Alexandru Ioan Cuza University
of Iaşi

A Complete Semantics for Java

PhD Thesis Summary

Author:
Denis Bogdănaş

Adviser:
Prof. Dr.
Dorel Lucanu

May 2015

1 Introduction

Java is the second most popular programming language (`http://langpop.com/`), after C and followed by PHP. Both C and PHP have recently been given formal semantics . Like the authors of the C and PHP semantics, and many others, we firmly believe that programming languages *must* have formal semantics. Moreover, the semantics should be public and easily accessible, so inconsistencies are more easily spotted and fixed, and formal analysis tools should be based on such semantics, to eliminate the semantic gaps and thus errors in such tools. Without a formal semantics it is impossible to state or prove anything about the language with certainty, including that a program meets its specification, that a type system is sound, or that a compiler or interpreter is correct. While all analysis tools or implementations for the language invariably incorporate some variant of the language semantics, or a projection of it, these are hard to access and thus to asses.

To the best of our knowledge, the most notable attempts to give Java a formal semantics are ASM-Java , which uses abstract state machines, and JavaFAN , which uses term rewriting. However, as discussed in Chapter 2, these semantics are far from being complete or even well tested. Each comes with a few sample Java programs illustrating only the defined features, and each can execute only about half of the other's programs.

We present K-Java , a semantics for Java which syste-

matically defines every single feature listed in the official definition of Java 1.4, which is the Java Language Specification (JLS) , a 456-page 18-chapter document. Moreover, our semantics is thoroughly tested. In fact, we spent about half the time dedicated to this project to write tests, which are small Java programs exercising special cases of features or combinations of them. Specifically, we followed a Test Driven Development methodology to first develop the tests for the feature to be defined and interactions of it with previous features, and then defined the actual semantics of that feature. This way we produced a comprehensive set of 840 tests, which serves as a conformance test suite not only for our semantics, but also for testing various other Java tools. Considering that no such conformance test suite exists for Java, our tests can also be regarded as a contribution made by this thesis.

As a semantic framework and development tool for our Java semantics we chose $\mathbb{K}$. There are several appealing aspects of $\mathbb{K}$ that made it suitable for such a large project. $\mathbb{K}$ provides a convenient notation for modular semantics of languages, as well as automatically-generated execution and formal analysis tools for the defined languages, such as a parser and interpreter, state-space explorer for reachability, and model-checker.

To emphasize that our Java semantics is useful beyond just providing a reference model for the language, we show how the builtin model-checker of $\mathbb{K}$ can be used to model-check multi-threaded Java programs. While this illustrates only one possible application of the semantics, other

applications have the potential to be similarly derived from the language-independent tools that are under development as part of $\mathbb{K}$.

2 Contributions

The specific contributions of this thesis are:

- K-Java, the first complete semantics of Java 1.4, including multi-threading. More generally, K-Java is the first complete semantics for an imperative statically typed object-oriented concurrent language. In order to maintain clarity while handling the semantics great size we split the semantics into two parts, pipelined together: static semantics (Chapter 5) and dynamic semantics (Chapter 6).
- A demonstrative application — LTL model-checking of multithreaded programs (Chapter 7).
- A comprehensive test suite covering all Java constructs (Chapter 8).
- Application of the test suite to evaluate the completeness of other executable semantics of Java (Chapter 2).
- A language-independent Abstract Syntax Tree transformer, used to connect $\mathbb{K}$ framework to an external parser (Chapter 4).

3 Chapters Summary

2 Related Work

Here we discuss two other major formal executable semantics of Java and compare them with K-Java. We also recall other large language semantics that influenced the design of K-Java.

3 Introduction to $\mathbb{K}$ Framework

In this chapter we first include a tutorial for $\mathbb{K}$, for readers new to $\mathbb{K}$ framework, that will help understanding the rest of the thesis. Then we describe how $\mathbb{K}$ can be used to define semantics in Abstract Syntax Tree format.

4 Parsing Java Programs

Here we present a language-independent parser generator tool, which we developed alongside K-Java to produce a Java parser suitable for interfacing with $\mathbb{K}$. This tool was also used to generate a parser for the K-based PHP semantics.

5 Static Semantics

K-Java is divided into two separate definitions: static semantics (covered in this chapter) and dynamic semantics (the next chapter).

The static semantics takes as input the AST representation of a Java program and produces a preprocessed program as the output. It performs computations that could be done statically, and are referred in JLS as compile-time operations. Such computations include converting each simple class name into a fully qualified class name or computing the static type of each expression. The preprocessed AST is then passed to the dynamic semantics for execution.

We choose to present static K-Java from two complementing perspectives. First, the functionality is illustrated as a set of transformations over programs. Next, the inner workings are presented as a sequence of phases, detailing how the configuration content changes during each phase.

6 Dynamic Semantics

This chapter contains the actual definition, using $\mathbb{K}$ rules, of a wide portion of dynamic K-Java. The sections in this chapter contain:

1. The full configuration of dynamic K-Java.
2. Key auxiliary notation.
3. Selected expressions, including a few numeric and a few reference expressions.
4. Selected statements: if, block, while and exception handling.

5. The portion of the configuration representing the memory model.

6. Rules for variable access for various kinds of variables.

7. The instantiation of new objects.

8. Method invocation, among the most complex parts of dynamic K-Java.

9. Multithreading constructs: thread creation, synchronization and wait/notify mechanism.

7 Applications

Here we show how K-Java together with builtin $\mathbb{K}$ tools can be used to explore multi-threaded program behaviors. The first application is state space exploration and the second is LTL model-checking.

8 Testing

Testing K-Java took almost half of the overall development time. Here we describe our testing efforts, which resulted in what could be the first publicly available conformance test suite for Java.

9 Conclusion

We have presented K-Java, which to our knowledge is the first complete formal semantics of Java. The semantics has been split into a static and a dynamic semantics, and the static semantics was framed so that its output is also a valid Java program. This way, it can seamlessly be used as a frontend in other Java semantics or analysis tools. As a side contribution we have also developed a comprehensive conformance test suite for Java, to our knowledge the first public test suite of its kind, comprising more than 800 small Java programs that attempt to exercise all the corner cases of all the language constructs, as well as non-trivial interactions of them.