Refactoring for Parameterizing Java Classes

Adam Kieżun Michael D. Ernst
MIT CS&AI Lab
{akiezun,mernst}@csail.mit.edu

Frank Tip Robert M. Fuhrer
IBM T.J. Watson Research Center
{ftip,rfuhrer}@us.ibm.com

Abstract

Type safety and expressiveness of many existing Java libraries and their client applications would improve, if the libraries were upgraded to define generic classes. Efficient and accurate tools exist to assist client applications to use generic libraries, but so far the libraries themselves must be parameterized manually, which is a tedious, time-consuming, and error-prone task. We present a type-constraint-based algorithm for converting non-generic libraries to add type parameters. The algorithm handles the full Java language and preserves backward compatibility, thus making it safe for existing clients. Among other features, it is capable of inferring wildcard types and introducing type parameters for mutually-dependent classes. We have implemented the algorithm as a fully automatic refactoring in Eclipse.

We evaluated our work in two ways. First, our tool parameterized code that was lacking type parameters. We contacted the developers of several of these applications, and in all cases they confirmed that the resulting parameterizations were correct and useful. Second, to better quantify its effectiveness, our tool parameterized classes from already-generic libraries, and we compared the results to those that were created by the libraries' authors. Our tool performed the refactoring accurately—in 87% of cases the results were as good as those created manually by a human expert, in 9% of cases the tool results were better, and in 4% of cases the tool results were worse.

1 Introduction

Generics (a form of parametric polymorphism) are a feature of the Java 1.5 programming language. Generics enable the creation of type-safe reusable classes, which significantly reduces the need for potentially unsafe down-casts in source code. Much pre-1.5 Java code would benefit from being upgraded to use generics. Even new code can benefit, because a common programming methodology is to write non-generic code first and convert it later. The task of introducing generics to existing code can be viewed as two related technical problems [7]:

1. The *parameterization problem* consists of adding type parameters to an existing class definition so that it can be used in different contexts without the loss of type information. For example, one might convert the class definition `class ArrayList {...}` into `class ArrayList<T> {...}`, with certain uses of `Object` in the body replaced by `T`.
2. Once a class has been parameterized, the *instantiation problem* is the task of determining the type arguments that should be given to instances of the generic class in client code. For example, this might convert a declaration `ArrayList names;` into `ArrayList<String> names;`.

The former problem subsumes the latter because the introduction of type parameters often requires the instantiation of generic classes. For example, if class `HashSet` uses a `HashMap` as an internal representation of the set, then parameterizing the `HashSet` class requires instantiating the references to `HashMap` in the body of `HashSet`.

If no parameterization is necessary, the instantiation problem can be solved using completely automatic and scalable techniques [7, 10], and the INFER GENERIC TYPE ARGUMENTS refactoring in Eclipse 3.1 is based on our previous work [10]. However, to our knowledge, no previous practical and satisfactory solution to the parameterization problem exists. Thus far, class libraries such as the Java Collections Framework have been parameterized manually, and developers involved with this task described it as very time-consuming, tedious, and error-prone [11, 2].

We present a solution to the parameterization problem such that: (i) the behavior of any client of the parameterized classes is preserved, (ii) the translation produces a result similar to that which would be produced manually by a skilled programmer, and (iii) the approach is practical in that it admits an efficient implementation that is easy to use. Our approach fully supports Java 1.5 generics, including bounded and unbounded wildcards, and it has been implemented as a refactoring in Eclipse. Previous approaches for solving the parameterization problem [8, 6, 20] did not include a practical implementation, and produced incorrect or suboptimal results, as will be discussed in Section 5.

We evaluated our work in two ways. First, we parameterized non-generic classes, and examined the results to ensure that they were satisfactory and usable to clients. Second, we complemented that qualitative analysis with a quantitative one in which we compared its results to those produced by human programmers. Our tool computes a solution that is nearly identical to the hand-crafted one, and is sometimes

```
1 // A MultiSet may contain a given element more than once.
2 // Each element is associated with a count (a cardinality).
3 public class MultiSet {
4   // counts maps each element to its number of occurrences.
5   private Map counts = new HashMap();
6   public void add(Object t1) {
7     counts.put(t1, new Integer(getCount(t1) + 1));
8   }
9   public Object getMostCommon() {
10    return new SortSet(this).getMostCommon();
11  }
12  public void addAll(Collection c1) {
13    for (Iterator iter = c1.iterator();
14         iter.hasNext(); ) {
15      add(iter.next());
16    }
17  }
18  public boolean contains(Object o1) {
19    return counts.containsKey(o1);
20  }
21  public boolean containsAll(Collection c2) {
22    return getAllElements().containsAll(c2);
23  }
24  public int getCount(Object o2) {
25    return (! contains(o2)) ? 0 :
26           ((Integer)counts.get(o2)).intValue();
27  }
28  public Set getAllElements() {
29    return counts.keySet();
30  }
31 }
32
33 // A SortSet sorts the elements of a MultiSet by their cardinality.
34 class SortSet extends TreeSet {
35   public SortSet(final MultiSet m) {
36     super(new Comparator() {
37       public int compare(Object o3, Object o4) {
38         return m.getCount(o3) - m.getCount(o4);
39     }});
40     addAll(m.getAllElements());
41   }
42   public boolean addAll(Collection c3) {
43     return super.addAll(c3);
44   }
45   public Object getMostCommon() {
46     return isEmpty() ? null : first();
47   }
48 }
```

```
1 // A MultiSet may contain a given element more than once.
2 // Each element is associated with a count (a cardinality).
3 public class MultiSet<T1> {
4   // counts maps each element to its number of occurrences.
5   private Map<T1,Integer> counts = new HashMap<T1,Integer>();
6   public void add(T1 t1) {
7     counts.put(t1, new Integer(getCount(t1) + 1));
8   }
9   public T1 getMostCommon() {
10    return new SortSet<T1>(this).getMostCommon();
11  }
12  public void addAll(Collection<? extends T1> c1) {
13    for (Iterator<? extends T1> iter = c1.iterator();
14         iter.hasNext(); ) {
15      add(iter.next());
16    }
17  }
18  public boolean contains(Object o1) {
19    return counts.containsKey(o1);
20  }
21  public boolean containsAll(Collection<?> c2) {
22    return getAllElements().containsAll(c2);
23  }
24  public int getCount(Object o2) {
25    return (! contains(o2)) ? 0 :
26           (~~(Integer)~~counts.get(o2)).intValue();
27  }
28  public Set<T1> getAllElements() {
29    return counts.keySet();
30  }
31 }
32
33 // A SortSet sorts the elements of a MultiSet by their cardinality.
34 class SortSet<T2> extends TreeSet<T2> {
35   public SortSet(final MultiSet<? extends T2> m) {
36     super(new Comparator<T2>() {
37       public int compare(T2 o3, T2 o4) {
38         return m.getCount(o3) - m.getCount(o4);
39     }});
40     addAll(m.getAllElements());
41   }
42   public boolean addAll(Collection<? extends T2> c3) {
43     return super.addAll(c3);
44   }
45   public T2 getMostCommon() {
46     return isEmpty() ? null : first();
47   }
48 }
```

