1. A Primer in Programming Language Semantics

Lecture on Models of Concurrent Systems

(Summer 2022)

Stephan Mennicke

Apr 5, 2022

What is Computation?

P:
```
x := 1;
x := x + 1;
```

Q:
```
x := 2;
```

- What does **P** compute? What does **Q** compute? Are **P** and **Q** equivalent?
- What is needed to argue for this, formally?
- How to overcome the underspecification of the questions above?

Organization

Sessions in Presence

Tuesday, DS3 (11:10–12:40), APB E005
Wednesday, DS3 (11:10–12:40), APB E005

Exercises

In the spirit of this course: **interleaved** with the lectures.

Web Page

https://iccl.inf.tu-dresden.de/web/Concurrency_Theory_(SS2022)

Lecture Notes

Slides of current lecture will be Online.

Goals and Prerequisites

Learning Goals

- Semantics of concurrent programming languages
  - What is a process?
  - When are two processes equivalent?
- Advanced features of concurrent processes
- The coinductive proof method

Prerequisites

- No particular prior course needed
- Semantics of programming languages helpful
- General mathematical and theoretical computer science skills necessary

What is Computation?

P:
```
x := 1;
x := x + 1;
```

Q:
```
x := 2;
```

- What does **P** compute? What does **Q** compute? Are **P** and **Q** equivalent?
- What is needed to argue for this, formally?
  $\rightsquigarrow$ Semantics of Programming Languages
- How to overcome the underspecification of the questions above?

Recap: TheoLog@TUD

LOOP-Programme: Syntax

Definition: Die Programmiersprache LOOP basiert auf einer unendlichen Menge **V** von Variablen und der Menge $\mathbb{N}$ der natürlichen Zahlen. LOOP-Programme sind induktiv definiert:

- Die Ausdrücke

  $$\mathtt{x} := \mathtt{y} + n \quad \text{und} \quad \mathtt{x} := \mathtt{y} - n \quad \text{(Wertzuweisung)}$$

  sind LOOP-Programme für alle $\mathtt{x}, \mathtt{y} \in$ **V** und $n \in \mathbb{N}$.
- Wenn P_1 und P_2 LOOP-Programme sind, dann ist

  $$P_1 \,;\, P_2 \quad \text{(Hintereinanderausführung)}$$

  ein LOOP-Programm.
- Wenn P ein LOOP-Programm ist, dann ist

  $$\textbf{LOOP}\ \mathtt{x}\ \textbf{DO}\ P\ \textbf{END} \quad \text{(Schleife)}$$

  ein LOOP-Programm, für jede Variable $\mathtt{x} \in$ **V**.

Vereinfachung: Wir erlauben **;** in Programmen durch Zeilenumbrüche zu ersetzen

WHILE Programs

Definition 1.1: The language **WHILE** is based on a universe $\mathcal{V}$ of variables, which are assigned values from the set of integers $\mathbb{Z}$. A **WHILE** program is an expression derived from the following grammar:

$$P \quad ::= \quad \texttt{x := } a \;\Big|\; P\,\texttt{;}\,P \;\Big|\; \texttt{IF } b \texttt{ THEN } P \texttt{ ELSE } P \texttt{ END} \;\Big|\; \texttt{WHILE } b \texttt{ DO } P \texttt{ END}$$

where $\texttt{x} \in \mathcal{V}$, $n \in \mathbb{Z}$, and a are arithmetic expression of the form

$$a \quad ::= \quad \texttt{x} \;\Big|\; n \;\Big|\; a \texttt{ + } a \;\Big|\; a \texttt{ - } a \;\Big|\; a \texttt{ * } a$$

and b are Boolean expression of the form

$$b \quad ::= \quad \texttt{true} \;\Big|\; a \texttt{ = } a \;\Big|\; a \texttt{ != } a \;\Big|\; a \texttt{ <= } a \;\Big|\; \texttt{not } b \;\Big|\; b \texttt{ and } b$$

State Functions

We call a function $s : \mathcal{V} \to \mathbb{Z}$ a state function.

