


Lecture 7

Introduction to SAT and SMT

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 \vee tonic

Solution:

minor \mapsto T

gin \mapsto F

tonic \mapsto T

Satisfiability Modulo Theories

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

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

Wikipedia

Satisfiability Modulo Theories

$$(\text{gin} \vee \text{tonic}) \wedge (\text{age} < 21 \Rightarrow \text{abv} = 0) \wedge (\text{age} = 20) \wedge (\text{gin} \Rightarrow \text{abv} \geq 40)$$

theory of Linear Integer Arithmetic



$\text{age} \mapsto 20$

$\text{abv} \mapsto 0$

$\text{gin} \mapsto \text{F}$

$\text{tonic} \mapsto \text{T}$

Popular Solvers

Microsoft

Z3

Stanford

cvc4

```
(and (or (and (= x0 y0) (= y0 x1)) (and (= x0 z0) (= x1 z0))) (and (= x2 y1) (= y1 z1) (and (= x2 z2) (= z2 x3))) (not (= x0 x3)))
```

SRI

Yices2

JKU Linz, Austria

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

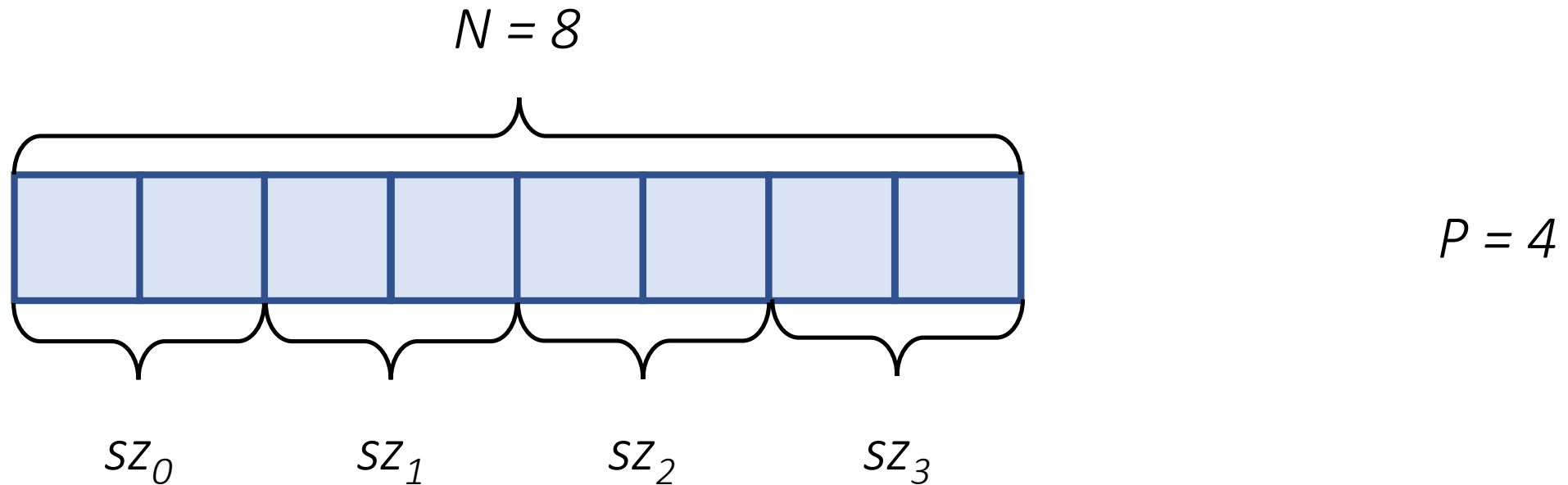
```
(declare-fun age () Int)
(declare-fun abv () Int)
(declare-fun gin () Bool)
(declare-fun tonic () Bool)
(assert (or gin tonic))
(assert (implies (< age 21) (= abv 0)))
(assert (= age 20))
(assert (implies gin (>= abv 40)))
(check-sat)
(get-model)
```

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?