Figure 1. Classes `MultiSet` and `SortSet` before and after parameterization by our tool. In the right column, modified declarations are underlined and a removed cast is struck through. The example uses collection classes from package `java.util` in the standard Java 1.5 libraries: `Map`, `HashMap`, `Set`, `Collection`, `TreeSet`.

even better (i.e., it permits more casts to be removed).

The remainder of this paper is organized as follows. Section 2 gives a motivating example to illustrate the problem and our solution. Section 3 presents our class parameterization algorithm. Section 4 describes the experiments we performed to evaluate our work. Section 5 overviews related work, and Section 6 concludes.

2 Example

Figure 1 shows an example program consisting of two classes, `MultiSet` and `SortSet`, before and after automatic parameterization by our tool. The following observations can be made about the refactored source code:

1. On line 6, the type of the parameter of `MultiSet.add()` has been changed to T_1, a new type parameter of class `MultiSet` that represents the type of its elements.
2. On line 9, the return type of `MultiSet.getMostCommon()` is now T_1. This, in turn, required parameterizing class `SortSet` with a type parameter T_2 (line 34) and changing the return type of `SortSet.getMostCommon()` (line 45) to T_2. This shows that parameterizing one class may require parameterizing others.
3. On line 12, the parameter of `MultiSet.addAll()` now has type `Collection<? extends T1>`, a bounded *wildcard* type that allows any `Collection` that is parameterized with a subtype of the receiver's type argument T_1 to be passed as an argument. The use of a wildcard is very important here. Suppose that the type `Collection<T1>` were used instead. Then a (safe) call to `addAll()` on a receiver of type `MultiSet<Number>` with an actual parameter of type `List<Integer>` would not compile; the client would be forbidden from using those (desirable) types.
4. On line 18, the type of the parameter of `MultiSet.contains()` remains `Object`. This is desirable and corresponds to the (manual) parameterization of the JDK libraries. Suppose the parameter of `contains()` had type T_1 instead, and consider a client that adds only `Integer`s to a `MultiSet`

and that passes an `Object` to `contains()` at least once on that `MultiSet`. Such a client would have to declare the `MultiSet` suboptimally as `MultiSet<Object>`, rather than `MultiSet<Integer>` as permitted by our solution.

5. On line 21, the type of the parameter of `MultiSet.containsAll()` has become an *unbounded wildcard* `Collection<?>` (which is shorthand for `Collection<? extends Object>`). Analogously with the `contains()` example above, use of `Collection<`T_1`>` would force a less precise parameterization of some instances of `MultiSet` in client code.
6. On line 28, the return type of `MultiSet.getAllElements()` is parameterized as `Set<`T_1`>`. It is important *not* to parameterize it with a wildcard, as that would severely constrain client uses of the method's return value (e.g., it would be illegal to add elements other than `null` to the returned set.)
7. On line 42, the type of the parameter of method `SortSet.addAll()` is parameterized as `Collection<? extends` T_2`>`. Any other parameterization would be incorrect because the method overrides the method `TreeSet.addAll()`, and the signatures of these methods must remain identical to preserve the overriding relationship [11].

Even for this simple example, the desired parameterization requires 19 non-trivial changes to the program's type annotations, and involves subtle reasoning. In short, class parameterization is a complex process, and automated tool assistance is highly desirable.

Finally, we remark that, although the example uses the standard (generic) Java libraries (e.g., `Map<K,V>`, `Set<E>`, etc.), our technique is also applicable to classes that do not depend on generic classes.

3 Algorithm

Our parameterization algorithm has 3 steps:

1. Create type constraints for all program constructs, and add additional constraints using a set of closure rules.
2. Solve the constraints to determine a type for each declaration.
3. Rewrite the program's source code: add new formal type parameters, rewrite program declarations, and remove redundant casts.

After Section 3.1 presents the notation used for representing type constraints, Sections 3.2–3.3 present the steps of the algorithm.

3.1 Type Constraints

This paper generalizes and extends a framework of type constraints [15] that has been used for refactoring [19, 5, 1] and, in particular, as the basis for a refactoring that solves the instantiation problem [10] (i.e., inferring the type arguments that should be given to generic classes in client code). Due to space limitations, the pre-existing parts of the type constraints formalism are described informally, and the presentation focuses on the new constraints notation and algorithmic contributions that are needed for solving the parameterization problem.

Type constraints are a formalism for expressing subtype relationships between program entities that are required for preserving the type-correctness of program constructs. Consider an assignment `x=y`. The constraint $[\texttt{y}] \leq [\texttt{x}]$ states that the type of `y` (represented by the *constraint variable* $[\texttt{y}]$) must be a subtype of the type of `x`. If the original program is type-correct, this constraint holds. The refactoring must preserve the subtype relationship $[\texttt{y}] \leq [\texttt{x}]$ so that the refactored program is type-correct. As another example, consider a method `Sub.foo(Object p)` that overrides a method `Super.foo(Object q)`. To preserve dynamic dispatch behavior, the refactored program must preserve the overriding relationship. This is guaranteed if the refactored program satisfies a constraint $[\texttt{p}] = [\texttt{q}]$ stating that the types of `p` and `q` are identical.

Our algorithm generates type constraints from a program's abstract syntax tree (AST) in a syntax-directed manner. A solution to the resulting constraint system corresponds to a refactored version of the program for which type-correctness and program behavior is preserved. Frequently, many legal solutions exist, all of which preserve the program's semantics, but some of which are more useful to clients. Our algorithm uses heuristics (Section 3.3.1) to choose among legal solutions, but it never violates the semantics of the program by changing behavior.

