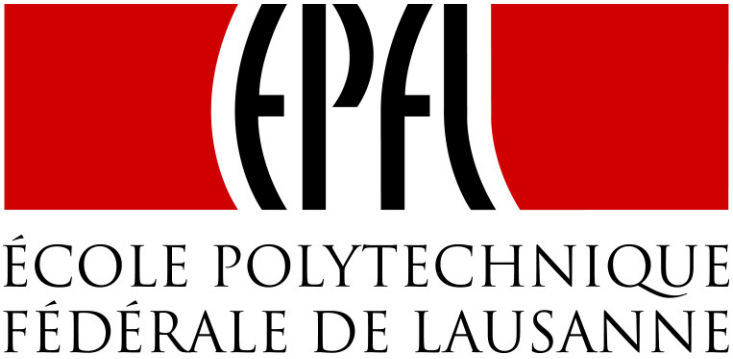
Value Class in Tool

Compiler Construction 2016 Final Report

Mikael Morales Gonzalez
mikael.moralesgonzalez@epfl.ch

Charles Parzy Turlat
charles.parzy-turlat@epfl.ch



1 Introduction

During the semester we implemented a compiler for a Java-like Language called Tool. It generates bytecode from tool program to be able to execute it.
The compiler is constituted of several steps.

1. The Lexer generates Tokens from the input files
2. The Parser which contains the LL1 grammar and checks if the program's syntax is valid. It also contains an AST constructor which creates abstract syntax trees.
3. The Name analyser in which we associate symbols to the declarations. We also check basic rules like inheritance cycles, overloadings ...
4. A Type Checker which rejects all the remaining invalid programs with type inconsistencies.
5. A Code generator which generates the bytecode contained in the created class files.

Our extension brings the concept of *value classes* to the language.
A value class is a special kind of class which contains only one field and is represented only by this field. The point of value classes is to avoid allocating runtime objects. In order to do so, instead of allocating memory to a value class, it pushes its representative on the stack.

2 Examples

Value class signatures are similar to class signatures in tool, with the keyword class preceded by **@value**, to differentiate a value class declaration from a standard class declaration.
Unlike scala, a value class does not have to extend AnyVal, we choose to forbid value classes to extend anything, and as in scala, value classes cannot be extended.
A value class can only contain strictly one field, which needs to be declared at the beginning of the value class, and then followed by method implementations. The field of a value class object cannot be modified after the instantiation of the object.

First of all, a very simple example of a value class which shows how a value class behaves with a primitive type field. The user creates an instance of the value class A, and initializes the field of the class A to 2, and calls the method sayHello defined in the value class A which returns "Hello" concatenated with the field, so in this case it returns "Hello2".

```
program Main {
    println(new A(2).sayHello());
}

@value class A {
    var x : Int;

        def sayHello(): String = {
                return "Hello" + x;
        }
}
```

The generated bytecode is the following.

```
public static void main(java.lang.String[]);
Code:
   0: getstatic #19
   3: iconst_2 // Push the constant 2
   4: invokestatic #25 // Method A.sayHello:(I)Ljava/lang/String;
   7: invokevirtual #31
  10: return


public class A {
  protected int x;

  static java.lang.String sayHello(int);
  Code:
     0: new #13
     3: dup
     4: invokespecial #17
     7: ldc #19 // String Hello
     9: invokevirtual #23
    12: iload_0
    13: invokevirtual #26
    16: invokevirtual #30
    19: areturn
}
```

We can see that the instantiation of Test(2) doesn't allocate any memory in the heap. It only results in pushing the constant 2 on the stack.
Since we don't allocate the instance of the class on the heap, we declare all

the methods of a value class as static. Moreover, we modify the method's signature to pass the field of the value class so they can access it.

