SMT: **satisfiability modulo theories**

Recap and overview

- In the previous lecture, we introduced **symbolic execution**.
- To be effective, symbolic execution requires an **efficient mechanism to prove path conditions**.
- **This lecture:** we explore how to handle **Boolean structures** when deciding **satisfiability modulo theories (SMT)**.
- In practice, we introduce the **foundations of SMT solvers**, which use **clever techniques to reason efficiently about Boolean and theory constraints**.

Recap: What is the SAT problem?

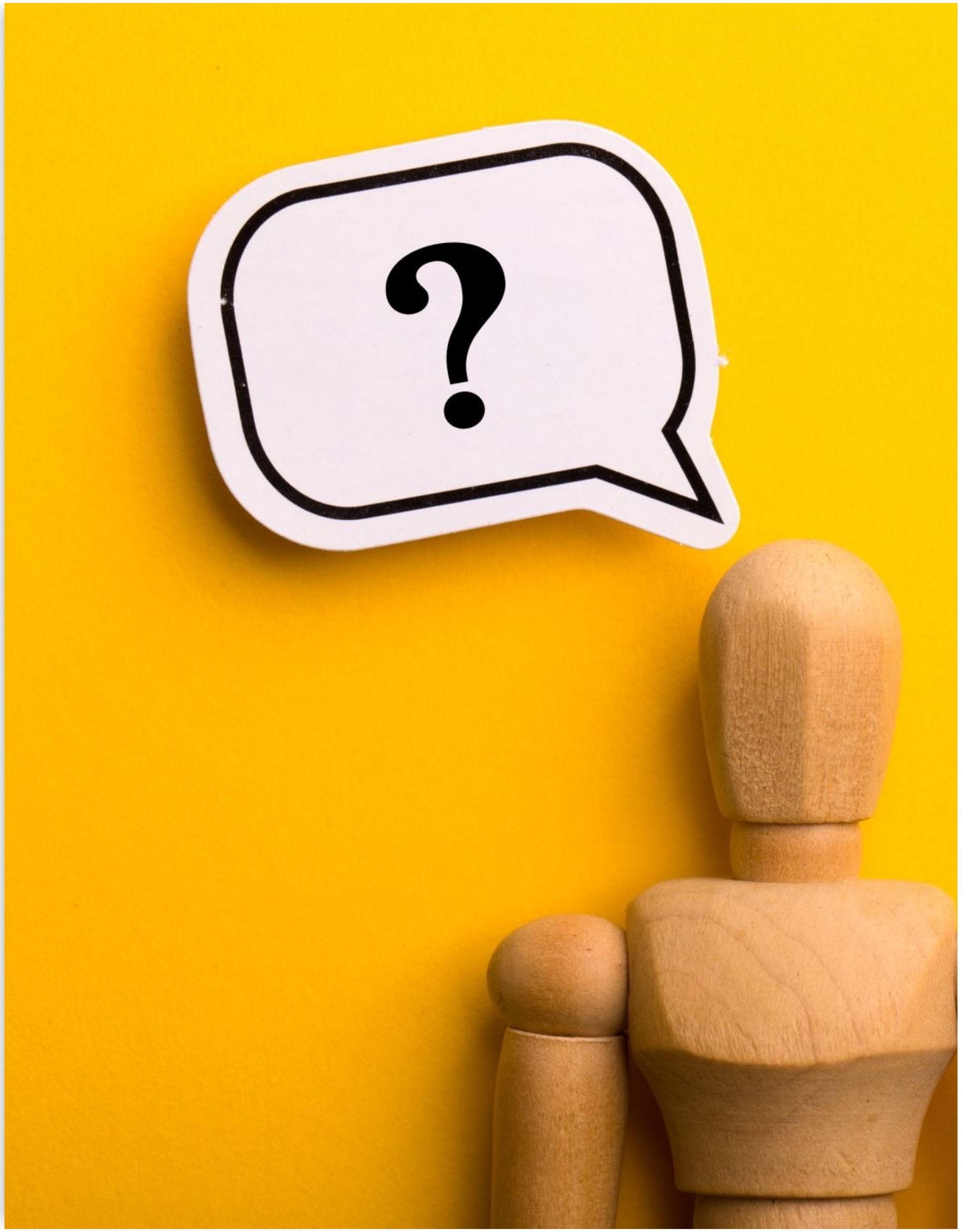
SAT stands for **Boolean Satisfiability**.

It is the problem of determining whether there exists a **truth assignment** to a set of Boolean variables that makes a given logical formula **true**.

Example: the formula

$$(p \lor q) \land (\neg p \lor r)$$

Is there an assignment of truth values to p, q, and r that makes this formula true?



Recap: What is the SAT problem?

SAT stands for **Boolean Satisfiability**.

Example: the formula

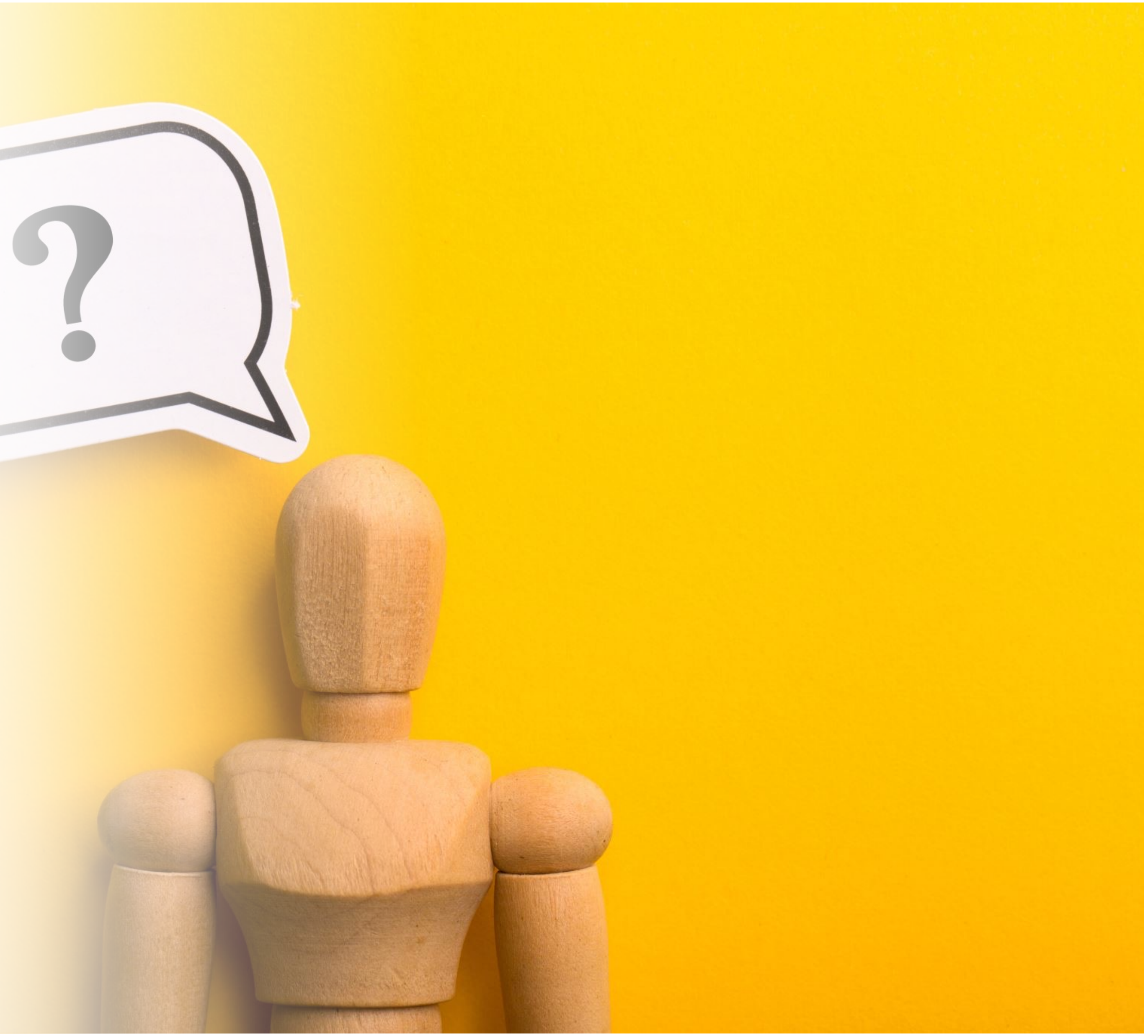
$$(p \lor q) \land (\neg p \lor r)$$

Is there an assignment of truth values to p, q, and r that makes this formula true?

Yes: Under the assignment

$$p = \mathit{false}, q = \mathit{true}, r = \mathit{true}$$

The formula evaluates to **true**, so it is **satisfiable**.



SAT: Boolean Satisfiability Problem

What is the SAT Problem?

Input: A Boolean formula

Question: *Is there an assignment of true/false values to variables that makes the formula true?*

The problem proven **NP-complete** (Cook, 1971)

SAT is hard to solve!!

- The search space is **exponential**: 2^n possible assignments for n variables
- SAT is **NP-complete**
  - No known polynomial-time algorithm for all inputs
- Many real-world problems reduce to SAT:
  - Circuit verification
  - Program analysis
  - Scheduling, planning
- Even **small changes** to input can change the solution space drastically

SAT Solver

OUTCOME

- **SAT** (satisfiable)
  - at least one assignment makes the formula true
- **UNSAT** (unsatisfiable)
  - no assignment satisfies the formula
- **UNKNOWN**
  - can’t determine (e.g., due timeout)

SMT SOLVERS

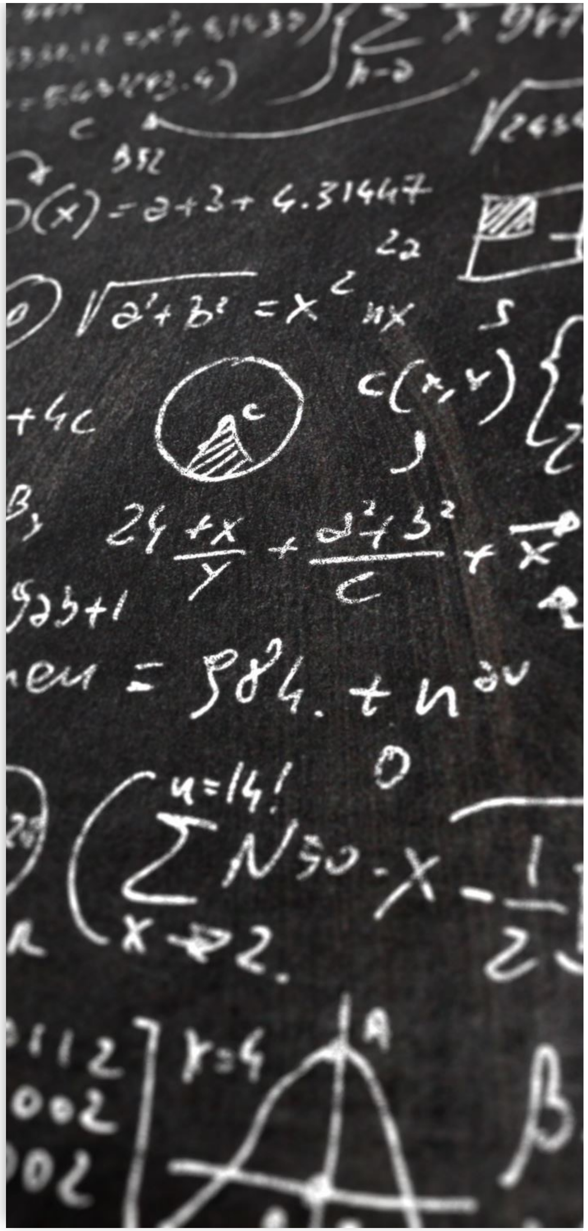
Key idea: SMT manages the **Boolean structure** of formulas, with one or more **theory solvers**, which handle the **theory-specific constraints** (e.g., arithmetic, arrays, bit-vectors).

SMT = SAT + Theories:

SAT solver manages Boolean structure,	**Theory solver** checks consistency within the chosen theory.

Operationally

- To use SAT solver, we construct a propositional formula, called **boolean abstraction**, that overapproximates satisfiability
- If boolean abstraction is **UNSAT**,
  - we are done
- If boolean abstraction is **SAT**
  - Use theory solver to check if assignment returned by **SAT** solver **is satisfiable modulo theory**
- If not, add additional **boolean constraints** (called **theory conflict clauses**) to guide the search for an assignment that is satisfiable modulo theory



Techniques to address SAT

How Modern SAT Solvers Behave

- **DPLL algorithm** and its modern variants
- **Heuristics** for variable selection
- **Preprocessing and simplification**

- Solvers like **MiniSAT**, **Z3**, and **CryptoMiniSat** make SAT feasible in many cases

DPLL

The **DPLL algorithm** (Davis–Putnam–Logemann–Loveland) is the foundation of most modern **SAT solvers**.

DPLL is a **complete backtracking-based search algorithm** for solving the **SAT problem**. It determines whether a **Boolean formula in CNF (conjunctive normal form)** is **satisfiable**.

DLLP improves brute-force search by introducing **smart pruning and inference techniques**.

Toward DPLL

Boolean Constraint Propagation

If a clause has **only one literal unassigned** (a *unit clause*), assign it in a way that satisfies the clause.

Clause: (x) the set x=true

Boolean Constraint Propagation: example

CNF Formula $\underbrace{(p_1 \vee \neg p_3 \vee \neg p_5)}_{C_1} \wedge \underbrace{(\neg p_1 \vee p_2)}_{C_2} \wedge \underbrace{(\neg p_1 \vee \neg p_3 \vee p_4)}_{C_3} \wedge \underbrace{(\neg p_1 \vee \neg p_2 \vee p_3)}_{C_5} \wedge \underbrace{(\neg p_4 \vee \neg p_2)}_{C_6}$

SAT begins with the assignment $\boldsymbol{p_1 = true}$

$$
\begin{aligned}
&(p_1 \vee \neg p_3 \vee \neg p_5) \wedge (\neg p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_3 \vee p_4) \wedge (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2)\\
\leftrightarrow\ &(\top \vee \neg p_3 \vee \neg p_5) \wedge (\bot \vee p_2) \wedge (\bot \vee \neg p_3 \vee p_4) \wedge (\bot \vee \neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2)\\
\leftrightarrow\ &\top \wedge p_2 \wedge (\neg p_3 \vee p_4) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2)\\
\leftrightarrow\ &p_2 \wedge (\neg p_3 \vee p_4) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2)
\end{aligned}
$$

Boolean Constraint Propagation: example

$$\underbrace{(p_1 \vee \neg p_3 \vee \neg p_5)}_{C_1} \wedge \underbrace{(\neg p_1 \vee p_2)}_{C_2} \wedge \underbrace{(\neg p_1 \vee \neg p_3 \vee p_4)}_{C_3} \wedge \underbrace{(\neg p_1 \vee \neg p_2 \vee p_3)}_{C_5} \wedge \underbrace{(\neg p_4 \vee \neg p_2)}_{C_6}$$

The assignment $p_1 = true\ allows\ one\ to$ reduce the original problem to the satisfaction of the formula

$$p_2 \wedge (\neg p_3 \vee p_4) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2)$$

Boolean Constraint Propagation: example

$$\underbrace{(p_1 \vee \neg p_3 \vee \neg p_5)}_{C_1} \wedge \underbrace{(\neg p_1 \vee p_2)}_{C_2} \wedge \underbrace{(\neg p_1 \vee \neg p_3 \vee p_4)}_{C_3} \wedge \underbrace{(\neg p_1 \vee \neg p_2 \vee p_3)}_{C_5} \wedge \underbrace{(\neg p_4 \vee \neg p_2)}_{C_6}$$

$$p_2 \wedge (\neg p_3 \vee p_4) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2)$$

any satisfying interpretation must contain the assignment $p_2 = true$

No choice to satisfy this formula.

