Scheming it all.

Recursion Scheme in Domain Specific Languages.

Abstract

In this paper we will introduce structure recursion as a way to derive program semantics. In particular, we are interested in denotational semantics which can be structured as a fold. This motivates us to look its generalised notion as a recursive operation: the *catamorphism.*

1 Introduction

Abstraction has proved to be one of the most influential and ubiquitous themes in computer science. It is unsurprising that one of its most notable triumphs has been its significance in the design of programming languages. The most successful will provide various techniques for abstraction at the software level such as higher-order functions, objects etc.

General Purpose Languages ("GPL") are programming languages designed to tackle problems in every problem domain. Their generality pushes programmers to adopt them in their repetoire, however this makes them convoluted to anyone lacking programming expertise. This motivates the idea of raising the abstraction level such that languages are designed for a particular problem domain: it is *domain specific.*

Domain Specific Languages ("DSL") are, usually declarative, programming languages that offer more expressivity over a specific problem domain. Their development involves a close analysis of the problem domain such that the entire semantics of the problem domain should be captured - no more and no less. The nature of these languages implies that they trade their generality for focused expressivity. This often makes DSLs small languages since only the essential features are captured, examples of DSLs include SQL, HTML, CSS etc.

There are two main ways of developing DSLs:

- Standalone.
- Embedded.

Standalone is a classical approach for implementing a new language. It involves implementing the language from scratch: everything that you need in a language has to be developed, no concessions are made. Its characteristics are tailored specifically to the problem domain. However, the drawback to this approach is that it is very costly, as a result standalone DSLs are rarely developed.

Embedded DSLs are implemented by extending a GPL, this approach uses the existing language constructs to build the language. They share the generic features of the base language thus the embedded DSL offer addition power of its base GPL as well as their domain specific expressive power. Embedded DSLs often extends functional languages - features part of this class of languages such as higher order functions, monads, algebraic data types makes the develop of embedded DSL much easier.

The nature of functional languages, especially Haskell, have a strong emphasis on maintaining the *purity* of its code. With no state or side-effects, many computation are naturally expressed as recursive functions. Unsurprisingly, many of which share the same recursive patterns which can be abstracted away. An example that many functional programmers will be familiar with is *fold* as a standard recursive operator, it captures the common pattern of traversing and processing a structurally inductive data structure. The abundant usage of folds is so extensive, in fact, the denotational semantics, an approach that gives mathematical models to the semantics of a program, can be structured and characterised by folding over its syntax [1]. This is why we can fold DSL with great success [2].

This motivates us to look closely at the generalisations of folds and unfolds as a set of combinators introduced by Meijer et al called *recursion schemes.* Category theory was crucial in its development, it provided key concepts such as algebras and functors as clean way to formalise the structure of traversing and evaluation of inductive data structures.

The structure of the report is as follows:

1. A brief introduction to Category Theory - Many ideas in Haskell have originated from category theory and arguably the reason why functional programming is so successful. This section introduces the necessary knowledge for understanding the report.
2. Explicit and Structure Recursion - We will begin by discussing the drawbacks of explicit recursion and why we should always use structured programming if possible.
3. Pattern and Fix Point of Functors - Given a recursive data structure we will show how to abstract away recursion at the type level.
4. Recursion Schemes - The generalised fold. We will discuss its derivation, useful theorems and termination.

2 Introduction to Category Theory

Category theory is one of the most abstract theory in mathematics, it is a study of mathematical structure. Its abstractness suggests it has more form than content which perhaps is one of its strengths, its general nature allows it to be applied to many areas of computer science: from the design of programming languages to automata theory [reference here?].

A *category* can be thought of as a family of mathematical structure coupled with the idea of structure preserving maps. It captures the idea composition in its definition which is, arguably, the nature of programming.Unsurprisingly, it can be used a model of computation i.e. in Haskell, `Hask` forms a category, where its types and functions can be modelled as a *category*, where the types form the objects and functions between two types, the morphism.

But what is a category?
A category C is an algebraic structure defined on a collection of:

- objects (denoted $A, B, C, ...$)
- morphism between a pair A, B denoted $C(A, B)$

additionally, they must satisfy:

- for each object A, there is an identity morphism.
- the morphisms are associative.
- for every object A, B, C a binary operation called the *composition of morphisms.*

Morphisms can be thought of as special functions between objects that preserves its structure satisfying associativity, composition and the existence of an identity for every object. Because of this, many category theorists believe that the morphisms are of greater importance than objects because they reveal the true underlying structure.

It is natural to consider a structure preserving map similar idea to morphism but for categories. The functor is a mapping between categories but with additional properties so that the categorical structure is preserved.
It is formally, a functor $F : C \to D$ consists of:

- mapping $A \to FA : C \to D$
- mapping $f \to Ff : C(A, B) \to D(FA, FB)$

such that:

- $Fid = id$
- $F(g \circ f) = Fg \circ Ff$

These additional laws preserve the nature of a morphism in the category by respecting the identity and the composition of morphisms laws.

