Algebraic Language Utilities

Uk'taad B'mal
The University of Kansas - ITTC
2335 Irving Hill Rd, Lawrence, KS 66045
lambda@ittc.ku.edu

February 13, 2007

Abstract

This literate script provides a collection of algebraic structures useful for defining languages and types. These idea are drawn from several sources in the literature and simply represent a documented, standard implementation for use in language definitions. Documentation is intended to be sufficient for those learning to use the algebraic structures in language processor design.

1 Introduction

The *LangUtils* module provides classes, data structures and functions for developing languages and language transformations for compositional language definition. The *Sum* type constructor is used to compose languages defined over value spaces. The *Functor* and *Algebra* constructors are used to provide transformation and interpretation functions for language elements. The *cata* function defines a catamorphism that combines transformation and interpretation functions and defines evaluation function for languages. The *Subsum* provides mechanisms for asserting subtype relationship between language elements and the main language.

```
{-# OPTIONS -fglasgow-exts -fallow-undecidable-instances -fallow-overlapping-instances #-}
module LangUtils where

import Control.Monad
```

```
import Control.Monad.Reader
import Control.Monad.Error
import Data.Maybe
import Data.List
```

2 The Sum Type

First define the *Sum* data type that will allow us to combine language elements. f and g are language elements and x is the value space over which the language elements are defined:

```
data Sum f g x = S (Either (f x) (g x))
unS (S x) = x
```

Note that the *Sum* type differs from the standard *Either* type with the presence of a third parameter.

3 Functors and Algebras

To understand the *Functor* and *Algebra* classes with respect to languages it is important to remember that in our definitions a is a value space and $f\ a$ is a term language defined over that value space. For example, if a is *Integer* and f is *List*, then $f\ a$ is the language of lists defined over integers. When we define functors, we are defining a transform, map_f, from a language defined over one value space to the same language defined over another value space. When we define algebras, we are defining a transform, ϕ, from a language defined over a value space to the value space itself. In effect, ϕ provides an evaluation function.

It is important to remember the definition of *Functor* provide in the standard Prelude:

```
class Functor f where
    map_f :: (a → b) → f a → f b
```

A *Functor* is some type constructor, f that we can define a map, map_f, over. The signature of map_f states that given a function mapping type a to type b and an instance of the functor $f\ a$, map_f will generate an instance of the functor over b, $f\ b$. For example, if f is a *List* type constructor, then map_f is the built in *map* function. In language processing, $f\ a$ is an abstract syntax tree representing a language and map_f is a set of transformations over language elements. If $a \equiv b$ then we are defining language transformations within the same language.

We define *Algebra* and *Coalgebra* classes that define ϕ and ψ respectively. If f is a functor, then $f\ a$ is an *Algebra* if ϕ can be defined mapping an application of f to it's argument type a. Conversely, the *Coalgebra* defines a mapping ψ from the argument type to a *Functor* application.

```
class Functor f => Algebra f a where
    φ :: f a -> a

class Functor f => Coalgebra f a where
    ψ :: a -> f a
```

From an algebraic perspective, a is the carrier set over which the algebra f is defined. f defines the terms of the algebra. ϕ is the mapping from terms to the carrier set. Thus, ϕ defines the value from the carrier set associated with terms in f. That is why we use ϕ for evaluation when defining languages.

4 Fixed Point Types

The *Rec* type constructor defines a fixed point type for some type f having one type argument. The fixed point type creates a parameter-less type from the single parameter type that represents all possible individual terms in the language by instantiating the carrier set parameter with the language itself. The constructor *In* is necessary `Haskell` machinery for defining a data type. *out* is a utility function for unpackaging the fixed point from the type constructor.

```
data Rec f = In (f (Rec f))
out :: Rec f -> f (Rec f)
out (In x) = x
```

To understand fixed point types and what they achieve, consider the following data type definitions:

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

type Lang0 = Expr Int
```

The elements of *Lang0* are things that can be constructed using *Val* and *Add* over *Int*. Thus, (*Add* 1 2) and (*Val* 3) are correct elements of the language. However, (*Add* (*Val* 2) (*Val* 4)) and (*Add* (*Add* 1 2) 3) are not because the arguments to *Add* are not of the type used to isntantiate *e*,specifically *Int*.

The first fix to this problem would be to instantate *e* with *Expr Int* as follows:

