K-Java: A Complete Semantics of Java

Denis Bogdănaș [1] Grigore Roșu [2]

[1] University of Iași

[2] University of Illinois at Urbana-Champaign

January 16, 2015

K-Java: First Complete Semantics of Java

- Java 1.4, for now
- Defined in the $\mathbb{K}$ framework
  - `http://kframework.org`
  - Open source: `https://github.com/kframework`
- K-Java publicly available for download
  - Open source: `https://github.com/kframework/java-semantics`

Comparison with Existing Java Semantics – part 1

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	●	○	●

AJ = ASM-Java, 2001, Stärk etal.

JF = JavaFAN, 2006, Farzan etal.

KJ = our work

Comparison with Existing Java Semantics – part 2

Feature	AJ	JF	KJ
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	○	○	●

AJ = ASM-Java, 2001, Stärk etal.

JF = JavaFAN, 2006, Farzan etal.

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 + (long)2` vs `1 + 2`
  - Former has type `long`, latter has has type `int`
- Integer value range:
  - `1000` vs `(byte)1000`
  - Statc type affects expression value
- Method overloading:
  - `f(0)` vs `f((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

```
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

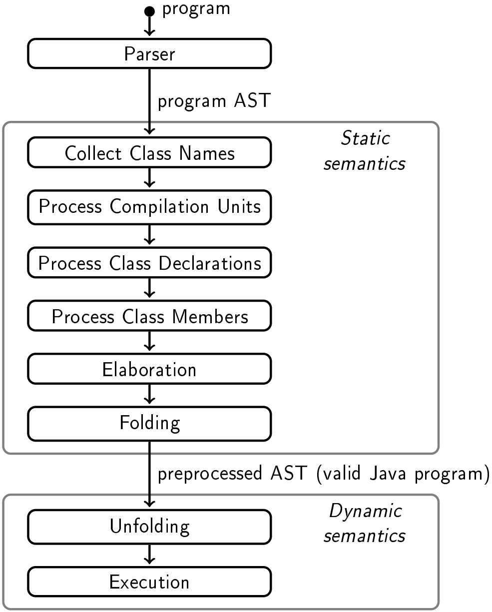
- The type of _?_:_ depends on types of operands 2 and 3
- Yet only one of them is executed
- We cannot defer computation of static types until execution (although this is tempting)
- Types should be computed prior to execution

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

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

true ? i++ : i++ // = Integer.MIN_VALUE
```

K-Java structure



Static semantics

- Fully qualified class names:

```
Object    ⇒    java.lang.Object
```

Static semantics

- Fully qualified class names
- Expression types:

```
1 + 2L    ⇒    (long)((int)1 + (long)2)
```

Static semantics

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

```
void f(long a);
                    ⇒
f(1);                        f((long)(int)1);
```

Static semantics

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

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

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:

```
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 ⇒ inner classes:

```
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
```

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 ⇒ 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
- $\mathbb{K}$ features introduced on-the-fly, as needed

```
try {
  try {
    throw new B();
    print("unreachable");
  } catch (A a) {}
}
catch (B b) {
 print(b);
}
```

Dynamic Semantics of Exceptions – 5 Modular Rules

rule try

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

rule catchBlocks-dissolve

$$\frac{\texttt{catchBlocks}\ (\text{—})}{\cdot}$$

rule throw-match

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

requires $\texttt{subtype}\ (\mathbf{ThrowT}, \mathbf{CatchT})$

rule throw-not-match

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

requires $\neg_{Bool}\ \texttt{subtype}\ (\mathbf{ThrowT}, \mathbf{CatchT})$

rule throw-propagation

$$\texttt{throw}\ \text{—}\ \texttt{::}\ \text{—}\ \texttt{;}\ \curvearrowright\ \frac{\mathbf{KI}{:}\mathit{KItem}}{\cdot}$$

requires $\neg_{Bool}\ \texttt{interactsWithThrow}\ (\mathbf{KI})$

Dynamic semantics – exception handling

rule try

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

```
try {
  try {
    throw new B();
    print("unreachable");
  } catch (A a) {}
}
catch (B b) {
 print(b);
}
```

Dynamic semantics – exception handling

rule try

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

```
try {
  throw new B();
  print("unreachable");
} catch (A a) {}
 ↷ catchBlocks(
      catch (B b) {
        print(b);
      }
    )
```

Dynamic semantics – exception handling

rule throw-propagation

$$\texttt{throw}\ \boxed{\_ :: \_}\ ;\ \curvearrowright\ \underline{\boxed{\mathsf{KI}{:}\mathit{KItem}}}$$

requires $\neg_{Bool}$ `interactsWithThrow` (KI)

```
throw obj ::  B
 ↷ print("unreachable");
 ↷ catchBlocks( catch (A a) {} )
 ↷ catchBlocks(
     catch (B b) {
       print(b);
     }
   )
```

Dynamic semantics – exception handling

rule throw-not-match

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

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

```
throw obj :: B
 ↷ catchBlocks( catch (A a) {} )
 ↷ catchBlocks(
      catch (B b) {
        print(b);
      }
    )
```

⇒ [throw-propagation]

```
throw obj :: B
 ↷ catchBlocks(
      catch (B b) {
        print(b);
      }
    )
```

Dynamic semantics – exception handling

rule throw-match

$$\frac{\text{throw}\ \mathbf{V} :: \boxed{\mathbf{ThrowT}}\ ;\ \curvearrowright \text{catchBlocks}\ (\text{catch}\ (\boxed{\mathbf{CatchT}}\ \mathbf{X})\{\mathbf{CatchS}\}\ —)}{\{\mathbf{CatchT}\ \ \mathbf{X}\ ;\ \curvearrowright\ \mathbf{X} = \mathbf{V} :: \mathbf{CatchT}\ ;\ \curvearrowright\ \mathbf{CatchS}\}}$$

requires subtype (**ThrowT**, **CatchT**)

```
throw obj :: B
 ↷ catchBlocks(
      catch (B b) {
        print(b);
      }
    )
```

⇒

```
{B b; ↷ b = obj :: B; ↷ print(b);}
```

K-Java Statistics

	Static	Dynamic	Common	Lib	Total
Source lines	3112	2035	539	497	6183
Comment lines	857	1189	383	98	2527
Size (KB)	168	139	52	22	381
Cells	31	36	28	–	95
Rules	497	347	183	47	**1074**
Auxilliary functions	111	83	79	9	282

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, $\mathbb{K}$ framework provides several other modules which can 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

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;
  }
}
```

- 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[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 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;
  }
}
```

Both `head` and `tail` are only incremented

LTL formula: $\square$(`this instanceof BlockingQueue` $\implies$ `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: $\Box$(`this instanceof BlockingQueue` $\implies$ `this.head <= 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

```
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-" + element + " ");
    notify();
  }
  synchronized int get() throws InterruptedException {
    while (tail-head == 0) wait();
    int element = array[head++ % capacity];
    System.out.print("get-" + element + " ");
    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 $\mathbb{K}$
  - 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 $\mathbb{K}$ framework team to develop them)