Jane Street

A Tale of Two Lambdas: A Haskeller's Journey into OCaml

Richard A. Eisenberg
Jane Street
reisenberg@janestreet.com

Friday, 17 October 2025
Haskell Symposium
Singapore

Goal

Spark conversation

But I need your help!

Context

Sep 2011:
First exposure to Haskell

Dec 2015:
-XTypeInType

July 2022:
Learn OCaml

Mar 2025:
OxCaml

May 2012:
First commit to GHC (kinds in Template Haskell)

Jan 2021:
Chair of Haskell Foundation

Aug 2022:
Start at Jane Street

Sep 2025:
Haskell Symposium

(not strictly to scale)

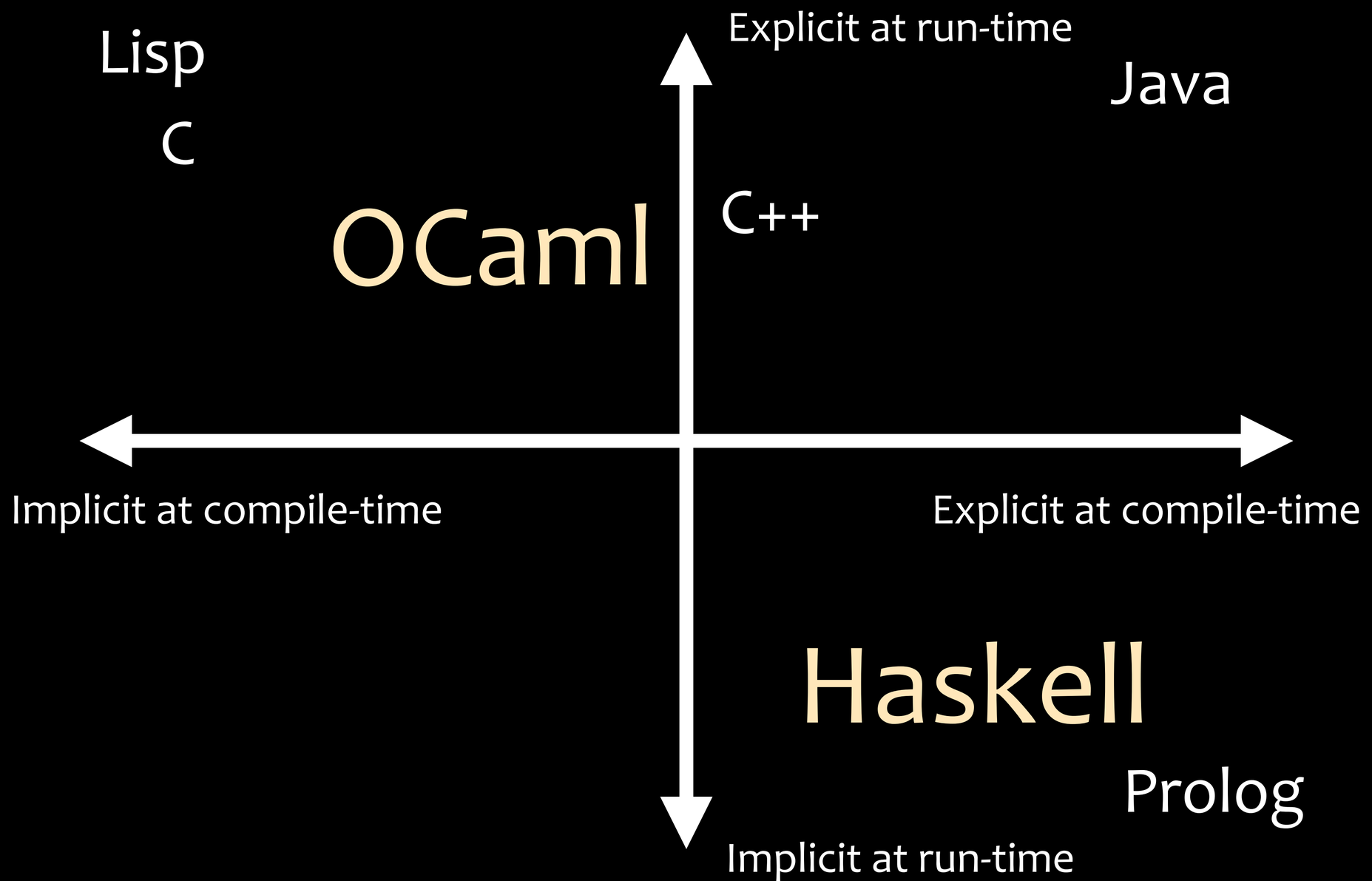
I have opinions.

