Introduction to Program Semantics — Dexter Kozen

Lecture 1 - *June 25, 2026*

1 Program Semantics

Program semantics is used to reason about program behavior. There are three main ways to give semantics: operational semantics, denotational semantics, axiomatic semantics. Each style emphasizes a different aspect: local states, global states, ad observations.

2 The Simple Imperative Language IMP

IMP has three syntactic categories: arithmetic expressions a, boolean expressions b, and commands c.

BNF (the "normal form" for grammars).

$$
\begin{aligned}
a &::= n \mid x \in \mathrm{Var} \mid a_1 + a_2 \mid a_1 * a_2 \mid \cdots \\
b &::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \mid \cdots \\
c &::= \texttt{skip} \mid x := a \mid c_1 \,;\, c_2 \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } b \texttt{ do } c
\end{aligned}
$$

These grammars also describe *abstract syntax trees*; in AST form there is no need to write parentheses, since the tree structure makes precedence and associativity unambiguous.

3 Configurations and Environments

A *configuration* is a snapshot of reality at a particular time: an abstraction of the "state of the world" for the purposes of execution.

Compiled By:
Yanjun Chen, Kunha Kim, Jonathon Yallop

- An **environment** (or state) is a mapping $s : \mathrm{Var} \to \mathbb{N}$ from variables to values. For now we take it to be a *total* function.
- A **configuration** is a pair $\langle c, s\rangle$: s records the current environment, and c records the commands still to execute.

There are two natural ways to describe how configurations evolve: *small-step* and *big-step*.

4 Small-Step Semantics of IMP

Small-step semantics is given as a **binary relation** on configurations.

We write $\langle c, s\rangle \longrightarrow_c \langle c', s'\rangle$ for "one step." The reflexive transitive closure $\longrightarrow^*$ (zero or more steps) captures longer runs:

$$\langle c, s\rangle \longrightarrow_c^* \langle c', s'\rangle.$$

It may be the case that a program does *not* terminate, in which case there is an infinite chain $\langle c, s\rangle \longrightarrow_c \langle c_1, s_1\rangle \longrightarrow_c \cdots$.

We need auxiliary relations $\longrightarrow_a$ and $\longrightarrow_b$ for arithmetic and boolean expressions.

4.1 Arithmetic expressions

$$\frac{}{\langle x, s\rangle \longrightarrow_a \langle s(x), s\rangle} \qquad \frac{\langle a_1, s\rangle \longrightarrow_a \langle a_1', s'\rangle}{\langle a_1 + a_2, s\rangle \longrightarrow_a \langle a_1' + a_2, s'\rangle} \qquad \frac{\langle a_2, s\rangle \longrightarrow_a \langle a_2', s'\rangle}{\langle n_1 + a_2, s\rangle \longrightarrow_a \langle n_1 + a_2', s'\rangle}$$

$$\langle n_1 + n_2, s\rangle \longrightarrow_a \langle n_3, s\rangle$$

where n_3 is the sum of $n_1 + n_2$.

In any configuration, *exactly one rule applies* – the relation is deterministic. For $a_1 + a_2$ we first reduce the left operand; only once it has become a numeral n_1 do we start reducing the right. The state is threaded through in the congruence rules (s on the premise, s' on the conclusion): although addition itself does not change the state, sub-expressions in a richer language can, and the small-step relation is the place where that order of effects is recorded. The actual arithmetic happens *only* when both sides are numerals; that is the mathematical identity $n_3 = n_1 + n_2$, and it leaves the state unchanged because at that point all sub-evaluation is complete.

