Scheming it all.

Recursion Scheme in Domain Specific Languages.

Abstract

This report discusses structured recursion as a way to derive program semantics. In particular, the denotational semantics which can be structured as a fold motivates a close examination at 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 its entire semantics should be captured - no more and no less. The nature of these languages imply 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 to develop and maintain, as a result standalone DSLs are developed when absolutely needed.

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 the addition power of its base GPL as well as their domain specific expressivity. Embedded DSLs often extends functional languages - features that is 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 computations are naturally expressed as recursive functions. Unsurprisingly, many of which share the same recursive pattern which can be abstracted away. An example that many functional programmers will know and love is *fold* a standard recursive operator, it captures the common pattern of traversing and processing a structurally inductive data structure. The abundant usage of folds is 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 DSL can be folded with great success [2].

This motivates us to look closely at the generalisations of folds as a set of combinators introduced by Meijer et al called *recursion schemes*. The authors used key concepts from Category theory such as algebras and functors for a clean way to formalise the structure of traversing and evaluation of recursive data structures.

The structure of the report is as follows:

1. A brief introduction to Category Theory - Many ideas in Haskell have origins in category theory and arguably the reason why functional programming is so successful. This section introduces the necessary knowledge for understanding the simple yet elegant derivation of the catamorphism.
2. Explicit and Structure Recursion - Discussing the drawbacks of explicit recursion and why structured recursion should always be used if possible.
3. Pattern and Fix Point of Functors - Given a recursive data structure the explicit recursion can be abstract away recursion at the type level.
4. Recursion Schemes - Since denotational semantics can be characterised by folds, which is captured in the set of recursion schemes as the catamorphisms. This report will show its derivation, theorems.
5. Termination Property - Catamorphism, unlike explicit recursion, allows the programmer to reason with the termination of the program. This section will explain the termination property of a small subset of recursion schemes.

2 Introduction to Category Theory

Category theory is the study of mathematical structure infamous for being one of the most abstract theories in mathematics. Its generality 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 structures coupled with the idea of structure preserving maps. It captures the idea composition in its definition which is, arguably, the nature of computation. Unsurprisingly, a category can be used a model of computation i.e. in Haskell, its types and functions can be modelled as a *category* called `Hask`, where the types form the objects and functions between two types, the morphisms.

Formally, 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 its morphisms can be composed called the *composition of morphisms*.

Morphisms can be thought of as special functions between objects that preserves 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 $A \to F(A) : C \to D$
- mapping $f \to F(f) : C(A, B) \to D(FA, FB)$

such that:

- $F\ id = id$
- $F(g \circ f) = F(g) \circ F(f)$

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

In Haskell, the definition of the functor class corresponds to the categorical endofunctor which is a functor from a category to itself. The functor class is defined as follows:

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

The additional properties that a categoric endofunctor 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
```

Functors are hidden and prevalent in functional programming. It captures the theme of compatibility between categories of data types and allows for function reusability by "promoting" it from type `a -> b` to `f a -> f b` over the functor instance `f`. The functional programmer can write their code in whichever category that is most appropriate and "lift" it with `fmap` to be applied to a different category.

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

- A is an object in C is called the *carrier* of the algebra.
- f is a morphism $F(A) \to A$.

A homomorphism from a F-algebra (A, α) to another F-algebra (B, β) is a morphism $C(A, B)$ such that:

- $f \circ a = F(f) \circ b$.

$$\begin{array}{ccccc} & F(A) & \underset{\alpha}{\to} & A & \\ F(f) & \downarrow & & \downarrow & f \\ & F(B) & \underset{\beta}{\to} & B & \end{array}$$

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

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

The F-algebra is an *initial F-algebra* when there exists exactly one morphism from the initial F-algebra to all other F-algebras. Initial algebras is an interesting concept which can be used to model our data type.

The category of F-algebras constitutes F-algebra as objects and the F-algebra homomorphisms, the morphisms. The category can be used to model how the data structure i.e. lists or trees can be transformed. The initial algebra of the category corresponds to our initial data structure and the algebras corresponds to the function that can be performed on the data structure.

It is interesting to see that the category of `Hask` does NOT form a category [Citation here]. However, it does not need to perfectly correspond to the formal definition of a category for language engineers to use it as a way to solve practical

problems. For example, monads, a concept in category theory, is used in Haskell as a way to program imperatively: allowing for unsafe IO, state etc. As long as the model of computation, a category, provides intuition and various categorical abstractions as a way to solve problems, it should be embraced.

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 explicitly recursive function will need to be theorised and proved over and over again which can be simply avoided by carefully abstracting away the 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 combinators that captures different types of recursion. The catamorphism captures the concept of iteration - a special case of primitive recursion which is captured 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.

It has been 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[citation]. 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 allows the programmer to reason with the their code like never before. In addition, there is a catalogue of useful theorems and laws which can used to improve each function that utilises structural recursion for free. Moreover, as a byproduct of abstracting away the recursive patterns, it separates how the function is computed from 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 it is done.

4 Hiding (explicit) recursion

This section will focus on 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
```

The recursive spot in `Expr` can be parameterised with `x` producing a near identical data type:

```
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`.

Notice that 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:

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

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,

$$FixExprF \cong Expr$$

This technique of redefining recursive data types is very powerful. Interestingly [4], the fixed point of functors corresponds to the initial algebra: the 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].

Denotational semantics can be structured as a fold [1] which in the zoo of recursion schemes is called the catamorphism. It motivates a close examination of 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 limited to recursion. They can be corecursive, the dual of recursion, which generates data. And refolds which uses a 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

$$\begin{array}{ccc} F(FixF) & \underset{In}{\rightarrow} & FixF \\ \downarrow & & \downarrow \\ F(A) & \underset{alg}{\rightarrow} & A \end{array} \quad \begin{array}{l} \\ fmap(cataalg) \\ \\ \end{array} \quad \begin{array}{l} \\ cataalg \\ \\ \end{array}$$

The catamorphism takes the data type in the form the initial algebra $(FixF, in)$ for some endofunctor F which represents the 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 the pattern functor F. The catamorphism corresponds to the observation of this homomorphism from the initial algebra to some algebra.

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
```

In the `number` function, `Nat` corresponds to the initial algebra, to be transformed by the algebra which in this case is defined within the scope of the function. Notice that this method of defining the function has no explicit recursion.

5.3 Theorems

By using catamorphisms, one of the many forms of structural recursion, one of the most pleasant results is that the catalogue of laws[3] can be used for free!

Fusion

Fusion law for catamorphism [3] 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 [3] 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
```

This theorem can be used to optimise code. 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 [3] which states that:

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

(&&&) :: (a -> b) -> (a -> c) -> (a -> (b , c))
f &&& g = \x -> (f x, g x)
```

See appendix.

5 Program Termination

As we have seen, by using the simplest example of the recursion schemes, we have an archive of extremely useful laws that is obtained 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 are made only on smaller elements of the inductively defined structure implying it will tend towards its base case, giving program termination. This is also true for the paramorphism. With explicit recursion, there is nothing to stop the programmer to recursively call the function on larger data type causing it to never terminate.

In the recursion schemes provided by Meijer et al., the property to guarantee termination is exclusive to the para and catamorphism. Conversely, the ana and apomorphism guarantees co-termination - it will keep producing data.

6 Conclusion

The purpose of this report was to show a in depth analysis of the catamorphism as an example from the set of recursion schemes. Additional, simple concepts in category theory was introduced motivated by the idea that many concepts in functional programming have origins in this theory. This allowed for a streamlined derivation of the catamorphism and important theorems were shown.

Though only the catamorphism was shown, it is quite restrictive and forms only a small part of the large class of recursion schemes. Since Meijer, more recursion schemes have been introduced to capture more variations of recursion - some more useful than others. This growth has turned the set into a zoo of morphisms with attempts to unify them.

However powerful these recursion schemes are, they are not turing complete. This is a property that (most) GPLs have but may not be necessary for DSLs. In general, it is true that DSLs are not and incorporating it as part of the language is unneeded. It will introduce unnecessary complexities in the language which is exactly what DSLs avoiding.

In the future, a similar approach for this report can be taken but with generalised unfolds e.g. anamorphism to derive and structure operational semantics.

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.