Scheming it all

Recursion Scheme in Domain Specific Languages - Jenning Lim

Abstract

This report discusses structured recursion to derive program semantics. In particular, the denotational semantics which can be structured as a fold motivates a close examination of 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 have 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 repertoire, however, this aspect 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: they are *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 so 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, LaTe 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 must 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 with its base language thus the embedded DSL offer the additional power of its base GPL, as well as their domain specific expressivity. Embedded DSLs often extend functional languages - features that are part of this class of languages such as higher order functions, monads, algebraic data types make the development of embedded DSL much easier (Hudak 1996).

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 use of folds is prevalent in functiuonal programming, 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 (Hutton 1998). This is why DSLs can be folded with great success (Gibbons and Wu 2014).

This motivates us to look closely at the generalisations of folds as a set of combinators introduced called *recursion schemes* (Meijer, Fokkinga, and Paterson 1991). The authors used key concepts from Category Theory such as algebras and functors for a clean way to formalise the structure of traversing and evaluating 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, which is 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. Recursion Schemes - Since denotational semantics can be characterised by folds, which is captured in the set of recursion schemes as the catamorphisms. This section will focus on this scheme.
4. Data types as Initial F-Algebras - To use the catamorphism, the data types much be represented as a initial object. This section will attempt to bridge the gap.
5. Program Termination - 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 (Jacobs 2006).

Category

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 as a model of computation i.e. in Haskell, its *category* is called `Hask`, where the types form the objects and functions between two types: the morphisms.

Formally, a category $\mathbb{C}$ is an algebraic structure defined on a collection of:

- objects: denoted $A, B, C, ...$
- morphism: between a pair A, B is denoted $\mathbb{C}(A, B)$.

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. This property is 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.

Functors

It is natural to consider a structure preserving map as a 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 $F : \mathbb{C} \to \mathbb{D}$ which consists of:

- mapping $A \to F(A) : \mathbb{C} \to \mathbb{D}$
- mapping $f \to F(f) : \mathbb{C}(A, B) \to \mathbb{D}(FA, FB)$

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 satisfy are captured in Haskell by the functor laws, that each and every instance must satisfy:

