Object as a Morphism
A Composable Structure Parameterized by Effects

Fumiaki Kinoshita
fumiexcel@gmail.com

Kazuhiro Yamamoto
IIJ Innovation Institute Inc.
kazu@iij.ad.jp

Abstract
To deal with states in a unified type, sum types or existential quantification are widely used in Haskell. However, they are intractable because they are not extensible in either data or operations. For this reason, Haskell has been considered inappropriate for domains involving complex states. On the other hand, object-oriented programming languages provide objects with subtyping. The convenience of polymorphism in a unified type makes them successful in those domains. This paper presents an encoding of objects within the system of Haskell with several common language extensions. These objects form a category; the composition of objects subsumes a considerable number of object-oriented notions, including inheritance, overriding, and Adapter and Template Method patterns. Distinctively, our method can be used to express the death of objects safely.

General Terms
Languages

Keywords
Objects, Composition, Category Theory

1. Introduction
Haskell [10] is a purely functional programming language which provides remarkable features. The type system distinguishes between pure and impure code, preventing unexpected bugs that come from side effects. Also, it has affinity for concurrency and parallelism [8]. Not only actions (i.e., effectful values) have their own types, but we can also define new types, or transformation between actions.

Even with such excellent features, we consider Haskell awkward to deal with dynamic, complex states of masses. For instance, in game programming, we often need to abstract the internal states of various entities (e.g., characters) in order to store them in a container. Typical approaches to deal with such states are sum types (in algebraic data types) whose inhabitants cannot be extended and existential quantification which spoils type inference and needs boilerplate wrapper per typeclass.

On the other hand, among statically typed programming languages, object-oriented programming languages (OOPL) such as C++, Java, C♯ and Objective C, made a success on dealing with stateful masses. Their key technology is the object system: objects encapsulate their manipulatable states by their operations and their inhabitants are extensible by inheritance. The OOPLs do not provide extensibility of operations of individual objects.

To handle masses of states in Haskell, this paper introduces composable objects whose operations are also extensible. Various pieces of research has been conducted to introduce object-oriented functionality to Haskell. The pieces of research are roughly divided to two streams: one is to use existing libraries on OOPLs [12, 13] and the other is to imitate object-oriented programming (OOP). The famous example of the latter is OO Haskell [6]. Unlike these existing research, our goal is to introduce a novel extensible structure to manage states without proposing any additional features for Haskell.

Our contributions are:

- A lightweight, extensible encoding of objects parameterized by a context and an interface. It is compatible with existing state manipulation approaches, so developers can use this approach without intense modification of existing code.
- Composition of objects and categorical interpretation that generalize inheritance and several design patterns. Our representation allows objects to be composable as functions and messages to be monadic. In traditional OOP implementation, neither objects nor messages are composable in this sense.
- A distinctive solution to the problem of mortal objects in real programs which is monadically composable.

The remainder of this paper is structured as follows. Section 2 presents a purely functional representation of objects, a few examples and instantiation of objects. Section 3 provides composition operations for objects and shows that objects form a category as morphisms. We describe how to extend our objects in Section 4. In Section 5, we discuss the application of our object, including a solution to the issue around death of objects. Section 6 explains technical characteristics of our objects. We compare our approach with others in Section 7 and 8 then state conclusion in Section 9.

The implementation of our idea is available as the objective package on hackage1. This package is actually used to implement games2.

2. The Final Encoding of Objects
We define an object as a notion which has an internal state and methods which would change the state on reception of corresponding messages. We encode objects as a data type which has two type parameters (note that the definition requires the Rank2Types extension):

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

The parameter t and g indicates the interface and context of the object, respectively. Interface is a type of messages while context is a type of methods. messages are actions that a object receives. Methods are also actions that the object generates. An action is a value to express an effect whose data type is a where a expresses interface or context and a represents the type of a result.

1 http://hackage.haskell.org/package/objective
2 One example is found in http://fumieval.github.io/rhythm-game-tutorial/
Because of the nature of corecursive structures, an object may have a state implicitly, without any references. An `Object` can be understood as a stateful translator from messages to methods. `runObject` is a function that accepts a message and returns a result and a succeeding object, translating \( f \) to \( g \). Effects are produced by the conversion from messages to methods.

Virtual function table (vtable) is a mechanism to dispatch methods widely used in implementations of OOPLs. Methods need to be stored discretely; in other words, the vtable-based approach is initially encoded. On the other hand, `Object` is finally encoded. Our representation of an object has just a function to dispatch methods, abstracting tables away. This function-based encoding has several advantages:

**Simplicity** No extra machinery is needed to implement the function-based encoding in Haskell.

**Generality** The vtable-based encoding requires that methods are enumerable. The function-based approach is more permissive in that it allows infinite behavior.

### 2.1 An example of objects

To show examples of our approach, this subsection describes objects which implement the following interface:

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

The interface `Counter` has two messages: `Increment` and `Print`. Note that this definition relies on the GADTs extension to restrict the types of the results.

