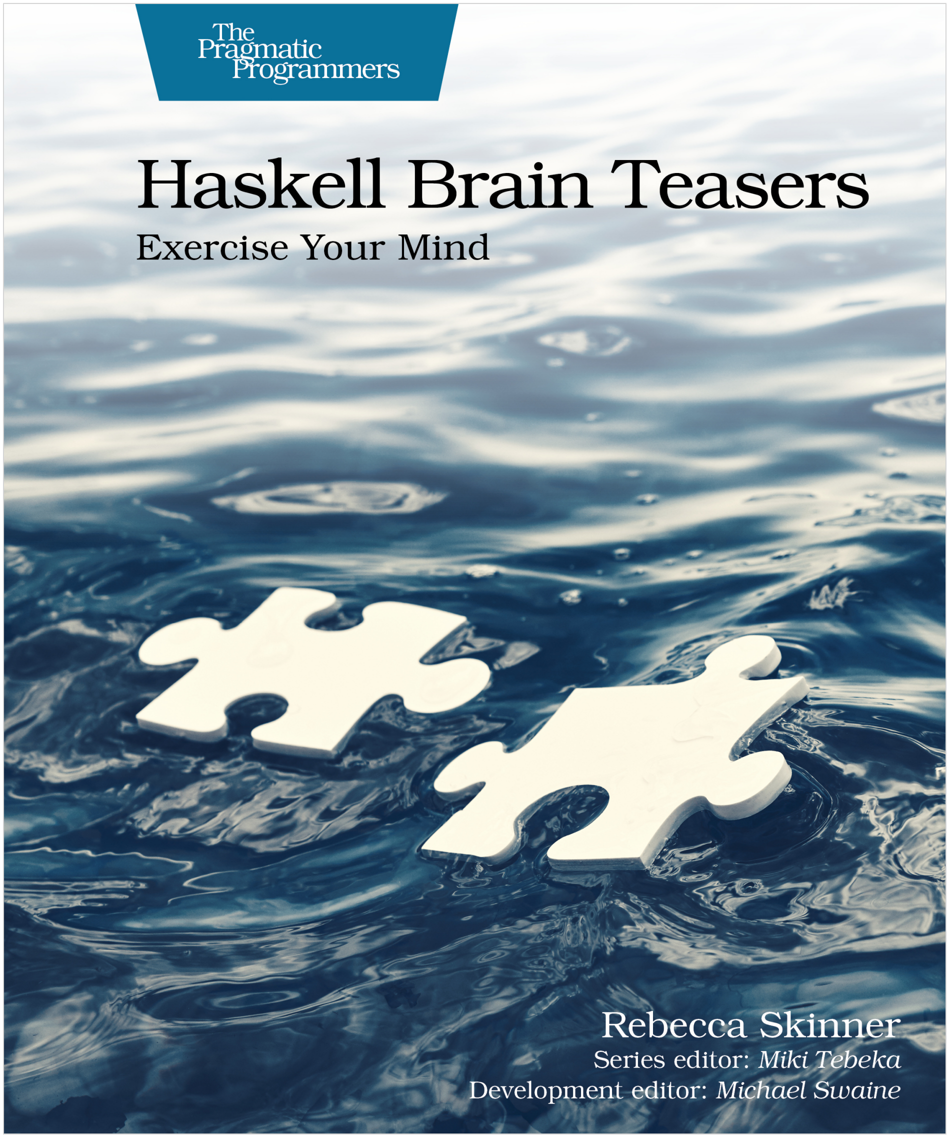
The Pragmatic Programmers

Haskell Brain Teasers

Exercise Your Mind

Rebecca Skinner

Series editor: *Miki Tebeka*

Development editor: *Michael Swaine*

Puzzle 6

Do You Even Lift?

src/HaskellBrainTeasers/UnsafeToTheMax.hs

```
module HaskellBrainTeasers.UnsafeToTheMax where
import System.IO.Unsafe
import Data.IORef
import Data.Foldable
import Control.Monad

unsafeMax :: [Int] -> IO Int
unsafeMax vals = do
  for_ vals $ \val -> do
    currentMax <- readIORef maxRef
    when (val > currentMax) $
      writeIORef maxRef val
  readIORef maxRef
  where
    maxRef = unsafePerformIO $ newIORef 0

main :: IO ()
main = do
  four <- unsafeMax [4,3,2]
  three <- unsafeMax [3,2,1]
  two <- unsafeMax [2,1,0]
  print [four, three, two]
```

Guess the Output

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

```
When built with optimizations, the program outputs
[4,4,4]

When built without optimizations, the program outputs
[4,3,2]
```

Discussion

This puzzle illustrates the dangers of unsafePerformIO. Although the program typechecks, its runtime behavior depends on how it's built. To fix the puzzle we'll need to understand how unsafePerformIO works, when it doesn't, and how to guard against things that might cause our program to behave unpredictably.

