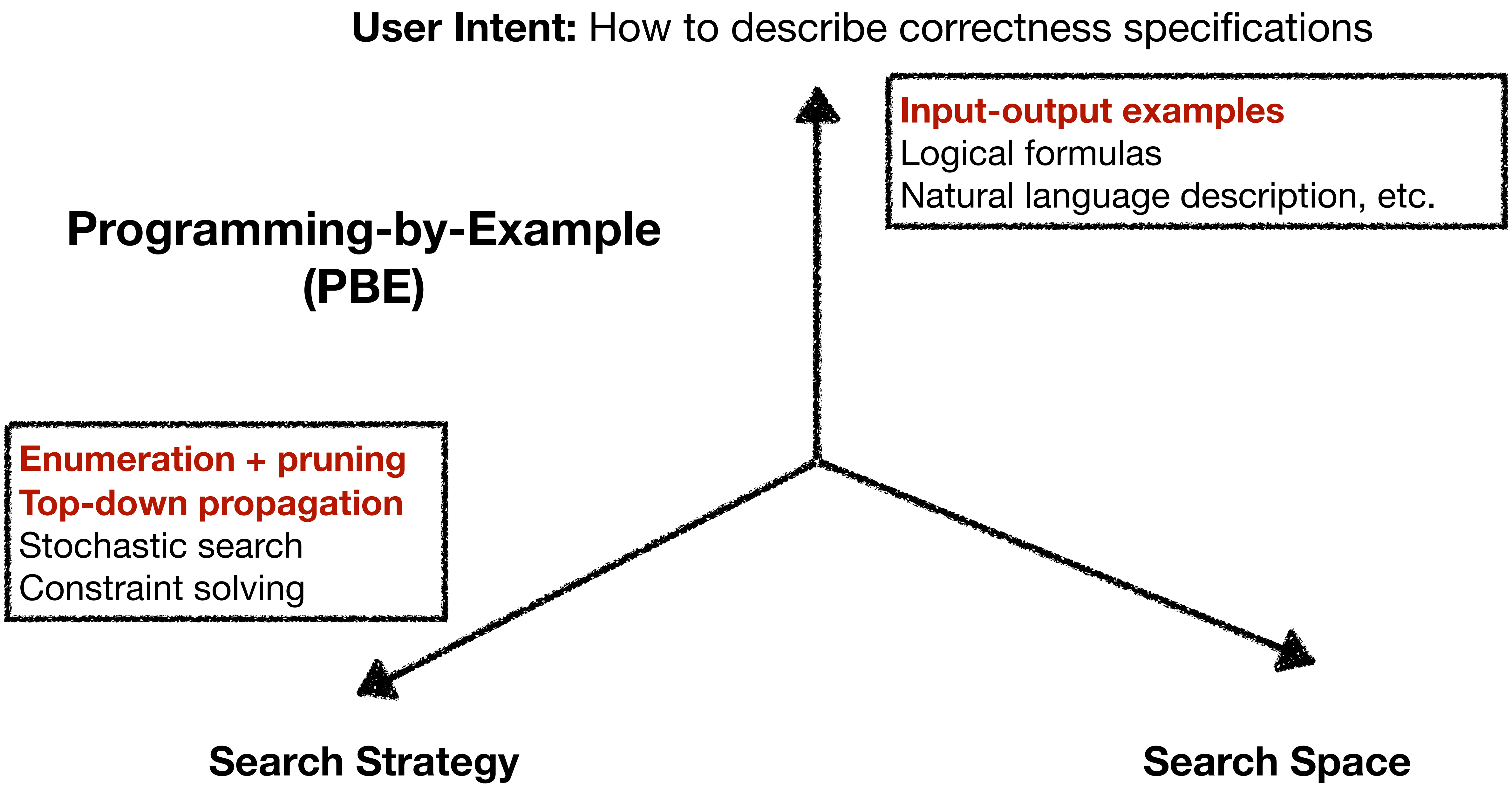
Enumerative Synthesis

Woosuk Lee

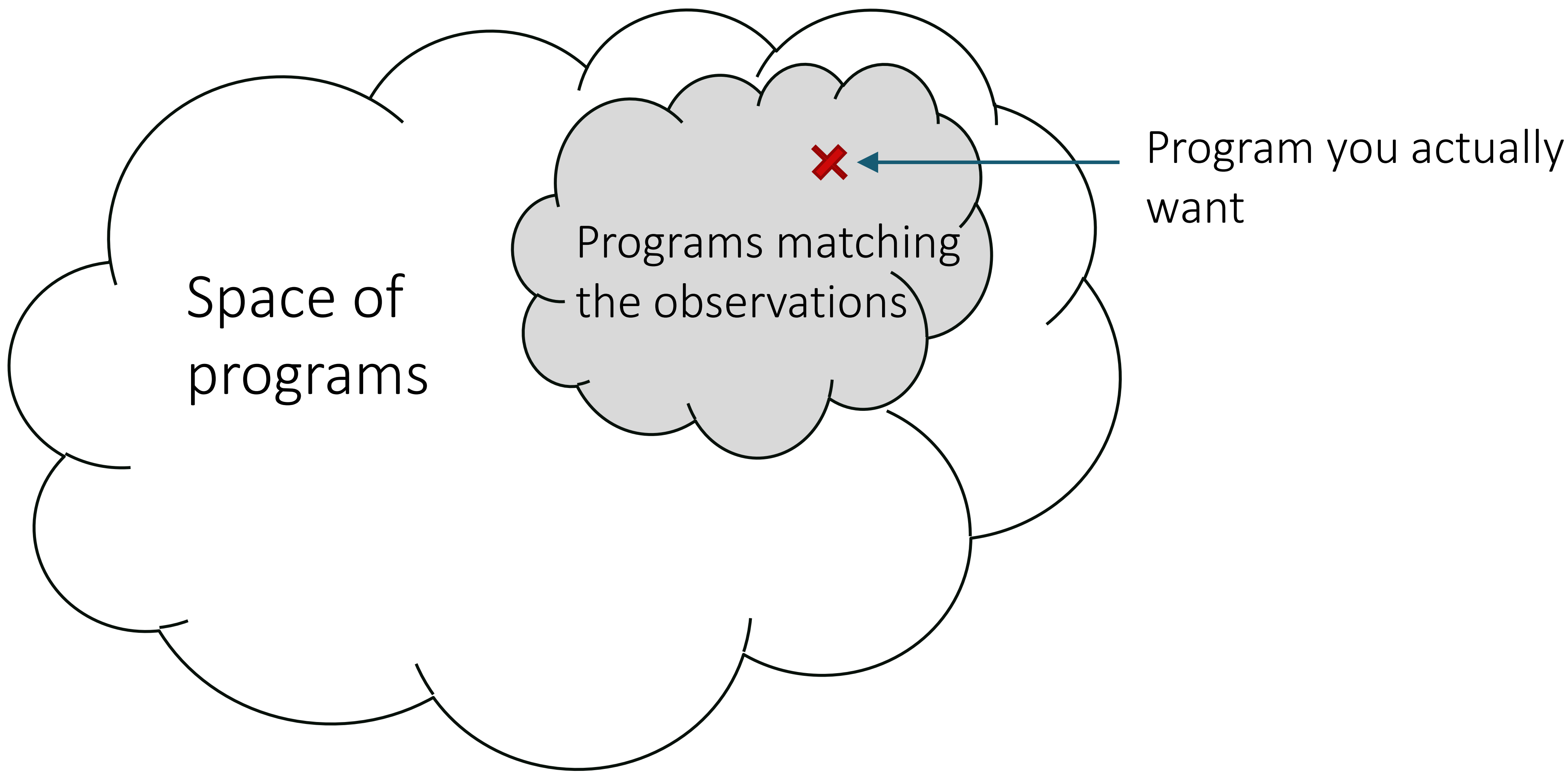
CSE9116 SPRING 2024

Hanyang University

Inductive Synthesis via Enumeration



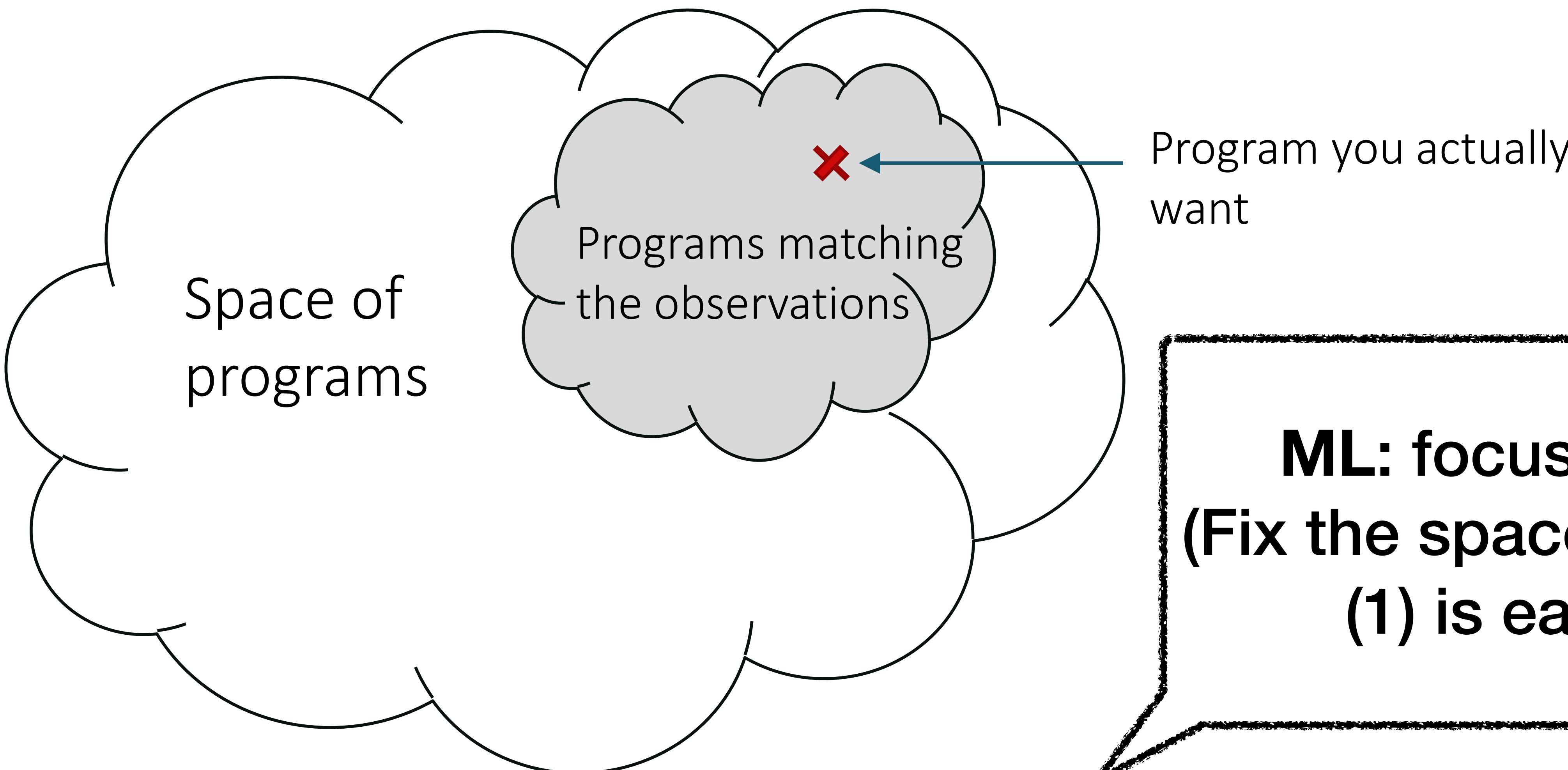
Two Challenges in Inductive Synthesis



1. How do you find a program matching the examples?
2. How do you know it is the program you're looking for? (i.e., avoiding overfitting)

From https://stellar.mit.edu/S/course/6/sp15/6.885/courseMaterial/topics/topic3/lectureNotes/Lecture21/Lecture2.pdf

Two Challenges in Inductive Synthesis

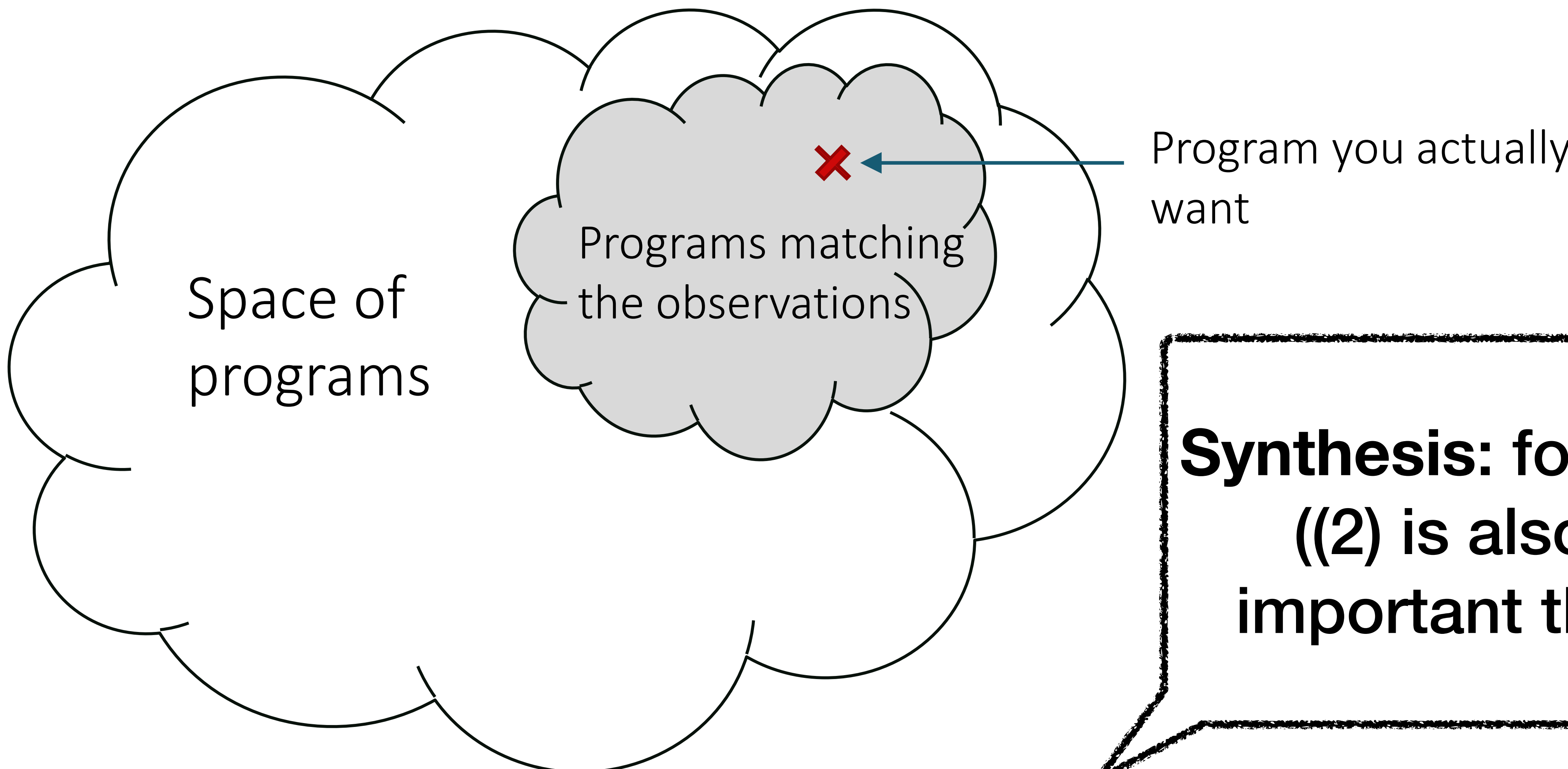


ML: focus on 2 (Fix the space so that (1) is easy)

1. How do you find a program matching the examples?
2. How do you know it is the program you're looking for? (i.e., avoiding overfitting)

From https://stellar.mit.edu/S/course/6/sp15/6.885/courseMaterial/topics/topic3/lectureNotes/Lecture21/Lecture2.pdf

Two Challenges in Inductive Synthesis



Synthesis: focus on 1 ((2) is also still important though)

1. How do you find a program matching the examples?

2. How do you know it is the program you're looking for? (i.e., avoiding overfitting)

From https://stellar.mit.edu/S/course/6/sp15/6.885/courseMaterial/topics/topic3/lectureNotes/Lecture21/Lecture2.pdf

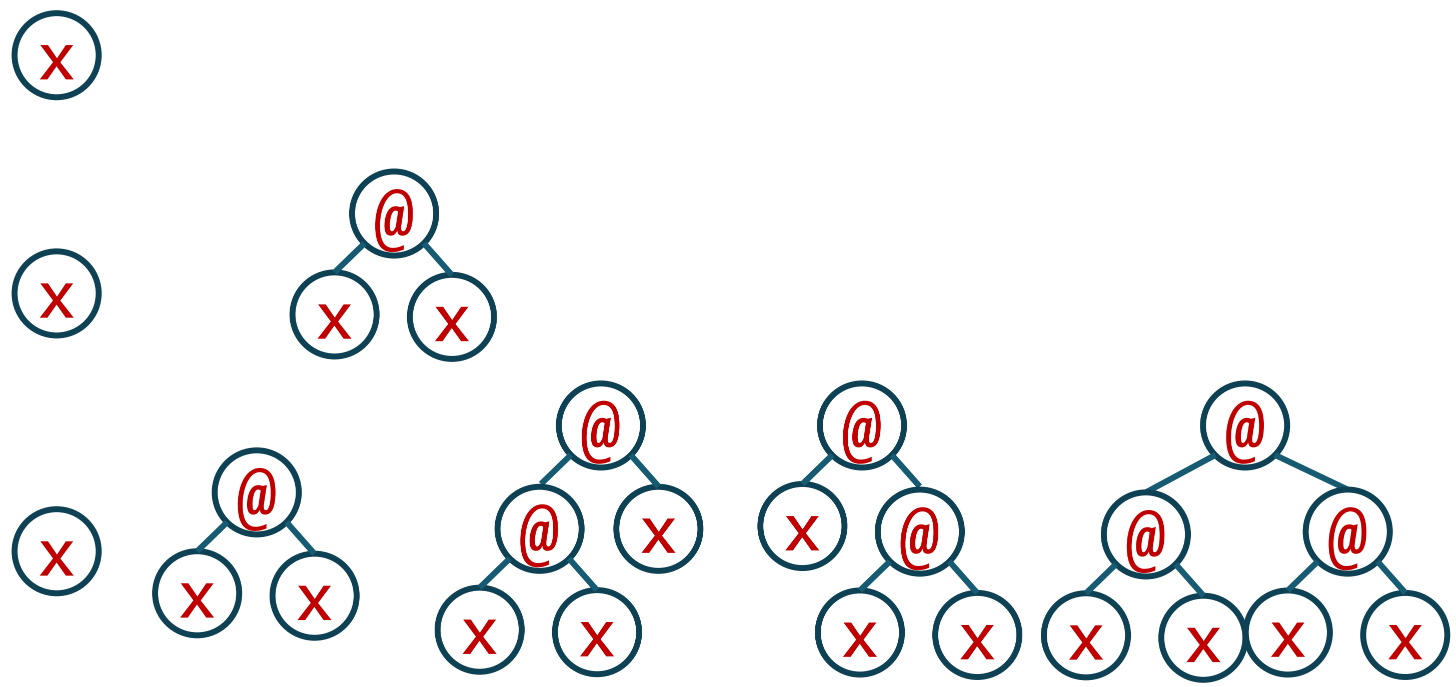
How Large is the Search Space?

E ::= x | E @ E

depth <= 0 — N(0) = 1

depth <= 1 — N(1) = 2

depth <= 2 — N(2) = 5



N(d) = 1 + N(d - 1)^2

From https://github.com/nadia-polikarpova/cse291-program-synthesis

How Large is the Search Space?

```
E ::= x | E @ E
```

$$N(d) = 1 + N(d - 1)^2 \qquad N(d) \sim c^{2^d} \qquad (c > 1)$$

N(1) = 1
N(2) = 2
N(3) = 5
N(4) = 26
N(5) = 677
N(6) = 458330
N(7) = 210066388901
N(8) = 44127887745906175987802
N(9) = 1947270476915296449559703445493848930452791205
N(10) = 3791862310265926082868235028027893277370233152247388584761734150717768254410341175325352026

How Large is the Search Space?

