InterpreterLib Examples

Uk'taad B'mal
Information and Telecommunications Technology Center
The University of Kansas
2335 Irving Hill Rd, Lawrence, KS 66045

November 18, 2005

Abstract

The purpose of this document is to provide an annoted example using the *InterprterLib* package to define simple interpreters. The document assumes some knowledge of modular interpreter construction and refers to earlier interpreters written by the uktab ad'bmal group. These are available upon request.

1 Introduction

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

2 Example Language

To demonstrate the use of the *InterpreterLib* definition capabilities, we will define an interpreter for an *Integer* language that implements some simple mathematical operations.

2.1 The Value Space

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>"
```

There are two values in this language, integers and integer functions.

2.2 Defining AST Elements

Next, we define types for each of the language's AST elements. 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.

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.

We will 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
    map_f f (EConst x) = EConst x

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

mkEConst = inn ∘ sleft ∘ EConst
```

Let's step through the definition for *Integer*. *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 from earlier interpreters. 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 map_f 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
    map_f f (EAdd x y) = (EAdd (f x) (f y))
    map_f 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
    map_f f (EMult x y) = (EMult (f x) (f y))
    map_f 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)

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

instance Functor ExprFun where
    map_f f (ELambda s ty t) = ELambda s ty (f t)
    map_f f (EApp t1 t2) = EApp (f t1) (f t2)
    map_f 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.

2.3 Composing AST Elements

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 from previous interpreters. *TermType* is the sum of term definitions and *TermLang* is the fixed point of the term definition.[1]

$$\textbf{type}\ \mathit{TermType} = (\mathit{ExprConst} :\$: (\mathit{ExprAdd} :\$: (\mathit{ExprMult} :\$: \mathit{ExprFun})))$$

$$\textbf{type}\ \mathit{TermLang} = \mathit{Fix}\ \mathit{TermType}$$

3 Language Semantics

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 an 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.

$$\textbf{type}\ \mathit{ValueMonad} = \mathit{Reader}\ \mathit{Env}\ \mathit{Value}$$

$$\textbf{data}\ \mathit{Env} = \mathit{Env}\{\mathit{variables} :: [(\mathit{String}, \mathit{Value})]\}$$

$$\mathit{lookupVal}\ \mathit{name}\ \mathit{env} = \mathit{lookup}\ \mathit{name}\ (\mathit{variables}\ \mathit{env})$$

$$\mathit{addVal}\ b\ \mathit{env} = \mathit{Env}\{\mathit{variables} = b : (\mathit{variables}\ \mathit{env})\}$$

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

3.1 Semantic Elements

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 (\v -> (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.

3.2 Composign Semantic Elements

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.[2]

$$
\begin{aligned}
\mathit{termAlg} = {} & (\mathit{AlgC}\ \mathit{phiConst}) \\
& @+@\,(\mathit{AlgAdd}\ \mathit{phiAdd}\ \mathit{phiSub}) \\
& @+@\,(\mathit{AlgMult}\ \mathit{phiMult}\ \mathit{phiDiv}) \\
& @+@\,(\mathit{AlgFun}\ \mathit{phiLambda}\ \mathit{phiApp}\ \mathit{phiVar})
\end{aligned}
$$

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*.

$$\mathit{evalFun} = \mathit{runReader} \circ (\mathit{cata}\ \mathit{termAlg})$$

An initial value for the environment must be provided to *evalFun* for the reader to evaluate completely. To evaluate *term1* starting with an empty environment, execute the following:

$$(\mathit{evalFun}\ \mathit{term1})\ \mathit{Env}\{\mathit{variables} = [\,]\}$$

4 A Second Interpreter

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.

$$\textbf{data}\ \mathit{OE} = \mathit{Odd} \mid \mathit{Even}\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})$$

