## Recursive types for free! in Haskell<sup>1</sup>

Traian Florin Şerbănuță

University of Bucharest and Runtime Verification

<sup>&</sup>lt;sup>1</sup>Based on Philip Wadler's Recursive types for free!

## GHC language extensions used

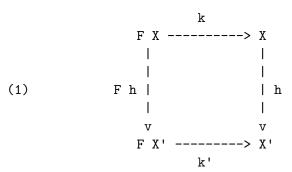
```
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE ExplicitForAll #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE Rank2Types #-}
```

#### Definitions of f-algebras and f-coalgebras

An F-algebra is a pair (X,k) consisting of an object X and an arrow  $k : F X \rightarrow X$ .

type Algebra 
$$f x = f x \rightarrow x$$

A morphism between (X,k) and (X',k') is given by an arrow  $h: X \to X'$  such that the following diagram commutes.



These form a category.

#### Definition of f-coalgebras

An F-coalgebra is a pair (X,k) consisting of an object X and an arrow  $k: X \rightarrow F X$ .

type 
$$CoAlgebra f x = x \rightarrow f x$$

A morphism between (X,k) and (X',k') is given by an arrow  $h: X \to X'$  such that the following diagram commutes.

These form a category.

## Least-fixpoints as (weak) initial algebras

```
newtype LFix f =
  LFix { unLFix :: (forall x . Algebra f x -> x) }
```

- LFix f embodies the idea of a type for terms associated to f.
- A term can be (uniquely) evaluated in any algebra.
- A term gives, for an algebra, a value for the term in the algebra.
- ▶ Whence the type for a term: forall x . Algebra f x -> x

```
Least-fixpoints as (weak) initial algebras
   newtype LFix f =
     LFix { unLFix :: (forall x . Algebra f x -> x) }
                             wInitialAlg
                  f (LFix f) -----> LFix f
   fmap (fold algebra) |
                                             fold algebra
                               algebra
   fold :: Algebra f a -> LFix f -> a
   fold algebra term = unLFix term algebra
   weakInitialAlgebra :: Functor f => Algebra f (LFix f)
   weakInitialAlgebra s =
     LFix ( \alg -> alg (fmap (fold alg) s) )
```

## Morphism condition for fold algebra

```
wInitialAlg
               f (LFix f) -----> LFix f
fmap (fold algebra) |
                                         fold algebra
                            algebra
(fold algebra . weakInitialAlgebra) fterm
== fold algebra (weakInitialAlgebra fterm)
== unLFix (LFix (\alg -> alg (fmap (fold alg) fterm)))
   algebra
== algebra (fmap (fold algebra) fterm)
== (algebra . fmap (fold algebra)) fterm
```

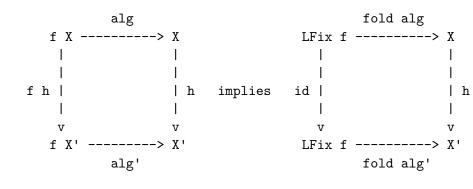
## Natural numbers as a least fix point

```
data NatF x = Zero | Succ x deriving Functor
type Nat = LFix NatF
zero :: Nat
zero = weakInitialAlgebra Zero
successor :: Nat -> Nat
successor n = weakInitialAlgebra (Succ n)
one :: Nat
one = successor zero
integral :: Integral n => Algebra NatF n
integral Zero = 0
integral (Succ x) = x + 1
natToIntegral :: Integral n => Nat -> n
natToIntegral = fold integral
```

### Lists as a least fix point

```
data ListF a x = Nil | LCons a x deriving Functor
type List a = LFix (ListF a)
nil :: List a
nil = weakInitialAlgebra Nil
cons :: a -> List a -> List a
cons a l = weakInitialAlgebra (LCons a l)
list :: Algebra (ListF a) [a]
                = []
list Nil
list (LCons a 1) = a:1
toList :: List a -> [a]
toList = fold list
```

## When is the least fix point actually initial?



h: (X,alg) -> (X',alg') implies h . fold alg == fold alg

Additionally, fold weakInitialAlgebra == id

# Initiallity consequences

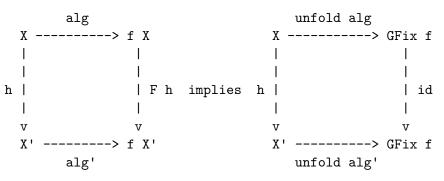
If (LFix f, weakInitialAlgebra) is initial, then
weakInitialAlgebra is an isomorphism and its inverse is:

```
weakInitialAlgebraInv :: Functor f => CoAlgebra f (LFix f)
weakInitialAlgebraInv = fold (fmap weakInitialAlgebra)
             f (LFix f) -----> LFix f
                        wInitialAlg
f wInitialAlgInv
                                         wInitialAlgInv
                        f wInitialAlg v
           f (f (LFix f)) -----> f (LFix f)
  f wInitialAlg
                                         wInitialAlg
                        wInitialAlg
            f (LFix f) -----> LFix f
```

# Greatest fix points as (weak) final co-algebras

```
data GFix f = forall x . GFix (CoAlgebra f x, x)
                    coalg
             X -----> f X
unfold coalg
                                f (unfold coalg)
          GFix f -----> f (GFix f)
                 wFinalCoalg
unfold :: CoAlgebra f a -> a -> GFix f
unfold coalg a = GFix (coalg, a)
weakFinalCoAlgebra :: Functor f => CoAlgebra f (GFix f)
weakFinalCoAlgebra (GFix (coalg, a)) = fmap (unfold coalg)
```

## When is the greatest fix point actually final?



weakFinalCoAlgebraInv :: Functor f => Algebra f (GFix f)
weakFinalCoAlgebraInv = unfold (fmap weakFinalCoAlgebra)

# Streams as a greatest fix point

```
data StreamF a x = SCons { headF :: a, tailF :: x }
   deriving Functor
type Stream a = GFix (StreamF a)
headS :: Stream a -> a
headS = headF . weakFinalCoAlgebra
tailS :: Stream a -> Stream a
tailS = tailF . weakFinalCoAlgebra
stream :: CoAlgebra (StreamF a) [a]
stream (a:as) = SCons a as
toStream :: [a] -> Stream a
toStream = unfold stream
type IStream a = LFix (StreamF a)
icons :: a -> IStream a -> IStream a
icons a s = weakInitialAlgebra (SCons a s)
```

### Introducing the recursion schemes Fix construction

```
newtype Fix f where
 Fix :: f (Fix f) -> Fix f -- Fix is an f-algebra
unFix :: Fix f -> f (Fix f) -- unFix is an f-coalgebra
unFix (Fix x) = x
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = go
 where
   go = alg . fmap go . unFix
ana :: Functor f => CoAlgebra f a -> a -> Fix f
ana coalg = go
 where
   go = Fix . fmap go . coalg
```

### Relating Fix with LFix and GFix

```
lFixToFix :: LFix f -> Fix f
lFixToFix = fold Fix
fixToLFix :: Functor f => Fix f -> LFix f
fixToLFix = cata weakInitialAlgebra
fold Fix . cata weakInitialAlgebra
= Fix . fmap (fold Fix . cata weakInitialAlgebra) . unFix
fixToGFix :: Functor f => Fix f -> GFix f
fixToGFix = unfold unFix
gFixToFix :: Functor f => GFix f -> Fix f
gFixToFix = ana weakFinalCoAlgebra
```

#### Read More

- ▶ Philip Wadler (1990) Recursive types for free!
- ▶ Bartosz Milewski (2013) Understanding F-Algebras
- ▶ Bartosz Milewski (2017) F-Algebras
- A formalization of the above in Coq with actual proofs