Arithmetic and Boolean expressions are evaluated over state functions. Let $\mathbf{A}$ and $\mathbf{B}$ the set of all arithmetic and Boolean expressions as defined before. A state function changes in the course of evaluating assignments of $\mathbf{WHILE}$ programs.

Define **semantic functions** $\mathcal{A}[\![\cdot]\!] : \mathbf{A} \times (\mathcal{V} \to \mathbb{Z}) \to \mathbb{Z}$ and $\mathcal{B}[\![\cdot]\!] : \mathbf{B} \times (\mathcal{V} \to \mathbb{Z}) \to \mathbf{2}$.

$$
\begin{aligned}
\mathcal{A}[\![\mathtt{x}]\!]s &= s(x) & \mathcal{A}[\![n]\!]s &= n \\
\mathcal{A}[\![a_1 \;\mathtt{+}\; a_2]\!]s &= \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s & \mathcal{A}[\![a_1 \;\mathtt{-}\; a_2]\!]s &= \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 \;\mathtt{*}\; a_2]\!]s &= \mathcal{A}[\![a_1]\!]s \cdot \mathcal{A}[\![a_2]\!]s
\end{aligned}
$$

We assume no association or distributivity for the arithmetic operators. We would use brackets to make the order of evaluation explicit.

Semantic Functions (cont'd)

$$\begin{aligned} \mathcal{B}[\![\texttt{true}]\!]s &= \top & \mathcal{B}[\![\texttt{not}\ b_1]\!]s &= \neg\mathcal{B}[\![b_1]\!]s \\ \mathcal{B}[\![b_1\ \texttt{and}\ b_2]\!]s &= \mathcal{B}[\![b_1]\!]s \wedge \mathcal{B}[\![b_2]\!]s & \mathcal{B}[\![b_1\ \texttt{or}\ b_2]\!]s &= \mathcal{B}[\![b_1]\!]s \vee \mathcal{B}[\![b_2]\!]s \end{aligned}$$

We would derive further Boolean operators as well as the keyword `false` as usual.

To determine the semantics of the other operators, we apply $\mathcal{A}[\![\cdot]\!]$ to the operands.

$$\mathcal{B}[\![a_1\ \texttt{=}\ a_2]\!]s = \begin{cases} \top & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \bot & \text{otherwise.} \end{cases}$$

$$\mathcal{B}[\![a_1\ \texttt{!=}\ a_2]\!]s = \begin{cases} \top & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \\ \bot & \text{otherwise.} \end{cases}$$

In other words, $\mathcal{B}[\![a_1\ \texttt{!=}\ a_2]\!]s = \mathcal{B}[\![\texttt{not}\ (a_1\ \texttt{=}\ a_2)]\!]s$. Other comparison operators (like <= for $\leq$) are implemented similarly.

Semantics of WHILE Programs

In **structural operational semantics**, we define transition rules between the **configurations** of a program according to the structure (syntax) of it.

A $\mathbf{WHILE}$ **configuration** is a pair $\langle P, s\rangle$ where $P \in \mathbf{WHILE}$ and s is a state function. Furthermore, a state function s is a configuration, called a **terminal configuration**. We start a program P in an initial configuration $\langle P, s_0\rangle$ with some state function s_0 (e. g., $s_0(\mathtt{x}) = 0$ for all $\mathtt{x} \in \mathcal{V}$). We will make use of a special $\mathbf{WHILE}$ statement: `skip` which just performs a step without changing the state function (e. g., $\mathtt{skip} = \mathtt{x\ :=\ x}$).

The first rule performs the assignment $x := a$, changing the value of x to the value of $\mathcal{A}[\![a]\!]s$:

$$\text{(ASS)}\ \frac{}{\langle \mathtt{x\ :=\ } a, s\rangle \Rightarrow s[\mathtt{x} \mapsto \mathcal{A}[\![a]\!]s]}$$

Note, rule (ASS) has an empty hypothesis, meaning that the rule performs unconditionally.

