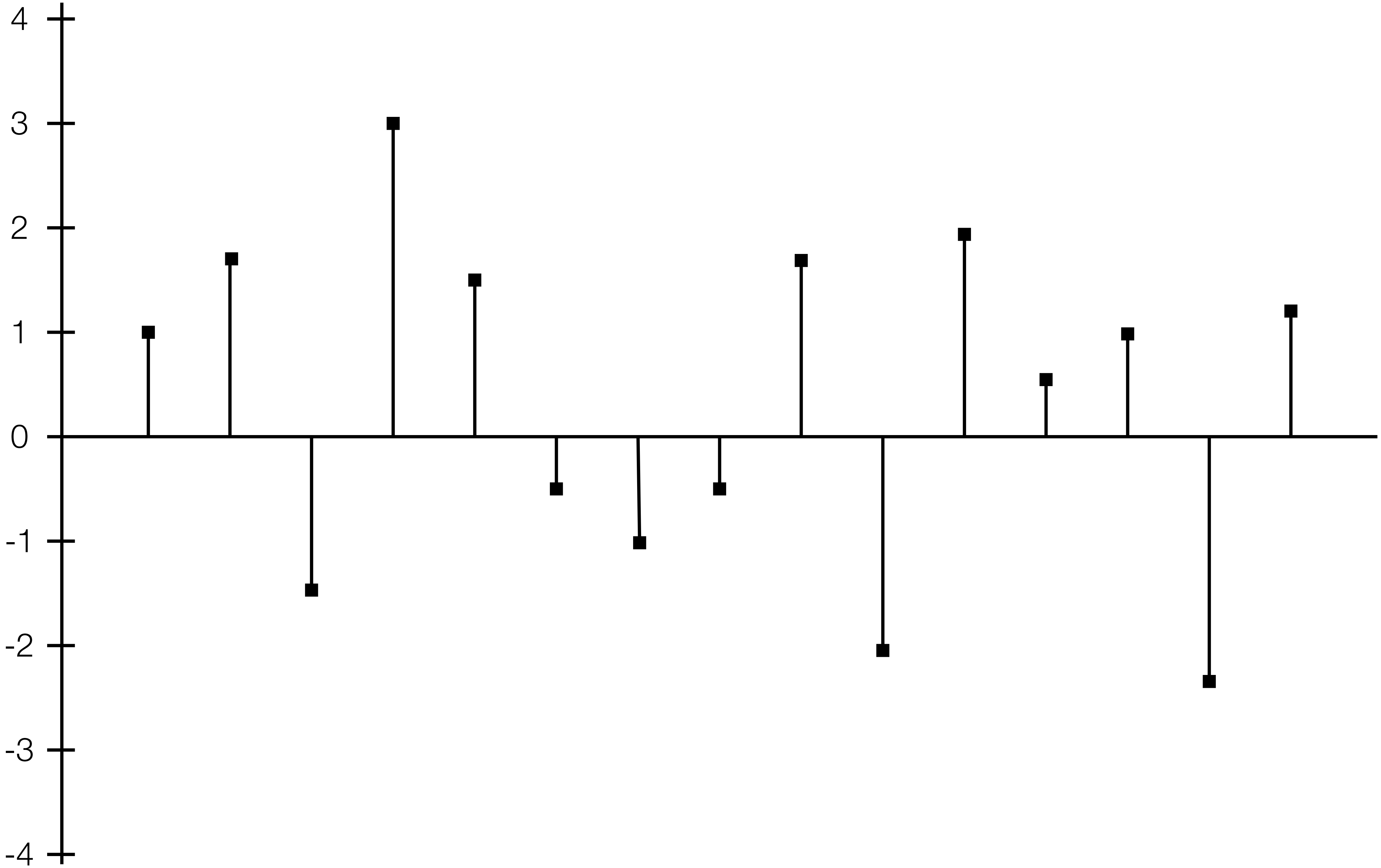
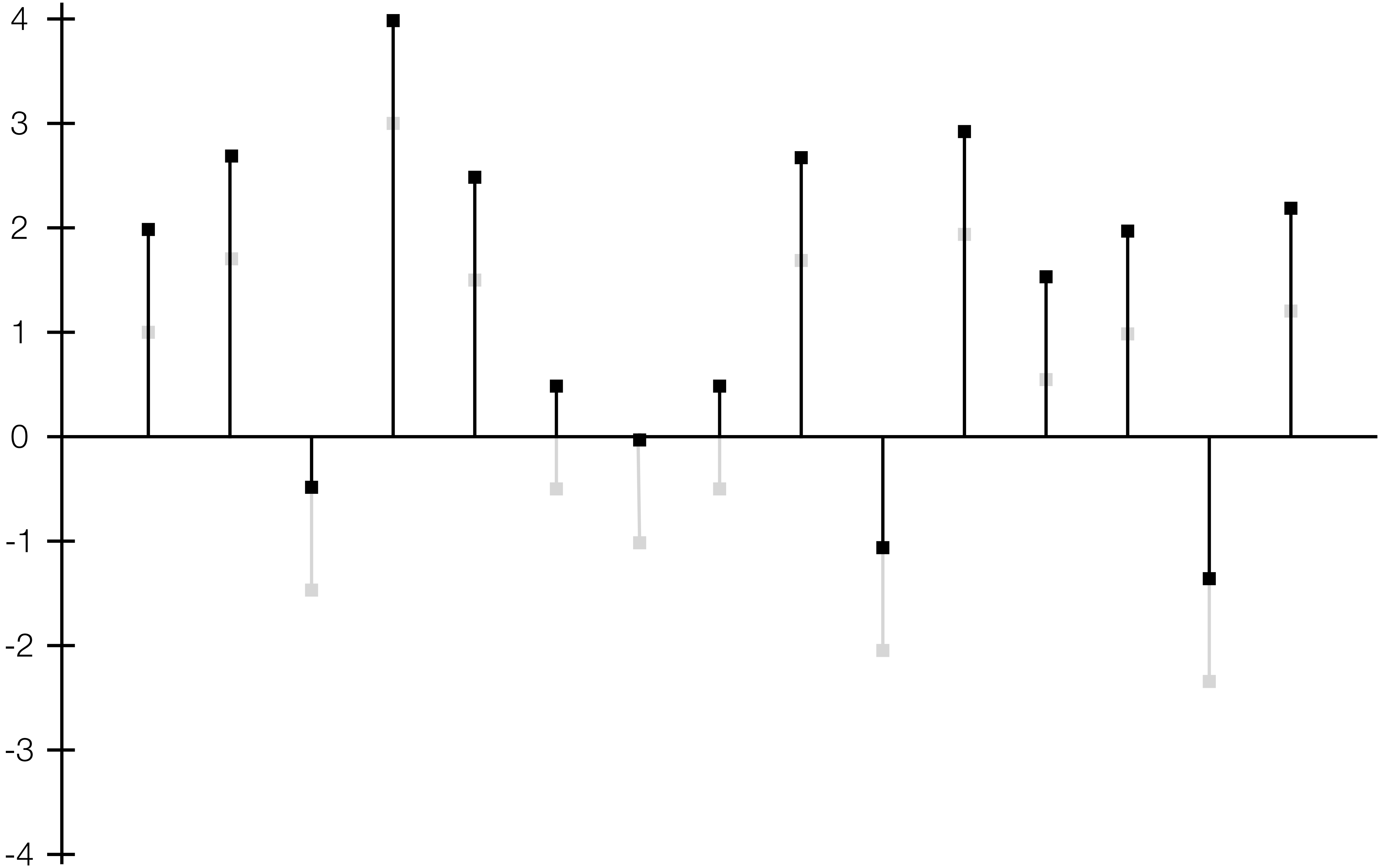
