REVERSE GENERICS
Parametrization After The Fact

Alexandre Bergel
PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Santiago, Chile
http://www.bergel.eu

Lorenzo Bettini
Dipartimento di Informatica, Università di Torino, Italy
http://www.di.unito.it/~bettini

Keywords: Generic Programming, Java Generics, C++ Templates

Abstract: By abstracting over types, generic programming enables one to write code that is independent from specific data type implementation. This style is supported by most mainstream languages, including C++ with templates and Java with generics. If some code is not designed in a generic way from the start, a major effort is required to convert this code to use generic types. This conversion is manually realized which is known to be tedious and error-prone.
We propose *Reverse Generics*, a general linguistic mechanism to define a generic class from a non-generic class. For a given set of types, a generic is formed by unbinding static dependencies contained in these types. This generalization and generic type instantiation may be done incrementally. This paper studies the possible application of this linguistic mechanism to C++ and Java and, in particular, it reviews limitations of Java generics against our proposal.

1 Introduction

The concept of generic programming (Dos Reis and Järvi, 2005), which has characterized functional programming for several decades, appeared in mainstream programming object-oriented languages such as C++, only in the late 80s, where it motivated from the beginning the design of the Standard Template Library (STL) (Musser and Stepanov, 1989; Musser and Saini, 1996; Austern, 1998). Generic programming was not available in the first versions of Java, and this limited code reuse, by forcing programmers to resort to unsafe operations, i.e., type casts. *Generics* are a feature of the Java 1.5 programming language. It enables the creation of reusable parameterized classes while guaranteeing type safety.

In spite of the limitations of Java generics, type parameterization allows the programmer to get rid of most type down-casts they were forced to use before Java generics; this also is reflected in part of the standard Java library which is now generic. However, much pre-1.5 Java code still needs to be upgraded to use generics. For example, a quick analysis on the AWT Java library shows that some classes perform more than 100 down-casts and up-casts and 70 uses of instanceof. This examination reveals that in many places the amount of up-casting subsequent down-casting that is used almost makes the programs behave like dynamically typed code.

Note, that the need to make existing code generic may arise also in languages where generic types were already available. In particular, either a library or a framework is designed in a type parametric way from the very beginning, or the "conversion" must be done manually, possibly breaking existing code. Several proposals have been made that promote an automatic conversion from non-generic code into generic code (Duggan, 1999; von Dincklage and Diwan, 2004; Kiezun et al., 2007). These approaches involve reverse-engineering techniques that infer the potential type candidates to be turned into parameters. This paper presents a different approach: instead of relying on the programming environment to pick up type parameters, a programmer may create a generic class starting from a non-generic one, by using a proper language construct. To our knowledge, no previous attempt to offer a language built-in mechanism to generalize classes has been proposed.

We propose to extend the way generics are expressed by defining a generic type (class or interface) from a non-generic type definition. Our approach consists of an extension to be applied to an existing object-oriented programming language. However, in this paper, we will investigate the possible application of this extension to Java and C++. We will also demonstrate how the implementation of generic types in the language affects the usability of this new linguistic extension. With this extension, programmers can create generic classes and interfaces starting from existing classes and interfaces by specifying the types that need to be turned into generic parameters. We call this linguistic mechanism *Reverse Generics*.

This approach is different from the above mentioned refactoring approaches since in our context there will be no refactored code: the starting class will continue to exist after it is used to create its generic version. However, we believe that the refactoring techniques in the literature could formulate a refactoring based on our reverse generics to actually implement the refactoring.

The paper is organized as follows. Section 2 presents and illustrates *Reverse Generics*. Section 3 enumerates several typing issues due to the Java type system with respect to generic programming. Section 4 presents briefly related work and Section 5 concludes and presents future work.

2 Reverse Generics

Reverse Generics is an extension for object-oriented programming languages that enables a generic class to be created starting from an existing class, and a generic interface from an existing interface. Some of the static type references contained in the original class or interface are transformed into type parameters in the resulting generic version. The process of obtaining a generic class from a class definition is called *generalization*. We consider *generalization* as the dual of the *instantiation* operation. We refer to *unbinding* a type when references of this type contained in the original class definition are not contained in the generic.

