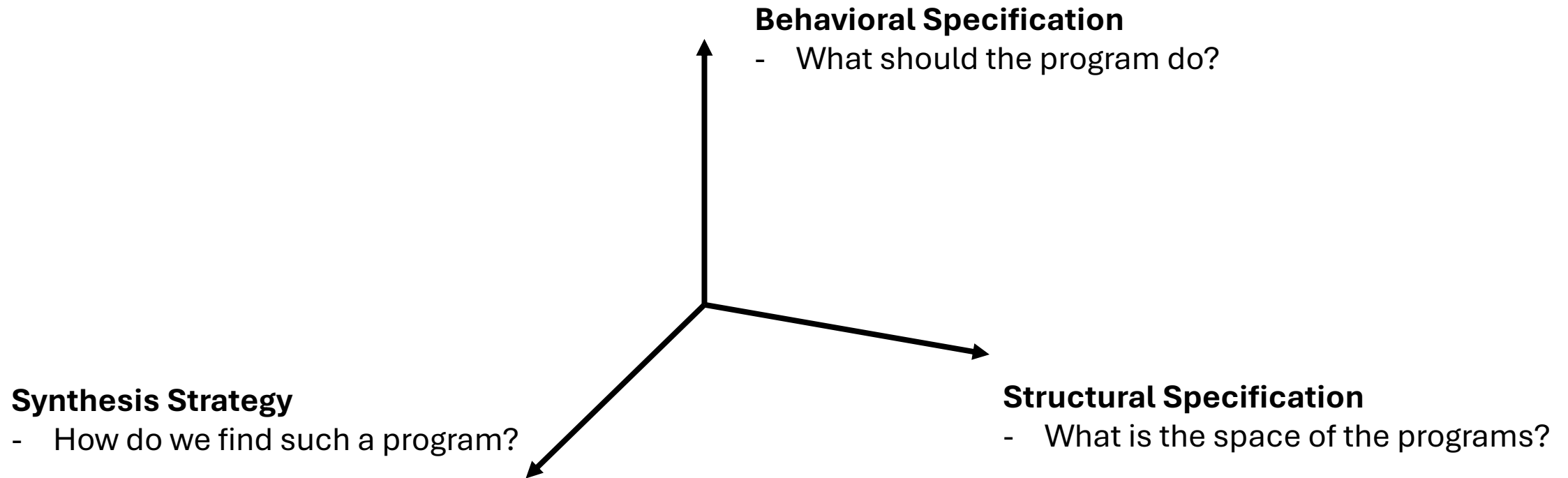


# Machine Programming

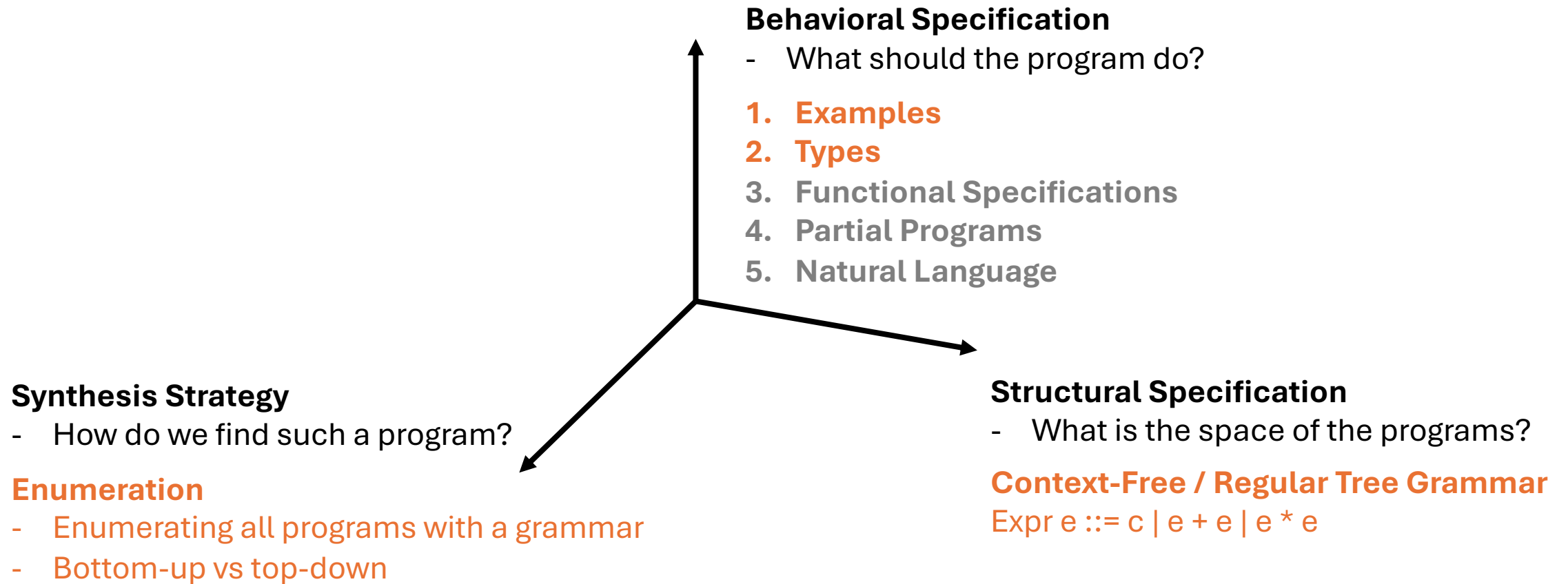
Lecture 4 – Functional Specifications for Synthesis

