

**CS 292C Computer-Aided Reasoning for Software**

# Lecture 2: Solver-Aided Programming I

Inspired by CSE507 from Emina Torlak

Yu Feng  
Fall 2020

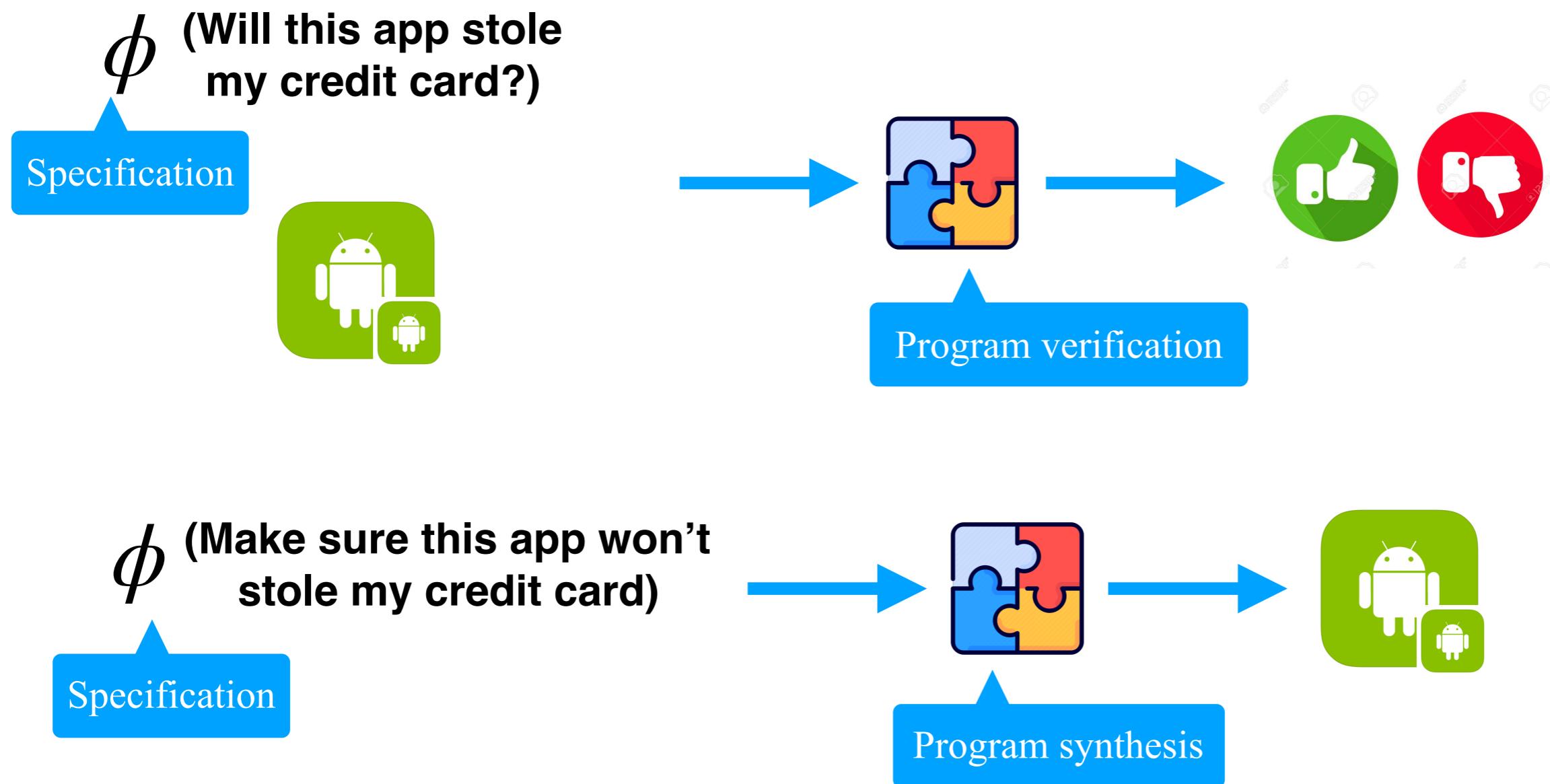
# Summary of previous lecture

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

# Example projects

- Attack synthesis for safety-critical compilers
- Synthesize smart contracts from Temporal Specs
- Data visualization from natural languages
- Exploit generation for Android apps
- Verifying fairness in smart contract/robotics
- Vulnerabilities repairing using “big-code”
- ...

