The Monad.Reader/Issue4/Why Attribute Grammars Matter

From HaskellWiki

< The Monad.Reader | Issue4

Contents

1 Why Attribute Grammars Matter

by Wouter Swierstra for The Monad.Reader Issue Four; 01-07-05

1.1 Introduction

Almost twenty years have passed since John Hughes influential paper Why Functional Programming Matters (http://www.math.chalmers.se/~rjmh/Papers/whyfp.html) . At the same time the first work on attribute grammars and their relation to functional programming appeared. Despite the growing popularity of functional programming, attribute grammars remain remarkably less renown.

The purpose of this article is twofold. On the one hand it illustrates how functional programming sometimes scales poorly and how attribute grammars can remedy these problems. On the other hand it aims to provide a gentle introduction to attribute grammars for seasoned functional programmers.

1.2 The problem

John Hughes argues that with the increasing complexity of modern software systems,

modularity has become of paramount importance to software development. Functional languages provide new kinds of *glue* that create new opportunities for more modular code. In particular, Hughes stresses the importance of higher-order functions and lazy evaluation. There are plenty of examples where this works nicely - yet situations arise where the glue that functional programming provides somehow isn't quite enough.

Perhaps a small example is in order. Suppose we want to write a function `diff :: [Float] -> [Float]` that given a list `xs`, calculates a new list where every element `x` is replaced with the difference between `x` and the average of `xs`. Similar problems pop up in any library for performing statistical calculations.

1.2.1 Higher-order functions

Let's tackle the problem with some of Haskell's most powerful glue - higher-order functions. Any beginning Haskell programmer should be able to concoct the solution presented in Listing One. The average is computed using functions from the Prelude. The obvious function using this average is then mapped over the original list. So far, so good.

```
--- Listing One ---

diff :: [Float] -> [Float]
diff xs = map (\x -> x - (avg xs)) xs

avg :: [Float] -> Float
avg xs = sum xs / genericLength xs
```

There are, however, a few things swept under the rug in this example. First of all, this simple problem requires three traversals of the original list. Computing additional values from the original list will require even more traversals.

Secondly, the solution is so concise because it depends on Prelude functions. If the values were stored in a slightly different data structure, the solution would require a lot of tedious work. We could, of course, define our own higher-order functions, such as `map` and `fold`, or even resort to generic programming. There are, however, more ways to skin this particular cat.

This problem illustrates the sheer elegance of functional programming. We do pay a price for the succinctness of the solution. Multiple traversals and boilerplate code can both be quite a head-ache. If we want to perform complex computations over custom data structures, we may want to consider an alternative approach.

Fortunately, as experienced functional programmers, we have another card up our sleeve.

1.2.2 Lazy evaluation

The second kind of glue that functional programming provides is lazy evaluation. In essence, lazy evaluation only evaluates expressions when they become absolutely necessary.

In particular, lazy evaluation enables the definition of *circular programs* that bear a

dangerous resemblance to undefined values. Circular programs tuple separate computations, relying on lazy evaluation to feed the results of one computation to another.

In our example, we could simply compute the length and sum of the list at the same time:

```
average :: [Float] -> Float
average xs = let
 nil = (0.0, 0.0)
 cons x (s,l) = (x + s, 1.0 + l)
 (sum,length) = foldr cons nil xs
 in sum / length
```

We can eliminate traversals by tupling computations! Can we compute the resulting list at the same time as computing the sum and length? Let's try:

```
diff :: [Float] -> [Float]
diff xs = let
 nil = (0.0, 0.0, [])
 cons x (s,l,rs) = (x+s, 1.0+l, (x - ....) : rs)
 (sum,length,res) = foldr cons nil xs
 in res
```

We run into trouble when we try to use the average to construct the resulting list. The problem is, that we haven't computed the average, but somehow want to use it during the traversal. To solve this, we don't actually compute the resulting list, but rather compute a function taking the average to the resulting list:

```
diff :: [Float] -> [Float]
diff xs =
 let
  nil = (0.0, 0.0, \avg -> [])
  cons x (s,l,rs) = (x+s, 1.0+l, \avg -> (x - avg) : rs avg)
  (sum,length,res) = foldr cons nil xs
 in
  res (sum / length)
```

