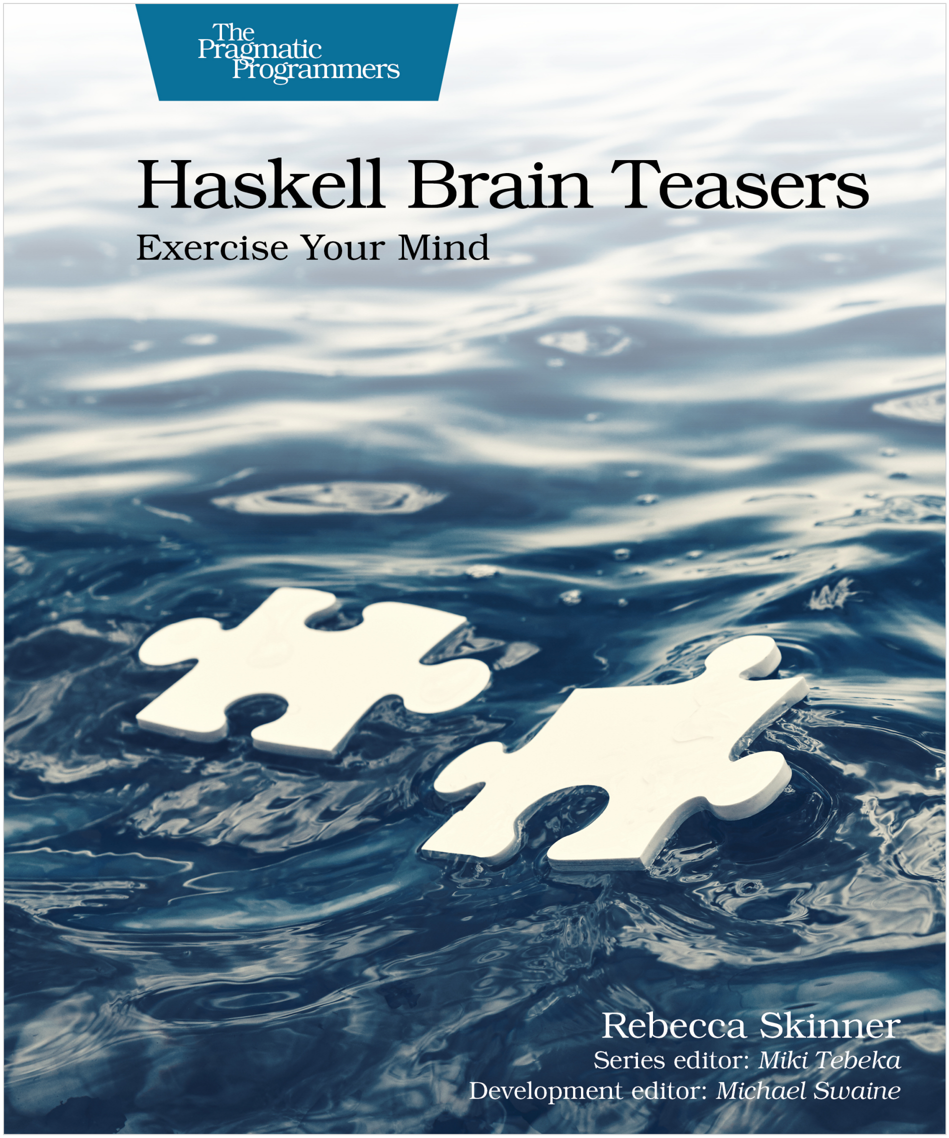
The Pragmatic Programmers

# Haskell Brain Teasers

Exercise Your Mind

Rebecca Skinner

Series editor: *Miki Tebeka*

Development editor: *Michael Swaine*

Puzzle 9

## Many Roads, One Destination

src/HaskellBrainTeasers/Bismuth.hs

```haskell
{-# LANGUAGE OrPatterns #-}
module HaskellBrainTeasers.Bismuth where

data Color = Black|White|Gray|Red|Green|Blue|Yellow|Purple|Orange
  deriving Show

isAchromatic :: Color -> Bool
isAchromatic (Black;White;Gray) = True
isAchromatic (Red;Green;Blue;Yellow;Purple;Orange) = False

main :: IO ()
main = print (isAchromatic Red) >> print (isAchromatic Gray)
```

**Guess the Output**

Before moving on to the next page, try to guess the output. Will it compile? Will it finish running?

```
The program will output
False
True
```

## Discussion

This puzzle demonstrates a newer GHC language extension called *Or Patterns*. Or patterns let you combine several otherwise identical pattern matches into a single larger pattern. Without Or Patterns we would have needed to choose between either a much more verbose implementation of our puzzle, or one that risked misbehaving in the future.

As an example, we could have written a version of isAchromatic that explicitly pattern matched on every color. This would have turned our two pattern matches into nine pattern matches. More likely, we would have written our function using a wildcard to save some typing:

src/HaskellBrainTeasers/Bismuth.hs

```haskell
isAchromaticWildcard :: Color -> Bool
isAchromaticWildcard Black = True
isAchromaticWildcard White = True
isAchromaticWildcard Gray = True
isAchromaticWildcard _ = False
```

This implementation of our function works well today, but it might not always behave as expected in the future. Imagine that we add two new achromatic colors: LightGray and DarkGray. If we were diligent about explicitly pattern matching on every color, the compiler would tell us that we now have an incomplete pattern and point us to the offending function. This is useful, but it means our already long function gets even longer. If, on the other hand, we took a shortcut and use a wildcard then our two new colors would hit the wildcard. The compiler wouldn't warn us, and we'd suddenly find ourselves claiming that neither LightGray nor DarkGray are achromatic.

As you can see in the puzzle, Or Patterns can help us get the best of both worlds by making it easier to explicitly enumerate every case easier and less verbosely.

### Limitations

Although Or Patterns are useful and can let us avoid wildcards or partial patterns, they do have some limitations that we need to work around in order to use them effective. Most notably: Or Patterns do not support binding variables or constraints.

Let's start by taking a look at variable binding. In a normal pattern match we can bind a variable to part of the pattern. For example, working with Either values you can normally bind a variable to the Left or Right value. For example, imagine that we want to write a function that takes a pair pair of Either values and checks to see if they are "mirrors" of one another:

src/HaskellBrainTeasers/Bismuth/MirrorTest.hs
```haskell
isMirrored :: (Either Int Int, Either Int Int) -> Bool
isMirrored (Left x, Right y) = x == y
isMirrored (Right x, Left y) = x == y
isMirrored ((Left _, Left _); (Right _, Right _)) = False
```

In this example, x and y are variables we're binding to the Left and Right values inside of our patterns. We can do this in the example, because the first two patterns we're using are not Or Patterns. In the final pattern we're using an Or Pattern to match the case when we get two left or two right values, but we're not binding a variable in that case so this also works as expected.

Looking at the example, you might notice that our first two patterns both implement the same logic, and consider wanting to combine them into an Or Pattern as well. For example:

src/HaskellBrainTeasers/Bismuth/MirrorTest.hs
```haskell
isMirrored :: (Either Int Int, Either Int Int) -> Bool
isMirrored ((Left x, Right y); (Right x; Left y)) = x == y
isMirrored ((Left {}, Left {}); (Right {}, Right {})) = False
```

Unfortunately this is where we run into the limitation on binding variables in Or Patterns. In this example we've combined our first two patterns into a single Or Pattern, including attempting to bind x and y. Since we need to bind variables in our pattern, we'll have to go back to our earlier example and reserve the Or Pattern only for the fall-through case.

Variable binding is something visible that we do intentionally when we need a value. Although we may wish to bind variables in an Or Pattern, it's clear when we're trying to do so and easy enough to revert back to matching patterns individually. The lack of support for binding constraints in Or Patterns is a little trickier, since this is something the compiler often does for us implicitly. Let's look at an example of some code we might write that works as expected without Or Patterns:

src/HaskellBrainTeasers/Bismuth/ValueTypes.hs
```haskell
{-# LANGUAGE OrPatterns #-}
{-# LANGUAGE GADTs #-}
```

```
module HaskellBrainTeasers.Bismuth.ValueTypes where
import Data.Word

newtype Each = Each Word32 deriving (Num, Show)
newtype Pounds = Pounds Double deriving (Num, Fractional, Show)

data Inventory a where
  Toasters :: Inventory Each
  Iguanas :: Inventory Each
  EngineGrease :: Inventory Pounds
  Antimatter :: Inventory Pounds

inventory :: Inventory a -> a
inventory Toasters = 0
inventory Iguanas = 0
inventory EngineGrease = 0
inventory Antimatter = 0
```

In this example we have an Inventory type that represents different items we might carry in a shop. We can define our store's inventory at a moment in time by providing a function from the inventory item name to the amount that we have in stock. Our current inventory says that we're out of stock on all of our items.

Knowing about Or Patterns, you might notice that since our inventory items are *symbolic* and we don't need to bind a variable, this could be a good opportunity to use Or Patterns. After all, this is exactly the kind of problem where we might want to avoid a wildcard pattern that might not be correct for new items we add to our inventory. Unfortunately, putting everything into a single Or Pattern doesn't work:

src/HaskellBrainTeasers/Bismuth/ValueTypes.hs
```
inventory :: Inventory a -> a
inventory (Toasters; Iguanas; EngineGrease; Antimatter) = 0
```

Unfortunately this won't work. The problem is that the type of value we return depends on the specific constructor we've matched on. We count our inventory by Each Toasters or Ignuanas, but by how many Pounds we have in stock of Engine-Grease or Antimatter. Your first thought might be to use two Or Patterns: one for Each items and one for Pounds items:

src/HaskellBrainTeasers/Bismuth/ValueTypes.hs
```
inventory :: Inventory a -> a
inventory (Toasters; Iguanas) = 0
inventory (EngineGrease; Antimatter) = 0
```

Disappointingly, this still won't work. Normally, when pattern match on a GADT the compiler will generate a constraint that tells us what type we're dealing with. In our working example, when we pattern match on Toasters we introduce a constraint that says the type of a has to be Each. Since Or Patterns don't bind constraints, we lose this information and the compiler can't be sure what we're returning is correct.

Or Patterns are a new feature as of GHC 9.12 and they may become more capable in future GHC versions. Today they can be very useful for functions that need to match on simple sum types with many common codepaths, but their limitations mean that they aren't always a viable choice even when they seem potentially useful.

## Further Reading

*Or Patterns in the GHC user manual*

https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/or_patterns.html

*GHC Proposal 0522: Or Patterns*

https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/or_patterns.html