A Haskell Companion to “The Essence of Dataflow Programming”

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

January 10, 2006

Before we start, everyone is to repeat to themselves the following mantra:

> A (co)monad is a computation. (co)unit and (co)bind operate on the computation but do not perform it. (co)unit defines the value space for comonad evaluation. (co)bind is in effect a Functor over (co)monads.

The operator used for constructing the *LVS* monad from the *LV* monad and *Stream* monad causes lhs2TeX fits. So, it’s been replaced with :! throughout.

module *EssenceDF* **where**

import *qualified Data.List*

The Comonad Class

The basic structure of a comonad is embodied in the *Comonad* class. Each comonad must have a *counit* and *cobind* function that correspond with *return* and *bind* in the *Monad* class. *counit* accepts a comonad instance and returns the value encapsulated by the comonad. It is, in essence, the opposite of *return* from the *Monad* class. The *cobind* function accepts a function from the comonad type to some value type, a comonad and returns a comonad over the value type.

class *Comonad d* **where**
  $counit :: d\ a \rightarrow a$
  $cobind :: (d\ a \rightarrow b) \rightarrow d\ a \rightarrow d\ b$

$cmap :: Comonad\ d \Rightarrow (a \rightarrow b) \rightarrow d\ a \rightarrow d\ b$
$cmap\ f = cobind\ (f \circ counit)$

The Id Comonad

The *Id* comonad is the simplest comonad. *counit* simply returns the value encapsulated by *Id*. *cobind* pushes the function k inside the comonad *Id*.

```
data Id a = Id a

instance Comonad Id where
    counit (Id a) = a
    cobind k d = Id (k d)
```

The Product Comonad

Life gets more interesting with the product comonad, *Prod*.

Note that e is never evaluated by *cobind* or *counit*. The *counit* observer simply returns the first value. The *cobind* observer creates a new product from the result of calling k over the argument comonad and composing it the the second product value, e. The only thing that makes this interesting is that e is never evaluated and can be infinite.

askP does access the second product element and thus must evaluate it. *localP* pushes a function inside the second product value. However, it is once again not evaluated. It appears that *askP* and *localP* provide a mechanism for accessing the second element and replacing the second element locally with a function over the second element. This would appear to be something like a *Writer* comonad.

```
data Prod e a = a :& e

instance Comonad (Prod e) where
    counit (a :& _) = a
    cobind k d@(_ :& e) = (k d) :& e

askP :: Prod e a → e
askP (_ :& e) = e

localP :: (e → e) → Prod e a → Prod e a
localP g (a :& e) = a :& (g e)
```

The Stream Comonad

The *Stream* comonad is much like *Prod* with the stream tail having more specific properties. Any stream is a current value, a and the rest of the stream az. Like the *Prod* comonand, *counit* only references a and does not evaluated the rest of the stream. *cobind* evaluates the current stream using k like *Prod*. However, the rest of the resulting stream is (*cobind* k az) pushing k down to the remaining elements of the stream.

```
data Stream a = a :< (Stream a) deriving Show

instance Comonad Stream where
    counit (a :< _) = a
    cobind k d@(_ :< az) = (k d) :< (cobind k az)

nextS :: Stream a → Stream a
nextS (a :< az) = az

takeS :: Int → Stream a → [a]
takeS 0 _ = []
```

$takeS\ (i+1)\ (a :< az) = a : takeS\ i\ az$

$str2fun :: Stream\ a \rightarrow Int \rightarrow a$
$str2fun\ (a :< az)\ 0 = a$
$str2fun\ (\_ :< az)\ (i+1) = str2fun\ az\ i$

$fun2str :: (Int \rightarrow a) \rightarrow Stream\ a$
$fun2str\ f = fun2str'\ f\ 0$
$\quad \textbf{where}\ fun2str'\ f\ i = (f\ i) :< (fun2str'\ f\ (i+1))$

The LVS Comonad

The LVS Comonad represents a process that consumes input from a stream, s, produces a current value, v, and records its past in a list, l. Thus, the title *LVS*. The comonad is defined by starting with a custom *List* and composing it with a value to define *LV*. The result is composed with a *Stream* to complete the *LVS* data type.

An *LVS* value is thus define as:

$$l := v :!\ s$$

where := and :! are infix constructors for *LV* and *LVS* respectively.

$\textbf{data}\ LVS\ a = (LV\ a) :!\ (Stream\ a)$

To make *LVS* a comonad, we must define *counit* and *cobind* as required by the *Comonad* typeclass. $counit :: (c\ a) \rightarrow a$ is define as one would expect, extracting the value from the *LVS* structure and returning it.

cobind is a bit more involved, but not substantially more complex. The function defines its value by composing (*cobindL lvs*), (*k lvs*) and (*cobindS lvs*) where *lvs* is the current *LVS* instance and *cobindL* and *cobindS* are local helper functions for generating the new *List* and *Stream* values. So, *cobind* is simply defined as:

$$(cobindL\ d) := (k\ d) :!\ (cobindS\ d)$$

$\textbf{instance}\ Comonad\ LVS\ \textbf{where}$
$\quad counit\ (past := a :!\ future) = a$
$\quad cobind\ k\ d = cobindL\ d := (k\ d) :!\ cobindS\ d$
$\qquad \textbf{where}\ cobindL\ (Nil := a :!\ future) = Nil$
$\qquad\qquad cobindL\ (past' :> a' := a :!\ future) = cobindL\ d' :> k\ d'$
$\qquad\qquad\quad \textbf{where}\ d' = past' := a' :!\ (a :< future)$
$\qquad\qquad cobindS\ (past := a :!\ (a' :< future')) = k\ d' :< cobindS\ d'$
$\qquad\qquad\quad \textbf{where}\ d' = past :> a := a' :!\ future'$

Taking a look at *cobindL* and *cobindS* reveal how the new *LVS* structure is defined. *cobindL* has two cases, one for each *List* constructor. The first says that if *past* is *Nil*, then it remains *Nil* in the new *LVS* structure.

In the case for $past' :> a'$ we're going to create a new past by recursively calling *cobindL*. The value of d' is a one-step unwinding of the previous *past* value. Specifically, $past' := a'$ is the value of *past* and v before a was added. $(a' :< future)$ is the value of *future* before a was processed. What is happening is that the entire *past* list is regenerated each time new *future* element is processed. Furthermore, elements of *past* appear to be the entire comonad representing past states.

cobindS is quite similar with the *future* generated. Here, d' represents the current comonad value with the first element of *future* bound to a' and the current value added to the current *past*. In effect, we are remembering the current value while preparing the next value. *cobindS* is called on d'. $k\ d'$ is added to *cobindS* d' to add the processing of the current *LVT* value to the future stream.

The question is why keep track of all this stuff? *past* seems to be every comonad, not just value, that has been seen. *future* seems to be a stream of every value generated followed by a computation representing future values. This seems rather redundant.

```
runLVS :: (LVS a → b) → Stream a → Stream b
runLVS k (a′ :< as′) = runLVS′ k (Nil := a′ :! as′)
                          where runLVS′ k d@(az := a :! (a′ :< as′)) = (k d) :< (runLVS′ k (az :> a := a′ :! as′))

fbyLVS :: a → (LVS a → a)
fbyLVS a0 (Nil := _ :! _) = a0
fbyLVS _ ((_ :> a′) := _ :! _) = a′

nextLVS :: LVS a → a
nextLVS (_ := _ :! (a :< _)) = a

showLVS :: (Show a) ⇒ LVS a → IO ()
showLVS (Nil := a0 :! future) = (putStr ∘ show) a0 ≫ putStr ", " ≫ showLVS′ future
   where showLVS′ (a′ :< as′) = (putStr ∘ show) a′ ≫ putStr ", " ≫ showLVS′ as′
```

The LV Comonad

The paper presents the LVS comonad first and the LV comonad later. I find it much easier to approach the LVS comonad if you look at LV first, however I understand why they did this and will discuss that in the context of *runLV*. The data structure for the *LV* comonad is a product composed of a list and a value. The contents of the list and the value are the same type. The single constructor is := forms the product. It is simpler for some people to think about the LV comonad as simply a pair rather than using the infix constructor. Your mileage may vary.[1]

```
data List a = Nil | (List a) :> a deriving Show

data LV a = (List a) := a
```

To make the *LV* data type a comonad, we must define *counit* and *cobind*. *counit*, like *unit* for a monad, simply returns the comonad value. As the name implies, it is the dual of *unit*, returning a value from an encapsulated value rather than encapsulating a given value. *cobind* is a mapping or functor over *LV* as it is for any comonad. This is important to remember as we get rolling - *cobind* operates on comonadic computation but does not perform the evaluation. *cobind* is defined by forming the product of *cobind k past* and $k\ d$ where d is the current *LV* value and k is a function we want to push into the comonad.

Given all this, $k\ d$ is the next V and *cobindL k past* is the next history list. By applying k to the current *LV* instance we get a new value of type b from the original value of type $LV\ a$. Remember that the type contained in L must be the same type as V. Thus, whatever we do to $L\ a$ it must result in something of type $L\ b$.

cobindL k past is the next history list. Remember that L must contain items of the same type is V. Thus, whatever we do to L of type $List\ a$, we have to end up with something of type $List\ b$. Looking at the

[1] I moved the code for the LV comonad here from the LVS definition

definition of *Comonad* reminds us that k has type $d\ a \rightarrow b$ where d is the comonad constructor. If L contains things of type $d\ a$ then cobind is just a kind of fold or map with respect to the comonad structure. Specifically, $cobind\ k\ L$ is of type $List\ b$ if L is of type $List\ (d\ a)$. In this case, d is LV. So, the history is a list of LV comonads representing past computations. So far, so good.

$$
\begin{array}{l}
\textbf{instance}\ Comonad\ LV\ \textbf{where} \\
\quad counit\ (\_ := a) = a \\
\quad cobind\ k\ d@(past := \_) = cobindL\ k\ past := (k\ d) \\
\qquad \textbf{where}\ cobindL\ k\ Nil = Nil \\
\qquad\qquad cobindL\ k\ (past :> a) = cobindL\ k\ past :> (k\ (past := a))
\end{array}
$$

cobind is a map, so let's define it like a map with a base case and a recursive case. Looking at the implementation, that's exactly what is done. *cobindL* is defined for both cases of the *List* constructor. $cobindL\ Nil = Nil$ indicating the end of the history list. Basically, *Nil* represents the beginning of execution and the termination of the recursive *combindL* function.

$cobindL\ k\ (past{:>}a)$ defines the case when there is a nonempty past to push the cobind through. $cobindL\ k\ past$ takes care of the recursion and will terminate when *Nil* is encountered. k requires an LV comonad, thus one is formed using $:=$ to combine *past* with a. Notice that *past* is the argument to *cobindL* with the most recent element, a removed. Thus, $past := a$ is the previous comonad and $(k\ (past := a))$ is performing the mapping operation. The recursive call to $cobindL\ k\ past$ steps through each element of the history list, creates a comonad and calls k on it.

runLV performs the computation defined by a comonad. The first argument is a function from $LV\ a$ to b. The k mapped into the comonad by *cobind* will be instantiated with this function when the comonad is evaluated. Think about the cobind as setting up a structure for evaluating k across the entire comonad. The second argument is a stream of inputs inputs. Evaluating the comonad results in a stream of outputs.

$$
\begin{array}{l}
runLV :: (LV\ a \rightarrow b) \rightarrow Stream\ a \rightarrow Stream\ b \\
runLV\ k\ (a' :< as') = runLV'\ k\ (Nil := a' :!\ as') \\
\quad \textbf{where}\ runLV'\ k\ (d@(az := a) :!\ (a' :< as')) = (k\ d) :< runLV'\ k\ (az :> a := a' :!\ as')
\end{array}
$$

The bulk of *runLV* is defined by a helper $runLV'$ that turns the LV comonad into an LVS comonad. I commented earlier that understanding the LV comonad is easier than LVS. The definition of *runLV* is why LVS is presented first. The initial LVS comonad is defined from the LV comonad in the following way:

$$(Nil := a' :!\ as')$$

where a' is the first element of the input stream and as' is the rest of the stream. Thus, the LVS comonad starts with no past, with the first stream element as the value and remaining stream elements as the future. What's interesting here is that the instantiated LVS uses the input stream type as the future type. This correctly suggests that the *runLVS* function will not be used to evaluate the LV comonad. I would suspect that it could, but that is left as an exercise for the reader.

$runLV'$ is remarkably straightforward given where we've already been. k is the function to evaluate, d is the original LV comonad split into az and a by pattern matching. a' is the current head of the input stream and as' is th rest of the stream. $runLV'$ is then a stream where $(k\ d)$ is the head. So, the current LV comonad is evaluated by k giving us something of type b just as we would expect. $runLV'$ is called recursively to generate the rest of the stream. The new past is az with a added. The new value is a', the first element of the current input stream. The new future is as', the rest of the input stream. What this implements is a mapping of k onto the input stream to generate an output stream. Exactly what we want.

The *fbyLV* function is a utility function that is pronounced "followed by" and implements exactly that function. Given a value $a0$ of type a and an LV comonad $(past := v)$, v is followed by $a0$ if *past* is empty. If *past* is not empty, then v is followed by the last element in *past*.

$$
\begin{aligned}
&\mathit{fbyLV} :: a \to (\mathit{LV}\ a \to a)\\
&\mathit{fbyLV}\ \mathit{a0}\ (\mathit{Nil} := \_) = \mathit{a0}\\
&\mathit{fbyLV}\ \_\ ((\_ :> a') := \_) = a'
\end{aligned}
$$

The following are examples of using *LV* to evaluate a function over a stream. *incLV* is a simple function that increments the value from the input stream. It does not depend on the past in any way, so we need only a single case that ignores the past.

$$
\begin{aligned}
&\mathit{incLV} :: \mathit{LV}\ \mathit{Integer} \to \mathit{Integer}\\
&\mathit{incLV}\ (\_ := x) = x + 1
\end{aligned}
$$

sumLV is similar to *incLV* except that it adds the input stream value to the previous input stream value. What is very interesting to note is that the input stream value is save in *past*, not the generated sum value. If we want to save a new state, then it may be necessary to modify the *LV* comonad or the *runLV* function. This example shows one way of looking into the past.

$$
\begin{aligned}
&\mathit{sumLV} :: \mathit{LV}\ \mathit{Integer} \to \mathit{Integer}\\
&\mathit{sumLV}\ (\mathit{Nil} := x) = 0 + x\\
&\mathit{sumLV}\ ((\_ :> y) := x) = y + x
\end{aligned}
$$

andLV operates on two streams of boolean values zipped together. In effect, these are the inputs to an and gate. The output is the result of conjuncting the boolean values. This example shows one way of dealing with multiple inputs.

$$
\begin{aligned}
&\mathit{andLV} :: \mathit{LV}\ (\mathit{Bool}, \mathit{Bool}) \to \mathit{Bool}\\
&\mathit{andLV}\ (\_ := (x, y)) = x \wedge y
\end{aligned}
$$

The most confusing thing about using *runLV* now that we have it is generating streams. Basically, there is no designator for the initial or first stream. With *List*, you start with *Nil* and build up the list. With *Stream* there is no *Nil* resulting in infinite things. *intStream* and *boolStream* are infinite streams of 1 and *True* respectively. Note that test cases for the interpreter below contain similar functions for generating infinite streams of things.

$$
\begin{aligned}
&\mathit{intStream} = 1 :< \mathit{intStream}\\
&\mathit{boolStream} = \mathit{True} :< \mathit{boolStream}
\end{aligned}
$$

The infinite streams can be input into the *runLV* function directly, but it's more interesting to put values in front of the infinite tail. Some examples of this include adding 1 to the stream $1, 2, 3, 0, 1, 1, 1, ...$:

$$\mathit{runLV}\ \mathit{incLV}\ (1 :< (2 :< (3 :< (4 :< (0 :< \mathit{intStream})))))$$

adding pairs of values $0 + 1, 1 + 2, 2 + 3, 3 + 4, 4 + 0, 0 + 1, 1 + 1$,...:

$$\mathit{runLV}\ \mathit{sumLV}\ (1 :< (2 :< (3 :< (4 :< (0 :< \mathit{intStream})))))$$

and running an "and" gate through it's possible inputs. You might want to evaluate the *zipS* separately to see the zip operation on streams. (I really think that *Stream* should be an instance of *ComonadZip* rather than use a separate function):

$$
\begin{aligned}
&\mathit{runLV}\ \mathit{andLV}\ (\mathit{zipS}\ (\mathit{True} :< (\mathit{False} :< (\mathit{True} :< (\mathit{False} :< \mathit{boolStream}))))\\
&\qquad\qquad\qquad\qquad (\mathit{True} :< (\mathit{True} :< (\mathit{False} :< (\mathit{False} :< \mathit{boolStream})))))
\end{aligned}
$$

