#9: Version Space Alegbra

Sankha Narayan Guria

EECS 700: Introduction to Program Synthesis

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

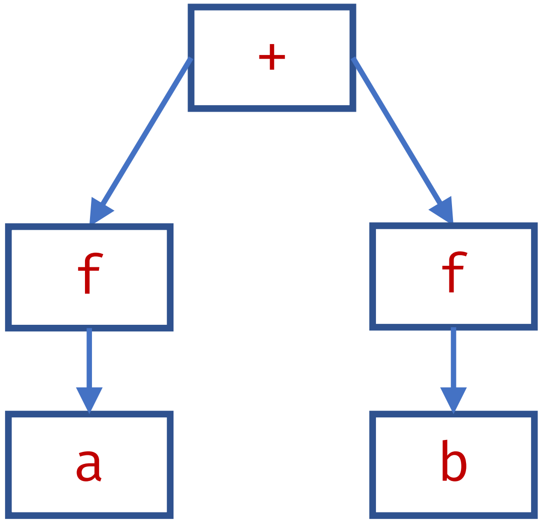
Idea:

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

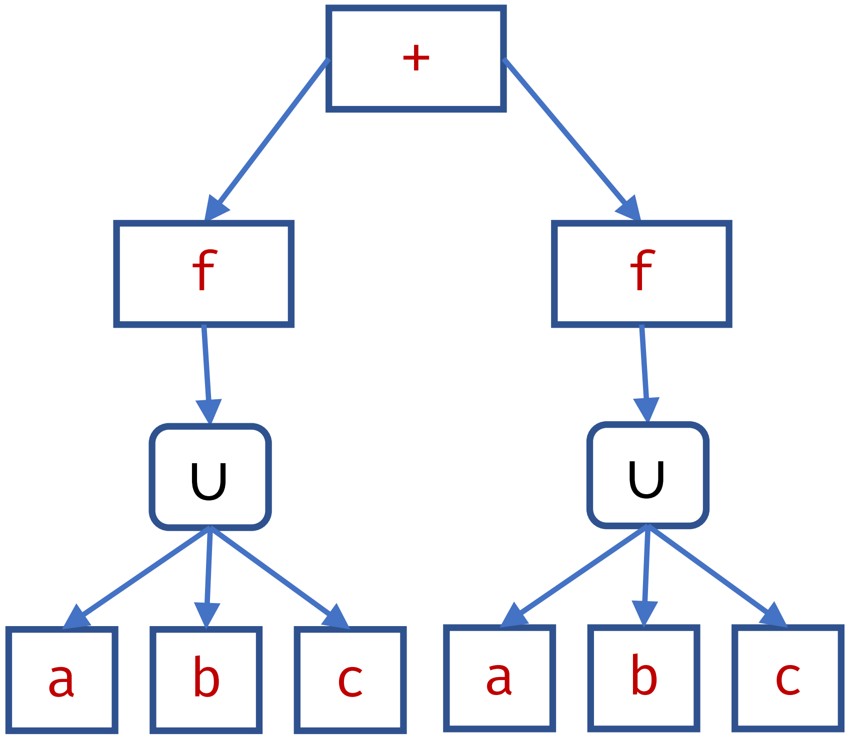
Compact term representation

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

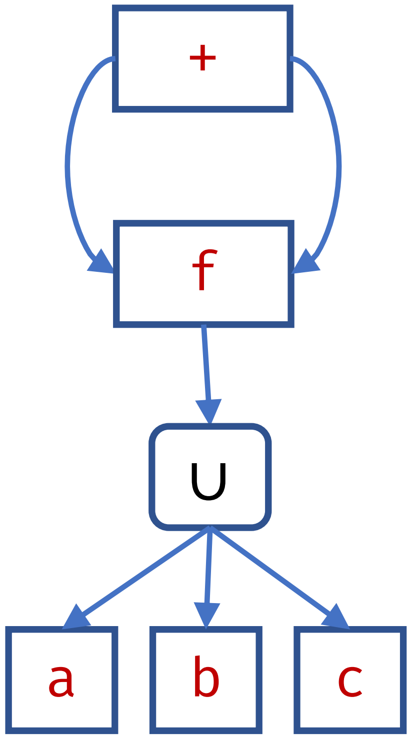
Consider the space of 9 programs:

```
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)
```

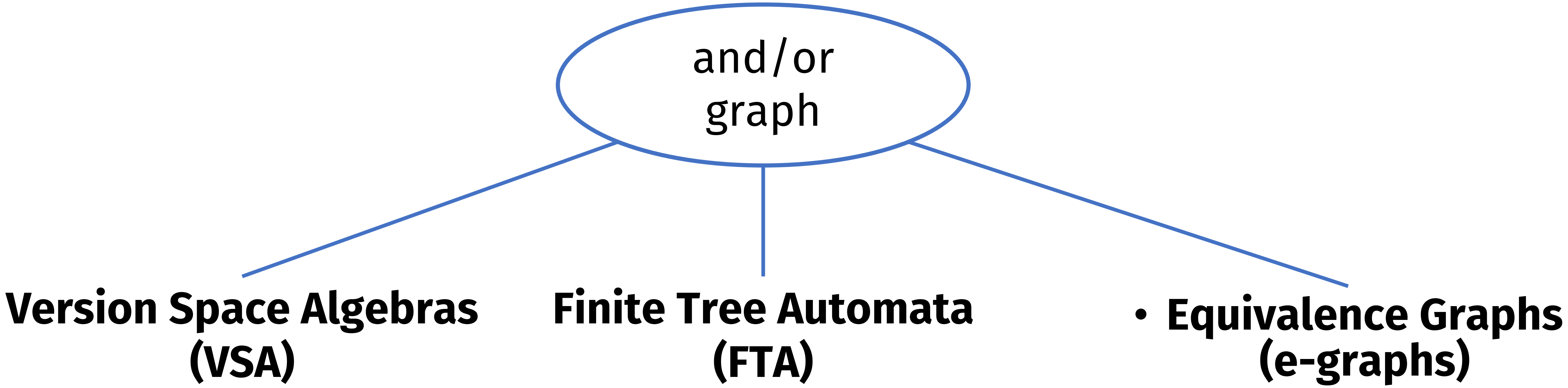
Can we represent this compactly?

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

Key idea: use and-or graph!



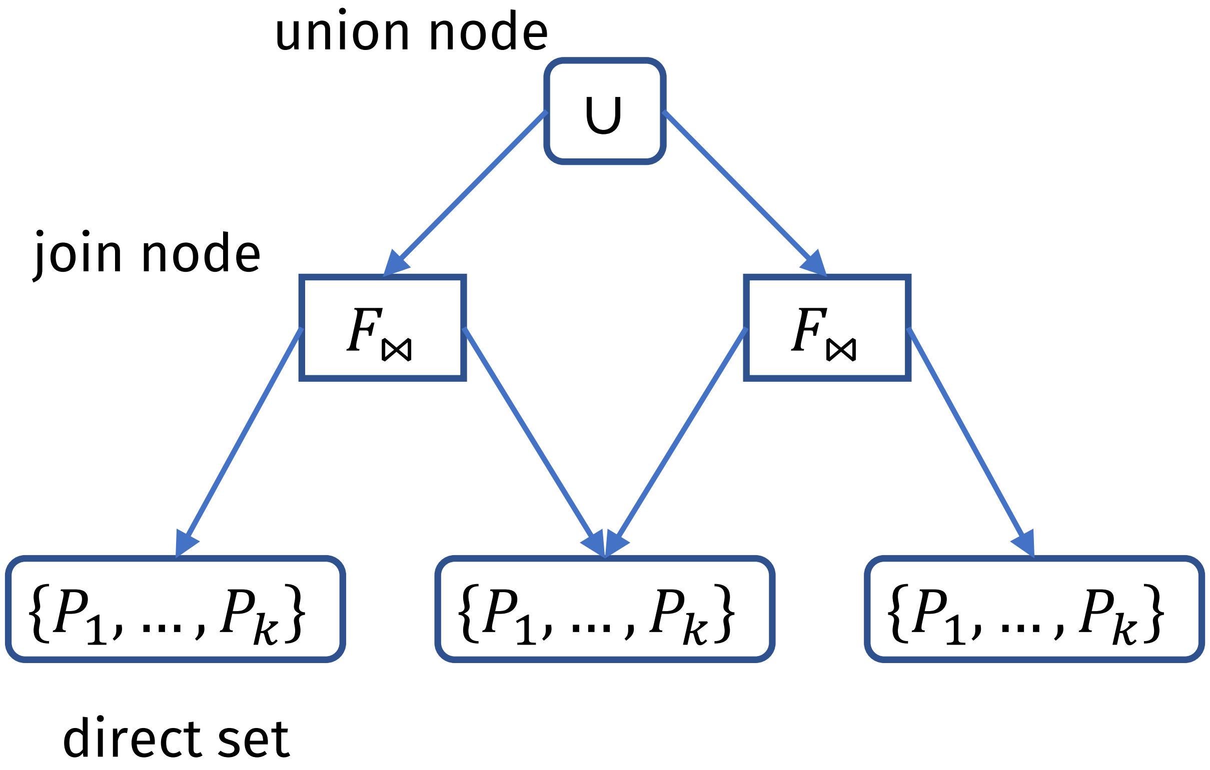
Representation-based search



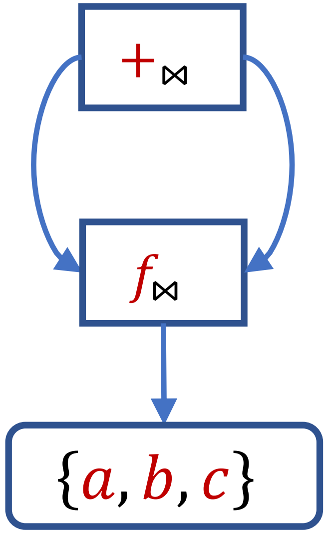
Version Space Algebra

- **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

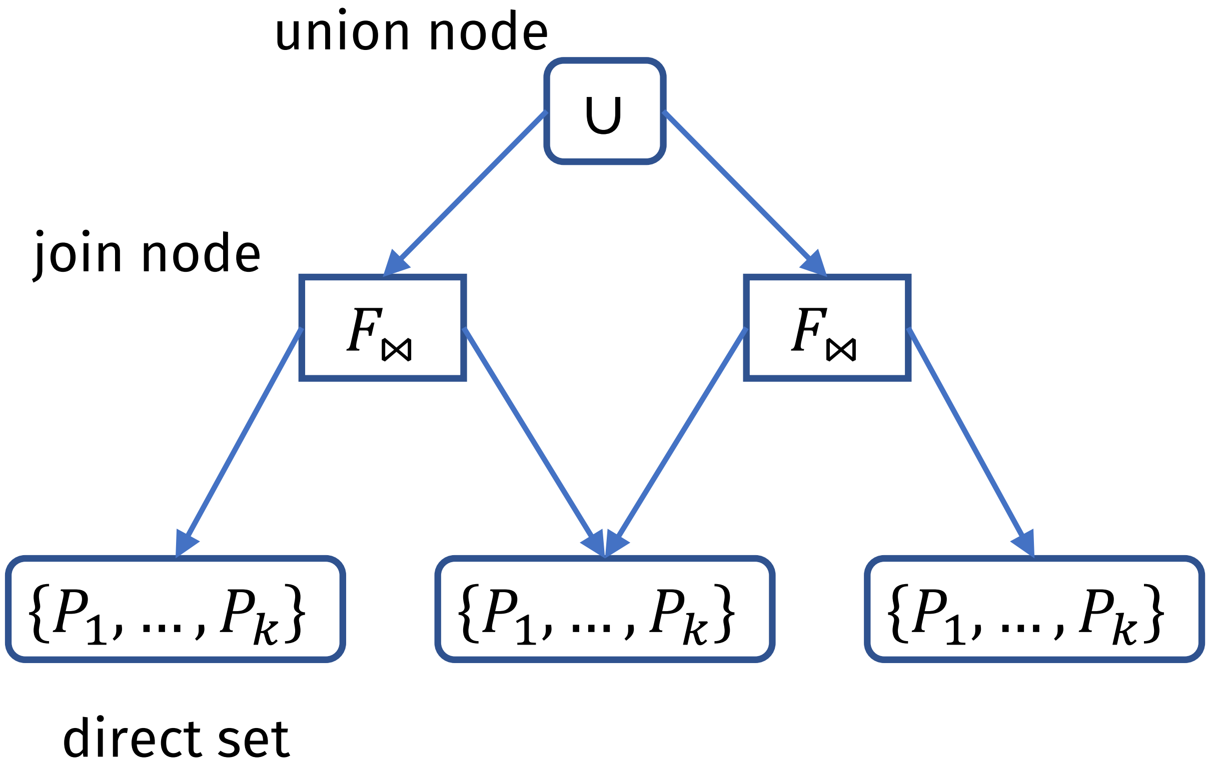
Version Space Algebra



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(T₁, ..., Tₙ)            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 2023" → "POPL'2023"
"Goodbye PLDI 2021" → "PLDI'2021"
```

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

