SMT String Solving in CVC4

Andrew Reynolds & Cesare Tinelli
The University of Iowa

MOSCA 2019
May 6, 2019

Satisfiability Modulo Theories (SMT) Solvers

Many applications:

- Software verification
- Automated theorem proving
- Symbolic execution
- Security analysis

In this talk:

- How SMT Solvers (CVC4) handle string constraints

The CVC4 SMT Solver

Support for many theories and features

- UF, (non)linear arithmetic, arrays
- Bit-vectors, floating point
- Finite sets and relations, (co)datatypes

$\Rightarrow$ Strings and regular expressions

Co-developed at Stanford and University of Iowa

- Project Leaders:

  Clark Barrett and Cesare Tinelli
- String solver developers:

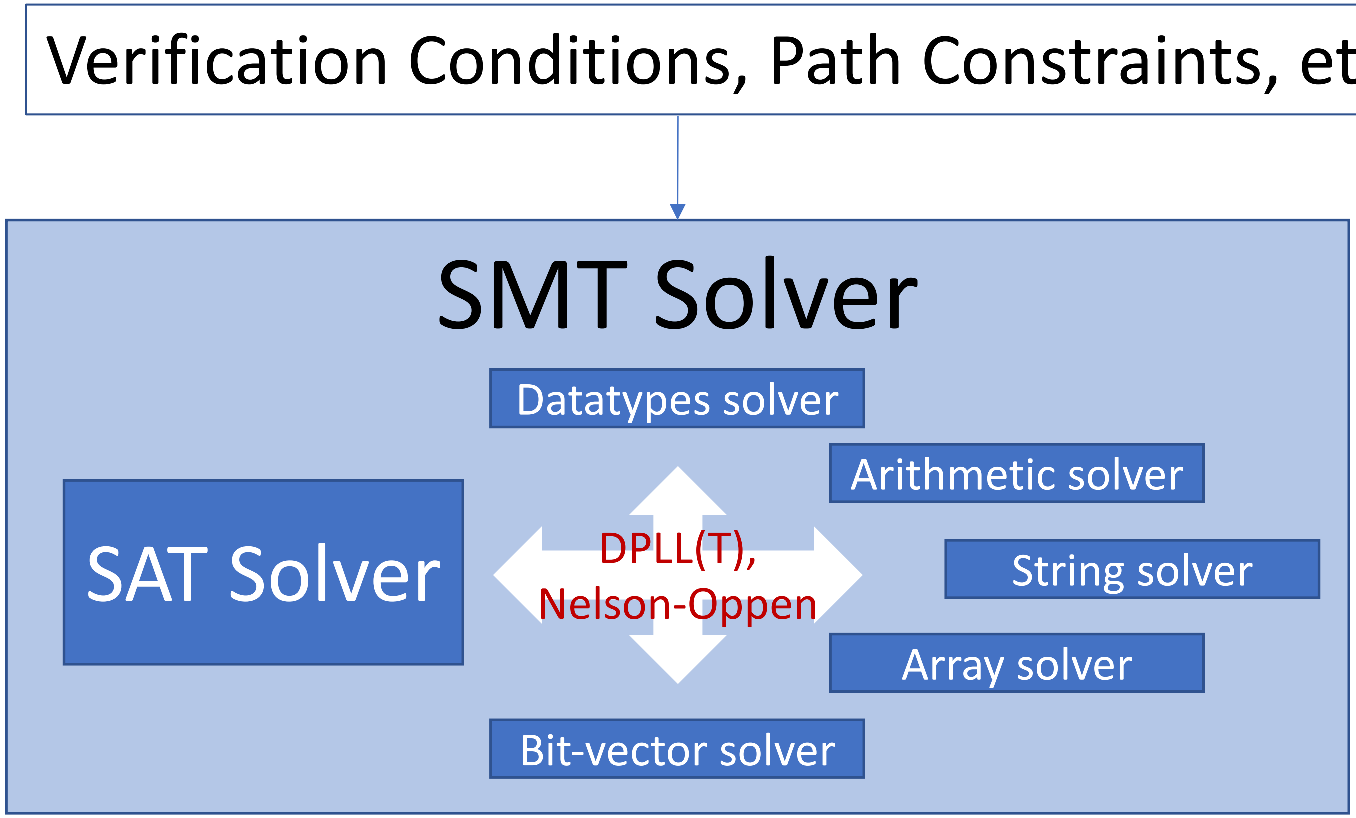
  Andrew Reynolds, Tianyi Liang, Nestan Tsiskaridze, Andres Noetzli

Overview

- How SMT string solvers work:
  - Basic architecture (DPLL(T))
  - Core Theory Solver for Word Equations with Length Constraints
  - Advanced Features
    - Finite model finding
    - Context-dependent simplification for extended string constraints
    - Regular expression elimination

SMT solvers

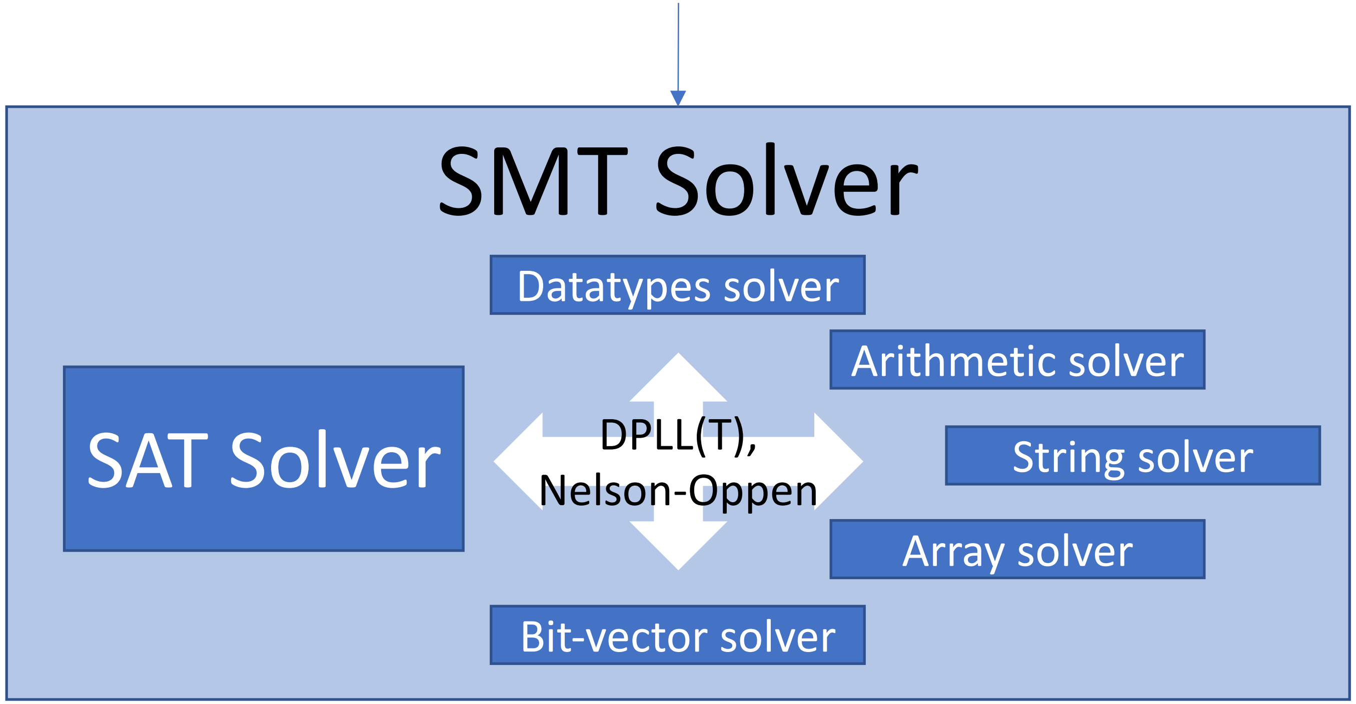
Efficient tools for satisfiability *modulo theories*



SMT solvers

Efficient tools for satisfiability *modulo theories*

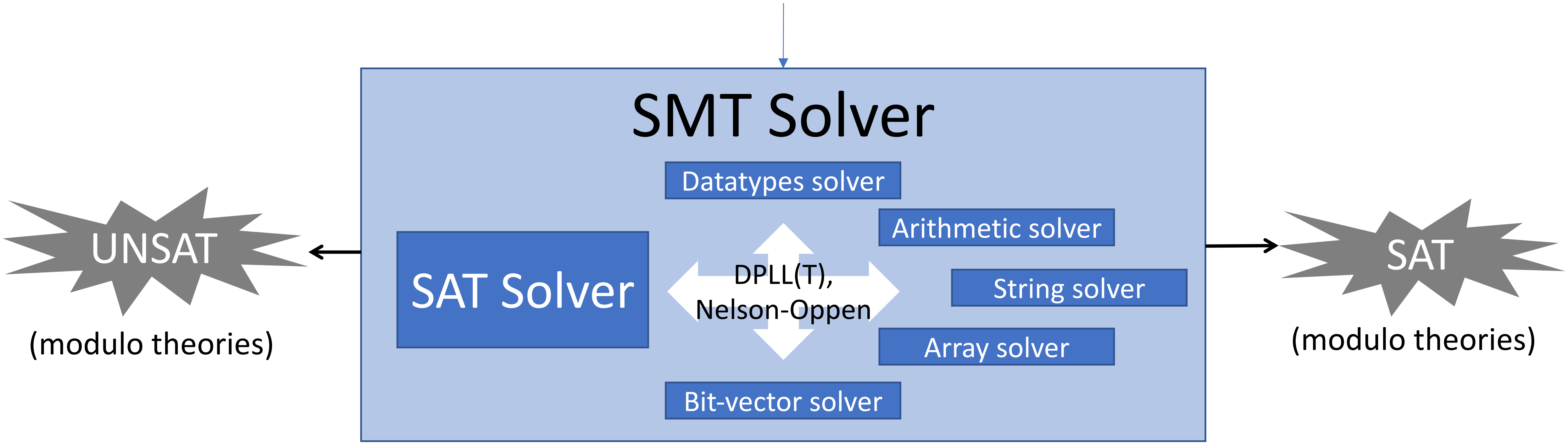
$$(A[x] + B[x] > 0 \vee x + y > 0) \wedge (\text{cons}(\text{“abc”}, d_1) \neq d_2 \vee x < 0)$$



SMT solvers

Efficient tools for satisfiability *modulo theories*

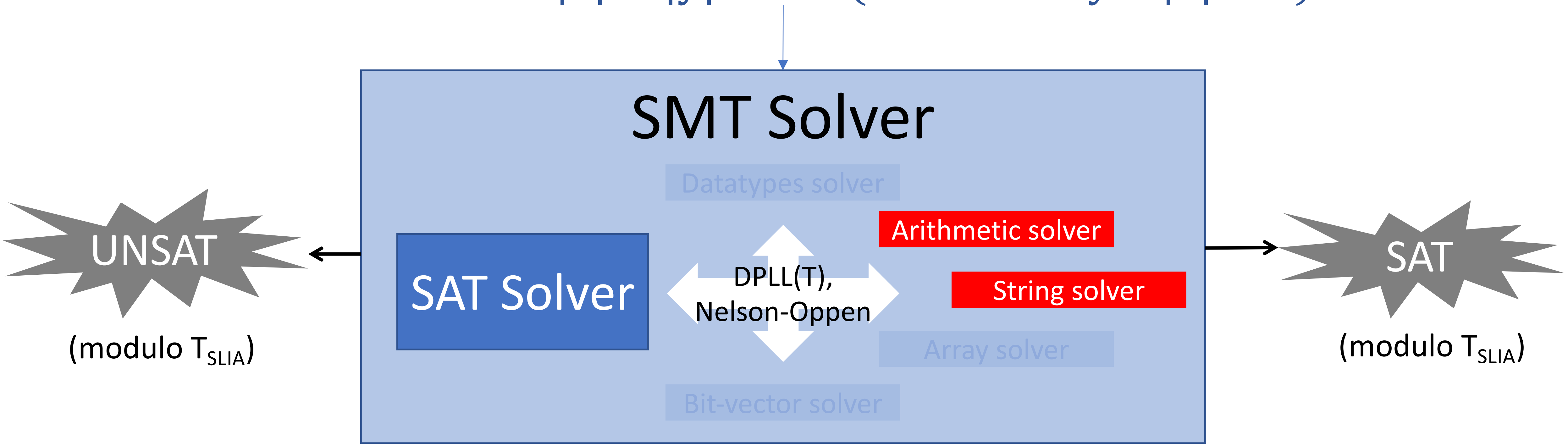
$$(A[x] + B[x] > 0 \vee x + y > 0) \wedge (cons(\text{“abc”}, d_1) \neq d_2 \vee x < 0)$$



SMT solvers

Our focus: the theory of strings and linear arithmetic T_{SLIA}

$$x = \text{"ab"} \cdot z \wedge |x| + |y| \leq 5 \wedge (\text{"abcd"} \cdot x = y \vee |x| > 5)$$



Theory of Strings + Linear Arithmetic (T_{SLIA})

Sorts:

- Integers `Int`
- Strings `String`, interpreted as A^* for finite alphabet A

Terms:

- String Variables: x , y , z
- Integer Variables: i, j, k
- String Constants: “”, “abc” , “AAAAA” , “http”
- String Concatenation: $x \cdot$“abc”, $x \cdot y \cdot z \cdot w$
- String Length: $|x|$

Formulas are:

- Equalities and disequalities between string terms
- *Linear* arithmetic constraints: $|x| + 4 > |y|$

Example: $x \cdot \text{“a”} = y,\ y \neq \text{”b”} \cdot z,\ |y| > |x| + 2$

Decidability: unknown, regardless, many problems can be solved efficiently in practice

T_{SLIA} String Solver for DPLL(T)

Achieved as a Cooperation between:

SAT Solver

Arithmetic Solver

String Solver

T_{SLIA} String Solver for DPLL(T)

$x = \text{“ab”} \cdot z$
$|x| + |y| \leq 5$
$\text{“abcd”} \cdot x = y \vee |x| > 5$

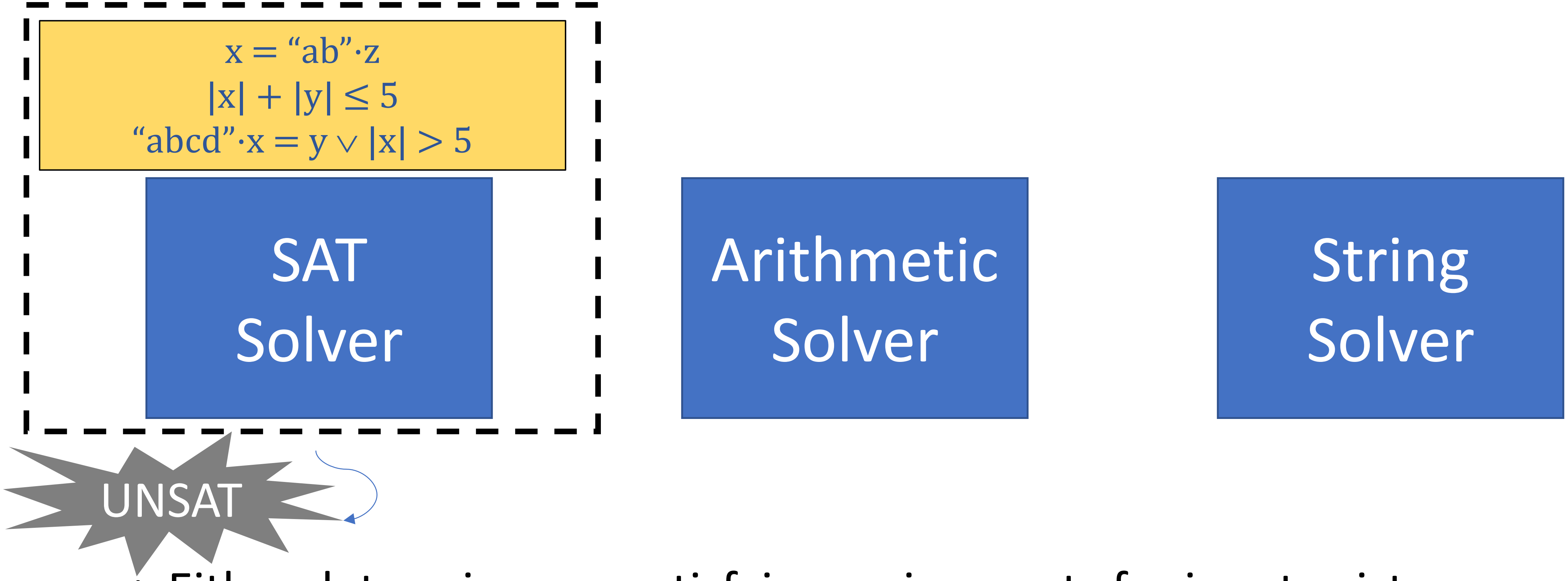
Set of T_{SLIA}-formulas in clausal normal form (CNF)

SAT Solver

Arithmetic Solver

String Solver

T_{SLIA} String Solver for DPLL(T)



⇒ Either determines no satisfying assignments for input exist

T_{SLIA} String Solver for DPLL(T)

$x = \text{“ab”} \cdot z$

$|x| + |y| \leq 5$

$\text{“abcd”} \cdot x = y \lor |x| > 5$

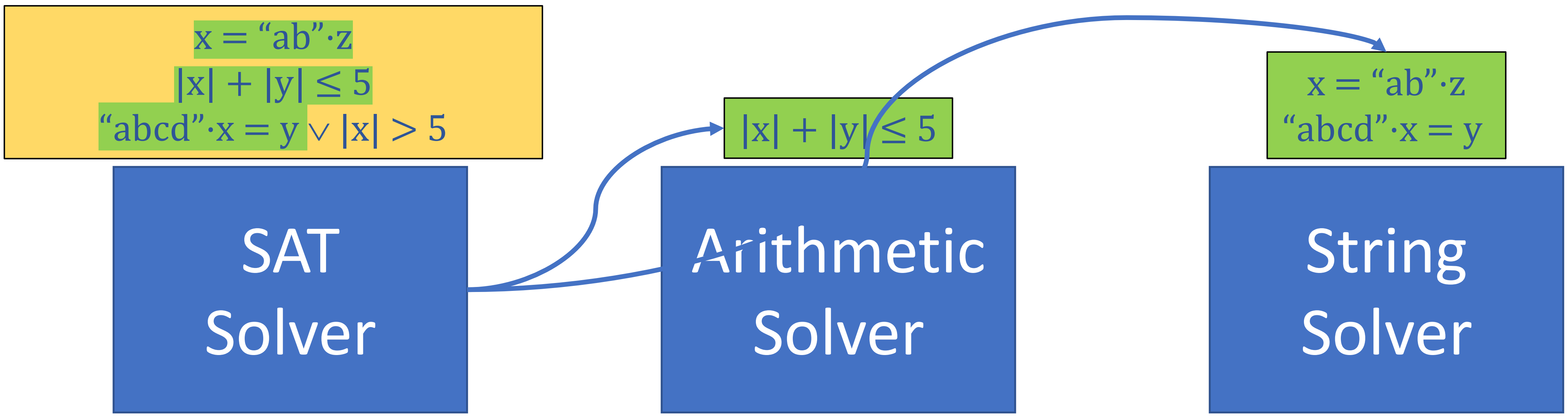
SAT Solver

Arithmetic Solver

String Solver

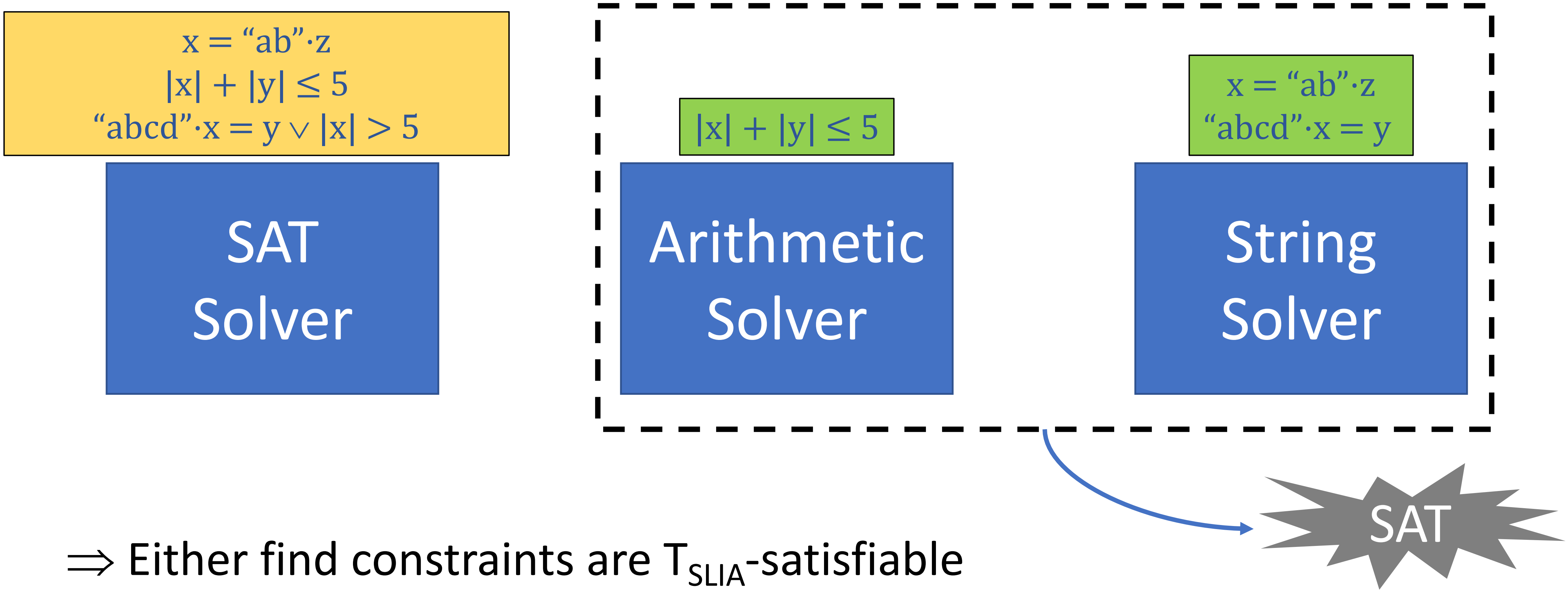
$\Rightarrow$ … or returns a propositionally satisfying assignment

T_{SLIA} String Solver for DPLL(T)



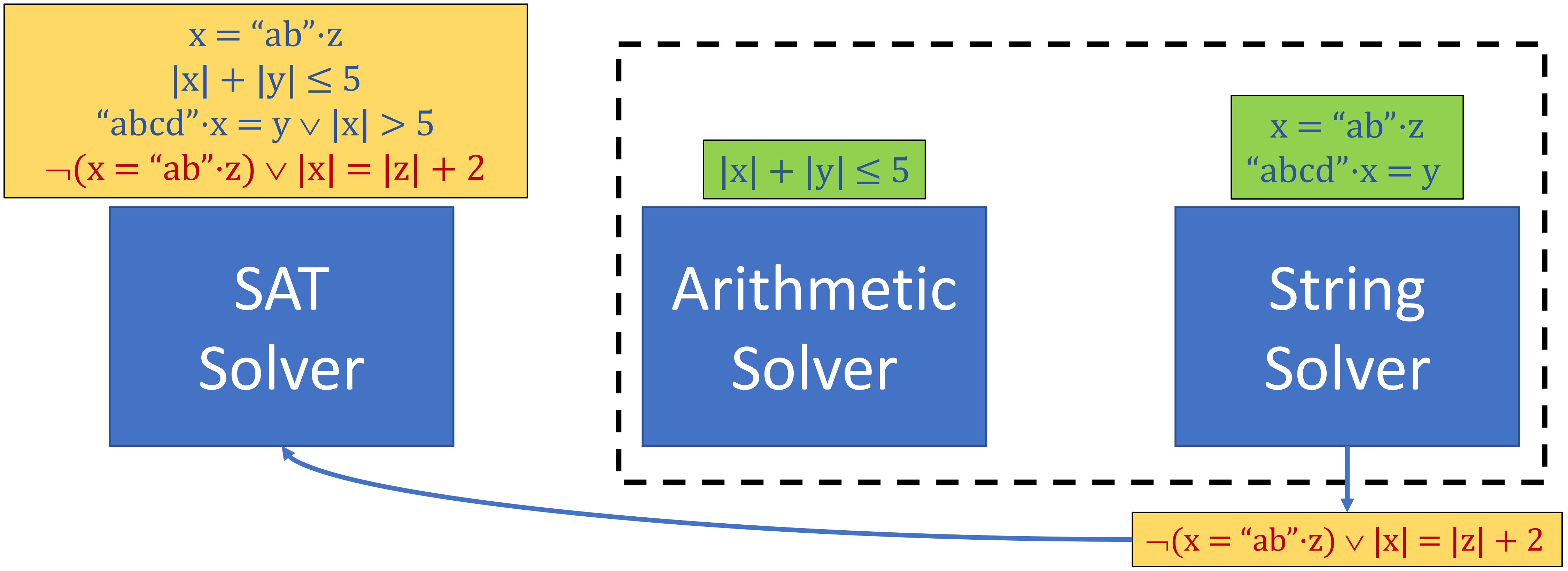
⇒ Constraints distributed to arithmetic and string solvers

T_{SLIA} String Solver for DPLL(T)



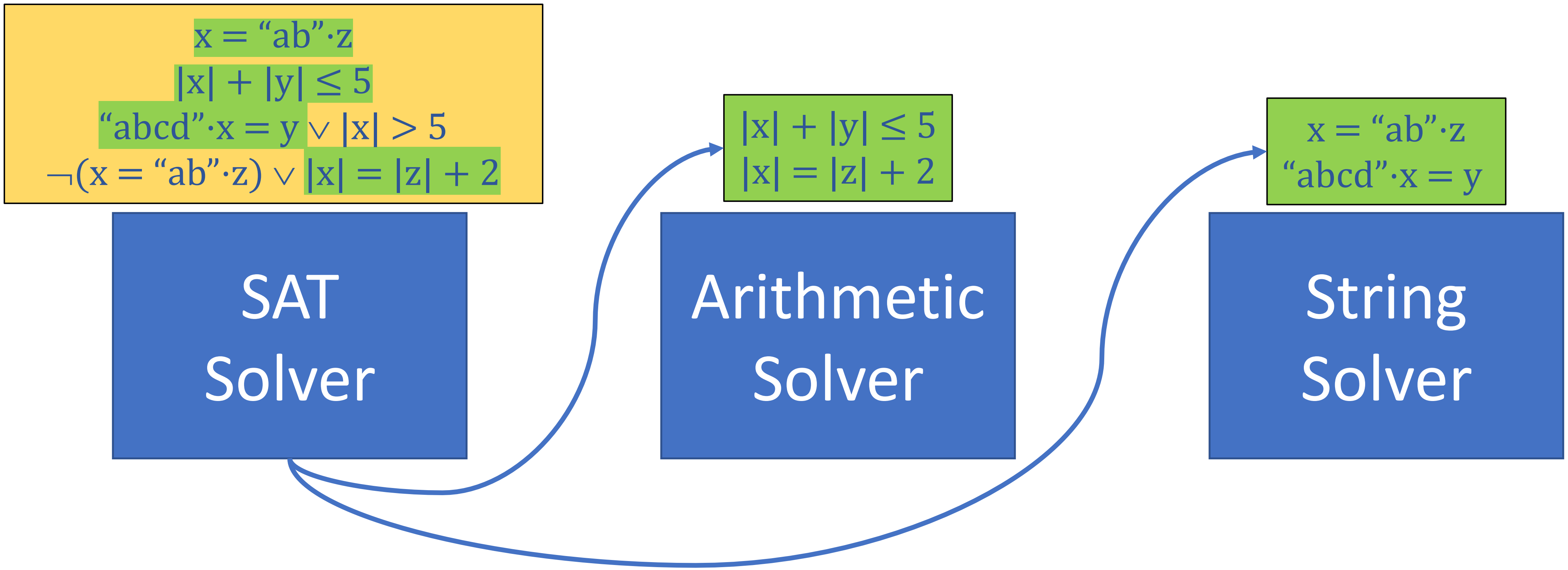
⇒ Either find constraints are T_{SLIA}-satisfiable

T_{SLIA} String Solver for DPLL(T)



⇒ or return *theory lemmas* (valid T_{LIA}/T_S-formulas) to SAT solver

T_{SLIA} String Solver for DPLL(T)



$\Rightarrow$ and repeat

Inside a DPLL(T) Theory Solver

Given a set of T-literals M_T,

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

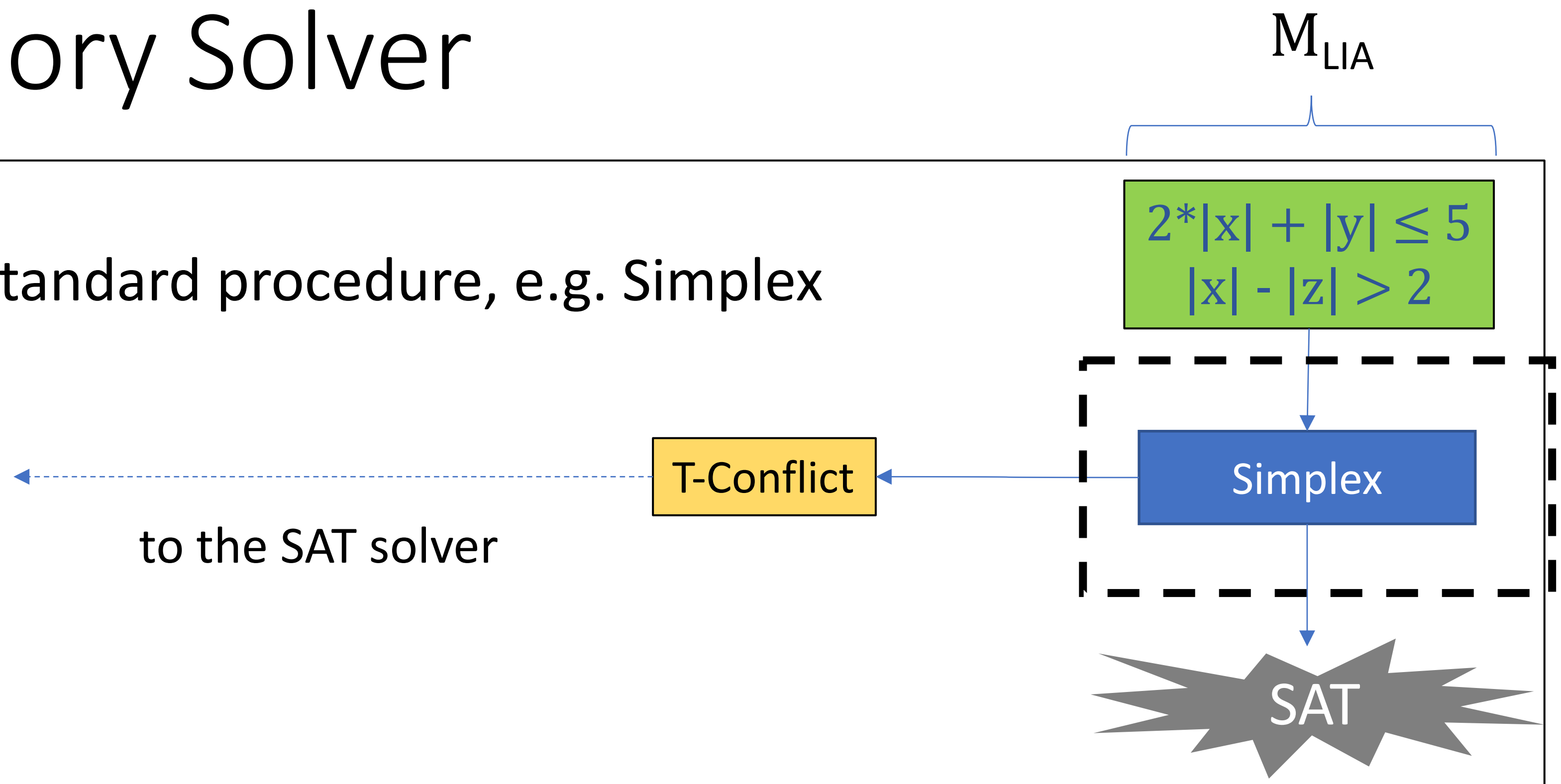
Should the solver send a theory lemma to the SAT solver?

