Bytecode generation in Java-TX

Fayez Abu Alia

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
abualia@hb.dhbw-stuttgart.de

Abstract. Java-TX is an extension of Java. The main new features are global type inference and real function types for lambda expressions. The type inference algorithm infers usually more than one correct typing. In the old version of Java-TX, the user must select a result to use for bytecode generation. In the future the selection should be made automatically. To achieve this, all correct typings must be used in the bytecode generation. In the official standard of Java methods with generic data types can not be overloaded. This problem should be solved in Java-TX by coding the descriptors of parameterized types into valid class names. For each parameterized type, an empty class must be generated which should has the new name and inherit from the actual class so that it can perform all functions of that class.

1 Bytecode Generation

The Java compiler does not translate Java code directly into a machine code, but into a bytecode. The generated bytecode is stored in a binary class file and then can be executed with the Java Virtual Machine (JVM).
Bytecode generation is the last component of the Java-TX project, in which the Abstract Syntax Tree (`AST`) that is generated by parser is translated into bytecode using the result of the type inference.
The following example should illustrate the compile process in Java-TX project.

Example 1. The class Example has one method that takes a parameter a. The parameter a is assigned to a string value and will be returned:

```
class Example {
        m(a) {
                a = "result";
                return a;
        }
}
```

Figure 1 shows the generated `AST` of the Java program in Example 1.

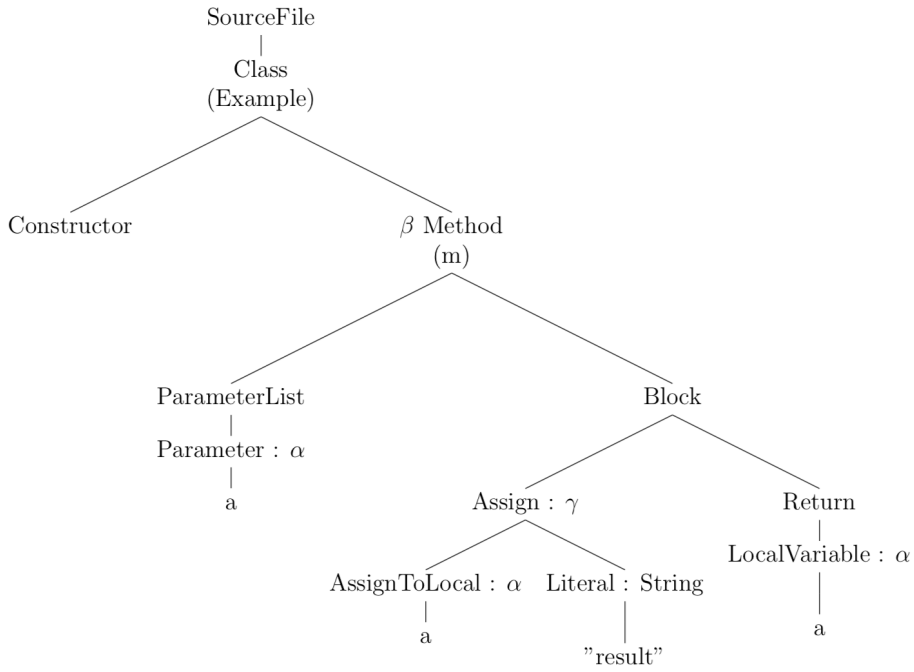

Fig. 1. The generated AST of the Java program

The type annotations are given as type placeholder (greek letters). The following typings are determined by type inference:

$$[[\alpha \mapsto \texttt{String}, \beta \mapsto \texttt{String}, \gamma \mapsto \texttt{String}]].$$

