Haskellでの合成可能なオブジェクトの構成とその応用

木下郁章 [1], 山本和彦 [2]

fumiexcel@gmail.com

[2] IIJ イノベーションイスティテュート

kazu@iij.ad.jp

概 要 Haskell で状態を管理する際は、一般的に代数データ型や型クラスが用いられるが、データが拡張できないか、動的な性質を持たない。そのため Haskell は、複雑な状態を扱う問題領域には適していないと考えられてきた。一方で、一般的なオブジェクト指向言語では、オブジェクトを提供することでこの問題領域で成功を収めている。本論文では、Haskell の言語仕様を変更することなしに、オブジェクト指向言語から着想を得たオブジェクトを実現する。本論文で提案するオブジェクトは圏を構成し、合成を用いて継承を表現できる。また、終了する運命にあるオブジェクトやストリーミングなどに応用でき、複雑な状態を扱うゲームの実装にも使われている。

1 導入

Haskell[1] は純粋関数型言語であり、型システムが純粋な関数と副作用のある関数を分離することを要請するため、Haskell で書かれたコードにはバグが入り込みにくい。また、この純粋性のおかげで並行性と並列性を提供できる稀有な言語でもある [2]。さらに、副作用を含む手続きは専用の型を持っているだけでなく、自分で手続きを定義し、それらの間の変換を記述できる。このように Haskell には優れた特長がある一方で、複雑な状態を管理するような問題領域には適していないと考えられてきた。

一方で、オブジェクト指向プログラミング言語 (以下、OOPL) では、内部状態を隠蔽しインターフェイスを通して統一的に扱えるオブジェクトを提供している。オブジェクトは継承や複製などによりデータの拡張性を持ち、複雑で多様な状態を扱う GUI やゲームなどの領域において成功を収めている。

そこで本論文では、Haskell の言語仕様には手を加えずに、オブジェクトを表す手法を提案する。これまで Haskell に OOPL の機能を取り込むためのさまざまな研究がなされてきた。その主流は、OOPL で書かれたライブラリに対して外部関数呼び出しを可能にし活用する研究 [3, 4] と伝統的なオブジェクト指向プログラミング (以下、OOP) を模倣する研究 [5] に分類できる。この論文の目的はこれらの既存研究とは異なり、拡張性のある状態の管理手法を導入し、OOP に相当する記述力を Haskell で獲得することであり、OOPL が提供する機能を網羅することは目的とはしない。Haskell は静的型付きでコンパイル型の言語であるため、比較対象として Java、C++、C♯ といった静的型付き OOPL を念頭におく。

この論文は以下のように構成される。まず、第 2 章で状態を隠蔽可能なオブジェクトを導入し、第 3 章でオブジェクトの具体例とインスタンス化の方法を示す。オブジェクトの合成とオブジェクトの圏の構成について第 4 章で説明する。第 5 章と第 6 章では、それぞれ継承・オーバーライドに相当するオブジェクトの拡張方法と、応用について議論する。最後に、評価と関連研究、および結論をそれぞれ第 7 章と第 8 章で述べる。

2　オブジェクトの定義と利用

この章では、本稿で提案するオブジェクトの定義やその性質について説明する。なお、これから示すオブジェクトや関連する構造は、公開している `objective` パッケージ [1] に格納されている。

2.1　オブジェクトの定義

「オブジェクト」とは、内部状態とメソッドを持ち、メッセージを受け取るとメソッドが起動され、アクションを返すとともに、その内部状態が変化しうる概念を表す。「アクション」とは、`m a` のような型を持ち、プログラムや何らかの動作を表現する値である。オブジェクトが受け取るアクションを特に「メッセージ」と呼び、メッセージの型を「インターフェイス」と呼ぶ。「メソッド」とは、メッセージからアクションへの写像である。オブジェクトは、抽象度の高いメッセージ (アクション) を受け取り、抽象度のより低いアクションを返す。

我々は、以下のようにオブジェクトを二種類の型をパラメータとして持つようなデータとして定義する。この型の定義には、Haskell の Rank2Types 拡張が必要である [2]。

```
newtype Object f g = Object {
    runObject :: forall a. f a -> g (a, Object f g)
  }
```

オブジェクトの同値性は余帰納的に定義される。

定義 1 任意の同値性が判定できる型 `a` について `g a` の同値性が判定できる `Functor g` と、オブジェクト `a :: Object f g`、`b :: Object f g` があるとする。任意メッセージの `f :: f a` に対し、`fmap snd (runObject a f)` と `fmap snd (runObject b f)` が同値ならば `runObject a f` と `runObject b f` が同値になるとき、`a` と `b` は同値である。

提案したオブジェクトは、ミーリマシンの性質と自然性を併せ持つ。以下順に説明する。

2.2　ミーリマシンとの類似性

前述のように、オブジェクトはメッセージを受け取ると、その結果を返すとともに、自分自身の状態を変化させうる。したがって、メッセージを受け取り結果を返す、ある種のミーリマシンとして捉えられる。以下に Haskell のミーリマシンの表現の一つを示す。このミーリマシンの定義と提案手法の定義は、類似していることが分かる。

```
newtype Mealy a b = Mealy {
    runMealy :: a -> (b, Mealy a b)
  }
```

しかし、`Mealy` の定義では、メッセージと結果が一つの型しか持てないため、たとえば「文字列を返すメソッド」と「数値を返すメソッド」の両方を含むオブジェクトを表現できない。その問題を解決するには、インターフェイスは結果の型をパラメータとして持つ型であるべきであり、種`* -> *` を持つはずである。

2.3　自然性

インターフェイスを `M`、生成するアクションの型を `N`、結果の型を `a` とすれば、メソッドの呼び出しは以下のような型を持つ関数として表される。

```
forall a. M a -> N a
```

[1] http://hackage.haskell.org/package/objective
[2] https://ghc.haskell.org/trac/haskell-prime/wiki/Rank2Types

M と N が関手ならば、これはちょうど関手間の写像である自然変換に対応する。Haskell では、自然変換は以下の Natural 型として定義できる。

```
newtype Natural f g = Natural {
    runNatural :: forall a. f a -> g a
  }
```

任意の関手 M、N、型 a、値 m :: M a、任意の関数 f、自然変換 nat :: Natural M N について、以下の等式が成立し、これを自然性と呼ぶ。

```
runNatural nat (fmap f m) ≡ fmap f (runNatural nat m)
```

インターフェイスが関手である場合、オブジェクトの自然性を定義できる。任意のオブジェクト obj :: Object M N について以下の性質が成り立つ。

```
runObject obj (fmap f m) ≡ fmap (f *** id) (runObject obj m)
```

ただし、(***) は射のペアリングをする演算子で、Haskell の標準ライブラリでは Control.Arrow モジュールで定義されている。

```
(***) :: (a -> c) -> (b -> d) -> (a, b) -> (c, d)
f *** g = \(x, y) = (f x, g y)
```

2.4 変換

提案手法が、ミーリマシンと自然変換の両方の表現力を併せ持つことは、Mealy および Natural を Object に変換する関数 fromNatural の存在により、明確に示される。なお、以下の Req の定義には、GHC の GADTs 拡張を使用している。

```
fromNatural :: Functor g => Natural f g -> Object f g
fromNatural (Natural t) = liftO t

liftO :: Functor g => (forall x. f x -> g x) -> Object f g
liftO t = Object $ fmap (\x -> (x, liftO t)) . t

data Req a b r where
    Req :: a -> Req a b b

fromMealy :: Mealy a b -> Object (Req a b) Identity
fromMealy (Mealy t) = Object $ \(Req a) -> let (b, m) = t a in Identity (b, fromMealy m)
```

Req a b r は a と同型であり、GADT(Generalized Algebraic Data Types) の制約から結果として返される値は必ず b の型を持つ。また、任意の x について x は Identity x と同型だから、Object (Req a b) Identity は Mealy a b と同型であることが分かる。

3 オブジェクトの具体例

この章では、オブジェクトの具体例を挙げる。インターフェイスとして、メッセージ Print と Increment を持つ Counter 型を用いる。以下に定義を示す。

```
data Counter a where
    Print :: Counter ()
    Increment :: Counter Int
```

3.1 counter オブジェクトの実装と実行

Counter 型のメッセージを受け取るオブジェクト counter を以下のように実装する。counter に Increment を送ると内部状態が 1 増え、Print を送ると現在の内部状態を出力する。

```
counter :: Int -> Object Counter IO
counter n = Object (handle n) where
    handle :: Int -> Counter a -> IO (a, Object Counter IO)
    handle n Increment = return (n, counter (n + 1))
    handle n Print     = print n >> return ((), counter n)
```

この実装では、counter と handle が相互再帰しているが、この相互再帰は第 4 章で説明する合成により取り除ける。以下は、GHCi[3] を使って、runObject で counter のメソッドを呼び出す例である。

```
> let obj1 = counter 0
> runObject obj1 Print
0
> (_, obj2) <- runObject obj1 Increment
> (_, obj3) <- runObject obj2 Print
1
```

3.2 インスタンス化

提案したオブジェクトは、メッセージを受け取ると次のオブジェクトを返す。参照を用いれば、次々に生成されるオブジェクトを一つのインスタンスとして扱える。ここでは参照として MVar[7] を使う方法を示す。Haskell で広く使われている参照型として IORef があるが、複数のスレッドからメソッドを呼び出した際に競合する恐れがある。MVar を用いると、メッセージを送ってから結果が返るまで他のメッセージをブロックできるため、スレッドセーフなインスタンスを表現できる。

インスタンスに対しメッセージを送信し、メソッドを呼び出す演算子 (.-) を以下のように実装する。

```
(.-) :: MVar (Object t IO) -> t a -> IO a
m .- e = do
    obj <- takeMVar m
    (a, obj') <- runObject obj e
    putMVar m obj'
    return a
```

新しいインスタンスの作成には、newMVar が利用できる。これらの操作により、以下のようにして参照に基づいたインスタンス化と、メッセージの送信を表現できる。

```
do i <- newMVar (counter 0)
   i .- Print -- 0 を表示
   i .- Increment
   i .- Increment
   i .- Print -- 2 を表示
```

4 合成

第 2 章で述べた counter の実装では、counter と handle が相互再帰によって実装されていた。counter と handle を分離できれば、handle はメッセージの解釈に集中できるため、handle の再利用性も高まる。我々はオブジェクトの合成というアイディアで、この分離を実現する。

[3] https://www.haskell.org/ghc/

4.1 オブジェクトの合成

`p :: Object f g` と `q :: Object g h` が与えられたとき、`p` が生成するアクション `g` を `q` に解釈させるという操作によって、オブジェクトの合成 `p @>>@ q :: Object f h` を考えることができる。本論文では、この演算を合成と呼ぶ。合成演算子 `(@>>@)` の定義は、以下のように与えられる。

```
(@>>@) :: Functor h => Object f g -> Object g h -> Object f h
Object m @>>@ Object n = Object $ fmap joinO . n . m

joinO :: Functor h => ((a, Object f g), Object g h) -> (a, Object f h)
joinO ((x, a), b) = (x, a @>>@ b)
```

合成の単位元は、以下の `echo` のような、受け取ったメッセージをそのまま返すようなオブジェクトとなる (付録 A を参照のこと)。

```
echo :: Functor f => Object f f
echo = Object (fmap (\x -> (x, echo)))
```

4.2 合成の例

`(@>>@)` と用いて、`counter` と `handle` を分離する。具体的には、`transformers` パッケージ [9] で提供されているモナド変換子 `StateT` を用い、`counter` を `StateT Int IO` で明示的に状態を扱うよう定義し直す。このアイディアの骨子を以下に示す。

```
counter :: Int -> Object Counter IO
counter n = counterE @>>@ variable n

counterE :: Object Counter (StateT Int IO)

variable :: Int -> Object (StateT Int IO) IO
```

インターフェイスとして `StateT s m` を持つオブジェクトは、メッセージとして `get :: Monad m => StateT s m` と、`put :: Monad m => s -> StateT s m ()` を受け取れる。典型的な OOP の言葉では、公開フィールド `s` を持つオブジェクトとみなせる。

そして、`StateT s m` のアクションを返すオブジェクトは、状態 `s` に依存した動作を表す。どちらも状態が明示的に型に表れているため、操作を拡張できる一方、カプセル化されていない。両者を合成することによって、状態 `s` は隠蔽され、カプセル化が実現される。

4.3 状態を保持するオブジェクト

状態を保持するオブジェクトである `variable` は、以下のように実装できる。

```
variable :: Monad m => s -> Object (StateT s m) m
variable s = Object $ \m -> runStateT m s >>=
                      \(a, s') -> return (a, variable s')
```

4.4 状態を利用するオブジェクト

`handle` 自体は状態を持っていないため、単にアクションの自然変換として記述される。第 2.4 節で示した `fromNatural` を用いれば、自然変換は自明にオブジェクトに変換できる。

```
counterE :: Object Counter (StateT Int IO)
counterE = fromNatural (Natural handle)

handle :: Counter a -> StateT Int IO a
handle Increment = do { n <- get; put (n + 1); return n }
handle Print     = get >>= \n -> lift (print n)
```

4.5 オブジェクトの圏

オブジェクトは変換元と変換先の二つのパラメータを持ち、オブジェクト同士を合成できることから、圏における射のような性質を持つことが類推される。実際に、アクションを対象、オブジェクトを射とすると圏をなす。結合則と単位元の存在は、等式変換により証明できる (付録 A を参照のこと)。

- アクション `f`、`g` に対し、オブジェクト `Object f g` がある。
- 任意のアクション `f`、`g`、`h` に対し、`(@>>@) :: Object f g -> Object g h -> Object f h` なる合成関数が存在する。
- 任意のアクション `f` に対し、アクションをおうむ返しするオブジェクト `echo :: Object f f` が存在する。
- 合成は結合的である
- `echo` は合成の左単位元かつ右単位元である。

アクションとオブジェクトの圏を考えることにより、典型的な OOP の枠組みを超え、広範に性質を議論できる。

インターフェイスを M、オブジェクトの起こすアクションを IO とすると、オブジェクトは M から IO への個々の射に対応し、オブジェクトの型はドメインとコドメインによってのみ決定される。そのため、Java、 C♯、C++のような一般的なクラスベースの静的型付き OOPL におけるクラスと違い、型に実装を対応させるものではない。

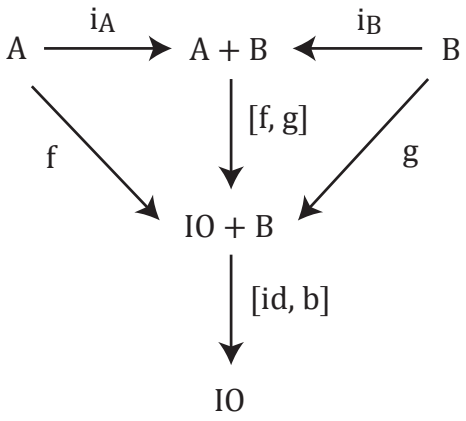
「$M \to IO$ は $N \to IO$ である」(is-a 関係) は、N から M への単射が存在、つまり N における任意のメッセージは、M に属するメッセージとして扱えることを意味する。この単射の存在から、$Hom(M, IO)$ に属するオブジェクトを、$Hom(N, IO)$ に属するオブジェクトに対応させられることを示しており、射の合成 $Hom(N, M) \times Hom(M, IO) \to Hom(N, IO)$ の特殊な場合として考えられる。

5 オブジェクトの拡張

オブジェクトの圏を用いれば、オブジェクトを拡張する典型的な手法である継承およびオーバーライドの仕組みも、単純な式によって表される。ここで、拡張する元のオブジェクトを $b : B \to IO$ とおくと、継承とオーバーライドは、以下のように表現できる。

継承 あるオブジェクトを継承した子は、親が受け取る B に加え、新たな種類のメッセージ A を扱える。したがって、メッセージの和 $A + B$ を受け取るオブジェクト $A + B \to IO$ と捉えられる。新たなメソッド A の実装は、親のインターフェイス B と親の文脈 IO を使えるべきである。アクションを組み合わせる方法として直和を用い、オブジェクト $f : A \to IO + B$ として定義する。

オーバライド メソッドのオーバーライドを、$g : B \to IO + B$ なるオブジェクトによって表す。何もオーバーライドしない場合として、自明な射 $i_B : B \to IO + B$ が存在する。



上の図式において、射 $[id, b] \circ [f, g]$ が、オーバーライドを伴って継承されたオブジェクトになる。

5.1 メッセージの和の実装

Haskell において、アクション同士の和は以下のように定義される。

```
data Sum f g a = InL (f a) | InR (g a)
```

オブジェクト f、g に対する余ペアリング $[f, g]$ は、以下のように実装できる。

```
(@||@) :: Functor m => Object f m -> Object g m -> Object (Sum f g) m
a @||@ b = Object $ \r -> case r of
  InL f -> fmap (fmap (@||@b)) (runObject a f)
  InR g -> fmap (fmap (a@||@)) (runObject b g)
```

しかし、たとえばメッセージセレクタを並べた (G)ADT や、Sum はモナドではないため、オブジェクトはメッセージを一度きりしか受け付けられない。モナドであれば、メッセージそのものが合成可能になるため、Smalltalk のカスケーディングのように複数のメッセージを送ることが可能になる。

かといって、メッセージを定義するたびにモナドのインスタンスも定義すると、コードの重複が発生するため非効率的である。

任意の`* -> *`の種を持つデータ型にモナドのインスタンスを対応させることができる手法として、Operational モナド [10] がある。Operational モナドの定義は以下の通りである。

```
data Program t a where
    Return :: a -> Program t a
    Bind :: t a -> (a -> Program t b) -> Program t b

instance Monad (Program t) where
    return = Return
    Return a >>= k = k a
    Bind t c >>= k = Bind t ((>>= k) . c)

liftP :: t a -> Program t a
liftP t = Bind t Return
```

`Program` は、任意のインターフェイスを、モナドとして合成可能なインターフェイスに拡張する。`m` がモナドならば、オブジェクト `Object t m` は、`Program t` を受け取るように変換できる。

```
sequential :: Monad m => Object t m -> Object (Program t) m
sequential r = Object $ liftM (fmap sequential) . inv r where
    inv obj (Return a) = return (a, obj)
    inv obj (Bind e cont) = runObject obj e >>= \(a, obj') -> inv obj' (cont a)
```

Program A、Program B からの射として liftL = liftP . InL、liftR = liftP . InR が存在する。また、以下に示す余ペアリングの存在から、Program (Sum A B) は和 Sum A B と同様の性質を持つことが分かる。

```
copair :: (Monad m)
  => Object (Program s) m -> Object (Program t) m
  -> Object (Program (Sum s t)) m
copair a0 b0 = sequential (go a0 b0) where
  go a b = Object $ \r -> case r of
    InL f -> fmap (\a' -> go a' b) `liftM` runObject a (liftP f)
    InR g -> fmap (\b' -> go a b') `liftM` runObject b (liftP g)
```

したがって、オブジェクトの圏においては、必要に応じてメッセージの和 Sum A B の代わりに Program (Sum A B) を使っても齟齬は生じない。

5.2 デザインパターン

OOP のデザインパターンを提案手法でどのように実現するかの例を示す。Adapter、Proxy、Decorator パターンは、既存のインターフェイスを変更し、新たなインターフェイスを持つオブジェクトを作るためのデザインパターンである。

特に Proxy パターンとして考えられる例として、第 2 章で定義した counter を拡張する例を紹介する。Print は、呼ばれた回数をカウントするとともに、元のオブジェクトの Print を呼ぶ。ただし、5 回呼ばれたら “Limit exceeded” と表示し、それ以降元のオブジェクトの Print を呼ばなくなる。

```
wrapper :: Int -> Object Counter (Program (Sum Counter IO))
wrapper n = Object $ \r -> case r of
    Print
      | n < 5     -> liftL Print >> return ((), wrapper (n + 1))
      | otherwise -> liftR (putStrLn "Limit exceeded") >> return ((), wrapper n)
    Increment     -> liftL Increment >>= \x -> return (x, wrapper n)

counter' :: Object Counter IO
counter' = wrapper 0 @>>@ sequential (counter @||@ echo)
```

一方、Template Method パターンは、アルゴリズムを抽象化し、具体的な実装をサブクラスとして実装するデザインパターンである。本稿で提案したオブジェクトの表現では、Adapter、Proxy、Decorator、Template Method に本質的な差異はなく、オブジェクトの合成として統一的に表すことができる。

6 応用

この章では、本稿におけるオブジェクトを定命のオブジェクトおよびストリームへ応用することについて説明する。

6.1 定命のオブジェクト

オブジェクトが永続せず、途中で終了するというケースは頻繁にある。たとえば、アクションゲームならば、倒した敵はフィールドから消えるなどの仕組みが必要である。典型的な OOP の場合、終了したと分かったオブジェクトへの参照を削除し、ガベージコレクションを待つという方法が使われている。しかし、すべての参照を削除する操作は自明ではないうえ、「終了した」とプログラム上で判定してもメソッドを呼ぶことは可能なため、予期せぬバグやメモリリークを引き起こす可能性がある。

