



IBM Research

Efficiently Refactoring Java Applications to use Generic Libraries

Frank Tip

IBM T.J. Watson Research Center

joint work with:

Robert Fuhrer

IBM T.J. Watson Research Center

Adam Kiezun

MIT

Julian Dolby

IBM T.J. Watson Research Center

Markus Keller

IBM Research, Zurich

Motivation

- **Java 1.5 **generics** enable the creation of type-safe, reusable collection class libraries**
- **the Java 1.5 standard libraries contain a generic version of the existing Java Collections Framework in package `java.util`**
 - e.g., `Vector<E>`, `HashMap<K,V>`
 - existing applications still compile, but with warnings due to type-unsafe usage
 - manually rewriting existing applications is tedious & error-prone
 - e.g., xtc, a 90,000 line Java 1.4 program from Purdue contains 668 declarations and 330 allocation sites that refer to the standard collections; compiler produces 583 warnings related to the use of “raw” references to generic types
- **goal: automated migration support**

Outline

- motivation
- **review of Java 1.5 generics**
- **the algorithm (by example; some simplifications)**
 - generation of type constraints
 - constraint solving
 - source rewriting
- **results**
- **related work**
- **conclusions & future work**

Quiz on Inheritance and Generics

Is the following code snippet legal?

```
List<String> ls =  
    new ArrayList<String>(); //1  
List<Object> lo = ls; //2
```

Quiz on Inheritance and Generics

Line 1 is certainly legal.

```
List<String> ls =  
    new ArrayList<String>(); //1
```

The trickier part of the question is line 2.

```
List<Object> lo = ls; //2
```

Is a List of String a List of Object?

“YES?”

Quiz on Inheritance and Generics

Well, take a look at the next few lines:

```
lo.add(new Object()); // 3  
String s = ls.get(0); // 4:  
    //attempts to assign an Object  
    //to a String!
```

Generics and Subtyping

- In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is **not** the case that G<Foo> is a subtype of G<Bar>.

Java 1.5 Generics

- a class or interface **C** may have formal type parameters T_1, \dots, T_n
 - notation: **C**< T_1, \dots, T_n >
 - example: `class MyList<E> { ... }`
 - the formal type parameters T_j can be used in non-static declarations in C
- formal type parameter T_j has one or more bounds B_j^1, \dots, B_j^k , at most one of which can be a class
 - notation: **T_j extends B_j^1 & ... & B_j^k**
 - example: `class X<T extends Number> { }`
- instantiating a generic class **C**< T_1, \dots, T_n > requires that n actual type parameters A_1, \dots, A_n be supplied
 - each actual type parameter A_j must satisfy all bounds of the corresponding formal type parameter T_j
 - example: `new X<Float>()`

Java Generics: Inheritance

- a class may inherit from a parameterized class, and use its formal type parameters to select a specific parameterization

```
class B<T1 extends Number> { ... }
```

```
class C<T2 extends Number> extends B<T2> { ... }
```

```
class D extends B<Integer> { ... }
```

```
B<Float> x = new C<Float>();
```

```
B<Integer> y = new D();
```

- **some limitations**
 - arrays of generic types (e.g., `B<Float>[]`) are not allowed
 - unlike arrays, generic types are not covariant:
 - `C<A>` is a subtype of `C` if and only if `A = B`

Reduced Need for Downcasts

using current collections

```
class Vector {
    public Object get(int);
    public boolean add(Object);
    ...
}

void client(){
    Vector v =
        new Vector();
    v.add("abc");
    ...
    String s = (String)v.get(0);
}
```

using generic collections

```
class Vector<T> {
    public T get(int);
    public boolean add(T);
    ...
}

void client(){
    Vector<String> v =
        new Vector<String>();
    v.add("abc");
    ...
    String s = v.get(0);
}
```

Java Generics: Generic Methods

- **formal type parameters of generic methods declared at beginning of signature, after qualifiers**

```
public static <T> void foo(T arg){ ... }
```

- **inheritance relations between type parameters can be useful:**

```
class MyList<E1> {  
    public <E2 extends E1> void addOther(MyList<E2> l){ ... }  
}
```

```
MyList<Number> list1 = new MyList<Number>();  
MyList<Float> list2 = new MyList<Float>();  
list1.addOther(list2);
```

Wildcards

Consider the problem of writing a routine that prints out all the elements in a collection.

