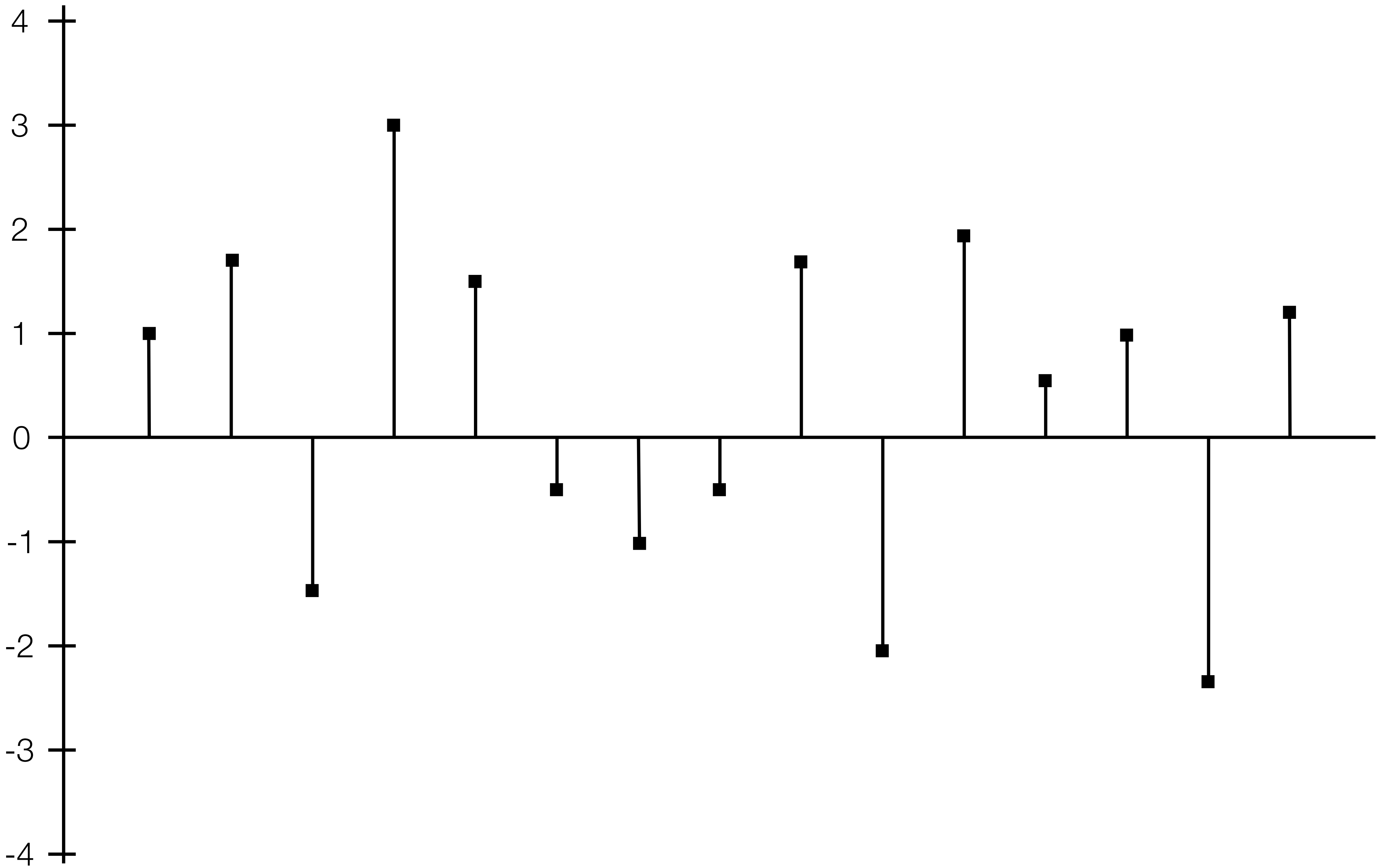
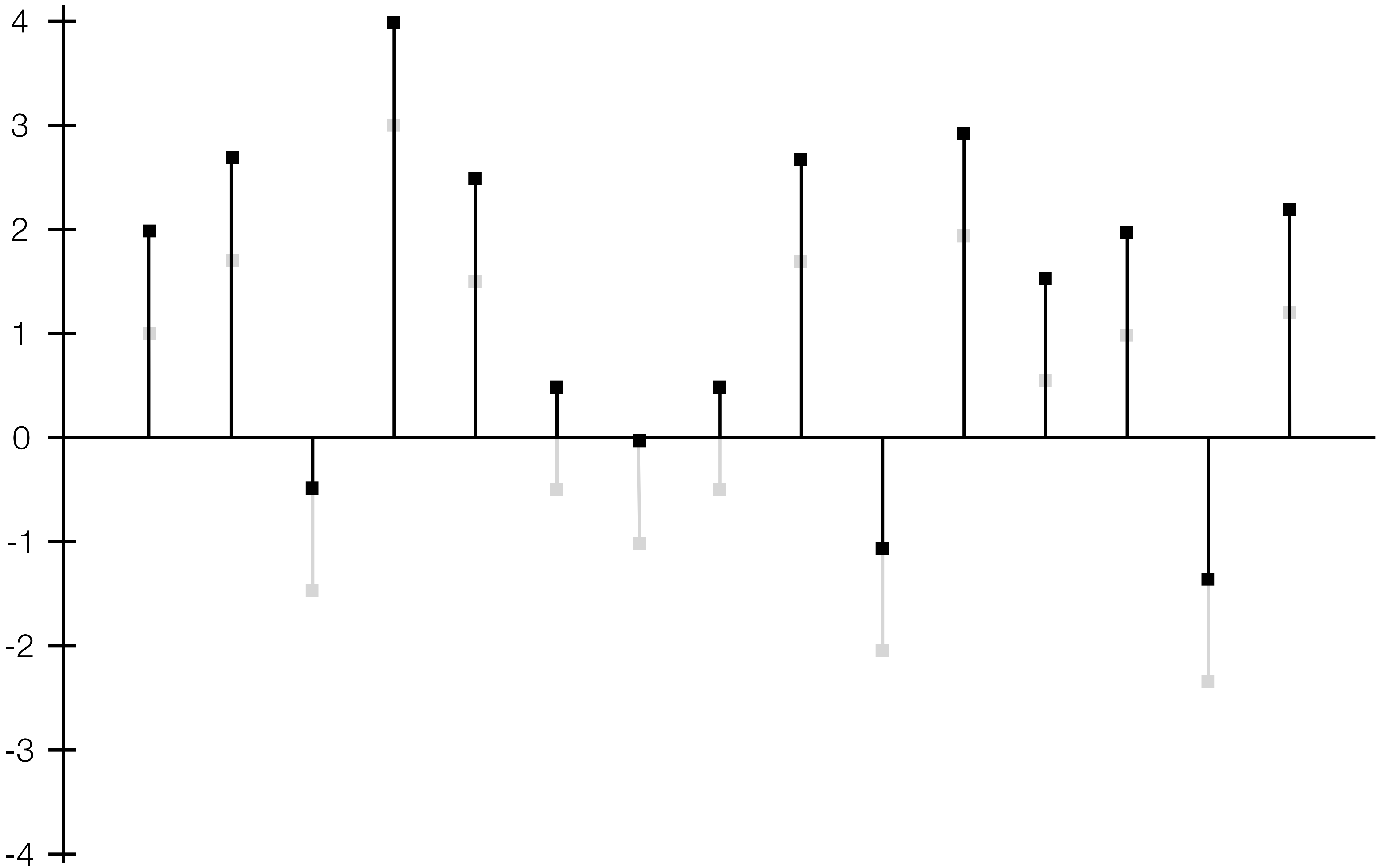
Machine Fusion is not Associative (or Commutative)

Ben Lippmeier, Amos Robinson
Shonan Meeting
Functional Stream Libraries and Fusion
2018/10/22

