Part 1:
DPLL(T) for SMT

Andrew Reynolds

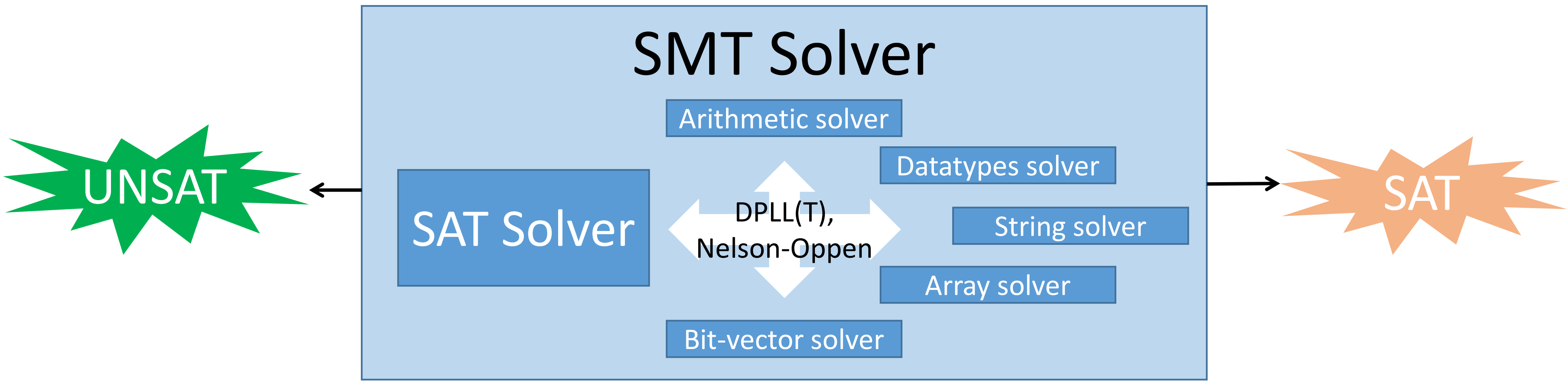
VTSA summer school

August 1, 2017

The University of Iowa

Overview: SMT solvers

- Efficient tools for satisfiability and satisfiability *modulo theories*
- In this talk: focus on *how they work*



- …and how they can be used for *verification and symbolic execution*

How They Work

- SAT : Satisfiability for Propositional Logic

  $(A \vee B) \wedge (C \vee D) \wedge \neg B$

  - Does there exist truth values for A, B, C, D that make this formula true?

    $\Rightarrow$ *Use DPLL algorithm*

How They Work

- SMT : Satisfiability Modulo Theories

  $(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

  - Does there exist integer values for x, y that make this formula true?

    $\Rightarrow$ *Use DPLL(T) algorithm*

How They Work

- SMT : Satisfiability Modulo Theories

  $(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

  - Does there exist integer values for x, y that make this formula true?

    $\Rightarrow$ *Use DPLL(T) algorithm*

- How theories T_1, T_2 can be combined via Nelson-Oppen combination
- *Decision procedures* for theories T
  - ***Theory solvers*** *for* arithmetic, datatypes, UF, arrays, bit-vectors, sets, strings

Propositional Logic

- Formulas are constructed by the following grammar:

$$\psi \quad = \text{Atom} \mid \neg\psi_1 \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \psi_1 \Rightarrow \psi_2 \mid \psi_1 \Leftrightarrow \psi_2$$

$$\text{Atom} \ = A, B, C, \ldots$$

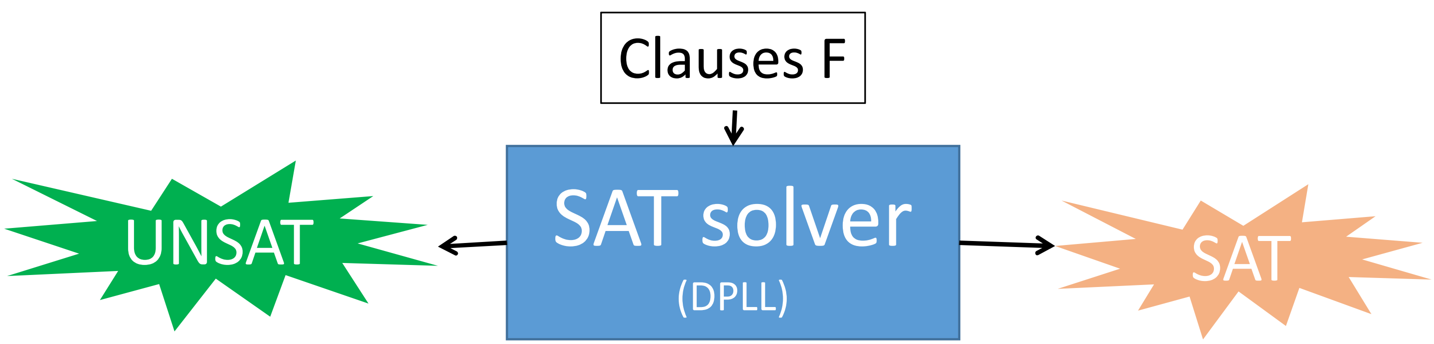
- A formula of the form $\neg A$ or A is called a *literal*
- A disjunction of literals $(L_1 \vee \ldots \vee L_n)$ is called a *clause*
- A *clausal normal form* (CNF) formula is a conjunction of clauses $C_1 \wedge \ldots \wedge C_n$

Propositional Logic

- An *interpretation* maps atoms to truth values:
  - E.g. $\mathscr{I}$= {A→T, B→⊥, C→⊥}
- An interpretation *satisfies* formula F it interprets F as true, e.g.
  - $\mathscr{I} \models A \vee B$ since $A^{\mathscr{I}} \vee B^{\mathscr{I}} = T \vee \bot = T$
  - $\mathscr{I} \models A \wedge (\neg B \vee C)$ since $A^{\mathscr{I}} \wedge (\neg B^{\mathscr{I}} \vee C^{\mathscr{I}}) = T \wedge (\neg\bot \vee \bot) = T$
  - $\mathscr{I} \not\models (\neg A \vee B) \wedge \neg C$ since $(\neg A^{\mathscr{I}} \vee B^{\mathscr{I}}) \wedge \neg C^{\mathscr{I}} = (\neg T \vee \bot) \wedge \neg\bot = \bot$
- A formula is *satisfiable* if there exists an interpretation that satisfies it
  - $(\neg A \vee B) \wedge \neg C$ is satisfiable, by interpretation $\mathscr{I}$={A→⊥, B→T, C→⊥}
  - $A \wedge \neg A$ is unsatisfiable (not satisfied by any interpretation)

DPLL

- DPLL algorithm for propositional satisfiability (SAT)
  - **Input :** a set of clauses F in clausal Normal Form (CNF)
  - Maintains a partial interpretation M, mapping atoms to { T, $\perp$ }
    - Also called an "assignment" or "context"
  - **Output:**
    - "SAT" if an interpretation can be found that satisfies F
    - "UNSAT" otherwise



DPLL

$$(\neg A \Rightarrow B) \wedge (C \vee D) \wedge \neg B$$

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

Convert to CNF

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

- Alternate between:
  - Propagations : assign values to atoms whose value is forced
  - Decisions : choose an arbitrary value for an unassigned atom

DPLL

$$(A \lor B) \land (C \lor D) \land \neg B$$

DPLL

$$(A \lor B) \land (C \lor D) \land \neg B$$

- DPLL algorithm
  - Propagate : B $\rightarrow \bot$

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

- DPLL algorithm
  - Propagate : $B \rightarrow \bot$

Context

$B \rightarrow \bot$

DPLL

$$(A \vee B) \wedge (C \vee D) \wedge \neg B$$

- DPLL algorithm
  - Propagate : B $\rightarrow$ false
  - Propagate : A $\rightarrow$ true

Context

$B \rightarrow \bot$

$A \rightarrow T$

DPLL

$$(A \lor B) \land (C \lor D) \land \neg B$$

- DPLL algorithm
  - Propagate : B $\rightarrow$ false
  - Propagate : A $\rightarrow$ true
  - Decide : C $\rightarrow$ true

Context

$B \rightarrow \bot$

$A \rightarrow T$

$C \rightarrow T^d$

DPLL

$$(A \lor B) \land (C \lor D) \land \neg B$$

- DPLL algorithm
  - Propagate : B $\rightarrow$ false
  - Propagate : A $\rightarrow$ true
  - Decide : C $\rightarrow$ true

$\Rightarrow$ Input is SAT by interpretation where $\{A \rightarrow T, B \rightarrow \bot, C \rightarrow T\}$

Context

$B \rightarrow \bot$

$A \rightarrow T$

$C \rightarrow T^d$

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

Context

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Decide : A $\rightarrow$ true

Context

$A \rightarrow T^{d}$

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Decide : A $\rightarrow$ true

Alternatively, can view context as set of literals

Context
A^d

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Decide : A $\rightarrow$ true
  - Propagate : B $\rightarrow$ true

Context
A^d
B

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Decide : A $\rightarrow$ true
  - Propagate : B $\rightarrow$ true
  - Propagate : C $\rightarrow$ false

Context
A^d
B
$\neg C$

DPLL

$$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$$

$\Rightarrow$ Conflicting clause!

(all literals are false)

- DPLL algorithm
  - Decide : A $\rightarrow$ true
  - Propagate : B $\rightarrow$ true
  - Propagate : C $\rightarrow$ false

Context
A^d
B
$\neg C$

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Decide : A $\rightarrow$ true
  - Propagate : B $\rightarrow$ true
  - Propagate : C $\rightarrow$ false

$\Rightarrow$ Conflicting clause!

(all literals are false)

…*backtrack* on a decision

Context

A^d

B

$\neg C$

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Backtrack : A $\rightarrow$ false

Context
$\neg A$

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Backtrack : A $\rightarrow$ false
  - Propagate : D $\rightarrow$ true

Context
$\neg A$
D

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Backtrack : A $\rightarrow$ false
  - Propagate : D $\rightarrow$ true
  - Decide : B $\rightarrow$ false

Context

$\neg A$

D

B^d

DPLL

$(\neg A \vee B) \wedge (\neg C \vee \neg B) \wedge (C \vee \neg B) \wedge (A \vee D)$

- DPLL algorithm
  - Backtrack : A $\rightarrow$ false
  - Propagate : D $\rightarrow$ true
  - Decide : B $\rightarrow$ false

$\Rightarrow$ Input is SAT by interpretation where $\{A \rightarrow \bot, B \rightarrow \bot, D \rightarrow T\}$

Context
$\neg A$
D
B^d

DPLL

- Important optimizations:
  - Two watched literals
  - Non-chronological backtracking
  - Conflict-driven clause learning (CDCL)
  - Decision heuristics
  - Preprocessing / in-processing

SAT

- Using an encoding of problems into propositional SAT:
  - **Pros** : Decidable, very efficient CDCL-based solvers available
  - **Cons** : Not expressive, may require exponentially large encoding

    $\Rightarrow$ Motivation for Satisfiability *Modulo Theories*

Satisfiability Modulo Theories (SMT)

- Extend SAT problems with reasoning about *theories*
  - E.g. linear integer arithmetic (LIA) : $(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4)$

Satisfiability Modulo Theories (SMT)

- Formally, a theory T is a pair (Σ_T, $\mathscr{I}_T$), where:
  - Σ_T is set of function symbols, the *signature* of T
    - E.g. $\Sigma_{LIA} = \{ +, -, <, \leq, >, \geq, 0, 1, 2, 3, \ldots \}$
  - $\mathscr{I}_T$ is a set of *interpretations* for T
    - E.g. each interpretation in $\mathscr{I}_{LIA}$ interprets functions in Σ_{LIA} in standard way:
      - $1+1 = 2$, $1+2 = 3$, …
      - $1 > 0$ = true, $0 > 1$ = false, …

Satisfiability Modulo Theories

- For theory like arithmetic (Σ_{LIA}, $\mathscr{I}_{LIA}$),
  - A formula F is *LIA-satisfiable* if there is an interpretation in $\mathscr{I}_{LIA}$ that satisfies F

- For example, let $\mathscr{I} \in \mathscr{I}_{LIA}$ where { x→7, y→1}
  - $\mathscr{I} \models$ `x>y+5` since $x^{\mathscr{I}} > y^{\mathscr{I}}+5$ … `7>1+5` … T
  - $\mathscr{I} \models$ `x=y` ∨ `y>0` since $x^{\mathscr{I}} = y^{\mathscr{I}} \vee y^{\mathscr{I}} > 0$ … `7=1`∨`1>0` … T
  - $\mathscr{I} \not\models$ `y-x>0` since $y^{\mathscr{I}} - x^{\mathscr{I}} > 0$ … `1-7>0` … ⊥

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(**T**) algorithm for satisfiability modulo T
  - Extends DPLL algorithm to incorporate reasoning about a theory **T**
  - Basic Idea:
    - Use DPLL algorithm to find assignments for propositional abstraction of formula
      - Use off-the-shelf SAT solver
    - Check the T-satisfiability of assignments found by SAT solver
      - Use *Theory Solver for T*

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm

Invoke DPLL(T) for theory T = LIA (linear integer arithmetic)

DPLL(T)

$$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0$$

- DPLL(LIA) algorithm

Context

DPLL(T)

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false

Context
$\neg x+y>0$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true

Context
$\neg x+y>0$
$x+1>0$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true

Context
$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true

Ĝ ***Unlike propositional SAT case, we must check T-satisfiability of context***

Context
$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on context : $\{ x+1>0, \neg x+y>0, x<0 \}$

Context
$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on context : $\{ x+1>0, \neg x+y>0, x<0 \}$

Context is LIA-unsatisfiable!
$\Rightarrow$ one of $x+1>0$, $x<0$ must be false

Context
$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge (\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on context : $\{ x+1>0, \neg x+y>0, x<0 \}$
    - Add *theory lemma* $(\neg x+1>0 \vee \neg x<0)$

Context
$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0 \land$
$(\neg x+1>0 \lor \neg x<0)$ $\Rightarrow$ Conflicting clause!

…*backtrack* on a decision

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on context : $\{ x+1>0, \neg x+y>0, x<0 \}$
    - Add *theory lemma* $(\neg x+1>0 \lor \neg x<0)$

Context
$\neg x+y>0$
$x+1>0$
$x<0^d$

DPLL(T)

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0 \land$
$(\neg x+1>0 \lor \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true

Context

$\neg x+y>0$

$x+1>0$

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$
$(\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - *Propagate* : $x<0 \rightarrow$ false

Context
$\neg x+y>0$
$x+1>0$
$\neg x<0$

DPLL(T)

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0 \land$
$(\neg x+1>0 \lor \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : x+y>0 $\rightarrow$ false
  - Propagate : x+1>0 $\rightarrow$ true
  - Propagate : x<0 $\rightarrow$ false
  - Propagate : x+y>4 $\rightarrow$ true

Context
$\neg$x+y>0
x+1>0
$\neg$x<0
x+y>4

DPLL(T)

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$
$(\neg x+1>0 \vee \neg x<0)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: $\{ x+1>0, \neg x+y>0, \neg x<0, x+y>4 \}$

Context

$\neg x+y>0$
$x+1>0$
$\neg x<0$
$x+y>4$

DPLL(T)

(x+1>0 $\lor$ x+y>0) $\land$ (x<0 $\lor$ x+y>4) $\land$ $\neg$x+y>0 $\land$ ($\neg$x+1>0 $\lor$ $\neg$x<0)

- DPLL(LIA) algorithm
  - Propagate : x+y>0 $\rightarrow$ false
  - Propagate : x+1>0 $\rightarrow$ true
  - Propagate : x<0 $\rightarrow$ false
  - Propagate : x+y>4 $\rightarrow$ true
  - Invoke theory solver for LIA on: { x+1>0, $\neg$x+y>0, $\neg$x<0, x+y>4 }

Context is LIA-unsatisfiable!
$\Rightarrow$ one of $\neg$x+y>0, x+y>4 must be false

Context
$\neg$x+y>0
x+1>0
$\neg$x<0
x+y>4

DPLL(T)

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0 \land$
$(\neg x+1>0 \lor \neg x<0) \land (\neg x+y>0 \lor x+y>4)$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \to$ false
  - Propagate : $x+1>0 \to$ true
  - Propagate : $x<0 \to$ false
  - Propagate : $x+y>4 \to$ true
  - Invoke theory solver for LIA on: $\{ x+1>0, \neg x+y>0, \neg x<0, x+y>4 \}$
    - Add *theory lemma* $(\neg x+y>0 \lor x+y>4)$

Context
$\neg x+y>0$
$x+1>0$
$\neg x<0$
$x+y>4$

DPLL(T)

$(\, x+1>0 \lor x+y>0 \,) \land (\, x<0 \lor x+y>4 \,) \land \neg x+y>0 \land$
$(\neg x+1>0 \lor \neg x<0 \,) \land (\, \neg x+y>0 \lor x+y>4 \,)$

$\Rightarrow$ Conflicting clause!

...no decision to backtrack

- DPLL(LIA) algorithm
  - Propagate : x+y>0 $\rightarrow$ false
  - Propagate : x+1>0 $\rightarrow$ true
  - Propagate : x<0 $\rightarrow$ false
  - Propagate : x+y>4 $\rightarrow$ true
  - Invoke theory solver for LIA on: { x+1>0, $\neg$x+y>0, $\neg$x<0, x+y>4 }
    - Add *theory lemma* ($\neg$x+y>0 $\lor$ x+y>4)

Context
$\neg$x+y>0
x+1>0
$\neg$x<0
x+y>4

DPLL(T)

$$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0 \wedge$$
$$(\neg x+1>0 \vee \neg x<0) \wedge (\neg x+y>0 \vee x+y>4)$$

⇒ Conflicting clause!

…no decision to backtrack

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Propagate : $x<0 \rightarrow$ false
  - Propagate : $x+y>4 \rightarrow$ true
  - Invoke theory solver for LIA on: $\{ x+1>0, \neg x+y>0, \neg x<0, x+y>4 \}$
    - Add *theory lemma* $(\neg x+y>0 \vee x+y>4)$

⇒ *Input is* LIA-unsat

Context
$\neg x+y>0$
$x+1>0$
$\neg x<0$
$x+y>4$

DPLL(T) : Exercise

$(x>y \vee x>z) \wedge (x+1<y \vee \neg x>y) \wedge (x>y \vee z>y)$ } Φ

- Determine if Φ is LIA-satisfiable
  - If SAT, give values for { x, y, z }
  - If UNSAT, give a set of clauses $C_1, \ldots, C_n$, where:
    - $\Phi, C_1, \ldots, C_n$, does not have a Boolean satisfying assignment
    - Each C_i is of the form $(l_1 \vee \ldots \vee l_m)$, where:
      - Each l_i is one of $(\neg)x>y$, $(\neg)x>z$, $(\neg)x+1<y$, $(\neg)z>y$
      - Negation of $l_1 \ldots l_m$ are LIA-unsatisfiable

DPLL(T) : Exercise

$(x>y \lor x>z) \land (x+1<y \lor \neg x>y) \land (x>y \lor z>y)$

- DPLL(LIA) algorithm

Context

DPLL(T) : Exercise

$(x>y \vee x>z) \wedge (x+1<y \vee \neg x>y) \wedge (x>y \vee z>y)$

- DPLL(LIA) algorithm
  - Decide : $x>y \rightarrow$ true

Context
$x>y^d$

DPLL(T) : Exercise

$(x>y \lor x>z) \land (x+1<y \lor \neg x>y) \land (x>y \lor z>y)$

- DPLL(LIA) algorithm
  - Decide : x>y $\rightarrow$ true
  - Propagate x+1<y $\rightarrow$ true

Context
$x>y^d$
$x+1<y$

DPLL(T) : Exercise

(x>y $\vee$ x>z) $\wedge$ (x+1<y $\vee$ $\neg$x>y) $\wedge$ (x>y $\vee$ z>y)

- DPLL(LIA) algorithm
  - Decide : x>y $\rightarrow$ true
  - Propagate x+1<y $\rightarrow$ true
  - Invoke theory solver for LIA on: { x>y, x+1<y }

Context
x>y^d
x+1<y

DPLL(T) : Exercise

Context
$x>y^d$
$x+1<y$

$(x>y \lor x>z) \land (x+1<y \lor \neg x>y) \land (x>y \lor z>y) \land (\neg x>y \lor \neg x+1<y)$

- DPLL(LIA) algorithm
  - Decide : $x>y \rightarrow$ true
  - Propagate $x+1<y \rightarrow$ true
  - Invoke theory solver for LIA on: $\{ x>y, x+1<y \}$
    - $x>y \land x+1<y$ is LIA-unsatisfiable, add $(\neg x>y \lor \neg x+1<y)$

DPLL(T) : Exercise

$(\ x>y \lor x>z\ ) \land (\ x+1<y \lor \neg x>y\ ) \land (\ x>y \lor z>y\ ) \land$
$(\neg x>y \lor \neg x+1<y)$ $\Rightarrow$ Conflicting clause!

...*backtrack* on a decision

- DPLL(LIA) algorithm
  - Decide : x>y $\rightarrow$ true
  - Propagate x+1<y $\rightarrow$ true
  - Invoke theory solver for LIA on: { x>y, x+1<y }
    - x>y $\land$ x+1<y is LIA-unsatisfiable, add ($\neg$x>y $\lor$ $\neg$x+1<y)

Context
$x>y^d$
x+1<y

DPLL(T) : Exercise

$(x>y \vee x>z) \wedge (x+1<y \vee \neg x>y) \wedge (x>y \vee z>y) \wedge$
$(\neg x>y \vee \neg x+1<y)$

- DPLL(LIA) algorithm
  - *Backtrack* : $x>y \rightarrow$ false

Context

$\neg x>y$

DPLL(T) : Exercise

$(x>y \vee x>z) \wedge (x+1<y \vee \neg x>y) \wedge (x>y \vee z>y) \wedge (\neg x>y \vee \neg x+1<y)$

- DPLL(LIA) algorithm
  - Backtrack : $x>y \rightarrow$ false
  - Propagate : $x>z \rightarrow$ true

Context

$\neg x>y$

$x>z$

DPLL(T) : Exercise

$(x>y \lor x>z) \land (x+1<y \lor \neg x>y) \land (x>y \lor z>y) \land (\neg x>y \lor \neg x+1<y)$

- DPLL(LIA) algorithm
  - Backtrack : $x>y \rightarrow$ false
  - Propagate : $x>z \rightarrow$ true
  - Propagate : $z>y \rightarrow$ true

Context

$\neg x>y$

$x>z$

$z>y$

DPLL(T) : Exercise

$(x>y \vee x>z) \wedge (x+1<y \vee \neg x>y) \wedge (x>y \vee z>y) \wedge$
$(\neg x>y \vee \neg x+1<y)$

- DPLL(LIA) algorithm
  - Backtrack : $x>y \rightarrow$ false
  - Propagate : $x>z \rightarrow$ true
  - Propagate : $z>y \rightarrow$ true
  - Invoke theory solver for LIA on: $\{ \neg x>y, x>z, z>y \}$

Context
$\neg x>y$
$x>z$
$z>y$

DPLL(T) : Exercise

$(x>y \vee x>z) \wedge (x+1<y \vee \neg x>y) \wedge (x>y \vee z>y) \wedge$
$(\neg x>y \vee \neg x+1<y) \wedge (x>y \vee \neg x>z \vee \neg z>y)$

- DPLL(LIA) algorithm
  - Backtrack : $x>y \rightarrow$ false
  - Propagate : $x>z \rightarrow$ true
  - Propagate : $z>y \rightarrow$ true
  - Invoke theory solver for LIA on: { $\neg x>y$, $x>z$, $z>y$ }
    - $\neg x>y \wedge x>z \wedge z>y$ is LIA-unsatisfiable, add $(x>y \vee \neg x>z \vee \neg z>y)$

Context

$\neg x>y$

$x>z$

$z>y$

DPLL(T) : Exercise

$(x>y \lor x>z) \land (x+1<y \lor \neg x>y) \land (x>y \lor z>y) \land$
$(\neg x>y \lor \neg x+1<y) \land (x>y \lor \neg x>z \lor \neg z>y)$

$\Rightarrow$ Conflicting clause!

…no decision to backtrack

- DPLL(LIA) algorithm
  - Backtrack : $x>y \rightarrow$ false
  - Propagate : $x>z \rightarrow$ true
  - Propagate : $z>y \rightarrow$ true
  - Invoke theory solver for LIA on: $\{ \neg x>y, x>z, z>y \}$
    - $\neg x>y \land x>z \land z>y$ is LIA-unsatisfiable, add $(x>y \lor \neg x>z \lor \neg z>y)$

Context
$\neg x>y$
$x>z$
$z>y$

DPLL(T) : Exercise

$(x>y \vee x>z) \wedge (x+1<y \vee \neg x>y) \wedge (x>y \vee z>y) \wedge$
$(\neg x>y \vee \neg x+1<y) \wedge (x>y \vee \neg x>z \vee \neg z>y)$

$\Rightarrow$ Conflicting clause!

…no decision to backtrack

- DPLL(LIA) algorithm
  - Backtrack : $x>y \rightarrow$ false
  - Propagate : $x>z \rightarrow$ true
  - Propagate : $z>y \rightarrow$ true
  - Invoke theory solver for LIA on: $\{ \neg x>y, x>z, z>y \}$
    - $\neg x>y \wedge x>z \wedge z>y$ is LIA-unsatisfiable, add $(x>y \vee \neg x>z \vee \neg z>y)$

$\Rightarrow$ *Input is* LIA-unsat

Context

$\neg x>y$

$x>z$

$z>y$

DPLL(T) : Exercise

$(x>y \vee x>z) \wedge (x+1<y \vee \neg x>y) \wedge (x>y \vee z>y) \wedge$
$(\neg x>y \vee \neg x+1<y) \wedge (x>y \vee \neg x>z \vee \neg z>y)$

- DPLL(LIA) algorithm
  - Backtrack : $x>y \rightarrow$ false
  - Propagate : $x>z \rightarrow$ true
  - Propagate : $z>y \rightarrow$ true
  - Invoke theory solver for LIA on: $\{ \neg x>y, x>z, z>y \}$
    - $\neg x>y \wedge x>z \wedge z>y$ is LIA-unsatisfiable, add $(x>y \vee \neg x>z \vee \neg z>y)$

$\Rightarrow$ *Input is* LIA-unsat

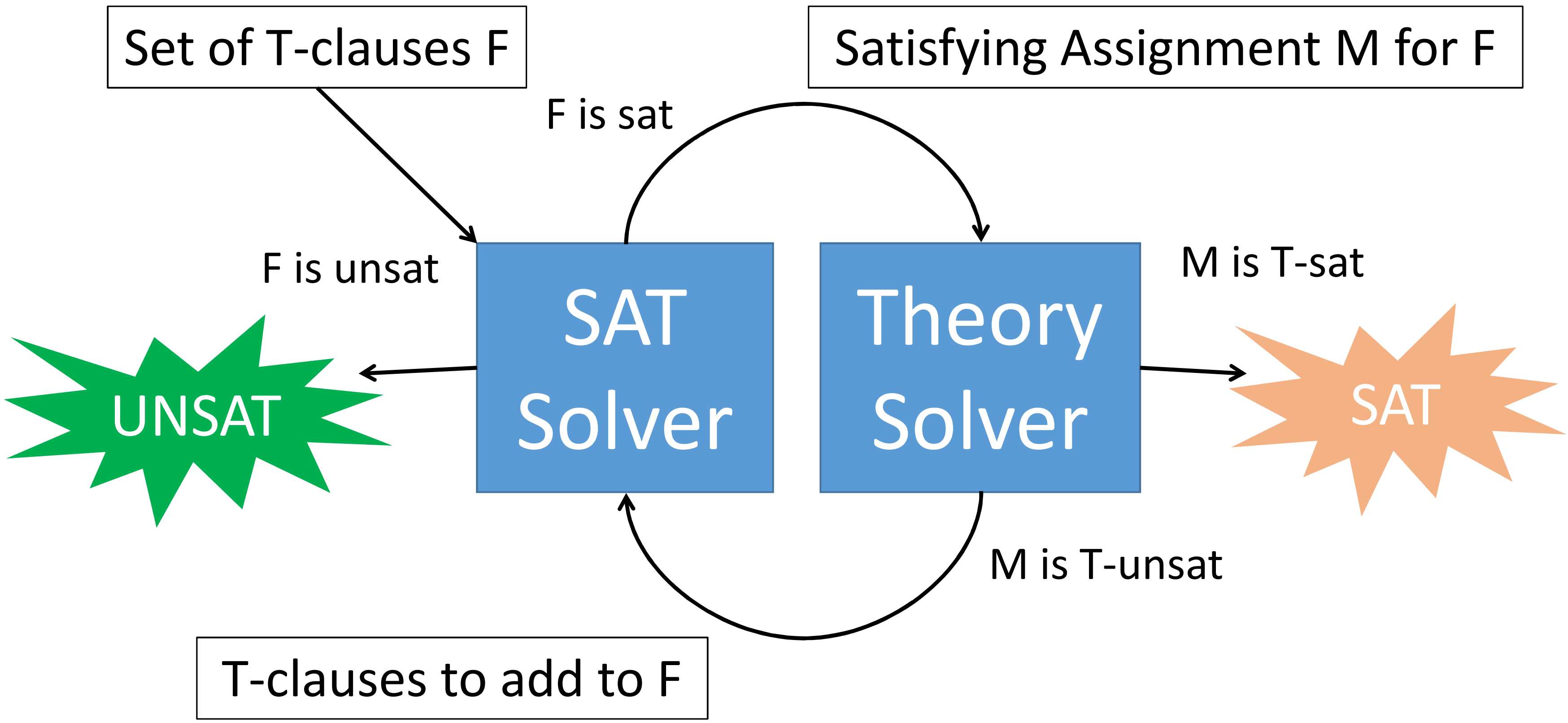
Context
$\neg x>y$
$x>z$
$z>y$

Encoding in *.smt2 format

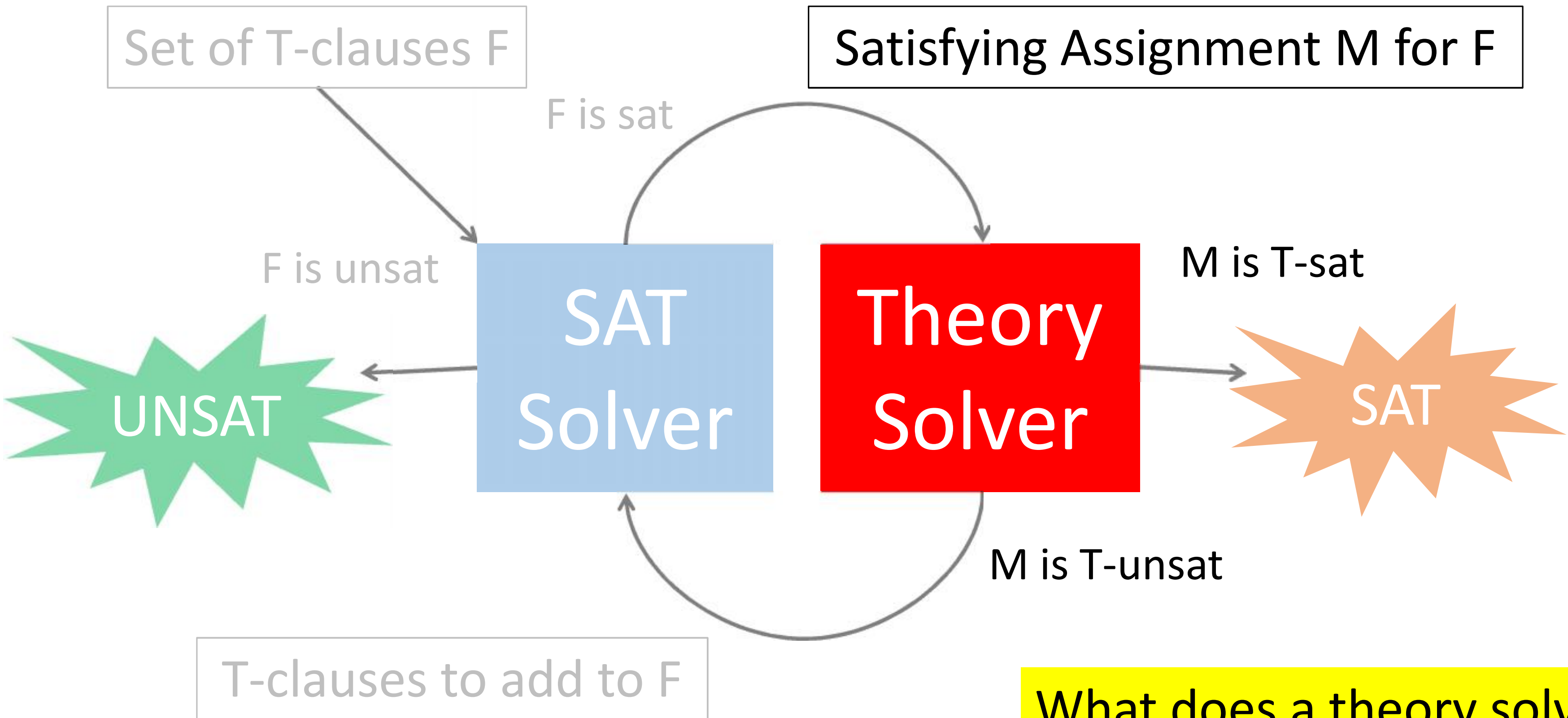
```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (or (> x y) (> x z)))
(assert (or (< (+ x 1) y) (not (> x y))))
(assert (or (> x y) (> z y))))
(check-sat)
```

EXAMPLE 1…

DPLL(T)


Set of T-clauses F
Satisfying Assignment M for F
F is sat
F is unsat
SAT Solver
Theory Solver
M is T-sat
UNSAT
SAT
M is T-unsat
T-clauses to add to F

DPLL(T)



What does a theory solver do?

