

CDCL: general idea

- We start with an empty PA: [ ].
- There are two ways in which it can be extended:
  - Unit rule. There is a unit clause. We extend the PA such that the unassigned literal becomes satisfied.
  - Decision. We cannot apply the unit rule. We pick some unassigned symbol a give it an arbitrary truth value.
- If a clause becomes conflicting, we have to backtrack to undo some decisions made before.
  - A new clause will be added in order not to end up with the same assignment again.

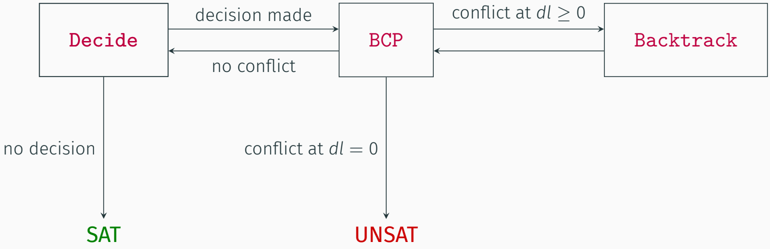
CDCL: implication graph

- The current PA will be represented by means of an **implication graph**.
- Vertices in the graph represent bindings in the PA:

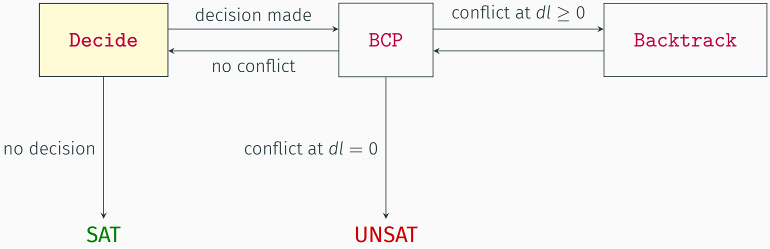
$$p@dl \qquad or \qquad \neg p@dl$$

  - $p@dl$ denotes the binding $[p \mapsto true]$ in the PA.
  - $\neg p@dl$ denotes the binding $[p \mapsto false]$ in the PA.
- dl is a natural number, which is the **decision level** in which the vertex was added to the graph.
  - Initially, $dl = 0$.
  - Whenever a decision is made, dl is incremented by one.
  - Backtracking to a level m amounts to removing from the graph all vertices whose decision level is $\geq m$.

CDCL: GENERAL SCHEME



CDCL: Decide



- Decide increments the current decision level, picks a variable p and assigns an arbitrary truth value to it.
- Depending on the value assigned, it adds the vertex $p@dl$ or $\neg p@dl$ to the implication graph.

CDCL: Decide

Example:

$$\begin{aligned}
c_1 &: \quad (\neg p_1 \vee \neg p_2) \\
c_2 &: \quad (\neg p_1 \vee p_3) \\
c_3 &: \quad (\neg p_4 \vee \neg p_3 \vee \neg p_5) \\
c_4 &: \quad (p_2 \vee p_5 \vee p_6) \\
c_5 &: \quad (p_5 \vee \neg p_7) \\
c_6 &: \quad (\neg p_6 \vee p_7) \\
c_7 &: \quad (\neg p_5 \vee \neg p_8 \vee \neg p_3)
\end{aligned}$$

- Initially, $dl = 0$.
- `Decide` sets dl to 1, picks p_1 and assigns *true* to it.

CDCL: Decide

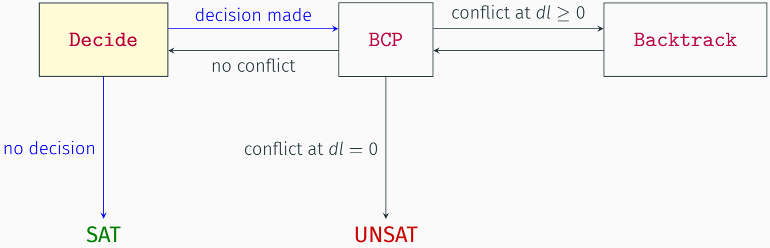
Current implication graph:

p_1@1

Current state:

c_1 :	$(\neg p_1 \vee \neg p_2)$	unit
c_2 :	$(\neg p_1 \vee p_3)$	unit
c_3 :	$(\neg p_4 \vee \neg p_3 \vee \neg p_5)$	
c_4 :	$(p_2 \vee p_5 \vee p_6)$	
c_5 :	$(p_5 \vee \neg p_7)$	
c_6 :	$(\neg p_6 \vee p_7)$	
c_7 :	$(\neg p_5 \vee \neg p_8 \vee \neg p_3)$	

CDCL: Decide



- When a decision is made, we jump to BCP.
- If there had not been decisions to be made (because all literals in the formula were already assigned), we would have found a satisfying assignment, and hence returned SAT.

CDCL: BCP

- BCP = Boolean Constant Propagation.
- It applies repeatedly the unit rule until:
  - a clause becomes conflicting, or
  - the unit rule can no longer be applied.
- For each unit clause:

$$c_j : lit_1 \lor lit_2 \lor \ldots \lor lit_n \lor lit \qquad \textbf{unit}$$

  - It adds vertex $lit@dl$ to the graph.
  - For each $i \in \{1..n\}$, it adds the following edge:

$$\neg lit_i@dl_i \xrightarrow{c_j} lit@dl$$

CDCL: BCP

In our example, we apply the unit rule with c_1 :$(\neg p_1 \lor \neg p_2)$

Implication graph:

$$p_1@1 \xrightarrow{c_1} \neg p_2@1$$

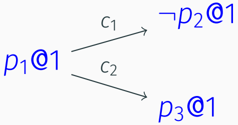
Current state:

c_1 :	$(\neg p_1 \lor \neg p_2)$	satisfied
c_2 :	$(\neg p_1 \lor p_3)$	unit
c_3 :	$(\neg p_4 \lor \neg p_3 \lor \neg p_5)$	
c_4 :	$(p_2 \lor p_5 \lor p_6)$	
c_5 :	$(p_5 \lor \neg p_7)$	
c_6 :	$(\neg p_6 \lor p_7)$	
c_7 :	$(\neg p_5 \lor \neg p_8 \lor \neg p_3)$	

CDCL: BCP

We apply the unit rule with c_2 :($\neg p_1$ $\lor p_3$)

Implication graph:



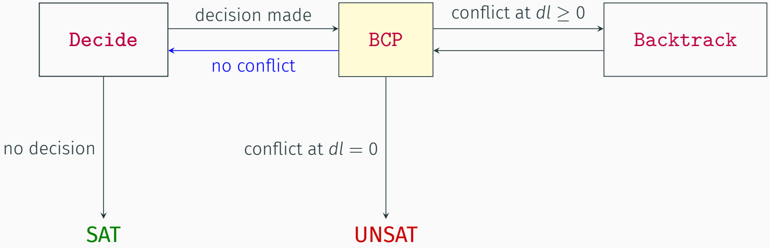
CDCL: BCP

Current state:

$$\begin{array}{lll}
c_1 : & (\neg p_1 \vee \neg p_2) & \text{satisfied} \\
c_2 : & (\neg p_1 \vee p_3) & \text{satisfied} \\
c_3 : & (\neg p_4 \vee \neg p_3 \vee \neg p_5) & \\
c_4 : & (p_2 \vee p_5 \vee p_6) & \\
c_5 : & (p_5 \vee \neg p_7) & \\
c_6 : & (\neg p_6 \vee p_7) & \\
c_7 : & (\neg p_5 \vee \neg p_8 \vee \neg p_3) &
\end{array}$$

There are no more unit clauses, so BCP has finished.

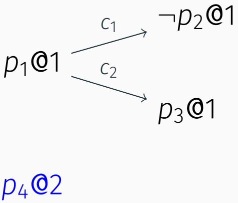
CDCL: BCP



- If there are no conflicting clauses after BCP has finished, we jump back to Decide to make another decision.
- We increment the decision level, so $dl = 2$.
- We choose p_4 and assign an arbitrary value (e.g. *true*) to it.

CDCL: Decide

Implication graph:



CDCL: Decide

Current state:

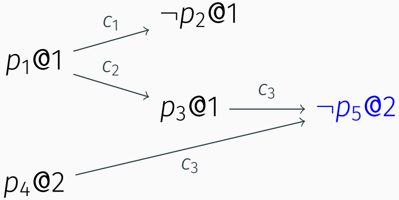
$$
\begin{array}{lll}
c_1 : & (\neg p_1 \lor \neg p_2) & \text{satisfied} \\
c_2 : & (\neg p_1 \lor p_3) & \text{satisfied} \\
c_3 : & (\neg p_4 \lor \neg p_3 \lor \neg p_5) & \text{unit} \\
c_4 : & (p_2 \lor p_5 \lor p_6) & \\
c_5 : & (p_5 \lor \neg p_7) & \\
c_6 : & (\neg p_6 \lor p_7) & \\
c_7 : & (\neg p_5 \lor \neg p_8 \lor \neg p_3) &
\end{array}
$$

After making the decision, we start BCP again.

CDCL: BCP

Applying unit rule with clause c_3 :($\neg p_4$ $\vee$ $\neg p_3$ $\vee \neg p_5$)

Implication graph:



$\neg p_2$@1

c_1

p_1@1

c_2

p_3@1

c_3

$\neg p_5$@2

p_4@2

c_3

CDCL: BCP

Current state:

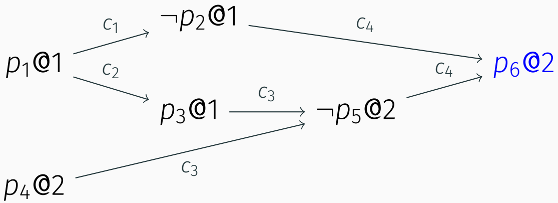
c_1 :	$(\neg p_1 \vee \neg p_2)$	satisfied
c_2 :	$(\neg p_1 \vee p_3)$	satisfied
c_3 :	$(\neg p_4 \vee \neg p_3 \vee \neg p_5)$	satisfied
c_4 :	$(p_2 \vee p_5 \vee p_6)$	unit
c_5 :	$(p_5 \vee \neg p_7)$	unit
c_6 :	$(\neg p_6 \vee p_7)$	
c_7 :	$(\neg p_5 \vee \neg p_8 \vee \neg p_3)$	satisfied

We still can apply the unit rule.

CDCL: BCP

Applying unit rule with clause c_4 :(p_2 $\vee$ p_5 $\vee$ p_6)

Implication graph:



CDCL: BCP

Current state:

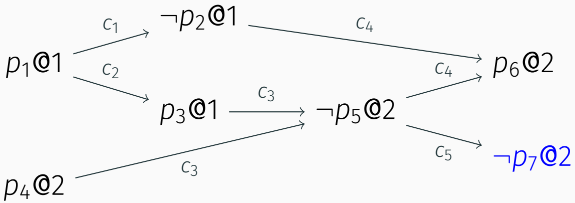
c_1 :	$(\neg p_1 \vee \neg p_2)$	satisfied
c_2 :	$(\neg p_1 \vee p_3)$	satisfied
c_3 :	$(\neg p_4 \vee \neg p_3 \vee \neg p_5)$	satisfied
c_4 :	$(p_2 \vee p_5 \vee p_6)$	satisfied
c_5 :	$(p_5 \vee \neg p_7)$	unit
c_6 :	$(\neg p_6 \vee p_7)$	unit
c_7 :	$(\neg p_5 \vee \neg p_8 \vee \neg p_3)$	satisfied

We still can apply the unit rule.

CDCL: BCP

Applying unit rule with clause $c_5 : (p_5 \lor \neg p_7)$

Implication graph:



CDCL: BCP

Current state:

c_1 :	$(\neg p_1 \vee \neg p_2)$	satisfied
c_2 :	$(\neg p_1 \vee p_3)$	satisfied
c_3 :	$(\neg p_4 \vee \neg p_3 \vee \neg p_5)$	satisfied
c_4 :	$(p_2 \vee p_5 \vee p_6)$	satisfied
c_5 :	$(p_5 \vee \neg p_7)$	satisfied
c_6 :	$(\neg p_6 \vee p_7)$	conflicting
c_7 :	$(\neg p_5 \vee \neg p_8 \vee \neg p_3)$	satisfied

A clause have become conflicting. BCP finishes.

