Data Parallel Data Flow in Repa 4

Ben Lippmeier

FP-Syd 2015/2/25

Image: A♥ / Aih. flickr. CC Generic.

Intro demo.

```
> import Data.Repa.Flow as F
> ws <- F.fromFiles
            [ “/usr/share/dict/words”
            , “/usr/share/dict/connectives” ]
            F.sourceLines

> F.sourcesArity ws
2
```

A flow consists of a bundle of individual streams. We create a bundle two stream sources, one for each file.

```
> :type ws

ws :: Sources N (Array F Char)

> :type N
> :type F

N :: Name N
F :: Name F
```

The layout name controls the the representation of the chunks that make up the streams.

(Nested Arrays)

(Foreign Arrays)

```
> print F.defaultChunkSize
65536

> import Data.Repa.Flow.Default.Debug
> more 0 ws

Just [“A”, “a", “aa", “aal", “aalii", “aam",
“Aani", “aardvark", “aardwolf", “Aaron", …
```

The more function shows the first few elements from the front of the next chunk. The streams are stateful, so pulling a chunk consumes it.

```
> more’ 0 100 ws

Just [“arbitrament”, “arbitrarily",
“arbitrariness", “arbitrary", “arbitrate”, …
```

```
> moret 1 ws

“the”
“of”
“and”
“to”
“a”
“in”
“that”
“is”
“was”

…
```

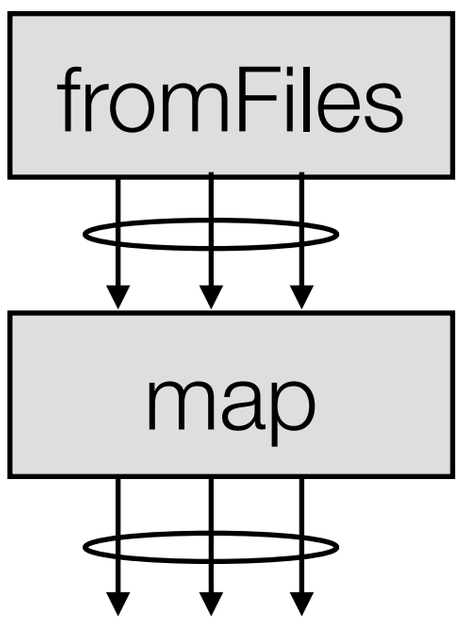
Show the next chunk of the second stream as a table.

```
> import Data.Char
> up <- map_i B (mapS U toUpper) ws
> more 0 up

Just [“BARRISTRESS", “BARROOM", “BARROW",
“BARROWFUL", “BARROWIST", “BARROWMAN",
“BARRULEE", “BARRULET", “BARRULETY", “BARRULY",
“BARRY", “BARRY", “BARSAC", “BARSE”, …
```

Apply a function to every element of the stream.
B = Boxed. U = Unboxed. map(S) ~ Sequential.
map(_i) ~ input version (more on this later).

```
> :type up

up :: Sources B (Array U Char)
```

```
> more 0 up

Just [“BARRISTRESS", “BARROOM", “BARROW",
“BARROWFUL", “BARROWIST", “BARROWMAN",
“BARRULEE", “BARRULET", “BARRULETY", “BARRULY",
“BARRY", “BARRY", “BARSAC", “BARSE”, …

> more 1 up

Just [“THE”, “OF", “AND", “TO", “A", “IN",
“THAT", “IS", “WAS", “HE", “FOR", “IT", ..
```



Flows are data parallel, so applying a function like map_i transforms all streams in the bundle.

```
:!mkdir -p tmp

> out <- toFiles [ "tmp/out1.txt"
                 , "tmp/out2.txt" ]
                 $ sinkLines B U

> :type out

out :: Sinks B (Array U Char)
```

Now we have a bundle of stream sinks.
Data pushed into the sinks gets written out
to the above files as text lines.

```
> :type drainP

drainP :: Source l a -> Sinks l a -> IO ()
```

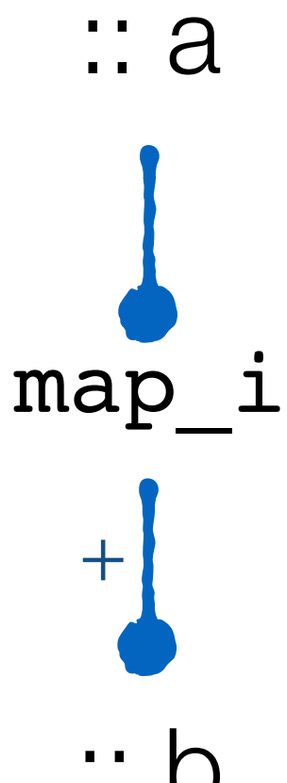
Drain all data from the sources into
the sinks, in parallel.

