CS 314 Principles of Programming Languages

Lecture 25: A Peek at Program Synthesis

Prof. Zheng Zhang

Rutgers University

December 7, 2018

Class Information

- Project 3 deadline 12/12.
- Homework 8 deadline 12/10.
- Final exam coverage:
  Lectures 1 - 21, hw 1-8, recitation (all), and corresponding book chapters.
- Final exam: 12/19 4pm — 7pm.
- Next class: Final exam Q & A.

Program Synthesis

- **Find a program P that meets a spec ϕ(input,output):**

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

Program Synthesis

- **Find a program P that meets a spec ϕ(input,output):**

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

- **Example**

spec:

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

$\phi(x, y)$: $y=$ **foo**(x)

partial program:

```
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;
}
```

solution found!

Synthesis as search over candidate programs

- Partial program (sketch) defines a candidate space

  we search this space for a program that meets ϕ

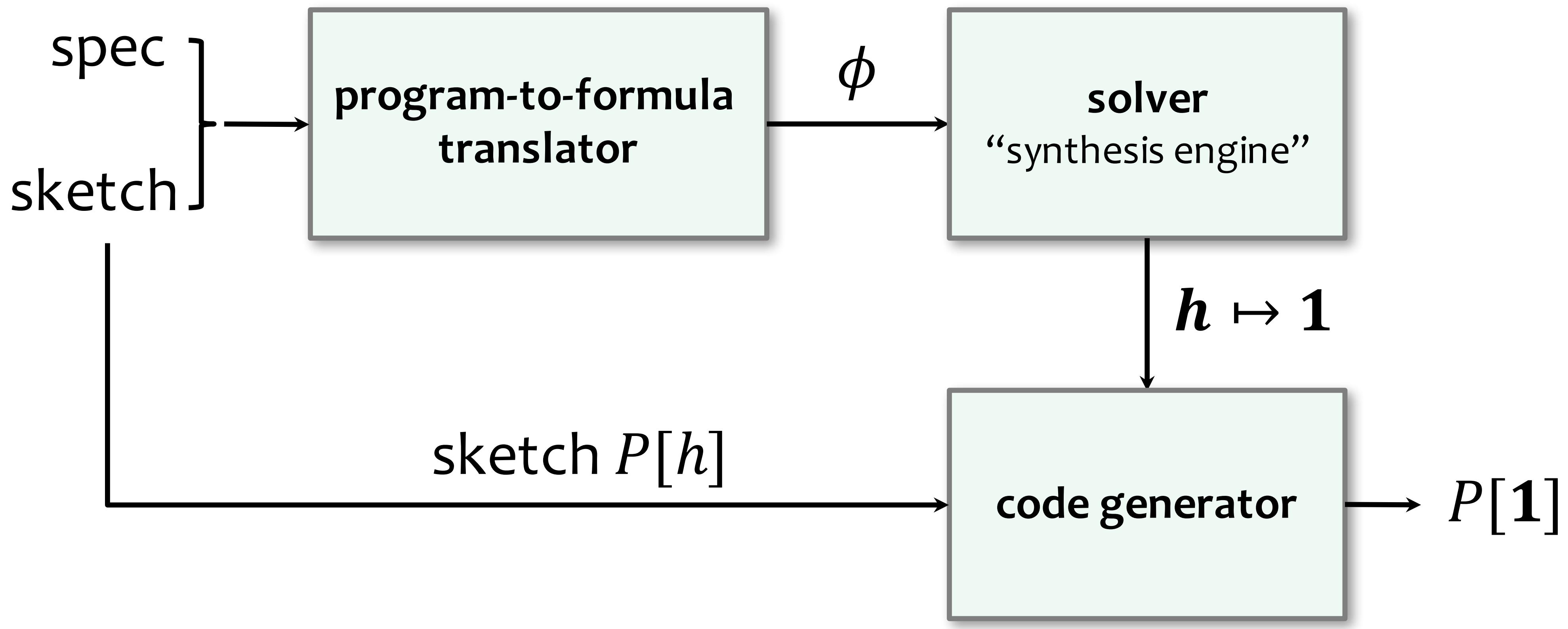
- Usually can't search this space by enumeration

  space too large ($\gg 10^{10}$)

- Describe the space symbolically

  solution to constraints encoded in a logical formula gives values of holes, indirectly identifying a correct program

Synthesis for Partial Programs



Program As a Formula

- Assume a formula $S_P(x,y)$ which holds iff program P(x) outputs value y

program: f(x) { return x + x }

formula: $S_f(x,y): y = x+x$

Program As a Formula

- Solver as an interpreter: given x, evaluate f(x)

  $S(x,y) \wedge x = 3$ solve for y $\boldsymbol{y \mapsto 6}$

- Solver as a program inverter: given f(x), find x

  $S(x,y) \wedge y = 6$ solve for x $\boldsymbol{x \mapsto 3}$

Search over a space of candidate solutions

- $S_P(x,h,y)$ holds iff sketch $P\,[h](x)$ outputs y.

```
spec(x) { return x + x }
sketch(x) { return x << ?? }
```

$S_{sketch}\_x,y,h\_ : y = x * 2^h$

- The solver computes h, thus synthesizing a program correct for the given x (here, x=2)

$S_{sketch}(x,y,h) \wedge x=2 \wedge y=4$ solve for h $\boldsymbol{h} \mapsto \mathbf{1}$

- Sometimes h must be constrained on several inputs

$S(x_1, y_1, h) \wedge x_1=0 \wedge y_1=0 \wedge$

$S(x_2, y_2, h) \wedge x_2=3 \wedge y_2=6$ solve for h $\boldsymbol{h} \mapsto \mathbf{1}$

Modeled as a constraint solving problem

Inductive Synthesis

- Definition of **inductive synthesis**:

  Ask for a program P correct on a few inputs.