$$E ::= x_1 \mid \ldots \mid x_k \mid E\ @_1\ E \mid \ldots \mid E\ @_m\ E$$

$$N(0) = k$$

$$N(d) = k + m * N(d - 1)^2$$

N(1) = 3
N(2) = 30
N(3) = 2703
N(4) = 21918630
N(5) = 1441279023230703
N(6) = 6231855668414547953818685622630
N(7) = 116508075215851596766492219468227024724121520304443212304350703

k = m = 3

Enumerative Search

- Sample candidates from a given grammar one by one and check if each candidate satisfies the spec
- How to sample?
  - *bottom-up* vs. *top-down*
- Various optimizations

Bottom-Up Enumeration

- Starting from terminal symbols
- Repeatedly composing smaller programs into larger ones
- E.g., Target function $f(x : \texttt{int}) : \texttt{int}$

  Syntactic constraint:

$$S \rightarrow x \mid 1 \mid -S \mid S + S \mid S \times S$$

  Semantic constraint:

$$f(1) = 2$$

Bottom-Up Enumeration

$$x = 1 \because f(1) = 2$$

Size 1	x 1
Size 2	$-x$ -1
Size 3	1×1 $1 \times x$ $x \times 1$ $x \times x$ $x + 1$

Bottom-Up Enumeration

$\boldsymbol{BottomUp}(\text{grammar } G, \text{specification } \phi):$

$P \leftarrow \text{set of all terminals in } G$

$\texttt{progSize} \leftarrow 1$

$\textbf{while} \text{ True } \textbf{do}$

$P \leftarrow \textsc{EnumerateExprs}(G, P, \texttt{progSize})$

$\textbf{foreach } p \in P$

$\textbf{if } \phi(p) \textbf{ then return } p$

$\texttt{progSize} \leftarrow \texttt{progSize} + 1$

Room for Optimization

- Enumerated candidates are complete programs
  - E.g., x + 1
- Thus “executable”
  → can apply “**observational equivalence**” for optimization

Pruning via Observational Equivalence

- Maintains a semantically unique set of expressions
  - i.e., no two expressions are functionally equivalent wrt inputs
- Applicable only if we want a single solution

Udupa et al.,TRANSIT: specifying protocols with concolic snippets. PLDI'13

After Optimization

$x = 1 \because f(1) = 2$

Size 1	x ~~1~~
Size 2	$-x$
Size 3	$x + x$

After Optimization

$x = 1 \because f(1) = 2$

Size 1

x 1

Size 2

$-x$ -1

Size 3

$1+1$ $1+x$ $x+1$

$x+x$

Only representatives of classes of

- programs that output **1**
- programs that output **(-1)**
- programs that output **2**

are explored.

Bottom-Up Enumeration (improved)

$\textbf{\textit{BottomUp}}(\text{grammar } G, \text{specification } \phi):$

$P \leftarrow$ set of all terminals in G

$\texttt{progSize} \leftarrow 1$

while True **do**

$P \leftarrow \textsc{EnumerateExprs}(G, P, \texttt{progSize})$

$P \leftarrow \{p' \in P \mid \forall p \in P.\ \neg\textsc{Equiv}(\phi, p, p')\}$ **Added**

foreach $p \in P$

if $\phi(p)$ **then return** p

$\texttt{progSize} \leftarrow \texttt{progSize} + 1$

Top-Down Enumeration

- Starting from the start non-terminal symbol
- Applying production rules repeatedly

Top-Down Enumeration

$$S \to x \mid 1 \mid -S \mid S + S \mid S \times S$$

Iter 0	S				
Iter 1	x	1	$-S$	$S + S$	$S \times S$
Iter 2	$-x$	-1	$-(-S)$	$x + S \cdots$	
Iter 3	$-(-x)$	$-(-1)$	$x + 1$		

Top-Down Enumeration

Non-terminals — Terminals — Production rules — Starting non-terminal

Grammar $G = \langle N, \Sigma, R, S \rangle$

$\textbf{\textit{TopDown}}(\text{grammar } G, \text{specification } \phi):$

$P \leftarrow \{S\}$

$\texttt{while } P \neq \emptyset:$

$p \leftarrow \mathit{Dequeue}(P)$

$\texttt{if } \phi(p) : \texttt{return } p$

$P' \leftarrow \mathit{Unroll}(G, p)$

$\texttt{forall } p' \in P':$

$P \leftarrow \mathit{Enqueue}(P, p')$

Candidates with fewer non-terminals first

$\textbf{\textit{Unroll}}(\text{grammar } G, \text{program } p)$

$P' \leftarrow \emptyset$

$\texttt{forall } A \in p:$ (Nonterminal)

$\texttt{forall } (A \rightarrow B) \in R:$

$p' \leftarrow p[B/A]$ (Replace A with B in p)

$P' \leftarrow P' \cup \{p'\}$

$\texttt{return } P'$

Optimizations

- Maintaining only representatives of equivalence by
- 1) considering *observationally equivalent sub-expressions*
    - E.g., only maintain "x + *S*" or "1 + *S*" in the queue as x = 1
- 2) breaking *symmetries*
    - E.g., only maintain "x + *S*" or "*S* + x"

      "1 x (*S* + *S*)" or "*S* + *S*"

Lee et al., Accelerating Search-based Program Synthesis using Learned Probabilistic Models, PLDI'18

Smith et al., Program Synthesis with Equivalence Reduction, VMCAI'19

Top-Down Enumeration (improved)

$\textbf{\textit{TopDown}}(\text{grammar } G, \text{specification } \phi):$

$P \leftarrow \{S\}$

$\texttt{while } P \neq \emptyset:$

$p \leftarrow Dequeue(P)$

$\texttt{if } \phi(p): \texttt{return } p$

$P' \leftarrow Unroll(G, p)$

$\texttt{forall } p' \in P':$

$\texttt{if } \neg Subsumed(P, p'):$ **Added**

$P \leftarrow Enqueue(P, p')$

$\textbf{\textit{Unroll}}(\text{grammar } G, \text{program } p)$

$P' \leftarrow \emptyset$

$\texttt{forall } A \in p:$

$\texttt{forall } (A \rightarrow B) \in G:$

$p' \leftarrow p[B/A]$

$P' \leftarrow P' \cup \{p'\}$

$\texttt{return } P'$

Other Optimizations

- Early pruning of hopeless candidates
    - E.g., when spec is $f(2) = 3$, $x \times S$ is not maintained in the queue
- Various deductive methods such as type inference, constraint solving, abstract interpretation are used.

Nadia Polikarpova, Ivan Kuraj, Armando Solar-Lezama: Program synthesis from Polymorphic Refinement Types. PLDI'16

Feng, Martins, Bastani, Dillig: Program synthesis using conflict-driven learning. PLDI'18

Implementation Details (Top-Down)

- A priority queue can be used to prioritize candidates with fewer non-terminals.

- Keeping track of the representatives of equivalence obtained so far can save computation.

