Lecture 5
Representation-based Search

Nadia Polikarpova

This week

Topics:

- Representation-based search
- Stochastic search

Paper: Rishabh Singh: BlinkFill: Semisupervised Programming By Example for Syntactic String Transformations. VLDB'16

Projects:

- Proposals due Friday
- 1 page, PDF or Google Doc
- Upload to "Proposals" inside the shared Google Folder
- Doc name **must be** TeamN, where N is your team ID

The problem statement

Behavioral constraints = examples

```
[1,4,7,2,0,6,9,2,5]  →  [1,2,4,7,0]
[0] → [0]
[5,1] → [1,5,0]
```

Search strategy?

Enumerative
Representation-based
Stochastic
Constraint-based

Structural constraints = grammar

```
L ::= sort(L)  |  L[N..N]
    |  L + L  |  [N]  |  x
N ::= find(L,N)  |  0
```

Representation-based search

Idea:

1. build a data structure that compactly represents good parts of the search space
2. extract solution from that data structure

Useful when:

- need to return multiple results / rank the results
- can pre-process search space and use for multiple queries

Representations

Version Space Algebra (VSA)

Finite Tree Automaton (FTA)

Type Transition Net (TTN)

Representations

Version Space Algebra (VSA)

Finite Tree Automaton (FTA)

Type Transition Net (TTN)

Mandelin, Xu, Bodik, Kimelman: *Jungloid mining: helping to navigate the API jungle.* PLDI'05

Gvero, Kuncak, Kuraj, Piskac: *Complete completion using types and weights.* PLDI'13

Feng, Martins, Wang, Dillig, Reps: *Component-based synthesis for complex APIs.* POPL'17

Guo, James, Justo, Zhou, Wang, Jhala, Polikarpova: *Synthesis by type-Guided Abstraction Refinement.* POPL'20

Version Space Algebra

Idea: build a data structure that succinctly represents the set of *all* programs consistent with examples

- called a **version space**

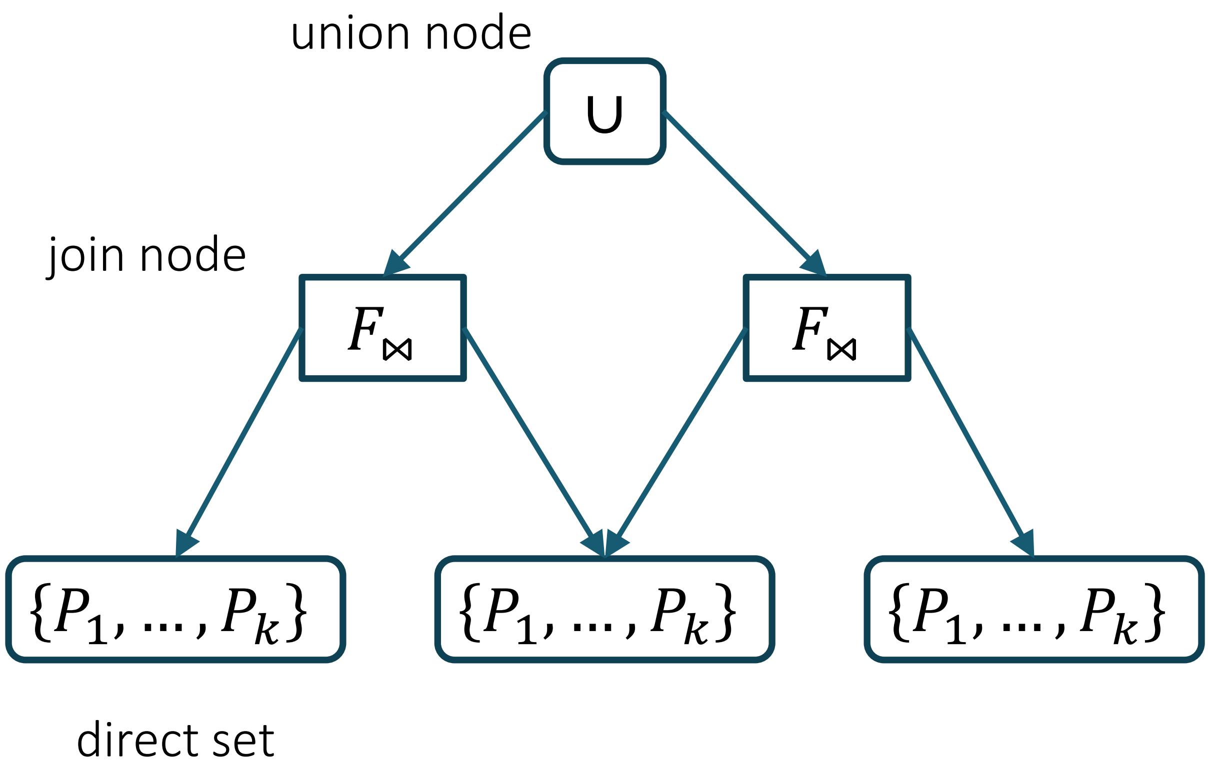
Operations on version spaces:

