The other side of functional programming

Haskell: purity, types, and a damn good time

In the beginning

- Haskell was developed by academics to unify many streams of research
- Committee began work in 1987
- Haskell Report 1.0 published April 1, 1990
- Comparable in age with Erlang

Principal concerns

- Laziness
- Purity
- Strong, static types

Being lazy with style

Laziness

- The original unifying theme of the designers
- Evaluate an expression when its result is needed
  - Evaluate only the minimum needed

A simple lazy example

- A simple Haskell function definition

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

- Define the name `square`
- Give it a free variable `x`
- The function body follows the =

Evaluate in a lazy world

- What is the result of `square 3` ?
  - The number `9` ?
  - Or the unevaluated expression `3 * 3` ?

- (It's the latter)

When do we evaluate?

- When does the expression `3 * 3` turn into something meaningful?
- For instance, when we need to print its result
- Evaluation is driven by *need*
    - Often referred to as *call-by-need*
    - Contrast with the more familiar *call-by-value*

Laziness as *default*

- Laziness is pervasive in Haskell code
  - But sometimes it is not desirable
- Option: use strict evaluation when necessary

- Many strict languages provide optional laziness
  - The apparent gulf isn't so big after all

Purity is the new black

Purity

- Haskell data is immutable
- Functions are pure
  - Only affected by their inputs
  - Not subject to mutable global state

(Again, these are *defaults*: mutability is available as an option)

Why purity?

- What's a side effect?
  - Mutating global state
  - Performing I/O
- Remember laziness?
  - Evaluation by need
- Laziness and side effects don't mix!

Laziness needs purity

- Haskell chose laziness by default
  - Therefore purity was inescapable
- This has *big* consequences
  - Composability: glue functions together
  - Safety: functions are black boxes
- Arguably a more important choice than laziness!

Adventures with types

Strong static types

- Valid Haskell expressions are assigned *types* at compile time

```
a :: String
a = “some text”
```

- The `::` says that `a` has the type `String`
  - This is called a type annotation

Wait ... *static* types?

- Aren't we supposed to hate static types?
- Didn't types cause us RSI in Java and C++?
- Wasn't that part of why we escaped to the dynamically typed languages?

- Crummy languages give static types a bad name

Yes, *static* types!

- A Haskell compiler *infers* the type of an expression
  - It does this automatically
- The type annotations that you’ve seen are *optional*
  - Handy for documentation, but superfluous

Simple use of types

- Any sensible language will reject stuff like this

  ```
  1 + "3foo"
  ```

  (Notable exception: Perl)

- Dynamic languages barf at runtime
- Languages like Haskell reject at compile time

Pattern matching

- Here's the classic way to calculate a list's length

```
length []       = 0
length (x:xs) = 1 + length xs
```

- We've defined a function using two equations
  - Choose which to use by input *structure*

Matching on structure

- If the input list is empty, the length is 0

```
length []       = 0
```

- If the input matches the list constructor `:`, bind the name `xs` to the list’s tail and recurse

```
length (x:xs) = 1 + length xs
```

Typing a list

- What is the type of `length`?

  `length :: [a] → Int`

- The `[a]` above means “a list of values of some unknown type `a`”

- The → means “returns”

- In other words, we have a function that does not know or care about the elements of its input list

Why use static types?

- Static types are about more than catching mistakes
- They let the compiler make complex decisions about the program’s behaviour

User-defined containers

- Here's a widely used Haskell type

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

- We can pattern-match to inspect the structure of a user-defined type

```
isJust (Just x) = True
isJust Nothing  = False
```

Algebraic data types

```
data ClientError =
    BadRequest
  | Unauthorized
  | Forbidden
  | NotFound
    ...
```

What does this buy?

- If my function takes a `HttpResponse`
  - The compiler *guarantees* that I'll never be given a `HttpRequest`
  - It *guarantees* that I'll never see an unknown `HttpResponse`
  - It *warns me* if a pattern match omits a valid response

Safety with types

- Static types give stronger guarantees than testing
- A simple example:
  - “I know my function can never receive an argument of an invalid type”
- More ambitious:
  - “This code can never perform I/O”

More serious type safety

- We can *omit* features that other languages bake in
  - Ship them as libraries instead
- A recent example:
  - Java-style checked exceptions *as a library*
  - Throwable exceptions are inferred

More serious types

- We can model and enforce complex behaviour
- Examples:
  - Information only flows from less secure to more secure code
  - Communicating processes follow a well-defined messaging protocol

Real world concerns

Performance

- Haskell is ranked #3 on the Alioth Shootout
  - Usually within 1x to 5x of C's performance
  - Great profiling tools help with tuning
- It's easy to write fast, concise Haskell
  - Community knowledge of *how* is a bit scattered

Going native

- Haskell has a beautiful FFI
  - Call into and out of C code easily
- Nifty libraries for other languages
  - Interop with .NET
  - Act as an Erlang node

Concurrency

- Haskell has a fantastic concurrent runtime
- Works with multiple cores
- Millions of concurrent threads
- Advanced, but easy to use programming model
- The default choices of immutable data and pure functions *really help* to write correct, scalable code

Thread synchronisation

- Software Transactional Memory
  - Database-like transactional concurrency to regular code
  - Much safer than mutexes
- Strongly typed message channels
  - Networked message support as a library

Parallel terminology

- Parallel and concurrent programming are *different*
    - Parallel: how do I get one answer faster?
    - Concurrent: how do I do 80,000 different things per second?

Parallel programming

- Mature support for making pure code parallel
- Development version of GHC scales well on modern multicore boxes
- Exciting research abounds
  - Nested data parallel vector code
  - GPU offload

Testing and assurance

- The famous QuickCheck library arose in Haskell
  - Randomised property-based testing
  - Beats the pants off unit tests when applicable
- Traditional unit testing libraries available too
- Excellent code coverage analysis tools

Libraries

- Over 1,000 libraries on http://hackage.haskell.org/
  - Game engines, bioinformatics, networking, database integration, music, compiler tools, ...
- Single-command install of any library and its dependencies

Community

- The best language community I know of
- Stellar researchers, informed OSS hackers
  - Atmosphere is friendly, welcoming, and smart
  - Notable absence of rock stars
- #haskell is 5th biggest channel on Freenode
- Many great online learning resources

Thank you for your time!