Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions

Woosuk Lee, Hangyeol Cho

Hanyang University, South Korea

@ POPL'23

Recursive Functional Synthesis

```
type nat = Z
         | S of nat
```

f : nat -> nat satisfying

Z ⟼ Z,

S(Z) ⟼ S(S(Z))

using

(+) : nat * nat -> nat


```
let rec f x =
 match x with
   Z -> Z
 | S n -> (f n) + S(S(Z))
```

Recursive Functional Synthesis

Input 1: custom data types

```
type nat = Z
         | S of nat
```

`f : nat -> nat` satisfying

Z ⟼ Z,

S(Z) ⟼ S(S(Z))

using

`(+) : nat * nat -> nat`

Synthesizer

```
let rec f x =
 match x with
   Z -> Z
 | S n -> (f n) + S(S(Z))
```

Recursive Functional Synthesis

```
type nat = Z
         | S of nat
```

f : nat -> nat satisfying

$Z \longmapsto Z,$

$S(Z) \longmapsto S(S(Z))$

using

(+) : nat * nat -> nat

Synthesizer

```
let rec f x =
 match x with
   Z -> Z
 | S n -> (f n) + S(S(Z))
```

Input 2: a type signature of the target function and I/O examples

Recursive Functional Synthesis

```
type nat = Z
         | S of nat
```

f : nat -> nat satisfying

Z ⟼ Z,

S(Z) ⟼ S(S(Z))

using

(+) : nat * nat -> nat


```
let rec f x =
 match x with
   Z -> Z
 | S n -> (f n) + S(S(Z))
```

Input 3: library of external operators

Recursive Functional Synthesis

```
type nat = Z
         | S of nat
```

`f : nat -> nat` satisfying

`Z` $\longmapsto$ `Z`,

`S(Z)` $\longmapsto$ `S(S(Z))`

using

`(+) : nat * nat -> nat`

Synthesizer

```
let rec f x =
 match x with
   Z -> Z
 | S n -> (f n) + S(S(Z))
```

Recursive solution program

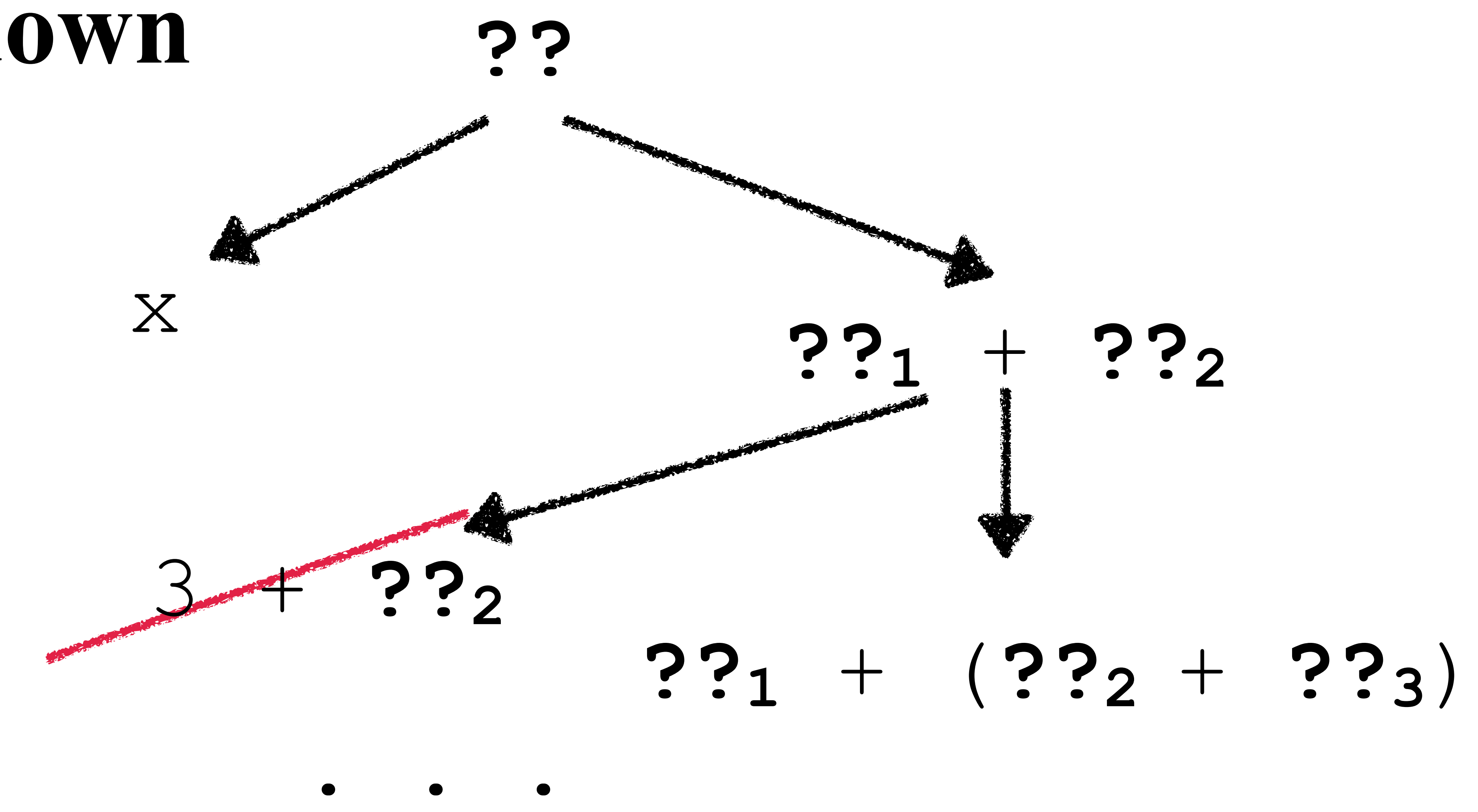
Recursive Functional Synthesis

- Long history [Summers 1977]
- Possible applications
  - End-user programming [Feser et al. 2015]
  - Invariant inference [Miltner et al. 2020]
  - Refactoring [Farzan et al. 2022]
  - …

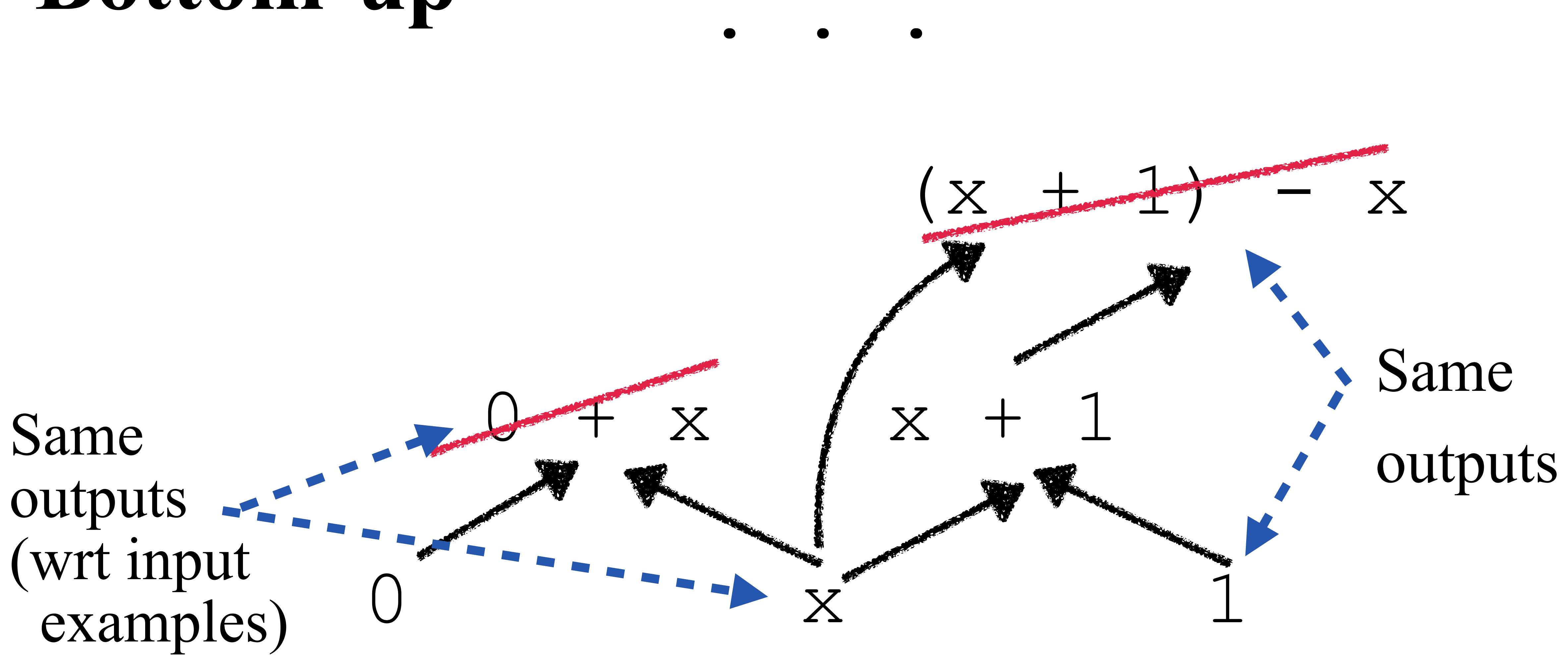
Two Strategies

`let rec f x = ?? (spec: 0 ⟼ 0, 1 ⟼ 2)`

Top-down



Bottom-up



- Starts from empty program, fills in holes
- Prune ***infeasible*** partial programs by domain-specific reasoning

- Builds larger programs from smaller ones
- Prune ***redundant*** subexpressions by evaluation (equivalence reduction)

Major Hurdle: Recursion

`let rec f x = ??` (spec: `0` ⟼ `0`, `1` ⟼ `2`)

Top-down

```
??
↓
. . .
↓
match x with
  Z -> 1 + ??
| S n -> f(n) + ??
```

Infeasible

???

To prune this candidate, we should approximate its possible behaviors, which is not easy due to recursion.

Bottom-up

```
. . .

???
f(x) + 1

x        1
```

To check if `f(x)+1` is redundant, we should evaluate it, which is impossible due to recursion.

Previous Approaches for Recursion

Top-down

- **Igor2** [Kitzelmann et al. 2006]: synthesize non-recursive programs first, and “fold” them
- **Myth** [Osera et al. 2015]: require all necessary behaviors of recursive calls as part of spec
- **SMyth** [Lubin et al. 2020]: forward + backward symbolic evaluation

Bottom-up

- **Escher** [Albarghouthi et al. 2013]: same as **Myth**
- **Burst** [Miltner et al. 2022]: repeatedly making and refuting assumptions over recursive calls (like CDCL)

Previous Approaches for Recursion

Unscalable

Top-down

- **Igor2** [Kitzelmann et al. 2006]: synthesize non-recursive programs first, and "fold" them

Burden on the user

- **Myth** [Osera et al. 2015]: require all necessary behaviors of recursive calls as part o

Slow when reasoning fails

- **SMyth** [Lubin et al. 2020]: forward + backward symbolic evaluation

Bottom-up

Burden on the user

- **Escher** [Albarghouthi et al. 2013]: same as **Myth**
- **Burst** [Miltner et al. 2022]: repeatedly m king and refuting ass lls (fi

Slow when too many backtracking

Our Contributions

- A novel and general method for synthesis of recursive programs
  - *Block-based pruning* (for handling recursion)
  - *Library sampling* (for handling library w/o complex domain-specific reasoning)
- Soundness & completeness guaranteed
- Tool (Trio) that outperforms the state-of-the-art
  https://github.com/pslhy/trio

Illustrative Example

Synthesize the double function

```
type nat = Z
         | S of nat
```

f : nat -> nat satisfying

```
Z ⟼ Z,
S(Z) ⟼ S(S(Z))
S(S(Z)) ⟼ S(S(S(S(Z))))
```

using

```
(+) : nat * nat -> nat
```

Synthesizer

```
let rec f x =
 match x with
   Z -> Z
 | S n -> (f n) + S(S(Z))
```

Illustrative Example

Synthesize the double function

```
type nat = Z
         | S of nat
```

`f : nat -> nat` satisfying

$0 \longmapsto 0,$

$1 \longmapsto 2,$

$2 \longmapsto 4$

using

`(+) : nat * nat -> nat`


```
let rec f x =
 match x with
   Z -> Z
 | S n -> (f n) + 2
```

Shortly,

`0 = Z, 1 = S(Z), 2 = S(S(Z)),...`

Our Key Idea: Two Phased Synthesis

We call them ***blocks***

(1) Synthesize all possible **recursion- and conditional-free expressions satisfying each I/O example**

(2) During top-down search for a recursive solution, prune candidates ***inconsistent* with the *blocks*** obtained in the previous step

Step 1: Synthesizing Blocks

I/O Example	Synthesized Blocks
`0` $\longmapsto$ `0`	`0, x, 0+0, 0+x, x+0, x+x, …`
`1` $\longmapsto$ `2`	`2, 1+1, 0+2, 2+0, x+1, 1+x, x+x, …`
`2` $\longmapsto$ `4`	`4, 1+3, 2+2, 3+1, x+2, 2+x, x+x, …`

Step 2: Top-Down Search w/ Block-based Pruning

Suppose we want to check feasibility of this partial program.

```
let rec f (x) = ??
        ↓
      . . .
        ↓
let rec f (x) =
  match x with
    Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

Step 2: Top-Down Search w/ Block-based Pruning (1/4)

Blocks for I/O example 1

I/O Example	Synthesized Blocks
`0 ⟼ 0`	`0, x, 0+0, 0+x, x+0, x+x, …`

Matched!

