Representation-Based Synthesis

Woosuk Lee

CSE9116 SPRING 2024

Hanyang University

Multiple Solutions in a Search Space

- So far: how to find *a* solution satisfying a spec
- How can we find an *optimal* solution among multiple ones?
  - *optimal*: determined by a measure (e.g., complexity, size, naturality, etc)
  - hard to find a "global optimum" due to infinite search space
- Need to efficiently explore multiple solution candidates and pick the best one

Three Data Structures

- *Version Space Algebra* (VSA)
  - With Top-down search
- *Finite Tree Automata* (FTA)
  - With Bottom-up search
- E-graph
  - With Equality saturation

Two Data Structures

- *Version Space Algebra* (VSA)
  - With Top-down search
- *Finite Tree Automata* (FTA)
  - With Bottom-up search
- E-graph
  - With Equality saturation

Version Space

- Set of all *hypotheses* consistent with observed data
- Hypothesis space H: space of possible functions in→out
- Version space $VS_{H,D} \subseteq H$
  - D: set of I/O examples $\{(i_j,o_j)\}$
  - $h \in VS_{H,D} \Leftrightarrow \forall (i,o) \in D.\ h(i) = o$

Mitchell et al., *Generalization as search*. AI 1982

Version Space Algebra

- A set of operations to manipulate and compose version spaces
- Compact *symbolic* representations for the version spaces (without explicitly enumerating hypotheses) → **efficient!**
- Operations:
  - Learn (i, o) : create a version space for i → o
  - $VS_1 \cup VS_2$, $VS_1 \cap VS_2$
  - Pick VS : pick the best one from VS

Lau et al., *Version space algebra and its application to programming by example*. ICML 2000

Syntax

$$\tilde{P} ::= \{P_1, \cdots, P_k\} \mid \cup(\tilde{P}_1, \cdots, \tilde{P}_k) \mid F_{\bowtie}(\tilde{P}_1, \cdots, \tilde{P}_k)$$

Direct Set

Union: "***symbolize***" union of multiple VSs

Join: "***symbolize***" relational join of multiple argument VSs

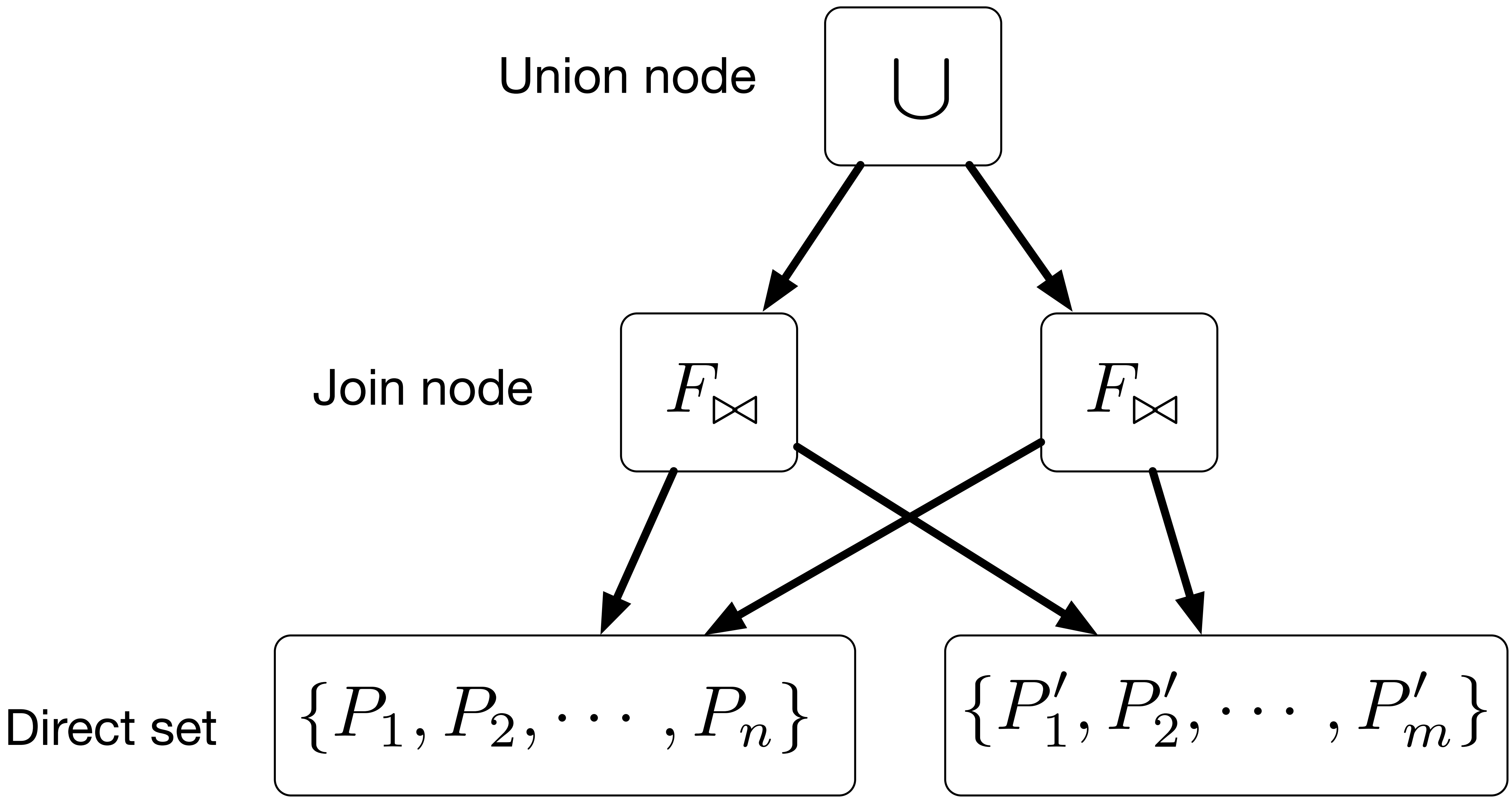
"***Symbolize***": just a representation without any actual union/join operation

Semantics

Program P belongs to VSA $\tilde{P}$ ($P \in \tilde{P}$) iff

- $\tilde{P} = \{P_1, \cdots, P_k\}$, P is either one of $P_1, \cdots, P_k$
- $\tilde{P} = \cup(\tilde{P}_1, \cdots, \tilde{P}_k)$, P belongs to either one of $\tilde{P}_1, \cdots, \tilde{P}_k$
- $\tilde{P} = F_{\bowtie}(\tilde{P}_1, \cdots, \tilde{P}_k)$, P = $F(P_1, \cdots, P_k)$, $P_i \in \tilde{P}_i$

VSA as a Graph



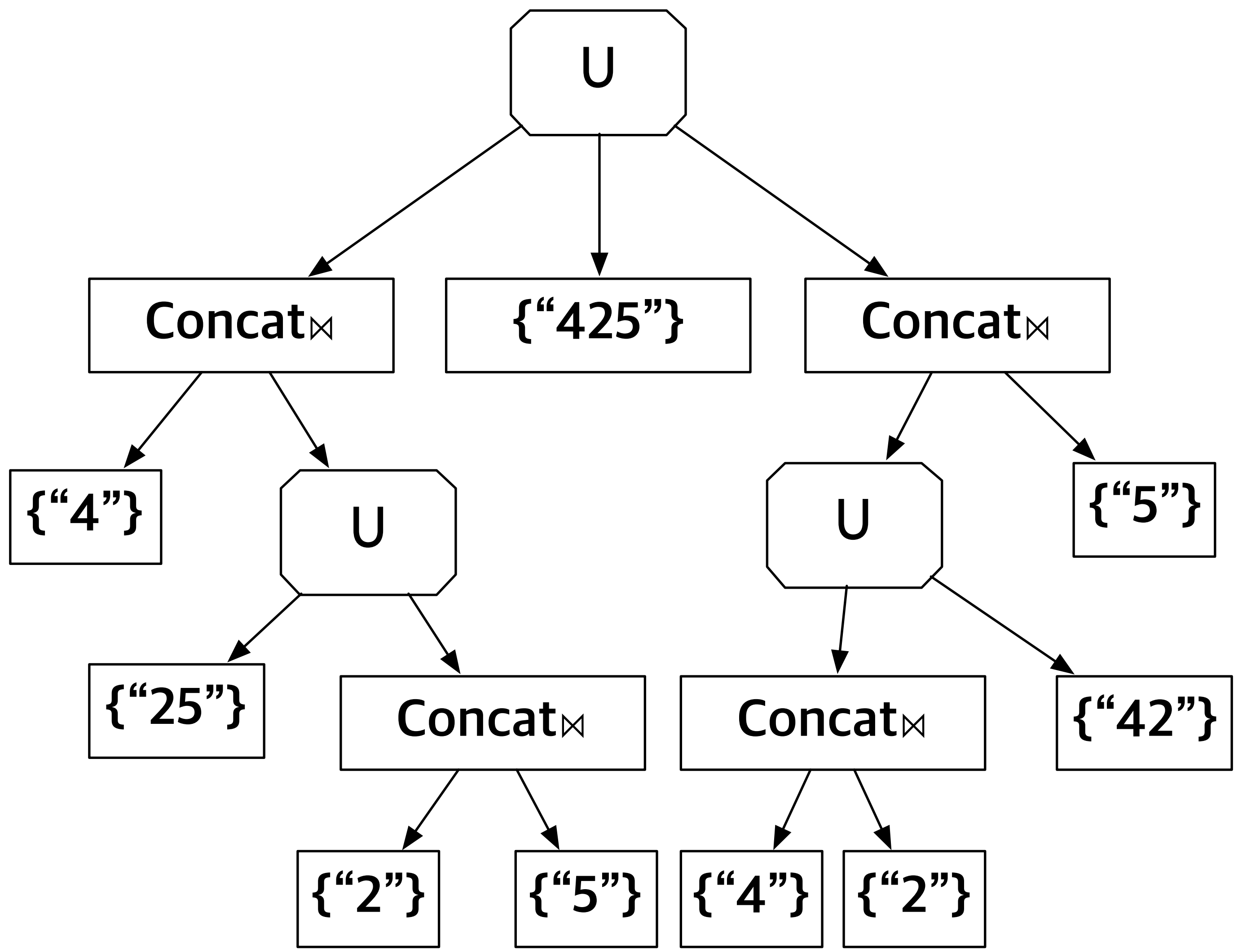
VSA Example

- Set of programs outputting string "425" when string concatenation operator (Concat) is allowed

```
{ “425”, Concat(“4”, “25”),
  Concat(“42”, “5”),
  Concat(“4”, Concat(“2, 5”)),
  Concat(Concat(“4”, “2”), “5”),
  …  }
```

as a VSA

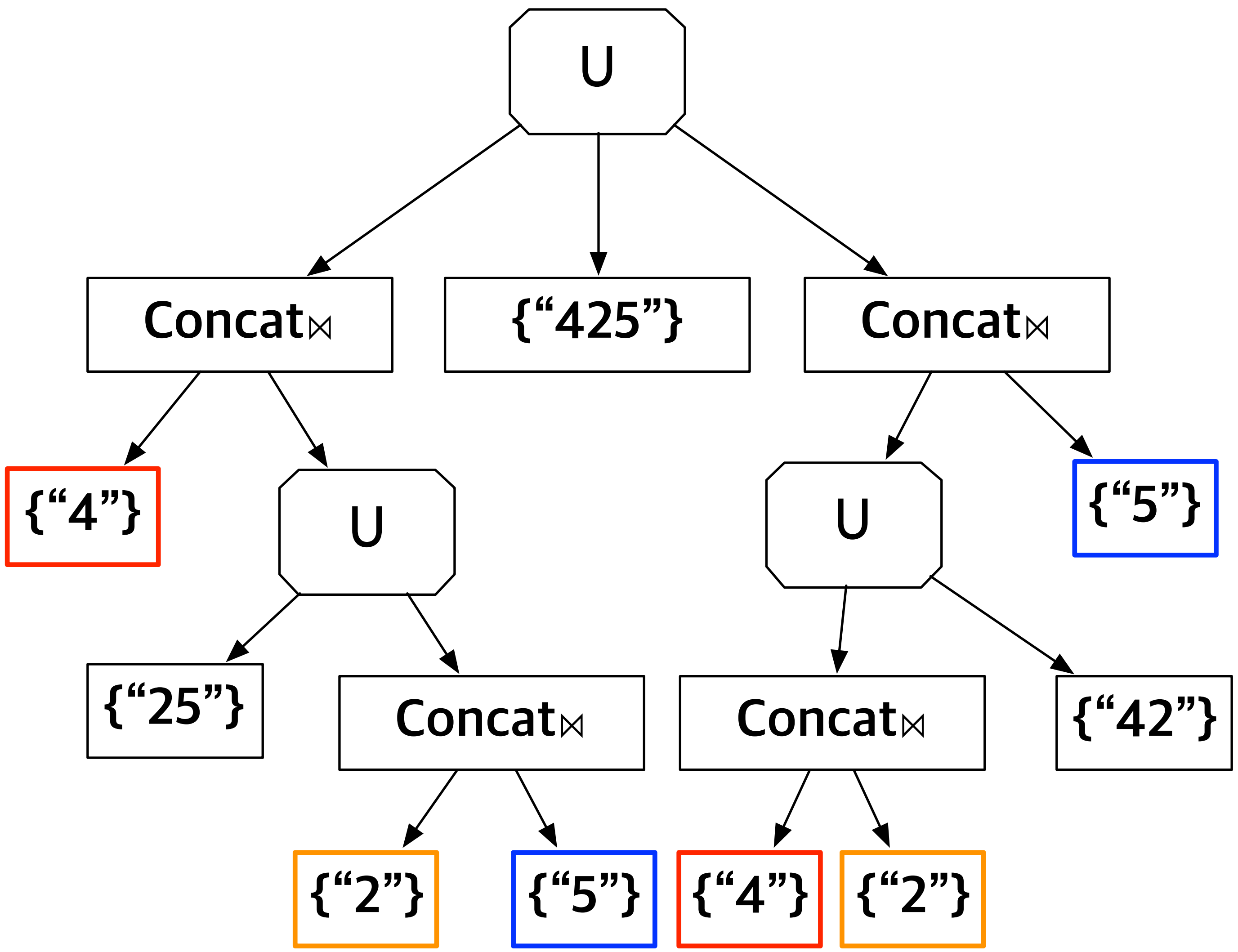
VSA Example



$$\textbf{Concat}_{\bowtie}(\textbf{VS}_1, \textbf{VS}_2) = \{\textbf{Concat}(P_1, P_2) \mid P_1 \in \textbf{VS}_1 , P_2 \in \textbf{VS}_2\}$$