One interesting observation is that *runLV2* operating on *sumLV* does not keep a running sum of the values in its past list. Initially, this is what I had hoped it would do. However, it's an easy modification to *runLV* to store the system state rather than the previous input. *runLV2* does exactly this. The **let** variable *kd* references $(k\ d)$ and is added to the past list. Thus, then the next input is processed, it is added to the past value rather than the past input.

$$
\begin{aligned}
&\mathit{runLV2} :: (\mathit{LV}\ a \rightarrow a) \rightarrow \mathit{Stream}\ a \rightarrow \mathit{Stream}\ a\\
&\mathit{runLV2}\ k\ (a' :< as') = \mathit{runLV}'\ k\ (\mathit{Nil} := a' :!\ as')\\
&\quad \mathbf{where}\ \mathit{runLV}'\ k\ (d@(az := a) :!\ (a' :< as')) =\\
&\qquad\qquad \mathbf{let}\ kd = (k\ d)\ \mathbf{in}\\
&\qquad\qquad kd :< \mathit{runLV}'\ k\ (az :> kd := a' :!\ as')
\end{aligned}
$$

You can run *sumLV* using *runLV2* to see the difference in the output.

$$
\mathit{runLV2}\ \mathit{sumLV}\ (1 :< (2 :< (3 :< (4 :< (0 :< \mathit{intStream})))))
$$

Note that the paper defines a proper sum function that uses *sumLV* that we will examine later.

The biggest problem here is that none of these examples halt. The input stream is infinite in each case, thus no termination. The utility function *takeS* solves this problem by defining a *take* function over streams. *takeS* i s takes the first i elements from stream s and returns them as a list. Thus:

$$
\mathit{take}\ 5\ (\mathit{runLV}\ \mathit{incLV}\ (1 :< (2 :< (3 :< (4 :< (0 :< \mathit{intStream}))))))
$$

will generate a list:

$$
[2, 3, 4, 5, 1]
$$

This should make testing quite a bit simpler.

Comonadic Zip

The *ComonadZip* class defines a function over a comonad that zips two comonads into a single comonand. The two comonads must be instances of the same class, but may encapsulate things of different types. The resulting type is an instance of the comonad class defined over the product of the originally encapsulated types. The function signature pretty much says it all.

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Comonad}\ d \Rightarrow \mathit{ComonadZip}\ d\ \mathbf{where}\\
&\quad \mathit{czip} :: d\ a \rightarrow d\ b \rightarrow d\ (a, b)
\end{aligned}
$$

Examples of *ComonadZip* are included for *Id*, *LVS* and *LV*. *Id* is quite simple with *czip* extracting comonad values, forming a product, and injecting the product back into *Id*.

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{ComonadZip}\ \mathit{Id}\ \mathbf{where}\\
&\quad \mathit{czip}\ (\mathit{Id}\ a)\ (\mathit{Id}\ b) = \mathit{Id}\ (a, b)
\end{aligned}
$$

czip for *LVS* and *LV* is not much more complicated. *zipL* takes care of zipping two lists together. Defined recursively, *zipL* forms the product of the first two *List* elements and adds that to zipping the remainder of the lists together. Note that zip ends when *either* list is empty.

$$
\begin{aligned}
&zipL :: List\ a \rightarrow List\ b \rightarrow List\ (a, b)\\
&zipL\ Nil\ \_ = Nil\\
&zipL\ \_\ Nil = Nil\\
&zipL\ (az :> a)\ (bz :> b) = (zipL\ az\ bz) :> (a, b)
\end{aligned}
$$

zipS for *Stream* is analogous to *zipL*. The product of the current stream values is added to zipping the remainder of the streams together.

$$
\begin{aligned}
&zipS :: Stream\ a \rightarrow Stream\ b \rightarrow Stream\ (a, b)\\
&zipS\ (a :< az)\ (b :< bz) = (a, b) :< (zipS\ az\ bz)
\end{aligned}
$$

With *zipS* and *zipL* defined, *czip* for *LVS* is quite simple. *zipL* combines the past lists, *zipS* combines the future streams, and product combines the current values.

$$
\begin{aligned}
&\mathbf{instance}\ ComonadZip\ LVS\ \mathbf{where}\\
&\quad czip\ (past := a :!\ future)\ (past' := b :!\ future') = zipL\ past\ past' := (a, b) :!\ zipS\ future\ future'
\end{aligned}
$$

czip for *LV* the same as *LVS* except that there is no future to zip together and add to the result. Same utility functions used in the same way.

$$
\begin{aligned}
&\mathbf{instance}\ ComonadZip\ LV\ \mathbf{where}\\
&\quad czip\ (past := a)\ (past' := b) = zipL\ past\ past' := (a, b)
\end{aligned}
$$

A Comonadic Interpreter

Among the most useful applications of monads is interpreter construction. Similarly, it appears, for comonads.

The data structure for the language AST remains unchanged from most monadic language ASTs. Note that this is not a composable AST due to its recursive nature. First define structures for th term space and the value space. In a monadic interpreter, the language elements are instances of *Monad* while values are basic Haskell types. Note here that terms are a constructed type while values are comonads.

$$
\begin{aligned}
&\mathbf{type}\ Var = String\\
&\\
&\mathbf{data}\ Tm = V\ Var \mid L\ Var\ Tm \mid Tm :@\ Tm \mid Rec\ Tm\\
&\qquad\quad \mid N\ Integer \mid Tm :+ Tm \mid Tm :- Tm \mid Tm :* Tm \mid Mod\ Tm\ Tm\\
&\qquad\quad \mid Tm :== Tm \mid Tm :/= Tm \mid TT \mid FF \mid Not\ Tm \mid Tm :\&\&\ Tm\\
&\qquad\quad \mid If\ Tm\ Tm\ Tm\\
&\qquad\quad \mid Next\ Tm\\
&\qquad\quad \mid Fby\ Tm\ Tm
\end{aligned}
$$

The value space consists of integers, booleans and functions. *Val* is parameterized over a comonad, *d* making the resulting values comonadic. Of particular interest is the definition of *F*, a comonadic function. Specifically, a *F* encapsulates a mapping from a comonadic value *d* (*Val d*) to a value *Val d*. Note that *Val d* is not comonadic but must know about the comonad because can be a function value.

$$\textbf{data}\ \mathit{Val}\ d = I\ \mathit{Integer} \mid B\ \mathit{Bool} \mid F\ (d\ (\mathit{Val}\ d) \rightarrow \mathit{Val}\ d)$$

$$\begin{array}{l} \textbf{instance}\ \mathit{Show}\ (\mathit{Val}\ d)\ \textbf{where} \\ \quad \mathit{show}\ (I\ i) = \mathit{show}\ i \\ \quad \mathit{show}\ (B\ b) = \mathit{show}\ b \\ \quad \mathit{show}\ (F\ f) = \texttt{"Func"} \end{array}$$

An environment is simply a list of *Var*, *Val* pairs where *Var* is a string type and *Val d* is simply a value as defined above. Neither *Env* or *Val d* are comonadic, but again must know about the comonad to define function values.

$$\textbf{type}\ \mathit{Env}\ d = [(\mathit{Var}, \mathit{Val}\ d)]$$

A an instance *ComonadEv* must have an evaluation function, *ev*, that accepts a term, a comonadic environment, and generates a value. *Env d* is not a comonad, but *d* (*Env d*) definitely is. Thus, it minimally has a cobind for applying operations and a counit for returning the environment value.

$$\begin{array}{l} \textbf{class}\ \mathit{ComonadZip}\ d \Rightarrow \mathit{ComonadEv}\ d\ \textbf{where} \\ \quad \mathit{ev} :: \mathit{Tm} \rightarrow d\ (\mathit{Env}\ d) \rightarrow \mathit{Val}\ d \end{array}$$

The ev' function is a helper for defining interpreters. It is, in essence, an interpeter core that implements basic λ-caculus style functions for operations, λ definition and application, and variable usage. The signature defines ev' as a function from a term and comonadic environment to a value. This should be expected. The comonad, *d*, must be an instance of *ComonadEv* having an *ev* function defined for function evaluation.

$$\mathit{ev}' :: \mathit{ComonadEv}\ d \Rightarrow \mathit{Tm} \rightarrow d\ (\mathit{Env}\ d) \rightarrow \mathit{Val}\ d$$

V is the variable constructor and is evaluted by looking it's value up in the environment. *counit* extracts the environment from the current environment, *denv*. *unsafeLookup* then calls the standard *lookup* function with the variable name and the environment's value.

$$\mathit{ev}'\ (V\ x)\ \mathit{denv} = \mathit{unsafeLookup}\ x\ (\mathit{counit}\ \mathit{denv})$$

L defines a lambda that when evaluated results in a function value. This definition is remarkably like the definition used in monadic interpreters in that a Haskell fuction is used to represent the evaluation. The expression, *e*, is evaluated in the context of the comonadic environment *denv* with the comonadic value *d* added to it.

d is instantiated when the function value is used, thus it represents the comonadic value the function is applied to. The Haskell function representing the fucntion value calls *ev e* to evaluate the expression encapsulated by *L* in an environment with *d* associated with the variable name *x*. The trick is generating the environment.

d is first zipped together with *denv* resulting in a comonad of value, environment pairs. *repair* ("re-pair" not fix) takes an value, environment pair and creates an evironment buy: (i) pairing *x* with the value; and (ii) consing the pair onto the environment. Using *cmap* to apply *repair* to each value, environment pair effectively adds the new binding.

$$\begin{array}{l} \mathit{ev}'\ (L\ x\ e)\ \mathit{denv} = F\ (\lambda d \rightarrow \mathit{ev}\ e\ (\mathit{cmap}\ \mathit{repair}\ (\mathit{czip}\ d\ \mathit{denv}))) \\ \quad \textbf{where}\ \mathit{repair}\ (a, \mathit{env}) = (x, a) : \mathit{env} \end{array}$$

The $e :@ e'$ function application syntax uses cobind to apply the evaluation of e' to then environment and calles the function value resulting from evaluating e to the result. The case expression evaluates e in *denv* and attempts to extract a function value from the result. *cobind* is then used to apply $(ev\ e')$ to the original environment. Finally, calling f evaluates the encapsulated function expression in the context of the new environment.

$$
\begin{array}{l}
ev'\ (e :@ e')\ denv = \textbf{case}\ ev\ e\ denv\ \textbf{of} \\
\qquad\qquad\qquad F\ f \rightarrow f\ (cobind\ (ev\ e')\ denv)
\end{array}
$$

The *Rec* constructor provides a mechanism for evaluating recursive operations. Specifically, it uses the definition of fixed-point to perform one-step expansion of a recursive function application. Note that cobind is called on ev' of *Rec e* giving us the desired recursive effect.

$$
\begin{array}{l}
ev'\ (Rec\ e)\ denv = \textbf{case}\ ev\ e\ denv\ \textbf{of} \\
\qquad\qquad\qquad F\ f \rightarrow f\ (cobind\ (ev'\ (Rec\ e))\ denv)
\end{array}
$$

The following cases implement operations over integers and booleans. In each case, arguments to the operator are evaluated in the current environment. If they are the correct type, then an operation from the host language is used to calculate a new value that is then returned.

