MGS 2007: ADV Lecture 3

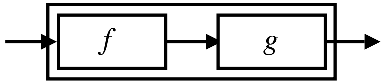
Arrows and Functional Reactive Programming

Henrik Nilsson

University of Nottingham, UK

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:

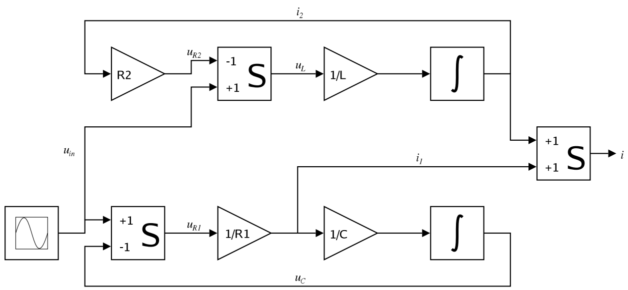


A *combinator* can be defined that captures this idea:

```
(>>>) :: 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:

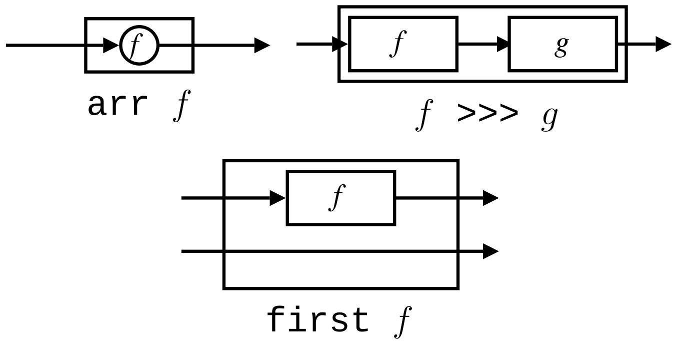
- Abstract data type interface for function-like types (or "blocks", if you prefer).
- Particularly suitable for types representing process-like computations.
- Related to ***monads***, since arrows are computations, but more general.
- Provides a minimal set of "wiring" combinators.

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 1: 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)
```

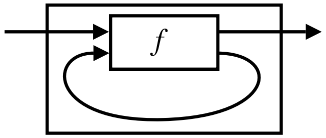

Some arrow laws

```
(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
```

Exercise 2: Draw diagrams illustrating the first and last law!

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)

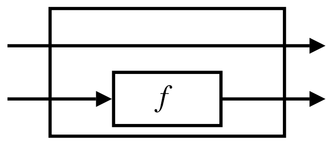
```
second :: Arrow a =>
    a b c -> a (d,b) (d,c)

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

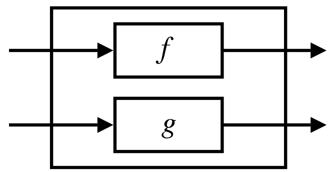
(&&&) :: Arrow a =>
    a b c -> a b d -> a b (c,d)
```


Some more arrow combinators (2)

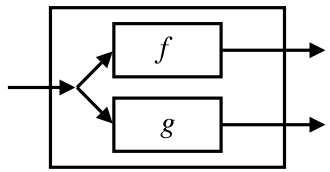
As diagrams:



`second` f



f `***` g



f `&&&` g

Some more arrow combinators (3)

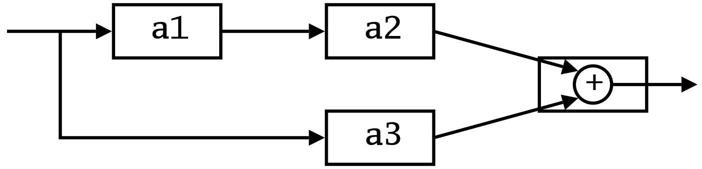
```
second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
swap (x,y) = (y,x)

(***) :: 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 3

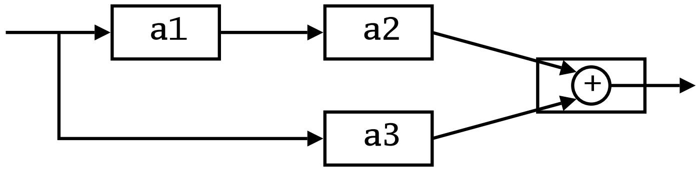
Describe the following circuit using arrow combinators:




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


Exercise 3: One solution

Exercise 3: Describe the following circuit using arrow combinators:

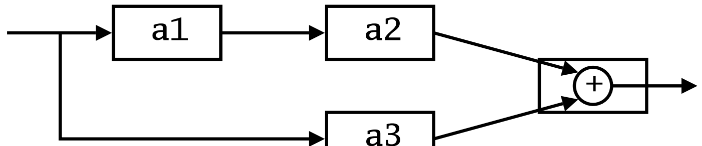