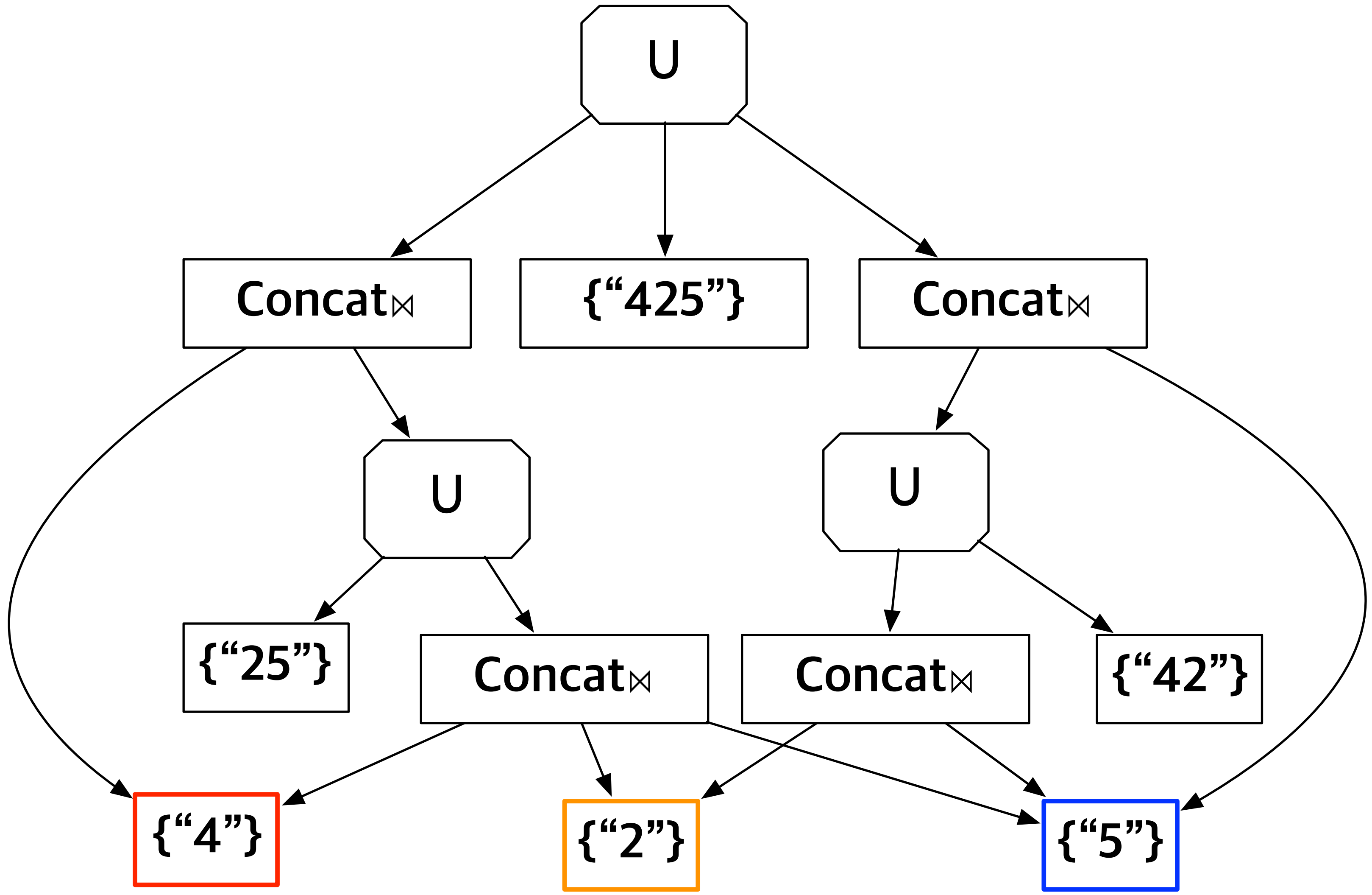
VSA Example

Duplicated nodes



VSA Example

Deduplication



Complexity

- V(VSA): # of nodes in VSA
- |VSA|: # of programs in VSA
- V(VSA) = O(log |VSA|)

How to Construct VSAs?

- Top-down propagation (a.k.a top-down deduction)
- Excel FlashFill
  - Gulwani: Automating string processing in spreadsheets using input-output examples. POPL 2011.

Example

- Goal: synthesize f that concatenates "U" and string x

- Spec

Syntax:

$$S \to x \mid \mathrm{ConCat}(S, S) \mid ConstStr$$

Semantics:

$$f(\text{“}SA\text{”}) = \text{“}USA\text{”} \wedge f(\text{“}AE\text{”}) = \text{“}UAE\text{”}$$

Solution: $\mathrm{ConCat}(\text{“}U\text{”}, x)$

Top-Down Propagation

- If a program of form $F(e_1, \cdots, e_k)$ outputs o, what should the argument expressions $e_1, \cdots, e_k$ output respectively?

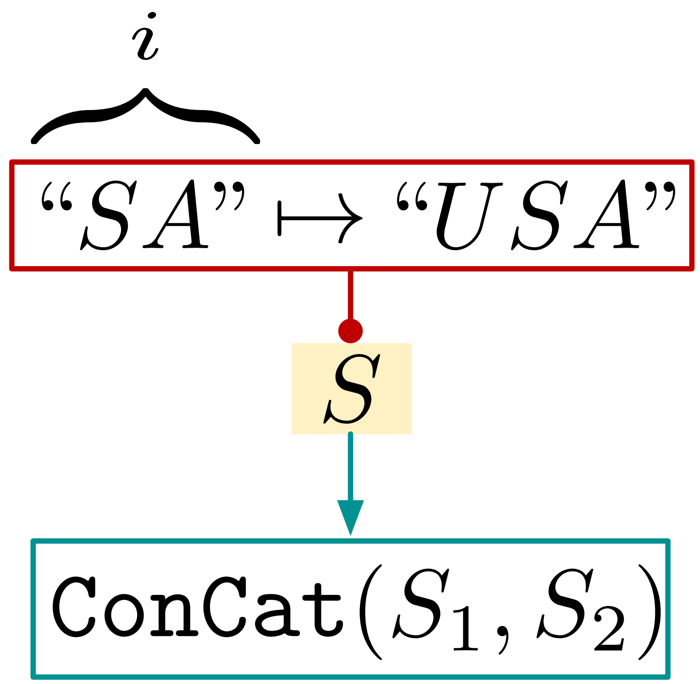
- Inverse semantics operator (a.k.a witness function)

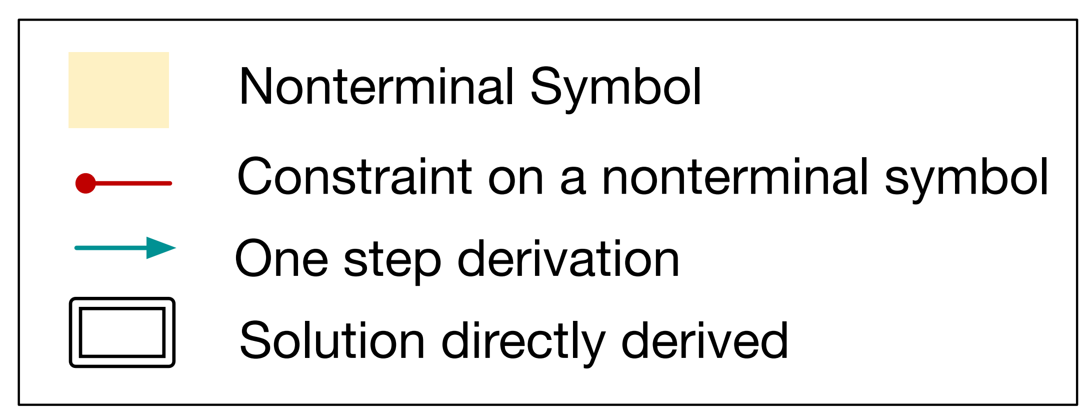
$$F^{-1}(o) = \{(a_1, \cdots, a_k) \mid F(a_1, \cdots, a_k) = o\}$$

- Possible inputs (inverse-set or pre-image):

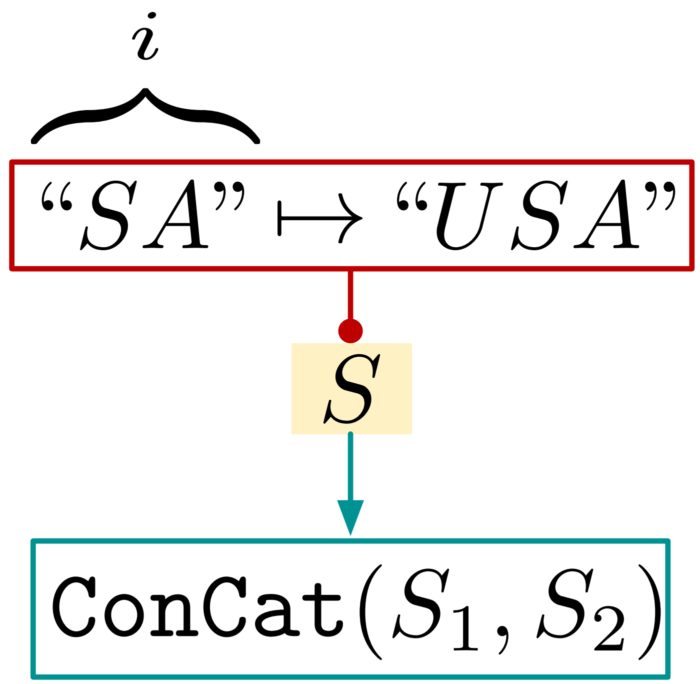
e.g., $\mathrm{ConCat}^{-1}(\text{“}USA\text{”}) = \{(\text{“}U\text{”}, \text{“}SA\text{”}), (\text{“}US\text{”}, \text{“}A\text{”})\}$

