The Essence of Dataflow Programming

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

Shall we be pure or impure? Today we shall be very pure. It must always be possible to contain impurities (i.e., non-functionalities), in a pure (i.e., functional) way.

The program

```
fact x  = if x <= 1 then 1 else fact (x - 1) * x
```

for factorial encodes a *pure function*.

The programs

```
factM x = (if x == 5 then raise else
             if x <= 1 then 1 else factM (x - 1) * x)
          `handle` (if x == 7 then 5040 else raise)
```

and

```
factL x = if x <= 1 then 1 else factL (x - 1) * (1 `choice` x)
```

represent "lossy" versions of the factorial function. The first yields an error on 5 and 6 whereas the second can fail to do some of the multiplications required for the normal factorial. These *impure* "functions" can be made sense of in the paradigms of *error raising/handling* and *non-deterministic* computations. Ever since the work by Moggi and Wadler [26,40,41], 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 [17] or Lucid Synchrone [11,31], the newer *French synchronous dataflow* languages. fby reads 'followed by' and means initialized unit delay of a discrete-time signal (cons of a stream).

Could it be that monads are capable of structuring notions of dataflow computation as well? No, there are simple reasons why this must be impossible. (We will discuss these.) As a substitute for monads, Hughes has therefore proposed a laxer framework that he has termed *arrow types* [19] (and Power et al. [32] proposed the same under the name of *Freyd categories*). But this is—we assert—overkill, at least as long as we are interested in dataflow computation. It turns out that something simpler and more standard, namely *comonads*, the dual of monads, does just as well. In fact, comonads are even better, as there is more structure to comonads than to arrow types. Arrow types are too general.

The message of this paper is just this last point: While notions of dataflow computation cannot be structured with monads, they can be structured perfectly with comonads. And more generally, comonads have received too little attention in programming language semantics compared to monads. Just as monads are good 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 [8] and Kieburtz [23], 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.

The paper contributes a novel approach to dataflow computation based on comonads. We show that general and causal stream functions, the basic entities in intensional resp. synchronous dataflow computation, are elegantly described in terms of comonads. Imitating monadic interpretation, we develop a generic comonadic interpreter. By instantiation, we obtain interpreters of a Lucid-like intensional language and a Lucid Synchrone-like synchronous dataflow language. Remarkably, we get elegant 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 give a short introduction to dataflow programming. In Section 3, we give a brief review of the Moggi-Wadler monad-based approach to programming with effect-producing functions in a pure language and to the semantics of corresponding impure languages. In particular, we recall monadic interpretation. In Section 4, we show that certain paradigms of computation, notably stream functions, do not fit into this framework, and introduce the substitute idea of arrow types/Freyd categories. In Section 5, 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 6, 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 7 is a summary of related work, while Section 8 lists our conclusions.

We assume that the reader is familiar with the basics of functional programming (in particular, Haskell programming) and denotational semantics and also knows about the Lambek-Lawvere correspondence between typed lambda calculi and cartesian closed categories (the types-as-objects, terms-as-morphisms correspondence). The paper contains a brief introduction to dataflow programming, but acquaintance with languages such as Lucid and Lustre or Lucid Synchrone will be of additional help. Concepts such as monads, comonads etc. are defined in the paper.

The paper is related to our earlier paper [37], which discussed comonad-based dataflow programming, but did not treat comonad-based processing of dataflow languages. A short version of the present paper (without introductions to dataflow languages, monads, monadic interpretation and arrows) appeared as [38].

2 Dataflow Programming

We begin with an informal quick introduction to dataflow programming as supported by languages of the Lucid family [2] and the Lustre and Lucid Synchrone languages [11,31]. We demonstrate a neutral syntax which we will use throughout the paper.

Dataflow programming is about programming with streams, thought about as signals in discrete time. The style of programming is functional, but any expression denotes a stream (a signal), or more exactly, the element of a stream at an understood position (the value of a signal the time instant understood as the present). Since the position is not mentioned, the stream is defined uniformly across all of its positions. Compare this to physics, where many quantities vary in time, but the time argument is always kept implicit and there is never any explicit dependency on its value.

