Software Architecture and Engineering: Part II

ETH Zurich, Spring 2014

Prof. Martin Vechev

SRL
SOFTWARE RELIABILITY LAB

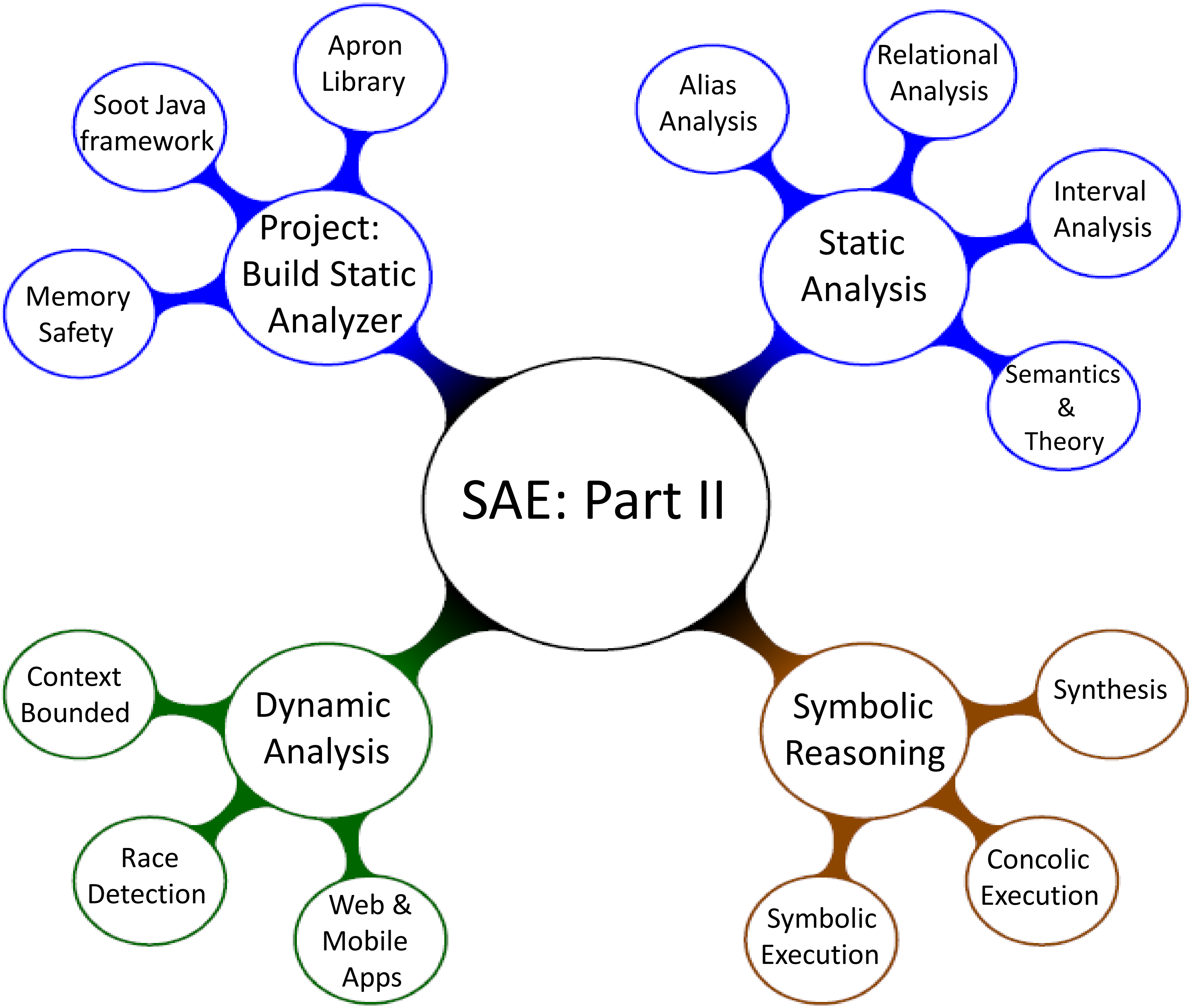
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