Top-Down Propagation

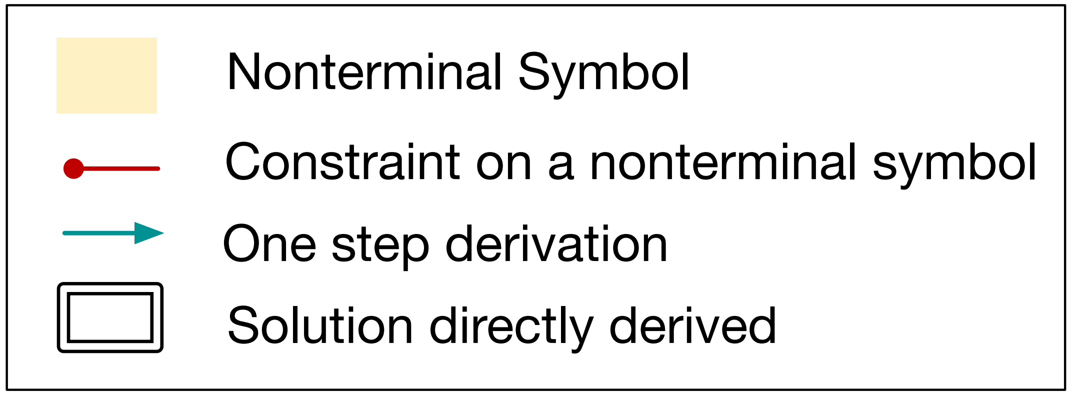




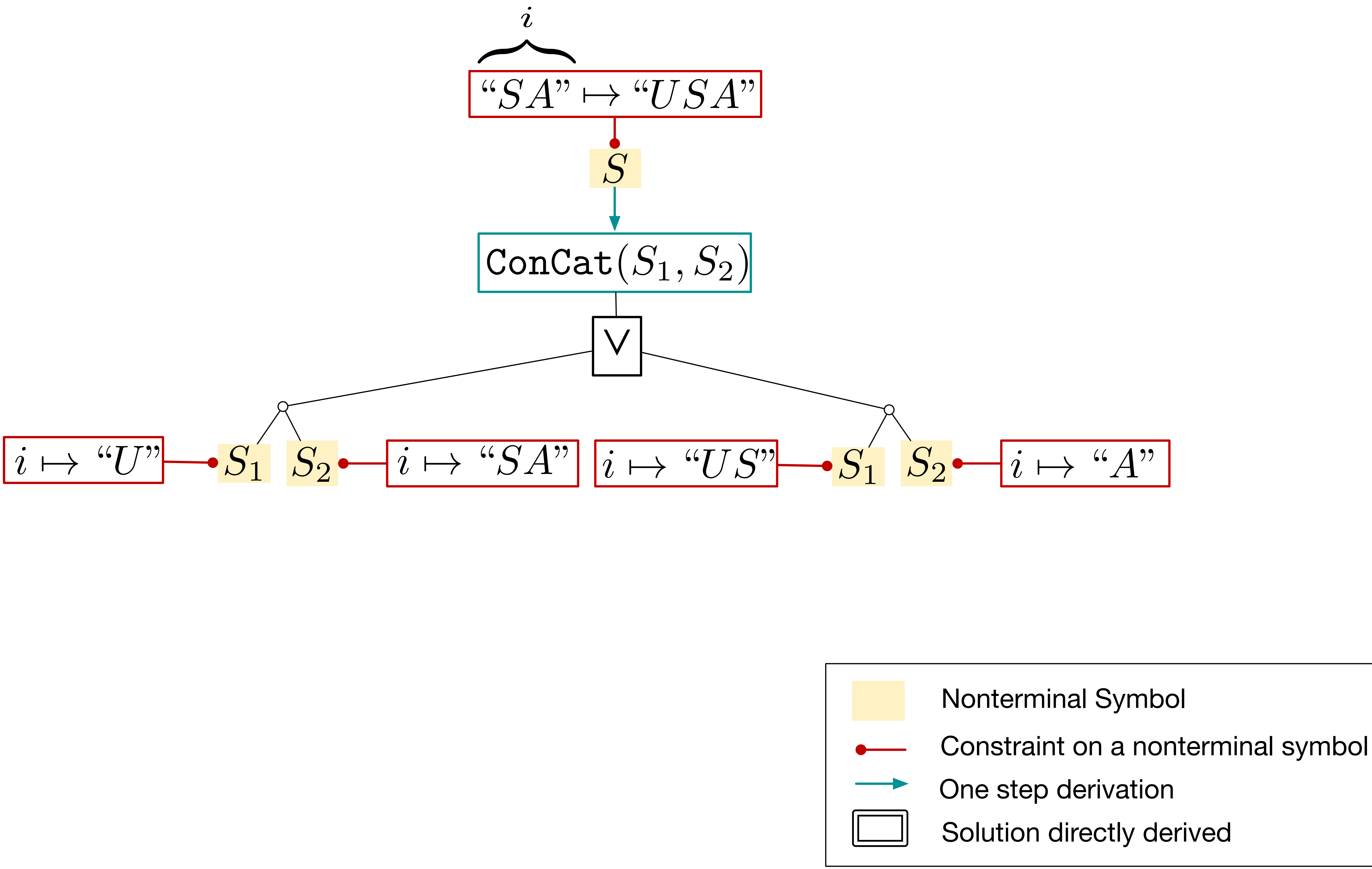
Top-Down Propagation



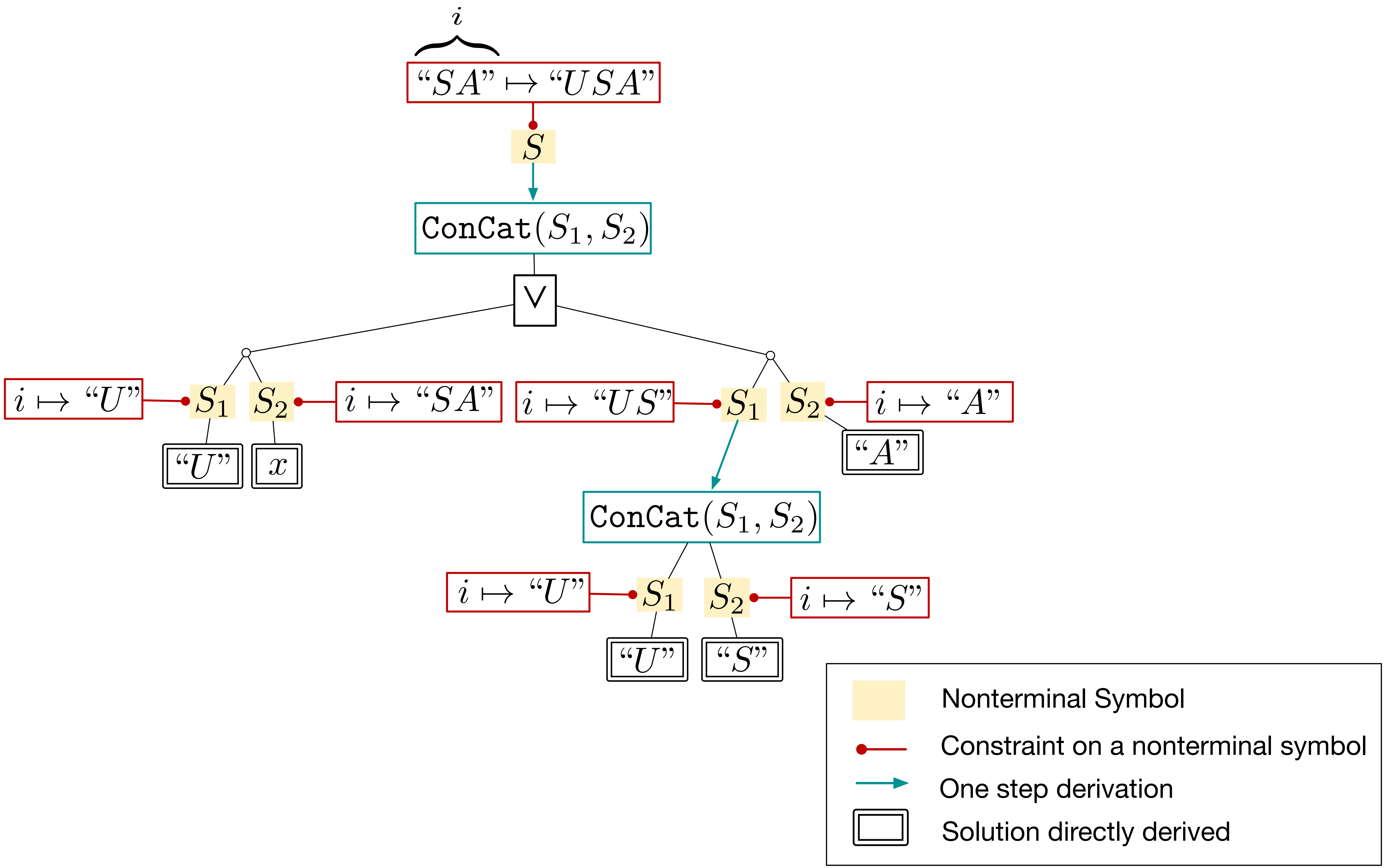
$$\mathrm{ConCat}^{-1}(\text{"}USA\text{"}) = \{(\text{"}U\text{"}, \text{"}SA\text{"}), (\text{"}US\text{"}, \text{"}A\text{"})\}$$



