Lecture 3: Semantics of Programming Languages

Concurrency Theory

Summer 2024

Dr. Stephan Mennicke

April 16th, 2024

TU Dresden, Knowledge-Based Systems Group

Review

Overview

Part 0: Completing the Introduction

- learning about *bisimilarity* and *bisimulations*

Part 1: Semantics of (Sequential) Programming Languages

- WHILE – an old friend (**today**)
- denotational semantics (a baseline and an exercise of the inductive method) (**also today**)
- natural semantics and (structural) operational semantics

Part 2: Towards Parallel Programming Languages

- bisimilarity and its success story
- deep-dive into induction and coinduction
- algebraic properties of bisimilarity

Part 3: Expressive Power

- Calculus of Communicating Systems (CCS)
- Petri nets

Semantics of Programming Languages

- sometimes, *pragmatics* included (not here :))

Syntax

- grammatical structure of programs

Example 1: The program

```
z := x; x := y; y := z
```

consists of three *statements* (separated by `;`). Each statement has the form of a variable followed by `:=` and an expression.

Semantics

- is about specifying the *meaning*, or *behavior*, of programs, hardware, or systems in general
  - to reveal ambiguities
  - to form the basis for implementation, analysis, and verification
- meaning of grammatically correct programs

Example 2: The meaning of the program

```
z := x; x := y; y := z
```

is the exchange of values of variables `x` and `y` (whereas the value of `z` is set to the final value of `y`).

- for a formal treatment we need to explain the meanings of
  - sequences of statements and
  - statements that are sequences of variables, `:=`, and expressions.

The Flavors of Semantics

Operational Semantics

- meaning = computation induced by the syntactic constructs
- it is important *how?* the effect of computation is produced

Denotational Semantics

- meaning = mathematical object that captures the effect of executing the program
- *only* the effect is important, not how it was obtained

Axiomatic Semantics

- properties of the effect of executing the program expressed as *assertions*
- some aspects of the computation may be neglected

Operational Semantics by Example

```
z := x; x := y; y := z
```

- how to execute the code?
  - execution of a sequence of statements (separated by `;`) is execution of individual statements one after the other
  - execution of statements with variable follows by `:=` followed by an expression means determining the value of the expression and assigning it to the first variable
- record the execution of programs in a *state* where x has value 5, y has value 7, and z has value 0:

$$\begin{aligned} & \langle \texttt{z := x; x := y; y := z}, [x \mapsto 5, y \mapsto 7, z \mapsto 0] \rangle \\ \Rightarrow\ & \langle \texttt{x := y; y := z}, [x \mapsto 5, y \mapsto 7, z \mapsto 5] \rangle \\ \Rightarrow\ & \langle \texttt{y := z}, [x \mapsto 7, y \mapsto 7, z \mapsto 5] \rangle \\ \Rightarrow\ & [x \mapsto 7, y \mapsto 5, z \mapsto 5] \end{aligned}$$

Another Operational Semantics by Example

```
z := x; x := y; y := z
```

- the semantics so far abstracted from the computing architecture (e.g., memory locations)
- we can even go further by so-called derivation trees:

$$\dfrac{\dfrac{\langle \texttt{z := x}, s_0\rangle \to s_1 \quad \langle \texttt{x := y}, s_1\rangle \to s_2}{\langle \texttt{z := x; x := y}, s_0\rangle \to s_2} \qquad \langle \texttt{y := z}, s_2\rangle \to s_3}{\langle \texttt{z := x; x := y; y := z}, s_0\rangle \to s_3}$$

where $s_0 = [x \mapsto 5, y \mapsto 7, z \mapsto 0]$, $s_1 = [x \mapsto 5, y \mapsto 7, z \mapsto 5]$, $s_2 = [x \mapsto 7, y \mapsto 7, z \mapsto 5]$, and $s_3 = [x \mapsto 7, y \mapsto 5, z \mapsto 5]$.