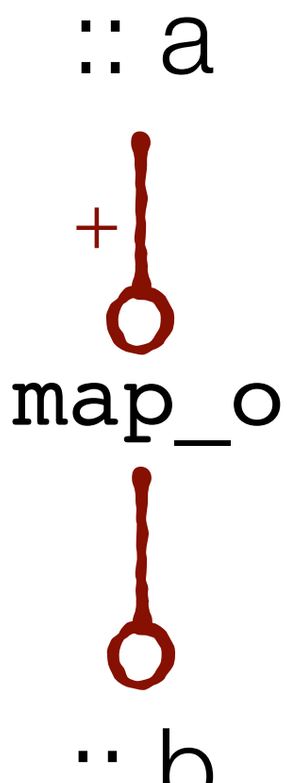
Polarity

Image: Axel Taferner. flickr. CC-NC-SA.

pull from output
induces
pull from input

"pully"





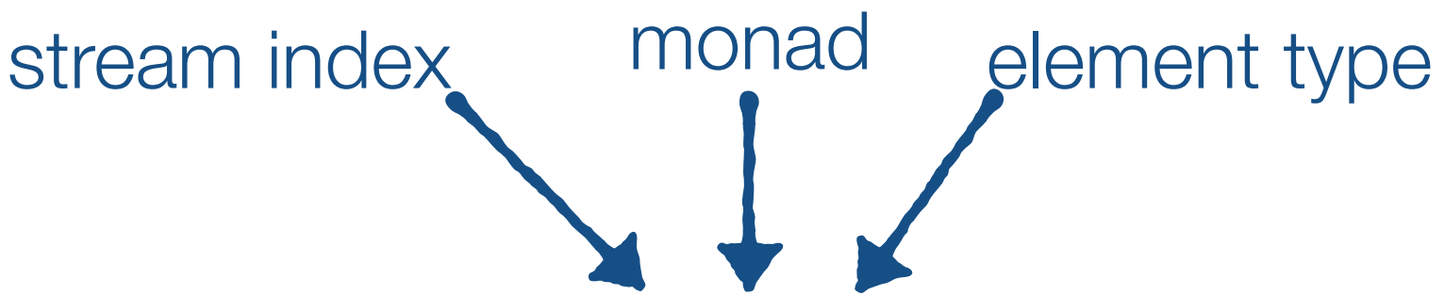
push to input
induces
push to output

"pushy"

```
map_i   :: Name l2 -> (a -> b)
        -> Sources l1 a -> m (Sources l2 b)

map_o   :: Name l2 -> (a -> b)
        -> Sinks l1 b   -> m (Sinks   l2 a)
```

"contramap"

```
module Data.Repa.Flow.Generic where
```




```
data Sources i m e
    = Sources
    { sourcesArity :: i
    , sourcesPull  :: i -> (e -> m ()) -> m ()
                   -> m ()}
```

eat

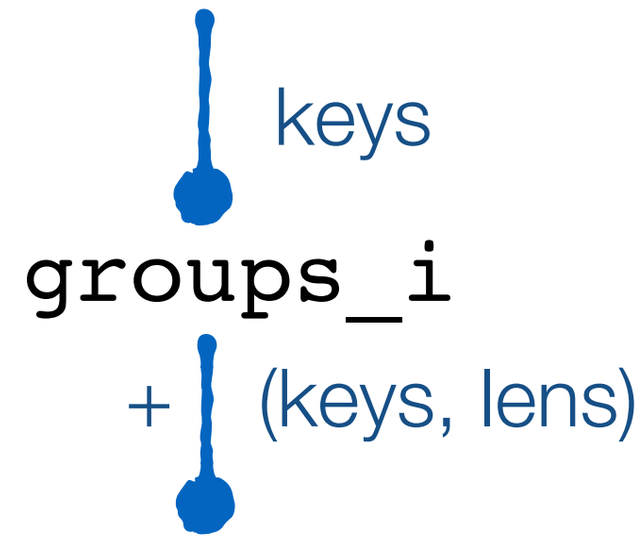
eject

```
data Sinks i m e
    = Sinks
    { sinksArity :: i
    , sinksPush  :: i -> e -> m ()
    , sinksEject :: i -> m () }
```

```
module Data.Repa.Flow.Chunked where
import Data.Repa.Flow.Generic as G

type Sources l a = G.Sources Int IO (Array l a)

type Sinks   l a = G.Sinks   Int IO (Array l a)
```

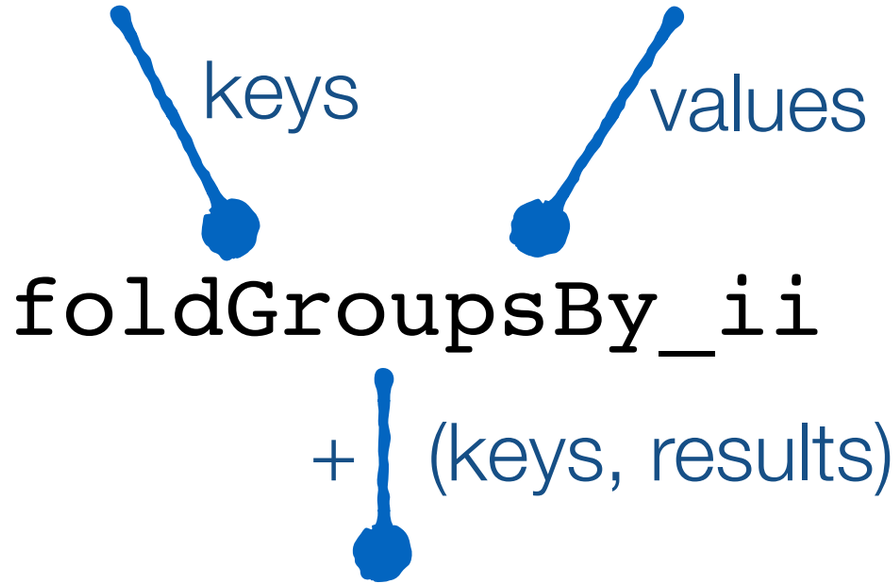
The repa-flow packages defines generic flows, then various instances with a more specific/simpler API.

```
groups_i
  :: Name lGrp
  -> Name lLen
  -> (a -> a -> Bool)
  -> Sources lVal a
  -> IO (Sources (T2 lGrp lLen) (a, Int))
```



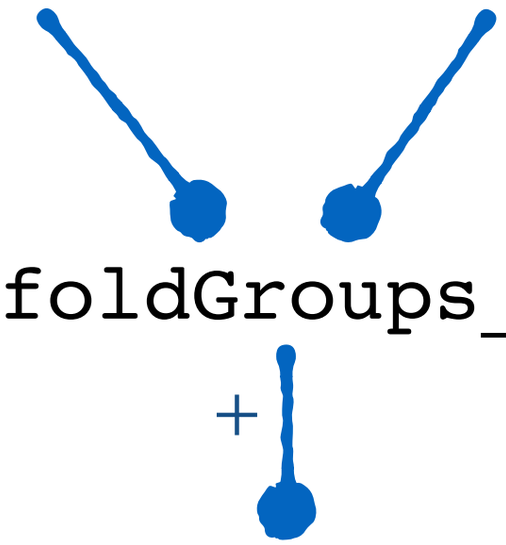

```
> toList1 0 =<< groupsBy_i U U (==)
            =<< fromList U 1 “waabbbbllee”

Just [ (‘w’, 1), (‘a’, 2)
     , (‘b’, 4), (‘l’, 2), (‘e’, 2)]
```

```
foldGroupsBy_i
 :: Name lGrp -> Name lRes
 => (n -> n -> Bool)
 -> (a -> b -> b)
 -> b
 -> Sources lSeg n
 -> Sources lVal a
 -> IO (Sources (T2 lGrp lRes) (n, b))
```

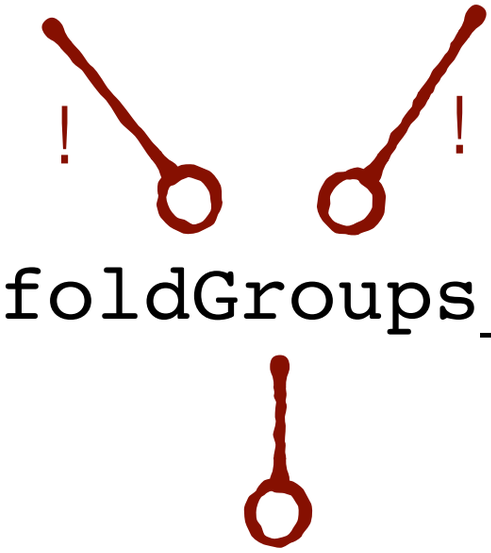



```
> sKeys <- fromList U 1 "waaaablle"
> sVals <- fromList U 1 [10, 20, 30, 40, 50 …
> toList1 0
    =<< map_i U (\(key, (acc, n)) -> (key, acc / n))
    =<< foldGroupsBy_i U U (==)
          (\x (acc, n) -> (acc + x, n + 1))
          (0, 0) sKeys sVals

Just [('w', 10.0), ('a', 35.0), ('b', 60.0) …
```

OK



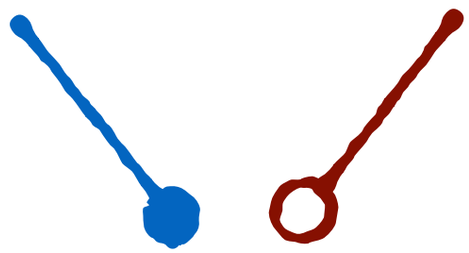
`foldGroups_ii`



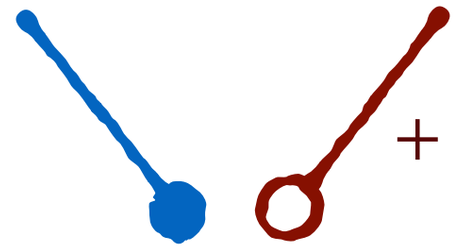
`foldGroups_oo`

Buffers

Blocks



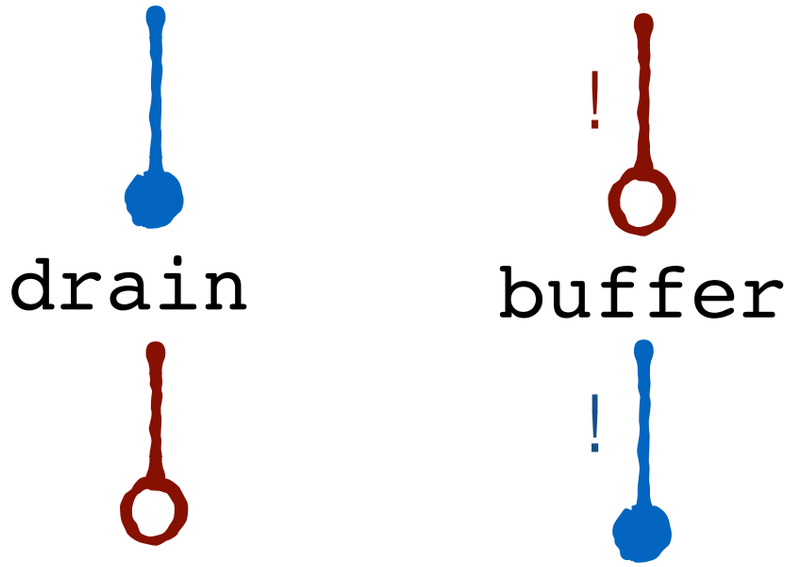
`foldGroups_xx`



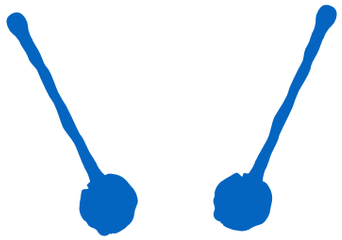
`foldGroups_io`

OK

```
foldGroups_ii :: .. -> Src k -> Src a -> Src (k, b)
foldGroups_oo :: .. -> Snk k -> Snk a -> Snk (k, b)
foldGroups_xx :: .. -> Src k -> Snk a -> Src (k, b)
foldGroups_io :: .. -> Src k -> Snk a -> Snk (k, b)
```


drain
!
buffer
!

(a * b)

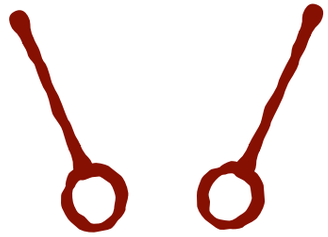


`zipDrain`

`drain`

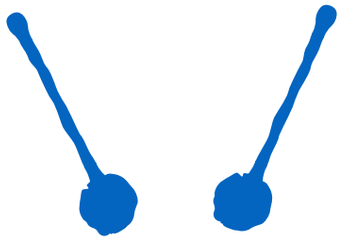
`buffer`

(a + b)



`altBuffer`

(a * b)

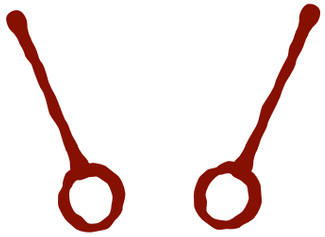


zipDrain

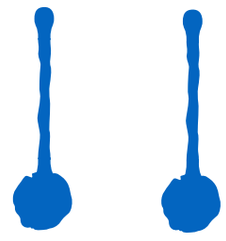
drain

buffer

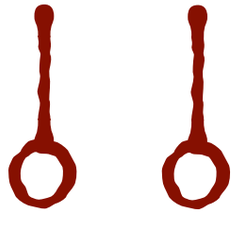
(a + b)