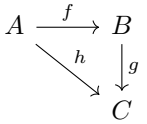
```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

Functors are hidden and prevalent in functional programming. They capture the theme of compatibility between categories of data types and allow for function reusability by "promoting" them 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.

Diagrams

Diagrams can be used in Category Theory for representing type information. A diagrams is said to *commute* if the map produced by following any path is the same.

For example:
Given $f : A \to B$, $g : B \to C$ and $h : A \to C$, the functions can be represented more intuitive with the following commutative diagram:



Although it is possible to formally define rules about the diagrams and their reasoning: it will be avoided in this report. Here, diagrams will be used for simple purposes - to give the reader a clear visual representation for the necessary type information, and in turn, hopefully, make it more intuitive.

F-Algebras

There is one category of interest for modelling our datatypes - the category of F-algebras.
Given a category $\mathbb{C}$ and an endofunctor $F : \mathbb{C} \to \mathbb{C}$, then an F-algebra is a tuple (A, f) where,

- A is an object in $\mathbb{C}$ and is called the *carrier* of the algebra.
- f is a morphism: $f : FA \to A$.

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

- $f \circ \alpha = \beta \circ Ff$.

$$\begin{array}{ccc} FA & \xrightarrow{\alpha} & A \\ \downarrow{\scriptstyle Ff} & & \downarrow{\scriptstyle f} \\ FB & \xrightarrow{\beta} & B \end{array}$$

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 called an *initial* when there exists a unique morphism from itself to all other F-algebras i.e. there is only one.

The category of F-algebras, denoted $\mathbb{ALG}(F)$, constitutes F-algebras 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 data structure and the algebras correspond to the function that can be performed on the data structure.

Products and Coproducts

Many new data types can be formed from composition or by tupling existing datatypes together. This concept exist in a category-theoretic perspective called: products and coproducts.

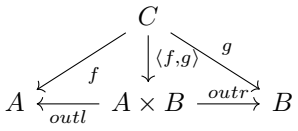
The product of two objects A and B in a category theory is formalised to capture the same concept as other constructions in mathematics such as the cartesian product in set theory. It (a product of two objects) is defined as:

- an object: denoted $A \times B$.
- two morphisms: denoted $outl : A \times B \to A$ and $outr : A \times B \to B$

satisfying for each arrow $f : C \to A$ and $g : C \to B$, there exists an arrow, denoted $\langle f, g \rangle : C \to A \times B$, such that the following holds:

$$h = \langle f, g \rangle \equiv outl \circ h = f \wedge outr \circ h = g$$

This is summarised by the following commutative diagram:



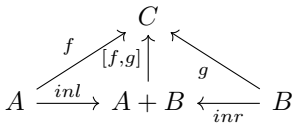
A coproduct is the categorical dual of a product, which means that the definition is the same except the morphisms are reversed. Despite its dual nature, they typically are drastically different from each other. It consists of:

- an object: denoted $A + B$
- two morphisms: denoted $inl : A \to A + B$ and $inr : B \to A + B$

satisfying for each arrow $f : A \to C$ and $g : B \to C$, there exists an arrow, denoted $[f, g] : A + B \to C$, such that the following holds:

$$h = [f, g] \equiv h \circ inl = f \wedge h \circ inr = g$$

This is summarised by the following commutative diagram:



Remarks

It is interesting to see that the category of `Hask` does not form a category, this is because of bottom value - two functions can be defined such that they represent the same morphism but with different values, which violates the definition of a category (Bauer 2016).

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.

Bibliographical Remarks

In the section, the use of category theory in the computer science community is well documented. There are numerous textbooks and resources available for those who wish to go further. The definitions here are summarised from (Bird and Moor 1997) and (Backhouse, Crole, and Gibbons 2002), this is because its definitions are easier to follow by foregoing some of its mathematical rigor.

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 of functional computation. Explicit recursion is a way of describing self referencing functions that is overused by 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. A set of recursion schemes (Meijer, Fokkinga, and Paterson 1991) introduced a set of combinators that capture 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 (Hutton 1998) where the meaning of the program is defined in terms of transition functions during program execution.

It has been well established that the use of `gotos` in imperative programming obscures the structure of the program and reduces the programmer's ability to reason with their code (Dijkstra 1968). 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 their code like never before. In addition, there is a catalogue of useful theorems and laws which can be 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 that programmers trained in the art of structuring recursion can concentrate on what the computation is doing, rather than how it is done.

4 Recursion Schemes

Recursion schemes have risen from attempts to tame the unyielding power of recursion. As a result, there is now a large zoo of formalised recursive operators that capture different types of recursion. There have been attempts to unify these schemes (Hinze, Wu, and Gibbons 2013).

Denotational semantics can be structured as a fold (Hutton 1998), which in the zoo of recursion schemes is called the catamorphism. It motivates a close examination of this scheme and will be the main focus of this section. References may 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.

4.1 Catamorphism

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

4.2 Derivation

Given an initial F-algebra (T, α), there is a unique homomorphism to all F-algebras in $\mathbb{ALG}(F)$ by definition of initiality. The catamorphism, denoted $(\!|f|\!)$, corresponds to the observation of this homomorphism from the initial algebra to some algebra.

$$\begin{array}{ccccc} & F\ T & \underset{\alpha}{\rightarrow} & T & \\ F(\!|f|\!) & \downarrow & & \downarrow & (\!|f|\!) \\ & F\ A & \underset{f}{\rightarrow} & A & \end{array}$$

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

```
data NatF k = Zero
            | Succ k
            deriving Functor

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

In the `number` function, `Fix NatF` corresponds to the initial algebra (see Section 5), to be transformed by the some 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.

4.3 Theorems

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

Fusion

Fusion law for catamorphism (Meijer, Fokkinga, and Paterson 1991) allows a composition of functions with a catamorphism to be transformed into a single catamorphism. It is one of the most important laws for program derivation which states that:

$$h \circ f = g \circ F\ h \Rightarrow h \circ (\!|f|\!) = (\!|g|\!)$$

This can be proved by producing a diagram (Bird and Moor 1997):

$$\begin{array}{ccc} & F\ T & \underset{\alpha}{\rightarrow}\ T & \\ F\ (\!|f|\!) & \downarrow & \downarrow & (\!|f|\!) \\ & F\ A & \underset{f}{\rightarrow}\ A & \\ F\ h & \downarrow & \downarrow & h \\ & F\ B & \underset{g}{\rightarrow}\ B & \end{array}$$

From the assumption, this diagram commutes. Since the algebra (T, α) is initial, by definition, there is a homomorphism to the algebra $(F\ B, g)$ which is unique so we have:

$$h \circ (\!|f|\!) = (\!|g|\!)$$

This law can be used in Haskell, with the synonymous proposition:

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

where,

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

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 (Meijer, Fokkinga, and Paterson 1991) which states that:

$$\langle (\!|h|\!), (\!|k|\!) \rangle = (\!|\langle h \circ F\ outl, k \circ F\ outr \rangle|\!)$$

To prove this it is enough to prove that the diagram below commutes (Bird and Moor 1997) i.e. $\langle (\!|h|\!), (\!|k|\!) \rangle \circ \alpha = \langle h \circ F\ outl, k \circ F\ outr \rangle \circ F\ \langle (\!|h|\!), (\!|k|\!) \rangle$

$$\begin{array}{ccc} & F\ T & \underset{\alpha}{\rightarrow} \quad T \\ F\ \langle (\!|h|\!), (\!|k|\!) \rangle & \downarrow & \downarrow \quad \langle (\!|h|\!), (\!|k|\!) \rangle \\ & F\ A & \underset{\langle h \circ F\ outl, k \circ F\ outr \rangle}{\rightarrow} \quad A \end{array}$$

Proof:

$$\begin{array}{l} \langle (\!|h|\!), (\!|k|\!) \rangle \circ \alpha \\ = split\ fusion \\ \langle (\!|h|\!) \circ \alpha, (\!|k|\!) \circ \alpha \rangle \\ = catamorphism \\ \langle h \circ F (\!|h|\!), k \circ F (\!|k|\!) \rangle \\ = split\ cancellation \\ \langle h \circ F(outl \circ \langle (\!|h|\!), (\!|h|\!) \rangle), k \circ F(outr \circ \langle (\!|h|\!), (\!|h|\!) \rangle) \rangle \\ = Composition\ law\ of\ functor\ F. \\ \langle h \circ F\ outl \circ F\ \langle (\!|h|\!), (\!|h|\!) \rangle), k \circ F\ outr \circ F\ \langle (\!|h|\!), (\!|h|\!) \rangle \rangle \\ = split\ fusion \\ \langle h \circ F\ outl, k \circ F\ outr \rangle \circ F\ \langle (\!|h|\!), (\!|k|\!) \rangle \end{array}$$

5 Datatypes as Initial F-Algebras

To yield the power of the catamorphism, it requires the data structure to be the initial algebra. This section will discuss how to do this.

5.1 Polynomial functors

Functors can be built from constants, products and coproducts. The result is called the polynomial functor which can be used as a model to describe the structure of the datatype (Bird and Moor 1997). It is defined inductively as follows:

- The identity id and the constant functions f are polynomial functors.

- If F and G are polynomial then their composition, sum and product is also polynomial.

For example, consider a very simple language of addition.

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

The equivalent polynomial functor F called the base functor is defined by:

- $F\ A = 1 + A \times A$
- $F\ h = id + h \times h$.

Observe that the mapping of the base functor F corresponds to the definition of the *pattern functor* defined as:

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

In addition, the functor instance definition, in Haskell, of `ExprF` also corresponds to the behaviour of the base functor:

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

Notice that the recursive spot in `Expr` has been parameterised with `x`. In this new definition of `Expr`, `ExprF`, its type has been defined in terms of its subexpression - it is almost identical to the original `Expr`. However, `ExprF` is not quite equivalent. It needs to somehow arbitrarily nest `ExprF` in the definition.

5.2 Initiality

Consider the following `Expr` forming operations:

$$Val : 1 \to ExprF\ A$$
$$Add : ExprF\ A \times ExprF\ A \to ExprF\ A$$

This can be combined into a single operation using the concept of a coproduct.

$$[Val, Add] : 1 + (ExprF\ A \times ExprF\ A) \to ExprF\ A$$

This forms an algebra and is, in fact, *initial*.

The initial F-algebra for a given endofunctor, is interestingly unique up to isomorphism. This means that, even though there could be many forms of the initial object, it does not matter which is used. From the above, it is seen that coproducts can produce initial F-algebras - this is not the only way. By taking the fix point of the pattern functor, it is initial (Wadler 1990) and exists (Malcolm 1990).

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, it 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`, the corresponding pattern functor can be used to define the *fixed point of functors*, which is isomorphic to the original definition,

$$Fix\ ExprF \cong Expr$$

6 Program Termination

As seen, by using the simplest example of the recursion schemes, there is an catalogue of extremely useful laws that are obtained for free by structuring recursion. Another byproduct of using certain recursion schemes is that it gives us the ability to reason with the termination of the program.

Catamorphism allows the programmer to guarantee its termination. The function calls are made only on smaller elements of the inductively defined structure, implying that it will tend towards its base case, giving program termination. This is also true for the paramorphism. However, with explicit recursion, there is nothing to stop the programmer from recursively calling the function on a larger data type causing it to never terminate.

Conclusion

The purpose of this report was to show an in-depth analysis of the catamorphism as an example from the set of recursion schemes. To achieve this, simple concepts in category theory were introduced, motivated by the fact that these ideas can be used to streamline the derivation process. Many of its core ideas have found their way into functional programs and thus, into this report. The reader must recognise how successful it is as a model of computation.

Though only the catamorphism was shown, it is quite restrictive and forms only a small part of the large class of recursion schemes. Since its introduction, 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 into a single scheme.

However, no matter how 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 them as part of a language is not needed. It will introduce unnecessary complexities in the language which is exactly what DSLs are trying to avoid. In my opinion, the fight to tame recursion will never be won; if it gains turing completeness, the ability to reason will be lost.

In the future:

- a similar approach to this report can be taken but with generalised unfolds to derive and structure operational semantics.
- a report could document the analysis of different representations of initial F-algebras as datatypes in program construction.
- an investigation could formulise how extensive and successful category theory is as a model for computation.

References

Backhouse, Roland, Roy Crole, and Jeremy Gibbons, eds. 2002. "Algebraic and Coalgebraic Methods in the Mathematics of Program Construction." In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Vol. 2297. Lecture Notes in Computer Science. Springer-Verlag.

Bauer, Andrej. 2016. "Hask Is Not a Category," august.

Bird, Richard, and Oege de Moor. 1997. *Algebra of Programming*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Dijkstra, Edsger W. 1968. "Letters to the Editor: Go to Statement Considered Harmful." *Commun. ACM* 11 (3). New York, NY, USA: ACM: 147–48.

Gibbons, Jeremy, and Nicolas Wu. 2014. "Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl)." *SIGPLAN Not.* 49 (9). New York, NY, USA: ACM: 339–47.

Hinze, Ralf, Nicolas Wu, and Jeremy Gibbons. 2013. "Unifying Structured Recursion Schemes." In *Proceedings of the 18th Acm Sigplan International Conference on Functional Programming*, 209–20. ICFP '13. New York, NY, USA: ACM.

Hudak, Paul. 1996. "Building Domain-Specific Embedded Languages." *ACM Comput. Surv.* 28 (4es). New York, NY, USA: ACM.

Hutton, Graham. 1998. "Fold and Unfold for Program Semantics." In *Proceedings of the Third Acm Sigplan International Conference on Functional Programming*, 280–88. ICFP '98. New York, NY, USA: ACM.

Jacobs, Bart. 2006. "A Bialgebraic Review of Deterministic Automata, Regular Expressions and Languages." In *Algebra, Meaning, and Computation: Essays Dedicated to Joseph a. Goguen on the Occasion of His 65th Birthday*, edited by Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, 375–404. Berlin, Heidelberg: Springer Berlin Heidelberg.

Malcolm, Grant. 1990. "Data Structures and Program Transformation." *Sci. Comput. Program.* 14 (2-3): 255–79.

Meijer, Erik, Maarten Fokkinga, and Ross Paterson. 1991. "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire." In, 124–44. Springer-Verlag.

Wadler, Paul. 1990. "Recursive Types for Free!" july.