Stream Arrows and Stream Lenses
Co-iterative Semantics of Dataflow in Haskell

M. Mendler

University of Bamberg

Introduction

Haskell Recursive Streams

```
Str :: * → *
data Str a = !a :< Str a -- bang pattern (!) for strictness
```

The stream constructor `:<` ("grumpy") induces destructors (`s_head`, `s_tail`) and pattern matching (`case e1 of !a :< as` → `e2`).

Haskell Recursive Streams

```
Str :: * → *
data Str a = !a :< Str a -- bang pattern (!) for strictness
```

The stream constructor `:<` ("grumpy") induces destructors (`s_head`, `s_tail`) and pattern matching (`case e1 of !a :< as → e2`).

Using recursion we can generate stream functions:

```
s_nats :: Str Int
s_nats = ns where
  ns = 0 :< s_inc ns -- produces 1 grumpy up front
  s_inc (n :< ns) = (n + 1) :< s_inc ns
      -- consumes and produces 1 grumpy
```

Haskell Recursive Streams

```
Str :: * → *
data Str a = !a :< Str a -- bang pattern (!) for strictness
```

The stream constructor `:<` ("grumpy") induces destructors (`s_head`, `s_tail`) and pattern matching (`case e1 of !a :< as → e2`).

Using recursion we can generate stream functions:

```
s_nats :: Str Int
s_nats = ns where
  ns = 0 :< s_inc ns -- produces 1 grumpy up front
  s_inc (n :< ns) = (n + 1) :< s_inc ns
      -- consumes and produces 1 grumpy
```

With lazy evaluation we can access any finite portion of a stream:

```
*> s_print 6 s_nats  ⟹  <0:1:2:3:4:5:..>
```

Raw Streams are too Shallow

The interaction of stream functions coded in this shallow way

- ... has difficult-to-predict grumpy production and consumption behaviour (deadlock and memory leaks)
- ... is scheduled by the Haskell lazy run-time, with little user control on sharing and buffering
- ... does not support destructive memory update, concurrency or IO interactions.

Raw Streams are too Shallow

The interaction of stream functions coded in this shallow way

- ... has difficult-to-predict grumpy production and consumption behaviour (deadlock and memory leaks)
- ... is scheduled by the Haskell lazy run-time, with little user control on sharing and buffering
- ... does not support destructive memory update, concurrency or IO interactions.

How can we make stream functions manage their own memory, permit IO interaction and be schedulable by the application itself?

Raw Streams are too Shallow

The interaction of stream functions coded in this shallow way

- ... has difficult-to-predict grumpy production and consumption behaviour (deadlock and memory leaks)
- ... is scheduled by the Haskell lazy run-time, with little user control on sharing and buffering
- ... does not support destructive memory update, concurrency or IO interactions.

How can we make stream functions manage their own memory, permit IO interaction and be schedulable by the application itself?

Idea: Replace the function type `Str a` $\rightarrow$ `Str b` by an abstract type

`KP a b` (“Kahn Process”)

that schedules the sending and waiting for grumpies in clocked computation cycles to synchronise with memory and IO.

Kahn Processes as Stream Reactive State Machines

Stream Reactive State Machines

Recall the co-iterative semantics of synchronous data flow (Caspi, Pouzet)

```
data SNode s a b = SNode s (s → a → (b, s))
```

Stream Reactive State Machines

Recall the co-iterative semantics of synchronous data flow (Caspi, Pouzet)

```
data SNode s a b = SNode s (s → a → (b, s))
```

Stream reactive machines for asynchronous data flow enrich `SNode`:

```
data SRM m a b = forall s. Applicative m ⇒
  SRM s (s → [a] → ([a], [b], m s))
```

Stream Reactive State Machines

Recall the co-iterative semantics of synchronous data flow (Caspi, Pouzet)

```
data SNode s a b = SNode s (s → a → (b, s))
```

Stream reactive machines for asynchronous data flow enrich `SNode`:

```
data SRM m a b = forall s. Applicative m ⇒
  SRM s (s → [a] → ([a], [b], m s))
```

- abstract (hidden) state type `s`

Stream Reactive State Machines

Recall the co-iterative semantics of synchronous data flow (Caspi, Pouzet)

```
data SNode s a b = SNode s (s → a → (b, s))
```

Stream reactive machines for asynchronous data flow enrich `SNode`:

```
data SRM m a b = forall s. Applicative m ⇒
  SRM s (s → [a] → ([a], [b], m s))
```

- abstract (hidden) state type `s`
- interaction via input and output lists `[a]` and `[b]`

Stream Reactive State Machines

Recall the co-iterative semantics of synchronous data flow (Caspi, Pouzet)

```
data SNode s a b = SNode s (s → a → (b, s))
```

Stream reactive machines for asynchronous data flow enrich `SNode`:

```
data SRM m a b = forall s. Applicative m ⇒
  SRM s (s → [a] → ([a], [b], m s))
```

- abstract (hidden) state type `s`
- interaction via input and output lists `[a]` and `[b]`
- reaction returns unconsumed input values `[a]`

Stream Reactive State Machines

Recall the co-iterative semantics of synchronous data flow (Caspi, Pouzet)

```
data SNode s a b = SNode s (s → a → (b, s))
```

Stream reactive machines for asynchronous data flow enrich `SNode`:

```
data SRM m a b = forall s. Applicative m ⇒
  SRM s (s → [a] → ([a], [b], m s))
```

- abstract (hidden) state type `s`
- interaction via input and output lists `[a]` and `[b]`
- reaction returns unconsumed input values `[a]`
- add state context `m : * → *` for control continuation, memory, IO.

Kahn Processes

Kahn processes are instances of SRM

```
type KP a b = SRM Df a b

data Df a = Pause a  -- for data flow the identity context
-- for more continuation control ...| Terminate | Exit |...
```

Kahn Processes

Kahn processes are instances of SRM

```
type KP a b = SRM Df a b

data Df a = Pause a  -- for data flow the identity context
-- for more continuation control ...| Terminate | Exit |...

class Applicative m where
  pure  :: a → m a
  (<*>) :: m (a → b) → m a → m b
```

Kahn Processes

Kahn processes are instances of SRM

```
type KP a b = SRM Df a b

data Df a = Pause a  -- for data flow the identity context
-- for more continuation control ...| Terminate | Exit |...

class Applicative m where
  pure  :: a → m a
  (<*>) :: m (a → b) → m a → m b
```

Other Instances

```
type IOF a b = SRM IO a b  -- for output and interaction
type PSF a b = SRM PSM a b -- policy-synchronised memory
```

(Haskell PSM presented at Synchron 2019)

Input-less KP = Scheduled Streams

In and Out of KP

```
schStr2SRM :: Str a → Str Int → KP () a
r_val :: KP () a → Str a   -- total value stream
r_clk :: KP () a → Str Int -- max response at each step
```

Input-less KP = Scheduled Streams

In and Out of KP

```
schStr2SRM :: Str a → Str Int → KP () a
r_val :: KP () a → Str a    -- total value stream
r_clk :: KP () a → Str Int -- max response at each step
```

Example: Scheduled Stream of Nats

```
r_nats_c :: KP () Int
r_nats_c = schStr2SRM (iter 0 (+1)) (iter 3 (+0))

*> r_print 5 $ r_nats_c
   ⟹ <[0,1,2]:[3,4,5]:[6,7,8]:[9,10,11]:[12,13,14]:..>

*> s_print 7 $ r_val r_nats_c ⟹ <0:1:2:3:4:5:6:..>
*> s_print 7 $ r_clk r_nats_c ⟹ <3:3:3:3:3:3:3:..>
```

Arrow Wiring with Feedback Loops

KP is not monadic (Kleisli arrow) but has the structure of general arrows ...

```
r_pure  :: (c → b) → KP c b
(>>>)   :: KP b c → KP c d → KP b d
r_first :: KP a b → KP (P a c) (P b c)
(&&&)   :: KP a b → KP a c → K a (P b c)
r_loop  :: KP (P a c) (P b c) → KP a b
```

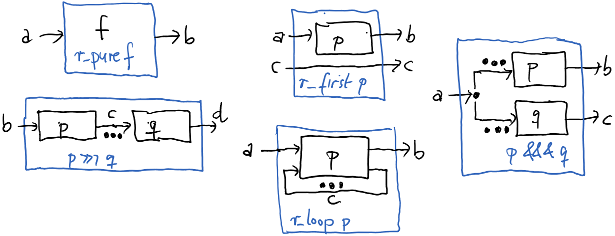
... where P is a pairing that commutes with lists (normal tuples do not work)

Arrow Wiring with Feedback Loops

KP is not monadic (Kleisli arrow) but has the structure of general arrows ...

```
r_pure  :: (c → b) → KP c b
(⋙)     :: KP b c → KP c d → KP b d
r_first :: KP a b → KP (P a c) (P b c)
(&&&)   :: KP a b → KP a c → K a (P b c)
r_loop  :: KP (P a c) (P b c) → KP a b
```

... where P is a pairing that commutes with lists (normal tuples do not work)



Arrow Wiring with Feedback Loops

`KP` is not monadic (Kleisli arrow) but has the structure of general arrows ...

```
r_pure  :: (c → b) → KP c b
(>>>)   :: KP b c → KP c d → KP b d
r_first :: KP a b → KP (P a c) (P b c)
(&&&)   :: KP a b → KP a c → K a (P b c)
r_loop  :: KP (P a c) (P b c) → KP a b
```

... where `P` is a pairing that commutes with lists (normal tuples do not work)

This (slightly) generalises:

- J. Hughes: Programming with Arrows. *Sc. of Comp. Progr.* 2000
- R. Paterson: Arrows and Computation. ICFP'2001
- Hudak, Courtney, Nilsson, Peterson: Arrows, Robots and Functional Reactive Programming. AFP'2002

... related to Freyd and trace monoidal categories (Joyal, Street, Verity).

