# Unsafe Haskell

CIS 194 Week 12
15 April 2015

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE TupleSections     #-}
import Control.Applicative
import Control.Monad
import Data.ByteString (ByteString)
import Data.IORef
import Data.List
import Data.Word
import Foreign.Marshal.Alloc
import Foreign.Marshal.Array
import Foreign.Ptr
import Foreign.Storable
import System.IO.Unsafe
import Unsafe.Coerce

import qualified Data.ByteString as BS
```

Think back to the first day of class. We learned that there are four things that characterize the Haskell language. Namely, Haskell is:

**Functional**

- Functions are first class values
- Evaulation is based on expressions rather than statements

**Pure**

- No mutable state
- No side effects
- Computations are repeatable

**Lazy**

- Expressions are not evaluated until they are needed
- Functions use call-by-name semantics

**Statically Typed**

- No runtime type errors
- The type of every expression is known *statically* (at compile time)

However, we saw later on that this is not entirely true. We can use the `IO` Monad to write programs that have side effects, and we can use `seq` and bang patterns to force expressions to be strictly evaluated. However, the lies continue…

## Mutable State

It isn't too difficult to get mutable state in Haskell. To work with mutable state, we introduce a new type called `IORef`. An `IORef` is basically just a reference to some mutable cell in memory. The value in the IO Ref can be manipulated using a few simple functions in the `Data.IORef` module:

```
-- Initialize an IO Ref
newIORef    :: a -> IO (IORef a)

-- Read from an IO Ref
readIORef   :: IORef a -> IO a

-- Write to an IO Ref
writeIORef  :: IORef a -> a -> IO ()

-- Update the value of an IO Ref
modifyIORef :: IORef a -> (a -> a) -> IO ()
```

Of course, IO Refs live exclusively in the `IO` Monad since modifying mutable state is a side effect. It should not come as a huge surprise that mutable state is possible in the IO Monad. Of course, you could implement mutable state yourself using operations that we already know about. You could use a file to store mutable state and use `readFile` and `writeFile` to get and modify the state. However, this would be incredibly inefficient. If you absolutely need mutable state in Haskell (you probably don't) then IO Refs are a much better way to go.

Now, let's take a look at few examples of how you might want to use IO Refs. First, you are given a list of `a`s and an element of type `a`, and you want to figure out how many times the element appears in the list. Let's try to do this using an IO Ref:

```
count :: Eq a => [a] -> a -> IO Int
count xs x = do
  r <- newIORef 0
  forM_ xs $ \y ->
    when (x == y) $
      modifyIORef r (+1)
  readIORef r
```

Basically, `count` just initializes an IO Ref with the value `0` and then for each element in the input list, it increments the value inside the IO Ref. There are a few things wrong with this implementation. For one, it looks very… *imperative*. It is not very elegant compared to the wholemeal style version:

```
count2 :: Eq a => [a] -> a -> Int
count2 xs x = length $ findIndices (==x) xs
```

But style aside, the first implementation of `count` is still not very good. It is pretty cumbersome and annoying that `count` returns its result in the `IO` Monad. This counting operation really should be *pure*, since it does not have any global side effects. To make it pure, we can use the `unsafePerformIO` function. Use of `unsafePerformIO` is generally discouraged, but is *allowable* in cases where the `IO` operation is definitely safe. Using IO Refs locally is usually safe, so we can *reluctantly* use `unsafePerformIO` here:

```
count3 :: Eq a => [a] -> a -> Int
count3 xs x = unsafePerformIO $ do
  r <- newIORef 0
  forM_ xs $ \y ->
    when (x == y) $
      modifyIORef' r (+1)
  readIORef r
```

Now we have a function that operates as we want it to, and is implemented with an IO Ref under the hood. The question we want to ask is: is it desirable to write functions in this way?

From a style standpoint, the answer is absolutely not. This code is ugly and makes use of `unsafePerformIO`, which is dubious at best. However, if style is not reason enough to convince you that this implementation is bad, then consider performance. It turns out that this code is more than 7x slower than the pure version. The reason for this is that GHC is optimized for wholemeal style programming. IO Refs cannot be implemented in pure Haskell, therefore the runtime system has a lot more work to do in order to execute this code.

So when would it actually be useful to use mutable state? Consider the following scenario: you have a tree structure an you want to assign unique identifiers to each node in the tree.

```
data Tree a = Node (Tree a) a (Tree a)
            | Empty
              deriving (Show)
```

We could do this in pure Haskell, but it is a bit annoying since we need to know always keep track of how many identifiers we have used so far.

```
assignIDs :: Tree a -> Tree (Int, a)
assignIDs = snd . assignIDsHelper 0
    where assignIDsHelper id Empty        = (id, Empty)
          assignIDsHelper id (Node l x r) = (id2, Node l' (id1, x) r')
              where (id1, l') = assignIDsHelper id l
                    (id2, r') = assignIDsHelper (id1 + 1) r
```

Instead, it would be nice to have a *mutable* counter that can keep track of how many IDs have been assigned so far. We can implement this counter using an IO Ref:

```
newCounter :: IO (IO Int)
newCounter = do
  r <- newIORef 0
  return $ do
    v <- readIORef r
    writeIORef r (v + 1)
    return v
```

The type of `newCounter` is a bit strange. `newCounter` is an IO action that initializes another IO action which increments an internal counter and returns the resulting value. Consider the following IO action:

```
printCounts :: IO ()
printCounts = do
  c <- newCounter
  print =<< c
  print =<< c
  print =<< c
```

When we evaulate this action, we get:

```
*> printCounts
0
1
2
```

Each time the action `c` is evaluated, the internal counter in incremented, so the next time it is used, it has a greater value. Now, we can use this to rewrite the `assignIDs` function:

```
assignIDs2 :: Tree a -> IO (Tree (Int, a))
assignIDs2 t = do
  c <- newCounter
  let helper Empty = return Empty
      helper (Node l x r) = do
        l' <- helper l
        id <- c
        r' <- helper r
        return $ Node l' (id, x) r'
  helper t
```

Alternatively, we could write it using Applicative Functors, since the value on the counter does not effect the control flow of the program:

```
assignIDs3 :: Tree a -> IO (Tree (Int, a))
assignIDs3 t = do
  c <- newCounter
  let helper Empty = return Empty
      helper (Node l x r) = Node
                            <$> helper l
                            <*> ( (,x) <$> c )
                            <*> helper r
  helper t
```

## Type Safety

The phrase "type safety" gets thrown around a lot, especially in the world of functional programming. You can ask two type programming language theorists what type safety is and you will get five different opinions. However, most people agree that type safe languages prevent *type errors*. This means that you can't simply use a value of one type as a value of a different type.

**Enter Unsafe Coerce**

Haskell is *not* type safe. There is a handy little function in the `Unsafe.Coerce` module called `unsafeCoerce` that allows you to do the unthinkable; use a value of one type as a value of a different type. Consider this simple example:

```
*> let x = 1 :: Float
*> let y = unsafeCoerce x :: Int
*> y
5360320512
```

Something strange happened, but it was not a catastrophic failure. Why is the value of `y` `5360320512`? Well, first we defined `x` to be the floating point number `1`, then we coerced it to be an `Int`. Floating point number are encoded very differently in binary than integers are. It just so happens that the floating point representation of `1` is equivalent to the integer representation of `5360320512`. Who would've thought?

One thing to note is that nothing went horribly wrong in this example because `Float`s and `Int`s are both encoded using 32 bits. This means that GHCi has no trouble accessing `x` as an `Int` instead of a `Float`, since the size of the chunk of memory it was looking in was correct. Let's see what happens when we try to access a datatype that is *larger* than the memory chunk we are looking in:

