Program verification 2019

Lecture 1: Program modeling and semantics

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2019-01-08

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
- Iphone cloud leaked pictures of JLaw
- ... etc.

Verification

- Much needed technology
- Undecidable problem
- Many fragments are NP-complete
- Open theoretical questions
- Difficult to implement algorithms
  - the field is mature enough for start-ups

Perfect field for a young bright mind to take a plunge

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

satisfiablity	$s \models F$?
validity	$\forall s : s \models F$?
implication	$F \Rightarrow G$?
quantifier elimination	given F, find G s.t. $\exists x : G(y) \equiv F(x, y)$
induction principle	$(F(0) \wedge \forall n : F(n) \Rightarrow F(n+1)) \Rightarrow \forall n : F(n)$
interpolation	find a simple I s.t. $A \Rightarrow I$ and $I \Rightarrow B$

In order to build verification tools, we need tools that automate the above questions.

Hence CS 433 (take this course next semester).

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

Topic 1.1

Course Logistics

Course structure for first half

We will have 14 meetings

- Lecture 1 is introduction (today)
- Lecture 1-4 Intro to software model checking
  - Understanding CEGAR and its variants
- Lecture 5-6 Building verification tools
  - C++, LLVM, Z3, toolchain
- Lecture 7-10 concurrent programming
  - Issues of concurrency, properties for the concurrent programs, mutual exclusion protocols, synchronization primitives, concurrent objects and their properties, weak memory models
- Lecture 11-14 verification of concurrent programs
  - Proof systems
  - CEGAR based verification of concurrent programs
  - Abstract interpretation based verification
  - ... depending where you guys like it to go

Course structure

Largely a reading and discussion course.

Initially, I will cover a few lectures.

Afterwards, I will present 30-40mins at the start of each lecture.

You guys have to present 1-2 papers every lecture, depending on the complexity of the papers. Preferably make slides.

Evaluation of first 50% :

- 30% presentations + assignments
- 20% midterms

Website

For further information

`https://www.cse.iitb.ac.in/~akg/courses/2019-cs310/`

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

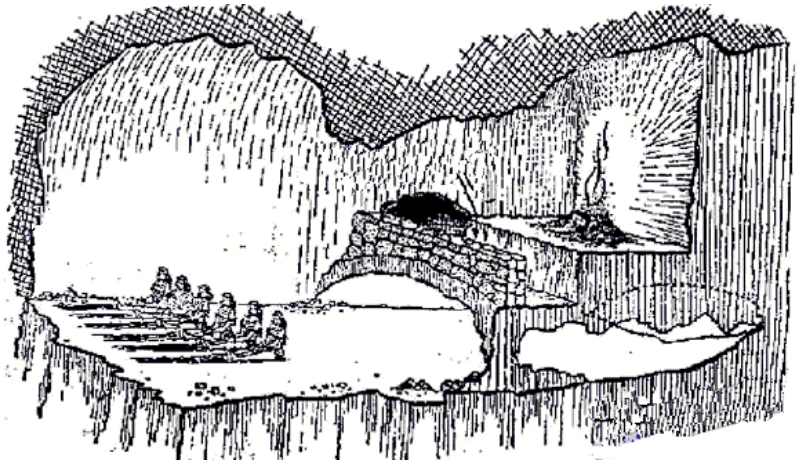
Please carefully read the course rules at the website

Topic 1.2

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 1.3

A simple language

A simple language : ingredients

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

*sometimes integer

A simple language: syntax

Definition 1.1

A program c is defined by the following grammar

$$
\begin{array}{rll}
c ::= & \texttt{x := exp} & \textit{(assignment)} \\
| & \texttt{x := havoc()} & \textit{(havoc)} \\
| & \texttt{assume(F)} & \textit{(assumption)} \\
| & \texttt{assert(F)} & \textit{(property)} \\
| & \texttt{skip} & \textit{(empty program)} \\
| & \texttt{c; c} & \textit{(sequential computation)} \\
| & \texttt{c [] c} & \textit{(nondet composition)} \\
| & \texttt{if(F) c else c} & \textit{(if-then-else)} \\
| & \texttt{while(F) c} & \textit{(loop)}
\end{array}
$$

data (assignment, havoc, assumption, property); *control* (empty program, sequential computation, nondet composition, if-then-else, loop)

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

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

Example: a simple language

Example 1.1

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

```
assume( r > 0 );
while( r > 0 ) {
  x := x + x;
  r := r - 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 $S \triangleq (\mathbb{Q}^{|V|} \times \mathcal{P}) \cup \{(\texttt{Error}, \texttt{skip})\}$.

The purpose of state $\{(\texttt{Error}, \texttt{skip})\}$ will be clear soon.

A simple language: semantics

Definition 1.4

Programs defines a transition relation $T \subseteq S \times S$.
T is the smallest relation that contains the following transitions.

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

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

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

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

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

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

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

A simple language: semantics (contd.)

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

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

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

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

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

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

T contains the meaning of all programs.

Executions and reachability

Definition 1.5

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 1.6

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

Definition 1.7

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

Example execution

Example 1.2

