Evaluation order and state tokens

From HaskellWiki

Let's start off with a little quiz. What will be the output of running the following program?

```
import System.IO.Unsafe

foo = unsafePerformIO $ putStrLn "foo"
bar = unsafePerformIO $ do
    putStrLn "bar"
    return "baz"
main = putStrLn $ foo `seq` bar
```

The answer is: it's undefined. If you want to understand why, keep reading.

Contents

1 Evaluation order basics

NOTE: When compiling or running your code without optimizations turned on, you will likely *never* see the "surprising" behavior described in this document. Furthermore, you won't reliably see such behavior even if you compile with `-O2`; that's what I mean by "undefined behavior".

Let's get back to something simpler. What's the output of this program?

```
helper i = print i >> return i

main = do
    one <- helper 1
    two <- helper 2
    print $ one + two
```

I think most people would agree: it will print 1, then 2, then 3. Now let's tweak this a little bit:

```
import System.IO.Unsafe

helper i = unsafePerformIO $ print i >> return i

main = do
    let one = helper 1
        two = helper 2
    print $ one + two
```

In this case, it's pretty easy to see that 3 will have to be printed after both 1 and 2, but it's unclear whether 1 or 2 will be printed first. The reason for this is evaluation order: in order to evaluate `one + two`, we'll need to evaluate both `one` and `two`. But there is nothing telling GHC which of those thunks should be evaluated first, and therefore GHC is at full liberty to choose whichever thunk to evaluate first. Now let's make it a bit more complicated:

```
import System.IO.Unsafe

helper i = unsafePerformIO $ print i >> return i

main = do
    let one = helper 1
        two = helper 2
    print $ one `seq` one + two
```

Now we've forced evaluation of `one` before evaluating the `one + two` expression, so presumably we should always print 1, then 2, then 3. But in fact, that's not true. To quote the docs for `seq`:

> A note on evaluation order: the expression `seq a b` does *not* guarantee that `a` will be evaluated before `b`. The only guarantee given by `seq` is that the both `a` and `b` will be evaluated before `seq` returns a value. In particular, this means that `b` may be evaluated before `a`.

(Note: this version of the docs has not yet been publicly released as of writing this document.)

In other words, `seq` ensures that both `one` and `one + two` will be evaluated before the result of the `seq` expression is returned, but we *still* don't know which one will be evaluated first. If we want to be certain about that ordering, we need to instead use `pseq`. To quote its docs:

> Semantically identical to `seq`, but with a subtle operational difference: `seq` is strict in both its arguments, so the compiler may, for example, rearrange ``a `seq` b`` into ``b `seq` a `seq` b``.

To see this in action:

```
import System.IO.Unsafe
import Control.Parallel

helper i = unsafePerformIO $ print i >> return i

main = do
    let one = helper 1
        two = helper 2
    print $ one `pseq` one + two
```

Notice that comment about being strict in its arguments. You might think (as I did) that we can get the same guaranteed ordering of evaluation by having a function which is only strict in one of its arguments:

```
{-# LANGUAGE BangPatterns #-}
import System.IO.Unsafe

helper i = unsafePerformIO $ print i >> return i

add !x y = x + y

main = do
    let one = helper 1
        two = helper 2
    print $ add one two
```

However, that's not the case: GHC is free to inline the definition of `add`. And if we have `+` on a typical numeric type (like `Int`), which is strict in both arguments, we'll be right back where we started with both arguments being strictly evaluated. The only way to guarantee ordering of evaluation in this case is with `pseq`.

Alright, one more higher-level twist. What do you think about this?

```
import System.IO.Unsafe

helper i = print i >> return i

main = do
    one <- helper 1
    let two = unsafePerformIO $ helper 2
    print $ one + two
```

This looks like it *should* be straightforward:

1. Run `helper 1`.
2. Create a thunk to run `helper 2`.
3. Evaluate `one + two`, forcing the `helper 2` thunk to be evaluated in the process.
4. Print the result of `one + two`, a.k.a. 3.