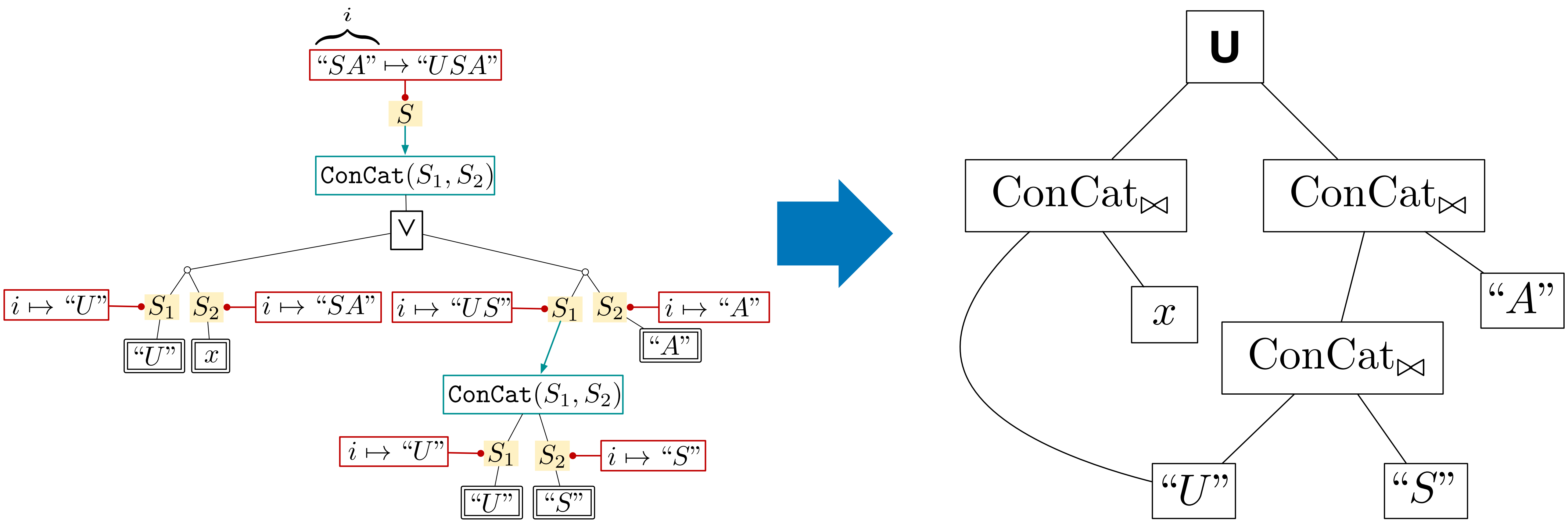
Top-Down Propagation



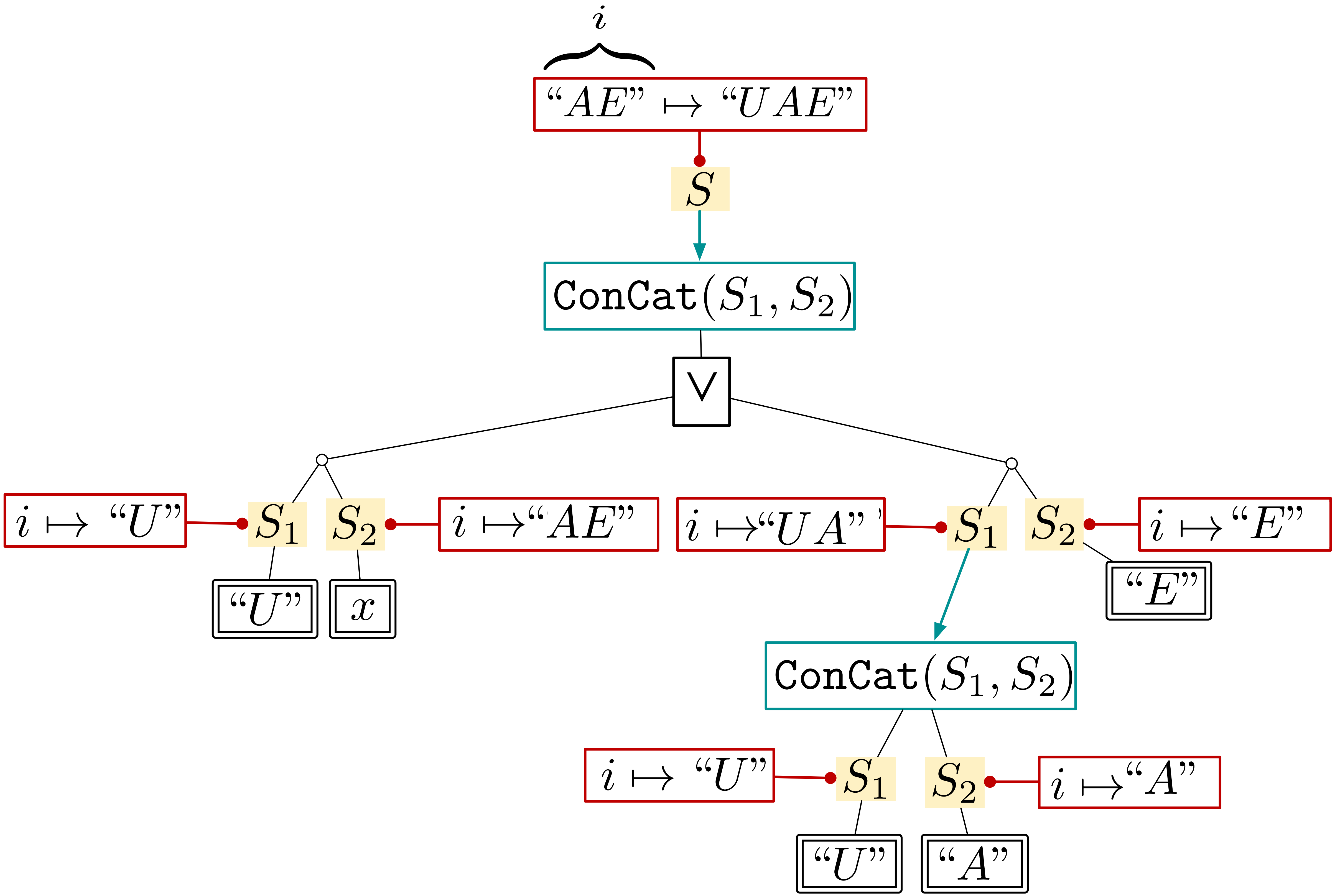
Top-Down Propagation



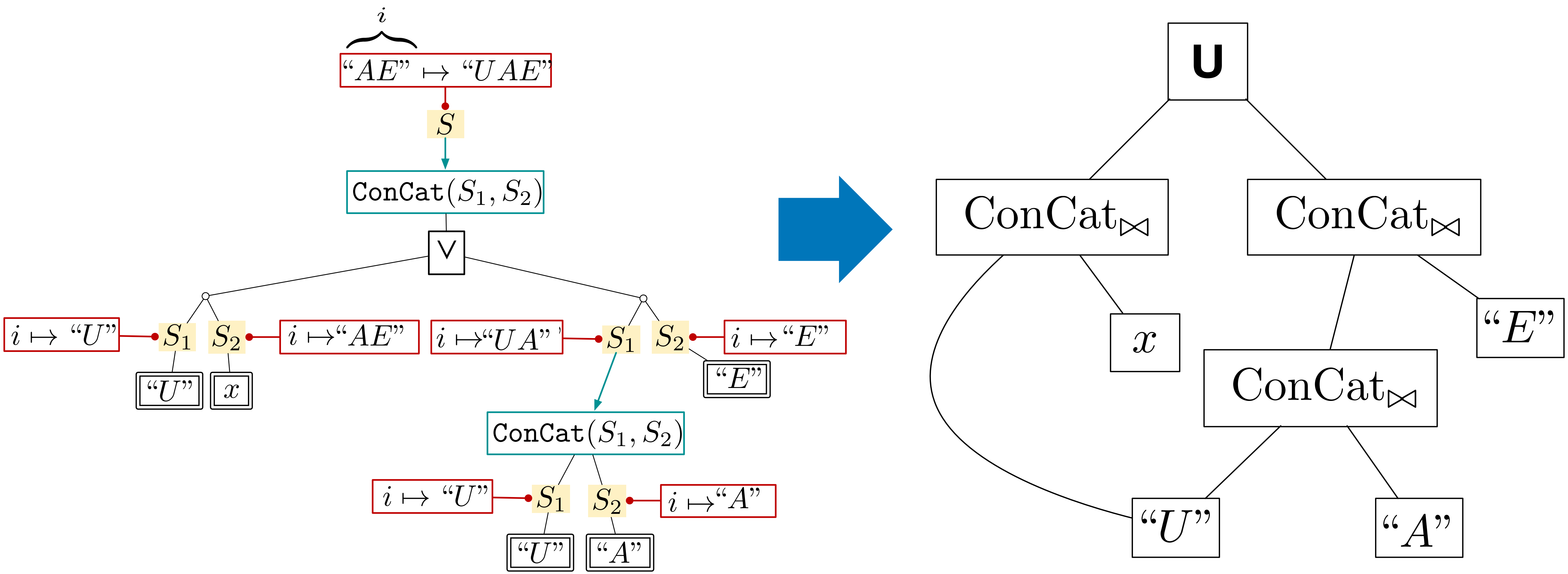
VSA Construction



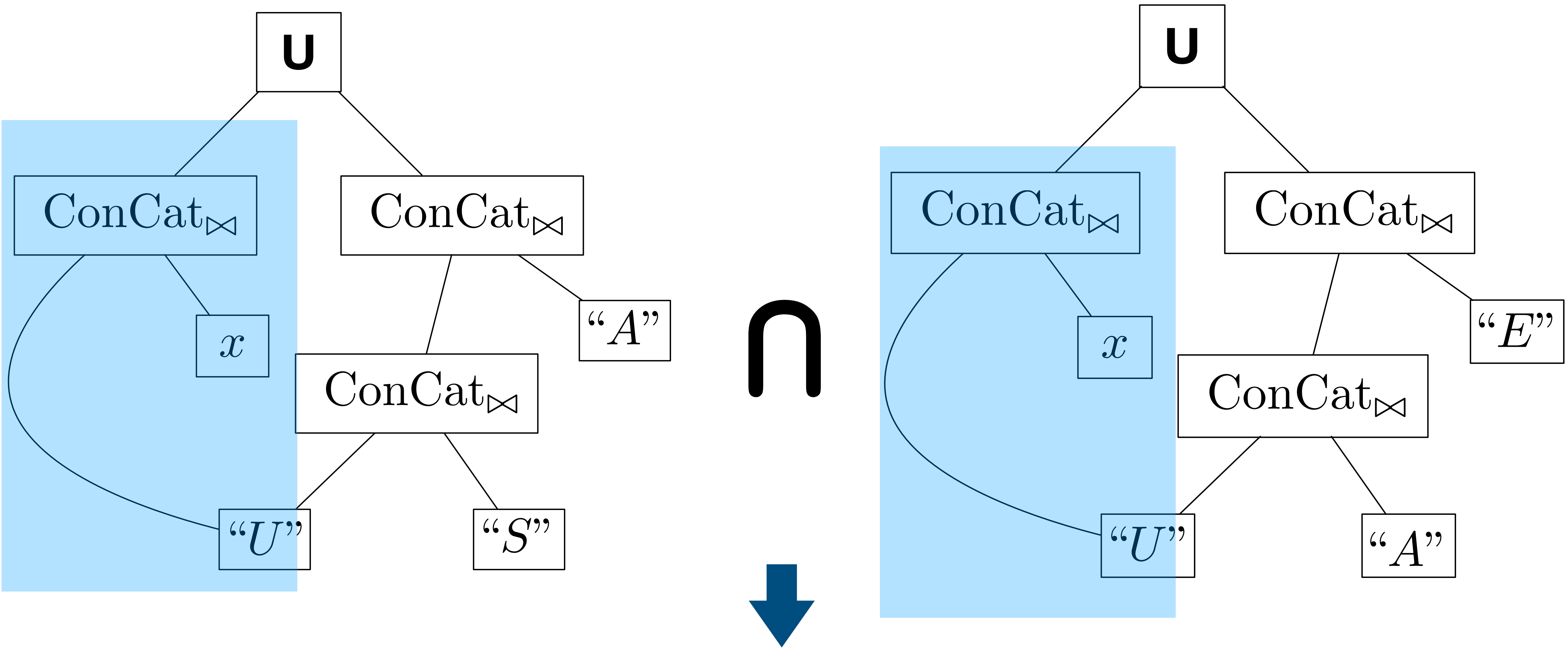
Similarly for the Other Example



VSA Construction



VSA Intersection



Final Solution

$$\mathrm{ConCat}(``U", x)$$

Complexity of VSA intersection: $O(V(VSA)^2)$

Finding Top-*k* Solutions in VSA

- Given a VSA $\tilde{P}$, and a scoring function $h : Program \to \mathbb{R}$ *monotonic over the program structure*

$$\forall i.\ h(P_i) > h(P_i') \implies h(F(P_1, \cdots, P_m)) > h(F(P_1', \cdots, P_m'))$$

- Function $\text{Top}_h(\tilde{P}, k)$ returns a set of programs corresponding to *k* highest values of *h* in $\tilde{P}$ is defined as follows:

Finding Top-*k* Solutions in VSA

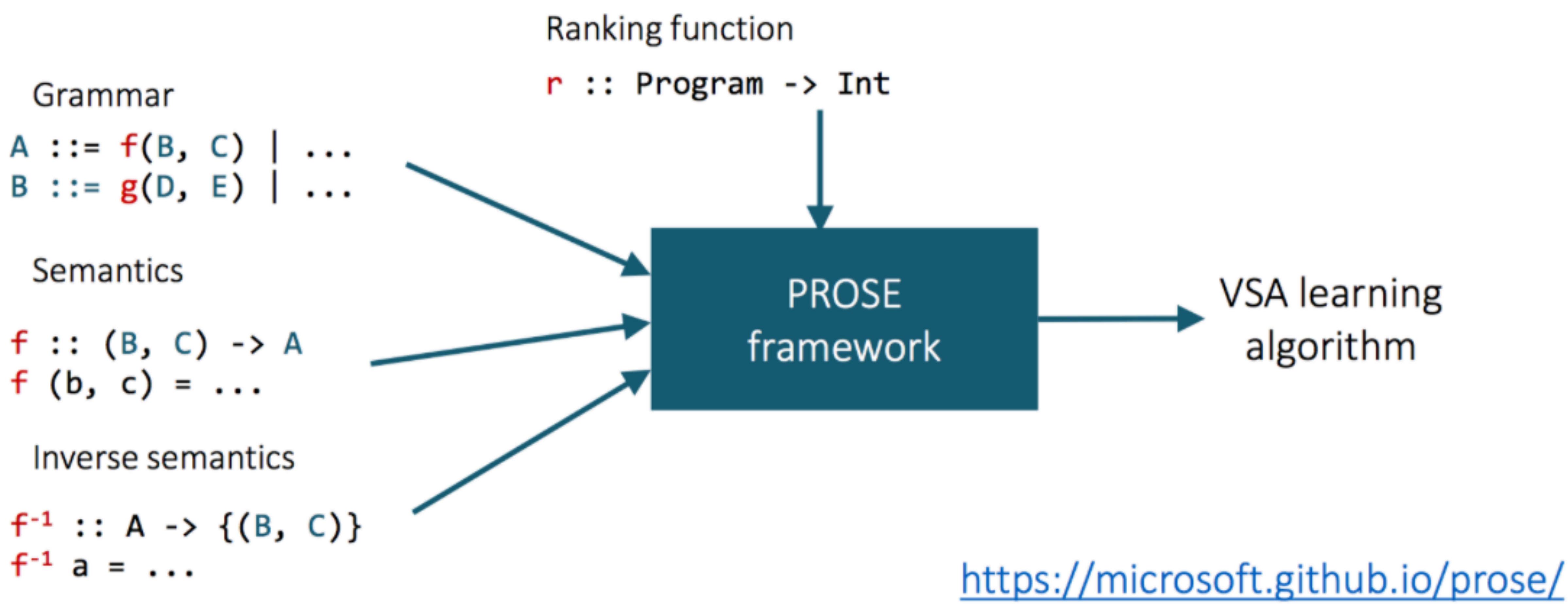
Pick top *k* elements among programs according to function *h*

$$\text{Top}_h(\{P_1, \cdots, P_m\}, k) \triangleq \text{Select}(h, k, \{P_1, \cdots, P_m\})$$

$$\text{Top}_h(\bigcup(\tilde{P}_1, \cdots, \tilde{P}_m), k) \triangleq \text{Select}(h, k, \bigcup_{i=1}^{n} \text{Top}_h(\tilde{P}_i, k))$$

$$\text{Top}_h(F_{\bowtie}(\tilde{P}_1, \cdots, \tilde{P}_m), k) \triangleq \text{Select}(h, k, \{F(P_1, \cdots, P_m) \mid P_i \in \text{Top}_h(\tilde{P}_i, k)\})$$

PROSE Framework



Polozov et al. FlashMeta: A Framework for Inductive Program Synthesis, OOPSLA 2015

Instances of PROSE Framework

- Data wrangling
  - Shipped with Microsoft Excel and Powershell
- Data extraction
  - FlashExtract: A Framework for Data Extraction by Examples, PLDI'14
  - Web Data Extraction using Hybrid Program Synthesis: A Combination of Top-down and Bottom-up Inference, SIGMOD'20
- Automated code refactoring
  - On the Fly Synthesis of Edit Suggestions, OOPSLA'19
  - Learning Syntactic Program Transformations from Examples, ICSE'17

  …

Pros and Cons of VSA-based Synthesis

- Pros: very efficient!
  - Shipped with commercial SW tools
- Cons: limited applicability
  - Inverse semantics operators should be able to efficiently compute a *finite* pre-image
  - Inverse semantics operators should generate strictly smaller subproblems.
    - Need to limit depth of grammar otherwise

Three Data Structures

- *Version Space Algebra* (VSA)
  - With Top-down search
- *Finite Tree Automata* (FTA)
  - With Bottom-up search
- E-graph
  - With Equality saturation

