Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification

Andrew Reynolds, Maverick Woo, Clark Barrett,

David Brumley, Tianyi Liang, Cesare Tinelli

CAV 2017

Importance of String Solvers

- Automated string solvers are essential for formal methods applications
- Security applications (e.g. finding XSS attacks) require *string solvers* that:
  - Are *highly efficient*
  - Reason about strings with *unbounded length* (not just bounded ones)
  - Accept a *rich language* of string constraints

In This Paper:

- DPLL(T) string solvers for *extended string constraints*
- New technique, *context-dependent simplification*, improves *scalability* of current string solvers
- Implemented in SMT solver *CVC4*
- *Experiments* show advantages using CVC4 as backend to *symbolic execution* engine PyEx

Basic String Constraints

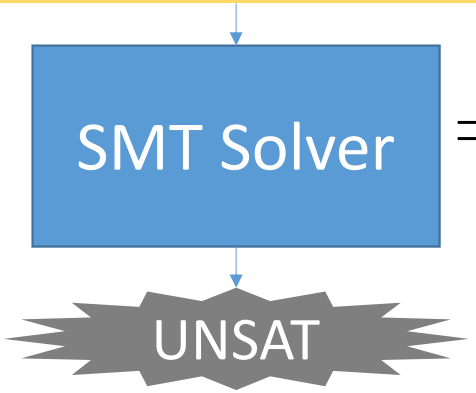
- Equalities and disequalities between:
  - *Basic string* terms
    - String constants: $\in$, “abc”
    - Concatenation: $x \cdot$“abc”
    - Length: $|\ x\ |$
  - *Linear arithmetic* terms: $x+4$, $y>2$

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

Basic String Constraints

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

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



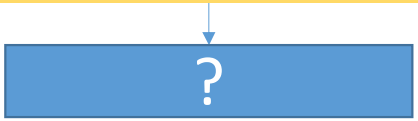
$\Rightarrow$ *Procedure for basic constraints in CVC4*

[Liang et al CAV2014]

Extended String Constraints

- Equalities and disequalities between:
  - *Basic string* terms
    - String constants: $\in$, “abc”
    - Concatenation: x·“abc”
    - Length: | x |
  - *Linear arithmetic* terms: x+4, y>2
  - *Extended* string terms:
    - Substring: substr(“abcde”,1,3)
    - String contains: contains(“abcde”,”cd”)
    - Find “index of”: indexof(“abcde”,”d”,0)
    - String replace: replace(x,“a”,“b”)

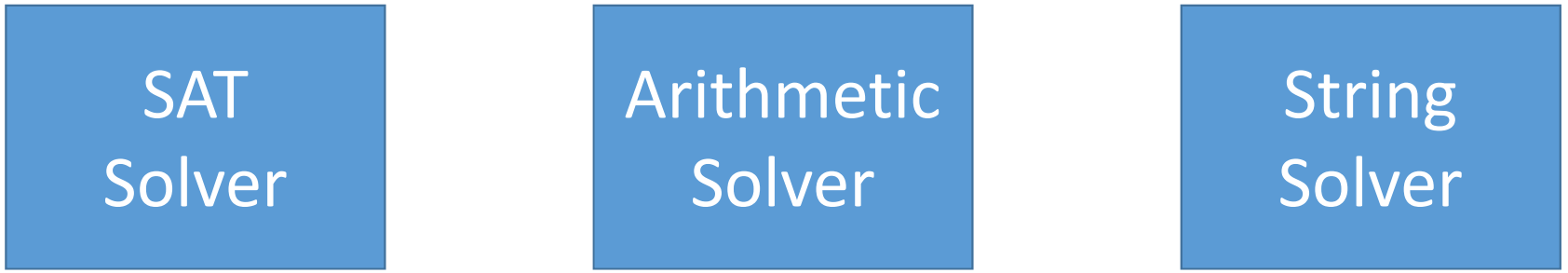
Example: $\dot{E}$contains(substr(x,0,3),“a”) $\wedge$ 0$\leq$indexof(x,“ab”,0)<4



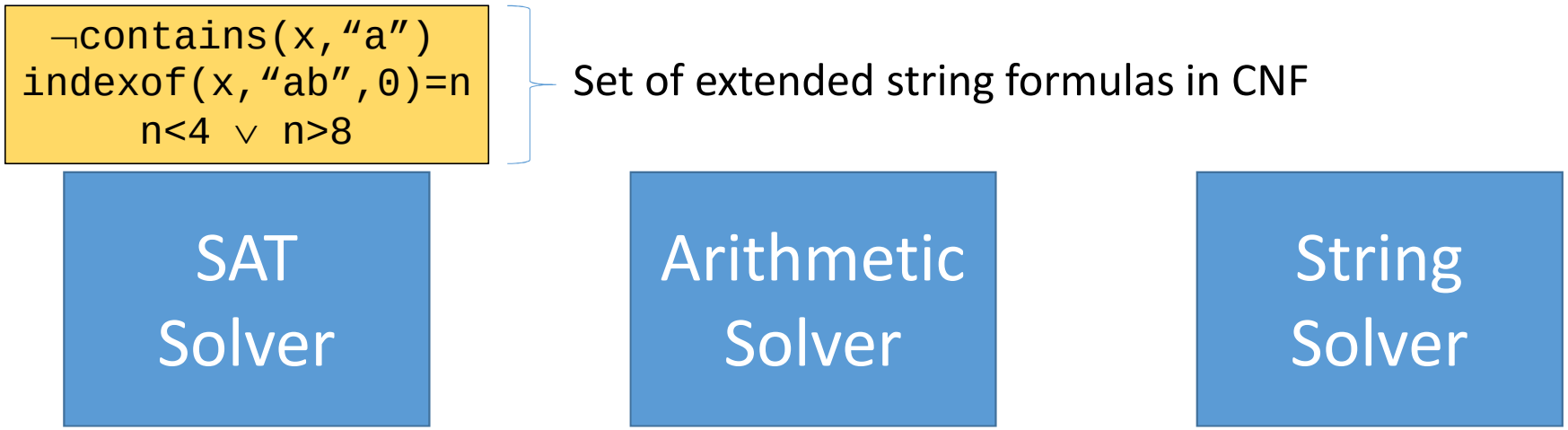
$\Rightarrow$ *Focus of this work*

DPLL(T) String Solvers

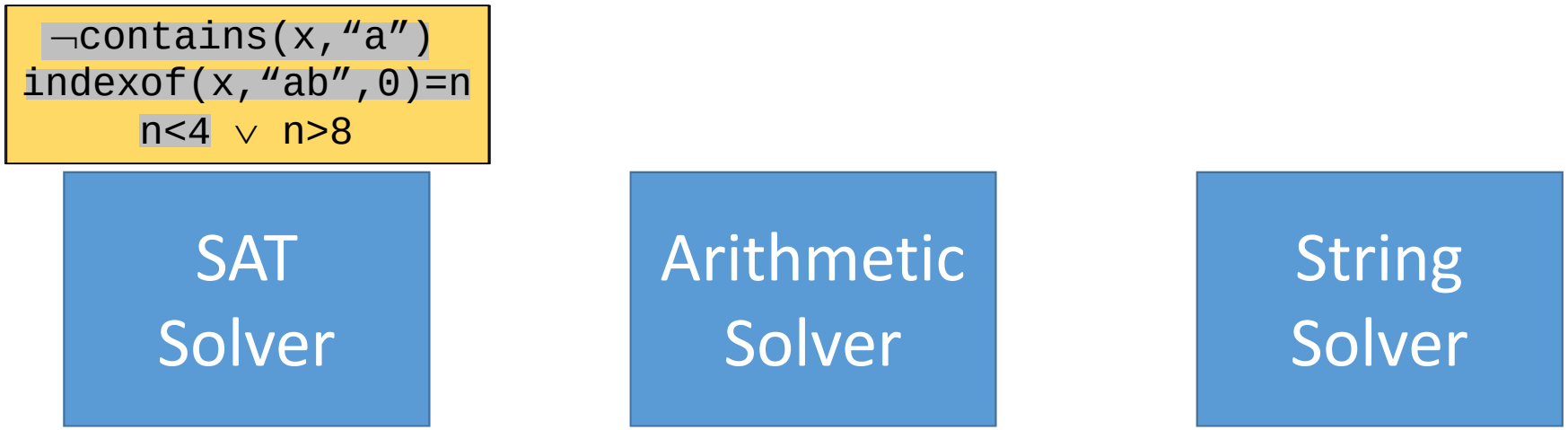
- Cooperation between:



DPLL(T) String Solvers

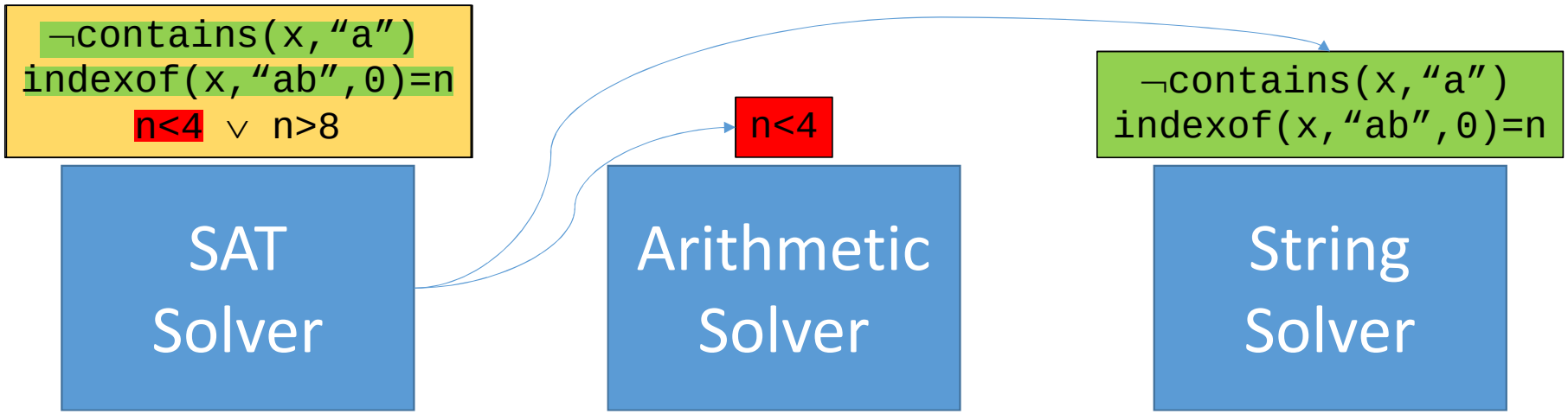


DPLL(T) String Solvers



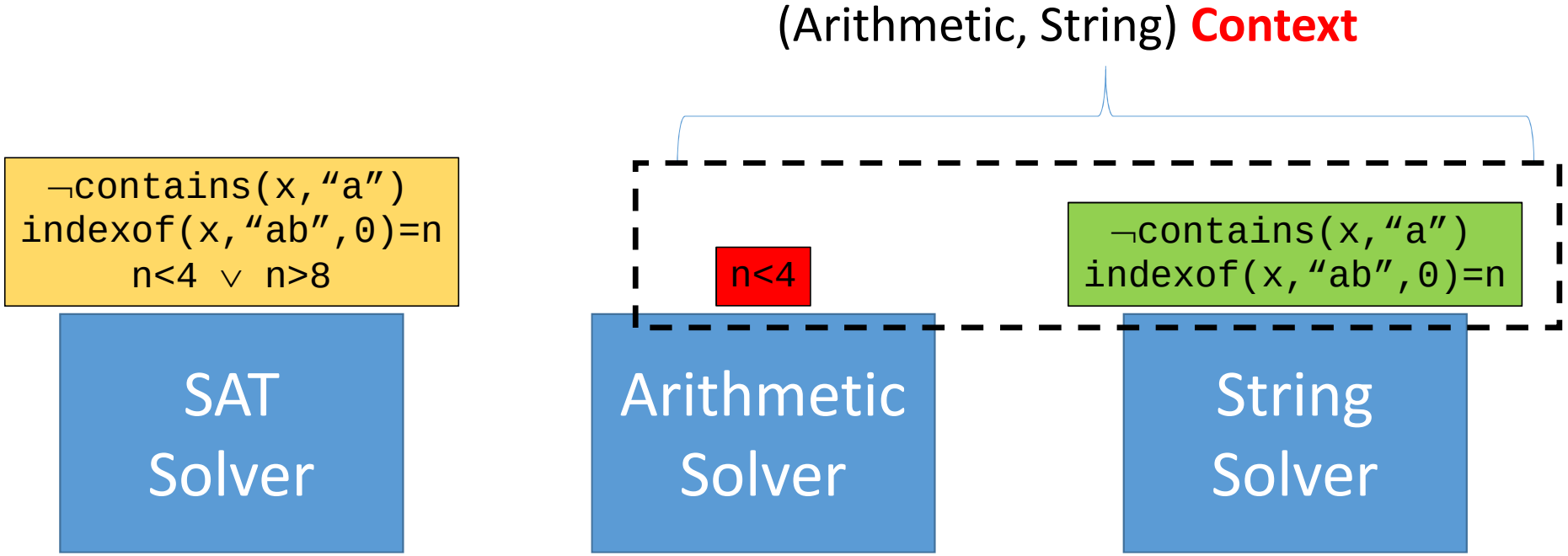
⇒ Find propositionally satisfying assignment

DPLL(T) String Solvers



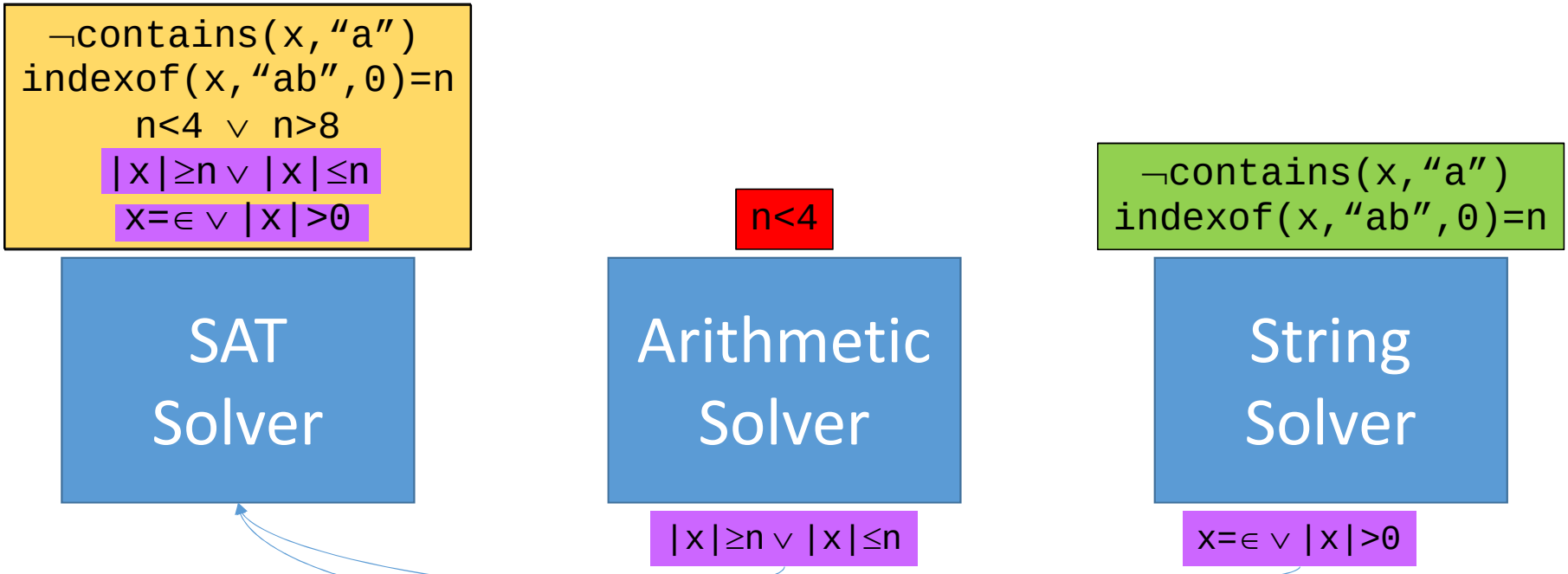
⇒ Distribute to arithmetic and string solvers

