47

Yin and Yang of Haskell

LambdaWorld Cádiz 2019

Alejandro Serrano Mena

Welcome!

The quest for being a Haskeller

Learn more and more abstractions?

The quest for being a Haskeller

Learn more and more abstractions?

- The simple abstractions get you a long way!

The quest for being a Haskeller

Learn more and more abstractions?

- The simple abstractions get you a long way!

You need to know more about the *run-time* system.

- When and how is my program executed?
- When and how is my memory allocated?
- How do I communicate with other resources?

Yin and Yang of Haskell

Each side influences the other:

- Laziness is possible because of purity;
- Laziness forces us to write purer programs.

Yin and Yang of Haskell

Each side influences the other:

- Laziness is possible because of purity;
- Laziness forces us to write purer programs.

- We can track effects because of the rich type system;
- Our aim to track finer-grained effects leads to richer types.

Yin and Yang of Haskell

Each side influences the other:

- Laziness is possible because of purity;
- Laziness forces us to write purer programs.

- We can track effects because of the rich type system;
- Our aim to track finer-grained effects leads to richer types.

We need to know about the two sides.

Practicalities

The workshop is organized in blocks:

- Some time of me talking and **you asking questions**.
- Followed by some time of you coding.

Drinks and food will be served here:

- With a longer break for lunch around 13.00.

`github.com/47deg/yin-yang-haskell-workshop`

Yin: Pure modelling with ADTs

Our domain: Pen-and-paper role-playing game

Pathfinder is a fork of Dungeons & Dragons 3.5.

- *Character Sheet*: Quite complex.
  - Absolute *scores* lead to *modifiers*.
  - These modifiers are used when throwing dice.
  - And may be supplemented by skills or ranks.
- *Combat*: A lot of possibilities.
  - Move, attack, use spell...
  - They take different amounts of time.

Modelling characters

Within the game we have different kinds of **characters**:

- Players, which have the most detailed stats.
- Non-playing characters (NPCs):
  - Monsters and other figthers, which just need enough information to execute combat.
  - Other characters with only basic stats.

The game and the story is directed by a **(game) master**.

Modelling characters

(Following Wlaschin's terminology)

```
A character is...
      a player  with basic stats AND combat stats AND skills
  OR a fighter with basic stats AND combat stats
  OR a NPC     with basic stats
```

Modelling characters

(Following Wlaschin's terminology)

```
A character is...
      a player  with basic stats AND combat stats AND skills
  OR a fighter with basic stats AND combat stats
  OR a NPC     with basic stats
```

```haskell
data Character
  = Player  BasicStats CombatStats Skills
  | Fighter BasicStats CombatStats
  | NPC     BasicStats
```

Pattern matching

We access the information by using different patterns:

```
basicStats :: Character -> BasicStats
basicStats c = case c of
                 Player  s _ _ -> s
                 Fighter s _   -> s
                 NPC     s     -> s
```

Pattern matching

We access the information by using different patterns:

```
basicStats :: Character -> BasicStats
basicStats c = case c of
                 Player  s _ _ -> s
                 Fighter s _   -> s
                 NPC     s     -> s

-- functions have an implicit top-level case
basicStats (Player  s _ _) = s
basicStats (Fighter s _)   = s
basicStats (NPC     s)     = s
```

Sums / choices / (discriminated, tagged) unions

This is an important difference with OOP:

- Do not think of open, extensible hierarchies;
- But of a closed set of possibilities.

Sums / choices / (discriminated, tagged) unions

This is an important difference with OOP:

- Do not think of open, extensible hierarchies;
- But of a closed set of possibilities.

Advantages

- *Modelling*: No fear of forward compatibility.
  - What if somebody overrides this method?
- *Compilation*: Ability to check for exhaustive coverage.

More examples of sums

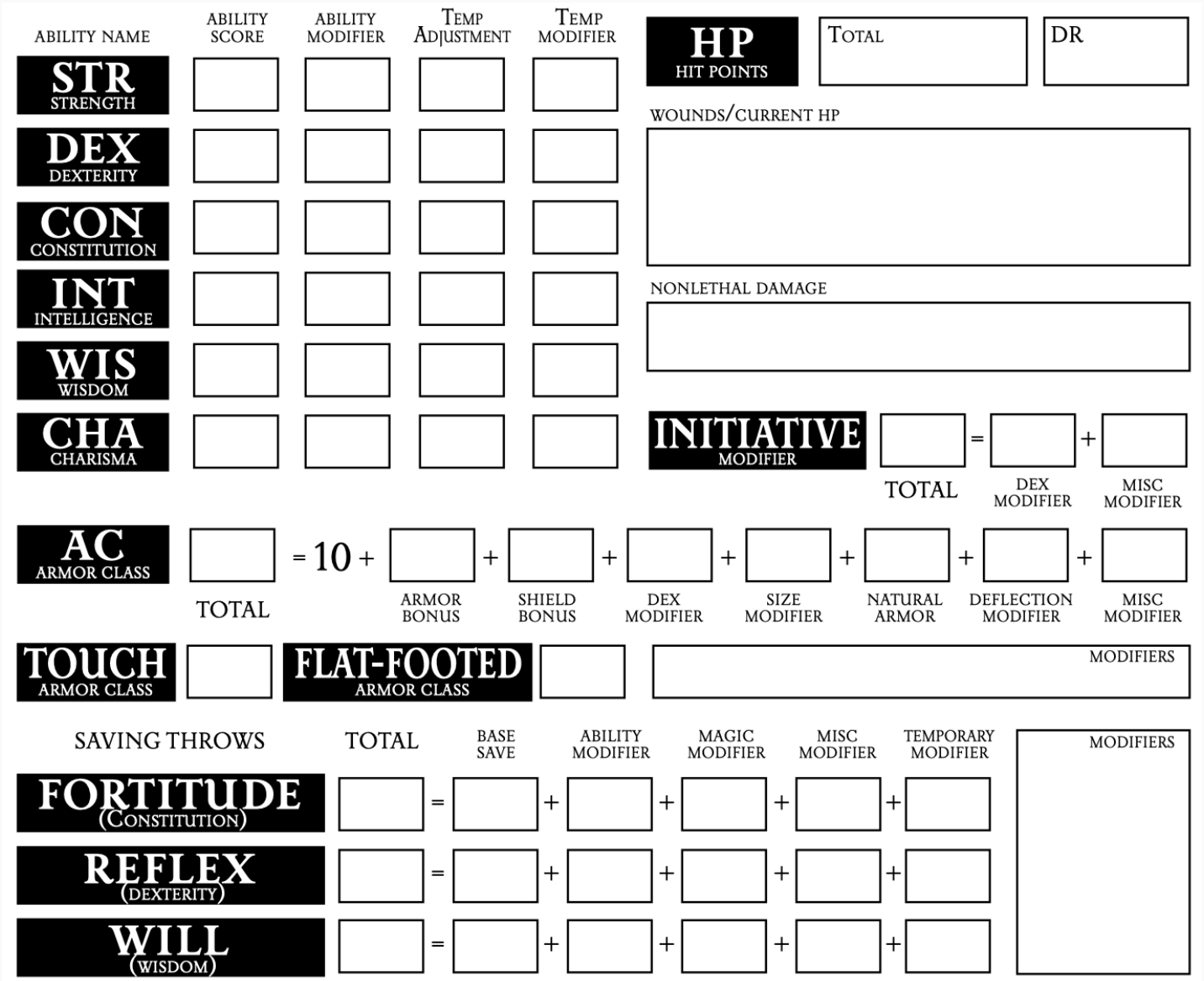
- States that a program or value may be in.
  - During combat, a character may be taking part, have fainted (0 HP), or be dead (-10 HP).
- Actions or commands that may be taken.
- Events that may arise.
  - Ties very well with CQRS or event sourcing.

More examples of sums

- States that a program or value may be in.
  - During combat, a character may be taking part, have fainted (0 HP), or be dead (-10 HP).
- Actions or commands that may be taken.
- Events that may arise.
  - Ties very well with CQRS or event sourcing.

In short, sum up your data types!

Modelling nightmare



Absolute values versus modifiers

```
data BasicStats
  = BasicStats { str :: Int, dex :: Int, con :: Int
               , int :: Int, wis :: Int, cha :: Int
               , fort :: Int, refl :: Int, will :: Int
               , ... }
```

Absolute values versus modifiers

```
data BasicStats
  = BasicStats { str :: Int, dex :: Int, con :: Int
               , int :: Int, wis :: Int, cha :: Int
               , fort :: Int, refl :: Int, will :: Int
               , ... }
```

Record syntax: Stating a name for each field in a constructor.

- Generates accessor functions for free.

```
str :: BasicStats -> Int
dex :: BasicStats -> Int
```

- Gives you extra powers for pattern match.

```
f BasicStats { str = s, dex = d } = ...
```

- Mostly used for single-constructor data types.

