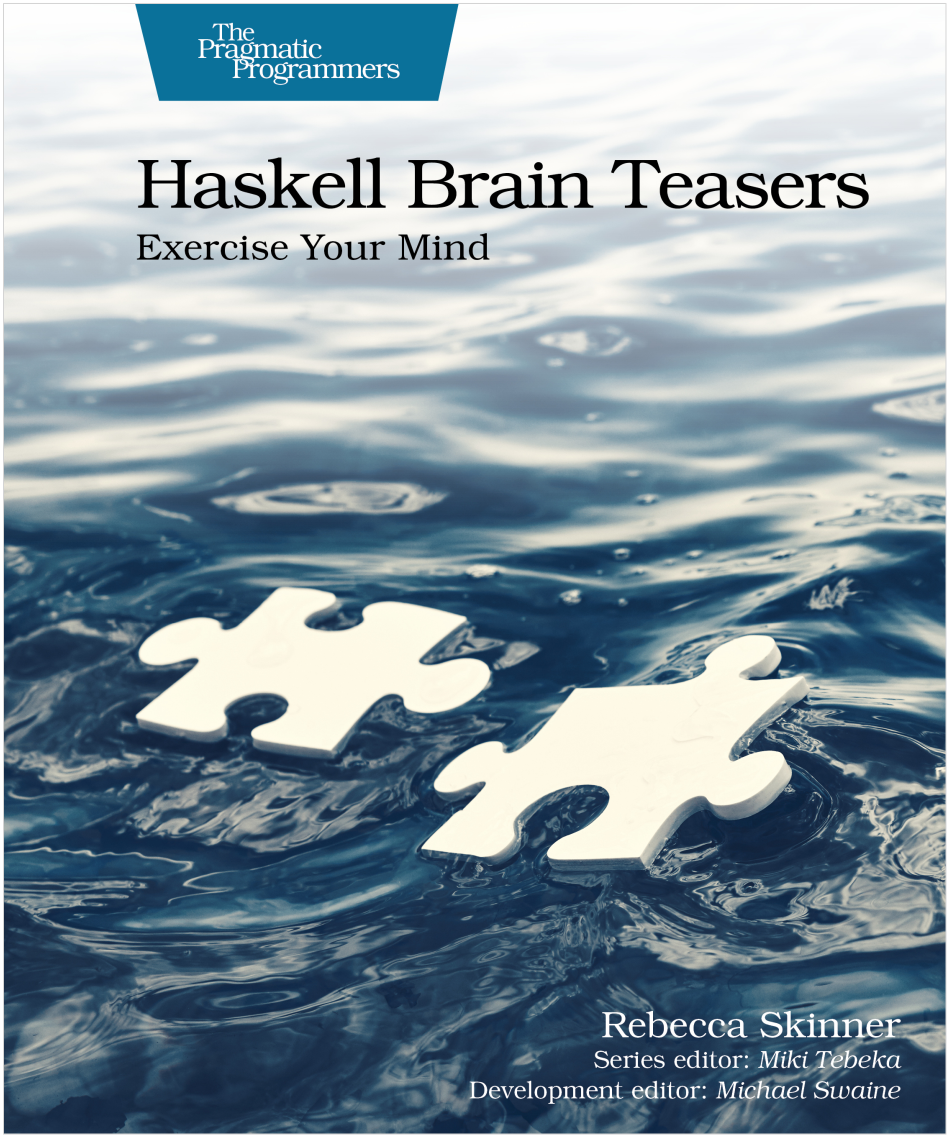

The Pragmatic Programmers

Haskell Brain Teasers

Exercise Your Mind

Rebecca Skinner

Series editor: *Miki Tebeka*

Development editor: *Michael Swaine*

Puzzle 15

Round Trip Tickets

src/HaskellBrainTeasers/RoundTripTicket.hs

```haskell
module HaskellBrainTeasers.RoundTripTicket where
roundTrip ::
  (Show a, Show b) => String -> a -> b -> String
roundTrip maxLenStr a b =
  take maxLen $ show (hither $ thither a, hither $ thither b)
  where
    maxLen = hither maxLenStr
    hither = read
    -- refactored from @thither x = show x@ to avoid hlint
    -- warnings to eta reduce
    thither = show

main :: IO ()
main = putStrLn $ roundTrip "5" 10 10
```

