K-Java: A Complete Semantics of Java

Denis Bogdănaş
Alexandru Ioan Cuza University, Iaşi, Romania
denis.bogdanas@info.uaic.ro

Grigore Roşu
University of Illinois at Urbana-Champaign
grosu@illinois.edu

Abstract

This paper presents K-Java, a complete executable formal semantics of Java 1.4. K-Java was extensively tested with a test suite developed alongside the project, following the Test Driven Development methodology. In order to maintain clarity while handling the great size of Java, the semantics was split into two separate definitions – a static semantics and a dynamic semantics. The output of the static semantics is a preprocessed Java program, which is passed as input to the dynamic semantics for execution. The preprocessed program is a valid Java program, which uses a subset of the features of Java. The semantics is applied to model-check multi-threaded programs. Both the test suite and the static semantics are generic and ready to be used in other Java-related projects.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

Keywords Java, mechanized semantics, K framework

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 [20, 22]. 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 [35], which uses abstract state machines, and JavaFAN [21], which uses term rewriting. However, as discussed in Section 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.

[Copyright notice will appear here once 'preprint' option is removed.]

We present K-Java, a semantics for Java which systematically defines every single feature listed in the official definition of Java 1.4, which is the Java Language Specification, 2nd edition (JLS) [25], 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 paper.

As a semantic framework and development tool for our Java semantics we chose $\mathbb{K}$ [33] (`http://kframework.org`). 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, model-checker, symbolic execution engine, deductive program verifier, and, recently, a translator of semantics to Coq. In particular, recent advances in matching and reachability logic [17, 32] have been incorporated in $\mathbb{K}$, making it unnecessary to define multiple semantics for the same language, e.g. an operational semantics for execution and an axiomatic semantics for program verification. The C [20] and PHP [22] semantics mentioned above have both been defined in $\mathbb{K}$, and demonstrated their usefulness using the generic $\mathbb{K}$ tools; the MatchC prover [5, 31], for example, automatically turns the C semantics into a Hoare-style deductive verifier of C programs.

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 can be similarly derived from the language-independent tools provided by $\mathbb{K}$.

Besides such immediate applications, we believe that our executable semantics of Java is also a convenient platform for experimenting with Java extensions. For example, [9] proposes to extend Java with traits and records, [23] with mixins, and [27] with immutable objects. The proposal to extend Java with generic types [14] made it to the language in Java 5. The widely debated lambda expression feature, with at least three evolving proposals [29], was finally incorporated in Java 8. Such extensions are easy to add to our semantics, and thanks to $\mathbb{K}$ one would immediately benefit not only from a reference model for them but also from all the formal analysis tools automatically offered by $\mathbb{K}$.

This paper makes the following two specific contributions:

- 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. Section 3 illustrates some challenges of formalising Java, Section 4 describes our definitional methodology and shows selected rules, and Section 5 shows an application to model-checking multithreaded programs.
- Comprehensive test suite covering all Java constructs (Section 6), also used to compare Java semantics in Section 2.

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.

2.1 Other Executable Semantics of Java

ASM-Java [35] is the first attempt to define a complete semantics of Java 1.0, and the most complete prior to K-Java, showing that is it feasible to define formal semantics to real-life programming languages. It was defined using Abstract State Machiness (ASMs) and covers almost all major features of Java, except packages. ASM-Java is also executable; with the kind help of its authors we were able make it work. ASM-Java comes with 58 tests that touch all their implemented features, which we used as one of our external test suites for K-Java. ASM-Java contains not only the semantics of Java, but also defines the compiler and the bytecode format, and gives a manual proof for the correctness of their compiler.

While K-Java uses a different formalism, it follows the ASM-Java methodology to separate the static and the dynamic semantics. Except for that, ASM-Java and K-Java are quite different. ASM-Java uses many auxiliary constructs in their preprocessed programs produced by the static semantics, while K-Java uses (a subset of) plain Java. ASM-Java is monolithic, while in K-Java the static and the dynamic semantics are two separate definitions that can be used independently. For example, other projects can use our static semantics to reduce the set of Java programs they need to handle.

JavaFAN [21] is another large-scale executable semantics of Java, defined using term rewriting in Maude [16]. Our testing revealed that JavaFAN is overall less complete than ASM-Java, although JavaFAN passed a few tests that ASM-Java failed to pass. However, unlike ASM-Java, thanks to the high-performance Maude model-checker [18], JavaFAN was successfully used to perform state-space exploration and model-checking of Java programs. For example, JavaFAN was able to detect the deadlock in the Dining Philosophers problem and prove the fixed program correct.

Comparison. Since both ASM-Java and JavaFAN are executable, we evaluated their completeness by running our comprehensive test suite (Section 6) with each. The results of our findings are presented in Figure 1. The list of Java features is divided into 10 large groups, separated by horizontal bars. The first 8 groups contain features introduced with Java 1.0: literals, expressions, statements, arrays, classes and instance members, interfaces, static members and static initialization, packages and identifier resolution. Group 9 includes features introduced with Java 1.2, and the last group includes the single new feature of Java 1.4 - `assert`.

Besides packages, ASM-Java does not define all literal forms, an important set of operators, the switch statement, array initializers, and complex cases of method overloading and method access modes. The remaining features of Java 1.0 are defined, except for some corner cases. JavaFAN, despite being more recent than ASM-Java, covers a smaller subset of Java. Yet, it surpasses ASM-Java in a few areas. JavaFAN supports a wider set of operators, but still not all of them, and it has better support for local variable shadowing. Yet many other features are not supported: switch and continue, advanced array features, many class-related features and interfaces. Also, JavaFAN makes no distinction between various integer types and integer overflow is not defined. From Java 1.2 and 1.4, it only supports the simplest feature, instance initializers.

Feature	AJ	JF	KJ
Basic integer, boolean, String literals	●	●	●
Other literals	○	○	●
Overflow, distinction between integer types	●	○	●
Prefix `++i` `--i`, also `+=` `-=` `...` `,` `||` `&&`	○	●	●
Bit-related operators: `|` `&` `^` `>>` `<<` `>>>`	○	○	●
Other integer operators	●	●	●
String + <other types>	◐	◐	●
Reference operators	●	◐	●
Basic statements	●	●	●
Switch	○	○	●
Try-catch-finally	●	○	●
Break	●	◐	●
Continue	●	○	●
Array basics	●	●	●
Array-related exceptions	●	○	●
Array length	●	○	●
Array polymorphism	●	○	●
Array initializers	○	○	●
Array default values	●	○	●
Basic OOP - classes, inheritance, polymorphism	●	●	●
Method overloading – distinct number of arguments	●	●	●
Method overloading without argument conversion	●	○	●
Method overloading with argument conversion	○	○	●
Method access modes	○	○	●
Instance field initializers	●	◐	●
Chained constructor calls via `this()` and `super()`	●	○	●
Keyword `super`	●	○	●
Interfaces	●	○	●
Interface fields	●	○	●
Static methods and fields	●	●	●
Accessing unqualified static fields	◐	○	●
Static initialization	◐	◐	●
Static initialization trigger	●	○	●
Packages	○	○	●
Shadowing	◐	●	●
Hiding	●	○	●
Instance initialization blocks	○	●	●
Static inner classes	○	○	●
Instance inner classes	○	○	●
Local & anonymous classes	○	○	●
Assert statement	○	○	●

Support level: ● = Full ◐ = Partial ○ = None

AJ represents ASM-Java [35], JF is JavaFAN [21] and KJ is our work.

Figure 1. Completeness comparison of various semantics of Java

Methodology. The level of granularity differs widely both among groups and among individual rows in Figure 1. We intentionally compressed large portions of Java into one row when all semantics defined them (e.g., Basic OOP). Yet, when we identified interesting features not supported by some of the semantics, we extracted them into individual rows. We were careful to interpret the test results objectively. In particular, we designed each test to touch as few Java features as possible other than the tested one, to minimize the number of reasons for failure (Section 6). In the rare cases when we were unable to identify why a particular test failed, we gave the semantics the benefit of the doubt (except for K-Java).

