Lecture 16: Solver-Aided Programming

Inspired by CSE507 from Emina Torlak

Yu Feng
Winter 2021

Outline of this lecture

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

SAT

Boolean Satisfiability

Boolean Formula

1. Variables $x_1 \; x_2 \; x_3 \begin{cases} \text{"true"} / 1 \\ \text{"false"} / 0 \end{cases}$
2. "not" $\overline{x_1}$ x_2

NP-complete Problems

Stephen Cook 1971 SAT

Richard Karp 1972 3SAT

Independent Set

Hamiltonian Cycle

Vertex Cover

CLIQUE

Traveling-Salesman Problem(TSP)

……

Subset-Sum

……

About 1000 NP-complete problems have been discovered since.

A classical way to use solvers

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

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

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

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

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

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

$$\implies y > 4$$

A classical way to use solvers

→ ??

→ ??

How to deal with complex systems?

A classical way to use solvers

$P(x)$

Program

S

Spec

$$\longrightarrow \quad \left.\begin{array}{c}\phi(x)\\ \\ \mu\end{array}\right\} \; \phi(x) \Longrightarrow \mu \quad \longrightarrow \quad \text{Z3}$$

Query as an SMT formula

Yes or no

Symbolic compilation: convert a program into formula

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

ASE'20-a, ASE'20-b

Browser engines

Biology

POPL'14

Education

Data science

PLDI'18, PLDI'17

Robotics

HPC

ASPLOS'16, OSDI'18

Gaming

Malware

NDSS'17

Visualization

POPL'20

Rosette constructs

Rosette = Racket +

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

symbolic values

```
(assert expr)
```

assertions

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

queries

A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. Emina Torlak and Rastislav Bodik. PLDI'14

Rosette constructs:symbolic values

A type that is efficiently supported by SMT solvers: booleans, integers, reals, bitvectors, uninterpreted functions.

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

symbolic values

```
(assert expr)
```

assertions

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

queries

define-symbolic creates a fresh symbolic constant of the given type and binds it to the variable id.

```
> (define (same-x)
   (define-symbolic x integer?)
  x)

> (same-x)
  x

> (eq? (same-x) (same-x))
  #t
```

id is bound to the same constant every time **define-symbolic** is evaluated.

Symbolic values of a given type can be used just like concrete values of that type.

Rosette constructs:symbolic values

A type that is efficiently supported by SMT solvers: booleans, integers, reals, bitvectors, uninterpreted functions.

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

symbolic values

```
(assert expr)
```

assertions

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

queries

define-symbolic* creates a fresh symbolic constant of the given type and binds it to the variable id.

```
> (define (new-x)
   (define-symbolic* x integer?)
  x)

> (new-x)
  x$0

> (eq? (new-x) (new-x))
  (= x$3 x$4)
```

id is bound to a **different** constant every time **define-symbolic*** is evaluated..

Symbolic values of a given type can be used just like concrete values of that type.

Rosette constructs: assert

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

```
(assert expr)
```

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

assert checks that expr evaluates to a true value.

```
> (assert (>= 2 1)) ; passes

> (assert (< 2 1)) ; fails
  assert: failed
```

symbolic values

```
> (define-symbolic* x integer?)
```

assertions

```
> (assert (>= x 1))
```

queries

```
>

  (list (<= 1 x$0) ...)
```

Symbolic expr gets added to the assertion store. Its meaning (true or false) is eventually determined by the solver in response to queries.

From assert to verify

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

symbolic values

```
(assert expr)
```

assertions

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

queries

Do poly and factored produce the same output on all inputs?

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

(define (factored x)
 (* x (+ x 1) (+ x 2) (+ x 2)))

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

; some tests ...
> (same poly fact 0) ; pass
> (same poly fact -1) ; pass
> (same poly fact -2) ; pass
```

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)
```

symbolic values

```
(assert expr)
```

assertions

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

queries

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

(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)
```

symbolic values

```
(assert expr)
```

assertions

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

queries

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

(define/debug (fact x)
    (* x (+ x 1) (+ x 2) (+ x 2)))

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

(render ; visualize the result
   (debug [integer?] (same poly fact -6))))
```

To use debug, require the debugging libraries, mark fact as the candidate for debugging, save the module to a file, and issue a debug query.

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)
```

symbolic values

```
(assert expr)
```

assertions

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

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 (??))))
```

Unknown is represented as ??

```
(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)))
```

To generate code, require the sketching library, save the module to a file, and issue a synthesize query.

Thank you for taking CS162!