The essence of dataflow programming

Tarmo Uustalu
Varmo Vene

Teooriapäevad Kokel^2, 4.-6.2.2005

Motivation

- Following Moggi and Wadler, it is standard in programming and semantics to analyze various notions of computation with an effect as monads.
- But there is a need for both finer and more permissive mathematical abstractions to uniformly describe the numerous function-like concepts encountered in programming.
- Some proposals: Lawvere theories (Power, Plotkin), Freyd categories (Power, Robinson).

- In functional programming, Hughes invented Freyd categories independently of Power, Robinson under the name of arrow types and has been promoting them an abstraction especially handy in programming with signals/flows.
- This has been picked up; there is by now both a library and specialized syntax for arrows in Haskell, as well as an arrows-based library for functional reactive programming.
- But what about comonads? They have not found extensive use (some examples by Brookes and Geva, Kieburtz, but mostly artificial).

This talk

- Thesis: Used properly, comonads are exactly the right tool for programming signal/flow functions, accounting both for general signal functions and for causal ones (where the output at a given time can only depend on the input until that time).

- This extends Moggi's modular approach to language semantics to languages for implicit context based paradigms such as intensional programming in Lucid or synchronous dataflow programming in Lustre/Lucid Synchrone: context relying functions are interpreted as pure functions via a comonad translation.

  For such languages, Moggi-style accounts have not been available thus far.

Outline

- Monads, monads in programming and semantics
- Freyd categories/arrow types and programming with stream functions
- Comonads for programming with stream functions, semantics
- A distributive law for programming with partial-stream functions, semantics

Monads

- A monad (in the Kleisli format) 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 $\eta_A : A \to TA$ (unit), and an operation $-^\star$ taking every map $k : A \to TB$ in $\mathcal{C}$ to a map $k^\star : TA \to TB$ (extension operation) such that
  - for any $k : A \to TB$, $k^\star \circ \eta_A = k$,
  - ${\eta_A}^\star = \mathrm{id}_{TA}$,
  - for any $k : A \to TB$, $\ell : B \to TC$, $(\ell^\star \circ k)^\star = \ell^\star \circ k^\star$.
- Any monad $(T, \eta, -^\star)$ defines a category $\mathcal{C}_T$ where $|\mathcal{C}_T| = |\mathcal{C}|$ and $\mathcal{C}_T(A, B) = \mathcal{C}(A, TB)$, $(\mathrm{id}_T)_A = \eta_A$, $\ell \circ_T k = \ell^\star \circ k$ (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 programming and semantics, monads are used to model notions of computation with an effect; TA is the type of computations of values of A.

  An function with an effect 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:
  - $TA = \mathsf{Maybe} A = A + 1$, error (partiality), $TA = A + E$, exceptions,
  - $TA = E \Rightarrow A$, environment,
  - $TA = \mathsf{List} A = \mu X.1 + A \times X$, non-determinism,
  - $TA = S \Rightarrow A \times S$, state,
  - $TA = (A \Rightarrow R) \Rightarrow R$, continuations,
  - $TA = \mu X.A + (U \Rightarrow X)$, interactive input,
  - $TA = \mu X.A + V \times X \cong A \times \mathsf{List} V$, interactive output,
  - $TA = \mu X.A + FX$, the free monad over F,
  - $TA = \nu X.A + FX$, the free completely iterative monad over F.

Monads in Haskell

- The monad class is defined in the Prelude:

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

- The error monad:

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

errorM :: Maybe a
errorM = Nothing

handleM :: Maybe a -> Maybe a -> Maybe a
Nothing `handleM` d = d
Just a  `handleM` _ = Just a
```

- The non-determinism monad:

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

deadlockL :: [a]
deadlockL = []

choiceL :: [a] -> [a] -> [a]
as0 `choiceL` as1 = as0 ++ as1
```

Monadic semantics

- Syntax:

```
type Var = String

data Tm = V Var | L Var Tm | Tm :@ Tm | Rec Tm
        | N Int | 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
```

- Semantic categories:

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

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

env0 :: Env t
env0 = []
```

- 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 (L x e) env = return (F (\ a -> ev e ((x, a) : env)))
_ev (e :@ e') env = ev e env >>= \ (F f) ->
                    ev e' env >>= \ a ->
                    f 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
```

- Evaluation cont'd:

```
instance MonadEv Maybe where
  ev Error env = errorM
  ev (e0 `Handle` e1) env = ev e0 env `handleM` ev e1 env
  ev e env = _ev e env

testM :: Tm -> Maybe (Val Maybe)
testM = ev e env0

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

testL :: Tm -> [Val []]
testL = ev e env0
```

Freyd categories / arrow types

- Freyd categories are a generalization of Kleisli categories of strong monads.

