Data types

Strictness annotations

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Data.Foldable (foldl')

data Foo = Foo Int
  deriving Show
data Bar = Bar !Int
  deriving Show
newtype Baz = Baz Int
  deriving Show

main :: IO ()
main = do
  print $ foldl'
    (\(Foo total) x -> Foo (total + x))
    (Foo 0)
    [1..1000000]
  print $ foldl'
    (\(Bar total) x -> Bar (total + x))
    (Bar 0)
    [1..1000000]
  print $ foldl'
    (\(Baz total) x -> Baz (total + x))
    (Baz 0)
    [1..1000000]
```

- Lots of thunk allocation with `Foo`
- What about `Bar` vs `Baz`? A few differences.

Advantages of strictness annotations:

- Avoid space leaks
- No hidden bottom values
- GHC errors out when you forget a field

Recommendation: if you don't need laziness in a field, make it strict.

Pattern matching

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Data.Foldable (foldl')
import UnliftIO.Exception (pureTry)

data Foo = Foo Int
  deriving Show
data Bar = Bar !Int
  deriving Show
newtype Baz = Baz Int
  deriving Show

main :: IO ()
main = do
  print $ pureTry $
    case Foo undefined of
      Foo _ -> "Hello World"
  print $ pureTry $
    case Bar undefined of
      Bar _ -> "Hello World"
  print $ pureTry $
    case Baz undefined of
      Baz _ -> "Hello World"
```

Memory layout

- `Foo` contains:
  - Data constructor (one word)
  - Pointer to `Int` (one word)
  - `Int` has a data constructor (one word)
  - `Int` has a payload `Int#` (we'll get to later, one word)
- `Bar` in theory has the exact same thing, but wait till next section
  - Strictness does not directly affect memory layout
- `Baz` is a newtype, guaranteed to have no runtime representation
  - `Int` itself is still two words
  - In ideal situations, GHC can avoid the data constructor, we'll say that with GHC core later

Unpack

That extra `Int` data constructor is annoying, get rid of it!

```
data Bar = Bar {-# UNPACK #-} !Int
```

- Inlines the contents of `Int` into the `Bar` representation
- Avoids extra headers and pointers
- Only works on:
  - Strict fields
  - Non-polymorphic fields
  - Type must have a single data constructor types
  - **Question** Why?
- Bonus: applied automatically to primitive types (like `Int`)

- In practice: don't need `{-# UNPACK #-}` on `Int`
  - Personally, I'll still add it for clarity often

Why not always unpack fields? It can be a pessimization with large data types due to copying lots of data instead of copying a single pointer. If the value is a machine word, it's *always* better to unpack, thus the primitive type optimization.

What's in an Int?

`Int` is defined in normal Haskell code, it's not a GHC built-in. Don't believe me?

https://www.stackage.org/haddock/lts-12.21/ghc-prim-0.5.2.0/GHC-Types.html#t:Int

```
data Int = I# Int#
data Word = W# Word#
```

Magic hash!

```
$ stack exec -- ghci -XMagicHash
GHCi, version 8.0.1: https://www.haskell.org/ghc/  :? for help
Prelude> import GHC.Prim
Prelude GHC.Prim> :k Int#
Int# :: TYPE 'GHC.Types.IntRep
```

`Int#` is the magic, built-in value provided by GHC, in the ghc-prim package.

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE MagicHash #-}
import GHC.Prim
import GHC.Types

main :: IO ()
main = print $ I# (5# +# 6#)
```

Going low level

High level, good code:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script

main :: IO ()
main = print $ sum [1..100 :: Int]
```

Hopefully GHC optimizes this into a tight loop. But let's write that tight loop manually:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE BangPatterns #-}
main :: IO ()
main = print $ loop 0 1
  where
    loop !total i
      | i > 100 = total
      | otherwise = loop (total + i) (i + 1)
```

- Why did I put a bang on `total` but not `i`?
- Can you rewrite that function without the bang?

OK, let's get primitive!

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE MagicHash #-}
import GHC.Prim
import GHC.Types

main :: IO ()
main = print $ I# (loop 0# 1#)
  where
    loop total i
      | isTrue# (i ># 100#) = total
      | otherwise = loop (total +# i) (i +# 1#)
```

- Basically no need to ever do this
- GHC should always be smart enough to do it for you
- But good to understand that GHC is optimizing to this
- Important for reading core

Sum type representation

- Data constructor: one word, tells us which constructor
- Remaining fields follow

Example:

```
data Foo = Bar !Int !Int | Baz !Int | Qux
```

How much memory needed for:

- `Bar 5 6`
- `Baz 5`
- `Qux`

Constructors with no fields (like `Qux` or `Nothing`): one copy in memory shared by all usages.

Compare the following:

```
data Result = Success !Int | Failure
data MaybeResult = SomeResult !Result | NoResult
```

Versus:

```
data MaybeResult = Success !Int
                 | Failure
                 | NoResult
```

- Semantically: identical
- Calculate memory representation of each

Takeaway: if performance is crucial, consider "inlining" layered sum types. Downside:

Pointer tagging

https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/HaskellExecution/PointerTagging

- Pointers are always machine-word aligned
- That means lowest 2 bits (3 bits on 64-bit archs) are always 0
- So... encode some useful info there!
    - All zeros == thunk
    - Other value tells you the constructor type
    - Can avoid a pointer indirection when `case`ing
- Pointer tagging only works for less than 4 data constuctors (8 on 64 bit)