

関数型プログラマのためのモナド理論 (1)

浜名誠

CAPS, IPL, 2007, November 6th

動機

- 「モナド」で Google すると、沢山の Haskell プログラムの苦勞が見える
- よく見る格言: Haskell のモナドを理解するのに圏論のモナドを理解する必要はない
- 「分かりやすい説明」というものが、本当に分かりやすい?
- モナドの理解を実行から追う、というのは話が逆
- 主張: 数学の方が簡単
- そもそもなぜモナドを使うとよいのか。プログラム意味論の歴史に理由がある
- それらを解説
- また、モナドから直ちに出てくる圏論的構成は有用
- それらを通して、計算機科学でなぜ圏論が有用なのかの一例

▶ feedback をお願いします。

全体の内容の予定

1. 圏論のモナドのアイデアと関連するいくつかの construction
2. 計算モナド: Computational λ -calculus と Computational Metalanguage
 - Moggi の 重要な仕事。
 - プログラム意味論を partial continuous function で再構成する pCpo のためのメタ言語からの影響
 - Haskell の do-notation 近いメタ言語
3. 伝統的: モナドと代数 (項代数, initial algebra, datatype への適用)
4. 計算モナド as 自由代数 [Power-Plotkin'01-]
5. (できれば) Haskell のモナド
IO モナドはモナドではない。state モナドと Haskell と η ルール

今日の内容

- モナドの定義は何を意味しているのか
- モナドからすぐ出てくる構造を3つ
- 副作用のモナドの話は今日はしない

準備

- 仮定する知識: constructive algorithmics で使う圏論ぐらい。
- 圏 \mathcal{C} いくつかの記法
対象 A から B への射の集合を

$$\mathcal{C}(A, B)$$

と書き、hom 集合 (hom-set) と呼ぶ。

- 対象を取り出す $A \in \mathcal{C}$ (または $A \in \text{ob } \mathcal{C}$)
- 射を取り出す $f \in \text{arr } \mathcal{C}$
- 必要な基本事項: opposite category \mathcal{C}^{op} , product, coproduct, isomorphism, functor, natural transformation, initial object, terminal object (final object)
- どんな圏論の本にも書いてある。
- 圏の例: 次のスライド

Category	Set	Cpo	Cppo	Cppo _⊥
object	set	cpo	cppo	cppo
bottom?	no	no	⊥	⊥
functions	all	cont.	cont.	strict
$D \times E$	+	+	+	+
$[D \rightarrow E]$	+	+	+	✓
$D + E$	+	+	no	
Initial object	∅	∅	no	{⊥}
Terminal object	{*}	{*}	{⊥}	{⊥}
$D \otimes E$			✓	✓
$[D \rightarrow_{\perp} E]$			✓	+
$D \oplus E$			✓	+
closed structure	ccc	ccc	ccc	monoidal closed
i.alg.=final coalg.				✓

モナド

Def. 圏 \mathcal{C} 上のモナドとは

- 関手 $T : \mathcal{C} \rightarrow \mathcal{C}$,
- 自然変換 $\eta : \text{Id} \rightarrow T$, $\mu : T^2 \rightarrow T$

から成る三つ組 (T, η, μ) で任意の $A \in \mathcal{C}$ に対して次の図式を可換にするもの:

$$\begin{array}{ccc} T^3 A & \xrightarrow{\mu_{TA}} & T^2 A \\ \downarrow T\mu_A & & \downarrow \mu_A \\ T^2 A & \xrightarrow{\mu_A} & TA \end{array} \qquad \begin{array}{ccc} TA & \xrightarrow{\eta_{TA}} & T^2 A \xleftarrow{T\eta_A} TA \\ & \searrow & \downarrow \mu_A \nearrow \\ & = & TA = \end{array}$$

Notation 関手の合成 $T \circ T$ を T^2 と書く。 T^3 などと同様。関手の射部分 $T(f)$ を Tf と書く。

▶ モナド as モノイド

モナド

Ex. Set 上のモナド

1. (a) Kleene 閉包 A^* (b) リスト $\text{List } A$
2. べき集合 $\mathcal{P}A$
3. リフティング $A + 1$ (Maybe A)
4. 項集合 $T_\Sigma(X) \dots X$ 変数の集合

乗法 μ は「括弧を壊す」操作、単位 ν は「括弧に入れる」操作である。クリーネ閉包 $\langle \dots \rangle$ 、リスト $[\dots]$ 、べき集合 $\{\{-\}, \dots, \{-\}\}$ はいずれも「括弧」、あるいは「殻」がある。モナドとはそのような「殻のある構造」を一般化したものと考えるとわかりやすい。

