An Overview of Java Semantics Implementation in $\mathbb{K}$ 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

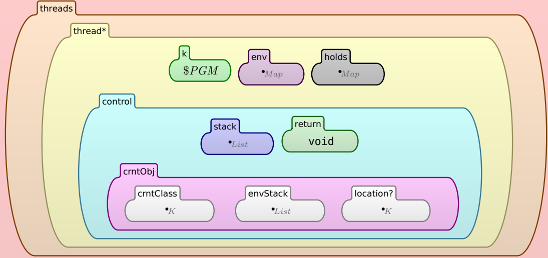
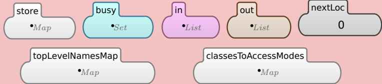
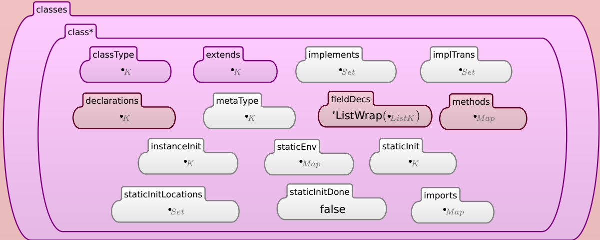
- many semantics for Java have been developed
- neither of them are complete (to author's knowledge)
- tools for program reasoning are not based on executable semantics

$\mathbb{K}$ 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


T
threads
thread*
k
$PGM
env
•Map
holds
•Map
control
stack
•List
return
void
crntObj
crntClass
•K
envStack
•List
location?
•K

store
•Map
busy
•Set
in
•List
out
•List
nextLoc
0
topLevelNamesMap
•Map
classesToAccessModes
•Map

classes
class*
classType
•K
extends
•K
implements
•Set
implTrans
•Set
declarations
•K
metaType
•K
fieldDecs
'ListWrap(•ListK)
methods
•Map
instanceInit
•K
staticEnv
•Map
staticInit
•K
staticInitLocations
•Set
staticInitDone
false
imports
•Map

mainClass
$MainClass
cuPackage
•K
cuClass
•K
compUnitImports
•Map
classLiteralsMap
•Map
computationPhase
ProcessTopLevelNamesPhase
programBackup
$PGM
dissolveEmptyK
true
dissolveClasses
true

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 approace - a separate stack for each block-defining statement
- $\mathbb{K}$ semantics using this approach - C, SIMPLE, KOOL
- Is it good for Java?

Control intensive statements - method call, return

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

k

```
return b
```

env

b $\mapsto$ *Loc2*

fstack

(a= $\square$ $\curvearrowright$`print("main")`, [a $\mapsto$ *Loc1*])

Control intensive statements - method call, return

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

k: return b

env: b ↦ *Loc2*

fstack: (a= □ ↷print("main"), [a ↦ *Loc1*])

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

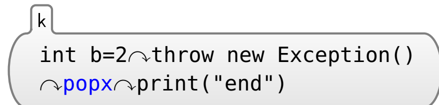
k: a=2↷print("main")

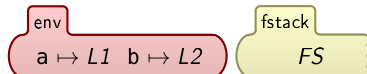
env: a ↦ *Loc1*

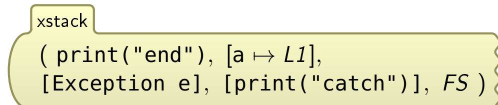
fstack:

Control intensive statements - adding try/catch, throw

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

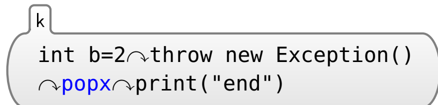


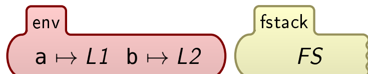


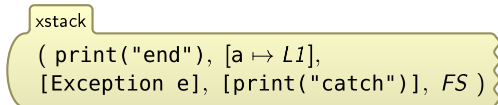


Control intensive statements - adding try/catch, throw

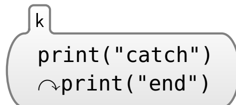
```
int a = 1;
try {
  int b = 2;
  throw new Exception();
} catch (Exception e) {
  print("catch");
}
print("end");
```

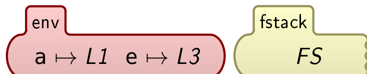


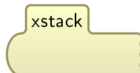





```
int a = 1;
try {
  int b = 2;
  throw new Exception();
} catch (Exception e) {
  print("catch");
}
print("end");
```

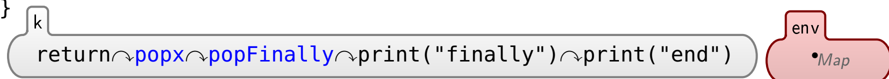


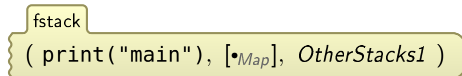


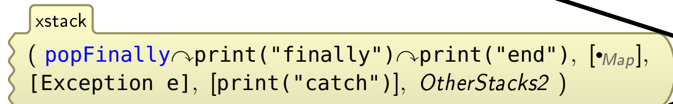


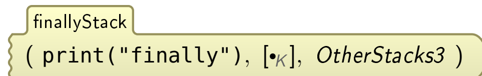
Control intensive statements - adding try/finally

