

# Programming Language Concepts and Semantics Part I

## Lecture 7 Denotational Semantics

Lecture VII introduces a different kind of semantic specification formalism, *denotational semantics*. This presents a far more abstract view of programming language semantics, which is more strongly decoupled from the idea of an execution of a program.

Denotational semantics is arguable the most general kind of semantics specification.

Lecture VII follows Winskel, Chapter 5.

## Denotational Semantics

In *Denotational Semantics* we specify *what* a program or program fragment computes leaving the idea of an execution model behind.

Denotational Semantics

- uses “meaning functions” (denotations)
- is based on Domain Theory  
(continuous functions and complete partial orders)

An important concept are *fixpoints* of functions which we will use to specify meaning functions.

## Denotations

The basic Idea is quite simple. Recall from our SOS definition of IMP how we specified the meaning of an arithmetic expression

The obvious way to rewrite this part of the specification to a function is to let it map an expression and a state to a value.

$$f : \mathbf{Aexp} \times \Sigma \rightarrow \mathbf{N}$$

$$f(a, \sigma) \mapsto n$$

This is **not** quite the way that denotational semantics specifies a meaning function. Instead we assign a meaning function  $\mathcal{A}$  that maps an expression  $a$  to a semantic function  $\mathcal{A}[[a]]$

$$\mathcal{A}[[a]] : \Sigma \rightarrow \mathbf{N}$$

such that the semantic function maps the environment to the value.

This obviously requires  $\mathcal{A}$  to be a function of type

$$\mathcal{A} : \mathbf{Aexp} \rightarrow (\Sigma \rightarrow \mathbf{N})$$

which is a curried version of the function  $f$  that was used above to intuitively capture the corresponding state transformation.

*Note:* Double brackets are traditionally used to indicate that the function is a denotation rather than a traditional function and that the argument therefore is a syntactic construct.

## Denotations for IMP

To specify IMP we define the following semantic functions:

$$\begin{aligned}\mathcal{A} : \mathbf{Aexp} &\rightarrow (\Sigma \rightarrow \mathbf{N}) \\ \mathcal{B} : \mathbf{Bexp} &\rightarrow (\Sigma \rightarrow \mathbf{T}) \\ \mathcal{C} : \mathbf{Com} &\rightarrow (\Sigma \rightarrow \Sigma)\end{aligned}$$

We see that the functions' types correspond to the left-hand side and right-hand side types of the SOS state transformations.

We will use *structural induction* to define  $\mathcal{A}, \mathcal{B}, \mathcal{C}$  mimicking the SOS rules in a straight forward way.

This might suggest that our denotational semantics still clings to an operational model (mimics the execution). However, this analogy only holds up to a point. This will become clear when we use fixpoints to specify the semantics of statements.

## Denotational Semantics of Arithmetic Expressions

For technical reasons we adopt a notation where we denote the semantics of an arithmetic expression not as a function from  $\Sigma$  to  $\mathbf{N}$  but instead as a relation on  $\Sigma$  and  $\mathbf{N}$ . Thus technically we are defining

$$\mathcal{A} : \mathbf{Aexp} \rightarrow \mathcal{P}(\Sigma \times \mathbf{N})$$

$$\begin{aligned}\mathcal{A}[[n]] &= \{(\sigma, n) \mid \sigma \in \Sigma\} \\ \mathcal{A}[[X]] &= \{(\sigma, \sigma(X)) \mid \sigma \in \Sigma\} \\ \mathcal{A}[[a_0 + a_1]] &= \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}[[a_0]] \wedge (\sigma, n_1) \in \mathcal{A}[[a_1]]\} \\ \mathcal{A}[[a_0 - a_1]] &= \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[[a_0]] \wedge (\sigma, n_1) \in \mathcal{A}[[a_1]]\} \\ \mathcal{A}[[a_0 * a_1]] &= \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in \mathcal{A}[[a_0]] \wedge (\sigma, n_1) \in \mathcal{A}[[a_1]]\}\end{aligned}$$

It is obvious that the above definition is equivalent to a function

$$\mathcal{A} : \mathbf{Aexp} \rightarrow (\Sigma \rightarrow \mathbf{N})$$

i.e. in

$$\mathcal{A}[[a]] : \sigma \mapsto n$$

$n$  is uniquely defined.

