Quiz (Week 8)

Phantom Types

Safe Division

Below we have a simple use of *phantom types* to avoid divide-by-zero errors.

```haskell
data Zero
data NonZero
newtype CheckedInt t = Checked Int

safeDiv :: Int -> CheckedInt NonZero -> Int
safeDiv x (Checked y) = x `div` y
```

Question 1

Which of the following statements are *true*?

1. ✗ The types `Zero` and `NonZero` are phantom types.
2. ✓ The type argument `t` is a phantom type parameter.
3. ✗ This code will not compile as `Zero` and `NonZero` have not been defined.
4. ✗ All types declared with `newtype` are phantom types.

> A *phantom type* parameter is a does not occur in the data type definition, i.e. `t` in this example. `Zero` and `NonZero` are types that do not have any constructors, not phantom types. `newtype` introduces a type with the same representation as the internal type, i.e. a cost-free new type, not a phantom type.

Question 2

What is the type of a *smart constructor* for the type `CheckedInt`?

1. ✗ `check :: Int -> CheckedInt t`
2. ✗ `check :: Int -> CheckedInt NonZero`
3. ✗ `check :: Int -> CheckedInt (Either Zero NonZero)`
4. ✓ `check :: Int -> Either (CheckedInt Zero) (CheckedInt NonZero)`

A smart constructor is a function that checks a condition in order to establish an invariant for a particular type. As it's meant to replace a regular data constructor, we expect it to be *total*, and so the type of Option 2 is incorrect, as `check 0` would be undefined. Option 1 is also incorrect, as this returns a `CheckedInt t` for any `t`, so it may for example apply the `NonZero` type tag to zero values. Option 3 is also incorrect, as `Either Zero NonZero` isn't a valid tag for our phantom type. The correct answer is option 4, which can return a `CheckedInt Zero` if the input is `0` and a `CheckedInt NonZero` otherwise.

Linear Algebra

Here Phantom types of a kind `Size` have been used to make vectors of various dimensions. Note that, unlike a GADT solution, the vector size is not statically checked if we use the default `V` data constructor. Rather we must be sure to only use the given *smart constructors* that establish the length invariant about the data:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}

data Size = OneD  | TwoD | ThreeD
newtype Vector (s :: Size) a = V [a]

vec1D :: a -> Vector OneD a
vec1D a = V [a]

vec2D :: (a, a) -> Vector TwoD a
vec2D (a,b) = V [a,b]

vec3D :: (a, a, a) -> Vector ThreeD a
vec3D (a,b,c) = V [a,b,c]
```

Question 3

Which of the following implementations of vector addition encode *statically* the precondition that the vectors must be the same size (and only that precondition), as well as the postcondition that the output vector will be the same size as the input?

1. ✗

   ```
   addV :: (Num n) => Vector a n -> Vector b n -> Vector c n
   addV (V xs) (V ys) = V (zipWith (+) xs ys)
   ```

2. ✓

```
addV :: (Num n) => Vector a n -> Vector a n -> Vector a n
addV (V xs) (V ys) = V (zipWith (+) xs ys)
```

3. ✗

```
addV :: (Num n) => Vector TwoD n -> Vector TwoD n -> Vector TwoD n
addV (V xs) (V ys) = V (zipWith (+) xs ys)
```

4. ✗

```
addV :: (Num n) => Vector a n -> Vector a n -> Vector b n
addV (V xs) (V ys) = V (zipWith (+) xs ys)
```

Option 1 allows the input vectors to be any size, and produces an output vector with an arbitrary size tag, which may not accurately reflect the size of the contents, so it is incorrect. Option 3 requires that both input vectors are two-dimensional and the output vector is also two-dimensional. This is too excessive a restriction, as we only require that the sizes are equal. Option 4 allows the output vector to have an arbitrary size tag, which is also incorrect.

Option 2 requires both input vectors to have the same size `a` and then returns a vector of the same size. This is the correct answer.

Question 4

Suppose we made a `Matrix` type using vectors of vectors:

```
type Matrix r c a = Vector r (Vector c a)
```

What is the correct type for a *matrix multiplication* function?

1. ✗ `mm :: Num a => Matrix r i a -> Matrix r i a -> Matrix r i a`
2. ✗ `mm :: Num a => Matrix r i a -> Matrix i c a -> Matrix r i a`
3. ✗ `mm :: Num a => Matrix r i a -> Matrix c i a -> Matrix i i a`
4. ✓ `mm :: Num a => Matrix r i a -> Matrix i c a -> Matrix r c a`

Matrix multiplication of $A \times B$ requires the number of columns in A be the same as the number of rows in B. The result will have the same number of rows as A and the same number of columns as B. Thus, option 4 is correct.

Option 1 requires the matrices have the same exact dimensions. Option 2 makes the output the same size as A. Option 3 incorrectly mixes the dimensionalities of the inputs.

GADTs

Emptiness-indexed types

Question 5

Here is an ordinary binary tree type presented using GADT syntax:

```
data Tree a where
  Leaf :: Tree a
  Branch :: a -> Tree a -> Tree a -> Tree a
```

Suppose we have this function to get the root element of a `Tree` :

```
root :: Tree a -> a
root (Branch v _ _) = v
```

Unfortunately, this function is *partial*, that is, it is not defined for its entire domain (all values of type `Tree a`). Which of the following GADT definitions of the `Tree` type would enable us to refine the type of `root` to express the precondition that the tree must not be empty?

1. ✗

```
data IsEmpty = Empty | NotEmpty
data Tree (t :: IsEmpty) a where
  Leaf :: Tree Empty a
  Branch :: a -> Tree t a -> Tree t a -> Tree NotEmpty a
```

2. ✗

```
data IsEmpty = Empty | NotEmpty
data Tree (t :: IsEmpty) a where
  Leaf :: Tree t a
  Branch :: a -> Tree t a -> Tree t a -> Tree t a
```

3. ✓

```
data IsEmpty = Empty | NotEmpty
data Tree (t :: IsEmpty) a where
  Leaf :: Tree Empty a
  Branch :: a -> Tree t1 a -> Tree t2 a -> Tree NotEmpty a
```

4. ✗

```
data IsEmpty = Empty | NotEmpty
data Tree (t :: IsEmpty) a where
  Leaf :: Tree t a
  Branch :: a -> Tree t1 a -> Tree t2 a -> Tree NotEmpty a
```

Option 1 is incorrect, as it forces the two children of any node to have the same `Empty` / `NotEmpty` status. Option 2 is incorrect, as it never places any constraint on the type variable `t`. Option 3 is correct, as requiring a `NotEmpty` input to `root` makes it a total function (rules out the `Leaf` case). Option 4 is incorrect, as it allows the `Leaf` node to be tagged `NotEmpty`, so the `root` function would potentially still be partial.

Typed Abstract Syntax

We have here a language of boolean *and* arithmetic expressions, annotated with their types.

```
data Exp a where
  Plus :: Exp Int -> Exp Int -> Exp Int
  Const :: (Show a) => a -> Exp a
  LessOrEq :: Exp Int -> Exp Int -> Exp Bool
  Not :: Exp Bool -> Exp Bool
  And :: Exp Bool -> Exp Bool -> Exp Bool
```

Question 6

The full evaluation function for this language has the following type:

```
eval :: Exp a -> a
```

What are the possible types that `eval` may return?

1. ✗ an `Int` or a `Bool`

2. ✔ an `Int` , a `Bool` , or a value of any type implementing `Show` .
3. ✗ It will always return `Int`
4. ✗ The evaluation function may return any type.

> The return type of `rval` is any `a` for which an `Exp a` can be constructed. This is either and `Int` (from `Plus` or `Const`), a `Bool` (from `LessOrEq` , `Not` , `And` , or `Const`), or any type implementing `Show` (from `Const`).

Question 7

Here is a function that evaluates only arithmetic expressions.

```
evalArith (Plus a b) = evalArith a + evalArith b
evalArith (Const a) = a
```

What type signature should be given to `evalArith` such that it is *total*?

1. ✗ `Exp a -> a`
2. ✗ `Exp Int -> a`
3. ✗ `Exp a -> Int`
4. ✔ `Exp Int -> Int`

> Options 1 and 3, which take an `Exp a` , would crash at runtime if given, e.g. an `And` expression. Option 2 does not typecheck. Thus Option 4 is correct.

Propositional Equality

Question 8

Suppose we had the following GADT:

```
data Id a b where
  Refl :: Id x x
```

Suppose we had a function `f` of type `Id a Int -> a` . Check all of the values below that this type signature indicates may be returned by `f` .

1. ✗ `True`

2. ✓ `34`
3. ✓ `0`
4. ✗ `Refl`
5. ✗ `a`

The only way a value of `Id a Int` could be constructed is if the type `a` was an `Int`. Thus, the function `f` will learn that `a` is an `Int` when it matches on `Refl`, and can then return any integer value, i.e. option 2 and option 3.

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