LiU-FP2010 Part II: Lecture 6

More about Monads and Other Notions of Effectful Computation

Henrik Nilsson

University of Nottingham, UK

This Lecture

- Monads in Haskell
- Some standard monads
- Combining effects: monad transformers
- Arrows
- FRP and Yampa

Monads in Haskell

In Haskell, the notion of a monad is captured by a ***Type Class***:

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

Allows names of the common functions to be overloaded and sharing of derived definitions.

The Maybe Monad in Haskell

```
instance Monad Maybe where
    -- return :: a -> Maybe a
    return = Just

    -- (>>=) :: Maybe a -> (a -> Maybe b)
    --           -> Maybe b
    Nothing  >>= _ = Nothing
    (Just x) >>= f = f x
```

Exercise 1: A State Monad in Haskell

Haskell 98 does not permit type synonyms to be instances of classes. Hence we have to define a new type:

```
newtype S a = S (Int -> (a, Int))

unS :: S a -> (Int -> (a, Int))
unS (S f) = f
```

Provide a `Monad` instance for `S`.

Exercise 1: Solution

```
instance Monad S where
    return a = S (\s -> (a, s))

    m >>= f = S $ \s ->
        let (a, s') = unS m s
        in unS (f a) s'
```


Monad-specific Operations (1)

To be useful, monads need to be equipped with additional operations specific to the effects in question. For example:

```
fail :: String -> Maybe a
fail s = Nothing

catch :: Maybe a -> Maybe a -> Maybe a
m1 `catch` m2 =
    case m1 of
        Just _  -> m1
        Nothing -> m2
```


Monad-specific Operations (2)

Typical operations on a state monad:

```
set :: Int -> S ()
set a = S (\_ -> ((), a))

get :: S Int
get = S (\s -> (s, s))
```

Moreover, need to “run” a computation. E.g.:

```
runS :: S a -> a
runS m = fst (unS m 0)
```

The do-notation (1)

Haskell provides convenient syntax for programming with monads:

$$\texttt{do}\\ \quad \texttt{a <- } exp_1\\ \quad \texttt{b <- } exp_2\\ \quad \texttt{return } exp_3$$

is syntactic sugar for

$$exp_1 \texttt{ >>= \textbackslash a ->}\\ exp_2 \texttt{ >>= \textbackslash b ->}\\ \texttt{return } exp_3$$

The do-notation (2)

Computations can be done solely for effect, ignoring the computed value:

$$\texttt{do}\\ \quad exp_1\\ \quad exp_2\\ \quad \texttt{return } exp_3$$

is syntactic sugar for

$$exp_1 \texttt{ >>= \textbackslash\_ ->}\\ exp_2 \texttt{ >>= \textbackslash\_ ->}\\ \texttt{return } exp_3$$

The do-notation (3)

A `let`-construct is also provided:

$$\texttt{do}\\ \quad \texttt{let a = } exp_1\\ \quad \quad \texttt{b = } exp_2\\ \quad \texttt{return } exp_3$$

is equivalent to

$$\texttt{do}\\ \quad \texttt{a <- return } exp_1\\ \quad \texttt{b <- return } exp_2\\ \quad \texttt{return } exp_3$$

Numbering Trees in do-notation

```
numberTree :: Tree a -> Tree Int
numberTree t = runS (ntAux t)
    where
        ntAux :: Tree a -> S (Tree Int)
        ntAux (Leaf _) = do
            n <- get
            set (n + 1)
            return (Leaf n)
        ntAux (Node t1 t2) = do
            t1' <- ntAux t1
            t2' <- ntAux t2
            return (Node t1' t2')
```

The Compiler Fragment Revisited (1)

Given a suitable “Diagnostics” monad `D` that collects error messages, `enterVar` can be turned from this:

```
enterVar :: Id -> Int -> Type -> Env
            -> Either Env ErrorMgs
```

into this:

```
enterVarD :: Id -> Int -> Type -> Env
             -> D Env
```

and then `identDefs` from this . . .

