Program verification 2016

Lecture 1: Program modeling

Instructor: Ashutosh Gupta

TIFR, India

Compile date: 2016-02-01

Programs

Our life depends on programs

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

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

Verification

- Much needed technology
- Undecidable problem
- Many fragments are NP-complete
- Open theoretical questions

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

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

The course

A course is not sufficient to cover the full breath of verification

The course

A course is not sufficient to cover the full breath of verification

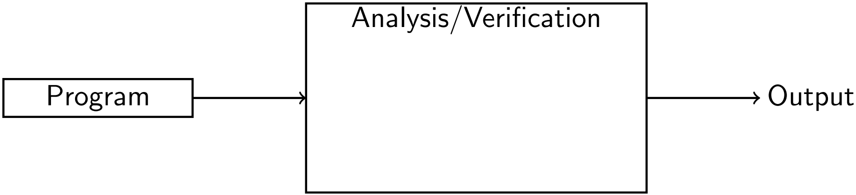
We will study only in the following two key directions

- Software verification for arithmetic programs
- Automated reasoning to support the verification problem

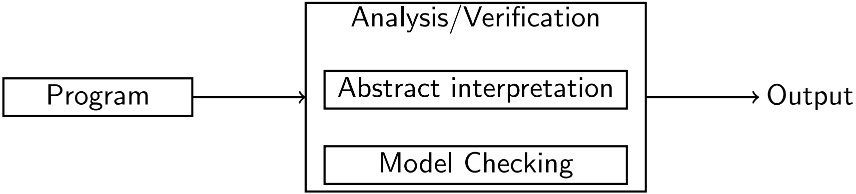
Topic 1.1

Course contents: verification module

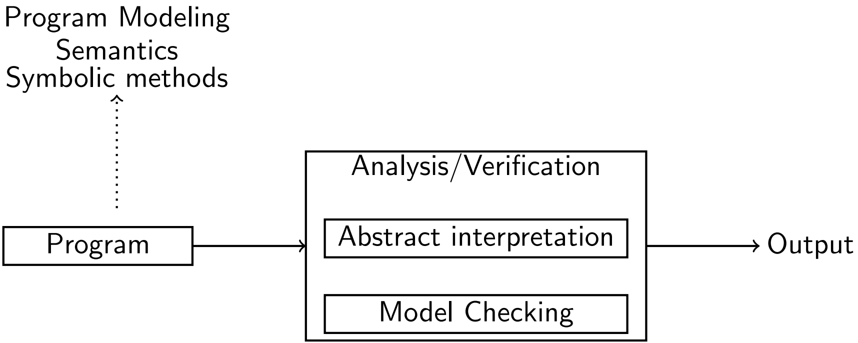
Overview : Software model checking



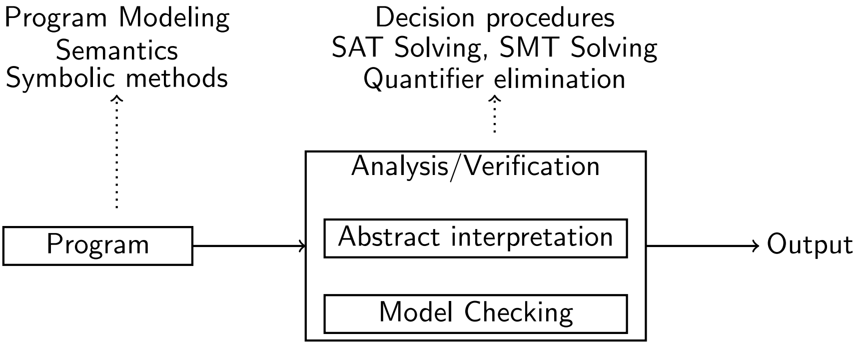
Overview : Software model checking



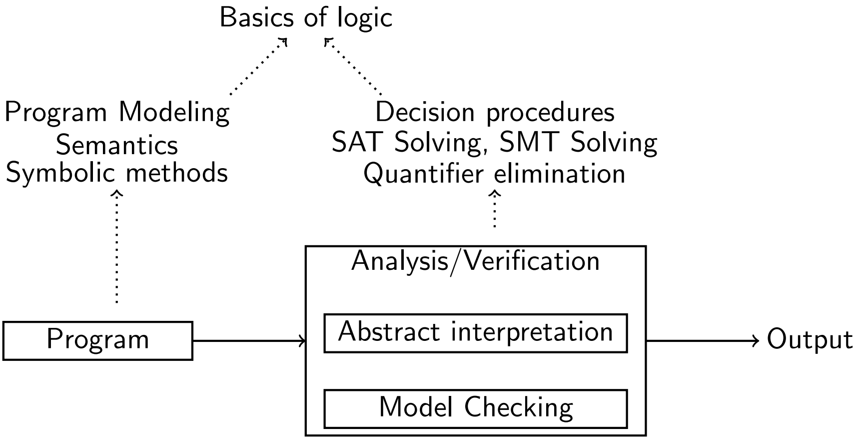
Overview : Software model checking



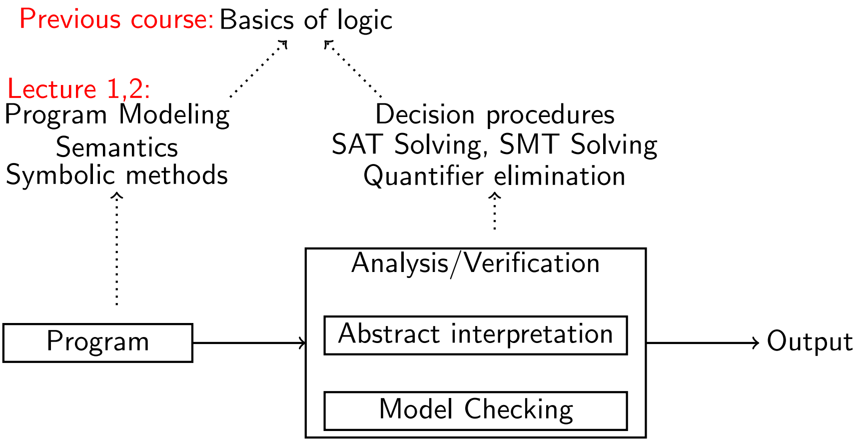
Overview : Software model checking



Overview : Software model checking



Overview : Software model checking



Overview : Software model checking

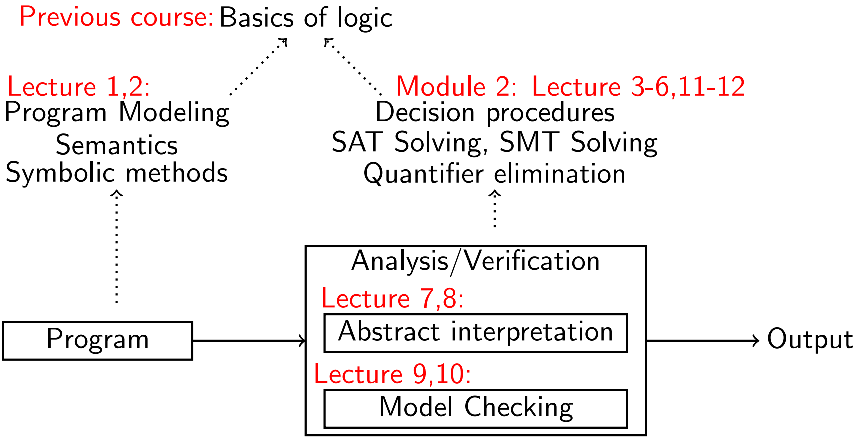
Previous course: Basics of logic

Lecture 1,2:
Program Modeling
Semantics
Symbolic methods

Module 2: Lecture 3-6,11-12
Decision procedures
SAT Solving, SMT Solving
Quantifier elimination

Program → Analysis/Verification (Abstract interpretation; Model Checking) → Output

Overview : Software model checking



Logic in verification

Differential equations
are the calculus of
Electrical engineering

Logic in verification

Differential equations
are the calculus of
Electrical engineering

Logic
is the calculus of
Computer science

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

$$P$$

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

$$F$$

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.

$$P \models F$$

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	$\forall s : s \models F$?

