Functional Programming

Final Solution

Friday, December 20 2019

Exercise 1: Pure Functional programming (10 points)

```
def uniq[T](list: List[T]): List[T] =
  def rec(ls: List[T], seen: Set[T]) =
    ls match
      case x :: xs =>
        if seen.contains(x) then rec(xs, seen)
        else x :: rec(xs, seen + x)
      case Nil => Nil
  rec(list, Set.empty).reverse
```

Exercise 2: Interpreter (10 points)

```
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))
    case FunByName(arg, body) =>
      Fun(arg, subst(desugar(body), arg, Call(Name(arg), Constant(0))))
      // or desugar(Fun(arg, subst(body, arg, Call(Name(arg), Constant(0)))))
    case CallByName(function, arg) =>
      Call(desugar(function), Fun("unused", desugar(arg)))
      // or desugar(Call(function, Fun("unused", arg)))
```

Exercise 3: Typeclasses (10 points)

(a)

```
given TripleHash[T, U, S](given Hash[T], Hash[U], Hash[S]): Hash[(T, U)]
  def hash(triple: (T, U, S)): Long =
    mix(mix(digest(triple._1), digest(triple._2)), digest(triple._3))
```

(b)

```
given ListHash[T](given Hash[T]): Hash[List[T]]
  def hash(xs: List[T]): Long =
    xs.foldLeft(0L)((acc, x) => mix(acc, digest(x)))
```

(c)

```
given EitherHash[L, R](given Hash[L], Hash[R]): Hash[Either[L, R]]
  def hash(either: Either[L, R]): Long =
    either match
      case Left(x) => mix(1L, digest(x))
      case Right(x) => mix(2L, digest(x))
```

(d)

```
given TreeHash: Hash[Tree]
  def hash(tree: Tree): Long =
    digest(tree match
      case Branch(children) => Left(children)
      case Leaf(id, elem) => Right((name, id))
    )
```

Exercise 4: Lazy Lists (10 points)

(a)

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

(b)

```
def next[T](elem: T, n: Int, input: LazyList[LazyList[T]]): LazyList[LazyList[T]] =
  input.flatMap(ls => (0 to n).map(idx => insert(ls, elem, idx)))
```

(c)

```
def permutations[T](input: LazyList[T]): LazyList[LazyList[T]] =
  input match {
    case LazyList() => LazyList(LazyList())
    case x #:: xs => nextPermut(x, xs.length, permutations(xs))
  }
```