The Compiler Fragment Revisited (2)

```
identDefs l env [] = ([], env, [])
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'', ms1++ms2++ms3)
  where
    (e', ms1) = identAux l env e
    (env', ms2) =
       case enterVar i l t env of
          Left env' -> (env', [])
          Right m   -> (env,  [m])
    (ds', env'', ms3) =
      identDefs l env' ds
```


The Compiler Fragment Revisited (3)

into this:

```
identDefsD l env [] = return ([], env)
identDefsD l env ((i,t,e) : ds) = do
    e'            <- identAuxD l env e
    env'          <- enterVarD i l t env
    (ds', env'') <- identDefsD l env' ds
    return ((i,t,e') : ds', env'')
```

(Suffix D just to remind us the types have changed.)

The Compiler Fragment Revisited (4)

Compare with the “core” identified earlier!

```
identDefs l env [] = ([], env)
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'')
  where
    e'            = identAux l env e
    env'          = enterVar i l t env
    (ds', env'') = identDefs l env' ds
```

The monadic version is very close to ideal, without sacrificing functionality, clarity, or pureness!

The List Monad

Computation with many possible results, “nondeterminism”:

```
instance Monad [] where
    return a = [a]
    m >>= f  = concat (map f m)
    fail s   = []
```

Example:

```
x <- [1, 2]
y <- ['a', 'b']
return (x,y)
```

Result:

```
[(1,'a'),(1,'b'),
 (2,'a'),(2,'b')]
```


The Reader Monad

Computation in an environment:

```
instance Monad ((->) e) where
    return a = const a
    m >>= f  = \e -> f (m e) e

getEnv :: ((->) e) e
getEnv = id
```


The Haskell IO Monad

In Haskell, IO is handled through the IO monad. IO is ***abstract***! Conceptually:

```
newtype IO a = IO (World -> (a, World))
```

Some operations:

```
putChar     :: Char -> IO ()
putStr      :: String -> IO ()
putStrLn    :: String -> IO ()
getChar     :: IO Char
getLine     :: IO String
getContents :: String
```


Monad Transformers (1)

What if we need to support more than one type of effect?

For example: State and Error/Partiality?

We could implement a suitable monad from scratch:

```
newtype SE s a = SE (s -> Maybe (a, s))
```

Monad Transformers (2)

However:

- Not always obvious how: e.g., should the combination of state and error have been

```
newtype SE s a = SE (s -> (Maybe a, s))
```

- Duplication of effort: similar patterns related to specific effects are going to be repeated over and over in the various combinations.

Monad Transformers (3)

Monad Transformers can help:

- A ***monad transformer*** transforms a monad by adding support for an additional effect.
- A library of monad transformers can be developed, each adding a specific effect (state, error, . . .), allowing the programmer to mix and match.
- A form of ***aspect-oriented programming***.

Monad Transformers in Haskell (1)

- A ***monad transformer*** maps monads to monads. Represented by a type constructor T of the following kind:

```
T :: (* -> *) -> (* -> *)
```

- Additionally, a monad transformer ***adds*** computational effects. A mapping `lift` from computations in the underlying monad to computations in the transformed monad is needed:

```
lift :: M a -> T M a
```


Monad Transformers in Haskell (2)

- These requirements are captured by the following (multi-parameter) type class:

```
class (Monad m, Monad (t m))
      => MonadTransformer t m where
    lift :: m a -> t m a
```

Classes for Specific Effects

A monad transformer adds specific effects to ***any*** monad. Thus the effect-specific operations needs to be overloaded. For example:

```
class Monad m => E m where
    eFail :: m a
    eHandle :: m a -> m a -> m a

class Monad m => S m s | m -> s where
    sSet :: s -> m ()
    sGet :: m s
```


