Lecture 10
Unbounded Verification

Nadia Polikarpova
(some material from Peter Müller, ETH Zurich)

Logistics

Projects

- I expect you to meet with me at least once before the presentation to discuss the progress
- Book a date through the spreadsheet
  - During Office Hours
  - Two teams per week

Last week

Sketch

- Programming language + constant holes + generators + assertions
- Can use generator libraries to build a "synthesis DSL"

Rosette: takes it a step further

- Racket (metaprogramming!) + symbolic variables + solver queries
- Can define full-fledged SDSLs (Solver-aided DSLs)

Map of the unit

Constraint-based synthesis

- How to solve constraints about infinitely many inputs? CEGIS
- How to encode semantics of looping / recursive programs?
    - Bounded reasoning
→ - Unbounded / deductive reasoning

Enumerative (and deductive) synthesis

- How to use deductive reasoning to guide the search?

Constraint-based synthesis from specifications

Behavioral constraints
= assertions, reference
implementation, pre/post

Structural constraints

encoding

$$\exists c \, . \, \forall x \, . \, Q(c, x)$$

Why is this hard?

```
gcd (int a, int b) returns (int x)
    requires  a > 0 ∧ b > 0
    ensures   a % x = 0 ∧ b % x = 0
              ∀c . 0 < c < x  ⇒  a % c ≠ 0  ∨  b % c ≠ 0
{
    int x , y := a, b;
    while (x != y) {
        if (x > y) x := ??*x + ??*y;
        else y := ??*x + ??*y;
}}
```

infinitely many inputs

cannot validate by testing

infinitely many paths!

hard to generate constraints

Last week

To encode a program as constraints, need to understand what it does

- $\mathcal{A}[\![\cdot]\!] : e \to \Sigma \to \mathbb{Z}$
- $\mathcal{C}[\![\cdot]\!] : c \to \Sigma \to \Sigma$

This is called *denotational semantic*: map a program to a function

- Hard for loops and recursion: need to find a function that satisfies a fixpoint equation
- We had to resort to bounded unrolling ☹

What's wrong with unrolling?

```
gcd (int a, int b) returns (int x)
   requires  a > 0 ∧ b > 0
   ensures   a % x = 0 ∧ b % x = 0
```
$$\forall c \ . \ 0 < c < x \ \Rightarrow \ a \% c \neq 0 \ \lor \ b \% c \neq 0$$
```
{
   int x , y := a, b;
   while (x != y) {
      if (x > y) x := ??*x + ??*y;
      else y := ??*x + ??*y;
}}
```

Unroll with
depth = 1

```
if (x != y) {
   if (x > y)
      x := ??*x + ??*y;
   else
      y := ??*x + ??*y;
   assert !(x != y);
}
```

Either unsatisfiable
sketch or unsound
solution!

Constraint-based synthesis from specifications

Behavioral constraints
= assertions, reference implementation, pre/post

Structural constraints

encoding

$$\exists c \, . \, \forall x \, . \, Q(c, x)$$

If we want to synthesize programs that are correct on all inputs, we need a better way to deal with loops!

Axiomatic semantics

Instead of asking "What does this program do?", ask "Does this property hold?"

- $\mathcal{A}[\![\cdot]\!] : e \rightarrow \Sigma \rightarrow \mathbb{Z}$
- ~~$\mathcal{C}[\![\cdot]\!] : c \rightarrow \Sigma \rightarrow \Sigma$~~
- $\{P\} \, c \, \{Q\}$ ("if P holds before executing c, then Q holds after")

Axiomatic semantics = program logic

- A set of rules for proving *judgments* about programs

Hoare logic = a program logic for simple imperative programs

The Imp language

```
e ::= n | x |
      e + e | e - e | e * e |
      e = e | e < e | !e | e && e
c ::= skip            |
      x := e          |
      c ; c           |
      if e then c else c |
      while e do c
```

