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 program space
2. extract solution from that data structure

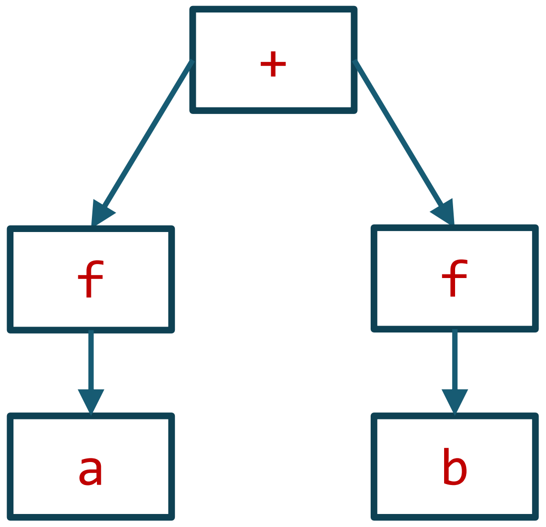
Compact term representation

Consider the space of 9 programs:

```
f(a) + f(a)   f(a) + f(b)   f(a) + f(c)
f(b) + f(a)   f(b) + f(b)   f(b) + f(c)
f(c) + f(a)   f(c) + f(b)   f(c) + f(c)
```

Can we represent this compactly?

- observation 1: same top level structure, independent subterms



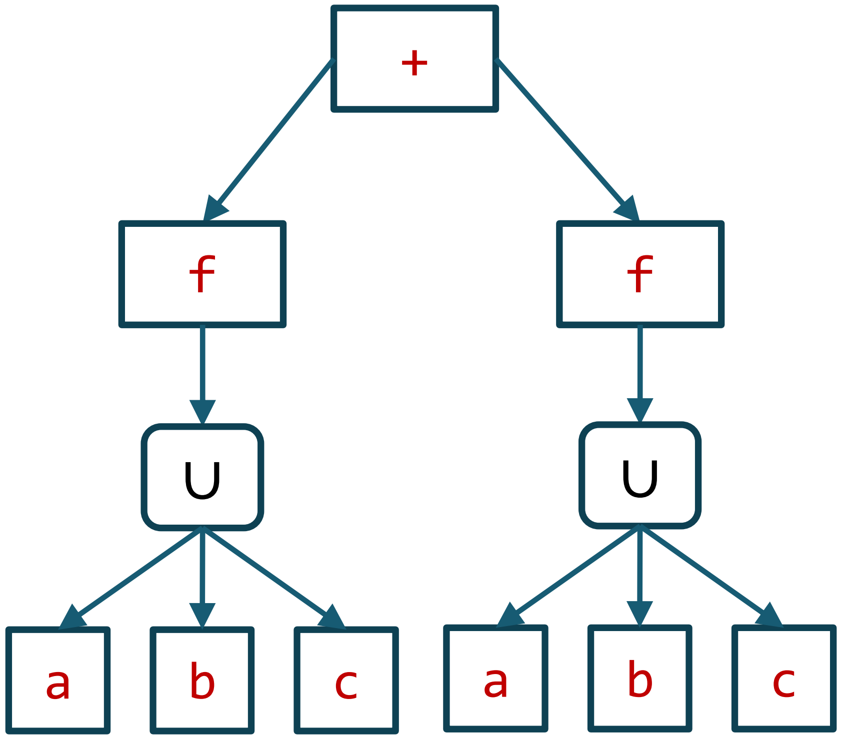
Compact term representation

Consider the space of 9 programs:

```
f(a) + f(a)   f(a) + f(b)   f(a) + f(c)
f(b) + f(a)   f(b) + f(b)   f(b) + f(c)
f(c) + f(a)   f(c) + f(b)   f(c) + f(c)
```

Can we represent this compactly?

- observation 1: same top level structure, independent subterms
- observation 2: shared sub-spaces



Compact term representation

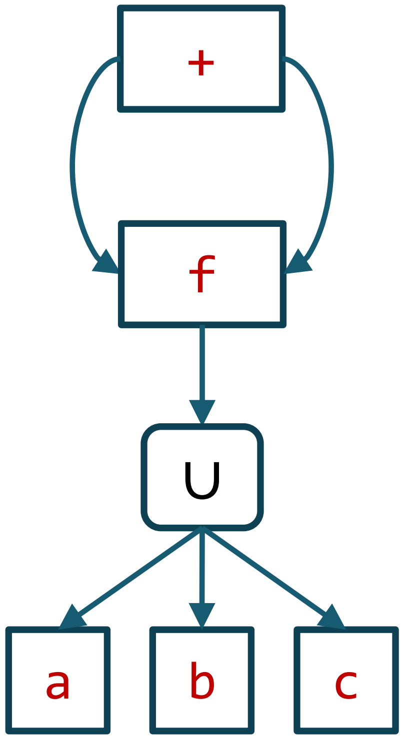
Consider the space of 9 programs:

```
f(a) + f(a)   f(a) + f(b)   f(a) + f(c)
f(b) + f(a)   f(b) + f(b)   f(b) + f(c)
f(c) + f(a)   f(c) + f(b)   f(c) + f(c)
```

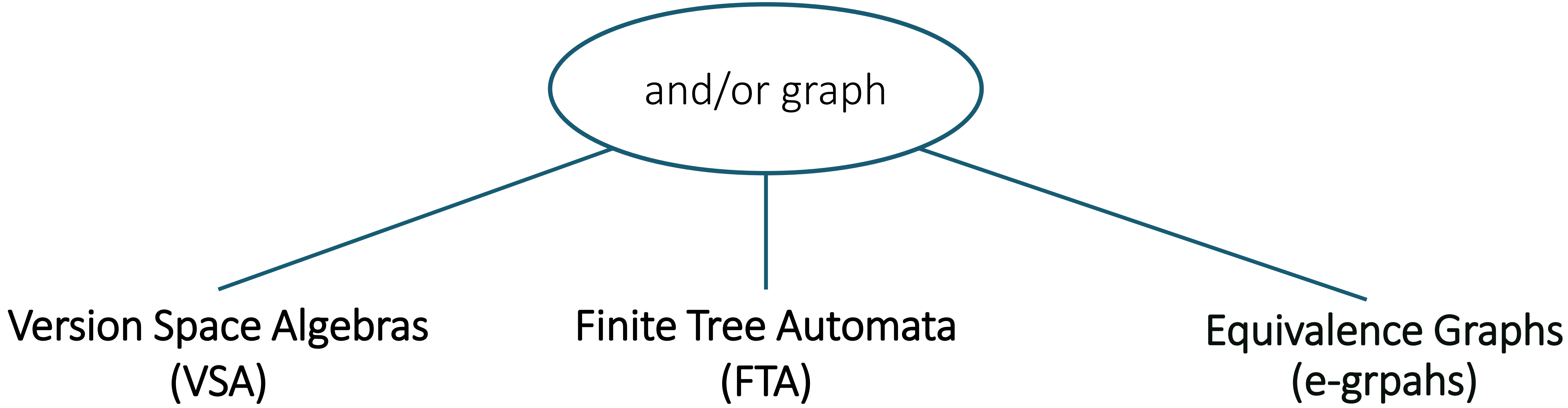
Can we represent this compactly?

- observation 1: same top level structure, independent subterms
- observation 2: shared sub-spaces

Key idea: use an **and-or** graph!



Representation-based search



Version Space Algebra

Idea: build a graph that succinctly represents the space of *all* programs consistent with examples

- called a **version space**

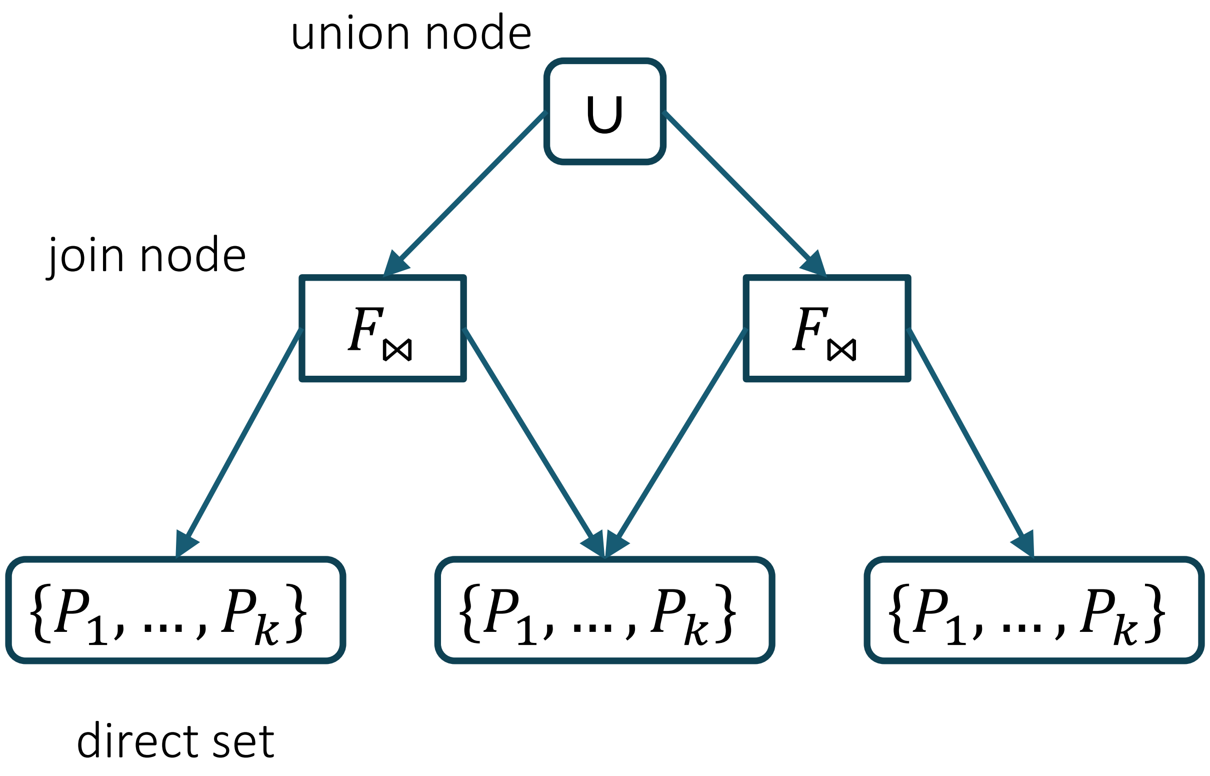
Operations on version spaces:

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

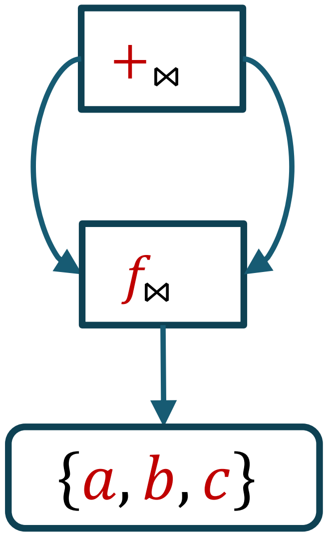
Algorithm:

