MGS 2005
Functional Reactive Programming

Lecture 2: Yampa Basics

Henrik Nilsson

School of Computer Science and Information Technology
University of Nottingham, UK

Outline

- Recap
- Notes on yesterday's exerceises
- Point-free vs. pointed programming: the arrow do-notation
- Basic Yampa programming

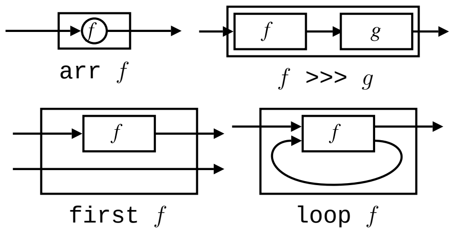
Recap: The arrow framework (1)

The following two Haskell type classes capture the notion of an arrow and of an arrow supporting feedback:

```
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)

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


Recap: The arrow framework (2)



`arr`, `>>>`, `first`, and `loop` are sufficient to express any conceivable "wiring"!

Recap: Further arrow combinators (1)

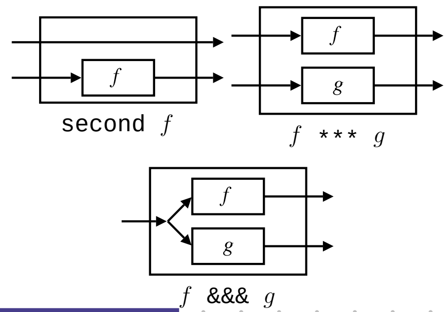
```
second :: Arrow a =>
    a b c -> a (d,b) (d,c)

(***) :: 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)
```

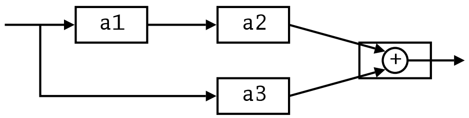

Recap: Further arrow combinators (2)

As diagrams:



Exercise 3: One solution

Exercise 3: Describe the following circuit using arrow combinators:

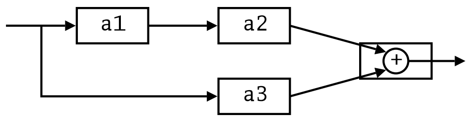



```
a1, a2, a3 :: A Double Double

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


Exercise 3: 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 (+))
```


Exercise 4: Solution

Exercise 4: Suggest definitions of `second`, `(***)`, and `(&&&)`.

```
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)
```

Note on the definition of (***) (1)

Are the following two definitions of (***) equivalent?

- `f *** g = first f >>> second g`
- `f *** g = second g >>> first f`

No, in general

$$\texttt{first}\ f \ggg \texttt{second}\ g \neq \texttt{second}\ g \ggg \texttt{first}\ f$$

since the ***order*** of the two possibly effectful computations f and g are different.

Note on the definition of (***) (2)

Similarly

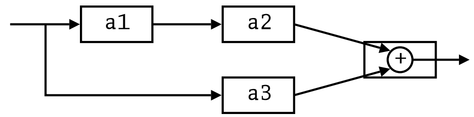
$$(f \ast\!\ast\!\ast\ g) \ggg (h \ast\!\ast\!\ast\ k) \neq (f \ggg h) \ast\!\ast\!\ast\ (g \ggg g)$$

since the order of f and g differs.

However, the following ***is*** true (an additional arrow law):

$$\begin{aligned} & \texttt{first}\ f \ggg \texttt{second}\ (\texttt{arr}\ g) \\ = \ & \texttt{second}\ (\texttt{arr}\ g) \ggg \texttt{first}\ f \end{aligned}$$

Yet an attempt at exercise 3