Ziyang Li

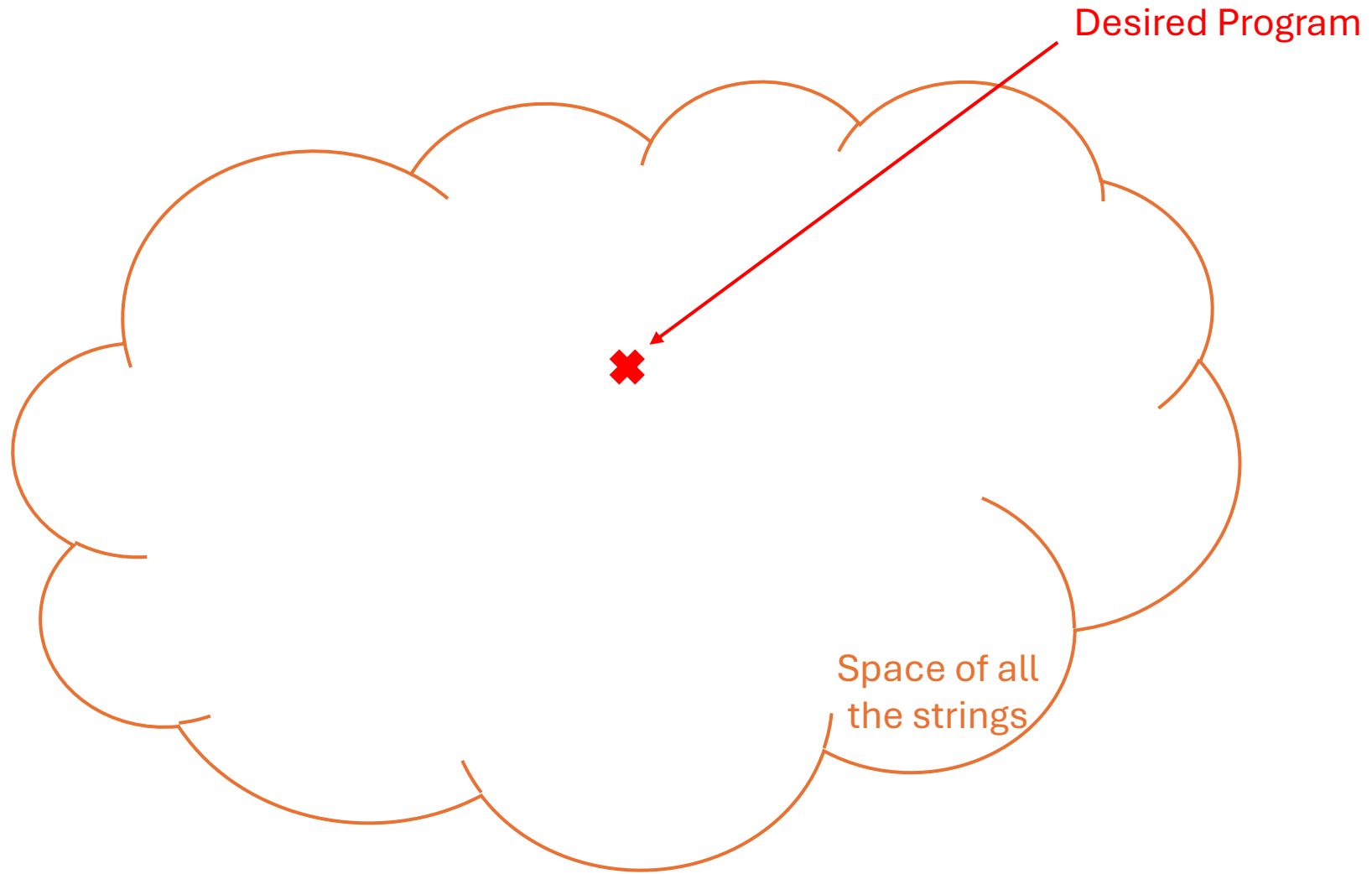
# Dimensions in Program Synthesis



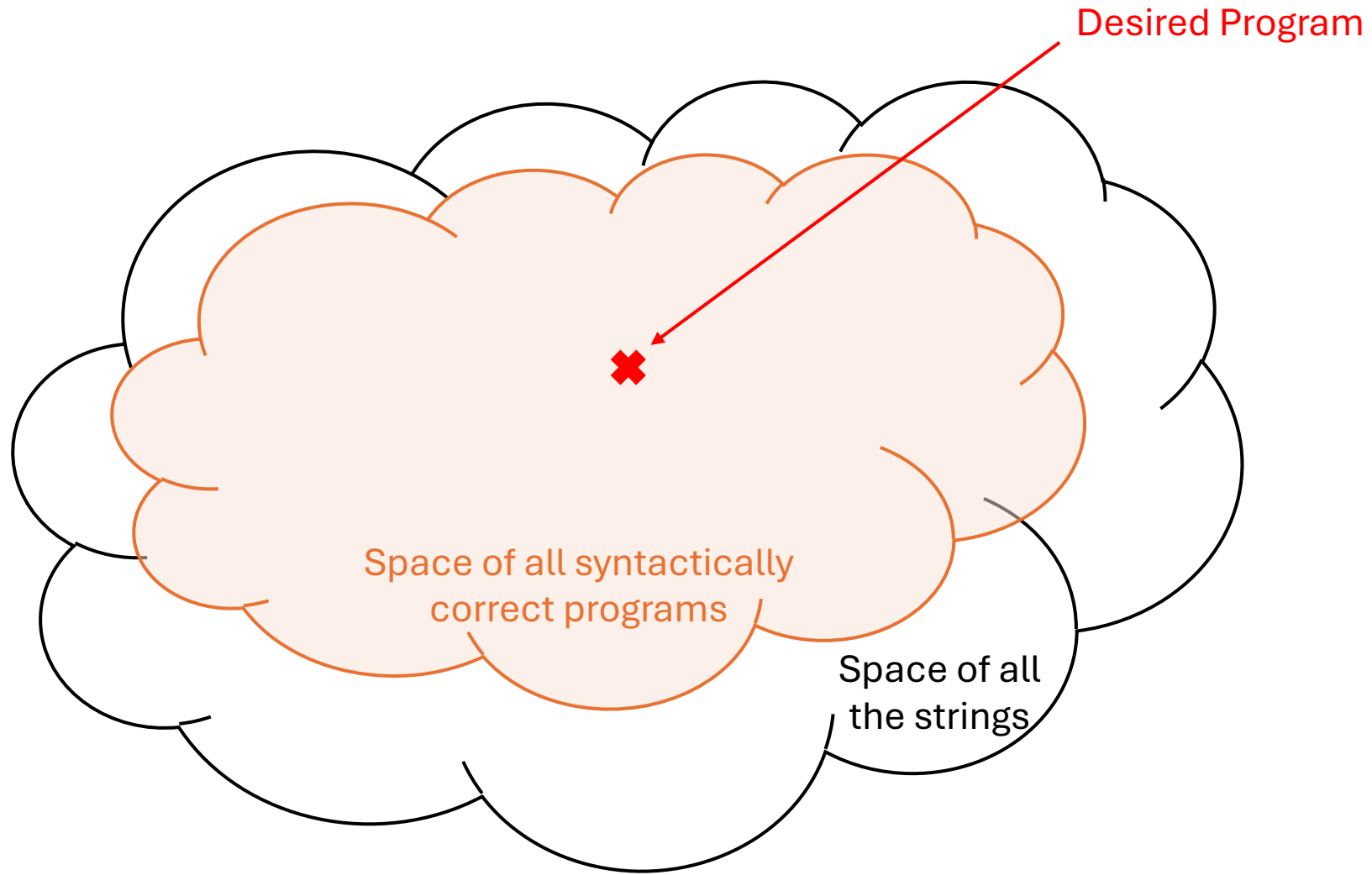
# The Course So Far



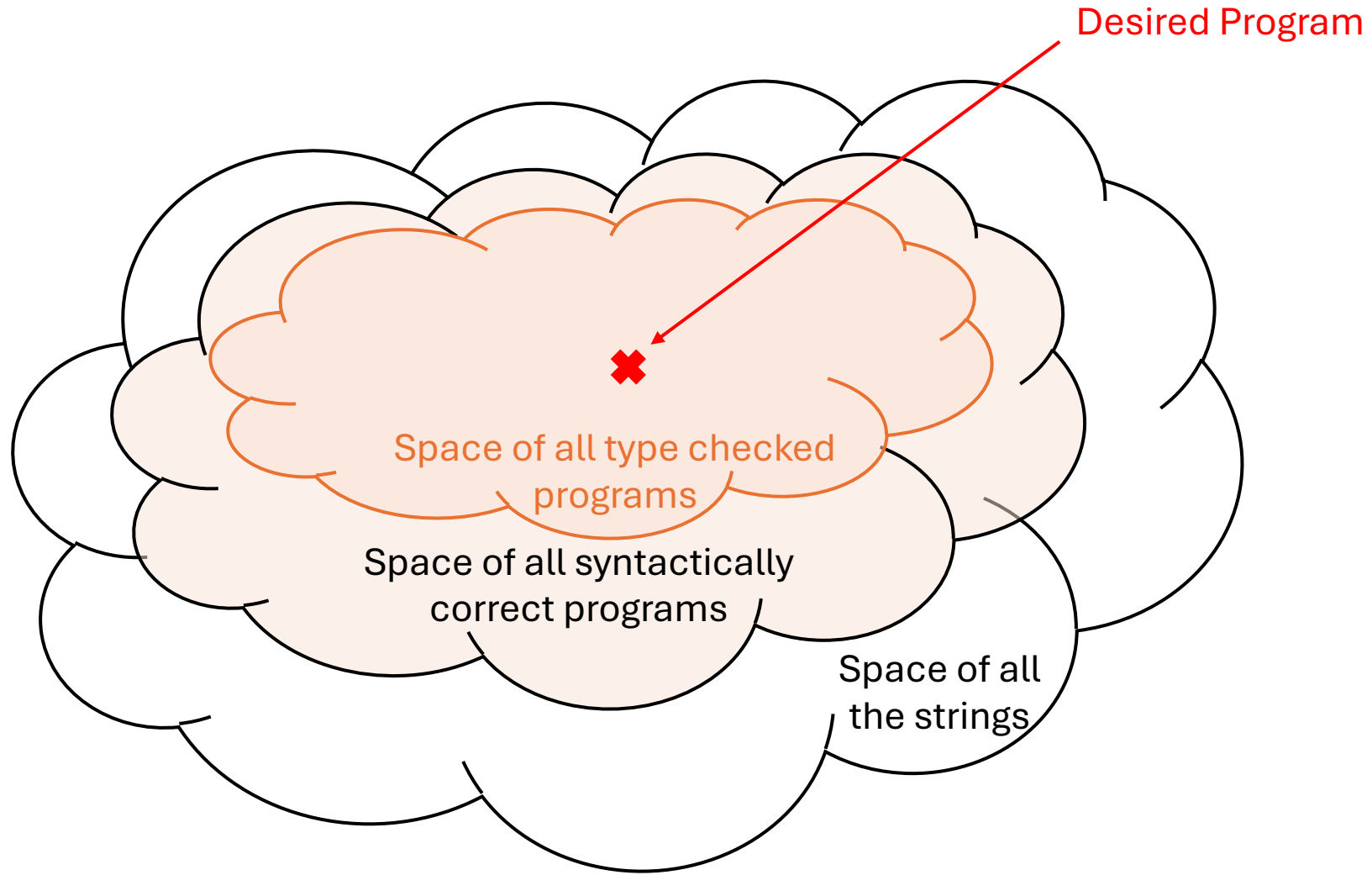
# High Level Picture



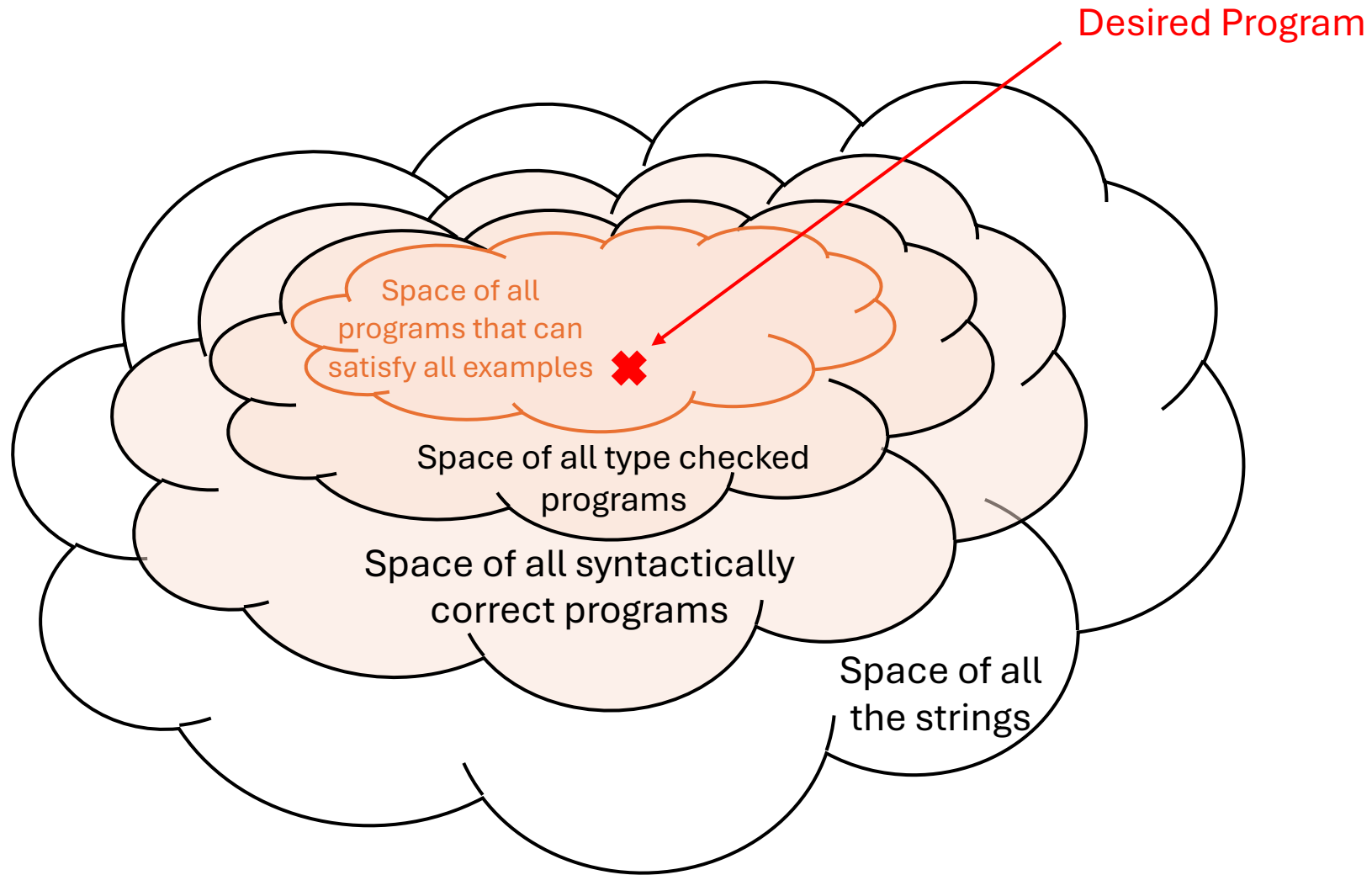
# High Level Picture



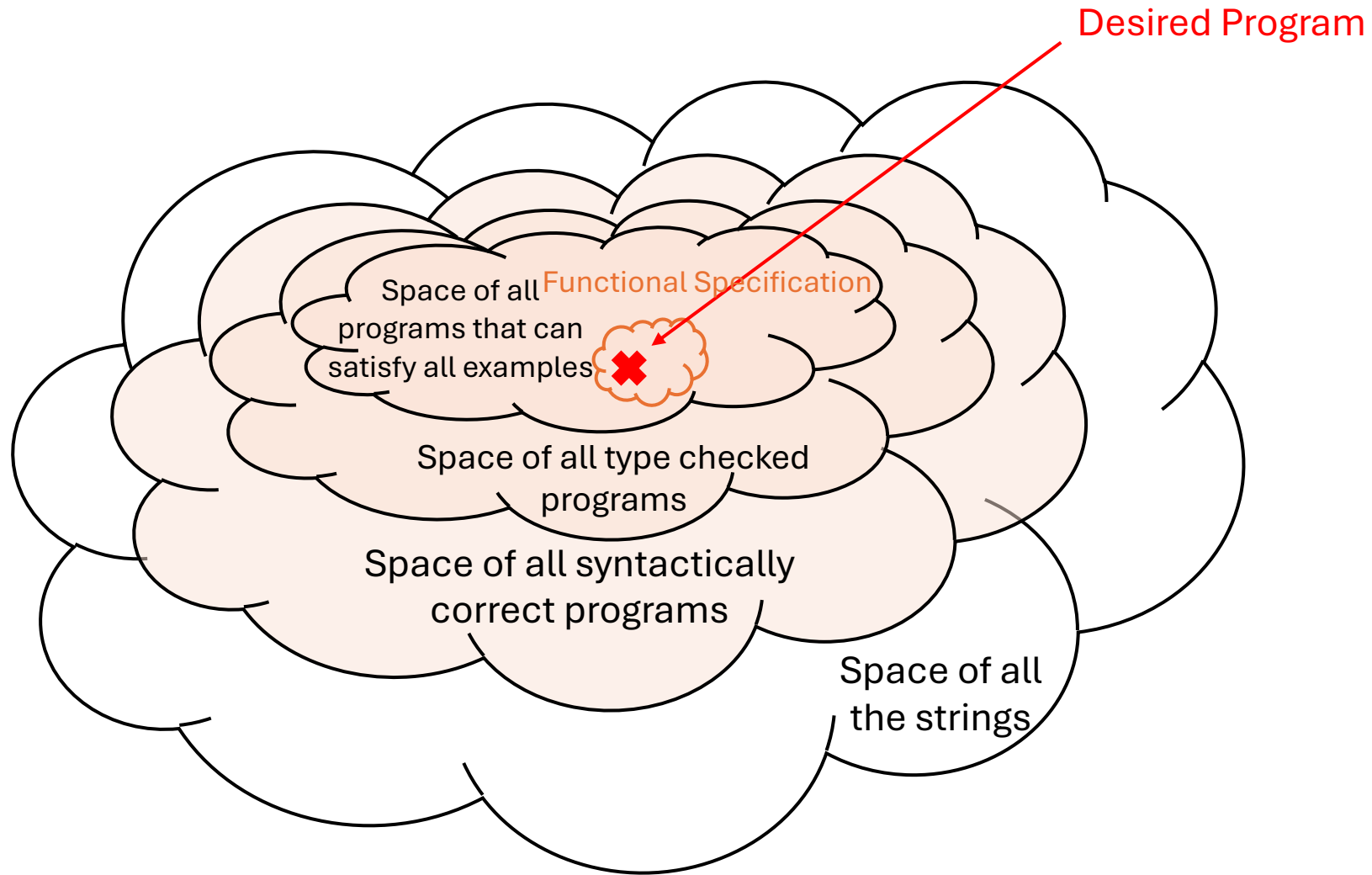
# High Level Picture



# High Level Picture

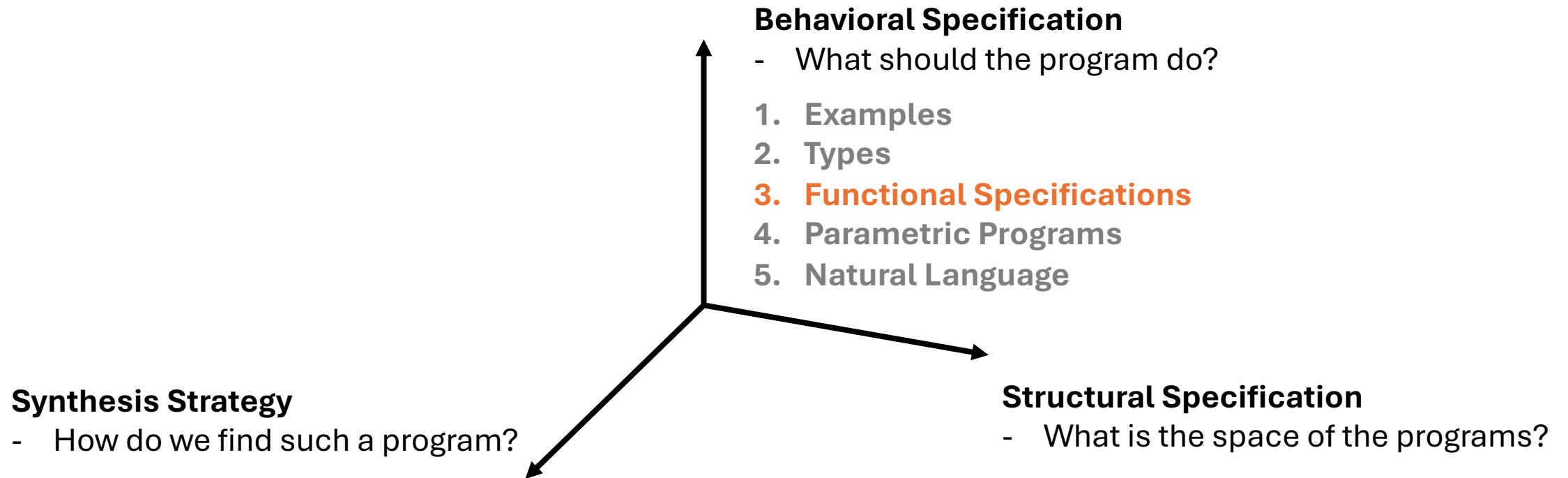


# High Level Picture





# Today



# Is Examples Enough?

$$F([3, 2, 1]) = [1, 2, 3]$$

$$F([2, 1]) = [1, 2]$$

$$F([]) = []$$

# Is Examples Enough?

$$F([3, 2, 1]) = [1, 2, 3]$$

$$F([2, 1]) = [1, 2]$$

$$F([]) = []$$

What is  $F$ ?

# Is Examples Enough?

`reverse( [3, 2, 1] ) = [1, 2, 3]`

`reverse( [2, 1] ) = [1, 2]`

`reverse( [] ) = []`

What is **F**?

# Is Examples Enough?

`sort`( [3, 2, 1] ) = [1, 2, 3]

`sort`( [2, 1] ) = [1, 2]

`sort`( [] ) = []

What is `F`?

# Is Examples Enough?

`reverse`( [3, 2, 1] ) = [1, 2, 3]

`reverse`( [2, 1] ) = [1, 2]

`reverse`( [] ) = []

`sort`( [3, 2, 1] ) = [1, 2, 3]

`sort`( [2, 1] ) = [1, 2]

`sort`( [] ) = []

What is **F**?

# Is Type Enough?

`reverse : List[int] -> List[int]`

`reverse([3, 2, 1]) = [1, 2, 3]`

`reverse([2, 1]) = [1, 2]`

`reverse([]) = []`

`sort : List[int] -> List[int]`

`sort([3, 2, 1]) = [1, 2, 3]`

`sort([2, 1]) = [1, 2]`

`sort([]) = []`

# Functional Specifications

$\forall x, y \in \text{List}[\text{Int}], \text{reverse}(x) = y \Rightarrow$

$$\text{len}(x) = \text{len}(y) \wedge \forall i \in \{1 \dots \text{len}(x)\}, x_i = y_{\text{len}(y)+1-i}$$

`reverse([3, 2, 1]) = [1, 2, 3]`

`reverse([2, 1]) = [1, 2]`

`reverse([]) = []`

$\forall x, y \in \text{List}[\text{Int}], \text{sort}(x) = y \Rightarrow$

$$\text{len}(x) = \text{len}(y) \wedge \forall i \in \{1 \dots \text{len}(x)\}, y_{i+1} > y_i$$

`sort([3, 2, 1]) = [1, 2, 3]`

`sort([2, 1]) = [1, 2]`

`sort([]) = []`



# Functional Specifications

$\forall x, y \in \text{List}[\text{Int}], \text{reverse}(x) = y \Rightarrow$

$$\text{len}(x) = \text{len}(y) \wedge \forall i \in \{1 \dots \text{len}(x)\}, x_i = y_{\text{len}(y)+1-i}$$

1. Input and output has the same length;
2. The  $i$ -th element in  $x$  is the same as  $(n + 1 - i)$ -th element in  $y$

$\forall x, y \in \text{List}[\text{Int}], \text{sort}(x) = y \Rightarrow$

$$\text{len}(x) = \text{len}(y) \wedge \forall i \in \{1 \dots \text{len}(x)\}, y_{i+1} > y_i$$

`reverse([3, 2, 1]) = [1, 2, 3]`

`reverse([2, 1]) = [1, 2]`

`reverse([]) = []`

`sort([3, 2, 1]) = [1, 2, 3]`

`sort([2, 1]) = [1, 2]`

`sort([]) = []`

# Functional Specifications

$\forall x, y \in \text{List}[\text{Int}], \text{reverse}(x) = y \Rightarrow$

$$\text{len}(x) = \text{len}(y) \wedge \forall i \in \{1 \dots \text{len}(x)\}, x_i = y_{\text{len}(y)+1-i}$$

1. Input and output has the same length;
2. The  $i$ -th element in  $x$  is the same as  $(n + 1 - i)$ -th element in  $y$

```
reverse([3, 2, 1]) = [1, 2, 3]
reverse([2, 1]) = [1, 2]
reverse([]) = []
```

$\forall x, y \in \text{List}[\text{Int}], \text{sort}(x) = y \Rightarrow$

$$\text{len}(x) = \text{len}(y) \wedge \forall i \in \{1 \dots \text{len}(x)\}, y_{i+1} > y_i$$

1. Input and output has the same length;
2. In result  $y$ , the  $i$ -th element is always less than or equal to the  $(i + 1)$ -th element

```
sort([3, 2, 1]) = [1, 2, 3]
sort([2, 1]) = [1, 2]
sort([]) = []
```

# Functional Specifications

**Pre-condition:**

$\forall x, y \in \text{List}[\text{Int}], \text{sort}(x) = y \Rightarrow$

$\text{len}(x) = \text{len}(y) \wedge \forall i \in \{1 \dots \text{len}(x)\}, y_{i+1} > y_i$

**Post-condition:**

# Functional Specifications

**Pre-condition:**

$x \in \text{List}[\text{Int}]$

$x$  is an integer list

$\forall x, y \in \text{List}[\text{Int}], \text{sort}(x) = y \Rightarrow$

$\text{len}(x) = \text{len}(y) \wedge \forall i \in \{1 \dots \text{len}(x)\}, y_{i+1} > y_i$

**Post-condition:**

# Functional Specifications

$\forall x, y \in \text{List}[\text{Int}], \text{sort}(x) = y \Rightarrow$

$\text{len}(x) = \text{len}(y) \wedge \forall i \in \{1 \dots \text{len}(x)\}, y_{i+1} > y_i$

## Pre-condition:

$x \in \text{List}[\text{Int}]$

$x$  is an integer list

## Post-condition:

$y \in \text{List}[\text{Int}]$

$y$  is an integer list

$\text{len}(x) = \text{len}(y)$

$y$  and  $x$  has the same length

$y_{i+1} \geq y_i$

Latter element in  $y$  always  
bigger than or equal to  
previous element

# Form: Pre- and Post-Conditions

$\forall x, y \in \text{List}[\text{Int}], \text{sort}(x) = y \Rightarrow$   
 $\text{len}(x) = \text{len}(y) \wedge \forall i \in \{1 \dots \text{len}(x)\}, y_{i+1} > y_i$

**Pre-condition:**

$x \in \text{List}[\text{Int}]$

$x$  is an integer list

**Premise:** All valid inputs to a function **MUST** satisfy

**Post-condition:**

$y \in \text{List}[\text{Int}]$

$y$  is an integer list

$\text{len}(x) = \text{len}(y)$

$y$  and  $x$  has the same length

$y_{i+1} \geq y_i$

Latter element in  $y$  always bigger than or equal to previous element

**Promise:** all outputs **WILL** satisfy if the premise holds

# Pre- and Post-Conditions in the Wild

Russol (Rust)

```
method Find(a: array<int>, key: int) returns (index: int)
  ensures 0 <= index ==> index < a.Length && a[index] == key
  ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key
{
  index := 0;
  while index < a.Length
  {
    if a[index] == key { return; }
    index := index + 1;
  }
  index := -1;
}
```

Dafny

```
int abs_val(int x)
  _Pre_ true
  _Post_ (retval >= 0)
{
  if (x < 0) {
    return -x;
  } else {
    return x;
  }
}
```

Checked-C

```
int main() {
  int i;
  int j=__VERIFIER_nondet_int();
  int n=__VERIFIER_nondet_int();
  assume_abort_if_not(n < 100000);
  int a[n];

  assume_abort_if_not(j>0 && j < 10000);
  for(i=1;i<n;i++) {
    int k=__VERIFIER_nondet_int();
    assume_abort_if_not(k>0 && k < 10000);
    a[i]=i+j+k;
  }

  for(i=1;i<n;i++)
    __VERIFIER_assert(a[i]>=(i+2));
  return 0;
}
```

```
#[requires(self.len() > 0)]
fn peek(&self) -> &T {
  todo!()
}
```

```
{{ True }}
while X ≠ 0 do
  X := X - 1
end
{{ X = 0 }}
```

Software Foundations /  
Hoare Logic (Coq)

SV-Comp (C)

# Pre- and Post-Condition Practice

```
def insert(list: List[int], elem: int) -> List[int]:
```

```
def sqrt(x: float) -> float:
```

```
def matrix_mul(a: Tensor, b: Tensor) -> Tensor:
```



# Completeness of Pre- and Post-Conditions

Pre-condition

$x \in \text{List}[\text{Int}] \quad n = \text{len}(x)$

$y = \text{sort}(x)$

Is this post-condition  
complete?

Post-condition

$y \in \text{List}[\text{Int}] \quad \forall i. 0 \leq i < n - 1 \Rightarrow y[i] \leq y[i + 1]$

# Completeness of Pre- and Post-Conditions

Input: [1, 2, 3, 4, 5]

Pre-condition

$x \in \text{List}[\text{Int}] \quad n = \text{len}(x)$

$y = \text{sort}(x)$

Post-condition

$y \in \text{List}[\text{Int}] \quad \forall i. 0 \leq i < n - 1 \Rightarrow y[i] \leq y[i + 1]$

Output: [1, 2, 3, 4, 5, 6]

Is this post-condition  
complete? **×**

# Completeness of Pre- and Post-Conditions

Pre-condition

$x \in \text{List}[\text{Int}] \quad n = \text{len}(x)$

$y = \text{sort}(x)$

Post-condition

$y \in \text{List}[\text{Int}] \quad \forall i. 0 \leq i < n - 1 \Rightarrow y[i] \leq y[i + 1]$   
 $\text{len}(y) = n$

**Attempt 2:**  
Is this post-condition  
complete?

# Completeness of Pre- and Post-Conditions

Input: [1, 2, 3, 4, 5]

Pre-condition

$x \in \text{List}[\text{Int}] \quad n = \text{len}(x)$

$y = \text{sort}(x)$

Post-condition

$y \in \text{List}[\text{Int}] \quad \forall i. 0 \leq i < n - 1 \Rightarrow y[i] \leq y[i + 1]$

$\text{len}(y) = n$

Output: [5, 5, 5, 5, 5]

**Attempt 2:**  
Is this post-condition  
complete? **✗**

# Completeness of Pre- and Post-Conditions

Pre-condition

$x \in \text{List}[\text{Int}] \quad n = \text{len}(x)$

$y = \text{sort}(x)$

**Attempt 3:**  
Is this post-condition  
complete?

Post-condition

$y \in \text{List}[\text{Int}] \quad \forall i. 0 \leq i < n - 1 \Rightarrow y[i] \leq y[i + 1]$

$\text{len}(y) = n$

$\forall i. 0 \leq i < n - 1 \Rightarrow \exists j. x[i] = y[j]$

# Completeness of Pre- and Post-Conditions

Pre-condition

$x \in \text{List}[\text{Int}] \quad n = \text{len}(x)$

$y = \text{sort}(x)$

**Attempt 3:**  
Is this post-condition  
complete?

Post-condition

$y \in \text{List}[\text{Int}] \quad \forall i. 0 \leq i < n - 1 \Rightarrow y[i] \leq y[i + 1]$

$\text{len}(y) = n$

$\forall i. 0 \leq i < n - 1 \Rightarrow \exists j. x[i] = y[j]$

$\forall i. 0 \leq i < n - 1 \Rightarrow \exists j. y[i] = x[j]$

# Completeness of Pre- and Post-Conditions

Input: [1, 2, 3, 4, 2]

Pre-condition

$x \in \text{List}[\text{Int}] \quad n = \text{len}(x)$

$y = \text{sort}(x)$

Post-condition

$y \in \text{List}[\text{Int}] \quad \forall i. 0 \leq i < n - 1 \Rightarrow y[i] \leq y[i + 1]$

$\text{len}(y) = n$

Output: [1, 2, 3, 4, 4]

$\forall i. 0 \leq i < n - 1 \Rightarrow \exists j. x[i] = y[j]$

$\forall i. 0 \leq i < n - 1 \Rightarrow \exists j. y[i] = x[j]$

**Attempt 3:**  
Is this post-condition  
complete? **✗**

# Completeness of Pre- and Post-Conditions

Pre-condition

$x \in \text{List}[\text{Int}] \quad n = \text{len}(x)$


$y = \text{sort}(x)$

Post-condition

$y \in \text{List}[\text{Int}] \quad \forall i. 0 \leq i < n - 1 \Rightarrow y[i] \leq y[i + 1]$

$\text{len}(y) = n$

$\exists p: \mathbb{Z}_n \rightarrow \mathbb{Z}_n, p \text{ is a permutation,}$   
 $\forall i. 0 \leq i < n \Rightarrow y[i] = x[p(i)]$

**Attempt 4:**  
Is this post-condition  
complete? 



# Difficult to specify complete functional specification!

## Leveraging Rust Types for Program Synthesis

JONÁŠ FIALA, ETH Zurich, Switzerland

SHACHAR ITZHAKY, Technion, Israel

PETER MÜLLER, ETH Zurich, Switzerland

NADIA POLIKARPOVA, University of California, San Diego, USA

ILYA SERGEY, National University of Singapore, Singapore

Rust type +  
functional spec

```
[#requires ...]  
[#ensures ...]  
fn target(x: T1...) -> T  
  
[#pure]  
fn f(x: T1...) -> T  
...
```

**Contributions.** In summary, this paper makes the following contributions:

- Synthetic Ownership Logic (SOL), a variant of Separation Logic that is targeted to program synthesis of well-typed Rust programs from type signatures and functional specifications.
- RusSOL, the first synthesizer for Rust code from functional correctness specifications. We built RusSOL by integrating SOL into SuSLik's general-purpose proof search framework.
- An extensive evaluation of RusSOL with regard to utility and performance. We show that it is capable of synthesizing a large number of non-trivial heap-manipulating Rust programs, in a matter of seconds, and that **required annotations are on average 27% shorter than the code.**

# Specification itself has a language

- First-order logic operators:
  - $\forall, \exists, =, \neq, \wedge, \vee, \neg, \Rightarrow, \dots$
- Base types, commonly used types, and their predicates
  - `Int`, `Bool`, `Set[T]`, `List[T]`, ...
  - $+, -, \times, \div, ::, \cup, \cap, \in, [.], \&\&, ||, !, len, \dots$
- Logic systems to specify program behaviors
  - Memory (Separation Logic):  $\{.\}, \mapsto, *, \dots$
  - Temporal Behavior (Temporal Logic): `Globally`, `Next`, `Finally`, ...
  - Mathematical Objects: `Permutation`, ...

```
{x ↦ a * y ↦ b} void swap(loc x, loc y) {x ↦ b * y ↦ a}
```

# Specification itself has a language

Syntax + Semantics

- First-order logic operators:
  - $\forall, \exists, =, \neq, \wedge, \vee, \neg, \Rightarrow, \dots$
- Base types, commonly used types, and their predicates
  - `Int, Bool, Set[T], List[T], ...`
  - `+, -,  $\times$ ,  $\div$ , ::,  $\cup$ ,  $\cap$ ,  $\in$ ,  $[\cdot]$ ,  $\&\&$ ,  $||$ ,  $!$ , len, ...`
- Logic systems to specify program behaviors
  - Memory (Separation Logic): `{ $\cdot$ },  $\mapsto$ ,  $*$ , ...`
  - Temporal Behavior (Temporal Logic): `Globally, Next, Finally, ...`
  - Mathematical Objects: `Permutation, ...`

# Specification itself has a language

Syntax

+

Semantics

- First-order logic operators:
  - $\forall, \exists, =, \neq, \wedge, \vee, \neg, \Rightarrow, \dots$
- Base types, commonly used types, and their predicates
  - `Int`, `Bool`, `Set[T]`, `List[T]`, ...
  - `+`, `-`, `×`, `÷`, `::`, `U`, `∩`, `∈`, `[. ]`, `&&`, `||`, `!`, `len`, ...
- Logic systems to specify program behaviors
  - Memory (Separation Logic): `{. }`, `↦`, `*`, ...
  - Temporal Behavior (Temporal Logic): `Globally`, `Next`, `Finally`, ...
  - Mathematical Objects: `Permutation`, ...

???

# Specification itself has a language

## Syntax

+

## Semantics

- First-order logic operators:
  - $\forall, \exists, =, \neq, \wedge, \vee, \neg, \Rightarrow, \dots$
- Base types, commonly used types, and their predicates
  - `Int`, `Bool`, `Set[T]`, `List[T]`, ...
  - `+`, `-`, `×`, `÷`, `::`, `U`, `∩`, `∈`, `[.]`, `&&`, `||`, `!`, `len`, ...
- Logic systems to specify program behaviors
  - Memory (Separation Logic): `{.}`, `↦`, `*`, ...
  - Temporal Behavior (Temporal Logic): `Globally`, `Next`, `Finally`, ...
  - Mathematical Objects: `Permutation`, ...

$$\{P\} c \{Q\}$$

Proof-Theoretic Semantics

$$[[c]] \models P \rightarrow Q$$

Operational Alignment Semantics

# Verification of Program with a Specification

{Pre-condition} Program {Post-condition}

$$\{P\} c \{Q\}$$

# Verification of Program with a Specification

{Pre-condition} Program {Post-condition}

$\{P\} c \{Q\}$

$\{x \mapsto a * y \mapsto b\}$  **void** swap(**loc** x, **loc** y)  $\{x \mapsto b * y \mapsto a\}$

[#requires ...]  
[#ensures ...]  
**fn** target(x: T<sub>1</sub>...) -> T

```
VERIFIER_assume(true);  
while (X ≠ 0) {  
    X = X - 1;  
}  
VERIFIER_assert(X = 0);
```

```
{{ True }}  
while X ≠ 0 do  
    X := X - 1  
end  
{{ X = 0 }}
```

# Verification of Program with a Specification

{Pre-condition} Program {Post-condition}

$$\{P\} c \{Q\}$$

Program  $c$ : a program state transformer

Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$



Program  $c$ : a program state transformer  
Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

$P$

$x: \text{Int}$

```
int abs(int x) {  
    int y;  
    if (x >= 0) // B1  
        y = x; // B2  
    else  
        y = -x; // B3  
    return y; // B4  
}
```

$c$

$Q$

$(y = -x \vee y = x) \wedge y \geq 0$

Program  $c$ : a program state transformer

Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
int abs(int x) {
```

```
  {x: Int}
```

```
  int y;
```

```
  if (x >= 0) // B1
```

```
    y = x; // B2
```

```
  else
```

```
    y = -x; // B3
```

```
  return y; // B4
```

```
}
```

Goal:  $(y = -x \vee y = x) \wedge y \geq 0$

Program  $c$ : a program state transformer

Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
int abs(int x) {  
    {x: Int}  
    int y;  
    {x: Int, y: Int}  
    if (x >= 0) // B1  
  
        y = x; // B2  
  
    else  
  
        y = -x; // B3  
  
    return y; // B4  
}
```

Goal:  $(y = -x \vee y = x) \wedge y \geq 0$

Program  $c$ : a program state transformer

Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
int abs(int x) {  
    {x: Int}  
    int y;  
    {x: Int, y: Int}  
    if (x >= 0) // B1  
        {x: Int, y: Int, x ≥ 0}  
        y = x; // B2  
  
    else  
  
        y = -x; // B3  
  
    return y; // B4  
}
```

Goal:  $(y = -x \vee y = x) \wedge y \geq 0$

Program  $c$ : a program state transformer

Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
int abs(int x) {  
    {x: Int}  
    int y;  
    {x: Int, y: Int}  
    if (x >= 0) // B1  
        {x: Int, y: Int, x ≥ 0}  
        y = x; // B2  
        {x: Int, y: Int, x ≥ 0, x = y}  
    else  
  
        y = -x; // B3  
  
    return y; // B4  
}
```