---

https://en.wikipedia.org/wiki/Priority_queue

Implementation Details (Bottom-Up)

- The `EnumerateExprs` procedure is typically implemented as a *generator.*
- A generator is a function that returns an array.
  - Instead of returning an array containing all the values at once, it *yields* the values one at a time.
  - Requires less memory

Further Optimization: Divide-and-Conquer

- E.g., Target function: $f(x : \texttt{int}, \texttt{y} : \texttt{int}) : \texttt{int}$

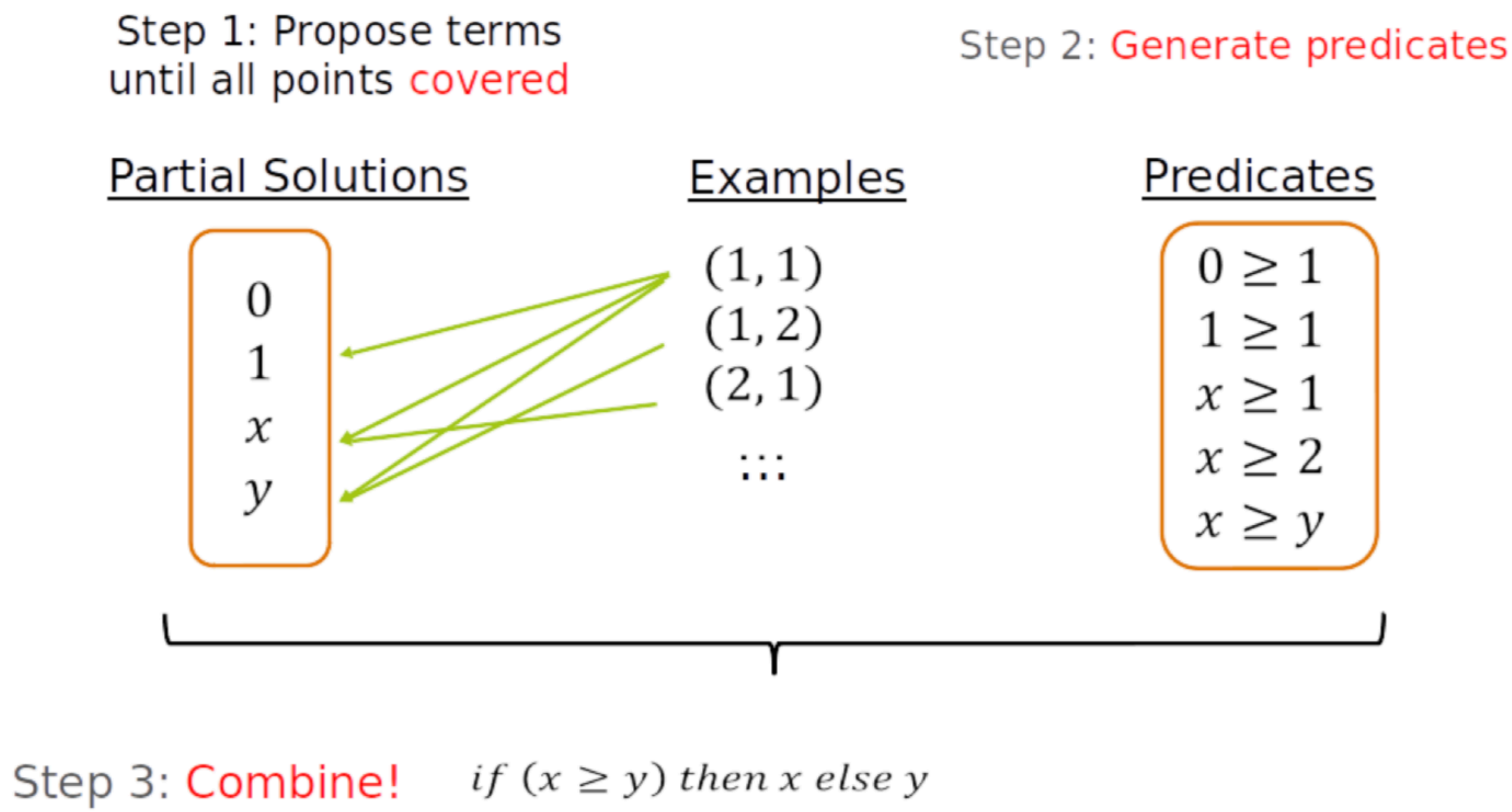
  Syntactic:

$$S \rightarrow x \mid y \mid S + S \mid S - S \mid \texttt{if}\ B\ S\ S \mid 0 \mid 1 \mid 2$$
$$B \rightarrow S \geq S$$

  Semantic: $f(1,1) = 1 \land f(1,2) = 2 \land f(2,1) = 2$

Further Optimization: Divide-and-Conquer

- Find expressions correct wrt *some I/O examples*
- And composing them with conditionals (via Decision tree learning)