Absolute values versus modifiers

```
data BasicStats
  = BasicStats { str :: Int, dex :: Int, con :: Int
               , int :: Int, wis :: Int, cha :: Int
               , fort :: Int, refl :: Int, will :: Int
               , ... }
```

We have used `Int` for everything, yet:

- Absolute values in Pathfinder never go below 0.
- In contrast, modifiers usually do.
- You can apply a modifier to an absolute value, but the combination of absolute values does not make sense.

One type per notion in your domain

In Haskell, you use a `newtype` to wrap an existing type:

```haskell
newtype AbsoluteValue = AbsoluteValue { unAbsoluteValue :: Int }
newtype Modifier      = Modifier { unModifier :: Int }

applyModifier :: AbsoluteValue -> Modifier -> AbsoluteValue
applyModifier (AbsoluteValue v) (Modifier m)
  = AbsoluteValue (v + m)
-- now using field accessors
applyModifier v m
  = AbsoluteValue (unAbsoluteValue v) (unModifier m)
```

Wrapping and unwrapping `newtypes` is memory and time free!

Derived combat statistics

We have given both `Players` and `Fighers` combat statistics, but in players, combat stats are derived from their skills.

```
data Character
  = Player  BasicStats Skills
  | Fighter BasicStats CombatStats
  | NPC     BasicStats
```

How to treat `Player` and `Figher` uniformly during combat?

View patterns

1. A new data type to express exactly what you need:

```haskell
data Combatant  -- cannot reuse Fighter :(
  = Combatant CombatStats
  | NonCombatant
```

2. A function to turn one into the other:

```haskell
asCombatant :: Character -> Combatant
asCombatant (Player stats skills) = Combatant ...
asCombatant (Figher _ cstats)     = Combatant cstats
asCombatant (NPC _)               = NonCombatant
```

View patterns

3. *View patterns* to apply that transformation before matching:

```
punch :: Character -> Character -> ...
punch one (asCombatant -> Combatant cstats) = ...
punch one (asCombatant -> NonCombatant)     = ...
```

Write as many views as you need for your types.

Optics

Another way to view types differently is by means of an *optic*.

- *Lens*: Access and modify a value that is always there.
  - Like the `BasicStats` in a `Character`.
- *Prism*: Access and modify a possibly-missing value.
  - Like `CombatStats` in a `Character`.
- Many others!

Main advantage: optics are composable.

Optics

Another way to view types differently is by means of an *optic*.

- *Lens*: Access and modify a value that is always there.
  - Like the `BasicStats` in a `Character`.
- *Prism*: Access and modify a possibly-missing value.
  - Like `CombatStats` in a `Character`.
- Many others!

Main advantage: optics are composable.

But, unfortunately, we do not have the time to cover them.

- Read `lens-tutorial` and then use `optics`.

Equipment is an open notion

`Equipment` does not lend itself to a closed representation:

- Lots of choices:

```
data Equipment = SmallSword  ...
               | MediumSword ...
               | SpellWand   ...
               | ... -- thousands of choices
```

- And they might grow in the future!

OOP says: New derived classes. What about FP?

Categorize and parametrize

```
data Equipment
  = Weapon { hit :: AbsoluteValue, ... }
  | MagicObject { execute :: Character -> Character, ... }
  | ...
```

Functions as data provide a way to model different behavior. But:

- Functions are hard to inspect,
- As a result, functions are hard to serialize.

More on programs-as-data afterwards!

In summary...

Modelling data with Algebraic Data Types:

- Think in terms of *sums*, not of class hierarchies;
- Use *one type per concept* in the domain;
- It is helpful to see the same data from different *views*,
  - View patterns and optics help you with this;
- *Open* data benefits from functional fields.

Time to code!

Exercise 1: *Modelling a Game*

- Speak and discuss with your peers,
- I might listen to the conversation and jump in :P

Exercise 0: *Game using Gloss*

Feel free to ask questions about the pre-workshop exercise.

Yang: Impure execution

FP compilers are magical creatures

GHC has no less than 3 intermediate languages.

Haskell $\rightarrow$ Core $\rightarrow$ STG $\rightarrow$ C– $\rightarrow$ (LLVM) $\rightarrow$ assembly

FP compilers are magical creatures

GHC has no less than 3 intermediate languages.

Haskell $\rightarrow$ Core $\rightarrow$ STG $\rightarrow$ C– $\rightarrow$ (LLVM) $\rightarrow$ assembly

STG = Spineless Tagless Graph Reduction Machine

The main difference

- Most languages have a strict order of execution:
  - For each function call, execute the code fully to obtain the arguments, then pass the resulting values to the function body.
  - Predictable, but potentially costly.
- Haskell has a lazy execution model:
  - Code is only executed "when needed."
  - It mostly works well, but when it doesn't...
  - How does it work when you communicate with the outside world?

GHC/Haskell's execution model

Some examples

Consider the function:

```
f []     = "empty"
f (_:_) = "non-empty"
```

Some examples

Consider the function:

```
f []    = "empty"
f (_:_) = "non-empty"
```

To make things more clear, we rewrite this as:

```
f xs = case xs of []    -> "empty"
                  (_:_) -> "non-empty"
```

Some examples

Consider a `takesLong` function that takes a lot of time to execute:

```
takesLong = go 10000000000000000
  where go 0 = []
        go n = go (n-1)
```

The question is, what happens when we do:

```
f takesLong
```

Some examples

Strict evaluation (the usual one):

```
  f takesLong
= -- execute takesLong
  f []
= "empty"
```

Lazy evaluation (Haskell's):

```
  f takesLong
= case takesLong of []    -> "empty"
                    (_:_) -> "non-empty"
= -- execute takesLong
  "empty"
```

Unused arguments

The `const` function forgets about its second argument.

```
const x y = x
```

Unused arguments

The `const` function forgets about its second argument.

```
const x y = x
```

Strict evaluation:

```
  const 3 takesLong
= -- execute 3 and takesLong
= 3
```

Lazy evaluation:

```
  const 3 takesLong
= 3  -- takesLong is not needed!
```

Making “when needed” precise

Haskell functions are only executed when we need to inspect the value to choose how to continue.

- In other words, when we are in a `case` statement.

Making “when needed” precise

Haskell functions are only executed when we need to inspect the value to choose how to continue.

- In other words, when we are in a `case` statement.

Even better, functions are only executed “as much as needed.”

Takes long in a different way

```
headTakesLong = [ go 10000000000000000, 2 ]
  where go 0 = 0 ; go n = go (n-1)
```

Takes long in a different way

```
headTakesLong = [ go 10000000000000000, 2 ]
  where go 0 = 0 ; go n = go (n-1)
```

Strict evaluation:

```
  f headTakesLong
= -- execute headTakesLong fully
  "non-empty"
```

Lazy evaluation:

```
  f headTakesLong = case headTakesLong of ...
= case [ go 10000000000000000, 2 ] of []    -> "empty"
                                      (_:_) -> "non-empty"
= "non-empty"  -- go 10000000000000000 is never executed!
```

Short-circuiting = lazy evaluation

Most C-like languages distinguish between `&` and `&&`.

```
False && _ = False
_     && y = y

x && y = case x of
           False -> False
           _     -> y
```

- `y` is only executed if `x` is `False`.

Build your own control structures

```
if_ True  t e = t
if_ False t e = e
```

In a strict language, `if_` does not work as a real `if`.

- Both arguments are executed, which defeats the purpose.
- We only want *one* of them to be executed.

In Haskell, `if_` is really an `if-then-else`.

Infinite loops can be avoided

```
infiniteLoop :: a -> b
infiniteLoop x = infiniteLoop x

f (infiniteLoop 3) -- does not terminate

  const 3 (infiniteLoop 3)
= 3

  f [ infiniteLoop 3, 2 ]
= case [ infiniteLoop 3, 2 ] of ...
= "non-empty"
```

Infinite structures can be manipulated

```haskell
-- allNumbers = [0, 1, 2, 3, ...]
allNumbers = go 0
  where go n = n : go (n + 1)
```

Infinite structures can be manipulated

```haskell
-- allNumbers = [0, 1, 2, 3, ...]
allNumbers = go 0
  where go n = n : go (n + 1)

  take 2 allNumbers
= [0, 1]  -- works!
```

Infinite structures can be manipulated

```haskell
-- allNumbers = [0, 1, 2, 3, ...]
allNumbers = go 0
  where go n = n : go (n + 1)

  take 2 allNumbers
= [0, 1]  -- works!

  take 2 (map (\x -> x * 3) allNumbers)
= [0, 3] -- works!
```

Enough magic!

Show me the trick!

Thunks

Each expression yet-to-be-executed is kept in a *thunk*:

- Includes the code to be executed, and
- References to any other required thunks.

Thunks

Each expression yet-to-be-executed is kept in a *thunk*:

- Includes the code to be executed, and
- References to any other required thunks.

`case` statements *force* thunks:

- The code inside of it is executed,
- Until a constructor appears on the top of the value,
  - At that point, `case` can take a choice,
- The thunk is updated in case it is needed afterwards.

Sharing thunks

```
let x = takesLong in (f x, null x)
```

- The first time we need to inspect one of the elements of the tuple, `takesLong` is executed.
- After the first execution, this thunk is updated to `[]`.
- Any subsequent uses of `x` return immediately.

Sharing thunks

```
let x = takesLong in (f x, null x)
```

- The first time we need to inspect one of the elements of the tuple, `takesLong` is executed.
- After the first execution, this thunk is updated to `[]`.
- Any subsequent uses of `x` return immediately.

Advantage: Haskell does the minimal amount of work.

Disadvantage: Potential thunks in memory which we never use.

- As a result, garbage collection must consider thunks.
  - GC looks at your data and your *code*.

Problems in paradise

Space leak: Your thunks grow bigger, or live longer than expected.

Problems in paradise

Space leak: Your thunks grow bigger, or live longer than expected.

The #1 performance bug in Haskell:

- (Moderably) easy to spot: Your code is slow, and your memory usage skyrockets.
- Hard to debug: You must consider both the definition of your code, and also every place where it is used.

Fold starting from the head

```
foldl _ v []     = v
foldl f v (x:xs) = foldl f (f v x) xs

foldl (-) 0 [1,2,3] = ((0 - 1) - 2) - 3
foldr (-) 0 [1,2,3] = 1 - (2 - (3 - 0))
```

Fold starting from the head

```
foldl _ v []     = v
foldl f v (x:xs) = foldl f (f v x) xs

foldl (-) 0 [1,2,3] = ((0 - 1) - 2) - 3
foldr (-) 0 [1,2,3] = 1 - (2 - (3 - 0))
```

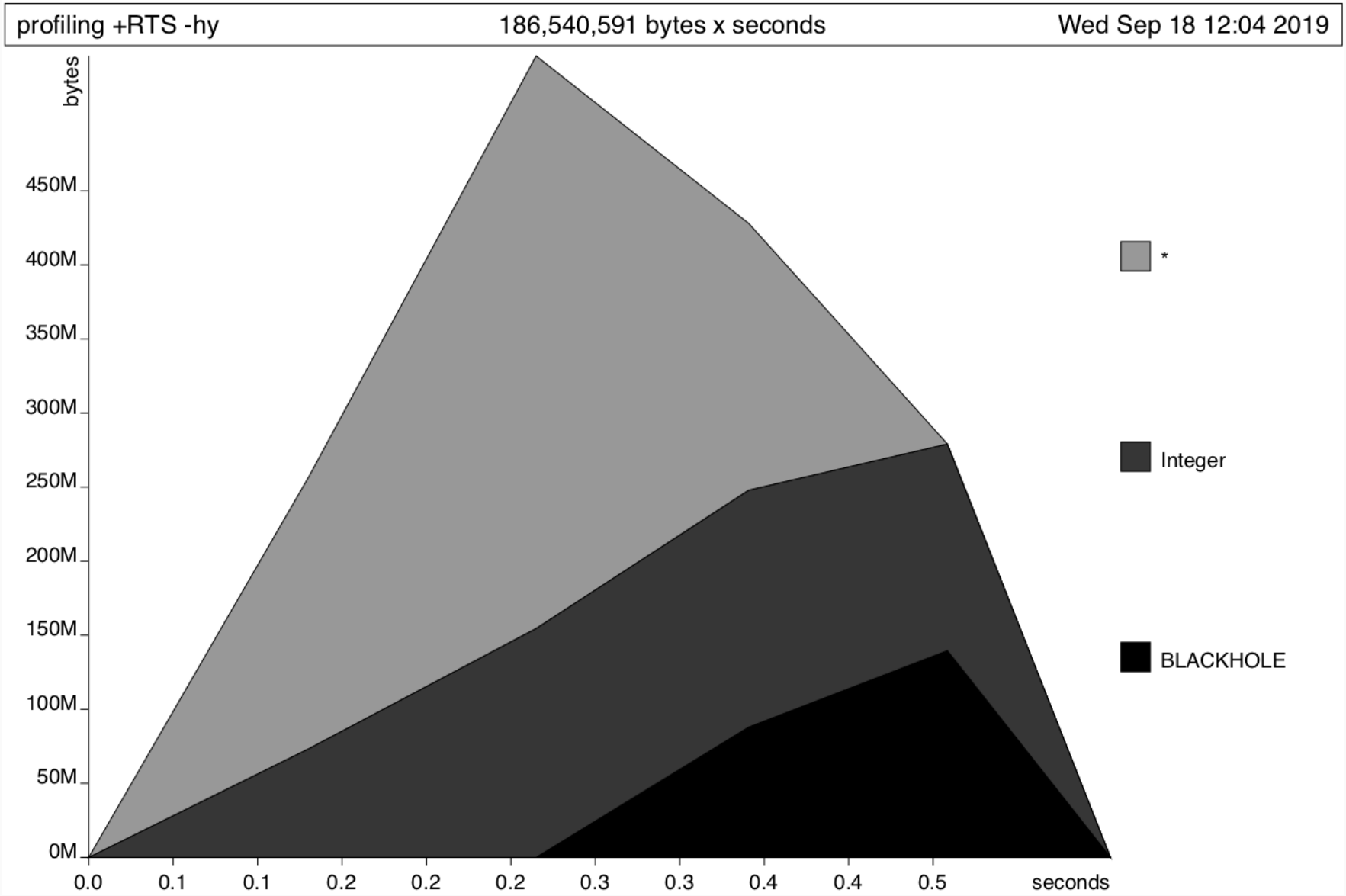
Let's do something simple (for a computer):

```
main = print (foldl (+) 0 [1 .. 10000000])
```

Profiling

1. Create a Cabal project with an executable.
2. Make the previous expression to be the `main`.
3. `stack build --profile`.
4. `long/path/to/your/bin +RTS -hy`.
5. `hp2ps <your-bin>.hp`
6. Open the `<your-bin>.ps` file.

Profiling results



Making sense of the results

`foldl` is creating a huge expression:

```
( ... (((0 + 1) + 2) + 3) + ...)
```

- We can only start evaluating at the end.
- Each subexpression is a new thunk:
  - We have one million thunks in memory!

Forcing additions

Adding two numbers is much faster than thunk (de)constructions!