DPLL(T) Theory Solvers

- Input : A set of T-literals M
- Output : either
  1. M is T-satisfiable
  2. $\{ l_1, \ldots, l_n \} \subseteq M$ is T-unsatisfiable
  3. Don't know

DPLL(T) Theory Solvers

- Input : A set of T-literals M
- Output : either
  1. M is T-satisfiable
     - Return *model*, e.g. { x $\rightarrow$ 2, y $\rightarrow$ 3, z $\rightarrow$ -3, … }
  2. { l_1, …, l_n } $\subseteq$ M is T-unsatisfiable
  3. Don’t know

DPLL(T) Theory Solvers

- Input : A set of T-literals M
- Output : either
  1. M is T-satisfiable
     - Return *model*, e.g. { x $\rightarrow$ 2, y $\rightarrow$ 3, z $\rightarrow$ -3, … }
  2. $\{ l_1, \ldots, l_n \} \subseteq M$ is T-unsatisfiable
     - Return *conflict* clause ($\neg l_1 \vee \ldots \vee \neg l_n$)
  3. Don’t know

DPLL(T) Theory Solvers

- Input : A set of T-literals M
- Output : either
  1. M is T-satisfiable
     - Return *model*, e.g. $\{ x \to 2, y \to 3, z \to -3, \ldots \}$
  2. $\{ l_1, \ldots, l_n \} \subseteq M$ is T-unsatisfiable
     - Return *conflict* clause $(\neg l_1 \vee \ldots \vee \neg l_n)$
  3. Don't know
     - Return lemma, e.g. splitting on demand $(x = y \vee \neg x=y)$

DPLL(T) Theory Solvers

- Input : A set of T-literals M
- Output : either
  1. M is T-satisfiable
     - Return *model*, e.g. { $x \to 2$, $y \to 3$, $z \to -3$, … }

     $\Rightarrow$ Should be ***solution-sound***
       - Answers "M is T-satisfiable" only if M is T-satisfiable
  2. $\{ l_1, \ldots, l_n \} \subseteq M$ is T-unsatisfiable
     - Return *conflict* clause ($\neg l_1 \vee \ldots \vee \neg l_n$)
  3. Don't know
     - Return lemma, e.g. splitting on demand ($x = y \vee \neg x=y$)

DPLL(T) Theory Solvers

- Input : A set of T-literals M
- Output : either
  1. M is T-satisfiable
     - Return *model*, e.g. { $x \rightarrow 2$, $y \rightarrow 3$, $z \rightarrow -3$, … }

     $\Rightarrow$ Should be ***solution-sound***
       - Answers “M is T-satisfiable” only if M is T-satisfiable
  2. { l_1, …, l_n } $\subseteq$ M is T-unsatisfiable
     - Return *conflict* clause ($\neg l_1 \vee \ldots \vee \neg l_n$)

     $\Rightarrow$ Should be ***refutation-sound***
       - Answers “{ l_1, …, l_n } is T-unsatisfiable” only if { l_1, …, l_n } is T-unsatisfiable
  3. Don’t know
     - Return lemma, e.g. splitting on demand ($x = y \vee \neg x=y$)

DPLL(T) Theory Solvers

- Input : A set of T-literals M
- Output : either
  1. M is T-satisfiable
     - Return *model*, e.g. { $x \rightarrow 2$, $y \rightarrow 3$, $z \rightarrow -3$, … }

     $\Rightarrow$ Should be *solution-sound*
       - Answers "M is T-satisfiable" only if M is T-satisfiable
  2. $\{ l_1, \ldots, l_n \} \subseteq M$ is T-unsatisfiable
     - Return *conflict* clause ($\neg l_1 \vee \ldots \vee \neg l_n$)

     $\Rightarrow$ Should be *refutation-sound*
       - Answers "$\{ l_1, \ldots, l_n \}$ is T-unsatisfiable" only if $\{ l_1, \ldots, l_n \}$ is T-unsatisfiable
  3. Don't know
     - Return lemma, e.g. splitting on demand ($x = y \vee \neg x=y$)

  $\Rightarrow$If solver is solution-sound, refutation-sound, and *terminating*,
     - Then it is a *decision procedure* for T

Design of DPLL(T) Theory Solvers

- A DPLL(T) theory solver:
  - Should be solution-sound, refutation-sound, terminating
  - Should produce models when M is T-satisfiable
  - Should produce T-conflicts of minimal size when M is T-unsatisfiable
  - Should be designed to work *incrementally*
    - M is constantly being appended to/backtracked upon
  - Can be designed to check T-satisfiability either:
    - Eagerly: Check if M is T-satisfiable immediately when any literal is added to M
    - Lazily: Check if M is T-satisfiable only when M is complete
  - Should cooperate with other theory solvers when combining theories
    - (see later)

DPLL(T) Theory Solvers : Examples

- SMT solvers incorporate:
  - Theory solvers that are *decision procedures* for e.g.:
    - Theory of Equality and Uninterpreted Functions (EUF)
    - Theory of Linear Integer/Real Arithmetic
    - Theory of Arrays
    - Theory of Bit Vectors
    - Theory of Inductive Datatypes
    - …and many others
  - Theory solvers that are *incomplete procedures* for e.g.:
    - Theory of Non-Linear Integer Arithmetic
    - Theory of Strings + Length constraints

DPLL(T) Theory Solvers : Examples

- SMT solvers incorporate:
  - Theory solvers that are *decision procedures* for e.g.:
    - Theory of Equality and Uninterpreted Functions (EUF)
    - Theory of Linear Integer/Real Arithmetic
    - Theory of Arrays
    - Theory of Bit Vectors
    - *Theory of Inductive Datatypes* } Focus of the next part
    - …and many others
  - Theory solvers that are *incomplete procedures* for e.g.:
    - Theory of Non-Linear Integer Arithmetic
    - Theory of Strings + Length constraints

Theory of Inductive Datatypes : Example

```
ClrList := cons( head : Clr, tail : ClrList ) | nil
Clr := red | green | blue
```

Theory of Inductive Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil Clr := red | green | blue

- Theory of Inductive Datatypes (DT) for ClrList and Clr
  - Σ_{DT} : { cons, head, tail, nil, red, green, blue }
  - Interpretations I_{DT} are such that:
    - Terms with different constructors are distinct
      - red $\neq$ green
    - Constructors are injective
      - If cons(c_1, l_1) = cons(c_2, l_2), then $c_1 = c_2$ and $l_1 = l_2$
    - Terms of a datatype must have one of its constructors as its topmost symbol
      - Each c is such that c = red or c = green or c = blue
    - Selectors access subfields
      - head(cons(c, l)) = c
    - Terms do not contain themselves as subterms
      - l $\neq$ cons(c, l)

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

$cons(x,nil)=cons(y,z) \wedge (x=red \vee \neg x=y) \wedge y = green$

Context

- DPLL(DT) algorithm

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green

Context

cons(x,nil)=cons(y,z)
y=green

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green

Context

cons(x,nil)=cons(y,z)
y=green
x=red^{d}

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Decide : x=red $\rightarrow$ true

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Decide : x=red $\rightarrow$ true
  - Invoke DT solver on {cons(x,nil)=cons(y,z),y=green,x=red}

Context

cons(x,nil)=cons(y,z)
y=green
$x=red^d$

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green

Context
cons(x,nil)=cons(y,z) y=green x=red^{d}

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Decide : x=red $\rightarrow$ true
  - Invoke DT solver on {cons(x,nil)=cons(y,z),y=green,x=red}

    $\Rightarrow$DT-unsatisfiable

    Since cons(x, nil) = cons(y, nil), it must be that x = y, but x = red and y = green and red $\neq$ green

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

$\text{cons}(x,\text{nil})=\text{cons}(y,z) \wedge (x=\text{red} \vee \neg x=y) \wedge y = \text{green}$

$(\neg\text{cons}(x,\text{nil})=\text{cons}(y,z) \vee \neg\, y=\text{green} \vee \neg\, x=\text{red})$

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Decide : x=red $\rightarrow$ true
  - Invoke DT solver on {cons(x,nil)=cons(y,z),y=green,x=red}

    $\Rightarrow$ …add theory lemma

Context
cons(x,nil)=cons(y,z)
y=green
$x=\text{red}^{d}$

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)

$\Rightarrow$ Conflicting clause!

...*backtrack* on a decision

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Decide : x=red $\rightarrow$ true
  - Invoke DT solver on {cons(x,nil)=cons(y,z),y=green,x=red}

    $\Rightarrow$ ...add theory lemma

Context
cons(x,nil)=cons(y,z)
y=green
x=red^d

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true

Context
cons(x,nil)=cons(y,z) y=green

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Propagate : x=red $\rightarrow$ false

Context
cons(x,nil)=cons(y,z) y=green $\neg$x=red

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Propagate : x=red $\rightarrow$ false
  - Propagate : x=y $\rightarrow$ false

Context

cons(x,nil)=cons(y,z)
y=green
$\neg$x=red
$\neg$x=y

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Propagate : x=red $\rightarrow$ false
  - Propagate : x=y $\rightarrow$ false
  - Invoke DT solver on {cons(x,nil)=cons(y,z), y=green, x=red, x=y}

Context
cons(x,nil)=cons(y,z)
y=green
$\neg$x=red
$\neg$x=y

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)
($\neg$cons(x,nil)=cons(y,z) $\vee$ x=y)

- DPLL(DT) algorithm
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Propagate : x=red $\rightarrow$ false
  - Propagate : x=y $\rightarrow$ false
  - Invoke DT solver on {cons(x,nil)=cons(y,z), y=green, x=red, x=y}

    $\Rightarrow$ DT-unsatisfiable, add theory lemma

Context

cons(x,nil)=cons(y,z)
y=green
$\neg$x=red
$\neg$x=y

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) $\wedge$ (x=red $\vee$ $\neg$x=y) $\wedge$ y = green
($\neg$cons(x,nil)=cons(y,z) $\vee$ $\neg$ y=green $\vee$ $\neg$ x=red)
($\neg$cons(x,nil)=cons(y,z) $\vee$ x=y) $\Rightarrow$ Conflicting clause!

- DPLL(DT) algorithm *…no decisions*
  - Propagate : cons(x,nil)=cons(y,z) $\rightarrow$ true
  - Propagate : y=green $\rightarrow$ true
  - Propagate : x=red $\rightarrow$ false
  - Propagate : x=y $\rightarrow$ false
  - Invoke DT solver on {cons(x,nil)=cons(y,z), y=green, x=red, x=y}

    $\Rightarrow$ DT-unsatisfiable, add theory lemma

Context
cons(x,nil)=cons(y,z) y=green $\neg$x=red $\neg$x=y

Datatypes : Example

ClrList := cons(head : Clr, tail : ClrList) | nil
Clr := red | green | blue

cons(x,nil)=cons(y,z) ∧ (x=red ∨ ¬x=y) ∧ y = green
(¬cons(x,nil)=cons(y,z) ∨ ¬ y=green ∨ ¬ x=red)
(¬cons(x,nil)=cons(y,z) ∨ x=y) ⇒ Conflicting clause!

- DPLL(DT) algorithm *...no decisions*
  - Propagate : cons(x,nil)=cons(y,z) → true
  - Propagate : y=green → true
  - Propagate : x=red → false
  - Propagate : x=y → false
  - Invoke DT solver on {cons(x,nil)=cons(y,z), y=green, x=red, x=y}

⇒ Input is DT-unsat

Context
cons(x,nil)=cons(y,z)
y=green
¬x=red
¬x=y

Encoding in *.smt2

```
(set-logic QF_DT)
(declare-datatypes ((ClrList 0) (Clr 0))(
 ((cons (head Clr) (tail ClrList)) (nil))
 ((red) (green) (blue))))
(declare-fun x () Clr)
(declare-fun y () Clr)
(declare-fun z () ClrList)
(assert (= (cons x nil) (cons y z))
(assert (or (= x red) (not (= x y))))
(assert (= y green))
(check-sat)
```

EXAMPLE 2…

Theory Combination

- What if we have:

$$\text{IntList} := \text{cons}(\ \text{head} : \text{Int},\ \text{tail} : \text{IntList}\ ) \mid \text{nil}$$

- Example input:

$$(\ \text{head}(\ x\ )+3 = y \vee x = \text{cons}(\ y+1, \text{nil}\ )\ ) \wedge \text{head}(\ x\ ) > y+1$$

$\Rightarrow$Requires reasoning about datatypes *and* integers

Theory Combination

- What if we have:

IntList := cons(head : Int, tail : IntList) | nil

- Example input:

$$(head(x)+3 = y \lor x = cons(y+1, nil)) \land head(x) > y+1$$

- Idea:
  - *Purify* the literals in the input
  - Use DPLL(LIA+DT): find satisfying assignments $M = M_{LIA} \cup M_{DT}$
    - Use existing solver for LIA to check if M_{LIA} is LIA-satisfiable
    - Use existing solver for DT to check if M_{DT} is DT-satisfiable
  - If either of { M_{LIA}, M_{DT} } is T-unsatisfiable, then M is T-unsatisfiable
  - If both { M_{LIA}, M_{DT} } are T-satisfiable, then solvers must combine models
    - Must agree on equalities between *shared variables*

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$$(\text{head}(x)+3 = y \lor x = \text{cons}(y+1, \text{nil})) \land \text{head}(x) > y+1$$

Context

- DPLL(LIA+DT) algorithm

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

(u_1 +3 = y $\vee$ x = cons(u_2 , nil)) $\wedge$ u_1 > y+1 $\wedge$
u_1=head(x) $\wedge$ u_2=y+1

Context

- DPLL(LIA+DT) algorithm
  - $\Rightarrow$First, purify the input
    - Introduce *shared variables* u_1, u_2

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

Context

- DPLL(LIA+DT) algorithm

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true

$IntList := cons(head : Int, tail : IntList) \mid nil$

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

$u_1+3=y^d$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Decide : $u_1+3=y \rightarrow$ true

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

Context
$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$u_1+3=y^d$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Decide : $u_1+3=y \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x)\}$ … DT-satisfiable

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

$u_1+3=y^d$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Decide : $u_1+3=y \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x)\}$ … DT-satisfiable
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, u_1+3=y\}$

$\text{IntList} := \text{cons}(\ \text{head} : \text{Int},\ \text{tail} : \text{IntList}\ )\ |\ \text{nil}$

Theory Combination

$(u_1+3=y \lor x = \text{cons}(u_2, \text{nil})\ ) \land u_1>y+1 \land u_1=\text{head}(x) \land u_2=y+1$

$(\neg u_1>y+1 \lor \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=\text{head}(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Decide : $u_1+3=y \rightarrow$ true
  - Invoke DT solver on $\{u_1=\text{head}(x)\}$ … DT-satisfiable
  - Invoke LIA solver on $\{u_1>y+1,\ u_2=y+1,\ u_1+3=y\}$…LIA-unsatisfiable

    $\Rightarrow$ Add theory lemma

Context

$u_1>y+1$

$u_1=\text{head}(x)$

$u_2=y+1$

$u_1+3=y^d$

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

$(\neg u_1>y+1 \vee \neg u_1+3=y)$

$\Rightarrow$ Conflicting clause!

…*backtrack* on a decision

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Decide : $u_1+3=y \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x)\}$ … DT-satisfiable
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, u_1+3=y\}$…LIA-unsatisfiable

    $\Rightarrow$ Add theory lemma

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

$u_1+3=y^d$

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true

Context
$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

$IntList := cons(head : Int, tail : IntList) \mid nil$

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

$(\neg u_1>y+1 \vee \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable

Context
$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

$(\neg u_1>y+1 \vee \neg u_1+3=y)$

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

$\neg u_1+3=y$

$x=cons(u_2, nil)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y\}$ … LIA-satisfiable

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y\}$ … LIA-satisfiable

  $\Rightarrow$Theory solvers must agree on shared variables u_1, u_2

Context
$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable, $u_1=u_2$
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y\}$ … LIA-satisfiable

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y)$

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable, $u_1=u_2$
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y\}$ … LIA-satisfiable, $u_1 \neq u_2$

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

$(\neg u_1>y+1 \vee \neg u_1+3=y)$

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

$\neg u_1+3=y$

$x=cons(u_2, nil)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable, **$u_1=u_2$**
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y\}$ … LIA-satisfiable, **$u_1 \div u_2$**

  $\Rightarrow$ *Theory solvers do not agree on $u_1=u_2$!*

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil)\}$ … DT-satisfiable, $u_1=u_2$
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y\}$ … LIA-satisfiable, $u_1 \neq u_2$

  $\Rightarrow$ *Theory solvers do not agree on $u_1=u_2$ … add splitting lemma for u_1, u_2*

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true

Context
$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \lor x = cons(u_2, nil)\ ) \land u_1>y+1 \land u_1=head(x) \land u_2=y+1$

$(\neg u_1>y+1 \lor \neg u_1+3=y) \land (\ u_1=u_2 \lor \neg u_1=u_2\ )$

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

$\neg u_1+3=y$

$x=cons(u_2, nil)$

$u_1=u_2{}^d$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Decide : $u_1=u_2 \rightarrow$ true

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$

$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Decide : $u_1=u_2 \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil), u_1=u_2\}$ … DT-satisfiable

Context

$u_1>y+1$

$u_1=head(x)$

$u_2=y+1$

$\neg u_1+3=y$

$x=cons(u_2, nil)$

$u_1=u_2^d$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \lor x = cons(u_2, nil)) \land u_1>y+1 \land u_1=head(x) \land u_2=y+1$
$(\neg u_1>y+1 \lor \neg u_1+3=y) \land (u_1=u_2 \lor \neg u_1=u_2) \land (\neg u_1>y+1 \lor$
$\neg u_2=y+1 \lor \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Decide : $u_1=u_2 \rightarrow$ true
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil), u_1=u_2\}$ … DT-satisfiable
  - Invoke LIA solver on $\{u_1>y+1, u_2=y+1, \neg u_1+3=y, u_1=u_2\}$ … LIA-unsatisfiable

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$
$u_1=u_2{}^d$

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2) \wedge (\neg u_1>y+1 \vee$
$\neg u_2=y+1 \vee \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true

Context
$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$

Theory Combination

IntList := cons(head : Int, tail : IntList) | nil

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2) \wedge (\neg u_1>y+1 \vee$
$\neg u_2=y+1 \vee \neg u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Propagate : $u_1=u_2 \rightarrow$ false

Context
$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$
$\neg u_1=u_2$ d

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2) \wedge (\neg u_1>y+1 \vee$
$\neg u_2=y+1 \vee \neg u_1=u_2) \wedge (\neg u_1=head(x) \vee \neg x=cons(u_2, nil) \vee u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Propagate : $u_1=u_2 \rightarrow$ false
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil), \neg u_1=u_2\}$ …DT-unsat

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$
$\neg u_1=u_2{}^d$

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2) \wedge (\neg u_1>y+1 \vee$
$\neg u_2=y+1 \vee \neg u_1=u_2) \wedge (\neg u_1=head(x) \vee \neg x=cons(u_2, nil) \vee u_1=u_2)$

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Propagate : $u_1=u_2 \rightarrow$ false
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil), \neg u_1=u_2\}$ …DT-unsat

$\Rightarrow$ Conflicting clause!

…no decisions

Context
$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$
$\neg u_1=u_2{}^d$

IntList := cons(head : Int, tail : IntList) | nil

Theory Combination

$(u_1+3=y \vee x = cons(u_2, nil)) \wedge u_1>y+1 \wedge u_1=head(x) \wedge u_2=y+1$
$(\neg u_1>y+1 \vee \neg u_1+3=y) \wedge (u_1=u_2 \vee \neg u_1=u_2) \wedge (\neg u_1>y+1 \vee$
$\neg u_2=y+1 \vee \neg u_1=u_2) \wedge (\neg u_1=head(x) \vee \neg x=cons(u_2, nil) \vee u_1=u_2)$

$\Rightarrow$ Conflicting clause!

...no decisions

- DPLL(LIA+DT) algorithm
  - Propagate : $u_1>y+1 \rightarrow$ true
  - Propagate : $u_1=head(x) \rightarrow$ true
  - Propagate : $u_2=y+1 \rightarrow$ true
  - Propagate : $u_1+3=y \rightarrow$ false
  - Propagate : $x=cons(u_2, nil) \rightarrow$ true
  - Propagate : $u_1=u_2 \rightarrow$ false
  - Invoke DT solver on $\{u_1=head(x), x=cons(u_2, nil), \neg u_1=u_2\}$ ...DT-unsat

$\Rightarrow$ Input is LIA+DT-unsat

Context

$u_1>y+1$
$u_1=head(x)$
$u_2=y+1$
$\neg u_1+3=y$
$x=cons(u_2, nil)$
$\neg u_1=u_2{}^d$

Encoding in *.smt2

```
(set-logic QF_DTLIA)
(declare-datatypes ((IntList 0))(
 ((cons (head Int) (tail IntList)) (nil))))
(declare-fun x () IntList)
(declare-fun y () Int)
(assert (= (+ (head x) 3) y))
(assert (= x (cons (+ y 1) nil)))
(assert (> (head x) (+ y 1)))
(check-sat)
```

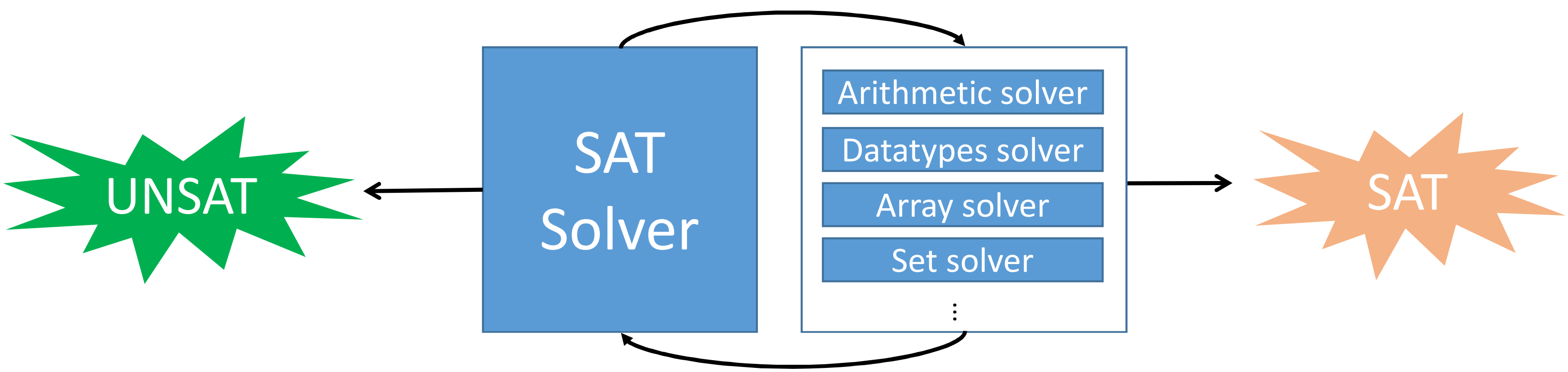
EXAMPLE 3...

DPLL(T) : Theory Combination

- Nelson-Oppen Theory Combination
  - SMT solvers use preexisting theory solvers for combined theories $T_1 + \ldots + T_n$
  - Partition and distribute context M to T_1-solver, …, T_n-solver
    - If any T_i-solver says "unsat", then M is unsatisfiable
    - If each T_i-solver says "sat", then solvers must agree on shared variables
  - Requires theory solvers to:
    - Have disjoint signatures
      - E.g. arithmetic has functions { +, <, 0, 1, … }, datatypes has functions { cons, head, tail, … }
    - Know equalities/disequalities between shared variables
      - E.g. are $u_1 = u_2$ equal?
    - Theories agree on cardinalities for shared types
      - E.g. LIA and DT may agree that Int has infinite cardinality

DPLL(T) : Summary

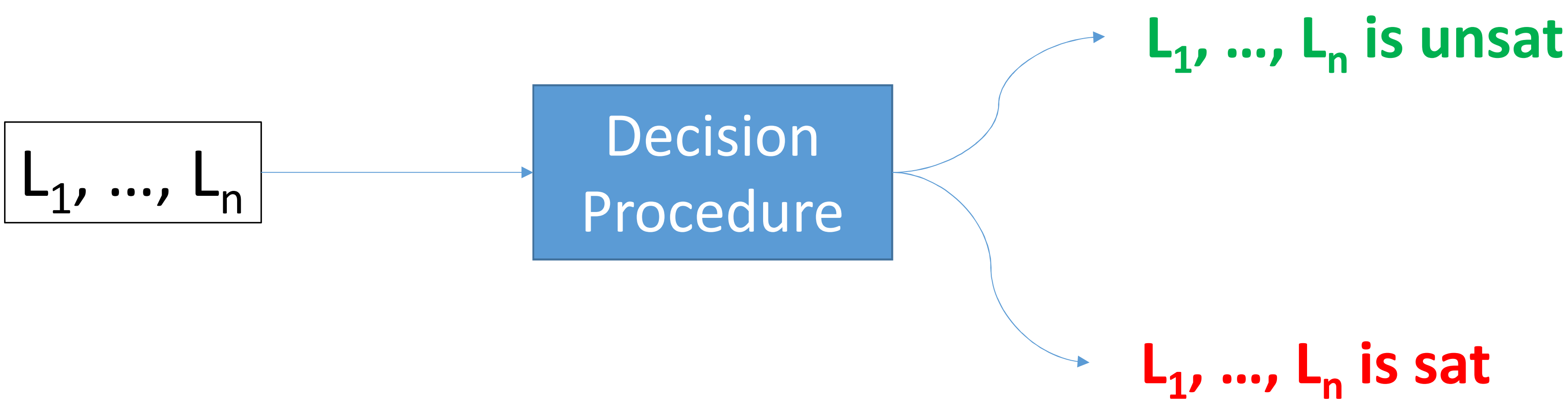
- SMT solvers use
  - DPLL(T) algorithm for theory T, which uses:
    - Off-the-shelf SAT solver
    - Theory solver(s) for T
  - Nelson-Oppen theory combination for combined theories $T_1 + T_2$, which uses:
    - Existing theory solvers for T_1 and T_2, combines them using a generic method



Decision Procedures

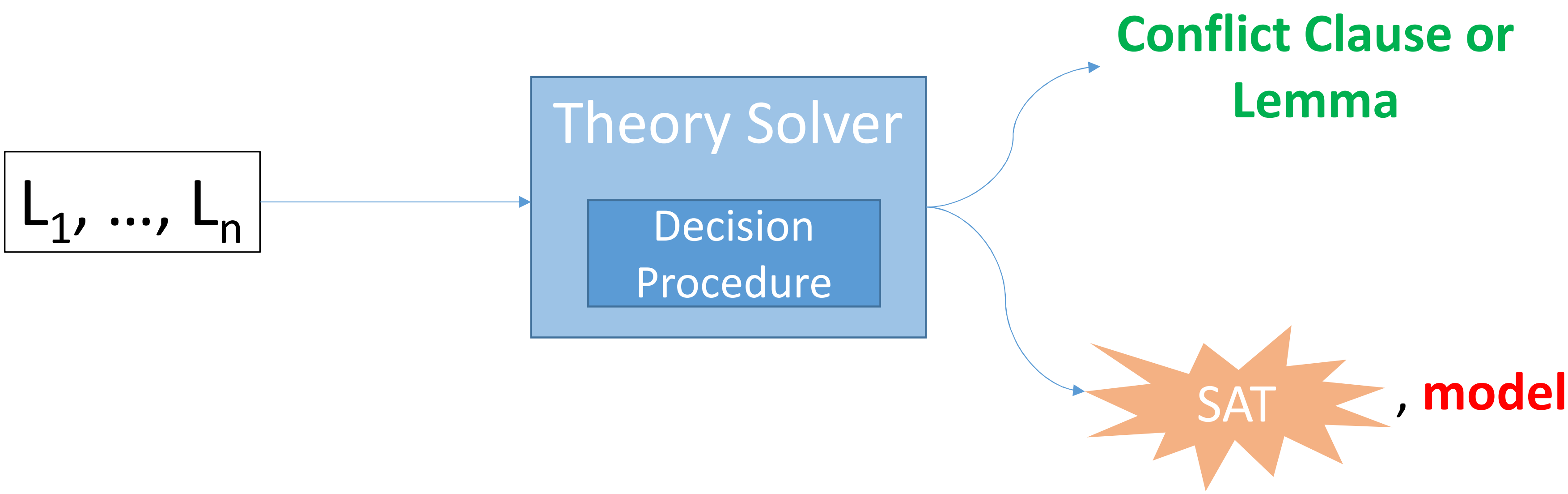
Theory Solvers in DPLL(T)

- SMT solvers use theory solvers that are *decision procedures*:



Theory Solvers in DPLL(T)

- SMT solvers use theory solvers that are *decision procedures*:



Theory Solvers: Linear Arithmetic

Linear Arithmetic

- Quantifier-free linear real and integer arithmetic
  QF_LRA, QF_LIA, QF_LIRA

- Given the linear inequalities

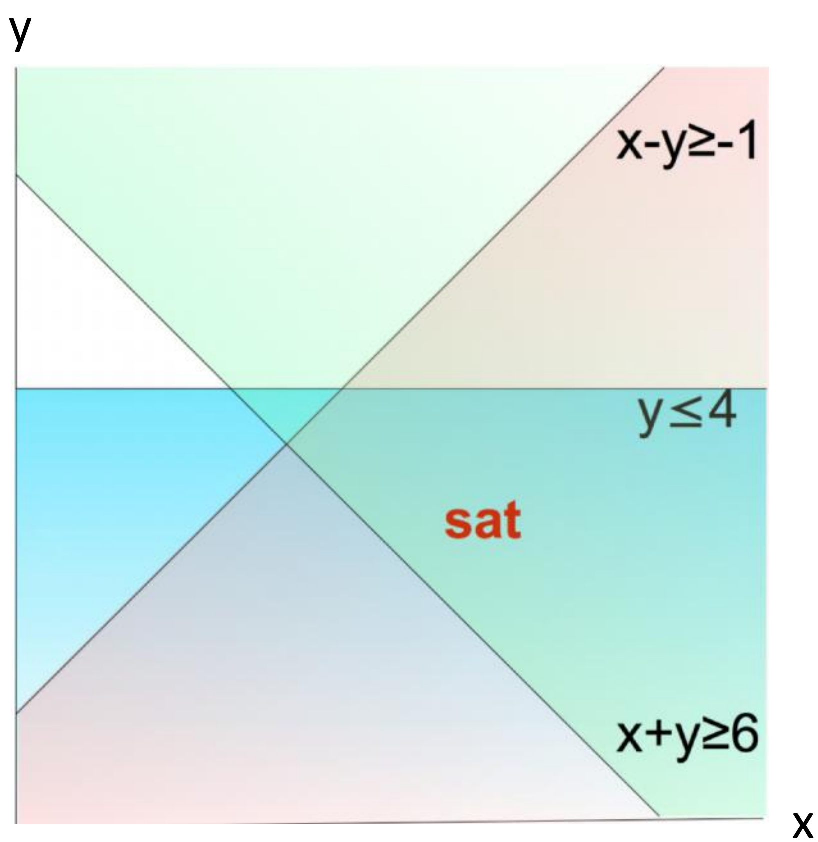
  x : Real; y : Real;

  $x - y \geq -1, \; y \leq 4, \; x + y \geq 6$

  is there an assignment to x and y that makes all of them true?

