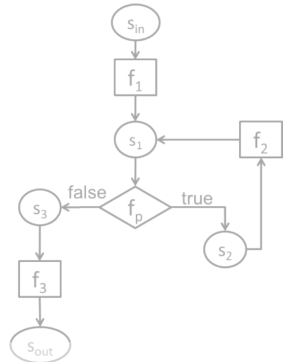
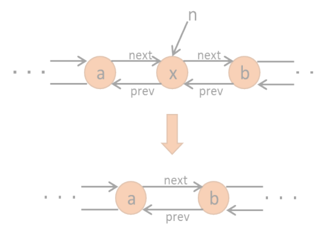
$$\exists c \forall in\; Q(c, in)$$

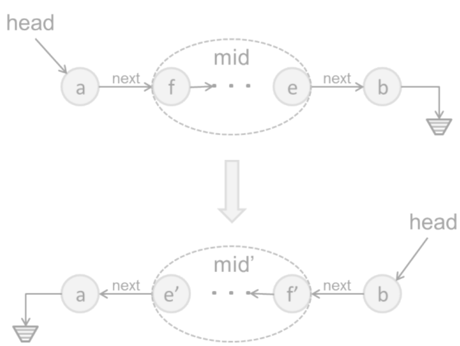
```
/* Average of x and y without using x+y (avoid overflow)*/
int avg(int x, int y){
  int t = expr({x/2, y/2, x%2, y%2, 2 }, {PLUS, DIV});
  assert t == (x+y)/2;
  return t;
}
```

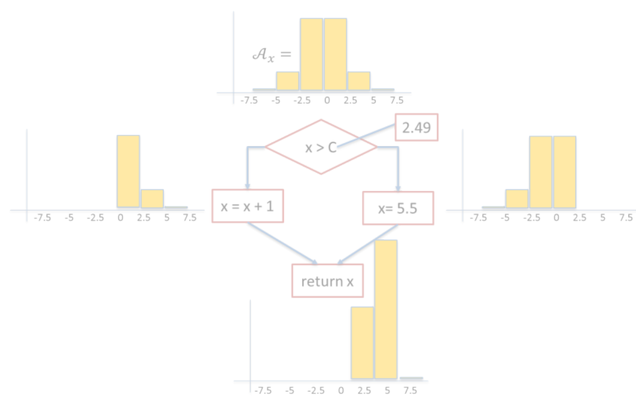





```
{
  s  = n.succ;
  p = n.pred;
  p.succ = s;
  s.pred = p;
}
```

