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,
- \( Act \) is 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.
The state of a Java program gives valuations local and global (heap) variables.

- $Q = \text{Heap} \times \text{Local}$
- $\text{Heap} = \text{Address} \rightarrow \text{Class} \times \text{seq Value}$
- $\text{Local} = \text{Identifier} \rightarrow \text{Value}$
- $\text{Value} = \mathbb{Z}, \text{Address} \subseteq \mathbb{Z}$

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

\[(\text{heap, lcl}) \xrightarrow{x \mapsto lcl(x)} (\text{heap, lcl})\]

axiom for evaluating constants:

\[(\text{heap, lcl}) \xrightarrow{c} (\text{heap, lcl})\]

rule for field access:

\[
(\text{heap, lcl}) \xrightarrow{e \uparrow v} (\text{heap}', lcl')
\]

\[
(\text{heap, lcl}) \xrightarrow{e \cdot \text{fld} \uparrow \text{heap}'(v)(idx)} (\text{heap}', lcl')
\]

where \(idx\) is the index of the field \(\text{fld}\) in the object \(\text{heap}'(v)\).
Rules for Assignment Expressions

rule for assignment to local:

\[
\begin{align*}
(\text{heap}, lcl) & \xrightarrow{e \triangleright v} (\text{heap}', lcl') \\
(\text{heap}, lcl) & \xrightarrow{x = e \triangleright v} (\text{heap}', lcl' \oplus \{x \mapsto v\})
\end{align*}
\]

rule for assignment to field:

\[
\begin{align*}
(\text{heap}_1, lcl_1) & \xrightarrow{e_1 \triangleright v_1} (\text{heap}_2, lcl_2) \\
(\text{heap}_2, lcl_2) & \xrightarrow{e_2 \triangleright v_2} (\text{heap}_3, lcl_3) \\
(\text{heap}_1, lcl_1) & \xrightarrow{e_1.\text{fld} = e_2 \triangleright v_2} (\text{heap}_4, lcl_3)
\end{align*}
\]

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):

\[
(heap, lcl) \xrightarrow{e.v} (heap', lcl') \\
(heap, lcl) \xrightarrow{e;} (heap', lcl')
\]

sequence of statements:

\[
(heap_1, lcl_1) \xrightarrow{s_1} (heap_2, lcl_2) \\
(heap_2, lcl_2) \xrightarrow{s_2} (heap_3, lcl_3) \\
(heap_1, lcl_1) \xrightarrow{s_1s_2} (heap_3, lcl_3)
\]
if statement:

\[
(\text{heap}_1, \text{lcl}_1) \xrightarrow{\text{if}(e)s_1\text{else}s_2} (\text{heap}_3, \text{lcl}_3), \text{where } v \neq 0
\]

\[
(\text{heap}_1, \text{lcl}_1) \xrightarrow{\text{if}(e)s_1\text{else}s_2} (\text{heap}_3, \text{lcl}_3)
\]

while statement:

\[
(\text{heap}_1, \text{lcl}_1) \xrightarrow{\text{if}(e)\{s\text{ while}(e)s\}} (\text{heap}_2, \text{lcl}_2)
\]

\[
(\text{heap}_1, \text{lcl}_1) \xrightarrow{\text{while}(e)s} (\text{heap}_2, \text{lcl}_2)
\]
Rule for Java Method Call

\[
(\text{heap}_1, \text{lcl}_1) \xrightarrow{e \triangleright v} (\text{heap}_2, \text{lcl}_2) \\
(\text{heap}_2, \text{lcl}_2) \xrightarrow{e_1 \triangleright v_1} (\text{heap}_3, \text{lcl}_3) \\
\vdots \\
(\text{heap}_{n+1}, \text{lcl}_{n+1}) \xrightarrow{e_n \triangleright v_n} (\text{heap}_{n+2}, \text{lcl}_{n+2}) \\
(\text{heap}_{n+2}, \text{mlcl}) \xrightarrow{\text{body}} (\text{heap}_{n+3}, \text{mlcl}') \\
(\text{heap}_1, \text{lcl}_1) \xrightarrow{e \cdot m(e_1, \ldots, e_n) \triangleright \text{mlcl}'(\text{result})} (\text{heap}_{n+3}, \text{lcl}_{n+2})
\]

