CS5733 Program Synthesis

#12.Sketching and constraints based search

Ashish Mishra, September 17, 2024

MCMC Based synthesis

Approach:

- Let χ be the space of programs
- Engineer a $K(x, y)$ such that $\pi(x)$ is high for “good programs” and low for “bad programs”
- Pick a random start state x_0
- Simulate the markov process for n steps for some large n.
- By the fundamental theorem, the probability of x_n is a good program will be higher than the probability that it is a bad program

Key step: Engineer K that has desired property for $\pi(x)$

Metropolis algorithm with symmetric Proposal distribution

- Start with a markov matrix $J(x, y)$ with $J\left(x, y\right) > 0 \leftrightarrow J\left(y, x\right) > 0$ and J(x, y) = J(y, x)
- Initialization: Chose an arbitrary x to be the first observation in the sample and initialize J to satisfy the above property.
- For each iteration say t.
  - Propose a candidate y for the next sample by picking from J(x_t, y).
  - Calculate the acceptance ratio A = $\pi(y)/\pi(x_t)$, which is used to decide whether to accept or reject the candidate.
  - Generate a uniform random number u $\in$ [0,1].
  - If u <= A then accept y and set x_{t+1} <- y
  - If u > A then reject the candidate y and set x_{t+1} <- x

Metropolis algorithm : Non symmetric case

- Start with a markov matrix $J(x, y)$ with $J\left(x, y\right) > 0 \leftrightarrow J\left(y, x\right) > 0$
- For each iteration say t.
  - Propose a candidate y for the next sample by picking from J(x_t, y).
  - Calculate the acceptance ratio A = $\frac{\pi(y)}{J(x_t, y)} / \frac{\pi(x_t)}{J(y, x_t)}$, which is used to decide whether to accept or reject the candidate.
  - If A >= 1 then accept y and set x_{t+1} <- y
  - If 0 < A < 1 then
    - accept candidate y and set x_{t+1} <- y with probability A
    - reject canditate y and set x_{t+1} <- x with probability (1- A)

How do we prove that K (x, y) gives a stationary distribution \pi

Detailed Balance Equation holds in the above construction:

$$\pi(x)K\left(x, y\right) = \pi\left(y\right)K(y, x)$$

Probability to be at a position x

and move to a position y

=

Probability to be at a position y

and move to a position x

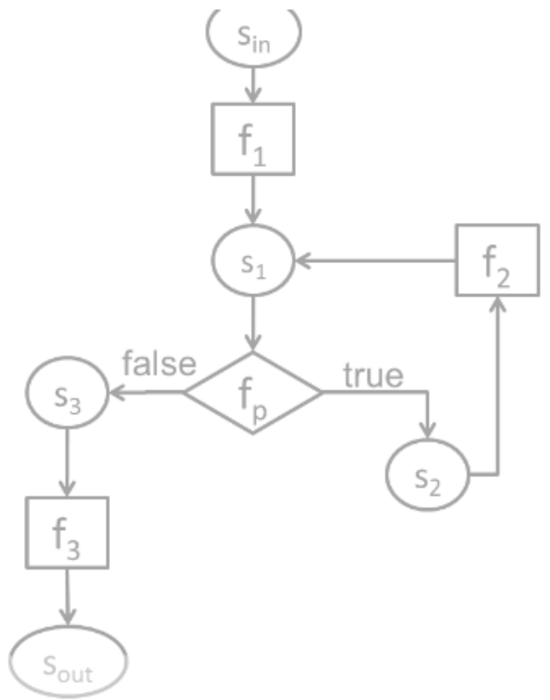
For any position our Markov chain can visit, there is as much in-flow as out-flow

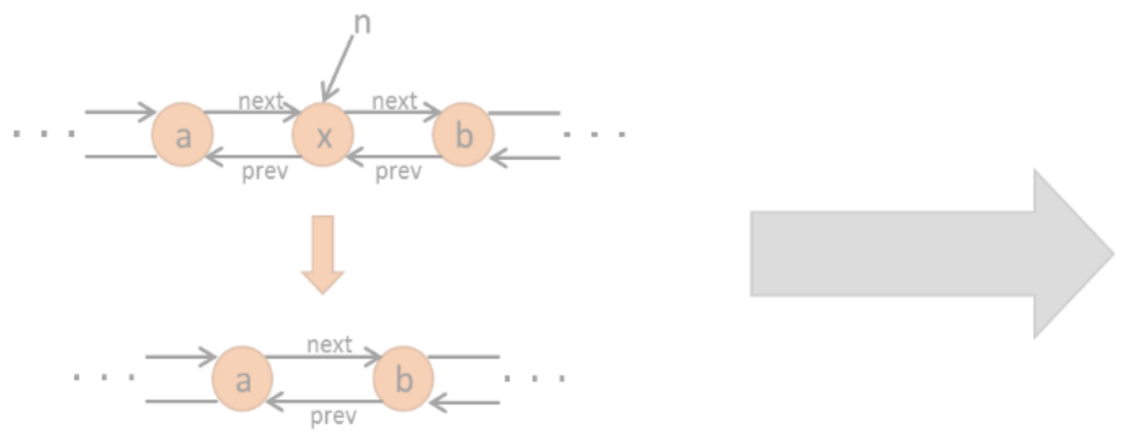
And thus the K(x, y) can no longer change, thus the calculate K is a unique stationary distribution

$$\sum_x \pi(x)K\left(x, y\right) = \sum_x \pi\left(y\right)K\left(y, x\right) = \pi\left(y\right)\sum_x K\left(y, x\right) = \pi(y)$$

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

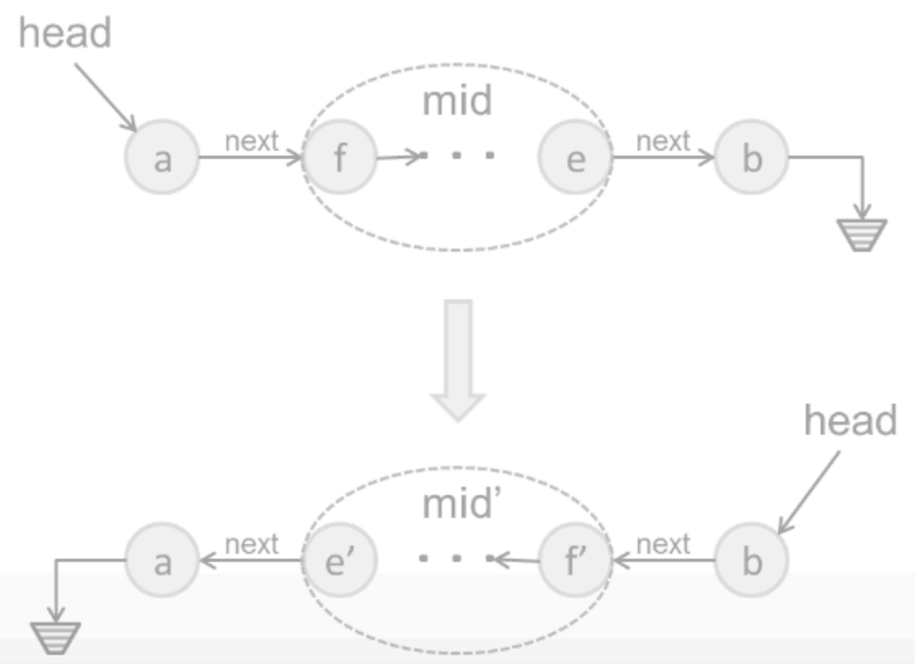
```
/* Average of x and y without using x+y (avoid overflow)*/
int avg(int x, int y){
  int t = expr({x/2, y/2, x%2, y%2, 2 }, {PLUS, DIV});
  assert t == (x+y)/2;
  return t;
}
```

