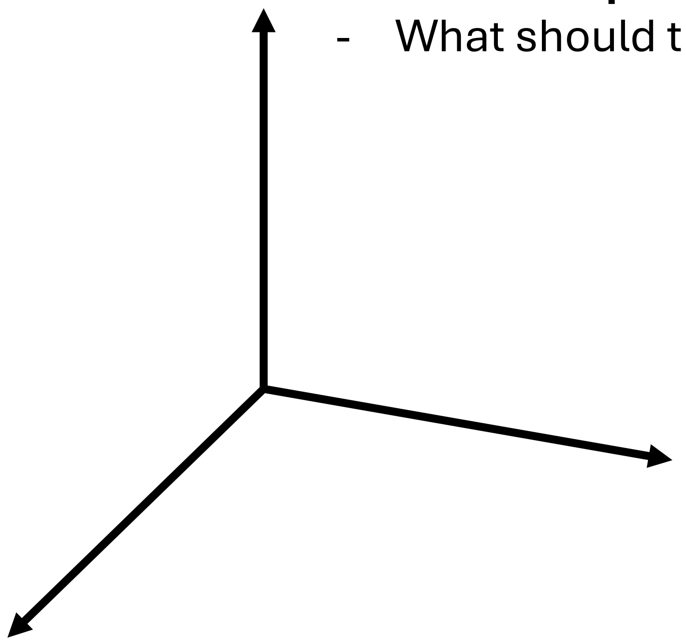
Machine Programming

Lecture 4 – Functional Specifications for Synthesis

Ziyang Li

Dimensions in Program Synthesis

Behavioral Specification
- What should the program do?



Synthesis Strategy
- How do we find such a program?

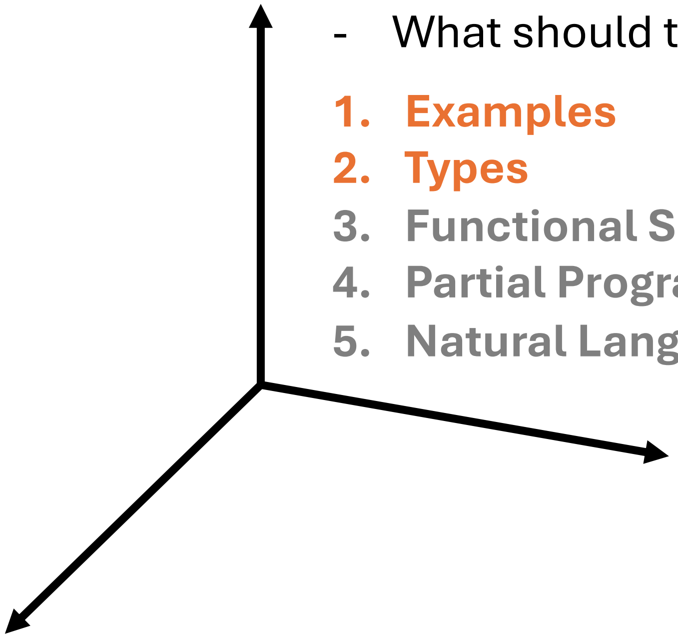
Structural Specification
- What is the space of the programs?

The Course So Far

Behavioral Specification

- What should the program do?

1. **Examples**
2. **Types**
3. **Functional Specifications**
4. **Partial Programs**
5. **Natural Language**



Synthesis Strategy

- How do we find such a program?

Enumeration

- Enumerating all programs with a grammar
- Bottom-up vs top-down

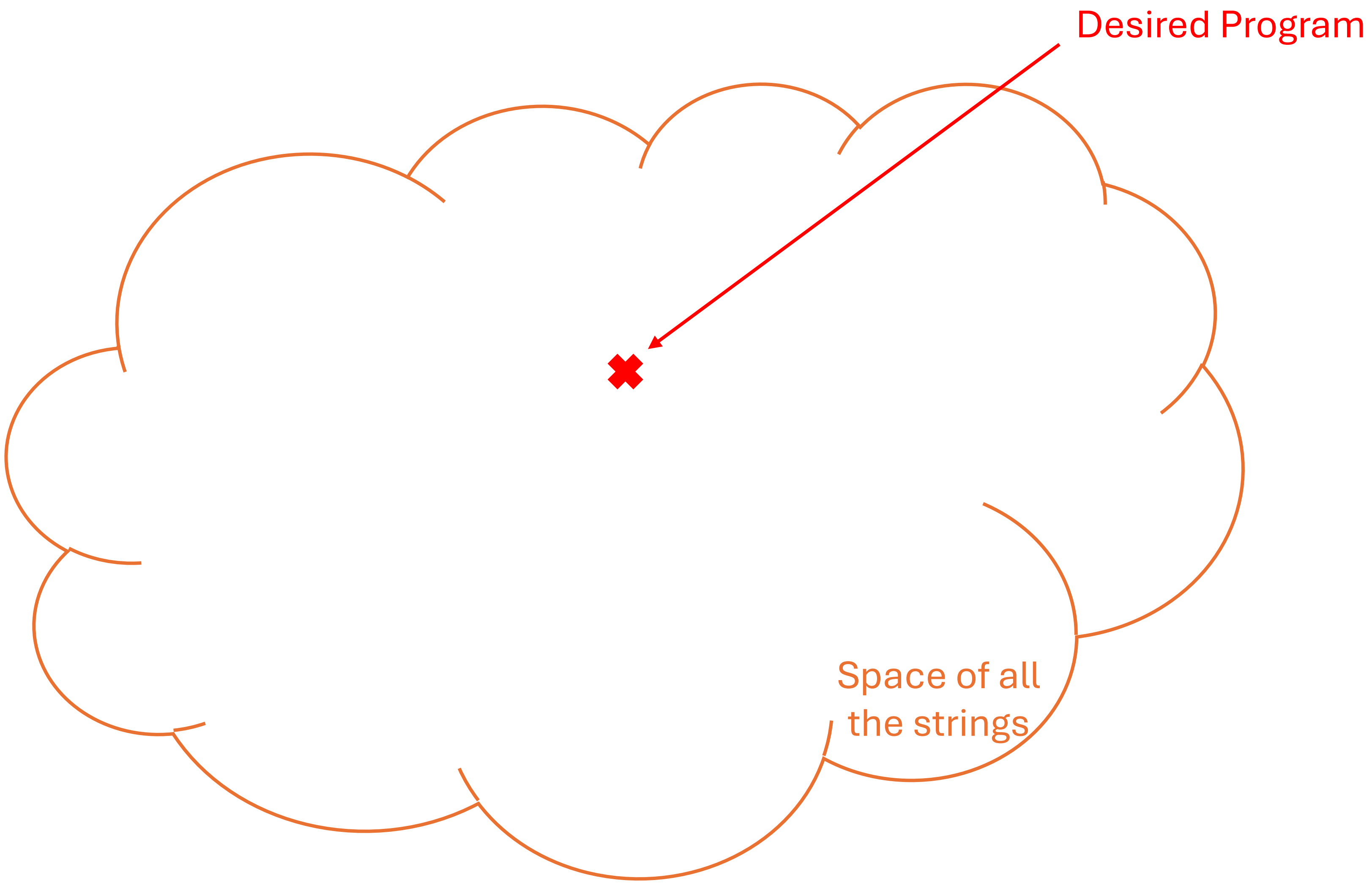
Structural Specification

- What is the space of the programs?

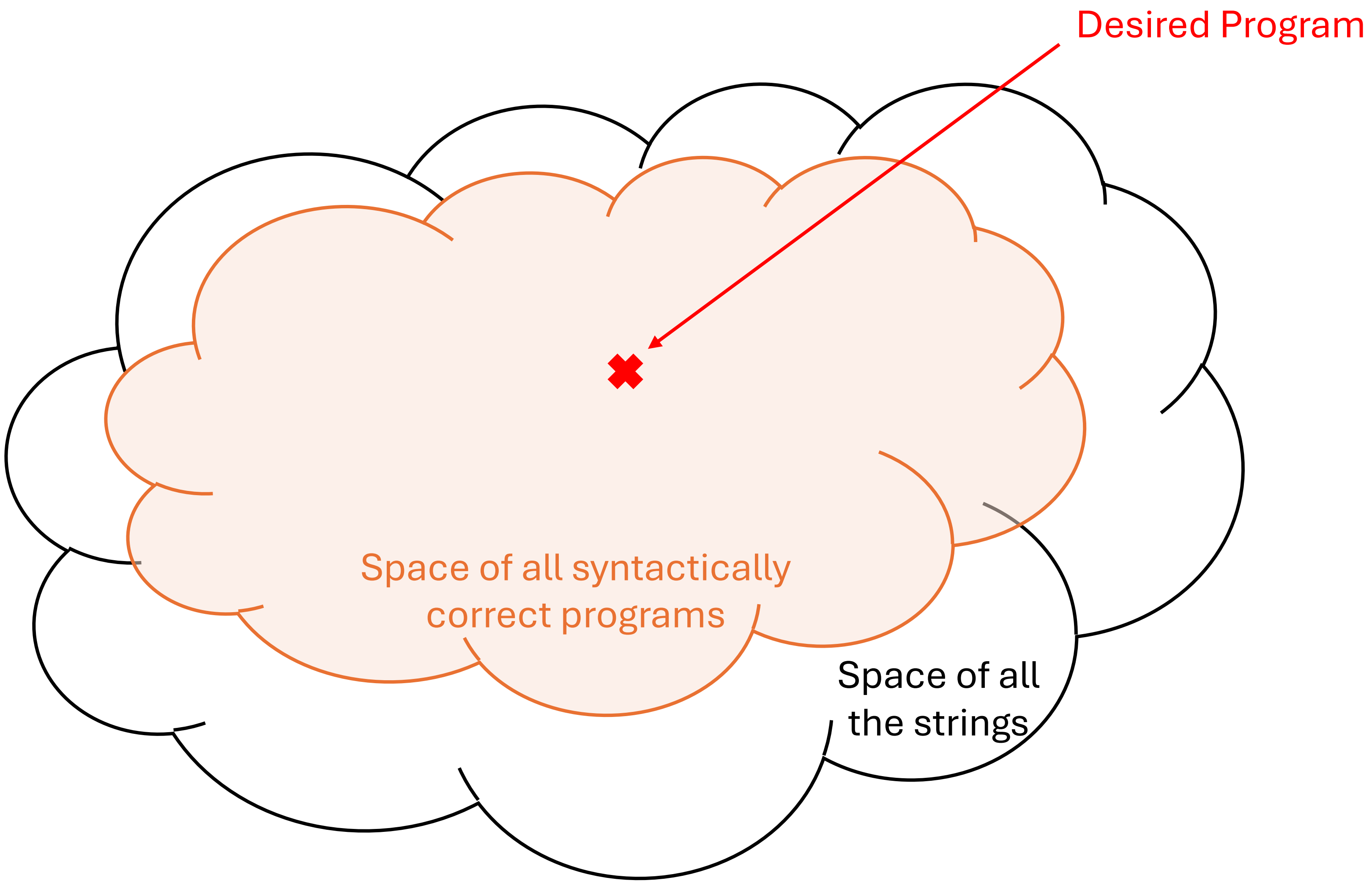
Context-Free / Regular Tree Grammar

Expr e ::= c | e + e | e * e

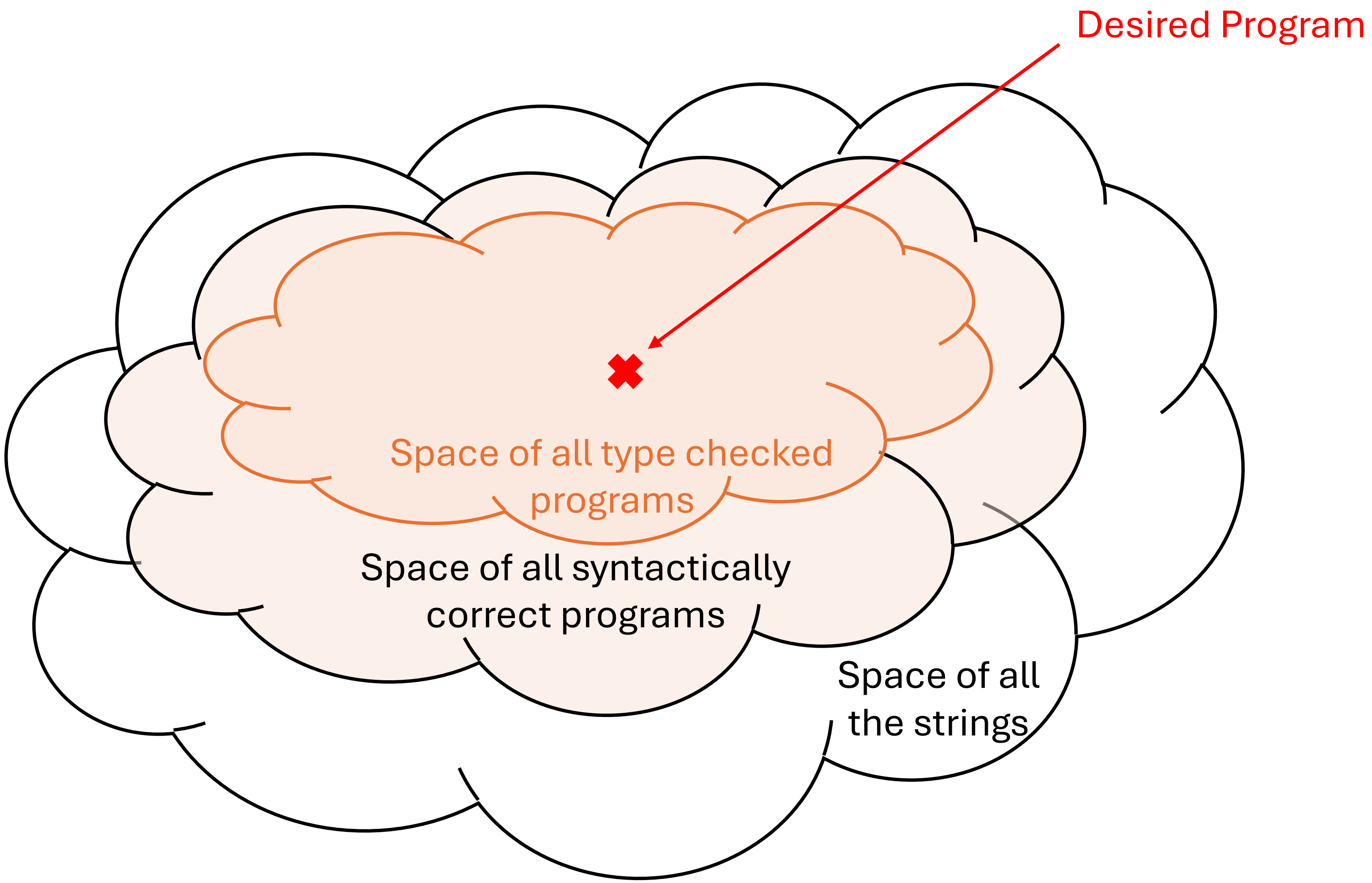
High Level Picture



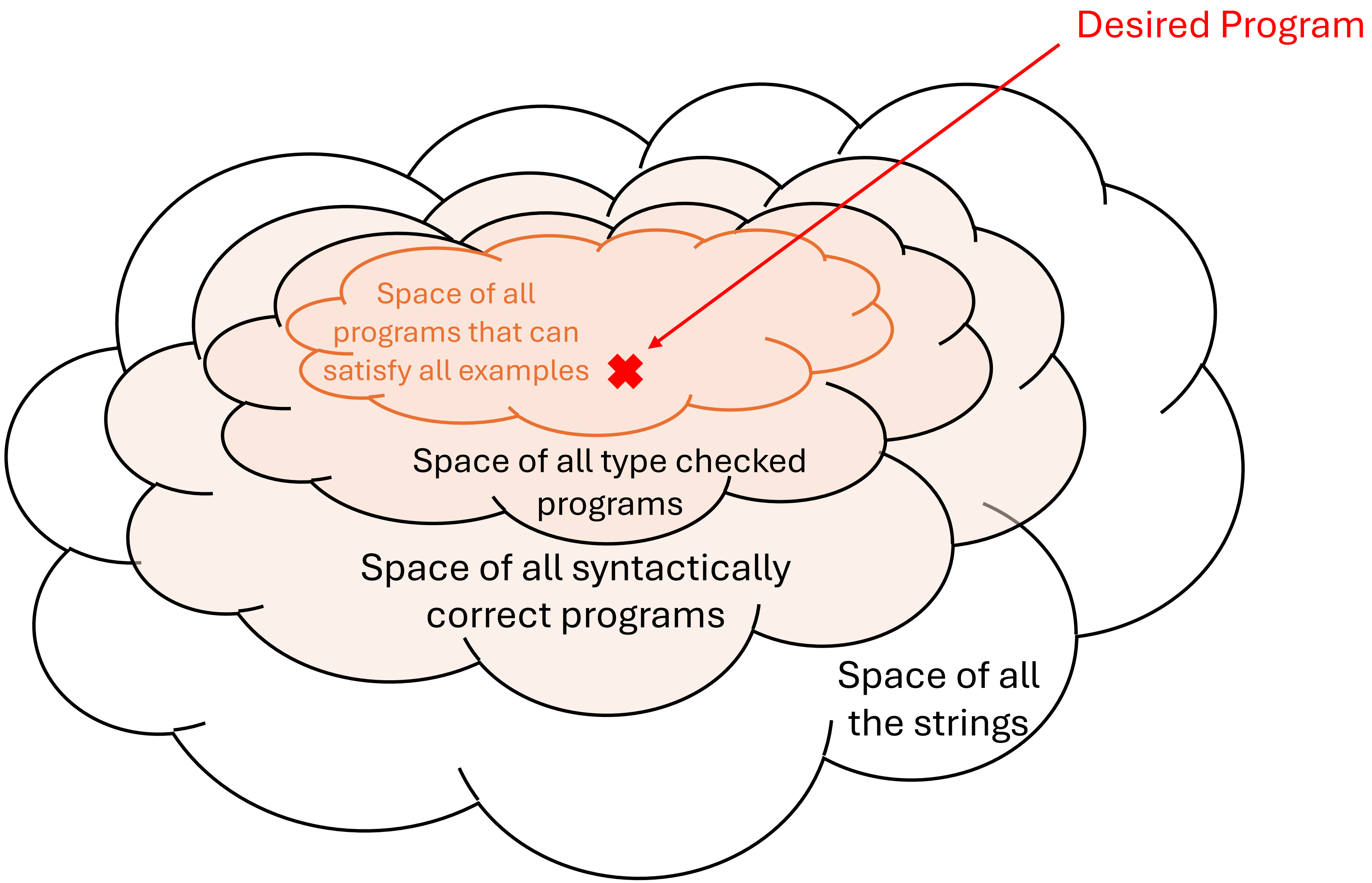
High Level Picture



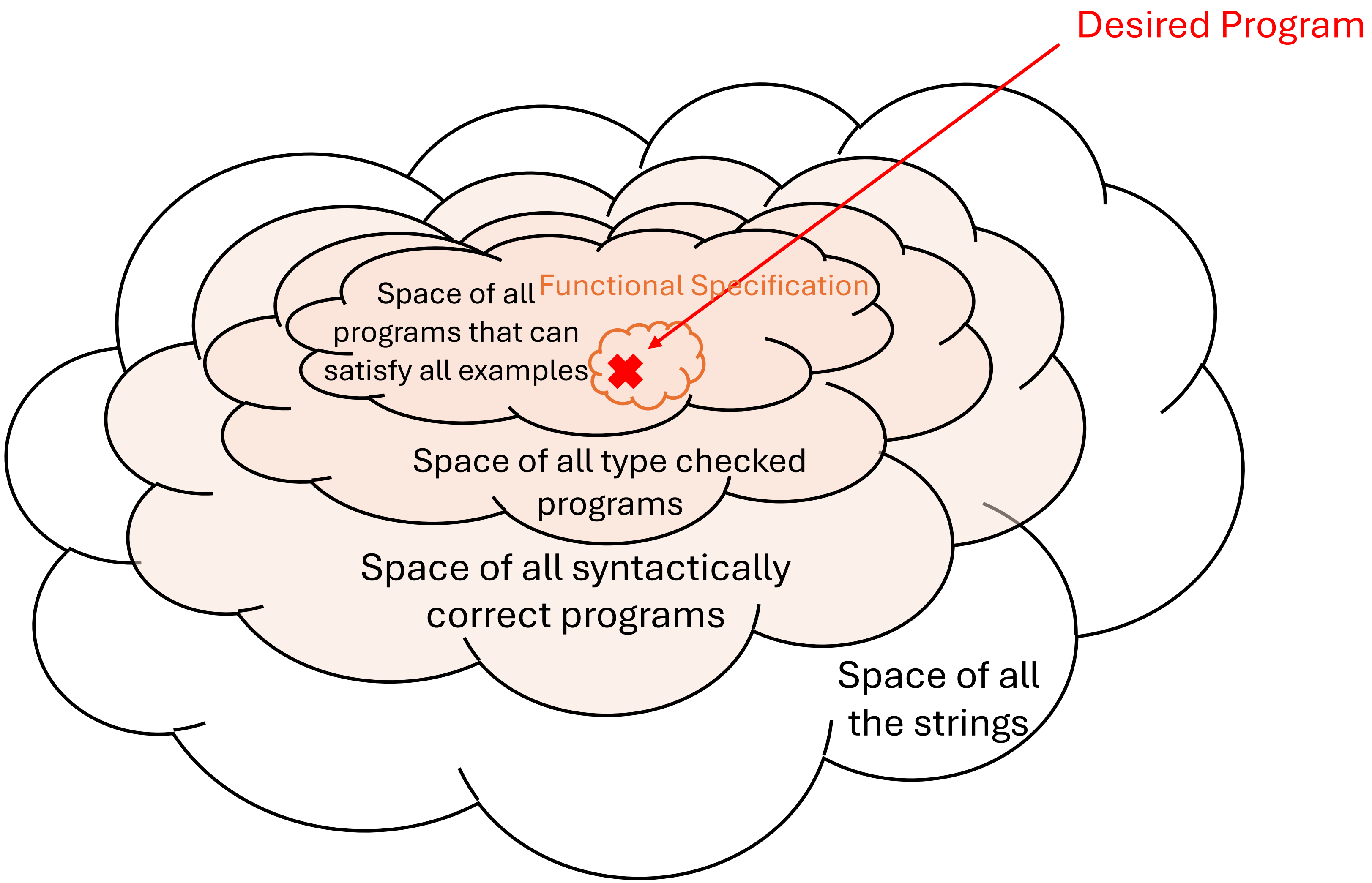
High Level Picture



High Level Picture



High Level Picture



Today

Behavioral Specification

- What should the program do?

1. Examples
2. Types
3. Functional Specifications
4. Parametric Programs
5. Natural Language

Synthesis Strategy

- How do we find such a program?

Structural Specification

- What is the space of the programs?

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

$$\forall x, y \in \text{List[Int]}, \boldsymbol{reverse}(x) = y \Rightarrow$$
$$len(x) = len(y) \land \forall i \in \{1 \ldots len(x)\}, x_i = y_{len(y)+1-i}$$

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

$$\forall x, y \in \text{List[Int]}, \boldsymbol{sort}(x) = y \Rightarrow$$
$$len(x) = len(y) \land \forall i \in [0, len(x) - 1), y_{i+1} > y_i$$

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

Functional Specifications

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

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

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([]) = []
```

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

$$len(x) = len(y) \land \forall i \in [0, len(x) - 1), y_{i+1} > y_i$$

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

Functional Specifications

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

$$len(x) = len(y) \land \forall i \in \{1 \ldots len(x)\}, x_i = y_{len(y)+1-i}$$

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([]) = []
```

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

$$len(x) = len(y) \land \forall i \in [0, len(x) - 1), y_{i+1} > y_i$$

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

$$\forall x, y \in \text{List[Int]}, \boldsymbol{sort}(x) = y \Rightarrow$$
$$len(x) = len(y) \wedge \forall i \in [0, len(x) - 1), y_{i+1} > y_i$$

Pre-condition:

Post-condition:

Functional Specifications

Pre-condition:

$x \in \text{List[Int]}$ x is an integer list

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

$len(x) = len(y) \wedge \forall i \in [0, len(x) - 1), y_{i+1} > y_i$

Post-condition:

Functional Specifications

$$\forall x, y \in \text{List[Int]}, \boldsymbol{sort}(x) = y \Rightarrow$$
$$len(x) = len(y) \wedge \forall i \in [0, len(x) - 1), y_{i+1} > y_i$$

Pre-condition:

$x \in \text{List[Int]}$	x is an integer list

Post-condition:

$y \in \text{List[Int]}$	y is an integer list
$len(x) = len(y)$	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

Premise: All valid inputs to a function MUST satisfy

Pre-condition:

$x \in \text{List[Int]}$	x is an integer list

Promise: all outputs WILL satisfy if the premise holds

$$\forall x, y \in \text{List[Int]}, \boldsymbol{sort}(x) = y \Rightarrow$$
$$len(x) = len(y) \wedge \forall i \in [0, len(x) - 1), y_{i+1} > y_i$$

Post-condition:

$y \in \text{List[Int]}$	y is an integer list
$len(x) = len(y)$	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

Pre- and Post-Conditions in the Wild

Dafny

```
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;
}
```

Checked-C

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

SV-Comp (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;
}
```

Russol (Rust)

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

Software Foundations / Hoare Logic (Coq)

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

Pre- and Post-Condition Practice

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

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

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

Completeness of Pre- and Post-Conditions

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

```
y = sort(x)
```

Is this post-condition complete?

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

Completeness of Pre- and Post-Conditions

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

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

```
y = sort(x)
```

Is this post-condition complete? ❌

Post-condition $y \in \text{List[Int]}$ $\forall i.\, 0 \le i < n - 1 \Rightarrow y[i] \le y[i+1]$

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

Completeness of Pre- and Post-Conditions

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

```
y = sort(x)
```

Attempt 2:
Is this post-condition complete?

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

$len(y) = n$

Completeness of Pre- and Post-Conditions

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

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

```
y = sort(x)
```

Attempt 2:
Is this post-condition complete? ❌

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

$len(y) = n$

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

Completeness of Pre- and Post-Conditions

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

```
y = sort(x)
```

Attempt 3:
Is this post-condition complete?

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