Semantics of WHILE Programs (cont'd)

The next two rules handle the cases for sequential composition $P_1 ; P_2$. Either P_1 terminates and P_2 takes up its state function, or P_1 computes and intermediate step.

$$\text{(SEQ1)}\ \frac{\langle P_1, s\rangle \Rightarrow s'}{\langle P_1 ; P_2, s\rangle \Rightarrow \langle P_2, s'\rangle}$$

$$\text{(SEQ2)}\ \frac{\langle P_1, s\rangle \Rightarrow \langle P_1', s'\rangle}{\langle P_1 ; P_2, s\rangle \Rightarrow \langle P_1' ; P_2, s'\rangle}$$

For branching we implement a case distinction, depending on whether the Boolean expression evaluates to $\top$ (true) or $\bot$ (false).

$$\text{(THEN)}\ \frac{}{\langle \texttt{IF}\ b\ \texttt{THEN}\ P_1\ \texttt{ELSE}\ P_2\ \texttt{END}, s\rangle \Rightarrow \langle P_1, s\rangle}\ \text{if } \mathcal{B}[\![b]\!]s = \top$$

$$\text{(ELSE)}\ \frac{}{\langle \texttt{IF}\ b\ \texttt{THEN}\ P_1\ \texttt{ELSE}\ P_2\ \texttt{END}, s\rangle \Rightarrow \langle P_2, s\rangle}\ \text{if } \mathcal{B}[\![b]\!]s = \bot$$

Semantics of WHILE Programs (FINISH)

For a **while-loop** we can build on the constructs we already have:

$$\text{(WHILE)}\ \frac{}{\langle \texttt{WHILE}\ b\ \texttt{DO}\ P\ \texttt{END}, s\rangle \Rightarrow \langle \texttt{IF}\ b\ \texttt{THEN}\ P\texttt{;}\ \texttt{WHILE}\ b\ \texttt{DO}\ P\ \texttt{END ELSE skip END}, s\rangle}$$

The semantics of a $\mathbf{WHILE}$ program P is defined as

$$\mathcal{S}[\![P]\!]s := \begin{cases} s' & \text{if } \langle P, s\rangle \Rightarrow^* s' \\ \textit{undefined} & \text{otherwise.} \end{cases}$$

Thus, $\mathcal{S}[\![\cdot]\!] : \mathbf{WHILE} \times (\mathcal{V} \to \mathbb{Z}) \to (\mathcal{V} \to \mathbb{Z})$ is (expectedly) a partial function.

What is Computation?

P:
```
x := 1;
x := x + 1;
```

Q:
```
x := 2;
```

- What does **P** compute? What does **Q** compute? Are **P** and **Q** equivalent?
  $\rightsquigarrow$ $\mathcal{S}[\![\mathbf{P}]\!]s = s[\mathrm{x} \mapsto 2]$ and $\mathcal{S}[\![\mathbf{Q}]\!]s = s[\mathrm{x} \mapsto 2]$
  $\rightsquigarrow$ **P** and **Q** are equivalent under $\mathcal{S}[\![\cdot]\!]$.
- What is needed to argue for this, formally?
  $\rightsquigarrow$ Semantics of Programming Languages
- How to overcome the underspecification of the questions above?
  $\rightsquigarrow$ For all **variable valuations** $s : \mathbf{V} \to \mathbb{N}$, does $\mathcal{S}[\![\mathbf{P}]\!]s = \mathcal{S}[\![\mathbf{Q}]\!]s$ hold? **or**
  For all contexts $C[\cdot]$, are $C[\mathbf{P}]$ and $C[\mathbf{Q}]$ equivalent in the above-mentioned sense?
- Class over? But title mentions the word **concurrent**!
  $\rightsquigarrow$ What about languages with explicit parallel operator, as in $P_1 \mid P_2$?