```
*> let x = 1 :: Int
*> let y = unsafeCoerce x :: (Int, Int)
*> y
Segmentation fault: 11
```

Well that's not good. We crashed GHCi using only *pure* code. Clearly this is not the behavior you would expect from a type safe language.

**Memory Safety**

Memory safe languages are languages that do not allow arbitrary access to the underlying bit patterns in memory. This is tightly linked to type safety since being able to manipulate individual bits can allow programmers to *abuse* the type system. It also means that the programmer can attempt to de-reference arbitrary memory locations. Memory is managed by the operating system and if you try to access memory that the operating system has not allocated to you, you will get a segmentation fault. Clearly Haskell isn't memory safe since we got a segfault in the example above. But it gets worse.

Let's think about everyone's favorite *unsafe* language, C. C has a wonderful type system; everything is just a sequence of bits. This includes pointers to memory locations. In C, you can take an arbitrary sequence of bits, treat it as a memory address and attempt to grab whatever value is stored there. This is just about the most unsafe thing you can do, but also allows programmers to write low level and high performance code. This is why people choose C for systems programming.

Luckily, Haskell has pointer manipulation too! Here is an excellent way to generate a segfault in a hurry:

```
*> peek nullPtr
Segmentation fault: 11
```

This is exactly equivalent to writing `*null` in C. When you try to dereference the null pointer, you will always get a segfault since the null pointer is never a valid memory location.

Haskell also has a number of facilities to allocate memory buffers on the stack and the heap. There is only one logical next step…

**Smashing the Stack for Fun and Profit**

Consider the following *extremely secure* login system. The users are stored in a list of username, password pairs.

```
users :: [(ByteString, ByteString)]
users =  [ ("spj",     "i-<3-GHC")
         , ("hcurry", "programsareproofs")
         ]

getPwd :: ByteString -> Maybe ByteString
getPwd user = snd <$> find ((== user) . fst) users
```

The person who implemented this wanted to be *smart* about password comparisons. Rather than simply using Haskell's built in equality operator `(==)`, a 16-byte memory buffer is allocated on the stack and then the two `ByteStrings` are copied in to it. The first `ByteString` occupies bytes 0-15 and the second one is in bytes 16-31 (because passwords can't be longer than 16 characters, right?)

```
checkPwd :: ByteString -> ByteString -> IO Bool
checkPwd s1 s2 =
  allocaArray 32 $ \b1 -> do
    let l1 = BS.length s1
        l2 = BS.length s2
        b2 = plusPtr b1 16
    pokeArray b2 (BS.unpack s2)
    when (l2 < 16) $ pokeArray (plusPtr b2 l2) (replicate (16 - l2) (0 :: Word8))
    pokeArray b1 (BS.unpack s1)
    when (l1 < 16) $ pokeArray (plusPtr b1 l1) (replicate (16 - l1) (0 :: Word8))
    and <$> forM [0..15] (\i -> do
      c1 <- peekByteOff b1 i :: IO Word8
      c2 <- peekByteOff b2 i
      return $ c1 == c2)
```

To login, you simply type your username and password. The password for that username then gets looked up in the `users` table and compared to what you typed. If there is a match, then you are logged in!

```
login :: IO ()
login = do
  putStr "Username: "
  user <- BS.getLine
  putStr "Password: "
  guess <- BS.getLine
  case getPwd user of
    Just actual -> do
      match <- checkPwd guess actual
      if match then
          putStrLn "You are logged in!"
      else
          putStrLn "Incorrect password"
    Nothing ->
        putStrLn "No such user"
```

Now, knowing about the questionable way that `checkPwd` was written, how can we exploit this code?

This login system is actually vulnerable to a very simple buffer overflow attack. Let's think about how the passwords are compared. First, the exprected password is copied in to the second half of some 32-byte memory buffer. Next, whatever the user typed is copied in to the first half of the buffer.

The key vulnerability here is that the length of the input is not checked before it is copied in. This means that is the string that the user types is longer than 16 characters, then it will overflow into the part of the buffer that is reserved for the expected password. This means that if the user types a 32 character password in which the first 16 and last 16 bytes match, then he/she will always get logged in! Let's try it out:

```
*> login
Username: hcurry
Password: WriteSafeHaskellWriteSafeHaskell
You are logged in!
```

Type safe languages are usually not prone to buffer overflow attacks. Unfortunately, Haskell isn't so lucky!

---

```
Generated 2015-04-17 09:25:02.845221
```