Logical toolbox

satisfiablity	$s \models F?$
validity	$\forall s : s \models F?$
implication	$F \Rightarrow G?$

Logical toolbox

satisfiablity	$s \models F$?
validity	$\forall s : s \models F$?
implication	$F \Rightarrow G$?
quantifier elimination	given F, find G s.t. $\exists x : G(y) \equiv F(x, y)$

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

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 $I \Rightarrow B$

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.

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 module II.

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 module II.

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

Topic 1.2

Course Logistics

Evaluation

- Assignments : 40% (4% each - 10 assignments)
- Midterm : 20% (1 hour)
- Presentation: 10% (15 min)
- Final project : 30% (expected some non-trivial programming)

Website

For further information

`http://www.tcs.tifr.res.in/~agupta/courses/2016-verification`

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

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

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

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

- $V \triangleq$ vector of rational* program variables

*sometimes integer

A simple language : ingredients

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

*sometimes integer

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*

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

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

A simple language: syntax

Definition 1.1

A *program* *c is defined by the following grammar*

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

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

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

A simple language: syntax

Definition 1.1

A program c is defined by the following grammar

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

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

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

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* $(v, \mathtt{c})$*, where*

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

Definition 1.3

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

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 state $\{(\mathtt{Error}, \mathtt{skip})\}$ 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.

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

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.

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

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

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.

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

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

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

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.

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

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

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

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

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.

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

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

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

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

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

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.

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

A simple language: semantics (contd.)

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

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

A simple language: semantics (contd.)

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

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

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

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

A simple language: semantics (contd.)

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

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

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

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

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

A simple language: semantics (contd.)

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

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

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

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

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

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

A simple language: semantics (contd.)

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

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

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

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

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

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

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* $\forall i \in 1..n$, $((v_{i-1}, c_{i-1}), (v_i, c_i)) \in T$.

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* $T^*(\mathbb{Q}^{|V|} \times \{c\})$

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* $(\mathtt{Error}, \mathtt{skip}) \notin T^*(\mathbb{Q}^{|V|} \times \{c\})$

Example execution

Example 1.2

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

Example execution

Example 1.2

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

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

Example execution

Example 1.2

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

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

An execution:

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

Example execution

Example 1.2

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

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

An execution:

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

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

Example execution

Example 1.2

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

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

An execution:

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

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

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

Example execution

Example 1.2

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

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

An execution:

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

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

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

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

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

Example execution

Example 1.2

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

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

An execution:

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

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

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

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

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

$\vdots$

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

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 );
```

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: 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:

$([2, 1], L1)$

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:

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

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?

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

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

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.

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?

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

Variation in semantics

There are different styles of assigning meanings to programs

- Operational semantics
- Denotational semantics
- Axiomatic semantics

We have used operational semantics style.

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

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

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.

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.

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

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.

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

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

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.*

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

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

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

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.

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

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

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

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

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.*

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

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

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

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

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

Example: big step semantics

Example 1.4

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

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

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:

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

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:

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

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

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:

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

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

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

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:

$\{(([n], L3), n) | n \geq 10\} \subseteq \Downarrow$

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:

$\{(([n], L3), n) | n \geq 10\} \subseteq \Downarrow$

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

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 ) {
L1:  if x > 0 then
L2:   x := x + 1
     else
L3:   skip
    }
L4:
```

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. $\square$

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. $\square$

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

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 1.5 (Questions for lunch)

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

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

Topic 1.5

Logical representation

Computing reachable states

- Proving safety is computing reachable states.

Computing reachable states

- Proving safety is computing reachable states.
- states are infinite $\implies$ enumeration impossible

Computing reachable states

- Proving safety is computing reachable states.
- states are infinite $\implies$ enumeration impossible
- To compute reachable states, we need
  - finite representations of transition relation and
  - ability to compute transitive closure of transition relation

Computing reachable states

- Proving safety is computing reachable states.
- states are infinite $\implies$ enumeration impossible
- To compute reachable states, we need
  - finite representations of transition relation and
  - ability to compute transitive closure of transition relation
- Idea: use logic for the above goals

Program statements as formulas (Notation)

- In logical representation, we add a new variable *err* in V to represent error state. Initially, $err = 0$ and $err = 1$ means error has occurred.

Program statements as formulas (Notation)

- In logical representation, we add a new variable *err* in V to represent error state. Initially, $err = 0$ and $err = 1$ means error has occurred.
- V' be the vector of variables obtained by adding prime after each variable in V. We use V' to denote next value of variables.

Program statements as formulas (Notation)

- In logical representation, we add a new variable *err* in V to represent error state. Initially, $err = 0$ and $err = 1$ means error has occurred.
- V' be the vector of variables obtained by adding prime after each variable in V. We use V' to denote next value of variables.
- $$\text{For } U \subseteq V, \text{ let } frame(U) \triangleq \bigwedge_{x \in V \setminus U} (x' = x)$$

  In case of singleton U, we only write the element as parameter.

Program statements as formulas (contd.)

We define logical formula ρ for the data statements as follows.

- $\rho(\texttt{x := exp}) \triangleq \texttt{x}' = \texttt{exp} \land \mathit{frame}(\texttt{x})$

Program statements as formulas (contd.)

We define logical formula ρ for the data statements as follows.

- $\rho(\texttt{x := exp}) \triangleq \texttt{x}' = \texttt{exp} \land \mathit{frame}(\texttt{x})$
- $\rho(\texttt{x := havoc()}) \triangleq \mathit{frame}(\texttt{x})$

Program statements as formulas (contd.)

We define logical formula ρ for the data statements as follows.

- $\rho(\texttt{x := exp}) \triangleq \texttt{x}' = \texttt{exp} \land \textit{frame}(\texttt{x})$
- $\rho(\texttt{x := havoc()}) \triangleq \textit{frame}(\texttt{x})$
- $\rho(\texttt{assume(F)}) \triangleq \texttt{F} \land \textit{frame}(\emptyset)$

Program statements as formulas (contd.)

We define logical formula ρ for the data statements as follows.

- $\rho(\texttt{x := exp}) \triangleq \texttt{x}' = \texttt{exp} \land \mathit{frame}(\texttt{x})$
- $\rho(\texttt{x := havoc()}) \triangleq \mathit{frame}(\texttt{x})$
- $\rho(\texttt{assume(F)}) \triangleq \texttt{F} \land \mathit{frame}(\emptyset)$
- $\rho(\texttt{assert(F)}) \triangleq \texttt{F} \Rightarrow \mathit{frame}(\emptyset)$

Program statements as formulas (contd.)

We define logical formula ρ for the data statements as follows.

- $\rho(\texttt{x := exp}) \triangleq \texttt{x}' = \texttt{exp} \land \mathit{frame}(\texttt{x})$
- $\rho(\texttt{x := havoc()}) \triangleq \mathit{frame}(\texttt{x})$
- $\rho(\texttt{assume(F)}) \triangleq \texttt{F} \land \mathit{frame}(\emptyset)$
- $\rho(\texttt{assert(F)}) \triangleq \texttt{F} \Rightarrow \mathit{frame}(\emptyset)$

Since control locations in a program are always finite, control statements need not be redefined.

Program statements as formulas (contd.)

We define logical formula ρ for the data statements as follows.

- $\rho(\texttt{x := exp}) \triangleq \texttt{x}' = \texttt{exp} \land \textit{frame}(\texttt{x})$
- $\rho(\texttt{x := havoc()}) \triangleq \textit{frame}(\texttt{x})$
- $\rho(\texttt{assume(F)}) \triangleq \texttt{F} \land \textit{frame}(\emptyset)$
- $\rho(\texttt{assert(F)}) \triangleq \texttt{F} \Rightarrow \textit{frame}(\emptyset)$

Since control locations in a program are always finite, control statements need not be redefined.

Example 1.5

Let $V = [x, y, err]$.

- $\rho(\texttt{x := y} + 1) =$

Program statements as formulas (contd.)

We define logical formula ρ for the data statements as follows.

- $\rho(\texttt{x := exp}) \triangleq \texttt{x}' = \texttt{exp} \land \mathit{frame}(\texttt{x})$
- $\rho(\texttt{x := havoc()}) \triangleq \mathit{frame}(\texttt{x})$
- $\rho(\texttt{assume(F)}) \triangleq \texttt{F} \land \mathit{frame}(\emptyset)$
- $\rho(\texttt{assert(F)}) \triangleq \texttt{F} \Rightarrow \mathit{frame}(\emptyset)$

Since control locations in a program are always finite, control statements need not be redefined.

Example 1.5

Let $V = [x, y, err]$.

- $\rho(\texttt{x := y} + 1) = (\texttt{x}' = \texttt{y} + 1 \land \texttt{y}' = \texttt{y} \land err' = err)$
- $\rho(\texttt{x := havoc()}) =$

Program statements as formulas (contd.)

We define logical formula ρ for the data statements as follows.

- $\rho(\mathtt{x} \ := \ \mathtt{exp}) \triangleq \mathtt{x}' = \mathtt{exp} \land \mathit{frame}(\mathtt{x})$
- $\rho(\mathtt{x} \ := \ \mathtt{havoc()}) \triangleq \mathit{frame}(\mathtt{x})$
- $\rho(\mathtt{assume(F)}) \triangleq \mathtt{F} \land \mathit{frame}(\emptyset)$
- $\rho(\mathtt{assert(F)}) \triangleq \mathtt{F} \Rightarrow \mathit{frame}(\emptyset)$

Since control locations in a program are always finite, control statements need not be redefined.

Example 1.5

Let $V = [x, y, err]$.

- $\rho(\mathtt{x} \ := \ \mathtt{y} + 1) = (\mathtt{x}' = \mathtt{y} + 1 \land \mathtt{y}' = \mathtt{y} \land \mathit{err}' = \mathit{err})$
- $\rho(\mathtt{x} \ := \ \mathtt{havoc()}) = (\mathtt{y}' = \mathtt{y} \land \mathit{err}' = \mathit{err})$
- $\rho(\mathtt{assume(x} > \mathtt{0))} =$

Program statements as formulas (contd.)

We define logical formula ρ for the data statements as follows.

- $\rho(\mathtt{x\ :=\ exp}) \triangleq \mathtt{x}' = \mathtt{exp} \wedge \mathit{frame}(\mathtt{x})$
- $\rho(\mathtt{x\ :=\ havoc()}) \triangleq \mathit{frame}(\mathtt{x})$
- $\rho(\mathtt{assume(F)}) \triangleq \mathtt{F} \wedge \mathit{frame}(\emptyset)$
- $\rho(\mathtt{assert(F)}) \triangleq \mathtt{F} \Rightarrow \mathit{frame}(\emptyset)$

Since control locations in a program are always finite, control statements need not be redefined.

Example 1.5

Let $V = [x, y, err]$*.*

- $\rho(\mathtt{x\ :=\ y} + 1) = (\mathtt{x}' = \mathtt{y} + 1 \wedge \mathtt{y}' = \mathtt{y} \wedge err' = err)$
- $\rho(\mathtt{x\ :=\ havoc()}) = (\mathtt{y}' = \mathtt{y} \wedge err' = err)$
- $\rho(\mathtt{assume(x} > 0)) = (\mathtt{x} > 0 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{y}' = \mathtt{y} \wedge err' = err)$
- $\rho(\mathtt{assert(x} > 0)) =$

Program statements as formulas (contd.)

We define logical formula ρ for the data statements as follows.

- $\rho(\mathtt{x\ :=\ exp}) \triangleq \mathtt{x}' = \mathtt{exp} \land \mathit{frame}(\mathtt{x})$
- $\rho(\mathtt{x\ :=\ havoc()}) \triangleq \mathit{frame}(\mathtt{x})$
- $\rho(\mathtt{assume(F)}) \triangleq \mathtt{F} \land \mathit{frame}(\emptyset)$
- $\rho(\mathtt{assert(F)}) \triangleq \mathtt{F} \Rightarrow \mathit{frame}(\emptyset)$

Since control locations in a program are always finite, control statements need not be redefined.

Example 1.5

Let $V = [x, y, err]$.

- $\rho(\mathtt{x\ :=\ y} + 1) = (\mathtt{x}' = \mathtt{y} + 1 \land \mathtt{y}' = \mathtt{y} \land err' = err)$
- $\rho(\mathtt{x\ :=\ havoc()}) = (\mathtt{y}' = \mathtt{y} \land err' = err)$
- $\rho(\mathtt{assume(x} > 0)) = (\mathtt{x} > 0 \land \mathtt{x}' = \mathtt{x} \land \mathtt{y}' = \mathtt{y} \land err' = err)$
- $\rho(\mathtt{assert(x} > 0)) = (\mathtt{x} > 0 \Rightarrow (\mathtt{x}' = \mathtt{x} \land \mathtt{y}' = \mathtt{y} \land err' = err))$

Program statements as formulas (contd.)

We define logical formula ρ for the data statements as follows.

- $\rho(\mathtt{x} \;:=\; \mathtt{exp}) \triangleq \mathtt{x}' = \mathtt{exp} \land \mathit{frame}(\mathtt{x})$
- $\rho(\mathtt{x} \;:=\; \mathtt{havoc}()) \triangleq \mathit{frame}(\mathtt{x})$
- $\rho(\mathtt{assume(F)}) \triangleq \mathtt{F} \land \mathit{frame}(\emptyset)$
- $\rho(\mathtt{assert(F)}) \triangleq \mathtt{F} \Rightarrow \mathit{frame}(\emptyset)$

Since control locations in a program are always finite, control statements need not be redefined.

Example 1.5

Let $V = [x, y, err]$.

