The essence of dataflow programming

Tarmo Uustalu[1]   Varmo Vene[2]

[1]Institute of Cybernetics
Tallinn University of Technology

[2]Department of Computer Science
University of Tartu

WG2.8 workshop, Kalvi, 1-4 September 2005

Motivation

- Moggi and Wadler showed that effectful computations can be structured with monads.
- An effect-producing function from A to B is a map $A \to B$ in the Kleisli category, i.e., a map $A \to TB$ in the base category.
- Some examples applied in semantics:
  - $TA = A$, the identity monad,
  - $TA = \text{Maybe}A = A + 1$, error (partiality)
  - $TA = A + E$, exceptions,
  - $TA = E \Rightarrow A$, environment,
  - $TA = \text{List}A = \mu X.1 + A \times X$, non-determinism,
  - $TA = S \Rightarrow A \times S$, state,
  - $TA = (A \Rightarrow R) \Rightarrow R$, continuations.

Motivation

- However, there are several impure features which are not captured by monads.
- But what about dataflow languages such as Lucid (general stream functions) or Lustre or Lucid Synchrone (causal stream functions)?
- Hughes proposed arrow types (also known under the name of *Freyd categories*) as a means to structure stream-based computations.
- But what about comonads? They have not found extensive use (some examples by Brookes and Geva, Kieburtz, but mostly artificial).

This talk

- Comonads provide the right level of abstraction to organize dataflow computation.
- In particular, we can define a generic comonadic semantics for dataflow(ish) languages, similar to Moggi's generic monadic semantics of languages for computation with effects.

Outline

1. Monadic Interpreters
2. Comonadic Interpreters
   - Comonads
   - Comonads for Stream Functions
   - Comonadic Interpreters
3. Distributive Interpreters
   - Distributive Laws
   - Distributive Interpreters
4. Conclusions

Monadic Interpreters

- Syntax:

```
type Var = String

data Tm = V Var | L Var Tm | Tm :@ Tm
        | N Int | Tm :+ Tm | ...
        | TT | FF | Not Tm | ...
        | If Tm Tm Tm
        | ...
        -- specific for Maybe
        | Raise | Tm ‘Handle‘ Tm
```

Monadic Interpreters

- Semantic categories:

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

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

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)
                  | a0 == a    = b
                  | otherwise = unsafeLookup a0 abs
```

Monadic Interpreters

- Evaluation:

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

_ev :: MonadEv t => Tm -> Env t -> t (Val t)
_ev (V x) env = return (unsafeLookup x env)
_ev (e :@ e') env
        = ev e  env >>= \ (F f) ->
          ev e' env >>= \ a ->
          f a
_ev (L x e) env
        = return (F (\ a -> ev e (update x a env)))
_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))
...
```

Monadic Interpreters

- Standard evaluator:

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

- Error handling evaluator:

```
instance MonadEv Maybe where
    ev Raise env = raise
    ev (e0 ‘Handle‘ e1) env
              = ev e0 env ‘handle‘ ev e1 env
    ev (e0 ‘Div‘ e1) env
              = ev e0 env >>= \ (I n0) ->
                ev e1 env >>= \ (I n1) ->
                if n1 == 0
                   then raise
                   else return (I (n0 ‘div‘ n1))
    ev (e0 ‘Mod‘ e1) env = ...
    ev e env = _ev e env
```

Outline

Comonads

- Comonads model notions of value in a context;
  - DA is the type of contextually situated values of A.
- A context-relying function from A to B
  is a map $A \to B$ in the coKleisli category,
  - i.e., a map $DA \to B$ in the base category.
- Some examples:
  - $DA = A$, the identity comonad,
  - $DA = A \times E$, the product comonad, environment,
  - $DA = \mathrm{Str}\, A = \nu X.A \times X$, the streams comonad.