DPLL(T) String Solvers



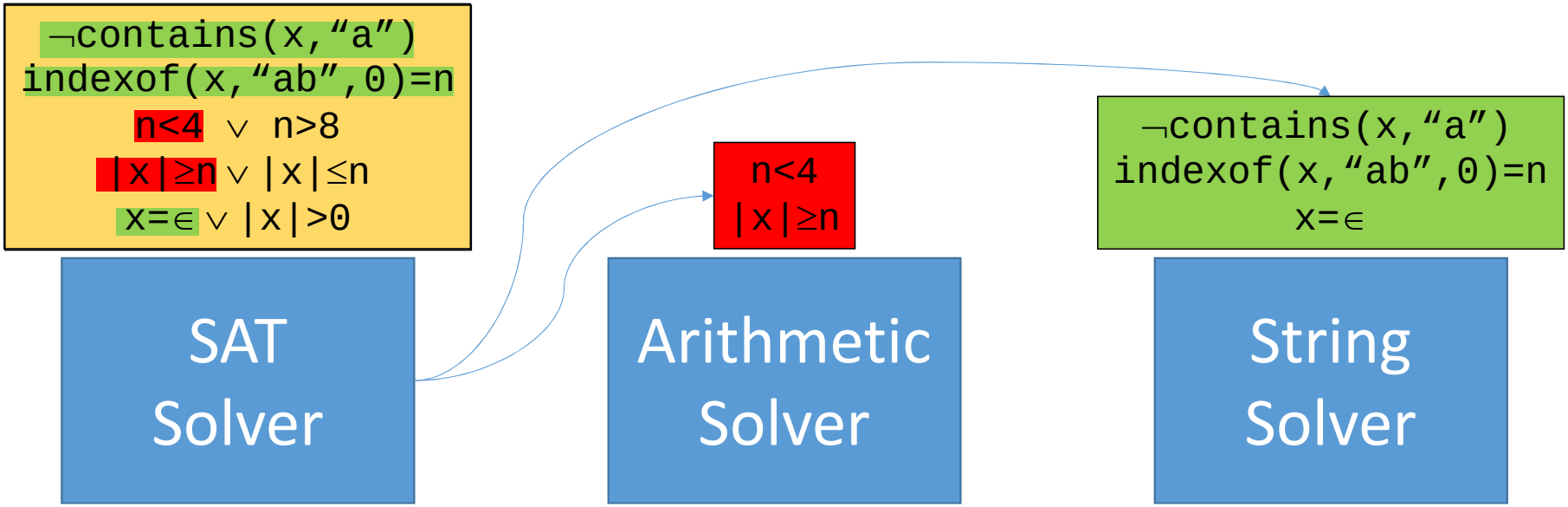
⇒ Solvers maintain a **context** (conjunction of theory literals)

DPLL(T) String Solvers



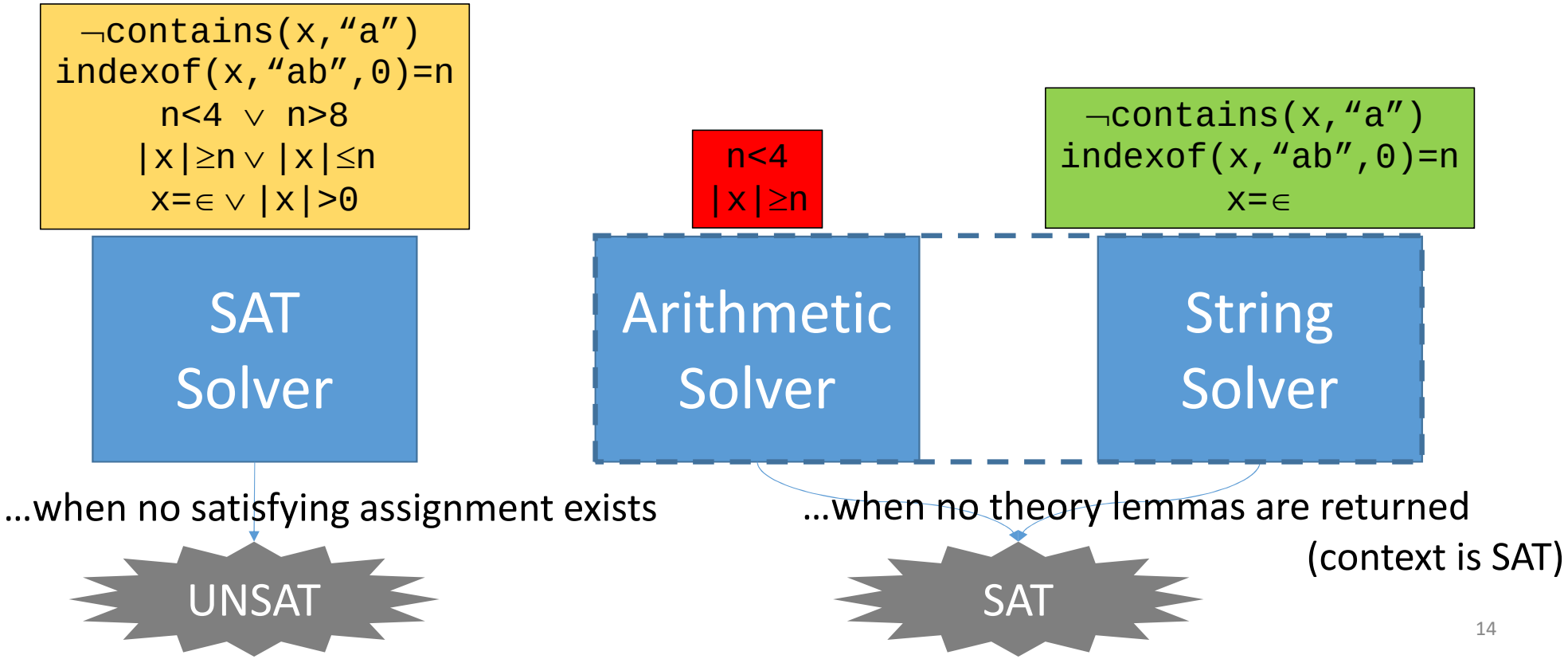
⇒ String and arithmetic solvers return theory lemmas to SAT solver

DPLL(T) String Solvers



⇒ …and repeat

DPLL(T) String Solvers



Properties of DPLL(T) String Solvers

- For *basic* constraints, DPLL(T) string solvers:
  - Can be used for "**sat**" and "**unsat**" answers
  - Are **incomplete** and/or **non-terminating** in general
- Expected, since *decidability is unknown*
  [Bjorner et al 2009,Ganesh et al 2011]
- Regardless, modern solvers are *efficient in practice*
  [Zheng et al 2013,Liang et al 2014,Abdulla et al 2015,Trinh et al 2016]

How do we handle Extended String Constraints?

`¬contains(x,“a”)`

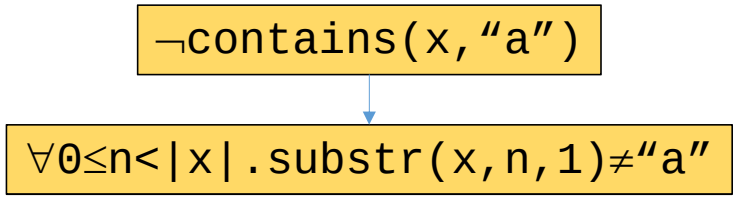
How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀

`¬contains(x,“a”)`

How do we handle Extended String Constraints?

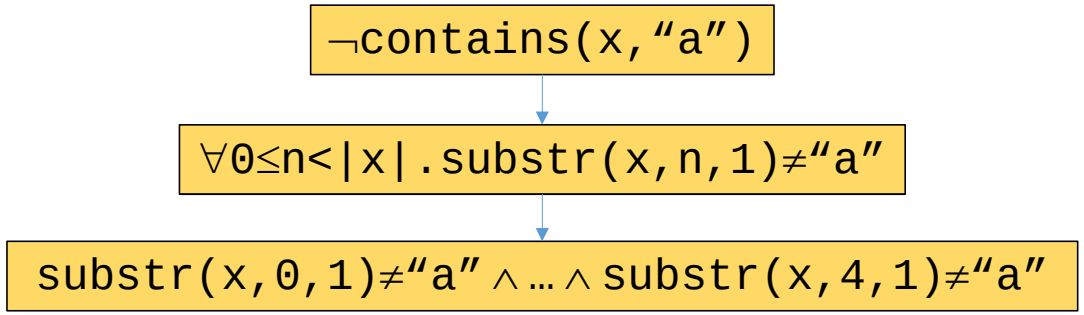
- Naively, by reduction to basic constraints + bounded ∀



Expand `contains`

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀

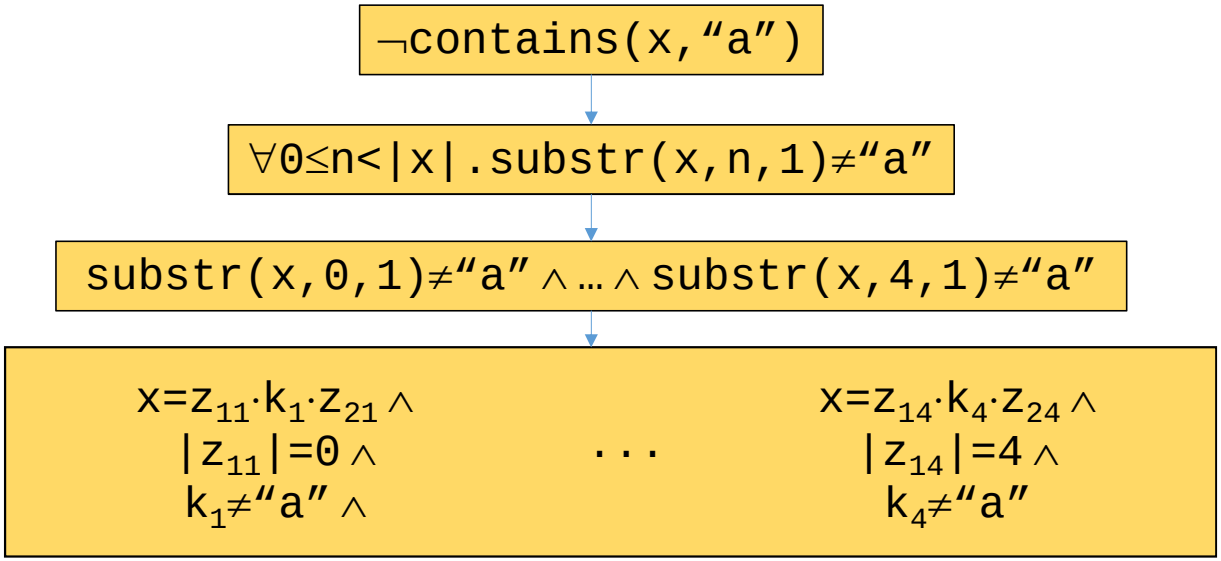


Expand `contains`

Assuming bound $|x| \leq 5$

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀



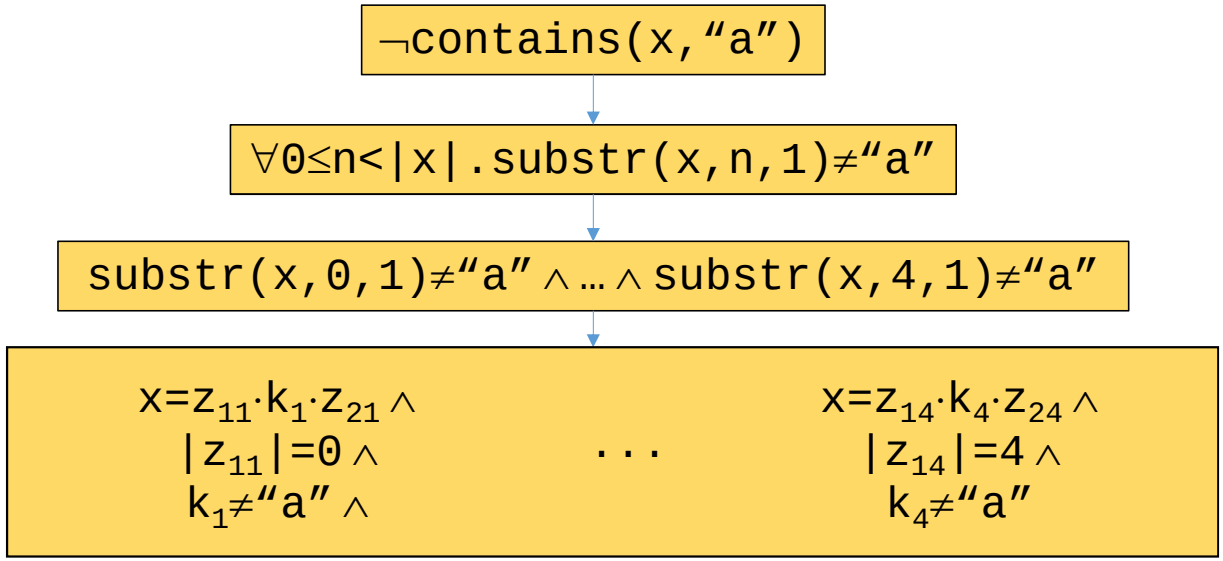
Expand `contains`

Assuming bound $|x| \leq 5$

Expand `substr`

How do we handle Extended String Constraints?

- Naively, by reduction to basic constraints + bounded ∀



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

(Eager) Expansion of Extended Constraints

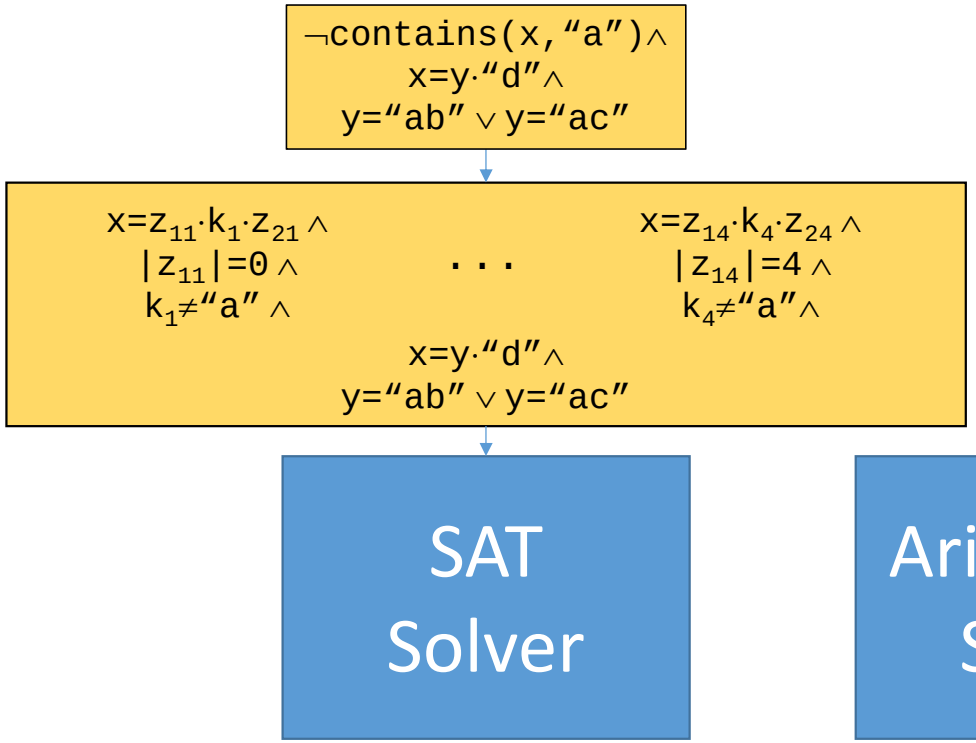
$\neg$contains(x,“a”)$\wedge$
x=y$\cdot$“d”$\wedge$
y=“ab”$\vee$y=“ac”

