Efficiently Refactoring Java Applications to Use Generic Libraries

Robert Fuhrer[1], Frank Tip[1], Adam Kieżun[2], Julian Dolby[1], and Markus Keller[3]

[1] IBM T.J. Watson Research Center, P.O.Box 704, Yorktown Heights, NY 10598, USA
{rfuhrer,ftip,dolby}@us.ibm.com
[2] MIT Computer Science & AI Lab, 32 Vassar St, Cambridge, MA 02139, USA
akiezun@mit.edu
[3] IBM Research, Oberdorfstrasse 8, CH-8001 Zürich, Switzerland
markus_keller@ch.ibm.com

Abstract. Java 1.5 generics enable the creation of reusable container classes with compiler-enforced type-safe usage. This eliminates the need for potentially unsafe down-casts when retrieving elements from containers. We present a *refactoring* that replaces raw references to generic library classes with parameterized references. The refactoring infers actual type parameters for allocation sites and declarations using an existing framework of type constraints, and removes casts that have been rendered redundant. The refactoring was implemented in Eclipse, a popular open-source development environment for Java, and laid the grounds for a similar refactoring in the forthcoming Eclipse 3.1 release. We evaluated our work by refactoring several Java programs that use the standard collections framework to use Java 1.5's generic version instead. In these benchmarks, on average, 48.6% of the casts are removed, and 91.2% of the compiler warnings related to the use of raw types are eliminated. Our approach distinguishes itself from the state-of-the-art [8] by being more scalable, by its ability to accommodate user-defined subtypes of generic library classes, and by being incorporated in a popular integrated development environment.

1 Introduction

Java 1.5 generics enable the creation of reusable class libraries with compiler-enforced type-safe usage. Generics are particularly useful for building homogeneous collections of elements that can be used in different contexts. Since the element type of each generic collection instance is explicitly specified, the compiler can statically check each access, and the need for potentially unsafe user-supplied downcasts at element retrieval sites is greatly reduced. Java's standard collections framework in package `java.util` undoubtedly provides the most compelling uses of generics. For Java 1.5, this framework was modified to include generic versions of existing container classes[4] such as `Vector`. For example, an application that instantiates `Vector<E>` with, say, `String`, obtaining `Vector<String>`, can only add and retrieve `String`s. In the previous, non-generic version of this class, the signatures of access methods such as `Vector.get()` refer to type `Object`, which prevents the compiler from ensuring

[4] For convenience, the word "class" will frequently be used to refer to a class or an interface.

type-safety of vector operations, and therefore down-casts to `String` are needed to recover the type of retrieved elements. When containers are misused, such downcasts fail at runtime, with `ClassCastException`s.

The premise of this research is that, now that generics are available, programmers will want to *refactor* [10] their applications to replace references to non-generic library classes with references to generic versions of those classes, but performing this transformation manually on large applications would be tedious and error-prone [15]. Therefore, we present a refactoring algorithm for determining the actual type parameters with which occurrences of generic library classes can be instantiated[5]. This refactoring rewrites declarations and allocation sites to specify actual type parameters that are inferred by type inference, and removes casts that have been rendered redundant. Program behavior is preserved in the sense that the resulting program is type-correct and the behavior of operations involving run-time types (i.e., method dispatch, casts, and instanceof tests) is preserved. Our approach is applicable to any class library for which a generic equivalent is available, but we will primarily use the standard collections framework to illustrate the approach.

Our algorithm was implemented in Eclipse (see `www.eclipse.org`), a popular open-source integrated development environment (IDE), and parts of this research implementation will be shipped with the forthcoming Eclipse 3.1 release. We evaluated the refactoring on a number of Java programs of up to 90,565 lines, by refactoring these to use Java 1.5's generic container classes. We measured the effectiveness of the refactoring by counting the number of removed downcasts and by measuring the reduction in the number of "unchecked warnings" issued by the Java 1.5 compiler. Such warnings are issued by the compiler upon encountering raw occurrences of generic classes (i.e., references to generic types without explicitly specified actual type parameters). In the benchmarks we analyzed, on average, 48.6% of all casts are removed, and 91.2% of the unchecked warnings are eliminated. Manual inspection of the results revealed that the majority of casts caused by the use of non-generic containers were removed by our refactoring, and that the remaining casts were necessary for other reasons. The refactoring scales well, and takes less than 2 minutes on the largest benchmark.

The precision of our algorithm is comparable to that by Donovan et al. [8], which is the state-of-the-art in the area, but significant differences exist between the two approaches. First, our approach is more scalable because it does not require context-sensitive analysis. Second, our method can infer generic supertypes for user-defined subtypes of generic library classes[6] (e.g., we can infer that a class `MyIterator` extends `Iterator<String>`). The approach of [8] is incapable of making such inferences, and therefore removes fewer casts on several of the benchmarks we analyzed. Third, Donovan et al. employ a strict notion of preserving behavior by demanding that the program's erasure [4] is preserved. In addition to this mode, our tool supports more relaxed notions of preserving behavior that allow the rewriting of other declarations. Our experiments show that, in some cases, this added flexibility enables the removal of more casts and unchecked warnings. Fourth, our implementation is more practical

[5] This problem is referred to as the instantiation problem in [8].

[6] The version of our refactoring that will be delivered in Eclipse 3.1 will not infer generic supertypes and will always preserve erasure.

because it operates on standard Java 1.5 source code, and because it is fully integrated in a popular IDE.

The remainder of the paper is organized as follows. Section 2 overviews the Java 1.5 generics, and Section 3 presents a motivating example to illustrate our refactoring. Sections 4–6 present the algorithm, which consists of the following steps. First, a set of *type constraints* [17] is inferred from the original program's abstract syntax tree (AST) using two sets of generation rules: (i) standard rules that are presented in Section 4 and (ii) generics-related rules that are presented in Section 5. Then, the resulting system of constraints is solved, the program's source code is updated to reflect the inferred actual type parameters, and redundant casts are removed, as discussed in Section 6. Section 7 discusses the implementation of our algorithm in Eclipse, and experimental results are reported in Section 8. We report on experiments with a context-sensitive version of our algorithm in Section 9. Finally, related work and conclusions are discussed in Sections 10 and 11, respectively.

2 Java Generics

This section presents a brief, informal discussion of Java generics. For more details, the reader is referred to the Java Language Specification [4], and to earlier work on the Pizza [16] and GJ [5, 13] languages.

In Java 1.5, a class or interface C may have *formal type parameters* $T_1, \cdots, T_n$ that can be used in non-`static` declarations within C. Type parameter T_j may be *bounded* by types $B_j^1, \cdots, B_j^k$, at most one of which may be a class. Instantiating a generic class $C<T_1, \cdots, T_n>$ requires that n *actual type parameters* $A_1, \cdots, A_n$ be supplied, where each A_j must satisfy the bounds (if any) of the corresponding formal type parameter T_j. Syntactically, (formal and actual) type parameters follow the class name in a comma-separated list between '$<$' and '$>$', and bounds on formal type parameters are specified using the keyword `extends` (multiple bounds are separated by '`&`'). A class may inherit from a parameterized class, and its formal type parameters may be used as actual type parameters in instantiating its superclass. For example:

```
class B<T1 extends Number>{ ... }
class C<T2 extends Number> extends B<T2>{ ... }
class D extends B<Integer>{ ... }
```

shows: (i) a class `B` that has a formal type parameter `T1` with an upper bound of `Number`, (ii) a class `C` with a formal type parameter `T2` (also bounded by `Number`) that extends `B<T2>`, and (iii) a non-parametric class `D` that extends `B<Integer>`. `B` and `C` can be instantiated with any subtype of `Number` such as `Float`, so one can write:

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

Unlike arrays, generic types are *not* covariant: $C<A>$ is a subtype of $C<B>$ if and only if $A = B$. Moreover, arrays of generic types are not allowed [14].

Type parameters may also be associated with methods. Such parameters are supplied at the beginning of the generic method's signature, after any qualifiers. For example, a class may declare a generic method as follows:

```
public <T3> void zap(T3 z){ ... }
```

Calls to generic methods do not need to supply actual type parameters because these can be inferred from context.

Wildcards [21] are unnamed type parameters that can be used in declarations. Wildcards can be bounded from above or below, as in `? extends B`, or `? super B`, respectively. For example, interface `Collection<E>` of the Java 1.5 standard collections library defines a method

```
boolean addAll(Collection<? extends E> c){ ... }
```

in which the wildcard specifies the "element type" of parameter `c` to be a subtype of formal type parameter `E`, thus permitting one to add a collection of, say, `Float`s to a collection of `Number`s.

For backward compatibility, one can refer to a generic class without specifying type parameters. Operations on such "raw types" result in compile-time "unchecked warnings"[7] in cases where type-safety cannot be guaranteed (e.g., when calling certain methods on a receiver expression of a raw type). Unchecked warnings indicate the potential for class-cast exceptions at run-time, and the number of such warnings is a rough measure of the potential lack of type safety in the program.

