JML and Java 1.5+

David R. Cok
Eastman Kodak Company, Research Laboratories

9 October 2008
SAVCBS08 workshop

Java 1.5 was a big step (in 2004)

- Tools built on or for Java had to make a considerable infrastructure investment
- Certainly the case for the Java Modeling Language tool set (JML2)
  - built on a research compiler
  - graduate student moved on...
- Older tool set is maintained, but there is a strong desire for JML tools that support current Java

Some goals for a new JML tool set

- Built on a supported compiler base
  - use other people’s work, timely updates
  - suitable licensing for research and application use
- Good for research use
  - easily extendable with a clear design
- Good for causal use
  - Clear diagnostics
  - Well-integrated into an IDE
  - Available as command-line tools also
- Good for practical/industrial use
  - Robust and reasonably complete
  - Well-integrated into a commonly used tool environments
  - Reasonable time and memory footprints

Two development efforts

- Eclipse-based (cf. Chalin et al.)
  - popular, well-used, easy to integrate IDE
  - actively supported Java infrastructure
  - can do command-line tools, but is a bit heavy-weight for that purpose
  - need to really get into the details of the compiler in order to extend it
  - Eclipse has efficient (and hence a bit complex) compilation and AST structure

Two development efforts

- Open JDK based
  - straightforward command-line based tools
  - quite extensible design
  - JML can be added almost completely by derivation
  - no IDE – can be bolted on to Eclipse in the way that many tools are (compilation processes run twice)
  - used for research in JML and static verification (limited resources)

The question for this talk:

- What specification language issues arise with the move to Java 1.5+?

- JML is a BISL with the philosophy that the specification language should be similar to the programming language.
- Should expect corresponding changes in JML as the Java language changes.

Java 1.5+ changes of note

- Generic types
- enhanced for
- auto boxing and unboxing
- annotations
- varargs
- static import
- enum types
- java.lang.SuppressWarnings
- new APIs: compiler, AST, annotation processing

Parsing

- Generic types (and all the other new constructs) can be used in JML specs as well as in Java
  - It helps greatly to be able to repurpose a compiler’s lexing/parsing/name and type resolution infrastructure for JML in addition to Java
  - There are JML constructs as well, so extension is essential

Refinement

specification:

```
class Exp<Q> {

   //@ ....
   <X> void m(X x);

   //@ ...
   Q m(int i, Q q);

}
```

java implementation:

```
class Exp<E> {

   <T> void m(T t) { ... }

   int m(int i, E e) { ... }


}
```

- **Need to match up methods (and fields and classes) in specs with methods in implementation**
  - **complicated by overloading**
  - **need type resolution before matching**

Other straightforward features...

- Enums
  - no changes needed
- varargs
  - no changes needed
- static import
  - no changes needed
- callable clause, \only_called predicate
  - change in syntax to allow specifying generic and variable argument methods:

```
callable <T> T [ ] collection<T>.toArray(T[ ]);
```

Autoboxing and unboxing

In JML < and <= are overloaded to designate a lock ordering on objects, so JML allows

```
Integer i, j;    //@ boolean b = i < j;
```

(i<j is illegal in Java 1.4)

Autoboxing and unboxing

In Java 1.5+: with int k, kk; Integer i, ii;

	Java	JML
k < kk	less than	less than
i < ii	less than	ambiguous
i < kk	less than	ambiguous

- Resolution: use a non-overloaded operator to represent lock ordering: <# and <#=

Enhanced for loop

Typical loop specification

```
int sum = 0;
//@ loop_invariant 0 <=i && i <= N;
//@ loop_invariant sum = i * ( i - 1) / 2;
//@ decreasing N – i;
for (int i=0; i<N; i++) {
        sum += i;
}
//@ assert sum = N * (N - 1) /2;
```

Enhanced for loop

Expands into

```
//@ assume 0 <= N;
int sum = 0;
int i = 0;
while (i<N) {
        //@ assert 0 <= i && i <= N;
        //@ assert sum == i * ( i - 1) / 2;
        sum += i;
        i++; // update
}
//@ assert 0 <= i && i <= N;
//@ assert sum == i * ( i - 1) / 2;
//@ assert sum == N * (N - 1) /2;
```

Enhanced for loop

An enhanced for loop has no loop variable to use in the loop invariants. Compare

```
int[ ] array = ...
int sum = 0;
for (int element: array) {
   sum += element;
}
```

with

```
int[ ] array = ...
int sum = 0;
//@ loop_invariant sum == (\sum int j; 0<=j && j<i; array[j]);
for (int i=0; i< array.length; i++) {
   sum += element;
}
```

Enhanced for loop

Introduce (readonly) ghost variables \values (cf. Spec#) and \index

- int \index
  - the index in the array of the current element
  - the number of complete iterations so far
- JMLList<T> \values
  - a list of values obtained so far