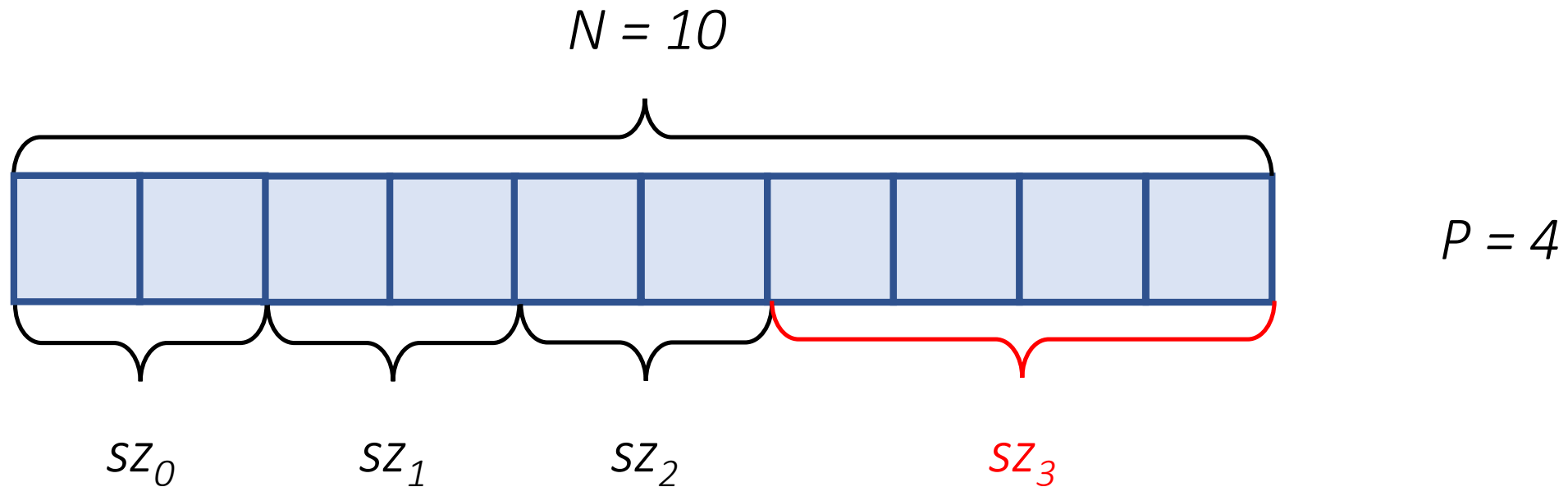
Problem: Array Partitioning

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



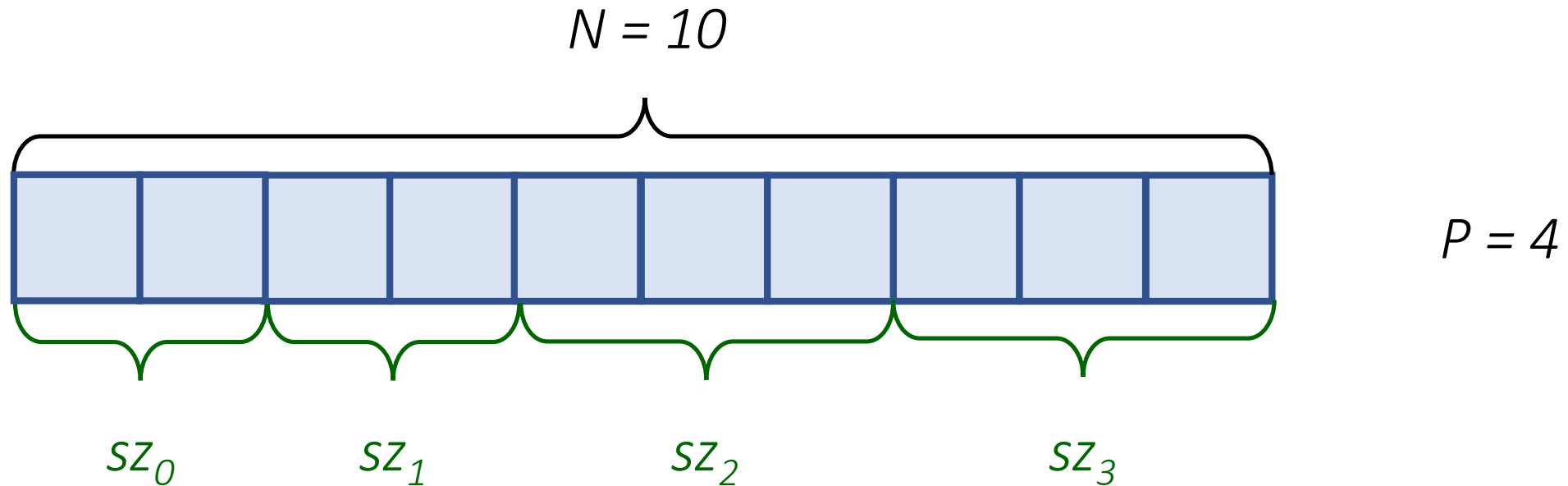
Problem: Array Partitioning

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!