Refactoring for parameterization is significantly more complex than previous work: it involves the introduction of formal type parameters with inheritance relations between them, while simultaneously rewriting existing declarations to refer to these new type parameters. This required nontrivial extensions and modifications to the type constraints formalism and the solver, including most notably:

1. the introduction of *context constraint variables* representing the type with which newly introduced type parameters are instantiated,
2. the introduction of *wildcard constraint variables* to accommodate wildcard types, and
3. the use of heuristics to guide the solver towards solutions preferred by human programmers (e.g., not introducing too many type parameters, and preferring solutions with wildcard types in certain cases), without violating program semantics.

Figure 2 presents the type constraint notation. Type constraints are of the form $\alpha \leq \alpha'$ or $\alpha = \alpha'$, where α and α' are constraint variables. Most forms of constraint variables are standard, but we discuss the new forms *context variables* and *wildcard variables*.

Context Variables. This paper introduces a new form of constraint variable that represents the type with which a (newly introduced) formal type parameter is instantiated. Such a *context variable* is of the form $\mathcal{I}_{\alpha'}(\alpha)$ and represents the *interpretation* of constraint variable α in a *context* given by constraint variable α'.

We give the intuition behind context variables by example.

Type constraint variables:

$\alpha ::=$	α_{nw}	non-wildcard variable
	`? extends` α_{nw}	wildcard type upper-bounded by α_{nw}
	`? super` α_{nw}	wildcard type lower-bounded by α_{nw}
$\alpha_{nw} ::=$	α_{ctxt}	contexts
	T	type parameter
	$\mathcal{I}_{\alpha_{ctxt}}(\alpha)$	α, interpreted in context α_{ctxt}
$\alpha_{ctxt} ::=$	$[e]$	type of expression e
	$[M_{\text{ret}}]$	return type of method M
	$[M_i]$	type of i^{th} formal parameter of method M
	C	monomorphic type constant

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'

Figure 2. Notation used for defining type constraints.

$\tau ::=$	τ_{nw}	non-wildcard type
	`? extends` τ_{nw}	upper-bounded wildcard type
	`? super` τ_{nw}	lower-bounded wildcard type
$\tau_{nw} ::=$	C	monomorphic type constant
	T `extends` τ_{nw}	type parameter

Figure 3. Grammar of types used in the analysis.

- Consider the JDK class `List<E>`. References to its type parameter `E` only make sense within the definition of `List`. In the context of an instance of `List<String>`, the interpretation of `E` is `String`, while in the context of an instance of `List<Number>`, the interpretation of `E` is `Number`. We write $\mathcal{I}_{[\texttt{x}]}(\texttt{E})$ for the interpretation of `E` in the context of variable `x`.
- Consider the call `counts.put(t1,...)` on line 7 of Figure 1. Java requires the type of the first actual parameter `t1` to be a subtype of the formal parameter `key` of `Map.put`. This is expressed by the constraint $[\texttt{t1}] \leq \mathcal{I}_{[\texttt{counts}]}(\texttt{key})$, which means that the type of `t1` is a subtype of the type of `key`, *as interpreted by interpretation function* $\mathcal{I}_{[\texttt{counts}]}$. This interpretation function maps the formal type parameters in the declared type of `Map` to the types with which they are instantiated in the declaration of `counts`.

  Using a context variable here is important. Generating a constraint without a context, i.e., $[\texttt{t1}] \leq [\texttt{key}]$, would be incorrect. Variable `key` has type `K`, and there is no direct subtype relationship between the type $\texttt{T}_1$ of $[\texttt{t1}]$ and the type parameter `K` of the distinct class `Map`. It would be likewise incorrect to require that $[\texttt{t1}] \leq \texttt{K}$.

  Our algorithm eventually resolves $[\texttt{t1}]$ to $\texttt{T}_1$, implying that $\mathcal{I}_{[\texttt{counts}]}$ maps `K` to $\texttt{T}_1$, and thus $\mathcal{I}_{[\texttt{counts}]}(\texttt{key})$ resolves to $\texttt{T}_1$.
- In some cases, a context α_{ctxt} is irrelevant. For example, $\mathcal{I}_{\alpha_{ctxt}}(\texttt{String})$ always resolves to `String`, regardless of the context α_{ctxt} in which it is interpreted.

Wildcard Variables. There are two situations where our algorithm introduces wildcards in the refactored program.

Wildcard variables are of the form `? extends` α or `? super` α (where α is another constraint variable), and are used in cases where Java's typing rules *require* the use of wildcard types. For example, in Figure 1, `SortSet.addAll()` (line 42) overrides `java.util.TreeSet.addAll()`. If `SortSet` becomes a generic class with formal type parameter $\texttt{T}_2$, then preserving this overriding relationship requires the formal parameter `c3` of `SortSet.addAll()` to have the same type as that of `TreeSet.addAll()`, which is declared in the Java standard libraries as `TreeSet.addAll(Collection<? extends E>)`. Three parts of our algorithm work together to accomplish this: (i) The type of `c3` is represented, using a context variable, as `Collection<`$\mathcal{I}_{[\texttt{c3}]}(E)$`>`, (ii) Type constraint generation (Section 3.2) produces $\mathcal{I}_{[\texttt{c3}]}(E) =$ `? extends` $\mathcal{I}_{\texttt{SortSet}}(E)$, which uses a wildcard variable, and (iii) Constraint solving (Section 3.3) resolves $\mathcal{I}_{\texttt{SortSet}}(E)$ to $\texttt{T}_2$.

Our algorithm also introduces wildcard types in cases where that results in a more flexible solution, as discussed in Section 3.3.1. However, this does not involve the use of wildcard variables.

$$\frac{\text{assignment } e_1{=}e_2}{\{[e_2] \leq [e_1]\}} \quad \text{(r1)}$$

$$\frac{\text{statement } \texttt{return } e_0 \text{ in method } M}{\{[e_0] \leq [M_{\text{ret}}]\}} \quad \text{(r2)}$$

$$\frac{\begin{array}{c}\text{call } e \equiv e_0\texttt{.}m\texttt{(}e_1\texttt{, ..., }e_k\texttt{)} \text{ to instance method } M \\ \mathit{canAddParams} \equiv \mathit{Decl}(M) \in \mathit{TargetClasses}\end{array}}{\begin{array}{ll}\mathit{CGen}([e], =, [M_{\text{ret}}], [e_0], \mathit{canAddParams}) \cup & \text{(r3)} \\ \bigcup_{1 \leq i \leq k} \mathit{CGen}([e_i], \leq, [M_i], [e_0], \mathit{canAddParams}) & \text{(r4)}\end{array}}$$

$$\frac{\alpha_1 \leq \alpha_2 \qquad \mathcal{I}_{\alpha'}(\alpha_1) \text{ or } \mathcal{I}_{\alpha'}(\alpha_2) \text{ exists}}{\mathcal{I}_{\alpha'}(\alpha_1) \leq \mathcal{I}_{\alpha'}(\alpha_2)} \quad \text{(r5)}$$

$$\frac{\alpha_1 \leq \alpha_2 \qquad \mathcal{I}_{\alpha_1}(\alpha) \text{ or } \mathcal{I}_{\alpha_2}(\alpha) \text{ exists}}{\mathcal{I}_{\alpha_1}(\alpha) = \mathcal{I}_{\alpha_2}(\alpha)} \quad \text{(r6)}$$

Figure 4. Representative examples of rules for generating type constraints from Java constructs (rules (r1)–(r4)) and of closure rules (rules (r5)–(r6)). Figure 5 shows auxiliary definitions used by the rules. *TargetClasses* is a set of classes that should be parameterized by adding type parameters. $\mathit{Decl}(M)$ denotes the class that declares method M.

3.2 Type Constraint Generation

Figure 4 shows a few representative rules for generating type constraints. The rules omitted from Figure 4 involve no significantly different analysis.

Rules (r1) and (r2) are from previous work. Rule (r1) states that the type of the right-hand side of an assignment must be equal to or a subtype of the left-hand side. Rule (r2) states that if a method contains a statement "`return` e_0", then the type of the returned expression e_0 must be equal to or a subtype of the method's declared return type. The complete set of rules [19, 10] is omitted for brevity and covers the entire Java language.

$\alpha_{\mathcal{P}}$ is the type, in the original program $\mathcal{P}$, of the program construct corresponding to α.

$CGen$ creates and returns a set of type constraints. The result constrains α and the interpretation of α' in context α''. The particular constraint ($=$, $\leq$, or $\geq$) between α and $\mathcal{I}_{\alpha''}(\alpha')$ is determined by op. $CGen$ is defined by case analysis on the type of its third parameter in the original program $\mathcal{P}$.

$$
\begin{array}{l}
CGen(\alpha, op, \alpha', \alpha'', canAddParams) = \\
\left\{ \begin{array}{ll}
\{\alpha\ op\ C\} & \text{when } \alpha'_{\mathcal{P}} \equiv C \text{ and } canAddParams \equiv false \quad \text{(c1)} \\
\{\alpha\ op\ \mathcal{I}_{\alpha''}(\alpha')\} & \text{when } \alpha'_{\mathcal{P}} \equiv C \text{ and } canAddParams \equiv true \quad \text{(c2)} \\
\{\alpha\ op\ C\} \cup \bigcup_{1 \leq i \leq m} CGen(\mathcal{I}_\alpha(W_i), =, [\tau_i], \alpha'', canAddParams) & \text{when } \alpha'_{\mathcal{P}} \equiv C\langle \tau_1, \ldots, \tau_m \rangle \text{ and } C \text{ is declared as } C\langle W_1, \ldots, W_m \rangle \quad \text{(c3)} \\
\{\alpha\ op\ \mathcal{I}_{\alpha''}(T)\} & \text{when } \alpha'_{\mathcal{P}} \equiv T \quad \text{(c4)} \\
CGen(\alpha, \leq, [\tau'], \alpha'', canAddParams) & \text{when } \alpha'_{\mathcal{P}} \equiv \texttt{? extends } \tau' \quad \text{(c5)} \\
CGen(\alpha, \geq, [\tau'], \alpha'', canAddParams) & \text{when } \alpha'_{\mathcal{P}} \equiv \texttt{? super } \tau' \quad \text{(c6)}
\end{array} \right.
\end{array}
$$

Figure 5. Auxiliary functions used by the constraint generation rules in Figure 4.

Rules (r3) and (r4) are among the new rules introduced in this research. Rules (r3) and (r4) govern method calls. Rule (r3) states that the type of the method call expression is the same as the return type of the method (in the context of the receiver). This corresponds to how the type checker treats a method call (i.e., the type of the call and the type of the method are the same). Rule (r4) relates the actual and formal type parameters of the call. The $TargetClasses$ set (a user input to the algorithm) indicates which classes should be refactored by adding type parameters (e.g., in Figure 1, classes `MultiSet`, `SortSet`, and the anonymous class declared on line 35 are assumed to be in $TargetClasses$). The auxiliary function *CGen*, defined in Figure 5, performs the actual generation of a set of constraints.

Java's type rules impose certain restrictions on parametric types. Closure rules such as (r5) and (r6) in Figure 4 enforce those restrictions. Rule (r5) requires that, given two formal type parameters[1] T_1 and T_2 such that $T_1 \leq T_2$ and any context α in which either actual type parameter $\mathcal{I}_\alpha(T_1)$ or $\mathcal{I}_\alpha(T_2)$ exists, the subtyping relationship $\mathcal{I}_\alpha(T_1) \leq \mathcal{I}_\alpha(T_2)$ must also hold. To illustrate this rule, consider a class `C<T1, T2 extends T1>` and any instantiation `C<C1, C2>`. Then, `C2` $\leq$ `C1` must hold, implying that e.g., `C<Number, Integer>` is legal but that `C<Integer, Number>` is not. Rule (r6) enforces invariant subtyping[2] of parametric types: $C\langle\tau\rangle$ is a subtype of $C\langle\tau'\rangle$ iff $\tau = \tau'$.

Examples. The following examples show how the rules of Figure 4 apply to three program constructs in the example of Figure 1. The examples assume that the set $TargetClasses$ is {`MultiSet`,`SortSet`}.

line 26: call `counts.get(o2)` to method `Map<K,V>.get(Object)`
$\xrightarrow{\text{(r3)}}$ $CGen([\texttt{counts.get(o2)}], =, [\texttt{Map.get}_{\text{ret}}], [\texttt{counts}], \textit{false})$
$\xrightarrow{\text{(c4)}}$ $\{\, [\texttt{counts.get(o2)}] = \mathcal{I}_{[\texttt{counts}]}(\texttt{V}) \,\}$
This constraint expresses that the type of the expression `[counts.get(o2)]` is the same as the return type of method `Map.get` in the context of the receiver `counts`.

line 7: call `counts.put(t1,...)` to method `Map<K,V>.put(K,V)`
$\xrightarrow{\text{(r4)}}$ $CGen([\texttt{t1}], \leq, [\texttt{Map.put}_1], [\texttt{counts}], \textit{false}) \cup \ldots$
$\xrightarrow{\text{(c4)}}$ $\{\, [\texttt{t1}] \leq \mathcal{I}_{[\texttt{counts}]}(\texttt{K}), \ldots \}$
In other words, the type of `t1` must be a subtype of the type of the first parameter of `Map.put` in the context of the type of `counts`.

line 15: call `add(iter.next())` to method `MultiSet.add(Object)`
$\xrightarrow{\text{(r4)}}$ $CGen([\texttt{iter.next()}], \leq, [t1], [\texttt{MultiSet.addAll.this}], \textit{true})$
$\xrightarrow{\text{(c2)}}$ $\{\, [\texttt{iter.next()}] \leq \mathcal{I}_{[\texttt{MultiSet.addAll.this}]}([\texttt{t1}]) \,\}$
This indicates that `iter.next()`'s type must be a subtype of the type of `t1` in the context of `MultiSet.addAll`.

