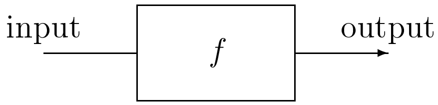
Arrows and Computations
1. Introduction

Ross Paterson
City University, London

9th November 2001

Notions of computation

Many programs involve “function-like” components:



$\mathit{input} \to \mathit{output}$	pure functions
$(\mathit{State}, \mathit{input}) \to (\mathit{State}, \mathit{output})$	state threading
$\mathit{input} \to [\mathit{output}]$	nondeterminism
$[\mathit{input}] \to [\mathit{output}]$	stream processors
...	

What do they have in common? [Hughes, 2000]

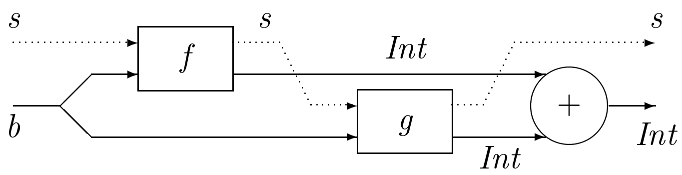
Test case: generalize

$$\mathit{add} :: (b \to \mathit{Int}) \to (b \to \mathit{Int}) \to b \to \mathit{Int}$$
$$\mathit{add}\ f\ g\ b = f\ b + g\ b$$

Adding the results of two state transformers

Instead of $b \to c$, use

$$\textbf{type}\ \mathit{StateTrans}\ s\ b\ c = (s, b) \to (s, c)$$



$$\mathit{addST} :: \mathit{StateTrans}\ s\ b\ \mathit{Int} \to \mathit{StateTrans}\ s\ b\ \mathit{Int} \to \mathit{StateTrans}\ s\ b\ \mathit{Int}$$
$$\begin{aligned} \mathit{addST}\ f\ g\ (s, b) = \textbf{let}\ & (s', x) = f\ (s, b) \\ & (s'', y) = g\ (s', b) \\ \textbf{in}\ & (s'', x + y) \end{aligned}$$

Backtracking

Represent nondeterministic computations (e.g. searches) as functions to lists.

$$\textbf{type}\ \mathit{Backtrack}\ b\ c = b \to [c]$$

Lift + by trying every combination:

$$\mathit{addBT} :: \mathit{Backtrack}\ b\ \mathit{Int} \to \mathit{Backtrack}\ b\ \mathit{Int} \to \mathit{Backtrack}\ b\ \mathit{Int}$$
$$\mathit{addBT}\ f\ g\ b = [x + y \mid x \leftarrow f\ b, y \leftarrow g\ b]$$

Or equivalently

$$\mathit{addBT}\ f\ g\ b = \mathit{concat}\ (\mathit{map}\ (\lambda x \to \mathit{map}\ (\lambda y \to x + y)\ (g\ b))\ (f\ b))$$

Stream processors

Kahn networks: use infinite lists to represent all the values that pass through a channel.

$$\textbf{type}\ \mathsf{StreamProc}\ b\ c = [b] \to [c]$$

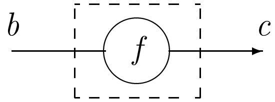
Lifting $+$ to infinite lists:

$$\mathsf{addSP} :: \mathsf{StreamProc}\ b\ \mathsf{Int} \to \mathsf{StreamProc}\ b\ \mathsf{Int} \to \mathsf{StreamProc}\ b\ \mathsf{Int}$$
$$\mathsf{addSP}\ f\ g\ bs = \mathsf{map}\ (\lambda(x, y) \to (x + y))\ (\mathsf{zip}\ (f\ bs)\ (g\ bs))$$

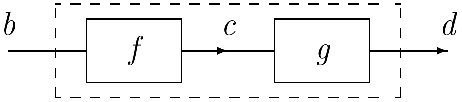
Part of the arrow class

class $\mathit{Arrow}\ a$ **where**

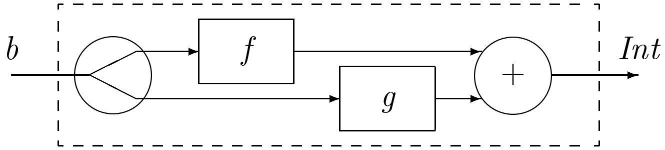
$$\mathsf{pure} :: (b \to c) \to a\ b\ c$$



$$(\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d$$



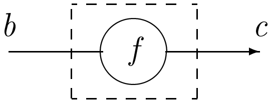
but this is not enough to write an add combinator:



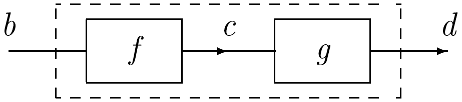
The arrow class

class $\mathit{Arrow}\ a$ **where**

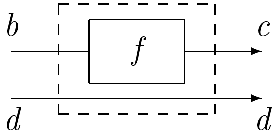
$$\mathsf{pure} :: (b \to c) \to a\ b\ c$$



$$(\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d$$

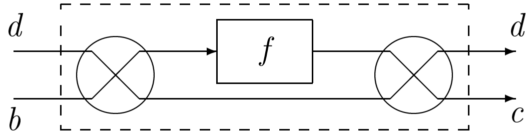


$$\mathsf{first} :: a\ b\ c \to a\ (b, d)\ (c, d)$$



There is no need to assume *second*

$$\mathsf{second} :: \mathsf{Arrow}\ a \Rightarrow a\ b\ c \to a\ (d, b)\ (d, c)$$
$$\mathsf{second}\ f = \mathsf{pure}\ \mathit{swap} \ggg \mathsf{first}\ f \ggg \mathsf{pure}\ \mathit{swap}$$
$$\textbf{where}\ \mathit{swap}\tilde{}(x, y) = (y, x)$$



Some instances

Pure functions

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Arrow}\ (\to)\ \textbf{where}\\
&\qquad \mathsf{pure}\ f = f\\
&\qquad f \ggg g = g \cdot f\\
&\qquad \mathsf{first}\ f\ (x, y) = (f\ x, y)
\end{aligned}
$$

State transformers

$$
\textbf{newtype}\ \mathsf{StateTrans}\ s\ b\ c = \mathsf{ST}\ ((s, b) \to (s, c))
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Arrow}\ (\mathsf{StateTrans}\ s)\ \textbf{where}\\
&\qquad \mathsf{pure}\ f = \mathsf{ST}\ (\lambda(s, b) \to (s, f\ b))\\
&\qquad \mathsf{ST}\ f \ggg \mathsf{ST}\ g = \mathsf{ST}\ (g \cdot f)\\
&\qquad \mathsf{first}\ (\mathsf{ST}\ f) = \mathsf{ST}\ (\lambda(s, (b, d)) \to\\
&\qquad\qquad \textbf{let}\ (s', c) = f\ (s, b)\\
&\qquad\qquad \textbf{in}\ (s', (c, d)))
\end{aligned}
$$

Backtracking

$$
\textbf{newtype}\ \mathsf{Backtrack}\ b\ c = \mathsf{BT}\ (b \to [c])
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Arrow}\ \mathsf{Backtrack}\ \textbf{where}\\
&\qquad \mathsf{pure}\ f = \mathsf{BT}\ (\lambda b \to [f\ b])\\
&\qquad \mathsf{BT}\ f \ggg \mathsf{BT}\ g = \mathsf{BT}\ (\lambda b \to [y \mid x \leftarrow f\ b, y \leftarrow g\ x])\\
&\qquad \mathsf{first}\ (\mathsf{BT}\ f) = \mathsf{BT}\ (\lambda(b, d) \to [(c, d) \mid c \leftarrow f\ b])
\end{aligned}
$$

Stream processors

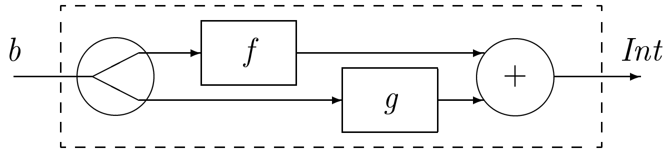
$$
\textbf{newtype}\ \mathsf{StreamProc}\ b\ c = \mathsf{SP}\ ([b] \to [c])
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Arrow}\ \mathsf{StreamProc}\ \textbf{where}\\
&\qquad \mathsf{pure}\ f = \mathsf{SP}\ (\mathsf{map}\ f)\\
&\qquad \mathsf{SP}\ f \ggg \mathsf{SP}\ g = \mathsf{SP}\ (g \cdot f)\\
&\qquad \mathsf{first}\ (\mathsf{SP}\ f) = \mathsf{SP}\ (\lambda bds \to\\
&\qquad\qquad \mathit{zip}\ (f\ (\mathsf{map}\ \mathsf{fst}\ bds))\ (\mathsf{map}\ \mathsf{snd}\ bds))
\end{aligned}
$$

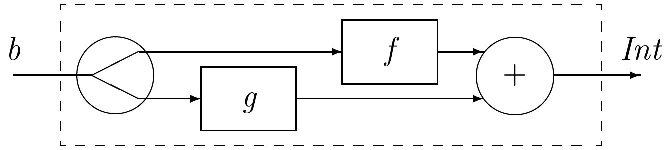
$$
\begin{aligned}
&\mathsf{delay} :: b \to \mathsf{StreamProc}\ b\ b\\
&\mathsf{delay}\ b = \mathsf{SP}\ (b{:})
\end{aligned}
$$

General add combinators

$$
\begin{aligned}
&\mathsf{addA} :: \mathit{Arrow}\ a \Rightarrow a\ b\ \mathsf{Int} \to a\ b\ \mathsf{Int} \to a\ b\ \mathsf{Int}\\
&\mathsf{addA}\ f\ g = \mathsf{pure}\ (\lambda b \to (b, b)) \ggg\\
&\qquad\qquad \mathsf{first}\ f \ggg \mathsf{second}\ g \ggg \mathsf{pure}\ (\lambda(x, y) \to x + y)
\end{aligned}
$$



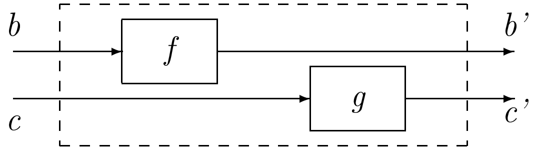
$$
\begin{aligned}
&\mathsf{addA}' :: \mathit{Arrow}\ a \Rightarrow a\ b\ \mathsf{Int} \to a\ b\ \mathsf{Int} \to a\ b\ \mathsf{Int}\\
&\mathsf{addA}'\ f\ g = \mathsf{pure}\ (\lambda b \to (b, b)) \ggg\\
&\qquad\qquad \mathsf{second}\ g \ggg \mathsf{first}\ f \ggg \mathsf{pure}\ (\lambda(x, y) \to x + y)
\end{aligned}
$$



Derived combinators

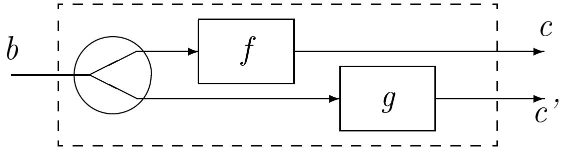
$(\ast\!\!\ast\!\!\ast) :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \to a\ b'\ c' \to a\ (b, b')\ (c, c')$

$f \ast\!\!\ast\!\!\ast\ g = \mathrm{first}\ f \ggg \mathrm{second}\ g$



$(\&\&\&) :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \to a\ b\ c' \to a\ b\ (c, c')$

$f\ \&\&\&\ g = \mathrm{pure}\ (\lambda b \to (b, b)) \ggg f \ast\!\!\ast\!\!\ast\ g$



$\mathrm{addA}\ f\ g = f\ \&\&\&\ g \ggg \mathrm{pure}\ (\lambda(x, y) \to x + y)$

Static properties

data $\mathrm{Count}\ a\ b\ c = \mathrm{Count}\ \mathrm{Int}\ (a\ b\ c)$

instance $\mathit{Arrow}\ a \Rightarrow \mathit{Arrow}\ (\mathrm{Count}\ a)$ **where**

$\quad \mathrm{pure}\ f = \mathrm{Count}\ 0\ (\mathrm{pure}\ f)$

$\quad \mathrm{Count}\ n1\ f1 \ggg \mathrm{Count}\ n2\ f2 =$

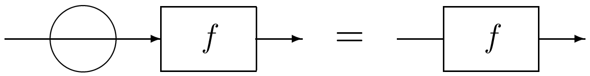
$\qquad \mathrm{Count}\ (n1 + n2)\ (f1 \ggg f2)$

$\quad \mathrm{first}\ (\mathrm{Count}\ n\ f) = \mathrm{Count}\ n\ (\mathrm{first}\ f)$

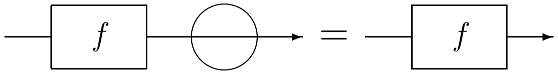
More complex example: Swierstra and Duponcheel's parser combinators.