All standard operations on basic types are understood pointwise (so in particular constants become constant streams). The if-construct is also understood pointwise.

x	x_0	x_1	x_2	x_3	x_4	x_5	...
y	y_0	y_1	y_2	y_3	y_4	y_5	...
x + y	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$	...
z	t	f	t	t	f	t	...
if z then x else y	x_0	y_1	x_2	x_3	y_4	x_5	...

If we had product types, the projections and the pairing construct would also be pointwise. With function spaces, it is not obvious what the design should be and we will not discuss any options at this stage. As a matter of fact, most dataflow languages are first-order: expressions with variables are of course allowed, but there are no first-class functions.

With the pointwise machinery, the current value of an expression is always determined by the current values of its variables. This is not really interesting. We should at least allow dependencies on the past values of the variables. This is offered by a construct known as `fby` (pronounced "followed by"). The expression `e0 fby e1` takes the initial value of `e0` at the beginning of the history, and at every other instant of time it takes the value that `e1` had at the immediately preceding instant. In other words, the signal `e0 fby e1` is the unit delay of the signal `e1`, initialized with the initial value of `e0`.

x	x_0	x_1	x_2	x_3	x_4	x_5	...
y	y_0	y_1	y_2	y_3	y_4	y_5	...
x fby y	x_0	y_0	y_1	y_2	y_3	y_4	...

With the `fby` operator, one can write many useful recursive definitions where the recursive calls are guarded by `fby` and there is no real circularity. Below are some classic examples of such feedback through a delay.

```
pos    = 0 fby pos + 1
sum x  = x + (0 fby sum x)
diff x = x - (0 fby x)
ini x  = x fby ini x
fact   = 1 fby (fact * (pos + 1))
fibo   = 0 fby (fibo + (1 fby fibo))
```

The value of `pos` is 0 at the beginning of the history and at every other instant it is the immediately preceding value incremented by one, i.e., `pos` generates the enumeration of all natural numbers. The function `sum` finds the accumulated sum of all values of the input up to the current instant. The function `diff` finds the difference between the current value and the immediately preceding value of the input. The function `ini` generates the constant sequence of the initial value of the input. Finally, `fact` and `fibo` generate the graphs of the factorial and Fibonacci functions respectively. Their behaviour is illustrated below.

`pos`	0	1	2	3	4	5	6	...
`sum pos`	0	1	3	6	10	15	21	...
`diff pos`	0	1	1	1	1	1	1	...
`ini pos`	0	0	0	0	0	0	0	...
`fact`	1	1	2	6	24	120	720	...
`fibo`	0	1	1	2	3	5	8	...

An expression written with pointwise constructs and `fby` is always causal in the sense that its present value can only depend on the past and present values of its variables. In languages à la Lucid, one can also write more general expressions with physically unrealistic dependencies on future values of the variables. This is supported by a construct called `next`. The value of `next e` at the current instant is the value of `e` at the immediately following instant, so the signal `next e` is the unit anticipation of the signal `e`.

`x`	x_0	x_1	x_2	x_3	x_4	x_5	...
`next x`	x_1	x_2	x_3	x_4	x_5	x_6	...

Combining `next` with recursion, it is possible to define functions whose present output value can depend on the value of the input in unboundedly distant future. For instance, the sieve of Eratosthenes can be defined as follows.

```
x wvr y = if ini y then x fby (next x wvr next y)
                   else (next x wvr next y)

sieve x = x fby sieve (x wvr x mod (ini x) /= 0)
eratosthenes = sieve (pos + 2)
```

The filtering function `wvr` (pronounced "whenever") returns the substream of the first input stream consisting of its elements from the positions where the second input stream is true-valued. (This is all well as long as there always is a future position where the second input stream has the value true, but poses a problem, if from some point on it is constantly false.) The function `sieve` outputs the initial element of the input stream and then recursively calls itself on the substream of the input stream that only contains the elements not divisible by the initial element.

`x`	x_0	x_1	x_2	x_3	x_4	x_5	...
`y`	t	f	t	t	f	t	...
`x wvr y`	x_0	x_2	x_3	x_5	...		
`pos` $+ 2$	2	3	4	5	6	7	...
`eratosthenes`	2	3	5	7	11	13	...