# Goals of this course



# Goals of this course



Open the  
blackbox

Satisfiability Modulo Theories  
Solver (SMT solver)

# Outline of this lecture

- The classical way for using SMT solvers
- Solver-aided programming
- Racket programming language in 10-mins
- The Rosette framework

# A classical way to use solvers

```
foo () {  
    x = 10;  
    y = 5;  
}
```

$$x = 10 \wedge y = 5$$

```
foo (int a) {  
    if (a > 0)  
        x = 10;  
    else  
        y = 5;  
}
```

$$a > 0 \implies x = 10 \wedge a \leq 0 \implies y = 5$$

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

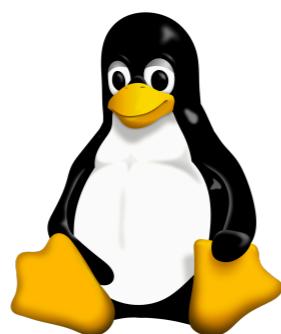
$$(a > 0 \implies x = 10 \wedge a \leq 0 \implies y = 5)$$

$$\implies y > 4$$

# A classical way to use solvers



??

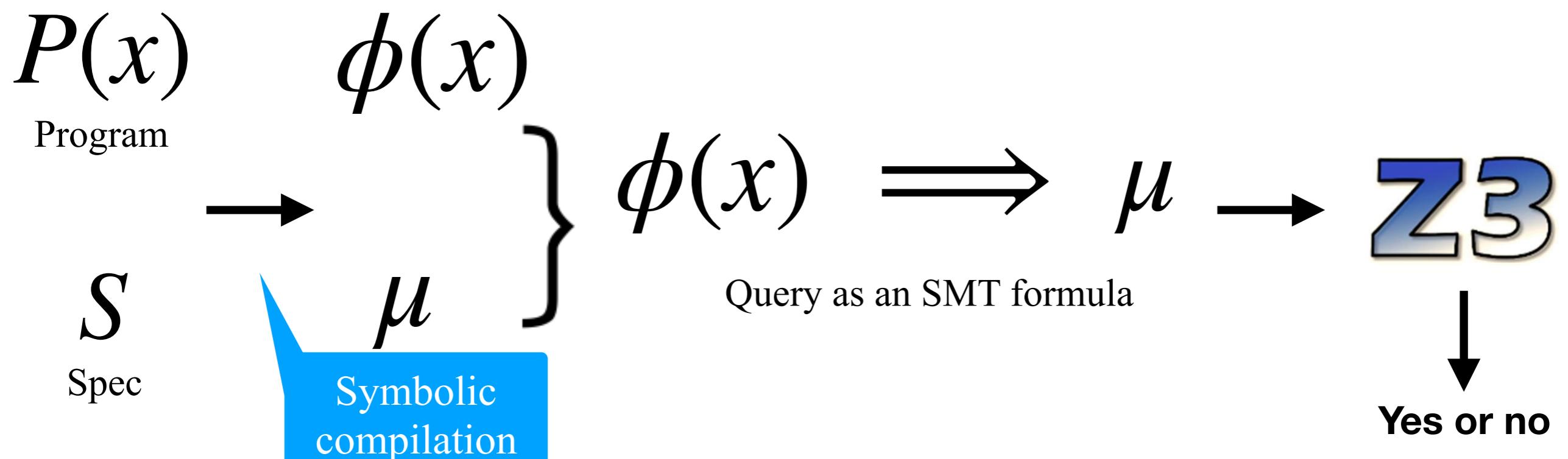


??