- `learn <i, o> → VS`
- $VS_1 \cap VS_2 \rightarrow VS$
- `pick VS → program`

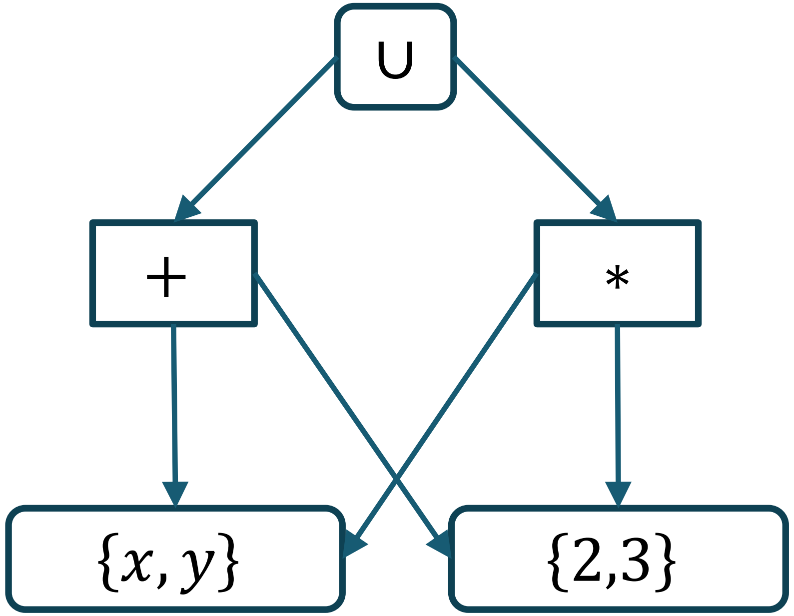
Algorithm:

1. learn a VS for each example
2. intersect them all
3. pick any (or best) program

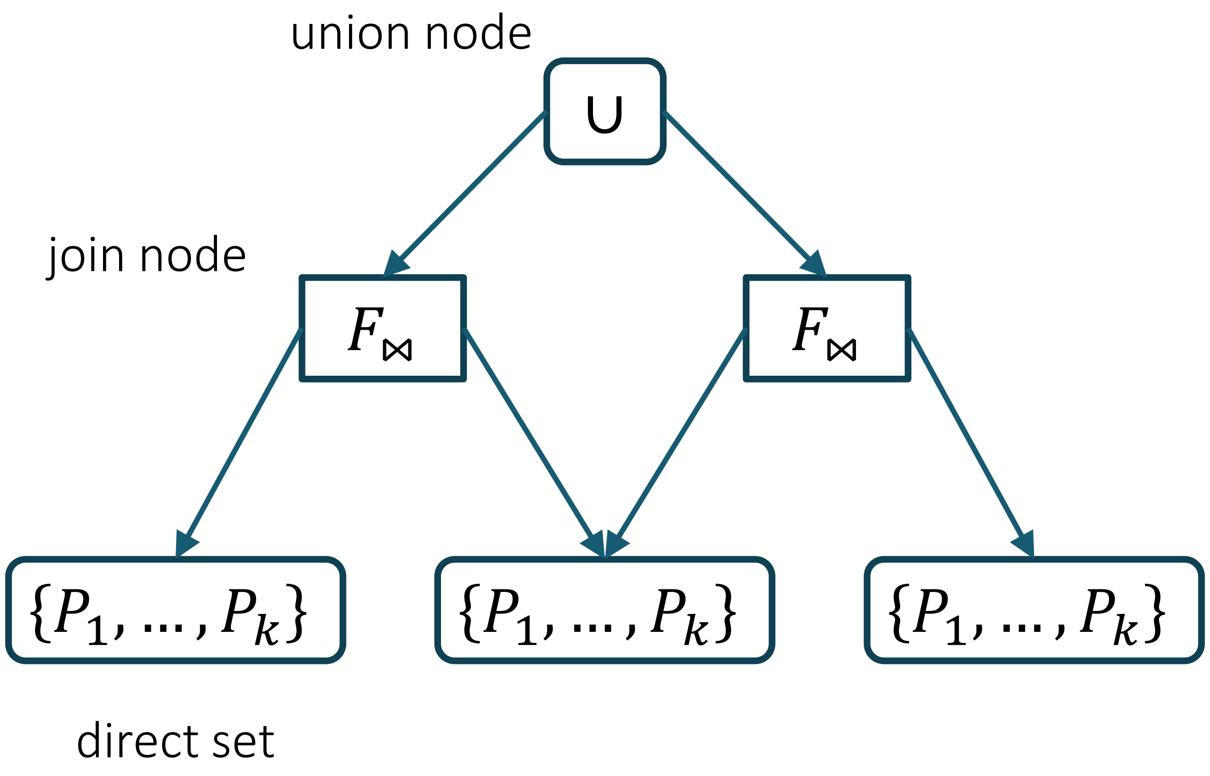
Version Space Algebra



example:



Version Space Algebra



Volume of a VSA
(the number of nodes) $V(VSA)$

Size of a VSA
(the number of programs) $|VSA|$

$$V(VSA) = O(\log|VSA|)$$

Version Space Algebra: history

Mitchell: *Generalization as search*. AI 1982

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

Gulwani: *Automating string processing in spreadsheets using input-output examples.* POPL 2011.

- BlinkFill, FlashExtract, FlashRelate, …
- generalized in the PROSE framework

FlashFill

[Gulwani '11]

Simplified grammar:

E ::= F | concat(F, E)		"Trace" expression
F ::= cstr(*str*) | sub(P, P)		Atomic expression
P ::= cpos(*num*) | pos(R, R)		Position expression
R ::= tokens(T_1, ..., T_n)		Regular expression
T ::= C | C+		Token expression
C ::= ws | digit | alpha | Alpha | $ | ^ | …		