- no => return unknown, or
  return a *model* (a satisfying assignment)
- yes => which lemma?
  - In typical DPLL(T) theory solvers (e.g. LIA) theory lemmas ⇔ *T-conflicts* $\neg(L_1 \wedge \ldots \wedge L_n)$ for some T-unsatisfiable $\{L_1, \ldots, L_n\} \subseteq M_T$
  - In string solver, theory lemmas may introduce new literals
  - Will describe a *strategy* for strings

Arithmetic Theory Solver

Decision procedure:

T-conflicts based on a standard procedure, e.g. Simplex



Properties:

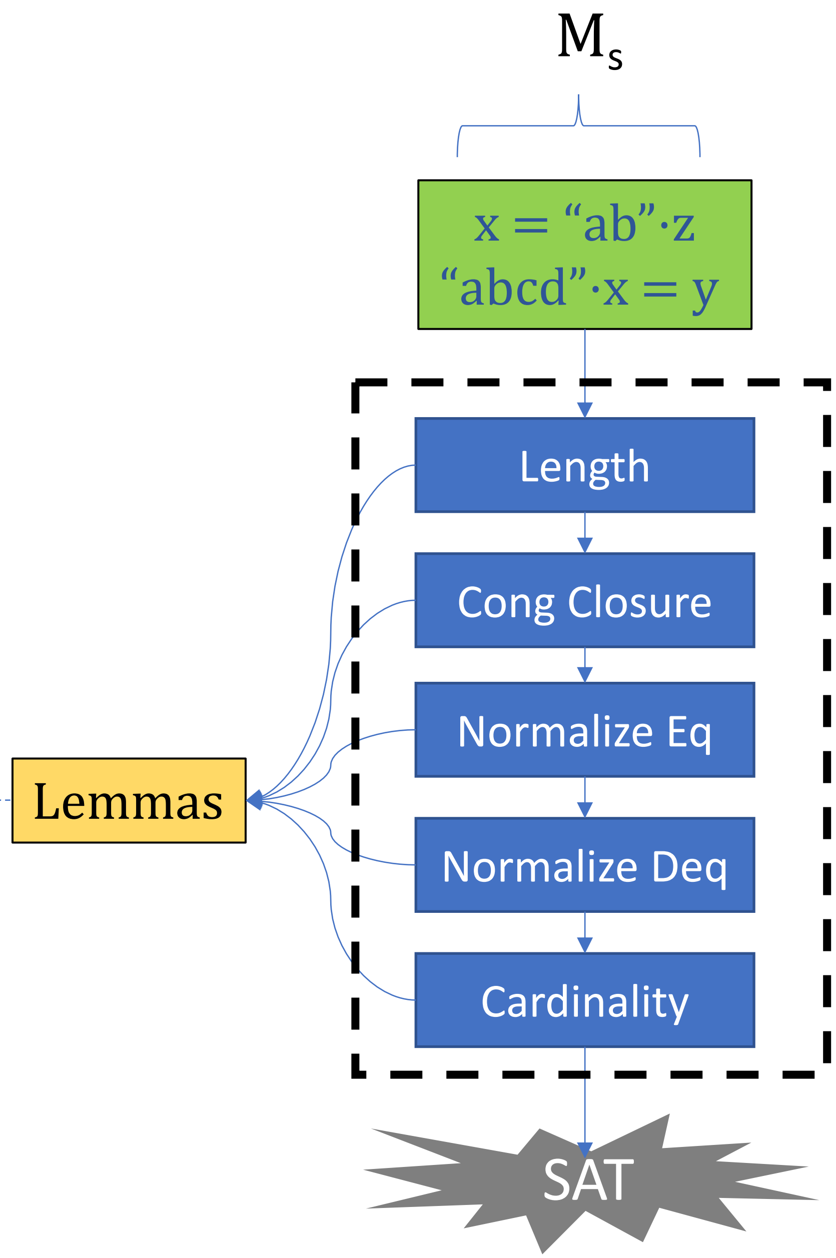
- Sound, lemmas it generates are LIA-valid
- Model-sound, "SAT" can be trusted
- Terminating, in the context of DPLL(T)
  Only generates finitely many lemmas

∴ Complete

String Theory Solver

Inference strategy:

1. Process length constraints
2. Check for equality conflicts (congruence closure)
3. Normalize string equalities
4. Normalize string disequalities
5. Check cardinality constraints

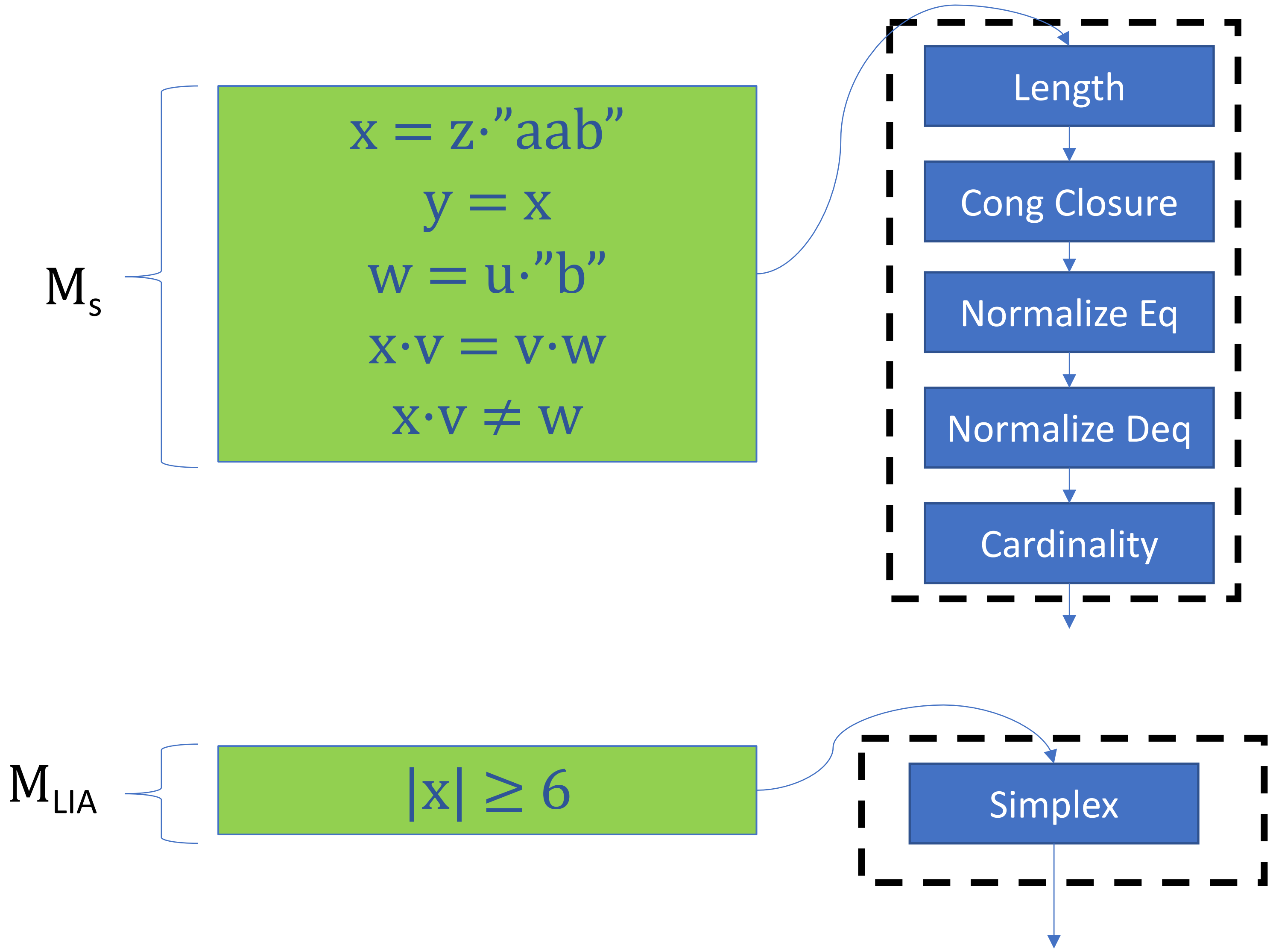


Properties:

- Sound, lemmas it generates are T_s-valid
- Model-sound, "SAT" can be trusted
- Non-terminating, in the context of DPLL(T)
  - May generate infinitely many lemmas

String Solver

Running example:

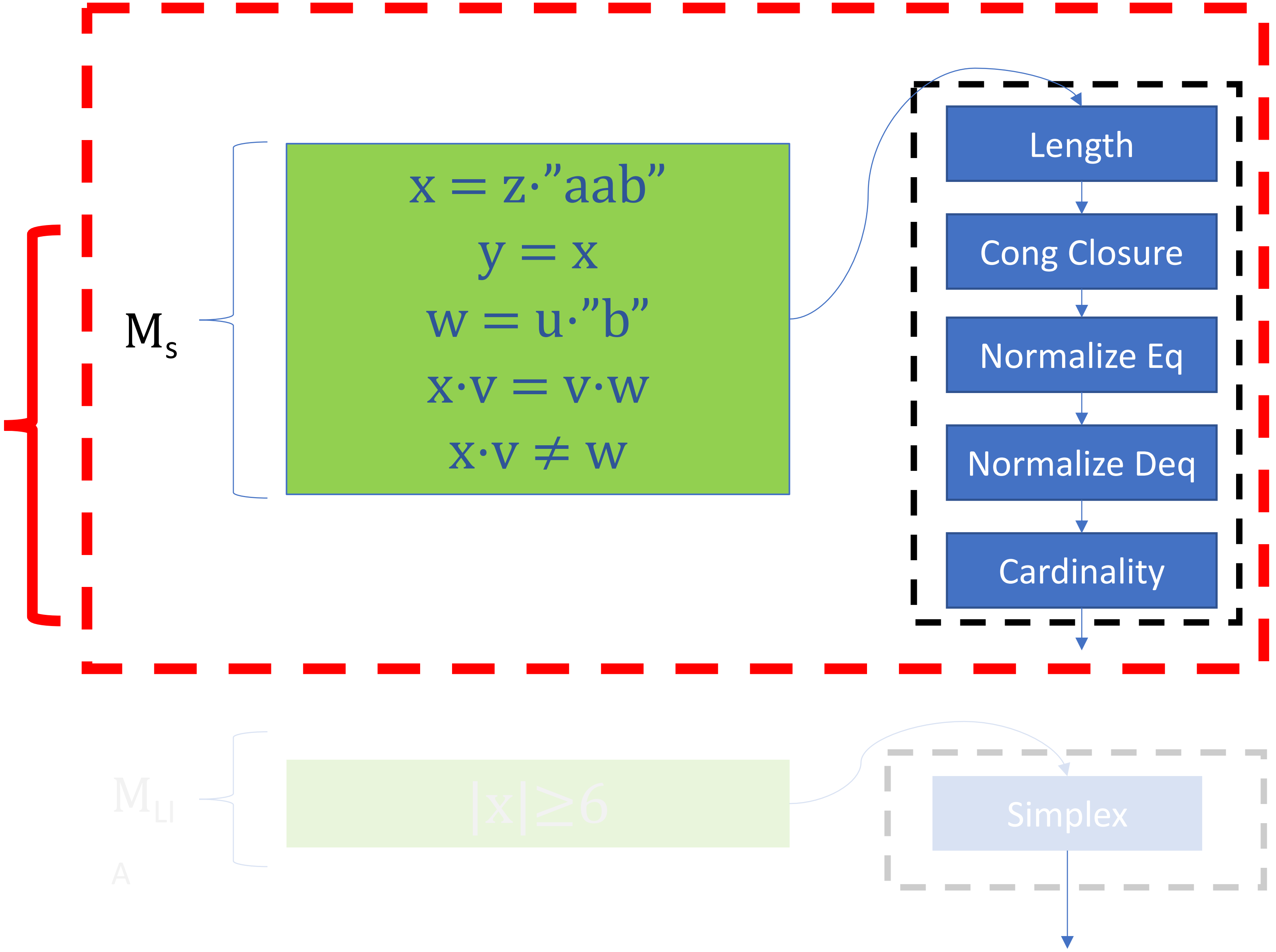


String Solver

Running example:

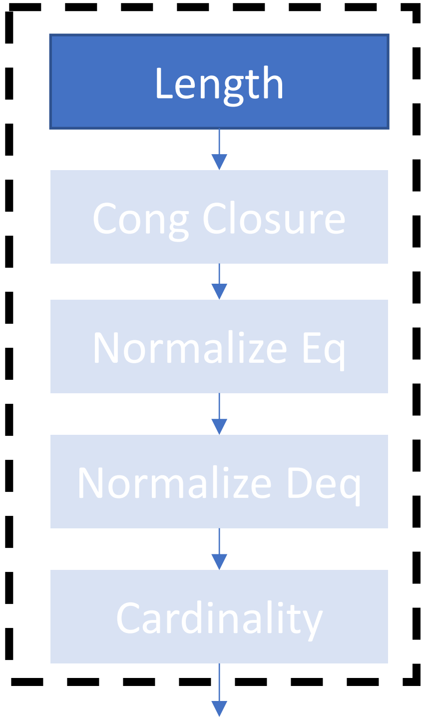
Will focus on string solver

[Liang et al. CAV2014]



String Solver: Process Length

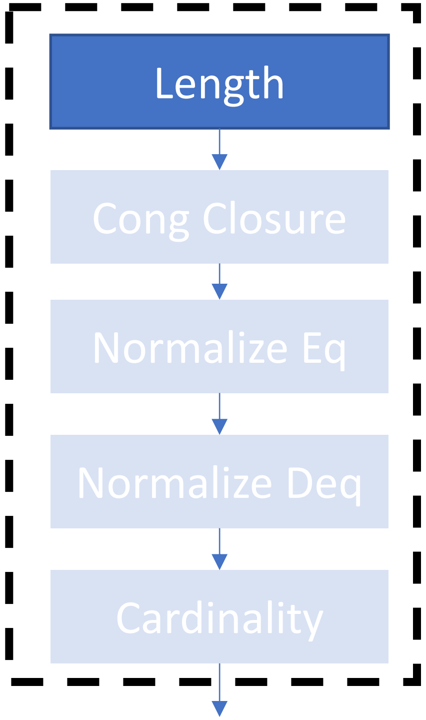
$$M_s \left\{ \begin{array}{c} x = z\cdot\text{"aab"} \\ y = x \\ w = u\cdot\text{"b"} \\ x\cdot v = v\cdot w \\ x\cdot v \neq w \end{array} \right.$$



String Solver: Process Length

M_s:

$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



- For each term of type string in M_s:

  returns a lemma giving the definition of its length:

  $|\text{"b"}| = 1$ $\quad$ $|\text{"aab"}| = 3$ $\quad$ $|x \cdot v| = |x| + |v|$

  $|z \cdot \text{"aab"}| = |z| + 3$ $\quad$ $|u \cdot \text{"b"}| = |u| + 1$ $\quad$ $|v \cdot w| = |v| + |w|$

- For each variable of type string in M_s:

  returns an emptiness splitting lemma:

  $x = \text{""} \vee |x| \geq 1$ $\quad$ $y = \text{""} \vee |y| \geq 1$ $\quad$ …

String Solver: Process Length

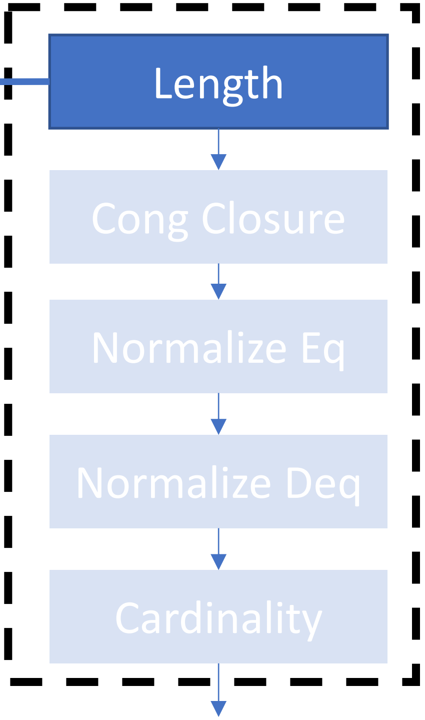
Lemmas

M_s

$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v = v\cdot w$$
$$x\cdot v \neq w$$

M_{LIA}

$$|x| \geq 6$$



String Solver: Process Length

SAT Solver

new propositional assignment

M_s

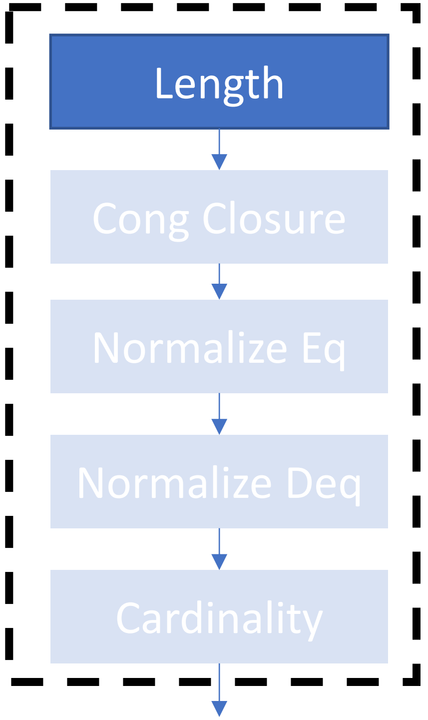
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$

M_{LIA}

$$|x| \geq 6$$
$$|\text{"b"}| = 1$$
$$|\text{"aab"}| = 3$$
$$|x \cdot v| = |x| + |v|$$
$$|z \cdot \text{"aab"}| = |z| + 3$$
$$|u \cdot \text{"b"}| = |u| + 3$$
$$|v \cdot w| = |v| + |w|$$
$$|x| \geq 1$$
...

UNSAT?

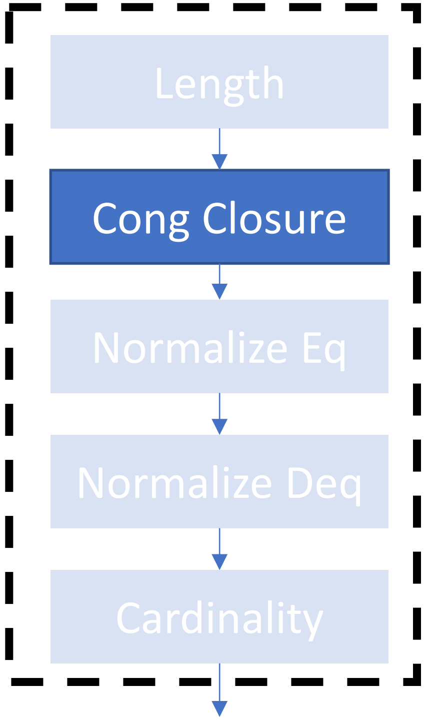
adds new constraints in arithmetic solver



String Solver: Congruence Closure

M_s

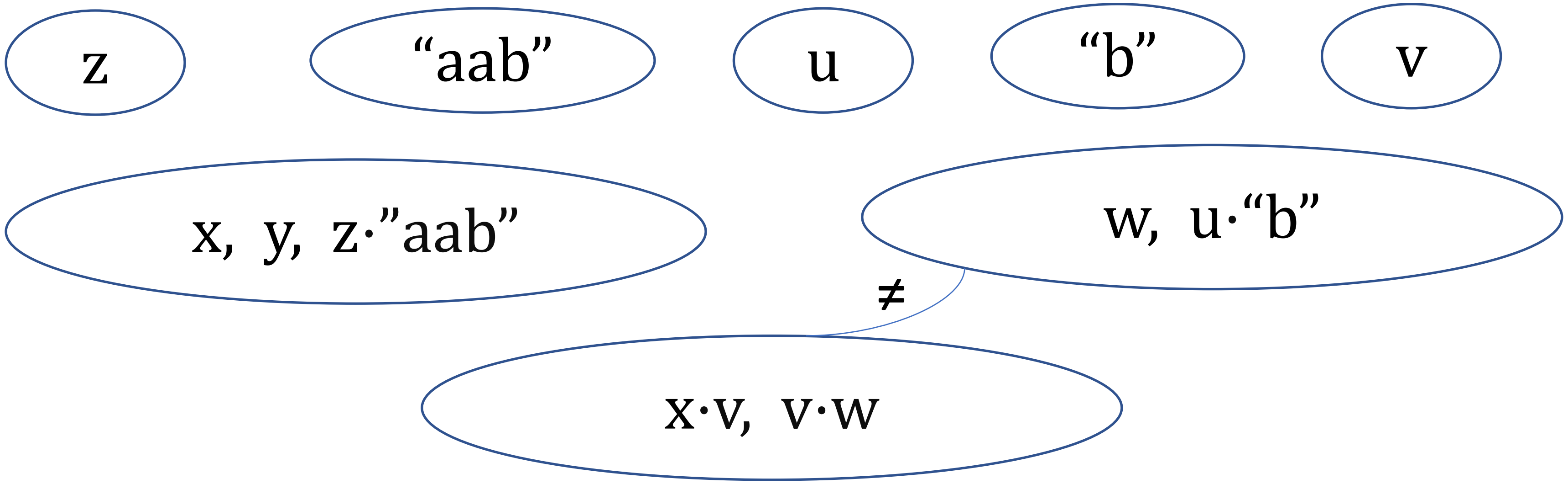
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$

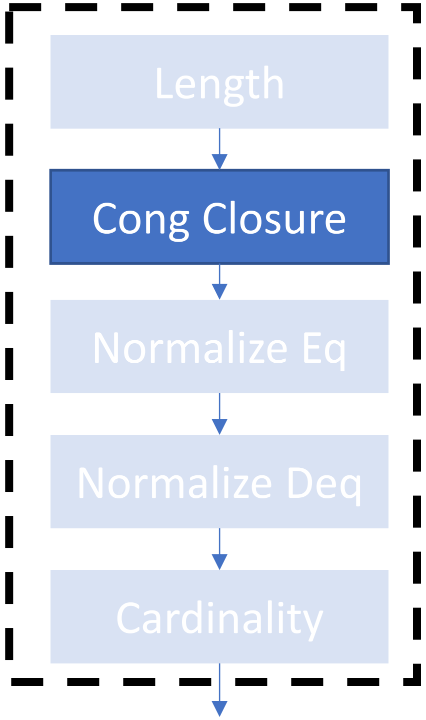


String Solver: Congruence Closure

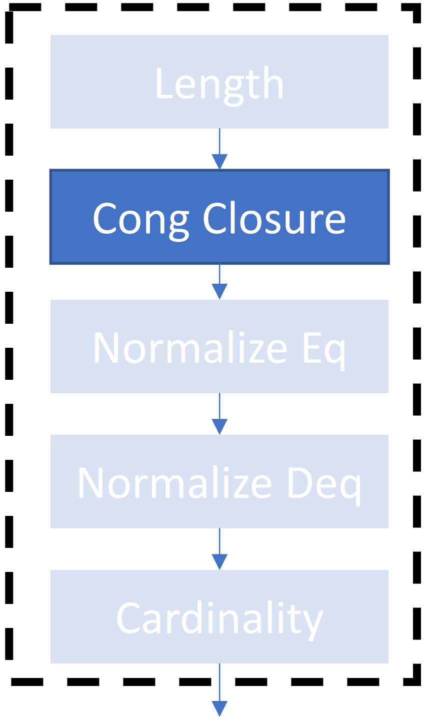
$$M_s \left\{ \begin{array}{c} x = z\cdot\text{"aab"} \\ y = x \\ w = u\cdot\text{"b"} \\ x\cdot v = v\cdot w \\ x\cdot v \neq w \end{array} \right.$$

- Group terms by *equivalence classes*:





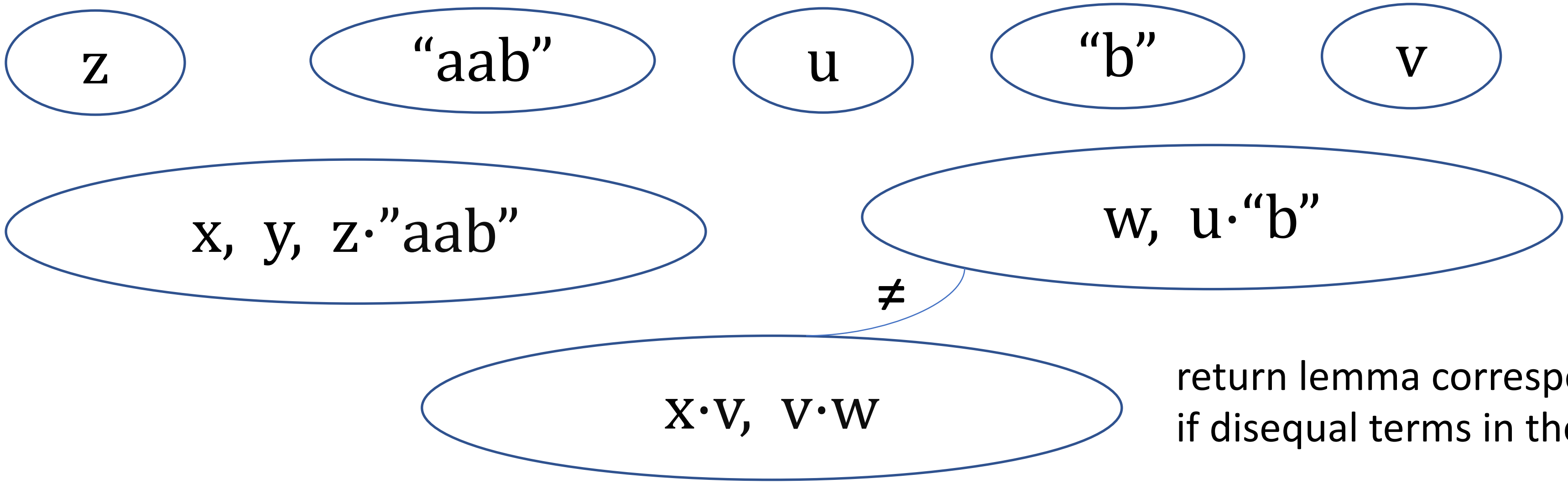
String Solver: Congruence Closure



M_s:

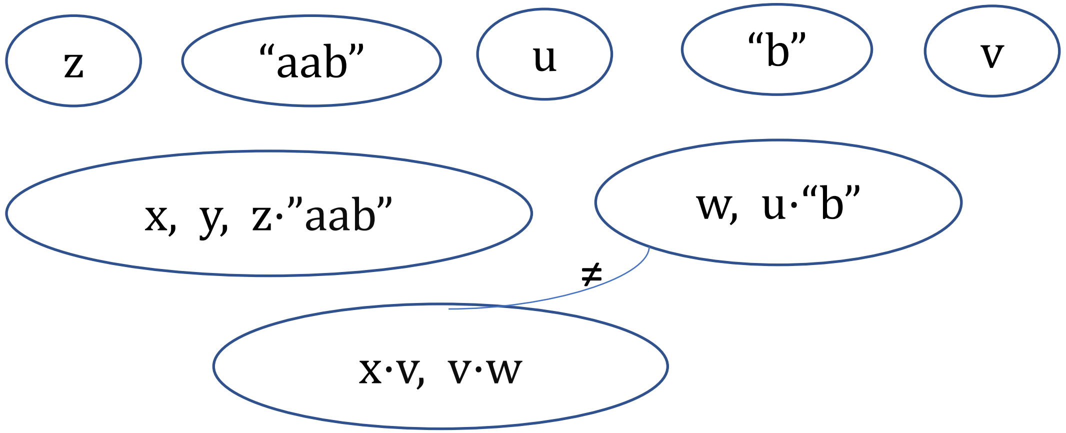
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$

