Formal Methods for Java
Lecture 28: Conclusion

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Feb 12, 2012
In KeY, the default rule is to inline the procedures. **Advantages:**
- No function contract needed.
- No separate proof for correctness of function needed.

But it has several disadvantages:
- Proof gets larger (especially important if proof is interactive).
- Proof has to be repeated for every function call.
- No recursive procedures possible.
The rule “Use Operation Contract” allows compositional proofs. It opens three subgoals:

- **Pre:** Show that pre-condition holds (this includes class invariants).
- **Post:** Show that with the post-condition, the remaining program is correct.
- **Exceptional Post:** Show that if called method throws an exception, the remaining program is correct.

**Note:** Use Operation Contract cannot be used for the method you are just proving.
Unfortunately, KeY has no direct support for recursive functions.

An induction proof can work. Ingredients:

- A precondition $pre$,
- A postcondition $post$,
- A ranking function $rank$.

Show by induction over $r$:
\[
\forall \text{int } x. \ (pre \land rank < r) \rightarrow \ \langle \text{result} = \text{methodcall}(x); \ \rangle \ post
\]
Lecture Summary
<table>
<thead>
<tr>
<th>Lecture</th>
<th>Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction to JML and JLS</td>
</tr>
<tr>
<td>2–3</td>
<td>Operational Semantics</td>
</tr>
<tr>
<td>4–5</td>
<td>JML</td>
</tr>
<tr>
<td>6–10</td>
<td>Java Pathfinder</td>
</tr>
<tr>
<td>11</td>
<td>ESC/Java</td>
</tr>
<tr>
<td>12–14</td>
<td>Ownership/Friendship and Invariants</td>
</tr>
<tr>
<td>15–19</td>
<td>Jahob</td>
</tr>
<tr>
<td>20–22</td>
<td>Sequent Calculus</td>
</tr>
<tr>
<td>23–28</td>
<td>Dynamic Logic and Proving with KeY</td>
</tr>
</tbody>
</table>
Motivations

Quality

- Leads to better understood code.
- Different view point reveals bugs.
- Formal proof can rule out bugs entirely.

Productivity

- Error detection in early stages of development.
- Modular specifications allow reuse of components.
- Documentation, maintenance.
- Automatic test case generation.
- Clearer specification leads to better software.
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.
The state of a Java program consists of a flow component and valuations for local and global (heap) variables.

- \( Q = Flow \times Heap \times Local \)
- \( Flow ::= \text{Norm} | \text{Ret} | \text{Exc} \langle \text{Address} \rangle \)
- \( Heap = Address \rightarrow \text{Class} \times \text{seq} \text{Value} \)
- \( Local = Identifier \rightarrow \text{Value} \)
- \( Value = \mathbb{Z}, Address \subseteq \mathbb{Z} \)

A state is denoted as \( q = (flow, heap, lcl) \), where \( flow : Flow \), \( heap : Heap \) and \( lcl : Local \).
Rules of Operational Semantics

\[
\begin{align*}
(Norm, heap, lcl) & \xrightarrow{e_{1} \triangleright v_{1}} q & q & \xrightarrow{e_{2} \triangleright v_{2}} q' \\
(Norm, heap, lcl) & \xrightarrow{e_{1} \cdot e_{2} \triangleright (v_{1} \cdot v_{2}) \mod 2^{32}} q' \\
(Norm, heap, lcl) & \xrightarrow{st_{1}} q & q & \xrightarrow{st_{2}} q' \\
(Norm, heap, lcl) & \xrightarrow{st_{1}; st_{2}} q' \\
(Norm, heap, lcl) & \xrightarrow{e \triangleright v} q & q & \xrightarrow{bl_{1}} q' \\
(Norm, heap, lcl) & \xrightarrow{if(e) bl_{1} else bl_{2}} q' \\
\end{align*}
\]

... and many more.
Rules for Exceptions

\[
(Norm, heap, lcl) \xrightarrow{e \triangleright v} (Norm, heap', lcl') \\
(Norm, heap, lcl) \xrightarrow{\text{throw } e} (Exc(v), heap', lcl')
\]

A null-pointer dereference works like a throw statement:

\[
(Norm, heap, lcl) \xrightarrow{e \triangleright 0} q' \\
q' \xrightarrow{\text{throw new } NullPointerException()} q'' \quad \text{where } v \text{ is some arbitrary value}
\]

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

Propagating exceptions:

\[
(flow, heap, lcl) \xrightarrow{\alpha} (flow, heap, lcl), \text{ where } flow \neq Norm
\]
public class ArrayOps {
    private /*@ spec_public @*/ Object[] a; // @ public invariant 0 < a.length;
    @requires 0 < arr.length;
    @ensures this.a == arr;
    @
    public void init(Object[] arr) {
        this.a = arr;
    }
}
The Java Modelling Language (JML)

JML is a behavioral interface specification language (BISL) for Java

- Proposed by G. Leavens, A. Baker, C. Ruby:
  * JML: A Notation for Detailed Design*, 1999

- It combines ideas from two approaches:
  - Eiffel with its built-in language for Design by Contract (DBC)
  - Larch/C++ a BISL for C++
Tools for JML

- http://www.jmlspecs.org/
- Release can be downloaded from http://sourceforge.net/projects/jmlspecs/files
- JML compiler (jmlc)
- JML runtime assertion checker (jmlrac)

External Tools:
- ESC/Java
- KeY
- and many more ...
Advantages of run-time checking:

- Easy to use.
- Supports a large sub-language of JML.
- No false warnings.

Disadvantages of run-time checking:

- Coverage only as good as test cases that are used.
- Does not prove absence of errors.
Advantages of model-checking:
- Almost as easy as testing.
- More exhaustive than simple testing.

Disadvantages of model-checking:
- State explosion problem.
- Runtime vs. coverage.
Advantages of static checking:
- Easy to use.
- No test cases needed.
- Better coverage than runtime checking.
- Can detect missing specification.

Disadvantages of static checking:
- Only a small subset of JML supported.
- Many spurious warnings (not complete).
Advantages of static checking:

- Prove of correctness.
- Both sound and complete (modulo Peano Axioms).

Disadvantages of static checking:

- Very difficult to use.
- Can require interactive proving.
Suggested order

1. Run-time checking, e.g. jmlrac and jmlunit.
2. Static checking, e.g. ESC/Java.
3. Model-checking, e.g. Java Pathfinder
4. Theorem proving, e.g. KeY.

Ensures that most bugs are already found before starting with theorem proving. Some prefer doing static checking before run-time checking (no test cases needed).