```
type Lang1 = Expr (Expr Int)
```

Now the arguments to constructors can be of type *Expr Int*. Now the constructions (*Add* (*Val* 1) (*Val* 2)) and (*Add* (*Add* 1 2) (*Const* 3)) are legal language elements. Unfortunately, we have only defered the problem. The construction *Add* (*Add* (*Add* 1 2) (*Add* 2 3)) is still illegal.

It should be clear that defining:

```
type Lang2 = Expr (Expr (Expr Int))
```

simply defers the problem further.

The problem is that the language we want where *Add* can be nested arbitrarily is an infinite recursion. No matter how *Expr* is nested above, the nesting will be finite. In this language, there will always be an *Add* instance whose arguments must be of type *Int*.

The fixed point constructor solves this problem by lazily finding the fixed point of langauge. What if we specified the following (ignoring Haskell details for the moment):

```
type Lang3 = Expr Lang3
```

In this interesting recursive structure, the parameter to *Expr* to create *Lang3* is *Lang3* itself. Thus, an *Add* can be instantiated with any element of *lang3* or any *Expr*. The reason for the *Val* constructor should nobe apparent because something must stop the recursive construction. Because *Val* does not depend on the e parameter, *Expr* values cannot be nested in *Val*.

Unfortunately, the semantics of **type** in `Haskell` do not allow the construction above. A datatype must be used requiring the inclusion of a constructor. Thus, a constructor for the data type must be defined to keep `Haskell` happy resulting in the generalized definition:

$$\textbf{data}\ Rec\ F = In\ (F\ (Rec\ F))$$

5 Catamorphism

The *cata* function represents a catamorphism, or fold, over a fixed point recursive structure. The constraint *Algebra* $f\ a$ assures that f is an algebra and transitively that f is a functor. Thus, both ϕ and map_f exist with respect to $f\ a$. *cata* is the composition of three functions, ϕ, $map_f\ cata$ and *out*. *out* takes a recursive type and removes the *In* type constructor introduced by *Rec*. $map_f\ cata$ maps the *cata* function over f effectively pushing the *cata* operation into f. This effectively evaluates the subterms of any term currently being evaluated. Finally, ϕ evaluates the result of $map_f\ cata$ by transforming the result of the catamorphism to a value in the carrier set. From a language perspective, $map_f\ cata$ pushes the evaluation into the sub-expression of the argument expression while ϕ evaluates the result to an element of the value space.

$$\begin{aligned}&cata :: (Algebra\ f\ a) \Rightarrow Rec\ f \rightarrow a\\&cata = \phi \circ map_f\ cata \circ out\end{aligned}$$

An interesting property of *cata* is that it provides an evaluation capability for *any* language described by an *Algebra* of the form $f\ a$. Thus, as long as the *Algebra* property is established, we automatically get *cata* for our evaluation function. As we shall see later, it is useful to actually define *eval* with a signature that directly constrains the types associated with *cata*.

6 Combining Language Elements

Earlier the *Sum* type was defined to combine data types. If we wish to combine language elements represented as data types using the *Sum*, then the resulting structure must also be a functor and map_f must be defined over the sum. This is quite easily done using the *Either* type encapsulated in the *Sum*. If f and g are functors, then $Sum\ f\ g$ is a functor where map_f selects the map_f associated with f or g depending on whether it is operating on the left or right part of the sum.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Functor}\ (\mathit{Sum}\ f\ g)\ \textbf{where} \\
\quad map_f\ h\ (S\ (\mathit{Prelude.Left}\ x)) = S\ (\mathit{Prelude.Left}\ (map_f\ h\ x)) \\
\quad map_f\ h\ (S\ (\mathit{Prelude.Right}\ x)) = S\ (\mathit{Prelude.Right}\ (map_f\ h\ x))
\end{array}
$$

