Generics without Type Erasure in Javali

Tim Linggi
Department of Computer Science
ETH Zurich
linggit@ethz.ch

Jonathan Meier
Department of Computer Science
ETH Zurich
jmeier@ethz.ch

Abstract

Generics as in Java and C# or templates in C++ parlance are a powerful language feature in many common object oriented programming languages. We introduce generics with syntax and semantics inspired by Java's generics to Javali, a programming language used in the Bachelor's Compiler Design course at ETH Zurich. Our implementation features new symbol types enabling type checking in the presence of generic types, as well as a type inference facility that greatly enhances the usability of generics. Moreover, in contrast to Java, we do not use type erasure. Instead, we consider each generic instantiation a separate type and present a novel approach called *runtime type tracking* allowing us to emit code of a generic class or method only once, independent of the number of instantiations.

1 Introduction

Generics allow to parametrize types in the definition of code constructs, such as classes or methods, which subsequently can be instantiated with different specific types all at compile time and thus remove the need for code duplication, where code constructs only differ by the set of types they use. A prominent use case is a class providing a list implementation, which can be parametrized with the type of the objects it should hold.

However, implementing generics can be done in various ways. For example, generics in Java are fundamentally different from generics in C# or templates in C++. While Java and C# do type checking directly on the generic class (*early type checking*), C++ type checks only after instantiation and on the instantiation itself (*late type checking*). Moreover, in Java, all instantiations of a generic class have the same type (*type erasure*), while in C# and C++ they are considered different types.

We bring generics to Javali in the form of an extension we call JavaliGen. Syntactically and semantically, our implementation is mostly inspired by Java's generics, since Javali itself is a subset of Java. However, we do not want to copy the exact behavior of generics from any language. Therefore, in contrast to Java, our generics do not use type erasure, that is as for C++ templates each generic instantiation is considered a separate type, which makes them more expressive. Moreover, as opposed to C++, which emits separate (assembly) code for each different instantiation (*monomorphic code*), we emit code only once, independent of the number of instantiation (*polymorphic code*), as is the case in Java.

In the remainder of this paper, we first describe the features of our implementation and their background in other languages in Section 2. In Sections 3, 4 and 5, we discuss details and design choices of the implementation of type checking, code generation and type inference, respectively. Finally, in Section 6, we evaluate both the technical and functional aspects of JavaliGen and lay out future work before we close with a conclusion in Section 7.

2 Features and Background

Generic classes are the bare minimum of generics, with generic methods being almost as important to make generics usable. Both, classes and methods, can have an arbitrary number of type parameters. Note that we distinguish between the terms *type parameter* and *type argument* and use them consistent with the definitions in the Java documentation[1].

Moreover, we allow bound constraints on the generic parameters, a feature that is available with Java's and C#'s generics but not in C++ templates. Upper bounds define the interface that can be used on the associated generic type parameters, therefore enabling us to do type checking on the generic type, as opposed to C++, where type checking is done on each different template instantiation separately. Since the default upper bound, the common base class `Object`, does not provide any functionality in Javali, having no upper bounds would severely limit the usability of the generics.

Furthermore, Java's generics have a feature called wildcards. However, in order to reduce the complexity of our implementation, we omit support for wildcards. In exchange, we remove some of Java's shortcomings originating from backwards compatibility constraints. In particular, and similar to C# generic types and C++ template types, our implementation does not do type erasure on generic types.

The absence of type erasure enables the implementation of various features, including:

- Method overloading with generic types
- Declaration of classes with the same name but different number of type parameters
- The creation of objects[2] and arrays whose type is a type parameter (as in C# with the `new()` constraint)

[1] https://docs.oracle.com/javase/tutorial/java/generics/types.html

[2] Provided there are no constructors (as in JavaliGen) or a mandatory default constructor that takes no arguments

Advanced Compiler Design AS'17, January 5th, ETH Zurich 2018.

