Advanced Functional Programming (4)
Monad Transformers and Arrows

Bastiaan Heeren

Center for Software Technology, Universiteit Utrecht
http://www.cs.uu.nl/groups/ST/

February 14, 2007

Overview

1. Monads (revisited)
2. Monad Transformers
3. Arrows
   - Pairs
   - Conditionals
   - Feedback
   - Higher-order arrows
   - Example: Fibonacci
   - Conclusion
4. Agenda

Monads

- It is useful to think of a monad as a strategy for combining computations into more complex computations.
- Two sorts:
  - Referentially transparent interface to imperative operations (IO and ST monad)
  - Structure purely functional code (e.g., Maybe, list, and state monad)
- By making a type an instance of the Monad type class, you get many useful functions for free

Monads (2)

- The Monad type class is defined as follows:

Prelude.hs

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
  (>>)   :: m a → m b → m b
  fail   :: String → m a

    -- Minimal complete definition: (>>=), return
  p >> q = p >>= λ_ → q
  fail s = error s
```

- All instances should respect the three monad laws
- The hierarchical libraries contain many useful instances

Defining a State Monad

$$\textbf{data}\ \mathit{State}\ s\ a = S\{\mathit{unS} :: s \to (a, s)\}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ (\mathit{State}\ s)\ \textbf{where}\\
&\quad \mathit{return}\ a = S\ (\lambda s \to (a, s))\\
&\quad m \gg\!= f\ \ = S\ (\lambda s \to \textbf{let}\ (a, s') = \mathit{unS}\ m\ s\\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{unS}\ (f\ a)\ s')
\end{aligned}$$

Defining a State Monad

$$\textbf{data}\ \mathit{State}\ s\ a = S\{\mathit{unS} :: s \to (a, s)\}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ (\mathit{State}\ s)\ \textbf{where}\\
&\quad \mathit{return}\ a = S\ (\lambda s \to (a, s))\\
&\quad m \gg\!= f\ \ = S\ (\lambda s \to \textbf{let}\ (a, s') = \mathit{unS}\ m\ s\\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{unS}\ (f\ a)\ s')
\end{aligned}$$

We can “run” a computation in the state monad by providing an initial state.

$$\begin{aligned}
&\mathit{runState} :: \mathit{State}\ s\ a \to s \to (a, s)\\
&\mathit{runState} = \mathit{unS}\\
&\\
&\mathit{evalState}\ m\ \ = \mathit{fst}\ \ \circ \mathit{runState}\ m\\
&\mathit{execState}\ m = \mathit{snd} \circ \mathit{runState}\ m
\end{aligned}$$

Defining a State Monad (2)

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \rightarrow s\ \textbf{where} \\
\quad \mathit{get} :: m\ s \\
\quad \mathit{put} :: s \rightarrow m\ () \\
\\
\textbf{instance}\ \mathit{MonadState}\ s\ (\mathit{State}\ s)\ \textbf{where} \\
\quad \mathit{get}\ \ \ = S\ (\lambda s \rightarrow (s, s)) \\
\quad \mathit{put}\ s = S\ (\lambda\_ \rightarrow ((), s))
\end{array}
$$

Defining a State Monad (2)

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \rightarrow s\ \textbf{where} \\
\quad \mathit{get} :: m\ s \\
\quad \mathit{put} :: s \rightarrow m\ () \\
\\
\textbf{instance}\ \mathit{MonadState}\ s\ (\mathit{State}\ s)\ \textbf{where} \\
\quad \mathit{get}\ \ \ = S\ (\lambda s \rightarrow (s, s)) \\
\quad \mathit{put}\ s = S\ (\lambda\_ \rightarrow ((), s))
\end{array}
$$

Two useful functions:

$$
\begin{array}{l}
\mathit{modify} :: \mathit{MonadState}\ s\ m \Rightarrow (s \rightarrow s) \rightarrow m\ () \\
\mathit{modify}\ f = \textbf{do}\ s \leftarrow \mathit{get}; \mathit{put}\ (f\ s) \\
\\
\mathit{gets} :: \mathit{MonadState}\ s\ m \Rightarrow (s \rightarrow a) \rightarrow m\ a \\
\mathit{gets}\ f = \textbf{do}\ s \leftarrow \mathit{get}; \mathit{return}\ (f\ s)
\end{array}
$$

State Monad: an example

Simon Thompson: The Craft of Functional Programming:

"Given an arbitrary tree, transform it to a tree of integers in which the original elements are replaced by natural numbers, starting from 0. The same element has to be replaced by the same number at every occurrence, and when we meet an as-yet-unvisited element we have to find a 'new' number to match it with."

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Nil} \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\ \textbf{deriving}\ \mathit{Show}$$

$$\mathit{transform} :: \mathit{Eq}\ a \Rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ \mathit{Int}$$

Question: which state information should we maintain?

State Monad: an example (2)

Helper function: determine the index of a value stored in a node.

```
numberNode :: (MonadState [a] m, Eq a) ⇒ a → m Int
numberNode a =
  do mi ← gets (elemIndex a)
     case mi of
       Just i    → return i
       Nothing →
         do modify (λas → as ++ [a])
            gets    (λas → length as − 1)
```

State Monad: an example (3)

$$
\begin{aligned}
&\mathit{numberTree} :: \mathit{Eq}\ a \Rightarrow \mathit{Tree}\ a \rightarrow \mathit{State}\ [a]\ (\mathit{Tree}\ \mathit{Int}) \\
&\mathit{numberTree}\ \mathit{Nil} = \mathit{return}\ \mathit{Nil} \\
&\mathit{numberTree}\ (\mathit{Node}\ a\ t_1\ t_2) = \\
&\quad \mathbf{do}\ i \leftarrow \mathit{numberNode}\ a \\
&\qquad\quad nt_1 \leftarrow \mathit{numberTree}\ t_1 \\
&\qquad\quad nt_2 \leftarrow \mathit{numberTree}\ t_2 \\
&\qquad\quad \mathit{return}\ (\mathit{Node}\ i\ nt_1\ nt_2)
\end{aligned}
$$

$$\mathit{transform} = \mathit{flip}\ \mathit{evalState}\ [\,] \circ \mathit{numberTree}$$

State Monad: an example (3)

$$
\begin{aligned}
&\mathit{numberTree} :: \mathit{Eq}\ a \Rightarrow \mathit{Tree}\ a \rightarrow \mathit{State}\ [a]\ (\mathit{Tree}\ \mathit{Int}) \\
&\mathit{numberTree}\ \mathit{Nil} = \mathit{return}\ \mathit{Nil} \\
&\mathit{numberTree}\ (\mathit{Node}\ a\ t_1\ t_2) = \\
&\quad \mathbf{do}\ i \leftarrow \mathit{numberNode}\ a \\
&\qquad\quad nt_1 \leftarrow \mathit{numberTree}\ t_1 \\
&\qquad\quad nt_2 \leftarrow \mathit{numberTree}\ t_2 \\
&\qquad\quad \mathit{return}\ (\mathit{Node}\ i\ nt_1\ nt_2)
\end{aligned}
$$

$$\mathit{transform} = \mathit{flip}\ \mathit{evalState}\ [\,] \circ \mathit{numberTree}$$

```
Prelude> transform (Node "a" (Node "b" (Node "a" Nil Nil)
                 (Node "c" Nil Nil)) (Node "b" Nil Nil))
Node 0 (Node 1 (Node 0 Nil Nil) (Node 2 Nil Nil)) (Node 1 Nil Nil)
```

Combining monads

Suppose that we want to perform IO actions within the state monad:

Attempt 1:

$$\textbf{data}\ \mathit{StateIO}\ s\ a = S\{\mathit{unS} :: s \to (\mathit{IO}\ (), a, s)\}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ (\mathit{StateIO}\ s)\ \textbf{where}\\
&\quad \mathit{return}\ a = S\ (\lambda s \to (\mathit{return}\ (), a, s))\\
&\quad m \gg\!= f\ \ = S\ (\lambda s \to \textbf{let}\ (\mathit{io}_1, a_1, s_1) = \mathit{unS}\ m\ s\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{io}_2, a_2, s_2) = \mathit{unS}\ (f\ a_1)\ s_1\\
&\qquad\qquad\qquad\qquad\qquad \textbf{in}\ (\mathit{io}_1 \gg \mathit{io}_2, a_2, s_2))
\end{aligned}$$

Combining monads

Suppose that we want to perform IO actions within the state monad:

Attempt 1:

$$\textbf{data}\ \mathit{StateIO}\ s\ a = S\{\mathit{unS} :: s \to (\mathit{IO}\ (), a, s)\}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ (\mathit{StateIO}\ s)\ \textbf{where}\\
&\quad \mathit{return}\ a = S\ (\lambda s \to (\mathit{return}\ (), a, s))\\
&\quad m \gg\!= f\ \ = S\ (\lambda s \to \textbf{let}\ (\mathit{io}_1, a_1, s_1) = \mathit{unS}\ m\ s\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{io}_2, a_2, s_2) = \mathit{unS}\ (f\ a_1)\ s_1\\
&\qquad\qquad\qquad\qquad\qquad \textbf{in}\ (\mathit{io}_1 \gg \mathit{io}_2, a_2, s_2))
\end{aligned}$$

$$\begin{aligned}
&\mathit{doInIO} :: \mathit{IO}\ a \to \mathit{StateIO}\ s\ a\\
&\mathit{doInIO}\ m = S\ (\lambda s \to \boxed{???})
\end{aligned}$$

Combining monads (2)

Attempt 2:

$$\textbf{data}\ \mathit{StateIO}\ s\ a = S\{\mathit{unS} :: s \to \mathit{IO}\ (a, s)\}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ (\mathit{StateIO}\ s)\ \textbf{where}\\
&\quad \mathit{return}\ a = S\ (\lambda s \to \mathit{return}\ (a, s))\\
&\quad m \gg\!= f\ \ = S\ (\lambda s \to\ \textbf{do}\ \ (a_1, s_1) \leftarrow \mathit{unS}\ m\ s\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unS}\ (f\ a_1)\ s_1)
\end{aligned}$$

Combining monads (2)

Attempt 2:

$$\textbf{data}\ \mathit{StateIO}\ s\ a = S\{\mathit{unS} :: s \to \mathit{IO}\ (a, s)\}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ (\mathit{StateIO}\ s)\ \textbf{where}\\
&\quad \mathit{return}\ a = S\ (\lambda s \to \mathit{return}\ (a, s))\\
&\quad m \gg\!= f\ \ = S\ (\lambda s \to\ \textbf{do}\ \ (a_1, s_1) \leftarrow \mathit{unS}\ m\ s\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unS}\ (f\ a_1)\ s_1)
\end{aligned}$$

$$\begin{aligned}
&\mathit{doInIO} :: \mathit{IO}\ a \to \mathit{StateIO}\ s\ a\\
&\mathit{doInIO}\ m = S\ (\lambda s \to\ \textbf{do}\ \ a \leftarrow m; \mathit{return}\ (a, s))
\end{aligned}$$

Combining monads (2)

Attempt 2:

$$\textbf{data}\ \mathit{StateIO}\ s\ a = S\{\mathit{unS} :: s \to \mathit{IO}\ (a, s)\}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ (\mathit{StateIO}\ s)\ \textbf{where}\\
&\quad \mathit{return}\ a = S\ (\lambda s \to \mathit{return}\ (a, s))\\
&\quad m \gg\!= f\ \ = S\ (\lambda s \to\ \textbf{do}\ \ (a_1, s_1) \leftarrow \mathit{unS}\ m\ s\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unS}\ (f\ a_1)\ s_1)
\end{aligned}$$

$$\begin{aligned}
&\mathit{doInIO} :: \mathit{IO}\ a \to \mathit{StateIO}\ s\ a\\
&\mathit{doInIO}\ m = S\ (\lambda s \to\ \textbf{do}\ \ a \leftarrow m; \mathit{return}\ (a, s))
\end{aligned}$$

$$\begin{aligned}
&\mathit{runStateIO} :: \mathit{StateIO}\ s\ a \to s \to \mathit{IO}\ (a, s)\\
&\mathit{runStateIO}\ m\ s = (\mathit{unS}\ m)\ s
\end{aligned}$$

State Transformer: a generalization

$$\textbf{data}\ \mathit{StateT}\ s\ m\ a = S\{\mathit{unS} :: s \to m\ (a, s)\}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathit{StateT}\ s\ m)\ \textbf{where}\\
&\quad \mathit{return}\ a = S\ (\lambda s \to \mathit{return}\ (a, s))\\
&\quad m \gg\!= f\ \ = S\ (\lambda s \to\ \textbf{do}\ \ (a_1, s_1) \leftarrow \mathit{unS}\ m\ s\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unS}\ (f\ a_1)\ s_1)
\end{aligned}$$

State Transformer: a generalization

$$\textbf{data}\ StateT\ s\ m\ a = S\{unS :: s \to m\ (a, s)\}$$

$$\begin{aligned}
&\textbf{instance}\ Monad\ m \Rightarrow Monad\ (StateT\ s\ m)\ \textbf{where}\\
&\quad return\ a = S\ (\lambda s \to return\ (a, s))\\
&\quad m \gg\!= f\ \ = S\ (\lambda s \to\ \textbf{do}\ \ (a_1, s_1) \leftarrow unS\ m\ s\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad unS\ (f\ a_1)\ s_1)
\end{aligned}$$

$$\begin{aligned}
&doInM :: Monad\ m \Rightarrow m\ a \to StateT\ s\ m\ a\\
&doInM\ m = S\ (\lambda s \to\ \textbf{do}\ \ a \leftarrow m; return\ (a, s))
\end{aligned}$$

$$\begin{aligned}
&runStateT :: StateT\ s\ m\ a \to s \to m\ (a, s)\\
&runStateT\ c\ s = (unS\ c)\ s
\end{aligned}$$

Type class for Monad Transformers

A monad transformer is a monad that carries a second (possibly different) monad. Computations in the “inner monad” can be performed using a *lift* function.

Idea: make a type class for monad transformers.

Type class for Monad Transformers

A monad transformer is a monad that carries a second (possibly different) monad. Computations in the "inner monad" can be performed using a *lift* function.

Idea: make a type class for monad transformers.

class *MonadTrans t* **where**
  $lift :: Monad\ m \Rightarrow m\ a \rightarrow t\ m\ a$

More Monad Transformers

Most monads have a monad transformer:

Monad	Transformer	Original Type	Combined Typed
State	*StateT*	$s \to (a,s)$	$s \to m\ (a,s)$
Error	*ErrorT*	*Either e a*	*m (Either e a)*
Reader	*ReaderT*	$r \to a$	$r \to m\ a$
Writer	*WriterT*	*(a, w)*	*m (a, w)*

Question:

Are *StateT s (Error e)* and *ErrorT e (State s)* the same?

More Monad Transformers

Most monads have a monad transformer:

Monad	Transformer	Original Type	Combined Typed
State	*StateT*	$s \to (a,s)$	$s \to m\ (a,s)$
Error	*ErrorT*	*Either e a*	*m (Either e a)*
Reader	*ReaderT*	$r \to a$	$r \to m\ a$
Writer	*WriterT*	*(a, w)*	*m (a, w)*

Question:

Are *StateT s (Error e)* and *ErrorT e (State s)* the same?

- $s \to$ *Either e (a, s)*
- $s \to$ *(Either e a, s)*

Order does matter!

Monad transformers: an example

$$\textbf{type}\ M\ s\ a = StateT\ s\ (Writer\ String)\ a$$

$$numberNode :: (MonadState\ [a]\ m, Eq\ a) \Rightarrow a \rightarrow m\ Int$$

Still works for M!

Monad transformers: an example

```
type M s a = StateT s (Writer String) a

numberNode :: (MonadState [a] m, Eq a) ⇒ a → m Int
```

```
tell :: MonadWriter w m ⇒ w → m ()
```

```
numberTree :: (Show a, Eq a) ⇒ Tree a → M [a] (Tree Int)
numberTree Nil = return Nil
numberTree (Node a t1 t2) =
  do i  ← numberNode a
     tell (show a ++ "=" ++ show i ++ ";")
     nt1 ← numberTree t1
     nt2 ← numberTree t2
     return (Node i nt1 nt2)
```

Monad transformers: an example (2)

$$\textbf{type}\ M\ s\ a = StateT\ s\ (Writer\ String)\ a$$

$$runWriter \quad :: Writer\ w\ a \rightarrow (a, w)$$
$$evalStateT :: Monad\ m \Rightarrow StateT\ s\ m\ a \rightarrow s \rightarrow m\ a$$

How to combine these functions?

Monad transformers: an example (2)

$$\textbf{type}\ M\ s\ a = StateT\ s\ (Writer\ String)\ a$$

$$runWriter \quad :: Writer\ w\ a \rightarrow (a, w)$$
$$evalStateT :: Monad\ m \Rightarrow StateT\ s\ m\ a \rightarrow s \rightarrow m\ a$$

How to combine these functions?

$$runM :: M\ s\ a \rightarrow s \rightarrow (a, String)$$
$$runM\ m\ s = runWriter\ (evalStateT\ m\ s)$$

$$transform :: (Show\ a, Eq\ a) \Rightarrow Tree\ a \rightarrow (Tree\ Int, String)$$
$$transform\ tree = runM\ (numberTree\ tree)\ [\,]$$

Monad transformers: an example (2)

$$\textbf{type}\ M\ s\ a = StateT\ s\ (Writer\ String)\ a$$

$$runWriter :: Writer\ w\ a \rightarrow (a, w)$$
$$evalStateT :: Monad\ m \Rightarrow StateT\ s\ m\ a \rightarrow s \rightarrow m\ a$$

How to combine these functions?

$$runM :: M\ s\ a \rightarrow s \rightarrow (a, String)$$
$$runM\ m\ s = runWriter\ (evalStateT\ m\ s)$$

$$transform :: (Show\ a, Eq\ a) \Rightarrow Tree\ a \rightarrow (Tree\ Int, String)$$
$$transform\ tree = runM\ (numberTree\ tree)\ [\,]$$

Prelude> $putStr\ \$\ snd\ (transform\ tree)$
$\texttt{"a"} = 0; \texttt{"b"} = 1; \texttt{"a"} = 0; \texttt{"c"} = 2; \texttt{"b"} = 1;$

Monad transformers: an example (2)

```
type M s a = StateT s (Writer String) a

runWriter   :: Writer w a → (a, w)
evalStateT  :: Monad m ⇒ StateT s m a → s → m a
```

How to combine these functions?

```
runM :: M s a → s → (a, String)
runM m s = runWriter (evalStateT m s)

transform :: (Show a, Eq a) → Tree a → (Tree Int, String)
transform t
```

```
tree = Node "a" (Node "b" (Node "a" Nil Nil)
                (Node "c" Nil Nil)) (Node "b" Nil Nil)
```

```
Prelude> putStr $ snd (transform tree)
"a" = 0; "b" = 1; "a" = 0; "c" = 2; "b" = 1;
```

Monad transformers: an example (3)

```
Prelude> runM (fail "hello") []
(***Exception : hello
```

Monad transformers: an example (3)

```
Prelude> runM (fail "hello") []
(***Exception : hello
```

Idea: extend monad *M* with the error monad

Monad transformers: an example (3)

```
Prelude> runM (fail "hello") []
(***Exception : hello
```

```
type M s a = StateT s (ErrorT String (Writer String)) a

runWriter   :: Writer w a → (a, w)
runErrorT   :: ErrorT e m a → m (Either e a)
evalStateT  :: Monad m ⇒ StateT s m a → s → m a

runM :: M s a → s → (Either String a, String)
runM m s = runWriter (runErrorT (evalStateT m s))
```

```
Prelude> runM (fail "hello") []
(Left "hello", "")
```

Arrows: an overview

Part 2:
Programming with Arrows

Point-free programming

$$count :: String \to String \to Int$$
$$count\ w = length \circ filter\ (\equiv w) \circ words$$

Advantages:

- Concise and readable
- Well suited to equational reasoning (Bird and Meertens)
- Assemble program from components

Point-free programming

$$count :: String \to String \to Int$$
$$count\ w = length \circ filter\ (\equiv w) \circ words$$

Advantages:

- Concise and readable
- Well suited to equational reasoning (Bird and Meertens)
- Assemble program from components

$$count\ w = print \circ length \circ filter\ (\equiv w) \circ words \circ readFile$$

Point-free programming

$$count :: String \to String \to Int$$
$$count\ w = length \circ filter\ (\equiv w) \circ words$$

Advantages:

- Concise and readable
- Well suited to equational reasoning (Bird and Meertens)
- Assemble program from components

$$count\ w = print \circ length \circ filter\ (\equiv w) \circ words \circ readFile$$

Rejected by the type checker!

Point-free programming with Monads

Solution 1:

$$liftM :: Monad\ m \Rightarrow (a \to b) \to m\ a \to m\ b$$

$$\begin{aligned} countM &:: String \to String \to IO\ () \\ countM\ w &= (\gg\!= print) \\ &\quad \circ\ liftM\ (length \circ filter\ (\equiv w) \circ words) \\ &\quad \circ\ readFile \end{aligned}$$

- $(\gg\!=)$ and $(\circ)$ are used to combine monads/functions
- *liftM* promotes a (pure) function to a monad
- Definition is no longer intuitive

Point-free programming in the Kleisli category

Solution 2:

type $Kleisli\ m\ a\ b = a \rightarrow m\ b$

Point-free programming in the Kleisli category

Solution 2:

$$\textbf{type}\ \mathit{Kleisli}\ m\ a\ b = a \to m\ b$$

An operator for “reverse composition”:

$$\begin{aligned}
&(\ggg) :: \mathit{Monad}\ m \Rightarrow \mathit{Kleisli}\ m\ a\ b \to \mathit{Kleisli}\ m\ b\ c \to \mathit{Kleisli}\ m\ a\ c \\
&(f \ggg g)\ a = \textbf{do}\ b \leftarrow f\ a \\
&\qquad\qquad\qquad\quad\ \ g\ b
\end{aligned}$$

Point-free programming in the Kleisli category

Solution 2:

$$\textbf{type}\ \mathit{Kleisli}\ m\ a\ b = a \to m\ b$$

An operator for "reverse composition":

$$\begin{aligned}
&(\ggg) :: \mathit{Monad}\ m \Rightarrow \mathit{Kleisli}\ m\ a\ b \to \mathit{Kleisli}\ m\ b\ c \to \mathit{Kleisli}\ m\ a\ c \\
&(f \ggg g)\ a = \textbf{do}\ b \leftarrow f\ a \\
&\qquad\qquad\qquad\quad g\ b
\end{aligned}$$

A function to convert pure functions:

$$\begin{aligned}
&\mathit{arr} :: \mathit{Monad}\ m \Rightarrow (a \to b) \to \mathit{Kleisli}\ m\ a\ b \\
&\mathit{arr}\ f = \mathit{return} \circ f
\end{aligned}$$

Point-free programming in the Kleisli category (2)

$$
\begin{aligned}
&\mathit{readFile} :: && \mathit{Kleisli\ IO\ String\ String} \\
&\mathit{print} :: \mathit{Show\ a} \Rightarrow && \mathit{Kleisli\ IO\ a}\ \ () \\
&\mathit{arr} :: \mathit{Monad\ m} \Rightarrow (a \to b) \to && \mathit{Kleisli\ m\ a\ b} \\
&(\ggg) :: \mathit{Monad\ m} \Rightarrow \mathit{Kleisli\ m\ a\ b} \to && \mathit{Kleisli\ m\ b\ c} \to \mathit{Kleisli\ m\ a\ c}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{countK} :: \mathit{String} \to \mathit{Kleisli\ IO\ String}\ () \\
&\mathit{countK}\ w\ = \ \mathit{readFile} \\
&\qquad \ggg \mathit{arr\ words} \\
&\qquad \ggg \mathit{arr}\ (\mathit{filter}\ (\equiv w)) \\
&\qquad \ggg \mathit{arr\ length} \\
&\qquad \ggg \mathit{print}
\end{aligned}
$$

The Arrow type class

Summary: two ways to write point-free definitions:

- Pure functions and function composition
- Kleisli arrows and kleisli composition

The Arrow type class

Summary: two ways to write point-free definitions:

- Pure functions and function composition
- Kleisli arrows and kleisli composition

Idea: overload the arrow operations:

```
class Arrow arr where
  arr     :: (a → b) → arr a b
  (>>>) :: arr a b → arr b c → arr a c
```

The Arrow type class

Summary: two ways to write point-free definitions:

- Pure functions and function composition
- Kleisli arrows and kleisli composition

Idea: overload the arrow operations:

```
class Arrow arr where
  arr     :: (a → b) → arr a b
  (>>>) :: arr a b → arr b c → arr a c
```

Make the function type $(\rightarrow)$ and *Kleisli* instances of *Arrow*:

