Lightweight higher-kinded polymorphism

Jeremy Yallop and Leo White

UNIVERSITY OF CAMBRIDGE

FLOPS 2014

Pining for Haskell

`(the story)`

Pining for Haskell: laziness?

```
fibs =
  0 : 1 : zipWith (+) fibs (tail fibs)
```

Pining for Haskell: laziness?

```
fibs =
  0 : 1 : zipWith (+) fibs (tail fibs)
```

```
let rec fibs =
  lazy (0 :: lazy (1 :: zipWith (+) fibs (tail fibs)))
```

Pining for Haskell: type classes?

```
find x l = case l of
  (y, z) : tl → if x == y then Just z
                else find x tl
  [] → Nothing
```

Pining for Haskell: type classes?

```
find x l = case l of
  (y, z) : tl → if x == y then Just z
                else find x tl
  [] → Nothing
```

```
let rec find (==) x = function
 | (y, z) :: tl → if x == y then Some z
                  else find (==) x tl
 | [] → None
```

Pining for Haskell: purity?

```
swap (x, y) = (y, x)
```

Pining for Haskell: purity?

```
swap (x, y) = (y, x)
```

```
let swap (x, y) = (y, x)
```

monads in Haskell *vs* monads in ML

`(the motivating example)`

Monads in Haskell

```
when c a = if c then a else return ()
```

Monads in Haskell

```
when :: Monad (m :: * → *) ⇒ Bool → m () → m ()
when c a = if c then a else return ()
```

Monads in Haskell . . . and ML

```
when :: Monad (m :: * → *) ⇒ Bool → m () → m ()
when c a = if c then a else return ()
```

```
module When (M: MONAD) =
struct
  let whn c a = if c then a else M.return ()
end
```

Monads in Haskell, continued

```
unless c a = when (not c) a
```

```
when True []
```

Monads in Haskell . . . and ML, continued

```
unless c a = when (not c) a

when True []
```

```
module Unless (M: MONAD) =
struct
  module When_M = When(M)
  let unless c a = When_M.whn (not c) a
end

let module WhenList = When(List_monad) in
  length (WhenList.whn true [])
```

Why all the verbosity? What’s missing?

`(the crisis)`

Why all the verbosity? What's missing?

type classes

higher kinds

Why all the verbosity? What's missing?

type classes

`Show a → a → String` ✓

higher kinds

Why all the verbosity? What's missing?

type classes

`Show a → a → String` ✓

`Show a ⇒ a → String` ✗

higher kinds

Why all the verbosity? What's missing?

type classes

`Show a → a → String` ✓

`Show a ⇒ a → String` ✗

higher kinds

`Maybe Int` ✓

Why all the verbosity? What's missing?

type classes

`Show a → a → String` ✓

`Show a ⇒ a → String` ✗

higher kinds

`Maybe Int` ✓

`Maybe a` ✓

Why all the verbosity? What's missing?

type classes

`Show a → a → String` ✓

`Show a ⇒ a → String` ✗

higher kinds

`Maybe Int` ✓

`Maybe a` ✓

`f Int` ✗

Why all the verbosity? What's missing?

type classes

`Show a → a → String` ✓

`Show a ⇒ a → String` ✗

higher kinds

`Maybe Int` ✓

`Maybe a` ✓

`f Int` ✗ ⟵ **this talk**

Defunctionalization

`(the inspiration)`

Defunctionalization: eliminating higher-order functions

```
(* fold : ('a × 'b → 'b) → 'b → 'a list → 'b *)
let rec fold cons nil = function
  [] → nil
| x :: xs → cons (x, fold cons nil xs)

let sum l = fold (fun (x, y) → x + y) 0 l

let map f l = fold (fun (x, xs) → f x :: xs) [] l
```

Defunctionalization: eliminating higher-order functions

```
(* fold : ('a × 'b → 'b) → 'b → 'a list → 'b *)
let rec fold cons nil = function
  [] → nil
| x :: xs → cons (x, fold cons nil xs)

let sum l = fold (fun (x, y) → x + y) 0 l

let map f l = fold (fun (x, xs) → f x :: xs) [] l
```

