Lecture 5
Representation-based Search

Nadia Polikarpova

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
Stochastic
Representation-based

Structural constraints = grammar

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

Representation-based search

Idea: pick a data structure that can succinctly represent a set of programs consistent with a spec

- called a **version space**

Operations on version spaces:

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

Sounds too good to be true?

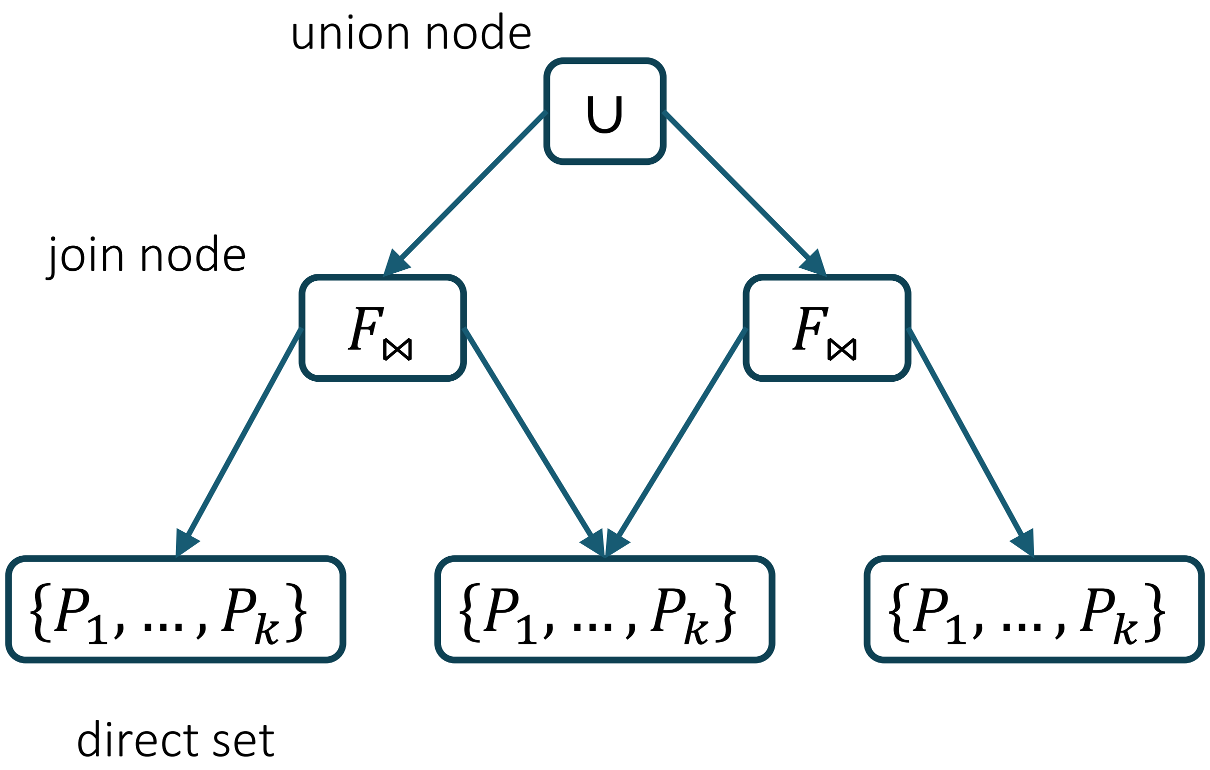
- Let’s see when it works

Representations?

Version Space Algebras

Finite Tree Automata

Version Space Algebra



Volume of a VSA
(the number of nodes)

$V(VSA)$

Size of a VSA
(the number of programs)