We can generalize this idea a bit further. Suppose that we want to compute other values that use the average. We could just add an `avg` argument to every element of the tuple that needs the average. It is a bit nicer, however, to lift the `avg` argument outside the tuple. Our final listing now becomes:

```
--- Listing Two ---

diff :: [Float] -> [Float]
diff xs =
 let
  nil avg = (0.0, 0.0, [])
  cons x fs avg =
   let
    (s,l,ds) = fs avg
   in
    (s+x,l+1.0,x-avg : ds)
  (sum,length,ds) = foldr cons nil xs (sum / length)
 in
```

```
  ds
```

Now every element of the tuple can refer to the average, rather than just the final list.

This *credit card transformation* eliminates multiple traversals by tupling computations. We use the average without worrying if we have actually managed to compute it. When we actually write the fold, however, we have to put our average where our mouth is. Fortunately, the `sum` and `length` don't depend on the average, so we are free to use these values to tie the recursive knot.

The code in Listing Two only needs a single traversal and one higher-order function. It apparently solves the problems with the code in Listing One.

Hold on a minute. What ever happened to the elegance of our previous solution? Our second solution appears to have sacrificed clarity for the sake of efficiency. Who in their right minds would want to write the code in Listing Two? I wouldn't. Maybe, just maybe, we can do a bit better.

1.3 Attribute Grammars

Before even explaining what an attribute grammar is, think back to when you first learned about *folds*. Initially, a fold seems like a silly abstraction. Why should I bother writing simple functions as folds? After all, I already know how to write the straightforward solution. It's only after a great deal of experience with functional programming that you learn to recognize folds as actually being a worthwhile abstraction. Learning about attribute grammars is similar in more ways than one.

So what are attribute grammars? I'll have a bit more to say about that later. For now, let's see what the attribute grammar solution to our running example looks like.

1.3.1 The attribute grammar solution

I'll introduce attribute grammars using the syntax of the Utrecht University Attribute Grammar (http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarSystem) system or UUAG for short. The UUAG system takes a file containing an attribute grammar definition and generates a Haskell module containing *semantic functions*, determined by the attribute grammar. The attribute grammar determines a computation over some data structure; the semantic functions correspond to the actual Haskell functions that perform the computation.

Although the UUAG system's syntax closely resembles Haskell, it is important to realize that the UUAG system is a Haskell pre-processor and not a complete Haskell compiler.

So what does an attribute grammar file look like? Well, first of all we have to declare the data structure we're working with. In our example, we simply have a list of Floats.

```
--- Listing Three ---

DATA Root
  | Root list : List
```

```
DATA List
  | Nil
  | Cons head : Float tail : List
```

Datatypes are declared with the keyword `DATA`, followed by a list of constructors. Every node explicitly gives the name and type of all its children. In our example we have an empty list `Nil` and a list constructor `Cons` with two children, `head` and `tail`. For reasons that will become apparent later on, we add an additional datatype corresponding to the root of our list.

So now that we've declared our datatype, let's add some *attributes*. If we want to compute the average element, we'll need the length of the list. Listing Four contains introduces our first attribute corresponding to a list's length.

```
--- Listing Four ---

ATTR List [|| length : Float]
SEM List
  | Nil lhs.length = 0.0
  | Cons lhs.length = 1.0 + @tail.length
```

Let's go over the code line by line.

An attribute has to be declared before it can actually be defined. An attribute is declared using the `ATTR` statement. This example declares a single *synthesized* attribute called `length` of type `Float`. A synthesized attribute is typically a value you are trying to compute bottom up. Synthesized attributes are declared to the right of the second vertical bar. We'll see other kinds attributes shortly.

Now that we've declared our first attribute, we can actually define it. A `SEM` statement begins by declaring for which data type attributes are being defined. In our example we want to define an attribute on a `List`, hence we write `SEM List`. We can subsequently give attribute definitions for the constructors of our `List` data type.

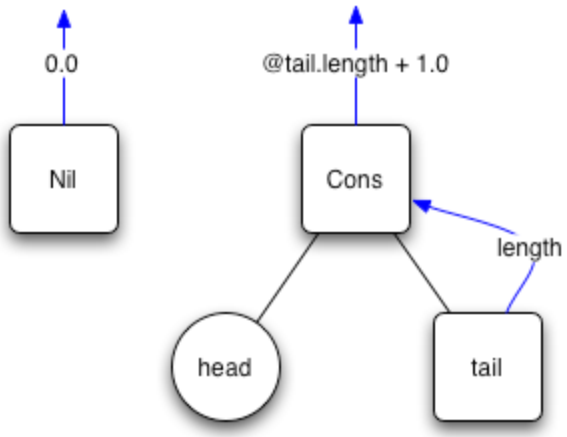
Every attribute definition consists of several parts. We begin by mentioning the constructor for which we define an attribute. In our example we give two definitions, one for `Nil` and one for `Cons`.

The second part of the attribute definition describes which attribute is being defined. In our example we define the attribute `length` for the *left-hand side*, or `lhs`. A lot of the terminology associated with attribute grammars comes from the world of context-free grammars. As this tutorial focuses on functional programmers, rather than formal language gurus, feel free to read `lhs` as "parent node". It seems a bit odd to write `lhs.length` explicitly, but we'll see later on why merely writing `length` doesn't suffice.

Basically, all we've only said that the two definitions define the `length` of `Nil` and `Cons`. We still have to fill in the necessary definition. The actual definition of the attributes takes place to the right of the equals sign. Programmers are free to write any valid Haskell expression. In fact, the UUAG system does not analyse the attribute definitions at all, but merely copies them straight into the resulting Haskell module. In our example, we want the length of the empty list to be `0.0`. The case for `Cons` is a bit trickier.

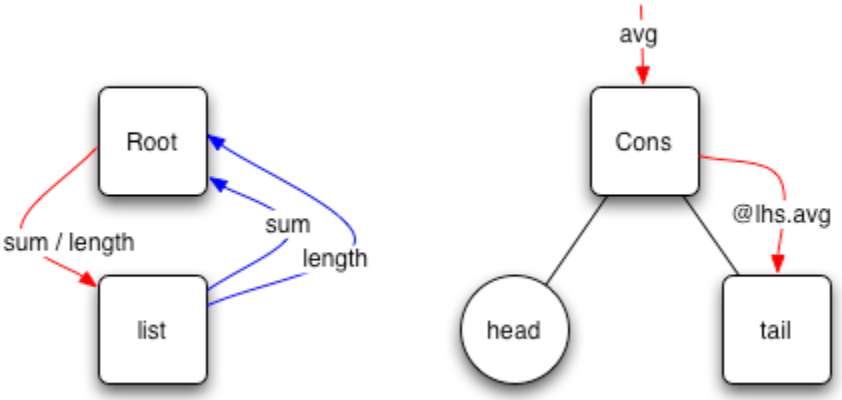
In the `Cons` case we want to be increment the length computed so far. To do so we need to be able to refer to other attributes. In particular we want to refer to the `length` attribute of the tail. The expression `@tail.length` does just that. In general, you're free to refer to any synthesized attribute *attr* of a child node *c* by writing `@c.attr`.

The `length` attribute can be depicted pictorally as follows:



Exercise: Declare and define a synthesized attribute `sum` that computes the sum of a `List`. You can refer to a value `val` stored at a node as `@val`. For instance, write `@head` to refer to the float stored in at a `Cons` node. Draw the corresponding picture if you're stuck.