Listing 1. Examples of features implemented in JavaliGen.

```
1  class A {}
2  class B {}
3  // generic class with two type parameters
4  // the second type parameter S has upper bound A
5  class Example<R, S extends A> {
6    // generic method with a single type parameter
7    // the type parameter has upper bound B
8    <T extends B> void bar(T t) {...}
9  }
10 // second generic class with same name but different
11 // number of type parameters
12 class Example<T> {
13   <S> void foo() {
14     S s; T[] t;
15     // object creation with a type parameter
16     s = new S();
17     // array creation with a type parameter
18     t = new T[1];
19   }
20   void bar() {
21     Example<A> a; boolean b;
22     a = new Example<A>();
23     // valid use of instanceof with a generic class
24     b = a instanceof Example<A>;
25   }
26 }
```

- Using the `instanceof` operator with generic classes (mostly equivalent to the `is`/`as` operators in C# and the `dynamic_cast` operator in C++)

All these features, as shown in Listing 1, are implemented in JavaliGen with the exception of method overloading. Javali does not support method overloading in the first place, therefore, we decided against introducing this feature.

To improve the user experience, a basic version of Java's type inference algorithm is implemented. While without type inference, the type arguments of a generic class or method have to be specified explicitly, type inference allows the compiler to infer those arguments from context for certain expressions.

3 Type Checking

Generics introduce an important new aspect in type checking not known from Javali: Not all types used in a program need to be defined explicitly in the code. More specifically, a generic class is defined once, however, the number of types derived from this generic class via instantiation is determined by the number of different instantiations throughout the whole program, which can be zero, one or more. Note in particular that a generic class definition alone does not introduce a type by itself. Moreover, observe that a non-generic class can simply be seen as a generic class with zero generic type parameters, which implies that the number of different instantiations in the program can be at most one. This observation greatly simplifies type checking, since we do not need to differentiate between generic and non-generic classes. Therefore, in the remainder of this section, we will refer to both generic and non-generic classes simply as classes.

Since JavaliGen is heavily inspired by Java and we emit polymorphic code, as described in more detail in Section 4, we apply early type checking. Each class is type checked exactly once, independent of the number of different instantiations. However, this is only possible since in JavaliGen, there is a single root of the inheritance hierarchy (`Object`) and therefore every generic type parameter can be upper bounded. For type checking a class, each generic type parameter is simply replaced by its upper bound, or by `Object` if there is no explicitly specified upper bound. If the upper bound itself is a generic type parameter, we recursively follow the upper bounds until we find an actual type. Some details on the implementation will be discussed in the following.

As described in the beginning of this section, there is a difference between a class definition and its derived types, i.e. instantiations. This differentiation leads us to the introduction of two different types of symbols in the type checker, namely abstract symbols and instantiated symbols. Abstract symbols always refer to a specific declaration or definition in the code, for example, a declaration of a field or variable or a definition of a method or class.

This entails that an abstract class symbol contains abstract symbols for its fields and methods and an abstract method symbol contains abstract symbols for its parameters and local variables. In contrast, instantiated symbols always refer to a use of a declaration or definition, e.g. accessing a field or variable or invoking a method. Moreover, instantiated symbols always wrap their corresponding abstract symbol and, most importantly, they complement the wrapped abstract symbol with the type information of the instantiation they represent. For example, an instantiated class symbol wraps its corresponding abstract class symbol and upon querying its fields and methods, it returns instantiated field and method symbols, that again wrap their corresponding abstract symbols from the abstract class symbol but additionally provide the type information from the instantiated class symbol.

The type information attached to an instantiated symbol is a mapping of generic type parameters to instantiated type symbols. This mapping allows us to resolve a generic type parameter to its actual instantiated type for a specific instantiation.

Consider Listing 2 as an example to illustrate how abstract and instantiated symbols play together in order to check whether Line 15 is semantically correct. The difficulty in this example lies in verifying, if the generic type parameter `S` actually extends the generic type parameter `R` on Line 3. Note that this upper bound constraint is necessary for method `foo` to be semantically correct, since only references to a subtype

of R can be assigned to r. However, for this example, we are not concerned with the actual type checking of method foo and simply consider it semantically correct. In the method invocation on Line 15, the upper bound constraint is satisfied, since both S and R resolve to the same type U, which happens to be a generic type parameter itself.

