Systeme Hoher Sicherheit und Qualität
Universität Bremen WS 2015/2016

Lecture 12 (18.01.2016)

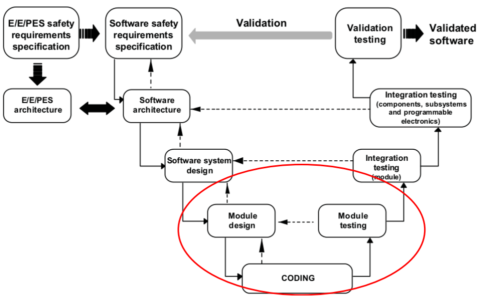
Semantics of Programming Languages

Christoph Lüth   Jan Peleska   Dieter Hutter

Universität Bremen

Where are we?

Semantics in the Development Process



Semantics — what does that mean?

> " Semantics: The meaning of words, phrases or systems. "
>
> — *Oxford Learner's Dictionaries*

- In mathematics and computer science, semantics is giving a meaning in mathematical terms. It can be contrasted with syntax, which specifies the notation.
- Here, we will talk about the meaning of programs. Their syntax is described by formal grammars, and their semantics in terms of mathematical structures.
- Why would we want to do that?

Why Semantics?

Semantics describes the meaning of a program (written in a programming language) in mathematical precise and unambiguous way. Here are three reasons why this is a good idea:

- It lets us write better compilers. In particular, it makes the language independent of a particular compiler implementation.
- If we know the precise meaning of a program, we know when it should produce a result and when not. In particular, we know which situations the program should avoid.
- Finally, it lets us reason about program correctness.

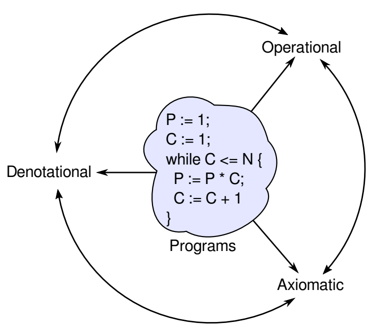
Empfohlene Literatur: Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* The MIT Press, 1993.

Semantics of Programming Languages

Historically, there are three ways to write down the semantics of a programming language:

- Operational semantics describes the meaning of a program by specifying how it executes on an abstract machine.
- Denotational semantics assigns each program to a partial function on the system state.
- Axiomatic semantics tries to give a meaning of a programming construct by giving proof rules. A prominent example of this is the Floyd-Hoare logic of previous lectures.

A Tale of Three Semantics



- Each semantics should be considered a view of the program.
- Importantly, all semantics should be equivalent. This means we have to put them into relation with each other, and show that they agree. Doing so is an important sanity check for the semantics.
- In the particular case of axiomatic semantics (Floyd-Hoare logic), it is the question of correctness of the rules.

Operational Semantics

- Evaluation is directed by the syntax.
- We inductively define relations $\rightarrow$ between configurations (a command or expression together with a state) to an integer, boolean or a state:

$$\rightarrow_A \subseteq (\mathbf{AExp}, \Sigma) \times \mathbb{Z}$$
$$\rightarrow_B \subseteq (\mathbf{BExp}, \Sigma) \times \mathit{Bool}$$
$$\rightarrow_S \subseteq (\mathbf{Com}, \Sigma) \times \Sigma$$

where the system state is defined as as

$$\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbb{Z}$$

- $(p, \sigma) \rightarrow_S \sigma'$ means that evaluating the program p in state σ results in state σ', and $(a, \sigma) \rightarrow_A i$ means evaluating expression a in state σ results in integer value i.

Structural Operational Semantics

- The evaluation relation is defined by rules of the form

$$\frac{\langle a, \sigma\rangle \to_A i}{\langle \mathtt{p}\ a_1, \sigma\rangle \to_A f(i)}$$

for each programming language construct p. This means that when the argument a of the construct has been evaluated, we can evaluate the whole expression.

  - This is called structural operational semantics.

- Note that this does not specify an evaluation strategy.
- This evaluation is partial and can be non-deterministic.

