# Software Verification for Java 5

## KeY Symposium 2007

Mattias Ulbrich

June 14, 2007

# Content

KeY + Java 5

Typesafe Enumeration Datatypes

Enhanced For Loops

Generic Classes

# Why KeY + Java 5 ?

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

$$\textbf{enum}\ \mathsf{E}\ \{\ \mathsf{e}_1,\ \mathsf{e}_2,\ \ldots,\ \mathsf{e}_n\ \}$$

- 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)

$\Longrightarrow$ Use the mechanisms available for reference types.

### The object repository $C{::}\langle\mathrm{get}\rangle() : \mathrm{Nat} \rightarrowtail\!\!\!\!\twoheadrightarrow C$

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

# Using the object repository

Enumerations are reference types (special classes in fact)

$\Longrightarrow$ Use the mechanisms available for reference types.

## The object repository $C{::}\langle\mathrm{get}\rangle() : \mathrm{Nat} \rightarrowtail\!\!\!\!\twoheadrightarrow C$

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

## Repository access for Enums:

$$\begin{array}{rcl} E.e_1 & \doteq & E{::}\langle\mathrm{get}\rangle(0) \\ E.e_2 & \doteq & E{::}\langle\mathrm{get}\rangle(1) \\ & \ldots & \\ E.e_n & \doteq & E{::}\langle\mathrm{get}\rangle(n-1) \\ E{::}\langle\mathrm{nextToCreate}\rangle & \doteq & n \end{array}$$

# 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.

# 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

```
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

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

## Java 5

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

# Equivalent loops

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

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

# Equivalent loops

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

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

$$\text{enhForArrayInv}\ \frac{\begin{array}{l}\textbf{Null Case}\\ \textbf{Base Case}\\ \textbf{Abnormal body termination}\\ \textbf{Invariant preserved}\\ \textbf{Use Case}\end{array}}{\Gamma \vdash \mathcal{U}\ \boxed{\langle}\texttt{for(}\mathit{ty}\,x\ : \mathit{se}\texttt{)\{}\ p\ \texttt{\}}\boxed{\rangle}\varphi,\ \Delta}$$

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.

| | **new rule** | **while rule** |
|---|---|---|
| Nodes in the proof tree | 374 | 1053 |
| Branches in the proof tree | 8 | 21 |
| Additional manual instantiations | 2 | 3 |

$\Longrightarrow$ 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

```
Vector v = new Vector();

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

### Java 5

```
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

### Traditional Java

```
Vector v = new Vector();

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

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

### Java 5

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

- 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

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

$head : Chain \longrightarrow Object$

# Polymorphic functions

## Attributes induce functions

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

$head : Chain \rightarrow Object$

## Polymorphic attributes induce polymorphic functions

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

$head : \forall T.Chain\langle T\rangle \rightarrow T$

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

# Infinite type system

## "Parametric recursion"

String

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<Vector<String>>>...>

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

# Infinite type system

### "Parametric recursion"

Vector<...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”

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

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

## Problem

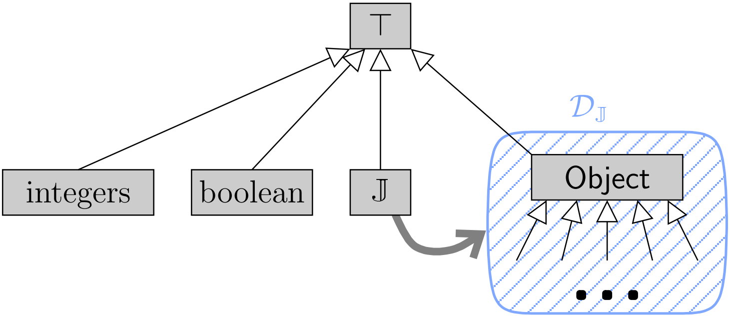
Some rules need a finite type system to enumerate types (method dispatch, dynamic subtypes, . . . )

## Handle this in JavaDL ...

... with existentially quantified type variables

$$\exists X.\ \mathit{object} \sqsubseteq_1 \mathrm{Vector}\langle X\rangle$$

# 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

$\Longrightarrow$ Allow quantification over types class

# Generic contracts

## Method contracts

Given a pre-condition *pre* prior to a method call `o.m()`, a post-condition *post* holds afterwards:

$$\mathit{pre} \rightarrow \langle \texttt{o.m();} \rangle \mathit{post}$$

# Generic contracts

## Method contracts

Given a pre-condition *pre* prior to a method call `o.m()`, a post-condition *post* holds afterwards:

$$\mathit{pre} \rightarrow \langle \texttt{o.m();} \rangle \mathit{post}$$

## Generic method contracts

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

$$\forall T : \mathbb{J}.\ \mathit{pre}(T) \rightarrow \langle \texttt{o.m();} \rangle \mathit{post}(T)$$

# 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 $\mathbb{J}$.
- Allow quantification over types ...
- ... and instantiations
- generic attributes lead to polymorphic functions in the logic.

$\Longrightarrow$ 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 ...

| **Feature** | **Needs Verif.** | **Supports Verif.** | **Fits** |
|---|---|---|---|
| Enums | YES | YES | YES |
| Enh. For | YES | YES | YES |
| Boxing | YES | NO | NO |
| Generics | NO* | YES | **NO** |

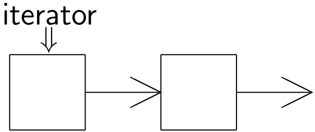
ThanK e You !

# 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.

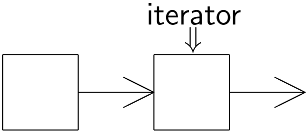


# Non-termination if iterating a collection

## 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.

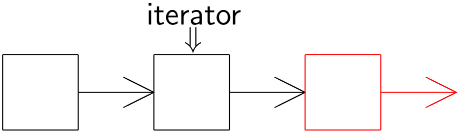


# Non-termination if iterating a collection

## Nicht 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.

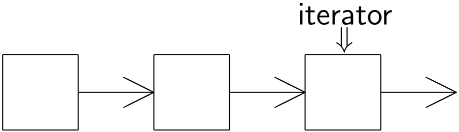


# Non-termination if iterating a collection

## 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.

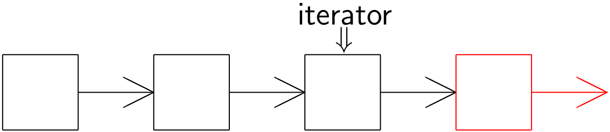


# Non-termination if iterating a collection

### Nicht 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.

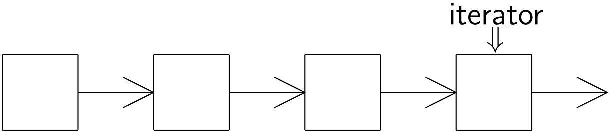


# Non-termination if iterating a collection

## 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.

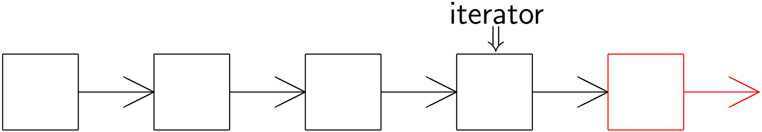


# Non-termination if iterating a collection

### Nicht 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.

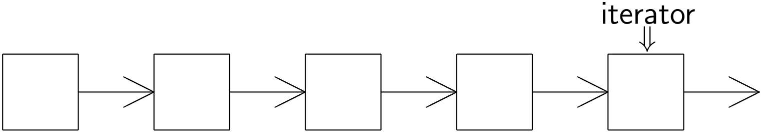


# Non-termination if iterating a collection

### 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.

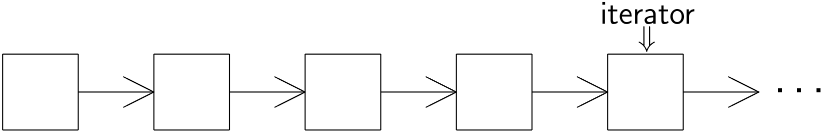


# Non-termination if iterating a collection

## Nicht uebertragbar

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

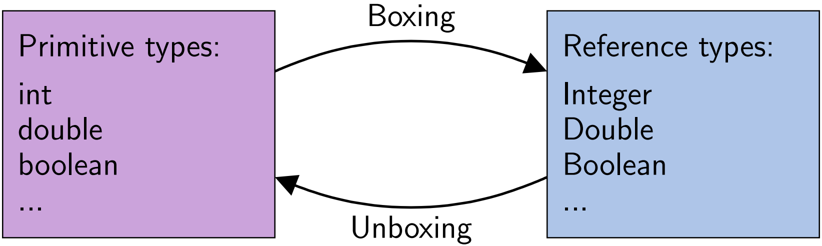
Consider a singly-chained list that is iterated and appended to at the same time: The iteration process will not terminate.



# Auto-Boxing and Unboxing

**Idea**

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



Primitive types:

int
double
boolean
...

Boxing

Unboxing

Reference types:

Integer
Double
Boolean
...

# 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

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

### Auto-boxing in Java 5

```
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:

$$\begin{aligned} \text{int list} \quad &<: \quad (\exists \alpha . \alpha \ \text{list}) \\ (\forall \alpha . \alpha \to \alpha) \quad &<: \quad \text{int} \to \text{int} \end{aligned}$$

# Borrowing from type theory

## Quantified types

In type theory there exist existential and universal types:

$$\begin{aligned} \text{int list} &<: (\exists \alpha . \alpha \text{ list}) \\ (\forall \alpha . \alpha \rightarrow \alpha) &<: \text{int} \rightarrow \text{int} \end{aligned}$$

## Similar ideas in JavaDL

Allow the creation of type variables and quantification over them.

$$\exists X . \mathit{object} \sqsubseteq \text{Vector}\langle X \rangle$$