  We hope (or test, verify) that P is correct on rest of inputs.

- Segment on Synthesis Algorithm will describe how to select suitable inputs

What can be synthesized?

- Networking stack

  ==> TCP protocol is a program ==> synthesize protocols
- Interpreter

  ==> embeds language semantics ==> languages may be synthesizable
- Spam filter

  ==> classifiers ==> learning of classifiers is synthesis
- Image gallery

  ==> compression algorithms or implementations
- OS scheduler

  ==> scheduling policy
- Parallelization

  ==> affine transformation parameters
- and … you name it!

Constraint Solving Problem

- **Find a program P that meets a spec ϕ(input,output):**

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

From ϕ to pre- and post-conditions:

- A precondition (denoted *pre(x)*) of a procedure *f* is a predicate (Boolean-valued function) over *f*'s parameters *x* that always holds when *f* is called.

  f can <u>assume</u> that *pre* holds

- A postcondition (*post(x,y)*) is a predicate over parameters of *f* and its return value *y* that holds when f returns

  f <u>ensures</u> that *post* holds

Background: Satisfiability Solvers

- A satisfiability solver accepts a formula $\phi(x, y, z)$ and checks if ϕ is satisfiable (SAT).

- If yes, the solver returns a model m, a valuation of x, y, z that satisfies ϕ, ie, m makes ϕ true.

- If the formula is unsatisfiable (UNSAT), some solvers return minimal unsat core of ϕ, a smallest set of clauses of ϕ that cannot be satisfied.

Example: $(x_2 \vee \neg x_{41} \vee x_{15}) \wedge (x_6 \vee \neg x_2) \wedge (x_{31} \vee \neg x_{41} \vee \neg x_6 \vee x_{156})$

SAT v.s. SMT Solvers

- SAT solvers accept propositional Boolean formulas
  typically in Conjunctive Normal Form form

- SMT (satisfiability modulo theories) solvers accept formulas in richer logics, eg uninterpreted functions, linear arithmetic, theory of arrays

- Z3 Solver (developed by Microsoft Research):
  https://rise4fun.com/z3/tutorial

Code Checking

- Correctness condition ϕ says that the program is correct for all valid inputs:

$$\forall x \;.\; pre(x) \Rightarrow SP(x,y) \wedge post(x,y)$$

- How to prove correctness for all inputs x?
  Search for *counterexample* x where ϕ does not hold.

$$\exists x \;.\; \neg(pre(x) \Rightarrow SP(x,y) \wedge post(x,y))$$

Verification Condition

- Some simplifications:

  $\exists x\ .\ \neg(pre(x)) \Rightarrow SP(x,y) \wedge post(x,y))$

  $\exists x\ .\ pre(x \wedge)\ \neg(S_P(x,y) \wedge post(x,y))$

- S_p always holds (we can always find y given x since S_P encodes program execution), so the verification formula is:

  $\exists x\ .\ pre(x) \wedge S_P(x,y) \wedge \neg post(x,y)$

Verification Example

- Triangle Classifier

Rosette (extension of the scheme lang)

```
(define (classify a b c)
  (if (and (>= a b) (>= b c))
      (if (or (= a c) (= b c))
          (if (and (= a b) (= a c))
              'EQUILATERAL
              'ISOSCELES)
          (if (not (= (* a a) (+ (* b b) (* c c))))
              (if  (< (* a a) (+ (* b b) (* c c)))
                   'ACUTE
                   ‘OBTUSE)
              'RIGHT))
      'ILLEGAL))
```

- This classifier contains a bug.
  We will solve it using constraint solver Z3.

Specification for Classify

- *pre*(*a*,*b*,*c*):

  $a,b,c > 0 \wedge a<b+c$

- *post*(*a*, *b*, *c*, *y*):

  where *y* is return value from classify(a,b,c)

  we'll specify *post* functionally, with a correct implementation of **classify**.