```
match x with
  Z -> 0+??
| S n -> 3+f(n)+??
```

Partial eval. →

```
match Z with
  Z -> 0+??
| S n -> 3+f(n)+??
```

Partial eval. →

```
0+??
```

Meaning: there exists a completion of the partial program that satisfies I/O example 1.

Step 2: Top-Down Search w/ Block-based Pruning (2/4)

Blocks for I/O example 2

I/O Example	Synthesized Blocks
`1` ⟼ `2`	`2, 1+1, 0+2, 2+0, x+1, 1+x, x+x,` …

```
match x with
  Z -> 0+??
| S n -> 3+f(n)+??
```

Partial eval. →

```
match S(Z) with
  Z -> 0+??
| S n -> 3+f(n)+??
```

Partial eval. →

Step 2: Top-Down Search w/ Block-based Pruning (3/4)

Blocks for I/O example 2

I/O Example	Synthesized Blocks
`1 ⟼ 2`	`2, 1+1, 0+2, 2+0, x+1, 1+x, x+x, …`

Partial eval. → `3+f(Z)+??` → Partial eval. →

```
3 +
(match Z with
    Z -> 0+??
  | S n -> … ) + ??
```

→ Partial eval. → `3 + 0 + ?? + ??`

No matched block!

(∵ `3+0+??+??` > 2 but the desired output is 2)

Meaning: there is no completion of the partial program that satisfies I/O example 2.

Step 2: Top-Down Search w/ Block-based Pruning (4/4)

The candidate is discarded since no completion of it satisfies the second I/O example.

```
let rec f (x) =
  match x with
    Z -> 0 + ??
  | S n -> 3 + f(n) + ??
```

Challenges

Challenge 1:
how to efficiently synthesize all the blocks in
an enormous amount?

Our two-phased synthesis

(1) Synthesize *all possible* recursion- and conditional-free expressions (i.e., straight line code) satisfying each I/O example (called ***blocks***)

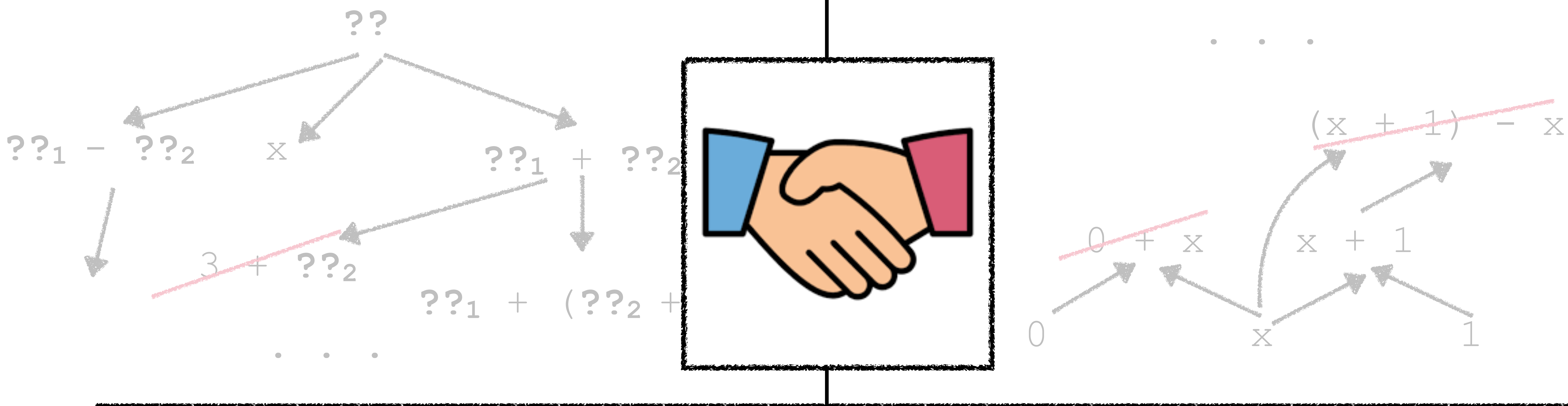
(2) During top-down search for a recursive solution, *prune candidates inconsistent* with the blocks

Challenge 2: how to efficiently
check the consistency with many blocks?

Key to Scalability 1: Top-Down + Bottom-Up

- Adapted the previous bidirectional search strategy† to our setting
- With *version spaces*, we can synthesize 10^{10} blocks within 0.1 sec!

Top-down **Bottom-up**



Can quickly synthesize all possible blocks (address **challenge 1**)

†Woosuk Lee, Combining the Top-Down Propagation and Bottom-Up Enumeration for Inductive Program Synthesis, POPL'21

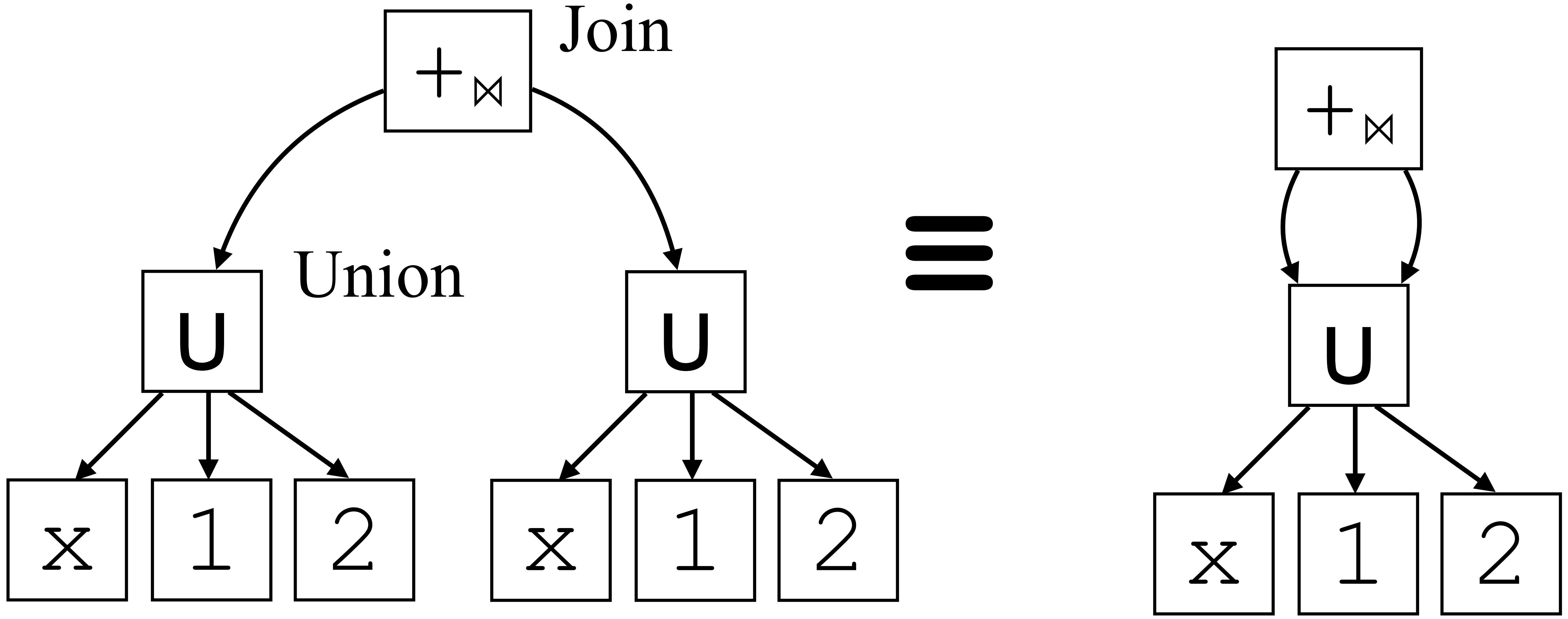
Key to Scalability 2: Version Spaces

Explicit set