3 Motivating Example

We will use the Java standard collections library in package `java.util` to illustrate our refactoring. In Java 1.5, `Collection` and its subtypes (e.g., `Vector` and `List`) have a type parameter representing the collection's element type, `Map` and its subtypes (e.g., `TreeMap` and `Hashtable`) have two type parameters representing the type of its key and its value, respectively, and `Iterator` has a single type parameter representing the type of object returned by the `next()` method.

Figure 1 shows a Java program making nontrivial use of several kinds of containers. In this program, class `IntList` contains an array of `int`s, and provides an iterator over its elements, and a method for summing its elements. The `iterator()` method creates a `ListIterator`, a local implementation of `Iterator` that returns `Integer` objects wrapping the values stored in an `IntList`. Class `Example`'s `main()` method creates `IntList`s as well as several objects of various standard library types. Executing the example program prints the list `[[2.0, 4.4]]`. The example program illustrates several salient aspects of the use of standard container classes:

- nested containers (here, a `Vector` of `Vector`s), on line 14,
- iterators over standard containers, on line 19,

[7] Unchecked warnings are issued by Sun's javac 1.5 compiler when given the `-Xlint:unchecked` option.

```
(1)  public class Example{
(2)    public static void main(String[] args){
(3)      Map m1 = new HashMap();
(4)      Double d1 = new Double(3.3);
(5)      Double d2 = new Double(4.4);
(6)      IntList list1 = new IntList(new int[]{ 16, 17 });
(7)      IntList list2 = new IntList(new int[]{ 18, 19 });
(8)      m1.put(d1, list1); m1.put(d2, list2);
(9)      Vector v1 = new Vector();
(10)     v1.add(new Float(2.0));
(11)     List list5 = new ArrayList();
(12)     list5.add(find(m1, 37));
(13)     v1.addAll(list5);
(14)     Vector v2 = new Vector();
(15)     v2.add(v1);
(16)     System.out.println(v2);
(17)   }
(18)   static Object find(Map m2, int i){
(19)     Iterator it = m2.keySet().iterator();
(20)     while (it.hasNext()){
(21)       Double d3 = (Double)it.next();
(22)       if (((IntList)m2.get(d3)).sum()==i) return d3;
(23)     }
(24)     return null;
(25)   }
(26) }
(27) class IntList{
(28)   IntList(int[] is){ e = is; }
(29)   Iterator iterator(){ return new ListIterator(this); }
(30)   int sum(){ return sum2(0); }
(31)   int sum2(int j){
         return (j==e.length ? 0 : e[j]+sum2(j+1)); }
(32)   int[] e;
(33) }
(34) class ListIterator implements Iterator{
(35)   ListIterator(IntList list3){
         list4 = list3; count = 0; }
(36)   public boolean hasNext(){
         return count+1 < list4.e.length; }
(37)   public Object next(){
         return new Integer(list4.e[count++]); }
(38)   public void remove(){
         throw new UnsupportedOperationException(); }
(39)   private int count;
(40)   private IntList list4;
(41) }
```

Fig. 1. Example program that uses non-generic container classes. Program constructs that give rise to unchecked warnings are indicated using wavy underlining.

```
(1)  public class Example{
(2)    public static void main(String[] args){
(3)      Map<Double,IntList> m1 = new HashMap<Double,IntList>();
(4)      Double d1 = new Double(3.3);
(5)      Double d2 = new Double(4.4);
(6)      IntList list1 = new IntList(new int[]{ 16, 17 });
(7)      IntList list2 = new IntList(new int[]{ 18, 19 });
(8)      m1.put(d1, list1); m1.put(d2, list2);
(9)      Vector<Number> v1 = new Vector<Number>();
(10)     v1.add(new Float(2.0));
(11)     List<Double> list5 = new ArrayList<Double>();
(12)     list5.add(find(m1, 37));
(13)     v1.addAll(list5);
(14)     Vector<Vector<Number>> v2 = new Vector<Vector<Number>>();
(15)     v2.add(v1);
(16)     System.out.println(v2);
(17)   }
(18)   static Double find(Map<Double,IntList> m2, int i){
(19)     Iterator<Double> it = m2.keySet().iterator();
(20)     while (it.hasNext()){
(21)       Double d3 = it.next();
(22)       if ((m2.get(d3)).sum() == i) return d3;
(23)     }
(24)     return null;
(25)   }
(26) }
(27) class IntList{
(28)   IntList(int[] is){ e = is; }
(29)   ListIterator iterator(){ return new ListIterator(this); }
(30)   int sum(){ return sum2(0); }
(31)   int sum2(int j){
         return (j==e.length ? 0 : e[j]+sum2(j+1)); }
(32)   int[] e;
(33) }
(34) class ListIterator implements Iterator<Integer>{
(35)   ListIterator(IntList list3){
         list4 = list3; count = 0; }
(36)   public boolean hasNext(){
         return count+1 < list4.e.length; }
(37)   public Integer next(){
         return new Integer(list4.e[count++]); }
(38)   public void remove(){
         throw new UnsupportedOperationException(); }
(39)   private int count;
(40)   private IntList list4;
(41) }
```

Fig. 2. Refactored version of the program of Figure 1. Underlining indicates declarations and allocation sites for which a different type is inferred, and expressions from which casts have been removed.

– methods like `Collection.addAll()` that combine the contents of containers, on line 13,
– methods like `Map.keySet()` that expose the constituent components of standard containers (namely, a `java.util.Set` containing the `Map`'s keys), on line 19,
– a user-defined subtype (`ListIterator`) of a standard container type, on line 34, and
– the need for down-casts (lines 21 and 22) to recover type information.

Compiling the example program with Sun's javac 1.5 compiler yields six unchecked warnings, which are indicated in Figure 1 using wavy underlining. For example, for the call `m1.put(d1, list1)` on line 8, the following message is produced: "warning: [unchecked] unchecked call to `put(K,V)` as a member of the raw type `java.util.Map`".

Figure 2 shows the result of our refactoring algorithm on the program of Figure 1. Declarations and allocation sites have been rewritten to make use of generic types (on lines 3, 9, 11, 14, 18, and 19), and the down-casts have been removed (on lines 21 and 22). Moreover, note that `ListIterator` (line 34) now implements `Iterator<Integer>` instead of raw `Iterator`, and that the return type of `ListIterator.next()` on line 37 has been changed from `Object` to `Integer`. This latter change illustrates the fact that inferring a precise generic type for declarations and allocation sites may require changing the declared types of non-containers in some cases. The resulting program is type-correct, behaves as before, and compiling it does not produce any unchecked warnings.

4 Type Constraints

This paper extends a model of type constraints [17] previously used by several of the current authors for refactorings for generalization [20] and for the customization of Java container classes [7]. We only summarize the essential details of the type constraints framework here, and refer the reader to [20] for more details.

In the remainder of the paper, $\mathcal{P}$ will denote the original program. Type constraints are generated from $\mathcal{P}$'s abstract syntax tree (AST) in a syntax-directed manner. A set of constraint generation rules generates, for each program construct in $\mathcal{P}$, one or more type constraints that express the relationships that must exist between the declared types of the construct's constituent expressions, in order for that program construct to be type-correct. By definition, a program is *type-correct* if the type constraints for all constructs in that program are satisfied. In the remainder of this paper, we assume that $\mathcal{P}$ is type-correct.

Figure 3 shows the notation used to formulate type constraints. Figure 4 shows the syntax of type constraints[8]. Figure 5 shows constraint generation rules for a number

[8] In this paper, we assume that type information about identifiers and expressions is available from a compiler or type checker. Two syntactically identical identifiers will be represented by the same constraint variable if only if they refer to the same entity. Two syntactically identical expressions will be represented by the same constraint variable if and only if they correspond to the same node in the program's abstract syntax tree.

M, M'	methods (signature, return type, and a reference to the method's declaring class are assumed to be available)
m, m'	method names
F, F'	fields (name, type, and declaring class are assumed to be available)
f, f'	field names
C, C'	classes and interfaces
K, W, V, T	formal type parameters
$E, E', E_1, E_2, \ldots$	expressions (corresponding to a specific node in the program's AST)
$[E]$	the type of expression or declaration element E
$[E]_{\mathcal{P}}$	the type of E in the original program $\mathcal{P}$
$[M]$	the declared return type of method M
$[F]$	the declared type of field F
Decl(M)	the class that declares method M
Decl(F)	the class that declares field F
Param(M, i)	the i-th formal parameter of method M
$T(E)$	actual type parameter T in the type of the expression E
$T(C)$	actual type parameter T of class C
RootDefs(M)	$\{$ *Decl*(M') | M overrides M', and there exists no $M''(M'' \neq M')$ such that M' overrides M'' $\}$

Fig. 3. Notation used for defining 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** $\cdots$ **or** $\alpha \leq \alpha_k$	$\alpha \leq \alpha_i$ must hold for at least one i, $(1 \leq i \leq k)$

Fig. 4. Syntax of type constraints. Constraint variables α, α', $\ldots$ represent the types associated with program constructs and must be of one of the following forms: (i) a type constant, (ii) the type of an expression $[E]$, (iii) the type declaring a method *Decl*(M), or (iv) the type declaring a field *Decl*(F).

of language constructs. These rules are essentially the same as in [20, 7], but rely on a predicate *isLibraryClass* to avoid the generation of constraints for: (i) calls to methods (constructors, static methods, and instance methods) declared in generic library classes, (ii) accesses to fields in generic library classes, and (iii) overriding relationships involving methods declared in generic library classes. Note the assumption that the program is already using a generic version of the library. Therefore, $[E]_{\mathcal{P}}$ may denote a generic type. Section 5 will discuss the generation of constraints that are counterparts to (i)–(iii) for references to generic library classes.

We now study a few of the constraint generation rules of Figure 5. Rule (1) states that an assignment $E_1 = E_2$ is type correct if the type of E_2 is the same as or a subtype of the type of E_1. For a field-access expression $E \equiv E_0.f$ that accesses a field F declared in class C, rule (2) defines the type of E to be the same as the declared type of F and rule (3) requires that the type of expression E_0 be a subtype of the type C

$$\frac{\mathcal{P} \text{ contains assignment } E_1 = E_2}{[E_2] \leq [E_1]} \quad (1)$$

$$\frac{\mathcal{P} \text{ contains field access } E \equiv E_0.f \text{ to field } F,\ C = \mathit{Decl}(F),\ \neg \mathit{IsLibraryClass}(C)}{[E] = [F]} \quad (2)$$
$$[E_0] \leq C \quad (3)$$

$$\frac{\mathcal{P} \text{ contains constructor call } E \equiv \texttt{new}\ C(E_1, \cdots, E_k)}{[E] = C} \quad (4)$$

$$\frac{\mathcal{P} \text{ contains constructor call } \texttt{new}\ C(E_1, \cdots, E_k) \text{ to constructor } M,\ \neg \mathit{IsLibraryClass}(C),\ E'_i \equiv \mathit{Param}(M, i),\ 1 \leq i \leq k}{[E_i] \leq [E'_i]} \quad (5)$$

$$\frac{\mathcal{P} \text{ contains call } E_0.m(E_1, \cdots, E_k) \text{ to virtual method } M,\ \mathit{RootDefs}(M) = \{\ C_1, \cdots, C_q\ \}}{[E_0] \leq C_1\ \mathbf{or} \cdots \mathbf{or}\ [E_0] \leq C_q} \quad (6)$$

$$\frac{\mathcal{P} \text{ contains call } E \equiv E_0.m(E_1, \cdots, E_k) \text{ to virtual method } M,\ \neg \mathit{IsLibraryClass}(\mathit{Decl}(M)),\ E'_i \equiv \mathit{Param}(M, i),\ 1 \leq i \leq k}{[E] = [M]} \quad (7)$$
$$[E_i] \leq [E'_i] \quad (8)$$

$$\frac{\mathcal{P} \text{ contains direct call } E \equiv C.m(E_1, \cdots, E_k) \text{ to static method } M,\ \neg \mathit{IsLibraryClass}(C),\ E'_i \equiv \mathit{Param}(M, i),\ 1 \leq i \leq k}{[E] = [M]} \quad (9)$$
$$[E_i] \leq [E'_i] \quad (10)$$

$$\frac{\mathcal{P} \text{ contains cast expression } E \equiv (C)E_0}{[E] = C} \quad (11)$$

$$\frac{\mathcal{P} \text{ contains down-cast expression } E \equiv (C)E_0,\ C \text{ is not an interface, } [E_0]_{\mathcal{P}} \text{ is not an interface}}{C \leq [E_0]} \quad (12)$$

$$\frac{M \text{ contains an expression } E \equiv \texttt{this},\ C = \mathit{Decl}(M)}{[E] = C} \quad (13)$$

$$\frac{M \text{ contains an expression } E \equiv \texttt{return}\ E_0}{[E_0] \leq [M]} \quad (14)$$

$$\frac{M' \text{ overrides } M,\ 1 \leq i \leq \mathit{NrParams}(M'),\ E_i \equiv \mathit{Param}(M, i),\ E'_i \equiv \mathit{Param}(M', i),\ \neg \mathit{IsLibraryClass}(\mathit{Decl}(M))}{[E_i] = [E'_i]} \quad (15)$$
$$[M'] \leq [M] \quad (16)$$

Fig. 5. Inference rules for deriving type constraints from various Java constructs.

in which F is declared. Here, the predicate *IsLibraryClass*(C) is used to restrict the generation of these constraints to situations where class C is not a library type.

Rules (6)–(8) are concerned with a virtual method call $E \equiv E_0.m(E_1, \cdots, E_k)$ that refers to a method M. Rule (6) 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 $M_1, \cdots, M_q$ declared in supertypes $C_1, \cdots, C_q$ 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_i. This is expressed by way of a disjunction in rule (6) using auxiliary function *RootDefs* of Figure 3. Rule (7) defines the type of the entire call-expression E to be the same as M's return type. Further, the type of each actual parameter E_i must be the same as or a subtype of the type of the corresponding formal parameter E'_i (rule (8)).

Rules (11) and (12) are concerned with down-casts. The former defines the type of the entire cast expression to be the same as the target type C referred to in the cast. The latter requires this C to be a subtype of the expression E_0 being casted.

The constraints discussed so far are only concerned with type-correctness. Additional constraints are needed to ensure that program behavior is preserved. Rules (15) and (16) state that overriding relationships in $\mathcal{P}$ must be preserved in the refactored program (note that covariant return types are allowed in Java 1.5). Moreover, if a method $m(E_1, \cdots, E_k)$ overloads another method, then changing the declared type of any formal parameter E_i may affect the specificity ordering that is used for compile-time overload resolution [11]. To avoid such behavioral changes, we generate additional constraints $[E_i] = [E_i]_{\mathcal{P}}$ for all i $(1 \leq i \leq k)$ to ensure that the signatures of overloaded methods remain the same. Constraints that have the effect of preserving the existing type are also generated for actual parameters and return types used in calls to methods in classes for which source code cannot be modified.

5 Type Constraints for Generic Libraries

Additional categories of type constraints are needed for: (i) calls to methods in generic library classes, (ii) accesses to fields in generic library classes[9], and (iii) user classes that override methods in generic library classes. We first discuss a few concrete examples of these constraints, and then present rules that automate their generation.

5.1 New Forms of Type Constraints

Consider the call `m1.put(d1,list1)` on line (8) of Figure 1, which resolves to method `V Map<K, V>.put(K, V)`. This call is type-correct if: (i) the type of the first actual parameter, `d1`, is a subtype of the first actual type parameter of receiver `m1`, and (ii) the type of the second actual parameter, `list1`, is a subtype of the second actual type parameter of `m1`. These requirements are expressed by the constraints [`d1`]≤`K(m1)` and [`list1`]≤`V(m1)`, where the notation $T(E)$ is used for a new kind of

[9] These can be handled in the same way as calls to methods in generic library classes, and will not be discussed in detail.

$$\frac{\mathcal{P} \text{ contains call } E_{rec}.\texttt{put}(E_{key}, E_{value}) \text{ to method } \texttt{V Map} < \texttt{K}, \texttt{V} > .\texttt{put}(\texttt{K}, \texttt{V})}{\begin{array}{r}[E_{key}] \leq \texttt{K}(E_{rec}) \\ [E_{value}] \leq \texttt{V}(E_{rec}) \\ [E_{rec}.\texttt{put}(E_{key}, E_{value})] = \texttt{V}(E_{rec})\end{array}} \quad [\texttt{put}]$$

$$\frac{\mathcal{P} \text{ contains call } E_{rec}.\texttt{get}(E_{key}) \text{ to method } \texttt{V Map} < \texttt{K}, \texttt{V} > .\texttt{get}(\texttt{Object})}{[E_{rec}.\texttt{get}(E_{key})] = \texttt{V}(E_{rec})} \quad [\texttt{get}]$$

$$\frac{\begin{array}{c}\mathcal{P} \text{ contains call } E_{rec}.\texttt{addAll}(E_{arg}) \text{ to method} \\ \texttt{boolean Collection} < \texttt{E} > .\texttt{addAll}(\texttt{Collection} <? \texttt{ extends E} >)\end{array}}{\begin{array}{r}[E_{arg}] \leq \texttt{Collection} \\ \texttt{E}(E_{arg}) \leq \texttt{E}(E_{rec})\end{array}} \quad [\texttt{addAll}]$$

Fig. 6. Constraint generation rules for calls to `V Map<K,V>.put(K,V)`, `V Map<K,V>.get(Object)`, and `boolean Collection<E>.addAll(Collection<? extends E>)`.

constraint variable that denotes the value of actual type parameter T in the type of the expression E. Similar constraints are generated for return values of methods in generic library classes. For example, the call to `m2.get(d3)` on line (22) of Figure 1 refers to method `V Map<K, V>.get(Object)`. Here, the type of the entire expression has the same type as the second actual type parameter of the receiver expression `m2`, which is expressed by: [`m2.get(d3)`] = `V`(`m2`). Wildcards are handled similarly. For example, the call `v1.addAll(list5)` on line (13) of Figure 1 resolves to method `boolean Collection<E>.addAll(Collection<? extends E>)`. This call is type-correct if the actual type parameter of `list5` is a subtype of the actual type parameter of receiver `v1`: `E`(`list5`) ≤`E`(`v1`).