FlashFill: example

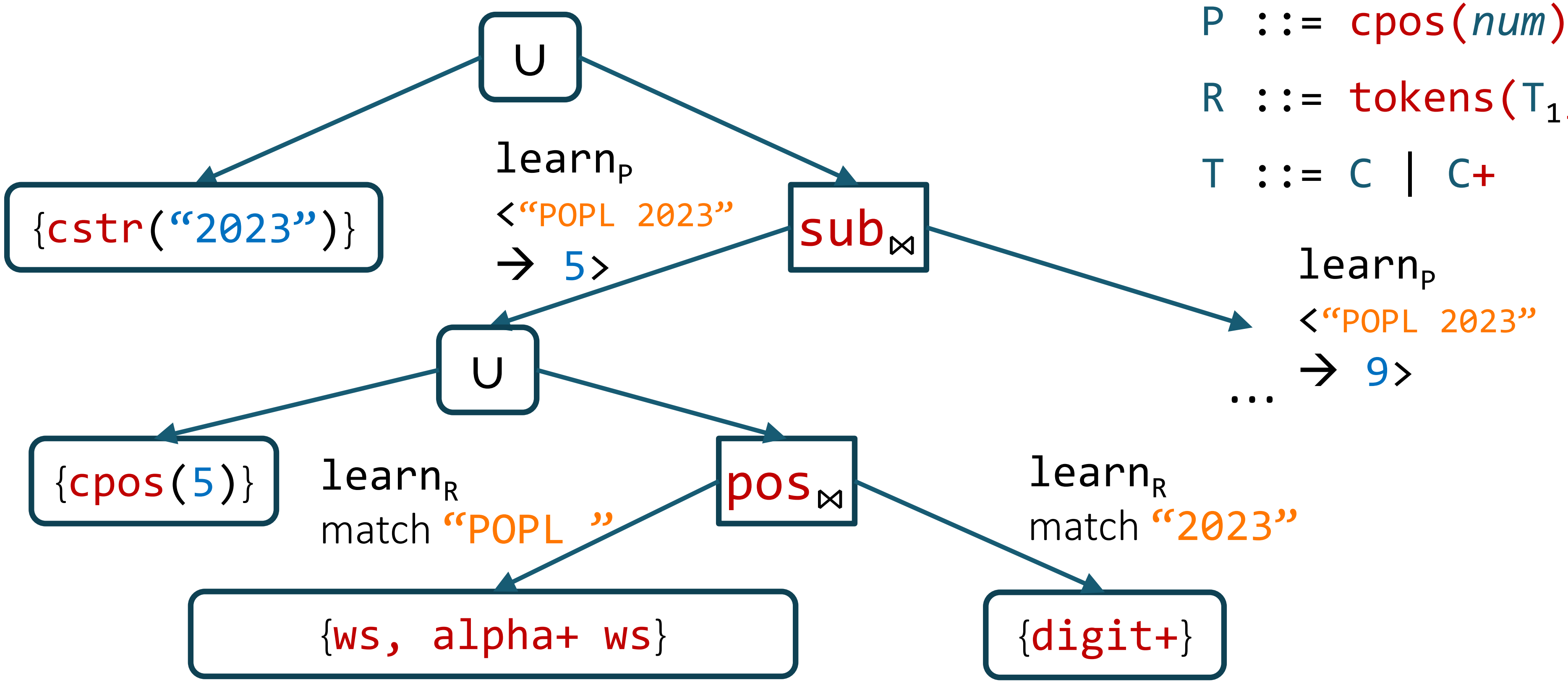
“Hello POPL 2023” → “POPL’2023”
“Goodbye PLDI 2021” → “PLDI’2021”

```
concat(
  sub(pos(ws, Alpha), pos(Alpha, ws)),
  concat(
    cstr(“’”),
    sub(pos(ws, digit), pos(digit, $))))
```

```
E ::= F | concat(F, E)
F ::= cstr(str) | sub(P, P)
P ::= cpos(num) | pos(R, R)
R ::= tokens(T1, ..., Tn)
T ::= C | C+
```

Learning atomic expressions

learn$_F$ <“POPL 2023” → “2023”>



F ::= cstr(*str*) | sub(P$_1$, P$_2$)

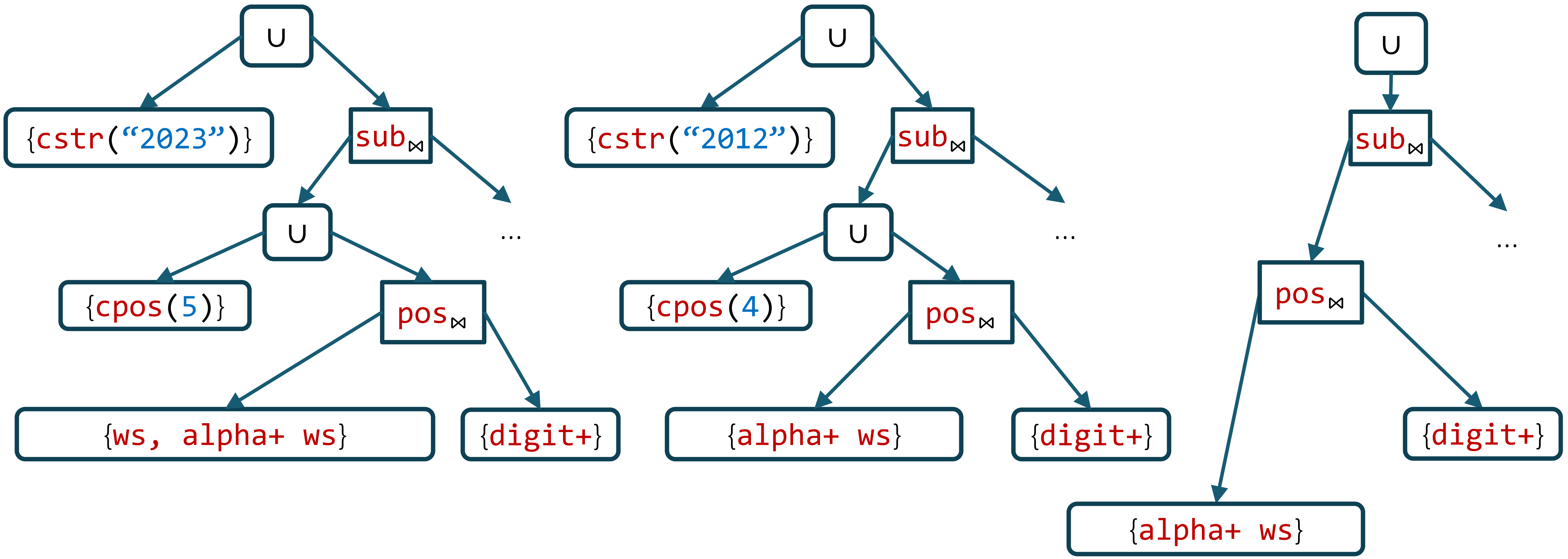
P ::= cpos(*num*) | pos(R$_1$, R$_2$)

