Quiz (Week 2)

Types and Constructors

Consider the following type definitions.

```
type T = Int
data E = C T
       | D E
       | E N
newtype N = N X
```

Question 1

Which of the following can identifiers can stand for *types* in the above definitions? Check all that apply.

1. ✗ `C`
2. ✗ `D`
3. ✓ `E`
4. ✓ `N`
5. ✓ `T`

The three types defined are `T`, `E`, and `N`.

Question 2

Which of the following identifiers can stand for *constructors* in the above definitions? Check all that apply.

1. ✓ `C`
2. ✓ `D`
3. ✓ `E`
4. ✓ `N`
5. ✗ `T`

The type `X` introduces three constructors, `C`, `D` and `E`, and the newtype definition defines a constructor `N`.

Types in Design

For each of the following use cases, choose the type definitions that best reflect this use case, eliminating as many possible errors or invalid values as possible.

Question 3

A connection is in one of three states: *connected*, *connecting* or *disconnected*. It also contains the following information:

- The destination IP address
- If it is in the *disconnected* state, the time it disconnected.
- If it is in the *connected* state, the arrival time of the most recent packet, if one exists.
- If it is in the *connected* state, the size of the most recent packet, if one exists.
- If it is in the *connecting* state, the time the connection was initiated.

1. ✗

```
data State = Connected | Connecting | Disconnected

data Connection
  = Connection
      { destination      :: IPAddress
      , state            :: State
      , timeDisconnected :: Maybe Time
      , timeInitiated    :: Maybe Time
      , packetArrival    :: Maybe Time
      , packetSize       :: Maybe Size
      }
```

2. ✗

```
data State
  = Connected    { packetArrival :: Maybe Time
                 , packetSize    :: Maybe Size
                 }
  | Connecting   { timeInitiated    :: Time }
  | Disconnected { timeDisconnected :: Time }
```

```
data Connection
  = Connection
      { destination :: IPAddress
      , state       :: State
      }
```

3. ✗

```
data State = Connected | Connecting | Disconnected

data Connection
  = Connection
      { destination      :: IPAddress
      , state            :: State
      , timeDisconnected :: Maybe Time
      , timeInitiated    :: Maybe Time
      , recentPacket     :: Maybe (Time, Size)
      }
```

4. ✓

```
data State
  = Connected    { recentPacket     :: Maybe (Time, Size) }
  | Connecting   { timeInitiated    :: Time }
  | Disconnected { timeDisconnected :: Time }

data Connection
  = Connection
      { destination :: IPAddress
      , state       :: State
      }
```

The first answer is very unstructured, as it does not enforce the relationship between the state the connection is in and the data that should be present in the connection. For example, this means that it's possible to have a connection in the disconnected state, but without a timestamp. It also allows a packet arrival time to be present *without* a size, which is not possible.

The second answer does enforce the relationship between data and state, but does not fix the problem with the packet size and time.

The third answer does fix the problem with the packet size and time, but does not fix the relationship between data and state.

The fourth answer is therefore correct.

Question 4

A message is encrypted using a password. The system will not allow messages to be encrypted with weak passwords. Messages can only be logged if encrypted.

1. ✗

```
checkStrength :: String -> Bool
encrypt       :: String -> String -> String
log           :: String -> Log -> Log
```

2. ✗

```
newtype Encrypted = E String
checkStrength :: String -> Bool
encrypt       :: String -> String -> Encrypted
log           :: Encrypted -> Log -> Log
```

3. ✓

```
newtype Password  = P String
newtype Encrypted = E String
checkStrength :: String -> Maybe Password
encrypt       :: String -> Password -> Encrypted
log           :: Encrypted -> Log -> Log
```

4. ✗

```
newtype Password  = P String
checkStrength :: String -> Maybe Password
encrypt       :: String -> Password -> String
log           :: String -> Log -> Log
```

There are three requirements:

1. We must prevent the password and message parameters to `encrypt` from being swapped.
2. We must not allow an unchecked password to be used for encryption.
3. We must not allow unencrypted messages to be logged.