```
circuit_v3 :: A Double Double
circuit_v3 = (a1 &&& a3)
             >>> first a2
             >>> arr (uncurry (+))
```

Are `circuit_v1`, `circuit_v2`, and `circuit_v3` all equivalent?

Point-free vs. pointed programming

What we have seen thus far is an example of ***point-free*** programming: the values being manipulated are not given any names.

This is often appropriate, especially for small definitions, and it facilitates equational reasoning as shown by Bird & Meertens (Bird 1990).

However, large programs are much better expressed in a ***pointed*** style, where names can be given to values being manipulated.

The arrow do notation (1)

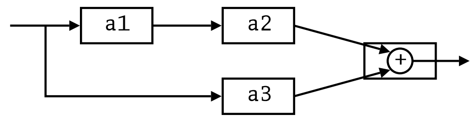
Ross Paterson's do-notation for arrows supports ***pointed*** arrow programming. Only ***syntactic sugar***.

$$\begin{aligned} & \texttt{proc}\ pat\ \texttt{->}\ \texttt{do}\ [\ \texttt{rec}\ ] \\ & \quad pat_1\ \texttt{<-}\ sfexp_1\ \texttt{-<}\ exp_1 \\ & \quad pat_2\ \texttt{<-}\ sfexp_2\ \texttt{-<}\ exp_2 \\ & \quad \ldots \\ & \quad pat_n\ \texttt{<-}\ sfexp_n\ \texttt{-<}\ exp_n \\ & \quad \texttt{returnA -<}\ exp \end{aligned}$$

Also: $\texttt{let}\ pat = exp \ \equiv\ pat\ \texttt{<- arr id -<}\ exp$

The arrow do notation (2)

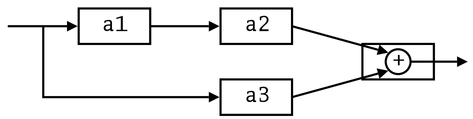
Let us redo exercise 3 using this notation:




```
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)

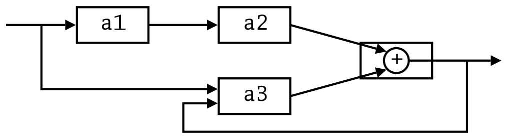
We can also mix and match:




```
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)

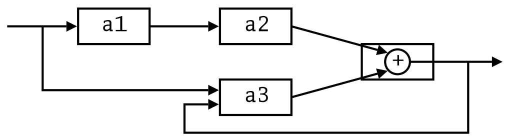
Exercise 5: Describe the following circuit using the arrow do-notation:




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

Exercise 6: As 5, but directly using only the arrow combinators.

Solution exercise 5



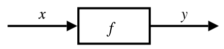

```
circuit = proc x -> do
    rec
        y1 <- a1 -< x
        y2 <- a2 -< y1
        y3 <- a3 -< (x, y)
        let y = y2 + y3
    returnA -< y
```

Some More Reading

- Richard S. Bird. A calculus of functions for program derivation. In *Research Topics in Functional Programming*, Addison-Wesley, 1990.
- Ross Paterson. A New Notation for Arrows. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, pp. 229–240, Firenze, Italy, 2001.

Recap: Signal functions (1)

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



Intuition:

$$
\begin{array}{l}
\texttt{Signal } \alpha \approx \texttt{Time} \rightarrow \alpha \\
\texttt{SF } \alpha\ \beta \approx \texttt{Signal } \alpha \rightarrow \texttt{Signal } \beta \\
x \ \texttt{:: Signal T1} \\
y \ \texttt{:: Signal T2} \\
f \ \texttt{:: SF T1 T2}
\end{array}
$$

SF is an instance of `Arrow` and `ArrowLoop`.

Recap: Signal functions (2)

Additionally, ***causality*** required: output at time t must be determined by input on interval $[0, t]$.

Signal functions are said to be

- ***pure*** or ***stateless*** if output at time t only depends on input at time t
- ***impure*** or ***stateful*** if output at time t depends on input over the interval $[0, t]$.

Some basic signal functions (1)

- `identity :: SF a a`
  `identity = arr id`
- `constant :: b -> SF a b`
  `constant b = arr (const b)`
- `integral :: VectorSpace a s=>SF a a`
  It is defined through:

$$y(t) = \int_0^t x(\tau)\, \mathrm{d}\tau$$

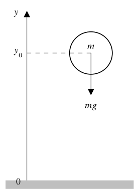
Some basic signal functions (2)

- `iPre :: a -> SF a a`
- `(^<<) :: (b->c) -> SF a b -> SF a c`
  `f (^<<) sf = sf >>> arr f`
- `time :: SF a Time`

  Quick Exercise: Define time!

  `time = constant 1.0 >>> integral`

A bouncing ball



$$
\begin{aligned}
y &= y_0 + \int v \,\mathrm{d}t \\
v &= v_0 + \int -9.81
\end{aligned}
$$

On impact:

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

(fully elastic collision)

Modelling the bouncing ball: part 1

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)
```


