

Formal Methods for Java

Lecture 4: Semantics of JML

Jochen Hoenicke



Software Engineering
Albert-Ludwigs-University Freiburg

November 2, 2012

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 or expressions.
 - $q \xrightarrow{e \triangleright v} q'$ means that in state q the expression e is evaluated to v and the side-effects change the state to q' .
 - $q \xrightarrow{st} q'$ means that in state q the statement st is executable and changes the state to q' .

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 $heap, heap', lcl, lcl'$ if $lcl(x) \geq 0$
and $(Norm, heap, lcl) \xrightarrow{body} (Ret, heap', lcl')$,
then $lcl'(\backslash result) \leq Math.sqrt(lcl(x)) < lcl'(\backslash result) + 1$ holds.

Hoare Triples

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

The JML code above states **partial** correctness of the Hoare triple

$$\{x \geq 0\}$$

body

$$\{\text{\result} \leq \text{Math.sqrt}(x) < \text{\result} + 1\}$$

It also states **total** correctness, as we will see later.

Post condition and input parameters

Is the following implementation correct?

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

No, because JML always evaluates input parameters always in the pre-state!

For all $heap, heap', lcl, lcl'$ if $lcl(x) \geq 0$
and $(Norm, heap, lcl) \xrightarrow{body} (Ret, heap', lcl')$,
then $lcl'(\backslash result) \leq Math.sqrt(lcl(x)) < lcl'(\backslash result) + 1$ 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  
}
```

The `signals_only` specification denotes that 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`.

The `signals` specification denotes that in that case `lcl` must satisfy `x < 0`.

The code is still allowed to throw an `Error` like a `OutOfMemoryError` or a `ClassNotFoundError`.

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 $lcl(x) \geq 0$ then the formula

$$lcl'(\backslash result) \leq \text{Math.sqrt}(lcl(x)) < lcl'(\backslash result + 1)$$

holds and $heap \subseteq heap'$.

What is the meaning of a formula

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

$x \geq 0$ holds in state $(heap, lcl)$, iff

$$(Norm, heap, lcl) \xrightarrow{x \geq 0 \triangleright 1} (Norm, heap', lcl')$$

An assertion may not have side-effects; it may create new objects, though, i.e., $heap \subseteq heap'$ and $lcl = lcl'$.

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

requires e_1

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 $\backslash old$:

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

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 $heap \subseteq heap'$ and $lcl = lcl'$.

Here, $heap \subseteq heap'$ indicates that the new heap may contain new (unreachable) objects.

Also $flow \neq Norm$ is possible. In that case the expression is considered to be false.

A tool should warn the user if $flow \neq Norm$ is possible.

Exceptions in Specification

There were some discussions on exceptions in JML specifications.

- $next == null \ || \ next.prev == this$ is okay. It never throws a null-pointer exception.
- $next.prev == this \ || \ next == null$ is not equivalent. It is not valid if $next$ is null.

Specifications that can throw an exception should be avoided.

Lightweight vs. Heavyweight Specifications

A lightweight specification

```
/*@ requires P;  
   @ assignable X;  
   @ ensures Q;  
   @*/  
public void foo() throws IOException;
```

is an abbreviation for the heavyweight specification

```
/*@ public behavior  
   @ requires P;  
   @ diverges false;  
   @ assignable X;  
   @ ensures Q;  
   @ signals_only IOException  
   @*/  
public void foo() throws IOException;
```

With the `behavior`-keyword there are no default values for `diverges`, `signals_only`, and `assignable`.

Making Exceptions Explicit

```
/*@ public normal_behavior
   @ requires x >= 0;
   @ assignable \nothing;
   @ ensures \result <= Math.sqrt(x) && Math.sqrt(x) < \result + 1;
   @ also
   @ public exceptional_behavior
   @ requires x < 0;
   @ assignable \nothing;
   @ signals (IllegalArgumentException) true;
   @*/
public static int isqrt(int x) throws IllegalArgumentException {
    if (x < 0)
        throw new IllegalArgumentException();
    body
}
```

Making Exceptions Explicit (2)

- If several specifications are given with `also`, the method must fulfill **all** specifications.
- Specifications with `normal_behavior` implicitly have the clause
`signals (java.lang.Exception) false`
so the method must not throw an exception.
- Specifications with `exceptional_behavior` implicitly have the clause
`ensures false`
so the method must not terminate normally.