SAT Solver

Arithmetic Solver

String Solver

(Eager) Expansion of Extended Constraints



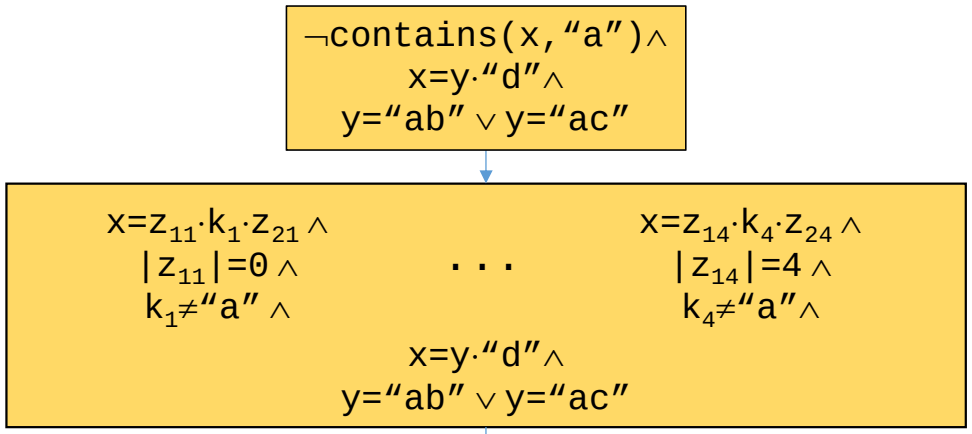
Expand and eliminate extended symbols

SAT Solver

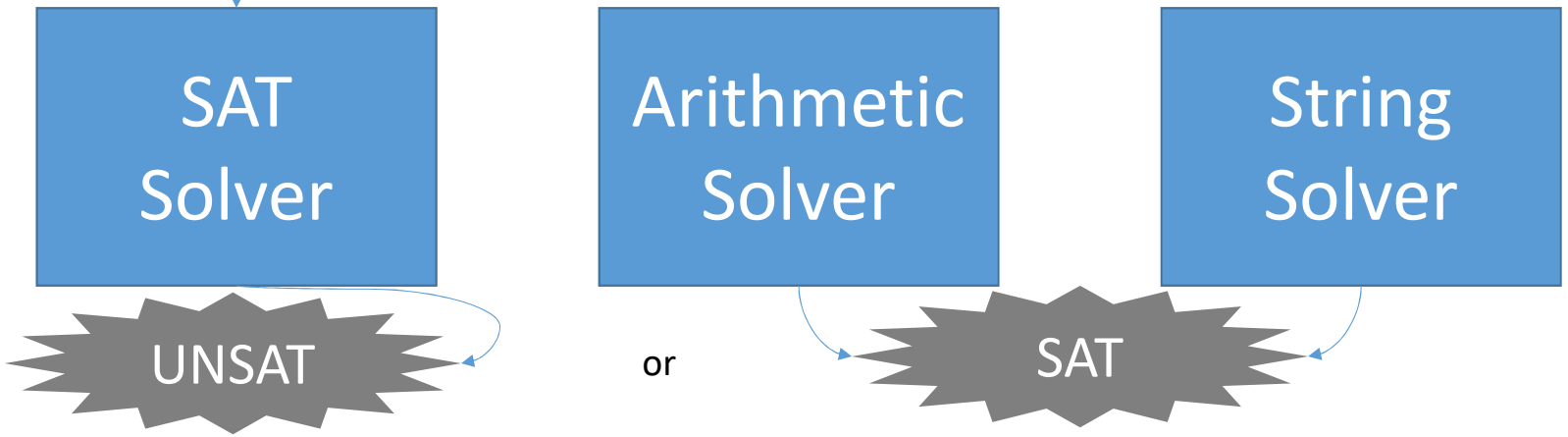
Arithmetic Solver

String Solver

(Eager) Expansion of Extended Constraints



Expand and eliminate extended symbols



(Lazy) Expansion of Extended Constraints

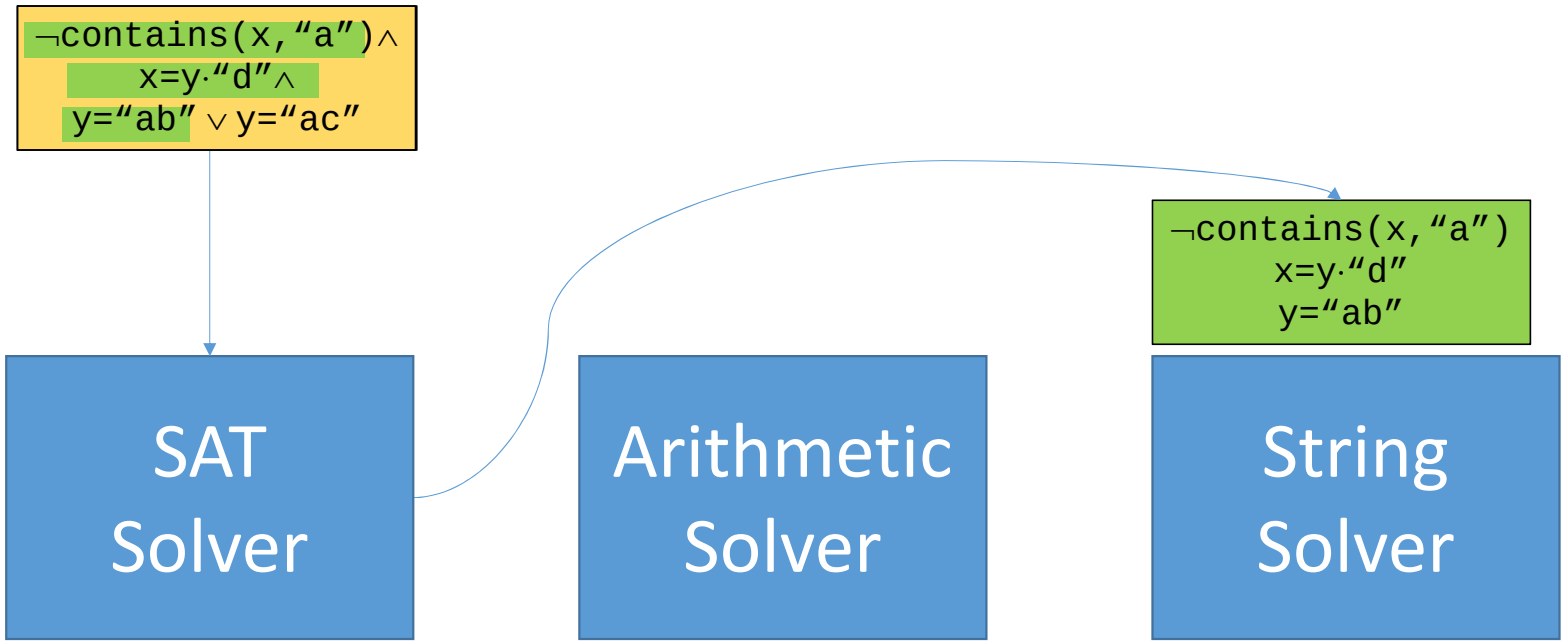
$\neg$contains(x,“a”)$\wedge$
x=y·“d”$\wedge$
y=“ab”$\vee$y=“ac”

SAT Solver

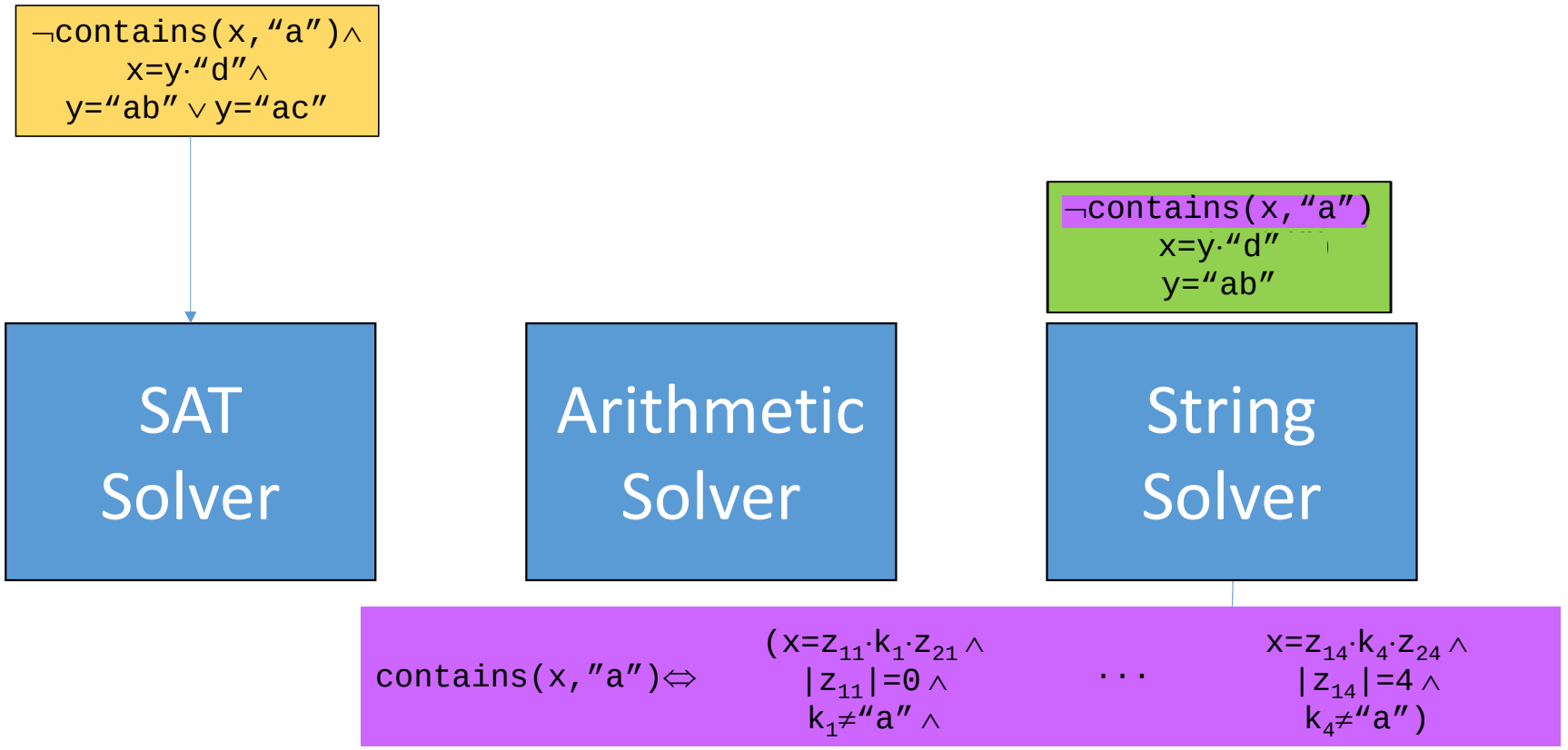
Arithmetic Solver

String Solver

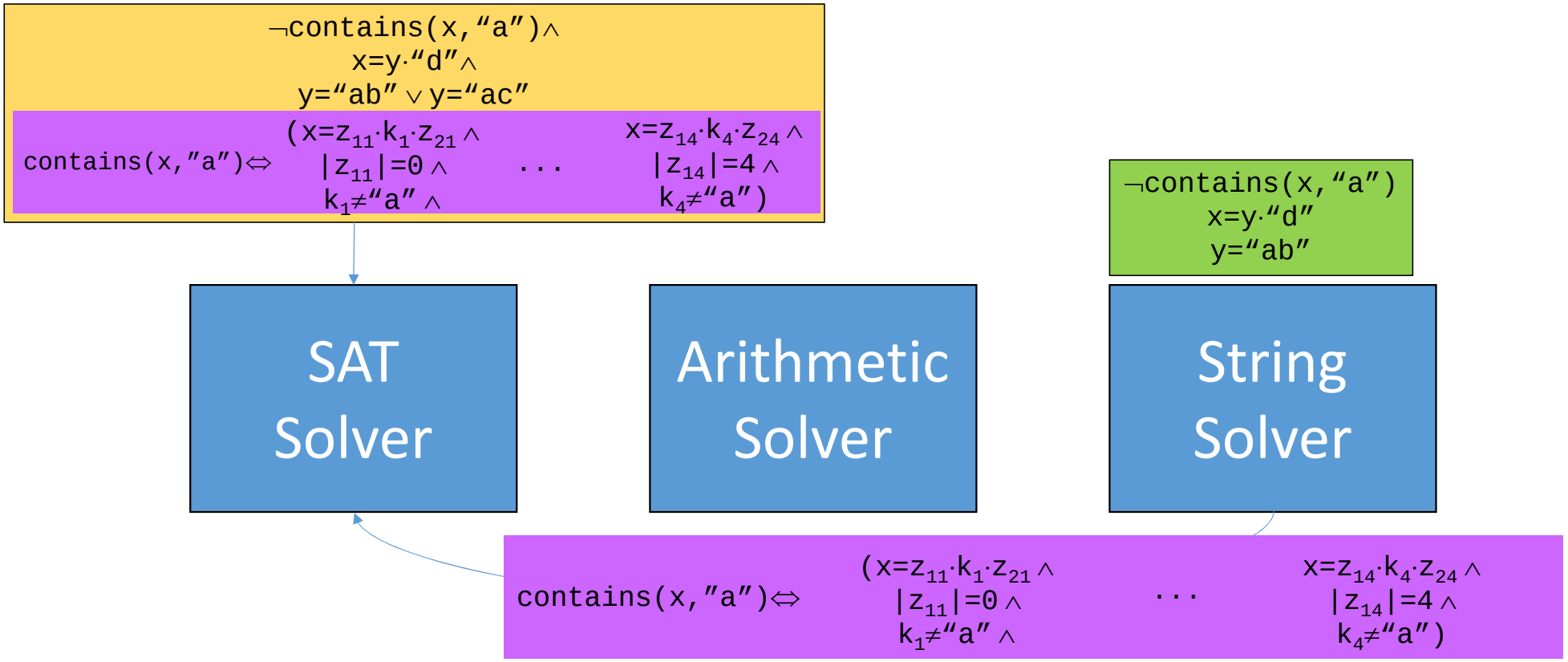
(Lazy) Expansion of Extended Constraints



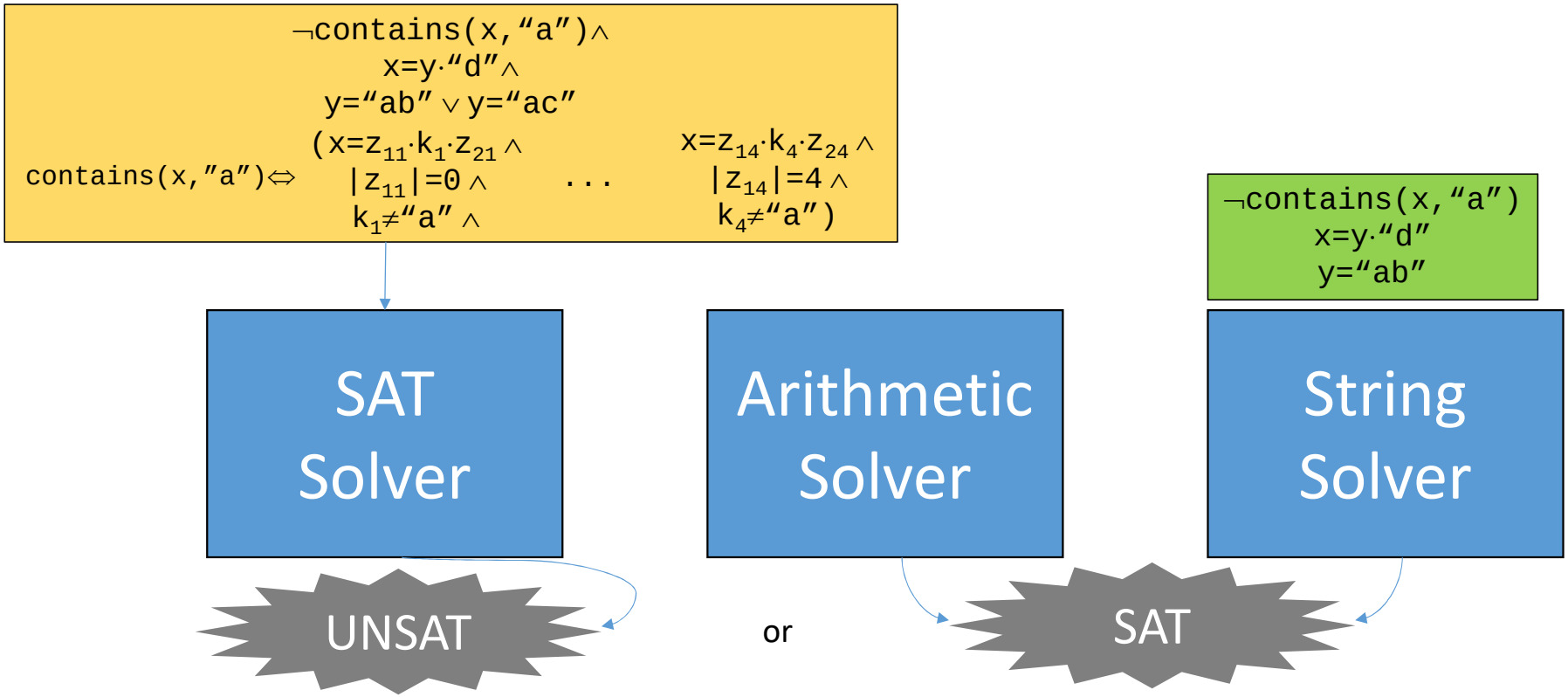
(Lazy) Expansion of Extended Constraints



