

Practical Haskell

October 20th 2025

System Programming Ghent
Ghent Functional Programming Group



Practicalities

- Public Wifi
 - AP: DV-Public
 - Password TODO
- Schedule
 - 19:00 Grab some food & drinks
 - 19:15 Presentation + breaks to experiment
 - 21: 00 Wrap-up + discuss
- Example Code
 - <https://github.com/kdkeyser/practical-haskell>
 - Branches step1, step2, step3, webservice

Who am I?

Koen De Keyser

- Programming (hobby + professional) in lots of different languages
 - Pascal, assembly, C++, C#, Python, Java, OCaml, Haskell, Go, Kotlin, Rust, Rescript
- 7 years full time functional programming in a production context
 - Haskell and OCaml @ Amplidata / Western Digital
 - Large scale storage systems
- Currently VP Engineering @ DoubleVerify, Publisher Division

DoubleVerify Ghent

- R&D site for DoubleVerify, ~ 25 people in Ghent
- DV Publisher Suite (SaaS BI product) & Ad Measurement technology
- Large scale!
 - 100B events every day, 10B for just Publisher, O(PB) data sizes
 -



Let's do some Haskell

Haskell, what tools do I need?

- Haskell tooling has historically been challenging
- Much better now, still challenging when using bleeding edge version of the compiler and tools
- RustUp → GHCUp
 - <https://www.haskell.org/ghcup/>
- 3 major tools
 - Compiler: **GHC**
 - Package manager & build system: **Cabal**
 - Language Server: **HLS**
- Package repository
 - <https://hackage.haskell.org>
- Search-by-type
 - <https://hoogle.haskell.org/>

Haskell, what tools do I need?

1. Install GHCUp

- <https://www.haskell.org/ghcup/>

2. Install GHC, Cabal and HLS

- `ghcup install ghc 9.12.2`
- `ghcup install cabal 3.14.2.0`
- `ghcup install hls 2.11.0.0`
- `ghcup set ghc 9.12.2`
- `ghcup set cabal 3.14.2`
- `cabal update`

3. Install tooling for your IDE

- VSCode: Haskell Extension, <https://marketplace.visualstudio.com/items?itemName=haskell.haskell>
- IntelliJ: Haskell Plugin, <https://plugins.jetbrains.com/plugin/24123-haskell-lsp>

Hello World of Haskell

- Create a new project
 - cabal init
- We will create a project with
 - The project definition
 - hello-world.cabal
 - A number of modules
 - These live in /src/...
 - A main entry point
 - This lives in /app/Main.hs
 - A test suite
 - This lives in /test/Main.hs
- Building, testing, running
 - cabal build
 - cabal test
 - cabal run


```
$ mkdir hello-world
$ cd hello-world
$ cabal init
What does the package build:
  1) Library
  * 2) Executable
  3) Library and Executable
  4) Test suite
Your choice? [default: Executable] 3
...
$ cabal run
Resolving dependencies...
Build profile: -w ghc-9.12.2 -O1
In order, the following will be built (use -v for more details):
 - hello-world-0.1.0.0 (lib) (first run)
 - hello-world-0.1.0.0 (exe:hello-world) (first run)
Configuring library for hello-world-0.1.0.0...
Preprocessing library for hello-world-0.1.0.0...
Building library for hello-world-0.1.0.0...
[1 of 1] Compiling MyLib          ( src/MyLib.hs,
dist-newstyle/build/x86_64-linux/ghc-9.12.2/hello-world-0.1.0.0/build/MyLib.o,
dist-newstyle/build/x86_64-linux/ghc-9.12.2/hello-world-0.1.0.0/build/MyLib.dyn_o )
Configuring executable 'hello-world' for hello-world-0.1.0.0...
Preprocessing executable 'hello-world' for hello-world-0.1.0.0...
Building executable 'hello-world' for hello-world-0.1.0.0...
[1 of 1] Compiling Main          ( app/Main.hs,
dist-newstyle/build/x86_64-linux/ghc-9.12.2/hello-world-0.1.0.0/x/hello-world/build/hello-world/hello-world-tmp/Main.o )
[2 of 2] Linking
dist-newstyle/build/x86_64-linux/ghc-9.12.2/hello-world-0.1.0.0/x/hello-world/build/hello-world/hello-world
Hello, Haskell!
someFunc
```

Haskell, the language

Haskell, the language

Haskell is a

- *pure*
- *lazy*
- *functional*

programming language with

- *strong type checking*
- *type inference*

Haskell, a *functional* language

- The core concepts are the *function* and *function composition*
 - $\text{add } x \ y = x + y$
 - $\text{double } x = 2 * x$
 - $\text{double} . \text{add} = 2 * (x + y)$
- Functions are “first class primitives”
 - Functions can take other functions as arguments (higher order functions)
 - $\text{apply } f \ x \ y = f \ x \ y \Rightarrow \text{apply sum } 7 \ 5 = 7 + 5 = 12$
 - Functions can be partially applied
 - $\text{incrByFive} = \text{add } 5$

Haskell, a *lazy* language

- Calculations are delayed until the moment the result is needed
 - `let result = a + b`
 - when printing the result → calculation happens
- Thunks
 - Values that still need to be evaluated.
- Enables certain algorithms
 - Infinite lists (e.g. fibonacci), partial evaluation, ...
- Space leaks
 - Thunk size can grow very quickly in loops / folds. Keep an eye on the memory consumption.

Haskell, a *lazy* language

- Performance
 - Reasoning about Haskell performance can be difficult due to non-locality
 - A “simple” operation might take a lot of time because it forces evaluation of a large thunk
- Bang patterns
 - Force evaluation of a lazy expression using !
 - `let !result = 5 + 5`
 - Can be strict-by-default on a per module basis (`Strict` and `StrictData` language extensions)
- Tools
 - Flamegraphs (using GHC profiling output) → <https://github.com/fpco/ghc-prof-flamegraph>
 - Profiteur → <https://github.com/jaspervdj/profiteur>

Haskell, a *pure* language

- Functions are **deterministic** calculations: the output is completely defined by the input
- Functions do **not** have **side effects**
 - Cannot update or reference external variables (state)
- **Referential transparency**
 - Any function (application) can be replaced by the value that function would evaluate to
 - Memoization
- What about IO?
 - Pure function cannot have state, do IO, etc. (e.g. print)
 - Monadic IO: $f \text{ (world) } \rightarrow \text{ updated world }$
 - later more about monads and IO

Haskell, a *strongly-typed* language

- Every value has a well defined type at compile time
 - `5 :: Int`
 - `"Hello" :: String`
 - `add :: Int → Int → Int`
 - `apply :: (a → b → c) → a → b → c`
- Powerful type level features
 - Sum and product types, GADT, higher kinded types, polymorphism, type classes, type families, ...
- "If it compiles, it works"
- Type Inference
 - The compiler can (in many cases) derive the types from the expressions
 - Best practice: explicitly define types at the function level

Haskell, a *strongly-typed* language

- Haskell programmer mindset: leverage the type system as much as possible
 - Type driven development (→ Hoogle)
 - Data driven architecture
 - OOP encapsulation → Type construction
- “Parse, don’t validate”
 - Transform your input into a well-defined data type with invariants
 - Type and invariants are now guaranteed across your code
 - <https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/>
- Still doesn’t protect your from business logic misinterpretations / ill-defined specifications

Haskell, history and implications

- Haskell is originally a research language
 - Main contributors have been academia / research centers
 - Haskell can feel convoluted for non PLT researchers
 - Category Theory terminology can feel alien
 - “The study of mathematical structures and their relationships”
 - “*a monad is a monoid in the category of endofunctors*”
 - Inspiration for other (more commonly used) languages (e.g. Rust)
 - IO Monad introduced in 1995 (Haskell 1.3) → async IO
- Niche language in industry
 - https://wiki.haskell.org/Haskell_in_industry
 - Facebook (Simon Marlow)
 - Sigma, spam-fighting infrastructure, 1M+ req/s
 - Haxl, a DSL for optimizing access to distributed database backends

Haskell, why should you care?

- “Open your mind”
 - Think about what “computation” really means
 - In particular for parallel and concurrent programming
 - E.g. Software Transactional Memory, Execution “Strategies”
- Haskell typically enforces general best practices
 - Immutability, no global state, composability, ...
- Peek into the future of programming languages
 - You get a head start on ideas that might pop up in your favorite industry language
- Haskell in production? Your mileage may vary

Practical Haskell: a raytracer

Raytracer, step 1

- **Create a bitmap where the color depends on the pixel coordinates**
- Domain Concepts
 - Color & color spaces
 - Write a bitmap to a file
- Haskell Concepts
 - Sum & Product Type
 - Functions
 - Modules
 - Monadic IO and “do notation”

Raytracer, step 1 - Types

- Concrete Type

- Represents a concrete piece of data, i.e. no further parameters can be provided to the type
- Starts with capital letter
- Built-in
 - `() , Integer , Bool , Float , Char , ...`
- User defined (e.g. product type)
 - `data Color = Color Float Float Float`

- Polymorphic Type

- ~ Generics (Go, Java, ...), ~ Templates (C++)
- Contains one or more “placeholders” types (lowercase letter)
- Built-in
 - `(a , b) , [a] , Maybe a`
- User defined
 - `data Triple a b c = Triple a b c`

Raytracer, step 1 - Functions

- Functions contain a type definition and an implementation

```
srgb :: Float -> Float
srgb x =
  if x <= 0.0031308 then 12.92 * x else 1.055*(x ** (1.0 / 2.4)) - 0.05
```

- Functions can take other functions as arguments

```
generateImage :: (Int -> Int -> Color) -> Int -> Int -> Image Color
generateImage render width height =
  ...
```

Raytracer, step 1 - Modules

- Haskell structures code in “Modules”
 - Single file per module, e.g. `Color.hs`

```
{-# LANGUAGE ScopedTypeVariables #-}
```

```
module Color(
```

```
    Color (..),
```

```
    tosRGB
```

```
) where
```

```
data Color = Color Float Float Float
```

```
srgb :: Float -> Float
```

```
srgb x =
```

```
    if x <= 0.0031308 then 12.92 * x else 1.055*(x ** (1.0 / 2.4)) - 0.05
```

Language Extensions

Module name & exports

Type definitions

Functions

Raytracer, step 1 - Sum Types

- Sum Types
 - combines type A and type B
 - can represent all values that are either of type A, or of type B
- Enums
 - `data Color = Blue | Green | Red`
- Tagged Unions / Variant / ...
 - `data Color = Blue | Green | Red | Pantone String | Ral Int`
 - `data Maybe a = Nothing | Just a`

Raytracer, step 1 - Pattern Matching

- Case expression

```
f maybe =  
  case maybe of  
    Nothing -> "No value"  
    Just x   -> "Value: " ++ show x
```

- Pattern Match

```
f Nothing = "No value"  
f (Just x) = "Value: " ++ show x
```

- Careful with partial functions!
 - GHC will tell you: **“Pattern match(es) are non-exhaustive”**

Raytracer, step 1 - Product Types

- A product type
 - combines type A and type B
 - can represent all pairs consisting of 1 value of type A, and one of type B
 - `data User = User Int String -- (user ID / user name)`
 - `data Tuple a b = Tuple a b`
- Extendable to an arbitrary number of types
 - `data Quad a b c d = Quad a b c d`
- Often used with record syntax
 - More about this later

Raytracer, step 1 - Monadic IO



```
saveBmpImage  :: FilePath -> DynamicImage -> IO ()  
saveBmpImage filePath image = ...  
  
main :: IO ()  
main =  
    saveBmpImage "test.bmp" image
```

Raytracer, step 1 - Monadic IO



- Monad is a concept from category theory
 - But you can ignore that completely
- If we only have pure functions, how can we do IO?
 - Apply your “IO” to the world
 - `f :: World a → World b`
 - Need to be able to apply & sequence IO
 - `bind :: World a → (a → World b)`
- Monad definition
 - `bind :: m a → (a → m b)`
 - `>>= :: m a → (a → m b)`

Raytracer, step 1 - Monadic IO



- IO monad $\rightarrow m = \text{IO}$
 - `main () :: IO ()`
 - `>>= :: IO a \rightarrow (a \rightarrow IO b)`
- “do” notation

```
main :: IO ()
main = do
  r1 <- f1 ()
  r2 <- f2 r1
  f3 r2

main :: IO ()
main =
  f1 () >>= \r1 -> f2 r1 >>= \r2 -> f3 r2
```

Raytracer, step 1 - Monadic IO



- Monad is the most “powerful” way to chain side effects
 - Every side effect can inspect the result of the previous step, because `bind :: m a → (a → m b)`
 - includes all previous side effects
 - Need to execute sequentially (so sad!)
- Other options are possible!
 - Applicative functor
 - Arrows
 - Effect systems
- Inspecting side effects before execution
 - Haxl

Raytracer, step 1- “just do it”

- Create a bitmap where the color depends on the pixel coordinates
- Color & color spaces
 - Introduce a Color type that represents colors as 3 floats (0.0 ... 1.0) for red/green/blue
 - Transform the r,g,b colors to the sRGB color space
 - Corrects for non-linearity of output devices (i.c. monitor)
 - See previous slide for formula
- Write a bitmap to a file
 - JuicyPixels library (search on <https://hackage.haskell.org/>)
 - Use `saveBmpImage`, `generateImage` and `ImageRGBF`
- Use <https://hoogle.haskell.org/> to lookup helper / library functions
 - Search by type, e.g. `Int` → `Float`
- Type driven development: **`undefined`** is your best friend
 - Placeholder for any value / function



break

Raytracer, step 2

- Render a sphere in a uniform color in front of a camera
- Domain Concepts
 - Camera
 - Ray
 - Sphere
 - Vector
- Haskell Concepts
 - Product Type (named records)
 - Type Classes
 - Deriving

Raytracer, step 2: Product Type (2)

- A “Product Type” is the cartesian product of 2 or more other types
 - `data Color = Color Float Float Float`
- Easier to use as “records”
 - Names rather than positions
 - `data Vector = Vector { x :: Float, y :: Float, z :: Float }`
 - `v = Vector { x=1.0, y=0.5, z=0.7 }`
 - Can still be constructed positionally e.g. `v = Vector 1.0 0.5 0.7`
 - Fields names become getters, i.e. `xValue = x v`
- Records are a bit clunky in Haskell
 - Lenses, optics, ...

Raytracer, step 2: Type Classes

- If you know Rust \rightarrow Type Classes \sim Traits
- Definition
 - **a type class defines a set of functions a type needs to implement**

```
class Monad m where
  (>>=)  :: m a -> ( a -> m b) -> m b
  (>>)   :: m a -> m b           -> m b
  return :: a                   -> m a
```

- Built-in type classes
 - Show, Monad, Functor, ...
- You can specify and/or implement type classes yourself (you don't need to “own” the type)

Raytracer, step 2: Type Classes

- Type classes are typically constraints on polymorphic type parameters in functions

```
f :: (Show a) => a -> Bool -> String
```

Raytracer, step 2: Deriving

- Generate (trivial) typeclass implementation at compile time
 - Show

```
class Show a where  
  show :: a -> String  
  ...
```



```
data Vector = Vector {  
  x :: Float,  
  y :: Float,  
  z :: Float  
} deriving Show
```

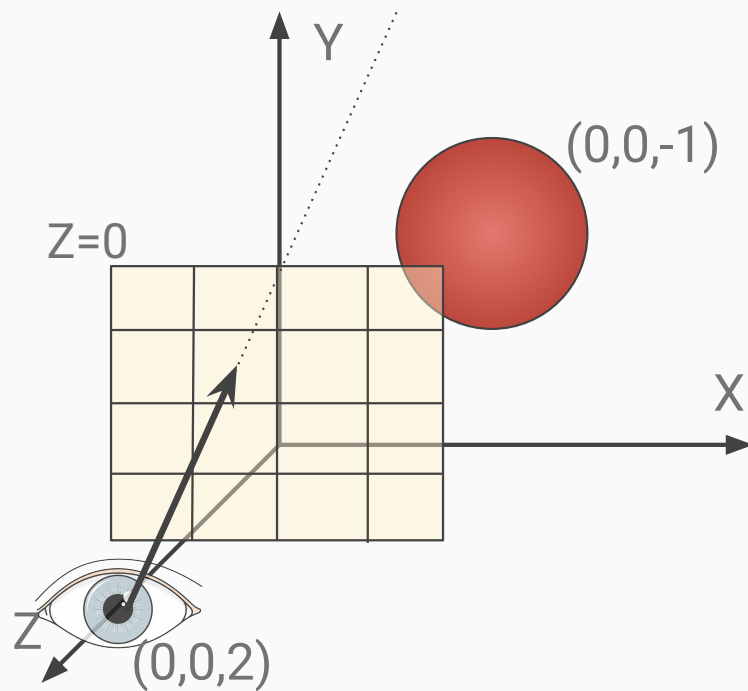
or

```
data Vector = Vector {  
  x :: Float,  
  y :: Float,  
  z :: Float  
}  
deriving instance Show (Vector)
```