Query Fusion for the biggest data

Ben Lippmeier
3/10/2014

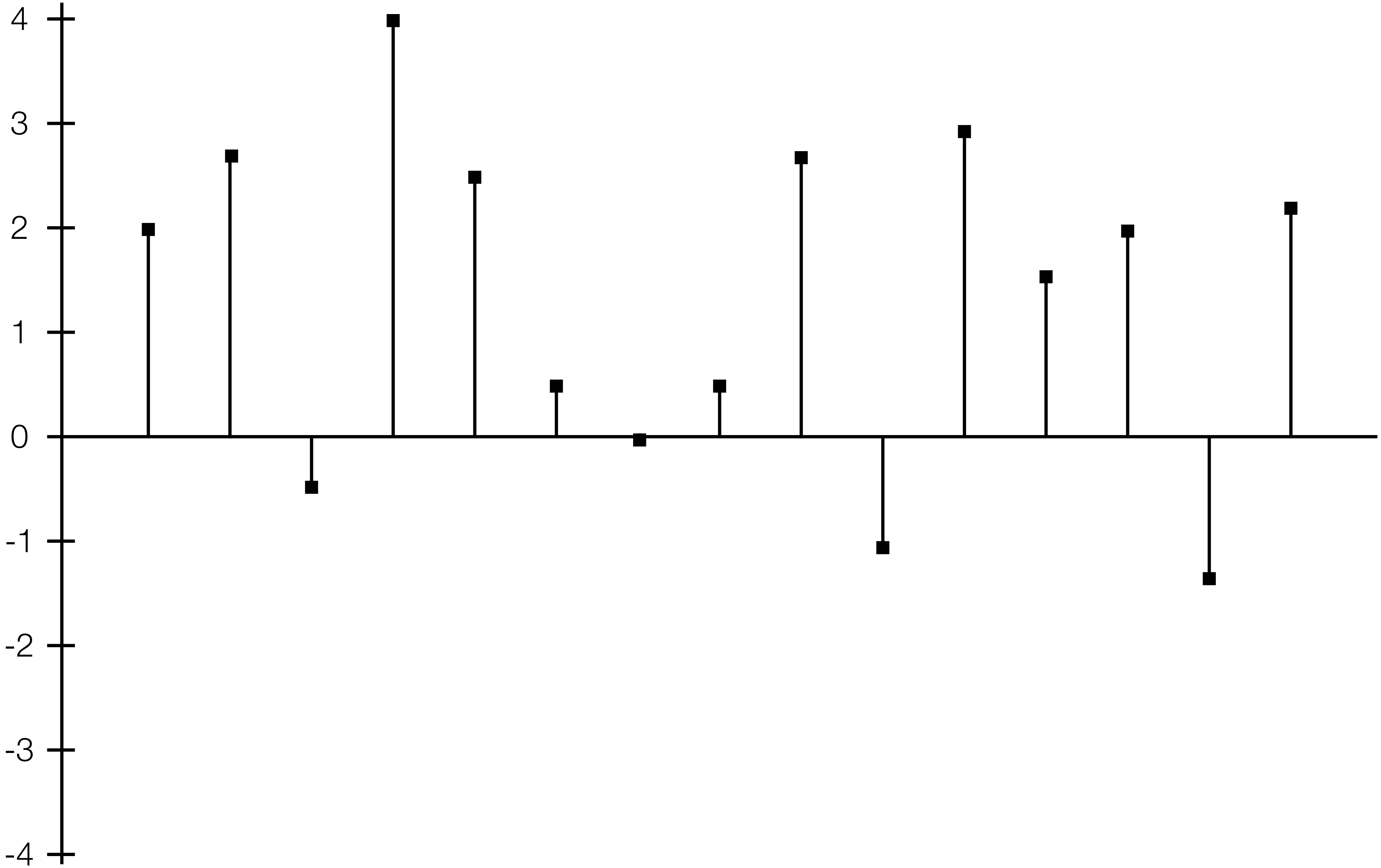