Scaling Enumerative Program Synthesis via Divide and Conquer, TACAS'17

Further Optimization: Divide-and-Conquer

Term $S \to x \mid y \mid S + S \mid S - S \mid 0 \mid 1 \mid 2$

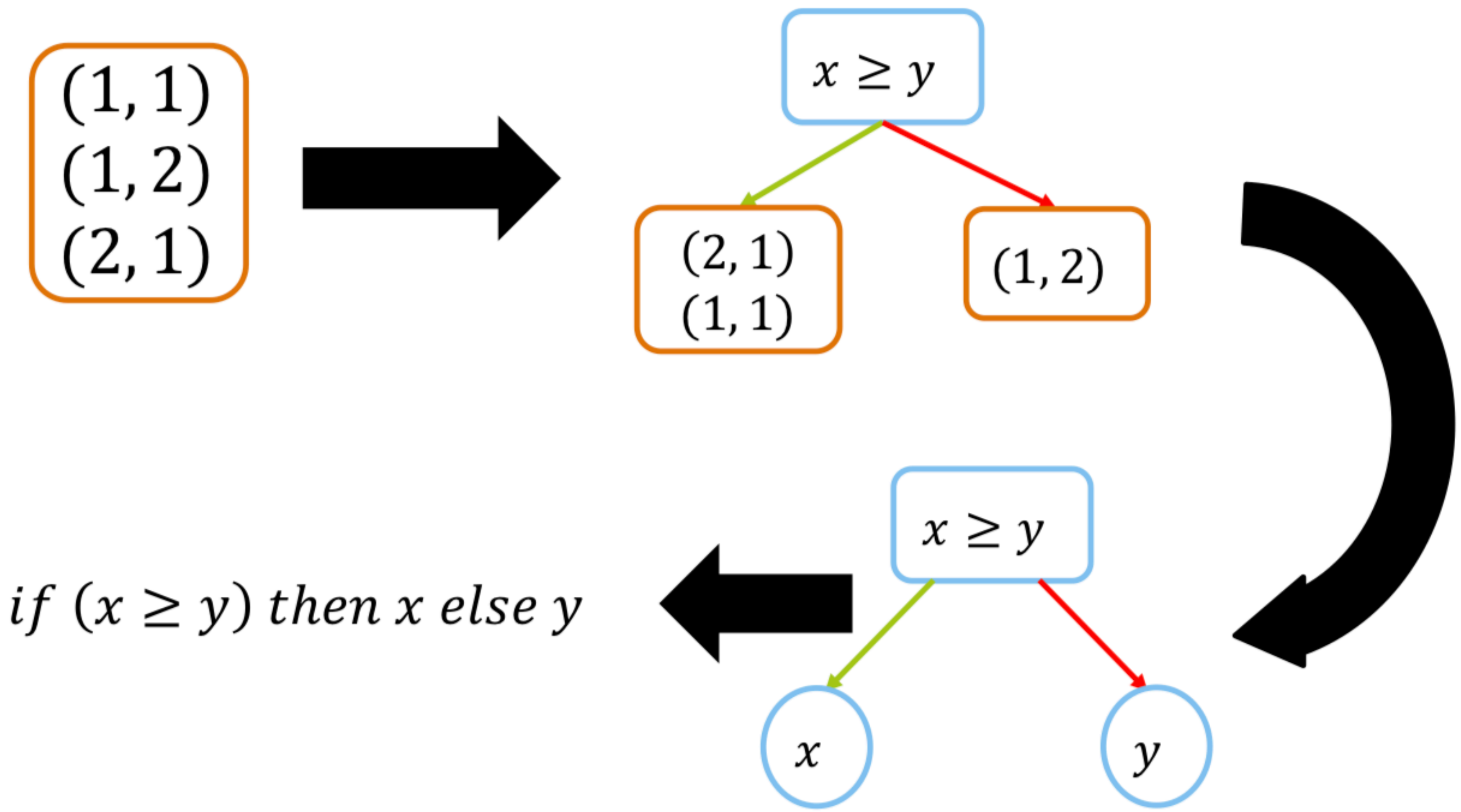
Predicate $B \to S \geq S$

Terms in order of increasing size

0
1
2
x
y

Predicates in order of increasing size

$0 \geq 1$
$1 \geq 1$
$x \geq 1$
$x \geq 2$
$x \geq y$



Divide-and-Conquer Enumeration

Algorithm 2 DCSolve: The divide-and-conquer enumeration algorithm

Require: Conditional expression grammar $G = \langle G_T, G_P \rangle$
Require: Specification Φ
Ensure: Expression e s.t. $e \in [\![G]\!] \wedge e \models \Phi$

$\mathsf{pts} \leftarrow \emptyset$
while `true` **do**
    $\mathsf{terms} \leftarrow \emptyset$; $\mathsf{preds} \leftarrow \emptyset$; $\mathsf{cover} \leftarrow \emptyset$; $DT = \bot$
    while $\bigcup_{t \in \mathsf{terms}} \mathsf{cover}[t] \neq \mathsf{pts}$ **do** ▷ Term solver
        $\mathsf{terms} \leftarrow \mathsf{terms} \cup \textsc{NextDistinctTerm}(\mathsf{pts}, \mathsf{terms}, \mathsf{cover})$
    while $DT = \bot$ **do** ▷ Unifier
        $\mathsf{terms} \leftarrow \mathsf{terms} \cup \textsc{NextDistinctTerm}(\mathsf{pts}, \mathsf{terms}, \mathsf{cover})$
        $\mathsf{preds} \leftarrow \mathsf{preds} \cup \textsc{enumerate}(G_P, \mathsf{pts})$
        $DT \leftarrow \textsc{LearnDT}(\mathsf{terms}, \mathsf{preds})$
    $e \leftarrow \mathsf{expr}(DT)$; $\mathsf{cexpt} \leftarrow \mathsf{verify}(e, \Phi)$ ▷ Verifier
    if $\mathsf{cexpt} = \bot$ **then return** e
    $\mathsf{pts} \leftarrow \mathsf{pts} \cup \mathsf{cexpt}$

Divide-and-Conquer Enumeration

Algorithm 3 Learning Decision Trees

Require: pts, terms, cover, preds

Ensure: Decision tree DT

if $\exists t : \mathsf{pts} \subseteq \mathsf{cover}[t]$ **then return** $\mathit{LeafNode}[\mathcal{L} \leftarrow t]$

if $\mathsf{preds} = \emptyset$ **then return** $\bot$

$p \leftarrow$ Pick predicate from preds

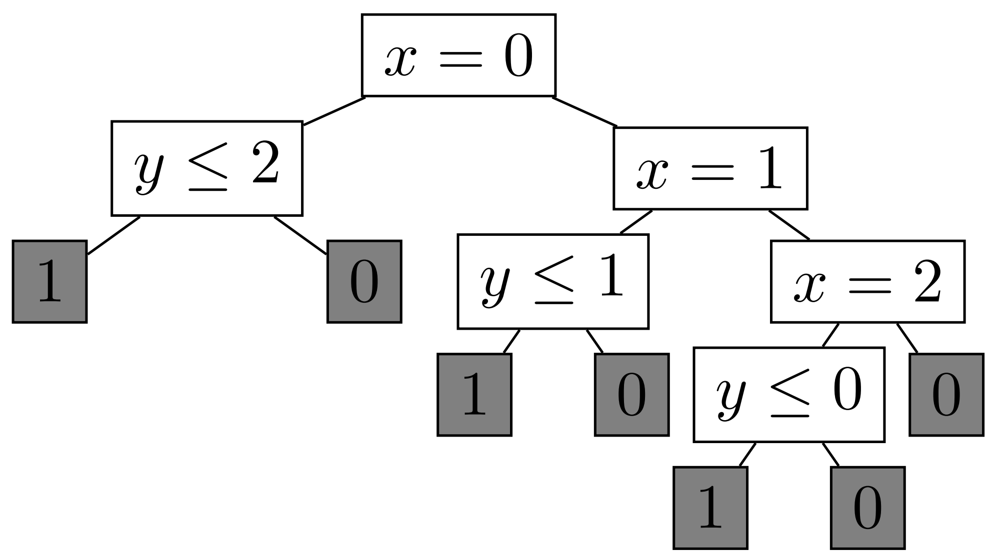
$L \leftarrow \textsc{LearnDT}(\{\mathsf{pt} \mid p[\mathsf{pt}]\}, \mathsf{terms}, \mathsf{cover}, \mathsf{preds} \setminus \{p\})$

$R \leftarrow \textsc{LearnDT}(\{\mathsf{pt} \mid \neg p[\mathsf{pt}]\}, \mathsf{terms}, \mathsf{cover}, \mathsf{preds} \setminus \{p\})$

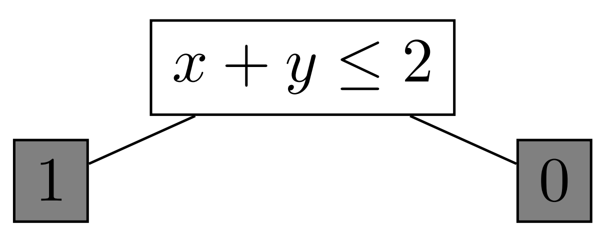
return $\mathit{InternalNode}[\mathcal{A} \leftarrow p, \mathit{left} \leftarrow L, \mathit{right} \leftarrow R]$

Divide-and-Conquer Enumeration

- To synthesize conditional programs as small as possible,
  - The information gain heuristic is used.
  - More predicates are collected.