As a next step, bytecode is to be generated. In bytecode generation each node of the `AST` will be visited starting from the root node to the leaves. In each node the necessary informations will be translated into bytecode and the result obtained from the type inference will compensate for the missing types.
In the beginning of bytecode generation the root node `SourceFile` is visited. This node does not contain any necessary informations for bytecode so we do nothing in this node, then the child node `Class` is visited. It includes the access flags, the name, the superclass and the implemented interfaces of the class. We translate all these informations into bytecode. The node `Class` has two child nodes which are `Consturctor` and `Method`. The node `Constructor` is visited first. It is the default constructor which is automatically inserted in the bytecode by the compiler. In this node the following bytecode is generated:

```
public Example();
descriptor: ()V
flags: (0x0001) ACC_PUBLIC
Code:
```

```
stack=1, locals=1, args_size=1
0: aload_0
1: invokespecial #9 //Method java/lang/Object."<init>":()V
4: return
```

The default constructor is public and takes no arguments. The opcode `aload_0` pushes the `this`-reference onto the operand stack. The opcode `invokespecial` invokes the constructor of this class's superclass, in this case it is `Object`.

Afterwards, the node `Method` is visited. It contains the necessary informations for bytecode generation which are the modifiers, the name and the descriptor of the method `m`. First we generate the method descriptor using the return type of the method and the type of the parameter `a` which are calculated by type inference. The following descriptor is generated:

```
(Ljava/lang/String;)Ljava/lang/String;
```

We translate all the above informations into bytecode.

Thereafter the node `ParameterList` is visited which contains just one parameter. The position of the parameter in the array of local variables, and the type of parameter are stored for later use.

Then the node `Block` is visited. It has two child nodes which are `Assign` and `Return`. First we visit the node `Assign` in which there is no necessary informations for bytecode generation, then we visit the node `Literal`. It is a string literal, and it has the value `"result"`. This value has to be pushed onto the operand stack using the bytecode instruction `ldc`. Afterwards the node `AssignToLocal` is visited. In this node the value on the stack should be popped and then stored in position `1` (the position of parameter `a`) in the array of the local variables. This step could be done using the bytecode instruction `astore_1`.

Finally the node `return` is visited in which the parameter `a` has to be returned. For this the patemeter `a` must be pushed from the array of the local variables onto the operand stack using the bytecode instruction `aload_1`, then it has to be returned using the bytecode instruction `areturn`.

When all nodes in the `AST` have been visited and all necessary informations have been translated into bytecode, then it will be stored in a binary class file.

The following cutout of the translation of the class `Example` shows the generated bytecode of the method `m`:

```
java.lang.String m(java.lang.String);
descriptor: (Ljava/lang/String;)Ljava/lang/String;
flags: (0x0000)
Code:
stack=1, locals=2, args_size=2
0: ldc #2 // String result
2: astore_1
```

```
3: aload_1
4: areturn
```

Example 2. The translated bytecode class corresponds to

```
class Example {
    String m(String a) {
        a = "result";
        return a;
    }
}
```

2 Using all typings

Usually the type inference does not only calculate one set of results, but also a lot of results, such as class `Add` in Example 3.

Example 3. The class Add illustrates intersection types caused by overloading.

```
class Add {
        m(a) {
                return a + a;
        }
}
```

In the method `add` the plus operator is performed on the parameter `a`. As mentioned in Example 3, the plus operator can be an addition, as well as a concatenation in java. The addition can be performed on the numeric types which include `Integer`, `Double`, `Long` and `Float`, whereas the concatenation can be preformed on `String`.
According to the above, under the assumptions that the type placeholder α stands for the argument type and the type placeholder β stands for the return type of `add`, the following typings are determined by type inference:
$[[\alpha \mapsto \texttt{Integer}, \beta \mapsto \texttt{Integer}], [\alpha \mapsto \texttt{Double}, \beta \mapsto \texttt{Double}],$
$[\alpha \mapsto \texttt{Long}, \beta \mapsto \texttt{Long}], [\alpha \mapsto \texttt{Float}, \beta \mapsto \texttt{Float}],$
$[\alpha \mapsto \texttt{String}, \beta \mapsto \texttt{String}]]$
In the old version of Java-TX, the user must select a result to use for bytecode generation. However, choosing a result restricts the functionality of the class.
At present we aim to use all possible results of the type inference in order to generate a general class as possible. The user should not have to select a result for bytecode generation; the selection should be made automatically. In order to obtain the most general class, we generate a method for each typing in the bytecode so that it is overloaded. The bytecode corresponds to the following Java class:

```
class Add {
  Integer add(Integer a) { return a + a; }

  Double add(Double a) { return a + a; }

  Long add(Long a) { return a + a; }

  Float add(Float a) { return a + a; }

  String add(String a) { return a + a; }
}
```

