The Rise of SMT Solvers for String Constraints

Andrew Reynolds

TPSS workshop

July 19, 2021

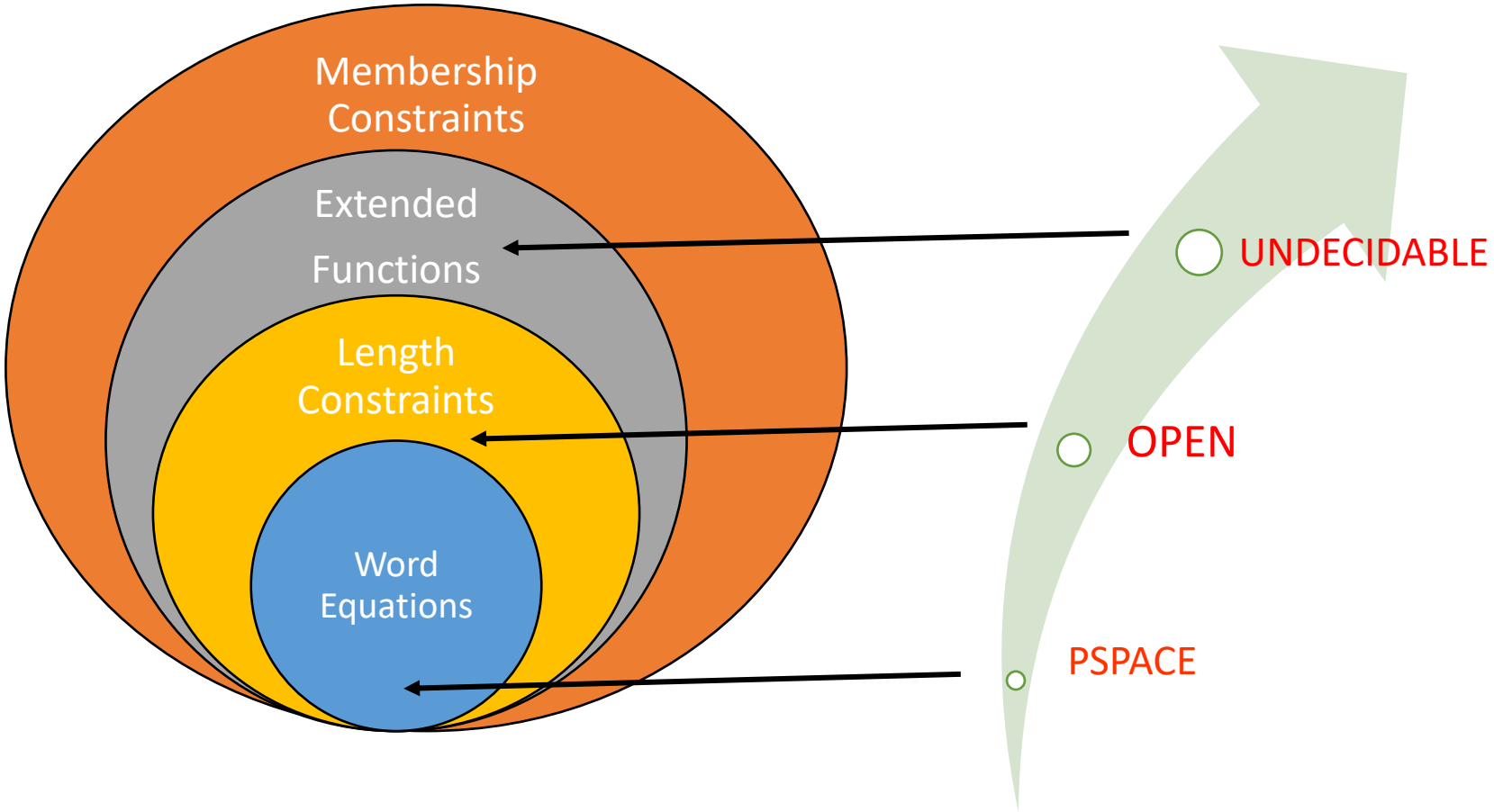
Satisfiability Modulo Theories (SMT) Solvers

- Many applications:
  - Software verification
  - Automated theorem proving
  - Symbolic execution
  - Security analysis

- **Traditionally:**
  - Efficient solvers for quantifier-free constraints over (combinations of) theories
    - Arithmetic, Arrays, Bit vectors

- **In this talk:**
  - SMT solvers for *string* constraints

Strings and RegExp: Theoretical Challenges



- Many applications require *extended string functions* and *RegEx memberships*
  - `to_int(x)≠44, to_lower(x)="abc", x∈range("A","Z")`

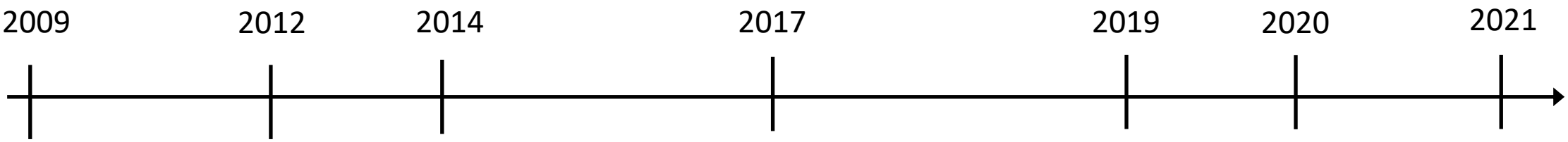
The CVC~~4~~5 SMT Solver

- Support for many theories and features
  - UF, (non)linear arithmetic, arrays
  - Bit-vectors, floating points
  - Sets, relations, 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, Andres Noetzli

Stanford
University

Timeline



CVC4 project begins

First release CVC4 1.0

CVC4 theory solver for strings + RE

[Liang et al CAV 14]

Context-Dependent Simplification

[Reynolds et al CAV 17]

Aggressive Rewriting

[Reynolds et al CAV 19]

Witness sharing, RE-elim

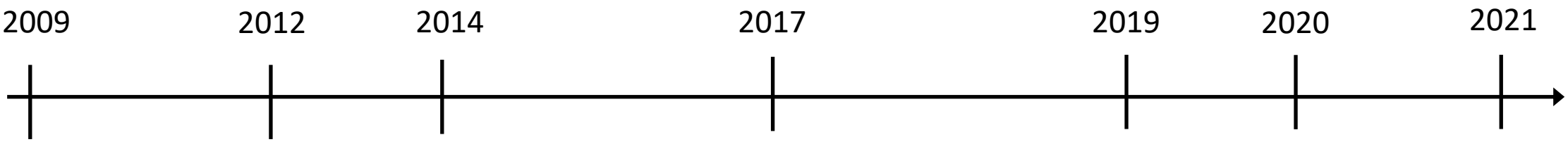
[Reynolds et al FMCAD 20]

String-to-Code Conversion

[Reynolds et al IJCAR 20]

CVC4 -> CVC5

Timeline



CVC4 project begins

First release CVC4 1.0

CVC4 theory solver for strings + RE

[Liang et al CAV 14]

Context-Dependent Simplification

[Reynolds et al CAV 17]

Aggressive Rewriting

[Reynolds et al CAV 19]

Witness sharing, RE-elim

[Reynolds et al FMCAD 20]

String-to-Code Conversion

[Reynolds et al IJCAR 20]

CVC4 -> CVC5

Other SMT String Solvers

Z3Str

S3
Norn

Z3Str2

Z3Seq

Trau

Z3Str3

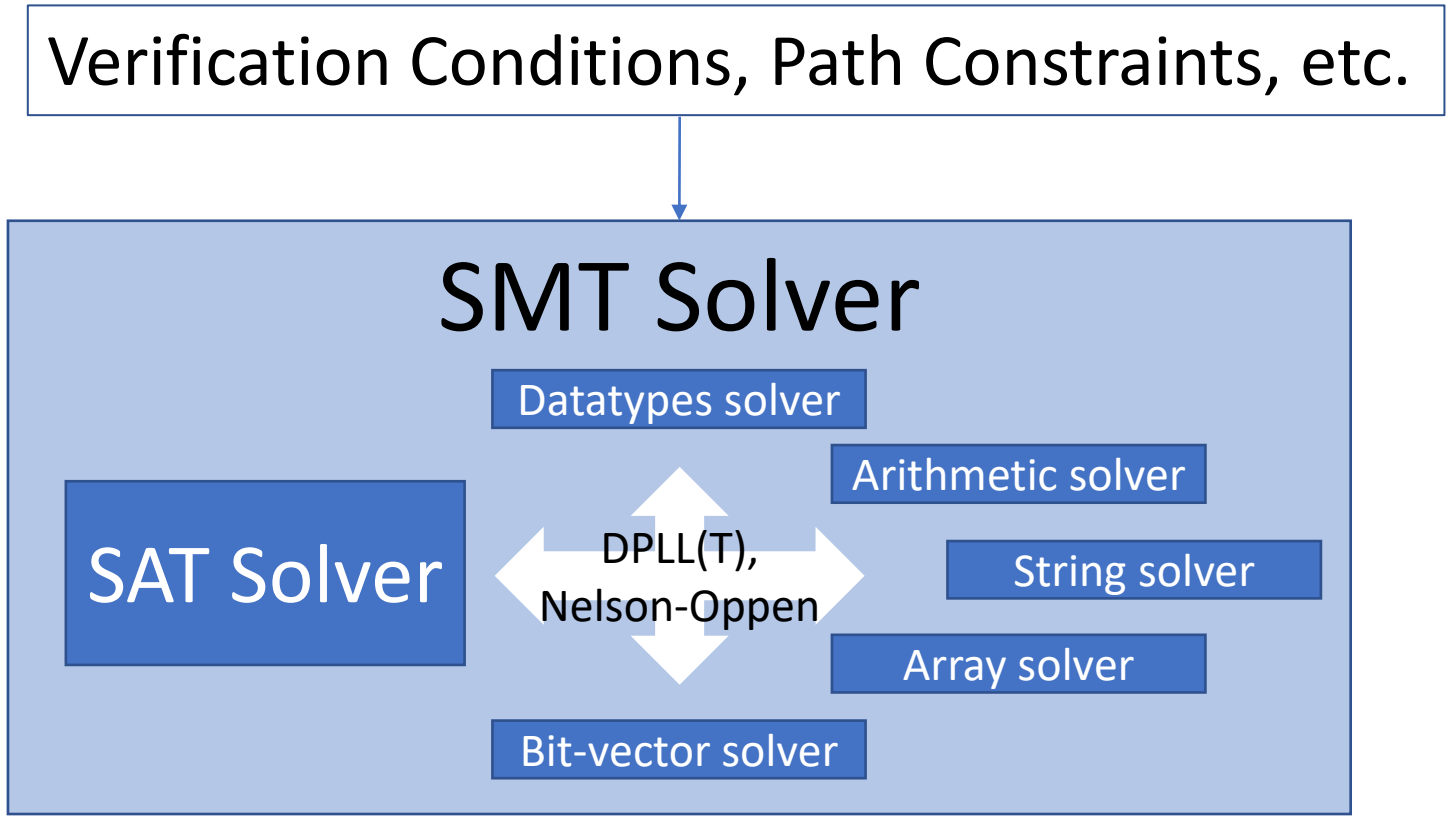
Z3Str4

….

SMT Solvers and DPLL(T)

SMT solvers

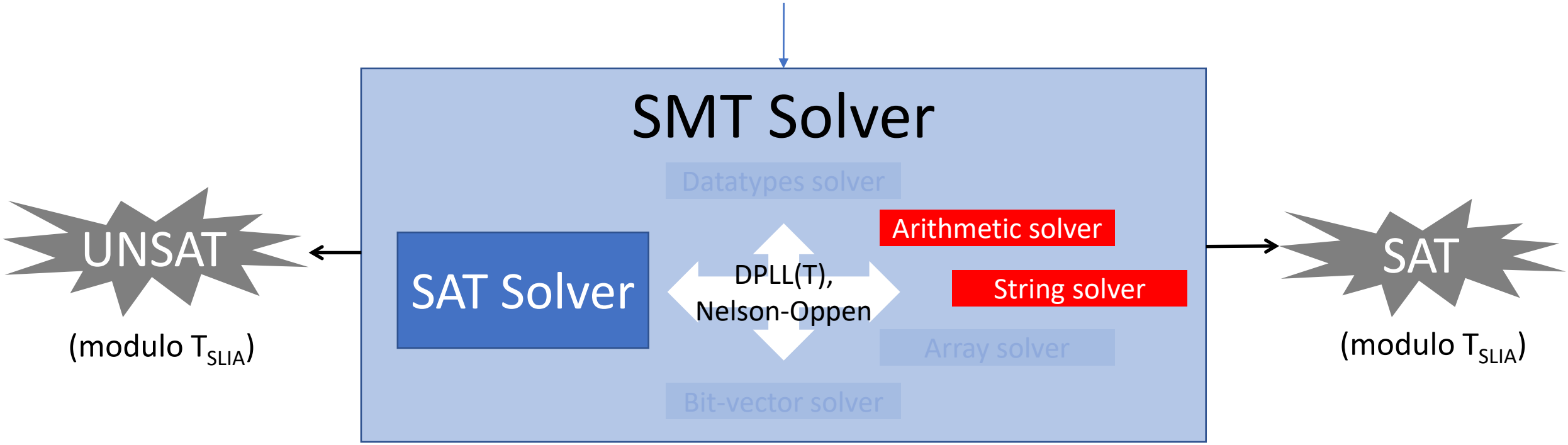
- Efficient tools for satisfiability *modulo theories*



SMT solvers

- Efficient tools for satisfiability *modulo 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 Σ^* for finite alphabet Σ
- Terms:
  - String Variables: `x, y, z`
  - Integer Variables: `i, j, k`
  - String Constants: ε, `"abc", "AAAAA", "http"`
  - String Concatenation: `x·"abc", x·y·z·w`
  - String Length: `|x|`
- Formulas are:
  - Equalities and disequalities between string terms
  - *Linear* arithmetic constraints: `|x|+4>|y|`

Example: `x·"a"=y, y ≠"b"·z, |y|>|x|+2`

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

DPLL(T) String Solvers

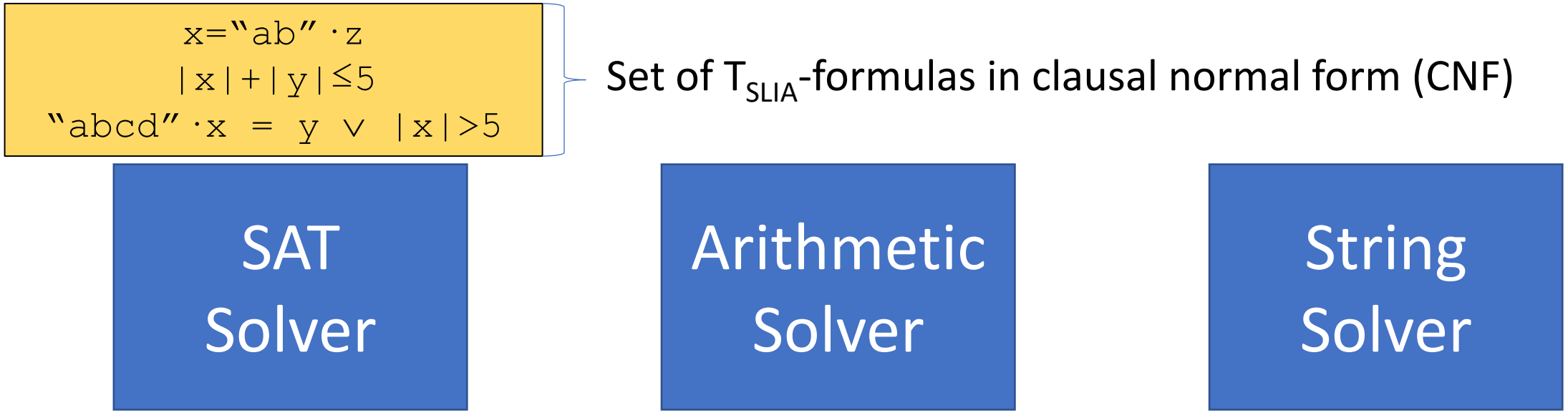
- Cooperation between:

SAT Solver

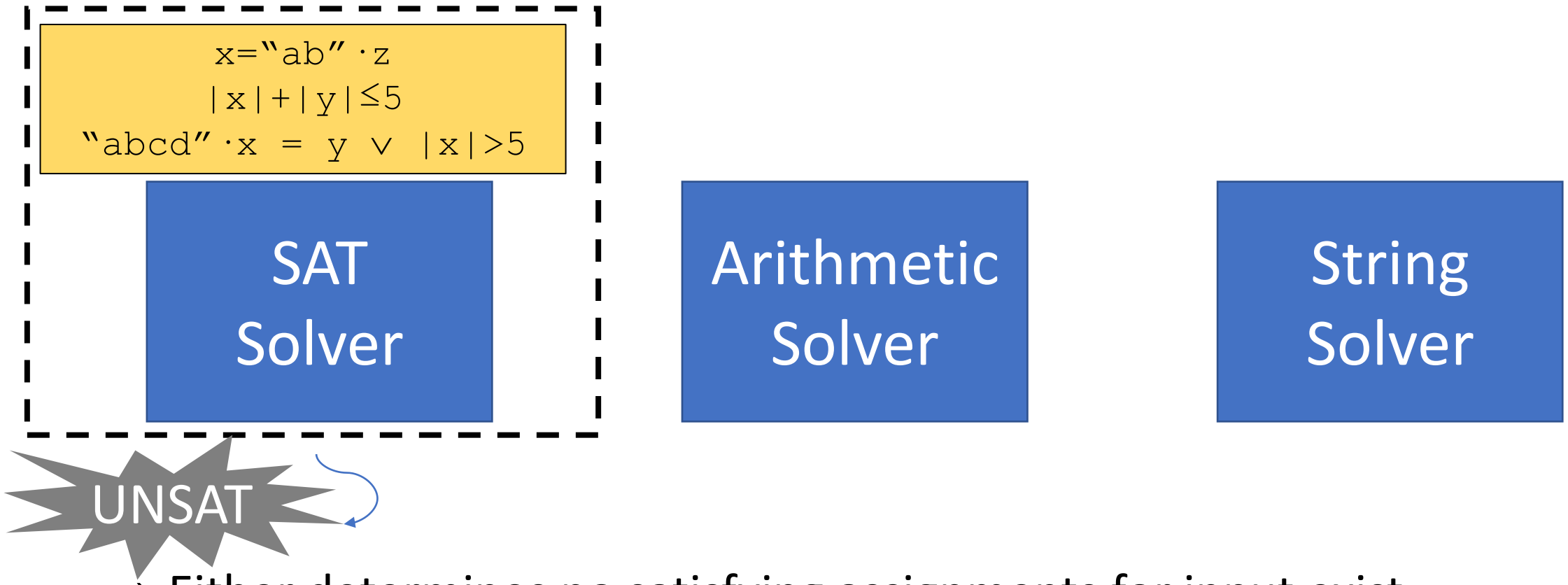
Arithmetic Solver

String Solver

DPLL(T) String Solvers



DPLL(T) String Solvers

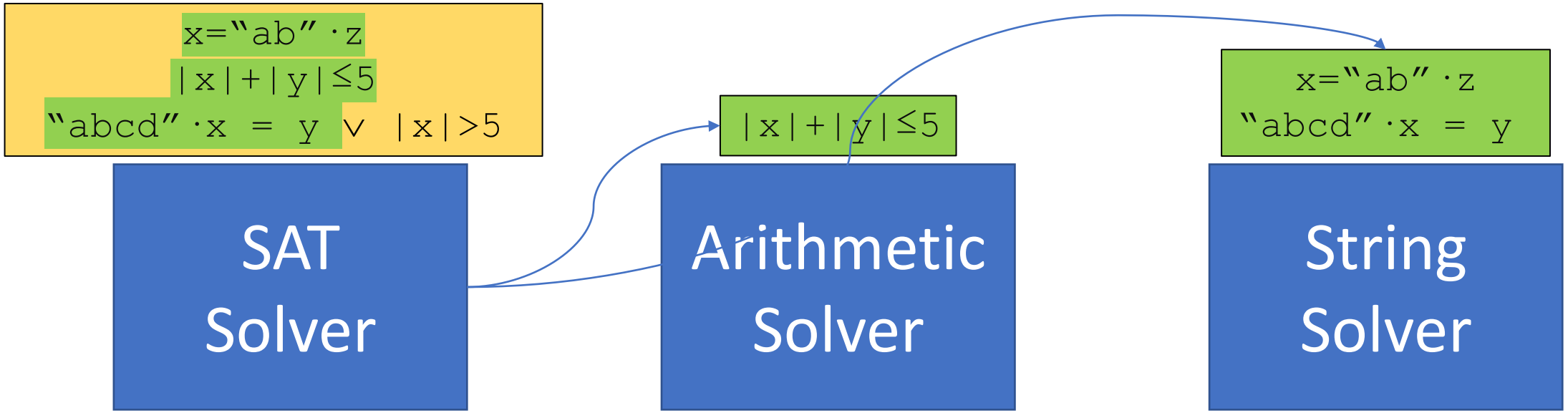


⇒ Either determines no satisfying assignments for input exist

DPLL(T) String Solvers

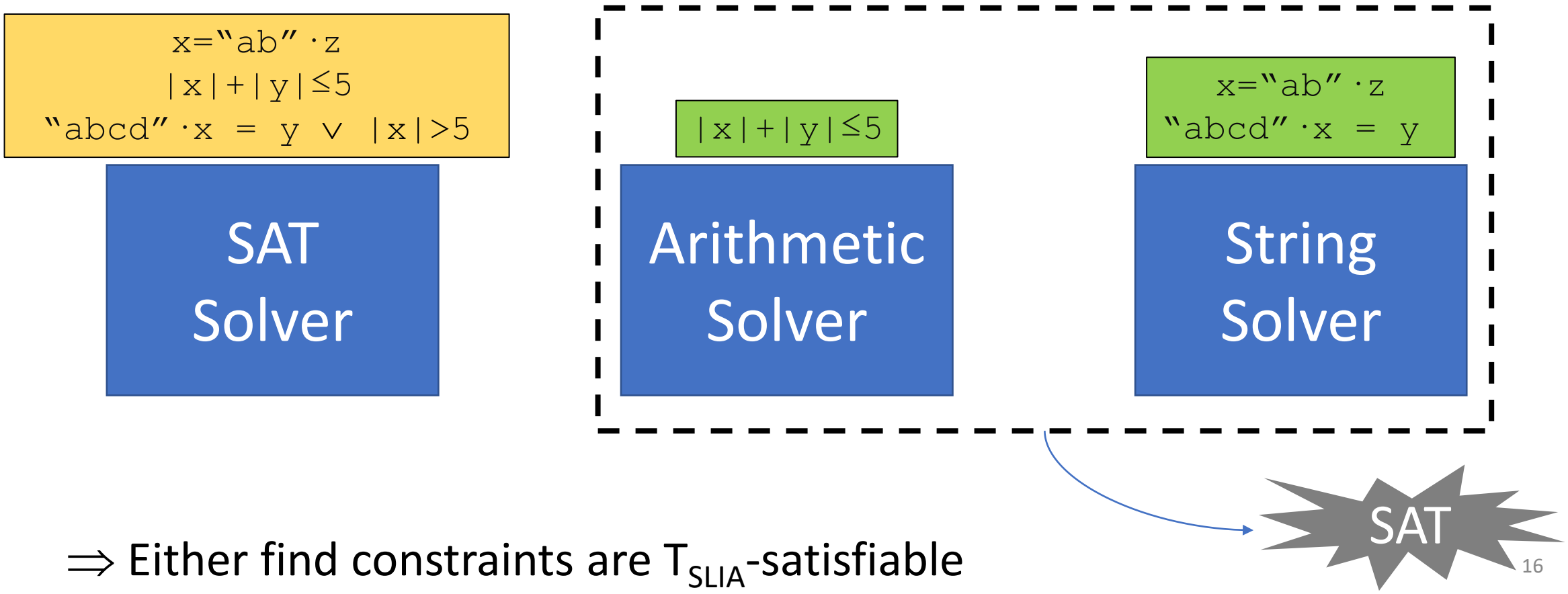
⇒ …or returns a propositionally satisfying assignment

DPLL(T) String Solvers



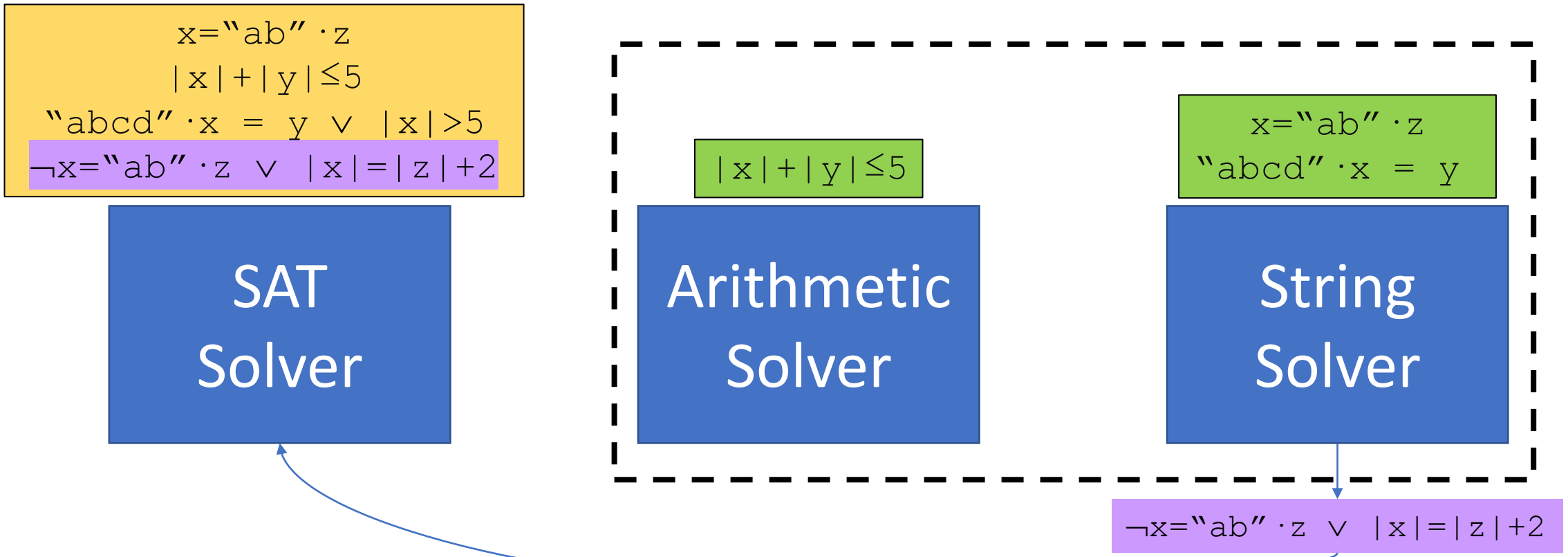
⇒ Constraints distributed to arithmetic and string solvers

DPLL(T) String Solvers



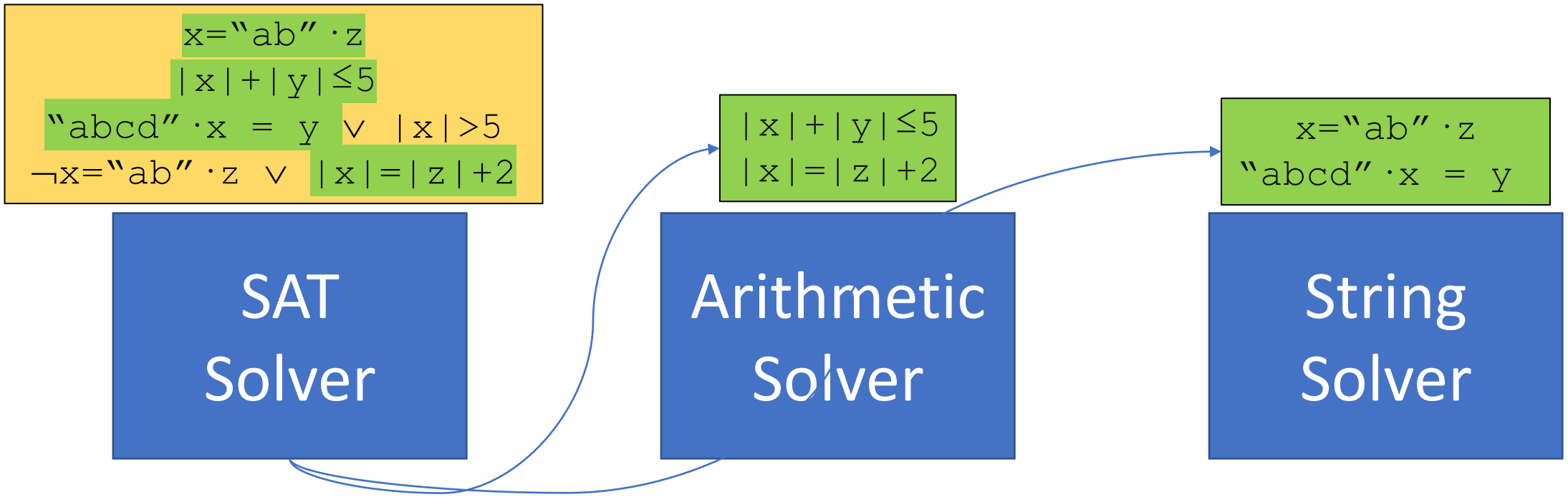
⇒ Either find constraints are T_{SLIA}-satisfiable

DPLL(T) String Solvers



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

DPLL(T) String Solvers



⇒ …and repeat

A Theory Solver for Strings

[Liang/Reynolds/Deters/Tinelli/Barrett CAV 14]

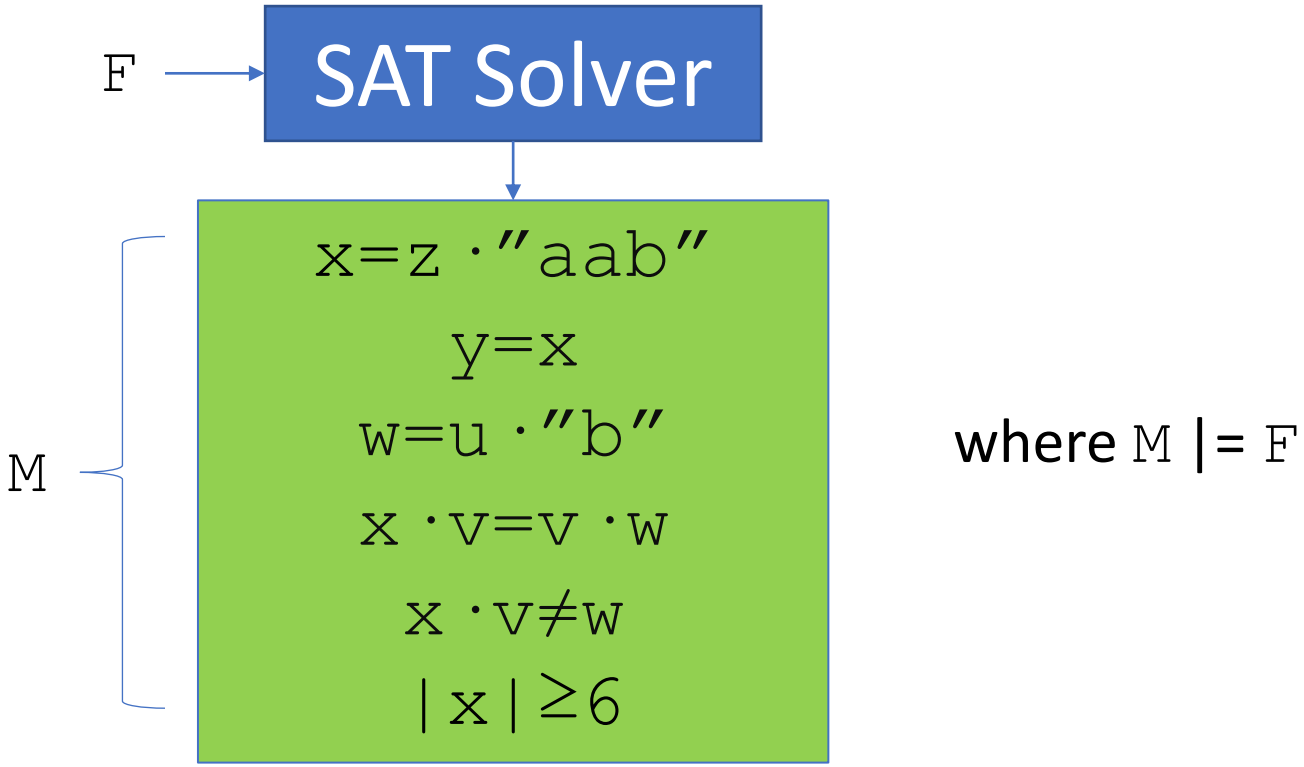
Solving Strings in CVC5

- Cooperation between:
  - SAT solver
  - Arithmetic theory solver

  $\Rightarrow$ *String theory solver*

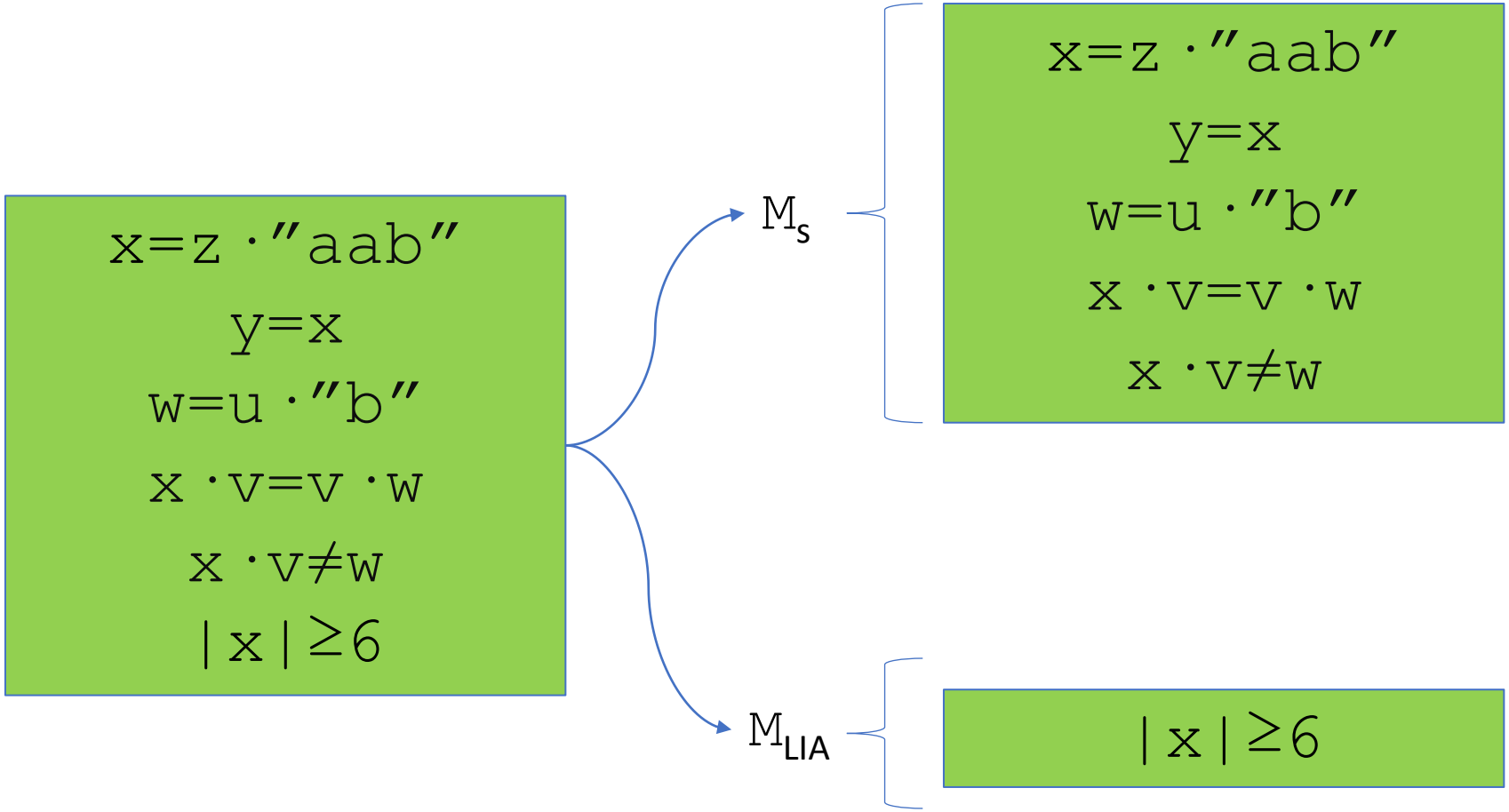
- Overall properties:
  - **Sound** (“UNSAT” can be trusted)
  - **Model-sound** (“SAT” can be trusted)
  - Not necessarily **terminating**
    - …due to “looping” word equations $x \cdot s = t \cdot x$
- Nevertheless, highly effective in practice

String Solver

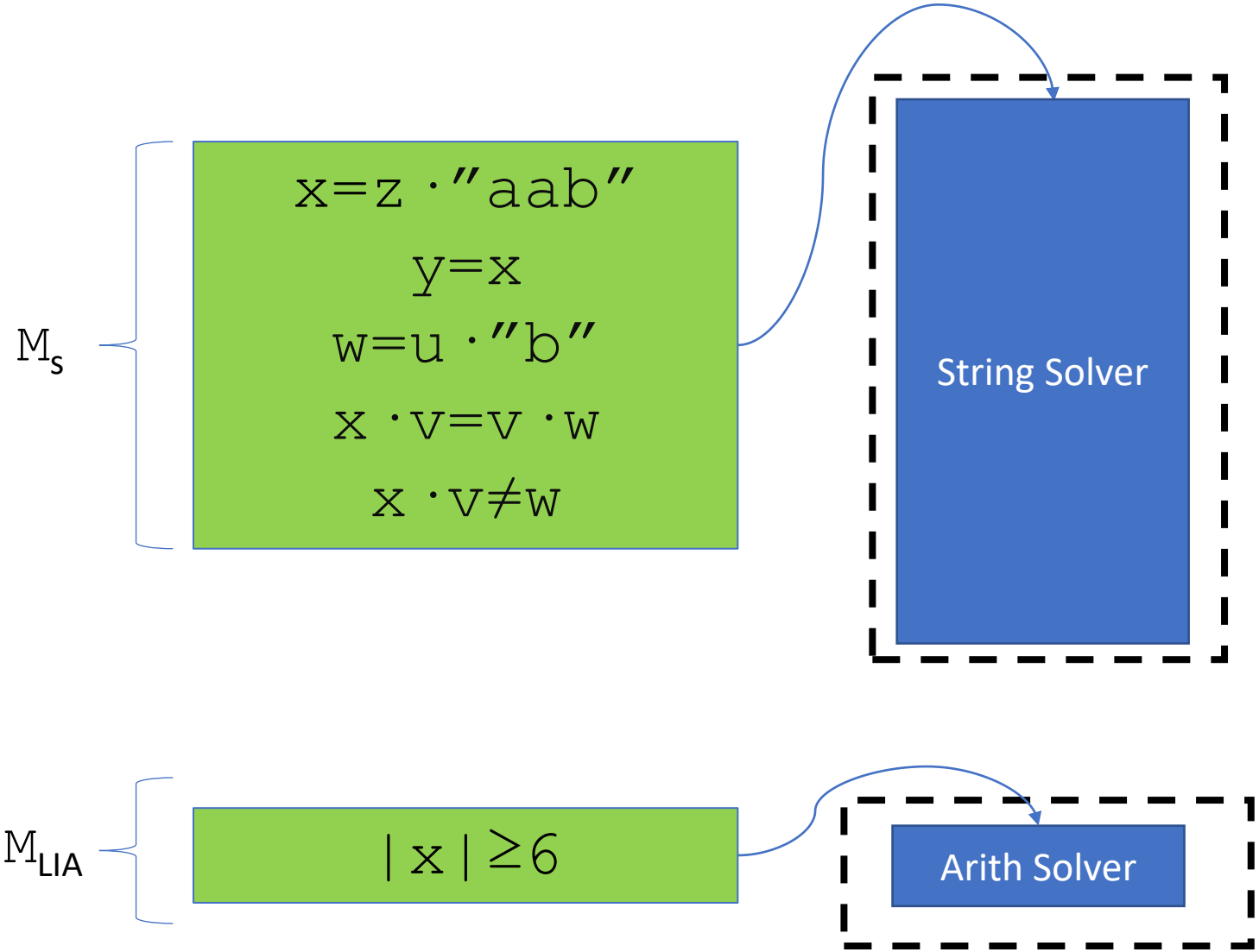


where M |= F

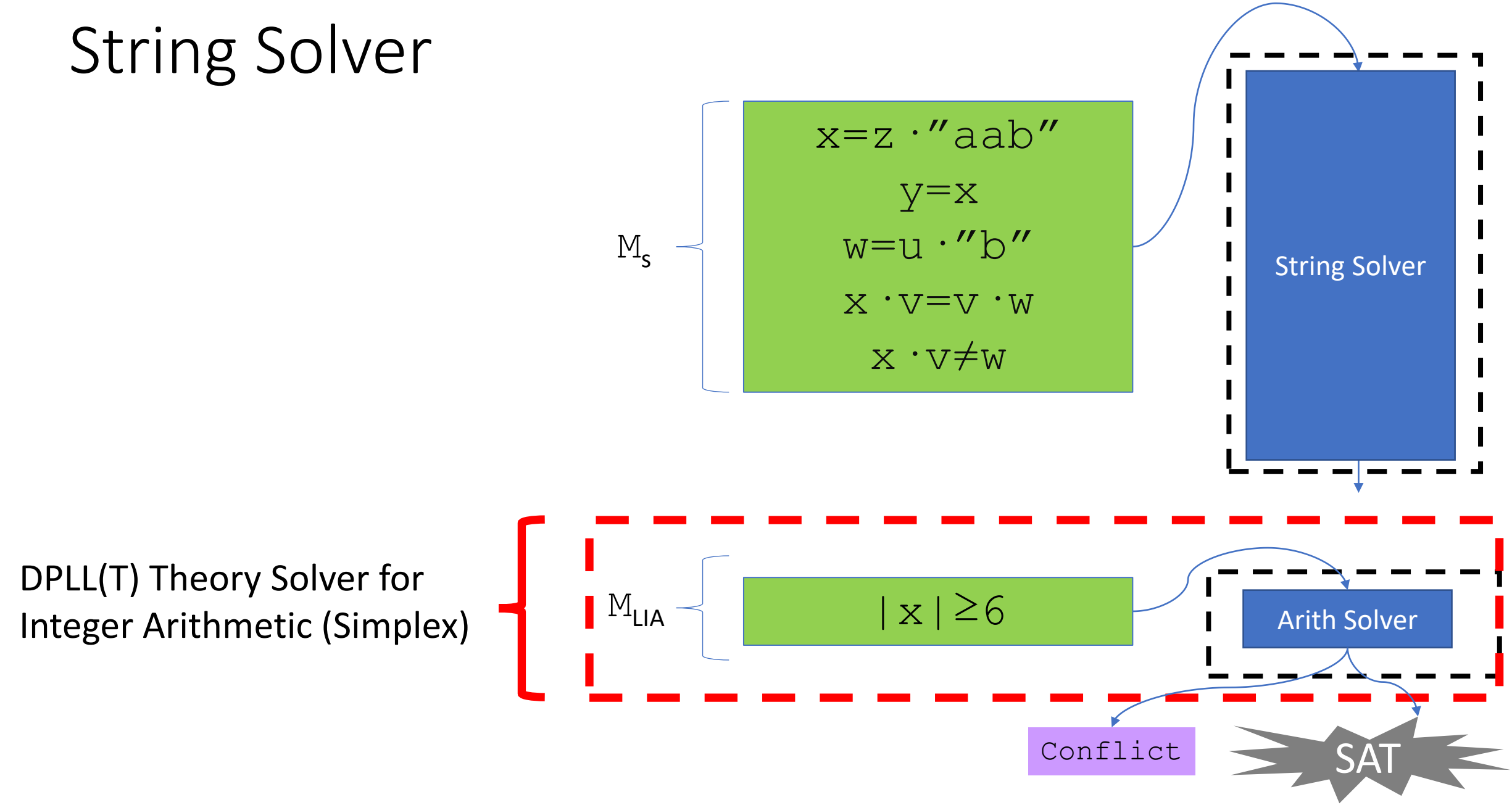
String Solver



String Solver



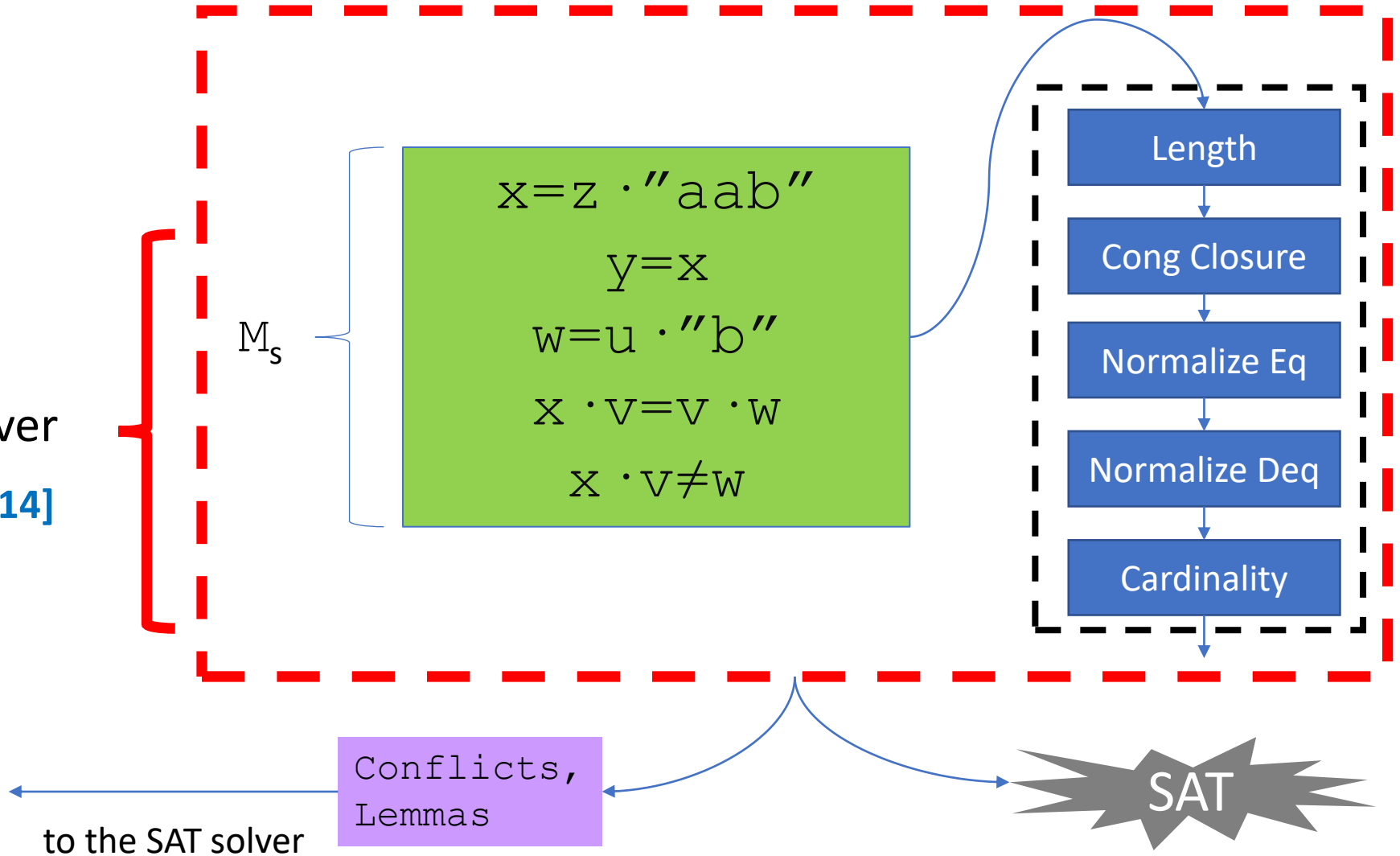
String Solver



String Solver

Will focus on string solver

[Liang et al CAV2014]



String Theory Solver

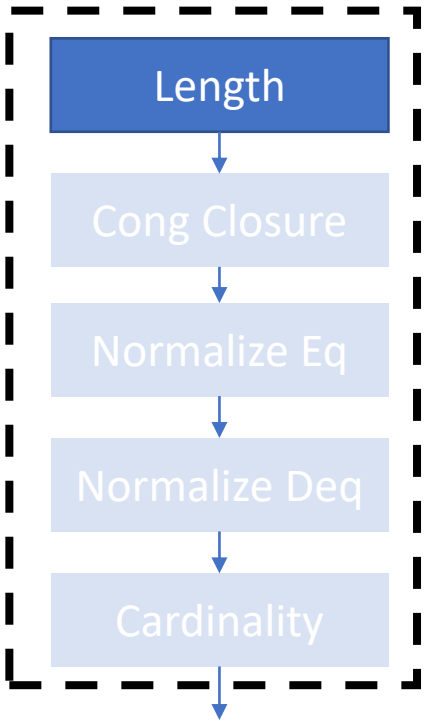
- Inference strategy:
  1. Elaborate length constraints
  2. Check for equality conflicts (congruence closure)
  3. Normalize string equalities
  4. Normalize string disequalities
  5. Check cardinality constraints

- Each step may add lemma or a conflict
- If no step adds a lemma or conflict, we are 

String Solver: Elaborate Length

M_s

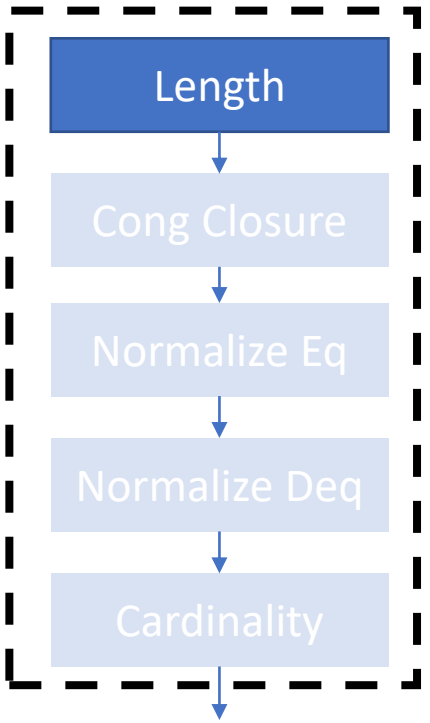
$$x = z \cdot "aab"$$
$$y = x$$
$$w = u \cdot "b"$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



String Solver: Elaborate Length

M_s

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



- For each term of type string in M_s:
  - Add lemma giving the definition of its length:

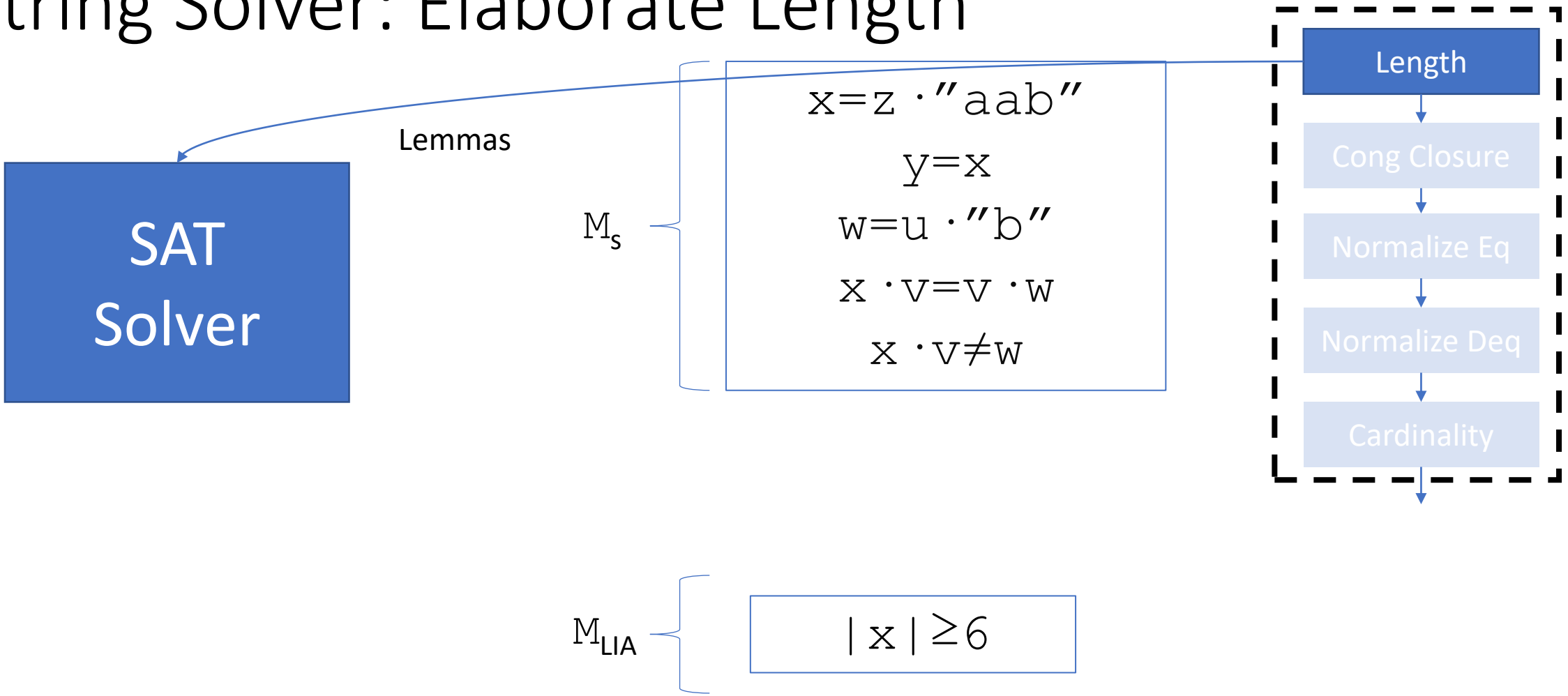
$|"b"| = 1$ $\quad$ $|"aab"| = 3$ $\quad$ $|x \cdot v| = |x| + |v|$

$|z \cdot "aab"| = |z| + 3$ $\quad$ $|u \cdot "b"| = |u| + 3$ $\quad$ $|v \cdot w| = |v| + |w|$

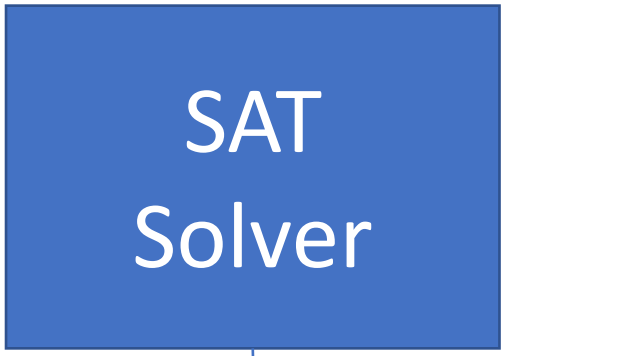
- For each variable of type string in M_s:
  - Add a emptiness splitting lemmas:

$x = "" \ \vee |x| \geq 1$ $\quad$ $y = "" \ \vee |y| \geq 1$ $\quad$ ...

String Solver: Elaborate Length



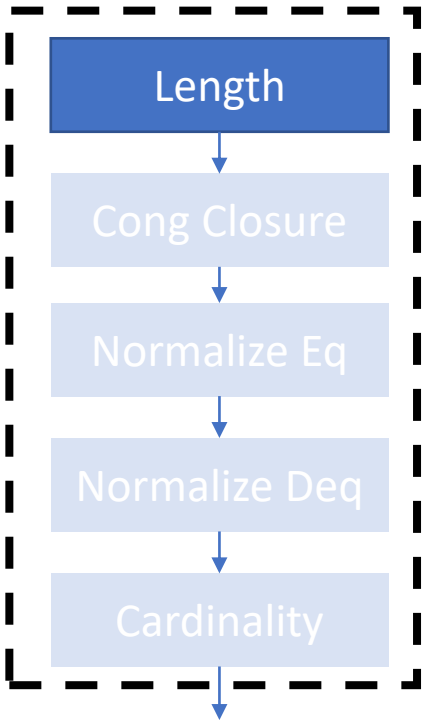
String Solver: Elaborate Length



New assignment

M_s

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



M_{LIA}

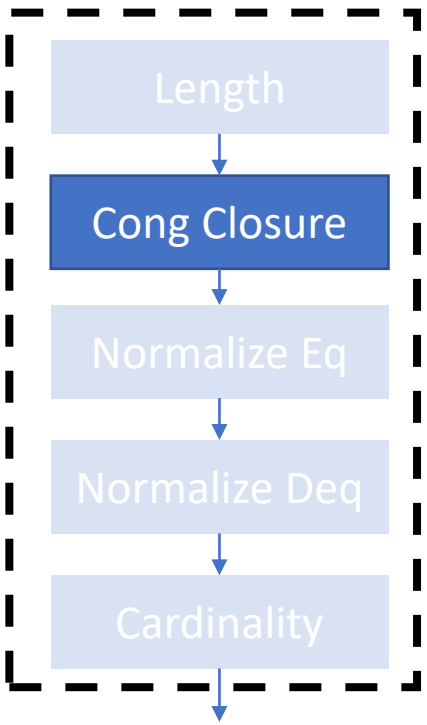
$$|x| \geq 6$$
$$|"b"| = 1$$
$$|"aab"| = 3$$
$$|x \cdot v| = |x| + |v|$$
$$|z \cdot "aab"| = |z| + 3$$
$$|u \cdot "b"| = |u| + 3$$
$$|v \cdot w| = |v| + |w|$$
$$|x| \geq 1$$
$$\ldots$$

- …will trigger new constraints in arithmetic solver

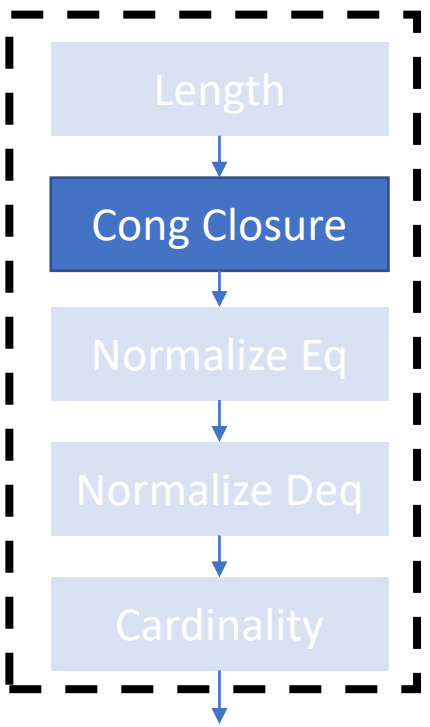
String Solver: Congruence Closure

M_s

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



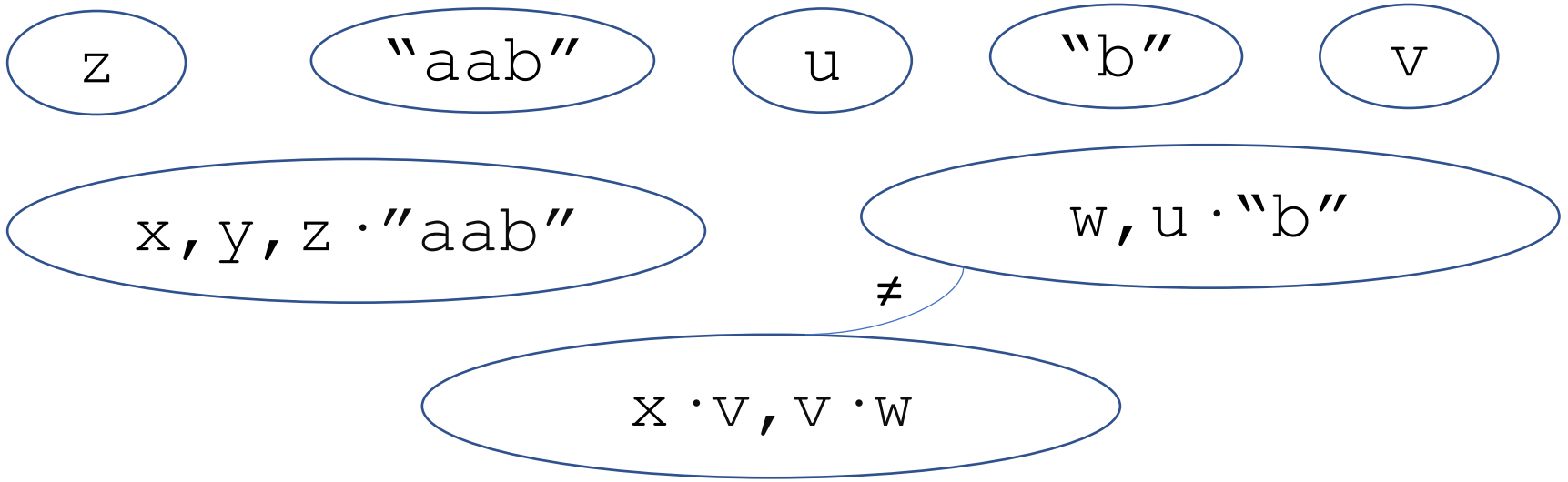
String Solver: Congruence Closure



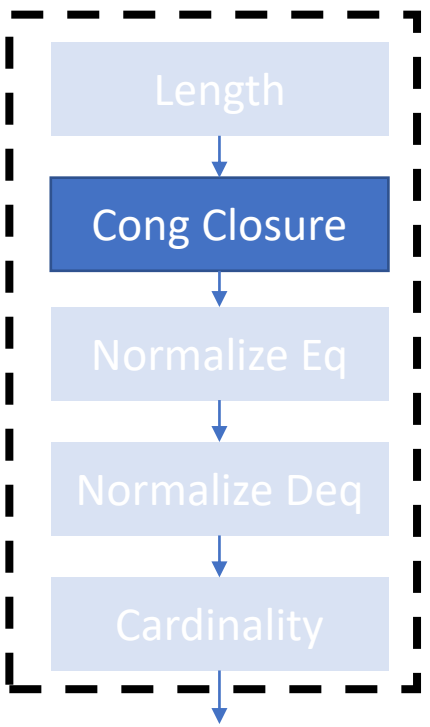
M_s

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

- Group terms by *equivalence classes*:



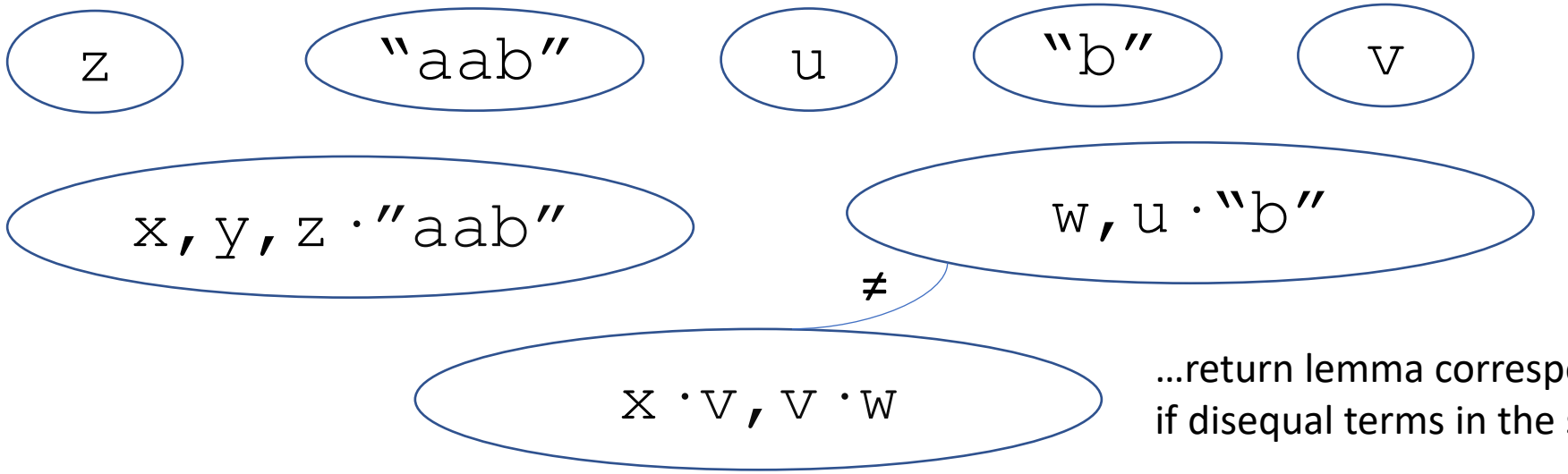
String Solver: Congruence Closure



M_s

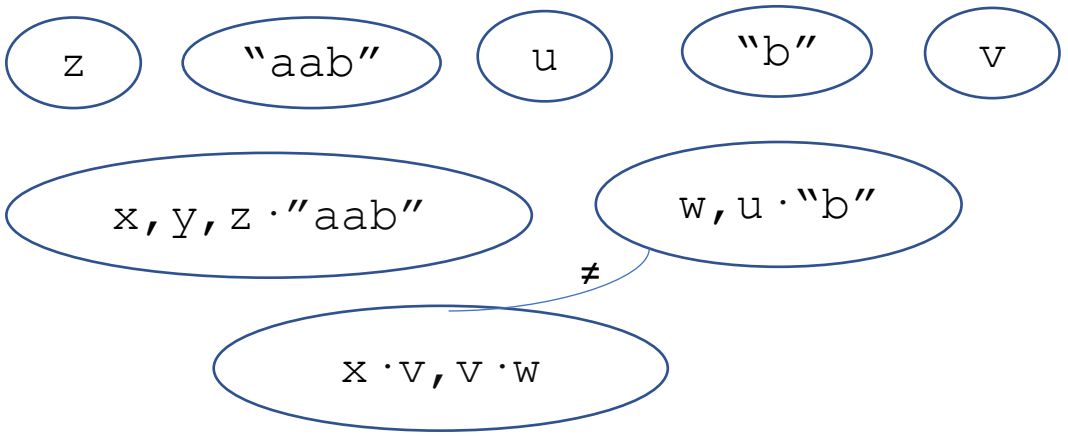
$$x = z \cdot \text{“aab”}$$
$$y = x$$
$$w = u \cdot \text{“b”}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$