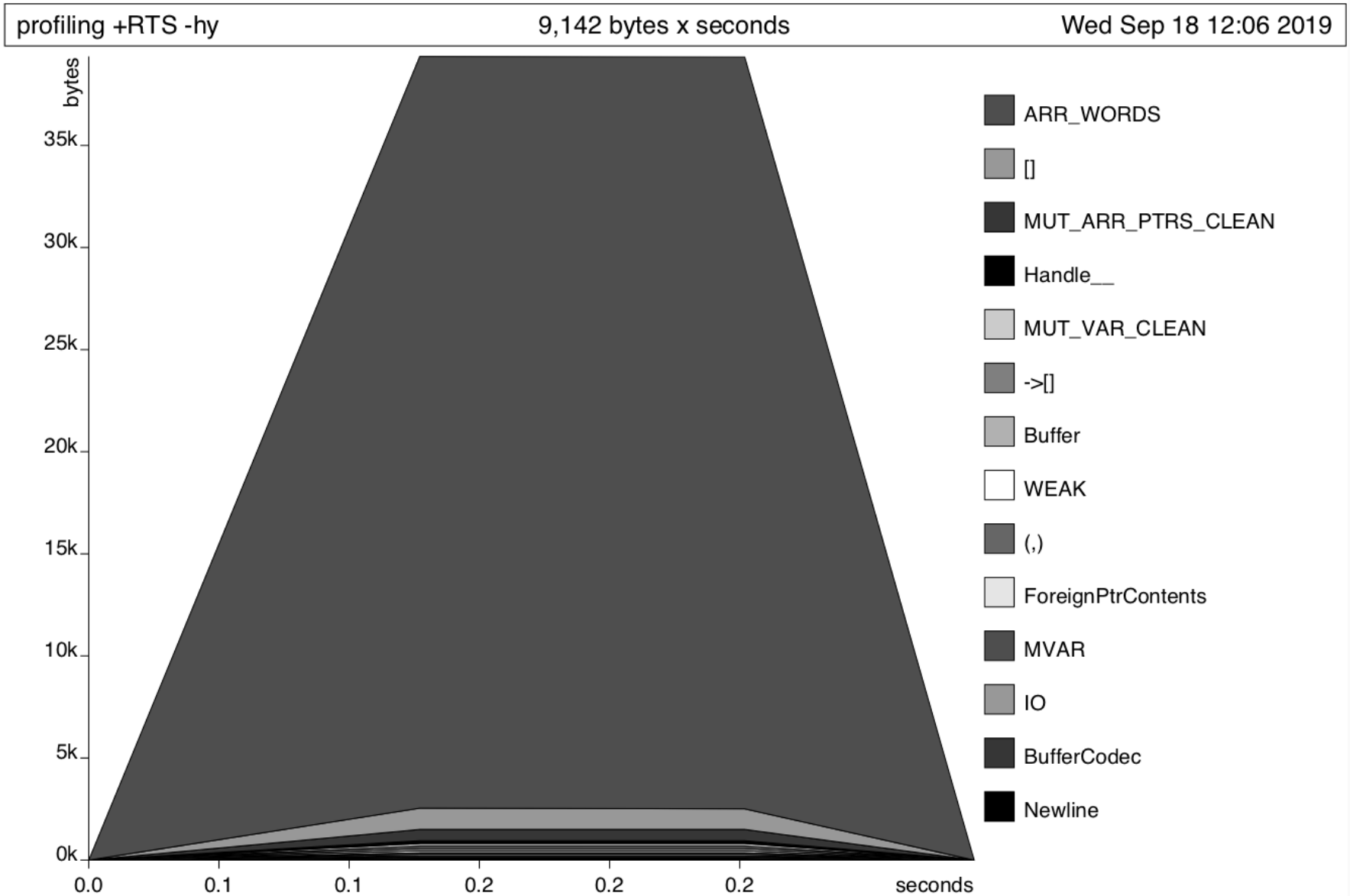
Forcing additions

Adding two numbers is much faster than thunk (de)constructions!

Let us kindly ask GHC to execute a thunk *now* instead of keeping it for *later* by using `seq`:

```
foldl' _ v []     = v
foldl' f v (x:xs) = let acc = f v x
                    in acc `seq` foldl' f acc xs
```

Profiling results



Bang patterns

```haskell
{-# language BangPatterns #-}
foldl' _ v []     = v
foldl' f v (x:xs) = let !acc = f v x
                    in foldl' f acc xs
```

You can also add them to fields to ensure that they are always forced at construction time:

```haskell
data Point = Point !Float !Float
```

Force versus deep force

`seq` only forces as much as a `case` statement:

- Figure out which is the top-most constructor,
  - Is it `Nothing` or `Just`? Is it `[]` or cons?
- But do *not* force any of the contents.

You can get *predictability* at the expense of *potentially useless* execution:

- You can force all the way down with `deepseq`.
- But note that you cannot force through functions.

Strict containers

`containers` is one of the most used and most useful packages in the Haskell ecosystem.

- It provides all sorts of data structures: Dictionaries, sets, graphs, sequences...
- Hash-based containers live in `unordered-containers`.

For each of these structures, you usually have a `Strict` module.

- They force the elements before saving them.

Summary for lazy people

- Haskell expressions are executed "when needed":
  - The evaluation is deferred in form of a *thunk*.
  - Inspecting a value (using `case`) *forces* a thunk to be evaluated "as much as needed"
- Good things we get:
  - Don't need to worry about doing too much work.
  - Manipulation of infinite structures.
- Terrible things that may happen:
  - *Space leak*: Thunks taking more time and space than the actual computation.
  - Use `seq` to manually force evaluation.
  - The profiler can help you find the right spot.

Dealing with the outside world

The “outside world”

Up to now, we have only talked about the *pure* part of Haskell.

- Functions whose only computational content is to obtain an output value based on the value of its arguments.

The “outside world”

Up to now, we have only talked about the *pure* part of Haskell.

- Functions whose only computational content is to obtain an output value based on the value of its arguments.

Most programs need to do other stuff:

- Interact with the user,
  - Hidden from you in the Gloss exercise;
- Communicate over a network,
- Persist information.

All these are *side effects*, and we call function with those *impure*.

Purity as a goal

Pure has almost a *moral* meaning.

- But here, impure is not strictly bad.

Purity as a goal

Pure has almost a *moral* meaning.

- But here, impure is not strictly bad.

The goal is to separate as much as possible the pure from the impure in your program.

- Later, even different kinds of impurity from each other.
- We want to be *precise* about purity.

Simple console input and output

```
main = do putStrLn "What is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name ++ "!")
```

Simple console input and output

```
main = do putStrLn "What is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name ++ "!")
```

- Input/output blocks are marked with `do`.
- Each line is a *statement*, with some effects, and optionally a return value.
  - We bind the value to a variable with `<-`.
- You can use pure functions freely.

Referential transparency

When does `e == e` return the same as `let x = e in x == x`?

Referential transparency

When does `e == e` return the same as `let x = e in x == x`?

```
take 2 [1,2,3] == take 2 [1,2,3]
let x = take 2 [1,2,3] in x == x

-- Asks the user *twice*
getLine == getLine
-- Asks the user *once*
let x = getLine in x == x
```

We say `getLine` is not *referentially transparent*.

- RT is one of the components of purity.
- Actually, the term *pure* is up for debate.

The IO type

How does Haskell ensure that `getLine` is not used wrong?

```
> getLine == getLine
<interactive>:1:1: error:
    • No instance for (Eq (IO String)) arising from ...
    • In the expression: getLine == getLine

> :t getLine
getLine :: IO String
```

The `IO` type

Functions with side effects are marked with `IO`:

```
putStrLn :: String -> IO ()
getLine  ::           IO String
```

The IO type

Functions with side effects are marked with `IO`:

```
putStrLn :: String -> IO ()
getLine  ::           IO String
```

```
> putStrLn getLine
<interactive>:3:10: error:
    • Couldn't match type ‘IO String’ with ‘String’
```

```
> do n <- getLine ; putStrLn n
Hello
Hello
```

Pure computations in `do`

If we want to have a name for a pure expression, we use `let` without `in`.

```
main = do putStrLn "What is your name?"
          name <- getLine
          let greeting = "Hello, " ++ name ++ "!"
          putStrLn greeting
```

Naïve `IO`

Using `IO` in this way brings two problems:

- Some functions may fail and raise an *exception*.
- Laziness means that order is not guaranteed.

Exceptions

```
readFile :: FilePath -> IO String
```

simply reads the entire contents of a file as a string.

```
> readFile "elquijote.md"
"En un lugar de la Mancha" ...
> readFile "ghostfile.md"
*** Exception: ghostfile.md: openFile: does not exist
    (No such file or directory)
```

Wait! What?

Wasn't Haskell all about being super-precise about the types?

Why not have the following signature?

```
readFile :: FilePath -> IO (Either IOError String)
```

Wait! What?

Wasn't Haskell all about being super-precise about the types?

Why not have the following signature?

```
readFile :: FilePath -> IO (Either IOError String)
```

We have to strike a **balance** between preciseness and ease of use.

`IO` = can do anything *and throw exceptions*

Catching exceptions

Everything related to exceptions lives in `Control.Exception`.

```
catch :: Exception e => IO a -> (e -> IO a) -> IO a
```

Catching exceptions

Everything related to exceptions lives in `Control.Exception`.

```
catch :: Exception e => IO a -> (e -> IO a) -> IO a
```

- `catch` is mostly used infix.
- We have to declare which exceptions we want to handle.

```
{-# language ScopedTypeVariables #-}

main = do txt <- readFile "ghostfile.md"
          putStrLn txt
       `catch` \(e :: IOException) ->
          putStrLn "file not found"
```

Different kinds of exceptions

```
print (1 `div` 0)
`catch` \(e :: ArithException) ->
  putStrLn "computing went wrong"

print (head [])
`catch` \(ErrorCall e) ->
  putStrLn "you should not do that"
```

Different kinds of exceptions

```
print (1 `div` 0)
`catch` \(e :: ArithException) ->
  putStrLn "computing went wrong"

print (head [])
`catch` \(ErrorCall e) ->
  putStrLn "you should not do that"
```

Exceptions may be raised from pure functions:

```
error :: String -> a
throw :: Exception e => e -> a
```

Laziness and IO

IO computations are executed lazily too.

```
-- Read a file as if we were C programmers
main = > do h <- openFile "existing.md" ReadMode
            c <- hGetContents h
            hClose h
            print (length c)
*** Exception: existing.md: hGetContents: illegal operation
    (delayed read on closed handle)
```

Laziness and IO

`IO` computations are executed lazily too.

```
-- Read a file as if we were C programmers
main = > do h <- openFile "existing.md" ReadMode
            c <- hGetContents h
            hClose h
            print (length c)
*** Exception: existing.md: hGetContents: illegal operation
    (delayed read on closed handle)
```

By the time we force `c`, the handle `h` has already been closed.

Too complicated?

The problem is that our input/output is *too low-level*.

Solution: Use the `bracket` pattern: Functions that take care of resources correctly (even in the case of exceptions).

```
withFile "elquijote.md" ReadMode $ \h ->
  do c <- getContents
     print (length c)
```

Too complicated?

The problem is that our input/output is *too low-level*.

Solution: Use the `bracket` pattern: Functions that take care of resources correctly (even in the case of exceptions).

```
withFile "elquijote.md" ReadMode $ \h ->
  do c <- getContents
     print (length c)
```

This solution has some problems:

1. It leads to deeply nested code,
2. Resources are deallocated in inverse order of allocation,
3. It only takes care of one resource.

Higher-level resource management

Use `managed` to manage your independent resources.

```
runManaged $ do
  h <- managed (withFile "elquijote.md" ReadMode)
  c <- liftIO getContents
  liftIO $ print (length c)
```

Higher-level resource management

Use `managed` to manage your independent resources.

```
runManaged $ do
  h <- managed (withFile "elquijote.md" ReadMode)
  c <- liftIO getContents
  liftIO $ print (length c)
```

Use `resource-pool` to manage pools of resources.

- For example, database connections.

IO Summary

- Haskell distinguishes pure from side effectful computations.
  - The latter is marked with `IO`.
  - `IO` comes with its own `do` notation.
- Functions may throw *exceptions*.
  - Use `Control.Exception` to handle them.
- Lazy `IO` can be very problematic.
  - Especially with (de)allocation of resources.
  - Use `managed` or `resource-pool` for safe higher-level manipulation.

Time to code!

Exercise 2: *A simple TCP server*

- Do some real `IO` stuff!
- Learn about "generics."

Remember: *Pair programming* helps you break the monotomy and helps you share thoughts!

Deep darkness: Variables in Haskell

A single-value TCP server

We want to develop a very stupid TCP server:

- `{ command: "write", value: "hello" }` updates the current value in our (memory-based) store to `"hello"` and sends back the previous one.
- `{ command: "read" }` sends the current value over the wire.

Concurrent connections

From the `network-simple` docs:

> *[...] A single TCP server can sequentially accept many incoming connections, possibly handling each one concurrently.*

What can go wrong?

Concurrent connections

From the `network-simple` docs:

> *[...] A single TCP server can sequentially accept many incoming connections, possibly handling each one concurrently.*

What can go wrong?

What can go wrong????? Concurrent updates, reading dirty values, maybe laziness... And how do we even get variables in a pure language???

Software Transactional Memory

- (Transactional) *variables* are represented by `TVar a`.
- *Transactions* are represented by `STM` computations.
  - Atomic and isolated ("all-or-nothing").
- Use `atomically` to run a transaction.

The server, part 1

```haskell
main :: IO ()
main = do
  v <- newTVarIO "hello"  --- !!1
  serve (Host "127.0.0.1") "8080" $ \(socket, _) -> do
        content <- recv socket 10000
        case content >>= decodeStrict of
           Just (Command "read" _) -> do
             currentValue <- readTVarIO v  -- !!2
             send socket (pack currentValue)
```

1. Create a new transactional variable.
   - They always have a value.
2. Read a transactional variable.
   - This operation happens in `IO`.

The server, part 2

```
case content >>= decodeStrict of
  ...
  Just (Command "write" (Just newValue)) -> do
    oldValue <- atomically $ do
      old <- readTVar v      -- T
      writeTVar v newValue   -- T
      return old             -- T
    send socket (pack oldValue)
```

Everything marked with `T` is a single *transaction*.

- `STM` is a stripped-down version of `IO`.
- `return` gives back something from the block.

```
atomically :: STM a -> IO a
```

Other STM structures

- `TMVar`s work like locks or semaphores with a value:
  - They might be empty or full.
  - `readTMVar` on an empty one is blocking.
  - `putTMVar` on a full one makes the transaction fail.
- `TChan` and `TQueue` provide *FIFO queues*.

STM transactions may fail

You may want to check some invariants before proceeding.

```
check :: Bool -> STM a
```

This has two effects:

- The current transaction is *rolled back*.
- The current thread is blocked until any of the `TVars` change.
  - Then the transaction is performed again.

Time to code!

Keep working on Exercise 2: *A simple TCP server*

- Now add some concurrent functionality.

Remember: *Pair programming* helps you break the monotomy and helps you share thoughts!

Lunch time!

Yin: Pure modelling

Polymorphic programs

The Orc strategy

```
orcStrategy :: Character -> World -> IO World
orcStrategy orc w = do
  let p = position orc w
      cs = [c | s <- surroundings p
              , Just (Combatant c) <- combatantInfo s w ]
  case cs of
    [] -> do d <- randomElement [N .. W]
             w' <- move orc d w
             redraw w'
             orcStrategy orc w'
    _  -> do c <- randomElement cs
             w' <- attack orc c w
             redraw w'
             orcStrategy orc w'
```

(So many) problems

- Logic is mixed with calls to `redraw`.
- Random execution is very difficult to test.
  - The calls to `randomElement` tie us to `IO`.
  - Which implies that we cannot use this in a pure environment.

First step: Extract dependencies

```
orcStrategy :: ([a] -> IO a) -> (World -> IO ())
            -> Character -> World -> IO World
orcStrategy rand redraw orc w = do
  let ...
  case cs of
    [] -> do d <- rand [N .. W]
             w' <- move orc d w
             redraw w'
             orcStrategy orc w'
    _  -> do c <- rand cs
             w' <- attack rand orc c w
             redraw w'
             orcStrategy orc w'
```

First step: Extract dependencies

```
orcStrategy :: ([a] -> IO a) -> (World -> IO ())
            -> Character -> World -> IO World
```

- Dependencies are explicit.
  - We can easily swap them for testing purposes.
- However...
  - They require a lot of manual passing (look at `attack`).
  - We are still tied to `IO`.

Reminder: Type classes

Type classes are Haskell version of *interfaces*, *traits*, or *protocols*.

- Define some *methods* to be supported by a type.

```haskell
class Eq a where
  equals :: a -> a -> Bool
```

- A type declares its support for a class by providing an *instance*.

```haskell
instance Eq a => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _      == _      = False
```

Second step: Introduce type classes

```haskell
class Gamble m where
  roll :: [a] -> m a
class DrawWorld m where
  redraw :: World -> m ()
```

Second step: Introduce type classes

```haskell
class Gamble m where
  roll :: [a] -> m a
class DrawWorld m where
  redraw :: World -> m ()

orcStrategy :: (Gamble m, DrawWorld m)
            => Character -> World -> m World
orcStrategy orc w = do
  let ...
  case cs of
    [] -> ...
    _  -> do c <- roll cs
             w' <- attack orc c w
             redraw w'
             orcStrategy orc w'
```

Going one step further

In `orcStrategy` we implicitly assume that `position` is pure.

- What about durable persistence?
- What about dealing with a shared concurrent state?

Using a `Character` to index the position is also an implementation detail, which we can abstract over.

Going one step further

In `orcStrategy` we implicitly assume that `position` is pure.

- What about durable persistence?
- What about dealing with a shared concurrent state?

Using a `Character` to index the position is also an implementation detail, which we can abstract over.

```
-- In this class there is an implicit 'me',
-- which is the combatant the script talks about.
class Combat m where
  myPosition :: m Position
  move       :: Direction -> m ()
  attack     :: Combatant -> m AttackResult
```

Abstracting over effects

Why do we try so hard to abstract over `IO`?

- `IO` means that any side effect may happen.
- We cannot swap an implementation for a mock.

Abstracting over effects

Why do we try so hard to abstract over `IO`?

- `IO` means that any side effect may happen.
- We cannot swap an implementation for a mock.

```haskell
instance Gamble TreeOfPossibilities where
  roll = -- save all possibilities, not only one

instance DrawWorld STM where
  redraw = -- do something concurrent
```

The code above does not compile!!!

```
No instance for (Monad m) arising from a do statement
```

We managed to not speak about *monads* for more than half a day!

The code above does not compile!!!

```
No instance for (Monad m) arising from a do statement
```

We managed to not speak about *monads* for more than half a day!

The key point here is *in a* `do` *statement*.

Simple intro to monads

To specify how computation with effects works, you need to tell:

1. The primitive effects that one may use.
   - `roll`, `redraw`, and so on.
2. How to **stitch** those effects together.

```
diceSum = do x <- roll [1 .. 6]
             y <- roll [1 .. 6]
             return (x + y)
```

   - In other words, how to compose statements.
   - This is the information `Monad` gives us.

Monad as computations

```
class Monad m where
```

- If you have a value, you can create a computation with no effects that just returns that value.

```
return, pure :: a -> m a
```

- If you have a computation, and a way to keep working, you can compose in a sequential manner.

```
(>>=) :: m a -> (a -> m b) -> m b  -- "bind"
```

  - Similar to applying a function to a value:

```
(&) :: a -> (a -> b) -> b
x & f = f x
```

`Maybe` as a monad

`Maybe` models optional values (called `Option` in other languages).

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
```

- If you have a value, you can create a computation with no effects that just returns that value.

  ```
  return x = Just x
  ```

- If you have a computation, and a way to keep working, you can compose in a sequential manner.