The literal p_2 is called the unit literal

Boolean Constraint Propagation: example

$$\underbrace{(p_1 \lor \neg p_3 \lor \neg p_5)}_{C_1} \land \underbrace{(\neg p_1 \lor p_2)}_{C_2} \land \underbrace{(\neg p_1 \lor \neg p_3 \lor p_4)}_{C_3} \land \underbrace{(\neg p_1 \lor \neg p_2 \lor p_3)}_{C_5} \land \underbrace{(\neg p_4 \lor \neg p_2)}_{C_6}$$

$$p_2 \land (\neg p_3 \lor p_4) \land (\neg p_2 \lor p_3) \land (\neg p_4 \lor \neg p_2)$$

Set $p_2 = true$

$$\begin{aligned} & \top \land (\neg p_3 \lor p_4) \land (\neg \top \lor p_3) \land (\neg p_4 \lor \neg \top) \\ \leftrightarrow\ & (\neg p_3 \lor p_4) \land (\bot \lor p_3) \land (\neg p_4 \lor \bot) \\ \leftrightarrow\ & (\neg p_3 \lor p_4) \land p_3 \land \neg p_4 \end{aligned}$$

Boolean Constraint Propagation: example

$$\underbrace{(p_1 \vee \neg p_3 \vee \neg p_5)}_{C_1} \wedge \underbrace{(\neg p_1 \vee p_2)}_{C_2} \wedge \underbrace{(\neg p_1 \vee \neg p_3 \vee p_4)}_{C_3} \wedge \underbrace{(\neg p_1 \vee \neg p_2 \vee p_3)}_{C_5} \wedge \underbrace{(\neg p_4 \vee \neg p_2)}_{C_6}$$

$$p_2 \wedge (\neg p_3 \vee p_4) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2)$$

$$(\neg p_3 \vee p_4) \wedge p_3 \wedge \neg p_4$$

Set $p_3 = true$

$$(\neg \top \vee p_4) \wedge \top \wedge \neg p_4$$

$$\leftrightarrow (\bot \vee p_4) \wedge \neg p_4$$

$$\leftrightarrow p_4 \wedge \neg p_4$$

Overall

$$\underbrace{(p_1 \vee \neg p_3 \vee \neg p_5)}_{C_1} \wedge \underbrace{(\neg p_1 \vee p_2)}_{C_2} \wedge \underbrace{(\neg p_1 \vee \neg p_3 \vee p_4)}_{C_3} \wedge \underbrace{(\neg p_1 \vee \neg p_2 \vee p_3)}_{C_5} \wedge \underbrace{(\neg p_4 \vee \neg p_2)}_{C_6}$$

$$p_2 \wedge (\neg p_3 \vee p_4) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2)$$

