Programming Language Semantics: Chapter 03 - Formal Semantics

Tarek Boutefara – t_boutefara@univ-jijel.dz – Version 0.1, 13-10-2024

Table of Contents

1. Introduction

Testing a programme is an essential part of ensuring that it does the job it was designed to do. The concept of testing is not unique to the IT field: tests are applied in all areas of engineering to assess the performance of the system or product being built.

Two approaches can be used to test a programme:

1. Use unit tests: these tests are based on a dataset containing the data to be passed as input and the expected results as output. Obviously, this approach depends heavily on the quality of the test dataset. What's more, this approach makes it possible to detect that a programme is not working correctly, but without any additional precision, as it treats the programme under test as a black box. So it was necessary to think about developing another, more precise and accurate approach.
2. Understanding the programme's behaviour, semantics and function. This second approach is known as program proof

and is part of the study of the semantics of programming languages.

However, there are two aspects to describing a program:

1. Its syntax,
2. Its semantics.

In addition, there are several forms of description:

1. By defining how the compiler works,
2. By using natural language,
3. By using a mathematical model.

The operation of the compiler is described using grammars (such as BNF notation). This is a formal grammar that provides a precise, concise and clear description. However, this notation is limited to the syntax of the language.

Natural language description is the most widely used approach for describing the semantics of an instruction or method. However, natural language is ambiguous and imprecise. So it was necessary to think of a formal approach to describing the semantics of a program, in the same way that formal grammars describe the syntax of a language.

2. Definition

Formal Semantics in the context of programming languages is a branch of computer science that provides a precise mathematical specification of the meaning of a programming language. It defines how programs written in the language will be executed or interpreted. This definition is typically given in terms of a formal system, such as a set of axioms and inference rules, that can be used to reason about the behavior of programs.

2.1. Example

In Java, we have two operators:

- `+` : the binary operator is used to calculate the sum of two operands. For example: `5 + 7` gives `12`. If the operator is used several times, the order of execution is from left to right. For example: `5 + 7 + 9` is evaluated as `(5 + 7) + 9` which gives `12 + 9` then `21`.
- ++: The ++ operator is used to increment the value of an integer variable while using its value in an expression. For example: if `x = 3`, then `x++ + 7` gives `10` with x incremented to `4`.

Consider the following expression:

```
x++ + x++ + x++
```

If x has the value 3, what will be the result of this expression?

We can put this expression into a program:

JAVA

```
public class Exemple {

    public static void main (String args[]) {
        int x = 3;
        int y = x++ + x++ + x++;
        System.out.println("La valeur de y est " + y);
    }
}
```

Despite the semantic definition of how the two operators work. From the description given above, it is difficult to envisage the result.

3. Formal Semantics application domains

Formal semantics, with its rigorous mathematical approach to defining programming language meaning, has found applications in various domains within computer science.

3.1. Programming Language Design and Implementation

- **Compiler and Interpreter Development:** Formal semantics provides a solid foundation for constructing correct and efficient compilers and interpreters. By defining the language's semantics precisely, developers can ensure that their implementations adhere to the intended behavior.
- **Language Extensions and Evolution:** When adding new features or modifying existing ones, formal semantics can help assess the impact on the language's overall consistency and correctness.
- **Type Systems:** Formal semantics is crucial for designing and verifying type systems, which play a significant role in preventing errors and improving program reliability.

3.2. Program Verification and Analysis

- **Correctness Proofs:** Formal methods can be used to prove the correctness of programs, ensuring that they meet their specifications and are free from bugs.
- **Static Analysis:** Tools that analyze programs without executing them can use formal semantics to detect potential errors, such as type mismatches or runtime exceptions.
- **Security Analysis:** Formal methods can help identify security vulnerabilities in software systems by analyzing their behavior under various conditions.

3.3. Hardware Design and Verification

- **Hardware Description Languages (HDLs):** Formal semantics is used to define the behavior of hardware components described in HDLs, such as Verilog or VHDL.
- **Hardware Verification:** Formal verification techniques can be applied to prove the correctness of hardware designs, ensuring that they meet their functional specifications.

3.4. Software Engineering and Testing

- **Software Testing:** Formal methods can be used to generate test cases that are more likely to uncover defects in software systems.
- **Model-Driven Development:** Formal semantics can provide a foundation for model-driven development approaches,

where software systems are designed and implemented based on abstract models.