altBuffer

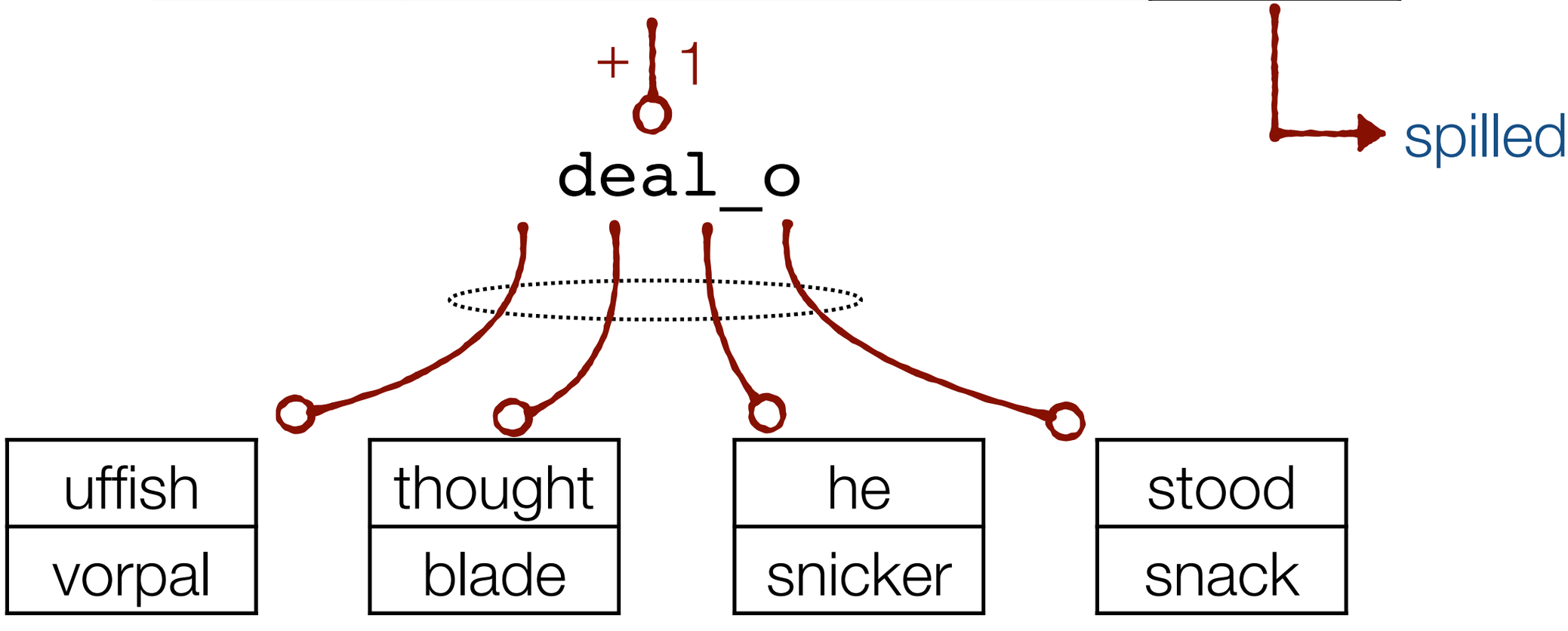


drain2

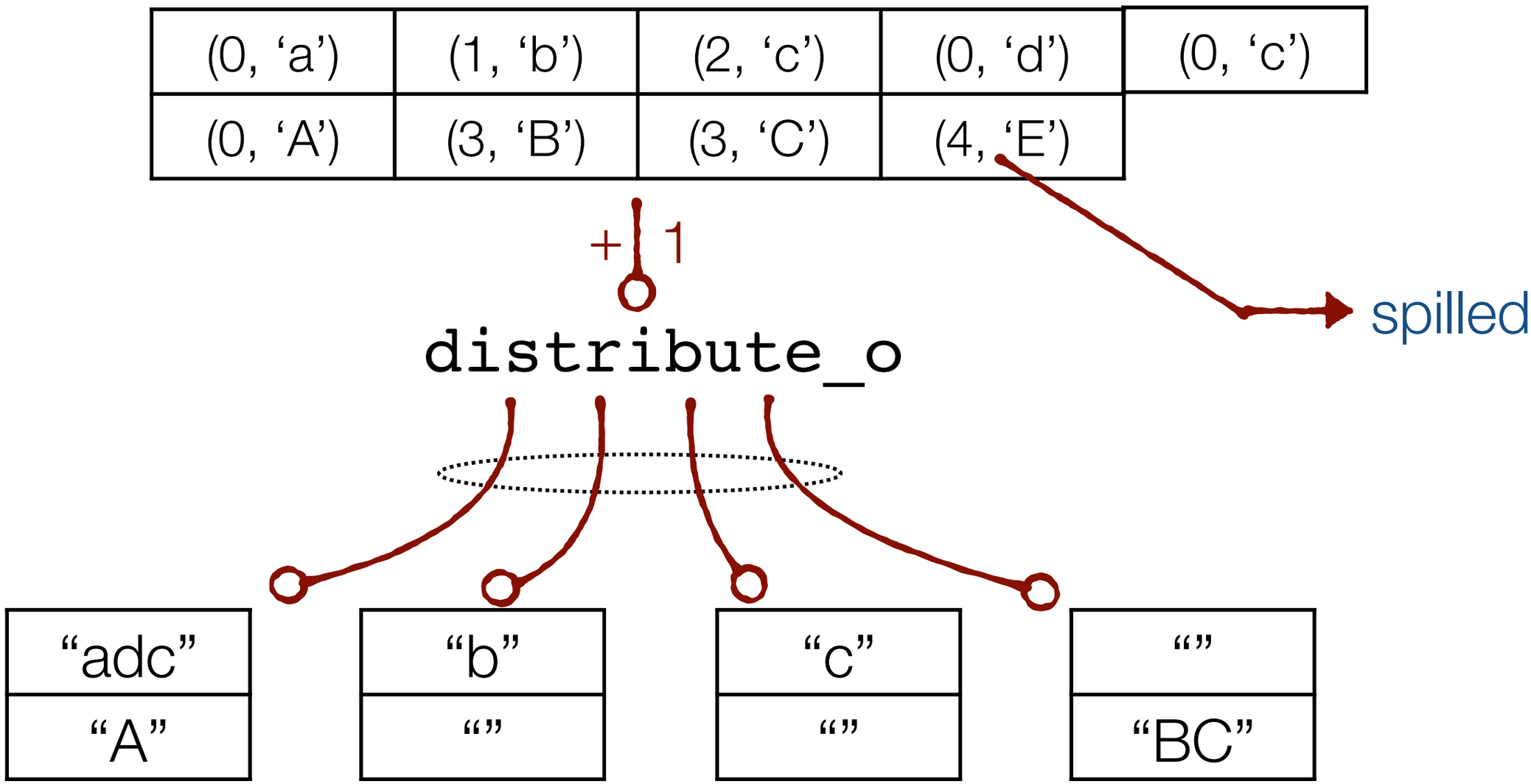


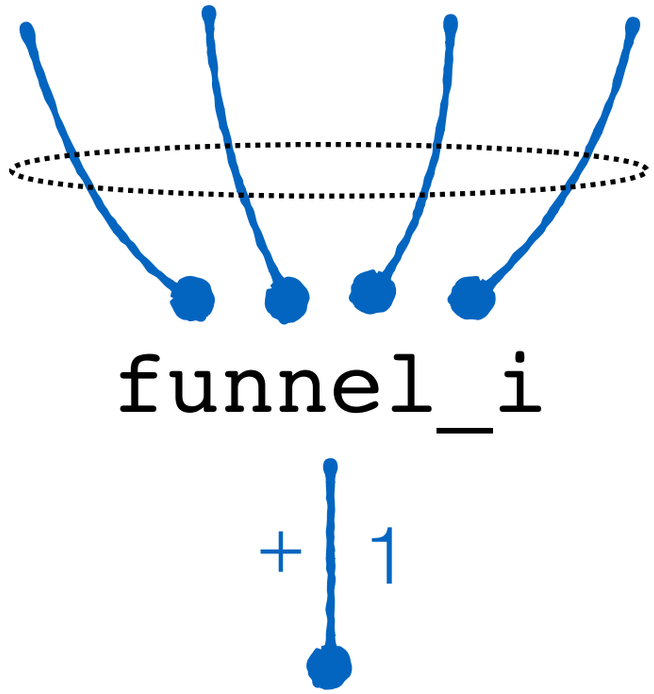
```
deal_o
 :: (Int -> a -> IO ())                    (spill function)
 -> Sinks Int IO a                         (output)
 -> IO (Sinks () IO (Array l a))
```