CDCL: CONFLICTS

- Conflicts are represented by means of a conflict node κ.
- Given a conflicting clause:

$$c_j : \quad \boxed{lit_1} \vee \ldots \vee \boxed{lit_n}$$

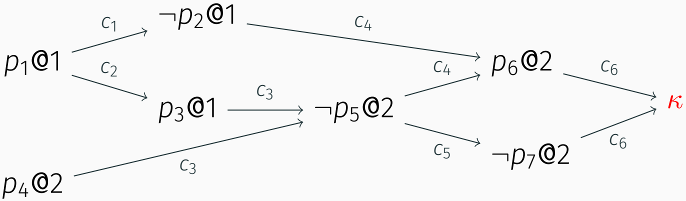
- We add an edge from each $\neg lit_i$ to the conflict node:

$$\neg lit_i \xrightarrow{c_j} \kappa$$

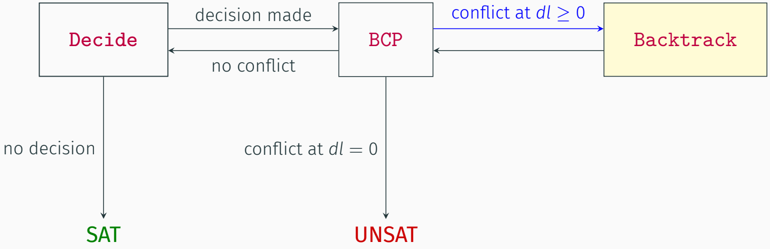
CDCL: conflicts

In our example, the conflicting clause is c_6 :($\neg p_6$ $\vee$ p_7)

Implication graph:

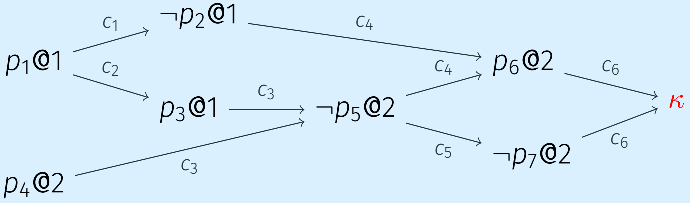


CDCL: Backtrack



- When a conflict is found, the algorithm has to undo some of the decisions made.
- This is what Backtrack does.

Which nodes are responsible for the conflict?

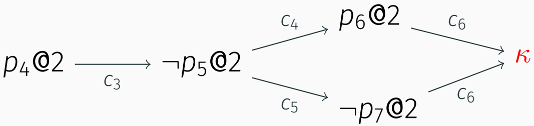


CDCL: `Backtrack`

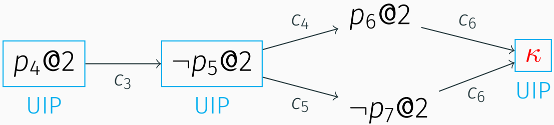
- There are several schemes which allow us to decide which set of nodes is to blame for the conflict.
- Here we shall use the first UIP (1-UIP) scheme.

CDCL: unique implication points (UIPs)

- Let us focus on the subgraph corresponding to the current decision level:

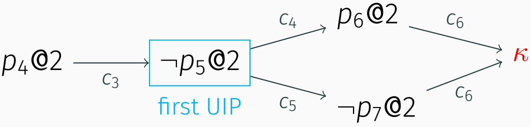


- A **unique implication point** (UIP) is a vertex in the subgraph that is on all paths from the decision node to the conflict node.



CDCL: first UIP

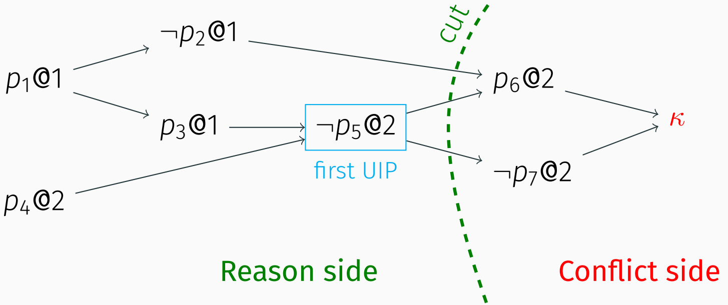
- A first UIP (1-UIP) is the UIP that is closest to the conflict node, excluding the conflict node itself.



CDCL: GRAPH CUT

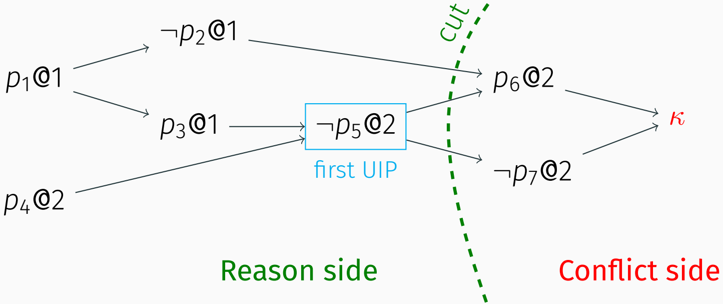
Let us partition the vertices of the graph into two subsets:

- **Conflict side**: vertices belonging to a path between the 1-UIP and the conflict node (excluding the 1-UIP).
- **Reason side**: the remaining vertices.



CDCL: 1-UIP scheme

The 1-UIP scheme blames those nodes in the reason side having an outgoing edge to some node in the conflict side.



In our example: $\neg p_2$ and $\neg p_5$.

CDCL: LEARNING NEW CLAUSES

- In the following we would like to remember that both $\neg p_2$ and $\neg p_5$ lead to a contradiction:
- More formally:

$$\neg p_2 \wedge \neg p_5 \Rightarrow \mathit{false}$$

- or, equivalently,

$$p_2 \vee p_5 \qquad \longleftarrow \textbf{New clause!}$$

- The clause created is a conflict clause.

CDCL: which level backtrack to?

- We backtrack to the higher (i.e. most recent) decision level in the conflict clause, excluding that of the 1-UIP.
- If the conflict clause is unary (i.e. only contains the 1-UIP), we backtrack to level 0.

In our example, which level should we backtrack to?

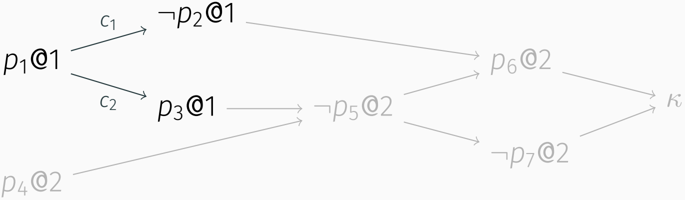
In our example: $p_2 \vee p_5$ is the conflict clause.

-
-

CDCL: Backtrack

Backtracking to level 1 implies:

- Set the current *dl* to 1.
- Remove from the graph all vertices having a decision level greater than 1.



CDCL: Backtrack

Current state after backtracking:

$c_1 : \quad (\neg p_1 \vee \neg p_2)$ satisfied

$c_2 : \quad (\neg p_1 \vee p_3)$ satisfied

$c_3 : \quad (\neg p_4 \vee \neg p_3 \vee \neg p_5)$

$c_4 : \quad (p_2 \vee p_5 \vee p_6)$

$c_5 : \quad (p_5 \vee \neg p_7)$

$c_6 : \quad (\neg p_6 \vee p_7)$

$c_7 : \quad (\neg p_5 \vee \neg p_8 \vee \neg p_3)$

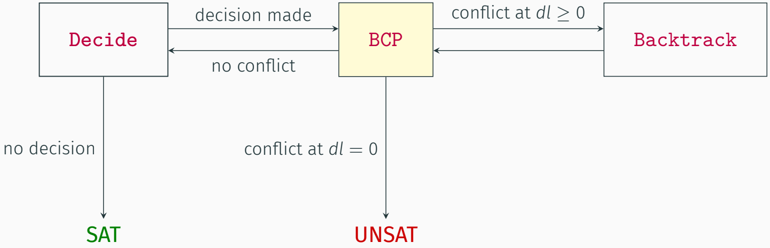
NEW! → $c_8 : \quad (p_2 \vee p_5)$ unit

CDCL: properties of the conflict clause

The newly added clause has the following properties:

- It is a logical consequence of the remaining clauses.
  - Hence adding it does not affect satisfiability.
- It was a conflicting clause before backtracking.
- It becomes unit after backtracking.

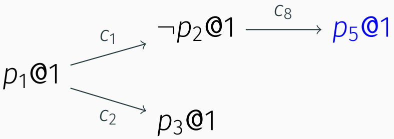
CDCL: BCP



After backtracking we go back to BCP.

CDCL: BCP

Applying unit rule with $c_8 : (p_2 \vee p_5)$

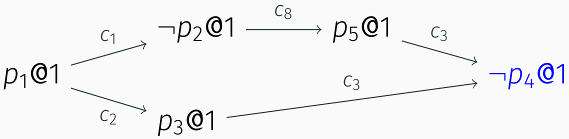


Current state:

c_1 :	$(\neg p_1 \vee \neg p_2)$	satisfied
c_2 :	$(\neg p_1 \vee p_3)$	satisfied
c_3 :	$(\neg p_4 \vee \neg p_3 \vee \neg p_5)$	unit
c_4 :	$(p_2 \vee p_5 \vee p_6)$	satisfied
c_5 :	$(p_5 \vee \neg p_7)$	satisfied
c_6 :	$(\neg p_6 \vee p_7)$	
c_7 :	$(\neg p_5 \vee \neg p_8 \vee \neg p_3)$	unit
c_8 :	$(p_2 \vee p_5)$	satisfied

CDCL: BCP

Applying unit rule with c_3 :$(\neg p_4 \vee \neg p_3 \vee \neg p_5)$

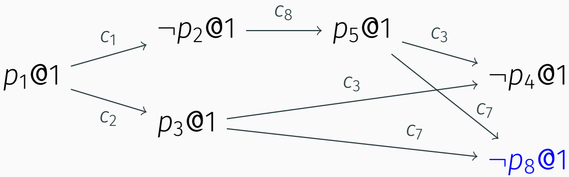


Current state:

c_1 :	$(\neg p_1 \vee \neg p_2)$	satisfied
c_2 :	$(\neg p_1 \vee p_3)$	satisfied
c_3 :	$(\neg p_4 \vee \neg p_3 \vee \neg p_5)$	satisfied
c_4 :	$(p_2 \vee p_5 \vee p_6)$	satisfied
c_5 :	$(p_5 \vee \neg p_7)$	satisfied
c_6 :	$(\neg p_6 \vee p_7)$	
c_7 :	$(\neg p_5 \vee \neg p_8 \vee \neg p_3)$	unit
c_8 :	$(p_2 \vee p_5)$	satisfied

CDCL: BCP

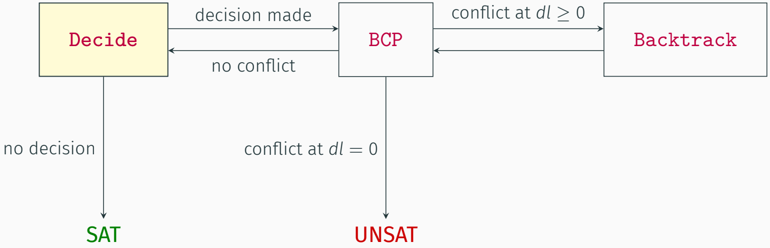
Applying unit rule with c_7 :$(\neg p_5 \vee \neg p_8 \vee \neg p_3)$



Current state:

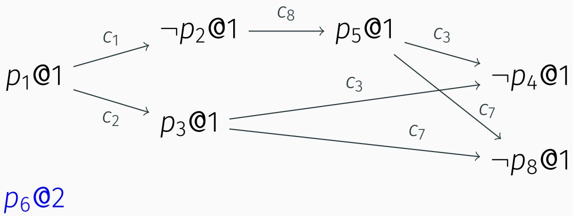
c_1 :	$(\neg p_1 \vee \neg p_2)$	satisfied
c_2 :	$(\neg p_1 \vee p_3)$	satisfied
c_3 :	$(\neg p_4 \vee \neg p_3 \vee \neg p_5)$	satisfied
c_4 :	$(p_2 \vee p_5 \vee p_6)$	satisfied
c_5 :	$(p_5 \vee \neg p_7)$	satisfied
c_6 :	$(\neg p_6 \vee p_7)$	
c_7 :	$(\neg p_5 \vee \neg p_8 \vee \neg p_3)$	satisfied
c_8 :	$(p_2 \vee p_5)$	satisfied

CDCL: BCP



- We can no longer apply the unit rule, so we go to `Decide`.
- Let us pick p_6 with *true*, and set *dl* to 2.