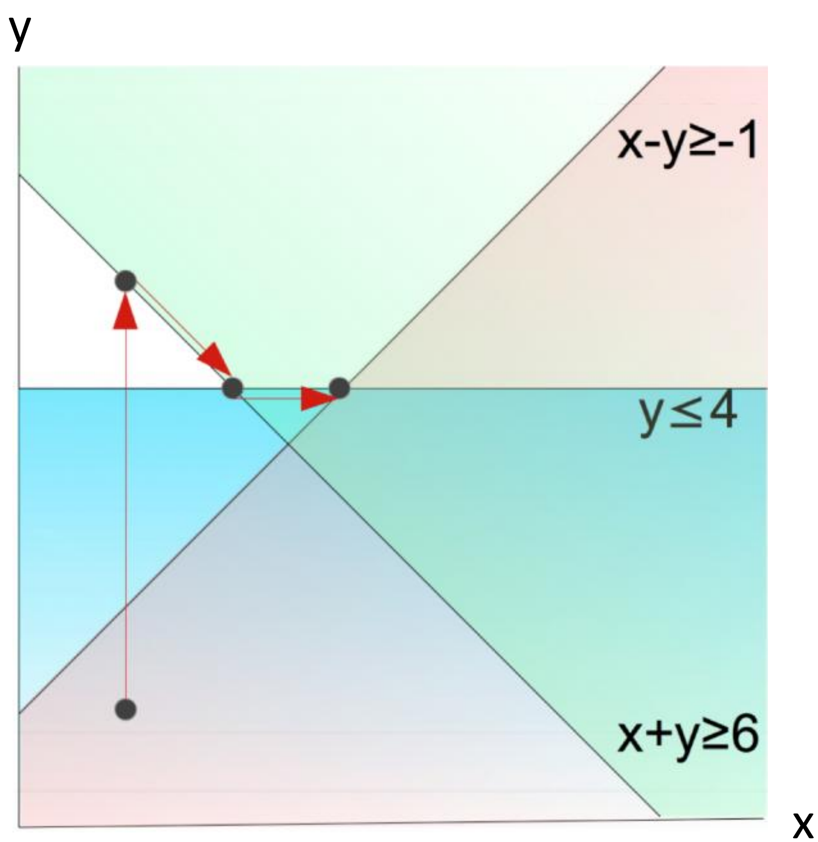
- Solve using simplex-based approaches **[Dutertre/de Moura 2006]**

Simplex Search



Is an intersection of half planes empty?

Simplex Search



From Reals to (Mixed) Integers

- Add `isInt(x)` constraints
- First solve real relaxation
  - Ignore `isInt(x)` constraints
- If real relaxation is sat:
  - Check if current assignment `M(x)` satisfies `isInt(x)` constraints
  - If not, refine by branching **[Dillig 2006, Griggio 2012, Jovanovic/de Moura 2013]**

isInt(x) ∧ M(x)=1	…	SAT
isInt(x) ∧ M(x)=1.5	…	Add lemma (x≥2∨x≤1)
isInt(x) ∧ M(x)=2	…	SAT

Theory Solvers: Equality + Uninterpreted Functions (EUF)

Theory of Uninterpreted Functions (UF)

- Equalities and disequalities between terms built from UF, e.g.

$$a=b,\ f(a,f(b,g(a)))=d,\ g(b)=c,\ f(a,c)=c,\ f(a,c)\neq d$$

…where signature is:

"uninterpreted sort" U

$a,b,c,d : U$

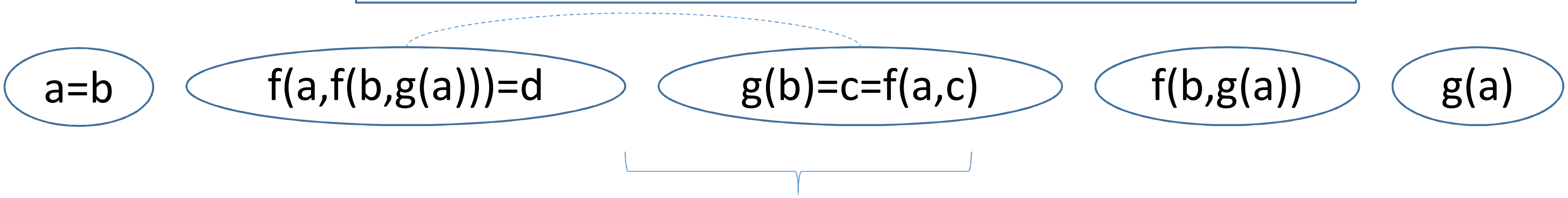
$g : U \to U$

$f : U \times U \to U$

$\Rightarrow$ UF are useful for abstracting processes, other symbols not natively supported by solver

Congruence Closure

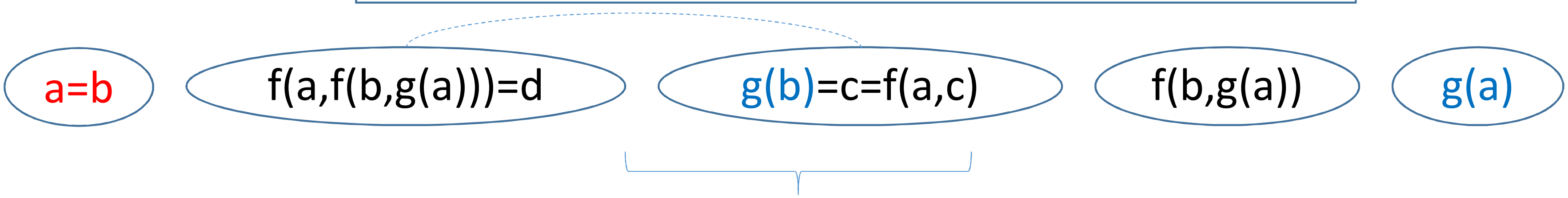
a=b, f(a,f(b,g(a)))=d, g(b)=c, f(a,c)=c, f(a,c)≠d



Compute *equivalence classes* of all (sub)terms

Congruence Closure

a=b, f(a,f(b,g(a)))=d, g(b)=c, f(a,c)=c, f(a,c)≠d

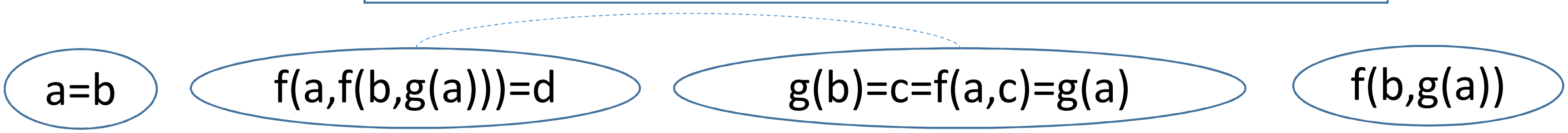


…merge congruent terms

since a=b, we know g(a)=g(b)

Congruence Closure

a=b, f(a,f(b,g(a)))=d, g(b)=c, f(a,c)=c, f(a,c)≠d



Congruence Closure

$a=b,\ f(a,f(b,g(a)))=d,\ g(b)=c,\ f(a,c)=c,\ f(a,c)\neq d$

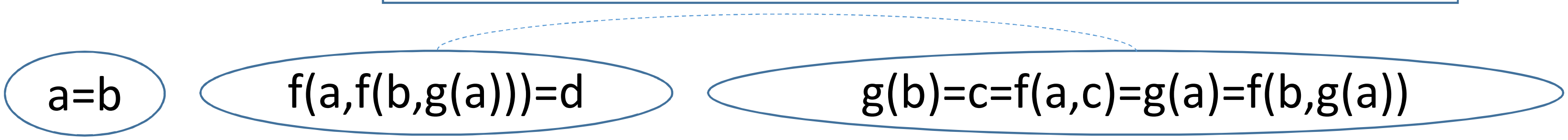
$a=b$ $f(a,f(b,g(a)))=d$ $g(b)=c=f(a,c)=g(a)$ $f(b,g(a))$

…merge congruent terms

since $a=b$ and $c=g(a)$, we know $f(a,c)=f(b,g(a))$

Congruence Closure

a=b, f(a,f(b,g(a)))=d, g(b)=c, f(a,c)=c, f(a,c)≠d



Congruence Closure

a=b, f(a,f(b,g(a)))=d, g(b)=c, f(a,c)=c, f(a,c)≠d

a=b

f(a,f(b,g(a)))=d

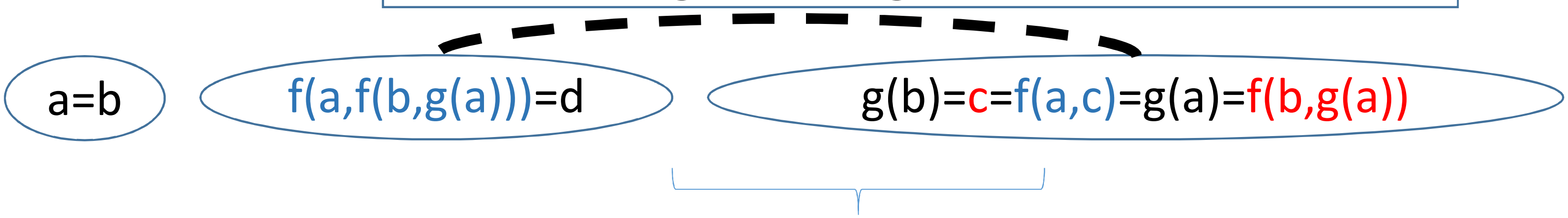
g(b)=c=f(a,c)=g(a)=f(b,g(a))

…merge congruent terms

since f(b,g(a))=c, we know f(a,f(b,g(a)))=f(a,c)

Congruence Closure

a=b, f(a,f(b,g(a)))=d, g(b)=c, f(a,c)=c, f(a,c)≠d



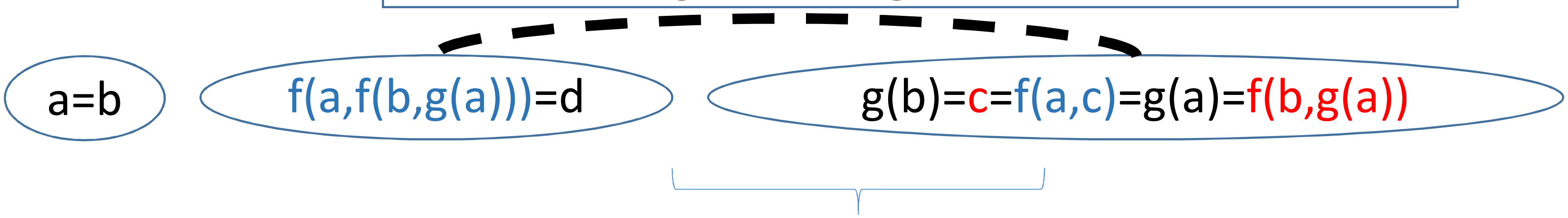
…merge congruent terms

since f(b,g(a))=c, we know f(a,f(b,g(a)))=f(a,c)

⇒ Must merge disequal equivalence classes … conflict!

Congruence Closure

a=b, f(a,f(b,g(a)))=d, g(b)=c, f(a,c)=c, f(a,c)≠d



…merge congruent terms

since f(b,g(a))=c, we know f(a,f(b,g(a)))=f(a,c)

$\Rightarrow$ Must merge disequal equivalence classes … conflict!

Congruence closure is important building block for many decision procedures

Theory Solvers: Arrays

Arrays : Signature Σ

Types:

(Array $T1\ T2$) : Arrays with index type $T1$, element type $T2$

Operators:

(store $a\ i\ v$) : result of writing v at index i in Array a

(select $a\ i$) : result of reading from index i of Array a

(Constants):

(as (const v) (Array $T1\ T2$)) : Array initialized with v at all indices

$\Rightarrow$ Arrays are useful for modelling memory, data structures

Procedure for Arrays with Extensionality

- Procedure uses McCarthy axioms:

$i \neq j \Rightarrow$ (select (store A i k) j) = (select A j)
$i = j \Rightarrow$ (select (store A i k) j) = k

(read over write)

$A \neq B \Rightarrow$ (select A k) $\neq$ (select B k) for some k

(extensionality)

- Builds on top of congruence closure
- Instantiates array axioms lazily
- Design/implementation uses efficient ways to instantiate

  [Goel/Krstic/Fuchs 2008, de Moura/Bjorner 2009]

Procedure for Arrays with Extensionality

i=j, select(A,i)≠select(store(A,k,5),j),j≠k

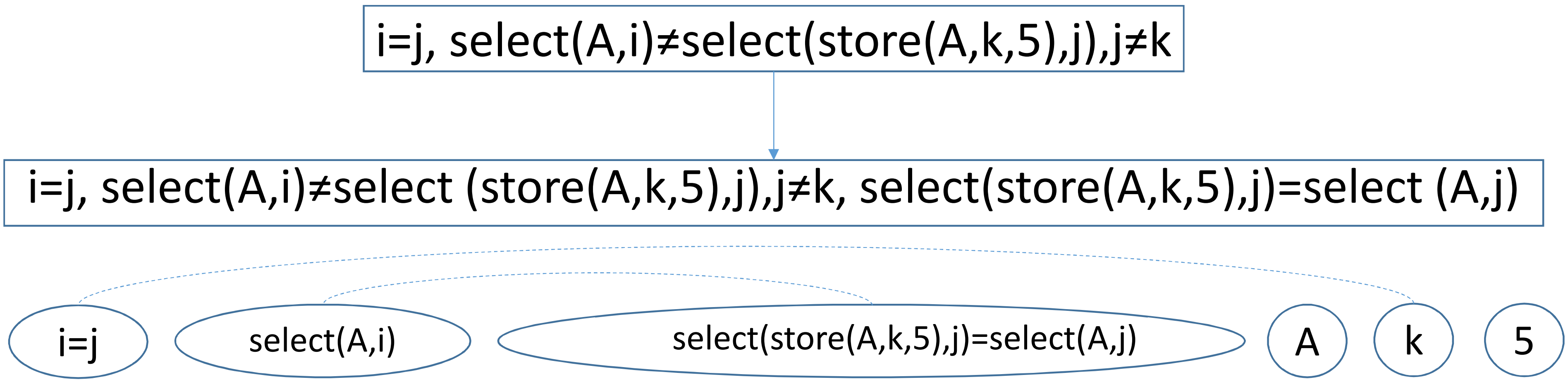
Procedure for Arrays with Extensionality

i=j, select(A,i)≠select(store(A,k,5),j),j≠k

(read over write)

i=j, select(A,i)≠select (store(A,k,5),j),j≠k, select(store(A,k,5),j)=select (A,j)

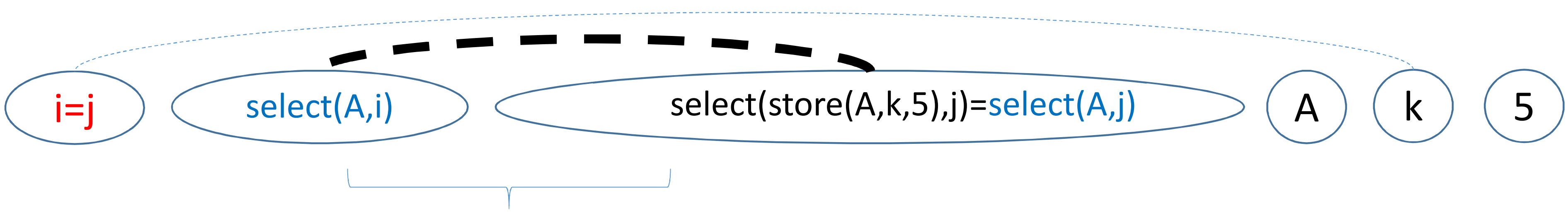
Procedure for Arrays with Extensionality



Procedure for Arrays with Extensionality

i=j, select(A,i)≠select(store(A,k,5),j),j≠k

i=j, select(A,i)≠select (store(A,k,5),j),j≠k, select(store(A,k,5),j)=select (A,j)



…merge congruent terms

since i=j, we know select(A,i)=select(A,j)

conflict!

Theory Solvers: Bitvectors

Bit Vectors

- Bit-vectors parameterized by a bit-width

  (_ BitVec 2) : #b00,#b01,#b10,#b11

  (_ BitVec 10) : #b0000000000,#b0101010101,#b1111111110,…

  …

- For each bit-width, large signature containing operators for:
  - (Modular) arithmetic
  - Bitwise logical operations
  - Bit-shifting
  - Concatenation/extraction

$\Rightarrow$ Bit-vectors are useful for modelling machine integers, circuits

SMT-LIB Bitvectors

(_ BitVec n)

(_ #bv$X$ n)

#bX

#xX

(bvshl s t)

(bvlshr s t)

(bvashr s t)

(concat s t)

((_ extract i j) s)

((_ repeat i) s)

((_ zero_extend i) s)

((_ sign_extend i) s)

((_ rotate_left i) s)

((_ rotate_right i) s)

(bvnot s)

(bvand s t)

(bvnand s t)

(bvor s t)

(bvnor s t)

(bvxor s t)

(bvxnor s t)

SMT-LIB Bitvectors

(bvneg s)
(bvadd s t)
(bvsub s t)
(bvmul s t)

(bvudiv s t)
(bvurem s t)
(bvsdiv s t)
(bvsrem s t)
(bvsmod s t)

(bvcomp s t)
(bvult s t)
(bvule s t)
(bvugt s t)
(bvuge s t)
(bvslt s t)
(bvsle s t)
(bvsgt s t)
(bvsge s t)

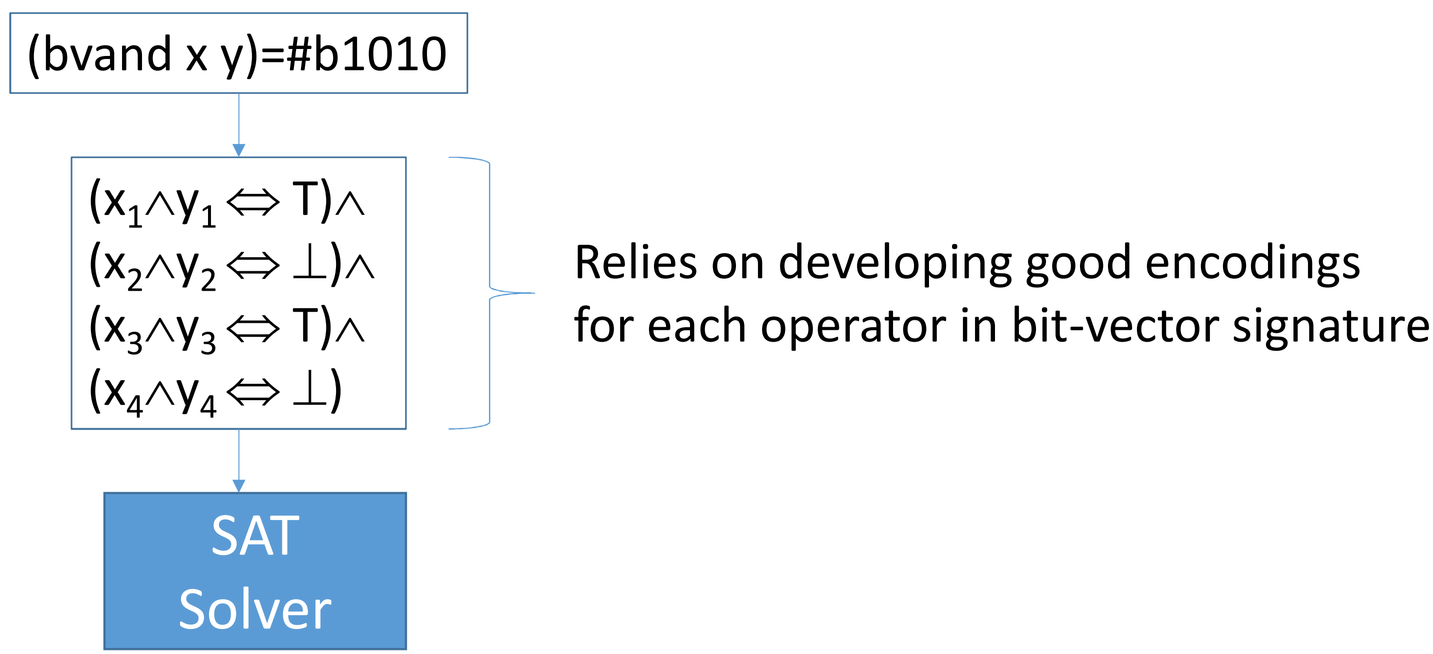
Solving Bit Vectors

- Typically, bit-vector constraints are solved by bit-blasting
  - ⇒Eager reduction to propositional satisfiability
- For example:

(bvand x y)=#b1010

Solving Bit Vectors

- For example:



Solving Bit Vectors

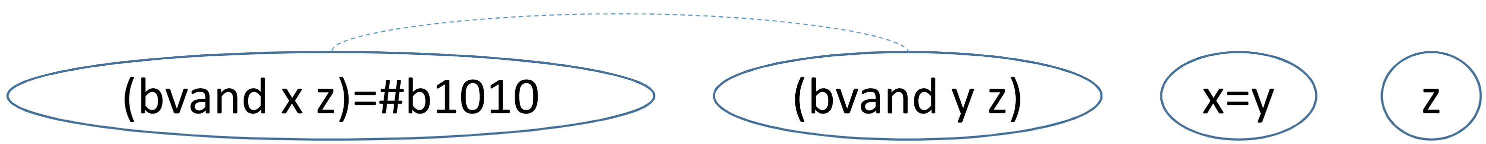
- Bit-blasting can also be done lazily **[Bruttomesso et al 2007,Hadarean et al 2014]**
- Instead of:

(bvand x z)=#b1010, (bvand y z)≠#b1010, x=y

$$
\begin{aligned}
&(x_1 \wedge z_1 \Leftrightarrow T) \wedge (y_1 \wedge z_1 \Leftrightarrow T) \wedge (x_1 \Leftrightarrow y_1) \wedge \\
&(x_2 \wedge z_2 \Leftrightarrow \bot) \wedge (y_2 \wedge z_2 \Leftrightarrow \bot) \wedge (x_2 \Leftrightarrow y_2) \wedge \\
&(x_3 \wedge z_3 \Leftrightarrow T) \wedge (y_3 \wedge z_3 \Leftrightarrow \bot) \wedge (x_3 \Leftrightarrow y_3) \wedge \\
&(x_4 \wedge z_4 \Leftrightarrow \bot) \wedge (y_4 \wedge z_4 \Leftrightarrow T) \wedge (x_4 \Leftrightarrow y_4)
\end{aligned}
$$

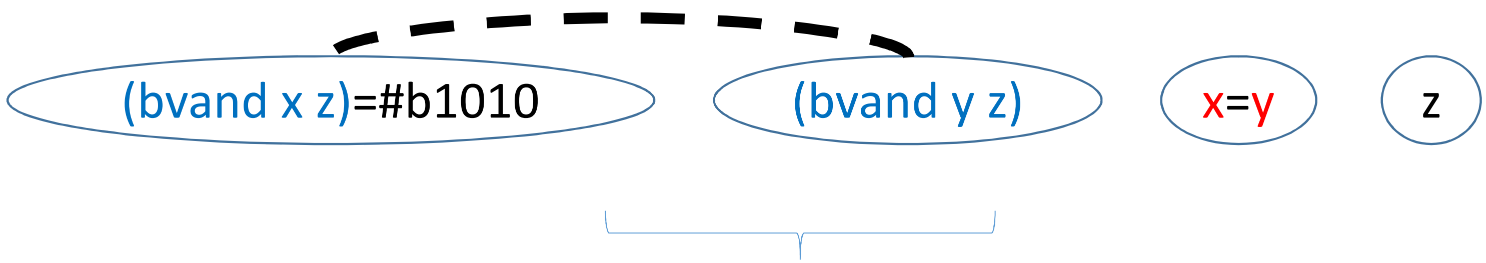
Solving Bit Vectors

(bvand x z)=#b1010, (bvand y z)≠#b1010, x=y



Solving Bit Vectors

(bvand x z)=#b1010, (bvand y z)≠#b1010, x=y



…merge congruent terms

since x=y, we know (bvand x z)=(bvand y z)

⇒ ***Conflict***, *before resorting to bit-blasting*

Theory Solvers: Finite Sets + Cardinality

Theory of Finite Sets + Cardinality

- Parametric theory of finite sets of elements E
- Signature Σ_{Set}:
  - Empty set $\emptyset$, Singleton `{a}`
  - Membership $\in$: E x `Set` $\rightarrow$ `Bool`
  - Subset $\subseteq$: `Set` x `Set` $\rightarrow$ `Bool`
  - Set connectives $\cup$, $\cap$, $\setminus$: `Set` x `Set` $\rightarrow$ `Set`

- Example input: $x = y \cap z \wedge a+5 \in x \wedge y \subseteq w$

$\Rightarrow$Sets are important in databases, knowledge representation, programming languages, e.g. Alloy

Theory of Finite Sets + Cardinality

- Extended signature of theory to include:
  - Cardinality $|.| : \texttt{Set} \rightarrow \texttt{Int}$
- Extended decision procedure for cardinality constraints [Bansal et al IJCAR2016]

- Example input: $x = y \cup z \wedge |x| = 14 \wedge |y| \geq |z| + 5$

Theory of Finite Sets + Cardinality

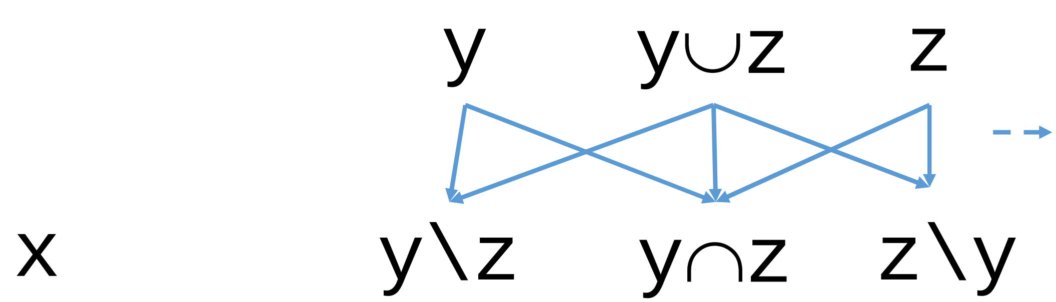
- Decision procedure builds cardinality graph where
  - Cardinality of leaves are disjoint sum of parents

$x = y \cup z$
$|x| = 14$
$|y| \geq |z| + 5$

[Bansal/Reynolds/Barrett/Tinelli IJCAR2016]

Theory of Finite Sets + Cardinality

- Decision procedure builds cardinality graph where
  - Cardinality of leaves are disjoint sum of parents



$x = y \cup z$

$|x| = 14$

$|y| \geq |z| + 5$

$|y| = |y \setminus z| + |y \cap z|$

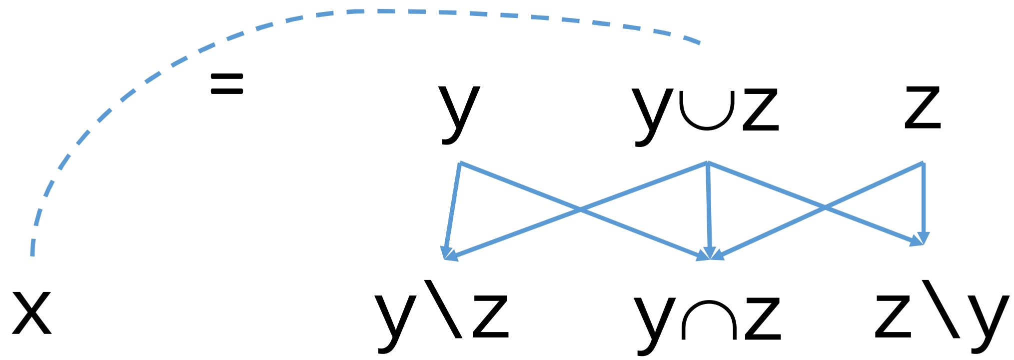
$|z| = |z \setminus y| + |y \cap z|$

$|y \cup z| = |y \setminus z| + |y \cap z| + |z \setminus y|$

[Bansal/Reynolds/Barrett/Tinelli IJCAR2016]

Theory of Finite Sets + Cardinality

- Decision procedure builds cardinality graph where
  - Cardinality of leaves are disjoint sum of parents
    - Equalities between sets



$x = y \cup z$

$|x| = 14$

$|y| \geq |z| + 5$

$|y| = |y \setminus z| + |y \cap z|$

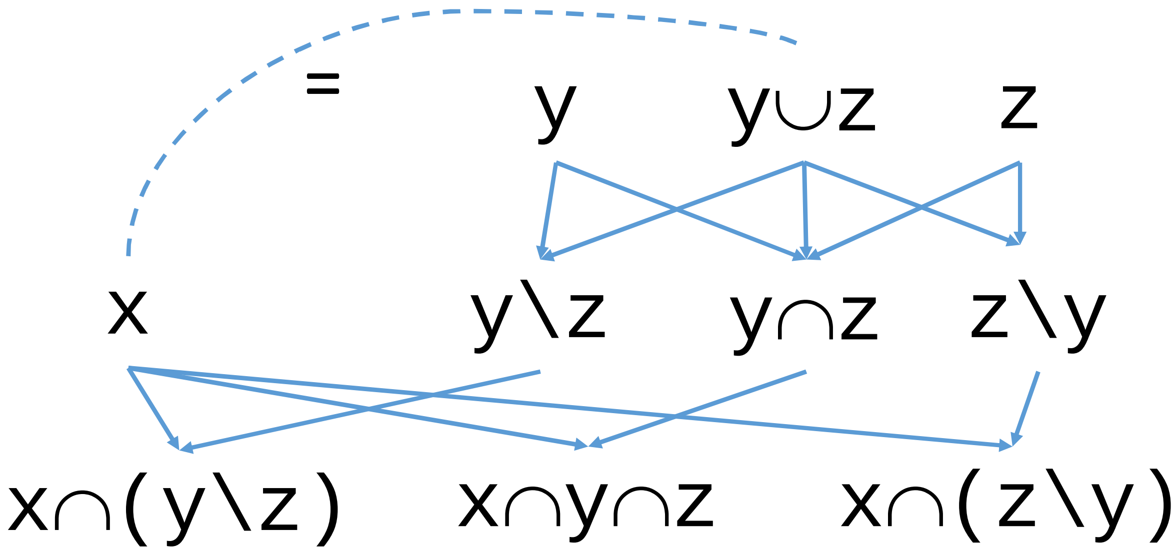
$|z| = |z \setminus y| + |y \cap z|$

$|y \cup z| = |y \setminus z| + |y \cap z| + |z \setminus y|$

[Bansal/Reynolds/Barrett/Tinelli IJCAR2016]

Theory of Finite Sets + Cardinality

- Decision procedure builds cardinality graph where
  - Cardinality of leaves are disjoint sum of parents
    - Equalities between sets → merge leaves



$$x = y \cup z$$
$$|x| = 14$$
$$|y| \geq |z| + 5$$
$$|y| = |y \setminus z| + |y \cap z|$$
$$|z| = |z \setminus y| + |y \cap z|$$
$$|y \cup z| = |y \setminus z| + |y \cap z| + |z \setminus y|$$
$$x = y \cup z \Rightarrow$$
$$|x| = |x \cap (y \setminus z)| +$$
$$|x \cap y \cap z| + |x \cap (z \setminus y)|$$

[Bansal/Reynolds/Barrett/Tinelli IJCAR2016]