$|VSA|$

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

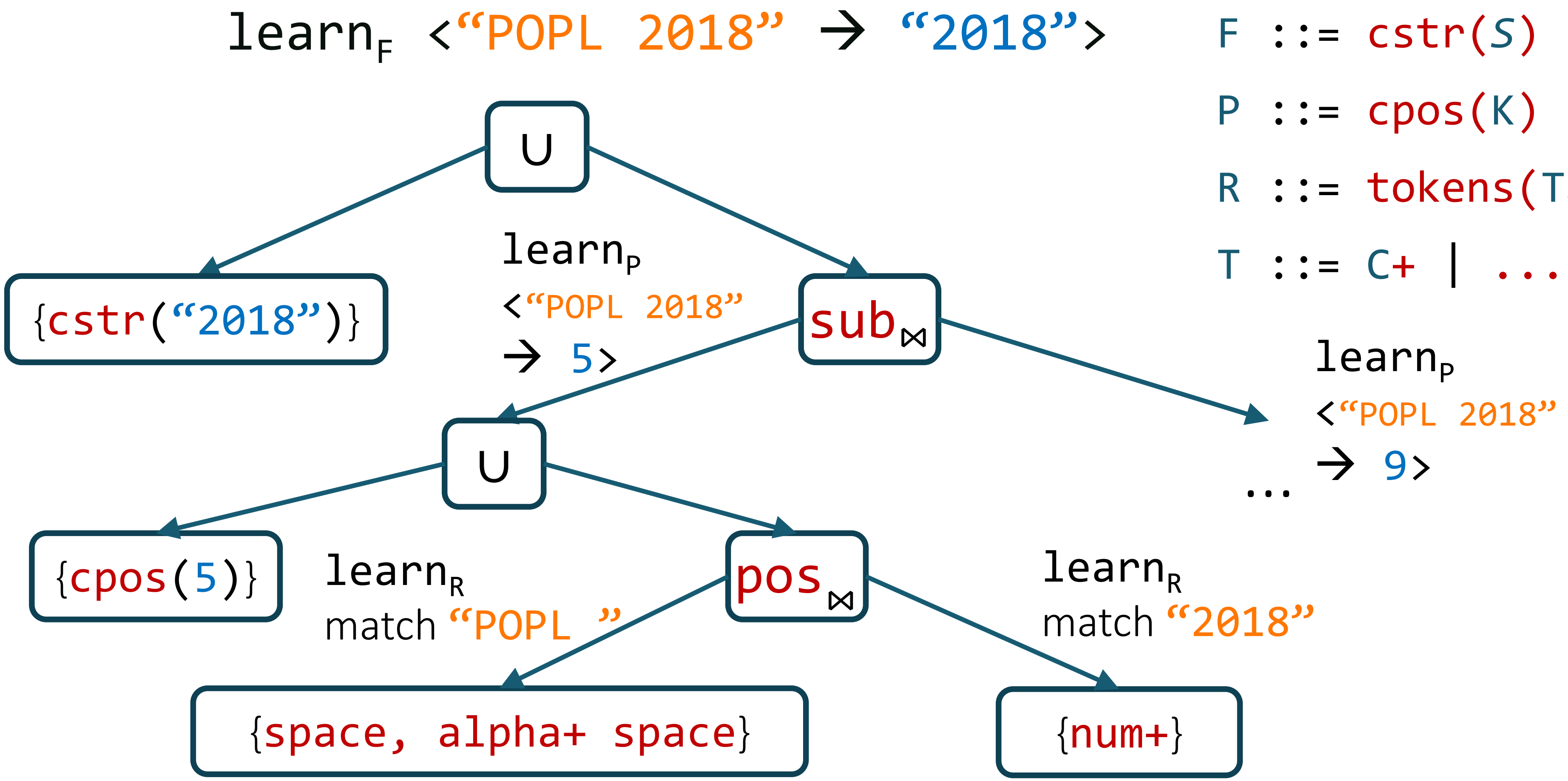
Example: FlashFill

[Gulwani '11]

Simplified grammar:

`E ::= F | concat(F, E)`		"Trace" expression
`F ::= cstr(S) | sub(P, P)`		Atomic expression
`P ::= cpos(K) | pos(R, R)`		Position expression
`R ::= tokens(T1, ..., Tn)`		Regular expression
`T ::= C+ | ...`		Token expression