CDCL: Decide

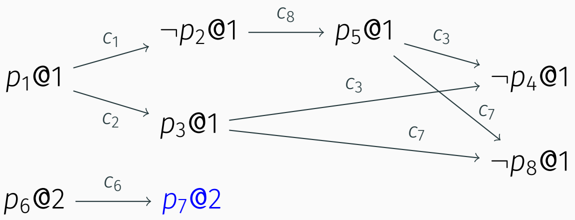


Current state:

c_1 :	$(\neg p_1 \lor \neg p_2)$	satisfied
c_2 :	$(\neg p_1 \lor p_3)$	satisfied
c_3 :	$(\neg p_4 \lor \neg p_3 \lor \neg p_5)$	satisfied
c_4 :	$(p_2 \lor p_5 \lor p_6)$	satisfied
c_5 :	$(p_5 \lor \neg p_7)$	satisfied
c_6 :	$(\neg p_6 \lor p_7)$	unit
c_7 :	$(\neg p_5 \lor \neg p_8 \lor \neg p_3)$	satisfied
c_8 :	$(p_2 \lor p_5)$	satisfied

CDCL: BCP

Back to BCP, we apply the unit rule with c_6 :($\neg p_6$ $\vee$ p_7):

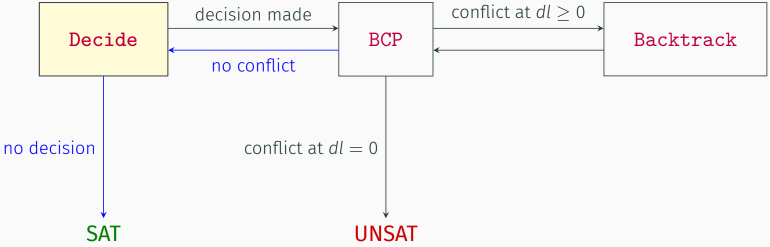


Current state:

c_1 :	$(\neg p_1 \vee \neg p_2)$	satisfied
c_2 :	$(\neg p_1 \vee p_3)$	satisfied
c_3 :	$(\neg p_4 \vee \neg p_3 \vee \neg p_5)$	satisfied
c_4 :	$(p_2 \vee p_5 \vee p_6)$	satisfied
c_5 :	$(p_5 \vee \neg p_7)$	satisfied
c_6 :	$(\neg p_6 \vee p_7)$	satisfied
c_7 :	$(\neg p_5 \vee \neg p_8 \vee \neg p_3)$	satisfied
c_8 :	$(p_2 \vee p_5)$	satisfied

CDCL: Decide

Since there are no conflicts, we jump back into Decide

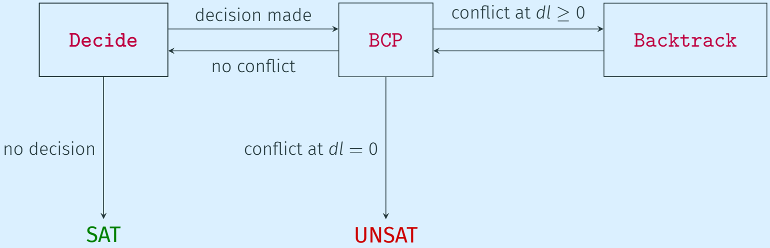


However, there are no unassigned variables, so we return SAT with the following assignment:

$$\alpha = \begin{bmatrix} p_1 \mapsto true, & p_3 \mapsto true, & p_5 \mapsto true, & p_7 \mapsto true, \\ p_2 \mapsto false, & p_4 \mapsto false, & p_6 \mapsto true, & p_8 \mapsto false \end{bmatrix}$$

IN WHICH PHASE SHOULD THE CDCL ALGORITHM START?

Decide or BCP?



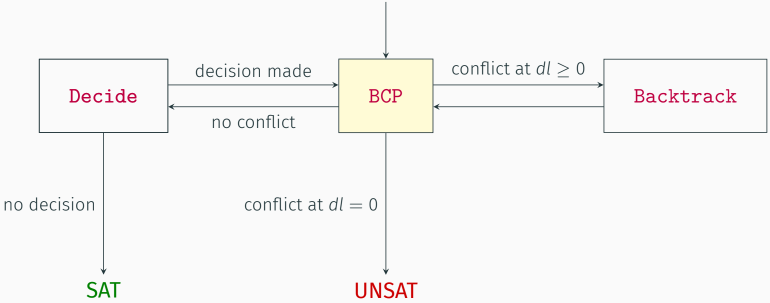
CDCL: Where do we start?

- It makes sense applying the unit rule even if we have not made any decision yet.
    - For example, when the input formula has singleton clauses,

$$c_j : \quad p_i \qquad \text{unit}$$

    - In this case we would add the node p_i@0 to the graph.

CDCL: Where do we start?



Therefore, the algorithm should start in BCP.

Decision heuristics

- In the previous example we have picked a variable at random whenever we have to make a decision.
- There are several decision heuristics that determine which variable to choose.
- These heuristics may greatly improve algorithm's performance.
- Many decision heuristics compute a score for each literal according to some criteria and choose the literal with the highest score.

Decision heuristics

Some score-based heuristics:

- Jeroslow-Wang formula:

$$Score(lit) = \sum_{\substack{i \in \{1..n\} \\ lit \in c_i}} 2^{-|c_i|}$$

  where $|c_i|$ is the length of the clause c_i.
- DLIS: Dynamic Largest Individual Sum.

$$Score(lit) = \begin{array}{l}\text{Number of currently unsatisfied clauses} \\ \text{that would become satisfied by } lit\end{array}$$

- VSIDS: Variable State Independent Decaying Sum:
  - Whenever a conflict is found, add 1 to the score of all literals occurring in the conflicting clause.
  - The score of all literals is periodically divided by 2.

Solving SMT

Solving SMT

[SMT] Given a formula φ without quantifiers in a theory T. Is φ satisfiable?

Example

- Theory of linear integer arithmetic:

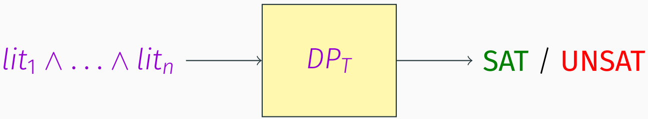
$$x > 10 \land \neg(x \leq 3)$$
$$(x + y \leq 5 \land x > 5) \lor y < 3$$

- Theory of arrays:

$$a[i] = 1 \land i = j \land \neg(a[j] = 1)$$

Can we solve it in a generic way?

- Yes, if we isolate the theory-specific part.
- For a theory T we assume that there are a decision procedure DP_T that can determine the satisfiability of a **conjunction of literals** in T.



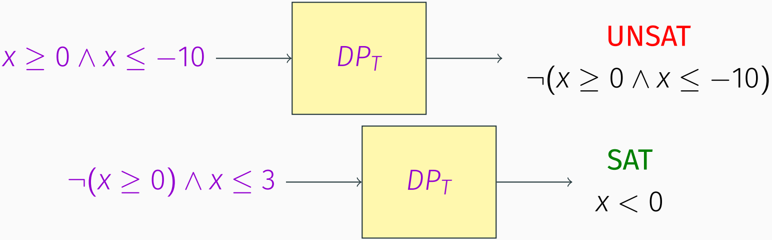
$lit_1 \wedge \ldots \wedge lit_n$

DP_T

SAT / UNSAT

Additional conditions on DP_T

- Besides **SAT** / **UNSAT**, DP_T also returns a T-valid formula (lemma).



$x \geq 0 \wedge x \leq -10 \longrightarrow \boxed{DP_T} \longrightarrow$ UNSAT $\neg(x \geq 0 \wedge x \leq -10)$

$\neg(x \geq 0) \wedge x \leq 3 \longrightarrow \boxed{DP_T} \longrightarrow$ SAT $x < 0$

Solving SMT

- Here we shall use the **DPLL(T)** algorithm.
- DPLL(T) is a generalization of CDCL to the SMT decision problem.

$$\boxed{\text{DPLL}(T) = \text{CDCL} + DP_T}$$

DPLL(T): OVERVIEW BY EXAMPLE

- Given the theory T of linear integer arithmetic, assume we want to check satisfiability of the following formula in CNF:

$$\varphi : \quad (\neg(x \geq 0) \vee \neg(x \geq 3)) \wedge x > 5 \wedge x \geq 0$$

- We associate a propositional variable with every atom in the theory:

$$\varphi : \quad (\neg(\underbrace{x \geq 0}_{p}) \vee \neg(\underbrace{x \geq 3}_{q})) \wedge \underbrace{x > 5}_{r} \wedge \underbrace{x \geq 0}_{p} \quad ,$$

- so we get the **propositional skeleton** of φ:

$$(\neg p \vee \neg q) \wedge r \wedge p$$

Which of the following is true?

1. If the propositional skeleton is satisfiable, so is the original formula.
2. If the propositional skeleton is unsatisfiable, so is the original formula.

DPLL(T): OVERVIEW BY EXAMPLE

- Let us apply CDCL to solve the formula $(\neg p \vee \neg q) \wedge r \wedge p$.
- CDCL returns SAT with $\alpha = [p \mapsto \mathit{true}, q \mapsto \mathit{false}, r \mapsto \mathit{true}]$.
- We unfold the encodings of the propositional variables:

$$[\overbrace{(x \geq 0)}^{p} \mapsto \mathit{true}, \overbrace{(x \geq 3)}^{q} \mapsto \mathit{false}, \overbrace{(x > 5)}^{r} \mapsto \mathit{true}]$$

- We send the following formula to DP_T:

$$\varphi' : \quad x \geq 0 \wedge \neg(x \geq 3) \wedge x > 5$$

- DP_T returns UNSAT, but with which lemma?
    - A naive solver DP_T would return $\neg\varphi'$ or, equivalently, $\neg(x \geq 0) \vee x \geq 3 \vee \neg(x > 5)$.
    - A less naive solver would realize that $\neg(x \geq 3)$ and $x > 5$ cannot hold together, so it would return $x \geq 3 \vee \neg(x > 5)$.

DPLL(T): OVERVIEW BY EXAMPLE

- Assume that DP_T returns **UNSAT** with $x \geq 3 \vee \neg(x > 5)$.
- We can apply the previous encoding

$$\underbrace{x \geq 3}_{q} \vee \neg(\underbrace{x > 5}_{r}) \qquad \text{so as to get} \qquad q \vee \neg r$$

New clause!

- Now CDCL determines satisfiability of:

$$(\neg p \vee \neg q) \wedge r \wedge p \wedge \overbrace{(q \vee \neg r)}$$

- CDCL returns **UNSAT**, and so does DPLL(T). The formula φ is unsatisfiable.

DPLL(T): FORMAL DEFINITIONS

- Assume we want to check satisfiability of a T-formula φ.
- Let $at(\varphi)$ denote the set of atoms in φ.
- Let Ats be a finite set of atoms such that $at(\varphi) \subseteq Ats$.
- A **boolean encoding** e is an injective mapping from atoms in Ats to propositional variables.
- Given an atom $a \in Ats$, we say that $e(a)$ is the encoder of a.

Example

Given $\varphi : x \geq 0 \wedge y \geq x \Rightarrow y \geq 0$ and

$Ats = \{x \geq 0, y \geq x, y \geq 0\}$

An encoding would be $e = \begin{bmatrix} x \geq 0 & \mapsto & p_1, \\ y \geq x & \mapsto & p_2, \\ y \geq 0 & \mapsto & p_3 \end{bmatrix}$.

DPLL(T): FORMAL DEFINITIONS

- Given a formula φ and an encoding e, the **propositional skeleton** of φ, denoted by $e(\varphi)$, is the propositional formula that results from replacing every atom in φ with its encoder.

Example

$$e(\varphi): \quad p_1 \land p_2 \Rightarrow p_3$$

DPLL(T): FORMAL DEFINITIONS

- Assume a partial assignment α to the encoders in $e(\varphi)$, and p one of these encoders. We define $Th(p, \alpha)$ as follows:

$$Th(p, \alpha) = \begin{cases} e^{-1}(p) & \text{if } \alpha(p) = \mathit{true} \\ \neg e^{-1}(p) & \text{if } \alpha(p) = \mathit{false} \end{cases}$$

Example

Given $\alpha = [p_1 \mapsto \mathit{false}, p_3 \mapsto \mathit{true}]$, we get:

$$\begin{aligned} Th(p_1, \alpha) &= \neg(x \geq 0) \\ Th(p_3, \alpha) &= y \geq 0 \end{aligned}$$

DPLL(T): formal definitions

- Assume an assignment α to the encoders in $e(\varphi)$.
- Let $\{p_1, \ldots, p_n\}$ be the set of propositional variables assigned by α.
- We denote by $\widehat{Th}(\alpha)$ the following formula:

$$\widehat{Th}(\alpha) = Th(p_1, \alpha) \wedge \cdots \wedge Th(p_n, \alpha)$$

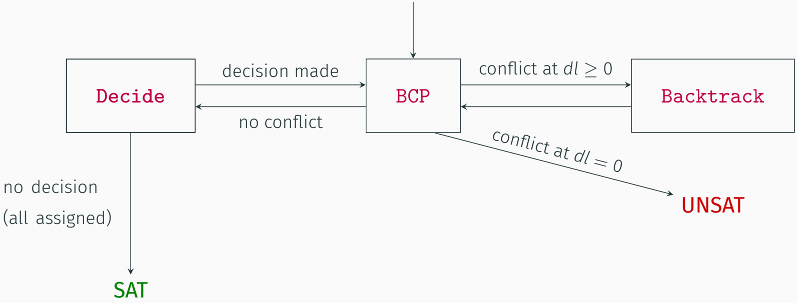
Example

Given $\alpha = [p_1 \mapsto \mathit{false}, p_3 \mapsto \mathit{true}]$, we get:

$$\widehat{Th}(\alpha) = \neg(x \geq 0) \wedge y \geq 0$$

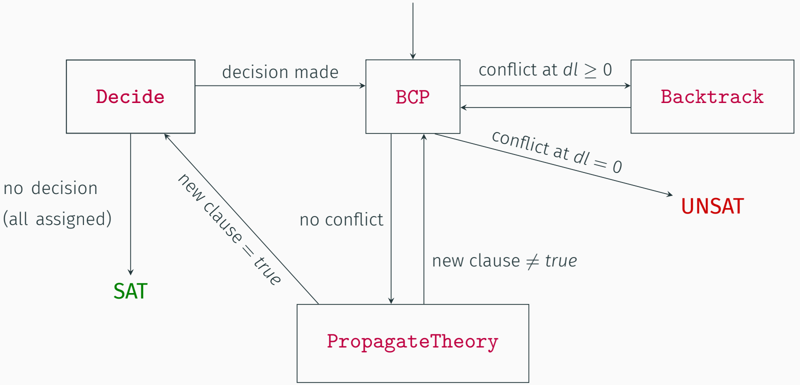
DPLL(T): general scheme

Let us recall the outline of CDCL:



DPLL(T): GENERAL SCHEME

We add a new phase: PropagateTheory



DPLL(T): `PropagateTheory`

It tries to deduce new T-implied literals and sends them back to CDCL. This is known as **theory propagation**.

1. Take the current partial assignment α and build $\widehat{Th}(\alpha)$.
2. Send the formula $\widehat{Th}(\alpha)$ to DP_T, which yields a lemma φ in return.
3. Let C be the encoding of that lemma. That is, $C = e(\varphi)$.
4. If C is *true*, jump to `Decide`.
5. Otherwise, add C to the current set of clauses and jump back to `BCP`.

Is this correct?

The algorithm is correct if DP_T satisfies these conditions:

- The lemma returned by DP_T is T-valid, or at least implied by the formula given as input to $DPLL(T)$.
- The atoms in that lemma belong to Ats.
    - Ats is finite, so termination is guaranteed in this way.
- If $\widehat{Th}(\alpha)$ is unsatisfiable and α is a total assignment, C is required to be **conflicting** under α.
    - If α is partial, C should be conflicting, but this is not necessary to guarantee correctness.
- If $\widehat{Th}(\alpha)$ is satisfiable, then either C is *true*, or C has to force an of application of unit rule after returning to `BCP`.
    - This is to guarantee termination.

DPPL(T): EXAMPLE

- Prove that the following formula is valid under the theory of equality:

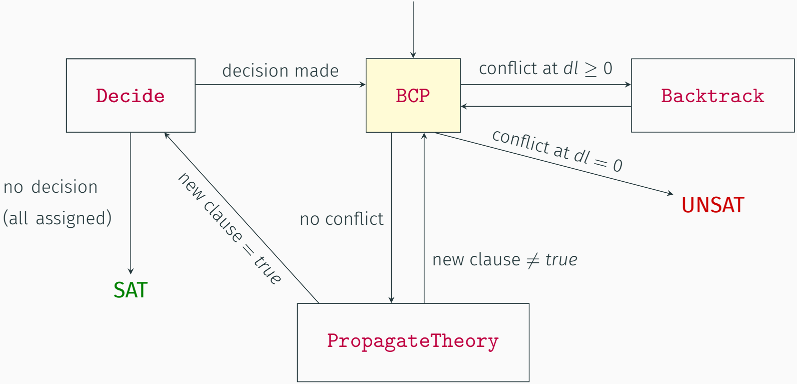
$$\varphi : \quad (x = y \wedge z = w \wedge (z = x \vee z = y)) \Rightarrow y = w$$

- Let us check whether $\neg\varphi$ is unsatisfiable. If it is, φ is valid.
- After transforming φ to CNF:

$$\left.\begin{array}{ll} c_1 : & x = y \\ c_2 : & z = w \\ c_3 : & z = x \vee z = y \\ c_4 : & \neg(y = w) \end{array}\right\} \xrightarrow{encoding} \left\{\begin{array}{ll} c_1 : & p_1 \\ c_2 : & p_2 \\ c_3 : & p_3 \vee p_4 \\ c_4 : & \neg p_5 \end{array}\right.$$

What implication graph do we get if we apply BCP?

DPLL(T): EXAMPLE



We start at BCP and apply the unit rule three times to obtain the following graph:

p_1@0 p_2@0 $\neg p_5$@0

DPLL(T): example

Current state:

c_1 :	p_1	satisfied
c_2 :	p_2	satisfied
c_3 :	$p_3 \vee p_4$	
c_4 :	$\neg p_5$	satisfied

- No conflicting clauses, so we go to `PropagateTheory`.

What is $\widehat{Th}(\alpha)$? Is it T-satisfiable?

- .

DPLL(T): EXAMPLE

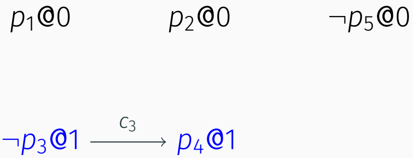
- Assume that DP_T returns SAT with *true* as lemma.
- We go back to `Decide`, which picks p_3 and assigns the value *false* to it. We add $\neg p_3$@1 to the implication graph.

DPLL(T): EXAMPLE

Current state:

c_1 :	p_1	satisfied
c_2 :	p_2	satisfied
c_3 :	$p_3 \vee p_4$	unit
c_4 :	$\neg p_5$	satisfied

- Back to BCP, which adds vertex p_4@1:



p_1@0 p_2@0 $\neg p_5$@0

$\neg p_3$@1 $\xrightarrow{c_3}$ p_4@1

DPLL(T): EXAMPLE

Current state:

c_1 :	p_1	satisfied
c_2 :	p_2	satisfied
c_3 :	$p_3 \vee p_4$	satisfied
c_4 :	$\neg p_5$	satisfied

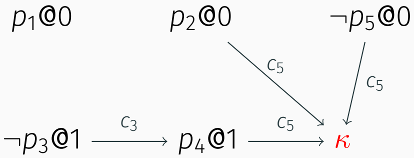
- Back to `PropagateTheory`.
- $\widehat{Th}(\alpha)$: $x = y \wedge z = w \wedge \neg(z = x) \wedge z = y \wedge \neg(y = w)$
- DP_T returns **UNSAT**.
- Assume it yields lemma $z = w \wedge z = y \Rightarrow y = w$, which is transformed and encoded as $\neg p_2 \vee \neg p_4 \vee p_5$.
- We add a new clause $c_5 : \neg p_2 \vee \neg p_4 \vee p_5$ and go back to `BCP`.

DPLL(T): EXAMPLE

Current state:

c_1 :	p_1	satisfied
c_2 :	p_2	satisfied
c_3 :	$p_3 \vee p_4$	satisfied
c_4 :	$\neg p_5$	satisfied
c_5 :	$\neg p_2 \vee \neg p_4 \vee p_5$	conflicting

- The new clause is conflicting, as expected.



Which level do we backtrack to?

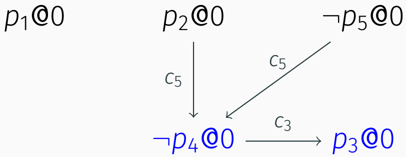
-

DPLL(T): EXAMPLE

Current state:

c_1 :	p_1	satisfied
c_2 :	p_2	satisfied
c_3 :	$p_3 \vee p_4$	
c_4 :	$\neg p_5$	satisfied
c_5 :	$\neg p_2 \vee \neg p_4 \vee p_5$	unit

After BCP:



DPLL(T): EXAMPLE

Current state:

c_1 :	p_1	satisfied
c_2 :	p_2	satisfied
c_3 :	$p_3 \vee p_4$	satisfied
c_4 :	$\neg p_5$	satisfied
c_5 :	$\neg p_2 \vee \neg p_4 \vee p_5$	satisfied

- Back to `PropagateTheory`.
- $\widehat{Th}(\alpha)$: $x = y \wedge z = w \wedge z = x \wedge \neg(z = y) \wedge \neg(y = w)$
- Again, DP_T returns **UNSAT**.
- Assume it yields lemma $x = y \wedge z = x \Rightarrow z = y$, which is transformed and encoded as $\neg p_1 \vee \neg p_3 \vee p_4$.

DPLL(T): EXAMPLE

Current state:

c_1 :	p_1	satisfied
c_2 :	p_2	satisfied
c_3 :	$p_3 \vee p_4$	satisfied
c_4 :	$\neg p_5$	satisfied
c_5 :	$\neg p_2 \vee \neg p_4 \vee p_5$	satisfied
c_6 :	$\neg p_1 \vee \neg p_3 \vee p_4$	conflicting

- We have found a conflict at level 0.
- Therefore, the algorithm finishes and returns UNSAT.

Combining theories

- Sometimes we are interested in deciding satisfiability of formulas mixing terms from different theories.

Example

The following formula:

$$f(2x) = 3 \wedge x + 7 = 1$$

contains linear arithmetic and an uninterpreted function f.

- The combination of two theories contains the union of signatures and axioms of each theory.

Assume that T_1 and T_2 are two decidable theories.

Is their combination decidable?

Satisfiability in a combined theory

- Satisfiability problem applied to the combination of two theories T_1 and T_2 might be **undecidable**.
  - even if T_1 and T_2 are decidable separately!
- **Nelson-Oppen** procedure: solves this problem for those theories satisfying the following restrictions:
  - T_1 and T_2 are quantifier-free theories with equality.
  - T_1 and T_2 are decidable separately.
  - T_1 are T_2 have disjoint signatures (except equality).
  - T_1 are T_2 are interpreted over an infinite domain.

Convex theories

- We sketch a simpler variant of Nelson-Oppen algorithm that poses an additional condition: T_1 and T_2 have to be **convex**.
- A theory T is **convex** if for every conjunctive formula φ, the T-validity of

  $$\varphi \Rightarrow x_1 = y_1 \vee \cdots \vee x_n = y_n$$

  implies that the following is valid for some $i \in \{1..n\}$:

  $$\varphi \Rightarrow x_i = y_i$$

Convex theories

Example

Linear integer arithmetic is not convex. The following holds:

$$x \geq 1 \land x \leq 2 \Rightarrow x = 1 \lor x = 2$$

But it does not hold either

$$x \geq 1 \land x \leq 2 \Rightarrow x = 1, \text{ or}$$
$$x \geq 1 \land x \leq 2 \Rightarrow x = 2$$

Would the previous counterexample be appliable under linear **real** arithmetic?