- this style is called the *natural semantics* or *big step semantics*

```
z := x; x := y; y := z
```

- the *effect* of the computation is modeled by mathematical functions:
- the effect of a sequence of statements is the function composition of the individual effects
- the effect of a statement consisting of a variable, followed by `:=` and an expression is the function that takes a *state* (i.e., a mapping from variables to values) and transforms it into a state mapping the variable in question to its new value
- for the example we get $\mathcal{S}[\![\texttt{z := x}]\!]$, $\mathcal{S}[\![\texttt{x := y}]\!]$, and $\mathcal{S}[\![\texttt{y := z}]\!]$ to obtain the meaning

$$\mathcal{S}[\![\texttt{z := x; x := y; y := z}]\!] = \mathcal{S}[\![\texttt{y := z}]\!] \circ \mathcal{S}[\![\texttt{x := y}]\!] \circ \mathcal{S}[\![\texttt{z := x}]\!]$$

Remark on Order and Function Composition

Function composition is read in the reverse order: Functions $g : A \to B$ and $f : B \to C$ compose to $f \circ g$ such that for all $x \in A$, $(f \circ g)(x) := f(g(x))$.

Axiomatic Semantics by Example

$$\{x = n \land y = m\}\texttt{z := x; x := y; y := z}\{x = m \land y = n\}$$

- precondition ($\{x = n \land y = m\}$) and postcondition ($\{x = m \land y = n\}$)
- viewed as a specification focusing on particular aspect of the semantics
- *partial correctness* (i.e., upon termination) and *total correctness*
- once again, a derivation tree is appropriate
- axiomatic semantics tells us how to step-wise transform preconditions into postconditions:

$$[\text{ass}]\,\frac{}{\{P[x \mapsto n]\}\texttt{x := }n\{P\}}$$

$$[\text{comp}]\,\frac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}\ S_1\,;S_2\ \{R\}}$$

The Language of WHILE-Programs

Syntactic Categories

The following categories are pairwise disjoint sets.

- **Num** is the set of numerals (e.g., $n, n_1, n_2, \ldots$)
- **Var** is the set of variables (e.g., $x, y, z, \ldots$)
- **Aexp** is the set of arithmetic expressions (e.g., $a, a_1 \star a_2, \ldots$)
- **Bexp** is the set of Boolean expressions (e.g., **true**, $\neg b$, $a_1 < a_2, \ldots$)
- **Stm** is the set of all statements (to be defined next)

Syntax of WHILE Programs

$$a \ ::= \ n \mid x \mid a \oplus a \mid a \star a \mid a \ominus a$$

$$b \ ::= \ \mathbf{true} \mid \mathbf{false} \mid a \equiv a \mid a \leqq a \mid \neg b \mid b \wedge b$$

$$S \ ::= \ x \texttt{ := } a \mid \texttt{skip} \mid S \texttt{ ; } S \mid \texttt{if } b \texttt{ then } S \texttt{ else } S \mid \texttt{while } b \texttt{ do } S$$

where $n \in \text{Num}$ and $x \in \text{Var}$.

These are *all* the syntactic categories, rigorously defined by grammars. Really all?

Exercise: Provide a definition for numerals and variables.

Semantic Functions

Assumptions:

1. numerals are given in decimal notation
2. semantic function $\mathcal{N}[\![\cdot]\!] : \mathrm{Num} \to \mathbb{Z}$

A *state* is a function from variables to $\mathbb{Z}$.

$$\mathbf{State} = \mathbb{Z}^{\mathrm{Var}}$$

Need semantic functions for the syntactic categories

- **Aexp** $\mathcal{A} : \mathbf{Aexp} \to (\mathbf{State} \to \mathbb{Z})$
- **Bexp** $\mathcal{B} : \mathbf{Bexp} \to (\mathbf{State} \to \mathbb{B})$
- **Stm** $\mathcal{S} : \mathbf{Stm} \to (\ ??\ )$

?? should be replaced by *partial functions* $\mathbf{State} \hookrightarrow \mathbf{State}$.

Total and Partial Functions

A function $f : A \to B$ is an object $f \subseteq A \times B$ such that (1) $\forall a \in A : \exists b \in B : (a, b) \in f$ and (2) if for $a \in A$ we have $b_1, b_2 \in B$ with $(a, b_1) \in f$ and $(a, b_2) \in f$, then $b_1 = b_2$. In contrast, a *partial function* $g : A \hookrightarrow B$ removes requirement (1).

If for $a \in A$ there is a $b \in B$ such that $(a, b) \in g$, we write $g(a) = b$. If for all $b \in B$, $(a, b) \notin g$, we write $g(a) = \perp$ where $\perp \notin B$ is assumed to be the symbol for *undefined value*.