Haskell and the case for purely functional programming

Adrian Sieber

2020-10-29

Haskell Logo

- Large number of programming languages have been invented over the years
- Only a small number of popular languages make up the majority of the code written today

Why? Are they better than the rest?

Developers often choose more popular languages, because more popular means:

- Easy to get started
- A bigger ecosystem
- More eyes to find bugs
- More companies backing the language
- More documentation on the Internet
- Higher likelihood of future support
- If it wasn't any good it wouldn't have become so popular

But maybe it can just as well mean:

- Easy to get started, but not easy to maintain
- A big ecosystem of low quality
- Company pushed the language for self promotion
- More companies have to use the language in order to appear “modern”
- Hard to find high quality documentation
- Susceptible to superficial trends
- Popular because of an unfair monopoly (e.g. JavaScript in the Browser)

History of FP Languages

- 1920 - Combinatory Logic by Moses Schönfinkel and Haskell Curry
  A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments.
- 1930 - (Typed) Lambda Calculus by Alonzo Church
- 1936 - Turing Machine

- 1950 - *Lisp* - First functional programming language (but not pure!)
- 1973 - *ML* (Meta Language) - Uses Polymorphic Hindley–Milner type system
  - 1987 - *Caml* - Dialect of ML
  - 1996 - *OCaml* - Extends the Caml with object-oriented features
- 1985 - *Miranda* - Lazy & purely functional language, but proprietary

1990 - *Haskell* - Committee consolidated existing functional languages to serve as a basis for future research

- 2007 - *Idris* - Dependent types (list with x entries), totality checker
- 2012 - *Elm* - DSL for building webapps
- 2015 - *PureScript* - Compiles to general purpose JavaScript

What makes Haskell so great then?

Static Types with **Global** Type Inference

Every single expression in Haskell is statically (i.e. at compile time) typed.

Don't think "Java, C++"" => More like Python + Types

Types are globally inferred (i.e. across function and file boundaries).

$=>$ Explicitly stating type not necessary

$=>$ Write thousands of lines of code without a single type signature

```
> nameAndIsMember = ("John Doe", True)
> :t nameAndIsMember
nameAndIsMember :: ([Char], Bool)
```

Annotations not necessary!

Disclaimer

> *Examples in this article try to use equivalent language constructs for better comparability even though there might be more idiomatic versions in each language.*

```haskell
data ShirtSize = Small | Medium | Large

johnsSize = Medium

main =
  putStrLn (case johnsSize of
      Small -> "Eat more spinach!"
      Medium -> "You're just average."
      Large -> "Is the air thinner up there?"
    )
```

```haskell
data ShirtSize = Small | Medium | Large
```

- FP speak: Algebraic data type (sum type)
- Otherwise: Union type
- Create new types for every- and anything!

What happens if we later decide to add size `Huge` and update the first line to be:

```
data ShirtSize = Small | Medium | Large | Huge
```

```
shirt-size.hs:6:13: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In a case alternative: Patterns not matched: Huge
  |
6 |   putStrLn (case johnsSize of
  |             ^^^^^^^^^^^^^^^^...
```

GHC (Glasgow Haskell Compiler - the default compiler) warns us!

In other languages:

- Would have crashed at runtime if there was a person with a `Huge` shirt size.
- Would just ignore it silently E.g. Java with an `enum`, a `switch` statement, and no `default` case.

How would you even model the `ShirtSize` type in dynamic languages?

With strings?

Who would save you from typos?

In Haskell on the other hand:

```
shirt-size.hs:3:13: error:
    • Data constructor not in scope: Medum
    • Perhaps you meant ‘Medium’ (line 1)
  |
3 | johnsSize = Medum
  |             ^^^^^
```

Even if you find a workaround (Python supports enums since 3.4) it will hardly be as elegant and straight forward as in Haskell.

Expressive type system + powerful compiler => data modeling is one of Haskell's core strengths

Pure Functions

All functions in Haskell are pure.

- Return value depends only on its input arguments
- Function can not perform any side effects (if not explicitly stated)

Short example why this is desirable:

You see following innocent function in a big code base. **What does it do?**

```
const alertText = reverseAndShout('JS is the best!')
```

```
// Reverses and capitalizes a string
// (and is evil)
function reverseAndShout (string) {
  const reversed = string.split('').reverse().join('')
  console.info(`Reversed: ${reversed}`)  // IO
  const shouted = reversed.toUpperCase()
  dataBase.write(shouted)  // Might cause an error
  lastShouted = shouted  // Mutating global state
  return shouted
}
```

Haskell

```haskell
reverseAndShout :: Text -> Text
reverseAndShout text =
  let
    reversed = reverse text
    shouted = toUpper reversed
  in
    shouted
```

Want to print to terminal?

```
reverseAndShout :: Text -> IO Text
```

Want to change global state?

```
reverseAndShout ::
  (GlobalState, Text) -> (GlobalState, Text)
```

Sidenote

Type signature `reverseAndShout :: Text -> Text` is optional, but considered good practice, as it acts as documentation and improves error messages during development.

Tell the compiler what your function is supposed to do and it will help you implement it!

Sidenote 2

The idiomatic version of the function would actually be:

```
reverseAndShout = toUpper . reverse
```

This is called “pointfree style”

Strong Immutability

Guarantee that values can not be mutated (changed).

```
function reverseAndShoutName (person) {
  person.fullName = person.firstName + person.lastName
  setTimeout(
    () => {delete person.firstName; person.lastName = 'Smith'},
    1
  )
  return person.fullName
    .split('').reverse().join('')
    .toUpperCase()
}

const john = {firstName: 'John', lastName: 'Doe'}

console.log(reverseAndShoutName(john))
console.log(john.fullName)
setTimeout(() => console.log(john), 5)
```

Output of the program when running with `node`:

```
$ node reverseAndShoutName.js
EODNHOJ
JohnDoe
{ lastName: 'Smith', fullName: 'JohnDoe' }
```

You have absolutely **no idea or guarantees** what will happen to a object during and after a function call **without carefully checking the complete code** of the function!

```haskell
data Person = Person
  {firstName :: Text, lastName :: Text}

reverseAndShoutName :: Person -> Text
reverseAndShoutName person =
  let fullName = firstName person <> lastName person
  in fullName & reverse & toUpper

john :: Person
john = Person "John" "Doe"

main :: IO ()
main = putStrLn (reverseAndShoutName john)
```

Maybe it feels like I'm just not implementing the same code?

That's the point!

With

- `data Person = …`
- `reverseAndShoutName :: Person -> Text`
- `john :: Person`
- `main :: IO ()`

there is really not much I can do to break the code.

What if you want to use a changed version of John?

You make a copy with a changed field!

```
john = Person "John" "Doe"
john2 = john {lastName = "Smith"}
```

And don't worry about performance: GHC heavily optimizes such scenarios by reusing existing elements.

Lazy Evaluation

- Most unique feature of Haskell
- Only major language (besides “Miranda”) which is lazy per default

Some artificial Python code:

```
valueA = expensiveComputation()
valueB = anotherExpensiveComputation()

if valueToPrint == 'A':
  print(valueA)
else:
  print(valueB)
```

Badly implemented:

- Whatever value is supposed to be printed $=>$ calculates both of them
- Python eagerly evaluates the code as soon as a line of code is executed

Haskell

- GHC registers how `valueA` and `valueB` can be computed
- Starts to evaluate the code as soon as the value is needed
- It evaluated the code lazily

```
valueA = expensiveComputation
valueB = anotherExpensiveComputation

main =
  putStrLn (if valueToPrint == "A"
    then valueA
    else valueB)
```

It becomes harder and harder to notice such missteps in a larger code base.

Haskell does the right thing!

Cool side effect of lazy evaluation: **Infinite lists**

```
allNumbers = [1..]
allNumbersDoubled = allNumbers & map (*2)

main =
  allNumbersDoubled
  & take 5
  & print

-- Prints [2, 4, 6, 8, 10]
```

There is no other programming language where this can be written as concisely and beautifully.

And don't tell me be about Python's list comprehension. This was actually invented by Haskell and is still supported, but considered bad practice. A few simple functions can achieve the same thing more readable and without the overhead of introducing another syntax construct.

The Rest

- Best REPL (maybe except for some Lisps)
  - Functions don't interact with a global state => perfectly suited for being tweaked and tested in the REPL
- Unobtrusive Syntax
  - Prime candidate for writing EDSLs (generate a non type safe language like e.g. HTML or CSS in a safe way, while keeping the looks of the original language.)

github.com/hadolint/language-docker lets you write Dockerfiles in Haskell, which look like normal Dockerfiles, but now they are type safe and it's harder to make mistakes.

```
import Language.Docker

main = putStr $ toDockerfileStr $ do
  from "node"
  run "apt-get update"
  runArgs ["apt-get", "install", "something"]
```

- Many syntax constructs are actually just normal functions
- Defined in the standard library “Prelude”

e.g. the `&` I used before is actually just a function with the signature:

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

- Takes a value of some type `a`
- Applies a function to the value
- Returns the result of type `b`

And voila: A function which reverses the application order

- Awesome ecosystem of packages on stackage.org
- Excellent quality of 3rd party tools => Haskell developers care a lot about writing correct, stable, and secure software
- Good documentation through default documentation generator "Haddock"
- Awesome community
- Considered "Best in class" for parallelism and concurrency

Maintainability and Refactorability

Quote from Gabriel Gonzalez:

> *Haskell is unbelievably awesome for maintaining large projects. There's nothing that I can say that will fully convey how nice it is to modify existing Haskell code. You can only appreciate this through experience.*

The compiler is guiding one through any refactoring based on the types.

The Not so Good

- `[Char]` vs `String` vs `Text` vs `ByteString` vs ...
- Laziness can apparently cause excessive memory usage
- No packages for niche software
- Cold compilation times

Example Apps - Most Starred Projects on GitHub

- **ShellCheck** - Static analysis tool for shell scripts
- **Pandoc** - Universal document converter
- **Hasura** - Blazing fast, instant realtime GraphQL APIs on Postgres
- **PostgREST** - Automatic REST API for any Postgres database
- **github/semantic** - Parsing, analyzing, and comparing source code for many languages
- **PureScript** - Strongly-typed language that compiles to JavaScript.
- **Elm** - Strongly-typed language that compiles to JavaScript.
- **facebook/Haxl** - Simplify access to remote data (E.g. databases and web-based services)

Try It Out

- tryhaskell.org
- repl.it