Comonads in Haskell

- The comonad class:

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

- Derived operations:

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

cdup :: Comonad d => d a -> d (d a)
cdup   = cobind id
```

- The identity comonad:

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

Comonads in Haskell

- The product comonad:

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

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

- Operations specific for product comonad:

```
askP :: Prod e a -> e
askP (_ :& e) = e

localP :: (e -> e) -> Prod e a -> Prod e a
localP g (a :& e) = (a :& g e)
```

Comonads in Haskell

- The streams comonad:

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

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

- Operations specific for streams comonad:

```
fbyS :: a -> Stream a -> Stream a
fbyS a as = a :< as

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

Outline

Comonads for Stream Functions

- Streams model signals in discrete time.
- They are naturally isomorphic to functions from natural numbers:

$$\mathsf{Str}A \;\cong\; \mathsf{Nat} \Rightarrow A$$

- Haskell implementation of the isomorphism:

```
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)
```

Comonads for Stream Functions

- Dataflow programs are modelled by stream functions.
- General stream functions $\mathsf{Str}\,A \to \mathsf{Str}\,B$ are in natural bijection with maps:

$$\begin{aligned}
\mathsf{Str}\,A \to \mathsf{Str}\,B &\cong \mathsf{Str}\,A \to (\mathsf{Nat} \Rightarrow B) \\
&\cong \mathsf{Str}\,A \times \mathsf{Nat} \to B \\
&= \mathsf{StrPos}\,A \to B \\
&\cong (\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \to B \\
&= \mathsf{FunArg}\,A \to B \\
&\cong \mathsf{List}\,A \times A \times \mathsf{Str}\,A \to B \\
&= \mathsf{LVS}\,A \to B
\end{aligned}$$

- The values of A in context for causal stream functions are:

$$\mathsf{LV}\,A = \mathsf{List}\,A \times A$$

Comonads for Stream Functions

- Dataflow programs are modelled by stream functions.
- General stream functions $\mathsf{Str}\,A \to \mathsf{Str}\,B$ are in natural bijection with maps:

$$
\begin{array}{rcl}
\mathsf{Str}\,A \to \mathsf{Str}\,B & \cong & \mathsf{Str}\,A \to (\mathsf{Nat} \Rightarrow B) \\
& \cong & \mathsf{Str}\,A \times \mathsf{Nat} \to B \\
& = & \mathsf{StrPos}\,A \to B \\
& \cong & (\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \to B \\
& = & \mathsf{FunArg}\,A \to B \\
& \cong & \mathsf{List}\,A \times A \times \mathsf{Str}\,A \to B \\
& = & \mathsf{LVS}\,A \to B
\end{array}
$$

- The values of A in context for causal stream functions are:

$$\mathsf{LV}\,A = \mathsf{List}\,A \times A$$

Comonads for Stream Functions

- Dataflow programs are modelled by stream functions.
- General stream functions $\mathsf{Str}\,A \to \mathsf{Str}\,B$ are in natural bijection with maps:

$$
\begin{array}{rcl}
\mathsf{Str}\,A \to \mathsf{Str}\,B & \cong & \mathsf{Str}\,A \to (\mathsf{Nat} \Rightarrow B) \\
& \cong & \mathsf{Str}\,A \times \mathsf{Nat} \to B \\
& = & \mathsf{StrPos}\,A \to B \\
& \cong & (\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \to B \\
& = & \mathsf{FunArg}\,A \to B \\
& \cong & \mathsf{List}\,A \times A \times \mathsf{Str}\,A \to B \\
& = & \mathsf{LVS}\,A \to B
\end{array}
$$

- The values of A in context for causal stream functions are:

$$\mathsf{LV}\,A \;=\; \mathsf{List}\,A \times A$$

Comonads for Stream Functions

- Dataflow programs are modelled by stream functions.
- General stream functions $\mathsf{Str}\,A \to \mathsf{Str}\,B$ are in natural bijection with maps:

$$
\begin{aligned}
\mathsf{Str}\,A \to \mathsf{Str}\,B &\cong \mathsf{Str}\,A \to (\mathsf{Nat} \Rightarrow B) \\
&\cong \mathsf{Str}\,A \times \mathsf{Nat} \to B \\
&= \mathsf{StrPos}\,A \to B \\
&\cong (\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \to B \\
&= \mathsf{FunArg}\,A \to B \\
&\cong \mathsf{List}\,A \times A \times \mathsf{Str}\,A \to B \\
&= \mathsf{LVS}\,A \to B
\end{aligned}
$$