3.3 Constraint Solving

A solution to the system of type constraints is computed using the iterative worklist algorithm of Figure 6. During solving, each variable α has an associated type estimate $Est(\alpha)$. An estimate is a set of types, where types are as defined in Figure 3. Each estimate is initialized to the set of all possible (non-parametric) types and shrinks monotonically as the algorithm progresses. When the algorithm terminates, each estimate consists of exactly one type. Because type estimates do not contain parametric types, they are finite sets, and algebraic operations such as intersection can be performed directly. As an optimization, our implementation uses a symbolic representation for type estimates.

Algorithm Details. First, the algorithm initializes the type estimate for each constraint variable, at lines 2 and 15–22 in Figure 6.

The algorithm uses a workset P containing those constraint variables that it has decided shall become type parameters, but for which that decision has yet to be executed. The set P is initially seeded with the constraint variable that corresponds to the declaration that is selected either by a heuristic or by the user (line 3). The inner loop of parameterize() (lines 5–11) repeatedly removes an element from P and sets its estimate to a singleton type parameter. For new type parameters, the upper bound is the declared type in the original (unparameterized) program.

Whenever a type estimate changes, those changes must be propagated through the type constraints, possibly reduc-

[1] or constraint variables that could become formal type parameters

[2] In the presence of wildcard types, Java uses the more relaxed 'containment' subtyping [11]: `? extends Number` is *contained* in `? extends Object` and therefore `Set`$\langle$`? extends Number`$\rangle$ is a subtype of `Set`$\langle$`? extends Object`$\rangle$). In this paper and in our implementation, we conservatively assume invariant subtyping even with wildcard types.

Notation:

$Est(\alpha)$	a set of types, the type estimate of constraint variable α
$\alpha_{\mathcal{P}}$	type of constraint variable α in the original program
$Sub(\tau)$	set of all non-wildcard subtypes of τ
$Wild(X)$	set of wildcard types (both lower- and upper-bounded) for all types in type estimate X
U_E^S	universe of all types, including all wildcard types (i.e., both `super` and `extends` wildcards)

```
 1 Subroutine parameterize():
 2 initialize()
   // P is a set of variables known to be type parameters
 3 P ⟵ {automatically- or user-selected variable}
 4 repeat until all variables have single-type estimates
 5   while P is not empty do
 6     α_tp ⟵ remove element from P
 7     if Est(α_tp) contains a type parameter then
 8       Est(α_tp) ⟵ {type parameter from Est(α_tp)}
 9     else
10       Est(α_tp) ⟵ {create new type parameter}
11     propagate()
12   if ∃α.|Est(α)| > 1 then
13     Est(α) ⟵ {select a type from Est(α)}
14     propagate()

   // Set initial type estimate for each constraint variable
15 Subroutine initialize():
16 foreach non-context variable α do
17   if α cannot have wildcard type then
18     Est(α) = Sub(α_P)
19   else
20     Est(α) = Sub(α_P) ∪ Wild(Sub(α_P))
21 foreach context variable I_α'(α) do
22   Est(I_α'(α)) = U_E^S

   // Reconcile the left and right sides of each type inequality
23 Subroutine propagate():
24 repeat until fixed point (i.e., until estimates stop changing)
25   foreach constraint α ≤ α' do
26     Remove from Est(α) all types that are not a subtype of
       a type in Est(α')
27     Remove from Est(α') all types that are not a supertype
       of a type in Est(α)
28     if Est(α) or Est(α') is empty then
29       stop: "No solution"
30 foreach context variable I_α'(α) do
31   if Est(I_α'(α)) is a singleton set with type parameter T
     and Est(α) does not contain T then
32     add α to P
```

Figure 6: Pseudo-code for the constraint solving algorithm.

ing the type estimates of other variables as well. The propagate() subroutine performs this operation, ensuring that the estimates on both sides of a type constraint contain only types that are consistent with the relation. Whenever a context variable $\mathcal{I}_{\alpha'}(\alpha)$ gets resolved to a type parameter, α must also get resolved to type parameter (line 30). To see why, suppose that α gets resolved to a non-type parameter type, C. In that case, the context is irrelevant (as mentioned in Section 3.1), and thus $\mathcal{I}_{\alpha'}(\alpha)$ also must get resolved to C (i.e., *not* a type parameter). This is a contradiction. This step propagates parameterization choices between classes.

Assembly of Parametric Types. The type estimates created during the constraint solution algorithm are all non-parametric, even for constraint variables that represent program entities whose type was parametric (e.g., `c1` on line 12 in Figure 1), or will be parametric after refactoring (e.g., `t1` on line 6 in Figure 1). A straightforward algorithm, applied after solving, assembles these results into parametric types. For instance, the type `Collection` for $[$`c1`$]$ and the type `? extends` T_1 for $\mathcal{I}_{[\texttt{c1}]}(E)$ are assembled into the type `Collection<? extends` T_1`>` for $[$`c1`$]$.

A technical report [13] walks through a detailed example of the solving algorithm. It also discusses how our system handles interfaces and abstract classes.

Some classes are not parameterizable by any tool [13]. If the presented algorithm is applied to such a class (e.g., `String`), then the algorithm either signals that parameterization is impossible (line 28 in Figure 6) or else produces a result in which the type parameter is used in only 1 or 2 places. An implementation could issue a warning in this case, but our prototype does not have this feature.

Our analysis, like many others, does not account for the effects of reflective method calls, but use of parameterized reflection-related classes (such as `java.lang.Class<T>`) poses no problems. The analysis can approximate the flow of objects in a native method based on the signature.