- $\rho(\mathtt{x} \;:=\; \mathtt{y} + 1) = (\mathtt{x}' = \mathtt{y} + 1 \land \mathtt{y}' = \mathtt{y} \land \mathit{err}' = \mathit{err})$
- $\rho(\mathtt{x} \;:=\; \mathtt{havoc}()) = (\mathtt{y}' = \mathtt{y} \land \mathit{err}' = \mathit{err})$
- $\rho(\mathtt{assume}(\mathtt{x} > 0)) = (\mathtt{x} > 0 \land \mathtt{x}' = \mathtt{x} \land \mathtt{y}' = \mathtt{y} \land \mathit{err}' = \mathit{err})$
- $\rho(\mathtt{assert}(\mathtt{x} > 0)) = (\mathtt{x} > 0 \Rightarrow (\mathtt{x}' = \mathtt{x} \land \mathtt{y}' = \mathtt{y} \land \mathit{err}' = \mathit{err}))$

Exercise 1.6

Show ρ correctly model assert statement

Topic 1.6

Aggregated semantics

Strongest post: set of valuations to set of valuations

Definition 1.9

Strongest post operator $sp : \mathfrak{p}(\mathbb{Q}^{|V|}) \times \mathcal{P} \to \mathfrak{p}(\mathbb{Q}^{|V|})$ *is defined as follows.*

$$sp(X, \texttt{c}) \triangleq \{v' | \exists v : v \in X \land (v', \texttt{skip}) \in T^*((v, \texttt{c}))\},$$

where $X \subseteq \mathbb{Q}^{|V|}$ *and* $\texttt{c}$ *is a program.*

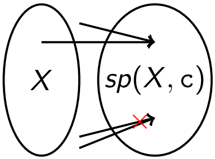
Strongest post: set of valuations to set of valuations

Definition 1.9

Strongest post operator $sp : \mathfrak{p}(\mathbb{Q}^{|V|}) \times \mathcal{P} \to \mathfrak{p}(\mathbb{Q}^{|V|})$ *is defined as follows.*

$$sp(X, \mathtt{c}) \triangleq \{v' | \exists v : v \in X \wedge (v', \mathtt{skip}) \in T^*((v, \mathtt{c}))\},$$

where $X \subseteq \mathbb{Q}^{|V|}$ *and* $\mathtt{c}$ *is a program.*



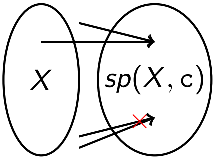
Strongest post: set of valuations to set of valuations

Definition 1.9

Strongest post operator $sp : \mathfrak{p}(\mathbb{Q}^{|V|}) \times \mathcal{P} \to \mathfrak{p}(\mathbb{Q}^{|V|})$ *is defined as follows.*

$$sp(X, \mathtt{c}) \triangleq \{v' | \exists v : v \in X \wedge (v', \mathtt{skip}) \in T^*((v, \mathtt{c}))\},$$

where $X \subseteq \mathbb{Q}^{|V|}$ *and* $\mathtt{c}$ *is a program.*



Example 1.6

Consider $V = [\mathrm{x}]$ *and* $X = \{[n] | n > 0\}$.

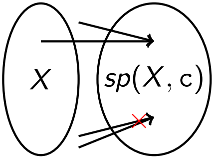
Strongest post: set of valuations to set of valuations

Definition 1.9

Strongest post operator $sp : \mathfrak{p}(\mathbb{Q}^{|V|}) \times \mathcal{P} \rightarrow \mathfrak{p}(\mathbb{Q}^{|V|})$ *is defined as follows.*

$$sp(X, \mathtt{c}) \triangleq \{v' | \exists v : v \in X \wedge (v', \mathtt{skip}) \in T^*((v, \mathtt{c}))\},$$

where $X \subseteq \mathbb{Q}^{|V|}$ *and* $\mathtt{c}$ *is a program.*



Example 1.6

Consider $V = [\mathtt{x}]$ *and* $X = \{[n] | n > 0\}$.

$sp(X, \mathtt{x} := \mathtt{x} + 1) =$

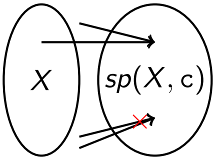
Strongest post: set of valuations to set of valuations

Definition 1.9

Strongest post operator $sp : \mathfrak{p}(\mathbb{Q}^{|V|}) \times \mathcal{P} \to \mathfrak{p}(\mathbb{Q}^{|V|})$ *is defined as follows.*

$$sp(X, \mathtt{c}) \triangleq \{v' | \exists v : v \in X \wedge (v', \mathtt{skip}) \in T^*((v, \mathtt{c}))\},$$

where $X \subseteq \mathbb{Q}^{|V|}$ *and* $\mathtt{c}$ *is a program.*



Example 1.6

Consider $V = [\mathrm{x}]$ *and* $X = \{[n] | n > 0\}$.
$sp(X, \mathrm{x} := \mathrm{x} + 1) = \{[n] | n > 1\}$

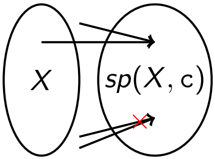
Strongest post: set of valuations to set of valuations

Definition 1.9

Strongest post operator $sp : \mathfrak{p}(\mathbb{Q}^{|V|}) \times \mathcal{P} \to \mathfrak{p}(\mathbb{Q}^{|V|})$ *is defined as follows.*

$$sp(X, \mathtt{c}) \triangleq \{v' | \exists v : v \in X \wedge (v', \mathtt{skip}) \in T^*((v, \mathtt{c}))\},$$

where $X \subseteq \mathbb{Q}^{|V|}$ *and* $\mathtt{c}$ *is a program.*



Example 1.6

Consider $V = [\mathtt{x}]$ *and* $X = \{[n] | n > 0\}$.
$sp(X, \mathtt{x} := \mathtt{x} + 1) = \{[n] | n > 1\}$

Exercise 1.7

Why use of word "strongest"?

Symbolic sp

A formula in $\Sigma(V)$ represents a set of valuations.
Hence, we define symbolic sp that transforms formulas.

$$sp : \Sigma(V) \times \mathcal{P} \to \Sigma(V)$$

Symbolic sp

A formula in $\Sigma(V)$ represents a set of valuations.
Hence, we define symbolic sp that transforms formulas.

$$sp : \Sigma(V) \times \mathcal{P} \rightarrow \Sigma(V)$$

For data statements, the equivalent definition of symbolic sp is

$$sp(F, \mathsf{c}) \triangleq (\exists V : F \wedge \rho(\mathsf{c}))[V/V'].$$

Symbolic sp

A formula in $\Sigma(V)$ represents a set of valuations.
Hence, we define symbolic sp that transforms formulas.

$$sp : \Sigma(V) \times \mathcal{P} \rightarrow \Sigma(V)$$

For data statements, the equivalent definition of symbolic sp is

$$sp(F, \mathsf{c}) \triangleq (\exists V : F \wedge \rho(\mathsf{c}))[V/V'].$$

Example 1.7

Let $V = [\mathrm{x}, \mathrm{y}, err]$ *and* $\mathsf{c} = \mathrm{x} := \mathrm{y} + 1$.

Symbolic sp

A formula in $\Sigma(V)$ represents a set of valuations.
Hence, we define symbolic sp that transforms formulas.

$$sp : \Sigma(V) \times \mathcal{P} \to \Sigma(V)$$

For data statements, the equivalent definition of symbolic sp is

$$sp(F, \mathrm{c}) \triangleq (\exists V : F \wedge \rho(\mathrm{c}))[V/V'].$$

Example 1.7

Let $V = [\mathrm{x}, \mathrm{y}, err]$ *and* $\mathrm{c} = \mathrm{x} := \mathrm{y} + 1$.
$sp(\mathrm{y} > 2, \mathrm{c}) = (\exists \mathrm{x}, \mathrm{y}, err.\ (\mathrm{y} > 2 \wedge \mathrm{x}' = \mathrm{y} + 1 \wedge \mathrm{y}' = \mathrm{y} \wedge err' = err))[V/V']$

Symbolic sp

A formula in $\Sigma(V)$ represents a set of valuations.
Hence, we define symbolic sp that transforms formulas.

$$sp : \Sigma(V) \times \mathcal{P} \to \Sigma(V)$$

For data statements, the equivalent definition of symbolic sp is

$$sp(F, \mathtt{c}) \triangleq (\exists V : F \wedge \rho(\mathtt{c}))[V/V'].$$

Example 1.7

Let $V = [\mathrm{x}, \mathrm{y}, err]$ *and* $\mathtt{c} = \mathrm{x} := \mathrm{y} + 1$.
$sp(\mathrm{y} > 2, \mathtt{c}) = (\exists \mathrm{x}, \mathrm{y}, err.\ (\mathrm{y} > 2 \wedge \mathrm{x}' = \mathrm{y} + 1 \wedge \mathrm{y}' = \mathrm{y} \wedge err' = err))[V/V']$
$= (\mathrm{y}' > 2 \wedge \mathrm{x}' > 3)[V/V']$

Symbolic sp

A formula in $\Sigma(V)$ represents a set of valuations.
Hence, we define symbolic sp that transforms formulas.

$$sp : \Sigma(V) \times \mathcal{P} \to \Sigma(V)$$

For data statements, the equivalent definition of symbolic sp is

$$sp(F, \mathrm{c}) \triangleq (\exists V : F \wedge \rho(\mathrm{c}))[V/V'].$$

Example 1.7

Let $V = [\mathrm{x}, \mathrm{y}, err]$ *and* $\mathtt{c} = \mathrm{x} := \mathrm{y} + 1$.
$sp(\mathrm{y} > 2, \mathrm{c}) = (\exists \mathrm{x}, \mathrm{y}, err.\ (\mathrm{y} > 2 \wedge \mathrm{x}' = \mathrm{y} + 1 \wedge \mathrm{y}' = \mathrm{y} \wedge err' = err))[V/V']$
$= (\mathrm{y}' > 2 \wedge \mathrm{x}' > 3)[V/V'] = (\mathrm{y} > 2 \wedge \mathrm{x} > 3)$

Symbolic sp

A formula in $\Sigma(V)$ represents a set of valuations.
Hence, we define symbolic sp that transforms formulas.

$$sp : \Sigma(V) \times \mathcal{P} \rightarrow \Sigma(V)$$

For data statements, the equivalent definition of symbolic sp is

$$sp(F, \mathtt{c}) \triangleq (\exists V : F \wedge \rho(\mathtt{c}))[V/V'].$$

Example 1.7

Let $V = [\mathtt{x}, \mathtt{y}, err]$ *and* $\mathtt{c} = \mathtt{x} := \mathtt{y} + 1$.
$sp(\mathtt{y} > 2, \mathtt{c}) = (\exists \mathtt{x}, \mathtt{y}, err.\ (\mathtt{y} > 2 \wedge \mathtt{x}' = \mathtt{y} + 1 \wedge \mathtt{y}' = \mathtt{y} \wedge err' = err))[V/V']$
$= (\mathtt{y}' > 2 \wedge \mathtt{x}' > 3)[V/V'] = (\mathtt{y} > 2 \wedge \mathtt{x} > 3)$

Exercise 1.8

- $sp(\mathtt{y} > 2 \wedge err = 0, \mathtt{x} := \mathtt{havoc}()) =$

Symbolic sp

A formula in $\Sigma(V)$ represents a set of valuations.
Hence, we define symbolic sp that transforms formulas.

$$sp : \Sigma(V) \times \mathcal{P} \to \Sigma(V)$$

For data statements, the equivalent definition of symbolic sp is

$$sp(F, \mathrm{c}) \triangleq (\exists V : F \wedge \rho(\mathrm{c}))[V/V'].$$

Example 1.7

Let $V = [\mathrm{x}, \mathrm{y}, err]$ *and* $\mathrm{c} = \mathrm{x} := \mathrm{y} + 1$.
$sp(\mathrm{y} > 2, \mathrm{c}) = (\exists \mathrm{x}, \mathrm{y}, err.\ (\mathrm{y} > 2 \wedge \mathrm{x}' = \mathrm{y} + 1 \wedge \mathrm{y}' = \mathrm{y} \wedge err' = err))[V/V']$
$= (\mathrm{y}' > 2 \wedge \mathrm{x}' > 3)[V/V'] = (\mathrm{y} > 2 \wedge \mathrm{x} > 3)$

Exercise 1.8

- $sp(\mathrm{y} > 2 \wedge err = 0, \mathrm{x} := \mathtt{havoc}()) = \ (\mathrm{y} > 2 \wedge err = 0)$
- $sp(\mathrm{y} > 2 \wedge err = 0, \mathtt{assume}(\mathrm{y} < 10)) =$

Symbolic sp

A formula in $\Sigma(V)$ represents a set of valuations.
Hence, we define symbolic sp that transforms formulas.

$$sp : \Sigma(V) \times \mathcal{P} \to \Sigma(V)$$

For data statements, the equivalent definition of symbolic sp is

$$sp(F, \mathrm{c}) \triangleq (\exists V : F \wedge \rho(\mathrm{c}))[V/V'].$$

Example 1.7

Let $V = [\mathrm{x}, \mathrm{y}, err]$ *and* $\mathrm{c} = \mathrm{x} := \mathrm{y} + 1$.
$sp(\mathrm{y} > 2, \mathrm{c}) = (\exists \mathrm{x}, \mathrm{y}, err.\ (\mathrm{y} > 2 \wedge \mathrm{x}' = \mathrm{y} + 1 \wedge \mathrm{y}' = \mathrm{y} \wedge err' = err))[V/V']$
$= (\mathrm{y}' > 2 \wedge \mathrm{x}' > 3)[V/V'] = (\mathrm{y} > 2 \wedge \mathrm{x} > 3)$

Exercise 1.8

- $sp(\mathrm{y} > 2 \wedge err = 0, \mathrm{x} := \mathtt{havoc}()) =$ $(\mathrm{y} > 2 \wedge err = 0)$
- $sp(\mathrm{y} > 2 \wedge err = 0, \mathtt{assume}(\mathrm{y} < 10)) =$ $(10 > \mathrm{y} > 2 \wedge err = 0)$
- $sp(\mathrm{y} > 2 \wedge err = 0, \mathtt{assert}(\mathrm{y} < 0)) =$

Symbolic sp

A formula in $\Sigma(V)$ represents a set of valuations.
Hence, we define symbolic sp that transforms formulas.