$len(y) = n$

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

Completeness of Pre- and Post-Conditions

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

```
y = sort(x)
```

Attempt 3:
Is this post-condition complete?

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

$len(y) = n$

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

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

Completeness of Pre- and Post-Conditions

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

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

```
y = sort(x)
```

Attempt 3:
Is this post-condition complete? ❌

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

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

$len(y) = n$

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

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

Completeness of Pre- and Post-Conditions

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

```
y = sort(x)
```

Attempt 4:
Is this post-condition complete? ✅

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

$len(y) = n$

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

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 $\boxed{\text{required annotations are on average 27\% shorter than the code.}}$

Specification itself has a language

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

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

Structuring the Synthesis of Heap-Manipulating Programs, Polikarpova et. al., 2019

Specification itself has a language

Syntax + Semantics

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

Specification itself has a language

Syntax + Semantics

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

???

Specification itself has a language

Syntax + Semantics

Syntax

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

Semantics

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

Hoare Triple

$$[[c]] \vDash Q$$

Operational Alignment

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)` $\{x \mapsto b * y \mapsto a\}$

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

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

```
VERIFIER_assume(true);
while (X ≠ 0) {
  X = X - 1;
}
VERIFIER_assert(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 $\quad x$: Int

c

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

Q $\quad (y = -x \lor y = x) \land y \geq 0$

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) {
```

$\{x: \text{Int}\}$

