Aflevering 11

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

22. april 2019

Opgave 111

```
object TypeChecker {
  // ...
  def typeCheck(e: Exp, tenv: TypeEnv, ctenv: ClassTypeEnv): Type = e match {
    // ...
    case BlockExp(vals, vars, defs, classes, exps) =>
    var tenv_updated = tenv //ValDecl
      for (d <- vals) {
        val t = typeCheck(d.exp, tenv_updated,ctenv)
        checkTypesEqual(t, getType(d.opttype,ctenv,d), d)
        tenv_updated += (d.x -> d.opttype.getOrElse(t))
      }
      //VarsDecl
      for(d <- vars){
        val (v, sto2) = eval(d.exp, env1, cenv, sto1)
        val loc = nextLoc(sto2)
        val ot =getType(d.opttype, cenv)
        env1 = env1 + (d.x -> RefVal(loc, ot))
        sto1 = sto2 + (loc -> v)
      }
      //DefDecl with mututal recursion, via lecture 6, slide 36
      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("Some error",
              BlockExp(vals,vars, defs,classes, exps)))) //tau_1 = type(t_1)
              paramtype
        }
                        //Theta' [x ->tau-1] |-e:tau_2
        checkTypesEqual(typeCheck(d.body, tenvy,ctenv), d.optrestype, BlockExp(vals
            ,vars, defs,classes, exps))  //tau_2 = type(t_2) resttype
      }
      //ClassDecl
      var ctenvUpdated = ctenv
      for(cd <- classes) {
        val tau = cd.params.map(a => getType(a.opttype,ctenvUpdated,BlockExp(vals,
            vars, defs, classes, exps)))
        ctenvUpdated += (cd.klass -> getConstructorType(cd,ctenvUpdated,classes))
        var typeenvUpdated = tenv
        for (x <- cd.params) {
          typeenvUpdated += (x.x -> getType(x.opttype.getOrElse(throw new TypeError
              ("",BlockExp(vals, vars, defs, classes, exps))), ctenvUpdated))
        }
            //x.opttype.getOrElse(throw new TypeError("Missing paramater type
                annotation",BlockExp(vals, vars, defs, classes, exps))))
        typeCheck(cd.body,typeenvUpdated,ctenvUpdated)
      }
```

```
44
45        //Block2 and block empty
46        var res : Type = unitType
47        for (exp <- exps){
48          res = typeCheck(exp,tenv_updated,ctenvUpdated)
49        }
50        res
51      // ...
52      case NewObjExp(klass, args) =>
53        val constructor = ctenv.getOrElse(klass,throw new TypeError("Class not foiund
              ",NewObjExp(klass, args)))
54        val paramtypes = constructor.params.map(b => b.opttype)
55        if(!(args.length == constructor.params.length)){throw new TypeError("Wrong
              number of arguments",NewObjExp(klass,args))}
56        for (i <- args.indices){
57          checkTypesEqual(typeCheck(args(i),tenv,ctenv),paramtypes(i),NewObjExp(klass
                , args))
58        }
59        constructor
60      case LookupExp(objexp, member) =>
61        typeCheck(objexp,tenv,ctenv) match {
62          case ConstructorType(k,tau1,m) => getType(m.getOrElse(member,throw new
                TypeError("Not a function on this object",LookupExp(objexp, member))),
                ctenv)
63          case _ => throw new TypeError("Not an object",objexp)
64        }
65    }
66    // ...
67  }
```