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

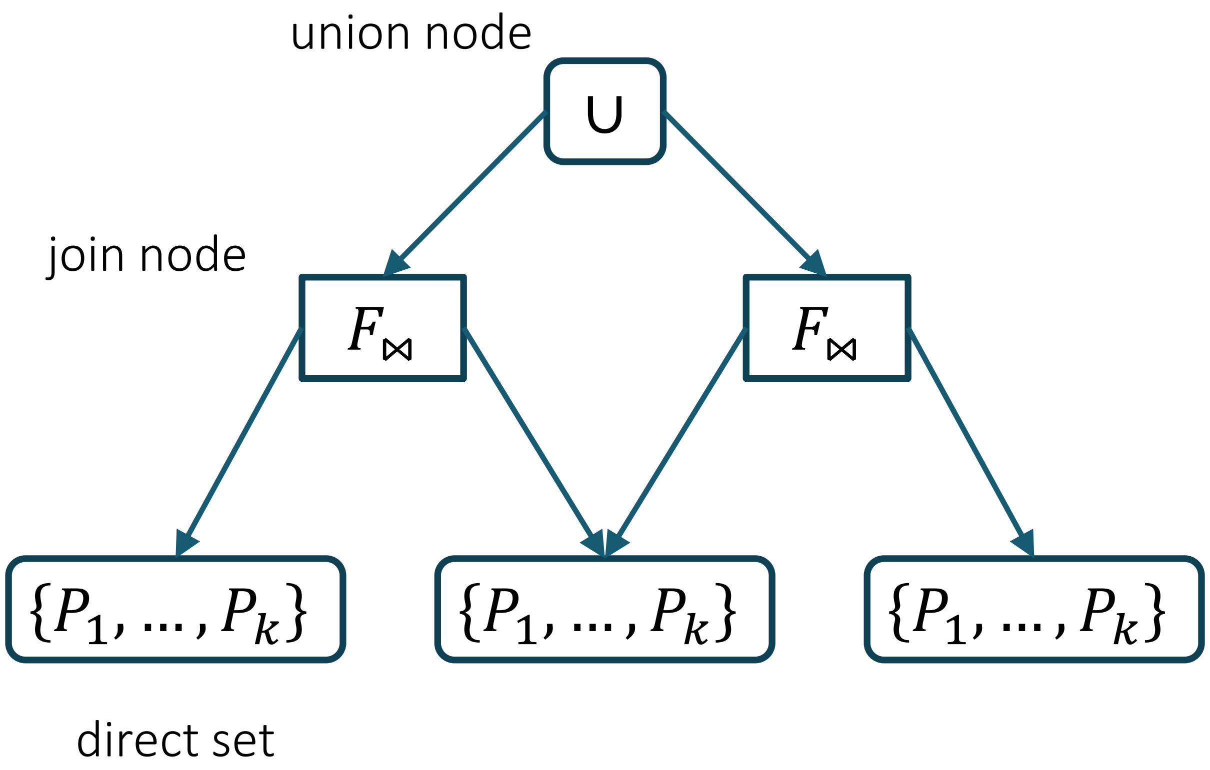
Version Space Algebra



example:



Version Space Algebra



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

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

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

VSA-based search

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.

- Follow-up work: 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

```
 0 1 2 3 4 5 6 7 8 9 …
“Hello POPL 2024” → “POPL’2024”
“Goodbye PLDI 2021” → “PLDI’2021”
```

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

$$
\begin{aligned}
E &::= F \mid \mathtt{concat}(F, E)\\
F &::= \mathtt{cstr}(\mathit{str}) \mid \mathtt{sub}(P, P)\\
P &::= \mathtt{cpos}(\mathit{num}) \mid \mathtt{pos}(R, R)\\
R &::= \mathtt{tokens}(T_1, \ldots, T_n)\\
T &::= C \mid C+
\end{aligned}
$$

VSAs for Flashfill

Recall operations on version spaces:

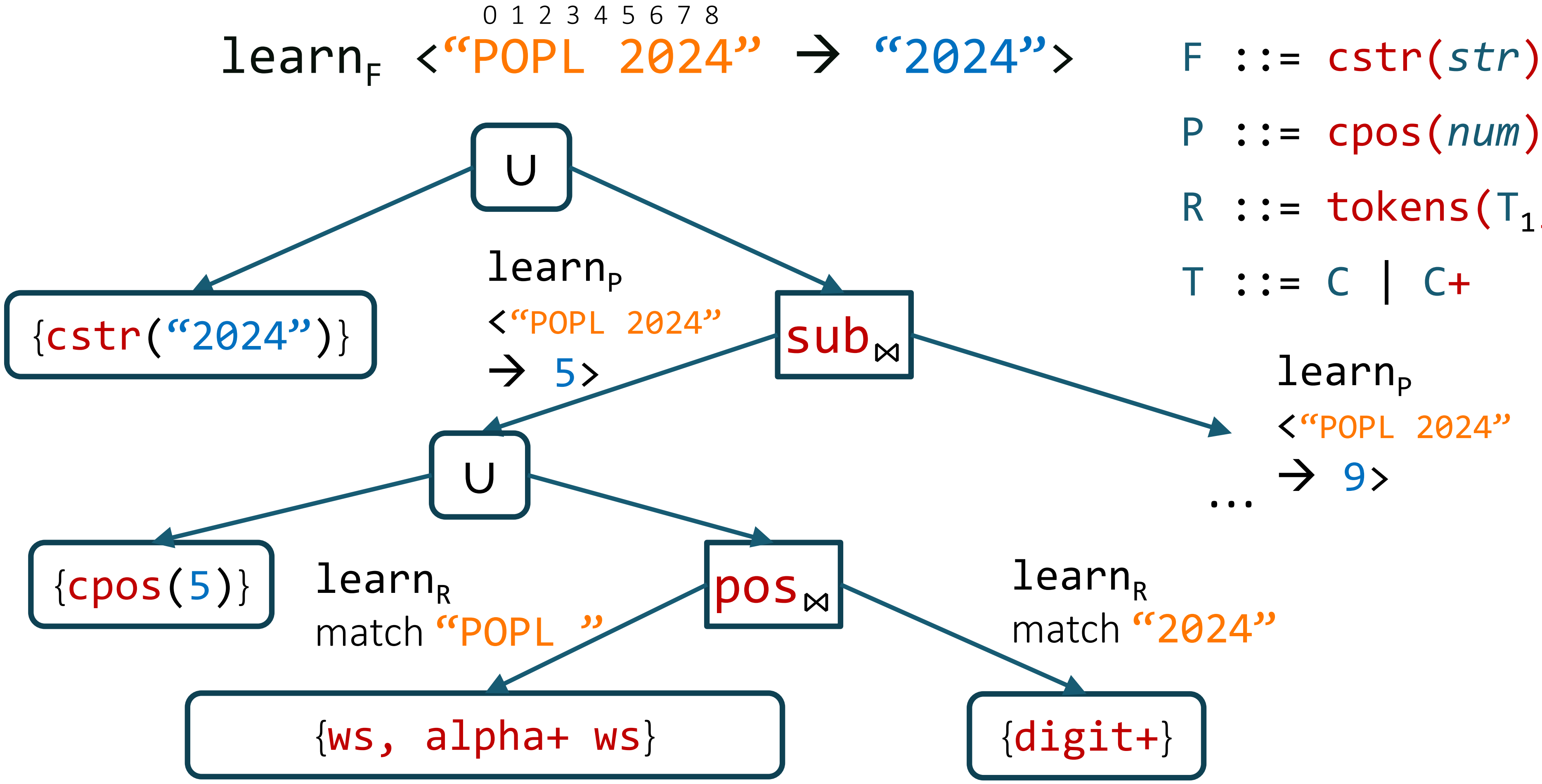
- `learn <i, o> → VS`
- $VS_1 \cap VS_2 \rightarrow VS$
- `extract VS → program`

How do we implement `learn`?

- define $\mathtt{learn}_N$ `<i, o>` for every non-terminal `N`
- build VS top-down, propagating `<i, o>` the example

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

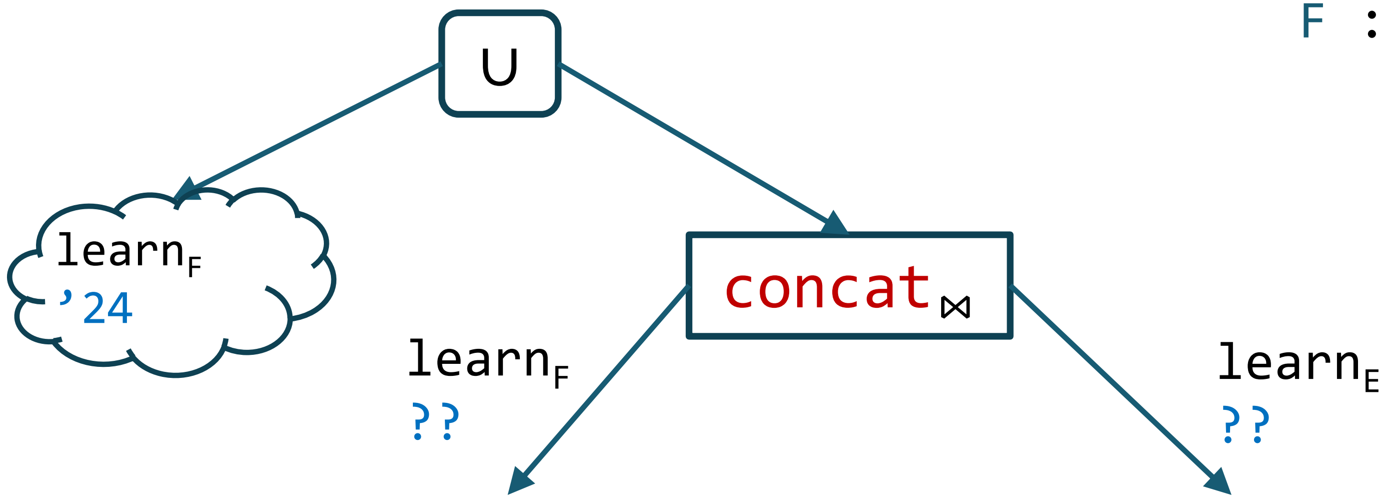
Learning atomic expressions




```
F ::= cstr(str) | sub(P₁, P₂)
P ::= cpos(num) | pos(R₁, R₂)
R ::= tokens(T₁, ..., Tₙ)
T ::= C | C+
```

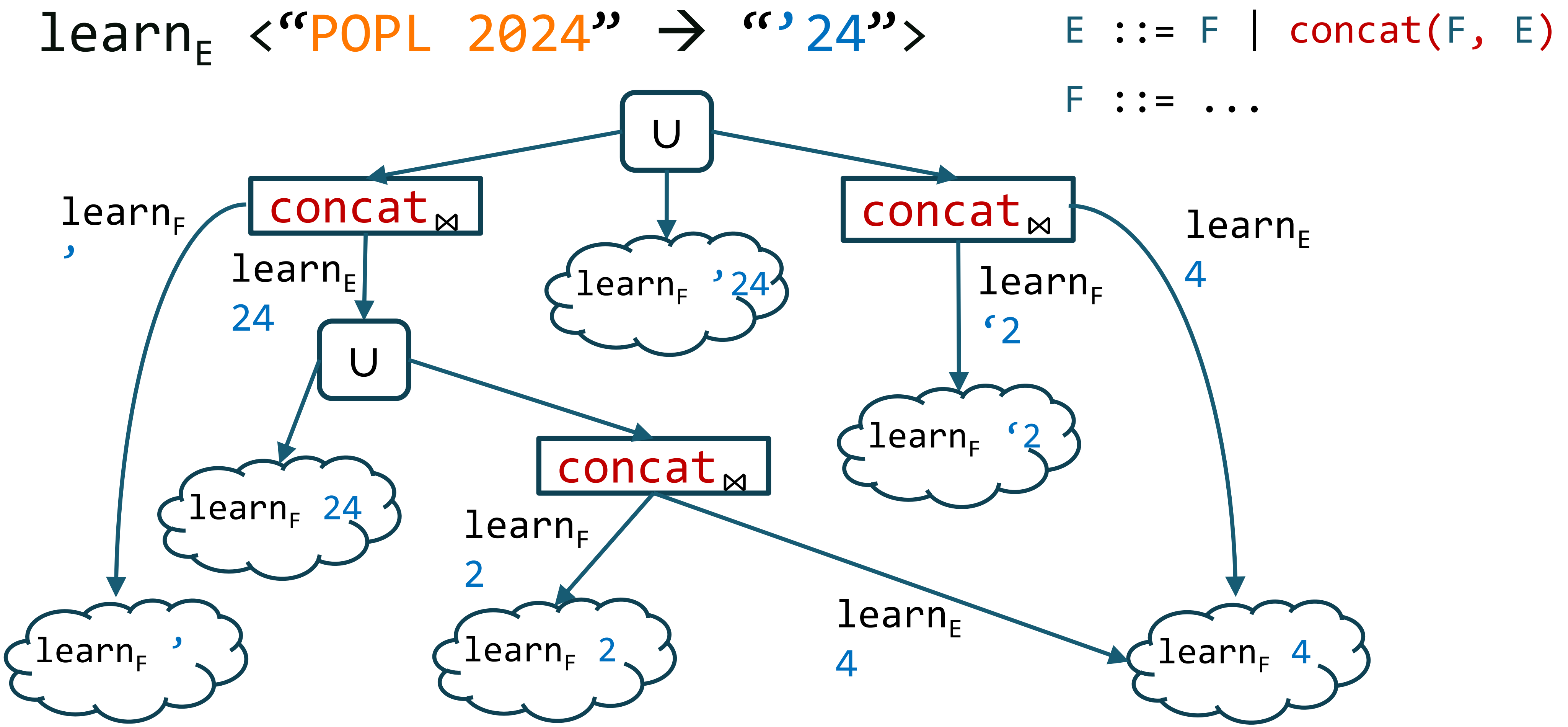
Learning trace expressions

$\texttt{learn}_E$ <"POPL 2024" → "'24">




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

Learning trace expressions



VSAs for Flashfill

Recall operations on version spaces:

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

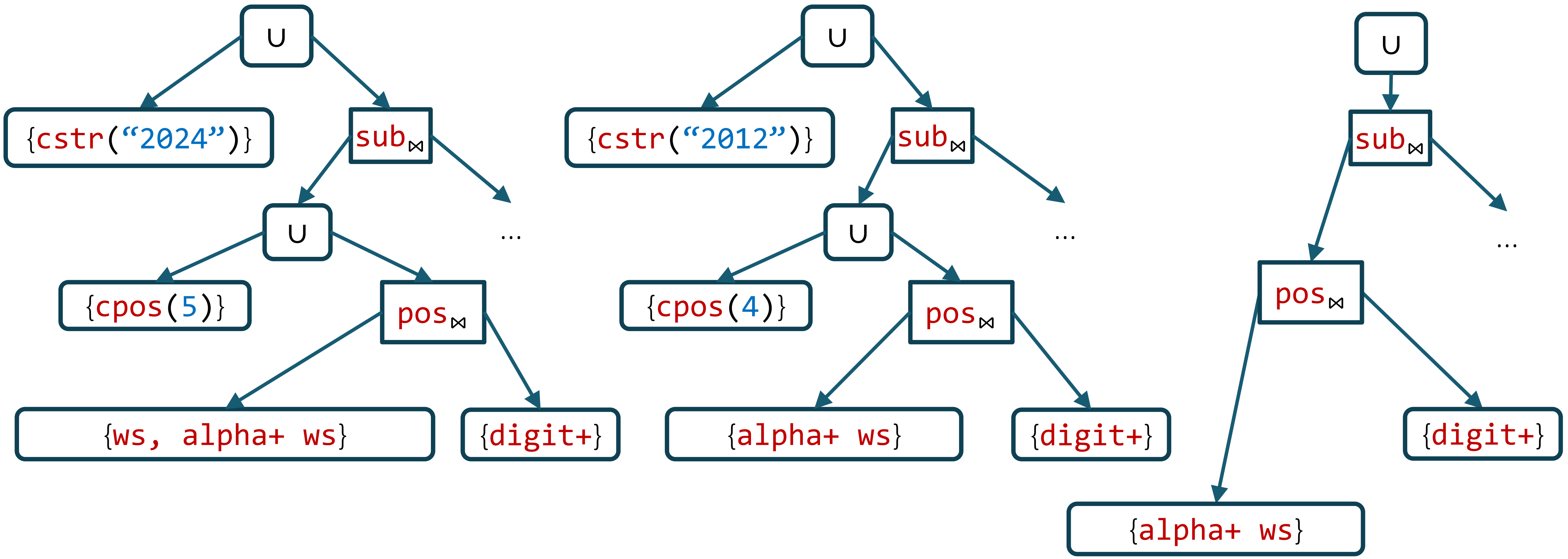
How do we implement intersection?

- top-down
- union: intersect all pairs of children
- join: intersect children pairwise

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

Intersection

“POPL 2024” → “2024” “ 3M 2012” → “2012”



VSAs for Flashfill

Recall operations on version spaces:

- `learn <i, o> → VS`
- $\texttt{VS}_1 \cap \texttt{VS}_2 \rightarrow \texttt{VS}$
- `extract VS → program`

How do we implement extract?

- any program: just pick one child from every union
- best program: shortest path in a DAG

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

Discussion

What do VSAs remind you of in the enumerative world?

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

How are they different?

- Caching of sub-problems (DAG!)
- Can construct one per example and intersect
- This allows it to guess arbitrary constants!

Discussion

Why could we build a finite representation of all solutions?

- Could we do it for this language?

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

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

- What about this language?

```
E ::= E + 1 | x
```

DSL restrictions: efficiently invertible

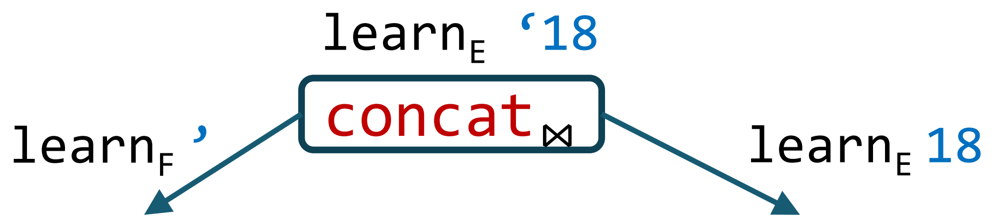
Every operator has a small, easily computable inverse

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

The space of sub-specs is finite

- either non-recursive grammar
- or finite space of values for the recursive non-terminal (e.g. bit-vectors)
- or every recursive production generates a strictly smaller spec

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



PROSE

[Polozov, Gulwani '15]

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 = ...
```

Ranking function

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

PROSE framework

VSA learning algorithm

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

Representation-based search

and/or graph

Version Space Algebras (VSA)

Finite Tree Automata (FTA)

Equivalence Graphs (e-grpahs)

ops: learn-1, intersect, extract
DSL: efficiently invertible
similar to: top-down prop, but can infer constants

Finite Tree Automata

states

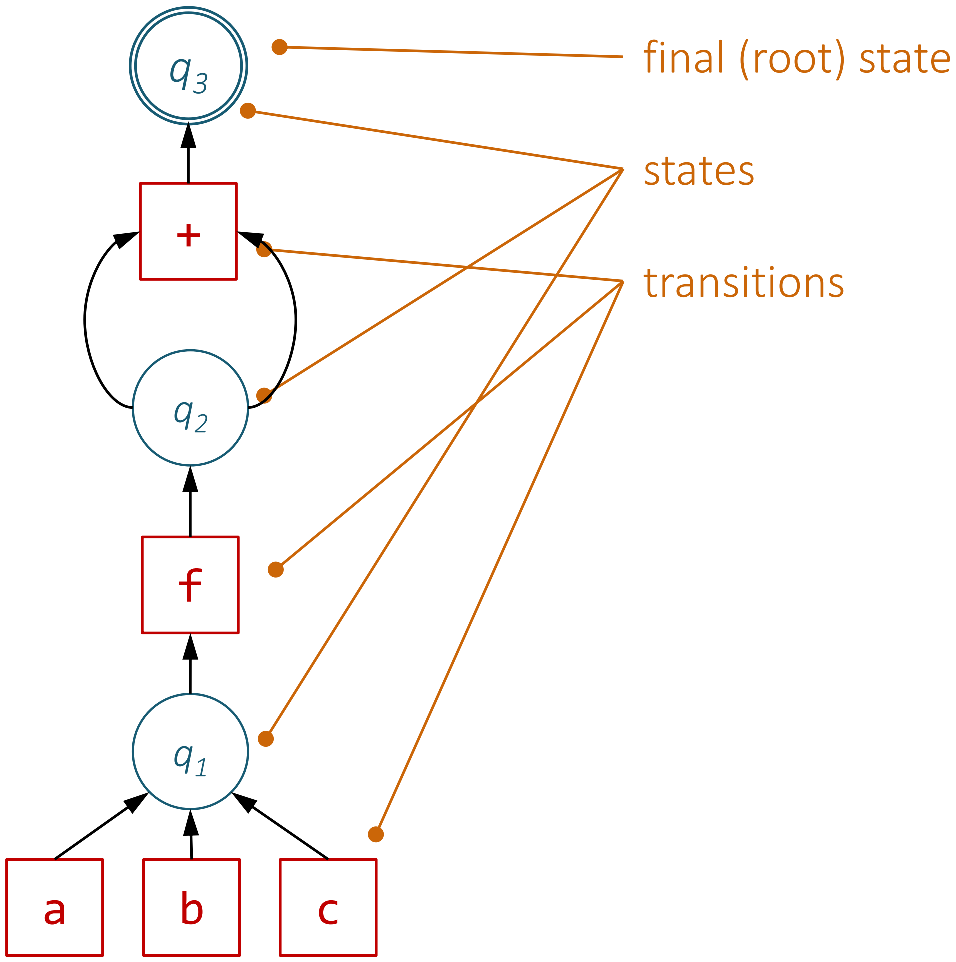
final state(s)

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

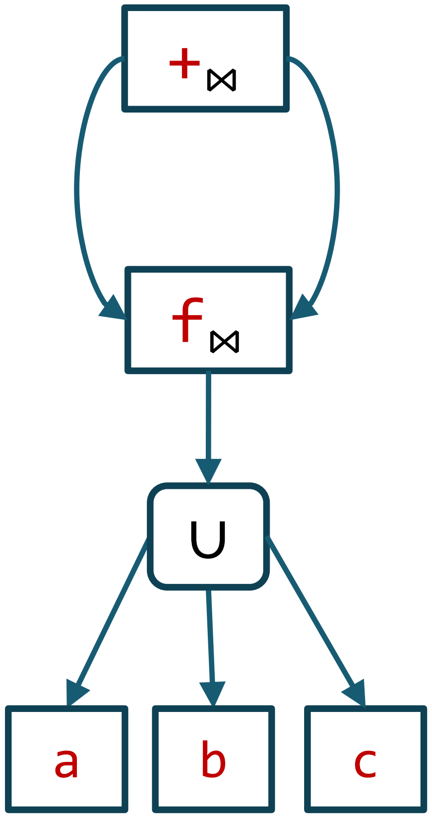
ranked alphabet

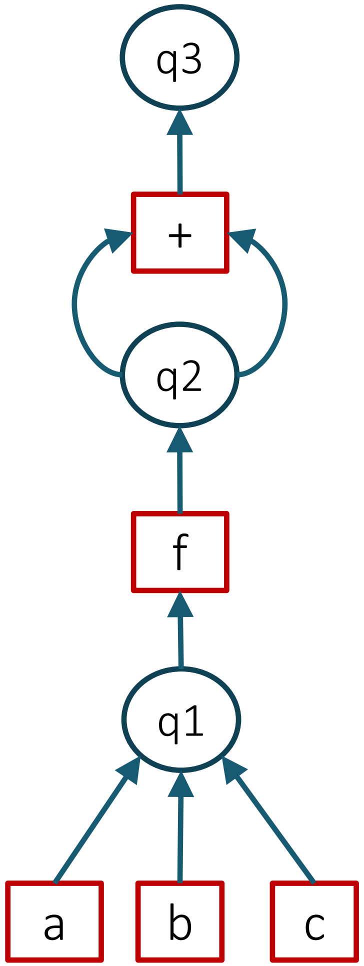
transitions

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



VSA vs FTA





Both are and-or graphs

FTA state = VSA union node

- in VSAs singleton unions are omitted

FTA transition = VSA join node

FTA-based search

Synthesis of Data Completion Scripts using Finite Tree Automata
Xinyu Wang, Isil Dillig, Rishabh Singh, *OOPSLA'17*

Program Synthesis using Abstraction Refinement
Xinyu Wang, Isil Dillig, Rishabh Singh, *POPL'18*

Searching Entangled Program Spaces
James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, Nadia Polikarpova. *ICFP'22*

FTA-based search

Synthesis of Data Completion Scripts using Finite Tree Automata
Xinyu Wang, Isil Dillig, Rishabh Singh, *OOPSLA'17*

Program Synthesis using Abstraction Refinement
Xinyu Wang, Isil Dillig, Rishabh Singh, *POPL'18*

Searching Entangled Program Spaces
James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, Nadia Polikarpova. *ICFP'22*

Example

Grammar

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

Spec

1 → 9

PBE with 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, +, *`

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

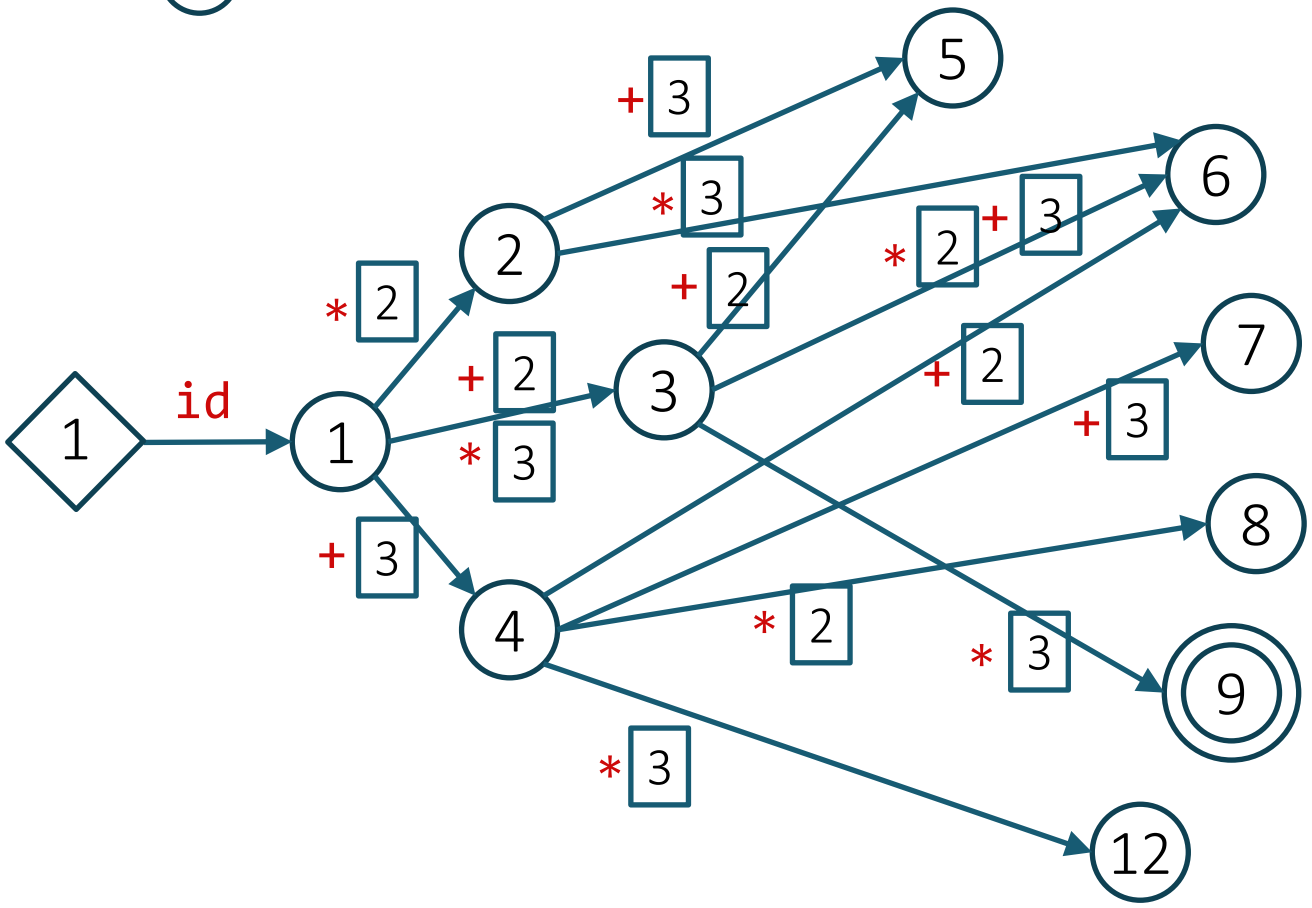
`+(<N,1>,<T,2>) → <N,3>`

`...`

PBE with Finite Tree Automata

```
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
- More incremental in the CEGIS context!

FTA-based search

Synthesis of Data Completion Scripts using Finite Tree Automata
Xinyu Wang, Isil Dillig, Rishabh Singh, *OOPSLA'17*

Program Synthesis using Abstraction Refinement
Xinyu Wang, Isil Dillig, Rishabh Singh, *POPL'18*

Searching Entangled Program Spaces
James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, Nadia Polikarpova. *ICFP'22*

Abstract FTA

Challenge: FTA still has too many states

Idea:

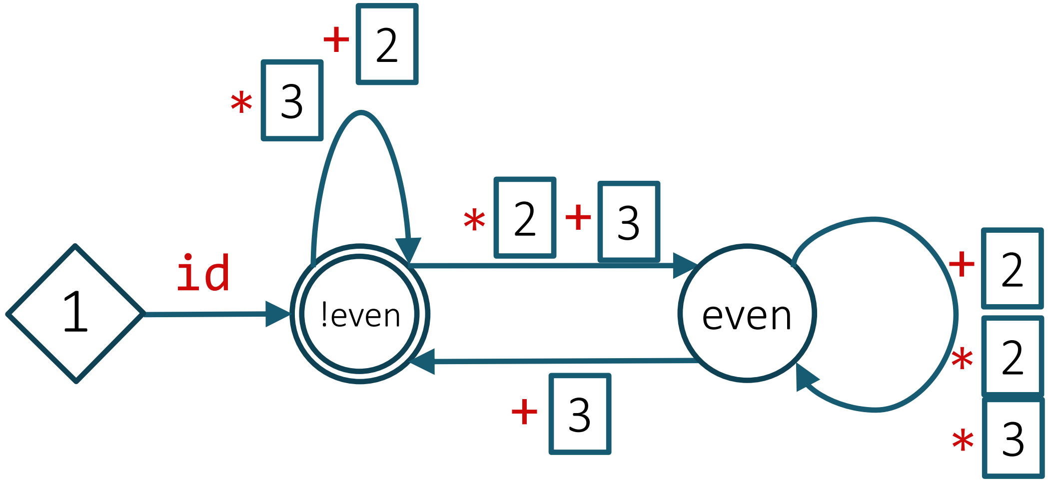
- instead of one state = one value
- we can do one state = set of values (= abstract value)