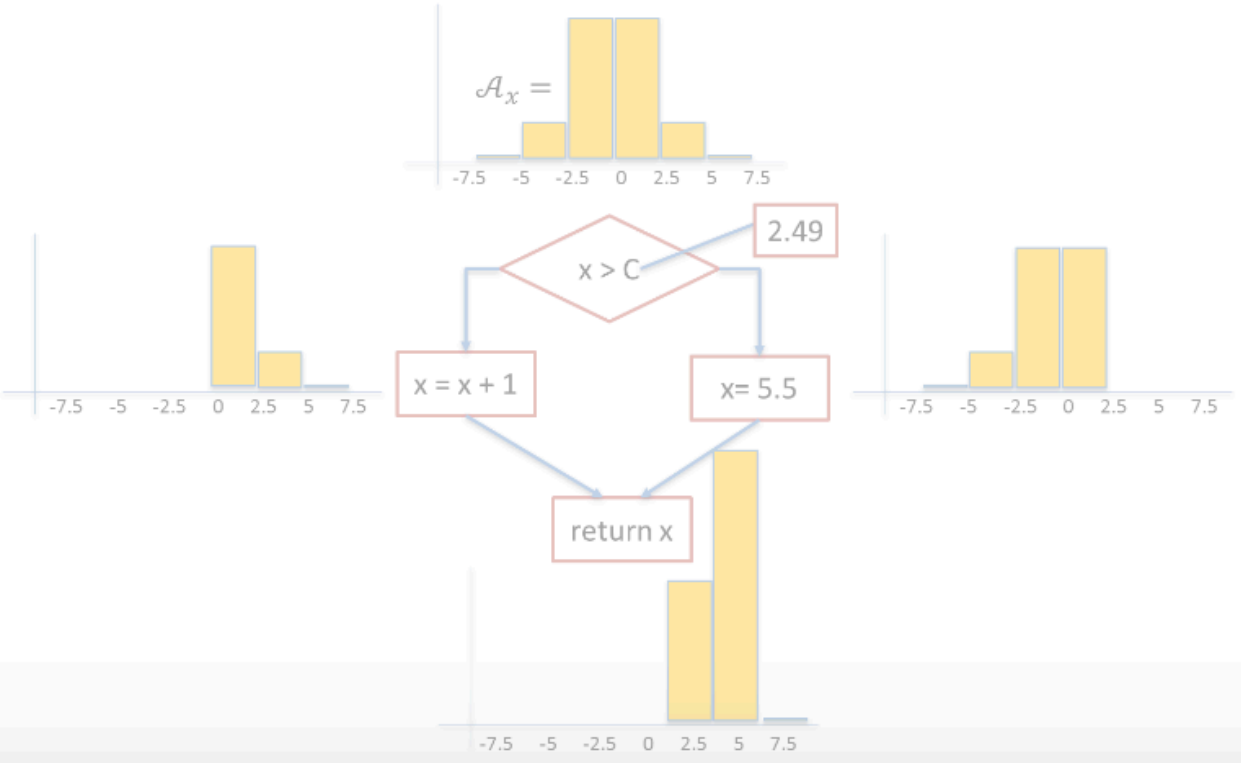




```
{
   s  = n.succ;
   p = n.pred;
   p.succ = s;
   s.pred = p;
}
```

Module II: Synthesizing Complex Programs

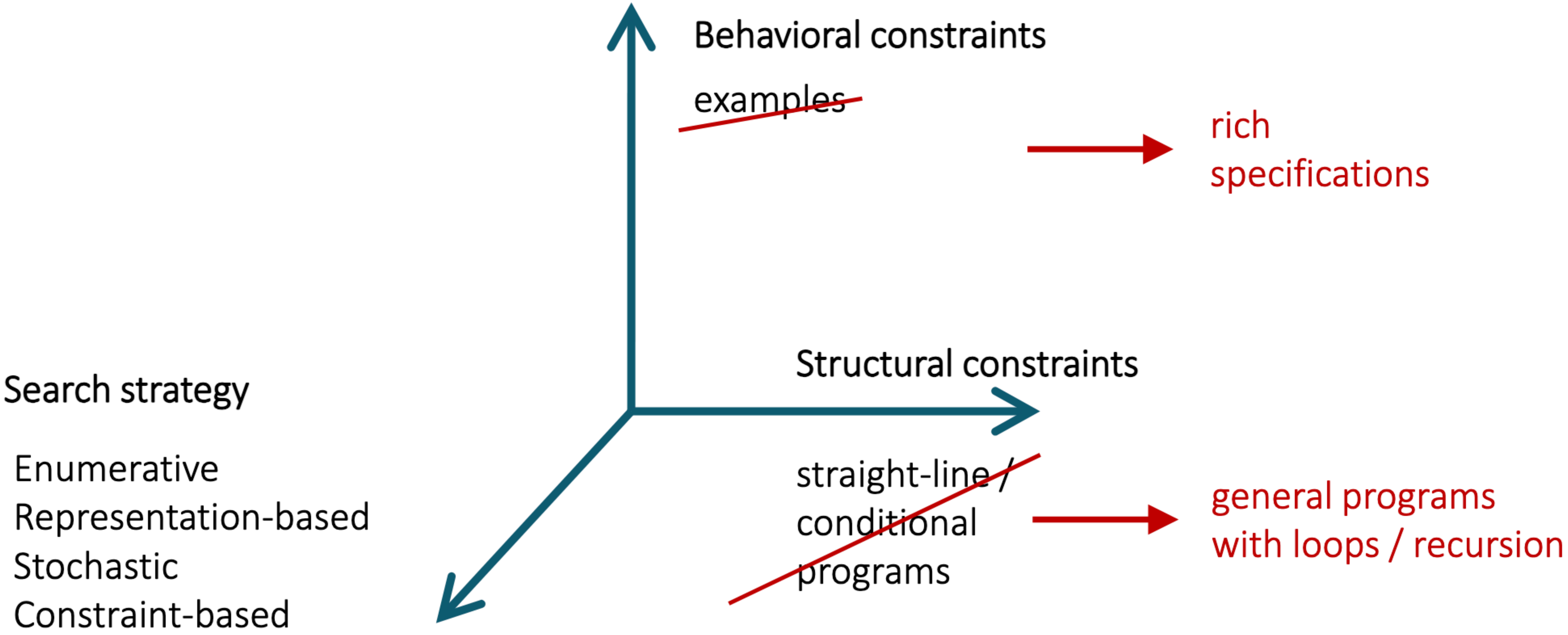




$$\varphi(p)$$

$$Sk[c](in)$$

Module I vs II



Behavioral constraints

~~examples~~ → rich specifications

Structural constraints

~~straight-line / conditional programs~~ → general programs with loops / recursion

Search strategy

Enumerative
Representation-based
Stochastic
Constraint-based

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 $a > 0 \wedge b > 0$
  ensures  $a \% c = 0 \;\wedge\; b \% c = 0$
           $\forall d \;.\; c < d \;\;\Rightarrow\;\; \mathrm{a} \% \mathrm{d} \neq 0 \;\vee\; b \% d \neq 0$
{
  int x , y := a, b;
  while (x != y) {
    if (x > y) x := ?;
    else y := ?;
}}
```

infinitely many inputs

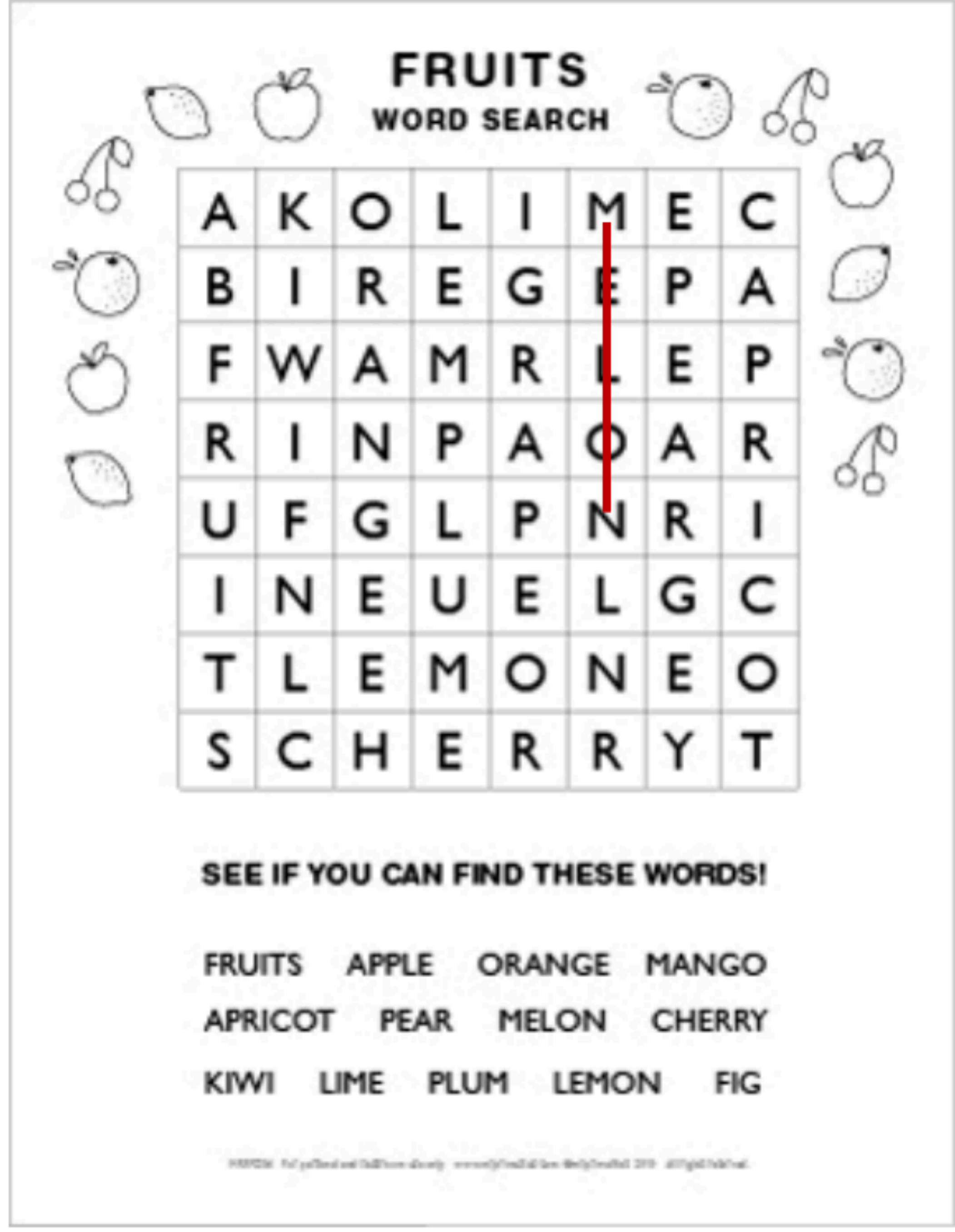
cannot validate by testing

infinitely many paths!

hard to generate constraints

Why is this hard?

Synthesis from examples



FRUITS
WORD SEARCH

A	K	O	L	I	M	E	C
B	I	R	E	G	E	P	A
F	W	A	M	R	L	E	P
R	I	N	P	A	O	A	R
U	F	G	L	P	N	R	I
I	N	E	U	E	L	G	C
T	L	E	M	O	N	E	O
S	C	H	E	R	R	Y	T

SEE IF YOU CAN FIND THESE WORDS!

FRUITS APPLE ORANGE MANGO
APRICOT PEAR MELON CHERRY
KIWI LIME PLUM LEMON FIG

validation was easy!

Synthesis from specifications

SEE IF YOU CAN FIND ANY KLINGON FRUIT!

validation is hard!
(and search is still hard)

Inductive generalization vs Deductive specialization

Constraint-based synthesis with Program Sketching

Reading: https://link.springer.com/article/10.1007/s10009-012-0249-7

Constraint-based synthesis

Key idea1:

- Search as “curve fitting”
- “curve” is a parameterized family of functions
- $H = \{\ P[c] \,\big|\, c \in C\}$

Neo did something along these lines

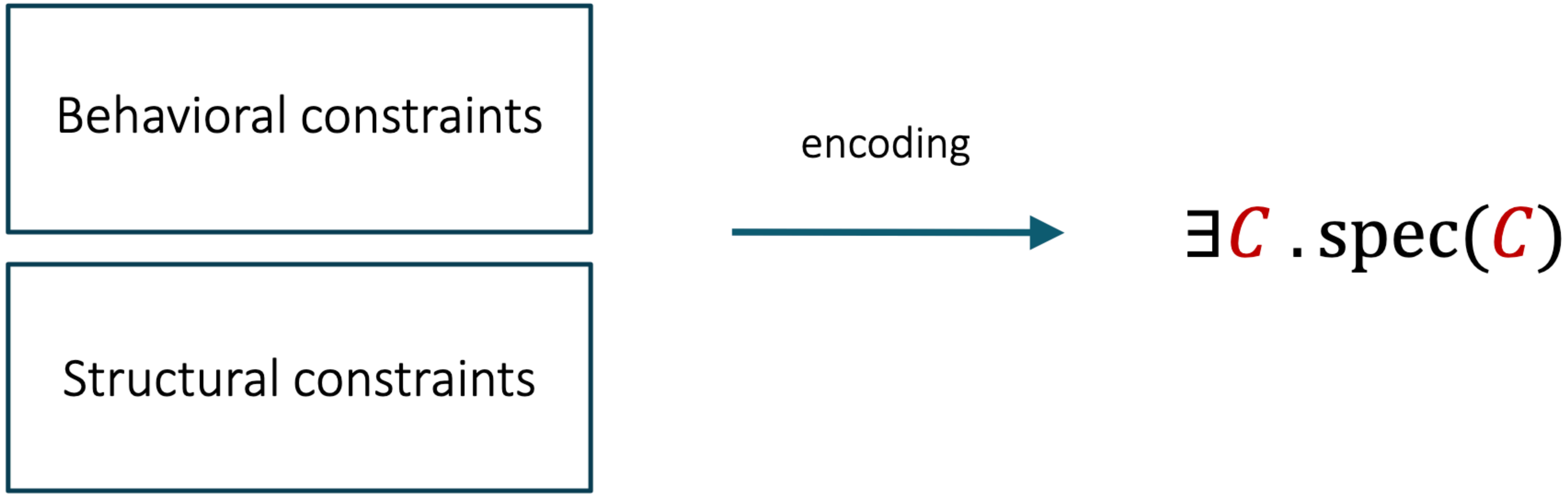
Key idea 2:

- Define a language to describe parameterized programs

Key idea 3:

- “Solve” instead of search

Constraint-based synthesis



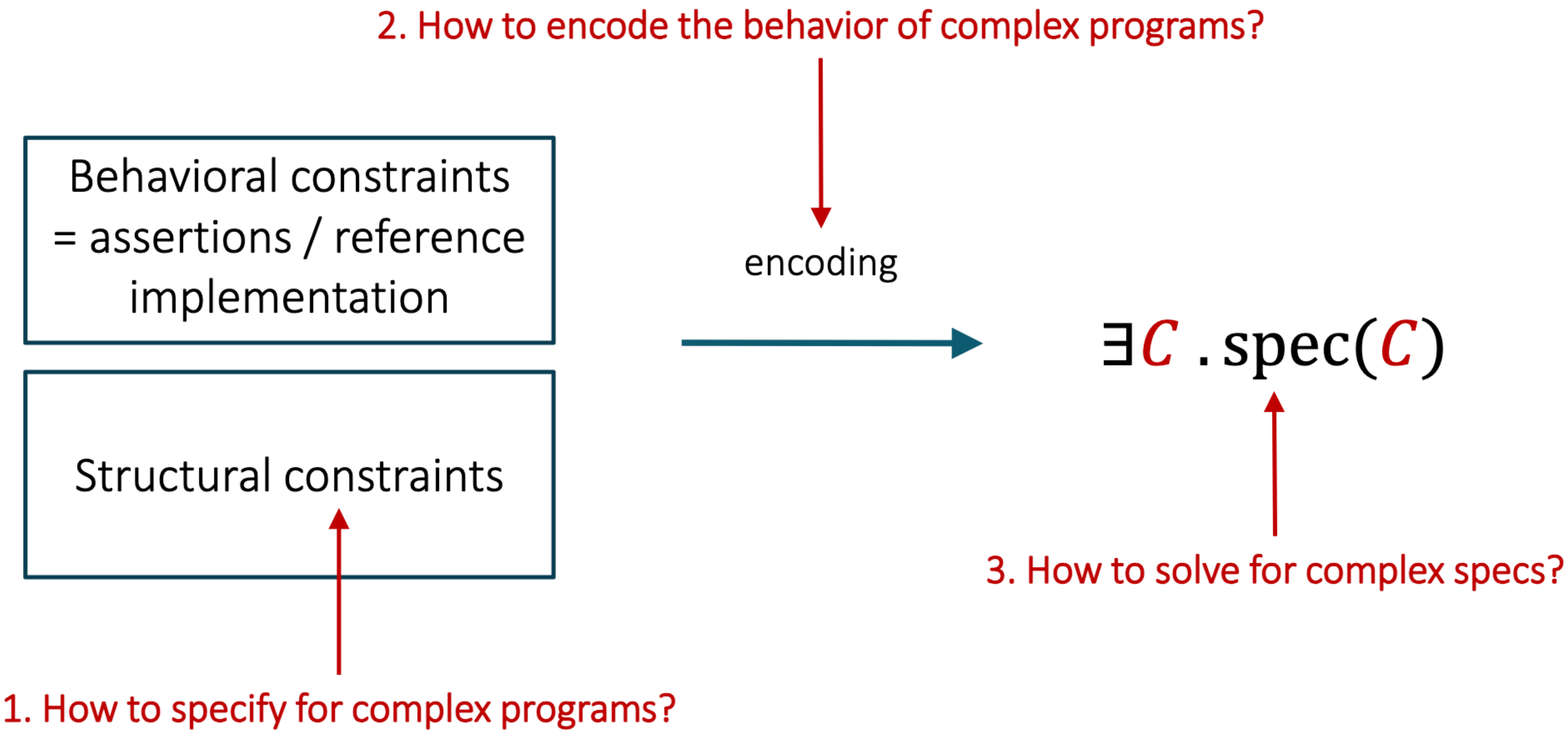
Behavioral constraints

Structural constraints

encoding

$\exists C\,.\,\mathrm{spec}(C)$

CBS for complex programs



2. How to encode the behavior of complex programs?

Behavioral constraints = assertions / reference implementation

Structural constraints

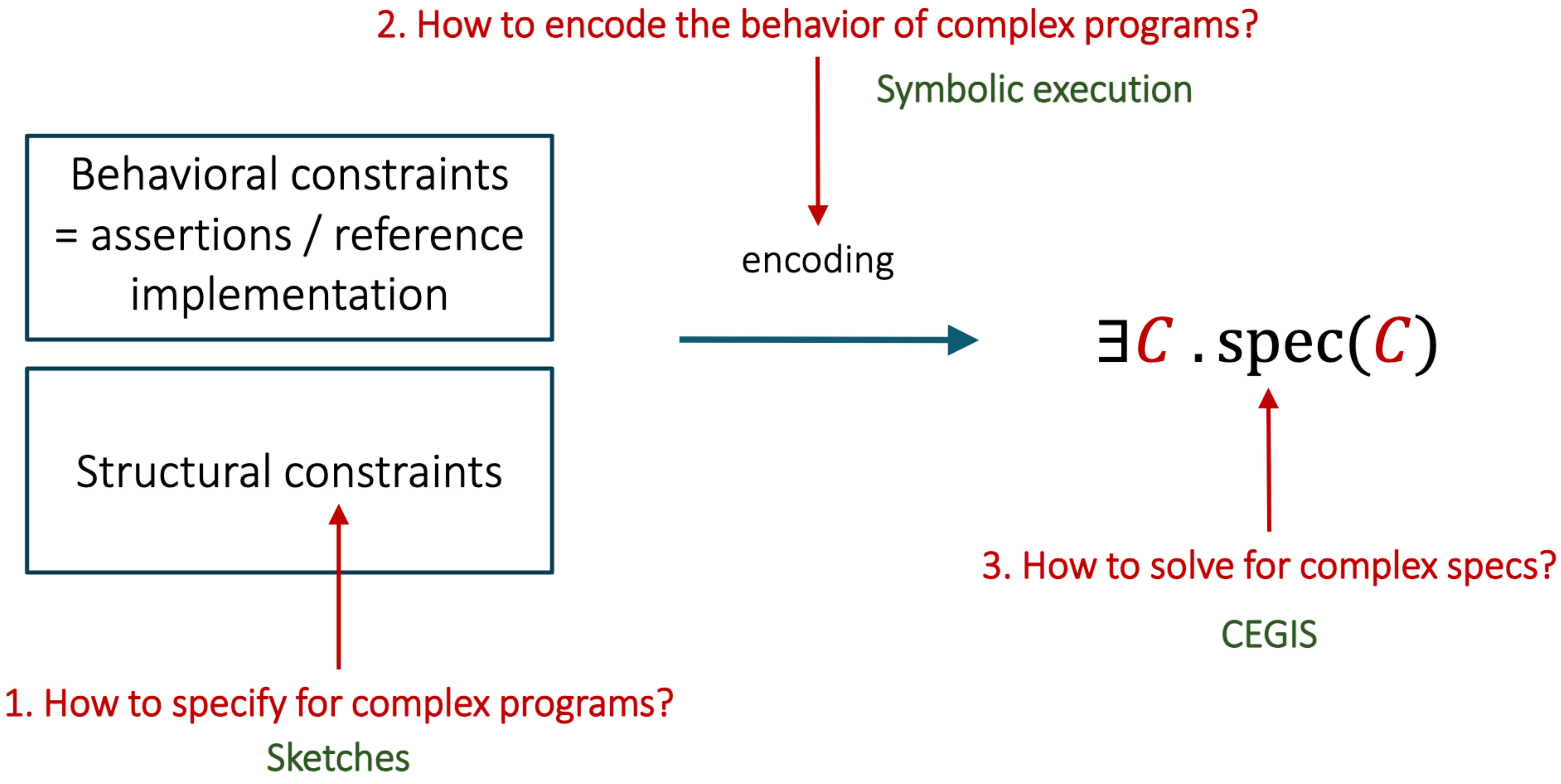
encoding

$\exists C\,.\,\mathrm{spec}(C)$

3. How to solve for complex specs?

1. How to specify for complex programs?

Program Sketching



2. How to encode the behavior of complex programs?

Symbolic execution

Behavioral constraints = assertions / reference implementation

Structural constraints

encoding

$\exists C\ .\, \text{spec}(C)$

3. How to solve for complex specs?

CEGIS

1. How to specify for complex programs?

Sketches

Synthesis with constraints

Overview of the Sketch language

Turning synthesis problems into constraints

Efficient constraint solving

Language Design Strategy

- Two main approaches for CBS
  - First: Give the user a high level notation to define the program space
    - Then use a compiler to translate that into a parametirct program P[c].
    - Brahma (bag of components)
    - SyGuS (CFG)
  - Second: provide the user with a rich and expressive language for directly writing parametric programs.
    - significant control over the program space.
    - More complicated inputs required.
    - Sketching

The Sketch Language

- simple imperative language very similar to Java
  - heap allocated structures, high-order functions and polymorphism (generics in Java), etc.
- Additional Unique features:
  - Unknown constants
  - Harnesses
  - Generator functions

Unknown Constants

Extend base language with one construct

Constant hole: **??**

Type is inferred from the context

```
int bar (int x)
{
    int t = x * ??;
    assert t == x +
x;
    return t;
}
```

```
int bar (int x)
{
    int t = x * 2;
    assert t == x +
x;
    return t;
}
```

Synthesizer replaces **??** with a constant

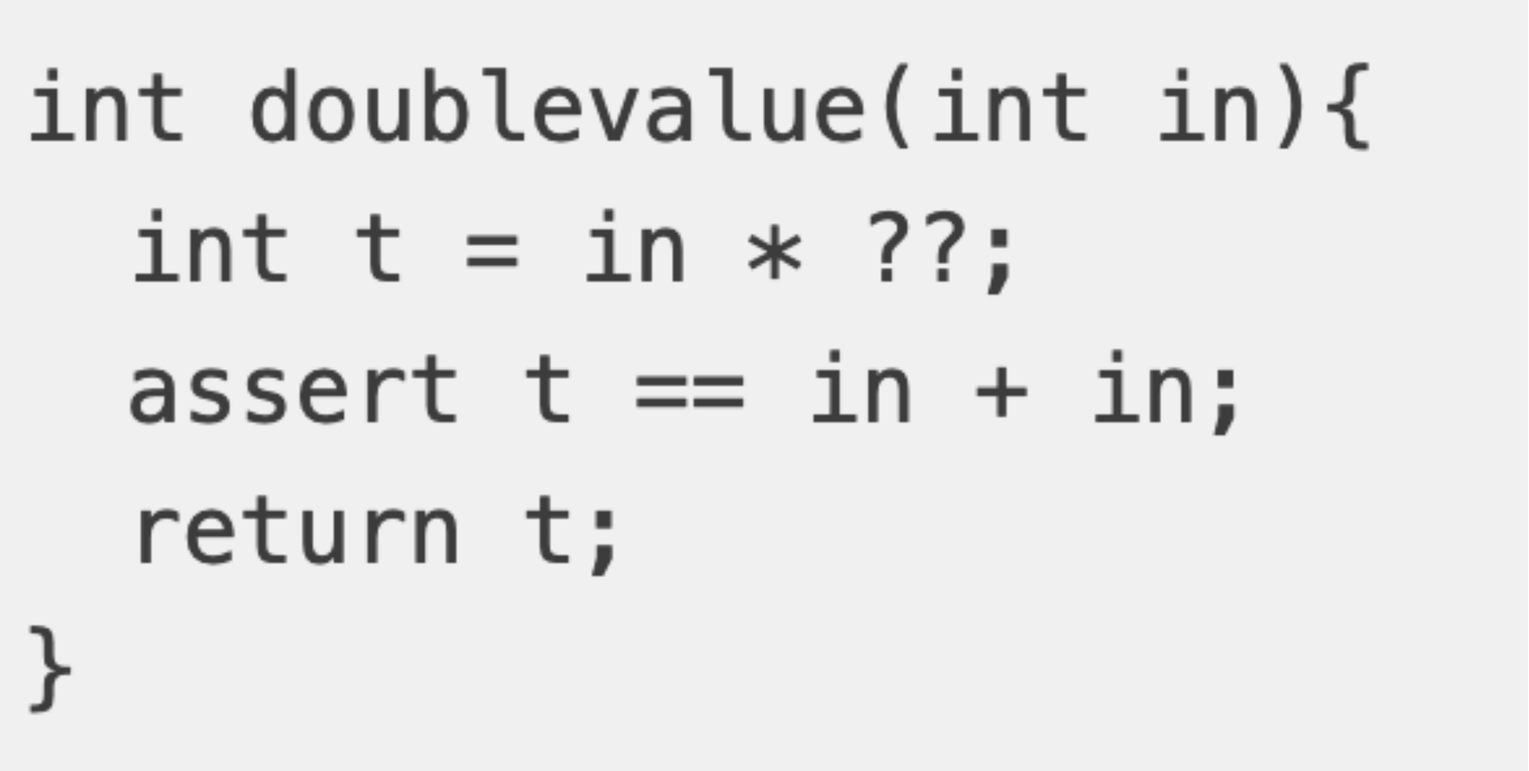
High-level constructs defined in terms of **??**

Unknown constant→ Sets of Expressions

- Expressions with `??` == sets of expressions
  - linear expressions `x*?? + y*??`
  - polynomials `x*x*?? + x*?? + ??`
  - sets of variables `?? ? x : y`

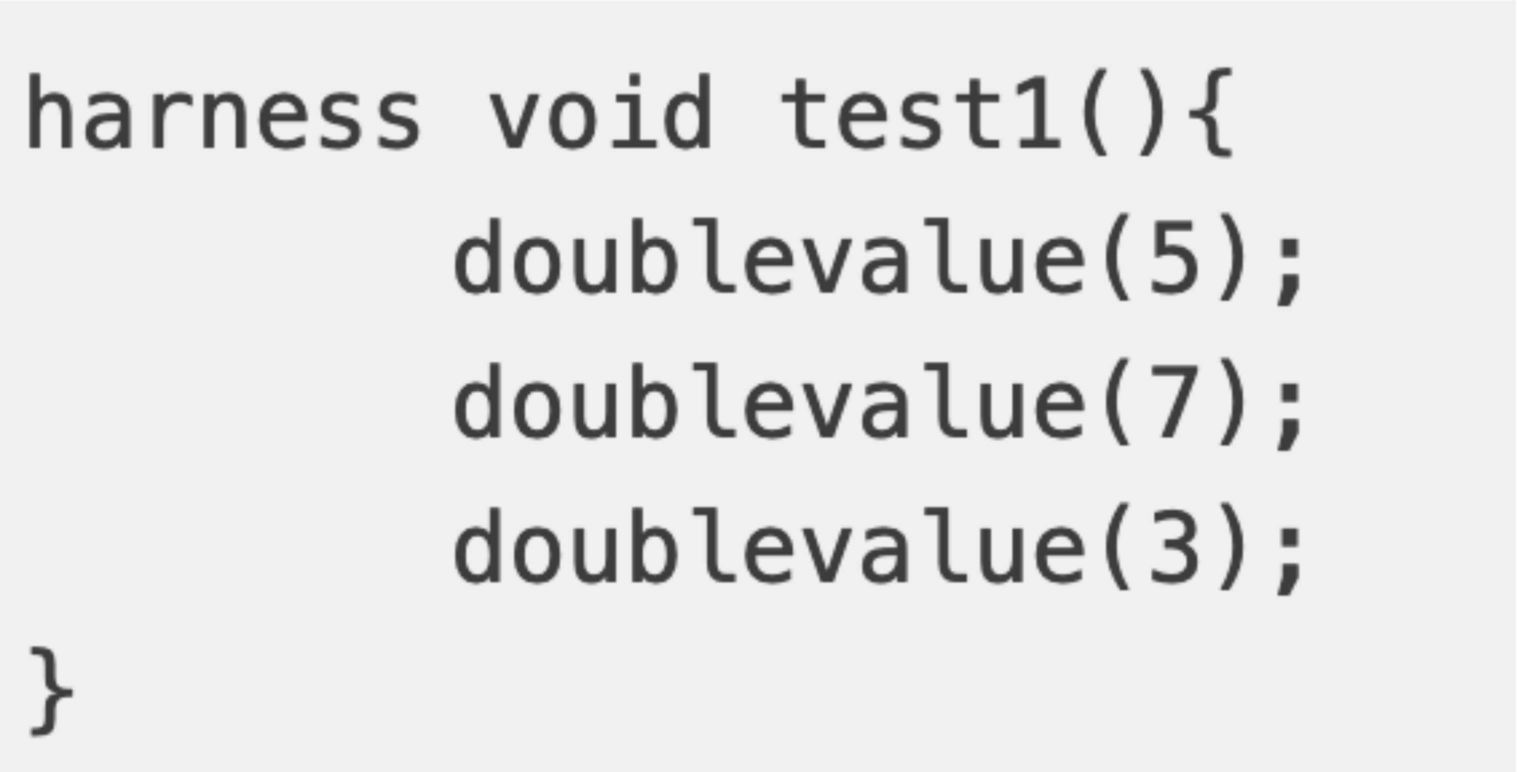
Harnesses/Test Harness

- a function that when invoked must not trigger any assertion violations.



```
int doublevalue(int in){
  int t = in * ??;
  assert t == in + in;
  return t;
}
```

A sketch example



```
harness void test1(){
        doublevalue(5);
        doublevalue(7);
        doublevalue(3);
}
```

A test harness

- A test harness can also take inputs on their own.

Example: Registerless Swap

- Swap two words without an extra temporary

```
int W = 32;

void swap(ref bit[W] x, ref bit[W] y){
    if(??){ x = x ^ y;}else{ y = x ^ y; }
    if(??){ x = x ^ y;}else{ y = x ^ y; }
    if(??){ x = x ^ y;}else{ y = x ^ y; }
}

harness void main(bit[W] x, bit[W] y){
    bit[W] tx = x; bit[W] ty = y;
    swap(x, y);
    assert x==ty && y == tx;
}
```

From simple to complex holes

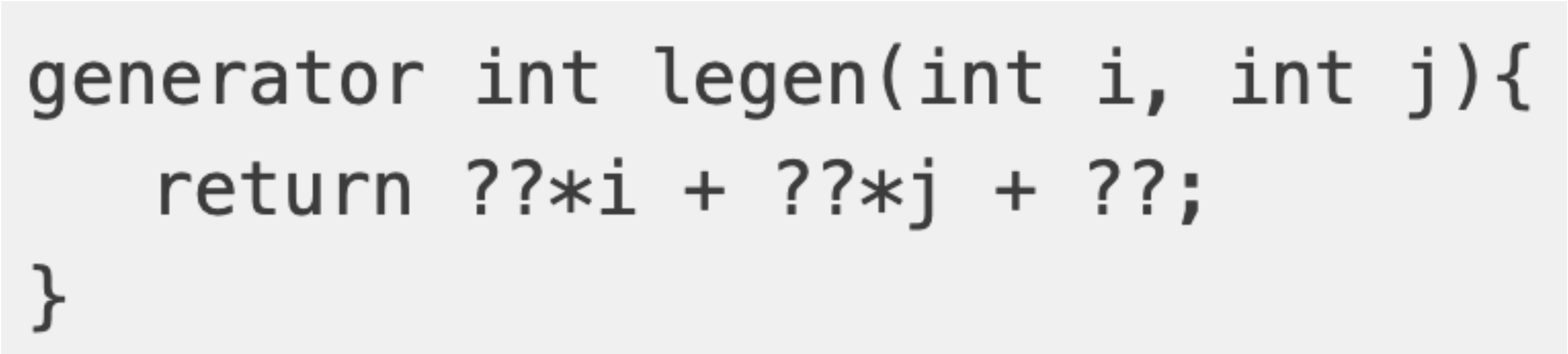
- We need to compose ?? to form complex holes

- Borrow ideas from generative programming
  - Define generators to produce families of functions
  - Use partial evaluation aggressively

Generators

- Look like a function
  - but are partially evaluated into their calling context

- Key feature:
  - Different invocations → Different code
  - Can recursively define arbitrary families of programs



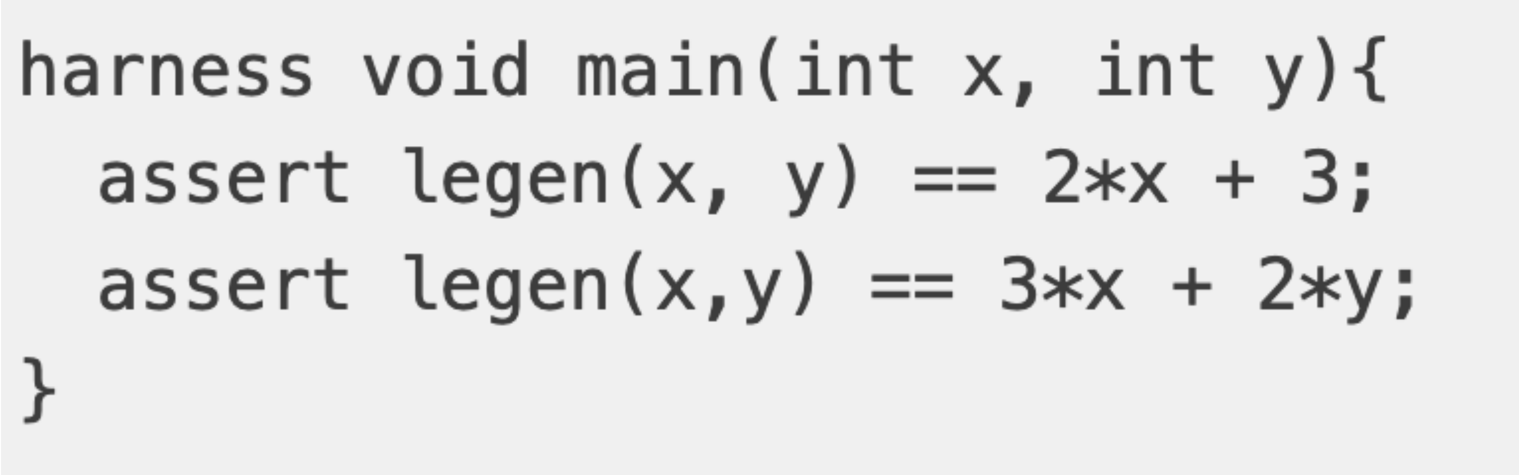
```
generator int legen(int i, int j){
   return ??*i + ??*j + ??;
}
```

A simple generator for set of linear function of two parameters

Properties of Generators

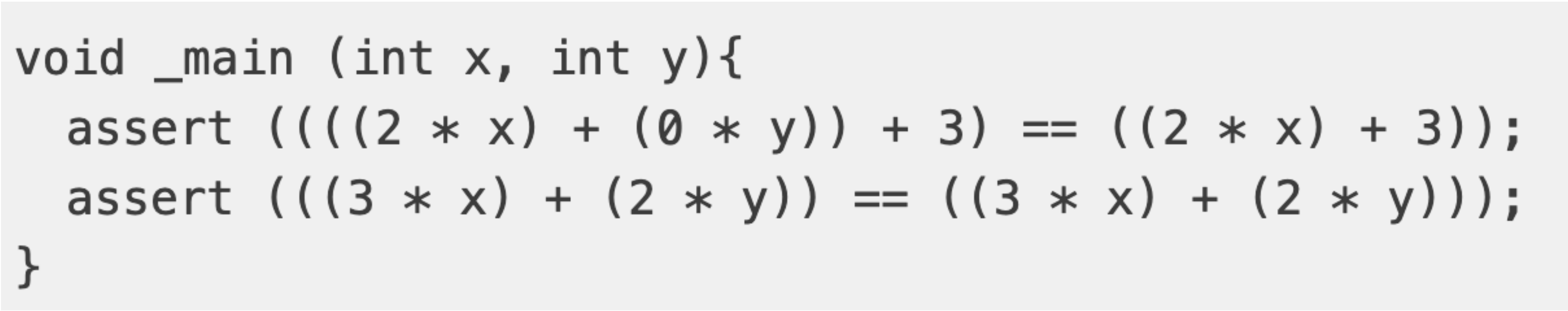
Are these just glorified Macros?

- Generator function can be used anywhere in the code in the same way a function.
- However different semantics.
  - every call replaced by a concrete piece of code in the space of code fragments defined by the generator.
  - Different calls to the generator function can produce different code fragments.



```
harness void main(int x, int y){
  assert legen(x, y) == 2*x + 3;
  assert legen(x,y) == 3*x + 2*y;
}
```

Harness using the generator



```
void _main (int x, int y){
  assert ((((2 * x) + (0 * y)) + 3) == ((2 * x) + 3));
  assert (((3 * x) + (2 * y)) == ((3 * x) + (2 * y)));
}
```

Concerete program after solving

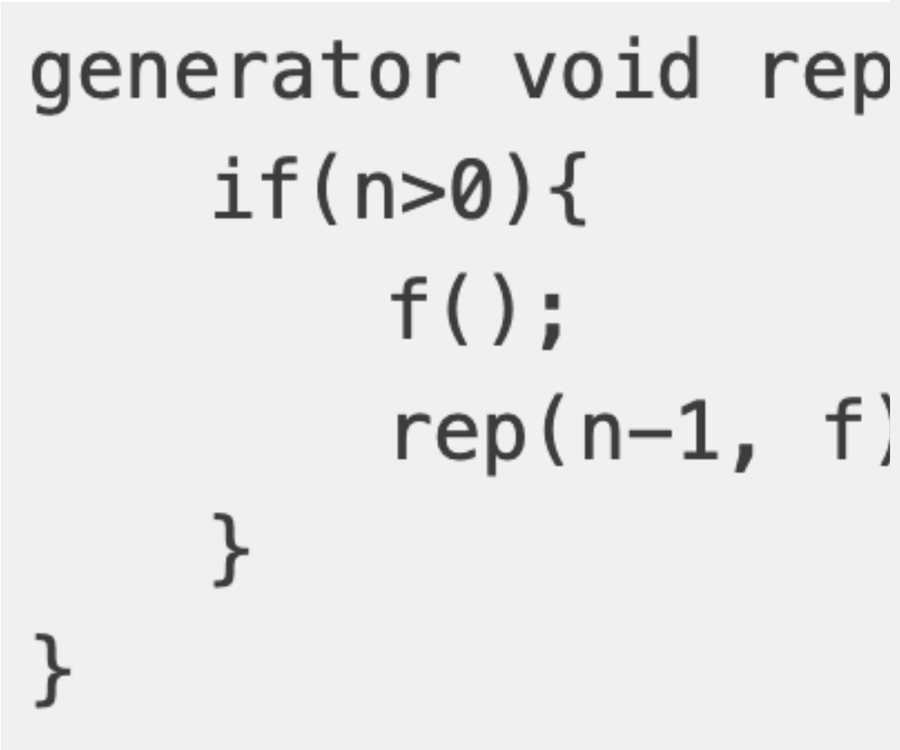
Real Power: Recursion

```
/**
 * Generate the set of all bit-vector expressions
 * involving +, &, xor and bitwise negation (~).
 * the bnd param limits the size of the generated expression.
 */

generator bit[W] gen(bit[W] x, int bnd){
    assert bnd > 0;
    if(??) return x;
    if(??) return ??;
    if(??) return ~gen(x, bnd-1);
    if(??){
        return {| gen(x, bnd-1) (+ | & | ^) gen(x, bnd-1) |};
    }
}
```

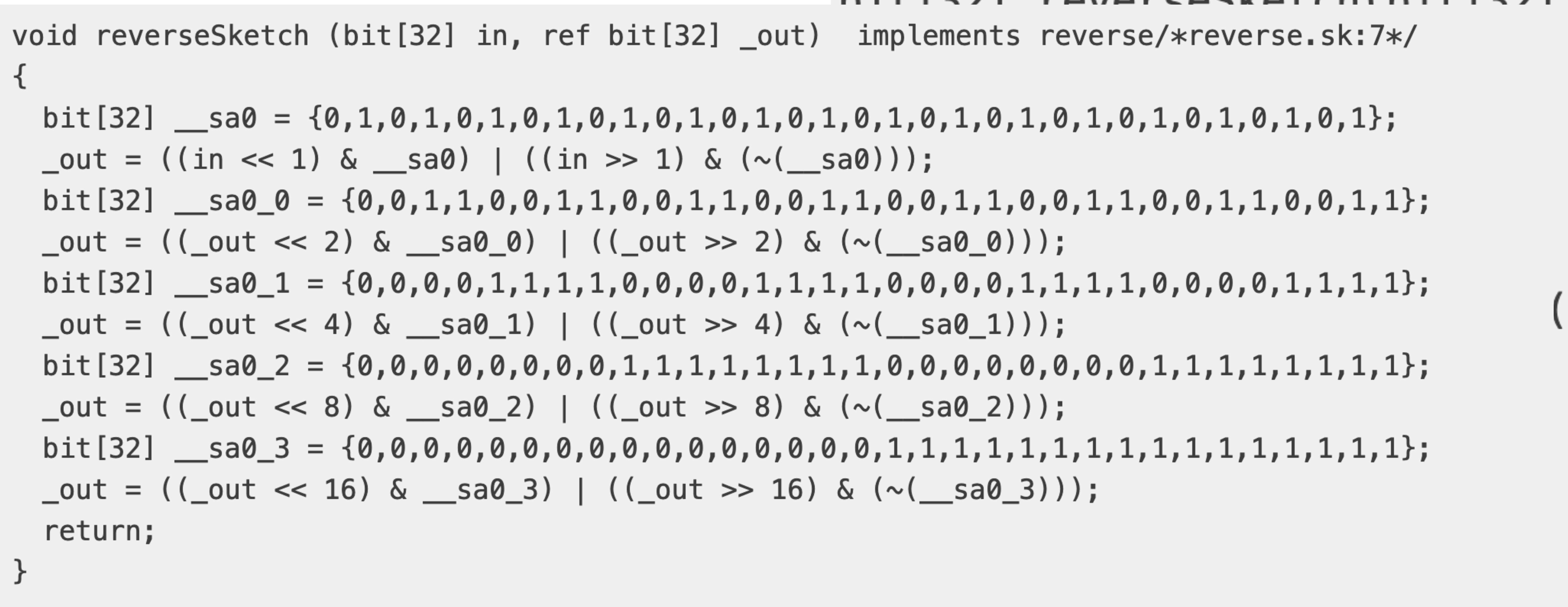
Real Power: Closures + High Order Generators

```
generator void rep(int n, fun f){
    if(n>0){
        f();
        rep(n-1, f);
    }
}

bit[16] reverseSketch(bit[16] in) {
    bit[16]  t = in;
    int s = 1;
    generator void tmp(){
        bit[16] m = ??;
        t = ((t << s)&m )| ((t >> s)&(~m));
        s = s*??;
    }
    rep(??, tmp);
    return t;
}
```

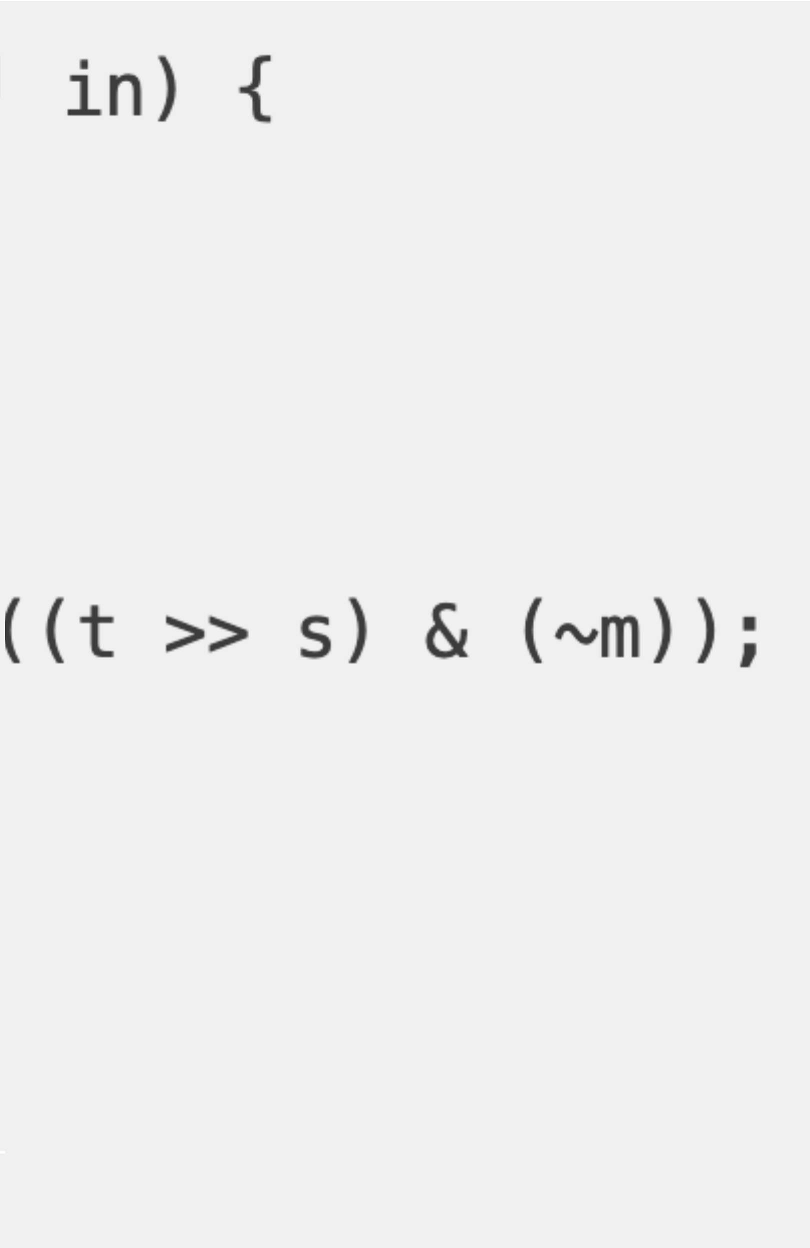
Real Power: Higher Order terms + Closures



```
generator void rep
    if(n>0){
        f();
        rep(n-1, f)
    }
}
```



```
void reverseSketch (bit[32] in, ref bit[32] _out)  implements reverse/*reverse.sk:7*/
{
  bit[32] __sa0 = {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1};
  _out = ((in << 1) & __sa0) | ((in >> 1) & (~(__sa0)));
  bit[32] __sa0_0 = {0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1};
  _out = ((_out << 2) & __sa0_0) | ((_out >> 2) & (~(__sa0_0)));
  bit[32] __sa0_1 = {0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1};
  _out = ((_out << 4) & __sa0_1) | ((_out >> 4) & (~(__sa0_1)));
  bit[32] __sa0_2 = {0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1};
  _out = ((_out << 8) & __sa0_2) | ((_out >> 8) & (~(__sa0_2)));
  bit[32] __sa0_3 = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
  _out = ((_out << 16) & __sa0_3) | ((_out >> 16) & (~(__sa0_3)));
  return;
}
```



```
bit[32] reverseSketch(bit[32] in) {

                                   ((t >> s) & (~m));

}
```

Takes a function/

it n times.

Interesting comp. pattern: a particular kind of

of operation to be repeated with each iteration a distinct operation

Syntactic Sugar

- {| RegExp |}

- RegExp supports choice ‘|’ and optional ‘?’
  - can be used arbitrarily within an expression
    - to select operands `{| (x | y | z) + 1 |}`
    - to select operators `{| x (+ | -) y |}`
    - to select fields `{| n(.prev | .next)? |}`
    - to select arguments `{| foo( x | y, z) |}`

- Set must respect the type system
  - all expressions in the set must type-check
  - all must be of the same type

repeat

- Avoid copying and pasting
  - `repeat(n){ s}` → $\underbrace{\texttt{s;s;…s;}}_{n}$
  - each of the n copies may resolve to a distinct stmt
  - `n` can be a hole too.

Example: Reversing bits

```
pragma options "--bnd-cbits 3 ";

int W = 32;

bit[W] reverseSketch(bit[W] in) {

    bit[W]  t = in;
    int s = 1;
    int r = ??;
    repeat(??){
        bit[W] tmp1 = (t << s);
        bit[W] tmp2 = (t >> s);
        t = tmp1 {|} tmp2;
          // Syntactic sugar for m=??, (tmp1&m | tmp2&~m).
        s = s*r;
    }
    return t;
}
```

Framing the synthesis problem

- Goal: Find a function from holes to values
  - Easy in the absence of generators

```
bit[W] isolateSk (bit[W] x) implements isolate0 {

      return !(x + ??1) & (x + ??2) ;
}
```

  - Finite set of holes so function is just a table
  - Call this function ϕ and the program thus is parameterized with ϕ.