$$(\neg p_3 \vee p_4) \wedge p_3 \wedge \neg p_4$$

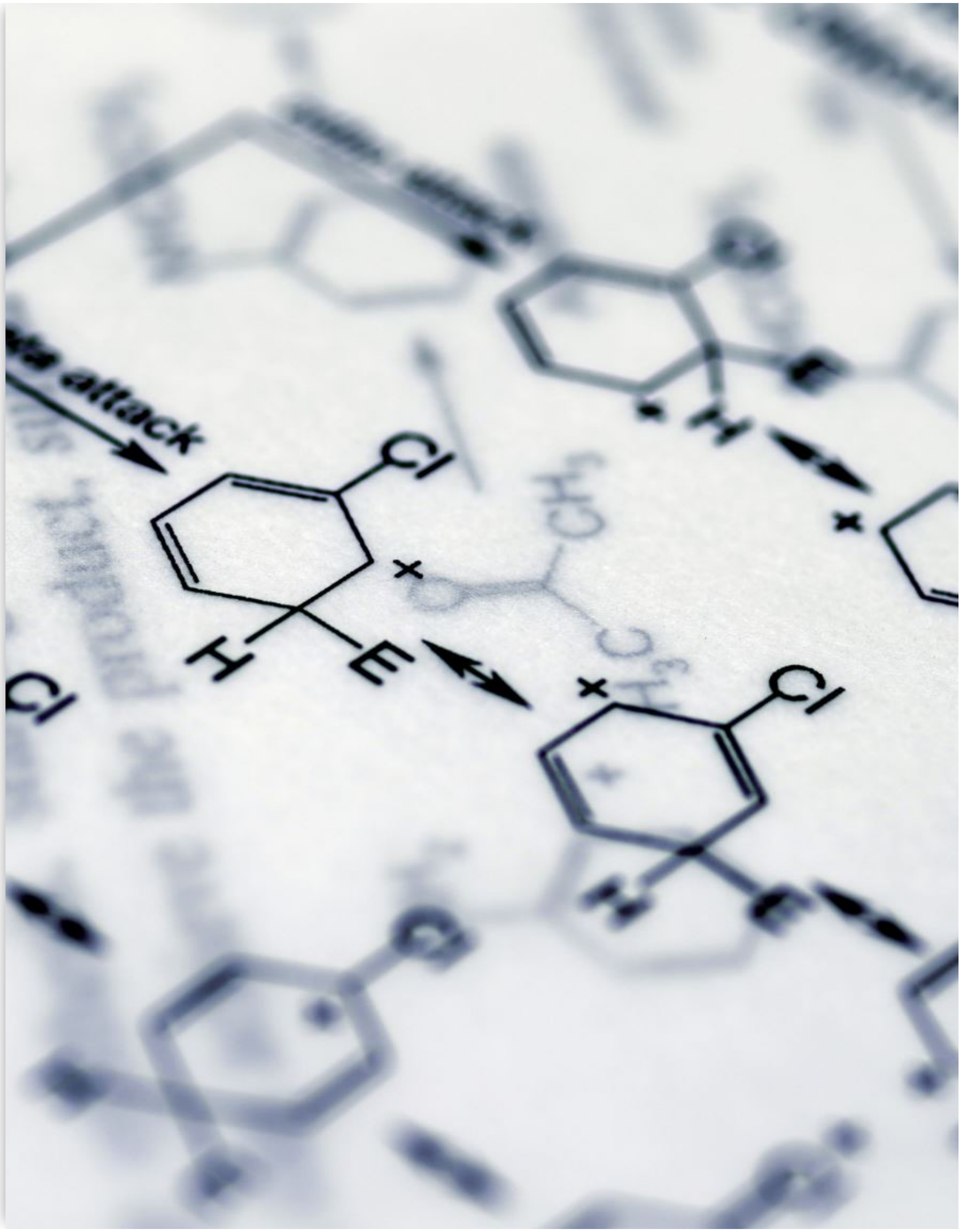
Wih the assignment $p_1 = true$

$$(\neg \top \vee p_4) \wedge \top \wedge \neg p_4$$
$$\leftrightarrow (\bot \vee p_4) \wedge \neg p_4$$
$$\leftrightarrow p_4 \wedge \neg p_4$$

the formula is not satisfiable

Unit propagation

The process we described is called Boolean constraint propagation (BCP), or sometimes unit propagation for short.



How DLLP Works

Boolean Constrait Propagation (or Unit Propagation)

- If a clause has **only one literal unassigned** (a *unit clause*), assign it in a way that satisfies the clause.
  - Clause: (x) the set x=true

Pure Literal Elimination (optional)

- If a literal appears with **only one polarity** (always positive or always negative) in all clauses, assign it to satisfy all such clauses.
  - If x appears only as x (never as ¬x), set x = true.

Variable Assignment (Decision Step)

- Pick an unassigned variable and try assigning true or false.

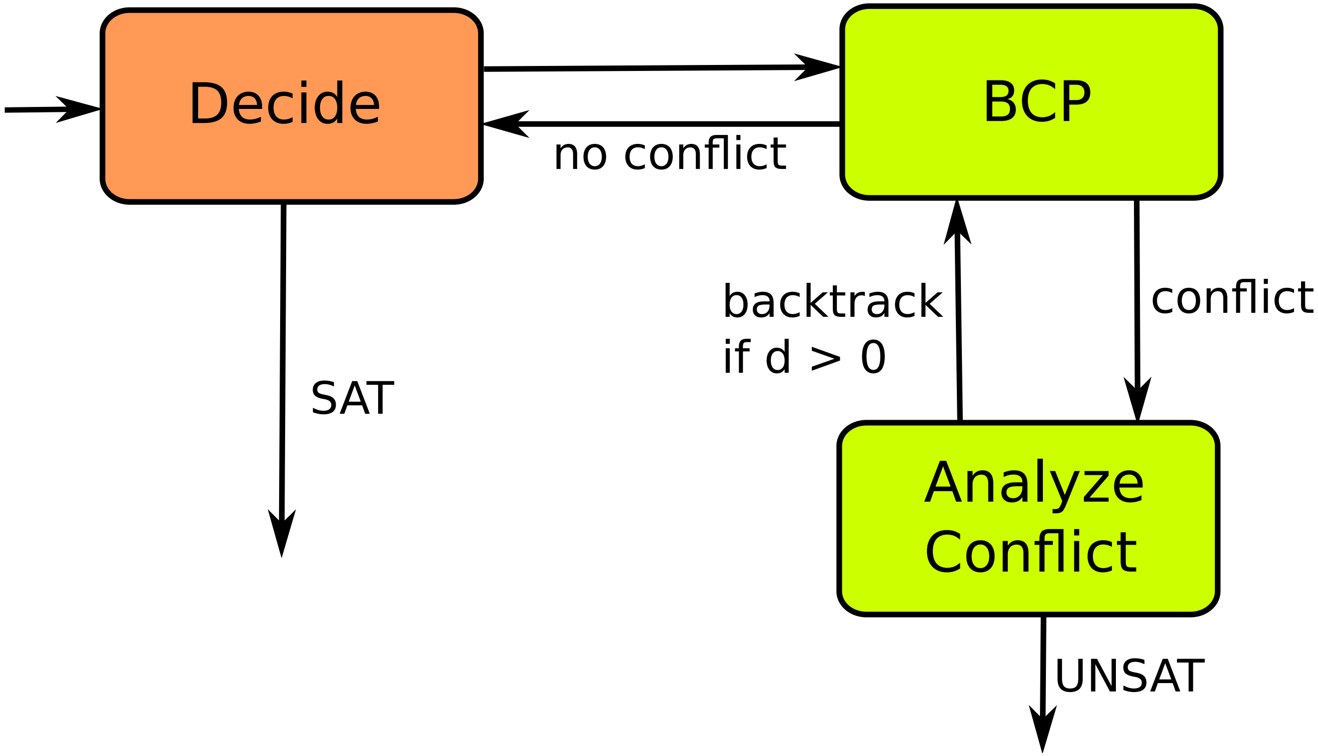
Backtracking

- If the current assignment leads to a **conflict** (unsatisfiable clause), **backtrack** and try the other assignment.
- If both fail, backtrack further.

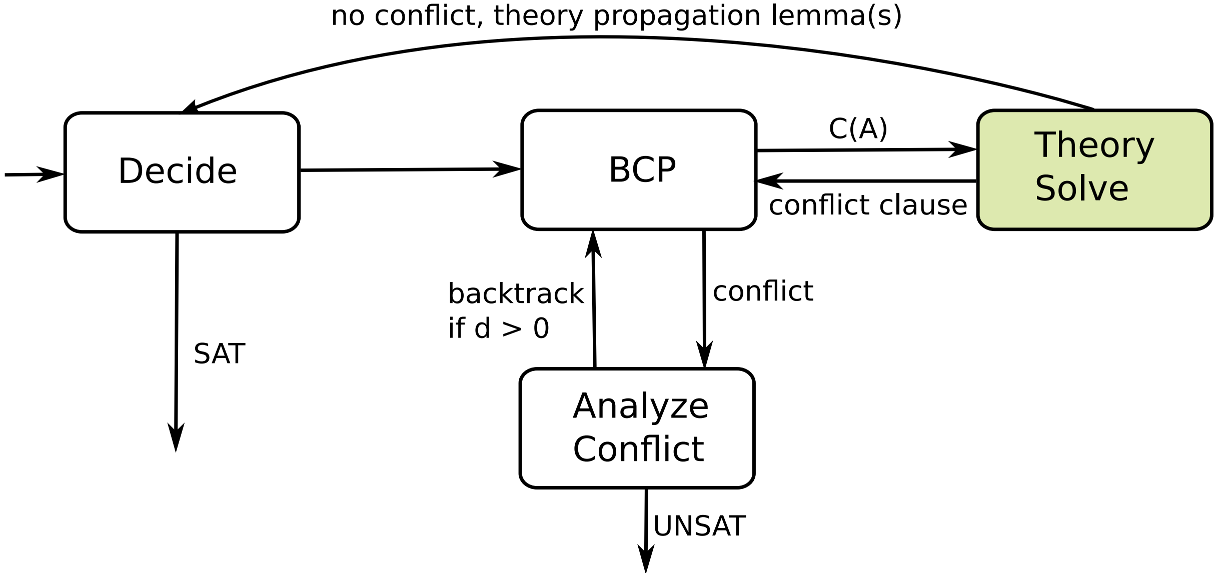
The code

```
let rec dpll (f: formula) : bool =
  let fp = bcp f in
  match fp with
  | Some True -> true
  | Some False -> false
  | None ->
    begin
    let p = choose_var f in
    let ft = (subst_var f p true) in
    let ff = (subst_var f p false) in
    dpll ft || dpll ff
    end
  end
```

Discussion


Decide
BCP
no conflict
SAT
backtrack
if d > 0
conflict
Analyze
Conflict
UNSAT

What about theory?



- Combination of DPLL-based SAT solver and decision procedure for conjunctive $\mathcal{T}$ formula called DPLL($\mathcal{T}$) framework

Theory propagation

Assume that the original formula contains literals

$$x = y, y = z, x < z$$

with corresponding boolean variables $b_1, b_2 and\ b_3$

Theory propagation

Assume that the original formula contains literals

$$x = y, y = z, x < z$$

with corresponding boolean variables $b_1, b_2 and\ b_3$

Assume SAT make the (partial) assignment

$$b_1 = true, b_2 = true$$

Next computationalk step may decide to assign

$$b_3 = true\ or\ b_3 = false$$

Theory propagation

Assume that the original formula contains literals

$$x = y, y = z, x < z$$

with corresponding boolean variables $b_1, b_2 and\ b_3$

Assume SAT make the (partial) assignment

$$b_1 = true, b_2 = true$$

Next computational step may decide to assign

$$b_3 = true\ or\ b_3 = false$$

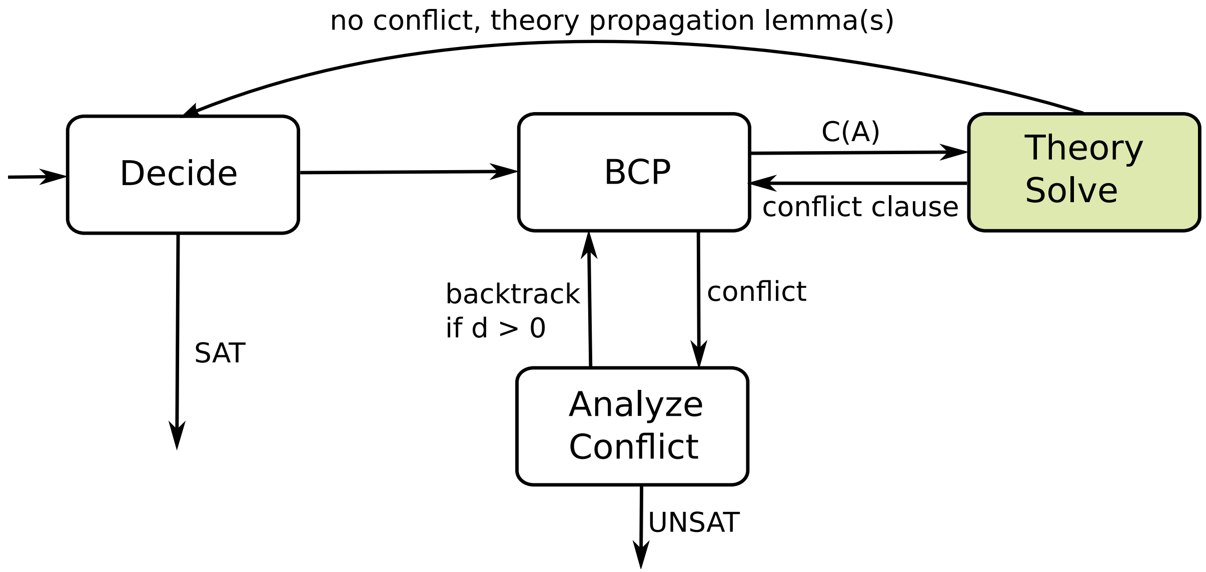
?? $b_3 = true$??