本稿で提案したオブジェクトの表現を使うと、オブジェクト自身が自発的に終了する場合も扱うことができる。`Object f Maybe` はメッセージを受け取ったとき、結果と次の状態が生成されない可能性があるオブジェクトを表現する。また、`Object f (Either a)` は最終結果 `a` を残して終了するオブジェクトを意味する。特に、後者は `newtype` を用いて異なる型 (`Mortal` と呼ぶ) にすれば、モナドのインスタンスにすることができる。本論文では、このような途中で終了する運命にあるオブジェクトを、「定命のオブジェクト」と呼ぶ[4]。

```
newtype Mortal f a = Mortal { unMortal :: Object f (Either a) }

mortal :: (forall x. f x -> Either a (x, Mortal f a)) -> Mortal f a
mortal f = Mortal $ Object (fmap (fmap unMortal) . f)

runMortal :: Mortal f a -> f x -> Either a (x, Mortal f a)
runMortal m = fmap (fmap Mortal) . runObject (unMortal m)

instance Monad (Mortal f) where
  return a = mortal $ const $ Left a
  m >>= k = mortal $ \f -> case runMortal m f of
    Left a -> runMortal (k a) f
    Right (x, m') -> return (x, m' >>= k)
```

`Mortal` モナドにおける `return a` は、メッセージを一切受け取らず結果 `a` を残して停止するオブジェクトを表す。一方、`m >>= k` は、`m` が終了したとき最終結果を `k` に渡し、得られた新しいオブジェクトで引き継ぐ。オブジェクトの引き継ぎは、ゲームの実装において、「敵が死亡する際に自爆する」「プレイヤーが死亡したとき交代する」などの動作の表現に対し柔軟な表現を与える。

`Either` の代わりに、`EitherT`[5] を使うと、`Mortal` をモナド変換子として定義できる。これは次節で利用する。

参照は自発的に消滅できないため、定命のオブジェクトは、参照を用いたインスタンスではうまく表現できない点に注意する必要がある。定命のオブジェクトは、リストなどのコンテナに直接格納するのに適している。以下の `announce` 関数は、あるリストに格納されているすべての `Mortal` に対しメソッドを呼び出し、その結果を取得する。

```
announce :: Monad m => f a -> StateT [Mortal f r] m ([a], [r])
announce f = StateT $ return . h . fmap unzip . partitionEithers
                      . map (`runMortal` f)
  where h (a, (b, c)) = ((a, b), c)
```

`announce` は、`[Mortal f r]` を格納する `variable` オブジェクトへメッセージとして渡すことができる。ゲームにおいて敵キャラクターなどの集まりを管理するとき、すべてに対しメソッドを呼び出し、消滅したキャラクターを削除するという処理はこの `announce` 一つによって実現される。

6.2 ストリームの表現

オブジェクトは、ストリームの生産者や消費者の役割を果たすこともできる。オブジェクトの合成を用いて、値を返すメソッドを持つオブジェクトは生産者として、また値を受け取るメソッドを持つオブジェクトは消費者として扱える。そのため、ストリームとして機能させたい概念に対し、共通の API を提供できる。

オブジェクトが受け取るアクションの型が `(->) a` のとき、そのオブジェクトは生産者として機能する。オブジェクト `Object ((->) a) m` にメッセージ `f :: a -> b` を渡すと、結果 `b` が返され

[4] 我々が知る限り mortal に対する一般的な日本語の訳が存在しないため、仏教における「定められた寿命」を意味する語を借用する。

[5] http://hackage.haskell.org/package/either

る。つまり、オブジェクトは f に型 a の値を渡す。たとえば、整数を順番に生成していくオブジェクトは以下のように表される。

```
genNaturals :: Monad m => Int -> Object ((->) Int) m
genNaturals n = Object $ \f -> return (f n, genNaturals (n + 1))
```

逆に、受け取るアクションが (,) a のときは、消費者となる。オブジェクト Object ((,) a) m にメッセージ (a, x) を渡すと結果 x がそのまま返るが、オブジェクトは a を利用できる。そして、生産者と消費者とを結合できる。結合演算子 ($$) の実装を以下に示す。

```
($$) :: (Monad m) => Object ((->) a) m -> Object ((,) a) m -> m x
a $$ b = do
  (x, a') <- runObject a id
  ((), b') <- runObject b (x, ())
  a' $$ b'
```

(->) a や (,) a を受け取る定命のオブジェクトは、有限のストリームを表現する。定命のオブジェクトでも ($$) は利用可能である。EitherT を返すオブジェクトについて、($$) は以下のような型を持ちうる。

```
($$) :: (Monad m) => Object ((->) a) (EitherT a m)
  -> Object ((,) a) (EitherT a m)
  -> EitherT a m Void
```

EitherT a m Void は、eitherT return absurd の適用によって m a に変換される。ただし、Void は値が存在しないデータ型[6] で、absurd :: Void -> a は「Void の値が存在する」という矛盾から任意の値の存在を示す関数である。

定命のオブジェクトと不死のオブジェクトを結合したい場合、不死の方を定命の形に変換する必要がある。EitherT の場合、liftO lift :: Object m (EitherT a m) を後ろに合成することで、定命の型になる。