IMP: Arithmetic Expressions

Numbers: $\langle n, \sigma\rangle \to_A n$

Variables: $\langle \mathtt{X}, \sigma\rangle \to_A \sigma(\mathtt{X})$

Addition: $$\frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 + a_1, \sigma\rangle \to_A n + m}$$

Subtraction: $$\frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 \text{ - } a_1, \sigma\rangle \to_A n - m}$$

Multiplication: $$\frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 \text{ * } a_1, \sigma\rangle \to_A n \cdot m}$$

IMP: Boolean Expressions (Constants, Relations)

$\langle \mathbf{true}, \sigma\rangle \to_B \mathit{True}$ $\quad$ $\langle \mathbf{false}, \sigma\rangle \to \mathit{False}$

$$\frac{\langle b, \sigma\rangle \to_B \mathit{False}}{\langle \mathbf{not}\ b, \sigma\rangle \to_B \mathit{True}} \qquad \frac{\langle b, \sigma\rangle \to_B \mathit{True}}{\langle \mathbf{not}\ b, \sigma\rangle \to_B \mathit{False}}$$

$$\frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 = a_1, \sigma\rangle \to_B \mathit{True}}\ n = m \qquad \frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 = a_1, \sigma\rangle \to_B \mathit{False}}\ n \neq m$$

$$\frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 < a_1, \sigma\rangle \to_B \mathit{True}}\ n < m \qquad \frac{\langle a_0, \sigma\rangle \to_A n \quad \langle a_1, \sigma\rangle \to_A m}{\langle a_0 < a_1, \sigma\rangle \to_B \mathit{False}}\ n \geq m$$

IMP: Boolean Expressions (Operators)

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{False} \quad \langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{False}} \qquad \frac{\langle b_0, \sigma\rangle \to_B \mathit{False} \quad \langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{False}}$$

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{True} \quad \langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{False}} \qquad \frac{\langle b_0, \sigma\rangle \to_B \mathit{True} \quad \langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{True}}$$

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{True} \quad \langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{True}} \qquad \frac{\langle b_0, \sigma\rangle \to_B \mathit{True} \quad \langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{True}}$$

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{False} \quad \langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{True}} \qquad \frac{\langle b_0, \sigma\rangle \to_B \mathit{False} \quad \langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{False}}$$

IMP: Boolean Expressions (Operators — Variation)

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{False}} \qquad \frac{\langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{False}}$$

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{True} \quad \langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{False}} \qquad \frac{\langle b_0, \sigma\rangle \to_B \mathit{True} \quad \langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ and } b_1, \sigma\rangle \to_B \mathit{True}}$$

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{True}} \qquad \frac{\langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{True}}$$

$$\frac{\langle b_0, \sigma\rangle \to_B \mathit{False} \quad \langle b_1, \sigma\rangle \to_B \mathit{True}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{True}} \qquad \frac{\langle b_0, \sigma\rangle \to_B \mathit{False} \quad \langle b_1, \sigma\rangle \to_B \mathit{False}}{\langle b_0 \text{ or } b_1, \sigma\rangle \to_B \mathit{False}}$$

What is the difference?

Operational Semantics of IMP: Statements

$\langle \mathbf{skip}, \sigma\rangle \to_S \sigma$

$$\frac{\langle a, \sigma\rangle \to_S n}{\langle X := a, \sigma\rangle \to_S \sigma[n/X]} \qquad \frac{\langle c_0, \sigma\rangle \to_S \tau \quad \langle c_1, \tau\rangle \to_S \tau'}{\langle c_0; c_1, \sigma\rangle \to_S \tau'}$$

$$\frac{\langle b, \sigma\rangle \to_B \mathit{True} \quad \langle c_0, \sigma\rangle \to_S \tau}{\langle \mathbf{if}\ b\ \{c_0\}\ \mathbf{else}\ \{c_1\}, \sigma\rangle \to_S \tau} \qquad \frac{\langle b, \sigma\rangle \to \mathit{False} \quad \langle c_1, \sigma\rangle \to_S \tau}{\langle \mathbf{if}\ b\ \{c_0\}\ \mathbf{else}\ \{c_1\}, \sigma\rangle \to_S \tau}$$

