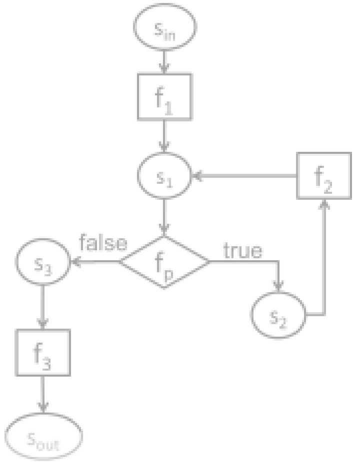
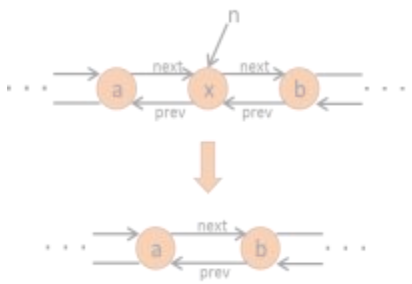
$$\exists c \forall in\ Q(c, in)$$

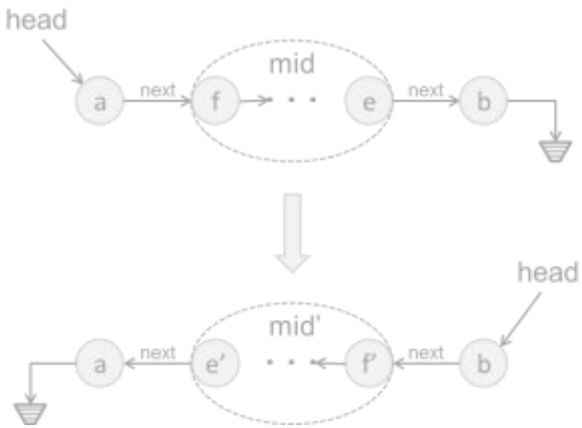
```
/* Average of x and y without using x+y (avoid overflow)*/
int avg(int x, int y){
  int t = expr({x/2, y/2, x%2, y%2, 2 }, {PLUS, DIV});
  assert t == (x+y)/2;
  return t;
}
```

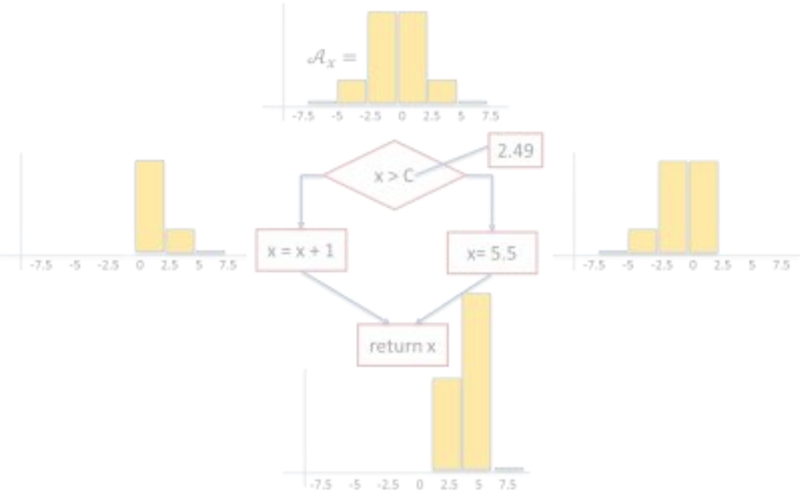





```
{
  s = n.succ;
  p = n.pred;
  p.succ = s;
  s.pred = p;
}
```