```
{ 1+1, 1+x, 1+2,
  x+1, x+x, x+2,
  2+1, 2+x, 2+2 }
```

≡

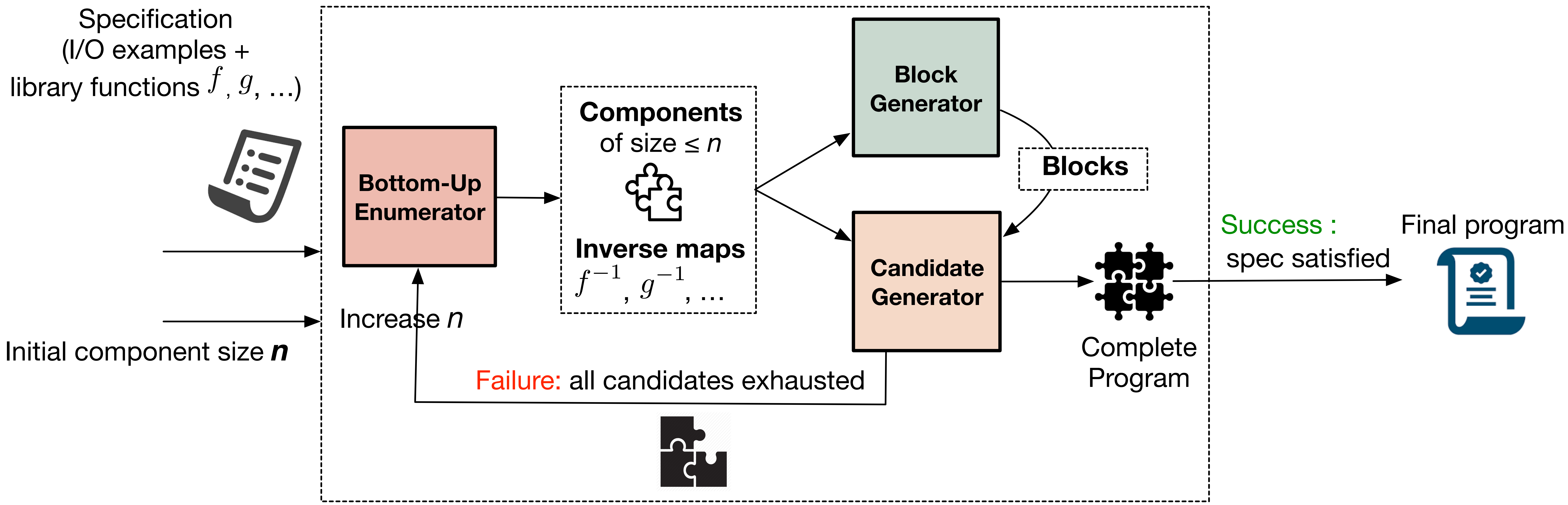
Version space



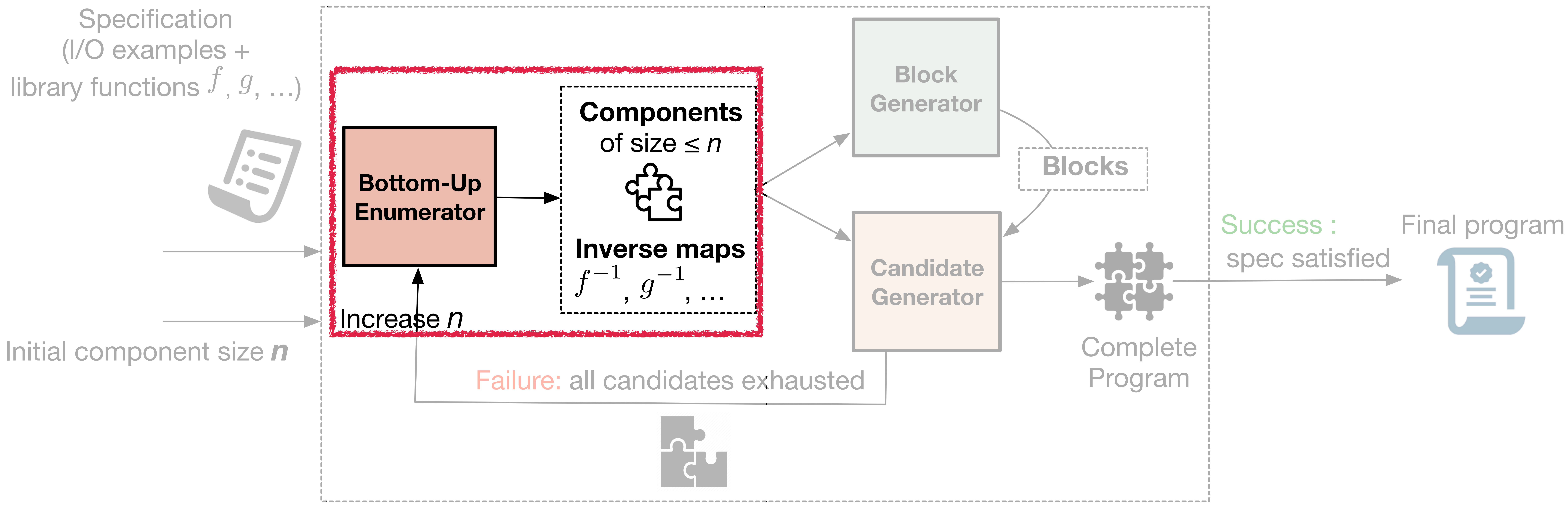
#. of nodes = O(log #. of programs)

Compactly represent a large set of blocks (address **challenge 2**)

Our Trio System



Bottom-Up Enumerator



Generation of Components

- Components = sub-expressions that may be used in a solution
- Components of size ≤ n are generated
- Suppose we obtain the following component set:

$$\mathbf{C} = \{\ \texttt{x},\ 0,\ 1,\ 2,\ \texttt{x} + 1\ \}$$

Library Sampling: Generation of Inverse Maps (1/2)

- Inverse map: output → inputs of a library function

$$+^{-1}(0) = \{(0,0)\}, \qquad +^{-1}(1) = \{(0,1),(1,0)\},$$

$$+^{-1}(2) = \{(0,2),(2,0),(1,1)\}, \qquad +^{-1}(3) = \{(1,2),(2,1)\},$$

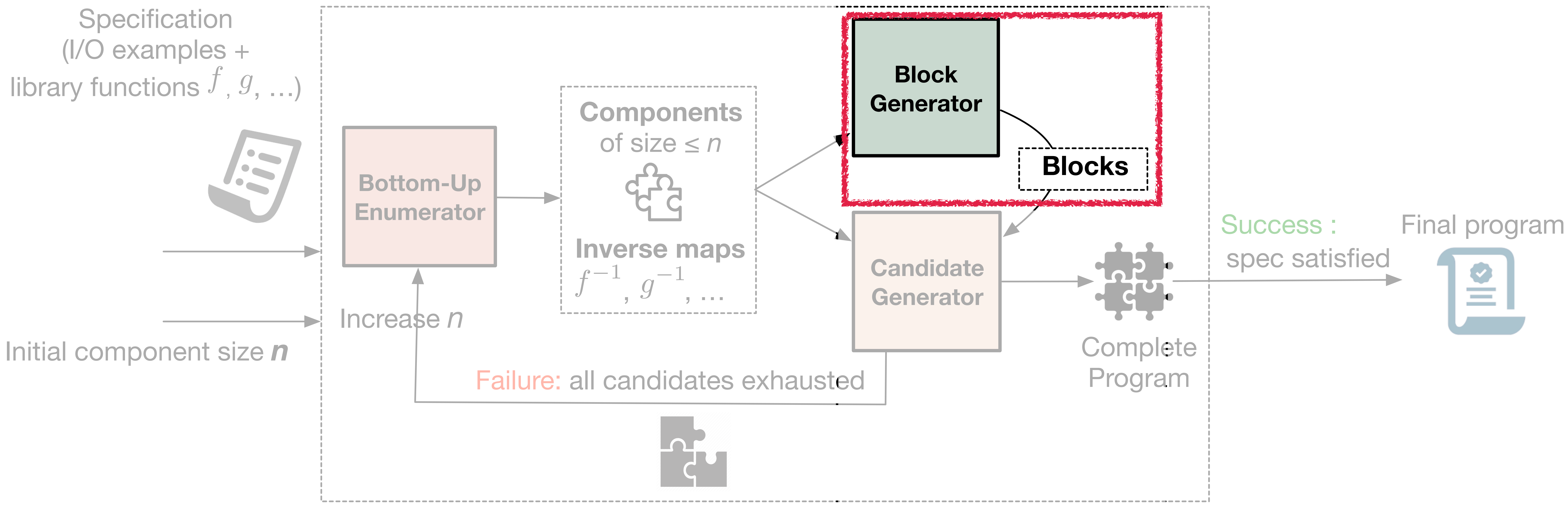
$$+^{-1}(4) = \{(2,2)\}$$

- From input-output samples of library functions

Library Sampling: Generation of Inverse Maps (1/2)

- Inputs for sampling: values NOT greater than the "maximum" input example