For class A<R>, there is an abstract class symbol that contains both an abstract variable and method symbol for field r and method foo. For class B<T>, there is an abstract class symbol that contains an abstract variable symbol for field at. Abstract variable symbols have an instantiated type symbol attached. In this case, it is an instantiated class symbol for the type A<T> wrapping the abstract class symbol for class A<R>. The type information of the instantiated class symbol for the type A<T> is a mapping from the generic type parameter R to the instantiated generic type parameter T. Note that all classes have an instantiated type symbol for each of its generic type parameters. For class C<U>, there is an abstract class symbol that contains both an abstract variable and method symbol for field bu and method bar. Again, the abstract variable symbol for field bu has an instantiated class symbol for the type B<U> attached. The mapping for this instantiated class type maps the generic type parameter T to the instantiated type parameter U.

Finally, on Line 15, we first need to find the instantiated class symbol for the field at of the object stored in bu. Therefore, we can query the instantiated class symbol attached to bu for the field at. This will return an instantiated variable symbol wrapping the abstract variable symbol of the field at with a type mapping that maps the generic type parameter T to the instantiated generic type symbol U. If we now query the instantiated variable symbol for its type, this will give us an instantiated class symbol whose mapping now resolved to the generic type parameter R mapping to the instantiated generic type parameter U. From this newly resolved instantiated class symbol, we can get the instantiated method symbol for foo, that already resolves the generic type parameter R to the instantiated generic type parameter U. The method invocation itself provides the argument for the generic type parameter S, namely the instantiated generic type parameter U. Thus, resolving S and R in the context of this method invocation will result in the instantiated generic type parameter U for both types, which of course satisfies the constraint, since instantiated generic type parameters extend themselves.

Listing 2. Example to illustrate type checking of Line 15. The difficulty is to resolve both generic type parameters S and R of method foo to the generic type argument U in the method invocation, in order to check the upper bound constraint.

```
1  class A<R> {
2    R r;
3    <S extends R> void foo(S s) {
4      r = s;
5    }
6  }
7  class B<T> {
8    A<T> at;
9  }
10 class C<U> {
11   B<U> bu;
12   void bar(U u) {
13     // at is of type A<U>, so calling foo with type
14     // U is perfectly fine
15     bu.at.<U>foo(u);
16   }
17 }
```

4 Code Generation

There are basically two different approaches on code generation: monomorphic and polymorphic code emission. Monomorphic code emission is what C++ uses. For each instantiation of a class, the code of the class, or at least the code of those methods depending on the generic type parameters, is emitted separately. Polymorphic code emission means, that for each defined class, the whole code is emitted only once, independent of the number of different instantiations. Through type erasure, Java actually does polymorphic code emission, since instantiations of generic classes do not exist in the bytecode anymore.

The advantage of monomorphic code emission is that for each instantiation, the code can be optimized separately, however, it can also increase the binary size significantly, if there are many different instantiations of large classes. The advantage of polymorphic code emission is that for many different instantiations of large classes, the binary size increases less than with monomorphic code emission, however, no individual optimization per instantiation is possible. Since JAVALIGEN is mostly based on Java, which, as mentioned above, uses polymorphic code emission, we decided to emit polymorphic code for JAVALIGEN as well.

Consider the example in Listing 3. We want to emit polymorphic code for method foo. Note that on Line 12 there are two instantiations of the generic class D<T>, hence foo has to be polymorphic for both types D<A> and D<B>. Consequently, when invoking foo on Line 14, we need to create a new object of type C<A> and C<B> inside foo on Line 7.

Creating a new object involves allocating memory, but, more importantly in our example, it also requires writing the virtual table (*vtable*) address of the object type into the first address of the newly allocated memory. Since we have no type erasure, each type, i.e. each instantiation of a class, requires its own vtable, but inside foo the vtable address is not known at compile time. In the following, we introduce a mechanism we call *runtime type tracking* that enables finding the correct vtable address efficiently at runtime.

Each vtable is augmented with instantiation indexes. An instantiation index corresponds to a class and is a unique

value per different instantiation of this class between zero and the number of different instantiations minus one. The order of the index is not important and therefore it is assigned arbitrarily by the compiler. A vtable is composed of a pointer to the parent vtable and a copy of the contents from the parent vtable followed by the own additional entries. The instantiation index is added as an entry after the own additional entries. The content of the parent vtable is copied, including the parent instantiation index.

