All about strictness

Blog post version

Haskell is—perhaps infamously—a *lazy* language. The basic idea of laziness is pretty easy to sum up in one sentence: values are only computed when they're needed. But the implications of this are more subtle. In particular, it's important to understand some crucial topics if you want to write memory- and time-efficient code:

- Weak head normal form (WHNF) versus normal form (NF)
- How to use the `seq` and `deepseq` functions (and related concepts)
- Strictness annotations on data types
- Bang patterns
- Strictness of data structures: lazy, spine-strict, and value-strict
- Choosing the appropriate helper functions, especially with folds

This blog post was inspired by some questions around writing efficient conduit code, so I'll try to address some of that directly at the end. The concepts, though, are general, and will transfer to not only other streaming libraries, but non-streaming data libraries too.

NOTE This blog post will mostly treat laziness as a problem to be solved, as opposed to the reality: laziness is sometimes an asset, and sometimes a liability. I'm focusing on the negative exclusively, because our goal here is to understand the rough edges and how to avoid them. There are many great things about laziness that I'm not even hinting at. I trust my readers to add some great links to articles speaking on the virtues of laziness in the comments :)

Basics of laziness

Let's elaborate on my one liner above:

> Values are only computed when they're needed

Let's explore this by comparison with a strict language: C.

```c
#include <stdio.h>

int add(int x, int y) {
  return x + y;
}

int main() {
  int five = add(1 + 1, 1 + 2);
  int seven = add(1 + 2, 1 + 3);

  printf("Five: %d\n", five);
  return 0;
}
```

Our function `add` is *strict* in both of its arguments. And its result is also strict. This means that:

- Before `add` is called the first time, we will compute the result of both `1 + 1` and `1 + 2`.
- We will call the `add` function on `2` and `3`, get a result of `5`, and place that value in memory pointed at by the variable `five`.
- Then we'll do the same thing with `1 + 2`, `1 + 3`, and placing `7` in `seven`.
- Then we'll call `printf` with our `five` value, which is already fully computed.

Let's compare that to the equivalent Haskell code:

```haskell
add :: Int -> Int -> Int
add x y = x + y

main :: IO ()
main = do
  let five = add (1 + 1) (1 + 2)
      seven = add (1 + 2) (1 + 3)

  putStrLn $ "Five: " ++ show five
```

There's something called *strictness analysis* which will result in something more efficient than what I'll describe here in practice, but semantically, we'll end up with the following:

- Instead of immediately computing `1 + 1` and `1 + 2`, the compiler will create a *thunk* (which you can think of as a *promise*) for tho computations, and pass those thunks to the `add` function.
- Except: we won't call the `add` function right away either: `five` will be a thunk representing the application of the `add` function to th thunk for `1 + 1` and `1 + 2`.
- We'll end up doing the same thing with `seven`: it will be a thunk for applying `add` to two other thunks.
- When we finally try to print out the value `five`, we need to know the actual number. This is called *forcing evaluation*. We'll get into more detail on when and how this happens below, but for now, suffice it to say that when `putStrLn` is executed, it forces evaluatio of `five`, which forces evaluation of `1 + 1` and `1 + 2`, converting the thunks into real values (`2`, `3`, and ultimately `5`).
- Because `seven` is never used, it remains a thunk, and we don't spend time evaluating it.

Compared to the C (strict) evaluation, there is one clear benefit: we don't bother wasting time evaluating the `seven` value at all. That's three addition operations bypassed, woohoo! And in a real world scenario, instead of being three additions, that could be a seriously expensive operation.

However, it's not all rosey. Creating a thunk does not come for free: we need to allocate space for the thunk, which costs both allocation,

and causes GC pressure for freeing them afterwards. Perhaps most importantly: the thunked version of an expression can be far more costly than the evaluated version. Ignoring some confusing overhead from data constructors (which only make the problem worse), let's compare our two representations of `five`. In C, `five` takes up exactly one machine word*. In Haskell, our `five` thunk will take up roughl

* Or perhaps less, as `int` is probably only 32 bits, and you're probably on a 64 bit machine. But then you get into alignment issues, and registers... so let's just say one machine word.

- One machine word to say "I'm a thunk"
- Within that thunk, pointers to the `add` function, and the `1 + 1` and `1 + 2` thunks (one machine word each). So three machine wor total.
- Within the `1 + 1` thunk, one machine word for the thunk, and then again a pointer to the `+` operator, and the `1` values. (GHC has ar optimization where it will keep small int values in dedicated parts of memory, avoiding extra overhead for the ints themselves. But you could theoretically add in an extra machine word for each.) Again, conservatively: three machine words.
- Same logic for the `1 + 2` thunk, so three more machine words.
- For a whopping total of **10 machine words**, or 10 times the memory usage as C!

Now in practice, it's not going to work out that way. I mentioned the strictness analysis step, which will say "hey, wait a second, it's totally better to just add two numbers than allocate a thunk, I'mma do that now, kthxbye." But it's vital when writing Haskell to understand all of these places where laziness and thunks can creep in.

Bang!