Defunctionalization: eliminating higher-order functions

```
(* fold : ('a × 'b → 'b) → 'b → 'a list → 'b *)
let rec fold cons nil = function
  [] → nil
| x :: xs → cons (x, fold cons nil xs)

let sum l = fold (fun (x, y) → x + y) 0 l

let map f l = fold (fun (x, xs) → f x :: xs) [] l
```

Defunctionalization: eliminating higher-order functions

```
(* fold : ('a × 'b → 'b) → 'b → 'a list → 'b *)
let rec fold cons nil = function
  [] → nil
| x :: xs → cons (x, fold cons nil xs)

let sum l = fold (fun (x, y) → x + y) 0 l

let map f l = fold (fun (x, xs) → f x :: xs) [] l
```

Defunctionalization: eliminating higher-order functions

```
(* fold : ('a × 'b, 'b) arr → 'b → 'a list → 'b *)
let rec fold cons nil = function
  [] → nil
| x :: xs → apply (cons, (x, (fold cons nil xs)))

let sum l = fold Plus 0 l

let map f l = fold (FCons f) [] l
```

Defunctionalization: eliminating higher-order functions

```
(* fold : ('a × 'b, 'b) arr → 'b → 'a list → 'b *)
let rec fold cons nil = function
  [] → nil
| x :: xs → apply (cons, (x, (fold cons nil xs)))

let sum l = fold Plus 0 l

let map f l = fold (FCons f) [] l

type ('a, 'b) arr =
  Plus : (int × int, int) arr
| FCons : ('a → 'b) → ('a × 'b list, 'b list) arr

let apply : type a b. (a, b) arr × a → b = function
  Plus, (x, y) → x + y
| FCons f, (x, xs) → f x :: xs
```

Defunctionalization: eliminating higher-order functions

Function types

`a → b`

Defunctionalization: eliminating higher-order functions

Function types

$$\texttt{a} \to \texttt{b} \quad \rightsquigarrow \quad \texttt{(a, b) arr}$$

Defunctionalization: eliminating higher-order functions

Function types

`a → b` ⇝ `(a, b) arr`

Building functions

```
fun (x, y) → x + y
```

Defunctionalization: eliminating higher-order functions

Function types

$$\texttt{a} \rightarrow \texttt{b} \quad \rightsquigarrow \quad \texttt{(a, b) arr}$$

Building functions

$$\textbf{fun}\ (\texttt{x}, \texttt{y}) \rightarrow \texttt{x + y} \quad \rightsquigarrow \quad \texttt{Plus}$$

Defunctionalization: eliminating higher-order functions

Function types

```
a → b    ⇝    (a, b) arr
```

Building functions

```
fun (x,y) → x + y    ⇝    Plus
```

Calling functions

```
f (x, y)
```

Defunctionalization: eliminating higher-order functions

Function types

```
a → b          ⇝        (a, b) arr
```

Building functions

```
fun (x,y) → x + y     ⇝    Plus
```

Calling functions

```
f (x, y)      ⇝    apply (f, (x, y))
```

Eliminating higher-kinded type expressions

`(the trick)`

Eliminating higher-kinded type expressions

```
(* 'm monad → bool → unit 'm → unit 'm *)
let whn m c a =
  if c then a else m#return ()

let unless m c a =
  whn m (not c) a

length (whn mlist true [])
```

Eliminating higher-kinded type expressions

```
(* 'm monad → bool → unit 'm → unit 'm *)
let whn m c a =
  if c then a else m#return ()

let unless m c a =
  whn m (not c) a

length (whn mlist true [])
```

Eliminating higher-kinded type expressions

```
(* 'm monad → bool → unit 'm → unit 'm *)
let whn m c a =
  if c then a else m#return ()

let unless m c a =
  whn m (not c) a

length (whn mlist true [])
```