Example Problem

Grammar

```
N ::= id(V) | N + T | N * T
T ::= 2 | 3
V ::= x
```

Spec

1 → 9

From https://github.com/nadia-polikarpova/cse291-program-synthesis/

Finite Tree Automata

$\langle A, \mathbb{Z} \rangle$

$A \in \{\mathsf{N}, \mathsf{T}, \mathsf{X}\}$

$\{\langle \mathsf{N}, 9 \rangle\}$

states

final states

$$\mathcal{A} = \langle Q, F, Q_f, \Delta \rangle$$

alphabet

transitions

`id`, `+`, `*`

$f(q_1, \ldots, q_n) \rightarrow q$

+(<N,1>,<T,2>) → <N,3>

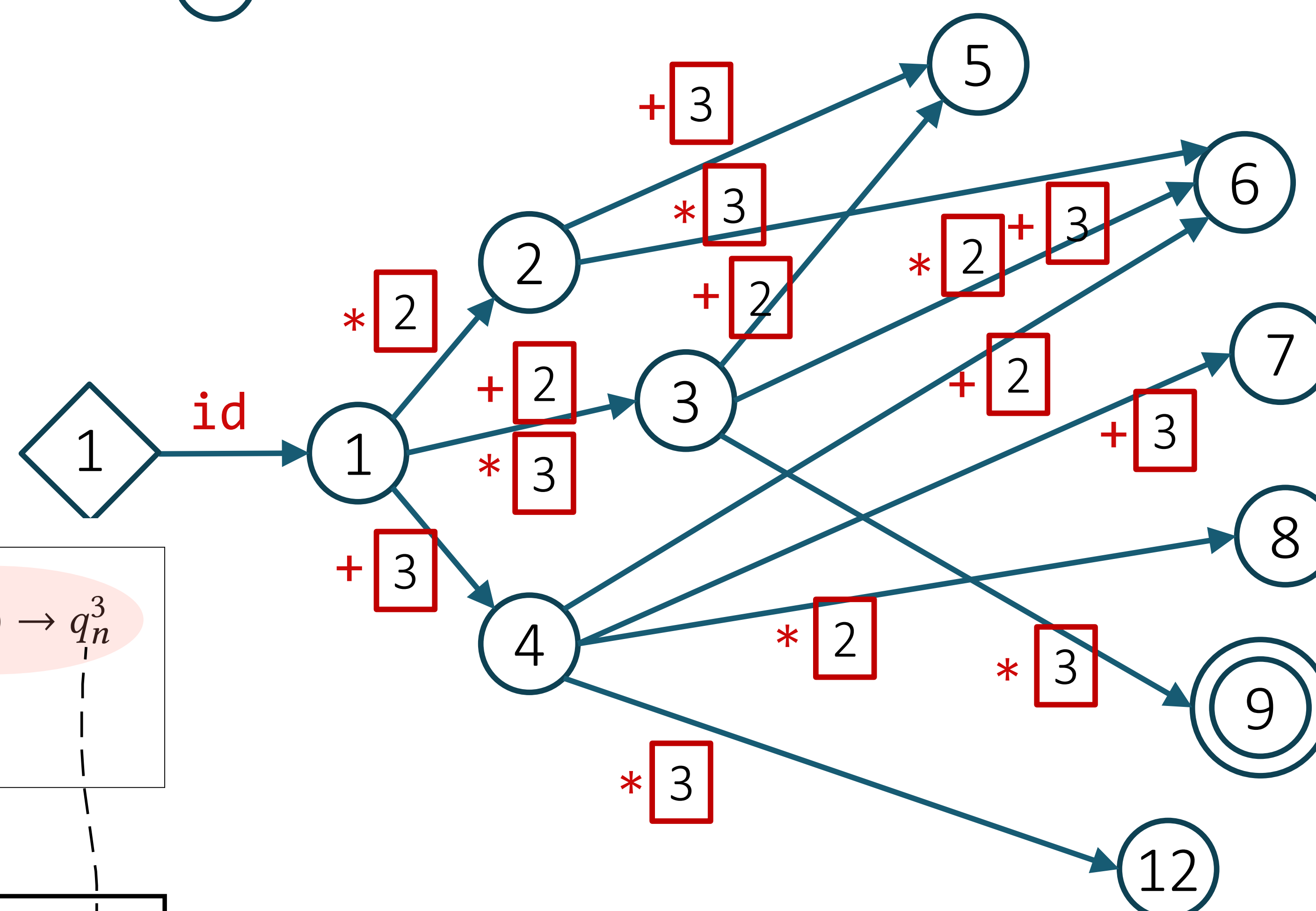
...

From https://github.com/nadia-polikarpova/cse291-program-synthesis/

Finite Tree Automata Example

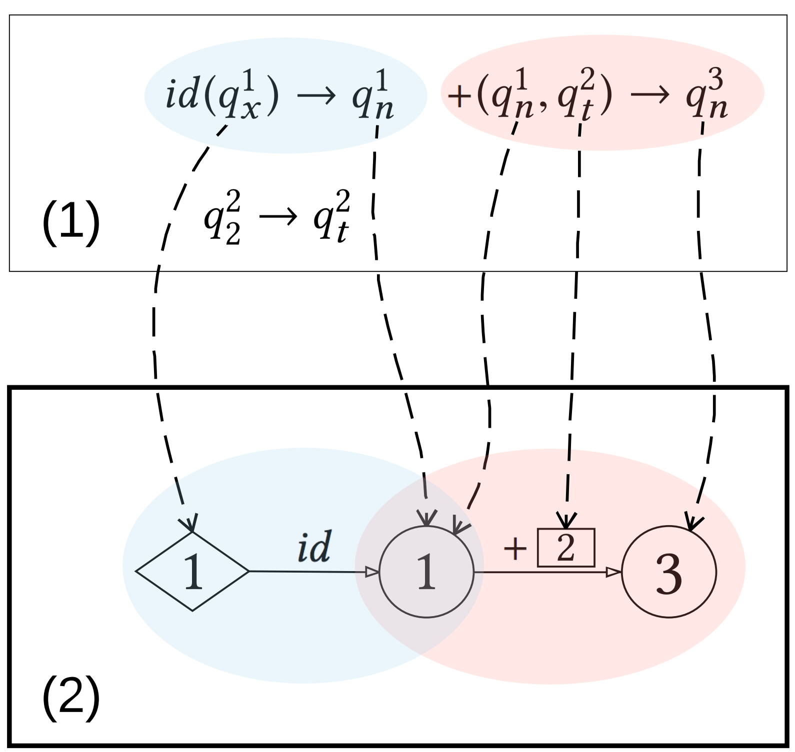
```
N ::= id(V) | N + T | N * T
T ::= 2 | 3
V ::= x
```

1 → 9



Transitions

Representation we use in examples



Synthesis of Data Completion Scripts using Finite Tree Automata . OOPSLA'17.

From https://github.com/nadia-polikarpova/cse291-program-synthesis/

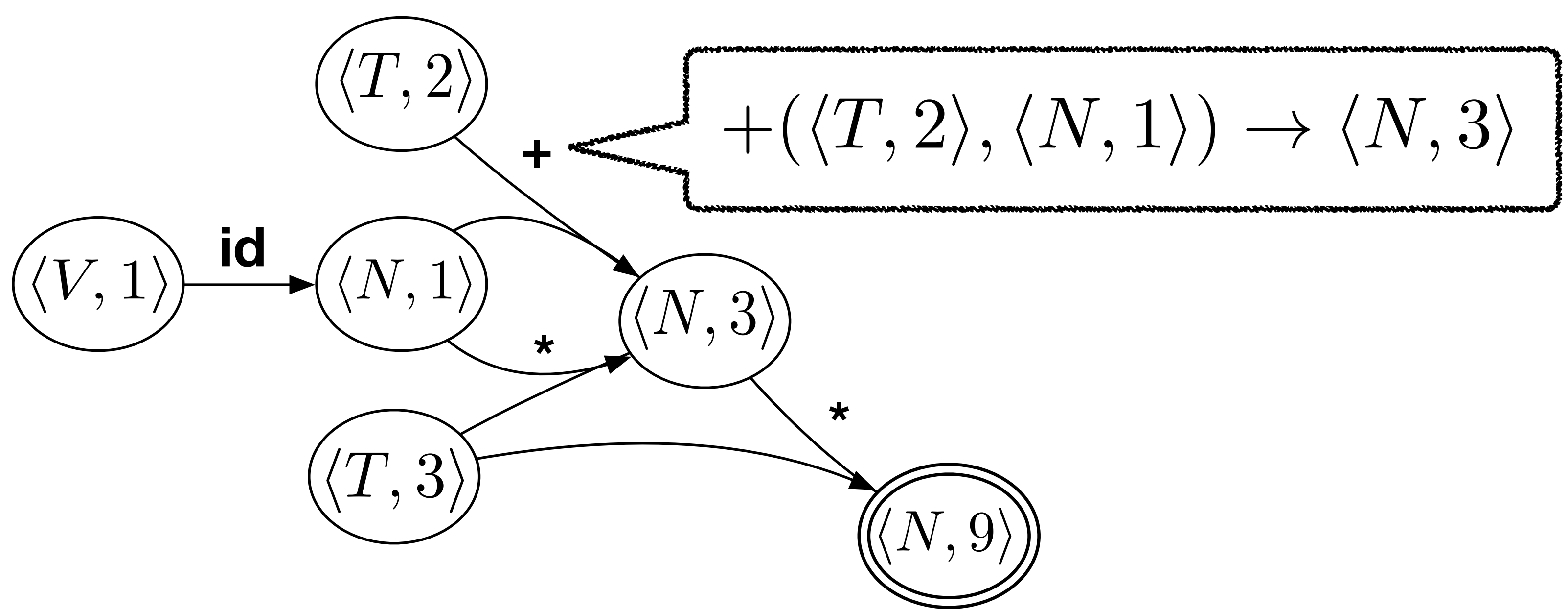
Finding a Best Solution from FTA

1. Prepare a *compositional* scoring function *h*

$$h(F(P_1, \cdots, P_n)) = h(F) + \sum_{i=1}^{n} h(P_i)$$

2. Represent an FTA as a hyper graph, a generalization of graphs

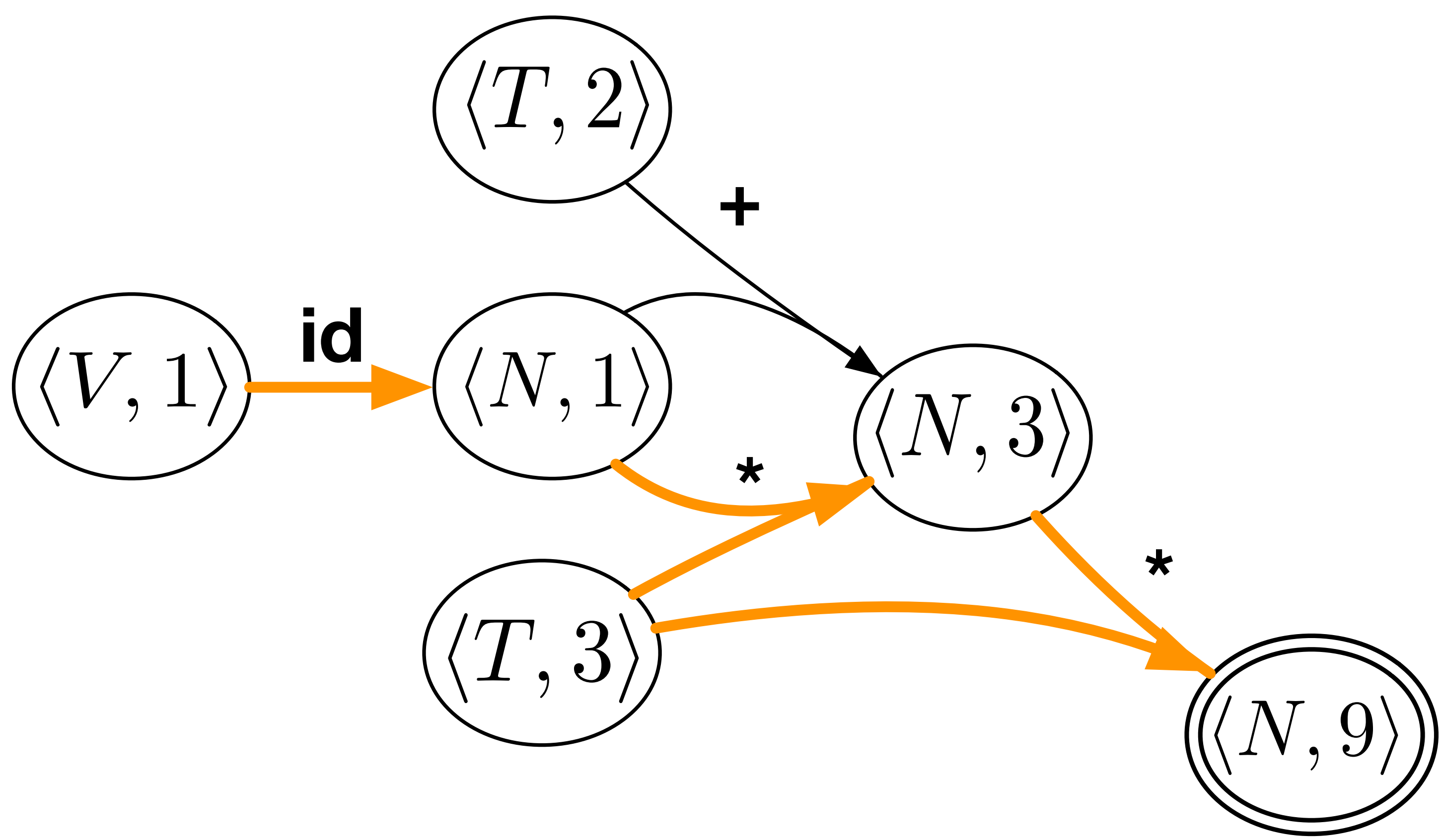
- Nodes: FTA states, edges: FTA transitions (***nodes*** → node)