Given a class name, ClassName, and a type name, TypeName, the generic version of a class is denoted by the following syntactic form:

ClassName>TypeName<

All references to TypeName contained in the class ClassName are abstracted. A name is associated to the abstract type in order to be concretized later on. Several type names may be abstracted using the following writing:

ClassName>TypeName$_1$, ..., TypeName$_n$<

The resulting generic class should then be assigned to a class definition, which depends on the actual programming language (and in particular on its syntax for parameterized types); thus, in Java we would write:

```
class MyGenericClass<T extends TypeName> =
        ClassName>TypeName<;
```

In C++ we would instead write

```
template<typename T>
class MyGenericClass<T> =
        ClassName>TypeName<;
```

The class resulting from a reverse generic should be intended as a standard class in the underlying language. We could simply instantiate the type parameters of a generic class and then create an object, e.g.,

```
new MyGenericClass<MyTypeName>();
```

However, there might be cases (e.g., when using partial instantiation, Section 2) where it is useful to simply instantiate a generic class and assign it another class name; thus, we also consider this syntax:

```
class MyClass = MyGenericClass<MyTypeName>;
```

In the following, we informally describe and illustrate Reverse Generics with several examples resulting from an experiment we conducted on the AWT graphical user interface Java library. The same mechanism may be applied to C++ templates.

Class Generalization. The class EventQueue is a platform-independent class that queues events. It relies on AWTEvent, the AWT definition of event. The code below is an excerpt of the class EventQueue:

```
public class EventQueue {
  private synchronized AWTEvent getCurrentEventImpl() {
    return (Thread.currentThread() == dispatchThread)
      ? ((AWTEvent)currentEvent.get()): null;
  }
  public AWTEvent getNextEvent()
    throws InterruptedException {
    ...
  }
  public void postEvent(AWTEvent theEvent) {
    ...
    boolean notifyID = (theEvent.getID() == this.waitForID);
    ...
  }
  ...
}
```

In some situations, the EventQueue class may have to be used with one kind of event, say KeyEvent. This will significantly reduce the number of runtime downcasts and ensure type safety when such a queue has to be used.

By using reverse generics we can define the generic class GEventQueue<T extends AWTEvent> from the non-generic class EventQueue as follows:

```
class GEventQueue<T extends AWTEvent> =
        EventQueue>AWTEvent<;
```

GEventQueue<T extends AWTEvent> is a generic definition of EventQueue that contains a particular data type, T. A type has to be provided to GEventQueue in order to form a complete class definition. To satisfy the Java type checker, a constraint has to be set on T by enforcing it to be a subtype of AWTEvent. For example, in the method postEvent(...), getID() is invoked on an event. The getID() method is defined in the class AWTEvent. Without the constraint on T, GEventQueue<T> would be rejected by the Java compiler since it can not statically guarantee the presence of getID().

The generic GEventQueue<T> resulting from the snippet of code given above is equivalent to:

```
public class GEventQueue<T extends AWTEvent> {
    private synchronized T getCurrentEventImpl() {
        return (Thread.currentThread() == dispatchThread)
            ? ((T)currentEvent.get()): null;
    }
    public T getNextEvent() throws InterruptedException {
        ...
    }
    public void postEvent(T theEvent) {
        ...
        boolean notifyID = (theEvent.getID() == this.waitForID);
        ...
    }
    ...
}
```

References of AWTEvent have been replaced by the type parameter T. GEventQueue is free from references of the AWT event class definition. The queue may be employed with KeyEvent then, a subclass of AWTEvent:

```
GEventQueue<KeyEvent> keyEventsQueue
        = new GEventQueue<KeyEvent>();
keyEventsQueue.postEvent(new KeyEvent(...));
try {
    KeyEvent event = keyEventsQueue.getNextEvent();
} catch(Exception e) {} // getNextEvent() is throwable
```

Interface Generalization. The mechanism for classes described above may be applied to interfaces. For example, the AWT ActionListener interface is defined as follows:

```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

This interface may be generalized with the following declaration:

```
public interface GActionListener<T> =
        ActionListener>ActionEvent<;
```

The benefit of this generalization is the ability to reuse the interface ActionListener with a different event API.

Incremental Generalization. A generic class obtained using reverse generics may be generalized further by unbinding other remaining static type references. For instance, let us consider the class EventDispatchThread, which is a package-private AWT class which takes events off the EventQueue and dispatches them to the appropriate AWT components. EventDispatchThread is used in the EventQueue class as follows:

```
public class EventQueue {
  ...
  private EventDispatchThread dispatchThread;

  final void initDispatchThread() {
    synchronized (this) {
      if (dispatchThread == null &&
      !threadGroup.isDestroyed()) {
        dispatchThread = (EventDispatchThread)
          AccessController.doPrivileged(new PrivilegedAction()
      {...}}}}
```

In the situation where some assumptions may have to be made on the type of the event dispatcher, the GEventQueue<T extends AWTEvent> may be generalized further:

```
public class
GenericEventQueue<Dispatcher extends EventDispatchThread>
    =GEventQueue>EventDispatchThread<;
```

The generic GenericEventQueue has two type parameters, T and Dispatcher. Note that the definition above is equivalent to the generic class obtained by unbinding both the AWTEvent and the EventDispatchThread type in one single step:

```
public class GenericEventQueue
    <T extends AWTEvent,
    Dispatcher extends EventDispatchThread>
        =EventQueue>AWTEvent, EventDispatchThread<;
```

The class GenericEventQueue<T,Dispatcher> can be instantiated by providing the proper two type parameters.

At the current stage of reverse generic, Incremental Generalization assumes that the two parameters are distinct types. If not, then the generalization cannot be applied.

Partial Instantiation. The generic GenericEventQueue described above may be partially instantiated by fulfilling only some of its type parameters. For example, an event queue dedicated to handle key events may be formulated:

```
public class GKeyEventQueue =
    GenericEventQueue<KeyEvent>;
```

One type argument has still to be provided to GKeyEventQueue, i.e., the one corresponding to the type parameter Dispatcher. A complete instantiation may be:

```
public class OptimizedKeyEventQueue
    = GKeyEventQueue<OptimizedEventDispatchThread>;
```

OptimizedKeyEventQueue has all the type parameters instantiated and can be used to create objects:

```
OptimizedKeyEventQueue keyEventsQueue
        = new OptimizedKeyEventQueue();
keyEventsQueue.postEvent(new KeyEvent(...));
try {
    KeyEvent event = keyEventsQueue.getNextEvent();
} catch(Exception e) {} // getNextEvent() is throwable
```

3 Dealing with the language features

Reverse generics is a general mechanism that can be used to extend an existing programming language that already provides a generic programming mechanism. However, since this mechanism relies on the existing generic programming features provided by the language under examination, we need to investigate whether such existing mechanisms are enough to create a complete "reversed generic" class. This section summarizes the various technical limitations of Java generics and their impact on reverse generics. In this respect, C++ does not seem to put limitations for reverse generics. A broader comparison between the generic programming mechanisms provided by Java and C++ may be found in the literature (Ghosh, 2004; Batov, 2004).

Class instantiation. A crucial requirement in the design of the addition of generic in Java 1.5 was backward compatibility, and in particular to leave the Java execution model unchanged. *Erasure* (Odersky and Wadler, 1997; Bracha et al., 1998) is the front-end that converts generic code into class definitions. It behaves as a source-to-source translation. Because of erasure, List<Integer> and List<String> are the same class. Only one class is generated when compiling the List<T> class. At runtime, those two instantiations of the same generic List<T> are just Lists. As a result, constructing variables whose type is identified by a generic type parameter is problematic.

Let's consider the following code:

```
class PointFactory {
    Point create() { return new Point(); }
}
```

One might want to generalize PointFactory the following way:

```
class Factory<T> = PointFactory>Point<;
```

The corresponding reversed generics class would correspond to the following generic class:

```
class Factory<T> {
    T create() { return new T(); }
}
```

However, this class would not be well-typed in Java: a compile-time error is raised since new Point() cannot be translated into new T() for the reason given above. As a result, the class PointFactory cannot be made generic in Java. Enabling a generic type parameter to be instantiated is considered to be a hard problem (Allen and Cartwright, 2006). A proposal has been made to eliminate such restrictions (Allen et al., 2003), in order to let generic types appear in any context where conventional types appear.

On the contrary, in C++, the above code for Factory<T> (with the corresponding C++ template syntax adjustments) is perfectly legal, and instantiation of generic types is also used in the STL itself.

Static methods. Generic class type parameters cannot be used in a static method[1]. A generic method has to be used instead.

For example, the class EventQueue contains the static method eventToCacheIndex(...):

```
private static int eventToCacheIndex(AWTEvent e) {
    switch(e.getID()) {
        case PaintEvent.PAINT: return PAINT;
        case PaintEvent.UPDATE: return UPDATE;
        case MouseEvent.MOUSE_MOVED: return MOVE;
        case MouseEvent.MOUSE_DRAGGED: return DRAG;
        default: return e instanceof PeerEvent ? PEER : -1;
    }
}
```

This method is generalized in GenericEventQueue as the following:

```
private static <U extends AWTEvent>
    int eventToCacheIndex(U e) {
    ...
}
```

The type parameter U cannot be equal to T since eventToCacheIndex(...) is a static method. This means that eventToCacheIndex(...) may be employed with a type T_1 even if GenericEventQueue has been invoked with a type T_2. For example, we might have:

```
class GEventQueue<T extends AWTEvent>
    = EventQueue>AWTEvent<;...
GEventQueue<KeyEvent> keyEventQueue
```

[1] http://java.sun.com/docs/books/tutorial/-extra/generics/methods.html

```
    = new GEventQueue<KeyEvent>();
GEventQueue.eventToCacheIndex(new ActionEvent(...));
```

The expression GEventQueue<KeyEvent> instantiates the generic with the type KeyEvent. However, the static method eventToCacheIndex is performed with an ActionEvent, a subclass of AWTEvent living in a different class hierarchy than KeyEvent. While this does not undermine type safety, we believe that it might represent an abnormal situation with respect the initial design intentions. This issue suggests an extension of reverse generic to handle static methods as a possible further investigation.

On the contrary, C++ deals with generic class type parameters used in static methods, as in the following code (the implicit type constraint is that the operator << is defined for T, which is the case for the basic types we use in main):

```
template<typename T>
class ClassWithStaticMethod {
    T myField;
public:
    // constructor initializing the field
    ClassWithStaticMethod(const T& t) : myField(t) {}

    static void m(T t) {
        cout << "t is: " << t << endl;
    }
};

int main() {
    ClassWithStaticMethod<int>::m(10);
    ClassWithStaticMethod<float>::m(10.20);
    ClassWithStaticMethod<string>::m("foobar");

    // create an object of class ClassWithStaticMethod<string>
    // passing a string argument
    ClassWithStaticMethod<string> c("value");
    c.m("hello");
    c.m(10); // compile ERROR!
}
```

Note also how the C++ compiler correctly detects the misuse of a static method in the last line: the static method of a class where the generic type is instantiated with string is being used with another type (int)[2].

Abstract class. Turning a type contained in a method signature into a type parameter may make the resulting generic class abstract in Java. Consider the following two class definitions:

```
abstract class AbstractCell {
    public abstract void set (Object obj);
    public abstract Object get ();
}
```

[2]A static method invoked on an instance corresponds to the static method invoked on the instance's class.

```
class Cell extends AbstractCell {
    private Object object;
    public void set (Object obj) { this.object = obj; }
    public Object get () { return this.object; }
}
```

One may want to write the following generic to make Cell operate on Number instead of Object:

```
class GCell<T extends Number> = Cell>Object<;
```

However, GCell is abstract since set(Object) is not implemented. The solution is to make AbstractCell generic by abstracting Object. In C++, since it does not type check a generic class, but only its instantiations (*i.e.,* generated classes), the situation is different, as illustrated in the next section.

Method erasure uniqueness. Consider the previous code excerpt of AbstractCell and Cell. Let us assume one wants to make Cell operate on any arbitrary type instead of Object. Naively, one may write the following definition:

```
class GCell<T> = Cell>Object<;
```

It is the same definition of GCell provided above without the upper type. This definition results in a compile error. The reason is that the method set has two different erasures without being overriding. This is a further limitation of the Java type erasure.

To fully understand why, consider the following example. This code is rejected by the Java compiler (with the error *"GCell is not abstract and does not override abstract method set(java.lang.Object) in AbstractCell"*):

```
class GCell<T> extends AbstractCell {
    private T object;
    public void set (T obj) { this.object = obj; }
    public T get () { return this.object; }
}
```

This is due to the Java erasure mechanism which prevents two methods from having the same erasure if one does not override the other. The method set(T) in GCell<T> and set(Object) in AbstractCell have the same erasure. The former does not override the latter, but overloads it. An illustration of this limitation is:

```
class MyClass<U,V> {
    // These two overloaded methods are ambiguous
    void set (U x) { }
    void set (V x) { }
}
```

Defining an upper bound of the type T will enforce this overloading and removes the method erasure ambiguity. A compilable version could be:

```
class MyClass<U extends Object, V extends java.awt.Frame> {
    void set (U x) { }
    void set (V x) { }
}
```

Let us now consider a possible reversed generic GCell generated in C++, which could correspond to the following one:

```
class AbstractCell {
public:
    virtual void set(int o) = 0;
    virtual int get() = 0;
};

template<typename T>
class GCell: public AbstractCell {
    T object;
public:
    GCell(T o) : object(o) {}
    virtual void set(T o) { object = o; }
    virtual T get() { return object; }
};

int main() {
    AbstractCell *cell = new GCell<int> (10);
    cout << cell->get() << endl;
    cell->set(20);
    cout << cell->get() << endl;
    AbstractCell *cell2 =
        new GCell<string> ("foo"); // compile ERROR
}
```

C++ correctly considers GCell<int> as a concrete class since the abstract methods of the base class are defined[3]. However, if we tried to instantiate GCell<string> we would get a compiler error, since GCell<string> is considered abstract: in fact, the get/set methods in the abstract class are not implemented (GCell<string> defines the overloaded version with string, not with int).

Primitive types. Arithmetic operators in Java have to be used in a direct presence of numerical types only. As an example, the + and - operators can only be used with primitive types and values. The autoboxing mechanism of Java makes it operate with the types Integer, Float.

For example, the following generic method is illegal since Number objects cannot be arguments of + and -:

```
public class T<U extends Number> {
    public int sum (U x, U y) {
        return x + y;
    }
}
```

Instead, the following declaration of sum is legal:

```
public class T<U extends Integer> {
    public int sum (U x, U y) {
        return x + y;
    }
}
```

This means that one can reverse generic a class by abstracting the type Integer into a parameter U extends Integer. However, this would not be highly useful since Integer is a final class, sum can be applied only with the type Integer.

The use of arithmetic operations prevents the operand types from being turned into type parameters in a generic way. This is not a problem in C++ thanks to operator overloading (a feature that is still missing in Java), as also illustrated in the following section.

Operators. Java does not provide operator overloading, but it implements internally the overloading of +, for instance, for Integer and String. Thus, the two classes are legal in Java:

```
public class IntSum {
    public static Integer sum (Integer x, Integer y) {
        return x + y;
    }
}

public class StringSum {
    public static String sum (String x, String y) {
        return x + y;
    }
}
```

But there is no way to extract a generic version, since there is no way to write a correct type constraint[4].

This is not a problem in C++ thanks to operator overloading. However, we think that this problem is not strictly related to the absence of operator overloading in Java. Again, It is due to type erasure and how the type-checking is performed in Java. C++ does not perform type-checking on the generic class: upon type parameter instantiation it type-checks the resulting (implicitly) instantated class; thus, we can write in C++ such a generic class with method sum, which will have only some accepted (well-typed) instantiations, i.e., those that satisfy the implicitly inferred constraints (in our case, the operator + must be defined on the instantiated type). On the contrary, Java type-checks the generic class itself, using the explicit constraint, which in our case, cannot be expressed in such a way that it is generic enough.

[3] To keep the example simple we used int as a type, since no Object is available in C++.

[4] This might be solved, possibly, with a *union type* (Igarashi and Nagira, 2007) constraint such as, e.g., extends Integer $\vee$ String.

4 Related Work

To our knowledge, no programming language construct to build a generic class from a complete class definition has been presented in the literature. This section presents the closest work to Reverse Generics.

Reverse engineering parameterized types. A first attempt to automatically extract generic class definitions from an existing library has been conveyed by Duggan (Duggan, 1999), well before the introduction of generics into Java.

Beside the reverse engineering aspect, Duggan's work diverges from Reverse Generics regarding downcast insertion and parameter instantiation. Duggan makes use of *dynamic subtype constraint* that inserts runtime downcast. Parameterized type may be instantiated, which requires some type-checking rules for the creation of an object: the actual type arguments must satisfy the upper bounds to the formal type parameters in the class type. Moreover, the version of generics presented in his work with PolyJava differs from Java 1.5 in several important ways that prevent his results from being applied to Java generics.

Modular type-based reverse engineering. Kiezun et al. proposes a type-constraints-based algorithm for converting non-generic libraries to add type parameters (Kiezun et al., 2007). It handles the full Java language and preserves backward compatibility. It is capable of inferring wildcard types and introducing type parameters for mutually-dependent classes.

Reverse engineering approaches ensure that a library conversion preserves the original behavior of the legacy code. This is a natural intent since such a conversion is exploited as a refactoring. The purpose of Reverse Generics is to replace static types references contained in existing classes with specialized ones. Section 3 shows that a generic obtained from a complete class may have to be set abstract. This illustrates that the original behavior of the complete class may not be preserved in the generic ones. Method signatures may be differently resolved in the generic class.

Type construction polymorphism. A well-known limitation of generic programming in mainstream languages is to not be able to abstract over a type constructor. For instance, in List<T>, List is a type constructor, since, given an argument for T, e.g., Integer, it builds a new type, i.e., List<Integer>. However, the type constructor List itself cannot be abstracted (this is a well known limitation of first-order parametric polymorphism). Thus, one cannot pass a type constructor as a type argument to another type constructor. Moors, Piessens and Odersky (Moors et al., 2008) extend the Scala language (Odersky et al., 2008) with type construction polymorphism to allow type constructors as type parameters. Thus, it is possible not only to abstract over a type, but also over a type constructor; for instance, a class can be parameterized over Container[T][5], where Container is a type constructor which is itself abstracted and can be instantiated with the actual collection, e.g., List or Stack, which are type constructors themselves.

Reverse generics act at the same level of first-order parametric polymorphism, thus, it shares the same limitations, e.g., the following reverse generic operation cannot be performed:

```
class MyClass {
    List<Integer> mylist;
}

class MyClassG = MyClass>List<;
```

An interesting extension is to switch to the higher level of type constructor polymorphim, but this is an issue that still needs to be investigated, and, most important, it should be experimented with a programming language that provides type constructor polymorphism, and, with this respect, Scala seems the only choice compared to Java and C++.

5 Conclusion

Genericity in programming languages appeared in the beginning of the 70s. It gained a large adoption by being adopted in mainstream languages. All the generic mechanisms we are aware of enable a parameterization only if the code has been prepared for being parametrized. This paper goes against this implicitly established mindset. Reverse generics promote a generalization for code that has not been prepared for it.

Since highly parameterized software is harder to understand (Gamma et al., 1995), we may think of a programming methodology where a specific class is developed and tested in a non-generic way, and then it is available to the users via its "reversed" generic version (thus, in this case, we really need the non generic version for testing purposes, so the code must not be refactored). For example, C++ debuggers may have problems when setting a breakpoint for debug

[5] Scala uses [] instead of <>.

purposes within a template from a source file: they may either miss setting the breakpoint in the actual instantiation desired or may set a breakpoint in every place the template is instantiated. Another well-known problem with programming using templates is that usually the C++ compilers issue quite long compilation errors in case of problems with template usage and in particular with template instantiations; these errors are also hard to understand due to the presence of parametric types.

Thus, reverse generics can be used as a development methodology, not only as a way to turn previous classes into generic: one can develop, debug and test a class with all the types instantiated, and then expose to the "external world" the generic version created through reverse generics. Provided that an explicit dependency among reversed generic classes and the original ones is assumed (e.g., by using makefiles), the reversed generic version of a class will be automatically kept in sync with the original one.

Classes obtained with reverse generics are not related to the original classes. We think that this is the only sensible design choice since generic types and inheritance are basically two distinguished features that should not be mixed; indeed the main design choices of Java generics tend to couple generics and class based inheritance (again, for backward compatibility), relying on type erasure, and, as we discussed throughout the paper, this highly limits the expressivity and usability of generics in a generic programming methodology. C++ keeps the two above features unrelated; in particular, the STL library basically does not rely on inheritance at all (Musser and Stepanov, 1989; Musser and Saini, 1996; Austern, 1998), leading to a real usable generic library (not to mention that, avoiding inheritance and virtual methods also leads to an optimized performance).

We currently described reverse generics in a very informal way by describing a surface syntax and its application to Java and C++. We plan to investigate the applicability of reverse generics also to other programming languages with generic programming capabilities such as, e.g., C# and Eiffel (Meyer, 1992).

As a future work, we will seek a stronger and deeper theoretical foundation. The starting point could be Featherweight Java (Igarashi et al., 2001), a calculus for a subset of Java which was also used for the formalization of Java generics. Alternatively, we might use the framework of (Siek and Taha, 2006), which, working on C++ templates that provide many more features than Java generics, as we saw throughout the paper, seem to be a better candidate for studying the advanced features of reverse generics.

REFERENCES

Allen, E., Bannet, J., and Cartwright, R. (2003). A First-Class Approach to Genericity. In *Proc. of OOPSLA*, pages 96–114. ACM.

Allen, E. E. and Cartwright, R. (2006). Safe instantiation in generic java. *Sci. Comput. Program.*, 59(1-2):26–37.

Austern, M. H. (1998). *Generic Programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley.

Batov, V. (2004). Java generics and C++ templates. *C/C++ Users Journal*, 22(7):16–21.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. (1998). Making the future safe for the past: adding genericity to the Java programming language. In *Proc. of OOPSLA*, pages 183–200. ACM.

Dos Reis, G. and Järvi, J. (2005). What is generic programming? In *Proc. of LCSD*.

Duggan, D. (1999). Modular type-based reverse engineering of parameterized types in java code. In *Proc. of OOPSLA*, pages 97–113. ACM.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

Ghosh, D. (2004). Generics in Java and C++: a comparative model. *ACM SIGPLAN Notices*, 39(5):40–47.

Igarashi, A. and Nagira, H. (2007). Union Types for Object Oriented Programming. *Journal of Object Technology*, 6(2):31–52.

Igarashi, A., Pierce, B., and Wadler, P. (2001). Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450.

Kiezun, A., Ernst, M. D., Tip, F., and Fuhrer, R. M. (2007). Refactoring for parameterizing java classes. In *Proc. of ICSE*, pages 437–446. IEEE.

Meyer, B. (1992). *Eiffel: The Language*. Prentice-Hall.

Moors, A., Piessens, F., and Odersky, M. (2008). Generics of a higher kind. In *Proc. of OOPSLA*, pages 423–438. ACM.

Musser, D. R. and Saini, A. (1996). *STL Tutorial and Reference Guide*. Addison Wesley.

Musser, D. R. and Stepanov, A. A. (1989). Generic programming. In Gianni, P. P., editor, *Proc. of ISSAC*, volume 358 of *LNCS*, pages 13–25. Springer.

Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala*. Artima.

Odersky, M. and Wadler, P. (1997). Pizza into Java: Translating theory into practice. In *Proc. of POPL*, pages 146–159. ACM.

Siek, J. and Taha, W. (2006). A semantic analysis of C++ templates. In *Proc. of ECOOP*, volume 4067 of *LNCS*, pages 304–327. Springer.

von Dincklage, D. and Diwan, A. (2004). Converting Java classes to use generics. In *Proc. pf OOPSLA*, pages 1–14. ACM.