#20: Program Sketching and CEGIS - II

Sankha Narayan Guria

EECS 700: Introduction to Program Synthesis

Symbolic execution

1. Semantics of a simple imperative language
2. How to use it for symbolic execution?
3. Adding while loops
4. Adding holes

Semantics of a simple language

```
e  :=  n | x | e₁ + e₂
c  :=  x := e | assert e
     | c₁ ; c₂ | if e then c₁ else c₂ | while e do c
```

- What does an expression mean?
  - An expression reads the state and produces a value
  - The state is modeled as a map σ from variables to values
  - $\mathcal{A}[\![\cdot]\!] : e \to \Sigma \to \mathbb{Z}$
- Ex:
  - $\mathcal{A}[\![x]\!] = \lambda\sigma.\,\sigma[x]$
  - $\mathcal{A}[\![n]\!] = \lambda\sigma.\,n$
  - $\mathcal{A}[\![e_1 + e_2]\!] = \lambda\sigma.\,\mathcal{A}[\![e_1]\!]\sigma + \mathcal{A}[\![e_2]\!]\sigma$

Semantics of a simple language

```
e  :=  n | x | e₁ + e₂
c  :=  x := e | assert e
     | c₁ ; c₂ | if e then c₁ else c₂ | while e do c
```

- What does a command mean?
  - A command modifies the state
  - $\mathcal{C}[\![\cdot]\!] : c \to \Sigma \to \Sigma$
- Ex:
  - $\mathcal{C}[\![x \coloneqq e]\!] = \lambda\sigma.\,\sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$
  - $\mathcal{C}[\![c_1; c_2]\!] = \lambda\sigma.\,\mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma)$
  - $\mathcal{C}[\![\text{if } e \text{ then } c_1 \text{ else } c_2]\!] = \lambda\sigma.\,\mathcal{A}[\![e]\!]\sigma \neq 0 \;\;?\;\; \mathcal{C}[\![c_1]\!]\sigma \;:\; \mathcal{C}[\![c_2]\!]\sigma$

Semantics of assertions

```
e  :=  n | x | e₁ + e₂
c  :=  x := e | assert e
     | c₁ ; c₂ | if e then c₁ else c₂ | while e do c
```

- What does a command mean?
  - Commands also generate constraints on valid executions
  - $\mathcal{C}[\![\cdot]\!] : c \to \langle\Sigma, \Psi\rangle \to \langle\Sigma, \Psi\rangle$

Constraints on values in initial σ

- Ex:
  - $\mathcal{C}[\![assert\ e]\!] = \lambda\langle\sigma,\psi\rangle.\langle\sigma,\psi \wedge \mathcal{A}[\![e]\!]\sigma \neq 0\rangle$

Symbolic execution

1. Semantics of a simple imperative language
2. How to use it for symbolic execution?
3. Adding while loops
4. Adding holes

Concrete execution: example 1

Let's run this with x = 2

```
void main(int x){
  int y = 2 * x;
  assert y > x;
}
```

$\sigma = \{x \to 2\}, \quad \psi = \top$

$\sigma = \{x \to 2, y \to 4\}, \psi = \top$

$\sigma = \{x \to 2, y \to 4\}, \psi = \{4 > 2\}$

Test passed

Symbolic execution: example 1

```
void main(int x){
  int y = 2 * x;
  assert y > x;
}
```

$\sigma = \{x \to X\}, \psi = \top$

$\sigma = \{x \to X, y \to 2X\}$

$\psi = \{\, 2X > X\}$

$$\mathcal{C}[\![p]\!]\langle\{\}, \top\rangle = \langle\{x \to X, y \to 2X\}, 2X > X\rangle$$

Verification constraint

$\forall X.\, 2X > X$

SMT solver

$\{X \mapsto 0\}$

Symbolic execution: example 2

```
  void main(int x, int u){
→   int y = 0;
→   if (u > 0) {
      y = 2 * x;
→   } else {
      y = x + x;
→   }
→   assert y == 2*x;
  }
```

$\sigma = \{x \to X, u \to U\}$

$\sigma = \{x \to X, u \to U, y \to 0\}$

$\sigma = \{x \to X, u \to U, y \to 2X\}$

$\sigma = \{x \to X, u \to U, y \to X + X\}$

$\sigma = \{x \to X, u \to U, y \to U > 0 \,?\, 2X : X + X\}$

$\psi = \{(U > 0 \,?\, 2X : X + X) = 2X\}$ ✔

Symbolic execution

1. Semantics of a simple imperative language
2. How to use it for symbolic execution?
3. Adding while loops
4. Adding holes

What about loops?