Theory Solvers: Strings

Basic String Constraints

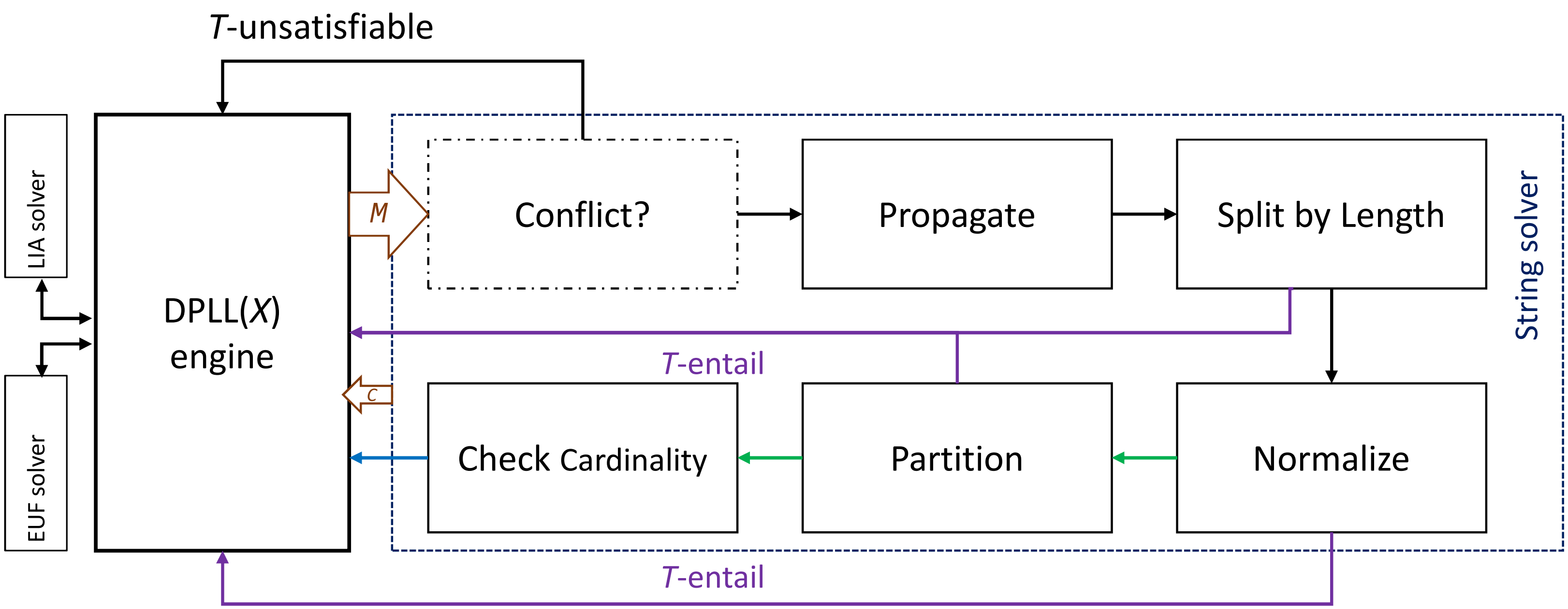
- Equalities and disequalities between:
  - *Basic string* terms
    - String constants: $\in$, "abc"
    - Concatenation: $x \cdot$ "abc"
    - Length: $|x|$
  - *Linear arithmetic* terms: $x+4$, $y>2$

Example: $x \cdot \text{"a"} = y \wedge |y| > |x| + 2$

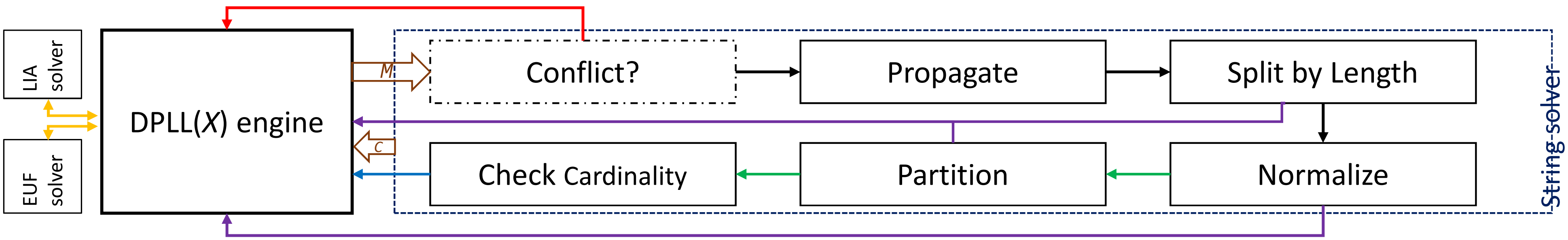
Procedures in **[Abdulla et al CAV2014, Liang et al CAV2014]**

$\Rightarrow$Strings are important in security applications, e.g. for detecting attack vulnerabilities

General String Solver Architecture



DPLL(*T*): Find Satisfying Assignment



|x|>|y|

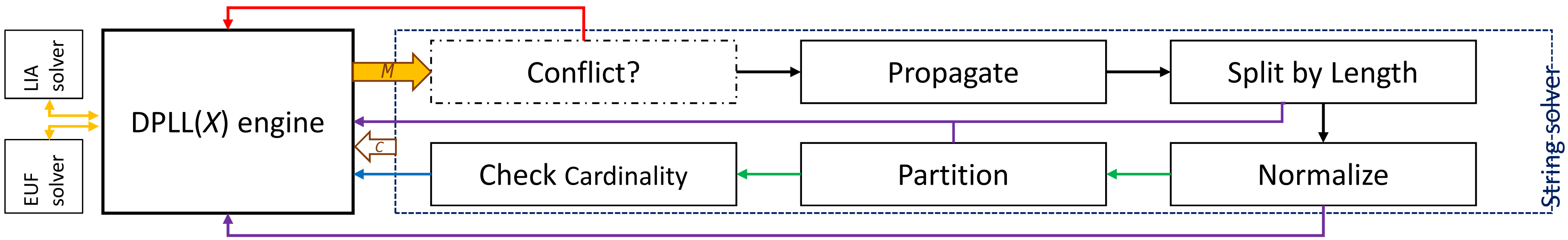
(x·z=y·w·“ab” ∨ x=y)

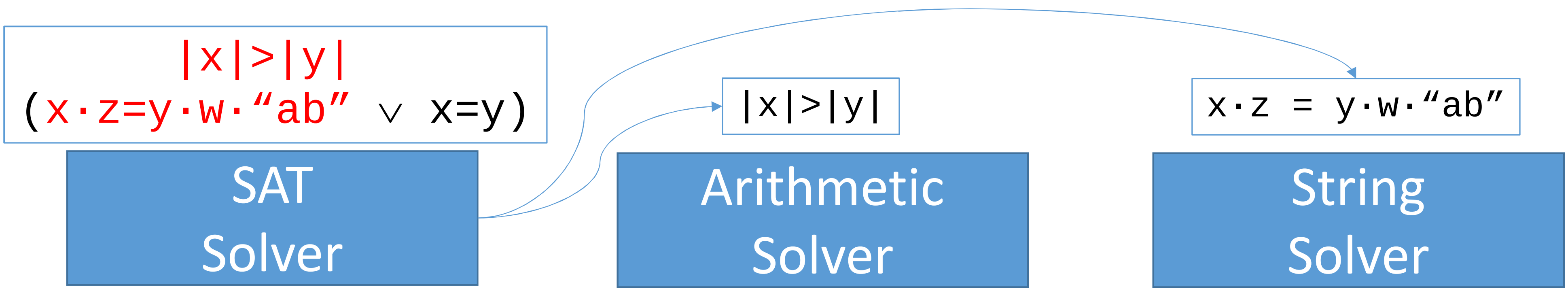
SAT Solver

Arithmetic Solver

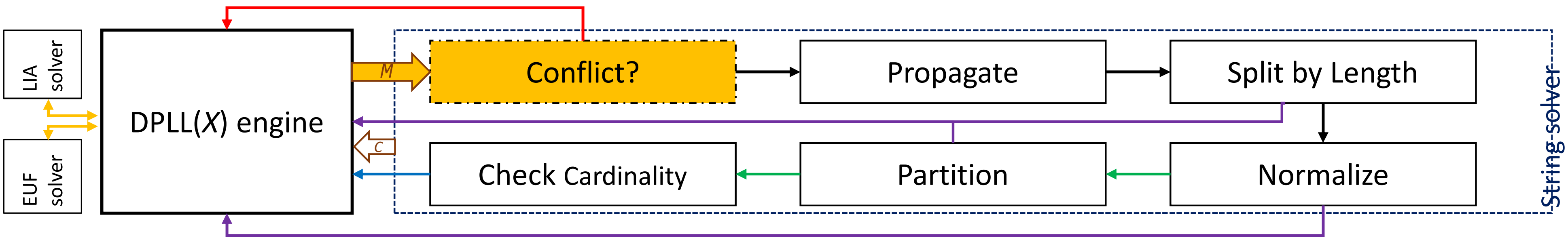
String Solver

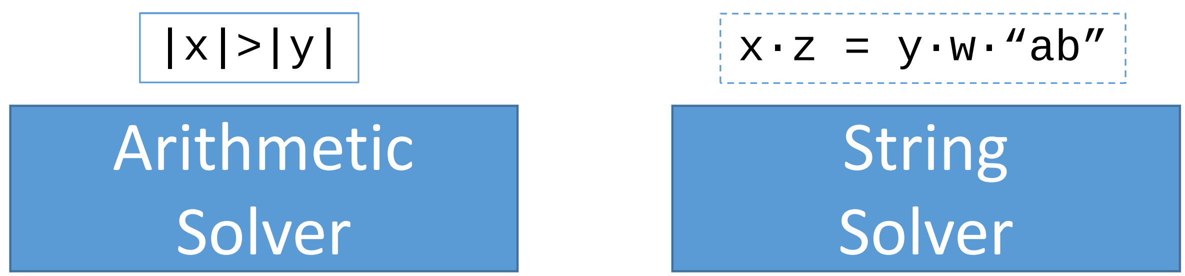
DPLL(*T*): Find Satisfying Assignment





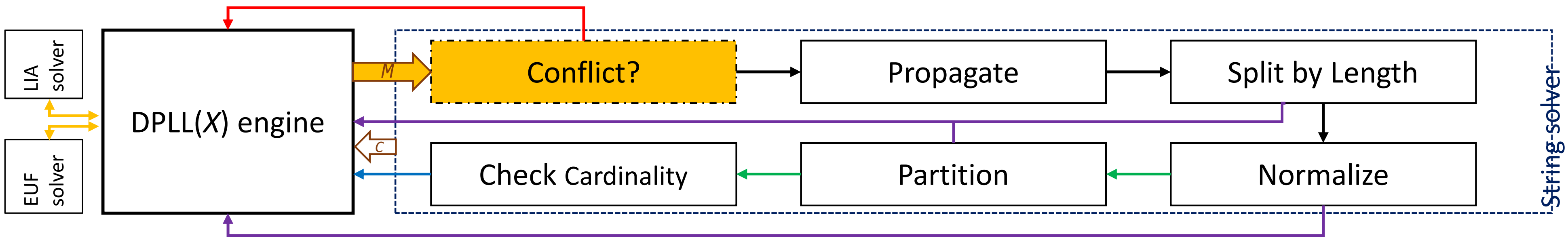
Conflict Checking





To check whether context contains dis-equalities like: s≠s

Conflict Checking



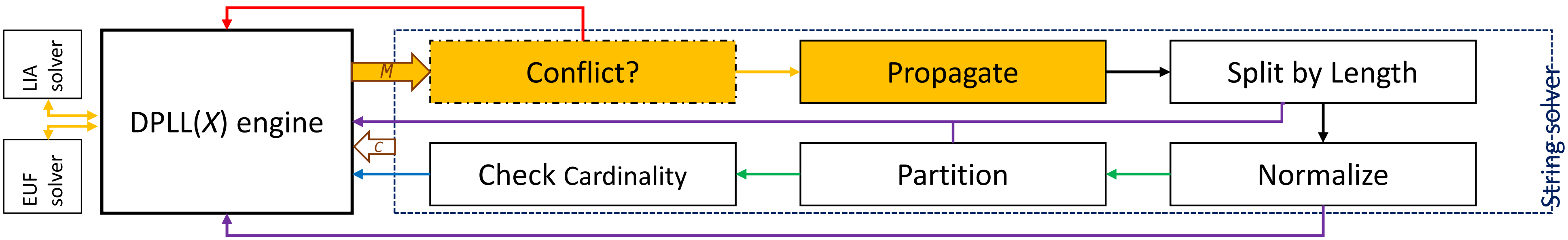
|x| > |y|

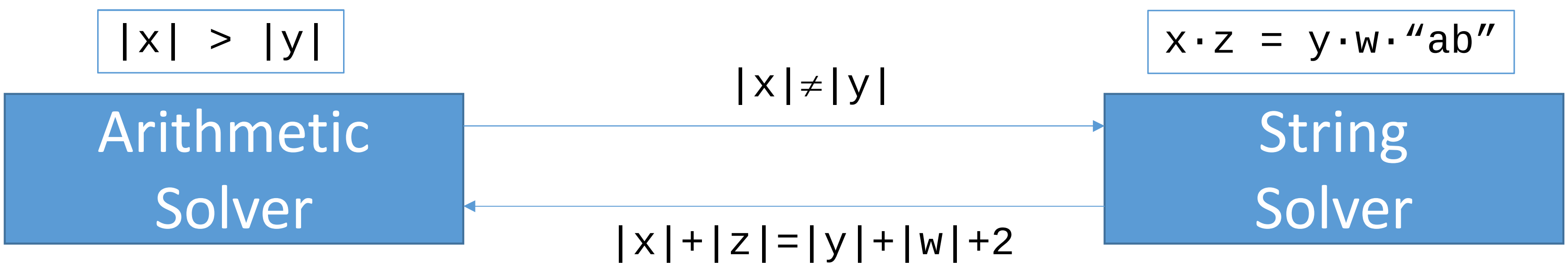
Arithmetic Solver

x·z = y·w·"ab"

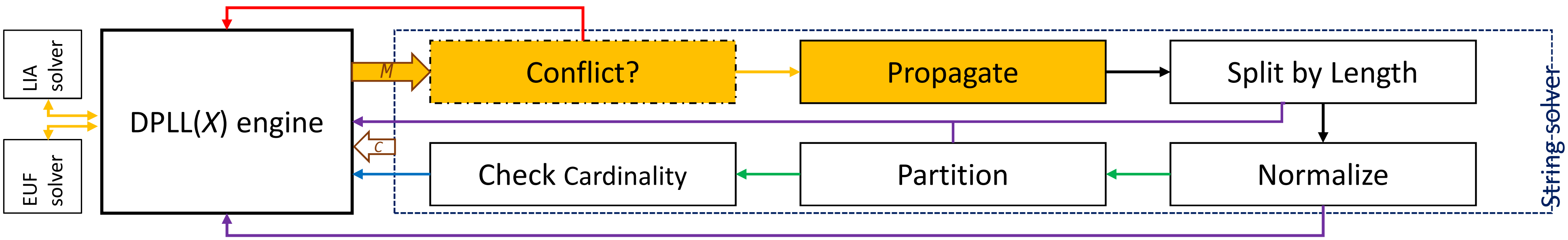
String Solver

Shared Term Propagation





Shared Term Propagation



$|x|+|z|=|y|+|w|+2$

$|x| > |y|$

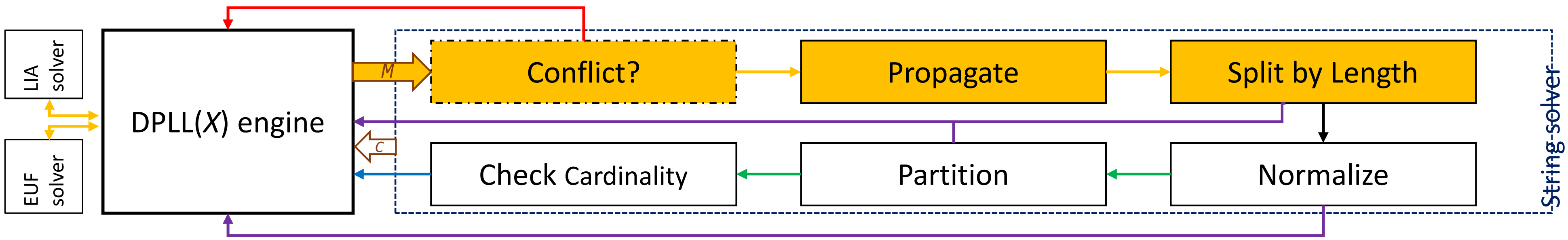
Arithmetic Solver

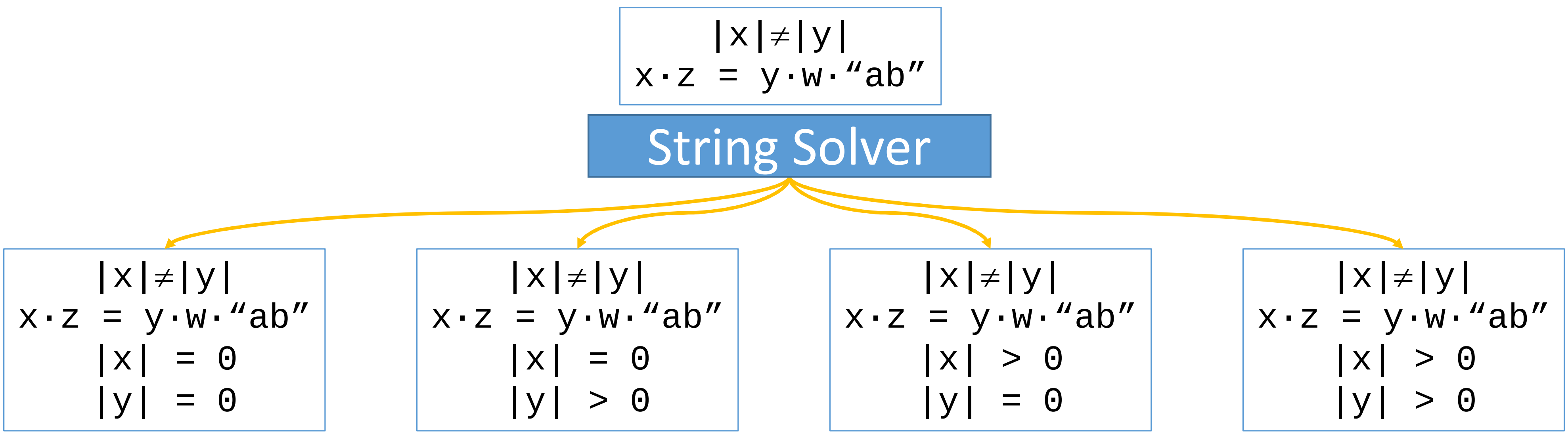
$|x| \neq |y|$

$x \cdot z = y \cdot w \cdot \text{“ab”}$

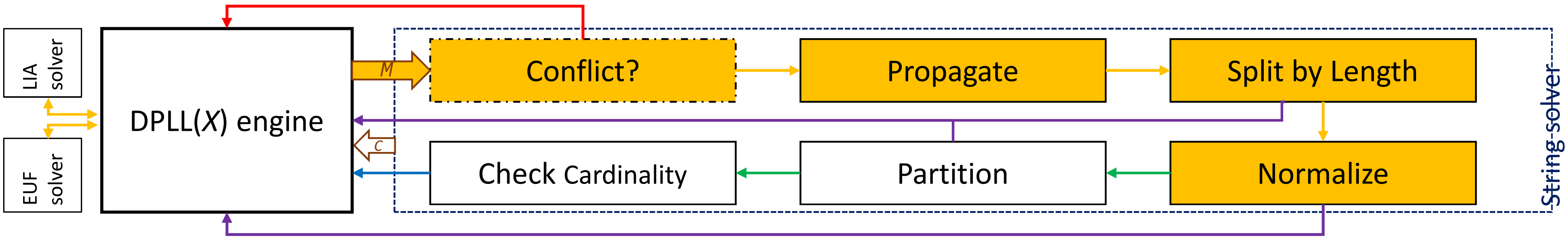
String Solver

Length Splitting



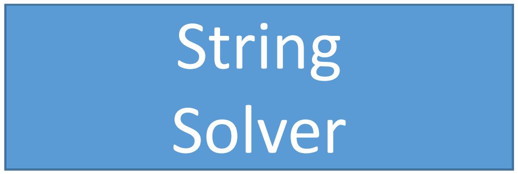


Normalization



E

```
|x|≠|y|
x·z = y·w·"ab"
```

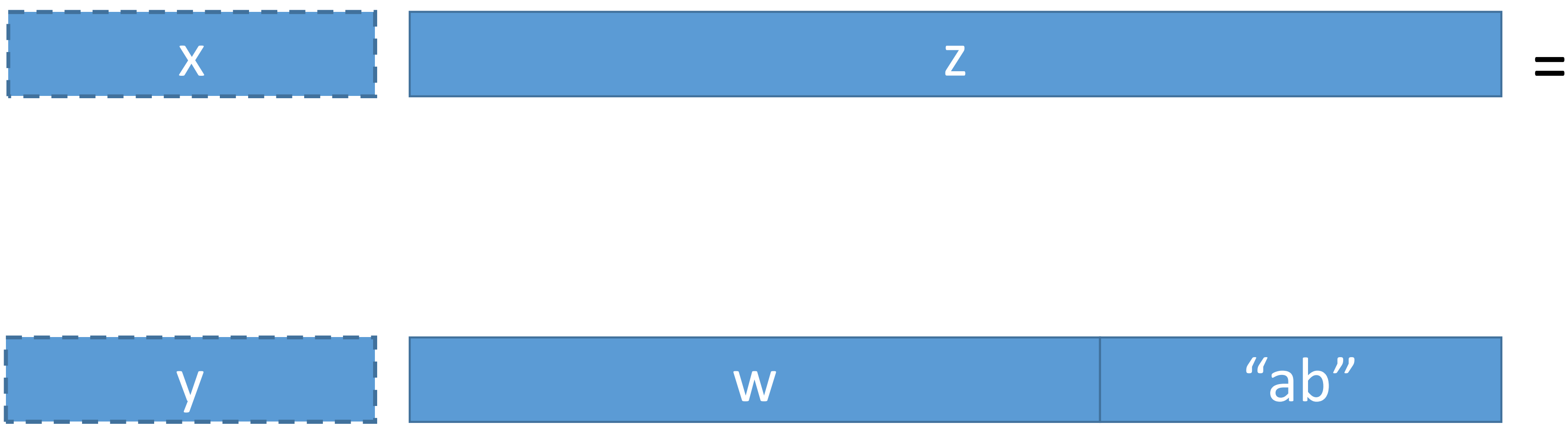


To check satisfiability of equalities in **E**

- Add additional equalities to **E**
- Until pairs of equiv. terms have same normal form

Normalize Equalities

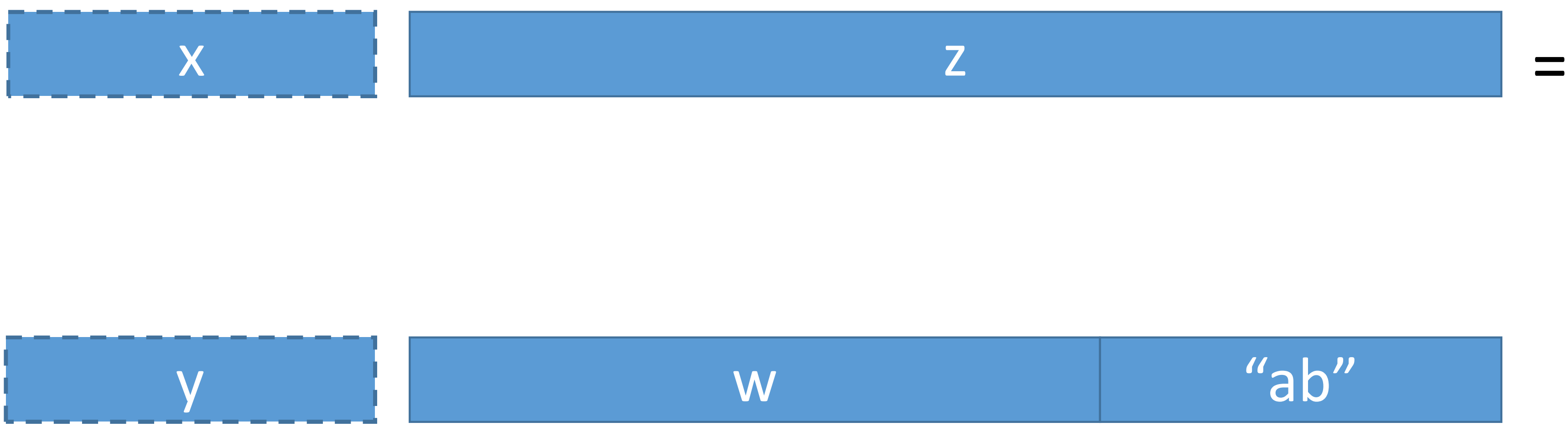
E: |x|≠|y|, x·z=y·w·"ab"



Normalize Equalities

S { x·z=y·w·“ab”

L { |x|≠|y|



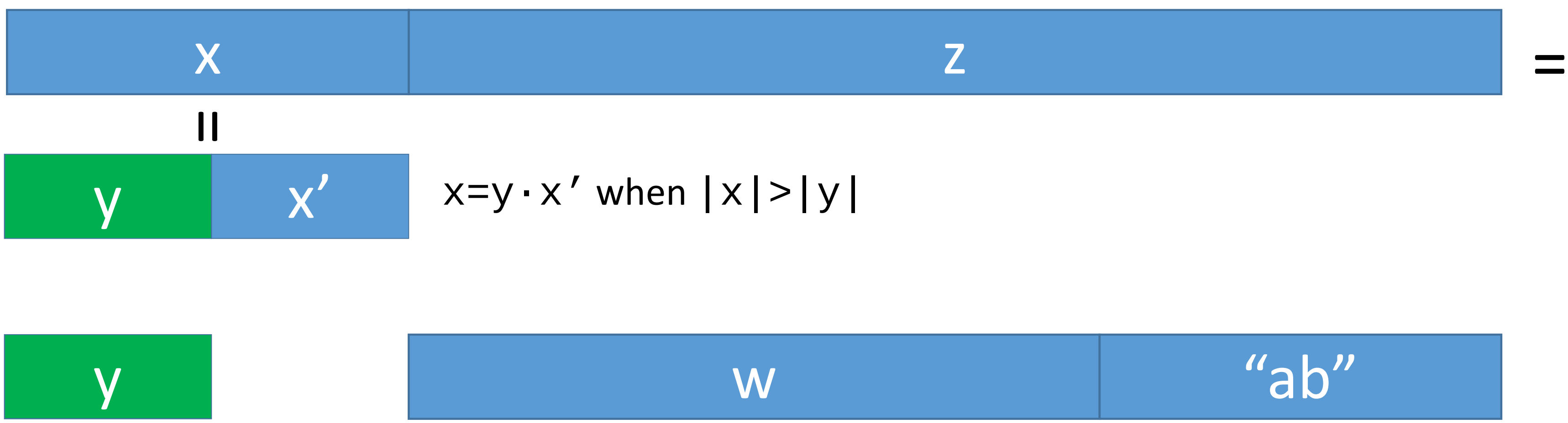
Normalize Equalities

S

x·z=y·w·"ab"
x=y·x'

L

|x|>|y|

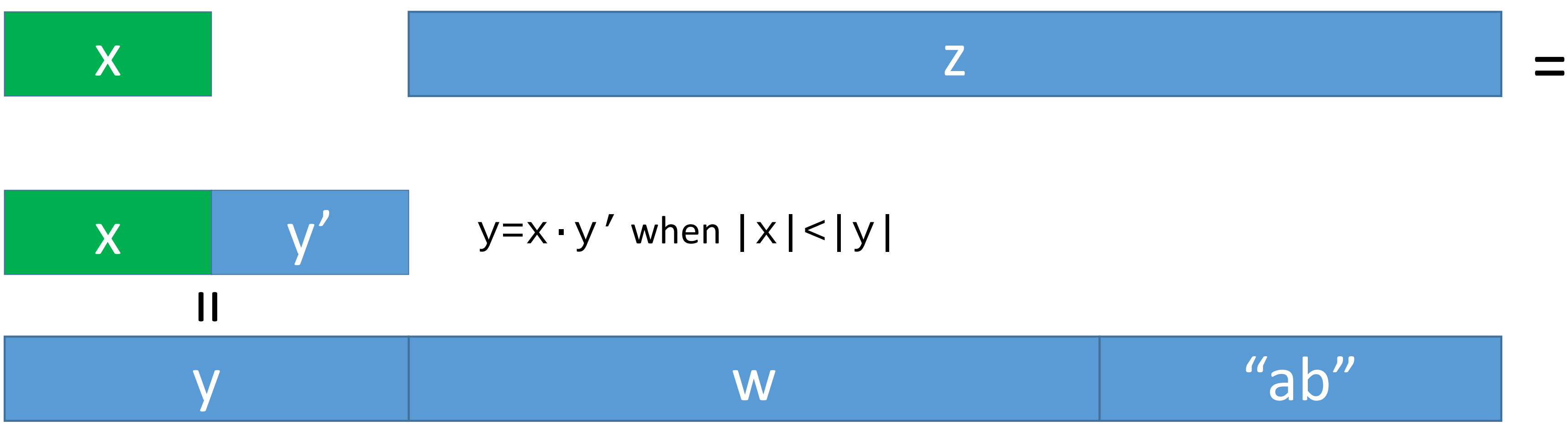


x=y·x' when |x|>|y|

Normalize Equalities

S: x·z=y·w·“ab”
y=x·y’

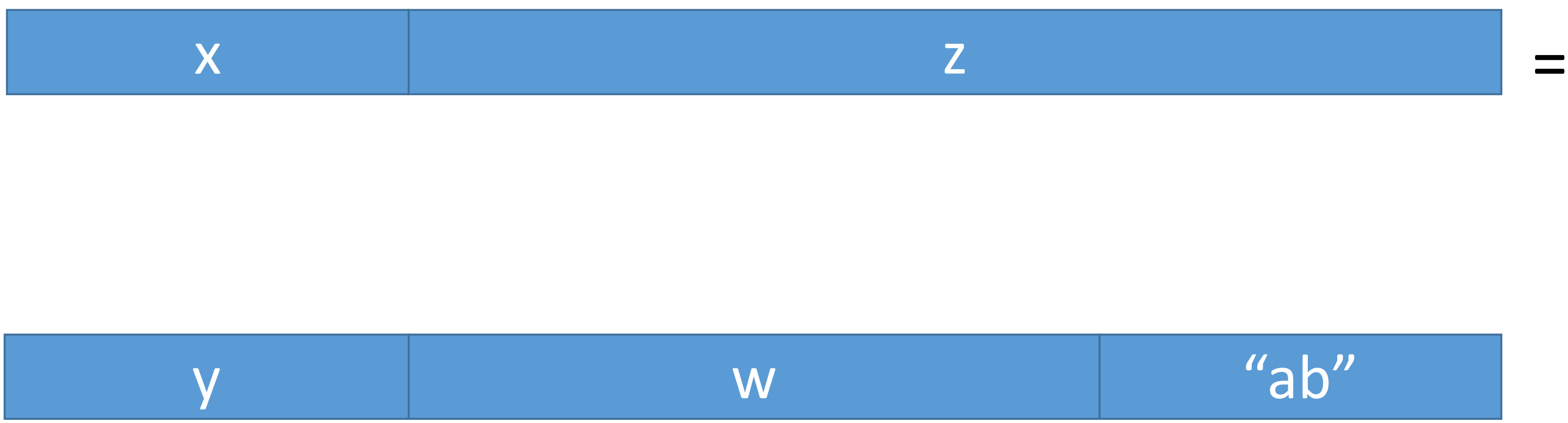
L: |x|<|y|



Normalize Equalities

S { x·z=y·w·“ab”