Now we've defined `length` and `sum`, let's compute the average. We'll know the sum and the length of the entire list at the Root node. Using those attributes we can compute the average and *broadcast* the average through the rest of the list. Let's start with the picture this time:



The previous synthesized attributes, `length` and `sum`, defined bottom-up computations. We're now in the situation, however, where we want to pass information through the tree from a parent node to its child nodes using an *inherited* attribute. Listing Five defines an inherited attribute `avg` that corresponds to the picture we just drew.

```
--- Listing Five ---
ATTR List [ avg : Float|| ]
```

```
SEM Root
  | Root list.avg = @list.sum / @list.length

SEM List
  | Cons tail.avg = @lhs.avg
```

Inherited attributes are declared to the left of the two vertical bars. Once we've declared an inherited attribute `avg` on lists, we're obliged to define how every constructor passes an `avg` to its children of type `List`.

In our example, there are only two constructors with children of type `List`, namely `Root` and `Cons`. At the `Root` we compute the average, using the synthesized attributes `sum` and `length`, and pass the result to the `list` child. At the `Cons` node, we merely copy down the `avg` we received from our parent. Analogous to synthesized attributes, we can refer to an inherited attribute `attr` by writing `@lhs.attr`.

Admittedly, this inherited attribute is not terribly interesting. There are plenty of other examples, however, where an inherited attribute represents important contextual information. Think of passing around the set of assumptions when writing a type checker, for instance.

Exercise: To complete the attribute grammar, define an attribute `res` that computes the resulting list. Should it be inherited or synthesized? You may want to draw a picture.

1.3.2 Running the UUAG

Now suppose you've completed the exercises and copied the examples in a single file called `Diff.ag`. How do we actually use the attribute grammar? This is were the UUAG compiler steps in. Running the UUAG compiler on the source attribute grammar file generates a new `Diff.hs` file, which we can then compile like any other Haskell file.

```
> uuagc -a Diff.ag
> ghci Diff.hs
```

The `Diff.hs` file contains several ingredients.

Firstly, new Haskell datatypes are generated corresponding to `DATA` declarations in the attribute grammar. For every generated datatype a corresponding `fold` is generated. The attribute definitions determine the arguments passed to the folds. Browsing through the generated code can actually be quite instructive.

Inherited attributes are passed to recursive calls of the fold. Synthesized attributes are tupled and returned as the result of the computation. In essence, we've reproduced our original solution in Listing Two - but now without the hassle associated with spelling out catamorphisms with a higher order domain and a compound codomain.

The attribute grammar solution is just as efficient as our earlier solution relying on lazy evaluation, yet the code is hardly different from what we would write in a straightforward Haskell solution. It really is the best of both worlds. The two types of glue that John Hughes pinpoints in his original article just aren't enough. I would like to think that

sometimes attribute grammars are sometimes capable of providing just the right missing bit of glue.

1.3.3 What are attribute grammars?

So just what are attribute grammars? Well, that depends on who you ask, really. I've tried to sum up some different views below.

Attribute grammars add semantics to a context free grammar. Although it is easy enough to describe a language's syntax using a context free grammar, accurately describing a language's semantics is notoriously difficult. Attribute grammars specify a language's semantics by 'decorating' a context free grammar with those attributes you are interested in.

Attribute grammars describe tree traversals. All imperative implementations of attribute grammar systems perform tree traversals to compute some value. Basically an attribute grammar declares *which* values to compute and an attribute grammar system executes these computations. Once you've made this observation, the close relation to functional programming should not come as a surprise.

Attribute grammars are a formalism for writing catamorphisms in a compositional fashion. Basically, the only thing the UUAG compiler does is generate large folds that I couldn't be bothered writing myself. It takes away all the elbow grease involved with maintaining and extending such code. In a sense the compiler does absolutely nothing new; it just makes life a lot easier.

Attribute grammars provide framework for aspect oriented programming in functional languages. Lately there has been a lot of buzz about the importance of *aspects* and *aspect oriented programming*. Attribute grammars provide a clear and well-established framework for splitting code into separate aspects. By spreading attribute definitions over several different files and grouping them according to aspect, attribute grammars provide a natural setting for aspect oriented programming.

How do attribute grammars relate to other Haskell abstractions? I'll try to put my finger on some of the more obvious connections, but I'm pretty sure there's a great deal more that I don't cover here.

