The InterpreterLib Explicit Algebra Package

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

August 18, 2005

Abstract

The use of *composable interpreters* has proven to be useful in the development of language parsers in our group. However, some aspects of techniques used in papers from the literature do not scale well to larger projects. More specifically, using polymorphism to select ϕ as is done in the *LangUtils* library will not work when multiple interpreters exist in the same environment. Furthermore, having instances of *Algebra* be opaque causes serious problems when we start looking at composing algebras.

1 Introduction

The *InterpreterLib* libraries are a collection of support packages for writing *composable interpreters* using *explicit algebras*. The term composable describes interpreters that are composed of modules defining interpreters for language components. Instead of writing a monolithic interpreter, we write individual components and assemble those components as needed for a specific language. The literature describes several approaches for writing composable interpreters [4, 2, 1, 3, 5]. All of them share the construction and integration of interpreter components.[1]

The approach we take in our interpreters is writing *functors* and *semantic algebras*. A functor is simply a specialized fold for a language construct. Recall that fold is a mechanism for recursively applying a function to a composite data structure and accumulating results. Functors for language elements "push" functions into language constructs. Each functor defines a function, *fmap*, that performs this function. For example, *fmap* over an **if** construct might have the following definition:

$$\mathit{fmap}\ g\ (\mathit{IfExpr}\ c\ t\ f) = (\mathit{IfExpr}\ (g\ c)\ (g\ t)\ (g\ f))$$

Thus, if we wanted to apply an interpretation function or some transformation function to a specific *IfExpr* we simply call *fmap fun* on the expression. If we define a functor for each language construct, we can fold a function into any term we might write.

A semantic algebra does exactly what its name implies by defining a semantics for each language construct. Algebras for language elements define how they are evaluated. Each algebra defines a function, traditionally called ϕ, that maps its associated language construct to a value. Using *fmap* to fold ϕ onto a composite language structure implements an interpreter for the language.[2] For example, ϕ for an **if** construct might have the following definition:

[1] The Lambda Group and SLDG lab have reports documenting the Hutton and Duponcheel approaches as well as example interpreters. The *LangUtils* module is also worth looking at for other examples.

[2] In the *InterpreterLib* modules, we use explicit algebras where ϕ is replaced by *apply*, but the principle is similar.

$\phi\ (IFExpr\ c\ t\ f) = \mathbf{do}\ \{ c' \leftarrow c$
$;\ return\ \$\ \mathbf{if}\ c' \equiv ETrue\ \mathbf{then}\ t\ \mathbf{else}\ f$
$\}$

The general idea is that we can write new semantic algebras and reuse functors to quickly generate new interpreters. What *InterpreterLib* does that *LangUtils* does not is provides a way to explicitly specify the algebra used by an interpreter. Further, the algebra structure defined is a `Haskell` data structure that can be manipulated like any other structure. Thus, defining traditional functors between algebras as well as algebra combinators is now possible.

2 Functors

```
{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Functors where
infixr 5: $ :
```

Define the standard fixed point for types. The constructor *In* is necessary to keep Haskell happy during type checking.

```
newtype Fix f = In (f (Fix f))
inn = In
out (In x) = x
```

Define the standard *Sum* type by encapsulating *Either*. Again, *S* is around to keep Haskell happy during type checking. Note the definition of an infix constructor alias.

```
newtype Sum f g x = S (Either (f x) (g x))
unS (S x) = x
type x :$: y = Sum x y
```

Define a sum of functors to be a functor by defining fmap appropriately. If we have two functors, a new functor can be defined from the *Sum* by using *Left* and *Right* to guide the application of *fmap*. Not much to it really, just stripping away the constructor, applying the *fmap* to the carried data, and putting the constructor back together.

```
instance (Functor f, Functor g) => Functor (Sum f g)
  where fmap h (S (Left x)) = S (Left $ fmap h x)
        fmap h (S (Right x)) = S (Right $ fmap h x)
```

Define *SubFunctor* in a manner very similar to the way *SubSum*. *injF* injects the subfunctor into its super functor. *prjF* does the opposite if it can. Note the use of the *Maybe* type.

```
class SubFunctor f g where
    injF :: f x -> g x
    prjF :: g x -> Maybe (f x)
```

An *Functor* is a *SubFunctor* of itself.

```
instance SubFunctor f f where
    injF f = f
    prjF = Just
```

Any *Functor* is a *SubFunctor* of a *Sum* that includes it as the left side of the pair.

```
instance SubFunctor f (Sum f x) where
    injF = S ∘ Left
    prjF (S (Left f)) = Just f
    prjF (S (Right x)) = Nothing
```

If f is a *SubFunctor* of g, then it is also a *SubFunctor* of the *Sum* of any x and g.

```
instance (SubFunctor f g) ⇒ SubFunctor f (Sum x g) where
    injF = S ∘ Right ∘ injF
    prjF (S (Left x)) = Nothing
    prjF (S (Right b)) = prjF b
```

Remaining functions seem to be helper functions of various kinds. *toS* seems to inject a term defined over a fixed point into a fixed point. The various *mkTerm* functions are used to create terms of various arities.

```
toS :: (SubFunctor f g, Functor g) ⇒ f (Fix g) → Fix g
toS x = inn $ injF x

mkTerm0 = toS
mkTerm f = toS ∘ f
mkTerm2 f = curry $ toS ∘ (uncurry f)
mkTerm3 f = curry $ curry $ toS ∘ (uncurry (uncurry f))
```

ZipFunctor provides a function that performs the standard *Functor* operation, but over two inputs rather than one. Remember that Functor takes a function, $(a \to b)$, and a term defined using a parameterized type, $f\ a$, and changes the term to use the carrier set b rather than a. *ZipFunctor* does the same thing, but the input langauge parameterized over two types.

```
class ZipFunctor f where
  zipFunctor :: Monad m ⇒ (a → b → c) → f a → f b → m (f c)
```

3 Algebras

```
{-# OPTIONS -fglasgow-exts -fno-monomorphism-restriction #-}
module InterpreterLib.Algebras where

import InterpreterLib.Functors
import Monad (liftM, liftM2)
import List ((\\))
import InterpreterLib.SubType

infixr 5@+@
```

One of the problems that we have with algebras in Duponcheel's and Gayo's work is that they are overloaded functions, meaning only one algebra is possible for a given functor/value space pair without giving us overlapping instances. It would be nice to be able to pass the algebra as a first class value, as well as do things like algebra extension, where one algebra extends the other.

Essentially an algebra is a parameterized function with a type $F\ a \to a$ for a given functor F and a *carrier set* a. There are some Haskell reasons we want to make the actual type of algebras abstract, instead just **type** $\mathit{AType}\ f\ a = f\ a \to a$, because we may want to manipulate algebras when combining two of them, and functions are opaque. A more transparent form, such as the record format we use below, gives us more flexibility for doing these manipulations. Unfortunately, we have to have the functional dependency $f \to \mathit{alg}$ ("the term type f uniquely determines the algebra representation alg"). Note that this means the algebra representation - the packaging around the function $f\ a \to a$, and not the actual function itself is uniquely. The problem with this is that we can't just combine two different algebra representations or even have them visible in the same namespace. One research question is how to combine heterogeneous algebra representations. One possibility is to use the *Arrows* abstraction of John Hughes.

6/12 - updated functional dependencies so that the dependency is reflexive, allowing the pairAlg stuff to work.

$$\textbf{type}\ \mathit{AlgSig}\ f\ a = f\ a \to a$$

Define a different *Algebra* than that used in standard approaches. This *Algebra* is parameterized over the traditional carrier set and an algebra structure. The algebra structure is the data type that provides definitions for the *Algebra*. Functional dependencies state that f can be uniquely determined from alg. As is typical, for something to be and *Algebra* it must also be a *Functor*.

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Algebra}\ f\ \mathit{alg}\ a \mid \mathit{alg} \to f\ \textbf{where}\\ \quad \mathit{apply} :: \mathit{alg}\ a \to f\ a \to a\end{array}$$

Algebra builder seems to define a mechanism for transforming some type into an algebra. I'm guessing this is for defining standard algebra formers

$$\begin{array}{l}\textbf{class}\ \mathit{Algebra}\ f\ \mathit{alg}\ a \Rightarrow \mathit{AlgebraBuilder}\ f\ \mathit{fType}\ \mathit{alg}\ a \mid \mathit{fType} \to f, \mathit{fType} \to a, \mathit{fType} \to \mathit{alg}\ \textbf{where}\\ \quad \mathit{mkAlgebra} :: \mathit{fType} \to (\mathit{alg}\ a)\end{array}$$

Parameterize *cata* over *Algebra*. Nice. $(\mathit{apply}\ \mathit{alg})$ effectively produces the function ϕ from the *alg* structure. Remember, *alg* is an algebra structure that contains functions defined for an *Algebra*. When used in this manner with *apply*, the appropriate function is extracted from the algebra and applied.

$$\mathit{cata}\ \mathit{alg} = (\mathit{apply}\ \mathit{alg}) \circ (\mathit{fmap}\ (\mathit{cata}\ \mathit{alg})) \circ \mathit{out}$$

pairAlg is a mechanism for pairing algebras.

$$\begin{array}{l}\mathit{pairAlg}\ \mathit{a1}\ \mathit{a2} = \mathit{mkAlgebra}\ \mathit{pa}\\ \quad \textbf{where}\ \mathit{pa}\ \mathit{term} = (((\mathit{apply}\ \mathit{a1}) \circ (\mathit{fmap}\ \mathit{fst}))\ \mathit{term}, ((\mathit{apply}\ \mathit{a2}) \circ (\mathit{fmap}\ \mathit{snd}))\ \mathit{term})\end{array}$$

Define sums of algebras in the same manner as sums of terms. Note that the same functions are defined, but with different argument types. The first definition seems to be a spurrious example and should be ignored for the time being.

```
sumAlg a1 a2 = SumAlgebra{ left = sumAlg',
                           right = sumAlg' }
  where sumAlg' (S (Left x)) = apply a1 x
        sumAlg' (S (Right x)) = apply a2 x

(@+@) = sumAlg

data SumAlgebra f g a = SumAlgebra{ left :: Sum f g a -> a,
                                    right :: Sum f g a -> a }


instance (Functor f, Functor g) => Algebra (Sum f g) (SumAlgebra f g) a where
  apply alg t@(S (Left _)) = left alg t
  apply alg t@(S (Right _)) = right alg t


instance (Functor f, Functor g) =>
  AlgebraBuilder (Sum f g) (Sum f g a -> a) (SumAlgebra f g) a where
  mkAlgebra f = SumAlgebra f f
```

4 Modules

```
module InterpreterLib.Modules where

class Module mod opened | mod -> opened where
  open :: mod -> opened
```

5 Sample Interpreter

```
 {-# OPTIONS -fglasgow-exts -fno-monomorphism-restriction #-}
import InterpreterLib.Algebras
import InterpreterLib.Functors
import InterpreterLib.SubType
import Monad
import Control.Monad.Reader
```

6 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 a type for the interpreter's value space:

```
data Value
    = ValNum Int
    | ValLambda (ValueMonad -> ValueMonad)

instance Show Value where
```

```
show (ValNum x) = show x
show (ValLambda _) = "<Function Value>"
```

This definition is unchanged from other interpreters where we want the value space defined separately from the language itself. The *Value* type is made an instance of *Show* to allow printing of interpreter results.

Next, we define types for each of the language's AST elements. We start with the definition for integer constants:

```
data AlgConst t = AlgC ((ExprConst t) → t)

data ExprConst e = EConst Int
                   deriving (Show, Eq)

instance Functor ExprConst where
    fmap f (EConst x) = EConst x

instance Algebra ExprConst AlgConst a where
    apply (AlgC f) x@(EConst _) = (f x)

mkEConst = inn ∘ sleft ∘ EConst
```

For each AST element we define 5 elements: (i) the *Algebra* data type; (ii) the *Term* data type; (iii) the *Functor* instance; (iv) the *Algebra* instance; and (v) a helper function to create terms. The *Term* data structure and *Functor* instance remain unchanged from previous modular interpreters. The *Term* structure defines a non-recursive type for representing terms while the *Functor* defines a mechanism for folding operations onto the term structure. These definitions remain unchanged from previous modular interpreters.

The algebra function does not change from previous interpreters. It continues to provide a mapping from a term to a value. The distinction is the *Algebra* type requires three parameters - the carrier set and the term type as before, plus the algebra structure used for evaluation. Note that the algebra is the instance of *Algebra* while the algebra structure provides the definitions used by the algebra.

The *apply* function defined for all *Algebra* instances takes the place of ϕ. It extracts what was ϕ from the algebra structure and applies it. Thus, *apply alg t* applies the interpretation function from *alg* to the term t. In affect, $\phi = (apply\ alg)$. One important difference should be noted. Frequently, ϕ used parameter matching to pull apart an argument and process its parts. *apply* is virtually always called with the argument intact.

As an example, let's step through this definition. *AlgConst* is the algebra structure defining interpretation of constants. It's only parameter is a function that maps *ExperConst* instances over some carrier set, t to t. This is precisely the signature of ϕ. However, instead of using polymorphism to find ϕ, we'll get it directly from the algebra when we invoke the catamorphism.

ExprConst is the datatype associated with constants and is an instance of *Functor*. *ExprConst* is an instance of *Functor* and *fmap* is defined in the canonical fashion to simply return the constant it is passed.

ExprConst is also an instance of *Algebra*. Here, the definition is different because *apply* takes two arguments - an algebra structure and a term - rather than one as it did in earlier implementations. In this case, *apply* first extracts the interpretation function, f, from the algebra structure. It then makes sure the term argument is the correct type and associates it with x. With the evaluation function and the term available, *apply* simply calls the evaluation function on the term.

The *mkEConst* function is a helper function that constructs a complete *ExprConstant* term. Defining terms is a real pain with all of the *Sum* and *Fix* cruft floating around. I suspect that these helper functions will need to be rewritten whenever the language changes due to the structure of the *Sum*. There may be a way around this similar to the techniques used in earlier languages.

The addition and multiplication terms are defined similarly:

```
data AlgAdd t = AlgAdd{ add :: (ExprAdd t) → t,
                        sub :: (ExprAdd t) → t}

data ExprAdd e = EAdd e e
               | ESub e e
               deriving (Show, Eq)

instance Functor ExprAdd where
    fmap f (EAdd x y) = (EAdd (f x) (f y))
    fmap f (ESub x y) = (ESub (f x) (f y))

instance Algebra ExprAdd AlgAdd a where
    apply alg x@(EAdd _ _) = (add alg x)
    apply alg x@(ESub _ _) = (sub alg x)

mkEAdd x y = inn $ sright $ sleft $ EAdd x y
mkESub x y = inn $ sright $ sleft $ ESub x y

data AlgMult t = AlgMult{ mult :: (ExprMult t) → t,
                          divi :: (ExprMult t) → t}

data ExprMult e = EMult e e
                | EDiv e e
                deriving (Show, Eq)

instance Functor ExprMult where
    fmap f (EMult x y) = (EMult (f x) (f y))
    fmap f (EDiv x y) = (EDiv (f x) (f y))

instance Algebra ExprMult AlgMult a where
    apply alg x@(EMult _ _) = (mult alg x)
    apply alg x@(EDiv _ _) = (divi alg x)

mkEMult x y = inn $ sright $ sright $ sleft $ EMult x y
mkEDiv x y = inn $ sright $ sright $ sleft $ EDiv x y
```

At this point we have elements of a simple language for arithmetic with no variables or functions. We can add lambdas, applications and variables using techniques similar to those from earlier interpreters:

```
data ExprFun t
    = ELambda String TyValue t
    | EApp t t
    | EVar String
     deriving (Show, Eq)
```

```haskell
data AlgFun t = AlgFun{ lam :: ExprFun t → t,
                        app :: ExprFun t → t,
                        var :: ExprFun t → t}

instance Functor ExprFun where
    fmap f (ELambda s ty t) = ELambda s ty (f t)
    fmap f (EApp t1 t2) = EApp (f t1) (f t2)
    fmap f (EVar s) = (EVar s)

instance Algebra ExprFun AlgFun a where
    apply alg x@(EApp _ _) = (app alg x)
    apply alg x@(ELambda _ _ _) = (lam alg x)
    apply alg x@(EVar _) = (var alg x)

mkEVar x = inn $ sright $ sright $ sright $ EVar x
mkELambda x ty y = inn $ sright $ sright $ sright $ ELambda x ty y
mkEApp x y = inn $ sright $ sright $ sright $ EApp x y
```

Note that the same five elements are defined for the collection of lambda terms as for previous language elements.

The lambda implemented here uses a *Reader* monad to maintain variables and their values in the execution environment as lambdas are applied to values. As each application is processes, the variable being replaced is paired with the value specified by the application. This is stored in the environment and used to determine the value of a variable when it is referenced.

The full language is now defined as the fixed point of the sum of language components. Here we have defined :$: as an infix form of *Sum*. However, the semantics is unchanged. *TermType* is the sum of term definitions and *TermLang* is the fixed point of the term definition.[3]

```haskell
type TermType = (ExprConst :$: (ExprAdd :$: (ExprMult :$: ExprFun)))

type TermLang = Fix TermType
```

We've now set up types for defining interpreters over this simple language, but we've not defined a specific semantics for the language. This is done by defining a specific algebra structure that provides *apply* for each term AST and summing the result to form an algebra for the complete language.

We start by defining types and functions for manipulating the environment. *ValueMonad* is the monad used to maintain the environment as values are calculated for terms. *Env* is the environment an is defined as a single element record containing a list *String*, *Value* pairs associating values with variables. *lookupVal* and *addVal* are helper functions for looking up and adding variable values to the environment.

```haskell
type ValueMonad = Reader Env Value

data Env = Env{ variables :: [(String, Value)]}

lookupVal name env = lookup name (variables env)

addVal b env = Env{ variables = b : (variables env)}
```

[3] *Fix* and :$: are both defined in module *Functor*.

Now we define helper functions that specify how each term type is evaluated. One function is defined for each AST construct. These definitions could easily be directly embedded in algebra structures and not defined separately. However, the algebra structure definition is greatly simplified by using this approach. They will each be inserted into an algebra structure prior to their use.

$$phiConst\ (EConst\ x) = return\ (ValNum\ x)$$

$$vPlus\ (ValNum\ x)\ (ValNum\ y) = ValNum\ (x + y)$$
$$vSub\ (ValNum\ x)\ (ValNum\ y) = ValNum\ (x - y)$$

$$phiAdd\ (EAdd\ x1\ x2) = liftM2\ vPlus\ x1\ x2$$
$$phiSub\ (ESub\ x1\ x2) = liftM2\ vSub\ x1\ x2$$

$$vMult\ (ValNum\ x)\ (ValNum\ y) = ValNum\ (x * y)$$
$$vDiv\ (ValNum\ x)\ (ValNum\ y) = ValNum\ ((div)\ x\ y)$$

$$phiMult\ (EMult\ x1\ x2) = liftM2\ vMult\ x1\ x2$$
$$phiDiv\ (EDiv\ x1\ x2) = liftM2\ vDiv\ x1\ x2$$

```
phiLambda (ELambda s _ t) =
    do { env ← ask
       ; return $ ValLambda (λv → (do { v′ ← v
                                       ; (local (const (addVal (s, v′) env)) t)
                                       }))}
phiApp (EApp x1 x2) =
    do { x1′ ← x1
       ; case x1′ of
         (ValLambda f) → (f x2)
         _ → error "Cannot apply non-lambda value"
       }
phiVar (EVar s) = do { v ← asks (lookupVal s)
                     ; case v of
                       (Just x) → return x
                       Nothing → error "Variable not found"
                     }
```

Note the use of *liftM2* to evaluate *x1* and *x2* prior to applying the actual evaluation function. In effect, *x1* and *x2* are evaluated in a **do** construct, then the specified function applied and the result packaged back into the monad using *return*. The definition of *liftM2* is in the *Control.Monad* package, but is repeated at the end of this file for documentation purposes.