- Group terms by *equivalence classes*:

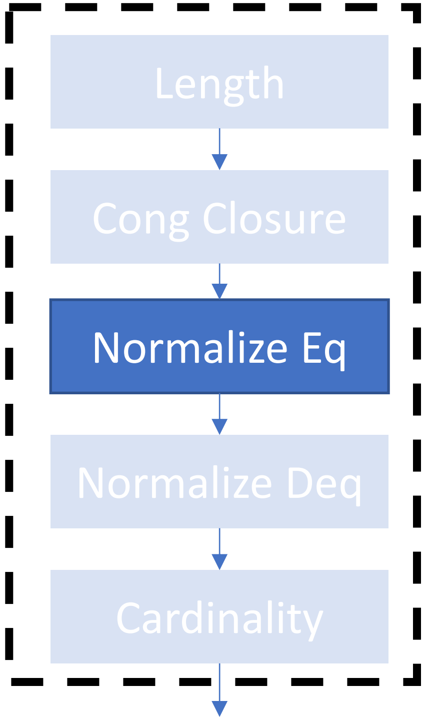


return lemma corresponding to T_s-conflict
if disequal terms in the same equivalence class

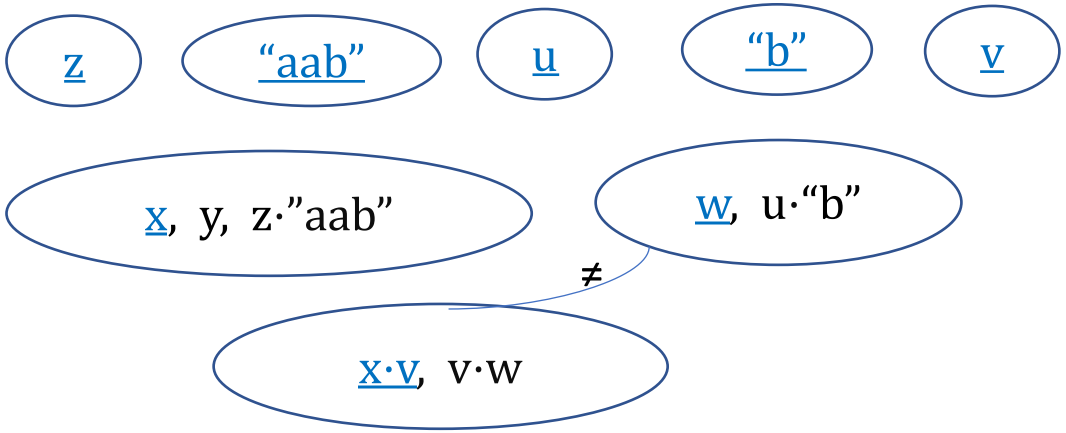
String Solver: Normalize Equality



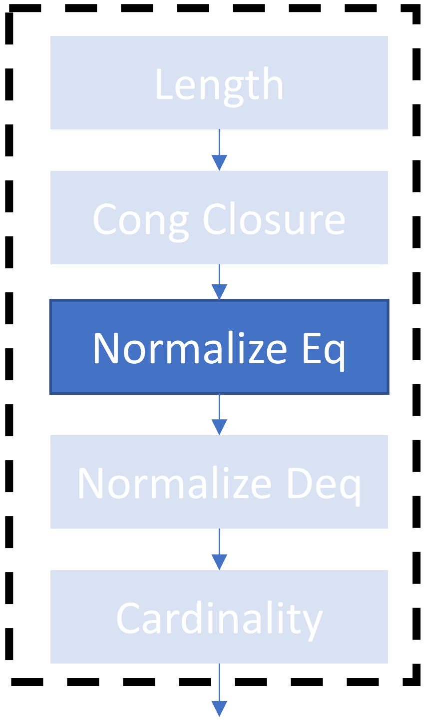
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



String Solver: Normalize Equality

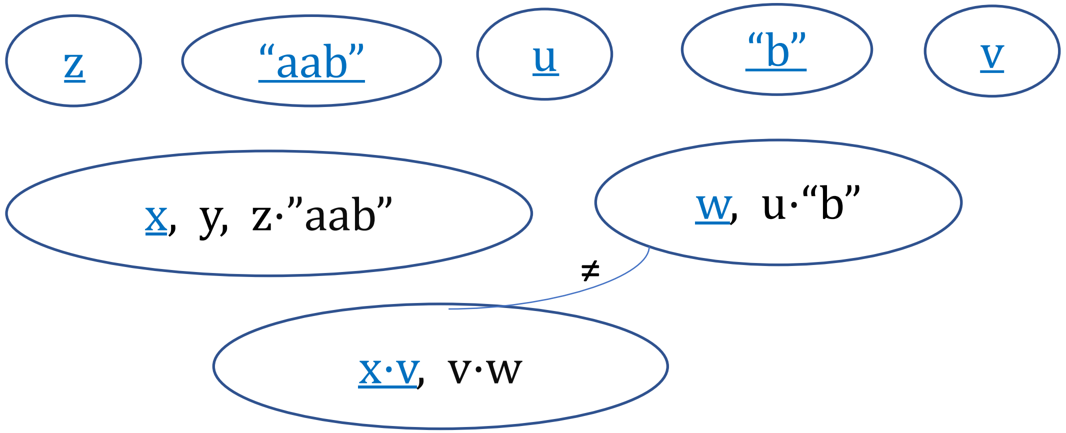


$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v = v\cdot w$$
$$x\cdot v \neq w$$

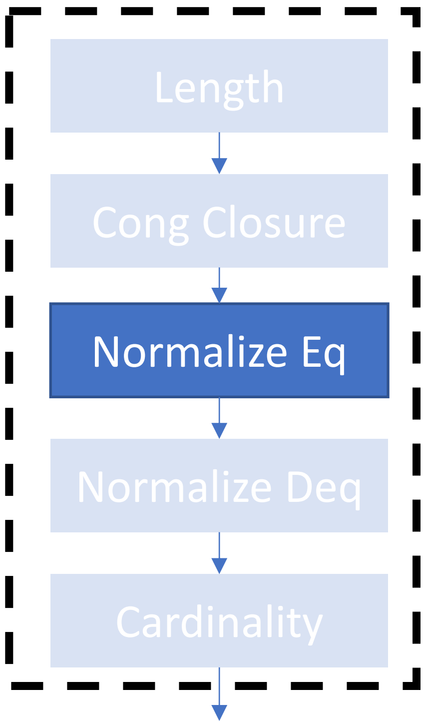


- Compute *normal forms* for equivalence classes
  - A normal form is a concatenation of string terms $r_1\cdot\ldots\cdot r_n$
    where each ri_i is the representative of its equivalence class
    Restriction: string constants must be chosen as representatives
  - An equivalence class can be assigned a normal form $r_1\cdot\ldots\cdot r_n$ if:
    Each non-variable term in it can be expanded (modulo equality and rewriting) to $r_1\cdot\ldots\cdot r_n$

String Solver: Normalize Equality

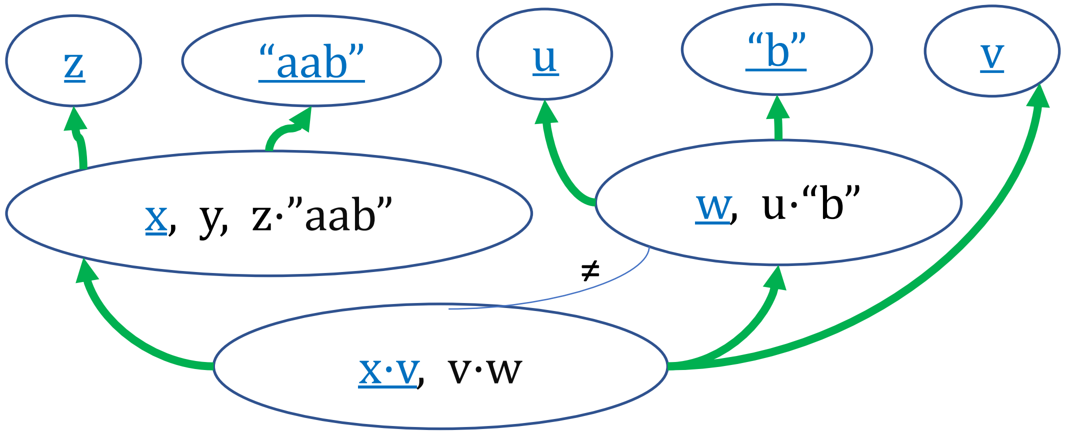


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$

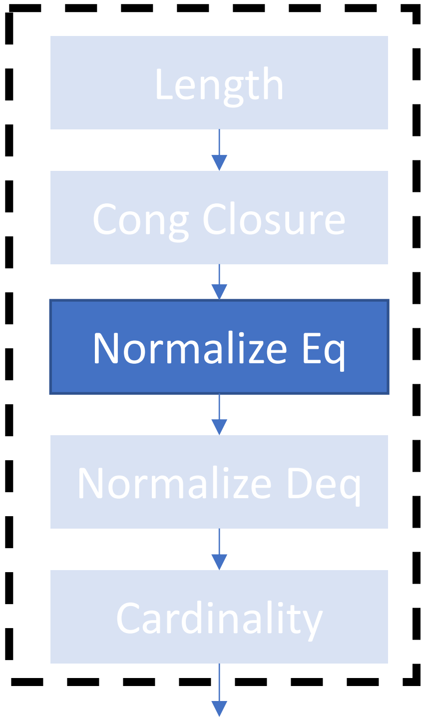


Normal forms computed by a bottom-up procedure

String Solver: Normalize Equality



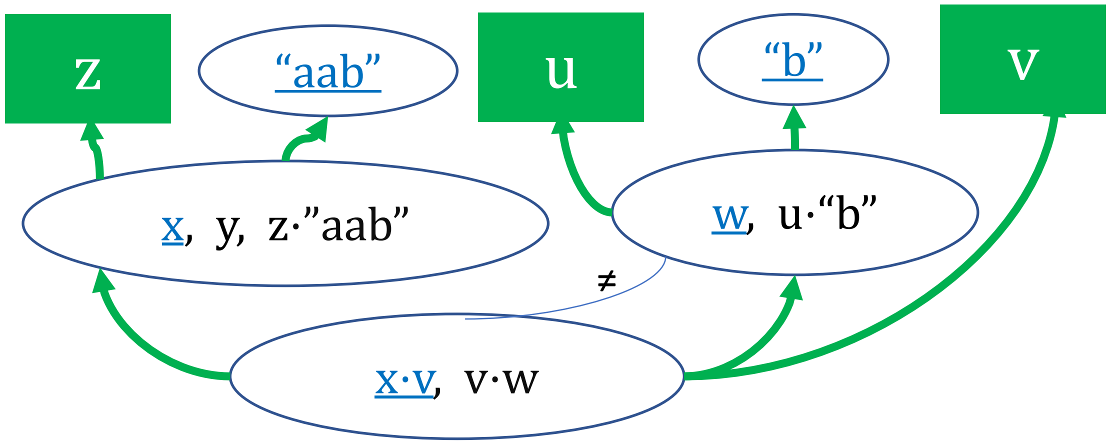
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



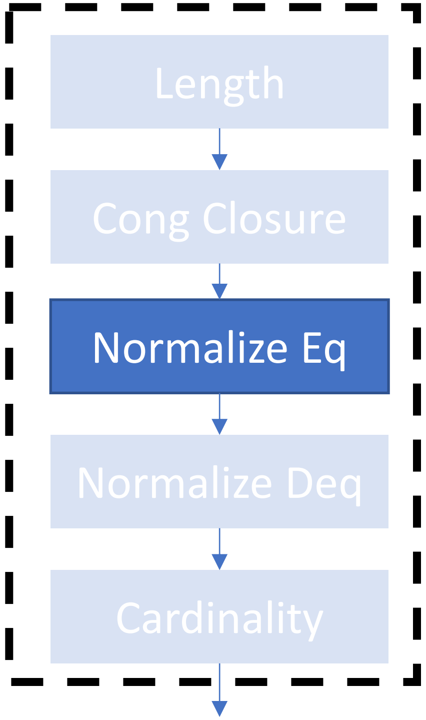
Normal forms computed by a bottom-up procedure

- First, compute containment relation induced by concatenation terms
  - To compute a n.f. for eq-class of $x \cdot v$, we must first compute a n.f. for eq-class of x and v
  - This relation is guaranteed to be acyclic due to length elaboration step (cycle $\Rightarrow$ LIA-conflict)

String Solver: Normalize Equality



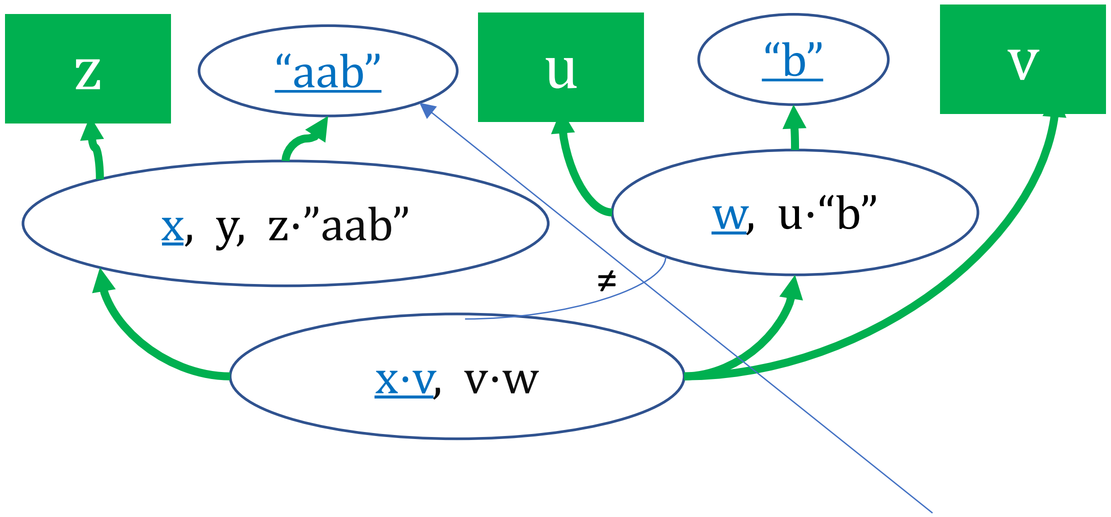
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



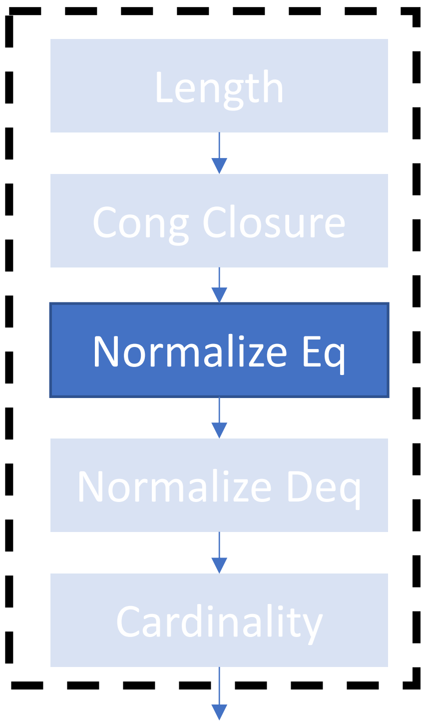
Normal forms computed by a bottom-up procedure

- First, compute containment relation induced by concatenation terms
  - To compute a n.f. for eq-class of $x \cdot v$, we must first compute a n.f. for eq-class of x and v
  - This relation is guaranteed to be acyclic due to length processing step (cycle $\Rightarrow$ LIA-conflict)
- Base case: eqc containing only variables can be assigned representative as a normal form
- Inductive case: compare the expanded form $t_1,\ldots,t_n$ of each non-variable term t
  - If $t_1 \cong \ldots \cong t_n$, assign to t. If there exists distinct t_i, t_j, then propagate or split

String Solver: Normalize Equality

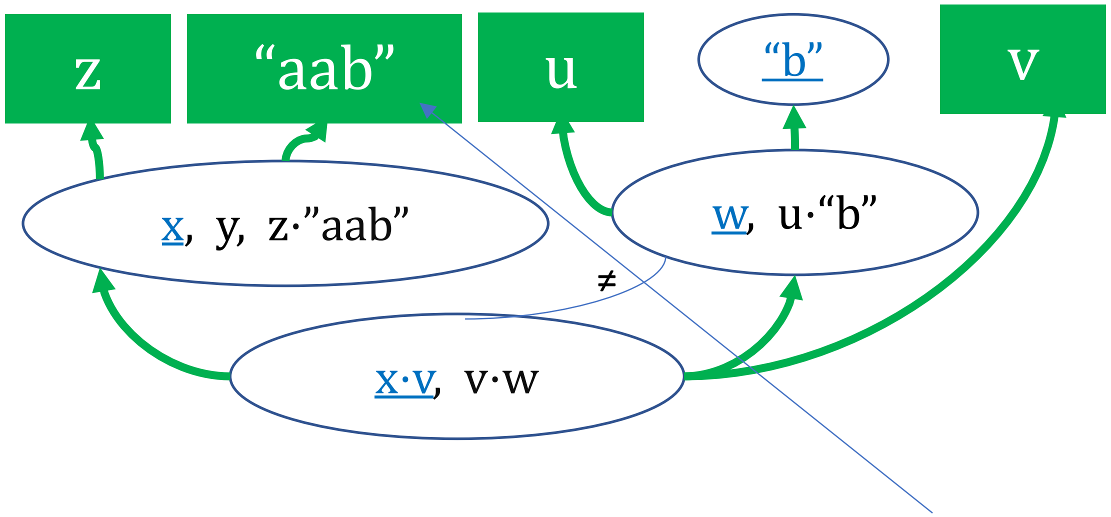


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



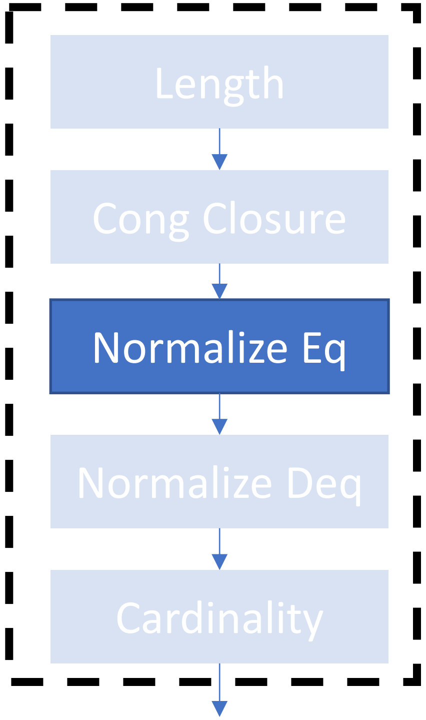
Single non-variable string term $\Rightarrow$ assign

String Solver: Normalize Equality

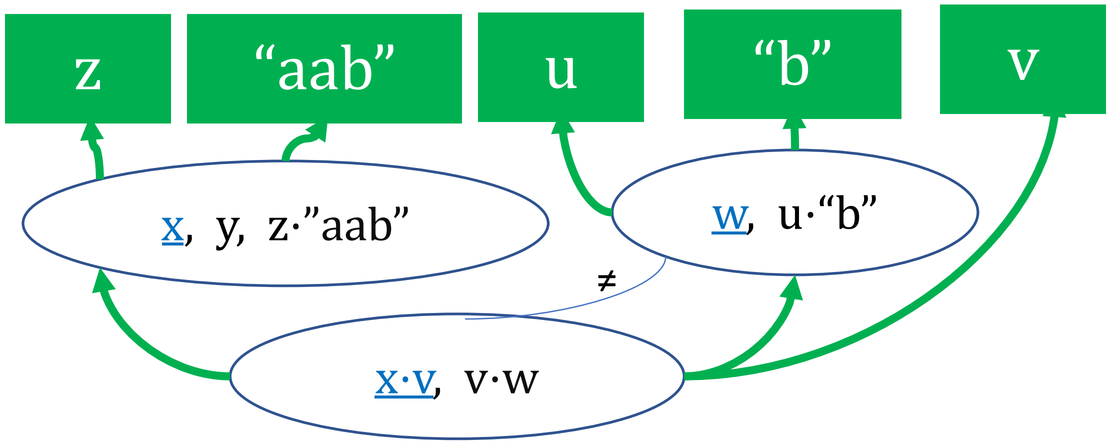


$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v = v\cdot w$$
$$x\cdot v \neq w$$

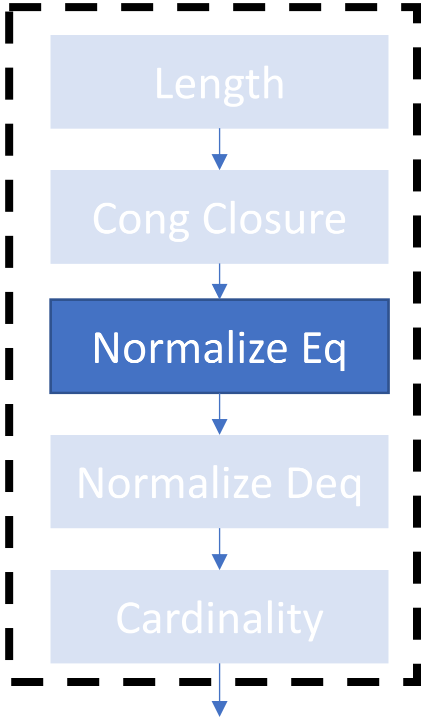
Single non-variable string term ⇒ assign



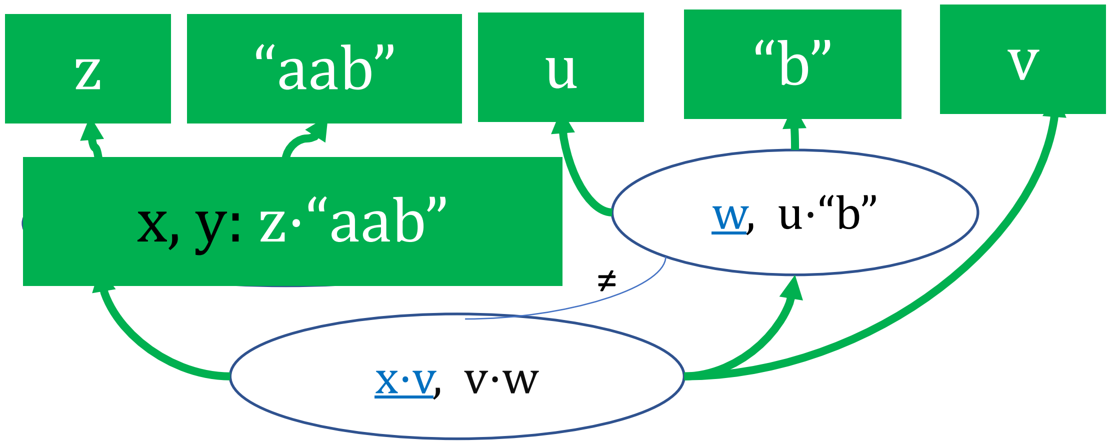
String Solver: Normalize Equality



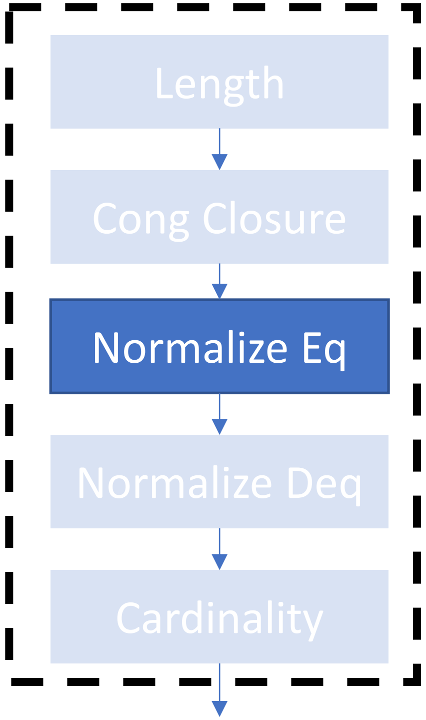
$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v = v\cdot w$$
$$x\cdot v \neq w$$



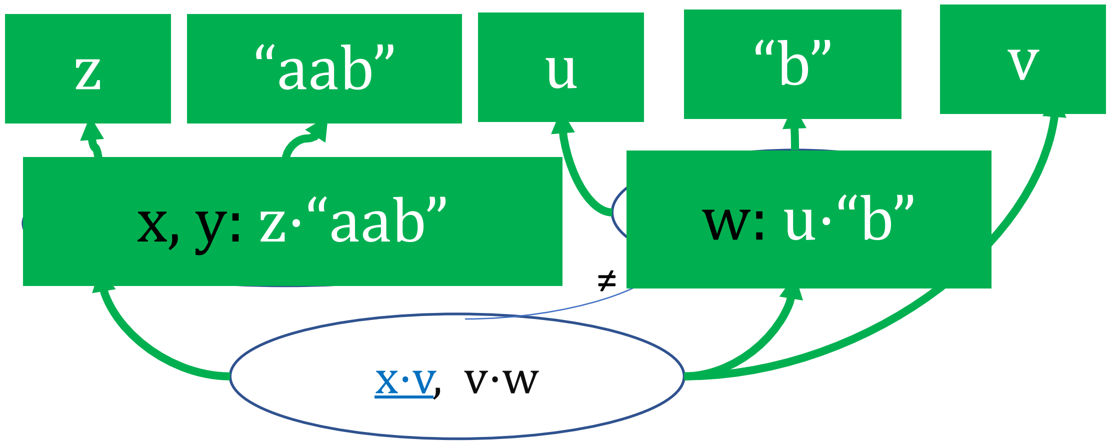
String Solver: Normalize Equality



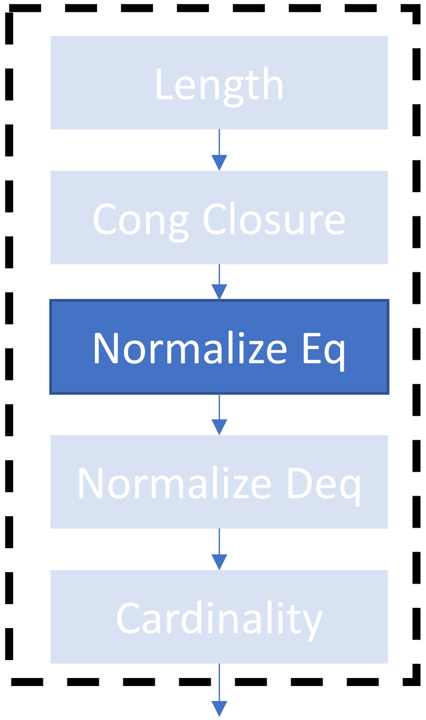
$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v = v\cdot w$$
$$x\cdot v \neq w$$



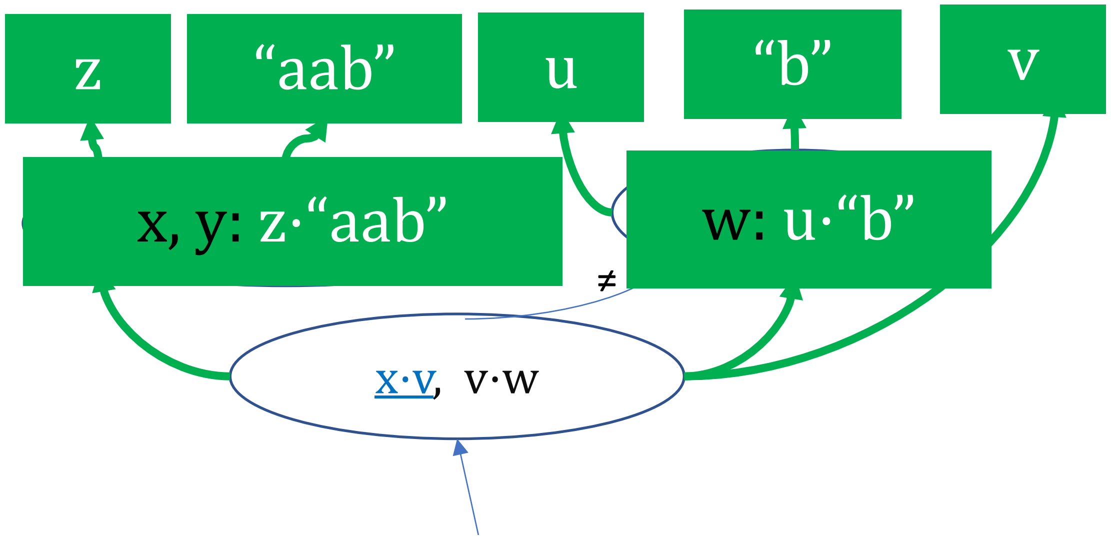
String Solver: Normalize Equality



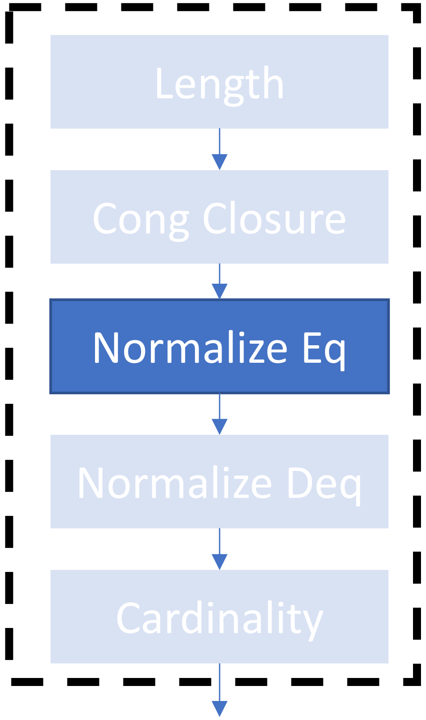
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