殻のある構造をつくるには必ず殻に入れる操作が必要。それが単位。殻が二重になっていたら壊す操作もすぐに思いつく。その場合壊すやり方として、(1) 外側の括弧から壊すか、(2) 内側の括弧から壊すか、二通り考えられる。その二通りのやり方が結合則の可換図の二通りの道筋である。

モナド

Ex. Cppo_\perp 上のモナド

1. リフティング $(-)_\perp : \text{Cppo}_\perp \rightarrow \text{Cppo}_\perp$

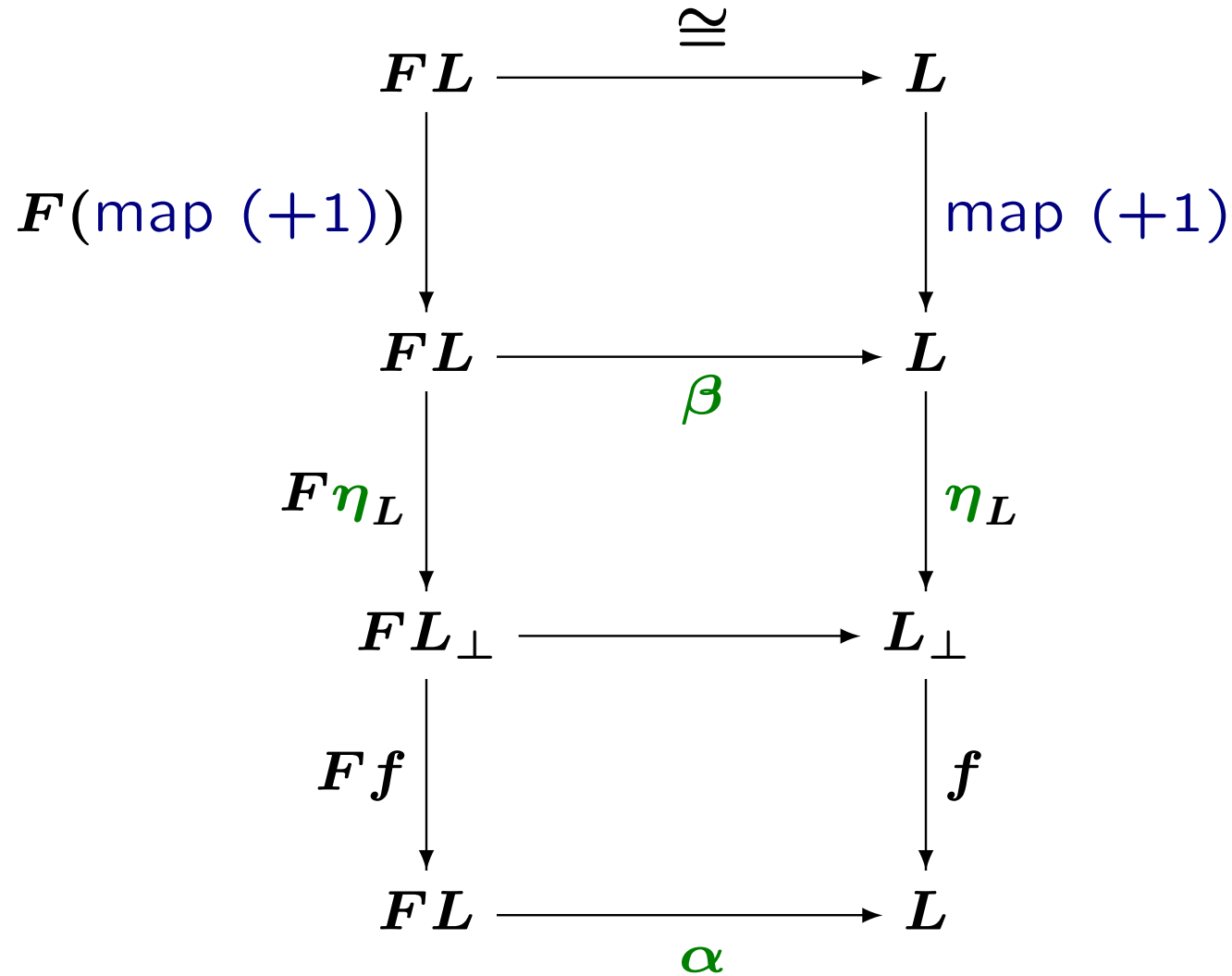
$$D_\perp = D + \{\perp\}$$

$$\eta_D : D \rightarrow_\perp D_\perp$$

$$\mu_D : (D_\perp)_\perp \rightarrow_\perp D_\perp$$

応用 酒井さんの non-strict function との fusion を catamorphism \mathbf{Cppo}_\perp で考えてみる。