- A symmetric premonoidal category is the same as a symmetric monoidal category except that the tensor is not required not be bifunctorial, only functorial in each of its two arguments separately. A map $f : A \to B$ of such a category is called central if the two composites $A \otimes C \to B \otimes D$ agree and the two composites $C \otimes A \to D \otimes B$ agree for every map $g : C \to D$.

  A 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.

- Freyd categories a.k.a. arrow types in Haskell (as 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)
```

- Kleisli arrows as arrows:

```
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) ->
                                    k a >>= \ b -> return (b, c))
```

- The general stream functions arrow type (to model transformers of signals in discrete time):

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

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

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)
```

- Delay:

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

Comonads

- Comonads are the formal dual of monads.
- 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
  - for any $k : DA \to B$, $\varepsilon_B \circ k^\dagger = k$,
  - ${\varepsilon_A}^\dagger = \mathsf{id}_{DA}$,
  - for any $k : DA \to B$, $\ell : DB \to C$, $(\ell \circ k^\dagger)^\dagger = \ell^\dagger \circ k^\dagger$.
- 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 usable to model notions of value in a context; DA would be the type of contextually situated values of A.

  A context-relying function from A to B would be a map $A \to B$ in the coKleisli category, i.e., a map $DA \to B$ in the base category.

- Some examples:
  - $DA = A \times E$, 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,
  - $DA = \mu X. A \times FA$, the cofree recursive comonad over F.

Comonads in Haskell

- The basic implementation:

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

- The product comonad:

```
data With e a = a :- e

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

- 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
```

Comonads for general and causal stream functions

- Streams (signals in discrete time) are naturally isomorphic to functions from natural numbers: $\mathsf{Str}A \cong \mathsf{Nat} \Rightarrow A$.

- General stream functions $\mathsf{Str}A \to \mathsf{Str}B$ are thus in natural bijection with maps $\mathsf{Str}A \times \mathsf{Nat} \to B$.

- Hence the values of A in context for general stream functions are $\mathsf{StrPos}A = \mathsf{Str}A \times \mathsf{Nat} \cong \mathsf{LVS}A = \mathsf{List}A \times A \times \mathsf{Str}A$.

  A time point partitions a stream into its past (a list), present (a value) and future (a stream).

- The values of A in context for causal stream functions are $\mathsf{LV}A = \mathsf{List}A \times A \cong \mu X.A \times \mathsf{Maybe}X$.

  This is the cofree recursive comonad over the Maybe functor.

- Streams and isomorphism of streams to functions from naturals:

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

str2fun :: Stream a -> Int -> a
fun2str :: (Int -> a) -> Stream a
```

- Streams with a marked position: values in a context for general stream functions:

```
data StrPos a = SP (Stream a) Int

instance Comonad StrPos where
  counit (SP as i) = str2fun as i
  cobind k (SP as i) = SP (fun2str (\ i' -> k (SP as i'))) i

runSP :: (StrPos a -> b) -> Stream a -> Stream b
runSP k as = runSP' k as 0

runSP' k as i = k (SP as i) :< runSP' k as (i + 1)
```

- Delay ("followed by") operation:

```
fbySP :: a -> StrPos a -> a
fbySP a (SP as 0)       = a
fbySP _ (SP as (i + 1)) = str2fun as i
```

- Summation:

```
sumSP :: Num a => StrPos a -> a
sumSP (SP as 0)       = str2fun as 0
sumSP (SP as (i + 1)) = str2fun as (i + 1) + sumSP (SP as i)
```

- Compression (non-causal!):

```
compress :: StrPos a -> (a, a)
compress (SP as i) = (str2fun as (2 * i), str2fun as (2 * i + 1))
```

- List-value pairs, values in a context for causal stream functions:

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

data LV a = List a := a

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

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

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

- A feedback resolution combinator:

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

- Feedbacks can be run directly:

```
runbase :: (List (a, b) -> a -> b) -> Stream a -> Stream b
runbase k (a :< as) = runbase' k Nil a as

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

- Feedbacks can also be composed directly:

```
compbase :: (List (a, b) -> a -> b)
               -> (List ((a, b), c) -> (a, b) -> c)
                  -> List (a, (b, c)) -> a -> (b, c)
compbase k l e a
         = let
             e'  = fmap (\ (a, (b, c)) -> (a, b)) e
             e'' = fmap (\ (a, (b, c)) -> ((a, b), c)) e
             b   = k e' a
             c   = l e'' (a, b)
           in (b, c)
```

- Delay:

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

- Summation directly and with feedback:

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

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

Comonadic semantics of a dataflow language

- Comonads with zipping:

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

instance ComonadZip LV where
  czip (az := a) (bz := b) = czipP az bz := (a, b)
      where czipP Nil Nil = Nil
            czipP (az :> a) (bz :> b) = czipP az bz :> (a, b)
```

- Syntax:

```
type Var = String

data Tm = V Var | L Var Tm | Tm :@ Tm | Rec Tm
        | N Int | Tm :+ Tm | ...
        | Tm :== Tm | ...
        | TT | FF | Not Tm | ... | If Tm Tm Tm
        -- specific for LV
        | Fby Tm Tm
```

- Semantic domains:

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

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

env0 :: Int -> Env LV
env0 n = env0P n := []
          where env0P 0 = Nil
                env0P (n + 1) = env0P n :> []
```

- Evaluation:

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

_ev :: ComonadEv d => Tm -> Env d -> Val d
_ev (V x) env = unsafelookup x (counit env)
_ev (L x e) env = F (\ d -> ev e (cobind (repair . counit) (czip d env)))
                                            where repair (a, g) = (x, a) : g
_ev (e :@ e') env = case ev e env of
                       F f -> f (cobind (ev e') env)

_ev (N n) env = I n
_ev (e0 :+ e1) env = case ev e0 env of
                        I n0 -> case ev e1 env of
                           I n1 -> I (n1 + n2)
...
_ev TT env = B True
_ev FF env = B False
_ev (Not e) env = case ev e env of
                     B b -> B (not b)
...
_ev (If e e0 e1) env = case ev e env of
                          B b -> if b then ev e0 env else ev e1 env
```

- Evaluation cont'd:

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

testLV :: Tm -> Int -> LV (Val LV)
testLV e n = cobind (ev e) (env0 n)
```

- Examples:

```
pos = Rec (L "pos" (N 0 `Fby` (V "pos" :+ N 1)))

sums = L "x" (Rec (L "sumx" (V "x" :+ (N 0 `Fby` V "sumx"))))

diff = L "x" (V "x" :- (N 0 `Fby` V "x"))

fact = Rec (L "fact" (N 1 `Fby` (V "fact" :* (pos :+  N 1))))

fibo = Rec (L "fibo" (N 0 `Fby` (V "fibo" :+ (N 1 `Fby` V "fibo"))))
```

Distributive laws

- Given a comonad $(D, \varepsilon, -^\dagger)$ and a monad $(T, \eta, -^\star)$ 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.

  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.

A distributive law for causal partial-stream functions

- 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.
- Distributive laws in Haskell:

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

- A distributive law between LV and Maybe:

```
instance Dist LV Maybe where
  dist (az := Nothing) = Nothing
  dist (az := Just a)  = Just (filterJ az := a)
          where filterJ Nil = Nil
                filterJ (az :> Nothing) = filterJ az
                filterJ (az :> Just a)  = filterJ az :> a
```

- 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'
```

- The 'when' operation from dataflow languages:

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

Distributive law semantics of a clocked dataflow language

- Syntax:

```
type Var = String

data Tm = V Var | L Var Tm | Tm :@ Tm | Rec Tm
        | N Int | Tm :+ Tm | ...
        | Tm :== Tm | ...
        | TT | FF | Not Tm | ... | If Tm Tm Tm
        -- specific for LV
        | Fby Tm Tm
        -- specific for Maybe
        | Nosig | Merge Tm Tm
```

- Semantic domains:

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

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

env0 :: Int -> Env LV Maybe
env0 n = env0P n := []
          where env0P 0 = Nil
                env0P (n + 1) = env0P n :> []
```

- Evaluation:

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

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

_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
```

- Evaluation cont'd:

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

testLVM :: Tm -> Int -> LV (Maybe (Val LV Maybe))
testLVM e n = cobind (ev e) (env0 n)
```

- Example:

```
sieve = Rec (L "sieve" (L "x" (
               If (TT `Fby` FF)
                  (V "x")
                  (V "sieve" :@
                    (If ((V "x" `Mod` (first :@ V "x")) :/= N 0) (V "x") Nosig)))))

sieveMain = sieve :@ (pos :+ N 2)
```

Conclusions and future work

- A general framework for signal/flow based programming and for semantics. Based on a well-understood mathematical construction—comonad—, allowing generalizations from signal/flow processing to more sophisticated implicit context based paradigms of programming.
- Allows for modular simultaneous use of multiple notions of a context via combinations of multiple comonads (e.g., the multiple dimensions of Multidimensional Lucid) and for combinations of a context and an effect via combinations of a comonad and a monad (e.g., the partiality of Lustre/Lucid Synchrone).
- Allows for principled design of higher-order extensions for intensional and dataflow languages.
- In progress: From discrete time to continuous time, from clock-tick based to event based programming with signals.