Simple blog engine with shape functors and generic eliminators for ADTs

Andor Penzes

zerobuzz

September 15, 2016

Introduction

An experiment on a blog engine is complicated enough, to be a real world like problem. It is small enough to code it in a few hours after work.

Eliminators

This will be confusing as the same phenomenon has many names in the literature.

- Generic eliminator
- Eliminator
- Catamorhism
- Initial algebra
- Template function

The full power of the generic eliminators are connected to dependent typed programming.

Haskell is not dependent yet, let's use a simple approach.

In theory, there is no difference between theory and practice. But, in practice, there is. (Jan L. A. van de Snepscheut)

In theory, there is no difference between theory and practice. But, in practice, there is. (Jan L. A. van de Snepscheut)

This presentation is about the definitions and the practical use of generic eliminators.

```
-- Abstract deepsense. (Matthias Fishmann)
module Eliminators.Theory where
```

Basic example

```
data List a
  = Empty
  | Cons a (List a)

length :: List a -> Int
length Empty        = 0
length (Cons _ xs) = 1 + length xs
```

Abstract the recursion

For every ADT we can define an algebra based on its structure.

Abstract the recursion

For every ADT we can define an algebra based on its structure.

Eliminator for an ADT captures the structure structure of the ADT, when the ADT is recursive the eliminator is applied for the recursion.

Abstract the recursion

For every ADT we can define an algebra based on its structure.

Eliminator for an ADT captures the structure structure of the ADT, when the ADT is recursive the eliminator is applied for the recursion.

```
list_elim :: (b, a -> b -> b) -> List a -> b
list_elim (empty, cons) Empty       = empty
list_elim (empty, cons) (Cons a as) =
    cons a (list_elim (empty, cons) as)
```

Abstract the recursion

For every ADT we can define an algebra based on its structure.

Eliminator for an ADT captures the structure structure of the ADT, when the ADT is recursive the eliminator is applied for the recursion.

```
list_elim :: (b, a -> b -> b) -> List a -> b
list_elim (empty, cons) Empty        = empty
list_elim (empty, cons) (Cons a as) =
    cons a (list_elim (empty, cons) as)

length_alg :: (Int, a -> Int -> Int)
length_alg = (0, \_ n -> 1 + n)

length' :: List a -> Int
length' = list_elim length_alg
```

Abstract the list shape

Shape functors.

Eliminators for an ADT can be separated into the shape of the ADT and the recursion scheme.

```
data ListShape a rec
  = EmptyS
  | ConsS a rec
  deriving (Show)

newtype Fix f = In { unFix :: f (Fix f) }

type ListF a = Fix (ListShape a)

e   = In EmptyS
ae  = In (ConsS 1 e)
aae = In (ConsS 2 ae)
```

Abstract the list shape

```
listElim :: (b, a -> b -> b) -> ListF a -> b
listElim (empty, cons) (In EmptyS)       = empty
listElim (empty, cons) (In (ConsS a s)) =
    cons a (listElim (empty, cons) s)

length'' = listElim (0, (\_ n -> (1 + n)))
```

Abstract the cases of list shape

Let's rename listElim to listCata as we abstracting away from the value processing.

```
listCata :: (ListShape a s -> s) -> ListF a -> s
listCata alg (In EmptyS)        = alg EmptyS
listCata alg (In (ConsS a s)) = alg (ConsS a (listCata alg s))

lengthAlg :: ListShape a Int -> Int
lengthAlg EmptyS      = 0
lengthAlg (ConsS _ n) = 1 + n

length''' = listCata lengthAlg
```

Shape functor

```
instance Functor (ListShape a) where
    fmap f EmptyS       = EmptyS
    fmap f (ConsS a s) = ConsS a (f s)
```

Catamorphism

Let's abstract the shape functor. In Category theory the algebras are defined for functors. Many algebra can be defined for the given functor.

```
-- newtype Fix f = In { unFix :: f (Fix f) }

type Algebra f a = f a -> a

cata :: Functor f => Algebra f s -> Fix f -> s
cata alg = alg              -- Compute the result from the partials
         . fmap (cata alg) -- Compute the partial results
         . unFix           -- Step inside

length'''' = cata lengthAlg
```

Catamorphism is a recursion scheme.

More than cata

Factorial?

Catamorhisms are not powerfull enough, there is a zoo of morphisms. We need an another type of morphism to be able to define the factorial function.

http://hackage.haskell.org/package/fixplate

```
-- Concrete nonsense.
module Eliminators.Practice where
```

Real World Development

Find the balance between the abstractions and concreteness.

Real World Development

Find the balance between the abstractions and concreteness.

Real world development usually

- is not too abstract
- uses modular approach
- is powerful enough to cover the problems.

Real World Development

Find the balance between the abstractions and concreteness.

Real world development usually

- is not too abstract
- uses modular approach
- is powerful enough to cover the problems.

Use Fix, Shape functors and cata if your data types are tend to be recursive and there is a high probability of changes

Use Eliminators otherwise.

The godfather of all eliminators

First and well known lazy generic eliminator in every programming language!

```
boolElim t f e = if e then t else f
```

or

```
boolElim' t f e = case e of
    True  -> t
    False -> f
```

More eliminators

With Haskell we can create eliminators for every ADT, based on the structure of the ADT. With laziness generic eliminators can serve as template functions for the values we work with.

```
maybeElim n j m = case m of
    Nothing -> n
    Just x  -> j x

eitherElim l r e = case e of
    Left x  -> l x
    Right y -> r y
```

Composition

Composition of eliminators comes from the structural induction on the shape of ADT.

```
compExample =
    eitherElim
      (eitherElim
        (show . (1+))
        ("x=" ++))
      (maybeElim "NaN" (show . floor))
```

Using intendation helps a lot. It is very similar to the pointfree style.

Design recipe

Design recipe

- Create an ADT

Design recipe

- Create an ADT
- Create its eliminator based on the stucture

Design recipe

- Create an ADT
- Create its eliminator based on the stucture
- Encapsulate this definitions in a module

Design recipe

- Create an ADT
- Create its eliminator based on the stucture
- Encapsulate this definitions in a module
- Sometimes it is useful to add a typed hole, which can carry out information

Design recipe

- Create an ADT
- Create its eliminator based on the stucture
- Encapsulate this definitions in a module
- Sometimes it is useful to add a typed hole, which can carry out information
- Create algebras to define functions with eliminators

Real world data

In real world examples the information usualy organized in a tree shaped data.

Entry

```
data Entry a = Entry {
    e_hole  :: a
  , e_lines :: Pandoc
  } deriving (Functor, Eq, Show)

type EntryAlgebra a b = (a -> Pandoc -> b)

entryElim :: EntryAlgebra a b -> Entry a -> b
entryElim alg (Entry hole lines) = alg hole lines
```

Type hole in Entry. With a type hole we can expres more computational power and can convert our regular data type to a shape functor, and if we need we can use it in Fix compuations.

TopicName

```
data TopicName a = TopicName {
    tn_hole :: a
  , tn_name :: Pandoc
  } deriving (Functor, Eq, Show)

type TopicNameAlgebra a b = (a -> Pandoc -> b)

topicNameElim :: TopicNameAlgebra a b -> TopicName a -> b
topicNameElim alg (TopicName hole name) = alg hole name
```

Topic

```
data Topic a = Topic {
    t_hole      :: a
  , t_topicName :: TopicName a
  , t_entries   :: [Entry a]
  } deriving (Functor, Eq, Show)
```

Topic

```
data Topic a = Topic {
    t_hole      :: a
  , t_topicName :: TopicName a
  , t_entries   :: [Entry a]
  } deriving (Functor, Eq, Show)
```

How to define an eliminator and algebras for Topic?

```
type TopicAlgebra a t e es p =
    ( TopicNameAlgebra a t
    , EntryAlgebra a e
    , ListAlgebra e es
    , a -> t -> es -> p)
topicElim :: TopicAlgebra a t e es p -> Topic a -> p
topicElim (topicNameAlg, entryAlg, entriesAlg, combine)
            (Topic hole topicName entries)
  = combine
        hole
        (topicNameElim topicNameAlg topicName)
        (listElim_ entriesAlg (entryElim entryAlg <$> entries))
```

Blog

```
data Blog a = Blog {
    b_hole    :: a
  , b_summary :: Pandoc
  , b_topics  :: [Topic a]
  } deriving (Functor, Eq, Show)

type BlogAlgebra a t e es p bs b =
    ( TopicAlgebra a t e es p
    , ListAlgebra p bs
    , a -> Pandoc -> bs -> b )

blogElim :: BlogAlgebra a t e es p bs b -> Blog a -> b
blogElim (topic, topicList, combine)
         (Blog hole summary topics)
  = combine
        hole
        summary
        (listElim_ topicList (topicElim topic <$> topics))
```

Renderer

```
renderPages :: FilePath -> (NavPath -> Html -> Html)
            -> Blog FileProperties -> IO ()
renderPages outDir frame = blogElim render where
  render = (topic, sequence_, topicList)
  topic  = (topicName, entry, entryList, topicNameEntryList)

  entry fp pandoc = do
    writeFile (outDir </> (markdownPathToHTMLPathFP fp))
              (renderHtml . frame NavBackward $ pandoc2html pandoc)
    return (fp, firstHeader pandoc)

  entryList = (return [], \x xs -> (:) <$> x <*> xs)
  topicList _ _ ts = ts
  sequence_ = (return (), (>>)) -- Monoid instance of monads
```

Renderer

```
topicName fp pandoc = do
  createDirectoryIfMissing True $ outDir </> markdownPathToHTMLDir fp
  return (\content -> writeFile
                (outDir </> (markdownPathToHTMLPathFP fp))
                (renderHtml $ frame NavInPlace content)
         , pandoc
         )

topicNameEntryList _ topicName entryList = do
  (topicPage, pandoc) <- topicName
  headers <- entryList
  topicPage $ do -- :: Html
    pandoc2panel pandoc
    topicsList headers
```

Drawbacks / Solutions

Drawbacks:

Drawbacks / Solutions

Drawbacks:

- If the type is the same in every case eliminators can be easily swapped

Drawbacks / Solutions

Drawbacks:

- If the type is the same in every case eliminators can be easily swapped
- No names of the cunstructors are given

Drawbacks:

- If the type is the same in every case eliminators can be easily swapped
- No names of the cunstructors are given

Solutions:

Drawbacks / Solutions

Drawbacks:

- If the type is the same in every case eliminators can be easily swapped
- No names of the cunstructors are given

Solutions:

- Create an ADT for the algebra and name the constructors

Drawbacks:

- If the type is the same in every case eliminators can be easily swapped
- No names of the cunstructors are given

Solutions:

- Create an ADT for the algebra and name the constructors
- Use the new Symbol types as type parameter to name the different cases

Named parameters

Use named parameters

```
data Param (n :: Symbol) a = Param a

maybeElimNamed
    :: (Param "nothing" b) -> (Param "just" (a -> b)) -> Maybe a -> b
maybeElimNamed (Param nothing) (Param just) = \case
    Nothing -> nothing
    Just x  -> just x

test = maybeElimNamed
            (Param 0    :: Param "nothing" Int)
            (Param (1+) :: Param "just" (Int -> Int))
```

Connection to lenses

- Lenses are coalgebras, composition works via function composition
- Eliminators use algebras, composition works via tupling
- Eliminators are like universal properties for an ADT

Future work

More...

- Use template haskell to generate eliminators from the ADT
- Use generics-sop library to generate eliminators
- Create a library

Conclusion

Conclusion

- Similar to point free style
- Algorithms are compact, but still understandable
- Composition done by chaining or tupling of algebras

https://github.com/andorp/andorp.github.io/tree/master/haskell