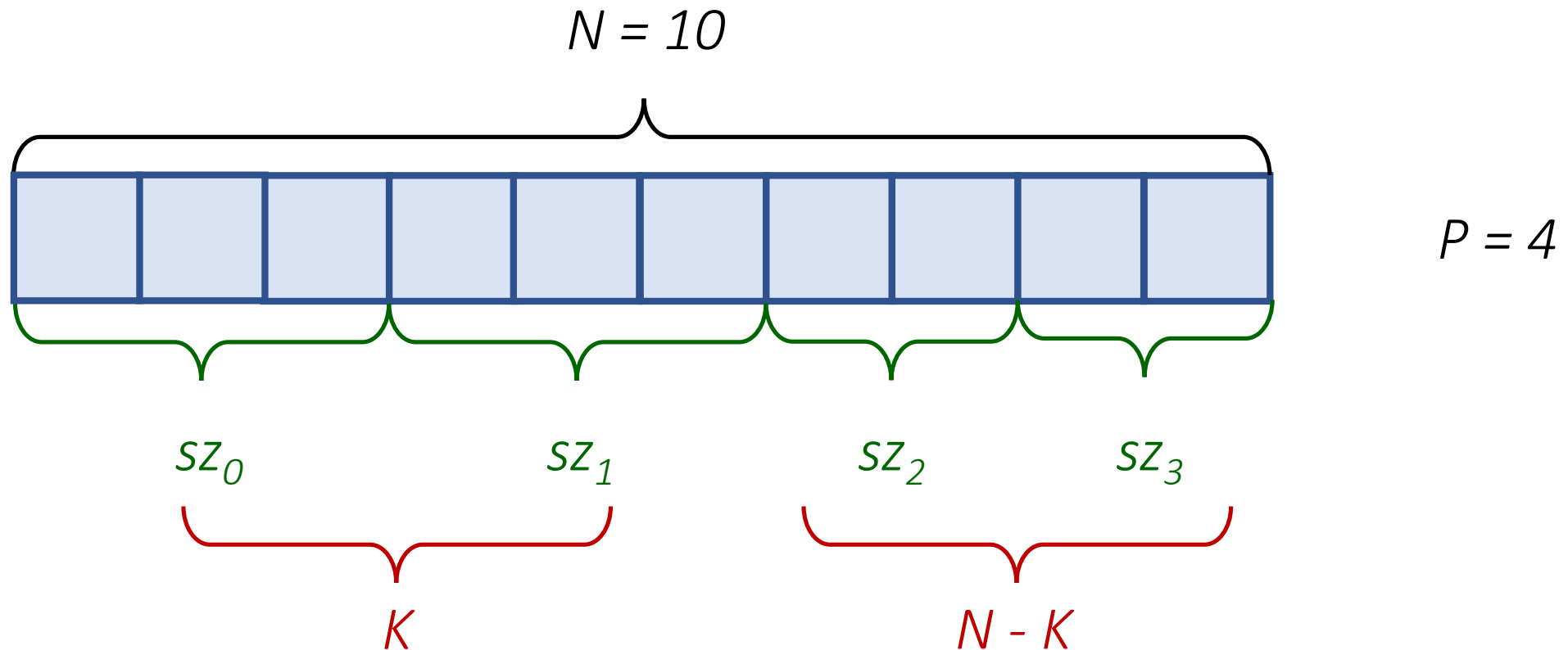
code: <https://github.com/nadia-polikarpova/smt-talk>

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?



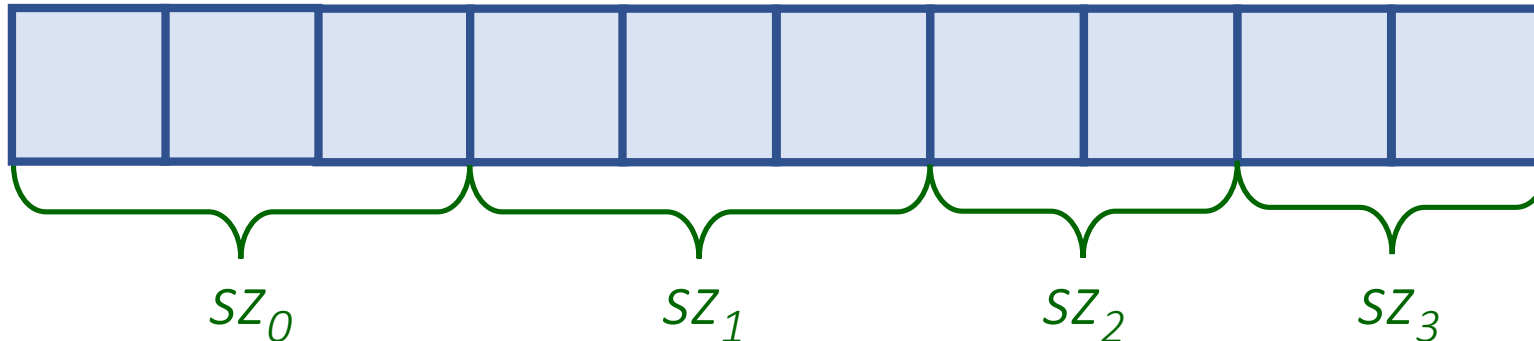
Let's generalize this into a program!



A program for partitioning

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

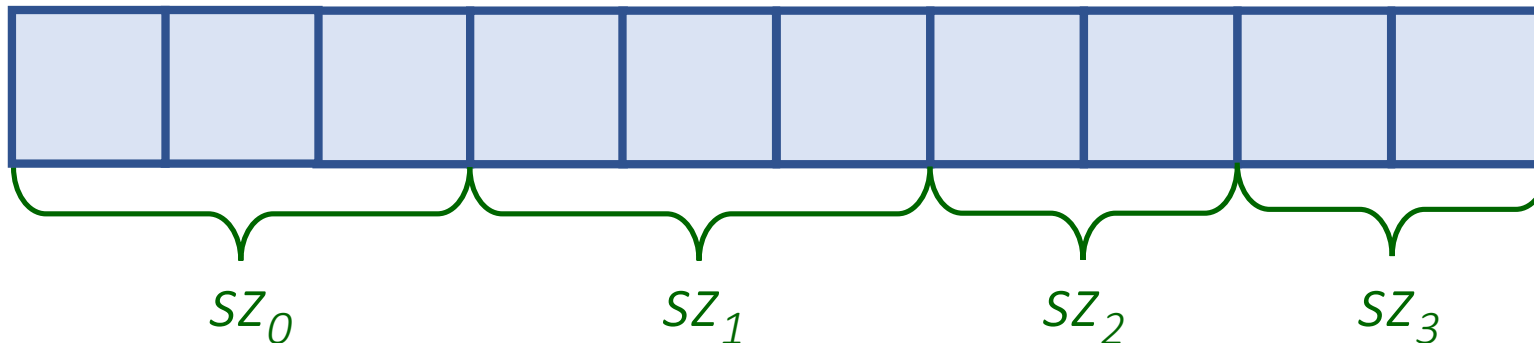
I want this program to work for all n !
What should my K be?



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 ?