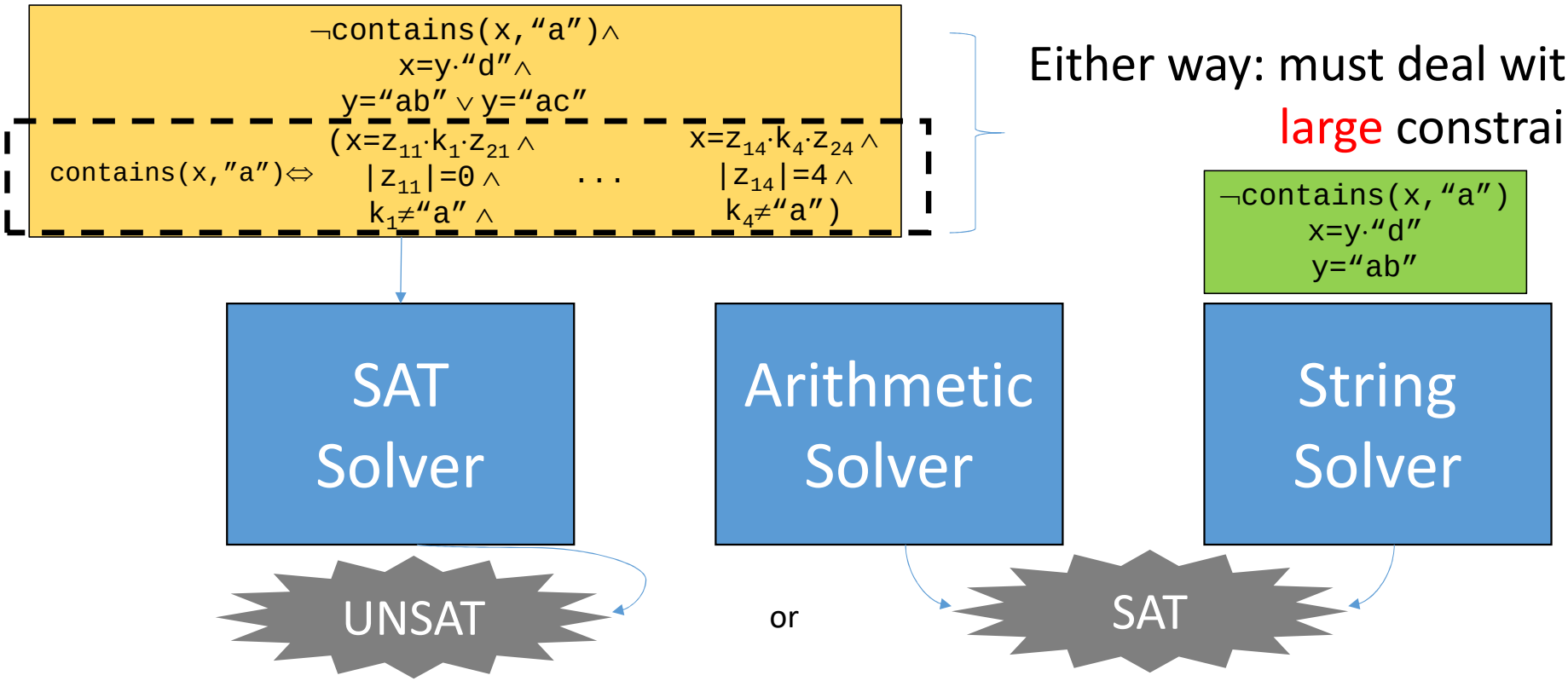
(Lazy) Expansion of Extended Constraints



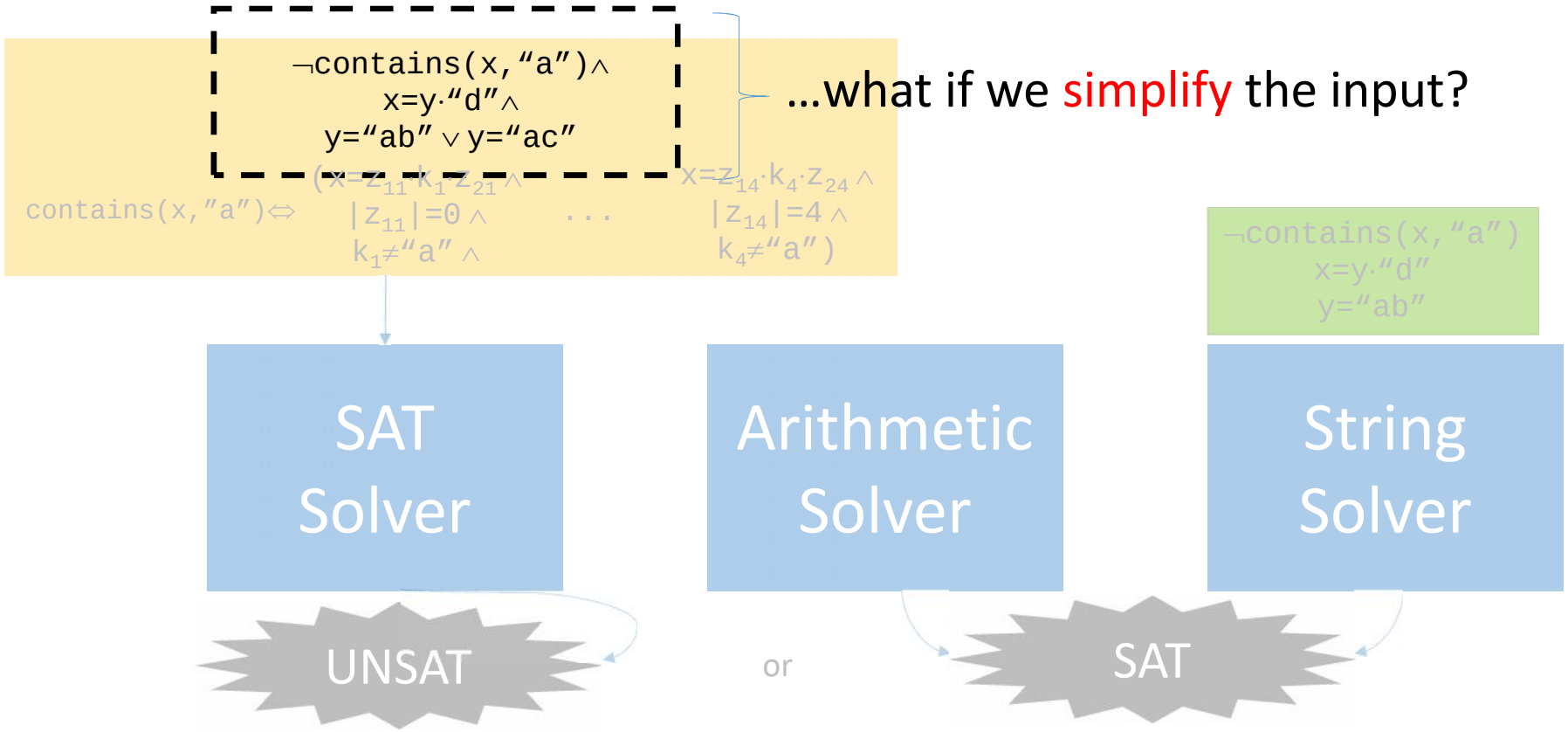
(Lazy) Expansion of Extended Constraints



(Lazy) Expansion of Extended Constraints



(Lazy) Expansion of Extended Constraints



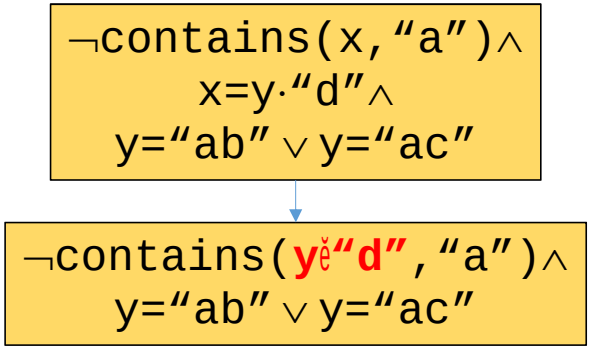
SMT Solvers + Simplification

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

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

SMT Solvers + Simplification

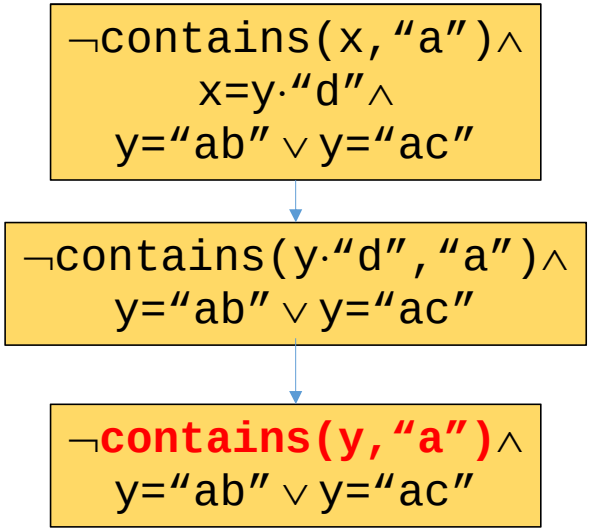
- All SMT solvers implement *simplification* techniques
  (also called *normalization* or *rewrite* rules)



since x=y·”d”

SMT Solvers + Simplification

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

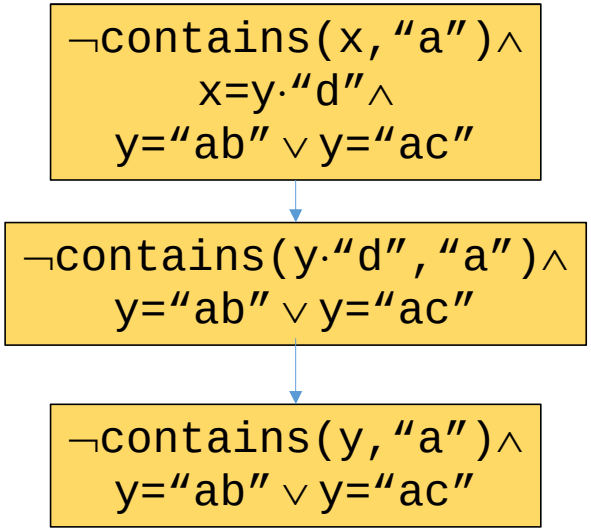


since x=y·”d”

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, simpler procedures

(Lazy) Expansion + Simplification

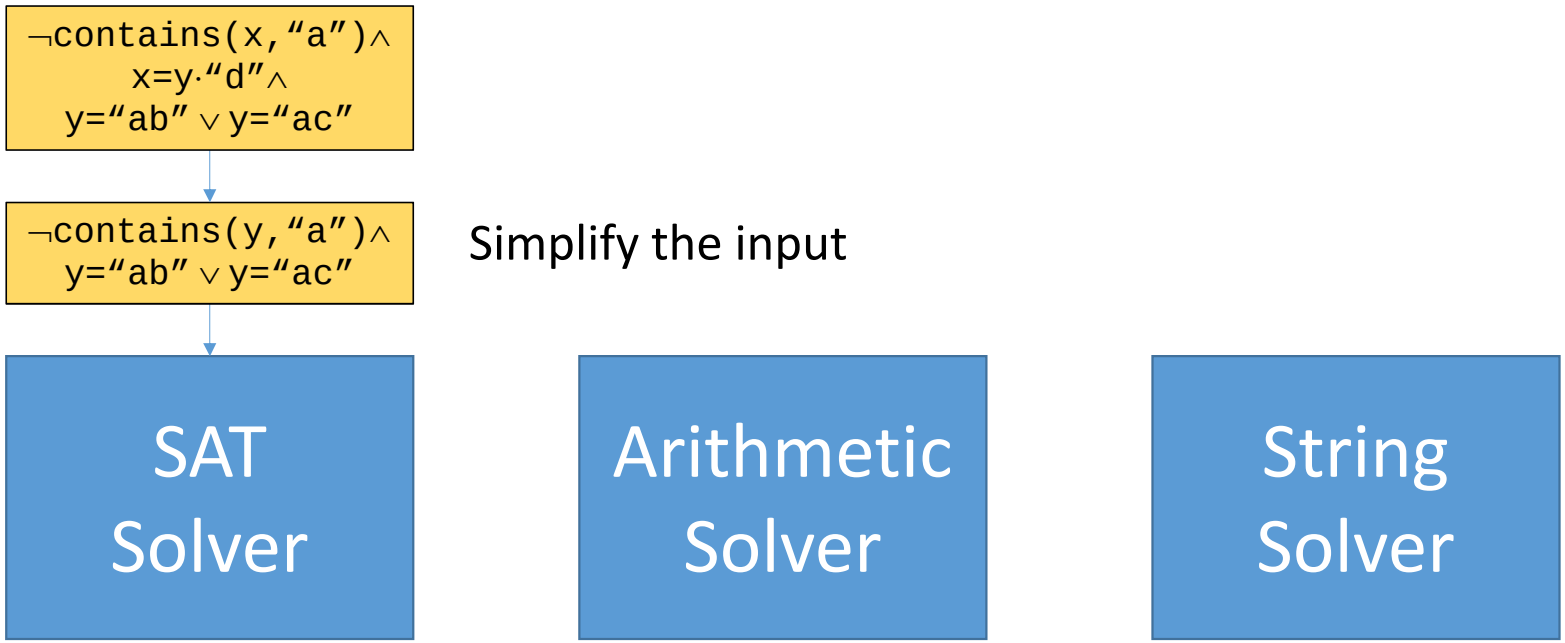
$\neg$contains(x,“a”)$\wedge$
x=y$\cdot$“d”$\wedge$
y=“ab”$\vee$y=“ac”

SAT Solver

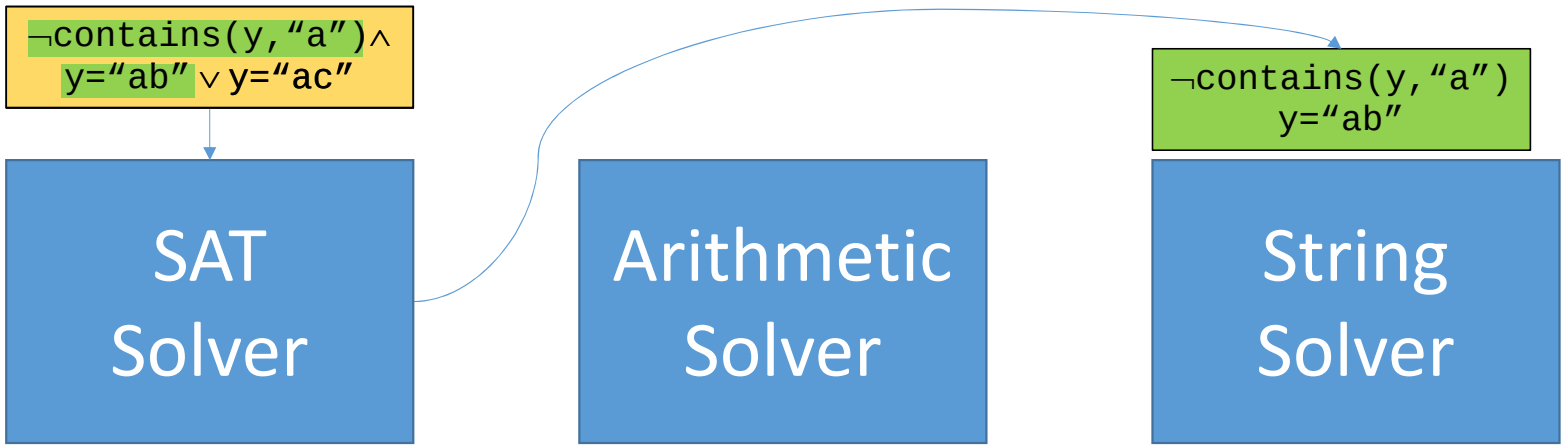
Arithmetic Solver

String Solver

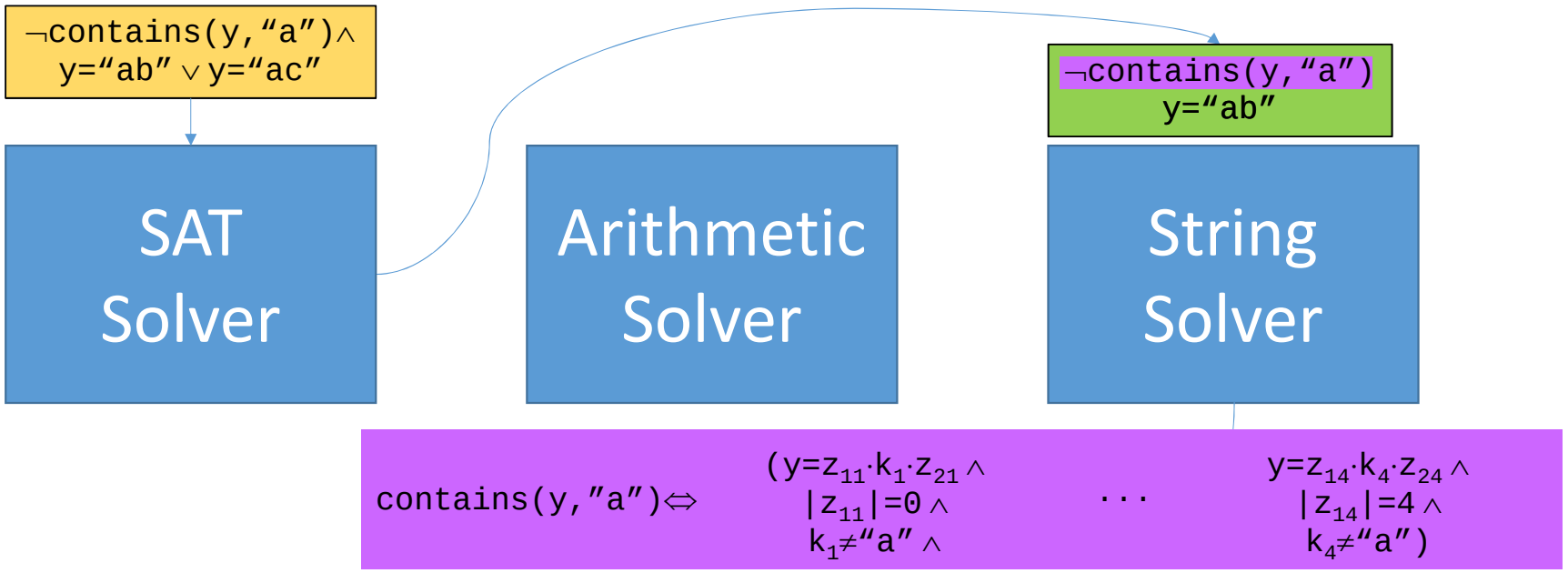
(Lazy) Expansion + Simplification



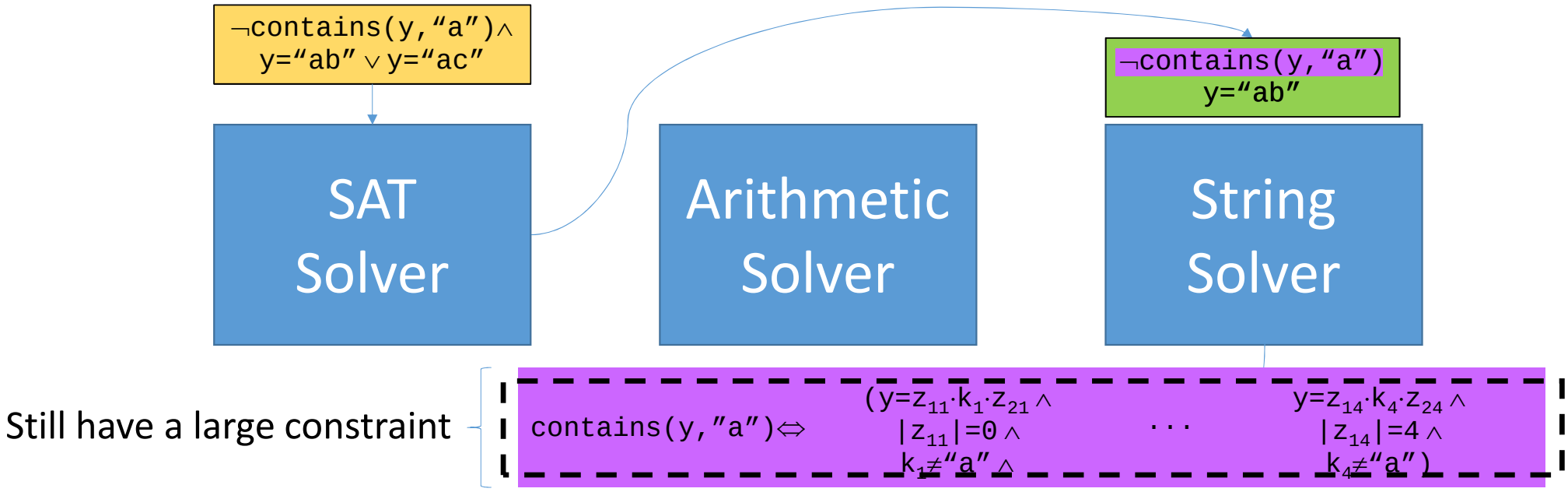
(Lazy) Expansion + Simplification



(Lazy) Expansion + Simplification

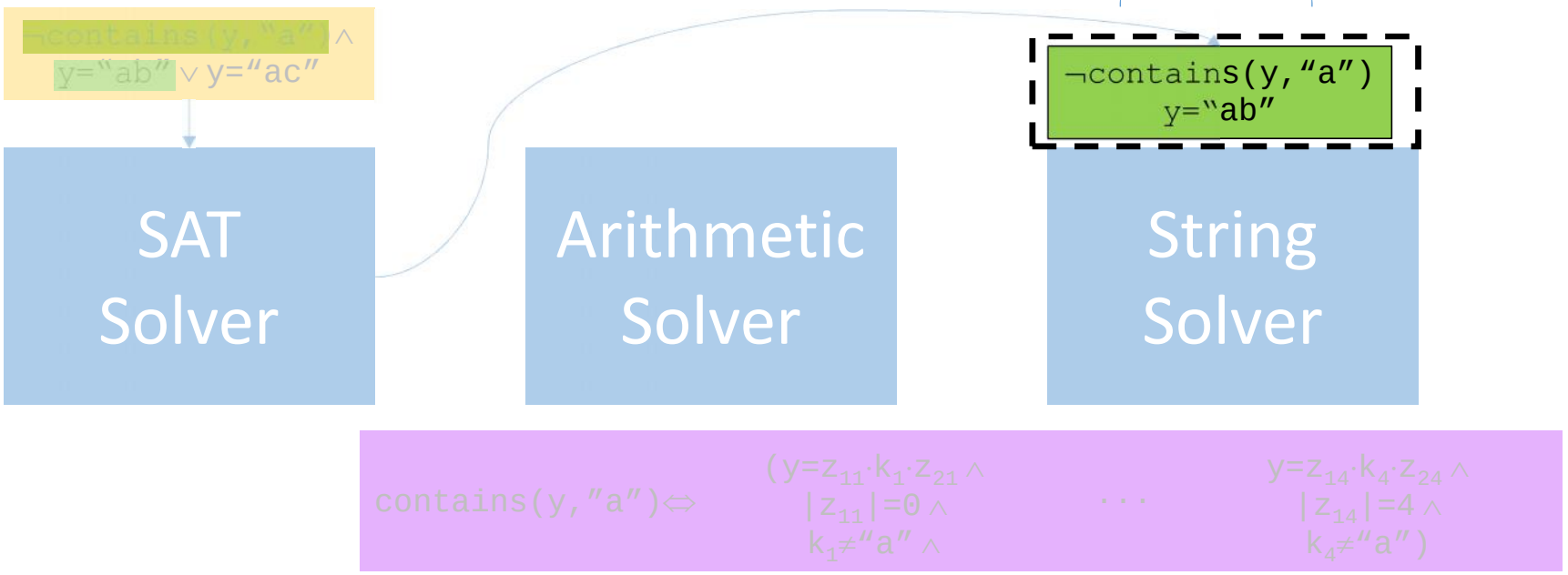


(Lazy) Expansion + Simplification



(Lazy) Expansion + Simplification

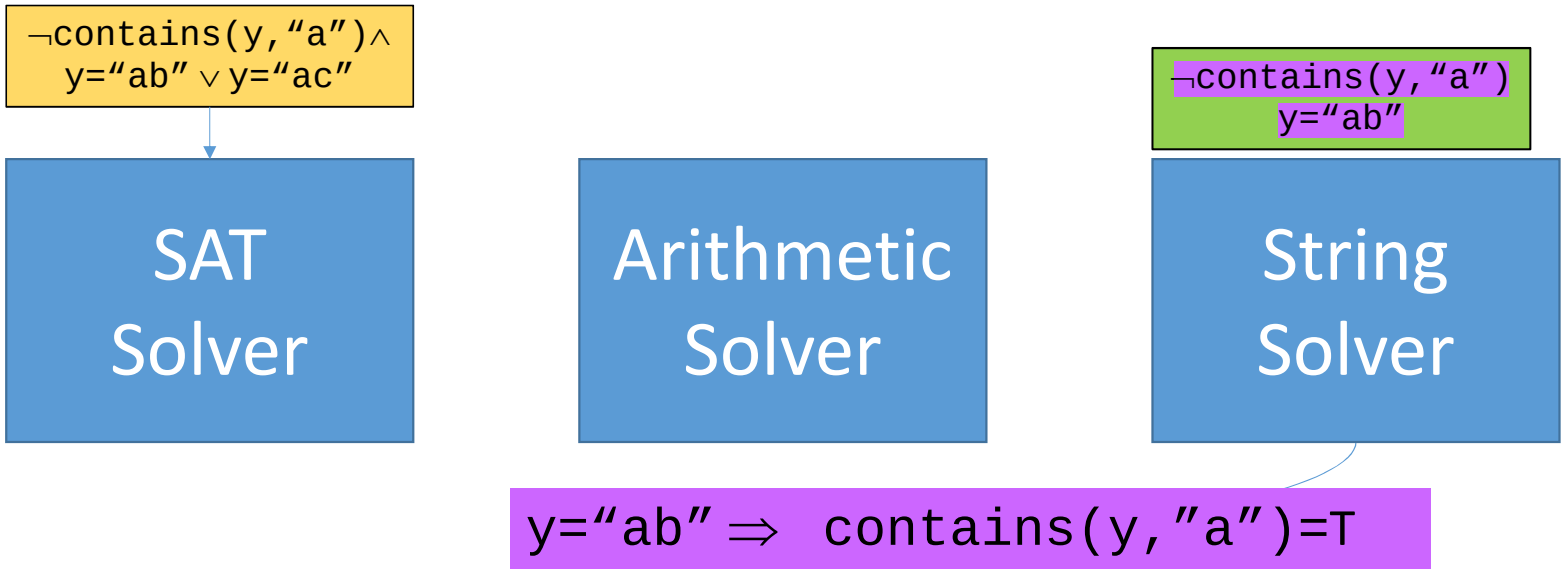
What if we simplify based on the context?



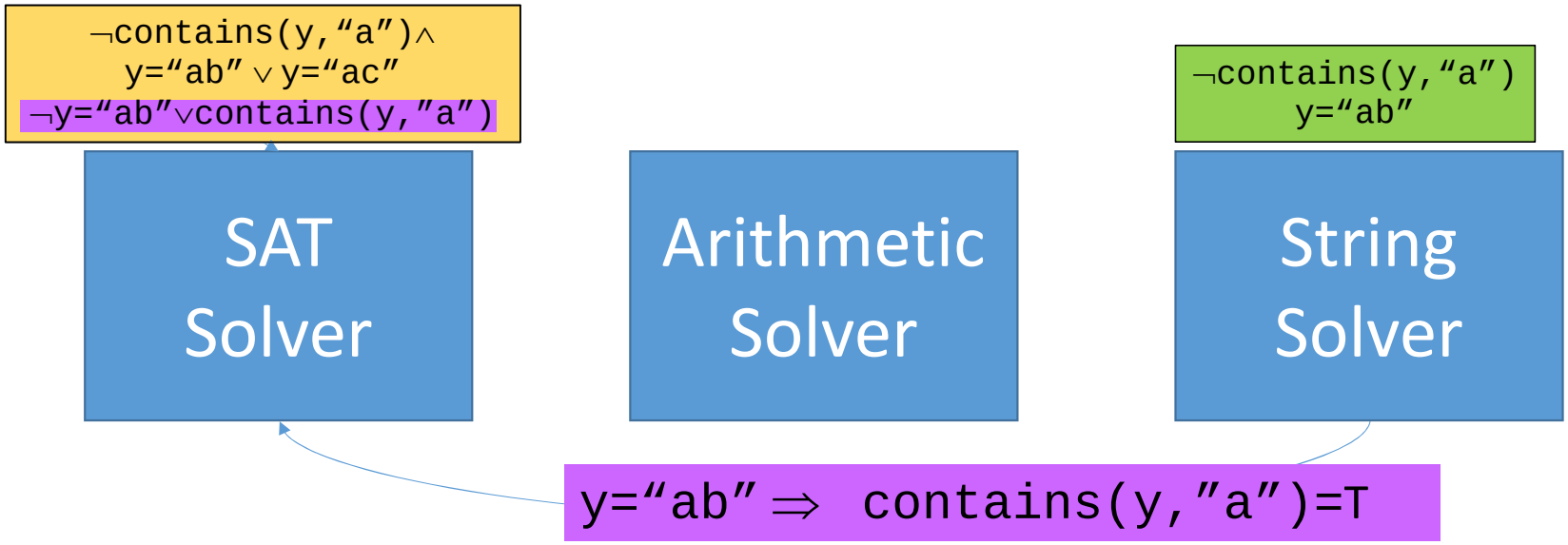
(Lazy) Expansion + Context-Dependent Simplification

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

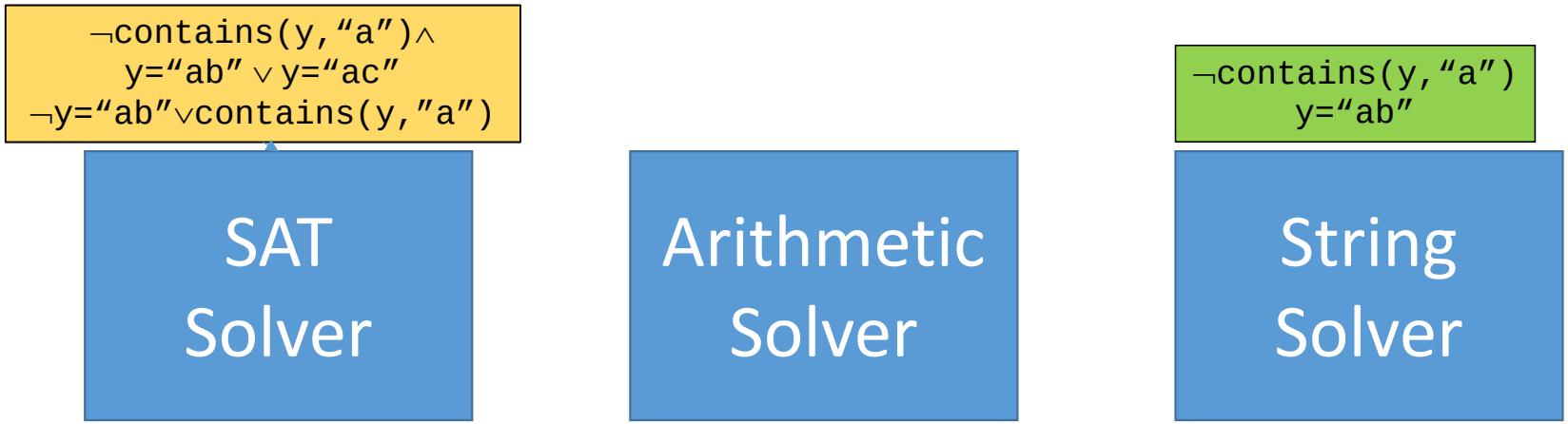
(Lazy) Expansion + Context-Dependent Simplification



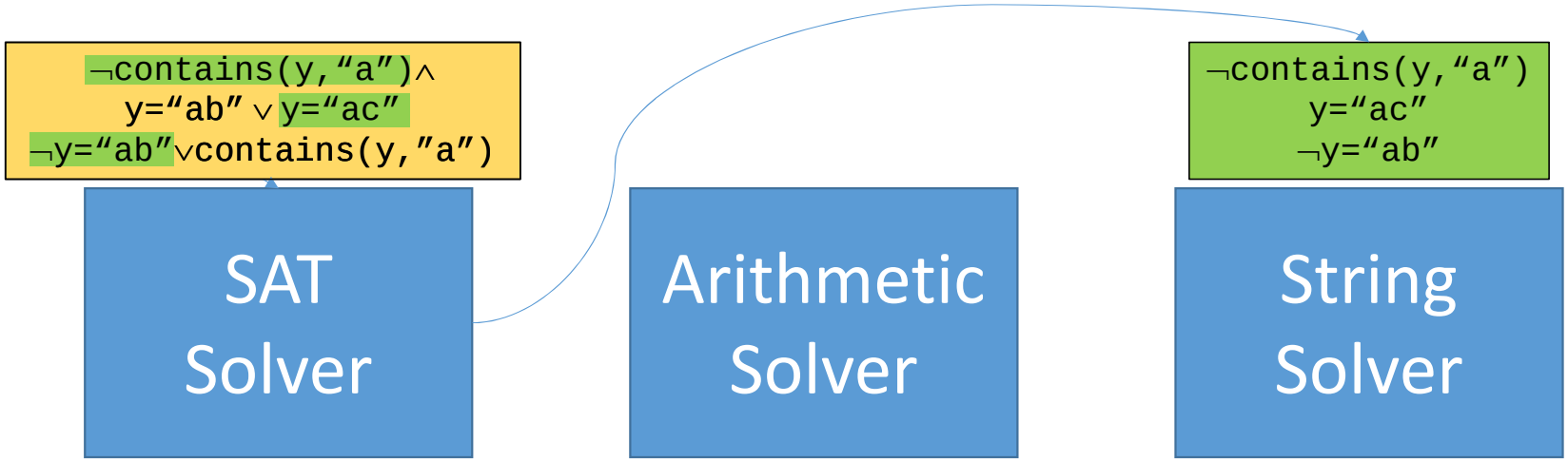
(Lazy) Expansion + Context-Dependent Simplification



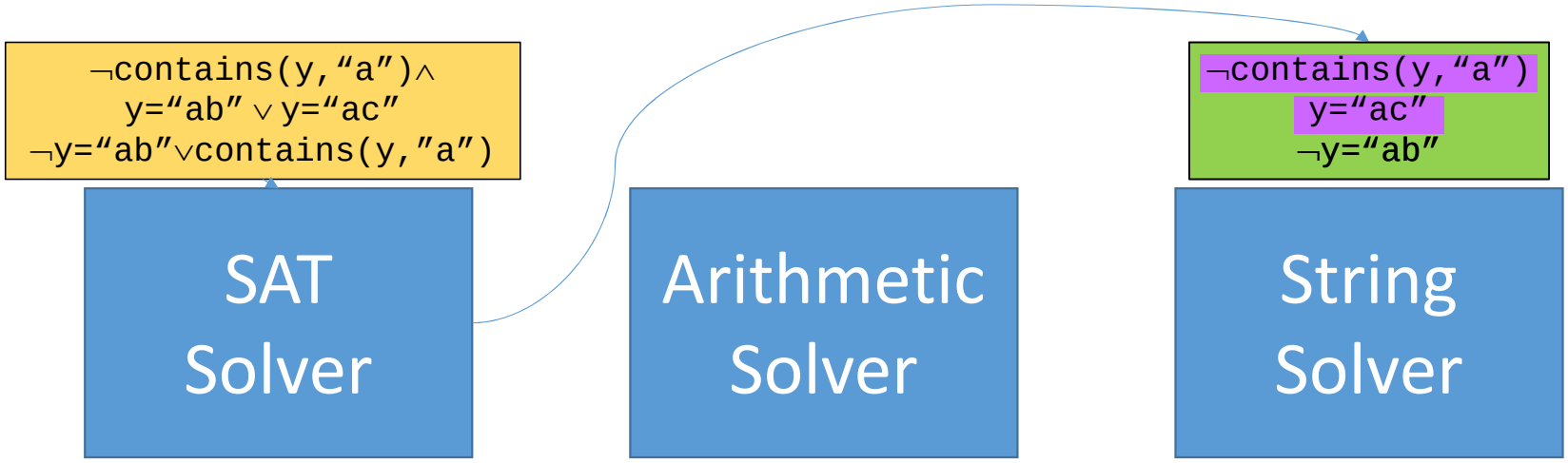
(Lazy) Expansion + Context-Dependent Simplification



(Lazy) Expansion + Context-Dependent Simplification

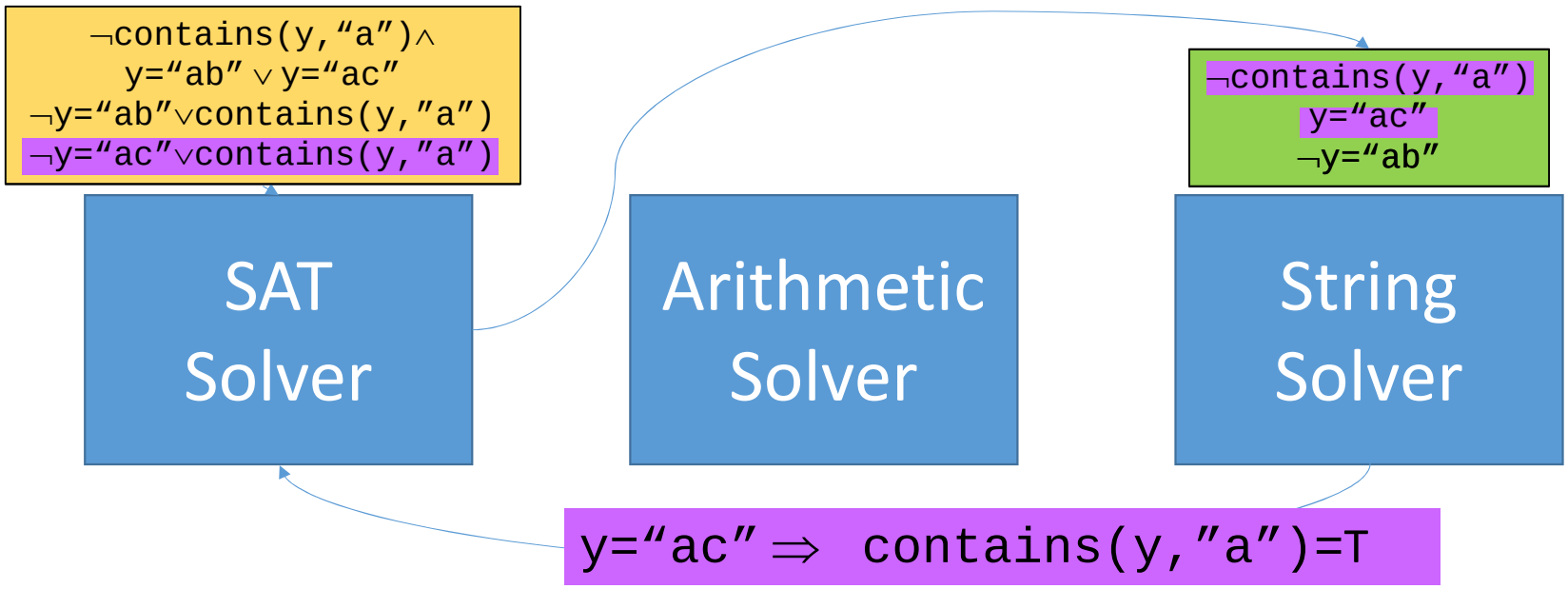


(Lazy) Expansion + Context-Dependent Simplification

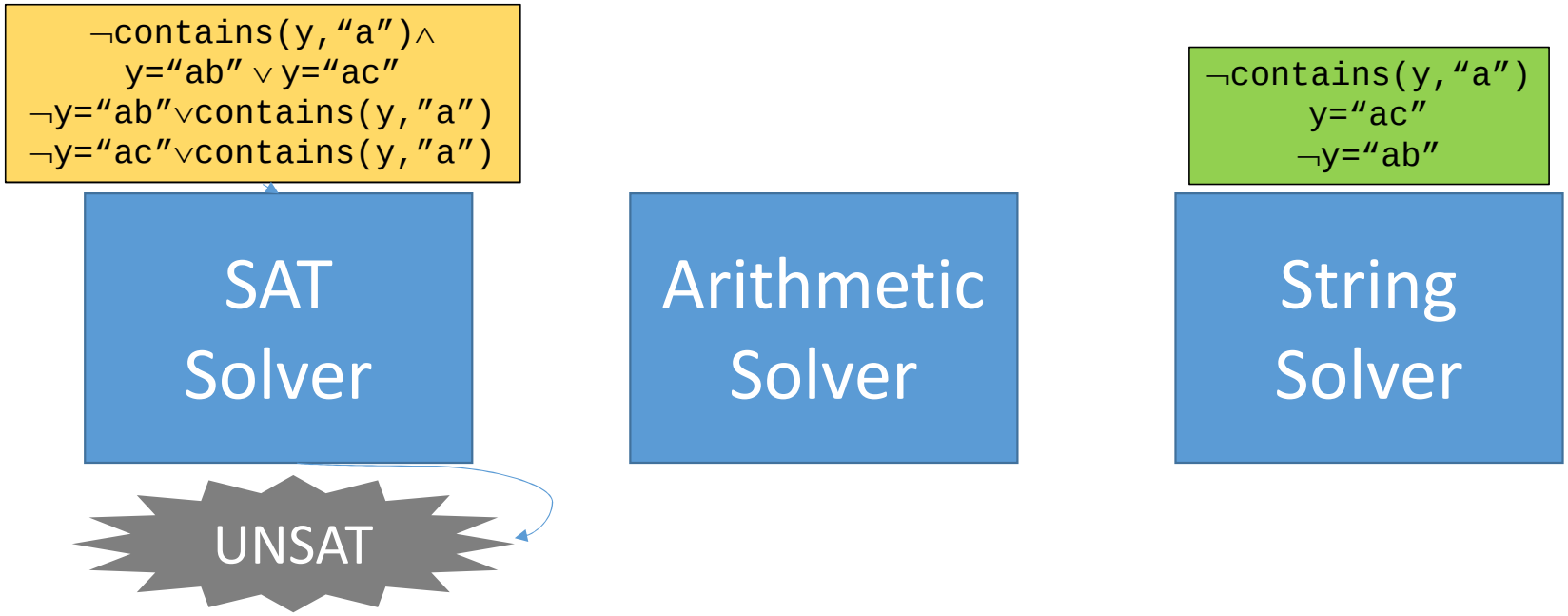


`contains(y,”a”)` is also true when `y=“ac”` …

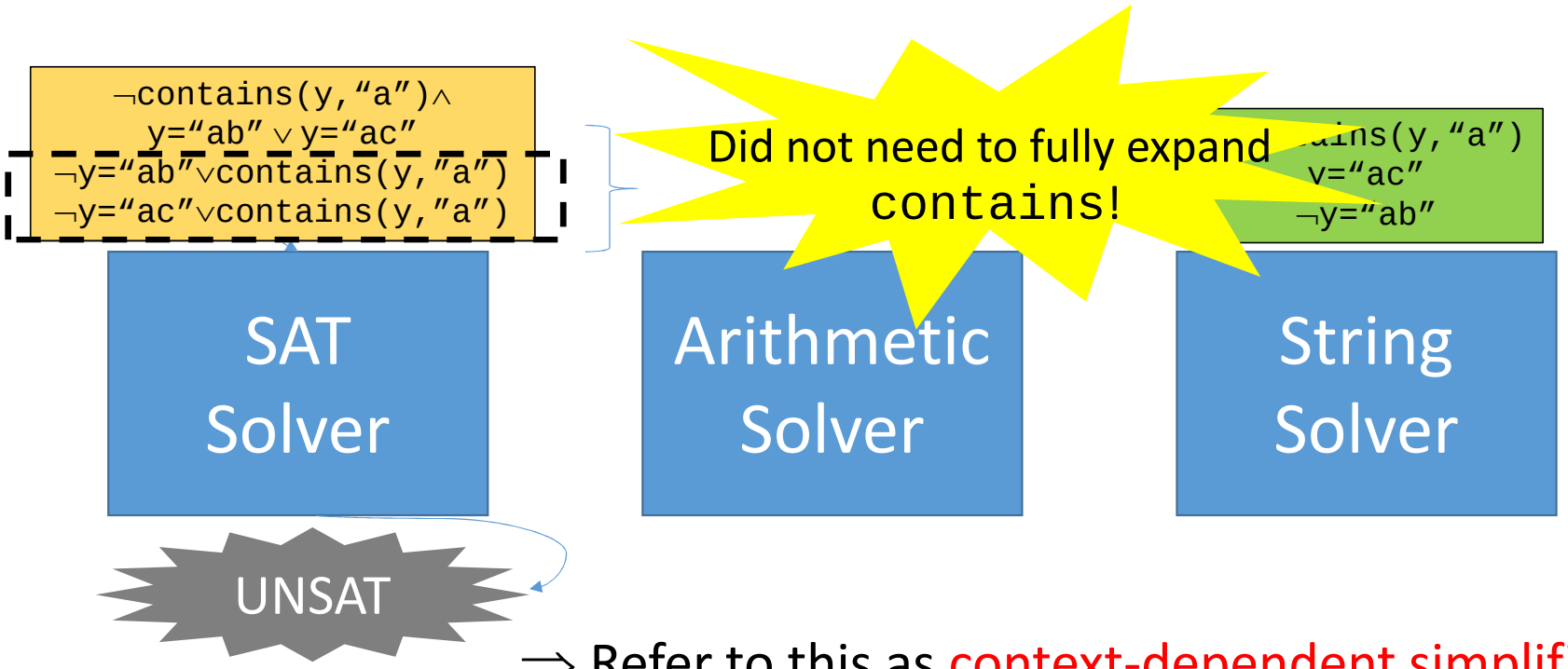
(Lazy) Expansion + Context-Dependent Simplification



