Lecture 19: Program Synthesis

17-355/17-665/17-819: Program Analysis

Rohan Padhye

March 31, 2022

* Course materials developed with Jonathan Aldrich amd Claire Le Goues
With slide inspiration gratitude to Emina Torlak and Ras Bodik

Program Synthesis Overview

- A mathematical characterization of program synthesis: prove that $$\exists P \,.\, \forall x \,.\, \varphi(x, P(x))$$
- In constructive logic, the witness to the proof of this statement is a program P that satisfies property φ for all input values x

Program Synthesis Overview

- A mathematical characterization of program synthesis: prove that

$$\exists P \,.\, \forall x \,.\, \varphi(x, P(x))$$

- In constructive logic, the witness to the proof of this statement is a program P that satisfies property φ for all input values x
- What could the inferred program P be?
  - Historically, a protocol, interpreter, classifier, compression algorithm, scheduling policy, cache coherence policy, …
- How is property φ expressed?
  - Historically, as a formula, a reference implementation, input/output pairs, traces, demonstrations, a sketch, …

Exercise: specify P_max(list)

- Specify a program $P_{max}(l)$ that finds the maximum number in a list l. How many different ways can you do it?

Expressing User Intent

- How do we constrain the program to be synthesized?
  - Express what we know about the problem and/or solution
  - Usually incomplete
- Two forms of specification can constrain synthesis
  - Observable behavior: input/output relations, executable specification, safety property
  - Structural properties: constraints on internal computation, such as a sketch, template, assertions about structure (e.g. number of iterations)

The Search Space of Programs

- Constraining the search space can help make synthesis feasible
  - Subset of a real programming language?
  - Grammar for combining fixed set of operators and control structures?
  - DSL?
  - Logic?

Two approaches to searching for programs

- Deductive synthesis
  - Maps a high-level specification to an implementation, using a theorem prover
  - Efficient, provably correct
  - Require complete specifications, sufficient axiomatization of the domain
    - Can be as complicated as writing the program itself!
  - Used for e.g. controllers
  - A lot like compilation!
- Inductive synthesis
  - Takes a partial, perhaps multi-modal specification and constructs a program that satisfies it
  - Flexible in specification requirements, require no axioms
  - May be less efficient, weaker guarantees on correctness/optimality
  - Search techniques: brute-force, probabilistic, genetic programming, logical reasoning
  - Major current focus of research

Inductive synthesis

Find a program correct on a set of inputs and hope (or verify) that it's correct on other inputs.

A **partial program** syntactically defines the candidate space.

Inductive synthesis search phrased as a **constraint problem**.

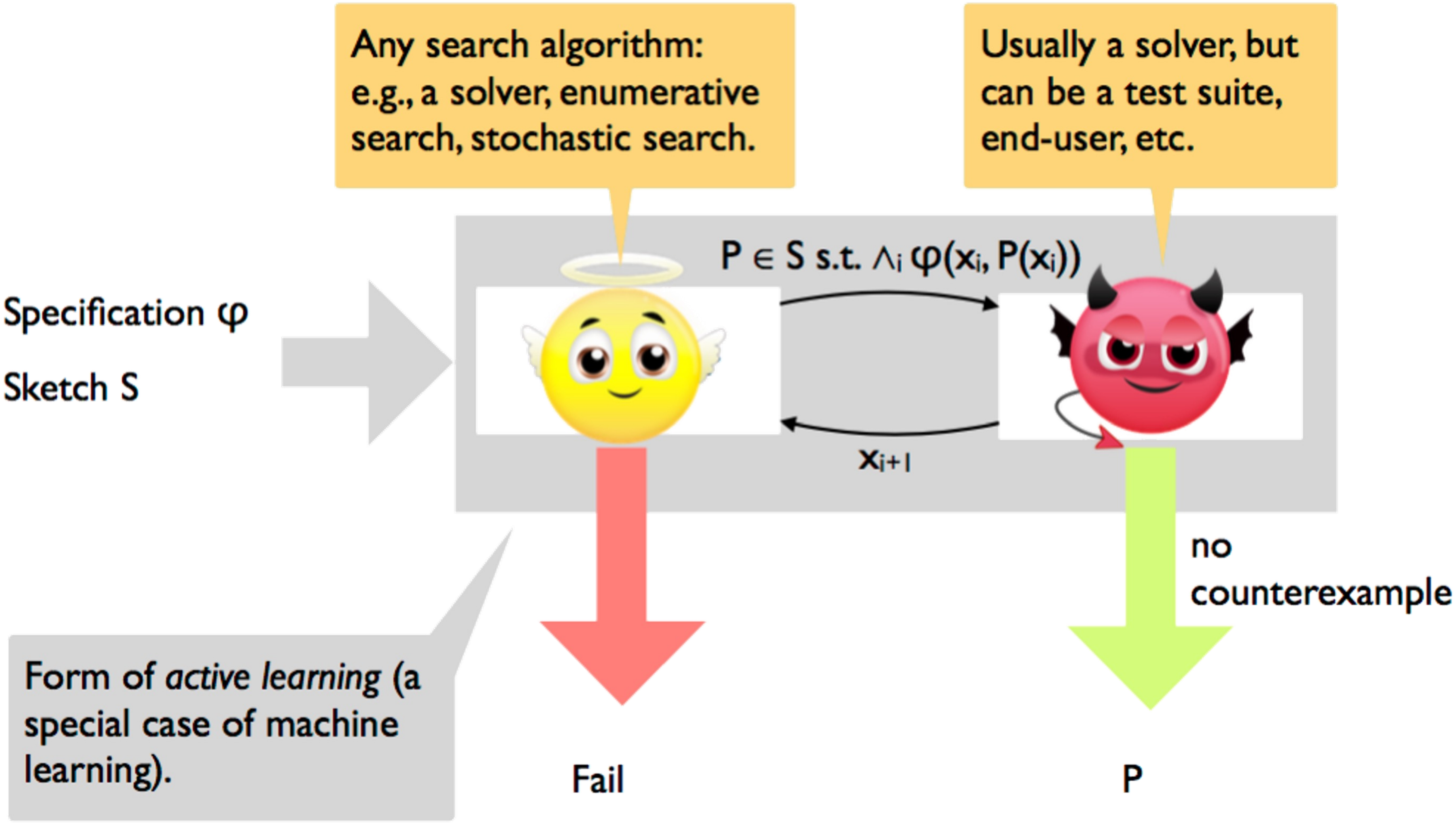
Program found by (symbolic) interpretation of a (space of) candidates, not by deriving the candidate.

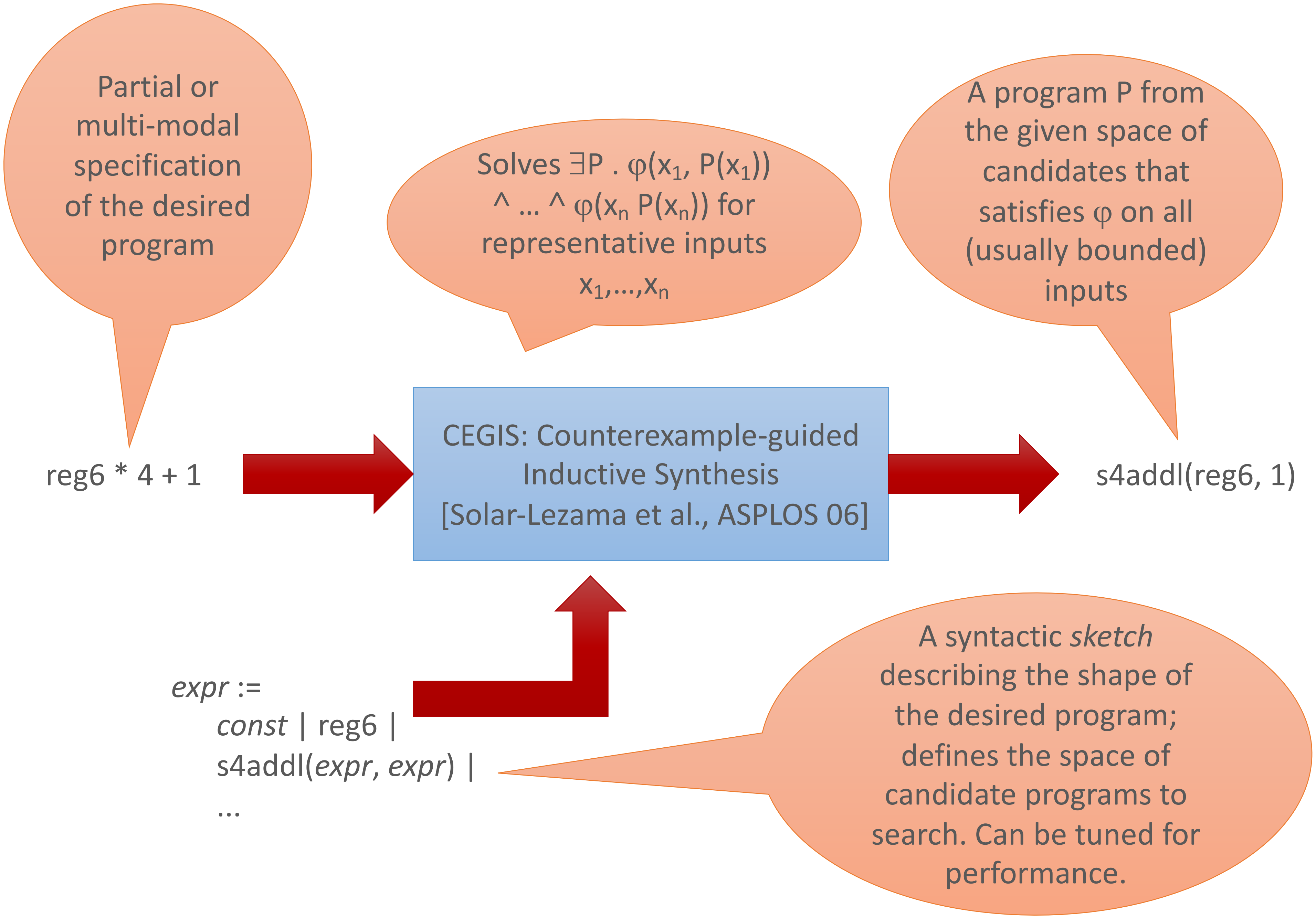
So, to find a program, we need only an interpreter,
not a sufficient set of derivation axioms.

Exercise: validate P_max(list)

- Given a candidate program $P_{max}(l)$ that finds the maximum number in a list l, how can you check if it is correct?

Overview of CEGIS




Partial or multi-modal specification of the desired program
Solves $\exists P . \varphi(x_1, P(x_1)) \wedge \ldots \wedge \varphi(x_n P(x_n))$ for representative inputs $x_1,\ldots,x_n$
A program P from the given space of candidates that satisfies φ on all (usually bounded) inputs
reg6 * 4 + 1
CEGIS: Counterexample-guided Inductive Synthesis [Solar-Lezama et al., ASPLOS 06]
s4addl(reg6, 1)
expr :=
const | reg6 |
s4addl(*expr*, *expr*) |
...
A syntactic *sketch* describing the shape of the desired program; defines the space of candidate programs to search. Can be tuned for performance.

Sketching intuition

Extend the language with two constructs

spec:
```
int foo (int x) {
    return x + x;
}
```

$\phi(x, y): y = \mathtt{foo}(x)$

sketch:
```
int bar (int x) implements foo {
    return x << ??;
}
```

`??` substituted with an int constant meeting ϕ

result:
```
int bar (int x) implements foo {
    return x << 1;
}
```

EXAMPLE: BIT COUNTING

```
bit[W] countBits(bit[W] x)
{
    int count = 0;
    for (int i = 0; i < W; i++) {
        if (x[i]) count++;
    }
    return count;
}
```

Intuition

```
Bit string: ABCDEFGHIJKLMNOP        Bit string: ABCDEFGHIJKLMNOP
Mask:       0101010101010101        Bits >> 1:  0ABCDEFGHIJKLMNO
Result:     0B0D0F0H0J0L0N0P        Mask:       0101010101010101
                                    Result:     0A0C0E0G0I0K0M0O

                 0B   0D   0F   0H   0J   0L   0N   0P
      +          0A   0C   0E   0G   0I   0K   0M   0O
      =          A+B  C+D  E+F  G+H  I+J  K+L  M+N  O+P

                      Repeat

    0 A+B+C+D 0 E+F+G+H 0 I+J+K+L 0 M+N+O+P

   0000 A+B+C+D+E+F+G+H 0000 I+J+K+L+M+N+O+P

 0000 0000 000 A+B+C+D+E+F+G+H+I+J+K+L+M+N+O+P
```

```
bit[W] countSketched(bit[W] x)
      implements countBits {
   loop (??) {
      x = (x & ??) +
                 ((x >> ??) & ?? );
   }
 return x;
}
```

```
bit[W] countSketched(bit[W] x)
{
 x = (x & 0x5555) +
        ((x >> 1) & 0x5555);
 x = (x & 0x3333) +
        ((x >> 2) & 0x3333);
 x = (x & 0x0077) +
        ((x >> 8) & 0x0077);
 x = (x & 0x000F) +
        ((x >> 4) & 0x000F);
 return x;
}
```

Oracle-Guided Inductive Synthesis

1. Generalize CEGIS (counterexample-guided inductive synthesis)
   - From sketches to arbitrary programs

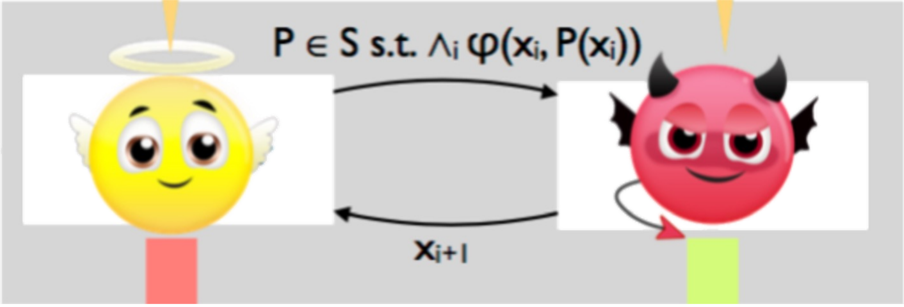
2. Synthesize programs from components

CEGIS: A Mathematical View

- Let's formalize Counterexample-Guided Inductive Synthesis (CEGIS)
- Consider a formalization of synthesizing a *max* function for lists

$$\exists P_{max} \forall l, m : P_{max}(l) = m \Rightarrow (m \in l) \wedge (\forall x \in l : m \geqslant x)$$

- CEGIS iterates between synthesis from examples and counterexample generation



- How do we generate a counterexample?

Counterexample generation, formalized

- Let's say we have a candidate program P_{max}. Does it meet the spec?
  - Here's how that can be formalized:

$$\forall l, m : P_{max}(l) = m \Rightarrow (m \in l) \wedge (\forall x \in l : m \geqslant x)$$

- By De Morgan's Law, this is equivalent to disproving the negation:

$$\exists l, m : (P_{max}(l) = m) \wedge (m \notin l \vee \exists x \in l : m < x)$$

- This finds a list l and a corresponding incorrect output m
- Let's tweak this to generate the correct output, m^*:

$$\exists l, m^* : (P_{max}(l) \neq m^*) \wedge (m^* \in l) \wedge (\forall x \in l : m^* \geqslant x)$$

- We can use this to help generate the next version of P_{max}

Oracle-Guided Component-Based Program Synthesis (from examples)

- Goal: given a set of N components $f_1, \ldots, f_N$ and a set of T input/output pairs $\langle\alpha_0, \beta_0\rangle \ldots \langle\alpha_T, \beta_T\rangle$, synthesize a function f such that: $\forall i \in [0, T]: f(\alpha_i) = \beta_i$.
- We search for programs of a particular form:

Put inputs in variables

Compute N functions, each of which has arguments

Choices: fill in the ?s
- What order are the functions in?
- What variables are passed to each function?
- What variable is returned?

$$
\begin{array}{ll}
0 & z_0 := \texttt{input}^0 \\
1 & z_1 := \texttt{input}^1 \\
\ldots & \ldots \\
m & z_m := \texttt{input}^m \\
m+1 & z_{m+1} := f_?(z_?, \ldots, z_?) \\
m+2 & z_{m+2} := f_?(z_?, \ldots, z_?) \\
\ldots & \ldots \\
m+N & z_{m+N} := f_?(z_?, \ldots, z_?) \\
m+N+1 & \texttt{return } z_?
\end{array}
$$

The program is defined by a set of variables

- Program input variable $\overrightarrow{Y}$
- Input to each component
- Output of each component
- Output of the program r

$$Q := \bigcup_{i=1}^{N} \overrightarrow{\chi}_i$$

$$R := \bigcup_{i=1}^{N} r_i$$

$$\begin{array}{ll}
0 & z_0 := \texttt{input}^0 \\
1 & z_1 := \texttt{input}^1 \\
\ldots & \ldots \\
m & z_m := \texttt{input}^m \\
m+1 & z_{m+1} := f_?(z_?, \ldots, z_?) \\
m+2 & z_{m+2} := f_?(z_?, \ldots, z_?) \\
\ldots & \ldots \\
m+N & z_{m+N} := f_?(z_?, \ldots, z_?) \\
m+N+1 & \texttt{return } z_?
\end{array}$$

Program variables are specified by location variables

- Location variable l_x specifies where x is defined
- L is the set of location variables

$$L \quad := \quad \{l_x | x \in Q \cup R \cup \overrightarrow{Y} \cup r\}$$

(again: component inputs, component results, program inputs, and program result)

$$\begin{array}{ll}
0 & z_0 := \texttt{input}^0 \\
1 & z_1 := \texttt{input}^1 \\
\ldots & \ldots \\
m & z_m := \texttt{input}^m \\
m+1 & z_{m+1} := f_?(z_?, \ldots, z_?) \\
m+2 & z_{m+2} := f_?(z_?, \ldots, z_?) \\
\ldots & \ldots \\
m+N & z_{m+N} := f_?(z_?, \ldots, z_?) \\
m+N+1 & \texttt{return}\ z_?
\end{array}$$

Example of Location Variables

- Imagine we have one input and one component, +
- Here's a sample program:

$$
\begin{array}{ll}
0 & z_0 := \texttt{input}^0 \\
1 & z_1 := z_0 + z_0 \\
2 & \texttt{return}\ z_1
\end{array}
$$

- This can be specified by the location variables

$$\{l_{r_+} \mapsto 1, l_{\chi_+^1} \mapsto 0, l_{\chi_+^2} \mapsto 0, l_r \mapsto 1, l_Y \mapsto 0\}$$

Practice with Location Variable Encodings

Assume two components, * and <<, each of which takes two inputs and produces a single output. Provide a map which assigns values to location variables that describe the following straight-line code. For your reference, the variables are: $\overrightarrow{Y}$ r $\overrightarrow{X}_i$ r_i

$z_0 = \text{input}_0$

$z_1 = \text{input}_1$

$z_2 = z_0 << z_1$ *// component <<*

$z_3 = z_2 * z_2$ *// component **

return z_2

Well-formedness constraints on the generated program

- Component inputs come from locations 0…M
  - M = number of inputs $|\vec{Y}|$ + number of functions N

$$\bigwedge_{x \in Q} (0 \leqslant l_x < M)$$

- Component outputs defined after program inputs

$$\bigwedge_{x \in R} (|\vec{Y}| \leqslant l_x < M)$$

- One component per line

$$\bigwedge_{x,y \in R, x \not\equiv y} (l_x \neq l_y)$$

- Component inputs are defined before use

$$\bigwedge_{i=1}^{N} \bigwedge_{x \in \vec{\chi}_i} l_x < l_{r_i}$$

$$\begin{array}{ll}
0 & z_0 := \texttt{input}^0 \\
1 & z_1 := \texttt{input}^1 \\
\ldots & \ldots \\
m & z_m := \texttt{input}^m \\
m+1 & z_{m+1} := f_?(z_?, \ldots, z_?) \\
m+2 & z_{m+2} := f_?(z_?, \ldots, z_?) \\
\ldots & \ldots \\
m+N & z_{m+N} := f_?(z_?, \ldots, z_?) \\
m+N+1 & \texttt{return } z_?
\end{array}$$

Functionality constraints

- Variables defined at the same location are the same (have the same value)
  - Basically: define value flow from definition to use

$$\bigwedge_{x,y \in Q \cup R \cup \overrightarrow{Y} \cup \{r\}} (l_x = l_y \Rightarrow x = y)$$

- The program inputs and outputs match a test case pair
  - We repeat this for all test cases

$$(\alpha = \overrightarrow{Y}) \wedge (\beta = r)$$

- Functional components obey their specification

$$(\bigwedge_{i=1}^{N} \phi_i(\overrightarrow{\chi}_i, r_i))$$

Component-Based Synthesis, Overall

- We conjoin the well-formedness and functionality constraints into one big formula
- We have an SMT solver solve that formula
- The result is a witness, assigning integer values to each location variable
  - We can then convert the witness into a program
  - Line i of the program:

$$z_i = f_j(z_{\sigma_1}, \ldots, z_{\sigma_\eta}) \text{ when } l_{r_j} == i \text{ and } \bigwedge_{k=1}^{\eta} (l_{\chi_j^k} == \sigma_k)$$

- We can then put this into a CEGIS loop: