Arrows (1A)

Young Won Lim
5/4/19

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Young Won Lim
5/4/19

Based on

aaa

An arrow type

a monadic type m a :
a computation delivering an **a**

an arrow type a b c :
a computation with input of type **b** delivering a **c**

the application of the parameterised type **a**
to the two parameters **b** and **c**

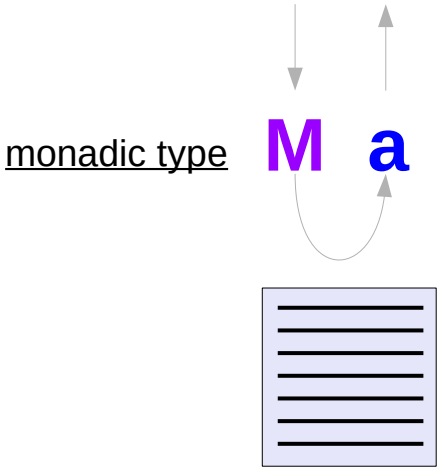
arrows make the dependence on **input** explicit

– John Hughes, Generalising Monads to Arrows [1]
Science of Computer Programming 37 (2000)
www.elsevier.nl/locate/scicoGeneralising monads to arrowsJohn Hughes

https://en.wikibooks.org/wiki/Haskell/Understanding_arrows

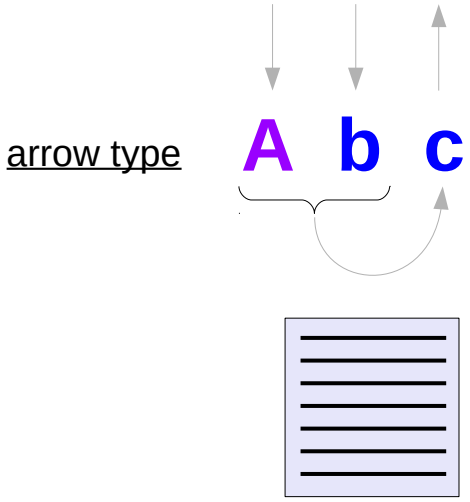
Monadic and Arrow types

computations resulting in output value



Monad

Computations with an input resulting in output value



Arrow

https://en.wikibooks.org/wiki/Haskell/Understanding_arrows

The Arrow

Arrow a b c

represents **a** process that
takes as input something of type **b** and
outputs something of type **c**.

the application of the parameterised type **a**
to the two parameters **b** and **c**

a b c

Function application

a ::Arrow

https://wiki.haskell.org/Arrow_tutorial

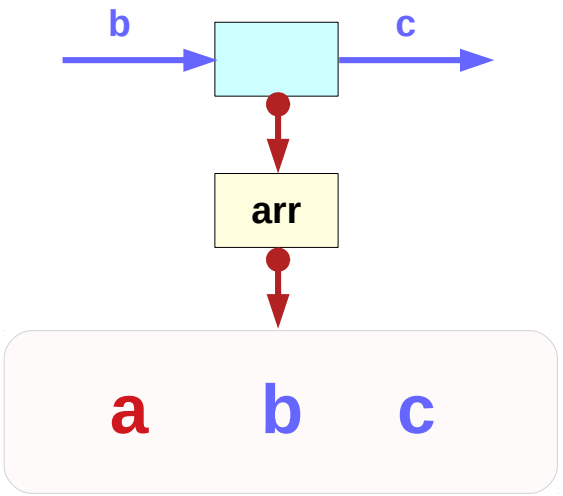
arr

arr builds an arrow out of a **function**.
This function is arrow-specific.

arr :: (Arrow a) => (b -> c) -> a b c

A computation a takes inputs of some type b and
produces outputs of another type c.

Each function (b -> c) may be treated as a computation



Function application

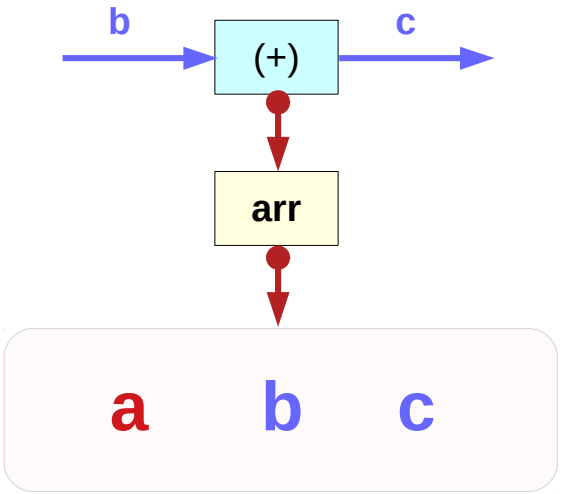
a ::Arrow

https://wiki.haskell.org/Arrow_tutorial

The Arrow

```
Prelude> import Control.Arrow
Prelude Control.Arrow> let a1 = arr (+)
Prelude Control.Arrow> :t a1
a1 :: (Arrow a1, Num a) => a1 a (a -> a)
Prelude Control.Arrow> a1 3 4
7


Prelude Control.Arrow> let a2 = arr (+3)
Prelude Control.Arrow> a2 4
7
Prelude Control.Arrow> :t a2
a2 :: (Arrow a, Num c) => a c c
```



Function application

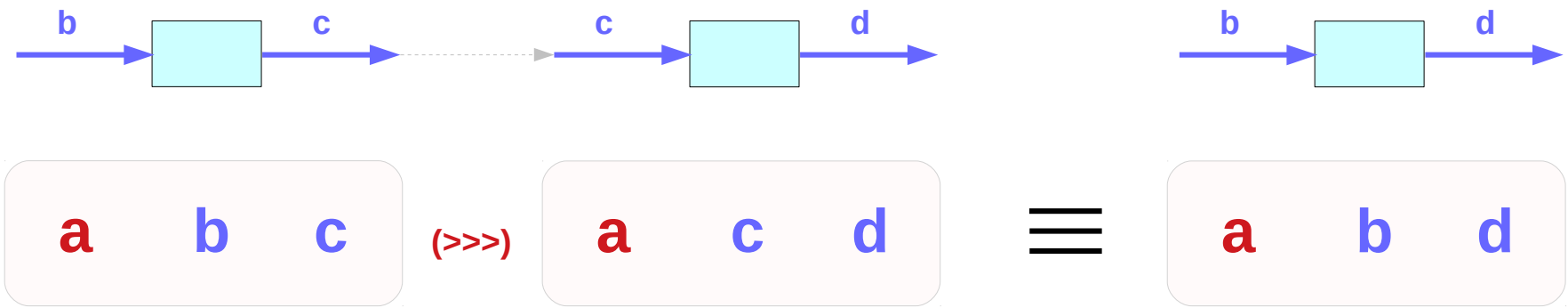
a ::Arrow

http://tuttlem.github.io/2014/07/26/practical-arrow-usage.html

The Arrow composition

Arrow composition is achieved with **(>>>)**.
This takes two **arrows** and **chains** them together,
one after another.
It is also arrow- specific.

(>>>) :: (Arrow a) => a b c -> a c d -> a b d



https://wiki.haskell.org/Arrow_tutorial

first and second

First and second make a new arrow out of an existing arrow.

They perform a transformation (given by their argument)
on either the first or the second item of a pair.
These definitions are arrow-specific.

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

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

https://wiki.haskell.org/Arrow_tutorial

first and second

data STRef s a

a value of type **STRef s a** is a mutable variable in state thread **s**, containing a value of type **a**

```
>>> :{
runST (do
   ref <- newSTRef "hello"
   x <- readSTRef ref
   writeSTRef ref (x ++ "world")
   readSTRef ref )
:}
"helloworld"
```

https://wiki.haskell.org/Arrow_tutorial

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf