Aflevering 6

Jens Kristian R. Nielsen, 201303862   Thomas D. Vinther , 201303874

11. marts 2019

Opgave 68

```
  sealed abstract class Val
  case class ClosureVal(params: List[FunParam], optrestype: Option[Type], body: Exp, env: Env, defs: List[DefDecl]) extends Val
  // ...
  case BlockExp(vals, defs, exp) =>
    var env1 = env
    trace("Calculating variable values and adding to variable environment")
    for (d <- vals) {
      val dexp = eval(d.exp,env1)
      checkValueType(dexp, d.opttype, d)
      env1 += (d.x -> dexp)
    }
    for (d <- defs){
      env1 += (d.fun -> ClosureVal(d.params,d.optrestype,d.body,env1,defs))
    }
    eval(exp, env1)
  // ...
  case LambdaExp(params, body) =>
    ClosureVal(params,None,body,env,List[DefDecl]())
  case CallExp(funexp, args) =>
    eval(funexp,env) match{
      case ClosureVal(params, optrestype, body, cenv, defs) =>
        if(args.length == params.length) {
          def halp(fp: FunParam): Id = fp.x
          var cenv_updated = cenv
          for(i <- args.indices){
            val argval = eval(args(i),env)
            checkValueType(argval,params(i).opttype,CallExp(funexp, args))
            cenv_updated += (halp(params(i)) -> argval)
          }
          for(d <- defs){ //rebind function defs, to achieve mutual recursion
            cenv_updated += (d.fun -> ClosureVal(d.params,d.optrestype,d.body,cenv,defs))
          }
          val res = eval(body,cenv_updated)
          checkValueType(res,optrestype,CallExp(funexp, args))
          res
        } else throw new InterpreterError("Wrong number of arguments",CallExp(funexp, args))
      case _ =>
        throw new InterpreterError("Not a function",funexp)
    }
}
```

Opgave 69

```
  def typeCheck(e: Exp, tenv: TypeEnv): Type = e match {
    // ...
    case BlockExp(vals, defs, exp) =>
      var tenv_updated = tenv
      for (d <- vals) {
        val t = typeCheck(d.exp, tenv_updated)
        tenv_updated += (d.x -> d.opttype.getOrElse(throw new TypeError("No type
            annotation",BlockExp(vals, defs, exp))))
        checkTypesEqual(t, d.opttype, d)
      }
      for (d <- defs){
        tenv_updated += (d.fun -> getFunType(d))
      }
      for (d <- defs){
        var tenvy = tenv_updated
        for (p <- d.params){
          tenvy += (p.x -> p.opttype.getOrElse(throw new TypeError("",p)))
        }
        checkTypesEqual(typeCheck(d.body,tenvy),d.optrestype,BlockExp(vals, defs,
            exp))
      }
      typeCheck(exp,tenv_updated)
    // ...
    case LambdaExp(params, body) =>
      val Jeppe = params.map(p => (p.x -> p.opttype.getOrElse(
        throw new TypeError("Missing type annotation",LambdaExp(params, body)))))
      FunType(Jeppe.unzip._2,typeCheck(body,tenv ++ Jeppe))
    case CallExp(funexp, args) => typeCheck(funexp,tenv) match{
      case FunType(params,restype) =>
        if(args.length == params.length){
          for(i<- args.indices){
            if(typeCheck(args(i),tenv) != params(i)){
              throw new TypeError("Fool of a Took",CallExp(funexp, args))
            }
          }
          return restype
        } else throw new TypeError("Wrong number of arguments",CallExp(funexp, args
          ))
      case _ => throw new TypeError("Not a function",funexp)
    }
```

```
object Test68 {
  def main(args: Array[String]): Unit = {
    testVal("{ def f(x) = x; f(2)}", IntVal(2))
    testTypeFail("{ def f(x) = x; f(2)}")
    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) }")
    val tests8a = "{val x: Int = 3; def use(f:((Int,Int)=>Int), y:Int): Int = f(x,
        y); def add(a: Int, b: Int): Int = a + b; def mult(a:Int, b:Int):Int = a * b
        ; use(add, 7) - use(mult, 13)}"
    val tests8b = "{def choose(c: Boolean):(Int,Int)=>Int = if (c) add else mult;
        def add(a: Int, b: Int): Int = a + b; def mult(a: Int, b: Int): Int = a * b
        ;{ val foo:((Int,Int)=>Int) = choose(true); foo(1, 2) - choose(false)(7, 13)
        }}"
    test(tests8a,IntVal(-29),IntType())
    test(tests8b,IntVal(-88),IntType())
    val tests9 = "{val x: Int = 1;val g:(Int => Int) = {val x: Int = 2;def f(a: Int
        ): Int = a+x;f};{val x: Int = 3;g(4)}}"
    test(tests9,IntVal(6),IntType())
    val tests29a = "{val inc: Int => Int = (x: Int) => x + 1;inc(3)}"
    val tests29b = "{val inc: Int => Int = (x: Int) => x + 1;def twice(f: Int =>
        Int, x: Int): Int = f(f(x));twice(inc, 3)}"
    val tests29c = "{val add: Int => (Int => Int) = (x: Int) => (y: Int) => x + y;
        val inc: (Int => Int) = add(1);add(1)(2) + inc(3)}"
    test(tests29a,IntVal(4),IntType())
    test(tests29b,IntVal(5),IntType())
    test(tests29c,IntVal(7),IntType())
    curryTest("def f(x,y)=x+y;curry(f)(2)(3)")
    curryTest("def hej(x) = 3*x; def med(y) = y+2; def dig(x,y) = hej(x)+med(y);
        curry(dig)(1)(2)")
    test("{ def te(u: Int => Int): Int = u(u(4)); te((u: Int) => u % 4) }",
    IntVal(0),IntType())
    testFail("{ def te(u: Boolean => Int): Int = u(u(4)); te((u: Int) => u % 4) }")
    testVal("{ def isEven(x) = if (x == 0) true else isOdd(x-1);" +
      " def isOdd(x) = if (x == 0) false else isEven(x-1);isEven(2) }",BoolVal(true
          ))
    curryTest("def f(x,y)=x+y;curry(f)(2)(3)")
    curryTest("def hej(x) = 3*x; def med(y) = y+2; def dig(x,y) = hej(x)+med(y);
        curry(dig)(1)(2)")
  }
  def curryTest(prg: String) = {
    val currytest = "{def curry(f) = (x) => (y) => f(x,y);def uncurry(f) = (x,y) =>
         f(x)(y);"+prg+"}"
    testingVal(currytest)
  }
  def testingVal(prg: String) = {
    testVal(prg,eval(parse(prg),Map[Id, Val]()))
  }
  def testingType(prg: String) = {
    testType(prg,typeCheck(parse(prg),Map[Id, Type]()))
  }
  // ...
}
```