# Constructing a Comonadic Stream Processor Using InterpreterLib: Putting the pieces together

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

January 24, 2006

**Abstract**

Abstract goes here...

## 1 Introduction

What we do in this report is take all of our interpreter pieces and put them together to write a stream transformer. First we define a simple expression language that supports lambda abstraction, conditionals, Booleans and integers. This language is in every way identical to earlier languages written with *InterpreterLib*. It is modular and monadic in nature. We will extend the abstract syntax elements from *InterpreterLib* to include an *fby* operation defining the dataflow concept of "followed by". To interpret *fby*, we must introduce data flow semantics. We choose to do this using a comonad as defined by Uustalu and Vene [1]. In effect, we are taking a simple lambda language and inserting it into a comonad to create a stream transformer.

If we take this new stream transformer and evaluate it using the *Id* comonad, the resulting system is simply the original interpreter. The *Id* comonad has no concept of past or future and thus processes a single value stream.

Moving to the *LV* or *Causal* comonad allows us to record the past as stream elements are processes. Specifically, the *LV* comonad accepts an input stream and produces a list representing past values. This is in effect a discrete event simulator where the events are input from the input stream.

Finally, moving to the *LVS* comonad allows us to move backwards and forwards along the input and output streams. *LVS* gives us a simulator that allows stopping, reversing and restarting during execution.

The ultimate question has to be why we care about this. In a way, we have created the ultimate in modular interpreter by introducing temporal events. The ordering of events in the input stream defines the temporal flow of the resulting simulator. *Id* gives us an interpreter because there is no temporal orderin. *LV* gives us causal simulation because we only move forward through the input stream. *LVS* gives us both backward and forward reference providing the most flexible temporal model. All this *without changing the interpreter*. We can change the time reference, explore branching time, or eliminate time altogether by changing the comonad use for evaluation or by changing the nature of the input stream. Thus, we have created a siulator where time is simply one aspect of the composable interpreter like any other.

## 2 Imports and Options

The interpreter imports virtually everything we have written to support interpreter construction. First the *InterpreterLib* package is loaded along with terms used in our abstract syntax. Note the import of *Causal*, a new module providing the *Fby* expression syntax. The *Comonad* packages are then loaded to construct the comonad that sequences interpretation. Finally, the *Monad* modules are loaded to construct the interpreter monadically.

```haskell
module DL where

{-# OPTIONS -fglasgow-exts -fno-monomorphism-restriction -fallow-overlapping-instances #-}

import InterpreterLib.Algebras
import InterpreterLib.Functors
import InterpreterLib.SubType

import InterpreterLib.Terms.Arith
import InterpreterLib.Terms.LambdaTerm
import InterpreterLib.Terms.FixTerm
import InterpreterLib.Terms.IfTerm

import Causal

import Comonad
import Comonad.LV
import Comonad.LVS
import Comonad.Stream

import Control.Monad
import Control.Monad.Reader
```

## 3 Term Space

The term language is a trivial language with the addition of a data flow construct. The first five term types define a trivial lambda language. (See earlier documentation of *InterpreterLib* if this is confusing.) The final term adds the followed by syntax to support moving interpretation through time. As usual, the full language is the fixed point of the non-recursive AST structure.

```haskell
type TermType = (LambdaTerm ()): $ :
                VarTerm: $ :
                FixTerm: $ :
                ArithTerm: $ :
                IfTerm: $ :
                Causal

type TermLang = Fix TermType
```

# 4 Value Space

The value space of tis interpreter includes integers, booleans and lambdas. There is no need for a non-recursive representation here.

```
data Val
  = I Int
  | B Bool
  | F (LV Val → VSpace Val)

instance Show Val where
    show (I i) = show i
    show (B b) = show b
    show (F _) = "<function value>"
λenb{code}

λsection{Environment}

The environment of this interpreter is...

λbegin{code}
type VSpace = Reader (LV Env)
newtype Env = Env [(String, VSpace Val)]
runVSpace = runReader
```

# 5 The Semantic Algebra

At this point we defined the explicit semantic algebra used by the intepreter. Functions for each language construct are defined first and then assembpled into the explicit algebra.