Events

Conceptually, ***discrete-time*** signals are only defined at discrete points in time, often associated with the occurrence of some ***event***.

Yampa models discrete-time signals by lifting the ***range*** of continuous-time signals:

`data Event a = NoEvent | Event a`

Discrete-time signal $=$ `Signal (Event` α`)`.

Associating information with an event occurrence:

`tag :: Event a -> b -> Event b`

Some basic event sources

- `never :: SF a (Event b)`
- `now :: b -> SF a (Event b)`
- `after :: Time -> b -> SF a (Event b)`
- `repeatedly ::`
  `Time -> b -> SF a (Event b)`
- `edge :: SF Bool (Event ())`

Stateful event suppression

- `notYet :: SF (Event a) (Event a)`
- `once :: SF (Event a) (Event a)`

Modelling the bouncing ball: part 2

Detecting when the ball goes through the floor:

```
fallingBall' ::
    Pos -> Vel
    -> SF () ((Pos,Vel), Event (Pos,Vel))
fallingBall' y0 v0 = proc () -> do
    yv@(y, _) <- fallingBall y0 v0 -< ()
    hit       <- edge              -< y <= 0
    returnA -< (yv, hit 'tag' yv)
```


Switching

Q: How and when do signal functions “start”?

A:
- ***Switchers*** “apply” a signal functions to its input signal at some point in time.
- This creates a “running” signal function ***instance***.
- The new signal function instance often replaces the previously running instance.

Switchers thus allow systems with ***varying structure*** to be described.

The basic switch (1)

Idea:

- Allows one signal function to be replaced by another.
- Switching takes place on the first occurrence of the switching event source.

```
switch ::
   SF a (b, Event c)
   -> (c -> SF a b)
   -> SF a b
```


The basic switch (2)

Exercise 7: Define an event counter `countFrom`

```
countFrom ::
     Int -> SF (Event a) Int
```

using

```
switch :: SF a (b, Event c)
            -> (c -> SF a b)
            -> SF a b
constant :: b -> SF a b
tag :: Event a -> b -> Event b
```


Solution exercise 7

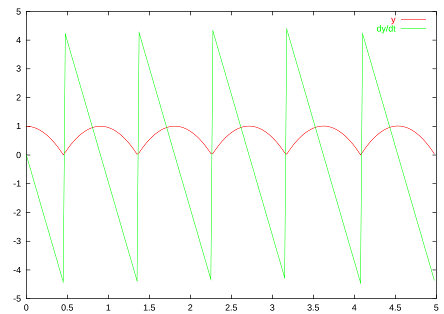
```
countFrom :: Int -> SF (Event a) Int
countFrom n =
    switch
        (constant n
         &&& arr (\e -> e 'tag' (n+1)))
        countFrom
```


Modelling the bouncing ball: part 3

Making the ball bounce:

```
bouncingBall :: Pos -> SF () (Pos, Vel)
bouncingBall y0 = bbAux y0 0.0
  where
    bbAux y0 v0 =
      switch (fallingBall' y0 v0) $ \(y,v) ->
      bbAux y (-v)
```


Simulation of bouncing ball



Modelling using impulses

From a modelling perspective, using a device like `switch` to model the interaction between the ball and the floor may seem rather unnatural.

A more appropriate account of what is going on is that an ***impulsive*** force is acting on the ball for a short time.

This can be abstracted into ***Dirac Impulses***: impulses that act instantaneously. See

> Henrik Nilsson. Functional Automatic Differentiation with Dirac Impulses. In *Proceedings of ICFP 2003*.