

# CS421 Lecture 16: Transition Semantics<sup>1</sup>

Mark Hills

`mhills@cs.uiuc.edu`

University of Illinois at Urbana-Champaign

July 14, 2008

---

<sup>1</sup>Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

Language Semantics

Transition Semantics

# Objectives

By the end of this lecture, you should

- ▶ have a high-level view of several different methods of giving semantics to programming languages
- ▶ be able to read and understand language definitions given using transition semantics

# Language Semantics

*Time flies like an arrow. Fruit flies like a banana.*

So far, we have syntax, but we don't know what any of it means!  
Language semantics provide a way for us to assign *meaning* to constructs in a programming language.

## Dynamic vs. Static

Semantics can be grouped into two general categories: *static* semantics and *dynamic* semantics.

- ▶ Static semantics provide meaning based solely on the form of a language construct, or the syntax – not based on execution. These are usually restricted to type checking and inference.
- ▶ Dynamic semantics provide meaning based on models of evaluation for the language. These tell us what it means to execute a program, for instance.

## Varieties of Semantics

There are many methods of giving semantics to a programming language. Several common methods include:

- ▶ Operational Semantics
- ▶ Axiomatic Semantics
- ▶ Denotational Semantics

Note that:

- ▶ it may be easier to represent certain languages with certain types of semantics
- ▶ the types of semantics are complementary – they are good for different purposes, with no method “the best”

## Operational Semantics

- ▶ First, start with a definition of our “machine” which we run programs on
- ▶ The machine state, including the program, is often called the *configuration*
- ▶ Next, describe how to execute programs in a given language by describing how to execute individual statements and parts of statements – rules follow the structure of the program
- ▶ A program’s “meaning” is defined by how it changes the configuration
- ▶ Useful as a basis for implementations

## Axiomatic Semantics

Axiomatic semantics are also called Floyd-Hoare logic

- ▶ based on logic – first-order predicate calculus
- ▶ semantics represented as a logical system build from *axioms* and *inference rules*
- ▶ mainly suited to simple imperative languages
- ▶ used to prove a post-condition from a pre-condition: given something holds in the starting state, we can show something else holds in the end state

$$\{\text{Precondition}\} \text{ Program } \{\text{Postcondition}\}$$

# Denotational Semantics

In denotational semantics, we want to find the meaning, or *denotation*, of phrases in our language.

- ▶ we construct a function  $\mathcal{M}$  assigning a mathematical meaning to each language construct;
- ▶ these functions are compositional – we can construct the meaning of a language construct by composing the meanings of its components
- ▶ useful for proving properties of programs – used for early type soundness proofs and in theorem provers (plus elsewhere)

## Alternative Methods

There are many other methods which have been devised to give semantics to languages.

- ▶ “Compiler-based” semantics – the meaning is whatever the compiler says it is
- ▶ Abstract Machines – similar to operational, an abstract machine with instructions, etc is devised and used to give semantics
- ▶ Term Rewriting – semantics are formulated as term rewriting systems, with evaluation given by the rewriting relation
- ▶ Rewriting Logic – an extension of equational logic that provides for concurrency

## Transition Semantics

Transition semantics is a form of operational semantics.

- ▶ Configurations include the code and machine state:  $(C, m)$
- ▶ Semantics specified as transitions between configurations, altering the machine state
- ▶ Rules of the form:  $\langle C, m \rangle \rightarrow \langle C', m' \rangle$
- ▶  $C, C'$  represents the code yet to be executed
- ▶  $m, m'$  represents the state (store, memory, etc), often a finite map from names to values
- ▶ May not need  $m$  – simple calculator languages with only numbers and operations don't, for instance

**Key point:** each transition indicates exactly one step of computation

## IMP – A Simple Imperative Language

We can use a simple imperative language, **IMP**, as a sample language for semantics. This language has the following syntactic categories:

- ▶ numbers **N**, which are the integers (including negatives)
- ▶ truth values **T** = {**true**, **false**}
- ▶ locations **Loc**
- ▶ arithmetic expressions **Aexp**
- ▶ boolean expressions **Bexp**
- ▶ commands **Com**

For shorthand,

$n, m \in \mathbf{N}$ ;  $X, Y \in \mathbf{Loc}$ ;  $a \in \mathbf{Aexp}$ ;  $b \in \mathbf{Bexp}$ ;  $c \in \mathbf{Com}$ .

# IMP Syntax

Using a variant of BNF, we can specify the syntax for IMP as:

$Aexp \quad a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$

$Bexp \quad b ::= \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$

$Com \quad c ::= \mathbf{skip} \mid X := a \mid c_0; c_1 \mid \mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1 \mid$   
 $\mathbf{while } b \mathbf{ do } c$

**Important Point:** We assume reasonable input programs that parse and we *don't* care about precedence, associativity, etc – we assume all that has been figured out for us

# IMP Configurations

Our configurations will contain two elements:

- ▶ The code ( $a$ ,  $b$ , or  $c$ )
- ▶ The state

We can define the set of states  $\Sigma$  as functions  $\sigma : \mathbf{Loc} \rightarrow \mathbf{N}$ , or functions from locations to integer values.

# Arithmetic Rules and Simple Expressions

Our semantic relation for arithmetic expressions will be of the form:

$$\langle a, \sigma \rangle \rightarrow n$$

We assume arithmetic expressions have no side-effects.

- ▶ We have a simple axiom for numbers:

$$\langle n, \sigma \rangle \rightarrow n$$

- ▶ Location lookup is also similar:

$$\langle X, \sigma \rangle \rightarrow \sigma(X)$$

## Arithmetic Expressions: Sums

$$\frac{\langle a_0, \sigma \rangle \rightarrow \langle a'_0, \sigma \rangle}{\langle a_0 + a_1, \sigma \rangle \rightarrow \langle a'_0 + a_1, \sigma \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle}{\langle n_0 + a_1, \sigma \rangle \rightarrow \langle n_0 + a'_1, \sigma \rangle}$$

$$\langle n_0 + n_1, \sigma \rangle \rightarrow n, \text{ where } n = n_0 +_{int} n_1$$

## Arithmetic Expressions: Subtraction

$$\frac{\langle a_0, \sigma \rangle \rightarrow \langle a'_0, \sigma \rangle}{\langle a_0 - a_1, \sigma \rangle \rightarrow \langle a'_0 - a_1, \sigma \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle}{\langle n_0 - a_1, \sigma \rangle \rightarrow \langle n_0 - a'_1, \sigma \rangle}$$

$$\langle n_0 - n_1, \sigma \rangle \rightarrow n, \text{ where } n = n_0 -_{int} n_1$$

## Arithmetic Expressions: Products

$$\frac{\langle a_0, \sigma \rangle \rightarrow \langle a'_0, \sigma \rangle}{\langle a_0 \times a_1, \sigma \rangle \rightarrow \langle a'_0 \times a_1, \sigma \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle}{\langle n_0 \times a_1, \sigma \rangle \rightarrow \langle n_0 \times a'_1, \sigma \rangle}$$

$$\langle n_0 \times n_1, \sigma \rangle \rightarrow n, \text{ where } n = n_0 \times_{int} n_1$$

## Boolean Rules and Simple Expressions

Our semantic relation for boolean expressions will be of the form:

$$\langle b, \sigma \rangle \rightarrow t$$

We assume boolean expressions have no side-effects.

- ▶ We have simple axioms for boolean constants, including true:

$$\langle \mathbf{true}, \sigma \rangle \rightarrow \mathbf{true}$$

- ▶ and false:

$$\langle \mathbf{false}, \sigma \rangle \rightarrow \mathbf{false}$$

## Boolean Expressions: Equality

$$\frac{\langle a_0, \sigma \rangle \rightarrow \langle a'_0, \sigma \rangle}{\langle a_0 = a_1, \sigma \rangle \rightarrow \langle a'_0 = a_1, \sigma \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle}{\langle n_0 = a_1, \sigma \rangle \rightarrow \langle n_0 = a'_1, \sigma \rangle}$$

$$\langle n_0 = n_1, \sigma \rangle \rightarrow b, \text{ where } b = (n_0 =_{int} n_1)$$

The rules for  $\leq$  are similar.

## Boolean Expressions: And

$$\frac{\langle b_0, \sigma \rangle \rightarrow \langle b'_0, \sigma \rangle}{\langle b_0 \wedge b_1, \sigma \rangle \rightarrow \langle b'_0 \wedge b_1, \sigma \rangle}$$

$$\frac{\langle b_1, \sigma \rangle \rightarrow \langle b'_1, \sigma \rangle}{\langle \mathbf{true} \wedge b_1, \sigma \rangle \rightarrow \langle b'_1, \sigma \rangle}$$

$$\langle \mathbf{false} \wedge b_1, \sigma \rangle \rightarrow \mathbf{false}$$

The rules for  $\vee$  and  $\neg$  are similar.

## Commands

While expressions in our language cannot have side effects, commands can. So, here we need to model the changes in state that occur when commands run. Here, our semantic relation will be of the form:

$$\langle c, \sigma \rangle \rightarrow \sigma'$$

So, when command  $c$  is fully evaluated, potentially altered memory  $\sigma'$  is returned.

## Simple Commands

The **skip** command does not alter the state:

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

## Assignment – Some New Notation

The assignment command does alter the state. One way we can view it is we get back a new state function which is the same everywhere except at the location we've updated, which now holds the new value. We can define this as:

$$\sigma[m/X](Y) = \begin{cases} m & \text{if } Y = X \\ \sigma(Y) & \text{if } Y \neq X \end{cases}$$

# Assignment

With our new notation, assignment can be shown as follows. Note we reduce the expression we are assigning to  $X$  first, before we do the actual assignment.

$$\frac{\langle a, \sigma \rangle \rightarrow \langle a', \sigma \rangle}{\langle X := a, \sigma \rangle \rightarrow \langle X := a', \sigma \rangle}$$

$$\langle X := n, \sigma \rangle \rightarrow \sigma[n/X]$$

## Sequencing

We can sequence commands as well in our language. We always complete execution of the first command before starting on the second. This gives us the following semantic rules:

$$\frac{\langle c_0, \sigma \rangle \rightarrow \langle c'_0, \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \rightarrow \langle c'_0; c_1, \sigma' \rangle}$$

$$\frac{\langle c_0, \sigma \rangle \rightarrow \sigma'}{\langle c_0; c_1, \sigma \rangle \rightarrow \langle c_1, \sigma' \rangle}$$

## Conditionals

We have a conditional statement in our language. We need to evaluate the guard first before deciding which branch to take. We can represent this as:

$$\frac{\langle b, \sigma \rangle \rightarrow \langle b', \sigma \rangle}{\langle \mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma \rangle \rightarrow \langle \mathbf{if } b' \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma \rangle}$$

$$\langle \mathbf{if true then } c_0 \mathbf{ else } c_1, \sigma \rangle \rightarrow \langle c_0, \sigma \rangle$$

$$\langle \mathbf{if false then } c_0 \mathbf{ else } c_1, \sigma \rangle \rightarrow \langle c_1, \sigma \rangle$$

# Loops

We have one loop, the while command. Here, we need to evaluate the guard – if it is still true, we want to evaluate the body, and we want to then evaluate the loop again. In some sense, this takes us back where we started, but most likely with a different state (if not, we probably won't terminate). We will expand to a conditional to do this, or else we would “lose” the guard when we evaluated it.

$$\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \langle \mathbf{if } b \mathbf{ then } (c; \mathbf{while } b \mathbf{ do } c) \mathbf{ else skip}, \sigma \rangle$$

## Example

To see how we can “evaluate” something operationally, start with:

$\langle \text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle$

In this state, we have at some point assigned the value 7 to location  $x$ .

## Example

$\langle \text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle$

Since this is a conditional, we first need to evaluate the guard until we get a value, either **true** or **false**;

$$\frac{\langle x, \{x \mapsto 7\} \rangle \rightarrow 7}{\langle \text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle \rightarrow \langle \text{if } 7 > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle}$$

## Example

$\langle \text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle$

Now that we just have values in the relational expression, we can use the appropriate rule to determine the truth or falsity of the guard.

$$\frac{\langle 7 > 5, \{x \mapsto 7\} \rangle \rightarrow \mathbf{true}}{\langle \text{if } 7 > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle \rightarrow \langle \text{if } \mathbf{true} \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle}$$

## Example

$\langle \text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle$

Now that we have a truth value for the guard, we can use the appropriate conditional rule to pick the correct statement to continue with.

$\langle \text{if } \mathbf{true} \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle \rightarrow$   
 $\langle y := 2 + 3, \{x \mapsto 7\} \rangle$

## Example

$\langle \text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle$

We can now evaluate the arithmetic expression, since we need to have an integer value before we can do assignment.

$$\frac{\langle 2 + 3, \{x \mapsto 7\} \rangle \rightarrow 5}{\langle y := 2 + 3, \{x \mapsto 7\} \rangle \rightarrow \langle y := 5, \{x \mapsto 7\} \rangle}$$

## Example

$\langle \text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle$

Finally, we can use the assignment rule to assign a value to  $y$ .

$\langle y := 5, \{x \mapsto 7\} \rangle \rightarrow \{x \mapsto 7, y \mapsto 5\}$

## Execution Summary

$\langle \text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle \rightarrow$   
 $\langle \text{if } 7 > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle \rightarrow$   
 $\langle \text{if } \mathbf{true} \text{ then } y := 2 + 3 \text{ else } y := 3 + 4, \{x \mapsto 7\} \rangle \rightarrow$   
 $\langle y := 2 + 3, \{x \mapsto 7\} \rangle \rightarrow$   
 $\langle y := 5, \{x \mapsto 7\} \rangle \rightarrow$   
 $\{x \mapsto 7, y \mapsto 5\}$

## Evaluation in Transition Semantics

As we saw above, we can view evaluation as a sequence of steps with trees of justifications for each step. This gives us a sequence of evaluation steps:

$$\langle C_1, m_1 \rangle \rightarrow \langle C_2, m_2 \rangle \rightarrow \dots \rightarrow m$$

We can then define  $\rightarrow^*$  as the transitive closure of  $\rightarrow$ , essentially giving us a relation that evaluates from  $\langle C_i, m_i \rangle$  to  $m$  (or, from a starting program and state to the final state).

## Adding Functions and Local Bindings

We don't currently have functions or let expressions in our language. How would adding them impact our semantics? We will change arithmetic expressions to just expressions (no need to change anything else defined so far, this just gives it a name more consistent with it's new use) and add syntax there.

*AExp*  $a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \mid$   
 $\text{let } X = a_0 \text{ in } a_1 \mid \text{fun } X \rightarrow a \mid a_0 a_1$

*Bexp*  $b ::= \text{true} \mid \text{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$

*Com*  $c ::= \text{skip} \mid X := a \mid c_0; c_1 \mid \text{if } b \text{ then } c_0 \text{ else } c_1 \mid$   
 $\text{while } b \text{ do } c$

## Semantics for Functions and Lets

- ▶ To keep track of bindings, we could use an *environment*, similar to the type environment we used for types – but in this case a finite map from locations to values
- ▶ This would “violate” what we mean by expression, though – an expression would then have a side effect
- ▶ So, we will use substitution here instead –  $[E'/X]E$ , meaning to replace all free  $X$  by  $E'$  in  $E$

We also have one axiom – a function, without application, just evaluates to itself:

$$\langle (\mathbf{fun} X \rightarrow a), \sigma \rangle \rightarrow (\mathbf{fun} X \rightarrow a)$$

## Call by Value

Using Call by Value, we will evaluate terms *before* substitution.  
 First, the let expression:

$$\langle \mathbf{let} \ X = n \ \mathbf{in} \ a, \sigma \rangle \rightarrow \langle [n/X]a, \sigma \rangle$$

$$\frac{\langle a_0, \sigma \rangle \rightarrow \langle a'_0, \sigma \rangle}{\langle \mathbf{let} \ X = a_0 \ \mathbf{in} \ a_1, \sigma \rangle \rightarrow \langle \mathbf{let} \ X = a'_0 \ \mathbf{in} \ a_1, \sigma \rangle}$$

# Call by Value

Next, function application:

$$\langle (\mathbf{fun} X \rightarrow a)n, \sigma \rangle \rightarrow \langle [n/X]a, \sigma \rangle$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle}{\langle (\mathbf{fun} X \rightarrow a_0) a_1, \sigma \rangle \rightarrow \langle (\mathbf{fun} X \rightarrow a_0) a'_1, \sigma \rangle}$$

## Call by Name

Using Call by Name, we will perform substitution first. This lets us get by without additional rules to reduce the arguments first:

$$\langle \mathbf{let} \ X = a_0 \ \mathbf{in} \ a_1, \sigma \rangle \rightarrow \langle [a_0/X]a_1, \sigma \rangle$$

$$\langle (\mathbf{fun} \ X \rightarrow a_0) a_1, \sigma \rangle \rightarrow \langle [a_1/X]a_0, \sigma \rangle$$

## Call by Value vs. Call by Name

**Question:** Is there any difference between the two?

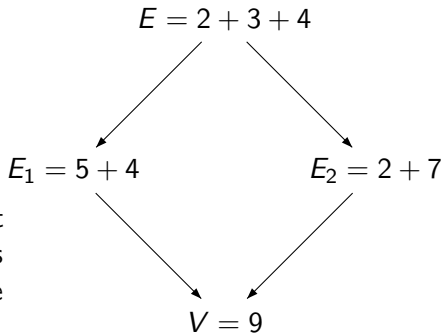
## Call by Value vs. Call by Name

**Question:** Is there any difference between the two?

**Answer:** Maybe, depending on the language...

# Church-Rosser

A language (reduction system) that is *Church-Rosser* is one in which reduction order does not affect the final result (although it can affect convergence): if  $E \rightarrow^* E_1$  and  $E \rightarrow^* E_2$  then there exists a value  $V$  such that  $E_1 \rightarrow^* V$  and  $E_2 \rightarrow^* V$ . This is also known as *confluence* or the *diamond property*.



## Are all Systems Church-Rosser?

**No.** Especially considering side-effects, most languages are *not* Church-Rosser.

- ▶ Anonzo Church and Barkley Rosser proved that the  $\lambda$ -calculus is Church-Rosser in 1936
- ▶ One benefit – can check equality of terms by evaluating them both to canonical forms (if this terminates!)