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 already mentioned 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 Parser

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 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 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 && expr.isInstanceOf[Identifier]) { ... }
6   // expression starting with "("
7   else { ... }
8
9 }
```

If you are using theoretical concepts, explain them first in this subsection. Even if they come from the course (eg. lattices), try to explain the essential points *in your own words*. Cite any reference work you used like this [**?**]. This should convince us that you know the theory behind what you coded.

3.3 Implementation Details

Describe all non-obvious tricks you used. Tell us what you thought was hard and why. If it took you time to figure out the solution to a problem, it probably means it wasn’t easy and you should definitely describe the solution in details here. If you used what you think is a cool algorithm for some problem, tell us. Do not however spend time describing trivial things (we what a tree traversal is, for instance).

After reading this section, we should be convinced that you knew what you were doing when you wrote your extension, and that you put some extra consideration for the harder parts.

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.