Here's how you might write it in an older version of the language:

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    for (k = 0; k < c.size(); k++)  
        System.out.println(i.next());  
}
```

Wildcards

And here is a naive attempt at writing it using generics

```
void printCollection(Collection<Object> c) {  
    for (Object e : c)  
        System.out.println(e);  
}
```

Wildcards

- The problem is that this new version is much less useful than the old one.
- Whereas the old code could be called with any kind of collection as a parameter, the new code only takes `Collection<Object>`
 - which, as we've just demonstrated, is *not* a supertype of all kinds of collections!

Wildcards

So what *is* the supertype of all kinds of collections?

`Collection<?>`

“collection of unknown”

```
void printCollection(Collection<?> c) {  
    for (Object e : c)  
        System.out.println(e);  
}
```

Wildcards

- *Now, we can call it with any type of collection.*
- Notice that inside `printCollection()`, we can still read elements from `c` and give them type `Object`.

```
void printCollection(Collection<?> c) {  
    for (Object e : c)  
        System.out.println(e);  
}
```

Wildcards

- This is always safe since whatever the actual type of the collection, it does contain objects.
- It isn't safe to add arbitrary objects to it **however:**

```
Collection<?> c = new ArrayList<String>();  
c.add(new Object()); // compile time error
```

Java Generics: Wildcards

- **wildcards** ~ unnamed formal type parameters; convenient to avoid syntactic clutter in method signatures
 - wildcards may have upper bounds or lower bounds
 - can be used in cases where named type parameters cannot be used (e.g., declarations of local variables, fields)

```
public class Collections {
    public static <T> void copy(List<? super T> dest,
                               List<? extends T> src) {
        for (int i=0; i<src.size(); i++)
            dest.set(i,src.get(i));
    }
}
...
List<Object> output = new ArrayList<Object>();
List<Long>     input = new ArrayList<Long>();
Collections.copy(output,input);
```


OK, use, e.g., T = Number

example taken from <http://www.langer.camelot.de/>

Java Generics: Raw Types

- **raw types**: reference to a generic class without specifying its parameterization
 - introduced for compatibility reasons
 - semantics: instantiate each type parameter with its bounds
 - compiler issues warning when encountering certain uses of raw types

```
Vector v = new Vector();  
v.add("hello");
```



Eclipse 3.1M3 compiler:

Type safety: The method `add(Object)` belongs to the raw type `Vector`. References to generic type `Vector<E>` should be parameterized

Java 1.5 Standard Collections Framework

- **in Java 1.5, the Standard Collections Framework has been made generic**
 - reduces the use for down-casts when retrieving elements
- **Collections and Maps**
 - `Collection<E>` with subclasses `List<E>`, `Vector<E>`, ...
 - `Map<K, V>` with subclasses `TreeMap<K, V>`, `HashMap<K, V>`, ...
- **Iterators and Enumerations**
 - `Iterator<E>` and `Enumeration<E>`
- **functionality for comparisons**
 - `Comparable<T>` and `Comparator<T>`
- **various generic helper methods in class `Collections` such as**

```
<T extends Comparable<? super T>>
    void Collections.sort(List<T> list)
```
- ***existing (Java 1.4) applications can still be compiled (using raw types)!***
 - *but lots of compiler warnings*

Example

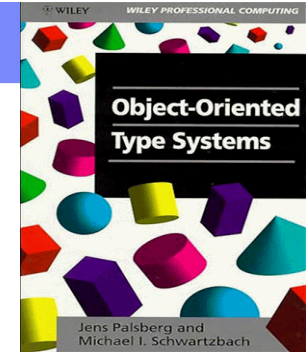
```
public class Test {
    public static void main(String[] args){
        Vector v1 = new Vector();
        List v2 = new ArrayList();
        v1.add(new Integer(17));
        v2.add(new Float(3.3));
        v1.addAll(v2);
        Integer i1 = (Integer)v1.get(0);
        for (Iterator it= new RevIterator(v2); it.hasNext(); ){
            Float f1 = (Float)it.next();
        }
    }
}

class RevIterator implements Iterator {
    RevIterator(List v3){ v4 = v3; count = v3.size(); }
    public boolean hasNext(){ return count > 0; }
    public Object next(){ count--; return v4.get(count); }
    private List v4; private int count;
}
```

Example

