Formal Methods for Java

Lecture 3: Operational Semantics (Part 2)

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

November 2, 2011

Operational Semantics for Java

Idea: define transition system for Java

Definition (Transition System)

A transition system (TS) is a structure $TS = (Q, Act, \rightarrow)$, where

- Q is a set of states,
- Act a set of actions,
- $\rightarrow \subseteq Q \times Act \times Q$ the transition relation.

- Q reflects the current dynamic state (heap and local variables).
- Act is the executed code.
- Idea from: D. v. Oheimb, T. Nipkow, Machine-checking the Java specification: Proving type-safety, 1999

State of a Java Program

The state of a Java program gives valuations local and global (heap) variables.

- $Q = \mathit{Heap} \times \mathit{Local}$
- $\mathit{Heap} = \mathit{Address} \to \mathit{Class} \times \mathrm{seq}\ \mathit{Value}$
- $\mathit{Local} = \mathit{Identifier} \to \mathit{Value}$
- $\mathit{Value} = \mathbb{Z}, \mathit{Address} \subseteq \mathbb{Z}$

A state is denoted as $(\mathit{heap}, \mathit{lcl})$, where $\mathit{heap} : \mathit{Heap}$ and $\mathit{lcl} : \mathit{Local}$.

Actions of a Java Program

An action of a Java Program is either

- the evaluation of an expression e to a value v, denoted as $e \triangleright v$, or
- a Java statement, or
- a Java code block.

Note that expressions with side-effects can modify the current state

Rules for Java Expressions

axiom for evaluating local variables:

$$(heap, lcl) \xrightarrow{x \rhd lcl(x)} (heap, lcl)$$

axiom for evaluating constants:

$$(heap, lcl) \xrightarrow{c \rhd c} (heap, lcl)$$

rule for field access:

$$\frac{(heap, lcl) \xrightarrow{e \rhd v} (heap', lcl')}{(heap, lcl) \xrightarrow{e.fld \rhd heap'(v)(idx)} (heap', lcl')},$$ where idx is the index of the field fld in the object $heap'(v)$

Rules for Assignment Expressions

rule for assignment to local:

$$\frac{(heap, lcl) \xrightarrow{e \rhd v} (heap', lcl')}{(heap, lcl) \xrightarrow{x=e \rhd v} (heap', lcl' \oplus \{x \mapsto v\})}$$

rule for assignment to field:

$$\frac{\begin{array}{c}(heap_1, lcl_1) \xrightarrow{e_1 \rhd v_1} (heap_2, lcl_2) \\ (heap_2, lcl_2) \xrightarrow{e_2 \rhd v_2} (heap_3, lcl_3)\end{array}}{(heap_1, lcl_1) \xrightarrow{e_1.fld=e_2 \rhd v_2} (heap_4, lcl_3)},$$

where $heap_4 = heap_3 \oplus \{(v_1, idx) \mapsto v_2\}$ and idx is the index of the field fld in the object at $heap_3(v_1)$.

Rules for Java Statements

expression statement (assignment or method call):

$$\frac{(heap, lcl) \xrightarrow{e \triangleright v} (heap', lcl')}{(heap, lcl) \xrightarrow{e;} (heap', lcl')}$$

sequence of statements:

$$\frac{(heap_1, lcl_1) \xrightarrow{s_1} (heap_2, lcl_2) \quad (heap_2, lcl_2) \xrightarrow{s_2} (heap_3, lcl_3)}{(heap_1, lcl_1) \xrightarrow{s_1\ s_2} (heap_3, lcl_3)}$$

Rules for Java Statements

if statement:

$$\frac{(heap_1, lcl_1) \xrightarrow{e \rhd v} (heap_2, lcl_2) \quad (heap_2, lcl_2) \xrightarrow{s_1} (heap_3, lcl_3)}{(heap_1, lcl_1) \xrightarrow{\textbf{if}(e)\, s_1 \textbf{else} s_2} (heap_3, lcl_3)}, \text{where } v \neq 0$$

$$\frac{(heap_1, lcl_1) \xrightarrow{e \rhd v} (heap_2, lcl_2) \quad (heap_2, lcl_2) \xrightarrow{s_2} (heap_3, lcl_3)}{(heap_1, lcl_1) \xrightarrow{\textbf{if}(e)\, s_1 \textbf{else} s_2} (heap_3, lcl_3)}, \text{where } v = 0$$

while statement:

$$\frac{(heap_1, lcl_1) \xrightarrow{\textbf{if}(e)\{s\ \textbf{while}(e)\, s\}} (heap_2, lcl_2)}{(heap_1, lcl_1) \xrightarrow{\textbf{while}(e)\, s} (heap_2, lcl_2)}$$