以下は、ファイルを一行ずつ読み込み、終端に達したらファイルを閉じて終了する定命オブジェクト lineReader と、文字列を一行ずつ書き出すオブジェクト lineWriter を定義する例である。main は両者を結合し、ファイル foo.txt を標準出力に書き出す。

```
import Control.Monad.Trans.Either
import Control.Monad.Trans
import System.IO
import Data.Void

lineReader :: FilePath -> IO (Object ((->) String) (EitherT () IO))
lineReader path = fmap go $ openFile path ReadMode
  where
    go h = liftO $ \cont -> lift (hIsEOF h) >>= \r -> if r
        then lift (hClose h) >> left ()
        else lift $ fmap cont $ hGetLine h

lineWriter :: Object ((,) String) IO
lineWriter = liftO $ \(s, x) -> putStrLn s >> return x

main = do
  r <- lineReader "foo.txt"
  eitherT return absurd $ r $$ (lineWriter @>>@ liftO lift)
```

[6] http://hackage.haskell.org/package/void

7　評価と関連研究

Java、C++、C♯が提供するオブジェクトは、抽象データ型からの派生であり、オブジェクトにシンボルを与えることで、メソッドを起動する。一方、提案するオブジェクトはメッセージ指向に基づいており、オブジェクトがメッセージを受け取るとメソッドを実行する。メッセージはオブジェクトとは独立に定義され、ファーストクラスの値である。

7.1　性能

性能の観点から提案したオブジェクトの実用性を確かめるため、音楽ゲームを作成した。この音楽ゲームでは、複数の音声データをオブジェクトで表現しており、1 秒に数万回の頻度で状態が更新される。また、1000 個以上のオブジェクトを扱うようなゲームの記述にも利用している。現在の一般的なコンピュータでは、両者とも問題なく動作する。

7.2　関連研究

関連した研究として前述の OOHaskell がある。OOHaskell では、メソッドの数々を型レベルでパラメータ化されたリストに格納することによってオブジェクトを表現している。しかし、フィールドやメソッド名を表すラベルは名前以外の型情報を持っていないため、単純なラベルの集まりとして型の付いたインターフェイスを表現できない。本稿で提案したオブジェクトは、名前ではなく手続きというインターフェイスと、それらの変換に着目した構造を持っているため、型が明確なだけでなく、合成などのさまざまな演算とその性質が導き出される。

OOHaskell では各フィールドの可変な状態を扱うのに参照 (`IORef`) を使うが、提案手法の表現では再帰的な構造を使い、状態をオブジェクトそのものに状態を埋め込んでいることも本質的に異なる。オブジェクトを扱う際に IO を要求しないため、ゲームのキャラクターなど、抽象的かつ複雑な状態を持つ概念の記述に適している。

7.3　Extensible effects

Extensible effects[12] が提案するような拡張可能なアクションは、アクション間の変換である提案手法と親和性を持つ。Extensible effects では、作用の種類の型レベルリストによってパラメータ化されたモナド、`Eff` によって作用を記述している。たとえば、文脈から値を取り出す `ask` と、文脈に値を与える `runReader` は、以下のような型を持っている。

```
ask :: (Typeable e, Member (Reader e) r) => Eff r e
runReader :: Typeable e => Eff (Reader e :> r) w -> e -> Eff r w
```

オブジェクトは自然変換を内包しているため、以下のように表すことができる。

```
runReader' :: Typeable e => e -> Object (Eff (Reader e :> r)) (Eff r)
```

`Foo`、`Bar`、`Baz` と、それを解釈する `objFoo`、`objBar`、`objBaz` を定義した場合を考える。ただし、`Foo` は `Baz` に依存している。これらはオブジェクトであるため、状態を持つことができる。

```
runFoo :: Member Baz r => Object (Eff (Foo :> r)) (Eff r)
runBar :: Object (Eff (Bar :> r)) (Eff r)
runBaz :: Object (Eff (Baz :> r)) (Eff r)
```

オブジェクトの合成により、Foo、Bar、Baz の機能を併せ持ったオブジェクトを構築できる。

```
runFoo @>>@ runBar @>>@ runBaz
  :: Object (Eff (Foo :> Bar :> Baz :> r)) (Eff r)
```

オブジェクトの圏においては、`Eff` モナドは和と同様の性質を持つ。Extensible effects の手法を用いることで、第 5 章のようなオブジェクトの拡張をより簡易に実現できる。

7.4 表現問題

表現問題 (expression problem)[7] の観点から、本論文で導入したオブジェクトを他の手法と比較する。表現問題とは、静的型付き言語において、新しいデータの追加と、データに対する新しい操作の追加が両立するような抽象化が難しいという問題である [11]。つまり、以下の二つの項目を、既存のコードを再コンパイルすることなく、型の安全性を保ったまま実現することは難しいと考えられている。

操作の拡張性 既存のデータ型に対して操作を追加できる。

データの拡張性 既存の型に属する新しいデータを追加できる。

さらに、本論文では新たな評価項目として、「振る舞いの動的性」を定義する。振る舞いの動的性を持つ場合、内部構造がどのように変化しても、インターフェイスが同一であれば継ぎ目なしに扱える。

例としてエルフ (Elf) とオーク (Orc) という対象を考える。エルフは、悪の呪文を受けると正気度が減り、正気度が 0 になるとオークとなる。オークは、悪の呪文を受けてもオークのままであり、内部状態を持たない。この例題において比較項目の具体例を示す。