The Identity Monad

We are going to construct monads by successive transformations of the identity monad:

```
newtype I a = I a
unI (I a) = a

instance Monad I where
    return a = I a
    m >>= f = f (unI m)

runI :: I a -> a
runI = unI
```


The Error Monad Transformer (1)

```
newtype ET m a = ET (m (Maybe a))
unET (ET m) = m
```

Any monad transformed by ET is a monad:

```
instance Monad m => Monad (ET m) where
    return a = ET (return (Just a))

    m >>= f = ET $ do
        ma <- unET m
        case ma of
            Nothing -> return Nothing
            Just a  -> unET (f a)
```


The Error Monad Transformer (2)

We need the ability to run transformed monads:

```
runET :: Monad m => ET m a -> m a
runET etm = do
    ma <- unET etm
    case ma of
        Just a  -> return a
        Nothing -> error "Should not happen"
```

ET is a monad transformer:

```
instance Monad m =>
         MonadTransformer ET m where
    lift m = ET (m >>= \a -> return (Just a))
```

The Error Monad Transformer (3)

Any monad transformed by ET is an instance of E:

```
instance Monad m => E (ET m) where
    eFail = ET (return Nothing)
    m1 ‘eHandle‘ m2 = ET $ do
        ma <- unET m1
        case ma of
            Nothing -> unET m2
            Just _  -> return ma
```


The Error Monad Transformer (4)

A state monad transformed by ET is a state monad:

```
instance S m s => S (ET m) s where
    sSet s = lift (sSet s)
    sGet = lift sGet
```


Exercise 2: Running Transf. Monads

Let

```
ex2 = eFail ‘eHandle‘ return 1
```

1. Suggest a possible type for `ex2`.
   (Assume `1 :: Int`.)
2. Given your type, use the appropriate combination of “run functions” to run `ex2`.

Exercise 2: Solution

```
ex2 :: ET I Int
ex2 = eFail ‘eHandle‘ return 1

ex2result :: Int
ex2result = runI (runET ex2)
```

The State Monad Transformer (1)

```
newtype ST s m a = ST (s -> m (a, s))
unST (ST m) = m
```

Any monad transformed by ST is a monad:

```
instance Monad m => Monad (ST s m) where
    return a = ST (\s -> return (a, s))

    m >>= f = ST $ \s -> do
        (a, s’) <- unST m s
        unST (f a) s’
```


The State Monad Transformer (2)

We need the ability to run transformed monads:

```
runST :: Monad m => ST s m a -> s -> m a
runST stf s0 = do
    (a, _) <- unST stf s0
    return a
```

ST is a monad transformer:

```
instance Monad m =>
         MonadTransformer (ST s) m where
    lift m = ST (\s -> m >>= \a ->
                         return (a, s))
```


The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:

```
instance Monad m => S (ST s m) s where
    sSet s = ST (\_ -> return ((), s))
    sGet   = ST (\s -> return (s, s))
```

An error monad transformed by ST is an error monad:

```
instance E m => E (ST s m) where
    eFail = lift eFail
    m1 ‘eHandle‘ m2 = ST $ \s ->
        unST m1 s ‘eHandle‘ unST m2 s
```


Exercise 3: Effect Ordering

Consider the code fragment

```
ex3a :: (ST Int (ET I)) Int
ex3a = (sSet 42 >> eFail) ‘eHandle‘ sGet
```

Note that the exact same code fragment also can be typed as follows:

```
ex3b :: (ET (ST Int I)) Int
ex3b = (sSet 42 >> eFail) ‘eHandle‘ sGet
```

What is

```
runI (runET (runST ex3a 0))
runI (runST (runET ex3b) 0)
```

Exercise 3: Solution

```
runI (runET (runST ex3a 0))  =  0
runI (runST (runET ex3b) 0)  =  42
```

Why? Because:

```
ST s (ET I) a ≅ s -> (ET I) (a, s)
              ≅ s -> I (Maybe (a, s))
              ≅ s -> Maybe (a, s)
ET (ST s I) a ≅ (ST s I) (Maybe a)
              ≅ s -> I (Maybe a, s)
              ≅ s -> (Maybe a, s)
```


Exercise 4: Alternative ST?

To think about.

Could ST have been defined in some other way, e.g.

```
newtype ST s m a = ST (m (s -> (a, s)))
```

or perhaps

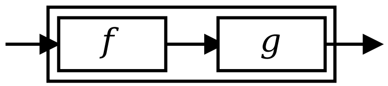
```
newtype ST s m a = ST (s -> (m a, s))
```


Problems with Monad Transformers

- With one transformer for each possible effect, we get a lot of combinations: the number grows quadratically; each has to be instantiated explicitly.
- Jaskelioff (2008,2009) has proposed a possible, more extensible alternative.

Arrows (1)

System descriptions in the form of block diagrams are very common. Blocks have inputs and outputs and can be combined into larger blocks. For example, serial composition:

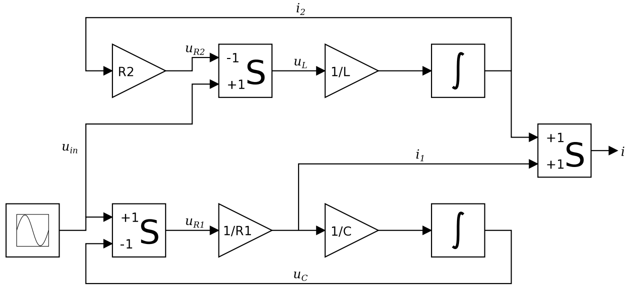


A *combinator* can be defined that captures this idea:

```
(>>>) :: B a b -> B b c -> B a c
```

Arrows (2)

But systems can be complex:



How many and what combinators do we need to be able to describe arbitrary systems?

Arrows (3)

John Hughes' ***arrow*** framework:

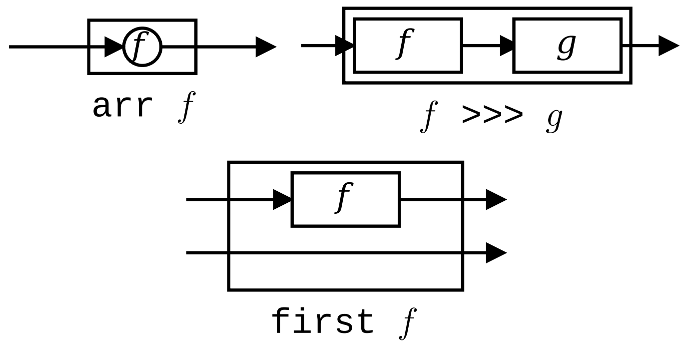
- Abstract data type interface for function-like types (or "blocks", if you prefer).
- Particularly suitable for types representing process-like computations.
- Related to ***monads***, since arrows are computations, but more general.
- Provides a minimal set of "wiring" combinators.

What is an arrow? (1)

- A ***type constructor*** a of arity two.
- Three operators:
  - ***lifting***:
    `arr :: (b->c) -> a b c`
  - ***composition***:
    `(>>>) :: a b c -> a c d -> a b d`
  - ***widening***:
    `first :: a b c -> a (b,d) (c,d)`
- A set of ***algebraic laws*** that must hold.

What is an arrow? (2)

These diagrams convey the general idea:



The `Arrow` class

In Haskell, a ***type class*** is used to capture these ideas (except for the laws):

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


Functions are arrows (1)

Functions are a simple example of arrows, with `(->)` as the arrow type constructor.

Exercise 5: Suggest suitable definitions of

- `arr`
- `(>>>)`
- `first`

for this case!

(We have not looked at what the laws are yet, but they are "natural".)