```
public class Test {
    public static void main(String[] args){
        Vector<Number> v1 = new Vector<Number>();
        List<Float> v2 = new ArrayList<Float>();
        v1.add(new Integer(17));
        v2.add(new Float(3.3));
        v1.addAll(v2);
        Integer i1 = (Integer)v1.get(0);
        for (Iterator<Float> it= new RevIterator(v2); it.hasNext(); ){
            Float f1 = (Float) it.next();
        }
    }
}

class RevIterator implements Iterator<Float> {
    RevIterator(List<Float> v3){ v4 = v3; count = v3.size(); }
    public boolean hasNext(){ return count > 0; }
    public Float next(){ count--; return v4.get(count); }
    private List<Float> v4; private int count;
}
```



Overview of Approach

- **generate type constraints (Palsberg & Schwartzbach 93)**
 - generate standard type constraints from program constructs
 - generate additional type constraints for calls to library methods, and for overriding of library methods
- **solve the system of type constraints**
 - solution consists of types to be used as actual type parameters in allocation sites and declarations that refer to library classes
 - types also inferred for all other declarations (in most but not all cases, same as in original program)
- **source rewriting**

Type Constraints Notation

[E]	the type of expression E
[M]	the declared return type of method M
[F]	the declared type of field F
K(E)	the type bound to formal type parameter K in the parametric type of E
T	type constant
Decl(M)	the type that contains method M
Param(M,i)	the i -th parameter of method M
≤	subtype relation
α, α', ...	constraint variables. these are of the forms [E], [M], [F], K(E), K(T), T

Syntax of Type Constraints

$\alpha = \alpha'$ **type α must be the same as type α'**

$\alpha \leq \alpha'$ **type α must be the same as,
or a subtype of type α'**

$\alpha \leq \alpha_1$ or ... or $\alpha \leq \alpha_k$

disjunction: at least one of sub-constraints

$\alpha \leq \alpha_1, \dots, \alpha \leq \alpha_k$ **must hold**

Constraints on Virtual Calls

$RootDefs(M) =$

$\{ Decl(M') \mid M \text{ overrides } M',$

and there exists no M''

$(M'' \neq M')$

such that M' overrides $M'' \}$

In English (as much as possible):

$RoofDef(M)$ is the nodes above $Decl(M)$ that M overrides a method, but that method, itself is not overridden.

Constraints on Virtual Calls

If P contains call $E_0.m(E_1, \dots, E_k)$ to virtual method M and $RootDefs(M) = \{C_1; \dots; C_q\}$

then

$$[E_0] \leq C_1 \text{ or } \dots \text{ or } [E_0] \leq C_q$$

In English?

Constraints on Virtual Calls

- states that a declaration of a method with the same signature as M must occur in some supertype of the type of E_0 .
- The complexity in this rule stems from the fact that M may override one or more methods $M_1 \dots M_q$ declared in supertypes C_1, \dots, C_q of $Decl(M)$, and the type-correctness of the method call only requires that the type of receiver expression E_0 be a subtype of one of these C_j .

Some Basic Type Constraint Generation Rules

assignment $E_1 = E_2$	$[E_2] \leq [E_1]$
access $E.f$ to field F	$[E.f] = [F]$ $[E] \leq \text{Decl}(F)$
return E in method M	$[E] \leq [M]$
call $\text{new } C(E_1, \dots, E_n)$ to constructor M	$[\text{new } C(E_1, \dots, E_n)] = C$ $[E_i] \leq [\text{Param}(M, i)]$
direct call $E.m(E_1, \dots, E_n)$ to method M	$[E.m(E_1, \dots, E_n)] = [M]$ $[E_i] \leq [\text{Param}(M, i)]$ $[E] \leq \text{Decl}(M)$

```

public class Test {
    public static void main(String[] args){
        Vector v1 = new Vector();
        List v2 = new ArrayList();
        v1.add(new Integer(17));
        v2.add(new Float(3.3));
        v1.addAll(v2);
        Integer i1 = (Integer)v1.get(0);
        for (Iterator it= new RevIterator(v2); it.hasNext(); ){
            Float f1 = (Float)it.next();
        }
    }
}

```

$|new ArrayList()| \leq |v2|$
 $|new ArrayList()| = ArrayList$
 $|new Float(3.3)| = Float$

$|v2| \leq |v3|$
 $|new RevIterator(v2)| \leq |it|$

