CMSC 433: Midterm Exam (Spring 2024)

Question 1 (20 points)

Consider the two standard folds in Haskell, `foldl` and `foldr`, whose definitions are given below:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b [] = b
foldr f b (x:xs) = f x (foldr f b xs)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f b [] = b
foldl f b (x:xs) = foldl f (f b x) xs
```

Now consider the following two very similar but distinct folds over lists of integers:

```
f1 :: [Int] -> Int
f1 = foldr (-) 0

f2 :: [Int] -> Int
f2 = foldl (-) 0
```

Write two Dafny programs, one corresponding to each of `f1` and `f2`, but operating on arrays rather than lists. We've given you the method signatures to get you started:

```
method f1(a:array<int>) returns (out:int) {

}
```

```
method f2(a:array<int>) returns (out:int) {

}
```

Question 2 (20 points)

For each of the Haskell expressions below, write their (most general) Haskell type or "ill-typed" if it contains a type error. The type signatures of all functions below are provided in the appendix at the end.

(a) `\x -> x`
Example answer:
`a -> a`

(b) `\x y -> (x,y)`

(c) `\x y -> if x == y then show x else show (x,y)`

(d) `\x l -> x : l ++ l ++ [x]`

(e) `foldr (const 1)`

(f) `(,"42")`

(g) `(,) "42"`

(h) `reverse . reverse`

(i) `\l -> filter (< 42) (l ++ l)`

(j) `let f x = x in (f 'a', f True)`

(k) `fmap (fmap (+1)) [Nothing]`

Question 3 (20 points)

For each of the types below, write a Haskell expression that has that type. Don't write trivial expressions (such as [], `Nothing`, or `undefined`) unless there is no other option. You can use any function from the appendix, do syntax, list comprehensions, or any valid Haskell.

(a) `Int -> Int`
Example answers:
`\x -> x + 1`
`(+1)`

(b) `Bool -> [Bool]`

(c) `a -> Maybe b`

(d) `(Int -> Char -> Bool) -> [Int] -> [Char] -> [Bool]`

(e) `(a -> b -> c) -> Maybe a -> Maybe b -> Maybe c`

(f) `(a -> b) -> (b -> Bool) -> [a] -> [b]`

(g) `Maybe a -> (a -> [b]) -> [(a,b)]`

(h) `Eq a => a -> [a] -> [a]`

(i) `Show a => [a] -> IO String`

(j) `(a,b) -> (a -> b -> c) -> c`

(k) `((a,b) -> c) -> c`

Question 4 (20 points)

Consider the following Dafny program that inefficiently calculates the cube of a number:

```
method Cube(m:int) returns (y:int)
  requires m > 0
  ensures y == m*m*m
{
  y := 0;
  var x := m * m;
  var z := m;
  while (z > 0)
  {
    z := z - 1;
    y := y + x;
  }
}
```

Fill in the annotations in the following program to show that the Hoare triple given by the outermost pre- and post- conditions is valid. Be completely precise and pedantic in the way you apply Hoare rules—write assertions in *exactly* the form given by the rules rather than just logically equivalent ones. The provided blanks have been constructed so that if you work backwards from the end of the program you should only need to use the rule of consequence in the places indicated with `->>`. These implication steps can (silently) rely on all the usual rules of arithmetic.

```
{{ m > 0 }} ->>
{{                                                    }}
  y := 0;
{{                                                    }}
  var x := m * m;
{{                                                    }}
  var z := m;
{{                                                    }}
  while (Z > 0) {
{{                                                    }} ->>
{{                                                    }}
    z := z - 1;
{{                                                    }}
    y := y + x;
{{                                                    }}
}
{{                                                    }} ->>
{{ y = m * m  * m}}
```

Question 5 (20 points)

Implement the following Haskell functions:

(a) Implement a function `weave` that given two lists with elements of the same type, returns a list with elements alternating between the two lists. For example:

```
weave [1,2,3] [4,5,6] = [1,4,2,5,3,6]
```

You can assume that the lists have the same length.

(b) Implement a function `powerset`, that given a list of elements of some type, computes the list of all of its sublists, including the empty list and the original list itself, in any order. For example:

```
powerset [1,2,3] = [[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

Typeclass Definitions

```
class Semigroup a where
  (<>)   :: a -> a -> a

class Semigroup m => Monoid m where
  mempty :: m

class  Show a  where
  show   :: a   -> String

class  Eq a  where
  (==)   :: a -> a -> Bool
  (/=)   :: a -> a -> Bool

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

class Functor f where
  fmap   :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure   :: a -> f a
  (<*>)  :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m

class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]

class Num a where
  (+), (-), (*)       :: a -> a -> a
  negate              :: a -> a
  abs                 :: a -> a
  signum              :: a -> a
  fromInteger         :: Integer -> a
```

Prelude Functions and Datatypes

Booleans

```
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
(||) :: Bool -> Bool -> Bool
not  :: Bool -> Bool
```

Lists

```
data [] a = [] | a : [a]

(++) :: [a] -> [a] -> [a]
head, last :: [a] -> a
tail, init :: [a] -> [a]

null   :: Foldable t => t a -> Bool
length :: Foldable t => t a -> Int
map :: (a -> b) -> [a] -> [b]
reverse :: [a] -> [a]
intersperse :: a -> [a] -> [a]
transpose :: [[a]] -> [[a]]
foldl  :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldr  :: Foldable t => (a -> b -> b) -> b -> t a -> b
concat :: Foldable t => t [a] -> [a]
and, or  :: Foldable t => t Bool -> Bool
any, all :: Foldable t => (a -> Bool) -> t a -> Bool
sum, product :: (Foldable t, Num a) => t a -> a
maximum, minimum :: (Foldable t, Ord a) => t a -> a
repeat :: a -> [a]
replicate :: Int -> a -> [a]
take, drop :: Int -> [a] -> [a]
splitAt :: Int -> [a] -> ([a], [a])
takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
group :: Eq a => [a] -> [[a]]
inits, tails :: [a] -> [[a]]
elem, notElem :: (Foldable t, Eq a) => a -> t a -> Bool
lookup :: Eq a => a -> [(a, b)] -> Maybe b
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
filter :: (a -> Bool) -> [a] -> [a]
zip :: [a] -> [b] -> [(a, b)]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
nub :: Eq a => [a] -> [a]
delete :: Eq a => a -> [a] -> [a]
sort :: Ord a => [a] -> [a]
sortOn :: Ord b => (a -> b) -> [a] -> [a]
```

Maybe

```
data Maybe a = Nothing | Just a

maybe :: b -> (a -> b) -> Maybe a -> b
isJust, isNothing :: Maybe a -> Bool
listToMaybe :: [a] -> Maybe a
maybeToList :: Maybe a -> [a]
catMaybes :: [Maybe a] -> [a]
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
```

Functors, Applicatives, and Monads

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$) :: Functor f => a -> f b -> f a
($>) :: Functor f => f a -> b -> f b

liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
(*>) :: Applicative f => f a -> f b -> f b
(<*) :: Applicative f => f a -> f b -> f a
when :: Applicative f => Bool -> f () -> f ()

(>>) :: Monad m => m a -> m b -> m b
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
sequence  :: Monad m => [m a] -> m [a]
join :: Monad m => m (m a) -> m a
zipWithM :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
foldM :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b
replicateM :: Applicative m => Int -> m a -> m [a]

liftM :: Monad m => (a1 -> r) -> m a1 -> m r
liftM2 :: Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
...
```

1 Hoare Rules

```
  { P } s1 { Q }
  { Q } s2 { R }
-------------------  (hoare_seq)
{ P } s1; s2 { R }


----------------------- (hoare_asgn)
{Q [x := a]} x := a {Q}


  {P'} c {Q'}
P ==> P'    Q' ==> Q
-------------------- (hoare_consequence)
   {P} c {Q}

         {P &&  b} s1 {Q}
         {P && !b} s2 {Q}
-------------------------------- (hoare_if)
{P} if b { s1 } else { s2 } {Q}

         {P && b} s {P}
---------------------------- (hoare_while)
{P} while b { s } {P && !b}
```