Gallo et al., Directed Hypergraphs and Applications. Discrete Appl. Math. 42, 2-3 (1993)

Finding a Best Solution from FTA

3. Finding a minimum weight path



Gallo et al., Directed Hypergraphs and Applications. Discrete Appl. Math. 42, 2-3 (1993)

Comparison to Bottom-Up Search

- More size-efficient: sub-terms in the bank are replicated, while in the FTA they are shared
- Can store all terms, not just one representative per class
- Can construct one FTA per example and intersect or construct one FTA from multiple examples simultaneously

Wang et al., Program Synthesis using Abstraction Refinement, POPL 2018
Milter et al., Bottom-Up Synthesis of Recursive Functional Programs, POPL 2022

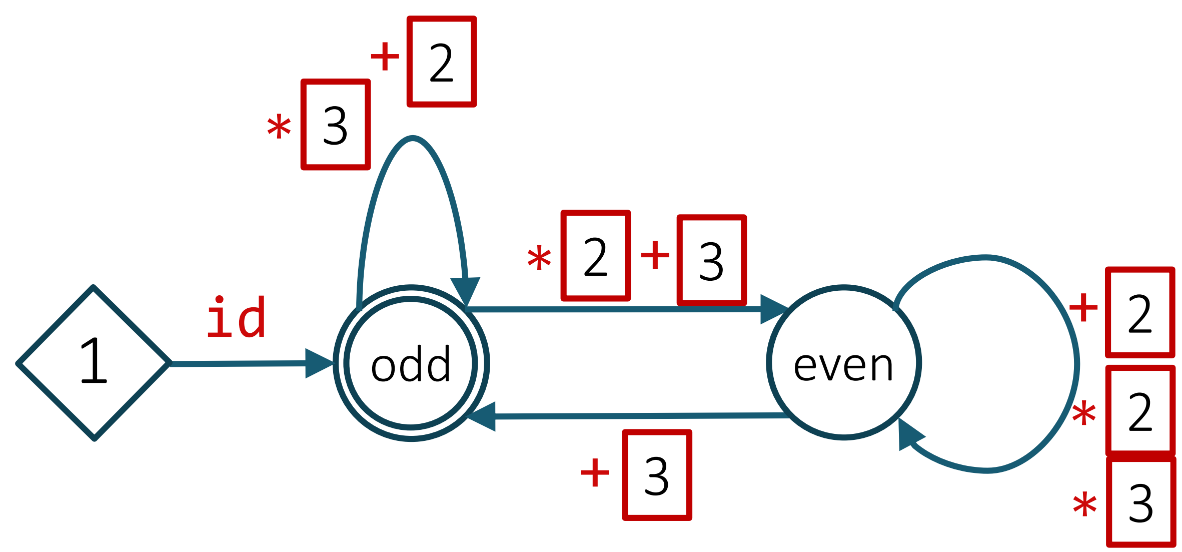
State Explosion in FTA

- Too many states generated while constructing an FTA!
- Idea: one state = one value ⇨ one state = multiple values via abstraction
  - e.g., {2, 4, 6, 8. .. } → even number
- Computations with abstract values — abstract interpretation

FTA + Abstraction

```
N ::= id(V) | N + T | N * T
T ::= 2 | 3
V ::= x
```

1 → 9



In the paper:

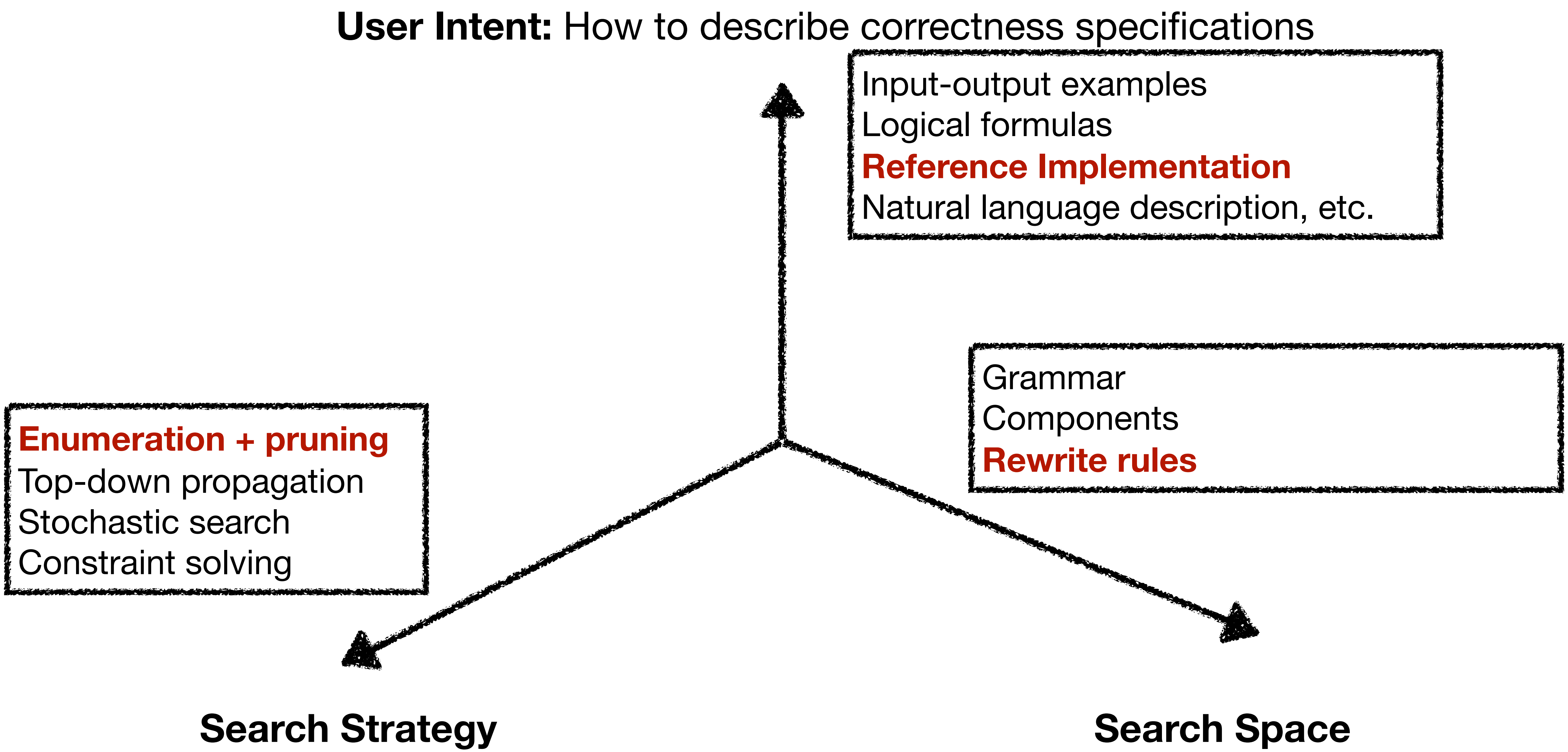
- different abstractions
- refining the abstractions to eliminate spurious paths

Program Synthesis using Abstraction Refinement. POPL'18.

Three Data Structures

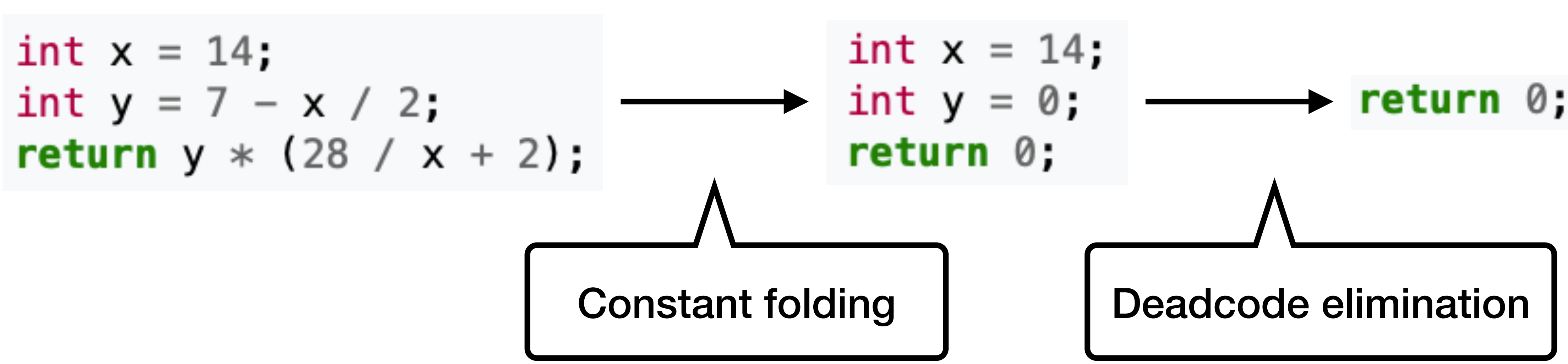
- *Version Space Algebra* (VSA)
  - With Top-down search
- *Finite Tree Automata* (FTA)
  - With Bottom-up search
- *E-graph*
  - With Equality saturation

Equality Saturation



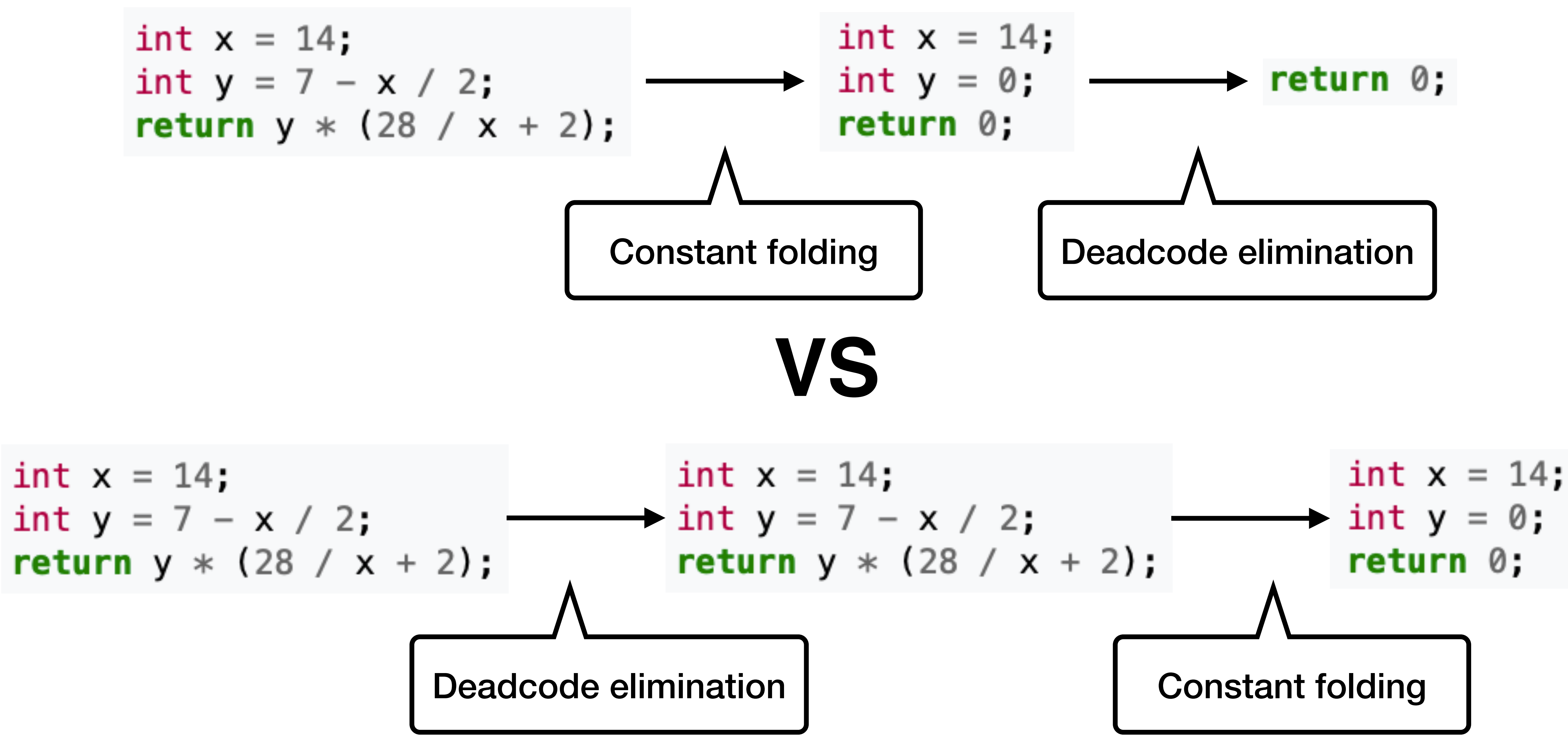
Background: Program Optimization

- Modifying a program to make it work more efficiently
  - Less memory, less power consumption, better speed, etc.
- By applying rewrite (i.e., transformation) rules



Background: Phase Ordering Problem

When multiple rewrite rules are applicable, different orderings of rules may lead to different results