Goal:  $(y = -x \vee y = x) \wedge y \geq 0$

Program  $c$ : a program state transformer  
Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
int abs(int x) {  
    {x: Int}  
    int y;  
    {x: Int, y: Int}  
    if (x >= 0) // B1  
        {x: Int, y: Int, x ≥ 0}  
        y = x; // B2  
        {x: Int, y: Int, x ≥ 0, x = y}  
    else  
        {x: Int, y: Int, x < 0}  
        y = -x; // B3  
  
    return y; // B4  
}
```

Goal:  $(y = -x \vee y = x) \wedge y \geq 0$

Program  $c$ : a program state transformer  
Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
int abs(int x) {  
    {x: Int}  
    int y;  
    {x: Int, y: Int}  
    if (x >= 0) // B1  
        {x: Int, y: Int, x ≥ 0}  
        y = x; // B2  
        {x: Int, y: Int, x ≥ 0, x = y}  
    else  
        {x: Int, y: Int, x < 0}  
        y = -x; // B3  
        {x: Int, y: Int, x < 0, y = -x}  
    return y; // B4  
}
```

Goal:  $(y = -x \vee y = x) \wedge y \geq 0$

Program  $c$ : a program state transformer  
Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
int abs(int x) {  
    {x: Int}  
    int y;  
    {x: Int, y: Int}  
    if (x >= 0) // B1  
        {x: Int, y: Int, x ≥ 0}  
        y = x; // B2  
        {x: Int, y: Int, x ≥ 0, x = y}  
    else  
        {x: Int, y: Int, x < 0}  
        y = -x; // B3  
        {x: Int, y: Int, x < 0, y = -x}  
    {x: Int, y: Int, (x < 0 ∧ y = -x) ∨ (x ≥ 0 ∧ x = y)}  
    return y; // B4  
}
```

Goal:  $(y = -x \vee y = x) \wedge y \geq 0$



Program  $c$ : a program state transformer

Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
int abs(int x) {  
    {x: Int}  
    int y;  
    {x: Int, y: Int}  
    if (x >= 0) // B1  
        {x: Int, y: Int, x ≥ 0}  
        y = x; // B2  
        {x: Int, y: Int, x ≥ 0, x = y}  
    else  
        {x: Int, y: Int, x < 0}  
        y = -x; // B3  
        {x: Int, y: Int, x < 0, y = -x}  
    {x: Int, y: Int, (x < 0 ∧ y = -x) ∨ (x ≥ 0 ∧ x = y)}  
    return y; // B4  
}
```

Goal:  $(y = -x \vee y = x) \wedge y \geq 0$

Program  $c$ : a program state transformer

Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
int abs(int x) {  
    {x: Int}  
    int y;  
    {x: Int, y: Int}  
    if (x >= 0) // B1  
        {x: Int, y: Int, x ≥ 0}  
        y = x; // B2  
        {x: Int, y: Int, x ≥ 0, x = y}  
    else  
        {x: Int, y: Int, x < 0}  
        y = -x; // B3  
        {x: Int, y: Int, x < 0, y = -x}  
  
    return y; // B4  
}
```

$\{x: \text{Int}, y: \text{Int}, (x < 0 \wedge y = -x) \vee (x \geq 0 \wedge x = y)\}$



Goal:  $(y = -x \vee y = x) \wedge y \geq 0$

Program  $c$ : a program state transformer

Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
int abs(int x) {  
  {x: Int}  
  int y;  
  {x: Int, y: Int}  
  if (x >= 0) // B1  
    {x: Int, y: Int, x ≥ 0}  
    y = x; // B2  
    {x: Int, y: Int, x ≥ 0, x = y}  
  else  
    {x: Int, y: Int, x < 0}  
    y = -x; // B3  
    {x: Int, y: Int, x < 0, y = -x}  
  
  return y; // B4  
}
```

$\{x: \text{Int}, y: \text{Int}, (x < 0 \wedge y = -x) \vee (x \geq 0 \wedge x = y)\}$



Goal:  $(y = -x \vee y = x) \wedge y \geq 0$

?

Program  $c$ : a program state transformer  
Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
int abs(int x) {  
  {x: Int}  
  int y;  
  {x: Int, y: Int}  
  if (x >= 0) // B1  
    {x: Int, y: Int, x ≥ 0}  
    y = x; // B2  
    {x: Int, y: Int, x ≥ 0, x = y}  
  else  
    {x: Int, y: Int, x < 0}  
    y = -x; // B3  
    {x: Int, y: Int, x < 0, y = -x}  
  
  return y; // B4  
}
```

$\{x: \text{Int}, y: \text{Int}, (x < 0 \wedge y = -x) \vee (x \geq 0 \wedge x = y)\}$



Goal:  $(y = -x \vee y = x) \wedge y \geq 0$



**Z3 Theorem Prover!**


Can you write a python program with Z3 to check the satisfiability of the following

$\{x:\text{Int}, y:\text{Int}, (x < 0 \wedge y = -x) \vee (x \geq 0 \wedge x = y)\} \Rightarrow \text{Goal: } (y = -x \vee y = x) \wedge y \geq 0$

Thought for 16s

Here's a tiny Z3Py script that checks **validity** of your implication by asking Z3 to find a counterexample to `pre  $\wedge$   $\neg$ goal`. If that is **unsat**, then the implication holds for all integers `x, y`.

python

 Copy code

```
# pip install z3-solver (if not already installed)
from z3 import *

# Vars
x, y = Ints('x y')

# Specification
pre = Or(And(x < 0, y == -x),
         And(x >= 0, x == y))

goal = And(Or(y == -x, y == x),
           y >= 0)

# Check validity by searching for a counterexample to pre  $\rightarrow$  goal
s = Solver()
s.add(pre, Not(goal))
res = s.check()
```

Program  $c$ : a program state transformer  
Pre- and Post-condition  $P, Q$ : a Boolean function over program states

$$\forall x, P(x) \Rightarrow Q(c(x))$$

```
int abs(int x) {  
    {x: Int}  
    int y;  
    {x: Int, y: Int}  
    if (x >= 0) // B1  
        {x: Int, y: Int, x ≥ 0}  
        y = x; // B2  
        {x: Int, y: Int, x ≥ 0, x = y}  
    else  
        {x: Int, y: Int, x < 0}  
        y = -x; // B3  
        {x: Int, y: Int, x < 0, y = -x}  
  
    return y; // B4  
}
```

$\{x: \text{Int}, y: \text{Int}, (x < 0 \wedge y = -x) \vee (x \geq 0 \wedge x = y)\}$



Goal:  $(y = -x \vee y = x) \wedge y \geq 0$



**Z3 Theorem Prover!**



## Software Foundations: Hoare Logic (Coq)

```

{{ True }} =>
{{ m = m }}
X := m

                                {{ X = m }} =>
                                {{ X = m ∧ p = p }};

Z := p;

                                {{ X = m ∧ Z = p }} =>
                                {{ Z - X = p - m }}

while X ≠ 0 do

                                {{ Z - X = p - m ∧ X ≠ 0 }} =>
                                {{ (Z - 1) - (X - 1) = p - m }}

    Z := Z - 1

                                {{ Z - (X - 1) = p - m }};

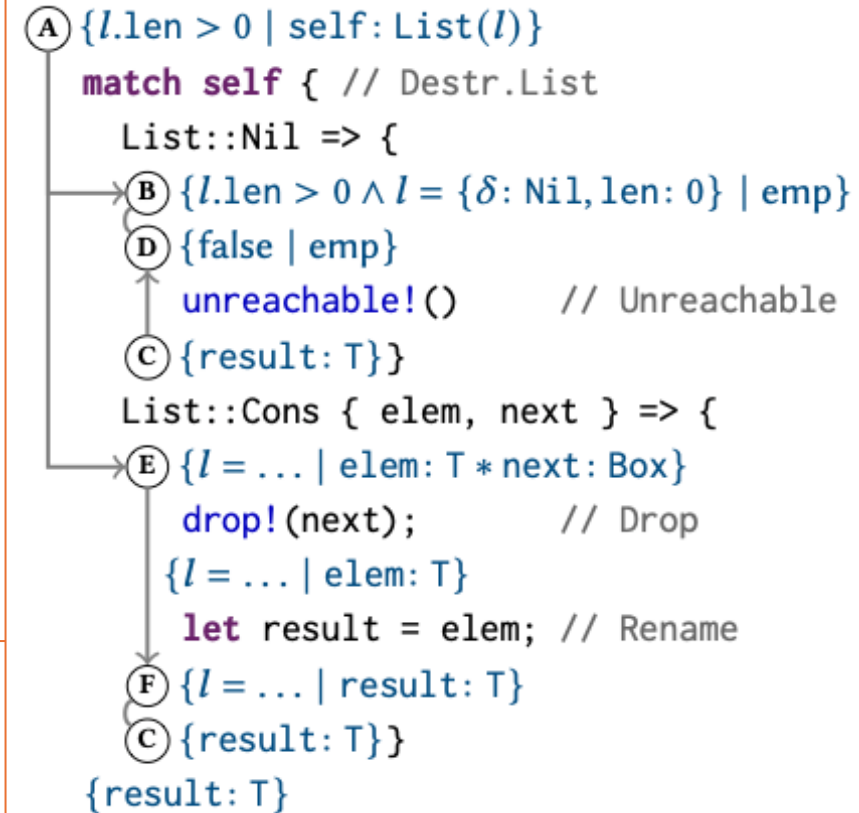
    X := X - 1

                                {{ Z - X = p - m }}

end

{{ Z - X = p - m ∧ ¬ (X ≠ 0) }} =>
{{ Z = p - m }}

```





## **Z3 Theorem Prover!**

Satisfiability Modulo Theories (SMT) Solver

Others: CVC3, CVC4, CVC5, Beaver, UCLID, veriT, ...



$$\{P\} c \{Q\}$$

Proof-Theoretic Semantics



## **Z3 Theorem Prover!**

Satisfiability Modulo Theories (SMT) Solver

Others: CVC3, CVC4, CVC5, Beaver, UCLID, veriT, ...

# Specification language

## Syntax

+

## Semantics

- First-order logic operators:
  - $\forall, \exists, =, \neq, \wedge, \vee, \neg, \Rightarrow, \dots$
- Base types, commonly used types, and their predicates
  - `Int`, `Bool`, `Set[T]`, `List[T]`, ...
  - `+`, `-`, `×`, `÷`, `::`, `U`, `∩`, `∈`, `[.]`, `&&`, `||`, `!`, `len`, ...
- Logic systems to specify program behaviors
  - Memory (Separation Logic): `{.}`, `↪`, `*`, ...
  - Temporal Behavior (Temporal Logic): `Globally`, `Next`, `Finally`, ...
  - Mathematical Objects: `Permutation`, ...

$$\{P\} c \{Q\}$$

Proof-Theoretic Semantics



# Functional Specification Guided Synthesis

## Structural Specification

Target Language