Sketch of Nelson-Oppen procedure for convex theories

- Assume the following conjunction. Is it satisfiable?

$$f(x) \leq y \land f(z) \geq y \land x \geq z \land z \geq x \land y - f(x) \neq 0$$

- We ensure that each atom contains symbols of only one theory (purification).
  - We introduce new variables if necessary.
  - Purification preserves satisfiability.

$$\underbrace{a \leq y \land b \geq y \land x \geq z \land z \geq x \land y - a \neq 0}_{\textit{linear real arithmetic}} \land \underbrace{a = f(x) \land b = f(z)}_{\textit{uninterpreted functions}}$$

Sketch of Nelson-Oppen procedure for convex theories

Linear real arithmetic

$$a \leq y$$
$$b \geq y$$
$$x \geq z$$
$$z \geq x$$
$$y - a \neq 0$$

Uninterpreted functions

$$a = f(x)$$
$$b = f(z)$$

1. If the set of formulas on either side is unsatisfiable, return UNSAT.
2. If a set of formulas implies an equality between variables not implied by the other set, propagate the equality to the other set and step back to 1.
3. If no more equalities are propagated, return SAT.

Sketch of Nelson-Oppen procedure for convex theories

Linear real arithmetic

$$
\begin{aligned}
a &\leq y \\
b &\geq y \\
x &\geq z \\
z &\geq x \\
y - a &\neq 0 \\
x &= z
\end{aligned}
$$

Uninterpreted functions

$$
\begin{aligned}
a &= f(x) \\
b &= f(z)
\end{aligned}
$$

- In this case, both sides are satisfiable.
- In the left-hand side, $x \geq z$ and $z \geq x$ implies $z = x$, which we propagate to the other side.

Sketch of Nelson-Oppen procedure for convex theories

Linear real arithmetic

$$
\begin{aligned}
a &\leq y\\
b &\geq y\\
x &\geq z\\
z &\geq x\\
y - a &\neq 0\\
x &= z
\end{aligned}
$$

Uninterpreted functions

$$
\begin{aligned}
a &= f(x)\\
b &= f(z)\\
x &= z\\
a &= b
\end{aligned}
$$

- Both sides are still satisfiable separately.
- Now, in the right-hand side, we can infer $a = b$. We propagate it to the left-hand side.

Sketch of Nelson-Oppen procedure for convex theories

Linear real arithmetic

$$\begin{aligned} &a \leq y \\ &b \geq y \\ &x \geq z \\ &z \geq x \\ &y - a \neq 0 \\ &x = z \\ &a = b \end{aligned}$$

Uninterpreted functions

$$\begin{aligned} &a = f(x) \\ &b = f(z) \\ &x = z \\ &a = b \end{aligned}$$

- The left-hand side is unsatisfiable.
  - $a = y, b \geq y$ implies $a \geq y$.
  - $a \leq y, a \geq y$ implies $a = y$.
  - $a = y, y - a \neq 0$ leads to contradiction.
- Hence, return UNSAT.

Extensions of Nelson-Oppen procedure

There is an extension supporting nonconvex theories:

D. Kroening, O. Strichman
Decision procedures: an algorithmic point of view, 2nd edition
Section 10.3.2, page 234
Springer (2015)

and also some other procedures involving theories interpreted over finite domains:

C. Tinelli, C. Zarba
Combining nonstably infinite theories
Journal of Automated Reasoning, 34(3), pages 209-238, 2008

Some SMT solvers

This course

- Z3 (Microsoft Research)
    - `https://github.com/Z3Prover/z3`
- CVC4 (Stanford University & U. Iowa)
    - `http://cvc4.cs.stanford.edu/web/`
- MATHSAT 5 (F. Bruno Kessler & U. Trento)
    - `http://mathsat.fbk.eu/`
- Alt-Ergo (LRI - OCamlPro)
    - `https://alt-ergo.ocamlpro.com/`

All these solvers support the SMT-LIB 2.0 standard, which is a common language for specifying SMT problems.

More info: `http://smtlib.cs.uiowa.edu/`

SMT-LIB: AN EXAMPLE

SAT problem shown in slide 19

```
; Predicate symbol declarations
(declare-const p1 Bool)
...
(declare-const p8 Bool)

; Clauses
(assert (or (not p1) (not p2)))
(assert (or (not p1) p3))
(assert (or (not p4) (not p3) (not p5)))
(assert (or p2 p5 p6))
(assert (or p5 (not p7)))
(assert (or (not p6) p7))
(assert (or (not p5) (not p8) (not p3)))

; Check satisfiability
(check-sat)
```

Propositional variables have type `Bool`

$\neg p_1 \vee \neg p_2$

SMT-LIB: an example

- We run Z3 with the following command:

```
z3 filename.smt
```

- The `(check-sat)` command outputs:

```
sat
```

  which means that the given set of formulas is satisfiable.

- By using `(get-value)` and `(get-model)` after checking satifiability we can get a satisfying assignment.

SMT-LIB: an example

```
...
(check-sat)
; Output:
; sat

(get-value (p1 p2))
; Output:
; ((p1 true)
;  (p2 false))

(get-model)
; Output:
; (model
;   (define-fun p7 () Bool false)
;   (define-fun p2 () Bool false)
;   ...
;   (define-fun p1 () Bool true))
```

Logical connectives

true	`true`
false	`false`
$\neg p$	`(not p)`
$p_1 \wedge p_2$	`(and p1 p2)`
$p_1 \vee p_2$	`(or p1 p2)`
$p_1 \Rightarrow p_2$	`(=> p1 p2)`
$p_1 \Leftrightarrow p_2$	`(= p1 p2)`
$p_1 \oplus p_2$	`(xor p1 p2)`

Exercise

Prove that:

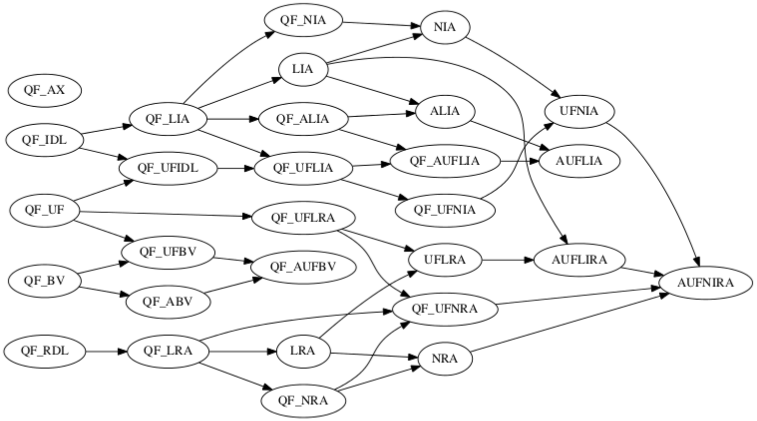
$$p \Rightarrow q \vee r$$
$$q \wedge s \Rightarrow v$$
$$r \Rightarrow \neg p \vee q$$

implies:

$$p \wedge s \Rightarrow v$$

Beyond propositional logic: theories

Z3 supports a number of theories:



Source: `http://smtlib.cs.uiowa.edu/logics.shtml`

Selecting a theory

We can choose a theory and a logic fragment with `set-logic`:

```
(set-logic QF_UF)
```

QF	Quantifier-free fragment
UF	Theory of uninterpreted functions

```
(set-logic QF_LRA)
```

QF	Quantifier-free fragment
LRA	Linear real arithmetic

Uninterpreted functions

- Theories are defined in a many-sorted logic.
- Each term has a type (sort).
- The sort of user-defined constants and uninterpreted functions must be defined in advance:

```
(declare-const c Int)
(declare-fun f (Int Int) Real)    ⟸ f : ℤ × ℤ → ℝ
```

- Linear Real Arithmetic (LRA) defines `Real` sort.
- We can declare our own sorts with `(declare-sort` *name arity*`)`.

```
(declare-sort PosInt 0) ; No sort arguments.
(declare-sort List 1)
; One sort argument (e.g. 'List Int')
```

Uninterpreted functions: example

Unsatisfiability of $a = b \wedge get(a, i) = k \wedge \neg(get(b, i) = k)$:

```
(set-logic QF_UF)

(declare-sort List 1)
(declare-sort IndexType 0)
(declare-sort ElementType 0)
(declare-const a (List ElementType))
(declare-const b (List ElementType))
(declare-const i IndexType)
(declare-const k ElementType)
(declare-fun get ((List ElementType) IndexType) ElementType)

(assert (= a b))
(assert (= (get a i) k))
(assert (not (= (get b i) k)))
(check-sat) ; returns 'unsat'
```

Arithmetic and relational operators

Integer arithmetic:

$-x$	`(- x)`
$x + y$	`(+ x y)`
$x - y$	`(- x y)`
$x * y$	`(* x y)`
$x \mathit{div}\, y$	`(div x y)`
$x \mathit{mod}\, y$	`(mod x y)`
$\|x\|$	`(abs x)`
$x \leq y$	`(<= x y)`
$x < y$	`(< x y)`
$x \geq y$	`(>= x y)`
$x > y$	`(> x y)`

Real arithmetic:

$-x$	`(- x)`
$x + y$	`(+ x y)`
$x - y$	`(- x y)`
$x * y$	`(* x y)`
x/y	`(/ x y)`
$x \leq y$	`(<= x y)`
$x < y$	`(< x y)`
$x \geq y$	`(>= x y)`
$x > y$	`(> x y)`

Linear real arithmetic: an example

```
(set-logic QF_LRA)

(declare-const x Real)
(declare-const y Real)
(assert (<= y (- 2 (* 3x))))
(assert (> x 0))
(assert (>= y 0))

(check-sat)          ; → sat
(get-value (x y))    ; → x ↦ 1/2, y ↦ 0
```

Combining theories: an example

```
(set-logic QF_UFLRA)
; Quantifier-free linear real arithmetic
; with uninterpreted functions

(declare-const x Real)
(declare-const y Real)
(declare-const z Real)
(declare-fun f (Real) Real)

(assert (<= (f x) y))
(assert (>= (f z) y))
(assert (>= x z))
(assert (>= z x))
(assert (not (= (- y (f x)) 0)))

(check-sat)         ; → unsat
```

What's next?

- So far we have seen that DPPL(T) combines the CDCL algorithm with a decision procedure DP_T, which depends on the theory we are dealing with.
- We have conceived DP_T as a **black box**.

$$lit_1 \wedge \ldots \wedge lit_n \longrightarrow \boxed{DP_T} \longrightarrow \text{SAT / UNSAT}$$

- In the following we shall study how this box works in different theories.

What's next?

Equality and uninterpreted functions

$$y = f(x, z) \land z = w \longrightarrow \boxed{DP_{QF\_UF}} \longrightarrow ??$$

Arrays

$$y = a[0] \land b = a[1 \to 4] \longrightarrow \boxed{DP_{QF\_AUF}} \longrightarrow ??$$

and we shall sketch some more: linear arithmetic, heaps, quantified formulas, etc.

Equality logic and uninterpreted functions

Equality logic with uninterpreted functions

- In QF_UF, atoms are defined by the following grammar:

$$
\begin{array}{rcll}
at & ::= & t_1 = t_2 & \{ \text{ term equality } \} \\
 & \mid & p(t_1, \ldots, t_n) & \{ \text{ predicate application } \} \\
 & & & \\
t & ::= & x & \{ \text{ variable } \} \\
 & \mid & c & \{ \text{ constant } \} \\
 & \mid & f(t_1, \ldots, t_n) & \{ \text{ function application } \}
\end{array}
$$

- We shall introduce a decision procedure for conjunctions of literals (i.e. of the form *at* or $\neg at$).

Axioms of QF_UF

- Reflexivity, symmetry and transitivity state that $=$ is an equivalence relation.

$$\begin{aligned}&\forall x.\, x = x\\ &\forall x.\, \forall y.\, x = y \Rightarrow y = x\\ &\forall x.\, \forall y.\, \forall z.\, x = y \land y = z \Rightarrow x = z\end{aligned}$$

- If f is a function symbol with n arguments, the congruence axiom specifies that the result of f depends solely on its arguments.

$$\begin{aligned}&\forall t_1, \ldots, t_n, t'_1, \ldots, t'_n.\\ &\qquad t_1 = t'_1 \land \ldots \land t_n = t'_n \Longrightarrow f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)\end{aligned}$$

- There is a similar axiom for predicate symbols.