```
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 1.1

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 1.2

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 1.3

```
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.

Expressive power of the simple language

Exercise 1.3

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 this course.
Relaxing any of the above restrictions is a whole field on its own.

Variation in semantics

There are different styles of assigning meanings to programs

- Operational semantics
- Denotational semantics
- Axiomatic semantics

We have used operational semantics style.

We will ignore the last two in this course (very important topic!).

Small vs big step semantics

There are two sub-styles in operational semantics

- Small step (our earlier semantics)
- Big step

To appreciate the subtle differences in the styles, now we will present big step operational semantics

Big step semantic ignores intermediate steps.
It only cares about the final results.

Big step operational semantics

Definition 1.8

$\mathcal{P}$ *defines a reduction relation* $\Downarrow : S \times (\texttt{Error} \cup \mathbb{Q}^{|V|})$ *via the following rules.*

$$\frac{}{(v, x := exp) \Downarrow v[x \mapsto exp(v)]} \qquad \frac{}{(v, x := havoc()) \Downarrow v[x \mapsto random()]}$$

$$\frac{v \models \texttt{F}}{(v, \texttt{assume(F)}) \Downarrow v} \qquad \frac{v \models \texttt{F}}{(v, \texttt{assert(F)}) \Downarrow v} \qquad \frac{v \not\models \texttt{F}}{(v, \texttt{assert(F)}) \Downarrow \texttt{Error}}$$

$$\frac{}{(v, \texttt{skip}) \Downarrow v} \qquad \frac{(v, c_1) \Downarrow v' \quad (v', c_2) \Downarrow v''}{(v, c_1; c_2) \Downarrow v''} \qquad \frac{(v, c_1) \Downarrow v'}{(v, c_1 [] c_2) \Downarrow v'} \qquad \frac{(v, c_2) \Downarrow v'}{(v, c_1 [] c_2) \Downarrow v'}$$

$$\frac{v \models \texttt{F} \quad (v, \texttt{c}_1) \Downarrow v'}{(v, \texttt{if(F) c}_1 \texttt{ else c}_2) \Downarrow v'} \qquad \frac{v \not\models \texttt{F} \quad (v, \texttt{c}_2) \Downarrow v'}{(v, \texttt{if(F) c}_1 \texttt{ else c}_2) \Downarrow v'}$$

$$\frac{v \not\models \texttt{F}}{(v, \texttt{while(F) c}) \Downarrow v} \qquad \frac{v \models \texttt{F} \quad (v, \texttt{c}) \Downarrow v' \quad (v', \texttt{while(F) c}) \Downarrow v''}{(v, \texttt{while(F) c}) \Downarrow v''}$$

Example: big step semantics

Example 1.4

Let $v = [x]$. Consider the following code.

```
L1: while( x < 10 ) {
L2:    x := x + 1
    }
L3:
```

Small step:

$\{(([n], L1), ([n], L3)) | n \geq 10\} \subseteq T$
$\{(([n], L1), ([n], L2)) | n < 10\} \subseteq T$
$\{(([n], L2), ([n+1], L1)) | n < 10\} \subseteq T$

Big step:

$\{(([n], L3), n) | n \geq 10\} \subseteq \Downarrow$
$\{(([n], L2), 10) | n < 9\} \cup \{(([n], L2), n+1) | n \geq 9\} \subseteq \Downarrow$
$\{(([n], L1), 10) | n < 10\} \cup \{(([n], L1), n) | n \geq 10\} \subseteq \Downarrow$

Exercise: big step semantics

Exercise 1.4

Let $v = [x]$. Consider the following code.

```
L1: while( x < 10 ) {
L2:  if x > 0 then
L3:   x := x + 1
     else
L4:   skip
    }
L5:
```

Write the relevant parts of T and $\Downarrow$ wrt to the above program.

Agreement between small and big step semantics

Theorem 1.1

$$(v', \texttt{skip}) \in T^*(\mathtt{c}, v) \quad \Leftrightarrow \quad (v, \mathtt{c}) \Downarrow v'$$

Proof.

Simple structural induction. □

This theorem is not that strong as it looks. Stuck and non-terminating executions are not compared in the above theorem.

Exercise 1.5 (Questions for lunch)

a. What are other differences between small and big step semantics?

b. What is denotational semantics? *... search web*

End of Lecture 1