Verification Formula for Z3

```
; precondition: triangle sides must be positive and
; must observe the triangular inequality

(define-fun pre ((a Int)(b Int)(c Int)) Bool
   (and  (> a 0)
         (> b 0)
         (> c 0)
         (< a (+ b c) ) ) )

; our postcondition is based on a debugged version of classify

(define-fun spec ((a Int)(b Int)(c Int)) TriangleType
          … ; a correct implementation comes here
)

(define-fun post ((a Int)(b Int)(c Int)(y TriangleType)) Bool
        (= y (spec a b c)))
```

Continued

```
; the verification condition

(declare-const x Int)
(declare-const y Int)
(declare-const z Int)

(assert ( and (pre x y z)
              (not (post x y z (classify x y z)))))

(check-sat)
(get-model)
```

Output from the Z3 solver

- Model of verification formula = counterexample input

```
sat
(model
  (define-fun z () Int 1)
  (define-fun y () Int 2)
  (define-fun x () Int 2)
)
```

- This counterexample input *refutes* correctness of classify

Verification Example

- Triangle Classifier

Rosette (extension of the scheme lang)

```
(define (classify a b c)
 (if (and (>= a b) (>= b c))
   (if (or (= a c) (= b c))
     (if (and (= a b) (= a c))
       'EQUILATERAL
       'ISOSCELES)
     (if (not (= (* a a) (+ (* b b) (* c c))))
       (if  (< (* a a) (+ (* b b) (* c c)))
          'ACUTE
         ‘OBTUSE)
       'RIGHT))
   'ILLEGAL))
```

Counter Example:
2, 2, 1

- This classifier contains a bug.
  We will solve it using constraint solver Z3.

Let's Correct the Classifier Bug with Synthesis

- We ask the synthesizer to replace the buggy expression, (or (= a c))(= b c), with a suitable expression from this grammar

```
hole --> e and e | e or e
e    --> var op var
var  --> a | b | c
op   --> = | <= | < | > | >=
```

…

- We want to write a partial program (sketch) that syntactically looks roughly as follows:

```
(define (classify a b c)
  (if (and (>= a b) (>= b c))
    (if (hole) ; this used to be (or (= a c))(= b c)
      (if (and (= a b) (= a c))
        ….
```

The Sketch in Z3

- First we define the “elementary” holes.

These are the values computed by the solver.

- These elementary holes determine which expression we will derive from the grammar (see next slides):

```
(declare-const h0 Int)
(declare-const h1 Int)
(declare-const h2 Int)
(declare-const h3 Int)
(declare-const h4 Int)
(declare-const h5 Int)
(declare-const h6 Int)
```

Encoding the "Hole" Grammar

- The call to function hole expands into an expression determined by the values of h0, …, h6, which are under solver's control.

```
(define-fun hole( (a Int) (b Int) (c Int) )  Bool
  (synth-connective h0
    (synth-comparator h1
          (synth-var h2 a b c)
          (synth-var h3 a b c))
    (synth-comparator h4
          (synth-var h5 a b c)
          (synth-var h6 a b c))))

(define-fun synth-var ( (h Int) (a Int) (b Int)(c Int)) Int
          (if (= h 0)
               a
             (if (= h 1) b c)))

(define-fun synth-connective ((h Int)(v1 Bool) (v2 Bool)) Bool
          (if (= h 0)
                (and v1 v2)
                (or v1 v2) ) )
```

Replace the Buggy Assertion with the Hole

- The hole expands to an expression from the grammar that will make the program correct (if one exists).

- The expression is over variables a, b, c, hence the arguments to the call to hole.

```
(define-fun classify ((a Int)(b Int)(c Int)  TriangleType
  (if (and (>= a b) (>= b c))
    (if (hole a b c)
       (if (and (= a b) (= a c))
              ……
```

The Synthesis Formula

- The partial program is now translated to a formula.
  Q: how many parameters does the formula have?
  A: h0, …, h6, a, b, c, (and, technically, also the return value)
- We are now ready to formulate the synthesis formula to be solved. It suffices to add i/o pair constraints:

```
(assert (= (classify  2 12 27) ILLEGAL))
(assert (= (classify  5  4  3) RIGHT))
(assert (= (classify 26 14 14) ISOSCELES))
(assert (= (classify 19 19 19) EQUILATERAL))
(assert (= (classify 9  6   4) OBTUSE))
... ; we have 8 input/output pairs in total
```

The Result of Synthesis

- These i/o pairs sufficed to obtain a program correct on all inputs. The program

  h0 -> 1
  h1 -> 0
  h2 -> 0
  h3 -> 1
  h4 -> 0
  h5 -> 1
  h6 ->2

- which means the hole is

  (or (= a b)(= b c))

Verification Example

- Triangle Classifier

Rosette (extension of the scheme lang)

```
(define (classify a b c)
  (if (and (>= a b) (>= b c))
    (if (  or (= a c) (= b c)   )
      (if (and (= a b) (= a c))
        'EQUILATERAL
        'ISOSCELES)
      (if (not (= (* a a) (+ (* b b) (* c c))))
        (if  (< (* a a) (+ (* b b) (* c c)))
           'ACUTE
           ‘OBTUSE)
        'RIGHT))
    'ILLEGAL))
```

or (= a b)(= b c)

Counter Example:
2, 2, 1

- This classifier contains a bug.
  Now we fix the bug!

Reading

- A lot of slides are adapted from the talk “Synthesizing Programs with Constraint Solvers” by Ras Bodik and Emina Torlak in the Computer Aided Verification (CAV) 2012 Tutorial.

SMT Solver (Z3)

- **https://rise4fun.com/z3/tutorial**