

Freer Monads, More Extensible Effects

Oleg Kiselyov

Tohoku University

Hiromi ISHII

University of Tsukuba

PPL 2016

岡山県玉野市 2016/03/08

(Originally Published in Haskell Symposium '15)

Table of Contents

- Background: Monad Transformer
- Extensible Effects
- More Extensible Effects
- Examples
- Benchmarks
- Conclusions

Background

Background

モナド：関数型言語で命令的に副作用を記述する方法の一つ

Background

モナド：関数型言語で命令的に副作用を記述する方法の一つ

- 問題：モナド（=副作用）の**合成**

Background

モナド：関数型言語で命令的に副作用を記述する方法の一つ

- 問題：モナド（=副作用）の**合成**
 - ・ 出来合いの物を合成し新たなモナドを創りたい

Background

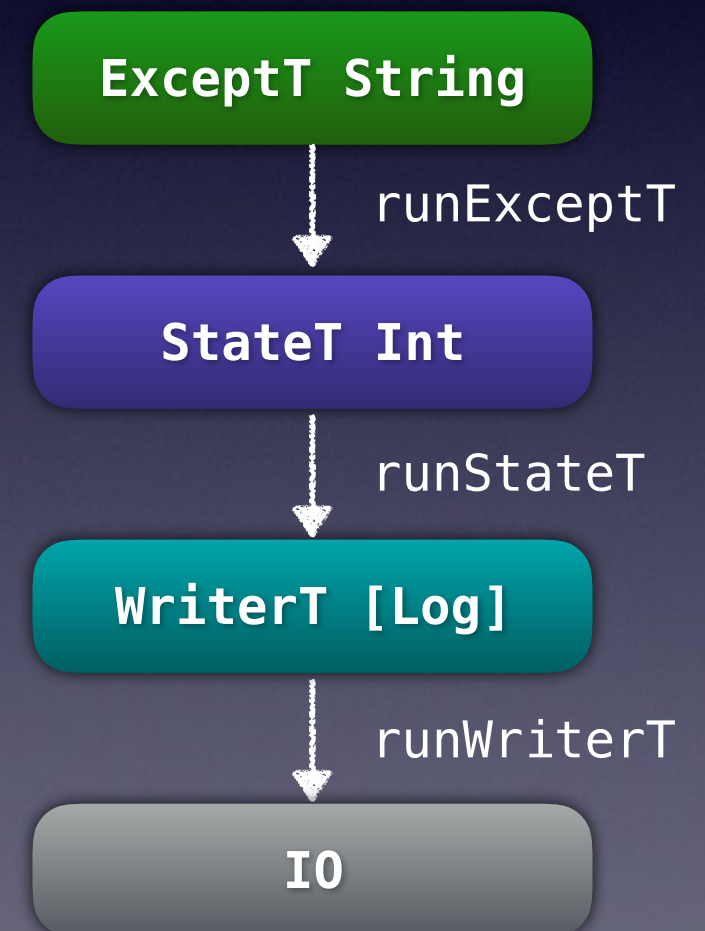
モナド：関数型言語で命令的に副作用を記述する方法の一つ

- 問題：モナド（=副作用）の**合成**
 - ・ 出来合いの物を合成し新たなモナドを創りたい
 - ・ 主流：**モナド変換子** (MTL)

Background

モナド：関数型言語で命令的に副作用を記述する方法の一つ

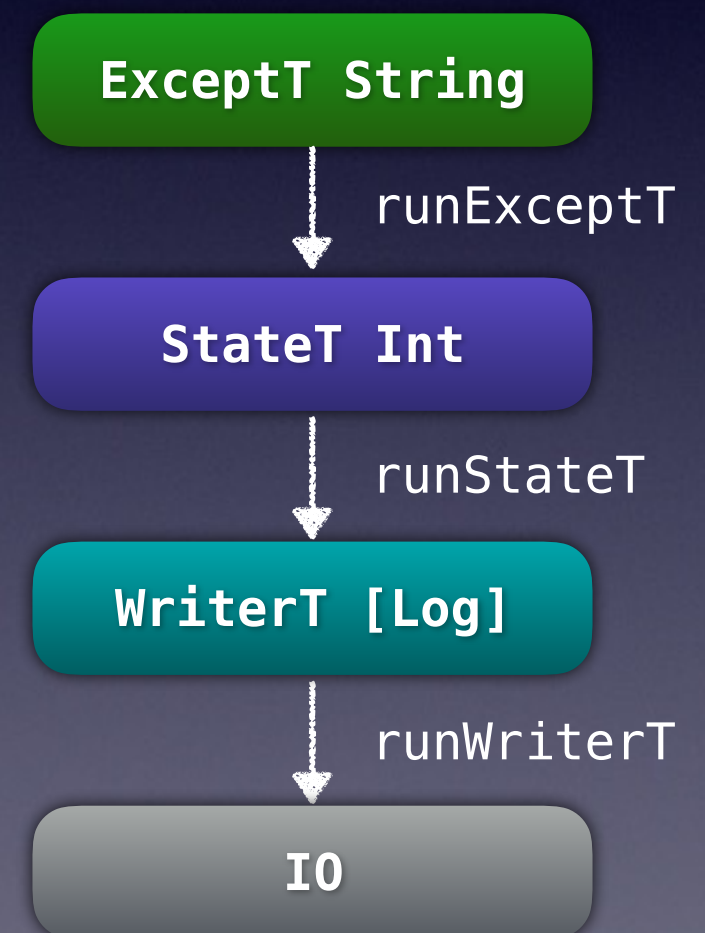
- 問題：モナド（=副作用）の**合成**
 - 出来合いの物を合成し新たなモナドを創りたい
 - 主流：**モナド変換子** (MTL)
 - 各副作用に特化した層を重ねる



Background

モナド：関数型言語で命令的に副作用を記述する方法の一つ

- 問題：モナド（=副作用）の**合成**
 - ・ 出来合いの物を合成し新たなモナドを創りたい
 - ・ 主流：**モナド変換子** (MTL)
 - ・ 各副作用に特化した層を重ねる
 - ・ ハンドラが担当する副作用を処理して下位層の効果に変換される

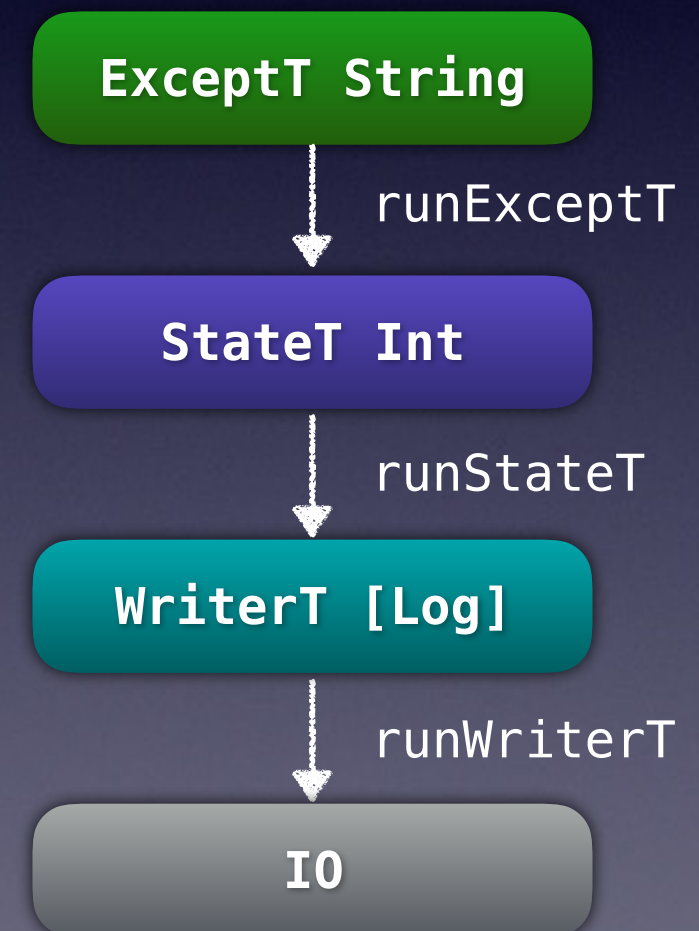


Background

モナド：関数型言語で命令的に副作用を記述する方法の一つ

- 問題：モナド（=副作用）の**合成**

- ・ 出来合いの物を合成し新たなモナドを創りたい
- ・ 主流：**モナド変換子** (MTL)
 - ・ 各副作用に特化した層を重ねる
 - ・ ハンドラが担当する副作用を処理して下位層の効果に変換される
 - ・ 下位層の副作用は **lift** か型クラスを用いて呼び出す



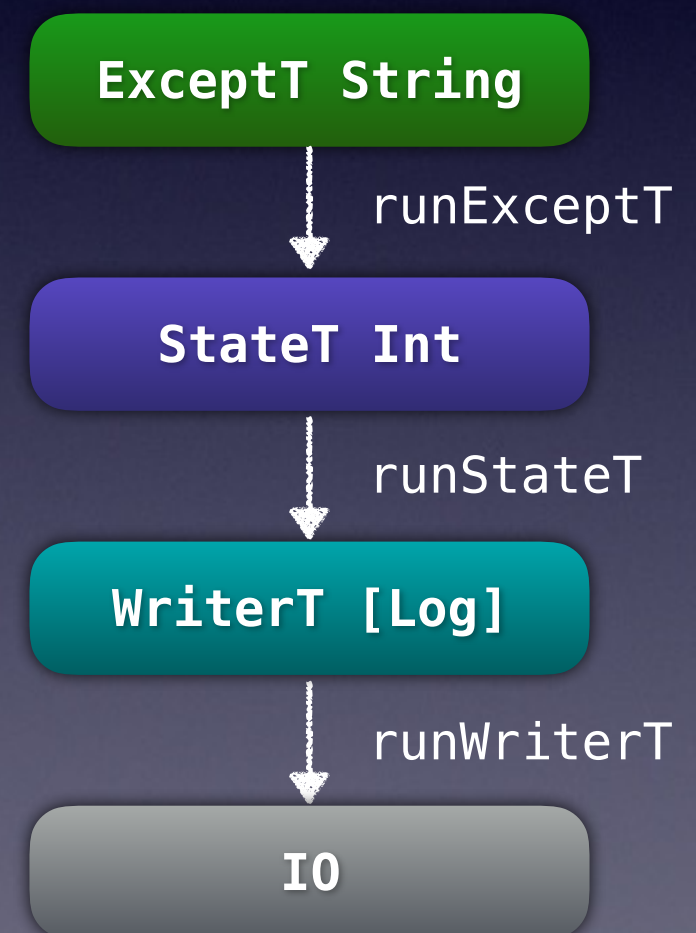
Background

モナド：関数型言語で命令的に副作用を記述する方法の一つ

- 問題：モナド（=副作用）の**合成**

- ・ 出来合いの物を合成し新たなモナドを創りたい
- ・ 主流：**モナド変換子** (MTL)
 - ・ 各副作用に特化した層を重ねる
 - ・ ハンドラが担当する副作用を処理して下位層の効果に変換される
 - ・ 下位層の副作用は **lift** か型クラスを用いて呼び出す