R ::= tokens(T$_1$, ..., T$_n$)

T ::= C | C+

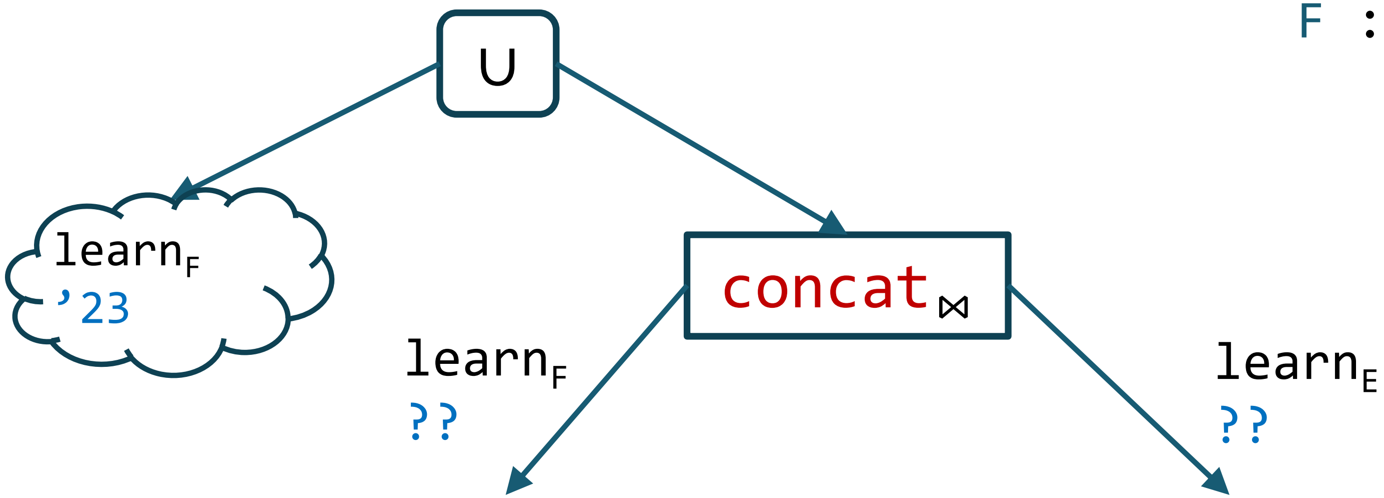
Intersection

“POPL 2023” → “2023” “ FM 2012” → “2012”



Learning trace expressions

learn$_E$ <“POPL 2023” → “’23”>



E ::= F | concat(F, E)

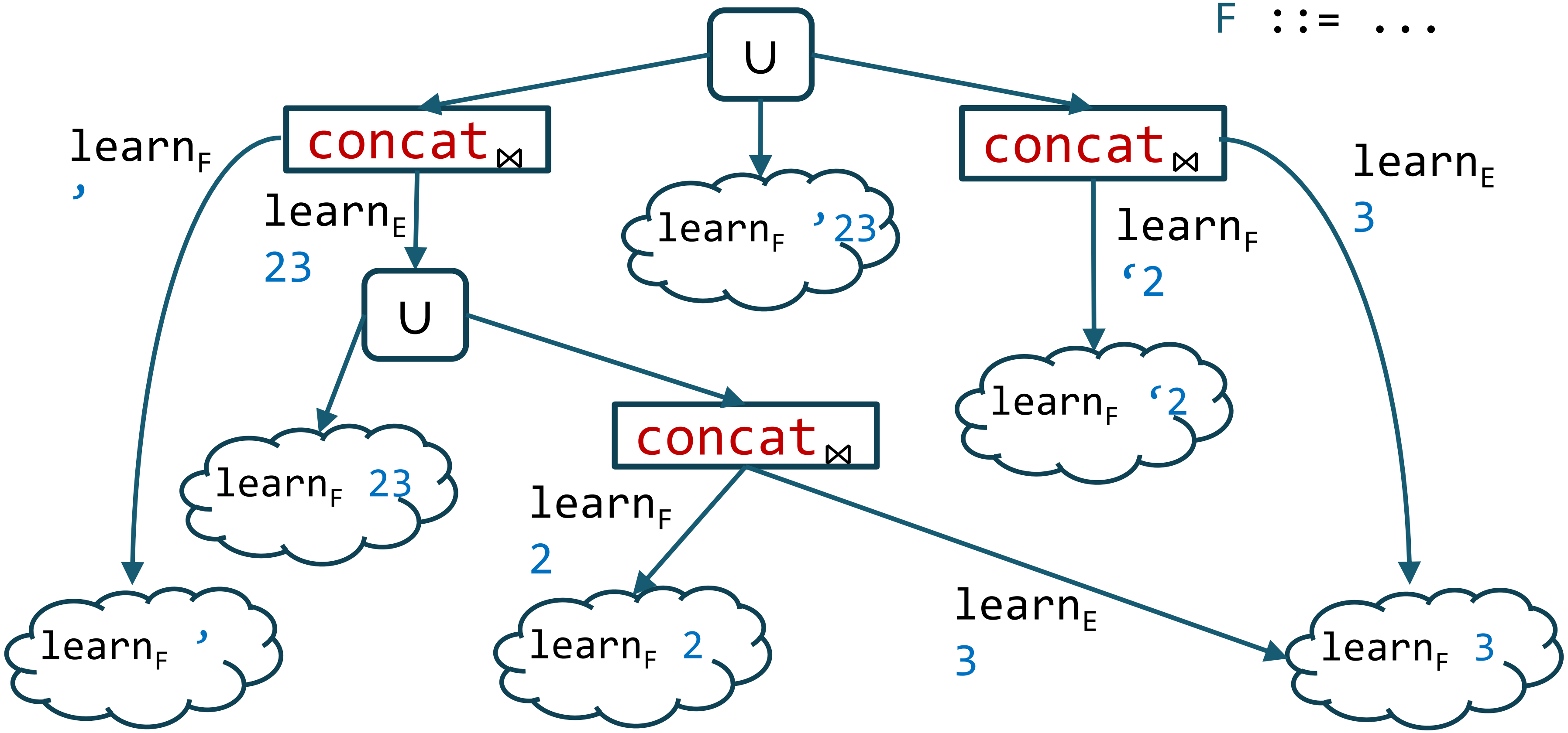
F ::= ...

Learning trace expressions

learn$_E$ <“POPL 2023” → “’23”>

E ::= F | concat(F, E)

F ::= ...



Discussion

Why could we build a finite representation of all expressions?

- Could we do it for this language?

```
E ::= F + F
F ::= k | x
```

$k \in \mathbb{Z}$ + is integer addition

- What about this language?

```
B ::= x | !B | B & B
```

B is a bit-vector

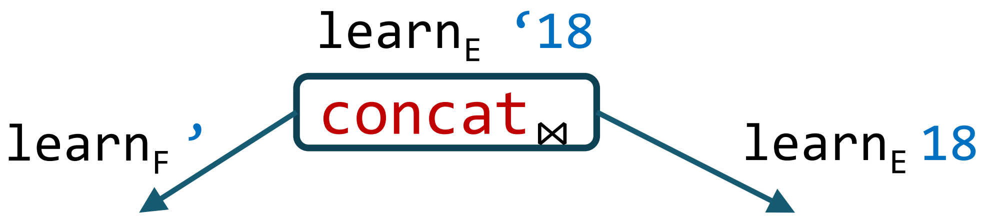
VSA: DSL restrictions

Every operator has a small, easily computable inverse

- Example when an inverse is small but hard to compute?

Every recursive rule generates a strictly smaller subproblem

```
E ::= F | concat(F, E)
```



PROSE

[Polozov, Gulwani ‘15]

Ranking function

```
r :: Program -> Int
```

Grammar

```
A ::= f(B, C) | ...
B ::= g(D, E) | ...
```

Semantics

```
f :: (B, C) -> A
f (b, c) = ...
```

Inverse semantics

```
f⁻¹ :: A -> {(B, C)}
f⁻¹ a = ...
```

PROSE framework

VSA learning algorithm

https://microsoft.github.io/prose/

Discussion

What do VSAs remind you of in the enumerative world?

- VSA ~ top-down search with top-down propagation

How are they different?

- Caching of sub-problems (DAG!)
- Easier to return a ranked list
- Can construct one per example and intersect

Representations

Version Space Algebra (VSA)

Finite Tree Automaton (FTA)

Type Transition Net (TTN)

Example

Grammar

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

Spec

$1 \rightarrow 9$

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, +, *

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

+(<N,1>,<T,2>) → <N,3>

...

Finite Tree Automata

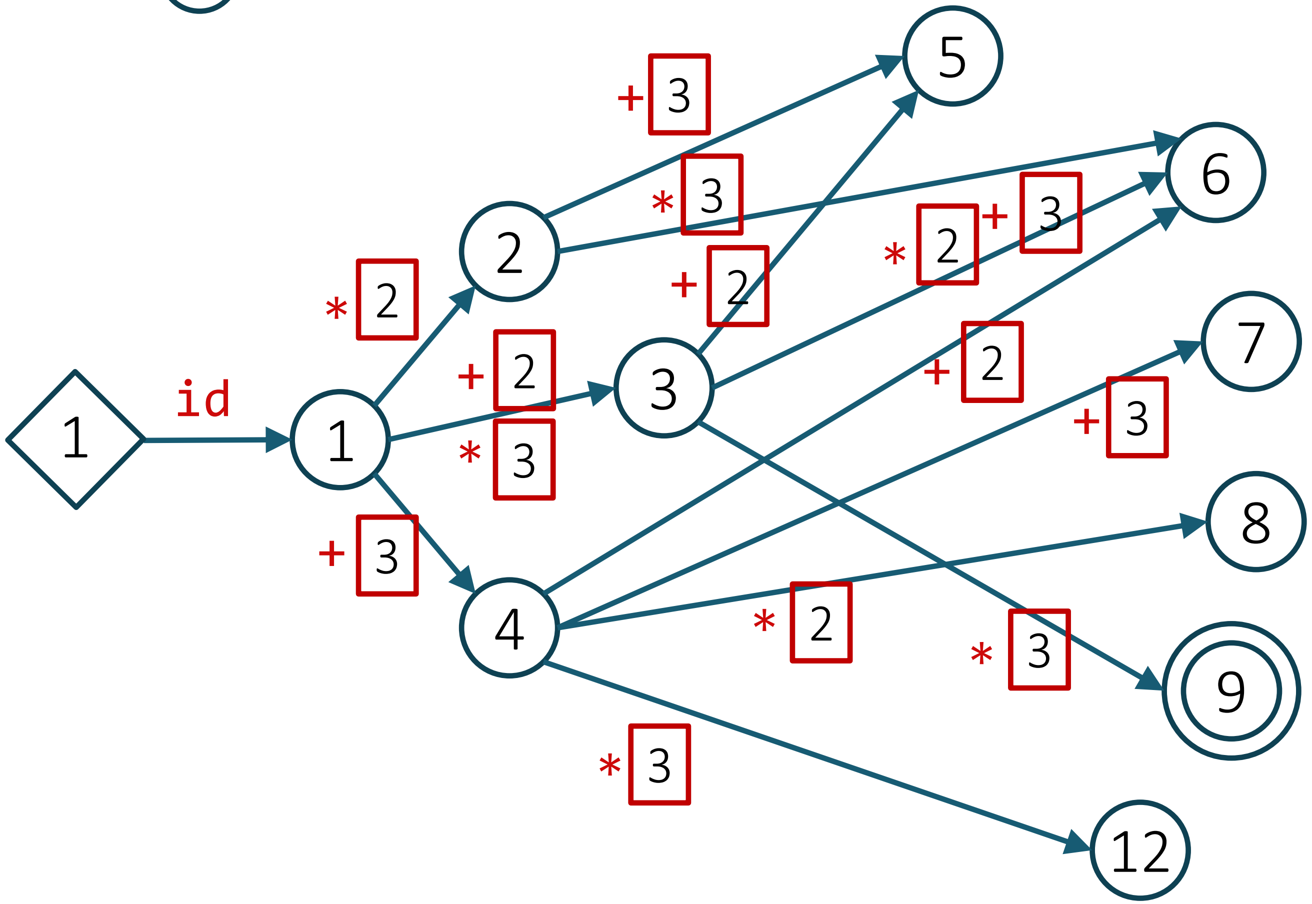
[Wang, Dillig, Singh OOPSLA‘17]

N ::= id(V) | N + T | N * T

T ::= 2 | 3

V ::= x

1 → 9



Discussion

What do FTAs remind you of in the enumerative world?

- FTA ~ bottom-up search with OE

How are they different?

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

Abstract FTA

Challenge: FTA still has too many states

Idea:

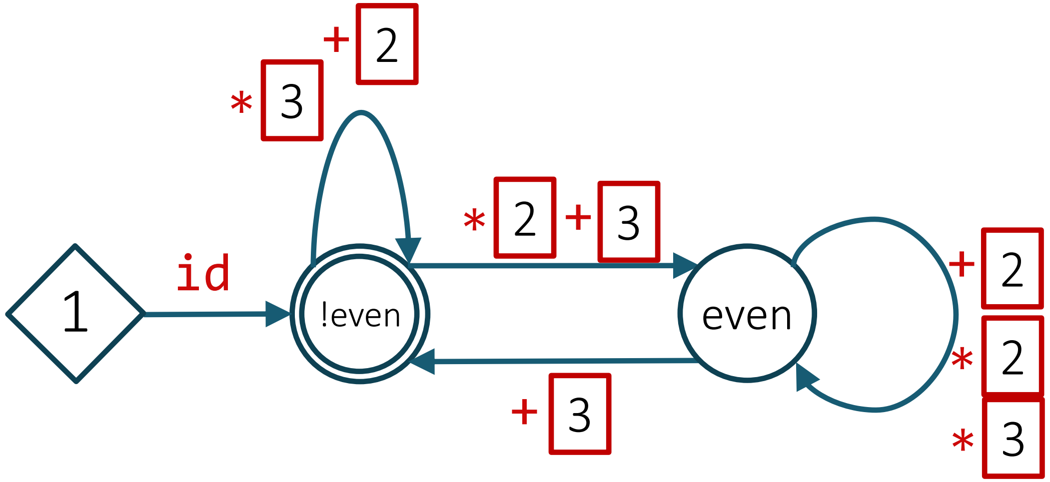
- instead of one state = one value
- we can do one state = set of values (= abstract value)

Abstract FTA

[Wang, Dillig, Singh POPL'18]

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

1 → 9



What now?

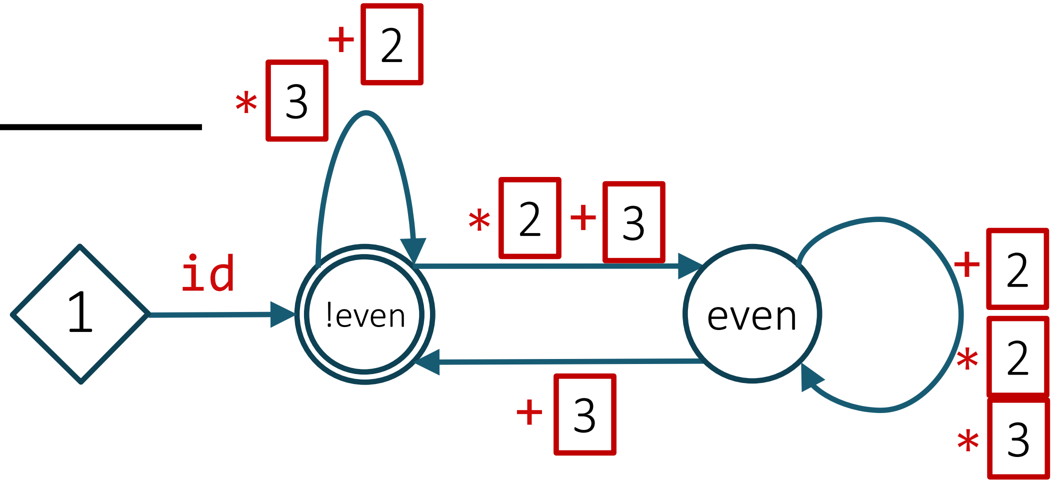
- idea 1: enumerate from reduced space
- idea 2: refine abstraction!