## 5.1 Lambda Evaluation

```
phiLam (Lam v () t) =
    do denv ← ask
       let repair (a, Env env) = Env $ (v, return a) : env
           extendDenv d = cmap repair (czip d denv)
       return $ F (λd → local (const (extendDenv d)) t)

phiLam (App t1 t2) =
    do denv ← ask
       (F f) ← t1
       f $ cobind (runVSpace t2) denv
```

## 5.2 Fixed Point Evaluation

```
phiFix tm@(FixTerm t) =
    do denv ← ask
       (F f) ← t
       f $ cobind (runVSpace (phiFix tm)) denv
```

## 5.3 Variable Evaluation

$$
\begin{array}{l}
\mathit{phiVar} :: (\mathit{VarTerm}\ (\mathit{VSpace}\ \mathit{Val})) \rightarrow (\mathit{VSpace}\ \mathit{Val})\\
\mathit{phiVar}\ (\mathit{VarTerm}\ v) =\\
\quad \textbf{do let}\ \mathit{unJust}\ (\mathit{Just}\ x) = x\\
\qquad\quad \mathit{get}\ (\mathit{Env}\ \mathit{env}) = \mathit{unJust}\ \$\ \mathit{lookup}\ v\ \mathit{env}\\
\qquad\quad \text{-- asks (get . counit)}\\
\qquad \mathit{denv} \leftarrow \mathit{ask}\\
\qquad \mathit{get}\ (\mathit{counit}\ \mathit{denv})
\end{array}
$$

$$\mathit{lift2Int2}\ \mathit{op}\ (I\ x)\ (I\ y) = I\ \$\ x\ `\mathit{op}`\ y$$

$$\mathit{lift2Int2Bool2}\ \mathit{op}\ (I\ x)\ (I\ y) = B\ \$\ x\ `\mathit{op}`\ y$$

## 5.4 Arithmetic Expression Evaluation

$$
\begin{array}{l}
\mathit{phiArith}\ (\mathit{Add}\ x\ y) = \mathit{liftM2}\ (\mathit{lift2Int2}\ (+))\ x\ y\\
\mathit{phiArith}\ (\mathit{Sub}\ x\ y) = \mathit{liftM2}\ (\mathit{lift2Int2}\ (-))\ x\ y\\
\mathit{phiArith}\ (\mathit{Mult}\ x\ y) = \mathit{liftM2}\ (\mathit{lift2Int2}\ (*))\ x\ y\\
\mathit{phiArith}\ (\mathit{Div}\ x\ y) = \mathit{liftM2}\ (\mathit{lift2Int2}\ \mathit{div})\ x\ y\\
\mathit{phiArith}\ (\mathit{NumEq}\ x\ y) = \mathit{liftM2}\ (\mathit{lift2Int2Bool2}\ (\equiv))\ x\ y\\
\mathit{phiArith}\ (\mathit{Num}\ i) = \mathit{return}\ \$\ I\ i
\end{array}
$$

## 5.5 IF Term Evaluation

$$
\begin{array}{l}
\mathit{phiIf}\ (\mathit{IfTerm}\ b\ t\ f) =\\
\quad \textbf{do}\ \mathit{denv} \leftarrow \mathit{ask}\\
\qquad (B\ b') \leftarrow b\\
\qquad \textbf{if}\ b'\ \textbf{then}\ t\ \textbf{else}\ f
\end{array}
$$

$$
\begin{array}{l}
\mathit{phiIf}\ \mathit{TrueTerm} = \mathit{return}\ (B\ \mathit{True})\\
\mathit{phiIf}\ \mathit{FalseTerm} = \mathit{return}\ (B\ \mathit{False})
\end{array}
$$

## 5.6 Followed By Evaluation

$$
\begin{array}{l}
\mathit{phiCausal}\ (\mathit{FBy}\ \mathit{t1}\ \mathit{t2}) =\\
\quad \textbf{do}\ \mathit{denv} \leftarrow \mathit{ask}\\
\qquad \mathit{v1} \leftarrow \mathit{t1}\\
\qquad \mathit{return}\ \$\ \mathit{v1}\ `\mathit{fbyLV}`\ \mathit{cobind}\ (\mathit{runVSpace}\ \mathit{t2})\ \mathit{denv}
\end{array}
$$

## 5.7 Forming The Semantic Algebra

$$
\begin{array}{l}
\mathit{alg} = (\mathit{mkAlgebra}\ \mathit{phiLam})@+@\\
\qquad (\mathit{mkAlgebra}\ \mathit{phiVar})@+@\\
\qquad (\mathit{mkAlgebra}\ \mathit{phiFix})@+@\\
\qquad (\mathit{mkAlgebra}\ \mathit{phiArith})@+@\\
\qquad (\mathit{mkAlgebra}\ \mathit{phiIf})@+@
\end{array}
$$

$(mkAlgebra\ phiCausal)$

$emptyS = (Env\ [\,]) :< emptyS$

$eval\ tm = runLV\ (runVSpace\ (cata\ alg\ tm))\ emptyS$

$pos :: TermLang$
$pos = makeFixTerm\ (makeLam$ `"pos"` $()\ ((makeNum\ 0)$ `makeFBy` $((makeVarTerm$ `"pos"`$)$ `makeAdd` $(makeNum\ 1$

## References


[1] Tarmo Uustalu and Varmo Vene. The essence of dataflow programming. In K. Yi, editor, *Proceedings of APLAS'05 - Lecture Notes in Computer Science*, volume 3780, pages 2–18. Springer-Verlag, 2005.