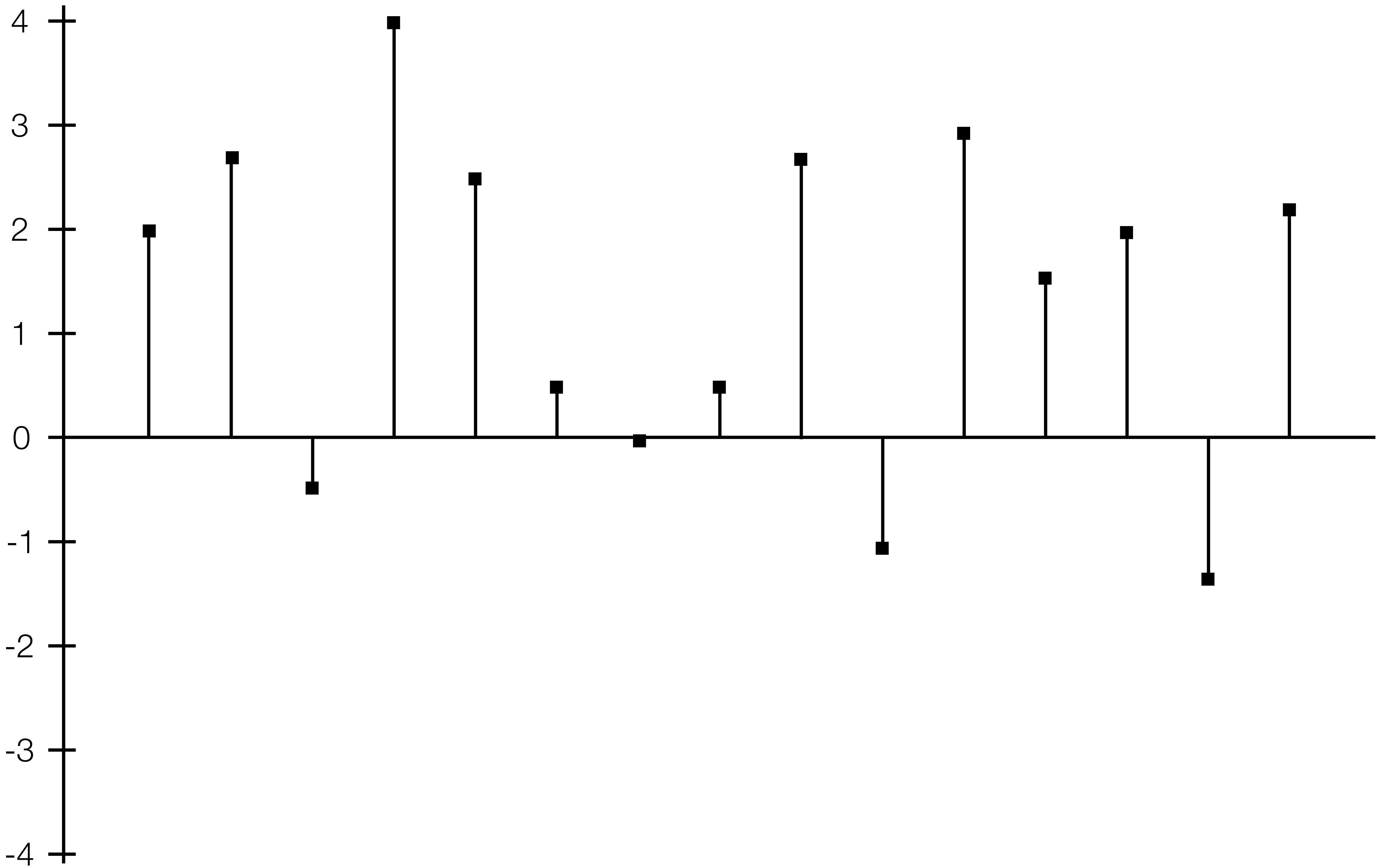
FilterMax



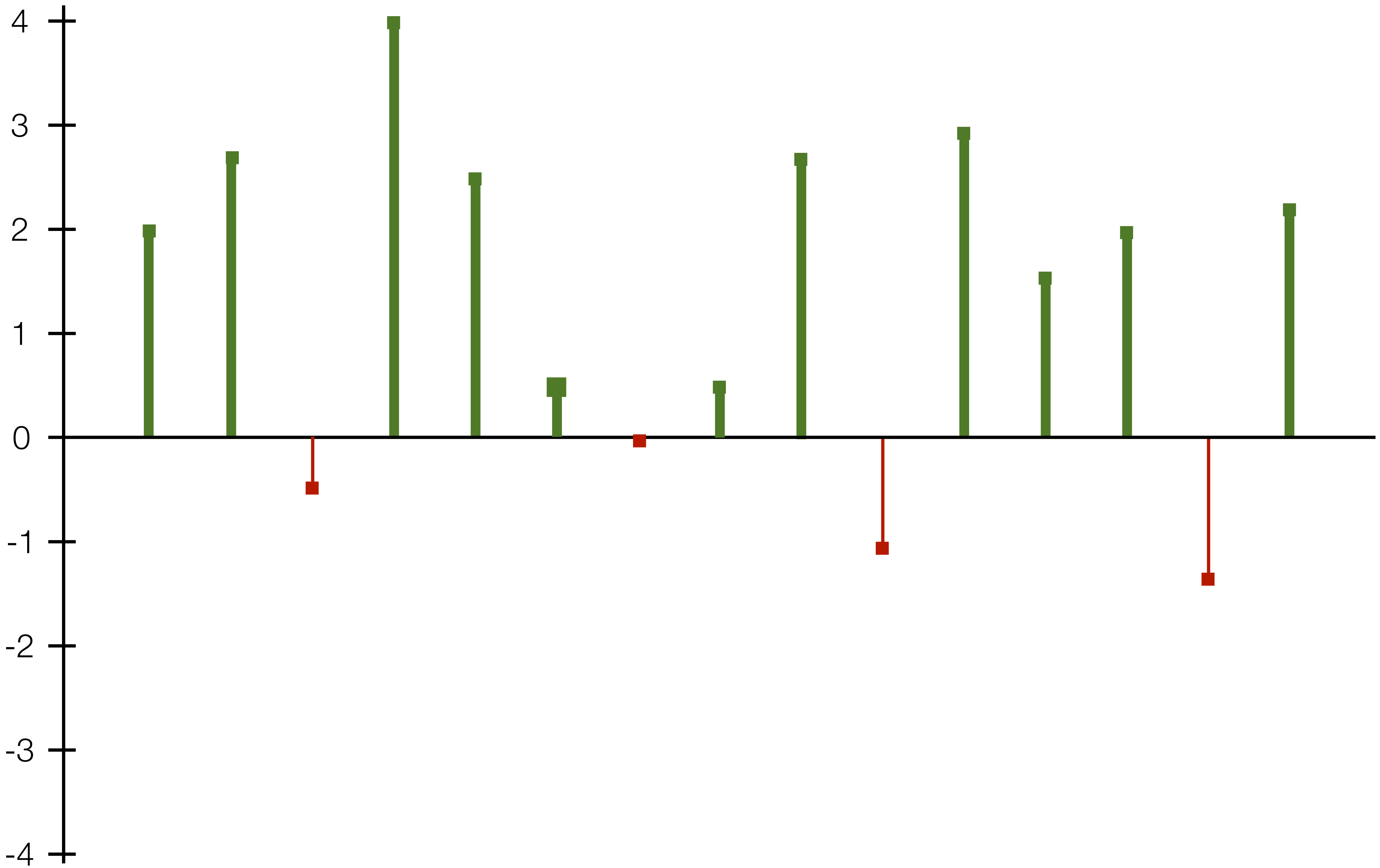
FilterMax



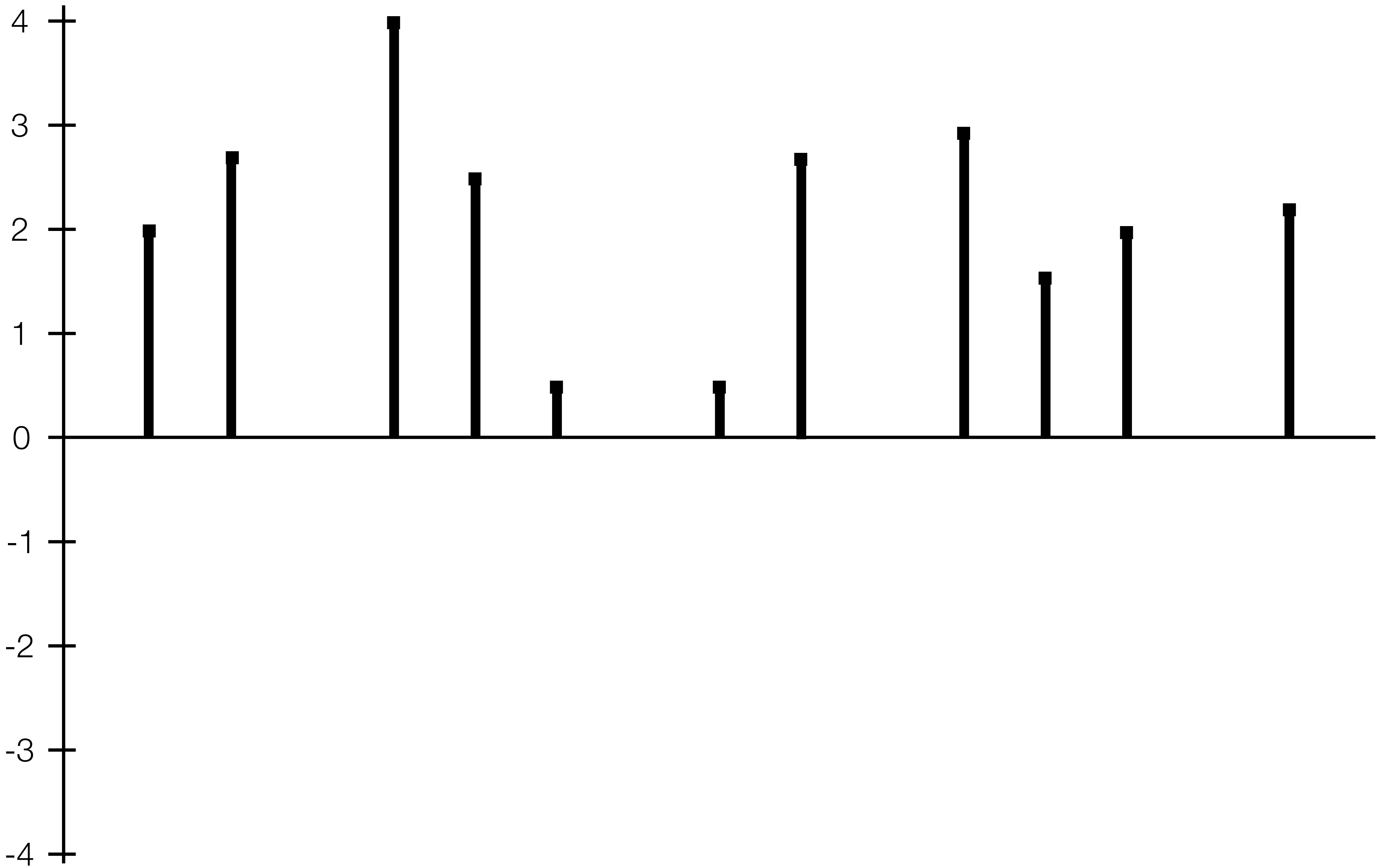
FilterMax



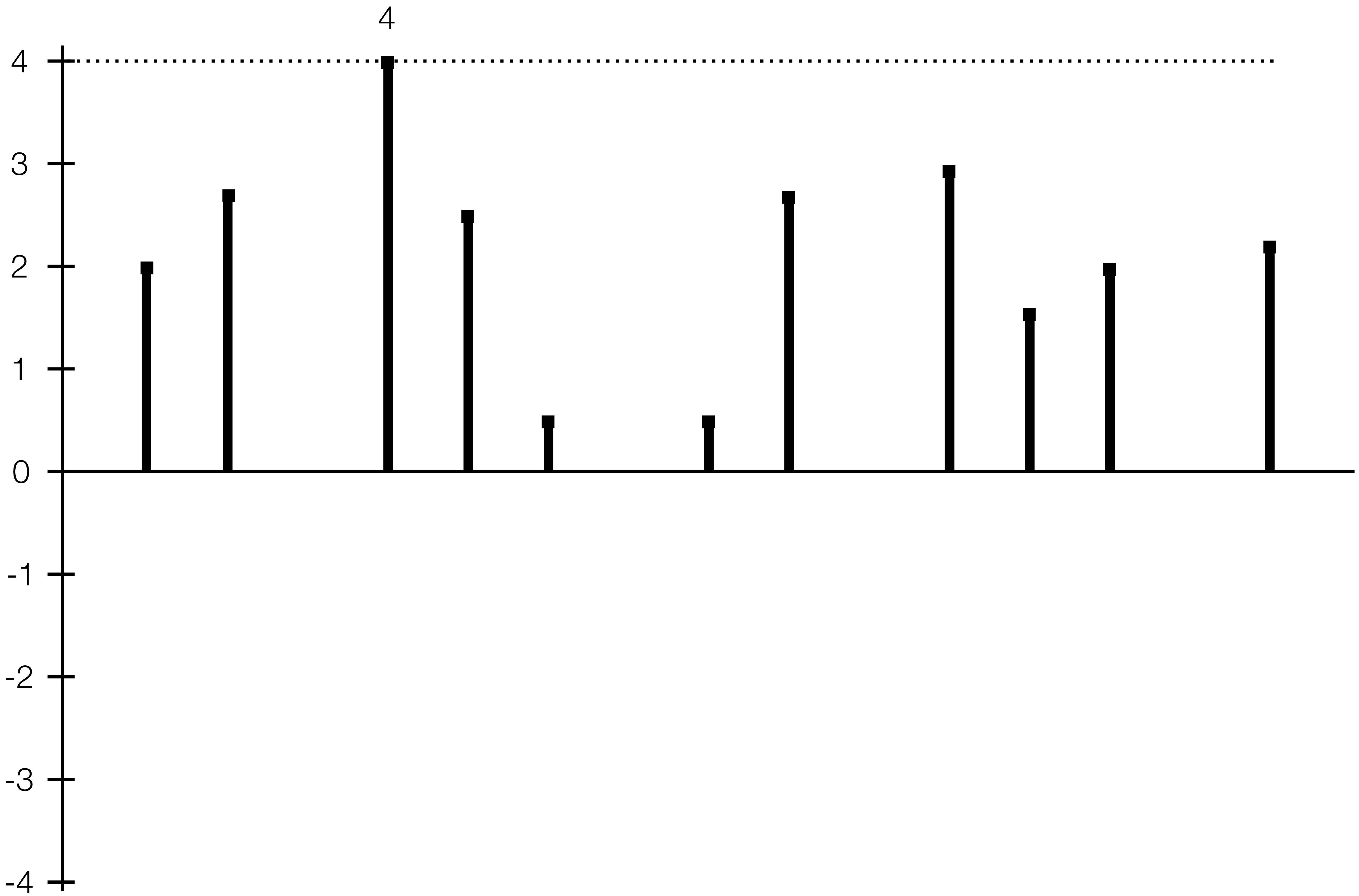
FilterMax



FilterMax



FilterMax



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let vec2 = map (+ 1) vec1
       vec3 = filter (> 0) vec2
       n    = fold max 0 vec3
   in  (vec3, n)
```

```haskell
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let vec2 = map (+ 1) vec1
       vec3 = filter (> 0) vec2
       n    = fold max 0 vec3
   in  (vec3, n)
```

```haskell
map f      = unstream . mapS f    . stream
filter p   = unstream . filterS p . stream
fold f z = foldS f z . stream
```

```haskell
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let vec2 = unstream (mapS (+ 1) (stream vec1))
       vec3 = unstream (filterS (> 0) (stream vec2))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

```haskell
map f    = unstream . mapS f    . stream
filter p = unstream . filterS p . stream
fold f z = foldS f z . stream
```

```haskell
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0)
                  (stream (unstream (mapS (+ 1)
                                 (stream vec1)))))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

```haskell
map f    = unstream . mapS f    . stream
filter p = unstream . filterS p . stream
fold f z = foldS f z . stream
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0)
                  (stream (unstream (mapS (+ 1)
                                (stream vec1)))))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0)
                  (stream (unstream (mapS (+ 1)
                                (stream vec1)))))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

```
RULE "stream/unstream"
     forall xs. stream (unstream xs) = xs
```

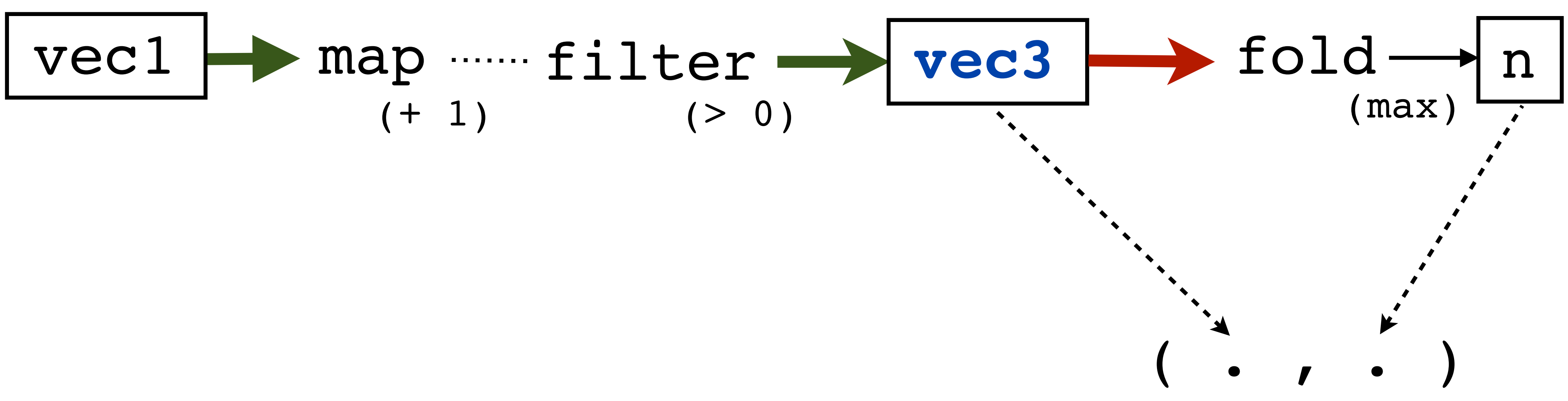
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0)
                 (stream (unstream (mapS (+ 1)
                                (stream vec1)))))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

```
RULE "stream/unstream"
     forall xs. stream (unstream xs) = xs
```

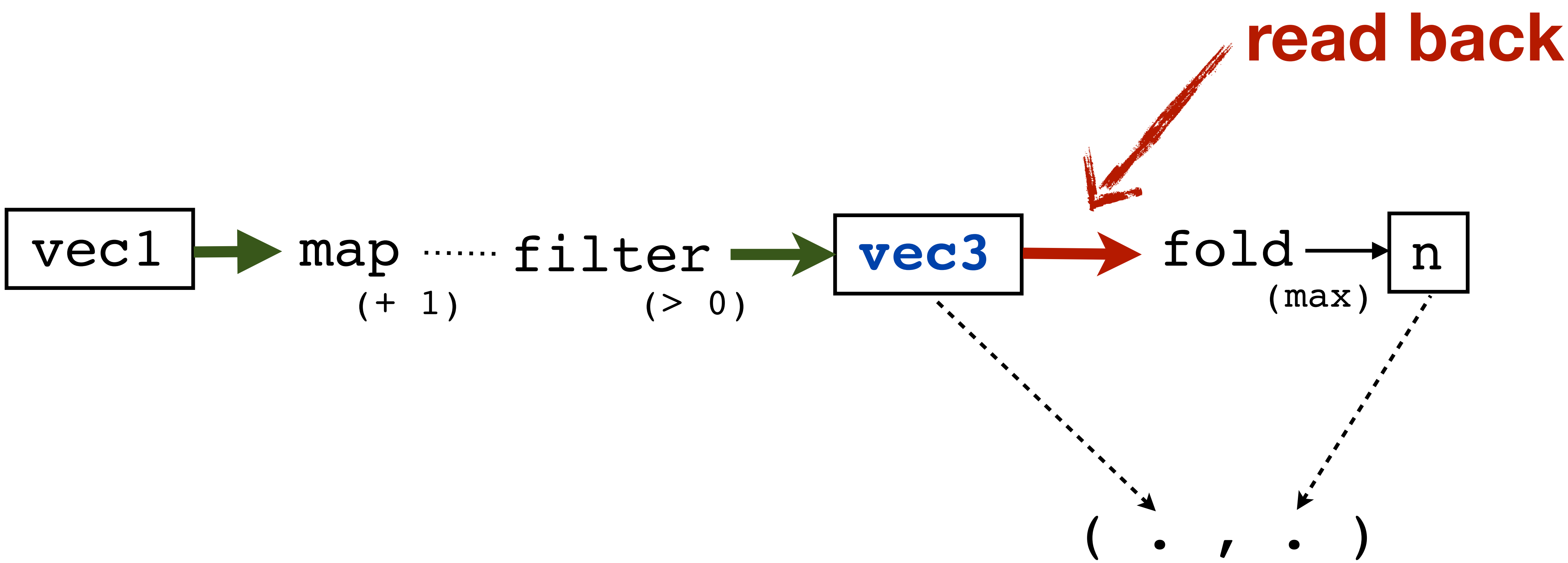
```haskell
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0) (mapS (+ 1)
                                  (stream vec1)))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

```haskell
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0) (mapS (+ 1)
                                  (stream vec1)))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```

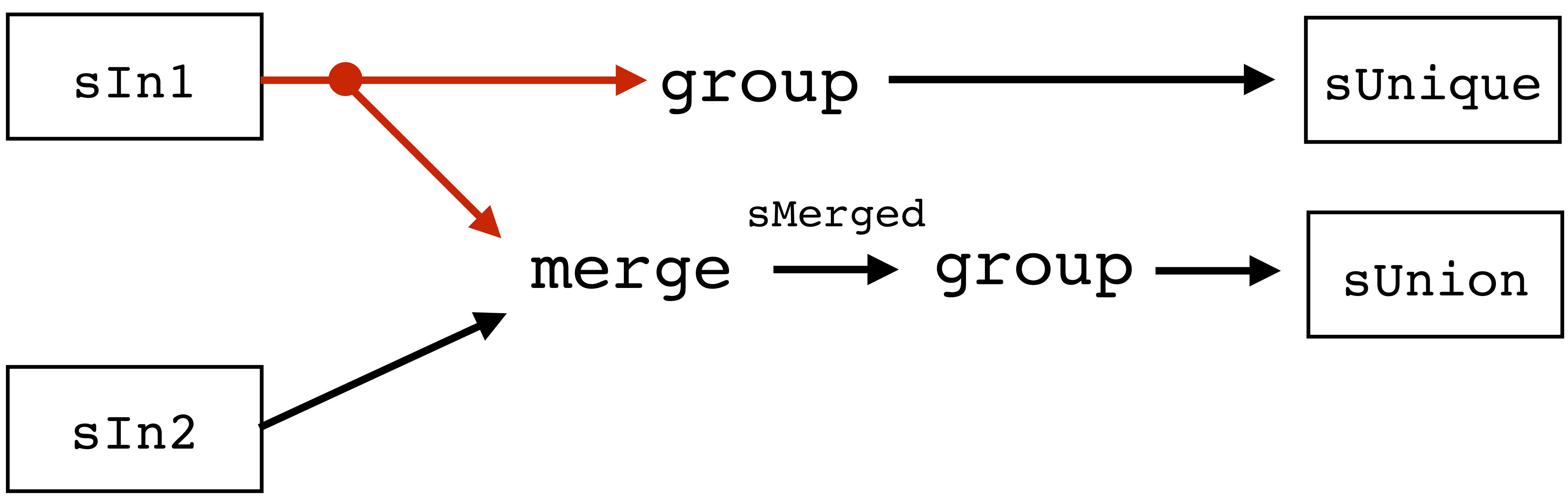
```haskell
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0) (mapS (+ 1)
                                   (stream vec1)))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```



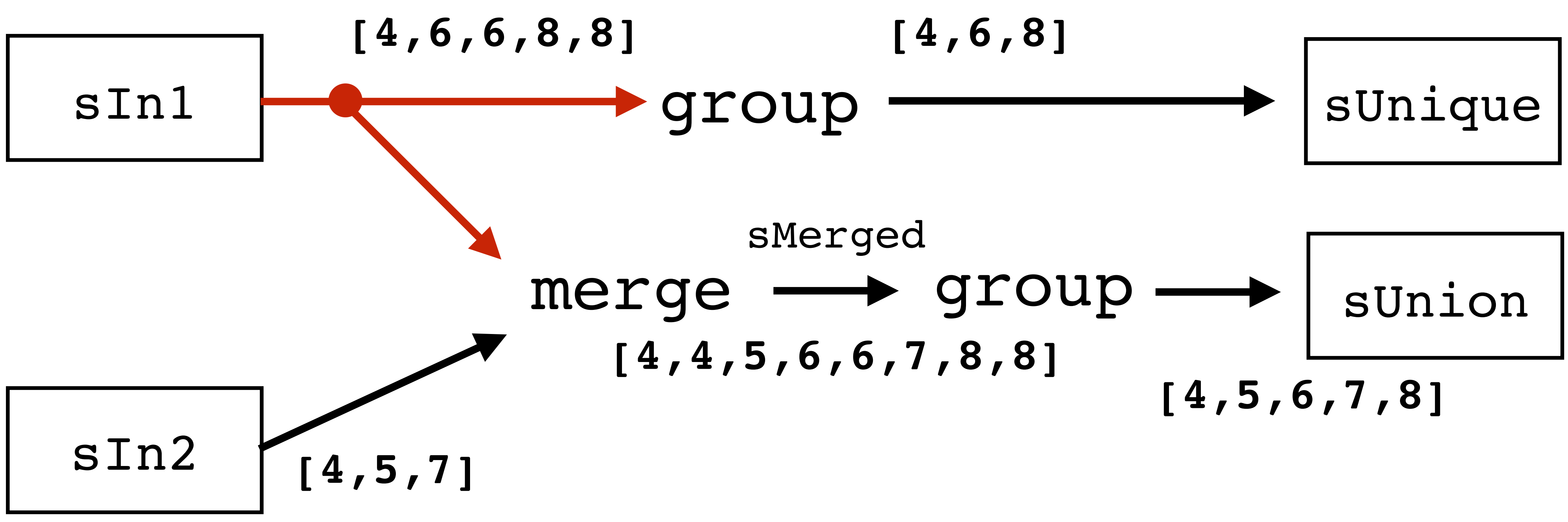
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = let
       vec3 = unstream (filterS (> 0) (mapS (+ 1)
                                  (stream vec1)))
       n    = foldS max 0 (stream vec3)
   in  (vec3, n)
```



```
uniquesUnion ::  Vector Nat -> Vector Nat
             -> (Vector Nat, Vector Nat)
uniquesUnion sIn1 sIn2
 = let sUnique = group sIn1
       sMerged = merge sIn1 sIn2
       sUnion  = group sMerged
   in  (sUnique, sUnion)
```



```
uniquesUnion ::  Vector Nat -> Vector Nat
             -> (Vector Nat, Vector Nat)
uniquesUnion sIn1 sIn2
 = let sUnique = group sIn1
       sMerged = merge sIn1 sIn2
       sUnion  = group sMerged
   in  (sUnique, sUnion)
```



```
f :: Stream (Int, Int) -> (Int, Int)
f s = let a1 = sum  (map fst s)
          a2 = prod (map snd s)
      in  (a1, a2)
```

```haskell
f :: Stream (Int, Int) -> (Int, Int)
f s = let a1 = sum  (map fst s)
          a2 = prod (map snd s)
      in  (a1, a2)
```

- Cannot implement lazy unzip with sequential execution semantics in a space efficient way.
  - Noticed by John Hughes in his PhD thesis (1983)
  - Told to me by Peter Gammie

- Pattern arises frequently in vectorised code from DPH. We often combine a single segment descriptor or selector vector with many data vectors.

Problem

Short-cut stream fusion cannot fuse a producer into multiple consumers

Problem’

Pull stream model does not support space efficient `unzip`

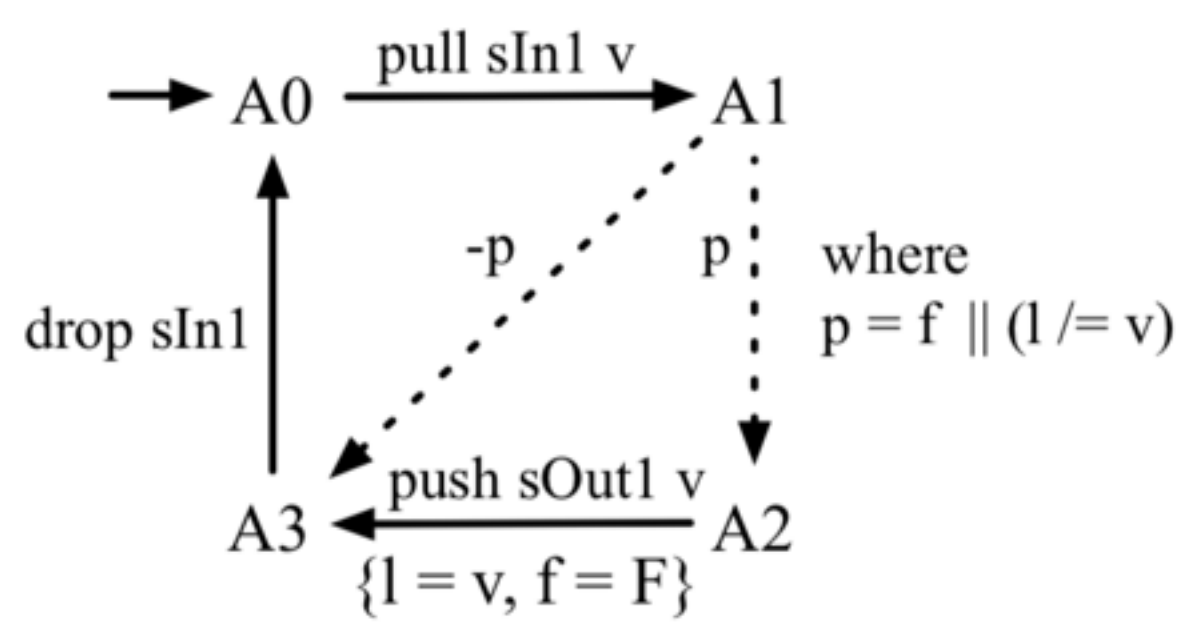
Push stream model does not support space efficient `zip`

(a pleasing* duality)

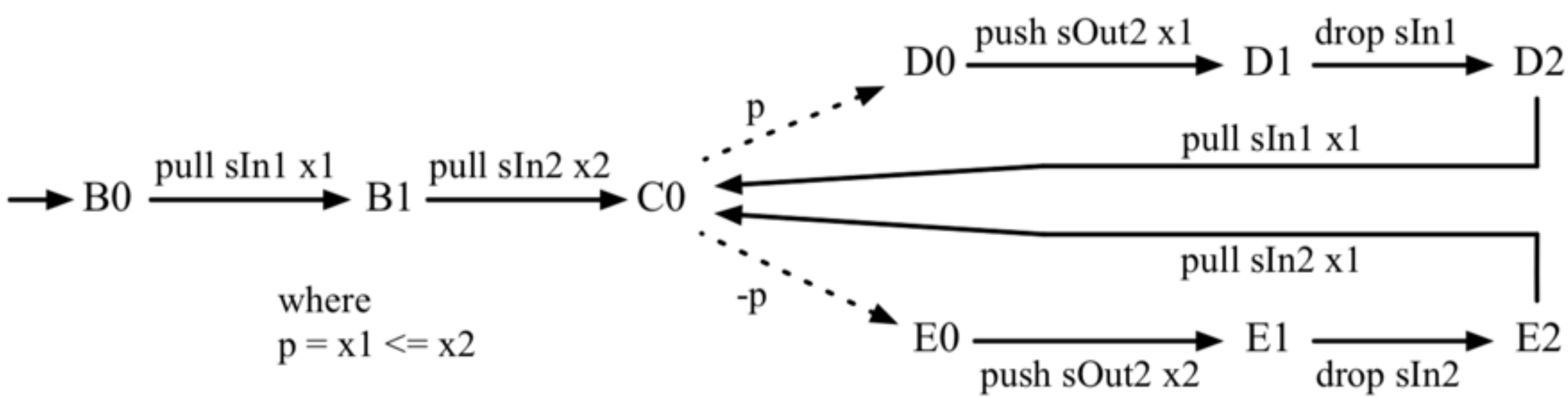
***only pleasing in theory, not in practice.**

We need both Pull and Push
(or maybe neither)

```
group
  = λ (sIn1: Stream Nat) (sOut1: Stream Nat).
    ν (f: Bool) (l: Nat) (v: Nat) (A0..A3: Label).
    process
     { ins:    { sIn1  }
     , outs:   { sOut1 }
     , heap:   { f = T, l = 0, v = 0 }
     , label:  A0
     , instrs: { A0 = pull sIn1 v           A1 []
               , A1 = case (f || (l /= v)) A2 []  A3 []
               , A2 = push sOut1 v          A3 [ l = v, f = F ]
               , A3 = drop sIn1             A0 [] } }
```



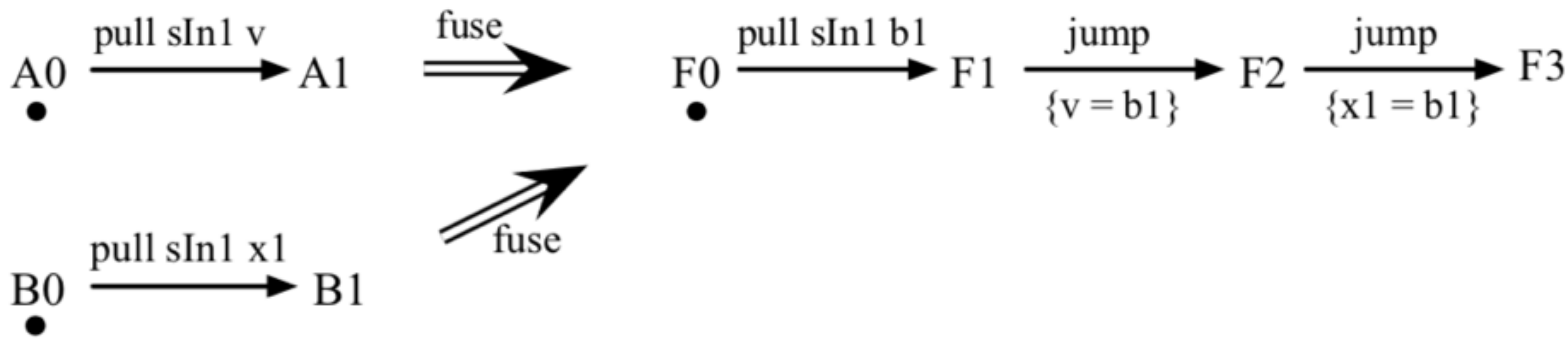
```
merge
  = λ (sIn1: Stream Nat) (sIn2: Stream Nat) (sOut2: Stream Nat).
    ν (x1: Nat) (x2: Nat) (B0..E2: Label).
    process
     { ins:    { sIn1, sIn2 }
     , outs:   { sOut2 }
     , heap:   { x1 = 0, x2 = 0 }
     , label:  B0
     , instrs: { B0 = pull sIn1  x1   B1 []                 , B1 = pull sIn2  x2   C0 []
               , C0 = case (x1 < x2)  D0 []  E0 []          , D0 = push sOut2 x1   D1 []
               , D1 = drop sIn1       D2 []                 , D2 = pull sIn1  x1   C0 []
               , E0 = push sOut2 x2   E1 []                 , E1 = drop sIn2       E2 []
               , E2 = pull sIn2 x2    C0 [] } }
```



group merge

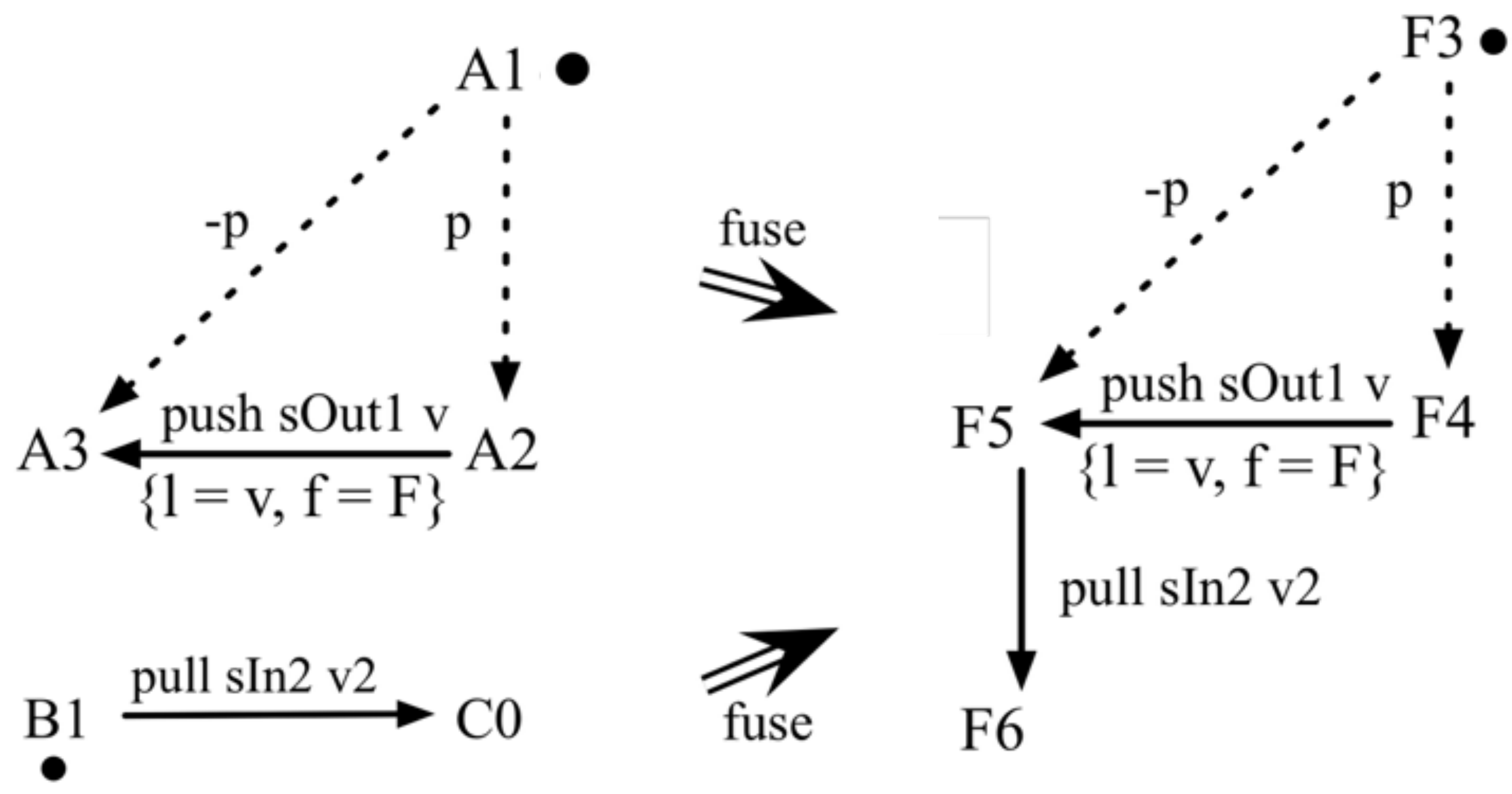
shared

```
process
{ ins:    { sIn1,  sIn2 }
, outs:   { sOut1, sOut2 }
, heap:   { f = T, l = 0, v = 0, x1 = 0, x2 = 0, b1 = 0 }
, label:  F0
, instrs:
  { F0  = pull sIn1 b1          F1  [ ]                  F0  = ((A0,{sIn1 = none}),    (B0, {sIn1 = none,    sIn2 = none}))
  , F1  = jump                  F2  [ v = b1 ]           F1  = ((A0,{sIn1 = pending}), (B0, {sIn1 = pending, sIn2 = none}))
  , F2  = jump                  F3  [ x1 = b1 ]          F2  = ((A1,{sIn1 = have}),    (B0, {sIn1 = pending, sIn2 = none}))
  , F3  = case (f || (l /= v))  F4  [ ]      F5  [ ]     F3  = ((A1,{sIn1 = have}),    (B1, {sIn1 = have,    sIn2 = none}))
  , F4  = push sOut1 v          F5  [ l = v, f = F ]     F4  = ((A2,{sIn1 = have}),    (B1, {sIn1 = have,    sIn2 = none}))
  , F5  = jump                  F6  [ ]                  F5  = ((A3,{sIn1 = have}),    (B1, {sIn1 = have,    sIn2 = none}))
  , F6  = pull sIn2 x2          F7  [ ]                  F6  = ((A0,{sIn1 = none}),    (B1, {sIn1 = have,    sIn2 = none}))

  , F7  = case (x1 < x2)        F8  [ ]      F16 [ ]     F7  = ((A0,{sIn1 = none}),    (C0, {sIn1 = have,    sIn2 = have}))

  , F8  = push sOut2 x1         F9  [ ]                  F8  = ((A0,{sIn1 = none}),    (D0, {sIn1 = have,    sIn2 = have}))
  , F9  = drop sIn1             F10 [ ]                  F9  = ((A0,{sIn1 = none}),    (D1, {sIn1 = none,    sIn2 = have}))
  , F10 = pull sIn1 b1          F11 [ ]                  F10 = ((A0,{sIn1 = none}),    (D2, {sIn1 = none,    sIn2 = have}))
  , F11 = jump                  F12 [ v = b1 ]           F11 = ((A0,{sIn1 = pending}), (D2, {sIn1 = pending, sIn2 = have}))
  , F12 = jump                  F13 [ x1 = b1 ]          F12 = ((A1,{sIn1 = have}),    (D2, {sIn1 = pending, sIn2 = have}))
  , F13 = case (f || (l /= v))  F14 [ ]      F15 [ ]     F13 = ((A1,{sIn1 = have}),    (C0, {sIn1 = have,    sIn2 = have}))
  , F14 = push sOut1 v          F15 [ l = v, f = F ]     F14 = ((A2,{sIn1 = have}),    (C0, {sIn1 = have,    sIn2 = have}))
  , F15 = jump                  F7  [ ]                  F15 = ((A3,{sIn1 = have}),    (C0, {sIn1 = have,    sIn2 = have}))

  , F16 = push sOut2 x2         F17 [ ]                  F16 = ((A0,{sIn1 = none}),    (E0, {sIn1 = have,    sIn2 = have}))
  , F17 = drop sIn2             F18 [ ]                  F17 = ((A0,{sIn1 = none}),    (E1, {sIn1 = have,    sIn2 = have}))
  , F18 = pull sIn2             F7  [ ]                  F18 = ((A0,{sIn1 = none}),    (E2, {sIn1 = have,    sIn2 = none}))
} }
```



where

F0 = ((A0, {sIn1 = none}), (B0, {sIn1 = none, sIn2 = none}))
F1 = ((A0, {sIn1 = pending}), (B0, {sIn1 = pending, sIn2 = none}))
F2 = ((A1, {sIn1 = have}), (B0, {sIn1 = pending, sIn2 = none}))
F3 = ((A1, {sIn1 = have}), (B1, {sIn1 = have, sIn2 = none}))



where

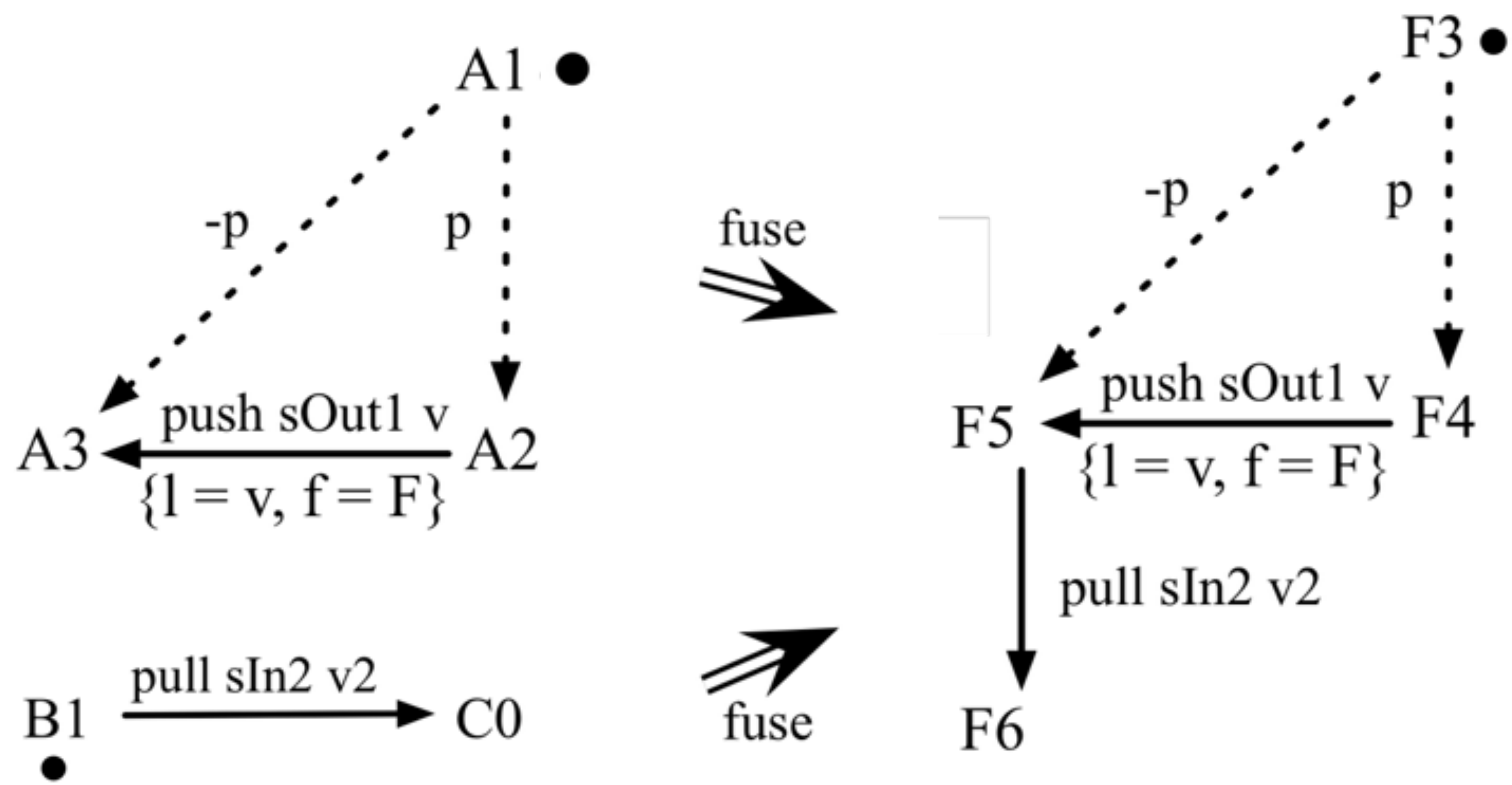
p = f || (l /= v)

F3 = ((A1,{sIn1 = have}), (B1,{sIn1 = have, sOut2 = none}))

F4 = ((A2,{sIn1 = have}), (B1,{sIn1 = have, sOut2 = none}))

F5 = ((A3,{sIn1 = have}), (B1,{sIn1 = have, sOut2 = none}))

F6 = ((A3,{sIn1 = have}), (C0,{sIn1 = have, sOut2 = have}))



where

$p = f \mid\mid (l \mathrel{/=} v)$

$F3 = ((A1, \{sIn1 = have\}),\ (B1, \{sIn1 = have, sOut2 = none\}))$

$F4 = ((A2, \{sIn1 = have\}),\ (B1, \{sIn1 = have, sOut2 = none\}))$

$F5 = ((A3, \{sIn1 = have\}),\ (B1, \{sIn1 = have, sOut2 = none\}))$

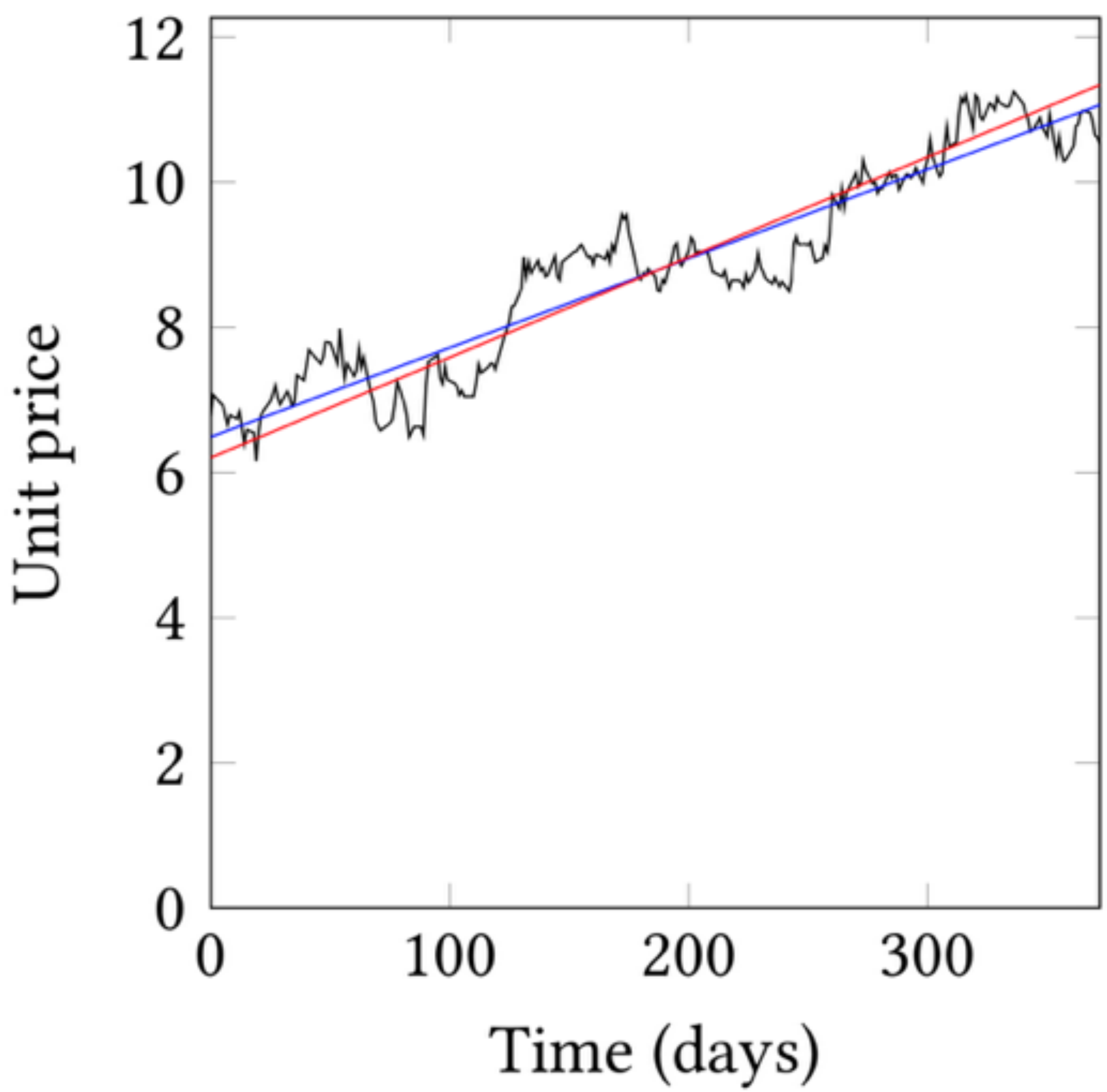
$F6 = ((A3, \{sIn1 = have\}),\ (C0, \{sIn1 = have, sOut2 = have\}))$