Figure 1 shows the two vtables for the types `D<A>` and `D<B>`. Since class `D<T>` inherits directly from `Object`, we see the vtable pointer to `Object` as the first entry. The `Object` vtable has no function pointers, so the only entry is its instantiation index, which is 0. Note that the instantiation index for all non-generic classes is always 0, since there can only ever be at most one instantiation. After the instantiation index of the parent, the entry for the method `foo` follows and finally the own instantiation index, 0 for `D<A>` and 1 for `D<B>`.

Moreover, on the right-hand side of the tables in Figure 1, we see a global list storing pointers to all vtables used in the program. This list is generated by the compiler and does not need to be in any particular order.

Finally, we can describe how the vtable pointer for the new object created on Line 7 is determined. From the vtable of the `this` reference, we can obtain the instantiation index of the type of the object on which `foo` was invoked. All we need now, is a translation from this instantiation index to the global vtable list, in order to pick the right vtable pointer. This is achieved by using a generic table (*gtable*) associated with the `new` operator as shown in Figure 1.

In summary, each time, type information that depends on a generic type parameter is required, as it is the case with the `new` operator in the example, we introduce a new gtable that translates the instantiation index to the required type information. Type information is needed for new object and new array expressions, as well as for the `instanceof` operator and casts. Moreover, since we allow new object expressions of the form `new T()`, where `T` is a generic type parameter, we can use the same approach with a global object size table, where we can lookup the size of memory that needs to be allocated in order to create an object of type `T`.

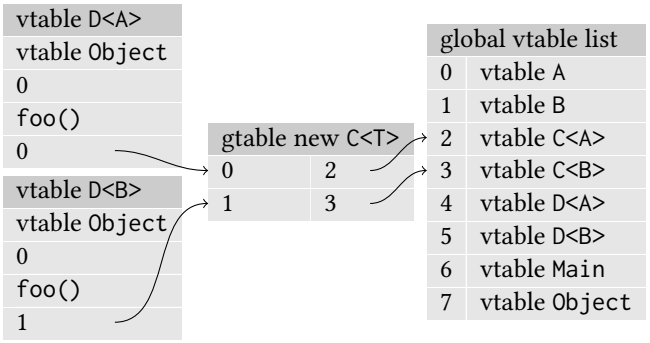
5 Type Inference

When using generics, type inference is a tool that vastly improves usability. It describes the concept of inferring the type arguments of generic classes or methods from context, so the user does not have to declare them explicitly. In JavaliGen, there are two expressions, where type inference can be used: instantiations of a generic class and method calls to generic methods. In the following subsections, we first discuss, how type inference is implemented in JavaliGen, followed by examples illustrating the described concepts.

Listing 3. Example to illustrate our runtime type tracking mechanism.

```
1  class A {}
2  class B {}
3  class C<T> {}
4  class D<T> {
5    C<T> ct;
6    void foo() {
7      ct = new C<T>();
8    }
9  }
10 class Main {
11   void main() {
12     D<A> da; D<B> db;
13     da = new D<A>(); db = new D<B>();
14     da.foo(); db.foo();
15   }
16 }
```

Figure 1. Tables and their relationships introduced for runtime type tracking when compiling the code shown in Listing 3.



5.1 Type Inference in JavaliGen

The type inference algorithm implemented in JavaliGen is inspired by Java's type inference[3]. However, since JavaliGen is less comprehensive than Java and, for example, does not support wildcards, lambda expressions or method overloading, type inference becomes much simpler.

Before describing the algorithm, we have to introduce a new abstraction of a type, called *inference variable*. It simply represents a type which needs to be inferred. For example, in an expression `Gen<A> g = new Gen<>()`, an inference variable is created which corresponds to the missing type argument in the new object expression. Inference variables are represented with Greek letters (usually α) and can occur in constraints and bounds, just like regular types. In the remainder of this section, inference variables will be referred to as α and regular types or type parameters as `T`.

Type inference can be split into three parts, *constraint collection*, *reduction* and *resolving*. First, a set of constraints

[3] https://docs.oracle.com/javase/specs/jls/se8/html/jls-18.html

is collected, which is then reduced to a bound set on the inference variables. In a third step, the bounds are resolved, which results in a lower and upper type bound for each inference variable. Each type between those two bounds in the type hierarchy is a valid type for the inference variable (i.e. satisfies all constraints). However, the most specific one is chosen, as it is done in the Java type inference.

