First-class Functions

Compiler Construction '12 Final Report

Stanislas Signoud Michael Schneeberger
EPFL
michael.schneeberger@epfl.ch

1. Introduction

After implementing the so-called "toolc"-compiler, which compiles a very basic computer language, we now would like to extend it with *first-class functions*. This means that we allow function declartions to be assigned to variables, to be passed as parameters to functions or to be returned from functions. The most difficult part consists in correctly attaching the non-local variables, which are internally used, to the first class function (*closure*). The problem is often referred as the *funarg problem*.

2. Example

Here is an example that summarizes the challenges of implementing first-class functions:

```
1  object firstClassExampleStarter{
2    def main() : Unit = {
3      println(new firstClassExample().go());
4    }
5  }
6
7  class firstClassExample {
8    var x : Int;
9
10   def getFunc() : (Int) => Int = {
11     var y : Int;
12     var func : (Int) => Int;
13
14     y = 7;
15     func = (z: Int) => {
16       x = x-z;
17       y = y+z;
18       return x+y;
19     };
20     return func;
21   }
22
23   def go() : Int = {
24     var func1 : (Int) => Int;
25     var func2 : (Int) => Int;
26
27     x = 44;
28     func1 = getFunc();
29     func2 = getFunc();
30     println(func1(2));   // will print x+y=42+9=51
31     println(func2(6));   // will print x+y=36+13=49
32     println(func1(1));   // will print x+y=35+10=45
33     return 0;
34   }
35 }
```

Short description of the code example:
(10) function declaration that returns a first-class function
(12+28+29) variable declarations, which take functional literals as values
(15) assigning a functional literal to a variable
(16+17) new assignement of a non-local variable
(30) invokation of the first-class function func1. We notice that y, which normally would be out of scope at this point in code, can still be accessed and is initially equal to 7. X, on the other hand, is not out of scope, it is equal to 44.
(31) invocation of the first-class function func2. We notice that y hasn't change and is intially equal to 7 again, whereas x is in the beginning of func2 invocation equal to 42.
(32) second invocation of func1. We notice that y is intially equal to 9, that is the same value as the value it had when the first invokation of func1 terminated on line 30.

3. Implementation

3.1 Theoretical Background

3.1.1 Funarg Problem

”Funarg Problem refers to the difficulty in implementing first-class functions (functions as first-class objects) in implementations with stack-based memory allocation.” (Wikipedia ”Funarg Problem”, Jan 2013) To illustrate this, let us consider y from the previous example. As we mentioned already, would y nomally be out of scope at the moment the func1 is invoked (line 30). But since it is used in a first-class function, it becomes a free variable (contrary to bound varaibles) and can still be accessed even if the context of the variable is closed. In order to make this possible we introduce *closures*, which binds the functional literals to their *free variables*. One way of implementing this is descriped by the *Lambda Lifting*.

3.2 New Elements in the Syntax Tree

Before solving the Funarg problem, however, we first are going to illustrate how other elements of the compiler have been implemented. To start off with the new syntax, which had to be introduced. We give here an overview:

```
1  sealed trait TypeTree extends Tree
2    //( type.. ) => returnType
3    case class FuncType(arguments: List[TypeTree],
4      context: List[TypeTree],
5      returnType: TypeTree) extends TypeTree
6    ...
7
8  sealed trait ExprTree extends Tree with Typed
9    // ( arguments.. ) => { varDecl; statemenets; }
10   case class FuncExpr(arguments: List[VarDecl],
11     variables: List[VarDecl], statements: List[StatTree],
12     returnExpr: Option[ExprTree])
13     extends ExprTree with Symbolic[MethodSymbol]
14   // func( arguments; )
15   case class FuncCall(function: ExprTree,
16     expressions: List[ExprTree]) extends ExprTree
17   ...
```

Corresponding examples for each snytax shall here briefly presented:

1. *FuncType* is used when a variable or a function to take or return a functional literal (line 10+12+24+25 in the example)
2. *FuncExpr* is used when a functional literal is assigned to a variable or returned in a function (line 15 in the example)
3. *FuncCall* is used when the functional literal is invoked (line 30+31+32)

3.3 Parser

As it comes to the parser, there is one rather difficult case to mention, which includes to distinguish a functional literal from an expression starting with an opening bracket:

```
1 f = (a: Int, b: Int) => {} // functional literal
2 f = (a + b) * c; // expression starting with "("
```