$$sp : \Sigma(V) \times \mathcal{P} \rightarrow \Sigma(V)$$

For data statements, the equivalent definition of symbolic sp is

$$sp(F, \mathtt{c}) \triangleq (\exists V : F \wedge \rho(\mathtt{c}))[V/V'].$$

Example 1.7

Let $V = [\mathtt{x}, \mathtt{y}, err]$ *and* $\mathtt{c} = \mathtt{x} := \mathtt{y} + 1$.
$sp(\mathtt{y} > 2, \mathtt{c}) = (\exists \mathtt{x}, \mathtt{y}, err.\ (\mathtt{y} > 2 \wedge \mathtt{x}' = \mathtt{y} + 1 \wedge \mathtt{y}' = \mathtt{y} \wedge err' = err))[V/V']$
$= (\mathtt{y}' > 2 \wedge \mathtt{x}' > 3)[V/V'] = (\mathtt{y} > 2 \wedge \mathtt{x} > 3)$

Exercise 1.8

- $sp(\mathtt{y} > 2 \wedge err = 0, \mathtt{x} := \mathtt{havoc}()) =\ (\mathtt{y} > 2 \wedge err = 0)$
- $sp(\mathtt{y} > 2 \wedge err = 0, \mathtt{assume}(\mathtt{y} < 10)) =\ (10 > \mathtt{y} > 2 \wedge err = 0)$
- $sp(\mathtt{y} > 2 \wedge err = 0, \mathtt{assert}(\mathtt{y} < 0)) =\ \top$

Symbolic sp for control statements

For control statements, the equivalent definitions of symbolic sp are

$$sp(F, c_1; c_2) \triangleq sp(sp(F, c_1), c_2)$$

Symbolic sp for control statements

For control statements, the equivalent definitions of symbolic sp are

$$sp(F, c_1; c_2) \triangleq sp(sp(F, c_1), c_2)$$
$$sp(F, c_1[]c_2) \triangleq sp(F, c_1) \vee sp(F, c_2)$$

Symbolic sp for control statements

For control statements, the equivalent definitions of symbolic sp are

$$sp(F, c_1; c_2) \triangleq sp(sp(F, c_1), c_2)$$
$$sp(F, c_1[]c_2) \triangleq sp(F, c_1) \lor sp(F, c_2)$$
$$sp(F, \texttt{if}(\texttt{F}_1)\ \texttt{c}_1\ \texttt{else}\ \texttt{c}_2) \triangleq sp(F, \texttt{assume}(\texttt{F}_1); c_1) \lor sp(F, \texttt{assume}(\neg\texttt{F}_1); c_2)$$

Symbolic sp for control statements

For control statements, the equivalent definitions of symbolic sp are

$$sp(F, c_1; c_2) \triangleq sp(sp(F, c_1), c_2)$$
$$sp(F, c_1[]c_2) \triangleq sp(F, c_1) \vee sp(F, c_2)$$
$$sp(F, \texttt{if}(\texttt{F}_1)\ \texttt{c}_1\ \texttt{else}\ \texttt{c}_2) \triangleq sp(F, \texttt{assume}(\texttt{F}_1); c_1) \vee sp(F, \texttt{assume}(\neg\texttt{F}_1); c_2)$$
$$sp(F, \texttt{while}(\texttt{G})\ \texttt{c}) \triangleq sp(lfp_{F'}(F \vee sp(F' \wedge \texttt{G}, \texttt{c})), \texttt{assume}(\neg\texttt{G}))$$

Symbolic sp for control statements

For control statements, the equivalent definitions of symbolic sp are

$$sp(F, c_1; c_2) \triangleq sp(sp(F, c_1), c_2)$$
$$sp(F, c_1[]c_2) \triangleq sp(F, c_1) \vee sp(F, c_2)$$
$$sp(F, \mathtt{if(F_1)\ c_1\ else\ c_2}) \triangleq sp(F, \mathtt{assume(F_1)}; c_1) \vee sp(F, \mathtt{assume(\neg F_1)}; c_2)$$
$$sp(F, \mathtt{while(G)\ c}) \triangleq sp(lfp_{F'}(F \vee sp(F' \wedge \mathtt{G}, \mathtt{c})), \mathtt{assume(\neg G)})$$

Example 1.8

$sp(x = 0, \mathtt{if(y > 0)\ x\ :=\ x + 1\ else\ x\ :=\ x - 1})$

Symbolic sp for control statements

For control statements, the equivalent definitions of symbolic sp are

$$sp(F, c_1; c_2) \triangleq sp(sp(F, c_1), c_2)$$
$$sp(F, c_1[]c_2) \triangleq sp(F, c_1) \vee sp(F, c_2)$$
$$sp(F, \texttt{if}(\texttt{F}_1)\ \texttt{c}_1\ \texttt{else}\ \texttt{c}_2) \triangleq sp(F, \texttt{assume}(\texttt{F}_1); c_1) \vee sp(F, \texttt{assume}(\neg\texttt{F}_1); c_2)$$
$$sp(F, \texttt{while}(\texttt{G})\ \texttt{c}) \triangleq sp(lfp_{F'}(F \vee sp(F' \wedge \texttt{G}, \texttt{c})), \texttt{assume}(\neg\texttt{G}))$$

Example 1.8

$sp(x = 0, \texttt{if}(\texttt{y} > 0)\ \texttt{x := x} + 1\ \texttt{else x := x} - 1)$
$= (y > 0 \wedge x = 1 \vee y \leq 0 \wedge x = -1)$

Symbolic sp for control statements

For control statements, the equivalent definitions of symbolic sp are

$$sp(F, c_1; c_2) \triangleq sp(sp(F, c_1), c_2)$$
$$sp(F, c_1[]c_2) \triangleq sp(F, c_1) \lor sp(F, c_2)$$
$$sp(F, \texttt{if}(\texttt{F}_1)\ \texttt{c}_1\ \texttt{else}\ \texttt{c}_2) \triangleq sp(F, \texttt{assume}(\texttt{F}_1); c_1) \lor sp(F, \texttt{assume}(\neg\texttt{F}_1); c_2)$$
$$sp(F, \texttt{while}(\texttt{G})\ \texttt{c}) \triangleq sp(lfp_{F'}(F \lor sp(F' \land \texttt{G}, \texttt{c})), \texttt{assume}(\neg\texttt{G}))$$

Example 1.8

$$sp(x = 0, \texttt{if}(\texttt{y} > \texttt{0})\ \texttt{x := x} + 1\ \texttt{else}\ \texttt{x := x} - 1)$$
$$= (y > 0 \land x = 1 \lor y \leq 0 \land x = -1)$$

Exercise 1.9

1. $sp(x + y > 0, \texttt{assume}(\texttt{x} > \texttt{0}); \texttt{y} := \texttt{y} + 1)$
2. $sp(y < 2, \texttt{while}(y < 10)\ \texttt{y} := \texttt{y} + 1)$
3. $sp(y > 2, \texttt{while}(y < 10)\ \texttt{y} := \texttt{y} + 1)$
4. $sp(y = 0, \texttt{while}(\top)\ \texttt{y} := \texttt{y} + 1)$

Safety and symbolic sp

Theorem 1.2

For a program c*, if* $\not\models sp(err = 0, c) \land err = 1$ *then* c *is safe.*

Exercise 1.10

Prove the above lemma.

Safety and symbolic sp

Theorem 1.2

For a program c*, if* $\not\models sp(err = 0, c) \wedge err = 1$ *then* c *is safe.*

Exercise 1.10

Prove the above lemma.

We need two key tools from logic to use *sp* as verification engine.