Abstract FTA

N ::= id(V) | N + T | N * T

T ::= 2 | 3

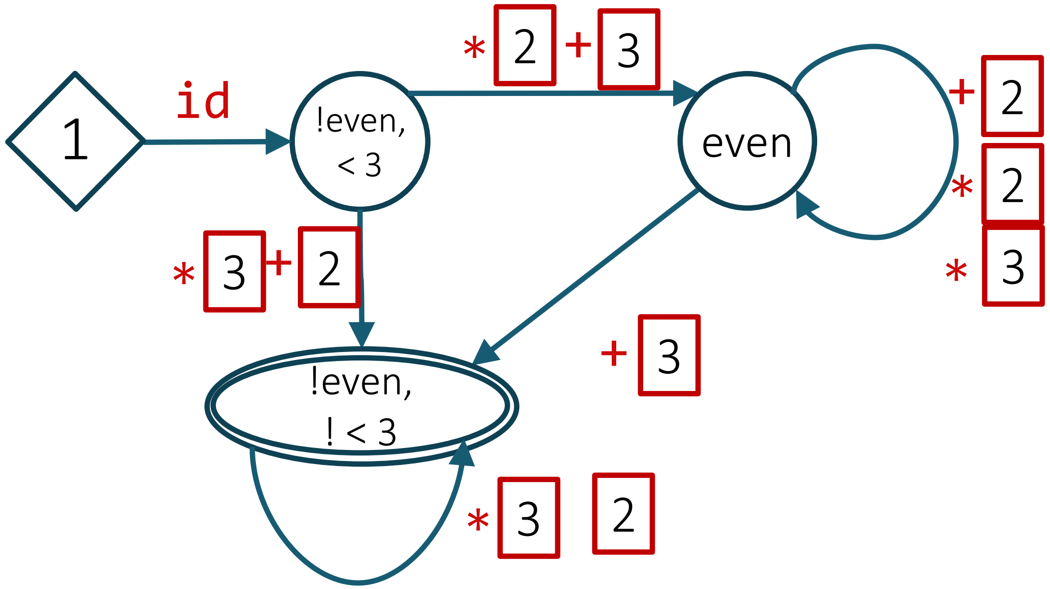
V ::= x



solution: id(x)