Arrow axioms

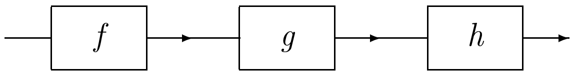
$\mathrm{pure}\ id \ggg f \quad = \quad f$



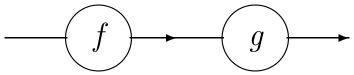
$f \ggg \mathrm{pure}\ id \quad = \quad f$



$(f \ggg g) \ggg h \quad = \quad f \ggg (g \ggg h)$

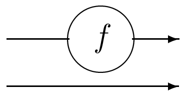


$\mathrm{pure}\ (g \cdot f) \quad = \quad \mathrm{pure}\ f \ggg \mathrm{pure}\ g$

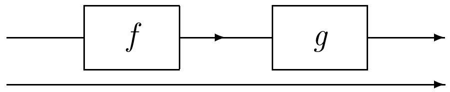


Axioms of first

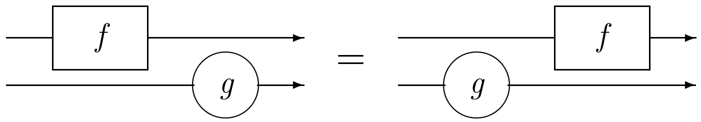
$\mathrm{first}\ (\mathrm{pure}\ f) \quad = \quad \mathrm{pure}\ (f \times id)$



$\mathrm{first}\ (f \ggg g) \quad = \quad \mathrm{first}\ f \ggg \mathrm{first}\ g$



$\mathrm{first}\ f \ggg \mathrm{pure}\ (id \times g) \quad = \quad \mathrm{pure}\ (id \times g) \ggg \mathrm{first}\ f$



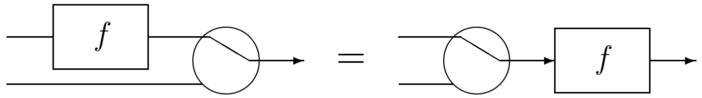
for the functor

$(\times) :: (a \to a') \to (b \to b') \to (a, b) \to (a', b')$

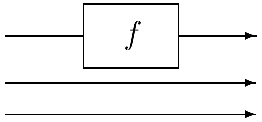
$(f \times g)\ (a, b) = (f\ a, g\ b)$

Elimination and associativity

$$\mathsf{first}\ f \ggg \mathsf{pure}\ \mathsf{fst} \quad = \quad \mathsf{pure}\ \mathsf{fst} \ggg f$$



$$\mathsf{first}\ (\mathsf{first}\ f) \ggg \mathsf{pure}\ \mathsf{assoc} \quad = \quad \mathsf{pure}\ \mathsf{assoc} \ggg \mathsf{first}\ f$$



for the function

$$\mathsf{assoc} :: ((a, b), c) \to (a, (b, c))$$
$$\mathsf{assoc}\ ((a, b), c) = (a, (b, c))$$

Exercises

class $\mathit{Arrow}\ a$ **where**
$$\mathsf{pure} :: (b \to c) \to a\ b\ c$$
$$(\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d$$
$$\mathsf{first} :: a\ b\ c \to a\ (b, d)\ (c, d)$$

Write Arrow instances for the following:

newtype $\mathit{Writer}\ b\ c = W\ (b \to (\mathsf{String}, c))$

newtype $\mathit{MapTrans}\ s\ b\ c = \mathsf{MT}\ ((s \to b) \to (s \to c))$

newtype $\mathsf{CPS}\ r\ b\ c = \mathsf{CPS}\ ((c \to r) \to (b \to r))$

newtype $\mathsf{Auto}\ b\ c = A\ (b \to (c, \mathsf{Auto}\ b\ c))$

Output as an arrow

newtype $\mathit{Writer}\ b\ c = W\ (b \to (\mathsf{String}, c))$

$$\mathit{write} :: \mathsf{Show}\ b \Rightarrow \mathit{Writer}\ b\ ()$$
$$\mathit{write} = W\ (\lambda b \to (\mathit{show}\ b, ()))$$

instance $\mathit{Arrow}\ \mathit{Writer}$ **where**
$$\mathsf{pure}\ f = W\ (\lambda b \to ([\,], f\ b))$$
$$W\ f \ggg W\ g = W\ (\lambda b \to$$
$$\quad \mathbf{let}\ (s1, c) = f\ b$$
$$\quad\quad (s2, d) = g\ c$$
$$\quad \mathbf{in}\ (s1 +\!\!+\ s2, d))$$
$$\mathsf{first}\ (W\ f) = W\ (\lambda (b, d) \to$$
$$\quad \mathbf{let}\ (s, c) = f\ b$$
$$\quad \mathbf{in}\ (s, (c, d)))$$

Map transformers

newtype $\mathit{MapTrans}\ s\ b\ c = \mathsf{MT}\ ((s \to b) \to (s \to c))$

If s represents time: behaviour transformers.