(a) Decision tree for predicates of size 3



(b) Decision tree for predicates of size 4

Overfitting

- Input-output examples can be an underspecification.
- E.g.,: The max function $f(x : \texttt{int}, \texttt{y} : \texttt{int}) : \texttt{int}$

Syntactic constraint:

$$S \rightarrow x \mid y \mid S + S \mid S - S \mid \texttt{if}\ B\ S\ S$$
$$B \rightarrow S \leq S \mid S = S$$

Semantic constraint: $f(3,1) = 3 \wedge f(1,2) = 2$

Bottom-Up Enumeration

Size 1	B ↦ { } S ↦ { x y }
Size 2	
Size 3	B ↦ { x ≤ x, x ≤ y, y ≤ x, y ≤ y, … } S ↦ { x + x, x + y, y + x,…, x * y }

Not the desired solution!

Counter-example Guided Inductive Synthesis (CEGIS)

- Enables inductive synthesis strategies beyond I/O examples
- E.g., The max function: $f(x : \texttt{int}, \texttt{y} : \texttt{int}) : \texttt{int}$

Syntactic constraint:

$$S \to x \mid y \mid S + S \mid S - S \mid \texttt{if}\ B\ S\ S$$
$$B \to S \leq S \mid S = S$$

Semantic constraint: ~~$f(3, 1) = 3 \wedge f(1, 2) = 2$~~

$$\forall x, y.\ f(x, y) \geq x \wedge f(x, y) \geq y \wedge (f(x, y) = x \vee f(x, y) = y)$$

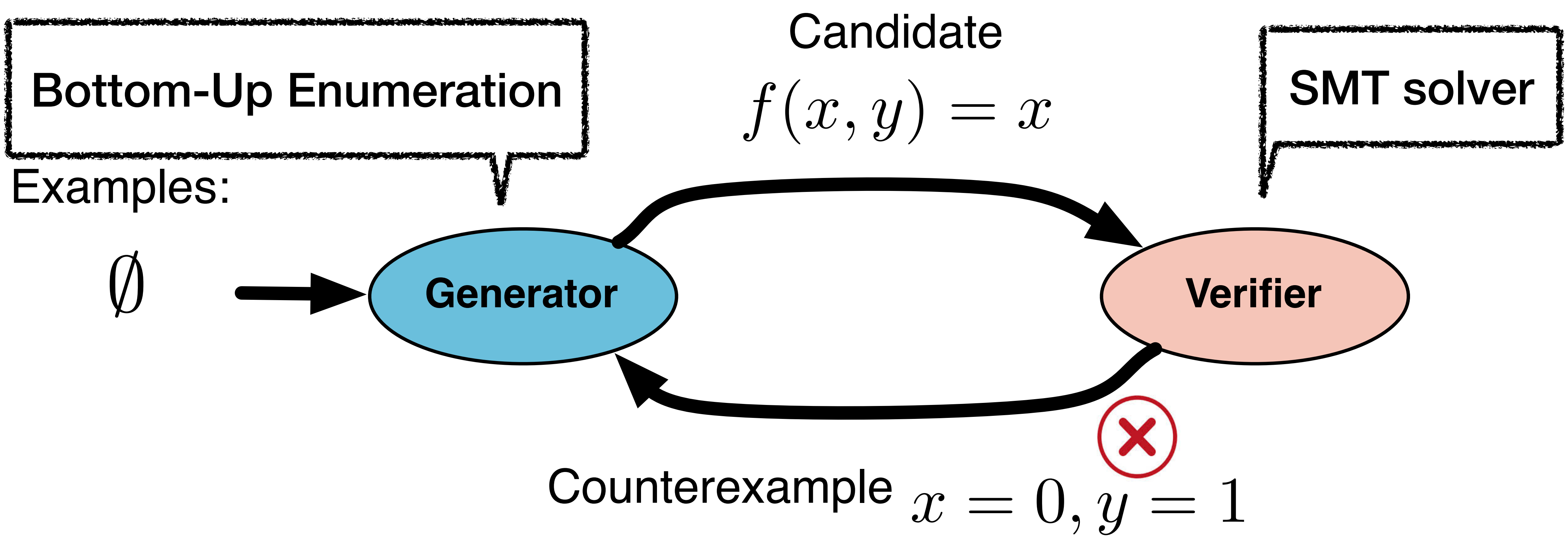
Counter-example Guided Inductive Synthesis (CEGIS)

- Makes inductive synthesis strategies applicable for beyond I/O examples

- *Generator* proposes candidates.

- *Verifier* checks correctness for each proposed candidate.