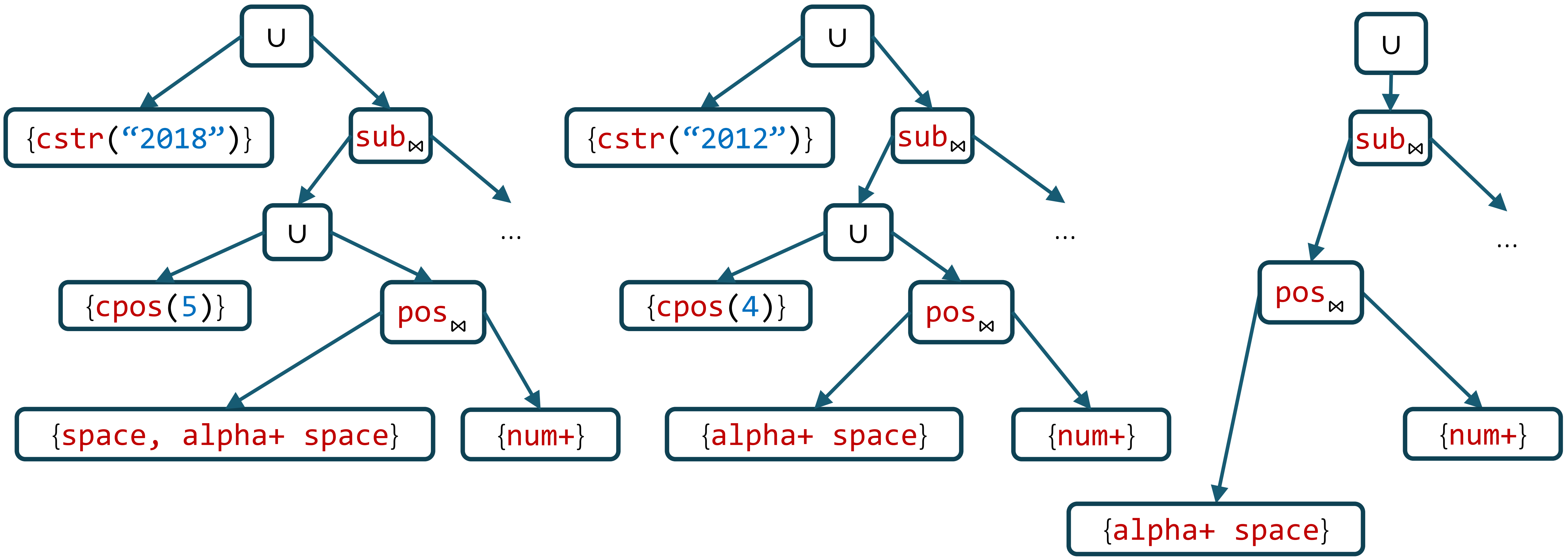
Learning atomic expressions




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

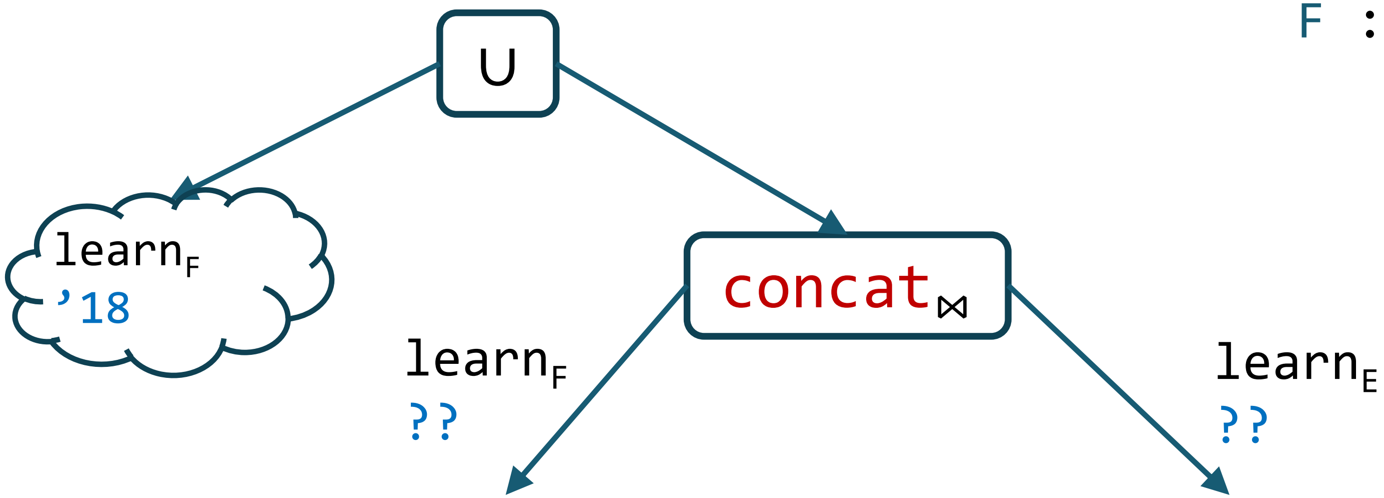
Intersection

"POPL 2018" → "2018"   " FM 2012" → "2012"



Learning trace expressions

learn$_E$ <“POPL 2018” → “’18”>



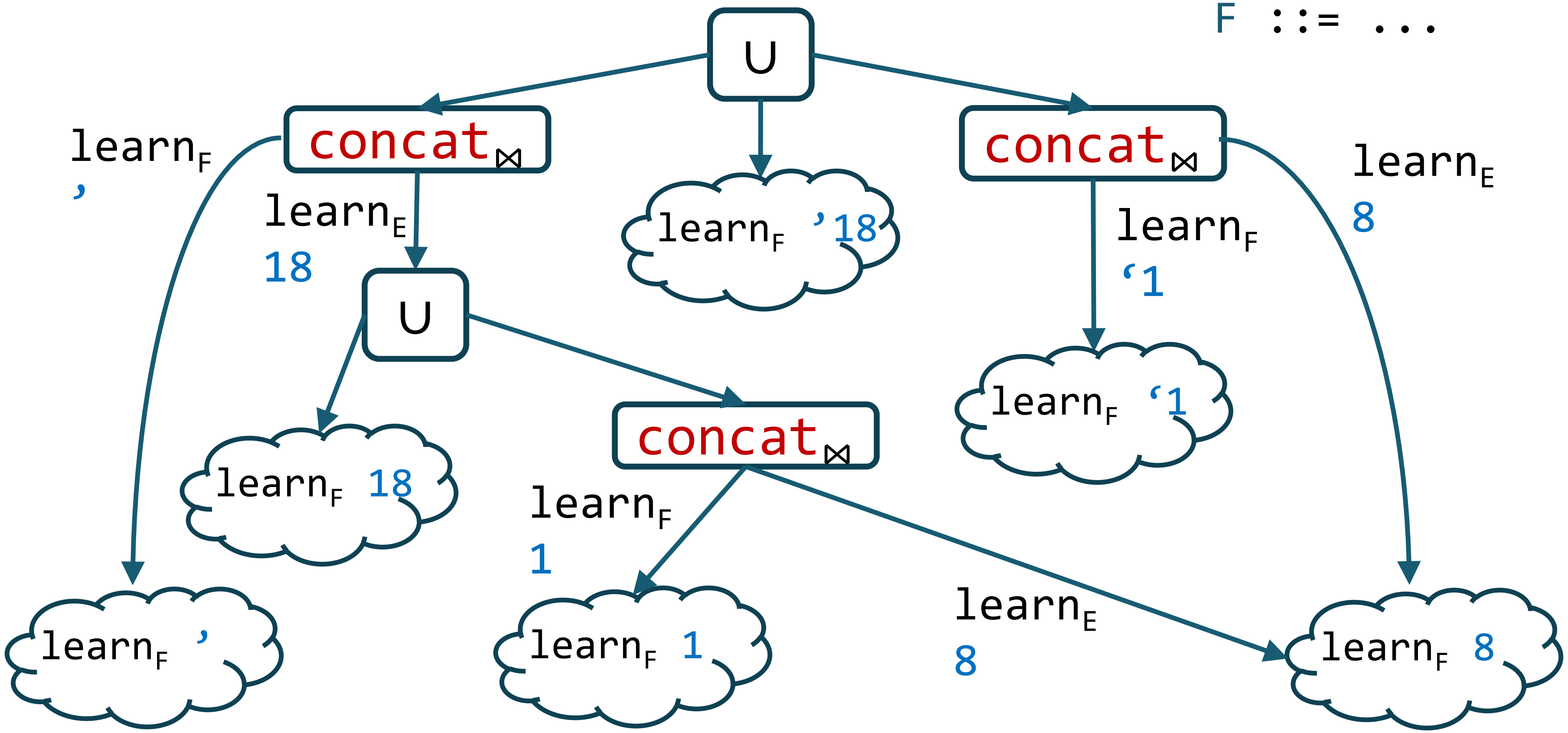

```
E ::= F | concat(F, E)
F ::= ...
```

Learning trace expressions

learn$_E$ <"POPL 2018" → "'18">

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?

