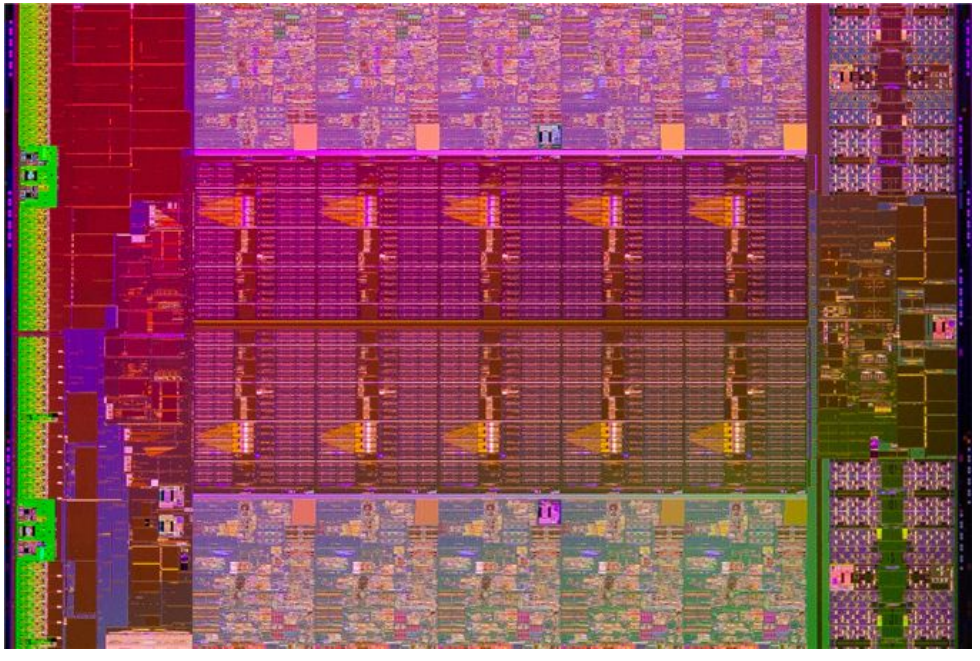
Practical Haskell Performance

Tim McGilchrist

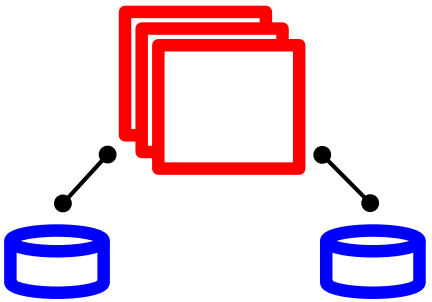
Introduction

Hardware



Typical Pipeline Program

Reads data off storage, does some processing and then writes back to storage.



Cost

Because we use AWS, there is a direct link between a days computing and the angriness of our pointy haired boss.

Optimising

"The art of poking stacks, trawling asm and black magic."

Haskell IO

100% of time spent reading and writing to disk?

Haskell IO

100% of time spent reading and writing to disk?

This raises two issues

1. How do we efficiently read in enough data without blowing our memory budget?
2. Is it safe? What about IO Exceptions and cleaning up?

ByteString Lazy vs cat

Using straight ByteString lazy

```
runStatisticsBSL :: InputPath -> OutputPath -> IO ()
runStatisticsBSL (InputPath i) (OutputPath o) =
  BSL.readFile i >>= BSL.writeFile o
```

ByteString Lazy vs cat

Using straight ByteString lazy

```
runStatisticsBSL :: InputPath -> OutputPath -> IO ()
runStatisticsBSL (InputPath i) (OutputPath o) =
  BSL.readFile i >>= BSL.writeFile o
```

```
> ./hatchet sample.psv output.psv
\ 2.72s user 35.20s system 53% cpu 1:10.79 total
```

ByteString Lazy vs cat

Using straight ByteString lazy

```
runStatisticsBSL :: InputPath -> OutputPath -> IO ()
runStatisticsBSL (InputPath i) (OutputPath o) =
  BSL.readFile i >>= BSL.writeFile o
```

```
> ./hatchet sample.psv output.psv
\ 2.72s user 35.20s system 53% cpu 1:10.79 total
```

```
> time cat sample.psv > output.psv
\ 1.70s user 49.06s system 65% cpu 1:17.49 total
```

Conduit

- Promises composable building blocks
- Constant memory usage on large / infinite streams

Conduit

- Promises composable building blocks
- Constant memory usage on large / infinite streams

```
runStatisticsC :: InputPath -> OutputPath -> IO ()
runStatisticsC (InputPath i) (OutputPath o) =
  runResourceT . runConduit $
    CB.sourceFile i =$= CB.sinkFile o
```