Opgave 112

```
1  object Interpreter {
2
3    // ...
4
5    def eval(e: Exp, env: Env, cenv: CEnv, sto: Sto): (Val, Sto) = e match {
6      // ...
7      case NullLit() =>
8         (RefVal(-1,None),sto)
9      // ...
10     case LookupExp(objexp, member) =>
11       trace("Evaluating base value")
12       val (objval, sto1) = eval(objexp, env, cenv, sto) //rho, kappa, sigma eval e
             - 1, sigma'
13       objval match {
14         case RefVal(loc, _) =>
15           trace("Location found, looking in storage")
16           if(loc == -1) {throw new InterpreterError("Null pointer exception",
                 LookupExp(objexp, member))}
17           sto1(loc) match {
18             case ObjectVal(members) =>
19               trace("Looking for matching class method")
20               (getValue(members.getOrElse(member, throw new InterpreterError(s"No
                     such member: $member", e)), sto1), sto1)
21             case v => throw new InterpreterError(s"Base value of lookup is not a
                   reference to an object: ${valueToString(v)}", e)
22           }
23         case _ => throw new InterpreterError(s"Base value of lookup is not a
               location: ${valueToString(objval)}", e)
24       }
```

```
  }
  // ...

  def getValue(v: Val, sto: Sto): Val = v match {
    case RefVal(loc, _) =>
      trace("Reference value found, looking in storage")
      if(loc == -1) {return v}
      sto(loc) match {
        case _: ObjectVal => //ObjectVal type
          trace("Object found, returning reference")
          v
        case stoval =>
          trace("Value found, returning value")
          stoval
      }
    case _ =>
      trace("Not a reference, no lookup in storage needed, returning value")
      v
  }

  // ...
  def checkValueType(v: Val, ot: Option[Type], n: AstNode): Unit = ot match {
    case Some(t) =>
      (v, t) match {
        case (IntVal(_), IntType()) |
             (BoolVal(_), BoolType()) |
             (FloatVal(_), FloatType()) |
             (IntVal(_), FloatType()) |
             (StringVal(_), StringType()) => // do nothing
        case (TupleVal(vs), TupleType(ts)) if vs.length == ts.length =>
          for ((vi, ti) <- vs.zip(ts))
            checkValueType(vi, Some(ti), n)
        case (ClosureVal(cparams, optcrestype, _, _, cenv,_), FunType(paramtypes,
            restype)) if cparams.length == paramtypes.length =>
          for ((p, t) <- cparams.zip(paramtypes))
            checkTypesEqual(t, getType(p.opttype, cenv), n)
          checkTypesEqual(restype, getType(optcrestype, cenv), n)
        case (RefVal(loc, Some(vd: ClassDeclType)), td: ClassDeclType) =>
          if(loc == -1){return }
          if(vd != td)
            throw new InterpreterError(s"Type mismatch: object of type ${unparse(vd
                )} does not match type ${unparse(td)}", n)
        case (RefVal(loc,_),_) =>
          if(loc == -1){return }
          println(loc)
          throw new InterpreterError("How did you end up here?",n)
        case _ =>
          throw new InterpreterError(s"Type mismatch: value ${valueToString(v)}
              does not match type ${unparse(t)}", n)
      }
    case None => // do nothing
  }

  // ...

  def valueToString(v: Val): String = v match {
    case IntVal(c) => c.toString
    case FloatVal(c) => c.toString
    case BoolVal(c) => c.toString
    case StringVal(c) => c
    case TupleVal(vs) => vs.map(v => valueToString(v)).mkString("(", ",", ")")
    case ClosureVal(params, _, exp, _, _,_) => // the resulting string ignores the
        result type annotation, the declaration environment, and the set of classes
```

```
      s"<(${params.map(p => unparse(p)).mkString(",")}), ${unparse(exp)}>"
    case RefVal(loc, _) =>
      if(loc== -1){return "null"}
      s"$loc" // the resulting string ignores the type annotation
    case ObjectVal(_) => "object" // (unreachable case)
  }

  // ...
}
```