Hoare triples

Properties of programs are specified as judgments

$$\{P\}\ c\ \{Q\}$$

where c is a command and $P, Q: \sigma \rightarrow Bool$ are predicates

- e.g. if $\sigma = [x \mapsto 2]$ and $P \equiv x > 0$ then $P\ \sigma = \top$

Terminology

- Judgments if this kind are called *(Hoare) triples*
- P is called precondition
- Q is called postcondition

Meaning of triples

The meaning of $\{P\}\, c\, \{Q\}$ is:

- **if** P holds in the initial state σ, and
- **if** the execution of c from σ terminates in a state σ'
- **then** Q holds in σ'

This interpretation is called *partial correctness*

- termination is not essential

Another possible interpretation: *total correctness*

- **if** P holds in the initial state σ
- **then** the execution of c from σ terminates in a state (call it σ')
- **and** Q holds in σ'

Example: swap

$\{\top\}$

    x := x + y; y := x - y; x := x - y

~~$\{x = y \wedge y = x\}$~~

We have to express that **y** in the final state is equal to **x** in the initial state!

Logical variables

$\{x = N \wedge y = M\}$

```
x := x + y; y := x - y; x := x - y
```

$\{x = M \wedge y = N\}$

Assertions can contain *logical variables*
- may occur only in pre- and postconditions, not in programs
- the states maps logical variables to their values, just like normal variables

Inference system

We formalize the semantics of a language by describing which judgments are valid about a program

An *inference system*

- a set of *axioms* and *inference rules* that describe how to derive a valid judgment

We combine axioms and inference rules to build *inference trees* (derivations)

Semantics of skip

`skip` does not modify the state

$$\{\, P \,\}\, \texttt{skip}\, \{\, P \,\}$$

Semantics of assignment

x := e assigns the value of e to variable x

$$\{\, P[x \mapsto e]\, \}\quad \text{x} := \text{e}\quad \{\, P\, \}$$

- Let σ be the initial state
- Precondition: $(P[x \mapsto e])\sigma$, i.e., $P(\sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma])$
- Final state: $\sigma' = \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$
- Consequently, P holds in the final state

Semantics of composition

Sequential composition **c1 ; c2** executes **c1** to produce an intermediate state and from there executes **c2**

$$\frac{\{P\}\, c_1\, \{R\} \qquad \{R\}\, c_2\, \{Q\}}{\{P\}\, c_1; c_2\, \{Q\}}$$

Example: swap

inference tree

leaves = axioms

$$\text{assign} \frac{}{\{x = N + M \wedge y = N\} \ \text{x := x - y} \ \{x = M \wedge y = N\}}$$

$$\text{assign} \frac{}{\{x = N + M \wedge y = M\} \ \text{y := x - y} \ \{x = N + M \wedge y = N\}}$$

edges = rules

$$\text{comp} \frac{}{\{x = N + M \wedge y = M\} \ \text{y := x - y; x := x - y} \ \{x = M \wedge y = N\}}$$

$$\text{assign} \frac{}{\{x = N \wedge y = M\} \ \text{x := x + y} \ \{x = N + M \wedge y = M\}}$$

$$\text{comp} \frac{}{\{x = N \wedge y = M\} \ \text{x := x + y; y := x - y; x := x - y} \ \{x = M \wedge y = N\}}$$

root = triple to prove

Proof outline

An alternative (more compact) representation of inference trees

$\{x = N \wedge y = M\}$

$\quad \Rightarrow$

$\{(x + y) - ((x + y) - y) = M \wedge (x + y) - y = N\}$

```
x = x + y;
```

$\{x - (x - y) = M \wedge x - y = N\}$

```
y = x - y;
```

$\{x - y = M \wedge y = N\}$

```
x = x - y
```

$\{x = M \wedge y = N\}$

Rule of consequence

$$\frac{\{P'\}\ c\ \{Q'\}}{\{P\}\ c\ \{Q\}}\ \text{ if }\ P \Rightarrow P' \wedge Q' \Rightarrow Q$$

Corresponds to adding $\Rightarrow$ steps in a proof outline

Here $R \Rightarrow S$ should be read as

- "We can prove for all states σ, that $R\ \sigma$ implies $S\ \sigma$"