- Could also do the pull first…

$$
\begin{array}{l}
\mathit{tryStep} \;:\; (\mathit{Channel} \mapsto \mathit{ChannelType2}) \to \mathit{LabelF} \to \mathit{Instruction} \to \mathit{LabelF} \to \mathit{Maybe\ Instruction} \\
\mathit{tryStep}\ \mathit{cs}\ (l_p, s_p)\ i_p\ (l_q, s_q) \;=\; \texttt{match}\ i_p\ \texttt{with}
\end{array}
$$

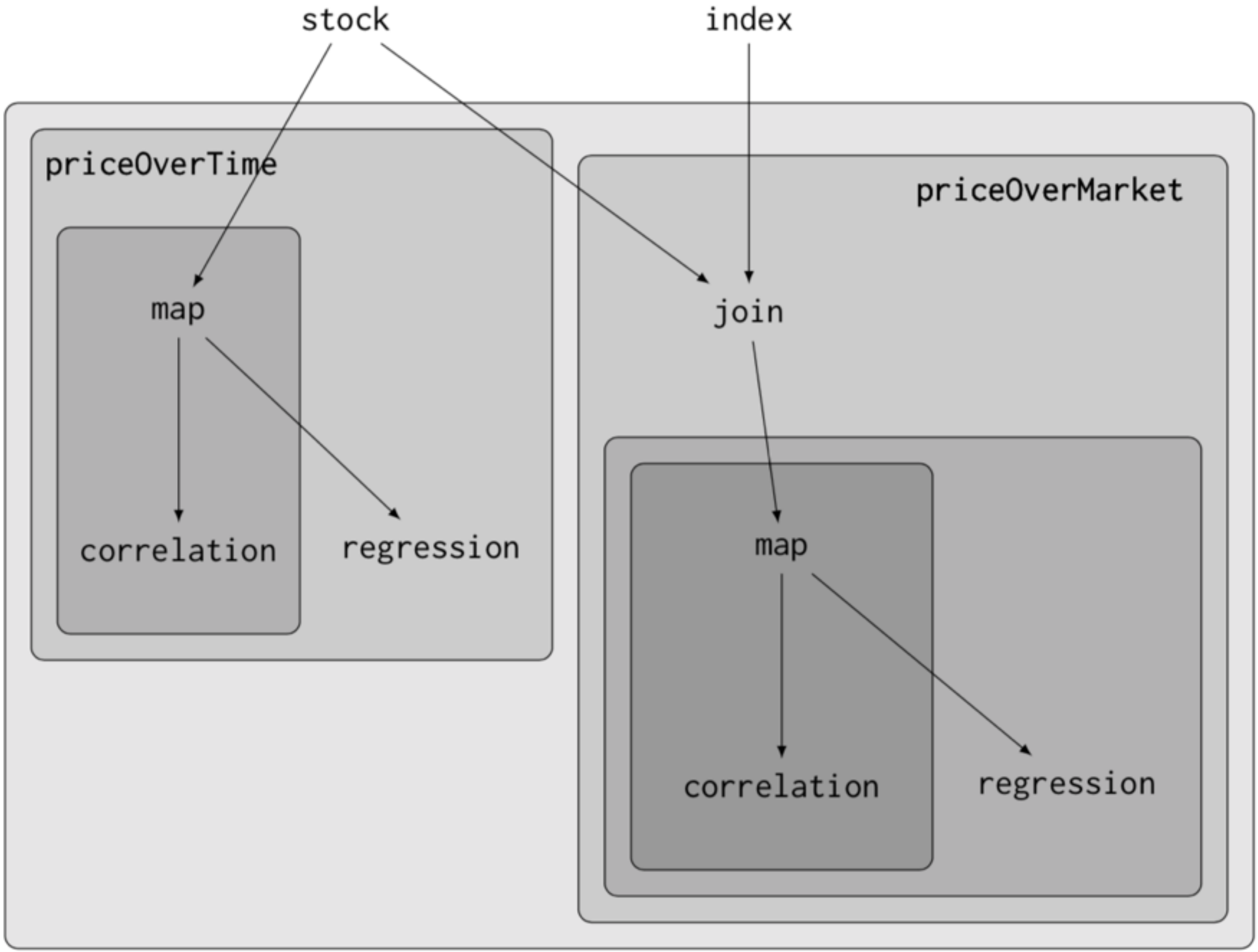
$$
\begin{array}{ll}
\texttt{jump}\ (l', u') & \text{(LocalJump)} \\
\quad \to \texttt{Just}\ (\texttt{jump}\ ((l',\ s_p),\ (l_q,\ s_q),\ u')) & \\[1ex]
\texttt{case}\ e\ (l'_t, u'_t)\ (l'_f, u'_f) & \text{(LocalCase)} \\
\quad \to \texttt{Just}\ (\texttt{case}\ e\ ((l'_t,\ s_p),\ (l_q,\ s_q),\ u'_t)\ ((l'_f,\ s_p),\ (l_q,\ s_q),\ u'_f)) & \\[1ex]
\texttt{push}\ c\ e\ (l', u') & \\
\quad \mid \mathit{cs}[c] = \texttt{out1} & \text{(LocalPush)} \\
\quad \to \texttt{Just}\ (\texttt{push}\ c\ e\ ((l',\ s_p),\ (l_q,\ s_q),\ u')) & \\
\quad \mid \mathit{cs}[c] = \texttt{in1out1} \wedge s_q[c] = \texttt{none}_F & \text{(SharedPush)} \\
\quad \to \texttt{Just}\ (\texttt{push}\ c\ e\ ((l',\ s_p),\ (l_q,\ s_q[c \mapsto \texttt{pending}_F]),\ u'[\texttt{chan}\ c \mapsto e])) & \\[1ex]
\texttt{pull}\ c\ x\ (l'_o, u'_o)\ (l'_c, u'_c) & \\
\quad \mid \mathit{cs}[c] = \texttt{in1} & \text{(LocalPull)} \\
\quad \to \texttt{Just}\ (\texttt{pull}\ c\ x\ ((l'_o,\ s_p),\ (l_q,\ s_q),\ u'_o)\ ((l'_c,\ s_p),\ (l_q,\ s_q),\ u'_c)) & \\[1ex]
\quad \mid (\mathit{cs}[c] = \texttt{in2} \vee \mathit{cs}[c] = \texttt{in1out1}) \wedge s_p[c] = \texttt{pending}_F & \text{(SharedPullPending)} \\
\quad \to \texttt{Just}\ (\texttt{jump}\ ((l'_o,\ s_p[c \mapsto \texttt{have}_F]),\ (l_q,\ s_q),\ u'_o[x \mapsto \texttt{chan}\ c])) & \\[1ex]
\quad \mid (\mathit{cs}[c] = \texttt{in2} \vee \mathit{cs}[c] = \texttt{in1out1}) \wedge s_p[c] = \texttt{closed}_F & \text{(SharedPullClosed)} \\
\quad \to \texttt{Just}\ (\texttt{jump}\ ((l'_c,\ s_p),\ (l_q,\ s_q),\ u'_c)) & \\[1ex]
\quad \mid \mathit{cs}[c] = \texttt{in2} \wedge s_p[c] = \texttt{none}_F \wedge s_q[c] = \texttt{none}_F & \text{(SharedPullInject)} \\
\quad \to \texttt{Just}\ (\texttt{pull}\ c\ (\texttt{chan}\ c) & \\
\qquad\qquad ((l_p,\ s_p[c \mapsto \texttt{pending}_F]),\ (l_q,\ s_q[c \mapsto \texttt{pending}_F]),\ []) & \\
\qquad\qquad ((l_p,\ s_p[c \mapsto \texttt{closed}_F]),\ (l_q,\ s_q[c \mapsto \texttt{closed}_F]),\ [])) &
\end{array}
$$

Stock Price Graph

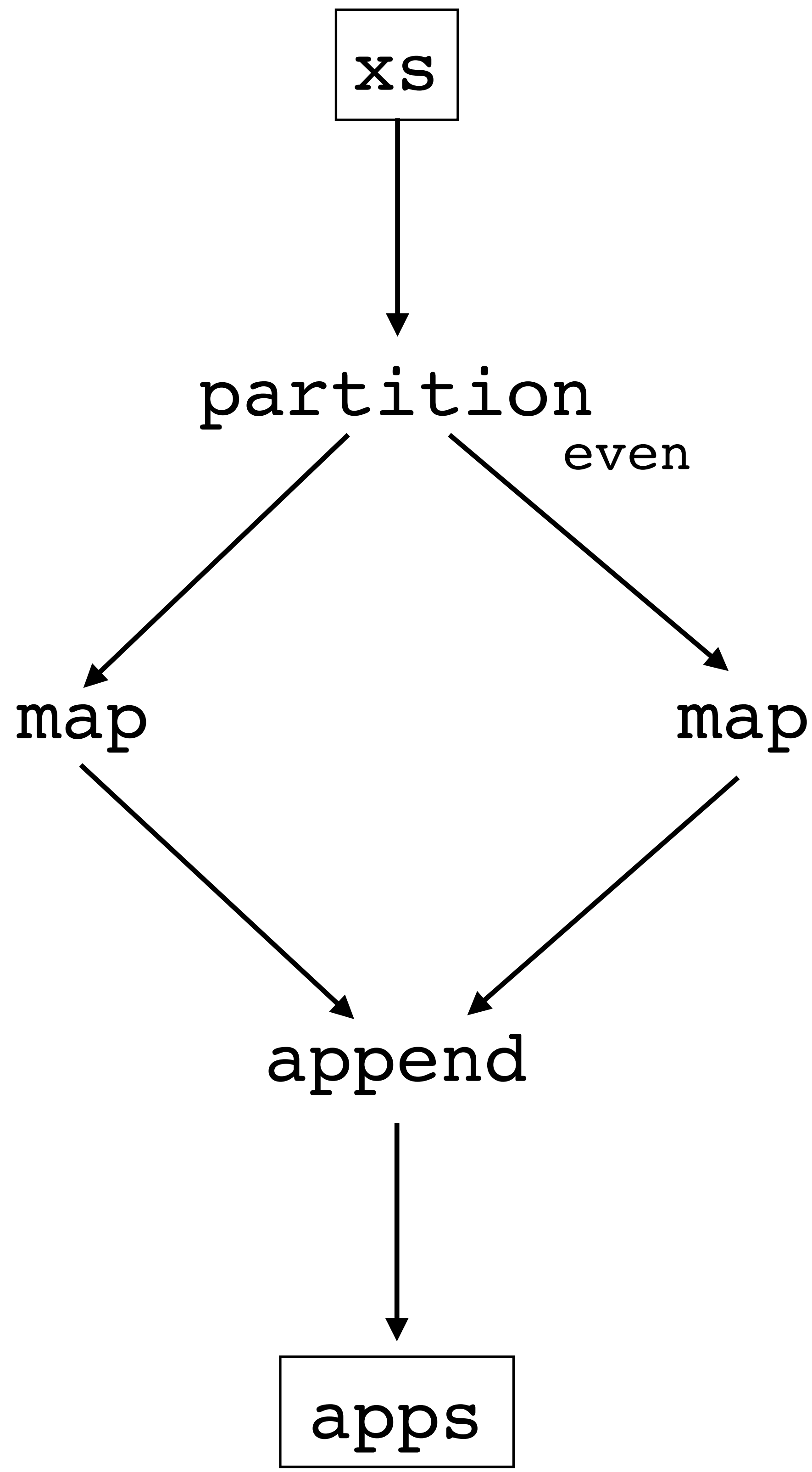



```haskell
data Record = Record
  { time  :: Time
  , price :: Double }
```

```haskell
priceAnalyses :: [Record] → [Record] → ((Line, Double), (Line, Double))
priceAnalyses stock index =
  let pot = priceOverTime   stock
      pom = priceOverMarket stock index
  in (pot, pom)

priceOverTime :: [Record] → (Line, Double)
priceOverTime stock =
  let timeprices = map (λr → (daysSinceEpoch (time r), price r)) stock
  in (regression timeprices, correlation timeprices)


priceOverMarket :: [Record] → [Record] → (Line, Double)
priceOverMarket stock index =
  let joined = join (λs i   → time s `compare` time i) stock index
      prices = map  (λ(s,i) → (price s, price i))      joined
  in (regression prices, correlation prices)
```

