LiU-FP2016: Lecture 13

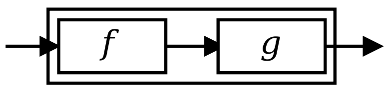
Arrows

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:

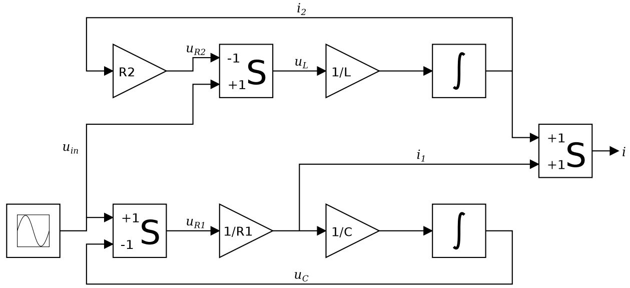


A *combinator* can be defined that captures this idea:

```
(>>>) :: 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:

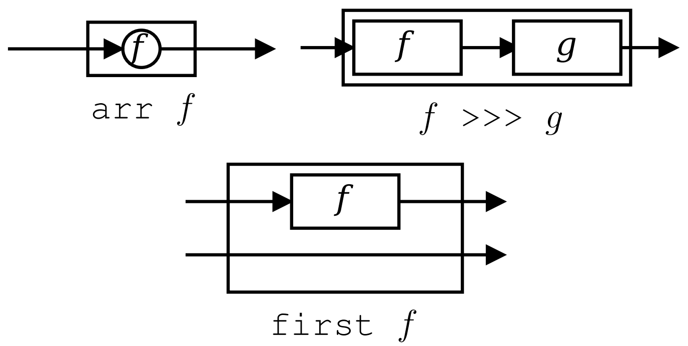
- Abstract data type interface for function-like types (or "blocks", if you prefer).
- Particularly suitable for types representing process-like computations.
- Related to ***monads***, since arrows are computations, but more general.
- Provides a minimal set of "wiring" combinators.

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:

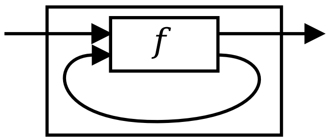
```
instance Arrow (->) where
    arr     = id
    (>>>)   = flip (.)
    first f = \(b,d) -> (f b,d)
```


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
```

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
```

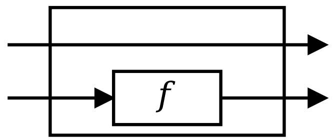
Remarkably, the four combinators `arr`, `>>>`, `first`, and `loop` are sufficient to express any conceivable wiring!

Some more arrow combinators (1)

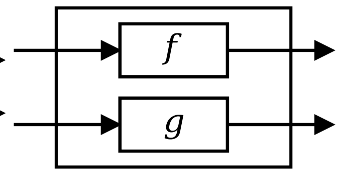
```
second :: Arrow a =>
    a b c -> a (d,b) (d,c)

(***) :: 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)

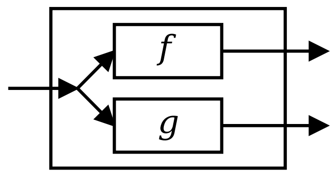
As diagrams:



`second` f



f `***` g



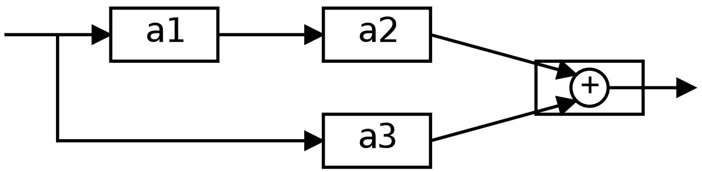
f `&&&` g

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 2

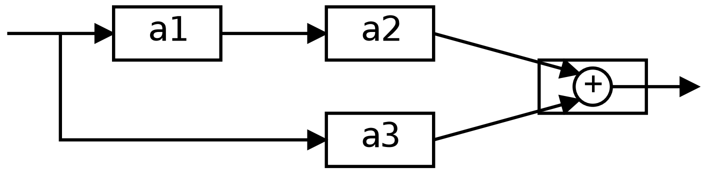
Describe the following circuit using arrow combinators:




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


Exercise 2: One solution

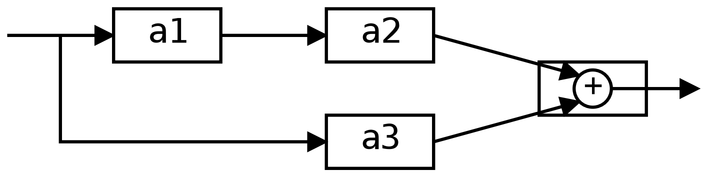
Exercise 2: Describe the following circuit using arrow combinators:




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

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


Exercise 2: Another solution

Exercise 2: 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)

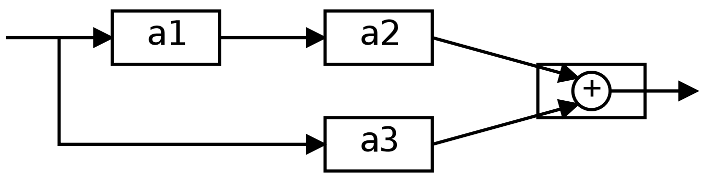
Ross Paterson's `do`-notation for arrows supports ***pointed*** arrow programming. Only ***syntactic sugar***.

```
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)

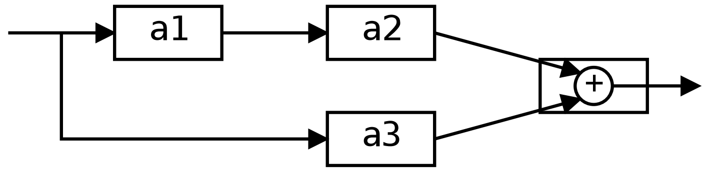
Let us redo exercise 2 using this notation:




```
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)

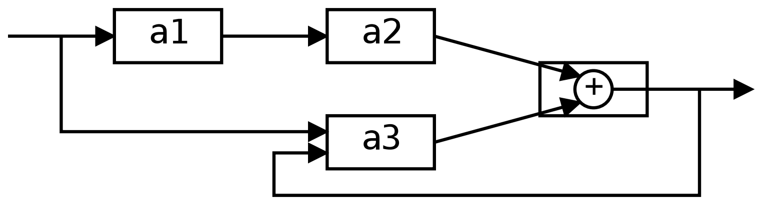
We can also mix and match:




```
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)

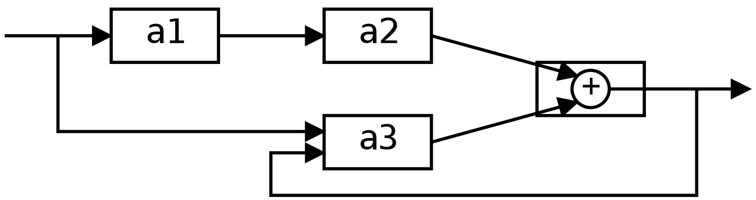
Recursive networks: `do`-notation:




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

Exercise 3: 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 4: 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).

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.