Functions are arrows (2)

Solution:

- `arr = id`
  To see this, recall
  ```
  id :: t -> t
  arr :: (b->c) -> a b c
  ```
  Instantiate with
  ```
  a = (->)
  t = b->c = (->) b c
  ```


Functions are arrows (3)

- `f >>> g = \a -> g (f a)` ***or***
- `f >>> g = g . f` ***or even***
- `(>>>) = flip (.)`
- `first f = \(b,d) -> (f b,d)`

Functions are arrows (4)

`Arrow` instance declaration for functions:

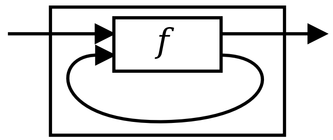
```
instance Arrow (->) where
    arr     = id
    (>>>)   = flip (.)
    first f = \(b,d) -> (f b,d)
```


Some arrow laws

```
(f >>> g) >>> h  =  f >>> (g >>> h)
  arr (f >>> g)  =  arr f >>> arr g
   arr id >>> f  =  f
              f  =  f >>> arr id
  first (arr f)  =  arr (first f)
first (f >>> g)  =  first f >>> first g
```


The `loop` combinator (1)

Another important operator is `loop`: a fixed-point operator used to express recursive arrows or ***feedback***:



`loop` f

The `loop` combinator (2)

Not all arrow instances support `loop`. It is thus a method of a separate class:

```
class Arrow a => ArrowLoop a where
    loop :: a (b, d) (c, d) -> a b c
```

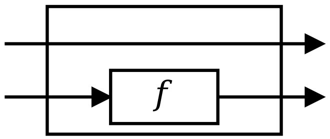
Remarkably, the four combinators `arr`, `>>>`, `first`, and `loop` are sufficient to express any conceivable wiring!

Some more arrow combinators (1)

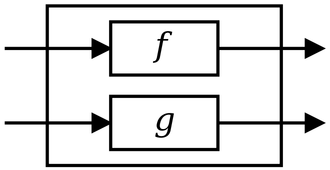
```
second :: Arrow a =>
    a b c -> a (d,b) (d,c)

(***) :: Arrow a =>
    a b c -> a d e -> a (b,d) (c,e)

(&&&) :: Arrow a =>
    a b c -> a b d -> a b (c,d)
```


Some more arrow combinators (2)

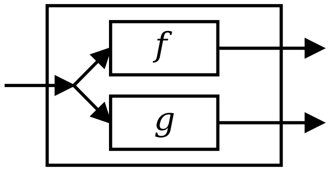
As diagrams:



second f



f *** g



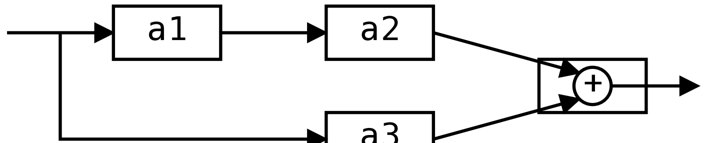
f &&& g

Some more arrow combinators (3)

```
second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
swap (x,y) = (y,x)

(***) :: Arrow a =>
    a b c -> a d e -> a (b,d) (c,e)
f *** g = first f >>> second g

(&&&) :: Arrow a => a b c -> a b d -> a b (c,d)
f &&& g = arr (\x->(x,x)) >>> (f *** g)
```


Exercise 6

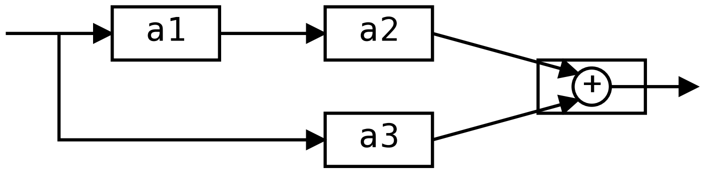
Describe the following circuit using arrow combinators:




```
a1, a2, a3 :: A Double Double
```

Exercise 6: One solution

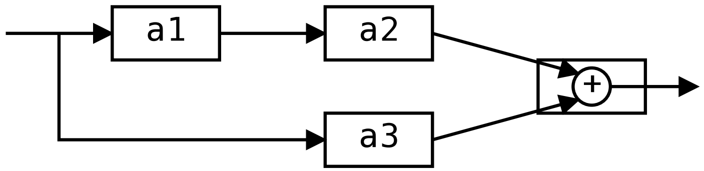
Exercise 3: Describe the following circuit using arrow combinators:




```
a1, a2, a3 :: A Double Double

circuit_v1 :: A Double Double
circuit_v1 = (a1 &&& arr id)
             >>> (a2 *** a3)
             >>> arr (uncurry (+))
```


Exercise 6: Another solution

Exercise 3: Describe the following circuit:




```
a1, a2, a3 :: A Double Double

circuit_v2 :: A Double Double
circuit_v2 = arr (\x -> (x,x))
             >>> first a1
             >>> (a2 *** a3)
             >>> arr (uncurry (+))
```


The arrow do notation (1)

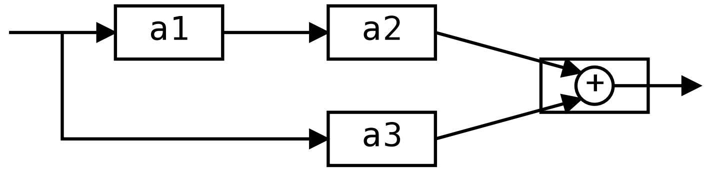
Ross Paterson's do-notation for arrows supports ***pointed*** arrow programming. Only ***syntactic sugar***.

proc pat -> do [rec]
  pat_1 <- $sfexp_1$ -< exp_1
  pat_2 <- $sfexp_2$ -< exp_2
  ...
  pat_n <- $sfexp_n$ -< exp_n
  returnA -< exp

Also: `let` pat = exp $\equiv$ pat `<- arr id -<` exp

The arrow do notation (2)

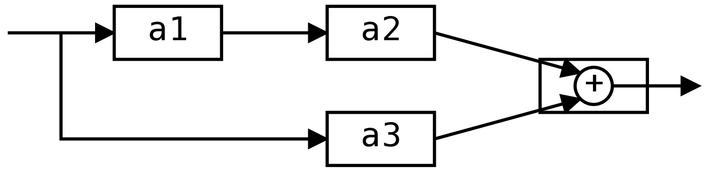
Let us redo exercise 3 using this notation:




```
circuit_v4 :: A Double Double
circuit_v4 = proc x -> do
    y1 <- a1 -< x
    y2 <- a2 -< y1
    y3 <- a3 -< x
    returnA -< y2 + y3
```

The arrow do notation (3)

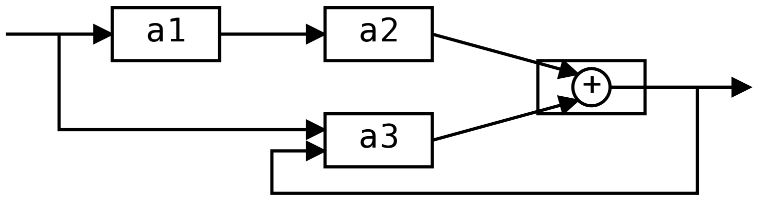
We can also mix and match:




```
circuit_v5 :: A Double Double
circuit_v5 = proc x -> do
    y2 <- a2 <<< a1 -< x
    y3 <- a3        -< x
    returnA -< y2 + y3
```


The arrow do notation (4)

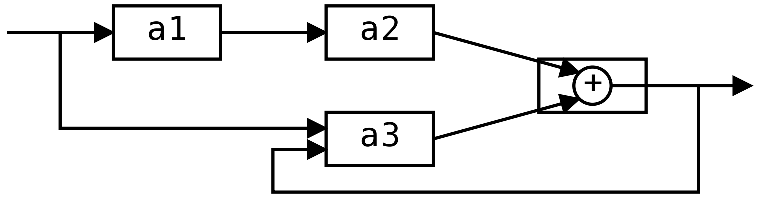
Recursive networks: do-notation:




```
a1, a2 :: A Double Double
a3 :: A (Double,Double) Double
```

Exercise 5: Describe this using only the arrow combinators.

The arrow do notation (5)




```
circuit = proc x -> do
    rec
        y1 <- a1 -< x
        y2 <- a2 -< y1
        y3 <- a3 -< (x, y)
        let y = y2 + y3
    returnA -< y
```


Arrows and Monads (1)

Arrows generalize monads: for every monad type there is an arrow, the ***Kleisli category*** for the monad:

```
newtype Kleisli m a b = K (a -> m b)

instance Monad m => Arrow (Kleisli m) where
    arr f       = K (\b -> return (f b))
    K f >>> K g = K (\b -> f b >>= g)
```

Arrows and Monads (2)

But not every arrow is a monad. However, arrows that support an additional `apply` operation ***are*** effectively monads:

```
apply :: Arrow a => a (a b c, b) c
```

Exercise 7: Verify that

```
newtype M b = M (A () b)
```

is a monad if A is an arrow supporting `apply`; i.e., define `return` and `bind` in terms of the arrow operations (and verify that the monad laws hold).

An application: FRP

Functional Reactive Programming (FRP):

- Paradigm for ***reactive programming*** in a functional setting:
  - Input arrives ***incrementally*** while system is running.
  - Output is generated in response to input in an interleaved and ***timely*** fashion.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.

Yampa

Yampa:

- The most recent Yale FRP implementation.
- ***Embedding*** in Haskell (a Haskell library).
- ***Arrows*** used as the basic structuring framework.
- ***Continuous time***.
- Discrete-time signals modelled by continuous-time signals and an option type.
- Advanced ***switching constructs*** allows for highly dynamic system structure.

Related languages

FRP related to:

- Synchronous languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink.

Distinguishing features of FRP:

- First class reactive components.
- Allows highly dynamic system structure.
- Supports hybrid (mixed continuous and discrete) systems.

FRP applications

Some domains where FRP has been used:

- Graphical Animation (Fran: Elliott, Hudak)
- Robotics (Frob: Peterson, Hager, Hudak, Elliott, Pembeci, Nilsson)
- Vision (FVision: Peterson, Hudak, Reid, Hager)
- GUIs (Fruit: Courtney)
- Hybrid modeling (Nilsson, Hudak, Peterson)

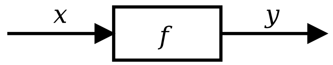
Yampa?

Yampa is a river with long calmly flowing sections and abrupt whitewater transitions in between.

A good metaphor for hybrid systems!

Signal functions

Key concept: ***functions on signals***.



Intuition:

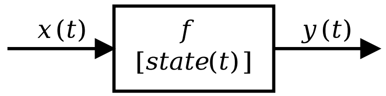
```
Signal α ≈ Time→α
x :: Signal T1
y :: Signal T2
SF α β ≈ Signal α →Signal β
f :: SF T1 T2
```

Additionally: ***causality*** requirement.

Signal functions and state

Alternative view:

Signal functions can encapsulate ***state***.



$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

Functions on signals are either:

- ***Stateful***: $y(t)$ depends on $x(t)$ and $state(t)$
- ***Stateless***: $y(t)$ depends only on $x(t)$

Yampa and Arrows

SF is an arrow. Signal function instances of core combinators:

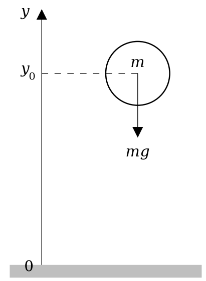
- `arr :: (a -> b) -> SF a b`
- `>>> :: SF a b -> SF b c -> SF a c`
- `first :: SF a b -> SF (a,c) (b,c)`
- `loop :: SF (a,c) (b,c) -> SF a b`

But `apply` has no useful meaning. Hence SF is ***not*** a monad.