Because anticipation is physically unimplementable and the use of it may result in unbounded lookaheads, most dataflow languages do not support it. Instead, some of them provide means to define partial streams, i.e., streams where some elements can be undefined (denoted below by $-$). The idea is that different signals may be on different clocks. Viewed as signals on the fastest (base) clock, they are not defined at every instant. They are only defined at those instants of the base clock that are also instants of their own clocks.

One possibility to specify partial streams is to introduce new constructs `nosig` and `merge` (also known as "default"). The constant `nosig` denotes a constantly undefined stream. The operator `merge` combines two partial streams into a partial stream that is defined at the positions where at least one of two given partial streams is defined (where both are defined, there the first one takes precedence).

`nosig`	–	–	–	–	–	–	...
`x`	x_0	–	–	x_3	–	–	...
`y`	–	–	y_2	y_3	–	y_5	...
`merge x y`	x_0	–	y_2	x_3	–	y_5	...

With the feature of partiality, it is possible to define the sieve of Eratosthenes without anticipation.

```
sieve x = if (tt fby ff) then x
          else sieve (if (x mod ini x /= 0) then x else nosig)
eratosthenes = sieve (pos + 2)
```

The initial element of the result of `sieve` is the initial element of the input stream whereas all other elements are given by a recursive call on the modified version of the input stream where all positions containing elements divisible by the initial element have been dropped.

`pos` $+ 2$	2	3	4	5	6	7	8	9	10	11	...
`eratosthenes`	2	3	–	5	–	7	–	–	–	11	...

3 Monads and Monadic Interpreters

3.1 Monads and Effect-Producing Functions

Now we proceed to monads and monadic interpreters. We begin with a brief recapitulation of the monad-based approach to representing effectful functions [26,40,41,6].

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 $f : A \to TB$, $k^\star \circ \eta_A = k$,
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.

- $TA = A$, the identity monad,
- $TA = \mathsf{Maybe}\, A = A + 1$, error (undefinedness), $TA = A + E$, exceptions,
- $TA = \mathsf{List}\, A = \mu X.1 + A \times X$, non-determinism,
- $TA = E \Rightarrow A$, readable environment,
- $TA = S \Rightarrow A \times S$, 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 [1].

(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)
```

return is the Haskell name for the unit and ($\gg=$) (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 identity monad is Haskell-implemented as follows.

```
newtype Id a = Id a

instance Monad Id where
  return a    = Id a
  Id a  >>= k = k a
```

The definitions of the maybe and list monads are the following.

```
data Maybe a = Just a | Nothing

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

data [a] = [] | a : [a]

instance Monad [] where
  return a       = [a]
  []       >>= k = []
  (a : as) >>= k = k a ++ (as >>= k)
```

The exponent and state monads are defined in the following fashion.

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

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

newtype State s a = State (s -> (a, s))

instance Monad (State s) where
  return a      = State (\ s -> (a, s))
  State f >>= k = State (\ s -> case f s of
                                   (a, s') -> case k a of
                                      State f' -> f' s')
```

In the case of these monads, the operations specific to the type of effect they characterize are raising and handling an error, nullary and binary nondeterministic choice, consulting and local modification of the environment, consulting and updating the state.

```
raise :: Maybe a
raise = Nothing

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

```
deadlock :: [a]
deadlock = []

choice :: [a] -> [a] -> [a]
choice as0 as1 = as0 ++ as1

askE :: Exp e e
askE = Exp id

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

get :: State s s
get = State (\ s -> (s, s))

put :: s -> State s ()
put s = State (\ _ -> ((), s))
```

3.2 Monadic Semantics

Monads are a perfect tool for formulating denotational semantics of languages for programming effectful functions. If this is done in a functional (meta-)language, one obtains a reference interpreter for free. Let us recall how this project was carried out by Moggi and Wadler. Of course we choose Haskell as our metalanguage.

We proceed from a simple strongly typed purely functional (object) language with two base types, integers and booleans, which we want to be able to extend with various types of effects. As particular examples of types of effects, we will consider errors and non-determinism.

The first thing to do is to define the syntax of the object language. Since Haskell gives us no support for extensible variants, it is simplest for us to include the constructs for the two example effects from the beginning. For errors, these are error raising and handling. For non-determinism, we consider nullary and binary branching.

```
type Var = String

data Tm = V Var | L Var Tm | Tm :@ Tm
        | N Integer | Tm :+ Tm | ...
        | Tm :== Tm | ... | TT | FF | Not Tm | ... | If Tm Tm Tm
        -- specific for Maybe
        | Error | Tm `Handle` Tm
        -- specific for []
        | Deadlock | Tm `Choice` Tm
```

In the definition above, the constructors V, L, (:@) correspond to variables, lambda-abstraction and application. The other names should be self-explanatory.

Next we have to define the semantic domains. Since Haskell is not dependently typed, we have to be a bit coarse here, collecting the semantic values of all object language types (for one particular type of effect) into a single type. But in reality, the semantic values of the different object language types are integers, booleans and functions, respectively, with no confusion. Importantly, a function takes a value to a value with an effect (where the effect can only be trivial in the pure case). An environment is a list of variable-value pairs, where the first occurrence of a variable in a pair in the list determines its value.

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

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

We will manipulate environment-like entities via the following three functions. (The safe lookup, that maybe returns a value, will be unnecessary, since we can type-check an object-language term before evaluating it. If this succeeds, we can be sure we will only be looking up variables in environments where they really occur.)

```
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
```

The syntax and the semantic domains of the possible object languages described, we can proceed to evaluation.

The pure core of an object language is interpreted uniformly in the type of effect that this language supports. Only the unit and bind operations of the corresponding monad have to be known to describe the meanings of the core constructs.

```
class Monad t => MonadEv t where
  ev :: Tm -> Env t -> t (Val t)

evClosed :: MonadEv t => Tm -> t (Val t)
evClosed e = ev e empty

_ev :: MonadEv t => Tm -> Env t -> t (Val t)
_ev (V x)        env = return (unsafeLookup x env)
_ev (L x e)      env = return (F (\ a -> ev e (update x a env)))
_ev (e :@ e')    env = ev e  env >>= \ (F k)  ->
                       ev e' env >>= \ a      ->
                       k a
_ev (N n)        env = return (I n)
_ev (e0 :+ e1)   env = ev e0 env >>= \ (I n0) ->
```

```
                       ev e1 env >>= \ (I n1) ->
                       return (I (n0 + n1))
...
_ev TT           env = return (B True )
_ev FF           env = return (B False)
_ev (Not e)      env = ev e  env >>= \ (B b)  ->
                       return (B (not b))
...
_ev (If e e0 e1) env = ev e  env >>= \ (B b)  ->
                       if b then ev e0 env else ev e1 env
```

To interpret the "native" constructs in each of the extensions, we have to use the "native" operations of the corresponding monad.

```
instance MonadEv Id where
  ev e env = _ev e env

instance MonadEv Maybe where
  ev Raise             env = raise
  ev (e0 `Handle` e1) env = ev e0 env `handle` ev e1 env
  ev e                 env = _ev e env

instance MonadEv [] where
  ev Deadlock          env = deadlock
  ev (e0 `Choice` e1) env = ev e0 env `choice` ev e1 env
  ev e                 env = _ev e env
```

We have achieved nearly perfect reference interpreters for the three languages. But there is one thing we have forgotten. To accomplish anything really interesting with integers, we need some form of recursion, say, the luxury of general recursion. So we would actually like to extend the definition of the syntax by the clause

```
data Tm = ... | Rec Tm
```

It would first look natural to extend the definition of the semantic intepretation by the clause

```
_ev (Rec e) env = ev  e       env >>= \ (F k) ->
                  _ev (Rec e) env >>= \ a     ->
                  k a
```

But unfortunately, this interprets Rec too eagerly, so no recursion will ever stop. For every recursive call in a recursion, the interpreter would want to know if it returns, even if the result is not needed at all.

So we have a problem. The solution is to use the MonadFix class (from Control.Monad.Fix), an invention of Erkök and Launchbury [16], which specifically supports the monadic form of general recursion [3]:

[3] Notice that 'Fix' in 'MonadFix' refers as much to fixing an unpleasant issue as it refers to a fixpoint combinator.

```
class Monad t => MonadFix t where
  mfix :: (a -> t a) -> t a

  -- the ideal uniform mfix which doesn't work
  -- mfix k = mfix k >>= k
```

The identity, maybe and list monads are instances (in an ad hoc way).

```
fix :: (a -> a) -> a
fix f = f (fix f)

instance MonadFix Id where
  mfix k = fix (k . unId)
           where unId (Id a) = a

instance MonadFix Maybe where
  mfix k = fix (k . unJust)
           where unJust (Just a) = a

instance MonadFix [] where
  mfix k = case fix (k . head) of
             []      -> []
             (a : _) -> a : mfix (tail . k)
```

Now, after subclassing MonadEv from MonadFix instead of Monad

```
class MonadFix t => MonadEv t where ...
```

we can define the meaning of Rec by the clause

```
_ev (Rec e) env = ev e  env >>= \ (F k) ->
                  mfix k
```

After this dirty fix (where however all dirt is contained) everything is clean and working. We can interpret our pure core language and the two extensions. The examples from the Introduction are handled by the interpreter exactly as expected. We can define:

```
fact  = Rec (L "fact" (L "x" (
                  If (V "x" :<= N 1)
                     (N 1)
                     ((V "fact" :@ (V "x" :- N 1)) :* V "x"))))

factM = Rec (L "fact" (L "x" (
               (If (V "x" :== N 5)
                  Raise
                  (If (V "x" :<= N 1)
                     (N 1)
                     ((V "fact" :@ (V "x" :- N 1)) :* V "x")))
               `Handle`
```

```
                (If (V "x" :== N 7)
                    (N 5040)
                    Raise))))

factL = Rec (L "fact" (L "x" (
                If (V "x" :<= N 1)
                   (N 1)
                   ((V "fact" :@ (V "x" :- N 1)) :*
                                                (N 1 `Choice` V "x")))))
```

Testing these, we get exactly the results we would expect.

```
> evClosed (fact  :@ N 6) :: Id (Val Id)
Id 720
> evClosed (factM :@ N 4) :: Maybe (Val Maybe)
Just 24
> evClosed (factM :@ N 6) :: Maybe (Val Maybe)
Nothing
> evClosed (factM :@ N 8) :: Maybe (Val Maybe)
Just 40320
> evClosed (factL :@ N 5) :: [Val []]
[1,5,4,20,3,15,12,60,2,10,8,40,6,30,24,120]
```

4 Arrows

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 [19] has been promoting what he has called arrow types to overcome this deficiency. In semantics, the same concept was invented for the same reason by Power and Robinson [32] under the name of a Freyd category.

Informally, a Freyd category is a symmetric premonoidal category together and 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 $\mathcal{K}$ 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 for every map $g : C \to D$ and so do the two composites $C \otimes A \to D \otimes B$. A natural transformation is called 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 pragmatics for impure computation is to have an inclusion from the base category of pure functions to a richer category of which is the home for impure functions (arrows), so that some aspects of the Cartesian structure of the base category are preserved. Importantly the Cartesian product $\times$ of $\mathcal{C}$ is bifunctorial, so $(B \times g) \circ (f \times C) = (f \times D) \circ (A \times g) : A \times C \to B \times D$ for any $f : A \to B$ and $g : C \to D$, but for the corresponding tensor operation $\oplus$ of $\mathcal{K}$ this is only mandatory if either f or g is pure (the idea being that different sequencings of impure functions must be able to give different results).

The basic example of a Freyd category is the Kleisli category of a strong monad. Another standard one is that of stateful functions. For a base category $\mathcal{C}$, the maps of the Freyd category are the maps $A \times S \to B \times S$ of $\mathcal{C}$ where S is some fixed object of $\mathcal{C}$. This is not very exciting, since if $\mathcal{C}$ also has exponents, the maps $A \times S \to B \times S$ are in a natural bijection with the maps $A \to S \Rightarrow B \times S$, which means that the Freyd category is essentially the same as the Kleisli category of the state monad. But probably the best known and most useful example is that of stream functions. In this case the maps $A \to B$ of the Freyd category are the maps $\mathsf{Str}A \to \mathsf{Str}B$ of $\mathcal{C}$ of $\mathcal{C}$ where $\mathsf{Str}A = \nu X.A \times X$ is the type of streams over the type A. Notice that differently from stateful functions from A to B, stream functions from A to B just cannot be viewed as Kleisli arrows.

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 of monads 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)
```