```

class RevIterator implements Iterator {
    RevIterator(List v3){ v4 = v3; count = v3.size(); }
    public boolean hasNext(){ return count > 0; }
    public Object next(){ count--; return v4.get(count); }
    private List v4; private int count;
}

```

$|v4.get(count)| \leq |RevIterator.next()|$

Constraint Generation for Calls to Library Methods

call E.get(..) to T List<T>.get(int)	 E.get(..) = T(E)
call E₀.add(E₁) to boolean List<T>.add(T)	[E₁] ≤ T(E₀)
call E₀.addAll(E₁) to boolean Collection<T>.addAll(Collection<? extends T>)	T(E₁) ≤ T(E₀)

In general, ≥ 1 rules for each call to library method:

- these rules are inferred from the method signatures
- needs to be done only once for each class library

```

public class Test {
    public static void main(String[] args){
        Vector v1 = new Vector();
        List v2 = new ArrayList();
        v1.add(new Integer(17));
        v2.add(new Float(3.3));
        v1.addAll(v2);
        Integer i1 = (Integer)v1.get(0);
        for (Iterator it= new RevIterator(v2); it.hasNext(); ){
            Float f1 = (Float)it.next();
        }
    }
}

class RevIterator implements Iterator {
    RevIterator(List v3){ v4 = v3; count = v3.size(); }
    public boolean hasNext(){ return count > 0; }
    public Object next(){ count--; return v4.get(count); }
    private List v4; private int count;
}

```

$|\text{new Float}(3.3)| \leq T(v2)$
 $T(v2) \leq T(v1)$

$|\text{v4.get(count)}| = T(v4)$

```

public class Test {
    public static void main(String[]
        Vector v1 = new Vector();
        List v2 = new ArrayList();
        v1.add(new Integer(17));
        v2.add(new Float(3.3));
        v1.addAll(v2);
        Integer i1 = (Integer)v1.get
        for (Iterator it= new RevIt
            Float f1 = (Float)it
        }
    }
}

class RevIterator implements Iterat
    RevIterator(List v3){ v4 = v3; co
    public boolean hasNext(){ return
    public Object next(){ count--; re
    private List v4; private int coun
}

```

```

|new Vector()| ≤ |v1|
|v1.get(0)| = T(v1)
|new ArrayList()| ≤ |v2|
Integer ≤ T(v1)
Float ≤ T(v2)
T(v2) ≤ T(v1)
Integer ≤ |v1.get(0)|
|v2| ≤ |v3|
|new RevIterator(v2)| ≤ |it|
Float ≤ |it.next()|
|it.next()| = T(it)
|v3| ≤ |v4|
|v4.get(count)| = T(v4)
|v4.get(count)| ≤ |RevIterator.next()|
|RevIterator.next()| = T(RevIterator)
T(new RevIterator(v2)) = T(RevIterator)
T(new RevIterator(v2)) = T(it)
T(new Vector()) = T(v1)
T(new ArrayList()) = T(v2)
T(v2) = T(v3)
T(v3) = T(v4)