Some further basic signal functions

- ```
  identity :: SF a a
  identity = arr id
  ```
- ```
  constant :: b -> SF a b
  constant b = arr (const b)
  ```
- `integral :: VectorSpace a s=>SF a a`
- ```
  time :: SF a Time
  time = constant 1.0 >>> integral
  ```
- ```
  (^<<) :: (b->c) -> SF a b -> SF a c
  f (^<<) sf = sf >>> arr f
  ```


Example: A bouncing ball



$$y = y_0 + \int v \, dt$$

$$v = v_0 + \int -9.81$$

On impact:

$$v = -v(t-)$$

(fully elastic collision)

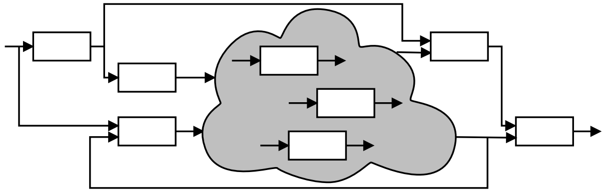
Part of a model of the bouncing ball

Free-falling ball:

```
type Pos = Double
type Vel = Double

fallingBall ::
    Pos -> Vel -> SF () (Pos, Vel)
fallingBall y0 v0 = proc () -> do
    v <- (v0 +) ^<< integral -< -9.81
    y <- (y0 +) ^<< integral -< v
    returnA -< (y, v)
```

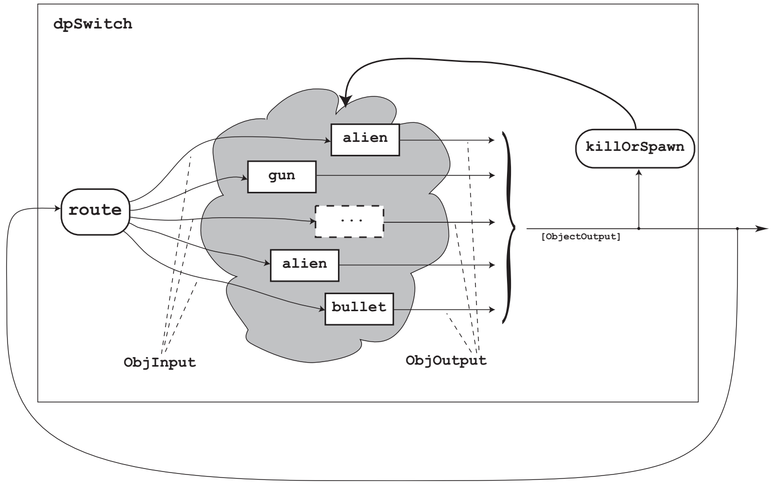
Dynamic system structure

Switching allows the structure of the system to evolve over time:



Example: Space Invaders

Overall game structure



Reading (1)

- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.
- Sheng Liang, Paul Hudak, Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, January 1995, San Francisco, California

Reading (2)

- Mauro Jaskelioff. Monatron: An Extensible Monad Transformer Library. In *Implementation of Functional Languages (IFL'08)*, 2008.
- Mauro Jaskelioff. Modular Monad Transformers. In *European Symposium on Programming (ESOP,09)*, 2009.

Reading (3)

- John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000
- John Hughes. Programming with arrows. In *Advanced Functional Programming*, 2004. To be published by Springer Verlag.
- Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 Haskell Workshop*, pp. 51–64, October 2002.

Reading (4)

- Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, 2002. LNCS 2638, pp. 159–187.
- Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, Uppsala, Sweden, 2003, pp 7–18.