The Essence of Dataflow Programming (Short Version)

Tarmo Uustalu[1] and Varmo Vene[2]

[1] Inst. of Cybernetics at Tallinn Univ. of Technology,
Akadeemia tee 21, EE-12618 Tallinn, Estonia; `tarmo@cs.ioc.ee`
[2] Dept. of Computer Science, Univ. of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia; `varmo@cs.ut.ee`

Abstract. We propose a novel, comonadic approach to dataflow (stream-based) computation. This is based on the observation that both general and causal stream functions can be characterized as coKleisli arrows of comonads and on the intuition that comonads in general must be a good means to structure context-dependent computation. In particular, we develop a generic comonadic interpreter of languages for context-dependent computation and instantiate it for stream-based computation. We also discuss distributive laws of a comonad over a monad as a means to structure combinations of effectful and context-dependent computation. We apply the latter to analyse clocked dataflow (partial stream based) computation.

1 Introduction

Ever since the work by Moggi and Wadler [20,33], we know how to reduce impure computations with errors and non-determinism to purely functional computations in a structured fashion using the maybe and list *monads*. We also know how to explain other types of *effect*, such as *continuations*, *state*, even *input/output*, using monads!

But what is more unnatural or hard about the following program?

```
pos  = 0 fby (pos + 1)
fact = 1 fby (fact * (pos + 1))
```

This represents a *dataflow* computation which produces two discrete-time signals or streams: the enumeration of the naturals and the graph of the factorial function. The syntax is essentially that of Lucid [2], which is an old *intensional* language, or Lustre [13] or Lucid Synchrone [8,25], the newer *French synchronous dataflow* languages. The operator fby reads 'followed by' and means initialized unit delay of a discrete-time signal (cons of a stream).

Notions of dataflow computation cannot be structured by monads. As a substitute, one can use the laxer framework of *Freyd categories* or *arrow types*, proposed independently by Power and Robinson [26] and Hughes [15]. The message of this paper is that while this works, one can alternatively use something much more simple and standard, namely *comonads*, the formal dual of comonads. Moreover, comonads are even better, as there is more relevant structure to them than to Freyd categories. We also mean to claim that comonads have received too little attention in semantics compared to monads. This is unfair, since just as monads are useful for speaking and reasoning about notions of functions that produce effects, comonads can handle context-dependent functions and are hence highly relevant. This has been suggested earlier, e.g., by Brookes and Geva [6] and Kieburtz [17], but never caught on because of a lack of compelling examples. But now dataflow computation provides clear examples and it hints at a direction in which there are more.

Technically, we show that general and causal stream functions, the basic entities in intensional and synchronous dataflow computation, are elegantly described in terms of comonads. Imitating monadic interpretation, we develop a generic comonadic interpreter for context-dependent computation. By instantiation, we obtain interpreters of a Lucid-like intensional language and a Lucid Synchrone-like synchronous dataflow language. Remarkably, we get higher-order language designs with almost no effort whereas the traditional dataflow languages are first-order and the question of the meaningfulness or right meaning of higher-order dataflow has been seen as controversial. We also show that clocked dataflow (i.e., partial-stream based) computation can be handled by distributive laws of the comonads for stream functions over the maybe monad.

The organization of the paper is as follows. In Section 2, we introduce comonads and argue that they structure computation with context-dependent functions. We show that both general and causal stream functions are smoothly described by comonads and develop a comonadic interpreter capable of handling dataflow languages. In Section 3, we show how effects and context-dependence can be combined in the presence of a distributive law of the comonad over the monad, show how this applies to partial-stream functions and present a distributivity-based interpreter which copes with clocked dataflow languages. Section 4 is a summary of related work, while Section 5 lists our conclusions.

We assume that the reader is familiar with the basics of functional programming (in particular, Haskell programming), denotational semantics and the Lambek-Lawvere correspondence between typed lambda calculi and cartesian closed categories (the types-as-objects, terms-as-morphisms correspondence). Acquaintance with dataflow programming (Lucid or Lucid Synchrone) will be of additional help. Monads and arrow types are not introduced in this short version of the paper, but comonads and distributive laws are.

The paper is related to our earlier paper [31], which discussed the relevance of comonads for dataflow computation but did not treat comonad-based processing of dataflow languages.

2 Comonads

2.1 Comonads and context-dependent functions

We start by defining what comonads are and explaining their intuitive relevance for notions of impure computation.