- Group terms by *equivalence classes*:

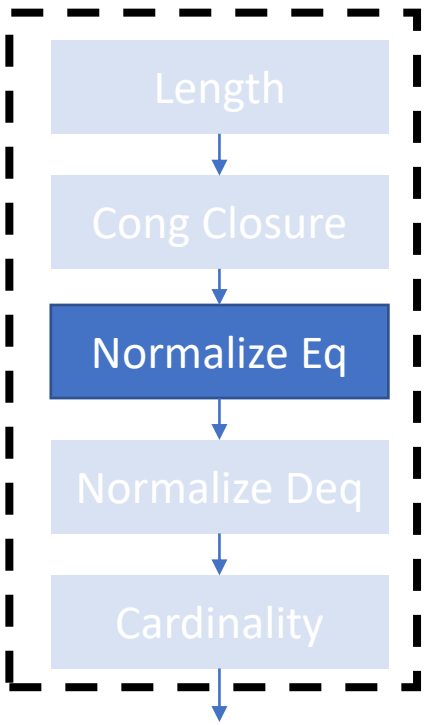


…return lemma corresponding to T_s-conflict if disequal terms in the same equivalence class

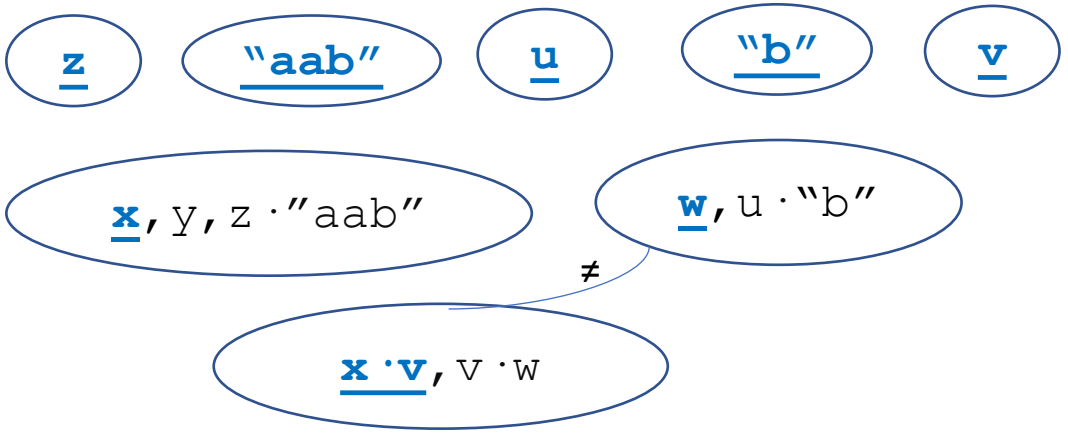
String Solver: Normalize Equality



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



String Solver: Normalize Equality



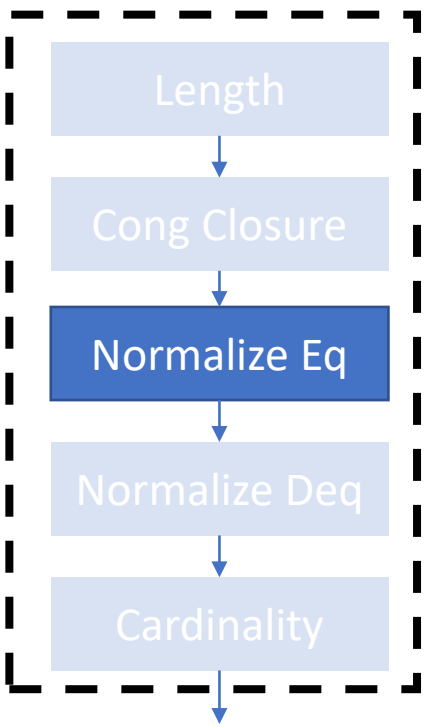
$x = z \cdot \text{"aab"}$

$y = x$

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

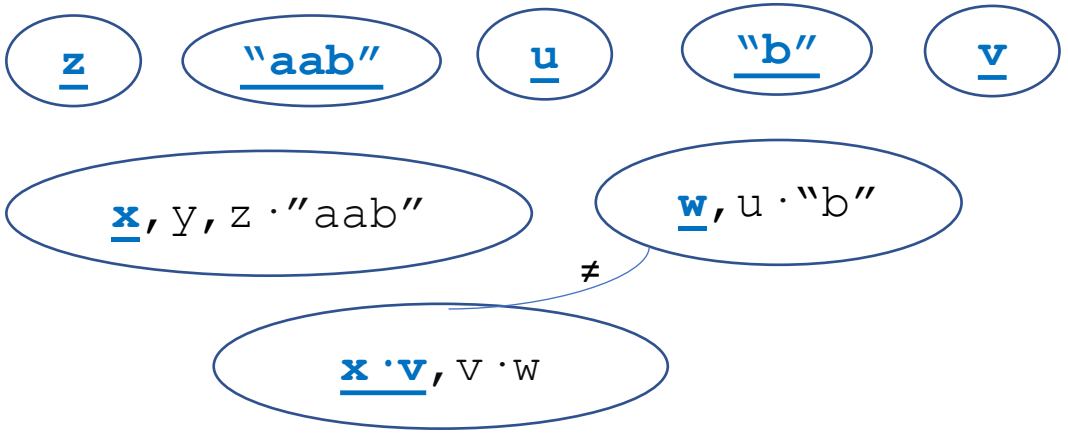
$x \cdot v = v \cdot w$

$x \cdot v \neq w$

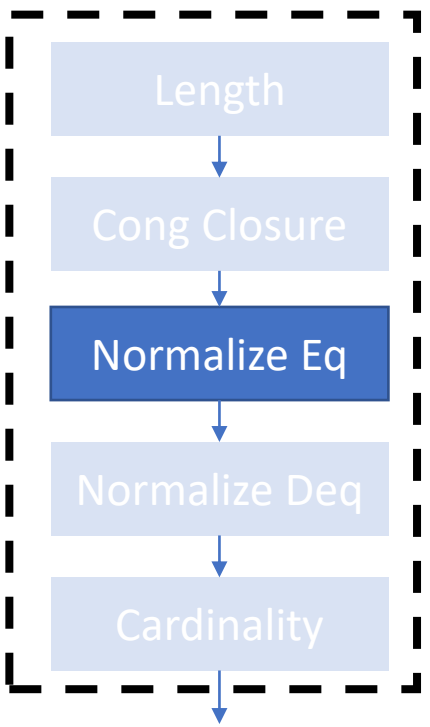


- Compute *normal forms* for equivalence classes
  - A normal form is a concatenation of string terms $r_1 \cdot \ldots \cdot r_n$
    - Where each $r1, \ldots, rn$ is the **representative** of its equivalence class
      - Restriction: string constants must be chosen as representatives
  - An equivalence 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

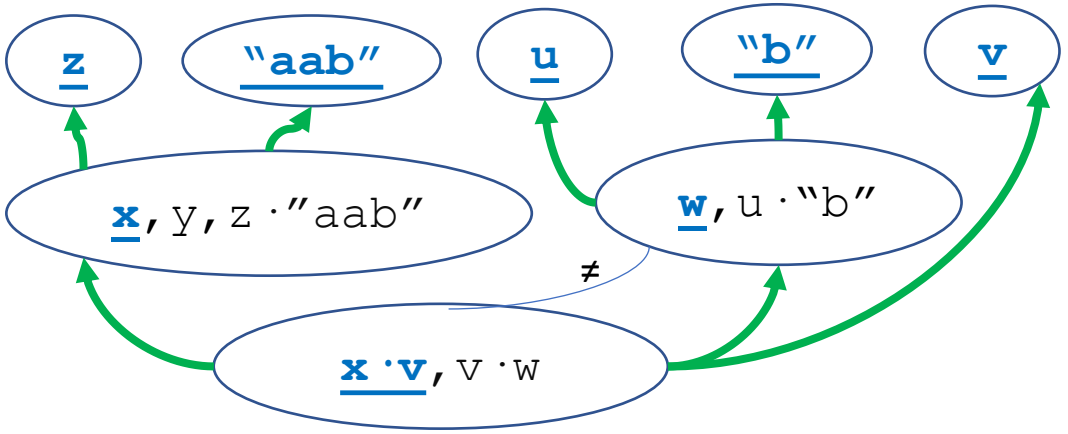


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$

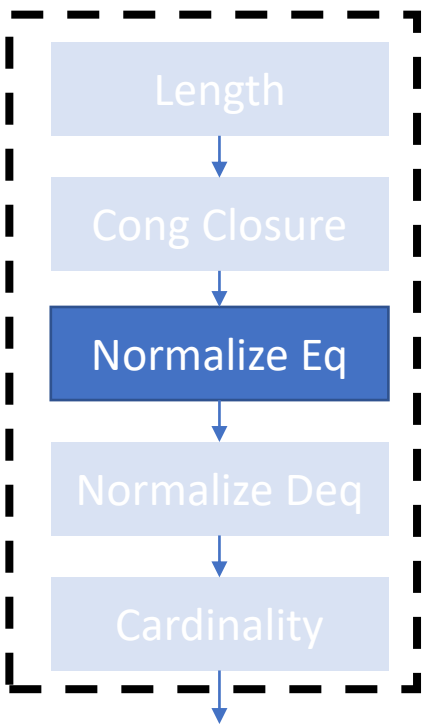


- Normal forms computed by a bottom-up procedure

String Solver: Normalize Equality

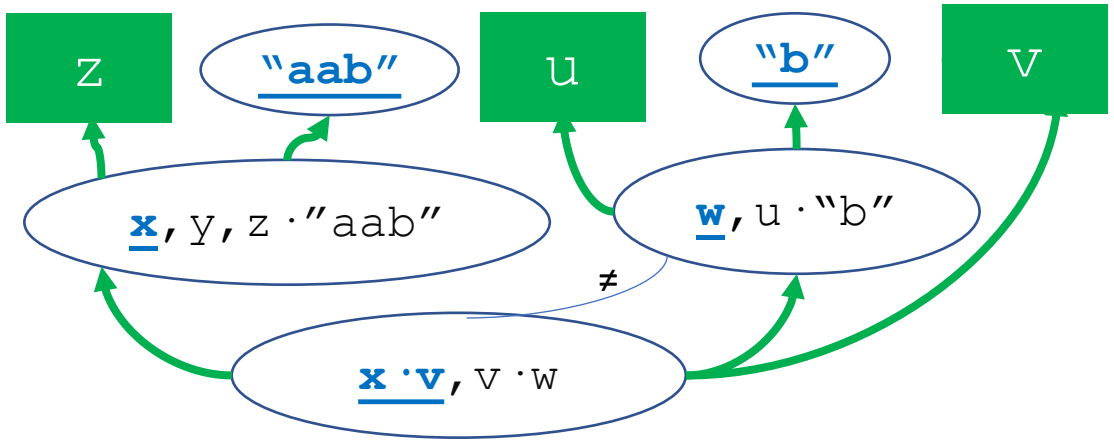


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$

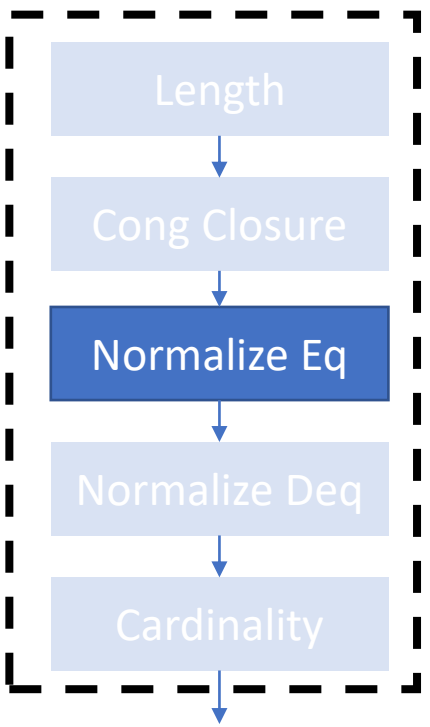


- Normal forms computed by a bottom-up procedure
  - First, compute ***containment relation*** induced by concatenation terms
    - To compute a n.f. for eq-class of $x \cdot v$, we must first compute a n.f. for eq-class of x and v
    - This relation is guaranteed to be acyclic due to length elaboration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

String Solver: Normalize Equality

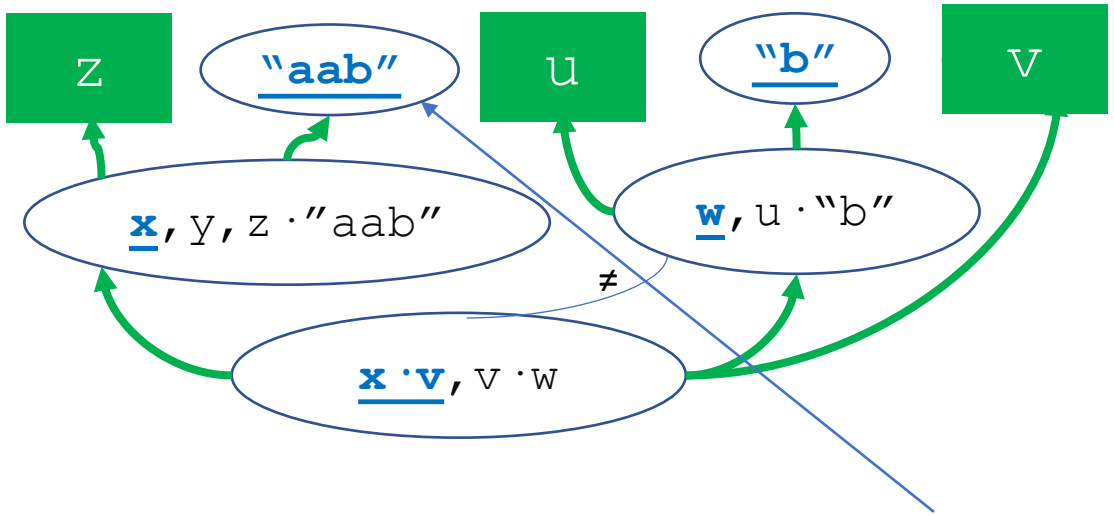


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



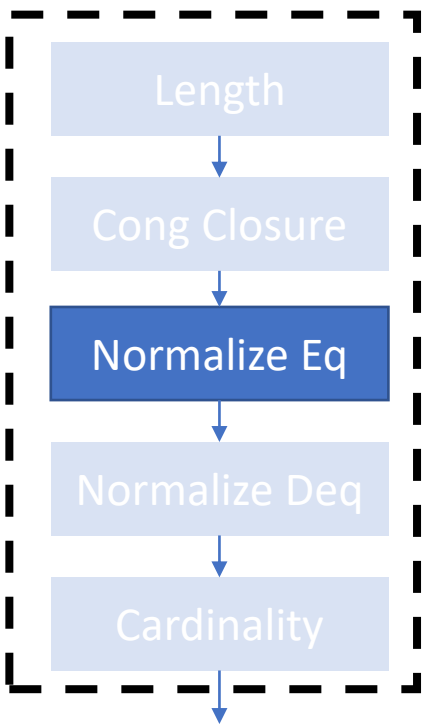
- Normal forms computed by a bottom-up procedure
  - First, compute containment relation induced by concatenation terms
    - To compute a n.f. for eq-class of $x \cdot v$, we must first compute a n.f. for eq-class of x and v
    - This relation is guaranteed to be acyclic due to length elaboration step (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
    - If $t_1 \cong \ldots \cong t_n$, assign. If there exists distinct t_i, t_j, then propagate or split

String Solver: Normalize Equality

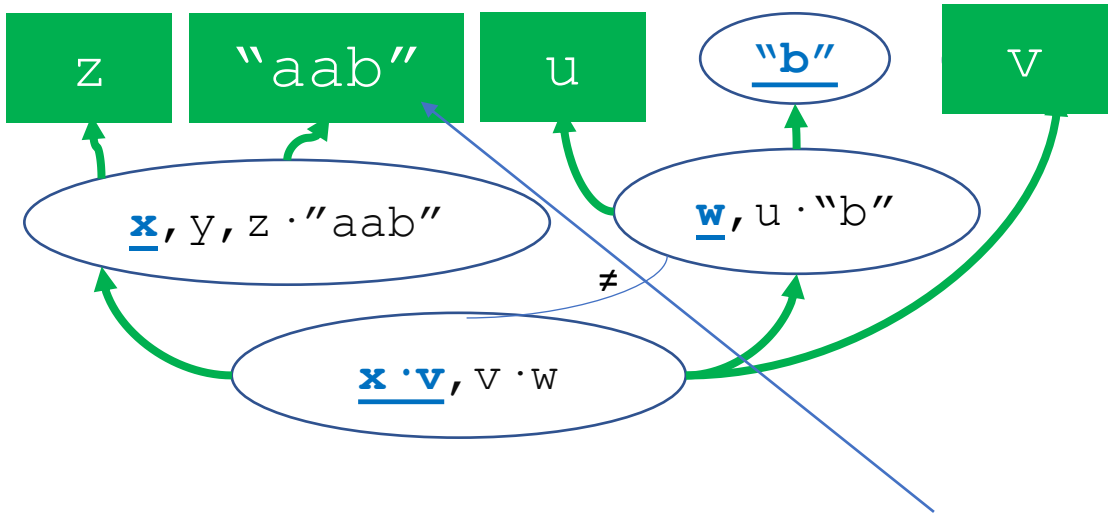


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$

Single non-variable string term ⇒ assign

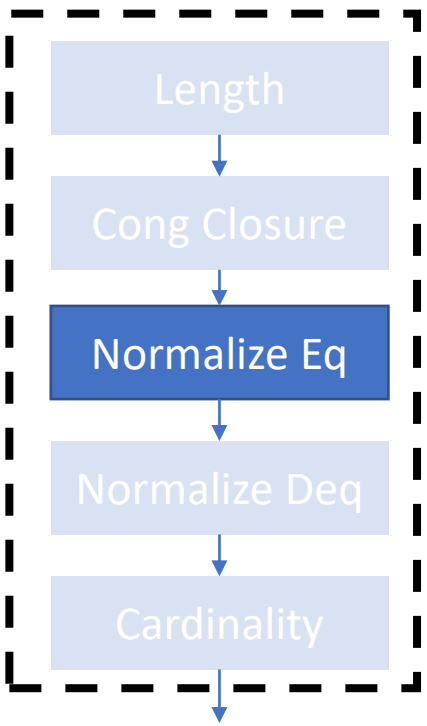


String Solver: Normalize Equality

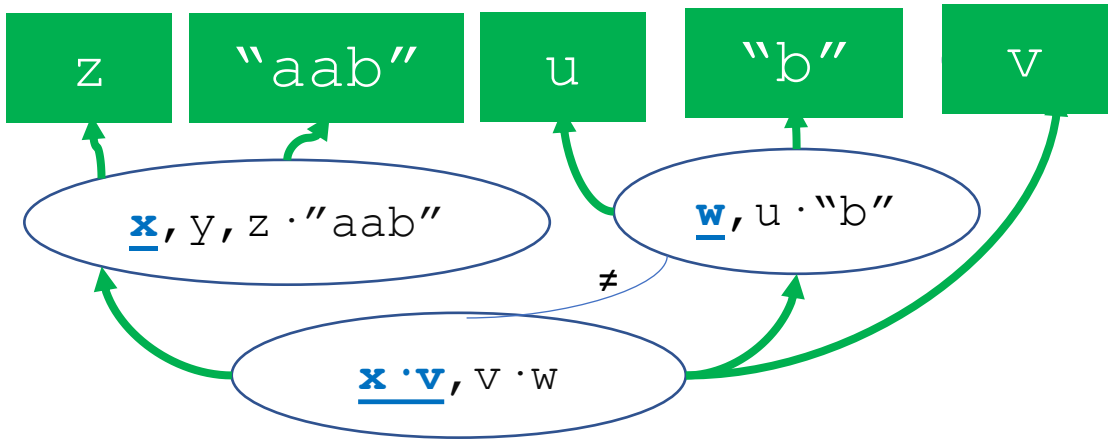


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$

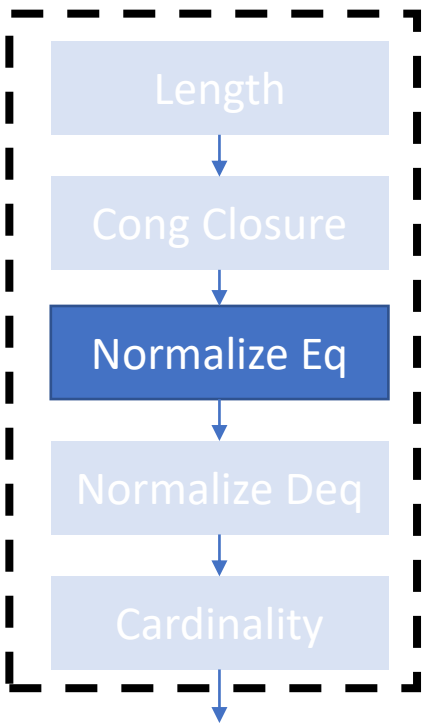
Single non-variable string term ⇒ assign



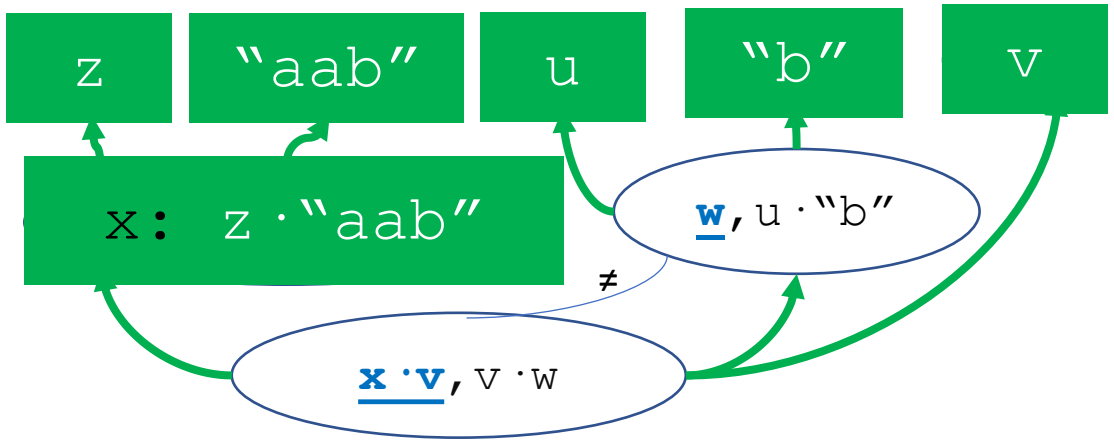
String Solver: Normalize Equality



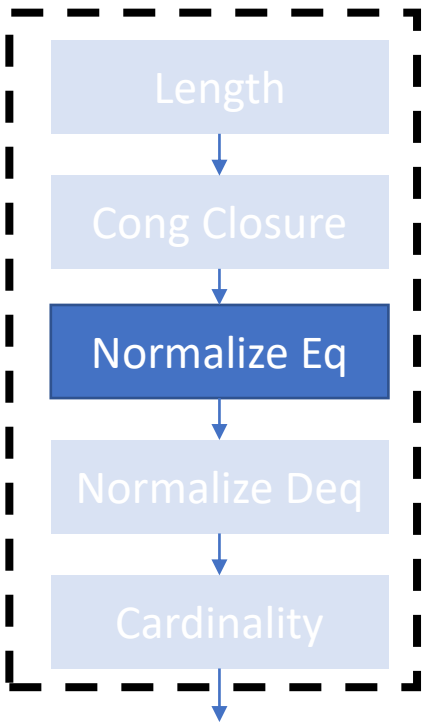
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



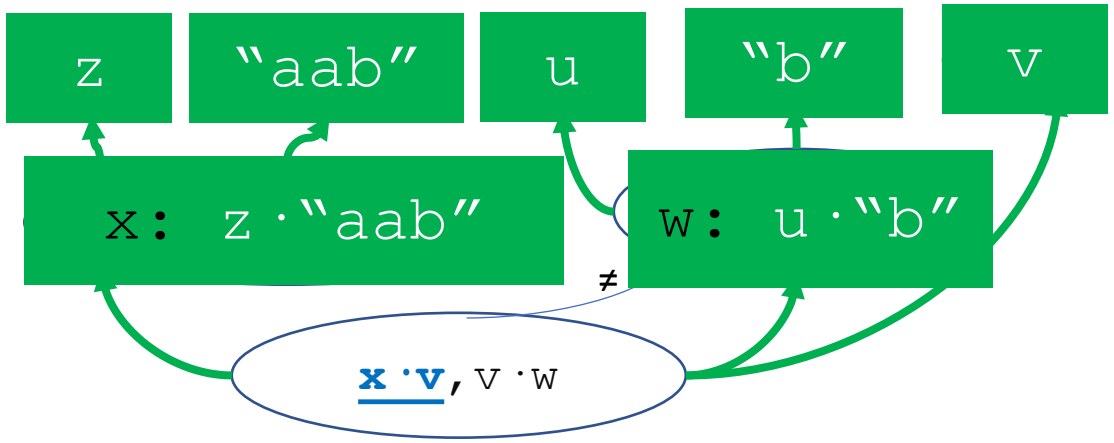
String Solver: Normalize Equality



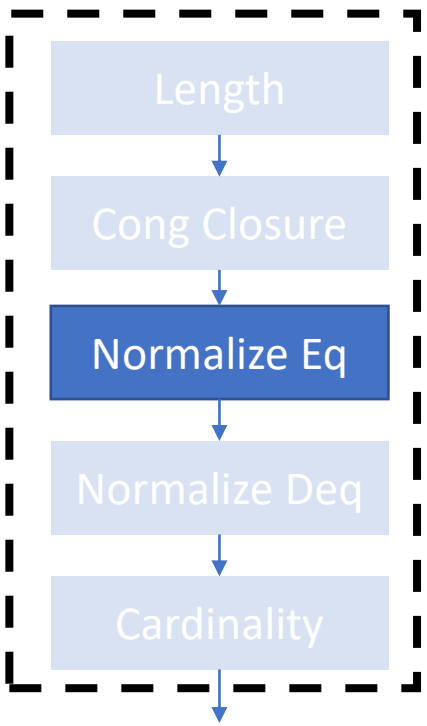
$$x = z \cdot "aab"$$
$$y = x$$
$$w = u \cdot "b"$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



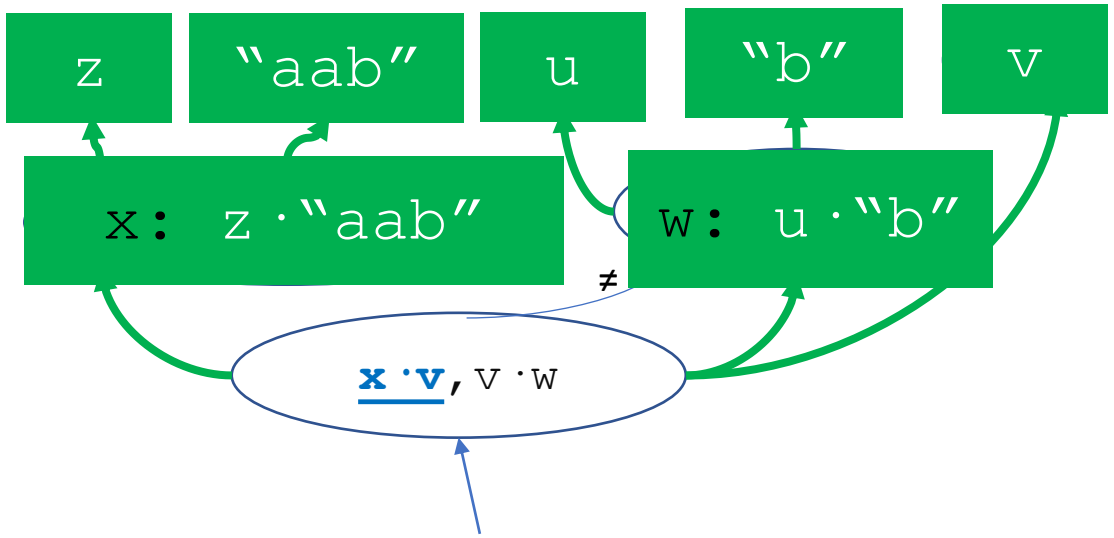
String Solver: Normalize Equality



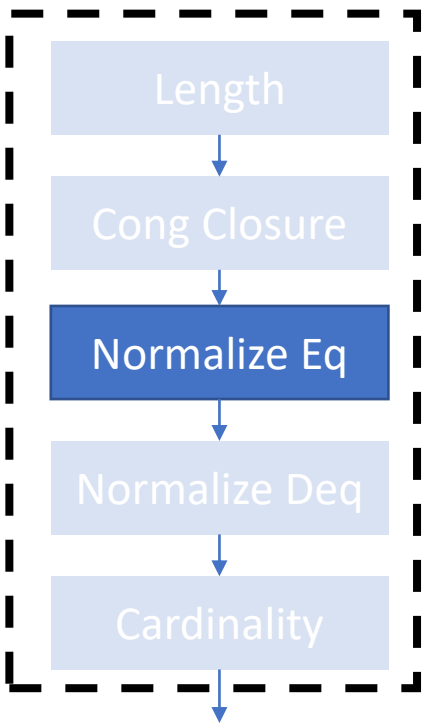
$$x = z \cdot "aab"$$
$$y = x$$
$$w = u \cdot "b"$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



String Solver: Normalize Equality

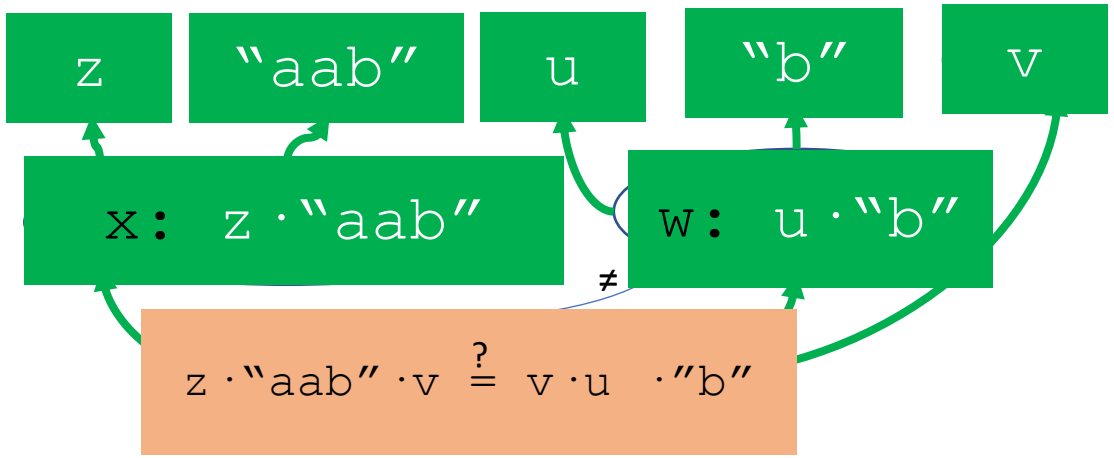


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$

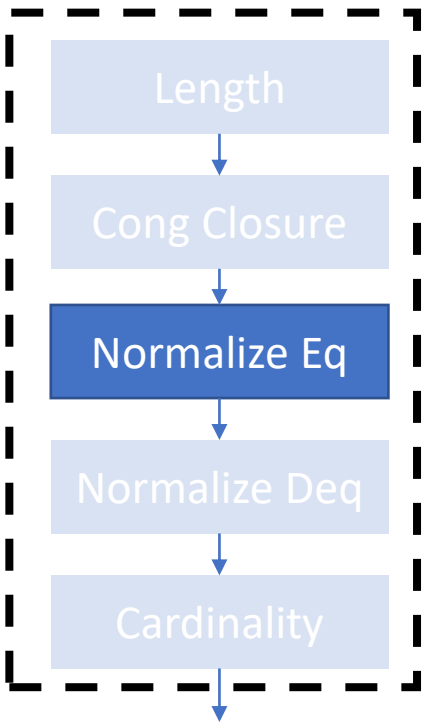


- Equivalence class with two non-variable terms with distinct expanded forms:
  - $x \cdot v = (z \cdot \text{"aab"}) \cdot v = \mathbf{z \cdot "aab" \cdot v}$
  - $v \cdot w = v \cdot (u \cdot \text{"b"}) = \mathbf{v \cdot u \cdot "b"}$

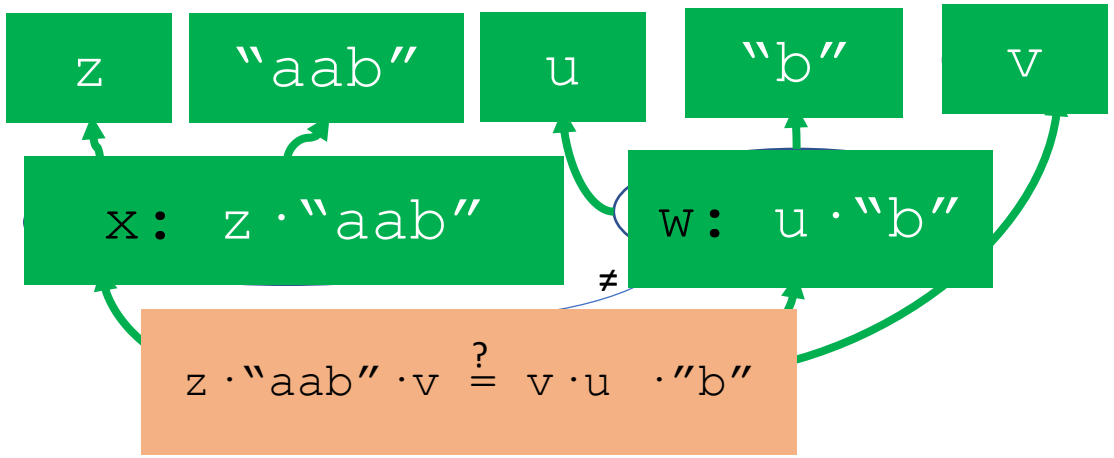
String Solver: Normalize Equality



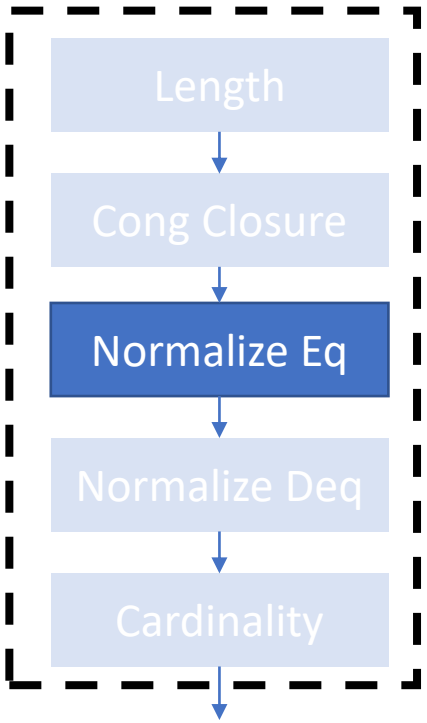
$$x = z \cdot "aab"$$
$$y = x$$
$$w = u \cdot "b"$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



String Solver: Normalize Equality

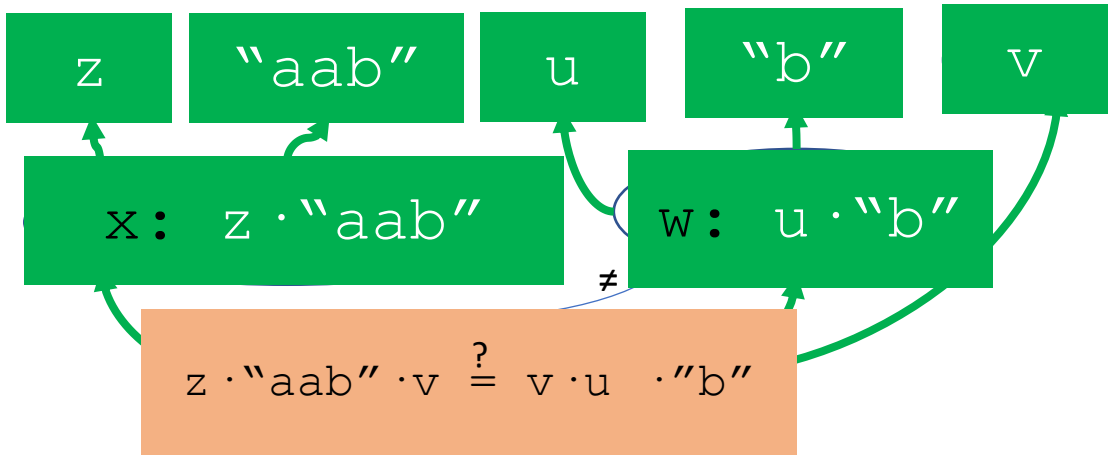


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$

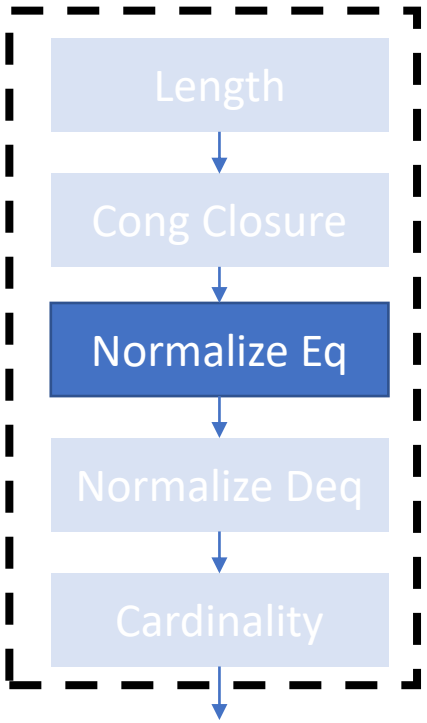


Goal: split strings such that *all* aligning components are equal

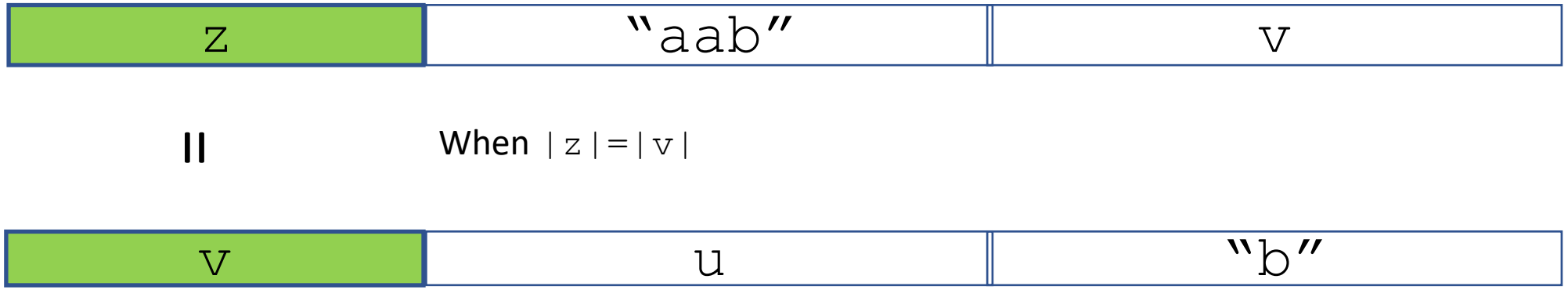
String Solver: Normalize Equality



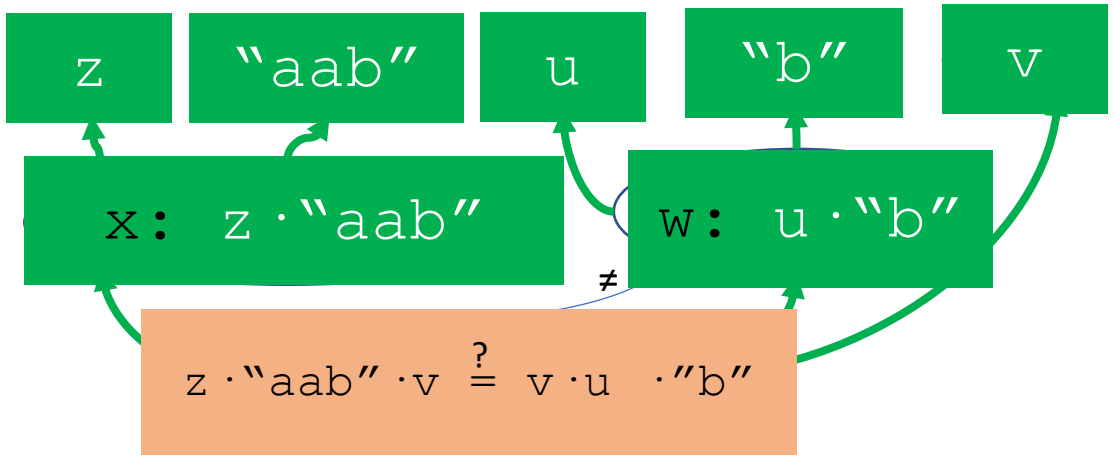
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



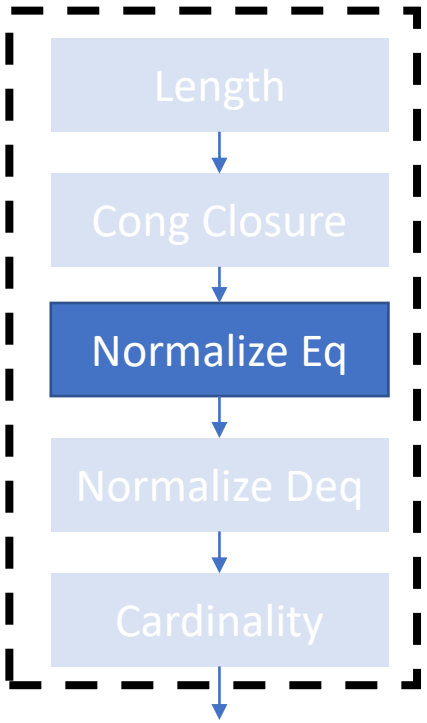
- Consider three cases for making these two terms equal:



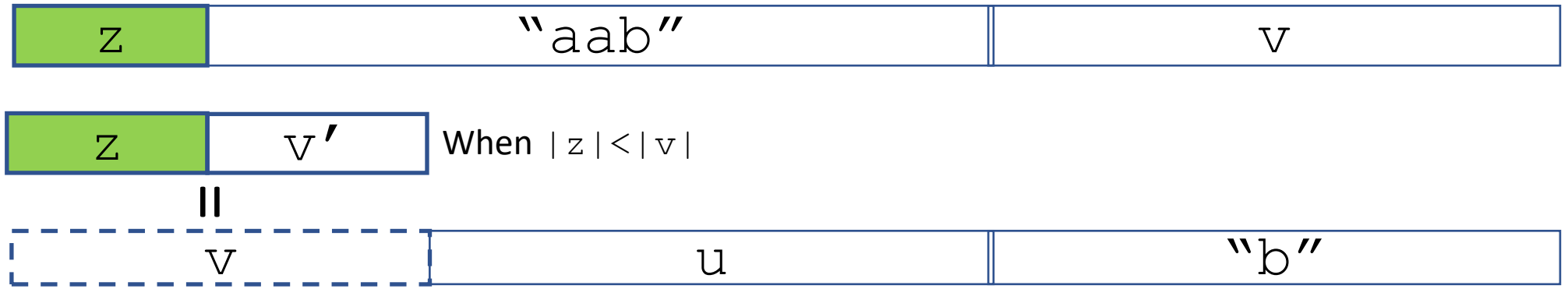
String Solver: Normalize Equality



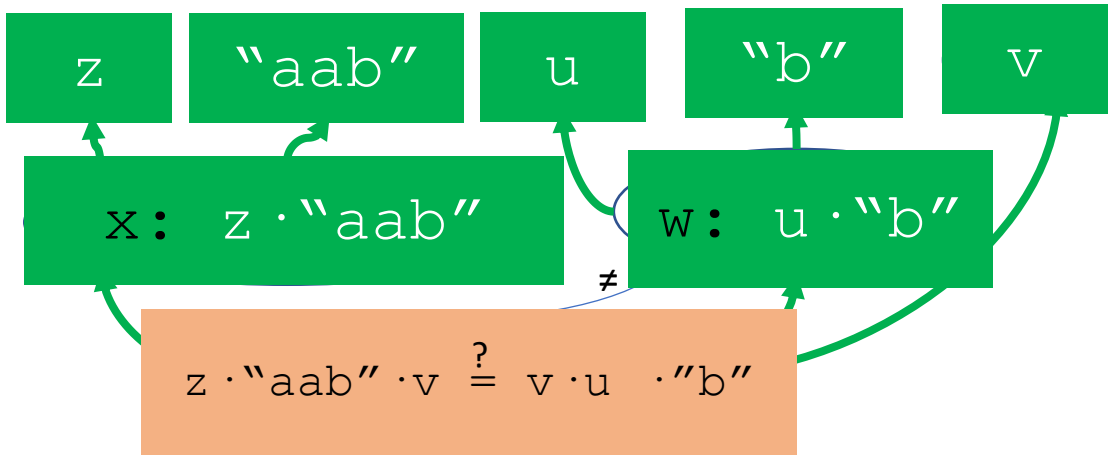
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$



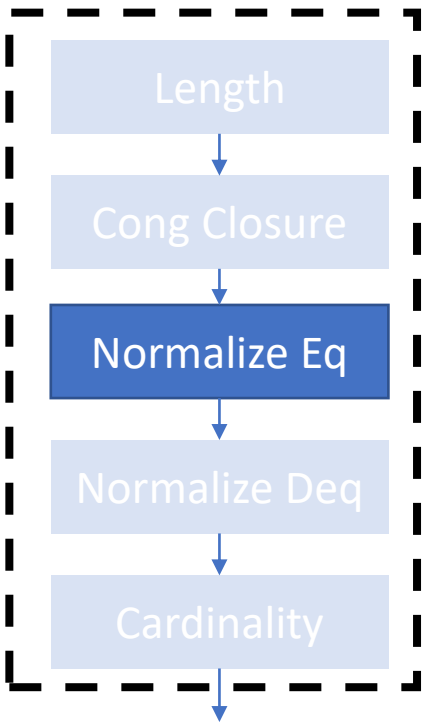
- Consider three cases for making these two terms equal:



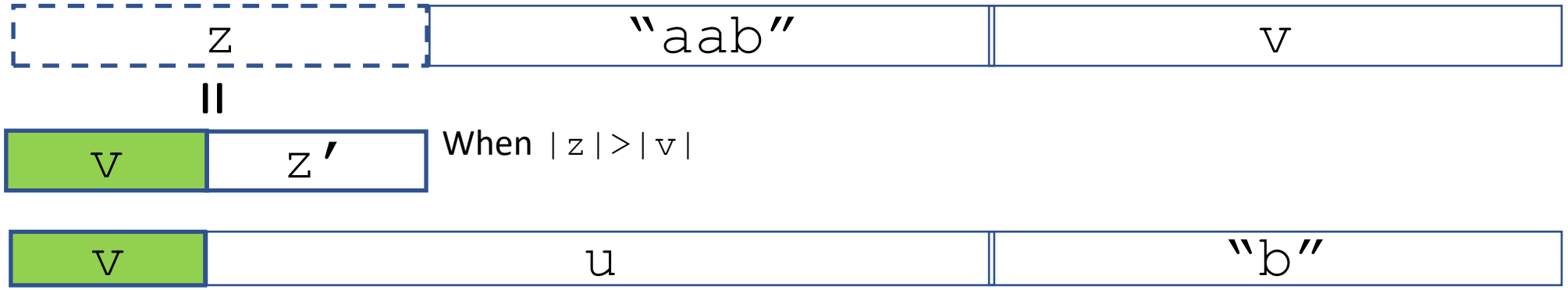
String Solver: Normalize Equality



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



- Consider three cases for making these two terms equal:



String Solver: Normalize Equality

$x = z \cdot \text{“aab”}$

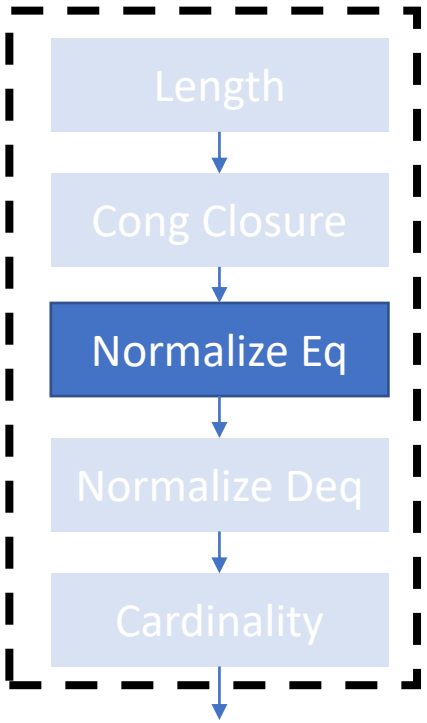
$y = x$

$w = u \cdot \text{“b”}$

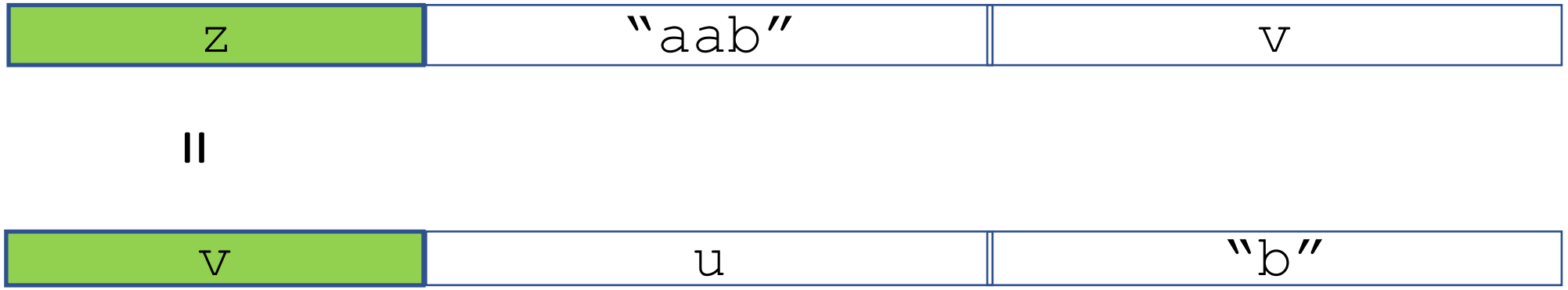
$x \cdot v = v \cdot w$

$x \cdot v \neq w$

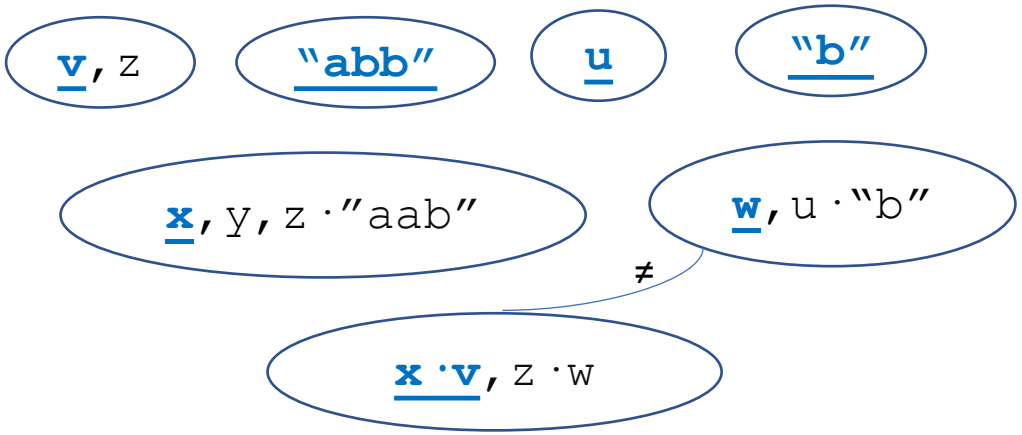
$z = v$



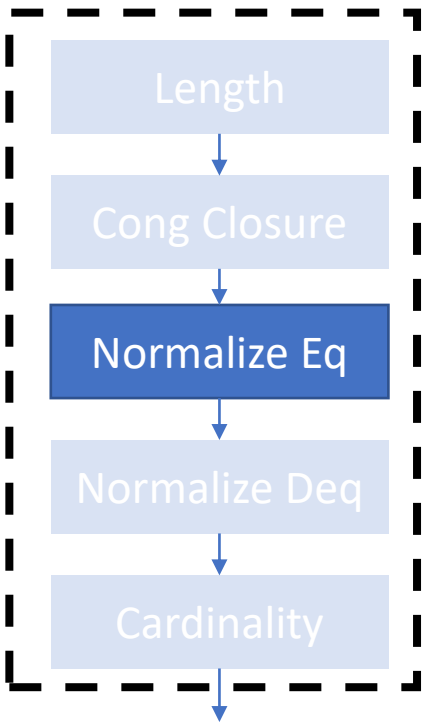
- Consider:



String Solver: Normalize Equality

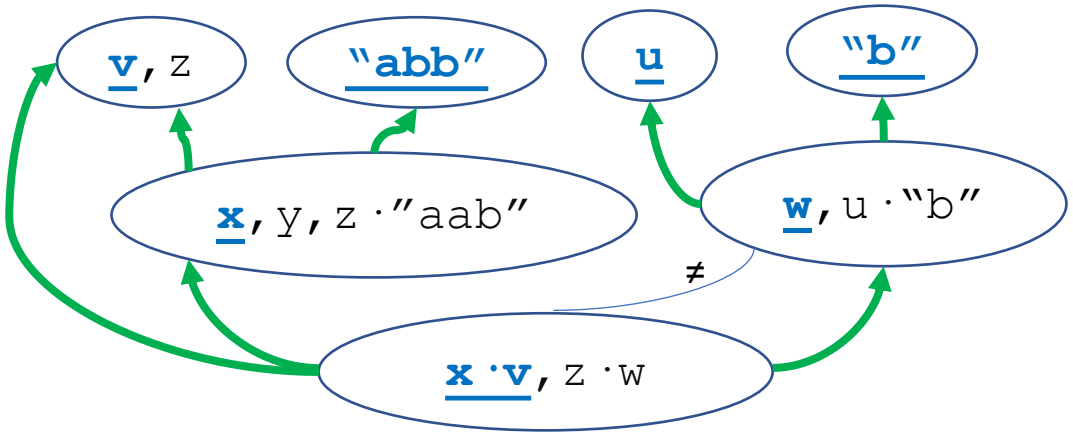


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$
$$z = v$$

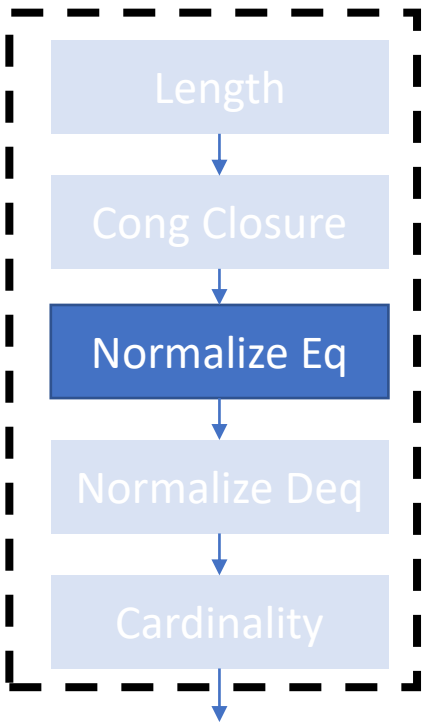


…Recompute *congruence closure*

String Solver: Normalize Equality

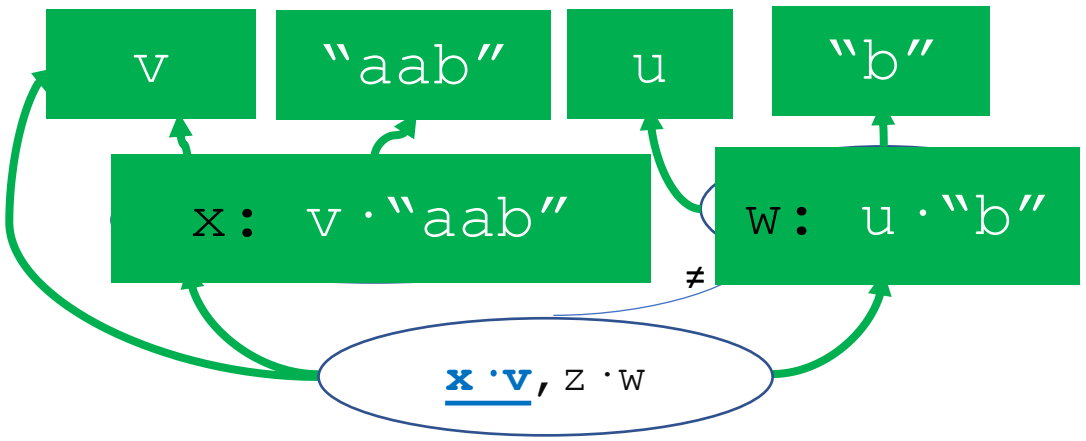


$$x = z \cdot \text{“aab”}$$
$$y = x$$
$$w = u \cdot \text{“b”}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$
$$z = v$$

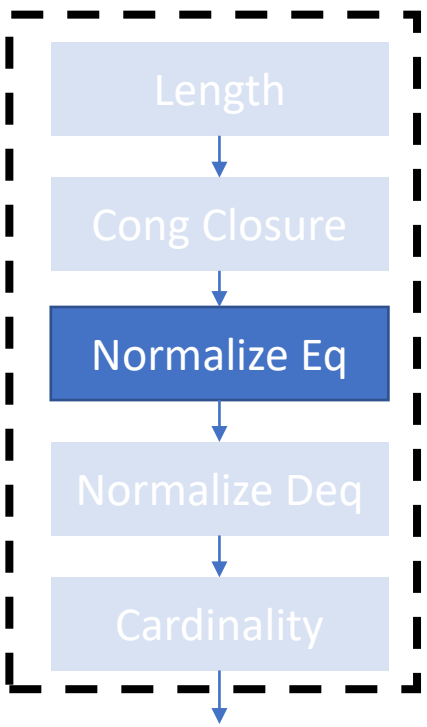


…Recompute congruence closure and *normal forms*

String Solver: Normalize Equality

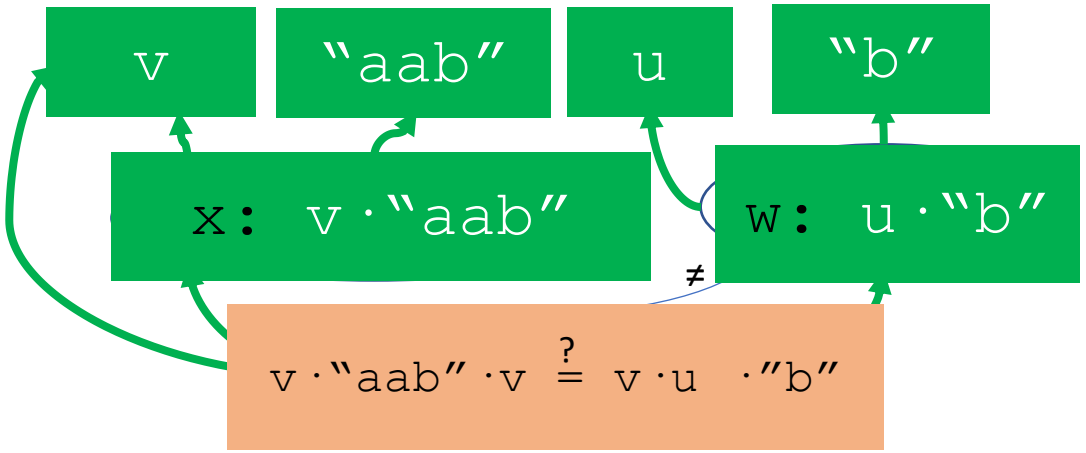


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$
$$z = v$$

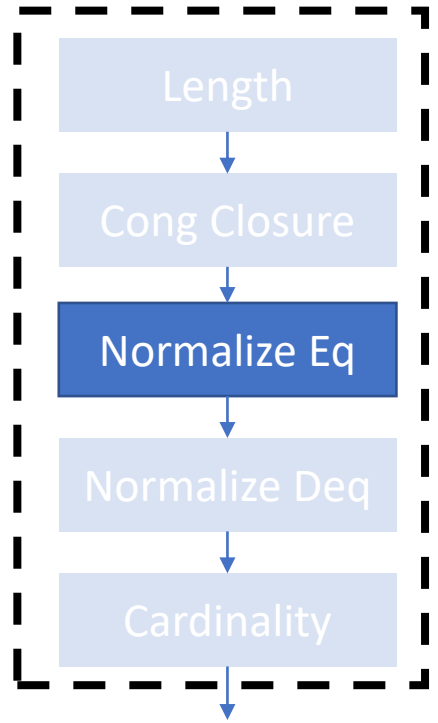


…Recompute congruence closure and *normal forms*

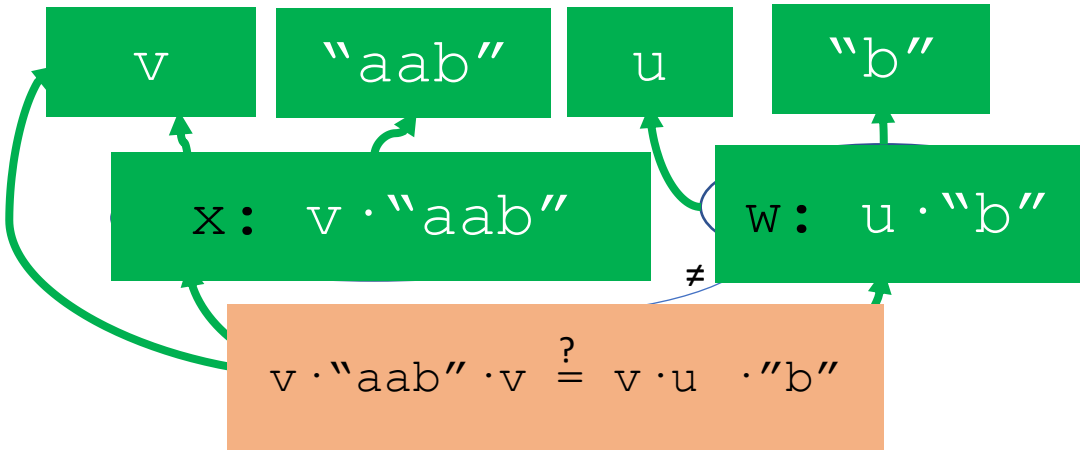
String Solver: Normalize Equality



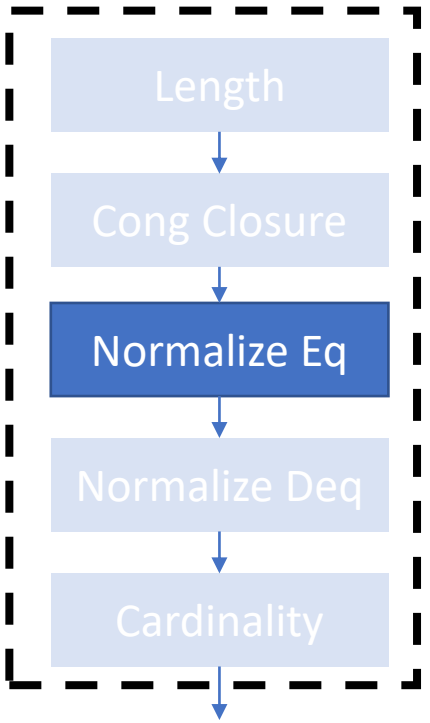
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v = v \cdot w$$
$$x \cdot v \neq w$$
$$z = v$$



String Solver: Normalize Equality



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



…repeat the process on these components

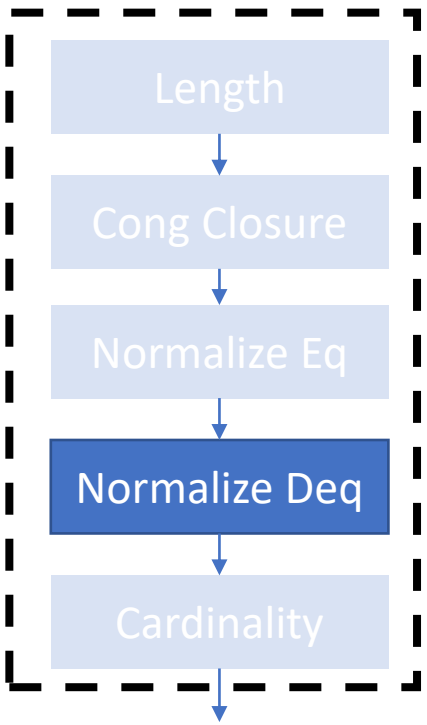
Splitting on String Equalities

- Choosing order of equalities is highly *sophisticated* 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 arithmetic solver, then $x \cdot w = y \cdot v \wedge |x| > |z| \Rightarrow x = y \cdot x'$
  - 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, instead reduce to RE membership
    - E.g. $x \cdot \text{“ba”} = \text{“ab”} \cdot x \Rightarrow x \in (\text{“ab”})^* \cdot \text{“a”}$