Impure Haskell

Although Haskell has a reputation for strictly enforcing *pure* functional programming, there are escape hatches that allow you to evaluate IO actions in pure code. These functions aren't just a novelty, unsafePerformIO and its cousins unsafeDupablePerformIO and unsafeInterleaveIO solve real problems, and they can be used to write safe and easy to use code. Even the amusingly named accursedUnutterablePerformIO has its place as more than simply being a trap for the incautious developer. Still, useful or not, this puzzle demonstrates that these functions have rightfully earned the *unsafe* part of their names.

To use unsafePerformIO safely you need to ensure that:

- The IO action doesn't interact with the external environment
- The call can't be moved elsewhere by the compiler
- You don't use unsafePerformIO to get a polymorphic value

Avoiding the External Environment

unsafePerformIO is best used for code that's still pure from the perspective of any consumers who use it. If you're careful, you can use unsafePerformIO to safely acquire and initialize resources from the system. For example, creating an IORef or MVar. Since these resources are created on demand and not shared, creating a new one isn't a *visible* side effect anywhere else in your code.

The difference between what's internal to your program and what's part of the external environment often comes down to your personal judgement, and risk tolerance, rather than clear objective guidelines. Reading or writing to ordinary files is risky because anything else inside or outside of your program could change or delete a file at any time, and the exact time that you'll read the contents of the file are non-deterministic. Initializing your program with the contents of a read-only file is probably safer, but still not guaranteed to work in the presence of a user with sudo access and a grudge. Opening a new log file at program startup is safer still, unless the filesystem has disappeared or our sudoer got chmod happy. unsafePerformIO is, in the end, not safe and it's up to you as a developer to decide when to use it.

Avoiding Relocation

One of the biggest risks when using unsafePerformIO is that our programs might behave unexpectedly because of optimizations GHC makes that are safe for pure functions, but break down when we introduce side effects. This usually happens because GHC has changed when or how often side effects get evaluated. There are three causes for this: inlining, let floating, and common subexpression elimination (cse).

When GHC inlines code, you'll run side effects more often than anticipated. Instead of running an IO action once to allocate and initialize a resource, your program will create a new resource each time the value is referenced. In common cases, like using unsafePerformIO to create an IORef or an MVar it can look like values aren't being set when they should be. What's actually happening is that each time you reference the IORef or MVar you're getting a newly initialized value instead of a shared value that you can write to and read from the way you expect.

You can disable inlining with the NOINLINE pragma. This tells GHC that it shouldn't inline an expression even when its heuristics might otherwise tell it that it should. You should almost always use NOINLINE anytime you use unsafePerformIO with a top-level identifier.

Where inlining can cause your side effects to be run more often than intended, *let floating* can cause them to be run less often than you intend. GHC uses let floating to move expressions up to a higher scope when it notices that they could be efficiently re-used. If an MVar or IORef is relocated because of let floating it may look like they are being initialized with an incorrect value. This happens because you're getting a shared reference with whatever value happened to be last written instead of a newly initialized reference. Let floating can be hard to predict, and you can't disable it on a per-function basis, but it's a good practice to disable it in any module that uses unsafePerformIO by adding {-# OPTIONS_GHC -fno-full-laziness #-} to the top of your module declaration.

The last optimization we'll look at that could break unsafePerformIO is called *common subexpression elimination*, or CSE. This happens when the compiler recognizes that you have the same expression being used multiple times and it combines them. The practical impact is that you run your side effects less often than anticipated, similar to let floating. GHC only does CSE in a few specific circumstances, but it can be hard to predict. To avoid potential errors you should add -fno-cse to the OPTIONS_GHC for your module, along side -fno-full-laziness.

Avoiding Polymorphic Values

The last problem we need to be aware of when using unsafePerformIO is that it can let us write type unsafe code. Without unsafePerformIO for example, we're never able to get a hold of an IORef value where the type of value being referenced can't be inferred by the compiler. Using unsafePerformIO we're able to get hold of an actual polymorphic IORef value that could hold any type. This allows us to write an unsafe coercion function that we can use to turn a value of any type into a value of any other type, potentially resulting in a runtime crash:

src/HaskellBrainTeasers/UnsafeToTheMax/UnsafeCoerce.hs

```haskell
{-# NOINLINE ref #-}
ref :: IORef a
ref = unsafePerformIO $ newIORef undefined

unsafeCoerce :: a -> b
unsafeCoerce from = unsafePerformIO $
  writeIORef ref from >> readIORef ref
```

Further Reading

INLINE pragma in the GHC Users Guide
: https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/pragmas.html#inline-pragma

unsafePerformIO on Hackage
: https://hackage.haskell.org/package/base-4.21.0.0/docs/System-IO-Unsafe.html#v:unsafePerformIO

accursedUnutterablePerformIO Documentation
: https://hackage.haskell.org/package/bytestring-0.12.2.0/docs/Data-ByteString-Internal.html#v:accursedUnutterablePerformIO

Referential Transparency on the Haskell Wiki
: https://wiki.haskell.org/index.php?title=Referential_transparency