uffish	thought	he	stood	
vorpal	blade	snicker	snack	burbled



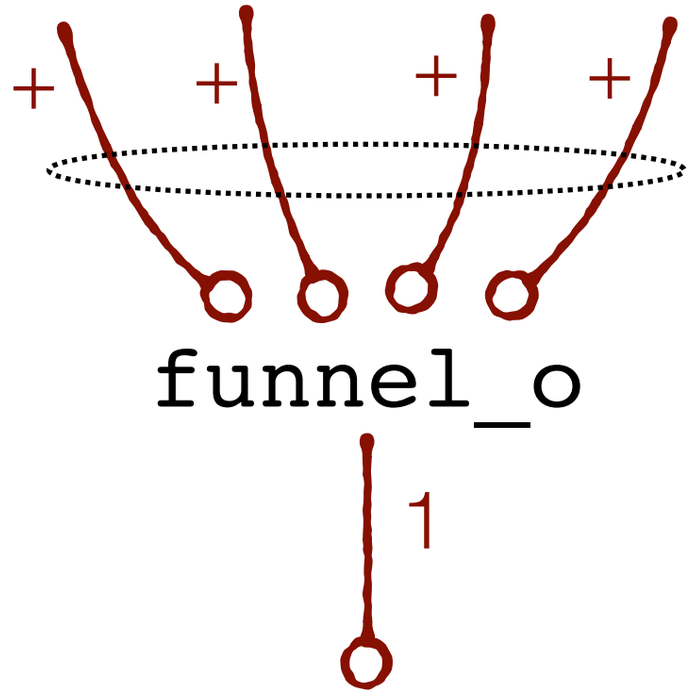
```
distribute_o
 :: Name lSrc
 -> (Int -> Array lDst a -> IO ())
 -> Sinks Int IO (Array lDst a)
 -> IO (Sinks () IO (Array lSrc (Int, a)))
```





naturally sequential
read from the input streams
one after the other

controlled order of consumption
drain entire stream first,
or round robin element-wise

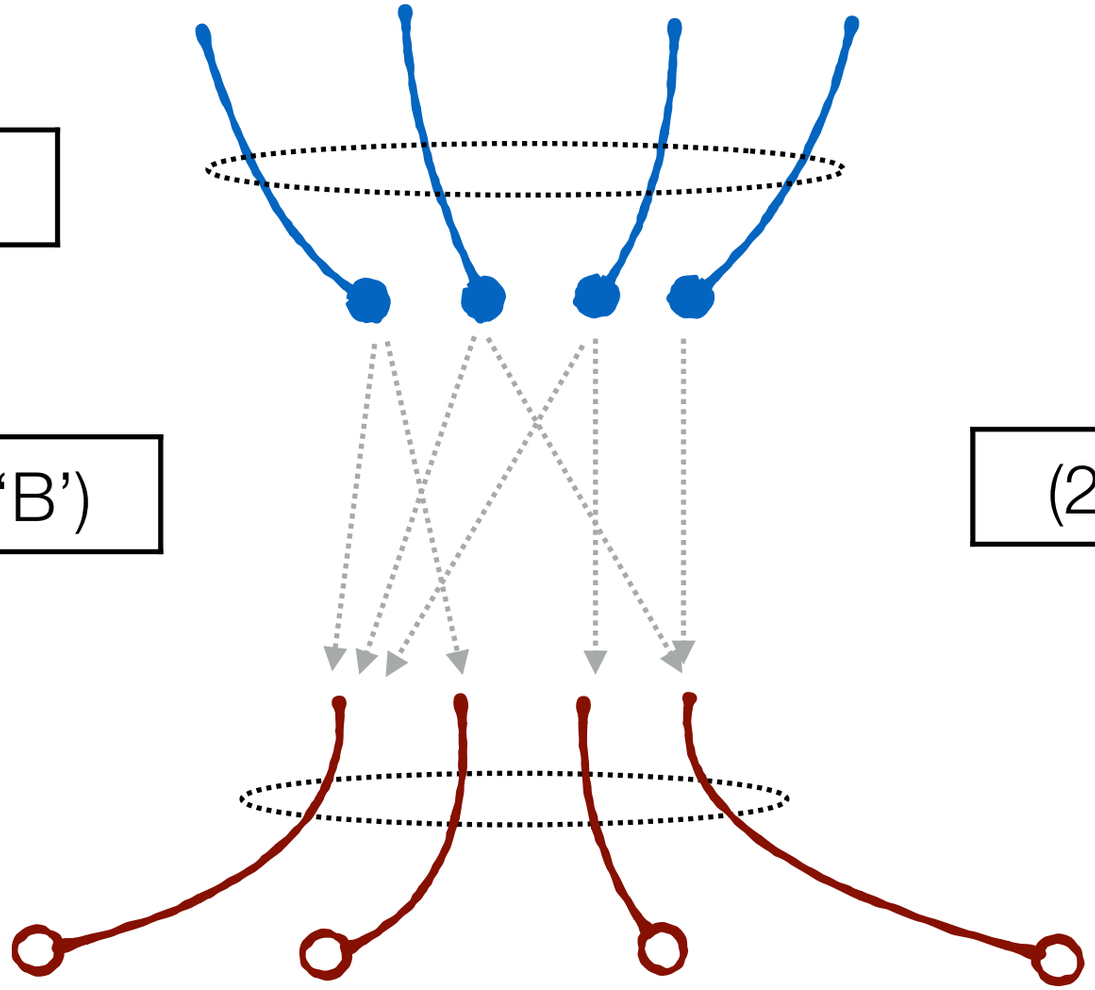


naturally concurrent
input streams are contending
for a shared output

uncontrolled order of consumption
elements pushed in
non-deterministic order

(0, ‘a’)	(1, ‘b’)

(0, ‘A’)	(3, ‘B’)

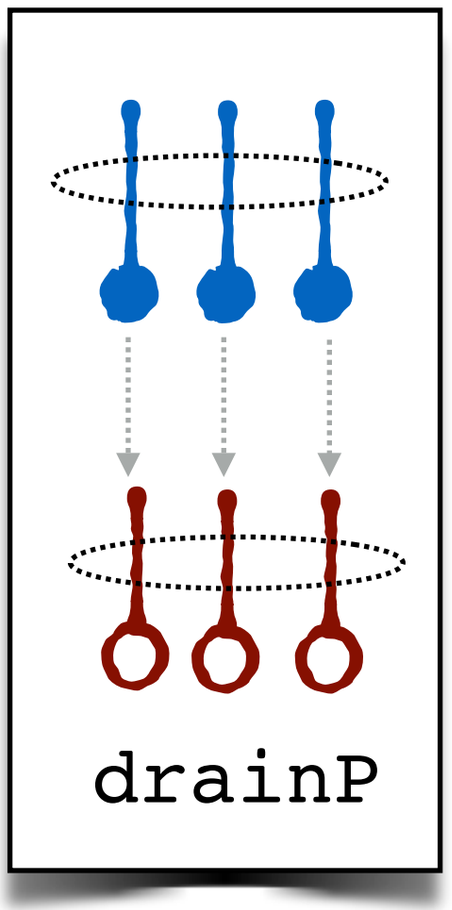


shuffleP

(3, ‘C’)	(4, ‘E’)

(2, ‘c’)	(0, ‘d’)

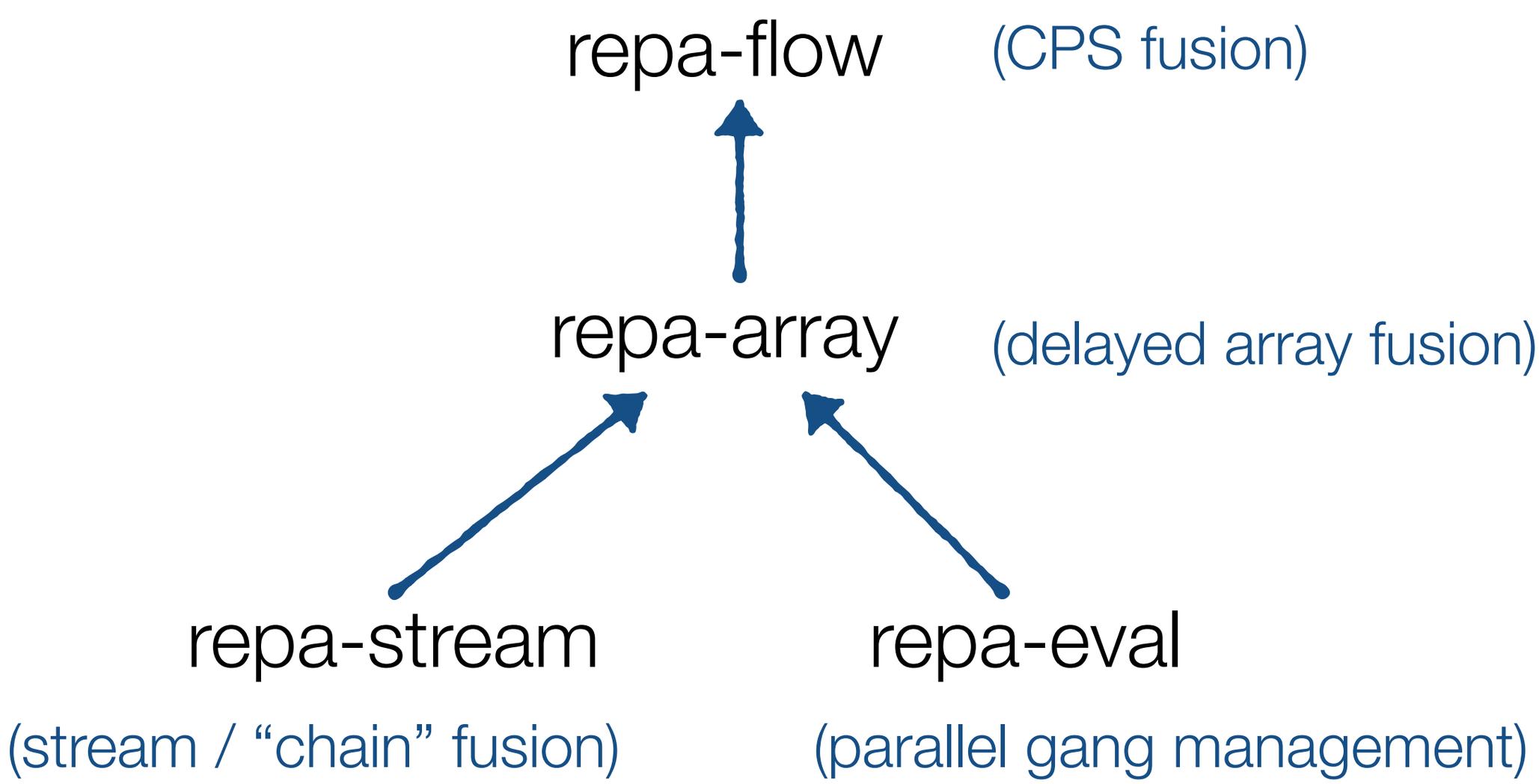
add type of shuffle:



drainP

Implementation

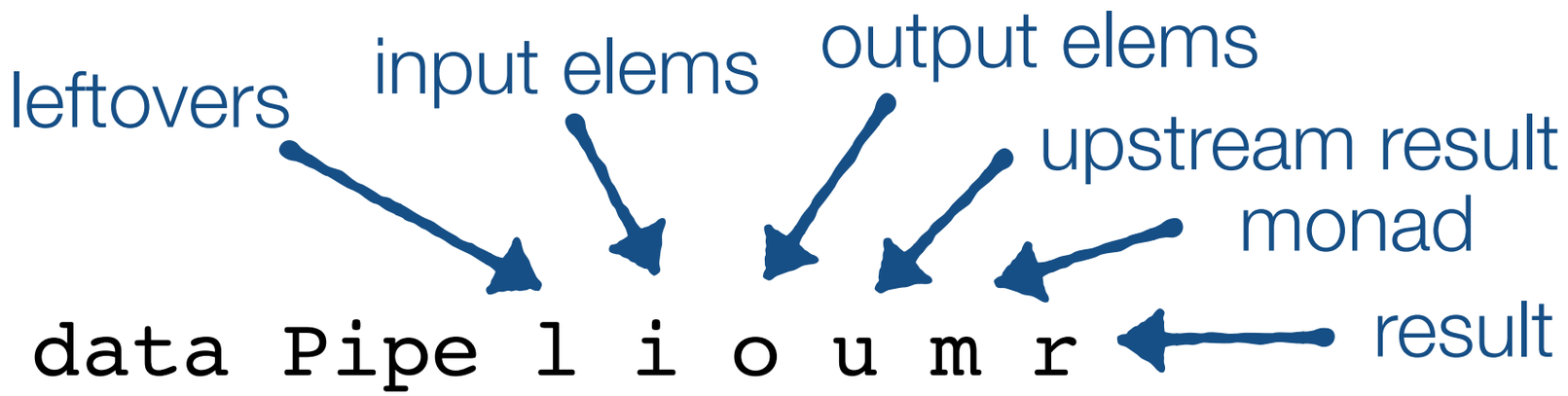
Image: gullevek. flickr. CC-NC-SA.


repa-flow
(CPS fusion)
repa-array
(delayed array fusion)
repa-stream
(stream / “chain” fusion)
repa-eval
(parallel gang management)

Code exploration.

Comparison

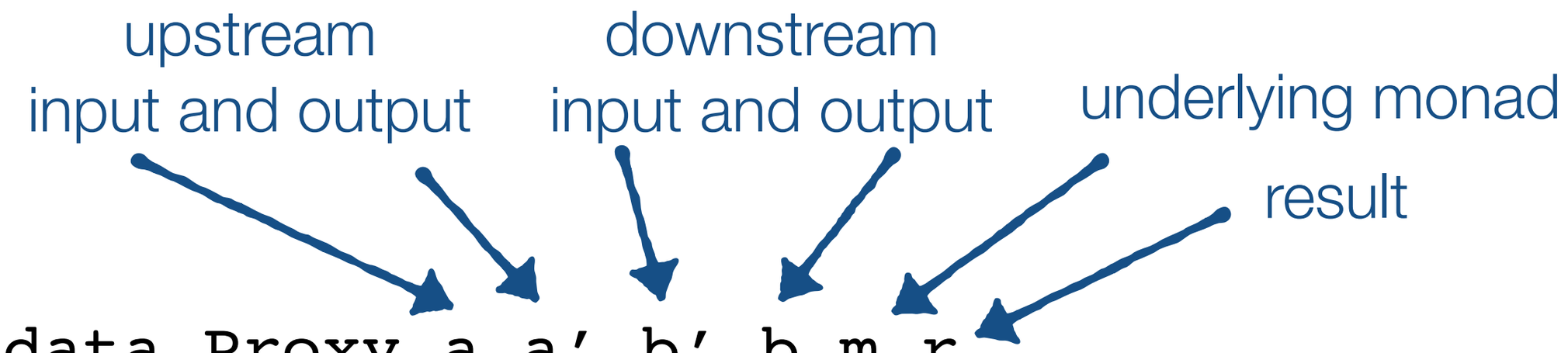
conduit – *Michael Snoyman*




```
data Pipe l i o u m r
  = HaveOutput (Pipe l i o u m r) (m ()) o
  | NeedInput  (i -> Pipe l i o u m r)
               (u -> Pipe l i o u m r)
  | Done r
  | PipeM (m (Pipe l i o u m r))
  | Leftover (Pipe l i o u m r))
```

- Pipe is an instance of Monad.
- Data can flow both ways through the pipe, and yield a final result.
- Single stream, single element at a time.
- Individual Sources created by ‘yield’ action.
- Combine pipes/conduits with fusion operators.

pipes – *Gabriel Gonzelez*




```
data Proxy a a' b' b m r

  = Request a' (a  -> Proxy a' a b' b m r)
  | Respond b  (b' -> Proxy a' a b' b m r)
  | M          (m    (Proxy a' a b' b m r))
  | Pure r
```

- Proxy / Pipe is an instance of Monad.
- Data can flow both ways through the pipe, and yield a final result.

machines – *Edward Kmett*

```
newtype MachineT m k o
  = MachineT
  { runMachine :: m (Step k o (MachineT m k o))

type Machine k o
  = forall m. Monad m => MachineT m k o

type Process a b = Machine (Is a) b)

type Source b    = forall k. Machine k b
```

- Like streams as used in Data.Vector stream fusion,
  except the step function returns a whole new Machine (stream)

- Clean and general API, but not sure if it supports array fusion.
  Machines library does not seem to attempt fusion.

`repa-flow vs others`

- Repa flow provides chunked, data parallel database-like operators with a straightforward API.

- Sources and Sinks are values rather than computations. The “Pipe” between them created implicitly in IO land.

- API focuses on simplicity and performance via stream and array fusion, rather than having the most general API.

- Suspect we could wrap single-stream Repa flow operators as either Pipes or Conduits, but neither of the former seem to naturally support data parallel flows.

α-quality, active development
code that's there should work ok,
but still some missing components

`https://github.com/DDCSF/repa`