However, this isn't guaranteed! GHC is allowed to rearrange evaluation of thunks however it wishes. So a perfectly valid sequence of events for GHC with this code is:

1. Create and evaluate the `helper 2` thunk.
2. Run `helper 1`.
3. Print the result of `one + two`.

And this would result in the output of 2, then 1, then 3. You might think you can work around this with the following:

```
import System.IO.Unsafe

helper i = print i >> return i

main = do
    one <- helper 1
    two <- return $ unsafePerformIO $ helper 2
    print $ one + two
```

However, this makes absolutely no difference! The `helper 2` thunk can still be reordered to before the `helper 1` call. The following, however, *does* ensure that the evaluation of `two` always occurs after `helper 1`:

```
import System.IO.Unsafe

helper i = print i >> return i

main = do
    one <- helper 1
    two <- unsafeInterleaveIO $ helper 2
    print $ one + two
```

The reason is that `unsafeInterleaveIO` provides an extra guarantee relative to `unsafePerformIO`. Namely, with the code:

```
do
    before
    unsafeInterleaveIO side
    after
```

We are guaranteed that effects in `side` will *always* happen after effects in `before`. However, effects in `side` may still occur interleaved with effects in `after`.

To understand how `unsafeInterleaveIO` provides these guarantees as opposed to `return . unsafePerformIO`, we need to drop down a layer of abstraction.

2 State tokens

When we have the code `print 1 >> print 2`, we know that 1 will be printed before 2. That's a feature of the IO monad: guaranteed ordering of effects. However, the second `print` call does not depend on the result of the first `print` call, so how do we know that GHC won't rearrange the `print` calls? The real answer is that our assumption was wrong: the result of `print 1` *is* in fact used by `print 2`. To see how, we need to look at the definition of `IO`:

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))

instance  Monad IO  where
    (>>=)     = bindIO

bindIO :: IO a -> (a -> IO b) -> IO b
bindIO (IO m) k = IO $ \ s -> case m s of (# new_s, a #) -> unIO (k a) new_s
```

Unwrapping the `newtype` and replacing the slightly unwieldy `State# RealWorld` with `S#`, we can see that the type of `print 1` is:

```
print 1 :: S# -> (# S#, () #)
```

So in fact, our `print 1` call produces *two* results: a new state token, and a unit value. If we inline the definition of `>>`, our `print 1 >> print 2` turns into:

```
\s0 ->
    case print 1 s0 of
        (# s1, _ignored #) -> print 2 s1
```

In fact, our call to `print 2` *does* rely on the result of `print 1`, in particular, the newly generated state token. This is the exact mechanism by which we ensure ordering of actions in both the `IO` and `ST` monads.

When you have a `main :: IO ()` function, GHC will generate a brand new state token at the start of the program, pass it in to the main function, and then throw away the state token generated at the end. By having these state tokens threaded through your program, we have a strict ordering of every single `IO` action in our program.

NB: In reality, GHC doesn't actually have any state tokens at runtime, they're a purely compile-time construct. So there's no actual "generating" and "throwing away."

And this is where the magic of `unsafePerformIO` comes into play. It does the same thing as GHC does with `main`, but with a subprogram instead. This in theory looks something like the following (though in practice is more complicated, we'll get to that later):

```
unsafePerformIO (IO f) =
    case f fakeStateToken of
        (# _ignoredStateToken, result #) -> result
```

With normal, safe `IO` code, running of the action is forced by evaluating the state token passed to the next `IO` action. But when we use `unsafePerformIO`, the state token is completely ignored. Therefore, we're only left with the `result` value, and evaluating that forces the action to be run.

NB: The real name for `fakeStateToken` is `realWorld#`, we'll use that from now on.

As a reminder, we had this somewhat surprising code above:

```
import System.IO.Unsafe

helper i = print i >> return i

main = do
    one <- helper 1
    let two = unsafePerformIO $ helper 2
    print $ one + two
```

I said that it was possible for 2 to be printed before 1. Understanding state tokens better, let's see why. A liberal translation of this code would be:

```
main s0 =
    case helper 1 s0 of
        (# s1, one #) ->
            case helper 2 realWorld# of
                (# _ignored, two #) ->
                    print (one + two) s1
```

But using normal Haskell reasoning, it's perfectly sane for me to rewrite that and change the ordering of the case expressions, since the results of the first expression are never used by the second or vice versa:

```
main s0 =
    case helper 2 realWorld# of
        (# _ignored, two #) ->
            case helper 1 s0 of
                (# s1, one #) ->
                    print (one + two) s1
```

I also above said that `unsafeInterleaveIO` would fix this ordering issue. To understand why *that's* the case, let's look at a simplified implementation of that function:

```
unsafeInterleaveIO (IO f) = IO $ \s0 ->
    case f s0 of
```

```
        (# _ignored, result #) ->
            (# s0, result #)
```

Like `unsafePerformIO`, `unsafeInterleaveIO` throws away its resulting state token, so that the only way to force evaluation of the thunk is by using its result, not its state token. *However*, `unsafeInterleaveIO` does *not* conjure a new state token out of thin air. Instead, it takes the state token from the current `IO` context. This means that, when you create a thunk with `unsafeInterleaveIO`, you are guaranteed that it will only be evaluated after all previous `IO` actions are run. To do a similar rewrite of our `unsafeIntereleaveIO` example from above:

```
main s0 =
    case helper 1 s0 of
        (# s1, one #) ->
            case helper 2 s1 of
                (# _ignored, two #) ->
                    print (one + two) s1
```

As you can see, it would no longer be legal to swap around the case expressions, since `helper 2 s1` depends on the result of `helper 1 s0`.

One final note about `unsafeInterleaveIO`: while it *does* force evaluation to occur after previous `IO` actions, it says nothing about actions that come later. To drive that home, the ordering in the following example is undefined:

```
import System.IO.Unsafe

helper i = print i >> return i

main = do
    one <- unsafeInterleaveIO $ helper 1
    two <- unsafeInterleaveIO $ helper 2
    print $ one + two
```

I encourage you to do a similar rewriting to case expressions as I did above to prove to yourself that either 1 or 2 may be printed first.

3 unsafePerformIO, unsafeDupablePerformIO, inlinePerformIO (aka he-who-shall-not-be-named)

To sum up: we now understand the difference between `unsafePerformIO` and `unsafeInterleaveIO`, the difference between `seq` and `pseq`, how state tokens force evaluation order, and the difference between running `IO` actions safely and evaluating unsafe thunks. There's one question left: why are there three different functions with the type signatures `IO a -> a`, namely: `unsafePerformIO`, `unsafeDupablePerformIO`, and `inlinePerformIO` (provided by `Data.ByteString.Internal`, also known as `unsafeInlineIO` (http://hackage.haskell.org/package/primitive-0.5.3.0/docs/Control-Monad-Primitive.html#v:unsafeInlineIO) , also known as `accursedUnutterablePerformIO` (http://www.reddit.com/r/haskell/comments/2cbgpz/flee_traveller_flee_or_you_will_be_corrupted_and/)).

What we were looking at previously is actually the implementation of `inlinePerformIO`, or to provide the actual code from bytestring (https://github.com/haskell/bytestring/blob/a562ab285eb8e9ffd51de104f88389ac125aa833/Data/ByteString/Internal.hs#L624) :

```
{-# INLINE accursedUnutterablePerformIO #-}
accursedUnutterablePerformIO :: IO a -> a
accursedUnutterablePerformIO (IO m) = case m realWorld# of (# _, r #) -> r
```

This looks straightforward, so why wouldn't we want to just use this in all cases? Let's consider a usage of this:

```
import Data.ByteString.Internal (inlinePerformIO)
import qualified Data.Vector as V
import qualified Data.Vector.Mutable as VM

vectorA = inlinePerformIO $ do
    mv <- VM.new 1
    VM.write mv 0 'A'
    V.unsafeFreeze mv

vectorB = inlinePerformIO $ do
    mv <- VM.new 1
    VM.write mv 0 'B'
    V.unsafeFreeze mv

main = do
    print vectorA
    print vectorB
```

When evaluating the `vectorA` thunk, we want to:

1. Create a new mutable vector.
2. Write 'A' into the first element of the vector.
3. Freeze the vector.

We then do the same thing with `vectorB`, writing 'B' instead. The goal should be getting two separate, immutable vectors, one containing "A", the other "B".

However, this may not be the case! In particular, both `vectorA` and `vectorB` start off with a call to `VM.new 1`. If we expand this using our `inlinePerformIO` implementation, this looks something like:

```
vectorA =
```

```
    case VM.new 1 realWorld# of
        (# s1, mv #) ->
            case VM.write mv 0 'A' s1 of
                (# s2, () #) ->
                    case V.unsafeFreeze mv s2 of
                        (# _, v #) -> v

vectorB =
    case VM.new 1 realWorld# of
        (# s1, mv #) ->
            case VM.write mv 0 'B' s1 of
                (# s2, () #) ->
                    case V.unsafeFreeze mv s2 of
                        (# _, v #) -> v
```

But notice how both `vectorA` and `vectorB` start in exactly the same way. It's valid to rewrite this code to use sharing:

```
(# s1, mv #) = VM.new 1 realWorld#

vectorA =
    case VM.write mv 0 'A' s1 of
        (# s2, () #) ->
            case V.unsafeFreeze mv s2 of
                (# _, v #) -> v

vectorB =
    case VM.write mv 0 'B' s1 of
        (# s2, () #) ->
            case V.unsafeFreeze mv s2 of
                (# _, v #) -> v
```

Do you see the problem with this? **Both vectors will point to the same block of memory!** This means that we'll first write 'A' into the vector, then overwrite that with 'B', and end up with "two vectors" both containing the same values. That's clearly not what we wanted!

The answer to this is to avoid the possibility of sharing. And to do that, we use magic, aka lazy (http://hackage.haskell.org/package/ghc-prim-0.3.1.0/docs/GHC-Magic.html#v:lazy) , and pragmas. To wit, the implementation of `unsafeDupablePerformIO` is:

```
{-# NOINLINE unsafeDupablePerformIO #-}
unsafeDupablePerformIO  :: IO a -> a
unsafeDupablePerformIO (IO m) = lazy (case m realWorld# of (# _, r #) -> r)
```

This has two changes from `inlinePerformIO`:

- The `NOINLINE` pragma ensures that the expression is never inlined, which stops any sharing.
- `lazy` prevents premature evaluation of the action. (*NB: I'm actually not completely certain of what `lazy` is ensuring here.*)

We can now safely allocate memory inside `unsafeDupablePerformIO`, and now that allocated memory won't be shared among other calls. But we pay a small performance cost by avoiding the inlining.

So what's the downside of this function? Pretty simple, actually: if you have two threads which evaluate a thunk at the same time, they may both start performing the action. And when the first thread completes the action, it may terminate the execution of the other thread. In other words, with:

```
import System.IO.Unsafe
import Control.Concurrent

thunk :: ()
thunk = unsafeDupablePerformIO $ do
    putStrLn "starting thunk"
    threadDelay 1000000
    putStrLn "finished thunk"

main :: IO ()
main = do
    forkIO $ print thunk
    threadDelay 500000
    print thunk
    threadDelay 1000000
```

"starting thunk" may be printed multiple times, and "finished thunk" may be printed less times than "starting thunk". This can't even be worked around by using something like `bracket`, since in this situation, the second thread doesn't receive an exception, it simply stops executing. The two upshots of this are:

- If you need to ensure that an action is only run once, this is problematic.
- If you need to guarantee some kind of resource cleanup, this is problematic.

Which leads us to our final function: `unsafePerformIO`. It has an implementation of:

```
unsafePerformIO :: IO a -> a
unsafePerformIO m = unsafeDupablePerformIO (noDuplicate >> m)
```

This uses some GHC internal magic to ensure that the action is only run by a single thread, fixing our two problems above. As you might guess, this *also* introduces a small performance cost, which is the motivation to avoidng it in favor of `unsafeDupablePerformIO` or `inlinePerformIO`.

4 Guidelines on using each function

Note that the guidelines below are cummulative: the requirements for using `unsafeInterleaveIO`, for example, apply to the other three functions as well.

- Whenever possible, avoid using unsafe functions.

- If you're certain that it's safe to perform the action only on evaluation, use `unsafeInterleaveIO`. You need to ensure that you don't mind which order the effects are performed in, relative to both the main I/O monad and other calls to `unsafePerformIO`.
- If you aren't in the `IO` monad at all, or it's acceptable if the action is performed before other `IO` actions, use `unsafePerformIO`.
- If you need extra speed, and it's acceptable for the action to be performed multiple times, and it's acceptable if this action is canceled halfway through its execution, use `unsafeDupablePerformIO`. Another way of saying this is that the action should be idempotent, and require no cleanup.
- If you need even extra speed, you're performing no actions that would be problematic if shared (e.g., memory allocation), and you're OK with the fact that your code is very likely broken, use `inlinePerformIO`. Seriously, be very, very, very careful.
  - Another guideline is to never perform writes inside `inlinePerformIO`. However, I believe that this is not actually strictly necessary, but instead a result of a long-standing GHC bug (https://ghc.haskell.org/trac/ghc/ticket/9390) , which should be fixed since GHC 7.8.4. Incidentally, that issue was the impetus for the writing of this document.

5 Interaction with STM

A related issue to point out is how `unsafePerformIO` interact with `STM`. Since `STM` transactions can be retried multiple times, an `unsafePerformIO` action may be run multiple times as well. In that sense, you should treat such calls similarly to how you would normally treat `unsafeDupablePerformIO`.

However, there's a long-standing runtime system bug where aborted STM transactions do not result in any exceptions being thrown, which leads to a similar behavior of missing cleanup actions as we have with `inlinePerformIO`. For more information, see:

- http://www.haskell.org/pipermail/haskell-cafe/2014-February/112555.html
- https://ghc.haskell.org/trac/ghc/ticket/2401