String Solver: Normalize Equality



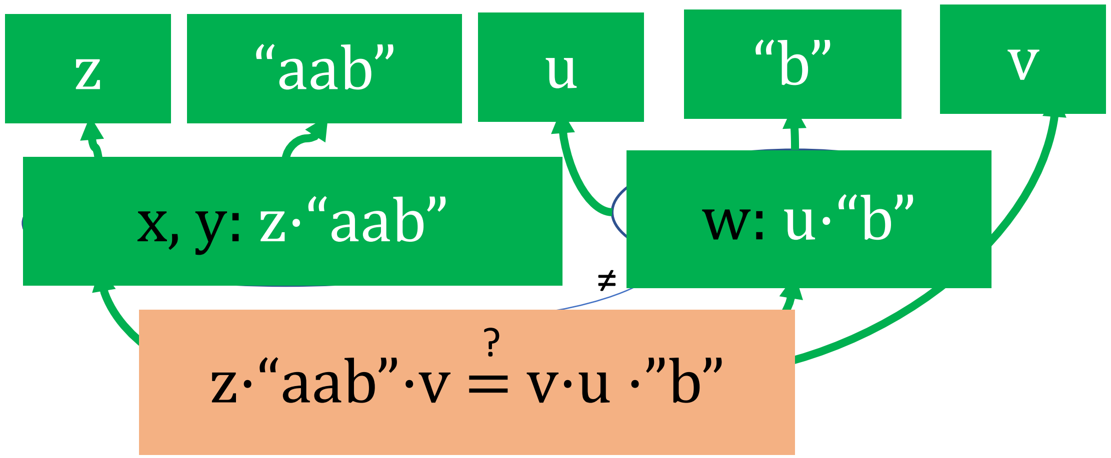
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



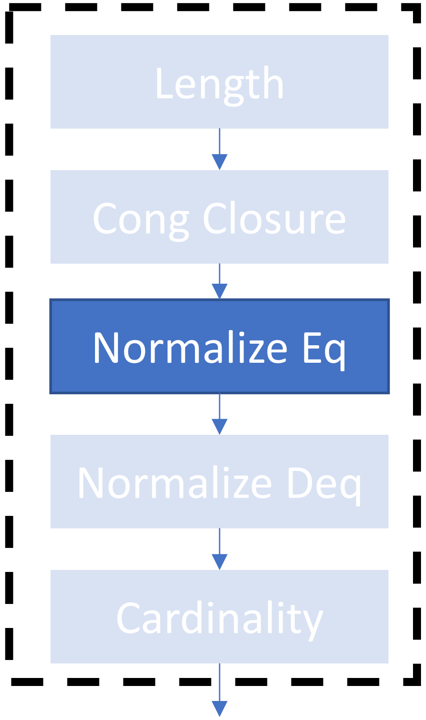
Equivalence class with two non-variable terms with distinct expanded forms:

- $x \cdot v = (z \cdot \text{"aab"}) \cdot v \quad = z \cdot \text{"aab"} \cdot v$
- $v \cdot w = v \cdot (u \cdot \text{"b"}) \quad = v \cdot u \cdot \text{"b"}$

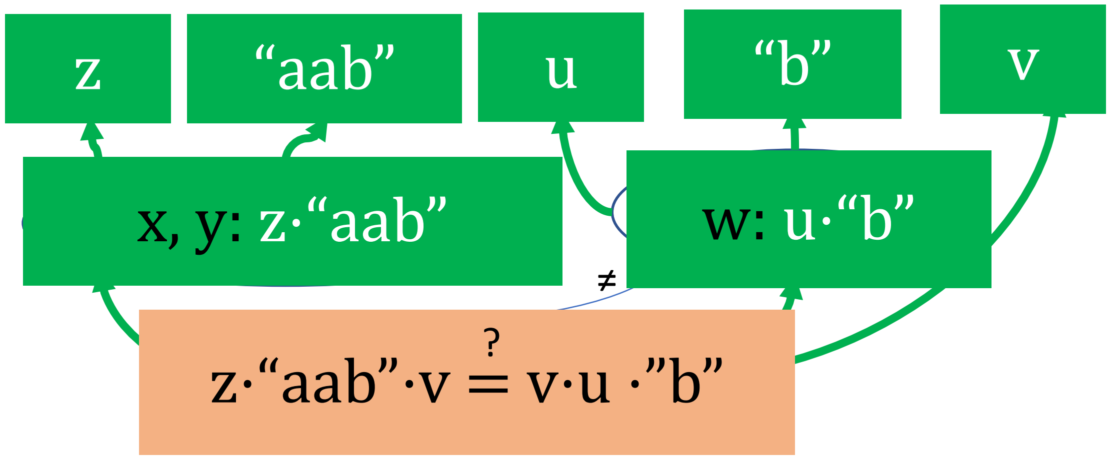
String Solver: Normalize Equality



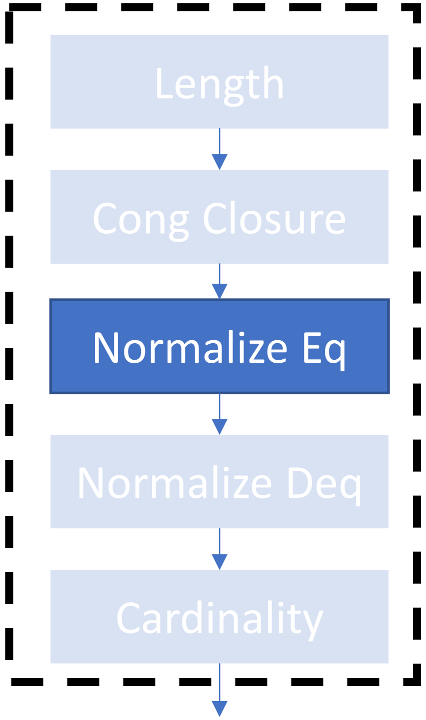
$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v = v\cdot w$$
$$x\cdot v \neq w$$



String Solver: Normalize Equality

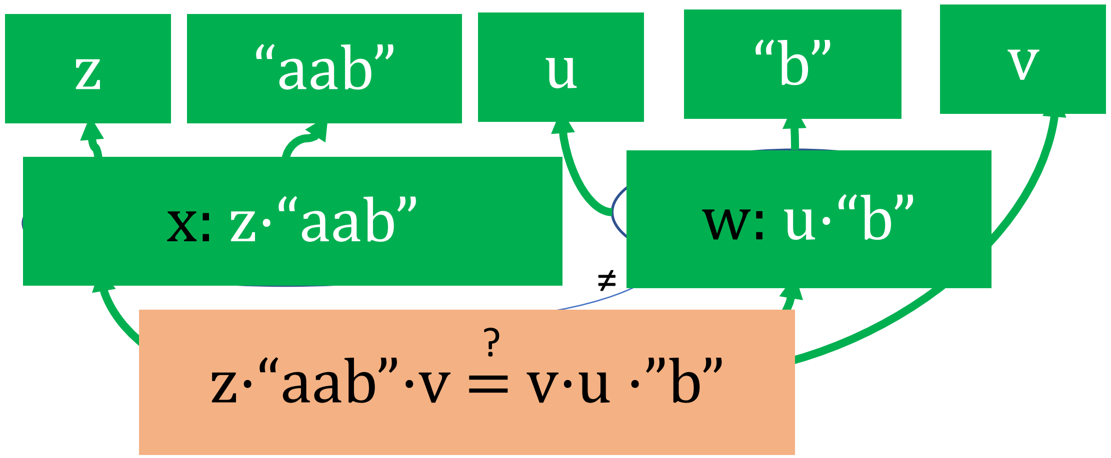


$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v = v\cdot w$$
$$x\cdot v \neq w$$

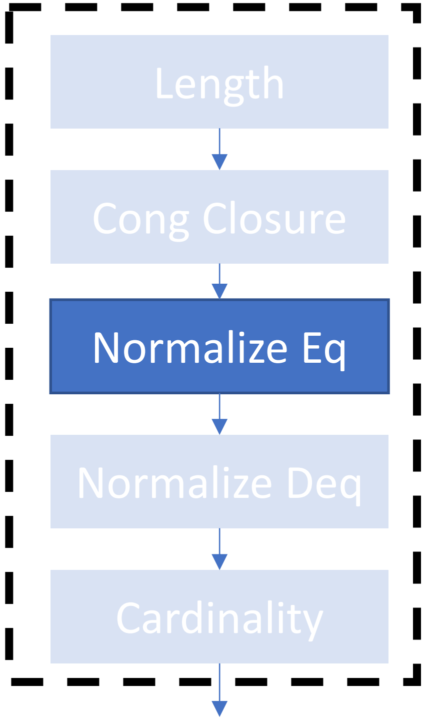


Goal: split strings so that all aligning components are equal

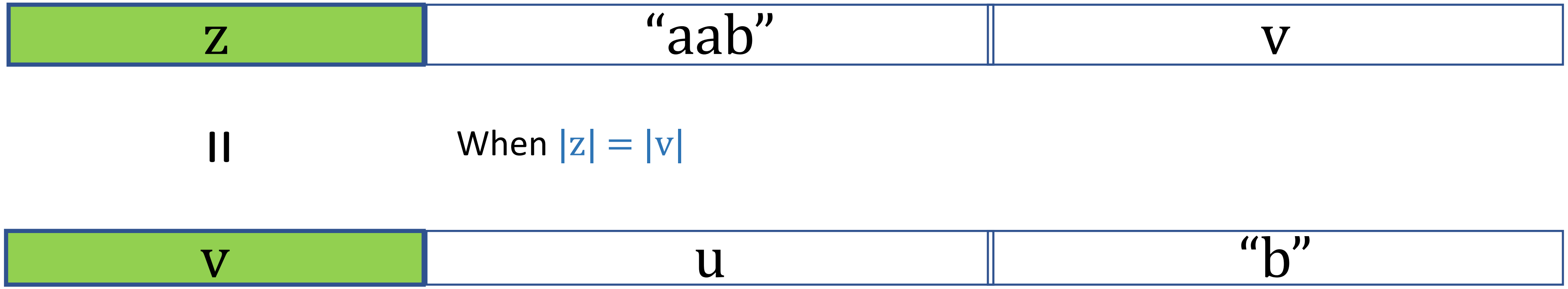
String Solver: Normalize Equality



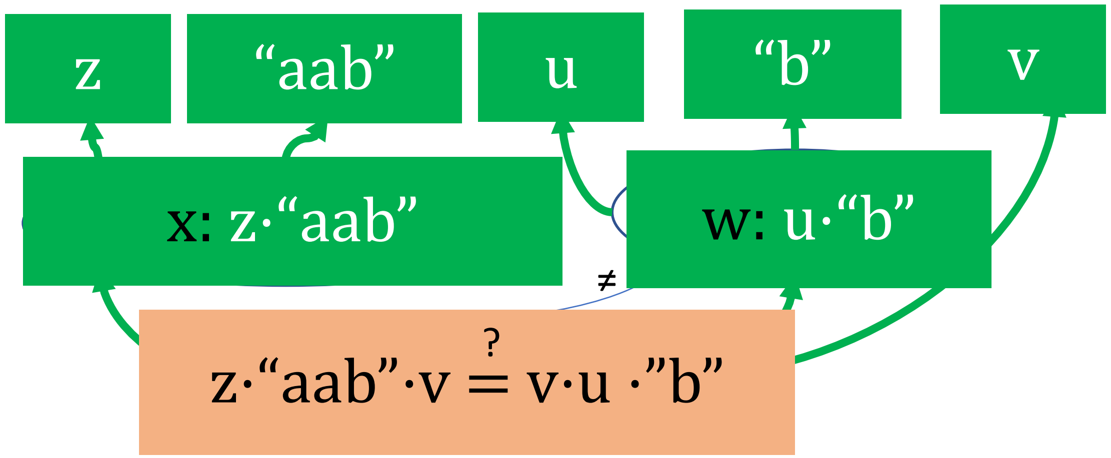
$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v = v\cdot w$$
$$x\cdot v \neq w$$



- Consider three cases for making these two terms equal:



String Solver: Normalize Equality



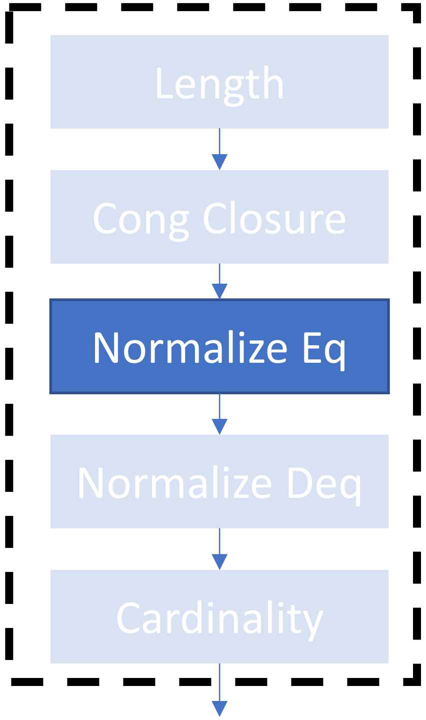
$x = z \cdot \text{"aab"}$

$y = x$

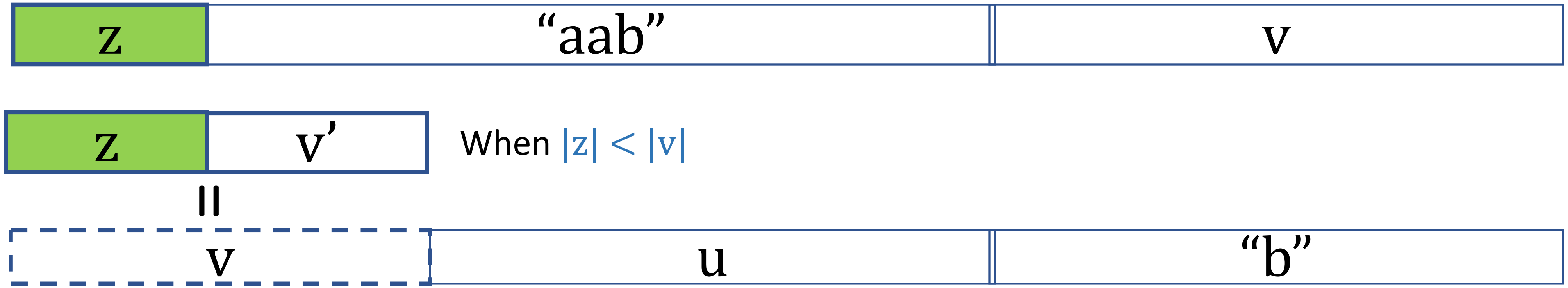
$w = u \cdot \text{"b"}$

$x \cdot v = v \cdot w$

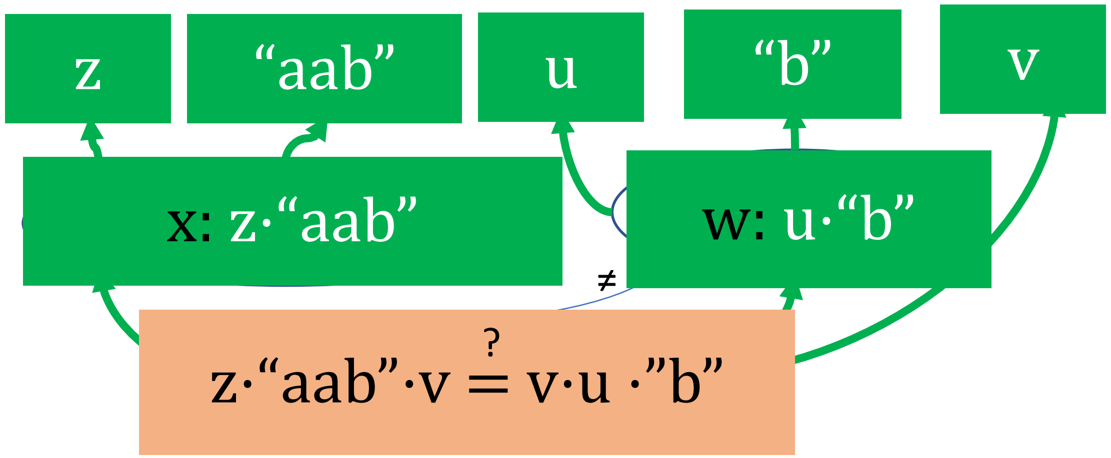
$x \cdot v \neq w$



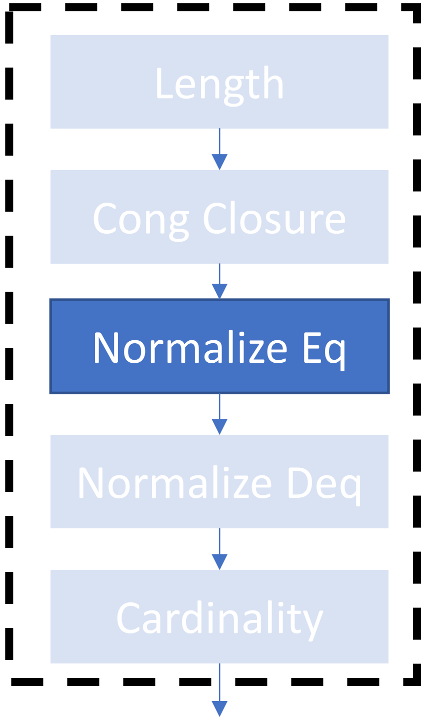
- Consider three cases for making these two terms equal:



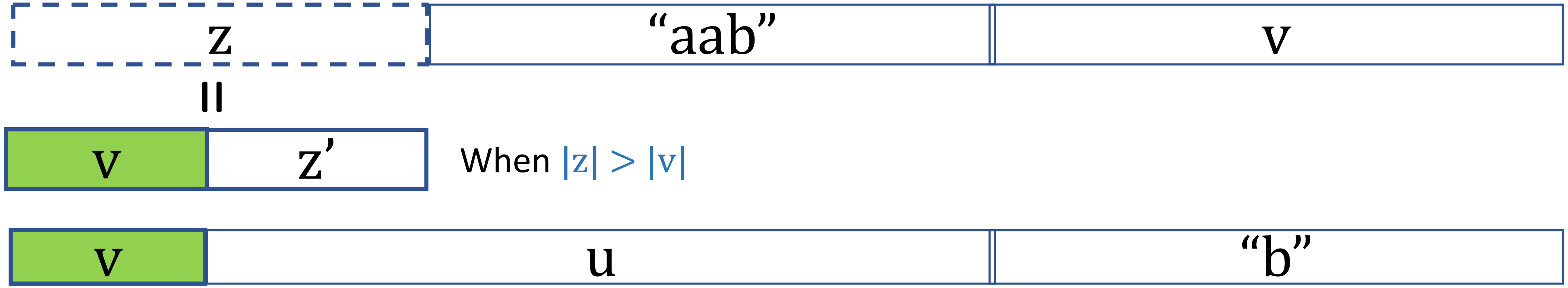
String Solver: Normalize Equality



$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v = v\cdot w$$
$$x\cdot v \neq w$$

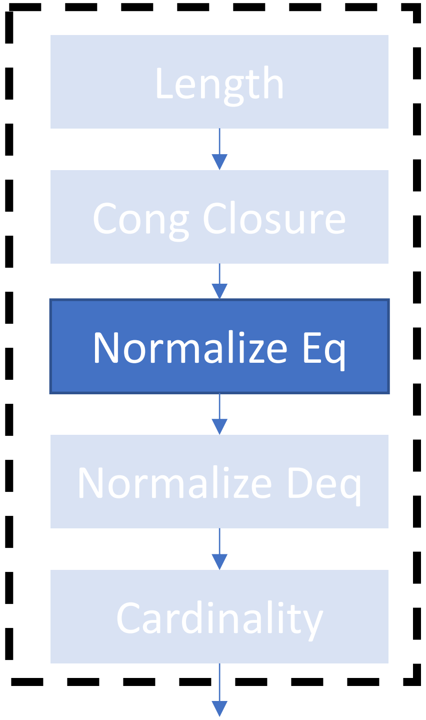


- Consider three cases for making these two terms equal:

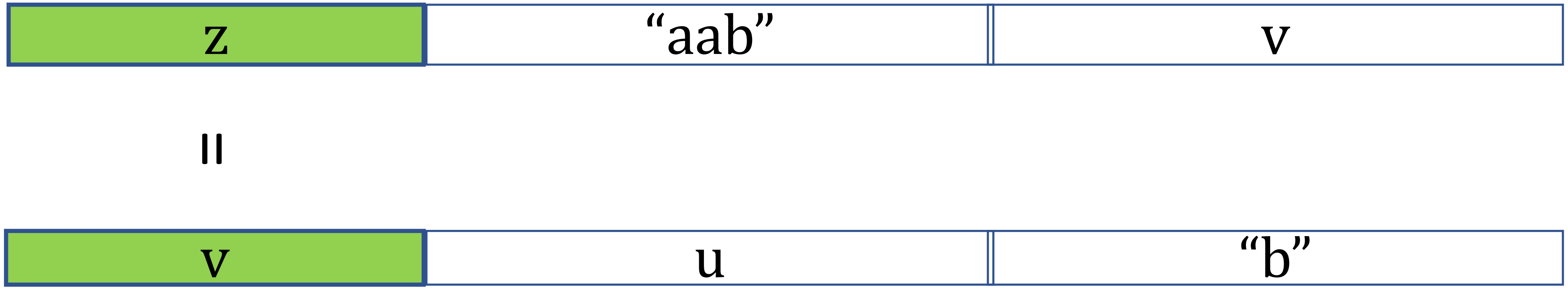


String Solver: Normalize Equality

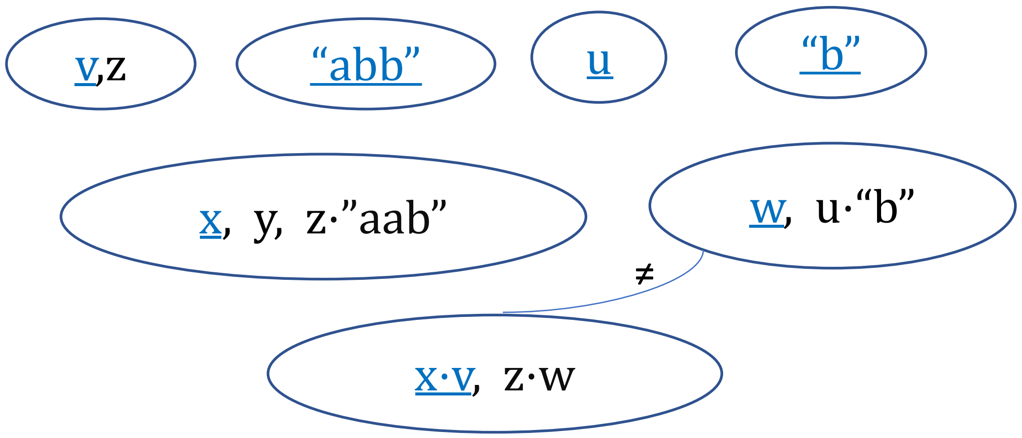
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$
$$z = v$$



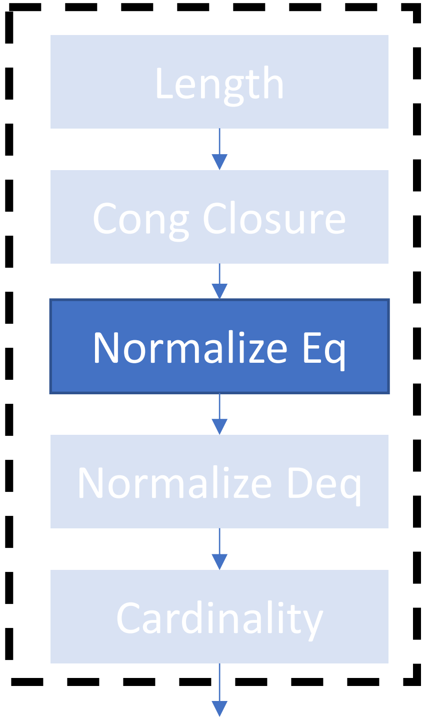
- Consider:



String Solver: Normalize Equality

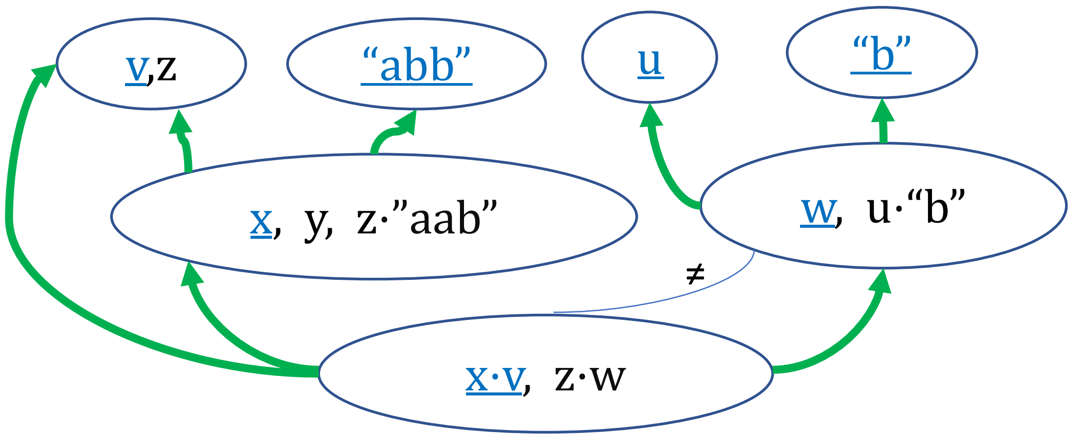


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$
$$z = v$$

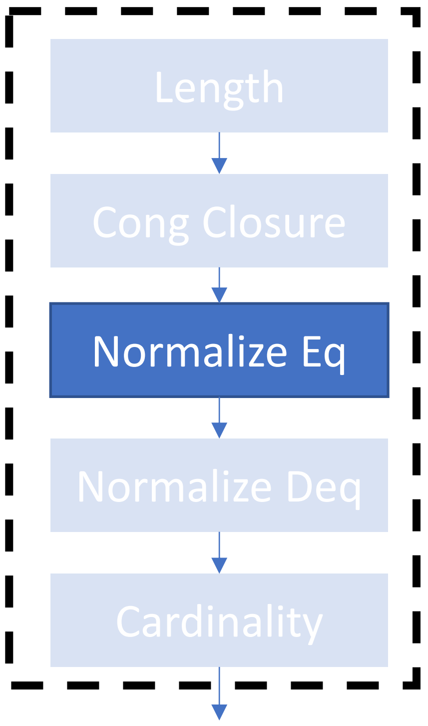


Recompute congruence closure

String Solver: Normalize Equality

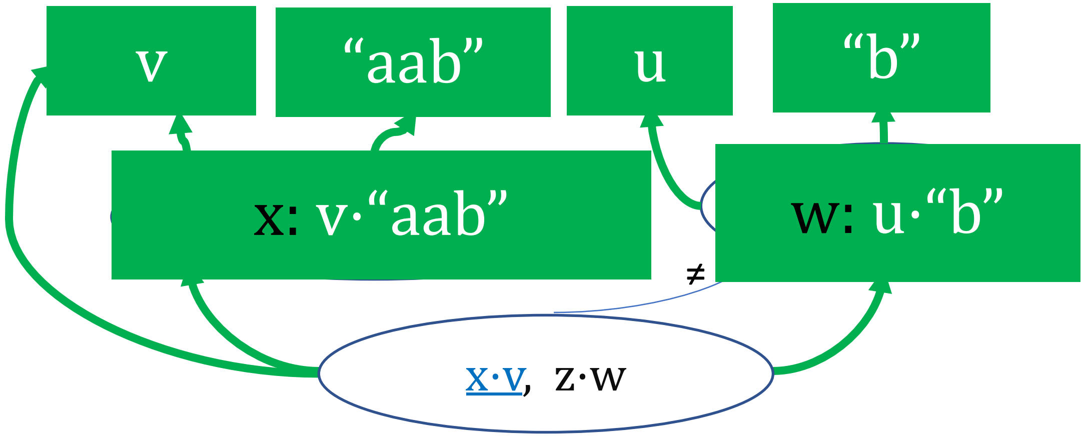


$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v = v\cdot w$$
$$x\cdot v \neq w$$
$$z = v$$