  E ::= F | F + E

  F ::= K | x

  $K \in [0,9]$ + is addition mod 10

  - Could be represented finitely if we allowed loops in a VSA!

Discussion

Why could we build a *compact* representation of all expressions?

- Could we do this for this language?

```
E ::= F & F
F ::= K | x
```

K is a 32-bit word, & is bit-and

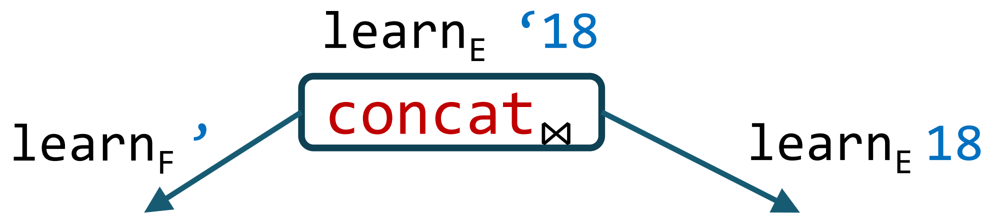
VSA: DSL restrictions

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)
```



- Otherwise, limit depth and "unroll" the grammar

PROSE

[Polozov, Gulwani '15]

Ranking function

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

Grammar

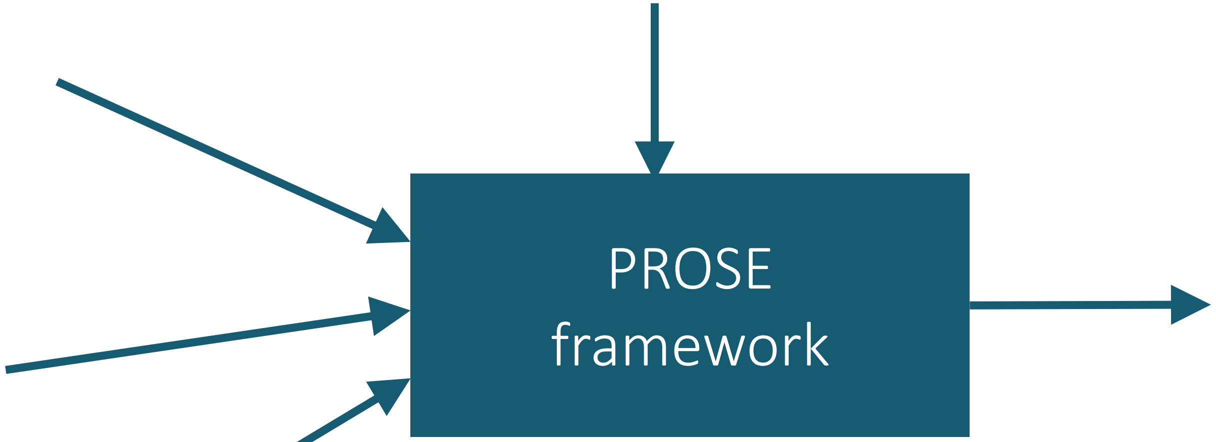
```
A ::= f(B, C) | ...
B ::= g(D, E) | ...
```

Semantics

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

Inverse semantics

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



VSA learning algorithm

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

Representations?

Version Space Algebras

Finite Tree Automata

Example

Grammar

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