Syntax & Semantics

Types

# Functional Specification Guided Synthesis

## Structural Specification

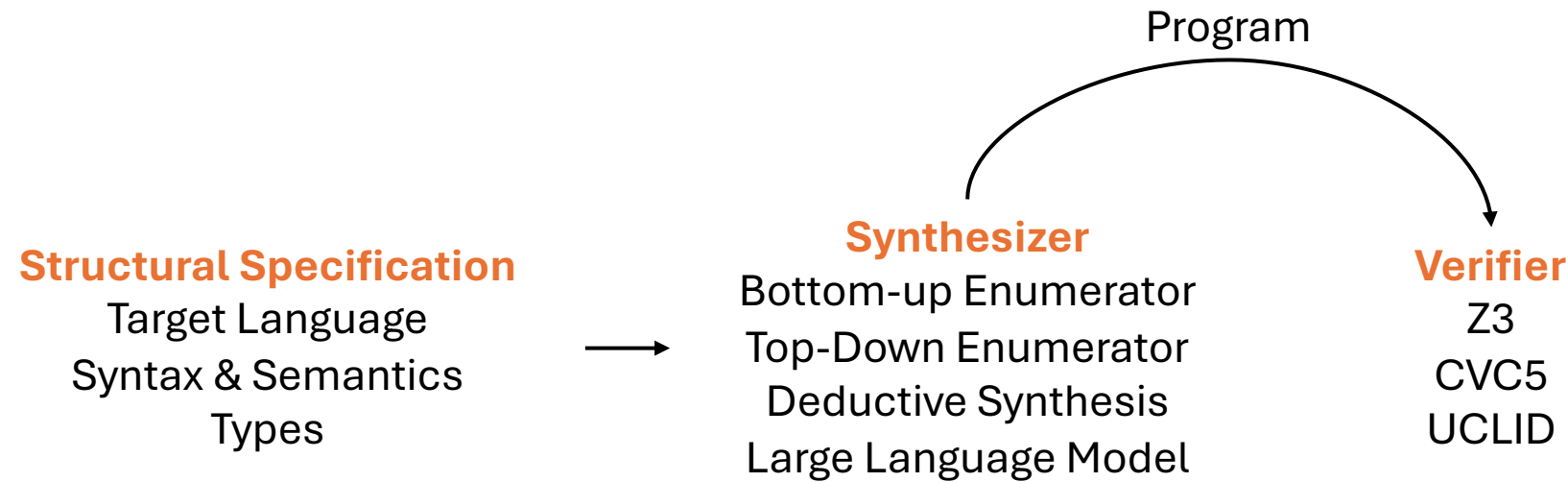
Target Language  
Syntax & Semantics  
Types



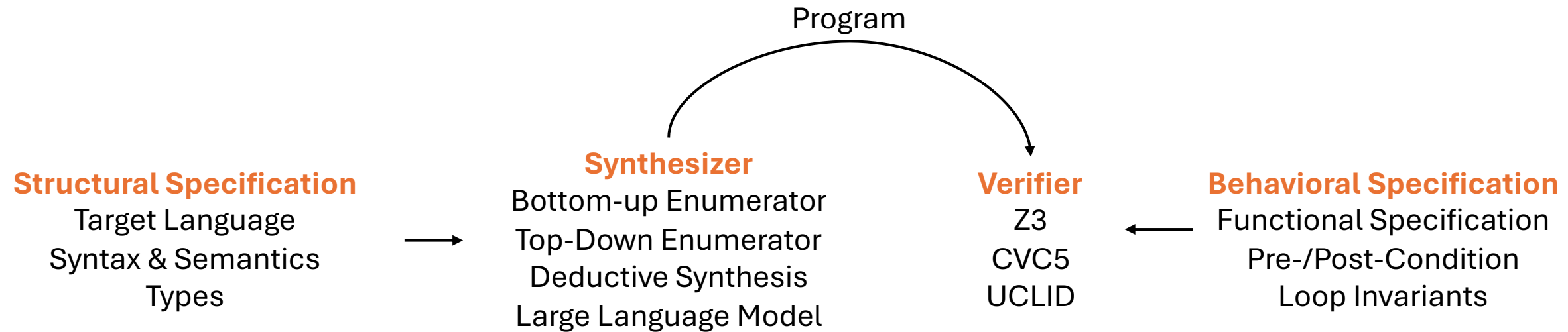
## Synthesizer

Bottom-up Enumerator  
Top-Down Enumerator  
Deductive Synthesis  
Large Language Model

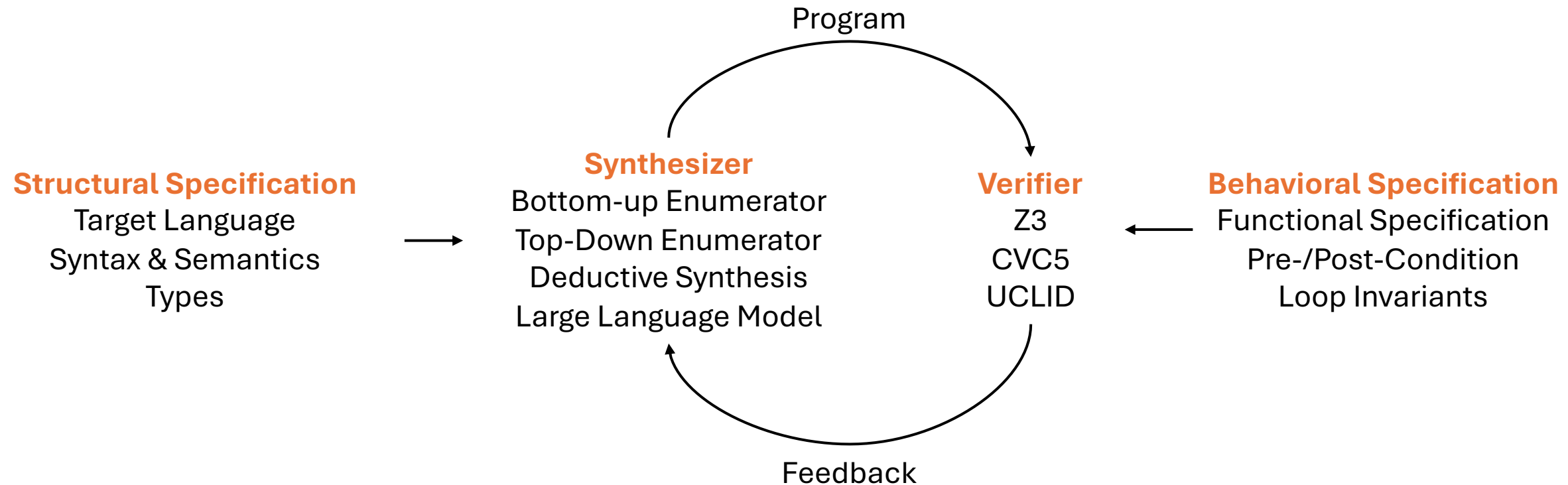
# Functional Specification Guided Synthesis



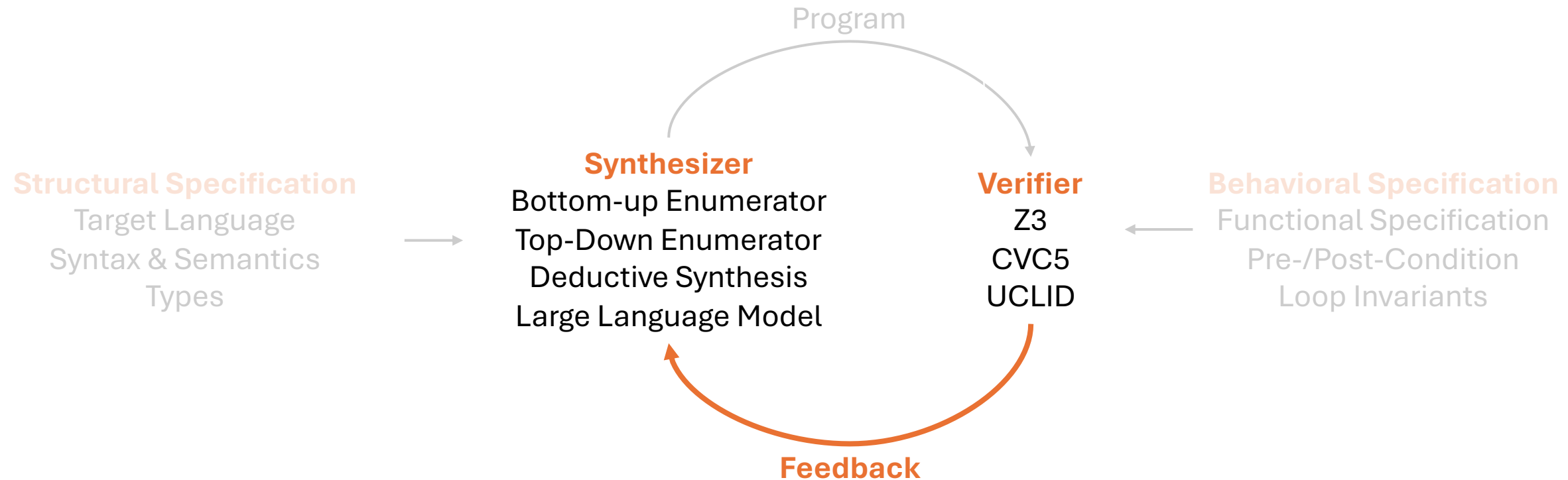
# Functional Specification Guided Synthesis



# Functional Specification Guided Synthesis



# Functional Specification Guided Synthesis






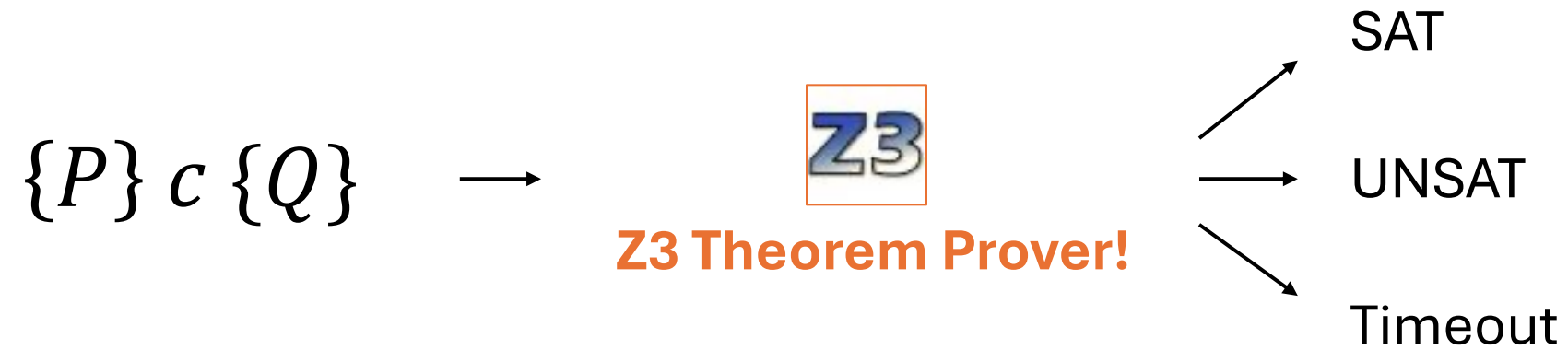
# Verifier's Feedback

$$\{P\} \ c \ \{Q\}$$

# Verifier's Feedback

$\{P\} c \{Q\} \longrightarrow$    
**Z3 Theorem Prover!**

# Verifier's Feedback



# Verifier's Feedback

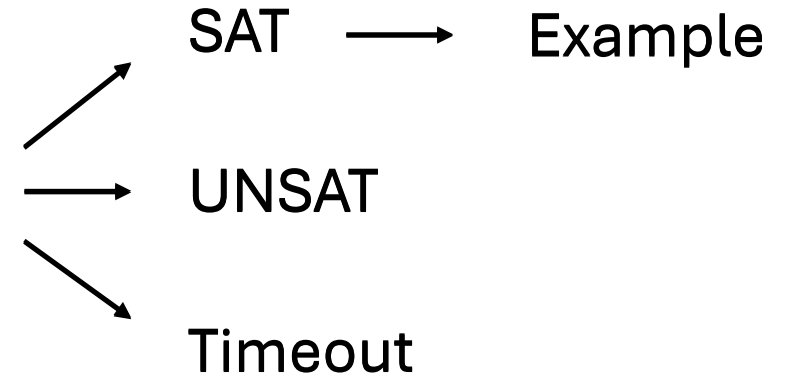


# Verifier's Feedback

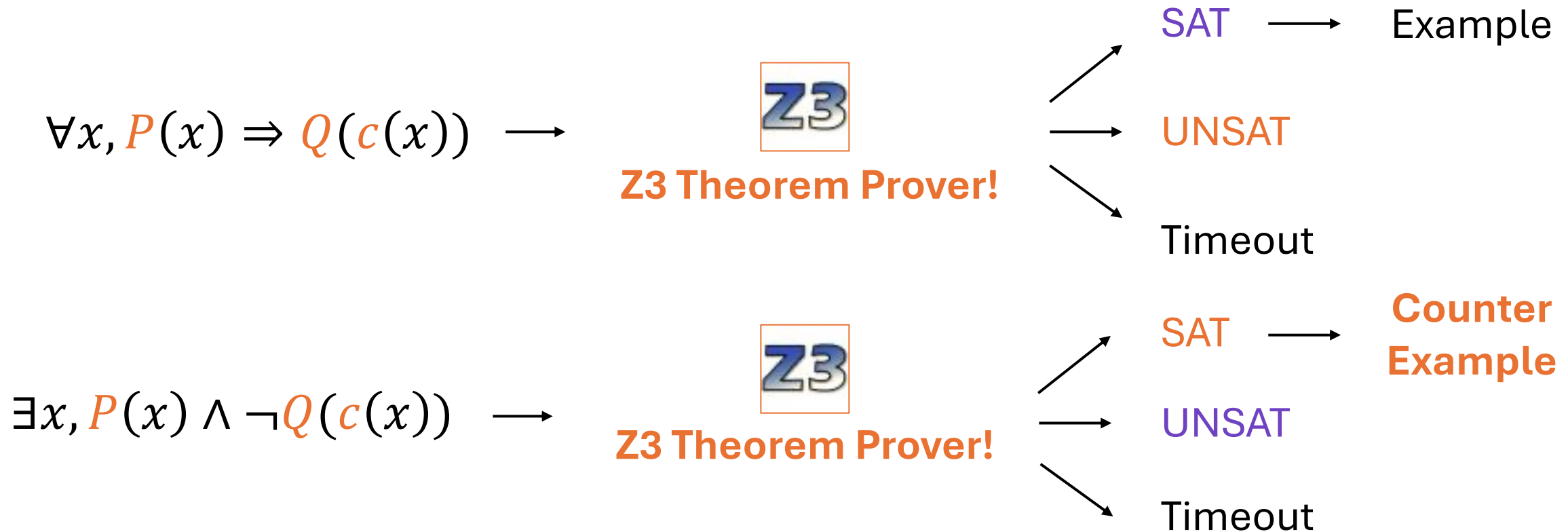
$\forall x, P(x) \Rightarrow Q(c(x)) \rightarrow$



**Z3 Theorem Prover!**



# Verifier's Feedback



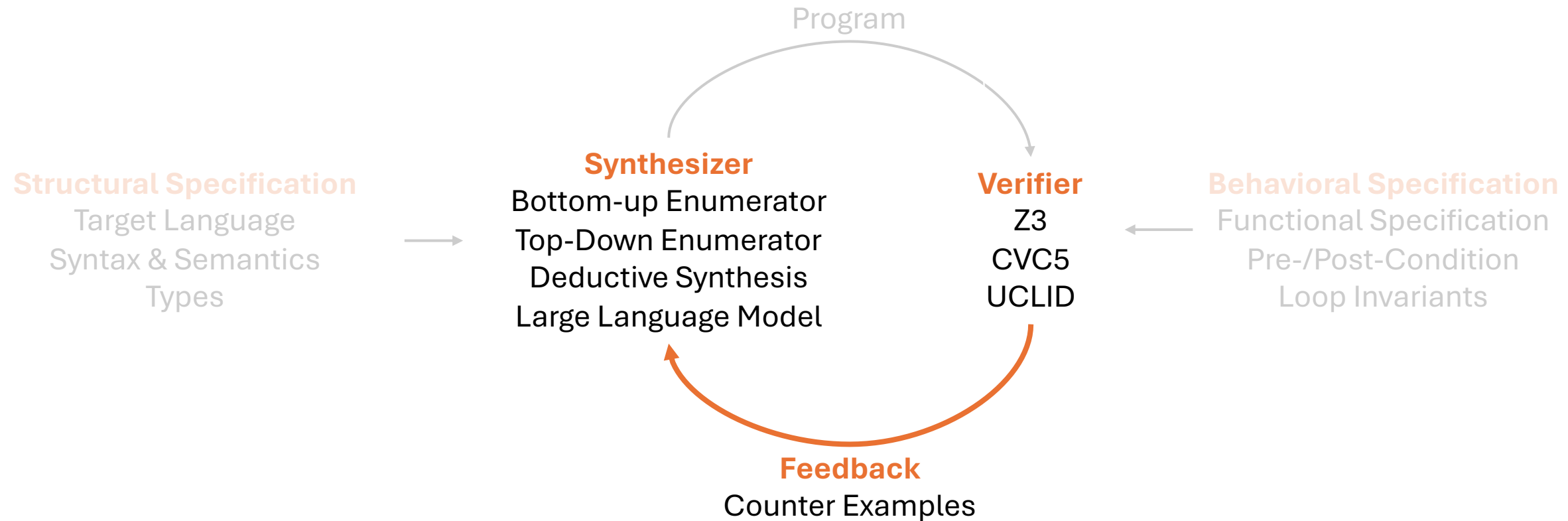
```
x, y = Ints('x y')