## Example

Assume that  $X$  is bound to 7 in  $\sigma$ .

$$\begin{aligned}\mathcal{A}[[3 + (5 * X)]]\sigma &= \\ \mathcal{A}[[3]]\sigma + \mathcal{A}[[5 * X]]\sigma &= \\ \mathcal{A}[[3]]\sigma + \mathcal{A}[[5]]\sigma * \mathcal{A}[[X]]\sigma &= \\ \mathcal{A}[[3]]\sigma + 5 * \mathcal{A}[[X]]\sigma &= \\ \mathcal{A}[[3]]\sigma + 5 * \sigma(X) &= \\ \mathcal{A}[[3]]\sigma + 5 * 7 &= \\ \mathcal{A}[[3]]\sigma + 35 &= \\ 3 + 35 &= \\ 38 &= \end{aligned}$$

Recall that the environment  $\sigma$  is a function  $\sigma : \mathbf{Var} \rightarrow \mathbf{N}$  mapping variable identifiers to values.

## Denotational Semantics of Boolean Expressions

In the same vein we specify the denotation  $\mathcal{B}$  of boolean expressions as a relation

$$\mathcal{B} : \mathbf{Bexp} \rightarrow \mathcal{P}(\Sigma \times \mathbf{T})$$

$$\begin{aligned} \mathcal{B}[\mathbf{true}] &= \{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma\} \\ \mathcal{B}[\mathbf{false}] &= \{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma\} \\ \mathcal{B}[a_0 = a_1] &= \{(\sigma, \mathbf{true}) \mid \mathcal{A}[a_0] = \mathcal{A}[a_1]\} \\ &\quad \cup \{(\sigma, \mathbf{false}) \mid \mathcal{A}[a_0] \neq \mathcal{A}[a_1]\} \\ \mathcal{B}[a_0 \leq a_1] &= \{(\sigma, \mathbf{true}) \mid \mathcal{A}[a_0] \leq \mathcal{A}[a_1]\} \\ &\quad \cup \{(\sigma, \mathbf{false}) \mid \mathcal{A}[a_0] \not\leq \mathcal{A}[a_1]\} \\ \mathcal{B}[\neg b] &= \{(\sigma, \neg_T t) \mid (\sigma, t) \in \mathcal{B}[b]\} \\ \mathcal{B}[b_0 \wedge b_1] &= \{(\sigma, t_0 \wedge_T t_1) \mid (\sigma, t_0) \in \mathcal{B}[b_0] \wedge_T (\sigma, t_1) \in \mathcal{B}[b_1]\} \\ \mathcal{B}[b_0 \vee b_1] &= \{(\sigma, t_0 \vee_T t_1) \mid (\sigma, t_0) \in \mathcal{B}[b_0] \vee_T (\sigma, t_1) \in \mathcal{B}[b_1]\} \end{aligned}$$

Note the difference between  $\wedge, \vee, \neg$  and  $\wedge_T, \vee_T, \neg_T$ , respectively. *wedge* etc. denote the syntactic elements of IMP whereas  $\wedge_T$  etc. denote the customary evaluable boolean operations.

## Denotational Semantics of Simple Statements

$$\mathcal{C} : \mathbf{Com} \rightarrow \mathcal{P}(\Sigma \times \Sigma)$$

defines the semantics of commands as a function of state to states.

This is easy for basic statements and conditionals.

- **Empty statements**

$$\mathcal{C}[[\mathbf{skip}]] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

Empty statements have no effect on the environment.

- **Assignments**

$$\mathcal{C}[[X := a]] = \{(\sigma, \sigma[n/X]) \mid \sigma \in \Sigma \wedge \mathcal{A}[[a]]\sigma = n\}$$

An assignment only modifies the environment by substituting the binding for the corresponding variable.

- **Sequence**

$$\mathcal{C}[[c_0; c_1]] = \mathcal{C}[[c_1]] \circ \mathcal{C}[[c_0]]$$

The semantics of a sequence is simply the composition of the meaning functions.

- **Conditional**

$$\begin{aligned} \mathcal{C}[[\mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1]] = \\ \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[[b]] \wedge_T (\sigma, \sigma') \in \mathcal{C}[[c_0]]\} \\ \cup \{(\sigma, \sigma') \mid (\sigma, \mathbf{false}) \in \mathcal{B}[[b]] \wedge_T (\sigma, \sigma') \in \mathcal{C}[[c_1]]\} \end{aligned}$$

## Denotational Semantics of Loops

The semantics of a loop is not as straight forward, since we do not know in advance when the execution will stop.