If $f\ a$ and $g\ a$ are algebras, then $Sum\ f\ g$ is also an algebra were ϕ is the sum of the ϕ functions from the original algebras. Here we use the built-in *either* function to select the appropriate ϕ. *unS* is composed with $either\ \phi\ \phi$ to remove the S constructor introduced by the *Sum*.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Algebra}\ f\ a, \mathit{Algebra}\ g\ a) \Rightarrow \mathit{Algebra}\ (\mathit{Sum}\ f\ g)\ a \\
\quad \textbf{where}\ \phi = \mathit{either}\ \phi\ \phi \circ \mathit{unS}
\end{array}
$$

With the definition of *Functor* and *Algebra* over the *Sum* constructor, any *Algebra*s composed using the *Sum* type are themselves *Algebra*s. The implication to languages is we can define individual language elements and compose them into larger languages using the *Sum* type. Furthermore, we know the resulting language is an instance of *Algebra* and *Functor* making *cata* available as an evaluation function.

7 Subtypes and Subsums

We define *Subtype* over two types in the classical way by defining $\uparrow$ (injection) and $\downarrow$ (projection) functions between the types. If a is a subtype of b, then it will always be possible to inject an element of a into the type b. This is not the case for the projection function, thus we use the *Maybe* type for the domain of $\downarrow$.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Subtype}\ a\ b\ \textbf{where} \\
\quad \uparrow :: a \rightarrow b \\
\quad \downarrow :: b \rightarrow \mathit{Maybe}\ a
\end{array}
$$

We can now define some standard subtype relationships using the *Subtype* constructor. First, every type x is a subtype of itself where $\uparrow = id$ and $\downarrow = Just$. When injecting a member of a type into itself, we simply want the element. Because we can always project a member of a type into itself, we simply use *Just* to encapsulate the type element.

$$\begin{array}{l} \textbf{instance } \mathit{Subtype}\ x\ x \textbf{ where} \\ \quad \uparrow = id \\ \quad \downarrow = \mathit{Just} \end{array}$$

To make things easier in a traditional interpreter structure, we define a projection function that uses the *Error* monad to throw an error message in the traditional manner. This function is technically not required, but does make using the definitions simpler in real interpreters. Skip this function if you find it difficult to follow as it is not used in the remainder of the paper.

$$\begin{array}{l} \mathit{mPrj} :: (\mathit{Error}\ e, (\mathit{MonadError}\ e\ m), \mathit{Show}\ b, \mathit{Subtype}\ a\ b) \Rightarrow b \rightarrow m\ a \\ \mathit{mPrj}\ x = \mathit{maybe}\ (\mathit{throwError}\ \$\ \mathit{strMsg}\ \mathit{errmsg})\ \mathit{return}\ (\downarrow x) \\ \quad \textbf{where} \\ \quad \mathit{errmsg} = \texttt{"Type error: Cannot project `"} +\!\!+ \\ \qquad\qquad (\mathit{show}\ x) +\!\!+ \\ \qquad\qquad \texttt{"' to the desired type"} \end{array}$$

$$\mathit{retInj}\ x = \mathit{return}\ (\uparrow x)$$

Subsum is a version of *Subtype* where f and g are parameterized over a common type. The reason for *Subsums*' existance is that we would like to define languages as compositions of algebras using language elements. All such language elements are parameterized over the value space or carrier set. This is difficult to deal with using the traditional *Subtype* definition above. The injection and projection functions are defined similarly using the *Maybe* type when defining projection.

$$\begin{array}{l} \textbf{class } \mathit{Subsum}\ f\ g \textbf{ where} \\ \quad \uparrow_S :: f\ x \rightarrow g\ x \\ \quad \downarrow_S :: g\ x \rightarrow \mathit{Maybe}\ (f\ x) \end{array}$$

As every f is a *Subtype* of itself, every f is a *Subsum* of itself. Again, the same logic is followed from the *Subtype* definition.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Subsum}\ f\ f\ \textbf{where} \\
\quad \uparrow_S f = f \\
\quad \downarrow_S = \mathit{Just}
\end{array}
$$

We will prescribe how *Sum* is used to construct types to enable the definition of injection and projection functions. Specifically, we will assume that *Sum* is used to compose types in the following manner:

$$
\begin{array}{l}
\mathit{newType} = (\mathit{Sum}\ T_0 \\
\qquad\qquad\quad (\mathit{Sum}\ T_1 \\
\qquad\qquad\qquad (\mathit{Sum}\ T_2 \\
\qquad\qquad\qquad\ \ ... \\
\qquad\qquad\qquad\quad (\mathit{Sum}\ \mathit{Tm}\ \mathit{Tn})...)))
\end{array}
$$

Given this, we know that:

$$
\begin{array}{l}
\mathit{Subsum}\ f\ (\mathit{Sum}\ f\ g) \\
\mathit{Subsum}\ f\ g \Rightarrow \mathit{Subsum}\ f\ (\mathit{Sum}\ x\ g)
\end{array}
$$

should always hold for any f and g. The structure of the *Sum* makes these definitions easier to write.

First, define f to be a *Subsum* of *Sum* f g by defining $\uparrow_S$ and $\downarrow_S$ for all f and g:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Subsum}\ f\ (\mathit{Sum}\ f\ g)\ \textbf{where} \\
\quad \uparrow_S = S \circ \mathit{Prelude.Left} \\
\quad \downarrow_S (S\ (\mathit{Prelude.Left}\ f)) = \mathit{Just}\ f \\
\quad \downarrow_S (S\ (\mathit{Prelude.Right}\ f)) = \mathit{Nothing}
\end{array}
$$

Because f is always the left element of the *Sum*, $\uparrow_S$ is defined as the composition of the *Sum* constructor, S, and the *Left* constructor from the built in *Either* type. If f is the left element of the *Sum*, then the projection function is simply *Just*. If it is the right element of the sum, then the projection is *Nothing* indicating that there is no project from the *Sum* back to f. This works specifically because the left element of the *Sum* is always a leaf implying there is no need to descend a subtree created by *Sum*.

Remember that f and g are type variables in the instance. Thus, *Subsum* f (*Sum* f g) holds for all types f and g. This is an important result because we will get this *Subsum* for free each time we define a *Sum*. The same can be said for other instances defined over type variables rather than specific instances.

Second, if $\mathit{Subsum}\ f\ g$ holds, then $\mathit{Subsum}\ f\ (\mathit{Sum}\ x\ g)$ also holds. Unlike the first case, here the subtype relationship occurs in the right element of the *Sum*. Because the right element can be a tree, we must descend the tree when creating the injection function. Here $\uparrow_S$ is virtually the same with the addition of a recursive call to $\uparrow_S$. This is an interesting function because it will recursively call $\uparrow_S$ until: (i) the injected element is in the type associated with the left element of a sum; or (ii) the injected element is the type associated with the right sum element of the bottom subtree.

The projection function is similarly structured. As long as the function is looking at the right subtree, $\downarrow_S$ is called recursively until the type is found. If the the left subtree is ever examined, then *Nothing* is generated. Note the lack of a case returning *Just*. This will occur only projecting a subtype from itself and is taken care of by an earlier definition.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Subsum}\ f\ g) \Rightarrow \mathit{Subsum}\ f\ (\mathit{Sum}\ x\ g)\ \textbf{where} \\
\quad \uparrow_S = S \circ \mathit{Prelude.Right} \circ \uparrow_S \\
\quad \downarrow_S (S\ (\mathit{Prelude.Left}\ x)) = \mathit{Nothing} \\
\quad \downarrow_S (S\ (\mathit{Prelude.Right}\ b)) = \downarrow_S b
\end{array}
$$

Again, because f and g are type variables, this instance is available for any two types that satisfy the instance constraints. Specifically, if $\mathit{Subsum}\ f\ g$ holds, we know that $\mathit{Subsum}\ f\ (\mathit{Sum}\ x\ g)$ is defined and can be used. This will prove valuable later when we start defining languages.

The *toSum* function is a utility function that will inject a syntax element into a *Sum*. Specifically, if we have a f defined over the fixed point of an expression language, g, we can automatically inject members of f into the fixed point of g if f is a *Subsum* of g. Here f is the individual language component and g is the *Sum* of a number of language components including f. Thus, f is a *Subsum* of g and an injection function exists. This injection function can then be used along with the *Sum* constructor to create language elements.

