λ Kansai in Summer 2026

HKT のない言語で Monad をどう表現するか

Standard ML の場合

芳賀 雅樹 / @silasolla

自己紹介

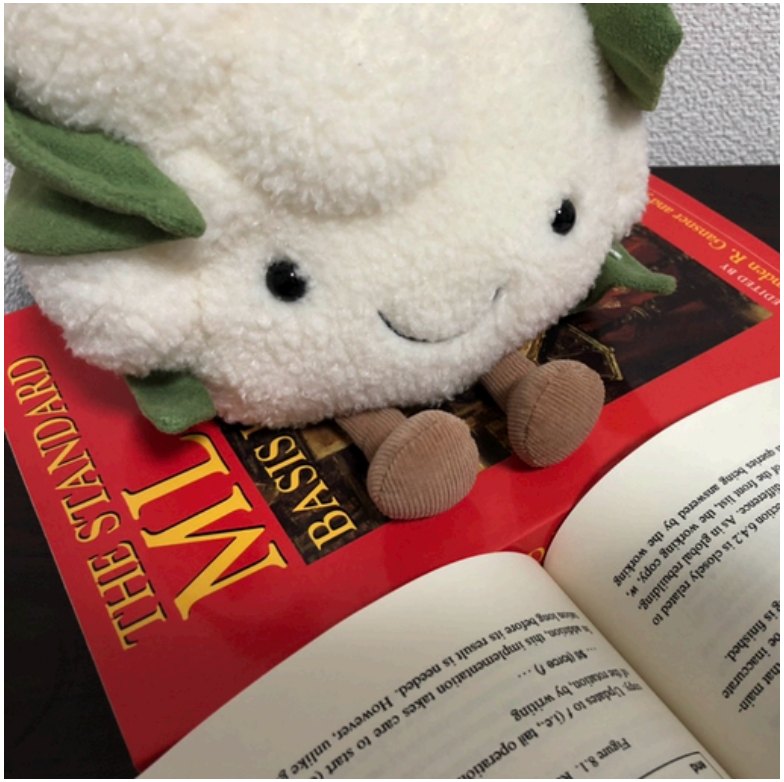
芳賀 雅樹 / @silasolla

Standard ML がすきです！

- 当然 Machine Learning ではなく Meta Language……！

3-shake Inc. で働いています！

- アプリ開発のモダナイゼーション支援をやっている
- トイルを無くすのが生業



インターネット近影

Haskell の Monad ってなんだっけ

Monad とはなんですか？

モナドとは単なる **自己関手の圏** における
モノイド対象 のことだよ

何か問題でも？

……という話はしない！！

Monad の嬉しいところ

do 記法は Maybe でも List でも IO でも

同じように書けば 動いて便利！

Monad のクラス定義

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

これらをインターフェースとして，各 Monad は `return` と `>>=` を実装

Monad laws (左右の単位元・結合則) を満たすように……！

Maybe Monad の実装と利用

```haskell
instance Monad Maybe where
  return x = Just x
  Nothing  >>= _ = Nothing
  (Just x) >>= f = f x

safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv a b = Just (a `div` b)

example :: Maybe Int
example = do          -- 文脈から型推論で Monad が決まる
  x <- safeDiv 100 5
  y <- safeDiv x 4
  return (x + y)      -- Just 25
```

List Monad の実装と利用

```haskell
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)

exampleList :: [Int]
exampleList = do      -- 文脈から型推論で Monad が決まる
  x <- [1, 2, 3]
  y <- [4, 5]
  return (x + y)      -- [5, 6, 6, 7, 7, 8]
```

Monad を支えているもの

型構築子の抽象化	`m a` って書くときの `m` の部分
型構築子の部分適用	`Either e` みたいに **カリー化** されてる
型に応じた実装の暗黙選択	どの `bind` や `return` を呼ぶのか
糖衣構文	`do` や `<-` を脱糖して `>>=` の連鎖に

Standard ML でどこまでやれるか見ていく……

モジュールシステムによる抽象化

Standard ML のモジュールシステム

`signature`	インターフェース (型と関数の公開する仕様)
`structure`	具体的な実装 (**`signature`** が **`structure`** の型になる)
`functor`	**`structure`** を受け取って **`structure`** を返す関数

注意：ML の `functor` は Haskell の `Functor` とは **別物** ！ (圏論のことは忘れて)

signature で MONAD を素朴に定義

```
signature MONAD =
sig
  (* 提供する型 *)
  type 'a t
  val pure : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
end
```

type 'a t が Haskell の **m a** に対応 (ML は型変数が前置)

Option (Maybe) Monad の実装

```
structure OptionMonad : MONAD =
struct
  type 'a t = 'a option
  fun pure x = SOME x
  fun bind NONE     _ = NONE
    | bind (SOME x) f = f x
end
```