Why are uninterpreted functions useful?

- Sometimes we want to establish a property involving some functions, regardless of their semantics.
- Example: are the following definitions equivalent?

```
int my_fun_1(int x) {
    int y;

    if (x >= 0) {
        y = f(x);
    } else {
        y = g(x);
    }

    if (x >= 1) { y = h(y); }

    return y;
}
```

```
int my_fun_2(int x) {
    if (x >= 1) {
        return h(f(x));
    } else if (x >= 0) {
        return f(x);
    } else {
        return g(x);
    }
}
```

Why are uninterpreted functions useful?

Given:

$$\begin{aligned}&x \geq 0 \Rightarrow y_1 = f(x)\\&\neg(x \geq 0) \Rightarrow y_1 = g(x)\\&x \geq 1 \Rightarrow y_2 = h(y_1)\\&\neg(x \geq 1) \Rightarrow y_2 = y_1\\&res = y_2\end{aligned}$$

$$\begin{aligned}&x \geq 1 \Rightarrow res' = h(f(x))\\&\neg(x \geq 1) \wedge (x \geq 0) \Rightarrow res' = f(x)\\&\neg(x \geq 1) \wedge \neg(x \geq 0) \Rightarrow res' = g(x);\end{aligned}$$

does it hold that $res = res'$?

Why are uninterpreted functions useful?

```
... ; declarations omitted

; Assumptions
(assert (=> (>= x 0) (= y1 (f x))))
(assert (=> (not (>= x 0)) (= y1 (g x))))
(assert (=> (>= x 1) (= y2 (h y1))))
(assert (=> (not (>= x 1)) (= y2 y1)))
(assert (= res y2))

(assert (=> (>= x 1) (= res_p (h (f x)))))
(assert (=> (and (not (>= x 1)) (>= x 0)) (= res_p (f x))))
(assert (=> (and (not (>= x 1)) (not (>= x 0)))
            (= res_p (g x))))

; Negation of conclusion
(assert (not (= res res_p)))

(check-sat)
```

Decision procedure for QF_UF

- Assume we are given a conjunction of literals, each one of the form $t_1 = t_2$ or $\neg(t_1 = t_2)$ for some terms t_1, t_2.
  - In the following we shall use $t_1 \neq t_2$ to denote $\neg(t_1 = t_2)$.
  - We leave predicate applications $p(t_1, \ldots, t_n)$ as an exercise.
- A term can be a constant, a variable or a function application $f(t_1, \ldots, t_n)$.

Example

Assume we want to check satisfiability of:

$$x = 0 \wedge y = 0 \wedge g(y) = 1 \wedge f(1) \neq f(g(x))$$

Decision procedure for QF_UF

Step 1 **Constant elimination**

- For each constant c_i in the input formula, create a fresh variable x_i.
- If $\{x_1, \ldots, x_n\}$ is the set of fresh variables generated, add the constant $x_i \neq x_j$ for each i, j such that $i \neq j$.

Example

$$x = 0 \wedge y = 0 \wedge g(y) = 1 \wedge f(1) \neq f(g(x))$$

After eliminating the constants 0 and 1 we get:

$$x = z_0 \wedge y = z_0 \wedge g(y) = z_1 \wedge f(z_1) \neq f(g(x)) \wedge z_0 \neq z_1$$

Decision procedure for QF_UF

Step 2 **Build equivalence classes**

- For each subterm t in the input formula, build a singleton equivalence class $\{t\}$.
- For each literal of the form $t_1 = t_2$ in the input formula, merge the equivalence classes of t_1 and t_2.
- Repeat until there are no more classes to be merged.

Example

We start with the following equivalence classes:

$$\{x\}, \{y\}, \{z_0\}, \{z_1\}, \{g(y)\}, \{f(z_1)\}, \{g(x)\}, \{f(g(x))\}$$

After applying $x = z_0$, $y = z_0$ and $g(y) = z_1$:

$$\{x, z_0, y\}, \{z_1, g(y)\}, \{f(z_1)\}, \{g(x)\}, \{f(g(x))\}$$

Decision procedure for QF_UF

Step 3 **Apply congruence axiom**

- For every pair of terms $f(t_1, \dots, t_n)$, $f(t'_1, \dots, t'_n)$ such that, for each $i \in \{1..n\}$, t_i and t'_i belong to the same equivalence class, merge the classes of $f(t_1, \dots, t_n)$ and $f(t'_1, \dots, t'_n)$.
- Repeat until there are no more classes to be merged.

Example

Since $x = y$, we merge the classes of $g(x)$ and $g(y)$:

$$\{x, z_0, y\}, \{z_1, g(y), g(x)\}, \{f(z_1)\}, \{f(g(x))\}$$

Now z_1 and $g(x)$ belong to the same class, so:

$$\{x, z_0, y\}, \{z_1, g(y), g(x)\}, \{f(z_1), f(g(x))\}$$

Decision procedure for QF_UF

Step 4 Discover contradictions

- For every inequality $t_1 \neq t_2$ in the formula, check whether t_1 and t_2 belong to the same equivalence class. If they do, return UNSAT.
- If no inequality satisfies this condition, return SAT.

Example

We check $z_0 \neq z_1$:

$$\{x, z_0, y\}, \{z_1, g(y), g(x)\}, \{f(z_1), f(g(x))\} \qquad \text{OK}$$

We check $f(z_1) \neq f(g(x))$:

$$\{x, z_0, y\}, \{z_1, g(y), g(x)\}, \{f(z_1), f(g(x))\} \qquad \text{return UNSAT}$$

Array theory

Array theory

- Arrays are essential in verification of software and hardware systems.
- An array can be considered as a function that maps **indices** of a set T_I to **elements** of another set T_E.

Example

The array $a = [34, 21, 43, 54, 65]$ can be represented as a function a such that $a(0) = 34$, $a(1) = 21$, etc.

- We assume that arrays have unbounded size.
  - If we want bounded arrays, we can store the length of an array a in an separate variable $length_a$.

Array theory

- Array theory provides two operations:
  - Reading: $a[i]$, where $i \in T_I$.
    It returns the element associated with the i-th index.
  - Writing: $a[i \leftarrow e]$, where $i \in T_I$ and $e \in T_E$.
    It returns another array a' with the same elements as a, except that of the i-th index, which is e.

Example

The following formula is valid in array theory:

$$\begin{aligned} &(\forall i.\, 0 \le i < n \Rightarrow a[i] = 0) \\ \wedge\ &(a' = a[n \leftarrow 0]) \\ \Rightarrow\ &(\forall i.\, 0 \le i \le n \Rightarrow a'[i] = 0) \end{aligned}$$

If arrays do not mutate, how can one verify this program?

```
int a[] = { 1, 2, 4 };
// Swap the two first components:
int e = a[0];
a[0] = a[1];
a[1] = e;
```

Axioms of array theory

1. **Element uniqueness.** For all arrays a_1, a_2 and indices i, j:

$$a_1 = a_2 \land i = j \Rightarrow a_1[i] = a_2[j]$$

2. **Read-over-write.** For every array a, element e and indices i, j:

$$(a[i \leftarrow e])[j] = \begin{cases} e & \text{if } i = j \\ a[j] & \text{otherwise} \end{cases}$$

3. **Extensionality.** For all a_1, a_2:

$$(\forall i.\, a_1[i] = a_2[i]) \Rightarrow a_1 = a_2$$

Decision problem in array theory

- Apparently, arrays can be transformed into uninterpreted functions:

Example

- Assume the formula $b = a[0 \leftarrow 23] \wedge b[1] = a[1]$.
- By introducing two uninterpreted functions a and b:

$$b(0) = 23 \wedge (\forall i.\ i \neq 0 \Rightarrow b(i) = a(i)) \wedge b(1) = a(1)$$

- **Problem:** If we want the theory to be useful in practice, we have to add quantification ($\forall$) over indices in T_I.
- Even if T_I and T_E are decidable, array theory could be **undecidable**.
- …unless we restrict the syntax of the formulas we want to prove.

Restricted syntax

- We restrict ourselves to **array properties**, which are formulas of the form $\forall i_1, \ldots, i_n.\, \varphi_I \Rightarrow \varphi_E$
  - φ_I is the **index guard**.
  - φ_E is the **value constraint**.
- Index guards have the following syntax:

$$
\begin{array}{rcl}
\varphi & ::= & true \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid at \\
at & ::= & e_1 \leq e_2 \mid e_1 = e_2 \\
e & ::= & i \mid pe \\
pe & ::= & n \mid n * x \mid pe + pe
\end{array}
$$

i	Index variable (one of $i_1, \ldots, i_n$).
n	Integer number.
x	Variable (distinct from $i_1, \ldots, i_n$).

Restricted syntax

- We restrict ourselves to **array properties**, which are formulas of the form $\forall i_1, \ldots, i_n .\, \varphi_I \Rightarrow \varphi_E$
  - φ_I is the **index guard**.
  - φ_E is the **value constraint**.
- The form of value constraints is also restricted:
  - Quantified variables $i_1, \ldots, i_n$ may **only** occur in expressions of the form $a[i]$.
  - No nested reads are allowed (e.g. $a_1[a_2[i]]$).

Which of these formulas can be transformed into array properties?

- $\forall i.\ 0 \leq i \leq n \Rightarrow a[i] = 0$
- $\forall i.\ 0 \leq i < n \Rightarrow a[i] = 0$
- $\forall i.\ i \neq 3 \Rightarrow a[i] = 0$
- $\forall i, j.\ j \leq i + 1 \Rightarrow a[i] \leq a[j]$
- $a[m] = 0$
- $\forall i.\ 0 \leq i < n - 1 \Rightarrow a[i] \leq a[i - 1]$

Decision procedure for array properties

- Assume we want to want to check the validity of:

$$
\begin{aligned}
&(\forall i.\ 0 \le i \le n - 1 \Rightarrow a[i] = 0) \\
\land\ &(a' = a[n \leftarrow 0]) \\
\Rightarrow\ &(\forall i.\ 0 \le i \le n \Rightarrow a'[i] = 0)
\end{aligned}
$$

- Validity holds iff the following is unsatisfiable:

$$
\begin{aligned}
&(\forall i.\ 0 \le i \le n - 1 \Rightarrow a[i] = 0) \\
\land\ &(a' = a[n \leftarrow 0]) \\
\land\ &(\exists i.\ 0 \le i \le n \land \neg(a'[i] = 0))
\end{aligned}
$$

Decision procedure for array properties

Step 1 Apply read-over-write axiom in order to remove array updates.

Example

We remove $a' = a[n \leftarrow 0]$ from the previous formula:

$$\begin{aligned} & (\forall i.\, 0 \leq i \leq n-1 \Rightarrow a[i] = 0) \\ \wedge\, & (a'[n] = 0) \\ \wedge\, & (\forall i.\, i \neq n \Rightarrow a'[i] = a[i]) \\ \wedge\, & (\exists i.\, 0 \leq i \leq n \wedge \neg(a'[i] = 0)) \end{aligned}$$

The guard $i \neq n$ can be transformed:

$$\begin{aligned} & (\forall i.\, 0 \leq i \leq n-1 \Rightarrow a[i] = 0) \\ \wedge\, & (a'[n] = 0) \\ \wedge\, & (\forall i.\, i \leq n-1 \vee n+1 \leq i \Rightarrow a'[i] = a[i]) \\ \wedge\, & (\exists i.\, 0 \leq i \leq n \wedge \neg(a'[i] = 0)) \end{aligned}$$

Decision procedure for array properties

Step 2 Remove existential quantifiers by replacing the quantified variables by fresh ones.

Example

We remove $\exists i$ from the formula by substituting a fresh variable x for i:

$$\begin{aligned} &(\forall i.\ 0 \leq i \leq n-1 \Rightarrow a[i] = 0) \\ \wedge\ &(a'[n] = 0) \\ \wedge\ &(\forall i.\ i \leq n-1 \vee n+1 \leq i \Rightarrow a'[i] = a[i]) \\ \wedge\ &(0 \leq x \leq n \wedge \neg(a'[x] = 0)) \end{aligned}$$

Aside: index set

- The index set of φ, denoted $\mathcal{I}(\varphi)$, is the union of:
  - The terms t appearing in expressions of the form $a[t]$, excluding quantified variables.
  - The set of top-level expressions pe (according to the grammar seen previously) occurring in index guards.
- If both are empty, then we assume that $\mathcal{I}(\varphi) = \{0\}$.