where \text{body} is the body of the method \text{m} in the object \text{heap}_{n+2}(v), and \text{mlcl} = \{ \text{this} \mapsto v, \text{param}_1 \mapsto v_1, \ldots, \text{param}_n \mapsto v_n \} where \text{param}_1, \ldots, \text{param}_n are the names of the parameters of \text{m}
Creating an Object is always combined with the call of a constructor:

\[
heap_1 = heap \cup \{ na \mapsto (Type, \langle 0, \ldots, 0 \rangle) \}
\]

\[
(\text{heap}_1, \text{lcl}) \xrightarrow{\text{na.<init>(e_1,\ldots,e_n)\triangleright v}} (\text{heap}', \text{lcl}')
\]

\[
(\text{heap}, \text{lcl}) \xrightarrow{\text{new Type(e_1,\ldots,e_n)\triangleright na}} (\text{heap}', \text{lcl}')
\]

Here \text{<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:
  \[ Q = \text{Flow} \times \text{Heap} \times \text{Local} \]

- \( \text{Flow} ::= \text{Norm} | \text{Ret} | \text{Exc} \langle \langle \text{Address} \rangle \rangle \)

We use the identifiers \( \text{flow} \in \text{Flow}, \text{heap} \in \text{Heap} \) and \( \text{lcl} \in \text{Local} \) 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:

\[
(flow, heap, lcl) \xrightarrow{E>^v} (flow, heap, lcl), \text{ where } flow \neq \text{Norm}
\]

\[
(flow, heap, lcl) \xrightarrow{s} (flow, heap, lcl), \text{ where } flow \neq \text{Norm}
\]
Expressions With Exceptions

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

\[
\begin{align*}
(Norm, heap, lcl) & \xrightarrow{e_1 \triangleright v_1} q \
q & \xrightarrow{e_2 \triangleright v_2} q' \\
(Norm, heap, lcl) & \xrightarrow{e_1 \ast e_2 \triangleright (v_1 \cdot v_2) \mod 2^{32}} q'
\end{align*}
\]

\[
\begin{align*}
(Norm, heap, lcl) & \xrightarrow{st_1} q \
q & \xrightarrow{st_2} q' \\
(Norm, heap, lcl) & \xrightarrow{st_1; st_2} q'
\end{align*}
\]

\[
\begin{align*}
(Norm, heap, lcl) & \xrightarrow{e \triangleright v} q \\
q & \xrightarrow{s_1} q' \\
(Norm, heap, lcl) & \xrightarrow{\text{if}(e) \ s_1 \ \text{else} \ s_2} q', \text{ where } v \neq 0
\end{align*}
\]

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

\[
\begin{align*}
(flow, heap, lcl) & \xrightarrow{e \triangleright v} (flow, heap, lcl), \text{ where } flow \neq Norm
\end{align*}
\]
Throwing Exceptions

\[
\begin{align*}
(Norm, \ heap, \ lcl) \xrightarrow{e \triangleright v} (Norm, \ heap', \ lcl') \\
(Norm, \ heap, \ lcl) \xrightarrow{\text{throw } e;} (Exc(v), \ heap', \ lcl')
\end{align*}
\]

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

\[
\begin{align*}
(Norm, \ heap, \ lcl) \xrightarrow{e \triangleright 0} q' \\
q' \xrightarrow{\text{throw new NullPointerException()}} q''
\end{align*}
\]

where \( v \) is some arbitrary value.

\[
\begin{align*}
(Norm, \ heap, \ lcl) \xrightarrow{e.fld \triangleright v} q''
\end{align*}
\]
Complete Rules for \textbf{throw}

\[
\begin{align*}
(Norm, heap, lcl) & \xrightarrow{e \triangleright v} (Norm, heap', lcl') \\
(Norm, heap, lcl) & \xrightarrow{\text{throw } e; } (Exc(v), heap', lcl')
\end{align*}
\]

where \( v \neq 0 \)

\[
\begin{align*}
(Norm, heap, lcl) & \xrightarrow{e \triangleright 0} q' \\
q' & \xrightarrow{\text{throw } \text{new } NullPointerException(\cdot) } q''
\end{align*}
\]

\[
\begin{align*}
(Norm, heap, lcl) & \xrightarrow{\text{throw } e; } q''
\end{align*}
\]

\[
\begin{align*}
(Norm, heap, lcl) & \xrightarrow{e \triangleright v} (flow', heap', lcl') \\
(Norm, heap, lcl) & \xrightarrow{\text{throw } e; } (flow', heap', lcl')
\end{align*}
\]

where \( flow' \neq Norm \)
Catching an exception:

\[(Norm, heap, lcl) \xrightarrow{s_1} (Exc(v), heap', lcl')\]

\[(Norm, heap', lcl' \cup \{ex \mapsto v\}) \xrightarrow{s_2} q'')\]

where \(v\) is an instance of \(Type\)

\[(Norm, heap, lcl) \xrightarrow{\text{try } s_1 \text{ catch}(Type \ ex) s_2} q''\]

No exception caught:

\[(Norm, h, l) \xrightarrow{s_1} (flow', h', l')\]

\[(Norm, h, l) \xrightarrow{\text{try } s_1 \text{ catch}(Type \ ex) s_2} (flow', h', l')\]

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:

\[
\begin{align*}
(Norm, heap, lcl) &\xrightarrow{e \triangleright v} (Norm, heap', lcl') \\
(Norm, heap, lcl) &\xrightarrow{\text{return } e} (Ret, heap', lcl' \oplus \{ \text{result } \mapsto v \})
\end{align*}
\]

But evaluating e can also throw exception:

\[
\begin{align*}
(Norm, heap, lcl) &\xrightarrow{e \triangleright v} (flow, heap', lcl') \\
(Norm, heap, lcl) &\xrightarrow{\text{return } e} (flow, heap', lcl')
\end{align*}
\]

where flow \neq Norm
Method Call (Normal Case)

\[
\begin{align*}
  (\text{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{\text{body}} (\text{Ret}, h_{n+3}, ml') \\

  (\text{Norm}, h_1, l_1) & \xrightarrow{e.m(e_1, \ldots, e_n) \triangleright ml'(	ext{\textbackslash result})} (\text{Norm}, \text{heap}_{n+3}, l_{n+2})
\end{align*}
\]

where \( param_1, \ldots, param_n \) are the names of the parameters and \( \text{body} \) is the body of the method \( m \) in the object \( \text{heap}_{n+2}(v) \), and

\[ ml = \{ \text{this} \mapsto v, param_1 \mapsto v_1, \ldots, param_n \mapsto v_n \} \]
Method Call With Exception

\[(Norm, h_1, l_1) \xrightarrow{e \triangleright v} q_2\]
\[q_2 \xrightarrow{e_1 \triangleright v_1} q_3\]
\[\vdots\]
\[q_{n+1} \xrightarrow{e_n \triangleright v_n} (f_{n+2}, h_{n+2}, l_{n+2})\]
\[(f_{n+2}, h_{n+2}, ml) \xrightarrow{\text{body}} (\text{Exc}(v_e), h_{n+3}, ml')\]
\[(Norm, h_1, l_1) \xrightarrow{\text{e.m}(e_1, \ldots, e_n) \triangleright ml'(\text{\textbackslash result})} (\text{Exc}(v_e), \text{heap}_{n+3}, l_{n+2})\]

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

\[(\text{Norm, heap, lcl}) \xrightarrow{\text{body}} (\text{Ret, heap}', \text{lcl}'),\]

if \(\text{lcl}(x) \geq 0\) then the formula

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

holds.
What About Exceptions?

```java
/*@ 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:

```java
/*@ 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

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

holds and \(heap = heap'\).
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

$\text{(Norm, heap, lcl)} \xrightarrow{x \geq 0 \triangleright v} (fl, heap, lcl)$

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

\[
\text{requires } e_1 \\
\text{ensures } e_2
\]

iff for all executions

\[
(Norm, heap, lcl) \xrightarrow{\text{body}} (Ret, heap', lcl')
\]

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 \( \text{old} \):

\[
(Norm, heap, lcl) \xrightarrow{e \triangleright v} q \quad \text{where } heap, lcl \text{ is the state of the program before } body \text{ was executed}
\]

\[
(Norm, heap', lcl') \xrightarrow{\text{old}(e) \triangleright v} q
\]
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;
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.