```
instance Arrow (→) where
  arr     = id
  (>>>) = flip (∘)
```

Arrows: a shared interface

Why is a shared interface so important?

Hughes' answer:

Arrows: a shared interface

Why is a shared interface so important?

Hughes' answer:

1. Experience shows that the Monad type class is a good one. Hence, using it provides users with a powerful tool.

Arrows: a shared interface

Why is a shared interface so important?

Hughes' answer:

1. Experience shows that the Monad type class is a good one. Hence, using it provides users with a powerful tool.
2. Library designer can exploit tools for implementing the interface systematically (for example, monad transformers)

Arrows: a shared interface

Why is a shared interface so important?

Hughes' answer:

1. Experience shows that the Monad type class is a good one. Hence, using it provides users with a powerful tool.
2. Library designer can exploit tools for implementing the interface systematically (for example, monad transformers)
3. Free functionality that works for many different libraries (for instance, monad combinators)

Arrows: a shared interface

Why is a shared interface so important?

Hughes' answer:

1. Experience shows that the Monad type class is a good one. Hence, using it provides users with a powerful tool.
2. Library designer can exploit tools for implementing the interface systematically (for example, monad transformers)
3. Free functionality that works for many different libraries (for instance, monad combinators)
4. Special language support may be added to support the interface (e.g., do notation)

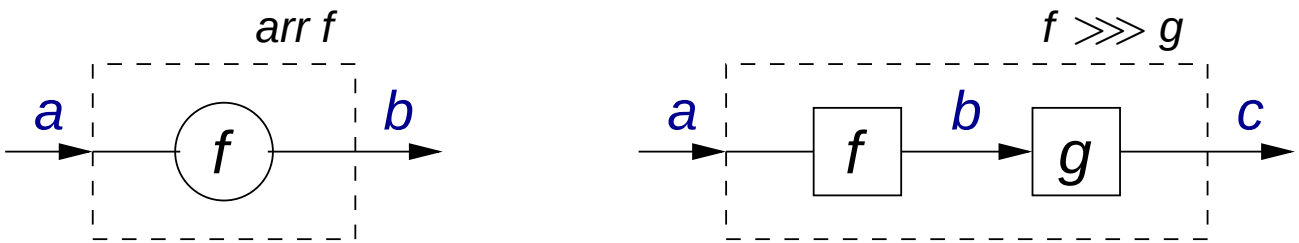
Arrows: a shared interface

Why is a shared interface so important?

Hughes' answer:

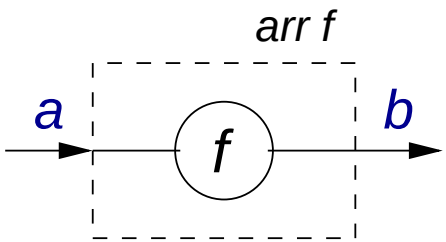
1. Experience shows that the Monad type class is a good one. Hence, using it provides users with a powerful tool.
2. Library designer can exploit tools for implementing the interface systematically (for example, monad transformers)
3. Free functionality that works for many different libraries (for instance, monad combinators)
4. Special language support may be added to support the interface (e.g., do notation)
5. Users familiar with the interface have to learn less to use a new library

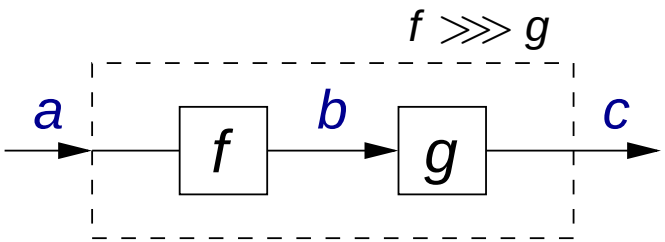
Arrows visualized



$$
\begin{aligned}
&arr && :: Arrow\ arr \Rightarrow (a \to b) \to arr\ a\ b \\
&(\ggg) && :: Arrow\ arr \Rightarrow arr\ a\ b \to arr\ b\ c \to arr\ a\ c
\end{aligned}
$$

Arrows visualized



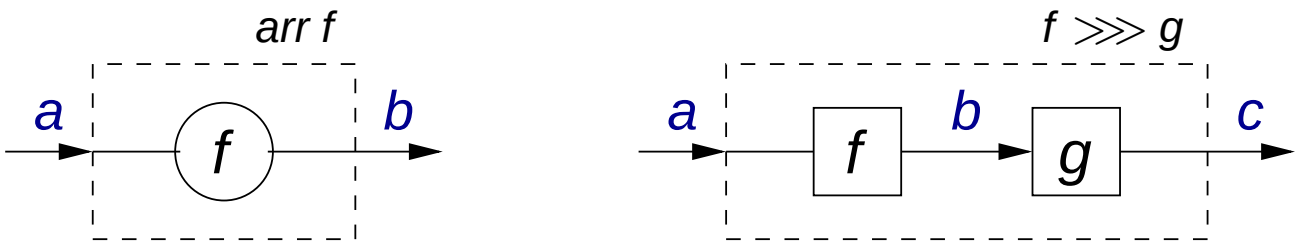


$arr \quad :: A$
$(\ggg) :: A$

Monads versus Arrows

- Both have similar member-functions:
  - pure computation: *arr* and *return*
  - sequencing: $(\ggg)$ and $(\gg\!=)$
- Monads are parameterized over their output, but not their input. Arrows are parameterized over both.
- The second argument of $(\ggg)$ is abstract

Arrows visualized



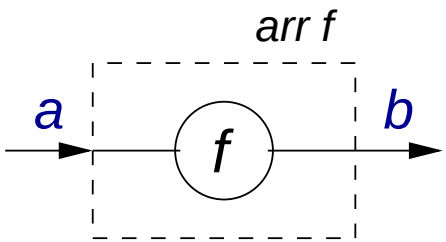
$$
\begin{aligned}
&\mathit{arr} && :: \mathit{Arrow}\ \mathit{arr} \Rightarrow (a \to b) \to \mathit{arr}\ a\ b \\
&(\ggg) && :: \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{arr}\ a\ b \to \mathit{arr}\ b\ c \to \mathit{arr}\ a\ c
\end{aligned}
$$

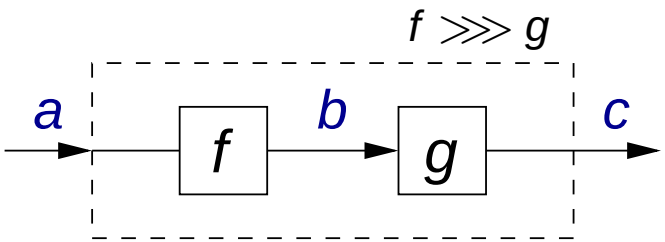
Another instance: the arrow of *stream functions*!

$$\textbf{newtype}\ \mathit{SF}\ a\ b = \mathit{SF}\{\mathit{runSF} :: [a] \to [b]\}$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Arrow}\ \mathit{SF}\ \textbf{where} \\
&\quad \mathit{arr}\ f && = \mathit{SF}\ (\mathit{map}\ f) \\
&\quad \mathit{SF}\ f \ggg \mathit{SF}\ g && = \mathit{SF}\ (f \ggg g)
\end{aligned}
$$

Arrows visualized





$$
\begin{aligned}
&\mathit{arr} && :: \mathit{Arrow}\ \mathit{arr} \Rightarrow (a \to b) \to \mathit{arr}\ a\ b \\
&(\ggg) && :: \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{arr}\ a\ b \to \mathit{arr}\ b\ c \to \mathit{arr}\ a\ c
\end{aligned}
$$

Another instance: the arrow of *stream functions*!

$$\textbf{newtype}\ \mathit{SF}\ a\ b = \mathit{SF}\{\mathit{runSF} :: [a] \to [b]\}$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Arrow}\ \mathit{SF}\ \textbf{where} \\
&\quad \mathit{arr}\ f && = \mathit{SF}\ (\mathit{map}\ f) \\
&\quad \mathit{SF}\ f \ggg \mathit{SF}\ g && = \mathit{SF}\ (f \ggg g)
\end{aligned}
$$

Uses $(\ggg)$ from instance $\mathit{Arrow}\ (\to)$

Stream functions: some examples

$SF >$ $\mathit{runSF}\ (\mathit{arr}\ (+1))\ [1\,..\,5]$
$[2, 3, 4, 5, 6]$

Stream functions: some examples

```
SF> runSF (arr (+1)) [1..5]
[2,3,4,5,6]
```

Delay a stream by one element (specific for *SF*):

```
delay :: a → SF a a
delay x = SF (x:)
```

```
SF> runSF (delay 0) [1..5]
[0,1,2,3,4,5]
```

Stream functions: some examples

```
SF> runSF (arr (+1)) [1..5]
[2,3,4,5,6]
```

Delay a stream by one element (specific for *SF*):

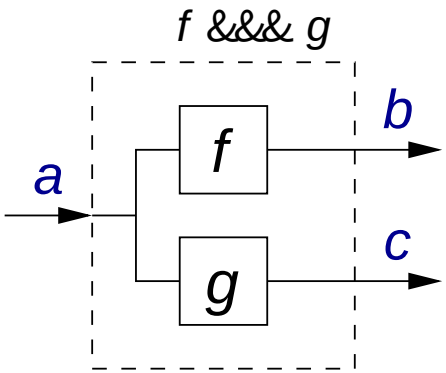
```
delay :: a → SF a a
delay x = SF (x:)
```

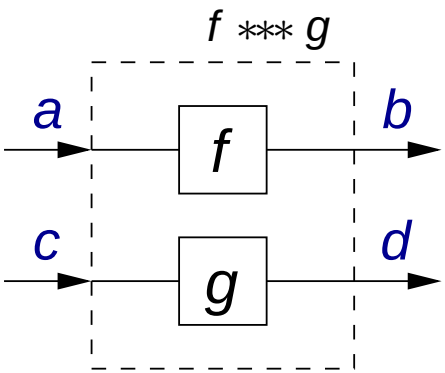
```
SF> runSF (delay 0) [1..5]
[0,1,2,3,4,5]
```

```
SF> runSF (delay 10 >>> arr square) [1..5]
[100,1,4,9,16,25]
```

Arrows and pairs

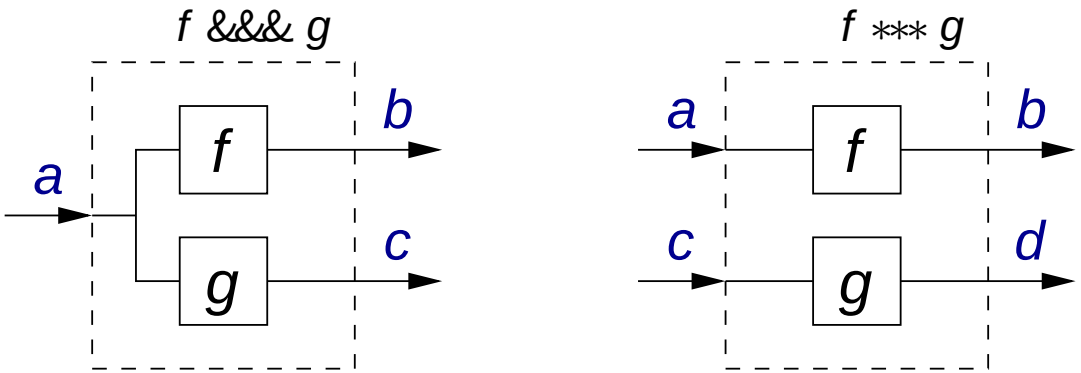
Suppose we have arrows f and g, which output integers from the same input. This cannot be expressed in terms of *arr* and $\ggg$.





Arrows and pairs

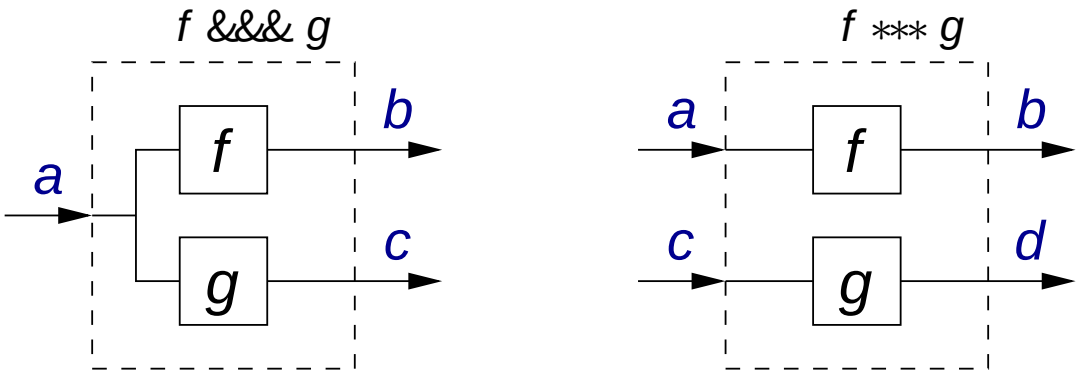
Suppose we have arrows f and g, which output integers from the same input. This cannot be expressed in terms of arr and $\ggg$.



$$(\&\&\&) :: Arrow\ arr \Rightarrow arr\ a\ b \rightarrow arr\ a\ c \rightarrow arr\ a\ (b, c)$$
$$(***) :: Arrow\ arr \Rightarrow arr\ a\ b \rightarrow arr\ c\ d \rightarrow arr\ (a, c)\ (b, d)$$

Arrows and pairs

Suppose we have arrows f and g, which output integers from the same input. This cannot be expressed in terms of arr and $\ggg$.

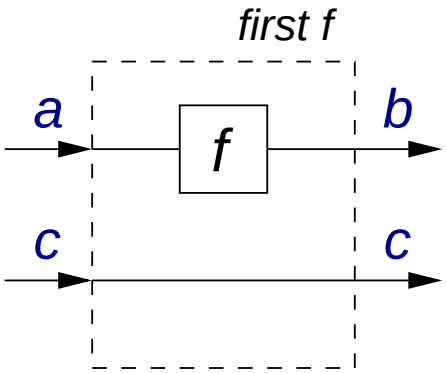


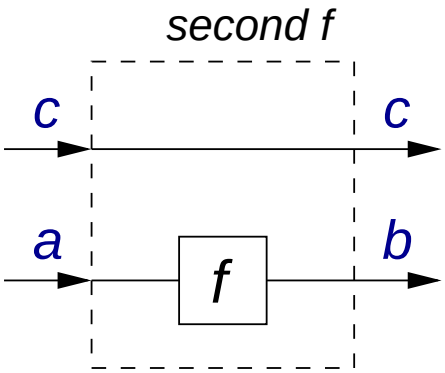
$$(\&\&\&) :: Arrow\ arr \Rightarrow arr\ a\ b \rightarrow arr\ a\ c \rightarrow arr\ a\ (b, c)$$
$$(***) \quad :: Arrow\ arr \Rightarrow arr\ a\ b \rightarrow arr\ c\ d \rightarrow arr\ (a, c)\ (b, d)$$

$$f\ \&\&\&\ g = arr\ (\lambda x \rightarrow (x, x)) \ggg f\ ***\ g$$

Arrows and pairs: more combinators

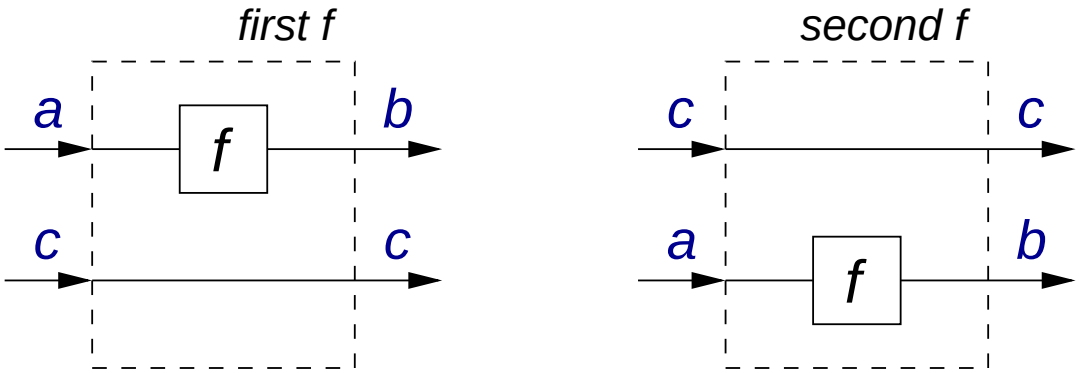
Two more combinators:





Arrows and pairs: more combinators

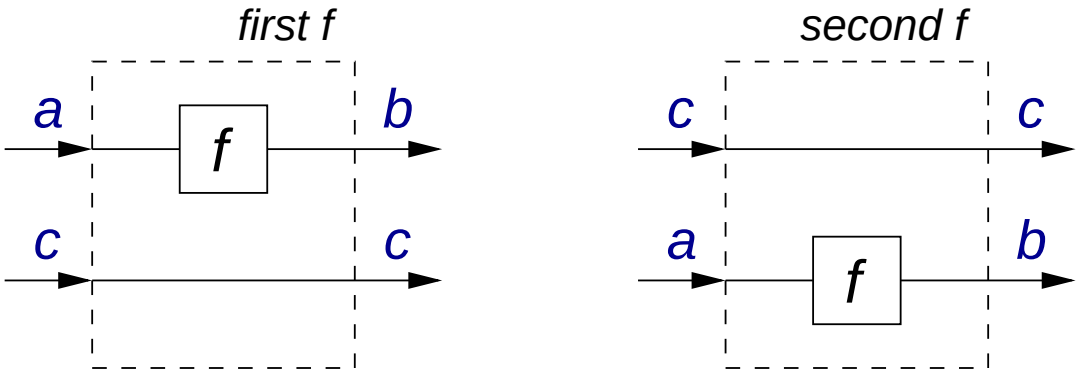
Two more combinators:



$$\begin{aligned} &\mathit{first} &&:: \mathit{Arrow\ arr} \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr}\ (a, c)\ (b, c) \\ &\mathit{second} &&:: \mathit{Arrow\ arr} \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr}\ (c, a)\ (c, b) \end{aligned}$$

Arrows and pairs: more combinators

Two more combinators:



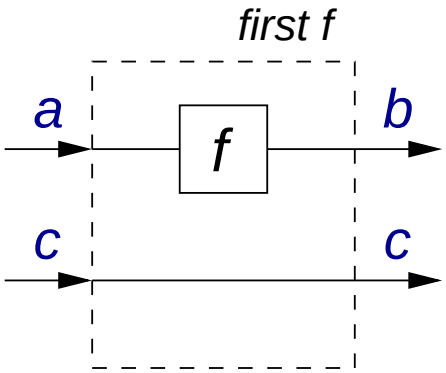
$$
\begin{aligned}
&\mathit{first} &&:: \mathit{Arrow\ arr} \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr}\ (a, c)\ (b, c) \\
&\mathit{second} &&:: \mathit{Arrow\ arr} \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr}\ (c, a)\ (c, b)
\end{aligned}
$$

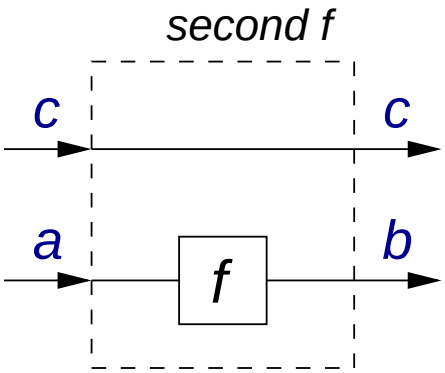
$$
\begin{aligned}
&\mathit{first}\ f &&= f \ast\!\!\!\ast\!\!\!\ast\ \mathit{arr\ id} \\
&\mathit{second}\ f &&= \mathit{arr\ swap} \ggg \mathit{first}\ f \ggg \mathit{arr\ swap} \\
&f \ast\!\!\!\ast\!\!\!\ast\ g &&= \mathit{first}\ f \ggg \mathit{second}\ g
\end{aligned}
$$

$$\mathit{swap}\ (x, y) = (y, x)$$

Arrows and pairs: more combinators

Two more combinators:





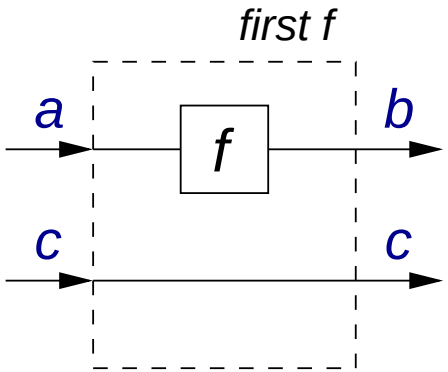
$$\begin{aligned} &\mathit{first} &&:: \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{arr}\ a\ b \to \mathit{arr}\ (a,c)\ (b,c) \\ &\mathit{second} &&:: \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{arr}\ a\ b \to \mathit{arr}\ (c,a)\ (c,b) \end{aligned}$$

$$\begin{aligned} &\mathit{first}\ f &&= f \ast\!\!\!\ast\!\!\!\ast\ a \\ &\mathit{second}\ f &&= \mathit{arr}\ \mathit{sw} \\ &f \ast\!\!\!\ast\!\!\!\ast\ g &&= \mathit{first}\ f \end{aligned}$$

$$\mathit{swap}\ (x,y) = (y,x)$$

- Define *arr*, $\ggg$, and *first*
- &&&, $\ast\!\!\!\ast\!\!\!\ast$, and *second* have a default definition

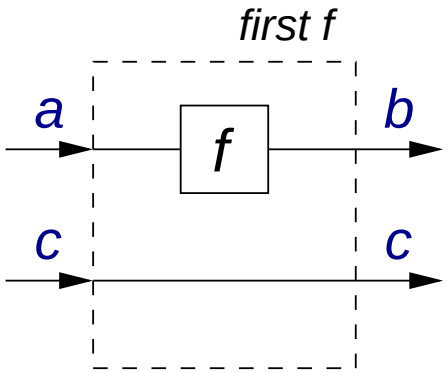
Arrows and pairs: instances



instance *Arrow* $(\rightarrow)$ **where**
$\quad \mathit{first}\ f\ (a, c) = (f\ a, c)$

instance *Arrow SF* **where**
$\quad \mathit{first}\ (\mathit{SF}\ f) = \mathit{SF}\ (\mathit{unzip} \ggg \mathit{first}\ f \ggg \mathit{uncurry}\ \mathit{zip})$

Arrows and pairs: instances

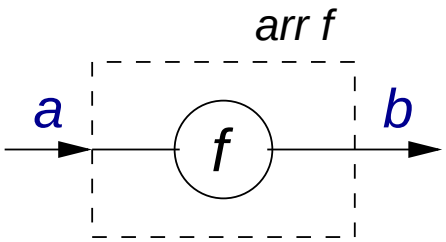


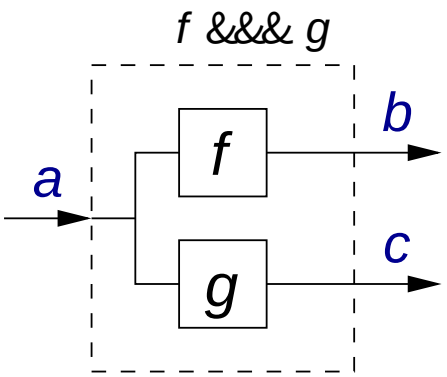
$$
\begin{aligned}
&\mathit{unzip} && :: [(a,c)] \to ([a],[c]) \\