★ **(More) Extensible Effects** はその代替



モナド変換子の問題点

1. 意味論の固定

- ・ 一つのモナド層に一つの解釈

2. Too many lifts!!!

- ・ 合成が深くなると lift が増える
- ・ 型クラスを使うと同種の副作用の使用に制限

3. 合成順の固定

- ・ 実行時に合成順が確定され、階層を跨いだ処理が不可能

4. 合成のコスト

→ Extensible Effects [KSS2013] はこれらを解決

モナド変換子の問題点

1. 意味論の固定

- ・ 一つのモナド層に一つの解釈

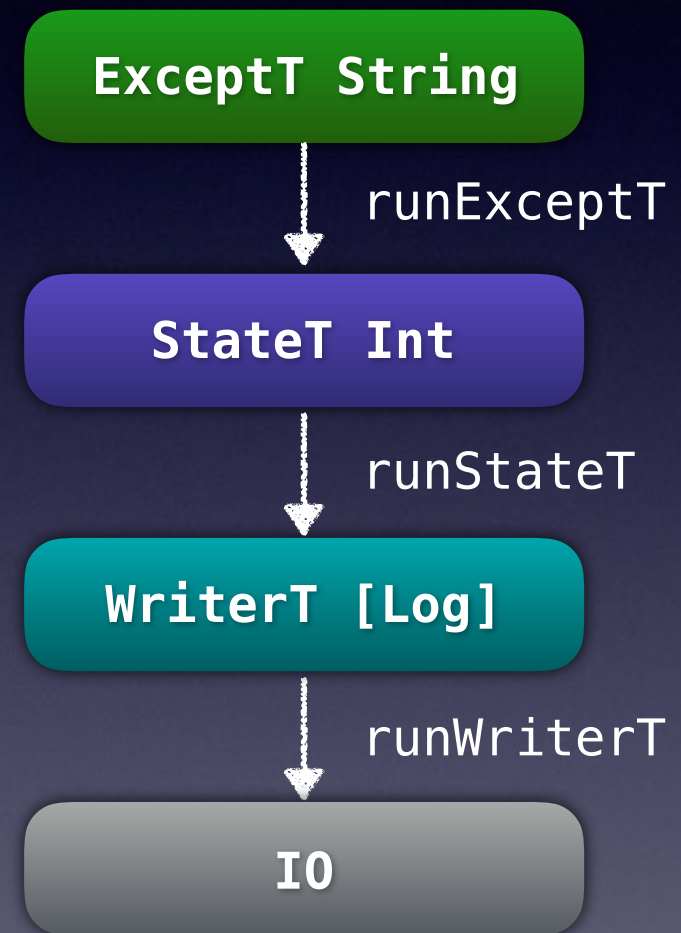
2. Too many lifts!!!

- ・ 合成が深くなると lift が増える
- ・ 型クラスを使うと同種の副作用の使用に制限

3. 合成順の固定

- ・ 実行時に合成順が確定され、階層を跨いだ処理が不可能

4. 合成のコスト



→ Extensible Effects [KSS2013] はこれらを解決

Extensible Effects

Extensible Effects

Extensible Effects

- ・ アイデア： **函手 (Functor)** の合成は簡単

Extensible Effects

- ・ アイデア： **函手 (Functor)** の合成は簡単
- ・ モナド = $\text{join} :: m (m a) \rightarrow m a$ と `return` を持つ Functor

Extensible Effects

- ・ アイデア： **函手 (Functor)** の合成は簡単
- ・ モナド = $\text{join} :: m (m a) \rightarrow m a$ と `return` を持つ Functor
- ・ 副作用を **函手の直和** で表し、その生成する **自由モナド** を考える

Extensible Effects

- ・ アイデア： **函手 (Functor)** の合成は簡単
 - ・ モナド = $\text{join} :: m (m a) \rightarrow m a$ と `return` を持つ Functor
 - ・ 副作用を **函手の直和** で表し、その生成する **自由モナド** を考える
- ・ 副作用の処理を **Client-Server モデル** で捉える

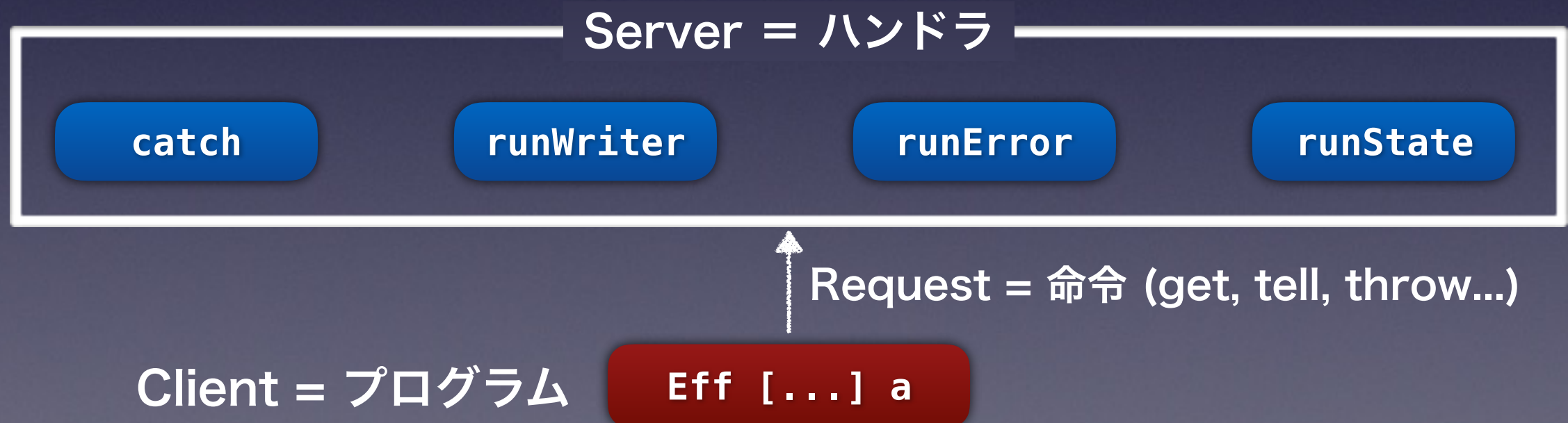
Extensible Effects

- アイデア： **関手 (Functor)** の合成は簡単
- モナド = $\text{join} :: m (m a) \rightarrow m a$ と `return` を持つ Functor
- 副作用を **関手の直和** で表し、その生成する **自由モナド** を考える
- 副作用の処理を **Client-Server モデル** で捉える



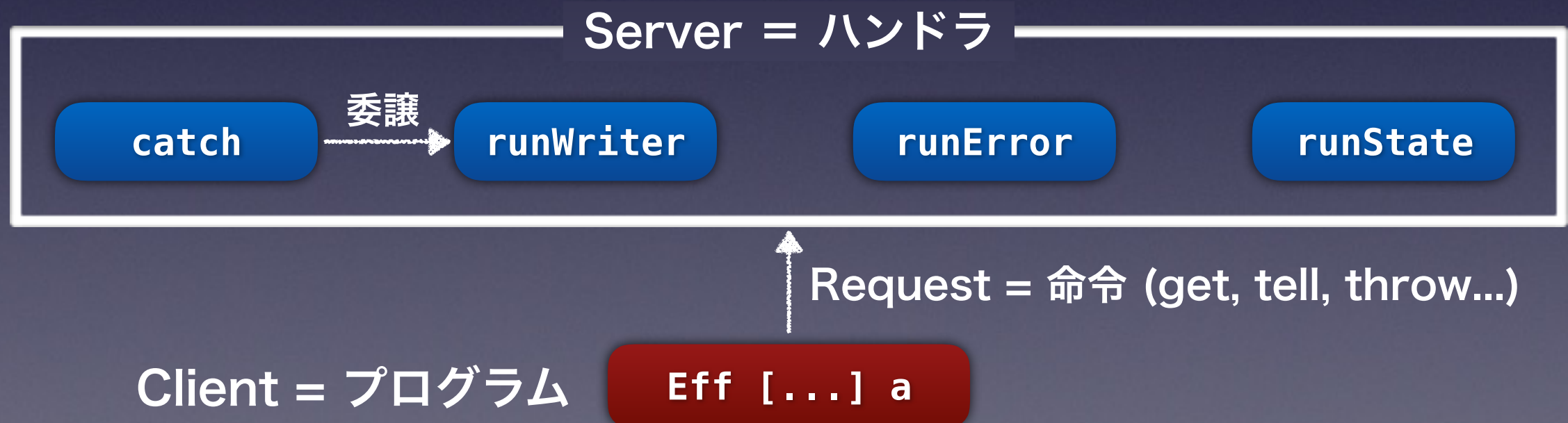
Extensible Effects

- アイデア： **函手 (Functor)** の合成は簡単
- モナド = $\text{join} :: m (m a) \rightarrow m a$ と `return` を持つ Functor
- 副作用を **函手の直和** で表し、その生成する **自由モナド** を考える
- 副作用の処理を **Client-Server モデル** で捉える



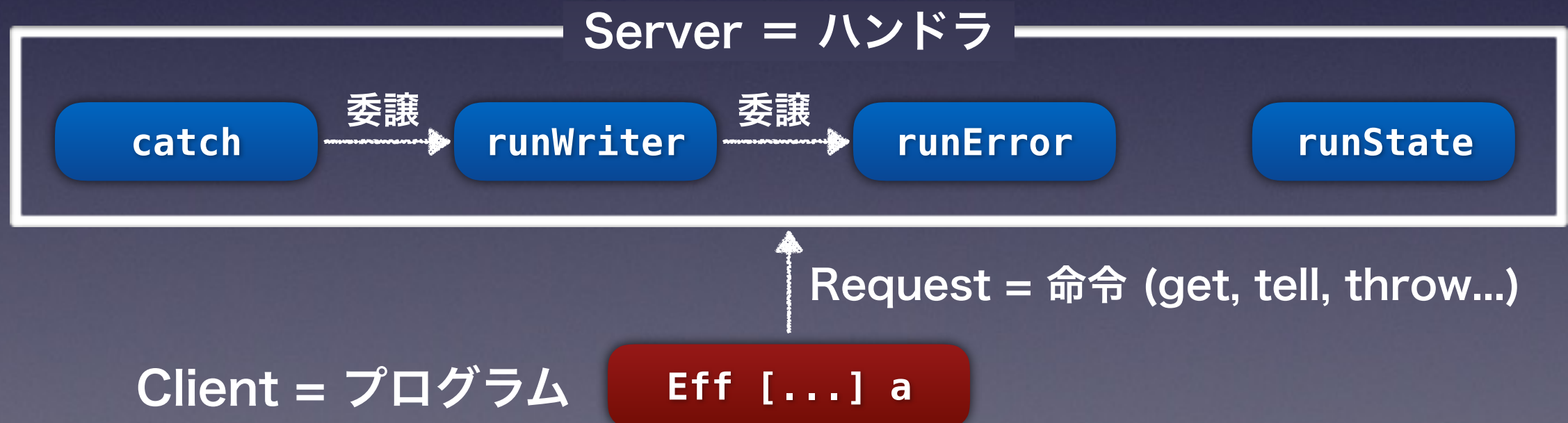
Extensible Effects

- アイデア： **函手 (Functor)** の合成は簡単
- モナド = $\text{join} :: m (m a) \rightarrow m a$ と `return` を持つ Functor
- 副作用を **函手の直和** で表し、その生成する **自由モナド** を考える
- 副作用の処理を **Client-Server モデル** で捉える



