RC23238 (W0406-045) June 10, 2004
Computer Science

IBM Research Report

Refactoring Techniques for Migrating Applications to Generic Java Container Classes

Frank Tip, Robert Fuhrer, Julian Dolby
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Adam Kiezun
MIT Computer Science and AI Laboratory
32 Vassar Street
Cambridge, MA 02139

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Refactoring Techniques for Migrating Applications to Generic Java Container Classes

Frank Tip Robert Fuhrer Julian Dolby
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
{ftip,rfuhrer,dolby}@us.ibm.com

Adam Kiezun
MIT Computer Science & AI Lab
32 Vassar St
Cambridge, MA 02139 USA
akiezun@mit.edu

ABSTRACT

Version 1.5 of the Java programming language will include *generics*, a language construct for associating type parameters with classes and methods. Generics are particularly useful for creating *statically type-safe*, reusable container classes such that a store of an inappropriate type causes a compile-time error, and that no down-casting is needed when retrieving elements. The standard libraries released with Java 1.5 will include generic versions of popular container classes such as `HashMap` and `ArrayList`. This paper presents a method for *refactoring* Java programs that use current container classes into equivalent Java 1.5 programs that use their new generic counterparts. Our method uses a variation on an existing model of type constraints to infer the element types of container objects, and it is parameterized by how much, if any, context sensitivity to exploit when generating these type constraints. We present both a context-insensitive instantiation of the framework and one using a low-cost variation on Agesen's Cartesian Product Algorithm. The method has been implemented in Eclipse, a popular open-source development environment for Java. We evaluated our approach on several small benchmark programs, and found that, in all but one case, between 40% and 100% of all casts can be removed.

1. INTRODUCTION

Java's class libraries provide a range of standard container data types, such as hash-tables and lists, in the `java.util` package. These containers enhance the productivity of Java programmers by allowing them to concentrate on the aspects unique to their application without being burdened with the unexciting task of building basic infrastructure. In our experience, nearly all Java applications use these standard containers, and many use them extensively.

A limitation of the current Java container classes is that they are not *statically type safe*. That is, access methods such as `get()` and `set()` all refer to type `Object`, which means there can be no compile-time type checking to enforce a programmer's notion of what types may be stored into particular containers. This also means a down-cast to a specific type is often needed when retrieving objects from such containers. When containers are misused, these casts fail at runtime, with `ClassCastException`s.

This problem will be ameliorated by *generics* [3] in Java 1.5; *generics* allows programmers to associate type parameters with classes and methods. Generics are particularly useful for creating reusable container classes with compiler-enforced type safe usage. The standard libraries that will be released with Java 1.5 will include generic versions of the container classes that are functionally equivalent to the current ones, but use type parameters to specify each container's element type, and refer to these type parameters in their accessor methods. That is, a `Collection` will become a `Collection<T>`, with accessors that enforce the parametric type `T`, e.g. `void add(T)` and `T get(int)`. See Figure 1.

The premise of this paper is that, once generics are available, programmers will want to *refactor* applications that use current container classes into equivalent Java 1.5 programs that use their new generic counterparts. We present a 3-step approach to address this problem:

1. A low-cost variation on Agesen's Cartesian Product Algorithm [1] is used to infer a set of *contexts* for each method. Roughly speaking, each context corresponds to a different combination of container objects on which the method operates. For methods that do not manipulate container objects, only one context is used.

2. A set of *type constraints* [13] is derived from the program. These type constraints are similar to those used in previous work by some of the present authors [17, 6], but differ from that previous work by explicitly representing context information and the element types of container objects. Solving the system of constraints produces element types for declarations and allocations of container class types.

3. The solution of step 2 is used to determine where declarations and allocations that refer to container classes can be made to refer to generic types, and where down-casts are rendered unnecessary. This involves an analysis of the results computed for the different contexts of each method, and introducing generic classes and generic methods where necessary.

We have implemented this approach as as a *source-to-source* refactoring in the existing refactoring framework [2] of Eclipse[1]. We applied the refactoring to several small benchmark programs that use the standard container classes, and our findings indicate that for these small programs, in all but one case, between 40% and 100% of all casts can be removed.

[1]See `www.eclipse.org`.

```
interface Collection<T1> {
  void add(T1 e);
  <T11 extends T1> void addAll(Collection<T11> c);
  Iterator<T1> iterator();
}
interface List<T2> extends Collection<T2> {
  void add(int index, T2 e);
  T2 get(int index);
}
class Vector<T3> implements List<T3> {
  ...
}
interface Iterator<T4> {
  boolean hasNext();
  T4 next();
}
```

Figure 1: Simplified generic Collection API.

2. JAVA GENERICS

In Java 1.5, a class C may have one or more *formal type parameters* $T_1, \cdots, T_n$, which 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. Syntactically, formal type parameters follow the class name in a comma-separated list between the symbols '<' and '>'. Bounds are specified using the keyword **extends**; multiple bounds are separated by '&'. Figure 2(b) shows an example of a generic Java program. Inheritance works just as for normal classes; type parameters of the subclass may be used in **extends** declarations (see Figure 1).

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 be a subtype of all the bounds of the corresponding formal type parameter T_j. Syntactically, actual type parameters follow the class name in a constructor call, in a comma-separated list between the symbols '<' and '>'. This syntax is also used in declarations. For example, method `foo()` of class `A` in Figure 2(b) instantiates a `Vector` with actual type parameter `String`.

Type parameters may also be associated with static or non-static methods. Syntactically, the type parameters of generic methods must be supplied at the beginning of the method's signature. In Figure 2(b), method `A.reverse()` has a type parameter `T`. Inner classes inherit the type parameters of their enclosing instance(s), but may also add their own.

A few important details regarding Java generics deserve mention. Java generics are implemented by *erasure*: the Java 1.5 compiler replaces each occurrence of a type variable by its (class) bound, and inserts down-casts at the appropriate places to ensure that the program is statically type-correct. These casts are guaranteed to succeed at run-time. Unlike arrays, generic types are not covariant: $C<B>$ is a subtype of $C<A>$ if and only if $B = A$. Primitive types cannot be used as actual type parameters of a generic class. However, *autoboxing*, another new Java 1.5 feature, effectively eliminates this limitation.

Java 1.5 also provides a mechanism, called *raw types*, for instantiating and referring to a generic class without supplying actual type parameters. This is equivalent to instantiating the class with the each parameter's bound as the corresponding actual type parameter. Raw types permit Java programs to interact with legacy code that does not use generics. Section 6.2 discusses the repercussions of such interactions to our refactoring.

For more details about Java generics and their erasure semantics, we refer the reader to the specification [3], and to earlier work on the Pizza [12] and GJ [4, 11] languages.

2.1 Generic Container Classes

Figure 1 shows a small hierarchy of container classes that are similar in spirit to the containers in the generic standard libraries, but omitting many details. Our algorithms easily extend to accommodate those as well. As Figure 1 shows, `Collection`'s allow adding elements (`add()`), adding from another `Collection` (`addAll()`), and iterating over its elements (`iterator()`). `Collection` and its subtypes each have a single type parameter that designates the type of the `Collection`'s elements. `Iterator`'s also have a single type parameter representing the type of the elements being iterated over.

2.2 Example

Figure 2(a) shows a Java program containing 3 `Vector` allocation sites in methods `foo()`, `bar()`, and `baz()`, labelled `L1`–`L3`. Method `insert()` (called from `foo()`) inserts into a `List` and `reverse` (called from `bar()` and `baz()`) reverses a `Vector`. In Figure 2(a), observe the down-casts needed at lines 13 and 22 because `Iterator.next()` returns `Object`.

Figure 2(b) shows the result of our refactoring algorithm on Figure 2(a). Changes are indicated by underlining. Declarations and allocation sites make use of generics at lines 3, 8, 12, 18, 24, and 27. Moreover, the down-casts at lines 13 and 22 have been removed.

In general, inferring a precise generic type for a container may require changing types of other declarations. For example, the type of parameter `o` of `A.insert()` is changed from `Object` to `String`. This change[2] is needed to infer type `List<String>` for `v4`, `v1` and allocation site `L1` since one cannot store `Object`'s into a `List<String>`.

