A Comparison of Generics in Major Imperative Programming Languages

Joe Einertson
University of Minnesota Morris
eine0017@morris.umn.edu

ABSTRACT

This paper discusses syntax and implementation of generics in three major programming languages: Java, C++ and C#. Additionally, it discusses the adoption of generics and evaluates claims about the efficacy of generics in improving code.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types, constraints, polymorphism*

General Terms

Design, Languages, Reliability, Standardization

Keywords

Java, C++, C#, Java, generic programming, generics, type erasure, templates

1. INTRODUCTION

The three programming languages discussed within this paper are Java, C++ and C#. All three are object-oriented languages.

C++ is a compiled language. In compiled languages, source code, the human-readable representation of a program, is translated directly into machine instructions. This process is called compilation.

Java and C# are not directly compiled languages. Source code in these languages is first translated into a representation called bytecode, which is neither human-readable nor directly executable[1] [1]. Bytecode is then interpreted and executed by a virtual machine (VM).

Generic types, commonly referred to as generics, are a way of defining data structures that may be used with a single data type. For example, a list which allows any object to be inserted is not generic. However, one which allows a programmer to specify the type of objects that may be inserted is generic.

[1]Microsoft refers to the C# analog of bytecode as *intermediate language* (IL)

UMM CSci Senior Seminar Conference, Spring 2012 Morris, MN.

2. FUNDAMENTALS OF GENERICS

Generic, or parametric, types is a term that refers to types which support formal parameters, or a symbol that denotes a type. An instance of a generic class replaces the formal parameter with an actual parameter, or concrete type. This allows developers to define structures which may hold any data type, but only one type per instance. For example, a generic list may contain numbers, characters, or any other type (but not all at once), without a different implementation required for each.

2.1 Advantages of Generics

The use of generics has several advantages in software development. First, due to the flexibility of generics, code may easily be reused and adapted to many different situations. Second, generics improve the safety of a program by rigidly defining and enforcing the types of data an object may contain or perform operations on [15]. Third, generic code is generally easier to read and understand than its non-generic equivalent. Consider the following Java code snippet which does not make use of generics:

```
ArrayList list = new ArrayList();
list.add(1);
Object element = list.get(0);
if (!(element instanceof Integer)) {
    throw new InvalidTypeException("Expected an
      Integer, but received a different type.");
}
Integer integerElement = (Integer) element;
```

In the above code, the `ArrayList` is not genericized; it allows any object to be added to the list. When an element in the list is retrieved using the `get()` method, we must first verify that it is of the expected type, Integer. If not, an error is thrown and execution ends. Once we have verified the element's type, we must typecast it to the correct type, Integer. Only then may we safely perform operations on the element. This results in duplication of type verification code throughout a software project [15].

The same code may be rewritten to utilize Java generics, simplifying the code and making it more type-safe. If we use generics to specify that our list is a list of Integers, the

Java VM will enforce this. Only Integers may be added to the list, and thus only Integers can be retrieved.

```
ArrayList<Integer> list
  = new ArrayList<Integer>();
list.add(1);
Integer element = list.get(0);
```

This is much more concise: five lines of code were eliminated. We begin by initializing a List, as before, but this time we add a type parameter `<Integer>`. This parameter specifies the type of element that may be added, and we can be sure the retrieved element is of the intended type. Any attempt to add a non-Integer item will automatically result in an error. This restriction that the list only contain integers is called *parametrization*: we say that the list was parametrized over Integers.

Although we used Java for all examples in this section, the issues discussed are not language-specific. Generics provide the same benefits in C# and C++ as in Java.

2.2 Generics in C# and C++

Generics in C# have a very similar syntax to generics in Java. Consider the following code sample:

```
List<int> IntList = new List<int>();
IntList.Add(1);
int element = IntList[0];
```

As before, we begin by initializing a List object. To parametrize our list over integers, we again add a type parameter `<int>` to our list declaration. This ensures that we may only add integers to and retrieve integers from our list; any attempt to insert another type will result in an error.

C++ also supports generics, although its syntax varies slightly from that of Java and C#. Consider the following C++ code sample, which is equivalent to the previous C# example:

```
list<int> intlist;
intlist.push_front(1);
int element = intlist.front();
```

We again begin by declaring our list, but this time we do not need the `new` keyword used in Java and C#. We can then add an integer to our list, and as before we can safely retrieve it with the guarantee that it is indeed an integer.