The full term algebra is formed by creating algebra structures for each term from the definitions above and summing those definitions together. *AlgC*, *AlgAdd*, *AlgMult* and *AlgFun* take a function and build an algebra structure around it. This is what the data type definitions earlier are for. @ + @ is an infix Sum operation for algebra structures. This works the same way as the term sum to combine algebra structures into a single structure.[4]

$$termAlg = (AlgC\ phiConst)$$

[4] I believe the order structure of this sum and the term sum must be the same. Specifically, *AlgCost* and *ExprConst* are both first; *AlgAdd* and *ExprAdd* are both second; and so forth. I have not tested this assumption, but it would make very good sense to do it this way.

```
    @+@ (AlgAdd phiAdd phiSub)
    @+@ (AlgMult phiMult phiDiv)
    @+@ (AlgFun phiLambda phiApp phiVar)
```

The *evalFun* function composes *runReader* and *cata* to define evaluation. The heart of this function is the polytypic fold, or catamorphism. (*cata termAlg*) instantiates the *cata* function with the evaluation algebra. When applied to a term, it will produce a *ValueMonad* that is then evaluated by *runReader*.

```
evalFun = runReader ∘ (cata termAlg)
```

An initial value for the environment must be provided to *evalFun* for the reader to evaluate completely. To evaluate $term_1$ starting with an empty environment, execute the following:

```
(evalFun term1) Env{variables = []}
```

Now let's have some fun and define a different evaluation function for this language. If all is well, we should be able to define a new algebra structure, use the same sum and evaluate the language over a different carrier set. For this experiment, we'll use the simple odd/even carrier set.

First define the odd/even data type and some helper functions. Probably could use instances and continue to use + and *, but that's more than we really want to do here.

```
data OE = Odd | Even deriving (Show, Eq)

oePlus :: OE → OE → OE
oePlus Odd Odd = Even
oePlus Odd Even = Odd
oePlus Even Odd = Odd
oePlus Even Even = Even

oeTimes :: OE → OE → OE
oeTimes Odd Odd = Odd
oeTimes Odd Even = Even
oeTimes Even Odd = Even
oeTimes Even Even = Even
```

Second, define the evaluation functions that we'll use with *apply*. For these definitions, we'll define α, the abstraction function for constant values, and define the various evaluation functions using α when necessary. This will allow us to check soundness later. Note that we have to be careful about the name space and use unique names here.

```
data AbsValue
    = AbsValNum OE
    | AbsValLambda (AbsValueMonad → AbsValueMonad)

instance Show AbsValue where
    show (AbsValNum v) = show v
    show (AbsValLambda _) = "<Abstract Function Value>"

instance Eq AbsValue where
```

```
(≡) (AbsValNum x) (AbsValNum y) = (x ≡ y)
(≡) (AbsValLambda x) (AbsValLambda y) = error "Cannot compare functions."

type AbsValueMonad = Reader AbsEnv AbsValue

data AbsEnv = AbsEnv{ absVariables :: [(String, AbsValue)]}

lookupAbsVal name env = lookup name (absVariables env)

addAbsVal b env = AbsEnv{ absVariables = b : (absVariables env)}

α x = AbsValNum $ if (odd x) then Odd else Even
```

Now define evaluation functions for each element of the AST:

```
phi1Const (EConst x) = return (α x)

aPlus (AbsValNum x) (AbsValNum y) = AbsValNum (oePlus x y)
phi1Add (EAdd x1 x2) = liftM2 aPlus x1 x2
phi1Sub (ESub x1 x2) = liftM2 aPlus x1 x2

aTimes (AbsValNum x) (AbsValNum y) = AbsValNum (oeTimes x y)
phi1Mult (EMult x1 x2) = liftM2 aTimes x1 x2
phi1Div (EDiv x1 x2) = liftM2 aTimes x1 x2

phi1Lambda (ELambda s _ t) =
    do { env ← ask
       ; return $ AbsValLambda (λv → (do { v′ ← v
                                          ; (local (const (addAbsVal (s, v′) env)) t)
                                          }))}

phi1App (EApp x1 x2) =
    do { x1′ ← x1
       ; case x1′ of
         (AbsValLambda f) → (f x2)
         _ → error "Cannot apply non-lambda value"
       }

phi1Var (EVar s) = do { v ← asks (lookupAbsVal s)
                      ; case v of
                        (Just x) → return x
                        Nothing → error "Variable not found"
                      }
```

Finally, create the algebra structures for each term and sum them together to create the term algebra. The helper function *evalPar* is identical to *evalFun* defined previously except it uses the abstract interpreter.

```
term1Alg = (AlgC phi1Const)
          @+@ (AlgAdd phi1Add phi1Sub)
          @+@ (AlgMult phi1Mult phi1Div)
```

```
        @+@ (AlgFun phi1Lambda phi1App phi1Var)

evalPar = runReader ∘ (cata term1Alg)
```

We're done. Now we can use the same terms as before to test the new abstract interpreter.

One more interesting thing before we quit is evaluating soundness of the abstract interpretation. Specifically, does applying the abstraction function after interpretation result in the same value as abstract interpretation? If so, then $\alpha(\phi_c(m)) = \phi_a(m)$ where ϕ_c is concrete interpretation and ϕ_a is abstract interpretation.

```
soundTest c a α x = do { x' ← cata c x
                       ; x'' ← cata a x
                       ; return (x'' ≡ (α x'))
                       }

sound x = case v of
         (ValNum a) → case av of
                        c@(AbsValNum b) → α a ≡ c
                        _ → False
         (ValLambda _) → case av of
                           (AbsValLambda _) → error "Cannot compare functions"
                           _ → False
   where v = (evalFun x Env{variables = [ ]});
         av = (evalPar x AbsEnv{absVariables = [ ]})
```

Now let's have even more fun. Using the same algebraic structure, we can define a type checker for our tiny language by once again defining a new value space and associated ϕ functions. We'll leave it to the reader to determine what's going on here.

```
data TyValue
   = TyInt
   | TyValue :->: TyValue
    deriving (Eq, Show)

data Gamma = Gamma{γ :: [(String, TyValue)]}

lookupTy name gam = lookup name (γ gam)

addBinding b gam = Gamma{γ = b : (γ gam)}

type TyMonad = Reader Gamma TyValue

tyConst (EConst x) = return TyInt

tPlus TyInt TyInt = TyInt
tPlus (_ :->: _) _ = error "Cannot add function value"
tPlus _ (_ :->: _) = error "Cannot add function value"

tSub TyInt TyInt = TyInt
tSub (_ :->: _) _ = error "Cannot subtract function value"
tSub _ (_ :->: _) = error "Cannot subtract function value"
```

$tyAdd\ (EAdd\ x1\ x2) = liftM2\ tPlus\ x1\ x2$
$tySub\ (ESub\ x1\ x2) = liftM2\ tSub\ x1\ x2$

$tMult\ TyInt\ TyInt = TyInt$
$tMult\ (\_ : - >: \_)\ \_ = error$ "Cannot multiply function value"
$tMult\ \_\ (\_ : - >: \_) = error$ "Cannot multiply function value"

$tDiv\ TyInt\ TyInt = TyInt$
$tDiv\ (\_ : - >: \_)\ \_ = error$ "Cannot divide function value"
$tDiv\ \_\ (\_ : - >: \_) = error$ "Cannot divide function value"