```
concat(
  sub(pos(ws, Alpha), pos(Alpha, ws)),
  concat(
    cstr("'"),
    sub(pos(ws, digit), pos(digit, $))))
```

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 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

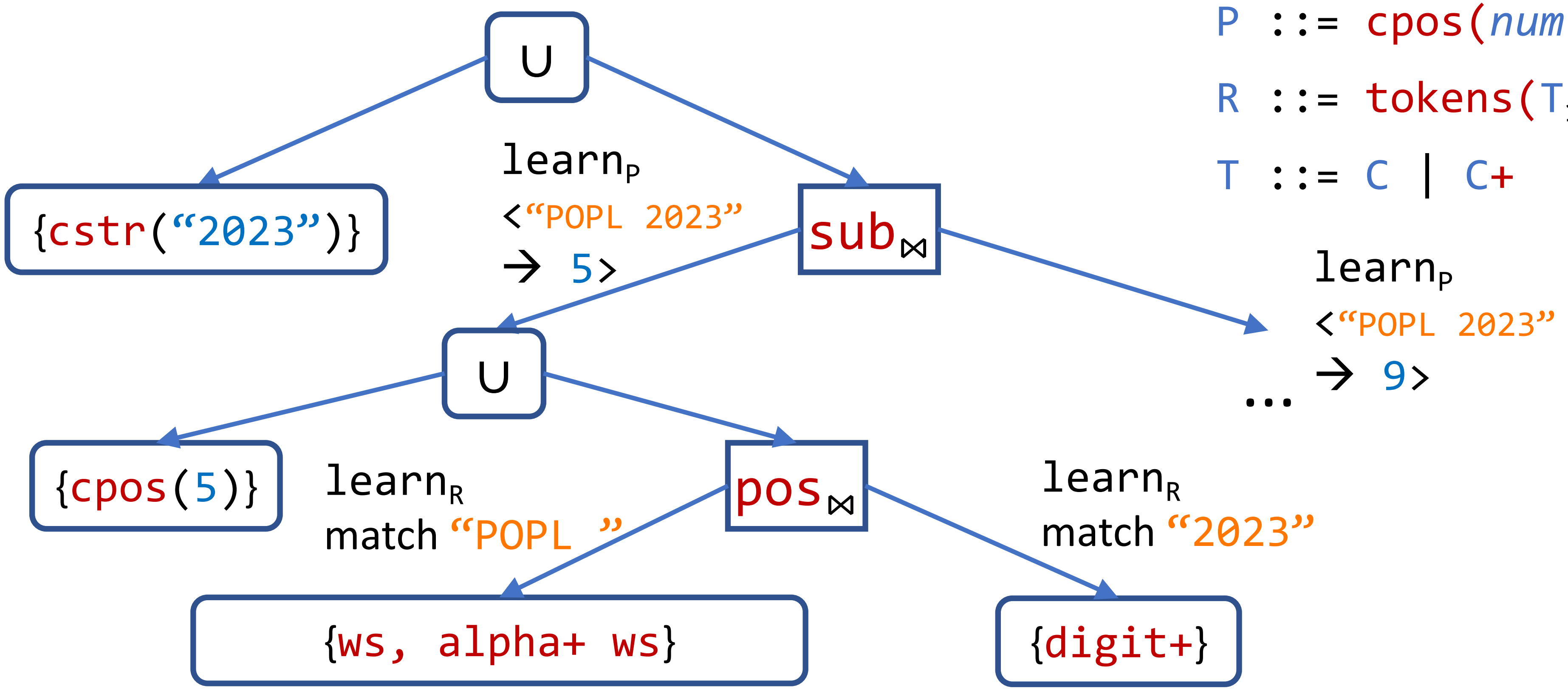
- $\texttt{learn}_F$ <"POPL 2023" → "2023">

F ::= cstr(*str*) | sub(P_1, P_2)

P ::= cpos(*num*) | pos(R_1, R_2)

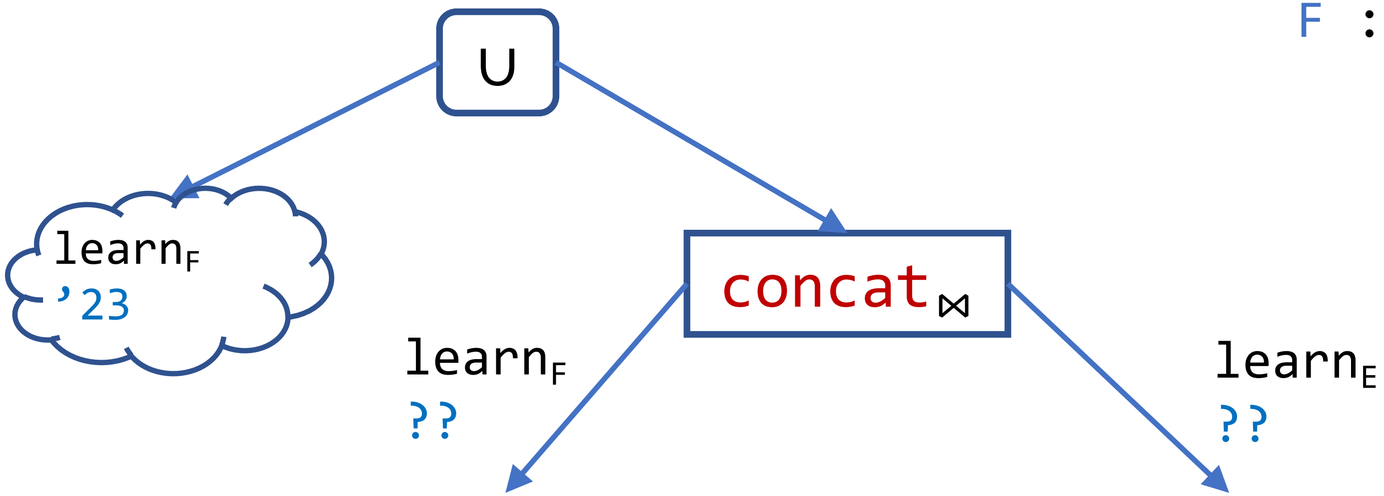
R ::= tokens(T_1, …, T_n)

T ::= C | C+



Learning trace expressions

- learn$_E$ <“POPL 2023” → “’23”>



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

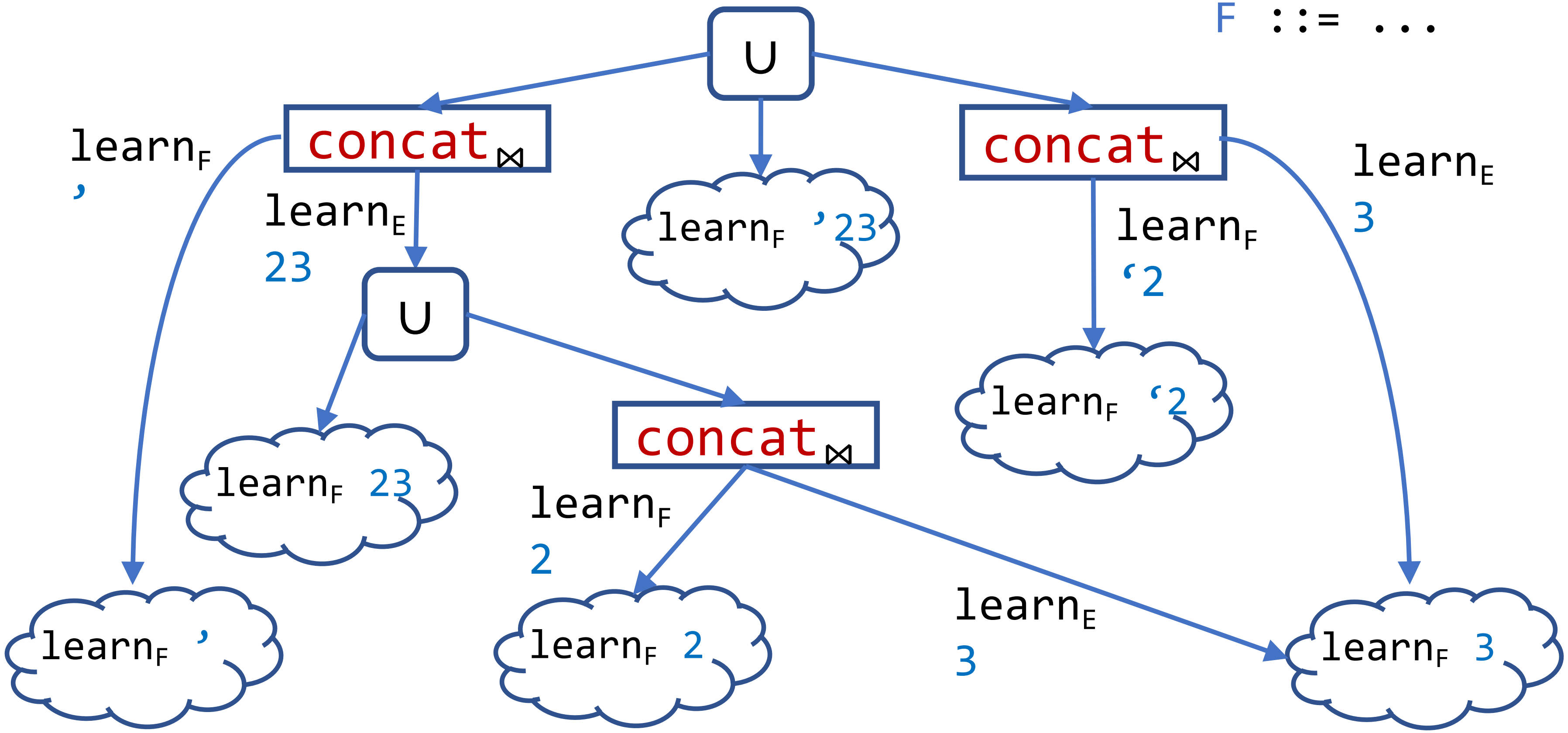
F ::= ...

Learning trace expressions

- learn_E <"POPL 2023" → "'23">

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

F ::= ...

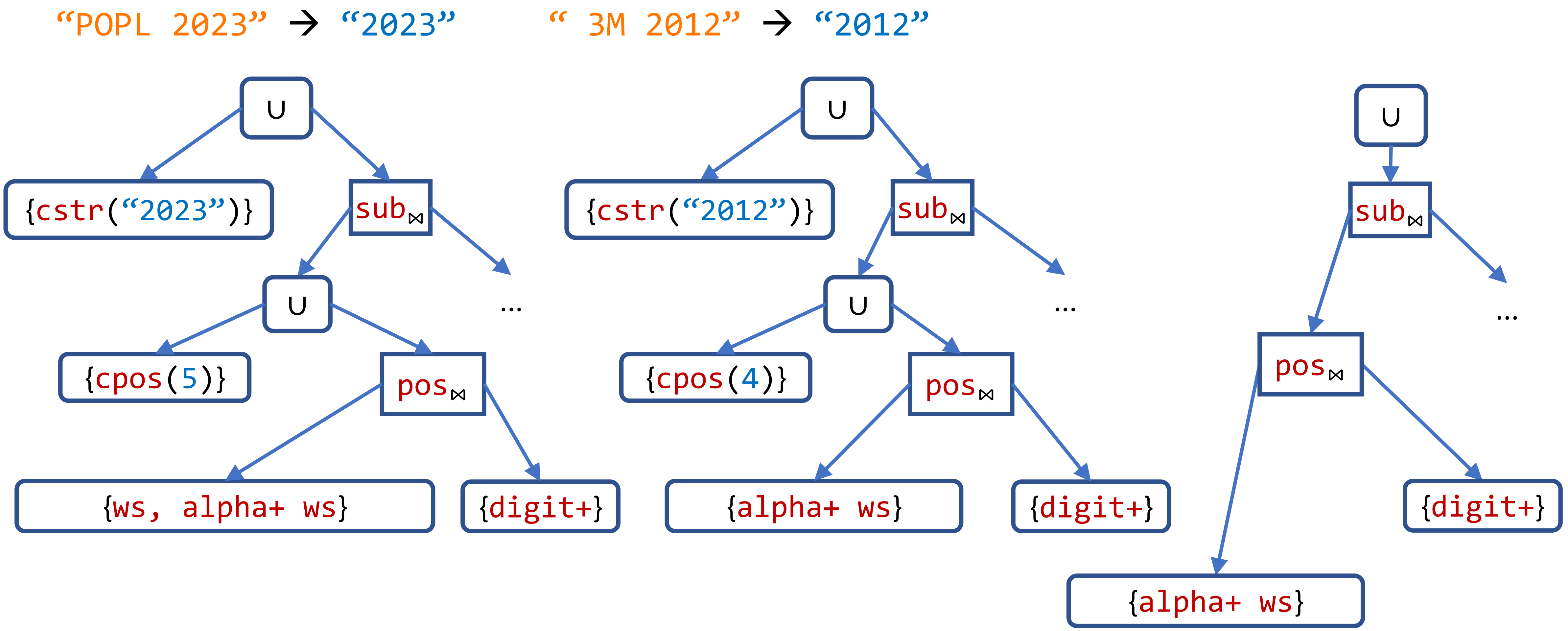


VSAs for Flashfill

- 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



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 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

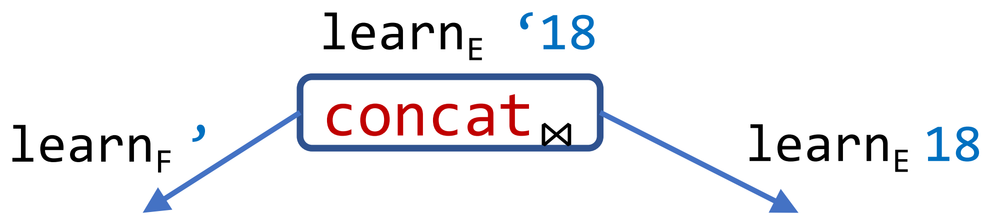
  - What about this language?

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]