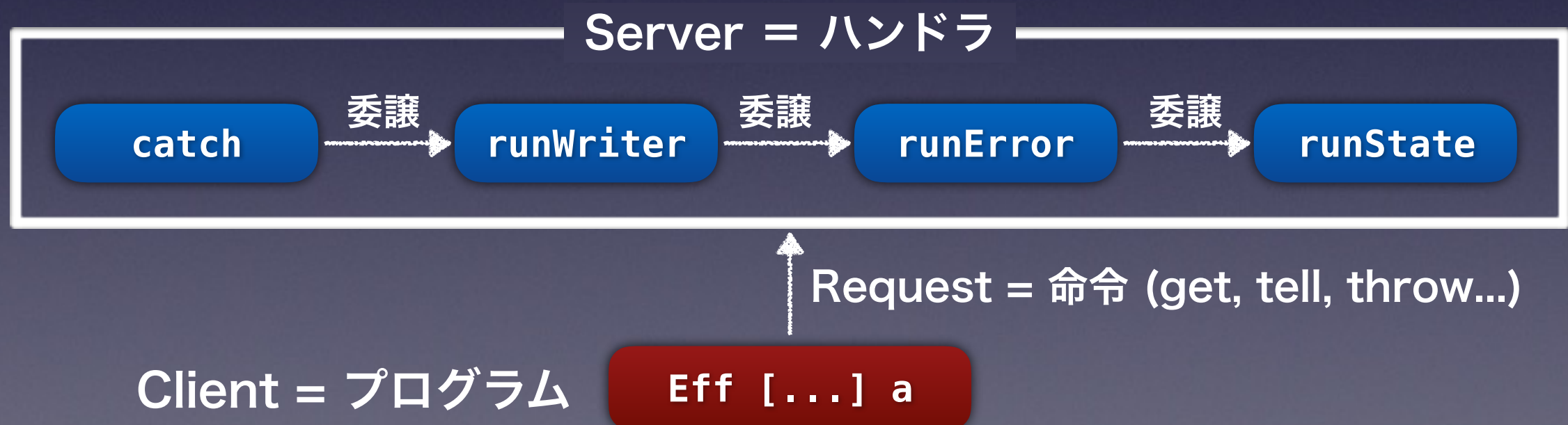
Extensible Effects

- アイデア： **函手 (Functor)** の合成は簡単
- モナド = $\text{join} :: m (m a) \rightarrow m a$ と `return` を持つ Functor
- 副作用を **函手の直和** で表し、その生成する **自由モナド** を考える
- 副作用の処理を **Client-Server モデル** で捉える



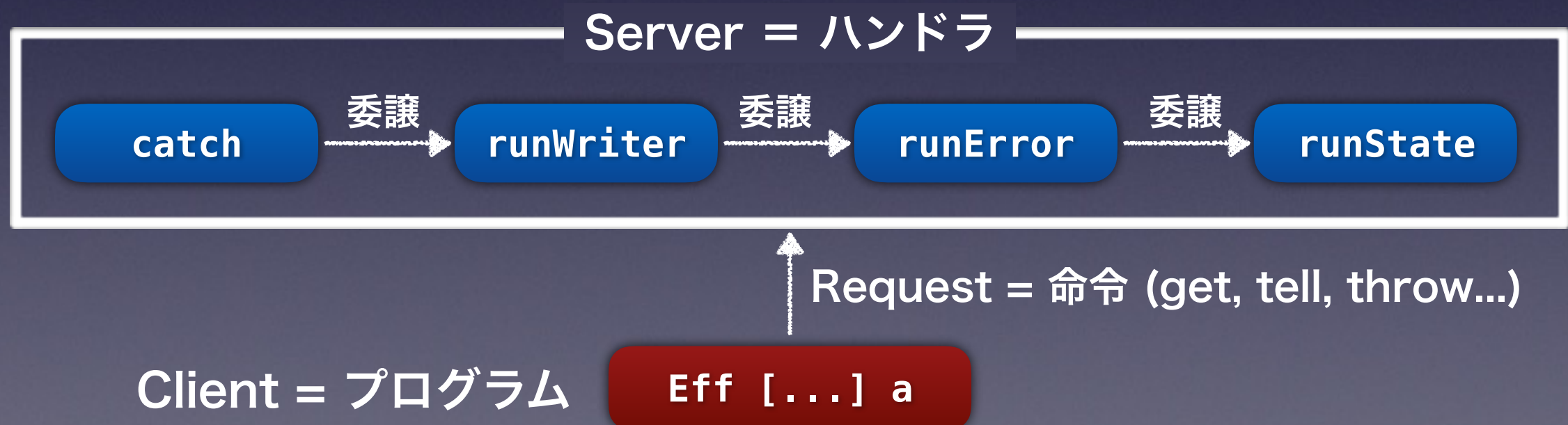
Extensible Effects

- アイデア： **函手 (Functor)** の合成は簡単
- モナド = $\text{join} :: m (m a) \rightarrow m a$ と `return` を持つ Functor
- 副作用を **函手の直和** で表し、その生成する **自由モナド** を考える
- 副作用の処理を **Client-Server モデル** で捉える



Extensible Effects

- アイデア： **函手 (Functor)** の合成は簡単
- モナド = $\text{join} :: m (m a) \rightarrow m a$ と `return` を持つ Functor
- 副作用を **函手の直和** で表し、その生成する **自由モナド** を考える
- 副作用の処理を **Client-Server モデル** で捉える



→ **lift が不要、階層を跨いだ処理が可能** (階層がない)

自由モナド

自由モナド

- ・ クリーネ閉包 Λ^* : Λ の有限文字列全体

$$\begin{array}{c} \Lambda^* \\ \uparrow \\ \{\cdot\} \\ \Lambda \end{array}$$

自由モナド

- ・ **クリーネ閉包 Λ^*** : Λ の有限文字列全体
 - ・ 代数的には **Λ 上の自由モノイド** に相当

$$\begin{array}{c} \Lambda^* \\ \uparrow \\ \{\cdot\} \\ \Lambda \end{array}$$

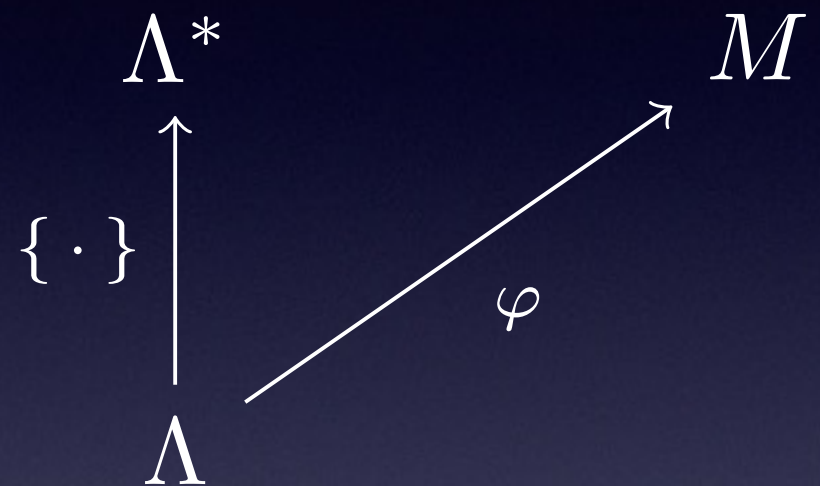
自由モナド

- ・ **クリーネ閉包 Λ^*** : Λ の有限文字列全体
 - ・ 代数的には **Λ 上の自由モノイド** に相当
 - ・ モノイド準同型 $\varphi : \Lambda \rightarrow M$ に対し foldMap
 $\varphi : \Lambda^* \rightarrow M$ は右図を可換にする唯一の写像

$$\begin{array}{c} \Lambda^* \\ \uparrow \\ \{ \cdot \} \\ \Lambda \end{array}$$

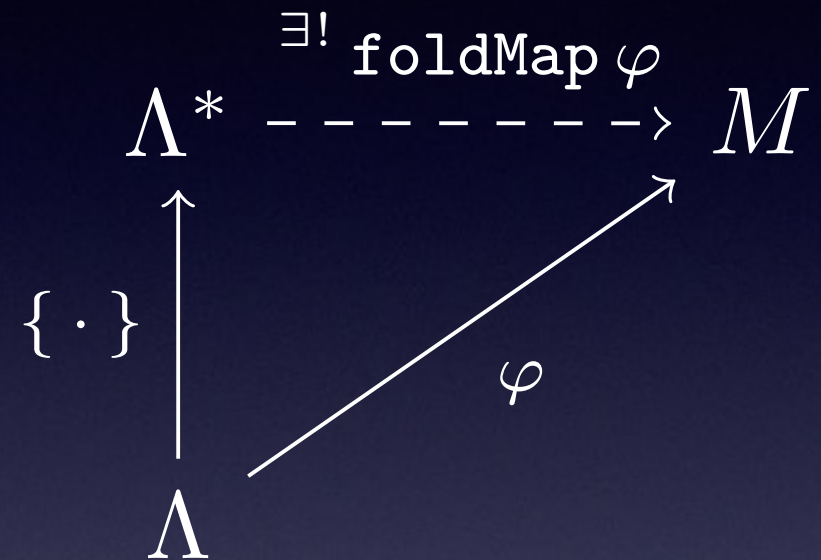
自由モナド

- ・ **クリーネ閉包 Λ^*** : Λ の有限文字列全体
 - ・ 代数的には **Λ 上の自由モノイド** に相当
 - ・ モノイド準同型 $\varphi : \Lambda \rightarrow M$ に対し `foldMap`
 $\varphi : \Lambda^* \rightarrow M$ は右図を可換にする唯一の写像



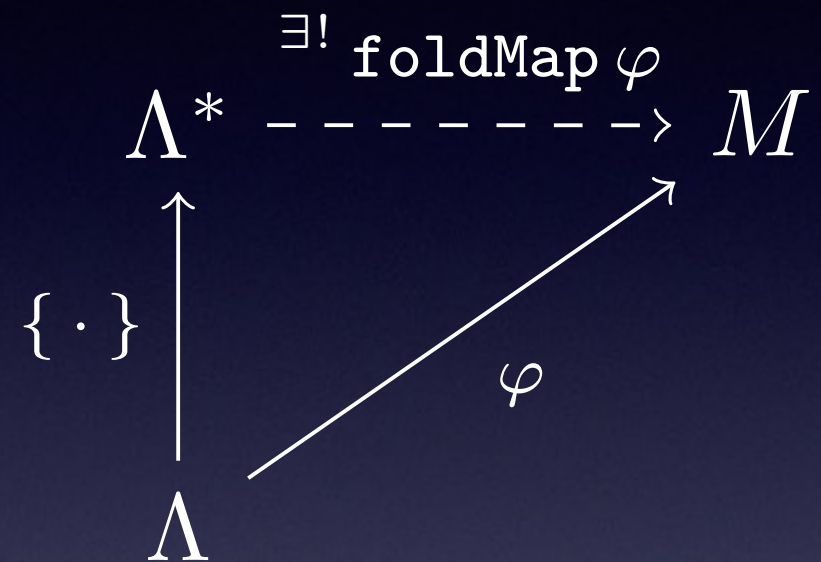
自由モナド

- ・ **クリーネ閉包 Λ^*** : Λ の有限文字列全体
 - ・ 代数的には **Λ 上の自由モノイド** に相当
 - ・ モノイド準同型 $\varphi : \Lambda \rightarrow M$ に対し $\text{foldMap } \varphi : \Lambda^* \rightarrow M$ は右図を可換にする唯一の写像



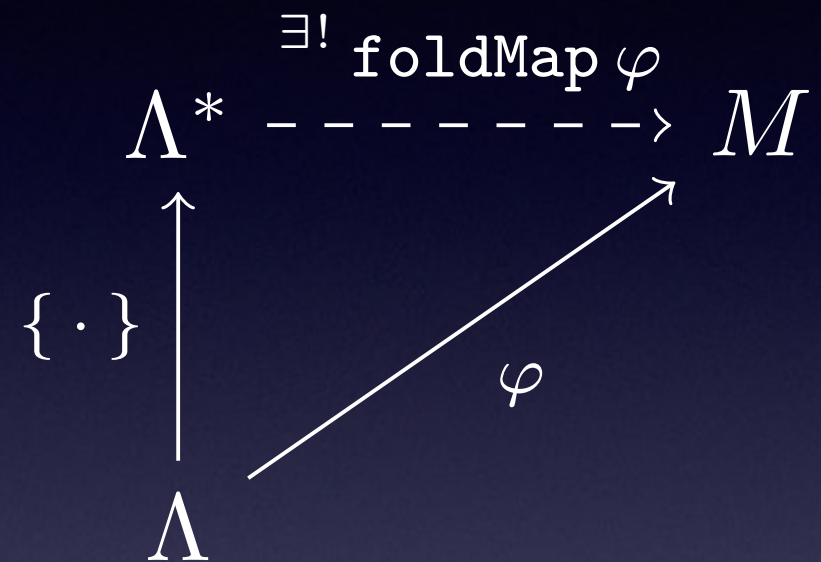
自由モナド

- ・ **クリーネ閉包 Λ^*** : Λ の有限文字列全体
 - ・ 代数的には **Λ 上の自由モノイド** に相当
 - ・ モノイド準同型 $\varphi : \Lambda \rightarrow M$ に対し $\text{foldMap } \varphi : \Lambda^* \rightarrow M$ は右図を可換にする唯一の写像
 - ・ Λ を「含む」最小の**モノイド**構造



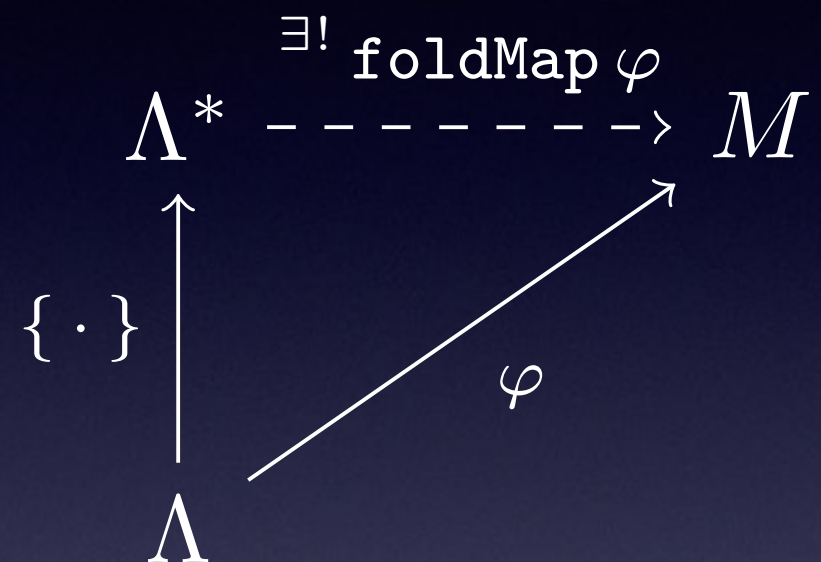
自由モナド

- ・ **クリーネ閉包 Λ^*** : Λ の有限文字列全体
 - ・ 代数的には **Λ 上の自由モノイド** に相当
 - ・ モノイド準同型 $\varphi : \Lambda \rightarrow M$ に対し $\text{foldMap } \varphi : \Lambda^* \rightarrow M$ は右図を可換にする唯一の写像
 - ・ Λ を「含む」最小の**モノイド**構造
- ・ **自由モナド** : 関手 f を命令とする抽象構文木全体



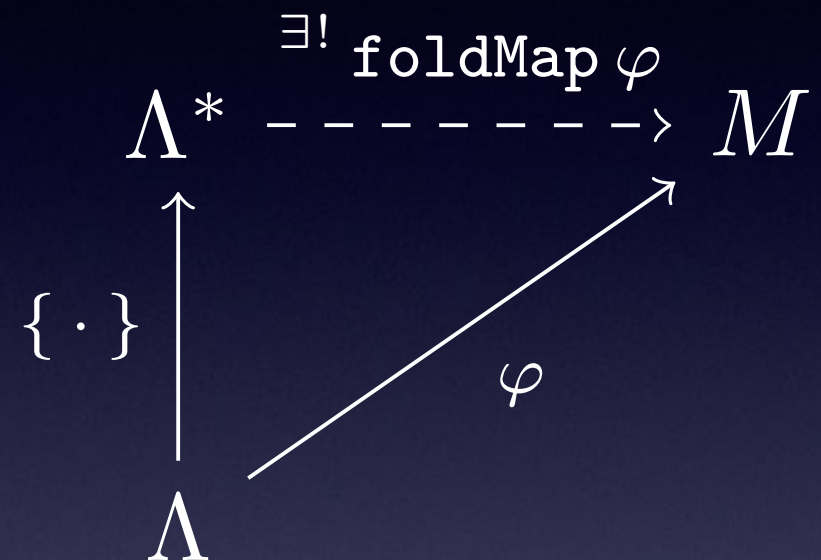
自由モナド

- ・ **クリーネ閉包 Λ^*** : Λ の有限文字列全体
 - ・ 代数的には **Λ 上の自由モノイド** に相当
 - ・ モノイド準同型 $\varphi : \Lambda \rightarrow M$ に対し $\text{foldMap } \varphi : \Lambda^* \rightarrow M$ は右図を可換にする唯一の写像
 - ・ Λ を「含む」最小の**モノイド**構造
- ・ **自由モナド** : 関手 f を命令とする**抽象構文木全体**
 - ・ 関手 f を「含む」最小の**モナド**



自由モナド

- ・ **クリーネ閉包 Λ^*** : Λ の有限文字列全体
 - ・ 代数的には **Λ 上の自由モノイド** に相当
 - ・ モノイド準同型 $\varphi : \Lambda \rightarrow M$ に対し $\text{foldMap } \varphi : \Lambda^* \rightarrow M$ は右図を可換にする唯一の写像
 - ・ Λ を「含む」最小の**モノイド**構造

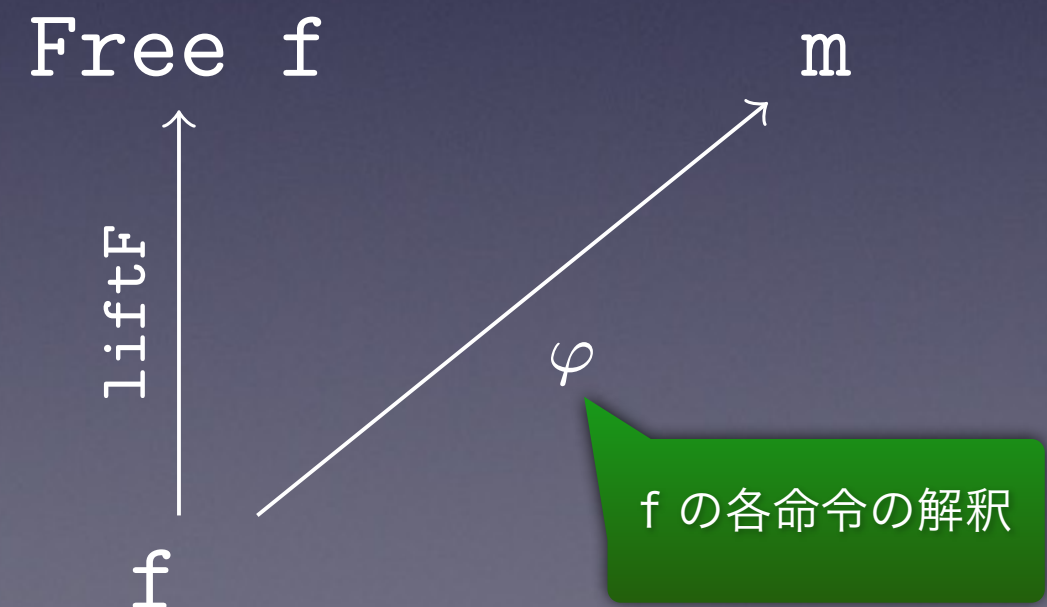
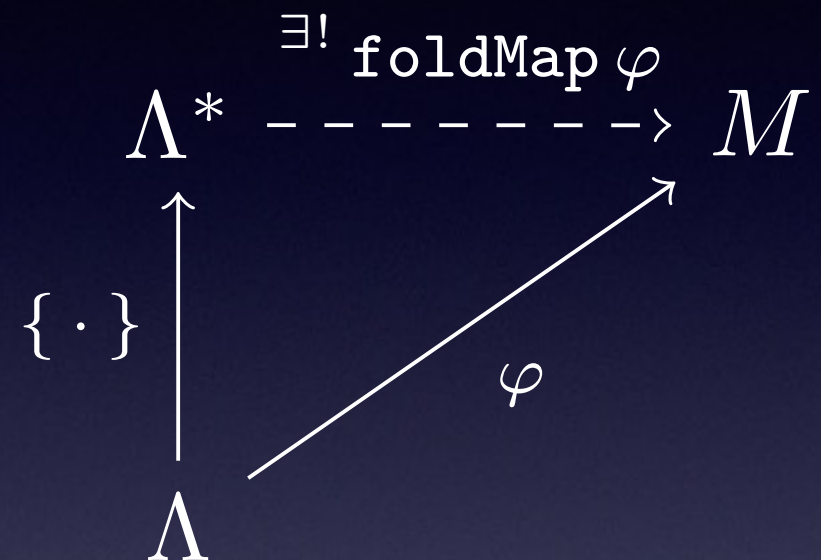


- ・ **自由モナド** : 関手 f を命令とする**抽象構文木全体**
 - ・ 関手 f を「含む」最小の**モナド**
 - ・ 普遍性は f の解釈 $\varphi : \forall \alpha. f \alpha \rightarrow m \alpha$ から再帰的に**インタプリタ** $\text{handle } \varphi : \forall \alpha. \text{Free } f \alpha \rightarrow m \alpha$ が定まる事に対応



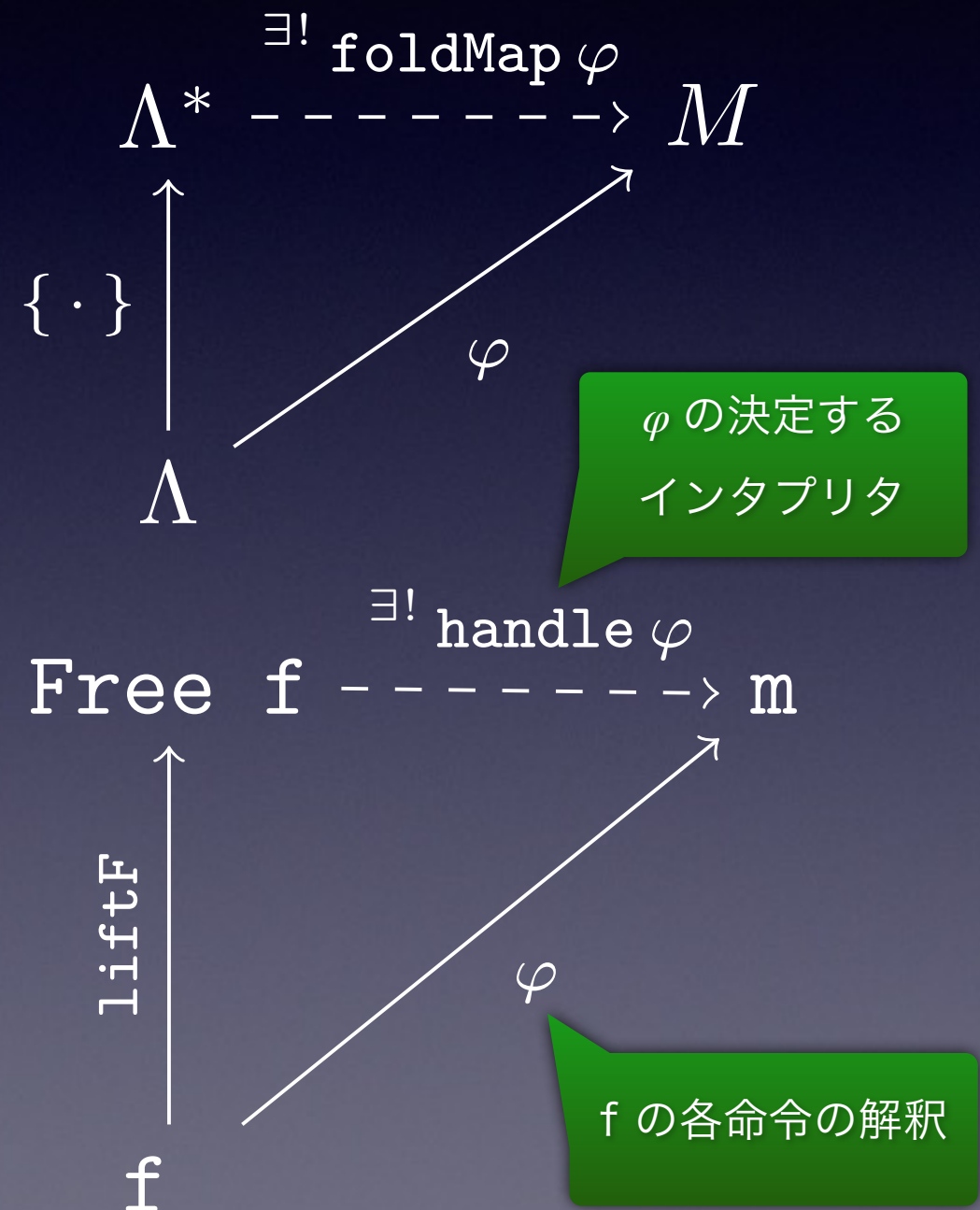
自由モナド

- ・ **クリーネ閉包 Λ^*** : Λ の有限文字列全体
 - ・ 代数的には **Λ 上の自由モノイド** に相当
 - ・ モノイド準同型 $\varphi : \Lambda \rightarrow M$ に対し $\text{foldMap } \varphi : \Lambda^* \rightarrow M$ は右図を可換にする唯一の写像
 - ・ Λ を「含む」最小の**モノイド**構造
- ・ **自由モナド** : 関手 f を命令とする抽象構文木全体
 - ・ 関手 f を「含む」最小の**モナド**
 - ・ 普遍性は f の解釈 $\varphi : \forall \alpha. f \alpha \rightarrow m \alpha$ から再帰的に**インタプリタ** $\text{handle } \varphi : \forall \alpha. \text{Free } f \alpha \rightarrow m \alpha$ が定まる事に対応



自由モナド

- ・ **クリーネ閉包 Λ^*** : Λ の有限文字列全体
 - ・ 代数的には **Λ 上の自由モノイド** に相当
 - ・ モノイド準同型 $\varphi : \Lambda \rightarrow M$ に対し $\text{foldMap } \varphi : \Lambda^* \rightarrow M$ は右図を可換にする唯一の写像
 - ・ Λ を「含む」最小の**モノイド**構造
- ・ **自由モナド** : 関手 f を命令とする**抽象構文木全体**
 - ・ 関手 f を「含む」最小の**モナド**
 - ・ 普遍性は f の解釈 $\varphi : \forall \alpha. f \alpha \rightarrow m \alpha$ から再帰的に**インタプリタ** $\text{handle } \varphi : \forall \alpha. \text{Free } f \alpha \rightarrow m \alpha$ が定まる事に対応



自由モナド

```
data Free f α = Pure α
              | Join (f (Free f α))
              deriving (Functor)

handle' :: Functor f => (f a → a) → Free f a → a

liftF :: Functor f => f a → Free f a

instance (Functor f) => Monad (Free f) where
    return = Pure
    Pure a >>= f = f a
    Join mma >>= f = Join (fmap (>>= f) mma)
```

自由モナド

- 抽象構文木：モナド = 関手 + **join** + **return**

```
data Free f α = Pure α
               | Join (f (Free f α))
               deriving (Functor)

handle' :: Functor f => (f a → a) → Free f a → a

liftF :: Functor f => f a → Free f a

instance (Functor f) => Monad (Free f) where
    return = Pure
    Pure a >>= f = f a
    Join mma >>= f = Join (fmap (>>= f) mma)
```

自由モナド

- ・ 抽象構文木：モナド = 関手 + **join** + **return**
 - ・ `Free f` はこれらを頂点に持つ木構造として定義出来る

```
data Free f α = Pure α
              | Join (f (Free f α))
              deriving (Functor)

handle' :: Functor f => (f a → a) → Free f a → a

liftF :: Functor f => f a → Free f a

instance (Functor f) => Monad (Free f) where
    return = Pure
    Pure a >>= f = f a
    Join mma >>= f = Join (fmap (>>= f) mma)
```


自由モナド

- ・ 抽象構文木：モナド = 関手 + **join** + **return**
 - ・ `Free f` はこれらを頂点に持つ木構造として定義出来る
 - ・ `Monad` の定義は型をグッと睨めば導出できる

```
data Free f α = Pure α
              | Join (f (Free f α))
              deriving (Functor)

handle' :: Functor f => (f a → a) → Free f a → a

liftF :: Functor f => f a → Free f a

instance (Functor f) => Monad (Free f) where
    return = Pure
    Pure a >>= f = f a
    Join mma >>= f = Join (fmap (>>= f) mma)
```

Reader (環境モナド)

```
data Ask i a = Ask (i → a) deriving (Functor)

ask :: Free (Ask e) e
ask = liftF (Ask id)

runReader :: e → Free (Ask e) a → a
runReader e = handle (λ (Ask k) → k e)

runIt :: [e] → Free (Ask e) a → a
runIt _ (Pure a) = a
runIt (x : xs) (Join (Ask k)) = runIt xs (k x)
```

- ・ 関手にするため副作用を継続渡し形式 (CPS) で記述
- ・ 複数の解釈が可能 (自由モナドの普遍性)


Reader (環境モナド)

```
data Ask i a = Ask (i → a) deriving (Functor)

ask :: Free (Ask e) e
ask = liftF (Ask id)

runReader :: e → Free (Ask e) a → a
runReader e = handle (λ (Ask k) → k e)

runIt :: [e] → Free (Ask e) a → a
runIt _ (Pure a) = a
runIt (x : xs) (Join (Ask k)) = runIt xs (k x)
```



- ・ 関手にするため副作用を継続渡し形式 (CPS) で記述
- ・ 複数の解釈が可能 (自由モナドの普遍性)

複数の解釈が可能

```
ghci> runReader 12                $ ask >> (,) <$> ask <*> ask  
    => (12,12)  
  
ghci> runIt      [12, 23, 34] $ ask >> (,) <$> ask <*> ask  
    => (23,34)
```


直和 : $\text{State} = \text{Ask} + \text{Tell}$

直和 : $\text{State} = \text{Ask} + \text{Tell}$

- Writer も同様に継続渡しで作れる

```
data Tell e a = Tell a e deriving (Functor)
```

直和 : State = Ask + Tell

- Writer も同様に継続渡しで作れる

継続

```
data Tell e a = Tell a e deriving (Functor)
```

直和 : State = Ask + Tell

- Writer も同様に継続渡しで作れる

```
data Tell e a = Tell a e deriving (Functor)
```

継続

ログ出力

直和：State = Ask + Tell

- Writer も同様に継続渡しで作れる

```
data Tell e a = Tell a e deriving (Functor)
```



- Tell と Ask で State を創るには？

直和 : $\text{State} = \text{Ask} + \text{Tell}$

- Writer も同様に継続渡しで作れる

```
data Tell e a = Tell a e deriving (Functor)
```

継続 ログ出力

- Tell と Ask で State を創るには？

→ 関手の**直和** $f \oplus g : f, g$ いずれかの構築子を持つ関手

```
data (f  $\oplus$  g) a = InL (f a)
                  | InR (g a) deriving (Functor)

runState :: s  $\rightarrow$  Free (Tell s  $\oplus$  Ask s) a  $\rightarrow$  (a, s)
runState _ (Join (InL (Tell fa e))) = runState e fa
runState s0 (Join (InR (Ask k))) = runState s0 (k s0)
```

直和 : $\text{State} = \text{Ask} + \text{Tell}$

- Writer も同様に継続渡しで作れる

```
data Tell e a = Tell a e deriving (Functor)
```

継続 ログ出力

- Tell と Ask で State を創るには？

→ 関手の**直和** $f \oplus g : f, g$ いずれかの構築子を持つ関手

```
data (f  $\oplus$  g) a = InL (f a)
                  | InR (g a) deriving (Functor)

runState :: s  $\rightarrow$  Free (Tell s  $\oplus$  Ask s) a  $\rightarrow$  (a, s)
runState _ (Join (InL (Tell fa e))) = runState e fa
runState s0 (Join (InR (Ask k))) = runState s0 (k s0)
```

- 更に複数の効果を合成するには、多項和に一般化すればよい

Open Union

- Open Union: 関手の合成を任意個に一般化

```
data Union (fs :: [★ → ★]) a
class f ∈ fs    -- 「f は関手のリスト fs の要素」

inj    :: f ∈ fs ⇒ f a → Union fs a    -- InL, InR に対応
prj    :: f ∈ fs ⇒ Union fs a → Maybe (f a)
decomp :: Union (f : fs) a → Either (Union fs a) (f a)
weaken :: Union fs a → Union (f : fs) a
absurd :: Union '[] a → b
```

- 旧 ExtEff [KSS13] では動的キャストを使用

完成：Extensible Effects

```
newtype Eff r a = Eff { unEff :: Free (Union r) a }  
                  deriving (Functor, Monad, Applicative)
```

- ・ 原論文では、これに更にCPS変換を施している
- ・ モナド変換子の問題のうち、以下は解決
 1. 意味論の固定
 2. Too many lifts!!!
 3. 合成順の固定
 - 3 については後ほど More の章で詳説

Extensible Effects まとめ

Extensible Effects まとめ

Extensible Effects

||

Free Monad

+

Open Union

+

継続渡し形式 (CPS)

Ext Eff の問題点

Ext Eff の問題点

- ・ 合成に **Functor が必要**、fmap にコスト

Ext Eff の問題点

- ・ 合成に **Functor が必要**、fmap にコスト
- ・ 自由モナドは効率が悪く、**reflectionにも不向き**

Ext Eff の問題点

- ・ 合成に **Functor が必要**、fmap にコスト
- ・ 自由モナドは効率が悪く、**reflectionにも不向き**
- ・ **継続ベース**で、ハンドラが書きづらい

Ext Eff の問題点

- ・ 合成に **Functor が必要**、fmap にコスト
- ・ 自由モナドは効率が悪く、**reflectionにも不向き**
- ・ **継続ベース**で、ハンドラが書きづらい
 - ・ ライブラリ開発が面倒

Ext Eff の問題点

- ・ 合成に **Functor が必要**、fmap にコスト
- ・ 自由モナドは効率が悪く、**reflectionにも不向き**
- ・ **継続ベース**で、ハンドラが書きづらい
 - ・ ライブラリ開発が面倒
- ・ 技術的な問題：**動的キャストの利用**

Ext Eff の問題点

- ・ 合成に **Functor が必要**、fmap にコスト
- ・ 自由モナドは効率が悪く、**reflectionにも不向き**
- ・ **継続ベース**で、ハンドラが書きづらい
 - ・ ライブラリ開発が面倒
- ・ 技術的な問題：**動的キャストの利用**
 - ・ **実行時コスト**が嵩む

Ext Eff の問題点

- ・ 合成に **Functor が必要**、fmap にコスト
- ・ 自由モナドは効率が悪く、**reflectionにも不向き**
- ・ **継続ベース**で、ハンドラが書きづらい
 - ・ ライブラリ開発が面倒
- ・ 技術的な問題：**動的キャストの利用**
 - ・ **実行時コスト**が嵩む
 - ・ 基底モナドに ST s など**SkoLem変数を含む型が使えない**

Ext Eff の問題点

- ・ 合成に **Functor が必要**、fmap にコスト
- ・ 自由モナドは効率が悪く、**reflectionにも不向き**
- ・ **継続ベース**で、ハンドラが書きづらい
 - ・ ライブラリ開発が面倒
- ・ 技術的な問題：**動的キャストの利用**
 - ・ **実行時コスト**が嵩む
 - ・ 基底モナドに ST s など**SkoLem変数を含む型が使えない**

★ **More Extensible Effects** (提案手法) はこれらを解決

Freer Monads, More Extensible Effects

Extensible Effects

Extensible Effects

||

Free Monad

+

Open Union

+

継続渡し形式 (CPS)

More Extensible Effects

More Extensible Effects

||

Free **r** Monad — = Operational
Monad

+

Open Union

+

Type-aligned Sequence

Freer Monad とは

Freer Monad とは

- ・ ?? 「Freer Monad は単項型の恒等関手に沿った左 Kan 拡張だよ。何か問題でも？」

Freer Monad とは

- ・ ?? 「Freer Monad は単項型の恒等関手に沿った左 Kan 拡張だよ。何か問題でも？」
 - 問題しかない (少なくとも私にとっては)

Freer Monad とは

- ・ ?? 「Freer Monad は単項型の恒等関手に沿った左 Kan 拡張だよ。何か問題でも？」
 - ➔ 問題しかない（少なくとも私にとっては）
- ・ これまでを踏まえた説明：

Freer Monad とは

- ・ ?? 「Freer Monad は単項型の恒等関手に沿った左 Kan 拡張だよ。何か問題でも？」
 - 問題しかない（少なくとも私にとっては）
- ・ これまでを踏まえた説明：
 - ・ Freer Monad = 自由 Functor + 自由モナド

自由 Functor と Freer Monad

```
data FFree f a =  $\forall b. \text{FMap } (b \rightarrow a) (f\ b)$   
  
instance Functor (FFree f) where  
    fmap f (FMap g a) = FMap (f  $\circ$  g) a  
  
type Freer f a = Free (FFree f) a
```

自由 Functor と Freer Monad

```
data FFree f a =  $\forall b$ . FMap (b  $\rightarrow$  a) (f b)
```

```
instance Functor (FFree f) where  
  fmap f (FMap g a) = FMap (f  $\circ$  g) a
```

```
type Freer f a = Free (FFree f) a
```

fmap !

自由 Functor と Freer Monad

```
data FFree f a =  $\forall b$ . FMap (b  $\rightarrow$  a) (f b)
```

```
instance Functor (FFree f) where  
  fmap f (FMap g a) = FMap (f  $\circ$  g) a
```

fmap !

```
type Freer f a = Free (FFree f) a
```

- 任意の単項型構築子 $f :: \star \rightarrow \star$ に対し、 $\text{FFree } f :: \star \rightarrow \star$ は自動的にFunctor (= 自由Functor = Idに沿った左Kan拡張)

自由 Functor と Freer Monad

```
data FFree f a =  $\forall b.$  FMap (b  $\rightarrow$  a) (f b)

instance Functor (FFree f) where
    fmap f (FMap g a) = FMap (f  $\circ$  g) a

type Freer f a = Free (FFree f) a
```

- 任意の単項型構築子 $f :: \star \rightarrow \star$ に対し、 $\text{FFree } f :: \star \rightarrow \star$ は自動的にFunctor (= 自由Functor = Idに沿った左Kan拡張)
- FFree と Free で **Functor**を仮定せずにモナドが出来る

Freer の定義の導出

Freer の定義の導出

- Free と FFree の定義を展開する

```
Freer f a = Free (FFree f) a
          ≈ Pure a
          | Impure (FFree f (Freer f a))
          ≈ Pure a
          | ∀ b. Impure (f b) (b → Freer f a)
```

Freer の定義の導出

- Free と FFree の定義を展開する

```
Freer f a = Free (FFree f) a
          ≈ Pure a
          | Impure (FFree f (Freer f a))
          ≈ Pure a
          | ∀ b. Impure (f b) (b → Freer f a)
```

- f を OpenUnion で置き換えれば：

```
data Eff r a
  = Pure a
  | ∀ b. Impure (Union r b) (b → Eff r a)
```

Freer の定義の導出

- Free と FFree の定義を展開する

```
Freer f a = Free (FFree f) a
          ≈ Pure a
          | Impure (FFree f (Freer f a))
          ≈ Pure a
          | ∀ b. Impure (f b) (b → Freer f a)
```

- f を OpenUnion で置き換えれば：

```
data Eff r a
  = Pure a
  | ∀ b. Impure (Union r b) (b → Eff r a)
```

 return

Freer の定義の導出

- Free と FFree の定義を展開する

```
Freer f a = Free (FFree f) a
          ≈ Pure a
          | Impure (FFree f (Freer f a))
          ≈ Pure a
          | ∀ b. Impure (f b) (b → Freer f a)
```

- f を OpenUnion で置き換えれば：

```
data Eff r a
  = Pure a
  | ∀ b. Impure (Union r b) (b → Eff r a)
```

return

bind (>>=)!

インスタンス定義

```
data Eff r a
  = Pure a
  | ∀ b. Impure (Union r b) (b → Eff r a)

instance Monad (Eff f) where
  return = Pure
  (Pure a)      >>= f = f a
  (Impure fa k) >>= f = Impure fa (k >=> f)
```

- ・ 継続を合成していけば良いだけの素直な実装
- ・ Functor 不要！ (fmap f a = (return . f) =<< a)

ハンドラ実装用便利関数

return !

```
handle_relay :: (a → Eff r w)
              → (∀ v. t v → (v → Eff r w) → Eff r w)
              → Eff (t ': r) a → Eff r w
```

(>>=) !

```
handle_relay ret h m = ...
```

```
send :: (t ∈ r) ⇒ t v → Eff r v
send cmd = Impure (inj cmd) Pure
```

- ・ 古い ExtEff だとここまで使い易い型にならない
- ・ リレーしていただくだけでこれらの実装も容易
- ・ ループ内部状態ありの版や、効果を取り除かない版も可能

例：Ask, Tell 再訪

```
data Ask i a where Ask :: Ask i i
data Tell i a where Tell :: i → Tell i ()

ask :: (Ask i ∈ r) ⇒ Eff r i
ask = send Ask

tell :: (Tell i ∈ r) ⇒ i → Eff r ()
tell = send ∘ Tell
```

例：Ask, Tell 再訪

```
data Ask i a where Ask :: Ask i i
data Tell i a where Tell :: i → Tell i ()

ask :: (Ask i ∈ r) ⇒ Eff r i
ask = send Ask

tell :: (Tell i ∈ r) ⇒ i → Eff r ()
tell = send ∘ Tell
```

- ・ プリミティブな関数とほぼ同じ型のデータ構築子を用意

例：Ask, Tell 再訪

```
data Ask i a where Ask :: Ask i i
data Tell i a where Tell :: i → Tell i ()

ask :: (Ask i ∈ r) ⇒ Eff r i
ask = send Ask

tell :: (Tell i ∈ r) ⇒ i → Eff r ()
tell = send ∘ Tell
```

- ・ プリミティブな関数とほぼ同じ型のデータ構築子を用意
→ 旧来の ExtEff での定義に比べて非常に素直

ハンドラの実装

```
runReader :: e → Eff (Ask e : r) a → Eff r a
runReader e = handle_relay return (λ Ask arr → arr e)

runIt :: [e] → Eff (Ask e : r) a → Eff r a
runIt = handle_relay_s (λ _ a → return a) $
  λ (x : xs) Ask arr → arr xs x

runWriter :: Eff (Tell e : r) a → Eff r (a, [e])
runWriter = handle_relay (λa → return (a, [])) $
  λ (Tell e) f → do { (a, es) ← f () ; return (a, e:es) }
```

- ・ 実質 return と (>>=) を書くだけなので楽

More Extensible Effects

More Extensible Effects

||

Freer Monad — = Operational
Monad

+

Open Union

+

Type-aligned Sequence

More Extensible Effects

More Extensible Effects

||

Freer Monad

+

Open Union

+

Type-aligned Sequence

これまでの実装の欠点

- Eff モナドインスタンスの定義

```
instance Monad (Eff f) where
  return = Pure
  (Pure x)      >>= f = f x
  (Impure fa k) >>= f = Impure fa (k >=> f)
```


これまでの実装の欠点

- Eff モナドインスタンスの定義

```
instance Monad (Eff f) where
  return = Pure
  (Pure x)      >>= f = f x
  (Impure fa k) >>= f = Impure fa (k >=> f)
```

- 継続が第二引数に左結合的に蓄積されていく

これまでの実装の欠点

- Eff モナドインスタンスの定義

```
instance Monad (Eff f) where
  return = Pure
  (Pure x)      >>= f = f x
  (Impure fa k) >>= f = Impure fa (k >=> f)
```

- 継続が第二引数に左結合的に蓄積されていく
 - ハンドラは命令列を頭から逐次実行する
 - 左結合だと、最初の数個だけが必要でも継続全体を走査する必要がある！

これまでの実装の欠点

- Eff モナドインスタンスの定義

```
instance Monad (Eff f) where
  return = Pure
  (Pure x)      >>= f = f x
  (Impure fa k) >>= f = Impure fa (k >=> f)
```

- 継続が第二引数に左結合的に蓄積されていく
 - ハンドラは命令列を頭から逐次実行する
 - 左結合だと、最初の数個だけが必要でも継続全体を走査する必要がある！

→ 解決策：Type-aligned Sequence [PK14]

Type-aligned Sequence

Type-aligned Sequence

- ・ 直感：関数を実際に合成する代わりに**関数の列**を考えて**必要になったら適宜評価**していけばよい

Type-aligned Sequence

- ・ 直感：関数を実際に合成する代わりに**関数の列**を考えて**必要になったら適宜評価**していけばよい
 - ・ 安価な snoc, cat, uncons を持つ二分木 + 型 Hack

Type-aligned Sequence

- ・ 直感：関数を実際に合成する代わりに**関数の列**を考えて**必要になったら適宜評価**していけばよい
 - ・ 安価な snoc, cat, uncons を持つ二分木 + 型 Hack
 - ・ 初出では Reflection w/o Remorse そのものを使っていた

Type-aligned Sequence

- ・ 直感：関数を実際に合成する代わりに**関数の列**を考えて**必要になったら適宜評価**していけばよい
- ・ 安価な snoc, cat, uncons を持つ二分木 + 型 Hack
- ・ 初出では Reflection w/o Remorse そのものを使っていた

```
type FTCQ (m :: ★ → ★) a b
tsingleton :: (a → m b) → FTCQ m a b
(▷) :: FTCQ m a c → (c → m b) → FTCQ m a b
(⊗) :: FTCQ m a c → FTCQ m c b → FTCQ m a b
data ViewL m a b where
  TOne    :: (a → m b) → ViewL m a b
  (:|)    :: (a → m c) → (FTCQ m c b) → ViewL m a b
tviewl :: FTCQ m a b → ViewL m a b
```

Type-aligned Sequence

- ・ 直感：関数を実際に合成する代わりに**関数の列**を考えて**必要になったら適宜評価**していけばよい
- ・ 安価な snoc, cat, uncons を持つ二分木 + 型 Hack
- ・ 初出では Reflection w/o Remorse そのものを使っていた

```
type FTCQ (m :: ★ → ★) a b
singleton :: (a → m b) → FTCQ m a b
(▷) :: FTCQ m a c → (c → m b) → FTCQ m a b
(⊗) :: FTCQ m a c → FTCQ m c b → FTCQ m a b
data ViewL m a b where
  TOne    :: (a → m b) → ViewL m a b
  (:|)    :: (a → m c) → (FTCQ m c b) → ViewL m a b
tviewl :: FTCQ m a b → ViewL m a b
```

$a \rightarrow m b$ に対応

Type-aligned Sequence

- ・ 直感：関数を実際に合成する代わりに**関数の列**を考えて**必要になったら適宜評価**していけばよい
- ・ 安価な snoc, cat, uncons を持つ二分木 + 型 Hack
- ・ 初出では Reflection w/o Remorse そのものを使っていた

```
type FTCQ (m :: ★ → ★) a b
tsingleton :: (a → m b) → FTCQ m a b
(▷) :: FTCQ m a c → (c → m b) → FTCQ m a b
(⊗) :: FTCQ m a c → FTCQ m c b → FTCQ m a b
data ViewL m a b where
  TOne    :: (a → m b) → ViewL m a b
  (:|)    :: (a → m c) → (FTCQ m c b) → ViewL m a b
tviewl :: FTCQ m a b → ViewL m a b
```

$a \rightarrow m b$ に対応

(\Rightarrow)

Type-aligned Sequence

- ・ 直感：関数を実際に合成する代わりに**関数の列**を考えて**必要になったら適宜評価**していけばよい
- ・ 安価な snoc, cat, uncons を持つ二分木 + 型 Hack
- ・ 初出では Reflection w/o Remorse そのものを使っていた

```
type FTCQ (m :: ★ → ★) a b
tsingleton :: (a → m b) → FTCQ m a b
(▷) :: FTCQ m a c → (c → m b) → FTCQ m a b
(⊗) :: FTCQ m a c → FTCQ m c b → FTCQ m a b
data ViewL m a b where
  TOne    :: (a → m b) → ViewL m a b
  (:|)    :: (a → m c) → (FTCQ m c b) → ViewL m a b
tviewl :: FTCQ m a b → ViewL m a b
```

$a \rightarrow m b$ に対応

(\Rightarrow)

uncons

最終結果

```
type Arrs r a b = FTCQ (Eff r) a b
data Eff r a = Pure a
              | ∀ b. Impure (Union r b) (Arrs r b a)

instance Monad (Eff r) where
  return = Pure
  (Pure a) >>= f = f a
  (Impure fa k) >>= f = Impure fa (k ▷ f)

handle_relay :: ...
```

- Union と FTCQ の実装を改善していけば、効率が劇的に改善する

Example

Example : 例外処理

Example : 例外処理

- ・ MTL が最も苦手とする物の一つ
- ・ 「階層を跨いだ処理」が効いてくる場面の一つ
- ・ (以下、わかりやすさのため Refl w/o Remorse 適用前のコード)

例外作用の定義

```
data Exc e a = ThrowError e

throwError :: Exc e ∈ r ⇒ e → Eff r a

catchError :: ∀ e r a. Exc e ∈ r
              ⇒ Eff r a → (e → Eff r a) → Eff r a
catchError act h = interpose return bind act where
  bind :: Exc e b → (b → Eff r a) → Eff r a
  bind (Exc e) _ = h e

runError :: Eff (Exc e : r) a → Eff r (Either e a)
runError = handle_relay (return ∘ Right) $
  λ(ThrowError e) _ → return (Left e)
```

例外作用の定義

```
data Exc e a = ThrowError e
throwError :: Exc e ∈ r ⇒ e → Eff r a
catchError :: ∀ e r a. Exc e ∈ r
              ⇒ Eff r a → (e → Eff r a) → Eff r a
catchError act h = interpose return bind act where
  bind :: Exc e b → (b → Eff r a) → Eff r a
  bind (Exc e) _ = h e
runError :: Eff (Exc e : r) a → Eff r (Either e a)
runError = handle_relay (return ∘ Right) $
  λ(ThrowError e) _ → return (Left e)
```

- **interpose**: 作用を除去しないハンドラを書くためのもの

例外作用の定義

```
data Exc e a = ThrowError e

throwError :: Exc e ∈ r ⇒ e → Eff r a

catchError :: ∀ e r a. Exc e ∈ r
              ⇒ Eff r a → (e → Eff r a) → Eff r a

catchError act h = interpose return bind act where
  bind :: Exc e b → (b → Eff r a) → Eff r a
  bind (Exc e) _ = h e

runError :: Eff (Exc e : r) a → Eff r (Either e a)
runError = handle_relay (return ∘ Right) $
  λ(ThrowError e) _ → return (Left e)
```

- **interpose**: 作用を除去しないハンドラを書くためのもの
- 例外があれば対処すれば良いだけなので素直で簡潔

トランザクション

```
transaction ::  $\forall s\ r\ a.$  (State s)  $\in$  r  
               $\Rightarrow$  Proxy s  $\rightarrow$  Eff r a  $\rightarrow$  Eff r a  
transaction _ act = do s  $\leftarrow$  get; loop s act where  
  loop :: s  $\rightarrow$  Eff r a  $\rightarrow$  Eff r a  
  loop s (Pure a) = put s  $\gg$  return a  
  loop s (Impure (u :: Union r b) k) =  
    case prj u :: Maybe (State s b) of  
      Just Get            $\rightarrow$  loop s $ k s  
      Just (Put s')       $\rightarrow$  loop s' $ k ()  
      Nothing             $\rightarrow$  Impure u (loop s  $\circ$  k)
```

- ・ 例外等で動作が中断した場合、継続は呼ばれない
 \rightarrow Pure で末端に達したら状態を大域的に反映すればよい

実行例

```
transTest
  = runLift $ flip runState True $
    flip runState 'a' $
    runError' (Proxy :: Proxy String) $ handled $
      transaction (Proxy :: Proxy Bool) $ do
        modify (succ :: Char → Char)
        modify not
        throwError "interrupted!"
where handled = flip catchError $ λ str →
  lift (putStrLn ("ignored: " ++ str))

ghci> transTest
ignored: interrupted!
==> ((Right (), 'b'), True)
```


実行例

```
transTest
  = runLift $ flip runState True $
    flip runState 'a' $
    runError' (Proxy :: Proxy String) $ handled $
      transaction (Proxy :: Proxy Bool) $ do
        modify (succ :: Char → Char)
        modify not
        throwError "interrupted!"
where handled = flip catchError $ λ str →
    lift (putStrLn ("ignored: " ++ str))
```

Bool の状態のみ
トランザクションで保護

```
ghci> transTest
ignored: interrupted!
==> ((Right (), 'b'), True)
```

実行例

```
transTest
= runLift $ flip runState True $
  flip runState 'a' $
  runError' (Proxy :: Proxy String) $ handled $
    transaction (Proxy :: Proxy Bool) $ do
      modify (succ :: Char → Char)
      modify not
      throwError "interrupted!"
where handled = flip catchError $ λ str →
  lift (putStrLn ("ignored: " ++ str))
```

Bool の状態のみ
トランザクションで保護

```
ghci> transTest
ignored: interrupted!
==> ((Right (), 'b'), True)
```

保護されていたBool
の変更は反映されない

MTLでは？

```
ghci> let handler = flip catchError $ \s →  
        liftIO (putStrLn ("ignored: " ++ s))  
  
ghci> runExceptT $ flip runStateT True $ handler $  
        (modify not >> throwError "hay")  
ignored: hay  
==> Right ((),True)  
  
ghci> flip runStateT True $ runExceptT $ handler $  
        (modify not >> throwError "hay")  
ignored: hay  
==> (Right (),False)
```