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

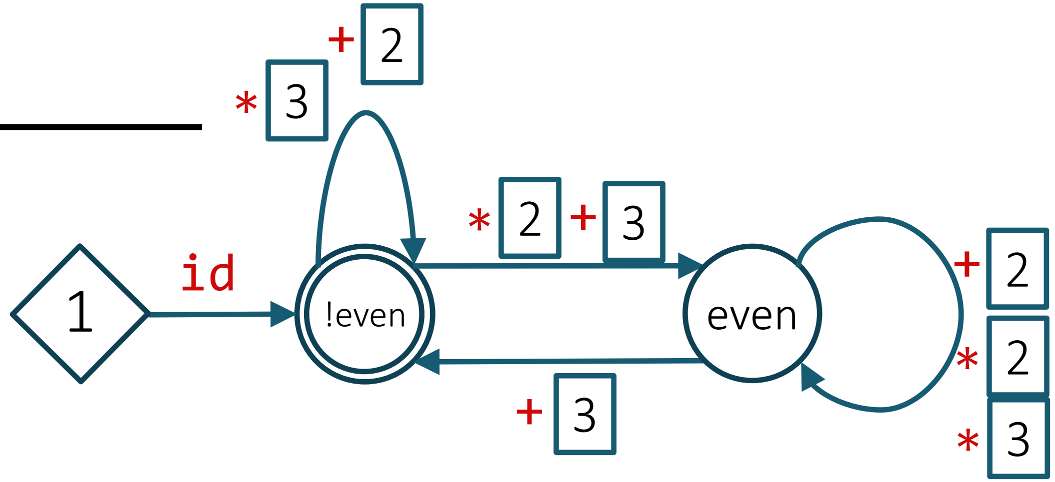
N ::= id(V) | N + T | N * T

T ::= 2 | 3

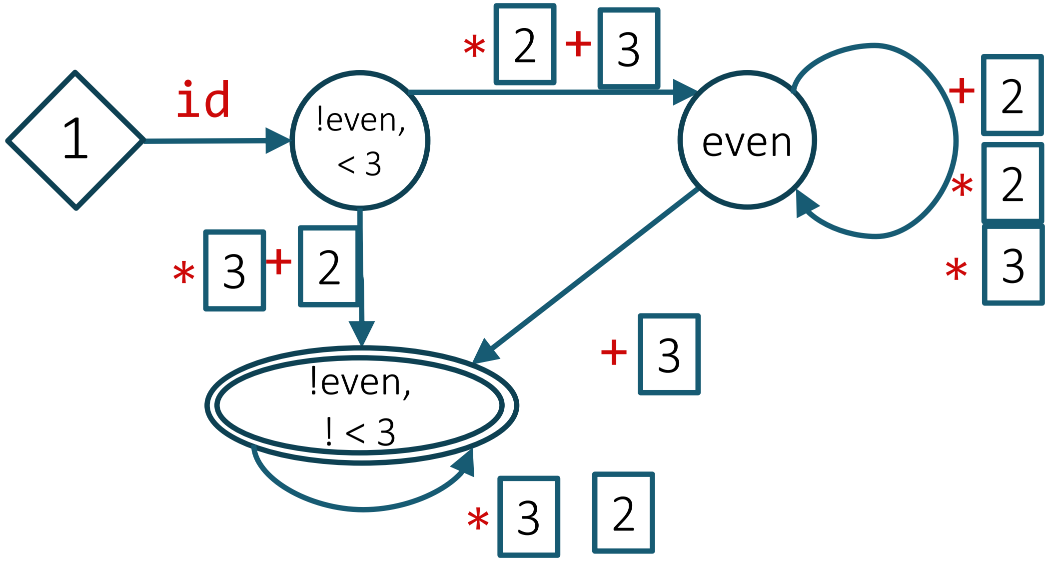
V ::= x

1 → 9

Predicates: {even, < 3, ...}



solution: id(x)



solution: id(x)*3

Representation-based search

and/or graph

Version Space Algebras
(VSA)

Finite Tree Automata
(FTA)

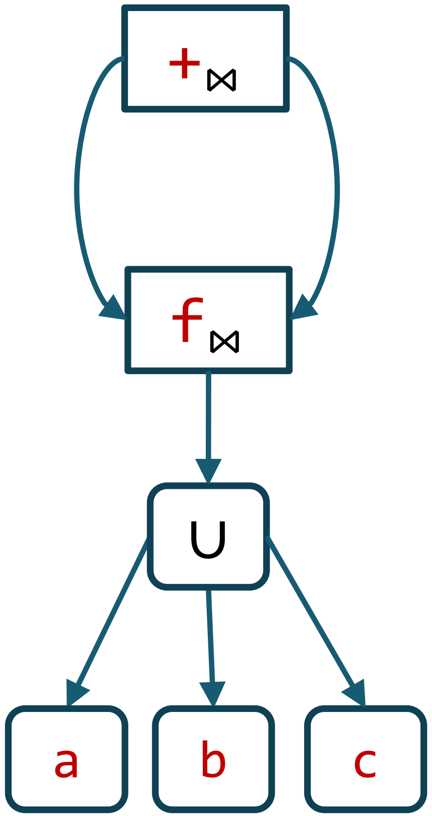
Equivalence Graphs
(e-grpahs)

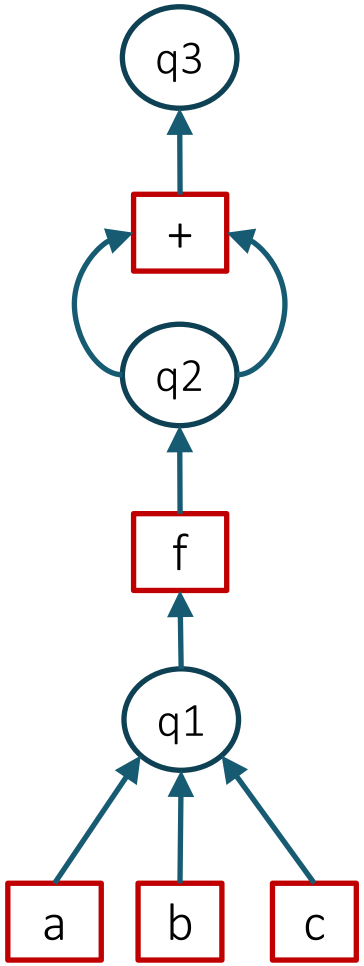
ops: learn-1, intersect, extract
DSL: efficiently invertible
similar to: top-down prop,
but can infer constants

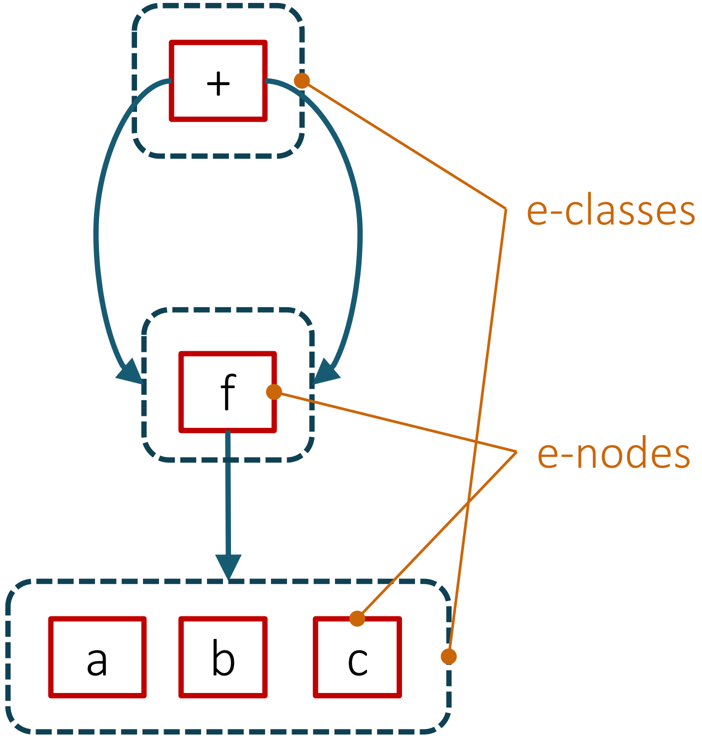
ops: learn-1, intersect, extract
DSL: efficiently enumerable
similar to: bottom-up with OE,
but can store all programs
(and add examples incrementally)

state: represents a set of
observationally-equivalent programs

VSA vs FTA vs E-Graphs







Program search with e-graphs

Equality saturation: a new approach to optimization
Ross Tate, Michael Stepp, Zachary Tatlock, Sorin Lerner, *POPL'09*

egg: Fast and Extensible Equality Saturation
Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, Pavel Panchekha, *POPL'21*

Semantic Code Search via Equational Reasoning
Varot Premtoon, James Koppel, Armando Solar-Lezama. *PLDI'20*

Equality saturation

Program optimization via rewriting:

```
    (a * 2) / 2
⇒  a * (2 / 2)
⇒  a * 1
⇒  a
```

useful rules:

```
(x * y) / z = x * (y / z)
      x / x = 1
      x * 1 = x
```

not so useful:

```
x * 2 = x << 1
x * y = y * x
```

Challenge: which ones to apply and in what order?

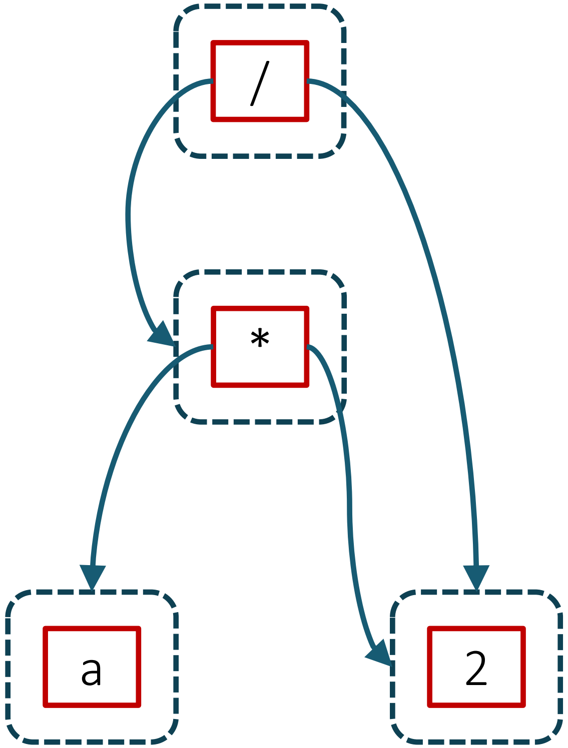
Idea: all of them all the time!