$$
\begin{array}{l}
ev'\ (N\ n)\ denv = I\ n \\
ev'\ (e0 :+ e1)\ denv = \textbf{case}\ ev\ e0\ denv\ \textbf{of} \\
\qquad\qquad I\ n0 \rightarrow \textbf{case}\ ev\ e1\ denv\ \textbf{of} \\
\qquad\qquad\qquad I\ n1 \rightarrow I\ (n0 + n1) \\
ev'\ (e0 :- e1)\ denv = \textbf{case}\ ev\ e0\ denv\ \textbf{of} \\
\qquad\qquad I\ n0 \rightarrow \textbf{case}\ ev\ e1\ denv\ \textbf{of} \\
\qquad\qquad\qquad I\ n1 \rightarrow I\ (n0 - n1) \\
ev'\ (e0 :* e1)\ denv = \textbf{case}\ ev\ e0\ denv\ \textbf{of} \\
\qquad\qquad I\ n0 \rightarrow \textbf{case}\ ev\ e1\ denv\ \textbf{of} \\
\qquad\qquad\qquad I\ n1 \rightarrow I\ (n0 * n1) \\
ev'\ (Mod\ e0\ e1)\ denv = \textbf{case}\ ev\ e0\ denv\ \textbf{of} \\
\qquad\qquad I\ n0 \rightarrow \textbf{case}\ ev\ e1\ denv\ \textbf{of} \\
\qquad\qquad\qquad I\ n1 \rightarrow I\ (n0\ `mod`\ n1) \\
ev'\ (e0 :== e1)\ denv = \textbf{case}\ ev\ e0\ denv\ \textbf{of} \\
\qquad\qquad I\ n0 \rightarrow \textbf{case}\ ev\ e1\ denv\ \textbf{of} \\
\qquad\qquad\qquad I\ n1 \rightarrow B\ (n0 \equiv n1) \\
\qquad\qquad B\ b0 \rightarrow \textbf{case}\ ev\ e1\ denv\ \textbf{of} \\
\qquad\qquad\qquad B\ b1 \rightarrow B\ (b0 \equiv b1) \\
ev'\ (e0 :/= e1)\ denv = \textbf{case}\ ev\ e0\ denv\ \textbf{of} \\
\qquad\qquad I\ n0 \rightarrow \textbf{case}\ ev\ e1\ denv\ \textbf{of} \\
\qquad\qquad\qquad I\ n1 \rightarrow B\ (n0 \not\equiv n1) \\
\qquad\qquad B\ b0 \rightarrow \textbf{case}\ ev\ e1\ denv\ \textbf{of} \\
\qquad\qquad\qquad B\ b1 \rightarrow B\ (b0 \not\equiv b1) \\
ev'\ TT\ denv = B\ True \\
ev'\ FF\ denv = B\ False \\
ev'\ (Not\ e)\ denv = \textbf{case}\ ev\ e\ denv\ \textbf{of} \\
\qquad\qquad B\ b \rightarrow B\ (\neg\ b) \\
ev'\ (e0 :\&\&\ e1)\ denv = \textbf{case}\ ev\ e0\ denv\ \textbf{of} \\
\qquad\qquad B\ b0 \rightarrow \textbf{case}\ ev\ e1\ denv\ \textbf{of} \\
\qquad\qquad\qquad B\ b1 \rightarrow B\ (b0 \wedge b1)
\end{array}
$$

If is defined in the classic, functional fashion as an expression rather than a statement.

$$\begin{aligned} ev'\ (If\ e\ e0\ e1)\ denv = & \ \mathbf{case}\ ev\ e\ denv\ \mathbf{of} \\ & \quad B\ b \rightarrow \mathbf{if}\ b\ \mathbf{then}\ ev\ e0\ denv\ \mathbf{else}\ ev\ e1\ denv \end{aligned}$$

With ev' fully defined, the authors define *Id*, *LVS* and *LV* to be instances of *ComonadEv* by defining *ev* for each structure. Basically, *ev* is ev' with additions sequencing and stream.

Evaluating an expression using *Id* comonad is simply the application of ev' to the expression. The instance definition could as easily be $ev = ev'$ as the definition provided. This provides one shot evaluation that cannot involve *Fby* or *Next* as they are not defined for the *ev* instance for *Id*.

$$\begin{aligned} & \mathbf{instance}\ ComonadEv\ Id\ \mathbf{where} \\ & \quad ev\ e\ denv = ev'\ e\ denv \end{aligned}$$

The *LV* interpreter adds *Fby* to the language interpreted by ev' allowing sequencing of calculations. Evaluating anything but an *Fby* expression is simply calling ev'. Evaluating *e0* ‘*Fby*‘ *e1* uses *fbyLV* to provide semantics for *Fby*.

$$\begin{aligned} & \mathbf{instance}\ ComonadEv\ LV\ \mathbf{where} \\ & \quad ev\ (e0\ `Fby`\ e1)\ denv = ev\ e0\ denv\ `fbyLV`\ cobind\ (ev\ e1)\ denv \\ & \quad ev\ e\ denv = ev'\ e\ denv \end{aligned}$$

The *LV* interpreter adds *Fby* and *Next* to the language interpreted by ev' allowing sequencing of calculations. Evaluating anything but an *Fby* or *Next* expression is simply calling ev'. Evaluating *e0* ‘*Fby*‘ *e1* uses *fbyLVS* to provide semantics for *Fby* in the same manner as *LV*. Evaluating *Next e* uses *nextLVS* to provide semantics for *Next*.

$$\begin{aligned} & \mathbf{instance}\ ComonadEv\ LVS\ \mathbf{where} \\ & \quad ev\ (e0\ `Fby`\ e1)\ denv = ev\ e0\ denv\ `fbyLVS`\ cobind\ (ev\ e1)\ denv \\ & \quad ev\ (Next\ e)\ denv = nextLVS\ (cobind\ (ev\ e)\ denv) \\ & \quad ev\ e\ denv = ev'\ e\ denv \end{aligned}$$

We have three interptseters that evaluate a base language. The *Id* monad interpreter handles basic expression evaluation, but cannot deal with the followed by or next operations. The *LV* monad interpreter adds *Fby* to the language while *LVS* adds followed by and next. However, the core language remains the same. Thus, we can drop any interpreter we want into this comonadic structure. This has implications on what compiling to a comonadic structure might result in.