&\mathit{first}\ f && :: ([a],[c]) \to ([b],[c]) \\
&\mathit{uncurry}\ \mathit{zip} && :: ([b],[c]) \to [(b,c)]
\end{aligned}
$$

instance *Arrow* $(\to)$ **where**
  $\mathit{first}\ f\ (a,c) = (f\ a, c)$

instance *Arrow SF* **where**
  $\mathit{first}\ (\mathit{SF}\ f) = \mathit{SF}\ (\mathit{unzip} \ggg \mathit{first}\ f \ggg \mathit{uncurry}\ \mathit{zip})$

Arrows and pairs: examples

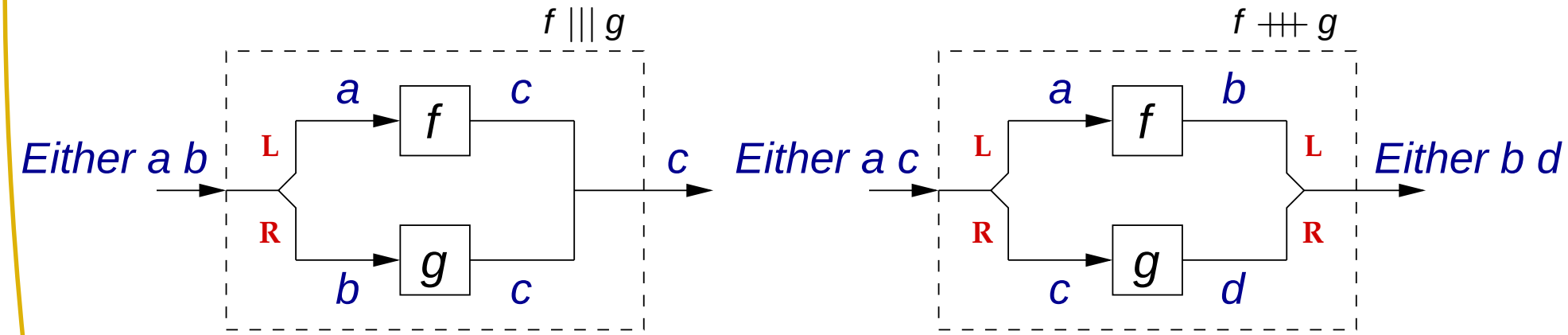




Arrows> *runSF* (*arr square* &&& *arr* $(+1)$) $[1\,..\,5]$
$[(1,2),(4,3),(9,4),(16,5),(25,6)]$

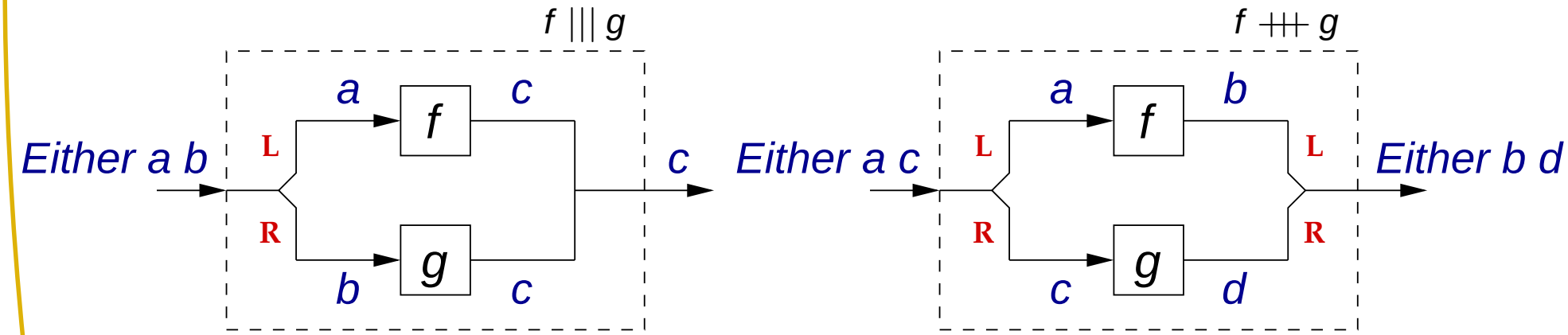
Arrows and conditionals

How can we make a choice between two arrows, based on a previous result. This cannot be expressed in terms of *arr*, $\ggg$, and *first*.



Arrows and conditionals

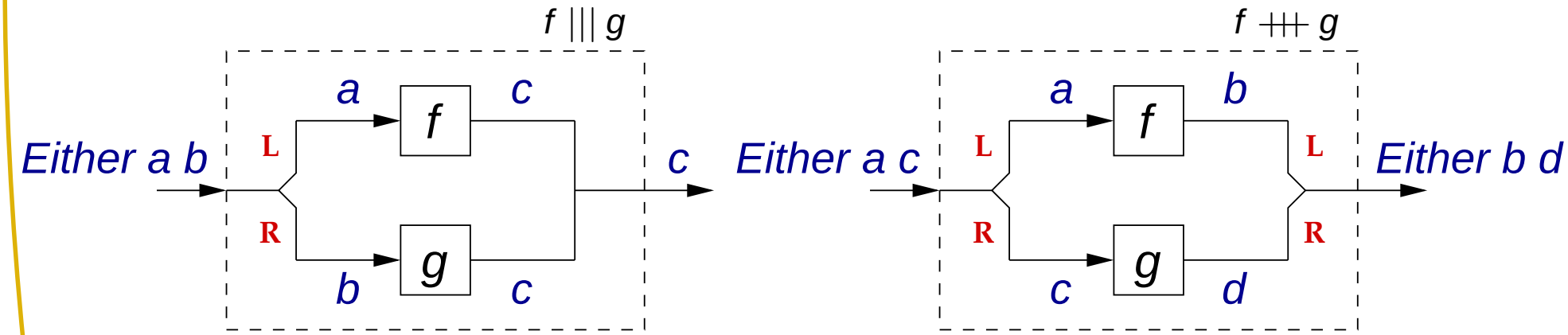
How can we make a choice between two arrows, based on a previous result. This cannot be expressed in terms of *arr*, $\ggg$, and *first*.



$$
\begin{aligned}
(|||) &:: \mathit{ArrowChoice\ arr} \Rightarrow \mathit{arr\ a\ c} \rightarrow \mathit{arr\ b\ c} \rightarrow \mathit{arr}\ (\mathit{Either\ a\ b})\ c \\
(+\!\!+\!\!+) &:: \mathit{ArrowChoice\ arr} \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr\ c\ d} \rightarrow \mathit{arr}\ (\mathit{Either\ a\ c})\ (\mathit{Either\ b\ d})
\end{aligned}
$$

Arrows and conditionals

How can we make a choice between two arrows, based on a previous result. This cannot be expressed in terms of *arr*, $\ggg$, and *first*.

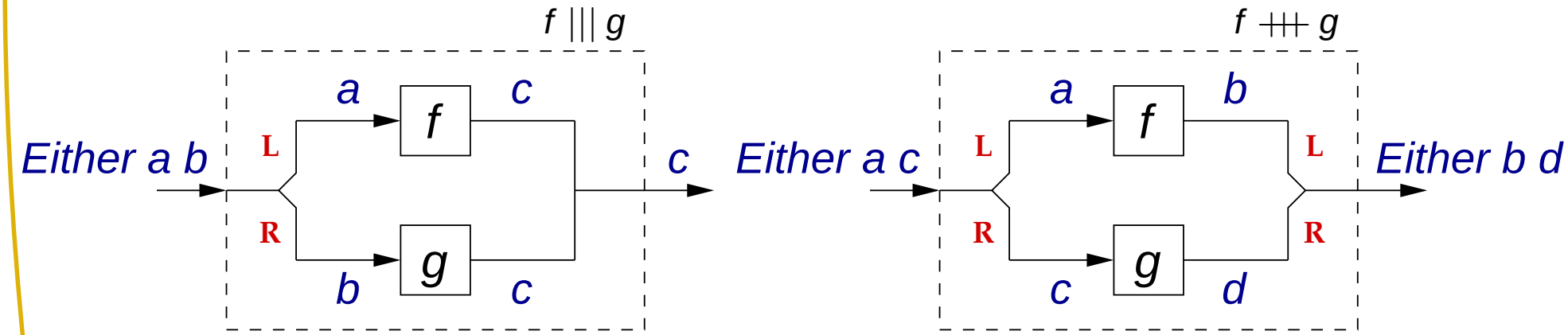


$$(|||) :: ArrowChoice\ arr \Rightarrow arr\ a\ c \rightarrow arr\ b\ c \rightarrow arr\ (Either\ a\ b)\ c$$
$$(+++) :: ArrowChoice\ arr \Rightarrow arr\ a\ b \rightarrow arr\ c\ d \rightarrow arr\ (Either\ a\ c)\ (Either\ b\ d)$$

Place these operators in a subclass *ArrowChoice*: not all arrows support the choice operators

Arrows and conditionals

How can we make a choice between two arrows, based on a previous result. This cannot be expressed in terms of *arr*, $\ggg$, and *first*.

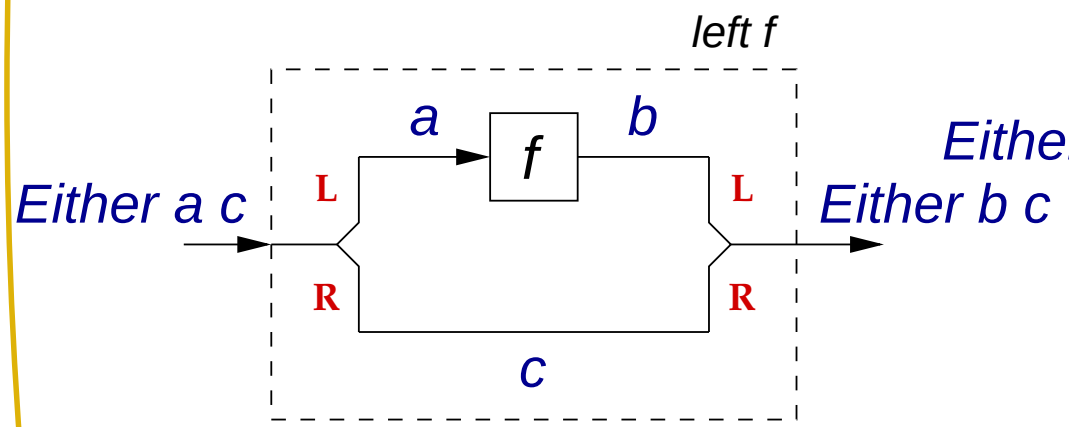


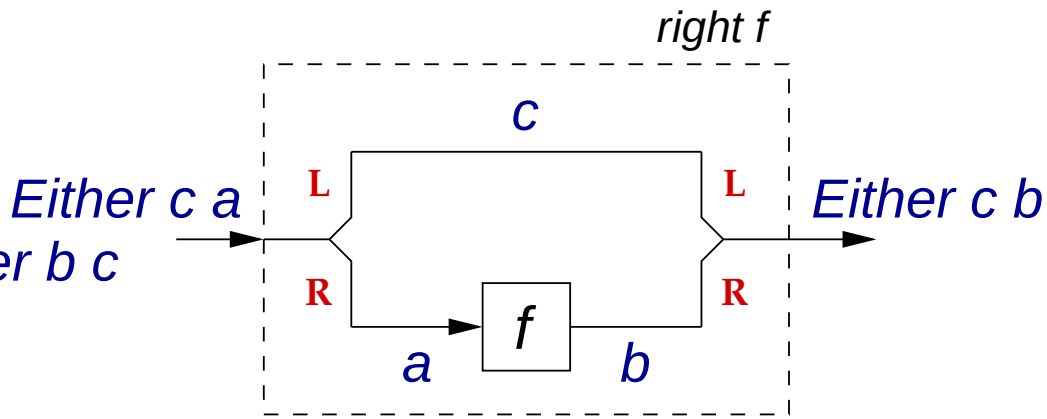
$$
\begin{aligned}
(|||) &:: \mathit{ArrowChoice\ arr} \Rightarrow \mathit{arr\ a\ c} \rightarrow \mathit{arr\ b\ c} \rightarrow \mathit{arr}\ (\mathit{Either\ a\ b})\ c \\
(+\!\!+\!\!+) &:: \mathit{ArrowChoice\ arr} \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr\ c\ d} \rightarrow \mathit{arr}\ (\mathit{Either\ a\ c})\ (\mathit{Either\ b\ d})
\end{aligned}
$$

$$
\begin{aligned}
&f \mid\mid\mid g = f +\!\!+\!\!+ g \ggg \mathit{arr\ join} \\
&\quad \textbf{where}\ \mathit{join} = \mathit{either\ id\ id}
\end{aligned}
$$

Arrows and conditionals: more combinators

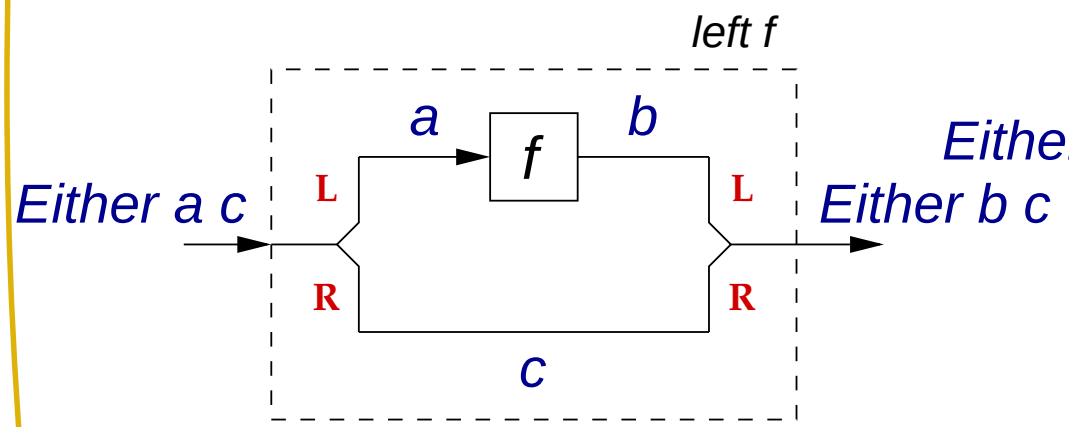
Two more combinators:

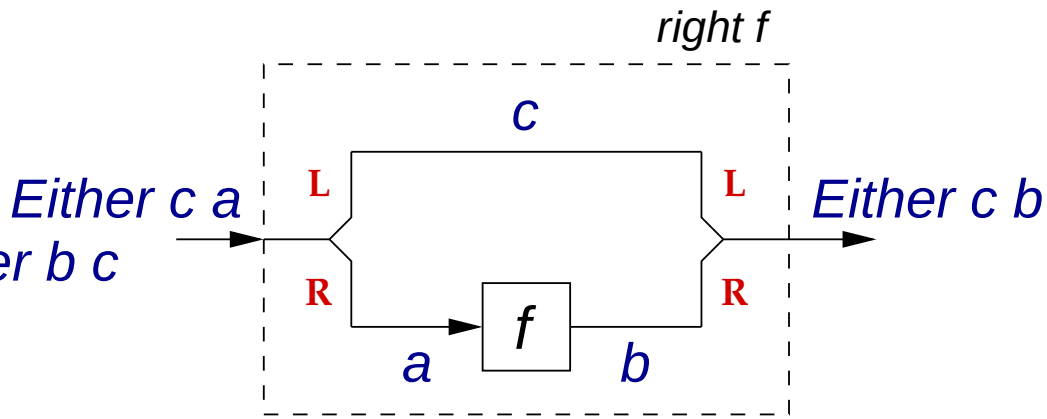




Arrows and conditionals: more combinators

Two more combinators:

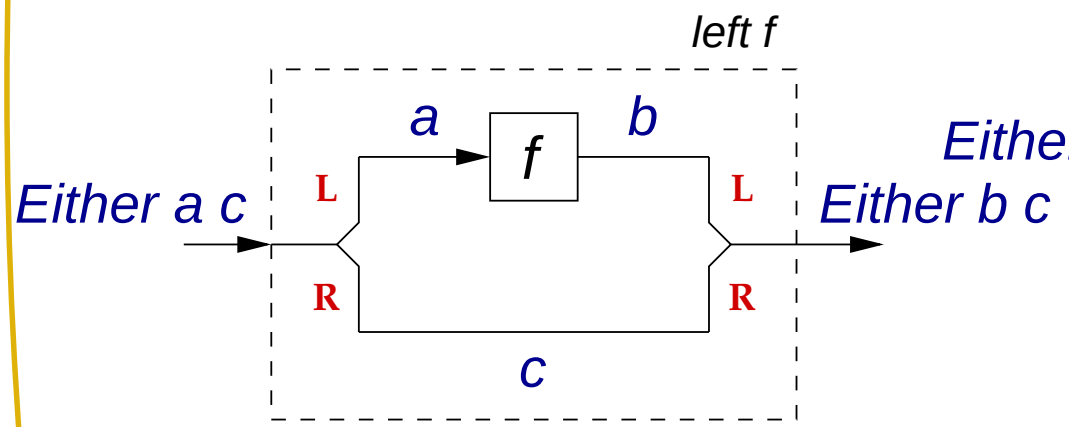


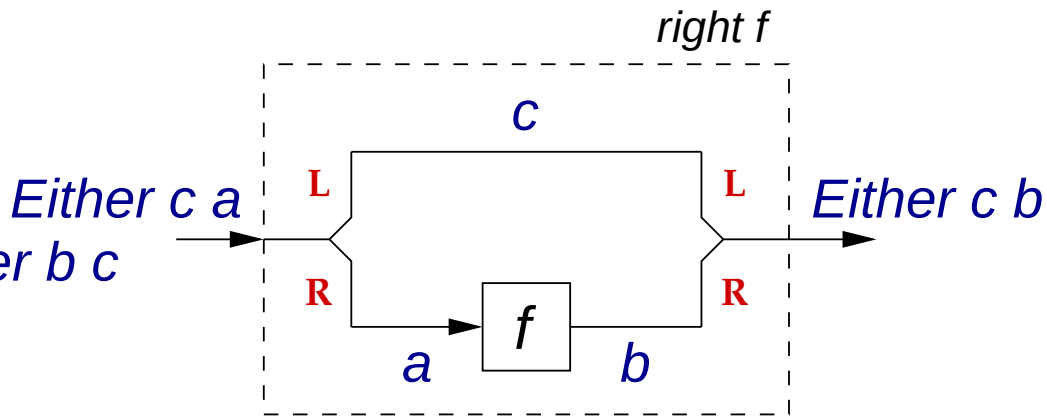


$$\begin{aligned} \mathit{left} &:: \mathit{ArrowChoice\ arr} \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr}\ (\mathit{Either\ a\ c})\ (\mathit{Either\ b\ c}) \\ \mathit{right} &:: \mathit{ArrowChoice\ arr} \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr}\ (\mathit{Either\ c\ a})\ (\mathit{Either\ c\ b}) \end{aligned}$$

Arrows and conditionals: more combinators

Two more combinators:



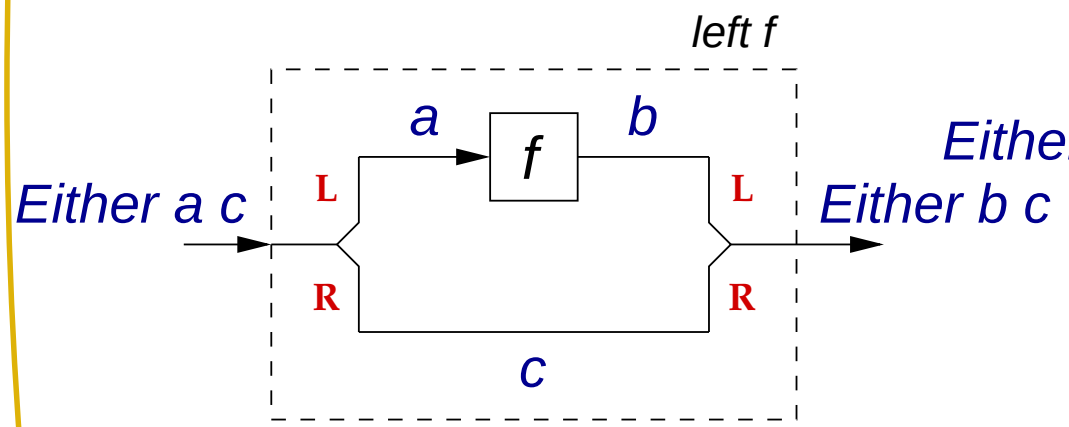


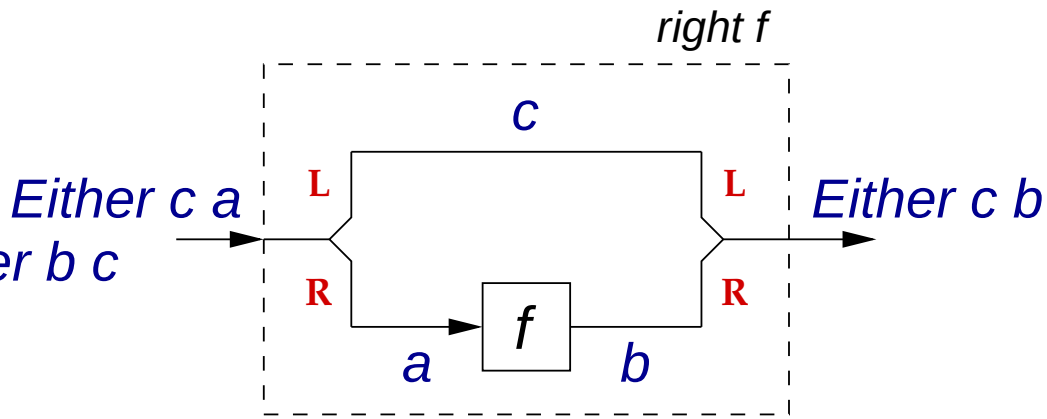