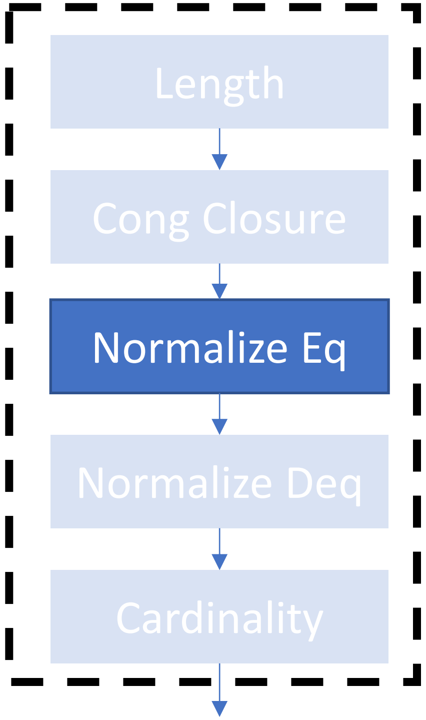


Recompute congruence closure and normal forms

String Solver: Normalize Equality

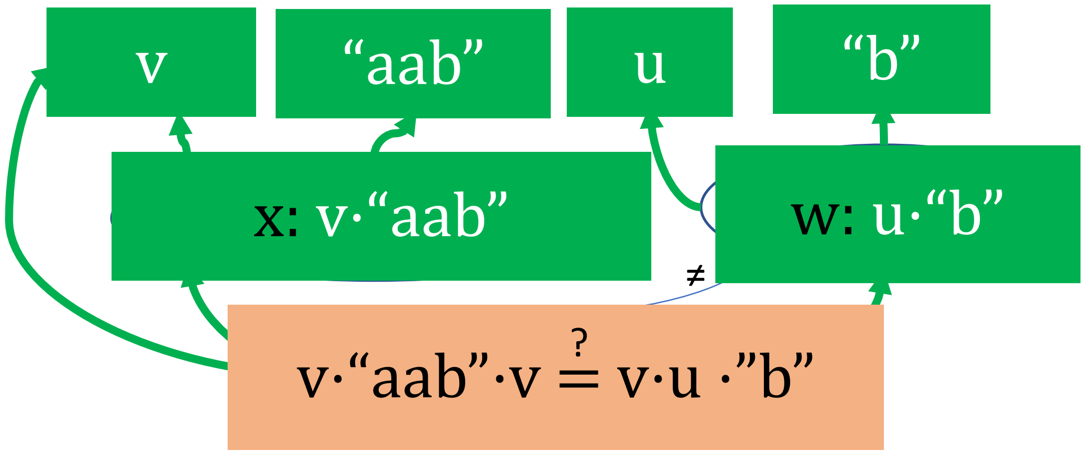


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$
$$z = v$$

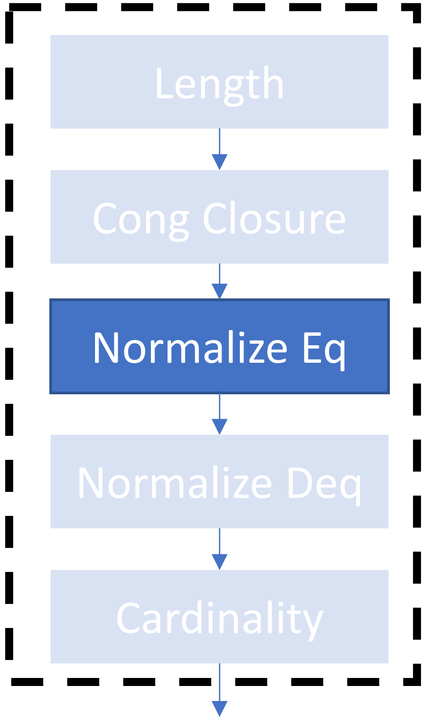


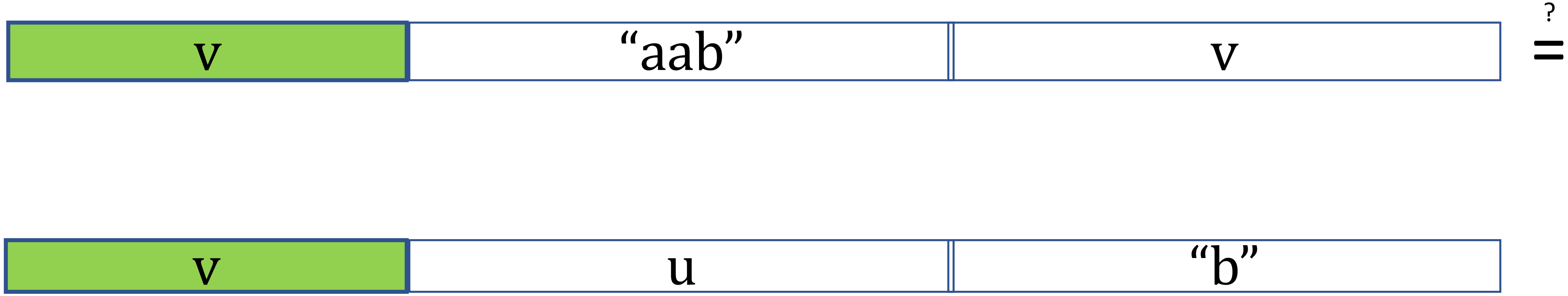
Recompute congruence closure and *normal forms*

String Solver: Normalize Equality

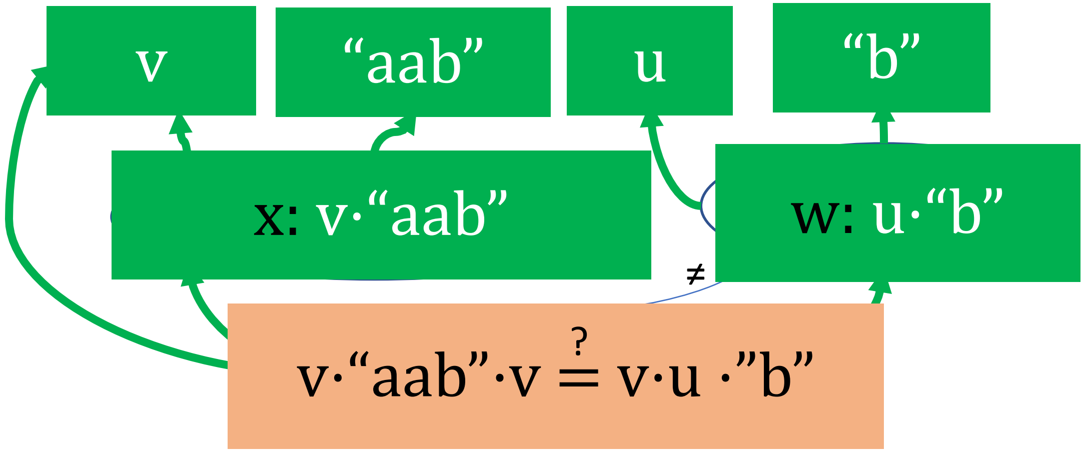


$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v = v\cdot w$$
$$x\cdot v \neq w$$
$$z = v$$

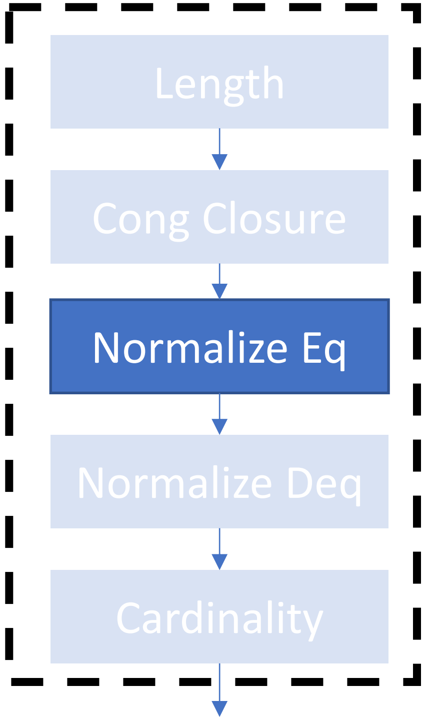




String Solver: Normalize Equality



$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v = v\cdot w$$
$$x\cdot v \neq w$$
$$z = v$$



v	“aab”	v

repeat the process on these components

v	u	“b”

Splitting on String Equalities

Choosing how to process equalities is
highly non-trivial and critical to performance:

- Prefer propagations over splits

  Infer $x \cdot w = y \cdot w \Rightarrow x = y$ before $x \cdot w = z \cdot v \Rightarrow (x = z \cdot x' \vee z = x \cdot z')$

- Can consider both the prefix and suffix of strings

  Infer $w \cdot x = w \cdot y \Rightarrow x = y$

- Use length entailment [Zheng et al 2015]

  If $|x| > |y|$ is entailed by the arith. solver, then $x \cdot w = y \cdot v \wedge |x| > |z| \Rightarrow x = y \cdot x'$

Splitting on String Equalities

Choosing how to process equalities is
highly non-trivial and critical to performance:

- Propagation based on adjacent constants
  $x \cdot \text{“b”} = \text{“aab”} \cdot y \Rightarrow x = \text{“aa”} \cdot x'$, since “b” cannot overlap with prefix “aa”

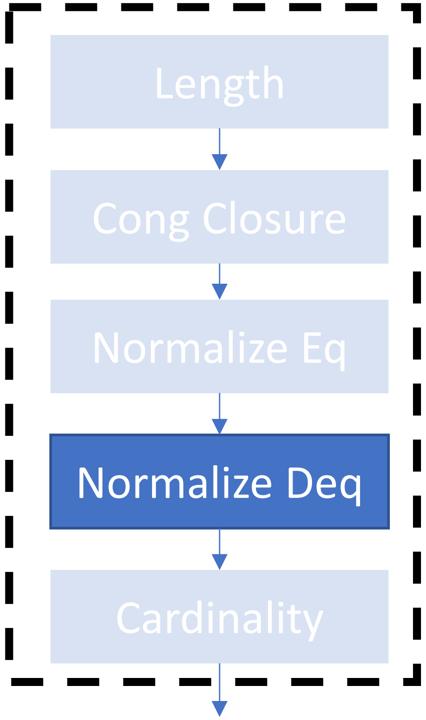
- Special treatment for looping word equations [Liang et al 2014]
  - splitting leads to non-termination; reduce to RE membership instead
  - e.g. $x \cdot \text{“ba”} = \text{“ab”} \cdot x \Rightarrow x \in (\text{“ab”})^* \cdot \text{”a”}$

- Deduced string equalities are not sent as unit lemmas
  instead they are maintained internally

String Solver: Normalize Disequalities

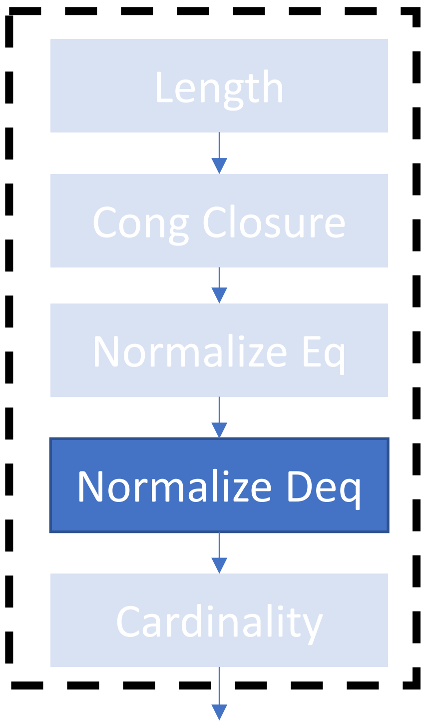
modified example

$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v \neq v \cdot w$$



String Solver: Normalize Disequalities

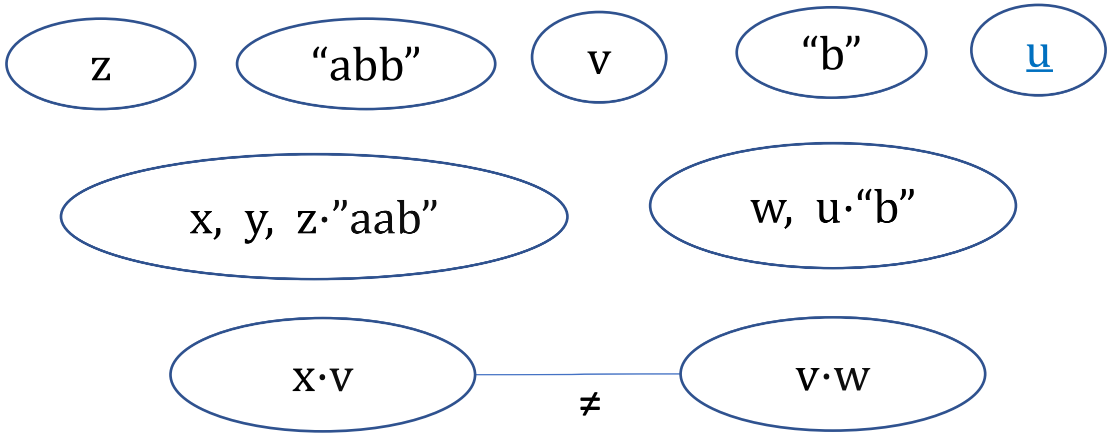
$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v \neq v\cdot w$$



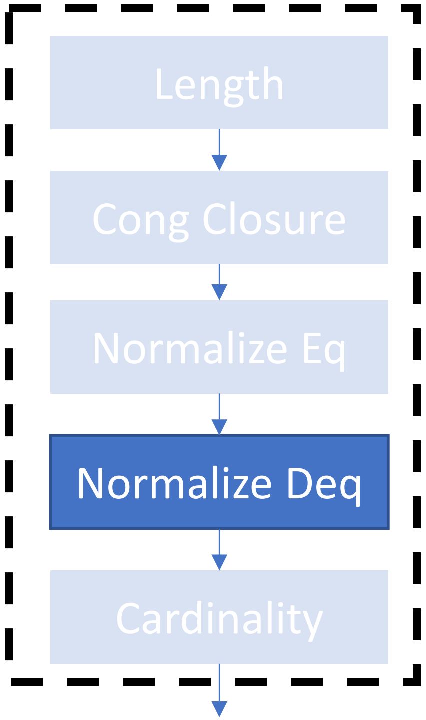
Disequalities are handled analogously to equalities

- If $|x\cdot v| \neq |v\cdot w|$, then trivially $x\cdot v \neq v\cdot w$
- Otherwise, consider the normal forms of $x\cdot v$ and $v\cdot w$ from previous step

String Solver: Normalize Disequalities

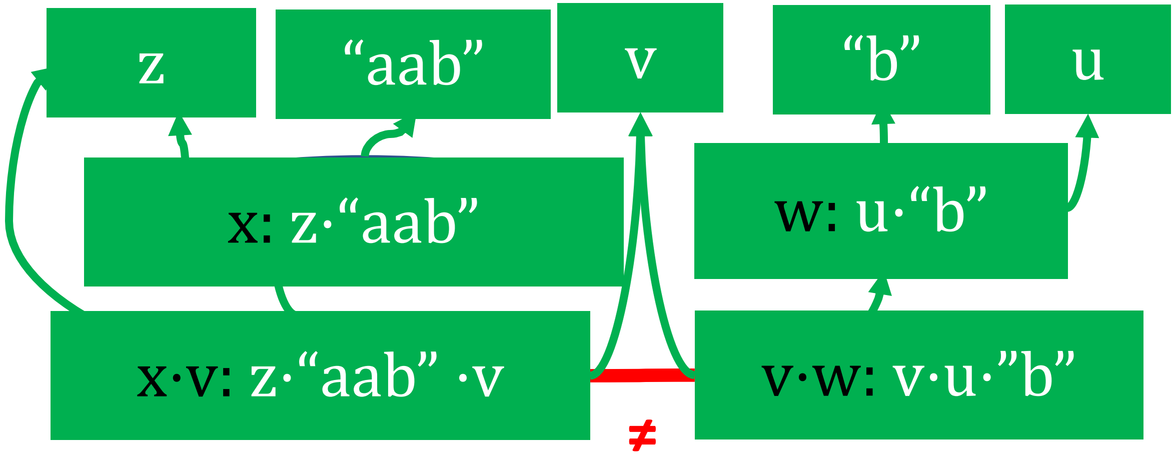


$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v \neq v\cdot w$$

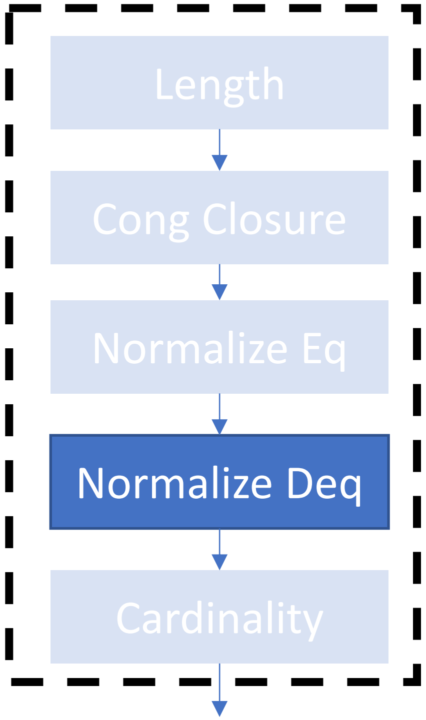


Disequalities are handled analogously to equalities

String Solver: Normalize Disequalities



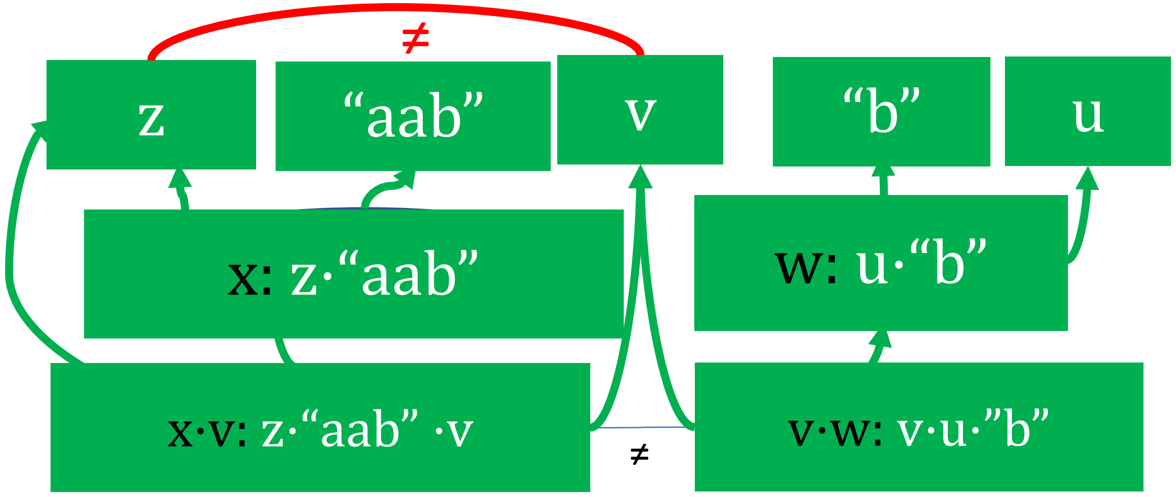
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v \neq v \cdot w$$



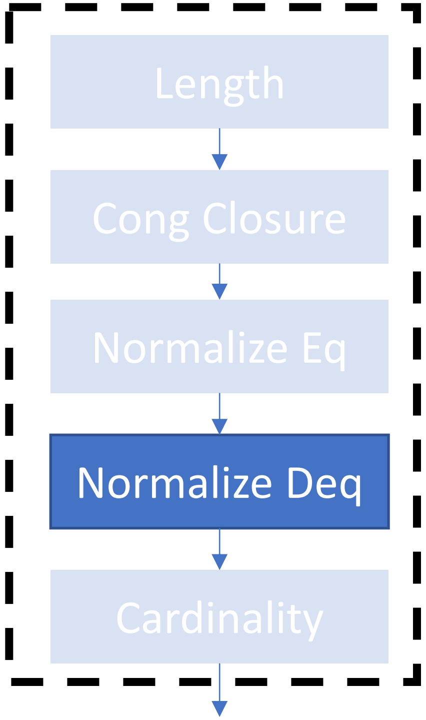
Disequalities are handled analogously to equalities

Goal: find any aligning component that is disequal

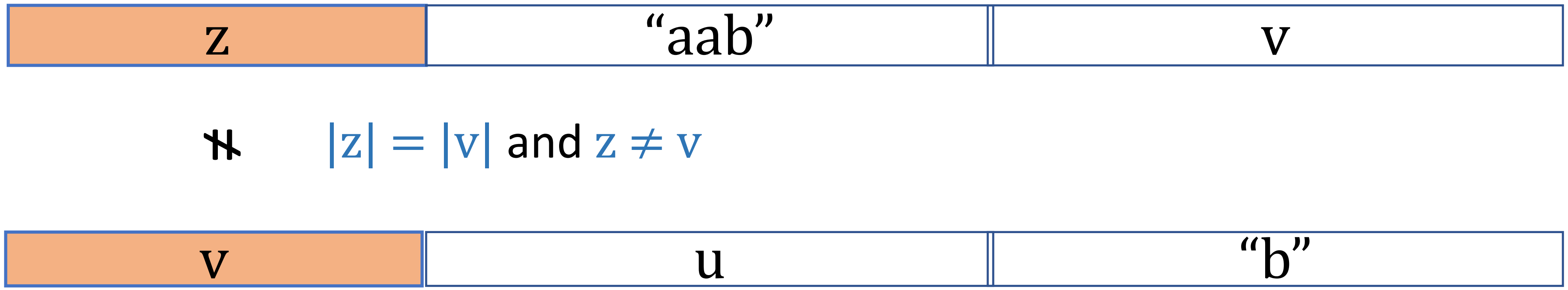
String Solver: Normalize Disequalities



$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v \neq v\cdot w$$
$$v \neq z$$

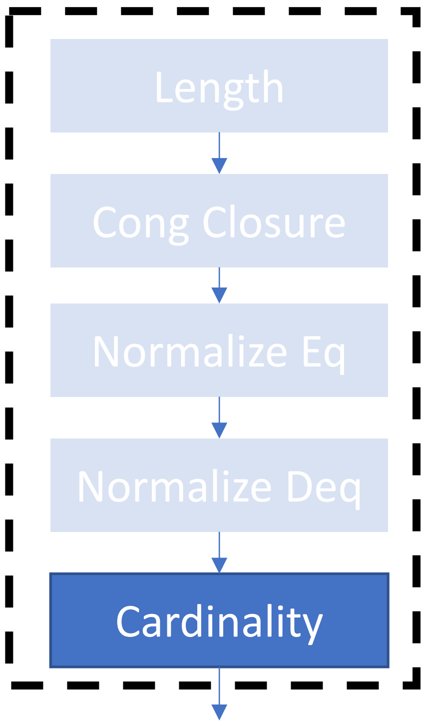


Disequalities are handled analogously to equalities



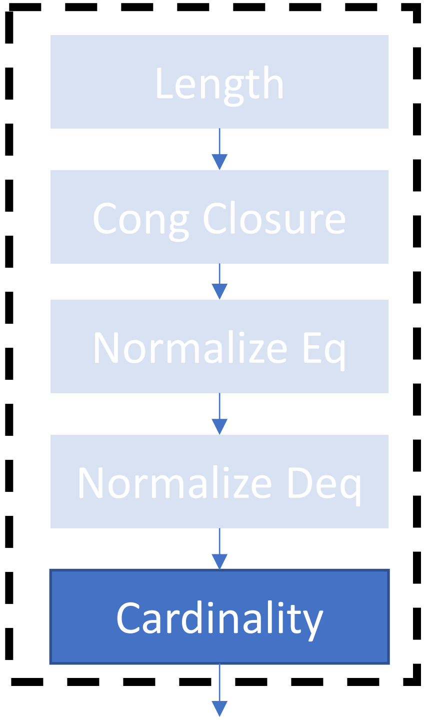
String Solver: Cardinality

$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v \neq v \cdot w$$
$$v \neq z$$



String Solver: Cardinality

$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v \neq v \cdot w$$
$$v \neq z$$



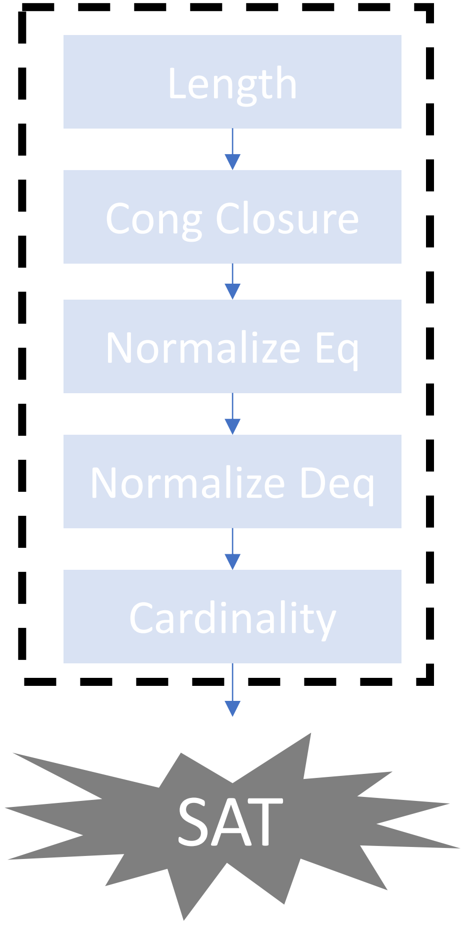
- M_S may be unsatisfiable since alphabet A is finite
- For instance, if:
  - A is a finite alphabet of 256 characters, and
  - M_S entails the existence of 257 distinct strings of length 1

  $\Rightarrow$ Then M_S is unsatisfiable

$\therefore$ $(\text{distinct}(s_1, \ldots, s_{257}) \wedge |s_1| = \ldots = |s_{257}|) \Rightarrow |s_1| > 1$

String Solver: Return SAT

$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v \neq v \cdot w$$
$$v \neq z$$

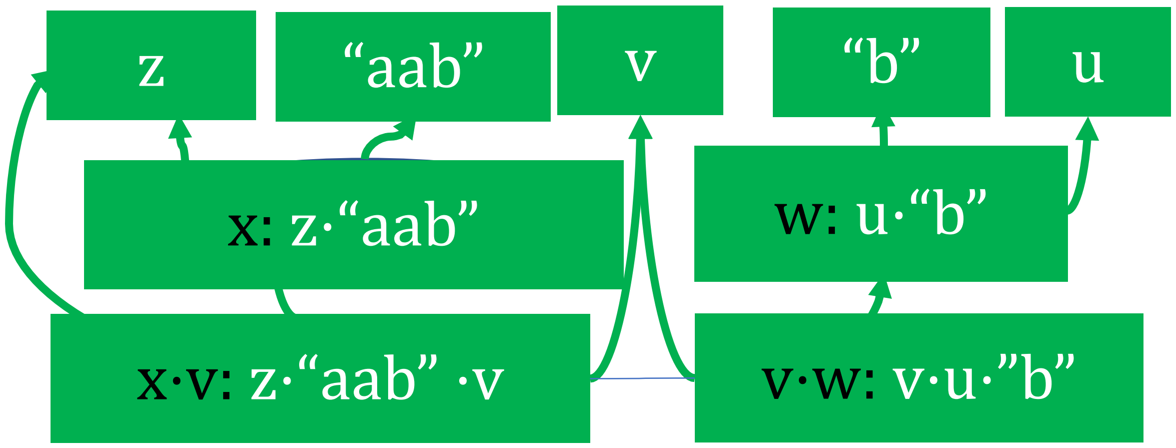


If all steps finish with no new lemmas:

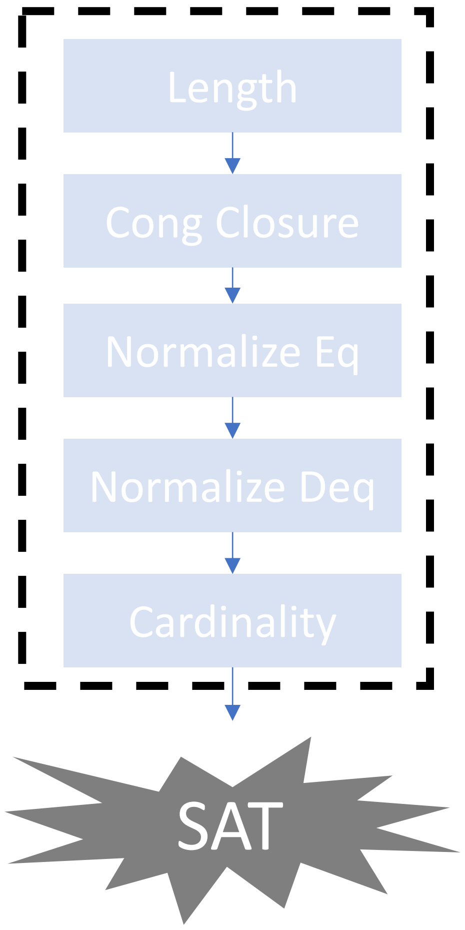
1. M_s is T_s-satisfiable
2. Model can be computed based on normal forms
   - String constants assigned to eq classes whose normal form is a variable

     Length fixed by model from arithmetic solver
   - Each variable interpreted as the valuation of the normal form of their eq class

String Solver: Return SAT



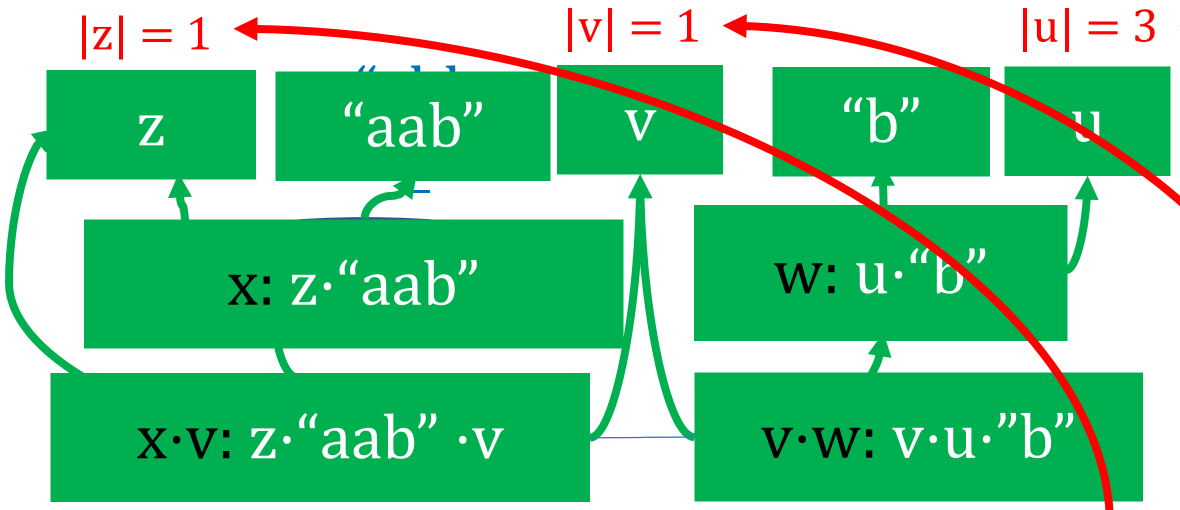
$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v \neq v\cdot w$$
$$v \neq z$$