操作の拡張性 たとえば「治療を受ける」という操作を追加できるか？

データの拡張性 ドワーフ (Dwarf) という新しい対象を追加できるか？

振る舞いの動的性 エルフはオークに変化できるか？

7.4.1 クラスベースの OOPL

一般的なクラスベースの静的型付き OOPL では、データの拡張性はあるクラスのサブクラスを増やすことで実現できる。一方で、操作はクラスの定義に結び付けられているため、操作の拡張は難しい。エルフが自身を返す代わりに、新たにオークのインスタンスを生成するようなメソッドでも、同一のインターフェイスを持つため、振る舞いの動的性を持っている。

7.4.2 代数的データ型

Haskell のような関数型言語では、`Entity` のような直和を定義し、エルフとオークを同一の型に格納する方法が一般的である。その実装の骨子を示す。

```
data Entity = Elf Int | Orc

spell :: Entity -> Entity
```

この実装では、型が同じだからエルフがオークになれるため、振る舞いの動的性を持っている。また、`spell` 以外の操作関数を追加するのは容易である。しかし、新たなデータ型を追加するには、`Entity` や操作関数 `spell` を変更する必要がある。

7.4.3 型クラス

Haskell では、データの拡張性を実現するために型クラスが使われることがある。以下に型クラスを用いた実装の骨子を示す。

```
data Elf = Elf Int
data Orc = Orc

class Spell s where
    spell :: s -> s
```

[7] http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

```
instance Spell Elf where
    spell = ... -- Orc を書けない

instance Spell Orc where
    spell = ...
```

この方法で定義された場合、存在量化を用いて、Elf と Orc を同一の型にまとめることができる。

```
data Entity = forall s. Spell s => Entity s
```

型クラスを用いると、新たなデータ型を追加するのは容易である。操作の追加も、別の型クラスを実装すればよい。しかし、エルフとオークは別の型であり、spell の型は別の型への変化を表現できないため、エルフはオークになれず、振る舞いの動的性を持たない。

逆に、以下のように操作をデータ型として定義し、型クラスとして内部状態への操作を表現する方法もある [8]。

```
newtype Spell a = Spell { spell :: a -> a }

class Action f where
    elf :: Elf -> f Elf
    orc :: Orc -> f Orc
```

こちらもデータと操作の拡張性を持つが、Spell の型が内部状態の型によって決まることから、エルフがオークになる場合を表現できず、こちらの場合も振る舞いの動的性を持たない。

7.4.4 提案手法

本稿で提案したオブジェクトを用いると、以下のように、呪文を表すメッセージ Spell と、それを受け取るオブジェクトである、elf と orc が定義される。

```
data Spell x where
    Spell :: Spell ()

type Entity = Object Spell IO

elf :: Int -> Entity
elf 0 = Object $ \Spell -> return ((), orc)
elf n = Object $ \Spell -> return ((), elf (n-1))

orc :: Entity
orc = Object $ \Spell -> return ((), orc)

spell :: [Entity] -> IO [Entity]
spell = mapM update where
    update = fmap snd . flip runObject Spell
```

オブジェクトは単なる値であるため、データの拡張性を持っている。そのため、dwarf のような新しいオブジェクトを定義することも容易である。

```
dwarf :: Int -> Entity
dwarf 0 = Object $ \Spell -> return ((), orc)
dwarf n = Object $ \Spell -> return ((), dwarf (n-1))
```

また、振る舞いの動的性を持っており、return ((), orc) という式が示すように、エルフやドワーフがオークに変化できることが分かる。オブジェクトは、値レベルで振る舞いを組み立てることが可能なため、以下のような型クラスを定義すれば、操作の拡張性も持たせることができる。

```
class Elf t where
    elf :: Object t IO

instance (Elf s, Elf t) => Elf (Sum s t) where
    elf = elf @||@ elf

instance Elf Spell where
    elf = ⋯

class Orc t where
    orc :: Object t IO

instance (Orc s, Orc t) => Elf (Sum s t) where
    orc = orc @||@ orc

instance Orc Spell where
    orc = ⋯
```

elf に対し Cure という操作を追加したい場合、Elf Cure のインスタンスを定義すればよい。すると、(Elf s, Elf t) => Elf (Sum s t) のインスタンスから、二種類の操作を受け付けるオブジェクト、elf :: Object (Sum Spell Cure) IO が自動的に得られる。

ここまでで紹介した評価を表にまとめる。

	操作の拡張性	データの拡張性	振る舞いの動的性
クラスベース OOP		✓	✓
代数データ型	✓		✓
型クラス	✓	✓	
提案手法	✓	✓	✓

8 結論

本論文では、Haskell で状態を柔軟に取り扱うための新たな手法であるオブジェクトを導入し、その有用性を示すとともに、その応用や代数的性質を議論した。複雑な状態を管理するのに適した OOP の手法は、データの拡張性と振る舞いの動的性を備えていることを指摘し、また Haskell で状態を扱う伝統的な手法である代数データ型と型クラスには、それらを同時に有していないことも示した。本稿で導入したオブジェクトは、表現問題を解決するだけでなく、振る舞いの動的性も備えている。このオブジェクトは、合成によって振る舞いを拡張または隠蔽していくことが可能であり、オブジェクトの部品化や、デザインパターンの表現の基礎となる。本稿で提案したオブジェクトは圏を構成するので、圏の性質を元に構造を考察できる。さらに、定命のオブジェクトは、既存の OOP ではうまく表現できなかった問題に対して解決方法を与えた。

謝辞

研究の内容に対して有益なコメントをいただいた Oleg Kiselyov 氏と小田朋宏氏に感謝する。また、本論文の構成に的確なアドバイスを下さった三名の匿名査読者の方々に深謝する。木下は、IIJ イノベーションイスティテュートのアルバイトとしてこの研究を進めた。

参考文献

[1] Simon Marlow et al, “Haskell 2010 Language Report”, 2010 年.

[2] Simon Marlow, “Parallel and Concurrent Programming in Haskell”, O’Reilly, 2013 年.

[3] Mark Shields and Simon Peyton Jones, “Object-Oriented Style Overloading for Haskell”, In the Workshop on Multi-Language Infrastructure and Interoperability (BABEL’01), 2001 年.

[4] Andr T. H. Pang and Manuel M. T. Chakravarty, “Interfacing Haskell with Object-Oriented Languages”, Implementation of Functional Languages, Lecture Notes in Computer Science Volume 3145, pp 20-35, 2005 年.

[5] Oleg Kiselyov and Ralf Laemmel, “Haskell’s overlooked object system”, http://arxiv.org/pdf/cs/0509027.pdf, 2005 年.

[6] Jeremy Gibbons and Oege de Moor, “The Fun of Programming”, Palgrave, p 203, 2003 年.

[7] Simon Peyton Jones, Andrew D. Gordon and Sigbjorn Finne, “Concurrent Haskell”, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PoPL), 1996 年.

[8] B. C. d. S., Oliveira, Ralf Hinze, Andres Löh, “Extensible and Modular Generics for the Masses”, Trends in Functional Programming, 2006 年.

[9] Mark P Jones, “Functional Programming with Overloading and Higher-Order Polymorphism”, Advanced School of Functional Programming, 1995 年.

[10] Heinrich Apfelmus, “The Operational Monad Tutorial”, The Monad.Reader Issue 15, http://themonadreader.files.wordpress.com/2010/01/issue15.pdf, 2010 年.

[11] Philip Wadler, “The Expression Problem”, Java Genericity mailing list, http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, 1998 年.

[12] Oleg Kiselyov et al, “Extensible Effects”, Proceedings of the 2013 ACM SIGPLAN, http://okmij.org/ftp/Haskell/extensible/exteff.pdf, 2013 年.

A 付録

可読性のため、Object と runObject をそれぞれ `inO`、`outO` と略記する。

A.1 合成の結合則

定理 1 任意のアクション `e`、`f`、`g`、関手 `h`、およびオブジェクト `(a :: Object e f)`、`(b :: Object f g)`、`(c :: Object g h)` について、結合則 `a @>>@ (b @>>@ c) = (a @>>@ b) @>>@ c` が成り立つ。

証明 1 等式変換により証明する。

```
outO (a @>>@ (b @>>@ c))
= {  (@>>@) の定義 }
fmap joinO . fmap joinO . outO c . outO b . outO a) f
= { fmap fusion }
fmap (joinO . joinO) . outO c . outO b . outO a
= { (joinO . joinO) の展開 }
= fmap (\(((x, ef), fg), gh) -> (x, ef @>>@ (fg @>>@ gh)))
        . outO c . outO b . outO a

outO ((a @>>@ b) @>>@ c)
= { (@>>@) の定義 }
fmap joinO . outO c . (fmap joinO . outO b . outO a) f
= { outO . inO = id }
fmap joinO . outO c . fmap joinO . outO b . outO a
= { 自然性 }
fmap joinO . fmap (first joinO) . outO c . outO b . outO a
= { fmap fusion }
fmap (joinO . first joinO) . outO c . outO b . outO a
```

```
= { joinO . first joinO の展開 }
= fmap (\(((x, ef), fg), gh) -> (x, (ef @>>@ fg) @>>@ gh))
        . outO c . outO b . outO a
= { 余帰納法 }
= fmap (\(((x, ef), fg), gh) -> (x, ef @>>@ (fg @>>@ gh)))
        . outO c . outO b . outO a
= { 左辺 }
outO (a @>>@ (b @>>@ c))
```

A.2 左単位元

定理 2 任意のオブジェクト `obj :: Object f g` について、`echo @>>@ obj = obj` である。

証明 2 等式変換により証明する。

```
outO (echo @>>@ obj)
= { echo の定義 }
fmap (\x -> (x, echo))) @>>@ obj
= { (@>>@) の定義 }
fmap joinO . outO obj . fmap (\x -> (x, echo))
= {  自然性 }
fmap joinO . fmap (first (\x -> (x, echo))) . outO obj
= { fmap fusion }
fmap (joinO . first (\x -> (x, echo))) . outO obj
= { joinO の定義 }
fmap (\(x, m) -> (x, echo @>>@ m)) . outO obj
= { 余帰納法 }
fmap (\(x, m) -> (x, m)) . outO obj
= { fmap id = id }
outO obj
```

A.3 右単位元

定理 3 任意のオブジェクト `obj :: Object f g` について、`obj @>>@ echo = obj` である。

証明 3 等式変換により証明する。

```
outO (obj @>>@ echo)
= { echo の定義 }
outO (obj @>>@ Object (fmap (\x -> (x, echo))))
= { (@>>@) の定義 }
fmap joinO . fmap (\x -> (x, echo)) . outO obj
= { 自然性 }
fmap (joinO . (\x -> (x, echo))) . outO obj
= { fmap fusion }
fmap (\(x, m) -> (x, m @>>@ echo)) . outO obj
= { 余帰納法 }
fmap (\(x, m) -> (x, m)) . outO obj
= { fmap id = id }
outO obj
```