Equality saturation

Initial term: `(a * 2) / 2`

Rewrite rules:

```
(x * y) / z = x * (y / z)
        x / x = 1
        x * 1 = x
      x * 2 = x << 1
       x * y = y * x
```

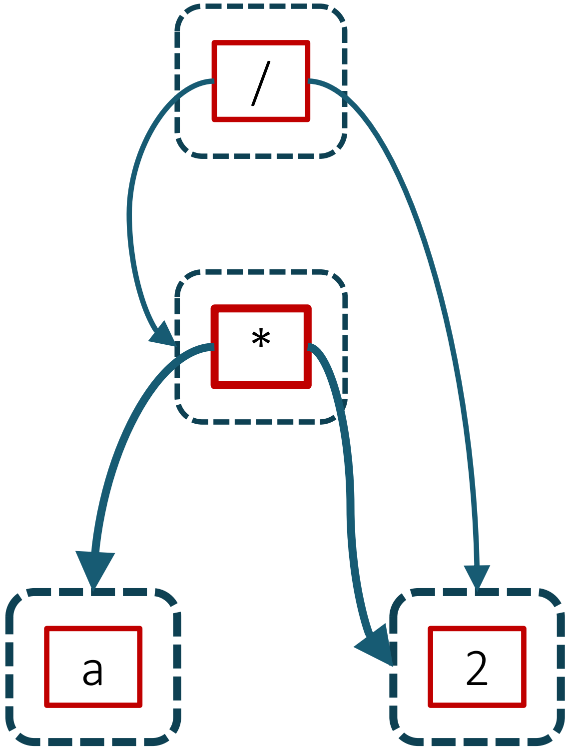


Equality saturation

Initial term: `(a * 2) / 2`

Rewrite rules:

```
(x * y) / z = x * (y / z)
      x / x = 1
      x * 1 = x
    x * 2 = x << 1
     x * y = y * x
```

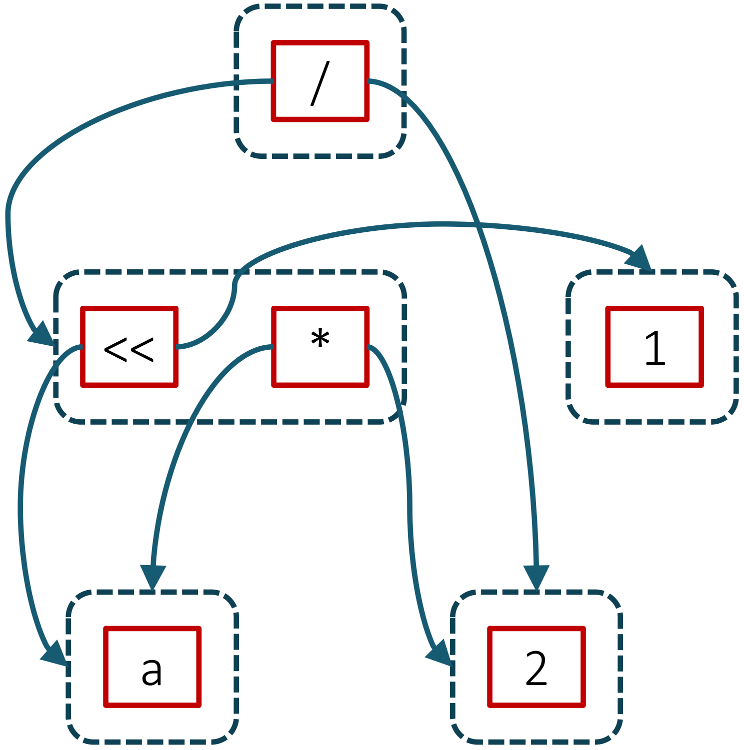


Equality saturation

Initial term: `(a * 2) / 2`

Rewrite rules:

```
(x * y) / z = x * (y / z)
      x / x = 1
      x * 1 = x
    x * 2 = x << 1
     x * y = y * x
```

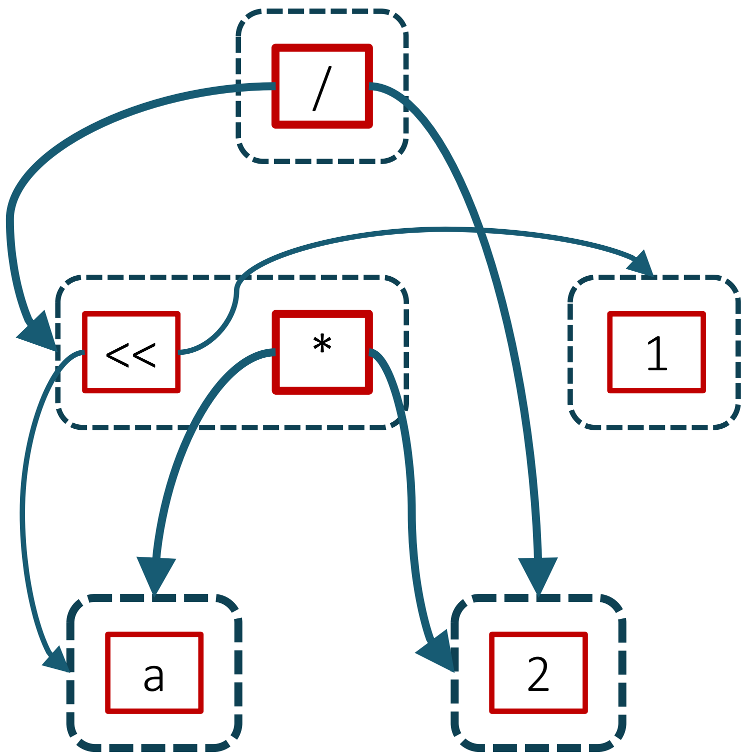


Equality saturation

Initial term: `(a * 2) / 2`

Rewrite rules:

```
(x * y) / z = x * (y / z)
      x / x = 1
      x * 1 = x
  x * 2 = x << 1
   x * y = y * x
```

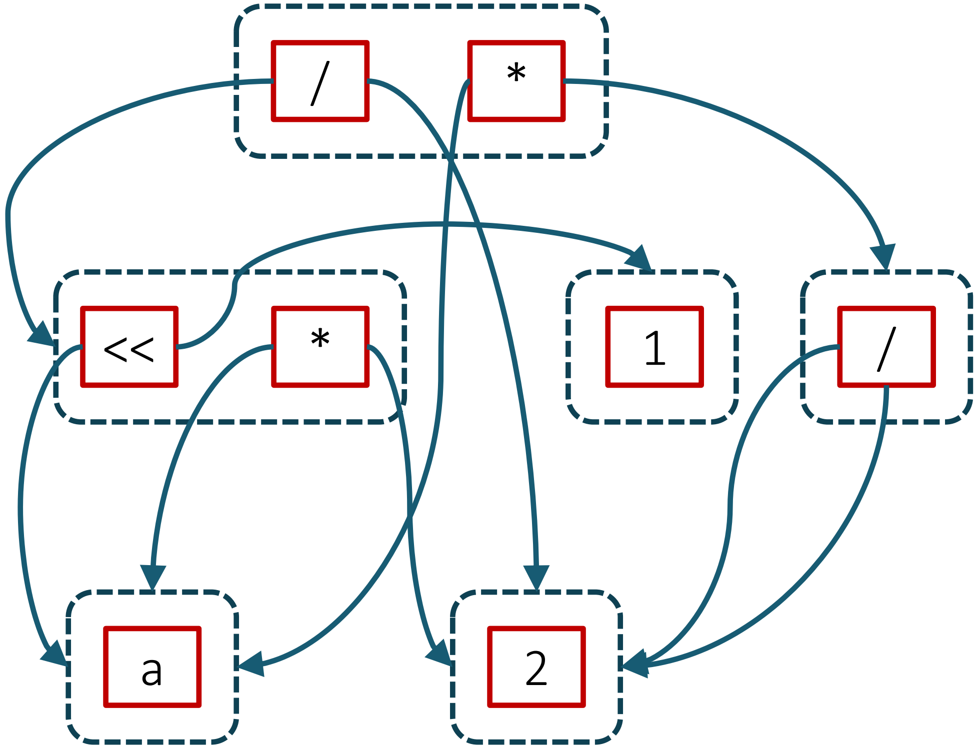


Equality saturation

Initial term: `(a * 2) / 2`

Rewrite rules:

```
(x * y) / z = x * (y / z)
x / x = 1
x * 1 = x
x * 2 = x << 1
x * y = y * x
```

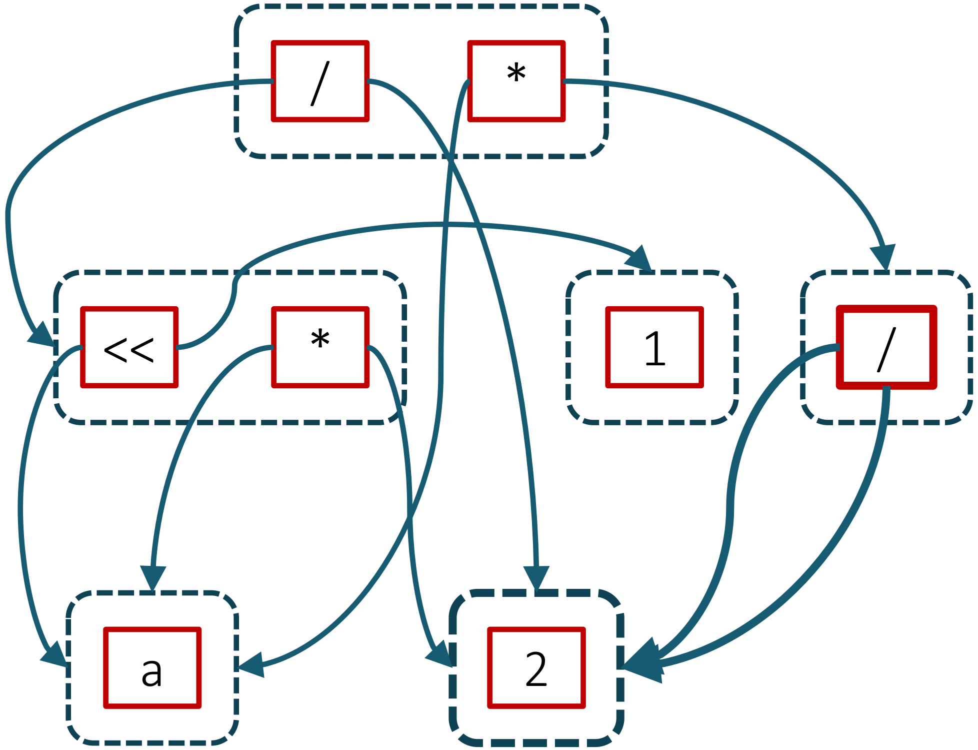


Equality saturation

Initial term: `(a * 2) / 2`

Rewrite rules:

```
(x * y) / z = x * (y / z)
x / x = 1
x * 1 = x
x * 2 = x << 1
x * y = y * x
```