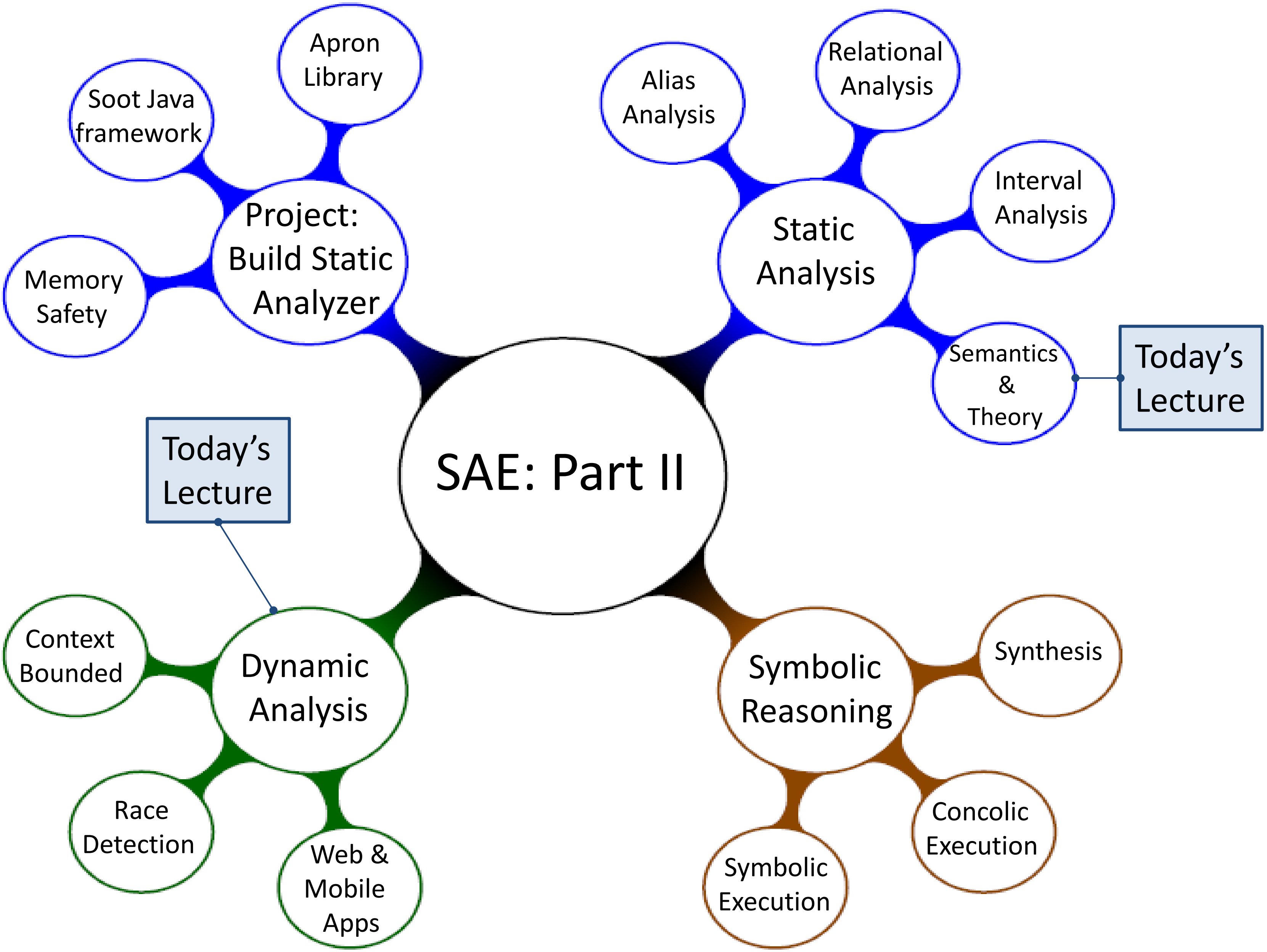
Announcements

- Andrei Dan’s exercise group merges into the other groups.

- Lectures slides will be uploaded typically a day before the lecture

- Anonymous Feedback form: http://tinyurl.com/ogbvbfx


SAE: Part II
Project: Build Static Analyzer
Soot Java framework
Apron Library
Memory Safety
Static Analysis
Alias Analysis
Relational Analysis
Interval Analysis
Semantics & Theory
Dynamic Analysis
Context Bounded
Race Detection
Web & Mobile Apps
Symbolic Reasoning
Synthesis
Concolic Execution
Symbolic Execution


SAE: Part II
Project: Build Static Analyzer
Soot Java framework
Apron Library
Memory Safety
Static Analysis
Alias Analysis
Relational Analysis
Interval Analysis
Semantics & Theory
Today's Lecture
Today's Lecture
Dynamic Analysis
Context Bounded
Race Detection
Web & Mobile Apps
Symbolic Reasoning
Synthesis
Concolic Execution
Symbolic Execution

Semantics

- Why Formal Semantics?
- Syntax of a SPL language
- Operational Semantics of SPL

Why Formal Semantics?

- Would this C program seg fault ?

```
int main(void ) {
    *(char*) NULL;
     return 0;
}
```

Why Formal Semantics?

- Can this C program ever enter the branch ?

```
int x;
…
if (x + 1 < x)
{
  printf("Overflow");
}
```

Why Formal Semantics?

- What does this C program return ?

```
int main(void ) {
   int x  = 0;
   return (x = 1) + (x = 2);
}
```

Why Formal Semantics?

- Is there a division-by-zero in this C program ?

```
int d  = 5;
int setDenom(int x) {
   return d = x;
}

int main(void ) {
   return (10/d) + setDenom(0);
}
```

Why Formal Semantics?

- Understand what a program does
- Implement a language
  - generate an interpreter/compiler
- Reasoning about program correctness
  - if you don't know what it does, how do you know its correct?

Semantics = assigning meaning to programs

“mathematical models of and methods for describing and reasoning about the behavior of programs”

The SPL Language: Syntax

$x \in$ Var set of integer variables
$v \in$ Z set of integer constants
$\ell \in$ Lab set of labels

$a \in$ AExp set of arithmetic expressions
$b \in$ BExp set of boolean expressions
$s \in$ Stmt set of statements

x, a, b, s are called meta-variables

$$a ::= v \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2$$

$$b ::= \texttt{true} \mid \texttt{false} \mid \neg\, b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid a_1 = a_2 \mid a_1 \leq a_2$$

$$s ::= x := a^{\ell} \mid \texttt{skip}^{\ell} \mid s_1 ; s_2 \mid \texttt{if } b^{\ell} \texttt{ then } s_1 \texttt{ else } s_2 \mid \texttt{while } b^{\ell} \texttt{ do } s$$

- variables are not declared
- expressions have no side-effects, all side-effects in statements
- only basic statements: no functions, heap, exceptions,…
- semantics usually specified at abstract syntax level

Sample Programs

- Is this a SPL program ?

```
x := 5;
while (0 ≤ x) do
  x := x – 1;
```

Sample Programs

- Is this a SPL program ?

```
x := 5;
while (0 ≤ x) do
  x := x – 1;
```

YES

Sample Programs

- Is this a SPL program ?

```
x := true + 5;
while (0 ≤ x) do
  x := x - 1;
```

Sample Programs

- Is this a SPL program ?

```
x := true + 5;
while (0 ≤ x) do
  x := x - 1;
```

NO

Sample Programs

- Is this a SPL program ?

```
x := 5;
while (false) do
  x := false;
```

Sample Programs

- Is this a SPL program ?

```
x := 5;
while (false) do
  x := false;
```

NO

Semantics: main questions

- What is the meaning of an expression?
- What is the meaning of a statement?
- How is such a meaning defined?

Semantics: three approaches

- Operational Semantics
  - How would I execute the statement ?
- Denotational Semantics
  - What is the statement computing ?
- Axiomatic Semantics
  - What is true after a statement is executed ?

Semantics: three approaches

- Operational Semantics
  - define a transition system, transition relation describes evaluation steps of a program
- Denotational Semantics
  - define an input/output relation that assigns meaning to each construct (denotation)
- Axiomatic Semantics
  - define the effect of each construct on logical statements about program store (assertions)

Operational Semantics

```
int double1(int x) {
  int t = 0;
  t = t + x;
  t = t + x;
  return t;
}
```

$x \longmapsto 2$

$[t \longmapsto 0, x \longmapsto 2]$

$[t \longmapsto 2, x \longmapsto 2]$

$[t \longmapsto 4, x \longmapsto 2]$

$[t \longmapsto 4, x \longmapsto 2]$

```
int double2(int x) {
  int t = 2*x;
  return t;
}
```

$[t \longmapsto 4, x \longmapsto 2]$

Denotational Semantics

```
int double1(int x) {
  int t = 0;
  t = t + x;
  t = t + x;
  return t;
}
```

$$\lambda x.2 * x$$

```
int double2(int x) {
  int t = 2*x;
  return t;
}
```

$$\lambda x.2 * x$$

Axiomatic Semantics

```
int double1(int x) {
  { x = x0 }
  int t = 0;
  { x = x0 ∧ t  = 0 }
  t = t + x;
  {x = x0 ∧ t  = x0 }
  t = t + x;
  {x = x0 ∧ t  = 2*x0 }
  return t;
}
```

```
int double2(int x) {
 { x = x0 }
 int t = 2x;
 { x = x0 ∧ t  = 2*x0 }
 return t;
}
```

Next: operational semantics

Operational Semantics

- Specifies how expressions and statements should be evaluated

- Evaluation depends on the shape of the expression/statement:
  - `1`, `2`, 3, … do not evaluate any further
  - `x + y` is evaluated further

- Think of it as an interpreter

Operational Semantics

- Evaluation depends on values of variables
  - what does x + y evaluate to ?
  - depends on the values of x and y

- Values of variables at any moment in time are given by a function $\sigma \in$ Store = Var $\longrightarrow$ Z
  - Z is the set of integers
  - to simplify presentation we assume Store denotes total functions
  - if σ is such that $x \mapsto 5$ and $y \mapsto 3$, then $x + y$ is 8

Operational Semantics for SPL

- Configurations: $c \in \Sigma$ where $\Sigma = (\text{Stmt} \times \text{Store}) \cup \text{Store}$
  - $\langle S, \sigma\rangle$ is a configuration
  - σ is also a configuration: a terminal configuration. All other configurations are non-terminal

- Transitions: $\rightarrow \subseteq \Sigma \times \Sigma$
  - steps between configurations

- Transition system: $(\Sigma, \rightarrow, I, F)$
  - $I \subseteq \Sigma$: initial configurations
  - $F \subseteq \text{Store}$: final configurations

Operational Semantics for SPL

- We write $c \longrightarrow c'$ when $(c, c') \in \longrightarrow$

- $\longrightarrow^*$ denotes the reflexive transitive closure of the relation $\longrightarrow$. We say $c \longrightarrow^* c'$ when:
  - $c = c_0$ and $c_n = c'$
  - there is a sequence $c_0 \longrightarrow c_1 \longrightarrow \ldots c_n$ for some $n \geq 0$

Notation: Rules of Inference

These are called evaluation rules

$$\frac{\text{Hypothesis}_1 \ \dots \ \text{Hypothesis}_n}{\text{Conclusion}}$$

Example:

$$\frac{\text{A is true} \qquad \text{B is true}}{\text{A} \land \text{B is true}}$$