Module II: Searching for Complex Programs



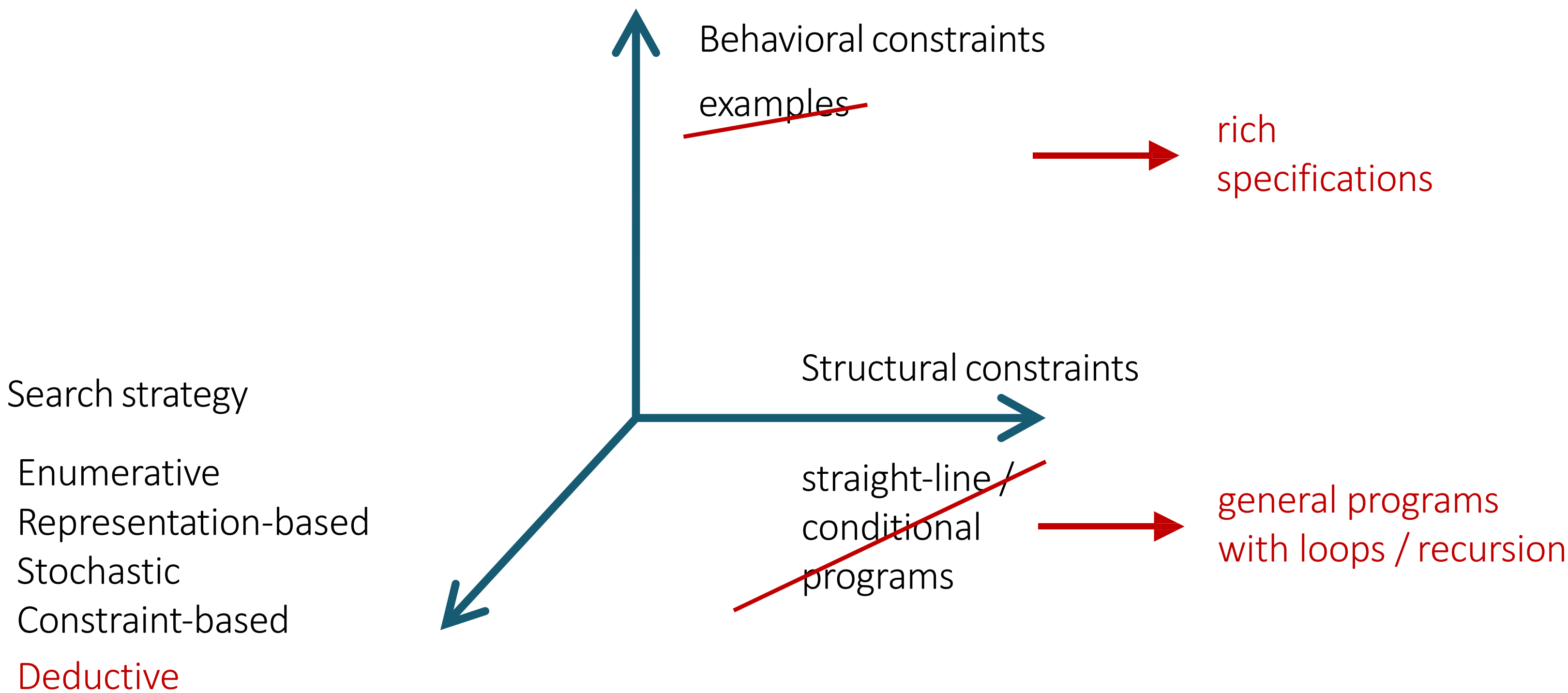


$$\varphi(p)$$

$$Sk[c](in)$$

Lecture 9
Specifications and Reduction to Inductive Synthesis

Module I vs Module II



Examples of rich specifications

Reference implementation

Assertions

Pre- and post-condition

Refinement type

Reference Implementation

Easy to compute the result, but hard to compute it efficiently or under structural constraints

```
bit[W] AES_round (bit[W] in, bit[W] rkey)
{
  ... // Transcribe NIST standard
}
bit[W] AES_round _sk (bit[W] in, bit[W] rkey) implements AES_round
{
  ... // Sketch for table lookup
}
```

Assertions

Hard to compute the result, but easy to check its desired properties

```
split_seconds (int totsec) {
  int h := ??;
  int m := ??;
  int s := ??;
  assert totsec == h*3600 + m*60 + s;
  assert 0 <= h && 0 <= m < 60 && 0 <= s < 60;
}
```

Pre-/post-conditions

Hard to compute the result but easy to express its properties in logic

```
sort (int[] in, int n) returns (int[] out)
  requires
  ensures
{
  ??
}
```

requires $n \geq 0$

ensures $\forall i\, j.\, 0 \leq i < j < n \Rightarrow out[i] \leq out[j]$

$\forall i\,.\, 0 \leq i < n \Rightarrow \exists j.\, 0 \leq j < n \wedge in\,[i] = out[j]$

Refinement types

Same as pre-/post-conditions but logic goes inside the types

binary search tree

red nodes have black children

```
data RBT a where
  Empty :: RBT a
  Node  :: x: a ->
    black: Bool ->
    left:  { RBT {a | _v < x} | !black ==> isBlack _v } ->
    right: { RBT {a | x < _v} | (!black ==> isBlack _v) &&
                (blackHeight _v == blackHeight left)} ->
    RBT a

insert :: x: a -> t: RBT a -> {RBT a | elems _v == elems t + [x]}
insert = ??
```

same number of black nodes on every path to leaves

Why go beyond examples?

Might need too many

- Example: Myth needs 12 for `insert_sorted`, 24 for `list_n_th`
- Examples contain *too little* information
- Successful tools use domain-specific ranking

Output difficult to construct

- Example: AES cypher, RBT
- Examples also contain *too much* information (concrete outputs)

Need strong guarantees

- Example: AES cypher

Reasoning about non-functional properties

- Example: security protocols

Why is this hard?

```
gcd (int a, int b) returns (int c)
  requires a > 0 ∧ b > 0
  ensures  a % c = 0 ∧ b % c = 0
           ∀d . c < d  ⇒  a % d ≠ 0  ∨  b % d ≠ 0
{
  int x , y := a, b;
  while (x != y) {
    if (x > y) x := ??;
    else y := ??;
}}
```

infinitely many inputs

cannot validate by testing

infinitely many paths!