$$\begin{aligned} &\mathit{left} \;\; :: \mathit{ArrowChoice\ arr} \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr}\ (\mathit{Either\ a\ c})\ (\mathit{Either\ b\ c}) \\ &\mathit{right} :: \mathit{ArrowChoice\ arr} \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr}\ (\mathit{Either\ c\ a})\ (\mathit{Either\ c\ b}) \end{aligned}$$

$$\begin{aligned} &\mathit{right\ f} \;\; = \mathit{arr\ mirror} \ggg \mathit{left\ f} \ggg \mathit{arr\ mirror} \\ &f \mathbin{+\!\!+\!\!+} g = \mathit{left\ f} \ggg \mathit{right\ g} \\ \\ &\mathit{mirror} \;\; = \mathit{either\ Right\ Left} \end{aligned}$$

Arrows and conditionals: more combinators

Two more combinators:





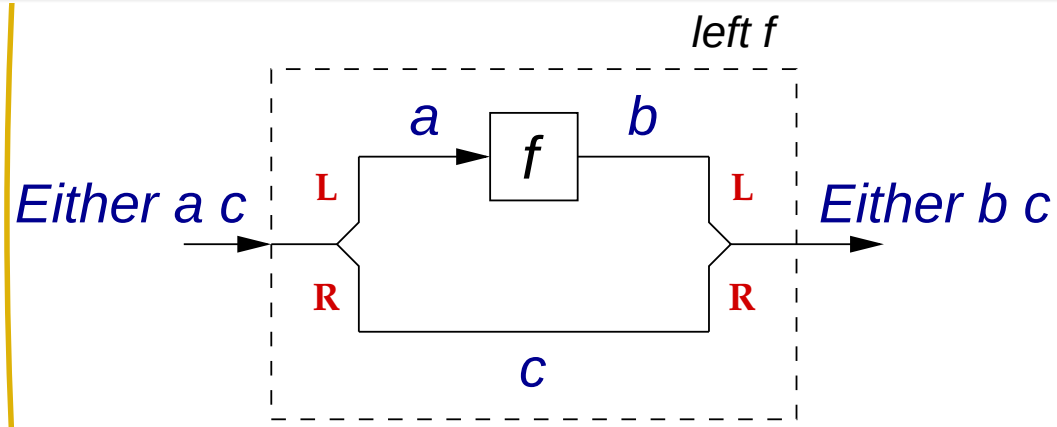
$$\begin{aligned} &\mathit{left} :: \mathit{ArrowChoice\ arr} \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr}\ (\mathit{Either\ a\ c})\ (\mathit{Either\ b\ c}) \\ &\mathit{right} :: \mathit{ArrowChoice\ arr} \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr}\ (\mathit{Either\ c\ a})\ (\mathit{Either\ c\ b}) \end{aligned}$$

$$\begin{aligned} \mathit{right\ f} &= \mathit{arr\ mirro} \\ f +\!\!+\!\!+ g &= \mathit{left\ f} \ggg \end{aligned}$$

- Define *left*
- $|||$, $+\!\!+\!\!+$, and *right* have a default definition

$$\mathit{mirror} = \mathit{either\ Right\ Left}$$

Arrows and conditionals: instances



instance *ArrowChoice* $(\rightarrow)$ **where**

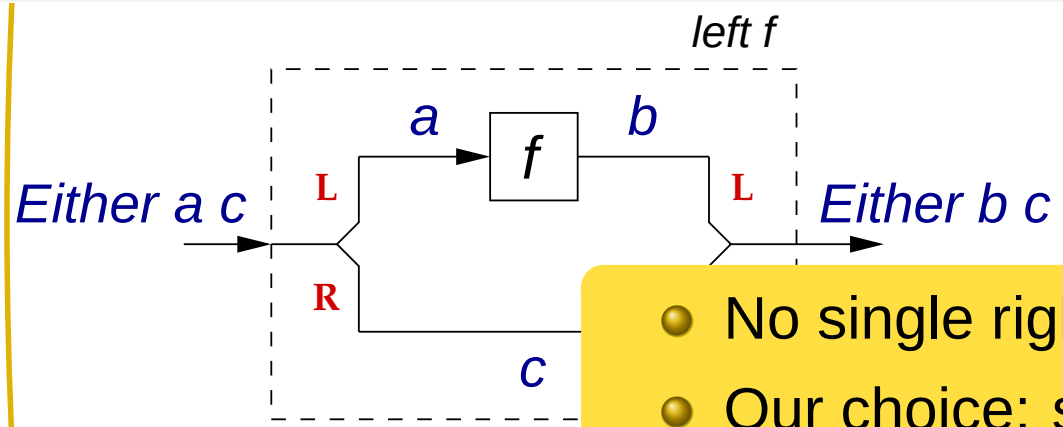
$$\begin{aligned} &\mathit{left}\ f\ (\mathit{Left}\ a) &&= \mathit{Left}\ (f\ a) \\ &\mathit{left}\ f\ (\mathit{Right}\ b) &&= \mathit{Right}\ b \end{aligned}$$

instance *ArrowChoice SF* **where**

$$\mathit{left}\ (\mathit{SF}\ f) = \mathit{SF}\ (\lambda xs \rightarrow \mathit{combine}\ xs\ (f\ [y \mid \mathit{Left}\ y \leftarrow xs]))$$

$$\begin{aligned} &\mathit{combine}\ (\mathit{Left}\ y : xs)\ \ (z : zs) &&= \mathit{Left}\ z\ \ : \mathit{combine}\ xs\ zs \\ &\mathit{combine}\ (\mathit{Right}\ y : xs)\ zs &&= \mathit{Right}\ y : \mathit{combine}\ xs\ zs \\ &\mathit{combine}\ [\,]\ \ zs &&= [\,] \end{aligned}$$

Arrows and conditionals: instances



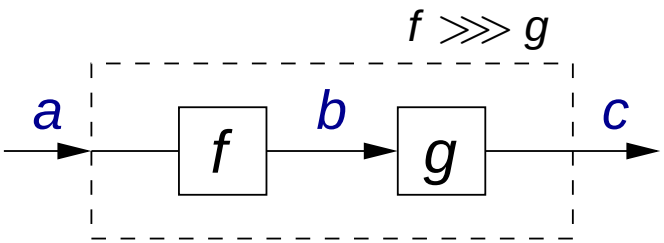
- No single right answer.
- Our choice: synchronous stream functions (one output per input).
- One element of *f*'s output for one element tagged *Left* in input.

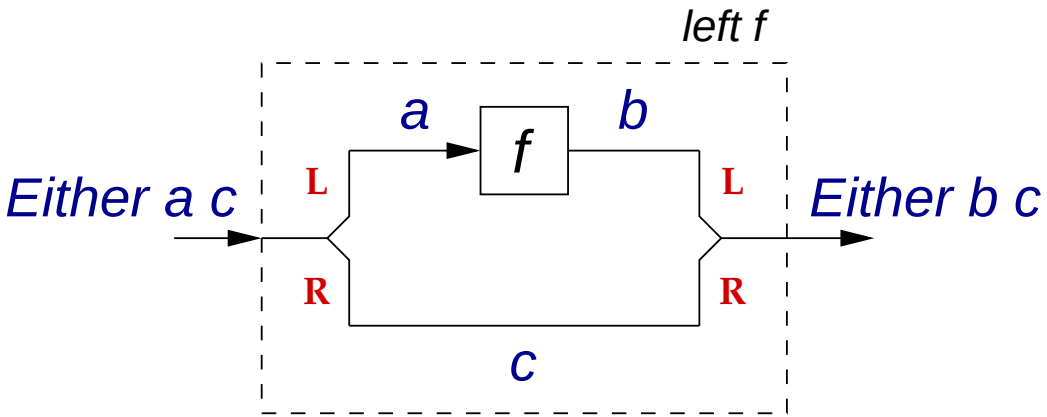
instance *ArrowCh*
  left f (*Left a*) =
  left f (*Right b*) =

instance *ArrowChoice SF* **where**
  $left\ (SF\ f) = SF\ (\lambda xs \rightarrow combine\ xs\ (f\ [y \mid Left\ y \leftarrow xs]))$

$$
\begin{array}{lll}
combine\ (Left\ y : xs) & (z : zs) & = Left\ z \quad : combine\ xs\ zs \\
combine\ (Right\ y : xs) & zs & = Right\ y : combine\ xs\ zs \\
combine\ [\,] & zs & = [\,]
\end{array}
$$

Arrows and conditionals: examples



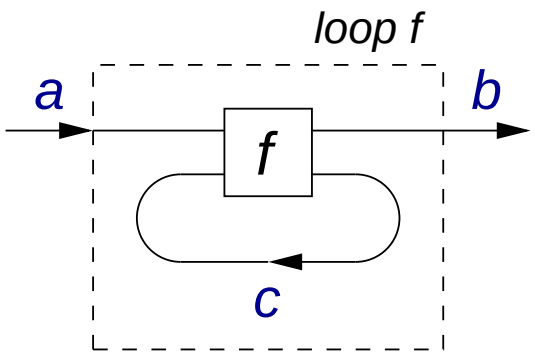


$isEven :: Int \rightarrow Either\ Int\ Int$
$isEven\ i = \textbf{if}\ even\ i\ \textbf{then}\ Left\ i\ \textbf{else}\ Right\ i$

Arrows> $runSF\ (arr\ isEven \ggg left\ (arr\ square))\ [0\,..\,5]$
$[Left\ 0, Right\ 1, Left\ 4, Right\ 3, Left\ 16, Right\ 5]$

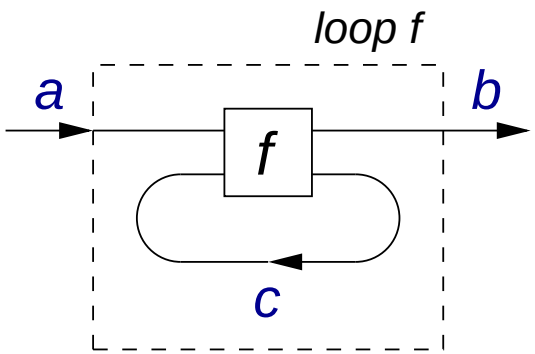
Arrows and feedback

Feed back the output to the input



Arrows and feedback

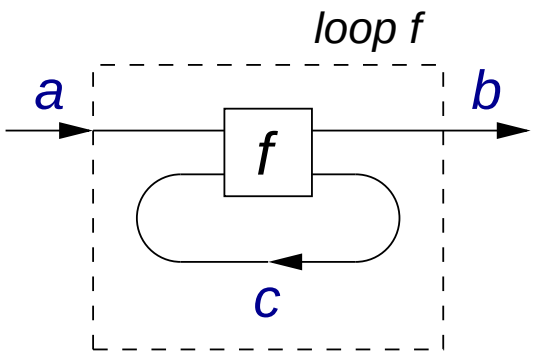
Feed back the output to the input



$$loop :: ArrowLoop\ arr \Rightarrow arr\ (a, c)\ (b, c) \rightarrow arr\ a\ b$$

Arrows and feedback

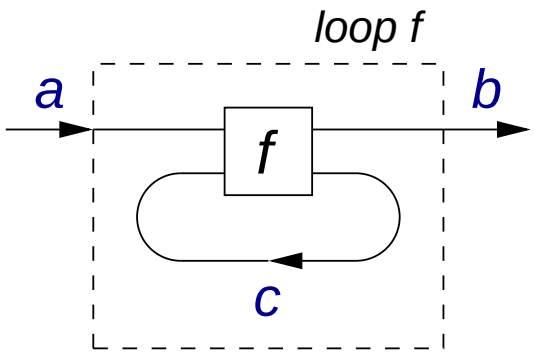
Feed back the output to the input



$$loop :: ArrowLoop\ arr \Rightarrow arr\ (a, c)\ (b, c) \rightarrow arr\ a\ b$$

ArrowLoop is yet another subclass of *Arrow*

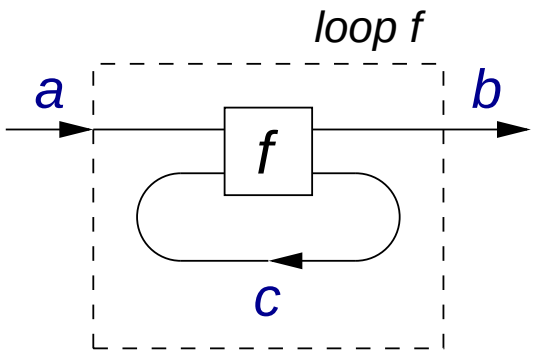
Arrows and feedback: instances




```
instance ArrowLoop (→) where
  loop f a = b
    where (b, c) = f (a, c)

instance ArrowLoop SF where
  loop (SF f) = SF $ λas →
    let (bs, cs) = unzip (f (zip as cs))
    in bs
```

Arrows and feedback: instances



This definition is too strict (*zip* and *unzip* are both strict).

```
instance ArrowLoop (→) where
  loop f a = b
    where (b, c) = f (a, c)

instance ArrowLoop SF where
  loop (SF f) = SF $ λas →
    let (bs, cs) = unzip (f (zip as cs))
    in bs
```

Arrows and feedback: instances (2)

A more lazy definition

```
instance ArrowLoop SF where
  loop (SF f) = SF $ λas →
    let (bs, cs) = unzip (f (zip as (stream cs)))
    in bs

stream ~(x : xs) = x : stream xs
```

Arrows and feedback: instances (2)

A more lazy definition

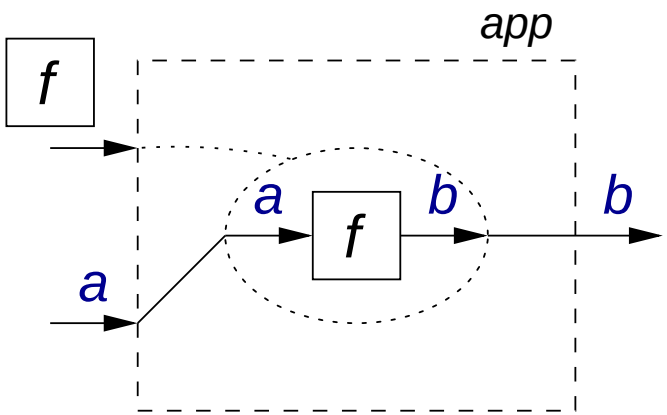
```
instance ArrowLoop SF where
  loop (SF f) = SF $ λas →
    let (bs, cs) = unzip (f (zip as (stream cs)))
    in bs

stream ~(x : xs) = x : stream xs
```

- The function *stream* returns an infinite list *without* evaluating the arguments
- (Exercise: Try to find a number of applications for *loop*)

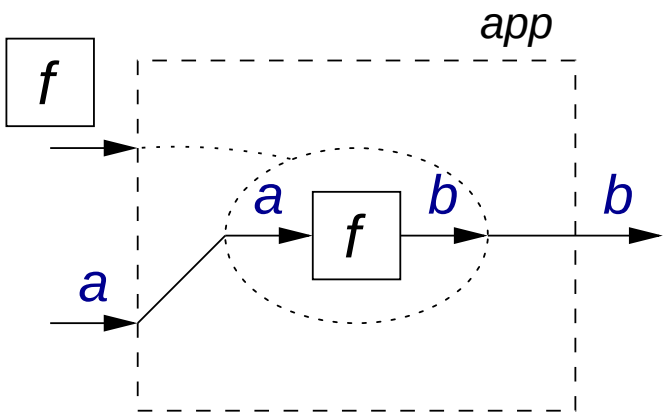
Higher-order arrows

Higher-order programming with arrows:



Higher-order arrows

Higher-order programming with arrows:



$app :: ArrowApply\ arr \Rightarrow arr\ (arr\ a\ b, a)\ b$

Once more, we introduce a new subclass of *Arrow*

Fibonacci numbers

$0, 1, 1, 2, 3, 5, 8, 13, \ldots$

Can we define the sequence of Fibonacci numbers as a *stream function* using arrows?

Fibonacci numbers

$0, 1, 1, 2, 3, 5, 8, 13, \ldots$

Can we define the sequence of Fibonacci numbers as a *stream function* using arrows?

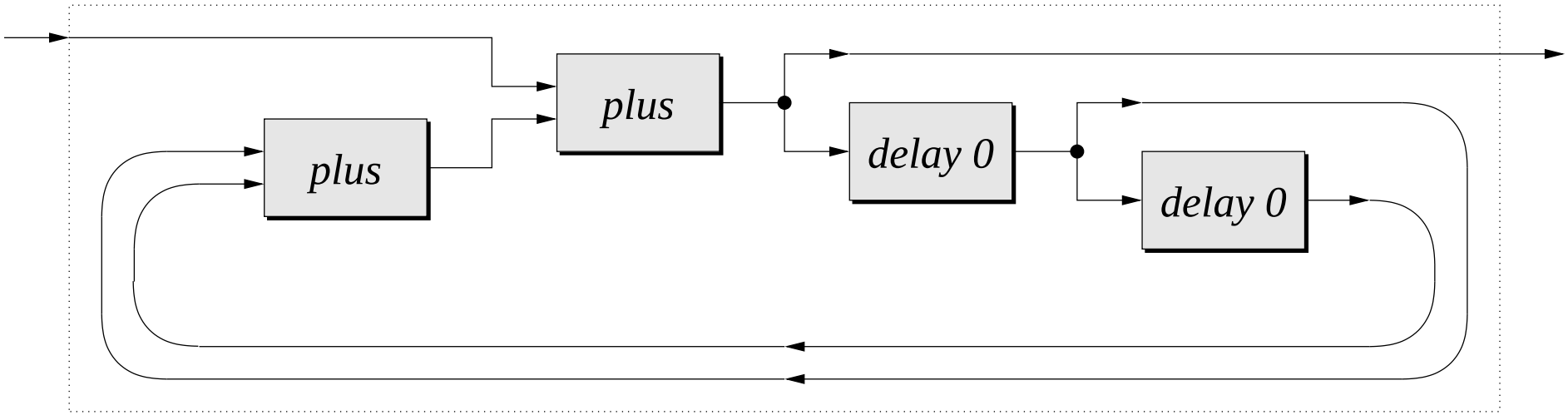
We need:

- *delay* a stream by one element
- *plus* for adding two values of type *Int*
- arrow combinators

$$\begin{aligned}
&\mathit{delay} :: a \to \mathit{SF}\ a\ a \\
&\mathit{delay}\ x = \mathit{SF}\ (x:) \\
\\
&\mathit{plus} :: \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{arr}\ (\mathit{Int}, \mathit{Int})\ \mathit{Int} \\
&\mathit{plus} = \mathit{arr}\ (\mathit{uncurry}\ (+))
\end{aligned}$$

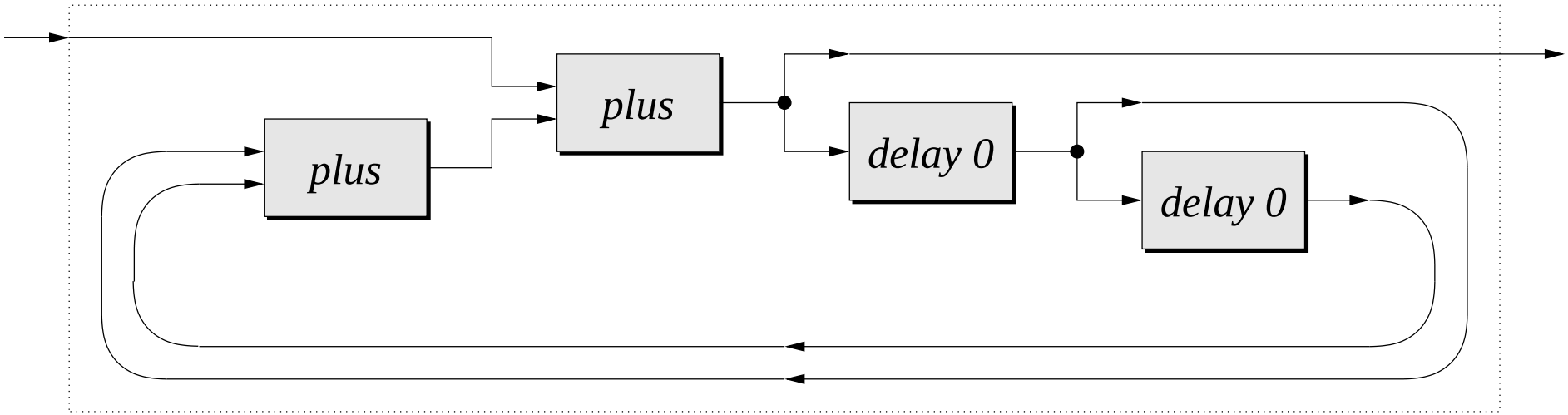
Remaining question: how to wire our *Fibonacci Machine*?

The Fibonacci machine



- How can we define the machine using the arrow combinators?

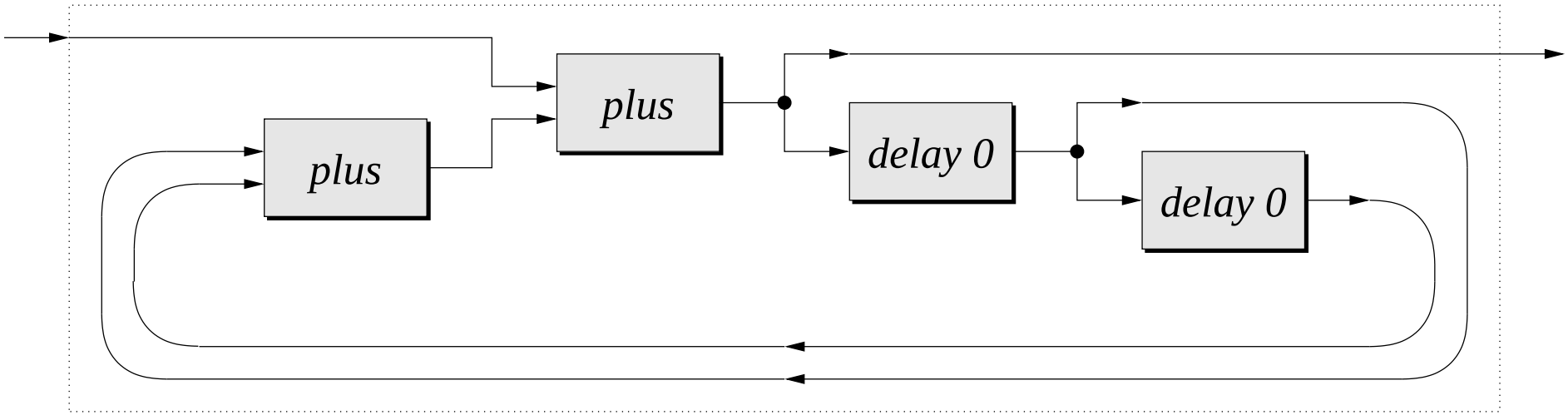
The Fibonacci machine



- How can we define the machine using the arrow combinators?

$$
\begin{aligned}
&\mathit{machine} :: \mathit{SF}\ \mathit{Int}\ \mathit{Int} \\
&\mathit{machine} = \mathit{loop}\ (\mathit{second}\ \mathit{plus} \ggg \mathit{plus} \ggg \\
&\qquad\qquad \mathit{arr}\ \mathit{id}\ \&\&\&\ (\mathit{delay}\ 0 \ggg \mathit{arr}\ \mathit{id}\ \&\&\&\ \mathit{delay}\ 0))
\end{aligned}
$$

The Fibonacci machine

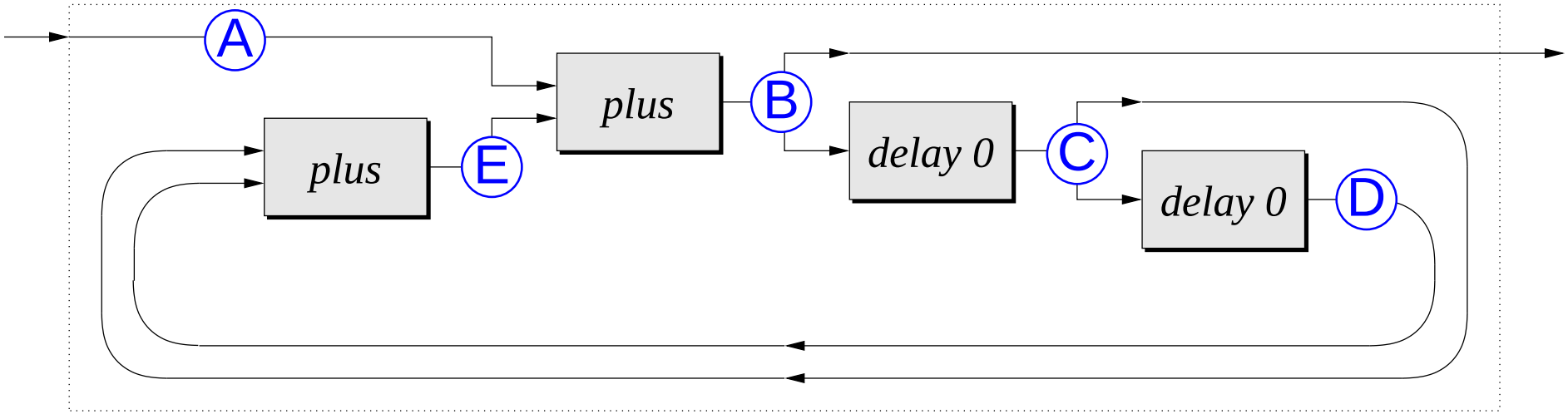


- How can we define the machine using the arrow combinators?

$$
\begin{aligned}
&\mathit{machine} :: \mathit{SF}\ \mathit{Int}\ \mathit{Int} \\
&\mathit{machine} = \mathit{loop}\ (\mathit{second}\ \mathit{plus} \ggg \mathit{plus} \ggg \\
&\qquad\qquad \mathit{arr}\ \mathit{id}\ \&\&\&\ (\mathit{delay}\ 0 \ggg \mathit{arr}\ \mathit{id}\ \&\&\&\ \mathit{delay}\ 0))
\end{aligned}
$$

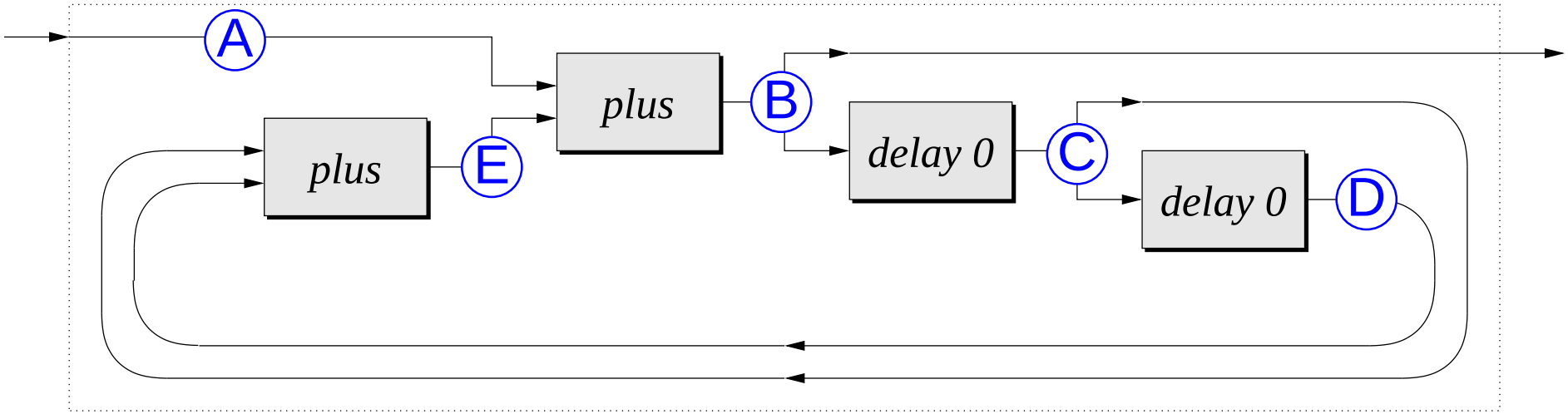
- Which input stream should we use to produce the sequence of Fibonacci numbers?

The Fibonacci machine in action



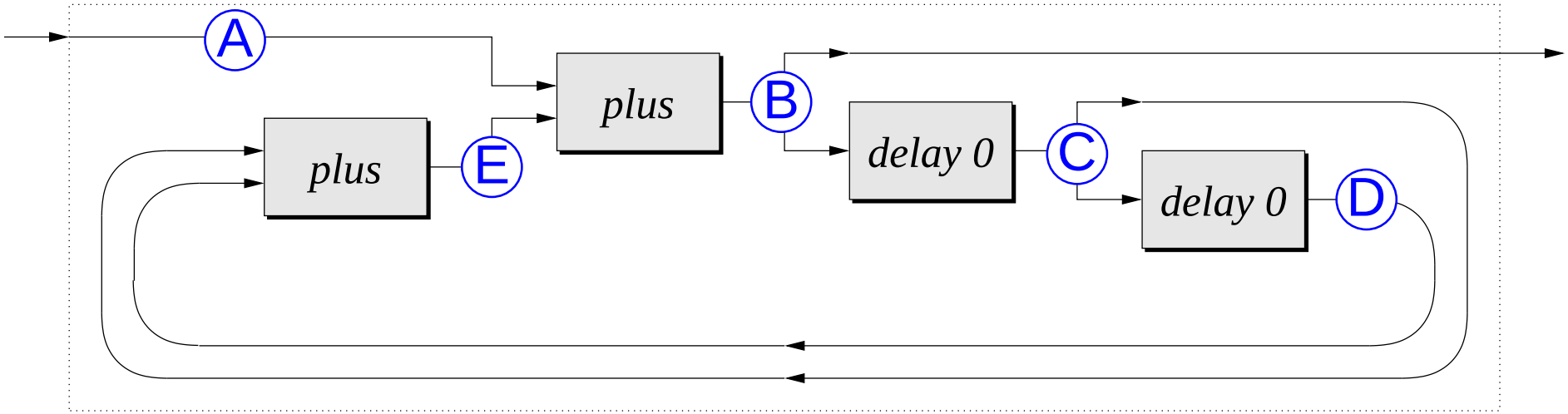
B:	?	?	?	?	?	?	?	?	
C:	0	?	?	?	?	?	?	?	
D:	0	0	?	?	?	?	?	?	
E:	0	?	?	?	?	?	?	?	
A:	0	1	0	0	0	0	0	0	-- input
B:	0	?	?	?	?	?	?	?	-- output

The Fibonacci machine in action



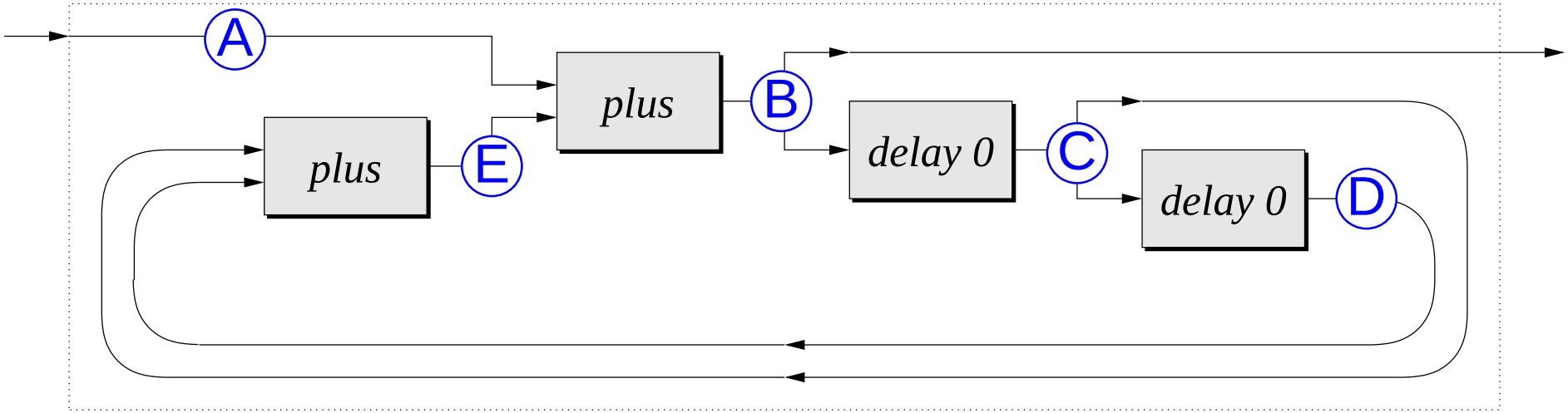
B:	0	?	?	?	?	?	?	?	
C:	0	0	?	?	?	?	?	?	
D:	0	0	0	?	?	?	?	?	
E:	0	0	?	?	?	?	?	?	
A:	0	1	0	0	0	0	0	0	-- input
B:	0	1	?	?	?	?	?	?	-- output

The Fibonacci machine in action



B:	0	1	?	?	?	?	?	?	
C:	0	0	1	?	?	?	?	?	
D:	0	0	0	1	?	?	?	?	
E:	0	0	1	?	?	?	?	?	
A:	0	1	0	0	0	0	0	0	-- input
B:	0	1	1	?	?	?	?	?	-- output

The Fibonacci machine in action



B:	0	1	1	2	3	5	8	13	
C:	0	0	1	1	2	3	5	8	
D:	0	0	0	1	1	2	3	5	
E:	0	0	1	2	3	5	8	13	
A:	0	1	0	0	0	0	0	0	-- input
B:	0	1	1	2	3	5	8	13	-- output

Arrows: an overview

- Inspired by category theory: new instance $\approx$ category!
- Arrow operators only offer compositions: truly interesting are the additional operations that are supported.
- GHC supports special arrow syntax
- Adopted to do Functional Reactive Programming (Yampa)
- Arrow combinators improve memory behavior (garbage collection)

Agenda

1. Read *"wxHaskell: A Portable and Concise GUI Library for Haskell" (Daan Leijen)*.
2. Read *"Chapter 2: H/Direct: a binary language interface for Haskell"* from Daan's PhD thesis.
3. Study Sections 6–8 of *"Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell" (Simon Peyton Jones)*.

Next lecture:

February 19 (Monday):

- wxHaskell
- Foreign Function Interface (FFI)