Due to C++'s implicit type conversions, operations with many basic types are not fully type-safe [10]. For example, adding a floating-point number such as 1.7 to our list is acceptable. Other less obvious conversions are also completely valid: adding the boolean `true` or the character `'a'` both compile and run without error. However, because they were converted into integers at the time they were added, we can be sure any item retrieved from the list will be an integer.

Unlike Java and C#, C++ offers support for the typedef structure. Typedef is essentially an aliasing command, enabling complicated generic objects to be referred to by another name [11]. Consider the following nested generic declaration in Java:

```
ArrayList<Node<Integer, String>> nodes
  = new ArrayList<Node<Integer, String>>();
```

The above code creates an ArrayList containing Nodes, each of which contain an Integer and a String. The declaration is cumbersome and repetitive, and Java does not provide a mechanism to simplify the code. Using a C++ typedef, however, we can define a similar structure under the name `nodes_t`:

```
typedef list<node<int, string>> nodes_t;
nodes_t nodes;
```

Using typedef, the lengthy generic declaration was reduced to the easily-readable `nodes_t`. Because typedef's aliasing treats our custom type as a full-featured type, we can create a list of nodes simply by declaring a variable of type `nodes_t` and giving it a name (such as nodes).

3. ADVANCED GENERIC FEATURES

So far we have discussed the basics of generics in Java, C++ and C#, as well as some basic differences. Aside from a few differences in syntax and language features unrelated to generics, all three languages have been largely similar. Now we will examine some of the more advanced features of generics, many of which vary wildly between languages.

3.1 Boxing and Unboxing

In Java, there is a distinction between *primitive types*, such as int, float or boolean, and *object types*, such as ArrayList, Integer or Node. By convention, primitive types are spelled lower-case, and object types spelled with capital first letters.

In Java, the only types which may be used as parameters to a generic object are object types. Attempting to declare a `List<int>` is invalid because the type `int` is a primitive type. This begs the question: how does one place `int`s into a generic object?

The answer is via *boxing* and *unboxing*. All primitive types have an equivalent object type which acts as a wrapper for the primitive type. For example, the primitive type `int` has an equivalent object type `Integer`. Primitive types may be stored in their object type, as in the following code:

```
int x = 3;
Integer y = new Integer(x);
int z = y.intValue();
```

The above code creates an instance of the object type `Integer` to hold the value of the primitive type `int`. This process is called boxing. The reverse process, retrieving a primitive type from its object type, is called unboxing. This is achieved through methods in the object type, as demonstrated on the last line of the above example.

Given that primitive types are known to be more efficient and easier to work with, most raw data is held in primitive types. Boxing primitive types for use in generic classes, however, is tedious and redundant. For this reason, Java performs auto-boxing and auto-unboxing.

When attempting to insert a primitive type into a generic object with an equivalent object type parameter, the Java compiler will perform the boxing operation automatically. Similarly, when attempting to retrieve a primitive type from a generic object, the Java compiler will automatically perform the unboxing operation.

Although C# also performs auto-boxing and auto-unboxing, it is less obvious. Object methods may be performed on primitive types (called *value types* in C#) without error, and value types may be declared as type parameters of generic classes [12]. This is in part due to the fact that C# generates specific, optimized implementations of generic classes for value types [6]. C++ avoids the issue altogether in that there is no practical distinction between primitive and object types, at least as related to generics.

3.2 Type Constraints

Up until now, all our generics examples have allowed any type parameter to be used (with the exception of primitive types in Java). This is not always desirable. For example, one may want to limit the type parameter in a generic class to objects that can be compared to each other, or objects upon which arithmetic may be performed. Consider the following pseudo-Java example:

```
class PriorityList<T> {
    T getMaxPriority() { ... }
    T getMinPriority() { ... }
}
```

The above example defines a class `PriorityList` which takes any type `T` and stores objects in some order. The user can then use the methods `getMaxPriority` and `getMinPriority` to determine the items in the list with the highest and lowest priorities. This raises an interesting question: what if the type `T` cannot be compared, or can be compared multiple ways? How does one compare two Files and determine which has a higher priority? Are Strings ordered by length or alphabetically?