For collecting the constraints, initially, on a per-method basis, the current method is visited and constraints are added for new object expressions and method call expressions. There are three kinds of constraints:

- `compatible`, e.g. `new Gen<>()` $\rightarrow$ `Gen<T>`
- `subtype`, e.g. α <: `T`
- `equals`, e.g. α = `T`

A `compatible` constraint implies that the expression on the left-hand side is compatible with the type on the right-hand side. `subtype` and `equals` constraints should be self-explanatory. Constraints can be derived from assignments, method arguments, method return types and upper bounds of type parameters.

In a second step, the set of constraints is reduced to a set of bounds in a process called reduction. There are three kinds of bounds on inference variables, namely:

- Upper bound: α <: `T`
- Lower bound: `T` <: α
- Equals bound: α = `T` or `T` = α

Additionally, there is a designated `FALSE` bound which implies that the set of constraints is not satisfiable by any mapping from inference variable to regular types. This leads to a semantic error.

The third and final step consists of resolving the set of bounds for each inference variable to get the most specific valid type, which is then inferred. For each inference variable, resolving is done separately. Initially, the lower bound is set to `null` and the upper bound to `Object`. Afterwards, each bound affecting the current inference variable is handled and the lower and upper bounds for the inference variable are adjusted accordingly. For this purpose, the *greatest lower bound* and the *least upper bound* of two types may need to be determined. If there is a contradiction while resolving, a semantic error is thrown. After processing all bounds, each inference variable has a lower and upper bound, which enclose the valid types for this inference variable. Finally, the lower bound is inferred, since the most specific type is required.

5.2 Examples

First, let us consider a basic example found on Line 13 in Listing 4 for inferring a type in a new object expression, which is a very common use case. The constraints set only contains a single constraint:

$$\{\texttt{new Gen<>()} \rightarrow \texttt{Gen<A>}\}$$

Listing 4. Various examples for type inference.

```
1  class A {
2     <T> void foo(T t) {...}
3     <T> void bar(T t1, T t2) {...}
4  }
5  class B {}
6  class C extends B {}
7  class D extends B {}
8  class Gen<T> {}
9  class Main {
10    void main() {
11       Gen<A> g; A a; C c; D d;
12       ... // create locals
13       g = new Gen<>(); // infer A
14       a.foo(a);        // infer A
15       a.bar(c, d);     // infer B
16    }
17 }
```

This constraint set is reduced to the bound set:

$$\{\alpha = \texttt{A}\}$$

It directly follows that the lower and upper bound of α must be `A`, therefore the inferred type is also `A`.

In the second example on Line 14 in Listing 4, the type argument of the generic method `foo` needs to be inferred. Initially two constraints are collected:

$$\{\texttt{a} \rightarrow \alpha, \alpha \texttt{ <: Object}\}$$

The first constraint can be derived from the argument of `foo`, which takes an argument with the type of the type parameter. The second constraint is derived from the upper bound of T, which in this case is just `Object`, i.e. does not add any additional restrictions. These constraints reduce to two bounds:

$$\{\texttt{A} <: \alpha, \alpha \texttt{ <: Object}\}$$

From these two bounds, it directly follows that the lower bound of α is `A` and the upper bound `Object`, i.e. `A` is inferred.

A more involved example can be found on Line 15 in Listing 4, where the type argument of `bar` needs to be inferred. Similar to the last example, three constraints are collected:

$$\{\texttt{c} \rightarrow \alpha, \texttt{d} \rightarrow \alpha, \alpha \texttt{ <: Object}\}$$

which are reduced to the corresponding three bounds:

$$\{\texttt{C} <: \alpha, \texttt{D} <: \alpha, \alpha \texttt{ <: Object}\}$$

However, resolving these bounds is not as straight-forward. Let us assume, `C` <: α gets resolved first, which leads to the lower bound of α being `C` and the upper bound being the initial bound `Object`. Now, the bound `D` <: α is processed. To determine the new lower bound for α which satisfies both bounds, the least upper bound of the current lower bound of α, `C`, and `D` has to be determined, which is `B`. Therefore, the new lower bound of α is `B` and since the third bound does not add any additional restrictions, `B` is inferred.