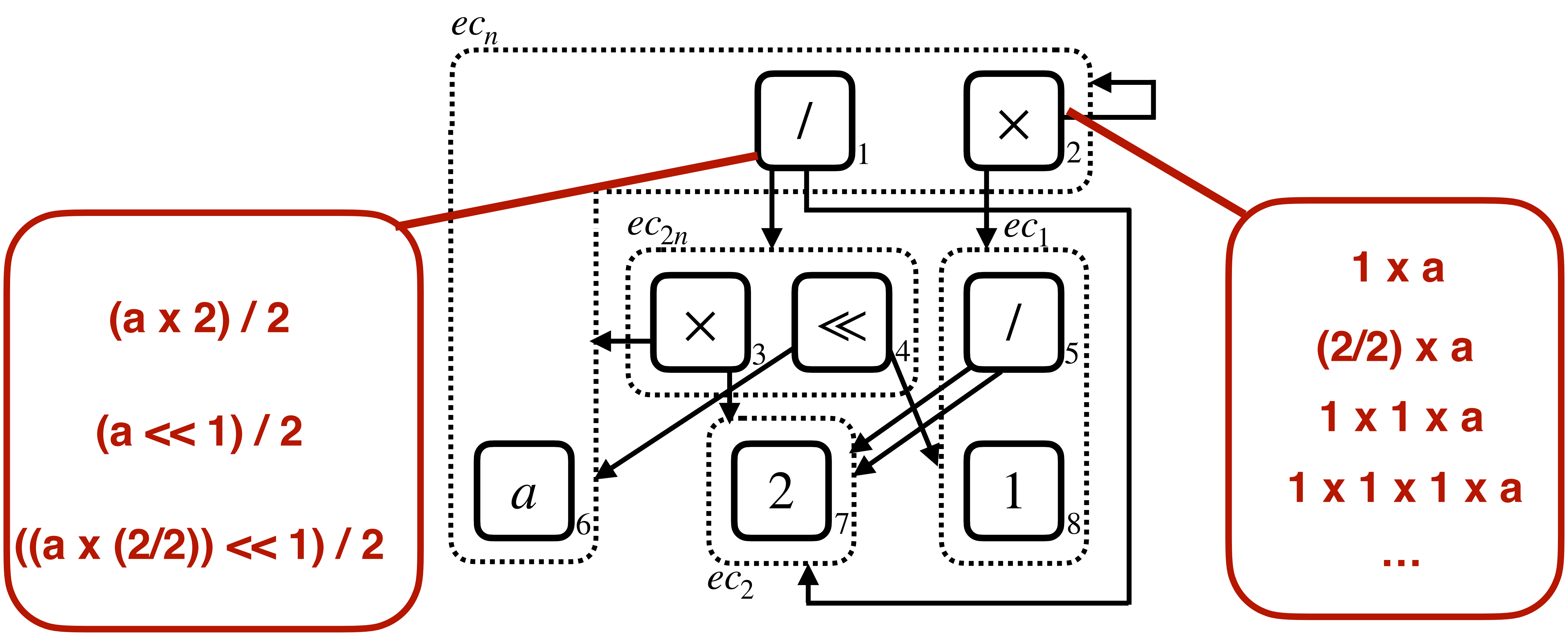


Equality Saturation

- A solution to the phase ordering problem
- Obtain results of all possible orderings and extract the best one among them
- Enabled by ***E-graph***, a very efficient data structure
  - E-graph = e-nodes + e-classes
  - E-classes = set of e-nodes
  - E-node = a node whose children are e-classes

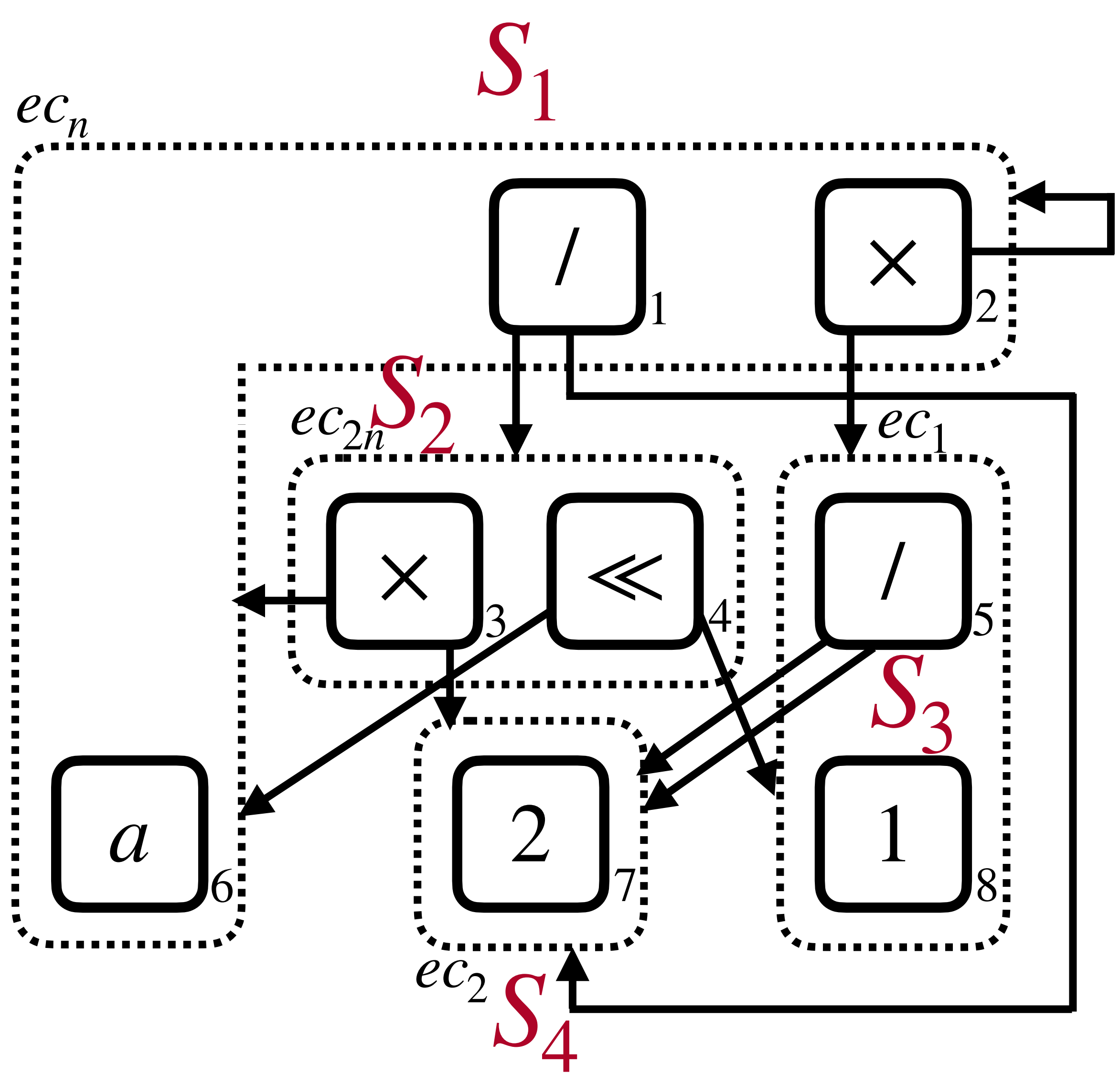
E-graph

- E-node (bold): expressions with sub-expressions represented by children e-classes
- E-class (doted): semantically equivalent e-nodes



E-graph

E-graph ≅ Grammar representing semantically equivalent exprs (E-class ≅ non-terminal, E-node ≅ production rule)



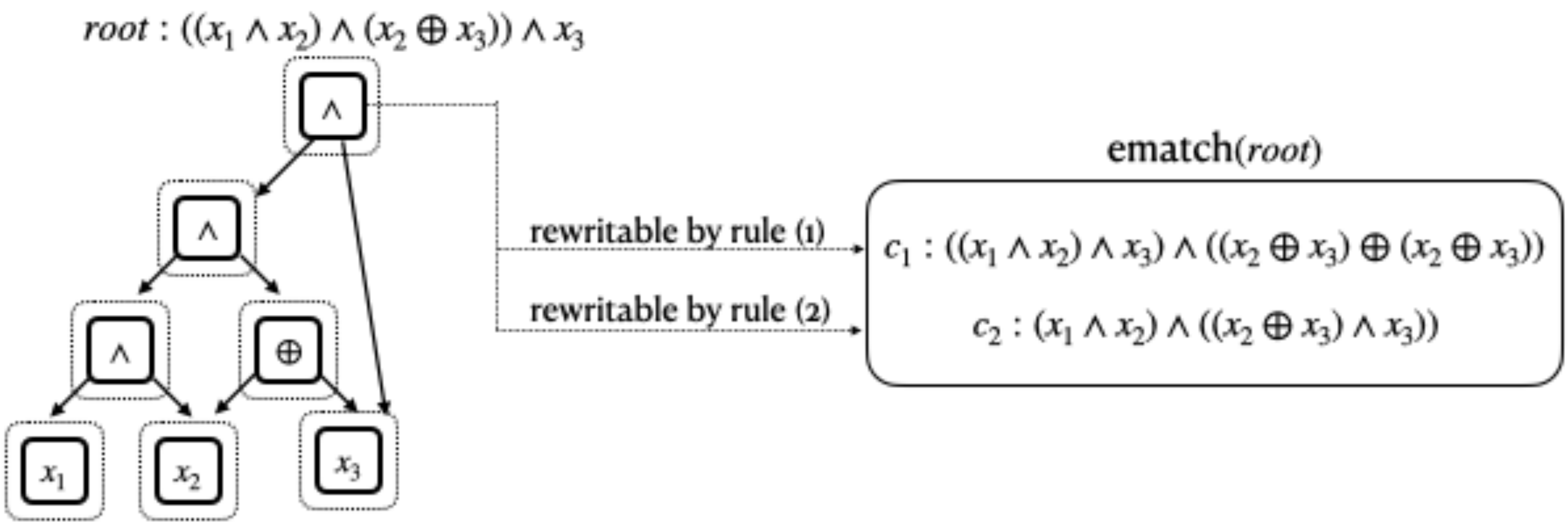
$$
\begin{aligned}
S_1 &\rightarrow S_2 \,/\, S_4 \\
&\mid\ S_1 * S_3 \\
&\mid\ a \\
S_2 &\rightarrow S_1 \ll S_3 \\
&\mid\ S_1 * S_4 \\
S_3 &\rightarrow S_4 \,/\, S_4 \\
&\mid\ 1 \\
S_4 &\rightarrow 2
\end{aligned}
$$

How to Obtain E-graph?

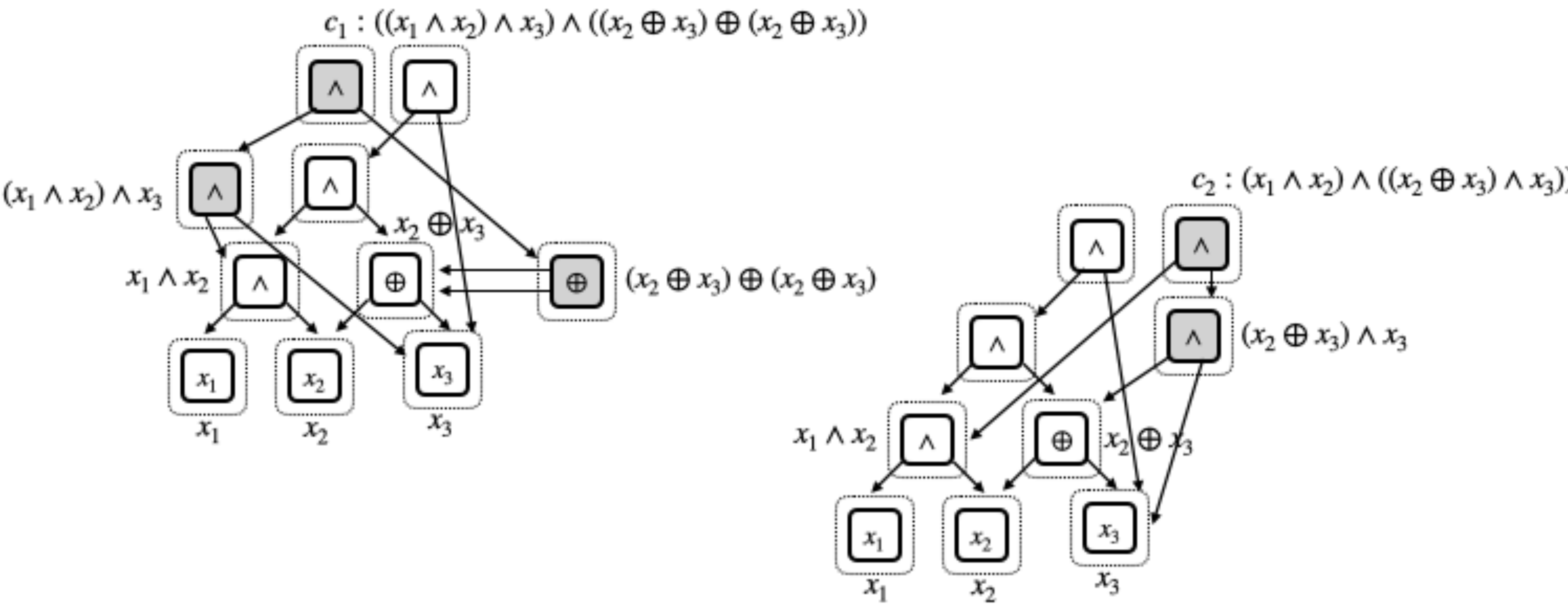
- Goal : given an expression and rewrite rules, find all semantically equivalent rule
- Repeat the following step until saturation
  - **Match**: fi nd e-node to which a rule can be applied
  - **Add**: add a new e-node into the e-graph
  - **Merge**: merge the existing and the new e-nodes into an e-class
- Get the best expression according to a score metric