Enhanced for loop

```
int sum = 0;
//@ loop_invariant sum == (\sum int k; 0<=k && k < \index; array[k]);
//@ loop_invariant 0 <= \index && \index <= array.length; // implicit
//@ decreasing array.length - \index;                     // implicit
for (int e: array) {
  sum += e;
  //@ assert e == array[\index];
}
```

Enhanced for loop

Using \values:

```
Set<Integer> set = ...
int max = Integer.MIN_VALUE;
/*@ maintains max == \values.size() == 0 ? Integer.MIN_VALUE :
              (max int k; \values.contains(k); k); */
for (Integer i: set) {
   if (max < i) max = i;
}
```

Enhanced for loop – a design option

```
for (int element: array) {
    ... body...
}
```

is

```
int \index = 0;
JMLList<Integer> \values = new ...
while (\index<array.length) {
   int element = array[\index];
   ... body ...
    \index ++;
    \values.add(element);
}
```

in ... body...:

\index == \values.size( )

but

element is not in \values
(it is in \values in the invariant)

Enhanced for loop – a design option

or

\index != \values.size() in the body
(but it is equal in the invariants)

element is in \values in the body

```
int \index = 0;
JMLList<Integer> \values = new ...
while (\index<array.length) {
   int element = array[\index];
   \values.add(element);
   ... body ...
    \index ++;
}
```

Type manipulation in JML

JML	Java 1.4
\TYPE	Class
\type(T)	T.class
\typeof(e)	e.getClass(); (e has ref type)
\typeof(p)	P.class (primitive type P)
t1 <: t2	t2.isAssignableFrom(t1)
\elemtype(at)	at.getComponentType();

Types – problems in Java 1.5+

Java does not keep type parameter information at runtime:

- Cannot write, e.g., List<Integer>.class
- The Class<?> value does not keep type parameter information
- isAssignableFrom does not reflect inheritance

```
Collection<Integer> ci = new LinkedList<Integer>();
 Collection<Boolean> cb = new LinkedList<Boolean>();
 boolean bb = ci.getClass().isAssignableFrom(cb.getClass()); // true
 ci = cb; // syntax error
```

- Cannot write, e.g., o instanceof LinkedList<Integer>

Limits what can be stated in JML about types

elementType idiom

Pre-generics, JML tracked a collection’s element type with a ghost field:

```
class LinkedList {
        //@ public ghost \TYPE elementType;
}
```

However, we cannot write

```
class LinkedList<T> {
        //@ public ghost \TYPE elementType;
        public LinkedList() {
                //@ set elementType = T.class;
        }
}
```

elementType idiom

- We should not need the elementType idiom anymore – use the class’s type parameter instead

- But Java syntax limitations do not allow treating the type parameter in the same way as a type name

Types – possible solutions for JML

- Wait until Java incorporates full type information at runtime

- Represent \TYPE as a class that incorporates Java type information and type parameter information
  (so \type, \typeof, <:, \elemtype would all act on \TYPE objects, with autoconversion from Class objects)
  - dedicated implementation, OR
  - use javax annotation api for types
    - » designed to represent existing source
    - » not as convenient for arbitrary types

Annotations

- JML provides nowarn (lexical)
- Java provides java.lang.SuppressWarnings
  - an annotation on classes, methods, declarations
  - not on statements
  - not as flexible as JML’s nowarn at present
  - much more standard
    - » but needs standard names for specification failures

Using annotations for specification

- JML has used special comments: //@ ...

- Qualifiers such as @Pure are easily enabled

- Possibility of using annotations: e.g.

  @Requires(“o != oo”)

  [cf. Boysen, ISU TR 08-03]

  - varying degrees of usability
  - need to be able to parse and typecheck the strings inside the annotations in the correct scope

Using annotations for specification

- Many different tools proposing various qualifiers, e.g. @Pure, @NonNull, @Positive, ...
  - JML: @Nullable, @NonNull
  - JSR-308: @Nullable, @NonNull
  - IntelliJ: @Nullable, @NotNull
  - JSR-305: @Nullable, @CheckForNull, @NonNull

  (and all in different packages)

- Need some cooperation and standardization of annotation names and packages

- Prefer a general mechanism rather than a plethora of specification names

Specifications

JML provides model classes

- these need to be converted to generic classes
- they need specifications vetted for efficient proving

JDK classes need extensive specifications

New APIs (Java 1.6)

- syntax tree API
  - readonly
  - still need extensions for JML features
  - provides type information

- annotation processing
  - perhaps recast runtime checking as an annotation processor that rewrites the source
  - perhaps recast static checking as an annotation processor, using the syntax tree API and type mirrors

Others

- \bigint, \real, java.lang.BigInteger, org.jmlspecs.lang.JMLReal

- generic axioms etc.

- \nonnullelements
  - make the signature \nonnullelements(Object[ ] t)

- set comprehension

- \lockset, \max, JMLSetType

Conclusions

- JML needs to evolve along with Java, particularly in handling type information as first-class objects
- Other more minor adjustments
- Need collaboration on names and semantics of annotations
- Need writing and experimentation with library specifications

Kodak