Generics in Go, Java, and OCaml

Amy Seo

Abstract

In this report, we will be looking into the newly added support for generics in Go using parameterized types, and how this compares to the use of generics in Java and OCaml.

1. Introduction

At Metahilang, our software is written mainly in Java and Ocaml and heavily relies on generic types. Although many of our customers have requested Go versions of our libraries, their lack of support for generics has always led us to decline. However, Go has recently added support for generics using parameterized types, so we will be looking into how this use of generics compares to what we have been doing with Java and OCaml in our currently existing libraries.

2. Type Inference

In regards to type inference in Go, Greisemer and Taylor write:

"The exact details of how type inference works are complicated, but using it is not: type inference either succeeds or fails. If it succeeds, type arguments can be omitted, and calling generic functions looks no different than calling ordinary functions. If type inference fails, the compiler will give an error message, and in those cases we can just provide the necessary type arguments."

1.1. Java

In Java, the statement holds true: calling a generic method would be the same as calling any ordinary method without having to specify the type. An example of this is (3):

```
public class BoxDemo {
  public static <U> void addBox(U u,
  java.util.List<Box<U>> boxes) {
            Box<U> box = new Box<>();
            box.set(u);
            boxes.add(box);
  }

  public static void main(String[] args) }
            java.util.ArrayList<Box<Integer>>
            listOfIntegerBoxes);
            BoxDemo.<Integer>addBox(Integer.valueOf
            (10), listOfIntegerBoxes);
            BoxDemo.addBox(Integer.valueOf(20),
            listOfIntegerBoxes);
  }
}
```

In this example, `addBox` is a generic function and the type of `U` is inferred to be an Integer, so there is no need to explicitly declare the Integer type. If type inference fails, the compiler would give an error and the type would have to be explicitly given. For example, the first call to `addBox` uses a type witness to explicitly tell the function that the type is Integer, but it is not necessary in this case. Thus, the statement holds true in Java.

1.2. OCaml

In OCaml, the statement does not hold entirely true. OCaml is implicitly typed and types are rarely ever explicitly given by the programmer. In this sense, every function could be considered a generic function and so there would be no "ordinary" function to compare it to. OCaml has something called type reconstruction which combines the type-checking and type-inference process. An example of OCaml code is:

```
# let g x = 5 + x;;
val g: int -> int  = <fun>
```

OCaml uses the Hindley-Milner (HM) type inference algorithm, which makes the functional qualities of OCaml possible. However, part of the given statement is true in that sometimes there are errors where the full type of an object is not known by the compiler. An example of this is (5):

```
# let x = ref None;;
val x : '_weak1 option ref = {contents = None}
```

which gives this error message:

```
The type of this expression, '_a option ref,
contains type variables that cannot be generalized
```

Providing the compiler with a type annotation fixes the problem:

```
# let x : string option ref = ref None;;
val x : string option ref = {contents = None}
```

Thus, the statement is somewhat true in OCaml, but due to its implicit type checking, the type inference works a lot differently compared to Go and Java.

2. When to Use Generics

Taylor writes about multiple different examples of when type parameters are useful and not useful. In each of these examples, we will be seeing if the same qualities apply equally well to Java.

2.1. When Type Parameters are Useful

(1) When using language-defined container types

This statement holds in Java. An example is when using a HashMap with generic types. We can provide the type parameters as follows:

```
Map< K, V > map = new HashMap< K, V >();
```

So that it can be generalized to any types K and V.

(2) General purpose data structures

In Java, this statement is sometimes true. For example, the tree data structure given in the article could also be created with generic types in Java. However, for some general purpose data structures that have methods that do not rely on the type, wildcards can be used in Java. A wildcard is a ? symbol that denotes an unknown type. In Java, generic methods with type parameters "express dependencies among the types of one or more arguments to a method and/or its return type" (9) but if there is no dependency, a wildcard is preferred. For example:

```
class Collections {
  public static <T> void copy(List<T> dest, List<? extends T> src) {
  ...
}
```

is preferred over

```
class Collections {
  public static <T, S extends T> void copy(List<T> dest, List<S> src) {
  ...
}
```

which could be applicable when writing code referring to general purpose data structures.