Example

If φ is the following formula:

$$\begin{aligned}
&(\forall i.\, 0 \leq i \leq n-1 \Rightarrow a[i] = 0) \\
\wedge\ &(a'[n] = 0) \\
\wedge\ &(\forall i.\, i \leq n-1 \vee n+1 \leq i \Rightarrow a'[i] = a[i]) \\
\wedge\ &(0 \leq x \leq n \wedge \neg(a'[x] = 0))
\end{aligned}$$

then $\mathcal{I}(\varphi) = \{0, n, n-1, n+1, x\}$

Decision procedure for array properties

Step 3 Remove universal quantifiers by replacing expressions of the form $\forall i.\ \psi(i)$ by $\bigwedge_{t \in \mathcal{I}(\varphi)} \psi(t)$.

Example

The quantified formula $\forall i.\ 0 \leq i \leq n-1 \Rightarrow a[i] = 0$ is replaced by:

$$\begin{aligned}
&(0 \leq 0 \leq n-1 \Rightarrow a[0] = 0) \\
\wedge\ &(0 \leq n \leq n-1 \Rightarrow a[n] = 0) \\
\wedge\ &(0 \leq n-1 \leq n-1 \Rightarrow a[n-1] = 0) \\
\wedge\ &(0 \leq n+1 \leq n-1 \Rightarrow a[n+1] = 0) \\
\wedge\ &(0 \leq x \leq n-1 \Rightarrow a[x] = 0)
\end{aligned}$$

Decision procedure for array properties

Step 3 Remove universal quantifiers by replacing expressions of the form $\forall i.\ \psi(i)$ by $\bigwedge_{t \in \mathcal{I}(\varphi)} \psi(t)$.

Example

Formula $\forall i.\ i \leq n - 1 \vee n + 1 \leq i \Rightarrow a'[i] = a[i]$ is replaced by:

$$\begin{aligned}
&(0 \leq n - 1 \vee n + 1 \leq 0 \Rightarrow a'[0] = a[0]) \\
\wedge\ &(n \leq n - 1 \vee n + 1 \leq n \Rightarrow a'[n] = a[n]) \\
\wedge\ &(n - 1 \leq n - 1 \vee n + 1 \leq n - 1 \Rightarrow a'[n - 1] = a[n - 1]) \\
\wedge\ &(n + 1 \leq n - 1 \vee n + 1 \leq n + 1 \Rightarrow a'[n + 1] = a[n + 1]) \\
\wedge\ &(x \leq n - 1 \vee n + 1 \leq x \Rightarrow a'[x] = a[x])
\end{aligned}$$

Decision procedure for array properties

Example

After simplifying and removing implications with trivial guards:

$$
\begin{aligned}
& (0 \leq n-1 \Rightarrow a[0] = 0) \\
\wedge\ & (0 \leq n-1 \Rightarrow a[n-1] = 0) \\
\wedge\ & (0 \leq x \leq n-1 \Rightarrow a[x] = 0) \\
\wedge\ & (a'[n] = 0) \\
\wedge\ & (0 \leq n-1 \vee n+1 \leq 0 \Rightarrow a'[0] = a[0]) \\
\wedge\ & (a'[n-1] = a[n-1]) \\
\wedge\ & (a'[n+1] = a[n+1]) \\
\wedge\ & (x \leq n-1 \vee n+1 \leq x \Rightarrow a'[x] = a[x]) \\
\wedge\ & (0 \leq x \leq n) \wedge\ \neg(a'[x] \neq 0)
\end{aligned}
$$

Decision procedure for array properties

Step 4 Replace array reads by uninterpreted functions:

Example

$$
\begin{aligned}
&(0 \leq n - 1 \Rightarrow a(0) = 0)\\
\wedge\ &(0 \leq n - 1 \Rightarrow a(n-1) = 0)\\
\wedge\ &(0 \leq x \leq n - 1 \Rightarrow a(x) = 0)\\
\wedge\ &(a'(n) = 0)\\
\wedge\ &(0 \leq n - 1 \vee n + 1 \leq 0 \Rightarrow a'(0) = a(0))\\
\wedge\ &(a'(n-1) = a(n-1))\\
\wedge\ &(a'(n+1) = a(n+1))\\
\wedge\ &(x \leq n - 1 \vee n + 1 \leq x \Rightarrow a'(x) = a(x))\\
\wedge\ &(0 \leq x \leq n) \wedge\ \neg(a'(x) = 0)
\end{aligned}
$$

Step 5 Apply decision procedure `QF_UF` for equality and uninterpreted functions.

Decision procedure for array properties

```
; ... declarations ommited

(assert (=> (<= 0 (- n 1)) (= (a 0) 0)))
(assert (=> (<= 0 (- n 1)) (= (a (- n 1)) 0)))
(assert (=> (and (<= 0 x) (<= x (- n 1))) (= (a x) 0)))
(assert (= (ap n) 0))
(assert (=> (or (<= 0 (- n 1)) (<= (+ n 1) 0))
            (= (ap 0) (a 0))))
(assert (= (ap (- n 1)) (a (- n 1))))
(assert (= (ap (+ n 1)) (a (+ n 1))))
(assert (=> (or (<= x (- n 1)) (<= (+ n 1) x))
            (= (ap x) (a x))))
(assert (and (<= 0 x) (<= x n)))
(assert (not (= (ap x) 0)))
(check-sat) ; outputs unsat, so the formula is valid
```

Array theory in Z3

SMT-LIB provides array-related functions when selecting one of the following logics:

- `QF_AX`: arrays with extensionality axiom
- `QF_ALIA`: arrays + linear integer arithmetic
- `QF_AUFLIA`: arrays + uninterpreted functions + linear integer arithmetic
- etc.

More info: `http://smtlib.cs.uiowa.edu/logics.shtml`

Array theory in Z3

- These logics introduce the `Array` sort:

```
(declare-const a (Array Int Bool))
    ; Type of indices => Int
    ; Type of elements => Bool
```

- They introduce the following functions:

$a[i]$	`(select a i)`
$a[i \leftarrow x]$	`(store a i x)`

- We can also specify quantified properties:

$$\texttt{(forall ((}\mathit{var}_1\ \mathit{sort}_1\texttt{) ... (}\mathit{var}_n\ \mathit{sort}_n\texttt{)) }\mathit{formula}\texttt{)}$$

$$\texttt{(exists ((}\mathit{var}_1\ \mathit{sort}_1\texttt{) ... (}\mathit{var}_n\ \mathit{sort}_n\texttt{)) }\mathit{formula}\texttt{)}$$

Example

```
(set-logic QF_ALIA)
(declare-const a (Array Int Int))
(declare-const ap (Array Int Int))
(declare-const n Int)

; Formula: ∀i. 0 ≤ i < n ⇒ a[i] = 0
(assert (forall ((i Int))
           (=> (and (<= 0 i) (< i n)) (= (select a i) 0))))

; Formula: a′ = a[n ← 0]
(assert (= ap (store a n 0)))

; Formula: ¬(∀i. 0 ≤ i ≤ n ⇒ a′[i] = 0)
(assert (not (forall ((i Int))
    (=> (and (<= 0 i) (<= i n)) (= (select ap i) 0)))))

(check-sat)
```

More efficient algorithms

- The algorithm explained before applies the read-over-write axiom **exhaustively** to all array updates.
- Some of these applications might be unnecessary.
- There are more sophisticated algorithms which apply these rules **incrementally** in combination with DPLL(T).

J. Christ, J. Hoenicke
Weakly Equivalent Arrays
10th International Symposium on Frontiers of Combining Systems (FroCoS 2015)

More quantifier-free theories

Linear arithmetic

- It is defined by the following syntax:

$$\begin{array}{rcll} at & ::= & t_1 \leq t_2 \mid t_1 < t_2 \mid t_1 = t_2 & \\ & & & \\ t & ::= & x & \{ \text{ variable } \} \\ & \mid & c & \{ \text{ constant } \} \\ & \mid & t_1 + t_2 & \{ \text{ addition } \} \end{array}$$

- Linear integer arithmetic (LIA): variables can take only integer values.
- Linear real arithmetic (LRA): variables can take real values.

Decision procedures for LIA and LRA

- Linear real arithmetic
  - Simplex algorithm without an objective function.
  - Fourier-Motzkin variable elimination.
- Linear integer arithmetic
  - Branch and bound combined with LRA.
  - Omega test.
- Difference logic
  - Constraints of the form $x - y \leq c$.
  - Decidable in polynomial time.

D. Kroening, O. Strichman
Decision procedures: an algorithmic point of view, 2nd edition
Chapter 5, page 97
Springer (2015)

Nonlinear arithmetic

- Syntax:

$$at \quad ::= \quad t_1 \leq t_2 \mid t_1 < t_2 \mid t_1 = t_2$$

$$\begin{array}{rcll} t & ::= & x & \{ \text{ variable } \} \\ & \mid & c & \{ \text{ constant } \} \\ & \mid & t_1 + t_2 & \{ \text{ addition } \} \\ & \mid & t_1 * t_2 & \{ \text{ multiplication } \} \end{array}$$

- Nonlinear real arithmetic is decidable.
  - Tarski decision procedure.
  - Cylindrical algebraic decomposition.
- Nonlinear integer arithmetic is undecidable.

Bitvectors

Are these programs equivalent?

```
bool f(int x, int y) {
    return x - y > 0;
}
```

```
bool f(int x, int y) {
    return x > y;
}
```

- From the point of view of ”mathematical” integers, they are obviously equivalent.
- But they are not if `y` is too large, because we might run into arithmetic overflow.

Bitvector logic

- Bitvector logic deals with numbers as if they were sequences of bits, so overflow is taken into account.

```
(set-logic QF_BV)

(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
    (not (= (bvugt (bvsub x y) (_ bv0 64))  ; x - y < 0
            (bvugt x y))))                  ; x < y

(check-sat) ; it returns sat!
```

Decision procedures for bitvector logic

- A vector of length *n* can be represented with *n* boolean variables.
- Arithmetic, logic and relational operators can be axiomatized bitwise.
- As a consequence, a satisfiability problem in bitvector logic can be translated into propositional logic (SAT).

Pointer logic

- It allows us to reason about programs that use pointers.
- For example, assume the following C program fragment:

```
int x = 5;
int *y = &x;
*y = 3;
```

- Can we prove that `x == 3` after the execution?
- **Pointer logic** allows us to express properties involving pointers in a **heap**.

Heaps and pointer logic

- A heap H is a mapping from **pointers** (memory locations) to integers.
- We also denote pointers by integer numbers.
- We have a mapping L that specifies the memory location of each variable.
- In our previous example, we have the following pair (L, H) after executing the program:

$$L = \begin{bmatrix} x & \mapsto & 25 \\ y & \mapsto & 26 \end{bmatrix} \qquad H = \begin{bmatrix} 25 & \mapsto & 3 \\ 26 & \mapsto & 25 \end{bmatrix}$$

Heaps as arrays

Both mappings H and L can be encoded as arrays:

- Initial H_0 and L_0:

```
(declare-sort Var 0)
(declare-const H0 (Array Int Int))
(declare-const L0 (Array Var Int))
```

- After initializing x with 5 we get H_1 and L_1:

```
(declare-const H1 (Array Int Int))
(declare-const L1 (Array Var Int))
(declare-const x Var)
(declare-const p1 Int)  ; Location of variable x
(assert (= L1 (store L0 x p1))) ; $L_1 = L_0[x \leftarrow p_1]$
(assert (= H1 (store H0 p1 5))) ; $H_1 = H_0[p_1 \leftarrow 5]$
```

Heaps as arrays

- After initializing `y` with `*x`:

```
(declare-const H2 (Array Int Int))
(declare-const L2 (Array Var Int))
(declare-const y Var)
(assert (not (= x y)))
(declare-const p2 Int)  ; Location of variable y
(assert (not (= p1 p2)))
(assert (= L2 (store L1 y p2))) ; $L_2 = L_1[x \leftarrow p_2]$
(assert (= H2 (store H1 p2 (select L1 x))))
                                ; $H_2 = H_1[p_2 \leftarrow L_1(x)]$
```

Heaps as arrays

- After executing `*y = 3`:

```
(declare-const H3 (Array Int Int))
(declare-const L3 (Array Var Int))
(assert (= L3 L2))
(assert
    (= H3 (store H2 (select H2 (select L2 y)) 3)))
                                   ; $H_3 = H_2[H_2(L_2(y)) \leftarrow 3]$
```

- Does it hold that `x == 3`?

```
(assert (not (= (select H3 (select L3 x)) 3)))
(check-sat)  ; → UNSAT, so it holds
```

Applications of pointer logic

- Verification of data structures.
- Verification in low-level languages.
- Nullity analysis.
- Alias analysis.
- Analysis of dangling pointers.
- etc.