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 fail to compile with an error
/tmp/OhEmDashXXX2728112-1.hs:9:60: error: [GHC-25897]
    " Couldn't match expected type 'a' with actual type 'b'
      'b' is a rigid type variable bound by
        the type signature for:
          roundTrip :: forall a b.
                       (Show a, Show b) =>
                       String -> a -> b -> String
        at /tmp/OhEmDashXXX2728112-1.hs:(6,1)-(7,48)
      'a' is a rigid type variable bound by
        the type signature for:
          roundTrip :: forall a b.
                       (Show a, Show b) =>
                       String -> a -> b -> String
        at /tmp/OhEmDashXXX2728112-1.hs:(6,1)-(7,48)
    " In the first argument of 'thither', namely 'b'
      In the second argument of '($)', namely 'thither b'
      In the expression: hither $ thither b
    " Relevant bindings include
        thither :: a -> String (bound at /tmp/OhEmDashXXX2728112-1.hs:15:5)
        b :: b (bound at /tmp/OhEmDashXXX2728112-1.hs:8:23)
        a :: a (bound at /tmp/OhEmDashXXX2728112-1.hs:8:21)
        roundTrip :: String -> a -> b -> String
          (bound at /tmp/OhEmDashXXX2728112-1.hs:8:1)
  |
9 |   take maxLen $ show (hither $ thither a, hither $ thither b)
  |                                                            ^
```

Discussion

In this puzzle a well-intentioned refactor has caused havoc by summoning the dread monomorphism restriction. The monomorphism restriction, and its close friend the ambiguous type error, are the cause of many *completely safe* refactors that are immediately followed by CI failures. Thankfully, by carefully studying the nuances of the monomorphism restriction and how typeclass resolution works, you can confidently continue to make this exact mistake for the rest of your career.

Let's start by taking a look at what the monomorphism restriction is, and when it applies to your code. Afterwards, we'll look at a few common ways that you can accidentally trip over it in your programs, and what to look for.

What is the Monomorphism Restriction?

The monomorphism restriction prevents GHC from inferring polymorphic types in some cases. Although the specifics of the grammar and when the restriction applies are outlined in the Haskell 2010 Report, the simplest way to understand it is that it applies when you bind variables in let expressions and where clauses. One of the most surprising things about the monomorphism restriction is that it only applies to bindings without arguments to the left of the equals sign. It's much easier to understand in code, so let's look at a few examples:

```
-- monomorphism restriction applies because value is an ordinary binding
let value = somethingCool "George" "Spaceships" in ...

-- monomorphism restriction doesn't apply, because cases aren't simple bindings
case somethingCool "George" "Spaceships" of { value -> ... }

-- monomorphism restriction applies, increment is a simple binding of a function
let increment = (+1)
let increment = \n -> 1 + n

-- monomorphism restriction doesn't apply, there is an explicit type annotation
let
  increment :: Num a => a -> a
  increment = (+1)

-- monomorphism restriction doesn't apply, arguments appear left of =
let increment n = 1 + n
let increment _unused = \n -> 1 + n
```

Notably, the monomorphism restriction means that eta reduction, or eta expansion (fancy people speak for adding or removing arguments) can change whether your program compiles. For example:

```
-- monomorphism restriction applies
let display = show
let parse = read

-- monomorphism restriction doesn't apply
let display val = show val
let parse input = read input
```

Somewhat less obviously, the monomorphism restriction is also disabled when you turn on the NoMonomorphismRestriction language extension.

Why is it a Problem?

The most subtle problem with the monomorphism restriction is that innocuous and seemingly stylistic refactors can cause your program to suddenly fail to compile. If you do somehow notice that your program has failed to compile, you may also notice that the errors have an unfortunate tendency to say nothing about the monomorphism restriction. Instead, the monomorphism restriction tends to cause type unification and ambiguous type errors.

Type unification errors most often occur when we eta reduce and suddenly find out that our newly pointfree code has chosen violence. For example:

```
showPair :: (Show a, Show b) => a -> b -> (String, String)
showPair a b = (display a, display b)
  where display = ("displays as: " <>) . show
```

Since display has no arguments to the left of the equals sign the monomorphism restriction kicks in and says that we have to pick a single concrete type. We call it with a first, so the compiler infers the type display :: a -> String. When we try to call display b the compiler helpfully lets us know that it can't prove that a is equal to b. If we rewrite this in a less pointless style with an argument on the left hand side, then the monomorphism restriction no longer applies and the code works as expected:

```
where display x = "displays as: " <> show x
```

Programmers, of course, hate nothing more than consistency. As you might expect, eta expansion can also confound the monomorphism restriction. Instead of type equality errors, eta expansion tends to result in ambiguous type errors. For example:

```
showMore :: (Monoid a, Show a) => a -> String
showMore a = display a <> display mempty
  where display x = show x
```

In this example, having faithfully learned the lesson that we should always eta expand, we've created a pointful version of display with an inferred type of display :: Show a => a -> String. Since the function is polymorphic each time we call it the compiler will try to pick out the appropriate type. Our first call is at type a, which has a Show instance and so everything goes as expected. Our second call passes in mempty, but that could be *any* type with a Monoid instance. Since display is polymorphic our earlier call does nothing to narrow down which type we're trying to use and the compiler eventually gives up and asks for help. A quick and apparently not so pointless pointfree refactor and the monomorphism restriction kicks in and saves our program:

```
where display = show
```

Real World Tips and Tricks

The examples of eta expansion and reduction we've seen aren't entirely contrived, these sorts of problems happen all the time during real world refactors. One of the most common scenarios is when you factor something out of a let or where binding to the top level and add an argument to pass in a value that had previously been in scope. That argument disables the monomorphism restriction and suddenly you find yourself with bonus type errors.

There are, thankfully, better ways of addressing this problem than stochastically permuting between pointfree and pointful styles. Visible type applications, for example, can neatly solve the problem of ambiguity:

```
showMore :: forall a. (Monoid a, Show a) => a -> String
showMore a = display a <> display (mempty @a)
  where display x = show x
```

Further Reading

The Monomorphism Restriction in The Haskell 2010 Report : https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-930004.5.5

The Monomorphism Restriction on the Haskell Wiki : https://wiki.haskell.org/index.php?title=Monomorphism_restriction

The MonoLocalBinds Extension : https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/let_generalisation.html

The NoMonomorphismRestriction Extension : https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/monomorphism.html#extension-MonomorphismRestriction