Quiz (Week 9)

Type Families and GADTs

Question 1

Here is a type-level implementation of addition on natural numbers.

```
data N = Z | S N
type family Add (n :: N) (m :: N) :: N
type instance Add Z m = m
type instance Add (S n) m = S (Add n m)
```

What is a correct definition of multiplication?

1. ✗

   ```
   type family Mul (n :: N) (m :: N) :: N
   type instance Mul Z m = m
   type instance Mul (S n) m = S (Mul n m)
   ```

2. ✗

   ```
   type family Mul (n :: N) (m :: N) :: N
   type instance Mul Z m = Z
   type instance Mul (S n) m = S (Mul n m)
   ```

3. ✗

   ```
   type family Mul (n :: N) (m :: N) :: N
   type instance Mul Z m = Z
   type instance Mul (S n) m = Add n (Mul n m)
   ```

4. ✔

   ```
   type family Mul (n :: N) (m :: N) :: N
   type instance Mul Z m = Z
   type instance Mul (S n) m = Add m (Mul n m)
   ```

The first definition is addition, not multiplication. The second just returns the first argument. The third returns the sum of each number up to the first argument. The fourth is actually multiplication.

Question 2

Given the below impementation of a length-indexed vector:

```
data Vec a :: N -> * where
  Nil :: Vec a Z
  Cons :: a -> Vec a n -> Vec a (S n)
```

Which of the following is a correct implementation of `append` for these vectors?

1. ✗

```
append :: Vec a n -> Vec a n -> Vec a n
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

2. ✓

```
append :: Vec a m -> Vec a n -> Vec a (Add m n)
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

3. ✗

```
append :: Vec a m -> Vec a n -> Vec a (Add m n)
append xs Nil = xs
append xs (Cons y ys) = Cons y (append xs ys)
```

4. ✗

```
append :: Vec a n -> Vec b n -> Vec (Add a b) n
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

The first answer says that the output vector will be the same size as the two input vectors, which is incorrect. The second answer is correct. The third answer may also seem correct from the type, but the body ends up placing the right hand list before the left hand list, which, aside from being incorrect, will not typecheck with the definition of `Add` provided. The fourth answer tries to add the types rather than the lengths, which is also incorrect.

Question 3

Now we wish to implement a function `concatAll`, which takes a vector of vectors and flattens them into a single vector.

Given this type:

```
concatAll :: Vec (Vec a n) m -> Vec a (Mul m n)
```

Which is the type-correct impementation?

1. ✔

   ```
   concatAll Nil = Nil
   concatAll (Cons v vs) = append v (concatAll vs)
   ```

2. ✗

   ```
   concatAll Nil = Nil
   concatAll (Cons v vs) = append (concatAll vs) v
   ```

3. ✗

   ```
   concatAll Nil = Nil
   concatAll (Cons v vs) = Cons v (concatAll vs)
   ```

4. ✗

   ```
   concatAll Nil = Nil
   concatAll (Cons v vs) = append (concatAll v) (concatAll vs)
   ```

Only the first answer type checks. The second answer appends them in reverse (and would not type check with the correct definition of `Mul`). The third answer just returns the given vector as-is, and the final answer tries to call `concatAll` on an element of the vector which is not type correct.

Existential Types

Question 4

Suppose I impement a `Shape` as an existential type that can be converted to a list of points, like so:

```
data Shape where
  MkShape :: a -> (a -> [(Int, Int)]) -> Shape
```

Comparing this to a more traditional `Shape` impementation:

```
data Shape' = Circle (Int, Int) Int | Polygon [(Int, Int)] -- etc
```

Select all the **true** statements from the following:

1. ✔ New variants of `Shape` can be defined anywhere in the program, whereas the `Shape'` type can only be extended by modifying the type definition.
2. ✔ The `Shape` type is equivalent to `[(Int,Int)]`.
3. ✔ A value of type `Shape'` can express strictly more shapes than `Shape`.
4. ✗ A value of type `Shape` can express strictly more shapes `Shape'`.

1 is correct as we can define new shapes simply by defining a new type and wrapping it in `MkShape`:

```
data Square = Square Int
```

Question 5

Suppose I have a type like the following:

```
data Foo where
  MkFoo :: (a -> (a,a)) -> ((a,a) -> a) -> Foo
```

Select all the **true** statements from the following:

1. ✗ There are no values of type `Foo`.
2. ✗ The only values of type `Foo` are `MkFoo (\x->(x,x)) fst` and `MkFoo (\x->(x,x)) snd`

3. ✔ There is no use for a value of type `Foo`, as the constructor `MkFoo` does not take a value of type `a`.
4. ✔ `Foo` is an existential type.

Question 6

Consider the following two Haskell type definitions.

```
data Wrapper a = Wrap a
```

```
data Wrapper' where
  Wrap' :: (Show a) => a -> Wrapper'
```

What is the difference between the types `Show a => [Wrapper a]` and `[Wrapper']`?

1. ✗ They are equivalent.
2. ✔ The first requires each list element inside the `Wrapper` to be the same type, the second allows each element inside the `Wrapper'` to be of different type.
3. ✗ The first requires each list element inside the `Wrapper` to be of a type implementing `Show`, whereas the second does not.
4. ✗ The second requires each list element inside the `Wrapper'` to be of a type implementing `Show`, whereas the first does not.

Higher-Rank Polymorphism

Question 7

Suppose I have the following program:

```
run f a b = (f a, f b) :: (String, String)
```

Which of the following type signatures would make it type check?

1. ✔ `(a -> String) -> a -> a -> (String, String)`
2. ✗ `(a -> String) -> a -> b -> (String, String)`

3. ✗ `(forall a. Show a => a -> String) -> a -> b -> (String, String)`
4. ✔ `(Show a, Show b) => (forall a. Show a => a -> String) -> a -> b -> (String, String)`

Question 8

This is the type of the built-in Haskell function `runST`, for modelling stateful computations with variables.

```
runST :: (forall s. ST s a) -> a
```

What property does the Rank-N type ensure?

1. ✗ That any type of state can be used in the `ST` computation.
2. ✔ That `STRef` variables allocated within the `ST` computation do not leak out through the return type `a`.
3. ✗ That the type variable `a` cannot occur in `s`.
4. ✗ That `STRef` variables used within the `ST` computation are not concurrently modified by other threads.

Submission is already closed for this quiz. You can click here to check your submission (if any).