Figure 6 shows rules that could be used to generate the type constraints for the calls to `put`, `get`, and `addAll` that were just discussed. Observe that the formal parameter of the `get` method has type `Object` and no relationship exists with the actual type parameter of the receiver expression on which `get` is called[10].

We generate similar constraints when a user class overrides a method in a generic library class, as was the case in the program of Figure 1 where `ListIterator.next()` overrides `Iterator.next()`. Specifically, if a user class C overrides a method in a library class L with a formal type parameter T, we introduce a new constraint variable $T(C)$ that represents the instantiation of L from which C inherits. Then, if a method M' in class C overrides a method M in L, and the signature of M refers to a type parameter T of L, we generate constraints that relate the corresponding parameter or return type of M' to $T(C)$.

[10] While it might seem more natural to define `get` as `V Map<K,V>.get(K)` instead of `V Map<K,V>.get(Object)`, this would require the actual parameter to be of type `K` at compile-time, and additional `instanceof`-tests and downcasts would need to be inserted if this were not the case. The designers of the Java 1.5 standard libraries apparently preferred the flexibility of being able to pass any kind of object over the additional checking provided by a tighter argument type. They adopted this approach consistently for all methods that do not write to a container (e.g., `contains`, `remove`, `indexOf`).

For example, method `ListIterator.next()` on line (37) of Figure 1 overrides `Iterator<E>.next()`. Since the return type of `Iterator.next()` is type parameter `E`, we generate a constraint [`ListIterator.next()`] = `E(ListIterator)`. Note that this constraint precisely captures the required overriding relationship because `ListIterator.next()` only overrides `Iterator<Integer>.next()` if the return type of `ListIterator.next()` is `Integer`.

5.2 Constraint Generation Rules for Generic Libraries

While type constraint generation rules such as those of Figure 6 can be written by the programmer, this is tedious and error-prone. Moreover, it is clear that their structure is regular, determined by occurrences of type parameters in signatures of methods in generic classes. Figure 7 shows rules for *generating constraints for calls to methods in generic classes*. For a given call, rule (r1) creates constraints that define the type of the method call expression, and rule (r2) creates constraints that require the type of actual parameters to be equal to or a subtype of the corresponding formal parameters. A recursive helper function *CGen* serves to generate the appropriate constraints, and is defined by case analysis on its second argument, $\mathcal{T}$. Case (c1) applies when $\mathcal{T}$ is a non-generic class, e.g., `String`. Case (c2) applies when $\mathcal{T}$ is a type parameter. In the remaining cases the function is defined recursively. Cases (c3) and (c4) apply when $\mathcal{T}$ is an upper or lower-bounded wildcard type, respectively. Finally, case (c5) applies when $\mathcal{T}$ is a generic type.

$$
\textit{CGen}(\alpha, \mathcal{T}, E, op) = \begin{cases}
\{\alpha \; op \; C\} & \text{when } \mathcal{T} \equiv \; C & \text{(c1)} \\
\{\alpha \; op \; T_i(E)\} & \text{when } \mathcal{T} \equiv \; T_i & \text{(c2)} \\
\textit{CGen}(\alpha, \tau, E, \leq) & \text{when } \mathcal{T} \equiv \; ?\; extends\; \tau & \text{(c3)} \\
\textit{CGen}(\alpha, \tau, E, \geq) & \text{when } \mathcal{T} \equiv \; ?\; super\; \tau & \text{(c4)} \\
\{\alpha \; op \; C\} \cup & \text{when } \mathcal{T} \equiv C < \tau_1, \ldots, \tau_m > & \text{(c5)} \\
\textit{CGen}(W_i(\alpha), \tau_i, E, =) & \text{and } C \text{ is declared as} & \\
 & C < W_1, \ldots, W_m >, 1 \leq i \leq m &
\end{cases}
$$

$$
\frac{\mathcal{P} \text{ contains call } E \equiv \; E_{rec}.m(E_1, \ldots, E_k) \text{ to method } M, 1 \leq i \leq k}{\begin{array}{ll} \textit{CGen}([E], \; [M]_{\mathcal{P}}, \; E_{rec}, \; =) \cup & \text{(r1)} \\ \textit{CGen}([E_i], \; [\textit{Param}(M, i)]_{\mathcal{P}}, \; E_{rec}, \; \leq) & \text{(r2)} \end{array}}
$$

Fig. 7. Constraint generation rules for calls to methods in generic classes.

We will now give a few examples that show how the rules of Figure 7 are used to generate type constraints such as those generated by the rules of Figure 6. As an example, consider again the call `m1.put(d1, list1)` to `V Map<K,V>.put(K,V)`

on line 8 of the original program $\mathcal{P}$. Applying rule (r1) of Figure 7 yields *CGen*([`m1.put(d1, list1)`], V, m1, =), and applying case (c2) of the definition of *CGen* produces the set of constraints {[`m1.put(d1, list1)`] = V(m1)}. Likewise, for parameter `d1` in the call `m1.put(d1, list1)` on line 8, we obtain `m1.put(d1, list1)` $\stackrel{r2}{\Rightarrow}$ *CGen*([`d1`], K, m1, $\leq$) $\stackrel{c2}{\Rightarrow}$ {[`d1`] $\leq$ K(m1)}. Two slightly more interesting cases are the following:

line 13: `v1.addAll(list5)` $\stackrel{r2}{\Rightarrow}$
CGen([`list5`], `Collection<? extends E>`, v1, $\leq$) $\stackrel{c5}{\Rightarrow}$
{[`list5`] $\leq$ `Collection`} $\cup$ *CGen*(E(l), `? extends E`, v1, =) $\stackrel{c3}{\Rightarrow}$
{[`list5`] $\leq$ `Collection`} $\cup$ *CGen*(E(l), E, v1, $\leq$) $\stackrel{c2}{\Rightarrow}$
{[`list5`] $\leq$ `Collection`} $\cup$ {E(l) $\leq$ E(v1)}

line 19: `m2.keySet()` $\stackrel{r1}{\Rightarrow}$ *CGen*([`m2.keySet()`], `Set<K>`, m2, =) $\stackrel{c5}{\Rightarrow}$
{[`m2.keySet()`] = `Set`} $\cup$ *CGen*(E(`m2.keySet()`), K, m2, =) $\stackrel{c2}{\Rightarrow}$
{[`m2.keySet()`] = `Set`} $\cup$ {E(`m2.keySet()`) = K(m2)}

Table 1 below shows the full set of generics-related type constraints computed for the example program in Figure 1. Here, the appropriate rules and cases of Figure 7 are indicated in the last two columns.

line	code	type constraint(s)	rule	cases
8	`m1.put(d1, list1)`	[`d1`] $\leq$ K(m1) [`list1`] $\leq$ V(m1) [`m1.put(d1, list1)`] = V(m1)	r2 r2 r1	c2 c2 c2
8	`m1.put(d2, list2)`	[`d2`] $\leq$ K(m1) [`list2`] $\leq$ V(m1) [`m1.put(d2, list2)`] = V(m1)	r2 r2 r1	c2 c2 c2
10	`v1.add(new Float(2.0))`	[`new Float(2.0)`] $\leq$ E(v1)	r2	c2
12	`list5.add(find(m1, 37))`	[`find(m1, 37)`] $\leq$ E(list5)	r2	c2
13	`v1.addAll(list5)`	[`list5`] $\leq$ `Collection` E(list5) $\leq$ E(v1)	r2	c5, c3, c2
15	`v2.add(v1)`	[`v1`] $\leq$ E(v2)	r2	c2
19	`m2.keySet()`	[`m2.keySet()`] = `Set` E(`m2.keySet()`) = K(m2)	r1	c5, c2
19	`m2.keySet().iterator()`	[`m2.keySet().iterator()`] = `Iterator` E(`m2.keySet().iterator()`) = E(`m2.keySet()`)	r1	c5, c2
21	`it.next()`	[`it.next()`] = E(it)	r1	c2
22	`m2.get(d3)`	[`m2.get(d3)`] = V(m2)	r1	c2
37	override of `E Iterator<E>.next()`	[`ListIterator.next()`]= E(ListIterator)		

Table 1. Generics-related type constraints created for code from Figure 1. The labels in the two rightmost columns refer to rules and cases in the definitions of Figure 7.

Our algorithm also creates type constraints for methods in application classes that override methods in generic library classes. For example, the last row of Table 1 shows a type constraint required for the overriding of method `E Iterator<E>.next()` in class `ListIterator`. The rules for generating such constraints are similar to those in Figure 7 and have been omitted due to space limitations.

5.3 Closure Rules

Thus far, we introduced additional constraint variables such as $K(E)$ to represent the actual type parameter bound to K in E's type, and we described how calls to methods in generic libraries give rise to constraints on these variables. However, we have not yet discussed how types inferred for actual type parameters are constrained by language constructs such as assignments and parameter passing. For example, consider an assignment `a = b`, where `a` and `b` are both declared of type `Vector<E>`. The lack of covariance for Java generics implies that `E(a)` = `E(b)`. The situation becomes more complicated in the presence of inheritance relations between generic classes. Consider a situation involving class declarations[11] such as:

```
interface List<E_l> { ... }
class Vector<E_v> implements List<E_v> { ... }
```

and two variables, `c` of type `List` and `d` of type `Vector`, and an assignment `c = d`. This assignment can only be type-correct if the same type is used to instantiate $\mathtt{E}_l$ in the type of `c` and $\mathtt{E}_v$ in the type of `d`. In other words, we need a constraint $\mathtt{E}_l(\mathtt{c}) = \mathtt{E}_v(\mathtt{d})$. The situation becomes yet more complicated if generic library classes are assigned to variables of non-generic supertypes such as `Object`. Consider the program fragment:

```
Vector v1 = new Vector();
v1.add("abc");
Object o = v1;
Vector v2 = (Vector)o;
```

Here, we would like to infer $\mathtt{E}_v(\mathtt{v1}) = \mathtt{E}_v(\mathtt{v2}) = \mathtt{String}$, which would require tracking the flow of actual type parameters through variable `o`[12].

The required constraints are generated by a set of closure rules that is given in Figure 8. These rules infer, from an existing system of constraints, a set of additional constraints that unify the actual type parameters as outlined in the examples above. In the rules of Figure 8, α and α' denote constraint variables that are not type constants. Rule (17) states that, if a subtype constraint $\alpha \leq \alpha'$ exists, and another constraint implies that the type of α' or α has formal type parameter T_1, then the types of α and α'

[11] In the Java collections library, the type formal parameters of both `Vector` and `List` have the same name, `E`. In this section, for disambiguation, we subscript them with v and l, respectively.

[12] In general, a cast to a parameterized type cannot be performed in a dynamically safe manner because type arguments are erased at run-time. In this case, however, our analysis is capable of determining that the resulting cast to `Vector<String>` would always succeed.

$$\frac{\alpha \leq \alpha' \qquad T_1(\alpha) \text{ or } T_1(\alpha') \text{ exists}}{T_1(\alpha) = T_1(\alpha')} \quad (17)$$

$$\frac{\begin{array}{c} T_1(\alpha) \text{ exists} \\ C_1\langle T_1 \rangle \texttt{ extends/implements } C_2\langle \mathcal{T} \rangle \\ C_2 \text{ is declared as } C_2\langle T_2 \rangle \end{array}}{\mathit{CGen}(T_2(\alpha), \mathcal{T}, \alpha, =)} \quad (18)$$

Fig. 8. Closure rules.

must have the same actual type parameter T_1[13]. This rule thus expresses the invariant subtyping among generic types. Observe that this has the effect of associating type parameters with variables of non-generic types, in order to ensure that the appropriate unification occurs in the presence of assignments to variables of non-generic types. For the example code fragment, a constraint variable $\mathtt{E}_v(\mathtt{o})$ is created by applying rule (17). Values computed for variables that denote type arguments of non-generic classes (such as `Object` in this example) are disregarded at the end of constraint solution.

Rule (18) is concerned with subtype relationships among generic library classes such as the one discussed above between classes `Vector` and `List`. The rule states that if a variable $T_1(\alpha)$ exists, then a set constraints is created to relate $T_1(\alpha)$ to the types of actual type parameters of its superclasses. Note that rule (18) uses the function *CGen*, defined in Figure 7. For example, if we have two variables, `c` of type `List` and `d` of type `Vector`, and an initial system of constraints $[\,\mathtt{d}\,] \leq [\,\mathtt{c}\,]$, and $\mathtt{String} \leq \mathtt{E}_v(\mathtt{d})$, then using the rules of Figure 8, we obtain the additional constraints $\mathtt{E}_v(\mathtt{d}) = \mathtt{E}_v(\mathtt{c})$, $\mathtt{E}_l(\mathtt{d}) = \mathtt{E}_v(\mathtt{d})$, $\mathtt{E}_l(\mathtt{c}) = \mathtt{E}_l(\mathtt{d})$ and $\mathtt{E}_l(\mathtt{c}) = \mathtt{E}_v(\mathtt{d})$.

We conclude this section with a remark about special treatment of the `clone()` method. Although methods that override `Object.clone()` may contain arbitrary code, we assume that implementations of `clone()` are well-behaved (in the sense that the returned object preserves the type arguments of the receiver expression) and generate constraints accordingly.

6 Constraint Solving

Constraint solution involves computing a set of legal types for each constraint variable and proceeds in standard iterative fashion. In the initialization phase, an initial type estimate is associated with each constraint variable, which is one of the following: (i) a singleton set containing a specific type (for constants, type literals, constructor calls, and references to declarations in library code), (ii) the singleton set $\{\, B \,\}$ (for each constraint variable $K(E)$ declared in library code, where K is a formal type parameter with bound B, to indicate that E should be left raw), or (iii) the type universe (in all other cases). In the iterative phase, a work-list is maintained of constraint variables whose estimate has recently changed. In each iteration, a constraint variable α is selected from the work-list, and all type constraints that refer to α are examined. For

[13] Unless wildcard types are inferred, which we do not consider in this paper.

each type constraint $t = \alpha \leq \alpha'$, the estimates associated with α and α' are updated by removing any element that would violate t, and α and/or α' are reentered on the work-list if appropriate (other forms of type constraints are processed similarly). As estimates monotonically decrease in size as constraint solution progresses, termination is guaranteed. The result of this process is a set of legal types for each constraint variable.

Since the constraint system is typically underconstrained, there is usually more than one legal type associated with each constraint variable. In the final solution, there needs to be a singleton type estimate for each constraint variable, but the estimates for different constraint variables are generally not independent. Therefore, a single type is chosen from each non-singleton estimate, after which the inferencer is run to propagate that choice to all related constraint variables, until quiescence. The optimization criterion of this step is nominally to select a type that maximizes the number of casts removed. As a simple approximation to this criterion, our algorithm selects an arbitrary most specific type from the current estimate (which is not necessarily unique). Although overly restrictive in general (a less specific type may suffice to remove the maximum number of casts/warnings), and potentially sub-optimal, the approach appears to be quite effective in practice. The type selection step also employs a filter that avoids selecting "tagging" interfaces such as `java.lang.Serializable` that define no methods, unless such are the only available choices[14].

In some cases, the actual type parameter inferred by our algorithm is equal to the bound of the corresponding formal type parameter (typically, `Object`). Since this does not provide any benefits over the existing situation (no additional casts can be removed), our algorithm leaves raw any declarations and allocation sites for which this result is inferred. The opposite situation, where the actual type parameter of an expression is completely unconstrained, may also happen, in particular for incomplete programs. In principle, any type can be used to instantiate the actual type parameter, but since each choice is arbitrary, our algorithm leaves such types raw as well.

There are several cases where raw types must be retained to ensure that program behavior is preserved. When an application passes an object o of a generic library class to an external library[15], nothing prevents that library from writing values into o's fields (either directly, or by calling methods on o). In such cases, we cannot be sure what actual type parameter should be inferred for o, and therefore generate an additional constraint that equates the actual type parameter of o to be the bound of the corresponding formal type parameter, which has the effect of leaving o's type raw. Finally, Java 1.5 does not allow arrays of generic types [4] (e.g., type `Vector<String>[]` is not allowed). In order to prevent the inference of arrays of generic types, our algorithm generates additional constraints that equate the actual type parameter to the bound of the corresponding formal type parameter, which has the effect of preserving rawness.

Constraint solution yields a unique type for each constraint variable. Allocation sites and declarations that refer to generic library classes are rewritten if at least one of its inferred actual type parameters is more specific than the bound of the corresponding formal type parameter. Other declarations are rewritten if their inferred type is more spe-

[14] Donovan et al. [8] apply the same kind of filtering.

[15] The situation where an application receives an object of a generic library type from an external library is analogous.

cific than its originally declared type. Any cast in the resulting program whose operand type is a subtype of its target type is removed.

7 Implementation

We implemented our algorithm in the context of Eclipse, using existing refactoring infrastructure [3], which provides abstract syntax trees (with symbol binding resolution), source rewriting, and standard user-interface componentry. The implementation also builds on the type constraint infrastructure that was developed as part of our earlier work on type-related refactorings [20]. Much engineering effort went into making the refactoring scalable, and we only mention a few of the most crucial optimizations. First, a custom-built type hierarchy representation that allows subtype tests to be performed in constant time turned out to be essential. This is currently accomplished by maintaining, for each type, hash-based sets representing its supertypes and subtypes. However, we plan to investigate the use of more space-efficient mechanisms [22]. Second, as solution progresses, certain constraint variables are identified as being identically constrained (either by explicit equality constraints, or by virtue of the fact that Java's generic types are invariant, as was discussed in Section 2). When this happens, the constraint variables are *unified* into an equivalence class, for which a single estimate is kept. A union-find data structure is used to record the unifications in effect as solution progresses. Third, a compact and efficient representation of type sets turned out to be crucially important. Type sets are represented using the following expressions (in the following, S denotes a set of types, and t, t' denote types): (i) *universe*, representing the universe of all types, (ii) $subTypes(S)$, representing the set of subtypes of types in S, (iii) $superTypes(S)$, representing the set of supertypes of types in S, (iv) $intersect(S, S)$, (v) $arrayOf(S)$, representing the set of array types whose elements are in S, and (vi) $\{t, t', \ldots\}$, i.e., explicitly enumerated sets. In practice, most subtype queries that arise during constraint solving can be reduced to expressions for which obvious closed forms exist, and relatively few sets are ever expanded into explicitly represented sets. Basic algebraic simplifications are performed as sets are created, to reduce their complexity, as in $intersect(subTypes(S), S)) = subTypes(S)$. Fourth, we use a new Eclipse compiler API that has been added to improve performance of global refactorings, by avoiding the repeated resolution of often-used symbol bindings.

The refactoring currently supports three modes of operation. In *basic* mode arbitrary declarations may be rewritten, and precise parametric supertypes may be inferred for user-defined subtypes of generic library classes. In *noderived* mode, arbitrary declarations may be rewritten, but we do not change the supertype of user-defined subtypes of generic library classes. The *preserve erasure* mode is the most restrictive because it does not change the supertype of user-defined subtypes of generic library classes and it preserves the erasure of all methods. In other words, it only adds type arguments to declarations and hence preserves binary compatibility.

The forthcoming Eclipse 3.1 release will contain a refactoring called INFER GENERIC TYPE ARGUMENTS, which is largely based on the concepts and models presented in this paper and has adopted important parts of the research implementation. Currently, only the *preserve erasure* mode is supported.

8 Experimental Results

We evaluated our method on a suite of moderate-sized Java programs[16] by inferring actual type parameters for declarations and allocation sites that refer to the standard collections. In each case, the transformed source was validated using Sun's `javac` 1.5 compiler. Table 2 states, for each benchmark, the number of types, methods, total source lines, non-blank non-comment source lines, and the total number of declarations, allocation sites, and casts. We also give the number of allocation sites of generic types, generic-typed declarations, subtypes of generic types, and "unchecked warnings."

benchmark	benchmark size							generics-related measures			
	types	methods	LOC	NBNC LOC	decls	allocs	casts	allocs	decls	subtypes	warnings
JUnit	59	382	5,265	2,394	1,012	305	54	24	48	0	27
V_poker	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
PMD	395	2,048	38,222	18,093	5,163	1,066	774	75	286	1	183
HTMLParser	232	1,957	50,799	20,332	4,895	1,668	793	72	136	2	205
Jax	272	2,222	53,897	22,197	7,266	1,280	821	119	261	3	158
xtc	1,556	5,564	90,565	37,792	14,672	3,994	1,114	330	668	1	583

Table 2. Benchmark characteristics.

benchmark	casts removed			unchecked warnings remaining			program entities rewritten *(basic)*				time (sec.)
	basic	*noderived*	*preserve erasure*	*basic*	*noderived*	*preserve erasure*	generic allocs	generic decls	all decls	generic subtypes	*(basic)*
JUnit	24	24	21	2	2	8	24	57	79	0	9.9
V_poker	32	25	25	0	0	1	12	31	31	1	8.4
JLex	48	47	47	6	6	6	16	28	29	1	5.7
DB	40	40	37	0	634	634	13	32	43	1	8.7
JavaCup	488	488	486	2	2	2	19	70	81	0	9.0
TelnetD	38	38	37	0	0	0	15	27	30	0	6.8
Jess	83	83	82	9	642	642	42	58	68	1	15.9
JBidWatcher	207	204	177	5	5	25	74	195	238	3	64.5
ANTLR	86	84	82	5	7	8	45	80	202	1	32.1
PMD	154	135	132	21	35	36	64	278	322	9	42.0
HTMLParser	172	170	168	7	13	13	70	154	220	2	34.6
Jax	158	139	132	82	82	82	87	188	301	2	45.4
xtc	398	394	327	71	73	136	315	664	1,138	3	113.9

Table 3. Experimental results.

We experimented with the three modes—*basic*, *noderived*, and *preserve erasure*—that were discussed in Section 7. The results of running our refactoring on the benchmarks appear in Table 3. The first six columns of the figure show, for each of the three modes, the number of casts removed and unchecked warnings eliminated. The next four columns show, for the *basic* mode only, the number of generic allocation sites rewritten, the number of generic declarations rewritten, the total number of declarations rewritten, and the number of user-defined subtypes for which a precise generic supertype is inferred, respectively. The final column of the figure shows the total processing time

[16] For more details, see: `www.junit.org`, `www.cs.princeton.edu/~appel/modern/java/JLex/`, `www.cs.princeton.edu/~appel/modern/java/CUP/`, `www.spec.org/osg/jvm98/`, `vpoker.sourceforge.net`, `telnetd.sourceforge.net`, `www.antlr.org`, `jbidwatcher.sourceforge.net`, `pmd.sourceforge.net`, `htmlparser.sourceforge.net`, and `www.ovmj.org/xtc/`.

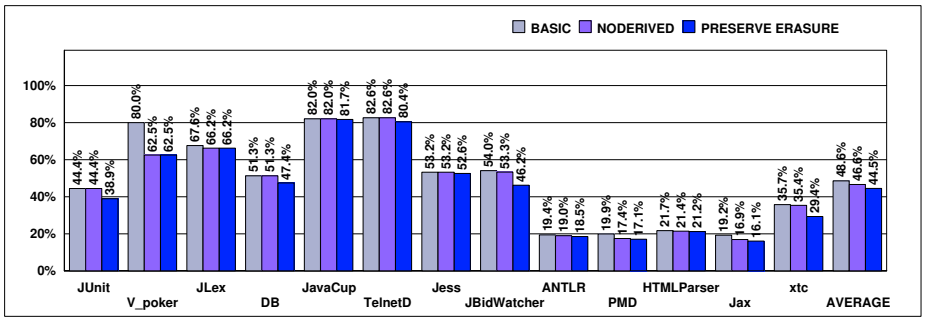


Fig. 9. Percentages of casts removed, for each of the three modes.

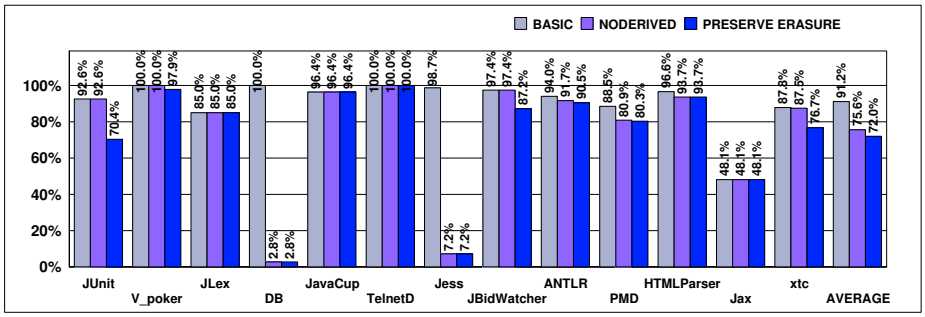


Fig. 10. Percentages of unchecked warnings eliminated, for each of the three modes.

in *basic* mode (the processing times for the other modes are similar). Processing *xtc*, our largest benchmark, took slightly under two minutes on a 1.6GHz Pentium M[17] and about 500Mb of heap space. These results clearly demonstrate our algorithm's scalability and we expect our technique to scale to programs of 500 KLOC or more.

8.1 Casts Removed

Figure 9 shows a bar chart that visualizes the percentage of casts removed in each benchmark, for each of the three modes. As can be seen from this figure, the *basic* mode removes an average of 48.6% of all casts from each benchmark, the *noderived* mode is slightly less effective with an average of 46.6% of all casts removed, and the *preserve erasure* mode is the least effective with 44.5% of all casts removed. When considering these numbers, the reader should note that the total number of casts given in Table 2 includes casts that are not related to the use of generic types. However, a manual inspection revealed that our tool removes the vast majority of generics-related casts, from roughly 75% to 100%. For example, we estimate that only one-fifth of *ANTLR*'s total number of casts relates to the use of collections, which is close to our tool's 19.4% removal rate.

[17] The processing time for *xtc* can be broken down as follows: 26.6 seconds for constraint generation, 71.1 seconds for constraint solving, and 16.3 seconds for source rewriting.

8.2 Unchecked Warnings Eliminated

A clearer indication of the effectiveness of our algorithm is apparent in the high proportion of "unchecked warnings" eliminated. This statistic is a rough measure of the improvement in the degree of type safety in the subject program. Figure 10 visualizes the percentage of unchecked warnings eliminated in each benchmark, for each of the three modes. As can be seen from this figure, the *basic* mode eliminates an average of 91.2% of all unchecked warnings for each benchmark, followed by the *noderived* mode with an average of 75.6% and the *preserve erasure* mode with 72.0%. Note that the lower averages for the *noderived* and and *preserve erasure* mode are largely due to the very low percentages of unchecked warnings removed on the *DB* and *Jess* benchmarks. We will discuss these cases in detail shortly.