4.2 Commands

We can define similar thing for Boolean expressions. As for commands, $\langle \texttt{skip}, s\rangle$ is the terminal configuration.

$$\frac{\langle a,\, s\rangle \longrightarrow_a \langle a',\, s\rangle}{\langle x := a,\, s\rangle \longrightarrow_c \langle x := a',\, s\rangle} \qquad \frac{}{\langle x := n,\, s\rangle \longrightarrow_c \langle \texttt{skip},\, s[n/x]\rangle} \qquad \frac{\langle c_1,\, s\rangle \longrightarrow_c \langle c_1',\, s'\rangle}{\langle c_1\,;c_2,\, s\rangle \longrightarrow_c \langle c_1'\,;c_2,\, s'\rangle}$$

$$\frac{}{\langle \texttt{skip}\,;c_2,\, s\rangle \longrightarrow_c \langle c_2,\, s\rangle} \qquad \frac{\langle b,\, s\rangle \longrightarrow_b \langle b',\, s\rangle}{\langle \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2,\, s\rangle \longrightarrow_c \langle \texttt{if } b' \texttt{ then } c_1 \texttt{ else } c_2,\, s\rangle}$$

$$\frac{}{\langle \texttt{if true then } c_1 \texttt{ else } c_2,\, s\rangle \longrightarrow_c \langle c_1,\, s\rangle} \qquad \frac{}{\langle \texttt{if false then } c_1 \texttt{ else } c_2,\, s\rangle \longrightarrow_c \langle c_2,\, s\rangle}$$

$$\frac{}{\langle \texttt{while } b \texttt{ do } c,\, s\rangle \longrightarrow_c \langle \texttt{if } b \texttt{ then } (c\,;\texttt{while } b \texttt{ do } c) \texttt{ else skip},\, s\rangle}$$

Discussion. Note that there is exactly one rule per command form, with one exception: `skip` has *no* rule – it represents a terminal configuration $\langle \texttt{skip},\, s\rangle$. Also, `while` simply *unfolds* the loop into an `if`. This is a trick: the semantics of `while` is defined by reduction to constructs already understood, and the (potential) non-termination is captured by the fact that this rule can fire over and over.

5 Big-Step Semantics of IMP

The big-step (or *natural*) semantics relates a configuration directly to its result. Unlike the small-step relation, the right-hand side is no longer a configuration: arithmetic expressions yield a value and commands yield a state.

$$\langle a,\, s\rangle \downarrow_a n, \qquad \langle c,\, s\rangle \downarrow_c s'.$$

Reading the shapes: $\langle a,\, s\rangle \downarrow_a n$ means "in state s, the expression a eventually evaluates to the numeral n," while $\langle c,\, s\rangle \downarrow s'$ means "starting from s, the command c eventually halts with final state s'." Boolean expressions follow the same pattern as arithmetic ($\langle b,\, s\rangle \downarrow_b v$); we omit them and focus on arithmetic and commands here.

5.1 Arithmetic Expressions

$$\frac{}{\langle n,\, s\rangle \downarrow_a n} \qquad \frac{}{\langle x,\, s\rangle \downarrow_a s(x)} \qquad \frac{\langle a_1,\, s\rangle \downarrow_a n_1 \qquad \langle a_2,\, s\rangle \downarrow_a n_2}{\langle a_1 + a_2,\, s\rangle \downarrow_a n_3}$$

where n_3 is the sum $n_1 + n_2$.

The rules for boolean operators and comparisons follow the same pattern: evaluate both sides to values, then apply the corresponding mathematical operator.

5.2 Commands

$$\frac{}{\langle \texttt{skip},\, s\rangle \downarrow_c s} \qquad \frac{\langle a,\, s\rangle \downarrow_a n}{\langle x := a,\, s\rangle \downarrow_c s[n/x]} \qquad \frac{\langle c_1,\, s\rangle \downarrow_c s' \qquad \langle c_2,\, s'\rangle \downarrow_c s''}{\langle c_1\,;\, c_2,\, s\rangle \downarrow_c s''}$$