The first answer doesn't meet any requirements. The second meets the third requirement only. The third is correct. The last one meets only the first and second requirements.

Monoids and Semigroups

Question 5

Which of the following `Semigroup` instances are *lawful*?

1. ✓

```
newtype X = X Int
instance Semigroup X where
  X a <> X b = X (a + b)
```

2. ✗

```
newtype X = X Int
instance Semigroup X where
  X a <> X b = X (a - b)
```

3. ✗

```
newtype X = X Int
instance Semigroup X where
  X a <> X b = X (abs (a + b))
```

4. ✓

```
newtype X = X Int
instance Semigroup X where
  X a <> X b = X a
```

Subtraction is not associative `(a - b) - c` is not equal to `a - (b - c)`. The terms `abs ((abs (a + b)) + c)` may also not be equal to `abs (a + (abs (b + c)))` such as when a is 0, b is -1 and c is 1.

Question 6

Which of the following is a valid *monoid*?

1. ✗ The type `Integer`, the `max` function and identity element `0`
2. ✓ The type `Bool`, the `(||)` function and identity element `False`
3. ✗ The type `Bool`, the `(||)` function and identity element `True`

4. ✗ The type `Integer`, the function `(\a b -> (a + b) `div` 2)`, and identity element `0`.

> The first answer is not a monoid because `0` is not an identity element for `max`. For example `max (-1) 0 == 0`.
>
> The second answer is a monoid, as `(||)` is associative and `False` is its identity.
>
> The third answer is not a monoid because `True` is not an identity element for `(||)`, as `a || True == True` for all `a`.
>
> The last is not a monoid, because the given average function is not associative.

Relations

Question 7

Check all of the following that are valid *equivalence relations*.

1. ✓ `(==)`
2. ✓ `\x y -> x `mod` 10 == y `mod` 10`
3. ✗ `(>=)`
4. ✗ `(/=)`

> Equality is an equivalence relation, as is congruence mod 10. That is because they satisfy reflexivity, transitivity and symmetry. The ordering `>=` is not symmetric, and inequality is not reflexive or transitive either.

Question 8

Here is a data type definition for a non-empty list in Haskell.

```
data NonEmptyTree a = Leaf a | Node (NonEmptyTree a) (NonEmptyTree a)
```

Which of the following are law-abiding `Functor` instances for `NonEmptyTree`?

1. ✗

```
instance Functor NonEmptyTree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Node l r) = Cons (fmap f r) (fmap f l)
```

2. ✗

```
instance Functor NonEmptyTree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Node l r) = fmap f l
```

3. ✓

```
instance Functor NonEmptyTree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Node l r) = Node (fmap f l) (fmap f r)
```

4. ✗

```
instance Functor NonEmptyTree where
  fmap f (Leaf x) = Leaf (f (f x))
  fmap f (Node l r) = Node (fmap f l) (fmap f r)
```

Option 3 obeys the functor laws. Proof by induction of the first law (`fmap id xs = xs`): Base case, when the second argument is `Leaf x` :

```
fmap id (Leaf x) = Leaf (id x)    -- Definition of fmap
                 = Leaf x         -- Definition of id
```

Inductive case, assuming the second argument is `Node l r` , with the inductive hypothesis that `fmap id t = t` :

```
fmap id (Node l r) = Node (fmap id l) (fmap id r)   -- Definition of fmap
                   = Node l r                       -- Inductive hypothesis
```

The composition law (`fmap f (fmap g t) = fmap (f . g) t`) follows from parametricity.

Options 1 and 3 do not obey the first law, as `fmap id (Node (Leaf 3) (Leaf 1))` does not equal `Node (Leaf 3) (Leaf 1)` , and option 4 is not type correct as `f :: a -> b` , not `a -> a` .

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