```
  int y;

  if (x >= 0) // B1

    y = x; // B2

  else

    y = -x; // B3

  return y; // B4
}
```

Goal: $(y = -x \lor y = x) \land y \geq 0$

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) {
```

$\{x\colon \mathrm{Int}\}$

```
  int y;
```

$\{x\colon \mathrm{Int}, y\colon \mathrm{Int}\}$

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

    y = x; // B2

  else

    y = -x; // B3

  return y; // B4
}
```

Goal: $(y = -x \lor y = x) \land y \geq 0$

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: $(y = -x \lor y = x) \land y \geq 0$

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) {
```

$\{x: \mathrm{Int}\}$

```
  int y;
```

$\{x: \mathrm{Int}, y: \mathrm{Int}\}$

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

$\{x: \mathrm{Int}, y: \mathrm{Int}, x \geq 0\}$

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

$\{x: \mathrm{Int}, y: \mathrm{Int}, x \geq 0, x = y\}$

```
  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) {
```

$\{x: \mathrm{Int}\}$

```
int y;
```

$\{x: \mathrm{Int}, y: \mathrm{Int}\}$

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

$\{x: \mathrm{Int}, y: \mathrm{Int}, x \geq 0\}$

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

$\{x: \mathrm{Int}, y: \mathrm{Int}, x \geq 0, x = y\}$

```
else
```

$\{x: \mathrm{Int}, y: \mathrm{Int}, x < 0\}$

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

$\{x: \mathrm{Int}, y: \mathrm{Int}, x < 0, y = -x\}$

$\{x: \mathrm{Int}, y: \mathrm{Int}, (x < 0 \land y = -x) \lor (x \geq 0 \land x = y)\}$

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

Goal: $(y = -x \lor y = x) \land y \geq 0$

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: $(y = -x \lor y = x) \land y \geq 0$

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
}
```

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

$$\Downarrow$$

Goal: $(y = -x \lor y = x) \land y \geq 0$

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
}
```

$\{x\colon \mathrm{Int}, y\colon \mathrm{Int}, (x < 0 \land y = -x) \lor (x \geq 0 \land x = y)\}$

$\Downarrow$

Goal: $(y = -x \lor y = x) \land y \geq 0$

?

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
}
```

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

$$\Downarrow$$

Goal: $(y = -x \lor y = x) \land y \geq 0$

Z3 Theorem Prover!

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

$\{x{:}\mathrm{Int}, y{:}\mathrm{Int}, (x<0 \wedge y=-x) \vee (x \geq 0 \wedge x=y)\}$ => 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 ∧ ¬goal`. If that is **unsat**, then the implication holds for all integers `x, y`.

python Copy code

```python
# 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 → 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
}
```

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

$$\Downarrow$$

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 }}
```

```
A {l.len > 0 | self: List(l)}
  match self { // Destr.List
    List::Nil => {
    B {l.len > 0 ∧ l = {δ: Nil, len: 0} | emp}
    D {false | emp}
      unreachable!()       // Unreachable
    C {result: T}}
    List::Cons { elem, next } => {
    E {l = ... | elem: T ∗ next: Box}
      drop!(next);         // Drop
     {l = ... | elem: T}
      let result = elem;  // Rename
    F {l = ... | result: T}
    C {result: T}}
  {result: T}
```

Leveraging Rust Types for Program Synthesis (Fiala et. al., 2023)

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

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

Hoare Triple

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

Program

Structural Specification
Target Language
Syntax & Semantics
Types

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

Verifier
Z3
CVC5
UCLID

Functional Specification Guided Synthesis

Program

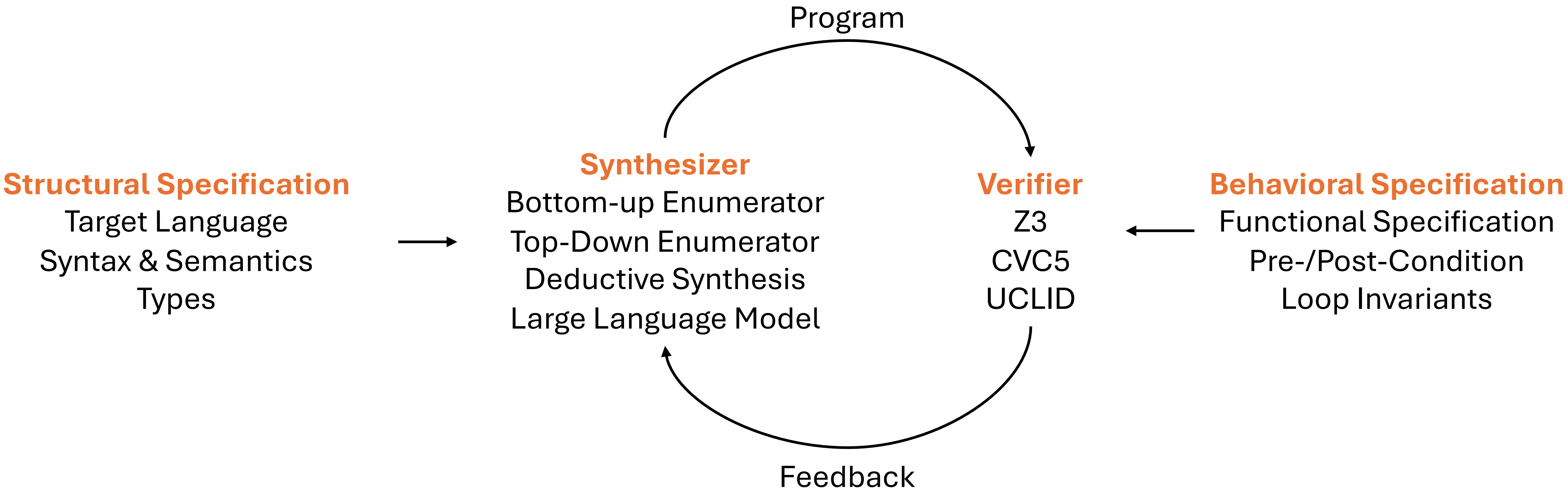
Structural Specification
Target Language
Syntax & Semantics
Types