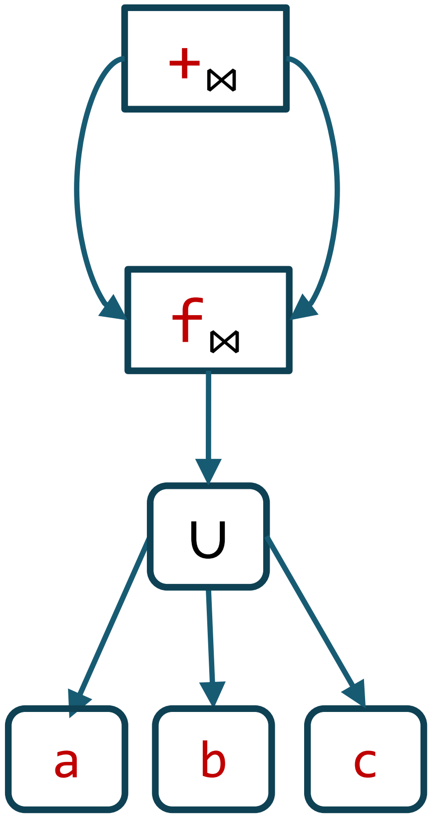
1 → 9

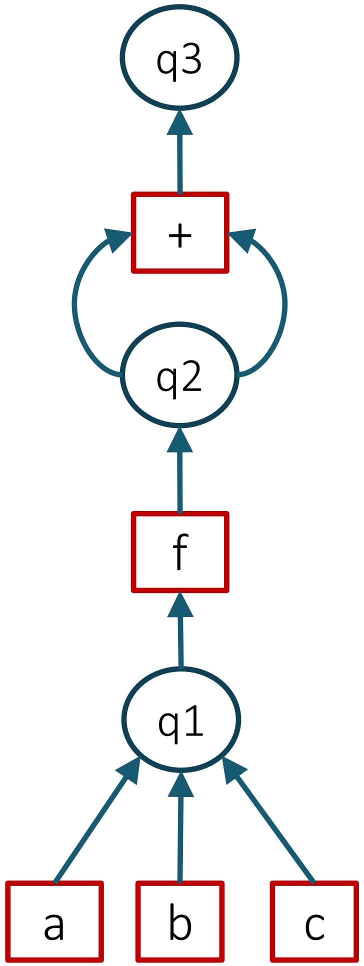
Predicates: {even, < 3, …}

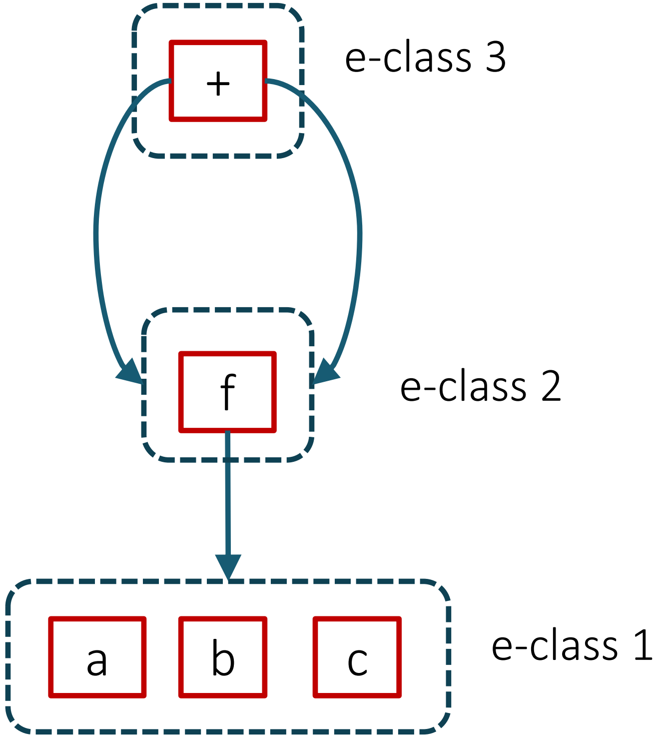


solution: id(x)*3

VSA vs FTA vs E-Graphs







BlinkFill

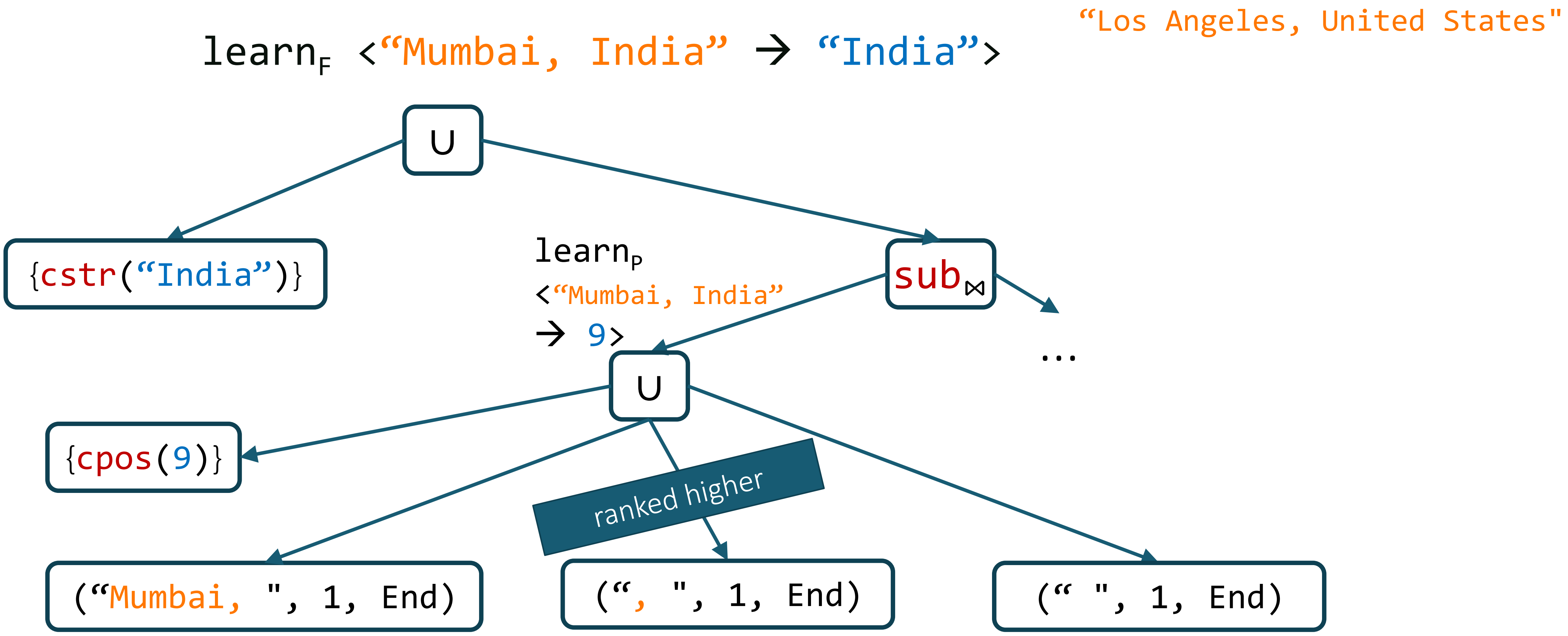
What does BlinkFill use as behavioral constraints? Structural constraints? Search strategy?

- input-output examples; custom string DSL; VSA

What is the main technical insight of BlinkFill wrt FlashFill?

- BlinkFill uses the available inputs (with no outputs) to infer structure (segmentation) common to all inputs
- it uses this structure to shrink the DAG and to rank substring expressions

Example



“Los Angeles, United States"
learn_F <“Mumbai, India” → “India”>
U
{cstr(“India”)}
learn_P
<“Mumbai, India”
→ 9>
sub⋈
…
U
{cpos(9)}
ranked higher
(“Mumbai, ", 1, End)
(“, ", 1, End)
(“ ", 1, End)

BlinkFill

Write a BlinkFill program that satisfies:

- "Programming Language Design and Implementation (PLDI), 2019, Phoenix AZ" -> "PLDI 2019"
- "Principles of Programming Languages (POPL), 2020, New Orleans LA" -> "POPL 2020"

- Between first parentheses and between first and last comma:

```
Concat(SubStr(v1, ("(", 1, End), (")",1, Start)),
       SubStr(v1, (",", 1, End), (",", -1, Start)))
```

BlinkFill

Could we extend the algorithm to support sequences of tokens?

- Each edge of the single-string IDG would have more labels
- Extra edges from 0 and to the last node
- More edges left after intersection (might be a problem, but unclear)
- Need fewer primitive tokens (no need for ProperCase)
- More expressive:
  - "Programming Language Design and Implementation: PLDI 2019" -> "PLDI 2019"
  - “POPL 2020 started on January 22" -> "POPL 2020“
  - `SubStr(v1, (C ws d, 1, Start), (C ws d, 1, End))`

BlinkFill

Strengths? Weaknesses?

- differences between FlashFill and BlinkFill language? which one is more expressive?