CS766: Analysis of concurrent programs 2020

Lecture 1: Program modeling and semantics

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2020-01-20

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

Concurrent software

- Important
- Complex
- Buggy

Ensuring reliability is a bigger challenge

What is so hard about concurrency?

Schedule blowup

Exercise 1.1

What is the number of schedules between two threads with number of instructions N_1 and N_2 ?

The blowup is not the only problem.

In the presence of synchronization primitives, the sets of allowed schedules appear deceptively simple, but are ugly beasts

e. g., locks, barriers, etc

Summarize interleavings

For an effective analysis of concurrent programs, one should be able to efficiently summarize valid interleavings

A few active lines of research

1. Environment computations
   - e.g. if global variable $g > 0$, some thread increases g by 2.
2. Sequentialization (Code transformation)
   - e.g. merge code of thread 1 and 2 such that the effect of the merged code is same as the original code.
3. Happens-before summaries
   - e.g. the write at line 10 must happen before the read at line 20
4. Bounded-context switches
   - e.g. maximum number of context switches during an execution is 5.

We will discuss the above methods.

Example: Peterson

```
int turn; int flag0 = 0, flag1 = 0;

void* p0(void *) {
  flag0 = 1;
  turn = 1;
  while((flag1 == 1) and (turn == 1));
  // critical section
  flag0 = 0;
}

void* p1(void *) {
  flag1 = 1;
  turn = 0;
  while((flag0 == 1) and (turn == 0));
  //critical section
  flag1 = 0;
}
```

Example: Concurrent object

Here is an concurrent implementation of queue

```
Vector q;

// x > 0
void* Enqueue(int x) {
  q.push_back(x)
}

int Dequeue() {
  while( true ) {
    l = q.length()
    for( i = 0 ; i < l; i ++ ) {
      x = swap( q, i, 0 )
      if ( x != 0) {
        return x
      }
    }
  }
```

Topic 1.1

Course contents

Topic 1.2

Course Logistics

Course structure for first half

We will have 13 meetings

- Lecture 1 is introduction (today)
- Lecture 1-5 Introduction to software model checking for sequential programs
  - Understanding CEGAR and its variants
-
- Lecture 6-9 concurrent programming
  - Issues of concurrency, properties for the concurrent programs, mutual exclusion protocols, synchronization primitives, concurrent objects and their properties
- Lecture 10-13 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

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

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 1.1

$V = [x, y]$

$x + y \in Exp(V)$

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

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*

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

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

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

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

Example: a simple language

Example 1.2

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

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

Exercise 1.2

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

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

Purpose of havoc

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 this state will be clear soon.

Example 1.3

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 1.4

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.

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

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

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

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

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

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

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

A simple language: semantics (contd.)

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

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

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

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

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

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

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 1.5

```
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 1.3

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 1.4

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 1.6

```
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 the 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 1.7

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

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

Exercise 1.5

Do real world programs have stuttering?

Non-termination

Example 1.8

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

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

Exercise 1.6

Do real world programs have non-termination?

Non-termination

Example 1.9

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

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

Exercise 1.7

Do real world programs have non-determinism?

Expressive power of the simple language

Exercise 1.8

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.

Topic 1.5

Logical toolbox

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 1.10

$$\{x \mapsto 1, y \mapsto 2\} \models x + y = 3.$$

model *formula*

Exercise 1.9

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

Exercise 1.10

Can we say something more about the last formula?

Logical toolbox : satisfiablity

Is there any model?

$$\models F?$$

Harder problem!

Exercise 1.11

- $\models x + y = 3 \wedge x > 0$?
- $\models x + y = 3 \wedge x > 0 \wedge y > 10$?
- $\models x > 0 \vee x < 1$?

disjunction

Exercise 1.12

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 1.11

$x > 0 \vee x < 1$ is valid because $x \leq 0 \wedge x \geq 1$ 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 1.12

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 1.13

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 1.13

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 1.14

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 1.14

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 1.6

Problems

End of Lecture 1

Topic 1.7

Extra topic: Big-step semantics

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.

Notation alert: deduction rules

$$\textsc{RuleName}\,\frac{\text{Stuff-already-there}}{\text{Stuff-to-be-added}}\,\text{Conditions to be met}$$

Big step operational semantics

Definition 1.11

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

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

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

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

Big step operational semantics (contd.)

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

$$\frac{v \models \mathtt{F} \quad (v, \mathtt{c}_1) \Downarrow v'}{(v, \mathtt{if(F)\ c_1\ else\ c_2}) \Downarrow v'} \qquad \frac{v \not\models \mathtt{F} \quad (v, \mathtt{c}_2) \Downarrow v'}{(v, \mathtt{if(F)\ c_1\ else\ c_2}) \Downarrow v'}$$

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

Example: big step semantics

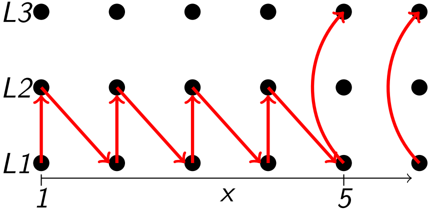
Example 1.15

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

```
L1: while( x < 5 ) {
L2:    x := x + 1
    }
L3:
```

Exercise 1.15

Draw $\Downarrow$ *edges.*



Small step:

$\{(([n], L1), ([n], L3)) | n \geq 5\} \subseteq T$
$\{(([n], L1), ([n], L2)) | n < 5\} \subseteq T$
$\{(([n], L2), ([n+1], L1)) | n < 5\} \subseteq T$

Big step:

$\{(([n], L3), n) | n \in \mathbb{Q}\} \subseteq \Downarrow$
$\{(([n], L2), 5) | n < 5\} \cup \{(([n], L2), n+1) | n \geq 5\} \subseteq \Downarrow$
$\{(([n], L1), 5) | n < 5\} \cup \{(([n], L1), n) | n \geq 5\} \subseteq \Downarrow$

Exercise: big step semantics

Exercise 1.16

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

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

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 1.17

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

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