  ```
  Nothing >>= _ = Nothing
  Just x  >>= f = f x
  ```

Using `Maybe` as a monad

We used this ability in Exercise 2:

```
do ...
   content <- recv socket 10000
   case content >>= decodeStrict of
     ...
```

```
content       :: Maybe ByteString
decodeStrict :: ByteString -> Maybe Command
-----------------------------------------
content >>= decodeStrict  ::  Maybe Command
```

Fixing our code

- Introduce a `Monad` constraint on each function:

```
orcStrategy :: (Monad m, Gamble m, DrawWorld m) => ...
```

- Make `Monad` a superclass of all of them:

```
class Monad m => Gamble    m where ...
class Monad m => DrawWorld m where ...
class Monad m => Combat    m where ...
```

Monads, applicatives, alternatives

- Monads represent *sequential* execution.

```
(>>=) :: m a -> (a -> m b) -> m b
```

- Applicatives represent *parallel* execution.

```
(&&&) :: m a -> m b -> m (a, b)
```

  - Sequential can simulate parallel, hence any monad is also an applicative.

- Alternatives represent *failure* and *choice*.

```
empty :: m a
(<|>) :: m a -> m a -> m a
```

Two approaches at service abstraction

1. *Extract dependencies* (Wlaschin's approach):
   - Start with a highly-coupled implementation.
   - Turn functions that come from a different bounded countext into arguments.
   - Gather sets of functions into type classes.
2. *Design an algebra* (Ghosh's approach):
   - During the modelling phase, think about the services that your domain provides.
   - Refine the services into a small set of core primitives.
   - Turn this small set into a type class.

A word on terminology

People refer to the idea of having type classes for operations with different names:

- *Tagless Final* (pioneered by Oleg Kiselyov)
- *Polymorphic Programs*
- *Monad Classes* and `mtl`-*style* (mostly in Haskell-land)

There is also some terminology about the elements:

- *Algebra* refers to the type class, which defines the *operations*
- Each instance is an *interpreter* or a *handler*

Restrictive monads

`IO` is the ultimate monad: Anything may happen there.

- There is a type class for monads that support `IO` operations.

```haskell
class Monad m => MonadIO m where
  liftIO :: IO a -> m a
```

The trend is to create *subsets* of `IO` as type classes:

- You can still interpret them in `IO`.
- You gain safety because operations are restricted.

Time to code!

Exercise 3: *`GambleMonad`*

- Extract the dependencies of a piece of code into type classes.
- Implement interpreters for those type classes, both impure and pure (a tree of all possibilities).

The `Monad` instances are provided, but you can also learn a lot from inspecting them.

Exercise 4: *Three Monads*

Write `Functor`, `Applicative`, `Alternative`, and `Monad` for lists, optionals, and computations with environments.

Generalized ADTs

You must escape your strings

Why not track the difference between both in the data type?

```
data Stringo = NotYetEscaped String
             | Escaped       String

escape :: Stringo -> Stringo
escape (NotYetEscaped s) = Escaped (... s ...)
escape (Escaped       s) = Escaped s
```

You must escape your strings

Why not track the difference between both in the data type?

```
data Stringo = NotYetEscaped String
             | Escaped       String

escape :: Stringo -> Stringo
escape (NotYetEscaped s) = Escaped (... s ...)
escape (Escaped       s) = Escaped s
```

However, now all functions using `Stringo` need to consider both cases (and eventually call `escape`):

- We pollute other functions with escaping.
- **We cannot guarantee that escaping is always performed.**

Tracking information in the types

1. Introduce two (empty) data types to describe the possible states of the string:

```
data Escaped
data NotEscaped
```

Tracking information in the types

1. Introduce two (empty) data types to describe the possible states of the string:

```haskell
data Escaped
data NotEscaped
```

2. Make `Stringo` use this type as *index*:
   - An *index* is a type argument that specifies some property.

```haskell
-- Stringo is indexed by e(scaping)
newtype Stringo e = Stringo String
```

   - As opposed to *parameter*, which defines the type of some sub-component of the type.

```haskell
data Maybe a = Nothing | Just a
```

Tracking information in the types

3. Provide two different ways to create `Stringo`s:

```
fromString :: String -> Stringo NotEscaped
escape     :: Stringo e -> Stringo Escaped
```

Tracking information in the types

3. Provide two different ways to create `Stringo`s:

```
fromString :: String -> Stringo NotEscaped
escape     :: Stringo e -> Stringo Escaped
```

4. Now functions may require the programmer to escape the string by asking for the proper type:

```
buildHtmlTitle :: Stringo Escaped -> HtmlElement
```

Tracking authentication

Imagine that we want to define *connection*s, which may be in three different states:

- *Closed*: No communication is possible.
- *Open*: Only anonymous communication possible.
- *Auth*: We have gone through an additional step of authentication.

We cannot use the same trick as `Stringo`, because different states have **different data**.

Representing data types by their constructors

```
{-# language GADTSyntax #-}

data Character where
  Player  :: BasicStats -> Skills      -> Character
  Fighter :: BasicStats -> CombatStats -> Character
  NPC     :: BasicStats                -> Character
```

Representing data types by their constructors

```haskell
{-# language GADTSyntax #-}

data Character where
  Player  :: BasicStats -> Skills      -> Character
  Fighter :: BasicStats -> CombatStats -> Character
  NPC     :: BasicStats                -> Character

data Maybe a = Nothing | Just a
-- can be written as
data Maybe a where
  Nothing ::      Maybe a
  Just    :: a -> Maybe a
```

Generalized ADTs

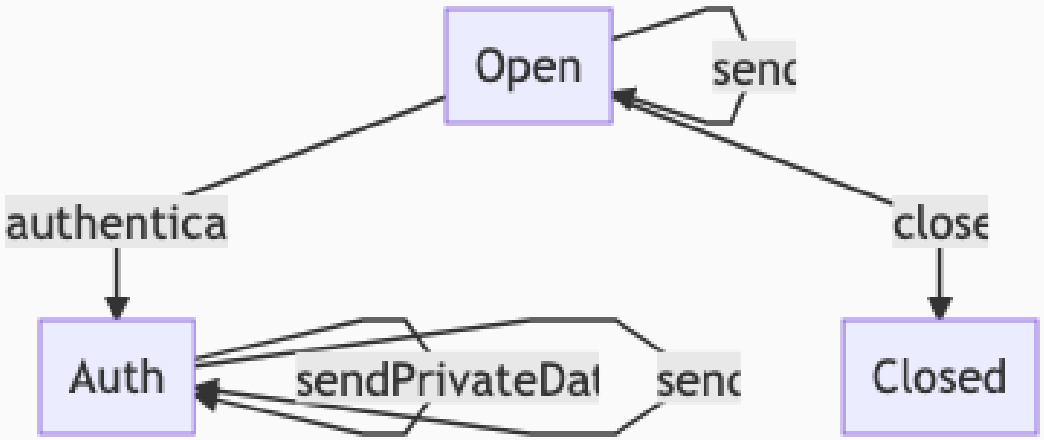
Normal ADTs do not allow us to use different indices for each constructor. **Generalized ADTs** (GADTs) do.

```
data Open
data Closed
data Auth

data Connection s where
  ConnClosed ::                          Connection Closed
  ConnOpen   :: Socket                -> Connection Open
  ConnAuth   :: Socket -> AuthData    -> Connection Auth
```

Connection state machine

```
connect         :: ConnData
                -> IO (Either ConnError (Connection Open))
authenticate    :: AuthData -> Connection Open
                -> IO (Either AuthError (Connection Auth))
sendPrivateData :: Connection Auth -> ByteString -> IO ()
close           :: Connection s -> Connection Closed
```



Sending from `Open` or `Auth`

This type signature is *wrong*: We cannot send if `s` is `Closed`.

```
send :: Connection s -> ByteString -> IO ()
```

Sending from `Open` or `Auth`

This type signature is *wrong*: We cannot send if `s` is `Closed`.