In the SOS definition we had stated the equivalence

$$W \sim \mathbf{if } b \mathbf{ then } c; w \mathbf{ else skip}$$

for

$$W \equiv \mathbf{while } b \mathbf{ do } c$$

simulating the one step expansion of  $w$ .

We try to use this equivalence to define  $\mathcal{C}[[w]]$ , i.e. to define  $\mathcal{C}[[w]]$  in terms of a conditional.

$$\begin{aligned} \mathcal{C}[[w]] &= \\ &\quad \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[[b]] \wedge_T (\sigma, \sigma') \in \mathcal{C}[[c; w]]\} \\ &\quad \cup \{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[[b]]\} \\ &= \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[[b]] \wedge_T (\sigma, \sigma') \in \mathcal{C}[[w]] \circ \mathcal{C}[[c]]\} \\ &\quad \cup \{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[[b]]\} \end{aligned}$$

The problem obviously is that we have defined  $\mathcal{C}[[w]]$  in terms of  $\mathcal{C}[[w]]$ . How can we clarify this definition?

## Fixpoint Construction

Imagine we want to construct the definition of  $\mathcal{C}[[w]]$  incrementally. All that we have to do is to extend it step by step. We first start with the simple cases where  $b$  is **false**. Call this the *known cases*.

Next we consider all states from which a single execution of  $c$  lets us reach one of the known cases. We can safely add these states to our known cases.

We then ask the same question again: From which states can we reach one of our (now augmented) set of known cases by a single application of  $c$  and extend the known cases by these.

If we can repeat this process until at some step no new cases are added to the set of known cases we can be sure that we have collected all parts of the definition of  $\mathcal{C}[[w]]$ .

Note that this is a conceptualization and cannot actually be performed because the  $\mathcal{C}[[w]]$  has infinitely many instances. Consider  $w \sim \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - 1$ . For any given environment  $\sigma = \mathcal{C}[[w]]\sigma'$  There is another  $\sigma'' = \sigma[x + 1/x]$  with  $\sigma = \mathcal{C}[[w]]\sigma''$ .

Thus  $\mathcal{C}[[w]]$  has infinitely many instances.

To be able to define  $\mathcal{C}[[w]]$  we introduce fixpoints.

## Fixpoints

Given a function

$$f : S \rightarrow S$$

we can attempt to iterate  $f$  from some starting point  $x$ :

$$x, f(x), f(f(x)), f(f(f(x))), \dots, f^{n-1}(x), f^n(x).$$

If  $f^{n-1}(x) = f^n(x)$  then  $f$  has a fixpoint at  $x$ .

More generally, a fixpoint  $x$  of  $f$  is defined by

$$x = f(x)$$

**Example 1 :** Let  $S = \mathbf{N}$ , let  $f = 2x$ . The only fixpoint of  $f$  is  $x = 0$ .

**Example 2 :** Let  $S$  be the set of Australian residents.

Let  $f(s) = s \cup \{r' \mid r \in (s \cup \{you\}) \wedge_T first\_degree\_relatives(r, r')\}$

The fixpoint of  $f$  is the set of all your relatives (down to Adam and Eve). Call this set  $s^*$ .

Note that we could add a complete other family lineage to  $s^*$  and it would remain a fixpoint. Call Bernd's family lineage  $s^{*'}$ .

$$f(s^{*' \cup s^*}) = s^{*' \cup s^*$$

and so this is a fixpoint, too.

In general we are interested in the *least fixpoints*. This is the fixpoint  $s^*$  that is included in all other fixpoints.

## Fixpoint Definition of While

We define a function  $\Gamma$  that “simulates” the single-step unfolding of a while loop. We had

$$\begin{aligned} \mathcal{C}[[w]] = & \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[[b]] \wedge_T (\sigma, \sigma') \in \mathcal{C}[[w]] \circ \mathcal{C}[[c]]\} \\ & \cup \{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[[b]] \wedge_T\} \end{aligned}$$