You do, too.

Major Insight

Both languages have much to learn from each other.

Minor, Probably Wrong Insight



Run-time Semantics

Haskell

Laziness
Type classes
Optimizer

OCaml

Strictness
Modules
Predictable performance

Example 1

```
render (triangulate polygons)
```

Haskell:

- When to triangulate? (laziness makes this hard)
- Does the result depend on type inference (due to type classes)?

OCaml:

- Simple!
- (but maybe slower)

Opinion

Laziness is confusing

Explicit laziness is lovely

but automatic forcing is nice

Opinion

Haskell: Allow users to require every field of every constructor to have a laziness/strictness annotation.

OCaml: Allow users to opt into lighter-weight laziness, without explicit forcing and (possibly) without a `lazy_t` type.

if this can be made performant

Example 2

```
map f (filter p input)
```

Does the intermediate structure get built?

Haskell: probably not
(but this is hard to know)

OCaml: yes

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. Haskell Workshop 2001.

Opinion

Optimizers are good, but fragile.

You need to check your work.

Try godbolt.org!

Type Inference (opinion)

OCaml type inference tries harder

... but GHC is more predictable.

Example 3

Haskell

```haskell
sumList :: [Int] -> Int
sumList []       = 0
sumList (x : xs) =
  x + sumList xs
```

OCaml

```ocaml
let rec sum_list = function
| [] -> 0
| x :: xs -> x + sum_list xs
```

Example 4

Haskell

```
add1 :: a -> a
add1 x = x + 1
```

❌

OCaml

```
let add1 : 'a -> 'a =
  fun x -> x + 1
```

✅ with `add1 : int -> int`

```
let add1 : 'a. 'a -> 'a =
  fun x -> x + 1
```

❌

Example 5

Haskell

```
data T a where
  Mk1 :: T Bool

f Mk1 = 4
```

❌

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, Martin Sulzmann. OutsideIn(X): Modular type inference with local assumptions. JFP 2011.

OCaml

```
type _ t =
  | Mk1 : bool t

let f x = match x with
  | Mk1 -> 4
```

✅ with `f : bool t -> int`

Jacques Garrigue and Didier Rémy. Ambivalent Types for Principal Type Inference with GADTs. ASPLAS 2013.

Example 6

Haskell

```
data T a where
  Mk1 :: T Bool
  Mk2 :: T Int

f Mk1 = 4
f Mk2 = 5
```

OCaml

```
type _ t =
  | Mk1 : bool t
  | Mk2 : int t

let f x = match x with
  | Mk1 -> 4
  | Mk2 -> 5
```

❌ what's the type of x?

Example 7

Haskell

```
data T a where
  Mk1 :: T Bool
  Mk2 :: T Int

f :: T a -> Int
f Mk1 = 4
f Mk2 = 5
```

✅

OCaml

```
type _ t =
  | Mk1 : bool t
  | Mk2 : int t

let f (type a) (x : a t) =
  match x with
  | Mk1 -> 4
  | Mk2 -> 5
```

✅

Type classes
Example 8

Haskell

```
f list1 list2 = do
  action1 list1
  action2 list2
```

What monad does this work in?
We know only by type inference.

Modules
Example 9

OCaml

```
module F (X : S) : sig
  include X
  include module type of F0(X)
  include S2 with module M := X
end
```

What is exported by `F(M0)`? Oof.

Type Classes vs. Modules

Both offer ways of abstracting over types with complex interfaces.

Opinion

Type classes or modules?

Both!

Modular implicits

Leo White Frédéric Bour Jeremy Yallop

We present *modular implicits*, an extension to the OCaml language for ad-hoc polymorphi

(2015)

Opinion

Type classes or modules?

Both!