Following are a vew utility functions that will prove handy when we start evaluating language elements. *emptyL i* produces a list of *i* empty environments. Similarly, *emptyS* produces a stream of infinitely many empty environments.

$$\begin{aligned} & emptyL :: Int \rightarrow List\ [(a, b)] \\ & emptyL\ 0 = Nil \\ & emptyL\ (i + 1) = emptyL\ i :> [\,] \\ \\ & emptyS :: Stream\ [(a, b)] \\ & emptyS = [\,] :< emptyS \end{aligned}$$

The *evMainI*, *evMainLVS* and *evMainLV* functions are signatures for evaluators written for *Id*, *LVS*, and *LV* comonads respectively. In each case, the value (*counit*) of the comonad is the current environment. Thus, the past and future are a *List* and *Stream* of environments respectively. When *evMainLVS* is called on some expression e and count i, *ev* is initialized with a list of i empty evironments, a current empty environment, and an infinite stream of empty environments. Both *evMainI* and *evMainLV* simply skip the initialization of elements they do not use.

In each case, the *evMain* functions will use the comonadic evaluator to find the i_{th} environment in the past environment sequence. (For *Id*, $i = 0$ because there is no past.) So, they generate i empty environments and an empty current environment.

$$\begin{aligned}
&\mathit{evMainI} :: \mathit{Tm} \to \mathit{Val}\ \mathit{Id} \\
&\mathit{evMainI}\ e = \mathit{ev}\ e\ (\mathit{Id}\ [\,])
\end{aligned}$$

$$\begin{aligned}
&\mathit{evMainLVS} :: \mathit{Tm} \to \mathit{Int} \to \mathit{Val}\ \mathit{LVS} \\
&\mathit{evMainLVS}\ e\ i = \mathit{ev}\ e\ (\mathit{emptyL}\ i := [\,]\ :!\ \mathit{emptyS})
\end{aligned}$$

$$\begin{aligned}
&\mathit{evMainLV} :: \mathit{Tm} \to \mathit{Int} \to \mathit{Val}\ \mathit{LV} \\
&\mathit{evMainLV}\ e\ i = \mathit{ev}\ e\ (\mathit{emptyL}\ i := [\,])
\end{aligned}$$

Alternatively, we can use *ev* and run functions to generate and return lists of history elements. In the following two examples, *ev* is used to generate a stream transformer from a term. *ev* is not provided with the input comonad, the input stream must provide it. *runLV* and *runLVS* apply that stream transformer to an empty stream of environments. *takeS* then takes the first several, in this case 5, elements of the output stream. If we wanted to process input from the stream, data can be passed to the transformer by using a stream other than *emptyS*.

$$\begin{aligned}
&\mathit{takeS}\ 5\ (\mathit{runLVS}\ (\mathit{ev}\ (\mathit{sum} :@\ (N\ 1)))\ \mathit{emptyS}) \\
&\mathit{takeS}\ 5\ (\mathit{runLV}\ (\mathit{ev}\ (\mathit{sum} :@\ (N\ 1)))\ \mathit{emptyS})
\end{aligned}$$

$$\begin{aligned}
&\mathit{runLV}\ (\mathit{ev}\ (\mathit{EssenceDF.sum} :@\ (N\ 1)))\ \mathit{emptyS} \\
&\mathit{takeS}\ 5\ (\mathit{runLV}\ (\mathit{ev}\ (\mathit{EssenceDF.sum} :@\ (N\ 1)))\ \mathit{emptyS})
\end{aligned}$$

Let's run through a collection of basic examples. First, let's look at a variable-free term that adds two numerical values together:

$$\mathit{constSum} = ((N\ 1) :+\ (N\ 2))$$

Running *evMainI constSum*, *evMainLV constSum* k and *evMainLVS constSum* k all result in 3 for any value of k. The reason is the expression does not depend on the input stream. Thus, no matter what the stream position is, the expression is always 3.

Now define a simple increment function:

$$\mathit{inc} = (L\ \texttt{"x"}\ ((V\ \texttt{"x"}) :+\ (N\ 1)))$$

and evaluate its application to 1:

$$\mathit{evMainI}\ (\mathit{inc} :@\ (N\ 1))$$

Again, the value does not depend on an input stream and is constant in any state. Therefore, evaluation with respect to any of the comonads will result in the same value in any context.

Now let's look at a simple application of *Fby* to see what it does. *Fby* is not defined for the *Id* comonad. It's *ev* function does not define a case for it. As there is no past or future in *Id*, the concept of 'followed by' has no meaning.