Let's look at how we can force Haskell to be more strict in its evaluation. Likely the easiest way to do this is with bang patterns. Let's look at the code first:

```
{-# LANGUAGE BangPatterns #-}
add :: Int -> Int -> Int
add !x !y = x + y

main :: IO ()
main = do
  let !five = add (1 + 1) (1 + 2)
      !seven = add (1 + 2) (1 + 3)

  putStrLn $ "Five: " ++ show five
```

This code now behaves exactly like the strict C code. Because we've put a bang (`!`) in front of the `x` and `y` in the `add` function, GHC know that it must force evaluation of those values before evaluating it. Similarly, by placing bangs on `five` and `seven`, GHC must evaluate the immediately, before getting to `putStrLn`.

As with many things in Haskell, however, bang patterns are just syntactic sugar for something else. And in this case, that something else the `seq` function. This function looks like:

```
seq :: a -> b -> b
```

You could implement this type signature yourself, of course, by just ignoring the `a` value:

```
badseq :: a -> b -> b
badseq a b = b
```

However, `seq` uses primitive operations from GHC itself to ensure that, when `b` is evaluated, `a` is evaluated too. Let's rewrite our `add`

function to use `seq` instead of bang patterns:

```
add :: Int -> Int -> Int
add x y =
  let part1 = seq x part2
      part2 = seq y answer
      answer = x + y
   in part1
-- Or more idiomatically
add x y = x `seq` y `seq` x + y
```

What this is saying is this:

- `part1` is an expression which will tell you the value of `part2`, after it evaluates `x`
- `part2` is an expression which will tell you the value of `answer`, after it evaluates `y`
- `answer` is just `x + y`

Of course, that's a long way to write this out, and the pattern is common enough that people will usually just use `seq` infix as demonstrat above.

EXERCISE What would happen if, instead of `in part1`, the code said `in part2`? How about `in answer`?

There is always a straightforward translation from bang patterns to usage of `let`. We can do the same with the `main` function:

```
main :: IO ()
main = do
  let five = add (1 + 1) (1 + 2)
      seven = add (1 + 2) (1 + 3)

  five `seq` seven `seq` putStrLn ("Five: " ++ show five)
```

It's vital to understand how `seq` is working, but there's no advantage to using it over bang patterns where the latter are clear and easy to read. Choose whichever option makes the code easiest to read, which will often be bang patterns.

Tracing evaluation

So far, you've just had to trust me about the evaluation of thunks occurring. Let's see a method to more directly observe evaluation. The `trace` function from `Debug.Trace` will print a message when it is evaluated. Take a guess at the output of these programs:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Debug.Trace

add :: Int -> Int -> Int
add x y = x + y

main :: IO ()
main = do
  let five = trace "five" (add (1 + 1) (1 + 2))
      seven = trace "seven" (add (1 + 2) (1 + 3))

  putStrLn $ "Five: " ++ show five
```

Versus:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE BangPatterns #-}
import Debug.Trace

add :: Int -> Int -> Int
add x y = x + y

main :: IO ()
main = do
  let !five = trace "five" (add (1 + 1) (1 + 2))
      !seven = trace "seven" (add (1 + 2) (1 + 3))

  putStrLn $ "Five: " ++ show five
```

Think about this before looking at the answer...

OK, hope you had a good think. Here's the answer:

- The first program will print both `five` and `Five: 5`. It will not bother printing `seven`, since that expression is never forced. (Due t strangeness around output buffering, you may see interleaving of these two output values.)
- The second will print both `five` and `seven`, because the bang patterns force their evaluation. *However*, the order of their output may be different than you expect. On my system, for example, `seven` prints before `five`. That's because GHC retains the right to rearrange order of evaluation in these cases.
- By contrast, if you use `five `seq` seven `seq` putStrLn ("Five: " ++ show five)`, it comes out in the order you would intuitively expect: first five, then seven, and then "Five: 5". This gives a bit of a lie to my claim that bang patterns are always a simple translation to `seq`s. However, the fact is that with an expression `x `seq` y`, GHC is free to choose whether it evaluates `x` o `y` first, as long as it ensure that when that expression finishes evaluating, both `x` and `y` are evaluated.

All that said: as long as your expressions are truly pure, you will be unable to observe the difference between `x` and `y` evaluating first. Onl the fact that we used `trace`, which is an impure function, allowed us to observe the order of evaluation.

QUESTION Does the result change at all if you put bangs on the `add` function? Why do bangs there affect (or not affect) the output?

