Software Verification for Java 5
KeY Symposium 2007

Mattias Ulbrich

June 14, 2007
KeY + Java 5

Typesafe Enumeration Datatypes

Enhanced For Loops

Generic Classes
1. Keep pace with the progress of the industrial standard

2. Examine KeY’s flexibility and adaptibility

3. Do the new features support verification?

4. Do they need verification?
Novelties in the language in Java 5

- Typesafe enumeration types
- Iteration loops
- Auto-Boxing of primitive types
- Generic classes
- Covariant return types
- Static imports
- Annotations
- Variable arguments
Novelties in the language in Java 5

- Typesafe enumeration types
- Iteration loops
- Auto-Boxing of primitive types
- Generic classes
- Covariant return types
- Static imports
- Annotations
- Variable arguments

No relevance for verification
Novelties in the language in Java 5

- Typesafe enumeration types ✔
- Iteration loops ✔
- Auto-Boxing of primitive types ✔
- Generic classes ❌
- Covariant return types ❌
- Static imports ❌
- Annotations ❌
- Variable arguments ❌

No relevance for verification
Typesafe Enumeration Datatypes
Typesafe Enumeration Datatypes

```
enum E { e₁, e₂, . . . , eₙ }
```

- A new keyword to declare enumeration types: `enum`
- followed by the name of the datatype
- followed by the `enum constants`

- `enum` declares reference types – not primitive types
- the enum constants *uniquely* enumerate *all* (non-null) instances

**Example**

```
enum Season { SPRING, SUMMER, AUTUMN, WINTER }
```
Using the object repository

Enumerations are reference types (special classes in fact)

Use the mechanisms available for reference types.

The object repository $C::\langle\text{get}\rangle() : \text{Nat} \mapsto C$

For every exact instance $o$ of a class $C$ there is an index $i \in \text{Nat}$
with $o = C::\langle\text{get}\rangle(i)$.
Using the object repository

Enumerations are reference types (special classes in fact)

⇒ Use the mechanisms available for reference types.

The object repository \( C::\langle\text{get}\rangle() : \text{Nat} \mapsto C \)

For every exact instance \( o \) of a class \( C \) there is an index \( i \in \text{Nat} \)
with \( o = C::\langle\text{get}\rangle(i) \).

Repository access for Enums:

\[
\begin{align*}
E.e_1 &= E::\langle\text{get}\rangle(0) \\
E.e_2 &= E::\langle\text{get}\rangle(1) \\
&\vdots \\
E.e_n &= E::\langle\text{get}\rangle(n-1) \\
E::\langle\text{nextToCreate}\rangle &= n
\end{align*}
\]
Advantages

Using the standard object repository is good:

- Only few new rules in the calculus to handle enums
- Use established techniques
- Problems on enum instances are reduced to problems on their indexes, thus natural numbers
- Scales well
Enhanced For Loops
**Enhanced For Loops**

**Purpose**

The enhanced for loop allows to iterate through a collection or an array without having to create an explicit Iterator or counter variable.
### Purpose

The enhanced for loop allows to iterate through a collection or an array without having to create an explicit Iterator or counter variable.

### Traditional Java

```java
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}
```
**Enhanced For Loops**

**Purpose**

The enhanced for loop allows to iterate through a collection or an array without having to create an explicit Iterator or counter variable.

**Traditional Java**

```java
for(int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}
```

**Java 5**

```java
for(int x : array) {
    System.out.println(x);
}
```


Equivalent loops

```java
int a[] = array;
for(int i = 0; i < a.length; i++) {
    int x = a[i];
    /* body */
}
```

```java
for(int x : array) {
    /* body */
}
```
Equivalent loops

```java
int a[] = array;
for (int i = 0; i < a.length; i++) {
    int x = a[i];
    /* body */
}
```

```java
for (int x : array) {
    /* body */
}
```

1. `a` and `i` are new variables not accessible from within `body`
2. `a.length` is constant in this context
3. The counter `i` is incremented in every iteration

⇒ There are finite many iterations

⇒⇒ The loop terminates if every iteration terminates.
Invariant rules with termination

Null Case
Base Case
Abnormal body termination
Invariant preserved
Use Case

\[
\text{enhForArrayInv} \quad \frac{\Gamma \vdash \mathcal{U} \langle \text{for}(ty \ x : se)\{ \ p \ \} \rangle \varphi, \ \Delta}{\text{Γ} \vdash \mathcal{U} \langle \cdot \rangle}
\]

1. uses the $\langle \cdot \rangle$-modality
2. the sequents contain more formulae: the encoded extra knowledge about the special loop.
“Enhanced For = Enhanced Performance”

Experimental results using this rule

Verification of the “maximum in an array” loop.

<table>
<thead>
<tr>
<th></th>
<th>new rule</th>
<th>while rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes in the proof tree</td>
<td>374</td>
<td>1053</td>
</tr>
<tr>
<td>Branches in the proof tree</td>
<td>8</td>
<td>21</td>
</tr>
<tr>
<td>Additional manual instantiations</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

⇒ Complexity reduced to roughly a third.

A syntactical entity that is specialised allows to retrieve more information and thereby shorten proofs.
Generic Classes

= Parametric Polymorphism
Generics* improve static typing and type safety

* if they were well-implemented
Generics* improve static typing and type safety

### Traditional Java

```java
Vector v = new Vector();
v.add("String");
String s = (String)v.get(0);
```

### Java 5

```java
Vector<String> v = new Vector<String>();
v.add("String");
String s = v.get(0);
```

* if they were well-implemented
## Generics* improve static typing and type safety

<table>
<thead>
<tr>
<th>Traditional Java</th>
<th>Java 5</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Vector v = new Vector();</code></td>
<td><code>Vector&lt;String&gt; v = new Vector&lt;String&gt;();</code></td>
</tr>
<tr>
<td><code>v.add(&quot;String&quot;);</code></td>
<td><code>v.add(&quot;String&quot;);</code></td>
</tr>
<tr>
<td>String s = (String)v.get(0);</td>
<td>String s = v.get(0);</td>
</tr>
</tbody>
</table>

- Type checking performed at run-time
- failure must be taken into account by verifier

- Type checking performed at compile-time
- no possible exception that must be taken into account by verifier

* if they were well-implemented
## Polymorphic functions

### Attributes induce functions

```java
class Chain {
    Chain tail;
    Object head;
}  

head : Chain → Object
```

This is a well-known concept in type-theory, but not in many-sorted logics.
Polymorphic functions

Attributes induce functions

```java
class Chain {
    Chain tail;
    Object head;
}
```

Polymorphic attributes induce polymorphic functions

```java
class Chain<T> {
    Chain<T> tail;
    T head;
}
```

This is a well-known concept in type-theory, but not in many-sorted logics.
## Infinite type system

### “Parametric recursion”

<table>
<thead>
<tr>
<th>String</th>
</tr>
</thead>
</table>

is a valid type that can show up at run-time.
Infinite type system

“Parametric recursion”

Vector<String>

is a valid type that can show up at run-time.
Infinite type system

“Parametric recursion”

Vector<Vector<String>>

is a valid type that can show up at run-time.
Infinite type system

“Parametric recursion”

Vector<Vector<Vector<String>>>

is a valid type that can show up at run-time.
Infinite type system

“Parametric recursion”

Vector<...Vector<Vector<String>>>...

is a valid type that can show up at run-time.
### Infinite type system

#### “Parametric recursion”

\[
\text{Vector<...Vector<Vector<String>>>...>}
\]

is a valid type that can show up at run-time.

#### Problem

Some rules need a finite type system to enumerate types
(method dispatch, dynamic subtypes, . . . )
Infinite type system

“Parametric recursion”

\[
\text{Vector} \langle \ldots \text{Vector} \langle \text{Vector} \langle \text{Vector} \langle \text{String} \rangle \rangle \rangle \rangle \ldots \rangle
\]

is a valid type that can show up at run-time.

Problem

Some rules need a finite type system to enumerate types (method dispatch, dynamic subtypes, \ldots )

Handle this in JavaDL ... 

... with existentially quantified type variables

\[
\exists X. \, \text{object} \, \in_1 \, \text{Vector} \langle X \rangle
\]
Type Meta-types

- Add the “type of reference types” \( \mathbb{J} \) to the type hierarchy.
- Add the reference types as new objects to the domain.
- Add appropriate function symbols to the signature.

\[ \Rightarrow \] Allow quantification over types class.
## Generic contracts

<table>
<thead>
<tr>
<th>Method contracts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a pre-condition $pre$ prior to a method call $o.m()$, a post-condition $post$ holds afterwards:</td>
</tr>
<tr>
<td>$pre \rightarrow \langle o.m() ; \rangle post$</td>
</tr>
</tbody>
</table>
Generic contracts

Method contracts

Given a pre-condition \( \text{pre} \) prior to a method call \( o.m() \), a post-condition \( \text{post} \) holds afterwards:

\[
\text{pre} \rightarrow \langle o.m(); \rangle \text{post}
\]

Generic method contracts

Contracts for methods in generic classes are implicitly universally quantified over all types \( T : \mathbb{J} \):

\[
\forall T : \mathbb{J}. \quad \text{pre}(T) \rightarrow \langle o.m(); \rangle \text{post}(T)
\]
Generics and JavaDL

- Adapt ideas from type theory to JavaDL.
- “Lift” types to the object level as type $\mathbb{J}$.
- Allow quantification over types ...
- ... and instantiations
- generic attributes lead to polymorphic functions in the logic.
Generics and JavaDL

- Adapt ideas from type theory to JavaDL.
- “Lift” types to the object level as type $\mathcal{J}$.
- Allow quantification over types ...
- ... and instantiations
- generic attributes lead to polymorphic functions in the logic.