3.3.1 Heuristics

The algorithm of Figure 6 makes an underconstrained choice on lines 3, 6, 12, and 13. (On line 8, there is only one possibility.) Any choice yields a correct (behavior-preserving and type-safe) result, but some results are more useful to clients (e.g., permit elimination of more casts). Our implementation makes an arbitrary choice at lines 6 and 12, but uses heuristics at lines 3 and 13 to guide the algorithm to a useful result.

Our tool lets a user select, with a mouse click, a type to parameterize. Otherwise, it uses the following heuristic.

1. If a generic supertype exists, use the supertype's signatures in the subtype. This is especially useful for customized container classes.
2. Parameterize the return value of a "retrieval" method. A retrieval method's result is downcasted by clients, or it has a name matching such strings as `get` and `elementAt`. Even classes that are not collections often have such retrieval methods [7].
3. Parameterize the formal parameter to an insertion method. An insertion method has a name matching such strings as `add` or `put`.

Given a type estimate to narrow, line 13 chooses one of its elements. The heuristic minimizes the use of casts in

library	parameterizable classes			time	diff vs. manual		
	classes	LOC	type uses	(sec.)	same	better	worse
concurrent	14	2715	415	115	353	37	25
apache	74	9904	1183	301	1011	116	56
jutil	9	305	80	1	65	15	0
jpaul	17	827	178	6	148	22	8
amadeus	8	604	129	5	125	1	3
dsa	9	791	162	3	158	4	0
antlr	10	601	140	6	n/a	n/a	n/a
eclipse	7	582	100	5	n/a	n/a	n/a
Total	148	16329	2387	442	1860	195	92

Figure 7. Experimental results. "Classes" is the number of parameterizable classes in the library, including their nested classes. "LOC" is lines of code. "Type uses" is the number of occurrences of a reference (non-primitive) type in the library; this is the maximal number of locations where a type parameter could be used instead. The next column shows the cumulative run time. The "diff vs. manual" columns indicate how our tool's output compares to the manual parameterization.

client code, while preserving flexibility in cases where this does not affect type-safety: it prefers (in this order):

i) types that preserve type erasure over those that do not,
ii) wildcard types over non-wildcard types, and
iii) type parameters over other types, but only if such a choice enables inference of type parameters for return types of methods.

4 Evaluation

A practical type parameterization tool must be correct, accurate, and usable. Correctness requires that run-time behavior is not modified for any client. Accuracy requires that the parameterization is close to what a human would have written by hand. Usability requires that the tool is easier to use than other available approaches. This section describes our experimental evaluation of these desiderata.

4.1 Implementation

We implemented our technique in a refactoring tool that is integrated with the Eclipse integrated development environment. Our previous work on the instantiation problem [10] was adopted by the Eclipse developers and made into Eclipse 3.1's INFER GENERIC TYPE ARGUMENTS refactoring. Our parameterization tool builds on that previous implementation work and is integrated with Eclipse in a similar way.

A programmer can use our tool interactively to direct a refactoring process (each step of which is automatic) by selecting (clicking on) an occurrence of a type in the program. The tool automatically rewrites (parameterizes) the class in which the mouse click occurred, and possibly other classes as well. Alternatively, a programmer can specify a set of classes to parameterize, and the tool heuristically selects type occurrences. The tool uses Eclipse's built-in support for displaying changes, and the user can examine them one-by-one, accept them, or back out of the changes.

4.2 Methodology

Our evaluation uses a combination of 6 libraries that have already been parameterized by their authors, and 2 libraries that have not yet been made generic; these two varieties of evaluation have complementary strengths and weaknesses. Use of already-parameterized libraries lets us evaluate our technique's accuracy by comparing it to the judgment of a human expert other than ourselves. However, it is possible that the library authors performed other refactorings at the same time as parameterization, to ease that task. Use of non-parameterized libraries avoids this potential problem, but the evaluation is more subjective, and a human reading the tool output may not notice as many problems as one who is performing the full task. (It would be easy for us to parameterize them ourselves, but such an approach has obvious methodological problems.)

Our experiments started with a complete, non-parameterized library. (For already-parameterized libraries, we first applied a tool that erased the formal and actual type parameters and added necessary type casts.) Not all classes are amenable to parameterization; we selected a subset of the library classes that we considered likely to be parameterizable. Our tool failed to parameterize 34-40% of selected classes due to limitations of the current implementation. For example, the implementation does not yet support inference of F-bounded type parameters, e.g., `T extends Comparable<T>`. Also, our prototype implementation still contains a few bugs that prevent it from processing all classes (but do not affect correctness when it does run).

The experiments processed the classes of the library in the following order. We built a dependence graph of the classes, then applied our tool to each strongly connected component, starting with those classes that depended on no other (to-be-parameterized) classes. This is the same order a programmer faced with the problem would choose.

All experiments used our tool's fully automatic mode. For example, at each execution of line 3 of Figure 6, it chose the lexicographically first candidate type use, according to the heuristics of Section 3.3.1. To make the experiment objective and reproducible, we did not apply our own insight, nor did we rewrite source code to make it easier for our tool to handle, even when doing so would have improved the results.

Figure 7 lists the subject programs. All of these libraries were written by people other than the authors of this paper. **concurrent** is the `java.util.concurrent` package from Sun JDK 1.5. **apache** is the Apache collections library (`larvalabs.com/collections/`). **jutil** is a Java Utility Library (`cscott.net/Projects/JUtil/`). **jpaul** is the Java Program Analysis Utility Library (`jpaul.sourceforge.net`). **amadeus** is a data structure library (`people.csail.mit.edu/adonovan/`).

dsa is a collection of generic data structures (`www.cs.fiu.edu/~weiss/#dsaajava2`). **antlr** is a parser generator (`www.antlr.org`). **eclipse** is a universal tooling platform (`www.eclipse.org`). The last two libraries have not been parameterized by their authors.

Most of the classes are relatively small (the largest is 1303 lines), but this is true of Java classes in general. Our tool processes each class or related group of classes independently, so there is no obstacle to applying it to large programs.

4.3 Results

4.3.1 Correctness

A parameterization is correct if it is backward-compatible and self-consistent. Backward compatibility requires that the erasure of the resulting parameterized classes is identical to the input. If so, then the compiled `.class` file behaves the same as the original, unparameterized version: for example, all method overriding relationships hold, exactly the same set of clients can be compiled and linked against it, etc. Consistency (type-correctness) requires that the parameterized classes satisfy the typing rules of Java generics; more specifically, that a Java compiler issues no errors when compiling the parameterized classes.

Our tool's output for all the tested library classes is correct: it is both backward-compatible and consistent.