The value of bottom

This is all well and good, but the more standard way to demonstrate evaluation order is to use bottom values, aka `undefined`. `undefined` is special in that, when it is evaluated, it throws a runtime exception. (The `error` function does the same thing, as do a few other special functions and values.) To demonstrate the same thing about `seven` not being evaluated without the bangs, compare these two programs:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE BangPatterns #-}

add :: Int -> Int -> Int
add x y = x + y

main :: IO ()
main = do
  let five = add (1 + 1) (1 + 2)
      seven = add (1 + 2) undefined -- (1 + 3)

  putStrLn $ "Five: " ++ show five
```

Versus:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE BangPatterns #-}

add :: Int -> Int -> Int
add x y = x + y

main :: IO ()
main = do
  let five = add (1 + 1) (1 + 2)
      !seven = add (1 + 2) undefined -- (1 + 3)

  putStrLn $ "Five: " ++ show five
```

The former completes without issue, since `seven` is never evaluated. However, in the latter, we have a bang pattern on `seven`. What GHC does here is:

- Evaluate the expression `add (1 + 2) undefined`
- This reduces to `(1 + 2) + undefined`
- But this is still an expression, not a value, so more evaluation is necessary
- In order to evaluate the `+` operator, it needs actual values for the two arguments, not just thunks. This can be seen as if `+` has bang patterns on its arguments. The correct way to say this is "`+` is strict in both of its arguments."
- GHC is free to choose to either evaluate `1 + 2` or `undefined` first. Let's assume it does `1 + 2` first. It will come up with two evaluated values (`1` and `2`), pass them to `+`, and get back `3`. All good.
- However, it then tries to evaluate `undefined`, which triggers a runtime exception to be thrown.

QUESTION Returning to the question above: does it look like bang patterns inside the `add` function actually accomplish anything? Think about what the output of this program will be:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE BangPatterns #-}

add :: Int -> Int -> Int
add !x !y = x + y

main :: IO ()
main = do
  let five = add (1 + 1) (1 + 2)
      seven = add (1 + 2) undefined -- (1 + 3)

  putStrLn $ "Five: " ++ show five
```

To compare this behavior to a strict language, we need a language with something like runtime exceptions. I'll use Rust's panics:

```
fn add(x: isize, y: isize) -> isize {
    println!("adding: {} and {}", x, y);
    x + y
}

fn main() {
    let five = add(1 + 1, 1 + 2);
    let seven = add(1 + 2, panic!());

    println!("Five: {}", five);
}
```

Firstly, to Rust's credit: it gives me a bunch of warnings about how this program is dumb. Fair enough, but I'm going to ignore those warnings and charge ahead with it. This program will first evaluate the `add(1 + 1, 1 + 2)` expression (which we can see in the outpu of `adding: 2 and 3`). Then, before it ever enters the `add` function the second time, it needs to evaluate both `1 + 2` and `panic!()`. The former works just fine, but the latter results in a panic being generated and short-circuiting the rest of our function.

If we want to regain Haskell's laziness properties, there's a straightforward way to do it: use a closure. A closure is, essentially, a thunk. Th Rust syntax for creating a closure is `|args| body`. We can create closures with no arguments to act like thunks, which gives us:

```
fn add<X, Y>(x: X, y: Y) -> isize
    where X: FnOnce() -> isize,
          Y: FnOnce() -> isize {
    let x = x();
    let y = y();
    println!("adding: {} and {}", x, y);
    x + y
}

fn main() {
    let five = || add(|| 1 + 1, || 1 + 2);
    let seven = || add(|| 1 + 2, || panic!());

    println!("Five: {}", five());
}
```

Again, the Rust compiler complains about the unused `seven`, but this program succeeds in running, since we never run the `seven` closu

Still not up to speed with Rust? Let's use everyone's favorite language: Javascript:

```
function add(x, y) {
    return x() + y();
}

function panic() {
    throw "Panic!"
}

var five = ignored => add(ignored => 1 + 1, ignored => 1 + 2);
var seven = ignored => add(ignored => 1 + 2, panic);
console.log("Five: " + five());
```

Alright, to summarize until now:

- Haskell is lazy by default
- You can use bang patterns and `seq` to make things strict
- By contrast, in strict languages, you can use closures to make things lazy
- You can see if a function is strict in its arguments by passing in bottom (`undefined`) and seeing if it explodes in your face
- The `trace` function can help you see this as well

This is all good, and make sure you have a solid grasp of these concepts before continuing. Consider rereading the sections above.

Average

Here's something we didn't address: what, exactly, does it mean to evaluate or force a value? To demonstrate the problem, let's implemen an average function. We'll use a helper datatype, called `RunningTotal`, to capture both the cumulative sum and the number of element we've seen so far.

```haskell
data RunningTotal = RunningTotal
  { sum :: Int
  , count :: Int
  }

printAverage :: RunningTotal -> IO ()
printAverage (RunningTotal sum count)
  | count == 0 = error "Need at least one value!"
  | otherwise = print (fromIntegral sum / fromIntegral count :: Double)

-- | A fold would be nicer... we'll see that later
printListAverage :: [Int] -> IO ()
printListAverage =
  go (RunningTotal 0 0)
  where
    go rt [] = printAverage rt
    go (RunningTotal sum count) (x:xs) =
      let rt = RunningTotal (sum + x) (count + 1)
       in go rt xs

main :: IO ()
main = printListAverage [1..1000000]
```

We're going to run this with run time statistics turned on so we can look at memory usage:

```
$ stack ghc average.hs && ./average +RTS -s
```

Lo and behold, our memory usage is through the roof!