$tyMult\ (EMult\ x1\ x2) = liftM2\ tMult\ x1\ x2$
$tyDiv\ (EDiv\ x1\ x2) = liftM2\ tDiv\ x1\ x2$

$tyLambda\ (ELambda\ s\ ty\ t) =$
  do $\{ g \leftarrow ask$
    $; t' \leftarrow (local\ (const\ (addBinding\ (s, TyInt)\ g))\ t)$
    $; return\ (ty : - >: t')$
    $\}$

$tyApp\ (EApp\ x1\ x2) =$
  do $\{ x1' \leftarrow x1$
    $; x2' \leftarrow x2$
    ; **case** $x1'$ **of**
     $(t1 : - >: t2) \rightarrow$ **if** $(t1 \equiv x2')$
                then $(return\ t2)$
                else $(error$ "Input parameter of wrong type"$)$
     $\_ \rightarrow error$ "Cannot apply non-lambda value"
    $\}$

$tyVar\ (EVar\ s) =$ **do** $\{ v \leftarrow asks\ (lookupTy\ s)$
         ; **case** v **of**
         $(Just\ x) \rightarrow return\ x$
         $Nothing \rightarrow error$ "Variable not found"
         $\}$

$tyAlg = (AlgC\ tyConst)$
    $@+@(AlgAdd\ tyAdd\ tySub)$
    $@+@(AlgMult\ tyMult\ tyDiv)$
    $@+@(AlgFun\ tyLambda\ tyApp\ tyVar)$

$typeof = runReader \circ (cata\ tyAlg)$

That's it. The type checker in less than a page. Not bad at all.

The remaining definitions are helper functions for creating terms and calling *cata* to perform evaluation. All are worth looking at to see the structure of terms and to see the use of monads during evaluation.

$sright = S \circ Right$

$sleft = S \circ Left$

```
term_1 = mkEConst 1
term_2 = mkEAdd term_1 term_1
term_3 = mkEMult term_2 term_2
term_4 = mkESub term_1 term_1
term_5 = mkEDiv term_1 term_1
term_6 = mkEVar "x"
term_7 = mkELambda "x" TyInt term_6
term_8 = mkEApp term_7 term_1
term_9 = mkELambda "x" TyInt (mkEAdd term_6 term_6)
term_10 = mkEApp term_9 term_1
term11 = mkELambda "x" TyInt (mkELambda "y" TyInt (mkEAdd term_1 term_1))
term12 = mkEApp (mkEApp term11 term_1) term_1

emptyG = Gamma{γ = [ ]}
emptyE = Env{variables = [ ]}
emptyAE = AbsEnv{absVariables = [ ]}
```

I always forget the definitions of *liftM* and *liftM2*, so I'll include them here in a specification block for reference.

```
    -- Defined in Control.Monad
liftM2 f m1 m2 = do x1 ← m1
                    x2 ← m2
                    return (f x1 x2)

liftM f m1 = do x ← m
                return (f x)

toTmLang :: (SubType f TermType) ⇒ f TermLang → TermLang
toTmLang = inj
```

7 Term Libraries

Included with the base *InterpreterLib* system are a collection of Imodules for building various terms and data structures. These Ilibraries simply provide boilerplate for structuring algebras. They Ido note define semantics for the abstract syntax structures they Idefine. The libraries are intended to serve as both documentation Iand building blocks for interpreters.

7.1 Arithmetic Terms

```
{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest

module InterpreterLib.Terms.ArithTerm where
import InterpreterLib.Algebras
import InterpreterLib.Functors
```

```haskell
data ArithTerm x = Add x x
                 | Sub x x
                 | Mult x x
                 | Div x x
                 | NumEq x x
                 | Num Int

instance Functor ArithTerm where
  fmap f (Add x y) = Add (f x) (f y)
  fmap f (Sub x y) = Sub (f x) (f y)
  fmap f (Mult x y) = Mult (f x) (f y)
  fmap f (Div x y) = Div (f x) (f y)
  fmap f (NumEq x y) = NumEq (f x) (f y)
  fmap f (Num x) = Num x

instance ZipFunctor ArithTerm where
  zipFunctor f (Add x y) (Add a b) = return $ Add (f x a) (f y b)
  zipFunctor f (Sub x y) (Sub a b) = return $ Sub (f x a) (f y b)
  zipFunctor f (Mult x y) (Mult a b) = return $ Mult (f x a) (f y b)
  zipFunctor f (Div x y) (Div a b) = return $ Div (f x a) (f y b)
  zipFunctor f (NumEq x y) (NumEq a b) = return $ NumEq (f x a) (f y b)
  zipFunctor f (Num x) (Num y) = return $ Num x
  zipFunctor f _ _ = fail "No match"

data ArithTermAlgebra a = ArithTermAlgebra{ add :: AlgSig ArithTerm a,
                                            sub :: AlgSig ArithTerm a,
                                            mult :: AlgSig ArithTerm a,
                                            divide :: AlgSig ArithTerm a,
                                            numEq :: AlgSig ArithTerm a,
                                            num :: AlgSig ArithTerm a
                                          }

instance Algebra ArithTerm ArithTermAlgebra a where
  apply alg t@(Add _ _) = add alg t
  apply alg t@(Sub _ _) = sub alg t
  apply alg t@(Mult _ _) = mult alg t
  apply alg t@(Div _ _) = divide alg t
  apply alg t@(NumEq _ _) = numEq alg t
  apply alg t@(Num _) = num alg t

instance AlgebraBuilder ArithTerm (ArithTerm a -> a) ArithTermAlgebra a where
  mkAlgebra f = ArithTermAlgebra f f f f f f

data BinOp = AddOp | SubOp | MultOp | DivOp | NumEqOp

decodeOp = fst . decode
decodeArgs = snd . decode

decode (Add x y) = (AddOp, (x, y))
decode (Sub x y) = (SubOp, (x, y))
decode (Mult x y) = (MultOp, (x, y))
```

```
decode (Div x y) = (DivOp, (x, y))
decode (NumEq x y) = (NumEqOp, (x, y))

mkAdd = mkTerm2 Add
mkSub = mkTerm2 Sub
mkMult = mkTerm2 Mult
mkDiv = mkTerm2 Div
mkNumEq = mkTerm2 NumEq
mkNum = mkTerm Num
```

7.2 Fixed Point Term

```
{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Terms.FixTerm where

import InterpreterLib.Algebras
import InterpreterLib.Functors

data FixTerm x = FixTerm x

instance Functor FixTerm where
  fmap f (FixTerm x) = FixTerm (f x)

instance ZipFunctor FixTerm where
  zipFunctor f (FixTerm x) (FixTerm y) = return (FixTerm (f x y))

data FixTermAlgebra a = FixTermAlgebra{fixTerm :: AlgSig FixTerm a}

instance Algebra FixTerm FixTermAlgebra a where
  apply alg t = fixTerm alg t

instance AlgebraBuilder FixTerm (AlgSig FixTerm a) FixTermAlgebra a where
  mkAlgebra φ = FixTermAlgebra φ

mkFix = mkTerm FixTerm
```

