Lecture 7
Introduction to SAT and SMT

Nadia Polikarpova

Why do we care?

1. Synthesis is combinatorial search, and so is SAT/SMT
2. SAT/SMT solvers are really good these days
3. ??? ← this week
4. Profit!!!

Boolean SATisfiability

gin ∨ tonic

Solution:

minor ↦ T
gin ↦ F
tonic ↦ T

Satisfiability Modulo Theories

$$(\mathit{gin} \vee \mathit{tonic}) \wedge (\mathit{age} < 21 \Rightarrow \mathit{abv} = 0) \wedge (\mathit{age} = 20)$$

In the United States, "gin" is defined as an alcoholic beverage of no less than 40% ABV…

Wikipedia

Satisfiability Modulo Theories

$$(\mathit{gin} \lor \mathit{tonic}) \land (\mathit{age} < 21 \Rightarrow \mathit{abv} = 0) \land (\mathit{age} = 20) \land (\mathit{gin} \Rightarrow \mathit{abv} \geq 40)$$

theory of Linear Integer Arithmetic

$\mathit{age} \mapsto 20$

$\mathit{abv} \mapsto 0$

$\mathit{gin} \mapsto F$

$\mathit{tonic} \mapsto T$

Popular Solvers

Microsoft	Stanford	SRI	JKU Linz, Austria
Z3	CVC4	Yices2	Boolector

SMT competition: http://smtcomp.sourceforge.net

```
.smt2                // SMTLib format

(declare-fun age () Int)
(declare-fun abv () Int)
```

SMT-LIB

Uniform format for SMT problems understood by all solvers

```
(declare-fun age () Int)
(declare-fun abv () Int)
(declare-fun gin () Bool)
(declare-fun tonic () Bool)
(assert (or gin tonic))
(assert (implies (< age 21) (= abv 0)))
(assert (= age 20))
(assert (implies gin (>= abv 40)))
(check-sat)
(get-model)
```

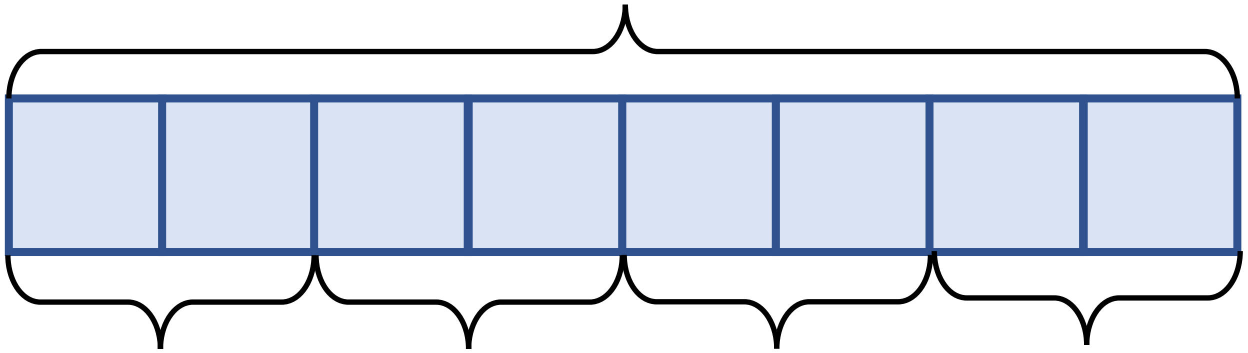
This lecture

1. Demo: how to use Z3 to
   - solve constraints
   - verify programs
   - synthesize programs
2. How do SAT solvers work?
3. How do SMT solvers work?

