Aflevering 7

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

18. marts 2019

Opgave 78

```
1  sealed abstract class List2[T]
2  case class Nil[T]() extends List2[T]
3  case class Cons[T](x: T, xs: List2[T]) extends List2[T]
4
5  def length[T](xs: List2[T]): Int = xs match {
6    case Nil() => 0
7    case Cons(_, ys) => 1 + length(ys)
8  }
9
10 type Set2[A] = List2[A]
11
12 def makeEmpty[A](): Set2[A] = Nil[A]()
13 def isEmpty[A](set: Set2[A]): Boolean = set match{
14   case Nil() => true
15   case Cons(_,_) => false
16 }
17 def size[A](set: Set2[A]): Int = length(set)
18 def add[A](set: Set2[A], x: A): Set2[A] = if(contains(set,x)) set else Cons(x,set
       )
19 def contains[A](set: Set2[A], x: A): Boolean = set match{
20   case Nil() => false
21   case Cons(x,ys) => true
22   case Cons(_,ys) => contains(ys,x)
23 }
24 def remove[A](set: Set2[A], x: A): Set2[A] = {
25   def r[T](s: Set2[T],x: T,ass: Set2[T]): Set2[T] = s match{
26     case Nil() => ass
27     case Cons(x,ys) => r(ys,x,ass)
28     case Cons(y,ys) => r(ys,x,Cons(y,ass))
29     }
30   r(set,x,Nil[A]())
31   }
32 def union[A](set1: Set2[A], set2: Set2[A]): Set2[A] = set2 match{
33   case Nil() => set1
34   case Cons(x,xs) => union(add(set1,x),xs)
35 }
36 def intersection[A](set1: Set2[A], set2: Set2[A]): Set2[A] = {
37   def inter[A](s1: Set2[A], s2: Set2[A], ass: Set2[A]): Set2[A] = s1 match{
38     case Nil() => ass
39     case Cons(x,xs) => if(contains(s2,x)) inter(xs,s2,Cons(x,ass)) else inter(xs,
         s2,ass)
40   }
41   inter(set1,set2,Nil())
42 }
43 def difference[A](set1: Set2[A], set2: Set2[A]): Set2[A] = {
44   def diff[A](s1: Set2[A], s2: Set2[A], ass: Set2[A]): Set2[A] = s1 match{
45     case Nil() => ass
46     case Cons(x,xs) => if(contains(s2,x)) diff(xs,s2,ass) else diff(xs,s2,Cons(x,
         ass))
```

```
    }
    diff(set1,set2,Nil())
  }
```

```
package miniscala

import miniscala.A7._
import miniscala.Ast._

/**
  * Computation of free variables (or rather, identifiers).
  */
object Vars {

  def freeVars(e: Exp): Set2[Id] = e match {
    case _: Literal => makeEmpty[Id]()
    case VarExp(x) => add(makeEmpty[Id](),x)
    case BinOpExp(leftexp, _, rightexp) => union(freeVars(leftexp),freeVars(
        rightexp))
    case UnOpExp(_, exp) => freeVars(exp)
    case IfThenElseExp(condexp, thenexp, elseexp) => union(union( freeVars(condexp)
        , freeVars(thenexp) ), freeVars(elseexp))
    case BlockExp(vals, defs, exp) =>
      var fv = freeVars(exp)
      for (d <- defs)
        fv = union(fv,freeVars(d))
      for (d <- defs)
        fv = difference(fv,declaredVars(d))
      for (d <- vals.reverse)
        fv = union(difference(fv,declaredVars(d)),freeVars(d))
      fv
    case TupleExp(exps) =>
      var fv = makeEmpty[Id]()
      for (exp <- exps)
        fv = union(fv,freeVars(exp))
      fv
    case MatchExp(exp, cases) =>
      var fv = freeVars(exp)
      for (c <- cases){
        fv = union(fv,difference(freeVars(c.exp),listToSet2[Id](c.pattern)))
      }
      fv
    case CallExp(funexp, args) =>
      var fv = makeEmpty[Id]()
      for (a <- args)
        fv = union(fv,freeVars(a))
      fv
    case LambdaExp(params, body) =>
       difference(freeVars(body),listToSet2(params.map(p => p.x)))
  }

  def freeVars(decl: Decl): Set2[Id] = decl match {
    case ValDecl(_, _, exp) => freeVars(exp)
    case DefDecl(_, params, _, body) => difference(freeVars(body),listToSet2(params
        .map(p => p.x)))
  }

  def declaredVars(decl: Decl): Set2[Id] = decl match {
    case ValDecl(x, _, _) => add(makeEmpty[Id](),x)
    case DefDecl(x, _, _, _) => add(makeEmpty[Id](),x)
  }

  def listToSet2[T](list: List[T]): Set2[T] = {
```

```
    var ass = makeEmpty[T]()
    for(a <- list){
      ass = add(ass,a)
    }
    ass
  }
}
  def foldRight[A,B](xs: Set2[A], z: B, f: (A, B) => B): B = xs match {
    case Nil() => z
    case Cons(y, ys) => f(y, foldRight(ys, z, f))
  }
  def foldLeft[A,B](xs: Set2[A], z: B, f: (B, A) => B): B = xs match {
    case Nil() => z
    case Cons(y, ys) => foldLeft(ys, f(z, y), f)
  }

  def makeInitialTypeEnv(program: Exp): TypeEnv = {
    val tenv: TypeEnv = Map()
    miniscala.A7.foldRight(Vars.freeVars(program),tenv,(x:Id,t: TypeEnv)=>t+(x->
        IntType()))
  }

  def makeInitialEnv(program: Exp): Env = {
    miniscala.A7.foldRight(Vars.freeVars(program),Map[Id, Val](),(x:Id,t:Env) => {
      print(s"Please provide an integer value for the variable $x: ")
      t + (x -> IntVal(StdIn.readInt()))
    })
  }
```

Opgave 79

```
object InterpreterOLD {
// ...
  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 TupleExp(exps) =>
      trace("Evaluation tuple of expressions")
      var vals = List[Val]()
      for (ex <- exps)
        vals = eval(ex, env) :: vals
      TupleVal(vals.reverse)
// ...
}

object Interpreter {
  // ...
  def eval(e: Exp, env: Env): Val = e match {
    // ...
    case BlockExp(vals, defs, exp) =>
      var env1 = env
      trace("Calculating variable values and adding to variable environment")
```

```
31        env1 = vals.foldLeft(env1)((en:Env,d:ValDecl) => {
32          val dexp = eval(d.exp, en)
33          checkValueType(dexp, d.opttype, d)
34          en + (d.x -> dexp)
35        })
36        env1 = defs.foldLeft(env1)((en: Env,d:DefDecl)=> {
37          en + (d.fun -> ClosureVal(d.params,d.optrestype,d.body,en,defs) )
38        })
39        eval(exp, env1)
40      case TupleExp(exps) =>
41        trace("Evaluation tuple of expressions")
42        TupleVal(exps.foldLeft(List[Val]())((v: List[Val],e:Exp)=> eval(e, env) :: v)
              .reverse)
43      // ...
44    }
45  }
```