Improved Arguments and Fields

Computer Language Processing '15 Final Report

Pierre-Antoine Desplaces Nicolas Ritter Robin Genolet

EPFL

pierre-antoine.desplaces@epfl.ch, nicolas.ritter@epfl.ch, robin.genolet@epfl.ch

1. Introduction

In the first part of this computer language processing project we built a simple compiler for TOOL (Toy Object Oriented Language). The pipeline through which an input TOOL program goes is composed of the following components: lexer (yields tokens from chars), parser (yields Abstact Syntax Trees, AST from tokens), name analyser (rejects bad inputs due to errors such as defining a class twice or overloading a method), type checker (rejects bad inputs due to type errors), code generator (yields jvm bytecode from ASTs).

The extension we've chosen to work on will enable the use of

- default scala-like method parameters
- default values for class and method variables
- arbitrary number of arguments to a function and therefore Bool, String and Object arrays.

2. Examples

We now give code examples that show where our extension is useful.

Listing 1. This program prints true, false, true

```
object BoolArray {
    def main() : Unit = {
        println(new A().foo());
    }
}

class A {
    def foo(): Bool = {
        var ba: Bool[];
        ba = new Bool[3];
        ba[0] = true;
        ba[1] = false;
        ba[2] = ba[0];
        println(ba[0]);
        println(ba[1]);

        return ba[2];
    }
}
```

Listing 2. VarArgs: this program prints "bar"

```
object StringVarArg {
    def main() : Unit = {
        println(new A().foo("b", "a", "r"));
    }
}

class A {
    def foo(array: String*): String = {
                var i: Int = 0;
        var a: String = "";
        while (i < array.length) {
            a = a + array[i];
            i = i + 1;
        }
                return a;
    }
}
```

Listing 3. Default arguments: this program prints "m: 6 n: 7 o: bar, m: 1 n: 2 o: foo"

```
object DefaultArg {
    def main() : Unit = {
        println(new A().foo(6));
        println(new A().foo(1, 2, "foo"));
    }
}

class A {
    def foo(m: Int, n: Int = 7, o: String = "bar"): Bool = {
        println("m: " + m + " n: " + n + " o: " + o);
```

```
        return true;
    }
}
```

3. Implementation

We explain in this section how we implemented these extensions.

3.1 Bool, String and Object arrays

This step was obviously needed to support varargs.
We began by enhancing the parser to be able to parse Bool[], String[], <Identifier>[] into the new trees that we added. In general this step was straightforward, but we had to read the JVM doc describing class file format to find what value was to be given to the cafebabe $NewArray$ abstract byte code, depending on what type the array was supposed to contain.

3.2 VarArgs

The parser was modified to recognize VarArgs (denoted by a trailing star). We added a default boolean value to the $Formal\,Tree$ that we use both in the name analysis and in the code generation phases.
We chose to allow only a single VarArg per method declaration. Moreover, if used, it must be defined as the last parameter of the method. This choice simplifies greatly the implementation and follows the decision of Java and Scala.
When adding a method to the cafebabe $ClassFile$, if the method has a VarArg, we specify this argument type to be the one of an array ($IntArray$, $BoolArray$ or $ObjectArray$). On a $MethodCall$, if the method has a VarArg, we create an array containing the parameters linked to the VarArg.

3.3 Default args

For the sake of simplicity we forbid a formal to be both a vararg and have a default value. We compare the difference between the number of arguments (formals) of a method declaration to the ones of a method call to know if a default value needs to be used. If such is the case, we insert the default value at the correct position in the method call.

3.4 Default members

Variable declarations can be assigned a default value using syntax of the form $var\ name : Type = expression$. The default value is stored in the variable symbol. The code generation phase then keeps track of which variables have been initialized, and if an uninitialized variable is accessed its default value is used.

4. Possible Extensions

We did not implement named arguments nor the copy() method, so an obvious possible extension would be to implement them.

We could also allow varargs to be in any position in the argument list as long as the types make it unambiguous but this might be difficult.