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

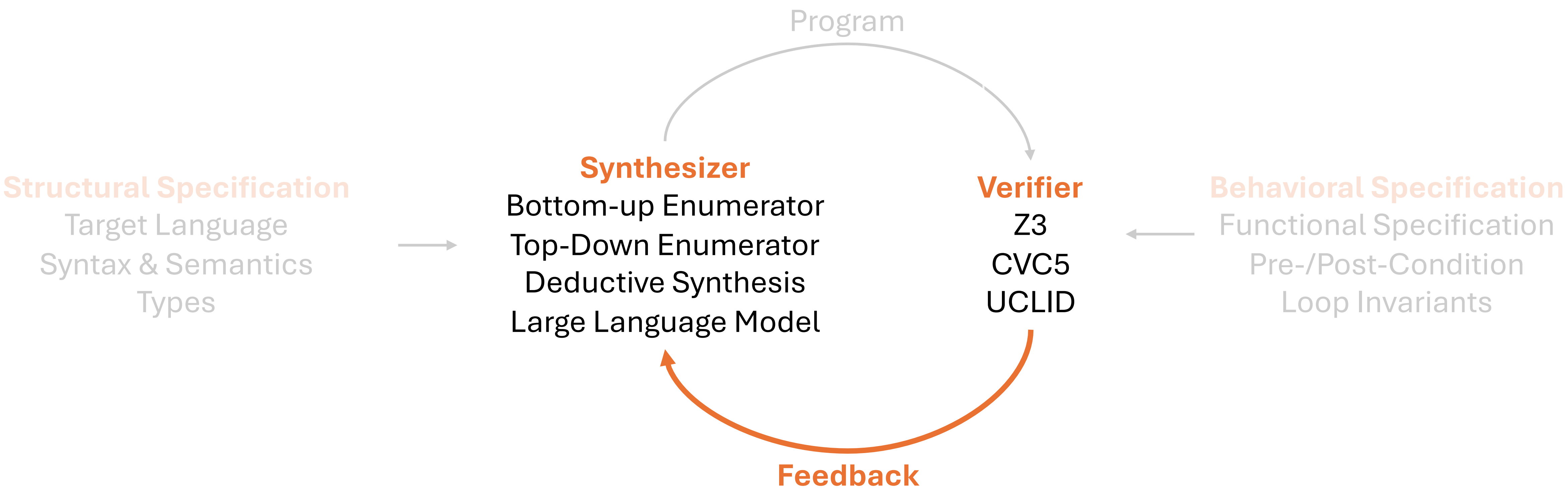
Verifier
Z3
CVC5
UCLID

Behavioral Specification
Functional Specification
Pre-/Post-Condition
Loop Invariants

Functional Specification Guided Synthesis



Functional Specification Guided Synthesis



Verifier’s Feedback

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

Verifier’s Feedback

$\{P\}\ c\ \{Q\} \quad \longrightarrow$

Z3 Theorem Prover!

Verifier's Feedback

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

Z3 Theorem Prover!

SAT

UNSAT

Timeout

Verifier’s Feedback

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

- $\nearrow$ SAT $\longrightarrow$ Example
- $\rightarrow$ UNSAT
- $\searrow$ Timeout

Verifier's Feedback

$\forall x, P(x) \Rightarrow Q(c(x)) \longrightarrow$ **Z3 Theorem Prover!**

- SAT $\longrightarrow$ Example
- UNSAT
- Timeout

Verifier's Feedback

$\forall x, P(x) \Rightarrow Q(c(x)) \longrightarrow$ Z3 Theorem Prover!

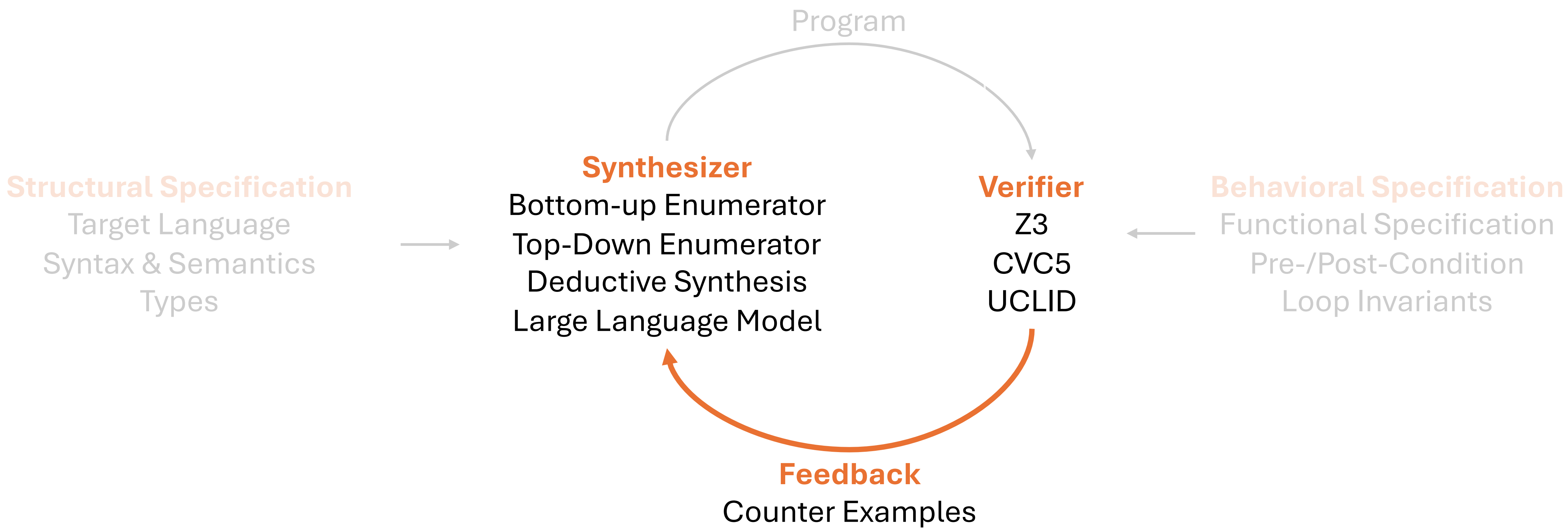
- SAT $\longrightarrow$ Example
- UNSAT
- Timeout

$\exists x, P(x) \land \neg Q(c(x)) \longrightarrow$ Z3 Theorem Prover!