To prevent these issues, Java and C# both provide some method of defining constraints on generic type parameters. Constraints allow one to limit type parameters to objects that support certain methods or implement certain interfaces. To modify the PriorityList to accept only comparable objects, we can declare a type constraint allowing only objects which implement an interface Comparable to be added:

```
interface Comparable<T> {
    int compareTo(T other);
}

class PriorityList<T extends Comparable<T>>
```

The constraint added enforces that any type `T` with which we create a List must implement an interface Comparable, and in particular, `T` must be comparable to other objects of type `T`. Because every object implementing Comparable must implement its method `compareTo`, we can be sure that we have a consistent way to compare and order objects in our `PriorityList`.

Note that although `Comparable<T>` is an interface, we use the keyword `extends` in our generic declaration. The Java compiler will not accept the keyword `implements` in a generic declaration.

C# uses a similar syntax to define constraints, but it is more flexible than Java's. As in Java, C# allows constraints to specify a required interface it must implement. In addition, it allows constraints limiting type parameters to object types (*reference types* in C#), primitive types (*value types*), or objects with a default constructor:

```
class PriorityList<T> where T: Comparable<T>
class Foo<T> where T: struct
class Bar<T> where T: class
class Baz<T> where T: new()
```

In the above example, the first declaration requires type `T` to implement the interface `Comparable<T>` as before. The next two lines limit `T` to value types and reference types, respectively. The final line requires `T` to have a default (empty) constructor. Constraints may be combined, and multiple type parameters may all have their own constraints, allowing declarations such as the following:

```
class Foo<T, U>
  where T: class, Comparable<T>
  where U: class, new(), FileInterface
```

The above class requires that type `T` be a reference type implementing the `Comparable<T>` interface. It also requires type `U` to be a reference type with a default constructor implementing some interface `FileInterface`.

C++ does not provide support for type constraints in generic types as a language feature [9]. However, type constraints can be emulated and the same effect achieved by the use of function pointers and exploiting C++'s static nature. Bjarne Stroustrup, designer of C++, advocates for the use of these tricks in place of constraints [16], but their implementation is complicated and beyond the scope of this paper.

3.3 Type Inference in Java 7

Java 7, released in October 2011, introduced type inference for generic types to reduce redundant type declarations [14]. Consider the nested generic declaration first discussed in section 2.2:

```
ArrayList<Node<Integer, String>> nodes
  = new ArrayList<Node<Integer, String>>();
```

Prior to Java 7, there was no way to shorten that declaration. With type inference, however, the generic type parameters may be dropped from the `new` declaration, giving the following code:

```
ArrayList<Node<Integer, String>> nodes
  = new ArrayList<>();
```

The Java 7 compiler will infer the type parameters for the ArrayList by copying the type parameters on the left-hand side of the assignment operator.

The purpose of generic type inference is broadly similar to the purpose of typedefs in C++: to reduce duplication in code. With type inference, the type parameters to a generic class are only required once per instantiation, as opposed to twice without type inference.

4. METHODS OF IMPLEMENTATION

We have now seen how generics benefit code quality and type safety in Java, C# and C++. Although the end result is very similar in all three languages, how each language implements generics internally varies considerably.

4.1 Instantiation in C++

C++ uses *instantiation* to implement generics. Instantiation means that for each parametrization of an object,

a separate non-generic class is created at compilation time. For example, in a program that utilizes both a `list<int>` and a `list<char>`, the C++ compiler creates two classes, one for `int`s and one for `char`s, under the covers [15].

Instantiation is beneficial in the case of C++ because its primary goal is efficiency. All types are known at linking time, and that includes generics. By creating fully-functioning copies of classes, all overhead related to generics is avoided. Although instantiation can result in some duplication in the compiled executable, this duplication is avoided in the source code, and the efficiency central to C++'s philosophy is preserved.

As we will see in section 5, in real life software projects, most generic classes are parametrized over relatively few types. This means even though C++ creates a separate class for each parametrization, in practice very little duplication actually occurs.

4.2 Type Erasure in Java

Java does not utilize instantiation in its implementation of generics. Instead, it uses a process called *type erasure*.

Java generics are a relatively recent addition to the language, having been added in Java 1.5, in 2004 [11]. To maintain compatability with pre-1.5 VMs which did not support generics, the decision was made to implement generics in such a way that generic code could be run on legacy VMs. This requires an implementation that does not require adding new features to the bytecode. The implementation chosen was type erasure [5].