Conduit

- Promises composable building blocks
- Constant memory usage on large / infinite streams

```
runStatisticsC :: InputPath -> OutputPath -> IO ()
runStatisticsC (InputPath i) (OutputPath o) =
  runResourceT . runConduit $
    CB.sourceFile i =$= CB.sinkFile o
```

```
> ./hatchet sample.psv output.psv
\ 4.46s user 34.81s system 49% cpu 1:19.35 total
```

Data Types

```haskell
data Row = Row {
    rowId    :: RowId
  , rowSpend :: Double
  , rowItems :: Int
  , rowDate  :: UTCTime
  , rowState :: Text
  } deriving (Eq, Show)
```

Each row in our file gets parsed into this structure.

Data Types

```haskell
data Row = Row {
    rowId    :: RowId
  , rowSpend :: Double
  , rowItems :: Int
  , rowDate  :: UTCTime
  , rowState :: Text
  } deriving (Eq, Show)
```

Each row in our file gets parsed into this structure.

```
some-id | 150.0 | 5 | 10/3/2016 | NSW
```

Data Type Size

```
some-id | 150.0 | 5 | 10/3/2016 | NSW
```

Raw size of a line is 38 Bytes.

Data Type Size

```
some-id | 150.0 | 5 | 10/3/2016 | NSW
```

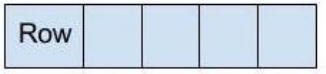
Raw size of a line is 38 Bytes.

```
  rowId    :: RowId  -- 6 words + 2N bytes
, rowSpend :: Double -- 3 words (2 on 32 bit)
, rowItems :: Int    -- 2 words
, rowDate  :: UTCTime -- 6 words
, rowState :: Text   -- 6 words + 2N bytes
```

Becomes 43 words * 8 or 344 Bytes on 64 bit machine.

Memory Layout

- Value Representation

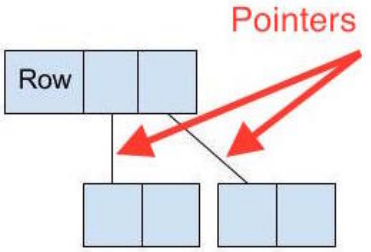



```haskell
data Row = Row {
    rowId    :: RowId
  , rowSpend :: Double
  , rowItems :: Int
  ...
  } deriving (Eq, Show)
```

Memory Layout

- Value Representation

- Reference Representation



Optimising via newtype

```haskell
newtype RowId = RowId Text deriving (Eq, Show)
```

- constructors defined via newtype are free
- purely a compile-time idea
- takes no space and costs no instructions

Boxed vs Unboxed Values

What are they?

- Boxed
- - most types in Haskell
  - represented by a pointer to a heap object
- Unboxed
  - primitive values e.g. raw machine types
  - cannot be directly defined

UNPACK All The Things

As a general rule, scalar fields, such as Int and Double, should always be unpacked.

UNPACK All The Things

As a general rule, scalar fields, such as Int and Double, should always be unpacked.

```
data Row = Row {
    rowId    :: {-# UNPACK #-} RowId
  , rowSpend :: {-# UNPACK #-} Double
  , rowItems :: {-# UNPACK #-} Int
  , rowDate  :: {-# UNPACK #-} UTCTime
  , rowState :: {-# UNPACK #-} Text
  } deriving (Eq, Show)
```

UNPACK All The Things

As a general rule, scalar fields, such as Int and Double, should always be unpacked.

```
data Row = Row {
    rowId    :: {-# UNPACK #-} RowId
  , rowSpend :: {-# UNPACK #-} Double
  , rowItems :: {-# UNPACK #-} Int
  , rowDate  :: {-# UNPACK #-} UTCTime
  , rowState :: {-# UNPACK #-} Text
  } deriving (Eq, Show)
```

- Syntatically noisey

Better Unpacking

- Use `-funbox-strict-fields` for all files which contain data types.

```
-- other language pragmas --
{-# OPTIONS_GHC -funbox-strict-fields #-}
module Hatchet.Data (Row(..)) where
```

How do I know?

Scalar fields can always be unpacked.

Things that can't be unpacked:

- Sum Types (Types with many constructors)
- Polymorphic fields (How would you know what's there?)

How do I know?

Scalar fields can always be unpacked.

Things that can't be unpacked:

- Sum Types (Types with many constructors)
- Polymorphic fields (How would you know what's there?)

Try draw boxes representing your data types, think about how they might be represented in memory.

Strict Data Types

By default always use strict data types.