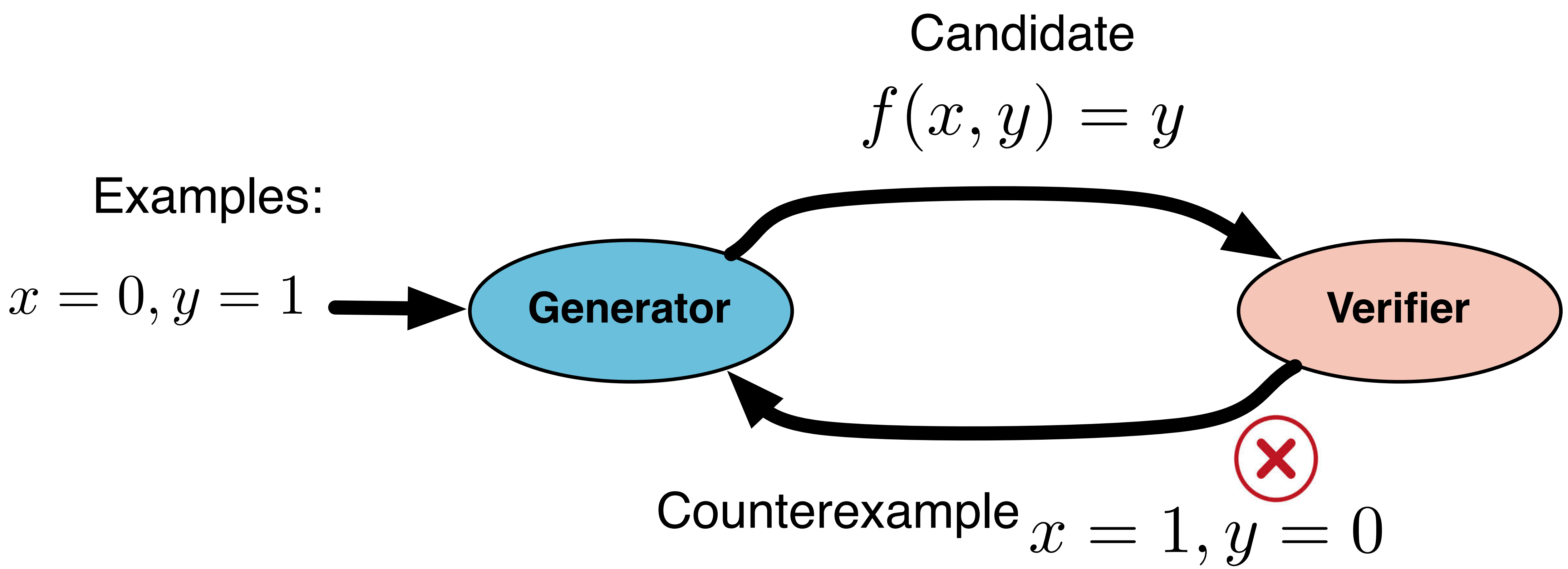
CEGIS + Bottom-Up Search



$$\because \quad f(0, 1) = 1$$

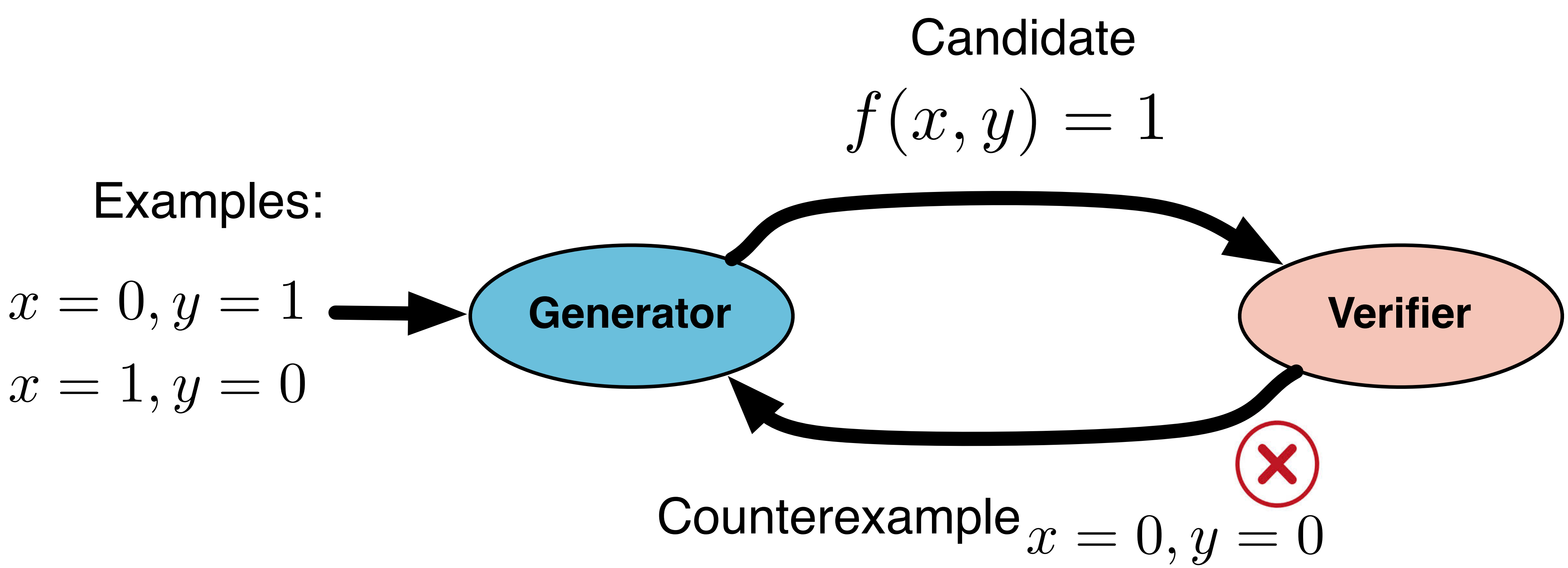
$$\neg \forall x, y.\ x \geq x \wedge x \geq y \wedge (x = x \vee x = y)$$

CEGIS + Bottom-Up Search



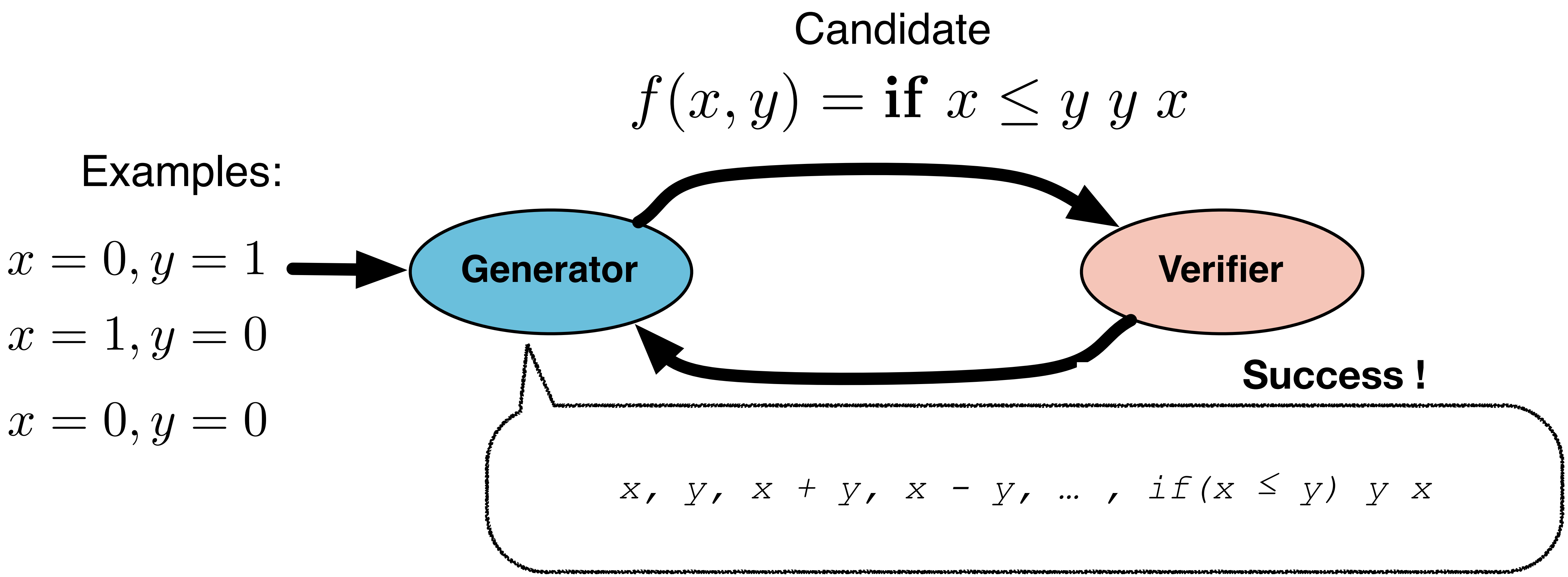
$$\because \quad f(1,0) = 1$$

CEGIS + Bottom-Up Search



$$\because \; f(0, 0) = 0$$

CEGIS + Bottom-Up Search



Benefits of CEGIS

- Generator and verifier are independent to each other.
- #. of CEGIS iteration is often small in practice.
  - i.e., candidate correct wrt a few examples is often a solution
  - Programmers often aspire to write programs correct wrt a few corner cases
  - which gives performance benefits for various generators
    - e.g., constraint solving-based synthesizers — may handle smaller constraints
    - e.g., enumeration-based synthesizer — can enjoy better optimization impacts such as observational equivalence

Limitations of Enumerative Synthesis

- Pros
  - Generally applicable to almost any kinds of specs
  - Easy to implement
- Cons
  - Limited scalability: cannot synthesize large programs!