Aflevering 4

Thomas Vinther, 201303874 Jens Kristian Nielsen, 201303862

18. februar 2019

Opgave 47

```
// ...
case class Closure(params: List[FunParam], optrestype: Option[Type], body: Exp,
    venv: VarEnv, fenv: FunEnv)
// ...
def eval(e: Exp, venv: VarEnv, fenv: FunEnv): Val = e match {
// ...
case BlockExp(vals, defs, exp) =>
  var venv1 = venv
  var fenv1 = fenv
  trace("Calculating variable values and adding to variable environment")
  for (d <- vals) {
    val dexp = eval(d.exp,venv1,fenv)
    checkValueType(dexp, d.opttype, d)
    venv1 += (d.x -> dexp)\\
  }
  for (d <- defs){
    fenv1 += (d.fun -> Closure(d.params,d.optrestype,d.body,venv1,fenv1))
  }
  eval(exp, venv1, fenv1)
// ...
case CallExp(fun, args) =>
  val close = fenv(fun)
  if(args.length == close.params.length){
    for(i <- 0 to args.length-1){
      checkValueType(eval(args(i),venv,fenv),close.params(i).opttype,CallExp(fun,
          args))
    }
    def fpGetVar(fp: FunParam): Var = fp.x
    val venv_update = close.params.map(fpGetVar).zip(args.map(exp => eval(exp,
        venv,fenv)))
    val res = eval(fenv(fun).body,close.venv++venv_update,close.fenv+(fun->close)
        )
    checkValueType(res,close.optrestype,CallExp(fun, args))
    return res
  }
  else throw new InterpreterError(fun+" failed due to mismatch of number of
      arguments",CallExp(fun, args))
}
// ...
```

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 operaitonel 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"variabel enviroment ρ and function enviroment 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 enviroment 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 funtion 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 variabel enviorment and function enviroment. Which concludes the numerator of the operationel semantic and also the denominator as we have now updated the function enviroment, without updating the variabel enviroment 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 paramater, there is probably a smarter way to do this. In line 27 we define the update we wish to make to the variable enviroment, 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 $e1$ according to the old variable and function enviroment and zipping these together. In line 28 we evluate the body of the funtion using the newly updated variabel enviorment and funtion enviroment 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
       this work?
13   for (d <- defs) {
14     val funType = getFunType(d)
15     val vtenv_update = d.params.map(fp => fp.x).zip(funType._1)
16     if(d.optrestype.isDefined) {
17       checkTypesEqual(typeCheck(d.body, vtenv ++ vtenv_update, ftenv1), d.
             optrestype, d)
```

```
      }
    }
    typeCheck(exp,vtenv1,ftenv1)
  // ...
  case CallExp(fun, args) =>
    val paramz = ftenv(fun)
    if(paramz._1.length == args.length){
      for(i <- paramz._1.indices){
        if(paramz._1(i) != typeCheck(args(i),vtenv,ftenv)){
          throw new TypeError(s"Argument nr $i has the wrong type",CallExp(fun,
              args))
        }
      }
      paramz._2
    } else throw new TypeError("Wrong number of arguments for the function "+fun,
      CallExp(fun, args))
  }
```

Opgave 49

```
  def main(args: Array[String]): Unit = {
    test("{ def f(x: Int): Int = x; f(2) }", IntVal(2),IntType())
    testVal("{ val x = 1;{ def q(a) = x + a;{ val x = 2;q(3)}}}",IntVal(4))
    testFail("{ def f(x: Int): Int = x; f(2, 3) }")
    test("{ "+
      "def isEven(n: Int): Boolean = " +
        "if (n == 0) true " +
        "else isOdd(n - 1) ; " +
      "def isOdd(n: Int): Boolean = " +
        "if (n == 0) false " +
        "else isEven(n - 1) ; " +
      "isEven(4) } ",BoolVal(true),BoolType())
    test("{ def f(x: Int): Int = 2*x; { def g(x: Int, y: Int): Int = if(x<y) 3*f(x)
         else 4*f(y); g(5,6) }}",IntVal(30),IntType())
    testFail("{ def f(x: Int): Int = 2*x; { def g(x: Int, y: Int): Int = if(x<y) 3*
        f(x) else 4*f(y); g(3,true) }}")
    testFail("{ def f(x: Int): Int = 2*x; { def g(x: Int, y: Int): Int = if(x<y) 3*
        f(x) else 4*f(y); g(3,4,5) }}")
    testVal("{ def f(x) = if(0<g(x)) g(x) else 0 ; def g(x) = 5 ; f(3) }",IntVal(5)
        )
    testType("{ def f(x: Int): Int = g(x) ; def g(x: Int): Int = f(x) ; f(3) }",
        IntType())
    test("{ def f(x: Int): Int = x; f(2) }", IntVal(2), IntType())
    test("{ def get(x: Int): Int = x; get(2) }", IntVal(2), IntType())
    test("{ def f(x: Int) : Int = x; if(true) f(5) else f(3)}",IntVal(5),IntType())
    test("{ def dyt(x: Int): Int = x*2; dyt(21)}",IntVal(42),IntType())
    test("{ def fac(n: Int) : Int = if (n == 0) 1 else n * fac(n - 1); fac(2)} ",
        IntVal(2),IntType())
    test("{ def f(y: Int): Boolean = (y == y); f(2)}",BoolVal(true),BoolType())
    testFail("{ def f(x: Int): Int = x; f(2, 3) }")
    testFail("{ def f(y: Int): Int = (y == y); f(2)}")
    testFail("{ def fac(n: Int): Boolean = if (n == 0) 1 else n * fac(n - 1); fac
        (2)} ")
    testFail("{ def f(x: Float): Int = x; f(2f) }")
  }
  //We didnt change the test methods
  def test(prg: String, rval: Val, rtype: Type) = {
  testVal(prg, rval)
  testType(prg, rtype)
  }

```

```
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  } .
```