Haskell の リスト関手 $FX = 1 \oplus \mathbb{N} \otimes X_\perp$



Kleisli トリプル

Def. 圏 \mathcal{C} 上の Kleisli トリプル とは

- 関数 $T : \text{ob } \mathcal{C} \rightarrow \text{ob } \mathcal{C}$
- 射 $\eta_A : A \rightarrow TA$ (各 $A \in \text{ob } \mathcal{C}$ について)
- 射 $f^* : TA \rightarrow TB$ を 各 $f : A \rightarrow TB \in \text{arr } \mathcal{C}$ についてつくる演算 $(-)^*$ (Kleisli リフティング)

からなる三つ組 $(T, \eta, (-)^*)$ で以下の等式を満たすもの:

$$\eta_A^* = \text{id}_{TA}$$

$$f^* \circ \eta_A = f \quad \text{for } f : A \rightarrow TB$$

$$g^* \circ f^* = (g^* \circ f)^* \quad \text{for } f : A \rightarrow TB, g : B \rightarrow TC$$

Kleisli トリプル in Haskell

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

- 型構成子 T が 関数 T
- `return` が unit η
- `flip (>>=)` :: $(a \rightarrow t b) \rightarrow (t a \rightarrow t b)$ が $(-)^*$

Haskell ではモナドではなく、なぜ Kleisli トリプルの定義を使うのか

- データとして圏論的データではなくて、単に関数を与えればよい:
 T の射の部分、 η の自然性 を示す必要がない。
- μ より、Kleisli リフティングの方が「inductive」に与えられる。
- 定義、意味はほぼ同じ: $(-)^*$ “substitution” or “concatination”
 μ “destruction”

Kleisli トリプル: 例

Ex. 項集合のモナド T_{Σ} と代入操作

Haskell Prelude

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

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

別の定義

```
instance Monad [] where
  [] >>= []
  (x:xs) >>= f = f x ++ (xs >>= f)
  return     = [x]
```

モナドに関連する構造

1. モナド T からすぐにつくれる構造

- Kleisli 圏 \mathcal{C}_T
- Eilenberg-Moore 代数 $TA \rightarrow A$

2. モナドをつくるための構造

- Adjoint

これら二つのものがさらに adjoint で関連している。

計算機科学的な理解もできる。

Kleisli 圏

Def. $T = (T, \eta, \mu)$ を \mathcal{C} 上のモナドとする。Kleisli 圏 \mathcal{C}_T は

- Objects: \mathcal{C} と同じ
- Arrows: $f : A \rightarrow B$, if $f : A \rightarrow TB \in \text{arr } \mathcal{C}$, i.e.

$$\mathcal{C}_T(A, B) = \mathcal{C}(A, TB)$$

- Composition:

$$A \xrightarrow{f} B \xrightarrow{g} C \quad \text{in } \mathcal{C}_T$$

を

$$A \xrightarrow{f} TB \xrightarrow{Tg} TTC \xrightarrow{\mu_C} TC \quad \text{in } \mathcal{C}$$

で与える。

- Identity: $\text{id}_A : A \rightarrow A \in \text{arr } \mathcal{C}_T$ は $\eta_A : A \rightarrow TA \in \text{arr } \mathcal{C}$ で。

使い方

- 計算型 (computational type) を返す関数のモデル化として計算モナド理論では重要な働きをする。
- アイデア: $\text{write} : \text{Int} \rightarrow \text{IO } 1$ いつも結果が計算型である世界 (手続き型言語) ならば、結果の計算型は省略しよう
 - Kleisli 圏 $\mathcal{C}_{\mathcal{T}}$ の世界
= Computational λ -calculus
 - Kleisli arrow と普通の arrow が混在する世界 \mathcal{C}
= Computational Metalanguage = 手続き型言語

Remark Haskell 98 Report にも Kleisli 圏という言葉は使われている

Adjunction

Def. \mathcal{C} から \mathcal{D} への adjoint とは以下を満たす三組 $(F, U, \widehat{(-)})$ をいう。
 F と U は functor のペア

$$\begin{array}{ccc} & F & \\ \mathcal{C} & \xrightarrow{\quad} & \mathcal{D} \\ & \perp & \\ & U & \\ & \xleftarrow{\quad} & \end{array}$$

で同型写像

$$\widehat{(-)} : \mathcal{D}(FX, Y) \xrightarrow{\cong} \mathcal{C}(X, UY)$$

が X と Y について自然。

意味 上は射の対応を言っている。さらに対応の仕方が $\widehat{(-)}$ で定まっていると。

$$\mathcal{D}(FA, B) \cong \mathcal{C}(A, UB) \quad \frac{f : FA \longrightarrow B}{\widehat{f} : A \longrightarrow UB}$$

三角の図式 ▶ ホワイトボード

Ex. 1. 項と代入 2. カリー化と CCC

Adjoint のよくある表現 adjunction にはいくつかの同等な定義の仕方がある。

- $(F, U, \eta, (\widehat{-})) \dots \eta$ は unit
- $(F, U, \epsilon, (\check{-})) \dots \epsilon$ は counit

$$\frac{\text{id}_{FA} : FA \longrightarrow FA}{\eta_A : A \longrightarrow UFA} \qquad \frac{\epsilon_B : FUB \longrightarrow B}{\text{id}_{UB} : UB \longrightarrow UB}$$

- F を left adjoint
- U を right adjoint (典型的には forgetful functor)

と呼び、先の図式のように書いたり $F \dashv U$ と書いたりする。

有用な定理 F は colimit を保存し、 U は limit を保存する。

- $F(A + B) \cong FA + FB$ TRS の modularity への応用
- initial algebra construction では type functor が ω -colimit を保存しなければならない。それを示すのに type constructor が left adjoint であることを言えば OK。

例: product type constructor \times は left adjoint

Adjoint と Monad

Thm.

$(F, U, \eta, \widehat{(-)})$ が adjoint のとき、 $(UF, \eta, U(\check{\text{id}}_{UF(-)}))$ はモナド。

$$\begin{array}{ccc} & F & \\ \mathcal{C} & \xrightarrow{\quad} & \mathcal{D} \\ & \perp & \\ & U & \\ & \downarrow & \end{array}$$

UF が モナド

Remark $\mu_A = U(\check{\text{id}}_{UFA}) = U(\epsilon_{FA})$

Adjoint と Monad

$T : \mathcal{C} \longrightarrow \mathcal{C}$ を モナド とする。

Kleisli 圏 と の 関 係

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{F_T} & \mathcal{C}_T \\ & \perp & \\ & \xleftarrow{U_T} & \end{array}$$

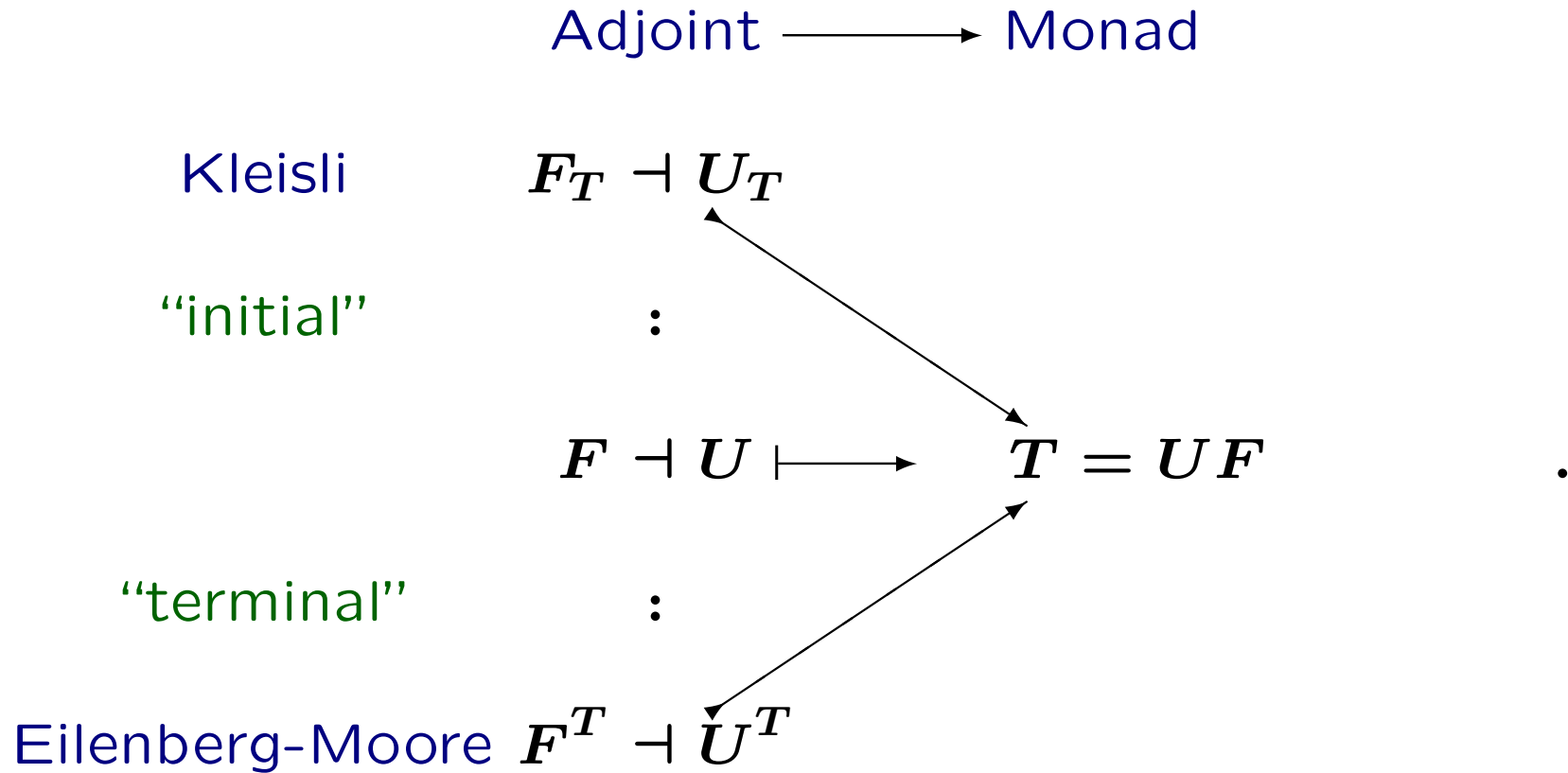
Eilenberg-Moore 圏 と の 関 係

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{F^T} & \mathcal{C}^T \\ & \perp & \\ & \xleftarrow{U^T} & \end{array}$$

Adjoint と Monad

すべてのモナドは adjoint からつくれる。

しかし、モナド T を adjoint に分解するやり方は複数ある。



計算機科学的には?

- 実際例に当てはめて考える
- 意味があるものが出ると、次々に関連する概念が得られるのが利点。
- 例: 自由 Σ モノイド

$$\text{Set}^{\mathbb{F}} \begin{array}{c} \xrightarrow{M_{\Sigma}} \\ \perp \\ \xleftarrow{U} \end{array} \Sigma\text{-Mon}$$

Eilenberg-Moore 代数

Def. $T = (T, \eta, \mu)$ を \mathcal{C} 上のモナドとする。Eilenberg-Moore 代数とは組 $(X, \theta : TX \rightarrow X)$ で以下の図式を可換にするもの:

$$\begin{array}{ccc} T^2 X & \xrightarrow{T\theta} & TX \\ \downarrow \mu_X & & \downarrow \theta \\ TX & \xrightarrow{\theta} & X \end{array} \qquad \begin{array}{ccc} X & \xrightarrow{\eta_X} & TX \\ & \searrow = & \downarrow \theta \\ & & X \end{array}$$

Eilenberg-Moore 代数 の準同型写像は普通のもの。

Remark Eilenberg-Moore 代数構造 θ とモナド構造 (η, μ) のインタラクションを言っている。モナド T の代数とも言う。

注意: initial algebra semantics で使う functor F の代数 (公理はなし) とは違うもの

Eilenberg-Moore 代数

Def. Eilenberg-Moore 圏 \mathcal{C}^T

- Objects: Eilenberg-Moore 代数 $TX \rightarrow X$
- Arrows: 準同型

▶ Eilenberg-Moore algebra とは何？

- 計算機科学で具体的な言葉で言うのは、よく分からないことも多い
- 特徴付けに使われる $\Sigma\text{-alg} \cong \mathbf{Set}^{T\Sigma}$
- 少なくとも どんな T についても、 $(TX, \mu_X : T(TX) \rightarrow TX)$ は EM-algebra。自由代数
- [Filinski-Stovring ICFP'07] (モナドがあるデータ型の推論/foldの話) で使われている。
- 環境？

同型 (Isomorphism)

Def. 圏 \mathcal{C} で射のペア

$$A \begin{array}{c} \xrightarrow{f} \\ \xleftarrow{g} \end{array} B$$

に対して $g \circ f = \text{id}_A$, $f \circ g = \text{id}_B$

のとき A と B は同型といい

$$A \cong B$$

と書く。また $f : A \xrightarrow{\cong} B$ のようにも書くことがある。

Remark

- Set の場合は f が全単射なら OK。i.e. 集合の数が同じ。
- Cpo の場合は全単射だけでなく、 f と g が continuous function。i.e. 順序を保って同じにする。
- hom 集合の場合は isomorphism は単に集合の isomorphism