  - e.g., use (0,0), (0,1), …, (2,2) when spec is $\{ 0 \mapsto 0, 2 \mapsto 4 \}$
  - Reason: we target *structurally recursive programs* where arguments of recursive calls are strictly decreasing (to guarantee termination of synthesized programs)

Block Generator



Generation of Blocks

- For each I/O example, we generate satisfying blocks.
- Each set of blocks is represented by a *version space*.

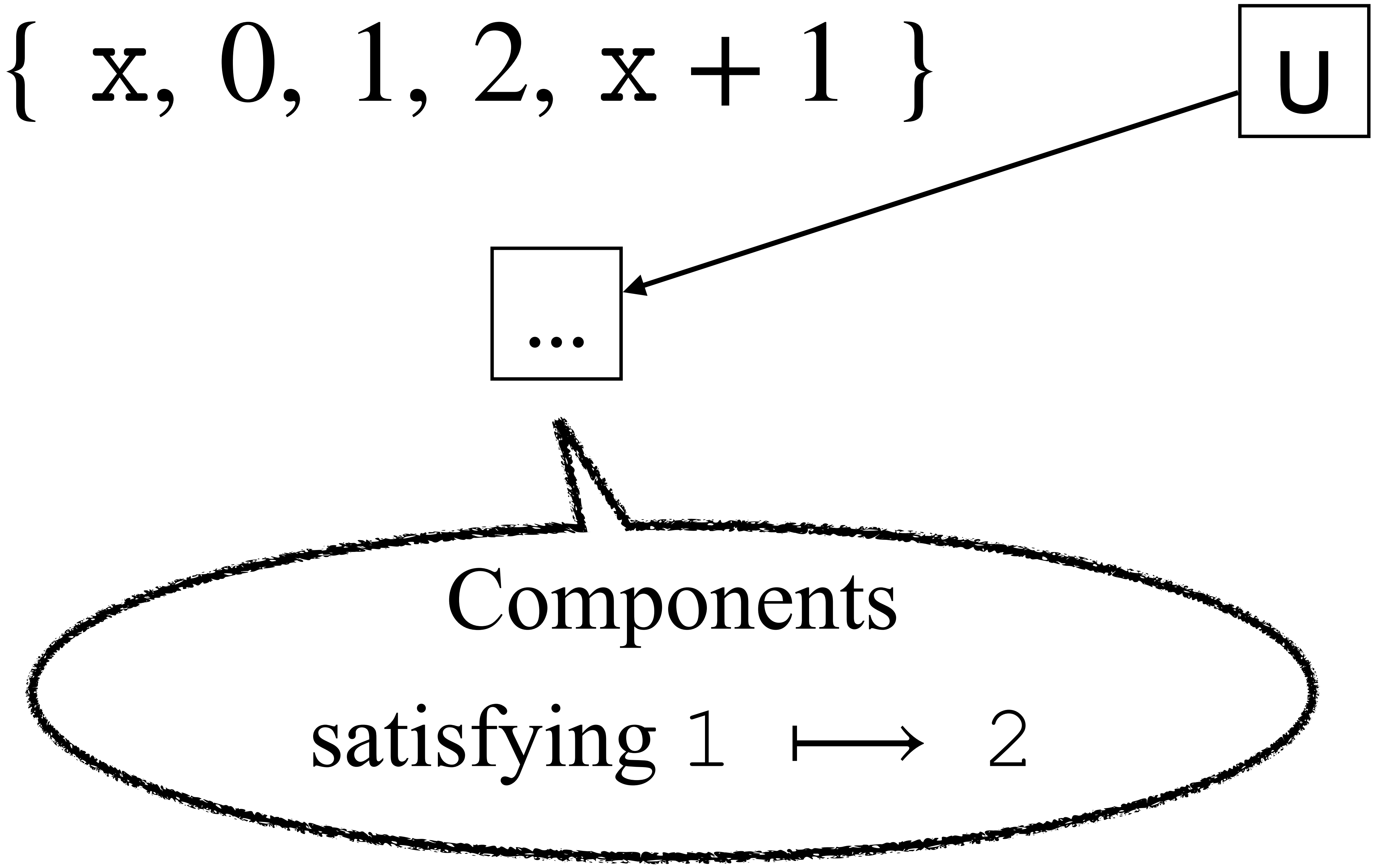
Generation of Blocks for I/O example `1` $\longmapsto$ `2`

C = { `x`, 0, 1, 2, `x` + 1 } $\boxed{\cup}$

We will generate
multiple blocks.

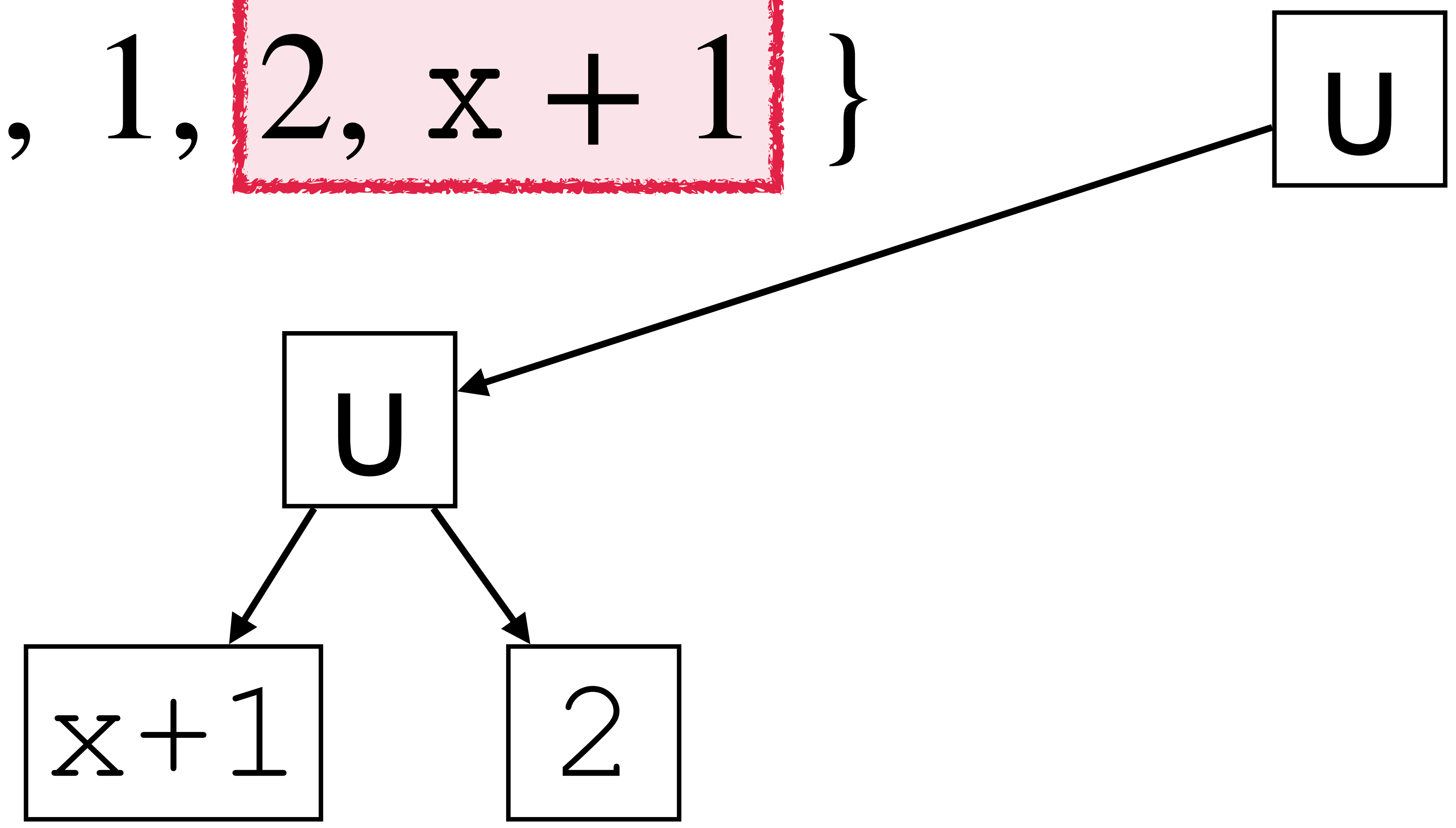
Generation of Blocks for I/O example `1` $\longmapsto$ `2`

C = { `x`, 0, 1, 2, `x` + 1 }



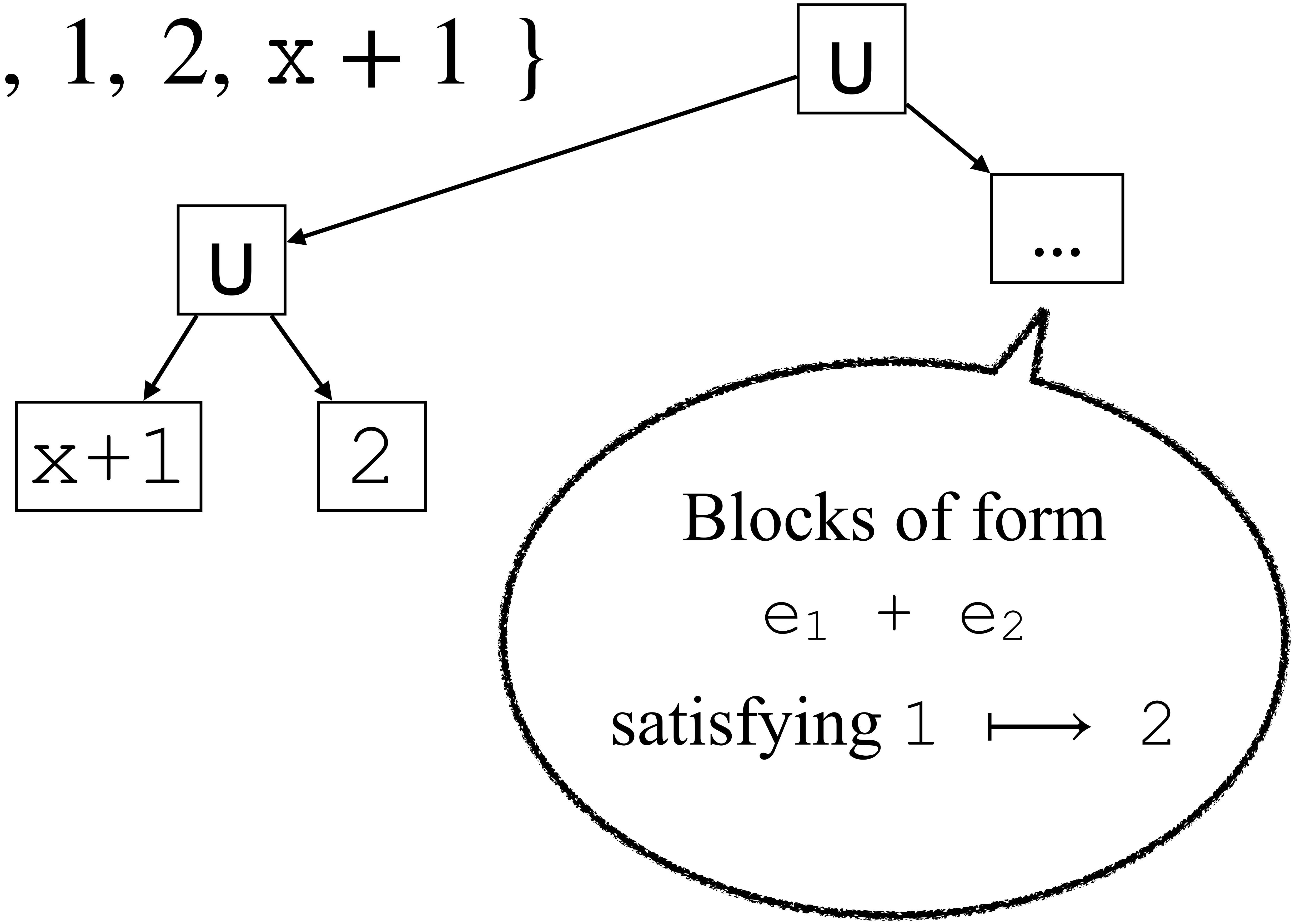
Generation of Blocks for I/O example $1 \longmapsto 2$

$C = \{ x, 0, 1, 2, x+1 \}$



Generation of Blocks for I/O example $1 \longmapsto 2$

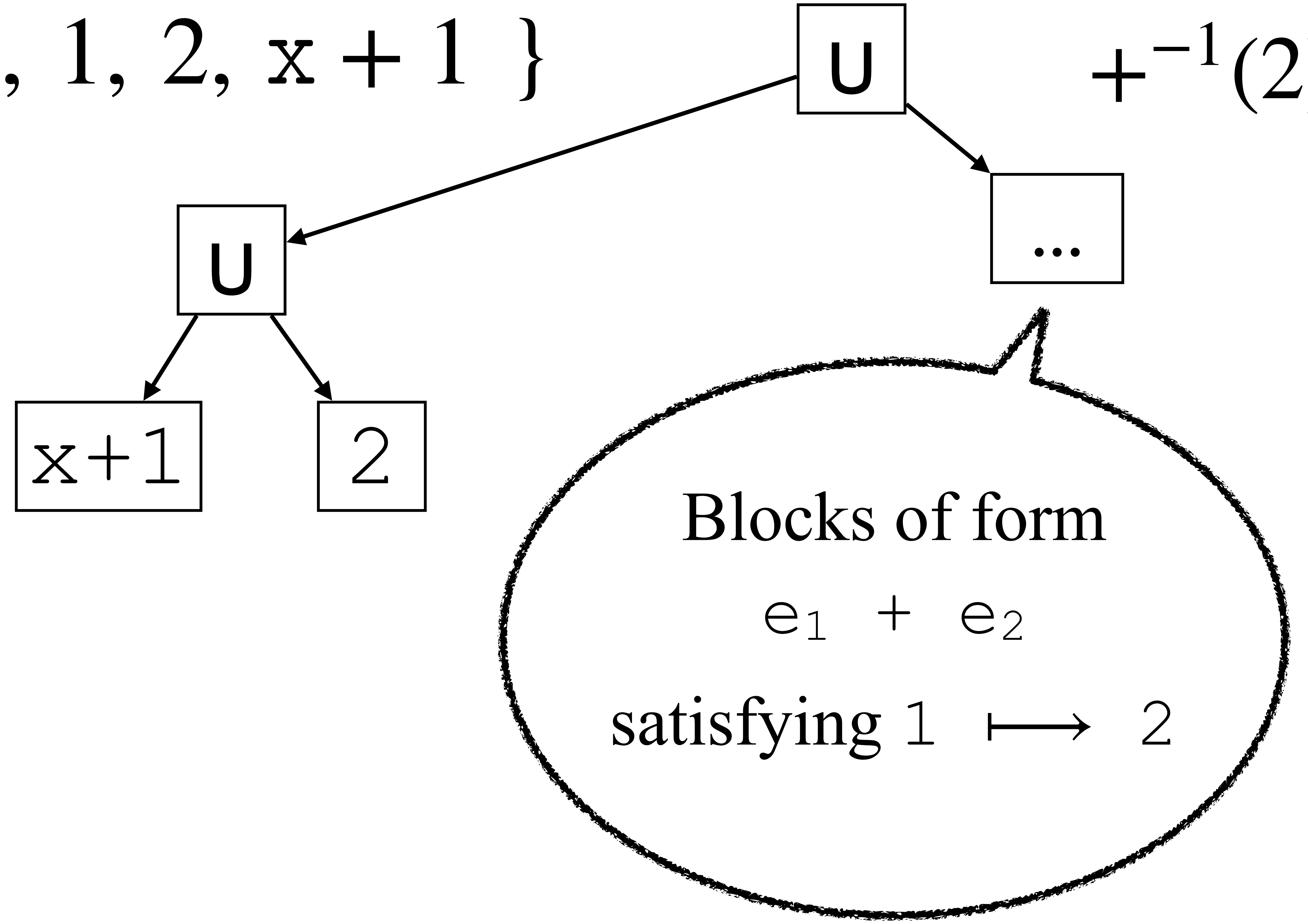
C = { x, 0, 1, 2, x + 1 }



Generation of Blocks for I/O example `1` $\longmapsto$ `2`

C = { x, 0, 1, 2, x + 1 }

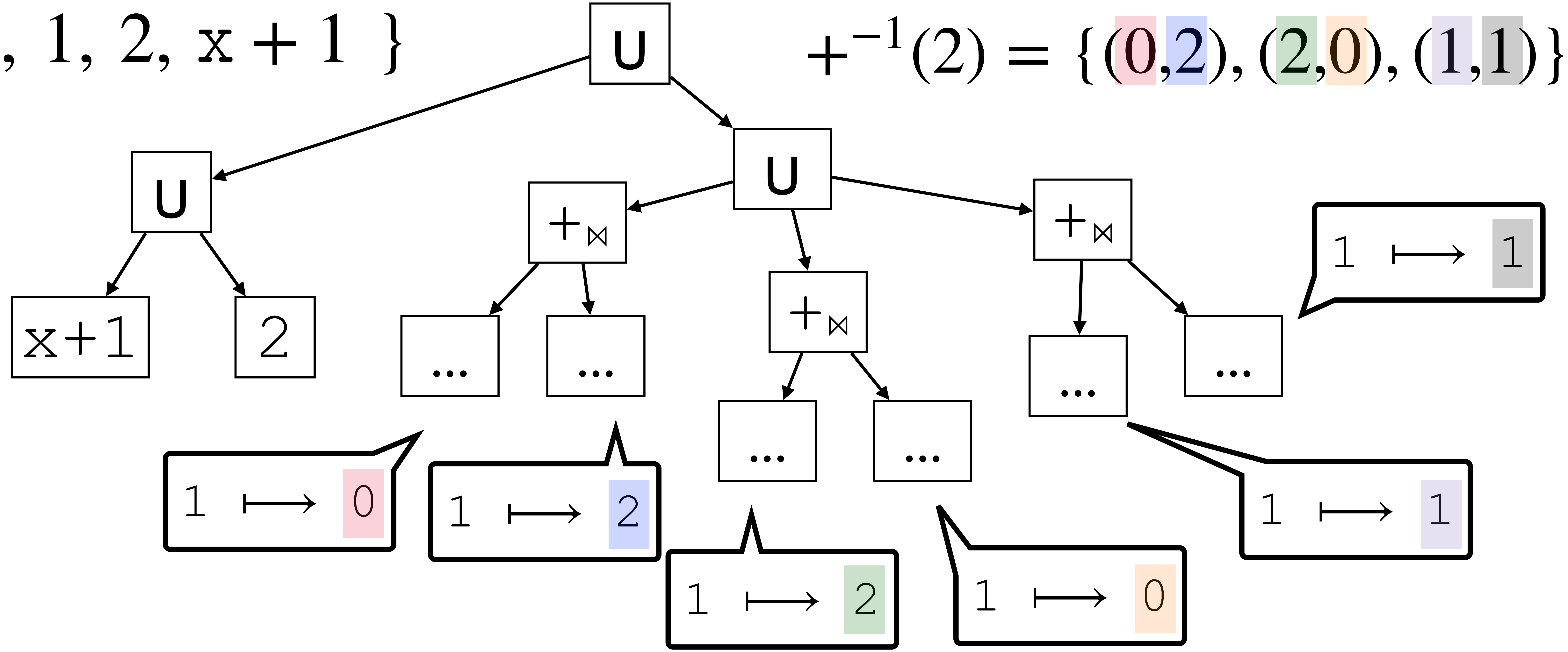