This is a correct Java program.
Summarized the following Java-TX program is correct:

```
class Add {
    add(a) { return a + a; }
}

class Main {
  public static void main(String[] args) {
    Add a = new Add();
    a.add(1);
    a.add("xxx");
  }
}
```

3 Generics in function types

Another problem is overloading methods with generic parameters. The generic type information is not available at runtime because the generic implementation uses `type erasure`. This problem arises especially with our function types.
Let us consider the following example:

Example 4. Let the following method be given:

```
class Generics {
  public String m(Fun0<String> f) {
    ...
  }
}
```

The following is an excerpt of the translation of the class `Generics`:

```
...
public java.lang.String m(Fun0<java.lang.String>);
```

```
descriptor: (LFun0;)Ljava/lang/String;
flags: ACC_PUBLIC
⋮
Signature:#108 //(LFun0<Ljava/lang/String;>;)Ljava/lang/String;
...
```

If we take a look the method descriptor in the generated bytecode, we will see that all the informations between < and > are removed by type erasure. Therefore, the information about the parameters are not available at runtime. We want to overload the method in the above example, so we define a new method with the same name but where its parameter has a different type. Let us consider the following example:

Example 5. Let the following Java code be given:

```
class Generics {
  m(f) {
    var ret = f.apply() + f.apply();
  }
}
```

The type inference infers the types `Fun0<Integer>`, ..., `Fun0<String>` for `f`. This would lead to the following program:

```
class Generics {
  Integer m(Fun0<Integer> f) {...}
  ...

  String m(Fun0<String> f) {...}
}
```

This is not a correct Java program. The compiler will return the following error message:

```
name clash: m(Fun0<Integer>) and m(Fun0<String>) have the same
erasure
```

This error message will be returned because both methods have the same descriptor. In order to prevent this problem, the Java Runtime Environment (JRE) should be able to distinguish the descriptors of the overloaded methods with generic parameters. Let us consider the signatures of parameterized types in Example 5:

```
LFun0<Ljava/lang/Integer;>;
LFun0<Ljava/lang/String;>;
```

As we can see above, the signatures of the parameterized types are not the same, but we cannot use this syntax in the descriptor because the JVM has a checker which checks that no parameterized types are used. Our approach in resolving this problem is to change the syntax of the signature by replacing the separator characters / by $$ and the characters < and > by $$$. Subsequently that we use the new syntax in the descriptors of the overloaded methods with a parameterized type as follows:

```
LFun0$$$Ljava$$lang$$Integer$$$;
LFun0$$$Ljava$$lang$$String$$$;
```

This means that the parameterized type is changed into a class name. Each parameterized type is presented as a separate class. Therefore, they are distinguishable from one another. Since these types are used in the descriptors of the overloaded methods in Example 5, they are also distinguishable from one another. The following are the new descriptors of both methods in Example 5:

```
(LFun0$$$Ljava$$lang$$Integer$$$;)Ljava/lang/Integer;
(LFun0$$$Ljava$$lang$$String$$$;)Ljava/lang/String;
```

Now, the JRE can distinguish the descriptors of the overloaded methods with parameterized types but it does not know the types. In order for the JRE to know the types, we have to generate a new class for each type that has the new name. These generated classes are empty, they do not have any variables or methods and they should inherit from the actual class so that they can perform all functions of that class.

4 Conclusion and future work

In this paper we presented the bytecode generation, which generates a method element in the bytecode for each inferred element of each method's intersection type, such that all inferred overloadings can be used. Furthermore, we described the possibility of using generics for function types in bytecode.
we plan to consider the *type erasure* in bytecode. In similar fashion as for the function types, the parameters should be used in bytecode for all parameterized types. There are indeed a number of challenges. A first approach could to generate a corresponding class for each type instantiation.