(3) For type parameters, prefer functions to methods

In the example given in the article, it mainly mentioned preferring functions over methods in the case where an external dependency is required (Tree needs a comparison function for its instantiated type). However, this functionality could be implemented with the use of an interface and requiring the comparison function. Thus, this does not always hold in Java as preferring a generic function over a constraint that requires a method is entirely up to the programmer's preference.

(4) Implementing a common method

In Java, this statement holds true because if methods are implemented the same way for any type, there is no need to create separate functions for each one when it can be written with one generic function. An example of this is:

```
public static <E> void printArray( E[] inputArray ) {
    for(E element : inputArray) {
      System.out.printf("%s ", element);
    }
    System.out.println();
  }
Integer[] intArray = { 1, 2, 3, 4, 5 };
Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
printArray(intArray);
printArray(doubleArray);
printArray(charArray);
```

Instead of creating separate functions for the int, double, and character types, one generic function with a type parameter is able to fulfill all these types, just like Go.

2.2 When Type Parameters are Not Useful

(1) Don't replace interface types with type parameters

In Java, interface types are also supported, and there is no need to replace them with type parameters if all that it needed with the value is to call a method on it. There is not really much of a performance difference, so replacing them would not be needed, making the statement hold true in Java.

(2) Don't use type parameters if method implementations differ

This applies to Java as well because if the method implementations differ, you can create a generic interface and have the methods inherit from this interface and write the separate implementations that way without a type parameter within those functions. An example of this would be:

```
interface SomeInterface<T> {
  T action(T t);
}

class Class1 implements SomeInterface<Integer> {
      public Integer action(Integer i) {
          return 1+i;
      }
}

class Class2 implements SomeInterface<String> {
      public String action(String s) {
```

```
            return s + "!";
        }
}

class Class3 implements SomeInterface<Double> {
        public Double action(Double d) {
            return 1.5+d;
        }
}
```

(3) Use reflection where appropriate

Java does support runtime reflection, however, it works differently than in Go. In Go, reflection allows you to write code that works with any type during runtime. However, in Java, "we can invoke a method through reflection if we know its name and parameter types" (7), which means that the types need to be known, which is not the case. Thus, since runtime reflection works differently in both languages, the statement does not hold entirely true in Java.

3. Conclusion

The newly added support for generics in Go definitely makes it more plausible as a new language in our software. Since Go is an extremely popular language for our customers, and its use of generics is familiar to us, Metahilang will definitely add creating Go libraries into our plans for the future.

References

(1) *When To Use Generics - The Go Programming Language*. https://go.dev/blog/when-generics. Accessed 29 May 2022.

(2) *Type Inference (The Java™ Tutorials > Learning the Java Language > Generics (Updated))*. https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html. Accessed 29 May 2022.

(3) "Guide to Understanding Generics in Java." *Stack Abuse*, 5 Apr. 2021, https://stackabuse.com/guide-to-understanding-generics-in-java/.

(4) *Lecture 26: Type Inference and Unification*. https://www.cs.cornell.edu/courses/cs3110/2011sp/Lectures/lec26-type-inference/type-inference.htm. Accessed 29 May 2022.

(5) "Common Error Messages · OCaml Tutorials." *OCaml*, https://ocaml.org/docs/common-errors. Accessed 29 May 2022.

(6) *Using an Interface as a Type (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance)*. https://docs.oracle.com/javase/tutorial/java/IandI/interfaceAsType.html. Accessed 30 May 2022.

(7) "Reflection in Java." *GeeksforGeeks*, 24 Mar. 2016, https://www.geeksforgeeks.org/reflection-in-java/.

(8) *Generic Methods (The Java™ Tutorials > Bonus > Generics)*. https://docs.oracle.com/javase/tutorial/extra/generics/methods.html. Accessed 30 May 2022.