Eliminating higher-kinded type expressions

```
(* 'm monad → bool → unit 'm → unit 'm *)
let whn m c a =
  if c then a else m#return ()

let unless m c a =
  whn m (not c) a

length (whn mlist true [])
```

Eliminating higher-kinded type expressions

```
(* 'm monad → bool → (unit, 'm) app → (unit, 'm) app *)
let whn m c a =
  if c then a else m#return ()

let unless m c a =
  whn m (not c) a

length (Lst.prj (whn mlist true (Lst.inj [])))
```

Eliminating higher-kinded type expressions

Type applications

```
a m
```

Eliminating higher-kinded type expressions

Type applications

`a m` ⇝ `(a, m) app`

Eliminating higher-kinded type expressions

Type applications

```
a m    ⇝    (a, m) app
```

Using type constructors polymorphically

```
whn d c []
```

Eliminating higher-kinded type expressions

Type applications

```
a m    ⇝    (a, m) app
```

Using type constructors polymorphically

```
whn d c []    ⇝    whn d c (Lst.inj [])
```

Eliminating higher-kinded type expressions

Type applications

```
a m    ⇝    (a, m) app
```

Using type constructors polymorphically

```
whn d c []    ⇝    whn d c (Lst.inj [])
```

Using applied constructors monomorphically

```
whn d c l
```

Eliminating higher-kinded type expressions

Type applications

`a m` ⇝ `(a, m) app`

Using type constructors polymorphically

`whn d c []` ⇝ `whn d c (Lst.inj [])`

Using applied constructors monomorphically

`whn d c l` ⇝ `Lst.prj (whn d c l)`

The *higher* interface

`(the library)`

The *higher* interface

```
type (_, _) app
```

The *higher* interface

```
type (_, _) app

module type NEWTYPE = sig
  type _ s
  type t
  val inj : 'a s → ('a, t) app
  val prj : ('a, t) app → 'a s
end
```

The *higher* interface

```
type (_, _) app

module type NEWTYPE = sig
  type _ s
  type t
  val inj : 'a s → ('a, t) app
  val prj : ('a, t) app → 'a s
end

module Newtype(T : sig type 'a t end) :
  NEWTYPE with type 'a s = 'a T.t
```

The *higher* interface

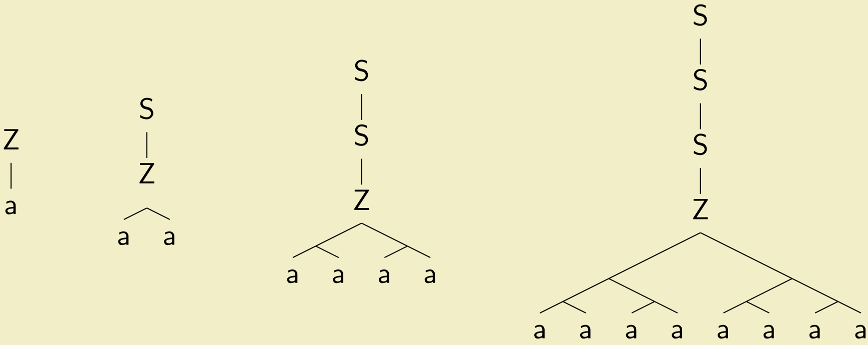
```
type (_, _) app

module type NEWTYPE = sig
  type _ s
  type t
  val inj : 'a s → ('a, t) app
  val prj : ('a, t) app → 'a s
end

module Newtype(T : sig type 'a t end) :
  NEWTYPE with type 'a s = 'a T.t
```

```
module Lst = Newtype(struct type 'a t = 'a list end)
```