Abbreviating  $\phi = \mathcal{C}[[w]]$ ,  $\beta = \mathcal{B}[[b]]$ ,  $\gamma = \mathcal{C}[[c]]$  we can have

$$\phi = \{(\sigma, \sigma') \mid \beta(\sigma) = \mathbf{true} \wedge_T (\sigma, \sigma') \in \phi \circ \gamma\} \cup \{(\sigma, \sigma) \mid \beta(\sigma) = \mathbf{false}\}$$

we can now introduce a function  $\Gamma$  the fixpoint of which describes the meaning of  $w$ :

$$\begin{aligned} \Gamma(\phi) = & \{(\sigma, \sigma') \mid \beta(\sigma) = \mathbf{true} \wedge_T (\sigma, \sigma') \in \phi \circ \gamma\} \cup \\ & \{(\sigma, \sigma) \mid \beta(\sigma) = \mathbf{false}\} \\ = & \{(\sigma, \sigma') \mid \exists \sigma''. \beta(\sigma) = \mathbf{true} \wedge_T (\sigma, \sigma'') \in \gamma \wedge_T (\sigma'', \sigma') \in \phi\} \cup \\ & \{(\sigma, \sigma) \mid \beta(\sigma) = \mathbf{false}\} \end{aligned}$$

We can now proceed define the semantics of while loops as the fixpoint of  $\gamma$

$$\mathcal{C}[[\mathbf{while} \ b \ \mathbf{do} \ c]] = \mathit{fixpoint}(\Gamma)$$

where

$$\begin{aligned} \Gamma(\phi) = & \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[[b]] \wedge_T (\sigma, \sigma') \in \phi \circ \mathcal{C}[[c]]\} \\ & \cup \{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[[b]]\} \end{aligned}$$

## Equivalence of Semantic Definitions

To make sure that the SOS definition and the denotational definition of IMP are consistent, we should prove that the defined semantics are equivalent.

Note that this is a different kind of equivalence: Here we are not checking whether two program fragments are equivalent (“do” the same or compute the same, depending on the type of equivalence), but we check whether two different semantic specifications for the same language are equivalent.

We check this separately for **Aexp**, **Bexp**, **Com** and use structural induction for all proofs.

## Equivalence of Basic Arithmetic Expressions

### Equivalence of Arithmetic Expressions

$$\forall a \in \mathbf{Aexp} . \mathcal{A}[[a]] = \{(\sigma, n) \mid \langle a, \sigma \rangle \rightarrow n\}$$

- **Numbers:** From the definition of  $\mathcal{A}$  we have

$$(\sigma, n) \in \mathcal{A}[[m]] \Leftrightarrow \sigma \in \Sigma \wedge_T n \equiv m$$

From the definition of  $\rightarrow$  we have  $\langle n, \sigma \rangle \rightarrow n$ .

- **Variables:** From the definition of  $\mathcal{A}$  we have

$$(\sigma, n) \in \mathcal{A}[[x]] \Leftrightarrow \sigma \in \Sigma \wedge_T n \equiv \sigma(X)$$

From the definition of  $\rightarrow$  we have  $\langle X, \sigma \rangle \rightarrow \sigma(X)$ .

## Equivalence of Arithmetic Operations

$$\forall a \in \mathbf{Aexp} . \mathcal{A}[[a]] = \{(\sigma, n) \mid \langle a, \sigma \rangle \rightarrow n\}$$

- **Operations:** Let  $a \equiv a_0 + a_1$ . On the basis of using *structural induction* we can assume that we have proven the equivalence for  $a_0$  and  $a_1$ .

“ $\Rightarrow$ ”: From the definition of  $\mathcal{A}$  we have

$$(\sigma, n) \in \mathcal{A}[[a_0 + a_1]] \Leftrightarrow \exists n_0, n_1 . n = n_0 + n_1 \wedge_T (\sigma, n_0) \in \mathcal{A}[[a_0]] \wedge_T (\sigma, n_1) \in \mathcal{A}[[a_1]]$$

On the basis of the induction for  $a_0, a_1$  we know that  $\langle a_0, \sigma \rangle \rightarrow n_0$  and  $\langle a_1, \sigma \rangle \rightarrow n_1$ . We can thus infer

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma \rangle \rightarrow n_0 + n_1}$$

“ $\Leftarrow$ ”: From the definition of  $\rightarrow$  we now that every derivation ending in  $\langle a_0 + a_1, \sigma \rangle \rightarrow n_0 + n_1$  must have the form

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma \rangle \rightarrow n_0 + n_1}$$

Thus  $\langle a_0, \sigma \rangle \rightarrow n_0$  and  $\langle a_1, \sigma \rangle \rightarrow n_1$ . On the basis of the induction, we know that  $(\sigma, n_0) \in \mathcal{A}[[a_0]]$  and  $(\sigma, n_1) \in \mathcal{A}[[a_1]]$ . We substitute this in the definition of  $\mathcal{A}[[a_0 + a_1]]$  to obtain  $(\sigma, n) \in \mathcal{A}[[a_0 + a_1]]$ .