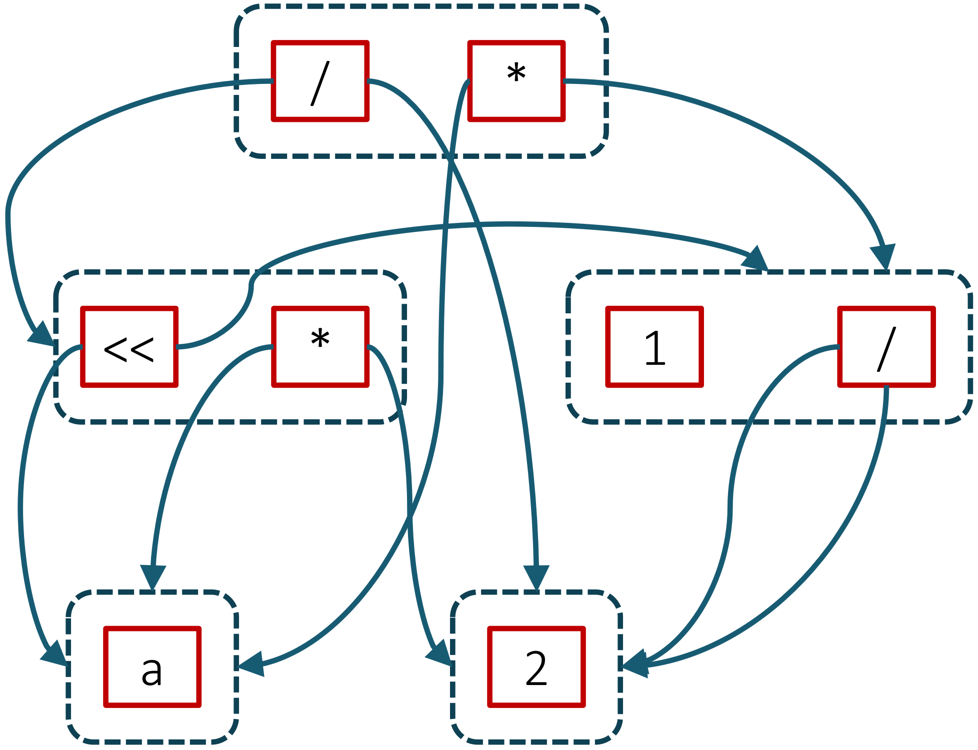


Equality saturation

Initial term: `(a * 2) / 2`

Rewrite rules:

```
(x * y) / z = x * (y / z)
x / x = 1
x * 1 = x
x * 2 = x << 1
x * y = y * x
```

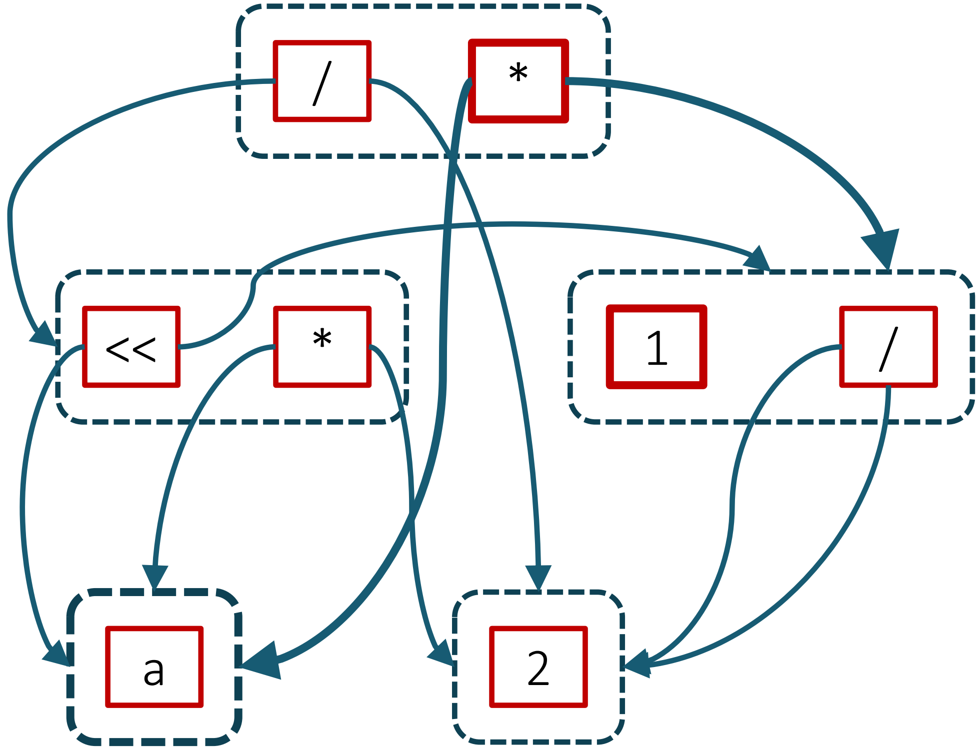


Equality saturation

Initial term: `(a * 2) / 2`

Rewrite rules:

```
(x * y) / z = x * (y / z)
        x / x = 1
        x * 1 = x
      x * 2 = x << 1
       x * y = y * x
```

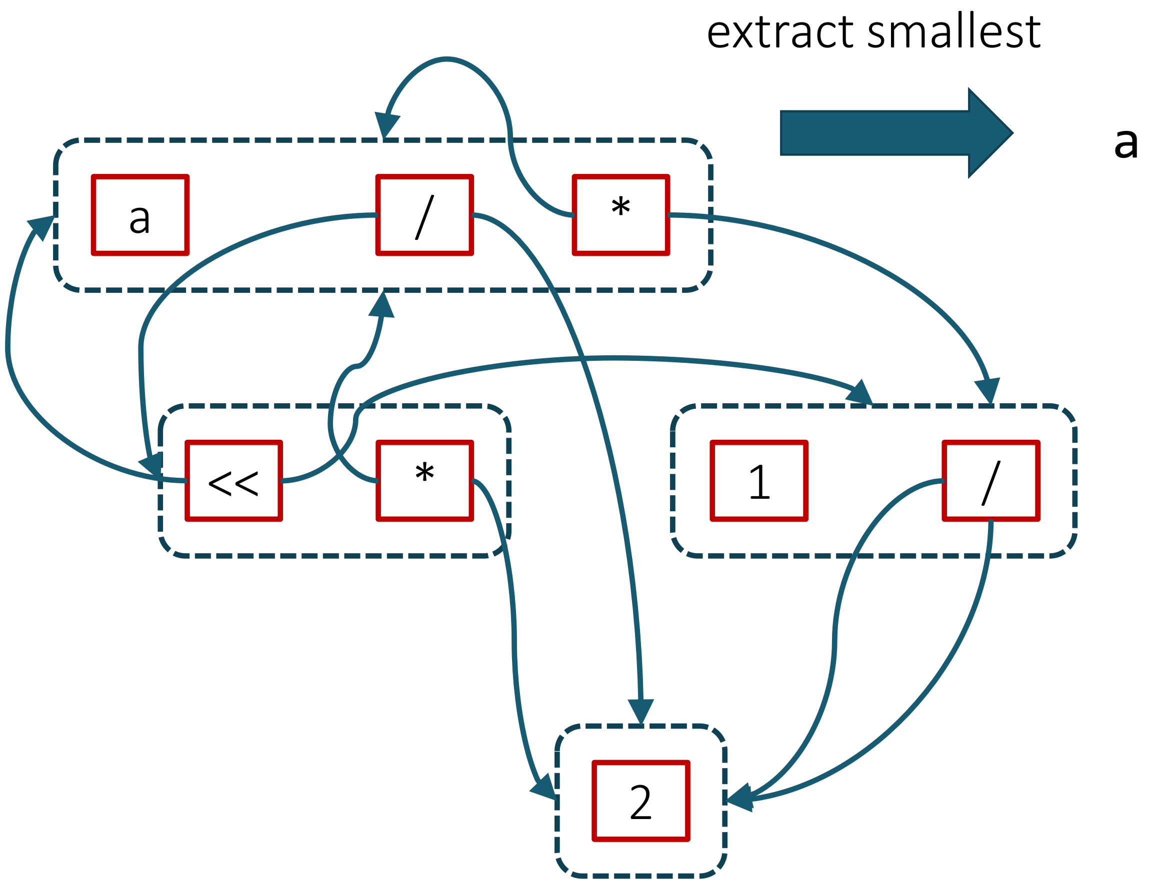


Equality saturation

Initial term: `(a * 2) / 2`

Rewrite rules:

```
(x * y) / z = x * (y / z)
x / x = 1
x * 1 = x
x * 2 = x << 1
x * y = y * x
```



Representation-based search

and/or graph

Version Space Algebras (VSA)	Finite Tree Automata (FTA)	Equivalence Graphs (e-grpahs)
ops: learn-1, intersect, extract	**ops:** learn-1, intersect, extract	**ops:** rewrite, extract
DSL: efficiently invertible	**DSL:** efficiently enumerable	
similar to: top-down prop, but can infer constants	**similar to:** bottom-up with OE, but can store all programs (and add examples incrementally)	**similar to:** term rewriting, but can store all programs

state: represents a set of observationally-equivalent programs

e-class: represents a set of programs equivalent up to rewrites

BlinkFill

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

- input-output examples (+ input examples)
- custom string DSL
- VSA

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"`

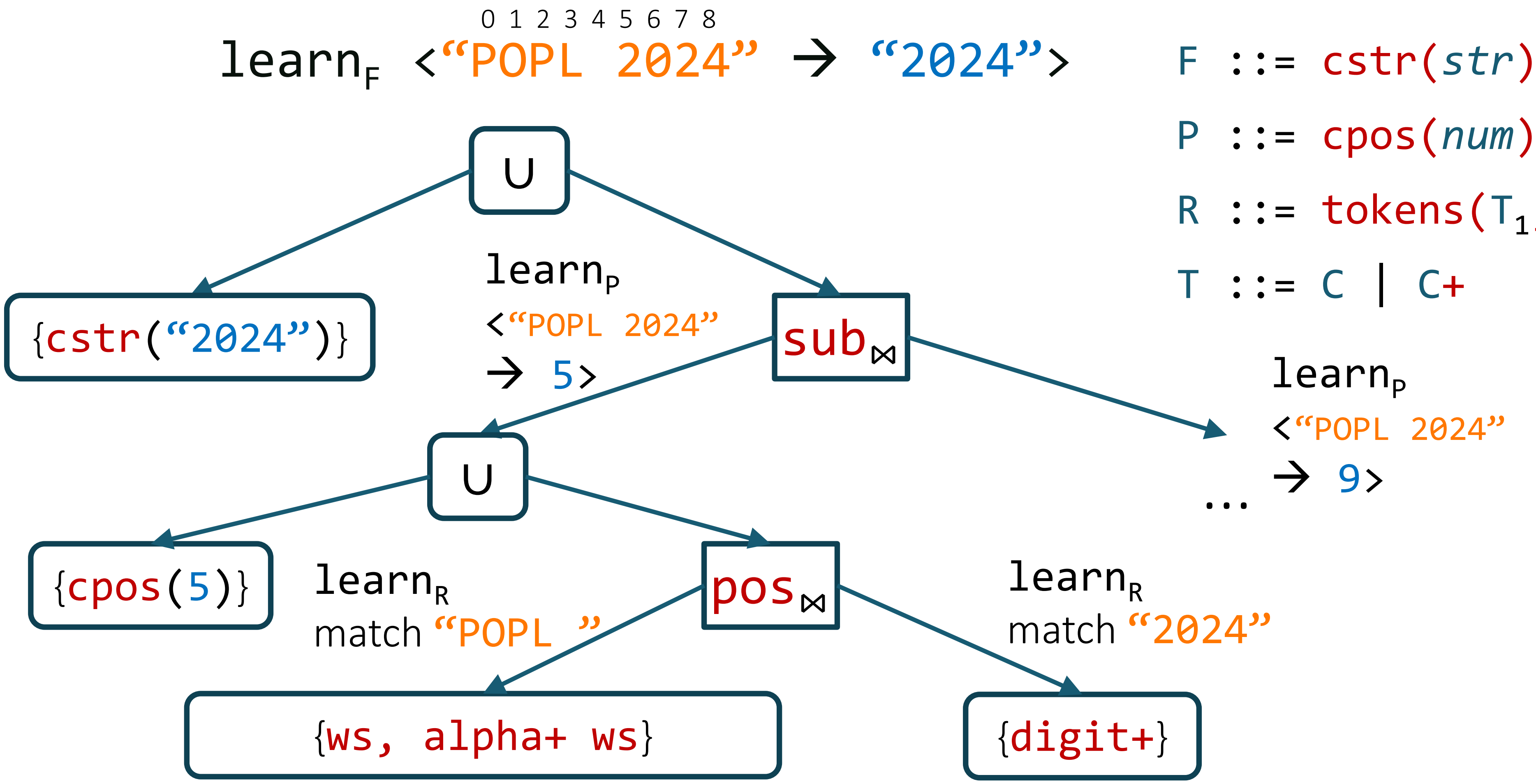
- Between first parentheses and between first and last comma:

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

BlinkFill

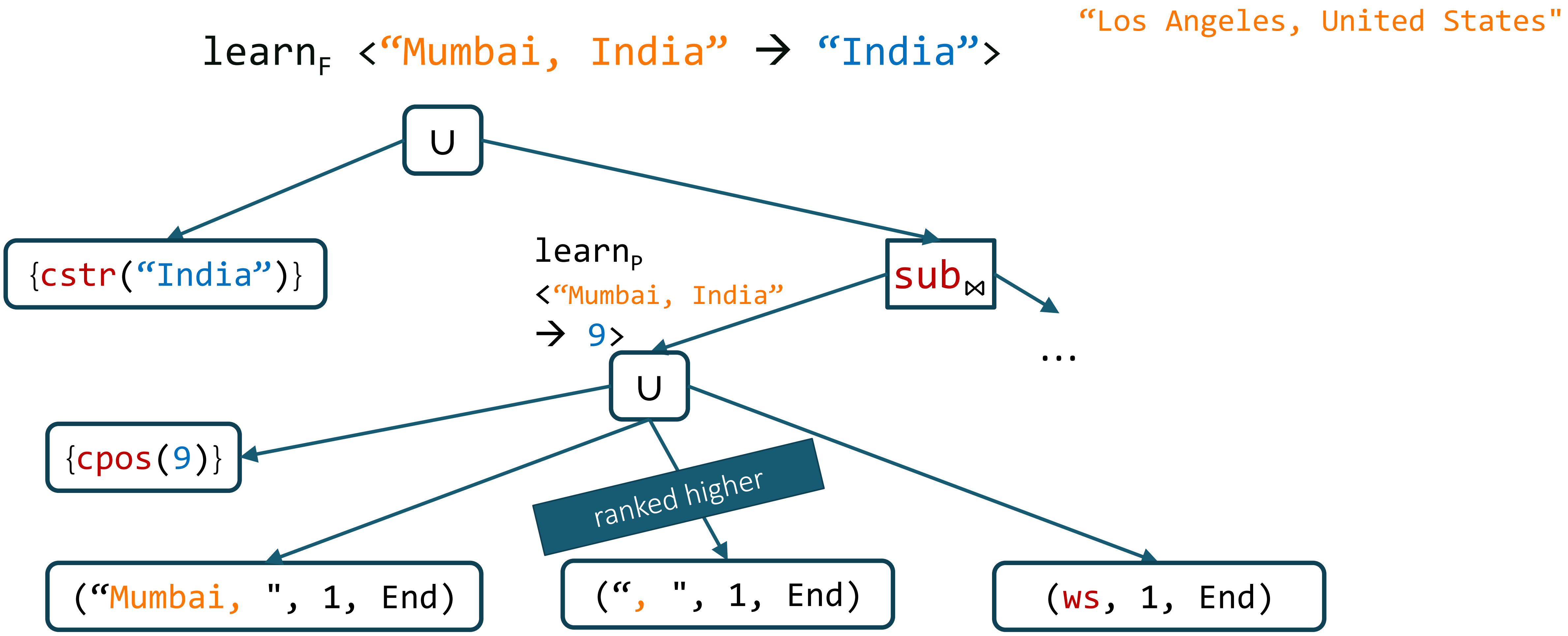
How does BlinkFill use the IDG to help synthesis?

Recall: Flashfill

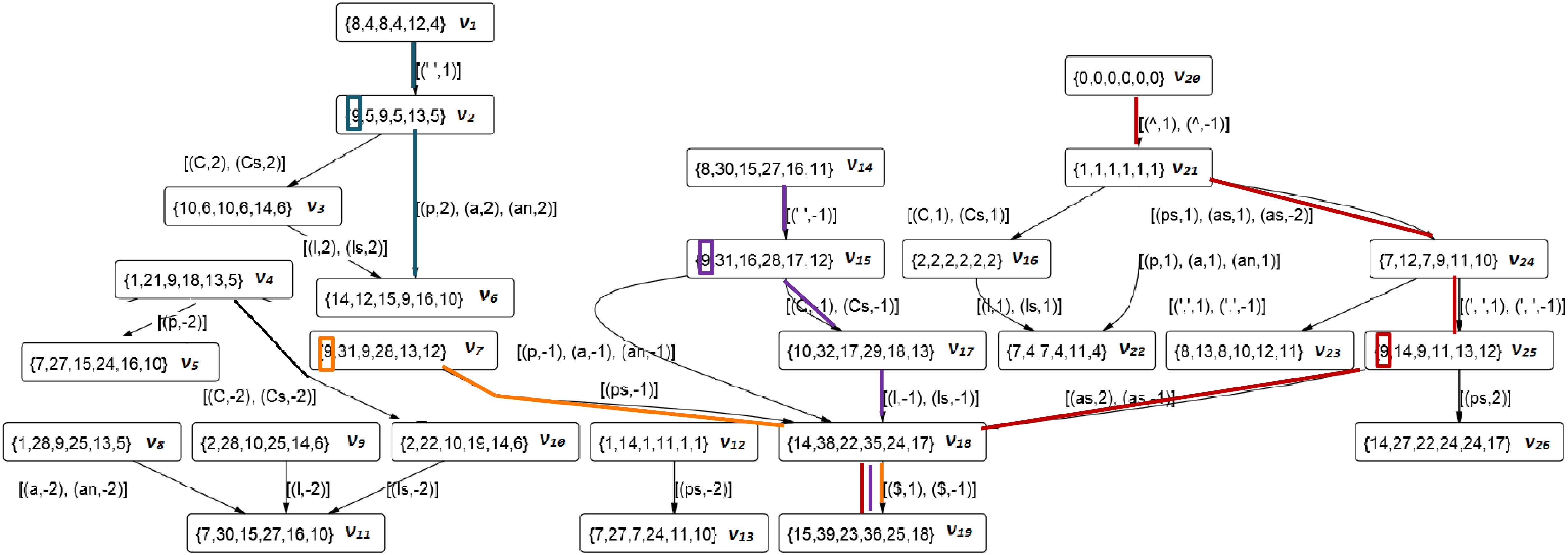



```
F ::= cstr(str) | sub(P1, P2)
P ::= cpos(num) | pos(R1, R2)
R ::= tokens(T1, ..., Tn)
T ::= C | C+
```

Blinkfill



"Los Angeles, United States"
learn_F <"Mumbai, India" → "India">
∪
{cstr("India")}
sub⋈
learn_P
<"Mumbai, India"
→ 9>
...
∪
{cpos(9)}
ranked higher
("Mumbai, ", 1, End)
(", ", 1, End)
(ws, 1, End)



{8,4,8,4,12,4} v1
[(' ',1)]
{9,5,9,5,13,5} v2
[(C,2), (Cs,2)]
[(p,2), (a,2), (an,2)]
{10,6,10,6,14,6} v3
[(l,2), (ls,2)]
{1,21,9,18,13,5} v4
{14,12,15,9,16,10} v6
[(p,-2)]
{7,27,15,24,16,10} v5
{9,31,9,28,13,12} v7
[(C,-2), (Cs,-2)]
{1,28,9,25,13,5} v8
{2,28,10,25,14,6} v9
{2,22,10,19,14,6} v10
[(a,-2), (an,-2)]
[(l,-2)]
[(ls,-2)]
{7,30,15,27,16,10} v11
{8,30,15,27,16,11} v14
[(' ',-1)]
{9,31,16,28,17,12} v15
[(p,-1), (a,-1), (an,-1)]
[(ps,-1)]
[(C,-1), (Cs,-1)]
{10,32,17,29,18,13} v17
[(l,-1), (ls,-1)]
{1,14,1,11,1,1} v12
[(ps,-2)]
{7,27,7,24,11,10} v13
{14,38,22,35,24,17} v18
[($,1), ($,-1)]
{15,39,23,36,25,18} v19
{0,0,0,0,0,0} v20
[(^,1), (^,-1)]
{1,1,1,1,1,1} v21
[(C,1), (Cs,1)]
[(ps,1), (as,1), (as,-2)]
{2,2,2,2,2,2} v16
[(p,1), (a,1), (an,1)]
{7,12,7,9,11,10} v24
[(l,1), (ls,1)]
[(',',1), (',',-1)]
[(' ',1), (' ',-1)]
{7,4,7,4,11,4} v22
{8,13,8,10,12,11} v23
{9,14,9,11,13,12} v25
[(as,2), (as,-1)]
[(ps,2)]
{14,27,22,24,24,17} v26

BlinkFill

How does BlinkFill use the IDG to help synthesis?

- the IDG stores common "parses" of all input substrings
- this information is used when building the VSA for `pos` expressions
- it is used for both *pruning* tokens that do not appear in some inputs and *prioritizing* tokens by the longest parse they participate in
- IDG makes the results better:
  - prevents overfitting of `pos` expressions
- it also makes synthesis faster
  - fewer CEGIS iterations; but also within one iteration:
  - faster enumeration of `pos` expressions
  - constant tokens allow simplifying the rest of the DSL

BlinkFill

Why can BlinkFill afford to use constant string tokens and FlashFill cannot?

- main reason: with too few examples, constant tokens would overfit
- also: in the absence of a smart algo to enumerate only those tokens that match all inputs, enumerating a large space of tokens would be slow

BlinkFill

Strengths? Weaknesses?

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