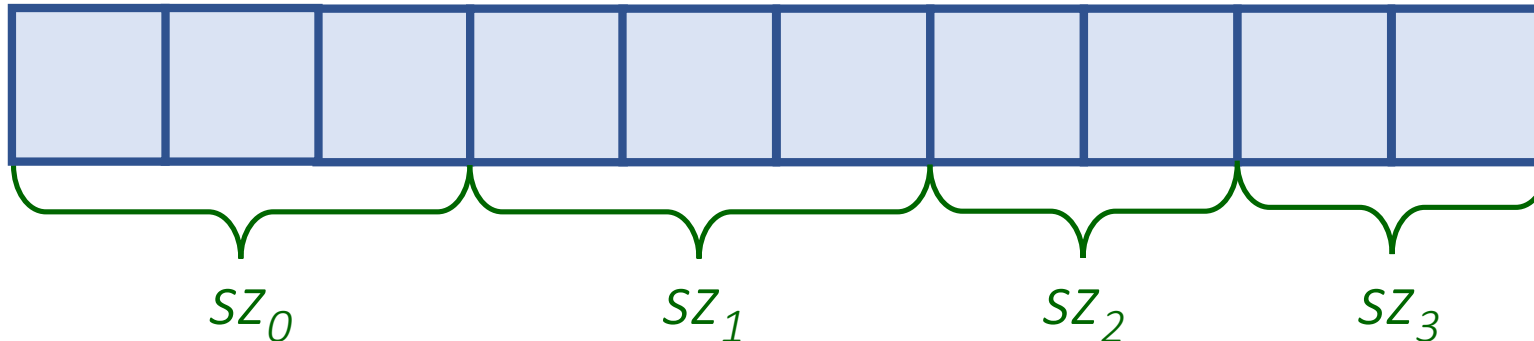
This lecture

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 ?



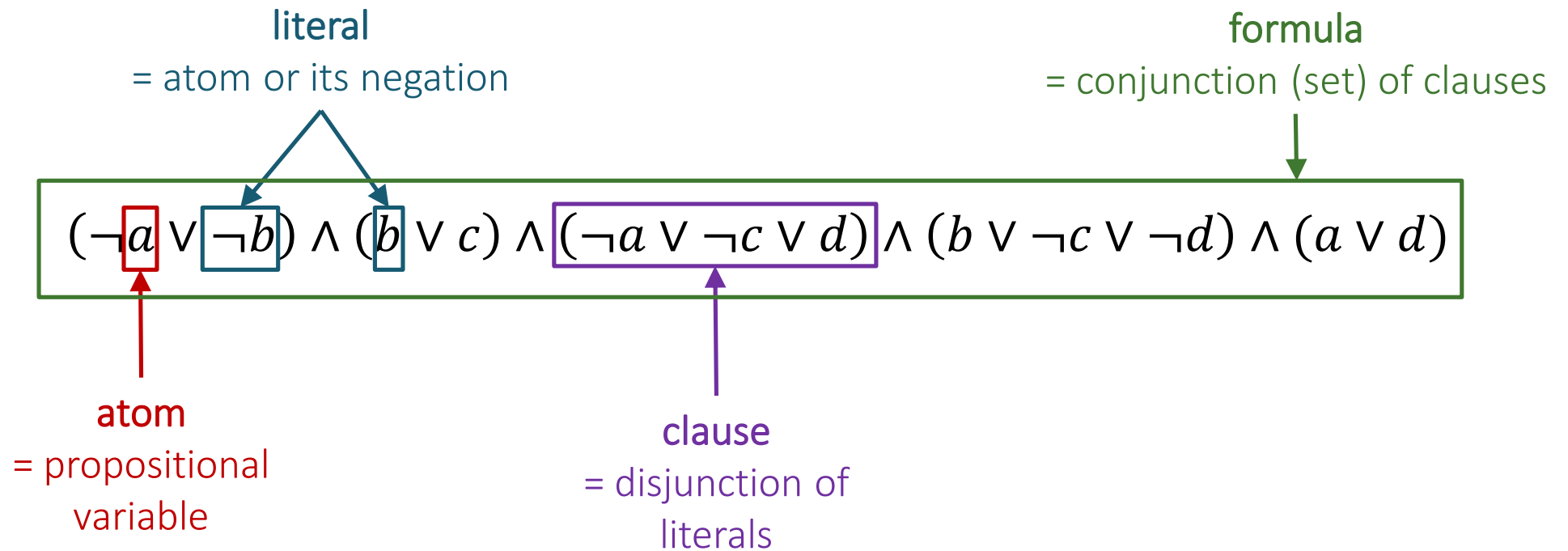
This lecture

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



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

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

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

Copyright 2005 M. Feenstra, Den Haag

		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		

Copyright 2005 M. Feenstra, Den Haag

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)

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

Transitions:

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

DPLL: example

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

$M =$	\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

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

$M =$

- \emptyset
- a^d
- $a^d b$
- $a^d b c^d$
- $a^d b c^d d$
- $a^d b c^d d e^d$
- $a^d b c^d d e^d \neg f$
- $a^d b c^d d \neg e$

Bad decision!

decide
unit-propagate
decide
unit-propagate
decide
unit-propagate
backtrack

Wait, but why?

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?

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

- 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, ...}, {=}, {

$$\forall x. x = x$$

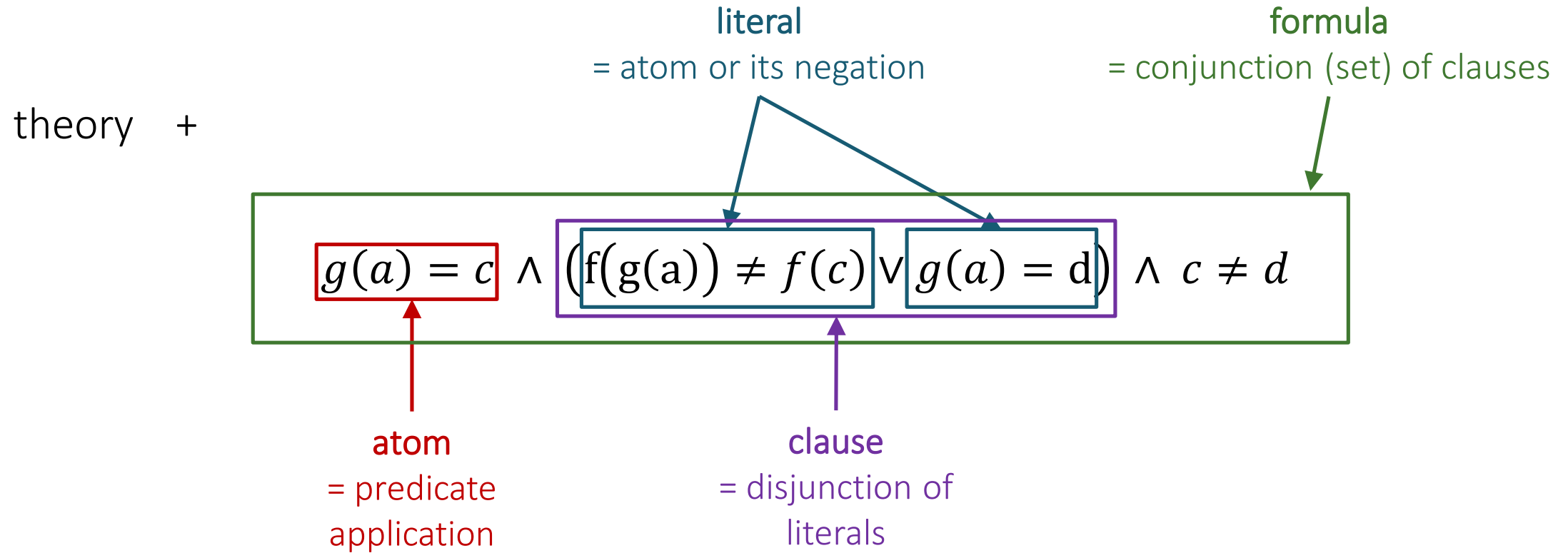
$$\forall x y. x = y \Rightarrow y = x$$