Semantics of conditionals

$$\frac{\{P \wedge \mathcal{A}[\![e]\!]\} \; c_1 \; \{Q\} \qquad \{P \wedge \neg \mathcal{A}[\![e]\!]\} \; c_2 \; \{Q\}}{\{P\} \; \text{if } e \text{ then } c_1 \text{ else } c_2 \; \{Q\}}$$
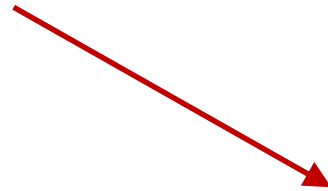
Example: absolute value

{⊤}

```
if x < 0 then
    x := -x
else
    skip
```

{x ≥ 0}

Semantics of loops

loop invariant

$$\frac{\{I \wedge \mathcal{A}[\![e]\!]\} \; c \; \{I\}}{\{I\} \; \texttt{while} \; e \; \texttt{do} \; c \; \{\neg \mathcal{A}[\![e]\!] \wedge I\}}$$

Example: GCD

$\{x = N \land y = M \land N > 0 \land M > 0\}$
$\Rightarrow$
$\{I\}$

```
  while x != y do
```
$\{I \land x \neq y\}$
```
    if x > y then
       x := x - y
    else
       y := y - x
```
$\{I\}$
$\{I \land x = y\}$
$\Rightarrow$
$\{x = \gcd(N, M)\}$

Guessing the loop invariant:

x	y
10	4
6	4
2	4
2	2

$$I \equiv \gcd(x, y) = \gcd(N, M)$$

Example: GCD

$$\{x = N \wedge y = M \wedge N > 0 \; \wedge M > 0\}$$
$$\Rightarrow$$
$$\{\gcd(x, y) = \gcd(N, M) \wedge x, y > 0\}$$

```
while x != y do
```
$$\{\gcd(x, y) = \gcd(N, M) \wedge x, y > 0 \wedge \; x \neq y\}$$
```
    if x > y then
```
$$\{\gcd(x, y) = \gcd(N, M) \wedge \; x \neq y \wedge x > y\}$$
$$\Rightarrow$$
$$\{\gcd(x - y, y) = \gcd(N, M) \wedge x - y, y > 0\}$$
```
        x := x - y
```
$$\{\gcd(x, y) = \gcd(N, M) \wedge x, y > 0\}$$
```
    else
        y := y - x
```
$$\{\gcd(x, y) = \gcd(N, M) \wedge x, y > 0\}$$
$$\{\gcd(x, y) = \gcd(N, M) \wedge x, y > 0 \wedge \; x = y\}$$
$$\Rightarrow$$
$$\{x = \gcd(N, M)\}$$

Termination

loop variant / ranking function / termination metric

$$\frac{\{I \wedge \mathcal{A}[\![e]\!] \wedge r = R\} \; c \; \{I \wedge r < R \wedge r \geq 0\}}{\{I\} \; \texttt{while} \; e \; \texttt{do} \; c \; \{\neg \mathcal{A}[\![e]\!] \wedge I\}}$$

Example: GCD

```
while x != y do

    if x > y then
        x := x - y
    else
        y := y - x
```

Example: GCD

$\{x = N \land y = M \land N > 0 \land M > 0\}$

$\quad \Rightarrow$

$\{\gcd(x, y) = \gcd(N, M) \land x, y > 0\}$

```
    while x != y do
```

$\qquad \{\gcd(x, y) = \gcd(N, M) \land x, y > 0 \land x + y = R \land x \neq y\}$

```
        if x > y then
            x := x - y
        else
            y := y - x
```

$\qquad \{\gcd(x, y) = \gcd(N, M) \land x, y > 0 \land x + y < R \land x + y \geq 0\}$

$\{\gcd(x, y) = \gcd(N, M) \land x, y > 0 \land x = y\}$

$\quad \Rightarrow$

$\{x = \gcd(N, M)\}$

Program verifiers

Dafny demo

https://rise4fun.com/Dafny/snKV