$$\frac{\langle b, \sigma\rangle \to_B \mathit{False}}{\langle \mathbf{while}\ b\ \{c\}, \sigma\rangle \to_S \sigma}$$

$$\frac{\langle b, \sigma\rangle \to_B \mathit{True} \quad \langle c, \sigma\rangle \to_S \tau' \quad \langle \mathbf{while}\ b\ \{c\}, \tau'\rangle \to_S \tau}{\langle \mathbf{while}\ b\ \{c\}, \sigma\rangle \to_S \tau}$$

Why Denotational Semantics?

- Denotational semantics takes an abstract view of program: if $c_1 \sim c_2$, they have the "same meaning".
- This allows us, for example, to compare programs in different programming languages.
- It also accommodates reasoning about programs far better than operational semantics. In particular, we can prove the correctness of the Floyd-Hoare rules.
- It gives us compositionality and referential transparency, mapping programming language construct p to denotation ϕ:

$$\mathcal{D}[\![\mathtt{p}(e_1, \ldots, e_n)]\!] = \phi(\mathcal{D}[\![e_1]\!], \ldots, \mathcal{D}[\![e_n]\!])$$

Denotational Semantics

- Programs are denoted by functions on states $\Sigma = \mathbf{Loc} \rightharpoonup \mathbb{Z}$.
- Semantic functions assign a meaning to statements and expressions:

Arithmetic expressions:	$\mathcal{E} : \mathbf{AExp} \to (\Sigma \to \mathbb{Z})$
Boolean expressions:	$\mathcal{B} : \mathbf{BExp} \to (\Sigma \to \mathit{Bool})$
Statements:	$\mathcal{D} : \mathbf{Com} \to (\Sigma \rightharpoonup \Sigma)$

- Note the meaning of a program p is a partial function, reflecting the fact that programs may not terminate.
  - Our expressions always do, but that is because our language is quite simple.

Denotational Semantics of IMP: Arithmetic Expressions

$$
\begin{aligned}
\mathcal{E}[\![n]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.n \\
\mathcal{E}[\![X]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\sigma(X) \\
\mathcal{E}[\![a_0 + a_1]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.(\mathcal{E}[\![a_0]\!]\sigma + \mathcal{E}[\![a_1]\!]\sigma) \\
\mathcal{E}[\![a_0 \text{ - } a_1]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.(\mathcal{E}[\![a_0]\!]\sigma - \mathcal{E}[\![a_1]\!]\sigma) \\
\mathcal{E}[\![a_0 * a_1]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.(\mathcal{E}[\![a_0]\!]\sigma \cdot \mathcal{E}[\![a_1]\!]\sigma)
\end{aligned}
$$

Denotational Semantics of IMP: Boolean Expressions

$$
\begin{aligned}
\mathcal{B}[\![\textbf{true}]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\mathit{True} \\
\mathcal{B}[\![\textbf{false}]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\mathit{False} \\
\mathcal{B}[\![\textbf{not } b]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\neg\mathcal{B}[\![b]\!]\sigma \\
\mathcal{B}[\![a_0 = a_1]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\begin{cases} \mathit{True} & \mathcal{E}[\![a_0]\!]\sigma = \mathcal{E}[\![a_1]\!]\sigma \\ \mathit{False} & \mathcal{E}[\![a_0]\!]\sigma \neq \mathcal{E}[\![a_1]\!]\sigma \end{cases} \\
\mathcal{B}[\![a_0 < a_1]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\begin{cases} \mathit{True} & \mathcal{E}[\![a_0]\!]\sigma < \mathcal{E}[\![a_1]\!]\sigma \\ \mathit{False} & \mathcal{E}[\![a_0]\!]\sigma \geq \mathcal{E}[\![a_1]\!]\sigma \end{cases} \\
\mathcal{B}[\![b_0 \text{ and } b_1]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\mathcal{B}[\![b_0]\!]\sigma \wedge \mathcal{B}[\![b_1]\!]\sigma \\
\mathcal{B}[\![b_0 \text{ or } b_1]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\mathcal{B}[\![b_0]\!]\sigma \vee \mathcal{B}[\![b_1]\!]\sigma
\end{aligned}
$$