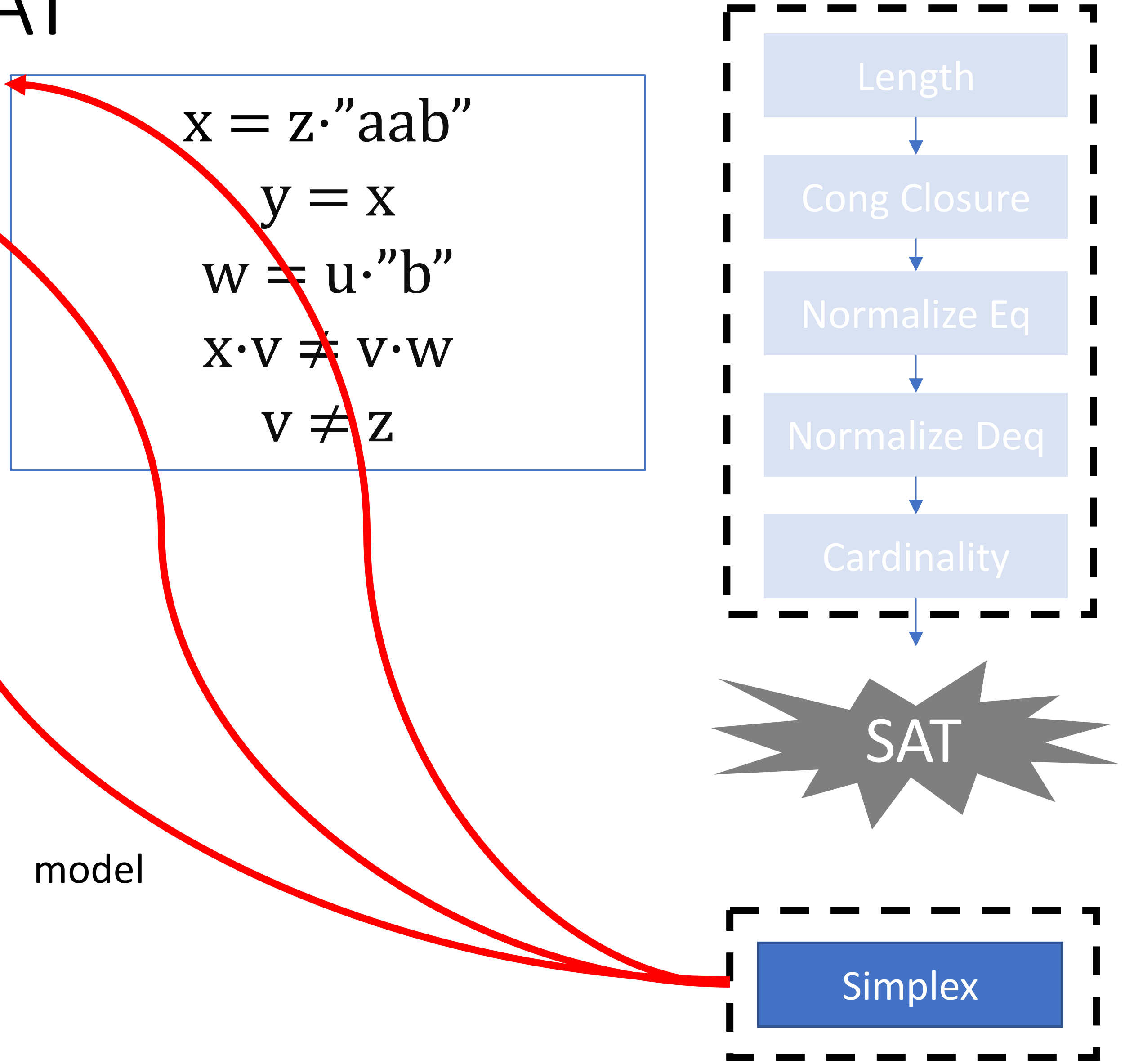
If all steps finish with no new lemmas:

1. M_s is T_s-satisfiable
2. Model can be computed based on normal forms
   - String constants assigned to eq classes whose normal form is a variable
     - Length fixed by model from arithmetic solver
   - Each variable interpreted as the valuation of the normal form of their eq class

String Solver: Return SAT

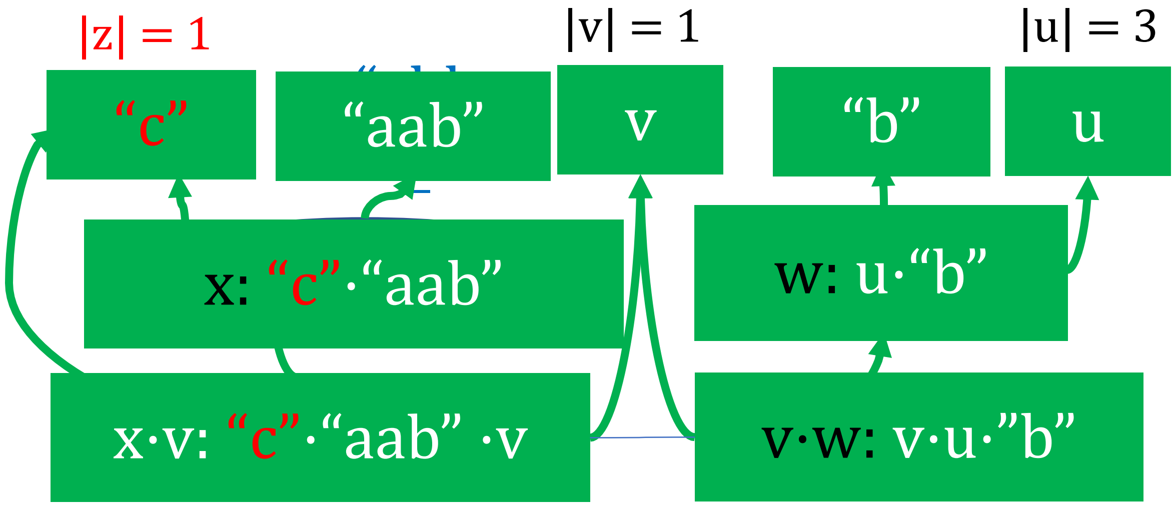


$x = z\cdot\text{"aab"}$
$y = x$
$w = u\cdot\text{"b"}$
$x\cdot v \neq v\cdot w$
$v \neq z$

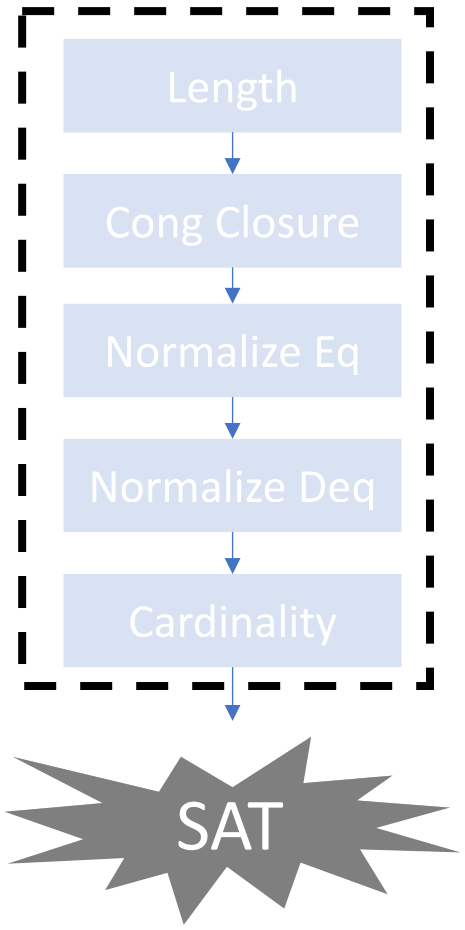


Example:

String Solver: Return SAT



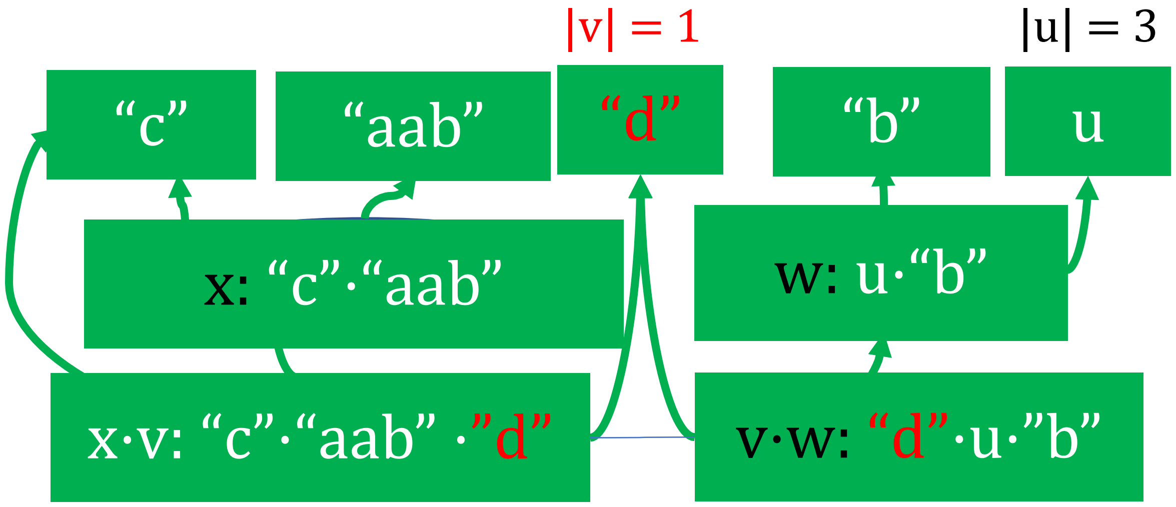
$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v \neq v\cdot w$$
$$v \neq z$$



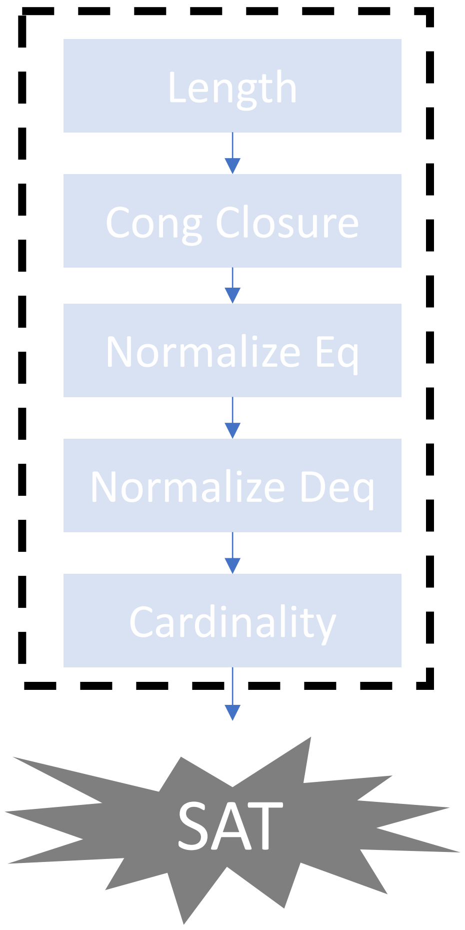
Example:

- z assigned to “c”

String Solver: Return SAT



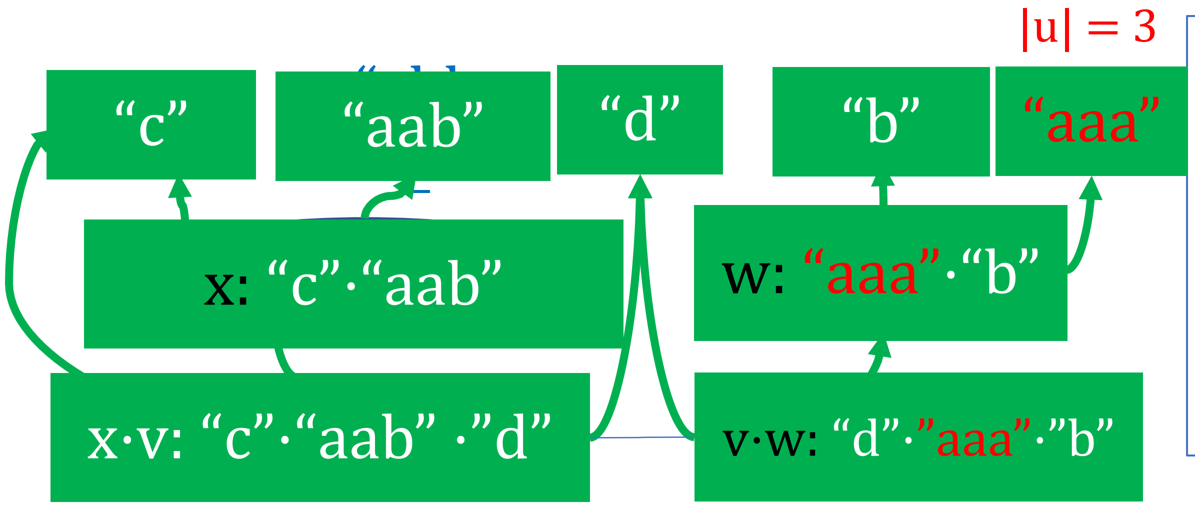
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v \neq v \cdot w$$
$$v \neq z$$



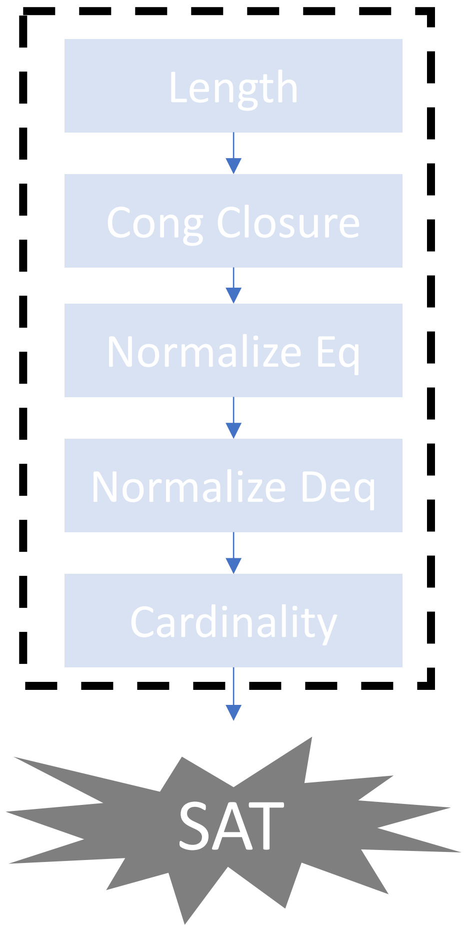
Example:

- z assigned to “c”
- v assigned to “d”

String Solver: Return SAT



$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v \neq v\cdot w$$
$$v \neq z$$

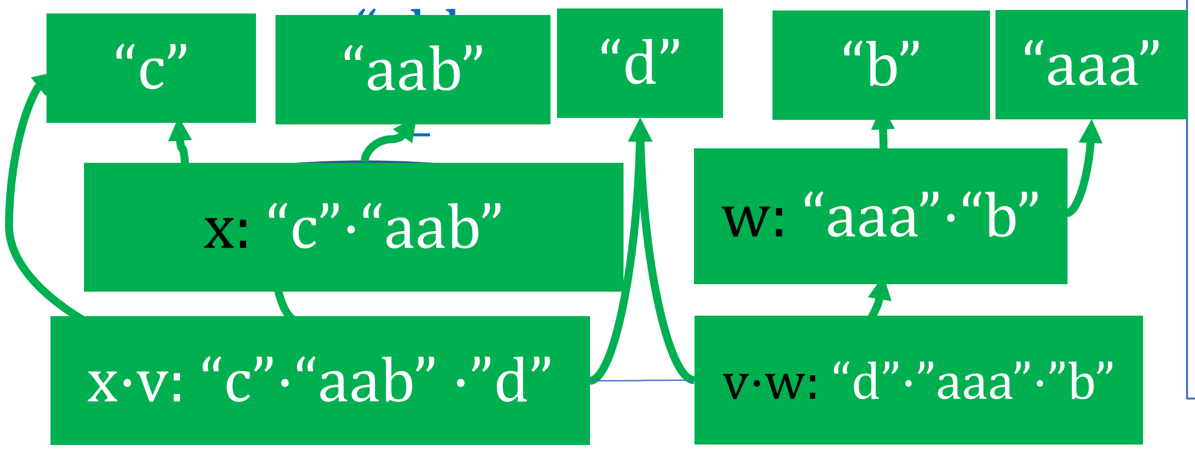


Example:

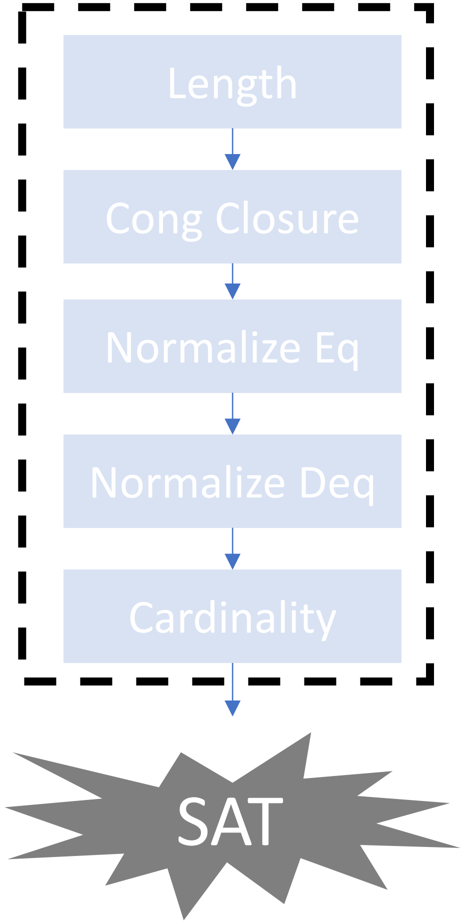
- z assigned to “c”
- v assigned to “d”
- u assigned to “aaa”

Cardinality step ensures enough enough constants exist

String Solver: Return SAT



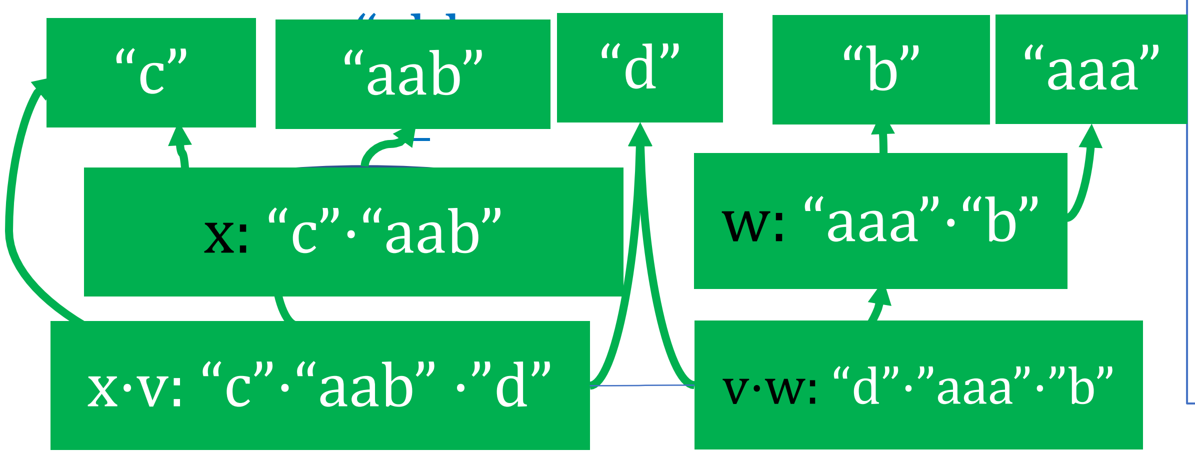
$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v \neq v\cdot w$$
$$v \neq z$$



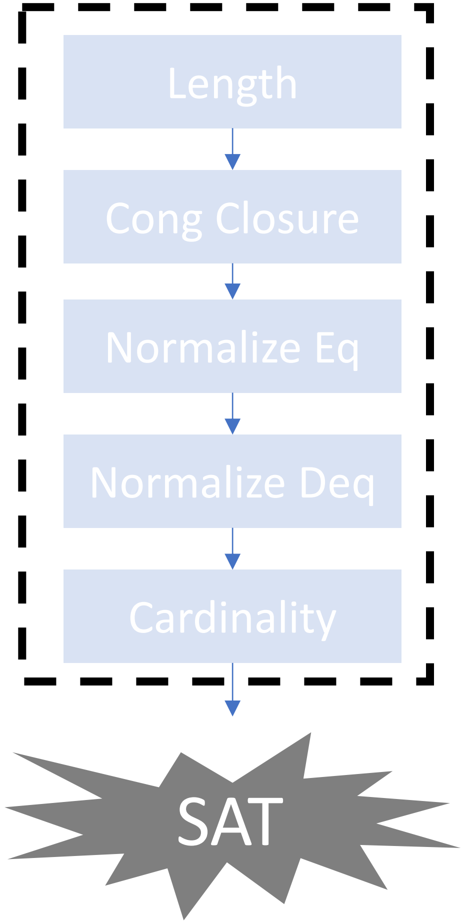
Example:

- z assigned to "c"
- v assigned to "d"
- u assigned to "aaa"
- Variables assigned to value of the normal form of their eq classes:
  - x,y assigned to "caab", w assigned to "aaab"

String Solver: Return SAT



$$x = z\cdot\text{"aab"}$$
$$y = x$$
$$w = u\cdot\text{"b"}$$
$$x\cdot v \neq v\cdot w$$
$$v \neq z$$



Example:

- z assigned to “c”
- v assigned to “d”
- u assigned to “aaa”
- Variables assigned to value of the normal form of their eq classes:
  - x,y assigned to “caab”, w assigned to “aaab”

Saturation criteria of procedure ensures this model satisfies M_s

Advanced Topics

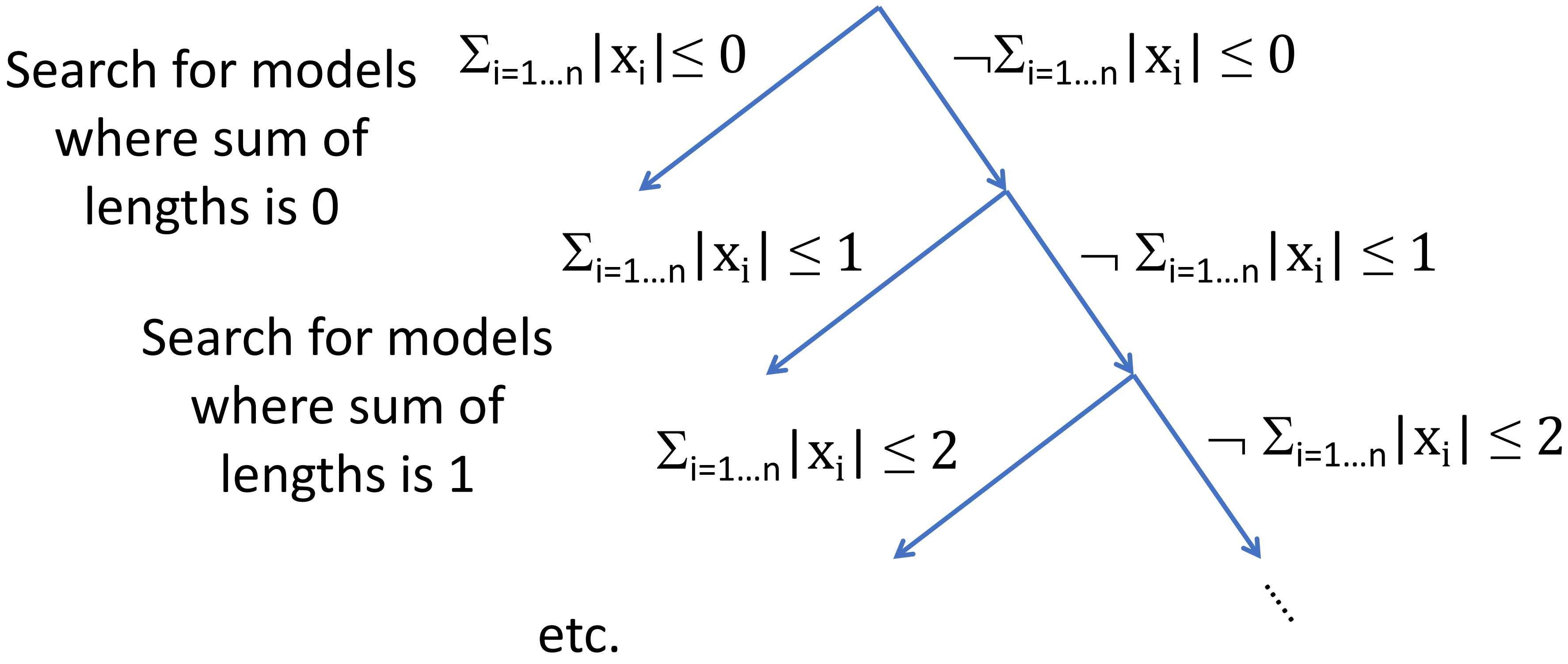
- Finite model finding for strings
- Context-dependent simplification for extended string constraints
- Regular expression elimination

Finite Model Finding for Strings

Finite Model Finding for Strings

Idea: Incrementally bound the lengths of input string variables $x_1, \ldots, x_n$

$\Rightarrow$ Improved solver's ability to answer "SAT" for problems with small models



Finite Model Finding

- Minimize sum of lengths $\Sigma_{i=1\ldots n} |x_i| \leq 0$
- Which variables have unbounded length?

$$x = \text{“ab”} \cdot z$$
$$x = y \cdot u \cdot v \vee u \neq \text{“abc”}$$
$$w = x \cdot \text{”ab”} \vee w = y \cdot \text{”cde”}$$

Finite Model Finding

- Minimize sum of lengths $\Sigma_{i=1\ldots n}|x_i| \leq 0$
- Which variables have unbounded length?

$$x = \text{“ab”}\cdot z$$
$$x = y\cdot u\cdot v \vee u \neq \text{“abc”}$$
$$w = x\cdot\text{”ab”} \vee w = y\cdot\text{”cde”}$$

- Can include a subset of the overall input variables in this sum
  - Above, upper bound on $|x + u|$ implies upper bounds on the length of z, y, w, v
- Reduces the overall sum of lengths

Context-Dependent Simplification for Extended String Constraints

Extended String Constraints

- *Basic* terms
  - String and integer variables, constants, concatenation, length, and LIA-terms
- *Extended* string terms:
  - Substring: $\mathrm{substr}(x, 1, 3)$

    (the substring of x starting at pos. 1 of length at most 3)
  - String contains: $\mathrm{contains}(x, \text{"abc"})$

    (true iff x contains the substring "abc")
  - Find "index of": $\mathrm{indexof}(x, \text{"d"}, 5)$

    (the pos. of the first occurrence of "d" in x, starting from position 5, or -1 if it does not exist)
  - String replace: $\mathrm{replace}(x, \text{"a"}, \text{"b"})$

    (the result of replacing the first occurrence of "a" in x, if any, with "b")

Example: $\neg\mathrm{contains}(\mathrm{substr}(x, 0, 3), \text{"a"}) \wedge 0 \leq \mathrm{indexof}(x, \text{"ab"}, 0) < 4$

Processing Extended String Constraints

$\neg$contains(x, “a”)

Processing Extended String Constraints

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

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

Processing Extended String Constraints

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

$\neg contains(x, \text{“a”})$

↓

$\forall 0 \leq n < |x|.\ substr(x, n, 1) \neq \text{“a”}$

Expand contains

Processing Extended String Constraints

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

$$\neg \text{contains}(x, \text{“a”})$$

$$\forall 0 \leq n < |x|.\ \text{substr}(x, n, 1) \neq \text{“a”}$$

Expand contains

$$\text{substr}(x, 0, 1) \neq \text{“a”} \wedge \ldots \wedge \text{substr}(x, 4, 1) \neq \text{“a”}$$

Assuming bound $|x| \leq 5$

Processing Extended String Constraints

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

$$\neg\text{contains}(x, \text{“a”})$$

$$\forall 0 \leq n < |x|.\ \text{substr}(x, n, 1) \neq \text{“a”}$$

Expand contains

$$\text{substr}(x, 0, 1) \neq \text{“a”} \land \ldots \land \text{substr}(x, 4, 1) \neq \text{“a”}$$

Assuming bound $|x| \leq 5$

$$x = z_{11} \cdot k_1 \cdot z_{21} \land |z_{11}| = 0 \land k_1 \neq \text{“a”} \land \quad \ldots \quad x = z_{14} \cdot k_4 \cdot z_{24} \land |z_{14}| = 4 \land k_4 \neq \text{“a”}$$

Expand substr

Processing Extended String Constraints

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

$$\neg\text{contains}(x, \text{“a”})$$

↓

$$\forall 0 \leq n < |x|.\ \text{substr}(x, n, 1) \neq \text{“a”}$$

Expand contains

↓

$$\text{substr}(x, 0, 1) \neq \text{“a”} \wedge \ldots \wedge \text{substr}(x, 4, 1) \neq \text{“a”}$$

Assuming bound $|x| \leq 5$

↓

$$x = z_{11} \cdot k_1 \cdot z_{21} \wedge \quad |z_{11}| = 0 \wedge \quad k_1 \neq \text{“a”} \wedge \quad \ldots \quad x = z_{14} \cdot k_4 \cdot z_{24} \wedge \quad |z_{14}| = 4 \wedge \quad k_4 \neq \text{“a”}$$