It makes it specially difficult since the toolc parser is some kind of a LL(1) parser. This means that the parser reads only one token after the other and does not remember passed token. In the previous example, the parser would first read the characters ”f=(”. At this point he knows, that an expression has to follow. But if the expression is just an identifier (”a” in the example), he does not know yet to distinguish the two cases. Only when reading one more token, he can identify a functional literal by colon ”:”.

```
1  if (currentToken.tokenClass == OPAREN) {
2    val expr = parseExpression;
3
4    // functional literal
5    if (currentToken.info == COLON
6        && expr.isInstanceOf[Identifier]) { ... }
7    // expression starting with "("
8    else { ... }
9
10 }
```

3.4 Analyser

The question that has to be asked, when implementing functional literals, is what kind of symbol has to be created for them. One way is to convert functional literals to MethodSymbols. This includes to give them a random name like ”anonfunc”, but also to add an additional constructor to the MethodSymbol, which indicates the parent context. This parent context points to function, where the functional literal is declared:

```
1 methodSymbol = new MethodSymbol("anonfunc",
2         classSymbol, Some(parentContext));
```

The additional parameter is needed to correctly assign symbols all free variables. Let us for this case consider the following situation:

```
1 def method1() {
2   var x: Int;
3   var func1: Unit => Unit;
4
5   func1 = Unit => {
6     var func2: Unit => Unit;\neq
7
8     func2 = Unit => {
9       x=x+1
10    }
11  }
12 }
```

x is clearly a free variable for func2 (as well as for func1), since it is declared outside of the scope of func2 (and func1). In order to attach the proper symbol to each appearance of the variable x, the analyser has to check not only if the variable has been declared in the current functional literal (which is func2) or the parent function (which is func1), but all functions in the tree structure of all parent context. In the previous example, we would also have to check if x is declared in the scope of method1. In case we would not find the declaration of x in method1 (contrary to the previous example), an error is generated, since method1 has no parent context and the declaration of x has not been found in the parent context tree structure.

3.5 Type Checker

A more difficult part in the type checker is when a function literal is invoked.

```
1 case FuncCall(function, expressions) =>
2   val funType = tcExpr(function, anyFunction)
3       .asInstanceOf[TFunction]
4
5   for((expected, localExpr)
6       <- funType.inputTypes.zip(expressions)) {
7     tcExpr(localExpr, expected);
8   }
9
10  funType.outputType
```

1. First the expression before the brackets with the function's parameters is verified to be a function (line 2). If func is for example a functional literal, that is defined as Unit=>(Unit=>Int). That means it returns another functional literal. One can invoke the nested function by writing func()(). The resulting type Unit=>(Unit=>Int) is then saved into funType.
2. The types of the variables given as parameters to the function are then verified to agree with the parameter types given from funType (line 5)
3. Finally the actual return type is compared with the return type of funType (line 10)

Another important aspect is function isSubTypeOf to identify, if a Type is subtype of another type:

```
1 // regular type (inputTypes...) => outputType
2 case class TFunction(inputTypes: List[Type],
3       outputType: Type) extends Type {
4   override def isSubTypeOf(tpe: Type): Boolean =
5     tpe match {
6       case TAny => true
7       case 'anyFunction' => true
8       case TFunction(iTs, oT) =>
9         iTs.size == inputTypes.size &&
10        oT.isSubTypeOf(outputType) &&
11        !(for((original, candidate) <- inputTypes.zip(iTs))
12          yield { candidate.isSubTypeOf(original) })
13            .contains(false)
14
15      case _ => false
16    }
```

3.6 Code Generator

For the code generator, we roughly proceeded as follows:

1. (All previous steps, typechecking)
2. Tag variables and members that are used inside functions as "to be boxed"
3. Create boxed classes for every needed boxing types
4. Replace all tagged occurences by boxed objects : declarations, assignations, usages
5. Create closure class from inline function declaration content
6. Replace inline function declaration by an initialization of the newly declared closure class
7. Replace function call by a call to the "apply" method on the closure class

4. Possible Extensions

If you did not finish what you had planned, explain here what's missing.

In any case, describe how you could further extend your compiler in the direction you chose. This section should convince us that you understand the challenges of writing a good compiler for high-level programming languages.