An endofunctor is an functor from a category to itself. In Haskell, the definition of a functor corresponds with the categorical endofunctor, defined as follows:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The additional properties that a categoric functor must satisfied, is captured in Haskell by the functor laws, that each and every instance must satisfy:

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

There is one category of particular useful interest, the category of F-algebras. Given a category C and an endofunctor $F : C \rightarrow C$ then an F-algebra is a tuple (A, f) where,

- A is an object in C
- f is a morphism $F(A) \rightarrow A$.

The object A is called the *carrier* of the algebra.

F-algebras can be used to represent data structures such as tree and lists.

These algebras will form the objects of the category. However, it needs to be coupled with morphisms to form a category. In this case, the morphisms is the homomorphism between F-algebras.
A homomorphism an F-algebra (A, a) to another F-algebra (B, b) is a morphism $C(A, B)$ such that $f \circ a = F(f) \circ b$.

In Haskell, the definition following definition is found in `Control.Functor.Algebra` to corresponds to the idea of an F-algebra.

```
type Algebra f a = f a -> a
```

The *initial algebra* is an F-algebra where there exists exactly one morphism from the initial algebra to all other algebras. Initial algebras are interesting for computer scientist which will be seen in Section 4.

3 Explicit and Structured Recursion

Recursion in its essence is something defined in terms of itself. It is a simple yet powerful concept that forms the bread and butter for functional computation. Explicit recursion is a way of describing self referencing functions that is overused for the uninitiated. Arbitrary properties of the function will need to be written and proved over and over again which can be simply avoided by carefully abstracting away common recursive patterns.

Its profuseness implies that by abstracting away common patterns, it could replace a plethora of explicit recursive functions. Meijer et al introduced a set of recursive operators that captures different types of recursion. The catamorphism models iteration which is a special case of primitive recursion which is modelled by the paramorphism. Meijer also introduced its duals for unfolds and corecursion, anamorphism and apomorphism. What is surprising perhaps, is that like the folds, unfolds can be used to structure and derive a type of program semantics called operational semantics [1] where the meaning of the program is defined in terms of transition functions during program execution. However, this lies outside the scope of this report.

We have known for a long time the use of `gotos` in imperative programming obscures the structure of the program and reduces the programmers ability to reason with their code. For the same reason `gotos` should be avoided, we should always use structured recursion whenever possible. This is because although explicit is more intuitive, structural recursion provides a way to reason with the programmer's code like never before. They provide us with a catalogues of useful theorems and properties which we can infer in our functions for free. Additionally, as a byproduct of abstracting away the format of traversals, it separates how the function is computed rather than its underlying purpose. This means for programmers, trained in the art of structuring recursion, can concentrate on what the computation is doing rather than how.

4 Hiding (explicit) recursion

In Section 3, we have detailed the differences between explicit and structured recursion and explained that whenever there is a choice between structured and explicit recursion, structured recursion should always be used.

This section, we will give the steps for abstracting away recursive algebraic data types at its type level by using the concept of a fixed point.

4.1 Parameterising Recursion

Let's consider a very simple language of addition and subtraction.

```
data Expr = Val Int
          | Add Expr Expr
          | Sub Expr Expr
```

We can have parameterised the recursion, by marking the recursive spot `x`:

```
data ExprF x = Val Int
             | Add x x
             | Sub x x
```

In the new definition of `Expr`, `ExprF`, we have parameterised this type in terms of its subexpression, this is called the *pattern functor* which is almost identical to the original `Expr`.

It is trivial to make ExprF an instance of `functor`.

```
instance Functor ExprF where
  fmap :: (a -> b) -> f a -> f b
  fmap f (Val Int) = Int
  fmap f (Add x y) = Add (f x) (f x)
  fmap f (Sub x y) = Sub (f x) (f y)
```

In fact, it is so trivial that GHC can derive it for us if we enable the following extension, called a language pragma:

```
{-# LANGUAGE DeriveFunctor #-}
```

and we can just use `Deriving Functor` in our data declaration.

However, `ExprF` is not quite equivalent, it need to somehow arbitrarily nest `ExprF` in the definition.

4.2 Fix Point of Functors

In lambda calculus, it is not possible to refer to the function definition in its body; there is no feature for (explicit) recursion. However, by using the paradoxical Y combinator, we can replicate recursive behaviour. It is, by definition, a higher-order function, f, that takes a non-recursive function that satisfies the following:

$$yf = f(yf)\forall f$$

This concept can be defined in Haskell's type definition as follows:

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

By using `Fix`, we can define our corresponding pattern functor in such a way, called the fixed point of functors, which is isomorphic to the original definition,

Fix ExprF ~ Expr

This technique of redefining recursive data types is very powerful. Interestingly [4], the fixed point of functors corresponds to the initial algebra, an F-algebra (Fix f, In).

5 Recursion Schemes

Recursion schemes has risen from attempts to tame the unyielding power of recursion, as a results, there is now a large zoo of formalised recursive operators that captures different types of recursion. There has been attempts to unify these schemes [citations here]; interesting in its own right, but outside the scope of this paper.

The nature of denotational semantics can be structured as a fold [1] which in the zoo of recursion schemes is called the catamorphism, It is for this reason, we are interested in this particular scheme and it will be the main focus of this section. References might be made to other recursive operators to provide the idea that these schemes are not restricted to recursion. They can be corecursive, the dual of recursion, which generates data. And refolds - combination of both recursion and corecursion.

5.1 Catamorphism

Catamorphism are generalisations of folds, it replicates the behaviour of iterative functions by destroying the data structure while traversing it.

5.2 Derivation

$$\begin{tikzcd}
F(FixF) \arrow[r] \arrow[d, "fmap(cata \circ alg)"'] & FixF \arrow[d, "cata \circ alg"] \\
F(A) \arrow[r, "alg"'] & A
\end{tikzcd}$$

The catamorphism takes the starting data type in the form the initial algebra $(FixF, in)$ for some endofunctor F which, in this case, is represented by a pattern functor. Since it is initial, there exists a unique F-algebra homomorphism to some arbitrary algebra (A, alg) in the category of F-algebras for endofunctor F. This will be our catamorphism.

This is can be implemented in Haskell as follows:

```
 cata :: Functor f => Algebra a -> Fix f -> a
 cata alg = alg . fmap (cata alg) . out
```

For example, consider the natural numbers

```
type Nat = Fix NatF
data NatF k = Zero
             | Succ k
             deriving Functor

number :: Nat -> Int
number = cata alg
  where alg (Zero)   = 0
        alg (Succ k) = k + 1
```

{Explain}

5.3 Theorems

By using catamorphism, one of the many forms of structural recursion, one of the most pleasant results is that the catalogue of laws[4] can be utilised for free!

Fusion

Fusion law for catamorphism [4] allows a composition of functions with a catamorphism to transformed to a single catamorphism. This will eliminate all intermediate data types and is arguable one of the most important laws.

```
h . f = g . fmap h => h . cata f = cata g
```

where,

```
f :: f a -> a
g :: f b -> b
h :: a -> b
```

See Appendix. The example given is one of pretty printing functions are called "prettyFast" and "prettySlow".

Composition

It is not true generally that catamorphisms compose but there is a special case. The compose law [4] implies that the number of traversals required by a function that satisfies the special case can be reduced which theoretically speeds up performance.

It states that:

```
cata f . cata (Fix . h) = cata (f . h)
```

where,

```
f :: f a -> a
h :: g a -> f a
```

See Appendix.

Banana split theorem

Algebras that are over the same functor but with different carrier types can be combined. This means that more than one catamorphism can be performed at the same time. This is called the banana-split theorem [4] which states that:

```
cata f &&& cata g = cata ( f . fmap fst &&& g . fmap snd )
```

{example}

4.4 Program Termination

As we have seen, by using the simplest example of a recursion scheme, we have an archive of extremely useful laws that we obtain for free. Another byproduct of using certain recursion schemes is that it gives us the ability to reason with the termination of the program.

Catamorphism gives us the ability to guarantee its termination. The function calls made are only on smaller elements of the inductively defined structure implying it will tend towards its base case, giving us termination. This is also true for the paramorphism. However, with explicit recursion, there is nothing to stop the programmer to recursively call the function on larger data type causing it to never terminate, we are unable to reason with its termination.

In the recursion schemes provided by Meijer et al., the property to guarantee of termination is exclusive to the para and catamorphism. Conversely, the ana and apomorphism guarantees co-termination - it will keep producing data. The hylomorphism is an interesting recursion scheme in its own right, it consists of the composition of cata and anamorphism which captures general recursion - a more powerful type of recursion than primitive. It also gives us turing completeness thus losing all guarantees of termination.

6 Conclusion

In this report, I have given a brief introduction to category theory. Concepts introduced are essential for giving us a streamlined derivation of the catamorphism, just one of many from the zoo of recursion schemes.

We have discussed why structured recursion should always be used if possible over explicit recursion. Its strength lies in the properties that we can imply by using them as well as empowering our ability to reason with our code. Thus, I have given what I perceive to be the most useful.

References

[1] Fold and Unfold for Program Semantics.
[2] Folding DSL: Deep and Shallow Embedding.
[3] Functional programming with bananas, lenses, envelopes and barbed wire.
[4] Recursive types for free.
[5] Building Domain Specific Embedded Language.