Spec

$1 \rightarrow 9$

Finite Tree Automata

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

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

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

states

final states

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

alphabet

transitions

`id, +, *`

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

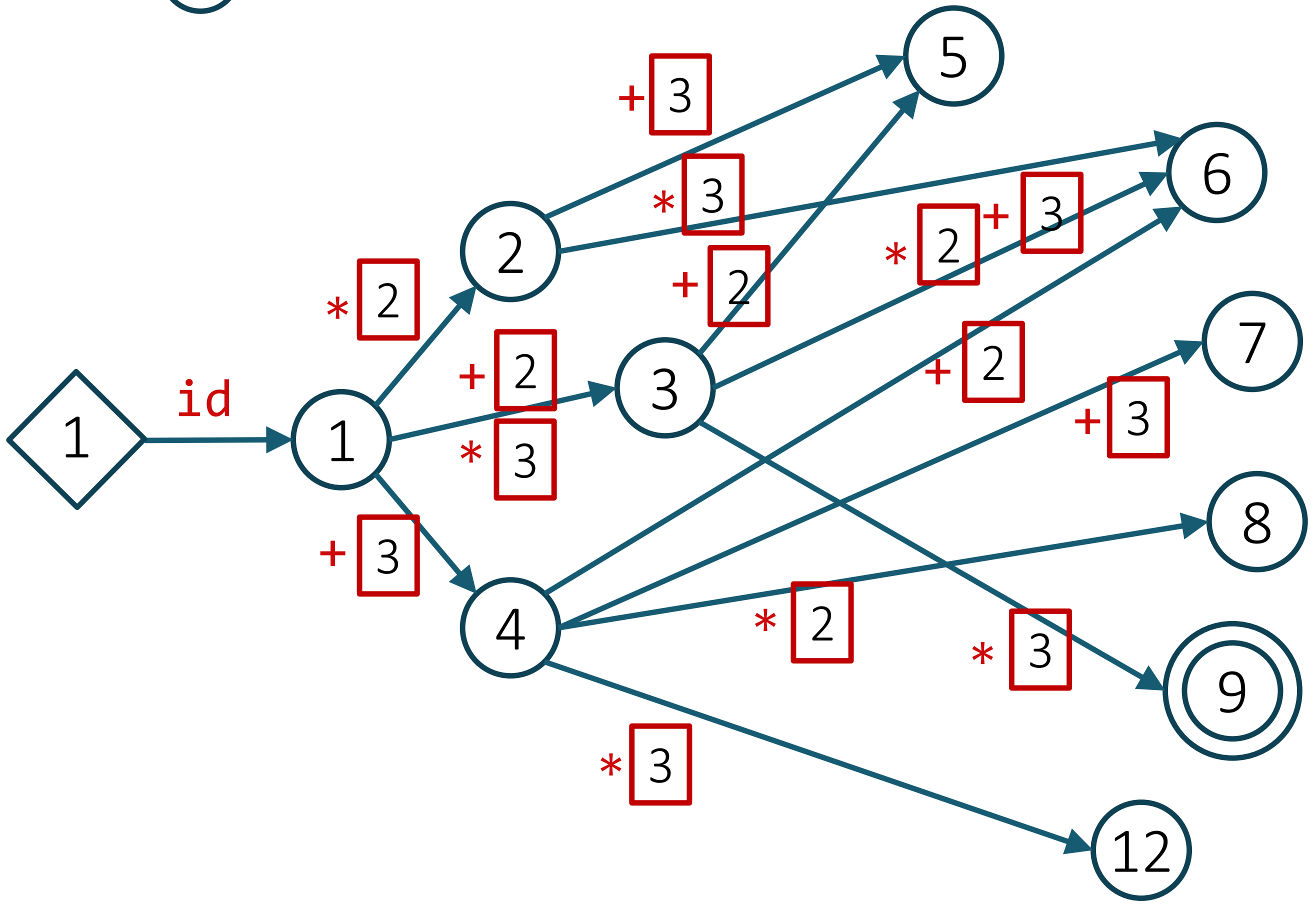
```
+(<N,1>,<T,2>) → <N,3>
...
```

Finite Tree Automata

[Wang, Dillig, Singh ‘17]

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

1 → 9



Discussion

Representation-based vs enumerative

- Enumerative unfolds the search space in time, while representation-based stores it in memory
- Benefits / limitations?

FTA ~ bottom-up

- with observational equivalence

VSA ~ top-down

- with top-down propagation

Next week

Topics:

- Constraint solving and constraint-based search

Paper: Jha, Gulwani, Seshia, Tiwari: Oracle-guided component-based program synthesis

- Questions coming soon

Project: proposals due next Friday