Introduction to Formal Methods
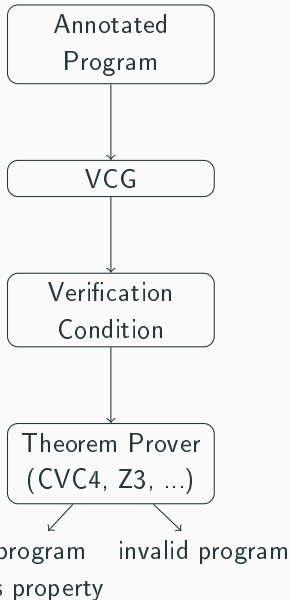
Lecture 5

From Programs to Formulas

Hossein Hojjat & Fatemeh Ghassemi

October 7, 2018

Verification-Condition Generation
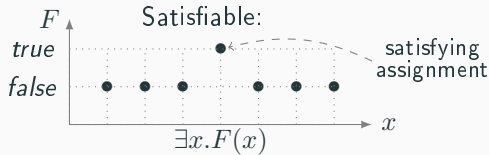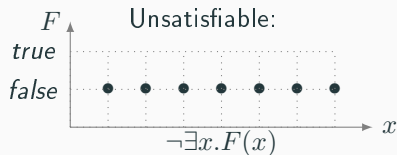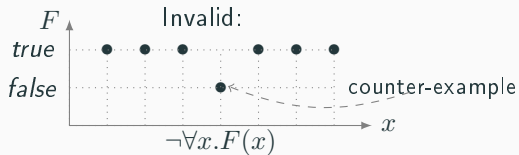

Steps in Verification

- Generate formula whose validity implies correctness of program
- Attempt to prove formula
  - If formula is **valid**, program is correct
  - If formula has a **counterexample**, it indicates one of these:
    - error in the program
    - error in the property
    - error in auxiliary statements (e.g. loop invariants)

Terminology

- Generated formulas:
    verification conditions
- Generation process:
    verification-condition generation
- Program that generates formulas:
    Verification-Condition Generator(VCG)$_1$

Validity and Satisfiability

$F(x)$: formula with free variable(s) x

General Situation:

Valid: $\forall x.F(x)$

Invalid: $\neg\forall x.F(x)$ — counter-example

Unsatisfiable: $\neg\exists x.F(x)$

Satisfiable: $\exists x.F(x)$ — satisfying assignment
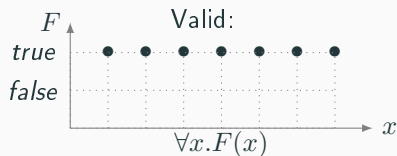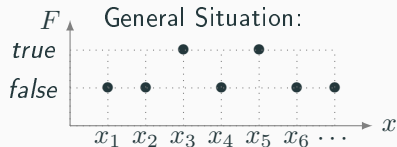
F is valid $\Leftrightarrow \overline{F}$ is unsatisfiable

F is invalid $\Leftrightarrow \overline{F}$ is satisfiable

2

We examine algorithms for going from programs to their verification
conditions

```
if (x > 0)
        res = x * 2 + 1;
else
        res = 24;
assert(res > 0);
```

For the following formula, we check validity:
all variables are universally quantified

$$\Big(\big((x > 0) \land (res = 2x + 1)\big) \lor \big(\neg(x > 0) \land (res = 24)\big)\Big) \to (res > 0)$$

```
x = T
if (F) c1 else c2
c1 ; c2        ← ordinary control structure
while (F) c1
```

$$c \quad ::= x = T \mid (\text{if } (F) \ c \ \text{else } c) \mid c; c \mid (\text{while } (F) \ c)$$
$$T \quad ::= K \mid V \mid (T+T) \mid (T-T) \mid (K*T) \mid (T/K) \mid (T\%K)$$
$$F \quad ::= (T == T) \mid (T < T) \mid (T > T) \mid (\ F) \mid (F\&\&F) \mid (F||F)$$
$$V \quad ::= x \mid y \mid z \mid \cdots$$
$$K \quad ::= 0 \mid 1 \mid 2 \mid \cdots$$

terms like in Integer arithmetic

Boolean formulas without quantifiers

Prove this program always terminates for any natural number x:

```
while (x > 1) {
  if (x % 2 == 0) x = x / 2;
    else x = 3*x+1;
}
```

"Mathematics is not yet ripe for such problems"   -Paul Erdös

- This language is Turing-complete
- Every possible program (Turing machine) can be encoded into computation on integers (computed integers can become very large)
- Problem of taking a program and checking whether it terminates is undecidable
- **Rice**'s Theorem: all properties of programs that are expressed in terms of the results that the programs compute (and not in terms of the structure of programs) are undecidable

In real programming languages we have bounded integers, but we have other sources of unboundedness, e.g.

- `BigInt` data type of Java and Scala
  (sequence of digits of any length)
- example: sizes of linked lists and of other data structure
- Program syntax trees for an interpreter or compiler
  (we would like to handle programs of any size)

- Checking satisfiability of Presburger arithmetic formulas (even with quantifiers) is decidable
- Checking if there exists an input to a program in our language for which program computes a given value (e.g. 1) is undecidable
- Quantifiers in Presburger arithmetic cannot be used to define `z=x*y`
    - But we can write a program that computes `x*z` and stores it in `z`
- Programs without loops can be translated into Presburger arithmetic
- Loops give much more expressive power to Presburger arithmetic than quantifiers
  (situation can be different if we did not work with Presburger arithmetic)

VC Generation for Programs

- Program can be represented by a formula relating initial and final state
- Consider program with variables `x`, `y`, `z`

program: `x = x + 2; y = x + 12`
relation: $\{(x, y, z, x', y', z') \mid x' = x + 2 \land y' = x' + 12 \land z' = z\}$
formula: $x' = x + 2 \land y' = x' + 12 \land z' = z$

Examples

Relation between initial and all possible final states

```
x = x + 3;
x = x + 2;
```
$\{(x, x') \mid x' = x + 5\}$

```
x = x + x;
```
$\{(x, x') \mid x' = 2x\}$

```
while (x != 10) {
   x = x + 1;
}
```
$\{(x, x') \mid x \leq 10 \land x' = 10\}$

```
while (5 == 5) {
   x = x;
}
```
$\emptyset$

The meaning is, in general, an arbitrary relation. Therefore:

- For certain states there will be no results
    - In particular, if a computation starting at a state does not terminate
- For certain states there will be multiple results
    - This means execution starting in a state will sometimes compute one and sometimes other result
    - Verification of such program must account for both possibilities
- Multiple results are important for modeling e.g. concurrency, as well as approximating behavior that we do not know
  (e.g. what the operating system or environment will do, or what the result of complex computation is)

```
x = randomInteger ()
if (x > 10) {
 y = y+1
} else {
 y = y+2
}
```

- Relation between the initial and the final y:

$$\{(y, y') \mid (y' = y + 1 \lor y' = y + 2)\} =$$

$$\{\cdots, (100, 101), (100, 102), (101, 102), (101, 103), \cdots\}$$

- obviously, not a function

- Cartesian product: $A \times B = \{(x, y) \mid x \in A \wedge y \in B\}$
- Relation $r \subseteq A \times B$
- Diagonal relation: $\Delta_A = \{(x, x) | x \in A\}$
- Partial function $f : A \hookrightarrow B$

$$\forall x \in A, y_1 \in B, y_2 \in B.(x, y_1) \in f \wedge (x, y_2) \in f \rightarrow y_1 = y_2$$

- Partial function is total iff

$$\forall x \in A.\exists y \in B.(x, y) \in r$$

- Function $f : A \rightarrow B$ when f is partial function and total on $A \times B$

$$
\begin{aligned}
\mathsf{dom}(r) &= \quad \{x \mid \exists y.(x,y) \in r\} && \text{domain} \\
\mathsf{ran}(r) &= \quad \{y \mid \exists x.(x,y) \in r\} && \text{range}
\end{aligned}
$$

- $f : A \hookrightarrow B$, $g :\hookrightarrow B$

$$
f \oplus g = \Big\{(x,y) \mid \big((x,y) \in f \wedge x \notin \mathsf{dom}(g)\big) \vee (x,y) \in g\Big\}
$$

- $f[x := v]$ means $f \oplus \{(x,v)\}$

$$
(f[x := v])(y) = \begin{cases} v & \text{if } y = x \\ f(y) & \text{if } y \neq x \end{cases}
$$

Relation Composition: $\quad t \circ r \quad = \quad \{(x, z) \mid \exists y.(x, y) \in t \wedge (y, z) \in r\}$

Relation Image: $\qquad\quad S \bullet r \quad = \quad \{y \mid \exists x \in S.(x, y) \in r\}$

Theorem. For $r \subseteq A \times A$ and $S \subseteq A$

$$S \bullet r = \mathsf{ran}(\Delta_S \circ r)$$

$$r \subseteq A^2$$
$$r^0 = \Delta_A$$
$$r^1 = r \circ \Delta_A = r$$
$$r^{n+1} = r \circ r^n = r^n \circ r$$
$$r* = \bigcup_{i \geq 0} r^i = \Delta_A \cup r \cup r^2 \cup \cdots$$

Theorem.

$$\bigcap \{S \mid \Delta_A \cup S \circ r \subseteq S\} = r*$$

($r*$ is the least S satisfying the recursive condition)

Guarded Command Language

assume(F)   block all executions where F does not hold

s1 ; s2   do first s1, then s2

s1 [] s2   do either s1 or s2 arbitrarily

s*   execute s zero, once, or more times

- c imperative command
- $R(c)$ formula describing relation between initial and final states of execution of c
- If $\rho(c)$ describes the relation, then $R(c)$ is formula such that

$$\rho(c) = \{(\vec{v}, \vec{v'}) \mid R(c)\}$$

- $R(c)$ is a formula between unprimed variables $\vec{v}$ and primed variables $\vec{v'}$

$$x = t$$

$$x = t$$

$R(x = t)$:

$$x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v = v'$$

- Note that the formula must explicitly state which variables remain the same (here: all except x)
- Otherwise, those variables would not be constrained by the relation, so they could take arbitrary value in the state after the command

$$\texttt{assume}(F)$$

$$\texttt{assume}(F)$$

$R(\texttt{assume}(F))$:

$$F \wedge \bigwedge_{v \in V} v = v'$$

$$c_1 \; [] \; c_2$$

$$c_1 \, [\!] \, c_2$$

$R(c_1 \, [\!] \, c_2)$:

$$R(c_1) \vee R(c_2)$$

- translation is simply a disjunction - this is why construct is interesting
- corresponds to branching in control-flow graphs

$$c_1; c_2$$

Reminder about relation composition and its definition:

$$r_1 \circ r_2 = \{(a, c) \mid \exists b.(a, b) \in r_1 \land (b, c) \in r_2\}$$

What are $R(c_1)$ and $R(c_2)$ and in terms of which variables they are expressed?

$$c_1; c_2$$

Reminder about relation composition and its definition:

$$r_1 \circ r_2 = \{(a, c) \mid \exists b.(a, b) \in r_1 \wedge (b, c) \in r_2\}$$

What are $R(c_1)$ and $R(c_2)$ and in terms of which variables they are expressed?

$R(c_1; c_2)$:

$$\exists \vec{z}.R(c_1)[\vec{x'} := \vec{z}] \wedge R(c_2)[\vec{x} := \vec{z}]$$

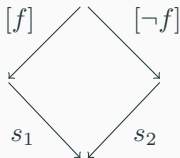where $\vec{z}$ are freshly picked names of intermediate states

```
if (F)
   s1
     else
      s2
```

$(\mathrm{assume}(F); s1)$

$[]$

$(\mathrm{assume}(\neg F); s2)$

$[f]$ $[\neg f]$

s_1 s_2

Example: Absolute Value

```
if (x > 0)
    y = x;
else
    y = −x;
```

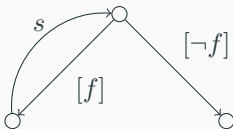- Assignments and assume statements generate equalities, many of which can be eliminated by one-point rule

$$(\exists x.x = t \land F) \leftrightarrow F[x := t]$$

- There are more complex quantifier elimination procedures that can be used in principle as well

```
while (F)
  s
```

$(\text{assume}(F); s) * ;$

$(\text{assume}(\neg F))$

```
int x = 0;
while (x < 2)
        x = x + 1;
```

$$
\left(
\begin{array}{l}
(x = 0) \lor \\
(x = 0 \land x < 2 \land x_1 = x + 1) \lor \\
(x = 0 \land x < 2 \land x_1 = x + 1 \land x_1 < 2 \land x_2 = x_1 + 1) \lor \\
(x = 0 \land x < 2 \land x_1 = x + 1 \land x_1 < 2 \land x_2 = x_1 + 1 \land x_2 < 2 \land x_3 = x_2 + 1) \lor \\
\quad \vdots \qquad \vdots \qquad \vdots \qquad \vdots
\end{array}
\right)
$$

$;$

$x \geq 2$

```
int x = 0;
while (x < 2)
        x = x + 1;
```

$$\left(\begin{array}{l} (x = 0) \vee \\ (x = 0 \wedge x < 2 \wedge x_1 = x + 1) \vee \\ (x = 0 \wedge x < 2 \wedge x_1 = x + 1 \wedge x_1 < 2 \wedge x_2 = x_1 + 1) \vee \\ (x = 0 \wedge x < 2 \wedge x_1 = x + 1 \wedge x_1 < 2 \wedge x_2 = x_1 + 1 \wedge x_2 < 2 \wedge x_3 = x_2 + 1) \vee \\ \vdots \qquad \vdots \qquad \vdots \qquad \vdots \end{array} \right)$$

$$; \\ x \geq 2$$

```
    int x = 0;
    while (x < 2)
            x = x + 1;
```

$$\left(\begin{array}{l} (x = 0) \vee \\ (x = 0 \wedge x < 2 \wedge x_1 = x + 1) \vee \\ (x = 0 \wedge x < 2 \wedge x_1 = x + 1 \wedge x_1 < 2 \wedge x_2 = x_1 + 1) \\ \\ \\ \end{array}\right)$$

;

$x \geq 2$

```
int x = 0;
while (x < 2)
        x = x + 1;
```

$$
\begin{pmatrix}
(x = 0 \land x \geq 2) \lor \\
(x = 0 \land x < 2 \land x_1 = x + 1 \land x_1 \geq 2) \lor \\
(x = 0 \land x < 2 \land x_1 = x + 1 \land x_1 < 2 \land x_2 = x_1 + 1 \land x_2 \geq 2)
\end{pmatrix}
$$

```
int x = 0;
while (x < 2)
        x = x + 1;
```

$$
\left(
\begin{array}{l}
\cancel{(x = 0 \land x \geq 2)} \lor \\
\cancel{(x = 0 \land x < 2 \land x_1 = x + 1 \land x_1 \geq 2)} \lor \\
(x = 0 \land x < 2 \land x_1 = x + 1 \land x_1 < 2 \land x_2 = x_1 + 1 \land x_2 \geq 2)
\end{array}
\right)
$$

```
int x = 0;
while (x < 2)
          x = x + 1;
```

$$\left(\begin{array}{l} (x = 0 \land x \geq 2) \lor \\ (x = 0 \land x < 2 \land x_1 = x + 1 \land x_1 \geq 2) \lor \\ (x = 0 \land x < 2 \land x_1 = x + 1 \land x_1 < 2 \land x_2 = x_1 + 1 \land x_2 \geq 2) \\ \\ \\ x' = 2 \end{array} \right)$$

- Change a given variable arbitrarily

$$R(\mathrm{havoc}(x)) = \{(x, y, z, x', y', z') \mid y' = y \wedge z' = z\}$$

- We can prove that the following equality holds when `x` does not occur in `E`

  `x = E`   is   `havoc(x); assume(x=E)`

- In other words, assigning a variable is the same as changing it arbitrarily and then assuming that it has the right value