`A.reverse()` on line 27 illustrates how *context-sensitive* analysis may be needed to compute tight generic types. `A.reverse()` is called on both lines (11) and (21). On line (11), argument `v2` refers to a `Vector` of `String`s. On line (21), argument `v3` is a `Vector` of `Integer`s. Two approaches for declaring `v2`, `v3`, and `v4` as generic `Vector`s produce a type-correct program:

1. Use *the same* concrete[3] type T as the type parameter in the declarations of `v2`, `v3`, and `v4`. T must satisfy all type constraints imposed on `v2`, `v3`, and `v4`. In particular, T must be a supertype of `String` because of the call `v2.add(s2)` in `bar()` and it must be a supertype of `Integer` because of the call `v3.add(i1)` in `baz()`. Therefore, giving `v2`, `v3`, and `v4`, e.g., type `Vector<Object>` is correct. However, no choice of type for T permits the removal of the down-casts on lines (13) and (22).

2. Make `reverse()` a generic method, with `v4` becoming a `Vector<T>`. If `v2` is declared `Vector<String>` and `v3` `Vector<Integer>`, both can be passed correctly to `reverse` and the casts on lines (13) and (22) are obviated. Figure 2(b) illustrates this approach.

The advantage of (1) is that it is easy to compute via unification of element types, but offers limited potential for removing casts when a "helper method" is invoked at multiple sites. Approach (2) enables the removal of more casts, but requires determining that the `Vector`s that are passed to `reverse()` from `bar()` do not flow to `baz()`, and vice versa. In other words, a context-sensitive analysis is required to compute the solution of Figure 2(b).

2.3 Scope and Assumptions

[2]Other options include turning `insert()` into a generic method `<T> public void insert(List<T> v4, T o)`, or leaving the type of `v4` raw.

[3]We will use the term *concrete* type in this paper to refer to any type that is not a type parameter.

```
(1) class A {
(2)   public void foo(){
(3)     Vector v1 = new Vector^L1();
(4)     String s1 = new String("aaa");
(5)     this.insert(v1, s1);
(6)   }
(7)   public void bar(){
(8)     List v2 = new Vector^L2();
(9)     String s2 = new String("bbb");
(10)    v2.add(s2);
(11)    this.reverse(v2);
(12)    for (Iterator it = v2.iterator();
              it.hasNext(); ){
(13)      String s3 = (String)it.next();
(14)      System.out.println(s3);
(15)    }
(16)  }
(17)  public void baz(){
(18)    List v3 = new Vector^L3();
(19)    Integer i1 = new Integer(17);
(20)    v3.add(i1);
(21)    this.reverse(v3);
(22)    Integer i2 = (Integer)v3.iterator().next();
(23)  }
(24)  public void insert(List v4, Object o){
(25)    v4.add(o);
(26)  }
(27)  public void reverse(Vector v5){
(28)    for (int t=0; t < v5.size()/2; t++){
(29)      Object temp = v5.get(v5.size()-t);
(30)      v5.add(v5.size()-1, v5.get(t));
(31)      v5.add(t, temp);
(32)    }
(33)  }
(34)}
```

(a)

```
(1) class A {
(2)   public void foo(){
(3)     Vector<String> v1 = new Vector<String>^L1();
(4)     String s1 = new String("aaa");
(5)     this.insert(v1, s1);
(6)   }
(7)   public void bar(){
(8)     List<String> v2 = new Vector<String>^L2();
(9)     String s2 = new String("bbb");
(10)    v2.add(s2);
(11)    this.reverse(v2);
(12)    for (Iterator<String> it = v2.iterator();
              it.hasNext(); ){
(13)      String s3 = it.next();
(14)      System.out.println(s3);
(15)    }
(16)  }
(17)  public void baz(){
(18)    List<Integer> v3 = new Vector<Integer>^L3();
(19)    Integer i1 = new Integer(17);
(20)    v3.add(i1);
(21)    this.reverse(v3);
(22)    Integer i2 = v3.iterator().next();
(23)  }
(24)  public void insert(List<String> v4, String o){
(25)    v4.add(o);
(26)  }
(27)  <T> public void reverse(Vector<T> v5){
(28)    for (int t=0; t < v5.size()/2; t++){
(29)      T temp = v5.get(v5.size()-t);
(30)      v5.add(v5.size()-1, v5.get(t));
(31)      v5.add(t, temp);
(32)    }
(33)  }
(34)}
```

(b)

Figure 2: (a) Example program that uses non-generic container classes. (b) Refactored version of the program of (a). Here, underlining is used to indicate declarations and allocation sites for which a different type is referred, and assignment statements in which casts have been removed.

In the remainder of this paper, we assume that the original program is type-correct, and, moreover, does not contain any up-casts (i.e., casts $(C)E$ in which the type of E is a subclass of C)[4].

Furthermore, we assume that the original program does not contain any generic types (primarily because the Eclipse infrastructure that our implementation relies upon does not support them yet). Finally, we assume that user programs do not define subclasses of `Collection`.

3. CONTEXT INFERENCE

This section describes a context-sensitive points-to analysis that computes: (i) a set of *contexts* for each method, (ii) for each expression, a points-to set for that expression in each of the contexts of its containing method, and (iii) calling relations between a call site (for a given context of its containing method), and contexts of methods reachable from that call site. The resulting context-sensitive call graph and points-to information serve as input for the type inference algorithm described in the next section.

3.1 Classification of the Analysis

The points-to analysis that we will use is a subset-based flow-insensitive, context-sensitive, *field-sensitive*[5] [15], points-to analysis that is a variation on Agesen's Cartesian Product Algorithm [1] in which distinct allocation sites are maintained for container-related types, but where all other allocation sites are unified into a single logical allocation site.

We use a *model* of the container classes in which a single class `CollectionModel` represents all subtypes of `java.util.Collection`. This class `CollectionModel` has a single instance field `elem`, and calls to methods such as `Collection.add()` and `List.get()` are modeled as writing and reading this field, respectively. This container class model deserves a few additional remarks. In particular, the `Iterator` objects that are returned by methods in a container class (e.g., `Vector.iterator()`) are modeled by identifying the returned iterator object with the container object that is being iterated over. Furthermore, container methods that combine containers (e.g., `Collection.addAll()`) are modeled using assignments between their `elem` fields.

Several important pragmatic issues arise, including the treatment of subtypes of `Map` and array types, and dealing with incomplete applications. With respect to the latter issue, we make a distinction between *application code* for which source code is available, and *external code* (including that in the JDK libraries) for which it is not. Our approach for dealing with these issues is discussed in Section 6.2.

3.2 Notation and Terminology

In what follows, m, m' denote methods, f, f' denote fields, C, C' denote classes, I, I' denote interfaces, T, T' denote types[6], and

[4]Up-casts are only needed in rare cases for the explicit resolution of overloaded methods and shadowed fields, and their treatment is analogous to that of down-casts.

[5]That is, each field is analyzed separately for each allocation site that contains it.

[6]In this paper, the term *type* will denote a class or an interface, and the subtype relationship '$\leq$' is derived from the program's class hierarchy.

X, X' denote type variables. Moreover, the notation E, E' will be used to denote an expression or declaration, corresponding to a specific node in the program's abstract syntax tree. It is assumed that type information about expressions is available from the compiler.

A method m is *virtual* if m is not a constructor, not private and not static. A virtual method m in type C *overrides* a virtual method m' in type B if m and m' have identical signatures and C is equal to B or C is a subtype of B. In this case, we also say that m' is overridden by m. Note that, using this definition, a virtual method overrides itself.

3.3 Information Computed

We will use a set of *labels* $\mathcal{L}$ to identify occurrences of the constant `null` and allocation sites. Specifically, we assume each allocation site $E \equiv$ `new T`$(E_1, \cdots, E_k)$ to be labeled with a unique label $L \in \mathcal{L}$ if T is a subtype of `Collection`, and that each occurrence of the constant `null` is labeled similarly. Furthermore, a distinct label $L_{ext} \in \mathcal{L}$ will be used to represent all container objects that are allocated outside of the application[7]. Finally, a single "blob" label $\bullet \in \mathcal{L}$ is used to represent the allocation sites of all types that are not subtypes of `Collection`.

Our algorithm computes, for each method $T.m(T_1, \cdots, T_n)$, a set of contexts $\mathit{Contexts}(T.m(T_1, \cdots, T_n))$, where each context $\alpha \in \mathit{Contexts}(m(T_1, \cdots, T_n))$ is a list of the form $[p_1, \cdots, p_n]$ and where $p_i \in \mathcal{L}$, $(1 \leq i \leq n)$. Intuitively, each p_i identifies the allocation site of objects bound to the i^{th} formal parameter of its method. For virtual methods and constructors, the `this` pointer is considered to be the method's first parameter.

The inference rules of Figure 3 also determine for each expression E that occurs in method m a set of objects $\mathit{Objects}_\alpha(E) \subseteq 2^{\mathcal{L}}$ that expression E may point to when m is executed in context α. Points-to sets are also computed for fields. For each field $T.f$, let $p_1, \cdots, p_n$ be the allocation sites of type T. Then, the value stored in field f of the objects allocated at allocation site p_i will be denoted by $\mathit{Objects}_{p_i}(f) \subseteq 2^{\mathcal{L}}$. Note that, because the allocation sites of all non-container types are represented by the logical allocation site $\bullet$, only one set $\mathit{Objects}_\bullet(f)$ will be computed for any field f that is declared in a non-container type.

3.4 Description of the Inference Rules

Our analysis assumes that the user has specified a number of entry point methods that serve as the entry points for the analysis. Definition 3.1 below defines the set of objects that can be bound to a parameter of an entry point method. There are three cases: (i) if the type of a formal parameter is a subtype of `Collection`, then it is made L_{ext}, (ii) if the type of a formal parameter is a primitive or class type that *cannot* be a `Collection`, then it is made bound $\bullet$, and (iii) otherwise it can be bound to $\bullet$ and L_{ext}.

Def. 3.1 (*ExternalObjects*). *Let T be a type. Define:*

$$\mathit{ExternalObjects}(T) = \begin{cases} \{ L_{ext} \} & \text{if } T \leq \texttt{Collection} \\ \{ \bullet \} & \text{if } T \not\leq \texttt{Collection} \\ \{ L_{ext}, \bullet \} & \text{otherwise} \end{cases}$$

Rule **(C1)** of Figure 3 defines the set of contexts for method $T.m(T_1, \cdots, T_k)$ using these type estimates. Rule **(C2)** defines the points-to relations for the method's formal parameters based on these type estimates. Here, the auxiliary notion $\mathrm{Param}(m', i)$ is used to refer to the expression that constitutes the i^{th} formal parameter of m' (`this` is considered to be the first formal parameter of m').

[7] Alternatively, one could create a more fine-grained model by analyzing the byte-codes for unavailable classes, similar to [7].

Rule **(C3)** models the flow of values through assignment statements using a subset relationship of their points-to sets. Rule **(C4)** is concerned with container allocation expressions of the form $E \equiv$ `new T`^{L} $(E_1, \cdots, E_k)$ (with $T \leq$`java.util.Collection`) that occur in a method m. The rule states that object L is a member of $\mathit{Objects}_\alpha(E)$, for each $\alpha \in \mathit{Contexts}(m)$. Rules **(C5)–(C7)** are concerned with constructor calls of the form $E_0 \equiv$ `new T`$(E_1, \cdots, E_k)$ to a constructor method m', where $T \not\leq$`java.util.Collection`. For each method m in which such an expression occurs, and each $\alpha \in \mathit{Contexts}(m)$, we first determine the contexts α' associated with the constructor method, using the auxiliary function *SelectContexts* (see Definition 3.2). *SelectContexts* takes two arguments: a context α of the calling method, and the actual parameters expressions $E_0, \cdots, E_k$, and creates contexts α' for m' based on the points-to sets associated with these actual parameters in context α. Rule **(C5)** adds these contexts α' to the set of contexts associated with m'.

Def. 3.2 (*SelectContexts*). *Let $E_0, \cdots, E_k$ be expressions.*

$$\mathit{SelectContexts}(\alpha, E_0, \cdots, E_k) = \{ [p_0, \cdots, p_k] \mid p_i \in \mathit{Objects}_\alpha(E_i),\ 0 \leq i \leq k \}$$

The points-to set associated with the entire allocation expression is $\bullet$ because T is not a container type (see Rule **(C6)**). Rule **(C7)** models the effect on points-to relationships of binding the actual parameter expressions $E_0, \cdots, E_k$ to the corresponding formal parameters of m'.

Modeling the effect of direct and virtual method calls on context creation and points-to sets is defined by rules **(C8)–(C10)** and **(C11)–(C13)**, respectively. These rules are very similar to those for constructor calls except for the fact that there is no allocated object that must be included in the points-to set for the call expression. Moreover, passing of return values from callees to callers is modeled by associating an additional parameter $\texttt{return}_m$ with each method m, and generating an inclusion constraint involving $\texttt{return}_m$ and the call expression. For virtual method calls, dynamic dispatch is approximated by assuming that a virtual call to method m' can resolve to any method m'' that overrides m'. This effectively amounts to using Class Hierarchy Analysis (CHA) [9] for virtual call resolution.

Rule **(C14)** is concerned with `return` statements of the form `return` E that occur in a method m for which $\alpha \in \mathit{Contexts}(m)$. In this case, the set of objects associated with E in context α is a subset of the set of objects associated with m's return parameter in context α.

Rules **(C15)** and **(C16)** handle field accesses. These generate inclusion constraints for the rval or lval expression respectively from or to the appropriate field cell for each allocation site of the container E'. Rule **(C17)** states that the set of objects associated with a cast expression $E \equiv (T)E'$ is a subset of set of objects associated with E'. Rules **(C18)** and **(C19)** define the points-to sets associated with literal values. Literal references besides `null` cannot flow to `Collection` typed expression, so they are all defined to be $\bullet$ (Rule **(C18)**). Some `null` literals may be assigned to `Collection` typed variables, so we represent them as allocation sites (Rule **(C19)**).

3.5 Example

Figure 2(a) shows an example program that contains methods `foo()`, `bar()`, `baz()`, `insert()`, and `reverse()` and three container allocation sites (labeled `L1`—`L3`) as well as three (unlabeled) allocation sites that are not container-related.

Table 1 shows all contexts inferred for the example program, assuming that all 5 methods have been designated as en-

method	context
`A.foo()`	$\alpha_1 \equiv [\bullet]$
`A.bar()`	$\alpha_2 \equiv [\bullet]$
`A.baz()`	$\alpha_3 \equiv [\bullet]$
`A.insert(Vector,Object)`	$\alpha_4 \equiv [\bullet, L_{ext}, \bullet]$
`A.insert(Vector,Object)`	$\alpha_5 \equiv [\bullet, L_{ext}, L_{ext}]$
`A.insert(Vector,Object)`	$\alpha_7 \equiv [\bullet, L1, \bullet]$
`A.reverse(Vector)`	$\alpha_6 \equiv [\bullet, L_{ext}]$
`A.reverse(Vector)`	$\alpha_8 \equiv [\bullet, L2]$
`A.reverse(Vector)`	$\alpha_9 \equiv [\bullet, L3]$

Table 1: Contexts inferred for the example program.

try points, and using *ExternalObjects*(`Vector`) $= \{ L_{ext} \}$ and *ExternalObjects*(`Object`) $= \{ \bullet, L_{ext} \}$ (see Definition 3.1).

4. TYPE INFERENCE

Once contexts have been inferred, a set of type constraints is generated for each context of each method, using an adaptation of the formalism of [17, 13]. Solving the resulting set of type constraints produces inferred types for declarations and allocation sites of container classes.