not the right choice: conflict in the equational theory

Theory propagation

- Theory solver tracks which literals are implied by the current assignment.
- Our example: literal $\neg(x < y)$ is implied by the partial assignment $\boldsymbol{b_1 = true, b_2 = true}$
- The implication $\boldsymbol{b_1 \wedge b_2 \rightarrow b_3}$ can be safely added to the knowledge of clauses (the clause database)

DPLL(T) framework



- Adding theory propagation lemmas prevents bad assignments to boolean abstraction

Z3 SAT SOLVER

- **Z3** is an **SMT** solver developed by Microsoft Research
  - https://github.com/Z3Prover/z3
- **Z3 Input**: A set of **declarative constraints**, often expressed in logic over various domains:
  - Integers, reals, booleans, bitvectors, arrays, strings, etc.
- Converts the problem into a combination of **Boolean SAT solving** + **theory solvers** (for strings, integers, arrays, etc.)
- Uses **efficient heuristics and decision procedures** to prune infeasible choices

Z3 in Action

(x > 0 && x < 10) (x * 2 == 15) {x:int}

```
Z3>

(declare-const x Int)
      (assert (> x 0))
      (assert (< x 10))
      (assert (= (* x 2) 15))
(check-sat)
```

```
Z3 >

unsat
```

```
from z3 import *

# declare multiple variables
x, y = Bools('x y’)

# create a solver instance
s = Solver()

# add conjuncts
s.add( Implies(x, y) )
s.add( Implies(y, x) )

# check satisfiability
print( s.check() )
print( s.model() )

s.add( x )
s.add( Not(y) )

# check satisfiability
print( s.check() )
```

The first two conjuncts are satisfiable, we get a model

```
sat
[y = False, x = False]
unsat
```

All four conjuncts together are unsatifiable

Z3 Theories

Linear integer/real arithmetic

$$19 * x + 2 * y = 42$$

- (Unbounded) arithmetic is often used to approximate int and float
- Multiplication by constants is supported

Non-linear integer/real arithmetic

$$x * y + 2 * x * y + 1 = (x + y) * (x + y)$$

- Useful for programs that perform multiplication and division, e.g., crypto libraries

Equality logic with uninterpreted functions

$$(x = y \land u = v) \Rightarrow f(x, u) = f(y, v)$$

- Universal mechanism to encode operations not natively supported by a theory

Fixed-size bitvector arithmetic

$$x \,\&\, y \quad \leq \quad x \mid y$$

- To encode bit-level operations
- To perform bit-precise reasoning, e.g., floats

Array theory

$$\textit{read}(\textit{write}(a, i, v), i) = v$$

- To encode data types such as arrays

```
from z3 import *

# -----------------------------------------------------------
# 1. Declare a new abstract sort (type) for pairs
# -----------------------------------------------------------
Pair = DeclareSort('Pair')

# -----------------------------------------------------------
# 2. Define constants and functions over the Pair sort
# -----------------------------------------------------------
null  = Const('null', Pair)                 # A constant representinghe empty pair
cons  = Function('cons', IntSort(), IntSort(), Pair)  # Constructor: builds a Pair from two Ints
first = Function('first', Pair, IntSort())     # Selector: extracts the first Int from a Pair

# -----------------------------------------------------------
# 3. Define axioms describing the behavior of our abstract model
# -----------------------------------------------------------
# Axiom 1: the constant 'null' is equivalent to cons(0, 0)
ax1 = (null == cons(0, 0))

# Axiom 2: for all integers x, y — the first element of cons(x, y) is x
x, y = Ints('x y')
ax2 = ForAll([x, y], first(cons(x, y)) == x)
```

```
# ----------------------------------------------------------
# 4. Create a solver and add the axioms
# ----------------------------------------------------------
s = Solver()
s.add(ax1)
s.add(ax2)

# ----------------------------------------------------------
# 5. Define the formula we want to prove: first(null) == 0
# ----------------------------------------------------------
F = (first(null) == 0)

# ----------------------------------------------------------
# 6. Check validity of F
#    To prove F is valid, we check whether ¬F is unsatisfiable
# ----------------------------------------------------------
s.add(Not(F))
print("Checking validity of F = first(null) == 0 ...")
print("Result:", s.check())
```

When you run this, Z3 prints:

```
Checking validity of F = first(null) == 0 ...
Result: unsat
```

`unsat` means that the **negation** of the formula (`first(null) != 0`) is unsatisfiable — so the original formula `first(null) == 0` is **logically valid**, given the axioms.