Denotational Semantics of IMP: Statements

The simple part:

$$
\begin{aligned}
\mathcal{D}[\![\textbf{skip}]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\sigma \\
\mathcal{D}[\![X := a]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\sigma[\mathcal{E}[\![a]\!]\sigma/X] \\
\mathcal{D}[\![c_0; c_1]\!] &\stackrel{def}{=} \mathcal{D}[\![c_1]\!] \circ \mathcal{D}[\![c_0]\!] \\
\mathcal{D}[\![\textbf{if } b\ \{c_0\} \textbf{ else } \{c_1\}]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\begin{cases} \mathcal{D}[\![c_0]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{True} \\ \mathcal{D}[\![c_1]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{False} \end{cases}
\end{aligned}
$$

The hard part:

$$
\mathcal{D}[\![\textbf{while } b\ \{c\}]\!] = \lambda\sigma \in \Sigma.\begin{cases} \sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{False} \\ (\mathcal{D}[\![\textbf{while } b\ \{c\}]\!] \circ \mathcal{D}[\![c]\!])\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{True} \end{cases}
$$

This recursive definition is not constructive — it does not tell us how to construct the function. Worse, it is unclear it even exists in general.

Partial Orders and Least Upper Bounds

To construct fixpoints of the form $x = f(x)$, we need the theory of complete partial orders (cpo's).

Definition (Partial Order)

Given a set X, a partial order $\sqsubseteq\ \subseteq X \times X$ is

(i) transitive: if $x \sqsubseteq y, y \sqsubseteq z$, then $x \sqsubseteq z$

(ii) reflexive: $x \sqsubseteq x$

(iii) anti-symmetric: if $x \sqsubseteq y, y \sqsubseteq x$ then $x = y$

Definition (Least Upper Bound)

For $Y \subseteq X$, the least upper bound $\bigsqcup Y \in X$ is:

(i) $\forall y \in Y. y \sqsubseteq \bigsqcup Y$

(ii) for any $z \in X$ such that $\forall y \in Y. y \sqsubseteq z$, we have $\bigsqcup Y \sqsubseteq z$

Complete Partial Orders

Definition (Complete Partial Order)

A partial order $\sqsubseteq$ is complete (a cpo) if any ω-chain $x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq x_4 \ldots = \{x_i \mid i \in \omega\}$ has a least upper bound $\bigsqcup_{i\in\omega} x_i \in X$.

A cpo is called pointed (pcpo), if there is a smallest element $\bot \in X$. (Note some authors assume all cpos to be pointed.)

Definition (Continuous Function)

Given cpos $(X, \sqsubseteq)$ and $(Y, \leq)$. A function $f : X \to Y$ is

(i) monotone, if $x \sqsubseteq y$ then $f(x) \leq f(y)$

(ii) continuous, if monotone and $f(\bigsqcup_{i\in\omega} x_i) = \bigsqcup_{i\in\omega} f(x_i)$

Fixpoints

Theorem (Each continuous function has a least fixpoint)

Let $(X, \sqsubseteq)$ be a pcpo, and $f : X \to X$ continuous, then f has a least fixpoint $fix(f)$, given as

$$
fix(f) = \bigsqcup_{n\in\omega} f^n(\bot)
$$

- In our case, the state Σ is made into a pcpo $\Sigma_\bot$ by 'adjoining' a new element $\bot$, ordered as $\bot \sqsubseteq \sigma$.
- This models partial functions: $\Sigma \rightharpoonup \Sigma \cong \Sigma \to \Sigma_\bot$
- $\Sigma \to \Sigma_\bot$ ist a pcpo, ordered as

$$
f \sqsubseteq g \longleftrightarrow \forall x. f(x) \sqsubseteq g(x)
$$

Concretely, $f \sqsubseteq g$ means that f is defined on fewer states than g.