As the first example, we define an object counter from `Counter` to `IO`. When the object receives `Increment`, it increments the internal state. Upon receiving the message `Print` it prints the state.

```hs
counter :: Int -> Object Counter IO
counter n = Object $ \case
  Increment -> return ((), counter (n + 1))
  Print -> do print n
             return ((), counter n)
```

Note that we abbreviate `\( \_r \rightarrow \text{case } r \text{ of } t \rightarrow \) ` case \( r \) of \( t \rightarrow \) case using the LambdaCase extension. In this implementation, expressions in right-hand side of case are methods. Here is an example usage of counter in GHCi [9].

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

`mockCounter` is another example of object `Object Counter IO`.

```hs
mockCounter :: Object Counter IO
mockCounter = go 0 0 where
  go :: Int -> Int -> Object Counter IO
  go m n = Object $ \case
    Increment -> do
      putStrLn ("Increment: " ++ show m)
      return ((), go (m + 1) n)
    Print -> do
      putStrLn ("Print: " ++ show n)
      return ((), go m (n + 1))
```

Although the internal state is different, it has the same type as `counter`. Thus, objects can be used to unite various states.

### 2.2 Unfolding

The following functions provide convenient ways to construct objects:

```hs
unfold0 :: Functor g => (forall a. r -> f a -> g (a, r)) -> r -> Object f g
unfold0 h = go
  where
    go r = Object $ fmap (fmap go) . h r

(@~) :: Functor g => s -> (forall a. f a -> StateT s g a) -> Object f g
s0 @~ h = unfold0 \( s \rightarrow (\text{runStateT } (h \_r) s) \) s0
```

`unfold0` performs unfolding of an object; an extra state \( r \) is passed along with a message. \((@~)\) has the same power as `unfold0`, but is wrapped with `StateT` provided by the transformers package [4].

```hs
With \((@~)\), the implementation of counter can be shorter:

counter' :: Int -> Object Counter IO
counter' n = n @~ \case
  increment -> modify (+1)
  Print -> get >>= lift . print
```

### 2.3 Instances with references

Most OOPLs use references to identify objects. In the same way, we can use references to do so in Haskell. For thread safety, we use `MVar` [5]. Thus, `newMVar` creates a new instance. To send a message to an instance, we define an operation which applies `runObject` within an `MVar`:

```hs
(... :: MVar (Object t IO) -> t a -> IO a
  m .- e = modifyMVar m $ \obj ->
    fmap swap $ runObject obj e

  swap :: (a, b) -> (b, a)
```

Using these operations, we can express instances based upon references as well as other OOPLs.

```hs
> i <- newMVar (counter 0)
> i .- Print
0
> i .- Increment
> i .- Increment
> i .- Print
2
```

If any exception is thrown during invocation of a method, it will replace the object with the original state by the semantics of `modifyMVar`. This behavior prevents contamination of a half-done state.

### 2.4 Message cascading for free

Naïve enumeration of messages (by GADTs) aren’t monads, restricting a message to be called just once; we want to make them monads to send cascaded messages.

We make use of operational monads [1] which express a chain of operations. The following code implements an operational monad `Program t` parameterized by the set of operations (i.e. an interface), \( t \).

```hs
data MonadView t m a where
  Return :: a -> MonadView t m a
  (:>>=) :: t a -> (a -> m b) -> MonadView t m b
infixl 1 :>>=
```

```hs
m .- e = modifyMVar m $ \obj ->
    fmap swap $ runObject obj e

  swap :: (a, b) -> (b, a)
```

Using these operations, we can express instances based upon references as well as other OOPLs.

```hs
> i <- newMVar (counter 0)
> i .- Print
0
> i .- Increment
> i .- Increment
> i .- Print
2
```

If any exception is thrown during invocation of a method, it will replace the object with the original state by the semantics of `modifyMVar`. This behavior prevents contamination of a half-done state.
cascadeObject::Monad g => Object f g  
= unfoldO cascading

cascading::Monad g => Object (Program f) g  
= Object (Program (t :>>= k))

instance Monad (Program t) where
  return = Program . Return
  Program (Return a) >>= k = k a
  Program (t :>>= j) >>= k = Program (t :>>= (k . j))

liftP::Monad g => t a -> Program t a
liftP t = Program (t :>>= Program . Return)

Program is expressive enough to describe a chain of operations.  
The following incNPrint is an example of message cascading with  
the Counter interface calling increment n times and then calling  
Print once:

incNPrint::Int -> Program Counter ()
incNPrint n = do  
  replicateM_ n $ liftP Increment
  liftP Print

An interpreter for Program may have some states.  As an example,  
we define an interpreter which is quite similar to the counter  
object using Counter:

runCounter::Int -> Program Counter a -> IO (a, Int)
runCounter n m = case view m of
  Return a -> return (a, n)
  Print n >> runCounter n (k ())
  Increment n >> runCounter (n + 1) (k ())

Taking a closer look, both counter and runCounter call themselves  
with next states.  The main difference is that runCounter  
captures the entire computation at once, while counter handles  
one operation.  Our object is a kind of extract of the pattern which  
can be defined separately from interpretation of instructions.  
cascadeObject is like runCounter, but takes Object as a first  
argument.

cascadeObject::Monad g
  => Object f g -> Program f a
  = cascadeObject (counterP 0) (incNPrint 42)

> cascadeObject (counterP 0) (incNPrint 42)
42

cascadeObject can be turned into an object.  The cascading  
function enhances an interface of an object.

cascading::Monad g => Object f g
  => Object (Program f) g
  = unfoldO cascadeObject

This cascading grants an object capability to handle cascaded  
messages.  The function counterP below is a derivative of counter  
enriched in this way.

counterP::Int -> Object (Program Counter) IO
counterP = cascading . counter

counterP is exactly unfoldO runCounter.  They have native  
capability of cascading:

> runObject (counterP 0) (incNPrint 42)
42

cascading is reversible because we have a function which lifts  
an instruction.  For every object m, uncascading (cascading m)  
is equivalent to m.

uncascading::Functor g
  => Object (Program f) g => Object f g
  = fmap (uncascading) . runObject obj . liftP

Thus, there is an isomorphism between Object (Program f) g and  
Objects f g.

The following laws should hold for every object whose both the  
interface and the base context are monads:

runObject obj (return a) = return a
runObject obj (m >>= k) = runObject obj mytt
  >>= \(a, obj') -> runObject obj' (k a)

Objects created by cascading holds the properties.

2.5 Instances without references

Our objects are tractable even without references.  The invokesOf  
fuction below invokes methods of targets of a traversal [11].

type LensLike' f s a = (a -> f a) -> s -> f s

invokesOf::Monad m
  => LensLike' (WriterT r m) s (Object f m)
  -> f a
  -> (a -> r)
  -> StateT n m r
  = StateT $ liftM swap . runWriterT . l go
    where
        go obj = WriterT $ runObject obj f
             >>= \(a, obj') -> return (obj', k a)

invokes::(Monad m, Monoid r, Traversable t)
  => f a
  -> (a -> r)
  -> StateT t (Object f m) m r
  = invokesOf traverse

When a traversal t is given, invokesOf t f k sends f :: f a  
to all the objects, passes each result to k :: a -> r, and  
combine the yields using the monoid Monoid r which is required  
by Applicative (WriterT r m).  invokesOf behaves like foldMap; it  
allows us to collect the results in a simple way.  When id is supplied,  
it combines the results over a monoid.  When empty is supplied, it  
simply discards the results.  If t is a lens (i.e. single target traversal),  
r does not need to be a monoid.  invokes is a specialized version of  
invokesOf that acts on a Traversable container.

> let xs0 = [counter 0, counter 1]
> xs1 <- execStateT (invokes Print id) xs0
0
  1
> xs2 <- execStateT (invokes Increment id) xs1
> xs3 <- execStateT (invokes Print id) xs2
1
  2

3. Composition

Objects are just as composable as functions.  In this section we  
introduce two important ways to compose objects; the vertical  
composition and the lateral composition.  Also, we show that our  
objects form a category.
3.1 Vertical composition

The vertical composition (\(\bowtie\)) is defined as follows:

\[
(\bowtie) :: \text{Functor } h \rightarrow \text{Object } f \rightarrow \text{Object } h

\rightarrow \text{Object } f
\]

Object \(m \bowtie \bowtie \bowtie \text{Object } n = \text{Object } \# \text{fmap} \text{joinO} \# \text{n} \# \text{m}
\]

joinO :: Functor h \rightarrow ((a, \text{Object } f \# g), \text{Object } h \# h)

\rightarrow (a, \text{Object } f \# h)

joinO ((x, a), b) = (x, a \# \bowtie \bowtie b)
\]

\(f\) is translated to \(g\), and \(g\) is translated to \(h\). The composition consolidates the flow hiding \(g\). This operation is associative. The identity element of composition is an object \text{echo} which parrots messages. For the proof of associativity and identity, see Appendix A.

\[
\text{echo} :: \text{Functor } f \Rightarrow \text{Object } f \# f

\text{echo} = \text{Object } (\text{fmap} \langle x \rightarrow (x, \text{echo}) \rangle)
\]

While the combination of \((\bowtie)^{-1}\) and \text{invokesOf}\ allows traditional object composition into practice, the vertical composition \((\bowtie)\) allows us to edit the behavior of objects from both sides. Consider the following two actors:

\[
\text{slime} :: \text{Object } \text{ActorBehavior World}

\text{zombie} :: \text{Object } \text{ActorBehavior World}
\]

Assume that we want to programmatically control the actors, we want to define \text{Object NPC} (\text{Program ActorBehavior}) rather than building \text{Object ActorBehavior World} in where NPC is an interface of non player characters:

\[
\text{offensive} :: \text{Object } \text{NPC (Program ActorBehavior)}

\text{defensive} :: \text{Object } \text{NPC (Program ActorBehavior)}
\]

The composition can be utilized to attach automated behavior with actors, keeping them loosely coupled.

\[
\text{offensiveZombie} :: \text{Object } \text{NPC World}

\text{offensiveZombie} = \text{offensive} \bowtie \bowtie \bowtie \text{cascading zombie}
\]

\[
\text{defensiveSlime} :: \text{Object } \text{NPC World}

\text{defensiveSlime} = \text{defensive} \bowtie \bowtie \bowtie \text{cascading slime}
\]

Composition glues a context of an object into an interface of another. We can consider that this example implements the Template Method pattern. offensive and defensive provides abstract methods, and zombie and slime implement concrete methods. This approach can be applied to Adapter, Decorator, Facade and Proxy as well.

Not just this encourages not only the extensibility of behavior and bodies, but it also allows them to be composed dynamically at runtime, depending on the difficulty setting for instance.

3.2 Category of action types

There is a category \text{Eff} whose objects are types of actions and morphisms are objects. Considering the types of objects, we will be able to discuss objects’ properties beyond the typical OOP.

\text{liftO} characterizes a functor from \text{End(Haskell)} to \text{Eff}.

\[
\text{liftO} :: \text{Functor } g \Rightarrow (\forall x. f x \rightarrow g x)

\rightarrow \text{Object } f \# g
\]

\[
\text{liftO} t = \text{Object } \# \text{fmap} \langle x \rightarrow (x, \text{lift0} t) \rangle \# t
\]

\text{liftO} preserves the composition of natural transformations:

\[
\text{liftO} (g \# f) = \text{liftO} f \# \bowtie \bowtie \bowtie \text{liftO} g
\]

\text{liftO} also preserves the identity transformation:

\[
\text{liftO} \# \text{id} = \text{echo}
\]

Therefore, natural transformations can be turned into objects losslessly.

3.3 Lateral composition

In our objects, the co-pairing makes sense in contrast to arrows’ pairing. The co-pairing for \text{Sum} of interfaces is straightforward:

\[
\text{data } \text{Sum } f \# g \# a = \text{InL } (f a) \# \text{InR } (g a)
\]

\[
(\bowtie) :: \text{Functor } m \Rightarrow \text{Object } f \# m \rightarrow \text{Object } h \# m

\rightarrow \text{Object } (\text{Sum } f \# g)
\]

\text{a} \bowtie \bowtie \bowtie \text{b} = \text{Object } \# \text{case}

\text{InL } f \rightarrow \text{fmap} (\text{fmap} (\text{a} \bowtie \bowtie \bowtie \text{b})) \text{runObject } a \# f

\text{InR } g \rightarrow \text{fmap} (\text{fmap} (\text{a} \bowtie \bowtie \bowtie \text{b})) \text{runObject } b \# g
\]

This operation divides incoming messages up to two objects, combining functionalities of objects laterally. The lateral composition \((\bowtie)\) is another significant way to compose objects. As in the example below, the lateral composition holds two objects together directly.

\[
\text{> i <- newMVar } \# \text{counter } \bowtie \bowtie \bowtie \text{counter}
\]

\[
\text{> i .- InL } \text{Increment}
\]

\[
\text{> i .- InR } \text{Print}
\]

\[
\text{0}
\]

4. Inheritance

The common mechanics for inheritance can be expressed on the category of objects.

4.1 Adding methods

In this subsection, we consider to extend a base object to another object with additional methods without modifying the base methods. Let \(b : B \rightarrow M\) be a base object where \(B\) is a base interface and \(M\) is a base context. Also, let \(ab : A + B \rightarrow M\) be an extended object where \(A\) is additional interface. When \(ab\) receives \(B\), it should pass \(B\) to \(b\). When \(ab\) receives \(A\), it should translate \(A\) into \(M + B\).

The following illustrates the commutative diagram to express objects to add new methods.

\[
\text{A} \rightarrow \text{B} \rightarrow \text{M}
\]

In the diagram above, \(f\) is an implementation of new methods. \(i_A\) and \(i_B\) are injections for sums. The morphism \((\text{id}, b) \circ (f, i_B)\) is the inherited object.

Because of the isomorphism between \text{Object} (\text{Program } t) \# m and \text{Object } t \# m, \text{Program } T has the same property as \(T\) in the category. We can use \text{Program} (\text{Sum } A B) instead of \text{Sum } A B if necessary.

The following example is a counter extended with \text{TwiceInc} that sends \text{Increment} twice. The pain of \text{InL} and \text{InR} can be killed by using automated injection; see Section 8.5 for the detail.

\[
\text{counterWithTwice} :: \text{Object } (\text{Sum TwiceInc Counter}) \text{IO}
\]

\[
\text{counterWithTwice} = (f \bowtie \bowtie \bowtie \text{liftO} (\text{liftO } \# \text{InL}))
\]
4.2 Overriding methods

In this subsection, we consider to extend a base object to another object by overriding base methods. Let \( b : B \to M \) be a base object where \( B \) is a base interface and \( M \) is a base context. Let \( b' : B \to M \) be an extended object with base methods overridden. When \( b' \) receives \( B \), it should translate \( B \) to \( M + B \).

The following illustrates the commutative diagram to express objects to add new methods.

\[
\begin{array}{c}
B \\
g \\
M + B \\
\quad [id, b] \\
M
\end{array}
\]

In the diagram above, \( g : B \to M + B \) is morphism for overriding. The morphism \([id, b]@g\) is the object overridden by \( g \). We will give an example of this inheritance in Section 4.3.

