CS615: Formal Specification and Verification of Programs 2019

Lecture 1: Program modeling and semantics

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2019-07-31

Programs

Our life depends on programs

- airplanes fly by wire
- autonomous vehicles
- flipkart,amazon, etc
- QR-code - our food

Programs have to work in hostile conditions

- NSA
- Heartbleed bug in SSH
- 737Max is falling from the sky
- ... etc.

Verification

- Much needed technology
- Undecidable problem
- Many fragments are hard
- Open theoretical questions
- Difficult to implement algorithms
  - the field is full of start-ups

Perfect field for a young bright mind to take a plunge

Topic 1.1

Course contents

The course

A course is not sufficient to cover the full breath of verification but we will try

First half (core)

1. Program semantics
2. Supporting technology
3. Theory of abstraction
4. Two methods: abstract interpretation and model checking

Lecture plan for the first half

CS 719: Foundation of formal methods

Lecture 1-4:
Program Modeling
Semantics
Symbolic methods

Lecture 5-7:
Decision procedures
SAT/SMT Solving
Quantifier elimination

Lecture 8-10:
Theory of abstraction

Program → Analysis/Verification → Output

Analysis/Verification

Lecture 11-13:
Abstract interpretation

Lecture 14-16:
Model Checking

Lecture 17: Tools

The course (contd.)

Second half (more stuff)

1. Program with features: functions, pointers, time, etc — 6 lectures
   - Exploiting structures of programs
2. Beyond safety — 2 lectures
   - liveness and security
3. Practical verification — 2 lectures
   - "limited" guarantees
4. Latest in verification — 2 lectures
   - Aspects of learning
   - Synthesis

This part is adaptive and depends on your interest.
Please give active **feedback**.

Logic in verification

Differential equations
are the calculus of
Electrical engineering

Logic
is the calculus of
Computer science

Logic provides tools to define/manipulate computational objects

Applications of logic in Verification

- **Defining Semantics:** Logic allows us to assign "mathematical meaning" to programs

$$P$$

- **Defining properties:** Logic provides a language of describing the "mathematically-precise" intended behaviors of the programs

$$F$$

- **Proving properties:** Logic provides algorithms that allow us to prove the following mathematical theorem.

$$P \models F$$

The rest of the lecture is about making sense of "$\models$"

Logical toolbox

We need several logical operations to implement verification methods.

Let us go over some of those.

Logical toolbox : satisfiablity

$$s \models F?$$

Example 1.1

$$\underbrace{\{x \mapsto 1, y \mapsto 2\}}_{model} \models \underbrace{x + y = 3}_{formula}.$$

Exercise 1.1

- $\{x \rightarrow 1\} \models x > 0$?
- $\{x \rightarrow 1, y \rightarrow 2\} \models x + y = 3 \wedge x > 0$?
- $\{x \rightarrow 1, y \rightarrow 2\} \models x + y = 3 \wedge x > 0 \wedge y > 10$?

Exercise 1.2

Can we say something more about the last formula?

Logical toolbox : satisfiablity

Is there any model?

$$\models F?$$

Harder problem!

Exercise 1.3

- $\models x + y = 3 \land x > 0?$
- $\models x + y = 3 \land x > 0 \land y > 10?$
- $\models x > 0 \lor x < 1?$

disjunction

Exercise 1.4

Can we say something more about the last formula?

Logical toolbox : validity

Is the formula true for all models?

$$\forall s : s \models F?$$

Even harder problem?

We can simply check satisfiability of $\neg F$.

Example 1.2

$x > 0 \lor x < 1$ *is valid because* $x \leq 0 \land x \geq 1$ *is unsatisfiable.*

Logical toolbox : implication

$$F \Rightarrow G?$$

We need to check $F \Rightarrow G$ is a valid formula.
We check if $\neg(F \Rightarrow G)$ is unsatisfiable, which is equivalent to checking if $F \wedge \neg G$ is unsatisfiable.

Example 1.3

Consider the following implication

$$x = y + 1 \wedge y \geq z + 3 \Rightarrow x \geq z$$

After negating the implication, we obtain $x = y + 1 \wedge y \geq z + 3 \wedge x < z$.

After simplification, we obtain $x - z \geq 4 \wedge x - z < 0$.

Therefore, the negation is unsatisfiable and the implication is valid.

Logical toolbox : quantifier elimination

$$\text{given } F\text{, find } G \text{ such that}$$
$$G(y) \equiv \exists x.\, F(x, y)$$

Is this harder problem?

Example 1.4

Consider formula $\exists x.\, x > 0 \land x' = x + 1$

After substituting x *by* $x' - 1$, $\exists x.\, x' - 1 > 0$.

Since x *is not in the formula, we drop the quantifier and obtain* $x' > 1$.

Exercise 1.5

a. *Eliminate quantifiers:* $\exists x, y.\, x > 2 \land y > 3 \land y' = x + y$

b. *What do we do when* $\lor$ *in the formula?*

c. *How to eliminate universal quantifiers?*

Logical toolbox : induction principle

$$F(0) \wedge \forall n : F(n) \Rightarrow F(n+1)$$
$$\Rightarrow$$
$$\forall n : F(n)$$

Example 1.5

We prove $F(n) = (\sum_{i=0}^{n} i = n(n+1)/2)$ *by induction principle as follows*

- $F(0) = (\sum_{i=0}^{0} i = 0(0+1)/2)$
- *We show that implication* $F(n) \Rightarrow F(n+1)$ *is valid, which is*

$$(\sum_{i=0}^{n} i = n(n+1)/2) \Rightarrow (\sum_{i=0}^{n+1} i = (n+1)(n+2)/2).$$

Exercise 1.6

Show the above implication holds using a satisfiability checker.

Logical toolbox : interpolation

find a simple I such that

$$A \Rightarrow I \text{ and } I \Rightarrow B$$

For now, no trivial to see the important of interpolation.

Logical toolbox

In order to build verification tools, we need tools that automate the logical questions/queries.

Hence CS 433: automated reasoning.

In the first four lectures, we will see the need for automation.

In this course, we will briefly review available logical tool boxes.

Topic 1.2

Course Logistics

Evaluation

- Assignments : 45% (about 10% each - 4 assignments)
- Quizzes : 10% (5% each)
- Midterm : 20% (2 hour)
- Presentation: 10% (15 min)
- Final : 15% (2 hour)

Website

For further information

`https://www.cse.iitb.ac.in/~akg/courses/2019-cs615/`

All the assignments and slides will be posted at the website.

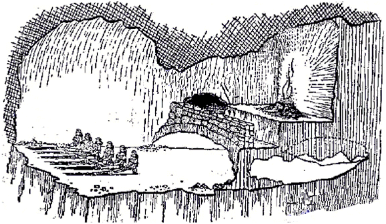
Please carefully read the course rules at the website

Topic 1.3

Program modeling

Modeling

- Object of study is often inaccessible, we only analyze its shadow



Plato's cave

- Almost impossible to define the true semantics
  of a program running on a machine
- All models (shadows) exclude many hairy details of a program
- It is almost a "matter of faith" that any result of analysis of model
  is also true for the program

Topic 1.4

A simple language

A simple language : ingredients

sometimes integer

- $V \triangleq$ vector of rational program variables
- $Exp(V) \triangleq$ linear expressions over V
- $\Sigma(V) \triangleq$ linear formulas over V

Example 1.6

$V = [x, y]$

$x + y \in Exp(V)$

$x + y \leq 3 \in \Sigma(V)$

But, $x^2 + y \leq 3 \notin \Sigma(V)$ *(why?)*

A simple language: syntax

Definition 1.1

A program *c is defined by the following grammar*

$$
\left.
\begin{array}{rll}
c ::= & \texttt{x} \ := \ \texttt{exp} & \textit{(assignment)} \\
| & \texttt{x} \ := \ \texttt{havoc}() & \textit{(havoc)} \\
| & \texttt{assume}(\texttt{F}) & \textit{(assumption)} \\
| & \texttt{assert}(\texttt{F}) & \textit{(property)}
\end{array}
\right\} \textit{data}
$$

$$
\left.
\begin{array}{rll}
| & \texttt{skip} & \textit{(empty program)} \\
| & \texttt{c}; \texttt{c} & \textit{(sequential computation)} \\
| & \texttt{c} \ [] \ \texttt{c} & \textit{(nondet composition)} \\
| & \texttt{if}(\texttt{F}) \ \texttt{c} \ \texttt{else} \ \texttt{c} & \textit{(if-then-else)} \\
| & \texttt{while}(\texttt{F}) \ \texttt{c} & \textit{(loop)}
\end{array}
\right\} \textit{control}
$$

where $\texttt{x} \in V$, $\texttt{exp} \in Exp(V)$, *and* $\texttt{F} \in \Sigma(V)$.

Let $\mathcal{P}$ be the set of all programs over variables V.

Example: a simple language

Example 1.7

Let $V = \{r, x\}$.

```
assume( r > 0 );
while( r > 0 ) {
  x := x + x;
  r := r - 1;
}
```

Exercise 1.7

Write a simple program equivalent of the following without using `if()`.

```
if( r > 0 )
  x := x + x;
else
  x := x - 1;
```

A simple language: states

Definition 1.2

A *state* *s is a pair (v,c), where*

- $v : V \to \mathbb{Q}$ *and*
- c *is yet to be executed part of program.*

Definition 1.3

The set of states is $S \triangleq (\mathbb{Q}^{|V|} \times \mathcal{P}) \cup \{(\mathtt{Error}, \mathtt{skip})\}$.

The purpose of this state will be clear soon.

Example 1.8

The following is a state, where $V = [\mathtt{r}, \mathtt{x}]$

$$(\underbrace{[2, 1]}_{v}, \underbrace{\mathtt{x} := \mathtt{x} + \mathtt{x}; \mathtt{r} := \mathtt{r} - 1}_{c})$$

Some supporting functions and notations

Definition 1.4

Let $exp \in Exp(V)$ *and* $v \in V \to \mathbb{Q}$, *let* $exp(v)$ *denote the evaluation of exp at* v.

Example 1.9

Let $V = [\mathtt{x}]$. *Let* $exp = \mathtt{x} + 1$ *and* $v = [2]$.

$$(\mathtt{x} + 1)([2]) = 3$$

Definition 1.5

Let $random()$ *returns a random rational number.*

Definition 1.6

Let f *be a function and* k *be a value. We define* $f[\mathtt{x} \to k]$ *as follows.*

$$\text{for each } \mathtt{y} \in domain(f) \qquad f[\mathtt{x} \to k](\mathtt{y}) = \begin{cases} k & \mathtt{x} == \mathtt{y} \\ f(y) & \text{otherwise} \end{cases}$$

A simple language: semantics

Definition 1.7

The set of programs defines a transition relation $T \subseteq S \times S$.
T is the smallest relation that contains the following transitions.

$$((v, \mathtt{x} \ := \ \mathtt{exp}), (v[\mathtt{x} \mapsto exp(v)], \mathtt{skip})) \in T$$

$$((v, \mathtt{x} \ := \ \mathtt{havoc()}), (v[\mathtt{x} \mapsto random()], \mathtt{skip})) \in T$$

$$((v, \mathtt{assume(F)}), (v, \mathtt{skip})) \in T \text{ if } v \models F$$

$$((v, \mathtt{assert(F)}), (v, \mathtt{skip})) \in T \text{ if } v \models F$$

$$((v, \mathtt{assert(F)}), (\mathtt{Error}, \mathtt{skip})) \in T \text{ if } v \not\models F$$

$$((v, \mathtt{c}_1; \mathtt{c}_2), (v', \mathtt{c}_1'; \mathtt{c}_2)) \in T \text{ if } ((v, \mathtt{c}_1), (v', \mathtt{c}_1')) \in T$$

$$((v, \mathtt{skip}; \mathtt{c}_2), (v, \mathtt{c}_2)) \in T$$

A simple language: semantics (contd.)

$$((v, \mathtt{c}_1[]\mathtt{c}_2), (v, \mathtt{c}_1)) \in T$$

$$((v, \mathtt{c}_1[]\mathtt{c}_2), (v, \mathtt{c}_2)) \in T$$

$$((v, \mathtt{if(F)}\ \mathtt{c}_1\ \mathtt{else}\ \mathtt{c}_2), (v, \mathtt{c}_1)) \in T \text{ if } v \models F$$

$$((v, \mathtt{if(F)}\ \mathtt{c}_1\ \mathtt{else}\ \mathtt{c}_2), (v, \mathtt{c}_2)) \in T \text{ if } v \not\models F$$

$$((v, \mathtt{while(F)}\ \mathtt{c}_1), (v, \mathtt{c}_1; \mathtt{while(F)}\ \mathtt{c}_1)) \in T \text{ if } v \models F$$

$$((v, \mathtt{while(F)}\ \mathtt{c}_1), (v, \mathtt{skip})) \in T \text{ if } v \not\models F$$

T contains the meaning of all programs.

Executions and reachability

Definition 1.8

A (in)finite sequence of states $(v_0, c_0), (v_1, c_1), ..., (v_n, c_n)$ *is an execution of program* c *if* $c_0 = c$ *and* $\forall i \in 1..n$, $((v_{i-1}, c_{i-1}), (v_i, c_i)) \in T$.

Definition 1.9

For a program c, *the reachable states are* $T^*(\mathbb{Q}^{|V|} \times \{c\})$

Definition 1.10

c *is safe if* $(\texttt{Error}, \texttt{skip}) \notin T^*(\mathbb{Q}^{|V|} \times \{c\})$

Example execution

Example 1.10

```
assume( r > 0 );
while( r > 0 ) {
  x := x + x;
  r := r - 1
}
```

$V = [\mathtt{r}, \mathtt{x}]$

An execution:

$([2,1], \mathtt{assume}(\mathtt{r} > 0); \mathtt{while}(r > 0)\{\mathtt{x} := \mathtt{x} + \mathtt{x}; \mathtt{r} := \mathtt{r} - 1;\})$

$([2,1], \mathtt{while}(r > 0)\{\mathtt{x} := \mathtt{x} + \mathtt{x}; \mathtt{r} := \mathtt{r} - 1;\})$

$([2,1], \mathtt{x} := \mathtt{x} + \mathtt{x}; \mathtt{r} := \mathtt{r} - 1; \mathtt{while}(r > 0)\{\mathtt{x} := \mathtt{x} + \mathtt{x}; \mathtt{r} := \mathtt{r} - 1;\})$

$([2,2], \mathtt{r} := \mathtt{r} - 1; \mathtt{while}(r > 0)\{\mathtt{x} := \mathtt{x} + \mathtt{x}; \mathtt{r} := \mathtt{r} - 1;\})$

$([1,2], \mathtt{while}(r > 0)\{\mathtt{x} := \mathtt{x} + \mathtt{x}; \mathtt{r} := \mathtt{r} - 1;\})$

$\vdots$

$([0,4], \mathtt{while}(r > 0)\{\mathtt{x} := \mathtt{x} + \mathtt{x}; \mathtt{r} := \mathtt{r} - 1;\})$

$([0,4], \mathtt{skip})$

Exercise: executions

Exercise 1.8

Execute the following code.

Let $v = [x]$*. Initial value* $v = [1]$*.*

```
assume( x > 0 );
x := x - 1 [] x := x + 1;
assert( x > 0 );
```

Now consider initial value $v = [0]$*.*

Exercise 1.9

Execute the following code.

Let $v = [x, y]$*.*

Initial value $v = [-1000, 2]$*.*

```
x := havoc();
y := havoc();
assume( x+y > 0 );
x := 2x + 2y + 5;
assert( x > 0 )
```

Trailing code == program locations

Example 1.11

```
L1: assume( r > 0 );
L2: while( r > 0 ) {
L3:   x := x + x;
L4:   r := r - 1
    }
L5:
```

$V = [\mathtt{r}, \mathtt{x}]$

An execution:

$([2, 1], L1)$
$([2, 1], L2)$
$([2, 1], L3)$
$([2, 2], L4)$
$([1, 2], L2)$
$\vdots$
$([0, 4], L2)$
$([0, 4], L5)$

We need not carry around trailing program. Program locations are enough.

Stuttering, non-termination, and non-determinism

The programs allow the following not so intuitive behaviors.

- Stuttering
- Non-termination
- Non-determinism

Stuttering

Example 1.12

The following program will get stuck if the initial value of `x` *is negative.*

```
assume( x > 0 );
x = 2
```

Exercise 1.10

Do real world programs have stuttering?

Non-termination

Example 1.13

The following program will not finish if the initial value of `x` *is negative.*

```
while( x < 0 ) {
 x = x - 1;
}
```

Exercise 1.11

Do real world programs have non-termination?

Non-termination

Example 1.14

The following program can execute in two ways for each initial state.

```
x = x - 1 [] x = x + 1
```

Exercise 1.12

Do real world programs have non-determinism?

Expressive power of the simple language

Exercise 1.13

Which details of real programs are ignored by this model?

- heap and pointers
- numbers with fixed bit width
- functions and stack memory
- recursion
- other data types, e.g., strings, integer, etc.
- ....any thing else?

We will live with these limitations in the first of the course.
Relaxing any of the above restrictions is a whole field on its own.

End of Lecture 1