4.3.2 Accuracy

We determined our tool's accuracy in different ways for libraries for which no generic version is available (**antlr** and **eclipse**) and those for which a generic version is available (all others).

When no generic version of a library was available, its developers examined every change made by our tool and gave their opinion of the result. A developer of Eclipse concluded that the changes were "good and useful for code migration to Java 5.0." He mentioned only 1 instance (out of 100 uses of types in the Eclipse classes we parameterized) where the inferred result, while correct, could be improved. A developer of ANTLR stated that the changes made by our tool are "absolutely correct". He mentioned 1 instance (out of 140 uses of types in the parameterized classes) where the inferred result, while correct, could be improved.

When a generic version of a library is available, we examined each difference between the pre-existing parameterization and our tool's output. For 87% of all type annotations, the output of our tool is identical or equally good. For 4% of annotations, the output of our tool is worse than that created by the human. For 9% of annotations, the tool output is better than that created by the human. Figure 7 tabulates the results.

Given two parameterizations, we used two criteria to decide which was better. The first, and more important, is which one allows more casts to be removed—in clients or in the library itself. The secondary criterion is which one more closely follows the style used in the JDK collections; they were developed and refined over many years by a large group of experts and can be reasonably considered models of style. (When multiple styles appear in the JDK, we did not count differences in either the "better" or "worse" category.) The two criteria are in close agreement. We present three examples from each category.

Examples when the output of our tool was worse:

i. Our tool does not instantiate the field `next` in member type `LinkedBlockingQueue.Node` (in **concurrent**) as `Node<E>`, but leaves it raw. Such a choice is safe, but it is less desirable than the manual parameterization.
ii. Our tool does not infer type parameters for methods; for example, **apache**'s `PredicatedCollection.decorate`.
iii. Our tool inferred two separate type parameters for interface `Buffer` in the **apache** library. In this case the manual parameterization had only one.

Examples when the output of our tool was better (in each case, the developers of the package agreed the inferred solution was better than their manual parameterization):

i. Our tool adds a formal type parameter to member class `SynchronousQueue.Node` in **concurrent**. The parameter allows elimination of several casts inside `SynchronousQueue`.
ii. In method `VerboseWorkSet.containsAll` in **jpaul**, our tool inferred an upper-bounded type parameter wildcard for the `Collection` parameter. This permits more flexible use and fewer casts by clients, and also adheres to the standard collections style from the JDK.
iii. Our tool inferred `Object` as the type of the parameter of method `Canonical.getIndex` in **amadeus**. This is more flexible with fewer casts (and follows the JDK style). A similar case occurred in **jpaul** (our tool inferred `Object` for the parameter of `WorkSet.contains`).

4.3.3 Usability

Our tool operated fully automatically, processing each class in under 3 seconds on average. A user who elected to manually select type uses would only need to make 89 mouse clicks to add 135 type parameters to 148 classes. As mentioned, 4% of the computed results are sub-optimal, requiring manual correction.

By comparison, manual parameterization requires making 1655 edits to add generic types — after reading the code to decide what edits to make. The human result was sub-optimal 9% of the time, so adjusting the results after finishing is even more work than in the tool-assisted case.

We cannot compare our tool to any other tool because we know of none that supports this refactoring.

Those results illustrate that manual parameterization requires a significant amount of work. Parameterization of the **apache** collections took "a few weeks of programming", according to one of the developers. It is an error-prone activity and, to quote the same developer, "the main advantage we had was the over 11,000 test cases included with the project, that let us know we hadn't broken anything too badly."

5 Related Work

Duggan [8] presents an automatic approach for parameterizing classes written in PolyJava, a Java subset extended with parameterized types. Duggan's type inference infers one type parameter for each declaration in a class. Even after applying simplifications to reduce the number of useless type parameters, Duggan's analysis leads to classes with excess type parameters. Duggan's analysis is inapplicable to Java because PolyJava differs from Java 1.5 in several important ways, and Duggan does not address issues related to raw types, arrays of generic types, and wildcard types that arise in practice. Duggan does not report an implementation or empirical results.

Donovan and Ernst [6] present another automated approach for the parameterization of Java classes. The technique determines both type parameters and where declarations should refer to those type parameters. The approach first performs an intra-class analysis that constructs a type constraint graph using dataflow rules. Then, after collapsing strongly connected components and making additional graph simplifications, an inter-class analysis fuses type parameters where required to preserve method overriding. The algorithm also determines how references to generic classes should be updated, by inferring actual type parameters. Our work differs in several significant ways. Although Donovan and Ernst state that the desired solution is computed for several examples, they report that "often the class is over-generalized" (has too many type parameters). Their work pre-dates Java 1.5 generics and targets a translation to GJ [3]. As a result, they may infer arrays of generic types (disallowed in Java 1.5 generics), and do not consider the inference of wildcard types. They report no empirical results.

Von Dincklage and Diwan [20] also present a combined approach for the parameterization of classes and for the inference of actual type parameters in clients of those classes. Similarly to Duggan [8], their tool (Ilwith) creates one type parameter per declaration, then uses heuristics to merge type parameters. Our system differs in its (1) algorithm, (2) implementation, and (3) evaluation. (1) Ilwith is unsound, due to insufficient type constraints — it is missing those for preserving erasure for methods and fields, and overriding relationships between methods. As a result, the behavior of both the library and its clients may change after parameterization, without warning; this makes the technique unsuitable in practice. By contrast, our approach is correct (see Section 4.3.1) and uses heuristics only to choose among legal solutions. Unlike our approach, Ilwith does not handle key features of Java generics such as raw types and wildcards. To control run time and the number of constraint variables, Ilwith uses special cases in the algorithm to handle other Java features, such as calls to static methods and methods in generic classes, and context-, field-, and instance-sensitivity; by contrast, our system is more uniform, and we have not found performance to be a problem. Ilwith creates maximally many type parameters and then tries to merge them via heuristics (though other heuristics, such as the requirement that every field declaration mentions a type parameter, may leave the result over-general). By contrast, our technique starts with no type parameters and incrementally adds them. (2) We mention only two differences between the two implementations. First, Ilwith does not rewrite source code, but merely prints method signatures without providing details on how method *bodies* should be transformed. Second, Ilwith took "less than 2 minutes" per class on a 2.66 GHz machine, whereas our implementation averaged less than 3 seconds per class on a 2.2 GHz machine. (3) The experimental evaluation of the two tools differs as well. Ilwith was evaluated on 9 data structures (5 lists, 1 stack, 1 set, and 2 maps) chosen from two libraries (47 classes altogether, including inner classes, interfaces and abstract classes). The authors made whatever edits were necessary to enable Ilwith to succeed, so the classes are most like the pre-parameterized libraries in our evaluation. However, the authors did not evaluate the accuracy of the solution, either via examination by a Java expert or via comparison to existing parameterized versions of the libraries (then available, for example, from the GJ project and from JDK 1.5 beta releases). Even the example signatures shown in the paper differ from what a programmer would have written manually, for example in `addAll`, `contains`, `equals`, `putAll`, `remove`, and `removeEntryForKey`. (The authors state that the results are consistent with Eiffel, but a Java programmer performing a refactoring is more likely to care about Java semantics and backward compatibility.)

Previous work by the present authors includes two algorithms [7, 10] that, given a set of generic classes, infer how client code can be updated by inferring actual type parameters and removing casts that have been rendered redundant. This paper extends the constraint formalism and implementation of [10]. As discussed in Section 3.1, the most significant of these extensions are the introduction of context constraint variables and wildcard constraint variables with corresponding extensions of the solver, and the use of heuristics to guide the solver towards solutions preferred by human programmers. Although [10] includes a mode for inferring method type parameters by means of a context-sensitive analysis, it does not infer class type parameters. Other previous work uses type constraints for refactorings related to generalization [19], customization of library classes [5], and refactorings for migrating applications between similar library classes [1]. The INFER TYPE refactoring by Steimann *et al.* [18] lets a programmer select a given variable and determines a minimal interface that can be used as the type for that variable. If such an interface does not yet exist, it is created automatically. Steimann *et al.* only present their type inference algorithm informally, but they constraints appear similar to those of [19].

A number of authors have explored compile-time para-

metric type inference to ease the burden of explicit parameterization in languages supporting parametric types [14, 16, 12]. Many of these approaches were applied to functional programming languages, and thus focus on introducing type parameters for functions, rather than for classes or modules. Such languages differ from Java in the lack of a class hierarchy with inheritance and overriding, or in the use of structural (cf. nominal) subtyping, or in the lack of the notion of type erasure. These differences in semantics and structure necessitate significantly different constraint systems. Moreover, the type systems in many functional languages (e.g., ML) induce a unique *principle type* for each program variable, whereas in our case the constraint system leaves the possibility to select a desirable result from a software engineering perspective, which is a critical concern for source-to-source transformations. Yet other works describe parametric type inference even for languages with mandatory explicit parameterization for the purpose of statically type checking such diverse languages as Cecil, constraint logic programming [9] or ML. Again, these works differ from ours in many critical details of the language's type system.

Siff and Reps [17] translate C functions into C++ function templates by using type inference to detect latent polymorphism. Opportunities for introducing polymorphism stem from operator overloading, references to constants that can be wrapped by constructor calls, and from structure subtyping. De Sutter et al. [4] perform code compaction via link-time inference of reusable C++ object code fragments. Their work reconstitutes type-parametric functions from template instantiations created during compilation, but does so using code similarity detection rather than type inference, and on a low-level representation (object code).

6 Conclusion

We have presented a solution to the parameterization problem, which involves adding type parameters to existing, non-generic class declarations. This is a complex task due to requirements of backward compatibility and behavior preservation, the existence of multiple type-correct solutions, complications posed by raw and wildcard types, and the necessity to simultaneously parameterize and (generically) instantiate multiple interrelated classes. Our algorithm handles all these issues and the full Java language.

Our parameterization algorithm subsumes previous algorithms for generic instantiation, which change library clients to take advantage of libraries that have been made generic. Our analysis computes an instantiation at the same time as it performs parameterization.

We have implemented our algorithm in the context of the Eclipse IDE and run experiments to verify its correctness, accuracy, and usability. The results are correct: they are backward-compatible and they maintain behavior. The results are even more accurate than parameterizations performed by the library authors: 9% of the tool results are better, and 4% of the tool results are worse.

Acknowledgments

Gilad Bracha suggested the "parameterize like superclass" tool feature. Martin Aeschlimann and Terence Parr examined the parameterizations created by our tool. This work was supported by DARPA contracts NBCH30390004 and FA8750-04-2-0254.

References

[1] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA*, pages 265–279, Oct. 2005.

[2] G. Bracha, 2005. Personal communication.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, pages 183–200, Oct. 1998.

[4] B. De Sutter, B. De Bus, and K. De Bosschere. Sifting out the mud: Low level C++ code reuse. In *OOPSLA*, pages 275–291, Oct. 2002.

[5] B. De Sutter, F. Tip, and J. Dolby. Customization of Java library classes using type constraints and profile information. In *ECOOP*, pages 585–610, June 2004.

[6] A. Donovan and M. Ernst. Inference of generic types in Java. Technical Report MIT/LCS/TR-889, MIT, Mar. 2003.

[7] A. Donovan, A. Kieżun, M. S. Tschantz, and M. D. Ernst. Converting Java programs to use generic libraries. In *OOPSLA*, pages 15–34, Oct. 2004.

[8] D. Duggan. Modular type-based reverse engineering of parameterized types in Java code. In *OOPSLA*, pages 97–113, Nov. 1999.

[9] F. Fages and E. Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1(6):751–777, 2001.

[10] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP*, pages 71–96, July 2005.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, third edition, 2005.

[12] S. P. Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP*, Sept. 2006.

[13] A. Kieżun, M. D. Ernst, F. Tip, and R. M. Fuhrer. Refactoring for parameterizing Java classes. Technical report, MIT, Sept. 2006.

[14] A. Ohori and P. Buneman. Static type inference for parametric classes. In *OOPSLA*, pages 445–456, Oct. 1989.

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

[16] A. Rodriguez, J. Jeuring, and A. Loh. Type inference for generic Haskell. Technical Report UU-CS-2005-060, Utrecht Univ., 2005.

[17] M. Siff and T. Reps. Program generalization for software reuse: From C to C++. In *FSE*, pages 135–146, Oct. 1996.

[18] F. Steimann, P. Mayer, and A. Meißner. Decoupling classes with inferred interfaces. In *SAC*, pages 1404–1408, Apr. 2006.

[19] F. Tip, A. Kieżun, and D. Bäumer. Refactoring for generalization using type constraints. In *OOPSLA*, pages 13–26, Nov. 2003.

[20] D. von Dincklage and A. Diwan. Converting Java classes to use generics. In *OOPSLA*, pages 1–14, Oct. 2004.