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 $\epsilon_A : DA \to A$ (*counit*), and an operation $-^\star$ 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 $k : DA \to B$, $\epsilon_B \circ k^\dagger = k$,
2. ${\epsilon_A}^\dagger = \mathsf{id}_{DA}$,
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 $(D, \epsilon, -^\dagger)$ defines a category $\mathcal{C}_D$ where $|\mathcal{C}_D| = |\mathcal{C}|$ and $\mathcal{C}_D(A, B) = \mathcal{C}(DA, B)$, $(\mathsf{id}_D)_A = \epsilon_A$, $\ell \circ_D k = \ell \circ k^\dagger$ (*coKleisli category*) and an identity on objects functor $J : \mathcal{C} \to \mathcal{C}_D$ where $Jf = f \circ \epsilon_A$ 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-dependent 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 $\epsilon_A : DA \to A$ 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:

– $DA = A$, the identity 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 [30].

Accidentally, the pragmatics of the product comonad is the same as that of the exponent monad, viz. representation of functions reading an environment. The reason is simple: 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 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 subsection (it is NOT the comonad to represent general or causal stream functions!).

For Haskell, there is no standard comonad library[3]. But of course comonads are easily defined as a type constructor class analogously to the definition of monads in Prelude and Monad.

[3] There is, however, a contributed library by Dave Menendez, see `http://www.eyrie.org/~zednenem/2004/hsce/`.

```
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)
```

The identity and product comonads are defined as instances in the following fashion.

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

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 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 -> Stream a
nextS (a :< as) = as
```

2.2 Comonads for general and causal stream functions

The general pragmatics of comonads introduced, we are now ready to discuss the representation of general and causal stream functions via comonads.

The first observation to make is that 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)
```

General 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., $\mathsf{FunArg}\,\mathsf{Nat}\,A \to B$ where $\mathsf{FunArg}\,S\,A = (S \Rightarrow A) \times S$. 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 $\mathsf{FunArg}\,S$ is a comonad (in fact, it is the "state-in-context" comonad considered by Kieburtz [17]) and, what is of crucial importance, the coKleisli identities and composition as well as the coKleisli lifting of $\mathsf{FunArg}\,\mathsf{Nat}$ 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 parameterized comonad FunArg and the interpretation of the coKleisli arrows of $\mathsf{FunArg}\,\mathsf{Nat}$ as stream functions are implemented as follows.

```
data FunArg s a = (s -> a) :# s

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

runFA :: (FunArg Int 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 $\mathsf{FunArg}\,\mathsf{Nat}$ can also be presented equivalently without using natural numbers to deal with positions. The idea for this 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 a given type A. This gives us an equivalent comonad LVS for representing of 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'
```

(Notice the visual purpose of our constructor naming. In values of types $\mathsf{LVS}\,A$, both the cons constructors (:>) of the list (the past) and the cons constructors (:<) of the stream (the future) point to the present which is enclosed between the constructors (:=) and (: |).)

The interpretation of the coKleisli arrows of the comonad LVS as stream functions is implemented as follows.

```
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')
```

Delay and anticipation are easily formulated in terms of both $\mathsf{FunArg}\ \mathsf{Nat}$ and LVS.

```
fbyFA :: a -> (FunArg Int a -> a)         fbyLVS :: a -> (LVS a -> a)
fbyFA a0 (f :# 0)       = a0              fbyLVS a0 (Nil       := _ :| _) = a0
fbyFA _  (f :# (i + 1)) = f i             fbyLVS _  ((_ :> a') := _ :| _) = a'

nextFA :: FunArg Int a -> a               nextLVS :: LVS a -> a
nextFA (f :# i) = f (i + 1)               nextLVS (_ := _ :| (a :< _)) = a
```

Let us call a stream function *causal*, if the present of the output signal only depends on the past and present of the input signal and not on its future[4].

Is there a way to 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 LVS the factor of the future. We are left with the object mapping LV 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 [30]). 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 LV 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')
```

With the LV comonad, anticipation is no longer possible, but delay is unproblematic.

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

[4] The standard terminology is '*synchronous* stream functions', but we want to avoid it because 'synchrony' also refers to all signals being on the same clock and to instantaneous reactions.

Analogously to causal stream functions, one might also consider *anticausal* stream functions, i.e., functions for which the present value of the output signal only depends on the present and future values of the input signal. Since $A \times \mathsf{Str}\, A \cong \mathsf{Str}\, A$, it is not surprising now anymore that the comonad for anticausal stream functions is the comonad Str, which we introduced earlier and which is very canonical by being the cofree comonad generated by the identity functor. However, in real life, causality is much more relevant than anticausality!

2.3 Comonadic semantics

Is the comonadic approach to context-dependent computation of any use? We will now demonstrate that it is indeed by developing a generic comonadic interpreter instantiable to various specific comonads, in particular to those that characterize general and causal stream functions. In the development, we mimic the monadic interpreter of Moggi and Wadler [20,33].

As the first thing we must fix the syntax of our object language. We will support a purely functional core and additions corresponding to various notions of context.