# Original pre and goal
pre = Or(And(x < 0, y == -x),
         And(x >= 0, x == y))

goal = And(Or(y == -x, y == x),
           y >= 0)

# 1) Try to find a counterexample to the original implication
s = Solver()
s.add(pre, Not(goal))
r = s.check()
print("Counterexample to original (pre → goal)?", r)
if r == sat:
    m = s.model()
    print("Counterexample model:", "x =", m[x], ", y =", m[y])
else:
    print("No counterexample; implication is VALID.")
```

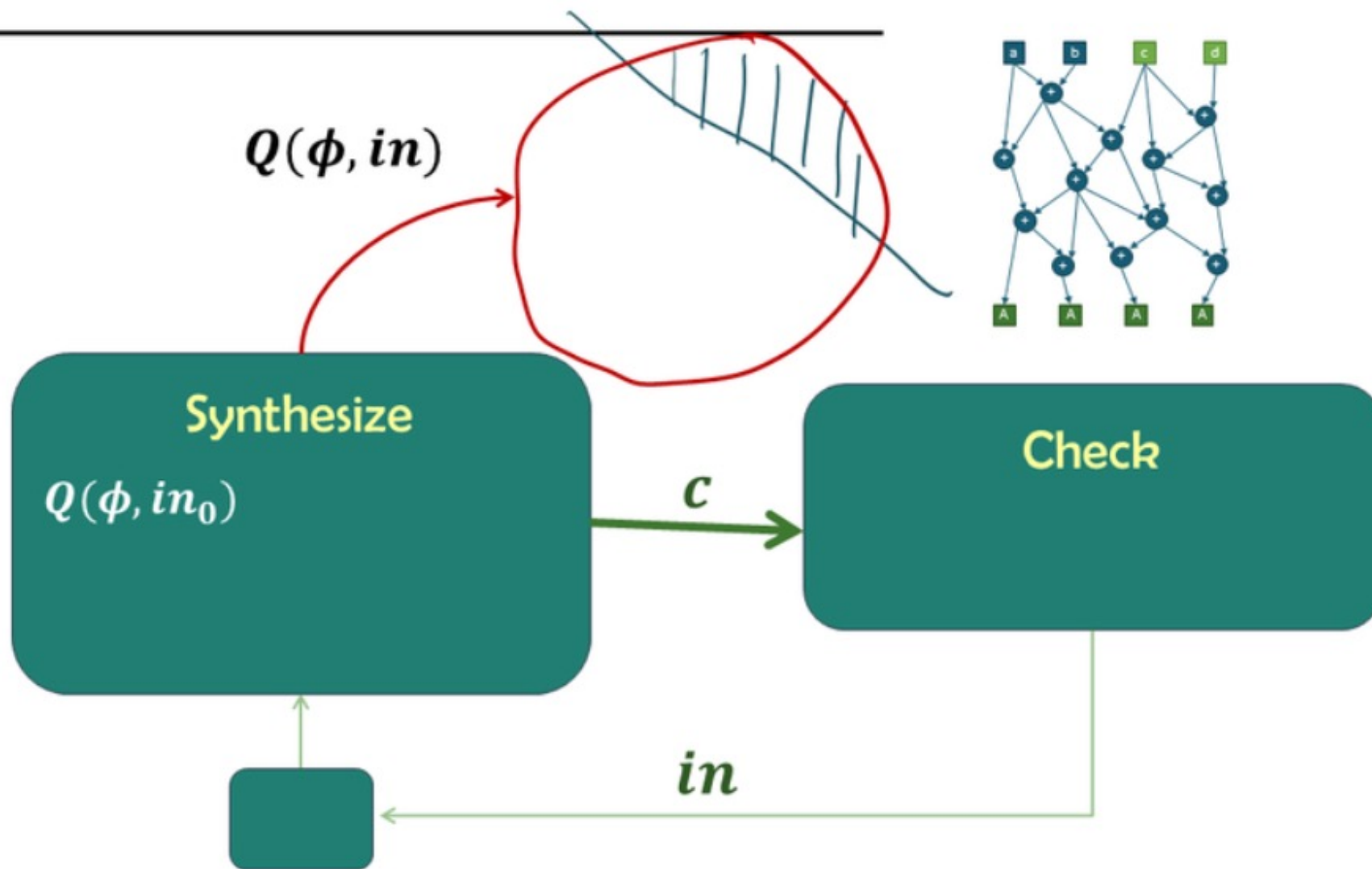
# Functional Specification Guided Synthesis



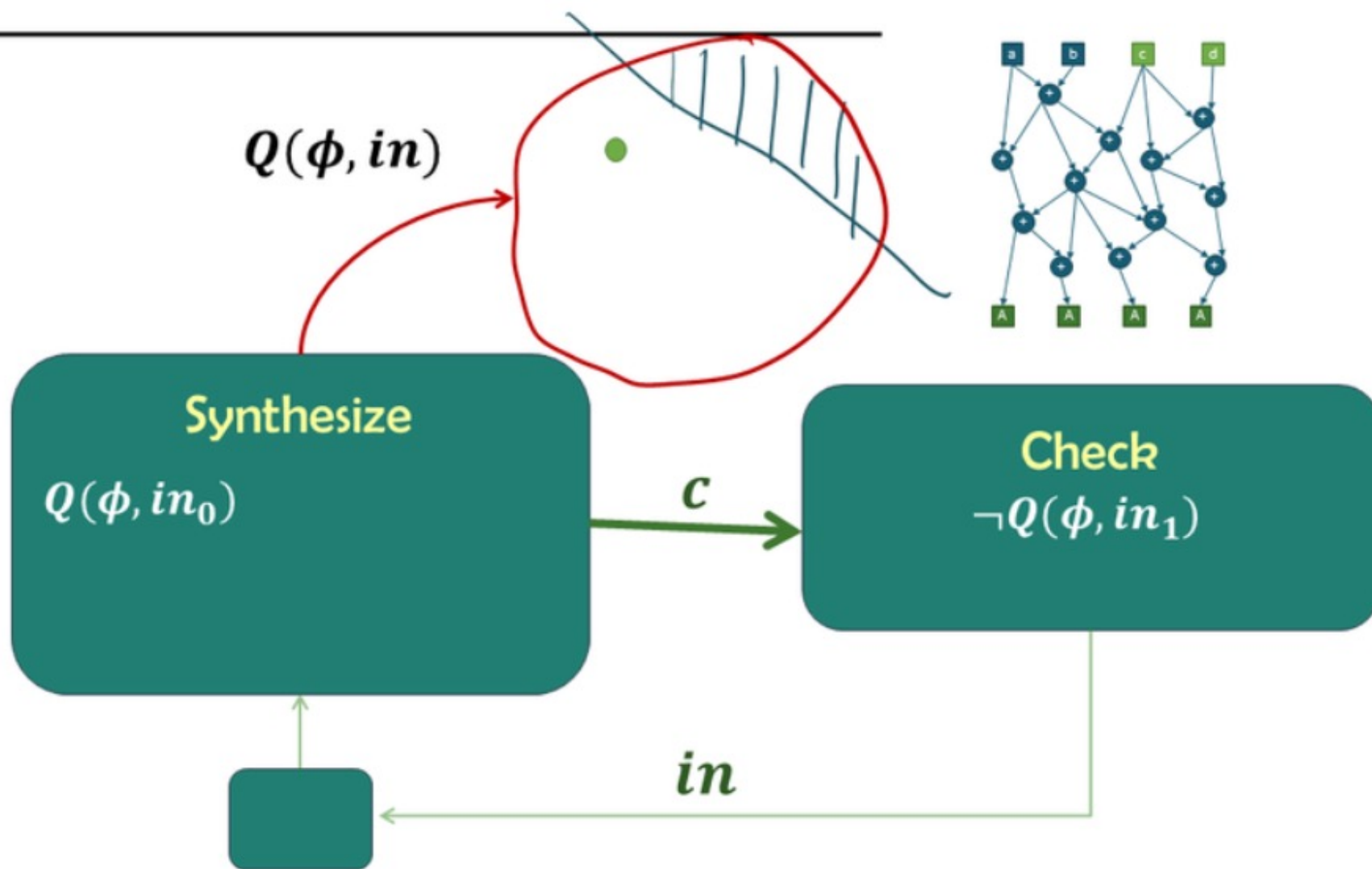




# Constraint-based CEGIS

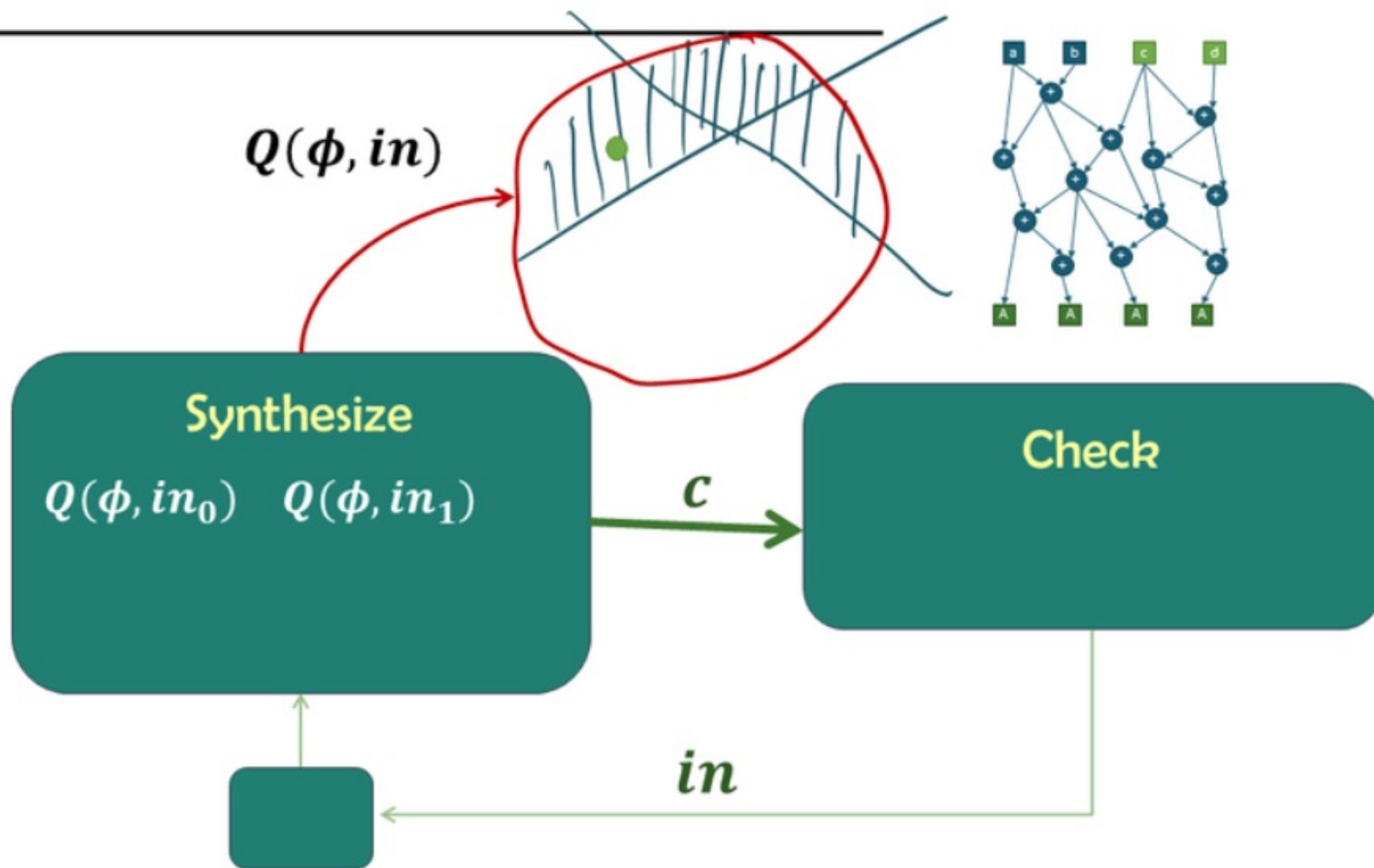


# Constraint-based CEGIS

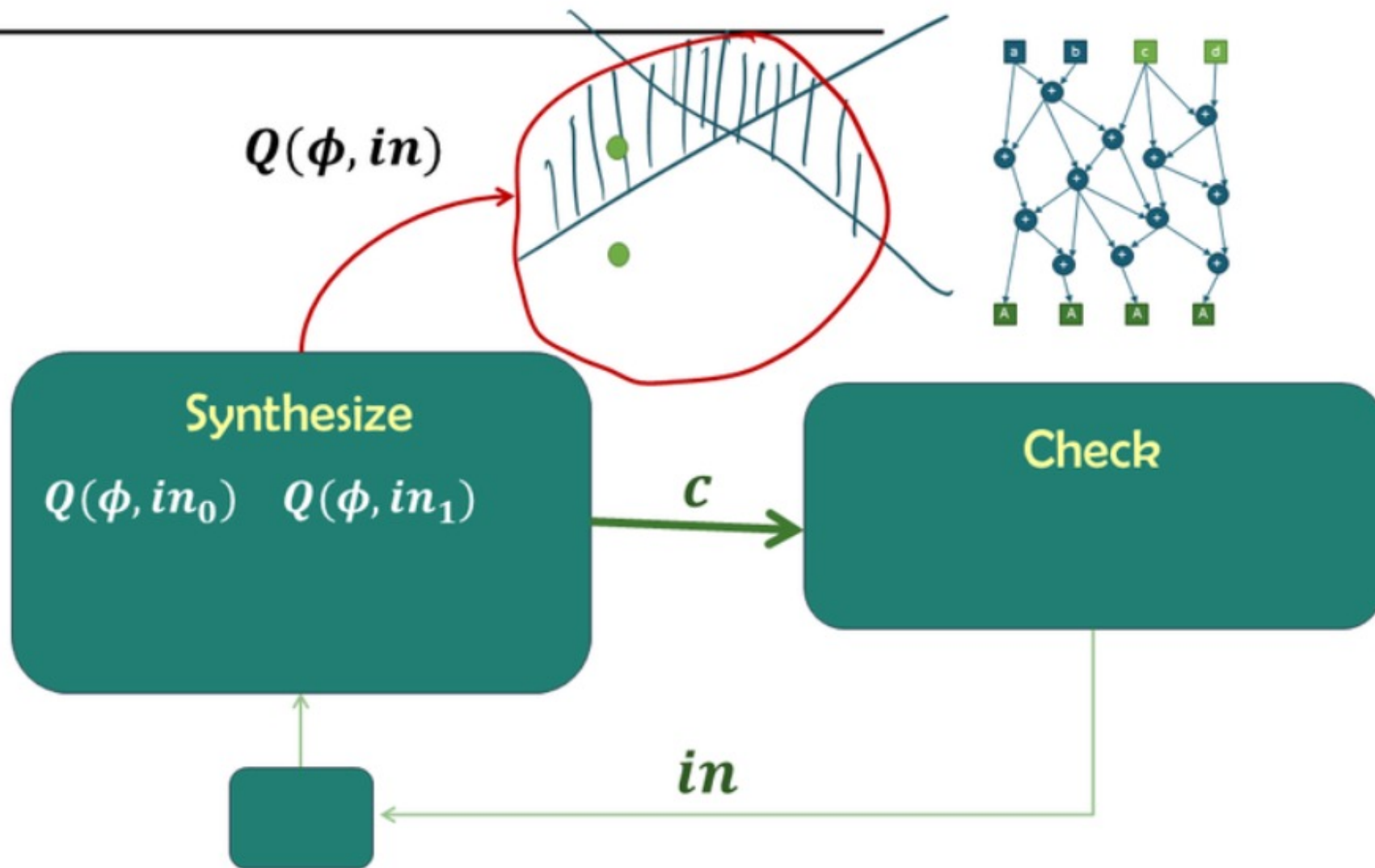




# Constraint-based CEGIS



# Constraint-based CEGIS



# Notes

- There is **NO FREE LUNCH!**
  - Where do you get the functional specifications?
    - Where do you get the language for functional specifications? Is it expressive enough to model program behaviors?
    - Is your functional specification complete?
  - How do you deal with Loops?
    - Loops need “Loop Invariants” for an automated theorem prover to function
    - How do you get the loop invariants?
  - Z3: How long does it take for Z3 to verify your  $\{P\} \text{ c } \{Q\}$  proposition?
    - Often time, Z3 cannot deal with extremely large programs, and will timeout
    - Who is doing the simplification?
  - Verification  $\leftrightarrow$  Synthesis interplay overhead and deficiency?

# Notes

- Refinement Type as Functional Specification:

```
def abs(x: int) -> int:
```

```
...
```

```
def abs(x: int) -> {v: int | v >= 0}:
```

```
...
```

# Notes

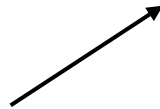
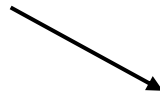
- “Multi-Modal” Program Synthesis

## Structural Specification

Target Language  
Syntax & Semantics  
Partial Program; Templates  
...

## Behavioral Specification

Examples  
Counter Examples  
Functional Specifications  
Refinement Types  
Natural Language  
...



## Synthesizer

Bottom-up Enumerator  
Top-Down Enumerator  
Deductive Synthesis  
Large Language Model

# Notes

- “Multi-Modal” Program Synthesis

## Structural Specification

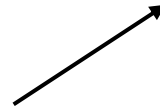
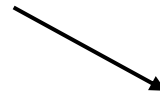
Target Language  
Syntax & Semantics  
Partial Program; Templates  
...

## Behavioral Specification

Examples  
Counter Examples  
Functional Specifications  
Refinement Types

## Natural Language

...



## Synthesizer

Bottom-up Enumerator  
Top-Down Enumerator  
Deductive Synthesis  
Large Language Model



# Week 2

- Assignment 1
  - <https://github.com/machine-programming/assignment-1>
  - Autograder Up on GradeScope (Finally!)
  - There is OOM issue on GradeScope that you need to manage!
  - Due Date Sep 16 (Extended for 5 days from Sep 11)
- Logistics
  - There should be an increase in class capacity (25-30)
  - Everyone who has been following should be able to get in
  - Send me an email if you are newly enrolled