(Lazy) Expansion + Context-Dependent Simplification

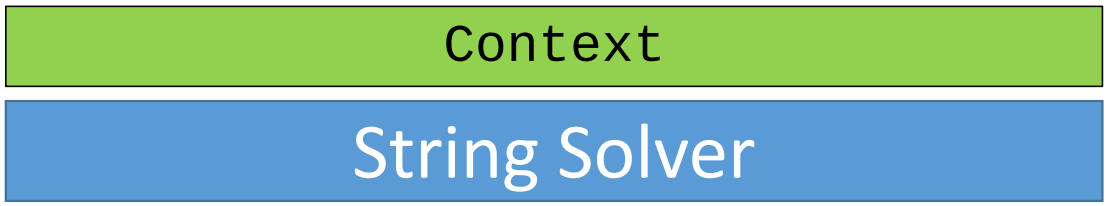


(Lazy) Expansion + Context-Dependent Simplification

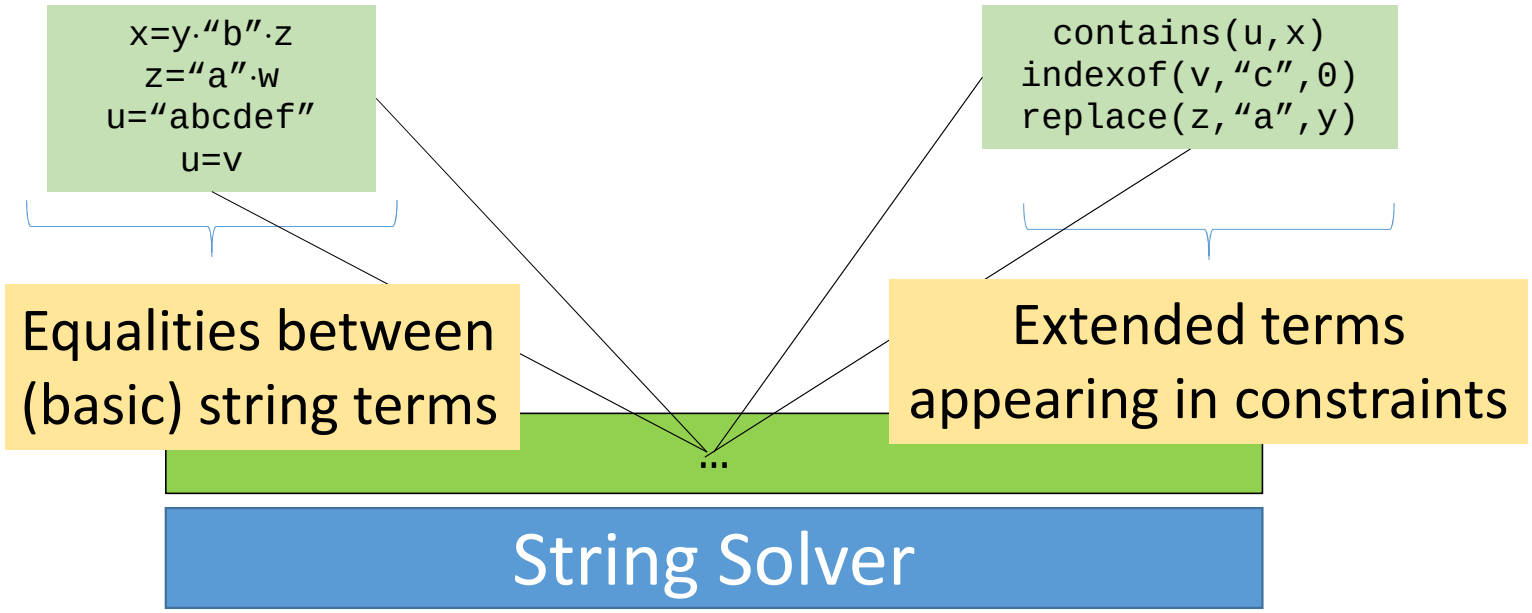


⇒ Refer to this as context-dependent simplification

Context-Dependent Simplification



Context-Dependent Simplification

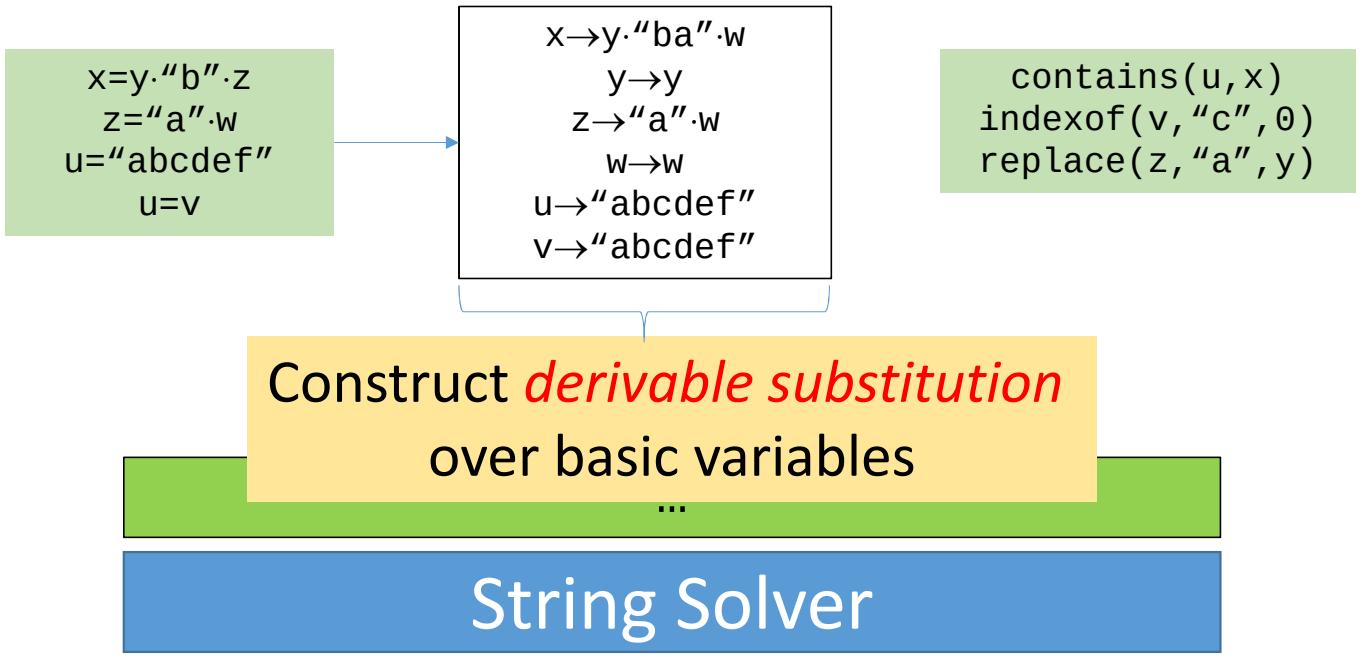


Context-Dependent Simplification

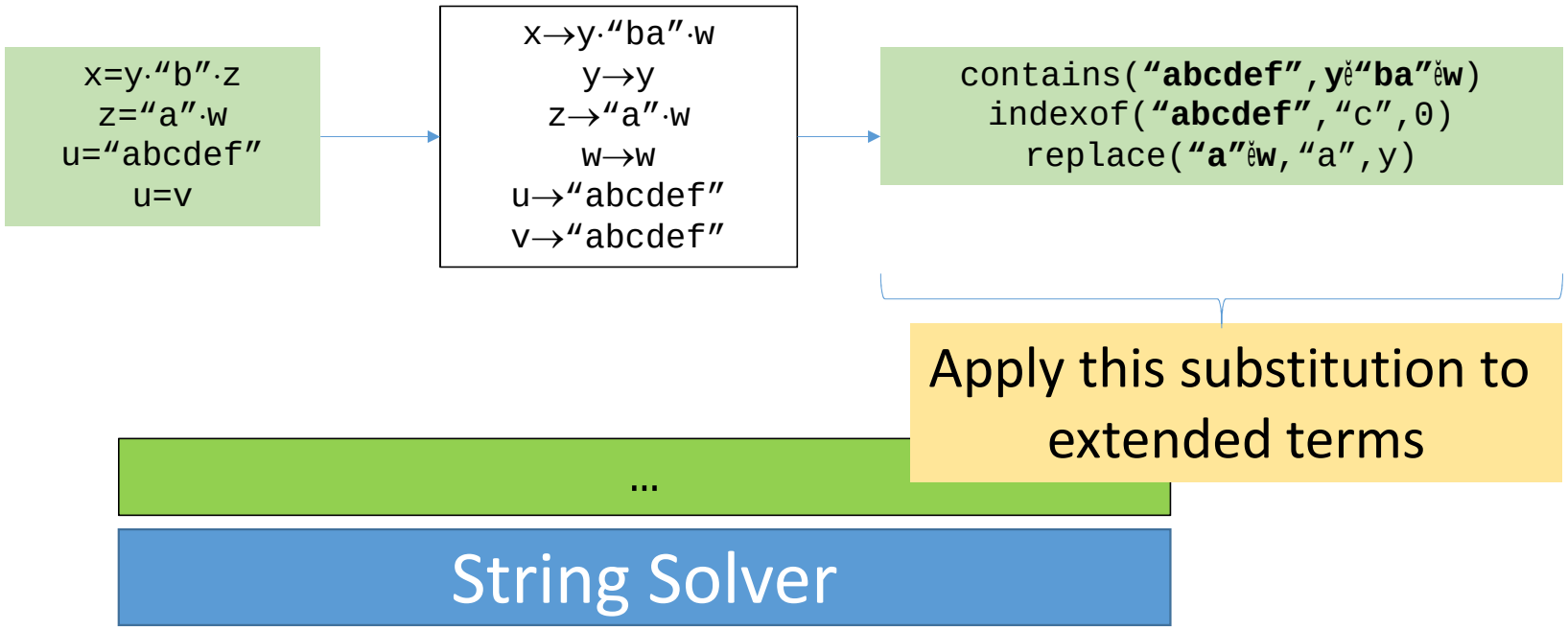
```
x=y·“b”·z
z=“a”·w
u=“abcdef”
u=v
```