In type erasure, the type parameters of generic classes are erased at compile time. In their place, all generic references are replaced with references to the `Object` class. All objects in Java descend from the `Object` class, so theoretically, any object could be placed into the erased implementation [1]. To prevent this, all generic references are enforced at compilation time. Additionally, any variables which receive their value from a method which returns a generic type are given typecasts, as in the following example:

```
String s = stringList.get(0);
String s = (String) stringList.get(0);
```

The first line of the above code is what a developer would write. At compilation time, the bytecode representation would be replaced with the second line. This combination of type erasure and compiler enforcement create a link between the type parameters of generic classes and the non-generic bytecode.

At compilation time, generic classes are erased to their simplest form in the bytecode representation. This means all type parameters are removed. For example, the method `void add(T element)`, which accepts any element of type `T`, is translated to `void add(Object element)` at compilation time. Nested generic declarations are also erased. The declaration `List<Map<Integer, String>>` is erased to `List` at compilation time, and any references to a specific item from the list would be erased to `Map`.

In some instances, specifically when a generic class narrows the bound of another generic class, type erasure is not sufficient to enforce the declared type. In these instances, *bridge methods* are automatically created [5]. Bridge methods are methods with the specific type parameters required by the code that feed calls to the bridge method into the wider bounded method.

Now the add method accepts any Object, but performs typecasts within the method body to ensure objects added are truly Strings [5]. Any attempt to add an object which can not be typecast to String, such as Integer, will result in a ClassCastException being thrown [15].

The downside to type erasure is that the VM cannot determine the type of a generic class at runtime; its type has been erased. This also leads to some non-obvious compiler errors. Consider the following code snippet from a Java class:

```
int foo(List<String> list) {
    /* Do something with Strings */
}
int foo(List<Integer> list) {
    /* Do something different with Integers */
}
```

The above code will fail to compile because the type parameters of both Lists will be erased. This results in two methods with the same signature, `int foo(List)` [13]. If the return type or method name of either were changed, it would compile. To accomplish the intended effect of creating a method which will work for both Strings and Integers, however, one must instead create a method `int foo(List)`, and then test the contents of the list for their types. Given that generics are supposed to eliminate such tedious concerns, type erasure in this case backfires, rendering generics useless for the given application.

Type erasure in Java also prevents arrays of generic objects from being created. For example, creating an array `List<String>[]` would be erased to `List[]` at compilation time. Because arrays are low-level data structures with very little overhead, the VM has no way of enforcing that a given `List` in the array is in fact a `List<String>` [6].

4.3 Reification in C#

In contrast to both instantiation and type erasure, C# implements generics in such a way that eliminates duplication without resorting to the removal of generic parameters. Although this required the introduction of new instructions in the C# bytecode, thus breaking backwards compatibility, it results in a more flexible generics implementation [12]. The implementation used is called *reification*. Reification is a general term referring to a process by which an abstract concept such as generics can be accessed and referred to concretely [3].

Reification in C# results in several features not available in Java, mostly as related to *reflection*. In computer science, reflection refers to a computer program able to access and modify type information at run time. For example, in C#, a program may determine whether an object is generic, and if so, what its actual type parameters are [12]. A C# program may even modify existing generic parameters, or create new generic types and methods from existing ones. Additionally, arrays of generics may be created in C#. None of this is possible in Java because all references to generic types are fully erased by runtime.

5. GENERICS ADOPTION

Generics were introduced to Java in 2004 [15], and to C# in 2005 [2]. In excess of seven years have passed since their introduction, begging the question: did generics make a difference in program length, complexity and type safety?

5.1 Java: A Case Study

Given that Java is widely used and studied, we will focus on the adoption of Java generics. Many claims have been made about the effects of generics, including the following:

- Generics reduce code duplication [7]
- Generics reduce or eliminate runtime type errors [15]
- Generics encourage and enable standardization and consistency [7]
- Generics lower programmers' cognitive load [15]

Are these claims substantiated by empirical studies? Although it appears very few studies have studied the adoption and efficacy of generic programming, the data seems encouraging [15].

5.2 Generics in Standard Libraries

First, we will briefly examine the effect of generics adoption in standard Java libraries. A study conducted in 2005 observed a code duplication rate of 68% in the non-generic Java Buffer library. By converting the existing code to use generics where possible, 40% of the duplicate code was removed [4]. Similarly, a separate 2005 study showed that by refactoring several Java programs, 91% of compiler warnings were eliminated, as well as nearly half the casts [8].