Module II: Synthesizing Complex Programs



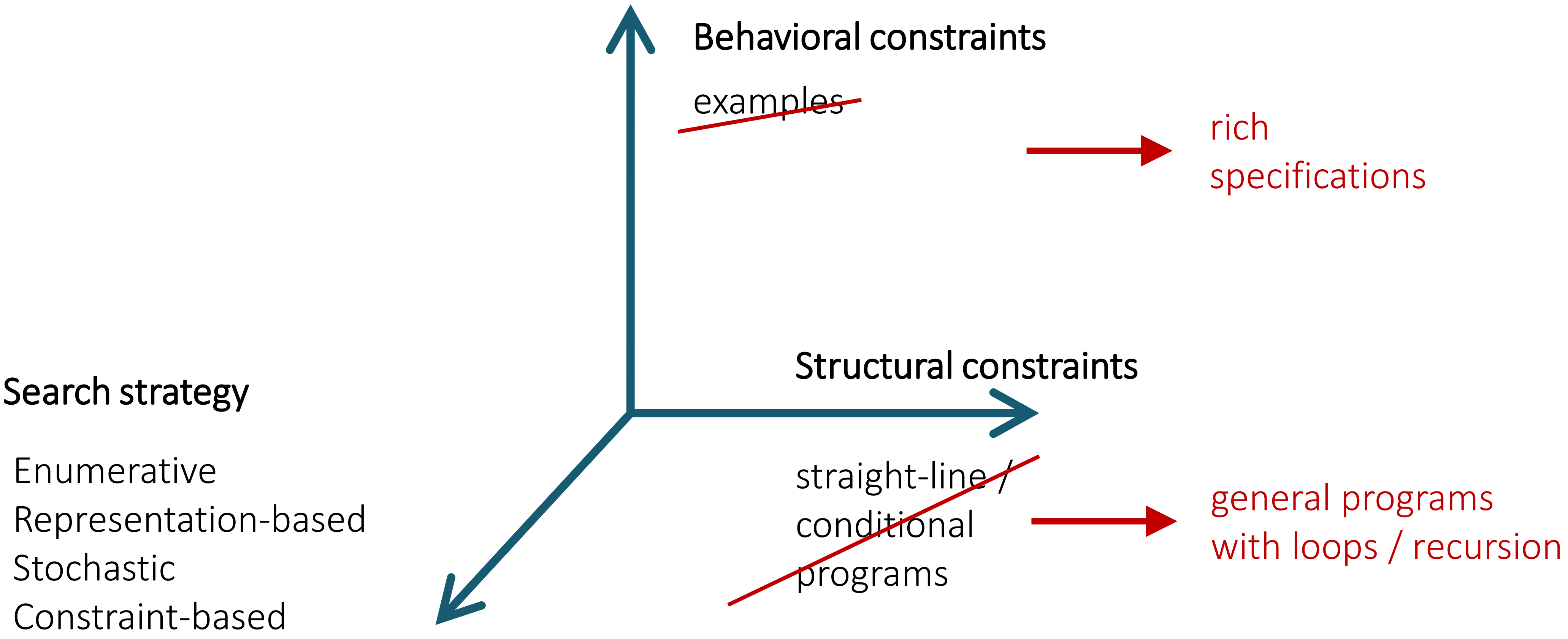


$$\varphi(p)$$

$$Sk[c](in)$$

Lecture 9
Specifications

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  n ≥ 0
  ensures   ∀i j. 0 ≤ i < j < n ⇒ out[i] ≤ out[j]
            ∀i . 0 ≤ i < n ⇒ ∃j. 0 ≤ j < n ∧ 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!

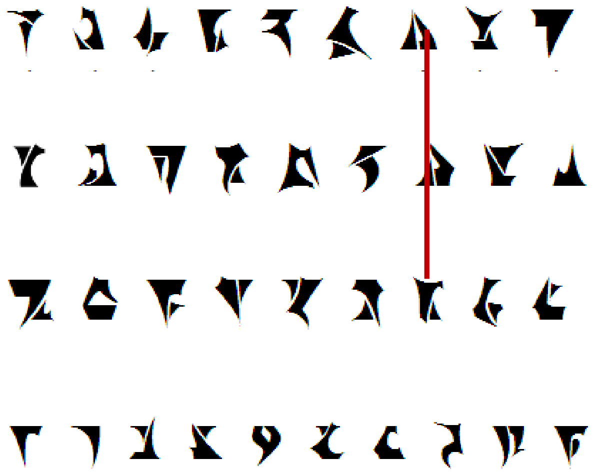
hard to generate constraints

Why is this hard?