- The values of A in context for causal stream functions are:

$$\mathsf{LV}\,A = \mathsf{List}\,A \times A$$

Comonads for Stream Functions

- Dataflow programs are modelled by stream functions.
- General stream functions $\mathsf{Str}\,A \to \mathsf{Str}\,B$ are in natural bijection with maps:

$$
\begin{aligned}
\mathsf{Str}\,A \to \mathsf{Str}\,B &\cong \mathsf{Str}\,A \to (\mathsf{Nat} \Rightarrow B) \\
&\cong \mathsf{Str}\,A \times \mathsf{Nat} \to B \\
&= \mathsf{StrPos}\,A \to B \\
&\cong (\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \to B \\
&= \mathsf{FunArg}\,A \to B \\
&\cong \mathsf{List}\,A \times A \times \mathsf{Str}\,A \to B \\
&= \mathsf{LVS}\,A \to B
\end{aligned}
$$

- The values of A in context for causal stream functions are:

$$\mathsf{LV}\,A = \mathsf{List}\,A \times A$$

Comonads for Stream Functions

- A comonad for general stream functions

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

instance Comonad FunArg where
   counit (f :# i)    = f i
   cobind k (f :# i) = (\ i’ -> k (f :# i’)) :# i
```

- Operations specific for FunArg comonad:

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

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

Comonads for Stream Functions

- A comonad for causal stream functions

```
data List a  = Nil  |  List a :> a
data LV a    = List a := a

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)
```

- Operations specific for LV comonad:

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

Comonads for Stream Functions

- Interpreting coKleisli arrows as stream functions:

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

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


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

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

Outline

Comonadic Interpreters

- Syntax:

```
data Tm = ...
        -- for general and causal stream fun-s
        | Tm `Fby` Tm
        -- for general stream fun-s only
        | Next Tm
```

- Semantic categories:

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

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

Comonadic Interpreters

- Evaluation:

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

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

- But how to define the function extend?

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

Comonadic Interpreters

- Evaluation:

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

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

- But how to define the function extend?

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

Comonadic Interpreters

- Comonads with zipping

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

- Instances of ComonadZip

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

instance ComonadZip FunArg where
      czip (f :# i) (g :# j)
         | i == j = (\ n -> (f n, g n)) :# i

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

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

Comonadic Interpreters

- Evaluation of λ-expressions:

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

_ev :: ComonadEv d => Tm -> d (Env d) -> Val d
...
_ev (L x e) denv
      = F (\ d -> ev e (cmap repair (czip d denv)))
      where repair (a, env) = update x a env
```

Comonadic Interpreters

- Standard evaluator:

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

- Syncronous evaluator:

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

- Evaluator of general stream functions:

```
instance ComonadEv FunArg where
    ev (e0 ‘Fby‘ e1) denv
          = ev e0 denv ‘fbyFA‘ cobind (ev e1) denv
    ev (Next e) denv
          = nextFA (cobind (ev e) denv)
    ev e denv = _ev e denv
```

Outline

Distributive Laws

Definition

Given a comonad (D, ε, δ) and a monad (T, η, μ) on a category $\mathcal{C}$, a distributive law of D over T is a natural transformation λ with components $DTA \to TDA$ subject to four coherence conditions:

$$\begin{tikzcd} DA \arrow[r, "D\eta_A"] \arrow[dr, "\eta_{DA}"'] & DTA \arrow[d, "\lambda_A"] \\ & TDA \end{tikzcd} \qquad \begin{tikzcd} DT^2A \arrow[rr, "D\mu_A"] \arrow[d, "\lambda_{TA}"'] & & DTA \arrow[d, "\lambda_A"] \\ TDTA \arrow[r, "T\lambda_A"'] & T^2DA \arrow[r, "\mu_{DA}"'] & TDA \end{tikzcd}$$

$$\begin{tikzcd} DTA \arrow[r, "\lambda_A"] \arrow[dr, "\varepsilon_{TA}"'] & TDA \arrow[d, "T\varepsilon_A"] \\ & TA \end{tikzcd} \qquad \begin{tikzcd} DTA \arrow[rr, "\lambda_A"] \arrow[d, "\delta_{TA}"'] & & TDA \arrow[d, "T\delta_A"] \\ D^2TA \arrow[r, "D\lambda_A"'] & DTDA \arrow[r, "\lambda_{DA}"'] & TD^2A \end{tikzcd}$$

Distributive Laws

Definition

A distributive law of D over T defines a biKleisli category $\mathcal{C}_{D,T}$ with

- objects $|\mathcal{C}_{D,T}| = |\mathcal{C}|$,
- morphisms $\mathcal{C}_{D,T}(A, B) = \mathcal{C}(DA, TB)$,
- identities $(\mathrm{id}_{D,T})_A = \eta_A \circ \varepsilon_A$
- composition $g \circ_{D,T} f = g^\star \circ \lambda_B \circ f^\dagger$.

Distributive Laws in Haskell

- The distributive law class:

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

- Distributive laws for identity (co)monad:

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

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

- Distributive law for Prod and Maybe:

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

Distributive Laws in Haskell

- The type of partial streams (clocked signals in discrete time) over a type A is $\mathsf{Str}(\mathsf{Maybe}A)$.
- Strict causal partial-stream functions are representable as biKleisli arrows of a distributive law of LV over Maybe.
- A distributive law between LV and Maybe:

```
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)
```

Distributive Laws in Haskell

- Interpreting a biKleisli arrow as a partial-stream function:

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

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

Outline

Distributivity-based Interpreters

- Syntax:

```
data Tm = ...
        -- specific for LV
        | Tm `Fby` Tm
        -- specific for Maybe
        | Nosig | Merge Tm Tm
```

- Semantic categories:

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

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

Distributivity-based Interpreters

- Evaluation:

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

_ev :: DistEv d t =>
       Tm -> d (Env d t) -> t (Val d t)
_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))
...
```

Distributivity-based Interpreters

- Evaluation (cont.):

```
_ev :: DistEv d t =>
       Tm -> d (Env d t) -> t (Val d t)
...
_ev (V x) d denv
    = return (unsafeLookup x (counit denv))
_ev (e :@ e') denv
    = ev e denv >>= \ (F f) ->
      dist (cobind (ev e') denv) >>= \ d ->
      f d
_ev (L x e) denv
    = return (F (\ d -> ev e (cmap repair
                                  (czip d denv))))
    where repair (a, env) = update x a env
```

Comonadic Interpreters

- Evaluator of partial stream functions:

```
instance DistEv LV Maybe where
  ev (e0 `Fby` e1) denv
          = ev e0 denv >>= \ a ->
            dist (cobind (ev e1) denv) >>= \ d ->
            return (fbyLV a d)
  ev Nosig denv = raise
  ev (e0 `Merge` e1) denv
          = ev e0 denv `handle` ev e1 denv
  ev e denv = _ev e denv
```

Conclusions and Future Work

- A general framework for dataflow programming and for semantics based on comonads and distributive laws.
- The first-order dataflow language designs delivered by the framework agree very well with the designs by the dataflow people.
- But the framework settles also the meaning of higher-order dataflow computation.
- Comonadic semantics extends also to non-linear datatypes (TFP'05).
- To do: Instantiation for timed dataflow programming
    - reactive functional programming (eg. a la Yampa)
- Comonadic IO ?!
    - cf. OI comonad by Kieburtz
    - stream based IO in Haskell 1.0
- Special syntax ??