8.3 Analysis of Results

We conducted a detailed manual inspection of the source code of the refactored benchmarks, in order to understand the limitations of our analysis. Below is a list of several issues that influenced the effectiveness of our analysis.

arrays. Several benchmarks create arrays of collections. For example, *JLex* creates an array of `Vector`s, and *xtc* creates several arrays of `HashMap`s. Since Java 1.5 does not permit arrays of generic types, raw types have to be used, resulting in several unchecked warnings, and preventing some casts from being removed (8 casts in the case of *JLex*).

wildcard usage. Several benchmarks (*JBidWatcher*, *HTMLParser*, *JUnit*, *Jax* and *xtc*) override library methods such as `java.lang.ClassLoader.loadClass()` that return wildcard types such as `java.lang.Class<?>`. Our method is incapable of inferring wildcard types, and leaves the return types in the overriding method definitions raw, resulting in unchecked warnings.

polymorphic containers. In several benchmarks (*JBidWatcher*, *Jax*, *Jess*, and *xtc*), unrelated types of objects are stored into a container. In such cases, the common upper bound of the stored objects is `java.lang.Object`, and the reference is left raw. The most egregious case occurs in *Jax*, where many different `Hashtable`s are stored in a single local variable. Splitting this local variable prior to the refactoring results in the elimination of an additional 71 unchecked warnings.

use of clone(). Various benchmarks (*JBidWatcher*, *JUnit*, *JavaCup*, *Jess*, *ANTLR*, and *xtc*) invoke the `clone()` method on container objects, and cast the result to a raw container type. Although our analysis tracks the flow of types through calls to `clone()`, rewriting the cast is not helpful, because the compiler would still produce a warning[18]. Our tool does not introduce casts to parameterized types, which means that unchecked warnings will remain.

static fields. The *xtc* benchmark contains 11 references to `Collections.EMPTY_LIST`, a static field of the raw type `List`. Several

[18] While casts to parameterized types such as `Vector<String>` are allowed in Java 1.5, such casts will succeed if the expression being casted is an instance of the corresponding erased type (`Vector`), and compilers produce a warning to inform users of this unintuitive behavior.

declarations will need to remain raw, resulting in unchecked warnings. It is interesting to note that the Java 1.5 standard libraries provide a generic method `<T> List<T> emptyList()` that enables polymorphic use of a shared empty list.

user-defined subtypes of generic library classes. In most cases, the inference of precise generic supertypes for user-defined subclasses of generic library classes has little impact on the number of casts removed and warnings eliminated. However, the *DB* and *Jess* benchmarks both declare a subclass `TableOfExistingFiles` of `java.util.Hashtable` that contains 600+ calls of the form `super.put(s1,s2)`, where `s1` and `s2` are `String`s. In *basic* mode, `TableOfExistingFiles` is made a subclass of `Hashtable<String,String>` and the unchecked warnings for these super-calls are eliminated. In the *noderived* and *preserve erasure* modes, `TableOfExistingFiles` remains a subclass of raw `Hashtable`, and a warning remains for each call to `put`, thus explaining the huge difference in the number of unchecked warnings.

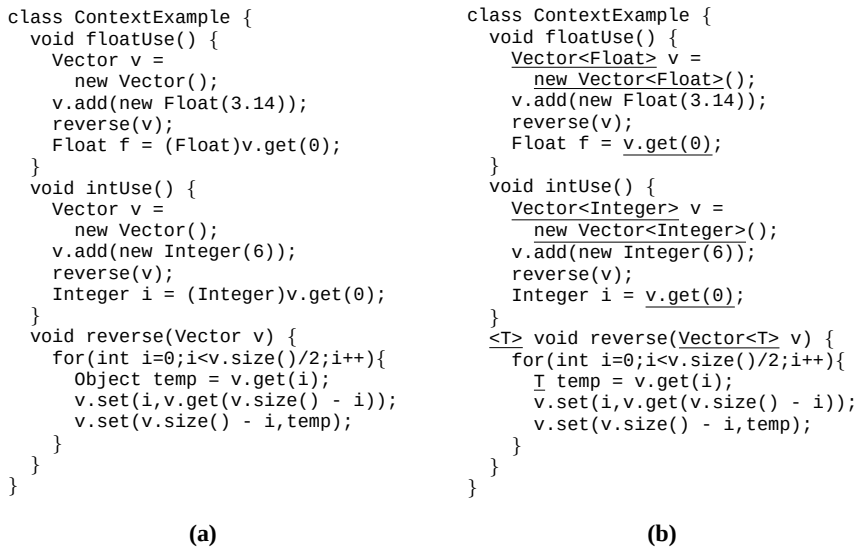
9 Context Sensitivity

Conceptually, our analysis can be extended with context-sensitivity by simply generating multiple sets of constraints for a method, one for each context. In principle, this can result in tighter bounds on parametric types when collections are used in polymorphic methods, and in the removal of more casts. Moreover, we could introduce type parameters on such polymorphic methods to accommodate their use with collections with different type parameters.

Figure 11(a) shows an example program that illustrates this scenario using a method `reverse()` for reversing the contents of a `Vector`. The `reverse()` method is invoked by methods `floatUse()` and `intUse()`, which pass it `Vector`s of `Float`s and `Integer`s, respectively. Applying the previously presented analysis would determine that both vectors reach method `reverse()` and infer an element type that is a common upper bound of `Float` and `Integer` such as `Number`. Therefore, all allocation sites and declarations in the program would be rewritten to `Vector<Number>`, and neither of the two casts could be removed.

However, if we create two analysis contexts for `reverse`—one for each call site—then one can infer bounds of `Float` and `Integer` for the two creation sites of vectors. Conceptually, this is equivalent to analyzing a transformed version of the program that contains two clones of the `reverse()` method, one of which is called from `intUse()`, the other from `floatUse()`. The two contexts of `reverse` would receive different type estimates for parameter `v`, and our code transformation could exploit this information by transforming `reverse()` into a generic method, and remove both casts. This result is shown in Figure 11(b).

We implemented a context-sensitive version of the previously presented algorithm, in which we used a low-cost variant of Agesen's Cartesian Product Algorithm [1, 2] to determine when different contexts should be created for a method, and reported the results in a previous technical report [19]. To our surprise, we could not find any non-synthetic benchmarks where the use of context-sensitive analysis resulted in the removal



```
class ContextExample {
  void floatUse() {
    Vector v =
      new Vector();
    v.add(new Float(3.14));
    reverse(v);
    Float f = (Float)v.get(0);
  }
  void intUse() {
    Vector v =
      new Vector();
    v.add(new Integer(6));
    reverse(v);
    Integer i = (Integer)v.get(0);
  }
  void reverse(Vector v) {
    for(int i=0;i<v.size()/2;i++){
      Object temp = v.get(i);
      v.set(i,v.get(v.size() - i));
      v.set(v.size() - i,temp);
    }
  }
}
```

(a)

```
class ContextExample {
  void floatUse() {
    Vector<Float> v =
      new Vector<Float>();
    v.add(new Float(3.14));
    reverse(v);
    Float f = v.get(0);
  }
  void intUse() {
    Vector<Integer> v =
      new Vector<Integer>();
    v.add(new Integer(6));
    reverse(v);
    Integer i = v.get(0);
  }
  <T> void reverse(Vector<T> v) {
    for(int i=0;i<v.size()/2;i++){
      T temp = v.get(i);
      v.set(i,v.get(v.size() - i));
      v.set(v.size() - i,temp);
    }
  }
}
```

(b)

Fig. 11. Example program that illustrates the need for context-sensitive analysis.

of additional casts. We believe that there are two major reasons why context-sensitive analysis was not useful. The first is that the standard libraries already provide a rich set of functionality, and there is relatively little need for writing additional helper methods. Second, the relatively few applications that do define helper methods that operate on collections tend to use these methods monomorphically. An investigation of larger applications might turn up more opportunities for context-sensitive analysis, but it is our conjecture that there will be relatively few such opportunities.

10 Related Work

The work most closely related to ours is that by Donovan et al. [8], who also designed and implemented a refactoring for migrating an application to a generic version of a class library that it uses. Like us, Donovan et al. evaluate their algorithm by inferring generic types for occurrences of the standard collections in a set of Java benchmarks, and measure their success in terms of the number of casts that can be removed. There are a number of significant differences between the two approaches.

First, the approach by Donovan et al. relies on a context-sensitive pointer analysis[19] based on [24, 1] to determine the types stored in each allocation site for a generic library class. Moreover, Donovan et al. create "guarded" constraints that may or may not be applied to the type constraint system depending on the rawness of a particular declaration, and their solving algorithm may require (limited) backtracking if such a rawness

[19] The context-sensitive variant of our algorithm [19] discussed in Section 9 is also based on the Cartesian Product Algorithm, but it uses context-sensitivity for a different purpose, namely to identify when it is useful to create generic methods.

decision leads to a contradiction later on. Our approach is simpler because it requires neither context-sensitive analysis nor backtracking, and therefore has greater potential for scaling to large applications. The differences in observed running times seem to bear this out (Donovan et al. report a running time of 462 seconds on the ∼27 KLOC *HTMLParser* using a 3GHz Pentium 4 with 200Mb heap, while our tool requires 113.9 seconds on a ∼90 KLOC program using a 1.6GHz Pentium M using 512Mb heap).

Second, there are several differences in the kinds of source transformations allowed in the two works: (i) Donovan et al. restrict themselves to transformations that do not affect the erasure of a class, while our approach allows the modification of declarations, (ii) Donovan's work was done prior to the release of Java 1.5 and their refactoring tool conforms to an earlier specification of Java generics, which does not contain wildcard types and which allows arrays of generic types, and (iii) our method is capable of inferring precise generic supertypes for subtypes of generic library classes that are defined in application code (see, e.g., Figure 2 in which we infer that class `MyIterator` extends `Iterator<String>`). Third, our tool is more practical because it is fully integrated in a popular integrated development environment.

For a more concrete comparison, we manually inspected the source generated by both tools for 5 of the 7 benchmarks analyzed in [8]: *JLex*, *JavaCup*, *JUnit*, *V_poker* and *TelnetD*. A head-to-head comparison on *ANTLR* and *HTMLParser* was impossible due to differences in the experimental approach taken[20].

In most cases, our tool was able to remove the same or a higher number of generics-related casts than did Donovan's, in a small fraction of the time. The differences in casts removed derive from several distinct causes. First, our tool's ability to infer type parameters for user-defined subtypes of parametric types permits the removal of additional casts (e.g., 6 additional casts could be removed in *V_poker* in clients of a local class extending `Hashtable`). Second, Donovan's tool was implemented before the final Java 1.5 specification was available and conforms to an early draft, in which parameterized types were permitted to be stored in arrays; the final specification, however, requires that such generic types be left raw. As a result, Donovan's tool infers non-raw types for certain containers in *JLex* that our tool (correctly) leaves raw, preventing the removal of certain casts. Third, our algorithm models `Object.clone()` so that type parameter information is not lost across the call boundary. As a result, our tool removes all 24 generics-related casts from *JUnit*, while Donovan's tool only removes 16.

Von Dincklage and Diwan [23] address the problems of converting non-generic Java classes to use generics (parameterization) and updating non-generic usages of generic classes (instantiation). Their approach, like ours, is based on constraints. Von Dincklage's tool employs a suite of heuristics that resulted in the successful parameterization of several classes from the Java standard collections. However, the code of those classes had to be manually modified to eliminate unhandled language constructs before the tool could be applied. The tool's correctness is based on several unsound assumptions (e.g., public fields are assumed not to be accessed from outside their class, and the type of

[20] Donovan et al. identified classes in these benchmarks that could be made generic, manually rewrote them accordingly, and treated them as part of the libraries. As a result, the number of removed casts that they report cannot be directly compared to ours, as it includes casts rendered redundant by the generics that they manually introduced.

the argument to `equals` is assumed to be identical to the receiver's type), and it can alter program behavior by modifying virtual method dispatch due to changed overriding relationships between methods. No results are given about how successful the tool is in instantiating non-generic classes with generic information.

The problem of introducing generic types into a program to broaden its use has been approached before by several researchers. Siff and Reps [18] focused on translating C functions into C++ function templates by using type inference to detect latent polymorphism. In this work, opportunities for introducing polymorphism stem from operator overloading, references to constants that can be wrapped by constructor calls, and from structure subtyping. Duggan [9] gives an algorithm (not implemented) for genericizing classes in a small Java-like language into a particular polymorphic variant of that language. This language predated the Java 1.5 generics standard by several years and differs in a nontrivial number of respects. Duggan does not address the problem of migrating non-generic code to use generics. The programming environments CodeGuide [6] and IntelliJ IDEA [12] provide "Generify'' refactorings that are similar in spirit to ours. We are not aware of the details of these implementations, nor of the quality of their results.

11 Conclusions and Future Work

We have presented a refactoring that assists programmers with the adoption of a generic version of an existing class library. The method infers actual type parameters for declarations and allocation sites that refer to generic library classes using an existing framework of type constraints. We implemented this refactoring in Eclipse, and evaluated the work by migrating a number of moderate-sized Java applications that use the Java collections framework to Java 1.5's generic collection classes. We found that, on average, 48.6% of the casts related to the use of collections can be removed, and that 91.2% of the unchecked warnings are eliminated. Our approach distinguishes itself from the state-of-the-art [8] by being more scalable and by its ability to accommodate user-defined subtypes of generic library classes. The "Infer Generic Type Arguments" in the forthcoming Eclipse 3.1 release is largely based on the concepts presented in this paper, and has adopted important parts of our implementation.

Plans for future work include the inference of wildcard types. As indicated in Section 8.3, doing so will help remove additional casts and unchecked warnings.

Acknowledgments

We are grateful to Alan Donovan and Michael Ernst for many useful discussions about comparisons between our two algorithms, and for sharing information about the benchmarks used in [8]. The anonymous ECOOP reviewers, Michael Ernst, and Bartek Klin provided many helpful comments and suggestions.

References

1. AGESEN, O. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proc. of ECOOP* (1995), pp. 2–26.

2. AGESEN, O. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Stanford University, December 1995.
3. BÄUMER, D., GAMMA, E., AND KIEŻUN, A. Integrating refactoring support into a Java development tool. In *OOPSLA'01 Companion* (October 2001).
4. BRACHA, G., COHEN, N., KEMPER, C., ODERSKY, M., STOUTAMIRE, D., THORUP, K., AND WADLER, P. Adding generics to the Java programming language, final release. Tech. rep., Java Community Process JSR-000014, September 2004.
5. BRACHA, G., ODERSKY, M., STOUTAMIRE, D., AND WADLER, P. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of OOPSLA* (1998), pp. 183–200.
6. Omnicore codeguide. `http://www.omnicore.com/codeguide.htm`.
7. DE SUTTER, B., TIP, F., AND DOLBY, J. Customization of Java library classes using type constraints and profile information. In *Proc. of ECOOP* (2004), pp. 585–610.
8. DONOVAN, A., KIEŻUN, A., TSCHANTZ, M., AND ERNST, M. Converting Java programs to use generic libraries. In *Proc. of OOPSLA* (Vancouver, BC, Canada, 2004), pp. 15–34.
9. DUGGAN, D. Modular type-based reverse engineering of parameterized types in Java code. In *Proc. of OOPSLA* (1999), pp. 97–113.
10. FOWLER, M. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.
11. GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *The Java Language Specification (3rd Edition)*. Addison-Wesley, 2000.
12. JetBrains IntelliJ IDEA. `http://www.intellij.com/idea/`.
13. IGARASHI, A., PIERCE, B. C., AND WADLER, P. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS 23*, 3 (2001), 396–450.
14. LANGER, A., AND KREFT, K. Arrays in Java Generics. Manuscript `http://www.langer.camelot.de`.
15. MUNSIL, W. Case study: Converting to Java 1.5 type-safe collections. *Journal of Object Technology 3*, 8 (2004), 7–14.
16. ODERSKY, M., AND WADLER, P. Pizza into Java: Translating theory into practice. In *Proc. of POPL* (1997), pp. 146–159.
17. PALSBERG, J., AND SCHWARTZBACH, M. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.
18. SIFF, M., AND REPS, T. W. Program generalization for software reuse: From C to C++. In *Foundations of Software Engineering* (1996), pp. 135–146.
19. TIP, F., FUHRER, R., DOLBY, J., AND KIEŻUN, A. Refactoring techniques for migrating applications to generic Java container classes. Tech. Rep. Research Report RC 23238, IBM Research, June 2004.
20. TIP, F., KIEŻUN, A., AND BÄUMER, D. Refactoring for generalization using type constraints. In *Proc. of OOPSLA* (Anaheim, CA, 2003), pp. 13–26.
21. TORGERSEN, M., HANSEN, C. P., ERNST, E., VON DER AHÉ, P., BRACHA, G., AND GAFTER, N. M. Adding wildcards to the Java programming language. In *Proc. of ACM Symposium on Applied Computing (SAC)* (Nicosia, Cyprus, 2004), pp. 1289–1296.
22. VITEK, J., HORSPOOL, R. N., AND KRALL, A. Efficient type inclusion tests. In *Proc. of OOPSLA* (1997), pp. 142–157. *SIGPLAN Notices* 32(10).
23. VON DINCKLAGE, D., AND DIWAN, A. Converting Java classes to use generics. In *Proc. of OOPSLA* (Vancouver, BC, Canada, 2004), pp. 1–14.
24. WANG, T., AND SMITH, S. Precise constraint-based type inference for Java. In *Proc. of ECOOP* (2001), pp. 99–117.