This concludes the induction. □

*The proof for the remaining operations is structurally identical to the case for addition.*

## Equivalence of Boolean Expressions

$$\forall b \in \mathbf{Bexp} . \mathcal{B}[[b]] = \{(\sigma, t) \mid \langle b, \sigma \rangle \rightarrow t\}$$

To prove this equivalence with structural induction we need to analyze the cases for boolean values **true** and **false**, arithmetic predicates ( $=$  and  $\leq$ ) and boolean operations  $\wedge, \vee$ .

The proof for boolean values is structurally identical to the case of numbers in arithmetic expressions.

The proof for boolean operations is structurally identical to the case of operations in arithmetic expressions.

So is the proof for arithmetic predicates, with the slight technical difference that the type of the expression changes (the induced expression is boolean whereas the induction assumptions refer to arithmetic expressions).

*The details are left as an exercise.*

## Equivalence of Commands

$$\forall c \in \mathbf{Com} . \mathcal{C}[[c]] = \{(\sigma, \sigma') \mid \langle c, \sigma \rangle \rightarrow \sigma'\}$$

This is equivalent to

$$\langle c, \sigma \rangle \rightarrow \sigma' \Leftrightarrow (\sigma, \sigma') \in \mathcal{C}[[c]]$$

**To prove the “ $\Rightarrow$ ” direction we can use rule induction.**

This means to show that for every inference rule of the operational semantics, if the condition holds for all the premises of the rule, then it holds for the conclusion.

We illustrate this with the transformation rule for the non-exit case of a while loop.

## Equivalence of *while* Definitions (“ $\Rightarrow$ ”)

$$\frac{\langle b, \sigma \rangle \rightarrow \mathbf{true} \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma'' \rangle \rightarrow \sigma'}{\langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle \rightarrow \sigma'}$$

Let  $w \equiv \mathbf{while} \ b \ \mathbf{do} \ c$ .

From the precondition we obtain

$$\begin{aligned} \langle b, \sigma \rangle &\rightarrow \mathbf{true} \wedge_T \\ \langle c, \sigma \rangle &\rightarrow \sigma'' \wedge_T \\ \langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma'' \rangle &\rightarrow \sigma' \end{aligned}$$

From this we obtain via induction

$$(\sigma, \sigma'') \in \mathcal{C}[[c]] \wedge_T (\sigma'', \sigma') \in \mathcal{C}[[w]]$$

We also have  $\mathcal{B}[[b]] = \mathbf{true}$  from the equivalence of semantic definitions for boolean expressions (above).

From this and the denotational definition of **while** we obtain

$$\mathcal{C}[[w]]\sigma = (\mathcal{C}[[w]] \circ \mathcal{C}[[c]])\sigma = \mathcal{C}[[c; w]]\sigma = \mathcal{C}[[w]](\mathcal{C}[[c]]\sigma) = \mathcal{C}[[w]]\sigma'' = \sigma'$$

This means that  $\mathcal{C}[[w]]\sigma = \sigma'$  and thus the assumption holds for the conclusion of the inference rule.

Other rules can be proven in the same way.

As all rules preserve the property, the “ $\Rightarrow$ ” is complete. □.

## Equivalence of Commands (“ $\Leftarrow$ ”)

To prove the “ $\Leftarrow$ ” direction, we use structural induction:

$$\langle c, \sigma \rangle \rightarrow \sigma' \Leftarrow (\sigma, \sigma') \in \mathcal{C}[[c]]$$

- **Empty Statements:**  $\mathcal{C}[[\mathbf{skip}]] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$ . Thus for  $(\sigma, \sigma') \in \mathcal{C}[[c]]$  we have  $\sigma = \sigma'$  and so  $\langle \mathbf{skip}, \sigma \rangle \rightarrow \sigma'$  by definition.
- **Assignments:**  $\mathcal{C}[[X := a]] = \{(\sigma, \sigma[n/X]) \mid \sigma \in \Sigma \wedge \mathcal{A}[[X]]\sigma = n\}$ . Thus  $\sigma' = \sigma[n/X]$ . From the equivalence of arithmetic expressions  $\langle a, \sigma \rangle \rightarrow n$ . From the definition of  $\rightarrow$ ,  $\langle c, \sigma \rangle \rightarrow \sigma'$ .
- **Sequence:**  $\mathcal{C}[[c_0; c_1]] = \mathcal{C}[[c_1]] \circ \mathcal{C}[[c_0]]$ . There must be some state  $\sigma''$  such that  $(\sigma, \sigma'') \in \mathcal{C}[[c_0]]$  and  $(\sigma'', \sigma') \in \mathcal{C}[[c_1]]$ . Hence  $\langle c_0; c_1, \sigma \rangle \rightarrow \sigma'$ .
- **Conditionals:**