4.3 Proxy pattern

As an example of overriding, we show an expression of the Proxy pattern applied to \( \text{counter} \). The extended object accepts \( \text{Print} \) up to 5 times; It shows "Limit exceeded" if we tried to send \( \text{Print} \) more. Since \( \text{Program} \) is a monad, it is capable of expressing conditional or sequential actions.

```haskell
wrapper :: Int -> Object Counter (Program (Sum Counter IO))
wrapper n = Object $ \case
  | n < 5 -> do
    liftP $ InR Print
    return ((), wrapper (n + 1))
  | otherwise -> do
    liftP $ InL (putStrLn "Limit exceeded")
    return ((), wrapper n)

Increment -> do
  x <- liftP $ InR Increment
  return (x, wrapper n)

limitedCounter :: Object Counter IO
limitedCounter = wrapper 0

@>>@ cascading (counter @||@ echo) where
  f = Object $ \TwiceInc -> do
    liftP $ InR Increment
    liftP $ InR Increment
    liftP $ InL $ putStrLn "Incremented"
    return ((), f)
```

Using \( \text{EitherT} \) instead of \( \text{Either} \), we can make \( \text{Mortal} \) a monad transformer:

```haskell
newtype Mortal f g a = Mortal {
  unMortal :: Object f (EitherT a g)
}
runMortal :: Mortal f g a -> f x
  -> EitherT a g (x, Mortal f g a)
runMortal m = fmap (fmap Mortal) .
  runObject (unMortal m)

Mortal forms a monad because of the additional parameter \( a \).

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)

instance MonadTrans (Mortal f) where
  lift m = mortal $ const $ EitherT $ liftM Left m
```

5. Application

In this section, we describe mortal objects, games and streams as applications of our objects.

5.1 Mortal objects

Objects may die; invoking methods to dead objects causes unexpected behavior. To prevent this, all references to dead objects should be eliminated. However, in typical OOP, there is no mechanism to ensure the elimination. Also, a lag between the death and the elimination makes it unreliable to prevent method invocation.

The proposed representation is capable of expressing mortals in a way that the mortality is ensured by the type. An object is mortal if the context may fail, for example \( \text{Maybe} \), or \( \text{Either} \). Object \( f \) maybe may not return the next state after receiving a message. Object \( f \) (Either a) yields a final result upon death. We define the latter as a new type, \( \text{Mortal} \):

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

mortal and runMortal mimic Object (constructor) and runObject respectively. These allows us to define mortals in the same manner as immortal objects.

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)
```

Typically, mortals are managed in an immutable container. Since dead objects are filtered, the size of a container change after
an operation. We need to introduce a filter, a variant of traversal that supports deletion in addition to update:

```haskell
type Filter' s a = forall f. Applicative f => FilterLike' f s a

type FilterLike' f s a = (a -> f (Maybe a)) -> s -> f s

class Filterable t where
    theFilter :: Filter' (t a) a
```

For instance, lists are an instance of Filterable:

```haskell
instance Filterable [] where
    theFilter f (x:xs) = maybe id (:) <$> f x <*> theFilter f xs
    theFilter _ [] = pure []
```

The apprisesOf function below sends a message to all the mortals through a filter, generalizing invokesOf.

```haskell
apprises :: (Monad m, Monoid r, Filterable t) => f a -> (a -> r) -> (b -> r) -> StateT (t (Mortal f m b)) m r
apprises = apprisesOf theFilter

apprisesOf :: Monad m => FilterLike (WriterT r m) s (Mortal f m b) -> f a -> (a -> r) -> (b -> r) -> StateT s m r
apprisesOf l f p q = StateT $ \t ->
    liftM swap $ runWriterT $ flip l t $ \obj -> WriterT $ liftM d $ runEitherT (runMortal obj f)
    where
d (Left r) = (Nothing, q r)
d (Right (x, obj')) = (Just obj', p x)
```

## 5.2 Games

We consider a very simple example of a game in this subsection. Consider a breakout game; a crucial component, block, will be broken upon hit.

```haskell
data V2 a = V2 a a deriving Show

data Picture = Block (V2 Float)
| Burst (V2 Float)
| Overlay Picture Picture
| Blank
deriving Show

drawPicture :: Picture -> IO ()
drawPicture (Block a) = putStrLn $ "Block " ++ show a
drawPicture (Burst _) = putStrLn "KABOOM!"
drawPicture (Overlay a b) = drawPicture a >>= drawPicture b
drawPicture Blank = return ()

instance Monoid Picture where
    mempty = Blank
    mappend = Overlay

Blocks can be expressed by Mortal:

data Entity x where
    Render :: Entity Picture
    Hit :: Entity ()

block :: Monad m => V2 Float

    -> Mortal Entity m (V2 Float)
    Render -> return (Block pos, block pos)
    Hit -> left pos

hardBlock :: Monad m => Int -> V2 Float
    -> Mortal Entity m (V2 Float)
hardBlock n pos = mortal $ \case
    Render -> return (Block pos, hardBlock n pos)
    Hit | n <= 1 -> left pos
    otherwise -> return (), hardBlock (n - 1) pos

block is so fragile that one Hit breaks it. hardBlock n needs n times to break. Although they have different internal states, they can be stored in a list as they have the identical type:

```haskell
> let bs0 = [block (V2 0 0), hardBlock 3 (V2 0 1)]
```

apprisesOf conquers the typical game loop process; it invokes methods of all the targets, and removes corpses.

```haskell
renderAll :: StateT [Mortal Entity IO b] IO ()
renderAll = apprises Render id mempty
>>= lift . drawPicture

renderAll gathers the results of Render and prints the result.

```haskell
> bs1 <- execStateT renderAll bs0
> Block: V2 0.0 0.0
> Block: V2 1.0 0.0
```

If they got hit, only the harder one will remain.