6 Evaluation

The following subsections cover a technical evaluation, where we address testing and runtime overhead, and a functional evaluation presenting the advantages of JavaliGen over Java and discussing specialization. Finally, we outline possible future work.

6.1 Technical Evaluation

For testing the newly implemented features, we wrote approximately 200 test cases. The main focus lies on testing a feature first in isolation and then in combination with other features to ensure that it is integrated correctly. It is also important to write test cases for expected errors, e.g. when type checking generics. Since large parts of the default Javali code have been changed and we want to remain backwards compatible, we ran approximately 1600 test cases without generics to detect possible bugs introduced which only affect the non-generic part of JavaliGen.

We also assessed the overhead introduced by runtime type tracking. Measuring the exact overhead is difficult, since many factors have to be considered, most importantly caching effects. In theory, the overhead should be relatively small. It requires exactly 4 memory loads to retrieve a vtable pointer via a gtable, independent of the number of classes or their instantiations, meaning the overhead per lookup is asymptotically constant. This is approximately comparable to twice the overhead induced by dynamic method dispatch, which is similar in nature and requires 2 memory loads. Indeed, in a simple experiment in which we have 1000 different classes and create 1000 objects of each class, once with and once without gtable lookups, we measured an overhead of less than 4%. Note that in this experiment more than half of the expressions require a gtable lookup. In a more realistic program, the percentage of expressions requiring a gtable lookup, and thus also the overhead, would be much lower.

6.2 Functional Evaluation

To evaluate the functionality of JavaliGen, we implemented a fully working `LinkedList`, `ArrayList` and `Set`, as well as a `Tuple` and various complex test cases which aim to use all implemented features at once. Discussing these test cases in detail does not add much value, therefore we focus on two smaller examples which show nice features available in JavaliGen but not in Java.

Lines 1 and 2 in Listing 5 show two classes `Tuple`, once with one and once with two type parameters. In JavaliGen, these classes are treated as two separate types. Therefore, there are no issues with double declaration or circular inheritance. In Java, this code would not compile, because type erasure reduces both definitions to the same raw type `Tuple`.

Other limitations imposed by type erasure arise when implementing generic data structures in Java. One example is that the method `toArray()`, defined in `Collection<T>`, does not return `T[]` but rather `Object[]`. This entails that one either has to downcast the result or use the generic method `<T> T[] toArray(T[] a)`. Both approaches are not desirable and result in unnecessary code. As illustrated in Listing 5, in JavaliGen, `toArray()` can directly be implemented to return `T[]` by creating an array of a type parameter.

Listing 5. Valid JavaliGen code, which is only possible due to the absence of type erasure.

```
1  class Tuple<S, T> {}
2  class Tuple<T> extends Tuple<T, T> {}
3  // provide List implementation with toArray method
4  class ExtendedList<T> extends List<T> {
5     T[] toArray() { // returns T[] instead of Object[]
6        T[] ret;
7        ret = new T[size()];
8        ... // fill ret
9        return ret;
10    }
11 }
```

Finally, specialization of generic classes, as known from C++ template specialization, turns out not to be useful in JavaliGen. Since we do early type checking, each specialization would have to provide the same public interface as its corresponding generic class. However, in JavaliGen, classes only have a public interface and there are no private fields or methods. Therefore, a specialization could only add new fields and methods or change the behavior of a method, which is essentially the same as simply inheriting from the specific instantiation.

6.3 Future Work

A prominent feature, which would profit from the absence of type erasure is method overloading. With our implementation, it would be possible to overload methods with generic arguments. Additionally, lower bounds on type parameters could be introduced to allow more functionality. In code generation, a consolidation (merging and reuse) and lookup optimization (caching of intermediate results) of the gtables would benefit performance.

7 Conclusion

We presented JavaliGen, which extends Javali with generics without using type erasure. Various features, like the distinction of generic classes with the same name but different number of type parameters or the creation of objects of generic type parameters, are implemented and stand for an improvement over Java's generics. JavaliGen is strongly typed and uses early type checking. To emit polymorphic code, we proposed the novel approach of runtime type tracking. Finally, a basic version of Java's type inference algorithm was presented, which allows the compiler to infer type arguments from context, improving the programmer's experience.