For each program construct, a set of type constraints specifies the relationships that must exist among the types of the construct's constituent expressions for the construct to be type-correct. By definition, a program is *type-correct* if the declared types satisfy the constraints of all of its constructs. Type constraints are expressed in the following notation. For a declaration or expression E, $[E]_\alpha$ denotes the type of E in context α. Furthermore, $[f]_L$ denotes the type of field f in objects with label L, and $[m]_\alpha$ denotes the return type of method m in context α. For a field f or a method m, *Decl*(f) and *Decl*(m) denote the type in which f or m is declared, respectively. Finally, for types T and T', $T' \leq T$ denotes that T' is equal to T, or T' is a subtype of T. Definition 4.1 defines, for a given method m, the set *RootDefs*(m) of methods m' that are overridden by m but do not override any methods but themselves. Since the original program is assumed type-correct, this set is guaranteed to be non-empty.

Def. 4.1 (RootDefs) . *Let m be a method. Define:*

$$RootDefs(m) = \{ m' \mid \begin{array}{l} m \text{ overrides } m', \text{ and there exists} \\ \text{no } m'' \ (m'' \neq m') \text{ such that} \\ m' \text{ overrides } m'' \end{array} \}$$

Def. 4.2. *Let T be a type. Define:*

$$NewType(T) = \begin{cases} T\langle X \rangle & \text{if } T \leq \texttt{Collection} \\ T & \text{otherwise} \end{cases}$$

A *constraint variable* c is either T (a type constant) or $[E]$ (the type of a declaration or expression E). A *type constraint* is a relationship between two or more constraint variables that must hold in order for a program to be type-correct. In this paper, a type constraint has one of the following forms: (i) $c_1 \triangleq c_2$, indicating that c_1 is defined to be the same as c_2 (ii) $c_1 \leq c_2$, indicating that c_1 must be equal to or be a subtype of c_2, (iii) $c_1 = c_2$, indicating that $c_1 \leq c_2$ and $c_2 \leq c_1$, or (iv) $c_1^L \leq c_1^R$ **or** $\cdots$ **or** $c_k^L \leq c_k^R$, indicating that $c_j^L \leq c_j^R$ must hold for at least one j, $1 \leq j \leq k$.

In discussions about types and subtype-relationships that occur in a specific program P, we will use the same notation as that for constraint variables with subscript P. For example, $[E]_P$ denotes the type of expression E in program P, and $T' \leq_P T$ denotes a subtype-relationship that occurs in program P. In cases where the program under consideration is unambiguous, we will frequently omit these P-subscripts.

4.1 Deriving Type Constraints

Figure 4 shows a set of inference rules (adapted from [13]) that infer type constraints for several important Java constructs, such as assignments, constructor calls, virtual method calls, field accesses, and cast expressions. Due to space limitations, we only describe a few rules in detail.

Constraint (B1) concerns assignments $E_1 = E_2$ that occur in method m such that $\alpha \in Contexts(m)$, and states that the assignment is type correct if the type of E_2 in context α is the same as or a subtype of that of E_1 in context α.

Rules (B6)–(B8) concern virtual calls of the form $E \equiv E_0.n(E_1, \cdots, E_k)$ that occur in method m for which $\alpha \in Contexts(m)$. We assume that α' is one of the contexts of callee m', and that m'' is any method that overrides m'. Rule (B6) defines the type of the call expression E in context α to be the return type of callee m' in context α'. Rule (B7) imposes the proper subtype-relationships between corresponding actual and formal parameters (including the `this` pointer). Finally, rule (B8) concerns the relationship between the type of receiver E_0 in context α and the root definition types $T_1, \cdots, T_q$ in which methods are declared that are overridden by m'. Because the behavior of the virtual method call only depends on the run-time type of E_0, we may change the declared type of E_0 to any subtype of any T_i without affecting behavior. This is expressed by using Definition 4.1 to compute the types $T_1, \cdots, T_q$, and generating an **or** constraint that requires the type of E_0 in context α to be a subtype of at least one T_i.

Rules (B11) and (B12) generate constraints for down-cast expressions of the form $E \equiv (T)E'$. Rule (B11) requires that the type of the cast expression E in context α is T. That is, the "target type" of the cast cannot be changed. This requirement, along with the fact that we change neither the types of allocation sites nor data-flow, guarantees that the run-time behavior of down-casts will be unchanged.

Rules (B14)—(B27) in Figure 4 concern the inference of `Collection` element types. For each expression E that occurs in method m for which $\alpha \in Contexts(m)$, these rules define a set $Types_\alpha(E)$ of types that may be stored in containers that E may point to in context α, and a type $Elem_\alpha(E)$ that is an upper bound of the types in $Types_\alpha(E)$ (the relationship between $Elem_\alpha(E)$ and $Types_\alpha(E)$ is expressed by rule (B24)). As an example of a `Collection`-related rule, consider Rule (B16), concerning calls to `Collection.add()`. For a call $E \equiv E_0.add(E_1)$ to method m' occurring in method m for which $\alpha \in Contexts(m)$, this rule adds the type of E_1 in context α to the set $Types_\alpha(E_0)$ of types that may be stored in `Collection` objects bound to E_0 in α. Figure 4 only shows rules for a representative subset of the Java Collections API; the remainder is handled similarly.

It is important to note that the type constraint rules (specifically, (B1), (B26), and (B27)) are carefully designed to allow specific container types such as `Vector` to be assigned to more general container types such as `List`, provided that the element types are the same. This reflects the fact that, e.g., `Vector<T>` is a subtype of `List<T>` for all `T`.

As an example, Figure 5 shows the type constraints generated for context α_9 of method `A.reverse()` of Figure 2.

5. SOURCE CODE TRANSFORMATION

Once contexts and type constraints have been inferred, the last step is to transform the program's source code. In the context-insensitive approach, this involves: (i) rewriting declarations and allocation sites to reflect the new types inferred from the constraint system, (ii) removing casts that have been rendered redundant, and

$$\frac{T_0.m(T_1,\cdots,T_n) \text{ is an entry point, } p_i \in \mathit{ExternalObjects}(T_i), \alpha = [p_0,\cdots,p_n], 1 \le i \le n}{\alpha \in \mathit{Contexts}(m(T_0,\cdots,T_n))} \quad \textbf{(C1)}$$
$$p_i \in \mathit{Objects}_\alpha(\text{Param}(m(T_0,\cdots,T_n),i)) \quad \textbf{(C2)}$$

$$\frac{m \text{ contains assignment } E_1 = E_2, \alpha \in \mathit{Contexts}(m)}{\mathit{Objects}_\alpha(E_2) \subseteq \mathit{Objects}_\alpha(E_1)} \quad \textbf{(C3)}$$

$$\frac{m \text{ contains call } E \equiv \texttt{new}\, T^L(E_1,\cdots,E_n) \text{ to constructor } m', T \le \texttt{Collection}, \alpha \in \mathit{Contexts}(m)}{L \in \mathit{Objects}_\alpha(E)} \quad \textbf{(C4)}$$

$$\frac{\begin{array}{c}m \text{ contains call } E_0 \equiv \texttt{new}\, T^L(E_1,\cdots,E_n) \text{ to constructor } m', T \not\le \texttt{Collection},\\ \alpha \in \mathit{Contexts}(m), \alpha' \in \mathit{SelectContexts}(\alpha, E_0,\cdots,E_n), 0 \le i \le n\end{array}}{\begin{array}{c}\alpha' \in \mathit{Contexts}(m') \quad \textbf{(C5)}\\ \bullet \in \mathit{Objects}_\alpha(E_0) \quad \textbf{(C6)}\\ \mathit{Objects}_\alpha(E_i) \subseteq \mathit{Objects}_{\alpha'}(\text{Param}(m',i)) \quad \textbf{(C7)}\end{array}}$$

$$\frac{m \text{ contains direct call } E \equiv T.m(E_1,\cdots,E_n), \alpha \in \mathit{Contexts}(m), \alpha' \in \mathit{SelectContexts}(\alpha, E_1,\cdots,E_n), 1 \le i \le n}{\begin{array}{c}\alpha' \in \mathit{Contexts}(m') \quad \textbf{(C8)}\\ \mathit{Objects}_\alpha(E_i) \subseteq \mathit{Objects}'_\alpha(\text{Param}(m',i)) \quad \textbf{(C9)}\\ \mathit{Objects}_{\alpha'}(\texttt{return}_{m'}) \subseteq \mathit{Objects}_\alpha(E) \quad \textbf{(C10)}\end{array}}$$

$$\frac{\begin{array}{c}m \text{ contains virtual call } E \equiv E_0.n(E_1,\cdots,E_n) \text{ to method } m',\\ \alpha \in \mathit{Contexts}(m), m'' \text{ overrides } m', \alpha' \in \mathit{SelectContexts}(\alpha, E_0,\cdots,E_k), 0 \le i \le n\end{array}}{\begin{array}{c}\alpha' \in \mathit{Contexts}(m'') \quad \textbf{(C11)}\\ \mathit{Objects}_\alpha(E_i) \subseteq \mathit{Objects}_{\alpha'}(\text{Param}(m'',i)) \quad \textbf{(C12)}\\ \mathit{Objects}_{\alpha'}(\texttt{return}_{m''}) \subseteq \mathit{Objects}_\alpha(E) \quad \textbf{(C13)}\end{array}}$$

$$\frac{m \text{ contains expression } \texttt{return}\ E, \alpha \in \mathit{Contexts}(m)}{\mathit{Objects}_\alpha(E) \subseteq \mathit{Objects}_\alpha(\texttt{return}_m)} \quad \textbf{(C14)}$$

$$\frac{m \text{ contains a read } E \equiv E'.x \text{ from field } f, \alpha \in \mathit{Contexts}(m), L \in \mathit{Objects}_\alpha(E')}{\mathit{Objects}_L(f) \subseteq \mathit{Objects}_\alpha(E)} \quad \textbf{(C15)}$$

$$\frac{m \text{ contains write } E \equiv E'.x \text{ to field } f, \alpha \in \mathit{Contexts}(m), L \in \mathit{Objects}_\alpha(E')}{\mathit{Objects}_\alpha(E) \subseteq \mathit{Objects}_L(f)} \quad \textbf{(C16)}$$

$$\frac{m \text{ contains cast expression } E \equiv (T)E', \alpha \in \mathit{Contexts}(m)}{\mathit{Objects}_\alpha(E') \subseteq \mathit{Objects}_\alpha(E)} \quad \textbf{(C17)}$$

$$\frac{m \text{ contains numeric/boolean/string constant expression } E, \alpha \in \mathit{Contexts}(m)}{\bullet \in \mathit{Objects}_\alpha(E)} \quad \textbf{(C18)}$$

$$\frac{m \text{ contains expression } E \equiv \texttt{null}^L, \alpha \in \mathit{Contexts}(m)}{L \in \mathit{Objects}_\alpha(E)} \quad \textbf{(C19)}$$

Figure 3: Rules for context inference.

(iii) rewriting certain object equality and type tests that are guaranteed to fail and that would lead to type-errors if left untransformed. The context-sensitive approach has the additional step of inserting type parameters into method signatures to allow multiple contexts with different inferred types to map to the same transformed method. We present the context-insensitive approach first, and then address the additional issues of the context-sensitive approach.

5.1 Context-Insensitive Code Generation

In the context-insensitive approach, the types of declarations and allocation sites are updated to reflect the types inferred from the constraint system. For a container-related declaration or allocation site E, this involves adding a type parameter $\mathit{Elem}(E)$ [8]. Note that declarations that do not refer to container types in the original program may be rewritten as well. In the example of Figure 2, the type of parameter `o` of method `A.insert()` was changed from `Object` to `String`.

We remove any down-cast $(T)E$ for which we infer that $[E]$ is equal to, or a subtype of T. For the example of Figure 2, the cast `(String)it.next()` on line (13) is removed because we inferred that $[\texttt{it.next()}] = \mathit{Elem}(\texttt{it}) = \texttt{String}$, which is the same as the target type of the cast, `String`

[8] If $\mathit{Elem}(E)$ is bound to a type variable, we associate a new type parameter with the method. Such situations may occur when analyzing incomplete applications or class libraries.

It is possible that when we tighten a declared type, the types of the operand expressions of operators such as `==` and `instanceof` and casts may become incomparable. Then offending construct is rewirtten into a boolean constant `false` (for `==` or `instanceof`) or to an expression `throw new ClassCastException()` (for down-casts).

5.2 Context-Sensitive Code Generation

With context-sensitive analysis, a method can have multiple contexts with different (element) types for a parameter. If a type parameters can be introduced for this argument, a single generic method can "fit□ the different contexts. We add type parameters when different *Elem* types are inferred for a `Collection`-typed parameter in different contexts. Since the Java type system places many painful restrictions on the use of generic types due to its erasure semantics, this limits when parameters can be introduced. In such cases, the more-restrictive context-insensitive solution is applied to parts of the program by unifying the results obtained for different contexts. In this section, we first present the criteria for introducing type parameters, and then discuss the code transformation. As we do so, we will illustrate the operations on the `reverse()` method of Figure 2.

The first step is to determine the set of declarations for which type parameters would ideally be introduced. This comprises any parameter v of a method m for which different types $\mathit{Elem}(v)$ are

$$\frac{m \text{ contains assignment } E_1 = E_2,\ \alpha \in \mathit{Contexts}(m)}{[E_2]_\alpha \leq [E_1]_\alpha} \quad \textbf{(B1)}$$

$$\frac{\begin{array}{c}m \text{ contains constructor call } E_0 \equiv \texttt{new}\ T(E_1, \cdots, E_k) \text{ to constructor } m',\ T \not\leq \texttt{Collection},\\ \alpha \in \mathit{Contexts}(m),\ \alpha' \in \mathit{SelectContexts}(\alpha, E_0, \cdots, E_k),\ E'_i = \text{Param}(m', i),\ 0 \leq i \leq k\end{array}}{\begin{array}{l}[E_0]_\alpha \triangleq T \quad \textbf{(B2)}\\ [E_i]_\alpha \leq [E'_i]_{\alpha'} \quad \textbf{(B3)}\end{array}}$$

$$\frac{\begin{array}{c}m \text{ contains direct call } E \equiv T.n(E_1, \cdots, E_k) \text{ to method } m',\ T \not\leq \texttt{Collection},\\ \alpha \in \mathit{Contexts}(m),\ \alpha' \in \mathit{SelectContexts}(\alpha, E_1, \cdots, E_k),\ E'_i = \text{Param}(m', i),\ 1 \leq i \leq k\end{array}}{\begin{array}{l}[E]_\alpha \triangleq [m']_{\alpha'} \quad \textbf{(B4)}\\ [E_i]_\alpha \leq [E'_i]_{\alpha'} \quad \textbf{(B5)}\end{array}}$$

$$\frac{\begin{array}{c}m \text{ contains call } E \equiv E_0.n(E_1, \cdots, E_k) \text{ to virtual method } m',\ \alpha \in \mathit{Contexts}(m),\ m'' \text{ overrides } m',\\ \mathit{RootDefs}(m') = \{\ T_1, \cdots, T_q\ \},\ \forall j (1 \leq j \leq q) : T_j \not\leq \texttt{Collection},\\ \alpha' \in \mathit{SelectContexts}(\alpha, E_0, \cdots, E_k),\ E'_i = \text{Param}(m'', i),\ 0 \leq i \leq k\end{array}}{\begin{array}{l}[E]_\alpha \triangleq [m']_{\alpha'} \quad \textbf{(B6)}\\ [E_i]_\alpha \leq [E'_i]_{\alpha'} \quad \textbf{(B7)}\\ [E_0]_\alpha \leq T_1 \textbf{ or } \cdots \textbf{ or } [E_0]_\alpha \leq T_q \quad \textbf{(B8)}\end{array}}$$

$$\frac{m \text{ contains field access } E \equiv E'.f \text{ to field } F,\ L \in \mathit{Objects}_\alpha(E'),\ T = \mathit{Decl}(f),\ T \not\leq \texttt{Collection},\ \alpha \in \mathit{Contexts}(m)}{\begin{array}{l}[E]_\alpha \triangleq [F]_L \quad \textbf{(B9)}\\ [E']_\alpha \leq T \quad \textbf{(B10)}\end{array}}$$

$$\frac{m \text{ contains down-cast expression } E \equiv (T)E',\ T \not\leq \texttt{Collection},\ \alpha \in \mathit{Contexts}(m)}{[E]_\alpha \triangleq T} \quad \textbf{(B11)}$$

$$\frac{m \text{ contains down-cast expression } E \equiv (T)E',\ T \not\leq \texttt{Collection},\ \alpha \in \mathit{Contexts}(m),\ T \text{ is a class},\ [E']_P \text{ is a class}}{T \leq [E']_\alpha} \quad \textbf{(B12)}$$

$$\frac{m \text{ contains an expression } E \equiv \texttt{this},\ T = \mathit{Decl}(m)}{E \triangleq T} \quad \textbf{(B13)}$$

$$\frac{m \text{ contains an expression } E \equiv \texttt{new}\ T(),\ T \leq \texttt{Collection},\ \alpha \in \mathit{Contexts}(m),\ T' = \mathit{NewType}(T)}{[E]_\alpha \triangleq T'} \quad \textbf{(B14)}$$

$$\frac{\begin{array}{c}m \text{ contains a call } E_0.n(E_1) \text{ to method } m',\ \alpha \in \mathit{Contexts}(m),\\ \mathit{RootDefs}(m') = \{\ T_1, \cdots, T_k\ \},\ 1 \leq i \leq k,\ (T_i \leq \texttt{Collection} \text{ or } T_i \leq \texttt{Iterator}),\ T'_i = \mathit{NewType}(T_i)\end{array}}{[E_0]_\alpha \leq T'_1 \textbf{ or } \cdots \textbf{ or } [E_0]_\alpha \leq T'_k} \quad \textbf{(B15)}$$

$$\frac{m \text{ contains a call } E_0.add(E_1) \text{ to method } m',\ \alpha \in \mathit{Contexts}(m),\ \mathit{Decl}(m') \leq \texttt{Collection}}{[E_1]_\alpha \in \mathit{Types}_\alpha(E_0)} \quad \textbf{(B16)}$$

$$\frac{m \text{ contains a call } E_0.add(\mathit{Int}, E_1) \text{ to method } m',\ \alpha \in \mathit{Contexts}(m),\ \mathit{Decl}(m') \leq \texttt{Collection}}{[E_1]_\alpha \in \mathit{Types}_\alpha(E_0)} \quad \textbf{(B17)}$$

$$\frac{m \text{ contains a call } E \equiv E_0.get(\mathit{Int}) \text{ to method } m',\ \alpha \in \mathit{Contexts}(m),\ \mathit{Decl}(m') \leq \texttt{Collection}}{[E]_\alpha \triangleq \mathit{Elem}_\alpha(E_0)} \quad \textbf{(B18)}$$

$$\frac{m \text{ contains a call } E_0.addAll(E_1) \text{ to method } m',\ \alpha \in \mathit{Contexts}(m),\ \mathit{Decl}(m') \leq \texttt{Collection}}{\mathit{Elem}_\alpha(E_1) \leq \mathit{Elem}_\alpha(E_0)} \quad \textbf{(B19)}$$

$$\frac{m \text{ contains a call } E \equiv E_0.\mathit{iterator}() \text{ to method } m',\ \alpha \in \mathit{Contexts}(m),\ \mathit{Decl}(m') \leq \texttt{Collection}}{\mathit{ItElem}_\alpha(E) \triangleq \mathit{Elem}_\alpha(E_0)} \quad \textbf{(B20)}$$

$$\frac{m \text{ contains a call } E \equiv E_0.\mathit{next}() \text{ to method } m',\ \alpha \in \mathit{Contexts}(m),\ \mathit{Decl}(m') \leq \texttt{Iterator}}{[E]_\alpha \triangleq \mathit{ItElem}_\alpha(E_0)} \quad \textbf{(B21)}$$

$$\frac{m \text{ contains down-cast expression } E \equiv (T)E',\ T \leq \texttt{Collection},\ \alpha \in \mathit{Contexts}(m),\ T' = \mathit{NewType}(T)}{[E]_\alpha \triangleq T'} \quad \textbf{(B22)}$$

$$\frac{\begin{array}{c}m \text{ contains down-cast expression } E \equiv (T)E',\ T \leq \texttt{Collection},\ \alpha \in \mathit{Contexts}(m),\\ T \text{ is a class},\ [E']_P \text{ is a class},\ T' = \mathit{NewType}(T)\end{array}}{T' \leq [E']_\alpha} \quad \textbf{(B23)}$$

$$\frac{T \in \mathit{Types}_\alpha(E)}{T \leq \mathit{Elem}_\alpha(E)} \quad \textbf{(B24)}$$

$$\mathit{Elem}_\alpha(T_0\texttt{<}T_1\texttt{>}) \triangleq T_1 \quad \textbf{(B25)}$$

$$\frac{\mathit{Elem}_\alpha(T_1) = T_2}{[T_1]_\alpha \leq \texttt{Collection<}T_2\texttt{>}} \quad \textbf{(B26)}$$

$$\frac{[E_1]_\alpha \leq [E_2]_{\alpha'}}{\mathit{Elem}_\alpha(E_1) = \mathit{Elem}_{\alpha'}(E_2)} \quad \textbf{(B27)}$$

Figure 4: Inference rules for deriving type constraints from various Java constructs.

inferred for multiple contexts associated with m. Since overriding requires identical argument signatures, we examine the contexts associated with any method that has an overriding relationship with m, as specified in Definition 5.1.

Def. 5.1. *MethodSet*(m) =

$$\left\{ m_i \;\middle|\; \begin{array}{l} m_i = m \vee \\ \exists m_j \left(\begin{array}{l} m_j \in MethodSet(m) \wedge \\ (\ m_j \text{ overrides } m_i \vee m_i \text{ overrides } m_j\ ) \end{array} \right) \end{array} \right\}$$

For method `A.reverse()` in Figure 2, we have that *MethodSet*(`A.reverse()`) = { `A.reverse()` }.

Definition 5.2 defines, for method m, the subset of its parameters for which different element types are inferred in different contexts associated.

Def. 5.2 (TypeParameterCandidates) . *TPC*(m) =

$$\left\{ p_i \;\middle|\; \begin{array}{l} p_i = \text{Param}(m, i) \wedge \\ \exists \alpha_1, \alpha_2, m_1, m_2 \\ \left(\begin{array}{l} m_1 \in \textit{MethodSet}(m) \wedge m_2 \in \textit{MethodSet}(m) \wedge \\ \alpha_1 \in \textit{Contexts}(m_1) \wedge \alpha_2 \in \textit{Contexts}(m_2) \wedge \\ \textit{Elem}_{\alpha_1}(\text{Param}(m_1, i)) \neq \textit{Elem}_{\alpha_2}(\text{Param}(m_2, i)) \end{array} \right) \end{array} \right\}$$

For method `A.reverse()` in Figure 2, we previously inferred the contexts $\alpha_6 = [\bullet, L_{ext}]$, $\alpha_8 = [\bullet, L2]$, and $\alpha_9 = [\bullet, L3]$. For parameter `v5` of `reverse()`, we have $Elem_{\alpha_6}$(`v5`) = `Object`[9], $Elem_{\alpha_8}$(`v5`) = `String`, and $Elem_{\alpha_9}$(`v5`) = `Integer`. Therefore, `v5` $\in$ *TPC*(`reverse()`).

Giving a method parameter a type parameter may require altering types within the method to also use that type parameter. Specifically, if E is given type parameter T, then expressions E' to which E is assigned must also be given type parameter T. Furthermore, expressions E'' which may also be assigned to E' must also be given type parameter T. This is expressed by Definition 5.3 below. Here, $\leq^*$ denotes the transitive and reflexive closure of $\leq$.

Def. 5.3. *Related*(E) =

$$\{ E_i \mid \exists E_j, \alpha : [E_j]_\alpha \leq^* [E_i]_\alpha \wedge [E_j]_\alpha \leq^* [E]_\alpha \}$$

For `v5` of the `reverse()` method in Figure 2, the set of related variables for `v5` contains `temp` because of the presence of [`temp`]$_{\alpha_9} \leq Elem_{\alpha_9}$(`v5`) and similar constraints. No other variables are related to `v5`.

It is legal to rewrite the relevant declarations to use the new type parameter only if no such declaration needs a type more precise than the inferred type. This could only happen in the presence of downcasts and `instanceof` tests that could fail. Definition 5.4 below

[9]This constraint would arise for calls to `reverse` from outside the program.

28	(B15)	[`v5`]$_{\alpha_9} \leq$`Collection`$< X_{25} >$
29	(B15)	[`v5`]$_{\alpha_9} \leq$`Collection`$< X_{26} >$
29	(B15)	[`v5`]$_{\alpha_9} \leq$`List`$< X_{27} >$
29	(B18)	[`v5.get(v5.size()-t)`]$_{\alpha_9} \triangleq Elem_{\alpha_9}$(`v5`)
29	(B1)	[`v5.get(v5.size()-t)`]$_{\alpha_9} \leq$[`temp`]$_{\alpha_9}$
30	(B15)	[`v5`]$_{\alpha_9} \leq$`List`$< X_{28} >$
30	(B18)	[`v5.get(t)`]$_{\alpha_9} \triangleq Elem_{\alpha_9}$(`v5`)
30	(B15)	[`v5`]$_{\alpha_9} \leq$`List`$< X_{29} >$
30	(B17)	[`v5.get(t)`]$_{\alpha_9} \in Types_{\alpha_9}$(`v5`)
31	(B15)	[`v5`]$_{\alpha_9} \leq$`List`$< X_{30} >$
31	(B15)	[`temp`]$_{\alpha_9} \in Types_{\alpha_9}$(`v5`)

Figure 5: Type constraints generated for context α_9 associated with method `A.reverse()`

```
public void reverse(Vector v5) {
  for (int t=0; t < v5.size()/2; t++) {
    Integer temp = (Integer)v5.get(v5.size()-t);
    v5.add(v5.size()-1, v5.get(t));
    v5.add(t, temp);
  }
}
```

Figure 6: `reverse` method with bad cast

states a condition sufficient to ensure this. For example, Figure 6 shows the `reverse` method altered to cast `temp` to an `Integer`. In this case, parameter `v5` of method `reverse()` will fail the test because in context α_7, we have that [`temp`]$_{\alpha_7}$ = `Integer` but $Elem_{\alpha_7}(v5)$ is `String`, which is not a subtype of `Integer`. If a parameter passes this test, we replace all declarations related to it with its type parameter. In the case of method `reverse()`, a type parameter can be associated with parameter `v5`, because the type inferred for its related variables (i.e., `temp`) is the same as the type inferred for `v5`, in each context.

Def. 5.4. *ParamOK*(E) =

$$\nexists E', \alpha \left(\begin{array}{l} E' \in \textit{Related}(E) \\ [E']_\alpha = T \\ [E]_\alpha \not\leq T \end{array} \right)$$

Bounds may need to be imposed on type parameters that are introduced. The *ParamOK* test ensures that all declared types of related variables are supertypes of the types we inferred for the parameter itself . Thus, when we introduce a type parameter, we can always choose bound that is a supertype of the types inferred for the parameter and is a subtype of all declarations. There may be more than one such type, of which any will do. This is specified in Definition 5.5, in which E_i is the ith parameter of method m.

Def. 5.5. *TypeBound*(E_i, m) =

$$\left\{ T \;\middle|\; \textit{Elem}(()E_i) \leq T \wedge \forall E, \alpha, P \left(\begin{array}{l} E_i \in \textit{TPC}(m) \wedge \\ E \in \textit{Related}(E_i) \\ T \leq [E]_\alpha \end{array} \right) \right\}$$

There is no need for a bound in our `reverse` example, since it could just be `Object`.

6. IMPLEMENTATION AND RESULTS

6.1 Implementation Details

We implemented our algorithms as an Eclipse refactoring [2], building on Eclipse's Java Development Toolkit (JDT), which provide Abstract Syntax Trees (AST's), searching (e.g., for call sites), and source rewriting. We extended its type constraint infrastructure, previously developed for generalization-related refactorings [17], in order to accommodate generic types and context-sensitivity. As the JDT does not yet support Java 1.5 generics, we verified the transformed code using a beta-release of Sun's JDK 1.5 compiler.

The context-inference algorithm of Section 3 is implemented as a classic propagation-based call-graph construction engine [9]. Starting from a root methods, data flow and call-graph construction intertwine: processing newly reached methods reveals new call sites and allocation sites, and as allocation sites reach call sites, new contexts are added to the call graph. The analysis engine is is currently a simple worklist-based solver and many well-known optimizations (e.g., topological ordering, efficient bit sets) remain to be incorporated.

The AST's are traversed to generate type constraints as presented in Section 4, with one exception. Similar to [6], we replace an **or**-constraint of the form $[E]_\alpha \leq T_1$ **or** $\cdots$ **or** $[E]_\alpha \leq T_N$ with one

of its "branches" $[E]_\alpha \leq T_j$ in order to simplify the solving process. While this restricts the set of types that can be given to E, the original type of E must be solution.

A graph is then constructed whose nodes are variables, fields, returns and expressions (as well as *Elem* variables), and whose edges encode the type constraints. Each node has a type estimate, either a finite set of types, or a type variable. Initial type estimates are: (i) for an *Elem* node, a distinct type variable[10], (ii) for nodes corresponding to entities in binary classes, the entity's type in the original program, and (iii) for any other node, the set of all types. The inference engine uses a work-list based approach that involves: removal of elements from concrete sets of types, unification of type parameters with concrete sets of types, and recursive unification of element type variables when processing nodes that have container sub-structure. Type estimate sets monotonically decrease in size, guaranteeing the algorithm's termination.

When constraint solution terminates, nodes may still have type estimate sets S of size > 1. The solver processes these nodes iteratively, by selecting a single specific type $s \in S$ for each such node n, and entering n on the work-list. Here, the least specific type in S is chosen for container-typed nodes, while the most specific type in S is chosen for other nodes, so as to maximize the possibility that casts may be removed.

6.2 Pragmatic Issues

Many language constructs and API limitations in Eclipse required pragmatic solutions. Space limitations prevent us from mentioning all but the most significant of these.

Support for anonymous and local classes in the Eclipse JDT is rudimentary, so we refactored them out of our benchmarks.

Our implementation preserves original types of declarations which lack source code. We do not infer parametric types for such declarations of `Collection`'s. This requires two steps: (i) an additional constraint is generated that forces the element type of such `Collection`-related declarations to be `Object`, and (ii) the type of any `Collection`-related declaration or allocation site for which element type `Object` is inferred is left "raw".

Our refactoring must be applicable to arbitrary groups of classes, not just to single programs. Therefore, we make conservative approximations about data flow within external classes, call-backs from library code, and reflection, similar in spirit to, e.g., [18, 14]. We use a single logical expression E_{ext} for all external code. Calls to external methods cause assignments between their arguments (and return value) and E_{ext}. We *ignore* calls to a few heavily used methods in the class libraries that are known to be benign. An example is the constructor of `java.lang.Object`.

The treatment of `Map`-style container classes such as `java.util.Hashtable` and `java.util.HashMap` is analogous to that of `Collection`'s, but two implicitly created `Collection`'s (one representing the `Map`'s set of keys, the other, its set of values) need to be modeled.

Arrays pose several interesting challenges. To reduce pollution, each array creation gets a distinct allocation site, similar to that of `Collection`'s and `Map`'s. Arrays are handled using an `ArrayModel` class similar to that for `Collection`'s, and reads/writes to/from arrays are modeled as calls to get/set methods in `ArrayModel`. Another array-related issue stems from limitations in Java 1.5's erasure semantics, which disallows arrays of generic types. The type inference engine ensures that element type `Object` is inferred for any container-related value that flows into an array. This technique effectively forces the use of the raw type for such declarations.