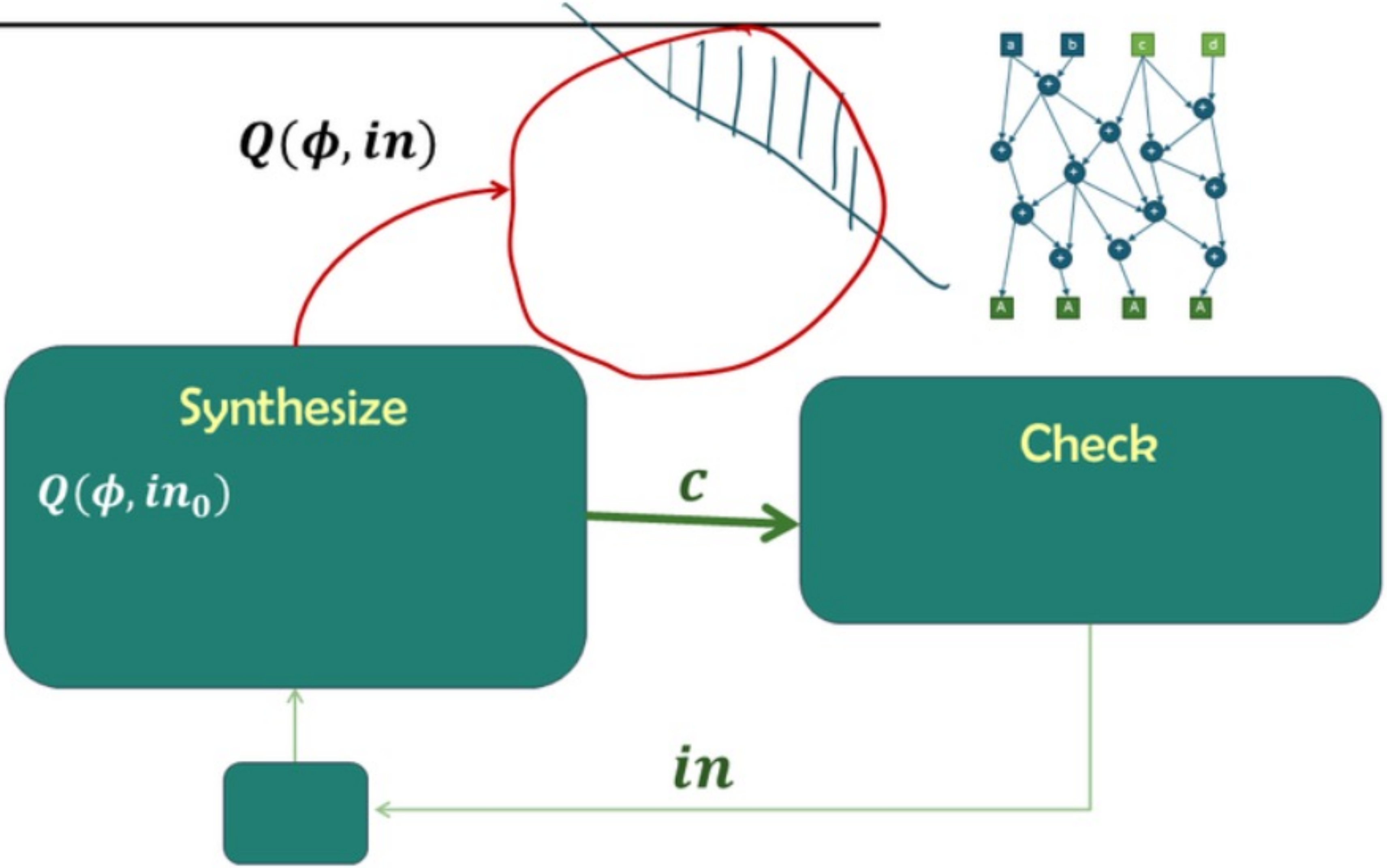
- SAT $\longrightarrow$ **Counter Example**
- UNSAT
- Timeout

```python
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.")
```

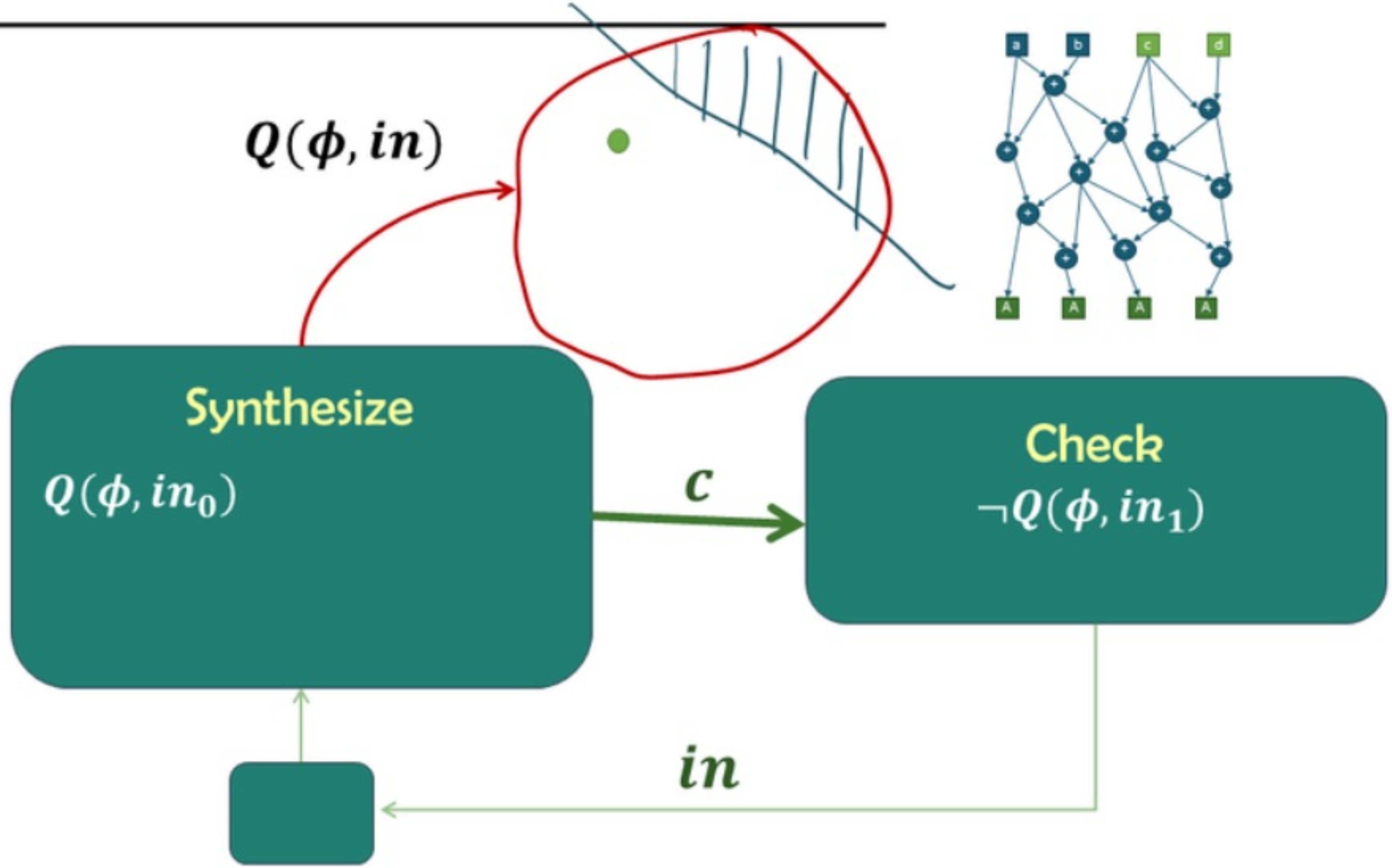
Functional Specification Guided Synthesis