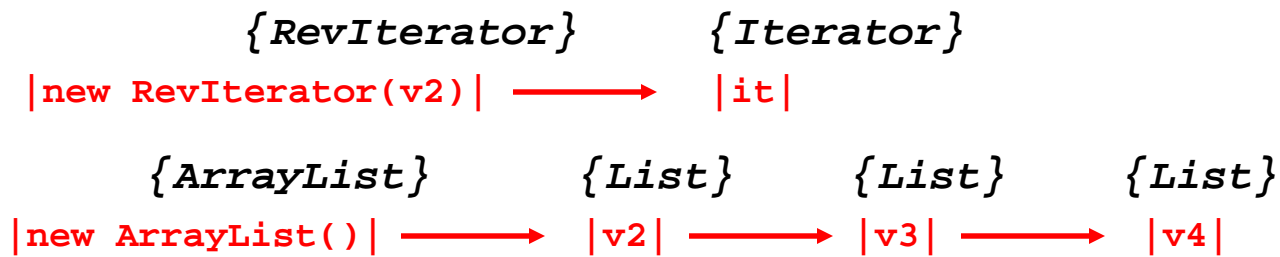
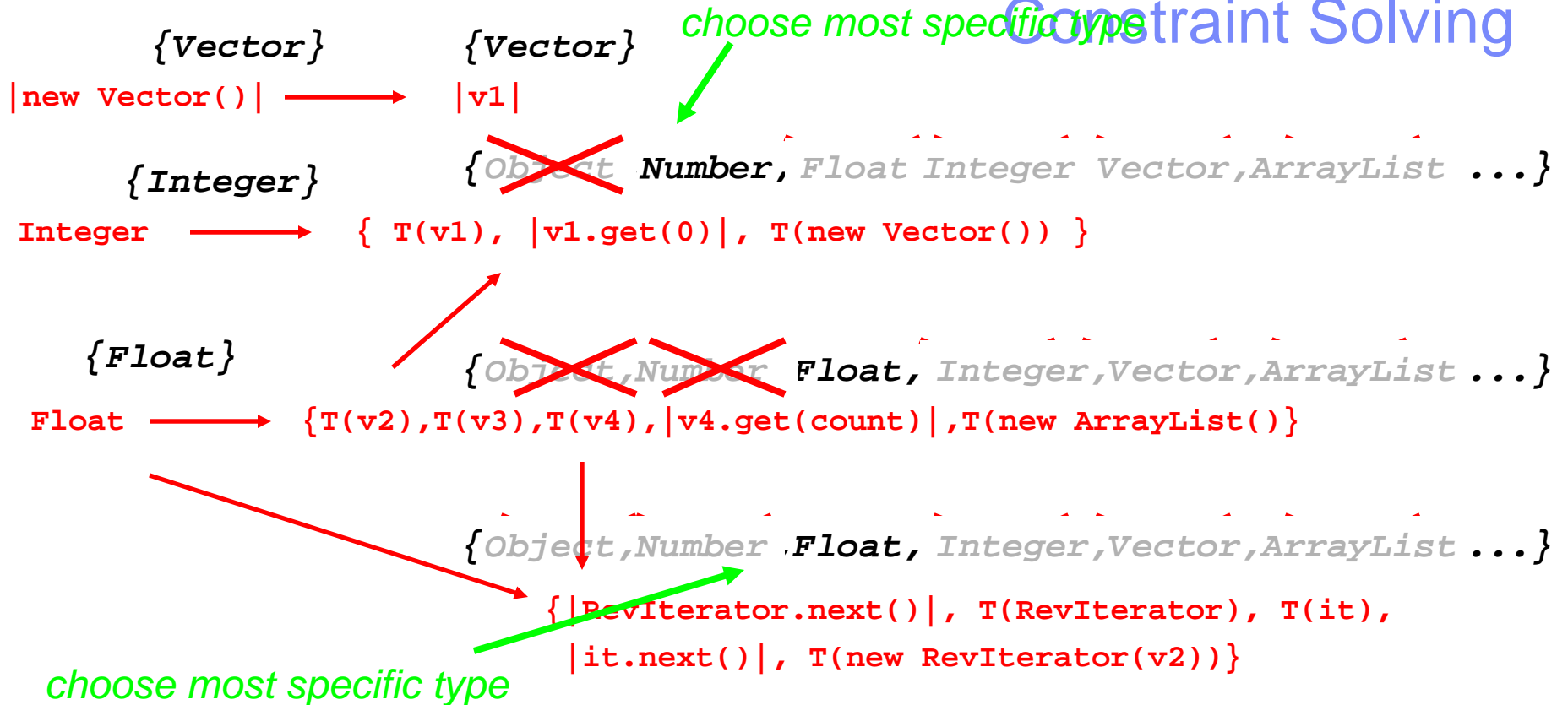
```

Constraint Solving

- **collapse cycles implied by \leq and $=$ constraints**
- **initial type estimate for each constraint variable:**
 - singleton set containing original type (for constants, type literals, constructor calls, ...)*
 - singleton set containing original type for variables of library type
 - the “universe of all types” for anything else
- **iterative solving step: for each type constraint $\alpha \leq \alpha'$**
 - remove from $\text{TypeEst}(\alpha)$ any element that is not a subtype of at least one element in $\text{TypeEst}(\alpha')$
 - remove from $\text{TypeEst}(\alpha')$ any element that is not a supertype of at least one element in $\text{TypeEst}(\alpha)$
- **the constraint system is usually underconstrained**
 - non-singleton type estimates left when solver has completed
 - in case of completely unconstrained type estimates, choose original type
 - in other cases, make choice, then run solver to propagate this choice (prefer more specific types and visible types; ignore useless types like Serializable whenever possible)
- **result: singleton type sets for all constraint variables**

*also used to preserve “rawness” when application exchanges objects with external libraries

Constraint Solving



Source Rewriting

1. rewrite declaration/allocation site **E** for which we inferred $|E| = C$ and $T(E) = C'$ to $C\langle C' \rangle$
2. remove any cast $(C)E$ for which we inferred $|E| \leq C$
3. add import statements as required

Result

```
public class Test {
    public static void main(String[] args){
        Vector<Number> v1 = new Vector<Number>();
        List<Float> v2 = new ArrayList<Float>();
        v1.add(new Integer(17));
        v2.add(new Float(3.3));
        v1.addAll(v2);
        Integer i1 = (Integer)v1.get(0);
        for (Iterator<Float> it= new RevIterator(v2); it.hasNext(); ){
            Float f1 = (Float) it.next();
        }
    }
}

class RevIterator implements Iterator<Float> {
    RevIterator(List<Float> v3){ v4 = v3; count = v3.size(); }
    public boolean hasNext(){ return count > 0; }
    public Float next(){ count--; return v4.get(count); }
    private List<Float> v4; private int count;
}
```

Implementation in Eclipse

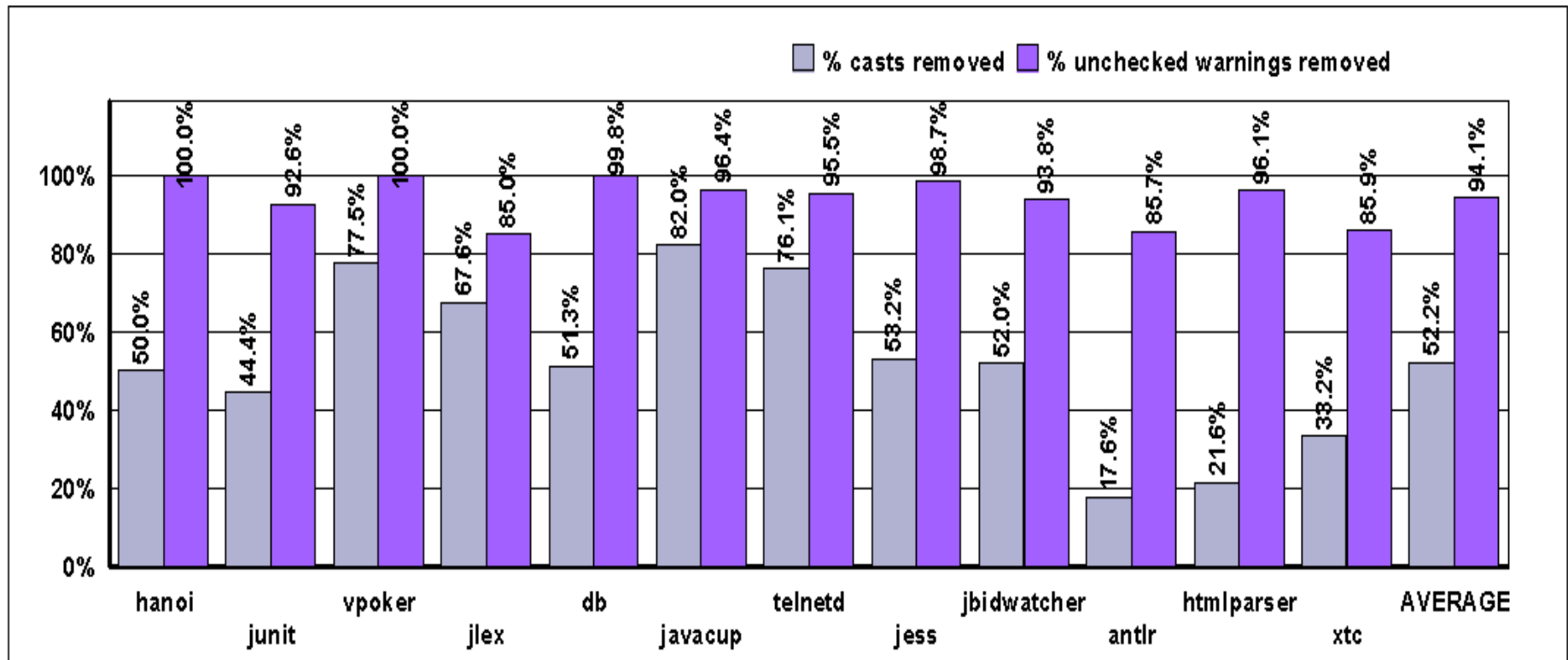
- **optimizations to achieve scalability**
 - custom type hierarchy representation to answer subtype-queries in constant time
 - unification of identically constrained type estimates
 - implicit representation for sets of types (subtypesof(C), supertypesof(C), universe, ...) with algebraic simplifications
- **dealing with raw types**
 - leave type raw when we infer an actual type parameter of Object
 - leave type raw when application calls methods in library classes that operate on raw Vectors
 - leave type raw when a bound is completely unconstrained
 - force rawness when encountering arrays of generic types (by constraining the type parameter to be Object)
- **miscellaneous issues**
 - infinite nesting possible; e.g., `Vector v = new Vector(); v.add(v);`
 - sometimes we find that a cast always fails at run-time; transform into code with a compile-time error

Benchmarks

benchmark	benchmark size							generics-related measures				
	types	methods	LOC	NBNC	LOC	decls	allocs	casts	allocs	decls	subtypes	warnings
hanoi	41	320	4,056		2,049	967	68	20	3	3	0	3
junit	59	382	5,265		2,394	1,012	305	54	24	48	0	27
vpoker	35	279	6,351		3,097	1,044	198	40	12	27	1	47
jlex	22	121	7,842		4,333	668	146	71	17	33	1	40
db	32	222	8,594		3,363	939	225	78	14	36	1	652
javacup	36	302	11,087		3,833	1,065	341	595	19	62	0	55
telnetd	52	397	11,239		3,219	995	128	46	16	28	0	22
jess	184	756	18,199		7,629	2,608	654	156	47	64	1	692
jbidwatcher	264	1,830	38,571		21,226	5,818	1,698	383	76	184	1	195
antlr	207	2,089	47,685		28,599	6,175	1,163	443	46	106	3	84
htmlparser	232	1,957	50,799		20,332	4,895	1,668	793	72	136	2	205
xtc	1,556	5,564	90,565		37,792	14,672	3,994	1,114	330	668	1	583

