CS 292C Computer-Aided Reasoning for Software

Lecture 2: Solver-Aided Programming I

Inspired by CSE507 from Emina Torlak

Yu Feng Fall 2019

Summary of previous lecture

- Introducing the cast
- Ideas about final project
- Course structure

Ideas for final projects

- Detect vulnerable smart contracts without specs (MSR)
- Verify complex properties (LTL) in smart contracts (MSR)
- Detect malware in third-party libraries for Android (Google security)
- Interactive data visualization synthesis (UW, UT)
- Synthesize DoS attacks for Solidity compilers (Solidity)
- Super-optimization via reinforcement learning (Google Brain)
- Type-directed synthesis for polymorphism
- •

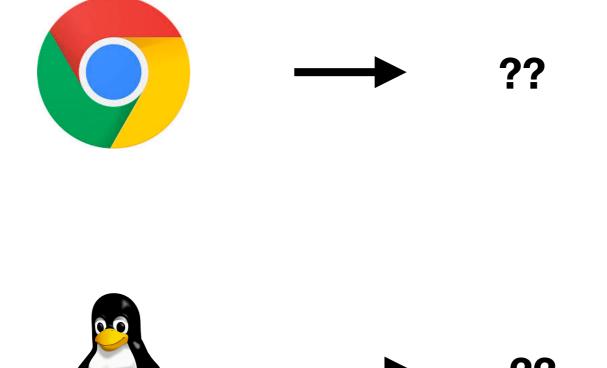
Outline of this lecture

- The classical way for using solvers
- Solver-aided programming
- Rosette constructs

A classical way to use solvers

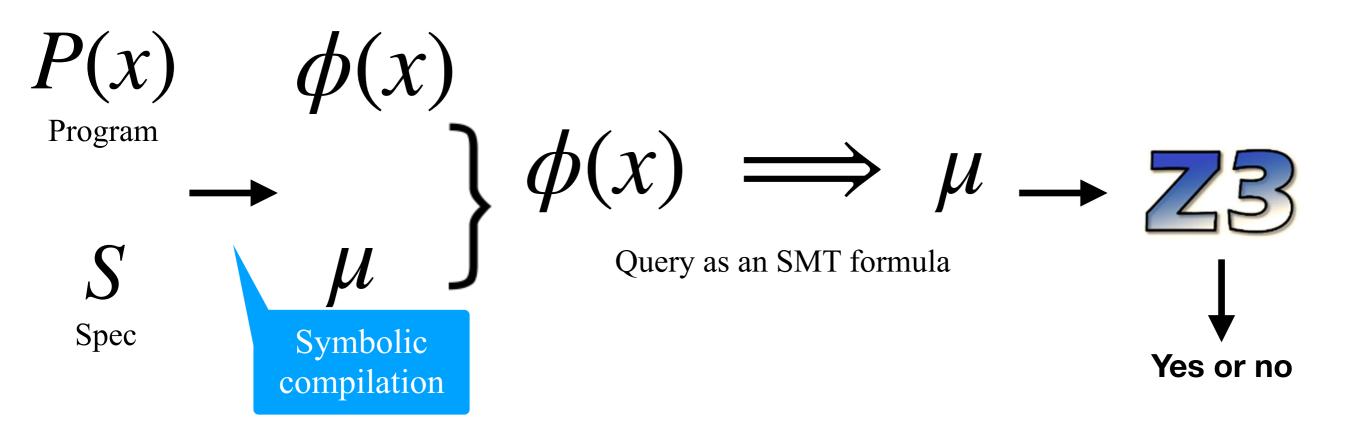
```
foo (int a) {
x = 10;
                               x = 10 \land y = 5
 y = 5;
foo (int a) {
if (a > 0)
 x = 10;
                    a > 0 \implies x = 10 \land a < = 0 \implies y = 5
 else
  y = 5;
foo (int a) {
 if (a > 0)
 x = 10;
                    a > 0 \implies x = 10 \land a < = 0 \implies y = 5
 else
 y = 5;
                     \implies y > 4
 assert y > 4
}
```

A classical way to use solvers



How to deal with complex systems?

A classical way to use solvers



Symbolic compilation can take years of effort!

A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and debugging.

Solver-aided programming

```
p(x) {
    v = 12

p(x) {
    v = ??
    ...
}
assert safe(x, p(x))
```

Find an input on which the program fails.

Localize bad parts of the program.

Find values that repair the failing run.

Find code that repairs the program.

Solver-aided applications

Systems

SOSP'19, OSDI'18, SOSP'17, OSDI'16

Blockchain

Browser engines

Biology

POPL'14

Education

Data science

Robotics

PLDI'18, PLDI'17

HPC

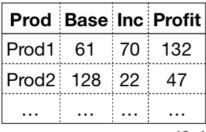
Gaming

ASPLOS'16, OSDI'18

Malware NDSS'17

Visualization

Solver-aided applications



13x4

```
Product
                         Prod11
value 20
                         Prod6
  10
     20
               60
         Profit %
    Line(Prod6, 45, 11, 45, 25)
   Line(Prod11,55,25,55,28)
```

point(Prod6,45,11) point(Prod11,55,25)

```
MultiLaver(
   Line(x=Profit,y=value,
        color=Product).
   Scatter(x=Profit,y=value,
       color=Product))
P1:
t=mutate(T,Top,Base+Inc)
t=gather(t,key=[Prod,Profit],
         value=[Base,Top])
t=select(t,[Profit,value,Product])
P2:
t=gather(T,[Prod,Profit],[Base])
t=select(t,[Profit,value,Product])
```

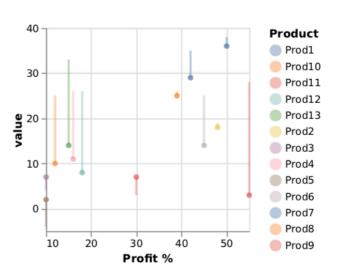
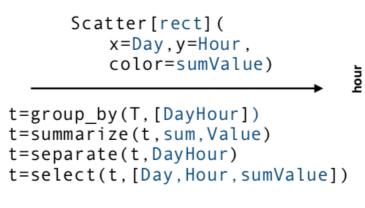


Fig. 16. Illustration of task #2.

DayHour	Site	Туре	Value
06_00	e90e50	PC	22
06_00	e90e50fx	PC	8
06_01	e90e50	Mobile	8
			1986x4

point(06,00,155) point(28,21,42) point(28,22,60) point(29,01,8)



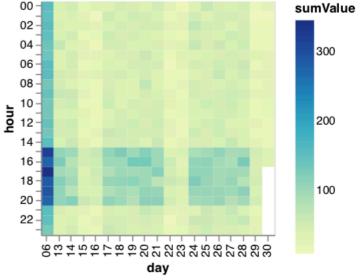
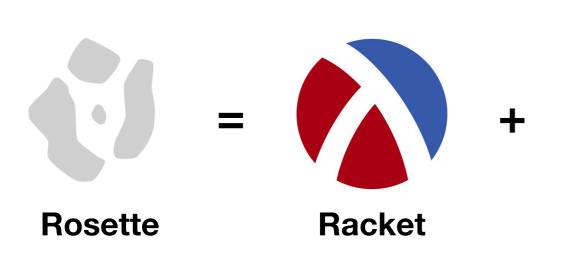


Fig. 17. Illustration of task #3.

Rosette constructs



```
(define-symbolic id type)
(define-symbolic* id type)

(assert expr)

(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
  #:forall expr
  #:guarantee expr)
```

symbolic values

assertions

queries

Rosette constructs: verify

Search for a binding of symbolic constants

#:forall expr

#:guarantee expr)

```
to concrete values that violates at least one
                                        (define (poly x))
            of the assertions
                                         (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))
                                        (define (factored x)
                                         (* x (+ x 1) (+ x 2) (+ x 2)))
(de ine-symbolic id type)
                              symbolic
(detine-symbolic* id type)
                                        (define (same p f x)
                              values
                                         (assert (= (p x) (f x)))
(assert expr)
                              assertions
(verify expr)
                                        (define-symbolic i integer?)
(debug [type ...+] expr)
                              queries
(solve expr)
                                        (define cex (verify (same poly factored i)))
(synthesize
                                        (evaluate i cex)
```