$+^{-1}(2) = \{(0,2), (2,0), (1,1)\}$



Generation of Blocks for I/O example $1 \longmapsto 2$

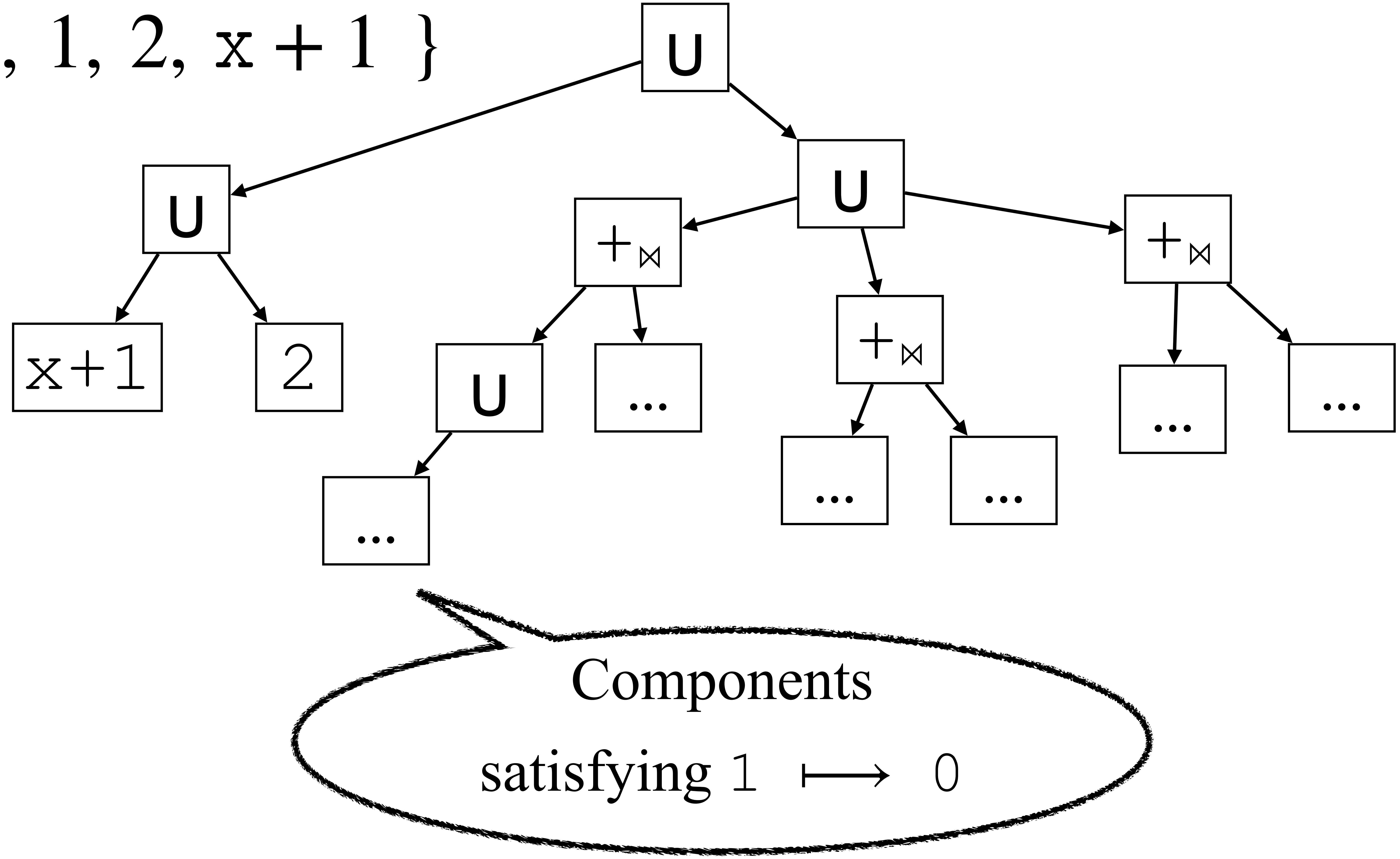
$\mathbf{C} = \{\ \mathtt{x},\ 0,\ 1,\ 2,\ \mathtt{x}+1\ \}$

$+^{-1}(2) = \{(0,2), (2,0), (1,1)\}$



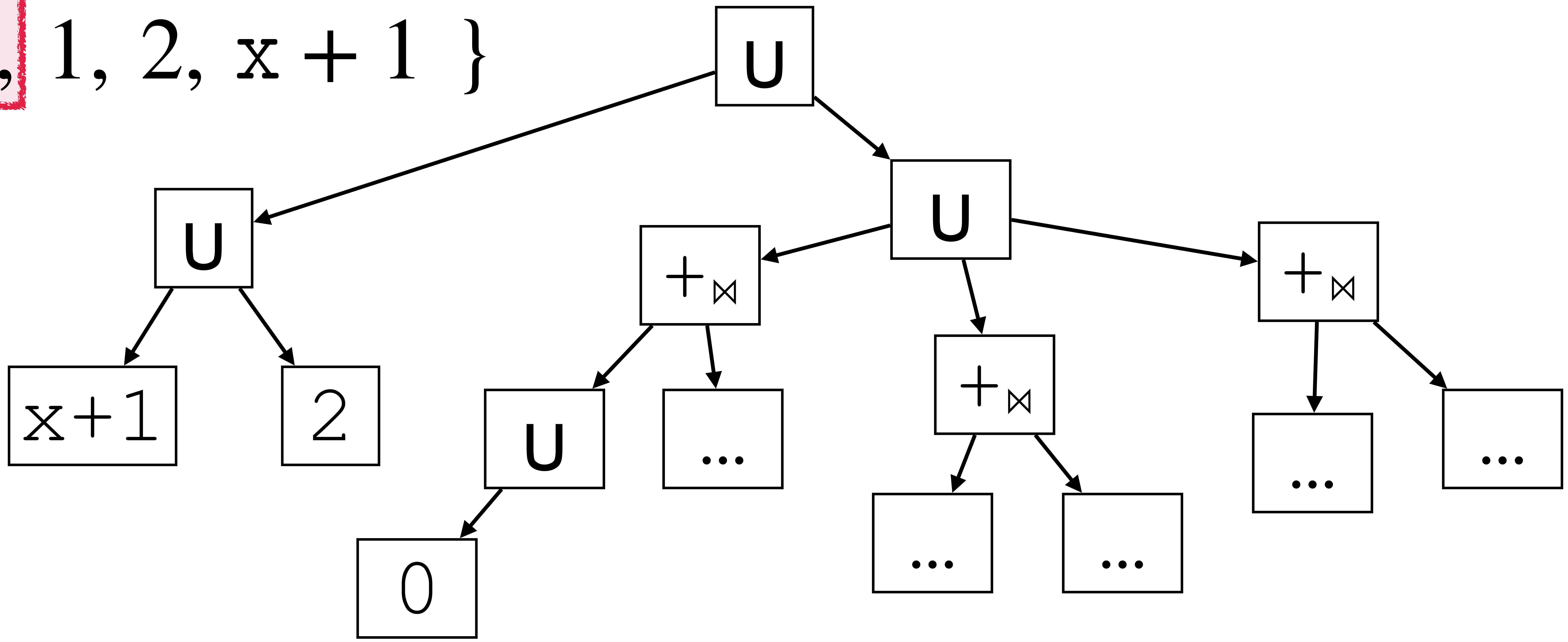
Generation of Blocks for I/O example $1 \longmapsto 2$

C = { x, 0, 1, 2, x + 1 }



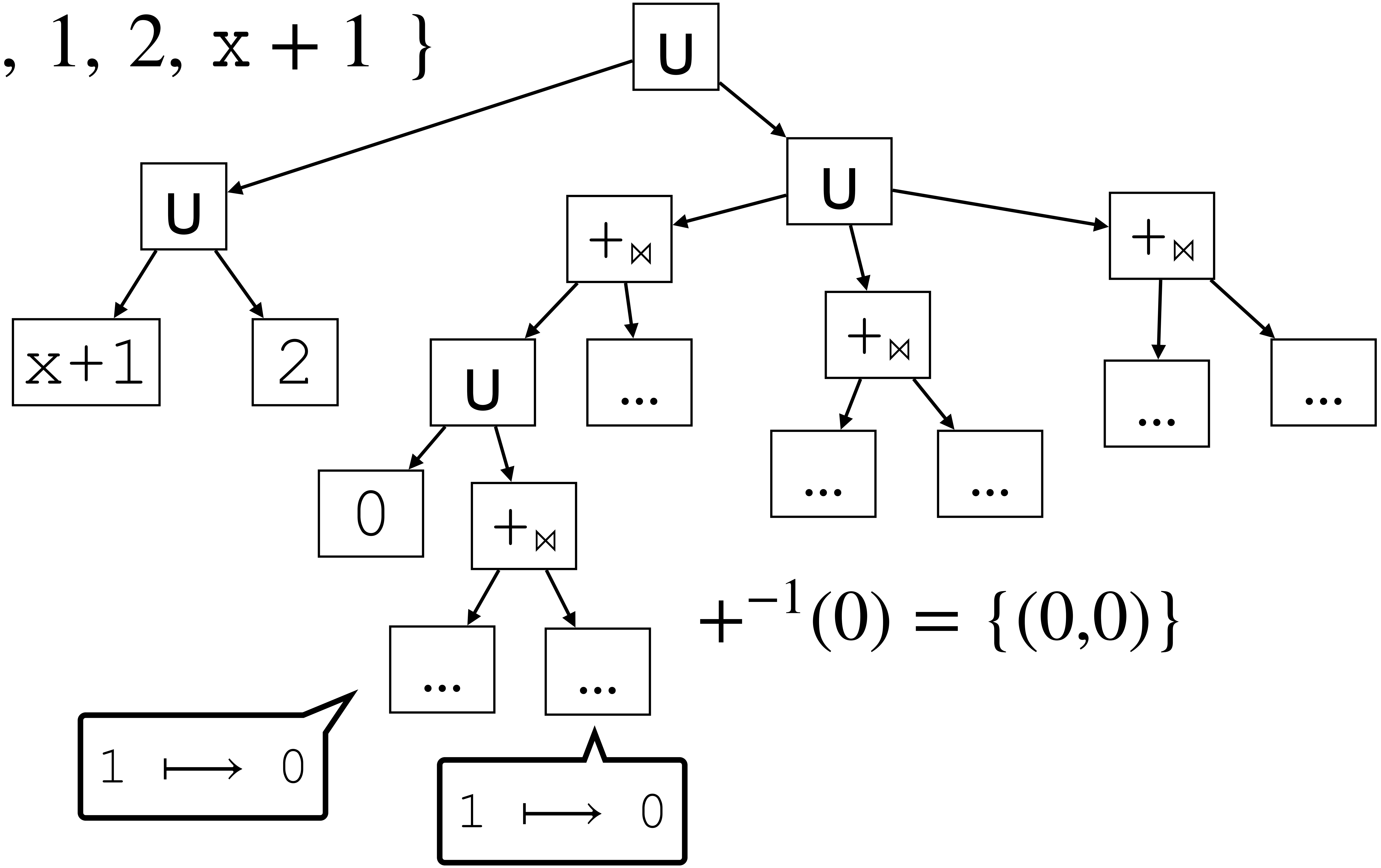
Generation of Blocks for I/O example $1 \longmapsto 2$

$C = \{\ x, 0, 1, 2, x + 1\ \}$



Generation of Blocks for I/O example $1 \longmapsto 2$

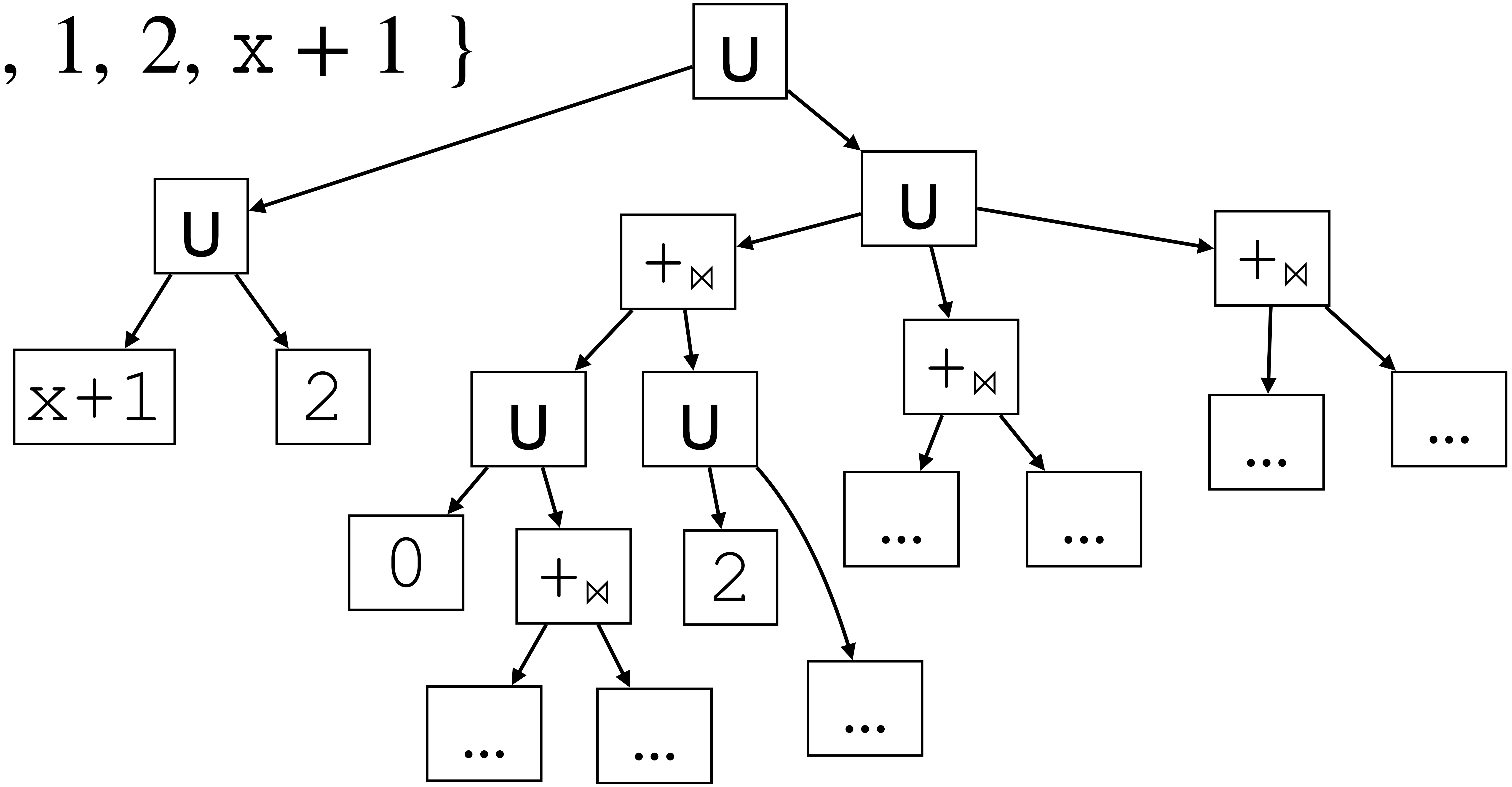
$\mathbf{C} = \{\ \mathtt{x},\ 0,\ 1,\ 2,\ \mathtt{x} + 1\ \}$



$+^{-1}(0) = \{(0,0)\}$

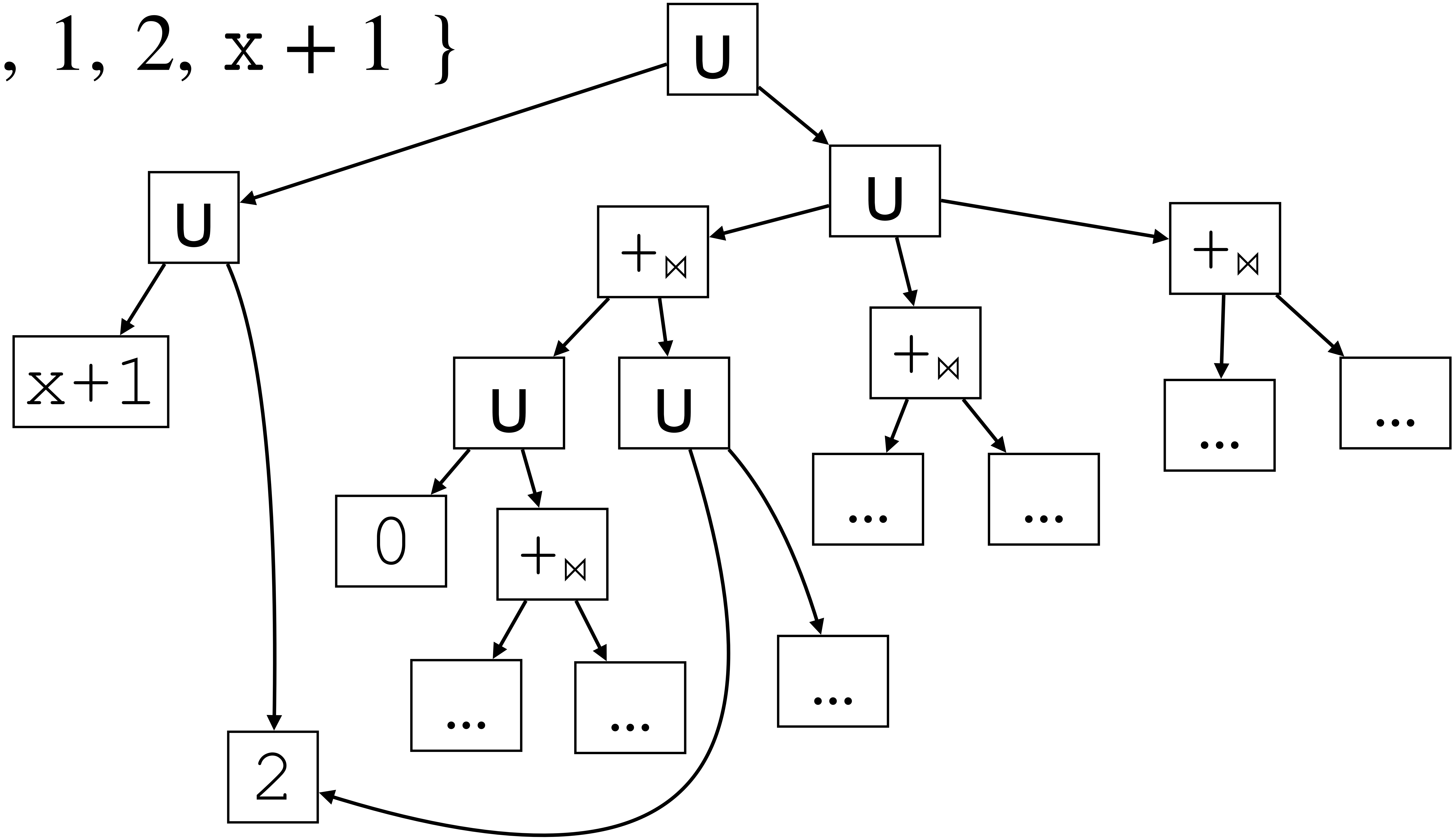
Generation of Blocks for I/O example $\texttt{1} \longmapsto \texttt{2}$

C = { x, 0, 1, 2, x + 1 }



Generation of Blocks for I/O example $\texttt{1} \longmapsto \texttt{2}$

C = { x, 0, 1, 2, x + 1 }



Candidate Generator

Top-Down Search for Recursive Programs

Suppose we want to check feasibility of this partial program.

```
let rec f (x) = ??

       ↓
     . . .
       ↓

let rec f (x) =
  match x with
    Z -> 0
  | S x' -> f(x') + ??
```

Search for Recursive Programs

For I/O example $1 \longmapsto 2$

```
match x with
  Z -> 0
| S x′ -> f(x′) + ??
```

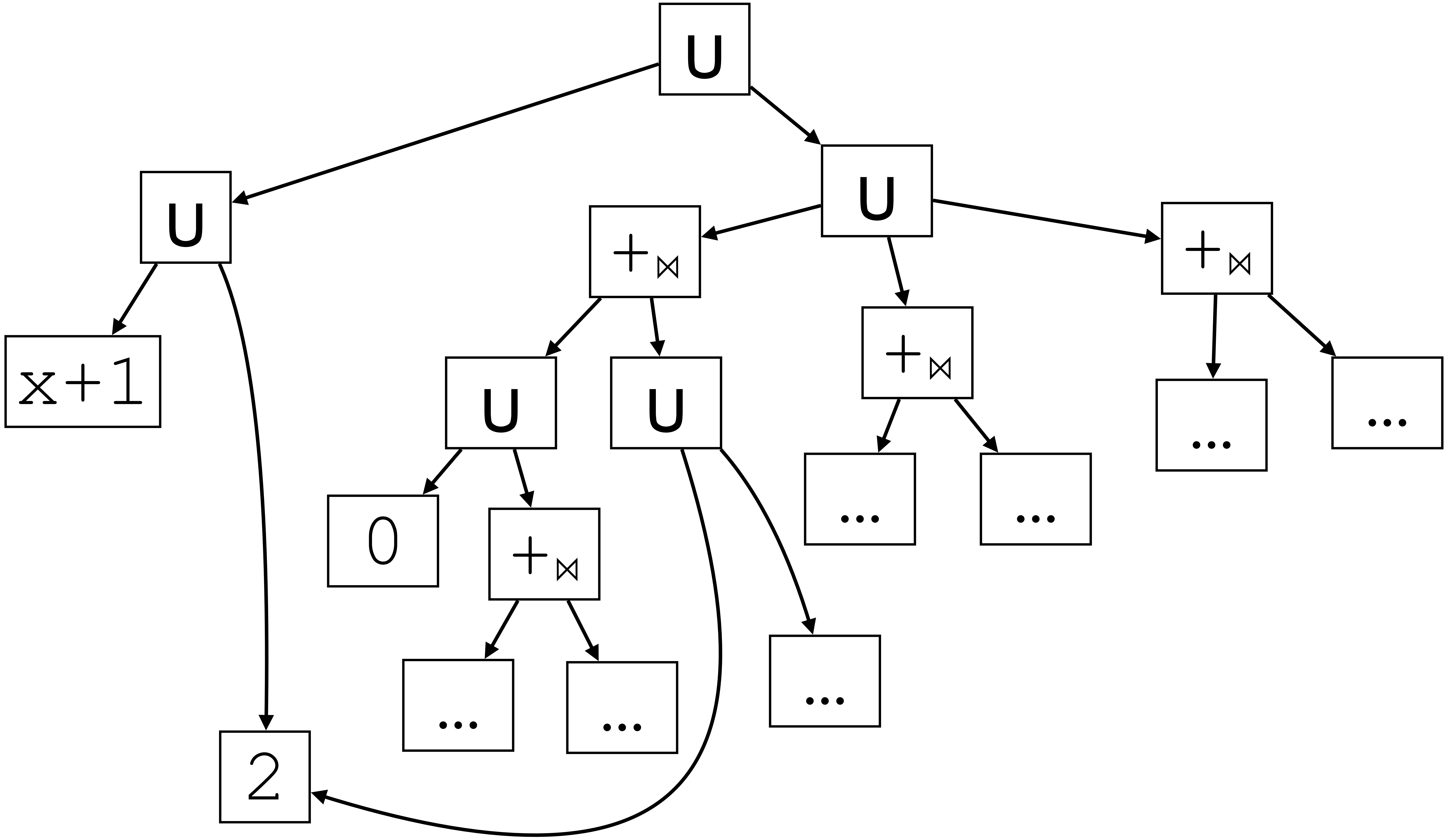
Partial eval*

→ `0 + ??`

Version Space Matching

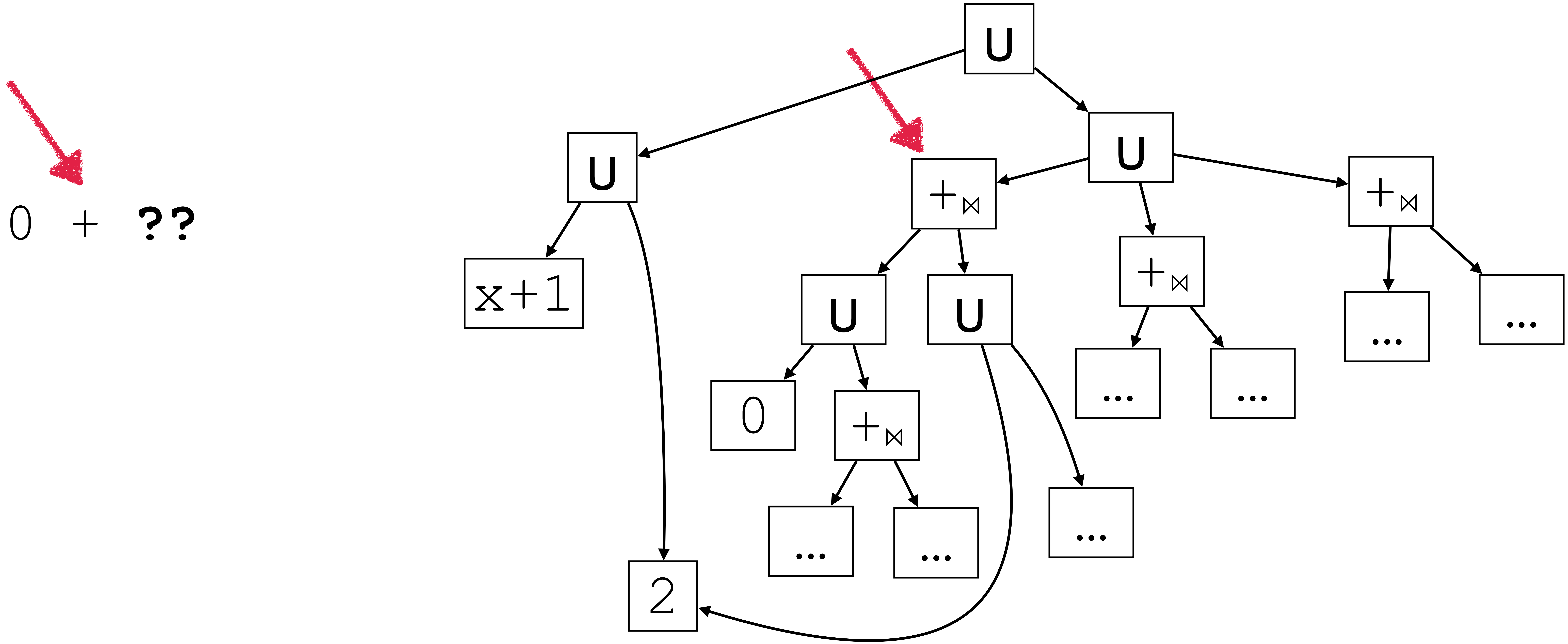
Check if `0 + ??` can be *matched with* any block satisfying `1` $\longmapsto$ `2`

`0 + ??`



Version Space Matching

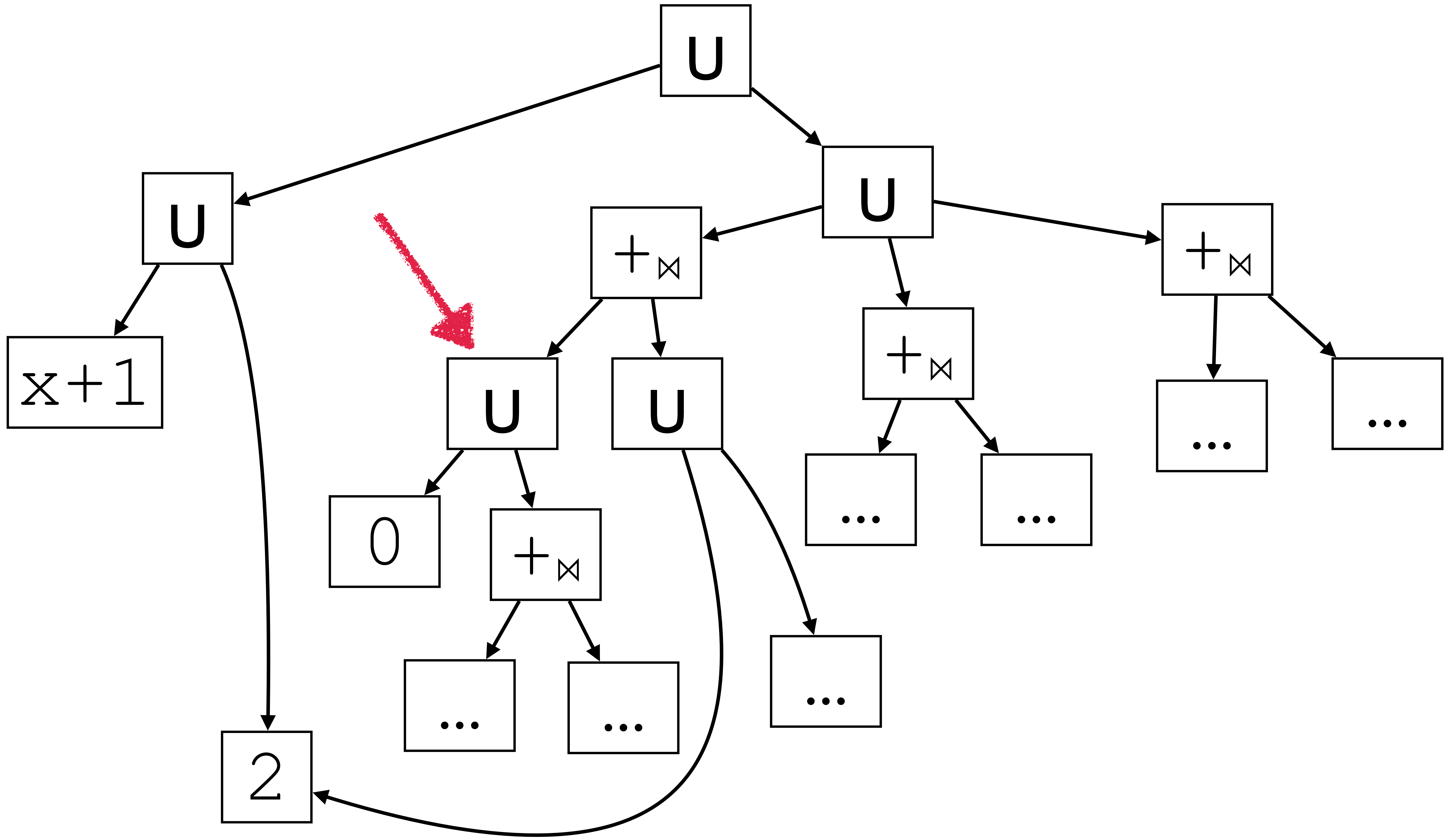
Check if `0 + ??` can be *matched with* any block satisfying `1` $\longmapsto$ `2`



Version Space Matching

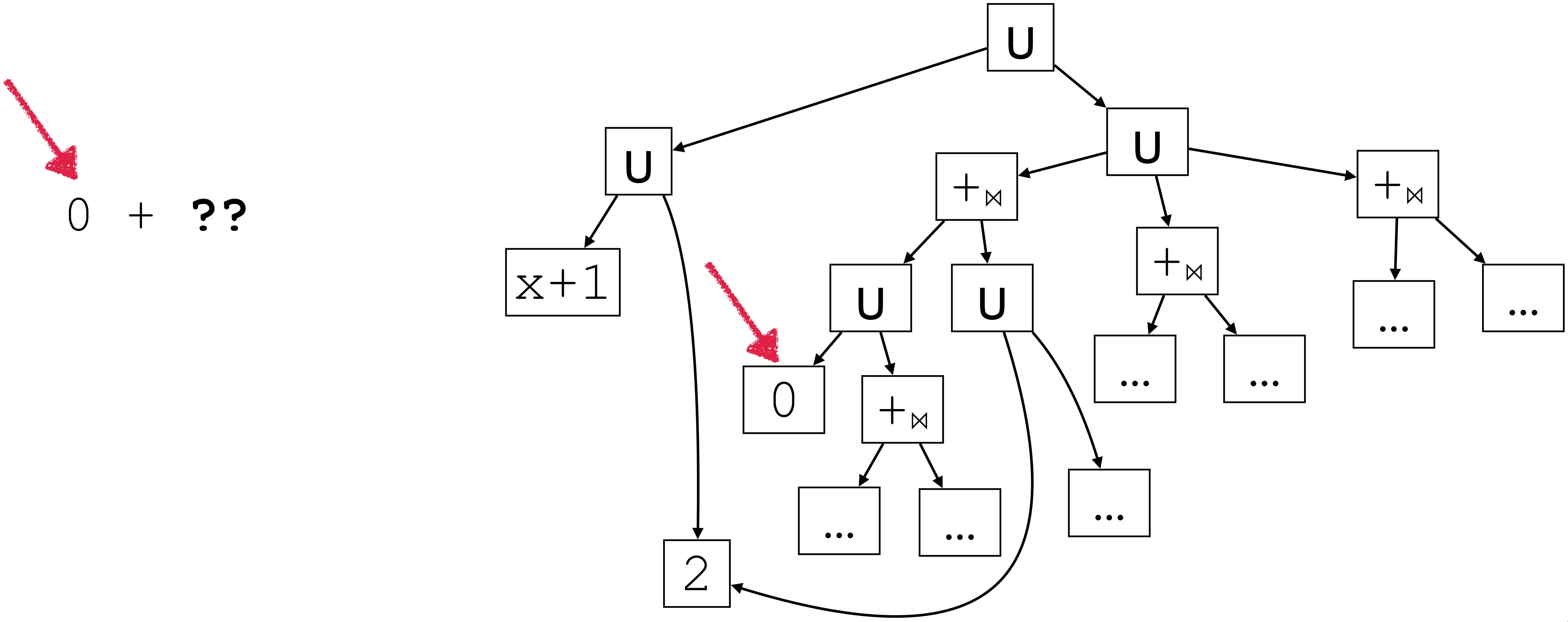
Check if `0 + ??` can be *matched with* any block satisfying `1` $\longmapsto$ `2`

`0 + ??`



Version Space Matching

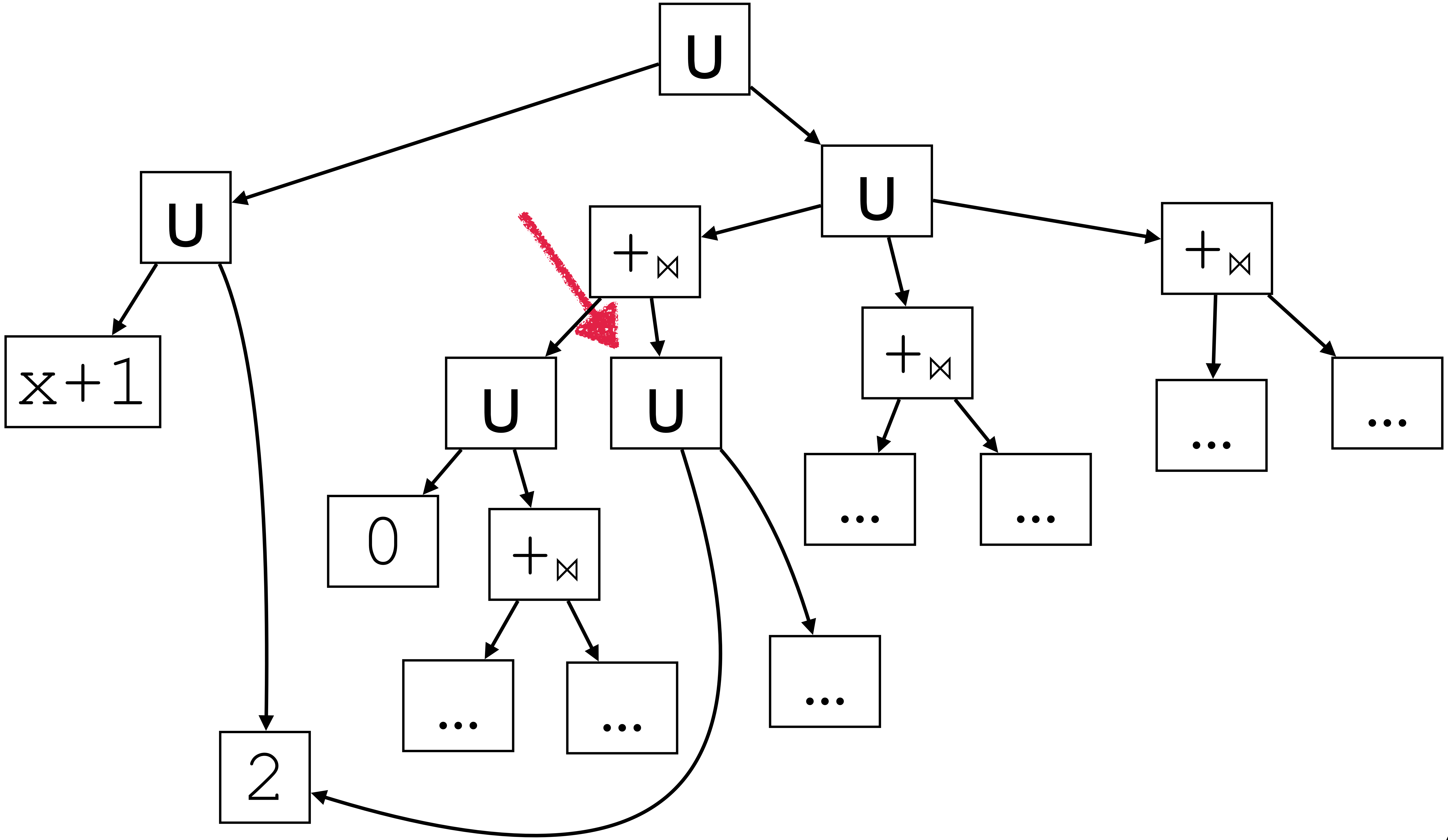