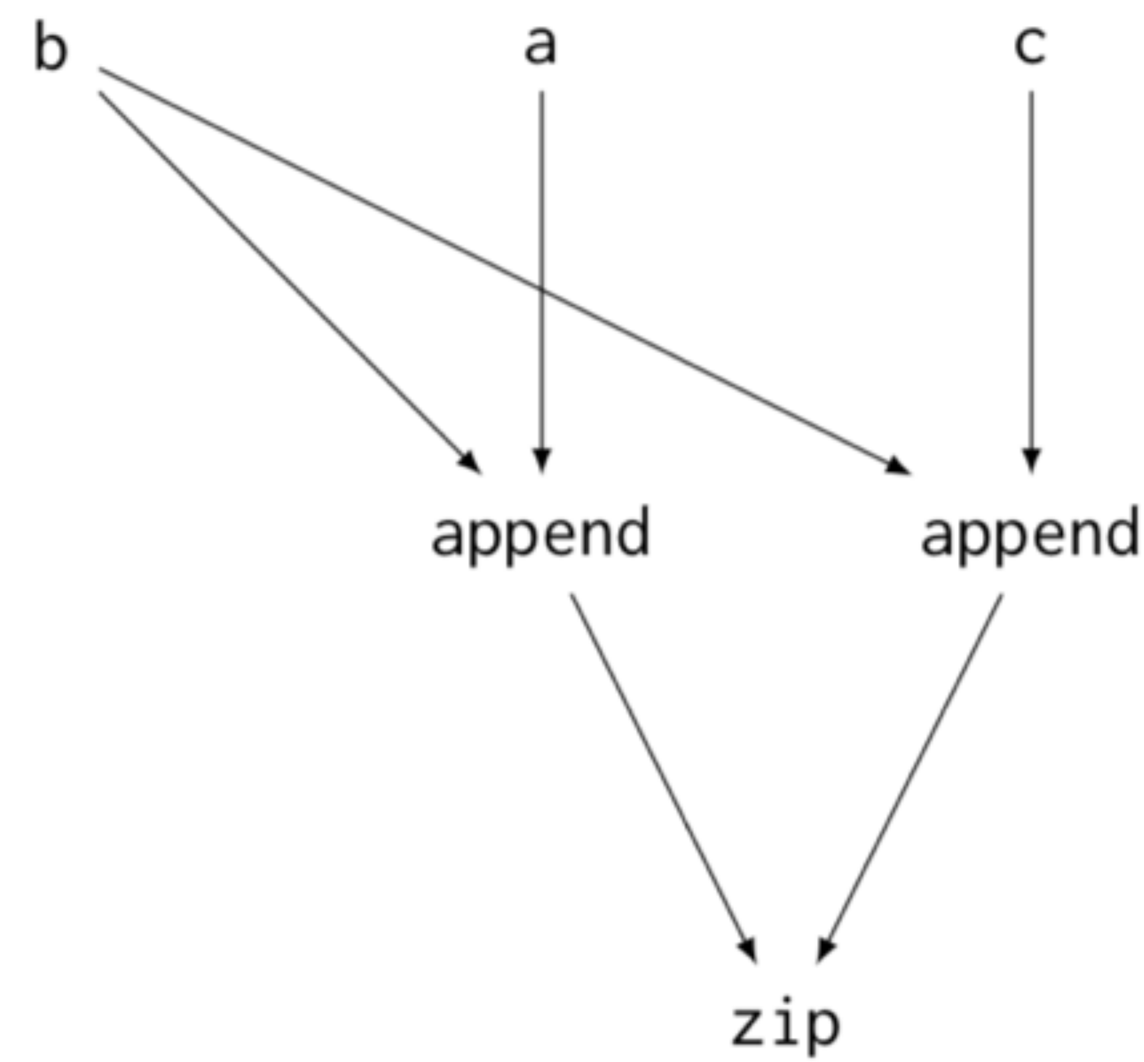

stock
index
priceOverTime
map
correlation
regression
priceOverMarket
join
map
correlation
regression

```
partitionAppendFailure :: Vector Int → IO (Vector Int)
partitionAppendFailure xs = do
  (apps,()) ← vectorSize xs $ λsnkApps →
    $$(fuse $ do
        x0           ← source    [|sourceOfVector xs|]
        (evens,odds) ← partition [|λi → even i      |] x0
        evens'       ← map       [|λi → i `div` 2   |] evens
        odds'        ← map       [|λi → i ∗ 2       |] odds
        apps         ← append evens' odds'
        sink apps                [|snkApps          |])
  return apps
```


xs
partition
even
map
map
append
apps

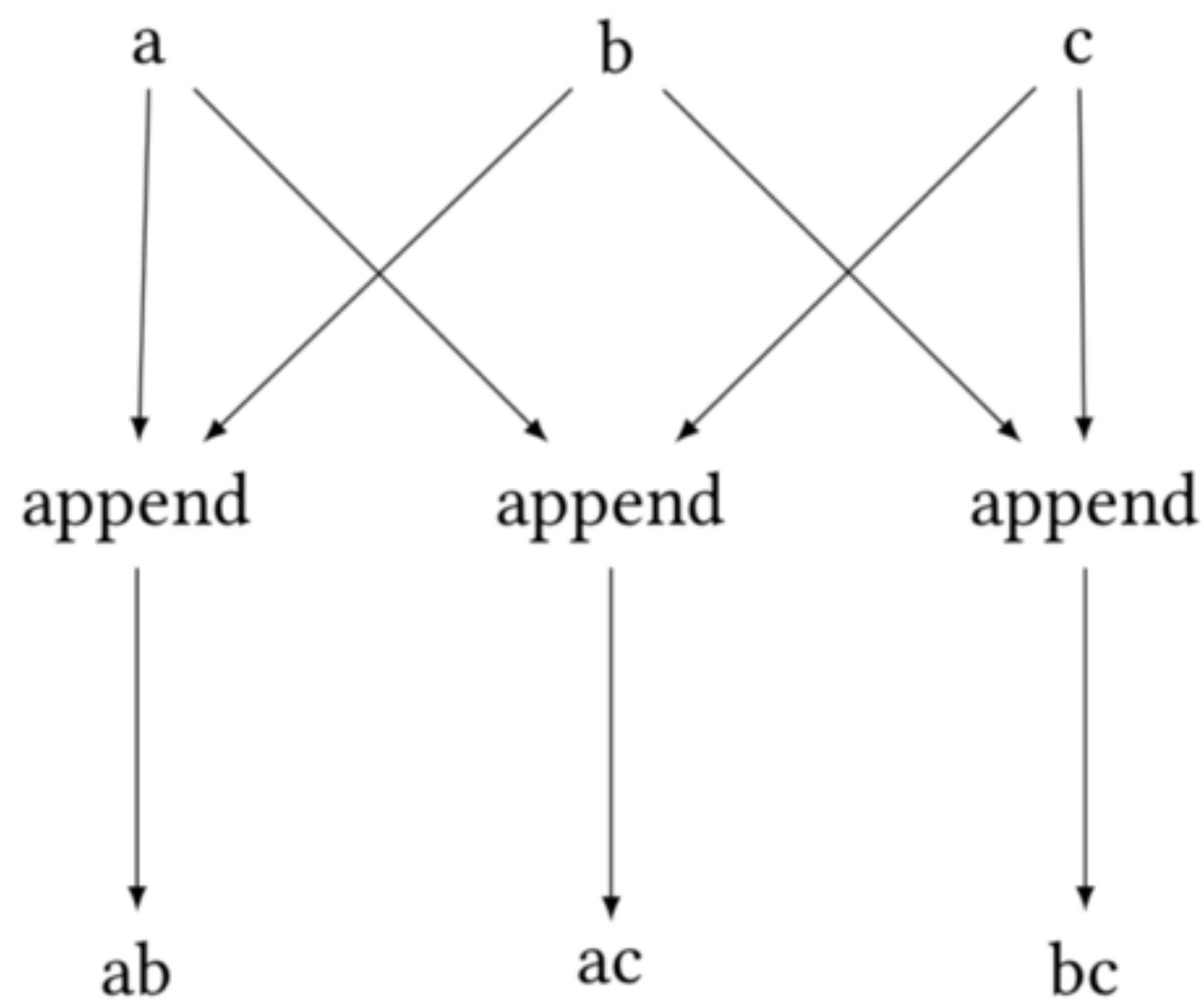
```
bench/Bench/PartitionAppend/Folderol.hs:18:8: warning:
  Maximum process count exceeded: there are 2 processes after fusion.
  Inserting unbounded communication channels between remaining processes.

  Input process network (4 processes):
    ()    ->-{sourceOfVector xs}--> C0
    C0    ->-----(partition)------> C1 C2
    C1    ->--------(map)---------> C3
    C2    ->--------(map)---------> C4
    C3 C4 ->------(append)--------> C5
    C5    ->------{snkApps}-------> ()

  Partially fused process network (2 processes):
    ()    ->-{sourceOfVector xs}--> C0
    C0    ->-----(partition)------> C1 C2
    C1 C2 ->-(map / map / append)-> C5
    C5    ->------{snkApps}-------> ()
```

```haskell
append2zip :: [a] → [a] → [a] → [(a,a)]
append2zip a b c =
  let ba = b ++ a
      bc = b ++ c
      z  = zip ba bc
  in  z
```



[(b0,b0), (b1,b1) … (bn,bn), (a0,c0), (a1,c1) …]

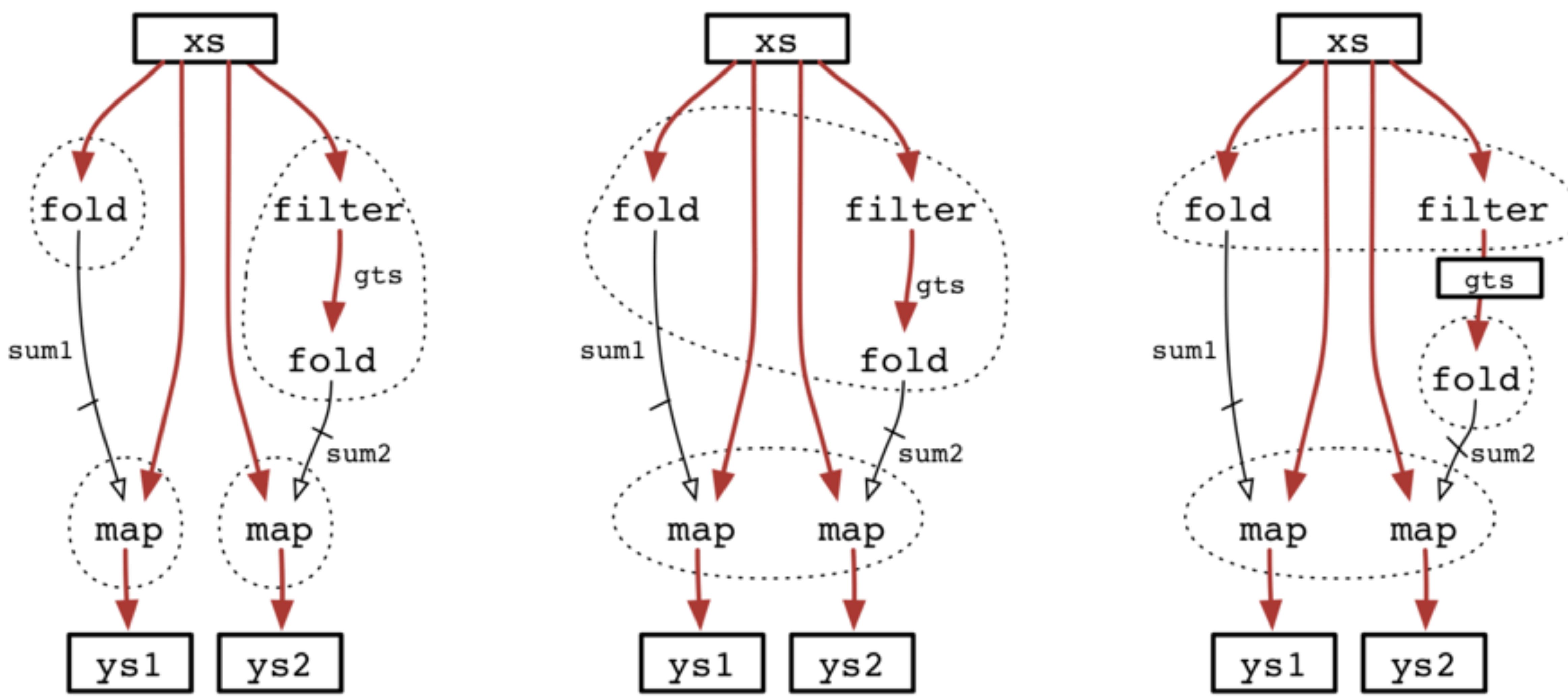
```haskell
append3 :: [a] → [a] → [a] → ([a],[a],[a])
append3 a b c =
  let ab = a ++ b
      ac = a ++ c
      bc = b ++ c
  in  (ab, ac, bc)
```