1.4 What else is out there?

Everyone loves monads. They're what makes IO possible in Haskell. There are extensive standard libraries and syntactic sugar specifically designed to make life with monads easier. There are an enormous number of Haskell libraries based on the monadic interface. They represent one of the most substantial developments of functional programming in the last decade.

Yet somehow, the single most common question asked by fledgling Haskell programmers is probably *What are monads?*. Beginners have a hard time grasping the concept of monads and yet connoisseurs recognize a monad in just about every code snippet. I think the more important question is: *What are monads good for?*

Monads provide a simple yet powerful abstract notion of computation. In essence, a monad describes how sequence computations. This is crucial in order to perform IO in a functional language; by constraining all IO actions to a single interface of sequenced computations, the programmer is prevented from creating utter chaos. The real power of monads is in the interface they provide.

John Hughes identified modularity as the single biggest blessing of functional programming. The obvious question is: how modular is the monadic interface? This really depends on you're definition of modularity. Let me be more specific. How can you combine two arbitrary monads? You can't. This is my greatest concern with monads. Once you choose your specific notion of computation, you have to stick to it through thick and thin.

What about monad transformers? Monad transformers allow you to add a specific monad's functionality on top of any existing monad. What seems like a solution, more often than not, turns out to introduce more problems than you bargained for. Adding new functionality to a monad involves lifting all the computations from the previous monad to the new one. Although I could learn to live with this, it gets even worse. As every monad transformer really changes the underlying monad. The order in which monad transformers are applied really makes a difference. If I want to add error reporting and state to some existing monad, should I be forced to consider the order in which I add them?

Monads are extremely worthwhile for the interface they provide. Monadic libraries are great, but changing and extending monadic code can be a pain. Can we do better? Well I probably wouldn't have started this monadic intermezzo if I didn't have some sort of answer.

Let's start off with `Reader` monads, for instance. Essentially, `Reader` monads adds an argument to some computation. Wait a minute, this reminds me of inherited attributes. What about `Writer` monads? They correspond to synthesized attributes of course. Finally, `State` monads correspond to *chained* attributes, or attributes that are both synthesized and inherited. The real edge attribute grammars hold over monad transformers is that you can define new attributes *without* worrying about the order in which you define them or adapting existing code.

Do other abstractions capture other notions related to attribute grammars? Of course they do! Just look at the function space arrows instance. The notion of combining two distinct computations using the `(&&&)` operator relates to the concept of *joining* two attribute grammars by collecting their attribute definitions. When you look at the `loop` combinator, I can only be grateful that an attribute grammar system deals with attribute dependencies automatically.

There really is a lot of related work. Implicit parameters? Inherited attributes! Linear implicit parameters? Chained attributes! Concepts that are so natural in the setting of attribute grammars, yet seem contrived when added to Haskell. This strengthens my belief that functional programmers can really benefit from even the most fleeting experience with attribute grammars; although I'd like to think that if you've read this far, you're hungry for more.

1.5 Further reading

This more or less covers the tutorial section of this article. The best way to learn more about attribute grammars is by actually using them. To conclude the tutorial, I've included a small example for you to play with. I've written a parser for a very simple wiki formatting language not entirely unlike the one used to produce this document. So far the HTML generated after parsing a document is fairly poor. It's up to you to improve it!

You can download the initial version here. Don't forget to install the [1] (http://www.cs.uu.nl/wiki/bin/view/HUT/Download) . It might be worthwhile to have a look at the UUAG manual (http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarManual) as there's a lot of technical detail that I haven't mentioned.

If you're particularly daring, you may want to take a look at the Essential Haskell Compiler (http://www.cs.uu.nl/wiki/Ehc/WebHome) being developed at Utrecht. It's almost completely written using the UUAG and is designed to be suitable for education and experimentation. The compiler was presented at the Summer School for Advanced Functional Programming in Tartu, Estonia last summer. As a result, there's a lot written about it already.

Dive on in!