2.2 Other Large-Scale Executable Semantics

Several large-scale semantics have been defined recently. Due to space constraints, we only mention those which had a direct influ-

ence on K-Java. The first large-scale semantics developed in $\mathbb{K}$ was that of C [20]. It covered all the C features, was tested using the GCC torture test suite [24], and was used to explore C undefinedness. A large fragment of PHP [22], also defined in $\mathbb{K}$, was tested for conformance using the PHP standard test suite and applied to prove simple properties using model-checking. Two different semantics of large fragments of Python were defined using $\mathbb{K}$ [26] and resp. PLT Redex [30]. Both were tested against publicly available test suites. The ASM-Java techniques were also successfully reused to define C# [13], benefiting from the fact that Java and C# are similar. JSCert [10] is an almost complete semantics for JavaScript in Coq, using a methodology that ensured a close visual resemblance to the standard, and tested against the JavaScript standard test suite.

3. Background and Challenges

Here we give background on Java and $\mathbb{K}$, and discuss some of the major challenges faced when giving semantics to Java.

3.1 The Java Language Specification

Java is a statically, strongly typed, object-oriented, multi-threaded language. It is completely defined, i.e., unlike other languages [10, 20], it has no undefined or implementation-dependent features. Except for threads, Java is completely deterministic. The official definition of Java 1.4 is the JLS [25]. JLS has 456 pages and 18 chapters; the part that defines the language has 377 pages. Java is distributed as part of the Java Development Kit (JDK), together with a several thousand class library. The class library, however, is *not* part of the JLS. Nevertheless, we defined semantics to all classes that are central to the language and mentioned in the JLS, such as `Object`, `String`, `Thread` and a dozen exception types.

Challenges. The K-Java project faced formidable challenges. The first challenge is the sheer size of Java. At the imperative level the language has 36 operators (JLS §15) and 18 statements (§16). Java values may be subject to a large number of conversions (§5). There are complex rules for resolving an identifier (§6). More precisely, an identifier in Java could mean one of the following: package, class, field, local variable or method. Method name could be decided based on the immediate syntactic context alone. However, disambiguating between the remaining categories is non-trivial and involves many special cases. One of the most complex features of Java is method overloading, that we illustrate in the next subsection. Classes have complex initialization rules that include static initialization (§8.3.2, 12.4) and instance initialization (§8.3.2, 12.5). The matter is further complicated by the ability to call on the first line of a constructor either a constructor of the same class through `this()` or a constructor of the base class through `super()`. Interfaces interact with a wide number of features as they may have methods, static fields, specific rules for initialization and method overwriting.

Java has a number of modularity features, such as packages, imports, and four categories of nested classes: static inner, non-static inner, local and anonymous. Since nested classes may access names from their enclosing classes, they bring a large number of special cases for name resolving. Packages are important to define access modes, and access modes have challenging interactions with the other Java features, as will be illustrated in Section 3.2.

The separation of the whole semantics into static and dynamic definition is a consequence of Java being statically typed. Dynamically typed languages like PHP or JavaScript need just a dynamic semantics. JLS clearly defines what computations should happen before the execution (compile-time) and what should happen during the execution (runtime). In Section 3.2 we present an example that illustrates the difficulties produced by static typing. While it might seem natural to have the two semantics for Java, we did not follow this approach from the beginning. How static typing influenced the design decisions of K-Java is discussed in Section 4.1.

Another challenge was the careful testing and implementation of all corner cases for each new feature. The difficulty arises when the new feature interacts with already defined features. For example, packages were among the last features added. We had to make sure packages were properly resolved in all contexts — in variable declarations, extends/implements clauses, cast expressions, etc. When we later added inner classes, we had to make sure inner classes work equally well in all the contexts above. For each context we had to test that inner classes might be referred both by simple names and by fully qualified names, that might contain package names. Our testing methodology is presented in Section 6.

Despite these challenges, we made no compromises and completely defined every single feature of Java 1.4.

JLS content unrelated to K-Java. Java was designed to be compiled into a bytecode format that is executed on a hardware-independent Java Virtual Machine (JVM). Consequently, some details of JLS deal specifically with the bytecode representation and are irrelevant here. Such parts are §12.1-12.3, 12.6-12.8 (details of JVM execution) and §13 (bytecode binary compatibility). JLS also defines all the compile-time errors that might be produced by incorrect programs. We do not cover them, as the focus of K-Java is to model the behavior of valid Java programs only.

Also, we do not cover Dynamic Class Loading (DCL), §12.2. Instead, in K-Java all classes are loaded at the beginning of the execution. JLS mentions the possibility to load classes at runtime, from other formats than bytecode, but sends the reader to the JVM specification for details. No other details are given. For this reason, it is fair to consider DCL a JVM rather than a Java feature, and to regard the default class loader (the one loading classes from ".class" files from the startup classpath) purely as a performance optimization, with no implication to the semantics.

Limitations. We do not define the Java Memory Model (JMM) that governs the allowed multi-threaded execution behaviors (§17). Instead, we define a sequentially consistent memory model with a global store and no caches, consistent with the JLS only if all fields are `volatile`. Defining JMM would be a significant effort on its own, relatively orthogonal to the present work. Also, JMM 1.4 was replaced by JMM 5, so is no longer actual. It has been demonstrated that $\mathbb{K}$ allows for modular replacement of the memory model [19, §4.2.6]; we plan to do the same for Java in future work.

3.2 A Flavor of Java: Static Typing and Access Modes

Every Java expression has a statically assigned type. Static types have various functions during execution:

- Subexpression types influence the type of the parent expression. For example, `1 + 2` is not the same as `1 + (long)2`.
- Integers' type gives their precise representation within the allowed range. When the range for a certain type is exceeded, overflow occurs: `1000` is different from `(byte)1000`.
- When an object has a member (field or static method) that hides an inherited member one can pick the right member by manipulating the type of the qualifier. Given `b` of type `B`, then `b.v` and `((A)b).v` could refer to different fields.
- Method overloading allows methods with the same name and number of arguments, but with different argument types. Then `f(0)` and `f((long)0)` may invoke different methods.

For most expressions, the static type might be computed at the same time as the actual value of the expression. One might think that static types could be computed during execution at the same time as the value of an expression is computed. We actually did this in an older version of the semantics, and it worked great for a while. However, we had to rethink this approach when the time came to define the ternary conditional operator `_?_:_`. An expression

```
class A {
  private String f(int a) {return "int";}
  String f(long a) {return "long";}

  String test() {
    return f((byte)0); }          //int
}
class B {
  String test() {
    return new A().f((byte)0); }  //long
}
```

Figure 2. Method overloading with access modes

```
package a;
public class A {
  void f() { ... }
  void g() { ... }
}

package a;
public class B extends A {
  protected void f() { ... }
  void g() { ... }
}

package b;
import a.*;
public class C extends B {
  protected void f() { ... }
  protected void g() { ... }
}

A c = new C(); c.f();   //C.f()
               c.g();   //B.g()
```

Figure 3. Package access mode and method overriding

`a ? b : c` evaluates to `b` when `a` is true, and to `c` otherwise. When `b` and `c` have different static types, the conditional expression will have the *join type* of the two[1]. Thus, the static type of the operator depends on the type of the two arguments, yet only one of these arguments is evaluated at runtime. It would be incorrect to compute the value of both `b` and `c` just for the sake of having their static type (due to possible side effects). Therefore the computation of static types had to be separated from the actual execution.

One of the most complex features of Java is method overloading. First, the arguments of a method call may have types that are different from the parameter types. When the method is overloaded, the version with the *most specific* types that are compatible with the argument types is invoked. Moreover, the choice of available versions of the method is influenced by access modes.

In Figure 2, both versions of `f()` are compatible with the call `f((byte)0)`, yet the set of accessible versions is different for the two call sites. The call from `A.test()` has access to both versions, and it chooses the more specific one. However, the call from `B.test()` cannot access the private `f()`; it calls the one with default access mode. The situation is even more complex when overloading is combined with inheritance and subtype polymorphism.

One might be tempted to think that access modes are only required for static semantics and could be discarded before execution.

