

Aflevering 4

Thomas Vinther, 201303874

Jens Kristian Nielsen, 201303862

18. februar 2019

Opgave 47

```
1  // ...
2  case class Closure(params: List[FunParam], optrestype: Option[Type], body: Exp,
   venv: VarEnv, fenv: FunEnv)
3  // ...
4  def eval(e: Exp, venv: VarEnv, fenv: FunEnv): Val = e match {
5  // ...
6  case BlockExp(vals, defs, exp) =>
7    var venv1 = venv
8    var fenv1 = fenv
9    trace("Calculating variable values and adding to variable environment")
10   for (d <- vals) {
11     val dexp = eval(d.exp, venv1, fenv)
12     checkValueType(dexp, d.opttype, d)
13     venv1 += (d.x -> dexp)
14   }
15   for (d <- defs) {
16     fenv1 += (d.fun -> Closure(d.params, d.optrestype, d.body, venv1, fenv1))
17   }
18   eval(exp, venv1, fenv1)
19 // ...
20 case CallExp(fun, args) =>
21   val close = fenv(fun)
22   if(args.length == close.params.length) {
23     for(i <- 0 to args.length-1) {
24       checkValueType(eval(args(i), venv, fenv), close.params(i).opttype, CallExp(fun,
25         args))
26     }
27     def fpGetVar(fp: FunParam): Var = fp.x
28     val venv_update = close.params.map(fpGetVar).zip(args.map(exp => eval(exp,
29       venv, fenv)))
30     val res = eval(fenv(fun).body, close.venv++venv_update, close.fenv+(fun->close)
31       )
32     checkValueType(res, close.optrestype, CallExp(fun, args))
33     return res
34   }
35   else throw new InterpreterError(fun+" failed due to mismatch of number of
36     arguments", CallExp(fun, args))
37 }
```

d)

d. Explain briefly how the implementation of the cases for DefDecl and CallExp in Interpreter.scala relate to the corresponding rules in the operational semantics (see in particular slides 23, 25, and 41).

DefDecl From the operational semantic we get:

$$\frac{\phi' = \phi[f \mapsto (x, t_1, e, t_2, \rho, \phi)]}{\rho, \phi \vdash \text{def } f(x : t_1) : t_2 = e \Rightarrow \rho, \phi'}$$

Which we implement in `Interpreter.scala` in our `BlockExp` match case, in line 7-8 and 15-16. In line 7-8, we retrieve our "old" variable environment ρ and function environment ϕ respectively, as seen in the denominator of the semantic. After `ValDecl` (line 10-14), in line 15, we iterate through each declaration of a function given in `BlockExp`, we then update the function environment in line 16. This is done according to the numerator of our semantic, where ϕ' is equal to `fenv1`, which is updated/created by mapping each function from the declaration to the closure of the function. The closure of the function contains a parameter list, which in turn optionally contains the types of the parameters. (x, t_1) in our semantic. The call is then made to `d.optrestype`, the optional type t_2 given to the closure. Then the body of the function e is called, and finally the old variable environment and function environment. Which concludes the numerator of the operational semantic and also the denominator as we have now updated the function environment, without updating the variable environment used in `BlockExp`.

CallExp: from the operational semantics we get

$$\frac{\phi(f) = (x, t_1, e_2, t_2, \rho_2, \phi_2) \quad \rho, \phi \vdash e_1 \Rightarrow v_1 \quad \rho_2[x \mapsto v_1], \phi_2[f \mapsto (x, t_1, e_2, t_2, \rho_2, \phi_2)] \vdash e_2 \Rightarrow v_2 \quad v_1 = t_1 \quad v_2 = t_2}{\rho, \phi \vdash f(e_1) \Rightarrow v_2}$$

In line 21 we get the closure of the function we wish to use. In lines 22 and 32 we check that the function call has the correct number of arguments, and if it does not we throw an `InterpreterError`. Having performed this check we are sure that the for loop in line 23-25 does not get an index out of bounds exception, and we can perform the checks " $v_1 = t_1$ ", i.e. that each argument has the type that the function expects, if it indeed expects a type. Here " v_1 " comes from `eval(args(i), venv(ρ), fenv(ϕ))` and it should have type `close.params(i).opttype`, which is the `Option[type]` that the i 'th argument expects, the last part of the `checkValueType` call is just for the error message. Next we use a small help function `fpGetVar` that returns the name of the parameter, there is probably a smarter way to do this. In line 27 we define the update we wish to make to the variable environment, i.e. the $[x \mapsto v_1]$ part of the semantics, simply by using our closure of the function, the small help function to get the variables, and by evaluating e_1 according to the old variable and function environment and zipping these together. In line 28 we evaluate the body of the function using the newly updated variable environment and function environment from the closure of f i.e. in $\rho_2[x \mapsto v_1], \phi_2[f \mapsto (x, t_1, e_2, t_2, \rho_2, \phi_2)]$ and thereby setting the immutable variable `res` equal to v_2 in our semantic. In line 29, we perform the return type check using the `checkValueType` method once more.