[10] except for *Elem* nodes of container-related array-index expressions, as noted in Section 6.2.

6.3 Experimental Results

We evaluated two variations of our technique on a number of small Java benchmarks, as indicated in Table 2. The *context-insensitive* (CI) variation uses one context per method, and the *context-sensitive* (CS) variation uses the variation on Agesen's algorithm described in Section 3.

We used the following benchmarks. *Hanoi*[11] is a simple animated AWT applet that solves the Towers of Hanoi problem and makes limited use of containers. *JUnit*[12] is a widely used framework for unit and regression testing that includes both Swing and AWT UI's. *JLex*[13] is a lexical-analyzer generator that makes significant use of vectors and maps. *JavaCup*[14] is an LALR(1) parser generator that uses tables heavily to represent shift and reduce actions. *Mango*[15] is a set of utility algorithms for searching, sorting and transforming collections. Its methods are relatively generic, and so client usage determines whether context sensitivity is required to genericize it.

For all benchmarks except *Mango1*, *Mango2*, and *Mango3* the application's `main()` routine(s) were designated as the sole entry point method. To evaluate our approach on the *Mango* library, we wrote three tests similar to *Mango*'s unit tests that exercise major portions of its functionality. *Mango1* tests the algorithms `find`, `count` and `remove`, that use object equality to examine collections. *Mango2* tests `findIf` and `removeIf`, which instead use a predicate function. *Mango3* exercises the `Transform` algorithms, which create new collections based upon existing ones.

The results of Table 2 can be summarized as follows. First, no benchmark except *Mango* required context-sensitivity because of the simple way in which container objects were used. In both *Mango1* and *Mango2*, the context-sensitive approach discussed in the paper was sufficient, while for *Mango3*, more context-sensitivity was required.

A reasonable measure of the effectiveness of our technique is the percentage of down-casts in the program that can be removed when generic types are inferred. As can be seen from Table 2, with the exception of *Mango3*, our method removes between 40% and 100% of the casts. A manual inspection of the refactored programs revealed that most of the remaining casts are not related to the use of container classes, or are constrained by the use of fixed/binary API's, such as AWT.

7. RELATED WORK

The problem of introducing generic types into a program to broaden its use has been approached before by several researchers.

Siff and Reps [16] focused on translating C into C++ by detecting latent polymorphism and introducing function templates. Our aim, in addition to genericising methods, is to specialize the use of generic containers in user code. Also, we introduce type parameters only when needed, while Siff and Reps add restraints to prevent over-generalizing.

Duggan [8] gives an algoritm (not implemented) for genericising classes in a small subset of Java into a particular polymorphic variant of that subset.

The programming environments CodeGuide [5] and IntelliJ IDEA [10] provide "Generify" refactorings with goals that are comparable to our tool's. No details of implementation or analysis are

[11] See `www.alphaworks.ibm.com/tech/jax`.

[12] See `www.junit.org`.

[13] See `www.cs.princeton.edu/~appel/modern/java/JLex/`.

[14] See `www.cs.princeton.edu/~appel/modern/java/CUP/`

[15] See `www.jezuk.co.uk/cgi-bin/view/mango`.

benchmark	#classes/methods/ fields/LOC	#container allocations	#container declarations	#casts	#casts removed (CI)	%casts removed (CI)	#casts removed (CS)	%casts removed (CS)
Hanoi	41/343/206/4028	3	6	20	14	70	14	70
JUnit	105/528/123/5317	24	63	54	21	39	21	39
JLex	26/151/235/7841	17	45	71	53	75	53	75
JavaCup	36/371/215/10598	19	78	502	373	74	373	74
Mango1	92/357/69/2808	2	9	2	0	0	2	100
Mango2	92/357/69/2808	3	13	4	0	0	2	50
Mango3	92/357/69/2808	1	17	10	0	0	0	0

Table 2: Benchmark program characteristics and results.

provided in either case so we cannot directly compare the results.

Donovan, Kiezun and Ernst [7] present a technique for converting non-generic Java code to use generic libraries. A context-sensitive concrete class analysis that discovers elements of containers (or any generic class) at their allocation site is followed by context-insensitive type constraint system creation and resolution. The result is a typing for references to generic types. The transformation preserves behavior by retaining the program's erasure, which may be, in our view, too conservative a constraint. Their work differs from ours in several other important ways: it targets a non-current version of the specification and will need adaptation to the final one. It does not introduce method type parameters, so certain declared types must remain unchanged (e.g., the parameter of method `reverse()` in Figure 2). Their approach is aimed at any generic library, while ours is targeted for what in our experience are the most widely used generic types—containers, which makes our model simpler and still extensible. Their analysis requires a modified compiler to create bytecode that retains source-level information required for source modifications. Our analysis is fully source-code based and thus more readily available.

Tip, Kiezun, and Baümer [17] give an algorithm in which type constraints are used for refactoring, to determine whether a set of declarations can be updated to refer to a superinterface of a given class. The goal of that work is that of generalization while the analyses presented here both *specialize* (references to container types) and *generalize* (by introducing generic methods) to produce a better typing for a program.

8. CONCLUSIONS AND FUTURE WORK

We presented context-sensitive and insensitive techniques for genericizing uses of the Java Collections API. Our evaluation suggests considerable scope for even a context-insensitive approach, at least for the relatively small programs we have evaluated so far. The context-sensitive approach was needed for Mango, which provides a layer of generic functionality on top of the Collections API, and hence benefits from the insertion of type parameters into its code. For all benchmarks except Mango3, our techniques remove between 40% and 100% of all down-casts.

Future work includes the evaluation of our techniques on more and larger benchmarks, and the release of our refactoring as part of Eclipse. In Mango, we encountered complex uses of containers for which a more precise context-sensitive analysis is needed to remove down-casts. We plan to explore scalable adaptive schemes that attempt to introduce additional context-sensitivity only where it is useful.

9. REFERENCES

[1] Agesen, O. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Stanford University, December 1995.

[2] Bäumer, D., Gamma, E., and Kiezun, A. Integrating refactoring support into a Java development tool. In *OOPSLA'01 Companion* (October 2001).

[3] Bracha, G., Cohen, N., Kemper, C., Odersky, M., Stoutamire, D., Thorup, K., and Wadler, P. Adding generics to the Java programming language: Public draft specification, version 2.0. Tech. rep., Java Community Process JSR-000014, June 23 2003.

[4] 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).

[5] Omnicore codeguide. `http://www.omnicore.com/codeguide.htm`.

[6] De Sutter, B., Tip, F., and Dolby, J. Customization of Java library classes using type constraints and profile information. In *Proc. of ECOOP* (2004).

[7] Donovan, A., Kiezun, A., and Ernst, M. Converting Java programs to use generic libraries. Submitted for publication.

[8] Duggan, D. Modular type-based reverse engineering of parameterized types in Java code. In *Proc. of OOPSLA* (1999).

[9] Grove, D., and Chambers, C. A framework for call graph construction algorithms. *ACM TOPLAS 23*, 6 (2001).

[10] JetBrains IntelliJ IDEA. `http://www.intellij.com/idea/`.

[11] Igarashi, A., Pierce, B. C., and Wadler, P. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS 23*, 3 (2001).

[12] Odersky, M., and Wadler, P. Pizza into Java: Translating theory into practice. In *Proc. of POPL* (1997).

[13] Palsberg, J., and Schwartzbach, M. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.

[14] Rountev, A., Ryder, B. G., and Landi, W. Data-flow analysis of program fragments. In *Proc. of FSE* (1999).

[15] Ryder, B. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proc. of CC* (2003).

[16] Siff, M., and Reps, T. W. Program generalization for software reuse: From C to C++. In *Foundations of Software Engineering* (1996), pp. 135–146.

[17] Tip, F., Kiezun, A., and Bäumer, D. Refactoring for generalization using type constraints. In *Proc. of OOPSLA* (2003).

[18] Tip, F., Sweeney, P. F., Laffra, C., Eisma, A., and Streeter, D. Practical extraction techniques for Java. *ACM TOPLAS 24*, 6 (2002).