String Solver: Normalize Disequalities

modified example

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



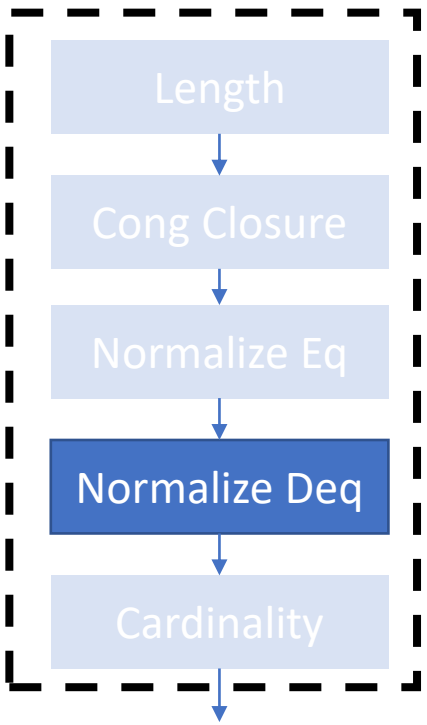
String Solver: Normalize Disequalities

$x = z \cdot "aab"$

$y = x$

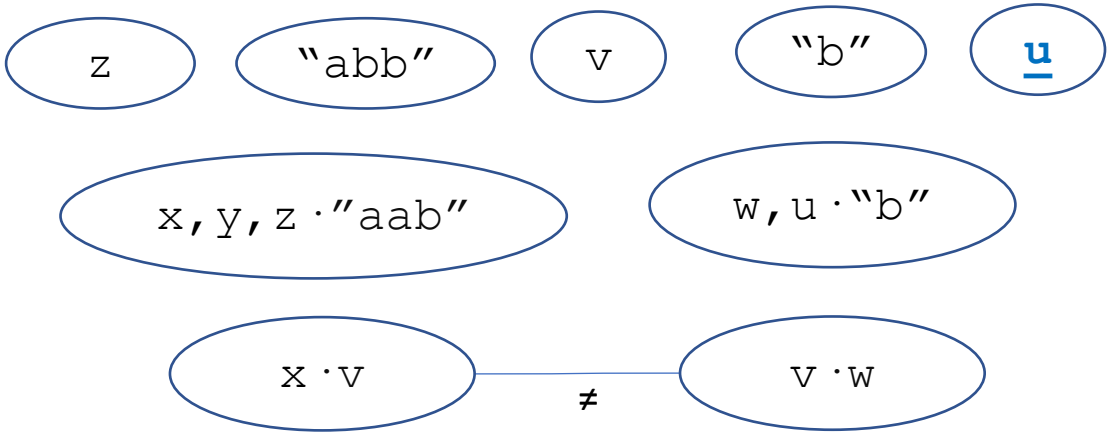
$w = u \cdot "b"$

$\mathbf{x \cdot v \neq v \cdot w}$

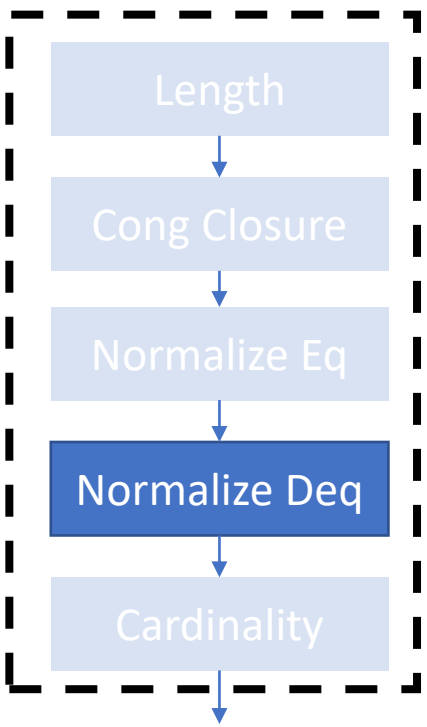


- Disequalities are handled in an analogous manner to equalities
  - If $|x \cdot v| \neq |v \cdot w|$, then trivially $x \cdot v \neq v \cdot w$
  - Otherwise, consider the normal forms of $x \cdot v$ and $v \cdot w$ from previous step

String Solver: Normalize Disequalities

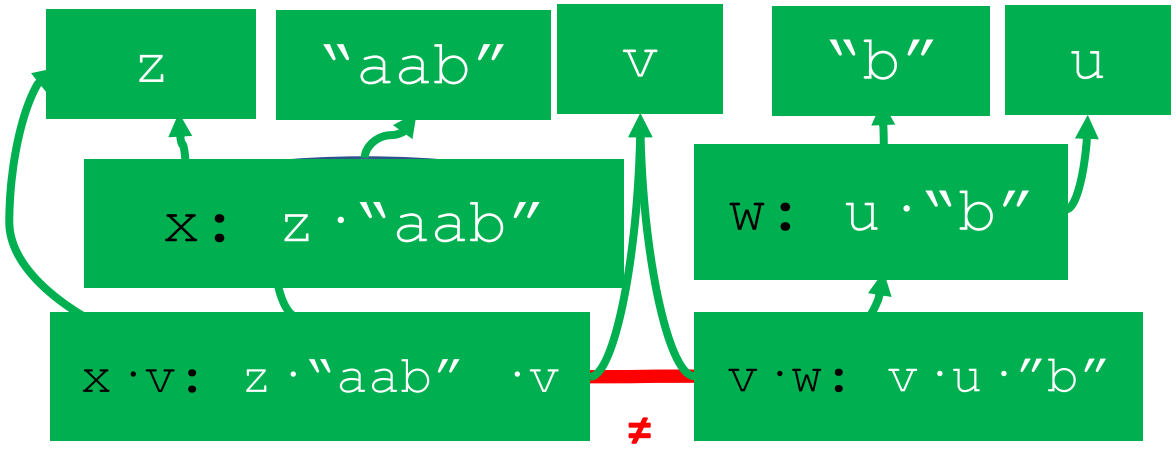


$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v \neq v \cdot w$$

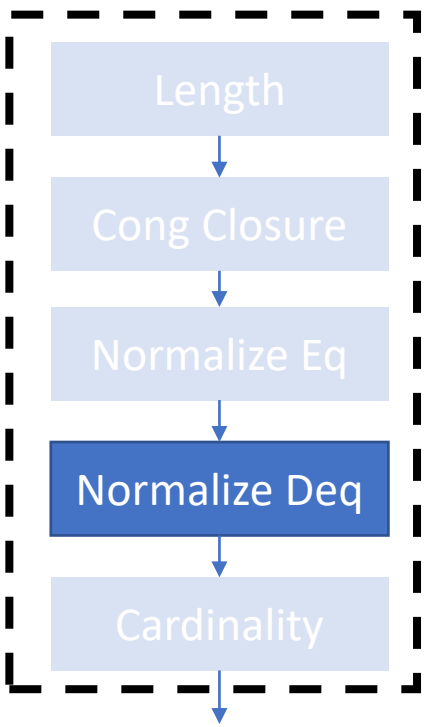


- Disequalities are handled in an analogous manner to equalities

String Solver: Normalize Disequalities



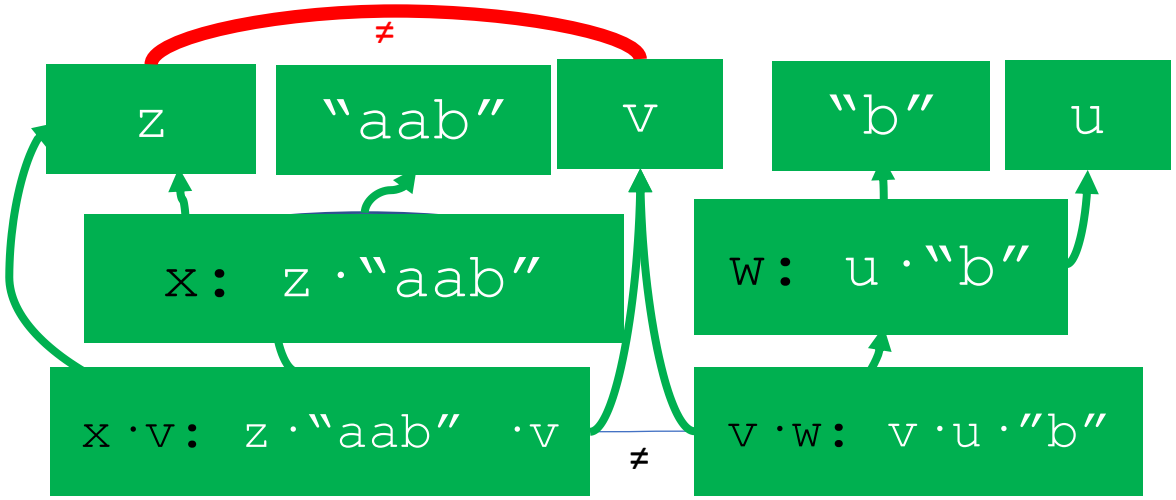
$$x = z \cdot \text{"aab"}$$
$$y = x$$
$$w = u \cdot \text{"b"}$$
$$x \cdot v \neq v \cdot w$$



- Disequalities are handled in an analogous manner to equalities

Goal: find *any* aligning component that is disequal

String Solver: Normalize Disequalities



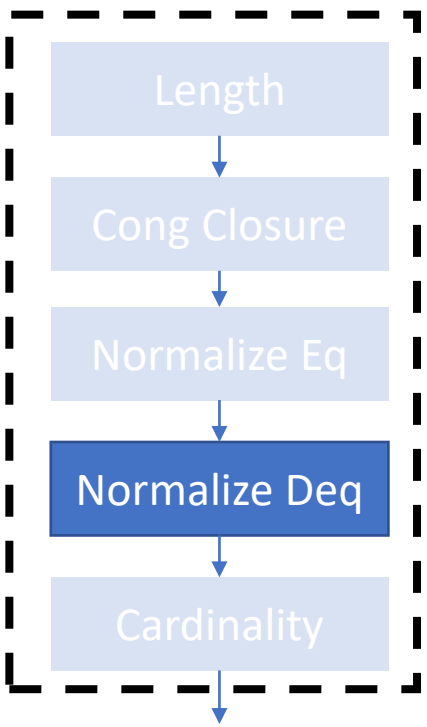
$x = z \cdot \text{"aab"}$

$y = x$

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

$x \cdot v \neq v \cdot w$

$v \neq z$

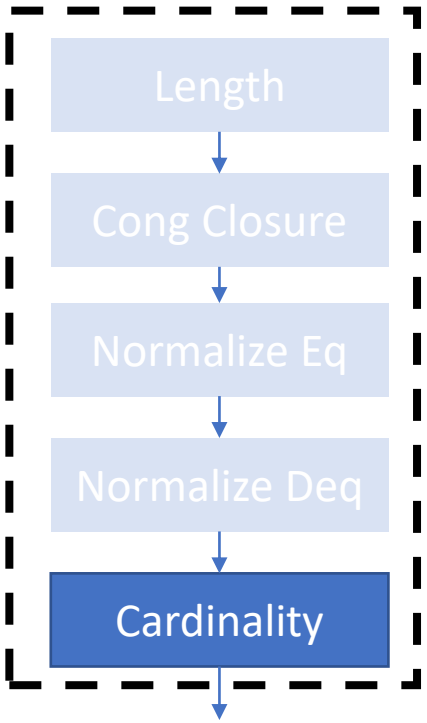


- Disequalities are handled in an analogous manner to equalities

$\not\Rightarrow$ $|z| = |v|$ and $z \neq v$

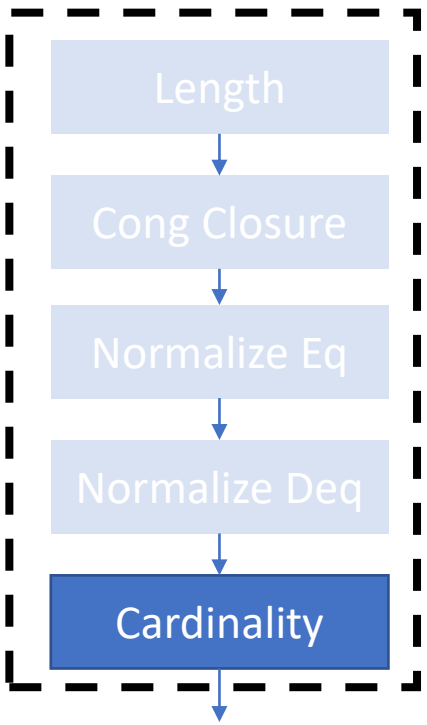
String Solver: Cardinality

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



String Solver: Cardinality

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



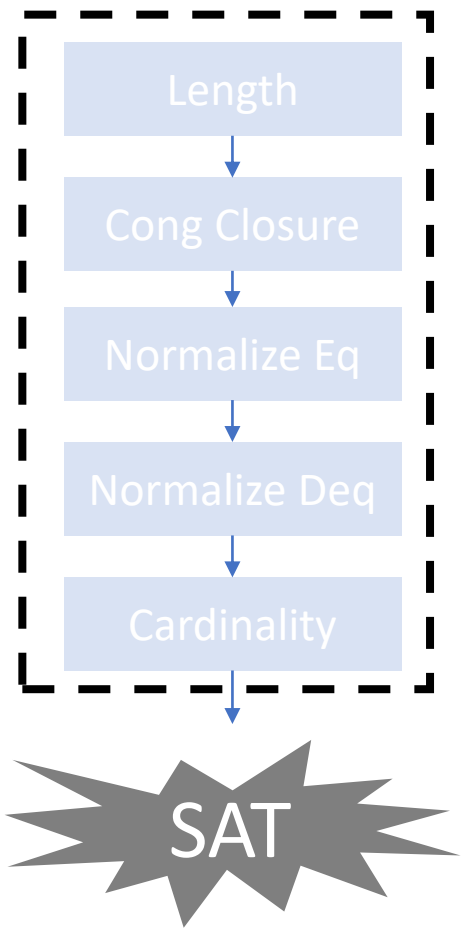
- May be unsatisfiable since Σ is finite
- For instance, if:
  - Σ is a finite alphabet of 256 characters, and
  - M_S entails that 257 distinct strings of length 1 exist

  $\Rightarrow$ Then M_S is unsatisfiable

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

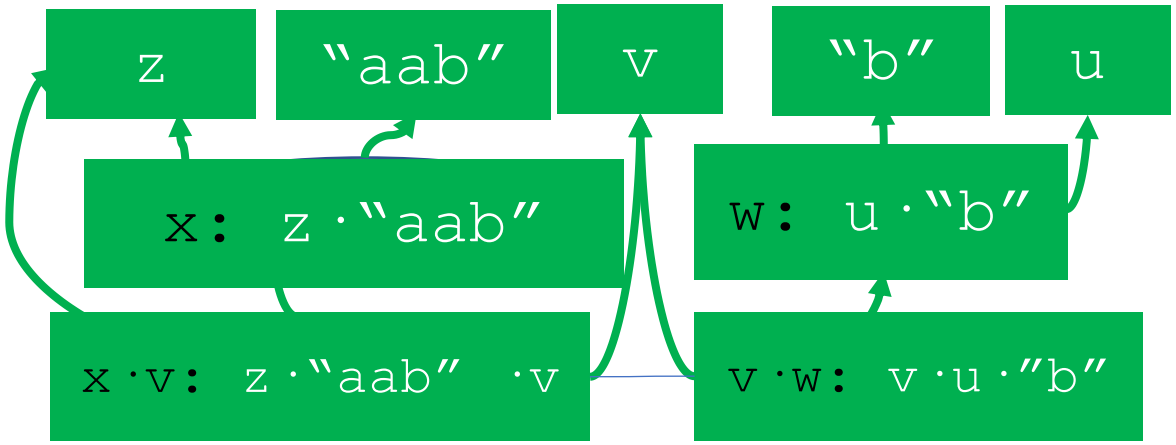
String Solver: Return SAT

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

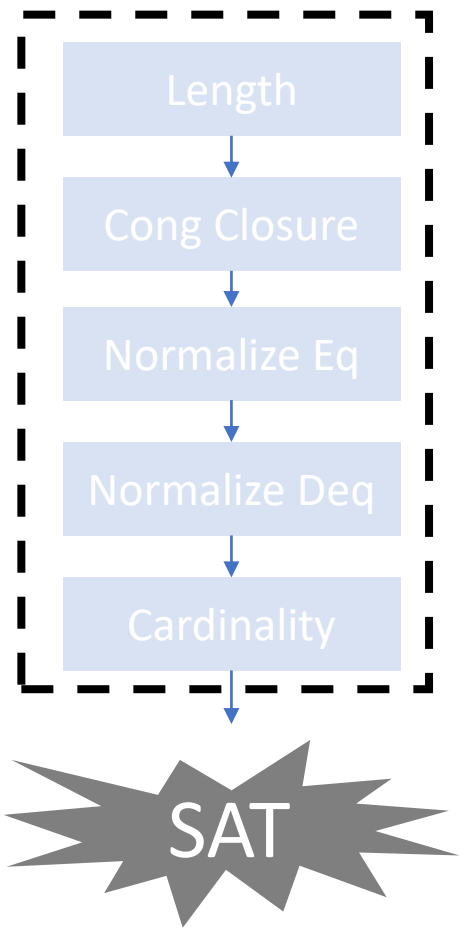


- If all steps finish with no new lemmas:
  - M_s is T_s-satisfiable
  - Model can be computed based on normal forms
    - String constants assigned to eq classes whose normal form is a variable
      - Length fixed by model from arithmetic solver
    - Each variable interpreted as the valuation of the normal form of their eq class

String Solver: Return SAT

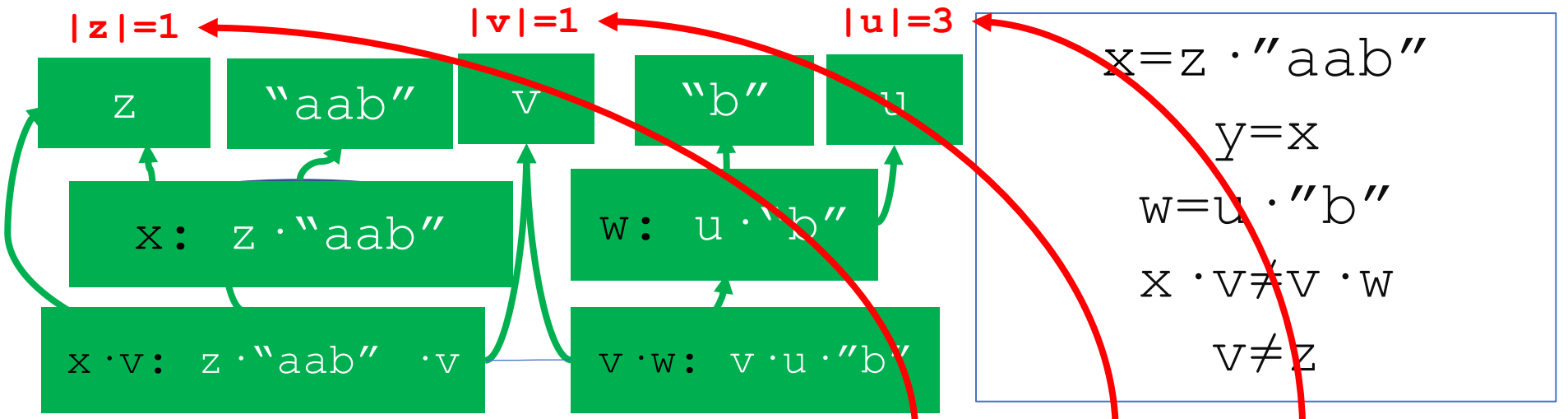


$$x = z \cdot "aab"$$
$$y = x$$
$$w = u \cdot "b"$$
$$x \cdot v \neq v \cdot w$$
$$v \neq z$$

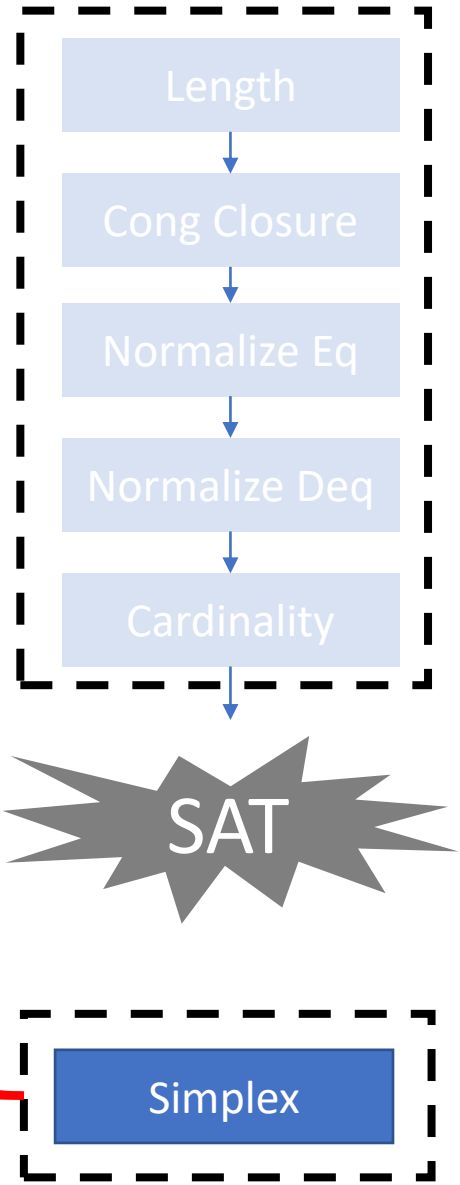


- If all steps finish with no new lemmas:
  - M_s is T_s-satisfiable
  - Model can be computed based on normal forms
    - String constants assigned to eq classes whose normal form is a variable
      - Length fixed by model from arithmetic solver
    - Each variable interpreted as the valuation of the normal form of their eq class

String Solver: Return SAT



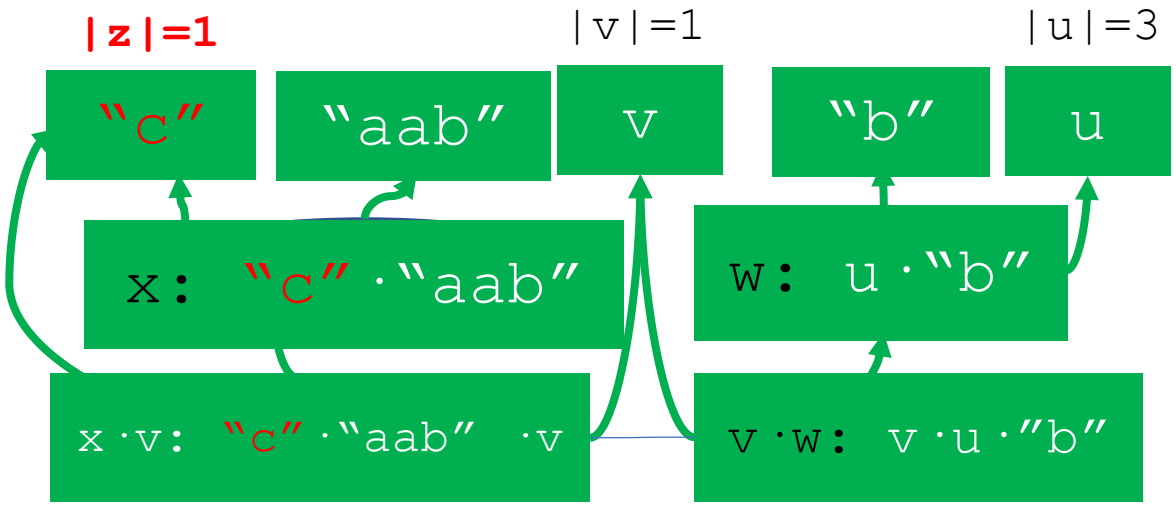
x=z·"aab"
y=x
w=u·"b"
x·v≠v·w
v≠z



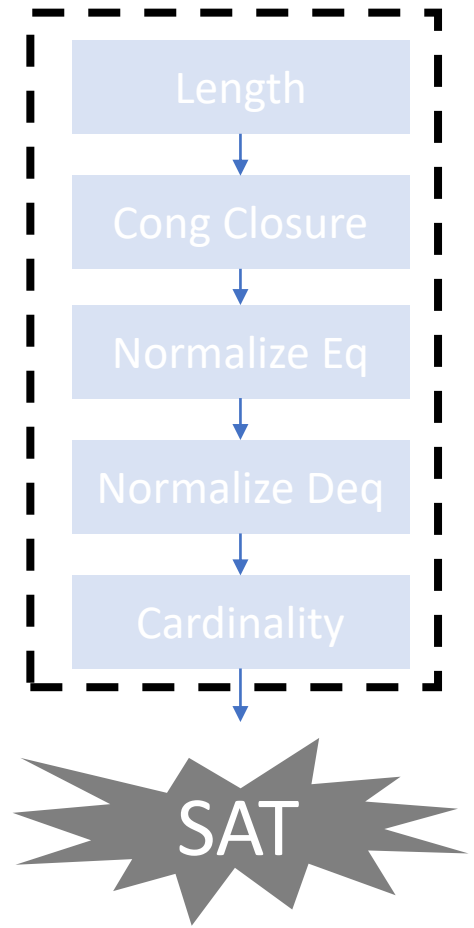
model

- For example:

String Solver: Return SAT

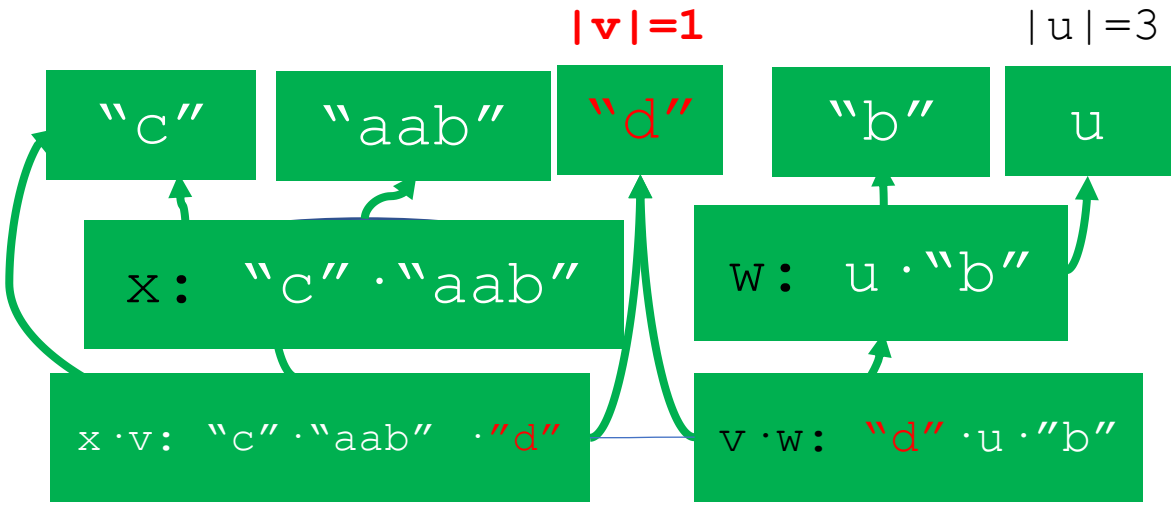


$$x = z \cdot "aab"$$
$$y = x$$
$$w = u \cdot "b"$$
$$x \cdot v \neq v \cdot w$$
$$v \neq z$$

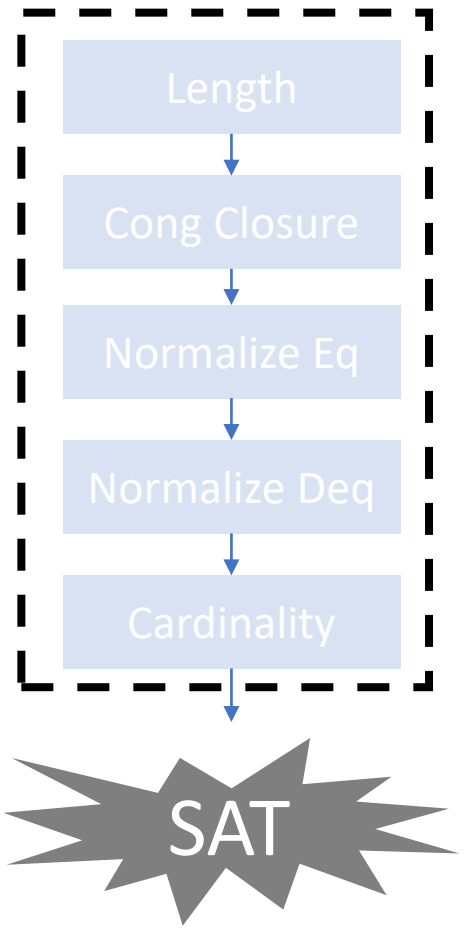


- For example:
  - z assigned to "c"

String Solver: Return SAT

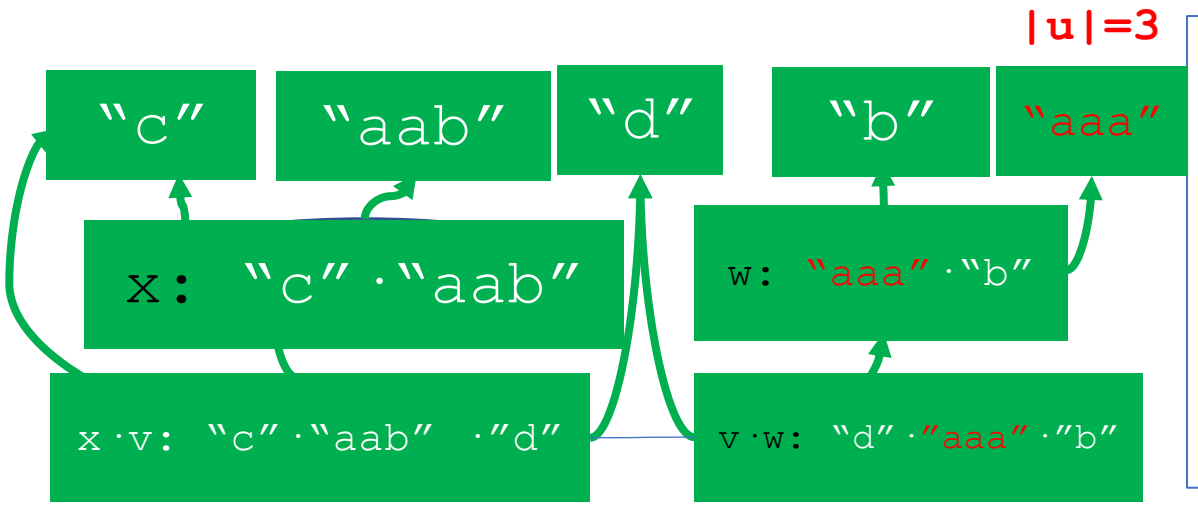


$$x = z \cdot \text{“aab”}$$
$$y = x$$
$$w = u \cdot \text{“b”}$$
$$x \cdot v \neq v \cdot w$$
$$v \neq z$$

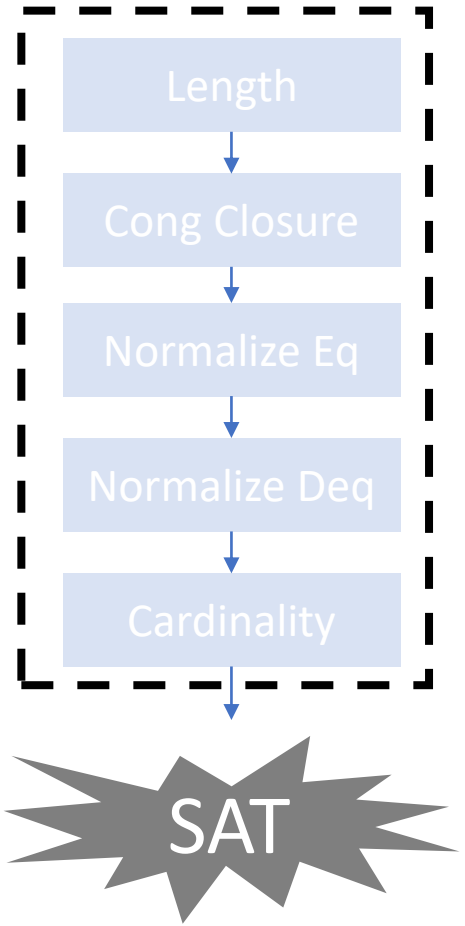


- For example:
  - z assigned to "c"
  - v assigned to "d"

String Solver: Return SAT



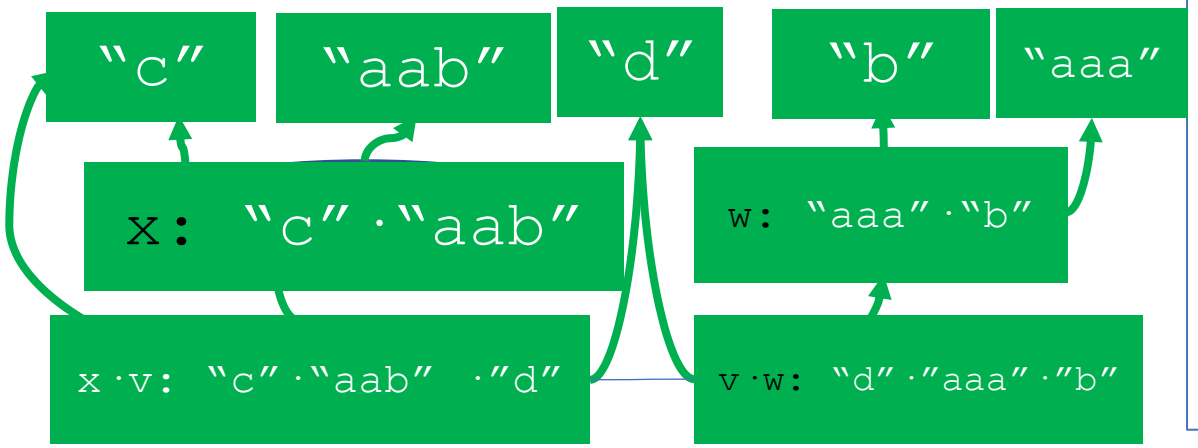
$$x = z \cdot "aab"$$
$$y = x$$
$$w = u \cdot "b"$$
$$x \cdot v \neq v \cdot w$$
$$v \neq z$$



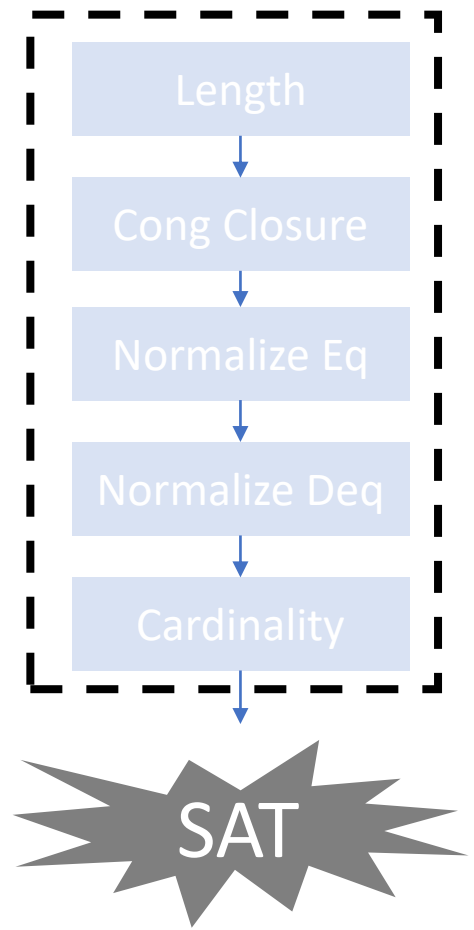
- For example:
  - z assigned to "c"
  - v assigned to "d"
  - u assigned to "aaa"

…cardinality step ensures enough fresh constants exist

String Solver: Return SAT

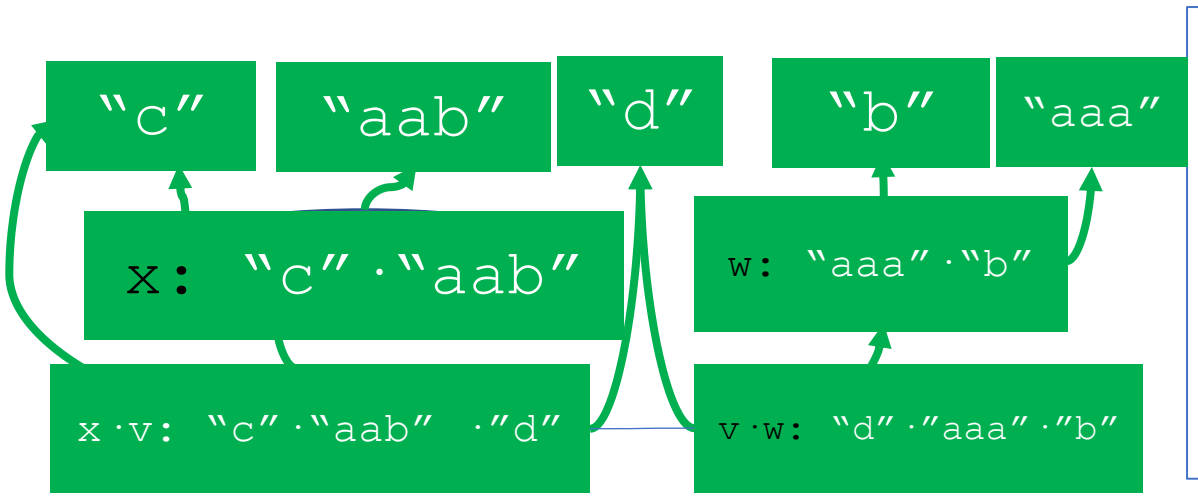


$$x = z \cdot "aab"$$
$$y = x$$
$$w = u \cdot "b"$$
$$x \cdot v \neq v \cdot w$$
$$v \neq z$$

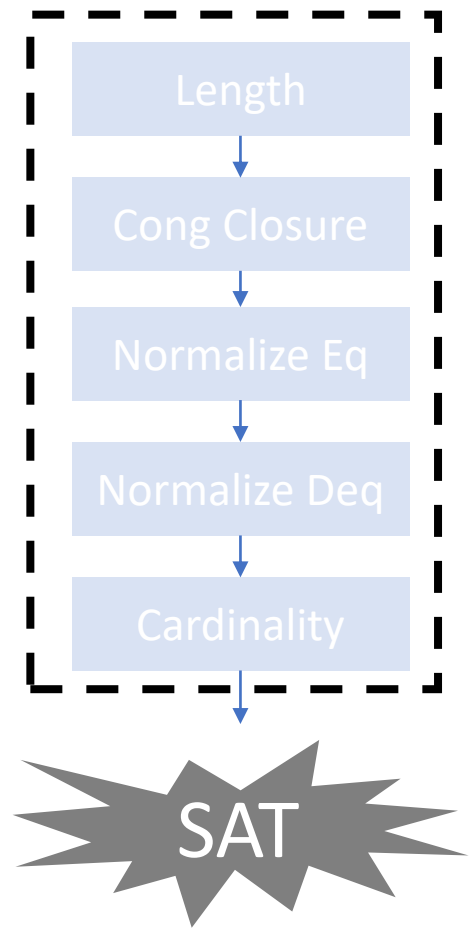


- For example:
  - z assigned to "c"
  - v assigned to "d"
  - u assigned to "aaa"
  - Variables assigned to value of the normal form of their eq classes:
    - x, y assigned to "caab", w assigned to "aaab"

String Solver: Return SAT



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



- For 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

Important Solving Techniques in CVC5

- Finite model finding
- Context-dependent simplification for extended string constraints
- Witness Sharing
- Regular expression elimination
- A procedure for string-to-code point conversion

Finite Model Finding for Strings

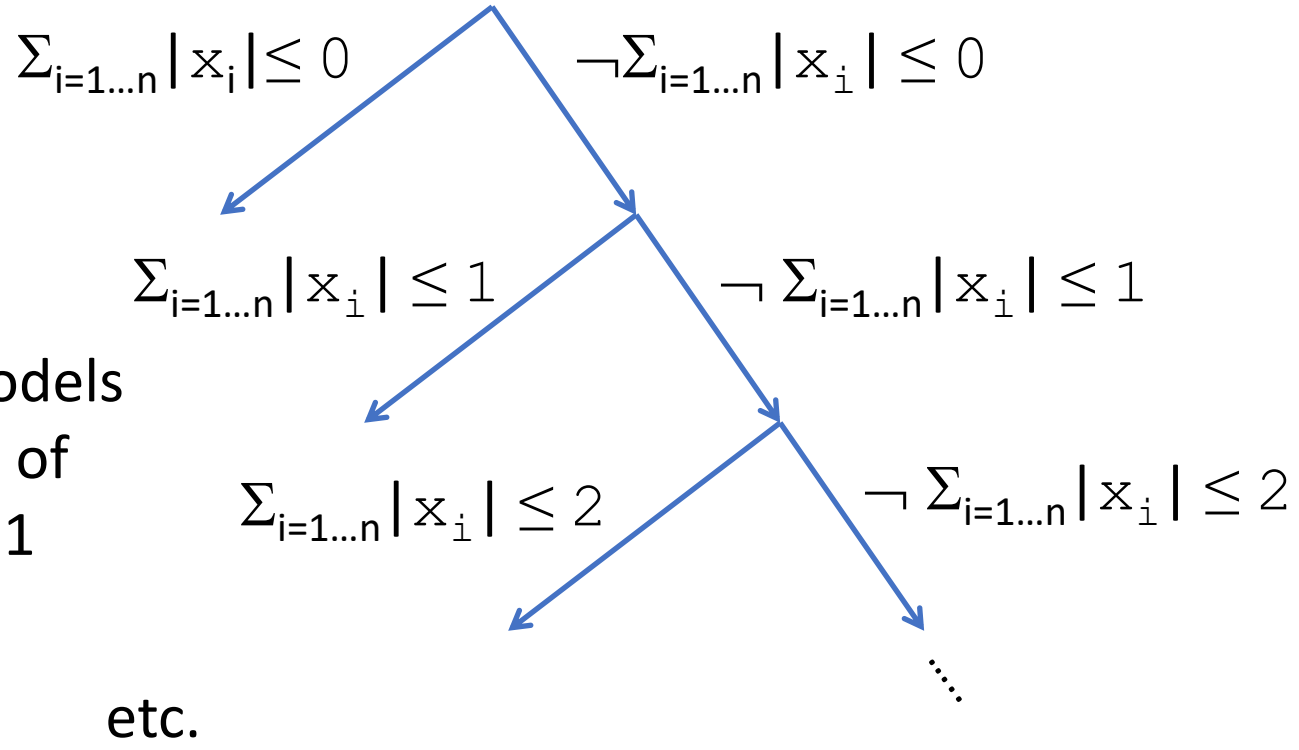
Finite Model Finding for Strings

- **Idea:** Incrementally bound the lengths of input string variables $x_i, \ldots, x_n$

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

Search for models where sum of lengths is 0

Search for models where sum of lengths is 1



etc.

Context-Dependent Simplification for Extended String Constraints

[Reynolds/Woo/Barrett/Brumley/Liang/Tinelli CAV 17]

Extended String Constraints

- Terms:
  - *Basic* terms
    - String and integer variables, constants, concatenation, length, and LIA-terms
  - *Extended* string terms:
    - Substring: `substr(x,1,3)`
      - i.e. the substring of `x` starting at position `1` of length at most `3`
    - String contains: `contains(x,"abc")`
      - i.e. true iff `x` contains the substring `"abc"`
    - Find "index of": `indexof(x,"d",5)`
      - i.e. the position of the first occurrence of `"d"` in `x`, starting from position `5`, or `−1` if it does not exist
    - String replace: `replace(x,"a","b")`
      - i.e. the result of replacing the first occurrence of `"a"` in `x` (if it exists) with `"b"`
- Constraints are:
  - Equalities and disequalities between extended string terms
  - Linear arithmetic inequalities

Example: $\neg$`contains(substr(x,0,3),"a")` $\wedge$ `0≤indexof(x,"ab",0)<4`

How do we handle Extended String Constraints?

```
¬contains(x,“a”)
```

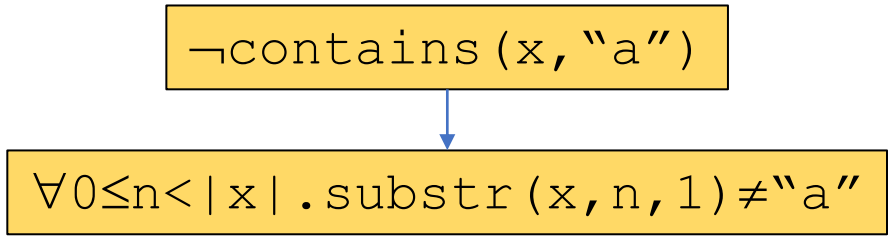
How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀

```
¬contains(x,“a”)
```

How do we handle Extended String Constraints?

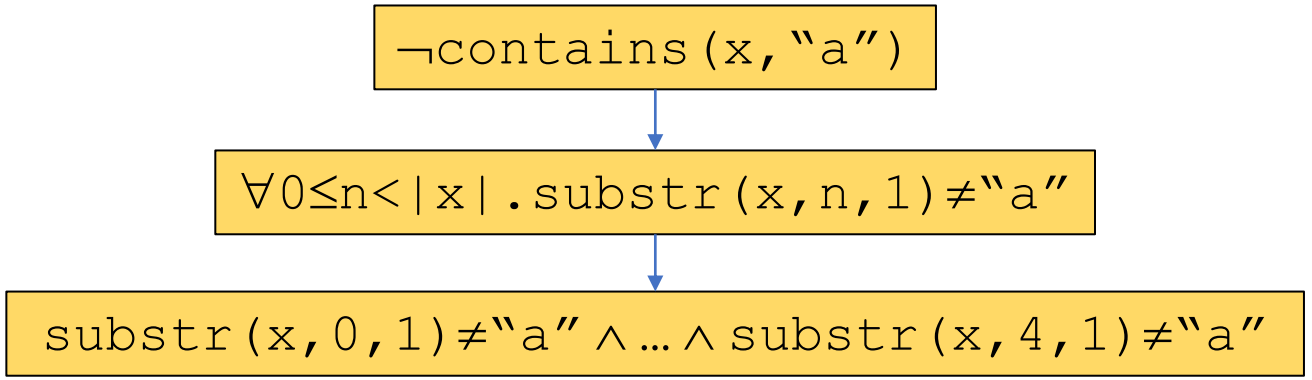
- Naively, by reduction to basic constraints + bounded ∀



Expand `contains`

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀

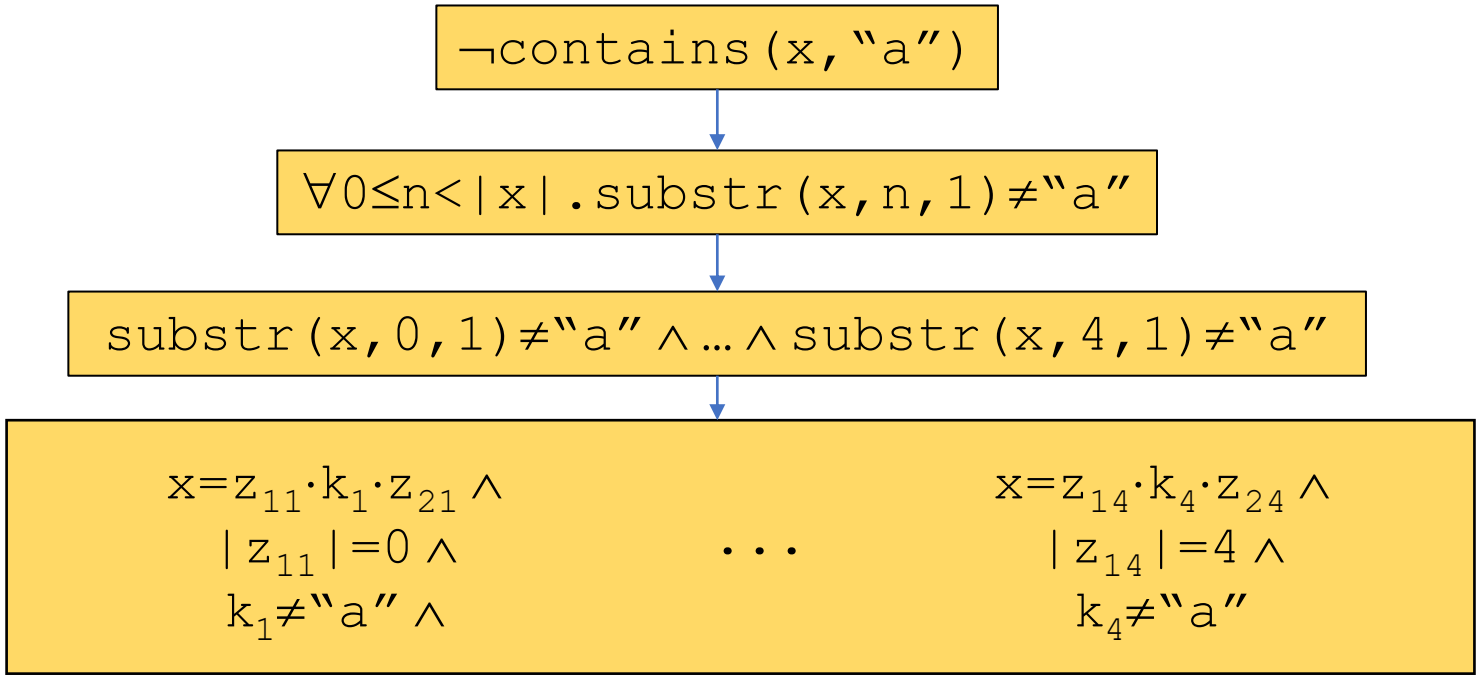


Expand `contains`

Assuming bound `|x|≤5`

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀



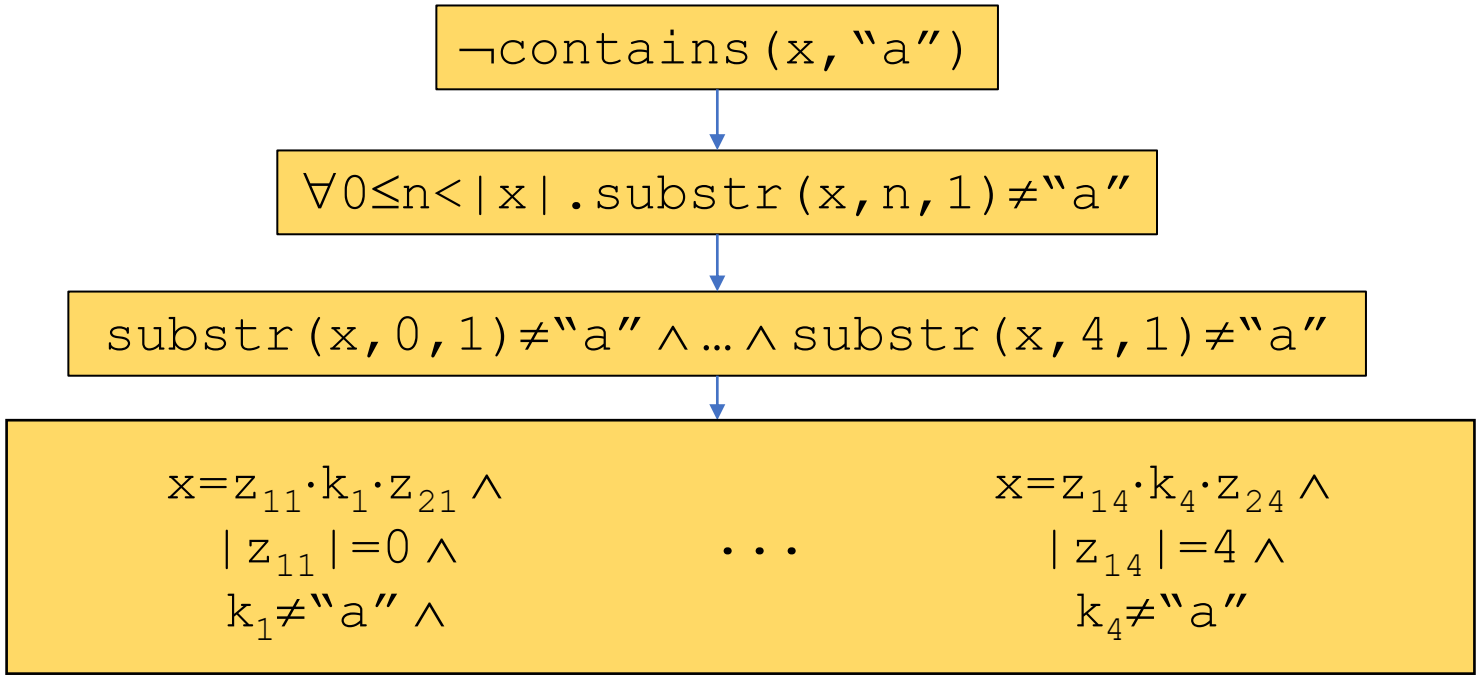
Expand `contains`

Assuming bound $|x| \leq 5$

Expand `substr`

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀



Expand `contains`

Assuming bound `|x|≤5`

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

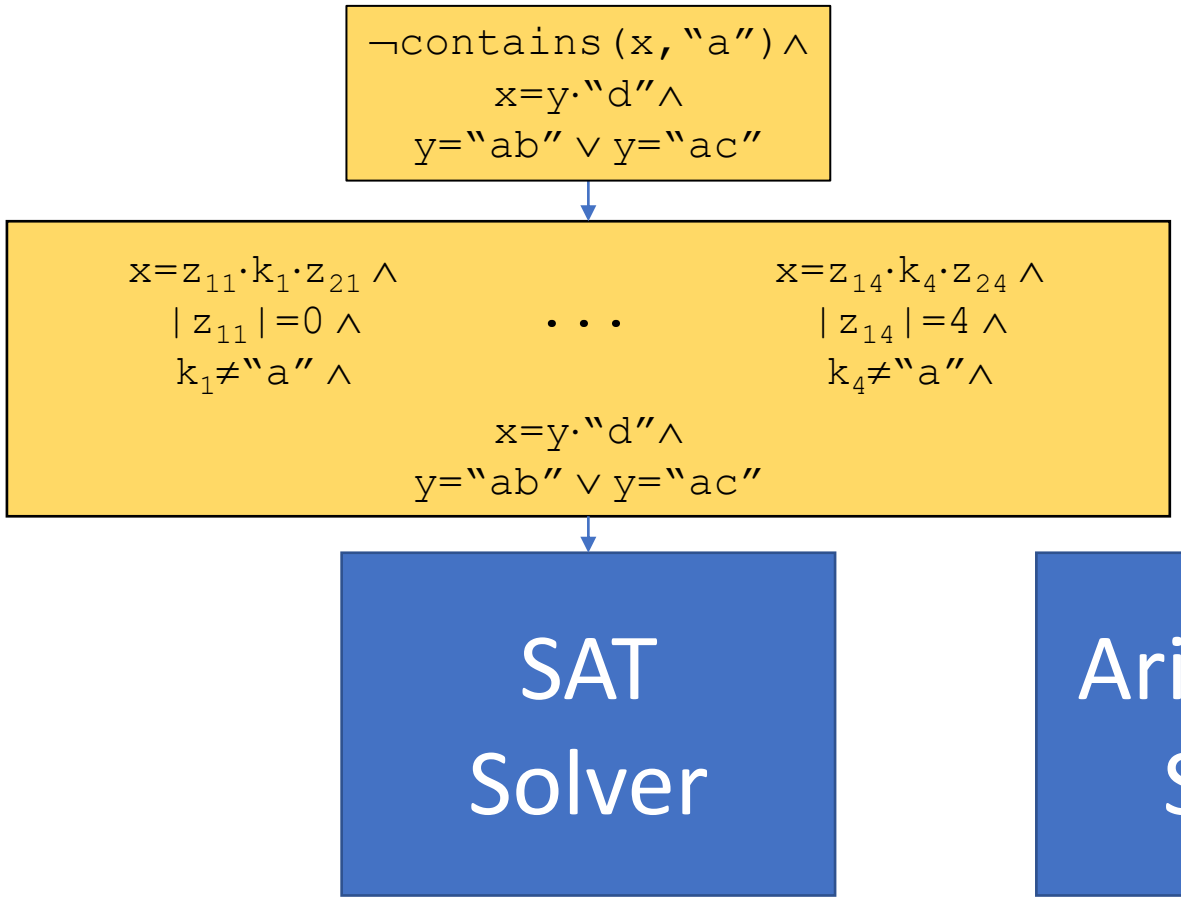
$\neg contains(x, \text{“a”}) \wedge$
$x = y \cdot \text{“d”} \wedge$
$y = \text{“ab”} \vee y = \text{“ac”}$

SAT Solver

Arithmetic Solver

String Solver

(Eager) Expansion of Extended Constraints



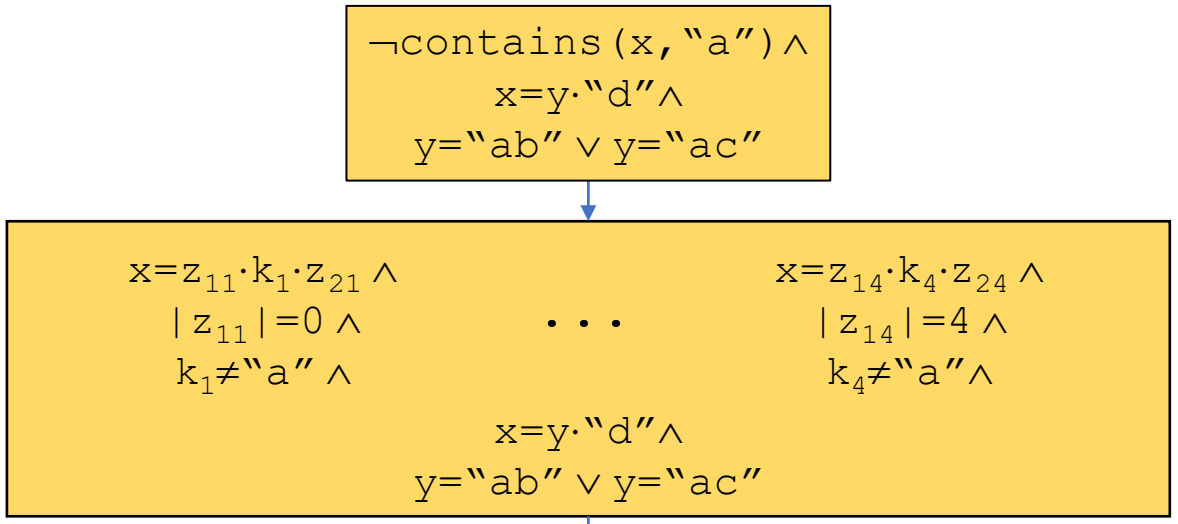
Expand and eliminate extended symbols

SAT Solver

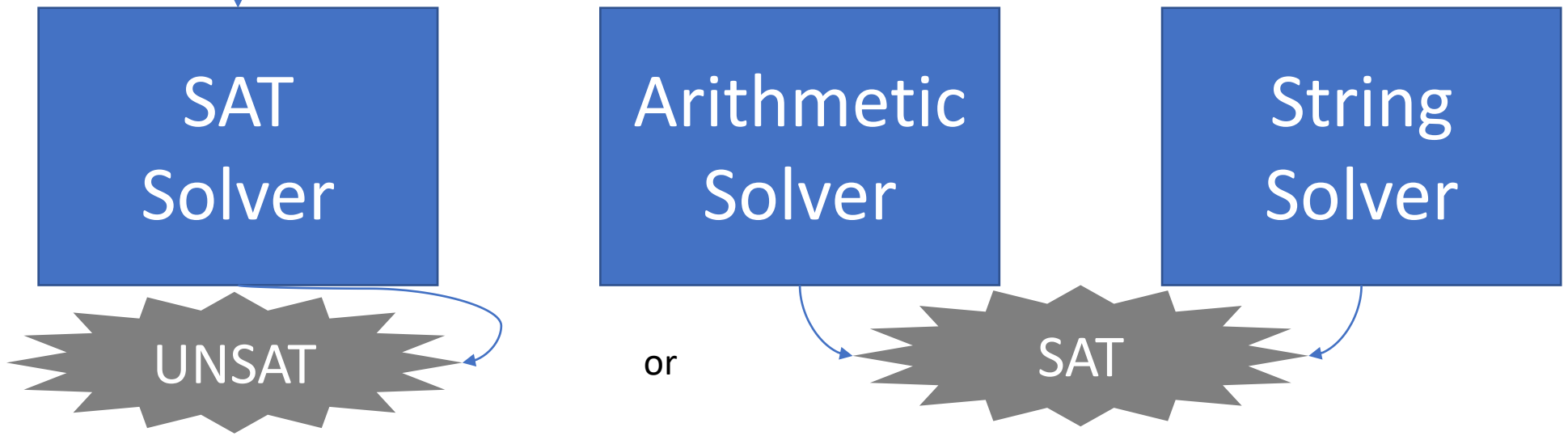
Arithmetic Solver

String Solver

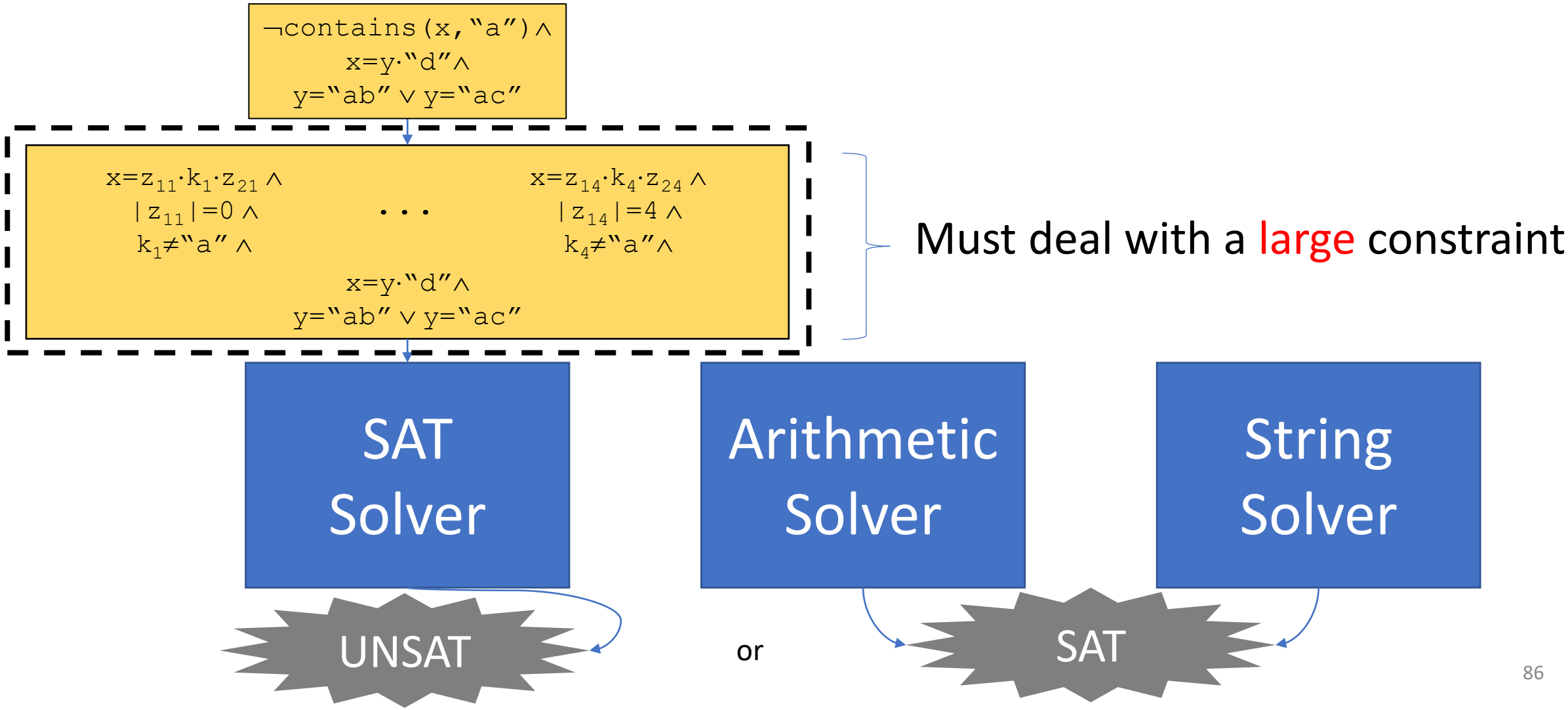
(Eager) Expansion of Extended Constraints