Opgave 113

```
object TypeChecker {
  // ...
  def typeCheck(e: Exp, tenv: TypeEnv, ctenv: ClassTypeEnv): Type = e match {
    // ...
    case NullLit() => NullType()
    // ...
    case BinOpExp(leftexp, op, rightexp) =>
      // ...
      op match {
       // ...
        case EqualBinOp =>
          BoolType()
        // ...
      }
    case BlockExp(vals, vars, defs, classes, exps) =>
      var tenv_updated = tenv //ValDecl
      for (d <- vals) {
        val t = typeCheck(d.exp, tenv_updated,ctenv)
        checkSubtype(t, getType(d.opttype,ctenv,d), d)
        tenv_updated += (d.x -> d.opttype.getOrElse(t))
      }
      //VarsDecl
      for(d <- vars){
        val dType = typeCheck(d.exp,tenv_updated,ctenv)     //theta e : tau
        val ot = getType(d.opttype,ctenv,d)
        checkSubtype(dType,ot,d)
        tenv_updated += (d.x->RefType(dType))          //theta' x -> ref(tau)
      }
      //DefDecl with mututal recursion, via lecture 6, slide 36
      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("Some error",
              BlockExp(vals,vars, defs,classes, exps)))) //tau_1 = type(t_1)
              paramtype
        }
                        //Theta' [x ->tau-1] |-e:tau_2
        checkSubtype(typeCheck(d.body, tenvy,ctenv), d.optrestype, BlockExp(vals,
            vars, defs,classes, exps))  //tau_2 = type(t_2) resttype
      }
      //ClassDecl
      var ctenvUpdated = ctenv
      for(cd <- classes) {
        val tau = cd.params.map(a => getType(a.opttype,ctenvUpdated,BlockExp(vals,
            vars, defs, classes, exps)))
        ctenvUpdated += (cd.klass -> getConstructorType(cd,ctenvUpdated,classes))
```

```
40          var typeenvUpdated = tenv
47          for (x <- cd.params) {
48            typeenvUpdated += (x.x -> getType(x.opttype.getOrElse(throw new TypeError
                ("",BlockExp(vals, vars, defs, classes, exps))), ctenvUpdated))
49          }
50              //x.opttype.getOrElse(throw new TypeError("Missing paramater type
                  annotation",BlockExp(vals, vars, defs, classes, exps))))
51          typeCheck(cd.body,typeenvUpdated,ctenvUpdated)
52        }
53
54        //Block2 and block empty
55        var res : Type = unitType
56        for (exp <- exps){
57          res = typeCheck(exp,tenv_updated,ctenvUpdated)
58        }
59        res
60        // ...
61      // ...
62      case CallExp(funexp, args) => typeCheck(funexp, tenv,ctenv) match {
63        case FunType(params, restype) =>
64          if (args.length == params.length) {
65            for (i <- args.indices) {
66              if (typeCheck(args(i), tenv,ctenv) != params(i))
67                throw new TypeError("Fool of a took", CallExp(funexp, args))
68            }
69            return restype
70          }
71          else throw new TypeError("Wrong number of arguments", CallExp(funexp, args)
                )
72        case _ => throw new TypeError("Not a function", funexp)
73      }
74      case AssignmentExp(x, exp) =>
75        tenv(x) match{
76          case RefType(a) => checkSubtype(typeCheck(exp,tenv,ctenv),Some(a),e)
77            return unitType
78          case _ => throw new TypeError("Not a var",e)
79        }
80        return unitType
81      case NewObjExp(klass, args) =>
82        val constructor = ctenv.getOrElse(klass,throw new TypeError("Class not foiund
              ",NewObjExp(klass, args)))
83        val paramtypes = constructor.params.map(b => b.opttype)
84        if(!(args.length == constructor.params.length)){throw new TypeError("Wrong
              number of arguments",NewObjExp(klass,args))}
85        for (i <- args.indices){
86          checkSubtype(typeCheck(args(i),tenv,ctenv),paramtypes(i),NewObjExp(klass,
                args))
87        }
88        constructor
89      // ...
90    }
91
92    // ...
93
94    def subtype(t1: Type, t2: Type): Boolean =
95      if(t1.equals(t2)) true //reflexivity
96      else (t1,t2) match  {
97      case (IntType(),FloatType()) => true
98      case (NullType(),ConstructorType(_,_,_)) => true
99      case (FunType(left1,right1),FunType(left2,right2)) =>
100       if (left1.length == left2.length ){
101         left1.zip(left2).foldRight[Boolean](true)((a,b) => subtype(a._2,a._1)&&b)
              &&
```

```
102         subtype(right1,right2)
103       } else false
104     case (TupleType(left1),TupleType(left2)) =>
105       if (left1.length == left2.length ){
106         left1.zip(left2).foldRight[Boolean](true)((a,b) => subtype(a._1,a._2)&&b)
107       } else false
108     case _ => false
109 }
```

