Functional Programming

Final Exam

Friday, December 20 2019

Your points are *precious*, don't let them go to waste!

Your Time All points are not equal. Note that we do not think that all exercises have the same difficulty, even if they have the same number of points.

Your Attention The exam problems are precisely and carefully formulated, some details can be subtle. Pay attention, because if you do not understand a problem, you cannot obtain full points.

Stay Functional You are strictly forbidden to use return statements, mutable state (vars) and mutable collections in your solutions.

Some Help The last page of this exam contains an appendix which is useful for formulating your solutions. You can detach this page and keep it aside.

Exercise	**Points**	**Points Achieved**
1	10	
2	10	
3	10	
4	10	
Total	40	

Exercise 1: Pure Functional programming (10 points)

The following function is intended to remove duplicate elements from a list:

```
def uniq[T](list: List[T]): List[T] =
  var result = List.empty[T]
  var cur = list
  while cur.nonEmpty do
    val h = cur.head
    cur = cur.tail
    if !result.contains(h)
      result = h :: result
  result
```

However, the implementation is undesirable for a few reasons:

1. it uses **`var`** and therefore is not functional
2. it has quadratic complexity (because `result.contains` is linear in size of `result`)
3. it reverses the order of the argument's elements

Your task is to reimplement `uniq` with the following requirements:

1. The returned list should contain all members of the original list, with duplicates removed
2. Your implementation cannot use any mutable state (no **`var`**, no mutable collections)
3. Your implementation can only use methods defined in the appendix (in particular, it cannot use `distict` method on `List`)
4. The order of elements in the result must be the same as in the original list
5. Your implementation must have better than quadratic complexity

Hint: to implement a subquadratic time solution, you need to somehow check in constant-time whether the result you have built up so far already contains the element you want to add. This can be achieved with a `Set` data structure.

Examples of how correctly implemented `uniq` should work:

```
uniq(List(1,2,3))       == List(1,2,3)
uniq(List(1,2,3,2))     == List(1,2,3)
uniq(List(1,2,3,1,3,4)) == List(1,2,3,4)
```

```scala
def uniq[T](list: List[T]): List[T] =
```

Exercise 2: Interpreter (10 points)

Your task in this exercise is to extend the interpreter presented in the lab to support by-name arguments. We start with an example that shows how by-name arguments work in Scala. Consider the `List.fill` function:

```
object List {
  // Produces a collection containing the results of some element computation a number of times.
  def fill[A](n: Int)(elem: => A): List[A] =
    if n != 0 then elem :: fill(n - 1)(elem)
    else Nil
}
```

Notice that `elem` is passed by name. As a result, in `List.fill(n)(expr)`, `expr` is evaluated `n` times. For example, we can construct `List(1, 2, 3 ,4)` with a **`var`**:

```
var count = 0
List.fill(4)({ count += 1; count }) // List(1, 2, 3, 4)
```

In Scala, this is implemented by transforming function with by-name arguments, and calls to these functions. The above definition of `fill` will be transformed as follows:

```
def fill[A](n: Int)(elem: () => A): List[A] =
  if n != 0 then elem() :: fill(n - 1)(() => elem())
  else Nil
```

Likewise, usages of `fill` are transformed as follows: `List.fill(4)(() => { count += 1; count })`.

More precisely:

- The type of by-name arguments become nullary-functions (functions with zero arguments); `elem: => A` becomes `elem: () => A`
- In the body of by-name function definitions, references to by-name arguments become nullary-function application; `elem` becomes `elem()`
- At call site, arguments to by-name functions are wrapped into a nullary-function; `List.fill(n)(expr)` becomes `List.fill(n)(() => expr)`

To add by-name to our `Expr` language, we extend the enum for `Expr` with two additional constructs, `FunByName` and `CallByName`:

```
enum Expr {
  case Constant(value: Int)
  case Name(name: String)
  case BinOp(op: BinOps, arg1: Expr, arg2: Expr)
  case IfNonzero(cond: Expr, caseTrue: Expr, caseFalse: Expr)
  case Call(function: Expr, arg: Expr)
  case Fun(arg: String, body: Expr)
  // Added for this exercise:
  case FunByName(arg: String, body: Expr)
  case CallByName(function: Expr, arg: Expr)
}
```