```
type Var = String

data Tm = V Var | L Var Tm | Tm :@ Tm | Rec Tm
        | N Integer | Tm :+ Tm | ... | Tm :== Tm | ... | TT | FF | Not Tm | ... | If Tm Tm Tm
        -- specific for both general and causal stream functions
        | Tm `Fby` Tm
        -- specific for general stream functions only
        | Next Tm
```

The type-unaware semantic domain contains integers, booleans and functions, but functions are context-dependent (coKleisli functions). Environments are lists of variable-value pairs as usual.

```
data Val d = I Integer | B Bool | F (d (Val d) -> Val d)

type Env d = [(Var, Val d)]
```

We will manipulate environment-like entities via the following three functions[5].

```
empty :: [(a, b)]
empty = []

update :: a -> b -> [(a, b)] -> [(a, b)]
update a b abs = (a, b) : abs

unsafeLookup :: Eq a => a -> [(a, b)] -> b
unsafeLookup a0 ((a, b) : abs) = if a0 == a then b else unsafeLookup a0 abs
```

And we are at evaluation. Of course terms must denote coKleisli arrows, so the typing of evaluation is uncontroversial.

[5] The safe lookup, that maybe returns a value, will be unnecessary, since we can type-check object-language terms before evaluating them. If this succeeds, we can be sure we will only look for variables in environments where they really occur.

```
class Comonad d => ComonadEv d where
  ev :: Tm -> d (Env d) -> Val d
```

But an interesting issue arises with evaluation of closed terms. In the case of a pure or a monadically interpreted language, closed terms are supposed to be evaluated in the empty environment. Now they must be evaluated in the empty environment placed in a context! What does this mean? This is easy to understand on the example of stream functions. By the types, evaluation of an expression returns a single value, not a stream. So the stream position of interest must be specified in the contextually situated environment that we provide. Very suitably, this is exactly the information that the empty environment in a context conveys. So we can define:

```
evClosedI :: Tm -> Val Id
evClosedI e = ev e (Id empty)

emptyL :: Int -> List   [(a, b)]
emptyL 0       = Nil
emptyL (i + 1) = emptyL i :> empty

emptyS :: Stream [(a, b)]
emptyS = empty :< emptyS

evClosedLVS :: Tm -> Int -> Val LVS
evClosedLVS e i = ev e (emptyL i := empty :| emptyS)

evClosedLV  :: Tm -> Int -> Val LV
evClosedLV  e i = ev e (emptyL i := empty)
```

Back to evaluation. For most of the core constructs, the types tell us what the defining clauses of their meanings must be—there is only one thing we can write and that is the right thing. In particular, everything is meaningfully predetermined about variables, application and recursion. E.g., for a variable, we must extract the environment from its context (e.g., history), and then do a lookup. For an application, we must evaluate the function wrt. the given contextually situated environment and then apply it. But since, according to the types, a function wants not just an isolated argument value, but a contextually situated one, the function has to be applied to the coextension of the denotation of the argument wrt. the given contextually situated environment.

```
_ev :: ComonadEv d => Tm -> d (Env d) -> Val d
_ev (V x)        denv = unsafeLookup x (counit denv)
_ev (e :@ e')    denv = case ev e denv of
                          F f -> f (cobind (ev e') denv)
_ev (Rec e)      denv = case ev e denv of
                          F f -> f (cobind (_ev (Rec e)) denv)
_ev (N n)        denv = I n
_ev (e0 :+ e1)   denv = case ev e0 denv of
                          I n0 -> case ev e1 denv of
                            I n1 -> I (n0 + n1)
...
_ev TT           denv = B True
_ev FF           denv = B False
_ev (Not e)      denv = case ev e denv of B b -> B (not b)
...
_ev (If e e0 e1) denv = case ev e denv of B b -> if b then ev e0 denv else ev e1 denv
```

There is, however, a problem with lambda-abstraction. For any potential contextually situated value of the lambda-variable, the evaluation function should recursively evaluate the body of the lambda-abstraction expression in the appropriately extended contextually situated environment. Schematically,

```
_ev (L x e)      denv = F (\ d -> ev e (extend x d denv))
```

where

```
extend :: Comonad d => Var -> d (Val d) -> d (Env d) -> d (Env d)
```

Note that we need to combine a contextually situated environment with a contextually situated value. One way to do this would be to use the strength of the comonad (we are in Haskell, so every comonad is strong), but in the case of the stream function comonads this would necessarily have the bad effect that either the history of the environment or that of the value would be lost. We would like to see that no information is lost, to have the histories zipped.

To solve the problem, we consider comonads equipped with a special zipping operation. We define a *zippable comonad* to be a comonad D with a natural transformation ζ with components $\zeta_{A,B} : DA \times DB \to D(A \times B)$ that satisfies coherence conditions such as $\epsilon_{A \times B} \circ \zeta_{A,B} = \epsilon_A \times \epsilon_B$.

In Haskell, we define a corresponding type constructor class.

```
class Comonad d => ComonadZip d where
  czip :: d a -> d b -> d (a, b)
```

The identity comonad, as well as LVS and LV are instances (and so are many other comonads, e.g., the product comonad).

```
instance ComonadZip Id  where
  czip (Id a) (Id b) = Id (a, b)

zipL :: List a -> List b -> List (a, b)
zipL Nil       _         = Nil
zipL _         Nil       = Nil
zipL (az :> a) (bz :> b) = zipL az bz :> (a, b)

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

instance ComonadZip LVS where
  czip (az := a :| as) (bz := b :| bs) = zipL az bz := (a, b) :| zipS as bs

instance ComonadZip LV  where
  czip (az := a)       (bz := b)       = zipL az bz := (a, b)
```

With the zip operation available, defining the meaning of lambda-abstractions is easy, but we must also update the typing of the evaluation function, so that zippability becomes required.

```
class ComonadZip d => ComonadEv d where ...

_ev (L x e)      denv = F (\ d -> ev e (cmap repair (czip d denv)))
                                where repair (a, env) = update x a env
```

It remains to define the meaning of the specific constructs of our example languages. The pure language has none. The dataflow languages have Fby and Next that are interpreted using the specific operations of the corresponding comonads. Since each of Fby and Next depends on the context of the value of its main argument, we need to apply the coextension operation to the denotation of that argument to have this context available.

```
instance ComonadEv Id  where
  ev e            denv = _ev e denv

instance ComonadEv LVS where
  ev (e0 `Fby` e1) denv = ev e0 denv `fbyLVS` cobind (ev e1) denv
  ev (Next e)      denv = nextLVS (cobind (ev e) denv)
  ev e             denv = _ev e denv

instance ComonadEv LV where
  ev (e0 `Fby` e1) denv = ev e0 denv `fbyLV`  cobind (ev e1) denv
  ev e denv = _ev e denv
```

In dataflow languages, the 'followed by' construct is usually defined to mean the delay of the second argument initialized by the initial value of the first argument, which may at first seem like an ad hoc decision (or so it seemed to us at least). Why give the initial position any priority? In our interpreter, we took the simplest possible solution of using the value of the first argument of Fby in the present position of the history of the environment. We did not use any explicit means to calculate the value of that argument wrt. the initial position. But the magic of the definition of fbyLVS is that it only ever uses its first argument when the second has a history with no past (which corresponds to the situation when the present actually is the initial position in the history of the environment). So our most straightforward naive design gave exactly the solution that has been adopted by the dataflow languages community, probably for entirely different reasons.

Notice also that we have obtained a generic comonads-inspired language design which supports higher-order functions and the solution was dictated by the types. This is remarkable since dataflow languages are traditionally first-order and the question of the right meaning of higher-order dataflow has been considered controversial. The key idea of our solution can be read off from the interpretation of application: the present value of a function application is the present value of the function applied to the history of the argument.

We can test the interpreter on a few classic examples from dataflow programming. The following examples make sense in both the general and causal stream function settings.

```
-- pos    = 0 fby pos + 1
pos  = Rec (L "pos" (N 0 `Fby` (V "pos" :+ N 1)))
-- sum x  = x + (0 fby sum x)
sum  = L "x" (Rec (L "sumx" (V "x" :+ (N 0 `Fby` V "sumx"))))
-- diff x = x - (0 fby x)
diff = L "x" (V "x" :- (N 0 `Fby` V "x"))
-- ini  x = x fby ini x
ini  = L "x" (Rec (L "inix" (V "x" `Fby` V "inix")))
-- fact = 1 fby (fact * (pos + 1))
fact = Rec (L "fact" (N 1 `Fby` (V "fact" :* (pos :+  N 1))))
-- fibo = 0 fby (fibo + (1 fby fibo))
fibo = Rec (L "fibo" (N 0 `Fby` (V "fibo" :+ (N 1 `Fby` V "fibo"))))
```

Testing gives expected results:

```
> runLV (ev pos) emptyS
0 :< (1 :< (2 :< (3 :< (4 :< (5 :< (6 :< (7 :< (8 :< (9 :< (10 :< ...
> runLV (ev (sum :@ pos)) emptyS
0 :< (1 :< (3 :< (6 :< (10 :< (15 :< (21 :< (28 :< (36 :< (45 :< (55 :< ...
> runLV (ev (diff :@ (sum :@ pos)) emptyS
0 :< (1 :< (2 :< (3 :< (4 :< (5 :< (6 :< (7 :< (8 :< (9 :< (10 :< ...
```

Here are two examples that are only allowed with general stream functions, because of using anticipation: the 'whenever' operation and the sieve of Eratosthenes.

```
-- x wvr y = if ini y then x fby (next x wvr next y) else (next x wvr next y)
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 x = x fby sieve (x wvr x mod (ini x) :/= N 0)
sieve = Rec (L "sieve" (L "x" (
              V "x" `Fby` (V "sieve" :@ (wvr :@ V "x" :@ (V "x" `Mod` (ini :@ (V "x")) :/= N 0))))))
-- eratosthenes = sieve (pos + 2)
eratosthenes = sieve :@ (pos :+ N 2)
```

Again, testing gives what one would like to get.

```
> runLVS (ev eratosthenes) emptyS
2 :< (3 :< (5 :< (7 :< (11 :< (13 :< (17 :< (19 :< (23 :< (29 :< ...
```

3 Distributive Laws

3.1 Distributive laws; a distributive law for causal partial-stream functions

While the comonadic approach is quite powerful, there are natural notions of impure computation that it does not cover. One example is clocked dataflow or partial-stream based computation. The idea of clocked dataflow is that different signals may be on different clocks. Clocked dataflow signals can be represented by partial streams. A partial stream is a stream that may have empty positions to indicate the pace of the clock of a signal wrt. the base clock. The idea is to get rid of the different clocks by aligning all signals wrt. the base clock.

A very good news is that although comonads alone do not cover clocked dataflow computation, a solution is still close at hand. General and causal partial-stream functions turn out to be describable in terms of distributive combinations of a comonad and a monad considered, e.g., in [6,27]. For reasons of space, we will only discuss causal partial-stream functions as more relevant. General partial-stream functions are handled completely analogously.

Given a comonad $(D, \epsilon, -^\dagger)$ 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 $|\mathcal{C}_{D,T}| = |\mathcal{C}|$, $\mathcal{C}_{D,T}(A, B) = \mathcal{C}(DA, TB)$,

$(\mathsf{id}_{D,T})_A = \eta_A \circ \epsilon_A$, $\ell \circ_{D,T} k = l^\star \circ \lambda_B \circ k^\dagger$ 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. If the monad T is strong, the biKleisli category is a Freyd category.

In Haskell, the distributive combination is implemented as follows.

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

The simplest examples of distributive laws are the distributivity of the identity comonad over any monad and the distributivity of any comonad over the identity monad.

```
instance Monad t => Dist Id t where
  dist (Id c) = mmap Id c

instance ComonadZip d => Dist d Id where
  dist d = Id (cmap unId d)
```

A more interesting example is the distributive law of the product comonad over the maybe monad.

```
data Maybe a = Just a | Nothing

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

instance Dist Prod Maybe where
  dist (Nothing :& _) = Nothing
  dist (Just a  :& e) = Just (a :& e)
```

For causal partial-stream functions, it is appropriate to combine the causal stream functions comonad LV with the maybe monad. And this is possible, since there is a distributive law which takes a partial list 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 coded 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'
```

3.2 Distributivity-based semantics

Just as with comonads, we demonstrate distributive laws in action by presenting an interpreter. This time this is an interpreter of languages featuring both context-dependence and effects.

As previously, our first step is to fix the syntax of the object language.

```
type Var = String

data Tm = V Var | L Var Tm | Tm :@ Tm | Rec Tm
        | N Integer | Tm :+ Tm | ... | Tm :== Tm | ... | TT | FF | Not Tm | ... | If Tm Tm Tm
        -- specific for causal stream functions
        | Tm `Fby` Tm
        -- specific for partiality
        | Nosig | Tm `Merge` Tm
```

In the partiality part, Nosig corresponds to a nowhere defined stream, i.e., a signal on an infinitely slow clock. The function of Merge is to combine two partial streams into one which is defined wherever at least one of the given partial streams is defined.

The semantic domains and environments are defined as before, except that functions are now biKleisli functions, i.e., they take contextually situated values to values with an effect.

```
data Val d t = I Integer | B Bool | F (d (Val d t) -> t (Val d t))

type Env d t = [(Var, Val d t)]
```

Evaluation sends terms to biKleisli arrows; closed terms are interpreted in the empty environment placed into a context of interest.

```
class Dist d t => DistEv d t where
  ev :: Tm -> d (Env d t) -> t (Val d t)

evMainLV :: DistEv LV t => Tm -> Int -> t (Val LV t)
evMainLV e i = ev e (emptyL i := empty)
```

The meaning of the core constructs are essentially dictated by the types.

```
_ev :: DistEv d t => Tm -> d (Env d t) -> t (Val d t)
_ev (V x)       denv = return (unsafeLookup x (counit denv))
_ev (L x e)     denv = return (F (\ d -> ev e (cmap repair (czip d denv))))
                               where repair (a, env) = update x a env
_ev (e :@ e')   denv = ev e denv >>= \ (F f) ->
                       dist (cobind (ev e') denv) >>= \ d ->
                       f d
_ev (Rec e)     denv = ev e denv >>= \ (F f) ->
                       dist (cobind (_ev (Rec e)) denv) >>= \ d ->
                       f d
_ev (N n)       denv = return (I n)
_ev (e0 :+ e1)  denv = ev e0 denv >>= \ (I n0) ->
                       ev e1 denv >>= \ (I n1) ->
                       return (I (n0 + n1))
...
_ev TT          denv = return (B True )
_ev FF          denv = return (B False)
_ev (Not e)     denv = ev e denv >>= \ (B b) -> return (B (not b))
_ev (If e e0 e1) denv = ev e denv >>= \ (B b) -> if b then ev e0 denv else ev e1 denv
```

In monadic interpretation, the Rec operator is problematic, because recursive calls get evaluated too eagerly. The solution is to equip monads with a specific monadic fixpoint combinator mfix by making them instances of a type constructor class MonadFix (from Control.Monad.Fix). The same problem occurs here and is remedied by introducing a type constructor class DistCheat with a member function cobindCheat. The distributive law of LV over Maybe is an instance.

```
class Dist d t => DistCheat d t where
  cobindCheat :: (d a -> t b) -> (d a -> d (t b))

instance DistCheat LV Maybe where
  cobindCheat k d@(az := _) = cobindL k az := return (unJust (k d))
                   where cobindL k Nil = Nil
                         cobindL k (az :> a) = cobindL k az :> k (az := a)
```

Using the operation of the DistCheat class, the meaning of Rec can be redefined to yield a working solution.

```
class DistCheat d t => DistEv d t where ...

_ev (Rec e)      denv = ev e denv >>= \ (F f) ->
                        dist (cobindCheat (_ev (Rec e)) denv) >>= \ d->
                        f d
```

The meanings of the constructs specific to the extension are also dictated by the types and here we can and must of course use the specific operations of the particular comonad and monad.

```
instance DistEv LV Maybe where
  ev (e0 `Fby` e1) denv = ev e0 denv `fbyLV` cobind (ev e1) denv
  ev Nosig           denv = raise
  ev (e0 `Merge` e1) denv = ev e0 denv `handle` ev e1 denv
```

Partiality makes it possible to define a version of Eratosthenes sieve even in the causal setting (recall that our previous version without partiality used anticipation).

```
-- sieve x = if (tt fby ff) then x else sieve (if (x mod ini x /= 0) then x else nosig)
sieve = Rec (L "sieve" (L "x" (
               If (TT `Fby` FF)
                  (V "x")
                  (V "sieve" :@
                    (If ((V "x" `Mod` (ini :@ V "x")) :/= N 0) (V "x") Nosig)))))
-- eratosthenes = sieve (pos + 2)
eratosthenes = sieve :@ (pos :+ N 2)
```

Indeed, testing the above program, we get exactly what we would wish.

```
> runLVM (ev eratosthenes) emptyS
Just 2 :< (Just 3 :< (Nothing :< (Just 5 :< (Nothing :< (Just 7 :< (Nothing :< (Nothing :<
(Nothing :< (Just 11 :< (Nothing :< (Just 13 :< (Nothing :< (Nothing :< (Nothing :< (Just 17 :< ...
```

4 Related Work

Semantic studies of Lucid, Lustre and Lucid Synchrone-like languages are not many and concentrate largely on the so-called clock calculus for static well-clockedness checking [7,8,11]. Relevantly for us, however, Colaço et al. [10] have very recently proposed a higher-order synchronous dataflow language extending Lucid Synchrone, with two type constructors of function spaces.

Hughes's arrows [15] have been picked up very well by the functional programming community (for overviews, see [24,16]). There exists by now not only a de facto standardized arrow library in Haskell, but even specialized syntax [23]. The main application is functional reactive programming with its specializations to animation, robotics etc. [21,14]. Functional reactive programming is continuous-time event-based dataflow programming.

Uses of comonads to structure notions of computation have been very few. Brookes and Geva [6] were the first to suggest this application. Kieburtz [17] made an attempt to draw the attention of functional programmers to comonads. Lewis et al. [19] must have contemplated employing the product comonad to handle implicit parameters, but did not carry out the project. Comonads have also been used in the semantics of linear logic and modal logics, with their applications in staged computation and elsewhere, see e.g., [12], and to analyse structured recursion schemes, see e.g., [32,22], but these are relatively unrelated to our application here.

Our comonadic approach to stream-based programming is, to the best of our knowledge, entirely new. This is surprising, given how elementary it is. Workers in dataflow langugaes have produced a number of papers exploiting the final coalgebraic structure of streams [9,18,4], but apparently nothing on stream functions and comonads. The same is true about works in universal coalgebra [28,29].

5 Conclusions and Future Work

We have shown that notions of dataflow computation can be structured by suitable comonads, thus reinforcing the old idea that one should be able to use comonads to structure notions of context-dependent computation. We have demonstrated that the approach is fruitful with generic comonadic and distributivity-based interpreters that effectively suggest designs of dataflow languages. This is thanks to the rich structure present in comonads and distributive laws which essentially forces many design decisions (compare this to the much weaker structure in arrow types). Remarkably, the language designs that these interpreters suggest either coincide with the designs known from the dataflow languages literature or improve on them (when it comes to higher-orderness or to the choice of the primitive constructs in the case of clocked dataflow). For us, this is a solid proof of the true essence and structure of dataflow computation lying in comonads.

For future work, we envisage the following directions, in each of which we have already taken the first steps. First, we wish to obtain a solid understanding of the mathematical properties of comonads with a zip operation and of our comonadic and distributivity-based semantics. Second, we plan to look at guarded recursion schemes associated to the comonads for stream functions and at language designs based on corresponding constructs. Third, we plan to test our interpreters on other comonads (e.g., decorated tree types) and see if they yield useful computation paradigms and language designs. Fourth, we also intend to study the pragmatics of the combination of two comonads via a distributive law. We believe that this will among other things explicate the underlying enabling structure of language designs such Multidimensional Lucid [3] where flows are multidimensional arrays. Fifth, the interpreters we have provided have been designed as reference specifications of language semantics. As implementations, they are grossly inefficient because of careless use of recursion, and we plan to investigate systematic efficient implementation of the languages they specify based on interpreter transformations. Sixth, we intend to take a close look at continuous-time event-based dataflow computation.

Acknowledgments We are grateful to Neil Ghani for his suggestion to also look into distributive laws. This work was partially supported by the Estonian Science Foundation grant No. 5567.

References

1. P. Aczel, J. Adámek, S. Milius, J. Velebil. Infinite trees and completely iterative theories: A coalgebraic view. *Theoret. Comput. Sci.*, 300 (1–3), pp. 1–45, 2003.
2. E. A. Ashcroft, W. W. Wadge. *LUCID, The Dataflow Programming Language.* Academic Press, New York, 1985.
3. E. A. Ashcroft, A. A. Faustini, R. Jagannathan, W. W. Wadge. *Multidimensional Programming.* Oxford University Press, New York, 1995.
4. B. Barbier. Solving stream equation systems. In *Actes 13mes Journées Francophones des Langages Applicatifs, JFLA 2002*, pp. 117–139. 2002.
5. N. Benton, J. Hughes, E. Moggi. Monads and effects. In G. Barthe, P. Dybjer, L. Pinto, J. Saraiva, eds., *Advanced Lectures from Int. Summer School on Applied Semantics, APPSEM 2000*, v. 2395 of *Lect. Notes in Comput. Sci.*, pp. 42–122. Springer-Verlag, Berlin, 2002.
6. S. Brookes, S. Geva. Computational comonads and intensional semantics. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, eds., *Applications of Categories in Computer Science*, v. 177 of *London Math. Society Lecture Note Series*, pp. 1–44. Cambridge Univ. Press, Cambridge, 1992.
7. P. Caspi. Clocks in dataflow languages. *Theoret. Comput. Sci.*, 94(1), pp. 125–140, 1992.
8. P. Caspi, M. Pouzet. Synchronous Kahn networks. In *Proc. of 1st ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'96*, pp. 226–238. ACM Press, New York, 1996. Also in *SIGPLAN Notices*, 31(6), pp. 226–238, 1996.
9. P. Caspi, M. Pouzet. A co-iterative characterization of synchronous stream functions. In B. Jacobs, L. Moss, H. Reichel, J. Rutten, eds., *Proc. of 1st Wksh. on Coalgebraic Methods in Computer Science, CMCS'98*, v. 11 of *Electron. Notes in Theoret. Comput. Sci.*. Elsevier, Amsterdam, 1998.
10. J.-L. Colaço, A. Girault, G. Hamon, M. Pouzet. Towards a higher-order synchronous data-flow language. In *Proc. of 4th ACM Int. Conf. on Embedded Software, EMSOFT'04*, pp. 230–239. ACM Press, New York, 2004.
11. J.-L. Colaço, M. Pouzet. Clocks and first class abstract types. In R. Alur, I. Lee, eds., *Proc. of 3rd Int. Conf. on Embedded Software, EMSOFT 2003*, v. 2855 of *Lect. Notes in Comput. Sci.*, pp. 134–155. Springer-Verlag, Berlin, 2003.

