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 struc-

ture 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

Z. Horváth (Ed.): CEFP 2005, LNCS 4164, pp. 135–167, 2006.

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 func-

tion. 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 cen-

tral one for programming and semantics is that any monad $(T, \eta, -^*)$ defines a category $\mathcal{C}_T$ where $|\mathcal{C}_T| = |\mathcal{C}|$ and $\mathcal{C}_T(A, B) = \mathcal{C}(A, TB)$, $(\mathsf{id}_T)_A = \eta_A$, $\ell \circ_T k = \ell^* \circ k$ (*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 (≫=) (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 [1]:

```
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).

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

```
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"
                    `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)) :* 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

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)
```

-- coinductive