Using an SMT solver to verify a program

$\{ a = 1 \wedge 0 \leq b*b - 4*c \}$

// Check that this entailment is valid (its negation is unsatisfiable)

$\{ b*b - 4*a*c < 0 \wedge \text{false} \vee$

$\neg(b*b - 4*a*c < 0) \wedge a*((-b + \sqrt{b*b - 4*a*c}) / 2)^2 + b*((-b + \sqrt{b*b - 4*a*c}) / 2) + c = 0 \}$

```
from z3 import *

a, b, c = Reals('a b c')
d = b*b - 4*a*c

PO = Implies(
       And(a == 1, 0 <= b*b - 4*c),
       Or( And(d < 0, False),
           And(Not(d < 0),
               a*((-b + Sqrt(d))/2)*((-b + Sqrt(d))/2) + b*((-b + Sqrt(d))/2) + c == 0
      )))

# check validity
s = Solver()
s.add(Not(PO)); print( s.check() )
```

- Z3 selects theories based on the features appearing in formulas
  - Most verification problems require a combination of many theories

> Quantifier-free linear integer arithmetic with uninterpreted functions
>
> $$17 * x + 23 * f(y) > x + y + 42$$

- Some theories are decidable, e.g., quantifier-free linear arithmetic
  - SMT solver will terminate and report either “sat” or “unsat”

- Some theories are undecidable, e.g., nonlinear integer arithmetic
  - Especially in combination with quantifiers
  - SMT solver uses heuristics and may not terminate or return “unknown”
  - Results can be flaky, e.g., depend on order of declarations or random seeds

Our first example

```
function f(a, b, c) {
  var x = y = z = 0;
  if (a) {
    x = -2;
  }
  if (b > 5) {
    if (!a && c) {
      y = 1;
    }
    z = 2;
  }
  assert(x + y + z != 3);
}
```

Our first example

```
function f(a, b, c) {
  var x = y = z = 0;
  if (a) {
    x = -2;
  }
  if (b > 5) {
    if (!a && c) {
      y = 1;
    }
    z = 2;
  }
  assert(x + y + z != 3);
}
```

```
from z3 import *

# Symbolic inputs
a0, b0, c0 = Ints('a0 b0 c0')

# Helpful predicates for "truthiness"
A = (a0 != 0)
B = (b0 > 5)
C = (c0 != 0)

def model_or_none(constraints):
   s = Solver()
   s.add(constraints)
   return s.model() if s.check() == sat else None

# Enumerate the relevant paths with their path conditions
(PC) and compute x,y,z symbolically
paths = []
```

```
function f(a, b, c) {
  var x = y = z = 0;
  if (a) {
    x = -2;
  }
  if (b > 5) {
    if (!a && c) {
      y = 1;
    }
    z = 2;
  }
  assert(x + y + z != 3);
}
```

```
# if (a)
#   then: x=-2
#   else: x=0
for condA, x_val in [(A, -2), (Not(A), 0)]:
    # if (b > 5)
    #   then:
    #     if (!a && c) y=1 else y=0
    #     z=2
    #   else: y=0; z=0
    # THEN branch of b>5
    pc_thenB = And(condA, B)
    y_thenB = If(And(Not(A), C), 1, 0)   # inner if only affects y
    z_thenB = 2
    s_thenB = x_val + y_thenB + z_thenB
    paths.append(("A?, B then", pc_thenB, x_val, y_thenB, z_thenB, s_thenB))

    # ELSE branch of b>5
    pc_elseB = And(condA, Not(B))
    y_elseB = 0
    z_elseB = 0
    s_elseB = x_val + y_elseB + z_elseB
    paths.append(("A?, B else", pc_elseB, x_val, y_elseB, z_elseB, s_elseB))
```

```javascript
function f(a, b, c) {
  var x = y = z = 0;
  if (a) {
    x = -2;
  }
  if (b > 5) {
    if (!a && c) {
      y = 1;
    }
    z = 2;
  }
  assert(x + y + z != 3);
}
```

```python
# Check each path: (1) feasibility; (2) assertion violation x+y+z != 3
for name, pc, x_sym, y_sym, z_sym, sum_sym in paths:
    # 1) Path feasibility
    m = model_or_none(pc)
    if m is None:
        continue  # infeasible path

    # 2) Try to violate the assertion: sum == 3 (since assert(sum != 3))
    m_bad = model_or_none(And(pc, sum_sym == 3))
    if m_bad:
        print(f"[ASSERTION FAIL] Path: {name}")
        print("  Example input:", {d.name(): m_bad[d] for d in [a0,b0,c0]})
        print("  Computed (x,y,z,sum) =",
              (x_sym if isinstance(x_sym, int) else m_bad.eval(x_sym),
               y_sym if isinstance(y_sym, int) else m_bad.eval(y_sym),
               z_sym if isinstance(z_sym, int) else m_bad.eval(z_sym),
               m_bad.eval(sum_sym)))
    else:
        # Also produce a concrete input that simply exercises the path (even if no bug)
        m_path = model_or_none(pc)
        print(f"[OK PATH] Path: {name}")
        print("  Example input:", {d.name(): m_path[d] for d in [a0,b0,c0]})
        print("  sum =", m_path.eval(sum_sym))
```