Haskell

Add namespace control...
but not proper modules

OCaml

Add type classes...
but without global coherence

And add higher-kinded types,
reducing the need for modules

Purity

Haskell case study: zonking in GHC

```
data Type =
  | UnifVar { name :: Name
            , value :: IORef (Maybe Type) }
  | ...
```

Opinion: Maintaining the pure/ impure separation is annoying.

Purity

Opinion: Maintaining the pure/ impure separation is annoying.

Opinion: But not having purity in OCaml is worse!

A Middle Ground?

OxCaml tracks purity more finely, focusing on immutability and referential transparency, not the lack of side effects.

Scope
Example 11

Haskell

```
nextFib n1 n2 =
  let n2 = n1 + n2
      n1 = n2 - n1
  in
  (n1, n2)
```

loops!

OCaml

```
let next_fib n1 n2 =
  let n2 = n1 + n2 in
  let n1 = n2 - n1 in
  n1, n2
```

✅

Opinion

Recursion should
be explicit.

Opinion

Recursion should be explicit.

Example 11

Haskell

```
f :: Int -> Int -> Int
f acc x = ...

f :: Int -> Int
f x = f 0 x
```

❌

OCaml

```
let f acc x = ...

let f x = f 0 x
```

✅

Opinion: bindings should be ordered.

Example 12

Haskell

```
import Data.Set
import Data.Map

e = empty
```

❌

OCaml

```
open Set.Make(Int)
open Map.Make(Int)

let e = empty
```

✅

Opinion: bindings should be ordered.

Example 13

Haskell

```
data T1 = A | B
data T2 = A | B

f A = 1
f B = 2
```

❌

OCaml

```
type t1 = A | B
type t2 = A | B

let f = function
  | A -> 1
  | B -> 2
```

✅, with `f : t2 -> int`

Opinion: bindings should be ordered.

Example 14

Haskell

```haskell
data T1 = A | B
data T2 = A | B

f :: T1 -> Int
f A = 1
f B = 2
```

❌

OCaml

```ocaml
type t1 = A | B
type t2 = A | B

let f : t1 -> _ = function
  | A -> 1
  | B -> 2
```

✅, with `f : t1 -> int`

Opinion: Type-directed name resolution is good.

Syntax

Haskell

- `camelCase`

OCaml

- `snake_case`

Opinion: Use both!
One for locals and one for globals

Syntax

Haskell

- `camelCase`
- `where`

OCaml

- `snake_case`
- only `let`

```
f x y = importantStuff
  where
    fiddly = ...
    little = ...
    details = ...
```

Opinion: I miss `where`.

OCaml should allow this.

(But only with purity.)

Syntax

Haskell

- `camelCase`
- `where`
- Indentation-sensitive

OCaml

- `snake_case`
- only `let`
- Whitespace-agnostic

Opinion: 🤷

Syntax

Haskell

- camelCase
- where
- Indentation-sensitive

OCaml

- snake_case
- only let
- Whitespace-agnostic

Opinion: 😱.

Let's just have 1 (or maybe 2).

- 5 namespaces:

  modules, variables, constructors, types, type variables

- 11 namespaces:

  module types, modules, classes, methods, variables, constructors, types, type variables, polymorphic variant constructors, record labels