`FunByName` is a function definition with a single by-name argument. `CallByName` is call to a function taking a by name argument. As an example, assuming we had extended `Expr` to also have `Cons` and `Empty`, `fill` would be encoded as follows:

```
// def fill[A](n: Int)(elem: => A): List[A] =
//   if n != 0 then elem :: fill(n - 1)(elem)
//   else Nil
Fun("n", FunByName("elem",
  IfNonzero(
    Name("n"),
    Cons(
      Name("elem"),
      CallByName(
        Call(
          Name("fill"),
          BinOp(BinOps.Minus, Name("n"), Constant(1))
        ),
        Name("elem")
      )
    ),
    Empty
  )
))
```

Your implementation will be done using desugaring. This means that expressions containing `FunByName` and `CallByName` will be replaced by equivalent expressions without those two constructs. Therefore, expressions that come out of desugaring should not contain any `FunByName` or `CallByName`. As a result, the implementation of `eval`, `subst` and `alphaConvert` don't need to be updated!

We provide you with a skeleton implementation of `desugar` that traverses expressions and applies the desugaring at every step. Your task is to complete the implementation of `desugar` to transform `FunByName` and `CallByName` into semantically equivalent expressions using `Call` and `Fun`.

Similarly to by-name arguments in Scala, your desugaring should rewrite by-name arguments according to the following scheme:

- In the body of by-name function definitions, references to by-name arguments become nullary-function applications
- At call site, arguments to by-name functions are wrapped into a nullary-function

Note that since in the `Expr` language all functions have exactly one argument, you need to emulate nullary-functions using dummy parameters. For instance an nullary-function definition can be emulated with `Fun("unused", expr)` where `unused` is unused in `expr`. Nullary-function application can be emulated by passing a dummy parameters, such as `Call(Name("f"), Constant(0))`.

Your `desugar` implementation can use other functions from the interpreter. The complete implementation of the interpreter is available as a reference in the appendix (copy pasted from the lab). Note that you are not meant to evaluate expression while desugaring them.

```
/** Evaluates a program e given a set of top level definition defs */
def eval(e: Expr, defs: DefEnv): Expr = ...

/** Substitutes Name(n) by r in e. */
def subst(e: Expr, n: String, r: Expr): Expr = ...

/** Computes the set of free variable in e. */
def freeVars(e: Expr): Set[String] = ...

/** Substitutes Name(n) by Name(m) in e. */
def alphaConvert(e: Expr, n: String, m: String): Expr = ...
```

```scala
def desugar(e: Expr): Expr =
  e match
    case Constant(_) => e
    case Name(_) => e
    case BinOp(op, arg1, arg2) => BinOp(op, desugar(arg1), desugar(arg2))
    case IfNonzero(cond, caseTrue, caseFalse) =>
      IfNonzero(desugar(cond), desugar(caseTrue), desugar(caseFalse))
    case Call(function, arg) => Call(desugar(function), desugar(arg))
    case Fun(arg, body) => Fun(arg, desugar(body))
    // TODO: Add extra cases
```

Exercise 3: Typeclasses (10 points)

You are writing a web application and you need a collision-resistant hash function to thwart hackers. Hashing to `Long` produces more collision-resistant hashes, so you decide to write a type-class which allows you to do just that.

Your hashing type-class is defined as follows:

```
trait Hash[T] {
  def hash(t: T): Long
}
```

A `digest` method is also provided to compute the hash of a specific element given a `Hash` for it.

```
def digest[T](t: T)(given h: Hash[T]): Long = h.hash(t)
```

The most common way to hash an object is to calculate the hashes of each member of the object, and then combine them.

One of the properties of a good hashing function is that the calculated hash values must be well distributed. Therefore, when calculating a hash of an object, it is not enough to simply calculate the hashes of a member of the object and then add them together. Instead of addition, an appropriate *mixing* function must be used.

Your task is to write a few `Hash` definitions. In each case, the value your `hash` method returns should be the result of mixing together the hashes of all members of the argument to the method. It is not an error if you mix in additional constant values (though it is not necessary).