Strict Data Types

By default always use strict data types.

Haskell being lazy, why should this be the case?

Strict Data Types

By default always use strict data types.

Haskell being lazy, why should this be the case?

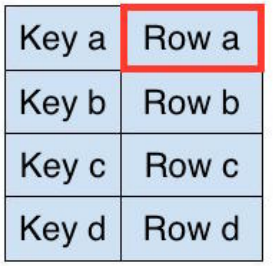
Think about repeated adding a value to a lazy data structure.

Strict Data Types

By default always use strict data types.

Haskell being lazy, why should this be the case?

Think about repeated adding a value to a lazy data structure.



Strict Row

```haskell
data Row = Row {
    rowId    :: !RowId
  , rowSpend :: !Double
  , rowItems :: !Int
  , rowDate  :: !UTCTime
  , rowState :: !Text
  } deriving (Eq, Show)
```

- Add bang patterns to everything

Strict Recursive

```
data Exp = Infix !Op Exp Exp
         | Constant !Type !Value

data Op = Add | Subtract | Multiply | Divide
```

- select the non-recursive parts to make strict

Data Summary

We've covered a bunch of things to optimise your data structures:

- Choosing the right types
- use newtype liberally
- make data structures strict
- unpacking data types

Final Data Type

```haskell
-- other language pragmas --
{-# OPTIONS_GHC -funbox-strict-fields #-}
module Hatchet.Data (Row(..)) where

data Row = Row {
    rowId    :: !RowId
  , rowSpend :: !Double
  , rowItems :: !Int
  , rowDate  :: !UTCTime
  , rowState :: !Text
  } deriving (Eq, Show)
```

Functions

Why Lazy Functions?

Consider this code

```
any :: (a -> Bool) -> [a] -> Bool
any p = or . map p
```

Forcing Computation

Consider this computation:

```
sum :: [Row] -> Int
sum =
  let loop acc = \case
      [] -> acc
      (x:xs) ->
        loop (acc + (rowSpend x)) xs
  in loop 0
```

Forcing Computation

Sometimes it does make sense to force the arguments to a function.

```
sum :: [Row] -> Int
sum =
  let loop !acc = \case      -- Force acc
      [] -> acc
      (x:xs) ->
        loop (acc + (rowSpend x)) xs
  in loop 0
```

Remove Polymorphism

Imagine we wrote our code with this type signature

```
add :: Real a => a -> a -> a
add rowSpend acc = acc + rowSpend
```

Remove Polymorphism

Imagine we wrote our code with this type signature

```
add :: Real a => a -> a -> a
add rowSpend acc = acc + rowSpend
```

```
add :: Int -> Int -> Int
```

Specialise

If however it really is used polymorphically you can use `SPECIALIZE` to give GHC a hint of what it's used for

```
{-# SPECIALIZE add :: Int -> Int -> Int #-}
{-# SPECIALIZE add :: Double -> Double -> Double #-}
add :: Real a => a -> a -> a
```

Tools

- Dump RTS statistics
  - raw information like GC Time, Memory usage
  - add `-rtsopts` to ghc-options in cabal
  - run with `+RTS -sstderr`

Tools

- Dump RTS statistics
  - raw information like GC Time, Memory usage
  - add `-rtsopts` to ghc-options in cabal
  - run with `+RTS -sstderr`
- Time profiling
  - shows where program is spending time
  - add `-prof -auto-all -caf-all -fforce-recomp`
  - run with `+RTS -p`

Tools

- Dump RTS statistics
  - raw information like GC Time, Memory usage
  - add `-rtsopts` to ghc-options in cabal
  - run with `+RTS -sstderr`
- Time profiling
  - shows where program is spending time
  - add `-prof -auto-all -caf-all -fforce-recomp`
  - run with `+RTS -p`
- Space profiling
  - shows memory usage over time
  - various options available
  - differ in how the live heap is broken down

Tools

- Dump RTS statistics
  - raw information like GC Time, Memory usage
  - add `-rtsopts` to ghc-options in cabal
  - run with `+RTS -sstderr`
- Time profiling
  - shows where program is spending time
  - add `-prof -auto-all -caf-all -fforce-recomp`
  - run with `+RTS -p`
- Space profiling
  - shows memory usage over time
  - various options available
  - differ in how the live heap is broken down
- Criterion
  - framework for executing and analysing benchmarks
  - pretty output and graphs

Conclusion

By default, use strict data types and lazy functions.

- Think about the evaluation model
- Plan your data types
- Force evaluation where necessary
- Measure performance

Thanks!

Tim McGilchrist @lambda_foo

Master Commander of Big Data @ Ambiata

Thanks!