Denotational Semantics of IMP: Statements

$$
\begin{aligned}
\mathcal{D}[\![\textbf{skip}]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\sigma \\
\mathcal{D}[\![X := a]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\sigma[\mathcal{E}[\![a]\!]\sigma/X] \\
\mathcal{D}[\![c_0; c_1]\!] &\stackrel{def}{=} \mathcal{D}[\![c_1]\!] \circ \mathcal{D}[\![c_0]\!] \\
\mathcal{D}[\![\textbf{if } b\ \{c_0\} \textbf{ else } \{c_1\}]\!] &\stackrel{def}{=} \lambda\sigma \in \Sigma.\begin{cases} \mathcal{D}[\![c_0]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{True} \\ \mathcal{D}[\![c_1]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{False} \end{cases} \\
\mathcal{D}[\![\textbf{while } b\ \{c\}]\!] &\stackrel{def}{=} fix(\Gamma) \\
\text{where } \Gamma(\phi) &\stackrel{def}{=} \lambda\sigma \in \Sigma.\begin{cases} \phi \circ \mathcal{D}[\![c]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{True} \\ \sigma & \mathcal{B}[\![b]\!]\sigma = \mathit{False} \end{cases}
\end{aligned}
$$

Equivalence of Semantics

Lemma

(i) *For $a \in$ **Aexp**, $n \in \mathbb{N}$, $\mathcal{E}[\![a]\!]\sigma = n$ iff $\langle a, \sigma\rangle \to_A n$*

(ii) *For $b \in$ **BExp**, $t \in Bool$, $\mathcal{B}[\![b]\!]\sigma = t$ iff $\langle b, \sigma\rangle \to_B t$*

Proof: Structural Induction on a and b. □

Lemma

*For $c \in$ **Com**, if $\langle c, \sigma\rangle \to_S \sigma'$ then $\mathcal{D}[\![c]\!]\sigma = \sigma'$*

Proof: Induction over deriviation of $\langle c, \sigma\rangle \to_S \sigma'$. □

Theorem (Equivalence of Semantics)

*For $c \in$ **Com**, and $\sigma, \sigma' \in \Sigma$,*

$$
\langle c, \sigma\rangle \to_S \sigma' \text{ iff } \mathcal{D}[\![c]\!]\sigma = \sigma'
$$

The proof of this theorem requires a technique called fixpoint induction which we will not go into detail about here.

Correctness of Floyd-Hoare Rules

Denotational semantics allows us to prove the correctness of the Floyd-Hoare rules.

- We extend the boolean semantic functions $\mathcal{E}$ and $\mathcal{B}$ to **AExpv** and **BExpv**, respectively.
- We can then define the validity of a Hoare triple in terms of denotations:

$$\models \{P\}\, c\, \{Q\} \text{ iff } \forall \sigma. \mathcal{B}[\![P]\!]\sigma \wedge \mathcal{D}[\![c]\!]\sigma \neq \bot \longrightarrow \mathcal{B}[\![Q]\!](\mathcal{D}[\![c]\!]\sigma)$$

- We can now show the rules preserve validity, *i.e.* if the preconditions are valid Hoare triples, then so is the conclusion.

Remarks

- Our language and semantics is quite simple-minded. We have not take into account:
  - undefined expressions (such as division by 0 or accessing an undefined variable),
  - side effects in expressions,
  - declaration of variables,
  - pointers, references, pointer arithmetic,
  - input/output (what is the semantic model?), or
  - concurrency.
- However, there are formal semantics for languages such as StandardML, C, or Java, although most of them concentrate on some aspect of the language (*e.g.* Java concurrency is not very well defined in the standard). Only StandardML has a language standard which is written as an operational semantics.

Conclusion

- Programming semantics come in three flavours: operational, denotational, axiomatic.
- Each of these has their own use case:
  - Operational semantics gives details about evaluation of programs, and is good for implementing the programming language.
  - Denotational semantics is abstract and good for high-level reasoning (*e.g.* correctness of program logics or tools).
  - Axiomatic semantics is about program logics, and reasoning about programs.
- Denotational semantics needs the mathematical toolkit of cpos to construct fixpoints.