3.5. Theoretical Computer Science

- **Semantics of Programming Constructs:** Formal semantics is used to study the meaning of various programming constructs, such as control flow statements, functions, and data structures.
- **Semantics of Type Systems:** Formal semantics can be used to analyze the properties of type systems and their impact on program correctness.

4. Semantic Design Decisions in Programming Languages

4.1. Data Types and Representation

- **Primitive data types:** What fundamental data types will the language support (e.g., integers, floating-point numbers, characters, booleans)?
- **User-defined data types:** How will users be able to define their own data structures (e.g., classes, structs, records)?
- **Type system:** Will the language be statically typed (types checked at compile time) or dynamically typed (types checked at runtime)?
- **Type inference:** Will the compiler be able to infer types automatically?

4.2. Expressions and Operators

- **Operator precedence:** How will the order of operations be determined (e.g., multiplication before addition)?
- **Operator overloading:** Will operators be allowed to have multiple meanings based on their operands?
- **Short-circuit evaluation:** How will logical operators (AND, OR) be evaluated (e.g., stopping evaluation as soon as the result is known)?

4.3. Control Flow

- **Conditional statements:** How will conditional branching be expressed (e.g., `if`, `else if`, `else`)?
- **Loops:** What types of loops will be supported (e.g., `for`, `while`, `do-while`)?
- **Exception handling:** How will errors and exceptional conditions be handled (e.g., `try`, `catch`, `finally`)?

4.4. Functions and Procedures

- **Parameter passing:** How will arguments be passed to functions (e.g., by value, by reference, by copy)?
- **Function overloading:** Will functions with the same name but different parameters be allowed?
- **Recursion:** Will the language support recursive function calls?

4.5. Memory Management

- **Automatic memory management:** Will the language automatically allocate and deallocate memory (e.g., garbage collection)?
- **Manual memory management:** Will programmers be responsible for managing memory explicitly (e.g., using `malloc` and `free`)?

4.6. Modules and Namespaces

- **Modularity:** How will programs be organized into reusable components (e.g., modules, packages)?
- **Name resolution:** How will conflicts between names be resolved (e.g., using namespaces)?

4.7. Concurrency and Parallelism

- **Threads and processes:** How will concurrent execution be supported (e.g., using threads, processes)?
- **Synchronization mechanisms:** What mechanisms will be provided to coordinate access to shared resources (e.g., locks, semaphores)?

4.8. Other Considerations

- **Type safety:** How will the language prevent type-related errors?
- **Performance:** What factors will influence the language's performance (e.g., memory usage, execution speed)?
- **Readability and maintainability:** How will the language's design contribute to code quality?

5. Types of Formal Semantics

5.1. Operational

It defines the meaning of a program in terms of the computational steps it performs in an idealised execution. Some definitions use structural operational semantics, in which intermediate states are described using the language itself; others use abstract machines.

The operational semantics define for each structure in the language how it is executed on the machine. Execution varies according to the programming paradigm used:

- Imperative programming: the effect of the structure on the state of the memory is defined. Finite automata can be used in this context.
- Functional programming: the value of each expression is defined (application/substitution).

It should be noted that the operational semantics also depend on the input data. However, it differs from physical execution in that it does not include notions linked to the architecture of the physical machine (such as registers, variable addresses, physical limits, etc.).

This semantics is compiler developer oriented.

5.2. Denotational

It defines the meaning of a programme as elements of an abstract mathematical structure, for example by considering the functions of the programming language as certain mathematical functions.

It is based on the definition of an application that associates each programming language structure with an element of the mathematical model. In this way, denotational semantics focuses on what the program calculates and not on how it calculates it. This provides a semantic that is independent of the input data.

This semantics is oriented towards the programming language designer.

6. Axiomatic or logical

It defines the meaning of a program indirectly, by giving the axioms of a logic of program properties. This semantics is based on Hoare's logic and can be seen as proofs of programs.

This semantics is therefore programmer-oriented.

7. Conclusion

Language Semantics is essential for expressing an aspect of programming languages that syntax alone cannot represent. Syntax allows us to define 'how' to write the different structures, but semantics allows us to define 'what' each structure does.

We have presented three types of Formal Semantics:

- Operational Semantics: intended for compiler designers,
- Denotational Semantics: intended for programming language designers,
- Axiomatic Semantics: intended for developers who want to prove programs.

Version 0.1
Last updated 2024-10-20 23:31:45 +0100