$$\frac{\langle b,\, s\rangle \downarrow_b \texttt{true} \qquad \langle c_1,\, s\rangle \downarrow_c s'}{\langle \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2,\, s\rangle \downarrow_c s'} \qquad \frac{\langle b,\, s\rangle \downarrow_b \texttt{false} \qquad \langle c_2,\, s\rangle \downarrow_c s'}{\langle \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2,\, s\rangle \downarrow_c s'} \qquad \frac{\langle b,\, s\rangle \downarrow_b \texttt{false}}{\langle \texttt{while } b \texttt{ do } c,\, s\rangle \downarrow_c s}$$

$$\frac{\langle b,\, s\rangle \downarrow_b \texttt{true} \qquad \langle c,\, s\rangle \downarrow s' \qquad \langle \texttt{while } b \texttt{ do } c,\, s'\rangle \downarrow_c s''}{\langle \texttt{while } b \texttt{ do } c,\, s\rangle \downarrow_c s''}$$

Discussion. The sequencing rule has two premises that share an existential "middle" state s': c_1 must take s to s', and c_2 must take that same s' onward to s''.

6 Small-step vs. Big-step

- **Big-step** is convenient for building a *recursive interpreter*: each rule corresponds to a clause in a recursive function. However, you lose the step-by-step "cursor" – order of evaluation, intermediate states, and partial progress are invisible.
- **Small-step** gives a finer view. It lets you reason about *termination*, non-determinism, concurrency, and stuck states.

Agreement theorem. The two semantics agree on terminating programs:

$$\langle c,\, s\rangle \downarrow_c s' \quad \Longleftrightarrow \quad \langle c,\, s\rangle \longrightarrow_c^* \langle \texttt{skip},\, s'\rangle.$$

The proof goes by induction (the single most-used technique in computer science).

Aside: well-founded induction. Recall that one can perform induction on a relation R iff R is *well-founded* (no infinite descending chains). This justifies induction on the height of a derivation, the length of a reduction, structural induction on syntax, etc.

7 Evaluation Contexts

Evaluation order rules say *where* the next reduction should happen. The order rules, although accurate, are tedious. A context E is a term with a single hole [] marking where a reduction is allowed.

Evaluation contexts for IMP.

$$\begin{aligned} E_a &::= [] \mid E_a + a \mid n + E_a \mid \cdots \\ E_b &::= [] \mid E_a = a \mid n = E_a \mid E_b \wedge b \mid \neg E_b \mid \cdots \\ E &::= [] \mid x := E_a \mid E\,;c \mid \texttt{if } E_b \texttt{ then } c_1 \texttt{ else } c_2 \end{aligned}$$

We write $E[e]$ for the term obtained by plugging e into the hole of E.

The "context rule."

$$\frac{\langle e,\, s\rangle \longrightarrow \langle e',\, s'\rangle}{\langle E[e],\, s\rangle \longrightarrow \langle E[e'],\, s'\rangle}$$

Now we only state the genuine reduction rules.

8 A Denotational View of Big-Step Semantics

We can recast big-step semantics as a binary relation on environments, and then think of it as a (partial) mathematical function.

Programs as relations.

$$[\![c]\!] \subseteq \text{Env} \times \text{Env}, \qquad (s, s') \in [\![c]\!] \iff \langle c,\, s\rangle \downarrow s'.$$

Because $\downarrow$ is deterministic, $[\![c]\!]$ is actually a partial *function*.

Compositional definition. Writing $\circ$ for relational composition, $\cup$ for union, id for the identity relation, $[\![b]\!]$ for the set of states where b holds (viewed as the partial identity $\{(s,s) \mid [\![b]\!](s) = \texttt{true}\}$), $*$ for transitive closure of relation, and for complement of the relation:

$$\begin{aligned} [\![\texttt{skip}]\!] &= \mathrm{id}_{\text{Env}} \\ [\![c_1\,;c_2]\!] &= [\![c_2]\!] \circ [\![c_1]\!] \\ [\![\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2]\!] &= ([\![c_1]\!] \circ [\![b]\!]) \cup ([\![c_2]\!] \circ [\![\neg b]\!]) \\ [\![\texttt{while } b \texttt{ do } c]\!] &= \big([\![c]\!] \circ [\![b]\!]\big)^* \circ [\![\ b]\!] \end{aligned}$$