Rule for Java Method Call

$$\frac{\begin{array}{c}(heap_1, lcl_1) \xrightarrow{e \triangleright v} (heap_2, lcl_2)\\ (heap_2, lcl_2) \xrightarrow{e_1 \triangleright v_1} (heap_3, lcl_3)\\ \vdots \\ (heap_{n+1}, lcl_{n+1}) \xrightarrow{e_n \triangleright v_n} (heap_{n+2}, lcl_{n+2})\\ (heap_{n+2}, mlcl) \xrightarrow{body} (heap_{n+3}, mlcl')\end{array}}{(heap_1, lcl_1) \xrightarrow{e.m(e_1,\ldots,e_n) \triangleright mlcl'(\backslash result)} (heap_{n+3}, lcl_{n+2})},$$

where $body$ is the body of the method m in the object $heap_{n+2}(v)$, and $mlcl = \{this \mapsto v, param_1 \mapsto v_1, \ldots, param_n \mapsto v_n\}$ where $param_1, \ldots, param_n$ are the names of the parameters of m

Creating Objects

Creating an Object is always combined with the call of a constructor:

$$\frac{\begin{array}{c} heap_1 = heap \cup \{na \mapsto (Type, \langle 0, \ldots, 0\rangle)\} \\ (heap_1, lcl) \xrightarrow{na.\texttt{<init>}(e_1,\ldots,e_n) \triangleright v} (heap', lcl') \end{array}}{(heap, lcl) \xrightarrow{\textbf{new}\ Type(e_1,\ldots,e_n) \triangleright na} (heap', lcl')}, \text{ where } na \notin \operatorname{dom} heap$$

Here `<init>` stands for the internal name of the constructor.

Exceptions

To handle exceptions a few changes are necessary:

- We extend the state by a flow component:
  $Q = \mathit{Flow} \times \mathit{Heap} \times \mathit{Local}$
- $\mathit{Flow} ::= \mathit{Norm} | \mathit{Ret} | \mathit{Exc}\langle\!\langle \mathit{Address} \rangle\!\rangle$

We use the identifiers $\mathit{flow} \in \mathit{Flow}$, $\mathit{heap} \in \mathit{Heap}$ and $\mathit{lcl} \in \mathit{Local}$ in the rules. Also $q \in Q$ stands for an arbitrary state.
The following axioms state that in an abnormal state statements are not executed:

$$(\mathit{flow}, \mathit{heap}, \mathit{lcl}) \xrightarrow{e \triangleright v} (\mathit{flow}, \mathit{heap}, \mathit{lcl}), \text{ where } \mathit{flow} \neq \mathit{Norm}$$

$$(\mathit{flow}, \mathit{heap}, \mathit{lcl}) \xrightarrow{s} (\mathit{flow}, \mathit{heap}, \mathit{lcl}), \text{ where } \mathit{flow} \neq \mathit{Norm}$$

Expressions With Exceptions

The previously defined rules are valid only if the left-hand-state is not an exception state.

$$\frac{(Norm, heap, lcl) \xrightarrow{e_1 \triangleright v_1} q \quad q \xrightarrow{e_2 \triangleright v_2} q'}{(Norm, heap, lcl) \xrightarrow{e_1 * e_2 \triangleright (v_1 \cdot v_2) \bmod 2^{32}} q'}$$

$$\frac{(Norm, heap, lcl) \xrightarrow{st_1} q \quad q \xrightarrow{st_2} q'}{(Norm, heap, lcl) \xrightarrow{st_1 ; st_2} q'}$$

$$\frac{(Norm, heap, lcl) \xrightarrow{e \triangleright v} q \quad q \xrightarrow{s_1} q'}{(Norm, heap, lcl) \xrightarrow{\mathbf{if}(e)\, s_1 \mathbf{else} s_2} q'}, \text{ where } v \neq 0$$

Note that exceptions are propagated using the axiom from the last slide.

$$(flow, heap, lcl) \xrightarrow{e \triangleright v} (flow, heap, lcl), \text{ where } flow \neq Norm$$

Throwing Exceptions

$$\frac{(Norm, heap, lcl) \xrightarrow{e \triangleright v} (Norm, heap', lcl')}{(Norm, heap, lcl) \xrightarrow{\textbf{throw}\ e;} (Exc(v), heap', lcl')}$$

What happens if in a field access the object is null?

$$\frac{\begin{array}{c}(Norm, heap, lcl) \xrightarrow{e \triangleright 0} q' \\ q' \xrightarrow{\textbf{throw new}\ NullPointerException()} q''\end{array}}{(Norm, heap, lcl) \xrightarrow{e.fld \triangleright v} q''}, \text{where } v \text{ is some arbitrary value}$$

Complete Rules for **throw**

$$\frac{(Norm, heap, lcl) \xrightarrow{e \triangleright v} (Norm, heap', lcl')}{(Norm, heap, lcl) \xrightarrow{\textbf{throw}\ e;} (Exc(v), heap', lcl')}, \text{ where } v \neq 0$$

$$\frac{\begin{array}{c}(Norm, heap, lcl) \xrightarrow{e \triangleright 0} q' \\ q' \xrightarrow{\textbf{throw new}\ NullPointerException()} q''\end{array}}{(Norm, heap, lcl) \xrightarrow{\textbf{throw}\ e;} q''}$$

$$\frac{(Norm, heap, lcl) \xrightarrow{e \triangleright v} (flow', heap', lcl')}{(Norm, heap, lcl) \xrightarrow{\textbf{throw}\ e;} (flow', heap', lcl')}, \text{ where } flow' \neq Norm$$

Catching Exceptions

Catching an exception:

$$\frac{(Norm, heap, lcl) \xrightarrow{s_1} (Exc(v), heap', lcl') \qquad (Norm, heap', lcl' \cup \{ex \mapsto v\}) \xrightarrow{s_2} q''}{(Norm, heap, lcl) \xrightarrow{\textbf{try}\ s_1\,\textbf{catch}(Type\ \ ex)s_2} q''}, \text{ where } v \text{ is an instance of } Type$$

No exception catched:

$$\frac{(Norm, h, l) \xrightarrow{s_1} (flow', h', l')}{(Norm, h, l) \xrightarrow{\textbf{try}\ s_1\,\textbf{catch}(Type\ \ ex)s_2} (flow', h', l')},$$

where flow' is not $Exc(v)$ or v is not an instance of *Type*

Return Statement

Return statement stores the value and signals the *Ret* in flow component:

$$\frac{(Norm, heap, lcl) \xrightarrow{e \triangleright v} (Norm, heap', lcl')}{(Norm, heap, lcl) \xrightarrow{\textbf{return}\ e} (Ret, heap', lcl' \oplus \{\backslash result \mapsto v\})}$$

But evaluating *e* can also throw exception:

$$\frac{(Norm, heap, lcl) \xrightarrow{e \triangleright v} (flow, heap', lcl')}{(Norm, heap, lcl) \xrightarrow{\textbf{return}\ e} (flow, heap', lcl')}, \text{ where } flow \neq Norm$$

Method Call (Normal Case)

$$\frac{\begin{array}{c}(Norm, h_1, l_1) \xrightarrow{e \triangleright v} q_2 \\ q_2 \xrightarrow{e_1 \triangleright v_1} q_3 \\ \vdots \\ q_{n+1} \xrightarrow{e_n \triangleright v_n} (f_{n+2}, h_{n+2}, l_{n+2}) \\ (f_{n+2}, h_{n+2}, ml) \xrightarrow{body} (Ret, h_{n+3}, ml')\end{array}}{(Norm, h_1, l_1) \xrightarrow{e.m(e_1, \dots, e_n) \triangleright ml'(\backslash result)} (Norm, heap_{n+3}, l_{n+2})},$$

where $param_1, \dots, param_n$ are the names of the parameters and $body$ is the body of the method m in the object $heap_{n+2}(v)$, and $ml = \{this \mapsto v, param_1 \mapsto v_1, \dots, param_n \mapsto v_n\}$

Method Call With Exception

$$\frac{\begin{array}{c}(Norm, h_1, l_1) \xrightarrow{e \triangleright v} q_2 \\ q_2 \xrightarrow{e_1 \triangleright v_1} q_3 \\ \vdots \\ q_{n+1} \xrightarrow{e_n \triangleright v_n} (f_{n+2}, h_{n+2}, l_{n+2}) \\ (f_{n+2}, h_{n+2}, ml) \xrightarrow{body} (Exc(v_e), h_{n+3}, ml')\end{array}}{(Norm, h_1, l_1) \xrightarrow{e.m(e_1,\dots,e_n) \triangleright ml'(\backslash result)} (Exc(v_e), heap_{n+3}, l_{n+2})},$$

where $param_1, \dots, param_n$ are the names of the parameters and *body* is the body of the method m in the object $heap_{n+2}(v)$, and $ml = \{this \mapsto v, param_1 \mapsto v_1, \dots, param_n \mapsto v_n\}$

Semantics of Specification

```
/*@ requires x >= 0;
  @ ensures \result <= Math.sqrt(x) && Math.sqrt(x) < \result + 1;
  @*/
public static int isqrt(int x) {
  body
}
```

Whenever the method is called with values that satisfy the requires-formula and the method terminates normally then the ensures-formula holds.
For all executions of the method,

$$(Norm, heap, lcl) \xrightarrow{body} (Ret, heap', lcl'),$$

if $lcl(x) >= 0$ then the formula

$$lcl'(\backslash result) <= Math.sqrt(lcl(x)) < lcl'(\backslash result) + 1$$

holds.