The field of a value class can be of any type that is allowed in Tool (Bool, String, Int, Int[], Class and another value class). Here is an example with several value classes and a class C as a root field.

```
program Main {
        println(new A(new B(new C())).get());
}

@value class A {
        var b: B;
        def get(): Int = {
                return b.get();
        }
}

@value class B {
        var c: C;
        def get(): Int = {
                return c.get();
        }
}

class C {
        def get(): Int = {
                return 2;
        }
}
```

The generated bytecode is the following.

```
public static void main(java.lang.String[]);
Code:
   0: getstatic #19
   3: new #21 // class C
   6: dup
   7: invokespecial #22 // Method C."<init>":()V
  10: invokestatic #28 // Method A.get:(LC;)I
  13: invokevirtual #34
  16: return
```

As you can see if the field is an object (Class, String or arrays), we allocate memory only for the field and not the value class.
Furthermore, if multiple value classes are wrapped between each other, we recursively fetch the field until we reach a non value class field.

```
program Main {
        println(new A(5).foo().foo().foo().foo().compute());
}

@value class A {
        var s: Int;

        def compute(): Int = {
                return s;
        }

        def foo(): A = {
                return this;
        }
}
```

The generated bytecode is the following.

```
public static void main(java.lang.String[]);
Code:
   0: getstatic #19
   3: iconst_5 // Push the constant 5
   4: invokestatic #25 // Method A.foo:(I)I
   7: invokestatic #25 // Method A.foo:(I)I
  10: invokestatic #25 // Method A.foo:(I)I
  13: invokestatic #25 // Method A.foo:(I)I
  16: invokestatic #28 // Method A.compute:(I)I
  19: invokevirtual #34
  22: return

public class A {
  protected int s;

  static int compute(int);
    Code:
       0: iload_0
       1: ireturn

  static int foo(int);
    Code:
       0: iload_0
       1: ireturn
}
```

Here we can see that when calling the method foo which returns this, we only push the field on the stack which allows us to concatenate calls of the method foo. Each call to the method foo uses the field which was pushed by the previous call of foo() as an argument.

3 Implementation

The extension involves many minor changes in every step of the pipeline and major changes in the code generation phase.

3.1 Theoretical Background

To understand the behavior of value classes, we studied the online documentation [Harrah(2011)]. It helped us to understand how value classes work, how they are represented in memory and their limitations.

3.2 Implementation Details

We detail in this section the changes in the implementation.

3.2.1 Lexer

We defined in the lexer a new keyword, **@value**, which is used to declare a value class.

3.2.2 Parser

```
'ClassDeclaration ::= CLASS() ~ 'Identifier ~ 'OptExtends ~ 'ClassBody
  | VALUE() ~ CLASS() ~ 'Identifier ~ 'ClassBody,

'NewExpr ::= NEW() ~ 'NewSeq | 'Factor,
'NewSeq ::= INT() ~ LBRACKET() ~ 'Expression ~ RBRACKET()
  | 'Identifier ~ LPAREN() ~ 'NewClassSeq,
'NewClassSeq ::= RPAREN() | 'Expression ~ RPAREN(),
```

We upgraded the grammar to recognize a value class declaration.
A value class declaration consists of the VALUE() token, followed by CLASS() token, an identifier and the class body.
We can see here that a value class cannot extend anything.
Furthermore, we defined the instantiation of a value class A as: **new A(x)** where **x** is an expression representing the field.

To be able to construct the abstract syntax tree, we defined a new class declaration called ValueClassDecl. Both class declarations extends a trait called Class to which a symbol is attached.
Even if a value class contains only one variable and has no parent, we kept a list of variables and set the parent to None, to increase modularity.

```
sealed trait Class extends DefTree
with Symbolic[AbstractClassSymbol] {
    val id: Identifier
    val methods: List[MethodDecl]
    val parent: Option[Identifier]
    val vars: List[VarDecl]
}

case class ClassDecl(id: Identifier, parent: Option[Identifier],
vars: List[VarDecl], methods: List[MethodDecl])
extends Class

case class ValueClassDecl(id: Identifier, vars: List[VarDecl],
methods: List[MethodDecl]) extends Class {
    override val parent = None
}
```