$$
\begin{array}{l}
\mathit{toSum} :: (\mathit{Subsum}\ f\ g, \mathit{Functor}\ g) \Rightarrow f\ (\mathit{Rec}\ g) \rightarrow \mathit{Rec}\ g \\
\mathit{toSum}\ x = \mathit{In}\ (\uparrow_S x)
\end{array}
$$

8 Example Language

To demonstrate the use of these language definition features, we will define an interpreter for an *Integer* language that implements simple mathematical operations. We start by defining types for each language element:

$$
\begin{aligned}
&\textbf{data}\ \mathit{ExprConst}\ e = \mathit{EConst}\ \mathit{Int}\\
&\qquad\qquad\qquad\qquad \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})\\[1ex]
&\textbf{instance}\ \mathit{Functor}\ \mathit{ExprConst}\ \textbf{where}\\
&\quad \mathit{map}_f\ f\ (\mathit{EConst}\ x) = \mathit{EConst}\ x\\[1ex]
&\textbf{instance}\ \mathit{Algebra}\ \mathit{ExprConst}\ \mathit{Int}\ \textbf{where}\\
&\quad \phi\ (\mathit{EConst}\ x) = x\\[1ex]
&\textbf{data}\ \mathit{ExprAdd}\ e = \mathit{EAdd}\ e\ e\\
&\qquad\qquad\qquad\qquad \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})\\[1ex]
&\textbf{instance}\ \mathit{Functor}\ \mathit{ExprAdd}\ \textbf{where}\\
&\quad \mathit{map}_f\ f\ (\mathit{EAdd}\ x\ y) = (\mathit{EAdd}\ (f\ x)\ (f\ y))\\[1ex]
&\textbf{instance}\ \mathit{Algebra}\ \mathit{ExprAdd}\ \mathit{Int}\ \textbf{where}\\
&\quad \phi\ (\mathit{EAdd}\ x\ y) = x + y\\[1ex]
&\textbf{data}\ \mathit{ExprMult}\ e = \mathit{EMult}\ e\ e\\
&\qquad\qquad\qquad\qquad \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})\\[1ex]
&\textbf{instance}\ \mathit{Functor}\ \mathit{ExprMult}\ \textbf{where}\\
&\quad \mathit{map}_f\ f\ (\mathit{EMult}\ x\ y) = (\mathit{EMult}\ (f\ x)\ (f\ y))\\[1ex]
&\textbf{instance}\ \mathit{Algebra}\ \mathit{ExprMult}\ \mathit{Int}\ \textbf{where}\\
&\quad \phi\ (\mathit{EMult}\ x\ y) = x * y
\end{aligned}
$$

We avoided defining expressions recursively by pulling the recursive instance out and making it a parameter, called e, in all the expression types. Thus, it is not possible to define an *ExprAdd* over another *ExprAdd* because we don't know what e is. We can use the *Rec* type constructor to define the fixed point for each language element. For example, if we evaluate the following definition:

$$ExprAddLang = Rec\ ExprAdd$$

ExprAddLang is the fixed point of ExprAdd. Every element of *ExprAddLang* is constructed of *ExprAdd* defined over other *ExprAdd* expressions. This is not particularly useful in itself because it does not include other language elements, only *ExprAdd*. Thus, we will not include these definitions.

Now we combine the individual types into a single type using *Sum*:

$$\begin{aligned}\textbf{type}\ ExprVal = (&Sum\ ExprConst\\ &(Sum\ ExprAdd\ ExprMult))\end{aligned}$$

The *ExprVal* type is still parameterized over the expression type e. Using *ExprVal* and *toSum* we can instantiate any of the term types as elements of *ExprVal*. However, we still cannot define terms over other terms because the parameter e remains unspecified. What we would like is for e the same set of terms that comprise *ExprVal*. This is exactly what the *Rec* constructor for fixed point types provides:

$$\textbf{type}\ ExprLang = Rec\ ExprVal$$

The type *ExprLang* defines the complete expression language as *ExprVal*, the constructor for expressions, instantiated with itself. Said differently, *ExprLang* is the collection of expressions defined over expressions.

Isn't *ExprLang* infinite? It certainly should be because an infinite number of terms can be defined in our expression language. If that is the case, shouldn't the definition of *ExprLang* be nonterminating? Lazy evaluation helps here. Remember that elements of the collection of terms are not calculated until they are needed.