Discussion of $[\![c_1\,;c_2]\!] = [\![c_2]\!] \circ [\![c_1]\!]$. It means

$$(s, s'') \in [\![c_1\,;c_2]\!] \iff \exists s'.\ (s, s') \in [\![c_1]\!] \ \wedge\ (s', s'') \in [\![c_2]\!].$$

Think of programs as "functions on the state": ; is composition, and many laws drop out for free, e.g. associativity $(c_1\,;c_2)\,;c_3 = c_1\,;(c_2\,;c_3)$ and the unit laws $\texttt{skip}\,;c = c = c\,;\texttt{skip}$. In the same vein, $[\![b \wedge b']\!] = [\![b]\!] \cap [\![b']\!]$ when we view b as the test set $\{s \mid [\![b]\!](s) = \texttt{true}\}$, i.e. tests form a Boolean subalgebra.

Note also the appearance of $(-)^*$ in the rule for `while`: that is the Kleene star, the very thing one uses in regular expressions.

9 Kleene Algebra and Programs

A *Kleene algebra* (KA) is a structure

$$(K,\ +,\ \cdot,\ ^*,\ 0,\ 1)$$

satisfying the axioms below. Intuitively, $+$ is nondeterministic choice, $\cdot$ is sequential composition, 0 is failure, 1 is the empty action (skip), and * is iteration "zero or more times."

9.1 Axioms

Idempotent semiring axioms. $(K, +, 0)$ is a commutative, idempotent monoid and $(K, \cdot, 1)$ is a monoid; $\cdot$ distributes over $+$ on both sides; and 0 annihilates $\cdot$. For all $p, q, r \in K$:

$$\begin{aligned} p + (q + r) &= (p + q) + r & p(qr) &= (pq)r \\ p + q &= q + p & 1p = p &= p1 \\ p + 0 &= p & p(q + r) &= pq + pr \\ p + p &= p & (p + q)r &= pr + qr \\ & & 0p = 0 &= p0 \end{aligned}$$

Idempotence $(p + p = p)$ is the crucial axiom that distinguishes a Kleene algebra from an ordinary semiring; it lets us define a *natural partial order* on K by

$$p \leq q \iff p + q = q.$$

The operator * is characterized by $1 + pp^* \leq p*$, $q + px \leq p^*qx$, $1 + p^*p \leq p*$, $q + xp \leq qp^*x$,

9.2 Kleene Algebra with Tests (KAT)

To model `if` and `while` we need a notion of *test*. A *Kleene algebra with tests* is a two-sorted structure

$$(K,\ B,\ +,\ \cdot,\ ^*,\ \text{-},\ 0,\ 1)$$

where

- $(K, +, \cdot, ^*, 0, 1)$ is a Kleene algebra;
- $B \subseteq K$ contains 0 and 1.
- $(B, +, \cdot, \bar{}, 0, 1)$ is a Boolean algebra.

Encoding control flow. In KAT one defines

$$\texttt{if } b \texttt{ then } p \texttt{ else } q \;:\; b \cdot p \;+\; \bar{b} \cdot q, \qquad \texttt{while } b \texttt{ do } p \;:\; (b \cdot p)^* \cdot \bar{b}.$$

They are the precise denotations of `if` and `while` that we already derived from the denotational reading. We can then use Kleene algebra to analyze program semantics.

We can use SKAT similarly to Hoare logic.

$$\{\varphi[t/x]\}\; x := t\; \{\varphi\}$$

is encoded in SKAT by any of equation

$$\varphi[t/x]; x := t; \varphi = \varphi[t/x]; x := t$$
$$\varphi[t/x]; x := t; \neg\varphi = 0$$
$$\varphi[t/x]; x := t \leq x := t; \neg\varphi$$