L { $|x|=|y|$
$|z|\neq|w|$



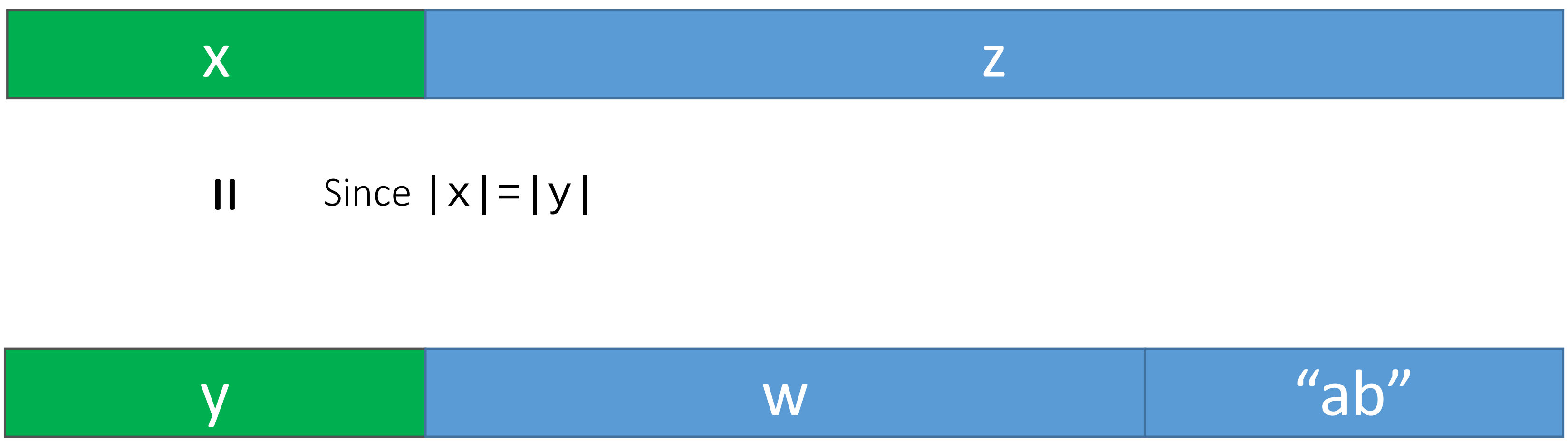
Normalize Equalities

S

x·z=y·w·“ab”
x=y

L

|x|=|y|
|z|≠|w|



Normalize Equalities

S

x·z=y·w·"ab"
x=y
z=w·z'

L

|x|=|y|
|z|≠|w|

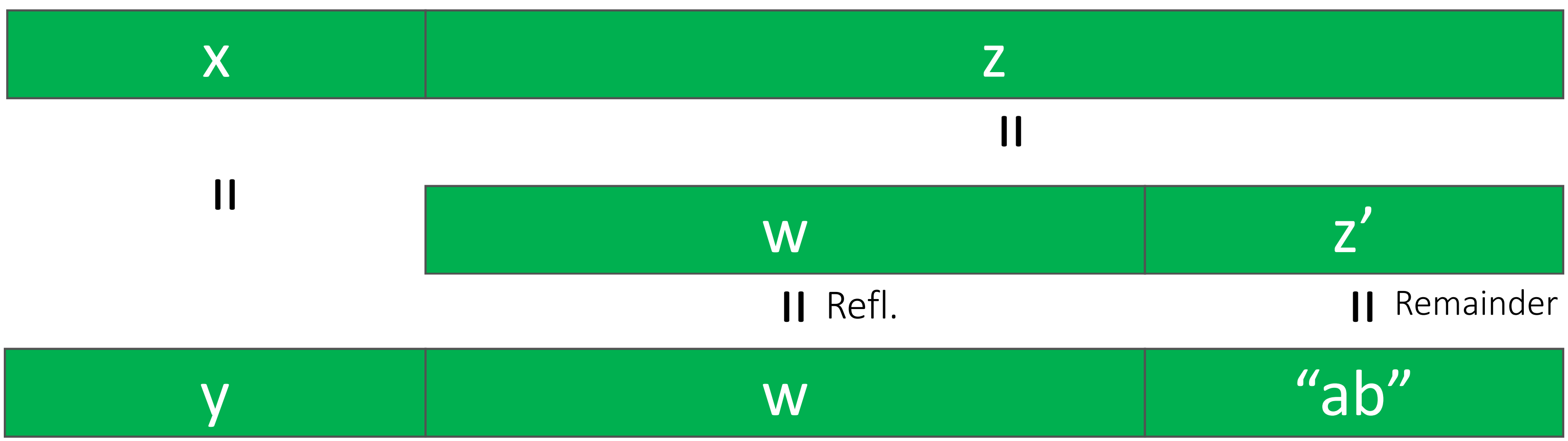
Normalize Equalities

S

```
x·z=y·w·“ab”
x=y
z=w·z’
z’=“ab”
```

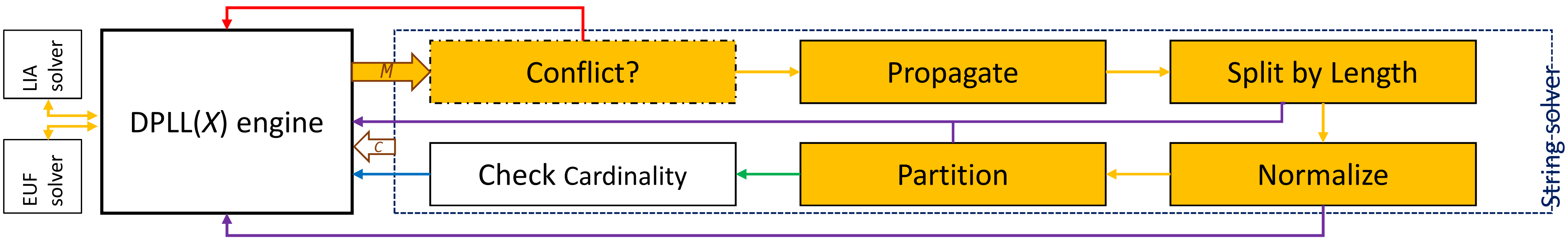
L

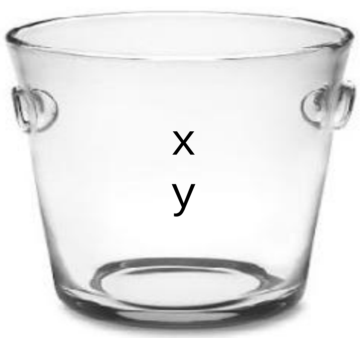
```
|x|=|y|
|z|≠|w|
```



$\Rightarrow S \cup L$ is satisfiable

Partition

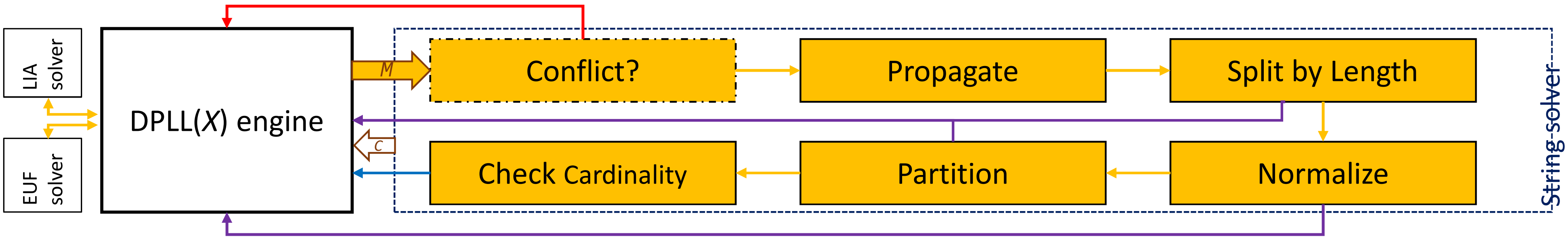




$|x| = |y|$
$x \neq y$

$|x| \neq |z| \neq |w|$

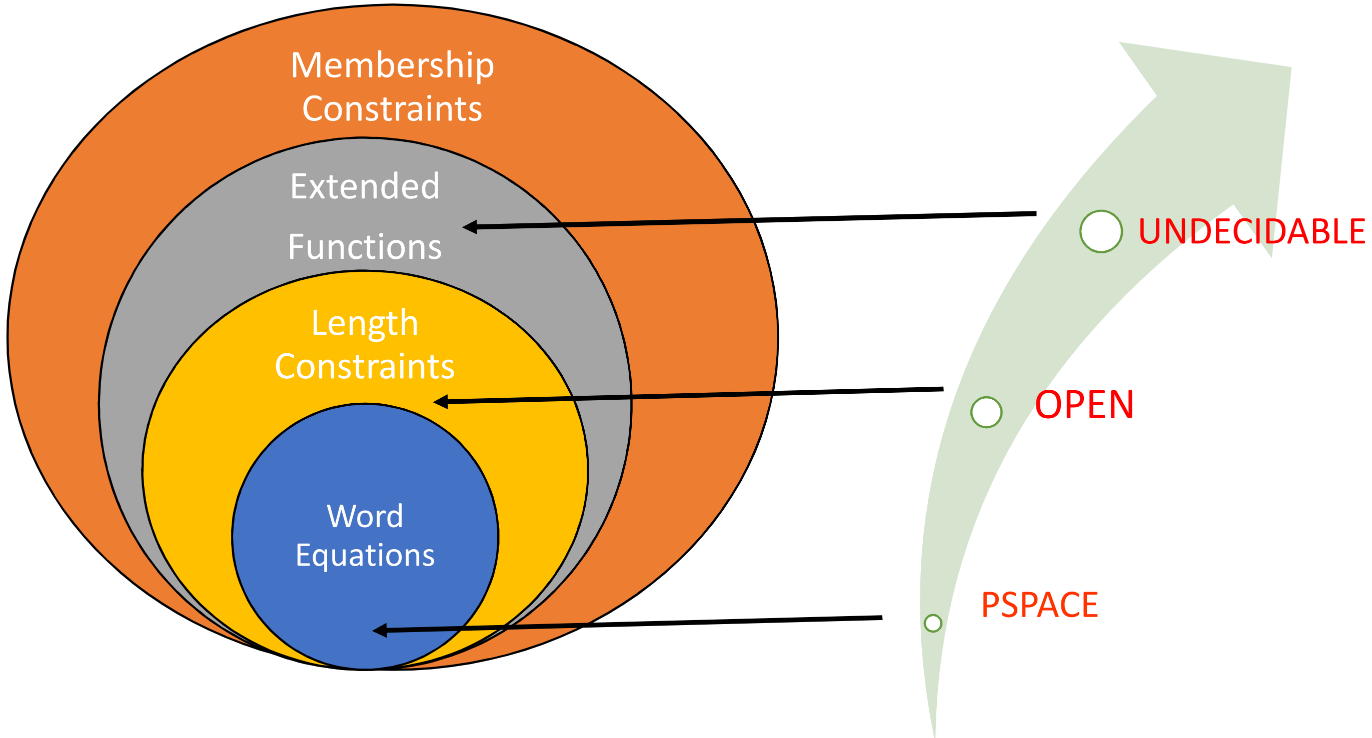
Check Cardinality of Σ



- S may be unsatisfiable since Σ is finite
- For instance, if:
  - Σ is a finite alphabet of 256 characters, and
  - S entails that 257 distinct strings of length 1 exist
- Then:
  - S is unsatisfiable

$|s_1|=1, \ldots, |s_{257}|=1, \ s_1 \neq \ldots \neq s_{257}$

Theoretical Complexity Challenges



Properties of DPLL(T) String Solvers

- For *basic* constraints, DPLL(T) string solvers:
  - Can be used for "**sat**" and "**unsat**" answers
  - Are **incomplete** and/or **non-terminating** in general

- Expected, since *decidability is unknown*
  [Ganesh et al 2011]

- Regardless, modern solvers are *efficient in practice*
  [Zheng et al 2013,Liang et al 2014,Abdulla et al 2015,Trinh et al 2016]

Extended String Constraints

- Equalities and disequalities between:
  - *Basic string* terms:

    Examples
    - String constants — `∈, “abc”`
    - String concatenation — `x·“abc”`
    - String length — `|x|`
  - *Linear arithmetic* terms — `x+4,y>2`
  - *Extended* string terms:
    - Substring — `substr(“abcde”,1,3)` = `“bcd”`
    - String Contains — `contains(“abcde”,”cd”)` = `T`
    - String find “index of” — `indexof(“abcde”,”d”,0)` = `3`
    - String Replace — `replace(“ab”,“b”,“c”)` = `“ac”`

Example: Ėcontains(substr(x,0,3),“a”) ∧ 0≤indexof(x,“ab”,0)<4

How do we handle Extended String Constraints?

```
¬contains(x,“a”)
```

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded $\forall$

```
¬contains(x,"a")
```

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀

$\neg$contains(x,“a”)

↓

∀0≤n<|x|.substr(x,n,1)≠“a”

Expand contains

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀

`¬contains(x,“a”)`

↓

`∀0≤n<|x|.substr(x,n,1)≠“a”` — Expand `contains`

↓

`substr(x,0,1)≠“a” ∧ … ∧ substr(x,4,1)≠“a”` — Assuming bound |x|≤5

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded $\forall$

$\neg$`contains(x,"a")`

↓

$\forall$`0≤n<|x|.substr(x,n,1)≠"a"` — Expand `contains`

↓

`substr(x,0,1)≠"a"` $\wedge \ldots \wedge$ `substr(x,4,1)≠"a"` — Assuming bound `|x|≤5`

↓

$x=z_{11}\cdot k_1\cdot z_{21} \wedge$
$|z_{11}|=0 \wedge$
$k_1\neq$"a" $\wedge$

...

$x=z_{14}\cdot k_4\cdot z_{24} \wedge$
$|z_{14}|=4 \wedge$
$k_4\neq$"a"

— Expand `substr`

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀

```
¬contains(x,"a")
```

↓

```
∀0≤n<|x|.substr(x,n,1)≠"a"
```

Expand `contains`

↓

```
substr(x,0,1)≠"a" ∧ … ∧ substr(x,4,1)≠"a"
```

Assuming bound |x|≤5

↓

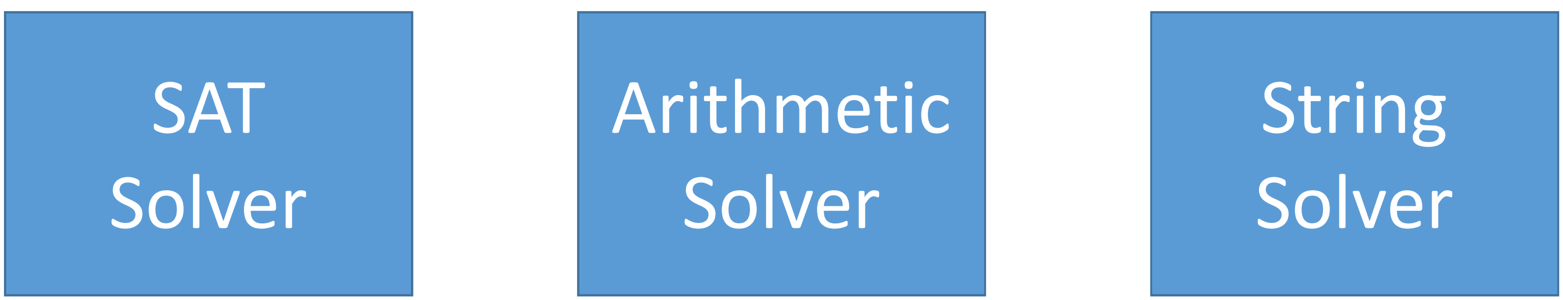
$x=z_{11}\cdot k_1\cdot z_{21} \wedge$ $\lvert z_{11}\rvert=0 \wedge$ $k_1\neq$"a" $\wedge$	...	$x=z_{14}\cdot k_4\cdot z_{24} \wedge$ $\lvert z_{14}\rvert=4 \wedge$ $k_4\neq$"a"

Expand `substr`

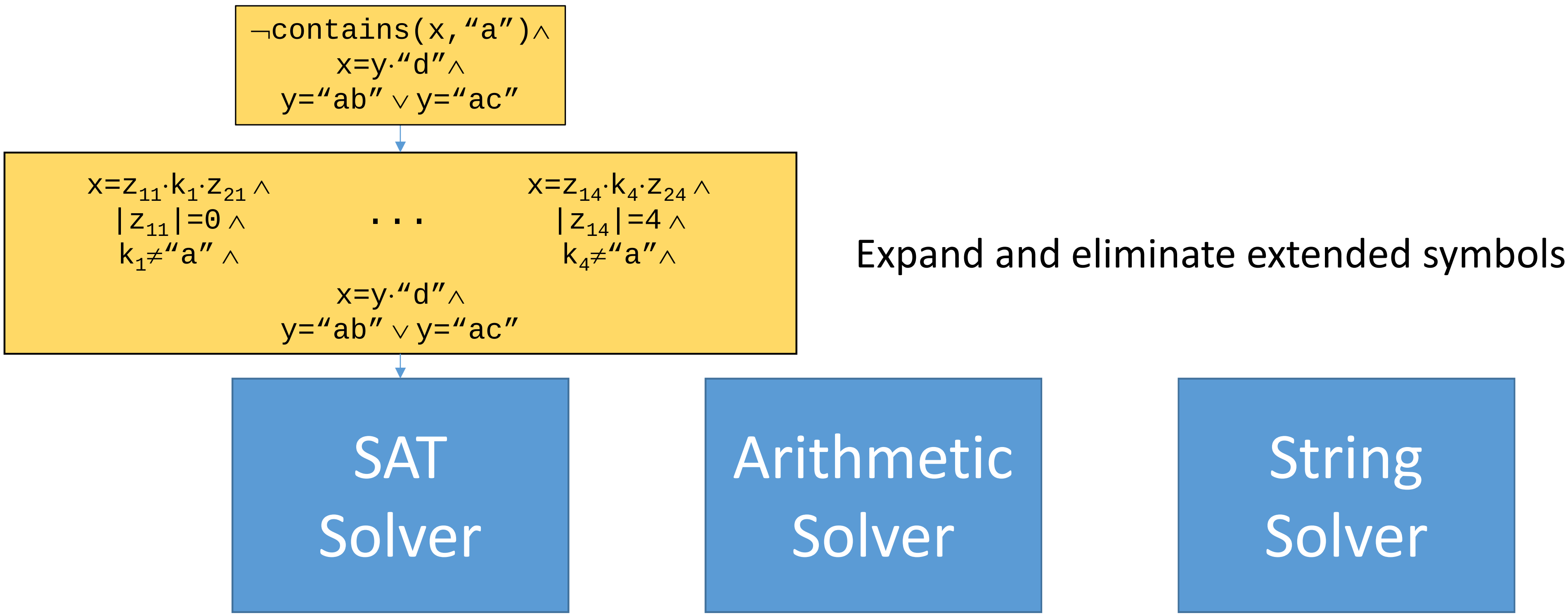
- Approach used by many current solvers
  [Bjorner et al 2009,Zheng et al 2013,Li et al 2013,Trinh et al 2014]

(Eager) Expansion of Extended Constraints

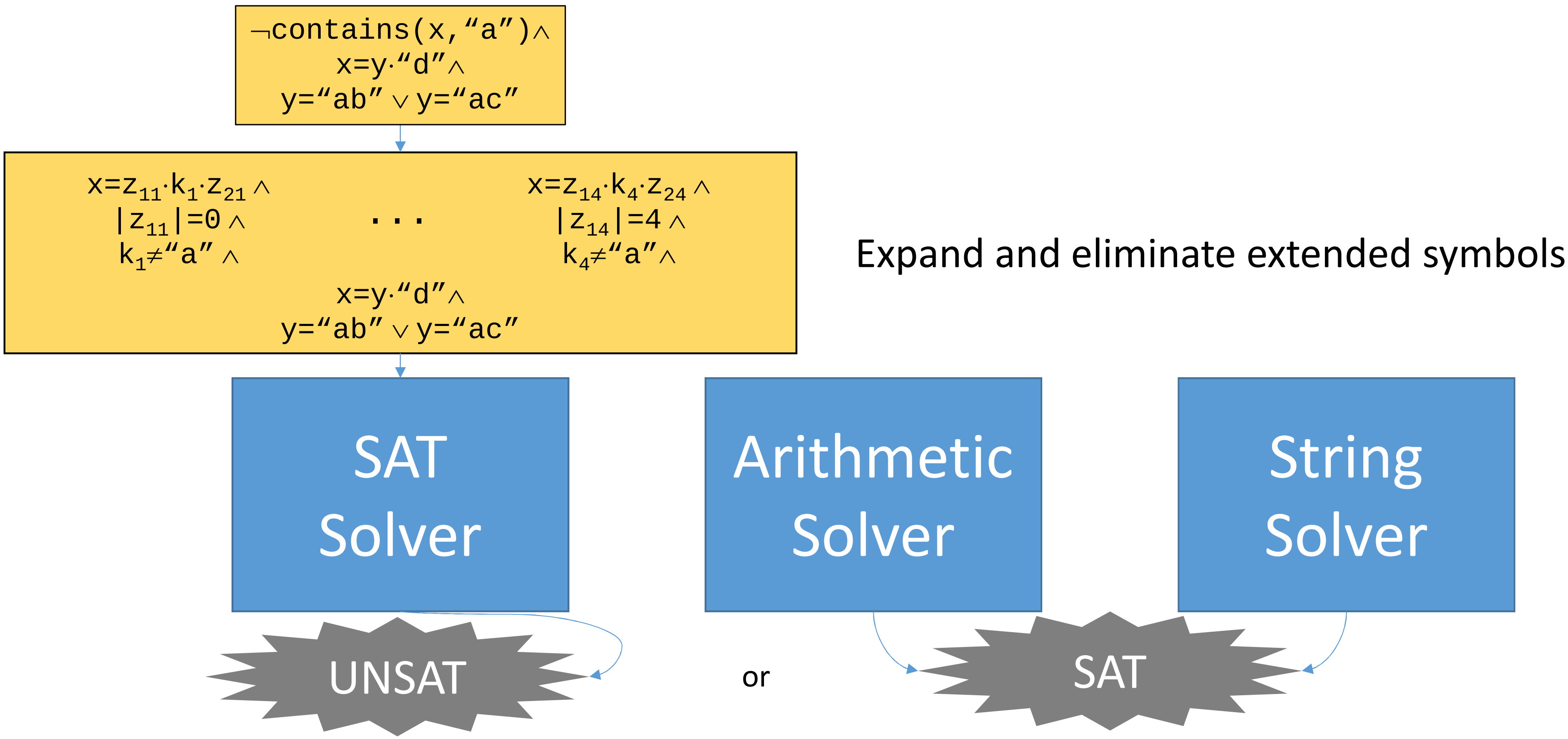
$\neg$contains(x,“a”)$\wedge$
x=y·“d”$\wedge$
y=“ab”$\vee$y=“ac”



(Eager) Expansion of Extended Constraints



(Eager) Expansion of Extended Constraints



(Lazy) Expansion of Extended Constraints

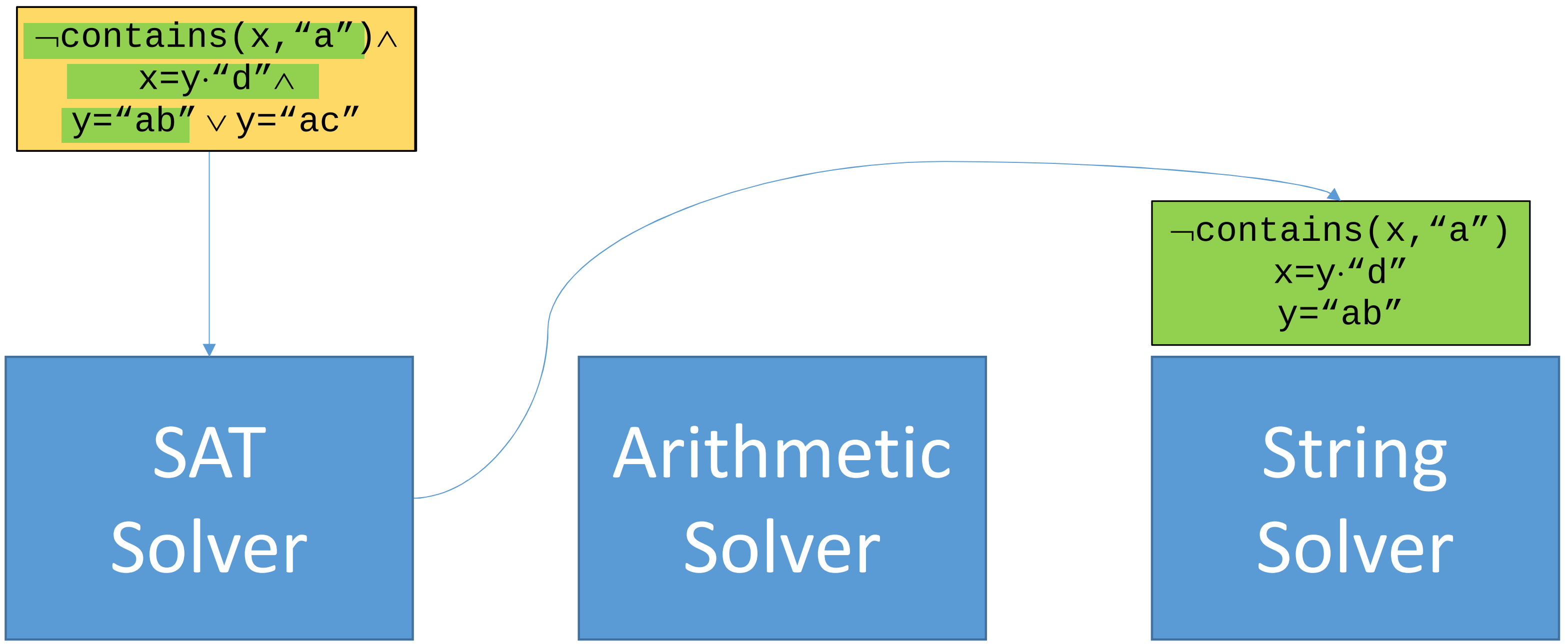
$\neg \texttt{contains}(x, \text{“a”}) \wedge$
$x = y \cdot \text{“d”} \wedge$
$y = \text{“ab”} \vee y = \text{“ac”}$

SAT Solver

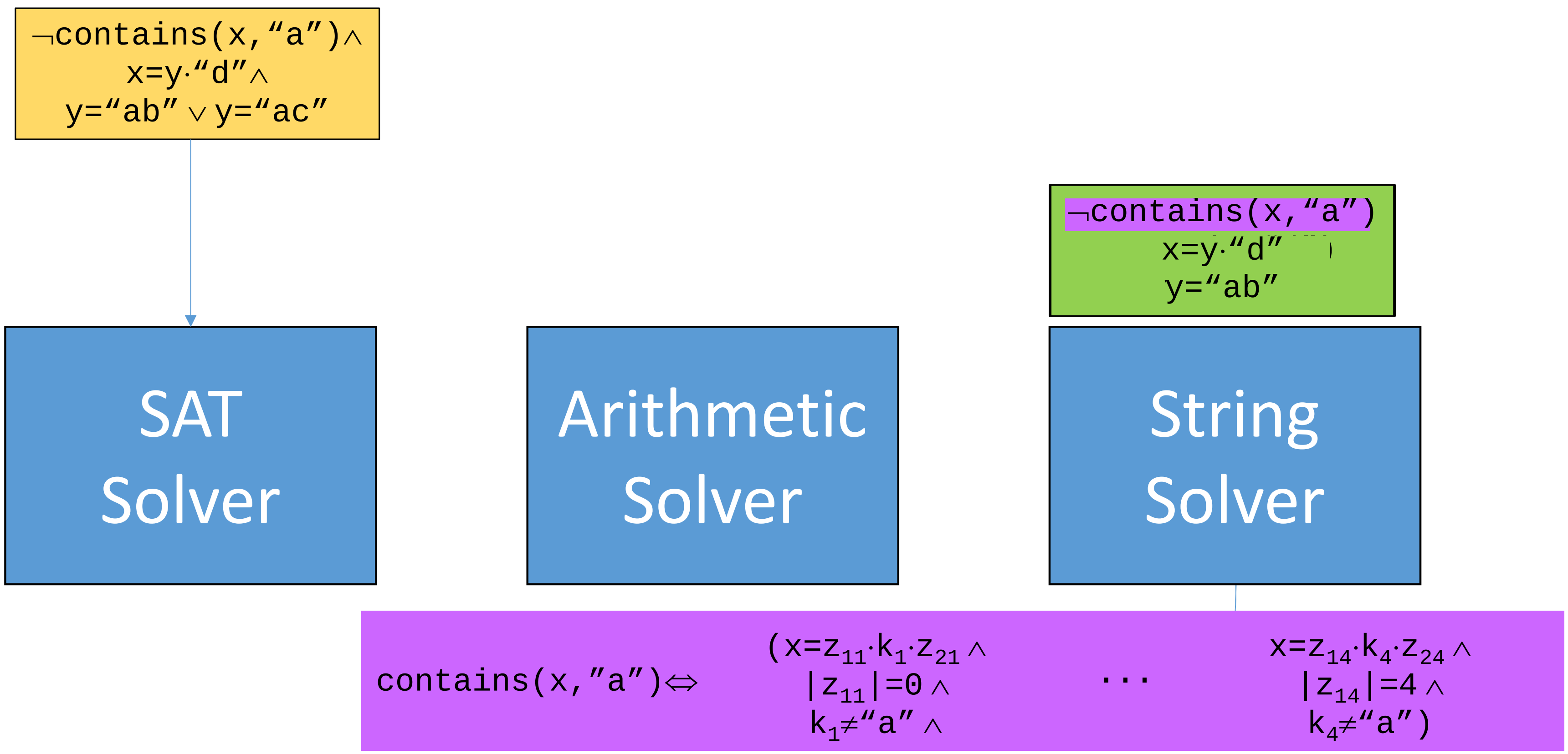
Arithmetic Solver

String Solver

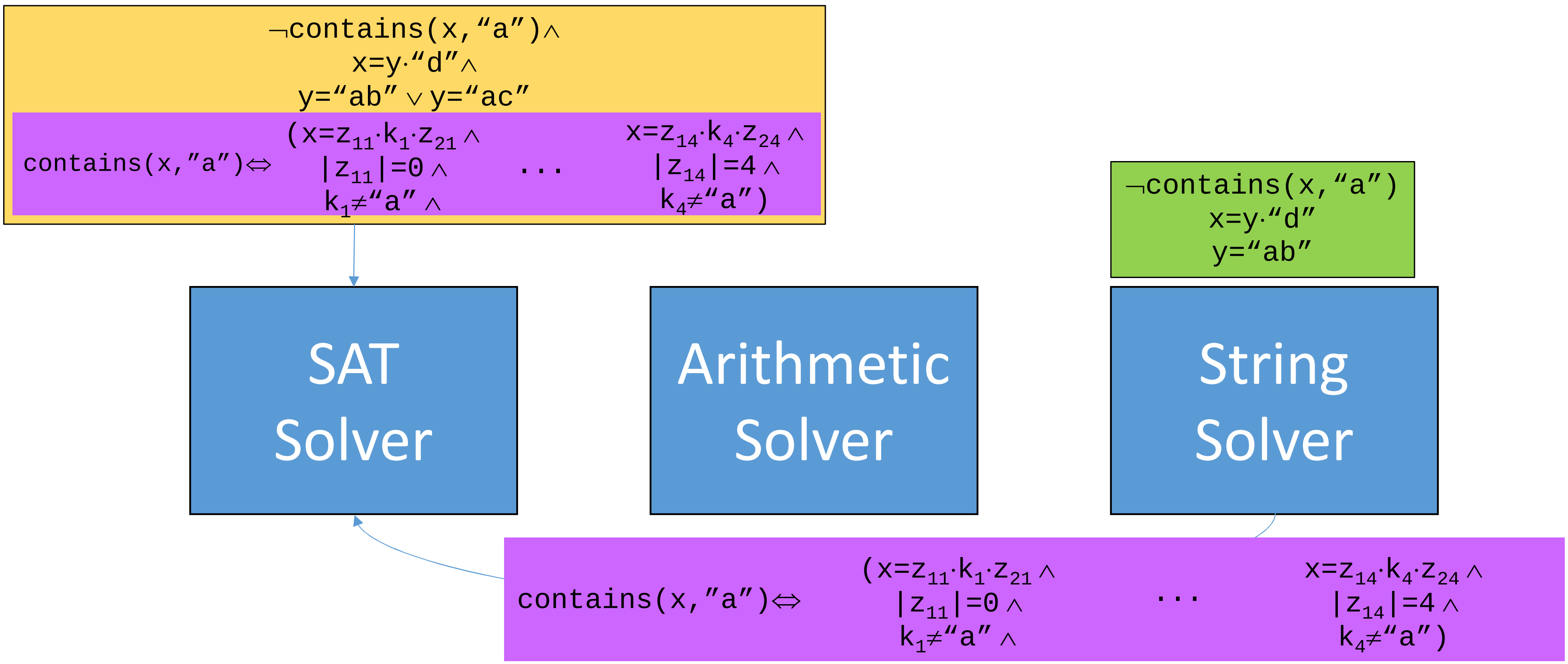
(Lazy) Expansion of Extended Constraints