```
send :: Connection s -> ByteString -> IO ()
```

```
class CanSend s where connSocket :: Connection s -> Socket
instance CanSend Open where ...
instance CanSend Auth where ...
```

```
send :: CanSend s => Connection s -> ByteString -> IO ()
```

GADTs are great!

Refine your indices with the *state* of your data.

Then functions work as *transitions* in a state machine.

Yang: Impure concurrent execution

Simple concurrency

Going back to this morning

We have already dealt with concurrency issues in the TCP server.

- Each connection gets its own thread.
- Communication was done using Software Transactional Memory.

Going back to this morning

We have already dealt with concurrency issues in the TCP server.

- Each connection gets its own thread.
- Communication was done using Software Transactional Memory.

How do you create your own threads?

Parallelism for pure code

Haskell code is great for parallelism:

- The input is immutable, so you can reorder the computations as you want.
- Some higher-order functions lean themselves to be parallelize.
  - Think of a `map` or an associative `fold`.

Parallelism for pure code

Haskell code is great for parallelism:

- The input is immutable, so you can reorder the computations as you want.
- Some higher-order functions lean themselves to be parallelize.
  - Think of a `map` or an associative `fold`.

Two main strategies in the Haskell ecosystem:

- `parallel` uses *strategies* to define how to parallelize.
  - "Divide this list in chunks of length 5, and do each one in parallel".
- `monad-par` uses a *promise*-based interface.

Spawn-based parallelism

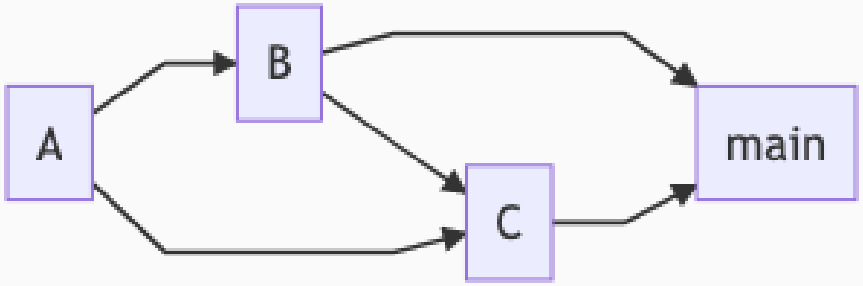
Similar to STM, we have a specialized `Par` monad.

```
-- ask to run computations in parallel
spawn  :: NFData a => Par a -> Par (IVar a)
spawnP :: NFData a =>      a -> Par (IVar a)

-- read the result of a computation
get :: IVar a -> Par a

-- actually execute the computation
runPar :: Par a -> a
```

Data flow parallelism




```
promiseA <- spawn codeA
promiseB <- spawn $ do valA <- read promiseA
                       codeB
promiseC <- spawn $ do valA <- read promiseA
                       valB <- read promiseB
                       codeC
valB <- read promiseB
valC <- read promiseC
mainComputation valB valC
```

Parallelism for `IO`

We usually want to parallelize input/output operations:

- Download two files in parallel.
- Send data to a database and an event store.

Parallelism for `IO`

We usually want to parallelize input/output operations:

- Download two files in parallel.
- Send data to a database and an event store.

Again, we have two library choices:

- `monad-par` provides `ParIO` in addition to `Par`.
  - No need to learn another set of functions.

```
runParIO :: ParIO a -> IO a
```

- `async` also provides a promise-based interface.
  - More control on execution: computations can be cancelled.
  - Exceptions are propagated correctly.

Example of `Async`

`simpleHTTP` comes from the `HTTP` library.

```
withAsync (simpleHTTP (getRequest "http://foo")) $ \req1 ->
withAsync (simpleHTTP (getRequest "http://bar")) $ \req2 ->
do rsp1 <- wait req1
   rsp2 <- wait req2
   case (rsp1, rsp2) of
     (Right res1, Right res2) -> ...
     _ -> putStrLn "error downloading"
```

Useful combinators

Spawning threads and waiting for all of them is quite common.

```
responses <- concurrently (simpleHTTP (getRequest ...))
                          (simpleHTTP (getRequest ...))
case responses of ...
```

Useful combinators

Spawning threads and waiting for all of them is quite common.

```
responses <- concurrently (simpleHTTP (getRequest ...))
                          (simpleHTTP (getRequest ...))
case responses of ...
```

There are more combinators in `async`:

```
   concurrently :: IO a -> IO b -> IO (a, b)
           race :: IO a -> IO b -> IO (Either a b)
mapConcurrently :: Traversable t
                => (a -> IO b) -> t a -> IO (t b)
```

Green threads

By default, Haskell threads are not real OS threads.

- We call them *green threads*.
- Main difference: Threads are managed by the run-time.
- Result: They are extremely cheap to create.

Most I/O operations use non-blocking primitives under the hood.

- The I/O manager takes care of pausing and resuming threads when the work they required has been done.
- There is a lot of work in a parallel I/O manager.

Haskell's `IO` versus ZIO / Cats Effects / ...

Haskell's ecosystem has all the building blocks:

- `IO` marks impure computations,
- `Async` is similar to a fiber,
- `STM` provides variables and queues,
- `Managed` handles resources gracefully,
- `Conduit` gives you streams.

Haskell's `IO` versus ZIO / Cats Effects / ...

Haskell's ecosystem has all the building blocks:

- `IO` marks impure computations,
- `Async` is similar to a fiber,
- `STM` provides variables and queues,
- `Managed` handles resources gracefully,
- `Conduit` gives you streams.

Missing:

- reified information over exceptions,
- fine-grained control over execution.

Time to code!

Exercise 5: *Tossing coins in different ways*

- Implement a simple function that produces n random Booleans using different styles of parallel and concurrent programming in Haskell.
- Measure the performance of each of them.

Streaming libraries

Manual input/output can be hard

We have already looked at several problems:

- Laziness means consuming at unexpected moments,
  - This may incur in huge memory consumption,
- Resource (de)allocation is hard,
- All of this, combined with correct exception handling.

Streaming libraries

`conduit`, `pipes`, and `streaming` try to solve these problems:

- Keep a constant memory usage,
  - No need to keep all data in memory,
- Handle resources correctly,
  - Close a file when you have arrived to the EOF,
- Common interface for data consumption and generation.

Streaming libraries

`conduit`, `pipes`, and `streaming` try to solve these problems:

- Keep a constant memory usage,
  - No need to keep all data in memory,
- Handle resources correctly,
  - Close a file when you have arrived to the EOF,
- Common interface for data consumption and generation.

`conduit` seems to have the most "adapters" in the ecosystem.

- Be aware of pre-1.3 documentation!

Core concepts

```
data ConduitT input output effects result
```

- Consumes a stream of values of type `input`,
- Produces a stream of values of type `output`,
- May produce effects from monad `effect`,
- At the end, produces a single value of type `result`.

Core concepts

```haskell
data ConduitT input output effects result
```

- Consumes a stream of values of type `input`,
- Produces a stream of values of type `output`,
- May produce effects from monad `effect`,
- At the end, produces a single value of type `result`.

The connect/fuse operator joins two of them:

```haskell
(.|) :: Monad m => ConduitM a b m ()
                -> ConduitM b c m r
                -> ConduitM a c m r
```

`ConduitT` is very general

Producers (or sources):

```
sourceFile :: FilePath -> ConduitT i ByteString m ()
```

Consumers (or sinks):

```
-- without interesting result
sinkFile :: FilePath -> ConduitT ByteString o m ()
-- with interesting result
sumC :: Num n => ConduitT n o m n
```

Transformers (or conduits):

```
mapC :: (a -> b) -> ConduitT a b m ()
```

Running a `ConduitT`

You need to "start the engine" by means of:

```
runConduit     :: Monad m
               => ConduitT () Void m r -> m r
runConduitRes  :: MonadUnliftIO m
               => ConduitT () Void (ResourceT m) r -> m r
runConduitPure :: ConduitT () Void Identity r -> r
```

Running a `ConduitT`

You need to "start the engine" by means of:

```
runConduit     :: Monad m
               => ConduitT () Void m r -> m r
runConduitRes  :: MonadUnliftIO m
               => ConduitT () Void (ResourceT m) r -> m r
runConduitPure :: ConduitT () Void Identity r -> r
```

- `ResourceT` takes care of resource management.
- `Identity` is the do-nothing monad.
- `Void` is a data type with *no* values.
  - That means that, by the end of your pipeline, all data must have been consumed.

Shout the contents of a file

In other words, print the contents of a file, but all in capitals.

```
sourceFile :: FilePath -> ConduitT i ByteString m ()
decodeUtf8LenientC ::     ConduitT ByteString Text m ()
mapC toUpper ::           ConduitT Text Text m ()
encodeUtf8C ::            ConduitT Text ByteString m ()
stdoutC ::                ConduitT ByteString o m ()
```

Shout the contents of a file

In other words, print the contents of a file, but all in capitals.