Fby is defined for both *LV* and *LVS* and means roughly the same thing in both. x '*Fby*' y takes the value x and is y thereafter. In some senses it is a unit delay or the insertion of a value. To understand what is going on, consider the following evaluation:

$$takeS\ 5\ (runLV\ (ev\ ((N\ 0\,`Fby`\ N\ 1)))\ emptyS)$$

The result of *runLV* is a stream whose first value is 0 and 1 thereafter. Nesting *Fby* instances allows creating streams of arbitrary value orderings. Next, let's nest the *Fby* in an expression:

$$takeS\ 5\ (runLV\ (ev\ ((N\ 1) :- (N\ 0\,`Fby`\ N\ 1)))\ emptyS)$$

The value of this expression is 1 minus the crrent value of $(N\ 0)\,`Fby`\,(N\ 1)$. We saw before that the followed by expression takes the value of 0 followed by 1 thereafter. Thus, this expression evaluates to a stream of 1 followed by 0 thereafter.

diff defines a function that uses the *Fby* operator over a variable. Specifically, the value 1 used in the previous example is abstracted out and replaced with the variable `"x"`. When the function is applied to a value, `"x"` is replaced with the value in all future environments. Thus, the value of *diff* $(N\ 1)$ is $1-0$ followed by $1-1$ or 1 followed by 0 thereafter.

```
    -- diff x = x - (0 `fby` x)
diff = L "x" (V "x" :- (N 0 `Fby` V "x"))
```

Evaluating *diff* on a particular number argument will result in a sequence whose first value is the input parameter and whose subsequent values are all 0. `"x"` is replace when the function is evaluated, $x - x$ will be 0 after the first sequence input.

The *pos* function is a simple function that returns the current position in the past list. For the first time, we use the *Rec* constructor to define a recursive function.

```
    -- pos = 0 `fby` (pos + 1)
pos = Rec (L "pos" (N 0 `Fby` (V "pos" :+ N 1)))

    -- sum x = sumx
    -- where sumx = x + (0 `fby` sumx)
    -- Adds a value to a running sum
sum = L "x" (Rec (L "sumx" (V "x" :+ (N 0 `Fby` V "sumx"))))

    -- ini x = inix
    -- where inix = x `fby` inix
    -- Generates a stream of initial values
ini = L "x" (Rec (L "inix" (V "x" `Fby` V "inix")))

    -- fact = 1 `fby` (fact * (pos + 1))
    -- Generates a stream of factorial values. Choosing the nth one
```

```
    -- gets you n!
fact = Rec (L "fact" (N 1 `Fby` (V "fact" :* (pos :+ N 1))))

    -- fibo = 0 `fby` (fibo + (1 `fby` fibo))
    -- Generates a stream of fibonacci values. Choosing the nth one
    -- gets you fib(n)
fibo = Rec (L "fibo" (N 0 `Fby` (V "fibo" :+ (N 1 `Fby` V "fibo"))))

wvr = Rec (L "wvr" (L "x" (L "y" (
                                       If (ini :@ (V "y"))
                                          (V "x" `Fby` (V "wvr" :@ (Next (V "x")) :@ (Next (V "y"))))
                                          ((V "wvr" :@ (Next (V "x"))) :@ (Next (V "y")))
                                   ))))

sieve = Rec (L "sieve" (L "x" (
                                   (V "x" `Fby` (
                                               V "sieve" :@ ((wvr :@ V "x") :@ (
                                                                                  V "x" `Mod` (ini :@ (V "x
                                                                                    ))
                                                 ))
                                   )))

eratosthenes = sieve :@ (pos :+ N 2)
```

The following code block is spec'ed and not loaded.

```
regsel le clr a = if le ∧ clr
                  then − 1
                  else if le ∧| clr
                       then a
                       else if | le ∧ clr
                            then 0
                            else − 2

reg le clr a = (regsel le clr a) `Fby` (reg le clr a)

regsel = L "le" (L "clr" (L "a" (L "last"
                                  (If ((V "le") :&& (V "clr"))
                                     (N (−1))
                                     (If ((V "le") :&& (Not (V "clr")))
                                        (V "a")
                                        (If ((Not (V "le")) :&& (V "clr"))
                                               (N 0)
                                               (V "last")
                                               )
                                     )
                                   )
                                 )))

reg = Rec (L "reg" (L "le" (L "clr" (L "a"
                                             ((((regsel :@ V "le") :@ V "clr") :@ V "a") :@ ((N (−2))
```

```
                                                                                      `Fby`
                                                                                      ((((V "reg") :
                                                                                      )
                                                  )
                                          ))))

reg = L "le" (L "clr" (L "a" (Rec (L "loop"
                                                (If ((V "le") :&& (V "clr"))
                                                      (N (-1))
                                                      (If ((V "le") :&& (Not (V "clr")))
                                                            (V "a")
                                                            (If ((Not (V "le")) :&& (V "clr"))
                                                                  (N 0)
                                                                  ((N (-2)) `Fby` (V "loop"))
                                                            )
                                                      )
                                                )
                                          ))))


    -- helps generate testing streams
bitStream' _ _ [] final = final
bitStream' now current scheds@((time, val) : scheds') final
    | now ≡ time = val `Fby` (bitStream' (now + 1) val scheds' final)
    | otherwise = current `Fby` (bitStream' (now + 1) current scheds final)

bitStream start scheds final = bitStream' 0 start scheds final


leTest1 = FF
clrTest1 = FF

leTest2 = FF
clrTest2 = bitStream FF [(1, TT)] FF

leTest3 = bitStream FF [(1, TT)] FF
clrTest3 = bitStream FF [(1, TT)] FF

leTest4 = bitStream FF [(3, TT)] FF
clrTest4 = bitStream FF [(1, TT)] FF

leTest5 = bitStream FF [(3, TT), (5, FF)] FF
clrTest5 = bitStream FF [(1, TT)] FF

leTest6 = bitStream FF [(3, TT), (4, FF), (5, FF)] FF
clrTest6 = bitStream FF [(1, TT)] FF

leTest7 = bitStream FF [(3, TT), (4, FF), (5, TT)] FF
clrTest7 = bitStream FF [(1, TT), (2, FF), (7, TT)] FF

unsafeLookup k al = unJust $ Data.List.lookup k al
    where unJust (Just a) = a
```