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

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

}>

The SMT problem



Theories for our purpose

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

solver

can decide consistency of
conjunctions of literals

$$f(a) = c$$

$$f(b) \neq d$$

$$c = d$$

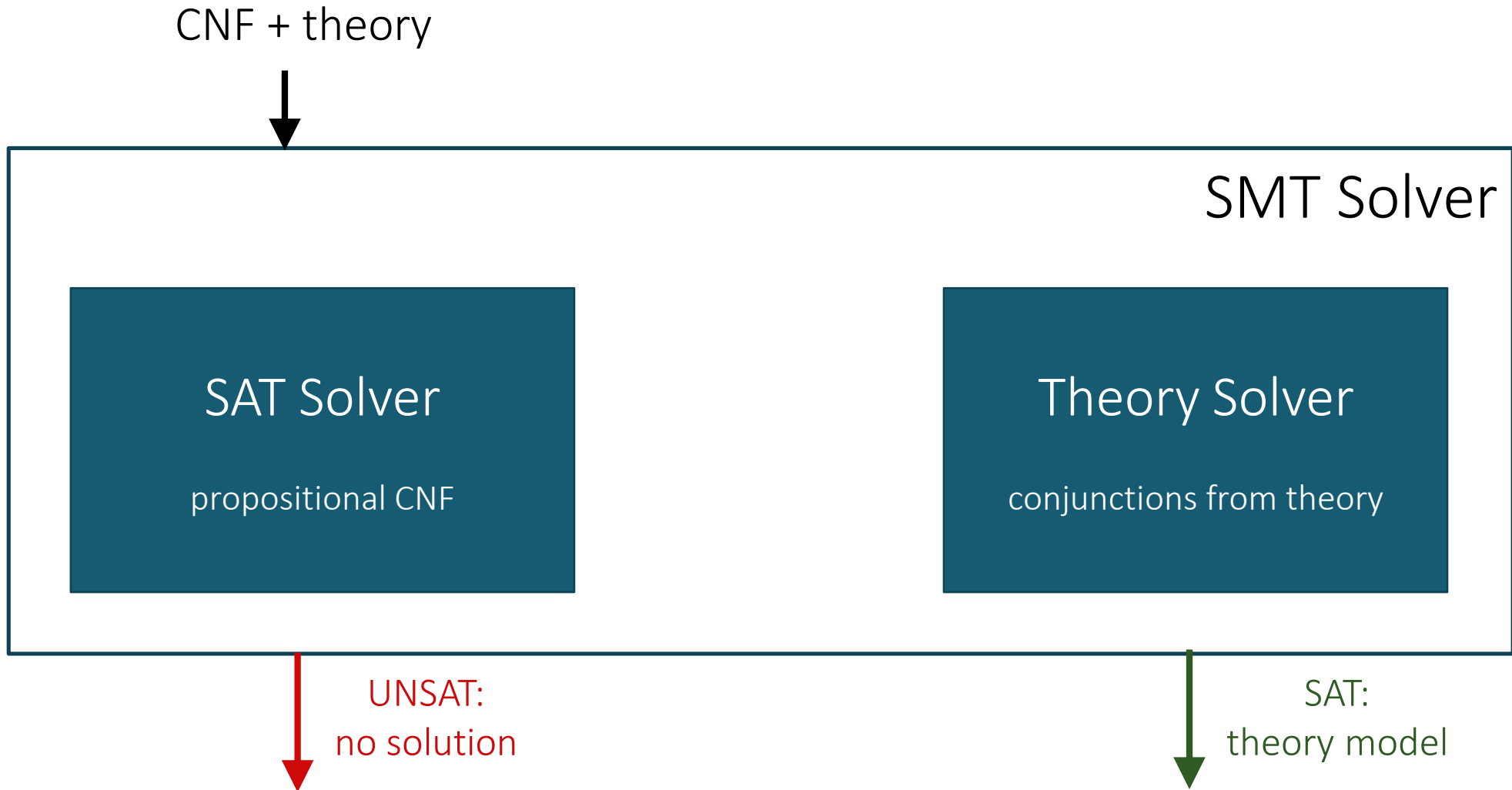
$$a = b$$

EUF solver



Inconsistent!

DPLL(T) architecture



Basic DPLL(T)

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

abstract atoms to
propositional variables

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

SAT solver

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

EUF solver

Inconsistent!