Extensionality

- Arrow contexts in general are not functionally extensional. There is no "hom-set" equivalence

$$\texttt{Arrow a b} \not\cong \texttt{Arrow () a} \longrightarrow \texttt{Arrow () b}.$$

Internal and external function spaces must be distinguished.

[1] see work by Guatto, Tasson, Vienot

Extensionality

- Arrow contexts in general are not functionally extensional. There is no "hom-set" equivalence

$$\texttt{Arrow a b} \not\cong \texttt{Arrow () a} \rightarrow \texttt{Arrow () b}.$$

Internal and external function spaces must be distinguished.

- This applies also to the standard co-iterative semantics:

$$\texttt{SNode a b} \not\cong \texttt{SNode () a} \rightarrow \texttt{SNode () b} \cong \texttt{Str a} \rightarrow \texttt{Str b}.$$

[1] see work by Guatto, Tasson, Vienot

Extensionality

- Arrow contexts in general are not functionally extensional. There is no "hom-set" equivalence

$$\texttt{Arrow a b} \not\cong \texttt{Arrow () a} \rightarrow \texttt{Arrow () b}.$$

Internal and external function spaces must be distinguished.

- This applies also to the standard co-iterative semantics:

$$\texttt{SNode a b} \not\cong \texttt{SNode () a} \rightarrow \texttt{SNode () b} \cong \texttt{Str a} \rightarrow \texttt{Str b}.$$

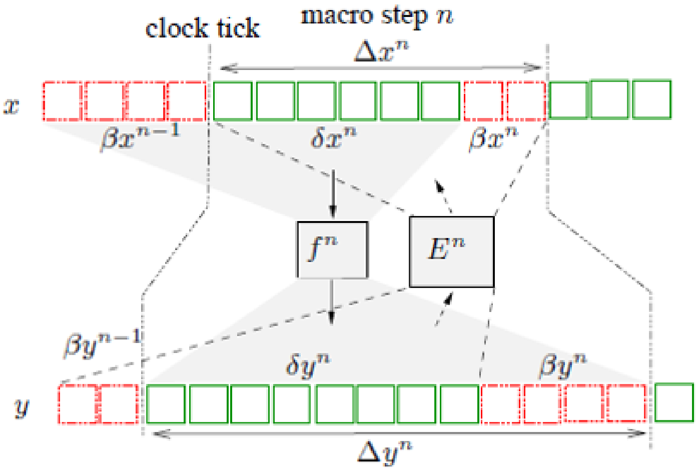
- For Kahn Processes the equivalence makes sense:

$$\texttt{KP a b} \cong \texttt{Str a} \rightarrow \texttt{Str b}$$

A continuous `f :: Str a → Str b` corresponds to a "differential"[1] reactive machine `Delta f :: KP a b`.

[1] see work by Guatto, Tasson, Vienot

Differential Interaction with Environment



Kahn Processes & Control Flow

ArrowPlus Structure

Data Flow: The arrow structure plus primitive building blocks

```
r_fby_n :: a → KP a a               -- initialised delay
r_merge :: KP (P Bool (P a a)) a    -- up-sampling
r_when  :: KP (P a Bool) a          -- down-sampling
```

obtains standard (e.g. Lucid Synchron) data-flow programming.

ArrowPlus Structure

Data Flow: The arrow structure plus primitive building blocks

```
r_fby_n :: a → KP a a                  -- initialised delay
r_merge :: KP (P Bool (P a a)) a       -- up-sampling
r_when  :: KP (P a Bool) a             -- down-sampling
```

obtains standard (e.g. Lucid Synchron) data-flow programming.

Control Flow Operators: KP arrows can also be programmed in Kahn-McQueen control-flow style:

```
r_out   ::  b → KP a b → KP a b        -- sending a value
r_in    :: (a → KP a b) → KP a b       -- receiving a value
r_pause :: KP a b → KP a b             -- pausing
```

ArrowPlus Structure

Data Flow: The arrow structure plus primitive building blocks

```
r_fby_n :: a → KP a a                -- initialised delay
r_merge :: KP (P Bool (P a a)) a     -- up-sampling
r_when  :: KP (P a Bool) a           -- down-sampling
```

obtains standard (e.g. Lucid Synchron) data-flow programming.

Control Flow Operators: KP arrows can also be programmed in Kahn-McQueen control-flow style:

```
r_out   ::  b → KP a b → KP a b      -- sending a value
r_in    :: (a → KP a b) → KP a b     -- receiving a value
r_pause :: KP a b → KP a b           -- pausing

r_srec :: (KP a b → KP a b) → KP a b
r_srec f = p where p = f p           -- state recursion
```

Examples - Output only

We explicitly schedule the value production in bursts:

```
r_halt :: KP a b
r_halt = r_srec r_pause                -- halting

*> r_print 6 r_halt ⟹ <[]:[]:[]:[]:[]:[]:..>
```

Examples - Output only

We explicitly schedule the value production in bursts:

```
r_halt :: KP a b
r_halt = r_srec r_pause              -- halting

*> r_print 6 r_halt ⟹ <[]:[]:[]:[]:[]:[]:..>

ex_4711_1 :: KP a Int
ex_4711_1 =
  r_pause $ r_out 4 $ r_out 7 $ r_pause $
  r_out 1 $ r_pause $ r_pause $ r_out 1 $ r_halt

*> r_print 6 ex_4711_1 ⟹ <[],[4,7],[1],[],[1],[],..>
```

Examples - Input and Output

Synchronous Delay: In each cycle 2 values passed forward

```
r_del_2x2 :: KP Int Int
r_del_2x2 = r_srec $ λend →
  r_in $ λx →                 -- read first value x
  r_in $ λy →                 -- read second y
  r_out x $ r_out y $         -- write first and second
  r_pause $ end               -- pause and repeat
```

Examples - Input and Output

Synchronous Delay: In each cycle 2 values passed forward

```
r_del_2x2 :: KP Int Int
r_del_2x2 = r_srec $ λend →
  r_in $ λx →                 -- read first value x
  r_in $ λy →                 -- read second y
  r_out x $ r_out y $         -- write first and second
  r_pause $ end               -- pause and repeat
```

Asynchronous Wire: pass forward instantaneously

```
r_del_inst :: KP Int Int
r_del_inst = r_srec $ λend →
    r_in $ λx →              -- read value
    r_out x $ end            -- write value, repeat w/o pause
```

Examples

The regularly clocked stream of nats...

```
r_nats_c :: KP () Int
*> r_print 4 $ r_nats_c
   ⟹ <[0,1,2]:[3,4,5]:[6,7,8]:[9,10,11]:..>
```

Examples

The regularly clocked stream of nats...

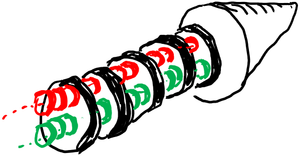
```
r_nats_c :: KP () Int
*> r_print 4 $ r_nats_c
   ⟹ <[0,1,2]:[3,4,5]:[6,7,8]:[9,10,11]:..>
```

... is passed through `r_del_2x2` and `r_del_inst` with different speed

```
*> i_print 5 $ r_nats_c ⋙ r_del_2x2
   ⟹ <[0,1](1):[2,3](2):[4,5](3):[6,7](4):[8,9](5):..>

*> i_print 4 $ r_nats_c ⋙ r_del_inst
   ⟹ <[0,1,2](0):[3,4,5](0):[6,7,8](0):[9,10,11](0):..>
```

Note: values (n) in brackets = buffer size

Stream Lenses:
Unifying Data & Control Flow



Communication Ports in Stream Contexts

A residual lens[2] `RLens a b c` implements an isomorphism of types $\mathtt{a} \cong (\mathtt{b}, \mathtt{c})$. It splits *a* into disjoint pieces `b` and `c` from which *a* can be recombined.

[2] `https://github.com/ekmett/lens/wiki`

Communication Ports in Stream Contexts

A residual lens[2] `RLens a b c` implements an isomorphism of types $\mathtt{a} \cong (\mathtt{b}, \mathtt{c})$. It splits *a* into disjoint pieces `b` and `c` from which *a* can be recombined.

Horizontal Decomposition of Streams

$\mathtt{type\ HPort\ a\ b\ c} = \mathtt{([a]} \rightarrow \mathtt{([b],\ [c]),\ [b]} \rightarrow \mathtt{[c]} \rightarrow \mathtt{[a])}$

implements a horizontal (data-flow) cut $\mathtt{[a]} \cong (\mathtt{[b]}, \mathtt{[c]})$

[2] https://github.com/ekmett/lens/wiki

Communication Ports in Stream Contexts

A residual lens[2] `RLens a b c` implements an isomorphism of types $\mathtt{a} \cong (\mathtt{b}, \mathtt{c})$. It splits *a* into disjoint pieces `b` and `c` from which *a* can be recombined.