- quantifier elimination (for data statements)
- *lfp* computation (for loop statement)

Safety and symbolic sp

Theorem 1.2
For a program c*, if* $\not\models sp(err = 0, c) \wedge err = 1$ *then* c *is safe.*

Exercise 1.10
Prove the above lemma.

We need two key tools from logic to use *sp* as verification engine.

- quantifier elimination (for data statements)
- *lfp* computation (for loop statement)

There are quantifier elimination algorithms for many logical theories, e.g., integer arithmetic.

Safety and symbolic sp

Theorem 1.2
For a program c*, if* $\not\models sp(err = 0, c) \land err = 1$ *then* c *is safe.*

Exercise 1.10
Prove the above lemma.

We need two key tools from logic to use *sp* as verification engine.

- quantifier elimination (for data statements)
- *lfp* computation (for loop statement)

There are quantifier elimination algorithms for many logical theories, e.g., integer arithmetic.

However, there is no general algorithm for computing *lfp*. Otherwise, the halting problem is decidable.

Field of verification

This course is all about developing

incomplete but sound methods for lfp

that work for

some of the programs of our interest.

Weakest pre — dual of sp

Now we define a an operator that executes the programs backwards!

Definition 1.10

Weakest pre operator wp : $\mathfrak{p}(\mathbb{Q}^{|V|}) \times \mathcal{P} \to \mathfrak{p}(\mathbb{Q}^{|V|})$ *is defined as follows.*

$$wp(X, \mathtt{c}) \triangleq \{v | \forall v' : (v', \mathtt{skip}) \in T^*((v, \mathtt{c})) \Rightarrow v' \in X\},$$

where $X \subseteq \mathbb{Q}^{|V|}$ *and* $\mathtt{c}$ *is a program.*

Weakest pre — dual of sp

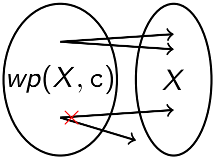
Now we define a an operator that executes the programs backwards!

Definition 1.10

Weakest pre operator $wp : \mathfrak{p}(\mathbb{Q}^{|V|}) \times \mathcal{P} \to \mathfrak{p}(\mathbb{Q}^{|V|})$ *is defined as follows.*

$$wp(X, \mathtt{c}) \triangleq \{v | \forall v' : (v', \mathtt{skip}) \in T^*((v, \mathtt{c})) \Rightarrow v' \in X\},$$

where $X \subseteq \mathbb{Q}^{|V|}$ *and* $\mathtt{c}$ *is a program.*



Weakest pre — dual of sp

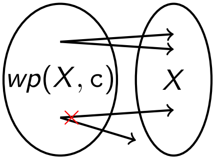
Now we define a an operator that executes the programs backwards!

Definition 1.10

Weakest pre operator $wp : \mathfrak{p}(\mathbb{Q}^{|V|}) \times \mathcal{P} \to \mathfrak{p}(\mathbb{Q}^{|V|})$ *is defined as follows.*

$$wp(X, \mathtt{c}) \triangleq \{v | \forall v' : (v', \mathtt{skip}) \in T^*((v, \mathtt{c})) \Rightarrow v' \in X\},$$

where $X \subseteq \mathbb{Q}^{|V|}$ *and* $\mathtt{c}$ *is a program.*



Example 1.9

Consider $V = [\mathtt{x}]$ *and* $X = \{[n] | 5 > n > 0\}$.

Weakest pre — dual of sp

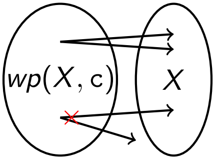
Now we define a an operator that executes the programs backwards!

Definition 1.10

Weakest pre operator wp $: \mathfrak{p}(\mathbb{Q}^{|V|}) \times \mathcal{P} \to \mathfrak{p}(\mathbb{Q}^{|V|})$ *is defined as follows.*

$$wp(X, \mathtt{c}) \triangleq \{v | \forall v' : (v', \mathtt{skip}) \in T^*((v, \mathtt{c})) \Rightarrow v' \in X\},$$

where $X \subseteq \mathbb{Q}^{|V|}$ *and* $\mathtt{c}$ *is a program.*



Example 1.9

Consider $V = [\mathtt{x}]$ *and* $X = \{[n] | 5 > n > 0\}$.

$wp(X, \mathtt{x} := \mathtt{x} + 1 [] \mathtt{x} := \mathtt{x} - 1) =$

Weakest pre — dual of sp

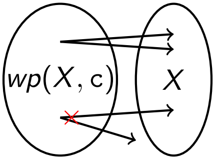
Now we define a an operator that executes the programs backwards!

Definition 1.10

Weakest pre operator $wp : \mathfrak{p}(\mathbb{Q}^{|V|}) \times \mathcal{P} \rightarrow \mathfrak{p}(\mathbb{Q}^{|V|})$ *is defined as follows.*

$$wp(X, \mathtt{c}) \triangleq \{v | \forall v' : (v', \mathtt{skip}) \in T^*((v, \mathtt{c})) \Rightarrow v' \in X\},$$

where $X \subseteq \mathbb{Q}^{|V|}$ *and* $\mathtt{c}$ *is a program.*



Example 1.9

Consider $V = [\mathtt{x}]$ *and* $X = \{[n] | 5 > n > 0\}$.
$wp(X, \mathtt{x} := \mathtt{x} + 1 [] \mathtt{x} := \mathtt{x} - 1) = \{[n] | 4 > n > 1\}$

Weakest pre — dual of sp

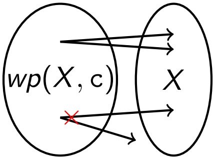
Now we define a an operator that executes the programs backwards!

Definition 1.10

Weakest pre operator $wp : \mathfrak{p}(\mathbb{Q}^{|V|}) \times \mathcal{P} \to \mathfrak{p}(\mathbb{Q}^{|V|})$ *is defined as follows.*

$$wp(X, \mathtt{c}) \triangleq \{v | \forall v' : (v', \mathtt{skip}) \in T^*((v, \mathtt{c})) \Rightarrow v' \in X\},$$

where $X \subseteq \mathbb{Q}^{|V|}$ *and* $\mathtt{c}$ *is a program.*



Example 1.9

Consider $V = [\mathtt{x}]$ *and* $X = \{[n] | 5 > n > 0\}$.
$wp(X, \mathtt{x} := \mathtt{x} + 1 [] \mathtt{x} := \mathtt{x} - 1) = \{[n] | 4 > n > 1\}$

Exercise 1.11

Why use of word "weakest"?

Logical weakest pre

We define symbolic wp that transforms formulas.

$$wp : \Sigma(V) \times \mathcal{P} \to \Sigma(V)$$

Logical weakest pre

We define symbolic wp that transforms formulas.

$$wp : \Sigma(V) \times \mathcal{P} \to \Sigma(V)$$

The equivalent definition of symbolic wp for data statements are

$$wp(F, \texttt{x := exp}) \triangleq F[\texttt{exp}/\texttt{x}]$$

Logical weakest pre

We define symbolic wp that transforms formulas.

$$wp : \Sigma(V) \times \mathcal{P} \to \Sigma(V)$$

The equivalent definition of symbolic wp for data statements are

$$wp(F, \texttt{x} \ := \ \texttt{exp}) \triangleq F[\texttt{exp}/\texttt{x}]$$
$$wp(F, \texttt{x} \ := \ \texttt{havoc()}) \triangleq \forall x.F$$

Logical weakest pre

We define symbolic wp that transforms formulas.

$$wp : \Sigma(V) \times \mathcal{P} \rightarrow \Sigma(V)$$

The equivalent definition of symbolic wp for data statements are

$$
\begin{aligned}
wp(F, \texttt{x := exp}) &\triangleq F[\texttt{exp}/\texttt{x}]\\
wp(F, \texttt{x := havoc()}) &\triangleq \forall x. F\\
wp(F, \texttt{assume(G)}) &\triangleq G \Rightarrow F
\end{aligned}
$$

Logical weakest pre

We define symbolic wp that transforms formulas.

$$wp : \Sigma(V) \times \mathcal{P} \to \Sigma(V)$$

The equivalent definition of symbolic wp for data statements are

$$\begin{aligned}
wp(F, \mathtt{x\ :=\ exp}) &\triangleq F[\mathtt{exp}/\mathtt{x}]\\
wp(F, \mathtt{x\ :=\ havoc()}) &\triangleq \forall x.F\\
wp(F, \mathtt{assume(G)}) &\triangleq G \Rightarrow F\\
wp(F, \mathtt{assert(G)}) &\triangleq G \wedge F
\end{aligned}$$

Logical weakest pre

We define symbolic wp that transforms formulas.

$$wp : \Sigma(V) \times \mathcal{P} \to \Sigma(V)$$

The equivalent definition of symbolic wp for data statements are

$$\begin{aligned}
wp(F, \mathtt{x} := \mathtt{exp}) &\triangleq F[\mathtt{exp}/\mathtt{x}] \\
wp(F, \mathtt{x} := \mathtt{havoc()}) &\triangleq \forall x. F \\
wp(F, \mathtt{assume(G)}) &\triangleq G \Rightarrow F \\
wp(F, \mathtt{assert(G)}) &\triangleq G \wedge F
\end{aligned}$$

Example 1.10

- $wp((i \leq 3 \wedge r = (i-1)z + 1), \mathtt{i} := 1) =$
- $wp((i < 3 \wedge r = iz + 1), \mathtt{r} := \mathtt{r} + \mathtt{z}) =$
- $wp(x < 0, \mathtt{assume}(\mathtt{x} > 0)) =$

Logical weakest pre

We define symbolic wp that transforms formulas.

$$wp : \Sigma(V) \times \mathcal{P} \rightarrow \Sigma(V)$$

The equivalent definition of symbolic wp for data statements are

$$\begin{aligned}
wp(F, \texttt{x := exp}) &\triangleq F[\texttt{exp}/\texttt{x}]\\
wp(F, \texttt{x := havoc()}) &\triangleq \forall x. F\\
wp(F, \texttt{assume(G)}) &\triangleq G \Rightarrow F\\
wp(F, \texttt{assert(G)}) &\triangleq G \wedge F
\end{aligned}$$

Example 1.10

- $wp((i \leq 3 \wedge r = (i-1)z + 1), \texttt{i := 1}) =$
- $wp((i < 3 \wedge r = iz + 1), \texttt{r := r + z}) =$
- $wp(x < 0, \texttt{assume(x > 0)}) =$

Logical weakest pre

The equivalent definition of symbolic wp for control statements are

$wp(F, c_1; c_2) \triangleq wp(wp(F, c_2), c_1)$

Logical weakest pre

The equivalent definition of symbolic wp for control statements are

$$wp(F, c_1; c_2) \triangleq wp(wp(F, c_2), c_1)$$
$$wp(F, c_1 [] c_2) \triangleq wp(F, c_1) \land wp(F, c_2)$$

Logical weakest pre

The equivalent definition of symbolic wp for control statements are

$wp(F, c_1; c_2) \triangleq wp(wp(F, c_2), c_1)$
$wp(F, c_1[]c_2) \triangleq wp(F, c_1) \land wp(F, c_2)$
$wp(F, \texttt{if}(\texttt{F}_1)\ \texttt{c}_1\ \texttt{else}\ \texttt{c}_2) \triangleq wp(F, \texttt{assume}(\texttt{F}_1); c_1) \land wp(F, \texttt{assume}(\neg\texttt{F}_1); c_2)$

Logical weakest pre

The equivalent definition of symbolic wp for control statements are

$wp(F, c_1; c_2) \triangleq wp(wp(F, c_2), c_1)$
$wp(F, c_1[]c_2) \triangleq wp(F, c_1) \land wp(F, c_2)$
$wp(F, \texttt{if}(\texttt{F}_1)\ \texttt{c}_1\ \texttt{else}\ \texttt{c}_2) \triangleq wp(F, \texttt{assume}(\texttt{F}_1); c_1) \land wp(F, \texttt{assume}(\neg\texttt{F}_1); c_2)$
$wp(F, \texttt{while}(G)c) \triangleq gfp_{F'}((G \lor F) \land wp(F', \texttt{assume(G)}; c))$

Logical weakest pre

The equivalent definition of symbolic wp for control statements are

$$wp(F, c_1; c_2) \triangleq wp(wp(F, c_2), c_1)$$
$$wp(F, c_1[]c_2) \triangleq wp(F, c_1) \wedge wp(F, c_2)$$
$$wp(F, \texttt{if}(\texttt{F}_1)\ \texttt{c}_1\ \texttt{else}\ \texttt{c}_2) \triangleq wp(F, \texttt{assume}(\texttt{F}_1); c_1) \wedge wp(F, \texttt{assume}(\neg\texttt{F}_1); c_2)$$
$$wp(F, \texttt{while}(G)c) \triangleq gfp_{F'}((G \vee F) \wedge wp(F', \texttt{assume}(\texttt{G}); c))$$

Lemma 1.1

For a program c, if $err = 0 \Rightarrow wp(err = 0, c)$ is valid then c *is safe.*

Exercise 1.12

Prove the above lemma.

Logical weakest pre

The equivalent definition of symbolic wp for control statements are

$$wp(F, c_1; c_2) \triangleq wp(wp(F, c_2), c_1)$$
$$wp(F, c_1[]c_2) \triangleq wp(F, c_1) \land wp(F, c_2)$$
$$wp(F, \texttt{if}(\texttt{F}_1)\ \texttt{c}_1\ \texttt{else}\ \texttt{c}_2) \triangleq wp(F, \texttt{assume}(\texttt{F}_1); c_1) \land wp(F, \texttt{assume}(\neg\texttt{F}_1); c_2)$$
$$wp(F, \texttt{while}(G)c) \triangleq gfp_{F'}((G \lor F) \land wp(F', \texttt{assume}(\texttt{G}); c))$$

Lemma 1.1

For a program c, if $err = 0 \Rightarrow wp(err = 0, c)$ is valid then c *is safe.*

Exercise 1.12

Prove the above lemma.

Note: Our definition of wp is usually called weakest liberal precondition(wlp)

Assignment

Exercise 1.13 (Assignment 1)

1. *(.5) Example 1.10*
2. *(.5) Discuss weakest precondition(wp) vs. weakest liberal precondition(wlp)*
3. *(1) Exercise 1.4*
4. *(1) Show* $sp(wp(F, \mathrm{c}), \mathrm{c}) \subseteq F \subseteq wp(sp(F, \mathrm{c}), \mathrm{c})$
5. *(1) Write a C++ program that reads a SMT2 formula from command line and performs quantifier elimination using Z3 for the variables that do not end with '*

End of Lecture 1