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

SAT solver

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

EUF solver

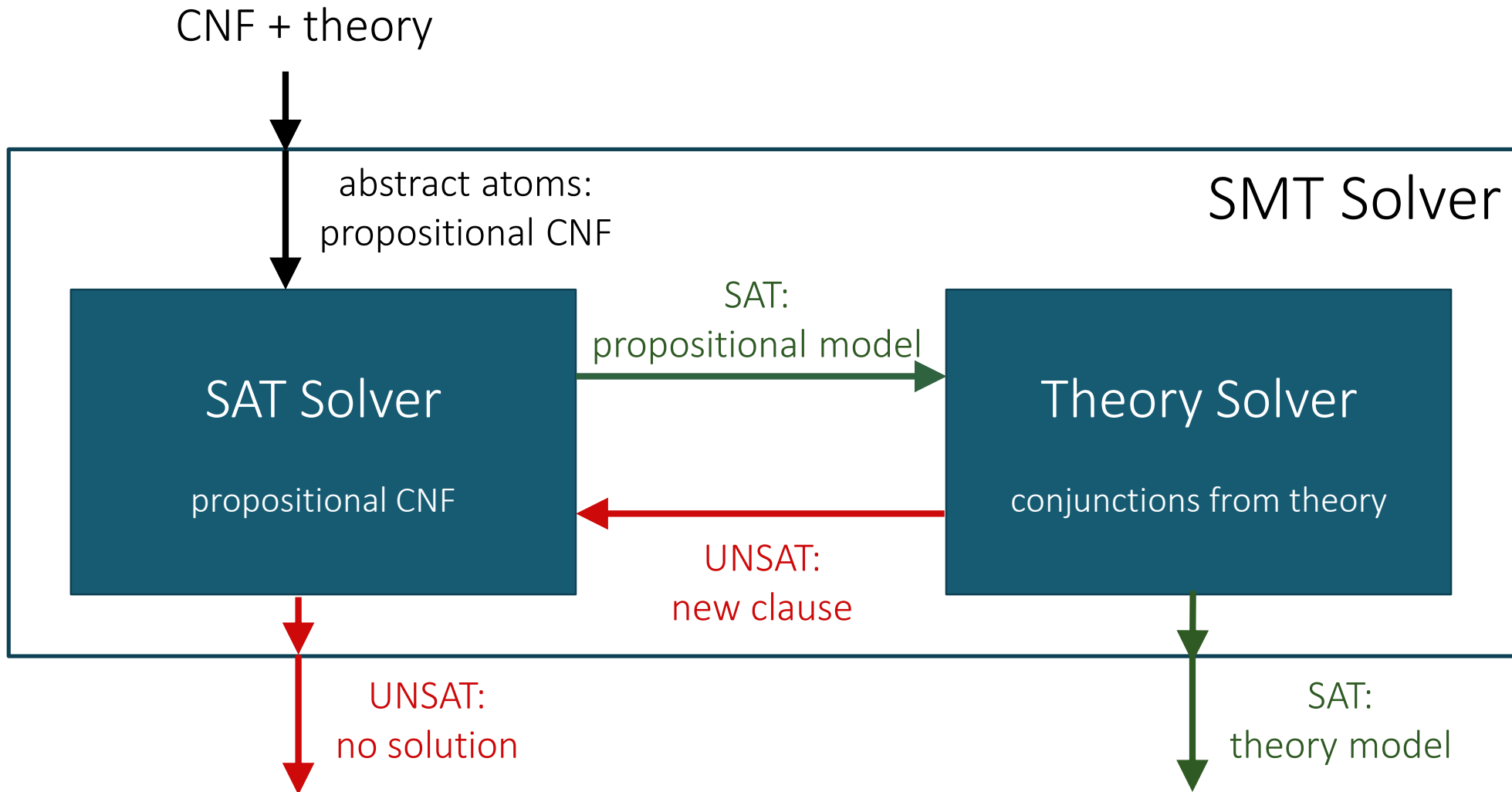
Inconsistent!

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

SAT solver

$$p \wedge (\neg q \vee r) \wedge \neg s \wedge (\neg p \vee q) \wedge (\neg p \wedge \neg r \wedge s) \longrightarrow \text{Unsat}$$

DPLL(T) architecture



Popular theories

Equality and Uninterpreted Functions

$\text{EUF} = \langle \{\mathbf{f}, \mathbf{g}, \mathbf{h}, \dots\}, \{=\}, \text{axioms of equality \& congruence} \rangle$

Linear Integer Arithmetic

$\text{LIA} = \langle \{\mathbf{0}, \mathbf{1}, \dots, +, -\}, \{=, \leq\}, \text{axioms of arithmetic} \rangle$

Arrays

$\text{Arrays} = \langle \{\mathbf{sel}, \mathbf{store}\}, \{=\}, \forall a \ i \ v. \mathbf{sel}(\mathbf{store}(a, i, v), i) = v$
 $\forall a \ i \ j \ v. i \neq j \Rightarrow \mathbf{sel}(\mathbf{store}(a, i, v), j) = \mathbf{sel}(a, j) \ \rangle$

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