$$\begin{aligned} \mathcal{C}[[\mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1]] = & \\ & \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[[b]] \wedge_T (\sigma, \sigma') \in \mathcal{C}[[c_0]]\} \\ & \cup \{(\sigma, \sigma') \mid (\sigma, \mathbf{false}) \in \mathcal{B}[[b]] \wedge_T (\sigma, \sigma') \in \mathcal{C}[[c_1]]\} \end{aligned}$$

We can analyze both cases separately by structural induction. The details are left as an exercise.

## Equivalence of Iteration (“ $\Leftarrow$ ”)

$$\mathcal{C}[\text{while } b \text{ do } c_0] = \text{fixpoint}(\Gamma)$$

where

$$\begin{aligned} \Gamma(\phi) = & \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[[b]] \wedge_T (\sigma, \sigma') \in \phi \circ \mathcal{C}[[c_0]]\} \\ & \cup \{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[[b]]\} \end{aligned}$$

This is equivalent to writing

$$\mathcal{C}[[c]] = \bigcup_{n=1 \dots \infty} \gamma_n$$

where  $\gamma_n = \Gamma^n(\emptyset)$  and

$$\begin{aligned} \gamma_0 &= \emptyset \\ \gamma_{n+1} &= \{(\sigma, \sigma') \mid (\sigma, \mathbf{true}) \in \mathcal{B}[[b]] \wedge_T (\sigma, \sigma') \in \gamma_n \circ \mathcal{C}[[c_0]]\} \\ &\quad \cup \{(\sigma, \sigma) \mid (\sigma, \mathbf{false}) \in \mathcal{B}[[b]]\} \end{aligned}$$

We can now use *ordinary induction* on the integers to show that

$$\forall \sigma, \sigma' \in \Sigma . (\sigma, \sigma') \in \gamma_n \Rightarrow \langle c, \sigma \rangle \rightarrow \sigma'$$

for all  $n \in \mathbb{N}$ .

## Equivalence of Iteration (“ $\Leftarrow$ ”) (cont.)

- $n = 0$  :  $\gamma_0 = \emptyset$ , so this case is trivial.
- $n > 0$  :  $(\sigma, t) \in \mathcal{B}[[b]]$ . We distinguish the two cases:
  - $t = \mathbf{true}$  and  $(\sigma, \sigma') \in \gamma_n \circ \mathcal{C}[[c_0]]$ : There must be some state  $\sigma''$  such that  $(\sigma'', \sigma') \in \gamma_n$  and  $(\sigma, \sigma'') \in \mathcal{C}[[c_0]]$ . From the induction on  $\gamma_n$  we obtain  $\langle c, \sigma'' \rangle \rightarrow \sigma'$ . From the structural induction on commands we obtain  $\langle c_0, \sigma \rangle \rightarrow \sigma''$ . Also, because of the equivalence of boolean expressions,  $\langle b, \sigma \rangle \rightarrow \mathbf{true}$ . Substituting this into the  $\rightarrow$  definition for *while* we obtain  $\langle c, \sigma \rangle \rightarrow \sigma'$ .
  - $t = \mathbf{false}$  and  $\sigma = \sigma'$ : From the equivalence of boolean expressions we obtain  $\langle b, \sigma \rangle \rightarrow \mathbf{false}$ . Substituting this and into the  $\rightarrow$  definition for *while* we can derive  $\langle c, \sigma \rangle \rightarrow \sigma$ .

This establishes the induction for all  $n \in N$ , and thus concludes the induction for *while*.

This completes the structural induction for all  $c \in \mathbf{Com}$ . □

## Summary

We have discussed denotational semantics

- Denotational semantics uses meaning functions that map environments to environments.
- An inductive denotational semantics definition can be given based on syntactic structure.
- Denotational semantics abstract from the operational model.
- A central proof technique for denotational semantics is induction.
- Fixpoints play an important role for denotational semantics definitions.

We have also given the denotational semantics for IMP and sketched the equivalence proof of operational and denotational IMP semantics.

## Homework

- Complete the missing parts of the equivalence proofs that were left as exercise.