Implementation

GHC

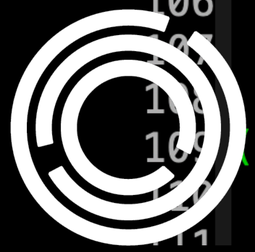
Typed intermediate language

- Safety net
- Easy way to add sound extensions

OCaml

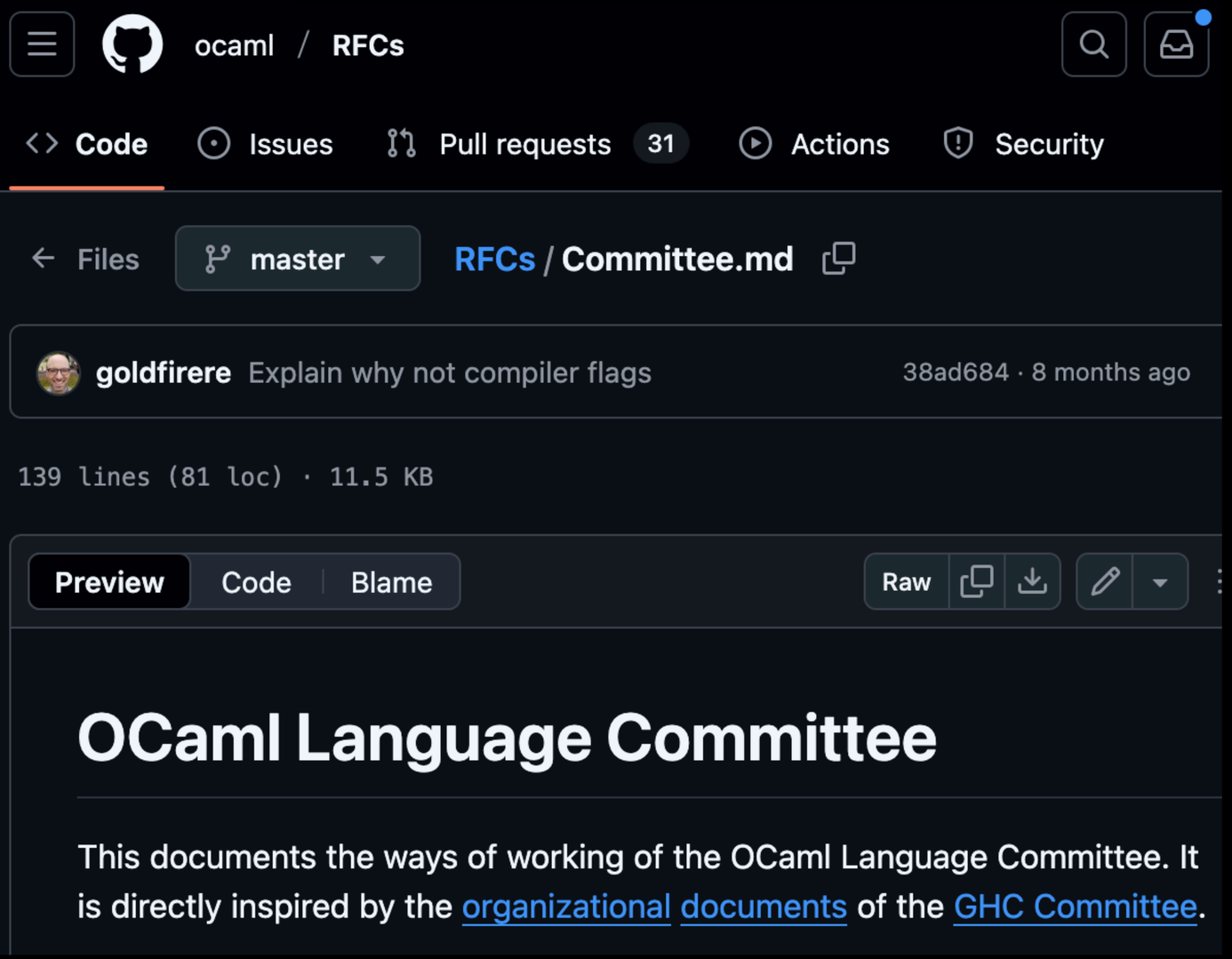
Performance.
Compiler builds in < 5 minutes.

```
{- Note [Quick Look overview]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
The implementation of Quick Look closely follows the QL paper
   A quick look at impredicativity, Serrano et al, ICFP 2020
   https://www.microsoft.com/en-us/research/publication/a-quick-look-at-impredicativity/

All the moving parts are in this module, GHC.Tc.Gen.App, so named
because it deal with n-ary application.  The main workhorse is tcApp.

Some notes relative to the paper

(QL1) The "instantiation variables" of the paper are ordinary unification
  variables.  We keep track of which variables are instantiation variables
  by giving them a TcLevel of QLInstVar, which is like "infinity".

(QL2) When we learn what an instantiation variable must be, we simply unify
  it with that type; this is done in qlUnify, which is the function mgu_ql(t1,t2)
  of the paper.  This may fill in a (mutable) instantiation variable with
  a polytype.

(QL3) When QL is done, we turn the instantiation variables into ordinary unification
  variables, using qlZonkTcType.  This function fully zonks the type (thereby
  revealing all the polytypes), and updates any instantiation variables with
  ordinary unification variables.
  See Note [Instantiation variables are short lived].

(QL4) We cleverly avoid the quadratic cost of QL, alluded to in the paper.
  See Note [Quick Look at value arguments]
```



Community


ghc-proposals / ghc-proposals
Code
Issues 59
Pull requests 124
Proposed compiler and language changes for GHC an


ocaml / RFCs
Code
Issues
Pull requests 31
Actions
Security
Files
master
RFCs / Committee.md
goldfirere Explain why not compiler flags
38ad684 · 8 months ago
139 lines (81 loc) · 11.5 KB
Preview
Code
Blame
Raw
OCaml Language Committee
This documents the ways of working of the OCaml Language Committee. It is directly inspired by the organizational documents of the GHC Committee.

Funding

Haskell

Gold

INPUT | OUTPUT · JUSPAY · MERCURY · standard chartered

Silver

TWEAG · Well-Typed The Haskell Consultants

Bronze

channable · DigitalOcean · Google · QBayLogic · TripShot

OCaml

OCaml Software Foundation

PLATINUM

Jane Street · ahrefs · Tezos

SILVER

LexiFi · Bloomberg

BRONZE

SimCorp · MITSUBISHI ELECTRIC *Changes for the Better*

Funding

Haskell

Full-time
executive director

(with desire to expand)

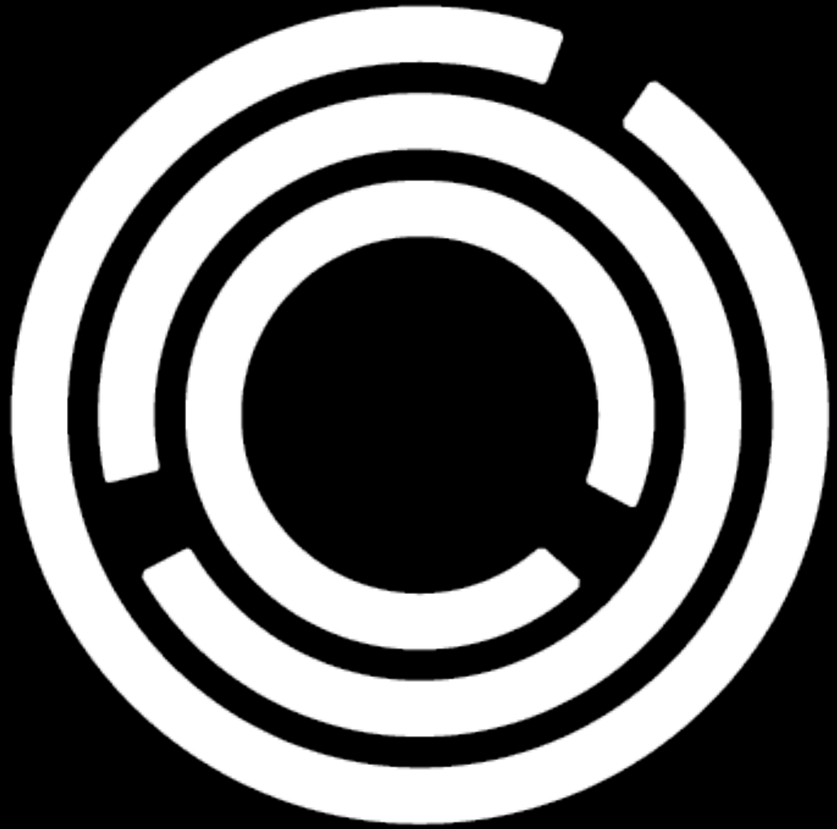
OCaml

Volunteer
organization

Funding

Haskell

OCaml



Shameless Plug

AmeriHac Haskell Hackathon

Jane Street Office

New York, NY

February 7-8

Free!

Controversial Proposal

Merge Haskell Symposium and ML Workshop into the Milner Language Symposium, accepting research on (or with) strongly-typed, inferred, functional languages.

Resurrect a 1-day Haskell Workshop.
Have an other-ML track at OCaml Workshop.

Create new conversations.

Jane Street

A Tale of Two Lambdas: A Haskeller's Journey into OCaml

Richard A. Eisenberg
Jane Street
reisenberg@janestreet.com

Friday, 17 October 2025
Haskell Symposium
Singapore