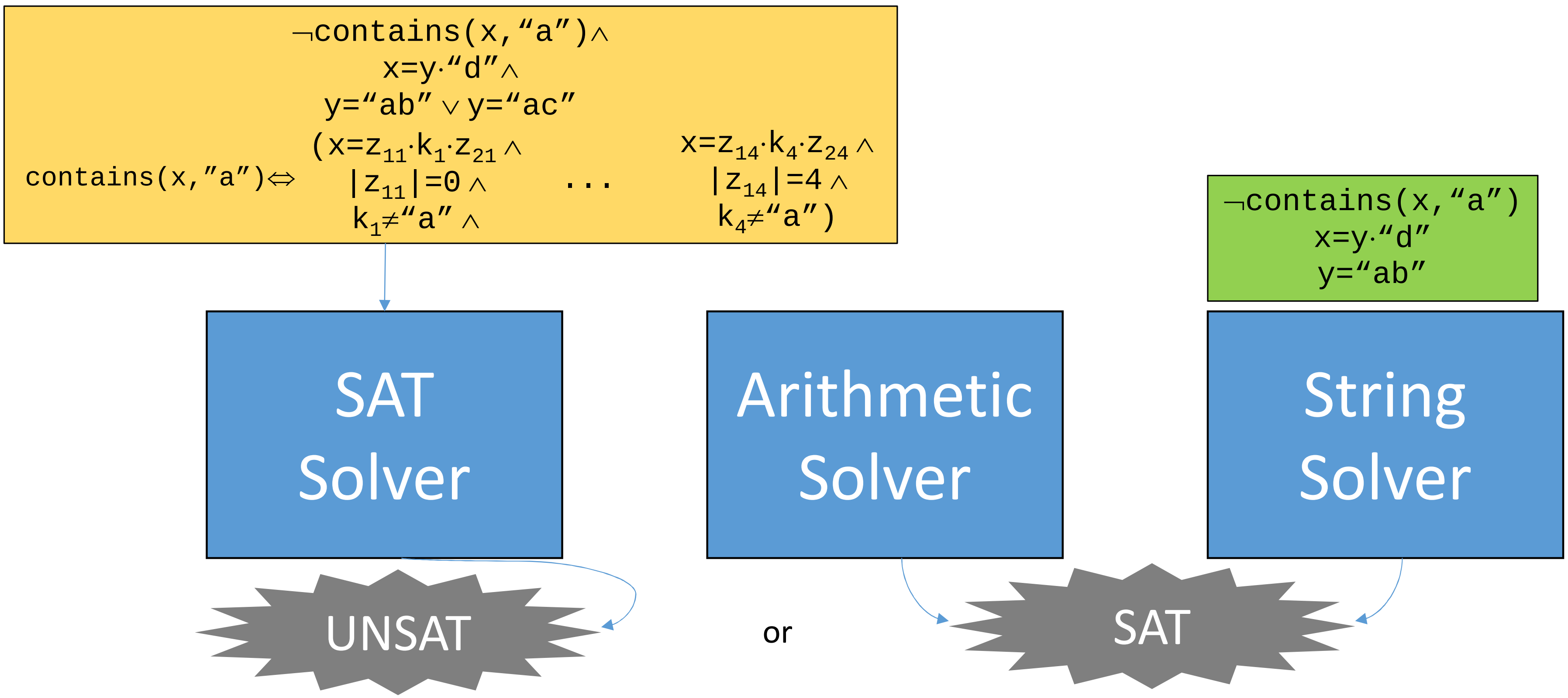
(Lazy) Expansion of Extended Constraints



(Lazy) Expansion of Extended Constraints

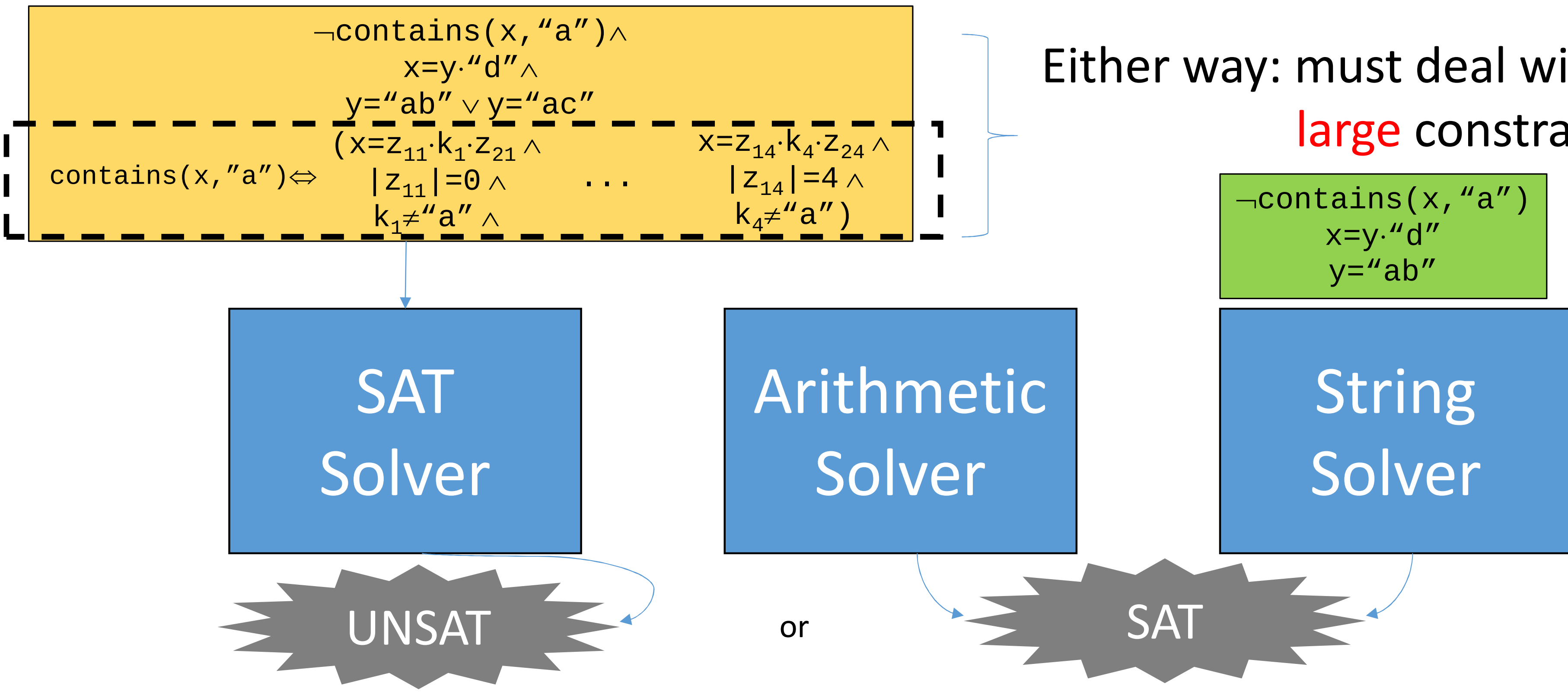


(Lazy) Expansion of Extended Constraints

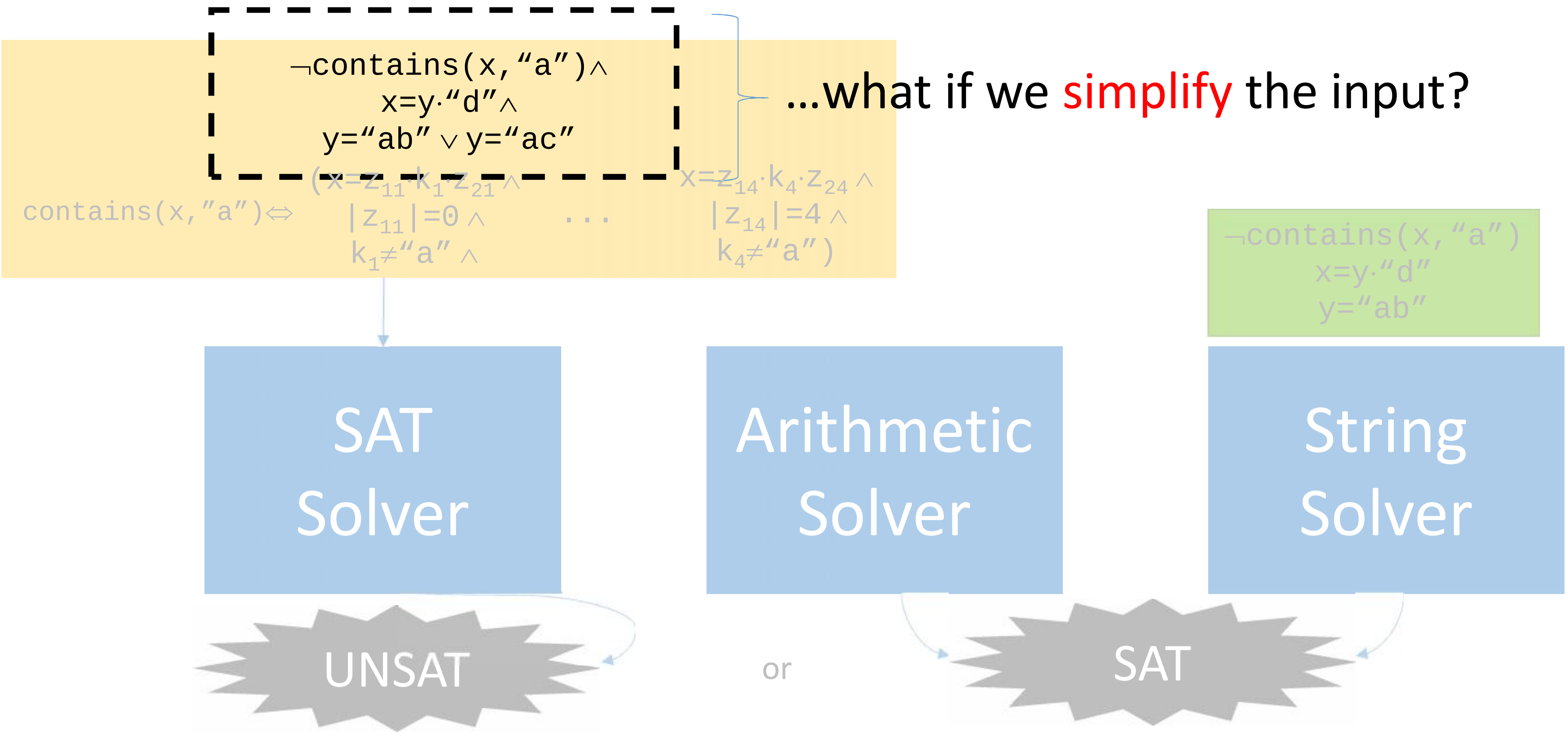


(Lazy) Expansion of Extended Constraints

Either way: must deal with a large constraint



(Lazy) Expansion of Extended Constraints



SMT Solvers + Simplification

- All SMT solvers implement *simplification* techniques

(also called *normalization* or *rewrite* rules)

```
¬contains(x,“a”)∧
x=y·“d”∧
y=“ab” ∨ y=“ac”
```

SMT Solvers + Simplification

- All SMT solvers implement *simplification* techniques

(also called *normalization* or *rewrite* rules)

¬contains(x,“a”)∧
x=y·“d”∧
y=“ab” ∨ y=“ac”

↓

¬contains(**y·“d”**,“a”)∧
y=“ab” ∨ y=“ac”

since x=y·”d”

SMT Solvers + Simplification

- All SMT solvers implement *simplification* techniques

(also called *normalization* or *rewrite* rules)

¬contains(x,“a”)∧
x=y·“d”∧
y=“ab” ∨ y=“ac”

↓

¬contains(y·“d”,“a”)∧
y=“ab” ∨ y=“ac”

since x=y·”d”

↓

¬**contains(y,“a”)**∧
y=“ab” ∨ y=“ac”

since contains(y·“d”,”a”)⇔contains(y,“a”)

SMT Solvers + Simplification

- All SMT solvers implement *simplification* techniques

(also called *normalization* or *rewrite* rules)

¬contains(x,“a”)∧
x=y·“d”∧
y=“ab” ∨ y=“ac”

↓

¬contains(y·“d”,“a”)∧
y=“ab” ∨ y=“ac”

since x=y·”d”

↓

¬contains(y,“a”)∧
y=“ab” ∨ y=“ac”

since contains(y·“d”,”a”)⇔contains(y,“a”)

- Leads to smaller inputs, simpler procedures

(Lazy) Expansion + Simplification

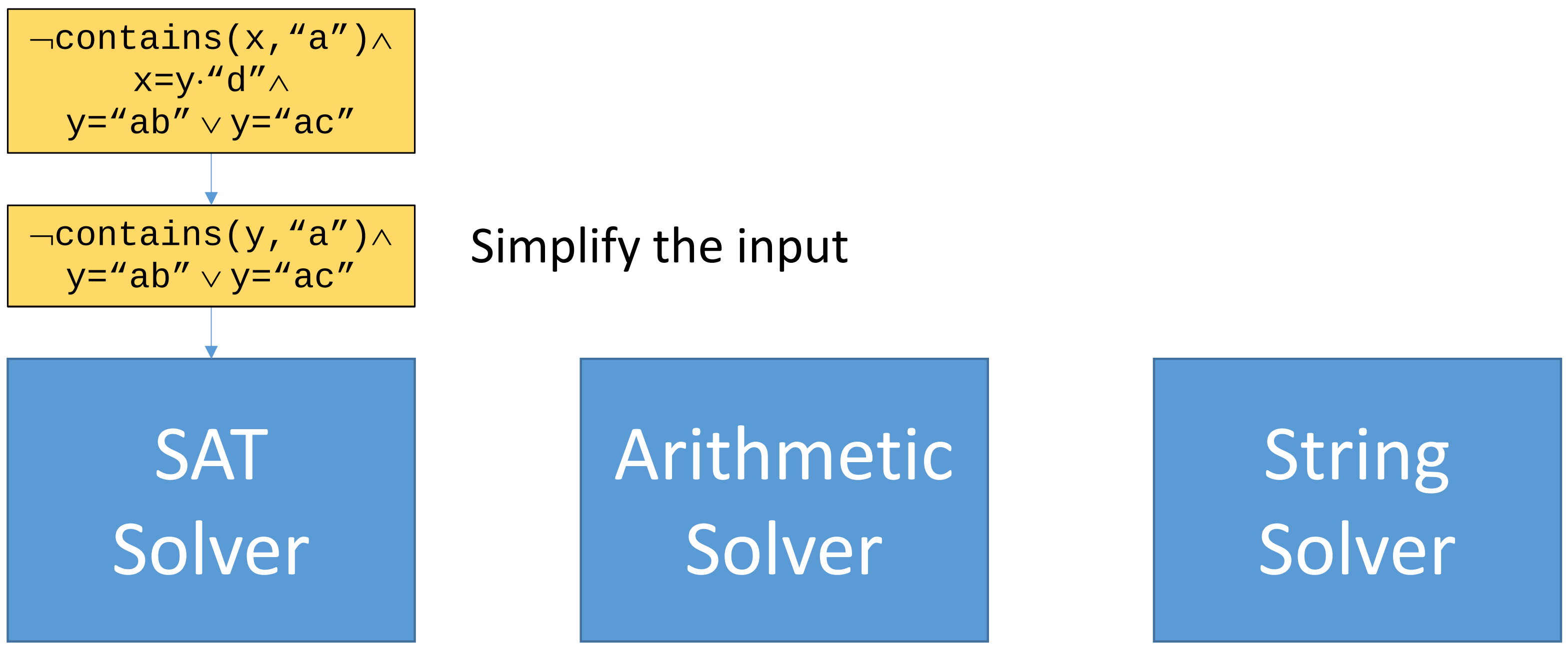
```
¬contains(x,“a”)∧
x=y·“d”∧
y=“ab” ∨ y=“ac”
```

SAT Solver

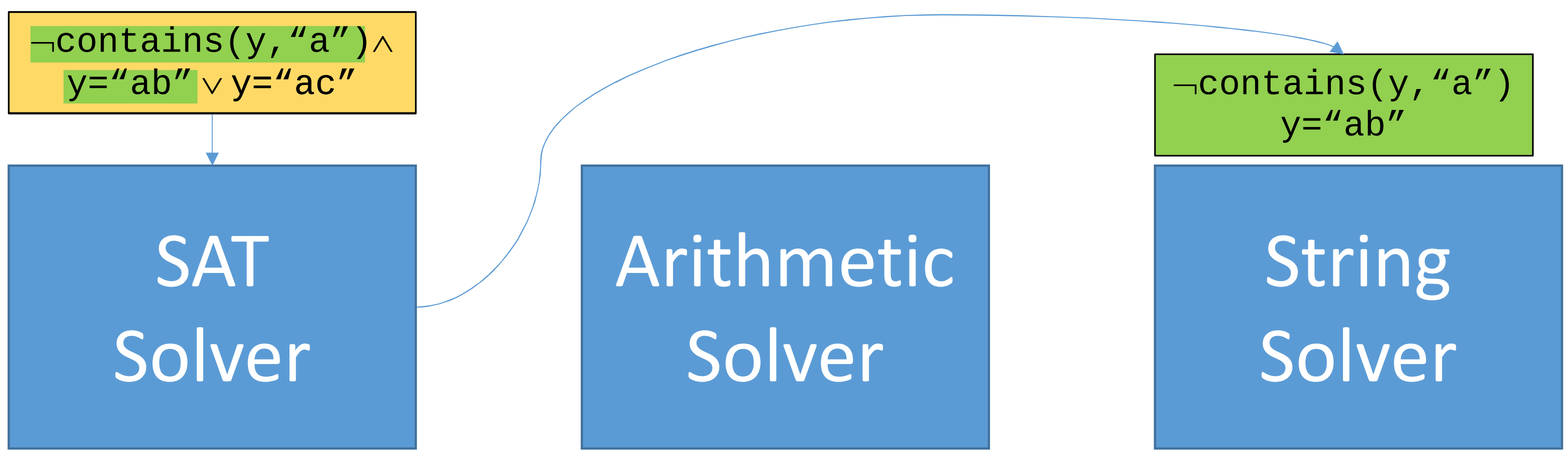
Arithmetic Solver

String Solver

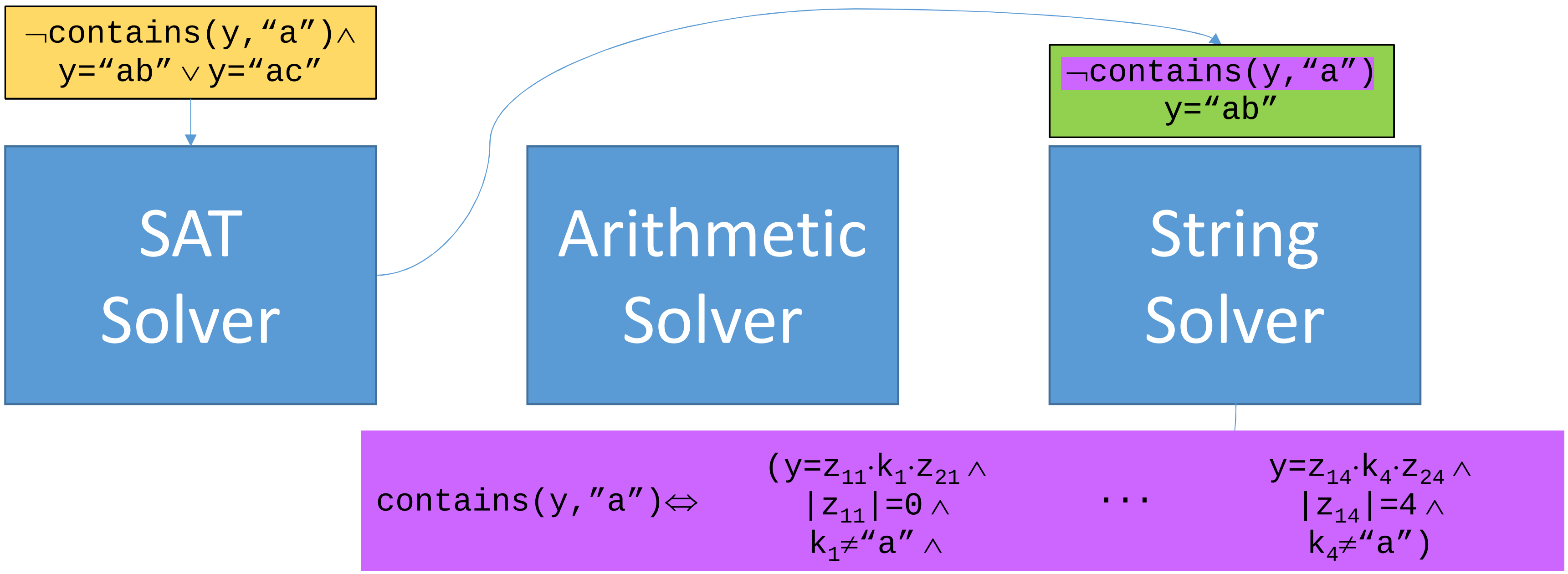
(Lazy) Expansion + Simplification



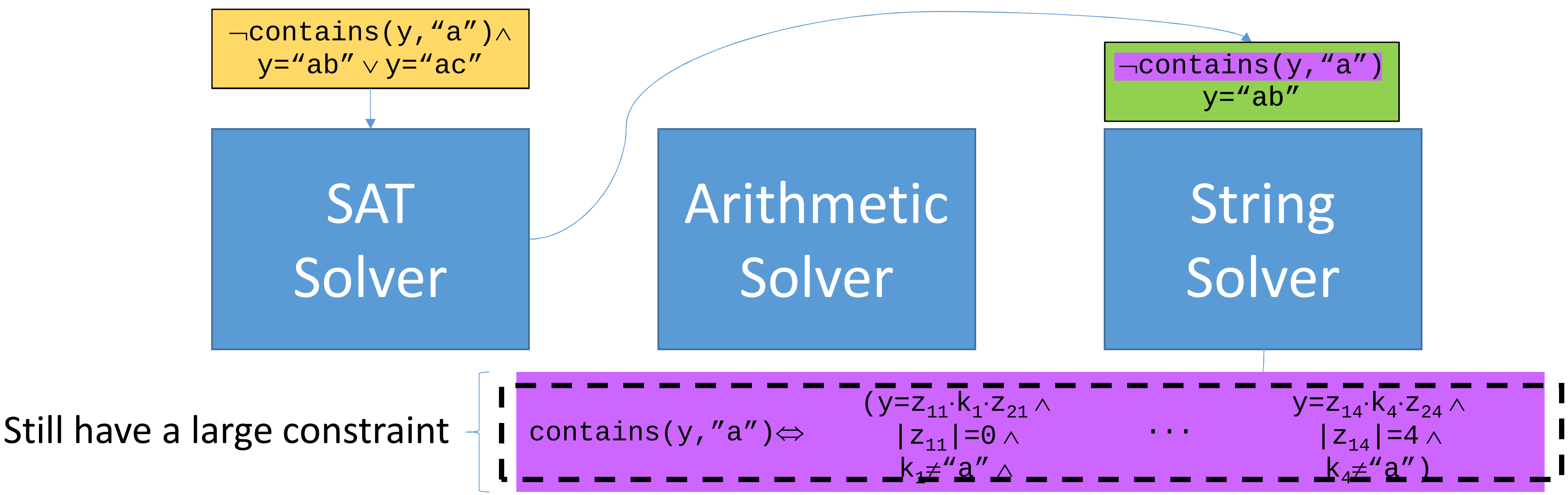
(Lazy) Expansion + Simplification



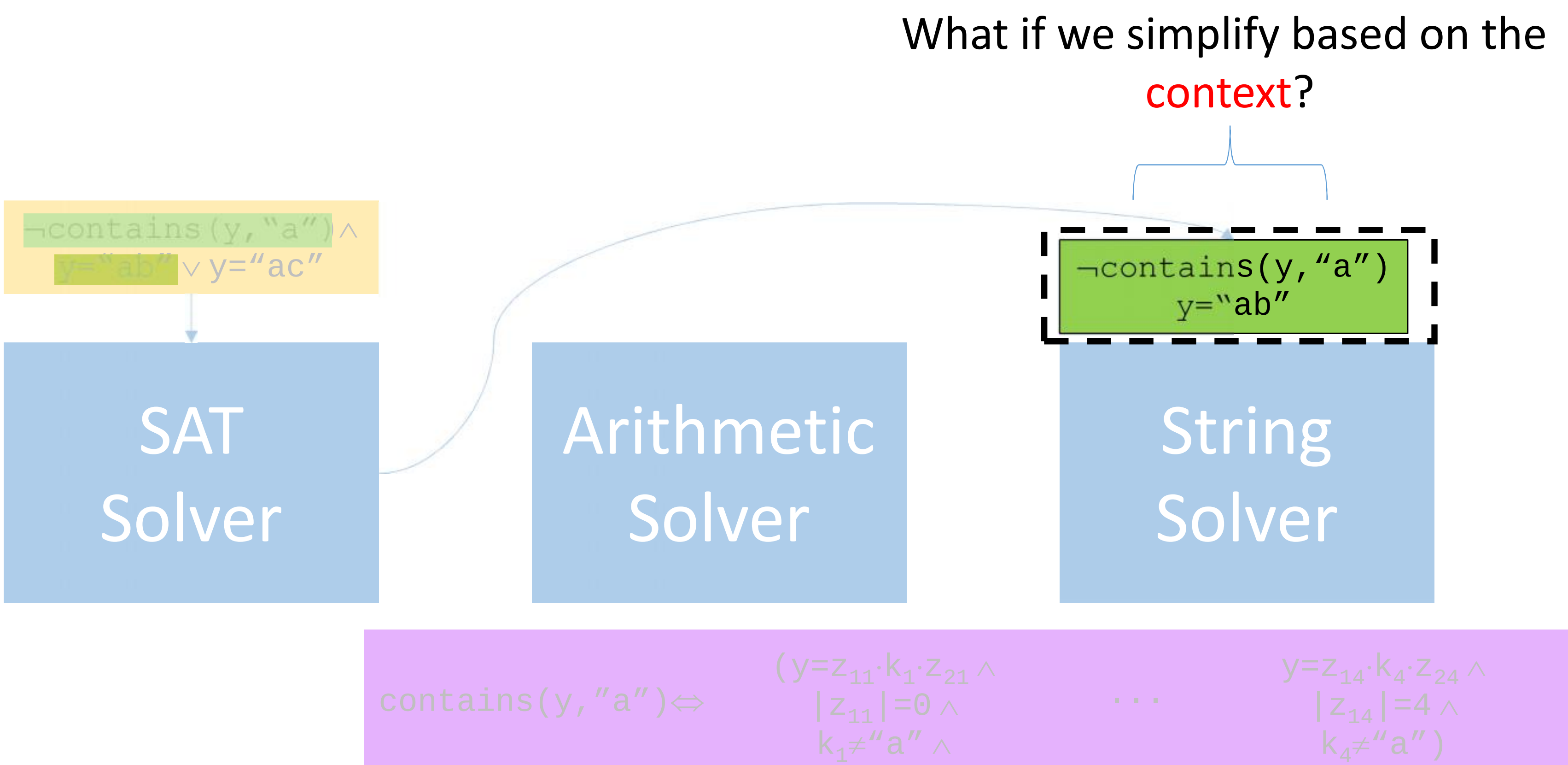
(Lazy) Expansion + Simplification



(Lazy) Expansion + Simplification



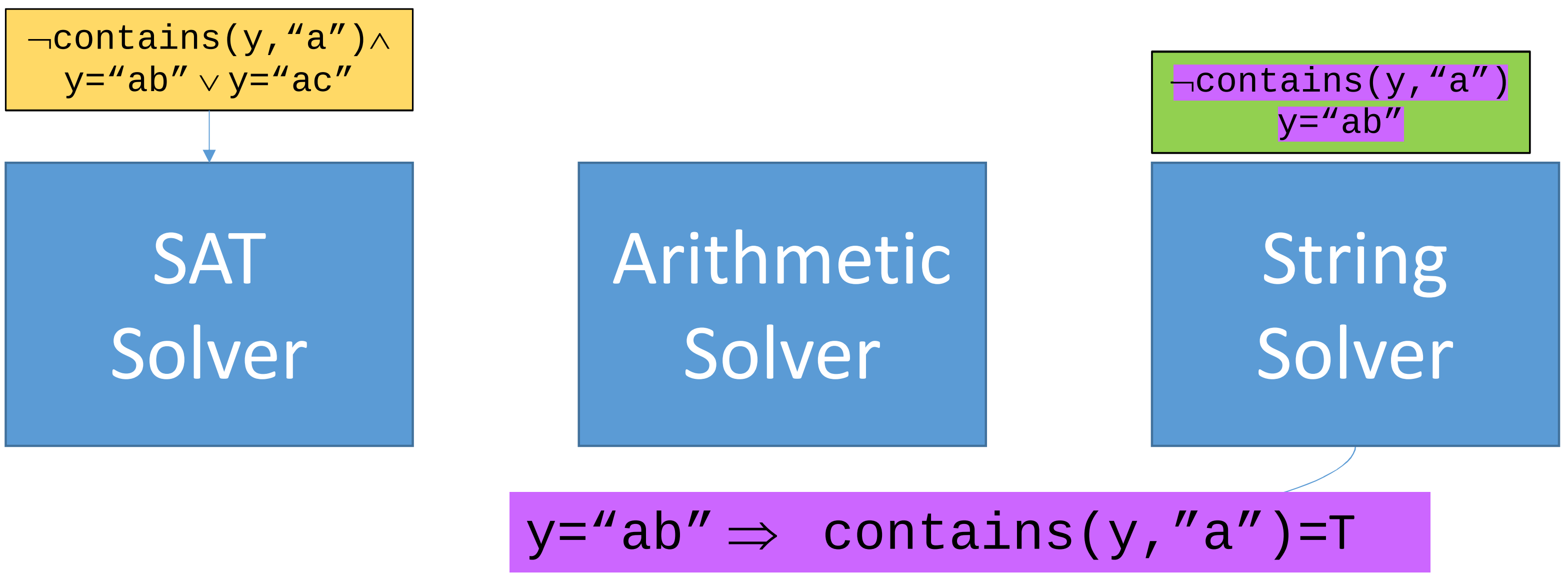
(Lazy) Expansion + Simplification



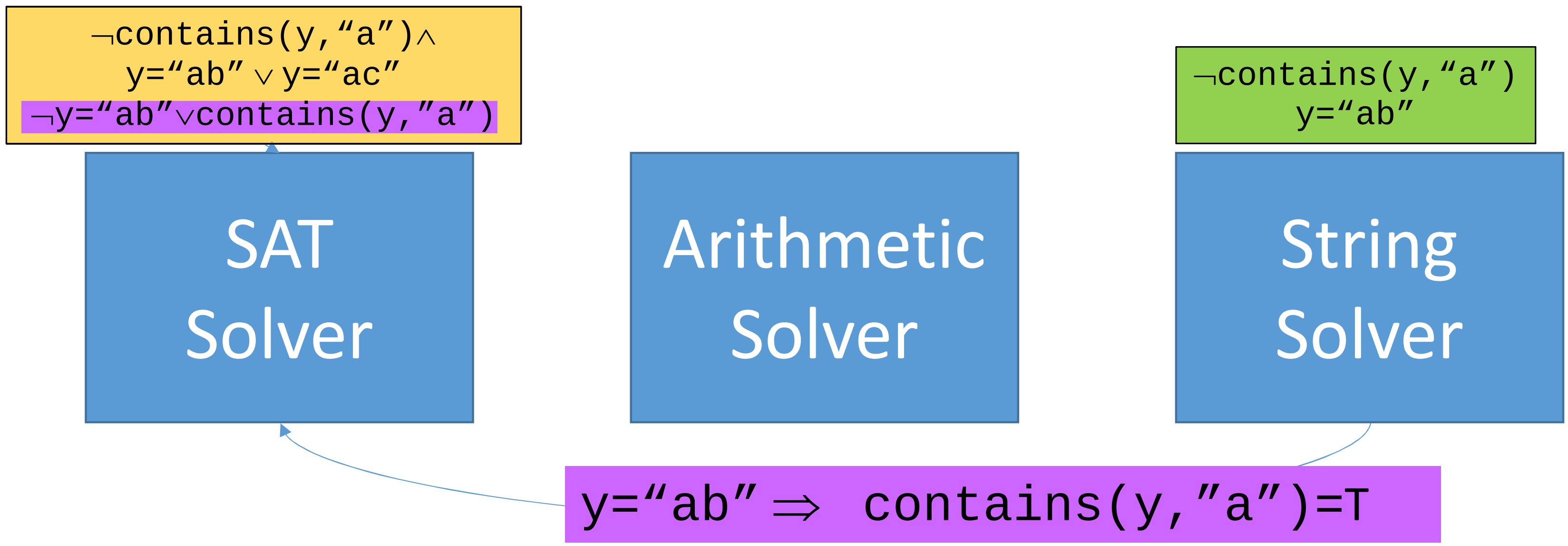
(Lazy) Expansion + Context-Dependent Simplification