7.3 IO Terms

```
{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Terms.IOTerm where

import InterpreterLib.Algebras
import Control.Monad (liftM)
import InterpreterLib.Functors
```

```haskell
data IOTerm a = WriteIO a
              | ReadIO

instance Functor IOTerm where
  fmap f (WriteIO x) = WriteIO (f x)
  fmap f ReadIO = ReadIO

instance ZipFunctor IOTerm where
  zipFunctor f (WriteIO x) (WriteIO y) = return $ WriteIO (f x y)
  zipFunctor f ReadIO ReadIO = return ReadIO
  zipFunctor _ _ _ = fail "zipFunctor"

data IOTermAlgebra a = IOTermAlgebra{ writeIOTerm :: IOTerm a -> a,
                                      readIOTerm :: IOTerm a -> a
                                    }

instance Algebra IOTerm IOTermAlgebra a where
  apply alg t@(WriteIO x) = writeIOTerm alg t
  apply alg t@ReadIO = readIOTerm alg t

instance AlgebraBuilder IOTerm (IOTerm a -> a) IOTermAlgebra a where
  mkAlgebra φ = IOTermAlgebra φ φ

mkWrite = mkTerm WriteIO
mkRead = mkTerm0 ReadIO
```

7.4 If Term

```haskell
{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Terms.IfTerm where

import InterpreterLib.Algebras
import InterpreterLib.Functors

data IfTerm a = IfTerm a a a
              | TrueTerm
              | FalseTerm

instance Functor IfTerm where
  fmap f (IfTerm x y z) = IfTerm (f x) (f y) (f z)
  fmap f TrueTerm = TrueTerm
  fmap f FalseTerm = FalseTerm

instance ZipFunctor IfTerm where
  zipFunctor f (IfTerm a b c) (IfTerm x y z) = return $ IfTerm (f a x) (f b y) (f c z)
  zipFunctor f TrueTerm TrueTerm = return TrueTerm
  zipFunctor f FalseTerm FalseTerm = return FalseTerm
  zipFunctor f _ _ = fail "ZipFunctor: Unlike constructors"
```

```
data IfTermAlgebra a = IfTermAlgebra{ ifTerm :: IfTerm a → a,
                                      trueTerm :: IfTerm a → a,
                                      falseTerm :: IfTerm a → a
                                    }

instance Algebra IfTerm IfTermAlgebra a where
  apply alg t@(IfTerm _ _ _) = ifTerm alg t
  apply alg t@TrueTerm = trueTerm alg t
  apply alg t@FalseTerm = falseTerm alg t

instance AlgebraBuilder IfTerm (IfTerm a → a) IfTermAlgebra a where
  mkAlgebra φ = IfTermAlgebra φ φ φ


mkIf = mkTerm3 IfTerm
mkTrue = mkTerm0 TrueTerm
mkFalse = mkTerm0 FalseTerm
```

7.5 Reference Terms

```
{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Terms.ImperativeTerm where

import InterpreterLib.Functors
import InterpreterLib.Algebras

data ImperativeTerm x = NewRef x
                      | DeRef x
                      | SeqTerm x x


instance Functor ImperativeTerm where
  fmap f (NewRef x) = NewRef (f x)
  fmap f (DeRef x) = DeRef (f x)
  fmap f (SeqTerm x y) = SeqTerm (f x) (f y)

instance ZipFunctor ImperativeTerm where
  zipFunctor f (NewRef x) (NewRef y) = return $ NewRef (f x y)
  zipFunctor f (DeRef x) (DeRef y) = return $ DeRef (f x y)
  zipFunctor f (SeqTerm x y) (SeqTerm u v) = return $ SeqTerm (f x u) (f y v)

data ImperativeTermAlgebra a =
    ImperativeTermAlgebra{ newRef :: ImperativeTerm a → a,
                           deRef :: ImperativeTerm a → a,
                           seqTerm :: ImperativeTerm a → a
                         }

instance Algebra ImperativeTerm ImperativeTermAlgebra a where
  apply alg t@(NewRef _) = newRef alg t
  apply alg t@(DeRef _) = deRef alg t
```

```
  apply alg t@(SeqTerm _ _) = seqTerm alg t


instance AlgebraBuilder ImperativeTerm
                        (ImperativeTerm a → a)
                        ImperativeTermAlgebra a where

  mkAlgebra f = ImperativeTermAlgebra f f f

mkNewRef = mkTerm NewRef
mkDeRef = mkTerm DeRef
mkSeqTerm = mkTerm2 SeqTerm
```

7.6 Lambda Terms

```
{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Terms.LambdaTerm (LambdaTerm (..),
                                        LambdaTermAlgebra (..),
                                        LambdaTermModule, lambdaModule) where

import InterpreterLib.Algebras
import InterpreterLib.Functors
import InterpreterLib.Modules

data LambdaTerm ty x = App x x
                     | Lam String ty x

instance Functor (LambdaTerm ty) where
  fmap f (App x y) = App (f x) (f y)
  fmap f (Lam s ty x) = Lam s ty (f x)

instance ZipFunctor (LambdaTerm ty) where
  zipFunctor f (App a b) (App x y) = return $ App (f a x) (f b y)
  zipFunctor f (Lam n ty x) (Lam _ _ y) = return $ Lam n ty (f x y)


data LambdaTermAlgebra ty a = LambdaTermAlgebra{ app :: LambdaTerm ty a → a,
                                                 lam :: LambdaTerm ty a → a
                                               }

instance Algebra (LambdaTerm ty) (LambdaTermAlgebra ty) a where
  apply alg t@(App _ _) = app alg t
  apply alg t@(Lam _ _ _) = lam alg t

instance AlgebraBuilder (LambdaTerm ty) (LambdaTerm ty a → a) (LambdaTermAlgebra ty) a where
  mkAlgebra f = LambdaTermAlgebra f f

{- data LambdaTermModule ty tm = LambdaTermModule  lambdaConstructor :: String -¿ ty -¿ tm -¿ LambdaTe
```

```
data LTMI ty tm =
  LTMI{ lc :: String → ty → tm → LambdaTerm ty tm,
        ac :: tm → tm → LambdaTerm ty tm
      }
```

```
data LambdaTermModule ty tm =
  LambdaTermModule{ getMkLambda :: String → ty → tm → tm,
                    getMkApp :: tm → tm → tm
                  }
```

```
mkModule (x :: ty) (y :: tm) = let mod :: LTMI ty tm
                                   mod = LTMI Lam App
                               in LambdaTermModule (mkTerm3 $ lc mod) (mkTerm2 $ ac mod)

lambdaModule = mkModule ⊥ ⊥
```

```
instance Module (LambdaTermModule ty tm) (String → ty → tm → tm,
                                          tm → tm → tm) where
  open (LambdaTermModule l a) = (l, a)
```

