Signals and Comonads

Tarmo Uustalu
Varmo Vene

TYPES'04, Jouy en Josas, 15–18 Dec. 2004

Motivation

- Following Moggi and Wadler, it is standard in programming and semantics to analyze various notions of computation with side-effects 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.
- Hughes has promoted arrows as an abstraction especially handy in programming with signals/flows.
- This has been taken up in functional reactive programming; there is by now both a library and specialized syntax for arrows in Haskell.
- However, there is also the standard concept of comonad which has not found extensive use (some examples by Kieburtz, but mostly artificial).

This talk

- Claim: 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: 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
- Arrows and programming with signals
- Comonads for programming with signals, semantics

Monads

- A monad (in the Kleisli format) on a category $(C, \mathsf{id}, \circ)$ is given by a mapping $T : |C| \to |C|$ together with a $|C|$-indexed family η of maps $\eta_A : A \to TA$ (unit), and an operation $-^\star$ taking every map $k : A \to TB$ in C to a map $k^\star : TA \to TB$ (extension operation) such that
  - for any $f : A \to TB$, $k^\star \circ \eta_A = k$,
  - ${\eta_A}^\star = \mathsf{id}_{TA}$,
  - 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 $(C_T, \mathsf{jd}, \bullet)$ where $|C_T| = |C|$ and $C_T(A, B) = C(A, TB)$, $\mathsf{jd}_A = \eta_A$, $\ell \bullet k = \ell^\star \circ k$ (Kleisli category).

- In programming and semantics, monads are used to model notions of computation with a side-effect; TA is the type of computations of values of A.

  An function with effect from A to B is a map $A \to B$ in the Kleisli category, which is a map $A \to TB$ in the base category.

- Some examples applied in semantics:
  - $TA = A + E$, error/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,
  - $TA = \mu X.A + Y \times X \cong A \times \mathsf{List}Y$, 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        = Just
    Just a  >>= k = k a
    Nothing >>= k = Nothing

error :: Maybe b
error = Nothing
```

- The non-determinism monad:

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

choose :: (a, a) -> [a]
choose a0 a1 = [a0, a1]

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

Arrows

- Arrows are a generalization of Kleisli arrows of a monad. They were invented in functional programming by Hughes to model function-like entities such as stream functions.
- Arrows in Haskell (as in Control.Arrow):

```
class Arrow r where
    arr :: (a -> b) -> r a b
    (>>>) :: r a b -> r b c -> r a c
    first :: r a b -> r (a, c) (b, c)

    second :: r a b -> r (d, a) (d, b)
    second f = arr swap >>> first f >>> arr swap
    (***) :: r a b -> r a' b' -> r (a, a') (b, b')
    f *** g = first f >>> second g
    (&&&) :: r a b -> r a b' -> r a (b, b')
    f &&& g = arr (\ a -> (a, a)) >>> f *** g
```

- The stream functions arrow (to model signal transformers in discrete time):

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

newtype SF a b = SF (Stream a -> Stream b)

instance Arrow SF where
    arr f = SF (\ (a :< as) -> f a :< arr f as)
    SF k >>> SF l = SF (l . k)
    first SF k = SF (uncurry zip .  (\ (as, ds) -> k as, ds) . unzip)

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

- Categorically, arrows are not so simple as monads (they are premonoidal or indexed categories with extra structure), nor are they as fundamental . . .

Comonads

- Comonads are the formal dual of monads.
- A comonad on a category $(C, \mathsf{id}, \circ)$ is given by a mapping $D : |C| \to |C|$ together with a $|C|$-indexed family η of maps $\varepsilon_A : DA \to A$ (counit), and an operation $-^\star$ taking every map $k : DA \to B$ in C to a map $k^\dagger : DA \to DB$ (coextension operation) such that
  - for any $k : DA \to B$, $\varepsilon_B \circ k^\dagger = k$,
  - ${\varepsilon_A}^\dagger = \mathsf{id}_{TA}$,
  - 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 $(C_D, \mathsf{jd}, \bullet)$ where $|C_D| = |C|$ and $C_D(A, B) = C(DA, B)$, $\mathsf{jd}_A = \varepsilon_A$, $\ell \bullet k = \ell \circ k^\dagger$ (coKleisli category).

- Comonads should be usable to model notions of value-in-context; DA would be the type of contextually situated values of A.

  A context-relying functions 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 p@(_ :- e) = k p :- e
```

- The streams comonad:

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

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

Comonads for signal transformers (discrete time)

- Streams (discrete-time signals) are 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 $(\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \to B$.
- Hence the values of A in context for general stream functions are $D^g A = (\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \cong \mathsf{Str}A \times \mathsf{Nat} \cong \mathsf{List}A \times A \times \mathsf{Str}A$.
- The values of A in context for causal stream functions are $D^c A = \mathsf{List}A \times A \cong \mu X.A \times (1 + X)$.

  This is the cofree recursive comonad over the functor $X \mapsto 1 + X$ (i.e., the Maybe functor).
- The idea is that at any time, the past of a value is a (snoc) list and the future is a stream.

- The isomorphism of streams to functions from naturals:

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

str2fun :: Stream a -> Int -> a
str2fun (a :< as) 0 = a
str2fun (a :< as) (i + 1) = str2fun as i

fun2str :: (Int -> a) -> Stream a
fun2str f = fun2str' f 0

fun2str' f i = f i :< fun2str' f (i + 1)
```

- Streams with a marked position: values-in-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:

```
delaySP :: a -> StrPos a -> a
delaySP a (SP as 0)       = a
delaySP _ (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-element pairs, values in context for causal stream functions:

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

data LE a = List a := a

instance Comonad LE where
  counit (_ := a) = a
  cobind k le@(al := _) = cobindp al := k le where
        cobindp Nil          = Nil
        cobindp (al' :> a') = cobindp al' :> k (al' := a')

runLE :: (LE a -> b) -> Stream a -> Stream b
runLE k (a :< as) = runLE' k Nil a as

runLE' k al a (a' :< as')
                  = k (al := a) :< runLE' k (al :> a) a' as'
```

- A feedback resolution combinator:

```
feedback :: (List (a, b) -> a -> b) -> (LE a -> b)
feedback k al = k abl' a
          where (abl' := (a, _))
                       = cobind (pair counit (feedback k)) al
```

- 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 abl a (a' :< as')
                    = b :< runbase' k (abl :> (a, b)) a' as'
                                                 where b = k abl 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:

```
delayLE :: a -> LE a -> a
delayLE a0 (Nil       := _) = a0
delayLE _  ((_ :> a') := _) = a'
```

- Summation directly and with feedback:

```
sumLE :: Num a => LE a -> a
sumLE (Nil          := a) = a
sumLE ((al' :> a') := a) = sumLE (al' := a') + a

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

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 context via combinations of comonads (e.g., multiple dimensions in multidimensional Lucid).
- In progress: From discrete time to continuous time, from clock-tick-based to event-based programming with signals.
- In progress: Semantics of languages like Lucid ("intensional", general stream functions), Lustre/Lucid Synchrone (synchronous dataflow, causal stream functions).