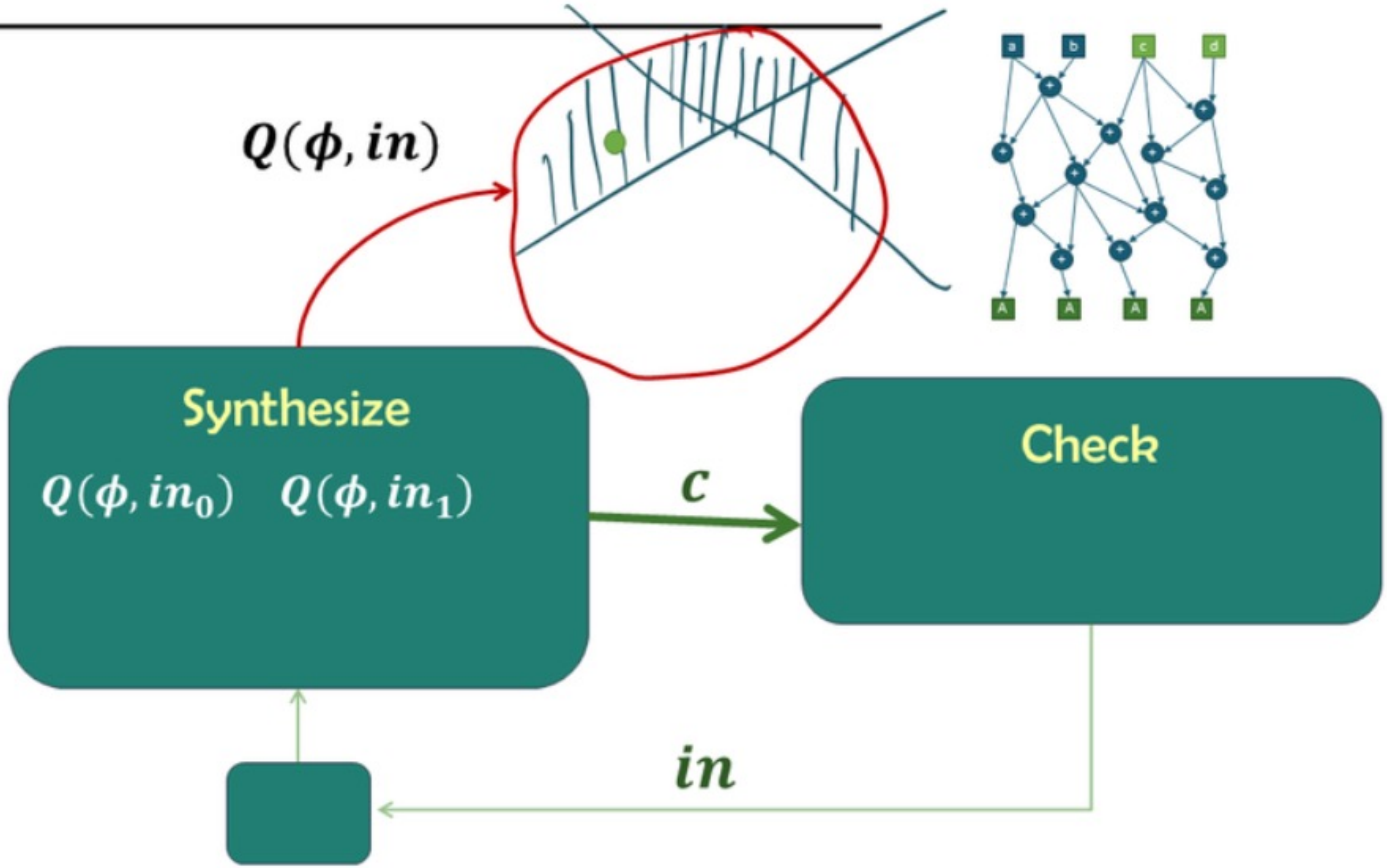
Constraint-based CEGIS



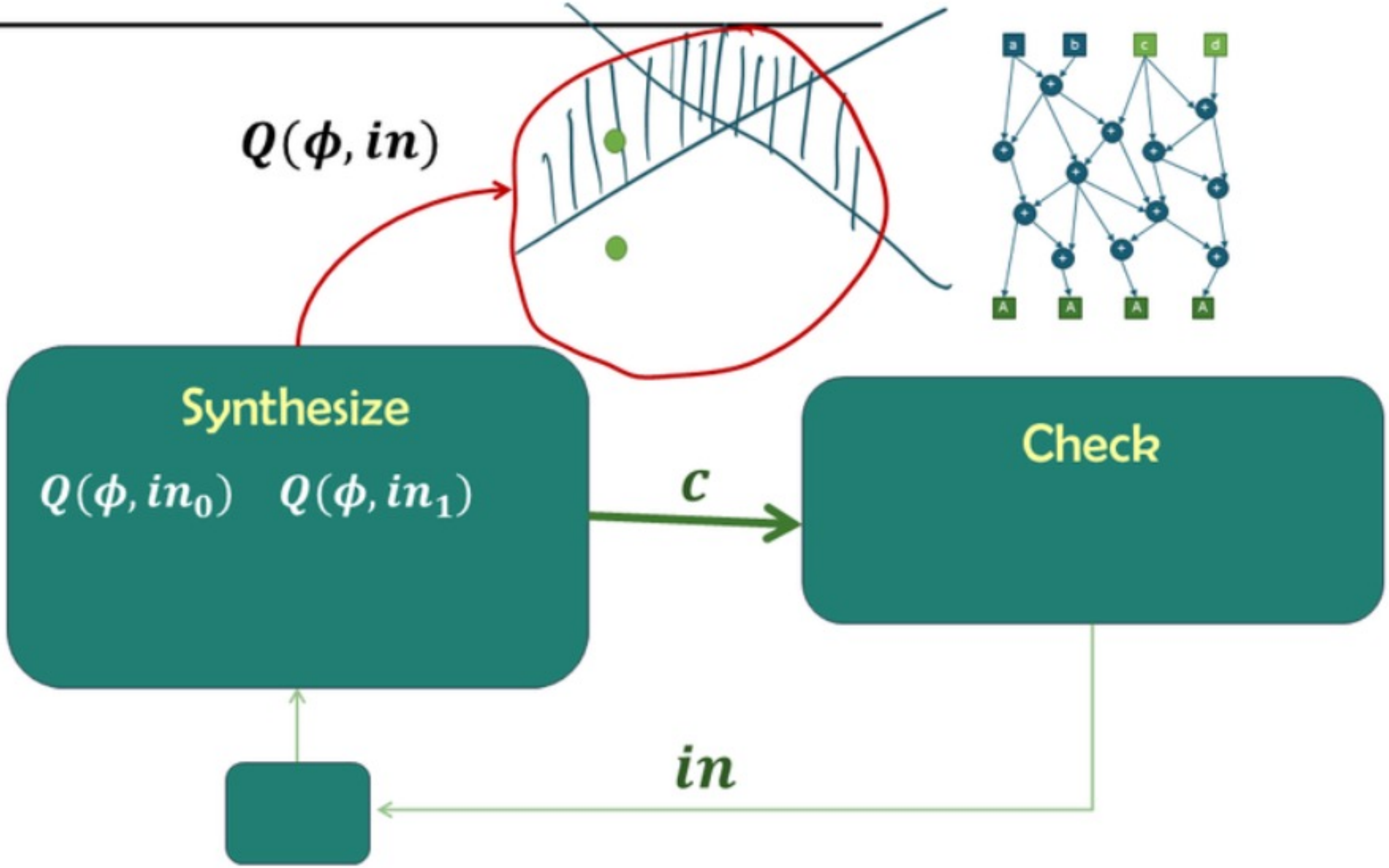
Constraint-based CEGIS



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 {P} c {Q}?
    - Often time, Z3 cannot deal with extremely large programs, and will timeout
    - Who is doing the simplification?
  - Verification <> Synthesis interplay overhead and deficiency?

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

Notes

- Refinement Type as Functional Specification:

```
def abs(x: int) -> int:
  …

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

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