- ・ 階層を跨げないので transaction に相当する関数は実装不能
- ・ しかも合成順によって挙動が違う！
 - ・ ExceptT が先頭なら全部コミット、そうでなきゃロールバック

まとめ・その他

- ・ (More) Extensible Effects では、階層を跨いだハンドラが実装出来る
 - ・ **MTL では実現不可能**だった処理も実装可能
 - ・ I/Oエラーの捕捉や、Error モナドへのリレーも容易
- ・ 原論文 [KI15] には論理モナドやリージョン計算など、他の例もあり
- ・ More EE それ自体をモナド変換子として用いる事も可能（基底モナド付き計算）

Benchmarks

Benchmarks

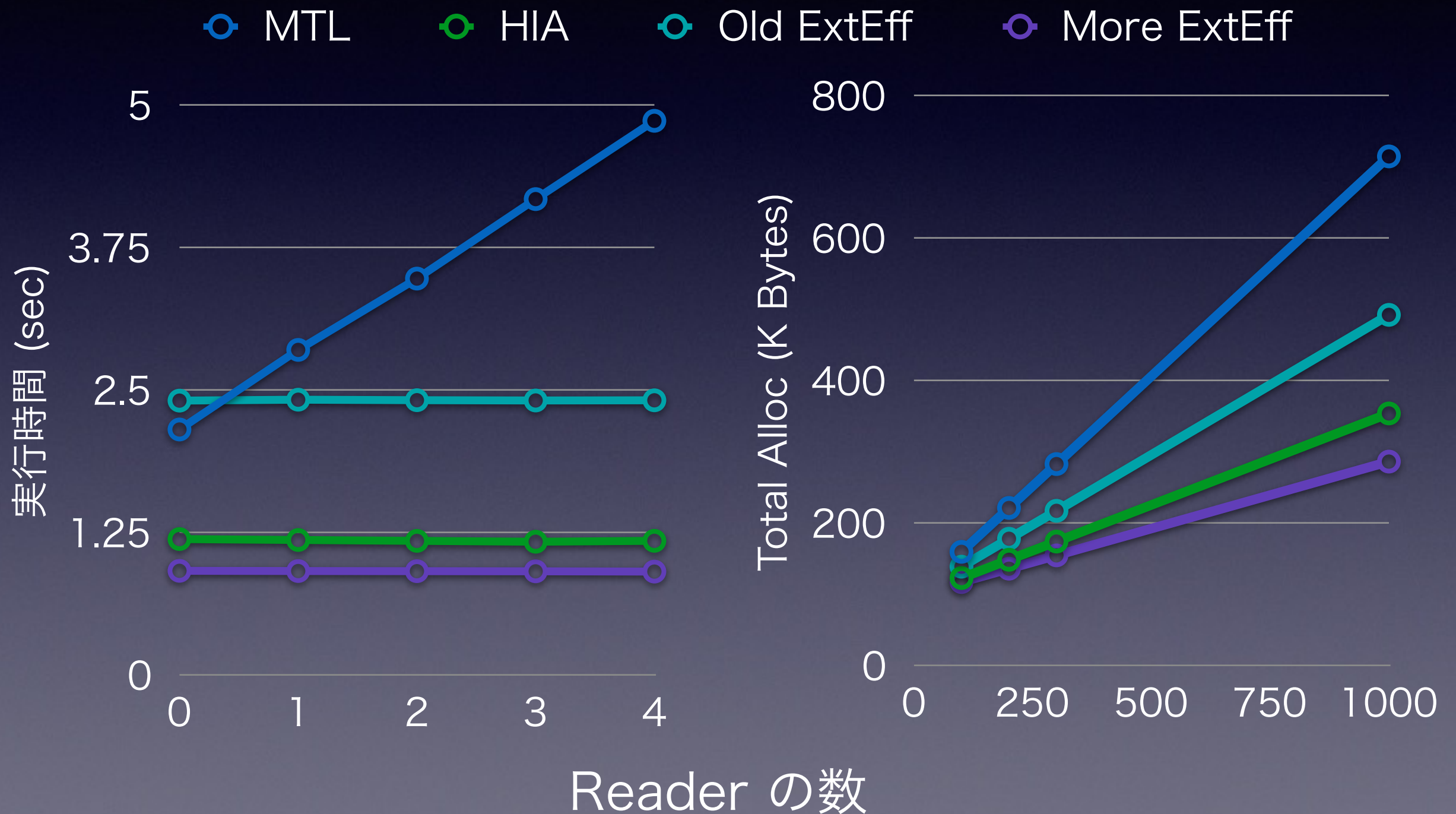
- ・ メイン：5の倍数を数える
- ・ その上下に余分な Reader 層を足し計算を実行
- ・ MTL、旧ExtEff、MoreEE と競合手法の HIA [KSS13] を比較
- ・ 環境：Intel Core i7 2.8GHz, 16GB RAM, GHC 7.10.3.
-threaded -02 -rtsopts

```
benchS ns = foldM f 1 ns
where
  f acc x | x `mod` 5 == 0 = do
    s ← S.get
    S.put $! (s+1)
    return $! max acc x
  f acc x = return $! max acc x

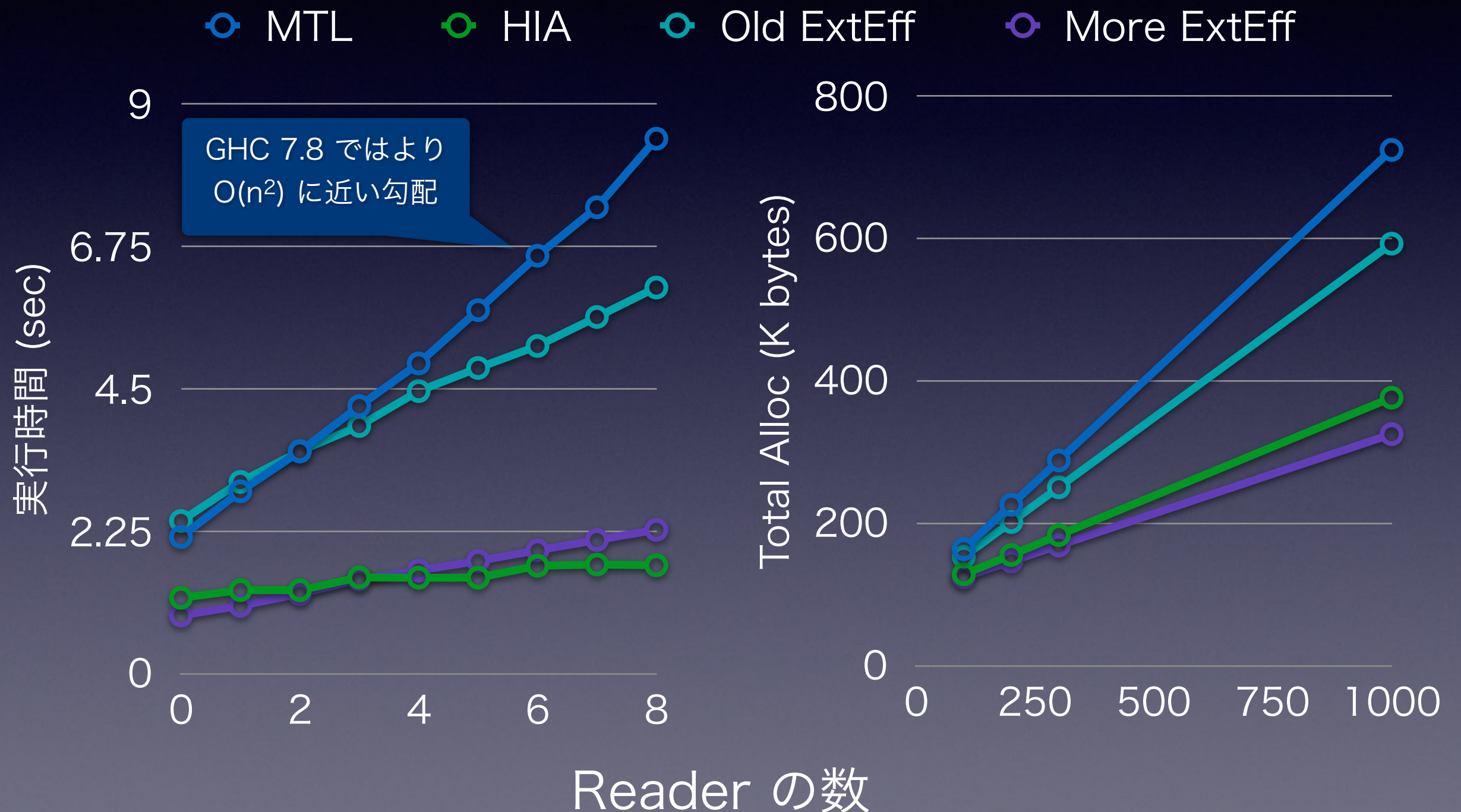
{-# NOINLINE benchS #-}
```

NOINLINE を付けて大きな計算と見做す
(GHC 7.8 までは付けなくてもMTLが圧倒的に悪かった)

Readers under State



Readers over State



補足と考察

- ・ 旧 Ext Eff から**大幅に改善**
- ・ Reader層を State 層に置き換えると、両方の場合についてNOINLINE 付きでも MTL が線型速度
- ・ 大きな計算の場合、時間・空間消費量ともに**More Ext Eff が最適**、競合手法とも互角
- ・ INLINE が効く小さい計算、あるいは単純な State / Reader 計算は最適化が効いて **MTL が勝つ**
 - ・ GHC の最適化機構の進化は凄まじい

Conclusions

Conclusions

- ・ More Extensible Effects は MTL の代替
 - ◎ **高効率**
 - ・ 時間・空間計算量が、競合手法の中でもかなり効率的
 - ・ 多数の副作用を合成する際に威力を発揮
 - ◎ **柔軟で高い表現力**
 - ・ 制約無しに副作用を合成可能
 - ・ 階層を跨いだ処理、複数の解釈を許容
 - ◎ **No More Lifts!**

参考文献

- [KI15] Kiselyov and ISHII, *Freer Monads, More Extensible Effects*. Haskell '15.
- [KLO13] Kammar, S. Lindley, and N. Oury, *Handlers in action*. ICFP '13.
- [KSS13] Kiselyov, Sabry and Swords, *Extensible Effects: An Alternative to Monad Transformers*. Haskell '13.
- [PK14] van der Ploeg and Kiselyov, *Reflection without Remorse: Revealing a hidden sequence to speed up monadic reflection*. Haskell '13.

Thank you!

Any Questions?

- ・ More Extensible Effects は MTL の代替
 - ◎ **高効率**
 - ・ 時間・空間計算量が、競合手法の中でもかなり効率的
 - ・ 多数の副作用を合成する際に威力を発揮
 - ◎ **柔軟で高い表現力**
 - ・ 制約無しに副作用を合成可能
 - ・ 階層を跨いだ処理、複数の解釈を許容
 - ◎ **No More Lifts!**