hard to generate constraints

Map of the module

Constraint-based synthesis

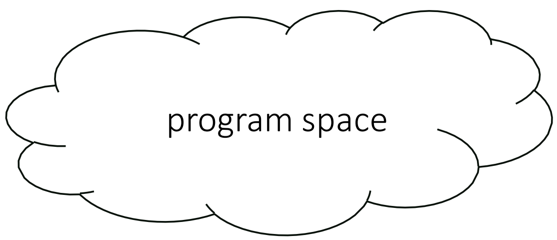
- How to solve constraints about infinitely many inputs? CEGIS
- How to encode semantics of looping / recursive programs?
  - Bounded reasoning
  - Unbounded reasoning

Enumerative and deductive synthesis

- How to use type systems to guide the search?
- How to use program logics to guide the search?

Constraint-based synthesis from specifications

CBS from examples



`E = [i → o]`

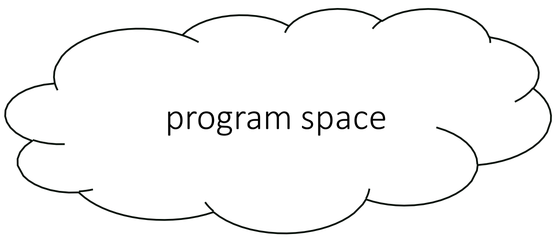
encoding

Parameters (control vector)

"*C* denotes a program that gives output *o* on input *i*"

$$\exists C \,.\, \bigwedge_{\langle i,o\rangle \in E} \varphi(C, i, o)$$

CBS from specifications



~~E = [i → o]~~

$\forall\, i\; o\,.\,\psi(i,o)$

$\xrightarrow{\text{encoding}}$

"*C* denotes a program that gives output *o* on input *i*"

$$\cancel{\exists C\,.\ \bigwedge_{\langle i,o\rangle \in E} \varphi(C,i,o)}$$

$$\boxed{\exists C\,.\,\forall i\; o}\,.\boxed{\varphi(C,i,o)} \Rightarrow \psi(i,o)$$

doubly-quantified constraint: not solver-friendly

hard to define for programs with loops / recursion

Example

$$\exists C\,.\,\forall i\ o\,.\,\varphi(C, i, o) \Rightarrow \psi(i, o)$$

```
harness void main(int x) {
  int y := ?? * x + ??;
  assert y - 1 == x + x;
}
```

encoding

$$\exists c_1 c_2\,.\,\forall x\ y\,.\,y = c_1 * x + c_2$$
$$\Rightarrow y - 1 = x + x$$

simplify

$$\exists c_1 c_2\,.\,\forall x\,.\,c_1 * x + c_2 - 1 = x + x$$

How do we solve this constraint?

CEGIS

$$\exists c \,.\, \forall x \,.\, Q(c, x)$$

Idea 1: Bounded Observation Hypothesis

- Assume there exists a small set of inputs $\mathrm{X} = \{x_1, x_2, \dots x_n\}$ such that whenever c satisfies

$$\bigwedge_{i \in 1..n} Q(c, x_i)$$

No quantifiers here, can give to SAT / SMT

it also satisfies

$$\forall x.\, Q(c, x)$$

Example

This is a linear constraint, two inputs are enough!

$$\exists c_1 c_2 . \forall x . c_1 * x + c_2 - 1 = x + x$$

$X = \{0, 1\}$

$$Q(c_1, c_2, 0) \equiv c_2 - 1 = 0$$
$$Q(c_1, c_2, 1) \equiv c_1 + c_2 - 1 = 2$$

$$\{c_1 \rightarrow 2, c_2 \rightarrow 1\}$$

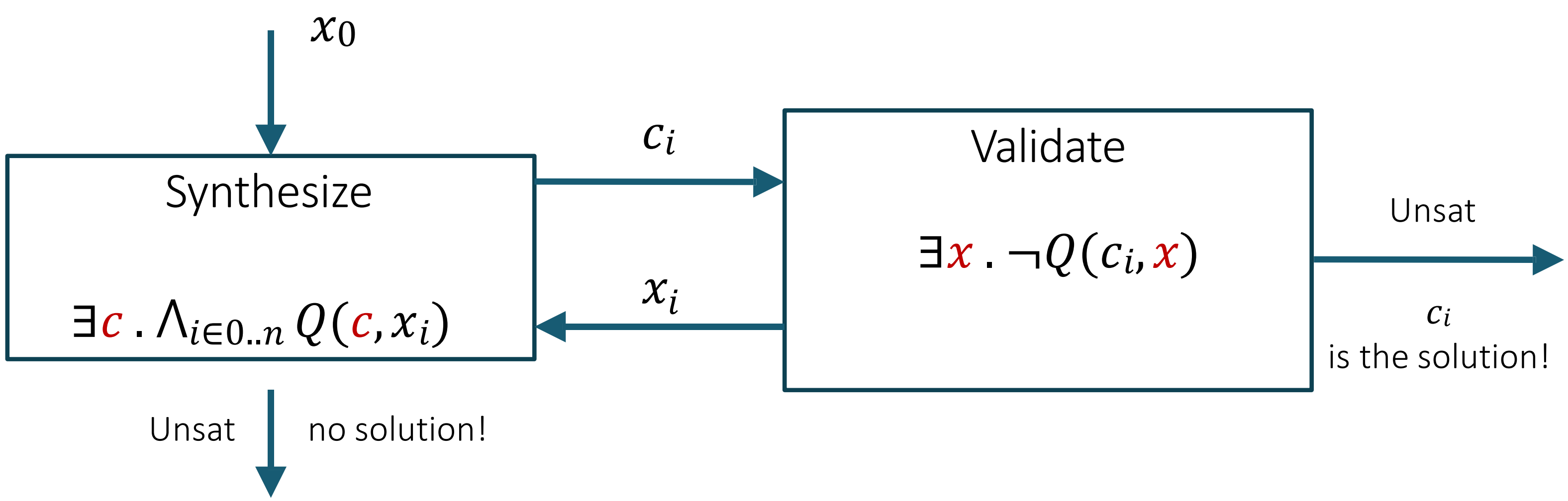
```
harness void main(int x) {
  int y := 2 * x + 1;
  assert y - 1 == x + x;
}
```

How do we find X in a general case?

CEGIS

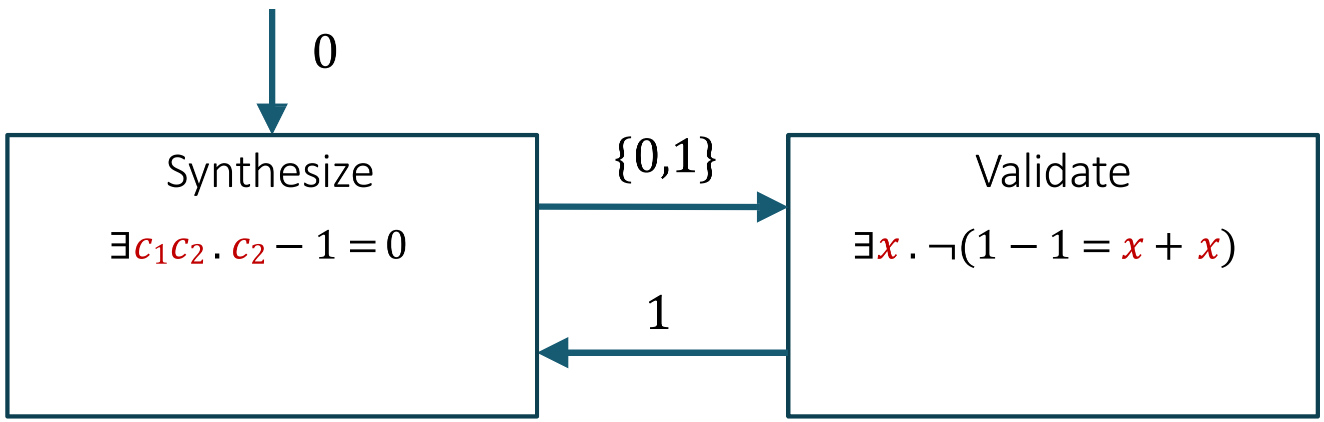
$$\exists c\,.\,\forall x\,.\,Q(c,x)$$

Idea 2: Rely on a validation oracle to generate counterexamples



Example

$$\exists c_1 c_2 \,.\, \forall x \,.\, c_1 * x + c_2 - 1 = x + x$$



Example

$$\exists c_1 c_2 \,.\, \forall x \,.\, c_1 * x + c_2 - 1 = x + x$$