Opgave 48

```

1  def typeCheck(e: Exp, vtenv: VarTypeEnv, ftenv: FunTypeEnv): Type = e match {
2    // ...
3    case BlockExp(vals, defs, exp) =>
4      var (vtenv1, ftenv1) = (vtenv, ftenv)
5      for (d <- vals) {
6        val t = typeCheck(d.exp, vtenv1, ftenv1)
7        checkTypesEqual(t, d.opttype, d)
8        vtenv1 += (d.x -> d.opttype.getOrElse(t))
9      }
10     for (d <- defs)
11       ftenv1 += (d.fun -> getFunType(d))
12     // ftenv1 ++ defs.map(d => d.fun).zip(defs.map(getFunType)), how do we make
13     // this work?
14     for (d <- defs) {
15       val funType = getFunType(d)
16       val vtenv_update = d.params.map(fp => fp.x).zip(funType._1)
17       if (d.optrestype.isDefined) {
18         checkTypesEqual(typeCheck(d.body, vtenv ++ vtenv_update, ftenv1), d.
19           optrestype, d)

```

```

18     }
19 }
20 typeCheck(exp, vtenv1, ftenv1)
21 // ...
22 case CallExp(fun, args) =>
23     val paramz = ftenv(fun)
24     if (paramz._1.length == args.length) {
25         for (i <- paramz._1.indices) {
26             if (paramz._1(i) != typeCheck(args(i), vtenv, ftenv)) {
27                 throw new TypeError(s"Argument nr $i has the wrong type", CallExp(fun,
28                     args))
29             }
30         }
31     } else throw new TypeError("Wrong number of arguments for the function "+fun,
32         CallExp(fun, args))
33 }

```

Opgave 49

```

1  def main(args: Array[String]): Unit = {
2      test("{ def f(x: Int): Int = x; f(2) }", IntVal(2), IntType())
3      testVal("{ val x = 1; { def q(a) = x + a; { val x = 2; q(3) } } }", IntVal(4))
4      testFail("{ def f(x: Int): Int = x; f(2, 3) }")
5      test("{ "+
6          "def isEven(n: Int): Boolean = " +
7          "if (n == 0) true " +
8          "else isOdd(n - 1) ; " +
9          "def isOdd(n: Int): Boolean = " +
10         "if (n == 0) false " +
11         "else isEven(n - 1) ; " +
12         "isEven(4) } ", BoolVal(true), BoolType())
13     test("{ def f(x: Int): Int = 2*x; { def g(x: Int, y: Int): Int = if(x<y) 3*f(x)
14         else 4*f(y); g(5,6) } }", IntVal(30), IntType())
15     testFail("{ def f(x: Int): Int = 2*x; { def g(x: Int, y: Int): Int = if(x<y) 3*
16         f(x) else 4*f(y); g(3,true) } }")
17     testFail("{ def f(x: Int): Int = 2*x; { def g(x: Int, y: Int): Int = if(x<y) 3*
18         f(x) else 4*f(y); g(3,4,5) } }")
19     testVal("{ def f(x) = if(0<g(x)) g(x) else 0 ; def g(x) = 5 ; f(3) }", IntVal(5))
20     testType("{ def f(x: Int): Int = g(x) ; def g(x: Int): Int = f(x) ; f(3) }",
21         IntType())
22     test("{ def f(x: Int): Int = x; f(2) }", IntVal(2), IntType())
23     test("{ def get(x: Int): Int = x; get(2) }", IntVal(2), IntType())
24     test("{ def f(x: Int) : Int = x; if(true) f(5) else f(3) }", IntVal(5), IntType())
25     test("{ def dyt(x: Int): Int = x*2; dyt(21) }", IntVal(42), IntType())
26     test("{ def fac(n: Int) : Int = if (n == 0) 1 else n * fac(n - 1); fac(2) } ",
27         IntVal(2), IntType())
28     test("{ def f(y: Int): Boolean = (y == y); f(2) }", BoolVal(true), BoolType())
29     testFail("{ def f(x: Int): Int = x; f(2, 3) }")
30     testFail("{ def f(y: Int): Int = (y == y); f(2) }")
31     testFail("{ def fac(n: Int): Boolean = if (n == 0) 1 else n * fac(n - 1); fac
32         (2) } ")
33     testFail("{ def f(x: Float): Int = x; f(2f) }")
34 }
35 //We didnt change the test methods
36 def test(prg: String, rval: Val, rtype: Type) = {
37     testVal(prg, rval)
38     testType(prg, rtype)
39 }

```

```

35 def testFail(prg: String) = {
36   testValFail(prg)
37   testTypeFail(prg)
38 }
39
40 def testVal(prg: String, value: Val, venv: VarEnv = Map[Var, Val](), fenv: FunEnv
    = Map[Var, Closure]()) = {
41   assert(eval(parse(prg), venv, fenv) == value)
42 }
43
44 def testType(prg: String, out: Type, venv: VarTypeEnv = Map[Var, Type](), fenv:
    FunTypeEnv = Map[Var, (List[Type], Type)]()) = {
45   assert(typeCheck(parse(prg), venv, fenv) == out)
46 }
47
48 def testValFail(prg: String) = {
49   try {
50     eval(parse(prg), Map[Var, Val](), Map[Var, Closure]())
51     assert(false)
52   } catch {
53     case _: InterpreterError => assert(true)
54   }
55 }
56
57 def testTypeFail(prg: String) = {
58   try {
59     typeCheck(parse(prg), Map[Var, Type](), Map[Var, (List[Type], Type)]())
60     assert(false)
61   } catch {
62     case _: TypeError => assert(true)
63   }
64 } .

```