- Semantics of a while loop
  - Let $W: \Sigma \to \Sigma = \mathcal{C}[\![while\ e\ do\ c]\!]$
  - W satisfies the following equation:
$$W\ \sigma \quad = \quad \mathcal{A}[\![e]\!]\sigma \neq 0 \quad ? \quad W(\mathcal{C}[\![c]\!]\sigma)\ :\ \sigma$$
  - One strategy: find a fixpoint (see later in class)
  - We'll settle for a simpler strategy: unroll k times and then give up

Symbolic execution: example 3

```
void main(int x){
  int y = 0;
  int i = 0;
  while (i < 2) {
    y = y + x;
    i = i + 1;
  }
  assert y == i * x;
}
```

Step 1: unroll with depth = 2

```
if (i < 2) {
  y = y + x;
  i = i + 1;
  if (i < 2) {
    y = y + x;
    i = i + 1;
    assert !(i < 2);
  }
}
```

Symbolic execution: example 3

```
void main(int x){
  int y = 0;
  int i = 0;
  if (i < 2) {
    y = y + x;
    i = i + 1;
    if (i < 2) {
      y = y + x;
      i = i + 1;
      assert !(i < 2);
    }
  }
  assert y == i*x;
}
```

$\sigma = \{x \to X\}$

$\sigma = \{x \to X, y \to 0, i \to 0\}$

$\sigma = \{x \to X, y \to X, i \to 1\}$

Simplified from $0 < 2\ ?\ (1 < 2\ ?\ X + X : X) : 0$

$\sigma = \{x \to X, y \to$ … $\to 2\}$

$\psi = \{\neg(2 > 2)\}$

$\sigma = \{x \to X, y \to X + X, i \to 2\}$

$\psi = \{\neg(2 > 2) \;\wedge\; X + X = 2X\}$

Symbolic execution

1. Semantics of a simple imperative language
2. How to use it for symbolic execution?
3. Adding while loops
4. Adding holes

Semantics of sketches

```
e  :=  n | x | e₁ + e₂ | ??ᵢ
c  :=  x := e | assert e
     | c₁ ; c₂ | if e then c₁ else c₂ | while e do c
```

- What does an expression mean?
  - Like before, but with a “hole environment” ϕ
  - $\mathcal{A}[\![\cdot]\!] : e \to \mathbf{\Phi} \to \Sigma \to \mathbb{Z}$
- Ex:
  - $\mathcal{A}[\![x]\!] = \lambda\phi.\lambda\sigma.\sigma[x]$
  - $\mathcal{A}[\![??_i]\!] = \lambda\phi.\lambda\sigma.\phi[i]$
  - $\mathcal{A}[\![e_1 + e_2]\!] = \lambda\phi.\lambda\sigma.\mathcal{A}[\![e_1]\!]\phi\sigma + \mathcal{A}[\![e_2]\!]\phi\sigma$

Symbolic Evaluation of Commands

- Commands have two roles
  - Modify the symbolic state
  - Generate constraints

$$\mathcal{C}[\![\cdot]\!] : c \to \Phi \to \langle \Sigma, \Psi \rangle \to (\Sigma, \Psi)$$

Symbolic Evaluation of Commands

- Example: assignment and assertion

$$\mathcal{C}[\![x := e]\!]\phi\langle\sigma, \psi\rangle = \langle\sigma[x \mapsto \mathcal{A}[\![e]\!]\phi\sigma], \psi\rangle$$

$$\mathcal{C}[\![\textbf{assert}\ e]\!]\phi\langle\sigma, \psi\rangle = \langle\sigma, \psi \wedge \mathcal{A}[\![e]\!]\phi\sigma \neq 0\rangle$$

Symbolic execution of sketches: example

```
void main(int x){
  int z = ??1 * x;
  int y = 0;
  int i = 0;
  if (i < 2) {
    y = y + x;
    i = i + 1;
    if (i < 2) {
      y = y + x;
      i = i + 1;
      assert !(i < 2);
    }
  }
  assert y == z;
}
```

$\sigma = \{x \to X\}$ $\quad \psi = \top$

$\sigma = \{x \to X, z \to \phi_1 * X, y \to 0, i \to 0\}$

$\sigma = \{x \to X, z \to \phi_1 * X, y \to X, i \to 1\}$

$\sigma = \{x \to X, z \to \phi_1 * X, y \to X + X, i \to 2\}$

$\psi = \{\neg(2 > 2)\}$

$\psi = \{\neg(2 > 2) \;\wedge\; X + X = \phi_1 * X\}$

$$\exists \phi_1 . \forall X . X + X = \phi_1 * X$$

CEGIS

$$\{\phi_1 \mapsto 2\}$$

Controls for generators

```
harness void main(int x, int y){
   z = mono(x)    +    mono(y);
→  assert z == x + x + 3;
}
```

$\sigma = \{z \to (\phi_1 \,?\, \phi_2 : X * \phi_2) + (\phi_1 \,?\, \phi_2 : Y * \phi_2)\}$

No solution!

```
generator int mono(int x) {
    if (??1) {return ??2;}
    else {return x * mono(x);}
}
```

unroll with depth = 1

```
if (??1) {return ??2;}
else {return x * ??2;}
```

- We need to map different calls to `mono` to different controls!

Controls for generators: context

```
harness void main(int x, int y){
  z = mono¹(x,1) +  mono²(y,2);
  assert z == x + x + 3;
}

generator int mono(int x, context τ) {
    if (??τ₁) {return ??τ₂;}
    else {return x * mono³(x, τ.3);}
}
```

$$\sigma = \{z \rightarrow (\phi_1^1 \,?\, \phi_2^1 : X * \phi_2^{1.3}) + (\phi_1^2 \,?\, \phi_2^2 : X * \phi_2^{2.3})\}$$

$$\{\phi_1^1 \mapsto 0, \phi_2^{1.3} \mapsto 2, \phi_1^2 \mapsto 1, \phi_2^{1.3} \mapsto 3\}$$