$\implies$ Severe changes in some fundamental concepts of the logic.
Summary
KeY + Java 5

Remember: Goals to examine

1. How the new features support / need verification
2. KeY’s flexibility and adaptibility
KeY + Java 5

Remember: Goals to examine

1. How the new features support / need verification
2. KeY’s flexibility and adaptibility

To sum it up ...

<table>
<thead>
<tr>
<th>Feature</th>
<th>Needs Verif.</th>
<th>Supports Verif.</th>
<th>Fits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enums</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Enh. For</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Boxing</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Generics</td>
<td>NO*</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>
Thank You!
Non-termination if iterating a collection

Nicht übertragbar

Results for arrays quite promising – but cannot be transferred to the iterator case as well.

Consider a singly-chained list that is iterated and appended to at the same time: The iteration process will not terminate.

\[\text{iterator} \downarrow\]

\[
\begin{array}{c}
\text{ } \\
\text{ } \\
\end{array}
\]

\[
\begin{array}{c}
\text{ } \\
\text{ } \\
\text{ } \\
\end{array}
\]
Non-termination if iterating a collection

Nicht übertragbar

Results for arrays quite promising – but cannot be transferred to the iterator case as well.

Consider a singly-chained list that is iterated and appended to at the same time: The iteration process will not terminate.
Non-termination if iterating a collection

Nicht übertragbar

Results for arrays quite promising – but cannot be transferred to the iterator case as well.

Consider a singly-chained list that is iterated and appended to at the same time: The iteration process will not terminate.
Non-termination if iterating a collection

Nicht übertragbar

Results for arrays quite promising – but cannot be transferred to the iterator case as well.

Consider a singly-chained list that is iterated and appended to at the same time: The iteration process will not terminate.
Non-termination if iterating a collection

Nicht uebertragbar

Results for arrays quite promising – but cannot be transferred to the iterator case as well.

Consider a singly-chained list that is iterated and appended to at the same time: The iteration process will not terminate.
Non-termination if iterating a collection

Nicht uebertragbar

Results for arrays quite promising – but cannot be transferred to the iterator case as well.

Consider a singly-chained list that is iterated and appended to at the same time: The iteration process will not terminate.
Non-termination if iterating a collection

Nicht übertragbar

Results for arrays quite promising – but cannot be transferred to the iterator case as well.

Consider a singly-chained list that is iterated and appended to at the same time: The iteration process will not terminate.
Non-termination if iterating a collection

Nicht übertragbar

Results for arrays quite promising – but cannot be transferred to the iterator case as well.

Consider a singly-chained list that is iterated and appended to at the same time: The iteration process will not terminate.
Non-termination if iterating a collection

Nicht übertragbar

Results for arrays quite promising – but cannot be transferred to the iterator case as well.

Consider a singly-chained list that is iterated and appended to at the same time: The iteration process will not terminate.
Auto-Boxing and Unboxing

Idea

Bring primitive datatypes and reference types closer together and make them more interoperable.

Primitive types:
- int
- double
- boolean
- ...

Reference types:
- Integer
- Double
- Boolean
- ...

Boxing

Unboxing
Auto-Boxing and Unboxing

Bring primitive datatypes and reference types closer together

**Manual boxing in traditional Java**

```
Integer intObj = new Integer(3);
int intvalue = intObj.intValue();
```

**Auto-boxing in Java 5**

```
Integer intObj = 3;
int intvalue = intObj;
```
Auto-Boxing and Unboxing

Bring primitive datatypes and reference types closer together

Manual boxing in traditional Java

```java
Integer intObj = new Integer(3);
int intValue = intObj.intValue();
```

Auto-boxing in Java 5

```java
Integer intObj = 3;
int intValue = intObj;
```

Important:

- parts of the behaviour left open by the specification
- Can give rise to unexpected NullPointerExceptions
Divide into 2 steps

1. Identify the boxing and unboxing locations in the source code

2. Handle them
Divide into 2 steps

1. Identify the boxing and unboxing locations in the source code

2. Handle them
Divide into 2 steps

1. Identify the boxing and unboxing locations in the source code
   - The assignment rule is too generous.

2. Handle them
   - Can be described pretty accurately by taclets.
Borrowing from type theory

Quantified types

In type theory there exist existential and universal types:

\[
\text{int list} <: (\exists \alpha. \alpha \text{ list}) \\
\quad (\forall \alpha. \alpha \to \alpha) <: \text{int} \to \text{int}
\]
## Borrowing from type theory

### Quantified types

In type theory there exist existential and universal types:

\[
\text{int list} <: (\exists \alpha. \alpha \text{ list}) \\
(\forall \alpha. \alpha \rightarrow \alpha) <: \text{int} \rightarrow \text{int}
\]

### Similar ideas in JavaDL

Allow the creation of type variables and quantification over them.

\[
\exists X. \text{object} \in \text{Vector}(X)
\]