- **MONAD** という **signature** を持つ **structure** として実装
  - MONAD に記述された情報しか公開しない
- **type 'a t** に **'a option** を割り当て
  - Standard ML の **: SIG** は透明なので **t** は外部の **option** と互換性あり
  - **:> SIG** (不透明) を使わないなら **where type** (OCaml の **with type**) 不要

Option Monad を使ってみる

```
open OptionMonad
infix 1 >>=
fun m >>= f = bind m f

val a = SOME 1 >>= (fn x =>
        SOME 2 >>= (fn y =>
        pure (x + y)))          (* SOME 3 *)

val b = a    >>= (fn x =>
        NONE >>= (fn y =>
        pure (x + y)))          (* NONE *)
```

List Monad の実装

```
structure ListMonad : MONAD =
struct
  type 'a t = 'a list
  fun pure x = [x]
  fun bind []      _ = []
    | bind (x::xs) f = f x @ bind xs f (* flatMap *)
end
```

- **MONAD** という signature を持つ structure として実装
  - MONAD に記述された情報しか公開しない
- **type 'a t** に **'a list** を割り当て
  - Standard ML の **: SIG** は透明なので **t** は外部の **list** と互換性あり
  - **:> SIG** (不透明) を使わないなら **where type** (OCaml の **with type**) 不要

List Monad を使ってみる

```
open ListMonad
infix 1 >>=
fun m >>= f = bind m f

val a = [1, 2, 3] >>= (fn x =>
        [4, 5]    >>= (fn y =>
        pure (x + y)))          (* [5,6,6,7,7,8] *)

val b = [6, 7, 8] >>= (fn x =>
        []        >>= (fn y =>
        pure (x + y)))          (* [] *)
```

Either Monad も実装したい

```
(* 素朴に either を定義 *)
datatype ('e, 'a) either = Left of 'e | Right of 'a
```

型構築子を Standard ML は **カリー化できない**

- 型変数が 2 つあるので **`type 'a t`** に合わない
- 構文レベルで **`e`** を固定できない

Haskell のように **`Either e`** と **部分適用** したい……！

functor による抽象化

```
(* 引数でエラー型を受け取る *)
functor EitherMonad (Err : sig type err end) : MONAD =
struct
  type 'a t = (Err.err, 'a) either
  fun pure x = Right x
  fun bind (Left e)  _ = Left e
    | bind (Right x) f = f x
end
```

Haskell の **型レベルの部分適用** を **モジュールの適用** で代替

エラーの部分適用

```
(* シンプルにメッセージ文字列 *)
structure StringEither = EitherMonad (struct type err = string end)

(* バリアントで詳細なエラー定義 *)
datatype parseErr = UnexpectedEof
                  | Expected of string
                  | UnexpectedChar of char

structure ParseEither = EitherMonad (
                          struct type err = parseErr end)
```

エラーの設計ごとに，専用の Monad が手に入る……！

Either Monad を使ってみる

```
datatype calcErr = NotInt of string | DivByZero
structure E = EitherMonad (struct type err = calcErr end)

infix 1 >>=
fun m >>= f = E.bind m f

fun parseInt s = case Int.fromString s of
                   SOME n => Right n
                 | NONE   => Left (NotInt s)
fun safeDiv a b = if b = 0 then Left DivByZero
                           else Right (a div b)

fun pipeline s t = parseInt s >>= (fn a =>
                   parseInt t >>= (fn b =>
                   safeDiv a b))
```

Applicative スタイルをやりたい

```
fun add x y = x + y
val a = add <$> SOME 1 <*> SOME 2  (* SOME 3 *)

fun pair x y = (x, y)
val b = pair <$> [1, 2, 3] <*> [4, 5]
        (* [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)] *)
```

手元に Monad (つよい) があるので **functor** で導出できる

- Applicative や Functor を個別に作ってもよい
- Functor → Applicative → Monad の階層を重ねてもよい

Monad から Applicative の導出

```
functor Monad2Applicative (M : MONAD) = struct
  infix 4 <$> <*>
  fun f  <$> m  = M.bind m  (fn x => M.pure (f x))
  fun mf <*> mx = M.bind mf (fn f =>
                  M.bind mx (fn x => M.pure (f x)))
end

structure OptionAp = Monad2Applicative (OptionMonad)
structure ListAp   = Monad2Applicative (ListMonad)
```

Haskell の **`liftM`** や **`ap`** と同じ (個々の Monad に依存しない)

Haskell のクラス階層 (補足)

現在の Haskell は **スーパークラス制約** で連なる

```haskell
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  return = pure
```