Stateful functions are a particularly simple instance.

```
newtype StateA s a b = StateA ((a, s) -> (b, s))

instance Arrow (State A s) where
  pure f = StateA (\ (a, s) -> (f a, s))
  StateA f >>> StateA g = StateA (g . f)
  first (StateA f) = StateA (\ ((a, c), s) -> case f (a, s) of
                                                  (b, s') -> ((b, c), s'))
```

Stream functions are declared to be arrows in the following fashion, relying on streams being mappable and zippable. (For reasons of readability that will become apparent in the next section, 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 because of its algebraically compact semantics, we want to make the distinction, as our work also applies to other, finer semantic models.)

```
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)
```

5 Comonads

5.1 Comonads and Context-Dependent Functions

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 [8,23,25], but there have been few useful examples. One of the goals of this paper is to show that general 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-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 $\varepsilon_A : DA \to A$ 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:

- $DA = A$, 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 [36].

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[4]. But of course comonads are easily defined as a type constructor class analogously to monads.

```
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.

```
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
```

Just as the Kleisli categories of strong monads are Freyd categories, so are the coKleisli categories of comonads.

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

pair f g x = (f x, g x)

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))
```

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

5.2 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:

```
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
      where 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 [23]) 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)
       where 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 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')
                 where 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 FunArg Nat and LVS.

```
fbyFA :: a -> (FunArg Int a -> a)
fbyFA a0 (f :# 0)       = a0
fbyFA _  (f :# (i + 1)) = f i

fbyLVS :: a -> (LVS a -> a)
fbyLVS a0 (Nil       := _ :| _) = a0
fbyLVS _  ((_ :> a') := _ :| _) = a'

nextFA :: FunArg Int a -> a
nextFA (f :# i) = f (i + 1)

nextLVS :: LVS a -> a
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[5]. 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, this can be found in [36]). 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')
                 where 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'
```

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. As $A \times \mathsf{Str}\,A \cong \mathsf{Str}\,A$, 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!

[5] The standard terminology is '*synchronous* stream functions', but want to avoid it, because 'synchrony' also refers to all signals being on the same clock and to the hypothesis on which the applications of synchronous dataflow languages are based: that in an embedded system the controller can react to an event in the plant in so little time that it can be considered instantaneous.

5.3 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.

As the first thing we again 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 as before, but now our 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)]
```

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

```
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 (and, for recursion, the obvious solution works). 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 an additional zipping operation. We define a *comonad with zipping* to be a comonad D coming with a natural transformation m with components $m_{A,B} : DA \times DB \to D(A \times B)$ that satisfies coherence conditions such as $\varepsilon_{A\times B} \circ m_{A,B} = \varepsilon_A \times \varepsilon_B$ (more mathematically, this is a symmetric semi-monoidal comonad).

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).

```
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[6].

```
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
```

[6] The name ‘repair’ in the code below alludes both to getting a small discrepancy in the types right and to rearranging some pairings.

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 the examples from Section 2. 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 :< ...
> runLV (ev (diff :@ (sum :@ pos))) emptyS
0 :< (1 :< (2 :< (3 :< (4 :< (5 :< (6 :< (7 :< (8 :< (9 :< (10 :< ...
```

The 'whenever' operator and the sieve of Eratosthenes, which use anticipation, are only allowed with general stream functions.

```
-- 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) /= 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 :< ...
```

6 Distributive Laws

6.1 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 [8,33]. 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 $(D, \varepsilon, -^\dagger)$ 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. 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)

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 ->
                          k (cmap fst d) >>= \ b ->
                          return (b, snd (counit d)))
```

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.

```
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'
                 where 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'
```

6.2 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)

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

The meanings 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
```

Similarly to the case with the monadic interpreter, the clause for of Rec in the above code this does not quite work, because recursive calls get evaluated too eagerly, but the situation can be remedied by introducing a type constructor class DistCheat of which LV with Maybe will be 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
```

The partial, causal version of the sieve of Eratosthenes from Section 2 is defined as follows.

```
-- 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) (cmap Just emptyS)
Just 2 :< (Just 3 :< (Nothing :< (Just 5 :< (Nothing :< (Just 7 :< (
Nothing :< (Nothing :< (Nothing :< (Just 11 :< (Nothing :< (Just 13 :< (
Nothing :< (Nothing :< (Nothing :< (Just 17 :< ...
```

7 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 [10,11,14]. Relevantly for us, however, Colaço et al. [13] have very recently proposed a higher-order synchronous dataflow language extending Lucid Synchrone, with two type constructors of function spaces.

Hughes's arrows [19] have been picked up very well by the functional programming community (for overviews, see [30,20]). There exists by now not only a de facto standardized arrow library in Haskell, but even specialized syntax [29]. The main application is functional reactive programming with its specializations to animation, robotics etc. [27,18]. Functional reactive programming is of course the same as dataflow programming, except that it is done by functional programmers rather than the traditional dataflow languages community. The exact relationship between Hughes's arrows and Power and Robinson's symmetric premonoidal categories has been established recently by Jacobs and colleagues [21,22].

Uses of comonads in semantics have been very few. Brookes and Geva [8] were the first to suggest to exploit comonads in semantics. They realized that, in order for the coKleisli category of a comonad to have exponential-like objects, the comonad has to come with a zip-like operation (they called it "merge"), but did not formulate the axioms of a symmetric monoidal comonad. Kieburtz [23] made an attempt to draw the attention of functional programmers to comonads. Lewis et al. [25] must have contemplated employing the product comonad to handle implicit parameters (see the conclusion of their paper), but did not carry out the project. Comonads have also been used in the semantics of intuitionistic linear logic and modal logics [5,7], with their applications in staged computation and elsewhere, see e.g., [15], and to analyse structured recursion schemes, see e.g., [39,28,9]. In the semantics of intuitionistic linear and modal logics, comonads are strong symmetric monoidal.

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 languages have produced a number of papers exploiting the final coalgebraic structure of streams [12,24,4], but apparently nothing on stream functions and comonads. The same is true about works in universal coalgebra [34,35].

8 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 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, G. Bierman, V. de Paiva, M. Hyland. Linear lambda-calculus and categorical models revisited. In E. Börger et al., eds, *Proc. of 6th Wksh. on Computer Science Logic, CSL '92*, v. 702 of *Lect. Notes in Comput. Sci.*, pp. 61–84. Springer-Verlag, Berlin, 1993.
6. 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.
7. G. Bierman, V. de Paiva. On an intuitionistic modal logic. *Studia Logica*, 65(3), pp. 383–416, 2000.
8. 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.

9. V. Capretta, T. Uustalu, V. Vene. Recursive coalgebras from comonads. *Inform. and Comput.*, 204(4), pp. 437–468, 2006.
10. P. Caspi. Clocks in dataflow languages. *Theoret. Comput. Sci.*, 94(1), pp. 125–140, 1992.
11. 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 *ACM SIGPLAN Notices*, 31(6), pp. 226–238, 1996.
12. 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.
13. 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.
14. 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.
15. R. Davies, F. Pfenning. A modal analysis of staged computation. *J. of ACM*, 48(3), pp. 555-604, 2001.
16. L. Erkök, J. Launchbury. Monadic recursive bindings. In *Proc. of 5th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'00*, pp. 174–185. ACM Press, New York, 2000. Also in *ACM SIGPLAN Notices*, 35(9), pp. 174–185, 2000.
17. 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.
18. 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. School on Advanced Functional Programming, AFP 2002*, v. 2638 of *Lect. Notes in Comput. Sci.*, pp. 159–187. Springer-Verlag, Berlin, 2003.
19. J. Hughes. Generalising monads to arrows. *Sci. of Comput. Program.*, 37(1–3), pp. 67–111, 2000.
20. J. Hughes. Programming with arrows. In V. Vene, T. Uustalu, eds., *Revised Lectures from 5th Int. School on Advanced Functional Programming, AFP 2004*, v. 3622 of *Lect. Notes in Comput. Sci.*, pp. 73–129. Springer-Verlag, Berlin, 2005.
21. C. Heunen, B. Jacobs. Arrows, like monads, are monoids. In S. Brookes, M. Mislove, eds., *Proc. of 22nd Ann. Conf. on Mathematical Foundations of Programming Semantics, MFPS XXII*, v. 158 of *Electron. Notes in Theoret. Comput. Sci.*, pp. 219–236. Elsevier, Amsterdam, 2006.
22. B. Jacobs, I. Hasuo. Freyd is Kleisli, for arrows. In C. McBride, T. Uustalu, *Proc. of Wksh. on Mathematically Structured Programming, MSFP 2006*, *Electron. Wkshs. in Computing*. BCS, 2006.
23. R. B. Kieburtz. Codata and comonads in Haskell. Unpublished manuscript, 1999.
24. R. B. Kieburtz. Coalgebraic techniques for reactive functional programming, In *Actes 11mes Journées Francophones des Langages Applicatifs, JFLA 2000*, pp. 131–157. 2000.
25. 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.
26. E. Moggi. Notions of computation and monads. *Inform. and Comput.*, 93(1), pp. 55–92, 1991.

27. 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.
28. A. Pardo. Generic accumulations. J. Gibbons, J. Jeuring, eds., *Proc. of IFIP TC2/WG2.1 Working Conf. on Generic Programming*, v. 243 of *IFIP Conf. Proc.*, pp. 49–78. Kluwer, Dordrecht, 2003.
29. 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 *ACM SIGPLAN Notices*, 36(10), pp. 229–240, 2001.
30. R. Paterson. Arrows and computation. In J. Gibbons, O. de Moor, eds., *The Fun of Programming*, *Cornerstones of Computing*, pp. 201–222. Palgrave MacMillan, Basingstoke / New York, 2003.
31. M. Pouzet. Lucid Synchrone: tutorial and reference manual. Unpublished manuscript, 2001.
32. J. Power, E. Robinson. Premonoidal categories and notions of computation. *Math. Structures in Comput. Sci.*, 7(5), pp. 453–468, 1997.
33. J. Power, H. Watanabe. Combining a monad and a comonad. *Theoret. Comput. Sci.*, 280(1–2), pp. 137–162, 2002.
34. J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoret. Comput. Sci.*, 249(1), pp. 3–80, 2000.
35. 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.
36. 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.
37. T. Uustalu, V. Vene. Signals and comonads. *J. of Univ. Comput. Sci.*, 11(7), pp. 1310-1326, 2005.
38. T. Uustalu, V. Vene. The essence of dataflow programming (short version). In K. Yi, ed., *Proc. of 3rd Asian Symp. on Programming Languages and Systems, APLAS 2005*, v. 3780 of *Lect. Notes in Comput. Sci.*, pp. 2–18. Springer-Verlag, Berlin, 2005.
39. T. Uustalu, V. Vene, A. Pardo. Recursion schemes from comonads. *Nordic J. of Computing*, 8(3), pp. 366–390, 2001.
40. 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.
41. P. Wadler. Monads for functional programming. In M. Broy, ed., *Program Design Calculi: Proc. of Marktoberdorf Summer School 1992*, v. 118 of *NATO ASI Series F*, pp. 233–264. Springer-Verlag, Berlin, 1993. Also in J. Jeuring, E. Meijer, eds., *Tutorial Text from 1st Int. Spring School on Advanced Functional Programming Techniques AFP '95*, v. 925 of *Lect. Notes in Comput. Sci.*, pp. 24–52. Springer-Verlag, Berlin, 1995.