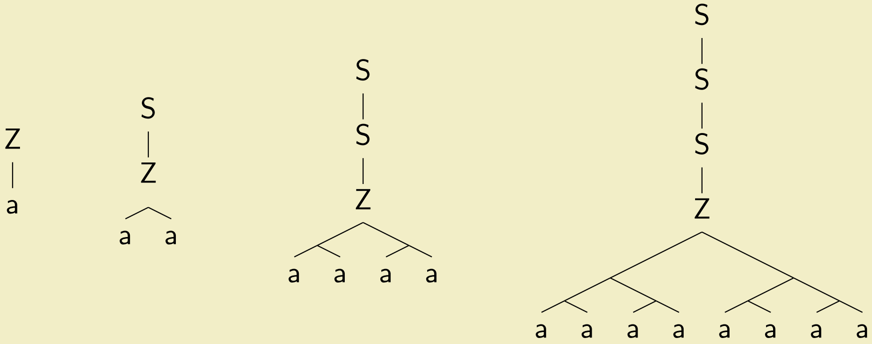
higher: not just for monads

`(the claim of generality)`

Perfect trees



Perfect trees: folds



z a

s (*z* (a,a))

s (*s* (*z* ((a,a),
(a,a))))

s (*s* (*s* (*z* ((a,a),(a,a)),
(a,a),(a,a))))

Perfect trees

```
type 'a perfect =
  Zero : 'a → 'a perfect
| Succ : ('a × 'a) perfect → 'a perfect
```

Perfect trees: folds

```
type 'a perfect =
  Zero : 'a → 'a perfect
| Succ : ('a × 'a) perfect → 'a perfect
```

```
val fold :
 ∀('f :: ⋆ → ⋆).
   (∀'a.'a → 'a 'f) →
   (∀'a.('a × 'a) 'f → 'a 'f) →
   ∀'b.'b perfect → 'b 'f
```

Perfect trees: folds

```
type 'a perfect =
  Zero : 'a → 'a perfect
| Succ : ('a × 'a) perfect → 'a perfect

type 'f folder = {
  z: 'a. 'a → ('a, 'f) app;
  s: 'a. ('a × 'a, 'f) app → ('a, 'f) app;
}

let rec fold : 'f 'b. 'f folder → 'b perfect → ('b,'f) app
```

Perfect trees: folds

```
type 'a perfect =
  Zero : 'a → 'a perfect
| Succ : ('a × 'a) perfect → 'a perfect

type 'f folder = {
  z: 'a. 'a → ('a, 'f) app;
  s: 'a. ('a × 'a, 'f) app → ('a, 'f) app;
}

let rec fold : 'f 'b. 'f folder → 'b perfect → ('b,'f) app =
  fun { z; s } → function
  | Zero l → z l
  | Succ p → s (fold { z; s } p)
```

Perfect trees: folds

```
module Perfect = Newtype(struct type 'a t = 'a perfect end)
```

Perfect trees

```
module Perfect = Newtype(struct type 'a t = 'a perfect end)

let zero x = inj (Zero x)
let succ x = inj (Succ (prj x))
```

Perfect trees

```
module Perfect = Newtype(struct type 'a t = 'a perfect end)

let zero x = inj (Zero x)
let succ x = inj (Succ (prj x))

let idp p = prj (fold { z = zero; s = succ } p)
```

In the paper

`(the sales pitch)`

In the paper

- ★ Implementation: extensible variants + generative functors

In the paper

- ★ Implementation: extensible variants + generative functors
- ★ Implementation: an unchecked coercion

In the paper

- ★ Implementation: extensible variants + generative functors
- ★ Implementation: an unchecked coercion

But wait — there's more!

In the paper

- ★ Implementation: extensible variants + generative functors
- ★ Implementation: an unchecked coercion

But wait — there's more!

- ★ Example: Leibniz equality!

In the paper

- ★ Implementation: extensible variants + generative functors
- ★ Implementation: an unchecked coercion

But wait — there's more!

- ★ Example: Leibniz equality!
- ★ Example: the codensity transform!

In the paper

★ Implementation: extensible variants + generative functors

★ Implementation: an unchecked coercion

But wait — there's more!

★ Example: Leibniz equality!

★ Example: the codensity transform!

★ Example: kind polymorphism[1]!

---

[1] Conditions apply