Evaluation rules with no premises are called axioms

$$\frac{}{\text{Conclusion}}$$

Next: operational semantics of SPL

Operational Semantics of SPL

- There are two kinds: big-step and small-step

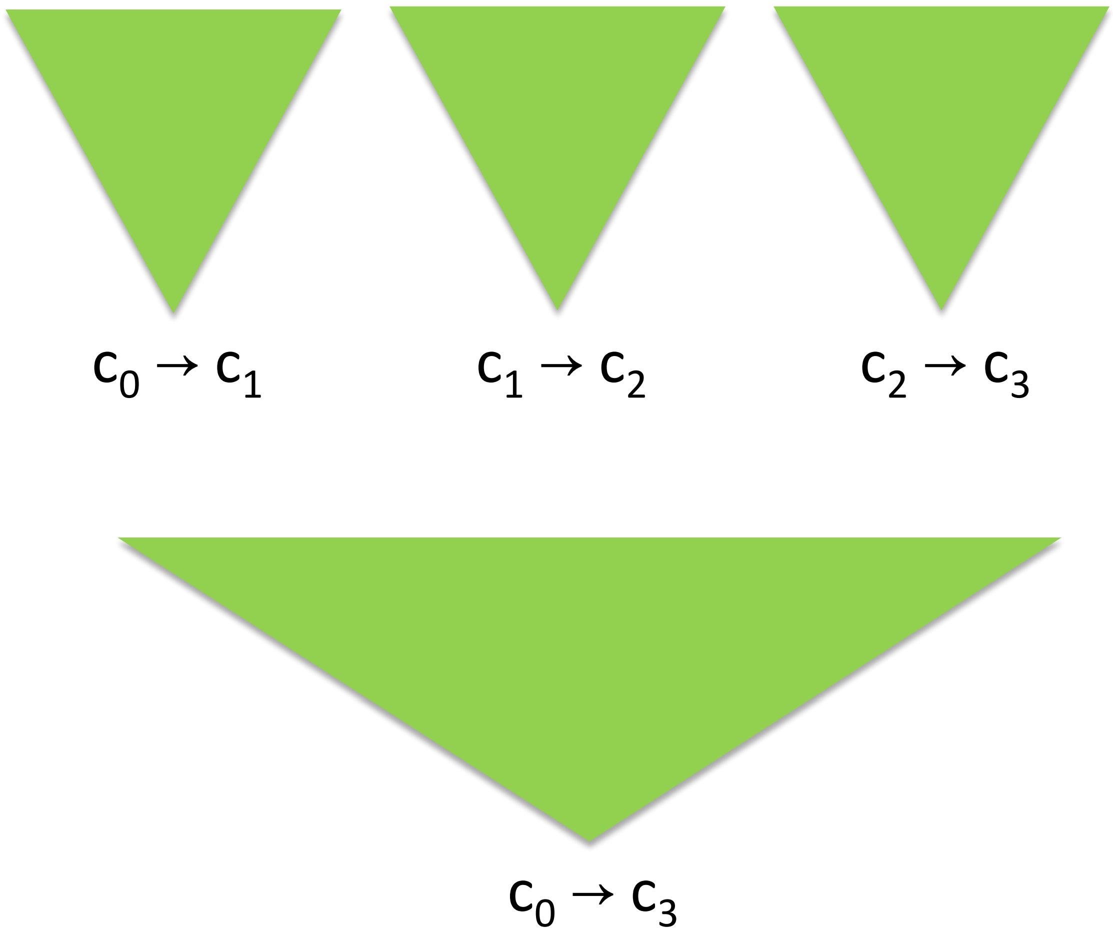
- Big-step
  - $c \longrightarrow c'$ describes the entire computation

- Small-step
  - $c \longrightarrow c'$ describes a single step of a larger computation

Small Step vs. Big Step

small step

big step



Operational Semantics of SPL

Next, we will give semantics of SPL. The statements will be evaluated in a small-step style, while the expressions will be evaluated in big-step style.

Auxiliary Relations

- To describe the semantics of AExp and BExp we use two auxiliary relations

$$\texttt{for AExp: } \Downarrow_a \subseteq (\texttt{AExp} \times \texttt{Store}) \times \texttt{Z}$$
$$\texttt{for BExp: } \Downarrow_b \subseteq (\texttt{BExp} \times \texttt{Store}) \times \{\texttt{true, false}\}$$

- Judgments such as

$$\langle a, \sigma \rangle \Downarrow_a v$$

are read as: "expression *a* evaluates to *v* in store σ"
Boolean expressions read similarly

Evaluation rules for AExp

$$\frac{\langle a_1, \sigma\rangle \Downarrow_a v_1 \qquad \langle a_2, \sigma\rangle \Downarrow_a v_2}{\langle a_1 + a_2, \sigma\rangle \Downarrow_a v_1 + v_2}$$

$$\frac{}{\langle x, \sigma\rangle \Downarrow_a \sigma(x)}$$

Evaluation rules for BExp

$$\frac{\langle a_1, \sigma\rangle \Downarrow_a v_1 \qquad \langle a_2, \sigma\rangle \Downarrow_a v_2}{\langle a_1 \leq a_2, \sigma\rangle \Downarrow_b bv} \quad bv \text{ is } v_1 \leq v_2$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow_a v_1 \qquad \langle a_2, \sigma\rangle \Downarrow_a v_2}{\langle a_1 = a_2, \sigma\rangle \Downarrow_b bv} \quad bv \text{ is } v_1 == v_2$$

$$\frac{}{\langle b_1 \land b_2, \sigma\rangle \Downarrow_b ???}$$

What about this ?