We will add a convenience function, *toExprLang*, that will take any of the language elements and project it into the full expression language. The function signature here is important as it adds a constraint to the application. Specifically, if f is a *Subsum* of *ExprVal*, then f over *ExprLang* can be mapped to *ExprLang* using *toSum*. What the signature does is constrain the types being manipulated by *toSum* and establish the existence of an injection function between f and *ExprVal*.

We know that if f is one of the expression elements, it is a *Subsum* of *ExprVal* because *ExprVal* is created with a *Sum*. This works like the inference that the *Sum* is a *Functor* and *Algebra*. Now we have $\downarrow$ defined between f and *ExprVal* as defined by the earlier type class. With that, *toSum* can do it's job over elements of *ExprLang*.

$$
\begin{aligned}
&\mathit{toExprLang} :: (\mathit{Subsum}\ f\ \mathit{ExprVal}) \Rightarrow f\ \mathit{ExprLang} \rightarrow \mathit{ExprLang}\\
&\mathit{toExprLang} = \mathit{toSum}
\end{aligned}
$$

We like to define have *ExprVal* to be a *Functor* and an *Algebra* giving us map_f and ϕ over the entire expression language. As it turns out, we get this for free from existing definitions. In the definitions associated with *Sum* of language element, we provided two definitions repeated here:

$$
\begin{aligned}
&\mathbf{instance}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Functor}\ (\mathit{Sum}\ f\ g)\ \mathbf{where}\\
&\quad map_f\ h\ (S\ (\mathit{Prelude.Left}\ x)) = S\ (\mathit{Prelude.Left}\ (map_f\ h\ x))\\
&\quad map_f\ h\ (S\ (\mathit{Prelude.Right}\ x)) = S\ (\mathit{Prelude.Right}\ (map_f\ h\ x))
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{instance}\ (\mathit{Algebra}\ f\ a, \mathit{Algebra}\ g\ a) \Rightarrow \mathit{Algebra}\ (\mathit{Sum}\ f\ g)\ a\\
&\quad \mathbf{where}\ \phi = \mathit{either}\ \phi\ \phi \circ \mathit{unS}
\end{aligned}
$$

Both definitions work in the same way. In the first, if f and g are instances of *Functor*, then *Sum* f g is also an instance of *Functor* with map_f defined as shown. Because all of the data types combined by the *Sum* are instances of *Functor*, so is the *Sum*. This comes for free from the definition. The same is true for *Algebra*. As long as new elements added to *ExprLangVal* are instances of *Functor* and *Algebra*, *ExprLangVal* will continue to be an instance of *Functor* and *Algebra*. This is a rather amazing result that will save substantial effort as we add to our language.

The easiest way to define an eval function for the language is to use the *cata* function. We'll include a type signature to help make types of return values work out:

$$
\begin{aligned}
&\mathit{eval0} :: \mathit{ExprLang} \rightarrow \mathit{Int}\\
&\mathit{eval0} = \mathit{cata}
\end{aligned}
$$

The penalty for defining languages in this way is a significantly more cryptic and complex abstract syntax definition. Using the abstract syntax directly requires projecting language elements into the main language before evaluation. Following are some specific examples that show how the *toExprLang* utility function is used to experiment with different abstract syntax constructs:

$$
\begin{array}{l}
test_0 :: ExprLang = toExprLang\ (EConst\ 1) \\
test_1 :: ExprLang = toExprLang\ (EAdd \\
\qquad\qquad\qquad\qquad\qquad (toExprLang\ (EConst\ 1)) \\
\qquad\qquad\qquad\qquad\qquad (toExprLang\ (EConst\ 2))) \\
test_2 :: ExprLang = toExprLang\ (EAdd \\
\qquad\qquad\qquad\qquad\qquad (toExprLang\ (EMult \\
\qquad\qquad\qquad\qquad\qquad\qquad (toExprLang\ (EConst\ 2)) \\
\qquad\qquad\qquad\qquad\qquad\qquad (toExprLang\ (EConst\ 4)))) \\
\qquad\qquad\qquad\qquad\qquad (toExprLang\ (EConst\ 1)))
\end{array}
$$

9 Conclusions

So why would anyone ever define an interpreter this way? Very simply because it is trivial to add new elements to the language. This involves a three step process of: (i) defining the new language element; (ii) making the element an instance of *Functor* and *Algebra*; and (iii) adding the new language element to the *Sum* defining the language. There is no need to touch the existing language elements or the *eval* function.