You are provided the following definitions:

```
def mix(l: Long, k: Long): Long = ...

given IntHash: Hash[Int]    = ...
given StrHash: Hash[String] = ...
```

As an example, here is how to calculate a hash of a pair:

```
given PairHash[T, U](given Hash[T], Hash[U]): Hash[(T, U)] {
  def hash(pair: (T, U)): Long =
    mix(digest(pair._1), digest(pair._2))
}
```

(a) `Hash[(T, U, S)]` (2.5 points)

Write a **`given`** definition to create `Hash[(T, U, S)]` from `Hash[T], Hash[U], Hash[S]`.

(b) `Hash[List[T]]` (2.5 points)

Write a **given** definition to create `Hash[List[T]]` from `Hash[T]`. Return `0L` for `Nil`.

(c) `Hash[Either[L, R]]` (2.5 points)

`Either` is defined as

```
enum Either[L, R] {
  case Left(l: L)
  case Right(r: R)
}
```

Write a **given** definition to create `Hash[Either[L, R]]` from `Hash[L], Hash[R]`. To avoid clashes of the hashes of `Left(x)` and `Right(x)`, you should `mix` an extra `1L` to left and a `2L` to `Right`.

(d) `Hash[Tree]` (2.5 points)

Write a **`given`** definition to create `Hash[Tree]` without using `mix` directly. You may use all the definitions you have written so far, as well as all the ones from the exercise description.

```
enum Tree {
  case Branch(children: List[Int])
  case Leaf(id: Int, elem: String)
}
```

Hint: you will need to create an object for which you already can calculate a hash, and which contains all the members of your `Tree` object.

Exercise 4: Lazy Lists (10 points)

In this exercise you will have to compute all the permutations of a `LazyList`. You will do this by induction and will be guided in your implementation with 3 subquestions.

(a) LazyList element insertion (3 points)

Implement element insertion into a `LazyList` at a given index `idx`. The resulting `LazyList` will be equivalent to the input `LazyList` until position `idx`, contains the inserted element at position `idx`, and then contains the remaining elements of the input.

For example:

```
insert(LazyList("a", "b", "d"), "c", 2)
```

Evaluates to:

```
LazyList("a", "b", "c", "d")
```

You can assume that the index passed to `insert` is never out of bounds, that is, `idx >= 0` and `idx <= input.size`, where `idx == 0` means inserting at the begining of the `LazyList` and `idx == input.size` means inserting at the end of the `LazyList`.

Complete the implementation of `insert`:

```
def insert[T](input: LazyList[T], elem: T, idx: Int): LazyList[T] =
```

(b) Induction step (3 points)

For this part you will implement one step of the induction.

Given a `LazyList` of all the permutations of the first `n` elements, and the element at position `n + 1`, compute all the permutations of the first `n + 1` elements. You may want to use `insert` in your implementation.

For example:

```
next("c", 2, LazyList(LazyList("a", "b"), LazyList("b", "a")))
```

Evaluates to:

```
LazyList(
  LazyList("c", "a", "b"), LazyList("a", "c", "b"), LazyList("a", "b", "c"),
  LazyList("c", "b", "a"), LazyList("b", "c", "a"), LazyList("b", "a", "c")
)
```

Note that your implementation does not need to produce elements in the same order as the example.

Complete the implementation of `next`:

```
def next[T](elem: T, n: Int, input: LazyList[LazyList[T]]): LazyList[LazyList[T]] =
```

(c) LazyList permutations (4 points)

Finally, recursively compute all the permutations of a `LazyList`. The induction is done by computing all the permutations of the first `n` elements of a `LazyList`, then using `next` to compute all the permutation of the first `n + 1` elements.

For example `permutations(LazyList())` evaluates to `LazyList(LazyList())`, and

```
permutations(LazyList("a", "b", "c"))
```

Evaluates to:

```
LazyList(
  LazyList("c", "a", "b"), LazyList("a", "c", "b"), LazyList("a", "b", "c"),
  LazyList("c", "b", "a"), LazyList("b", "c", "a"), LazyList("b", "a", "c")
)
```

Note that your implementation does not need to produce elements in the same order as the example.