Evaluation rules for BExp

$$\frac{\langle a_1, \sigma\rangle \Downarrow_a v_1 \qquad \langle a_2, \sigma\rangle \Downarrow_a v_2}{\langle a_1 \leq a_2, \sigma\rangle \Downarrow_b bv} \quad bv \text{ is } v_1 \leq v_2$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow_a v_1 \qquad \langle a_2, \sigma\rangle \Downarrow_a v_2}{\langle a_1 = a_2, \sigma\rangle \Downarrow_b bv} \quad bv \text{ is } v_1 == v_2$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow_b \text{true} \qquad \langle b_2, \sigma\rangle \Downarrow_b \text{true}}{\langle b_1 \wedge b_2, \sigma\rangle \Downarrow_b \text{true}}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow_b \text{false}}{\langle b_1 \wedge b_2, \sigma\rangle \Downarrow_b \text{false}} \qquad \frac{\langle b_2, \sigma\rangle \Downarrow_b \text{false}}{\langle b_1 \wedge b_2, \sigma\rangle \Downarrow_b \text{false}}$$

short-circuit evaluation

How to read the rules

- Top-down: like inference rules
  - If we know hypothesis holds, conclusion holds
  - If $\langle x, \sigma \rangle \Downarrow_a 5$ and $\langle y, \sigma \rangle \Downarrow_a 6$ then $\langle x + y, \sigma \rangle \Downarrow_a 11$

- Bottom-up: read by inversion
  - Suppose we want to evaluate $\langle x + y, \sigma \rangle \Downarrow_a 11$
  - Lets look at rules with conclusion that has $\langle x + y, \sigma \rangle$
  - Here: only 1 rule has it as a conclusion (the addition rule)
  - Repeat a recursive tree-walk

Example: Derivation Tree

Evaluate this: $\langle\, (x + 3) * (y + 4)\, ,\, \sigma \rangle$ where σ: $x \mapsto 1$, $y \mapsto 2$

Example: Derivation Tree

Evaluate this: $\langle\, (x + 3) * (y + 4) \,,\, \sigma \rangle$ where σ: $x \mapsto 1$, $y \mapsto 2$

$$\frac{}{\langle x , \sigma \rangle \Downarrow_a 1}$$

$$\frac{}{\langle y , \sigma \rangle \Downarrow_a 2}$$

$$\frac{\langle x , \sigma \rangle \Downarrow_a 1 \qquad \langle 3 , \sigma \rangle \Downarrow_a 3}{\langle x + 3 , \sigma \rangle \Downarrow_a 4}$$

$$\frac{\langle y , \sigma \rangle \Downarrow_a 2 \qquad \langle 4 , \sigma \rangle \Downarrow_a 4}{\langle y + 4 , \sigma \rangle \Downarrow_a 6}$$

$$\frac{\langle x + 3 , \sigma \rangle \Downarrow_a 4 \qquad \langle y + 4 , \sigma \rangle \Downarrow_a 6}{\langle\, (x + 3) * (y + 4) \,,\, \sigma \rangle \Downarrow_a 24}$$

Evaluation of Statements

- Evaluating a statement produces a new store
  - $\langle s , \sigma \rangle \rightarrow \langle s' , \sigma' \rangle$
- Evaluation order is important
  - In s_1 ; s_2 s_1 is evaluated before s_2
  - In if true then s_1 else s_2 s_2 is not evaluated
- Some constructs have multiple rules
  - conditionals and while

Evaluation rules for Stmt I

$$\frac{\langle s_1 , \sigma \rangle \to \langle s_2 , \sigma_1 \rangle}{\langle s_1 \,;\, s_3 , \sigma \rangle \to \langle s_2 \,;\, s_3 , \sigma_1 \rangle}$$

$$\frac{\langle s_1 , \sigma \rangle \to \sigma_1}{\langle s_1 \,;\, s_2 , \sigma \rangle \to \langle s_2 , \sigma_1 \rangle}$$

$$\frac{}{\langle \text{skip}, \sigma \rangle \to \sigma}$$

$$\frac{\langle a , \sigma \rangle \Downarrow_a v}{\langle x := a, \sigma \rangle \to \langle x := v , \sigma \rangle}$$