- benchmarks available in the public domain
- migration to the Java 1.5 generic standard collections framework

Results



- processing “xtc” takes 88.5 seconds (on 1.6Ghz Pentium M, with <500 Mb heap space)
- most of the smaller benchmarks are processed in <10 seconds
- results validated using javac 1.5 & by running the transformed benchmarks where possible
- manual inspection reveals that most of the remaining casts are not related to containers

Related Work

- **similar refactoring by Donovan, Kiezun, Tschantz, Ernst (OOPSLA'04)**
 - aims at stricter form of preserving behavior (preserving erasure)
 - conforms to pre-1.5 version of generics (that allows generic arrays)
 - based on byte-code analysis and requires modified version of javac compiler
 - less scalable because it involves context-sensitive alias analysis and backtracking
 - not capable of inferring parameterized supertypes for subtypes of generic library classes (e.g., MyIterator implements Iterator<String>)
 - similar precision (though incomparable results); we seem to be removing slightly more casts
- **inferring the parameterization of classes by Von Dincklage & Diwan (OOPSLA'04)**
 - largely heuristics-based, requires manual modification of program source, unsound
- **“generify” refactorings incorporated in other IDEs (IDEA, CodeGuide, ...)**
 - not aware of implementation details
- **previous applications of type constraints in (refactoring) tools**
 - refactoring for generalization (Tip, Kiezun, Baeumer OOPSLA'03)
 - customization of library classes (De Sutter, Tip, Dolby ECOOP'04)
 - more work in progress...

Conclusions

- **effective technique for migrating Java applications to a generic version of a class library**
 1. generate library-specific constraint generation-rules from the generic API of the library (to be done only once for each library)
 2. for a given application, generate constraints using standard constraint generation rules for various program constructs and the additional constraint generation rules obtained in step 1
 3. constraint solving
 4. update source code (update allocation sites and declarations, remove casts)
- **evaluated by migrating a number of benchmarks to the Java 1.5 Collections Framework**
 - success evident from casts removed, reduction in # warnings
 - implementation appears to scale well
- **to be released in Eclipse 3.1 in June 2005**