Opgave 117

Delspørgsmål a

$$\textsc{AndAndTrue}\ \frac{\rho,\kappa,\sigma \vdash e_1 \Rightarrow \text{true},\sigma' \qquad \rho,\kappa,\sigma' \vdash e_2 \Rightarrow \text{true},\sigma''}{\rho,\kappa,\sigma \vdash e_1\ \texttt{\&\&}\ e_2 \Rightarrow \text{true},\sigma''}$$

$$\textsc{AndAndFalse2}\ \frac{\rho,\kappa,\sigma \vdash e_1 \Rightarrow \text{true},\sigma' \qquad \rho,\kappa,\sigma' \vdash e_2 \Rightarrow \text{false},\sigma''}{\rho,\kappa,\sigma \vdash e_1\ \texttt{\&\&}\ e_2 \Rightarrow \text{false},\sigma''}$$

$$\textsc{AndAndFalse1}\ \frac{\rho,\kappa,\sigma \vdash e_1 \Rightarrow \text{false},\sigma'}{\rho,\kappa,\sigma \vdash e_1\ \texttt{\&\&}\ e_2 \Rightarrow \text{false},\sigma'}$$

$$\textsc{OrOrTrue1}\ \frac{\rho,\kappa,\sigma \vdash e_1 \Rightarrow \text{true},\sigma'}{\rho,\kappa,\sigma \vdash e_1\ \texttt{||}\ e_2 \Rightarrow \text{true},\sigma'}$$

$$\textsc{OrOrTrue2}\ \frac{\rho,\kappa,\sigma \vdash e_1 \Rightarrow \text{false},\sigma' \qquad \rho,\kappa,\sigma' \vdash e_1 \Rightarrow \text{true},\sigma''}{\rho,\kappa,\sigma \vdash e_1\ \texttt{||}\ e_2 \Rightarrow \text{true},\sigma''}$$

$$\textsc{OrOrFalse}\ \frac{\rho,\kappa,\sigma \vdash e_1 \Rightarrow \text{false},\sigma' \qquad \rho,\kappa,\sigma' \vdash e_1 \Rightarrow \text{false},\sigma''}{\rho,\kappa,\sigma \vdash e_1\ \texttt{||}\ e_2 \Rightarrow \text{false},\sigma''}$$

$$\textsc{DoWhile}\ \frac{\rho,\kappa,\sigma \vdash e_1 => v_1,\sigma' \qquad \rho,\kappa,\sigma' \vdash \texttt{while}\ (e_2)\ e_1 \Rightarrow v,\sigma''}{\rho,\kappa,\sigma \vdash \texttt{do}\ e_1\ \texttt{while}\ (\ e_2\ ) \Rightarrow v,\sigma''}$$