$$\frac{}{\langle x := v, \sigma \rangle \to \sigma[x \mapsto v]}$$

assignment not a single step

Evaluation rules for Stmt II

$$\frac{}{\langle \text{if true then } s_1 \text{ else } s_2, \sigma\rangle \rightarrow \langle s_1, \sigma\rangle}$$

$$\frac{}{\langle \text{if false then } s_1 \text{ else } s_2, \sigma\rangle \rightarrow \langle s_2, \sigma\rangle}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow_b bv}{\langle \text{if } b_1 \text{ then } s_1 \text{ else } s_2, \sigma\rangle \rightarrow \langle \text{if } bv \text{ then } s_1 \text{ else } s_2, \sigma\rangle}$$

$$\frac{}{\langle \text{while } b \text{ do } s, \sigma\rangle \rightarrow ???}$$

Evaluation rules for Stmt II

$$\frac{}{\langle \text{if true then } s_1 \text{ else } s_2, \sigma\rangle \rightarrow \langle s_1, \sigma\rangle}$$

$$\frac{}{\langle \text{if false then } s_1 \text{ else } s_2, \sigma\rangle \rightarrow \langle s_2, \sigma\rangle}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow_b bv}{\langle \text{if } b_1 \text{ then } s_1 \text{ else } s_2, \sigma\rangle \rightarrow \langle \text{if } bv \text{ then } s_1 \text{ else } s_2, \sigma\rangle}$$

$$\frac{}{\langle \text{while } b \text{ do } s, \sigma\rangle \rightarrow \langle \text{if } b \text{ then } (s; \text{ while } b \text{ do } s) \text{ else skip}, \sigma\rangle}$$

‘while’ expressed in terms of ‘if’

Sequences

Note that for a program S_0 the steps are formed via the relation $\rightarrow$

That is, sequences are $\langle S_0, \sigma_0 \rangle \rightarrow \langle S_1, \sigma_1 \rangle \rightarrow \ldots$

The relations $\Downarrow_a$ or $\Downarrow_b$ are only used to justify the step with $\rightarrow$

In other words, $\Downarrow_a$ or $\Downarrow_b$ are only used to build the relation $\rightarrow$

we understand semantics…
but what do we do with them ?

next: Program Analysis

What is the meaning of this code?

$$
\begin{array}{l}
\texttt{y := x}^{1}\texttt{;} \\
\texttt{z := 1}^{2}\texttt{;} \\
\texttt{while y > 0}^{3}\texttt{ do} \\
\quad\quad \texttt{z := z * y}^{4}\texttt{;} \\
\quad\quad \texttt{y := y - 1}^{5}\texttt{;} \\
\;\texttt{;} \\
\texttt{y := 0}^{6}
\end{array}
$$

Lets look at its traces

A program trace

$\langle$y:=x¹;z:=1²; while y > 0³ do z:=z∗y⁴; y:=y−1⁵; y:=0⁶, { x↦42, y↦0, z↦0 }$\rangle$

y := x¹

→ $\langle$z:=1²; while y > 0³ do z:=z∗y⁴; y:=y−1⁵; y:=0⁶, { x↦42, y↦42, z↦0 }$\rangle$

z := 1²

→ $\langle$while y > 0³ do z:=z∗y⁴; y:=y−1⁵; y:=0⁶, { x↦42, y↦42, z↦1 }$\rangle$

y > 0³

→ $\langle$z:=z∗y⁴; y:=y−1⁵; while y > 0³ do z:=z∗y⁴; y:=y−1⁵; y:=0⁶, {x↦42, y↦42, z↦1 }$\rangle$

→ …

Note: some steps are not shown.

Trace Semantics

- Trace semantics are the set of all program traces starting from initial configurations

$$[\![P]\!] = \{ c_0 \cdot c_1 \cdot \ldots \cdot c_{n-1} \mid n \geq 1 \land c_0 \in I \land \forall i \in [0, n-2]: c_i \rightarrow c_{i+1} \}$$

- Note that traces need not end in final configurations

- Traces are of finite length, but the number of initial configurations can be infinite. Hence, an infinite number of traces: computation is non-feasible

Approaches to Program Analysis