- Originally restricted to specific built-in typeclasses (Eq, Ord, Enum, Ix, Bounded, Read, Show)
- Now a lot can be derived (e.g. ToJson, FromJson, ...)
 - See GHC.Generics

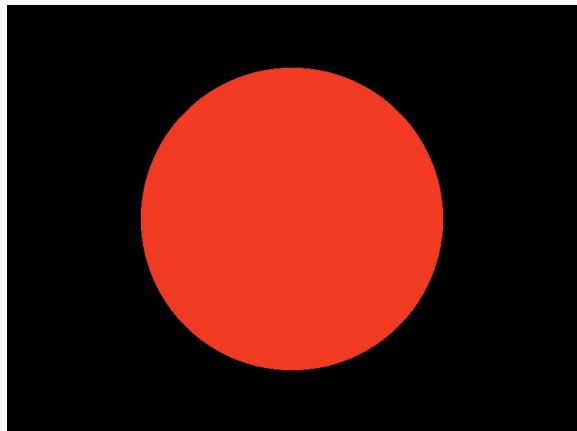
Raytracer, step 2 - “just do it”

- Vector
 - 3D vector, with x,y,z components
- Ray
 - Starts at a specific point in 3D space, and follows a specific direction
- Sphere
 - Center (3D point) and radius.
- Camera Screen
 - $Z=0$
- Problem to solve
 - For each pixel on the screen, follow a ray from the eye through the pixel, and check if it hits the sphere. If it does, the pixel is red, otherwise black.



Raytracer, step 2 - “just do it”

- Vector
 - 3D vector, with x,y,z components
- Ray
 - Starts at a specific point in 3D space, and follows a specific direction
- Sphere
 - Center (3D point) and radius.
- Camera Screen
 - $Z=0$
- Problem to solve
 - For each pixel on the screen, follow a ray from the eye through the pixel, and check if it hits the sphere. If it does, the pixel is red, otherwise black.



break

Raytracer, step 3

- Render a sphere using specular lighting
- Domain Concepts
 - Lights
 - Unit Vector
 - Normal Vector
- Haskell Concepts
 - GADT (Generalized Algebraic Data Type)

Raytracer, step 3 - GADTs

- Definition: sum type where the type parameter(s) depend on the specific constructor used

```
data Expr a where
  EBool  :: Bool      -> Expr Bool
  EInt   :: Int       -> Expr Int
  EEqual :: Expr Int -> Expr Int -> Expr Bool
```



Raytracer, step 3 - GADTs

- Common use case for GADTs: expressing invariants
- Vector
 - Can be **zero vs. non-zero**
 - Can be **unit vector vs. arbitrary length**
- Certain operations work only on vectors that have specific invariants
 - Cannot normalize a zero vector
 - A normal vector is expected to always be unit length
- Certain operations retain invariants, others don't
 - Vector rotation: retains length / zeroness
 - Vector addition: does not retain length / zeroness

Raytracer, step 3 - GADTs

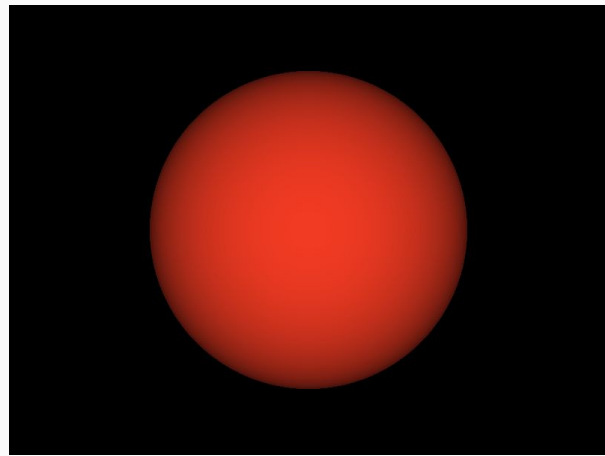
```
data Zero
data NonZero a
data UnitLength
data Unknown

data VectorGADT a where
  Zero :: VectorGADT Zero
  Unit :: Vector -> VectorGADT (NonZero UnitLength)
  NonZeroUnknownLength :: Vector -> VectorGADT (NonZero Unknown)
  Unknown :: Vector -> VectorGADT Unknown

normalize :: VectorGADT (NonZero a) -> VectorGADT (NonZero UnitLength)
rotate :: VectorGADT a -> Float -> VectorGADT a
add :: VectorGADT a -> VectorGADT b -> VectorGADT Unknown
nonZero :: VectorGADT a -> Maybe (VectorGADT (NonZero Unknown))
```

Raytracer, step 3 - “just do it”

- Specular lighting with a uniform light source
 - The color (intensity) is determined by the angle at which the light hits an object
 - Angle α between
 - the light source direction
 - the normal of the object at the point where the light hits
- Introduce
 - Normal vector for sphere (it's a unit vector!), i.e. perpendicular to the surface
 - Uniform light source (direction + intensity)
- Calculate specular lighting for a sphere
 - Intensity $\sim \cos^3 \alpha$
- Use vector dot product to calculate $\cos \alpha = v_1 \cdot v_2$
 - v_1 and v_2 must be unit vectors



break

Practical Haskell: a web service

Haskell for web services

Haskell is surprisingly well suited for network IO intensive applications

Why?

- Monadic IO >> Async IO
- Purity enables concurrency
- High performance runtime (MIO)
- Lots of high-quality libraries

Web Service Frameworks

Haskell has a variety of web frameworks, with various levels of complexity / completeness (*and levels of maintenance*)

- Core Web Server → Warp
 - HTTP(1/2)
 - Fast
 - Low-level
- Routing + high-level request / response logic
 - Scotty
 - Servant
- Batteries included frameworks
 - Yesod

Scotty example

Let's look at an example using Scotty which does:

- Routing
- Path parameter parsing
- JSON parsing + generation
- Cross-request state
 - Unique user id generation

Doesn't look to be async, but it is fully non-blocking behind the scenes

Scotty example

Performance is really good for a (very) high level language

- MIO, Haskell IO subsystem
 - Scales to 40+ cores
 - Typically scales better than Go, similar to Node.js with sufficient pre-forked processes
 - But no single-thread limitation for your logic
 - <https://share.google/uNaOe0mTpdaX2o8CW>
- Tell cabal to run your program across all cores
 - `ghc-options: -threaded -rtsopts -with-rtsopts=-N`

Scotty example

- Saturates all cores on 10 core Intel i7-13800H
- 500 connections, minimal work per request

```
$ wrk -c 500 -d 30 -t 8 http://localhost:3000/fast
Running 30s test @ http://localhost:3000/fast
 8 threads and 500 connections
Thread Stats      Avg          Stdev         Max    +/-  Stdev
   Latency       1.21ms       1.34ms     59.76ms    88.82%
   Req/Sec      51.09k        4.92k     63.64k     70.58%
12207630 requests in 30.04s, 2.08GB read
Requests/sec: 406430.48
Transfer/sec:      70.93MB
```

Scotty example

- 20 ms work per request
- 2000 connections, theoretical maximum of 100 000 req/s

```
$ wrk -c 2000 -d 30 -t 8 http://localhost:3000/slow/20
Running 30s test @ http://localhost:3000/slow/20
 8 threads and 2000 connections
  Thread Stats   Avg      Stdev     Max    +/-  Stdev
   Latency    24.18ms    4.06ms   84.46ms   87.17%
  Req/Sec    10.32k    1.11k   12.34k    72.00%
2465488 requests in 30.07s, 470.25MB read
Requests/sec: 81985.91
Transfer/sec:      15.64MB
```

Let's discuss