Structured Traversals for ((Multiply) Recursive) Algebraic Datatypes

G. Cassian Alexandru

January 16, 2021

Presentation generated from `.lhs` sources using `lhs2TeX`

```
length :: [a] → Int
length = λcase
[] → 0
(x:xs) → 1 + length xs

filter :: (a → Bool) → [a] → [a]
filter p = go where
  go [] = []
  go (x:xs) = if p x then [x] else [] ++ go xs
```

```
length :: [a] → Int
length = λcase
[] → 0
(x:xs) → 1 + length xs

filter :: (a → Bool) → [a] → [a]
filter p = go where
  go [] = []
  go (x:xs) = if p x then [x] else [] ++ go xs
```

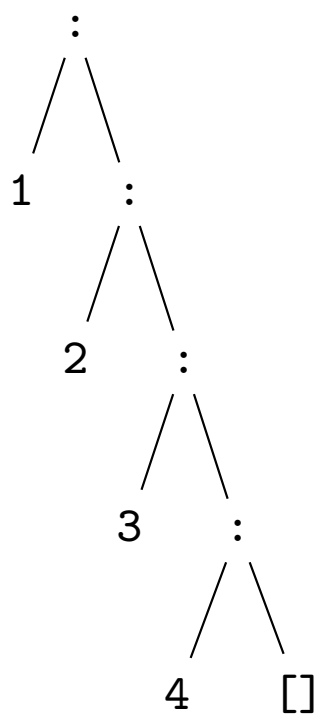
```
length :: [a] → Int
length = λcase
[] → 0
(x:xs) → 1 + length xs

filter :: (a → Bool) → [a] → [a]
filter p = go where
  go [] = []
  go (x:xs) = if p x then [x] else [] ++ go xs
```

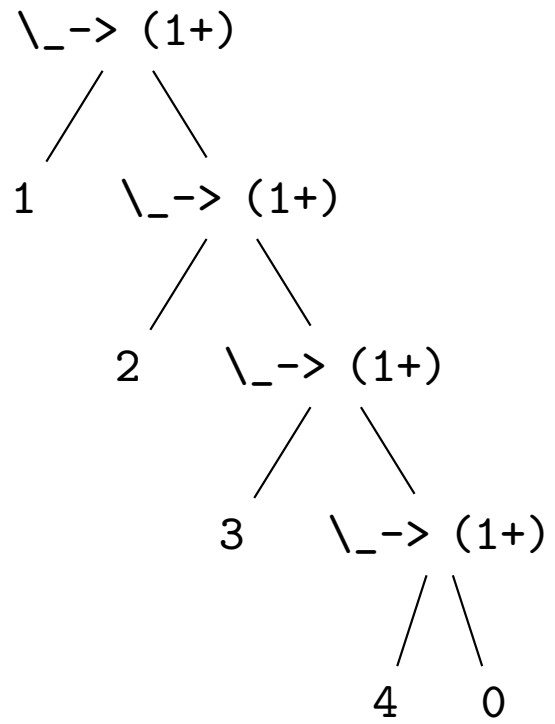
- List Design pattern?
- Design Patterns are a poor man's abstraction
- Recognize common structure & find correct abstract notion

Traversals

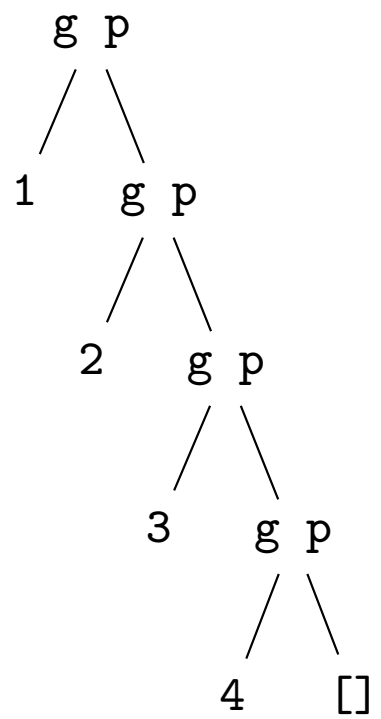
List



`length`



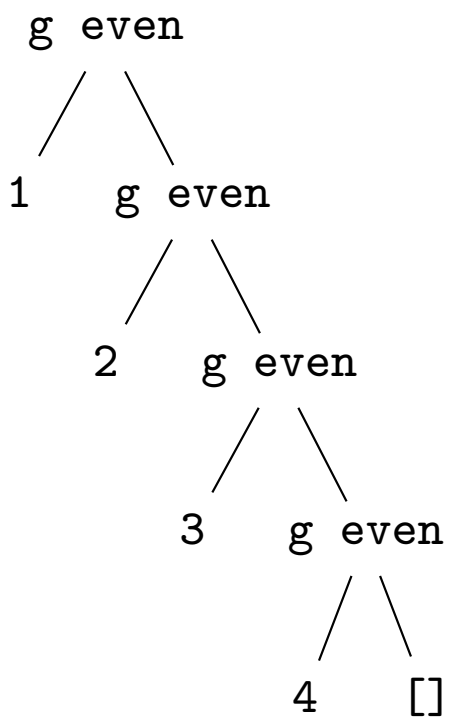
`filter p`




```
g p x xs =
  bool [] [x] (p x) ++ xs
```

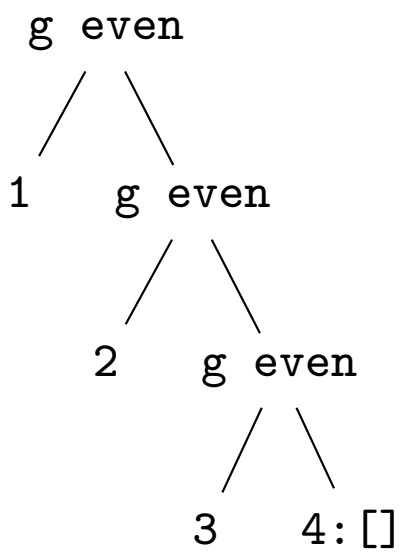
Example Evaluation of `filter even`

```
g p x xs =
  bool [] [x] (p x) ++ xs
```



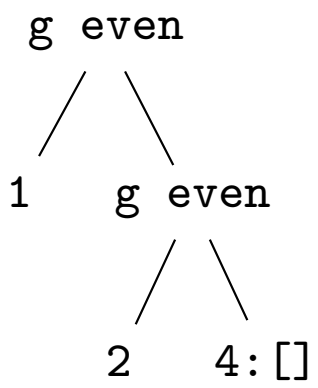
Example Evaluation of `filter even`

```
g p x xs =
  bool [] [x] (p x) ++ xs
```



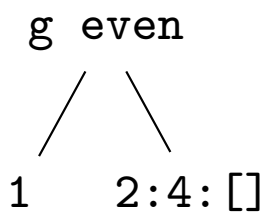
Example Evaluation of `filter even`

```
g p x xs =
  bool [] [x] (p x) ++ xs
```



Example Evaluation of `filter even`

```
g p x xs =
  bool [] [x] (p x) ++ xs
```



Example Evaluation of `filter even`

```
g p x xs =
  bool [] [x] (p x) ++ xs
```

```
2:4:[]
```

```
data List a = Nil | Cons a (List a)
data BooL = TT | FF
```

GADT Syntax:

```
data List a where
  Nil :: List a
  Cons :: a → (List a) → (List a)
data BooL where
  FF :: BooL
  TT :: BooL
```

```
data List a where
  Nil :: List a
  Cons :: a → (List a) → (List a)
data BooL where
  FF :: BooL
  TT :: BooL

list :: b → (a → b → b) → List a → b
list nil cons = fold where
  fold Nil = nil
  fold (x `Cons` xs) = x `cons` fold xs

booL :: b → b → BooL → b
booL tt ff = fold where
  fold FF = ff
  fold TT = tt
```

```
data List a where
  Nil :: List a
  Cons :: a → (List a) → (List a)
data BooL where
  FF :: BooL
  TT :: BooL

list :: b → (a → b → b) → List a → b
list nil cons = fold where
  fold Nil = nil
  fold (x `Cons` xs) = x `cons` fold xs

booL :: b → b → BooL → b
booL tt ff = fold where
  fold FF = ff
  fold TT = tt
```

Structural Functors

```
data ListF c x = NilF | ConsF c x -- deriving Functor
data BooLF x = TTF | FFF deriving Functor

instance Functor (ListF c) where
  fmap :: (a → b) → (((ListF c) a) → ((ListF c) b))
  fmap f = λcase
    NilF → NilF
    ConsF c a → ConsF c (f a)
```

If you are unfamiliar with functors: In our case regarding them as a computational context with holes suffices.

Structural Functors

```
data ListF c x = NilF | ConsF c x -- deriving Functor
data BooLF x = TTF | FFF deriving Functor

instance Functor (ListF c) where
  fmap :: (a → b) → (((ListF c) a) → ((ListF c) b))
  fmap f = λcase
    NilF → NilF
    ConsF c a → ConsF c (f a)
```

If you are unfamiliar with functors: In our case regarding them as a computational context with holes suffices.

- The structural functors encode where the recursion should happen
- Relation to original datatype not yet wholly clear
- $\rightsquigarrow$ Introduce a little Category Theory

Structure

Endofunctors

Let $\mathcal{C}$ be a Category. An *Endofunctor* is a pair of maps, on the objects and morphisms of the category respectively: $F_0 : \mathcal{C}_0 \to \mathcal{C}_0$, $F_1 : \mathcal{C}_1 \to \mathcal{C}_1$ Such that:

- $F_1(h : A \to B) : F_0 A \to F_0 B$
- $F_1(id_A) = id_{F_0 A}$
- $F_1(h \circ g) = (F_1 h) \circ (F_1 g)$

- `fmap @f (h :: a → b) :: f a → f b`
- `fmap @f (id @a) = id @(f a)`
- `fmap @f (h . g) = fmap @f h . fmap @f g`

The category we are regarding is *Hask*, where objects are Haskell types, and morphisms are functions between them. This is usually interpreted as a *CPO* (complete partial order). We will mention what notions are specic to *CPO*s.

Algebras

Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor $A_0 \in \mathcal{C}_0$, $\phi : A \to FA$. Then $A \xrightarrow{\phi} FA$ is an *Algebra*, and A its *Carrier*. We can phrase the business logic of the previously seen functions as such (using the transformation $A^B \times A^C \sim A^{B+C}$):

```
type Algebra f a = f a → a
boolBL :: b → b → Algebra BooLF b
boolBL tt ff = λcase
  TTF → tt
  FFF → ff
listBL :: b → (a → b → b) → Algebra (ListF a) b
listBL nil cons = λcase
  NilF → nil
  x `ConsF` b → x `cons` b
```

To conclude this exercise, we want to generalize the function for recursively applying the business logic (So a function subsuming `list` and booL seen earlier): `list :: b → (a → b → b) → List a → b` `booL :: b → b → BooL → b` We define a type family to associate the structural functors with their types.

```
type family CI (f :: * → *) :: *

type instance CI (ListF a) = List a
type instance CI (BooLF) = BooL
```

Then our that function would have type `cata :: UnwrapCI f => Algebra f b → CI f → b` . To get to its definition, we must interleave some more Cat.Th.

```
cata :: UnwrapCI f => Algebra f b → (CI f) → b
cata ψ = unwrap .> fmap (cata ψ) .> ψ
```

Algebra

$$\begin{array}{c} FA \\ \downarrow{\scriptstyle \phi} \\ A \end{array}$$

Algebra-Hom:

$$(A, \phi) \to (B, \psi)$$

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ \downarrow{\scriptstyle \phi} & \circlearrowleft & \downarrow{\scriptstyle \psi} \\ A & \xrightarrow{f} & B \end{array}$$

Initial Algebra: (A, κ)

$$\begin{array}{ccc} FA & \overset{Fh}{\dashrightarrow} & FB \\ {\scriptstyle \kappa^{-1}}\uparrow \downarrow{\scriptstyle \kappa} & \circlearrowleft & \downarrow{\scriptstyle \psi} \\ A & \overset{h}{\dashrightarrow} & B \end{array}$$

Legend: $-\!\!\to$: $\exists!$, $\begin{array}{c} \xrightarrow{f} \\ {\scriptstyle f'}\downarrow \circlearrowleft \downarrow{\scriptstyle g} \\ \xrightarrow[g']{} \end{array}$: $f; g = f'; g'$

Initiality requirement: $h = \kappa^{-1}; Fh; \psi$

h has exactly the type we want for `cata` ψ, when $CI\ F = A$ (CI stands for "Carrier Initial").

The initiality requirement gives us a function definition, we just still need κ^{-1}.

```
class Functor f => UnwrapCI f where
  unwrap :: (CI f) → f (CI f)
instance UnwrapCI BooLF where
  unwrap :: BooL → BooLF BooL
  unwrap = λcase
    TT → TTF
    FF → FFF
instance UnwrapCI (ListF a) where
  unwrap :: List a → ListF a (List a)
  unwrap = λcase
    Nil → NilF
    x `Cons` xs → x `ConsF` xs
```

As Program

```
newtype Fix f = In { out :: f (Fix f) }
```