Check if `0 + ??` can be *matched with* any block satisfying `1` ⟼ `2`



Version Space Matching

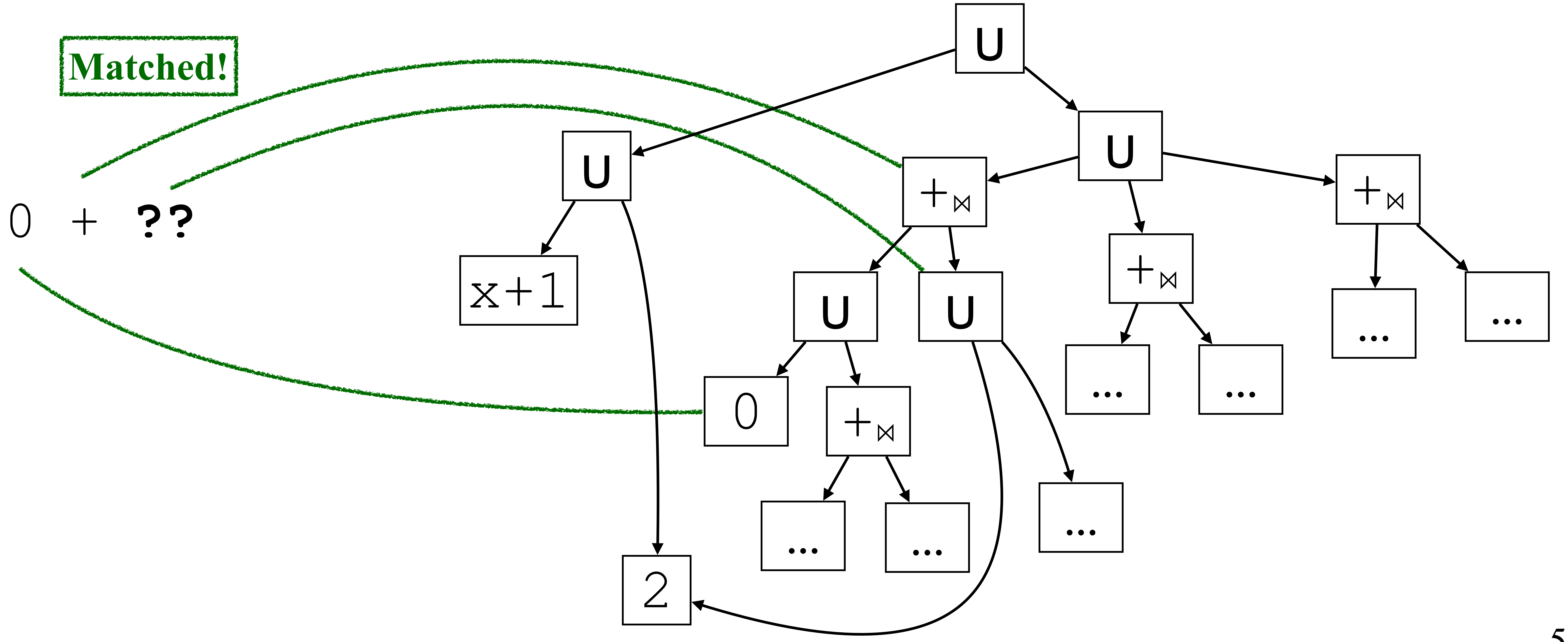
Check if `0 + ??` can be *matched with* any block satisfying `1` $\longmapsto$ `2`

`0 + ??`



Version Space Matching

Check if `0 + ??` can be *matched with* any block satisfying `1` $\longmapsto$ `2`

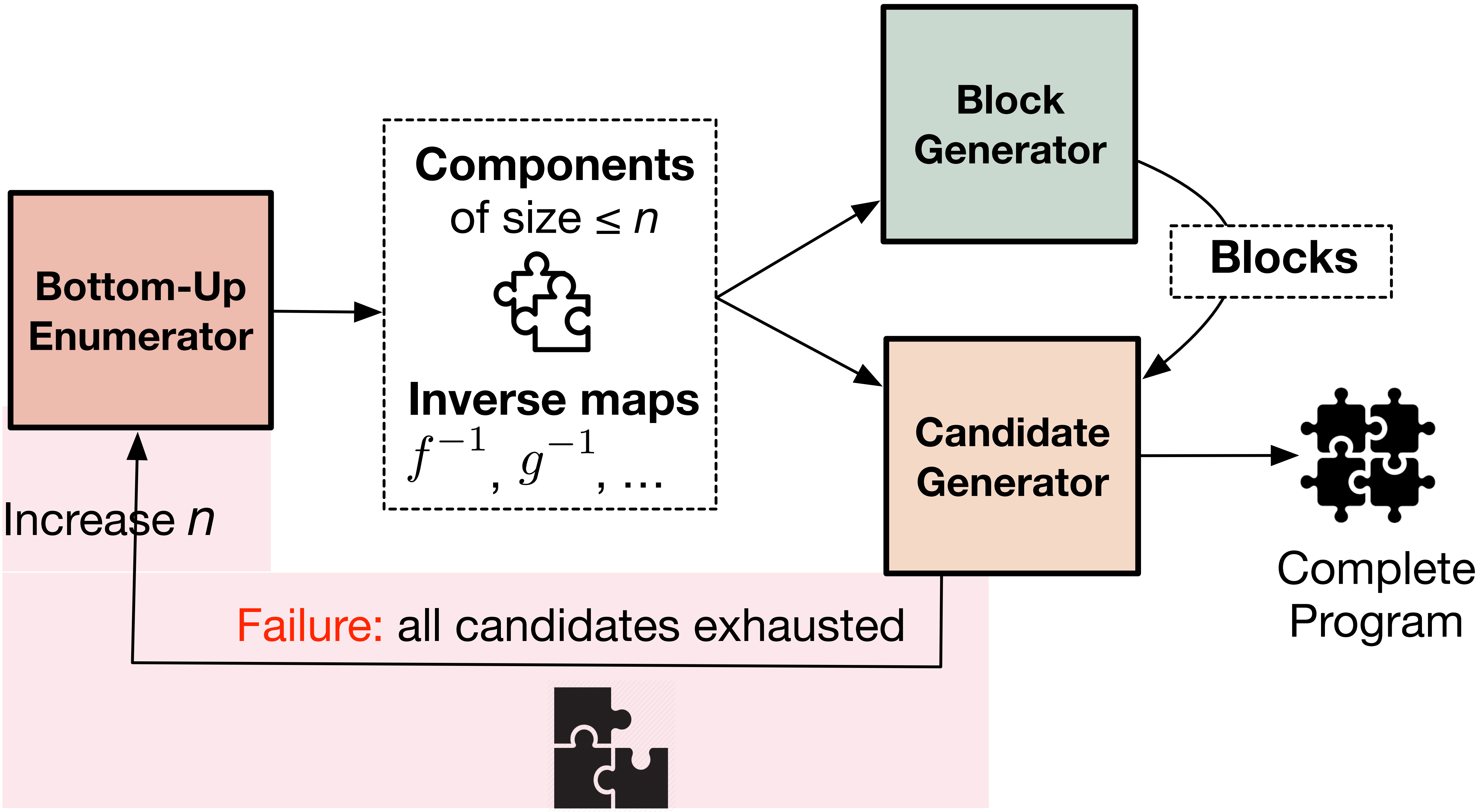


“Temporary” Unsoundness of Block-based Pruning

- For termination of the block generation process, we limit the height of version space.
- Because of this, blocks of a solution may not be generated.
- In this case, block-based pruning may be unsound.
  (i.e., partial programs leading to a solution may be pruned)
- Despite this pruning unsoundness, we never miss a solution.

Search Completeness

- If valid partial programs are pruned and we can't find the solution, we repeat the process after adding larger components.