[1] JLS §15.25 puts some restrictions on the types of arguments of `_?_:_`, that makes join in most cases to be the widest type of arguments 2 and 3.

In Java this is not the case. In Figure 3 the two methods `f()` and `g()` are very similar, yet the declaring class of the actually invoked methods is different. Since `A.f()` is declared with package access mode, and class `C` is in a different package, the method `C.f()` does not directly override `A.f()`. However, it does override `A.f()` *indirectly*, through the declaration `B.f()`. The method `A.f()` is accessible to `B` since both `A` and `B` are in the same package. As `B.f()` is declared with protected access mode, the method is accessible to any subclass of `B`, including to `C`. Thus, a link is established that allows `C.f()` to override `A.f()`. The very similar method `g()` is declared with the default (package) access mode in `B`, that prevents the connection between `B.g()` and `C.g()`. Hence, in order to determine the right method to be invoked, it is not sufficient to pick the last one visible to the dynamic (runtime) type of the qualifier. A correct semantics has to analyse the entire chain of classes between the static type of the qualifier and the dynamic type.

The examples above illustrate how static typing allows for a rich set of features, but it also brings significant complexity. Moreover, it is not enough to consider each language feature in isolation. It is required to define and *test* each combination that might lead to non-trivial interactions. Section 6 addresses testing. Some rules related to method invocation are presented in Section 4.3.

3.3 $\mathbb{K}$ (`http://kframework.org`)

$\mathbb{K}$ [33] is a framework for engineering language semantics. Given a syntax and a semantics of a language, $\mathbb{K}$ generates a parser, an interpreter, as well as formal analysis tools such as model checkers and deductive theorem provers, at no additional cost. It also supports various backends, such as Maude and, experimentally, Coq, that is, it can translate language semantics defined in $\mathbb{K}$ into Maude or Coq definitions. The interpreter allows the semanticists to continuously test their semantics, significantly increasing their effectiveness. Furthermore, the formal analysis tools facilitate formal reasoning for the given language semantics, which helps both in terms of applicability of the semantics and in terms of engineering the semantics itself; for example, the state-space exploration capability helps the language designer to cover all the non-deterministic behaviors of certain language constructs or combinations of them.

Here we describe $\mathbb{K}$ only very briefly. More details are given on an as-needed basis in the rest of the paper. In $\mathbb{K}$, a language syntax is given using conventional BNF annotated with semantic attributes. A language semantics is given as a set of reduction rules over configurations. A configuration is an algebraic structure of the program state, organized as nested labeled cells holding semantic information, including the program itself. Figure 4 shows a subset of the Java configuration. The order of cells is irrelevant in a configuration. Leaf cells contain pieces of the program state like a computation stack or continuation (e.g., k), environments (e.g., env), stacks (e.g., stack), etc. As this is all best understood through an example, let us consider a typical rule for reading a variable:

$$\left\langle \frac{X}{V} \cdots \right\rangle_{\mathsf{k}} \ \langle \cdots X \mapsto L \cdots \rangle_{\mathsf{env}} \ \langle \cdots L \mapsto V \cdots \rangle_{\mathsf{store}}$$

We see here three cells, k, env and store. The k cell represents a list (or stack) of computations waiting to be performed. The left-most (i.e., top) element of the stack is the next item to be computed. The env cell is simply a map of variables to their locations. And the cell store is a map from locations to values. The rule above says that if the next thing to be evaluated (which here we call a redex) is the variable lookup expression X, then one should match X in the environment to find its location L in memory, and match L in the store to find its value V. With this information, one should transform the redex into that value, V; the horizontal line represents a reduction. A cell with no horizontal line means that it is read

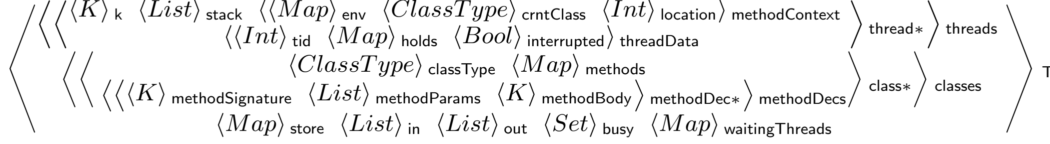

Figure 4. Subset of the Java Configuration

but does not change during the reduction. The "..." are structural frames, that is, they match portions of cells that are irrelevant.

This unconventional notation is actually quite useful. The above rule would be written out as a traditional rewrite rule like this:

$$\langle X \curvearrowright \kappa\rangle_{\mathsf{k}} \, \langle \rho_1, X \mapsto L, \rho_2\rangle_{\mathsf{env}} \, \langle \rho_3, L \mapsto V, \rho_4\rangle_{\mathsf{store}}$$
$$\Rightarrow \langle L \curvearrowright \kappa\rangle_{\mathsf{k}} \, \langle \rho_1, X \mapsto L, \rho_2\rangle_{\mathsf{env}} \, \langle \rho_3, L \mapsto V, \rho_4\rangle_{\mathsf{store}}$$

Items in the k cell are separated with "$\curvearrowright$", which can now be seen. The κ and ρ_1, ρ_2, ρ_3, ρ_4 take the place of the "⋯" above. The most important thing to notice is that nearly the entire rule is duplicated on the right-hand side (RHS). Duplication in a definition requires that changes be made in concert, in multiple places. If this duplication is not kept in sync, it leads to subtle semantic errors. In a complex language like Java, the configuration is much more complex, and would require including additional cells like methodContext, thread and threads (Figure 4). $\mathbb{K}$ automatically infers these intervening cells, keeping the rules local and modular.

In fact, one of the most appealing aspects of $\mathbb{K}$ is its modularity. It is very rarely the case that one needs to touch existing rules in order to add a new feature to the language. There are two $\mathbb{K}$ features specifically targeting modularity. First, the configuration can be structured as nested cells. Second, the language designer needs only to mention the cells that are needed in each rule, and only the needed portions of those cells. For example, the above rule only refers to the k, env and store cells, while the entire configuration contains many other cells as shown in Figure 4. This modularity contributes not only to compact and thus human readable semantics, but also to the overall effectiveness of the semantics development process. For example, even if a new cell is added to the configuration later on in order to support a new feature, the above rule does not change.

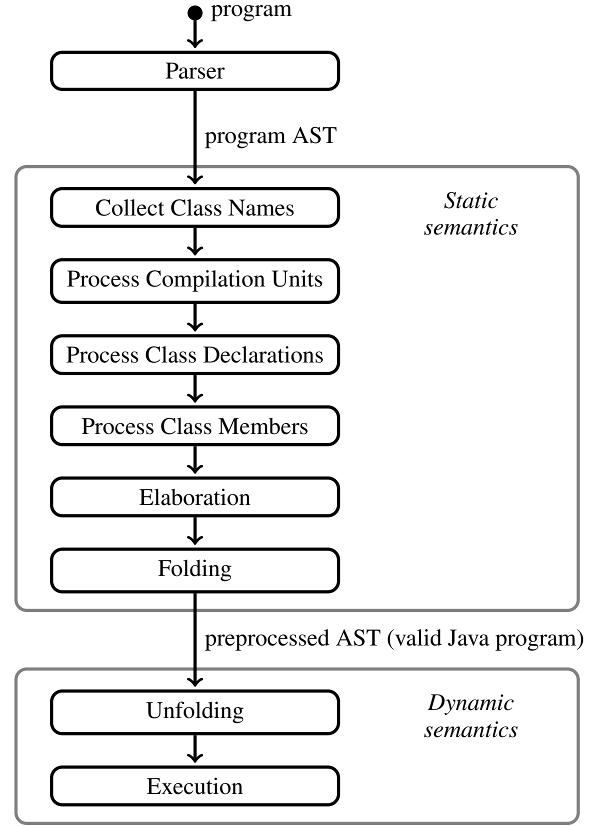

Figure 5. The structure of K-Java

4. K-Java: The Semantics of Java in $\mathbb{K}$

K-Java is divided into two separate definitions: static semantics and dynamic semantics (Figure 5). For parsing we use JavaFront [15], a Java grammar defined in SDF [28], which we turned into a parser compatible with $\mathbb{K}$ by using the SDF-to-K adapter [11].

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 passed to the dynamic semantics for execution.

Statistics We define the semantics for all 186 syntactic productions of Java. Here are some statistics for our semantics:

	Static	Dynamic	Common	Lib	Total
SLOC	3112	2035	539	497	6183
CLOC	857	1189	383	98	2527
Size (KB)	168	139	52	22	381
N cells	31	36	28	–	95
N rule	497	347	183	47	1074
N aux	111	83	79	9	282

The rows represent source lines of code (SLOC), comment lines of code (CLOC), total files size, number of cells, number of rules, number of auxiliary functions. The columns represent the static semantics, the dynamic semantics except class library, modules common for both static and dynamic semantics, class library (including multi-threading) and the total size.

Next we present an overview of the static semantics and continue with selected rules from the dynamic semantics.

4.1 Static Semantics

The static semantics consists of several phases, depicted in Figure 5. Each phase digs deeper into the syntactic structure of Java and either performs a set of transformations over the program or computes some new configuration cells.

Collect class names phase. In this phase the initial program is traversed and all class names are collected into a map. The names map serves two purposes at later phases: to resolve simple class names into fully qualified class names and to determine the set of classes available inside a package. Each class is traversed recursively, so that the inner classes are also registered. Yet the traversing does not proceed inside code blocks, thus classes defined at the block level (local and anonymous) are not discovered at this stage.

Process compilation units phase. In this phase the traversal starts again from the initial program AST and performs two tasks. The first is to move class content from the initial AST to the class cells inside the configuration, from where it will be easier to access during subsequent phases. From this point on the program is no longer a monolithic AST but a collection of class cells. The granularity of the initial class content is very rough. At this stage just the class declaration, including `extends/implements` clauses, is analysed, and not the class members. The class content is analysed recursively like in the the previous phase, to process all the inner classes. The second task of this phase is to process imports and produce the map of class names accessible for each top-level class.

Process class declarations phase. In this phase the simple class names inside `extends/implements` clauses of each class are converted into fully qualified class names. This allows us to compute the map of accessible class names for inner classes. JLS has complex rules for the names accessible inside inner classes that do not allow computing those names at an earlier stage.

Process class members phase. At this stage the body of each class is processed and each member is stored in an appropriate cell. For methods, the parameter and return types are now resolved.

Elaboration phase. This phase involves analysing the code blocks and is the most complex of all phases. The transformations here involve resolving class names inside blocks, resolving expression types and method overloading. Also at this stage block-level classes (local and anonymous) are first detected. Upon their detection, block-level classes must be passed through all the previous phases, to be normalized to the same format as the other classes.

Transformations. Figure 6 depicts most of the transformations performed by static K-Java. Left column contains sample input fragments together with the minimal relevant context. The right column represents the transformed fragment. These transformations illustrate the means by which it was possible to represent the preprocessed program as valid Java, without any extra annotations.

The simple class names are transformed into fully qualified class names (row 1). This allows us to eliminate imports clauses in preprocessed programs. Each expression is additionally wrapped into a cast expression that encodes its static type (row 2). During execution this cast is treated like a regular cast. The example in the figure also illustrates binary numeric promotion: the result type of the addition between an `int` and a `long` is a long.

In addition, for method calls each argument is wrapped into one more cast, namely to the expected parameter type (row 3). This allows us to effectively encode method overloading.

Row 4 illustrates the preprocessing of fields accessed by simple names. Each such field access is converted into a qualified access. If the field is non-static, it will be qualified by `this` (first case in row 4), and `this` should be additionally cast to the current class type, according to rule 2. If the field is static, it will be qualified with the class that defined the field (second case). The same transformation is performed for methods called without a qualifier.

This last transformation allows us to disambiguate among the four roles of a name, namely package, class, field or local variable:

Syntax	Syntactic Context	Resulting Role
`Id`	package	package
`Id`	package or class	package
`Id`	class	class
`((Type) Qual.Id)`	expression	field
`((Type) Id)`	expression	local var

An identifier inside a syntactic context where only a package is allowed will be considered a package. If either a package or a class is allowed, then the identifier will also be considered a package

Initial code	**Preprocessed code**
`Object`	`java.lang.Object`
`1 + 2L`	`(long)((int)1+(long)2)`
`void f(long a);` `f(1);`	 `f((long)(int)1);`

`class A{ int v;`
`  static int s;}`
`v`
`s`
	
	`(int)((A) this).v`
	`(int)A.s`

`class A{ A(){} }`	`class A {`
	`  A(){ super(); }`
	`}`

`class A {`
` static int a=1;`
` static{a++;}`
`}`
	`class A {`
	` static int a;`
	` static{a=1;a++;}`
	`}`

`class A {`
` int a=1;`
` {a++;}`
` A(){`
`  super();`
` }`
`}`
	`class A {`
	` int a;`
	` void $init(){`
	`  a=1;a++;`
	` }`
	` A(){`
	`  super();`
	`  $init();`
	` }`
	`}`

`class O {`
` void m() {`
`  final`
`   int a=1,b=2;`
`  class L {`
`   int f(){`
`    return a+b;`
`   }`
`  };`
`  new L().f();`
` }`
`}`
	`class O {`
	` void m() {`
	`  final int a=1,b=2;`
	`  LEnv $env`
	`    =new LEnv();`
	`  $env.a=a;`
	`  $env.b=b;`
	`  new L($env).f();`
	` } //m()`
	
	` class LEnv{`
	`  int a,b;`
	` }`
	` class L{`
	`  LEnv $env;`
	`  L(LEnv $env){`
	`   this.$env=$env;`
	`  }`
	`  int f(){`
	`   return`
	`     $env.a+$env.b;`
	`  }`
	` } //L`
	`} //O`

`final int a;`
`new A() {`
` int f() {`
`  return a;`
` }`
`}.f();`
	`final int a;`
	`class LocalA`
	`  extends A{...};`
	`new LocalA().f();`

Figure 6. Transformations performed by the static semantics

(here we consider classes that are used as qualifiers to refer to an inner class also as packages). A field or a local variable will always be part of a cast expression, thus will always be inside an expression context. If the respective identifier is qualified, then it is a field. Otherwise it is a local variable.

Continuing with row 5, if a constructor does not call another constructor on the first line, then it should call the base class constructor, `super()`, except if the current class is `Object`. For legibility, we omit the already discussed transformations (e.g., wrapping into cast expressions) in the remaining examples.

Initializers of static fields as well as static initializing blocks are collected into one big static initializing block (row 6), which is invoked during execution when the class is for the first time accessed by the program. For instance initializers the same procedure cannot be followed (row 7). Instead, all the instance initialization code is collected into the method `$init()`, that is called by the constructors right after the base constructor `super()` was called.

Arguably the most complex transformation is that of local classes. It is encoded into a separate module with 26 rules, one of the largest of K-Java. Local classes are converted into instance inner classes within the same enclosing class. The unusual difficulty of this transformation arises from the ability of local classes to access final local variables visible at the point when the class was declared. The most common case is presented in row 8. For each such local class `L`, an additional inner class `LEnv` is defined that stores the enclosing local environment of `L`. `LEnv` contains a field for each variable in the environment. Accesses to the local environment of `L` are encoded as accesses to fields of `LEnv`.

Finally, anonymous classes are first converted into local classes, which are in turn converted into inner classes (the last row).

Motivation for elaboration phase. Initially, K-Java was developed without an elaboration phase. For example expression types were computed along their value each time they were executed. Similarly, other computations that are now done in the elaboration phase were performed on the fly during execution. This approach worked up to a point, and produced a monolithic semantics which was more compact than the present one. However, when we reached the point to define the semantics of the conditional expression `_?_:_` and of the local classes, it eventually became clear that we had to compute the static types upfront. As described in Section 3.2, it is impossible to compute the type of a conditional expression by evaluating its operands dynamically. Also, local classes had to be discovered and preprocessed prior to execution, in order to cover all the corner cases. For this we needed a pre-execution phase capable to dig into code blocks.

Folding phase. The last and simplest phase of preprocessing, Folding, combines the information stored in class cells back into one big AST, acting as an interface to the dynamic semantics. Since all the transformations performed by the static semantics maintain the validity of the program, the resulting preprocessed AST represents a valid Java program as well. This phase is technically unnecessary for the dynamic semantics alone, because Unfolding phase in dynamic K-Java (Figure 5) is the exact mirror of Folding, distributing the AST back into the same cells that were used to produce it. Folding and Unfolding phases were required to complete the division between the static and the dynamic parts of K-Java.

The preprocessed AST. The output of the static semantics is the preprocessed AST. This AST has a valuable property: it corresponds to a valid Java program, equivalent to the initial one. More precisely, for every Java program P_1 there is a program P_2 such that the preprocessed AST of P_1 is equal to the AST of P_2. We do not prove this property, we only state it, supported by the observation that every transformation performed by the static semantics preserves program equivalence.

Maintaining this equivalence imposed some challenges. First, we were forced to use only transformations that preserve program validity. All the transformations presented in Figure 6 are also performed by the Java compiler (javac), with the difference that javac produces bytecode. In contrast, not all transformations performed by javac were accessible to static K-Java. For example, javac flattens inner classes into top-level classes. We could not do the same because we would lose method access modes, and access modes have to be preserved (Section 3.2). Also, we could not use any auxiliary constructs in the preprocessed AST. Despite the challenges, we believe producing a preprocessed AST with stated properties was worthwhile, since it enables our static semantics to be usable in conjunction with tools outside the $\mathbb{K}$ ecosystem.

4.2 Exceptions

We choose to illustrate the first rules from K-Java through the dynamic semantics of `throw` and `try/catch` (Figure 7). These rules are relatively self-contained and require just the cell k. By default, rules without a cell context (as are all the rules in Figure 7) automatically apply to the current computation task (the first task in k, which holds the current computation). Virtually all the rules in a language definition in $\mathbb{K}$ match the first task of k either explicitly or (like here) implicitly.

The first rule rewrites a `try/catch` into the body of `try`, S, followed by catch clauses wrapped into the auxiliary construct `catchBlocks`. The binary associative construct "$\curvearrowright$" (spelled "followed by") is builtin to $\mathbb{K}$ and stands for computation task sequentialization. The constructor's role is to separate individual tasks in the current computation. Now rules for other statements will apply over the content of S until it is either entirely consumed or a `throw` statement becomes the first task.

In the first case, if S was consumed, it means the execution of `try` block completed normally. Then the term `catchBlocks` reaches the top of computation and is discarded by the second rule. In the second rule, "_" is an anonymous variable, which can match anything. Here it can match one or more `catch` clauses. The term `catchBlocks` is rewritten into "$\cdot$", the unit (or empty) computation. In words, this rule says that if `catchBlocks` reaches the top of computation then discard it. Now the entire `try/catch` is consumed and the execution continues with the next statement.

In case S throws an exception, a `throw` statement will eventually reach the top of the computation. Figure 7 continues with the syntax definition of `throw`, as it has a semantic role. Before `throw` could be evaluated, the expression inside `throw` has to be evaluated to a value. $\mathbb{K}$ has a compact notation to achieve this — the $[strict]$ annotation, meaning: bring the non-terminals of the annotated syntax to the top of computation, and plug them back when they are evaluated to a final value. Specifically for the statement `throw`, this annotation is compiled into one heating and one cooling rule, as shown in the inner box figure. The heating rule brings the expression to the top of computation for evaluation. The notation "$\square$" (spelled "hole") is a placeholder marking the context from which the expression was heated. The rule should only apply if the expression is not a final result yet (here a typed value); this restriction is encoded in the side condition. The cooling rule brings the produced typed value back to its original context. The typed value is $V :: ThrowT$ — an auxiliary notation of K-Java. The left side of "::" is the value, while the right side is the static type. $\mathbb{K}$ also has more general forms of $[strict]$, allowing to control which arguments should be heated and the evaluation order.

Now the actual rules for `throw` are ready to apply. Rule 3 (THROW-MATCH) matches when the type of the exception carried by `throw` is compatible with the type accepted by the first `catch` clause, so the exception is caught. The compatibility of the exception type and the `catch` clause's type is verified by the expres-

sion subtype$(ThrowT, CatchT)$ in the side condition. In this case, both the `throw` statement and the `catchBlocks` construct are rewritten into a sequence of three statements: a declaration of the `catch` parameter X of type $CatchT$ as a local variable ($CatchT\,X$;), an initialization of X with the exception value typed with the type expected by X ($X = V :: CatchT$;), and the body of `catch` — $CatchS$. All three statements are wrapped inside a block `{ }`, to confine the variable `X` to its expected scope.

Rule 4 matches when the two types are not compatible. In this case the first `catch` clause is dissolved, bringing the next one (if any) to the top of the list. Rule 5 matches when `throw` cannot interact with the next statement (the side condition). Statements interacting with `throw` are the non-empty `catchBlocks` and `finally`. Thus, if the next computation item after `throw`, *KI:KItem*, is anything except the two, including the empty `catchBlocks` produced eventually by the fourth rule, that statement will be dissolved. The statement `throw` will remain the top of computation until it reaches a matching `catch` or remains the only rule inside the k cell. The notation *KI:KItem* means that variable *KI* should match only one item in the computation sequence separated by $\curvearrowright$. If we wrote just *KI*, then the entire remaining computation would be matched. If `throw` remains the only task in the computation, another rule will match that will ensure exception propagation outside the current method.

Loop statements with `break`/`continue` are defined similarly.

One may wonder why not desugar a `try` with multiple `catch` clauses into multiple `try` with one `catch` each. We actually did so initially, but it turned out to be incorrect. Indeed, the code

```
try { throw new A(); }
catch(A a) { throw new B(); }
catch(B b) { ... }
```

cannot be desugared into

```
try {
  try { throw new A(); }
  catch(A a) { throw new B(); }
}
catch(B b) { ... }
```

because in the second case the exception `B` would get caught by `catch(B b)`, and this is incorrect according to JLS. This (counter) example is now included in our test suite.

4.3 Method Invocation

The central rule of method invocation is the first rule in Figure 8. This is actually the last rule to be applied, after a long chain of computations that involves computing the qualifier, the arguments, loading the required information from various cells of configuration and deciding the actual method to be invoked, based on the access mode, as was mentioned in Section 3.2. When the actual invocation of the method is ready to be performed, the top of computation cell contains a term of the form $Qual$.methodRef$(Sig, DecC)(Args)$. This term is a Java qualified method invocation expression augmented with K-Java auxiliary syntax. Here $Qual$ is the qualifier—the object or class upon which the method is invoked. The term methodRef$(Sig, DecC)$ is a reference to a method with signature Sig declared in the class $DecC$. The variable $RestK$ denotes the remaining computation/continuation in cell k. This rule performs the following operations:

- The rest of computation ($RestK$) and the current content of methodContext are saved to a new entry (or frame) on top of the cell stack. This data will be used to restore the current computation context by the rules for `return`.
- The new method context is initialized with:
  - An empty local variable environment (env).
  - The declaring class of the invoked method (crntClass).
  - The current object location, computed by auxiliary function getOid$(Qual)$ as:
    - The location of the qualifier, if qualifier is an object.
    - No location, if qualifier is a class (for static methods).
- Current computation is rewritten into a sequence of four terms:
  - Static initialization of the new declaring class.
  - Initialization of the parameters.
  - The method body.
  - A return statement with no arguments. This will be reached only if the execution of the method body terminates without reaching another `return` or `throw` statement.

RULE TRY

$$\frac{\texttt{try}\ S\ \ CatchList}{S \curvearrowright \texttt{catchBlocks}\ (CatchList)}$$

RULE CATCHBLOCKS-DISSOLVE

$$\frac{\texttt{catchBlocks}\ (_)}{\cdot}$$

SYNTAX $Stmt$ ::= `throw` Exp ; [*strict*]
compiled into:

RULE THROW-HEAT

$$\frac{\cdot}{E} \curvearrowright \texttt{throw}\ \frac{E}{\square}\ ;$$

REQUIRES $\neg_{Bool}$ isTypedValue (E)

RULE THROW-COOL

$$\frac{V :: ThrowT}{\cdot} \curvearrowright \texttt{throw}\ \frac{\square}{V :: ThrowT}\ ;$$

RULE THROW-MATCH

$$\frac{\texttt{throw}\ V :: ThrowT\ ; \curvearrowright \texttt{catchBlocks}\ (\texttt{catch}\ (CatchT\ \ X)\{CatchS\}\ _)}{\{CatchT\ \ X\ ; \curvearrowright X = V :: CatchT\ ; \curvearrowright CatchS\}}$$

REQUIRES subtype $(ThrowT, CatchT)$

RULE THROW-NOT-MATCH

$$\texttt{throw}\ V :: ThrowT\ ; \curvearrowright \texttt{catchBlocks}\ \left(\frac{\texttt{catch}\ (CatchT\ \ X)\{CatchS\}}{\cdot}\ _\right)$$

REQUIRES $\neg_{Bool}$ subtype $(ThrowT, CatchT)$

RULE THROW-PROPAGATION

$$\texttt{throw}\ _ :: _\ ; \curvearrowright \frac{KI : KItem}{\cdot}$$

REQUIRES $\neg_{Bool}$ interactsWithThrow (KI)

Figure 7. Rules for `try/catch` and `throw`

The auxiliary function `staticInit()` triggers static initialization of the qualifying class, in case this class was not initialized yet. Repeated calls of this function have no effect.

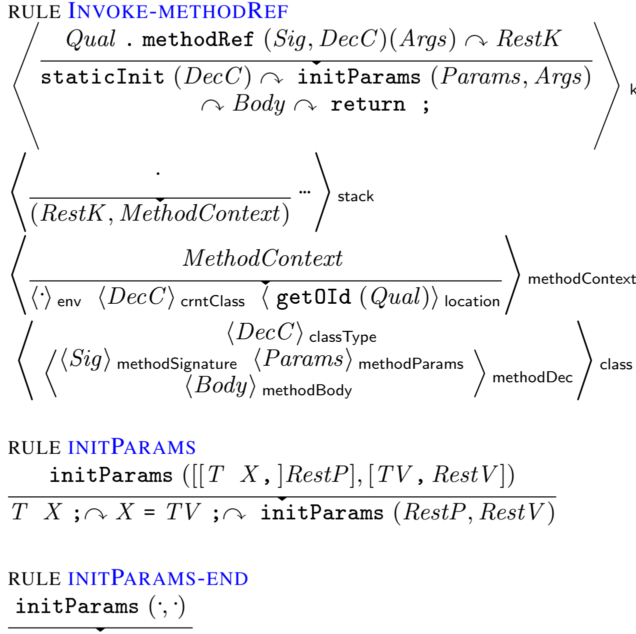

Figure 8. Final rules for method invocation

The function `initParams()` is defined by rules 2 and 3 in Figure 8. It takes as arguments two lists—the list of parameter declarations and the list of argument values. The first rule processes the first parameter declaration in the list. The parameter with name X of type T is rewritten into a local variable declaration $T\ X$; followed by an assignment expression that initializes X with the argument value. The third term in the RHS of the rewrite is `initParams` with the processed parameter and argument removed. Here we use `[ ]` as $\mathbb{K}$ brackets, to disambiguate terms grouping.

When all parameters are processed, both arguments of `initParams` become empty lists. In this case the third rule from the figure applies, dissolving the `initParams` term.

The complete semantics of method invocation, as well as the semantics of `new`, are discussed in the companion report [12].

4.4 Multithreading and Synchronization

K-Java has basic support for threads. First is the class Thread with methods `start()`, `join()` and `interrupt()`. We also support thread synchronization through the `synchronized` statement (JLS §14.19) and the `synchronized` modifier for methods (JLS §8.4.3.6). For more advanced usage we support threading-related methods from class `Object`: `wait()`, `notify()`, `notifyAll()`.

Below we discuss the key rules for methods `wait()` and `notify()`. Both methods have to be called from a `synchronized` block that holds the monitor (lock) of the target object. When `wait()` is called, the thread releases the lock on target object and blocks until another thread calls `notify()` on the same object. When `notify()` is called, the waiting thread does not wake up immediately, but only after the notifying thread exited the `synchronized` block. When waking up, the thread re-acquires the lock on the target object. When there are multiple threads waiting on the same monitor object, `notify()` will non-deterministically wake up one of them. Finally, if either `wait()` or `notify()` is called in a state where the current thread does not hold the appropriate monitor, an exception is thrown.

The rules are presented in Figure 9 (see Figure 4 for the configuration cells). The first rule [object-wait] corresponds to the invocation of `wait()`. Here the term `objectRef`$(Oid, _)$ :: _ represents

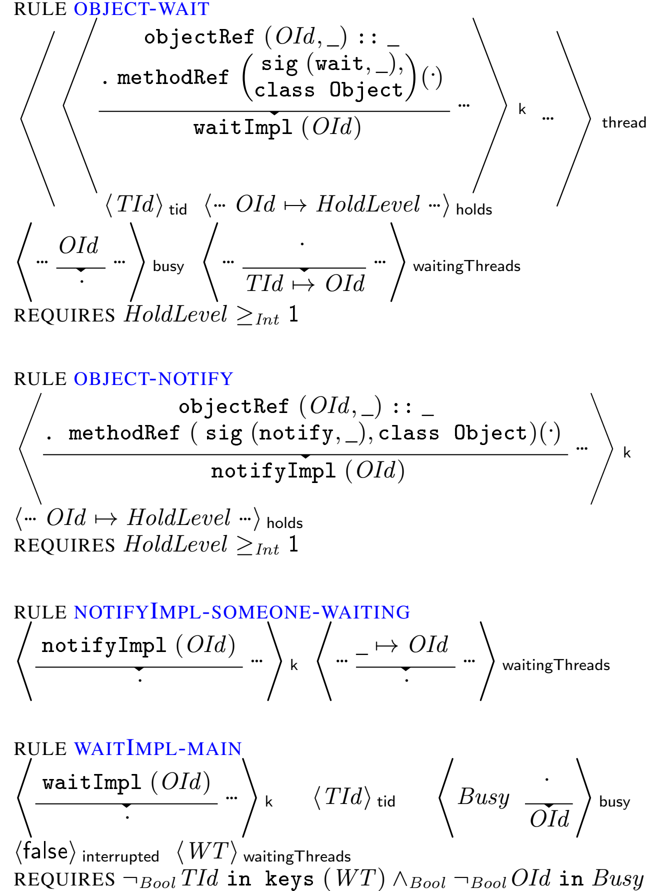

Figure 9. Rules for `Object.wait()` and `Object.notify()`

a typed object reference—the qualifier of the method call. The only part that is of interest to us from this term is Oid—the object identifier in the store, that is also used as synchronization key. The next term, `methodRef(sig(wait,_), class Object)`, represents a method reference to the method with signature `sig(wait,_)` (name `wait` and irrelevant arguments), declared in the class `Object`. The expression above the horizontal line (LHS of the rewrite) is a method call expression, for the given object reference, the given method reference and no arguments. This rule replaces the method call expression in cell k with `waitImpl()`—an auxiliary function used later, to exit from the waiting state. The id of the current thread (`TId`) has to be registered in the set in waitingThreads. The cell holds attached to each thread stores the number of times the current thread acquired the lock on each object. Here we use it to make sure that the current thread acquired the lock at least once (see the side condition). Another cell matched here is busy. This one, in contrast with holds is a global cell, unique for the whole configuration. It stores the set of objects that are currently used as synchronization monitors—arguments of blocks `synchronized`. When an object enters the waiting state, it has to release its ownership to the monitor; this is done by deleting `OL` from the set.

The second rule [object-notify] is the starting rule for `notify()`. The side condition ensures that the current thread holds the monitor on the target object. The actual logic of `notify()` is delegated to `notifyImpl()`. The construct `notifyImpl()` requires two rules for two cases—the case when there is at least one thread waiting on the target object, and the case when there is none.

Rule [notifyImpl-someone-waiting] is the rule for the first case. If there is a thread waiting on the current object, then the object

identifier `OId` will be present among the map values of waitingThreads. By deleting the whole entry associated to that value we enable the waiting thread to proceed. If there is no thread waiting for this object then term `notifyImpl()` is simply consumed.

At this stage, after a call to `notify()`, the rule for `waitImpl()` could match inside another thread (rule [waitImpl-main]). The rule checks in its side conditions that the current thread identifier, `TId`, is not among the waiting threads anymore. It also checks that the target object, represented by `OId`, is not busy. This is required because the thread exiting from waiting state has to reacquire the monitor on the target object. Finally, the rule has to make sure that the thread was not interrupted while it was waiting. Otherwise another rule will match and will throw the appropriate exception.

We only presented the key rules above, corresponding to the most common execution scenario. The corner cases, such as illegal calls that should result in various exceptions, are not included but are nevertheless fully supported and tested.

5. 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.

5.1 State Space Exploration

The next simplest way to use K-Java besides execution is state space exploration. When a program is run with the $\mathbb{K}$ runner and option `--search`, the tool outputs all of the possible executions for the program, exposing any possible non-deterministic behavior. This capability was successfully used in the semantics of C [20] to expose the non-determinism of expression evaluation. While single-threaded Java is deterministic, threads bring non-determinism. By running a multi-threaded Java program in search mode, we can produce all its interleaving behaviors.

Additionally, the option `--pattern` allows us to filter the search results according to a pattern. This feature may be effectively used, for example, to detect deadlocks. In K-Java, the cell $\langle\rangle_{\text{thread}}$ is dissolved when the corresponding thread finishes its execution. Consequently, we can detect successful execution paths by using the pattern $\langle\cdot\rangle_{\text{threads}}$. The pattern will match when there are no threads remaining. Conversely, the pattern $\langle_\rangle_{\text{thread}}$ would match the final states where at least one thread did not finish its execution, e.g. a deadlock. We successfully used this approach to detect the deadlock in the Dining Philosophers problem.

5.2 LTL Model-Checking

While state space search might be used to test some programs, $\mathbb{K}$ offers a more powerful capability for exploring non-deterministic behavior. Specifically, $\mathbb{K}$ provides linear temporal logic (LTL) model-checking capabilities through its Maude [16] backend. In this section we show how K-Java can be seamlessly used to verify LTL properties of multi-threaded applications.

Consider the program in Figure 10 (a modified version of [8]). The program contains a blocking queue – a classical producer-consumer example with one producer and one consumer thread. The inner class BlockingQueue contains two methods - `put()` and `get()`. The methods are synchronized, designed to be called from a multi-threaded environment. When `put()` is called on a full queue, the calling thread has to wait until some other thread dequeues an element. Similarly, the consumer thread calling `get()` has to wait when the queue is empty. The producer (the main thread) calls `put()` four times, while the consumer (anonymous thread inside `main()`) calls `get()` the same four times. Aside from demonstrating multi-threading capabilities, this small program illustrates many other features of Java: expressions with side effects,

```
public class QueueTest {
  static class BlockingQueue {
    int capacity = 2;
    int[] array = new int[capacity];
    int head=0, tail=0;
    synchronized void put(int element)
        throws InterruptedException {
      if (tail-head == capacity) {
        wait();                           // 1
      }
      array[tail++ % capacity] = element;
      System.out.print(0);
      notify();                           // 2
    }
    synchronized int get()
        throws InterruptedException {
      if (tail-head == 0) {
        wait();                           // 3
      }
      int element
          = array[head++ % capacity];
      System.out.print(1);
      notify();                           // 4
      return element;
    }
  }

  public static void main(String[] args)
      throws Exception {
    final BlockingQueue queue
        = new BlockingQueue();
    new Thread() {
      public void run() {
        for(int i=0; i<4; i++) {
          try {
            queue.get();
          }
          catch(InterruptedException e) {}
    } } }.start();
    for(int i=0; i<4; i++) {
      queue.put(i);
    }
  }
}
```

Figure 10. A two-threaded blocking queue

exception handling, arrays, static and instance methods, inner and anonymous classes. Running state space exploration for the example above correctly produces all eight expected interleavings of the output text (0 represents a call to `put()`, 1 – a call to `get()`):

```
00101011 00101101 00110011 00110101
01001011 01001101 01010011 01010101
```

The implementation of `BlockingQueue` contains a deliberate, subtle problem. The `wait()` call is within an `if` inside both `get()` and `put()`, thus is executed at most once. This is actually a correct behavior if we have just one producer and one consumer,[2] but leads to problems when the number of threads is at least three. Above, the only way a thread waiting on line labelled "3" could be awakened is from a call to `notify()` from method `put()`, line labeled "2". Since at the end of the method `put()` we have at least one element in the queue, method `get()` can safely extract an element.

In a scenario with one producer and two consumers the thread waiting on "3" could be awakened by a call to `notify()` from ei-

[2] We do not consider spurious wakeups here.

ther "2" or "4". If the thread executing `get()` was awakened by "2" (another `get()`), that other `get()` could have actually extracted the last element from the queue, thus rendering the queue empty. Unaware of the queue state, the freshly-awakened thread will execute the body of `get()` and will extract a non-existing element from the queue. To eliminate this possibility, we need to replace the `if` statements with `while` above both at "1" and "3". A programmer might make such a subtle mistake, when first designing the queue for a two-threaded usage, and then extending the context to more threads. It is difficult to expose this bug through traditional testing. We executed the incorrect three-threaded program a dozen times with the JVM; it never picked the buggy path.

Next, we would like to expose the problem above with LTL. In this implementation of queue `tail` is incremented each time an element is added to the queue, and `head` is incremented each time an element is extracted. They are never decremented. Since it is not possible to extract more elements than are added, a basic invariant of the queue is `head <= tail`. This is the exact invariant that is violated by the example above in the three-threaded scenario.

The LTL model checking capability of the $\mathbb{K}$ framework allows us to find the bug by model checking the following LTL formula:

```
□(this instanceof BlockingQueue
  ⟹ this.head <= this.tail)
```

The property was correctly proven true for the example in Figure 10, but was proven false for the three-threaded version of the same program. When we corrected the queue implementation, the model checker proved the formula correct. The version of the program in Figure 10 took 15 seconds to model check, produced 72 transition states and roughly two million rewrites.

Similar formal program analysis capabilities were demonstrated within the semantics of C [20] and PHP [22].

6. 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.

6.1 The Quest for a Test Suite

Virtually all recent executable language semantics projects [10, 20, 22, 30] used an external test suite for validation. Naturally, we tried to do the same. The official test suite for Java, targeting both the language and the class library, is Java Compatibility Kit (JCK) [2] from Oracle. JCK is *not* publicly available. Instead Oracle offers free access for non-profit organisations willing to implement the whole JDK, i.e., *both* the language *and* the class library. After a laborious application, Oracle rejected our request.

We also explored unofficial test suites. Jtreg, part of OpenJDK [7], is a regression test suite for the class library, but not for the language. Another test suite is Mauve [6], containing tests for classes and for the compiler, but not for the runtime. Tests targeting the compiler test its capability to distinguish between correct and incorrect Java programs, and to output the appropriate error message. Unfortunately, all these tests were unsuitable for our purpose.

There were actually two external test suites that we were able to use. One was the set of examples from ACM-Java [35] that we presented in Section 2. Another one was the list of examples from the book Java Precisely [34]. We used 44 out of 58 tests from ASM-Java and 63 out of 114 examples from Java Precisely. The programs that we did not use either illustrated compiler errors, or were not complete Java programs, or used classes that we do not support.

These two sets of examples, while useful, were far from enough. Their purpose was to illustrate Java, not to exercise every single corner case of the language. With no luxury of an available comprehensive test suite, we had no choice but to develop our own.

6.2 Test Development Methodology

When writing our tests we followed the Test Driven Development (TDD). The main principle of TDD is to write tests before implementing the actual feature under test. It was advantageous to use TDD for K-Java for two reasons. First, K-Java has a complete and final specification—JLS. Consequently, our tests are not expected to change as a result of changes in the specification. Second, tests for K-Java are self-contained Java programs, they do not depend on any part of the system under test (K-Java) to be written.

For every test, we compared the output produced by K-Java with the output produced by JDK. When developing a new feature, we followed the following steps. First we tried to cover all corner cases of the feature under test in isolation. For example, the first non-terminal of the `for` statement might be a list of variable initializers. In such case we will include in our tests statements with zero, one or more initializers. Second, we would define the new feature in the simplest way possible to pass all the tests. Sometimes, after inspecting the implementation, we would identify some corner cases that were not captured by the tests. We would add additional tests for such cases. Thus our methodology was a combination of white-box and black-box testing. In addition to the steps above, we wrote tests for each *combination* of language features whenever we thought that the two features may possibly unexpectedly interact.

We followed the development of Java starting from low-level features such as literals, expressions, statements, towards the higher level features. The order of development is approximately reflected by the order in which tests were written, which can be found in K-Java public repository [1]. We aimed at testing every detail specified in JLS; e.g., to test the precise order of execution of subexpressions inside an expression, we intentionally used subexpressions with side-effects and verified the correct order of evaluation by observing the correct order in which the side effects occurred.

Some features depended on other features to be properly tested. For example, in order to test the precise static type of various expressions we used method overloading. For example, if an expression `e` has to return type `A`, but a plausible erroneous semantics could also produce `B`, we tested the correct choice by calling `f(e)`, where `f` was an overloaded method that could accept an argument of type `A` or `B`. In all such cases we were careful to postpone the exhaustive testing of the freshly developed feature, and to write the tests later once all prerequisites are available.

Eventually, we produced a test suite consisting of 840 tests.

Multithreading. To test multi-threading we used state space exploration. We designed a test suite comprised of 28 programs explicitly aiming at covering all the behaviours of all the supported multi-threaded language constructs. For each program, we first produce all the possible solutions using $\mathbb{K}$. Then we compare the number of solutions with the expected, manually determined one.

Later changes, testing ASM-Java and JavaFAN. When we first tried to execute ASM-Java and JavaFAN over our test suite, the majority of the tests unexpectedly failed. It was because some very basic feature, used in most of the tests, was not supported. For example ASM-Java did not support addition between a string and a boolean. JavaFAN did not support escape characters, and we were not able to use `"\n"` in our tests for this reason. To overcome this problem, we inspected our tests and eliminated the most common causes of failure, when they were not the actual feature under test. This way we were able to produce the results reported in Section 2.

7. Conclusion and Discussion

We have discussed 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.

The skeptical reader may argue that there is no such thing as a 'complete' semantics of a large language like Java, because it is always possible to miss a feature or, worse, an interaction of features. While this is true in principle, we mention that completeness of the semantics was our major objective from the inception of this project, and that we have very carefully considered all possible interactions of features that were explicitly discussed in the JSL or that we could think of. Since there is no other attempt to completely formalize Java that we are aware of in order to formally compare with K-Java, due to all the above we believe that it is fair to claim that K-Java is the first complete formal semantics of Java.

K-Java can serve as a platform to experiment with Java extensions. Not only can one define such extensions rigorously, but one also gets a reference implementation for experimentation.

We plan to extend K-Java with semantics for the the latest versions of Java, starting with Java 5. Most of Java 5 features can be compiled into JVM 1.4 bytecode by using javac option `-target jsr14`: generics, varargs, for-each loop and autoboxing [3]. These features can be similarly defined in the static K-Java. Support in the dynamic K-Java would be required for enumerations and annotations. The best-known feature of Java 8 is lambda expressions. Lambdas could be desugared into anonymous classes implementing a functional interface. JDK uses a different strategy, yet the designers of Java suggest desugaring as an option [4].

We also plan to define dynamic class loading. It has already been defined in K-Python [26], so we see no impediments in adapting it for K-Java. This would allow support for a much wider set of classes from JDK, consequently, for more real life programs.

Finally, using $\mathbb{K}$'s support for Reachability Logic [17], it should be possible to deductively verify Java applications. Deductive verification requires more user involvement that model checking and would be a separate work on its own.

Acknowledgments

Partly supported by NSF grants CCF-1218605 and CCF-1318191.

References

[1] K-Java repository. URL `https://github.com/kframework/java-semantics/tree/popl-artifact-evaluation`.

[2] Gaining access to the JCK. URL `http://openjdk.java.net/groups/conformance/JckAccess/`.

[3] Using Java 5 features in earlier JDKs. URL `http://www.ibm.com/developerworks/java/library/j-jtp02277/index.html`.

[4] Translation of lambda expressions. URL `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html`.

[5] MatchC project. URL `http://fsl.cs.illinois.edu/index.php/Special:MatchCOnline`.

[6] Mauve. URL `https://sourceware.org/mauve/`.

[7] Openjdk project. URL `http://openjdk.java.net/`.

[8] A simple scenario using wait() and notify() in Java. URL `http://stackoverflow.com/questions/2536692/a-simple-scenario-using-wait-and-notify-in-java`.

[9] L. Bettini, F. Damiani, I. Schaefer, and F. Strocco. Traitrecordj: A programming language with traits and records. *Sci. Comput. Program.*, 78(5):521–541, May 2013.

[10] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In *POPL'14*, pages 87–100. ACM, 2014.

[11] D. Bogdănaş. Label-based programming language semantics in K framework with SDF. In *SYNASC '12*, pages 160–167. IEEE, 2012.

[12] D. Bogdănaş. K-Java: runtime semantics for method invocation and object instantiation. Technical report, University of Illinois, 2014. URL `http://hdl.handle.net/2142/55512`.

[13] E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of C#. *Theor. Comput. Sci.*, 336 (2-3):235–284, May 2005.

[14] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *OOPSLA '98*, pages 183–200. ACM, 1998.

[15] M. Bravenboer, R. Vermaas, R. de Groot, and E. Dolstra. Java Front. URL `http://strategoxt.org/Stratego/JavaFront`.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.6), 2011. URL `http://maude.cs.uiuc.edu/maude2-manual/`.

[17] A. Ştefănescu, c. Ciobâcă, R. Mereuţă, B. M. Moore, T. F. Şerbănuţă, and G. Roşu. All-path reachability logic. In *RTA-TLCA'14*, LNCS. Springer, 2014.

[18] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker and its implementation. In *SPIN'03*, pages 230–234. Springer-Verlag, 2003.

[19] C. Ellison. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois, July 2012.

[20] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *POPL '12*, pages 533–544. ACM, 2012.

[21] A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. In *CAV'04*, volume 3114 of *LNCS*, pages 501 – 505, 2004.

[22] D. Filaretti and S. Maffeis. An executable formal semantics of PHP. In *ECOOP'14*, pages 567–592, 2014.

[23] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269. Springer-Verlag, 1999.

[24] FSF. C language testsuites: "C-torture" version 4.4.2, 2010. URL `http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`.

[25] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition*. Addison-Wesley, 2nd edition, 2000.

[26] D. Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, July 2013.

[27] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a java-like language. In *ESOP'07*, volume 4421 of *LNCS*, pages 347–362. Springer Berlin Heidelberg, 2007.

[28] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF—reference manual—. *SIGPLAN Not.*, 24 (11):43–75, Nov. 1989.

[29] K. Kreft and A. Langer. Understanding the closures debate, 2008. URL `http://www.javaworld.com/article/2077869/scripting-jvm-languages/understanding-the-closures-debate.html`.

[30] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full monty. In *OOPSLA '13*, pages 217–232. ACM, 2013.

[31] G. Roşu and A. Ştefănescu. Checking reachability using matching logic. In *OOPSLA'12*, pages 555–574. ACM, 2012.

[32] G. Roşu, A. Ştefănescu, c. Ciobâcă, and B. M. Moore. One-path reachability logic. In *LICS'13*, pages 358–367. IEEE, June 2013.

[33] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *JLAP*, 79(6):397–434, 2010.

[34] P. Sestoft. *Java Precisely*. MIT Press, 2002.

[35] R. F. Stärk, E. Börger, and J. Schmid. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.