```
a1, a2, a3 :: A Double Double

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 (+))
```

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

first f >>> second g $\neq$ second g >>> first f

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

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

Similarly

$(f$ *** $g)$ >>> $(h$ *** $k)$ $\neq$ $(f$ >>> $h)$ *** $(g$ >>> $k)$

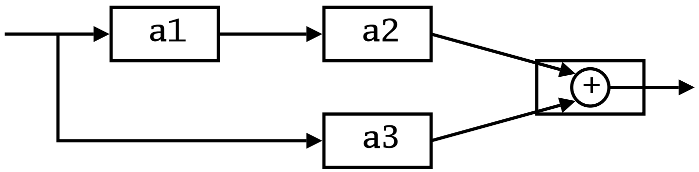
since the order of f and g differs.

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

first f >>> second (arr g)
$=$ second (arr g) >>> first f

However, for certain ***arrow instances*** equalites like the ones above do hold.

Yet an attempt at exercise 3




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

Exercise 4: Are `circuit_v1`, `circuit_v2`, and `circuit_v3` all equivalent?

The arrow do notation (1)

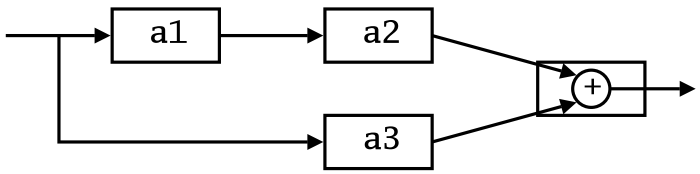
Ross Paterson's do-notation for arrows supports ***pointed*** arrow programming. Only ***syntactic sugar***.

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)

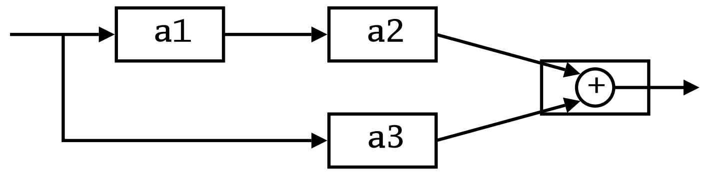
Let us redo exercise 3 using this notation:




```
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)

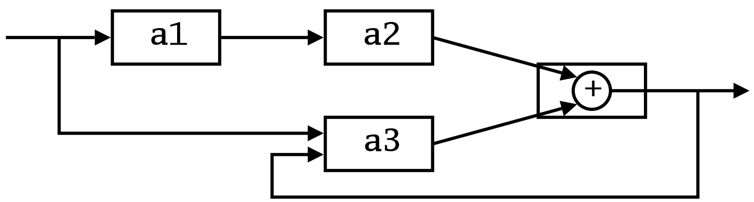
We can also mix and match:




```
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)

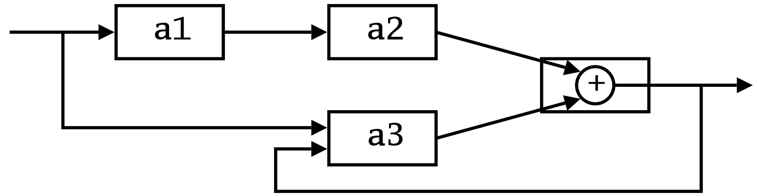
Recursive networks: do-notation:




```
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 6: 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)

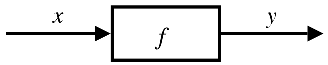
Yampa?

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:

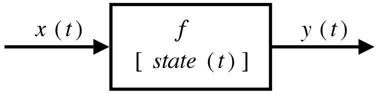
Signal $\alpha \approx$ Time$\to\alpha$
x :: Signal T1
y :: Signal T2
SF α β $\approx$ Signal α $\to$Signal β
f :: SF T1 T2

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:

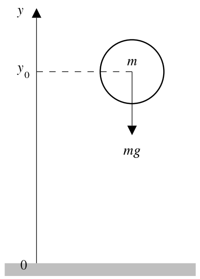
- `arr :: (a -> b) -> SF a b`
- `>>> :: SF a b -> SF b c -> SF a c`
- `first :: SF a b -> SF (a,c) (b,c)`
- `loop :: SF (a,c) (b,c) -> SF a b`

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



$$y = y_0 + \int v \, \mathrm{d}t$$

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

On impact:

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

(fully elastic collision)
  ```

Part of a model of the bouncing ball

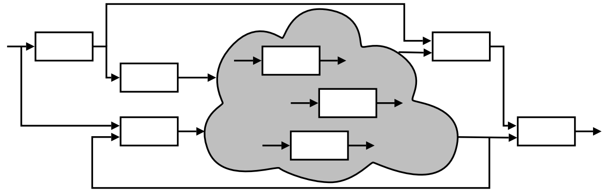
Free-falling ball:

```
type Pos = Double
type Vel = Double

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

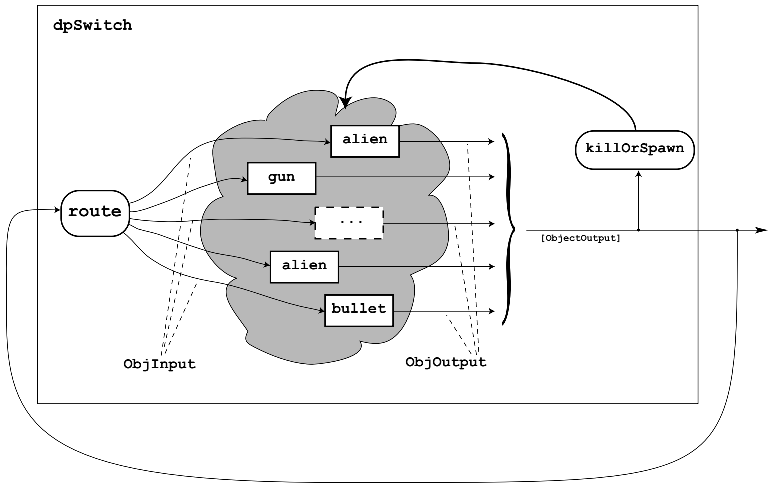

Dynamic system structure

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



Example: Space Invaders

Overall game structure



Reading

- 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 (2)

- 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.