```
sourceFile :: FilePath -> ConduitT i ByteString m ()
decodeUtf8LenientC ::     ConduitT ByteString Text m ()
mapC toUpper ::           ConduitT Text Text m ()
encodeUtf8C ::            ConduitT Text ByteString m ()
stdoutC ::                ConduitT ByteString o m ()

shoutFile fp = runConduitRes $
      sourceFile fp .| decodeUtf8LenientC
   .| mapC Data.Text.toUpper
   .| encodeUtf8C .| stdoutC
```

Where to look for conduits?

- `conduit` and `conduit-extra`
  - basic list-like operators (map, fold)
  - filesystem, network, binary serialization...
- `http-conduit`: Simple HTTP requests
- `stm-conduit`: Wraps STM channels
- `ndjson-conduit`: Wraps Aeson for JSON serialization
- `amqp-conduit`: Connects to RabbitMQ
- `hw-kafka-conduit`: Connects to Kafa
- and many more in Stackage and Hackage!

Writing our own conduit

Most of the time you use combinators, but the interface is not very complicated:

- `ConduitT i o m` is a monad.
  - `return` gives the result and marks the stream as finished.
- `await` obtains the next input value,
  - It returns `Nothing` if the stream is finished.
- `yield` and `yieldM` generate an output value.

Writing our own conduit

Most of the time you use combinators, but the interface is not very complicated:

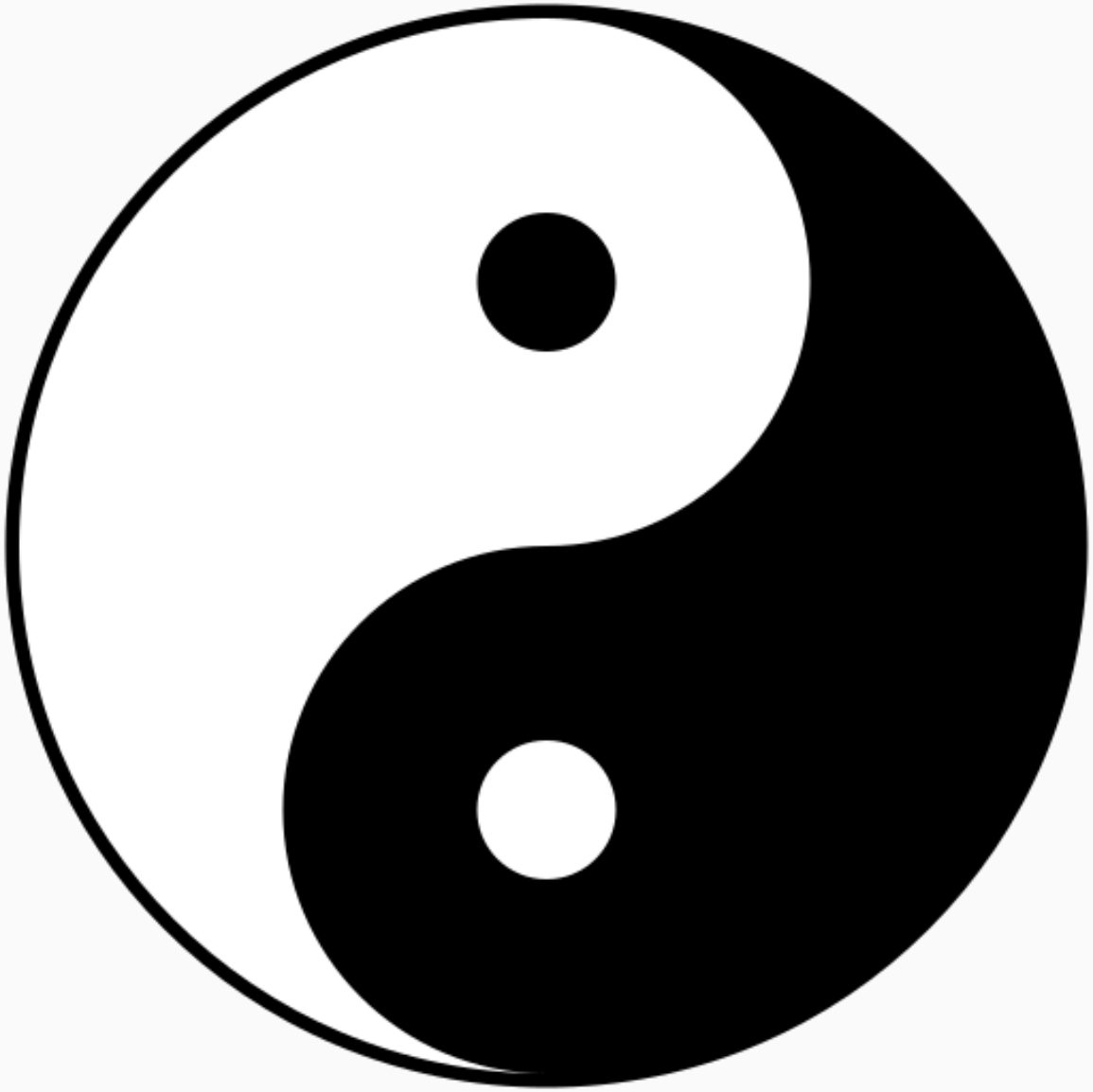
- `ConduitT i o m` is a monad.
  - `return` gives the result and marks the stream as finished.
- `await` obtains the next input value,
  - It returns `Nothing` if the stream is finished.
- `yield` and `yieldM` generate an output value.

```
duplicate :: Monad m => ConduitT v v m ()
duplicate = do v <- await
               case v of
                 Nothing -> return ()
                 Just w  -> yield w >> yield w >> duplicate
```

Time to code!

Exercise 6: *A chat server using streams*

- Implement a chat server using streams to communicate between clients and to wrap the connections.
- If time allows, implement private messages.



Common architectural patterns

The main question

What should be the types of my bussiness domain?

The main question

What should be the types of my bussiness domain?

The “frontiers” of an application must have some sort of `IO` type.

Answer 1: Hexagonal / onion architecture

Functions use the *polymorphic programs* technique to declare their *dependencies*.

```
orcStrategy :: (Monad m, Gamble m, DrawWorld m) => ...
```

- Each of the type classes works as a *port*.
- Each instance defines an *adapter*.
- The compiler *injects* the dependencies based on the types.

```
instance DrawWorld IO   where ...
instance DrawWorld Html where ...
```

Answer 2: Functional core, imperative shell

Functions are completely pure.

```
orcStrategy :: CombatData -> Character -> NextMove
```

- Pure functions are then composed in larger pipelines that consume the data in the system.
- A somehow dumb shell that moves data around the systems.

This is the same architetcure as in Gloss (or Elm, or even React).

Answer 2': Streaming FC/IS

Defining an application using streams is very close to the previous architecture:

- Pure stream transformers form the functional core.
- Around them you plug side-effectful sources and sinks, which make up the imperative shell.

What does a function represent?

- In FC/IS, functions are pure data manipulation, and type reflect the shape of the data they work with.
- In H/O architecture, types reflect the services from other contexts needed to run the application.

What does a function represent?

- In FC/IS, functions are pure data manipulation, and type reflect the shape of the data they work with.
- In H/O architecture, types reflect the services from other contexts needed to run the application.

Unfortunately, there is no answer to which point of view is best.

Summary

The Yin: Pure modelling

- Modelling data:
  - Make *sum types* part of your designs.
  - Distinguish between different concepts with *different types*.
  - Provide different *views* for the same data.
- Modelling services:
  - Polymorphic programs declare their *dependencies as type classes* (or algebras).
  - Using different instances, we can provide different implementations, including mocks.
  - *Monads* are the basic structure for sequential computation.

The Yang: Impure execution

- Laziness means the run-time evaluates "only when needed" and "as much as needed."
  - Space leaks: We keep too many thunks in memory.
  - Can be solved by selective forcing.
- Input/output is tricky.
  - Exceptions may arise
  - Laziness conflicts with resource management.
  - Streaming offers a good solution.
- Concurrency is simple.
  - GHC uses (lightweight) green threads by default.
  - `monad-par` and `async` provide promise-based interfaces.
  - STM adds transactional behavior to shared variables.

The Duality

Two main architectural patterns:

1. Hexagonal / onion archiecture.
    - Use polymorphic program technique in your domain.
2. Functional core, reactive shell
    - Keep the domain as pure data-manipulation functions.

- Pattern matching and view patterns.
- `do` notation for monads.
- Generics to derive serialization.
- Profiling and benchmarking.
- *Missing*: Applicative notation.

Where to go next?

- Look at property-based testing (QuickCheck and friends).
- Understand parsing (for example, `aeson`):
  - Learn about `(<$>)`, `(<*>)`, and `(<|>)`.
- More type-level programming:
  - Type families, data kinds, singletons.
  - Servant defines REST APIs at type level.
  - Persistent and Esqueleto define type-safe database queries.
- Reify programs as data:
  - Free and operational monads.

Where to go next?

- Look at property-based testing (QuickCheck and friends).
- Understand parsing (for example, `aeson`):
  - Learn about `(<$>)`, `(<*>)`, and `(<|>)`.
- More type-level programming:
  - Type families, data kinds, singletons.
  - Servant defines REST APIs at type level.
  - Persistent and Esqueleto define type-safe database queries.
- Reify programs as data:
  - Free and operational monads.

As usual, keep practicing, reading, and learning.

- Mailing list, blogs, Discourse, IRC.

One final warning

Haskell is also widely used as a vehicle for research.

- That you cannot read a research paper about type families, and understand the whole formalization, does not mean that you cannot use them.
- Be wise with the resources that you choose.

Enjoy Lambda World!

(I'll be around. Feel free to ask anything during these days)