Delspørgsmål b

$$\text{T-}\textsc{AndAnd}\ \frac{\theta,\gamma \vdash e_1 : \text{Bool} \qquad \theta,\gamma \vdash e_2 : \text{Bool}}{\theta,\gamma \vdash e_1\ \texttt{\&\&}\ e_2 : \text{Bool}}$$

$$\text{T-}\textsc{OrOr}\ \frac{\theta,\gamma \vdash e_1 : \text{Bool} \qquad \theta,\gamma \vdash e_2 : \text{Bool}}{\theta,\gamma \vdash e_1\ \texttt{||}\ e_2 : \text{Bool}}$$

$$\textsc{DoWhile}\ \frac{\theta,\gamma \vdash e_1 \Rightarrow \tau \qquad \theta,\gamma \vdash \texttt{while}\ (e_2)\ e_1 : \mathsf{Unit}}{\theta,\gamma \vdash \texttt{do}\ e_1\ \texttt{while}\ (\ e_2\ ) : \text{Unit}}$$

We know this DoWhile type system rule is a bit silly, we're sorry.

Delspørgsmål c

Listing 1: Interpreter.scala

```
1  object Interpreter {
2
3    // ...
4
5    def eval(e: Exp, env: Env, cenv: CEnv, sto: Sto): (Val, Sto) = e match {
6      // ...
7      case BinOpExp(left,AndAndBinOp(),right) =>
8        val (leftval, sto1) = eval(left, env, cenv, sto)
9        if(leftval == BoolVal(false)){
10         (leftval,sto1)
11       }
```

```
      else if(leftval == BoolVal(true)) {
        val res = eval(right, env, cenv, sto1)
        if(res._1 == BoolVal(true)){
          res
        } else if (res._1 == BoolVal(false)){
          res
        } else throw new InterpreterError("Argument not a boolean",e)
      } else throw new InterpreterError("Argument not a boolean",e)
    case BinOpExp(left,OrOrBinOp(),right) =>
      eval(left,env,cenv,sto) match{
        case (BoolVal(a),sto1) => if(!a){
          (BoolVal(a),sto1)
        } else
          eval(right,env,cenv,sto1) match{
            case (BoolVal(b),sto2) => (BoolVal(b||a),sto2)
            case _ => throw new InterpreterError("Argument not a boolean",e)
          }
        case _ => throw new InterpreterError("Argument not a boolean",e)
      }
    case DoWhileExp(body, guard) =>
      val dopeart = eval(body,env,cenv,sto)
      eval(WhileExp(cond,body),env,cenv,dopeart._2)
    // ...
  }

  // ...
}
```

Listing 2: TypeChecker.scala

```
object TypeChecker {

  // ...

  def typeCheck(e: Exp, tenv: TypeEnv, ctenv: ClassTypeEnv): Type = e match {
    // ...
    case BinOpExp(leftexp, op, rightexp) =>
      val lefttype = typeCheck(leftexp, tenv, ctenv)
      val righttype = typeCheck(rightexp, tenv, ctenv)
      op match {
        // ...
        case AndAndBinOp() =>
          (lefttype,righttype) match{
            case (BoolType(),BoolType())=> BoolType()
            case _=> throw new TypeError(s"Type mismatch at '${unparse(op)}',
                unexpected types ${unparse(lefttype)} and ${unparse(righttype)}", op
                )
          }
        case OrOrBinOp() =>
          (lefttype,righttype) match{
            case (BoolType(),BoolType())=> BoolType()
            case _ => throw new TypeError(s"Type mismatch at '${unparse(op)}',
                unexpected types ${unparse(lefttype)} and ${unparse(righttype)}", op
                )
          }
      }
    // ...
    case DoWhileExp(body,cond) =>
      val firstCheck = typeCheck(body,tenv,ctenv)
      typeCheck(WhileExp(cond,body),tenv,ctenv)
    // ...
  }

  // ...
```

```
}
```

Listing 3: Unparser.scala

```
object Unparser {

  def unparse(n: AstNode): String = n match {
    // ...
    case DoWhileExp(body,cond) =>
      val bod = unparse(body)
      val con = unparse(cond)
      "do "+bod+" while ("+con+")"

    // ...
    case AndAndBinOp() => "&&"
    case OrOrBinOp() => "||"
  }

  // ...
}
```