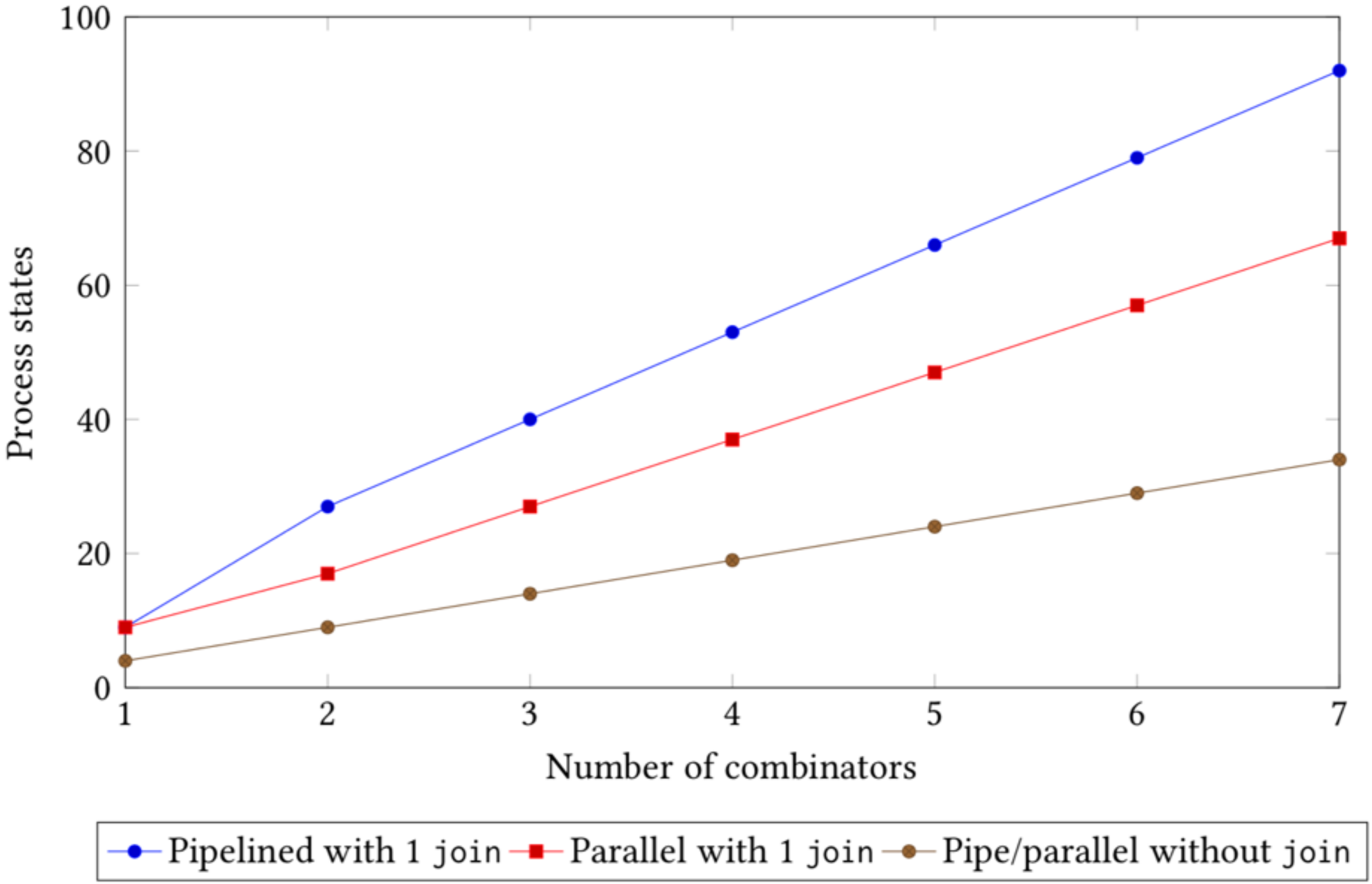
Fusion is neither Associative or Commutative.

- The access pattern of the result process depends on the order in which the source processes are fused.
- Not all orders produce a result process with an access pattern that can be fused with successive processes.
- We don’t have a way to decide on the fusion order other than heuristics and trying all the orders.
- Will likely cause combinatorial explosion in pathological cases.
- How do we prune the search space, session types?

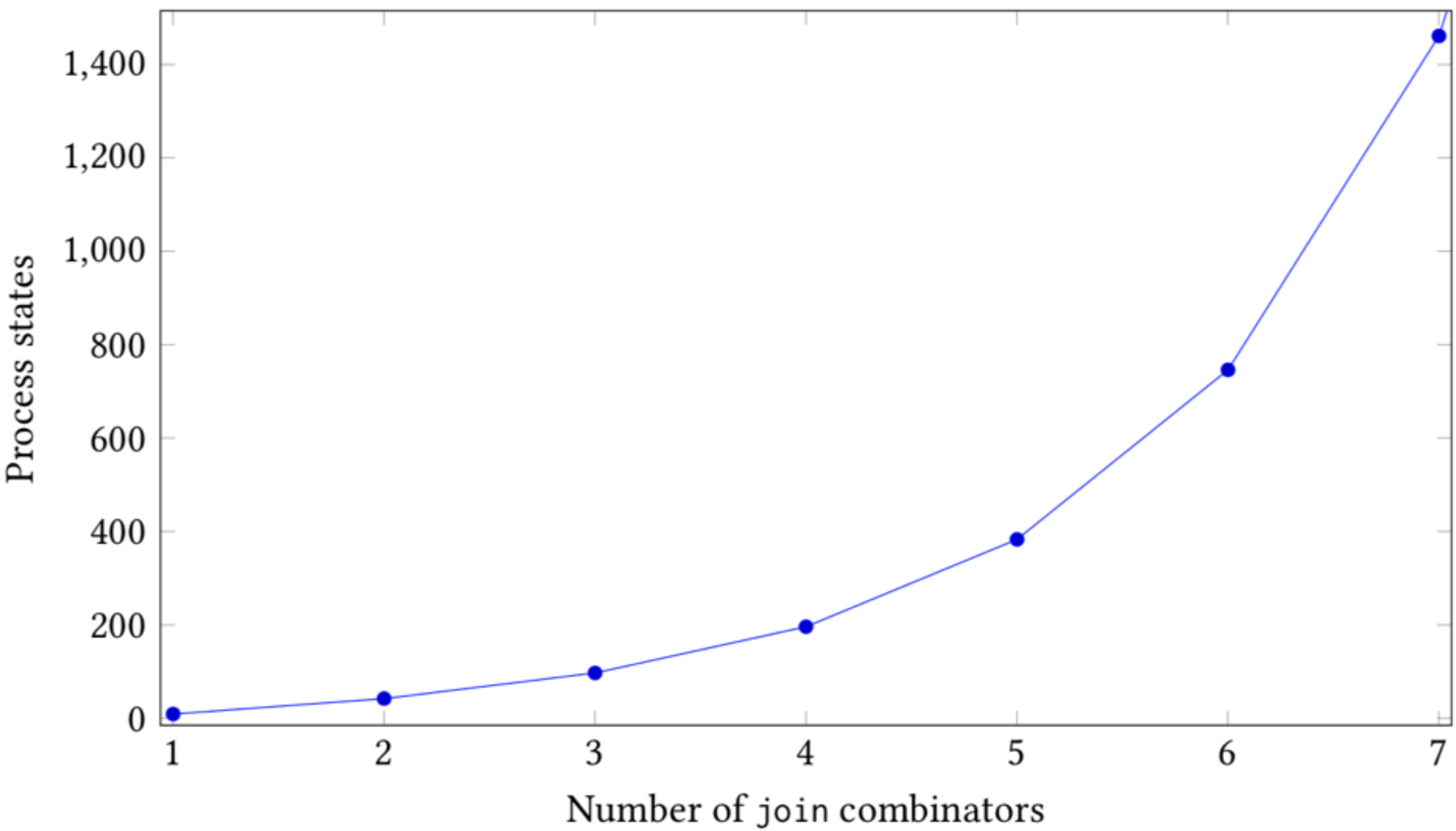
```
normalize2 :: Array Int -> (Array Int, Array Int)
normalize2 xs
 = let sum1 = fold   (+)  0    xs
       gts  = filter (>   0)   xs
       sum2 = fold   (+)  0    gts
       ys1  = map    (/ sum1)  xs
       ys2  = map    (/ sum2)  xs
   in (ys1, ys2)
```



$$
\begin{array}{llll}
\text{Minimise} & \multicolumn{3}{l}{25 \cdot x_{sum1,gts} + 1 \cdot x_{sum1,sum2} + 25 \cdot x_{sum1,ys2} +} \\
& \multicolumn{3}{l}{25 \cdot x_{gts,sum2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} +} \\
& \multicolumn{3}{l}{25 \cdot x_{ys1,ys2} + 5 \cdot c_{gts} + 5 \cdot c_{ys1} + 5 \cdot c_{ys2}} \\
\text{Subject to} & -5 \cdot x_{sum1,gts} & \leq \pi_{gts} - \pi_{sum1} & \leq 5 \cdot x_{sum1,gts} \\
& -5 \cdot x_{sum1,sum2} & \leq \pi_{sum2} - \pi_{sum1} & \leq 5 \cdot x_{sum1,sum2} \\
& -5 \cdot x_{sum1,ys2} & \leq \pi_{ys2} - \pi_{sum1} & \leq 5 \cdot x_{sum1,ys2} \\
& -5 \cdot x_{gts,ys1} & \leq \pi_{ys1} - \pi_{gts} & \leq 5 \cdot x_{gts,ys1} \\
& -5 \cdot x_{sum2,ys1} & \leq \pi_{ys1} - \pi_{sum2} & \leq 5 \cdot x_{sum2,ys1} \\
& -5 \cdot x_{ys1,ys2} & \leq \pi_{ys2} - \pi_{ys1} & \leq 5 \cdot x_{ys1,ys2} \\
& x_{gts,sum2} & \leq \pi_{sum2} - \pi_{gts} & \leq 5 \cdot x_{gts,sum2} \\
& & \pi_{sum1} < \pi_{ys1} & \\
& & \pi_{sum2} < \pi_{ys2} & \\
& x_{gts,sum2} & \leq c_{gts} & \\
& x_{gts,sum2} & \leq x_{sum1,sum2} & \\
& x_{sum1,sum1} & \leq x_{sum1,sum2} & \\
& x_{sum1,gts} & \leq x_{sum1,sum2} &
\end{array}
$$



100
80
60
40
20
0
Process states
1
2
3
4
5
6
7
Number of combinators
Pipelined with 1 join
Parallel with 1 join
Pipe/parallel without join



Process states
1,400
1,200
1,000
800
600
400
200
0
1
2
3
4
5
6
7
Number of join combinators