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

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

SAT Solver

Arithmetic Solver

String Solver

(Eager) Expansion of Extended Constraints

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

$x = z_{11} \cdot k_1 \cdot z_{21}$
$|z_{11}| = 0$
$k_1 \neq \text{“ a”}$

…

$x = z_{14} \cdot k_4 \cdot z_{24}$
$|z_{14}| = 4$
$k_4 \neq \text{“a”}$

$x = y \cdot \text{“d”}$
$y = \text{“ab”} \vee y = \text{“ac”}$

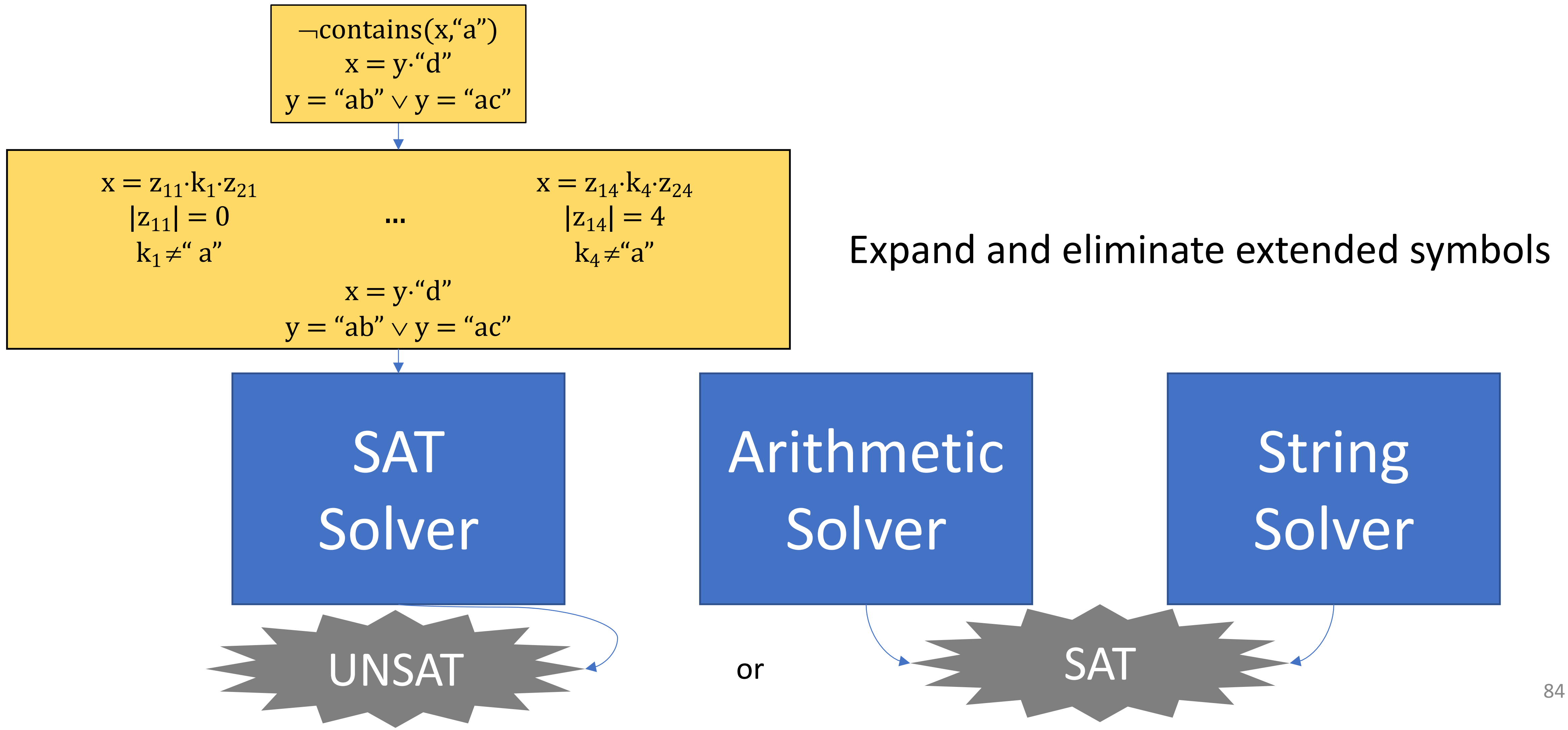
Expand and eliminate extended symbols

SAT Solver

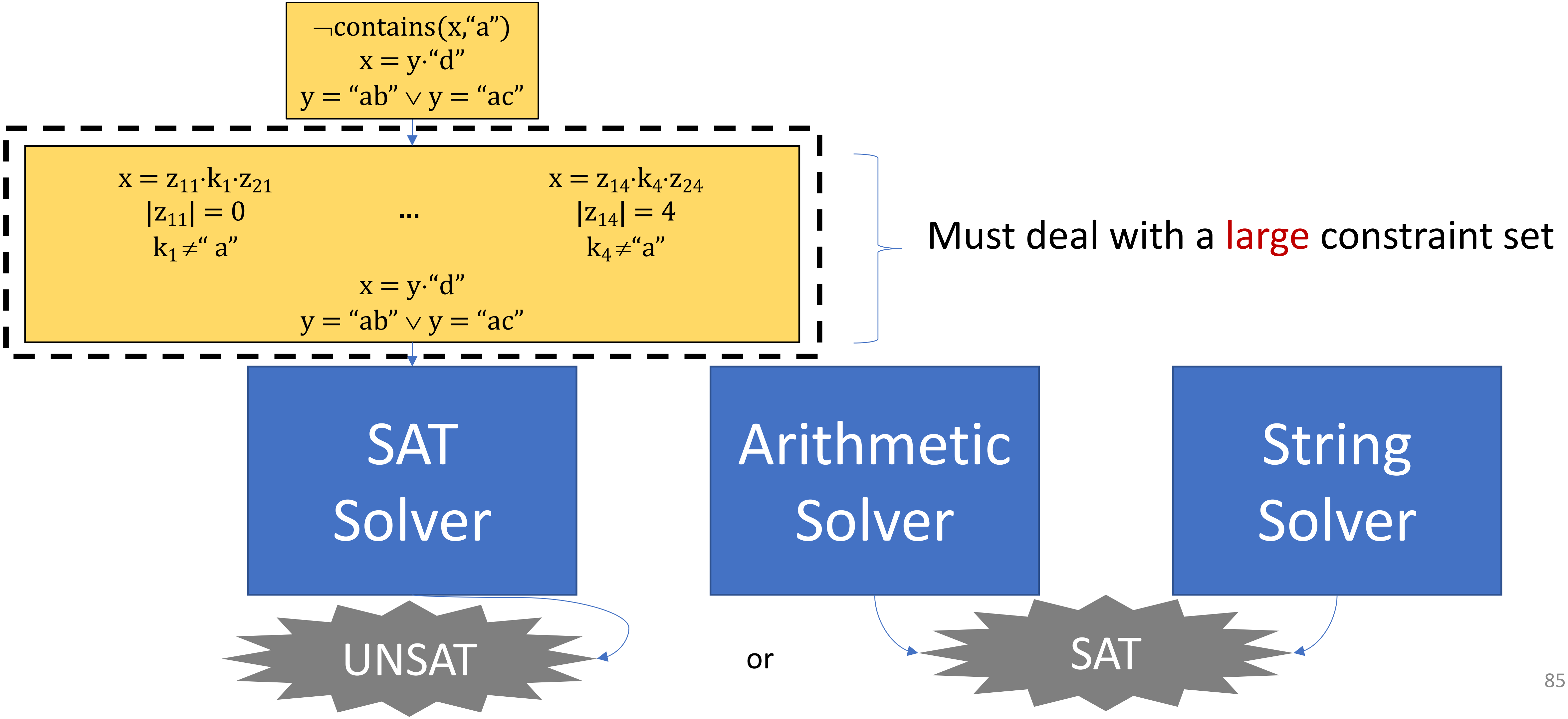
Arithmetic Solver

String Solver

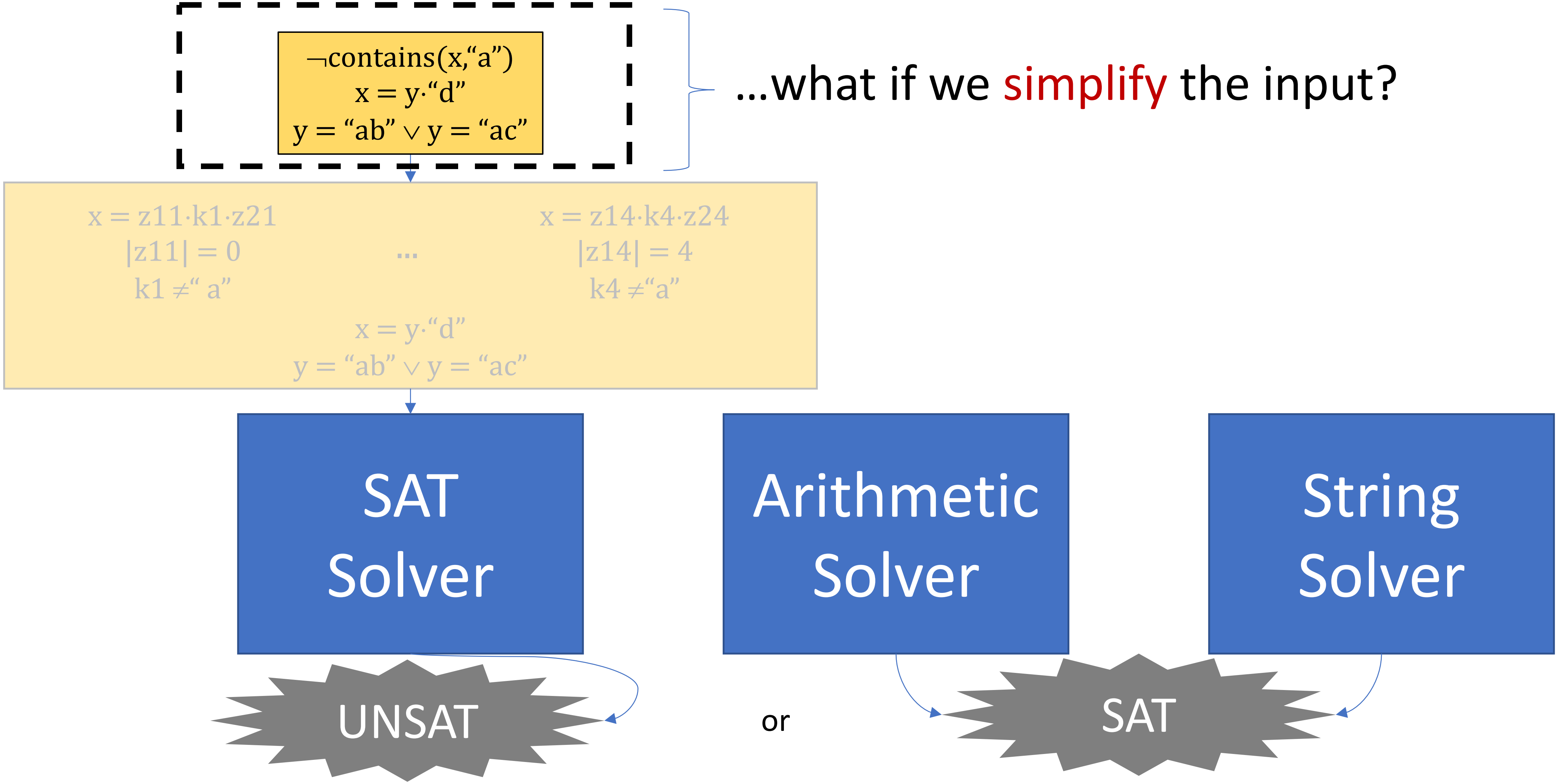
(Eager) Expansion of Extended Constraints



(Eager) Expansion of Extended Constraints



(Eager) Expansion of Extended Constraints



SMT Solvers + Simplification

All SMT solvers implement *simplification* techniques

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

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

SMT Solvers + Simplification

All SMT solvers implement *simplification* techniques

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

$\neg$contains(x, "a")
$x = y \cdot$"d"
$y =$ "ab" $\vee\ y =$ "ac"

↓

$\neg$contains($y \cdot$"d", "a")
$y =$ "ab" $\vee\ y =$ "ac"

since $x = y \cdot$"d"

SMT Solvers + Simplification

All SMT solvers implement *simplification* techniques

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

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

↓

$\neg\text{contains}(y \cdot \text{“d”}, \text{“a”}) \land$
$y = \text{“ab”} \lor y = \text{“ac”}$

since $x = y \cdot \text{”d”}$

↓

$\neg\text{contains}(y, \text{“a”}) \land$
$y = \text{“ab”} \lor y = \text{“ac”}$

since $\text{contains}(y \cdot \text{“d”}, \text{”a”}) \Leftrightarrow \text{contains}(y, \text{“a”})$

SMT Solvers + Simplification

All SMT solvers implement *simplification* techniques

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

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

↓

$$\neg\text{contains}(y \cdot \text{“d”}, \text{“a”}) \land \\ y = \text{“ab”} \lor y = \text{“ac”}$$

since $x = y \cdot \text{”d”}$

↓

$$\neg\text{contains}(y, \text{“a”}) \land \\ y = \text{“ab”} \lor y = \text{“ac”}$$

since $\text{contains}(y \cdot \text{“d”}, \text{”a”}) \Leftrightarrow \text{contains}(y, \text{“a”})$

Leads to smaller inputs

Some problems can be solved by simplification alone

(Lazy) Expansion + Simplification

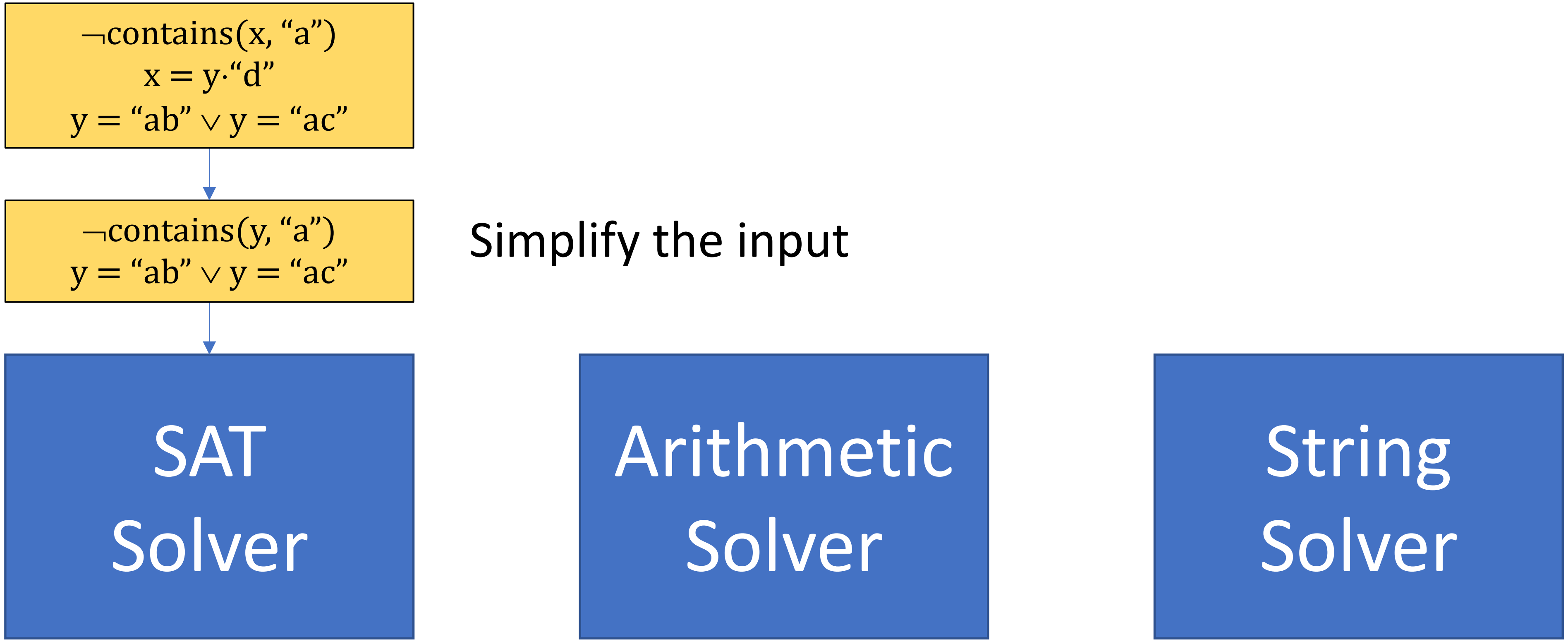
$\neg \text{contains}(x, \text{“a”})$
$x = y \cdot \text{“d”}$
$y = \text{“ab”} \vee y = \text{“ac”}$

SAT Solver

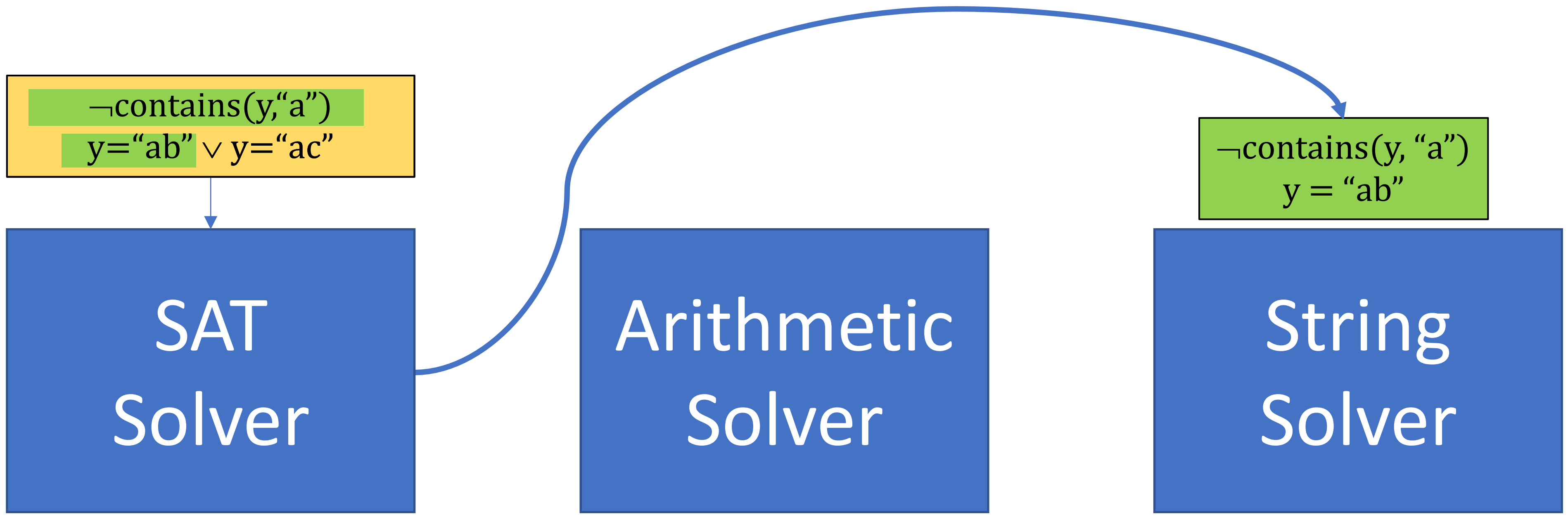
Arithmetic Solver

String Solver

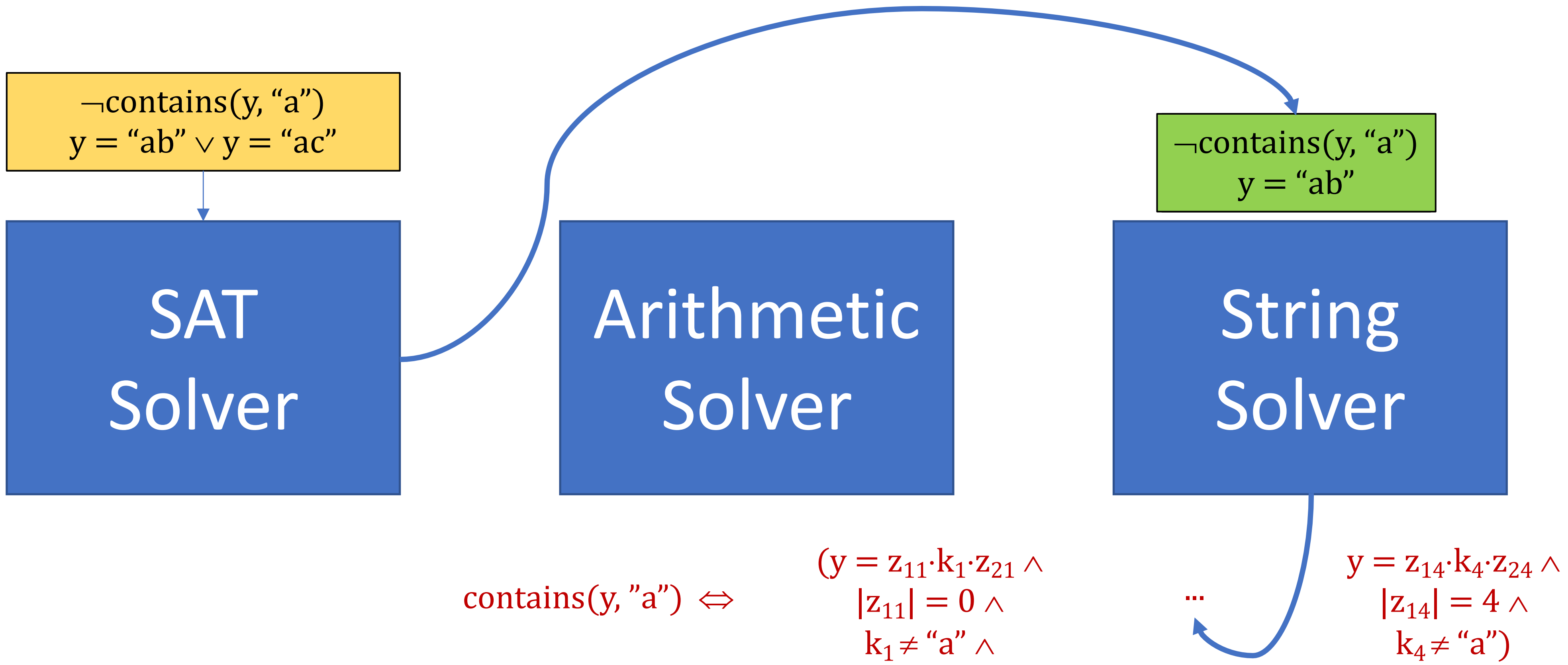
(Lazy) Expansion + Simplification



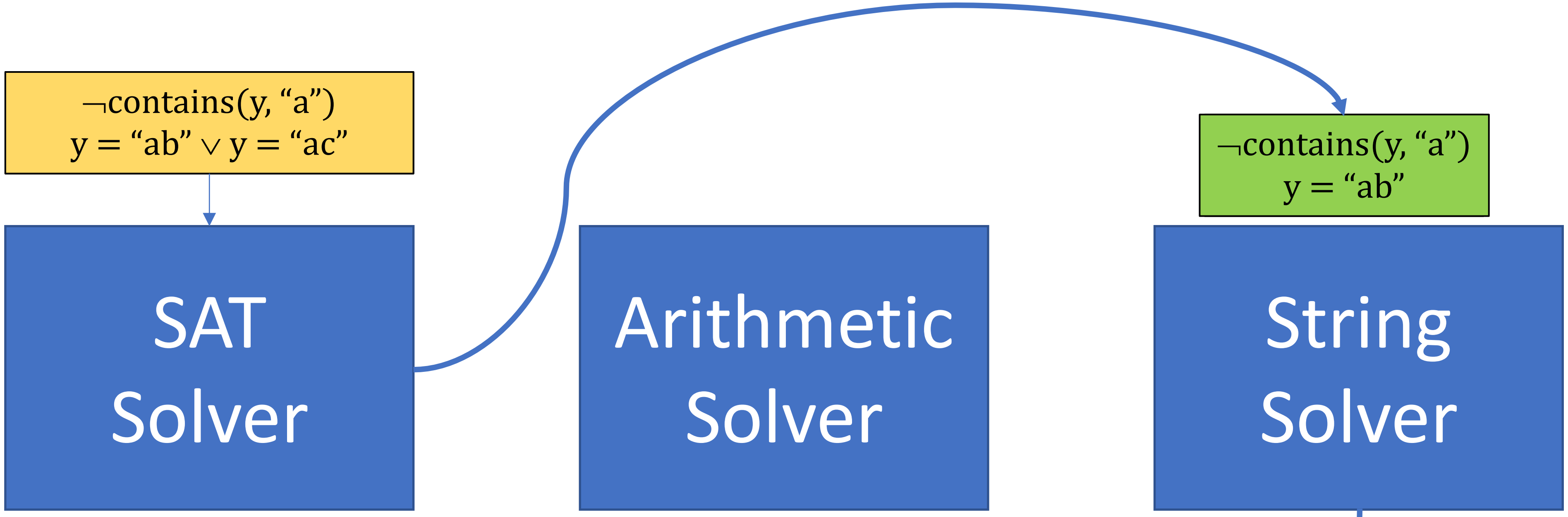
(Lazy) Expansion + Simplification



(Lazy) Expansion + Simplification



(Lazy) Expansion + Simplification

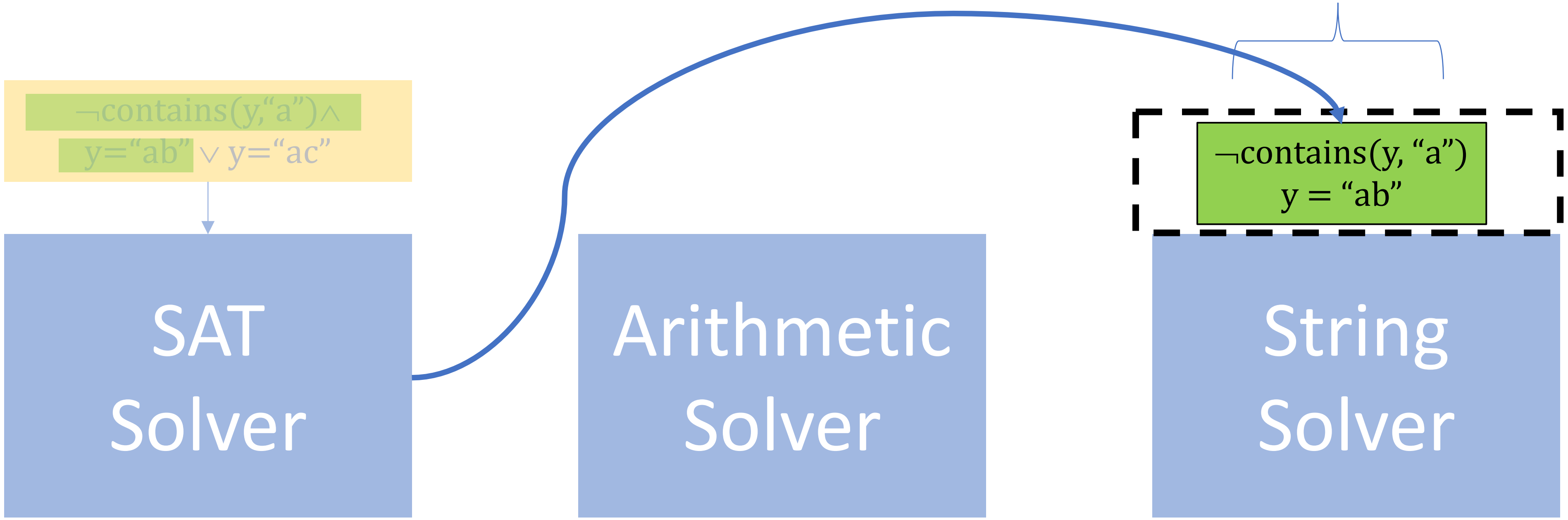


Still have a large constraint

$$\text{contains}(y, \text{"a"}) \Leftrightarrow (y = z_{11} \cdot k_1 \cdot z_{21} \wedge |z_{11}| = 0 \wedge k_1 \neq \text{"a"} \wedge \; \ldots \; y = z_{14} \cdot k_4 \cdot z_{24} \wedge |z_{14}| = 4 \wedge k_4 \neq \text{"a"})$$

(Lazy) Expansion + Simplification

What if we simplify based on the context?