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

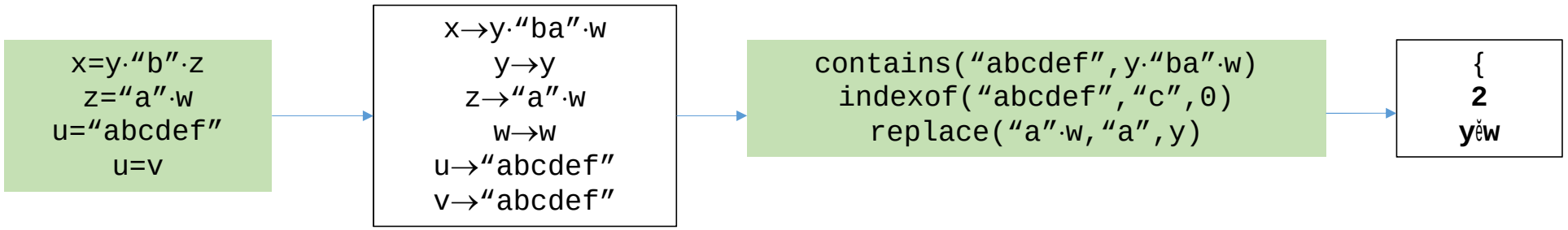
Context-Dependent Simplification



Context-Dependent Simplification

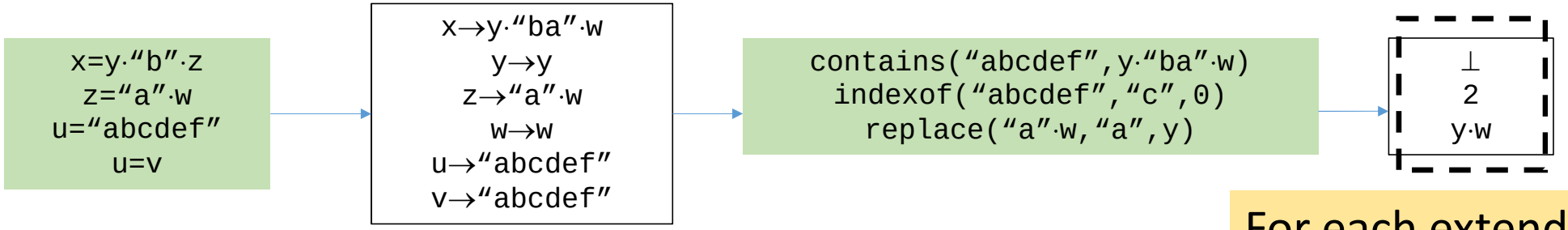


Context-Dependent Simplification



Simplify results

Context-Dependent Simplification

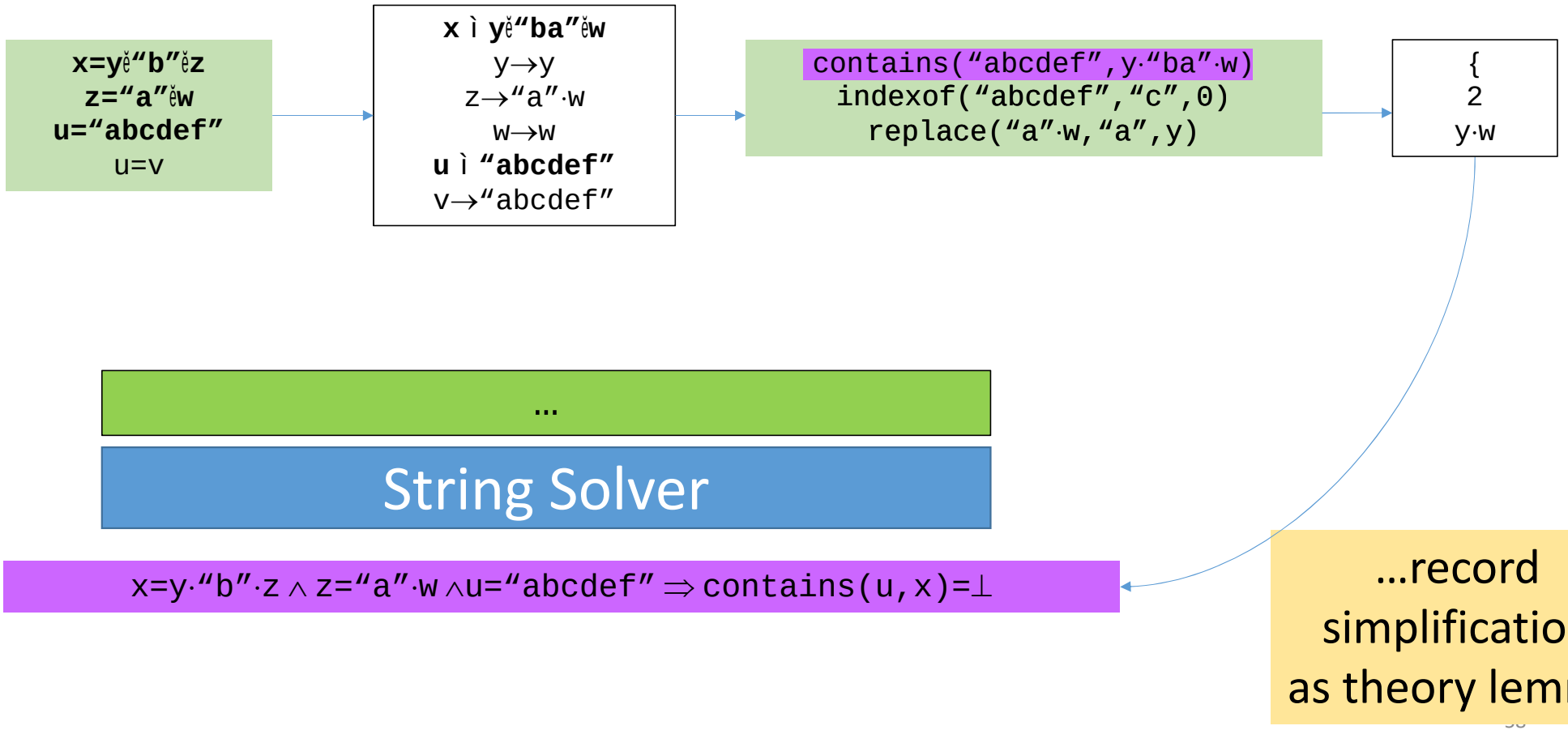


For each extended term that was simplified to a basic one…

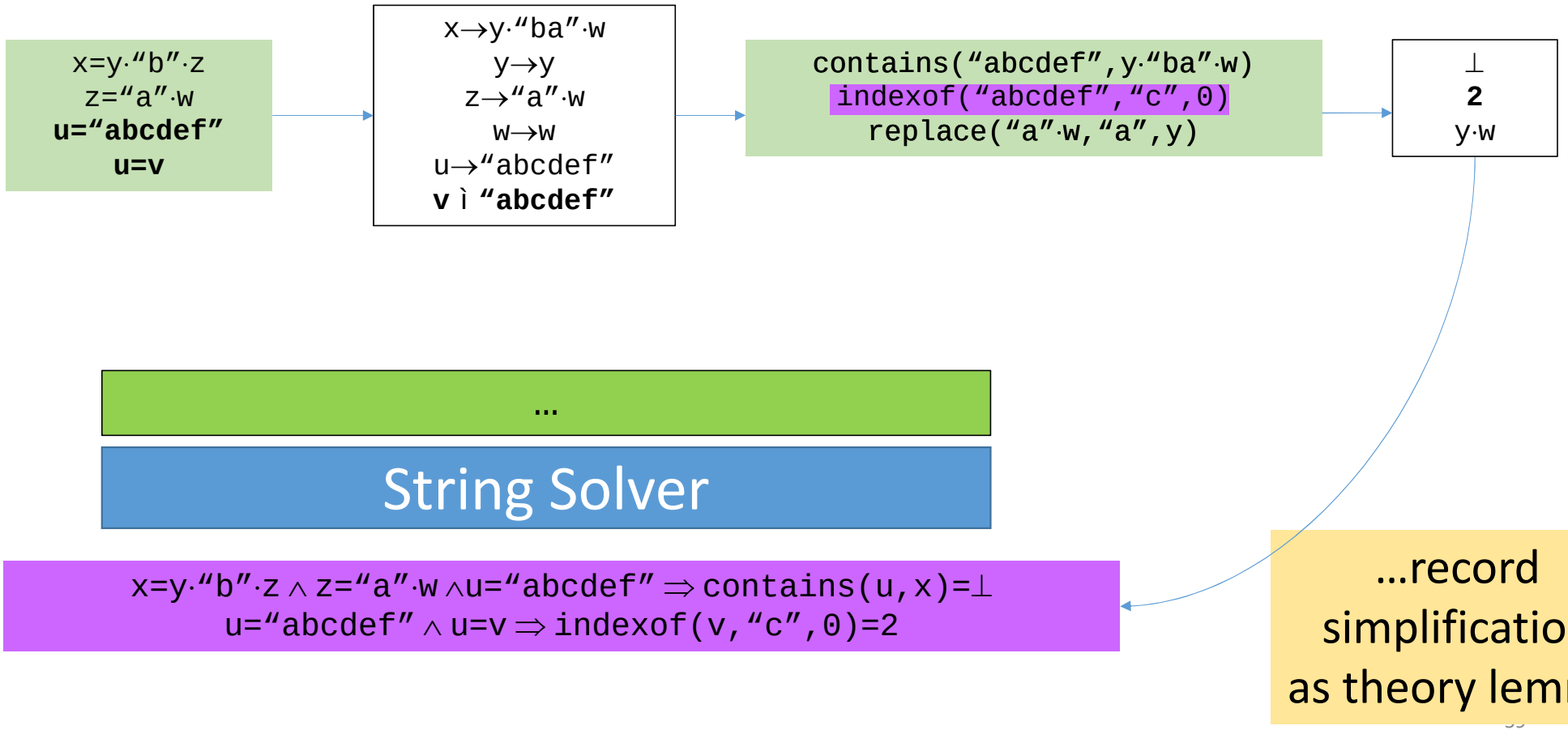
…

String Solver

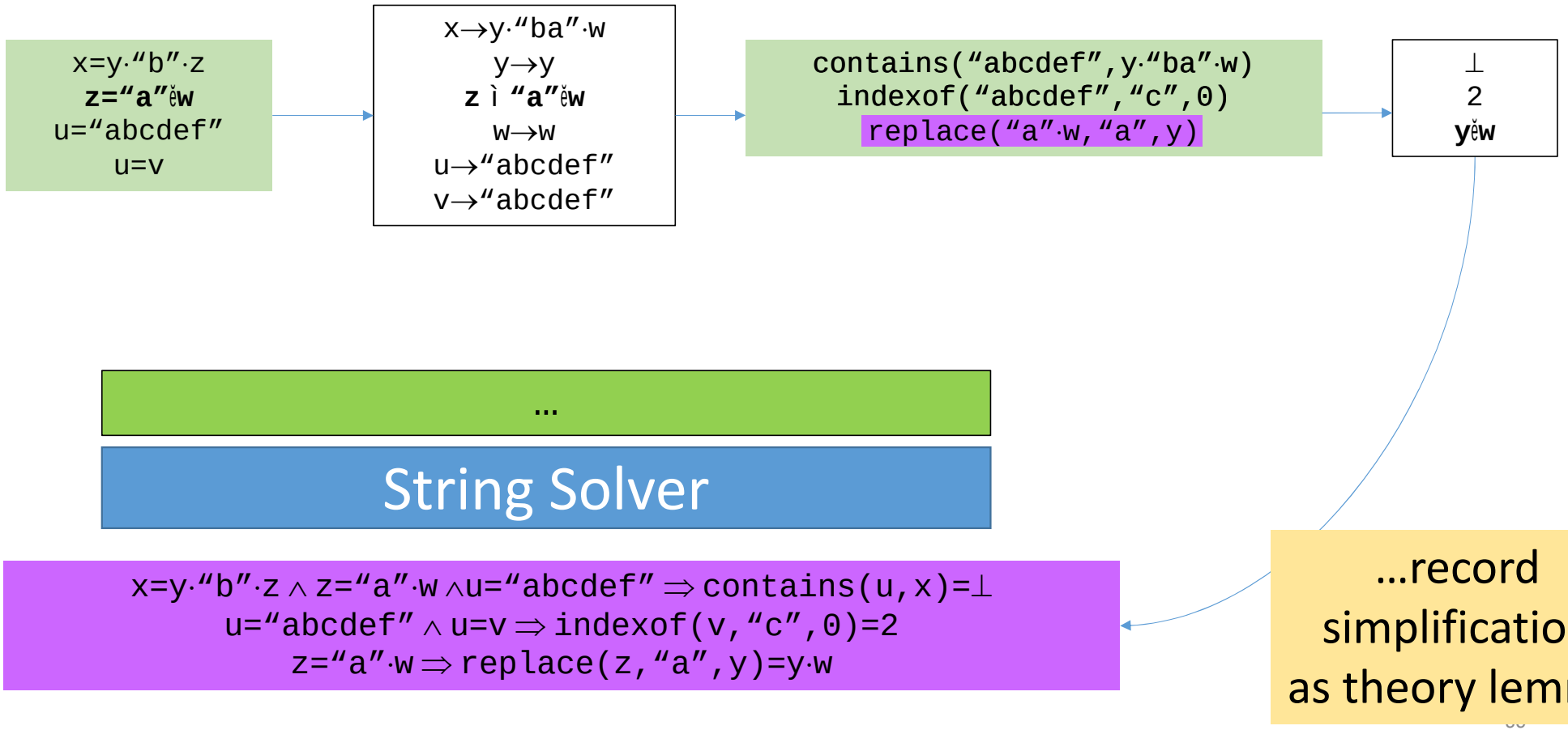
Context-Dependent Simplification



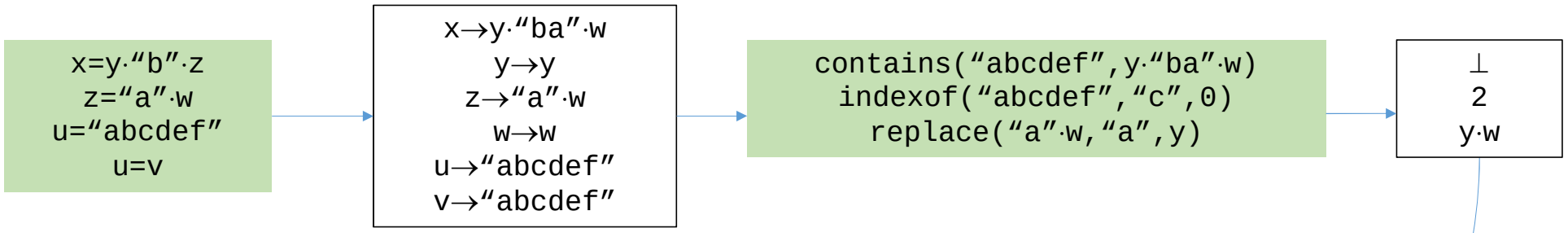
Context-Dependent Simplification



Context-Dependent Simplification



Context-Dependent Simplification



…

String Solver

$x=y\cdot\text{“b”}\cdot z \wedge z=\text{“a”}\cdot w \wedge u=\text{“abcdef”} \Rightarrow \texttt{contains}(u,x)=\bot$
$u=\text{“abcdef”} \wedge u=v \Rightarrow \texttt{indexof}(v,\text{“c”},0)=2$
$z=\text{“a”}\cdot w \Rightarrow \texttt{replace}(z,\text{“a”},y)=y\cdot w$

In our dataset, inference of this form is possible in 98% of contexts

Simplification Rules for Strings

- Unlike arithmetic:

`x+x+7*y=y-4` → `2*x+6*y+4=0`

…simplification rules for strings are highly non-trivial:

`substr(x·“abcd”,1+len(x),2)`	`“bc”`
`contains(“abcde”,“b”·x·“a”)`	⊥
`contains(x·“ac”·y,“b”)`	`contains(x,“b”)∨contains(y,“b”)`
`indexof(“abc”·x,“a”·x,1)`	`-1`
`replace(“a”·x,“b”,y)`	`con(“a”,replace(x,“b”,y))`

- Implemented in 3000+ lines of C++ code

Theoretical Contribution

- Approach described as a rule-based *calculus,* e.g.:

$$\dots$$

$$\text{Ext-Simplify}\ \frac{x \approx t \in \mathsf{X} \qquad \mathsf{E} \models \boldsymbol{y} \approx \boldsymbol{s} \qquad (t\{\boldsymbol{y} \mapsto \boldsymbol{s}\})\downarrow \text{ is a } \Sigma_{\mathsf{AS}}\text{-term}}{\mathsf{G} := \mathsf{G}, \lfloor x \approx (t\{\boldsymbol{y} \mapsto \boldsymbol{s}\})\downarrow \rfloor \qquad \mathsf{X} := \mathsf{X} \backslash \{x \approx t\}}$$

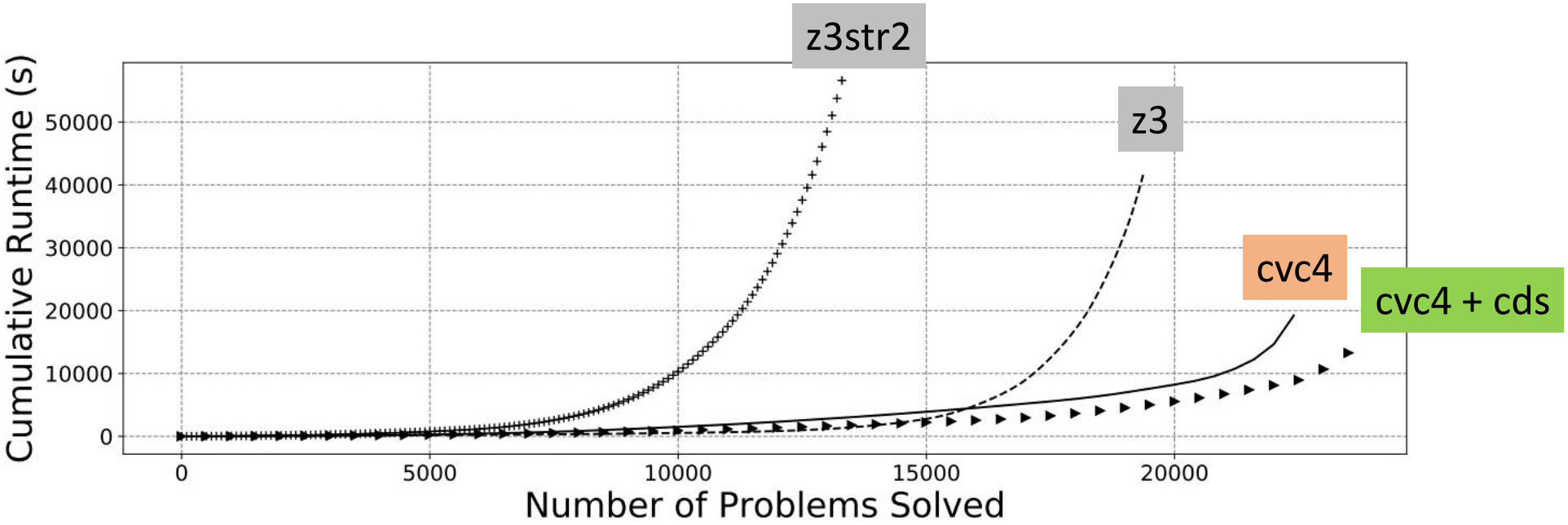
$$\dots$$

- Calculus is:
  - **Refutation-sound**
  - **Model-sound**
  - **Not terminating** in general (decidability is still unknown)

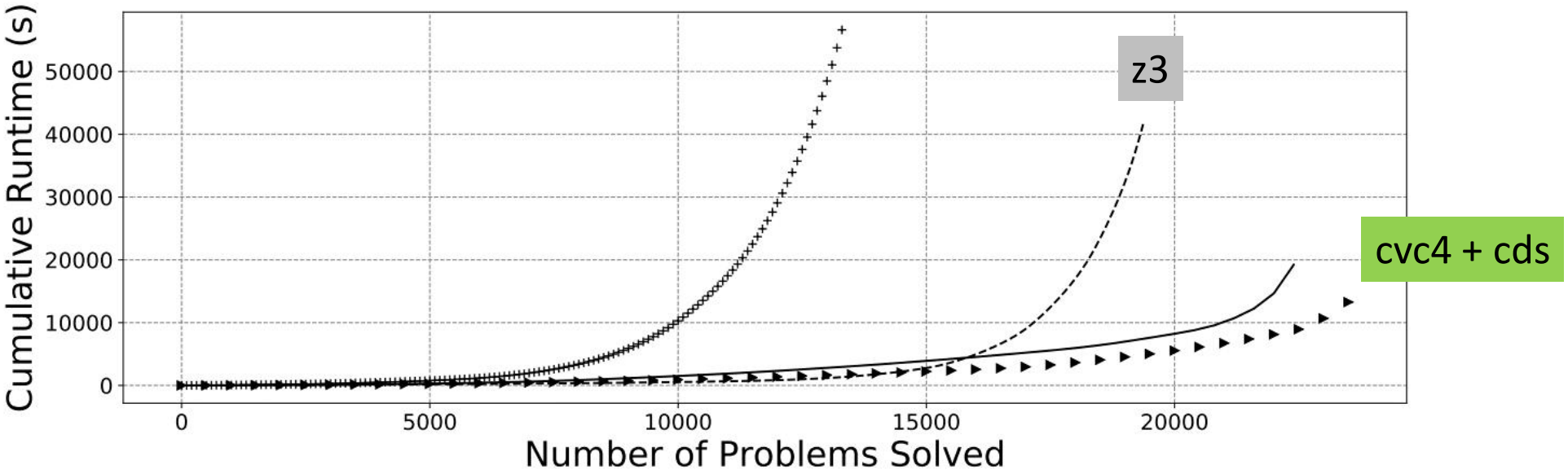
Experimental Results : PyEx Symbolic Execution

- Logged queries from PyEx symbolic execution engine (successor of PyExZ3)
  - Using z3str2, z3 and cvc4 as path constraint solver
- Total of 25,421 benchmarks over 3 runs
- Compared z3str2, z3, cvc4 w, w/o context-dependent simplification (cds)

Results : PyEx Symbolic Execution Benchmarks (25,421)

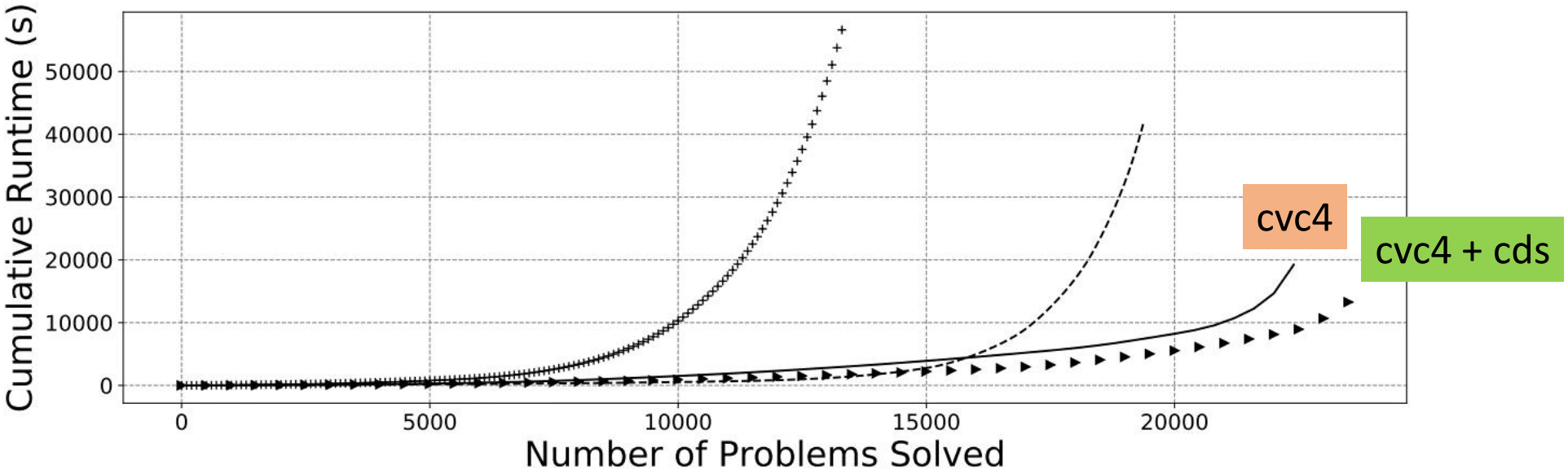


Results : PyEx Symbolic Execution Benchmarks (25,421)



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

Results : PyEx Symbolic Execution Benchmarks (25,421)



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

Impact on PyEx Symbolic Execution

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

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

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

Summary

- New technique context-dependent simplification implemented in CVC4's string solver
- Improves scalability on extended string constraints
- PyEx + CVC4 achieves comparable program coverage using 41% of the runtime as PyEx + nearest competitor

Future Work

- More aggressive simplification rules for strings
  - More powerful rules → better performance
- Quality of models for PyEx symbolic execution
  - Which models lead to higher code coverage?
- Apply context-dependent simplification to other theories:
  - Non-linear arithmetic
  - Lazy bit-blasting approaches to bit-vectors

  ⇒See **[Reynolds et al FroCoS 2017]** for details
- Simplification directly benefits Syntax-Guided Synthesis for strings in CVC4

- String solver in CVC4
  - Open source
  - Available at : http://cvc4.cs.stanford.edu/web/

- 25,421 new benchmarks from PyEx (*.smt2)
  - Available at : http://cvc4.cs.stanford.edu/papers/CAV2017-strings/

- …Thanks for listening!