```haskell
> bs3 <- execStateT (apprises Hit id mempty) bs1
> Block: V2 1.0 0.0
```

Suppose one day we’ve implemented an explosion effect.

```haskell
burst :: Monad m => V2 Float -> Mortal Entity m ()
burst pos = mortal $ \case
    Render -> return (Burst pos, return ())
    Hit -> return (), burst pos
```

What should we do to add the explosion effect to the blocks is quite simple; block pos >>= burst is an object that explodes when broken. Mortal objects provide yet another composability as a monad, making them easier to extend behavior along the time axis.

## 5.3 Stream

Objects also behave as consumers or producers. The following Req a b is a type of a message that sends a to receive b:

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

An object with interface Req a b behaves as a transducer from a to b. They can be composed in another way:

```haskell
(<$>) :: Monad m => Object (Req a b) m
    -> Object (Req b c) m
    -> Object (Req a c) m

s <$> t = Object $ \(Req a) -> do
    (b, s') <- runObject s (Req a)
    (c, t') <- runObject t (Req b)
    return (c, s' <$> t')
```

An object with interface Req () a functions as a producer for a. When we pass Req () to Object (Req () a) m, it returns m a. The code below is an example of an object that generates natural numbers.
We have defined objects but the property of our approach slightly differs from what she explained. An object has state and behavior, however, objects are not identifiable. There is a clear distinction between our persistent objects and instances. Encapsulation is achieved quite well. The only way to control objects is to send messages. The concept of Message Passing and Methods are quite simple in our object; it is just invocation of runObject. Since our object may contain different methods, our approach provides Polymorphism certainly. Abstraction is also achieved. Our object hides the internal state and the type is determined by the context and the interface.

Our approach does not employ this nor self reference, and thus open recursion is impossible. The absence of self reference does not impede our purpose that provides better state manipulation in Haskell. Note that the Template Method pattern is typically implemented with open recursion in other OOPLs but it can be implemented with the vertical composition in our approach as discussed in Section 3.1.

An interface like StateT s (Program t) can be considered that it has a public field s. Any interface including this can be hidden using vertical composition.

### 6.1 Natural Transformation and mealy machines

The proposed objects can be thought of as mealy machines augmented with naturality. The structure of our Object has actually been derived from these two perspectives. When we send a message to an object, it returns a result. Hence we consider objects as a kind of mealy machines that consumes messages and produces results. The following definition represents a simple mealy machine.

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

While Mealy can have a state implicitly, it restricts both input and output types to be monomorphic. We like to define methods with different types of result. To solve this, the input and output types should have a parameter so that it has the kind * -> *.

Given an interface M, a context N, and a type of result a, the invocation of method will have the following type:

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

It is a natural transformation when M and N are both functors. Natural transformations are expressed by the following natural type in Haskell:

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

For every functor M, N, a value m :: M a, function f, natural transformation nat :: Natural M N, the following equation is satisfied by parametricity, and we call it naturality.

```haskell
runNatural (fmap f m) = fmap f (runNatural nat m)
```

When the interface is a functor, we can define naturality of objects as well as natural transformations. For every object obj, the following stands by parametricity:

```haskell
runObject obj (fmap f m) = fmap (f *** id) (runObject obj m)
```

Note that (***) is an operator to pair arrows which is defined in Control.Arrow in the base package.

The fact that the proposed method subsumes both mealy machines and natural transformations is shown by simply writing a 3

---

3 http://hackage.haskell.org/package/void
transformation between them. The fromNatural function embeds a natural transformation in an Object:

fromNatural :: Functor g => Natural f g -> Object f g
fromNatural (Natural t) = lift0 t

fromMealy embeds a mealy machine in an Object:

fromMealy :: Mealy a b r -> Object (Req a b r)

let (b, m) = t a
let (b, m') = Object $ \(\text{Req} a) ->

let (b, m) = t a

in Identity (b, fromMealy m)

Req a b r defined in Section 5.3 is isomorphic to a and the result must be b by the restriction of the generalized algebraic data types (GADTs) constructor. Since Identity x is isomorphic to x, Object (Req a b) Identity is equivalent to Mealy a b.

The notion of objects and composition is closely related to arrows [3]. We can lift a function into an arrow:

arr :: Arrow a => (b -> c) -> a b c

And we have the following function that embeds a morphism into something more powerful:

fromNatural :: Natural f g -> Object f g

Every arrow is capable of pairing such that:

(&&&) :: Arrow a => a b c -> a b c' -> a b (c, c')

On the other hand, we have co-pairing for objects.

(¶¶) :: Arrow f h
-> Object g h
-> Object (Sum f g) h

7. State Manipulation in Haskell

This section compares some approaches to deal with states in Haskell.

7.1 Lens

Lenses and our objects are complementary. Lenses provide convenient access to a particular state. Our object encapsulates states to get them to be dealt with more easily. As shown in Section 2.5, lenses and traversals can be used not only to construct stateful objects, but also to manipulate objects without references.

7.2 Operational

We argued that our object is a generalization of operational monad interpreters in Section 2.4. An object that the interface is an operational monad is capable of cascading and the capability can be achieved without any effort.

7.3 OOHaskell

OOHaskell is another research to achieve OOP in Haskell. Our approach is based on functions while OOHaskell is based on extensible records implemented in the HList library. As discussed in Section 2, function-based allows infinite behavior; it allows cascading which implies infinite messages. But in record-based one, messages must be enumerable since a size of a record must be finite. Since OOHaskell relies on IORef to store states, it is only usable on either IO or ST. Besides the impurity, the implementation of the extensible record is quite complex and developers often need to deal with verbose type signatures. The corecursive structure of our approach provides native encapsulation of states, thus our objects themselves do not taint pure code as shown in Section 2.5.

7.4 Functional reactive programming

Functional reactive programming (FRP) establishes encapsulation of states. FRP is classified to two styles; Classical FRP and arrowized FRP [16]. Classical FRP introduces event and behavior to express input and output respectively. Arrowized FRP introduces a processing arrow type that transforms input to output. Both provide an elegant expression for signal flow; however, they force monomorphic input/output and often remain incompatibility with first-class effects. Bidirectionality or the presence of (side-)effects easily messes the code up.

Our object systems can be thought of as lifted arrowized FRP. Objects provide composition as signal functions do, and they can accumulate incoming values into local states. It is even possible to define filter for objects:

data Fallible t a where
    Fallible :: t a -> Fallible t (Maybe a)

filter0 :: Monad m
-> Object (Fallible t) m
then runObject obj t
>>= \(a, obj') -> return (Just a, filter0 p obj')
else return (Nothing, filter0 p obj)

filter0 p censorx messages using the predicate p. If a message is censored, the result becomes Nothing.

8. Expression Problem

Especially in game development, extensibility of data is quite important as we want to store various entities in one container dynamically, keeping possibility to add more. Extensibility of operations encourages reusability of entities.

However, the expression problem [15] claims that it is difficult to make an abstraction which achieves the following two properties without recompilation of existing definitions, keeping the type safety:

Extensibility of operations New operations can be easily added to an existing data type.

Extensibility of data New data (inhabitants) can be easily added to an existing data type.

Hence, this section compares our objects with related works on the expression problem.

As an example, we consider elves and orcs as data and curse as an operation. When elves get cursed, their sanity decreases. Orcs are immune to curse and have no state (in practice, we may also want to add health points, offensive power, etc.). If implementation of this example has extensibility of operations, a new operation, for instance, heal can be added. When elves get healed, their sanity increases. Orthogonally, if it has extensibility of data, a new type of inhabitant, for example, dwarves can be added.

8.1 Class-based OOPLs

Since Haskell is a statically typed programming language, we compare Haskell with statically typed OOPLs such as Java, C++, C# and Objective C. They have extensibility of data but does not have extensibility of operations. The extensibility of data can be achieved by adding subclasses. The example is implemented as follows in C#:

```csharp
interface Entity {
    void curse();
}
```
class Elf : Entity {
    private int sanity;
    Entity() {
        sanity = 10;
    }
    public void curse() {
        sanity -= 1;
    }
}

class Orc : Entity {
    public void curse() {}
}

However, it is impossible to extend operations without recompiling the code because operations are locked in a class.

8.2 Algebraic data types
It is common to use algebraic data types to integrate entities. Algebraic data types has extensibility of operations but does not have extensibility of data.

data Entity = Elf Int | Orc
curse :: Entity -> Entity
curse (Elf n) = Elf (n - 1)
curse Orc = Orc

It is easy to add new functions in addition to curse. But if we need to add dwarves, we have to edit both Entity and curse.

8.3 Existential quantification
Existential quantification has extensibility of data but does not have extensibility of operations. Typeclasses provide overloading; thanks to the mechanics, we can add new data for an existing typeclass, sharing the common methods.

data Elf = Elf Int
data Orc = Orc
class Curse s where
    curse :: s -> s
instance Curse Elf where
curse (Elf n) = Elf (n - 1)
curse Orc = Orc

The Entity integrates individuals using existential quantification:

data Entity = forall s. Curse s => Entity s

However, like other OOPLs, it is impossible to extend operations for Entity without modification of the Curse typeclass. We also need to define an existential wrapper per typeclass.

8.4 OO Haskell
The above example is implemented with OO Haskell as follows:

{-# LANGUAGE DataKinds #-}
import Data.HList

curse = Label :: Label "curse"

elf :: Int -> IO (Record 'Tagged "curse" (IO ()))
elf n = do
    sanity <- newIORef n
    return $ curse .=. modifyIORef sanity (subtract 1) .*. emptyRecord

orc :: IO (Record 'Tagged "curse" (IO ()))

It is easy to add new functions in addition to curse. But if we need to add dwarves, we have to edit both Entity and curse.

8.5 Polymorphic variants, or open unions
A polymorphic variant is extensible and is a popular solution to the expression problem in OCaml. In Haskell, type-indexed coproducts [7, 14] fulfill the role.

data Union (r :: [*]) -- abstract
    inj :: Member t r => t -> Union r
    prj :: Member t r => Union r -> Maybe t
    decomp :: Union (t ': r) -> Either (Union r) t

inj injects a value into a union; prj tries to extract a value. Unlike algebraic data types defined in the usual way, these do not restrict the type of the unions. It allows them to be extensible in both data and operations.

Our object is not an alternative. On the contrary, our object can be equipped with open unions. To construct Sum introduced in Section 3, we need to write InL or InR by hand. It can be unified to inj using open unions. Open unions encourage extensibility of behavior as well as extensible effects [7].

8.6 Data types à la carte
Swierstra’s data types à la carte[14] is a well known approach to the expression problem. The approach introduces a typeclass per datum and an instance per operation. The original expression is as follows. Typeclass Run represents one entity and the parameter f represents the type of actions for that:

class Run f where
run :: f (Mem -> (a, Mem)) -> Mem -> (a, Mem)

For simplicity, we roll Mem -> (a, Mem) into State Mem and then convert f (State RunS a) -> State RunS a of a -> State RunS a, eliminating the need for continuation passing:

class Run t where
run :: t a -> State Mem a

This solution fundamentally uses natural transformation between the operation and State s where s is the target-specific state. The following is implementation of the Elf and Orc example based on this style:

data ElfState = Int
class Elf t where
    elf :: t a -> State ElfState a

4 http://hackage.haskell.org/package/vinyl
Therefore, we can turn them into objects by just supplying initial states.

instance Elf f, Elf g) => Elf (Sum f g) where
  elf (InL f) = elf f
  elf (InR f) = elf f

instance Elf Curse where
  elf Curse = modify (subtract 1)

Data are extensible by adding new typeclasses.

type OrcState = ()

class Orc t where
  orc :: t a -> State OrcState a

instance (Orc f, Orc g) => Orc (Sum f g) where
  orc (InL f) = orc f
  orc (InR f) = orc f

instance Orc Curse where
  orc Curse = return ()

Operations are also extensible as we can add new instances.

data Heal a where
  Heal :: Heal ()

instance Elf Heal where
  elf Heal = modify (+ 1)

instance Orc Heal where
  orc Heal = return ()

However, this approach itself does not provide a way to integrate different states. Thus, it is impossible to store elf and orc in a container because these types are different:

elf :: Sum Curse Heal a -> State ElfState a
orc :: Sum Curse Heal a -> State OrcState a

8.7 Our object

Since our object is just a value, it is obvious that we can extend data. We take over the extensibility of operations of the data types à la carte approach. We reuse elf and orc defined in the previous subsection:

elf :: Sum Curse Heal a -> StateT ElfState Identity a
orc :: Sum Curse Heal a -> StateT OrcState Identity a

Remember that the (\$^\dagger\$) operator ties an initial state \(s\) with a natural transformation \(f \rightarrow StateT s \circ g a\) and produces `Object f g`.

\[
\begin{align*}
\circ \ 
\end{align*}
\]

\[
(\circ^\dagger) :: \text{Functor g} \ 
\rightarrow s 
\rightarrow (\forall a. \ f \rightarrow \text{StateT} s \circ g a) 
\rightarrow \text{Object f g}
\]

Therefore, we can turn them into objects by just supplying initial states:

10 \$^\dagger\$ elf :: `Object (Sum Curse Heal) Identity () \$^\dagger\$ orc :: `Object (Sum Curse Heal) Identity

Since the internal states, ElfState and OrcState are encapsulated and the types are identical, they can be stored in one container.

9. Conclusion

We have presented purely-functional remodeling of objects to deal with states flexibly. OOP has the advantage of extensibility of data while algebraic data types and typeclasses are clumsy. On the other hand, conventional objects have been not composable. Most encodings of objects are record-based and compositability of messages is also unreachable. The object we have introduced solves the expression problem in a composable manner; our object is a morphism in the category of effects. Since both objects and messages are just data, it provides far greater maneuverability than existing designs. Moreover, our solution for mortal objects provides an innovative way to deal with ephemeral objects comfortably.

Acknowledgements

We thank Oleg Kiselyov, Tomohiro Oda, Satoshi Ogasawara, Jacques Garrigue, Kei Hibino, Michael Snoyman and Gabriel Gonzalez for their valuable comments and discussions. We are also thankful to anonymous reviewers of Program and Programming Language Workshop 2015, to which the original paper submitted, for concise and helpful suggestions. We would like to express our gratitude for two anonymous reviewers of Haskell Symposium for their valuable comments to improve this paper. Kinoshita has been working on this research as a part-time of IIJ Innovation Institute.

References

A. Proofs

In this appendix, we prove associativity, left identity and right identity of our objects. For readability, Object and runObject are abbreviated as inO and outO respectively.

A.1 Associativity

Theorem 1 For arbitrary actions e, f, and g and objects a::Object e f, b::Object f g and c::Object g h, associativity a @>>@ (b @>>@ c) = (a @>>@ b) @>>@ c holds.

Proof 1 Using equational reasoning

\[
\text{outO (a @>>@ (b @>>@ c))} = \text{outO ((a @>>@ b) @>>@ c)}
\]

A.2 Left identity

Theorem 2 echo @>>@ obj = obj holds for every object obj::Object f g.

Proof 2 Using equational reasoning

\[
\text{outO (obj @>>@ Object (fmap \langle x \rightarrow (x, echo)\rangle))} = \text{outO obj}
\]

A.3 Right identity

Theorem 3 obj @>>@ echo = obj holds for every object obj::Object f g.

Proof 3 Using equational reasoning

\[
\text{outO (obj @>>@ echo)} = \text{outO obj}
\]