Synthesis from examples

validation was easy!

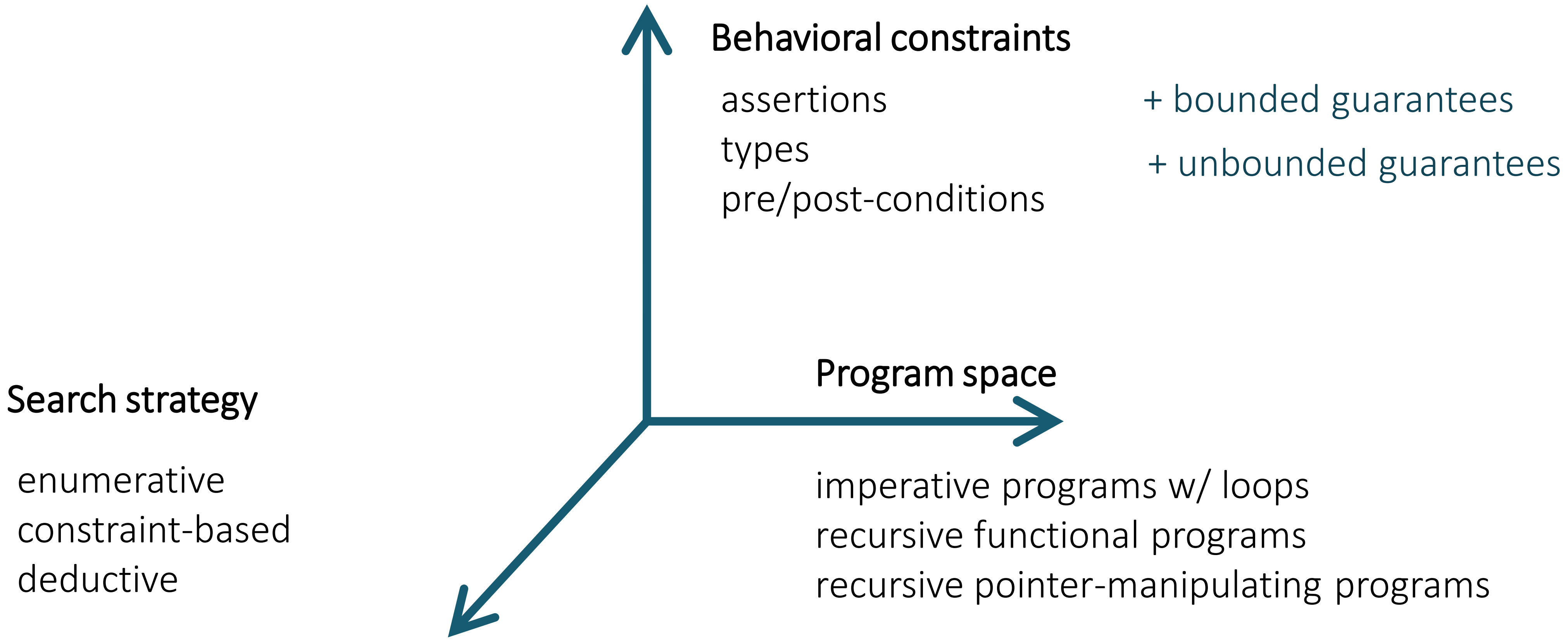
Synthesis from specifications



SEE IF YOU CAN FIND ANY KLINGON FRUIT!

validation is hard!
(and search is still hard)

Module II


Behavioral constraints
assertions
types
pre/post-conditions
+ bounded guarantees
+ unbounded guarantees
Search strategy
enumerative
constraint-based
deductive
Program space
imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs