Semantics of Programming Languages: Operational, Denotational, Axiomatic

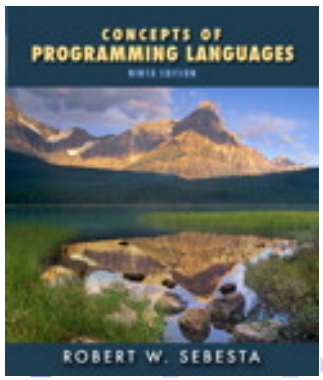
UNIVERSIDADE DA CORUÑA

GRAO EN ENXEÑERÍA INFORMÁTICA

DESEÑO DAS LINGUAXES DE PROGRAMACIÓN

Based on chapters 3 and 4 of: Franlyn Turbak and David Gifford, *Design Concepts in Programming Languages*. The MIT Press, 2008

and chapter 3 of: Robert W. Sebesta, Concepts of Programming Languages, Pearson, 2010.



Outline

Introduction

There are three main characteristics of programming languages:

- Syntax: What is the **appearance and structure** of its programs?
- Semantics: What is the **meaning** of programs?
  - The static semantics tells us which (syntactically valid) programs are semantically valid (i.e., which are type correct)
  - the dynamic semantics tells us **how to interpret the meaning of valid programs**
- Pragmatics: What is the **usability** of the language? How **easy is it to implement**? What **kinds of applications** does it suit?

Syntax vs. Semantics

- Syntax: the **form** or structure of the expressions, statements, and program units
- Semantics: the **meaning** of the expressions, statements, and program units
- Syntax and semantics provide a language's definition.

  Users of a language definition:
  - Other language designers
  - Implementers
  - Programmers (the users of the language)

Static semantics

- Nothing to do with meaning!!
- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Categories of constructs that are trouble:
  - Context-free, but cumbersome (e.g., types of operands in expressions)
  - Non-context-free (e.g., variables must be declared before they are used)

Attribute Grammars

- Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes
- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)

Computing atributes in AG

How are attribute values computed?

- If all attributes were inherited, the tree could be decorated in top-down order.
- If all attributes were synthesized, the tree could be decorated in bottom-up order.
- In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

Dynamic Semantics

There is no single widely acceptable notation or formalism for describing semantics

Several needs for a methodology and notation for semantics:

- Programmers need to know what statements mean
- Compiler writers must know exactly what language constructs do
- Correctness proofs would be possible
- Compiler generators would be possible
- Designers could detect ambiguities and inconsistencies

Uses of Semantic Specifications

Semantic specifications are useful for language designers to communicate with implementors as well as with programmers.

A precise standard for a computer implementation:

How should the language be implemented on different machines?

User documentation:

What is the meaning of a program, given a particular combination of language features?

A tool for design and analysis:

How can the language definition be tuned so that it can be implemented efficiently?

Input to a compiler generator:

How can a reference implementation be obtained from the specification?

Methods for Specifying Semantics

Operational Semantics:

$$[[\text{program}]] = \text{abstract machine program}$$

Denotational Semantics:

$$[[\ \text{program}\ ]] = \text{mathematical denotation}$$

Axiomatic Semantics:

$$[[\ \text{program}\ ]] = \text{set of properties}$$

Operational Semantics

Operational semantics formalizes the common intuition that program execution can be understood as a step-by-step process that evolves by the mechanical application of a fixed set of rules

Sometimes the rules describe how the state of some physical machine is changed by executing an instruction.

But the rules may also describe how language constructs affect the state of some abstract machine that provides a mathematical model for program execution. Each state of the abstract machine is called a configuration

Asbtract machine

- The code component: a program phrase that controls the rest of the computation.
- The state components: entities that are manipulated by the program during its execution.

Operational Execution

1. The program and its inputs are first mapped by an input function into an initial configuration of the abstract machine.
2. Rules governing the abstract machine are applied in an iterative fashion to yield a sequence of intermediate configurations. Each configuration is the result of one step in the step-by-step execution of the program.
3. The last step of execution is mapping the final configuration to an answer via an output function.

Termination

Sometimes an abstract machine never reaches a final configuration:

- The abstract machine may reach a nonfinal configuration to which no rules apply. Such a configuration is said to be a stuck state. Stuck states often model **error situations**

- The rule-applying process of the abstract machine might not terminate (remind the *halting problem* and the limits of computation)

Small-step Operational Semantics

A framework for describing program execution as an iterative sequence of small computational steps.

Characterized by the use of rewrite rules to specify the step-by-step transformation of configurations in an abstract machine

$$\frac{antecedents}{consequents}$$

More about this in the chapters about λ-calculus

Big-step Operational Semantics

We often want to evaluate a phrase by recursively evaluating its subphrases and then combining the results.

This idea underlies denotational semantics but also an alternative form of operational semantics, called big-step operational semantics (BOS), also known as natural semantics

Characteristics of Big-step Operational Semantics

- The steps of a BOS are "big": they tell how to go from a phrase to an answer (or something close to an answer)
- With BOS evaluations there is no notion of a stuck state. However, we can extend the BOS to include an explicit error token as a possible result and modify the rules to generate and propagate such a token.
- Usually BOS rules also do not specify the order in which operands are evaluated
- BOS rules may specify a relation, so it can describe nondeterministic evaluation

Comparing Small-step and Big-step Operational Semantics

- Big-step semantics is often more concise than a small-step semantics, and one of its proof trees can summarize the entire execution of a program
- The recursive nature of a big-step semantics corresponds more closely to the structure of interpreters for high-level languages than a small-step semantics does.
- The iterative step-by-step nature of a small-step semantics corresponds more closely to the way low-level languages are implemented
- Small-step semantics is often a better framework for reasoning about computational resources, errors, and termination
- Infinite loops are easy to model in a small-step semantics but not in a big-step semantics.

We will use small-step semantics as our default form of operational semantics throughout the rest of the course

Denotational Semantics

The essence of this framework is the notion that the meaning of a program phrase can be determined from the meaning of its parts

A denotational semantics determines the meaning of a phrase in a composi- tional way based on its static structure rather than on some sort of dynamically changing configuration.

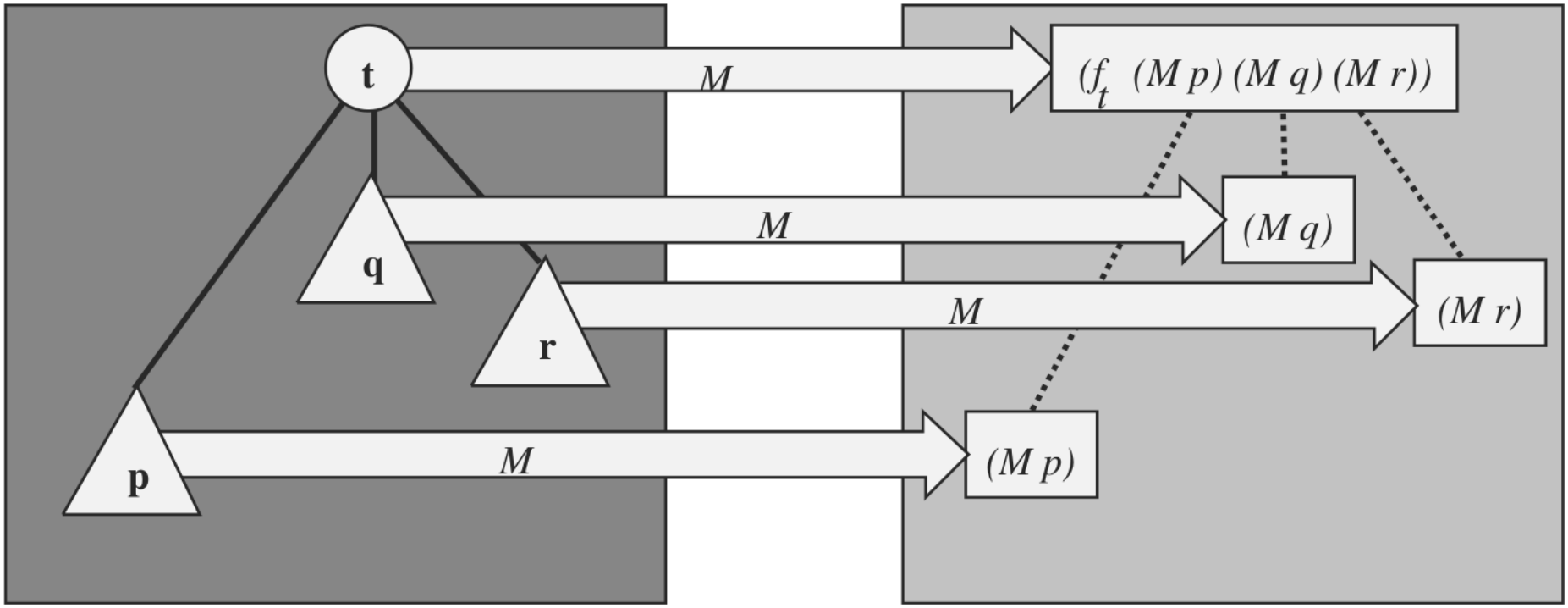
Unlike an operational semantics, a denotational semantics emphasizes what the meaning of a phrase is, not how the phrase is evaluated

The name “denotational semantics” is derived from its focus on the mathematical values that phrases “denote”

Structure of the denotational framework

- A syntactic algebra that describes the abstract syntax of the language under study.
- A semantic algebra that models the meaning of program phrases. A semantic algebra consists of a collection of semantic domains along with functions that manipulate these domains.
- A meaning function, a **homomorphism** between the syntactic algebra and the semantic algebra that maps elements of the syntactic algebra to their meanings in the semantic algebra.
  Each phrase is said to denote its image under the meaning function.
  In practice, the meaning function is specified by a collection of so-called valuation functions, one for each syntactic domain defined by the abstract syntax for the language.

Structure of the denotational framework



Semantic domains

The meaning of a program is an element of a function domain that maps context domains to an answer domain, where:

- Context domains are the denotational analogue of state components in an small-step operational semantics configuration.

  They model such entities as name/value associations, the current contents of memory, and control information.

- An answer domain represents the possible meanings of programs.

  In addition to a component that models what we normally think of as the result of a program phrase, the answer domain may also include components that model context information that was transformed by the program.

Compositionality

Restricting meaning functions to homomorphisms allow us to get a structure-preserving behavior that greatly simplifies reasoning:

the meaning of the whole is composed out of the meaning of the parts

- The meaning of a program remains the same when one of its phrases is replaced by another phrase with the same meaning
- Facilitates the implementation of programming languages: interpreters and translators based on denotational semantics have a natural recursive structure that mimics the recursive structure of the valuation functions and the abstract syntax trees they manipulate.

Denotational vs. Operational Semantics

- An operational semantics is usually a more natural medium for expressing the step-by-step nature of program execution
- An operational semantics provides a natural way to talk about nondeterminism (choice between steps) and concurrency (interleaving the steps of more than one process)
- From a mathematical perspective, the advantage of an operational semantics is that it's often much easier to construct than a denotational semantics:
  - Any small-step operational semantics with a deterministic set of rewrite rules specifies a well-defined behavior function from programs to answer expressions
  - Creating or extending a set of rewrite rules is fairly painless since it rarely requires any deep mathematical reasoning.
  - However, it's difficult to see how some local change to the rewrite rules affects the global properties of a language

Denotational vs. Operational Semantics

- Constructing a denotational semantics is mathematically much more intensive.
  - It is necessary to build consistent mathematical represen- tations for each kind of meaning object
  - Extending the set of meanings requires potentially difficult proofs that the extensions are sound, so most semanticists are content to stick with the well-understood meanings

Axiomatic Semantics

- Based on formal logic (predicate calculus)
- Original purpose: formal program verification
- Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)
- The logic expressions are called assertions

Recommended reading: David Gries. *The Science of Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1987.

Assertions

An assertion before a statement (a precondition) states the relationships and constraints among variables that are true at that point in execution

An assertion following a statement is a postcondition

A weakest precondition is the least restrictive precondition that will guarantee the postcondition

Axiomatic Semantics Form

re-, post form: $\{P\}$ statement $\{Q\}$

An example

- a = b + 1 $\{a > 1\}$
- One possible precondition: $\{b > 10\}$
- Weakest precondition: $\{b > 0\}$

Program Proof Process

The postcondition for the entire program is the desired result

Work back through the program to the first statement

If the precondition on the first statement is the same as the program specification, the program is correct.

Axioms

- Assignment statements

$$(x = E) : \{Q_{x \mapsto E}\}\ x = E\ \{Q\}$$

- The Rule of Consequence:

$$\frac{\{P\}\ S\ \{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

Axioms

- inference rule for sequences of the form S1; S2

$$\{P1\}\ S1\ \{P2\}$$

$$\{P2\}\ S2\ \{P3\}$$

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

Axioms

- inference rule for logical pretest loops

$$\{P\} \text{ while } B \text{ do } S \text{ end } \{Q\}$$

$$\frac{\{I \textit{ and } B\}\ S\ \{I\}}{\{I\} \text{ while } B \text{ do } S\ \{I \textit{ and } (\textit{not } B)\}}$$

where I is the loop invariant (the inductive hypothesis)

Characteristics of the loop invariant

Loop invariant I must meet the following conditions:

$P \rightarrow I$
: the loop invariant must be true initially

$\{I\}\ B\ \{I\}$
: evaluation of the Boolean must not change the validity of I

$\{I \text{ and } B\}\ S\ \{I\}$
: I is not changed by executing the body of the loop

$(I \text{ and } (\text{not } B)) \rightarrow Q$
: if I is true and B is false, Q is implied

The loop terminates can be difficult to prove

Characteristics of the loop invariant

- The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition
- *I* must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition

Summary of Axiomatic Semantics

- Developing axioms or inference rules for all of the statements in a language is difficult
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers
- Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers

End