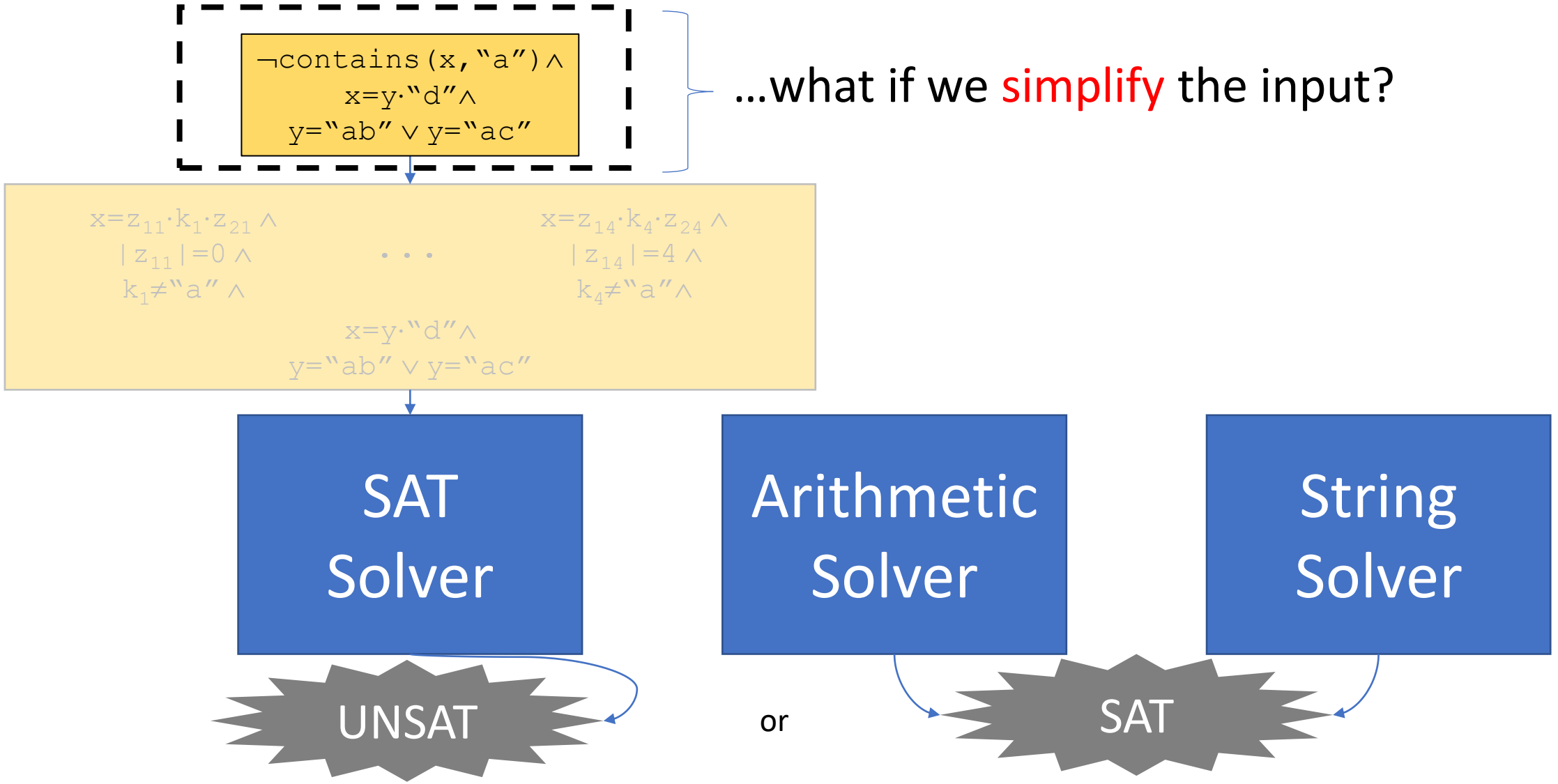
Expand and eliminate extended symbols



(Eager) Expansion of Extended Constraints



(Eager) Expansion of Extended Constraints



SMT Solvers + Simplification

- All SMT solvers implement *simplification* techniques

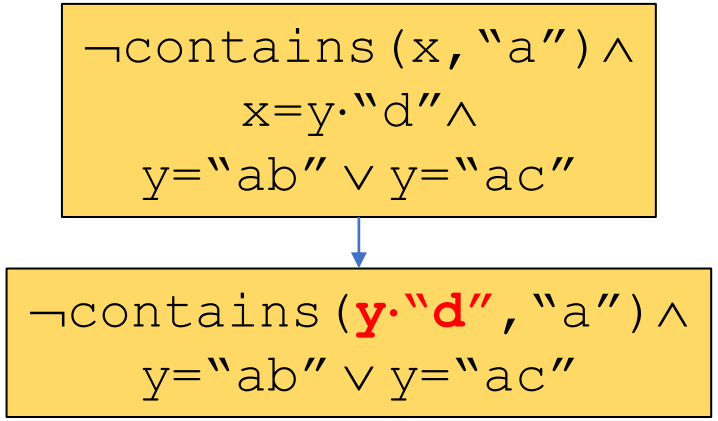
  (also called *normalization* or *rewrite* rules)

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

SMT Solvers + Simplification

- All SMT solvers implement *simplification* techniques

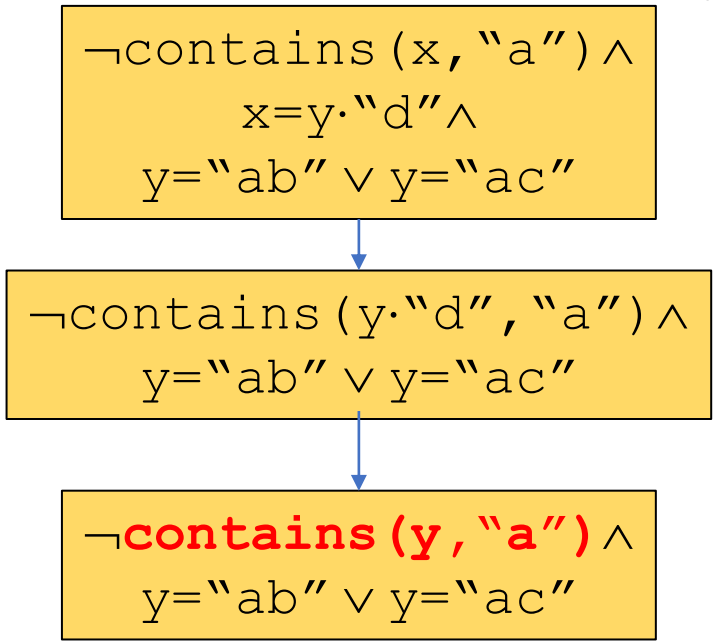
(also called *normalization* or *rewrite* rules)



since x=y·"d"

SMT Solvers + Simplification

- All SMT solvers implement *simplification* techniques
  (also called *normalization* or *rewrite* rules)



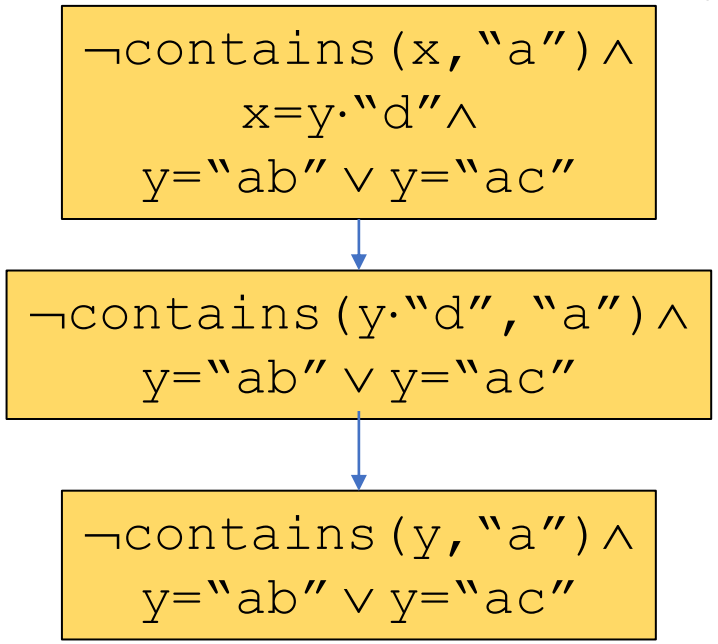
since x=y·"d"

since contains(y·"d","a")⇔contains(y,"a")

SMT Solvers + Simplification

- All SMT solvers implement *simplification* techniques

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



since `x=y·"d"`

since `contains(y·"d","a")⇔contains(y,"a")`

- Leads to smaller inputs
  - Some problems can be solved by simplification alone

(Lazy) Expansion + Simplification

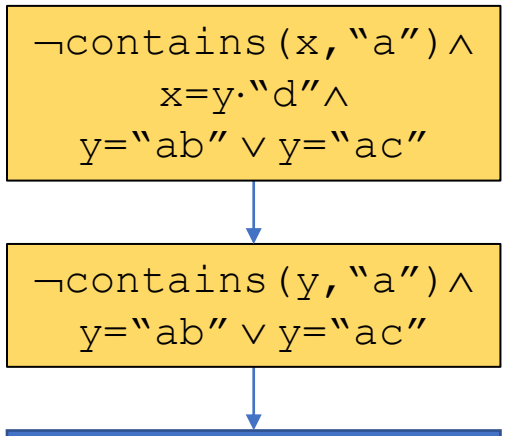
$\neg\text{contains}(x, \text{“a”}) \wedge$
$x = y \cdot \text{“d”} \wedge$
$y = \text{“ab”} \vee y = \text{“ac”}$

SAT Solver

Arithmetic Solver

String Solver

(Lazy) Expansion + Simplification



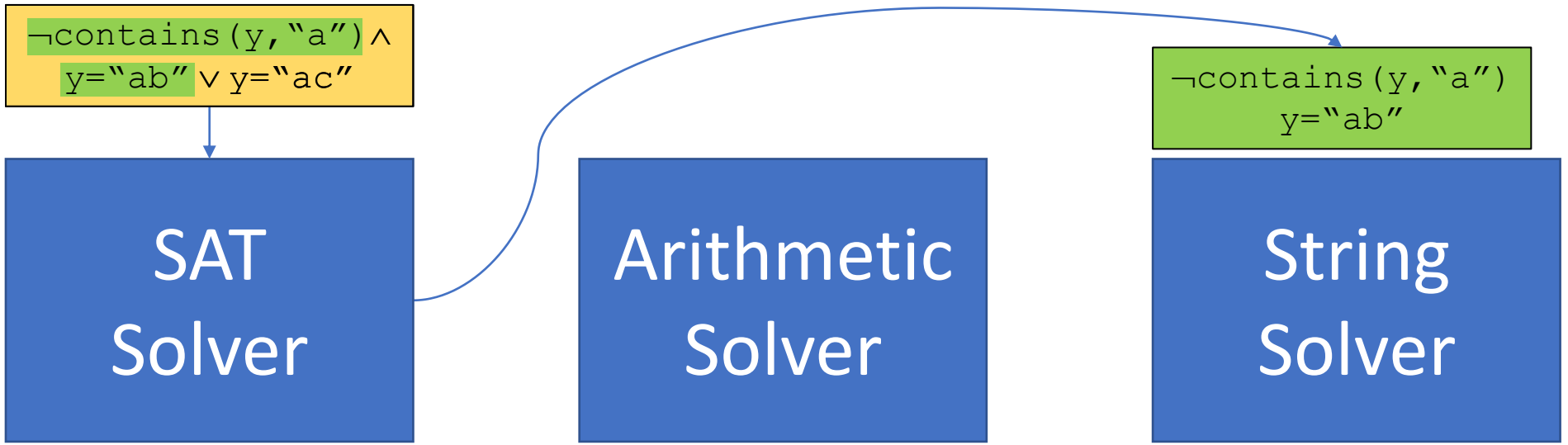
Simplify the input

SAT Solver

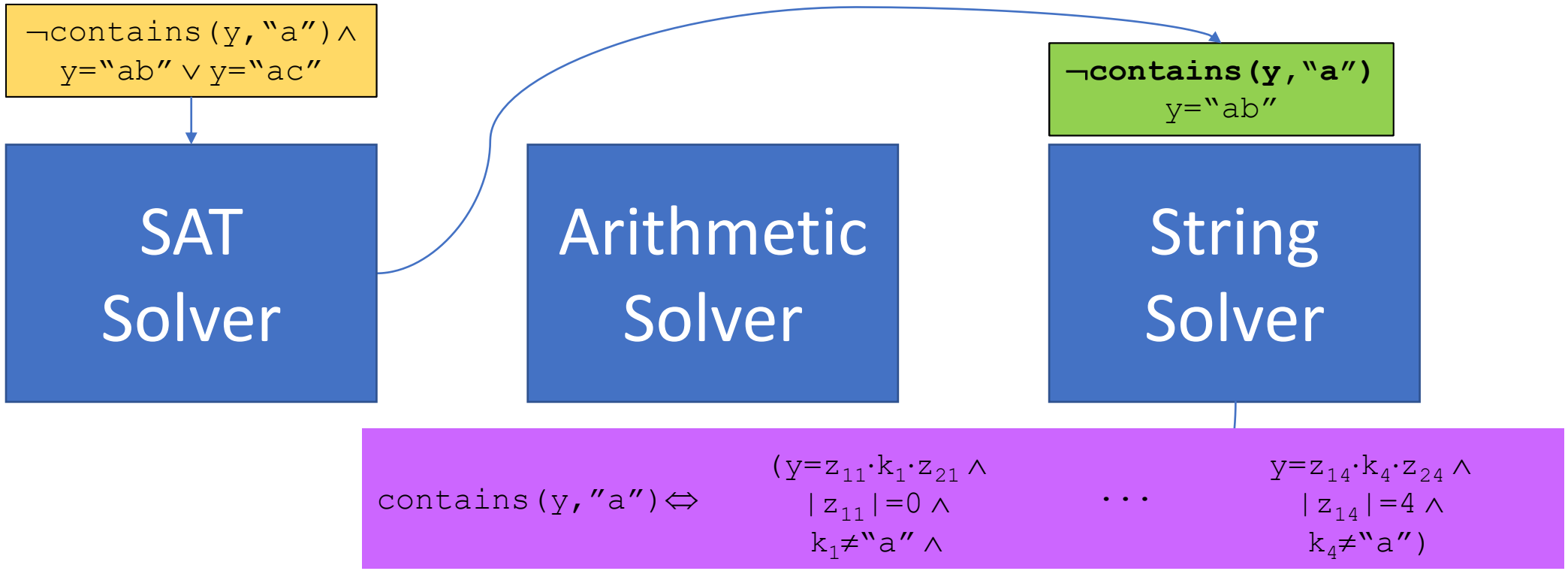
Arithmetic Solver

String Solver

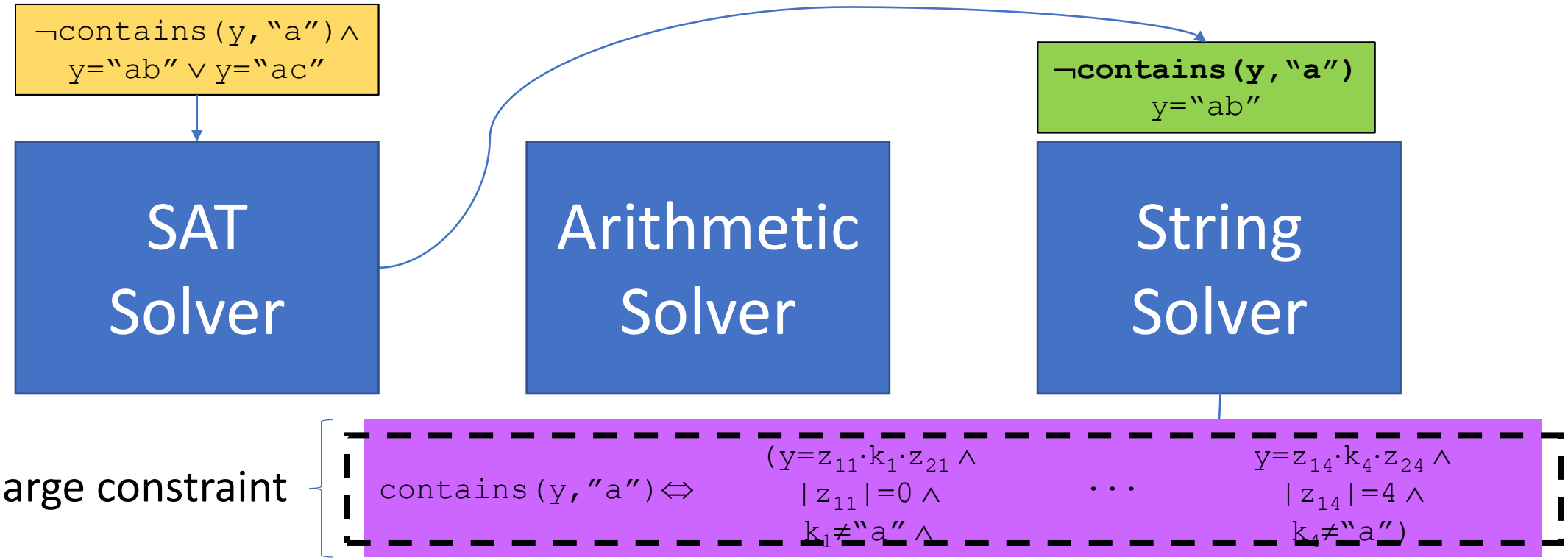
(Lazy) Expansion + Simplification



(Lazy) Expansion + Simplification



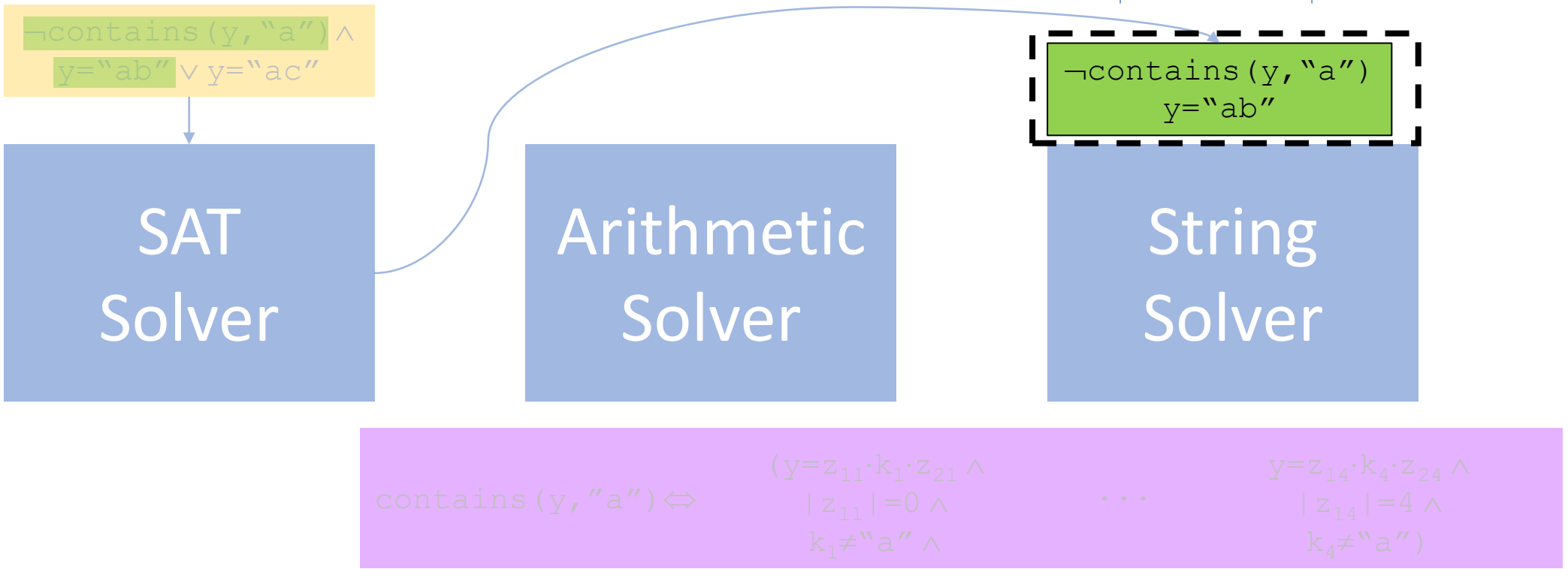
(Lazy) Expansion + Simplification



Still have a large constraint

(Lazy) Expansion + Simplification

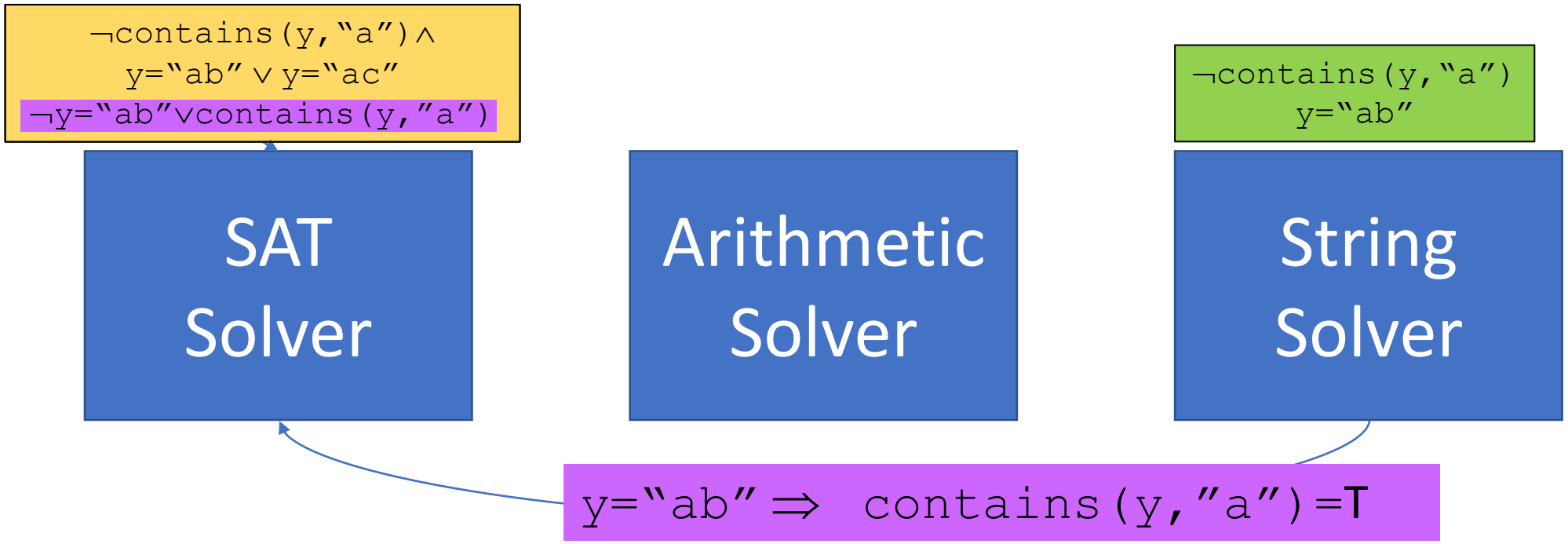
What if we simplify based on the context?



(Lazy) Expansion + Context-Dependent Simplification

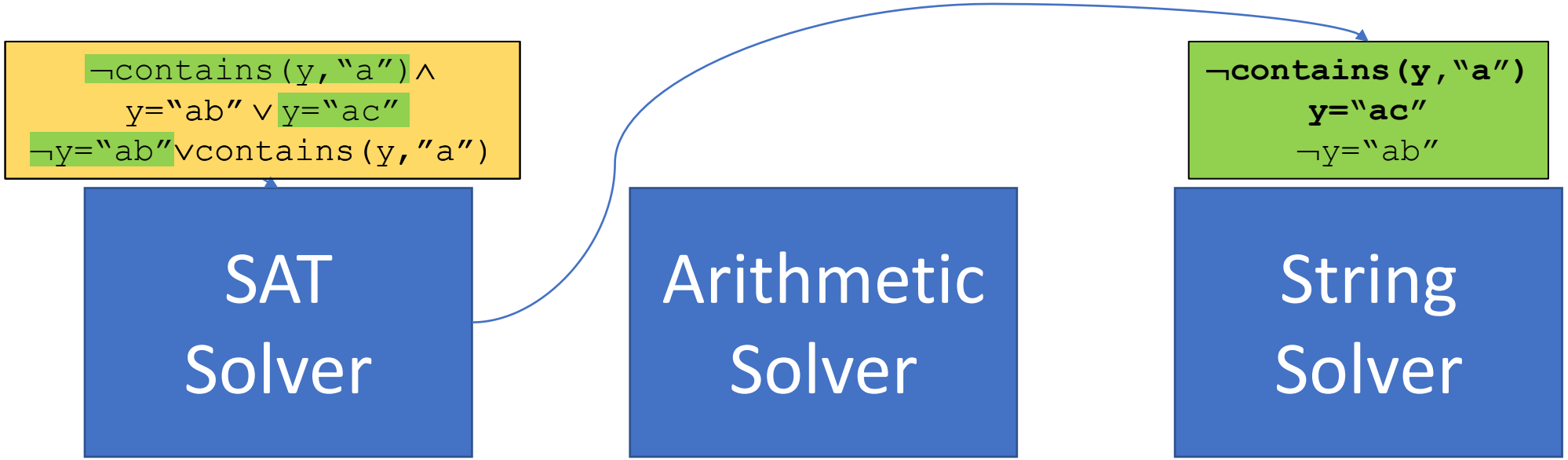
Since `contains(y,"a")` is true when `y="ab"` …

(Lazy) Expansion + Context-Dependent Simplification

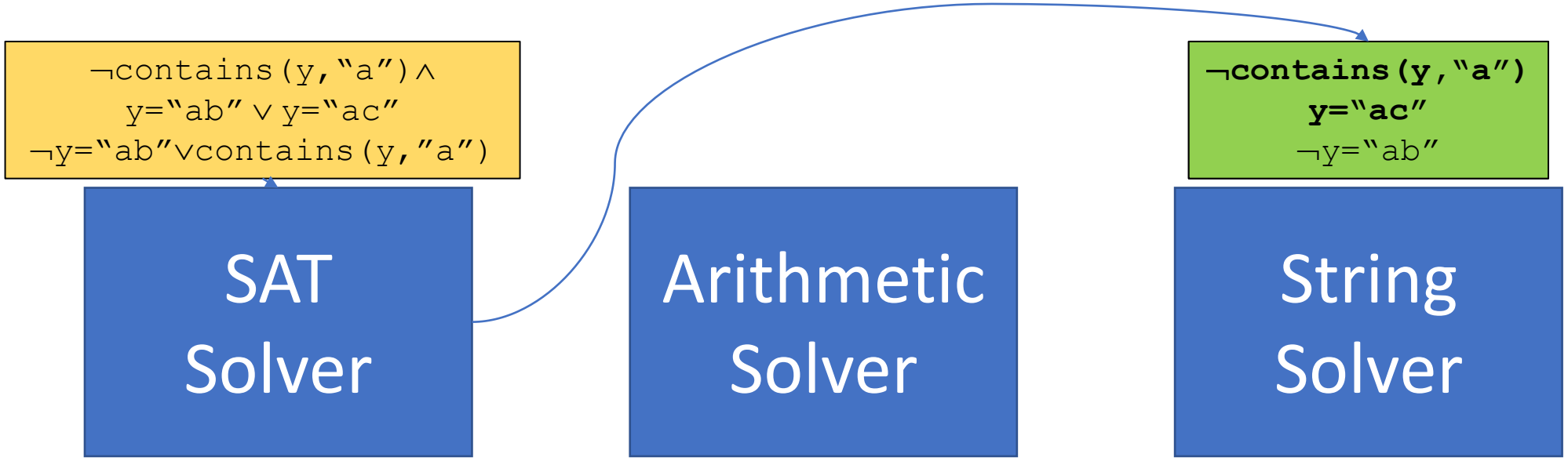


(Lazy) Expansion + Context-Dependent Simplification

(Lazy) Expansion + Context-Dependent Simplification

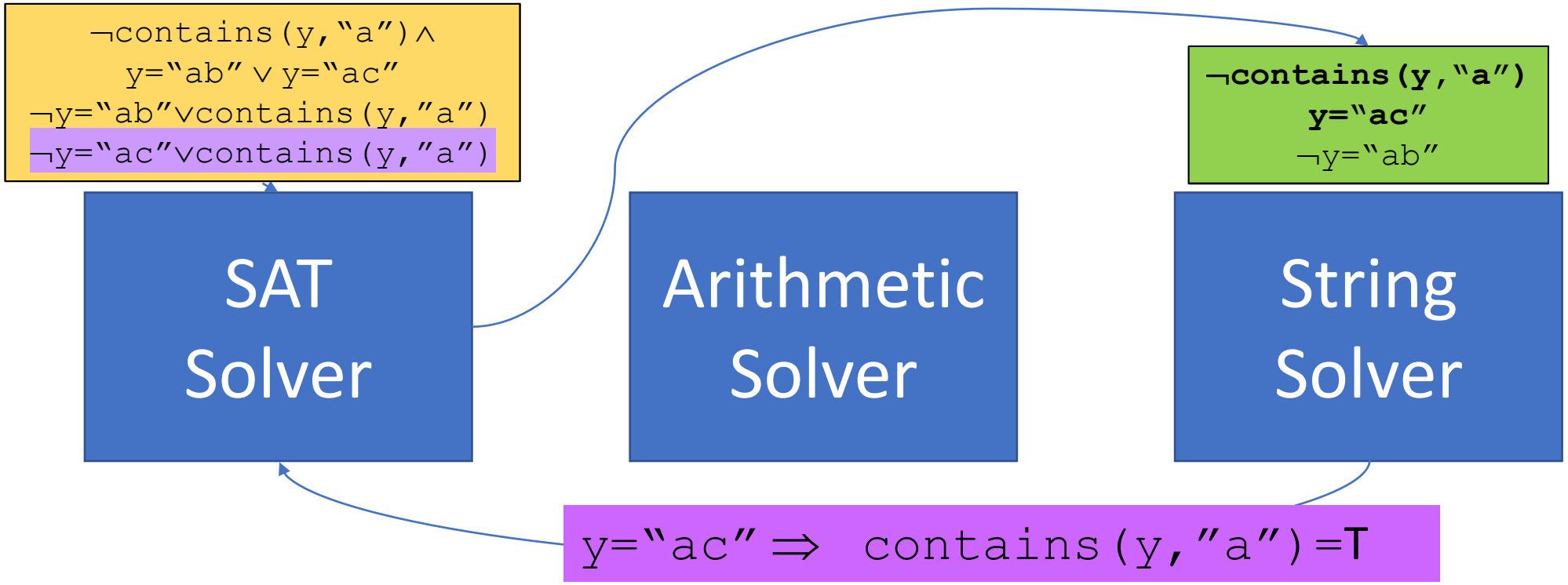


(Lazy) Expansion + Context-Dependent Simplification

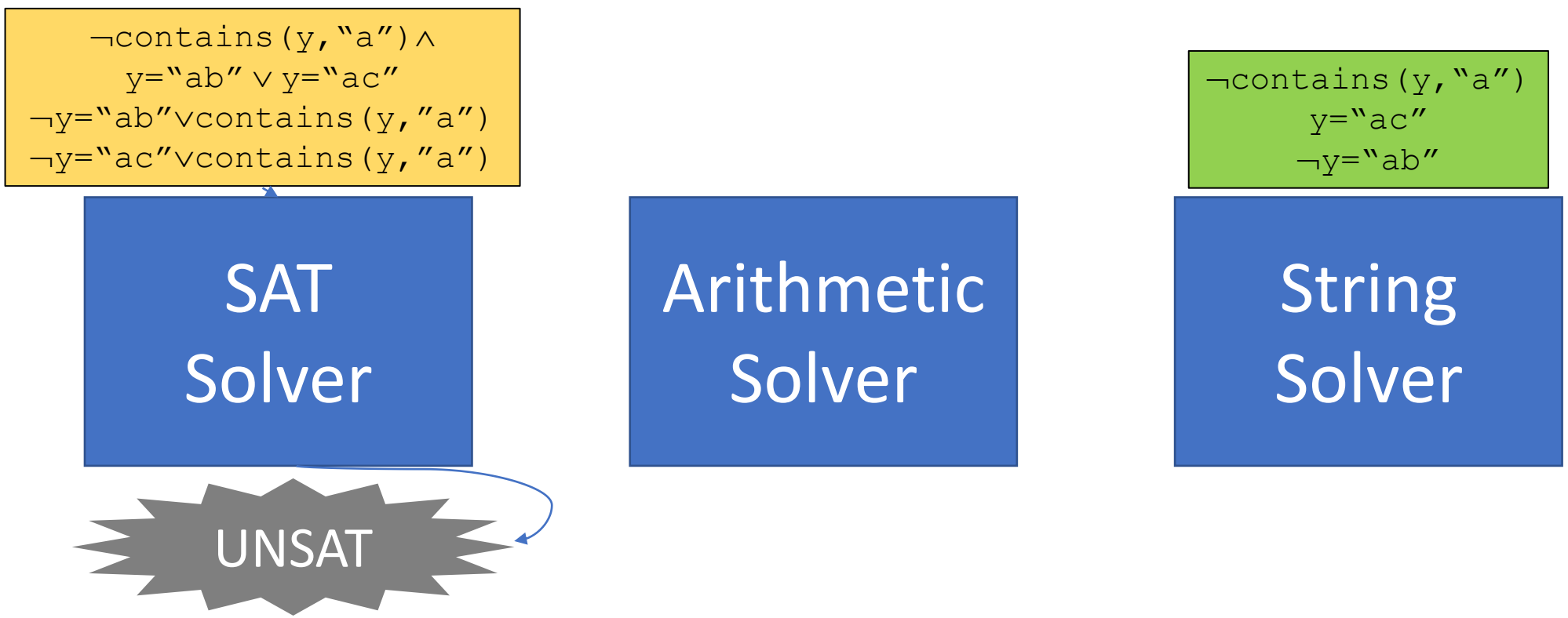


`contains(y,"a")` **is also true when** `y="ac"` **…**

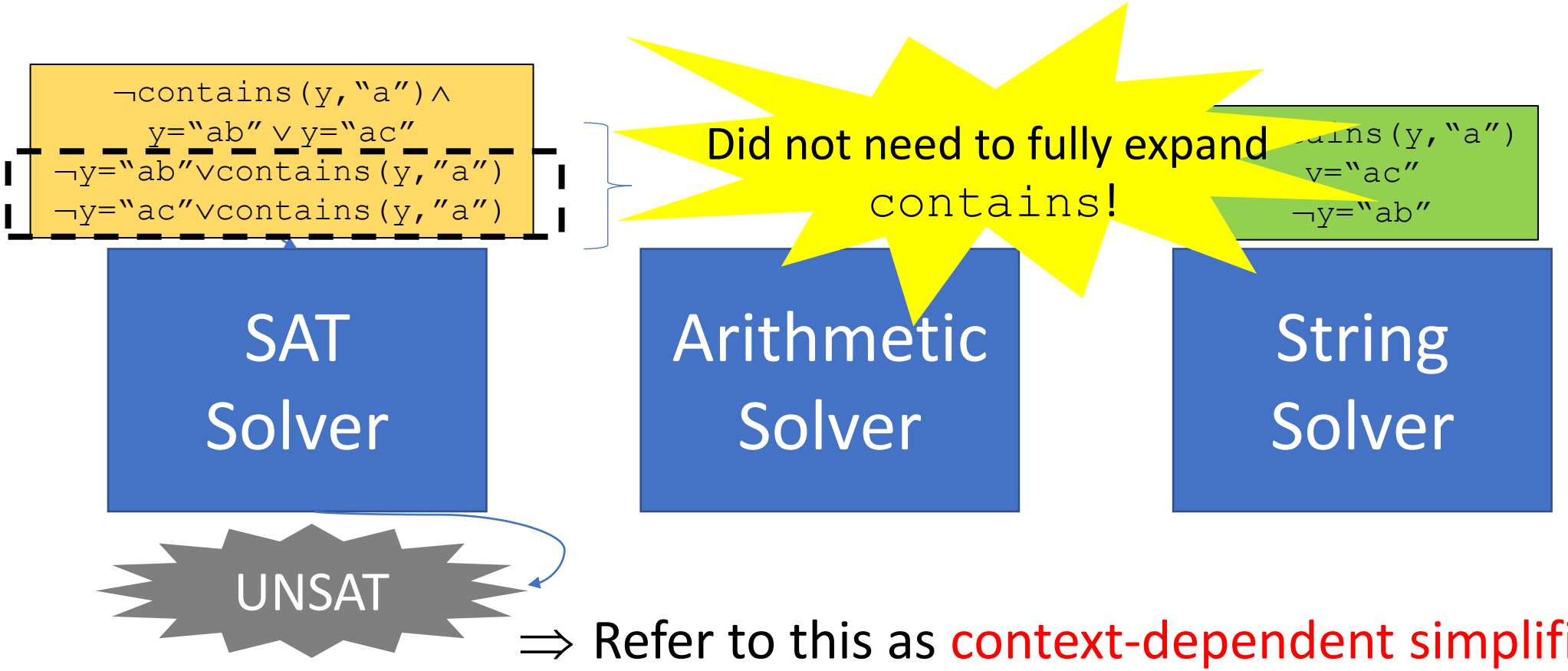
(Lazy) Expansion + Context-Dependent Simplification