Match

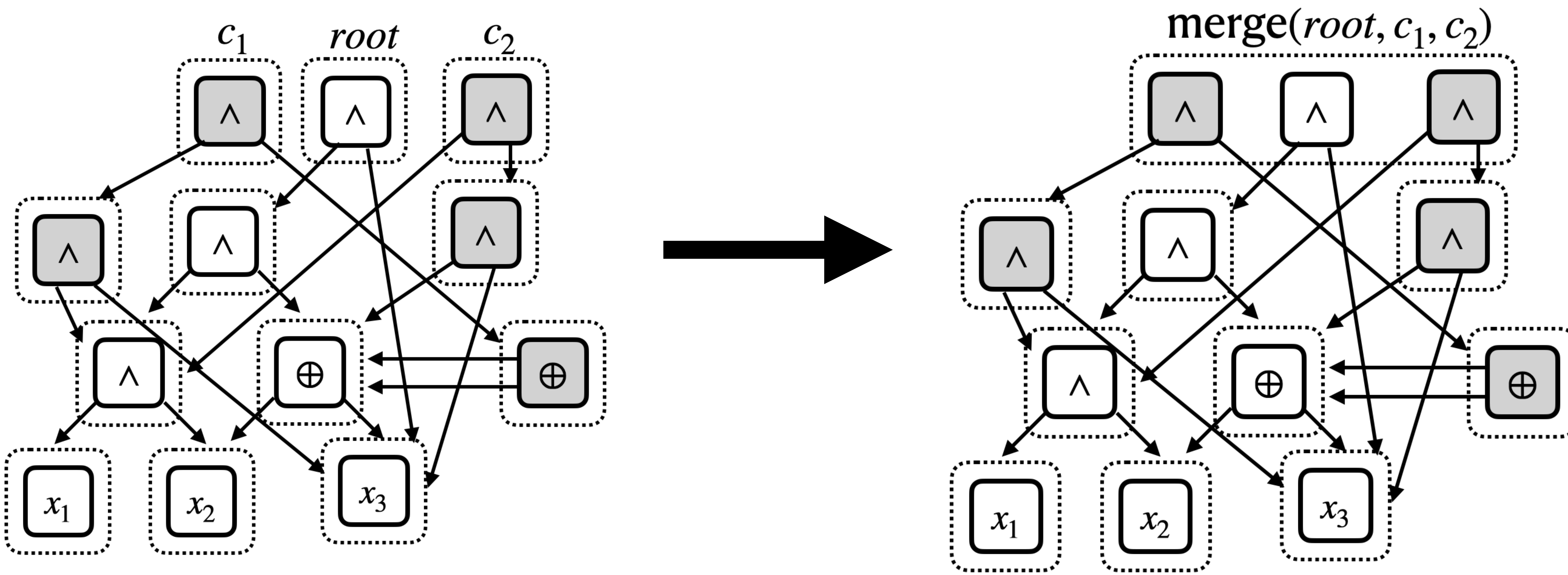
$$
\begin{array}{lrcl}
\text{rule (1)}: & ((v_1 \wedge v_2) \wedge v_3) \wedge v_4 & \rightarrow & ((v_1 \wedge v_2) \wedge v_4) \wedge ((v_2 \oplus v_4) \oplus v_3) \\
\text{rule (2)}: & ((v_1 \wedge v_2) \wedge v_3) \wedge v_4 & \rightarrow & (v_1 \wedge v_2) \wedge (v_3 \wedge v_4) \\
\text{rule (3)}: & (v_1 \oplus v_1) & \rightarrow & 0 \\
\text{rule (4)}: & (v_1 \wedge 0) & \rightarrow & 0
\end{array}
$$



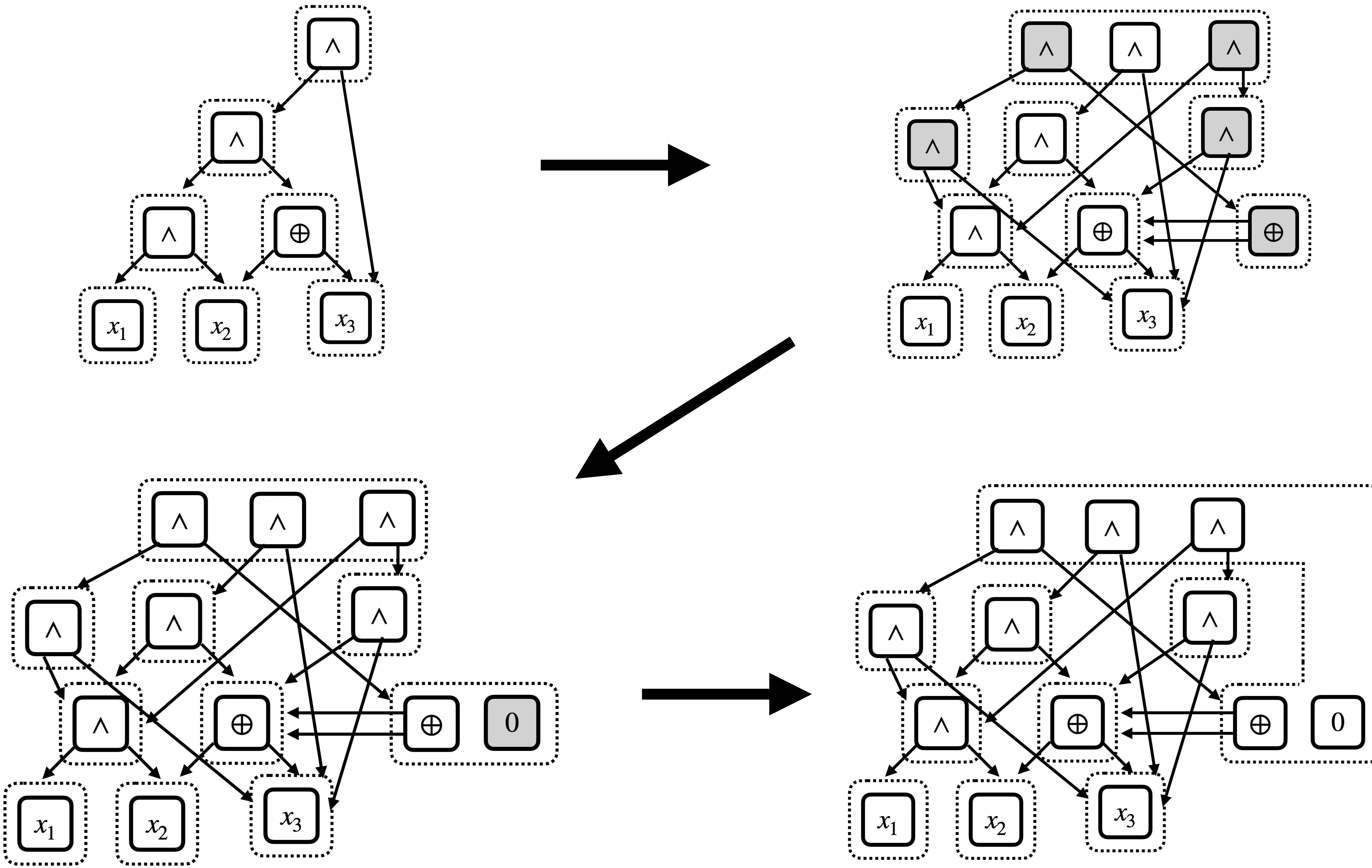
Add


c1 : ((x1 ∧ x2) ∧ x3) ∧ ((x2 ⊕ x3) ⊕ (x2 ⊕ x3))
(x1 ∧ x2) ∧ x3
x2 ⊕ x3
x1 ∧ x2
(x2 ⊕ x3) ⊕ (x2 ⊕ x3)
x1
x2
x3
c2 : (x1 ∧ x2) ∧ ((x2 ⊕ x3) ∧ x3))
(x2 ⊕ x3) ∧ x3
x1 ∧ x2
x2 ⊕ x3
x1
x2
x3

Merge

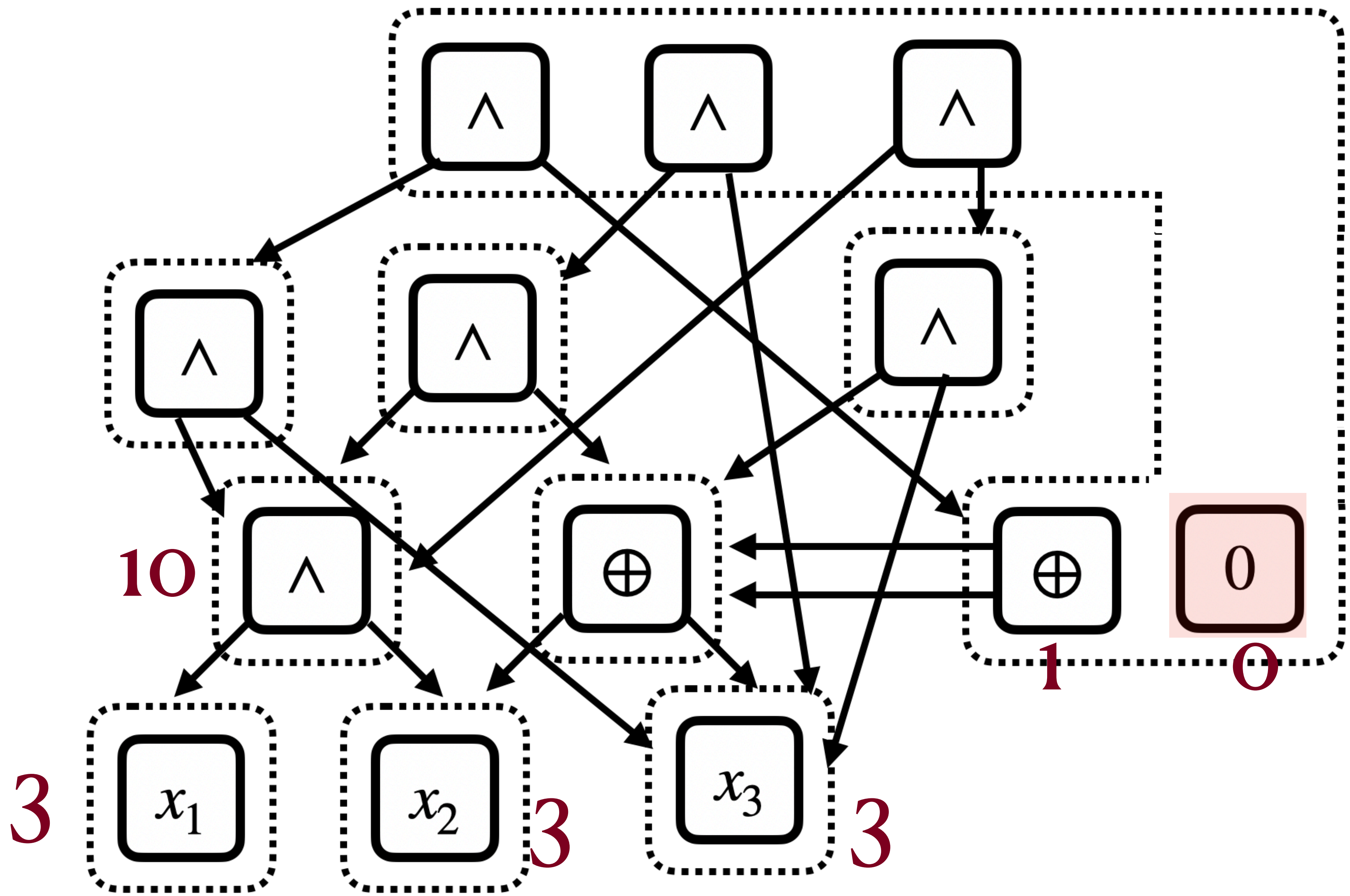


Repeat



Extract

- Extract an expression of the best score
  - e.g., greedy method using scores assigned for each kind of e-node



Egg

- A high-performance library for equality saturation
- https://egraphs-good.github.io/
- Various optimizers based on equality saturation
    - Tensat: deep learning computation graph optimizer
    - SPORES: linear algebra expression optimizer
    - Herbie: floating point expression optimizer
    - …

Summary

- Multiple solutions stored in a size-efficient data structure
  - Version space algebra (VSA), finite tree automata (FTA), e-graph
- Enable to find an optimal solution with an advanced search algorithm (as explicit enumeration for finding an optimal solution is often infeasible)
  - Top-down propagation
  - Abstract interpretation
  - Equality saturation