Since `contains(y,”a”)` is true when `y=“ab”` …

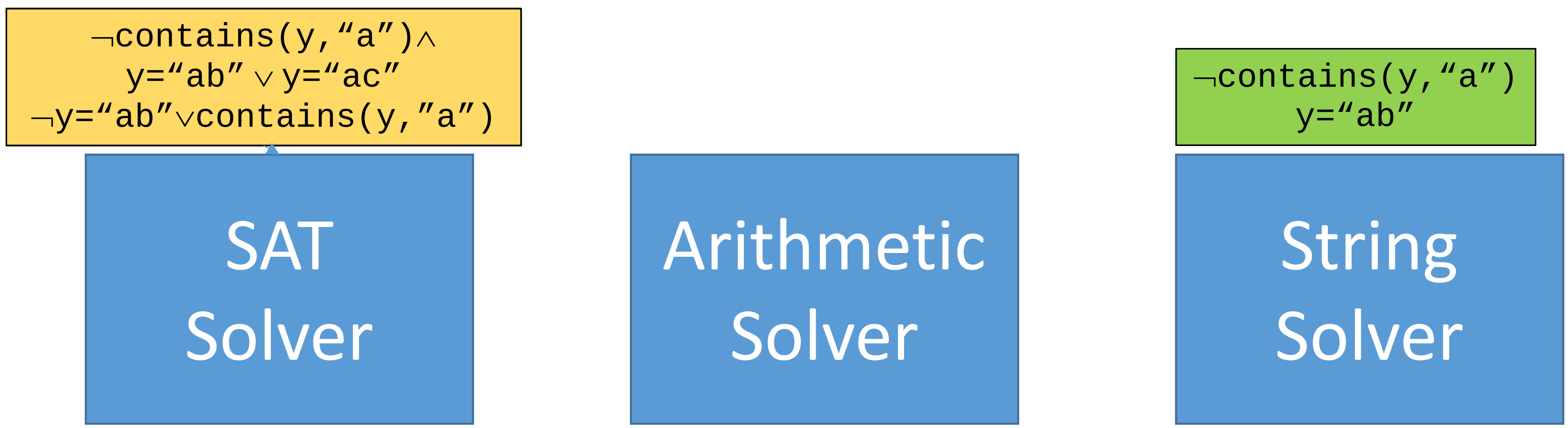
(Lazy) Expansion + Context-Dependent Simplification



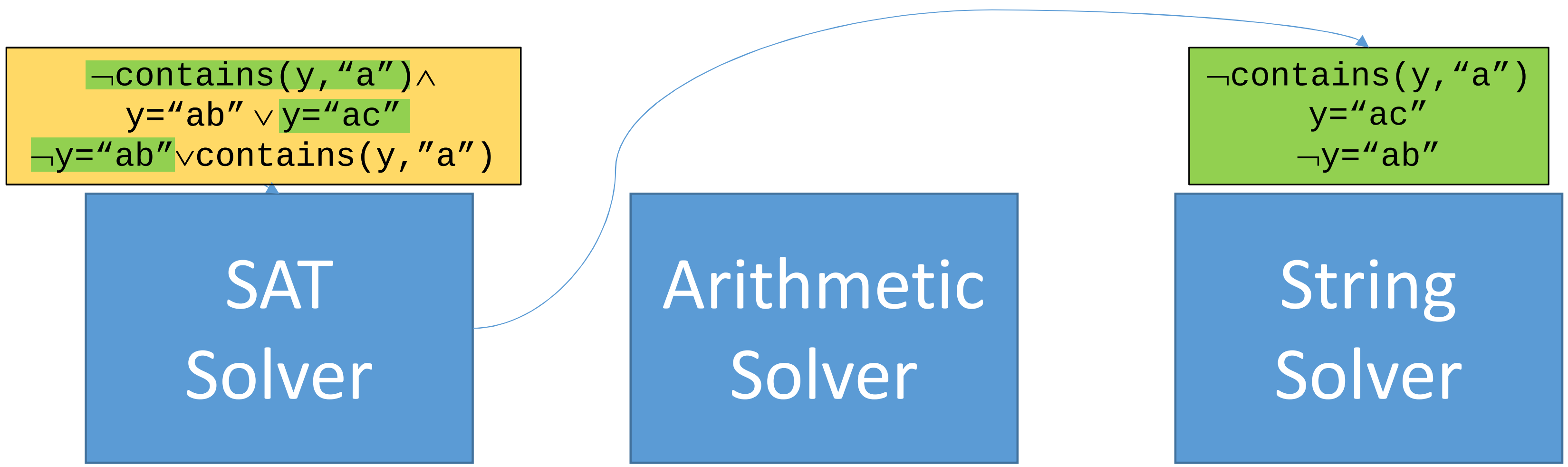
(Lazy) Expansion + Context-Dependent Simplification



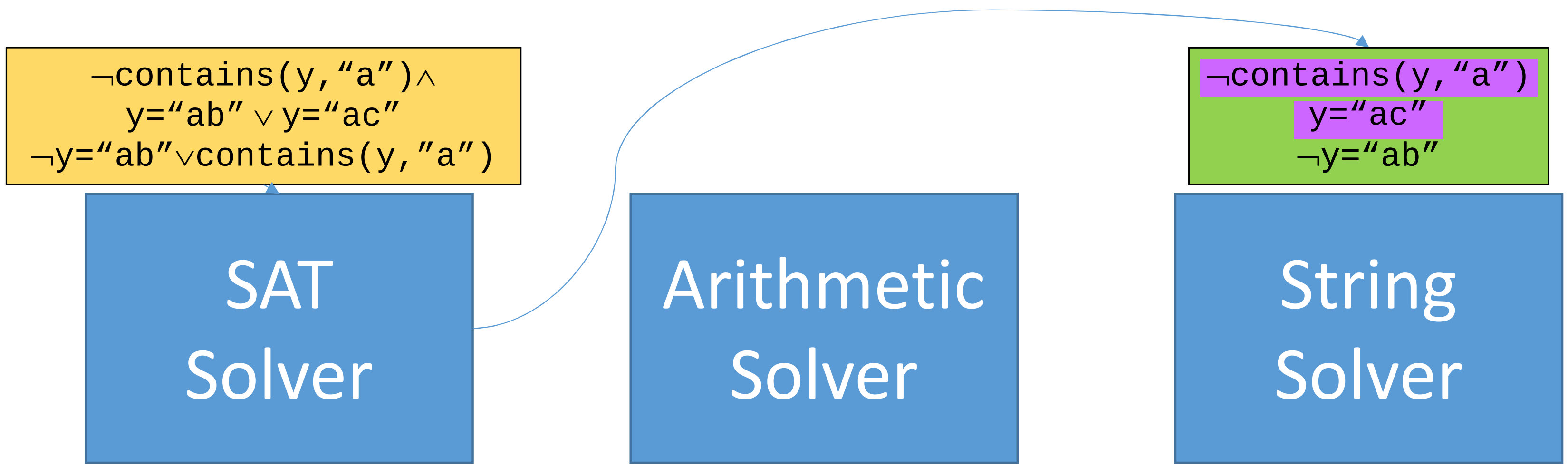
(Lazy) Expansion + Context-Dependent Simplification



(Lazy) Expansion + Context-Dependent Simplification

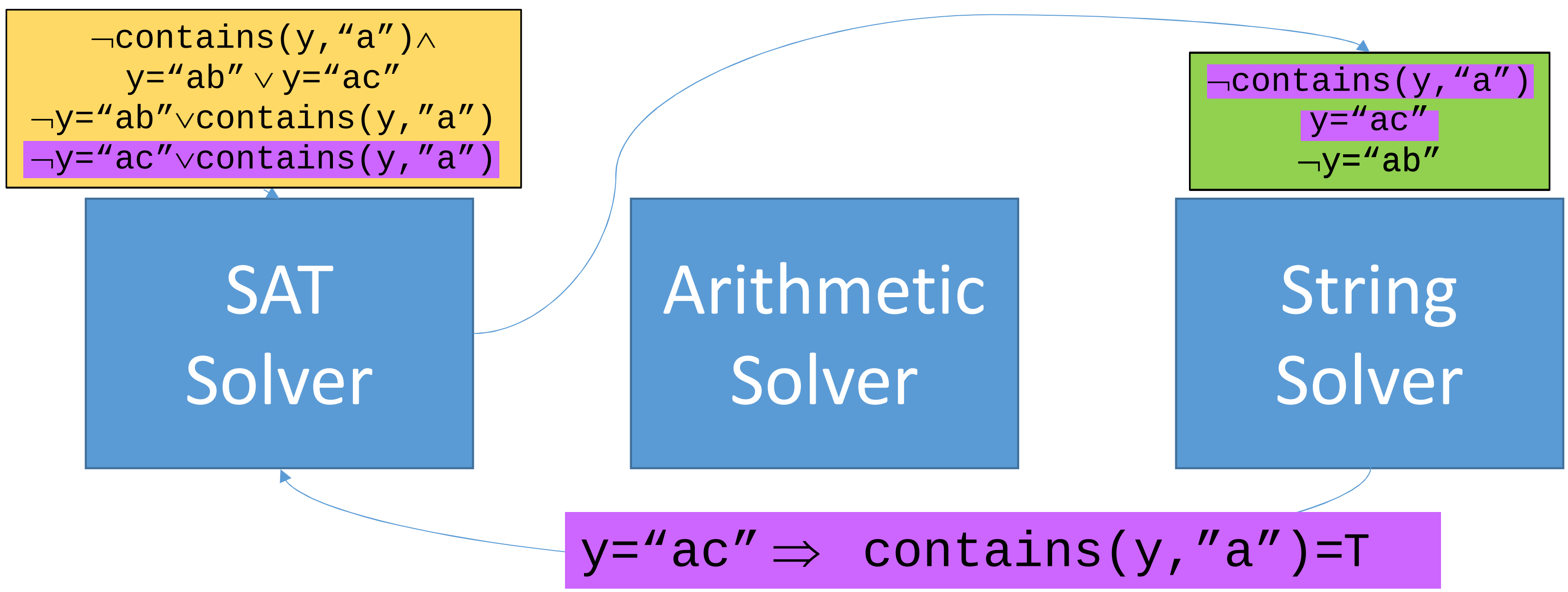


(Lazy) Expansion + Context-Dependent Simplification

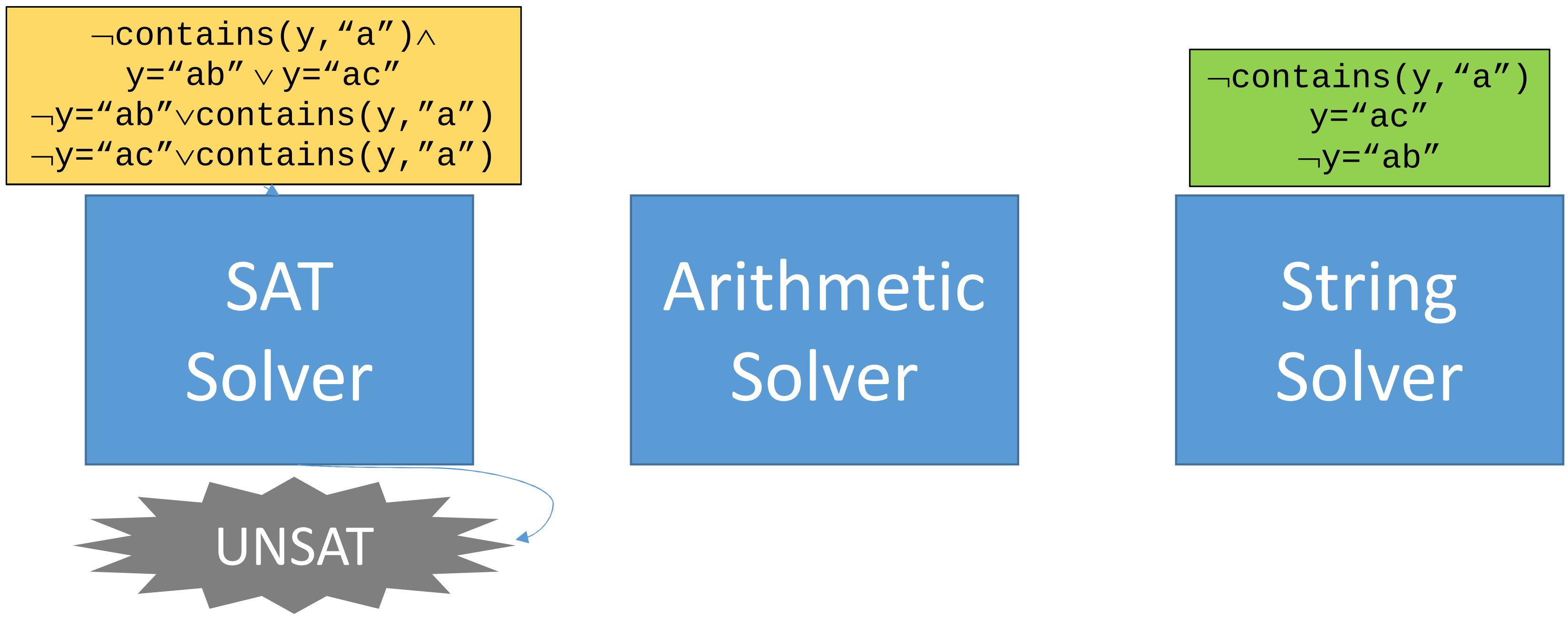


contains(y,”a”) is also true when y=“ac” …

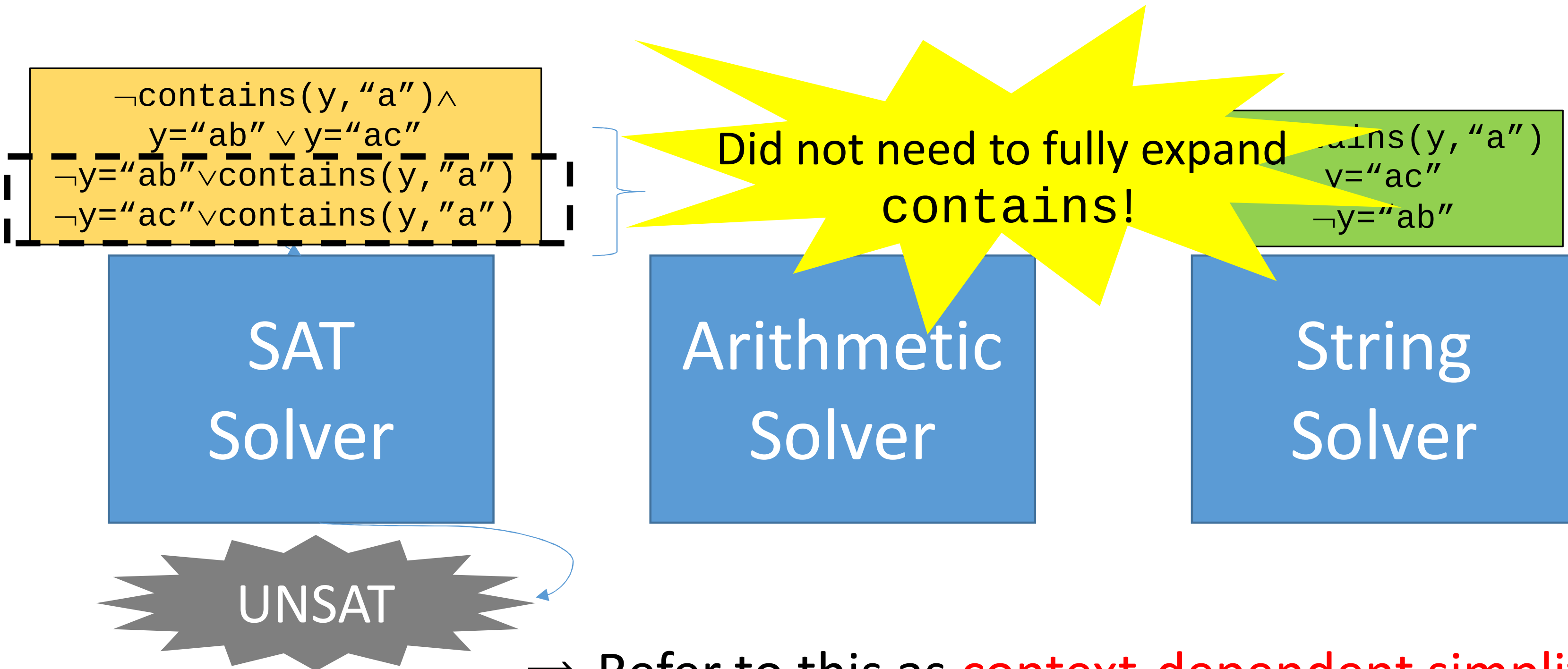
(Lazy) Expansion + Context-Dependent Simplification



(Lazy) Expansion + Context-Dependent Simplification

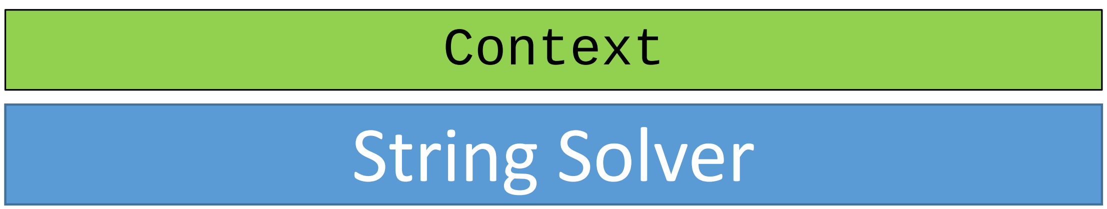


(Lazy) Expansion + Context-Dependent Simplification

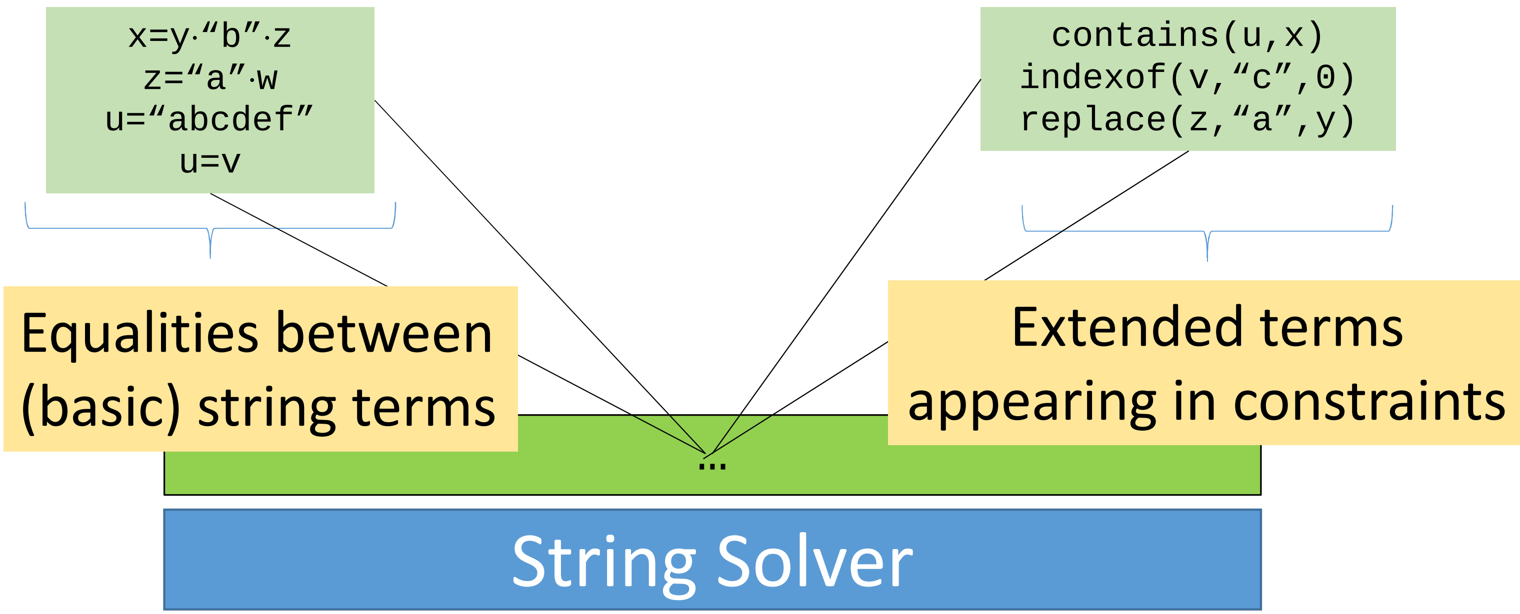


⇒ Refer to this as context-dependent simplification

Context-Dependent Simplification



Context-Dependent Simplification



Context-Dependent Simplification

```
x=y·"b"·z
z="a"·w
u="abcdef"
u=v
```

```
contains(u,x)
indexof(v,"c",0)
replace(z,"a",y)
```

…

String Solver

Context-Dependent Simplification

```
x=y·"b"·z
z="a"·w
u="abcdef"
u=v
```

```
x→y·"ba"·w
y→y
z→"a"·w
w→w
u→"abcdef"
v→"abcdef"
```

```
contains(u,x)
indexof(v,"c",0)
replace(z,"a",y)
```

Construct *derivable substitution* over basic variables

…

String Solver

Context-Dependent Simplification

x=y·“b”·z
z=“a”·w
u=“abcdef”
u=v

x→y·“ba”·w
y→y
z→“a”·w
w→w
u→“abcdef”
v→“abcdef”

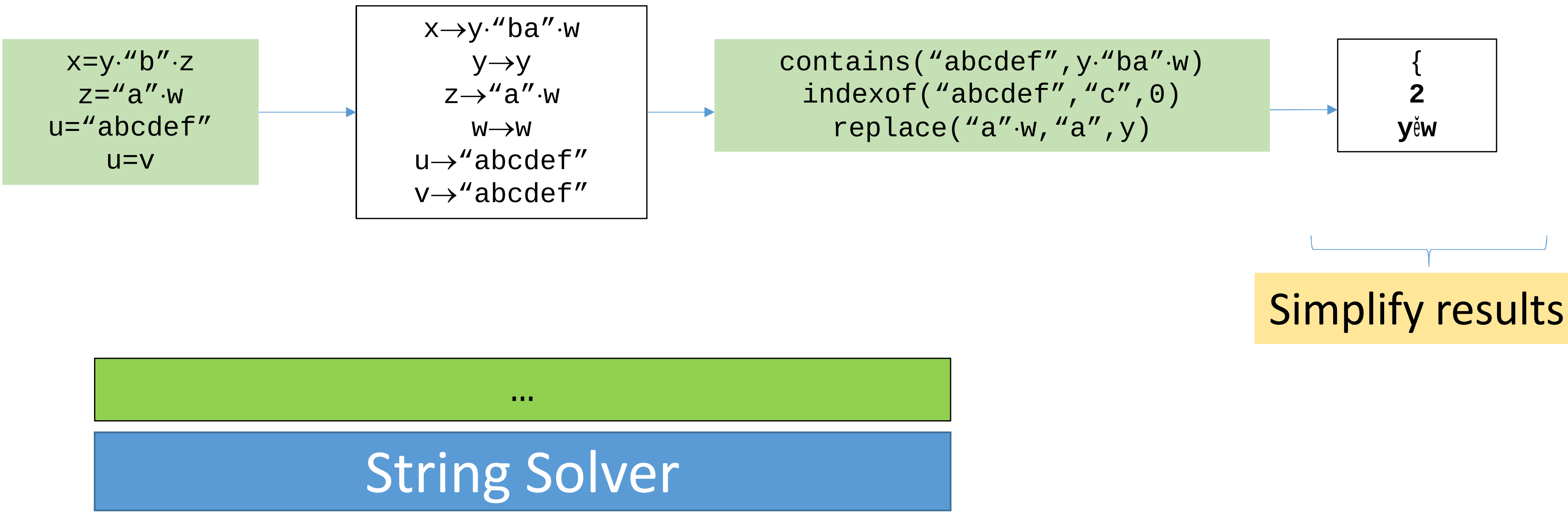
contains(**“abcdef”**,**y·“ba”·w**)
indexof(**“abcdef”**,“c”,0)
replace(**“a”·w**,“a”,y)

Apply this substitution to extended terms

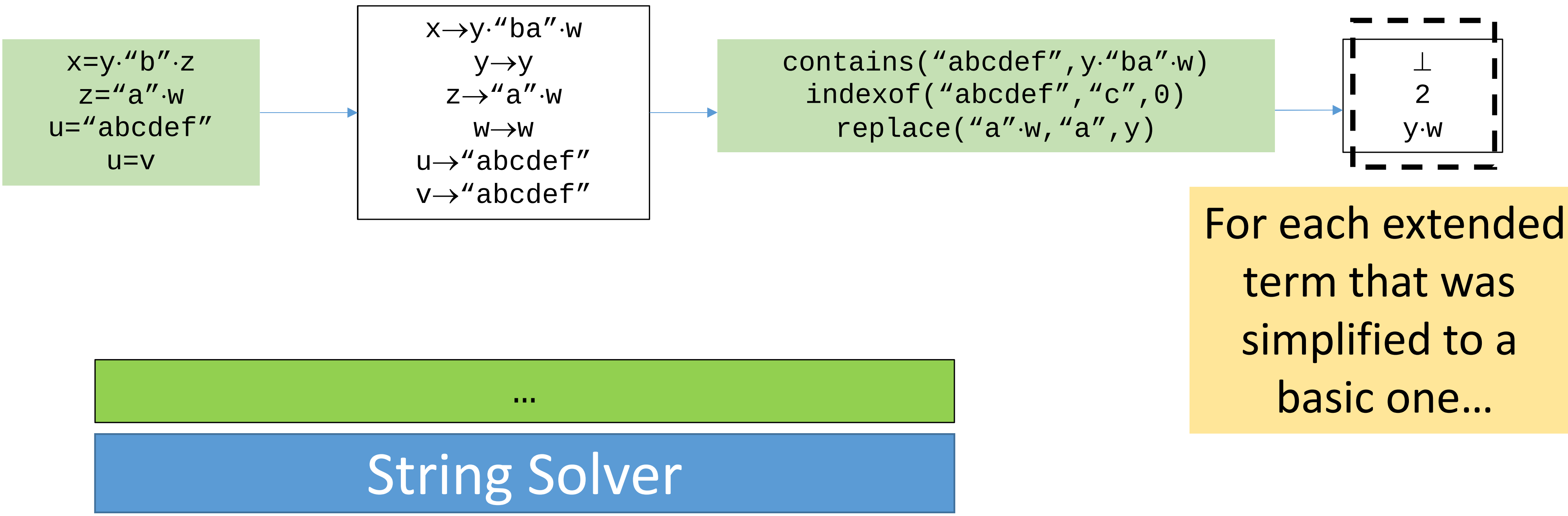
…

String Solver

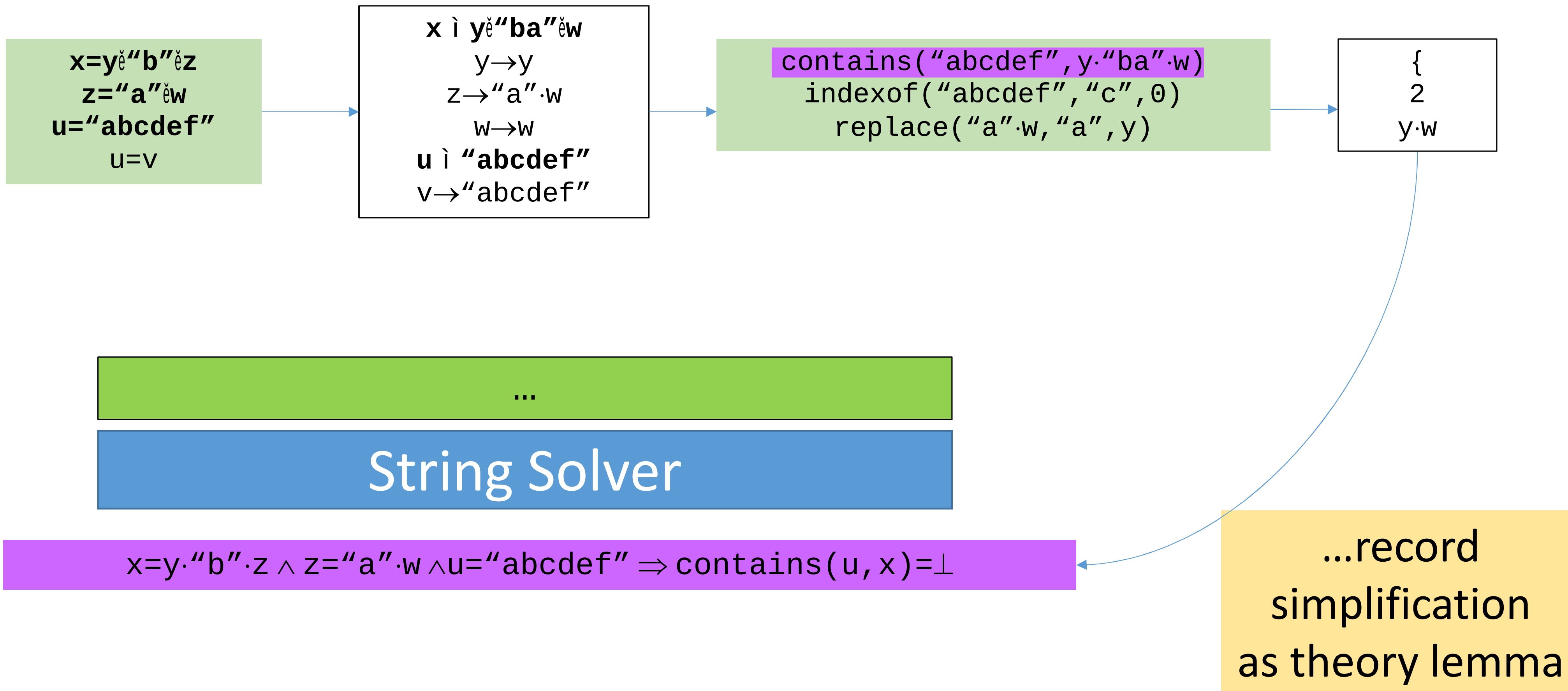
Context-Dependent Simplification



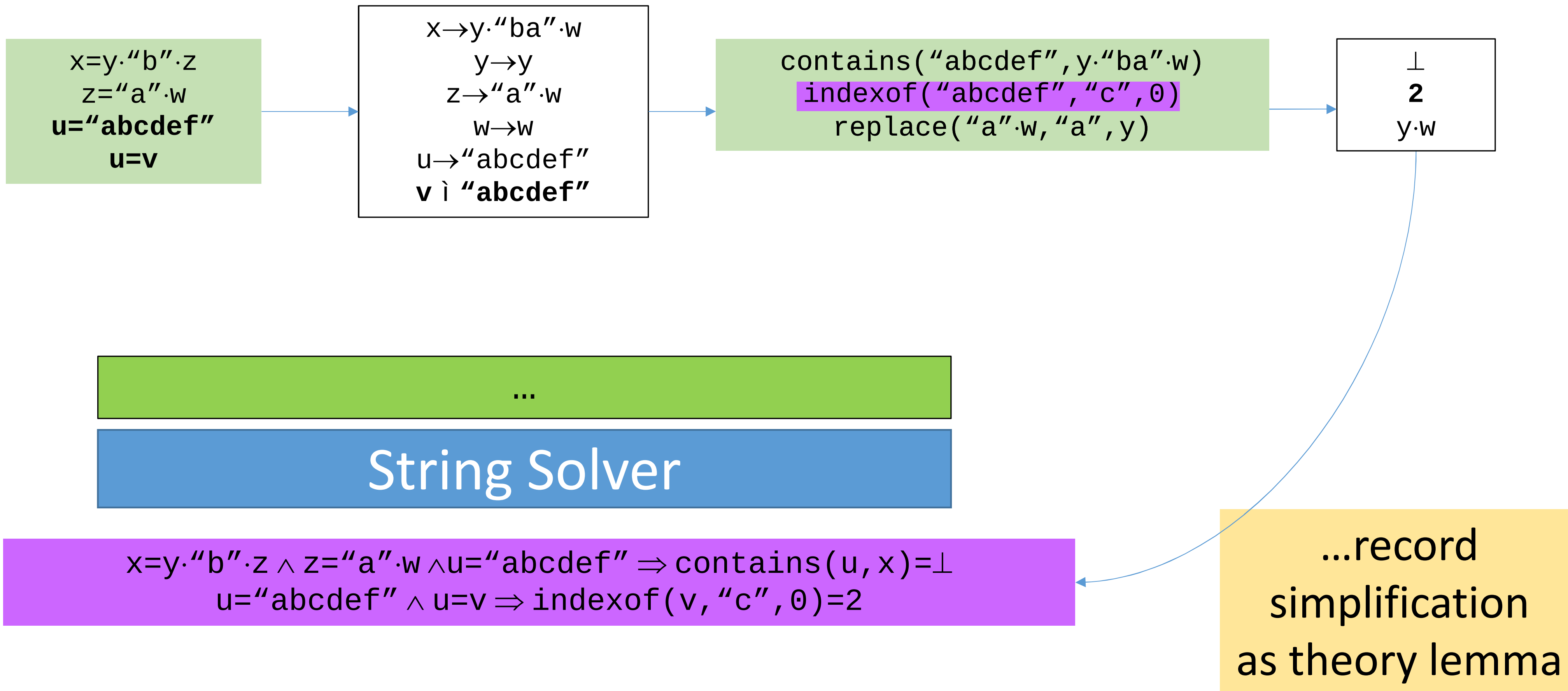
Context-Dependent Simplification



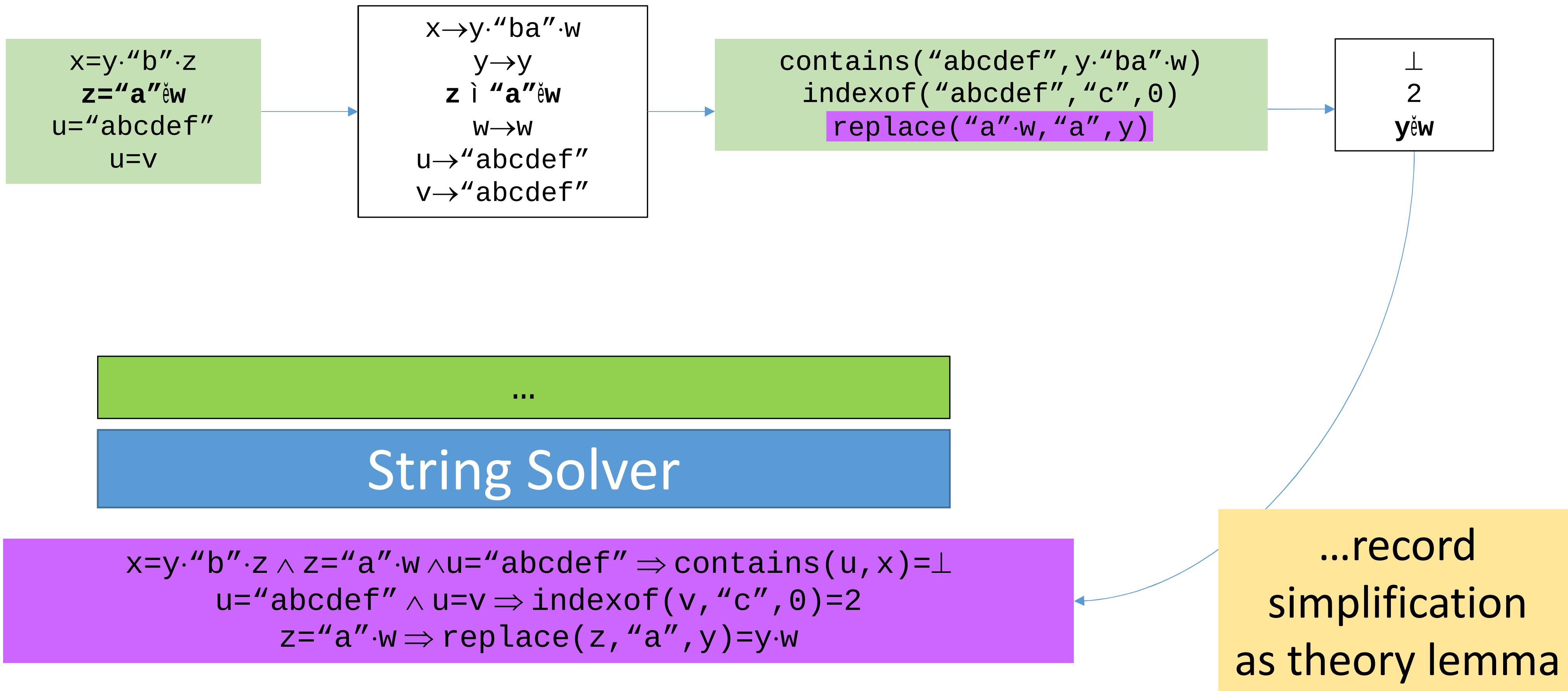
Context-Dependent Simplification



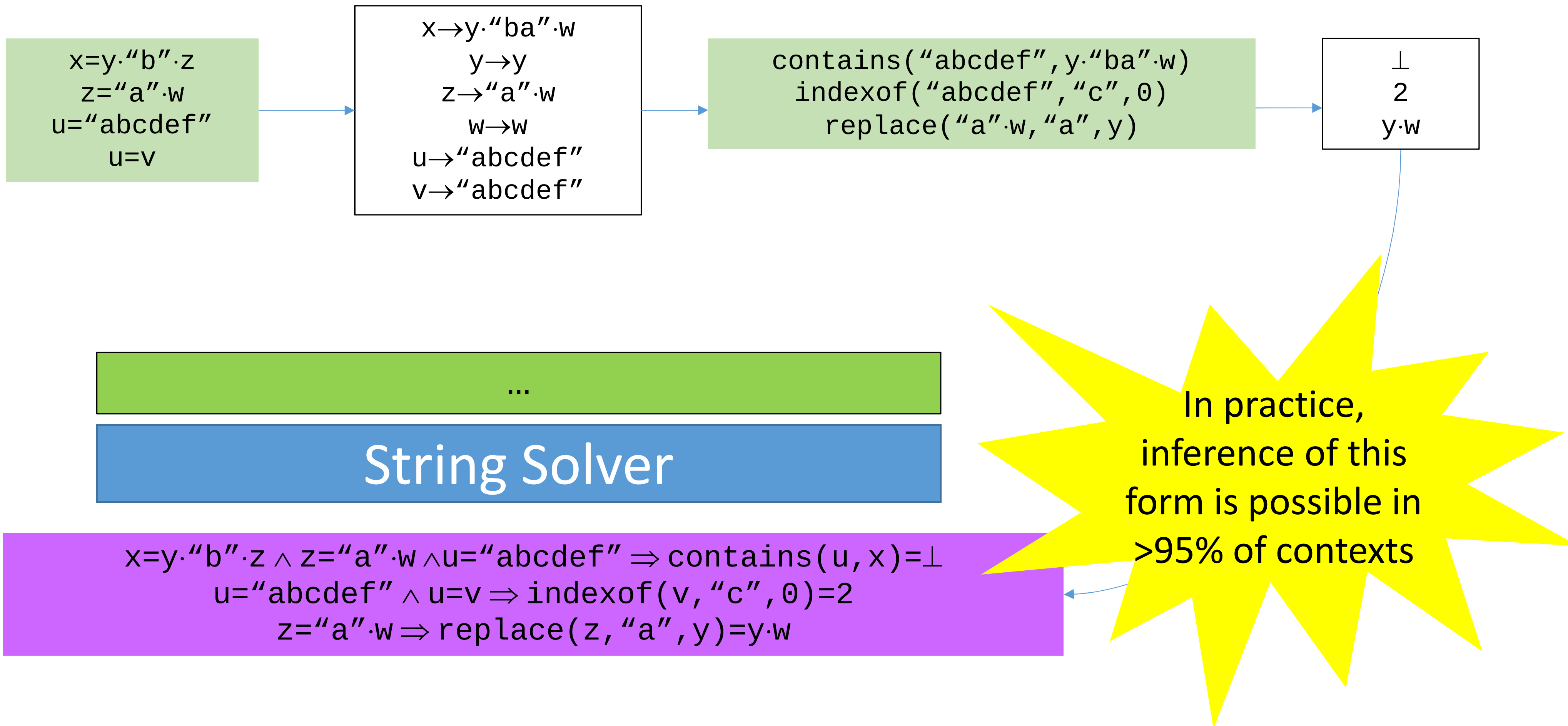
Context-Dependent Simplification



Context-Dependent Simplification



Context-Dependent Simplification



Simplification Rules for Strings

- Unlike arithmetic:

`x+x+7*y=y-4` → `2*x+6*y+4=0`

…simplification rules for strings are highly non-trivial:

`substr(x·“abcd”,1+len(x),2)` → `“bc”`

`contains(“abcde”,“b”·x·“a”)` → ⊥

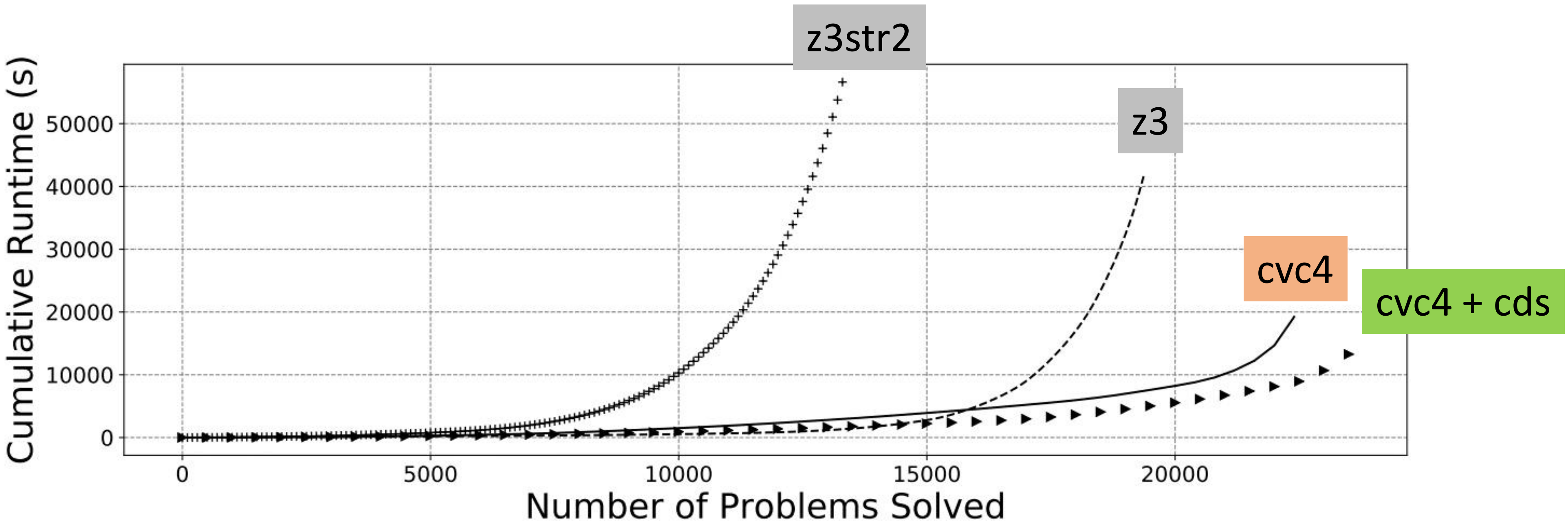
`contains(x·“ac”·y,“b”)` → `contains(x,“b”)∨contains(y,“b”)`

`indexof(“abc”·x,“a”·x,1)` → `-1`

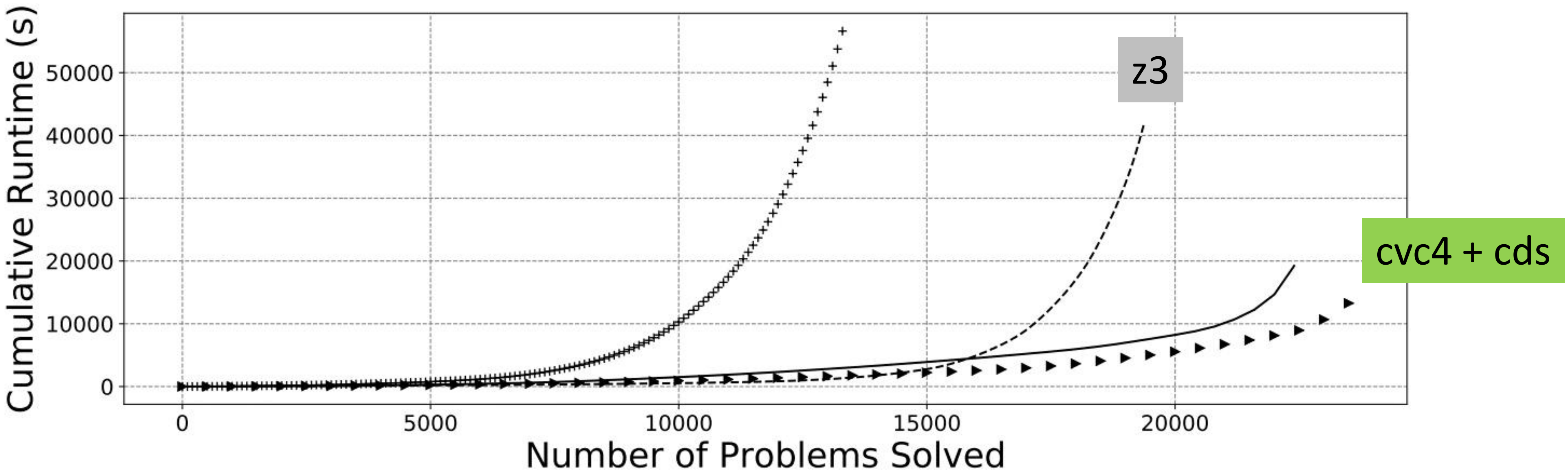
`replace(“a”·x,“b”,y)` → `con(“a”,replace(x,“b”,y))`

- Implemented in 3000+ lines of C++ code **[Reynolds et al CAV2017]**

Results : PyEx Symbolic Execution Benchmarks (25,421)

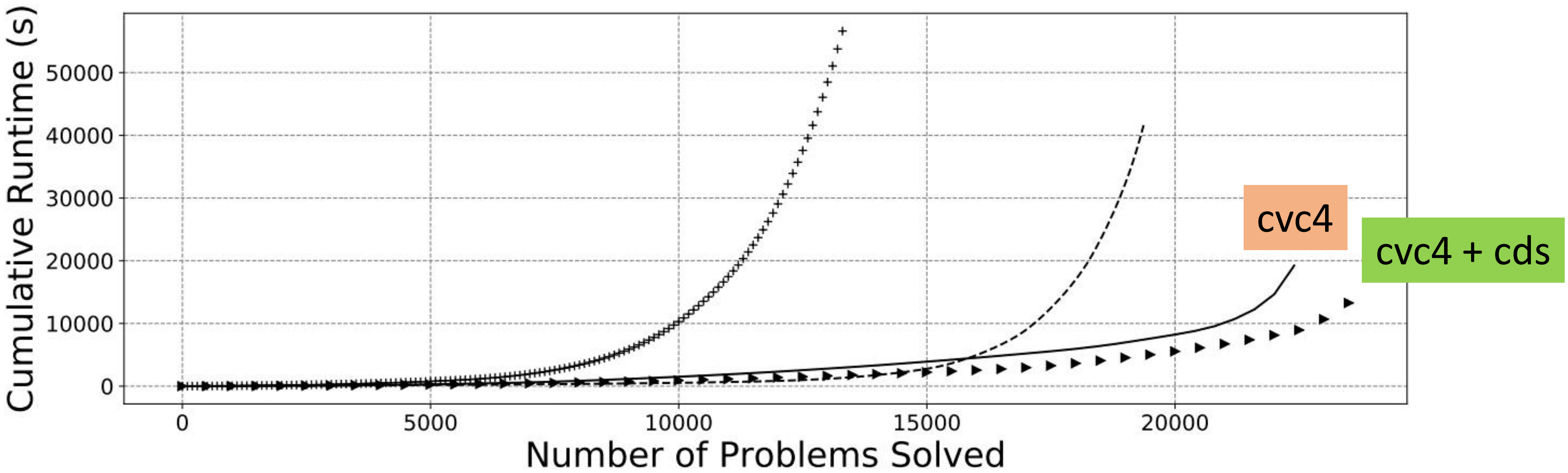


Results : PyEx Symbolic Execution Benchmarks (25,421)



- cvc4+context-dependent simplification solves 23,802 benchmarks in 5h8m
  - Nearest competitor z3 solves 19,368 benchmarks in 11h33m

Results : PyEx Symbolic Execution Benchmarks (25,421)



- By using context-dependent simplification:
  - cvc4+cds solves 536 benchmarks (+582 -46) w.r.t default cvc4
  - cvc4+cds expands 4.2x fewer extended terms per benchmark

Impact on PyEx Symbolic Execution

- Considered regression tests for 4 Python packages:
  - `httplib2`, `pip`, `pymongo`, `requests`
- Tested PyEx symbolic execution using different SMT backends:

Config	Time	Branch Coverage	Line Coverage
PyEx+z3str2	13h49m	3,500	8.34%
PyEx+z3	11h57m	**3,895**	8.41%
PyEx+cvc4	**4h55m**	3,612	**8.48%**

$\Rightarrow$ PyEx+cvc4 achieves comparable program coverage, much faster, wrt other solvers

Theories Not Covered:

- Non-linear Arithmetic **[Franzel et al 2007, Jovanovic/de Moura 2011, Corzilius et al 2012, …]**
- Floating-point **[Brain et al 2012]**
- Co-inductive datatypes (e.g. streams) **[Reynolds/Blanchette 2015]**
- Finite Relations **[Meng et al 2017]**
- Quantifiers $\forall\exists$

  $\Rightarrow$ Focus of the next part

Other Topics not Covered:

- Theory solvers that generate conflict clauses, propagations eagerly
- Optimizations for theory combination
  - e.g. model-based combination
- Other calculi besides DPLL(T) for SMT
  - e.g. mcSat

Applications / Examples

Contract-Based Verification

```
@precondition: P1[ x_in, y_in ]
void f( int& x, int& y )
{
     ...
}
@ensures: P2[ x_in, y_in, x_out, y_out ]
```

Property $\mathbf{P_1}$ should hold for all inputs x_{in}, y_{in} to function f

Property $\mathbf{P_2}$ is guaranteed to hold for x_{out}, y_{out}
(the state of x, y after calling f)

Contract-Based Verification

@precondition: $x_{in}>y_{in}$

```
0 void swap(int& x, int& y)
  {
1       x := x + y;
2       y := x – y;
3       x := x – y;
  }
```

@ensures: $x_{out}=y_{in}$ ė $y_{out}=x_{in}$?

EXAMPLE A1…

Contract-Based Verification

@precondition: $x_{in}>y_{in}$

```
void swap(int& x, int& y)
{
      x := x + y;
      y := x - y;
      x := x - y;
}
```

@ensures: $x_{out}=y_{in}$ ė $y_{out}=x_{in}$

Contract-Based Verification

`@precondition:` $\mathbf{x_{in} > y_{in}}$ — Is this necessary?

```
0  void swap(int& x, int& y)
   {
1       x := x + y;
2       y := x – y;
3       x := x – y;
   }
```

`@ensures:` $x_{out} = y_{in} \wedge y_{out} = x_{in}$

EXAMPLE A1-uc…

Contract-Based Verification

```
  @precondition: x_in>y_in      (struck through)
0 void swap(int& x, int& y)
  {
1       x := x + y;
2       y := x – y;
3       x := x – y;
  }
  @ensures: x_out=y_in ė y_out=x_in
```

Not necessary

x_{in}>y_{in} is not in the unsatisfiable core in the *proof* of **x_{out}=y_{in} ė y_{out}=x_{in}**

Ĝ *precondition is not necessary to show properties of* `swap`

Contract-Based Verification

```
0 void swap(int& x, int& y)
  {
1       x := x + y;
2       y := x - y;
3       x := x - y;
  }
```

@ensures: $x_{out} = y_{in} \wedge y_{out} = x_{in}$

Contract-Based Verification

```
void swap(int& x, int& y)
{
      x := x + y;
      y := x – y;
      x := x – y;
}
```
@ensures: $x_{out}=y_{in} \wedge y_{out}=x_{in}$

```
0 void setMax(int& x, int& y)
  {
1       if( y>x ){
          swap( x, y );
2       }
  }
```
@ensures: $x_{out}>y_{out}$?

EXAMPLE A2…

Contract-Based Verification

```
void swap(int& x, int& y)
{
      x := x + y;
      y := x – y;
      x := x – y;
}
```

@ensures: $x_{out}=y_{in} \wedge y_{out}=x_{in}$

```
0  void setMax(int& x, int& y)
   {
1        if( y>x ){
           swap( x, y );
2        }
   }
```

~~@ensures: $x_{out}>y_{out}$~~

…when $x_{in}=0$ and $y_{in}=0$

Contract-Based Verification

```
void swap(int& x, int& y)
{
      x := x + y;
      y := x – y;
      x := x – y;
}
```

@ensures: $x_{out}=y_{in} \wedge y_{out}=x_{in}$

@precondition: $x_{in} \neq y_{in}$

```
0 void setMax(int& x, int& y)
  {
1       if( y>x ){
          swap( x, y );
2       }
  }
```

@ensures: $x_{out} > y_{out}$

Contract-Based Verification

```
  @precondition: x_in>5
0 void resetX(int& x, int& y)
  {
1      if( x*y==3 ){
         x=-1;
       }
2 }
  @ensures: x_out>5 ?
```

EXAMPLE A3…

Contract-Based Verification

```
  @precondition: x_in>5
0 void resetX(int& x, int& y)
  {
1      if( x*y==3 ){
         x=-1;
       }
2 }
  @ensures: x_out>5
```

…using heuristic techniques for non-linear arithmetic
(incomplete, can get lucky)

Contract-Based Verification

@precondition: res_{in}==0

```
0 void cubes(int a, int b, int c, int& res)
  {
1       if( (a*a*a)+(b*b*b)+(c*c*c)==33 ){
          res = 1;
        }
2 }
```

@ensures: res_{out}==0 ?

EXAMPLE A4…

Contract-Based Verification

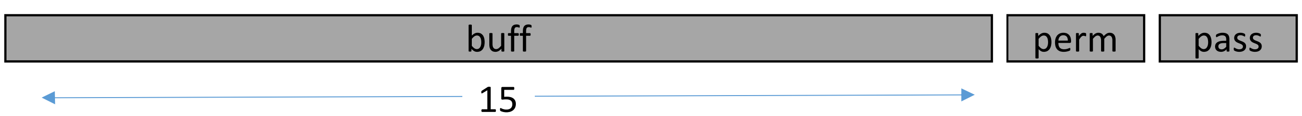
```
  @precondition: res_in==0
0 void cubes(int a, int b, int c, int& res)
  {
1      if( (a*a*a)+(b*b*b)+(c*c*c)==33 ){
         res = 1;
       }
2 }
  @ensures: res_out==0 ?
```

…the SMT solver will (typically) not solve open problems in mathematics!

Symbolic Execution

```
   char buff[15];
   char permission = ‘N’;
   char pass = ‘N’;
   cout << "Enter the password :";
0  gets(input);
1  if(strcmp(buff, “PASSWORD")) {
       cout << "Correct Password";
       permission= ’Y’;
       pass = ’Y’;
   } else {
       cout << "Wrong Password";
       pass= ‘N’;
   }
2  if(permission == ’Y’) {  //grant root access
     Assert(pass=‘Y’ );
   }
```

Memory:



EXAMPLE A5…

Symbolic Execution

```
  char buff[15];
  char permission = 'N';
  char pass = 'N';
  cout << "Enter the password :";
0 gets(input);
1 if(strcmp(buff, "PASSWORD")) {
      cout << "Correct Password";
      permission= 'Y';
      pass = 'Y';
  } else {
      cout << "Wrong Password";
      pass= 'N';
  }
2 if(permission == 'Y') {  //grant root access
    Assert(pass='Y' );
  }
```

Symbolic Execution

```
  char buff[15];
  char permission = ‘N’;
  char pass = ‘N’;
  cout << "Enter the password :";
0 gets(input);
1 if(strcmp(buff, “PASSWORD")) {
      cout << "Correct Password";
      permission= ’Y’;
      pass = ’Y’;
  } else {
      cout << "Wrong Password";
      pass= ‘N’;
  }
2 if(permission == ’Y’) {  //grant root access
    Assert(pass=‘Y’ );
  }
```

Symbolic Execution

```
  char buff[15];
  char permission = ‘N’;
  char pass = ‘N’;
  cout << "Enter the password :";
0 gets(input);
1 if(strcmp(buff, “PASSWORD")) {
      cout << "Correct Password";
      permission= ’Y’;
      pass = ’Y’;
  } else {
      cout << "Wrong Password";
      pass= ‘N’;
  }
2 if(permission == ’Y’) {  //grant root access
    Assert(pass=‘Y’ );
  }
```

EXAMPLE A5-inc…

Symbolic Execution

```
char buff[15];
char permission = 'N';
char pass = 'N';
cout << "Enter the password :";
gets(input);  ← "AAAAAAAAAAAAAAAY"
if(strcmp(buff, "PASSWORD")) {
    cout << "Correct Password";
    permission= 'Y';
    pass = 'Y';
} else {
    cout << "Wrong Password";
    pass= 'N';
}
if(permission == 'Y') {  //grant root access
  Assert(pass='Y' );  ← pass=="N"
}
```

Challenge Exercise

```
cout << "Enter text to print :";
string input;
gets(input);
string data= ‘p‘ ++ input ++ ‘;xcvc4;‘;
int index=0;
while(index<str.len(data)){
  int end=indexof(data,’;’,index+1);
  string cmd=substr(data,index+1,end-(index+1));
  if(cmd!=“”){
    if(substr(data,index,1)=‘p’){
      cout << curr_cmd << endl;
    }else if(substr(data,index,1)=‘x’){
      exec(cmd);
      Assert(cmd==“cvc4”);
    }else{
      cout << “Bad command” << endl;
    }
  }
  index=end+1;
}
```

EXAMPLE A6…

Challenge Exercise

```
cout << "Enter text to print :";
string input;
gets(input);
string data= ‘p‘ ++ input ++ ‘;xcvc4;‘;
int index=0;
while(index<str.len(data)){
  int end=indexof(data,’;’,index+1);
  string cmd=substr(data,index+1,end-(index+1));
  if(cmd!=“”){
    if(substr(data,index,1)=‘p’){
      cout << curr_cmd << endl;
    }else if(substr(data,index,1)=‘x’){
      exec(cmd);
      Assert(cmd==“cvc4”);
    }else{
      cout << “Bad command” << endl;
    }
  }
  index=end+1;
}
```

Expects data to be a string like:
p[TEXT1];x[PROG1];x[PROG2];p[TEXT2];…

Challenge Exercise

```
cout << "Enter text to print :";
string input;
gets(input);  ← ";xATTACK"
string data= 'p' ++ input ++ ';xcvc4;';
int index=0;
while(index<str.len(data)){
  int end=indexof(data,';',index+1);
  string cmd=substr(data,index+1,end-(index+1));
  if(cmd!=""){
    if(substr(data,index,1)='p'){
      cout << curr_cmd << endl;
    }else if(substr(data,index,1)='x'){
      exec(cmd);  ← exec("ATTACK")
      Assert(cmd=="cvc4");
    }else{
      cout << "Bad command" << endl;
    }
  }
  index=end+1;
}
```

Challenge Exercise #2

```
cout << "Enter text to print :";
string input;
char setup='N';
gets(input);
string data= 'xsetup;xroot;p' ++ input ++ ';';
int index=13;
while(0≤index<str.len(data)){
  int end=indexof(data,';',index+1);
  string cmd=substr(data,index+1,end-(index+1));
  runCmd( cmd, setup, index );
  index=end+1;
}
```

```
void runCmd( string cmd, char& setup, int& index ){
  if(substr(data,index,1)='p'){
    cout << curr_cmd << endl;
  }else if(substr(data,index,1)='g'){
    index := str.to.int(index);
  }else if(substr(data,index,1)='x'){
    exec(cmd);
    if(cmd=="setup"){
      setup='Y';
    }
    Assert(cmd=="root" Ĝ setup='Y');
  }else{
    cout << "Bad command" << endl;
  }
}
```

start index
↓
'xsetup;xroot;p' ++ input ++ ';';

EXAMPLE A7…

Challenge Exercise #2

```
cout << "Enter text to print :";
string input;
char setup='N';
gets(input);
if(contains(input,'x')){
  exit();
}
string data= 'xsetup;xroot;p' ++ input ++ ';';
int index=13;
while(0≤index<str.len(data)){
  int end=indexof(data,';',index+1);
  string cmd=substr(data,index+1,end-(index+1));
  runCmd( cmd, setup, index );
  index=end+1;
}
```

```
void runCmd( string cmd, char& setup, int& index ){
  if(substr(data,index,1)='p'){
    cout << curr_cmd << endl;
  }else if(substr(data,index,1)='g'){
    index := str.to.int(index);
  }else if(substr(data,index,1)='x'){
    exec(cmd);
    if(cmd=="setup"){
      setup='Y';
    }
    Assert(cmd=="root" Ĝ setup='Y');
  }else{
    cout << "Bad command" << endl;
  }
}
```

Challenge Exercise #2

```
cout << "Enter text to print :";
string input;
char setup='N';
gets(input);
if(contains(input,'x')){
  exit();
}
string data= 'xsetup;xroot;p' ++ input ++ ';';
int index=13;
while(0≤index<str.len(data)){
  int end=indexof(data,';',index+1);
  string cmd=substr(data,index+1,end-(index+1));
  runCmd( cmd, setup, index );
  index=end+1;
}
```

```
void runCmd( string cmd, char& setup, int& index ){
  if(substr(data,index,1)='p'){
    cout << curr_cmd << endl;
  }else if(substr(data,index,1)='g'){
    index := str.to.int(index);
  }else if(substr(data,index,1)='x'){
    exec(cmd);
    if(cmd=="setup"){
      setup='Y';
    }
    Assert(cmd=="root" Ĝ setup='Y');
  }else{
    cout << "Bad command" << endl;
  }
}
```

Expects data to be a string like:
p[TEXT1];x[PROG1];g[ADDRESS1];p[TEXT2];…

Challenge Exercise #2

```
cout << "Enter text to print :";
string input;    ← “;g7”
char setup=‘N’;
gets(input);
if(contains(input,’x’)){
  exit();
}
string data= ‘xsetup;xroot;p‘ ++ input ++ ‘;‘;
int index=13;
while(0≤index<str.len(data)){
  int end=indexof(data,’;’,index+1);
  string cmd=substr(data,index+1,end-(index+1));
  runCmd( cmd, setup, index );
  index=end+1;
}
```

```
void runCmd( string cmd, char& setup, int& index ){
  if(substr(data,index,1)=‘p’){
    cout << curr_cmd << endl;
  }else if(substr(data,index,1)=‘g’){
    index := str.to.int(index);
  }else if(substr(data,index,1)=‘x’){
    exec(cmd);
    if(cmd==“setup”){
      setup=‘Y’;
    }
    Assert(cmd==“root” Ĝ setup=‘Y’);
  }else{
    cout << “Bad command” << endl;
  }
}
```

‘xsetup;xroot;p;g7;’;