Horizontal Decomposition of Streams

```
type HPort a b c = ([a] → ([b], [c]), [b] → [c] → [a])
```

implements a horizontal (data-flow) cut $[\mathtt{a}] \cong ([\mathtt{b}], [\mathtt{c}])$

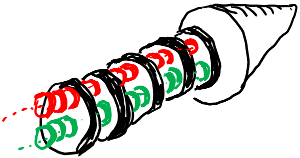
Vertical Decomposition of Streams

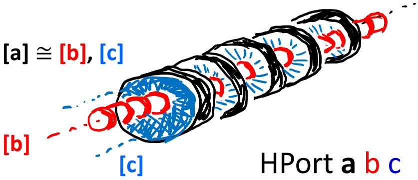
```
type VPort a b =
   ([a] → (Maybe b, [a]), Maybe b → [a] → [a])
```

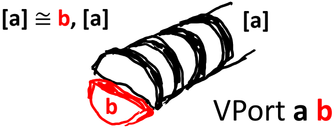
implements a vertical (state) cut $[\mathtt{a}] \cong (\mathtt{b}, [\mathtt{a}])$.

[2] https://github.com/ekmett/lens/wiki

Residual Stream Lenses







Residual Stream Lenses

Port Access Combinators (selected)

```
hFlow :: HPort a a () -- take full flow
hGoUp :: HPort a c b → HPort (P a d) c (P b d) -- go up
hGoDn :: HPort a c b → HPort (P d a) c (P d b) -- go down
```

Residual Stream Lenses

Port Access Combinators (selected)

```
hFlow :: HPort a a () -- take full flow
hGoUp :: HPort a c b → HPort (P a d) c (P b d) -- go up
hGoDn :: HPort a c b → HPort (P d a) c (P d b) -- go down

vState :: VPort a a   -- head slice (state) of flow
vGoUp  :: VPort a c → VPort (P a d) c  -- go up
vGoDn  :: VPort a c → VPort (P d a) c  -- go down
```

Residual Stream Lenses

Port Access Combinators (selected)

```
hFlow :: HPort a a () -- take full flow
hGoUp :: HPort a c b → HPort (P a d) c (P b d) -- go up
hGoDn :: HPort a c b → HPort (P d a) c (P d b) -- go down

vState :: VPort a a    -- head slice (state) of flow
vGoUp  :: VPort a c → VPort (P a d) c  -- go up
vGoDn  :: VPort a c → VPort (P d a) c  -- go down
```

Examples

```
hUp :: HPort (P a c) a c
hUp = hGoUp hFlow          -- pick upper flow

vDn :: VPort (P a c) c
vDn = vGoDn vState        -- head state of lower flow
```

Action Arrows through Lenses

Axiom

```
r_pure :: (a → b) → KP a b
```

Left Cell Introduction (r_first)

```
r_ext :: HPort a b c → KP b d → KP a d
```

Right Cell Introduction (&&&)

```
r_prod :: HPort d a b → KP c a → KP c b → KP c d
```

Feedback (r_loop)

```
r_rec :: HPort a c b → KP a c → KP b c
```

Asynchronous Input and Output

```
r_send :: VPort a c → c → KP d a → KP d a
r_wait :: VPort a c → (c → KP a d) → KP a d
```

Action Arrows through Lenses

Write $p : a \leadsto b$ for `p :: KP a b`.

$$\frac{}{\texttt{r\_id} : a \leadsto a} \qquad \frac{p : b \leadsto d \qquad \texttt{var} : [a] \cong [b], [c]}{\texttt{r\_ext var}\; p : a \leadsto d}$$

$$\frac{p : b \leadsto c \qquad q : c \leadsto d}{p \ggg q : b \leadsto d} \qquad \frac{p : a \leadsto c \qquad \texttt{split} : [a] \cong [c], [b]}{\texttt{r\_rec split}\; p : b \leadsto c}$$

$$\frac{p : c \leadsto a \qquad q : c \leadsto b \qquad \texttt{split} : [d] \cong [a], [b]}{\texttt{r\_prod split}\; p\, q \;: c \leadsto d}$$

$$\frac{p : d \leadsto a \qquad \texttt{var} : [a] \cong c, [a]}{\texttt{r\_send var}\; v\, p : d \leadsto a} \qquad \frac{x : c \vdash p : a \leadsto d \qquad \texttt{var} : [a] \cong c, [a]}{\texttt{r\_wait var}\; \lambda x.\, p : a \leadsto d}$$

Mergesort Example

(suggested by Marc Pouzet)

Mergesort of Streams

```
let node sort x y = c where
  rec xm = current (1 fby c) x
  and ym = current (1 fby (not c)) y
  and clock c  = xm ≤ ym
```

α	1	0	0	0	0	1	1	1	1	0	0
$\alpha_1 = 1 \cdot \alpha$	1	1	0	0	0	0	1	1	1	1	0
x	0	1	$*$	$*$	$*$	$*$	2	3	4	5	$*$
xms = current α_1 *xs*	0	1	1	1	1	1	2	3	4	5	5
$\alpha_2 = 1 \cdot$ not α	1	0	1	1	1	1	0	0	0	0	1
y	0	$*$	0	0	0	4	$*$	$*$	$*$	$*$	4
yms = current α_2 *ys*	0	0	0	0	0	4	4	4	4	4	4
xm $\leq$ ym	1	0	0	0	0	1	1	1	1	0	1
Dir	*B*	*L*	*R*	*R*	*R*	*R*	*L*	*L*	*L*	*L*	*R*

Mergesort of Streams

```
data Dir = L | R | B deriving Show
```

Mergesort of Streams

```
data Dir = L | R | B deriving Show
```

```
-- wait on first stream for a value greater than threshold
-- readx :: Int → KP (P Int Int) Dir
readx y =
  r_wait vUp $ λx →
  r_send vState L $                   -- signal ‘Left’
  if x ≤ y then r_pause $ readx y -- repeat
  else r_pause $ ready x
```

Mergesort of Streams

```
data Dir = L | R | B deriving Show
```

```
-- wait on first stream for a value greater than threshold
-- readx :: Int → KP (P Int Int) Dir
readx y =
  r_wait vUp $ λx →
  r_send vState L $                -- signal ‘Left’
  if x ≤ y then r_pause $ readx y -- repeat
  else r_pause $ ready x
```

```
-- wait on second stream for a value greater than threshold
-- ready :: Int → KP (P Int Int) Dir
ready x =
  r_wait vDn $ λy →
  r_send vState R $                 -- signal ‘Right’
  if y ≤ x then r_pause $ ready x  -- repeat
  else r_pause $ readx y
```

Mergesort of Streams

```
-- 'mergesort' two streams
-- sort :: KP (P Int Int) Dir
sort =
  r_wait vUp $ λx →        -- read first stream
  r_wait vDn $ λy →        -- read second
  r_send vState B $         -- signal 'Both'
  if x ≤ y then
       r_pause $ readx y  -- read 'Left' until larger
  else r_pause $ ready x  -- read 'Right' until larger
```

Mergesort of Streams

Unsynchronised Input Streams

```
lVal, rVal :: Str Int

*> s_print 10 $ lVal  ⟹  <0:1:2:3:4:5:6:7:8:9:..>
*> s_print 10 $ rVal  ⟹  <0:0:0:0:4:4:4:4:8:8:..>
```

Mergesort of Streams

Unsynchronised Input Streams

```
lVal, rVal :: Str Int

*> s_print 10 $ lVal  ⟹  <0:1:2:3:4:5:6:7:8:9:..>
*> s_print 10 $ rVal  ⟹  <0:0:0:0:4:4:4:4:8:8:..>
```

Base Clock

```
baseClk :: Str Int

*> s_print 10 $ baseClk  ⟹  <1:1:1:1:1:1:1:1:1:1:..>
```

Mergesort of Streams

Unsynchronised Input Streams

```
lVal, rVal :: Str Int
```

```
*> s_print 10 $ lVal  ⟹ <0:1:2:3:4:5:6:7:8:9:..>
*> s_print 10 $ rVal  ⟹ <0:0:0:0:4:4:4:4:8:8:..>
```

Base Clock

```
baseClk :: Str Int
```

```
*> s_print 10 $ baseClk  ⟹ <1:1:1:1:1:1:1:1:1:1:..>
```

Input Streams Synchronised at Base Clock

```
lBStr = schStr2SRM lVal baseClk
rBStr = schStr2SRM rVal baseClk
```

Mergesort of Streams

Input Streams Synchronised at Base Clock

```
*> r_print 10 $ lBStr
⟹ <[0]:[1]:[2]:[3]:[4]:[5]:[6]:[7]:[8]:[9]:..>
*> r_print 10 $ rBStr
⟹ <[0]:[0]:[0]:[0]:[4]:[4]:[4]:[4]:[8]:[8]:..>
```

Mergesort of Streams

Input Streams Synchronised at Base Clock

```
*> r_print 10 $ lBStr
⟹ <[0]:[1]:[2]:[3]:[4]:[5]:[6]:[7]:[8]:[9]:..>
*> r_print 10 $ rBStr
⟹ <[0]:[0]:[0]:[0]:[4]:[4]:[4]:[4]:[8]:[8]:..>
```

`lBStr` and `rBStr` get merged with a rythm depending on the streams' values (needs internal buffering)

```
*> r_print 10 $ (lBStr &&& rBStr) ⋙ sort
⟹ <[B]:[L]:[R]:[R]:[R]:[R]:[L]:[L]:[L]:[L]:..>
*> i_print 10 $ (lBStr &&& rBStr) ⋙ sort
⟹ <[B](0):[L](1):[R](1):[R](2):[R](3):[R](4):[L](4):[L](4):[L]
```

Mergesort of Streams

Input Streams Synchronised at Base Clock

```
*> r_print 10 $ lBStr
⟹ <[0]:[1]:[2]:[3]:[4]:[5]:[6]:[7]:[8]:[9]:..>
*> r_print 10 $ rBStr
⟹ <[0]:[0]:[0]:[0]:[4]:[4]:[4]:[4]:[8]:[8]:..>
```

`lBStr` and `rBStr` get merged with a rythm depending on the streams' values (needs internal buffering)

```
*> r_print 10 $ (lBStr &&& rBStr) >>> sort
⟹ <[B]:[L]:[R]:[R]:[R]:[R]:[L]:[L]:[L]:[L]:..>
*> i_print 10 $ (lBStr &&& rBStr) >>> sort
⟹ <[B](0):[L](1):[R](1):[R](2):[R](3):[R](4):[L](4):[L](4):[L]
```

Input Consumption Rates

```
*> s_print 10 $ lClk  ⟹ <1:1:0:0:0:0:1:1:1:1:..>
*> s_print 10 $ rClk  ⟹ <1:0:1:1:1:1:0:0:0:0:..>
```

Mergesort of Streams

The value streams synchronised according to their consumption rates:

```
-- lClk, rClk :: Str Int
lCStr, rCStr :: RFArrow m rsm ⇒ rsm m () Int
lCStr = schStr2SRM lVal lClk
rCStr = schStr2SRM rVal rClk

*> r_print 10 $ lCStr
⟹ <[0]:[1]:[]:[]:[]:[]:[2]:[3]:[4]:[5]:..>
*> r_print 13 $ rCStr
⟹ <[0]:[]:[0]:[0]:[0]:[4]:[]:[]:[]:[]:[4]:[4]:[4]:..>
```

Mergesort of Streams

The value streams synchronised according to their consumption rates:

```
-- lClk, rClk :: Str Int
lCStr, rCStr :: RFArrow m rsm ⇒ rsm m () Int
lCStr = schStr2SRM lVal lClk
rCStr = schStr2SRM rVal rClk

*> r_print 10 $ lCStr
⟹ <[0]:[1]:[]:[]:[]:[]:[2]:[3]:[4]:[5]:..>
*> r_print 13 $ rCStr
⟹ <[0]:[]:[0]:[0]:[0]:[4]:[]:[]:[]:[]:[4]:[4]:[4]:..>
```

Now we can operate without buffering

```
*> r_print 13 $ (lCStr &&& rCStr) ⋙ sort
⟹ <[B]:[L]:[R]:[R]:[R]:[R]:[L]:[L]:[L]:[L]:[R]:[R]:[R]:..>
*> i_print 13 $ (lCStr &&& rCStr) ⋙ sort
⟹ <[B](0):[L](0):[R](0):[R](0):[R](0):[R](0):[L](0):[L](0):[L]
```

Clock Typing of Mergesort?

If `sort :: KP (P Int Int) Dir` corresponds to a stream function

`sort :: Str Int` $\rightarrow$ `Str Int` $\rightarrow$ `Str Dir,`

then what is its clock type?

Clock Typing of Mergesort?

If `sort :: KP (P Int Int) Dir` corresponds to a stream function

`sort :: Str Int` $\to$ `Str Int` $\to$ `Str Dir`,

then what is its clock type?

- We extend clock type schemes σ by event types $\Sigma\alpha.\,\sigma$ and class constraints $C \Rightarrow \sigma$.

Clock Typing of Mergesort?

If `sort :: KP (P Int Int) Dir` corresponds to a stream function

`sort :: Str Int` $\rightarrow$ `Str Int` $\rightarrow$ `Str Dir`,

then what is its clock type?

- We extend clock type schemes σ by event types $\Sigma\alpha.\,\sigma$ and class constraints $C \Rightarrow \sigma$.

$$\texttt{sort} \quad : \quad \forall\alpha.\,\forall\alpha_1.\,\forall\alpha_2.\,\alpha_1 \rightarrow \alpha_2 \rightarrow \Sigma\alpha_3.\,C \Rightarrow \underline{1}$$

$$C \quad =_{\mathrm{df}} \quad \alpha_1 = \alpha \circ (1 \cdot \alpha_3) \wedge (\alpha_2 = \alpha \circ (1 \cdot \texttt{not}\,\alpha_3)).$$

Conclusion

Summary

- Lazy lists `[a]` as finite approximations of `Str a` to generate a schedulable encoding of `Str a` → `Str b` as an arrow `KP a b` (Haskell is algebraically compact)
- Stream lenses for uniform data and control flow programming.

Conclusion

Summary

- Lazy lists `[a]` as finite approximations of `Str a` to generate a schedulable encoding of `Str a` $\rightarrow$ `Str b` as an arrow `KP a b` (Haskell is algebraically compact)
- Stream lenses for uniform data and control flow programming.

Questions

- Are clocks properties of streams or properties of arrows?
- What is the clock of a multi-input, multi-output `KP` process?
- Can we give a clock type system that directly types the control flow primitives?

Stopwatch (simplified)

Synchronous Read and Write

Input ? and output ! act on the first stream value of the interface cell. We assume that only one value is consumed and produced per cycle.

```
(?) :: VPort a c → (c → KP a d) → KP a d
(!) :: VPort a c → c → KP d a → KP d a
```

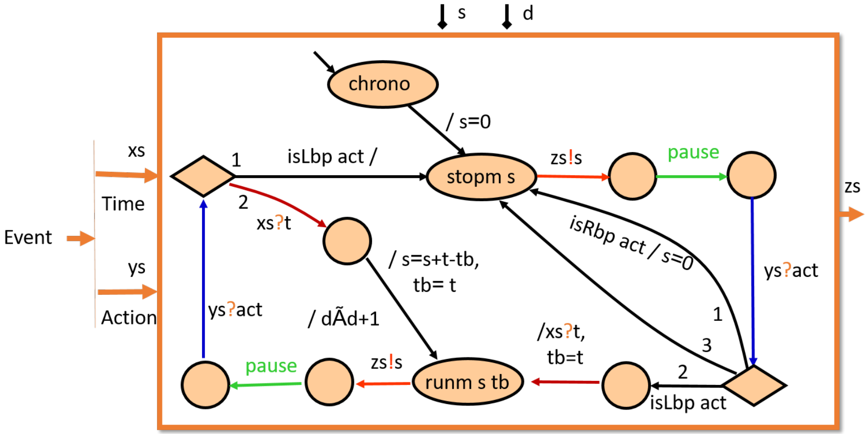
Demo

```
chrono :: KP Event Double
chrono = stopm 0                        -- the controller

flToDig :: KP Double Graphic
flToDig = r_lift1_n floatToDigits       -- the rendering

chronogr :: KP Event Graphic
chronogr = chrono ⋙ flToDig            -- assembly

react $ rdf_run chronogr                -- running the animation
```

Stopwatch (simplified)

```
stopm :: Double → KP Event Double
stopm s =
  vHd hFlow ! s $ r_pause $
  vHd hFlow ? λ(act,_) →
  if isRbp act then stopm 0
  else if isLbp act then (vHd hFlow ? λ(_,t) → runm s t)
       else stopm s

runm :: Double → Double → KP Event Double
runm s tb =
  vHd hFlow ! s $ r_pause $
  vHd hFlow ? λ(act,_) →
  if isLbp act then stopm s
  else (vHd hFlow ? λ(_,t) → runm (s + t - tb) t)
```

(simplified)


s
d
chrono
/ s=0
1
isLbp act /
stopm s
zs!s
pause
zs
xs
Time
Event
ys
Action
2
xs?t
/ s=s+t-tb,
tb= t
isRbp act / s=0
ys?act
ys?act
/ dÃd+1
1
3
/xs?t,
tb=t
2
pause
zs!s
runm s tb
isLbp act

Pairing & Projection -bkup-

In contrast to general arrows, `KP` also has a natural product structure:

```
(&&&)  :: KP c a → KP c b → KP c (P a b)
r_fst  :: KP (P a b) a
r_snd  :: KP (P a b) b
```

```
*> r_print 5 $ r_nats_e ⋙ (r_del_2x2 &&& r_del_inst)
 ⟹ <[]:[(∗,0)]:[(0,1),(1,2)]:[(2,3),(3,4),(∗,5)]:
       [(4,6),(5,7),(∗,8),(∗,9)]:..>
```

```
*> r_print 5 $ r_nats_e ⋙ (r_del_2x2 &&& r_del_inst) ⋙ r_fst
 ⟹ <[]:[]:[0,1]:[2,3]:[4,5]:..>
```

```
*> r_print 5 $ r_nats_e ⋙ (r_del_2x2 &&& r_del_inst) ⋙ r_snd
 ⟹ <[]:[0]:[1,2]:[3,4,5]:[6,7,8,9]:..>
```

Tensorial Pairing -bkup-

The product structure depends on "lazy tensorial" pairing which permits us to confuse a pair of lists with a list of pairs.

We use the `Maybe` monad to fill up missing list elements by a dummy value `Nothing` which is going to be printed as "$*$"

```
data P a b = P (Maybe a) (Maybe b)

sfst  :: [P a b] → [a]
ssnd  :: [P a b] → [b]
spair :: [a] → [b] → [P a b]

*> spair [1,2,3] [1,2,3,4,5,6,7]
   ⟹ [(1,1),(2,2),(3,3),(∗,4),(∗,5),(∗,6),(∗,7)]
*> sfst $ spair [1,2,3] [1,2,3,4,5,6,7]  ⟹ [1,2,3]
*> ssnd $ spair [1,2,3] [1,2,3,4,5,6,7]  ⟹ [1,2,3,4,5,6,7]
```