What About Exceptions?

```
/*@ requires true;
  @ ensures \result <= Math.sqrt(x) && Math.sqrt(x) < \result + 1;
  @ signals (IllegalArgumentException) x < 0;
  @ signals_only IllegalArgumentException;
  @*/
public static int isqrt(int x) {
  body
}
```

For all transitions

$$(Norm, heap, lcl) \xrightarrow{body} (Exc(v), heap', lcl')$$

where *lcl* satisfies the precondition and v is an Exception, v must be of type IllegalArgumentException. Furthermore, *lcl* must satisfy $x < 0$.
The code is still allowed to throw an Error like a OutOfMemoryError or a ClassNotFoundError.
If no signals_only clause is specified, JML assumes a sane default value: The method may throw only exceptions it declares with the `throws` keyword (in this case none).

Side-Effects

A method can change the heap in an unpredictable way.
The assignable clause restricts changes:

```
/*@ requires x >= 0;
  @ assignable \nothing;
  @ ensures \result <= Math.sqrt(x) && Math.sqrt(x) < \result + 1;
  @*/
public static int isqrt(int x) {
  body
}
```

For all executions of the method,

$$(Norm, heap, lcl) \xrightarrow{body} (Ret, heap', lcl'),$$

if $lcl(x) >= 0$ then the formula

$$lcl'(\backslash result) <= Math.sqrt(lcl(x)) < lcl'(\backslash result + 1)$$

holds and $heap = heap'$.

What is the meaning of a formula

A formula like $x >= 0$ is a Boolean Java expression. It can be evaluated with the operational semantics.

`x >= 0` holds in state $(\mathit{heap}, \mathit{lcl})$, iff

$$(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{\texttt{x >= 0} \triangleright v} (\mathit{fl}, \mathit{heap}, \mathit{lcl})$$

An assertion may not have side-effects.

For the ensures formula both the pre-state and the post-state are necessary to evaluate the formula.

Semantics of a Specification (formally)

A function satisfies the specification

$$\textbf{requires } e_1$$
$$\textbf{ensures } e_2$$

iff for all executions

$$(Norm, heap, lcl) \xrightarrow{body} (Ret, heap', lcl')$$

with $(Norm, heap, lcl) \xrightarrow{e_1 \triangleright v_1} q_1,\ v_1 \neq 0$, the post-condition holds, i. e., there exists v_2, q_2, such that

$$(Norm, heap', lcl') \xrightarrow{e_2 \triangleright v_2} q_2, \text{ where } v_2 \neq 0$$

However we need a new rule for evaluating $\backslash old$:

$$\frac{(Norm, heap, lcl) \xrightarrow{e \triangleright v} q}{(Norm, heap', lcl') \xrightarrow{\backslash old(e) \triangleright v} q},$$ where $heap, lcl$ is the state of the program before $body$ was executed

Method Parameters in Ensures-Clause

```
/*@ requires x >= 0;
  @ assignable \nothing;
  @ ensures \result <= Math.sqrt(x) && Math.sqrt(x) < \result + 1;
  @*/
public static int isqrt(int x) {
  x = 0;
  return 0;
}
```

Is this code a correct implementation of the specification?

No, because method parameters are always evaluated in the pre-state, so

```
\result <= Math.sqrt(x) && Math.sqrt(x) < \result + 1;
```

is the same as

```
\result <= Math.sqrt(\old(x)) && Math.sqrt(\old(x)) < \result + 1;
```

Side-Effects in Specification

In JML side-effects in specifications are forbidden:
If e is an expression in a specification and

$$(Norm, heap, lcl) \xrightarrow{e \triangleright v} (flow, heap', lcl')$$

then $heap = heap'$ and $lcl = lcl'$.
To be more precise, $heap \subseteq heap'$ since the new heap may contain new (unreachable) objects.
Also $flow \neq Norm$ is allowed. In that case the value of v may be unpredictable.
If the value of v is undefined the tools should assume the worst-case, i. e., report that code is buggy.