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 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.

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 DSLs 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 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 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. 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 show its derivation, theorems.
4. Pattern and Fix Point of Functors - Given a recursive data structure, the explicit recursion can be abstract away at the type level.
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?].

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 $\mathcal{C}$ 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.

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 $F : C \to D$ which 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 satisfy 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.

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 $h : A \to C$, it can represent with the following commutative diagram:

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[dr, "h"] & B \arrow[d, "g"] \\ & C \end{tikzcd}$$

Although it is possible to formal define rules about the diagrams and its 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, making it more intuitive.

F-Algebras

There is one category of interest for modelling our datatypes - 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 an F-algebra (A, α) to another F-algebra (B, β) is a morphism $C(A, B)$ such that:

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

$$\begin{tikzcd}
FA \arrow[r, "\alpha"] \arrow[d, "Ff"] & A \arrow[d, "f"] \\
FB \arrow[r, "\beta"] & D
\end{tikzcd}$$

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

The category of F-algebras 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 initial 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 tupling existing datatypes together. These concept exists in a category-theoretic perspective called: products and coproducts.

Product of two objects A, B in a category theory is formalised to capture the same concept as another constructions in other areas of mathematics e.g. the cartesian product in set theory. It (a product of two objects) is defined as:

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

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

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

This is summarised by the following diagram:

$$\begin{tikzcd}
 & A \arrow[dl, "f"'] \arrow[d, "\langle f,g \rangle"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "outl"] \arrow[r, "outr"] & B
\end{tikzcd}$$

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

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

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

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

This is summarised by the following diagram:

$$\begin{tikzcd}
 & A \arrow[dl, "f"'] \arrow[d, "{[f,g]}"] \arrow[dr, "g"] & \\
A \arrow[r, "inl"] & A + B & B \arrow[l, "inr"]
\end{tikzcd}$$

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 [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 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

[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 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[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 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 [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 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 Alg(F) by definition of initiality. The catamorphism, denoted $(\!|f|\!)$, corresponds to the observation of this homomorphism from the initial algebra to some algebra.

$$\begin{tikzcd}
F\,T \arrow[r, "\alpha"'] \arrow[d, "F(\!|f|\!)"'] & T \arrow[d, "(\!|f|\!)"] \\
F\,A \arrow[r, "f"'] & A
\end{tikzcd}$$

The initial algebra is the data type $(FixF, in)$, for some endofunctor F which represents our pattern functor.

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[3] can be used for free!

Fusion

Fusion law for catamorphism [@Meijer91functionalprogramming] allows a composition of functions with a catamorphism to transformed to a single catamorphism. It is one of the for program derivation. It states that:

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

This can be proved by producing a diagram.

$$\begin{tikzcd}
F\ FixF \arrow[r, "In"'] \arrow[d, "F\ (\!|f|\!)"'] & FixF \arrow[d, "(\!|f|\!)"] \\
F\ A \arrow[r, "f"'] \arrow[d, "F\ h"'] & A \arrow[d, "h"] \\
F\ B \arrow[r, "g"'] & B
\end{tikzcd}$$

From the assumption, this diagram commutes. Since the algebra (Fix, In) 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 [3] which states that:

$$\langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle = \llparenthesis \langle h \circ F\ outl, k \circ F\ outr \rangle \rrparenthesis$$

To prove this it is enough to prove that the diagram below commutes i.e.
$\langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle \circ In = \langle h \circ F\ outl, k \circ F\ outr \rangle \circ F\ \langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle$

$$\begin{tikzcd}
F\ FixF \arrow[r, "In"'] \arrow[d, "F\ \langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle"'] & FixF \arrow[d, "\langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle"] \\
F\ A \arrow[r, "\langle h \circ F\ outl, k \circ F outr \rangle"'] & A
\end{tikzcd}$$

Proof (Algebra of programming):

$$\begin{aligned}
&\langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle \circ In \\
&= split\ fusion \\
&\langle \llparenthesis h \rrparenthesis \circ In, \llparenthesis k \rrparenthesis \circ In \rangle \\
&= catamorphism \\
&\langle h \circ F \llparenthesis h \rrparenthesis, k \circ F \llparenthesis k \rrparenthesis \rangle \\
&= split\ cancellation \\
&\langle h \circ F(outl \circ \langle \llparenthesis h \rrparenthesis, \llparenthesis h \rrparenthesis \rangle), k \circ F(outr \circ \langle \llparenthesis h \rrparenthesis, \llparenthesis h \rrparenthesis \rangle) \rangle \\
&= Composition\ law\ of\ functor\ F. \\
&\langle h \circ F\ outl \circ F\ \langle \llparenthesis h \rrparenthesis, \llparenthesis h \rrparenthesis \rangle), k \circ F\ outr \circ F\ \langle \llparenthesis h \rrparenthesis, \llparenthesis h \rrparenthesis \rangle \rangle \\
&= split\ fusion \\
&\langle h \circ F\ outl, k \circ F\ outr \rangle \circ F\ \langle \llparenthesis h \rrparenthesis, \llparenthesis k \rrparenthesis \rangle
\end{aligned}$$

5 Datatypes as Initial F-Algebras

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

5.1 Polynomial functors

Functors can be built from constants, products and coproducts. This concept is called the polynomial of functors, this can be used as a model for data types. The model functor describes the structure of the data. 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 {citation} is defined by $F\ A = 1 + A \times A$ and $F\ h = id + h \times h$.
Observe that the mapping of the functor F for objects corresponds to the definition of the *pattern functor*:

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

The functor instance of `ExprF` also corresponds to the mapping of the functor for a morphism:

```
   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 be 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 need to somehow arbitrarily nest `ExprF` in the definition.

Consider the following `Expr` forming operations:

$$Val : 1 \rightarrow ExprFA$$
$$Add : ExprFA \times ExprFA \rightarrow ExprFA$$

This can be combined into a single operation using the concept of a co-product.

$$[Val, Add] : 1 \times (ExprA \times ExprA) \rightarrow ExprA$$

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

5.2 Initiality

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. By taking the fix point of the pattern functor, it is initial [4] and exists (malcom).

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 defined the *fixed point of functors*, which is isomorphic to the original definition,

$$FixExprF \cong Expr$$

6 Program Termination

As seen, by using the simplest example of the recursion schemes, there is an catalogue of extremely useful laws that is 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 to recursively call 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. Additionally, simple concepts in category theory was introduced motivated the fact that these ideas can be used as a model of computation. This allowed for a categorical perspective of inductively defined data types and streamlined derivation of the catamorphism with proof of its theorems.

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 try to avoid.

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

[@Meijer91functionalprogramming] Hutton, G. Fold and unfold for program semantics. Proc. 3rd ACM SIGPLAN International Conference on Functional Programming. (1998).
[2] Gibbons, Jeremy. Wu, Nicholas. Folding domain-specific languages: Deep and shallow embedding (functional pearl). (2014).
[3] Meijer, Erik. Fokkinga, Maarten. Paterson, Ross. Functional programming with bananas, lenses, envelopes and barbed wire. (1991). FPLCA. LNCS, vol. 523. Springer. Pages 124-144.
[4] Philip Wadler. Theorems for free ! In Proc. 1989 ACM Conference on Lisp and Functional Programming, pages 347{359, 1989.
[5] Building Domain Specific Embedded Language.
[6] Van Deursen, Arie. Klint, Paul. Visser, Joost. Domain-Specific Languages: An Annotated Bibliography. (2000). ACM SIGPLAN Notices.