(Lazy) Expansion + Context-Dependent Simplification



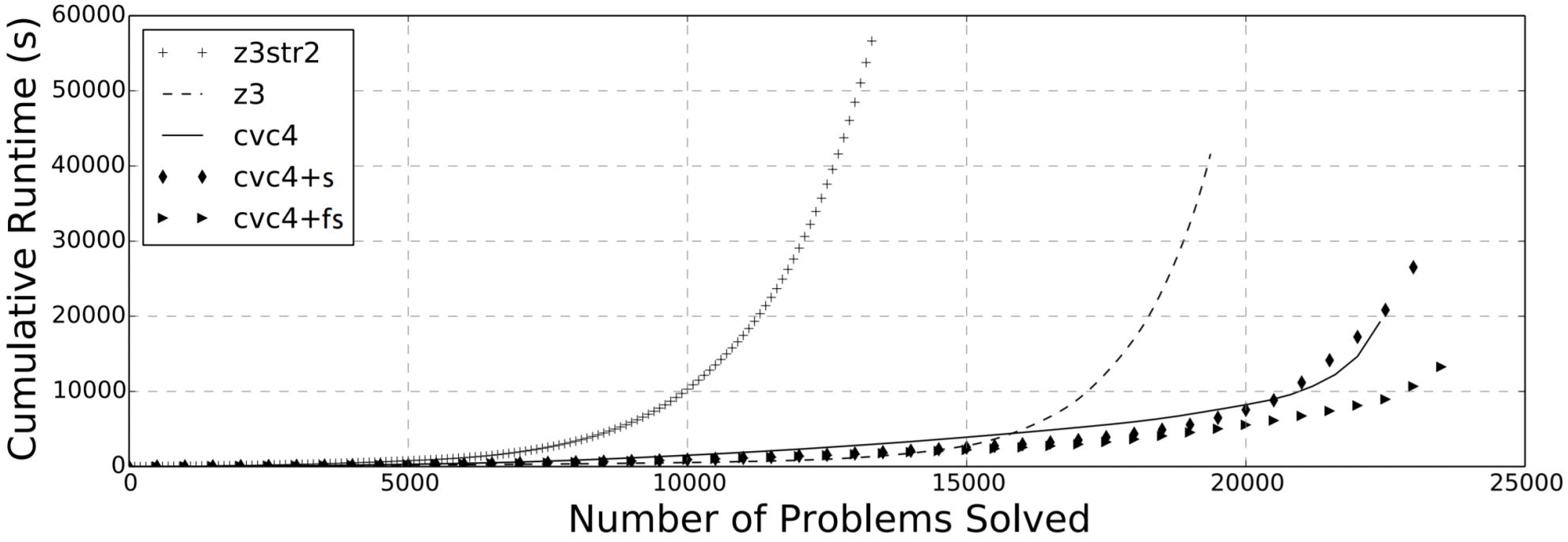
(Lazy) Expansion + Context-Dependent Simplification



⇒ Refer to this as context-dependent simplification

[Reynolds et al CAV 2017]

Results on Symbolic Execution [Reynolds et al CAV 17]



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

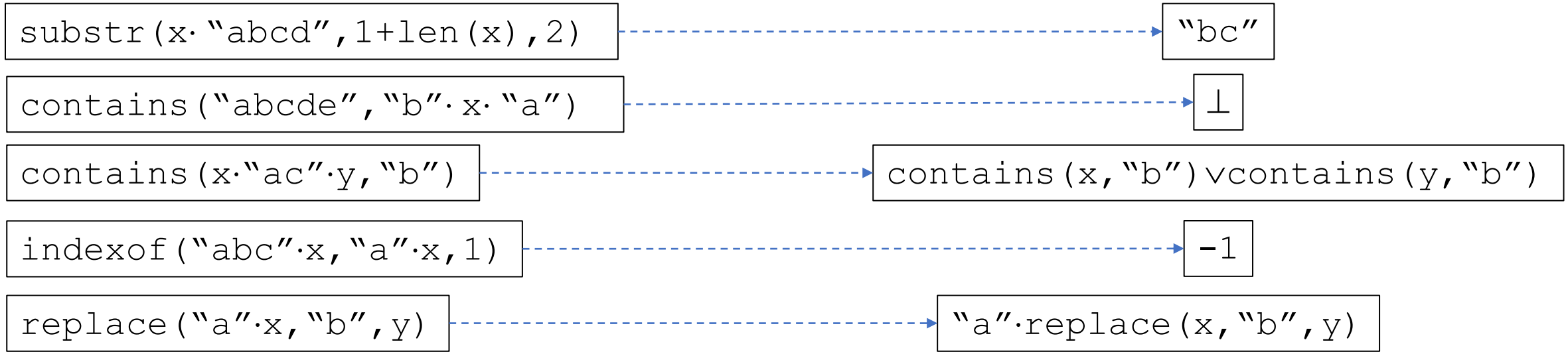
Aggressive Simpifications for Strings

[Reynolds/Noetzli/Tinelli/Barrett CAV 19]

Many Simplification Rules for Strings

- Unlike arithmetic:

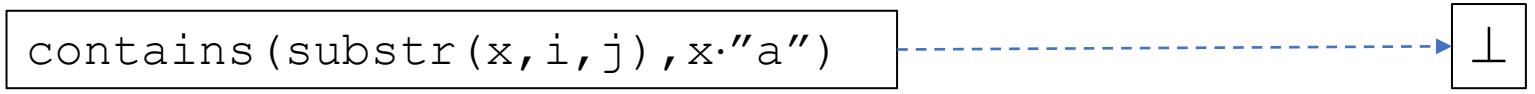
…simplification rules for strings are highly non-trivial:



Simplification based on High-Level Abstractions

[Reynolds/Noetzli/Tinelli/Barrett CAV 19]

- Rules based on high-level abstractions
  - When viewing strings as #characters (e.g. reasoning about their length):



…since the second argument is longer than the first

  - When considering the containment relationship between strings:

  - When viewing strings as multisets of characters:

…since LHS contains at least 1 more occurrences of "a"

Impact of Aggressive Simplification

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

[Reynolds/Noetzli/Tinelli/Barrett CAV 19]

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

- CVC5 implements >3000 lines of C++ for simplification rules (and growing)
  - Important aspect of modern string solving

Witness Sharing + RE elim

[Reynolds/Noetzli/Tinelli/Barrett FMCAD 20]

Witness Sharing

- **Observation:** equivalent ways of expressing the same thing
  - For strings `x`,`y`:

$$\exists z.x=z\cdot y \wedge \mathtt{len}(z)=1$$
$$\mathtt{substr}(x,\ 1,\ \mathtt{len}(x)-1)=y$$
$$x \in \Sigma\cdot\mathtt{to\_re}(y)$$

- Solving word equations, extended functions, RE introduce fresh variables

- **Idea:** Formalize the definition for each introduced variable “witness form”
  - Reuse variables whose witness forms are semantically equivalent

Witness Sharing (Example)

$$\frac{x \cdot w = \text{“A”} \cdot u \qquad |x| \neq 0}{x = \text{“A”} \cdot k_1}$$

$$\frac{x \in \Sigma \cdot R}{x = k_2 \cdot k_3 \ \wedge \ k_2 \in \Sigma \wedge \ k_3 \in R}$$

Witness Sharing (Example)

$$\frac{x \cdot w = \text{“A”} \cdot u \qquad |x| \neq 0}{x = \text{“A”} \cdot \mathbf{k_1}}$$

substr(x, 1, len(x)-1)

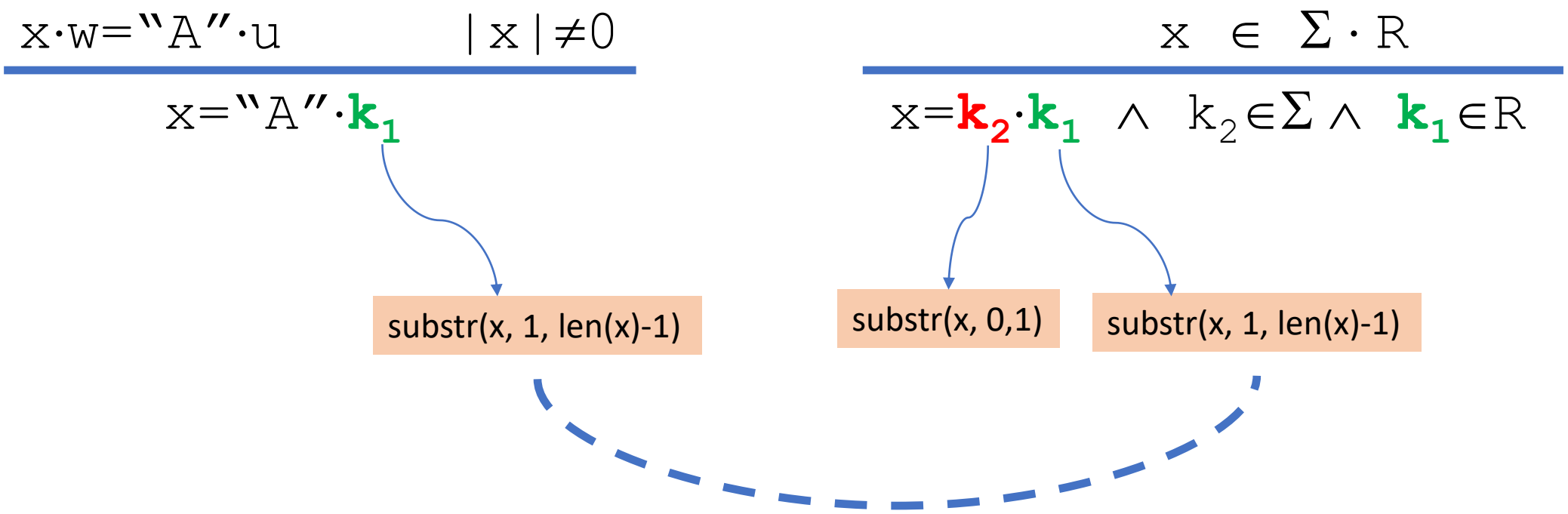
$$\frac{x \in \Sigma \cdot R}{x = \mathbf{k_2} \cdot \mathbf{k_3} \wedge k_2 \in \Sigma \wedge k_3 \in R}$$

substr(x, 0,1)

substr(x, 1, len(x)-1)

“Witness forms”

Witness Sharing (Example)



- Can reuse variables whose witness form are (semantically) equivalent
  - ⇒ *Can use aggressive simplification to show witness forms are equivalent*

Regular Expression Elimination

- **Idea:** reduce RE to extended string constraints
  - Possible for many regular expression memberships that occur in practice:

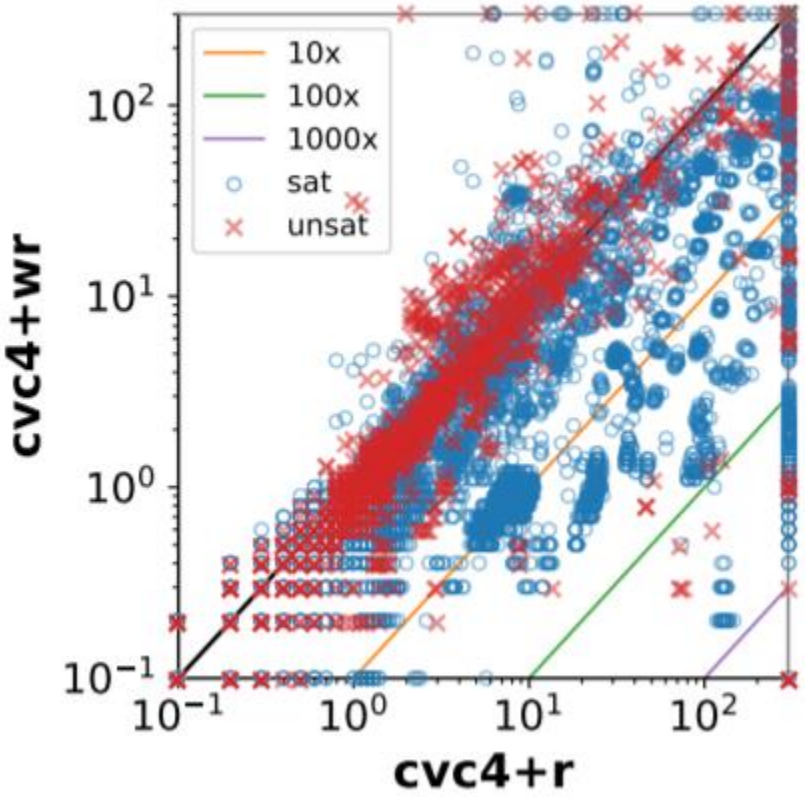
$x \in (\Sigma \cdot \Sigma^* \cdot \Sigma) \Leftrightarrow |x| \geq 2$

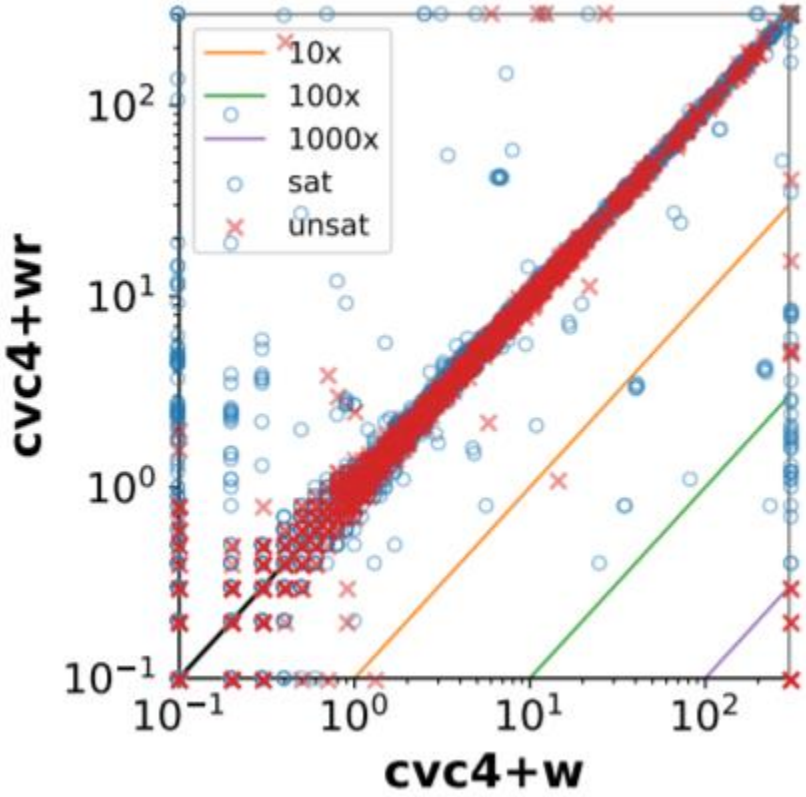
$x \in (\Sigma^* \cdot "abc" \cdot \Sigma^*) \Leftrightarrow \texttt{contains}(x, "abc")$

$x \in (\Sigma^* \cdot "a" \cdot \Sigma^* \cdot "bcd" \cdot \Sigma^*) \Leftrightarrow$

```
contains(x,"a")∧
contains(substr(x,indexof(x,"a",1)+1,|x|),
         "bcd")
```

Impact of Witness Sharing + RE elim

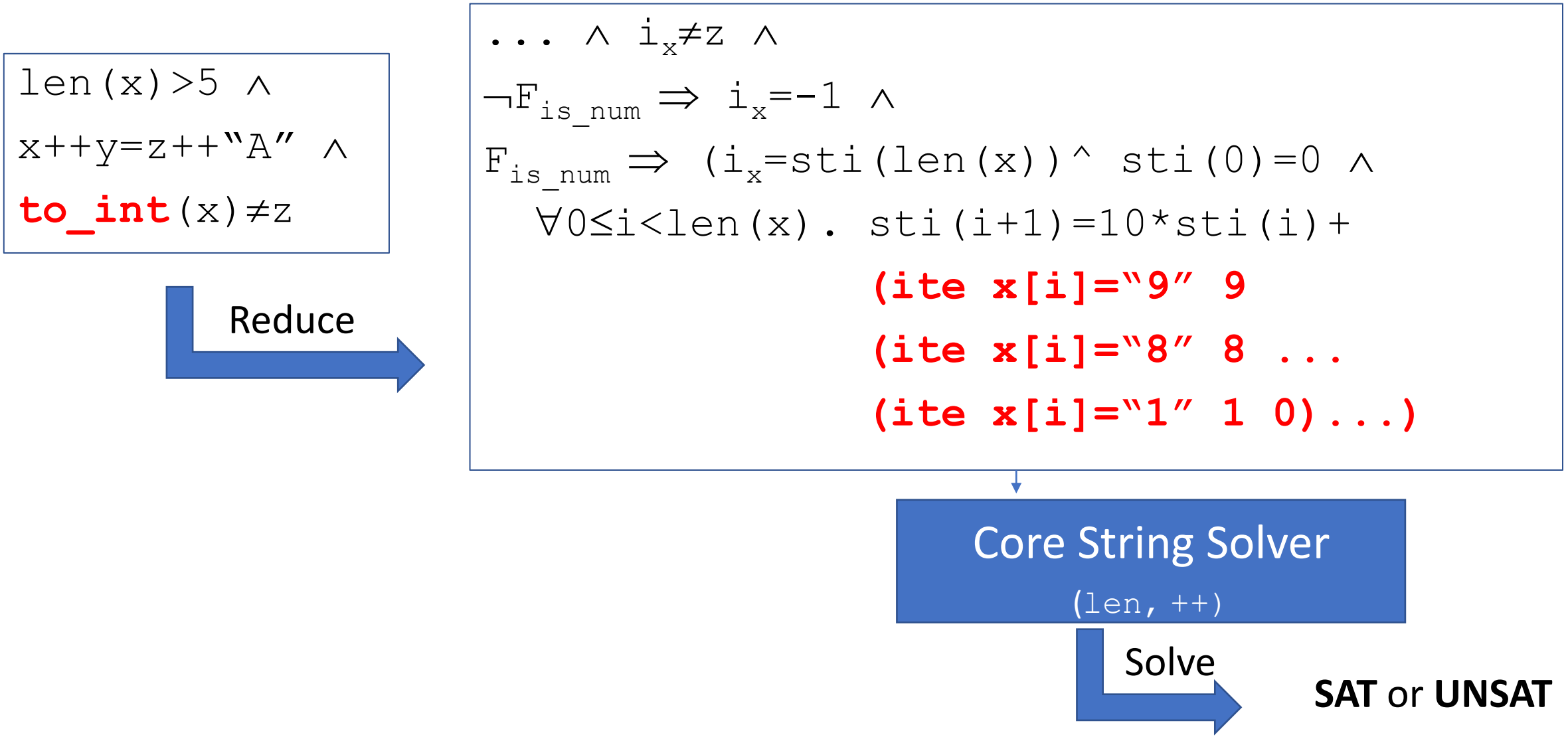




String to Code Point Conversion

[Reynolds/Noetzli/Tinelli/Barrett IJCAR 20]

SMT Solvers for Strings + Extended Functions



SMT Solvers for Strings + Extended Functions (via code)

$len(x)>5 \wedge$
$x{+}{+}y=z{+}{+}\text{“A”} \wedge$
$to\_int(x)\neq z$

Reduce

$\ldots \wedge i_x \neq z \wedge$
$\neg F_{is\_num} \Rightarrow i_x=-1 \wedge$
$F_{is\_num} \Rightarrow (i_x=sti(len(x)) \wedge sti(0)=0 \wedge$
$\forall 0 \leq i < len(x).\ sti(i+1)=10*sti(i)+$
(code(x[i])-48)

- Extend core solver for string-to-code point
- Enable efficient reductions for constraints in common applications

Core String Solver
(len, ++, ***code***)

Solve

SAT or **UNSAT**

String to code operator **code**

- Assume ordering on characters of alphabet $\mathcal{A}$:
  - $c_1 < \ldots < c_{|\mathcal{A}|-1}$
  - For each c_i, we call i its code point
- $\texttt{code} : \texttt{Str} \rightarrow \texttt{Int}$ is interpreted as:
  1. For w in $\mathcal{A}^1$, $\texttt{code}(w)$ is the code point of the single character in w
  2. For all other w, $\texttt{code}(w)$ is -1

- Fragment with string length + string code point (w/o concatenation):
  - Procedure [Reynolds et al IJCAR20] is **sound, complete, terminating**

Reductions: Conversion Functions

- More efficient reductions that leverage `code`, including:

- Conversion between strings and integers `to_int(x)`:
  - ⊗ `… ite(x[i]="9",9,ite(x[i]="8",8, …ite(x[i]="0",0,-1)…)`
  - ⇒ `… ite(48≤code(x[i])≤57,code(x[i])-48, -1)`

- Conversion between lowercase and uppercase strings `to_lower(x)`:
  - ⊗ `… ite(x[i]="A","a",ite(x[i]="B","b", … ite(x[i]="Z","z",x[i])…)`
  - ⇒ `… code(x[i])+ite(65≤code(x[i])≤90,32,0)`

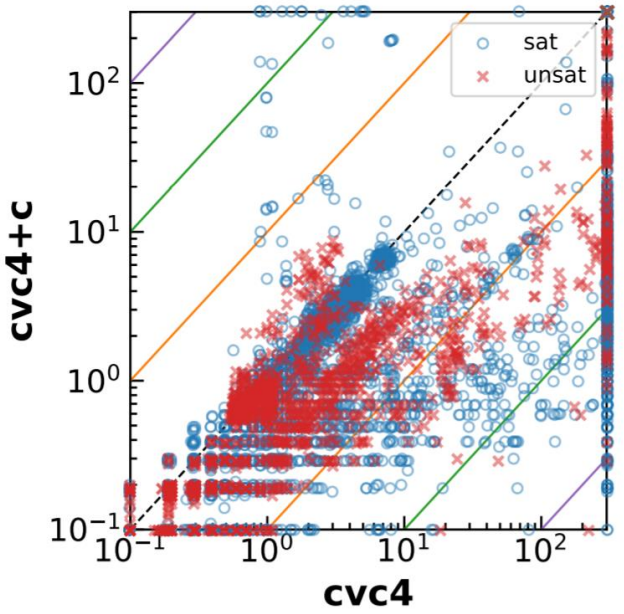
Reductions: Ordering, RegExp Range

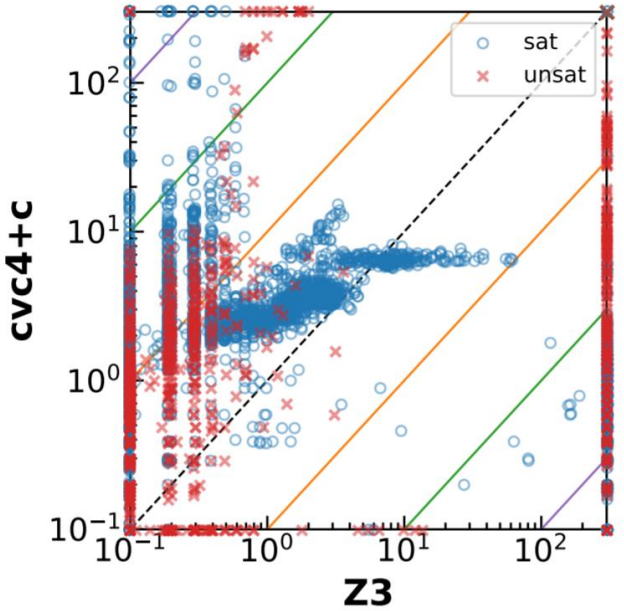
- Lexicographic ordering:
  - ⊗ $x \le y \Leftrightarrow \exists i \ldots (x[i]=y[i] \vee (x[i]=\text{“A”} \wedge y[i]=\text{“B”}) \vee (x[i]=\text{“A”} \wedge y[i]=\text{“C”}) \ldots)$
  - ⇒ $x \le y \Leftrightarrow \exists i \ldots \mathbf{code}(x[i]) \le \mathbf{code}(y[i])$

- Regular expression ranges:
  - ⊗ $x \in \text{range}(c_1, c_2) \Leftrightarrow \text{len}(x)=1 \wedge (x=c_1 \vee \ldots \vee x=c_2)$
  - ⇒ $x \in \text{range}(c_1, c_2) \Leftrightarrow \mathbf{code}(c_1) \le \mathbf{code}(x) \le \mathbf{code}(c_2)$

Experimental Results

Benchmark Set		cvc4+c	cvc4	z3
py-conbyte_cvc4	sat	**1344**	1104	1187
	unsat	**8576**	8547	8482
	×	13	282	264
py-conbyte_trauc	sat	**1009**	929	697
	unsat	1424	1407	**1428**
	×	13	110	321
py-conbyte_z3seq	sat	**1354**	1126	1343
	unsat	**5864**	5797	5719
	×	35	330	191
py-conbyte_z3str	sat	**711**	652	692
	unsat	**1227**	1223	1223
	×	3	66	26
Total	sat	**4418**	3811	3919
	unsat	**17091**	16974	16852
	×	64	788	802

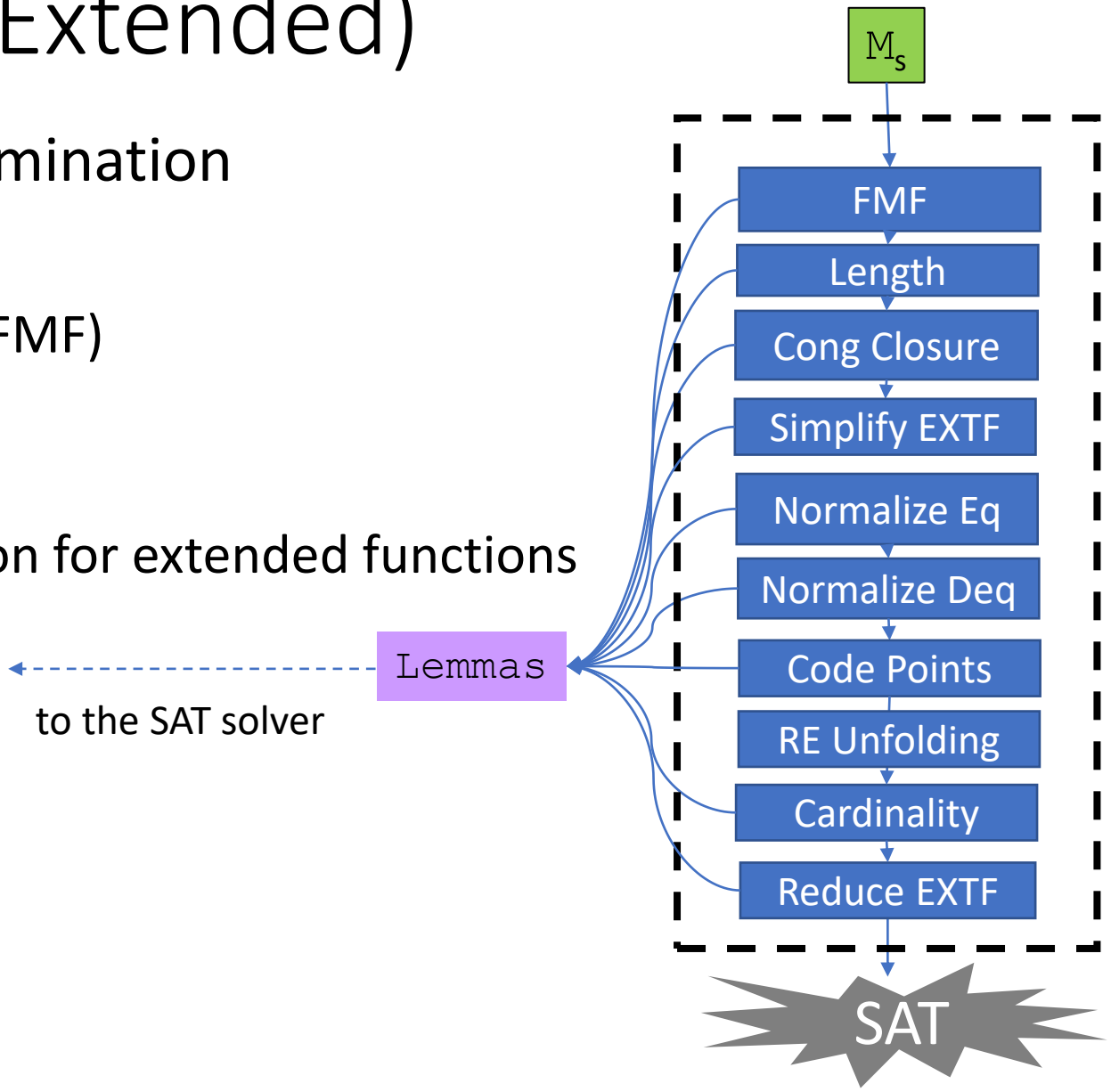




- 10x t/o reduction
- Faster runtimes
- Improvement wrt state-of-the-art

String Theory Solver (Extended)

- Preprocess based on reg-exp elimination
- Then, run inference strategy:
  1. Split on sum of lengths bound (FMF)
  2. Elaborate length constraints
  3. Congruence closure
  4. Context-dependent simplification for extended functions
  5. Normalize string equalities
  6. Normalize string disequalities
  7. Subprocedure for code points
  8. Regular expression unfolding
  9. Check cardinality constraints
  10. Reduce extended functions



Conclusions

- SMT solvers are:
  - Efficient (incomplete) procedure for word equations with length
  - FMF, context-dependent simplification, RE elimination, witness sharing, …
- Future work in cvc5:
  - Proofs
  - Array-like reasoning (update + extractions)
  - Eager context-dependent simplification

- cvc5 is open-source, available at https://cvc5.github.io/
  - Also supports theory of sequences, further extensions

- Thanks for listening!