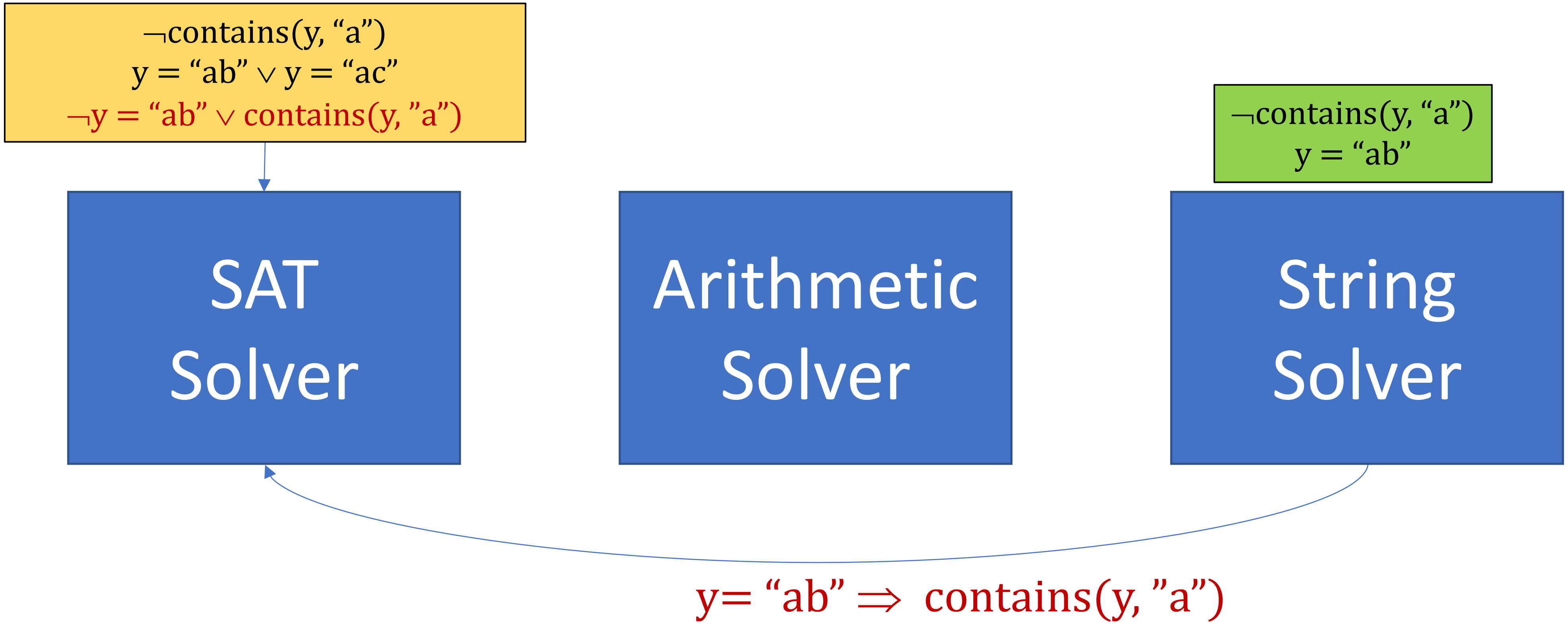


$\text{contains}(y, \text{"a"}) \Leftrightarrow$ $(y = z_{11} \cdot k_1 \cdot z_{21} \wedge |z_{11}| = 0 \wedge k_1 \neq \text{"a"} \wedge$ … $y = z_{14} \cdot k_4 \cdot z_{24} \wedge |z_{14}| = 4 \wedge k_4 \neq \text{"a"})$

(Lazy) Expansion + Context-Dependent Simplification

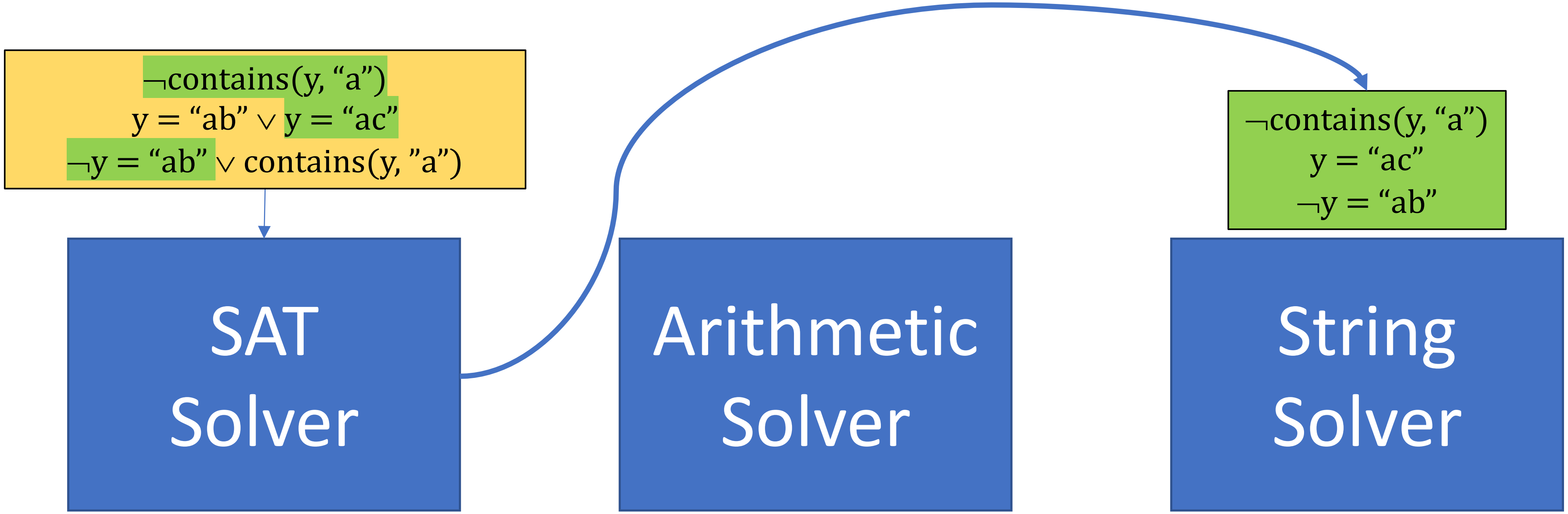
Since contains(y, ”a”) is true when y = “ab” …

(Lazy) Expansion + Context-Dependent Simplification

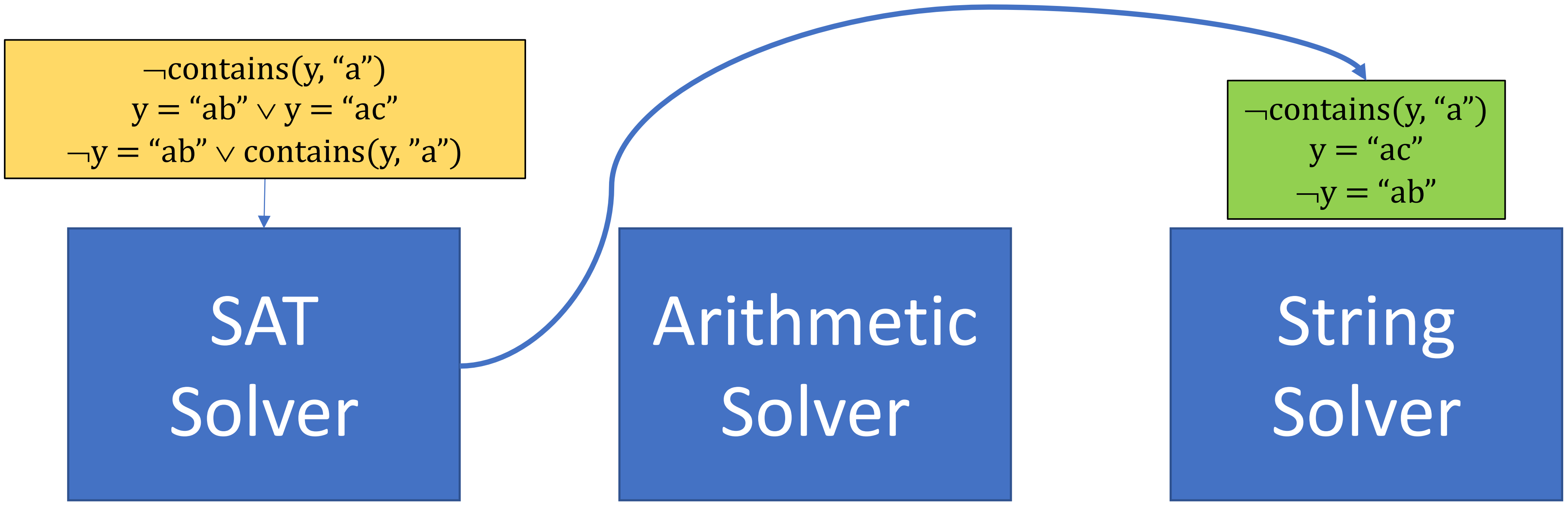


(Lazy) Expansion + Context-Dependent Simplification

(Lazy) Expansion + Context-Dependent Simplification

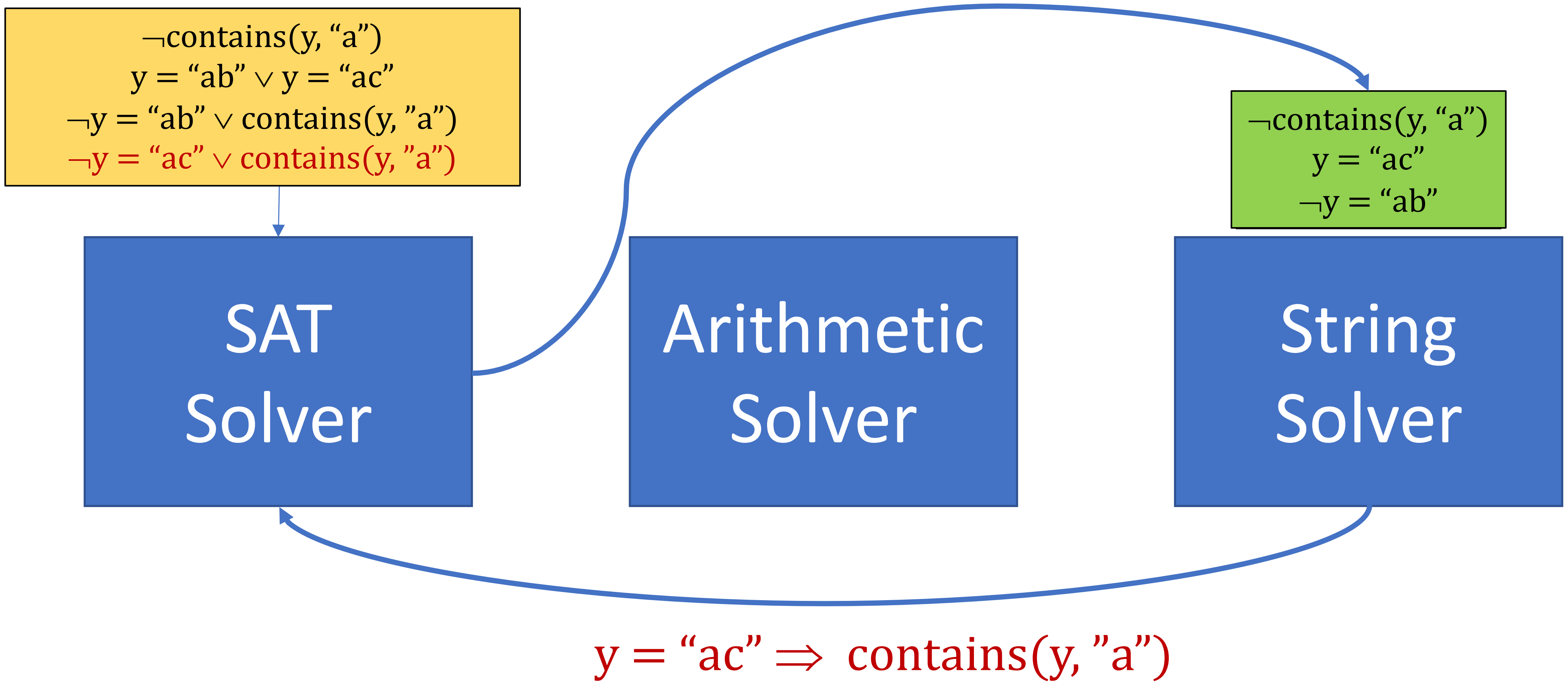


(Lazy) Expansion + Context-Dependent Simplification

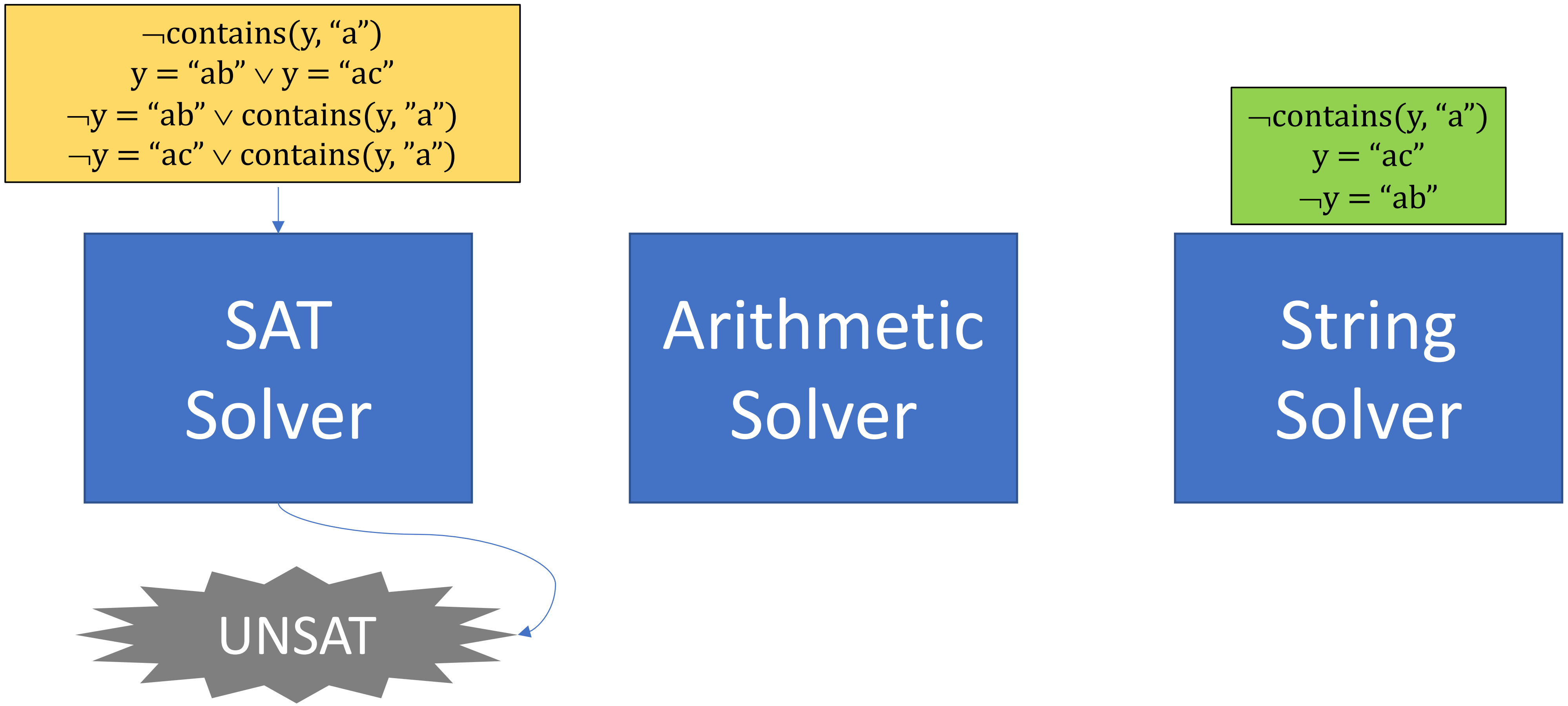


contains(y, ”a”) is also true when y = “ac” …

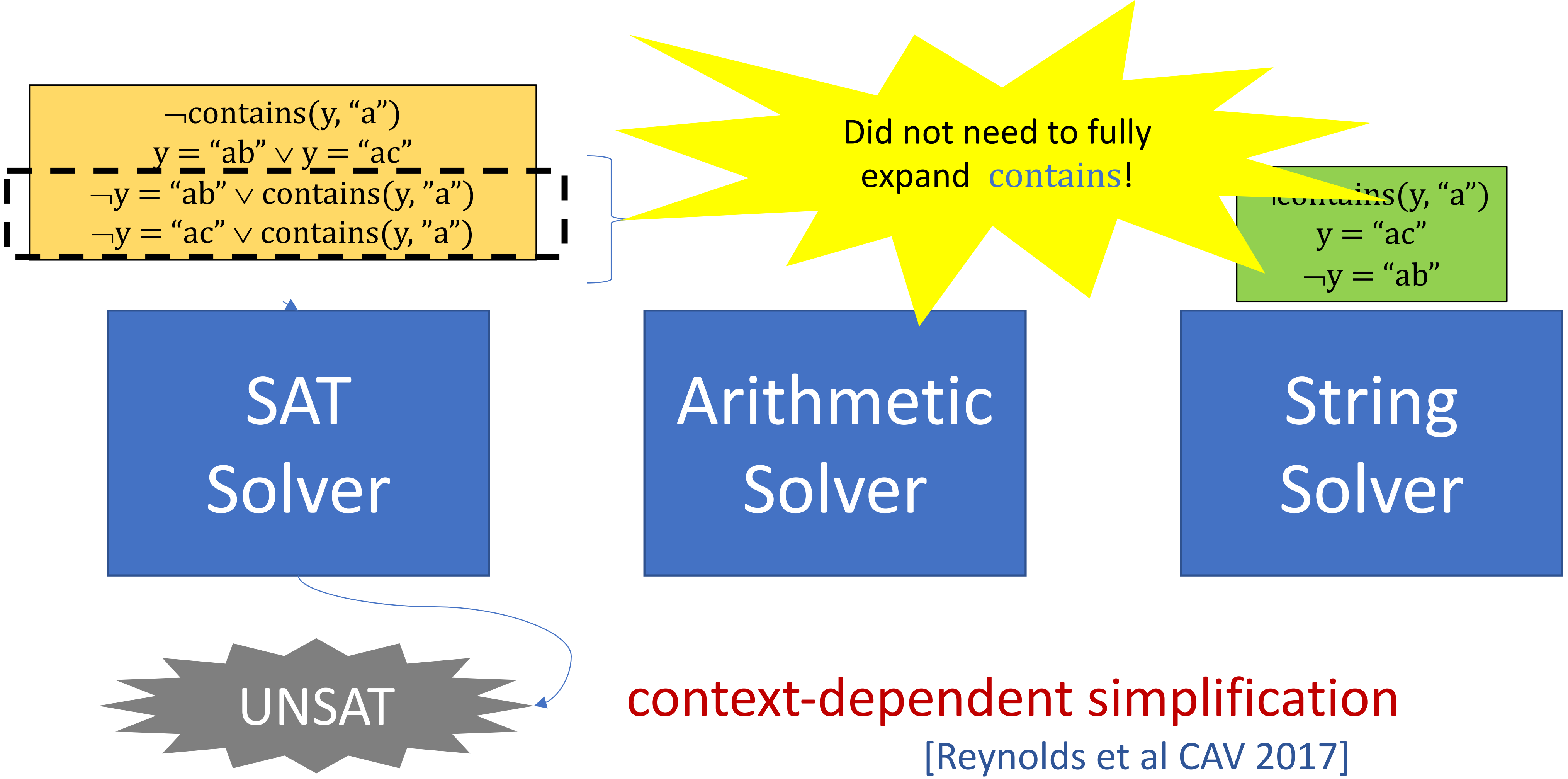
(Lazy) Expansion + Context-Dependent Simplification



(Lazy) Expansion + Context-Dependent Simplification



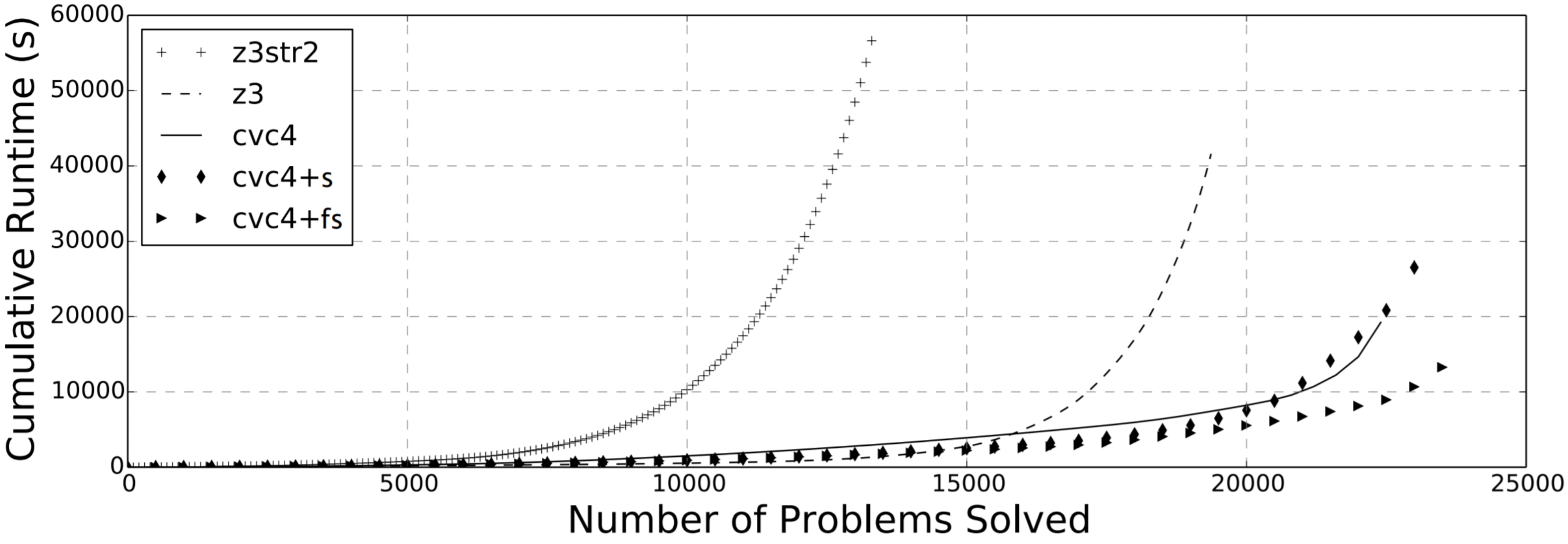
(Lazy) Expansion + Context-Dependent Simplification



context-dependent simplification

[Reynolds et al CAV 2017]

Results on Symbolic Execution [Reynolds et al. CAV 17]



- cvc4+fs (finite model finding + context-dependent simpl.) solves 23,802 benchmarks in 5h8m
- Without finite model finding, solves 23,266 benchmarks in 8h46m
- Without either finite model finding or cd-simplification, solves 22,607 benchmarks in 6h38m

Many 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”) → $\bot$

contains(x·“ac”·y, “b”) → contains(x, “b”) $\vee$ contains(y, “b”)

indexof(“abc”·x, “a”·x,1) → -1

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

Simplification based on High-Level Abstractions

[Reynolds et al. CAV 19]

Rules based on high-level abstractions

- When viewing strings as #characters (e.g. reasoning about their length):

$\text{contains}(\text{substr}(x, i, j), x \cdot \text{"a"}) \rightarrow \text{""}$

since the second argument is longer than the first

- When considering the containment relationship between strings:

$\text{contains}(\text{replace}(x, y, z), z) \rightarrow \text{contains}(x, y) \vee \text{contains}(x, z)$

- When viewing strings as multisets of characters:

$x \cdot x \cdot y \cdot \text{"ab"} = x \cdot \text{"bbbbbb"} \cdot y \rightarrow \bot$

since LHS contains at least 1 more occurrences of “a”

Impact of Aggressive Simplification

[Reynolds et al. CAV 19]

Set		all	-arith	-contain	-msets	Z3	OSTRICH
CMU	sat	7947	7746	**7948**	7946	4585	
	unsat	**66**	31	**66**	**66**	52	
	×	173	409	172	174	3549	
TERMEQ	sat	**10**	**10**	**10**	**10**	1	
	unsat	**49**	36	27	**49**	36	
	×	22	35	44	22	44	
SLOG	sat	**1302**	**1302**	**1302**	**1302**	1100	1289
	unsat	**2082**	**2082**	**2082**	**2082**	2075	**2082**
	×	7	7	7	7	216	20
APLAS	sat	**132**	**132**	**132**	**132**	10	
	unsat	**292**	291	171	171	94	
	×	159	160	280	280	479	
Total	sat	9391	9190	**9392**	9390	5696	1289
	unsat	**2489**	2440	2346	2368	2257	2082
	×	361	611	503	483	4288	8870

-arith: w/o arithmetic simplifications
-contain: w/o contain-based simplifications
-mset: w/o multiset-based simplifications

CVC4 implements >3000 lines of C++ for simplification rules (and growing)

Important aspect of modern string solving

Regular Expression Elimination

Regular Expression Elimination

CVC4 supports regular expressions, via:

- Decomposing memberships

  E.g. $x \in R_1 \cup R_2 \Rightarrow x \in R_1 \vee x \in R_2$, $x \in R_1 \cap R_2 \Rightarrow x \in R_1 \wedge x \in R_2$

- Intersection (modulo equality):

  E.g. $(x \in R_1 \wedge y \in R_2 \wedge x = y) \Rightarrow x \in$ compute_intersection(R_1, R_2)

- Unfolding

  E.g. $x \in R^* \Rightarrow x = "" \vee (x = x_1 \cdot x_2 \wedge x_1 \in R \wedge x_2 \in R^*)$ for fresh x_1, x_2

- Elimination based on reduction to extended string constraints

Regular Expression Elimination

Idea: reduce RE to extended string constraints

Possible for many regular expression memberships that occur in practice:

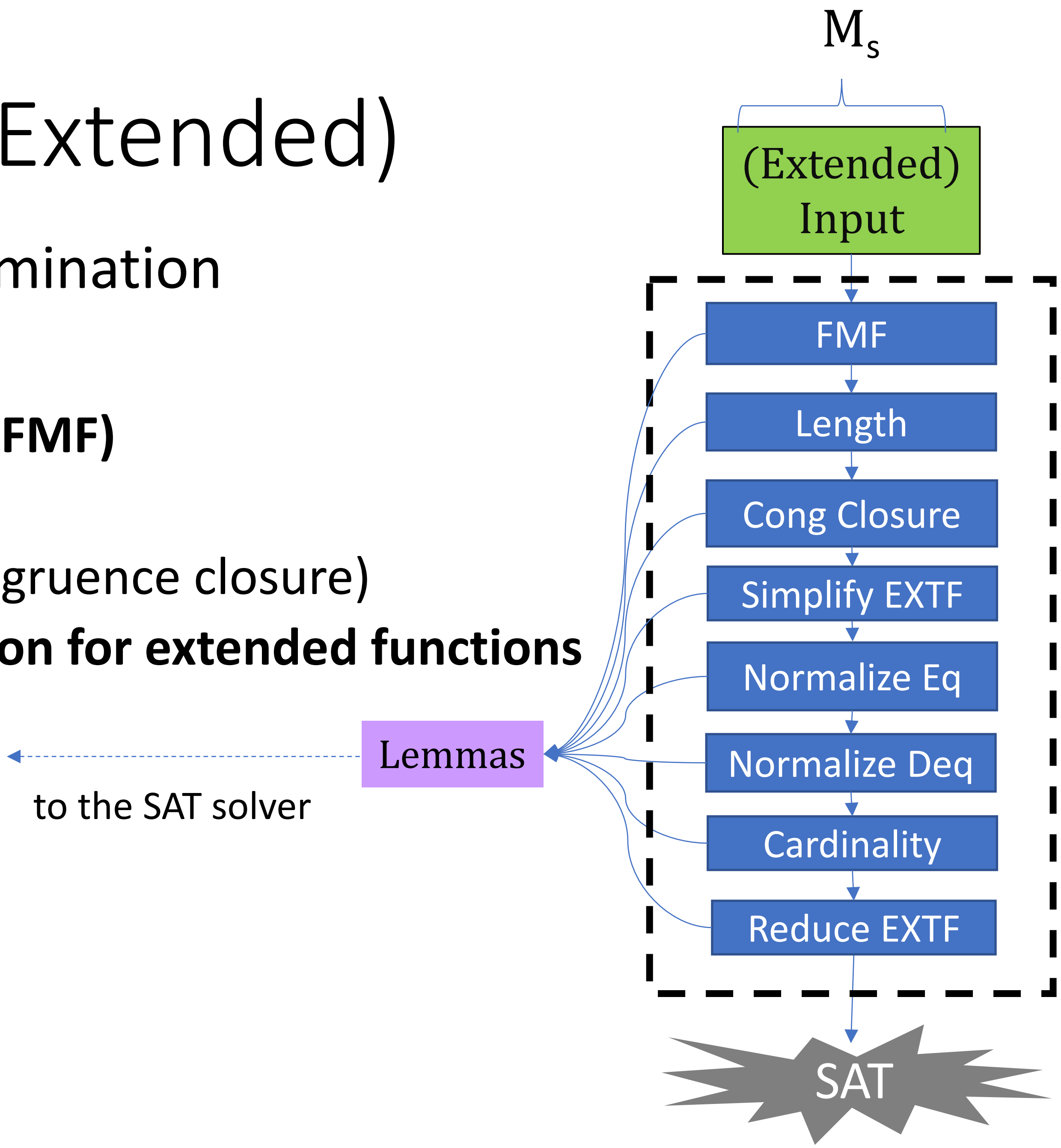
$x \in A\cdot A^*\cdot A$	$\Leftrightarrow$	$\lvert x\rvert \geq 2$
$x \in A^*\cdot$"abc"$\cdot A^*$	$\Leftrightarrow$	contains(x, "abc")
$x \in A^*\cdot$"a"$\cdot A^*\cdot$"bcd"$\cdot A^*$	$\Leftrightarrow$	contains(x, "a") $\wedge$ contains(substr(x, indexof(x, "a", 1) + 1, $\lvert x\rvert$), "bcd")

CVC4 supports (aggressive) elimination techniques for RE like those above

Utilizes existing support for extended functions

String Theory Solver (Extended)

- Preprocess based on reg-exp elimination
- Then, run inference strategy:
  1. **Split on sum of lengths bound (FMF)**
  2. Process length constraints
  3. Check for equality conflicts (congruence closure)
  4. **Context-dependent simplification for extended functions**
  5. Normalize string equalities
  6. Normalize string disequalities
  7. Check cardinality constraints
  8. **Reduce extended functions**



Conclusions

- CVC4 supports DPLL(T) theory solver for strings and regular expressions
  - Efficient in practice (incomplete) procedure for word equations with length
  - More advanced features like FMF, context-dependent simplification, RE elimination
  - Also supports: str.code, str.<=, str.to-int, str.from-int, str.replaceall
- Open-source, available at https://cvc4.github.io/

Thanks for listening