D. Kroening, O. Strichman
Decision procedures: an algorithmic point of view, 2nd edition
Chapter 8, page 173
Springer (2015)

Logic with quantifiers

Why quantifiers?

- Uninterpreted functions have no semantics.
- The only property we know about them is the axiom of congruence.
- Assume we have an uninterpreted function *add*, and we know that it is commutative.
- Without this information, we could not prove unsatisfiability of the following:

$$add(f(a), c)) \neq add(c, f(a))$$

What if we add an assumption saying that *add* is commutative?

$$(add(x,y) = add(y,x)) \wedge (add(f(a),c) \neq add(c,f(a)))$$

Why quantifiers?

- We have to convey that $add(x, y) = add(y, x)$ holds **for any** x and y:

$$(\forall x, y.\ add(x, y) = add(y, x)) \land (add(f(a), c) \neq add(c, f(a)))$$

- A solver should instantiate the quantified fact with $[x \mapsto f(a), y \mapsto c]$ so as to get:

$$(add(f(a), c) = add(c, f(a))) \land (add(f(a), c) \neq add(c, f(a)))$$

which now can be proved unsatisfiable by `QF_UF`.

Quantified formulas

- Arbitrary first-order theories with quantifiers are undecidable.
- However, there are decidable subsets. For example:
  - Quantified boolean formulas: $\forall x.\ (x \vee \exists y.\ (y \vee \neg x))$.
  - Quantified disjunctive linear arithmetic. $\forall x.\ \exists y.\ 2 * x \leq y$.
  - Effectively propositional logic: $\exists x_1, \ldots, x_n.\ \forall y_1, \ldots, y_m.\ \varphi$.
- We shall also tackle general quantification later.

But before...

- In the following we assume that the formula to be proved unsatisfiable does not contain free variables.
- If it contains free variables, we can transform it by adding existential quantifiers:

$$2 * x \leq y \text{ is transformed into } \exists x, y.\ 2 * x \leq y$$

- This transformation preserves satisfiability.

Quantifier elimination

- Quantifier elimination allows us to decide the following subsets:
  - Quantified boolean formulas: $\forall x.\ (x \vee \exists y.\ (y \vee \neg x))$.
  - Quantified disjunctive linear arithmetic. $\forall x.\ \exists y.\ 2 * x \leq y$.

Step 1 Transform the input formula into prenex normal form.

Step 2 Eliminate universal quantifiers:

- $\forall x.\ \varphi$ becomes $\neg\exists x.\ \neg\varphi$.

Step 3 Eliminate quantifiers starting from the innermost ones.

- We replace each formula $\exists x.\phi$ by an equivalent one ϕ' such that ϕ' does not contain x.

Step 4 Check satisfiability of the resulting quantifier-free formula.

How to eliminate quantifiers?

- **Quantified boolean formulas**: we replace $\exists x.\ \varphi$ by $\varphi[x/\mathit{true}] \vee \varphi[x/\mathit{false}]$
  - $\varphi[x/\mathit{true}]$ denotes the result of replacing all ocurrences of x in φ by *true*.
  - Analogously with $\varphi[x/\mathit{false}]$.

Example

Is $\exists x.\ \forall y.\ x \Rightarrow y$ satisfiable?

$$\begin{aligned}
& \exists x.\ \forall y.\ x \Rightarrow y \\
\rightsquigarrow\ & \exists x.\ \neg\exists y.\ \neg(x \Rightarrow y) \\
\rightsquigarrow\ & \exists x.\ \neg(\neg(x \Rightarrow \mathit{true}) \vee \neg(x \Rightarrow \mathit{false})) \\
\rightsquigarrow\ & \neg(\neg(\mathit{true} \Rightarrow \mathit{true}) \vee \neg(\mathit{true} \Rightarrow \mathit{false})) \\
& \vee \neg(\neg(\mathit{false} \Rightarrow \mathit{true}) \vee \neg(\mathit{false} \Rightarrow \mathit{false}))
\end{aligned}$$

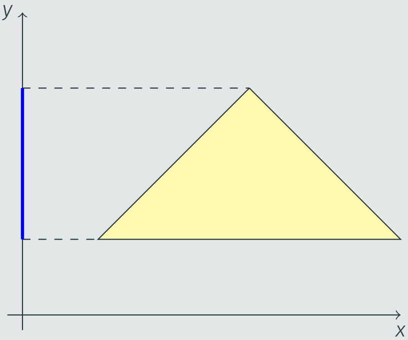
which is satisfiable.

How to eliminate quantifiers?

- **Disjunctive linear arithmetic**: we eliminate $\exists x.\varphi$ by projecting on the remaining axes.
  - This can be done by Fourier-Motzkin algorithm.

Example

Given $\exists y, x.\ y \leq x \wedge x + y \leq 6 \wedge y \geq 1$:



By projecting on the y axis we get:

$$\exists y.\ y \geq 1 \wedge y \leq 3$$

which is satisfiable.

Effectively propositional logic

- It consists in formulas of the form:

$$\exists x_1, \ldots, x_n .\, \forall y_1, \ldots, y_m .\, \varphi$$

  where φ does not contain function symbols, but it may contain predicate symbols.

Effectively propositional logic

How to solve this kind of formulas?

Step 1 Remove existential quantifiers, replacing the quantified variables by constants.

Step 2 Remove universal quantifiers, replacing each formula $\forall y.\varphi$ by $\varphi[y \mapsto c_1] \wedge \varphi[y \mapsto c_2] \wedge \ldots \wedge \varphi[y \mapsto c_n]$, where $c_1, \ldots, c_n$ are the constants generated in the previous step.

Step 3 Encode each atom generated by a propositional variable.

Step 4 Apply SAT solving techniques.

Effectively propositional logic

Example

Given $\exists x_1, x_2.\ \forall x_3, x_4.\ p(x_1, x_2, x_3) \vee q(x_4)$.

- After step 1: $\forall x_3, x_4.\ p(c_1, c_2, x_3) \vee q(x_4)$.
- After step 2:

$$\begin{aligned} & (p(c_1, c_2, c_1) \vee q(c_1)) \\ \wedge \quad & (p(c_1, c_2, c_1) \vee q(c_2)) \\ \wedge \quad & (p(c_1, c_2, c_2) \vee q(c_1)) \\ \wedge \quad & (p(c_1, c_2, c_2) \vee q(c_2)) \end{aligned}$$

- After step 3: $(p_1 \vee q_1) \wedge (p_1 \vee q_2) \wedge (p_2 \vee q_1) \wedge (p_2 \vee q_2)$.
- After step 4: **SAT**.

Effectively propositional logic

Effectively propositional logic is useful for encoding properties in set theory.

Example

Assume we want to prove:

$$A \subseteq B \land B \cap C = \emptyset \Rightarrow A \cap C = \emptyset$$

we introduce uninterpreted predicates p_A, p_B and p_C so that $p_A(x)$ means $x \in A$, etc.

$$\overbrace{\forall x.(p_A(x) \Rightarrow p_B(x))}^{A \subseteq B} \land \overbrace{\neg\exists x.(p_B(x) \land p_C(x))}^{B \cap C = \emptyset} \Rightarrow \overbrace{\neg\exists x.(p_A(x) \land p_C(x))}^{A \cap C = \emptyset}$$

General quantification

- If we allow any combination of quantifiers and logic formulas, satisfiability becomes undecidable.
- There are, however, procedures that can check satisfiability in many cases.
- We assume that the input formula has been transformed into **Skolem normal form**, so that it does not contain existential quantifiers:

$$\forall x_1, \ldots, x_n .\ \varphi$$

Example

$$(\forall x, y.\ add(x, y) = add(y, x)) \wedge (add(f(a), c) \neq add(c, f(a)))$$

Triggers

- Given a formula $\forall x_1, \ldots, x_n.\ \varphi$, a **trigger** is a term that contains all quantified variables.

Example

$$(\forall x, y.\ add(x, y) = add(y, x)) \land (add(f(a), c) \neq add(c, f(a)))$$

In the first conjunct, $add(x, y)$ is a trigger.

- Whenever the solver finds a ground term that **matches** a trigger, it instantiates the formula with the matching substitution.

Trigger

Example

$$(\forall x, y.\ add(x, y) = add(y, x)) \wedge (add(f(a), c) \neq add(c, f(a)))$$

Given the trigger $add(x, y)$, there are several ground terms that match this trigger. For example:

$$\begin{array}{llcccc}
\textbf{Trigger:} & add(& x & , & y &) \\
 & & \downarrow & & \downarrow &) \\
\textbf{Ground term:} & add(& f(a) & , & c &)
\end{array}$$

By applying the substitution $[x \mapsto f(a), y \mapsto c]$ to the quantified formula, we get the literal $add(f(a), c) = add(c, f(a))$, making the formula unsatisfiable.

Triggers and E-matching

- A naive solver tries to match a trigger with a ground term that already exists in the formula.
- But such a ground term might not exist.

Example

Given

$$(\forall x.\, f(g(x)) = x) \wedge a = g(b) \wedge b = c \wedge f(a) \neq c$$

There is no ground term matching trigger $f(g(x))$.

- However, there is an algorithm (E-matching) that exploits the knowledge on equalities between terms.

Example based on `https://rise4fun.com/z3/tutorialcontent/guide#h28`

Triggers and E-matching

Example

$$(\forall x.\, f(g(x)) = x) \land a = g(b) \land b = c \land f(a) \neq c$$

- No ground term in the formula matches $f(g(x))$.
- But there is a term $f(a)$, equivalent to $f(g(b))$ according to the equalities.
- Matching $f(g(x))$ with $f(g(b))$ yields $[x \mapsto b]$.
- After instantiating the quantified formula with $[x \mapsto b]$:

$$(\forall x.\, f(g(x)) = x) \land a = g(b) \land b = c \land f(a) \neq c \land f(g(b)) = b$$

which makes the formula unsatisfiable, since $f(a) = f(g(b)) = b = c$, contradicting $f(a) \neq c$.

Patterns in Z3

- Patterns allow us to specify how quantifiers are instantiated.
- They have the following syntax:

```
(forall (vars) (! formula :pattern (pat1 ...))
```

- By default, Z3 tries to find a suitable set of patterns for each quantified formula.
  - But by specifying our own set of patterns we can make it reduce the search space.

Patterns in Z3

```
(set-option :smt.auto-config false)
(set-option :smt.mbqi false)
(declare-fun f (Int) Int)
(declare-fun g (Int) Int)
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(assert (forall ((x Int)) (! (= (f (g x)) x)
                              :pattern ((f (g x))))))
(assert (= a (g b)))
(assert (= b c))
(assert (not (= (f a) c)))
(check-sat)
```

Compare execution times with and without `:pattern`.

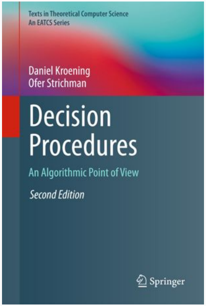
Taken from `https://rise4fun.com/z3/tutorialcontent/guide#h28`

More information

- L. de Moura, N. Bjørner
  Efficient E-Matching for SMT-Solvers
  21st International Conference on Automated Deduction, CADE-21 (2007)
- Microsoft Research
  Getting Started with Z3: A Guide
  `https://rise4fun.com/z3/tutorialcontent/guide`

Bibliography

D. Kroening, O. Strichman
Decision procedures: an algorithmic point of view, 2nd edition
Springer (2015)



Texts in Theoretical Computer Science
An EATCS Series

Daniel Kroening
Ofer Strichman

Decision Procedures

An Algorithmic Point of View

Second Edition

Springer