Ranking function

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

Grammar

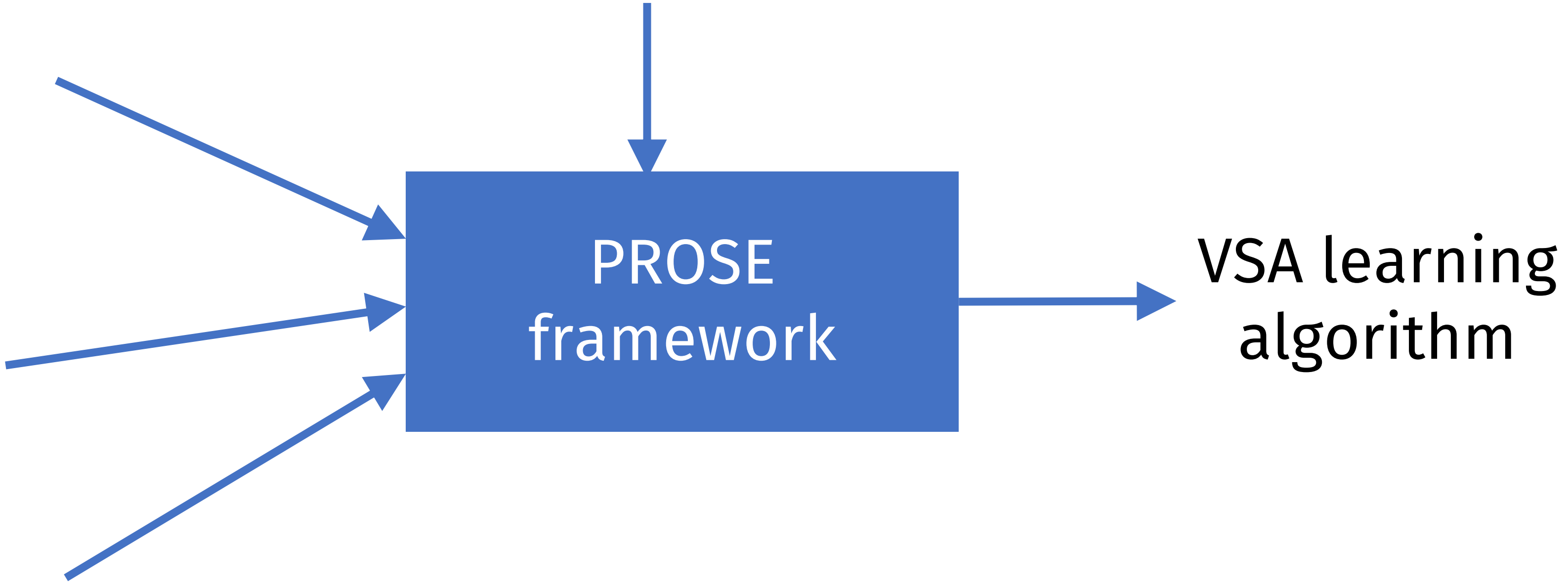
```
A ::= f(B, C) | ...
B ::= g(D, E) | ...
```

Semantics

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

Inverse semantics

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



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