```
[1 of 1] Compiling Main             ( average.hs, average.o )
Linking average ...
500000.5
     258,654,528 bytes allocated in the heap
     339,889,944 bytes copied during GC
      95,096,512 bytes maximum residency (9 sample(s))
       1,148,312 bytes maximum slop
             164 MB total memory in use (0 MB lost due to fragmentation)
```

We're allocating a total of 258MB, and keeping 95MB in memory at once. For something that should just be a tight inner loop, that's *ridiculously* large.

Bang!

You're probably thinking right now "shouldn't we use that `seq` stuff or those bang patterns?" Certainly that makes sense. And in fact, it looks really trivial to solve this problem with a single bang to force evaluation of the newly constructed `rt` before recursing back into `go`. For example, we can add `{-# LANGUAGE BangPatterns #-}` to the top of our file and then define `go` as:

```haskell
go !rt [] = printAverage rt
go (RunningTotal sum count) (x:xs) =
  let rt = RunningTotal (sum + x) (count + 1)
   in go rt xs
```

Unfortunately, this results in *exactly* the same memory usage as we had before. In order to understand why this is happening, we need to look at something called weak head normal form.

Weak Head Normal Form

Note in advance that there's a great Stack Overflow answer on this topic for further reading.

We've been talking about forcing values and evaluating expressions, but what exactly that means hasn't been totally clear. To start simple what will the output of this program be?

```
main = putStrLn $ undefined `seq` "Hello World"
```

You'd probably guess that it will print an error about `undefined`, since it will try to evaluate `undefined` before it will evaluate `"Hello World"`, and because `putStrLn` is strict in its argument. And you'd be correct. But let's try something a little bit different:

```
main = putStrLn $ Just undefined `seq` "Hello World"
```

If you assume that "evaluate" means "fully evaluate into something with no thunks left," you'll say that this, too, prints an `undefined` erro But in fact, it happily prints out "Hello World" with no exceptions. What gives?

It turns out that when we talk about forcing evaluation with `seq`, we're only talking about evaluating to *weak head normal form (WHNF)*. For most data types, this means unwrapping one layer of constructor. In the case of `Just undefined`, it means that we unwrap the `Jus` data constructor, but don't touch the `undefined` within it. (We'll see a few ways to deal with this differently below.)

It turns out that, with a standard data constructor*, the impact of using `seq` is the same as pattern matching the outermost constructor. I you want to monomorphise, for example, you can implement a function of type `seqMaybe :: Maybe a -> b -> b` and use it in the `main` example above. Go ahead and give it a shot... answer below.

* Hold your horses, we'll talk about `newtype`s later and then you'll understand this weird phrasing.

```
seqMaybe :: Maybe a -> b -> b
seqMaybe Nothing b = b
seqMaybe (Just _) b = b

main :: IO ()
main = do
  putStrLn $ Just undefined `seqMaybe` "Hello World"
  putStrLn $ undefined `seqMaybe` "Goodbye!"
```

Let's up the ante again. What do you think this program will print?

```
main = do
  putStrLn $ error `seq` "Hello"
  putStrLn $ (\x -> undefined) `seq` "World"
  putStrLn $ error "foo" `seq` "Goodbye!"
```

You might think that `error `seq` ...` would be a problem. After all, isn't `error` going to throw an exception? However, `error` is a function. There's no exception getting thrown, or no bottom value being provided, until `error` is given its `String` argument. As a result, evaluating does not, in fact, generate an error. The rule is: any function applied to too few values is automatically in WHNF.

A similar logic applies to `(\x -> undefined)`. Although it's a lambda expression, its type is a function which has not been applied to a

arguments. And therefore, it will not throw an exception when evaluated. In other words, it's already in WHNF.

However, `error "foo"` is a function fully applied to its arguments. It's no longer a function, it's a value. And when we try to evaluate it t WHNF, its exception blows up in our face.

EXERCISE Will the following throw exceptions when evaluated?

- `(+) undefined`
- `Just undefined`
- `undefined 5`
- `(error "foo" :: Int -> Double)`

Fixing average

Having understood WHNF, let's return to our example and see why our first bang pattern did nothing to help us:

```
go !rt [] = printAverage rt
go (RunningTotal sum count) (x:xs) =
  let rt = RunningTotal (sum + x) (count + 1)
   in go rt xs
```

In WHNF, forcing evaluation is the same as unwrapping the constructor, which we are already doing in the second clause! The problem is that the values contained inside the `RunningTotal` data constructor are not being evaluated, and therefore are accumulating thunks. Let's see two ways to solve this:

```
go rt [] = printAverage rt
go (RunningTotal !sum !count) (x:xs) =
  let rt = RunningTotal (sum + x) (count + 1)
   in go rt xs
```

Instead of putting the bangs on the `RunningTotal` value, I'm putting them on the values *within* the constructor, forcing them to be evaluated at each loop. We're no longer accumulating a huge chain of thunks, and our maximum residency drops to 44kb. (Total allocations, though, are still up around 192mb. We need to play around with other optimizations outside the scope of this post to deal with the total allocations, so we're going to ignore this value for the rest of the examples.) Another approach is:

```
go rt [] = printAverage rt
go (RunningTotal sum count) (x:xs) =
  let !sum' = sum + x
      !count' = count + 1
      rt = RunningTotal sum' count'
   in go rt xs
```

This one instead forces evaluation of the new sum and count *before* constructing the new `RunningTotal` value. I like this version a bit more, as it's forcing evaluation at the correct point: when creating the value, instead of on the next iteration of the loop when destructing

Moral of the story: make sure you're evaluating the thing you actually need to evaluate, not just its container!

deepseq

The fact that `seq` only evaluates to weak head normal form is annoying. There are lots of times when we would like to fully evaluate dow to *normal form* (NF), meaning all thunks have been evaluated inside our values. While there is nothing built into the language to handle this, there is a semi-standard (meaning it ships with GHC) library to handle this: `deepseq`. It works by providing an `NFData` type class th

defines how to __r__educe a value to __n__ormal __f__orm (via the `rnf` method).

```
{-# LANGUAGE BangPatterns #-}
import Control.DeepSeq

data RunningTotal = RunningTotal
  { sum :: Int
  , count :: Int
  }
instance NFData RunningTotal where
  rnf (RunningTotal sum count) = sum `deepseq` count `deepseq` ()

printAverage :: RunningTotal -> IO ()
printAverage (RunningTotal sum count)
  | count == 0 = error "Need at least one value!"
  | otherwise = print (fromIntegral sum / fromIntegral count :: Double)

-- | A fold would be nicer... we'll see that later
printListAverage :: [Int] -> IO ()
printListAverage =
  go (RunningTotal 0 0)
  where
    go rt [] = printAverage rt
    go (RunningTotal sum count) (x:xs) =
      let rt = RunningTotal (sum + x) (count + 1)
       in rt `deepseq` go rt xs

main :: IO ()
main = printListAverage [1..1000000]
```

This has a maximum residency, once again, of 44kb. We define our `NFData` instance, which includes an `rnf` method. The approach of simply `deepseq`ing all of the values within a data constructor is almost always the approach to take for `NFData` instances. In fact, it's so common, that you can get away with just using `Generic` deriving and have GHC do the work for you:

```
{-# LANGUAGE DeriveGeneric #-}
import GHC.Generics (Generic)
import Control.DeepSeq

data RunningTotal = RunningTotal
  { sum :: Int
  , count :: Int
  }
  deriving Generic
instance NFData RunningTotal
```

The true beauty of having `NFData` instances is the ability to abstract over many different data types. We can use this not only to avoid space leaks (as we're doing here), but also to avoid accidentally including exceptions inside thunks within a value. For an example of that check out the tryAnyDeep function from the safe-exceptions library.

EXERCISE Define the `deepseq` function yourself in terms of `rnf` and `seq`.

Strict data

These approaches work, but they are not ideal. The problem lies in our definition of `RunningTotal`. What we *want* to say is that, whenever you have a value of type `RunningTotal`, you in fact have two `Int`s. But because of laziness, what we're actually saying is tha `RunningTotal` value could contain two `Int`s, or it could contain thunks that will evaluate to `Int`s, or thunks that will throw exceptions.

Instead, we'd like to make it impossible to construct a `RunningTotal` value that has any laziness room left over. And to do that, we can use *strictness annotations* in our definition of the data type:

```
data RunningTotal = RunningTotal
  { sum :: !Int
  , count :: !Int
  }
  deriving Generic

printAverage :: RunningTotal -> IO ()
printAverage (RunningTotal sum count)
  | count == 0 = error "Need at least one value!"
  | otherwise = print (fromIntegral sum / fromIntegral count :: Double)

-- | A fold would be nicer... we'll see that later
printListAverage :: [Int] -> IO ()
printListAverage =
  go (RunningTotal 0 0)
  where
    go rt [] = printAverage rt
    go (RunningTotal sum count) (x:xs) =
      let rt = RunningTotal (sum + x) (count + 1)
       in go rt xs

main :: IO ()
main = printListAverage [1..1000000]
```

All we've done is put bangs in front of the `Int`s in the definition of `RunningTotal`. We have no other references to strictness or evaluation in our program. However, by placing the strictness annotations on those fields, we're saying something simple and yet profour

Whenever you evaluate a value of type `RunningTotal`, you must also evaluate the two `Int`s it contains

As we mentioned above, our second `go` clause forces evaluation of the `RunningTotal` value by taking apart its constructor. This act no automatically forces evaluation of `sum` and `count`, which we previously needed to achieve via a bang pattern.

There's one other advantage to this, which is slightly out of scope but worth mentioning. When dealing with small values like an `Int`, GH will automatically *unbox* strict fields. This means that, instead of keeping a pointer to an `Int` inside `RunningTotal`, it will keep the `Int` itself. This can further reduce memory usage.

You're probably asking a pretty good question right now: "how do I know if I should use a strictness annotation on my data fields?" This answer is slightly controversial, but my advice and recommended best practice: unless you know that you want laziness for a field, make strict. Making your fields strict helps in a few ways:

- Avoids accidental space leaks, like we're doing here
- Avoids accidentally including bottom values
- When constructing a value with record syntax, GHC will give you an error if you forget a strict field. It will only give you a warning fo non-strict fields.

The curious case of newtype

Let's define three very similar data types:

```
data Foo = Foo Int
data Bar = Bar !Int
newtype Baz = Baz Int
```

Let's play a game, and guess the output of the following potential bodies for `main`. Try to work through each case in your head before reading the explanation below.

1. `case undefined of { Foo _ -> putStrLn "Still alive!" }`
2. `case Foo undefined of { Foo _ -> putStrLn "Still alive!" }`
3. `case undefined of { Bar _ -> putStrLn "Still alive!" }`
4. `case Bar undefined of { Bar _ -> putStrLn "Still alive!" }`
5. `case undefined of { Baz _ -> putStrLn "Still alive!" }`
6. `case Baz undefined of { Baz _ -> putStrLn "Still alive!" }`

Case (1) is relatively straightforward: we try to unwrap one layer of data constructor (the `Foo`) and find a bottom value. So this thing throws an exception. The same thing applies to (3).

(2) does *not* throw an exception. We have a `Foo` data constructor in our expression, and it contains a bottom value. However, since there no strictness annotation on the `Int` in `Foo`, uwnrapping the `Foo` does not force evaluation of the `Int`, and therefore no exception is thrown. By contrast, in (4), we *do* have a strictness annotation, and therefore `case`ing on `Bar` throws an exception.

What about `newtype`s? What we know about `newtype`s is that they have no runtime representation. Therefore, it's impossible for the `Ba` data constructor to be hiding an extra layer of bottomness. In other words, `Baz undefined` and `undefined` are indistinguishable. That may sound like `Bar` at first, but interestingly it's not.

You see, unwrapping a `Baz` constructor can have no effect on runtime behavior, since it was never there in the first place. The pattern match inside (5), therefore, does *nothing*. It is equivalent to `case undefined of { _ -> putStrLn "Still alive!" }`. And sin we're not inspecting the `undefined` at all (because we're using a wildcard pattern and not a data constructor), no exception is thrown.

Similarly, in case (6), we've applied a `Baz` constructor to `undefined`, but since it has no runtime representation, it may as well not be there. So once again, no exception is thrown.

EXERCISE What is the output of the program `main = Baz undefined `seq` putStrLn "Still alive!"`? Why?

Convenience operators and functions

It can be inconvenient, as you may have noticed already, to use `seq` and `deepseq` all over the place. Bang patterns help, but there are other ways to force evaluation. Perhaps the most common is the `$!` operator, e.g.:

```
mysum :: [Int] -> Int
mysum list0 =
  go list0 0
  where
    go [] total = total
    go (x:xs) total = go xs $! total + x

main = print $ mysum [1..1000000]
```

This forces evaluation of `total + x` before recursing back into the `go` function, avoiding a space leak. (EXERCISE: do the same thing with a bang pattern, and with the `seq` function.)

The `$!!` operator is the same, except instead of working with `seq`, it uses `deepseq` and therefore evaluates to normal form.

```
import Control.DeepSeq

average :: [Int] -> Double
average list0 =
  go list0 (0, 0)
  where
    go [] (total, count) = fromIntegral total / count
    go (x:xs) (total, count) = go xs $!! (total + x, count + 1)

main = print $ average [1..1000000]
```

Another nice helper function is `force`. What this does is makes it that, when the expression you're looking at is evaluated to WHNF, it's *actually* evaluated to NF. For example, we can rewrite the `go` function above as:

```
go [] (total, count) = fromIntegral total / count
go (x:xs) (total, count) = go xs $! force (total + x, count + 1)
```

EXERCISE Define these convenience functions and operators yourself in terms of `seq` and `deepseq`.

Data structures

Alright, I swear that's all of the really complicated stuff. If you've absorbed all of those details, the rest of this just follows naturally and introduces a little bit more terminology to help us understand things.

Let's start off slowly: what's the output of this program:

```
data List a = Cons a (List a) | Nil

main = Cons undefined undefined `seq` putStrLn "Hello World"
```

Well, using our principles from above: `Cons undefined undefined` is already in WHNF, since we've got the outermost constructor available. So this program prints "Hello World", without any exceptions. Cool. Now let's realize that `Cons` is the same as the `:` data constructor for lists, and see that the above is identical to:

```
main = (undefined:undefined) `seq` putStrLn "Hello World"
```

This tells me that lists are a lazy data structure: I have a bottom value for the first element, a bottom value for the rest of the list, and yet this first cell is not bottom. Let's try something a little bit different:

```
data List a = Cons a !(List a) | Nil

main = Cons undefined undefined `seq` putStrLn "Hello World"
```

This is going to explode in our faces! We are now strict in the tail of the list. However, the following is fine:

```
data List a = Cons a !(List a) | Nil

main = Cons undefined (Cons undefined Nil) `seq` putStrLn "Hello World"
```

With this definition of a list, we need to know all the details about the list itself, but the values can remain undefined. This is called *spine strict*. By contrast, we can also be strict in the values and be *value strict*:

```
data List a = Cons !a !(List a) | Nil

main = Cons undefined (Cons undefined Nil) `seq` putStrLn "Hello World"
```

This will explode in our faces, as we'd expect.

There's one final definition of list you may be expecting, one strict in values but not in the tail:

```
data List a = Cons !a (List a) | Nil
```

In practice, I'm aware of no data structures in Haskell that follow this pattern, and therefore it doesn't have a name. (If there *are* such data structures, and this does have a name, please let me know, I'd be curious about the use cases for it.)

So standard lists are lazy. Let's look at a few other data types:

Vectors

The vectors in `Data.Vector` (also known as *boxed vectors*) are spine strict. Assuming an import of `import qualified Data.Vector as V`, what would be the results of the following programs?

1. `main = V.fromList [undefined] `seq` putStrLn "Hello World"`
2. `main = V.fromList (undefined:undefined) `seq` putStrLn "Hello World"`
3. `main = V.fromList undefined `seq` putStrLn "Hello World"`

The first succeeds: we have the full spine of the vector defined. The fact that it contains a bottom value is irrelevant. The second fails, since the spine of the tail of the list is undefined, making the spine undefined. And finally the third (of course) fails, since the entire list is undefined.

Now let's look at *unboxed vectors*. Because of inference issues, we need to help out GHC a little bit more. So starting with this head of a program:

```
import qualified Data.Vector.Unboxed as V

fromList :: [Int] -> V.Vector Int
fromList = V.fromList
```

What happens with the three cases above?

1. `main = fromList [undefined] `seq` putStrLn "Hello World"`
2. `main = fromList (undefined:undefined) `seq` putStrLn "Hello World"`
3. `main = fromList undefined `seq` putStrLn "Hello World"`

As you'd expect, (2) and (3) have the same behavior as with boxed vectors. However, (1) *also* throws an exception, since unboxed vectors are value strict, not just spine strict. The same applies to storable and primitive vectors.

Unfortunately, to my knowledge, there is no definition of a strict, boxed vector in a public library. Such a data type would be useful to help avoid space leaks (such as the original question that triggered this blog post).

Sets and Maps

If you look at the containers and unordered-containers packages, you may have noticed that the Map-like modules come in `Strict` and `Lazy` variants (e.g., `Data.HashMap.Strict` and `Data.HashMap.Lazy`) while the Set-like modules do not (e.g., `Data.IntSet`). This is because *all* of these containers are spine strict, and therefore must be strict in the keys. Since a set only has keys, no separate values, it must also be value strict.

A map, by contrast, has both keys and values. The lazy variants of the map-like modules are spine-strict, value-lazy, whereas the strict variants are both spine and value strict.

EXERCISE Analyze the `Data.Sequence.Seq` data type and classify it as either lazy, spine strict, or value strict.

Function arguments

A function is considered strict in one of its arguments if, when the function is applied to a bottom value for that argument, the result is bottom. As we saw way above, `+` for `Int` is strict in both of its arguments, since: `undefined + x` is bottom, and `x + undefined` is bottom.

By contrast, the `const` function, defined as `const a b = a`, is strict in its first argument and lazy in its second argument.

The `:` data constructor for lists is lazy in both its first and second argument. But if you have `data List = Cons !a !(List a) | Nil`, `Cons` is strict in both its first and second argument.

Folds

A common place to end up getting tripped up by laziness is dealing with folds. The most infamous example is the `foldl` function, which lulls you into a false sense of safety only to dash your hopes and destroy your dreams:

```
mysum :: [Int] -> Int
mysum = foldl (+) 0

main :: IO ()
main = print $ mysum [1..1000000]
```

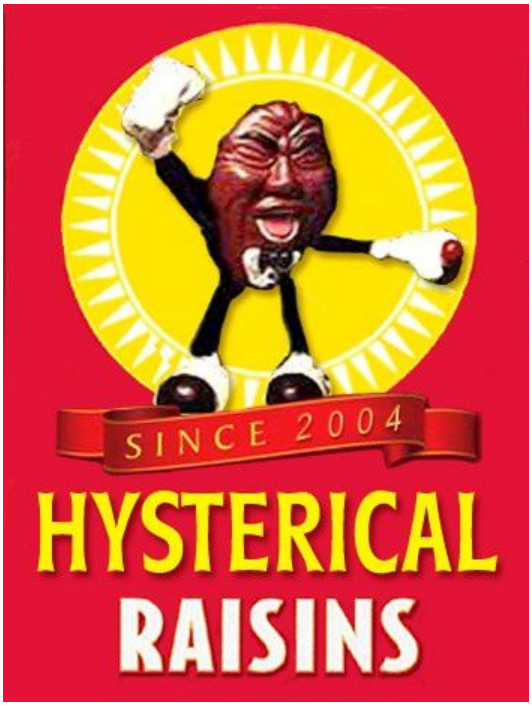
This is so close to correct, and yet uses 53mb of resident memory! The solution is but a tick away, using the strict left fold `foldl'` function:

```
import Data.List (foldl')

mysum :: [Int] -> Int
mysum = foldl' (+) 0

main :: IO ()
main = print $ mysum [1..1000000]
```

Why does the `Prelude` expose a function (`foldl`) which is almost always the wrong one to use?



But the important thing to note about almost all functions that claim to be strict is that they are only strict to weak head normal form. Pulling up our `average` example from before, this still has a space leak:

```
import Data.List (foldl')

average :: [Int] -> Double
average =
  divide . foldl' add (0, 0)
  where
    divide (total, count) = fromIntegral total / count
    add (total, count) x = (total + x, count + 1)

main :: IO ()
main = print $ average [1..1000000]
```

My advice is to use a helper data type with strict fields. But perhaps you don't want to do that, and you're frustrated that there is no `foldl'` that evaluates to normal form. Fortunately for you, by just throwing in a call to `force`, you can easily upgrade a WHNF fold into a NF fold:

```
import Data.List (foldl')
import Control.DeepSeq (force)

average :: [Int] -> Double
average =
  divide . foldl' add (0, 0)
  where
    divide (total, count) = fromIntegral total / count
    add (total, count) x = force (total + x, count + 1)

main :: IO ()
main = print $ average [1..1000000]
```

Like a good plumber, `force` patches that leak right up!

Streaming data

One of the claims of streaming data libraries (like conduit) is that they promote constant memory usage. This may make you think that yo can get away without worrying about space leaks. However, all of the comments about WHNF vs NF mentioned above apply. To prove the point, let's do average badly with conduit:

```
import Conduit

average :: Monad m => ConduitM Int o m Double
average =
  divide <$> foldlC add (0, 0)
  where
    divide (total, count) = fromIntegral total / count
    add (total, count) x = (total + x, count + 1)

main :: IO ()
main = print $ runConduitPure $ enumFromToC 1 1000000 .| average
```

You can test the memory usage of this with:

```
$ stack --resolver lts-12.21 ghc --package conduit-combinators -- Main.hs -O2
$ ./Main +RTS -s
```

EXERCISE Make this program run in constant resident memory, by using:

1. The `force` function
2. Bang patterns
3. A custom data type with strict fields

Chain reaction

Look at this super strict program. It's got a special value-strict list data type. I've liberally sprinkled bang patterns and calls to `seq` throughout. I've used `$!`. How much memory do you think it uses?

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE BangPatterns #-}

data StrictList a = Cons !a !(StrictList a) | Nil

strictMap :: (a -> b) -> StrictList a -> StrictList b
strictMap _ Nil = Nil
strictMap f (Cons a list) =
  let !b = f a
      !list' = strictMap f list
   in b `seq` list' `seq` Cons b list'

strictEnum :: Int -> Int -> StrictList Int
strictEnum low high =
  go low
  where
    go !x
      | x == high = Cons x Nil
      | otherwise = Cons x (go $! x + 1)

double :: Int -> Int
double !x = x * 2

evens :: StrictList Int
evens = strictMap double $! strictEnum 1 1000000

main :: IO ()
main = do
  let string = "Hello World"
      string' = evens `seq` string
  putStrLn string
```

Look carefully, read the code well, and make a guess. Ready? Good.

It uses 44kb of memory. "What?!" you may exclaim. "But this thing has to hold onto a million `Int`s in a strict linked list!" Ehh... almost. It's true, our program is going to do a hell of a lot of evaluation as soon as we force the `evens` value. And as soon as we force the `string'` value in `main`, we'll force `evens`.

However, our program never actually forces evaluation of either of these! If you look carefully, the last line in the program uses the `strir` value. It never looks at `string'` or `evens`. When executing our program, GHC is only interested in performing the `IO` actions it is told to perform by the `main` function. And `main` only says something about `putStrLn string`.

This is vital to understand. You can build up as many chains of evaluation using `seq` and `deepseq` as you want in your program. But ultimately, unless you force evaluation via some `IO` action of the value at the top of the chain, it will all remain an unevaluated thunk.

EXERCISES

1. Change `putStrLn string` to `putStrLn string'` and see what happens to memory usage. (Then undo that change for the other exercises.)
2. Use a bang pattern in `main` somewhere to get the great memory usage.
3. Add a `seq` somewhere in the `putStrLn string` line to force the greater memory usage.

Contact Us

Corporate Office
10130 Perimeter
Parkway
Suite 200
Charlotte, NC 28216

+1 858-617-0430

sales@fpcomplete.com

Services

- Custom Software Development
- DevSecOps
- Blockchain
- Rust
- Haskell
- Training
- All Services

Products

- Kube360®
- Zehut
- Konsole360
- Idiom
- Kafka Library

Resources

- Blog Posts
- Video Library
- Case Studies
- White Papers

Our Company

- Our Journey
- Our Mission
- Our Leadership
- Our Engineers
- Our Clients
- Jobs

Privacy Policy