

Language Enginering Report

A look into Catamorphisms and their role in parser design



Alex robinson

ar15247@my.bristol.ac.uk

# LANGUAGE ENGINERING REPORT

ALEX ROBINSON

Abstract. TODO

## 1. Introduction

1.1. Introduction. Brief introduction to category theory with the aim to understand vocabulary, move onto F-Algebras and then Catamorphisms with practical examples in Haskell. Then move onto designing a small combinatorial parsing library similar to parsec in Haskell. Implementation of a small BNF showing how BNF translates to parser combinators.

## 2. Category Theory, F-Algebras, and Catamorphisms

2.1. Category Theory. Category Theory (CT) is a generalisation of how various structural concepts are described in mathematics such as groups and rings. A Category is composed of:

* A collection of objects,
* A collection of morphisms, which relate to a pair of objects, . is known as the type and f is a morphism from source to target
* A composition on morphisms denoted

2.1.1. Axioms of categories:

1. Composition of morphisms is associative
2. Categories are closed under composition
3. Every object has an identify morphisms such that

Some objects represent empty in some sense for example is the empty set, these objects are unique i.e. there exists only one empty set. In CT this unique trait is called *initially*, an object is initial if for any object in a mapping there exists a single mapping from it to the object. There can only be only one initial in a category.

2.2. Functors. A homomorphism is a mapping between categories, a functor is a homomorphism preserving categorical structure. Given categories and we define a functor which can be applied to any object or morphism in to map into for example can be mapped to or becomes .

2.2.1. Axioms of functors:

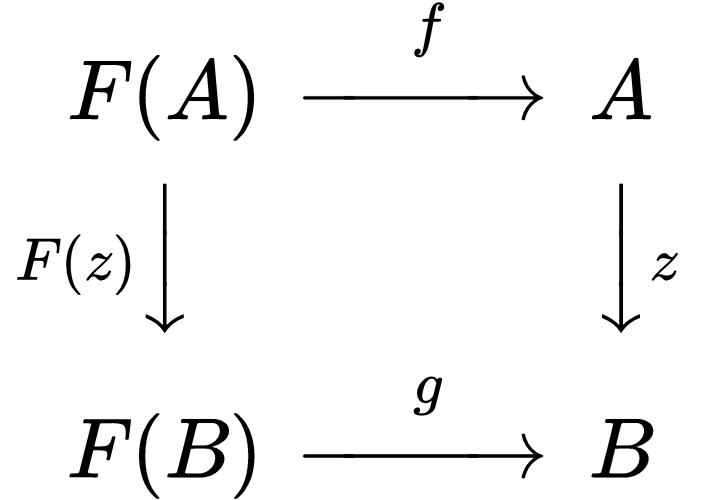
1. Given an identify morphism and object ,
2. Functors distribute over morphism composition

Axiom *2.* is practically useful when you imagine the functors as a list or similar container, the RHS is a two pass algorithm: mapping over structure applying g, and then again applying f. This axiom guarantees this can be transformed this into a single-pass algorithm performing . This is known as fusion.

2.3. F-algebras. Natural numbers can be represented as the recursive structure:

data Nat = Zero | Succ Nat

With this three can be represented as Succ(Succ(Succ Zero)). Nat is a category, and operators on natural numbers such as addition are functors which map Nat into itself, these are known as *endofunctors*. However, in order to transform a Nat into an integer a functor with signature , where is either an integer with addition or Nat.

This is an F-algebra. Let be a category with objects and an endofunctor . Let be a functor mapping to . There exists two F-algebras and .

Paths from to are equal, so . As the identity functor is a homomorphism and composing homomorphism produce a homomorphism this means F-algebras form a category of F-algebras. For the general class of functor, this category has an initial object (labelled above) which is the distinct functor from the initial algebra to any other F-algebra, this is called the *catamorphism*.

2.4. Catamorphisms and Fixed Point Recursion. Catamorphisms act like a generalisation of the fold on lists, it computes a value from a container like structure and a function. The program bellow demonstrates the practical use of catamorphisms in programming.

TODO: Talk about practical applications of catamorphism + fixed point recursion + move program to appendix and only use small example, reference data type created above!

type Algebra f a = f a -> a

-- Fix is FixedPoint

-- Fix also know as Mu

data Fix f = Fix {unFix :: f (Fix f)}

deriving instance (Show (f (Fix f))) => Show (Fix f)

cata :: Functor f => Algebra f a -> Fix f -> a

cata f = f . fmap (cata f) . unFix

data Nat n = Zero | Succ n deriving Show

instance Functor Nat where

fmap \_ Zero = Zero

fmap f (Succ n) = Succ (f n)

out :: Fix Nat -> Int

out = cata f where

f :: Nat Int -> Int

f Zero = 0

f (Succ n) = n + 1

in' :: Int -> Fix Nat

in' 0 = Fix Zero

in' n = Fix (Succ (in' (n-1)))

oneMore :: Fix Nat -> Fix Nat

oneMore = cata f where

f :: Nat (Fix Nat) -> Fix Nat

f Zero = Fix (Succ (Fix Zero))

f (Succ n) = Fix (Succ n)

double :: Fix Nat -> Fix Nat

double = cata f where

f :: Nat (Fix Nat) -> Fix Nat

f Zero = Fix Zero

f (Succ n) = Fix (Succ (Fix (Succ n)))

add :: Fix Nat -> Fix Nat -> Fix Nat

add a = cata (f a) where

f :: Fix Nat -> Nat (Fix Nat) -> Fix Nat

f b Zero = s b

f \_ (Succ n) = Fix (Succ n)

mul :: Fix Nat -> Fix Nat -> Fix Nat

mul a = cata (f a) where

f :: Fix Nat -> Nat (Fix Nat) -> Fix Nat

f \_ Zero = Fix Zero

f \_ (Succ n) = add a n

-- Is there a formal name for this?

s :: Fix Nat -> Fix Nat

s = cata f where

f :: Nat (Fix Nat) -> Fix Nat

f Zero = Fix Zero

f (Succ n) = Fix (Succ n)

2.4. Category Theory oriented Programming. Focus on type and operators, extreme generalisation, better modularity, better control through properties of types. TODO

## 3. Building a monadic parser

## 4. Conclusion

## References

## Appendix