3.2.3 Name Analysis

We defined a trait AbstractClassSymbol which is extended by ClassSymbol and ValueClassSymbol to keep a high modularity. We also added a field "fieldId" to ValueClassSymbol to get the field in the Map members more easily.
While going through the program, we set the symbols to the classes and we checked that value classes have a unique field.
Since we added the field of a value class as an argument to every method defined in this value class, we needed to make sure that there was no arguments with the same name as the field, to avoid having unexpected behaviours. So an error is thrown if there is an argument of a value class method with the same name as the field in this phase.

```
sealed trait AbstractClassSymbol extends Symbol {
    var methods = Map[String, MethodSymbol]()
    var members = Map[String, VariableSymbol]()
    var parent: Option[ClassSymbol] = None

    def lookupMethod(n: String): Option[MethodSymbol]
    def lookupVar(n: String): Option[VariableSymbol]
  }

class ClassSymbol(val name: String)
extends AbstractClassSymbol
{/*Implementation*/}

class ValueClassSymbol(val name: String,val fieldId: String)
extends AbstractClassSymbol
{
  override def getType = TValueClass(this)
  override def setType(t: Type) =
  sys.error("Cannot set the symbol of a ValueClassSymbol")

  def getField: Option[VariableSymbol] =
  members.get(fieldId)
  override def lookupMethod(n: String) =
  methods.get(n)
  override def lookupVar(n: String) =
  members.get(n)
 }
```

3.2.4 Type Checking

Firstly, we defined a new type called TValueClass which is a subtype of itself and of TValueObject, which represents the top of the value class hierarchy. After that, we upgraded the equality type checking to allow comparison between two value classes.
Finally, we also made sure that when we instantiate value classes, the expected type of the field is satisfied.

3.2.5 Code Generation

First, we removed the default constructor for value classes. Then as explained earlier, we forced value classes methods to have the field as the first argument. Since we set methods as static, they need to have access to the field.
Since conventions put the object reference at position zero in the local variables pool, we decided to put the field of the value class instead, since there is no concrete instance of a value class when generating the bytecode. It allows us to define "this" as:

```
case t@This() =>
  findRootType(t.getType) match {
    case TInt | TBoolean => ch << ILOAD_0
    case TIntArray | TString | TClass(_) =>
      ch << ALOAD_0
  }
```

where we always load what is contained in the position zero in the local variables pool.

We defined the method findRootType to access the type of the root field. If several value classes are wrapped between each other, the root field is the first non value class field. This method is used throughout the code generation phase to be able to pattern match on the types previously defined in Tool (TInt, TString, TBoolean, TIntArray, TClass).

```
def findRootType(tpe: Type): Type = {
  tpe match {
    case TValueClass(vcs) =>
      findRootType(vcs.getField.get.getType)
    case _ => tpe
  }
}
```

Assign Bellow is the modified code handling the assign statement.

```
case Assign(id, expr) =>
  mapping.get(id.value) match {
    case Some(pos) =>
      cGenExpr(expr)
      findRootType(id.getType) match {
        case TInt | TBoolean =>
          ch << IStore(pos)
        case TIntArray | TString | TClass(_) =>
          ch << AStore(pos)
        case _ =>
      }

    case None =>
      ch << ALOAD_0
      cGenExpr(expr)
      ch <<
        PutField(cname, id.value, typeToDescr(id.getType))
  }
```

For the Assign statement, we clearly see what we were talking about earlier. If you assign a value class to a variable, only the field will be really assigned, not the actual "instance" of the value class.

Equals We first compare the name of two value classes to be sure that they are "instances" of the same value class. If they are not, we return 0. Otherwise, we compare the two fields, and return the result of the comparison.

Method Call We add the type of the field of the value class as the first argument. Instead of calling InvokeVirtual, we call InvokeStatic since methods of value classes are static.

New Value Class Instead of allocating the instance of the value class, we generate bytecode only for the expression of the root field.

4 Possible Extensions

We actually have a fully working prototype. However our implementation is not production ready. We could have a scala-like syntax for value classes declarations, for example:

```
class A (val x: Int) { /* Implementation */ }
```

Furthermore, we could define another step in the pipeline before the code generation phase. In this new stage we would have modified the value classes method's signatures to add the field instead of doing it in the code generation. This would make these changes more localized and also lighten the code generation phase.

References

[Harrah(2011)] M. Harrah. Value Classes and Universal Traits, 2011. https://lc.cx/JwdM.