Complete the implementation of `permutations`:

```
def permutations[T](input: LazyList[T]): LazyList[LazyList[T]] =
```

Appendix: Scala Standard Library Methods

Here are some methods from the Scala standard library that you may find useful, on `List[A]`:

- `xs.head: A`: returns the first element of the list. Throws an exception if the list is empty.
- `xs.tail: List[A]`: returns the list `xs` without its first element. Throws an exception if the list is empty.
- `x :: (xs: List[A]): List[A]`: prepends the element `x` to the left of `xs`, returning a `List[A]`.
- `xs ++ (ys: List[A]): List[A]`: appends the list `ys` to the right of `xs`, returning a `List[A]`.
- `xs.apply(n: Int): A`, or `xs(n: Int): A`: returns the `n`-th element of `xs`. Throws an exception if there is no element at that index.
- `xs.drop(n: Int): List[A]`: returns a `List[A]` that contains all elements of `xs` except the first `n` ones. If there are less than `n` elements in `xs`, returns the empty list.
- `xs.filter(p: A => Boolean): List[A]`: returns all elements from `xs` that satisfy the predicate `p` as a `List[A]`.
- `xs.flatMap[B](f: A => List[B]): List[B]`: applies `f` to every element of the list `xs`, and flattens the result into a `List[B]`.
- `xs.foldLeft[B](z: B)(op: (B, A) => B): B`: applies the binary operator `op` to a start value and all elements of the list, going left to right.
- `xs.foldRight[B](z: B)(op: (A, B) => B): B`: applies the binary operator `op` to a start value and all elements of the list, going right to left.
- `xs.map[B](f: A => B): List[B]`: applies `f` to every element of the list `xs` and returns a new list of type `List[B]`.
- `xs.nonEmpty: Boolean`: returns **`true`** if the list has at least one element, **`false`** otherwise.
- `xs.reverse: List[A]`: reverses the elements of the list `xs`.
- `xs.take(n: Int): List[A]`: returns a `List[A]` containing the first `n` elements of `xs`. If there are less than `n` elements in `xs`, returns these elements.
- `xs.size: Int`: returns the number of elements in the list.
- `xs.zip(ys: List[B]): List[(A, B)]`: zips elements of `xs` and `ys` in a pairwise fashion. If one list is longer than the other one, remaining elements are discalaarded. Returns a `List[(A, B)]`.
- `xs.zipWithIndex: List[(A, Int)]`: zips elements of `xs` with their index, for example:

  ```
  List("a", "b").zipWithIndex == List(("a", 0), ("b", 1))
  ```

All these methods are also available on `LazyList[A]` except for `::`, but be careful about their behavior on infinite lazy lists, for example think about what might happen when reversing an infinite lazy list. The following additional methods may also be useful:

- `x #:: (xs: LazyList[A]): LazyList[A]`: prepends the element `x` to the left of `xs`, returning a `LazyList [A]`.

- `xs #::: (ys: LazyList[A]): LazyList[A]`: concatenate two LazyLists together.
- `LazyList.continually[A](elem: A): LazyList[A]`: return an infinite lazy list where all elements are equal to `elem`.
- `LazyList.from[A](start: Int): LazyList[Int]`: Create an infinite lazy list starting at `start` and incrementing by 1.

To pattern match on a `LazyList a`, use the following pattern:

```
a match
  case LazyList() =>
    // empty case ...
  case x #:: xs =>
    // non-empty case ...
```

Here are some methods from Scala standard library that you may find useful, on `Set[A]`:

- `Set.empty[A]`: create an empty set of values of type `A`
- `(as: Set[A]) + (a: A)`: create a new set that contains all members of `as` and also `a`
- `(as: Set[A]) ++ (bs: Set[A]): Set[A]`: return the union of two sets
- `as.map(f: A => B)`: applies `f` to every member of `as` and returns a new set of type `Set[B]`
- `as.filter(f)`: remove from `as` the members for which `f` returns **`false`**
- `as.flatMap(f: A => Set[B])`: applies `f` to every member
- `xs.empty: Boolean`: returns **`true`** if the set zero members, **`false`** otherwise.
- `xs.nonEmpty: Boolean`: returns **`true`** if the list has at least one member, **`false`** otherwise.
- `as.contains(a)`: returns **`true`** if `as` contains `a`
- `as.size`: returns the number of members of `as`
- `as.toList`: returns a list containing all members of `as`

Appendix: interpreter implementation

```
object RecursiveLanguage {
  /** Expression tree, also called Abstract Syntax Tree (AST) */
  enum Expr {
    case Constant(value: Int)
    case Name(name: String)
    case BinOp(op: BinOps, arg1: Expr, arg2: Expr)
    case IfNonzero(cond: Expr, caseTrue: Expr, caseFalse: Expr)
    case Call(function: Expr, arg: Expr)
    case Fun(param: String, body: Expr)
  }
  import Expr._

  /** Primitive operations that operation on constant values. */
  enum BinOps
    case Minus // Other operations omited

  def evalBinOp(op: BinOps)(ex: Expr, ey: Expr): Expr =
    (op, ex, ey) match
      case (BinOps.Minus, Constant(x), Constant(y)) => Constant(x - y)
      case _ => error(s"Type error in ${BinOp(op, ex, ey)}")

  type DefEnv = Map[String, Expr]

  /** Evaluates a progam e given a set of top level definition defs */
  def eval(e: Expr, defs: DefEnv): Expr =
    e match
      case Constant(c) => e
      case Name(n) =>
        defs.get(n) match
          case None => error(s"Unknown name $n")
          case Some(body) => eval(body, defs)
      case BinOp(op, e1, e2) =>
        evalBinOp(op)(eval(e1, defs), eval(e2, defs))
      case IfNonzero(cond, caseTrue, caseFalse) =>
        if eval(cond, defs) != Constant(0) then eval(caseTrue, defs)
        else eval(caseFalse, defs)
      case Fun(n, body) => e
      case Call(fun, arg) =>
        val eFun = eval(fun, defs)
        val eArg = eval(arg, defs)
        eFun match
          case Fun(n, body) =>
            val bodySub = subst(body, n, eArg)
            val res = eval(bodySub, defs)
            res
          case _ => error(s"Cannot apply non-function ${eFun} in a call")
```

```
  /** Substitutes Name(n) by r in e. */
  def subst(e: Expr, n: String, r: Expr): Expr =
    e match
      case Constant(c) => e
      case Name(s) => if s == n then r else e
      case BinOp(op, e1, e2) =>
        BinOp(op, subst(e1, n, r), subst(e2, n, r))
      case IfNonzero(cond, trueE, falseE) =>
        IfNonzero(subst(cond, n, r), subst(trueE, n, r), subst(falseE, n, r))
      case Call(f, arg) =>
        Call(subst(f, n, r), subst(arg, n, r))
      case Fun(param, body) =>
        if param == n then e
        else
          val fvs = freeVars(r)
          if fvs.contains(param) then
            val param1 = differentName(param, fvs)
            val body1 = alphaConvert(body, param, param1)
            Fun(param1, subst(body1, n, r))
          else
            Fun(param, subst(body, n, r))

  def differentName(n: String, s: Set[String]): String =
    if s.contains(n) then differentName(n + "'", s)
    else n

  /** Computes the set of free variable in e. */
  def freeVars(e: Expr): Set[String] =
    e match
      case Constant(c) => Set()
      case Name(s) => Set(s)
      case BinOp(op, e1, e2) => freeVars(e1) ++ freeVars(e2)
      case IfNonzero(cond, trueE, falseE) => freeVars(cond) ++ freeVars(trueE) ++ freeVars(
          falseE)
      case Call(f, arg) => freeVars(f) ++ freeVars(arg)
      case Fun(param, body) => freeVars(body) - param

  /** Substitutes Name(n) by Name(m) in e. */
  def alphaConvert(e: Expr, n: String, m: String): Expr =
    e match
      case Constant(c) => e
      case Name(s) => if s == n then Name(m) else e
      case BinOp(op, e1, e2) =>
        BinOp(op, alphaConvert(e1, n, m), alphaConvert(e2, n, m))
      case IfNonzero(cond, trueE, falseE) =>
        IfNonzero(alphaConvert(cond, n, m), alphaConvert(trueE, n, m), alphaConvert(falseE, n, m
            ))
      case Call(f, arg) =>
        Call(alphaConvert(f, n, m), alphaConvert(arg, n, m))
      case Fun(param, body) =>
        if param == n then e
        else Fun(param, alphaConvert(body, n, m))

  case class EvalException(msg: String) extends Exception(msg)

  def error(msg: String) = throw EvalException(msg)
}
```