- More components → More blocks
- Eventually the valid partial programs will not be pruned.

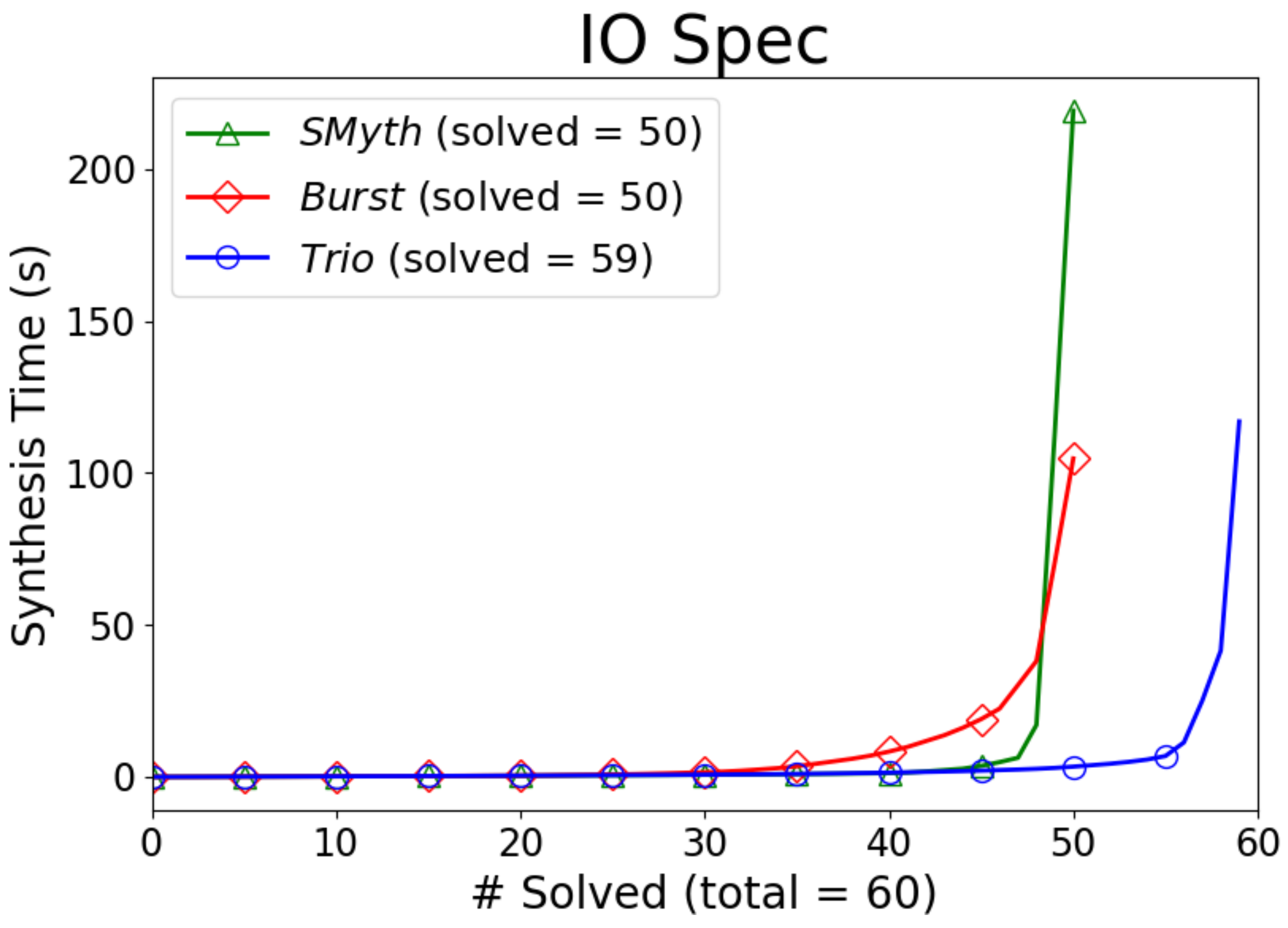


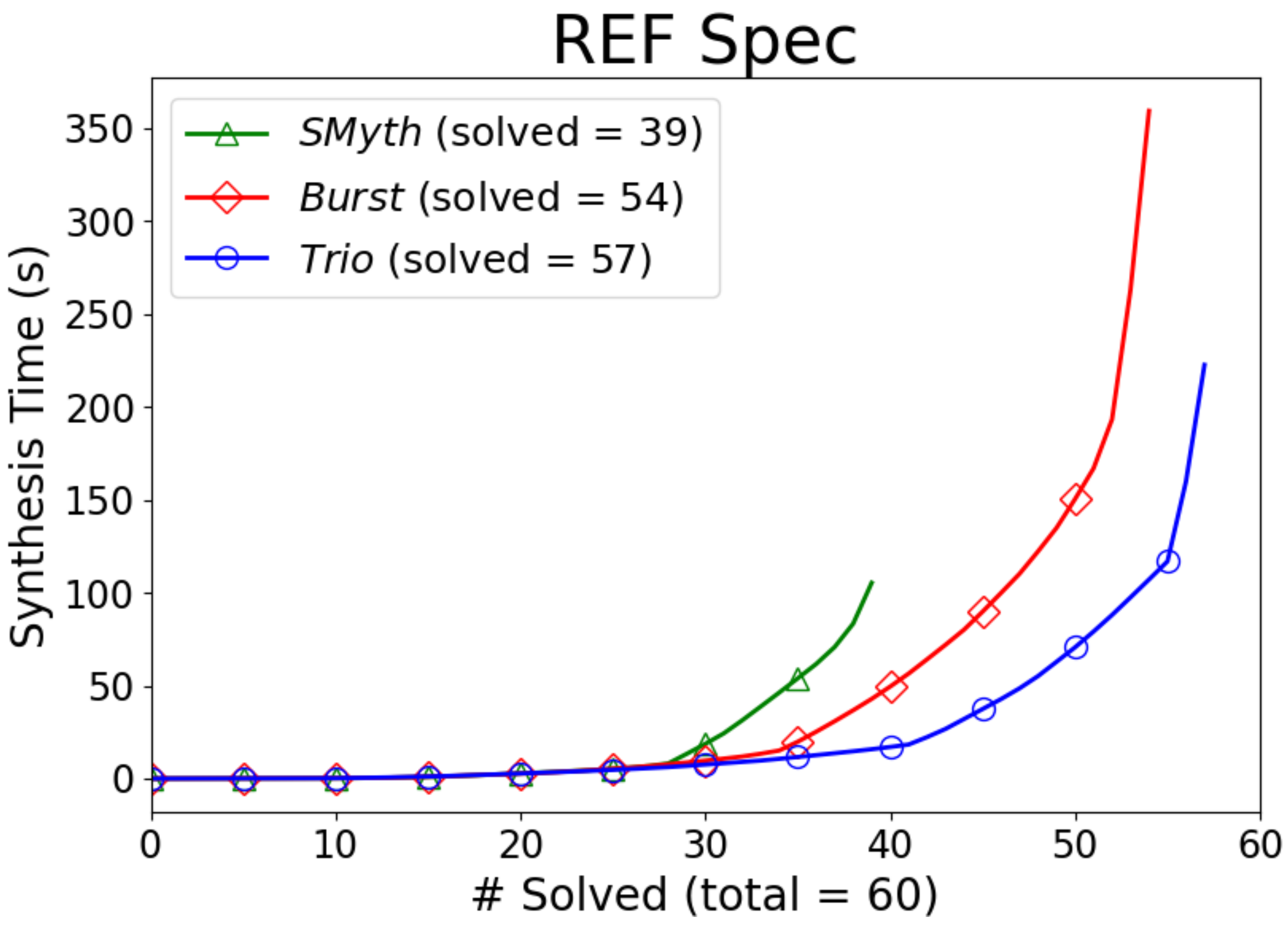
Evaluation

- Benchmark suite (**60** programs)
  - **45** from SMyth benchmark suite + **15** from OCaml tutorial
- Specifications : (1) **IO** examples, (2) **Ref**erence implementation
- Baselines
  - SMyth (ICFP'20): best top-down synthesizer
  - Burst (POPL'22) : best bottom-up synthesizer
- 2 minute timeout

Comparison to Prior Work

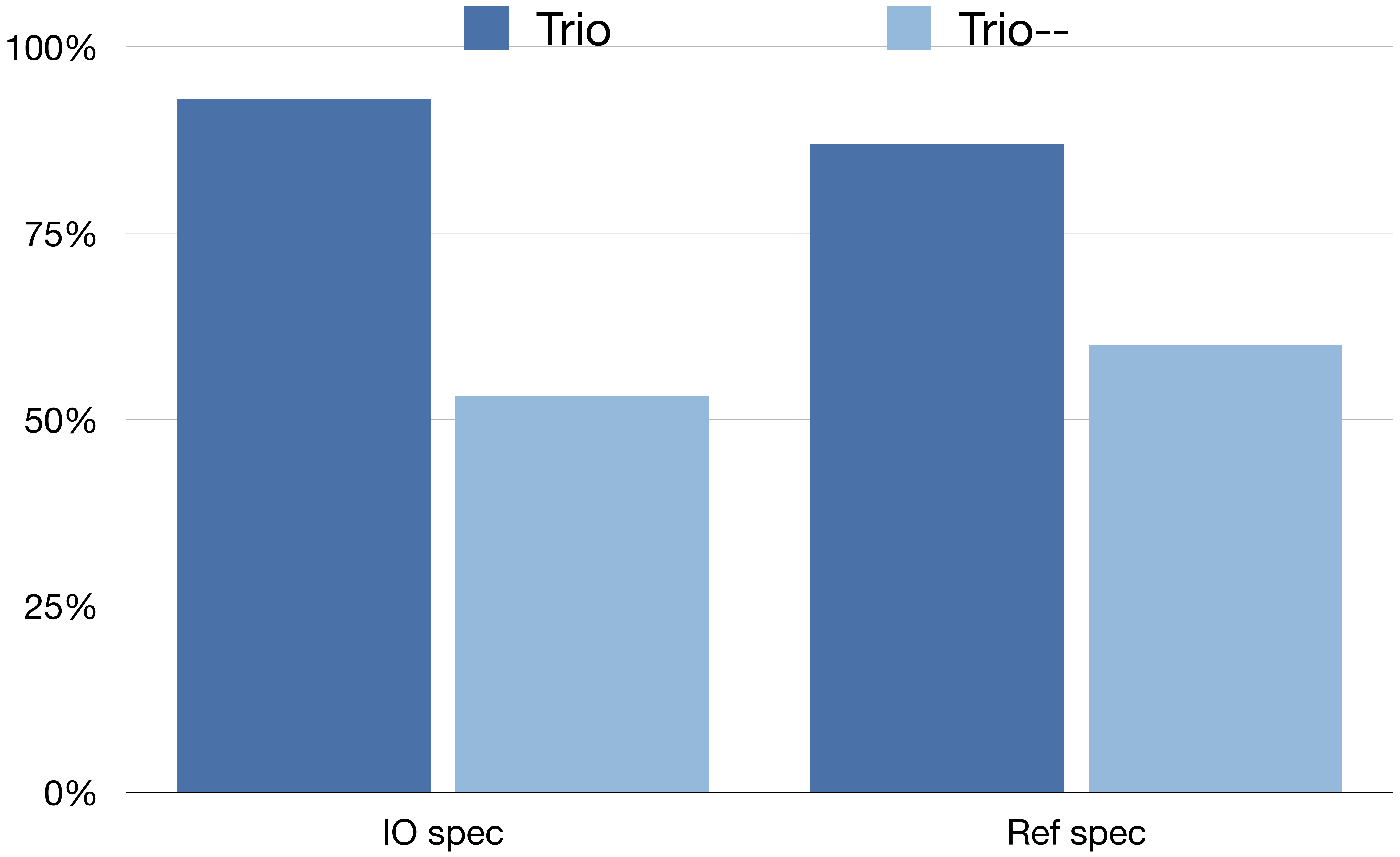
Trio (our tool) outperforms **SMyth** and **Burst**.





Ablation Study

Trio performs better using block-based pruning + library sampling



In the Paper…

- How to synthesize higher-order functions
- Optimizations
- Why our tool outperforms the existing tools (case study)

Thank you!