Monad の実装は **Functor** の実装が前提

汎用関数を書いてみる

```
functor MonadUtil (M : MONAD) = struct
  (* sequence :: Monad m => [m a] -> m [a] *)
  fun sequence []       = M.pure []
    | sequence (m::ms) = M.bind m (fn x =>
                           M.bind (sequence ms)
                                  (fn xs => M.pure (x::xs)))
  (* mapM :: Monad m => (a -> m b) -> [a] -> m [b] *)
  fun mapM f xs = sequence (List.map f xs)
end

structure OptU = MonadUtil (OptionMonad)
structure LstU = MonadUtil (ListMonad)

OptU.sequence [SOME 1, SOME 2, SOME 3] (* SOME [1,2,3] *)
OptU.sequence [SOME 1, NONE,   SOME 3] (* NONE *)
LstU.sequence [[1,2],[3,4]]            (* [[1,3],[1,4],[2,3],[2,4]] *)
```

モジュールで高階の抽象化をやれた

`MONAD` の `type 'a t` で型構築子 `* -> *` を 抽象化 できた……！

`functor` のモジュール適用で 型変数を固定 できた……！

高階 `functor` (`functor` をとったり返したりできる `functor`) があればもっと表現できる

- SML/NJ 拡張や OCaml などにあります
- 部分適用した `functor` を持ち回れます
- `(* -> *) -> (* -> *)` みたいな Transformer をとるカインドもいけます

ML のモジュールで高階の抽象化はやれる

モジュールでやれた部分

型構築子の抽象化	`m a` って書くときの `m` の部分 ✓ ✓
型構築子の部分適用	`Either e` みたいに **カリー化** されてる ✓
型に応じた実装の暗黙選択	どの `bind` や `return` を呼ぶのか
糖衣構文	`do` や `<-` を脱糖して `>>=` の連鎖に

コア言語レベルのサポートは無いが大部分を実現できた……！

トレードオフについての話

暗黙ディスパッチはできない

```
ListMonad.bind xs f (* SML: どの Monad の bind かを明示 *)
let* x = xs in ...  (* OCaml: binding operator でも Monad 固定 *)
xs >>= f            -- Haskell: 値の型から推論して選択
```

SML / OCaml は **自分で指定** する (**functor** で差し込めるようにしたり)

Haskell は **型から推論** される (暗黙ディスパッチ)

do 記法のサポートはない

```
bind m1 (fn x =>
bind m2 (fn y => pure (x + y)))   (* ネスト *)

m1 >>= (fn x => m2
   >>= (fn y => pure (x + y)))    (* 中置演算で緩和 *)

do with M;
    x <- m1;          (* PreML で模倣 *)
    y <- m2;
    return (x + y)
end
```

プリプロセッサや **infix** で寄せられるが Monad の固定は必要……

複数実装が共存できる (e.g. Applicative)

```
(* 相異なる <*> : ('a -> 'b) list -> 'a list -> 'b list が共存 *)
structure CartAp : AP = struct
  infix 4 <*>
  fun fs <*> xs = List.concat (map (fn f => map f xs) fs) (* 直積 *)
end

structure ZipAp : AP = struct
  infix 4 <*>
  fun fs <*> xs = ListPair.map (fn (f, x) => f x) (fs, xs) (* zip *)
end
```

- **SML**：**`functor`** を明示適用 (スコープ分け) ⇄ 複数実装が **`structure`** として共存
- **Haskell**：自動で辞書渡し ⇄ instance の一意制約 (**`newtype`** + **`coerce`** で使い分け)

型クラスとモジュールの関係

	Haskell の型クラス	SML のモジュール
インターフェース	`class Monad m`	`signature MONAD`
型の実装	`instance Monad Maybe`	`structure ... : MONAD`
選ばれかた	型から **暗黙** ／ **一意** に決定	名前で **明示** ／ **複数** 共存

型クラスはモジュールを **型駆動の暗黙ディスパッチ** に特化させた形

cf. Modular Type Classes (Dreyer et al., POPL 2007)

まとめ

`signature` と **`structure`** と **`functor`** があれば
Monad や Type Class 相当の抽象も **組み立てられる！**

`MultiParamTypeClasses`, `FunctionalDependencies`,
`TypeFamilies`, `FlexibleInstances`, …… とか考えなくていい

Standard ML はいいぞ……！

関数型まつり 2026 でも喋ります (宣伝)

2026 年に読む “The Definition of Standard ML”

Standard ML の言語仕様を見ながら設計のヒントを探る試み……

関数型まつり 2026 ／ 7/12 11:30 〜 Track A

発表の詳細は fortee.jp を見てください……！

参考文献

1. κeen「SML でモナド」https://keens.github.io/blog/2016/10/10/smldemonado/
2. Nagashima & O'Connor, Close Encounters of the Higher Kind: Emulating Constructor Classes in Standard ML, ML Workshop 2016 (arXiv:1608.03350)
3. Shan, Higher-order modules in System F_ω and Haskell, Web publication https://homes.luddy.indiana.edu/ccshan/xlate/xlate.pdf
4. Dreyer, Harper, Chakravarty, Keller, Modular Type Classes, POPL 2007
5. Yallop & White, Lightweight Higher-Kinded Polymorphism, FLOPS 2014 (OCaml での defunctionalize で HKT 符号化) https://github.com/yallop/higher