Journal of Universal Computer Science, vol. 11, no. 7 (2005), 1310-1326
submitted: 28/4/05, accepted: 9/6/05, appeared: 28/7/05 © J.UCS

Signals and Comonads

Tarmo Uustalu
(Institute of Cybernetics at Tallinn University of Technology, Estonia
tarmo@cs.ioc.ee)

Varmo Vene
(Dept. of Computer Science, University of Tartu, Estonia
varmo@cs.ut.ee)

Abstract: We propose a novel discipline for programming stream functions and for the semantic description of stream manipulation languages based on the observation that both general and causal stream functions can be characterized as coKleisli arrows of comonads. This seems to be a promising application for the old, but very little exploited idea that if monads abstract notions of computation of a value, comonads ought to be useable as an abstraction of notions of value in a context. We also show that causal partial-stream functions can be described in terms of a combination of a comonad and a monad.
Key Words: comonads, distributive laws, stream functions, dataflow computation
Category: D.3.1, F.3.2

1 Introduction

Following the seminal work of Moggi [1991] and Wadler [1992], it has become standard in functional programming and semantics to analyze various notions of computation of a value as *monads*. Recently, however, there has also been discussion about the need for mathematical abstractions more permissive than monads to get a better uniform grip on the diverse notions of impure function one can encounter in programming. One particular variety of such notions of function are the various classes of signal or flow functions that are central in functional reactive programming, intensional languages and synchronous dataflow languages. In this connection, Hughes [2000] has promoted *arrows* as an abstraction especially handy for programming with signals or flows. His idea has been taken up in functional reactive programming [Nilsson et al. 2002] and there exists by now not only an arrow library in Haskell but even specialized syntax [Paterson 2001]. The same concept has also emerged in semantics in the work by Power and Robinson [1997] under the name of Freyd categories.

There exists however also a considerably more standard and elementary categorical concept of *comonad* which should be useable to capture notions of value in a context. This fact has received some attention [Brookes and Geva 1992, Kieburtz 1999, Lewis et al. 2000], but the examples put forward have typically not been very interesting (e.g., the product comonad does not really give one more than the exponent monad) and in general comonads have not found extensive use in semantics or programming. We claim that this is unfair and argue that non-trivial comonads are quite relevant for signal or flow based programming and for the semantics of corresponding specialized programming languages. In this paper, which reports on the progress of an ongoing project, we study streams as signals in discrete time and show that both general stream functions and those that are causal in the sense that the present of the output signal can only depend on the past and the present of the input signal are representable via comonads. (Importantly, it is NOT the well-known streams comonad that can be used here.) In addition, we also study partial streams (streams with undefined elements) as a model of clocked signals and demonstrate that causal partial-stream functions are representable via a distributive law of a comonad over a monad.

To our best knowledge, our comonadic approach to stream function programming is new, which is surprising given how elementary it is. Workers in functional reactive programming and synchronous dataflow languages have produced a number of papers exploiting the final coalgebraic structure of streams, e.g., [Caspi and Pouzet 1998, Kieburtz 2000, Barbier 2002], and a number which analyze various classes of stream functions as arrows, e.g., [Hughes 2000, Paterson 2001, Paterson 2003, Nilsson et al. 2002], but apparently nothing on stream functions and comonads. The same is true about works in universal coalgebra [Rutten 2000, Rutten 2003]. Thus we hope that it will prove to be a useful tool in both programming and semantics and especially in the study of semantics of intensional languages like Lucid [Aschroft and Wadge 1985] and synchronous dataflow languages like Lustre and Lucid Synchrone [Halbwachs 1991, Caspi and Pouzet 1996, Pouzet 2001].

The paper is organized as follows. In Section 2, we recapitulate monads and their use as an abstraction of notions of computation with an effect as well as Hughes's arrow type constructors, also known as Freyd categories. In Section 3, we introduce comonads. Comonads for representing general and causal stream functions are the topic of Section 4. In Section 5, we show that causal partial-stream functions are elegantly described via a combination of a comonad and a monad. We conclude in Section 6. To keep the presentation as readable as possible for a programmer, we give most definitions only in the functional language Haskell, avoiding categorical notation.

Throughout the paper we will tacitly assume that pure functions are interpreted in a Cartesian closed base category such as **Set** or some model of parametric polymorphism. In the contexts where some coproducts, initial algebras or final coalgebras are explicitly named, these are assumed to exist too.

2 Monads and arrows

We begin with a brief recapitulation of the monad-based approach to representing effectful functions [Moggi 1991, Wadler 1992, Benton et al. 2002].

A *monad* (in extension form) on a category $\mathcal{C}$ is given by a mapping $T : |\mathcal{C}| \to |\mathcal{C}|$ together with a $|\mathcal{C}|$-indexed family η of maps $\eta_A : A \to TA$ of $\mathcal{C}$ (*unit*), and an operation $-^\star$ taking every map $k : A \to TB$ in $\mathcal{C}$ to a map $k^\star : TA \to TB$ of $\mathcal{C}$ (*extension operation*) such that

1. for any $k : A \to TB$, $k^\star \circ \eta_A = k$,
2. $\eta_A{}^\star = \mathsf{id}_{TA}$,
3. for any $k : A \to TB$, $\ell : B \to TC$, $(\ell^\star \circ k)^\star = \ell^\star \circ k^\star$.

Monads are a construction with many remarkable properties, but the central one for programming and semantics is that any monad $(T, \eta, -^\star)$ defines a category $\mathcal{C}_T$ where $|\mathcal{C}_T| = |\mathcal{C}|$ and $\mathcal{C}_T(A, B) = \mathcal{C}(A, TB)$, $(\mathsf{id}_T)_A = \eta_A$, $\ell \circ_T k = \ell^\star \circ k$ (*Kleisli category*) and an identity on objects functor $J : \mathcal{C} \to \mathcal{C}_T$ where $Jf = \eta_B \circ f$ for $f : A \to B$.

In the monadic approach to effectful functions, the underlying object mapping T of a monad is seen as an abstraction of the kind of effect considered and assigns to any type A a corresponding type TA of "computations of values" or "values with an effect". An effectful function from A to B is identified with a map $A \to B$ in the Kleisli category, i.e., a map $A \to TB$ in the base category. The unit of the monad makes it possible to view any pure function as an effectful one while the extension operation provides composition of effect-producing functions. Of course monads capture the structure that is common to all notions of effectful function. Operations specific to a particular type of effect are not part of the corresponding monad structure.

There are many standard examples of monads in semantics. Here is a brief list of examples. In each case, the object mapping T is a monad.

- $TA = \mathsf{Maybe}\, A + 1$, error (undefinedness), $TA = A + E$, exceptions,
- $TA = E \Rightarrow A$, readable environment,
- $TA = \mathsf{List}\, A = \mu X.1 + A \times X$, non-determinism,
- $TA = S \Rightarrow A \times S$, state,
- $TA = (A \Rightarrow R) \Rightarrow R$, continuations,
- $TA = \mu X.A + (U \Rightarrow X)$, interactive input,
- $TA = \mu X.A + V \times X \cong A \times \mathsf{List} V$, interactive output,

- $TA = \mu X.A + FX$, the free monad over F,
- $TA = \nu X.A + FX$, the free completely iterative monad over F [Aczel et al. 2003].

(By μ and ν we denote the least and greatest fixpoints of functors.)

In Haskell, monads are implemented as a type constructor class with two member functions (in the Prelude):

```
class Monad t where
  return :: a -> t a
  (>>=)  :: t a -> (a -> t b) -> t b

mmap :: Monad t => (a -> b) -> t a -> t b
mmap f c = c >>= (return . f)
```

Here, `return` is the Haskell name for the unit and `>>=` (pronounced 'bind') is the extension operation of the monad. Haskell also supports a special syntax for defining Kleisli arrows, but in this paper we will avoid it.

In Haskell, every monad is strong in the sense that carries an additional operation, known as strength, with additional coherence properties. This happens because the extension operations of Haskell monads are necessarily internal.

```
mstrength :: Monad t => t a -> b -> t (a, b)
mstrength c b = c >>= \ a -> return (a, b)
```

The maybe monad is Haskell-implemented as follows.

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
  return a      = Just a
  Just a  >>= k = k a
  Nothing >>= k = Nothing
```

The exponent monad is implemented as follows.

```
newtype Exp e a = Exp (e -> a)

instance Monad (Exp e) where
  return a    = Exp (\ _ -> a)
  Env f >>= k = Exp (\ e -> case k (f e) of
                              Exp f' -> f' e)
```

In the case of these two monads the operations characteristic to the particular effects are raising and handling an error and consulting and local modification of the environment.

```
raise :: Maybe a
raise = Nothing

handle :: Maybe a -> Maybe a -> Maybe a
Just a  `handle` _ = Just a
Nothing `handle` c = c

askE :: Exp e e
askE = Exp id

localE :: (e -> e) -> Exp e a -> Exp e a
localE g (Exp f) = Exp (f . g)
```

Despite their generality, monads do not cater for every possible notion of impure function. In particular, monads do not cater for stream functions, which are the central concept in dataflow programming. In functional programming, Hughes [2000] has been promoting what he calls arrow types to overcome this deficiency. In semantics, the same concept was invented for the same reason by Power and Robinson [1997] under the name of a Freyd category.

Informally, a Freyd category is a symmetric premonoidal category together with an inclusion from a base category. A symmetric premonoidal category is the same as a symmetric monoidal category except that the tensor need not be bifunctorial, only functorial in each of its two arguments separately. The exact definition is a bit more complicated: A *binoidal* category is a category $\mathcal{K}$ with a binary operation $\otimes$ on objects of $\mathcal{K}$ that is functorial in each of its two arguments. A map $f : A \to B$ of such a category is called *central* if the two composites $A \otimes C \to B \otimes D$ agree and the two composites $C \otimes A \to D \otimes B$ agree for every map $g : C \to D$. A natural transformation is central if its components are central. A *symmetric premonoidal category* is a binoidal category $(\mathcal{K}, \otimes)$ together with an object I and central natural transformations ρ, α, σ with components $A \to A \otimes I$, $(A \otimes B) \otimes C \to A \otimes (B \otimes C)$, $A \otimes B \to B \otimes A$, subject to a number of coherence conditions. A *Freyd category* over a Cartesian category $\mathcal{C}$ is a symmetric premonoidal category $\mathcal{K}$ together with an identity on objects functor $J : \mathcal{C} \to \mathcal{K}$ that preserves the symmetric premonoidal structure of $\mathcal{C}$ on the nose and also preserves centrality.

The basic example of a Freyd category is the Kleisli category of a strong monad. But probably the best known and most useful example is that of stream functions. For a base category $\mathcal{C}$, the maps $A \to B$ of this category are the maps $\mathsf{Str}A \to \mathsf{Str}B$ of $\mathcal{C}$ where $\mathsf{Str}A = \nu X.A \times X$ is the type of streams over the type A.

In Haskell, arrow type constructors are implemented by the following type constructor class (appearing in `Control.Arrow`).

```
class Arrow r where
  pure :: (a -> b) -> r a b
  (>>>) :: r a b -> r b c -> r a c
```

```
  first  :: r a b -> r (a, c) (b, c)

returnA :: Arrow r => r a a
returnA = pure id

second :: Arrow r => r c d -> r (a, c) (a, d)
second f = pure swap >>> first f >>> pure swap
```

`pure` says that every function is an arrow (so in particular identity arrows arise from identity functions). `(>>>)` provides composition of arrows and `first` provides functoriality in the first argument of the tensor of the arrow category.

In Haskell, Kleisli arrows are shown to be an instance of arrows as follows (recall that all Haskell monads are strong).

```
newtype Kleisli t a b = Kleisli (a -> t b)

instance Monad t => Arrow (Kleisli t) where
  pure f = Kleisli (return . f)
  Kleisli k >>> Kleisli l = Kleisli ((>>= l) . k)
  first  (Kleisli k) = Kleisli (\ (a, c) -> mstrength (k a) c)
```

Stream functions are declared to be arrows in the following fashion. (For reasons of readability, we introduce our own list and stream types with our own names for their nil and cons constructors. Also, although Haskell does not distinguish between inductive and coinductive types, we want to make the distinction.)

```
data Stream a = a :< Stream a                      -- coinductive

mapS :: (a -> b) -> Stream a -> Stream b
mapS f (a :< as) = f a :< mapS f as

zipS :: Stream a ->  Stream b -> Stream (a, b)
zipS (a :< as) (b :< bs) = (a, b) :< zipS as bs

unzipS :: Stream (a, b) -> (Stream a, Stream b)
unzipS abs = (mapS fst abs, mapS snd abs)

newtype SF a b = SF (Stream a -> Stream b)

instance Arrow SF where
  pure f = SF (mapS f)
  SF k >>> SF l = SF (l . k)
  first (SF k) = SF (uncurry zipS . (\ (as, ds) -> (k as, ds)) . unzipS)
```

Similarly to monads, every useful arrow type constructor has some operation specific to it. The main such operation for stream functions are the initialized unit delay operation ‘followed by’ of intensional and synchronous dataflow languages and the unit anticipation operation ‘next’ that only exists in intensional languages. These are really the cons and tail operations of streams.

```
fbySF :: a -> SF a a
fbySF a0 = SF (\ as -> a0 :< as)

nextSF :: SF a a
nextSF = SF (\ (a : as) -> as)
```

3 Comonads

While Freyd categories or arrow types are certainly general and cover significantly more notions of impure functions than monads, some non-monadic impurities should be explainable in more basic terms, namely via comonads, which are the dual of monads. This has been suggested [Brookes and Geva 1992, Kieburtz 1999, Lewis et al. 2000], but there have been few useful examples. The goal of this paper is to show that stream functions and causal stream functions are excellent new such examples.

A *comonad* on a category $\mathcal{C}$ is given by a mapping $D : |\mathcal{C}| \to |\mathcal{C}|$ together with a $|\mathcal{C}|$-indexed family ε of maps $\varepsilon_A : DA \to A$ (*counit*), and an operation $-^\dagger$ taking every map $k : DA \to B$ in $\mathcal{C}$ to a map $k^\dagger : DA \to DB$ (*coextension operation*) such that

1. for any $k : DA \to B$, $\varepsilon_B \circ k^\dagger = k$,
2. ${\varepsilon_A}^\dagger = \mathsf{id}_{DA}$,
3. for any $k : DA \to B$, $\ell : DB \to C$, $(\ell \circ k^\dagger)^\dagger = \ell^\dagger \circ k^\dagger$.

Analogously to Kleisli categories, any comonad $(D, \varepsilon, -^\dagger)$ defines a category $\mathcal{C}_D$ where $|\mathcal{C}_D| = |\mathcal{C}|$ and $\mathcal{C}_D(A, B) = \mathcal{C}(DA, B)$, $(\mathsf{id}_D)_A = \varepsilon_A$, $\ell \circ_D k = \ell \circ k^\dagger$ (*coKleisli category*) and an identity on objects functor $J : \mathcal{C} \to \mathcal{C}_D$ where $Jf = f \circ \varepsilon_A$ for $f : A \to B$.

Comonads should be fit to capture notions of "value in a context"; DA would be the type of contextually situated values of A. A context-relying function from A to B would then be a map $A \to B$ in the coKleisli category, i.e., a map $DA \to B$ in the base category. The function $\varepsilon_A : DA \to A$ discards the context of its input whereas the coextension $k^\dagger : DA \to DB$ of a function $k : DA \to B$ essentially duplicates it (to feed it to k and still have a copy left).

Some examples of comonads are the following. Each object mapping D below is a comonad:

– $DA = A \times E$, the product comonad,

– $DA = \mathsf{Str}A = \nu X.A \times X$, the streams comonad,

– $DA = \nu X.A \times FX$, the cofree comonad over F,

– $DA = \mu X.A \times FX$, the cofree recursive comonad over F [Uustalu and Vene 2002].

Accidentally, the pragmatics of the product comonad is the same as that of the exponent monad: The Kleisli arrows of the exponent monad are the maps $A \to (E \Rightarrow B)$ of the base category, which are of course in a natural bijection with the maps $A \times E \to B$ that are the coKleisli arrows of the product comonad. But in general, monads and comonads capture different notions of impure function. We defer the discussion of the pragmatics of the streams comonad until the next section.

In Haskell, comonads are implemented as follows.

```
class Comonad d where
  counit :: d a -> a
  cobind :: (d a -> b) -> d a -> d b

cmap :: Comonad d => (a -> b) -> d a -> d b
cmap f = cobind (f . counit)
```

CoKleisli arrows are shown to be an instance of arrows as follows.

```
newtype CoKleisli d a b = CoKleisli (d a -> b)

instance Comonad d => Arrow (CoKleisli d) where
  pure f = CoKleisli (f . counit)
  CoKleisli k >>> CoKleisli l = CoKleisli (l . cobind k)
  first (CoKleisli k) = CoKleisli (pair (k . cmap fst) (snd . counit))
```

The product comonad is implemented as follows.

```
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 streams comonad is implemented as follows.

```
data Stream a = a :< Stream a                      -- coinductive

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

nextS :: Stream a -> a
nextS (_ :< (a' :< _)) = a'
```

4 Comonads for general and causal stream functions

All necessary preparations made, we are now ready to discuss the representation of general and causal stream functions via comonads.

Streams (discrete time signals) are naturally isomorphic to functions from natural numbers: $\mathsf{Str}A = \nu X.\, A \times X \cong (\mu X.\, 1 + X) \Rightarrow A = \mathsf{Nat} \Rightarrow A$. In Haskell, this isomorphism is implemented as follows:

```
data Stream a = a :< Stream a                      -- coinductive

str2fun :: Stream a -> Int -> a
str2fun (a :< as) 0 = a
str2fun (a :< as) (i + 1) = str2fun as i

fun2str :: (Int -> a) -> Stream a
fun2str f = fun2str' f 0

fun2str' f i = f i :< fun2str' f (i + 1)
```

Stream functions $\mathsf{Str}A \to \mathsf{Str}B$ are thus in natural bijection with maps $\mathsf{Nat} \Rightarrow A \to \mathsf{Nat} \Rightarrow B$, which, in turn, are in natural bijection with maps $(\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \to B$, i.e., $\mathsf{FunArg}A \to B$ where $\mathsf{FunArg}A = (\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat}$. Hence, for general stream functions, a value from A in context is a stream (signal) over A together with a natural number identifying a distinguished stream position (the present time). Not surprisingly, the object mapping FunArg is a comonad (in fact, it is an instance of the "state-in-context" comonad considered by Kieburtz [2000]) and, what is of crucial importance, the coKleisli identities and composition as well as the coKleisli lifting of this comonad agree with the identities and composition of stream functions (which are really just function identities and composition) and with the lifting of functions to stream functions. In Haskell, the comonad structure and the interpretation of coKleisli arrows as stream functions are implemented as follows:

```
data FunArg a = (Int -> a) :@ Int

instance Comonad FunArg where
  counit (f :@ i) = f i
  cobind k (f :@ i) = (\ i' -> k (f :@ i')) :@ i

runFA :: (FunArg a -> b) -> Stream a -> Stream b
runFA k as = runFA' k (str2fun as :@ 0)

runFA' k d@(f :@ i) = k d :< runFA' k (f :@ (i + 1))
```

The comonad FunArg can also be presented without resorting to natural numbers to deal with positions. The idea for this equivalent alternative presentation is simple: given a stream and a distinguished stream position, the position splits the stream up into a list, a value of the base type and a stream (corresponding to the past, present and future of the signal). Put mathematically, there is a natural isomorphism $(\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \cong \mathsf{Str}A \times \mathsf{Nat} \cong (\mathsf{List}A \times A) \times \mathsf{Str}A$ where $\mathsf{List}A = \mu X.\ 1 + (A \times X)$ is the type of lists over the type A. This gives us an equivalent comonad LVS (LVS for 'list-value-stream') for representing stream functions with the following structure (we use snoc-lists instead of cons-lists to reflect the fact that the analysis order of the past of a signal will be the reverse direction of time):

```
data List a = Nil | List a :> a                    -- inductive

data LV a = List a := a

data LVS a = LV a :| Stream a

instance Comonad LVS where
  counit (az := a :| as) = a
  cobind k d = cobindL d := k d :| cobindS d
     where cobindL (Nil        := a :| as)  = Nil
           cobindL (az' :> a'  := a :| as)  = cobindL d' :> k d'
                                 where d' = az' := a' :| (a :< as)
           cobindS (az := a :| (a' :< as')) = k d' :< cobindS d'
                                 where d' = az :> a := a' :| as'

runLVS :: (LVS a -> b) -> Stream a -> Stream b
runLVS k (a' :< as') = runLVS' k (Nil := a' :| as')

runLVS' k d@(az := a :| (a' :< as'))
                   = k d :< runLVS' k (az :> a := a' :| as')
```

The conversions between the two comonads are obvious and Haskell implemented as follows:

```
fa2lvs :: FunArg a -> LVS a
fa2lvs (f :@ 0)       = Nil     := a' :| as'
              where (a' :< as') = fun2str f
fa2lvs (f :@ (i + 1)) = az :> a := a' :| as'
              where (az := a :| (a' :< as')) = fa2lvs (f :@ i)

lvs2fa :: LVS a -> FunArg a
lvs2fa (Nil       := a :| as) = str2fun (a :< as) :@ 0
lvs2fa (az' :> a' := a :| as) = f                 :@ (i + 1)
              where (f :@ i) = lvs2fa (az' := a' :| (a :< as))
```

We can now program various stream functions as coKleisli arrows of FunArg or LVS. For example, delay, summation and differencing can be programmed as follows:

```
fbyFA :: a -> FunArg a -> a
fbyFA a (f :@ 0)       = a
fbyFA _ (f :@ (i + 1)) = f i

sumFA :: Num a => FunArg a -> a
```

```
sumFA (f :@ 0)       = f 0
sumFA (f :@ (i + 1)) = f (i + 1) + sumFA (f :@ i)

diffFA :: Num a => FunArg a -> a
diffFA (f :@ 0)       = f 0
diffFA (f :@ (i + 1)) = f (i + 1) - f i
```

All these stream functions are causal, in that the present of the output signal only depends on the past and present of the input signal. It is also possible to program non-causal stream functions such as anticipation or the compressing function which needs the first $2 * n$ elements of the input stream to produce the first n elements of the output stream.

```
nextFA :: FunArg a -> a
nextFA (f :@ i) = f (i + 1)

compressFA :: FunArg a -> (a, a)
compressFA (f :@ i) = (f (2 * i), f (2 * i + 1))
```

Could we ban non-causal functions? Yes, the comonad LVS is easy to modify so that exactly those stream functions can be represented that are causal. All that needs to be done is to remove from the comonad the factor of the future. We are left with the object mapping LV (LV for 'list-value') where $\mathsf{LV}A = \mathsf{List}A \times A = (\mu X.\, 1 + A \times X) \times A \cong \mu X.\, A \times (1 + X)$, i.e., a non-empty list type constructor. This is a comonad as well and again the counit and the coextension operation are just correct in the sense that they deliver the desirable coKleisli identities, composition and lifting. In fact, the comonad LV is the cofree recursive comonad of the functor Maybe (we refrain from giving the definition of a recursive comonad here, but one can be found in Uustalu and Vene [2002]). It may be useful to notice that the type constructor LV carries a monad structure too, but the Kleisli arrows of that monad have nothing to do with causal stream functions!

In Haskell, the non-empty list comonad is defined as follows:

```
instance Comonad LV where
  counit (_ := a) = a
  cobind k d@(az := _) = cobindL k az := k d
          where cobindL k Nil = Nil
                cobindL k (az :> a) = cobindL k az :> k (az := a)

runLV :: (LV a -> b) -> Stream a -> Stream b
runLV k (a' :< as') = runLV' k (Nil := a' :| as')

runLV' k (d@(az := a) :| (a' :< as'))
                  = k d :< runLV' k (az :> a := a' :| as')
```

Relying on this comonad, we can easily program pointwise application, delay, summation and differencing essentially as before.

```
fbyLV :: a -> (LV a -> a)
fbyLV a0 (Nil       := _) = a0
fbyLV _  ((_ :> a') := _) = a'

sumLV :: Num a => (LV a -> a)
sumLV (Nil         := a) = a
sumLV ((az' :> a') := a) = a + sumLV (az' := a')

diffLV :: Num a => (LV a -> a)
diffLV (Nil      := a) = a
diffLV (_ :> a') := a) = a - a'
```

It is easy to notice that the above definition of summation is very inefficient: if the corresponding stream function is applied to some stream of numbers, the same partial sums will be computed over and over again. Luckily, a remedy is available. The recursiveness of the comonad LV means that it is acceptable to define the n-th element of the output stream (the present of the output signal) in terms of the elements at positions $0..n$ (the past and present of the input signal) of the input stream plus the elements at positions $0..(n-1)$ of the output stream (the past of the output signal). I.e., one can manufacture a map $\mathsf{LV}A \to B$ from a map $\mathsf{List}(A \times B) \times A \to B$. (This is almost to say that it suffices to have a map $\mathsf{LV}(A \times B) \to B$, i.e., $\mathsf{List}(A \times B) \times (A \times B) \to B$, but not quite: such map is only fine if it discards the B portion of the source, that is, factors in the obvious fashion through $\mathsf{List}(A \times B) \times A$.) In terms of circuits, a recursive specification is a feedback loop with a mandatory delay. In Haskell, the corresponding combinator may be defined as follows.

```
rec :: (List (a, b) -> a -> b) -> (LV a -> b)
rec k d = k abz a
          where (abz := (a, _)) = cobind (pair counit (rec k)) d
```

This definition is not structurally recursive, so it is not a priori wellformed. But it can be shown to be equivalent to the following structurally recursive definition [Uustalu and Vene 2002].

```
rec k d = b where (_ := (_, b)) = rec' k d

rec' ::  (List (a, b) -> a -> b) -> (LV a -> LV (a, b))
rec' k (Nil       := a) = Nil := (a, k Nil a)
rec' k (az' :> a' := a) = abz := (a, k abz a)
                       where (abz' := ab') = rec' k (az' := a')
                             abz = abz' :> ab'
```

With the help of the recursion combinator, summation can redefined as follows.

```
sumLV = rec sumbase

sumbase :: Num a => List (a, a) -> a -> a
sumbase Nil           a = a
sumbase (_ :> (_, s)) a = a + s
```

Efficient implementations of recursive specifications are obtained by their direct interpretation as stream functions. This accomplished by the following definition.

```
runRec :: (List (a, b) -> a -> b) -> Stream a -> Stream b

{-
runRec k = run (rec k)
-}

runRec k (a' :< as') = runRec' k Nil a' as'

runRec' k abz a (a' :< as')
                    = b :< runRec' k (abz :> (a, b)) a' as'
                                        where b = k abz a
```

A remarkable feature of recursive specifications is that they can also be composed directly, as witnessed by the following code.

```
compbase :: (List (a, b) -> a -> b)
               -> (List ((a, b), c) -> (a, b) -> c)
                  -> List (a, (b, c)) -> a -> (b, c)
compbase k l abcz a = (b, c)
     where b = k (mapL (\ (a, (b, c)) -> (a, b))      abcz) a
           c = l (mapL (\ (a, (b, c)) -> ((a, b), c)) abcz) (a, b)
```

We finish this section by pointing out that analogously to causal stream functions, one might consider anticausal stream functions, i.e., stream functions where the n-th element of the output stream (the present of the output signal) can only depend on the elements at positions $n..$ of the given input stream (the present and future of the input signal). The comonad for anticausal stream functions is the streams comonad Str which we discussed as an example of a comonad in the previous section and which is very canonical by being the cofree comonad over the identity functor. But in real life we are more interested in causal than anticausal stream functions!

5 A distributive law for causal partial-stream functions

We shall now show that the comonadic approach works also for causal functions transforming partial streams (used in synchronous dataflow languages to model clocked signals). A partial stream is a stream that may have empty positions to indicate the pace of the clock of the signal wrt. the base clock. Causal partial-stream functions turn out to be describable in terms of a distributive combination of a comonad and a monad considered, e.g., in Brookes and Geva [1992], Power and Watanabe [2002].

Given a comonad $(D, \varepsilon, -^\dagger)$ and a monad $(T, \eta, -^\star)$ on a category $\mathcal{C}$, a *distributive law* of the former over the latter is a natural transformation λ with components $DTA \to TDA$ subject to four coherence conditions. A distributive law of

D over T defines a category $\mathcal{C}_{D,T}$ where $|\mathcal{C}_{D,T}| = |\mathcal{C}|$, $\mathcal{C}_{D,T}(A,B) = \mathcal{C}(DA,TB)$, $(\mathsf{id}_{D,T})_A = \eta_A \circ \varepsilon_A$, $\ell \circ_{D,T} k = l^\star \circ \lambda_B \circ k^\dagger$ for $k : DA \to TB$, $\ell : DB \to TC$ (call it the *biKleisli category*), with inclusions to it from both the coKleisli category of D and Kleisli category of T.

In Haskell, the distributive combination is implemented as follows.

```
class (Comonad d, Monad t) => Dist d t where
  dist :: d (t a) -> t (d a)

newtype BiKleisli d t a b = BiKleisli (d a -> t b)

instance Dist d t => Arrow (BiKleisli d t) where
  pure f = BiKleisli (return . f . counit)
  BiKleisli k >>> BiKleisli l = BiKleisli ((>>= l) . dist . cobind k)
  first (BiKleisli k) = BiKleisli (\ d ->
                   mstrength (k (cmap fst d)) (snd (counit d))
```

For partial-stream functions, it is appropriate to combine the causal stream functions comonad LV with the error monad Maybe given by $\mathsf{Maybe}A = 1 + A$. There is a distributive law which takes a partial list and and a partial value (the past and present of the signal according to the base clock) and, depending on whether the partial value is undefined or defined, gives back the undefined list-value pair (the present time does not exist according to the signal's own clock) or a defined list-value pair, where the list is obtained from the partial list by leaving out its undefined elements (the past and present of the signal according to its own clock). In Haskell, this distributive law is encoded as follows.

```
filterL :: List (Maybe a) -> List a
filterL Nil = Nil
filterL (az :> Nothing) = filterL az
filterL (az :> Just a)  = filterL az :> a

instance Dist LV Maybe where
  dist (az := Nothing) = Nothing
  dist (az := Just a)  = Just (filterL az := a)
```

The biKleisli arrows of the distributive law are interpreted as partial-stream functions as follows.

```
runLVM :: (LV a -> Maybe b) -> Stream (Maybe a) -> Stream (Maybe b)
runLVM k (a' :< as') = runLVM' k Nil a' as'

runLVM' k az Nothing  (a' :< as')
                   = Nothing     :< runLVM' k az        a' as'
runLVM' k az (Just a) (a' :< as')
                   = k (az := a) :< runLVM' k (az :> a) a' as'
```

The 'when' operation, used in synchronous dataflow languages to put a signal on a slower clock (by omitting its value on certain ticks of its clock), is implemented as follows.

```
whenLVM :: LV (a, Bool) -> Maybe a
whenLVM (_ := (a, False)) = Nothing
whenLVM (_ := (a, True))  = Just a
```

A nice demonstration of the combination of LV and Maybe at work is the example of the sieve of Eratosthenes. This function is the composite of the function taking a partial stream over the unit type 1 to the enumeration of natural numbers ≥ 2 of the same definedness pattern and of the function sieving a given partial stream wrt. divisibility by its elements. The output of an important auxiliary partial-stream function is equal at all its positions to the first seen defined element of the input.

In terms of the operations of the LV comonad, Maybe monad and the distributive law between them, the relevant biKleisli arrows are programmed as follows:

```
ini :: LV a -> Maybe a
ini (Nil        := a) = return a
ini ((az' :> a) := _) = ini (az' := a)

sieve :: LV Int -> Maybe Int
sieve   (Nil      := a) = return a
sieve d@((_ :> _) := _) = dist (cobind k d) >>= sieve
    where k d = let a = counit d  in
                ini d >>= \ a0 ->
                if a `mod` a0 /= 0 then return a else Nothing

nats2 :: LV () -> Maybe Int
nats2 (Nil        := _) = return 2
nats2 ((az' :> a) := _) = nats2 (az' := a) >>= \ i -> return (i + 1)

eratosthenes :: LV () -> Maybe Int
eratosthenes d = dist (cobind nats2 d) >>= sieve
```

With a bit of rearrangement, the above definitions can be written without resorting to the definitions of the type constructors LV and Maybe and instead using the operations 'raise' and 'followed-by':

```
ini :: LV a -> Maybe a
ini d = return (counit d) `fbyLV` cobind ini d

atini :: LV a -> Maybe Bool
atini d = return True `fbyLV` cobind (const (return False)) d

sieve :: LV Int -> Maybe Int
sieve d = atini d >>= \ b ->
      if b then return (counit d) else dist (cobind k d) >>= sieve
           where k d = let a = counit d  in
                       ini d >>=  \ a0 ->
                       if a `mod` a0 /= 0 then return a else raise

nats2 :: LV () -> Maybe Int
```

```
nats2 d = return 2 `fbyLV` cobind k d
            where k d = nats2 d >>= \ i -> return (i + 1)

eratosthenes :: LV () -> Maybe Int
eratosthenes d = dist (cobind nats2 d) >>= sieve
```

This style is of course hard to write manually, but could be generated by a distributive-law-based interpreter of a synchronous dataflow language.

6 Conclusion and future work

We have shown that both general and causal stream functions are characterized as coKleisli arrows of comonads in a way which is intuitive and reflects the essence of the respective programming paradigms. Moreover, functions on causal partial streams, used in synchronous dataflow languages to model clocked signals, are describable in terms of a distributive combination of a comonad and a monad.

We are confident that the approach of this paper extends to timed signals, but would like to work out the details. Another line to pursue will be to develop a complete semantic description of a (non-causal) intensional language along the lines of Lucid [Aschroft and Wadge 1985] or a (causal) synchronous dataflow language such as Lustre [Halbwachs 1991] or Lucid Synchrone [Caspi and Pouzet 1996, Pouzet 2001] to assess the benefits of our approach. Here we have already obtained some very promising concrete results which will be reported in a separate paper.

Acknowledgements

This research was partially funded by the Estonian Science Foundation grant No. 5567. Neil Ghani advised us to check combinations of comonads with monads based on distributive laws. The referees provided useful comments.

References

[Aczel et al. 2003] Aczel, P., Adámek, J., Milius, S., Velebil, J.: “Infinite trees and completely iterative theories: A coalgebraic view”; Theoret. Comput. Sci., 300, 1–3 (2003), 1–45.

[Aschroft and Wadge 1985] Ashcroft, E. A., Wadge W. W.: “LUCID, The Dataflow Programming Language”; Academic Press, New York (1985).

[Barbier 2002] Barbier, B.: “Solving stream equation systems”; Actes 13mes Journées Francophones des Langages Applicatifs, JFLA 2002 (2002), 117–139.

[Benton et al. 2002] Benton, N., Hughes, J., E. Moggi, E.: “Monads and effects”; Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.), Advanced Lectures from Int. Summer School on Applied Semantics, APPSEM 2000, Lect. Notes in Comput. Sci., 2395, Springer-Verlag, Berlin (2002), 42–122.

[Brookes and Geva 1992] Brookes, S., Geva, S.: "Computational comonads and intensional semantics"; Fourman, M. P., Johnstone, P. T., Pitts, A. M. (eds.), Applications of Categories in Computer Science, London Math. Society Lecture Note Series, 177, Cambridge University Press, Cambridge (1992), 1–44.

[Caspi and Pouzet 1996] Caspi, P., Pouzet, M.: "Synchronous Kahn networks"; Proc. of 1st ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'96, ACM Press, New York (1996), 226–238; also SIGPLAN Notices, 31, 6 (1996), 226–238.

[Caspi and Pouzet 1998] Caspi, P., Pouzet, M.: "A co-iterative characterization of synchronous stream functions"; Jacobs, B., Moss, L., Reichel, H., Rutten, J. (eds.), Proc. of 1st Wksh. on Coalgebraic Methods in Computer Science, CMCS'98, Electron. Notes in Theoret. Comput. Sci., 11, Elsevier, Amsterdam (1998).

[Halbwachs 1991] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: "The synchronous data flow programming language LUSTRE"; Proc. of the IEEE, 79, 9 (1991), 1305–1320.

[Hughes 2000] Hughes, J.: "Generalising monads to arrows"; Sci. of Comput. Program., 37, 1–3 (2000), 67–111.

[Kieburtz 1999] Kieburtz, R. B.: "Codata and comonads in Haskell"; unpublished manuscript (1999).

[Kieburtz 2000] Kieburtz, R. B.: "Coalgebraic techniques for reactive functional programming"; Actes 11mes Journées Francophones des Langages Applicatifs, JFLA 2000 (2000), 131–157.

[Lewis et al. 2000] Lewis, J. R, Shields, M. B., Meijer, E., Launchbury, J.: "Implicit parameters: Dynamic scoping with static types"; Proc. of 27th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'00, ACM Press, New York (2000), 108–118.

[Moggi 1991] Moggi, E.: "Notions of computation and monads"; Inform. and Comput., 93, 1 (1991), 55–92.

[Nilsson et al. 2002] Nilsson, H., Courtney, A., Peterson, J.: "Functional reactive programming, continued"; Proc. of 2002 ACM SIGPLAN Wksh. on Haskell, Haskell'02, ACM Press, New York (2002), 51–64.

[Paterson 2001] Paterson, R.: "A new notation for arrows"; Proc. of 6th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'01, ACM Press, New York (2001),229–240; also SIGPLAN Notices, 36, 10 (2001), 229–240.

[Paterson 2003] Paterson, R.: "Arrows and computation"; Gibbons, J., de Moor, O. (Eds.), The Fun of Programming, Cornerstones of Computing, Palgrave MacMillan, Basingstoke / New York (2003), 201–222.

[Pouzet 2001] Pouzet, M.: "Lucid Synchrone: tutorial and reference manual"; unpublished manuscript (2001).

[Power and Robinson 1997] Power, J., Robinson, E.: "Premonoidal categories and notions of computation"; Math. Structures in Comput. Sci., 7, 5 (1997), 453–468.

[Power and Watanabe 2002] Power, J., Watanabe, H.: "Combining a monad and a comonad"; Theoret. Comput. Sci., 280, 1–2 (2002), 137–162.

[Rutten 2000] Rutten, J. J. M. M.: "Universal coalgebra: a theory of systems"; Theoret. Comput. Sci., 249, 1 (2000), 3–80.

[Rutten 2003] Rutten, J. J. M. M.: "Behavioural differential equations: a coinductive calculus of streams, automata, and power series"; Theoret. Comput. Sci., 308, 1–3 (2003), 1–53.

[Uustalu and Vene 2002] Uustalu, T., Vene, V.: "The dual of substitution is redecoration"; Hammond, K., Curtis, S. (eds.), Trends in Functional Programming 3, Intellect, Bristol / Portland, OR (2002), 99–110.

[Wadler 1992] Wadler, P.: "The essence of functional programming"; Conf. Record of 19th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'92, ACM Press, New York (1992), 1–14.