12. R. Davies, F. Pfenning. A modal analysis of staged computation. *J. of ACM*, 48(3), pp. 555-604, 2001.
13. N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. The synchronous data flow programming language LUSTRE. *Proc. of the IEEE*, 79(9), pp. 1305–1320, 1991.
14. P. Hudak, A. Courtney, H. Nilsson, J. Peterson. Arrows, robots, and functional programming. In J. Jeuring, S. Peyton Jones, eds., *Revised Lectures from 4th Int. Summer School on Advanced Functional Programming, AFP 2004*, v. 2638 of *Lect. Notes in Comput. Sci.*, pp. 159–187. Springer-Verlag, Berlin, 2003.
15. J. Hughes. Generalising monads to arrows. *Sci. of Comput. Program.*, 37(1–3), pp. 67–111, 2000.
16. J. Hughes. Programming with arrows. In V. Vene, T. Uustalu, eds., *Revised Lectures from 5th Int. Summer School on Advanced Functional Programming, AFP 2004*, v. 3622 of *Lect. Notes in Comput. Sci.*. Springer-Verlag, Berlin, to appear.
17. R. B. Kieburtz. Codata and comonads in Haskell. Unpublished manuscript, 1999.
18. R. B. Kieburtz. Coalgebraic techniques for reactive functional programming, In *Actes 11mes Journées Francophones des Langages Applicatifs, JFLA 2000*, pp. 131–157. 2000.
19. J. R. Lewis, M. B. Shields, E. Meijer, J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *Proc. of 27th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'00*, pp. 108–118. ACM Press, New York, 2000.
20. E. Moggi. Notions of computation and monads. *Inform. and Comput.*, 93(1), pp. 55–92, 1991.
21. H. Nilsson, A. Courtney, J. Peterson. Functional reactive programming, continued. In *Proc. of 2002 ACM SIGPLAN Wksh. on Haskell, Haskell'02*, pp. 51–64. ACM Press, New York, 2002.
22. A. Pardo. Generic accumulations. J. Gibbons, J. Jeuring, eds., *Proc. of IFIP TC2/WG2.1 Working Conference on Generic Programming*, v. 243 of *IFIP Conf. Proc.*, pp. 49–78. Kluwer, 2003.
23. R. Paterson. A new notation for arrows. In *Proc. of 6th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'01*, ACM Press, New York, pp. 229–240, 2001. Also in *SIGPLAN Notices*, 36(10), pp. 229–240, 2001.
24. R. Paterson. Arrows and computation. In J. Gibbons, O. de Moor, eds., *The Fun of Programming*, *Cornerstones of Computing*, pp. 201–222. Palgrave Macillan, Basingstoke / New York, 2003.
25. M. Pouzet. Lucid Synchrone: tutorial and reference manual. Unpublished manuscript, 2001.
26. J. Power, E. Robinson. Premonoidal categories and notions of computation. *Math. Structures in Comput. Sci.*, 7(5), pp. 453–468, 1997.
27. J. Power, H. Watanabe. Combining a monad and a comonad. *Theoret. Comput. Sci.*, 280(1–2), pp. 137–162, 2002.
28. J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoret. Comput. Sci.*, 249(1), pp. 3–80, 2000.
29. J. J. M. M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoret. Comput. Sci.*, 308(1–3), pp. 1–53, 2003.
30. T. Uustalu, V. Vene. The dual of substitution is redecoration. In K. Hammond, S. Curtis (Eds.), *Trends in Functional Programming 3*, pp. 99–110. Intellect, Bristol / Portland, OR, 2002.

31. T. Uustalu, V. Vene. Signals and comonads. In M. A. Musicante, R. M. F. Lima, ed., *Proc. of 9th Brazilian Symp. on Programming Languages, SBLP 2005*, pp. 215–228. Univ. de Pernambuco, 2005.
32. T. Uustalu, V. Vene, A. Pardo. Recursion schemes from comonads. *Nordic J. of Computing*, 8(3), pp. 366–390, 2001.
33. P. Wadler. The essence of functional programming. In *Conf. Record of 19th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'92*, pp. 1–14. ACM Press, New York, 1992.