Rosette constructs: debugging

```
(define (poly x))
 Searches for a minimal set of expressions
                                         (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))
that are responsible for the observed failure
                                        (define (factored x)
                                         (* (+ x (??)) (+ x 1) (+ x (??)) (+ x (??)))
(de ine-symbolic id type)
                                        (define (same p f x)
                              symbolic
(de ine-symbolic* id type)
                                         (assert (= (p x) (f x)))
                              values
(astert expr)
                              assertions
                                        (define-symbolic i integer?)
(verify expr)
(debug [type ...+] expr)
                              queries
(solve expr)
                                        (define binding
(synthesize
                                            (synthesize #:forall (list i)
 #:forall expr
                                                          #:guarantee (same poly factored i)
 #:guarantee expr)
```

Rosette constructs: synthesis

```
(define (poly x))
Search for a binding of symbolic constants
                                         (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))
to concrete values that satisfy the assertions
                                        (define (factored x)
                                         (* (+ x (??)) (+ x 1) (+ x (??)) (+ x (??)))
(de ne-symbolic id type)
                                        (define (same p f x)
                              symbolic
(detine-symbolic* id type)
                                         (assert (= (p x) (f x)))
                              values
(assert expr)
                              assertions
                                        (define-symbolic i integer?)
(verify expr)
(debug [type ...+] expr)
                              queries
(solve expr)
                                        (define binding
(synthesize
                                            (synthesize #:forall (list i)
 #:forall expr
                                                          #:guarantee (same poly factored i)
 #:guarantee expr)
```

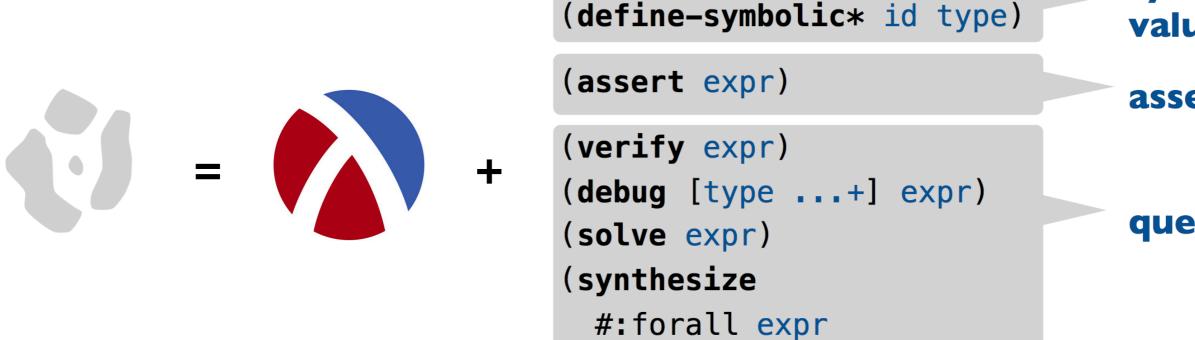
Rosette constructs: angelic execution

```
Searches for a minimal set of expressions
that are responsible for the observed failure
                                         (define-symbolic x y integer?)
                                         (define (poly x))
                                          (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))
(de ine-symbolic id type)
                              symbolic
(de ine-symbolic* id type)
                                         (define sol
                              values
                                              (solve (begin (assert (not (= x y)))
(astert expr)
                              assertions
                                                              (assert (< (abs x) 10))
(verify expr)
                                                              (assert (< (abs y) 10))
(debug [type ...+] expr)
                                                              (assert (not (= (poly x) \emptyset)))
                              queries
(solve expr)
                                                              (assert (= (poly x) (poly y)))))
(synthesize
 #:forall expr
 #:guarantee expr)
```

Rosette constructs: angelic execution

(define-symbolic id type)

#:guarantee expr)



symbolic

queries

TODOs by next lecture

- Paper reading list is out
- Install Rosette and Neo
 - Install Rosette: https://docs.racket-lang.org/rosette-guide/ch_getting-started.html
 - Install Neo: https://github.com/fredfeng/Trinity
- Start to look for partners for your final project!