K-Java: A Complete Semantics of Java

Denis Bogdănaş ¹  Grigore Roşu ²

¹University of Iaşi

²University of Illinois at Urbana-Champaign

January 16, 2015
K-Java: First Complete Semantics of Java

- Java 1.4, for now
- Defined in the K framework
  - [http://kframework.org](http://kframework.org)
  - Open source: [https://github.com/kframework](https://github.com/kframework)
- K-Java publicly available for download
  - Open source: [https://github.com/kframework/java-semantics](https://github.com/kframework/java-semantics)
## Comparison with Existing Java Semantics – part 1

<table>
<thead>
<tr>
<th>Feature</th>
<th>AJ</th>
<th>JF</th>
<th>KJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic integer, boolean, String literals</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Other literals</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Overflow, distinction between integer types</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Prefix ++i - -i, also += -= . . . ,</td>
<td></td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>Bit-related operators:</td>
<td>&amp; ^ &gt;&gt; &lt;&lt; &gt;&gt;&gt;</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Other integer operators</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>String + &lt;other types&gt;</td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Reference operators</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Basic statements</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Switch</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Try-catch-finally</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Break</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Continue</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Array basics</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Array-related exceptions</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Array length</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Array length</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Array polymorphism</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Array initializers</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Array default values</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
</tbody>
</table>
## Comparison with Existing Java Semantics – part 2

<table>
<thead>
<tr>
<th>Feature</th>
<th>AJ</th>
<th>JF</th>
<th>KJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic OOP - classes, inheritance, polymorphism</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Method overloading – distinct number of arguments</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Method overloading without argument conversion</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Method overloading with argument conversion</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Method access modes</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Instance field initializers</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Chained constructor calls via this() and super()</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Keyword super</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Interfaces</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Interface fields</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Static methods and fields</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Accessing unqualified static fields</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Static initialization</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Static initialization trigger</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Packages</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Shadowing</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Hiding</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Instance initialization blocks</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Static inner classes</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Instance inner classes</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Local &amp; anonymous classes</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
</tbody>
</table>

AJ = ASM-Java, 2001, Stärk et al.

JF = JavaFAN, 2006, Farzan et al.

KJ = our work
Contributions

- K-Java, the first complete semantics of Java 1.4
- Comprehensive test suite of 840 tests
- Completeness assessment of ASM-Java and JavaFAN using tests
- Application – LTL model-checking of multithreaded programs
Java formalization challenges – static typing

- Child expr type alters parent type:
  - $1 + (\text{long})2$ vs $1 + 2$
  - Former has type long, latter has has type int
- Integer value range:
  - $1000$ vs $(\text{byte})1000$
  - Static type affects expression value
- Method overloading:
  - $f(0)$ vs $f((\text{long})0)$
  - Static type affects method choice
Java formalization challenges – method overloading

- Non-trivial feature interactions
  - For example, between access modes and overloading
  - We cannot eliminate access modes at static phase

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

    String test() {
        return f((byte)0); } // f(int) version
}

class B {
    String test() {
        return new A().f((byte)0); } // f(long) version
}
```
Java formalization challenges – conditional operator

- The type of `_?_:` depends on types of operands 2 and 3
- Yet only one of them is executed
- We cannot defer computation of static types until execution (although this is tempting)
- Types should be computed prior to execution

```java
int i = Integer.MAX_VALUE;
long l;

true ? i++ : l++  // = Integer.MAX_VALUE + 1

true ? i++ : i++  // = Integer.MIN_VALUE
```
K-Java structure

- **Parser**
  - program
  - program AST
  - Collect Class Names
  - Process Compilation Units
  - Process Class Declarations
  - Process Class Members
  - Elaboration
  - Folding
  - preprocessed AST (valid Java program)
  - Unfolding
  - Execution

**Static semantics**

- Process Compilation Units
- Process Class Declarations
- Process Class Members
- Elaboration
- Folding

- Collect Class Names

**Dynamic semantics**

- Execution
Static semantics

- Fully qualified class names:

  Object $\Rightarrow$ java.lang.Object
Static semantics

- Fully qualified class names
- Expression types:

\[ 1 + 2L \Rightarrow (\text{long})((\text{int})1 + (\text{long})2) \]
Static semantics

- Fully qualified class names
- Expression types
- Signature of method calls:

```java
void f(long a);
f(1);  \Rightarrow  f((long)(int)1);
```
Static semantics

- Fully qualified class names
- Expression types
- Signature of method calls
- Discrimination between local vars and static/instance fields:

```java
class A{
    int v;
    static int s;
}
```

⇒

```java
v (int)((A) this).v
s (int)A.s
```
Static semantics

- Fully qualified class names
- Expression types
- Signature of method calls
- Discrimination between local vars and static/instance fields
- Other means to uniform classes:

```java
class A {
    A(){}
}
class B {
    static int a=1;
    static{a++;}
}
⇒
class A {
    A(){ super(); }
}
class B {
    static int a;
    static{a=1;a++;}
}
```
Static semantics

- Fully qualified class names
- Expression types
- Signature of method calls
- Discrimination between local vars and static/instance fields
- Other means to uniform classes
- Local classes \(\Rightarrow\) inner classes:

```java
class 0 {
    void m() {
        final int a=1, b=2;
        class L {
            int f(){ return a+b; } \Rightarrow
        }
        new L().f();
    }
}
```
Static semantics

- Fully qualified class names
- Expression types
- Signature of method calls
- Discrimination between local vars and static/instance fields
- Other means to uniform classes
- Local classes $\Rightarrow$ inner classes

Yet the preprocessed program is a *valid* Java program. So our static semantics is encapsulated as a behavior-preserving program specialization tool. Can be independently useful in other Java analysis projects, too.
Dynamic semantics

- We only discuss one feature: exceptions. See website for rest.
- We illustrate it by stepwise “executing” the statement below.
- \( K \) features introduced on-the-fly, as needed.

```java
try {
    try {
        throw new B();
        print("unreachable");
    } catch (A a) {}  
} catch (B b) {
    print(b);
}
```
Dynamic Semantics of Exceptions – 5 Modular Rules

rule try

\[
\text{try } S \text{ CatchList} \\
S \rightsquigarrow \text{catchBlocks (CatchList)}
\]

rule catchBlocks-dissolve

\[
\text{catchBlocks (---)}
\]

rule throw-match

\[
\text{throw } V :: \text{ ThrowT} ; \rightsquigarrow \text{catchBlocks (catch (CatchT X)\{CatchS\} ---)} \\
\{\text{CatchT X} ; \rightsquigarrow X = V :: \text{CatchT} ; \rightsquigarrow \text{CatchS}\}
\]

requires subtype (ThrowT, CatchT)

rule throw-not-match

\[
\text{throw } V :: \text{ ThrowT} ; \rightsquigarrow \text{catchBlocks (catch (CatchT X)\{CatchS\} ---)}
\]

requires \(\neg\) Bool subtype (ThrowT, CatchT)

rule throw-propagation

\[
\text{throw } \_ :: \_ ; \rightsquigarrow \text{KI:KItem}
\]

requires \(\neg\) Bool interactsWithThrow (KI)
Dynamic semantics – exception handling

rule try

\[\text{try } S \text{ CatchList} \quad \text{S ↦ catchBlocks}(\text{CatchList})\]

try {
  try {
    \text{throw new B();}
    \text{print("unreachable");}
  } \text{ catch (A a) {}}
} catch (B b) {
  \text{print(b);}
}
Dynamic semantics – exception handling

rule try

\[
\text{try } \underline{S} \ [\text{CatchList}] \\
\underline{S} \rightsquigarrow \text{catchBlocks(CatchList)}
\]

try {
    throw new B();
    print("unreachable");
} catch (A a) {}
\rightsquigarrow \text{catchBlocks(}
    catch (B b) {
        print(b);
    }
\)
Dynamic semantics – exception handling

rule throw-propagation

\[
\text{throw } \text{obj :: } B \\
\leadsto \text{print("unreachable");} \\
\leadsto \text{catchBlocks( catch (A a) {} )} \\
\leadsto \text{catchBlocks(} \\
\quad \text{catch (B b) {}} \\
\quad \quad \text{print(b);} \\
\quad \}
\]

requires \( \neg \text{Bool} \).interactsWithThrow(KI)
Dynamic semantics – exception handling

**rule throw-not-match**

\[
\text{throw } \rightarrow \text{:: } \text{ThrowT } \; ; \\
\neg \text{ catchBlocks } \left( \text{catch } (\text{CatchT} \; X)\{\text{CatchS}\} \; \rightarrow \right)
\]

**requires** \(\neg \text{Bool subtype}(\text{ThrowT, CatchT})\)

\[
\text{throw obj } \rightarrow \text{:: } B \\
\neg \text{ catchBlocks( catch (A a) {} )} \\
\neg \text{ catchBlocks(} \\
\quad \text{catch (B b) } \{ \\
\quad \quad \text{print}(b); \\
\quad \} \\
\quad ) \\
\Rightarrow \quad \text{[throw-propagation]}
\]

\[
\text{throw obj } \rightarrow \text{:: } B \\
\neg \text{ catchBlocks(} \\
\quad \text{catch (B b) } \{ \\
\quad \quad \text{print}(b); \\
\quad \} \\
\quad )
\]
rule throw-match

\[
\text{throw } V :: \text{ThrowT} ; \\
\leadsto \text{catchBlocks (catch (} \text{CatchT} X \text{)}\{\text{CatchS}\} \leftarrow ) \\
\{\text{CatchT} X ; \leadsto X = V :: \text{CatchT} ; \leftarrow \text{CatchS}\}
\]

requires subtype (ThrowT, CatchT)

\[
\text{throw } \text{obj} :: \text{B} \\
\leadsto \text{catchBlocks (catch (} \text{B} b \text{)} \{ \\
\text{print}(b); \\
\} \\
) \\
\Rightarrow \\
\{\text{B} b; \leftarrow b = \text{obj} :: \text{B}; \leftarrow \text{print}(b);\}
\]
## K-Java Statistics

<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>Dynamic</th>
<th>Common</th>
<th>Lib</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source lines</td>
<td>3112</td>
<td>2035</td>
<td>539</td>
<td>497</td>
<td>6183</td>
</tr>
<tr>
<td>Comment lines</td>
<td>857</td>
<td>1189</td>
<td>383</td>
<td>98</td>
<td>2527</td>
</tr>
<tr>
<td>Size (KB)</td>
<td>168</td>
<td>139</td>
<td>52</td>
<td>22</td>
<td>381</td>
</tr>
<tr>
<td>Cells</td>
<td>31</td>
<td>36</td>
<td>28</td>
<td>–</td>
<td>95</td>
</tr>
<tr>
<td>Rules</td>
<td>497</td>
<td>347</td>
<td>183</td>
<td>47</td>
<td>1074</td>
</tr>
<tr>
<td>Auxilliary functions</td>
<td>111</td>
<td>83</td>
<td>79</td>
<td>9</td>
<td>282</td>
</tr>
</tbody>
</table>
Testing/Validating K-Java

- Semantics defined in $\mathbb{K}$ are executable
  - Can be tested as compilers or interpreters are
- Surprisingly, there was no conformance testsuite for Java (until now)
- Attempted to obtain JDK testsuite from Oracle
  - After submitting application, Oracle refused to provide it
- Developed our own testsuite, in tandem with the semantics
  - 840 programs thoroughly testing all Java features, and combinations
  - Side contribution; useful for other Java analysis projects, too
- K-Java is the only Java formal semantics that passes all these tests
Applications

- Besides executability, K framework provides several other modules which can be used with any semantics:
  - State space exploration (useful to analyze non-determinism, concurrency)
  - LTL model checking
  - Symbolic execution
  - Deductive program verification

- These yield corresponding formal analysis tools for Java
  - Correct by construction

- We only discuss LTL model checking
LTL model checking

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();
        array[tail++ % capacity] = element;
        System.out.print("put-" + element + " ");
        notify();
    }
    synchronized int get() throws InterruptedException {
        if (tail-head == 0) wait();
        int element = array[head++ % capacity];
        System.out.print("get-" + element + " ");
        notify();
        return element;
    }
}

- put() waits when the queue is full
- get() waits when the queue is empty
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();
        array[tail++ % capacity] = element;
        System.out.print("put-" + element + " ");
        notify();
    }

    synchronized int get() throws InterruptedException {
        if (tail-head == 0) wait();
        int element = array[head++ % capacity];
        System.out.print("get-" + element + " ");
        notify();
        return element;
    }
}

A deliberate bug: both put() and get() waits almost once
Yet correct for 1 producer and 1 consumer
LTL model checking

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();
        array[\textcolor{green}{tail++} % capacity] = element;
        System.out.print("put-" + element + "");
        notify();
    }
    synchronized int get() throws InterruptedException {
        if (tail-head == 0) wait();
        int element = array[\textcolor{green}{head++} % capacity];
        System.out.print("get-" + element + "");
        notify();
        return element;
    }
}

Both head and tail are only incremented
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();
        array[tail++ % capacity] = element;
        System.out.print("put-" + element + " ");
        notify();
    }

    synchronized int get() throws InterruptedException {
        if (tail-head == 0) wait();
        int element = array[head++ % capacity];
        System.out.print("get-" + element + " ");
        notify();
        return element;
    }
}

Both head and tail are only incremented

LTL formula: □(this instanceof BlockingQueue ⇒ this.head <= this.tail)
LTL model checking

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();
        array[tail++] % capacity = element;
        System.out.print("put-" + element + " ");
        notify();
    }

    synchronized int get() throws InterruptedException {
        if (tail-head == 0) wait();
        int element = array[head++ % capacity];
        System.out.print("get-" + element + " ");
        notify();
        return element;
    }
}

LTL formula: $\square (\text{this instance of BlockingQueue} \implies \text{this.head} \leq \text{this.tail})$

Output (1 producer, 2 consumers, put & get 1, 2, 3, 4):
    put-0 put-1 get-0 get-1 put-2 get-2 get-1 put-3
LTL model checking

```java
class BlockingQueue {
    int capacity = 2;
    int[] array = new int[capacity];
    int head=0, tail=0;

    synchronized void put(int element) throws InterruptedException {
        while (tail-head == capacity) wait();
        array[tail++ % capacity] = element;
        System.out.print("put-");
        notify();
    }

    synchronized int get() throws InterruptedException {
        while (tail-head == 0) wait();
        int element = array[head++ % capacity];
        System.out.print("get-");
        notify();
        return element;
    }
}
```

- Corrected program
- Output: true
Conclusion and Future Work

- **K-Java**, the first complete semantics of Java 1.4
  - More than 1000 semantic rules in $K$
  - Static semantics as Java-to-Java translation, useful beyond K-Java

- Comprehensive test suite of 840 tests
  - Useful beyond K-Java

- Completeness assessment of ASM-Java and JavaFAN using tests

- Application – LTL model-checking of multithreaded programs

- Java 8

- More tools (waiting for $K$ framework team to develop them)