7.7 Let Term

```
{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Terms.LetTerm where

import InterpreterLib.Algebras
import InterpreterLib.Functors

type Name = String
data LetTerm ty a = LetTerm [(Name, ty, a)] a
                  | LetRecTerm [(Name, ty, a)] a
```

```
instance Functor (LetTerm ty) where
  fmap f (LetTerm bindings body) = LetTerm (map (λ(n, ty, v) → (n, ty, f v)) bindings) (f body)
  fmap f (LetRecTerm bindings body) = LetRecTerm (map (λ(n, ty, v) → (n, ty, f v)) bindings) (f body)
```

```
instance ZipFunctor (LetTerm ty) where
  zipFunctor f (LetTerm bs1 b1) (LetTerm bs2 b2) =
      return $ LetTerm (zipWith fun bs1 bs2) (f b1 b2)
    where fun (n1, ty, v1) (n2, _, v2) = (n1, ty, (f v1 v2))
```

```
  zipFunctor f (LetRecTerm bs1 b1) (LetRecTerm bs2 b2) =
      return $ LetRecTerm (zipWith fun bs1 bs2) (f b1 b2)
    where fun (n1, ty, v1) (n2, _, v2) = (n1, ty, (f v1 v2))

data LetTermAlgebra ty a = LetTermAlgebra{ letTerm :: AlgSig (LetTerm ty) a,
                                           letRecTerm :: AlgSig (LetTerm ty) a
                                         }

instance Algebra (LetTerm ty) (LetTermAlgebra ty) a where
  apply alg t@(LetTerm _ _) = letTerm alg t
  apply alg t@(LetRecTerm _ _) = letRecTerm alg t

instance AlgebraBuilder (LetTerm ty) (LetTerm ty a → a) (LetTermAlgebra ty) a where
  mkAlgebra φ = LetTermAlgebra φ φ

mkLet = mkTerm2 LetTerm
mkLetRec = mkTerm2 LetRecTerm
```

7.8 Procedure Term

```
{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Terms.ProcTerm where

import InterpreterLib.Algebras
import InterpreterLib.Functors

type Name = String
data ProcTerm x = Procedure Name [Name] x |
                  ProcCall x [x]

newtype ProcValue v = ProcValue ([v] → v)

instance Functor ProcTerm where
  fmap f (Procedure procname ns body) = Procedure procname ns (f body)
  fmap f (ProcCall fun args) = ProcCall (f fun) (map f args)


instance ZipFunctor ProcTerm where
  zipFunctor f (Procedure m ms b1) (Procedure n ns b2)
      | m ≡ n ∧ ms ≡ ns = return $ Procedure m ms (f b1 b2)
      | otherwise = fail "zipFunctor"
  zipFunctor f (ProcCall f1 a1) (ProcCall f2 a2) =
      return $ ProcCall (f f1 f2) (zipWith f a1 a2)

data ProcTermAlgebra a = ProcTermAlgebra{ procTerm :: ProcTerm a → a,
                                          callTerm :: ProcTerm a → a
                                        }

instance Algebra ProcTerm ProcTermAlgebra a where
  apply alg t@(Procedure _ _ _) = (procTerm alg) t
  apply alg t@(ProcCall _ _) = (callTerm alg) t
```

```
instance AlgebraBuilder ProcTerm (ProcTerm a → a) ProcTermAlgebra a where
  mkAlgebra f = ProcTermAlgebra f f



mkProc = mkTerm3 Procedure
mkCall = mkTerm2 ProcCall
```

7.9 RAL Term

```
{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Terms.RALTerm where

import InterpreterLib.Algebras
import InterpreterLib.Functors
import InterpreterLib.Terms.VarTerm


type RegionVar = String
data Place = RegionVar | Deallocated
data RALTerm x = RApp x Place
               | NewRegion RegionVar x
               | RegionAbs RegionVar x
               | At x Place

instance Functor RALTerm where
  fmap f (RApp x place) = RApp (f x) place
  fmap f (NewRegion v x) = NewRegion v (f x)
  fmap f (RegionAbs v x) = RegionAbs v (f x)
  fmap f (At x place) = At (f x) place


instance ZipFunctor RALTerm where
  zipFunctor f (RApp x place) (RApp y _) = return $ RApp (f x y) place
  zipFunctor f (NewRegion v x) (NewRegion _ y) = return $ NewRegion v (f x y)
  zipFunctor f (RegionAbs v x) (RegionAbs _ y) = return $ RegionAbs v (f x y)
  zipFunctor f (At x place) (At y _) = return $ At (f x y) place

data RALTermAlgebra a = RALTermAlgebra{ rApp :: RALTerm a → a,
                                        newRegion :: RALTerm a → a,
                                        regionAbs :: RALTerm a → a,
                                        at :: RALTerm a → a
                                      }

instance Algebra RALTerm RALTermAlgebra a where
  apply alg t@(RApp _ _) = rApp alg t
  apply alg t@(NewRegion _ _) = newRegion alg t
  apply alg t@(RegionAbs _ _) = regionAbs alg t
  apply alg t@(At _ _) = at alg t
```

```
instance AlgebraBuilder RALTerm (RALTerm a → a) RALTermAlgebra a where
   mkAlgebra φ = RALTermAlgebra φ φ φ φ

    -- mkRApp t place = inn injF RApp t place
mkRApp = mkTerm2 RApp
mkNewRegion = mkTerm2 NewRegion
mkRegionAbs = mkTerm2 RegionAbs
mkAt = mkTerm2 At
```

7.10 Record Terms

```
 {-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Terms.RecordTerm where

import InterpreterLib.Algebras
import InterpreterLib.Functors

data RecordTerm x = RecordTerm [x]
                  | ProjTerm x Int


instance Functor RecordTerm where
   fmap f (RecordTerm fields) = RecordTerm (fmap f fields)
   fmap f (ProjTerm x field) = ProjTerm (f x) field

instance ZipFunctor RecordTerm where
   zipFunctor f (RecordTerm fields) (RecordTerm fields′) =
         return (RecordTerm (zipWith f fields fields′))
   zipFunctor f (ProjTerm x l) (ProjTerm y l′) | l ≡ l′ = return (ProjTerm (f x y) l)
                                              | otherwise = fail "Field labels don't match"
   zipFunctor f _ _ = fail "ZipFunctor: Unlike constructors"

data RecordTermAlgebra a = RecordTermAlgebra{ recordTerm :: AlgSig RecordTerm a,
                                              projTerm :: AlgSig RecordTerm a}


instance Algebra RecordTerm RecordTermAlgebra a where
   apply alg t@(RecordTerm _) = recordTerm alg t
   apply alg t@(ProjTerm _ _) = projTerm alg t


instance AlgebraBuilder RecordTerm (AlgSig RecordTerm a) RecordTermAlgebra a where
   mkAlgebra φ = RecordTermAlgebra φ φ

mkRecord = mkTerm RecordTerm
mkProj = mkTerm2 ProjTerm
```