[2] 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.

```
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 *alpha*, the abstraction function for constant values, and define the various evaluation functions using *alpha* 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)}

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

Now define evaluation functions for each element of the AST:

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

$$
\begin{array}{l}
\mathit{aPlus}\ (\mathit{AbsValNum}\ x)\ (\mathit{AbsValNum}\ y) = \mathit{AbsValNum}\ (\mathit{oePlus}\ x\ y) \\
\mathit{phi1Add}\ (\mathit{EAdd}\ \mathit{x1}\ \mathit{x2}) = \mathit{liftM2}\ \mathit{aPlus}\ \mathit{x1}\ \mathit{x2} \\
\mathit{phi1Sub}\ (\mathit{ESub}\ \mathit{x1}\ \mathit{x2}) = \mathit{liftM2}\ \mathit{aPlus}\ \mathit{x1}\ \mathit{x2}
\end{array}
$$

$$
\begin{array}{l}
\mathit{aTimes}\ (\mathit{AbsValNum}\ x)\ (\mathit{AbsValNum}\ y) = \mathit{AbsValNum}\ (\mathit{oeTimes}\ x\ y) \\
\mathit{phi1Mult}\ (\mathit{EMult}\ \mathit{x1}\ \mathit{x2}) = \mathit{liftM2}\ \mathit{aTimes}\ \mathit{x1}\ \mathit{x2} \\
\mathit{phi1Div}\ (\mathit{EDiv}\ \mathit{x1}\ \mathit{x2}) = \mathit{liftM2}\ \mathit{aTimes}\ \mathit{x1}\ \mathit{x2}
\end{array}
$$

$$
\begin{array}{l}
\mathit{phi1Lambda}\ (\mathit{ELambda}\ s\ \_\ t) = \\
\quad \mathbf{do}\ \{\, \mathit{env} \leftarrow \mathit{ask} \\
\quad\quad ;\mathit{return}\ \$\ \mathit{AbsValLambda}\ (\lambda v \rightarrow (\mathbf{do}\ \{\, v' \leftarrow v \\
\quad\quad\quad\quad ;(\mathit{local}\ (\mathit{const}\ (\mathit{addAbsVal}\ (s, v')\ \mathit{env}))\ t) \\
\quad\quad\quad\quad \})) \}
\end{array}
$$

$$
\begin{array}{l}
\mathit{phi1App}\ (\mathit{EApp}\ \mathit{x1}\ \mathit{x2}) = \\
\quad \mathbf{do}\ \{\, \mathit{x1}' \leftarrow \mathit{x1} \\
\quad\quad ;\mathbf{case}\ \mathit{x1}'\ \mathbf{of} \\
\quad\quad\ (\mathit{AbsValLambda}\ f) \rightarrow (f\ \mathit{x2}) \\
\quad\quad\ \_ \rightarrow \mathit{error}\ \texttt{"Cannot apply non-lambda value"} \\
\quad\quad \}
\end{array}
$$

$$
\begin{array}{l}
\mathit{phi1Var}\ (\mathit{EVar}\ s) = \mathbf{do}\ \{\, v \leftarrow \mathit{asks}\ (\mathit{lookupAbsVal}\ s) \\
\quad\quad ;\mathbf{case}\ v\ \mathbf{of} \\
\quad\quad\ (\mathit{Just}\ x) \rightarrow \mathit{return}\ x \\
\quad\quad\ \mathit{Nothing} \rightarrow \mathit{error}\ \texttt{"Variable not found"} \\
\quad\quad \}
\end{array}
$$

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.

$$
\begin{array}{l}
\mathit{term1Alg} = (\mathit{AlgC}\ \mathit{phi1Const}) \\
\quad\quad @+@\ (\mathit{AlgAdd}\ \mathit{phi1Add}\ \mathit{phi1Sub}) \\
\quad\quad @+@\ (\mathit{AlgMult}\ \mathit{phi1Mult}\ \mathit{phi1Div}) \\
\quad\quad @+@\ (\mathit{AlgFun}\ \mathit{phi1Lambda}\ \mathit{phi1App}\ \mathit{phi1Var})
\end{array}
$$

$$
\mathit{evalPar} = \mathit{runReader} \circ (\mathit{cata}\ \mathit{term1Alg})
$$

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

5 Type Checking as Interpretation

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 Γ = Γ{γ :: [(String, TyValue)]}

lookupTy name gam = lookup name (γ gam)

addBinding b gam = Γ{γ = b : (γ gam)}

type TyMonad = Reader Γ 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 ← ask
       ; t' ← (local (const (addBinding (s, TyInt) g)) t)
       ; return (ty :-> t')
       }
```

```
tyApp (EApp x1 x2) =
    do { x1' ← x1
       ; x2' ← x2
       ; case x1' of
        (t1 :->: t2) → if (t1 ≡ x2')
                         then (return t2)
                         else (error "Input parameter of wrong type")
        _ → error "Cannot apply non-lambda value"
       }

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

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

typeof = runReader ∘ (cata tyAlg)
```

That's it. The type checker in just over a page. Not bad at all.

6 A Moment of Soundness

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 alpha x = do { x' ← cata c x
                           ; x'' ← cata a x
                           ; return (x'' ≡ (alpha x'))
                           }

sound x = case v of
          (ValNum a) → case av of
                       c@(AbsValNum b) → alpha a ≡ c
                       _ → False
          (ValLambda _) → case av of
```

```
                        (AbsValLambda _) → error "Cannot compare functions"
                        _ → False
    where v = (evalFun x Env{variables = [ ]});
          av = (evalPar x AbsEnv{absVariables = [ ]})
```

7 Testing Functions

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 ∘ Right

sleft = S ∘ Left

term1 = mkEConst 1
term2 = mkEAdd term1 term1
term3 = mkEMult term2 term2
term4 = mkESub term1 term1
term5 = mkEDiv term1 term1
term6 = mkEVar "x"
term7 = mkELambda "x" TyInt term6
term8 = mkEApp term7 term1
term9 = mkELambda "x" TyInt (mkEAdd term6 term6)
term10 = mkEApp term9 term1
term11 = mkELambda "x" TyInt (mkELambda "y" TyInt (mkEAdd term1 term1))
term12 = mkEApp (mkEApp term11 term1) term1

emptyG = Γ{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)
```

This does not work, so uncomment at your own risk. It's supposed to mimic SubSum from LangUtils, but insists on a fully instantiated type instead of a type constructor.

$$
\begin{aligned}
&\mathit{toTmLang} :: (\mathit{SubType}\ f\ \ \mathit{TermType}) \Rightarrow f\ \ \mathit{TermLang} \rightarrow \mathit{TermLang}\\
&\mathit{toTmLang} = \uparrow
\end{aligned}
$$