An Overview of Java Semantics Implementation in K Framework

Denis Bogdănaș

University Alexandru Ioan Cuza of Iași

January 20, 2013
Introduction

Project goal

- a complete and executable semantics for Java 1.4
- incremental extension to Java 5
- using the semantics for program analysis

Motivation

- many semantics for Java have been developed
- neither of them are complete (to author’s knowledge)
- tools for program reasoning are not based on executable semantics

K Framework (one of Monday’s tutorials)

- a formalism for defining operational semantics of programming languages
- produces executable semantics
- supports program reasoning
Implementation statistics

- Defined features - all of Java 1.4 except inner classes
- 500 hand-crafted test programs
- 3200 lines of code
- 600 rewrite rules
- 45 configuration cells
Control intensive statements - overview

- From 15 Java statements, 11 interact with the execution flow
  - Some of them define code blocks with special meaning - method call, try, labeled statement, loops
  - Others alter the execution flow and interact with those blocks - return, throw, break, continue

- Naive approach - a separate stack for each block-defining statement
  - K semantics using this approach - C, SIMPLE, KOOL
  - Is it good for Java?
Control intensive statements - method call, return

```java
void main() {
    int a = 1;
    a = f();
    print("main");
}
int f() {
    int b = 2;
    return b;
}
```

![Diagram of control flow]

Denis Bogdănaş (UAIC)  Java semantics  January 20, 2013  6 / 14
Control intensive statements - method call, return

```java
void main() {
    int a = 1;
    a = f();
    print("main");
}

int f() {
    int b = 2;
    return b;
}
```

```java
return b

env
b ↦ Loc2

fstack
( a= □ print("main"), [a ↦ Loc1] )
```
Control intensive statements - adding try/catch, throw

```java
int a = 1;
try {
    int b = 2;
    throw new Exception();
} catch (Exception e) {
    print("catch");
}
print("end");
```

```
int b=2.jsx
↶
throw new Exception()
↷
popx
↷
print("end")
```

```
env
(a ↦ L1  b ↦ L2)
fstack
FS
xstack
( print("end"), [a ↦ L1],
[Exception e], [print("catch")], FS )
```
Control intensive statements - adding try/catch, throw

```java
int a = 1;
try {
    int b = 2;
    throw new Exception();
} catch (Exception e) {
    print("catch");
}
print("end");
```

```java
int a = 1;
try {
    int b = 2;
    throw new Exception();
} catch (Exception e) {
    print("catch");
}
print("end");
```
Control intensive statements - adding try/finally

```java
void main() {
f(); print("main");
}
void f() {
    try {
        try {
            return;
        } catch (Exception e) { print("catch"); }
    } finally { print("finally"); }
    print("end");
}
```

Problem: Which one to pick?
Control intensive statements - the unified stack

```java
void main() {
    f();
    print("main");
}
void f() {
    try {
        try {
            return;
        } catch (Exception e) { print("catch"); }
    } finally { print("finally"); }
    print("end");
}
```

Each method call, try/catch, try/finally creates a stack slice of a different type. Nesting order is preserved.
Control intensive statements - example execution

```java
void main() {
    f();
    print("main");
}
void f() {
    try {
        try {
            return;
        } catch (Exception e) {
            print("catch");
        }
    } finally {
        print("finally");
    }
    print("end");
}
```

Return cannot interact with try/catch, thus `tryCatch` slice is silently deleted.
Control intensive statements - example execution

```java
void main() {
    f();
    print("main");
}

void f() {
    try {
        try {
            return;
        } catch (Exception e) { print("catch"); }
    } finally { print("finally"); }
    print("end");
}
```

Finally block is executed, but return statement still remains:
Control intensive statements - example execution

```java
void main() {
    f();
    print("main");
}

void f() {
    try {
        try {
            return;
        } catch (Exception e) { print("catch"); }
    } finally { print("finally"); }
    print("end");
}
```

Finally block is executed, but return statement still remains:

```
print("finally") return
```

```
( tryFinally, •, [•Map], print("finally"), • )
( method, print("main"), [•Map], •, • )
```
Control intensive statements - example execution

```java
void main() {
f(); print("main");
}
void f() {
    try {
        try {
            return;
        } catch (Exception e) { print("catch"); }
    } catch (Exception e) { print("catch"); }
    finally { print("finally"); }
    print("end");
}
```

Return statement is consumed and the environment of main method is restored:
Control intensive statements - example execution

```java
void main() {
f();
print("main");
}

void f() {
    try {
        try {
            return;
        } catch (Exception e) { print("catch"); }
    } catch (Exception e) { print("catch"); }
} finally { print("finally"); }
print("end");
}
```

Return statement is consumed and the environment of main method is restored:
### Control intensive statements - summary

<table>
<thead>
<tr>
<th>Stack slice</th>
<th>Created by</th>
<th>Consumed by</th>
<th>Silently deleted by</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>method, constructor call</td>
<td>return</td>
<td>throw</td>
</tr>
<tr>
<td>tryCatch</td>
<td>try / catch</td>
<td>matching throw</td>
<td>non-matching throw, return, break, continue</td>
</tr>
<tr>
<td>tryFinally</td>
<td>try / finally</td>
<td>return, throw, break, continue</td>
<td>-</td>
</tr>
<tr>
<td>loop</td>
<td>while, do-while, for</td>
<td>break, continue</td>
<td>throw, return</td>
</tr>
</tbody>
</table>
Conclusions

- The unified stack allows complex interactions between control-intensive statements
- New statements can be defined incrementally without altering existing rules
- Uniform stack slices structure allows reusable semantics rules
- K Framework is suitable for defining Java

Thank you for your attention!