7.11 String Terms

```
{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Terms.StringTerm where

import InterpreterLib.Algebras
import InterpreterLib.Functors

data StringTerm a = StringTerm String

instance Functor StringTerm where
  fmap f (StringTerm s) = (StringTerm s)

instance ZipFunctor StringTerm where
  zipFunctor f (StringTerm x) (StringTerm y) | x ≡ y = return $ StringTerm x
                                             | otherwise = fail "zipFunctor"

data StringTermAlgebra a = StringTermAlgebra{ stringTerm :: StringTerm a → a }

instance Algebra StringTerm StringTermAlgebra a where
  apply alg t@(StringTerm _) = stringTerm alg t

instance AlgebraBuilder StringTerm (StringTerm a → a) StringTermAlgebra a where
  mkAlgebra φ = StringTermAlgebra φ

mkString = mkTerm StringTerm
```

7.12 Sum Terms

```
{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Terms.SumTerm (SumTerm (..),
                                     SumTermModule, sumModule) where

import InterpreterLib.Algebras
import InterpreterLib.Functors
import InterpreterLib.Modules

data SumTerm ty x = SumLeft x ty
                  | SumRight x ty
                  | SumCase x (String, x) (String, x)

instance Functor (SumTerm ty) where
  fmap f (SumLeft x ty) = SumLeft (f x) ty
  fmap f (SumRight x ty) = SumRight (f x) ty
  fmap f (SumCase x (v1, y) (v2, z)) = SumCase (f x) (v1, (f y)) (v2, (f z))
```

```
instance ZipFunctor (SumTerm ty) where
  zipFunctor f (SumLeft x ty) (SumLeft y ty') = return (SumLeft (f x y) ty)
  zipFunctor f (SumRight x ty) (SumRight y ty') = return (SumRight (f x y) ty)
  zipFunctor f (SumCase x (v1, y) (v2, z)) (SumCase x' (_, y') (_, z')) =
      return (SumCase (f x x') (v1, (f y y')) (v2, (f z z')))
  zipFunctor f _ _ = fail "ZipFunctor: Unlike constructors"

data SumTermAlgebra ty a = SumTermAlgebra{ sumLeft :: AlgSig (SumTerm ty) a,
                                           sumRight :: AlgSig (SumTerm ty) a,
                                           sumCase :: AlgSig (SumTerm ty) a
                                         }
instance Algebra (SumTerm ty) (SumTermAlgebra ty) a where
  apply alg t@(SumLeft _ _) = sumLeft alg t
  apply alg t@(SumRight _ _) = sumRight alg t
  apply alg t@(SumCase _ _ _) = sumCase alg t


instance AlgebraBuilder (SumTerm ty) (AlgSig (SumTerm ty) a) (SumTermAlgebra ty) a where
  mkAlgebra φ = SumTermAlgebra φ φ φ

getMkLeft = mkTerm2 SumLeft
getMkRight = mkTerm2 SumRight
getMkCase = mkTerm3 SumCase



data STMI ty tm = STMI{ lc :: tm → ty → SumTerm ty tm,
                        rc :: tm → ty → SumTerm ty tm,
                        cc :: tm → (String, tm) → (String, tm) → SumTerm ty tm
                      }

data SumTermModule ty tm = SumTermModule (STMSig ty tm)

sumModule = mkModule ⊥ ⊥
  where mkModule (x :: ty) (y :: tm) = let mod :: STMI ty tm
                                           mod = STMI SumLeft SumRight SumCase
                                       in SumTermModule ((mkTerm2 $ lc mod),
                                                         (mkTerm2 $ rc mod),
                                                         (mkTerm3 $ cc mod))


type STMSig ty tm = (tm → ty → tm,
                     tm → ty → tm,
                     tm → (String, tm) → (String, tm) → tm)

instance Module (SumTermModule ty tm) (STMSig ty tm) where
  open (SumTermModule t) = t
```

7.13 Unit Term

{-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest

```
module InterpreterLib.Terms.UnitTerm where

import InterpreterLib.Algebras
import InterpreterLib.Functors

data UnitTerm x = UnitTerm

instance Functor UnitTerm where
  fmap f UnitTerm = UnitTerm



instance ZipFunctor UnitTerm where
  zipFunctor f UnitTerm UnitTerm = return UnitTerm

data UnitTermAlgebra a = UnitTermAlgebra{ unitTerm :: AlgSig UnitTerm a}

instance Algebra UnitTerm UnitTermAlgebra a where
  apply alg t = unitTerm alg t



instance AlgebraBuilder UnitTerm (AlgSig UnitTerm a) UnitTermAlgebra a where
  mkAlgebra φ = UnitTermAlgebra φ

mkUnit = mkTerm0 UnitTerm
```

7.14 Variable Terms

```
 {-# OPTIONS -fglasgow-exts -fallow-overlapping-instances -fallow-undecidable-instances -fno-monomorphism-rest
module InterpreterLib.Terms.VarTerm where

import InterpreterLib.Algebras
import InterpreterLib.Functors

type Name = String
data VarTerm a = VarTerm Name | DummyTerm a


instance Functor VarTerm where
  fmap f (VarTerm n) = VarTerm n

instance ZipFunctor VarTerm where
  zipFunctor _ (VarTerm x) (VarTerm y) | x ≡ y = return (VarTerm x)
                                       | otherwise = fail "Non-matching names"

data VarTermAlgebra a = VarTermAlgebra{ varTerm :: VarTerm a → a
                                      }

instance Algebra VarTerm VarTermAlgebra a where
```

$$apply\ alg\ t@(VarTerm\ \_) = varTerm\ alg\ t$$

$$\textbf{instance}\ AlgebraBuilder\ VarTerm\ (VarTerm\ a \to a)\ VarTermAlgebra\ a\ \textbf{where}$$
$$\quad mkAlgebra\ \phi = VarTermAlgebra\ \phi$$

$$mkVar = mkTerm\ VarTerm$$

8 Usage

This file is a template for transforming literate script into LaTeXand is not actually a `Haskell` interpreter implementation. Each section in this file is a separate module that can be loaded individually for experimentation.

Note that the interpreters have been developed under GHC and some require turning on the Glasgow Extensions. Your mileage may vary if you're using HUGS.

To build a LaTeXdocument from the interpreter files, use:

```
lhs2TeX --math InterpreterLib.lhs > InterpreterLib.tex
```

and run LaTeXon the result.

References

[1] Luc Duponcheel. Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters., 1995.

[2] David Espinosa. *Semantic Lego*. PhD thesis, Yale University, 1995.

[3] Mark P. Jones and Luc Duponcheel. Composing monads. Research report YALEU/DCS/RR-1004, Yale University, Yale University, New Haven, Connecticut, Dec 1993.

[4] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058, pages 219–234. Springer-Verlag, 1996.

[5] Guy L. Steele. Building interpreters by composing monads. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, NY, USA, 1994. ACM Press.