How to deal with complex systems?

It is undecidable!

# A classical way to use solvers



**Symbolic compilation can take years of effort!**

<https://github.com/Z3Prover/z3>

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, OSDI'20

## Blockchain

ASE'20



## Browser engines

PPoPP'13



## Biology

POPL'14

## Education

## Data science

PLDI'18, PLDI'17



## Healthcare

CAV'16



## HPC

ASPLOS'16, OSDI'18

## Gaming

## Malware

NDSS'17



## Visualization

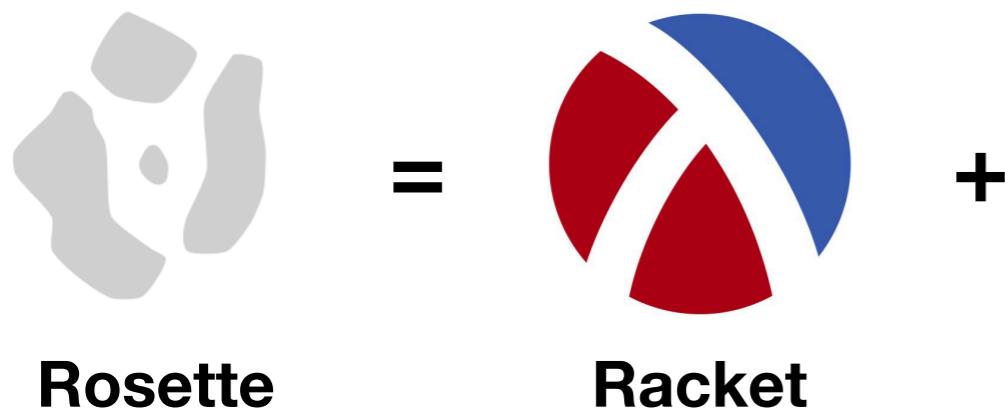
POPL'20



# Racket in 10-mins



# 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  
to concrete values that violates at least one  
of the assertions

```
(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**

$f(x) = x^4 + 6x^3 + 11x^2 + 6x$   
`(define (poly x) (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))`

$g(x) = x * (x + 1) * (x + 2) * (x + 2)$   
`(define (factored x) (* x (+ x 1) (+ x 2) (+ x 2)))`

`(define (same p f x)
 (assert (= (p x) (f x))))`

`(define-symbolic i integer?)`

`(define cex (verify (same poly factored i)))
 (evaluate i cex))`

# Rosette constructs: debugging

Searches for a minimal set of expressions  
that are responsible for the observed failure

```
(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**

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))
(define (factored x)
  (* (+ x (??)) (+ x 1) (+ x (??)) (+ x (??))))
(define (same p f x)
  (assert (= (p x) (f x))))
(define-symbolic i integer?)
(define binding
  (synthesize #:forall (list i)
    #:guarantee (same poly factored i)))
```

# Rosette constructs: synthesis

Search for a binding of symbolic constants  
to concrete values that satisfy the assertions

```
(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**

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))

(define (factored x)
  (* (+ x (??)) (+ x 1) (+ x (??)) (+ x (??)))

(define (same p f x)
  (assert (= (p x) (f x)))

(define-symbolic i integer?)

(define binding
  (synthesize #:forall (list i)
    #:guarantee (same poly factored i))
```

# Rosette constructs: angelic execution

Searches for a minimal set of expressions  
that are responsible for the observed failure

```
(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**

```
(define-symbolic x y integer?)
```

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))
```

```
(define sol
  (solve (begin (assert (not (= x y)))
                (assert (< (abs x) 10))
                (assert (< (abs y) 10))
                (assert (not (= (poly x) 0))))
                (assert (= (poly x) (poly y)))))
```

# 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](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!