Problem: Array Partitioning

Partition an array of size *N* evenly into *P* sub-ranges

$N = 8$

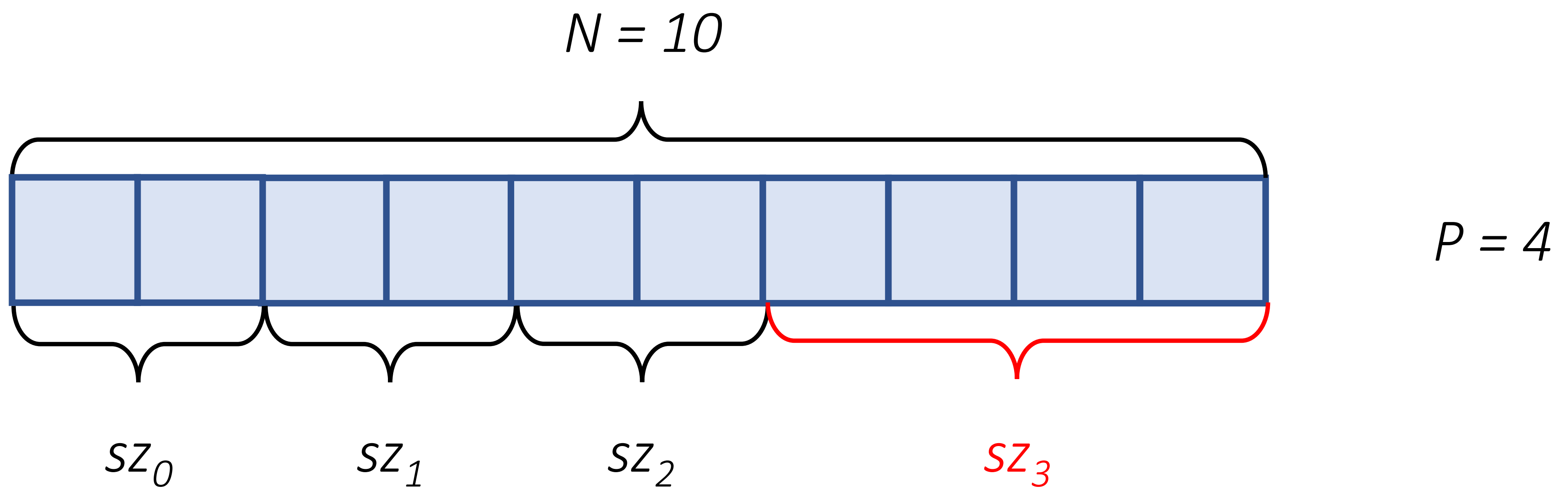


sz_0 sz_1 sz_2 sz_3

$P = 4$

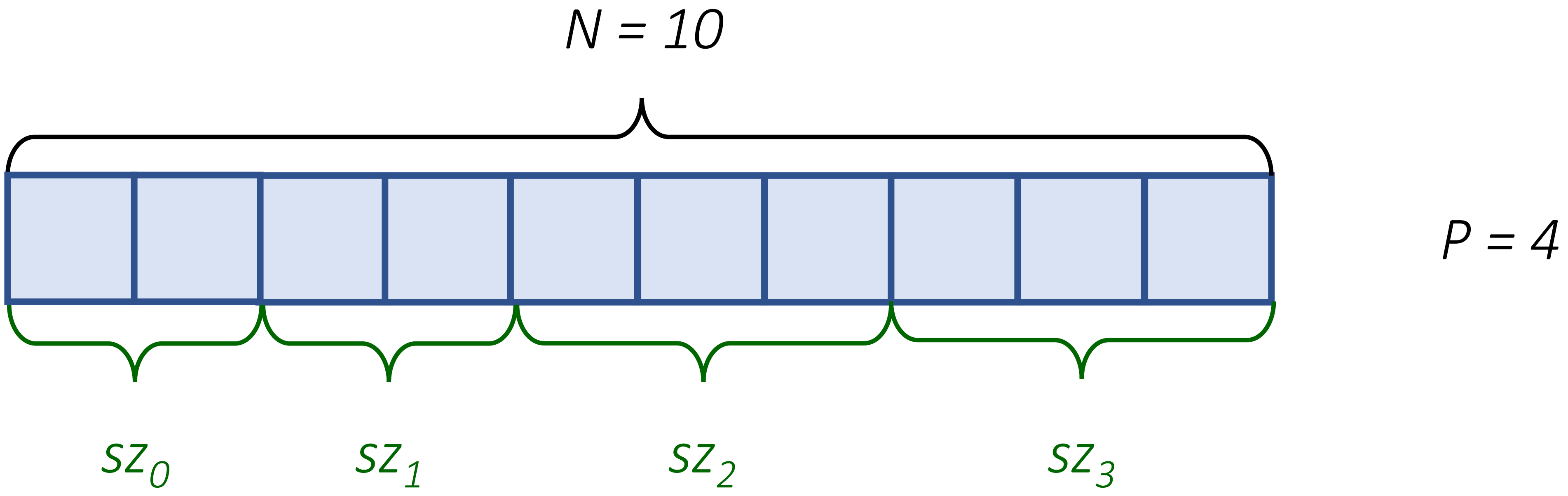
Problem: Array Partitioning

Partition an array of size N evenly into P sub-ranges



Problem: Array Partitioning

Partition an array of size N evenly into P sub-ranges



Can we always make them differ by at most 1?

Z3

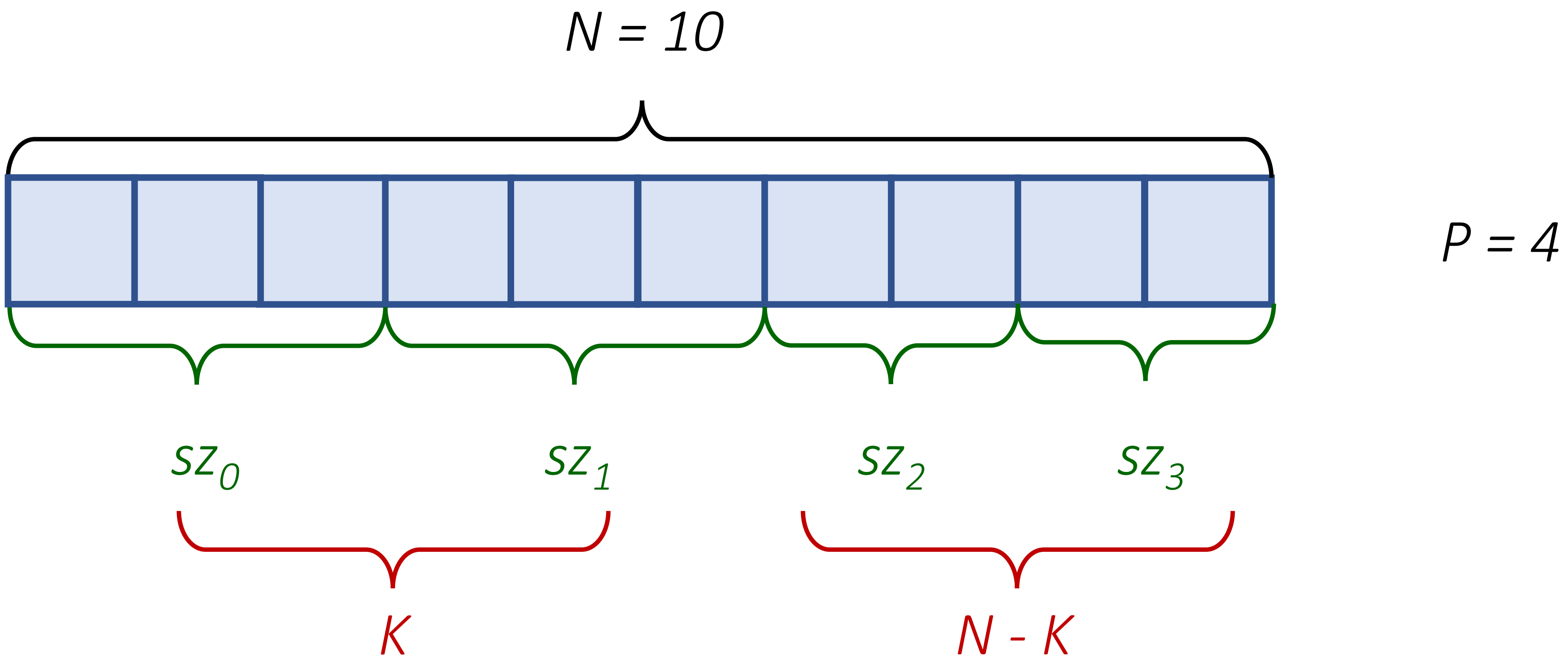
to the rescue!

code: https://github.com/nadia-polikarpova/smt-talk

This lecture

1. Demo: how to use Z3 to
   - solve constraints
   - verify programs
   - synthesize programs
2. How do SAT solvers work?
3. How do SMT solvers work?

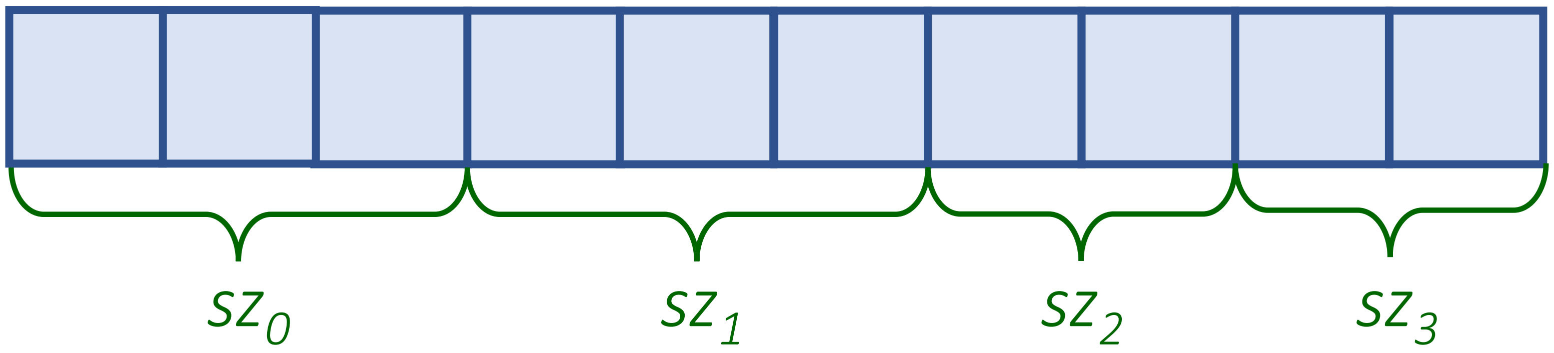
Let's generalize this into a program!



A program for partitioning

```
for i in range(P):
  sz[i] = if i < K: n/P + 1 else n/P
```

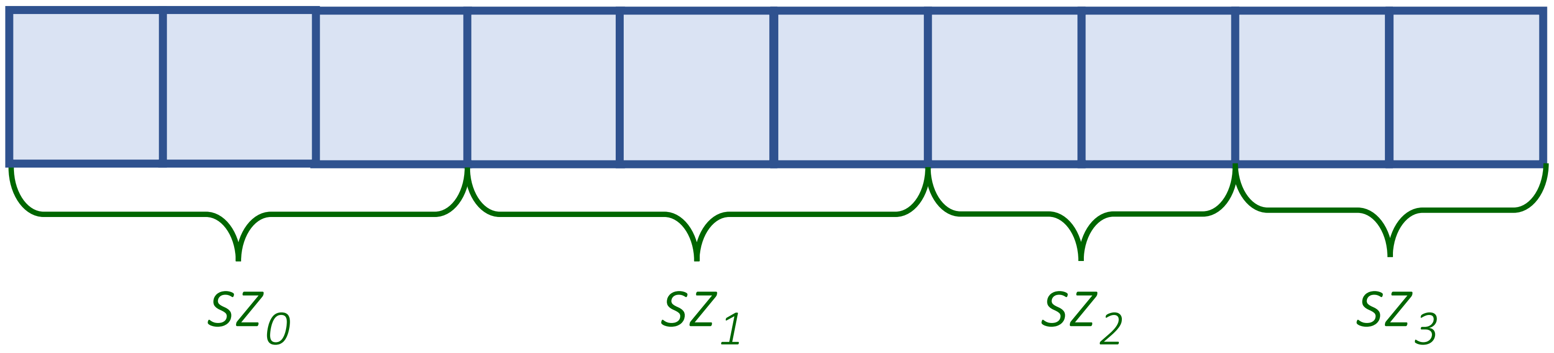
I want this program to work for all *n*!
What should my *K* be?



A program for partitioning

```
for i in range(P):
  sz[i] = if i < 2: n/P + 1 else n/P
```

How do I check whether this program works for all *n*?



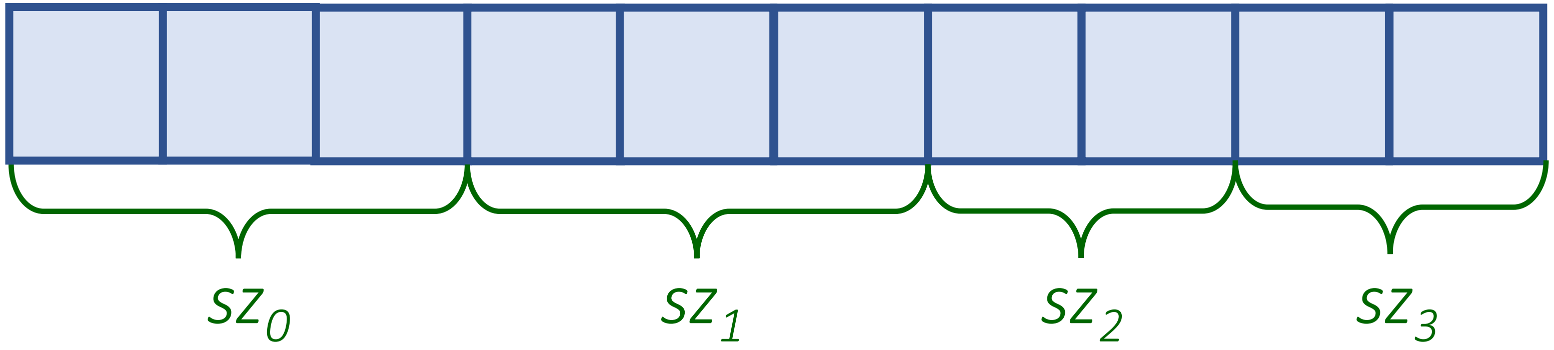
This lecture

1. Demo: how to use Z3 to
   - solve constraints
   - verify programs
   - synthesize programs
2. How do SAT solvers work?
3. How do SMT solvers work?

A program for partitioning

```
for i in range(P):
  sz[i] = if i < K: n/P + 1 else n/P
```

How do I ask *the solver* to pick the expression *K*?



This lecture

1. Demo: how to use Z3 to
    - solve constraints
    - verify programs
    - synthesize programs
2. How do SAT solvers work?
3. How do SMT solvers work?

The SAT problem

Input: propositional formula in CNF

literal
= atom or its negation

formula
= conjunction (set) of clauses

$$\boxed{(\neg \boxed{a} \vee \boxed{\neg b}) \wedge (\boxed{b} \vee c) \wedge \boxed{(\neg a \vee \neg c \vee d)} \wedge (b \vee \neg c \vee \neg d) \wedge (a \vee d)}$$

atom
= propositional variable

clause
= disjunction of literals

The SAT problem

Problem: find a *satisfying assignment* (also called a *model*)

- or determine that the formula is *unsatisfiable*

$$(\neg a \vee \neg b) \wedge (b \vee c) \wedge (\neg a \vee \neg c \vee d) \wedge (b \vee \neg c \vee \neg d) \wedge (a \vee d)$$

a satisfying assignment:

$\{a \mapsto 0, b \mapsto 1, c \mapsto 0, d \mapsto 1\}$

can be written as a set of literals:

$\{\neg a, b, \neg c, d\}$

or as a formula:

$\neg a \wedge b \wedge \neg c \wedge d$

Naive solution

$$(\neg a \lor \neg b) \land (b \lor c) \land (\neg a \lor \neg c \lor d) \land (b \lor \neg c \lor \neg d) \land (a \lor d)$$

Build a truth table!

- We can't do fundamentally better: it's an NP-complete problem
- But we can do way better in practice for common instances

$2^{|P|}$

0000	0
0001	0
0010	0
0011	0
0100	0
0101	1
0110	0
0111	1
…	

Intuition: Sudoku

Easy vs hard: what's the difference?

7	9					3		
					6	9		
8				3			7	6
			9	6	5			2
		5	4	1	8	7		
4			7	2	3			
6	1			9				8
		2	3					
		9					5	4

		9	7	4	8			
7								
	2		1		9			
		7				2	4	
	6	4		1		5	9	
	9	8				3		
			8		3		2	
								6
			2	7	5	9		

Most real-world SAT instances allow a lot of inference

DPLL algorithm

[Davis, Putnam '60]
[Davis, Logemann, Loveland '62]

State: current model *M* (a sequence of annotated literals)

decision literal

$$M = \boxed{a^d} \neg b \; c$$

Transitions:

- decide $\quad M \longrightarrow M\, l^d \quad$ if *l* undefined in *M*
- unit-propagate $\quad M \longrightarrow M\, l \quad$ if there is a clause where all literals are false except *l*, which is undefined
- backtrack $\quad M l^d M' \longrightarrow M\, \neg l \quad$ if there is a conflicting clause and *M'* has no decision literals
- fail $\quad M \longrightarrow Unsat \quad$ if there is a conflicting clause and no decision literals

DPLL: example

$$\boxed{(\neg a \vee \neg b)} \wedge \boxed{(b \vee c)} \wedge \boxed{(\neg a \vee \neg c \vee d)} \wedge \boxed{(b \vee \neg c \vee \neg d)} \wedge \boxed{(a \vee d)}$$

$M =$	$\emptyset$	decide
	a^d	unit-propagate
	$a^d\ \neg b$	unit-propagate
	$a^d\ \neg b\ c$	unit-propagate
	$a^d\ \neg b\ c\ d$	backtrack
	$\neg a$	unit-propagate
	$\neg a\ d$	decide
	$\neg a\ d \neg c^d$	unit-propagate
	$\neg a\ d \neg c^d\ b$	SAT!

DPLL + clause learning

$$\boxed{(\neg a \vee b)} \wedge \boxed{(\neg c \vee d)} \wedge \boxed{(\neg e \vee \neg f)} \wedge \boxed{(f \vee \neg b \vee \neg e)} \wedge (\neg a \vee \neg e)$$

$M =$ $\emptyset$	decide	
a^d	unit-propagate	
$a^d\ b$	decide	
$a^d\ b\ c^d$	unit-propagate	
$a^d\ b\ c^d\ d$	decide	
$a^d\ b\ c^d\ d\ e^d$	unit-propagate	
$a^d\ b\ c^d\ d\ e^d\ \neg f$	backtrack	Wait, but why?
$a^d\ b\ c^d\ d\ \neg e$		

Bad decision!

DPLL + clause learning

$$(\neg a \vee b) \wedge (\neg c \vee d) \wedge (\neg e \vee \neg f) \wedge (f \vee \neg b \vee \neg e) \wedge (\neg a \vee \neg e)$$

$M =$	$\emptyset$	decide
	a^d	unit-propagate
	$a^d\ b$	decide
	$a^d\ b\ c^d$	unit-propagate
	$a^d\ b\ c^d\ d$	decide
	$a^d\ b\ c^d\ d\ e^d$	unit-propagate
	$a^d\ b\ c^d\ d\ e^d\ \neg f$	**backjump**
	$a^d\ b\ \neg e$	

This lecture

1. Demo: how to use Z3 to
    - solve constraints
    - verify programs
    - synthesize programs
2. How do SAT solvers work?
3. How do SMT solvers work?

Beyond propositional logic

What if our formula looks like this?

$$(p \land \neg q \lor a = f(b - c)) \land (g(g(b) \neq c \lor a - c \leq 7)$$

- talks about integers, functions, sets, lists...

One idea: bit-blast everything and use SAT

- can only find solutions within bounds
- very inefficient, so bounds are small

Better idea: combine SAT with special **solvers** for **theories**

- they “natively understand” integers, functions, etc

First-order theories

theory = <function symbols, predicate symbols, axioms>

ground first-order formulas over functions and predicates

Example: theory of Equality and Uninterpreted Functions

EUF = <{f, g, h, ...}, {=}, {

$$\forall x. x = x$$

$$\forall x\, y. x = y \;\Rightarrow\; y = x$$

$$\forall x\, y\, z. x = y \wedge y = z \;\Rightarrow\; x = z$$

$$\forall x\, y. x = y \;\Rightarrow\; f(x) = f(y)$$

}>

The SMT problem

literal
= atom or its negation

formula
= conjunction (set) of clauses

theory +

$$\boxed{\boxed{g(a) = c} \wedge \boxed{\left(\boxed{f(g(a)) \neq f(c)} \vee \boxed{g(a) = d}\right)} \wedge c \neq d}$$

atom
= predicate application

clause
= disjunction of literals

Theories for our purpose

theory = <function symbols, predicate symbols, ~~axioms~~>

solver

can decide consistency of
conjunctions of literals

$f(a) = c$

$f(b) \neq d$

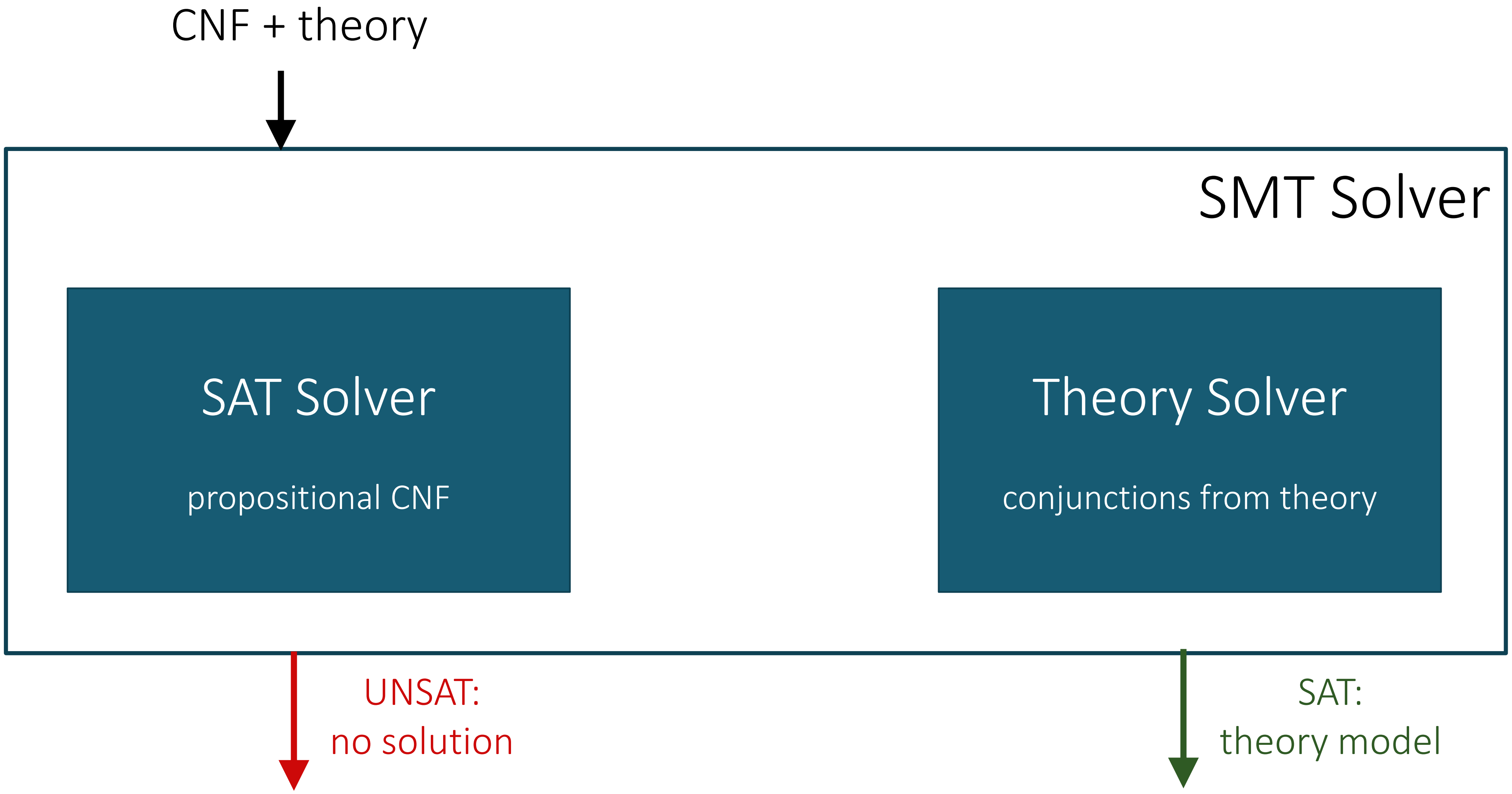
$c = d$

$a = b$

$\xrightarrow{\text{EUF solver}}$

Inconsistent!

DPLL(T) architecture



Basic DPLL(T)

$$\boxed{g(a) = c} \land (\boxed{f(g(a)) \neq f(c)} \lor \boxed{g(a) = d}) \land \boxed{c \neq d}$$

abstract atoms to propositional variables

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

$$p \land (\neg q \lor r) \land \neg s \xrightarrow{\text{SAT solver}} p \ \neg q \ \neg s$$

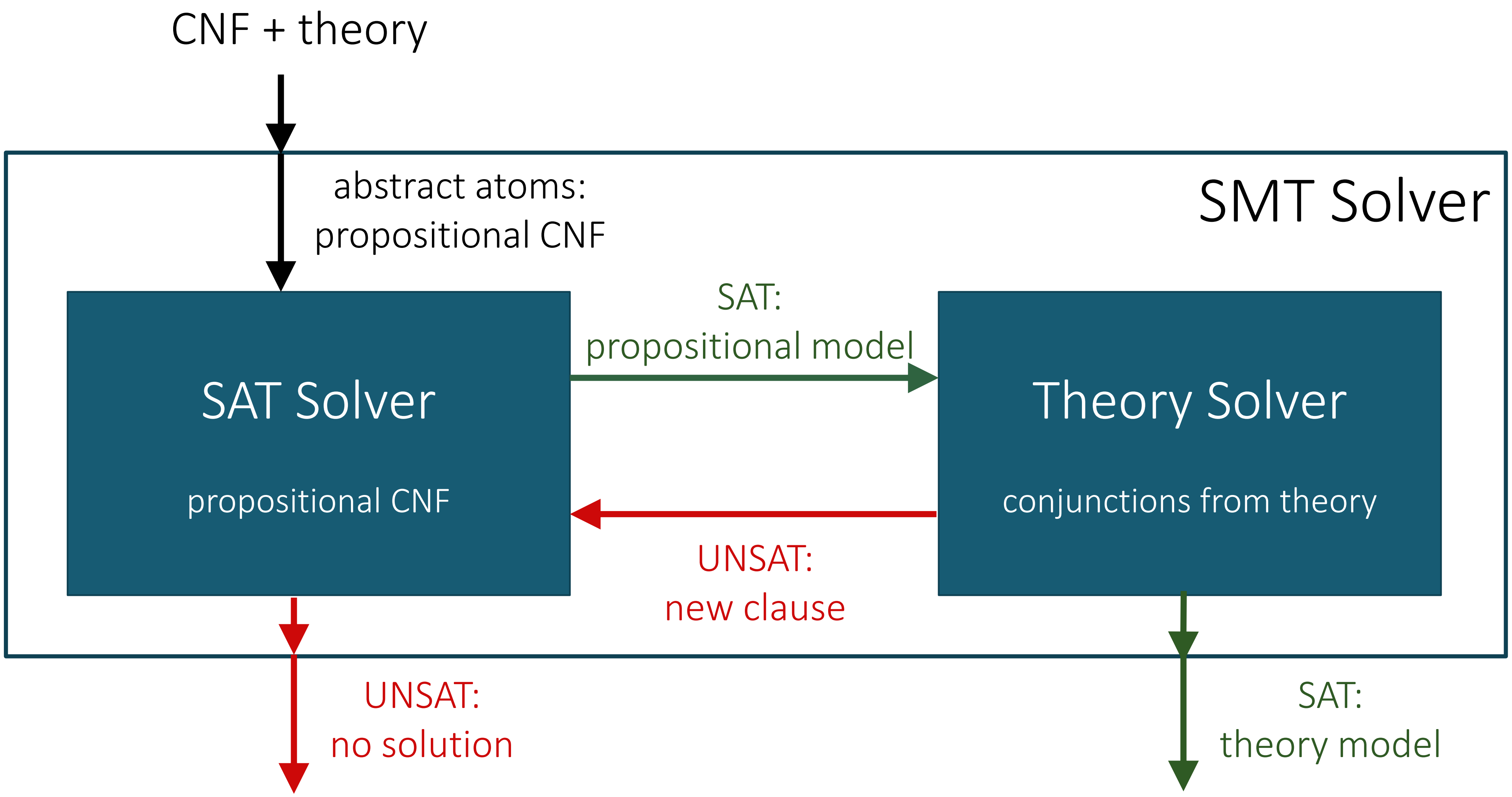
$$\text{Inconsistent!} \xleftarrow{\text{EUF solver}} \boxed{g(a) = c} \quad \boxed{f(g(a)) \neq f(c)} \quad c \neq d$$

$$p \land (\neg q \lor r) \land \neg s \land (\neg p \lor q) \xrightarrow{\text{SAT solver}} p \ q \ r \ \neg s$$

$$\text{Inconsistent!} \xleftarrow{\text{EUF solver}} \boxed{g(a) = c} \quad f(g(a)) = f(c) \quad \boxed{g(a) = d} \quad \boxed{c \neq d}$$

$$p \land (\neg q \lor r) \land \neg s \land (\neg p \lor q) \land (\neg p \land \neg r \land s) \xrightarrow{\text{SAT solver}} \text{Unsat}$$

DPLL(T) architecture



Popular theories

Equality and Uninterpreted Functions

EUF = <{f, g, h, ...}, {=}, axioms of equality & congruence>

Linear Integer Arithmetic

LIA = <{0, 1, ..., +, -}, {=, ≤}, axioms of arithmetic>

Arrays

Arrays = <{sel, store}, {=}, $\forall a\ i\ v.\, \mathrm{sel}(\mathrm{store}(a, i, v), i) = v$

$\forall a\ i\ j\ v.\, i \neq j \Rightarrow \mathrm{sel}(\mathrm{store}(a, i, v), j) = \mathrm{sel}(a, j)$ >

Theories can be combined!

Nelson-Oppen combination

Why do we care?

If we can encode a synthesis problem as SAT/SMT, we can use solvers to do the search for us

Get some inspiration from how solvers search

- Unit propagation similar to top-down propagation (pruning through inference of consequences of a guess)
- Backjumping / clause learning?
  - Feng, Martins, Bastani, Dillig: Program synthesis using conflict-driven learning. PLDI'18
- Coarse-grained reasoning and gradual refinement like in DPLL(T)?
  - Wang, Dillig, Singh: Program synthesis using abstraction refinement. POPL'18