If s represents place: data parallel computation.

instance $\mathit{Arrow}\ (\mathit{MapTrans}\ s)$ **where**
$$\mathsf{pure}\ f = \mathsf{MT}\ (\lambda h \to f \cdot h)$$
$$\mathsf{MT}\ f \ggg \mathsf{MT}\ g = \mathsf{MT}\ (g \cdot f)$$
$$\mathsf{first}\ (\mathsf{MT}\ f) = \mathsf{MT}\ (\lambda h\ s \to (f\ (\mathsf{fst} \cdot h)\ s, \mathsf{snd}\ (h\ s)))$$

Continuation-passing style

```
newtype CPS r b c = CPS ((c → r) → (b → r))

instance Arrow (CPS r) where
    pure f = CPS (λk → k · f)
    CPS f ⋙ CPS g = CPS (f · g)
    first (CPS f) = CPS (λk → λ(b, d) → f (λc → k (c, d)) b)
```


Simple automata

```
newtype Auto b c = A (b → (c, Auto b c))

instance Arrow Auto where
    pure f = A (λb → (f b, pure f))
    A f ⋙ A g = A (λb → let (c, f′) = f b
                            (d, g′) = g c
                        in (d, f′ ⋙ g′))
    first (A f) = A (λ(b, d) → let (c, f′) = f b
                               in ((c, d), first f′))

delay :: b → Auto b b
delay b = A (λb′ → (b, delay b′))
```


Conditionals

```
data Either a b = Left a | Right b

class Arrow a ⇒ ArrowChoice a where
    left :: a b c → a (Either b d) (Either c d)

instance ArrowChoice (→) where
    left f (Left x) = Left (f x)
    left f (Right y) = Right y

instance ArrowChoice (StateTrans s) where
    left (ST f) = ST (λ(s, x) → case x of
                        Left b → let (s′, c) = f (s, b)
                                 in (s′, Left c)
                        Right d → (s, Right d))
```


Auxilliary definitions

```
right :: ArrowChoice a ⇒ a b c → a (Either d b) (Either d c)
right f = pure mirror ⋙ left f ⋙ pure mirror
        where mirror (Left x) = Right x
              mirror (Right y) = Left y

(+++) :: ArrowChoice a ⇒ a b c → a b′ c′ →
        a (Either b b′) (Either c c′)
f +++ g = left f ⋙ right g

(|||) :: ArrowChoice a ⇒ a b d → a c d → a (Either b c) d
f ||| g = f +++ g ⋙ pure untag
        where untag (Left x) = x
              untag (Right y) = y
```

$ArrowChoice$ instances

instance $ArrowChoice$ Backtrack **where**
  left (BT f) = BT (λx $\to$ **case** x **of**
    Left b $\to$ map Left (f b)
    Right d $\to$ [Right d])

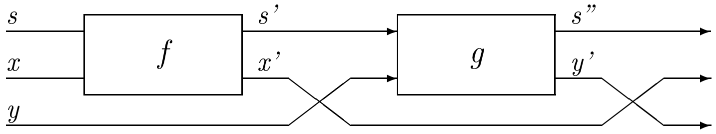
instance $ArrowChoice$ a $\Rightarrow$ $ArrowChoice$ (Count a) **where**
  left (Count n f) = Count n (left f)

instance $ArrowChoice$ Auto **where**
  left (A f) = A (λx $\to$ **case** x **of**
    Left b $\to$ **let** (c, f′) = f b
      in (Left c, left f′)
    Right d $\to$ (Right d, left (A f)))

Next week: arrow notation

- Arrows present a usefully general interface to computation.
- But they force a point-free style:

$$\begin{aligned}\mathrm{prodA}\ f\ g = {} & \mathrm{pure}\ (\lambda(s,(x,y)) \to ((s,x),y)) \ggg \mathrm{first}\ f \ggg \\ & \mathrm{pure}\ (\lambda((s',x'),y) \to ((s',y),x')) \ggg \mathrm{first}\ g \ggg \\ & \mathrm{pure}\ (\lambda((s'',y'),x') \to (s'',(x',y')))\end{aligned}$$



- Solution: a term notation for arrows.

Hyper-functions

Krstić, Launchbury and Pavlović [FoSSaCS 2001]:

newtype Hyper b c = H (Hyper c b $\to$ c)

instance $Arrow$ Hyper **where**
  pure f = H (λ(H k) $\to$ f (k (pure f)))
  f $\ggg$ H g = H (λk $\to$ g (k $\ggg$ f))
  first (H f) = H (λ(H k) $\to$
    (f (H (λk′ $\to$ fst (k (first k′)))),
    snd (k (first (H f)))))