These studies seem to support the claim that generics reduce code duplication and errors. Studies focusing primarily on standard libraries, however, fail to evaluate the usage and effectiveness of software projects "in the wild." This begs the question: how can one truly measure the adoption rate of a new language feature in new software projects? One answer is open source projects.

5.3 Generics in Open Source Projects

Open source software (OSS) projects have several advantages in evaluating generics adoption. First, due to the transparency of the OSS process, code is freely available for study. Second, most OSS projects use some form of version control, allowing one to track changes to a code base over time. Third, since most OSS projects have no central leadership or upper management structure, decisions about code style are made primarily by the programmers themselves. This allows studies of OSS projects to investigate the effectiveness of generics with the elimination of some outside variables.

In 2011, Parnin et al conducted an in-depth study of 20 major Java OSS projects and their adoption of generics, totalling over 500 million lines of code [15]. Their analysis of the data provides insight into whether projects choose to use generics and why.

Eight of the twenty projects analyzed utilized more parametrized types than raw (non-generic) types. The remaining twelve all had more raw types, including five which had no generic types at all [15]. It is obvious from this data that generics adoption is not universal, ranging from 0% adoption to 100% adoption. Interestingly, the size of projects seems not to affect the use of generics, implying other forces besides the size of the code base play into generics conversion.

Additionally, generics adoption was far from universal by individual developers as well. Only 14% of developers, and 27% of the most active developers, created or modified generic types [15]. This suggests that although active developers with more involvement tended towards generics adoption, it was not a priority for them.

Some patterns also emerged in common instantiations of generic types. In every project, `List<String>` was the most common instantiation. About 25% of instantiations were of the type `List<String>` or `Map<String, String>`.

Interestingly, most generic instantiations were parametrized over relatively few types. One-third of generic types were parametrized over a single type, and 80% over fewer than five types. This is especially interesting to note given that C++ creates a new representation of a class for each instantiation. Although C++'s method technically results in duplication once compiled, the amount of duplication may be rather modest in practice. It is unclear whether these statistics hold true in C++ as well as Java.

6. EVALUATING CLAIMS

We previously listed four main claims put forth by proponents of generics, and we will now evaluate these claims based on the data shown above.

6.1 Code Duplication

The first claim we listed is that using generics reduces code duplication. This is a related, although distinct, issue from C++'s duplication of machine code. To measure potential code duplication, Parnin et al devised a formula to calculate the total number of clones of each class required if generics were not used.

Before the introduction of generics, in situations where type safety was essential, programmers would often create specific classes tailored to a single type, such as `IntegerList` or `StringIntPair`. These are often referred to as *clones*. Had these same conventions been followed on every instantiation of the top ten genericized classes, over 4000 clones would have been required, an average of over 400 per class [15]. Of the remaining genericized classes, only 5.8 clones per class would have been required, still a non-trivial amount of duplication.

Obviously, generics did not supplant clones in every instance, or even a majority of instances, in the software projects. Regardless, given the inherent lack of type safety of non-generic classes, even a small fraction of clones migrating to generics represents a significant reduction in code duplication. Parnin et al estimate that in the top 27 genericized classes alone, over 107,000 lines of duplicate code were eliminated by the introduction of generics, indicating that generics do indeed reduce code duplication [15].

6.2 Errors

The second claim listed is that generics reduce runtime type errors. As generics are enforced at compilation time, it stands to reason that they would increase compilation errors at the expense of runtime errors.

Although it is difficult to directly analyze the incidence of errors in any software project, Parnin et al devised a formula which estimates errors as a function of duplicated lines and the number of revisions to those lines. In their analysis of the top 27 genericized classes, they estimate 400 errors were eliminated, based on an estimation of .01 errors per commit, and .0074 errors per line of code derived from a previous survey [15].

6.3 Standardization and Cognitive Load

The last two claims are that generics increase standardization and decrease cognitive load. These claims are more subjective than the previous two, but we can still evaluate them to some degree.

First, we evaluate the claim that generics increase standardization. There is no way to empirically measure standardization within a software project, but arguably the very idea of generics enables standardization. The philosophy behind generics is to allow data structures to hold a single type without requiring a separate implementation for each type. Therefore, by eliminating clones and other duplicate code, generics enforce standardization by unifying implementations of the same concept for different type parameters.

Second, we evaluate the claim that generics decrease programmers' cognitive load. As before, there is no way to empirically measure cognitive load, but we can infer something about cognitive load from the complexity of code. Generic code is designed to be simpler than non-generic code, and many claim it eliminates many instances of typecasts, a common source of errors and increased cognitive load. Parnin et al found little evidence of a correlation between generics adoption and a decrease in typecasts, however [15]. In fact, in at least one project, they found that generics adoption resulted in an *increase* in typecasts. Additionally, in only one software project was the correlation statistically significant. This does not necessarily mean generic code has no effect on cognitive load, but it does weaken the claim significantly.

7. CONCLUSION

Having examined the syntax and implementation of generics in three major languages, it is obvious that no single language or implementation is perfect, and indeed that each has weaknesses and strengths. However, all three do allow type parametrization, simplify code and increase type safety.

Java's use of type erasure results in some restrictions on the flexibility of generics, although whether these restrictions are issues in practice is debatable. C++ sidesteps these issues by using instantiation, but lacks native support for type constraints, eliminating some of the beneficial effects of generics. C# suffers from neither of these issues because it made a clean break with its past and introduced new bytecode commands, as well as expanding on Java's type constraint system. The disadvantage of this approach is that it breaks backwards compatibility.

It is clear from our discussion and evaluation of claims regarding the efficacy of generics that adding genericity to a language is beneficial. We can say with relative certainty that generics reduce code duplication. Additionally, given that errors are a function of program size, reducing the amount of code in a project is likely to reduce its number of errors.

Generics implicitly increase standardization, but there are very few ways to empirically measure this. Despite some circumstantial evidence, we found no conclusive evidence as to whether standardization affected cognitive load in any significant way.

To conclude, although generics should not be considered a panacea, they do have the ability to increase type safety and reduce errors in a program. Additionally, they generally simplify code while reducing duplication, both important measures of code quality. We conclude that generics are a beneficial addition to a language, and that even a generic system with some flaws is better than no generic system at all.

8. ACKNOWLEDGEMENTS

I would like to thank my faculty advisor Elena Machkasova for her tireless proofreading and revising, and my external reviewer Matthew Justin for his keen eye for even the smallest mistakes.

9. REFERENCES

[1] Java (programming language) - Wikipedia. `http://en.wikipedia.org/w/index.php?title=Java_%28programming_language%29`, March 2011.

[2] .NET Framework - Wikipedia. `http://en.wikipedia.org/w/index.php?title=.NET_Framework_version_history`, March 2011.

[3] Reification (computer science) - Wikipedia. `http://en.wikipedia.org/w/index.php?title=Reification_%28computer_science%29`, March 2011.

[4] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. An empirical study on limits of clone unification using generics. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE '05)*, pages 109–114, 2005.

[5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. *SIGPLAN Not.*, 33:183–200, October 1998.

[6] S. T. Devendra Gahlot, S. S. Sarangdevot. Run time polymorphism against virtual function in object oriented programming. In *International Journal of Computer Science and Information Technologies*, pages 569–571, 2011.

[7] A. Donovan, A. Kiežun, M. S. Tschantz, and M. D. Ernst. Converting Java programs to use generic libraries. *SIGPLAN Not.*, 39(10):15–34, Oct. 2004.

[8] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 71–96, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming, 2005.

[10] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. *SIGPLAN Not.*, 38:115–134, October 2003.

[11] D. Ghosh. Generics in Java and C++: a comparative model. *SIGPLAN Not.*, 39:40–47, May 2004.

[12] A. Kennedy and S. Don. Design and implementation of generics for the .NET Common Language Runtime. *SIGPLAN Not.*, 36:1–12, May 2001.

[13] M. Naftalin and P. Wadler. *Java Generics and Collections*. O'Reilly Media, 2007.

[14] Oracle. Type inference for generic instance creation. `http://docs.oracle.com/javase/7/docs/technotes/guides/language/type-inference-generic-instance-creation.html`, 2011.

[15] C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 3–12, New York, NY, USA, 2011. ACM.

[16] B. Stroustrup. C++ style and technique FAQ. `http://www2.research.att.com/~bs/bs_faq2.html`.