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





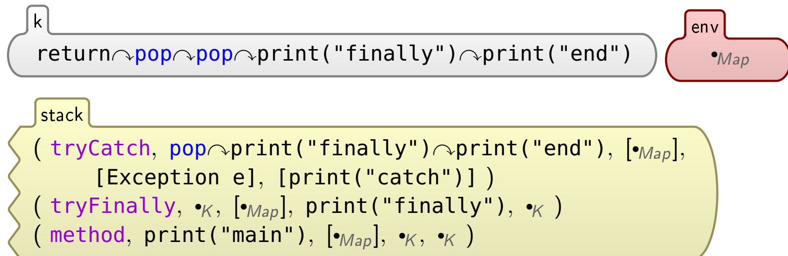




Problem: Which one to pick?

Control intensive statements - the unified stack

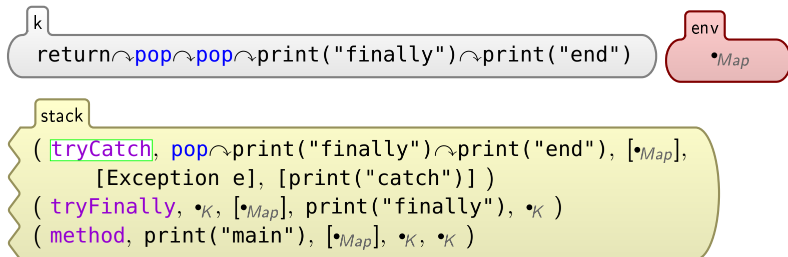
```
void main() {f(); print("main");}
void f() {
  try {
    try {
      return;
    } catch (Exception e) { print("catch"); }
  } finally { print("finally"); }
  print("end");
}
```



Each method call, try/catch, try/finally creates a stack slice of a different type. Nesting order is preserved.

Control intensive statements - example execution

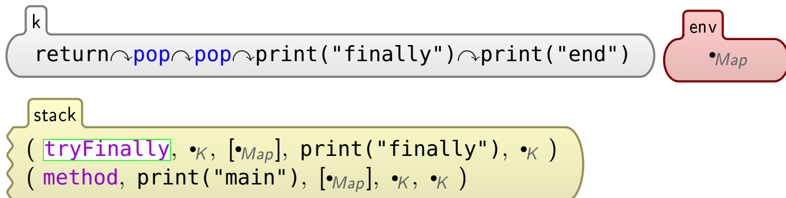
```
void main() {f(); print("main");}
void f() {
  try {
    try {
      return;
    } catch (Exception e) { print("catch"); }
  } finally { print("finally"); }
  print("end");
}
```



Return cannot interact with try/catch, thus tryCatch slice is silently deleted.

Control intensive statements - example execution

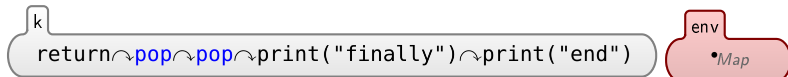
```
void main() {f(); print("main");}
void f() {
  try {
    try {
      return;
    } catch (Exception e) { print("catch"); }
  } finally { print("finally"); }
  print("end");
}
```

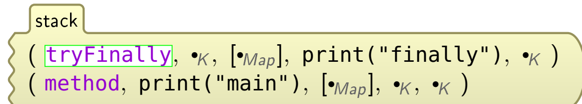


Finally block is executed, but return statement still remains:

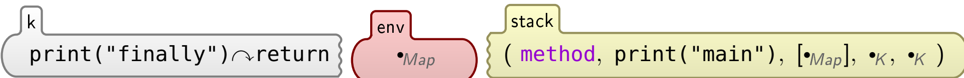
Control intensive statements - example execution

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



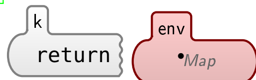


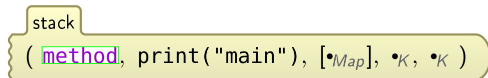
Finally block is executed, but return statement still remains:



Control intensive statements - example execution

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

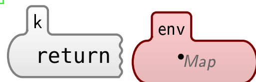


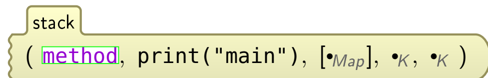


Return statement is consumed and the environment of main method is restored:

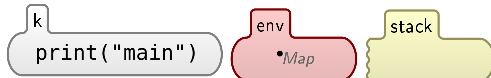
Control intensive statements - example execution

```
void main() {f(); print("main");}
void f() {
  try {
    try {
      return;
    } 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

Stack slice	Created by	Consumed by	Silently deleted by
method	method, constructor call	return	throw
tryCatch	try / catch	matching throw	non-matching throw, return, break, continue
tryFinally	try / finally	return, throw, break, continue	-
loop	while, do-while, for	break, continue	throw, return

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!