FilterMax



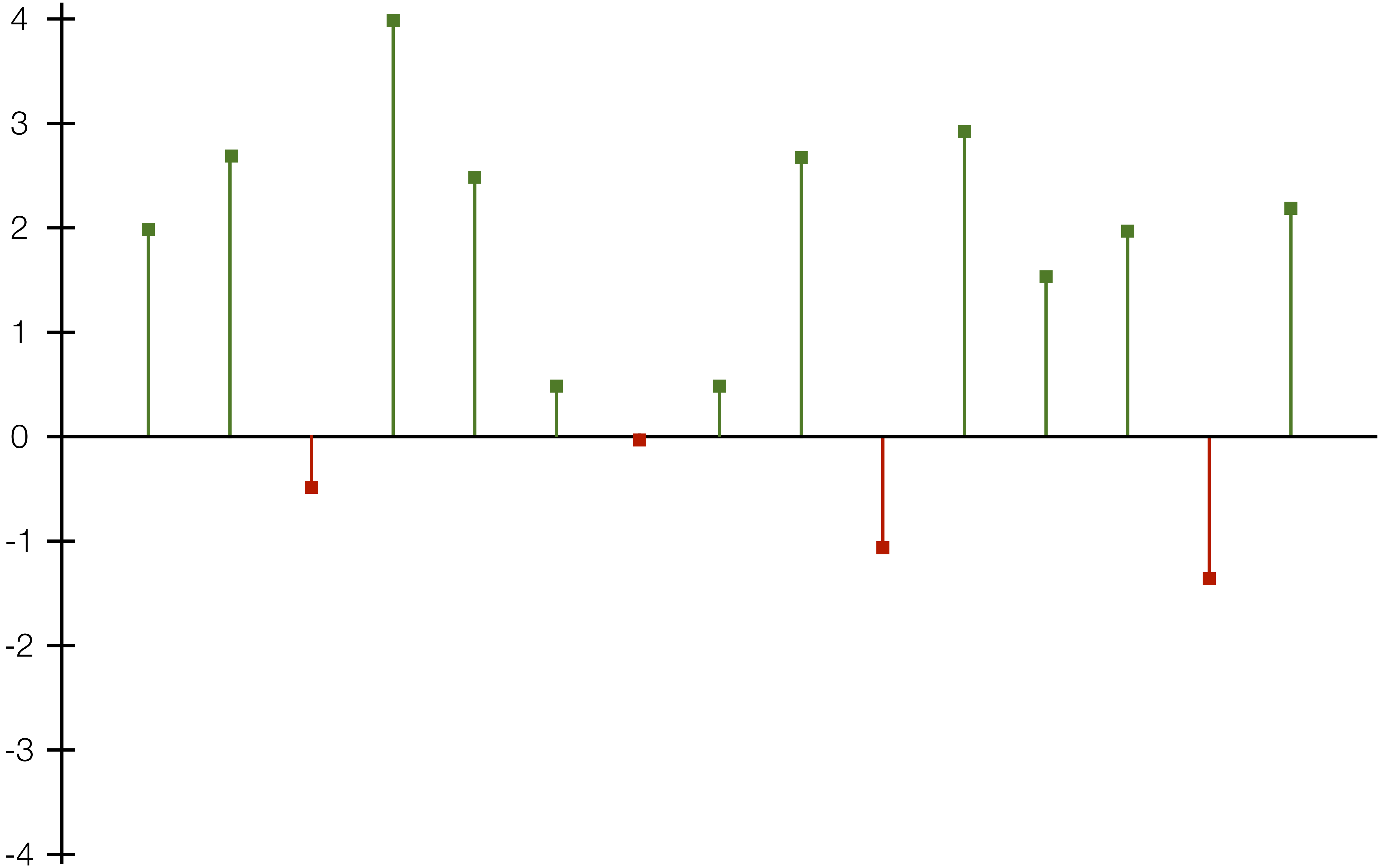
FilterMax



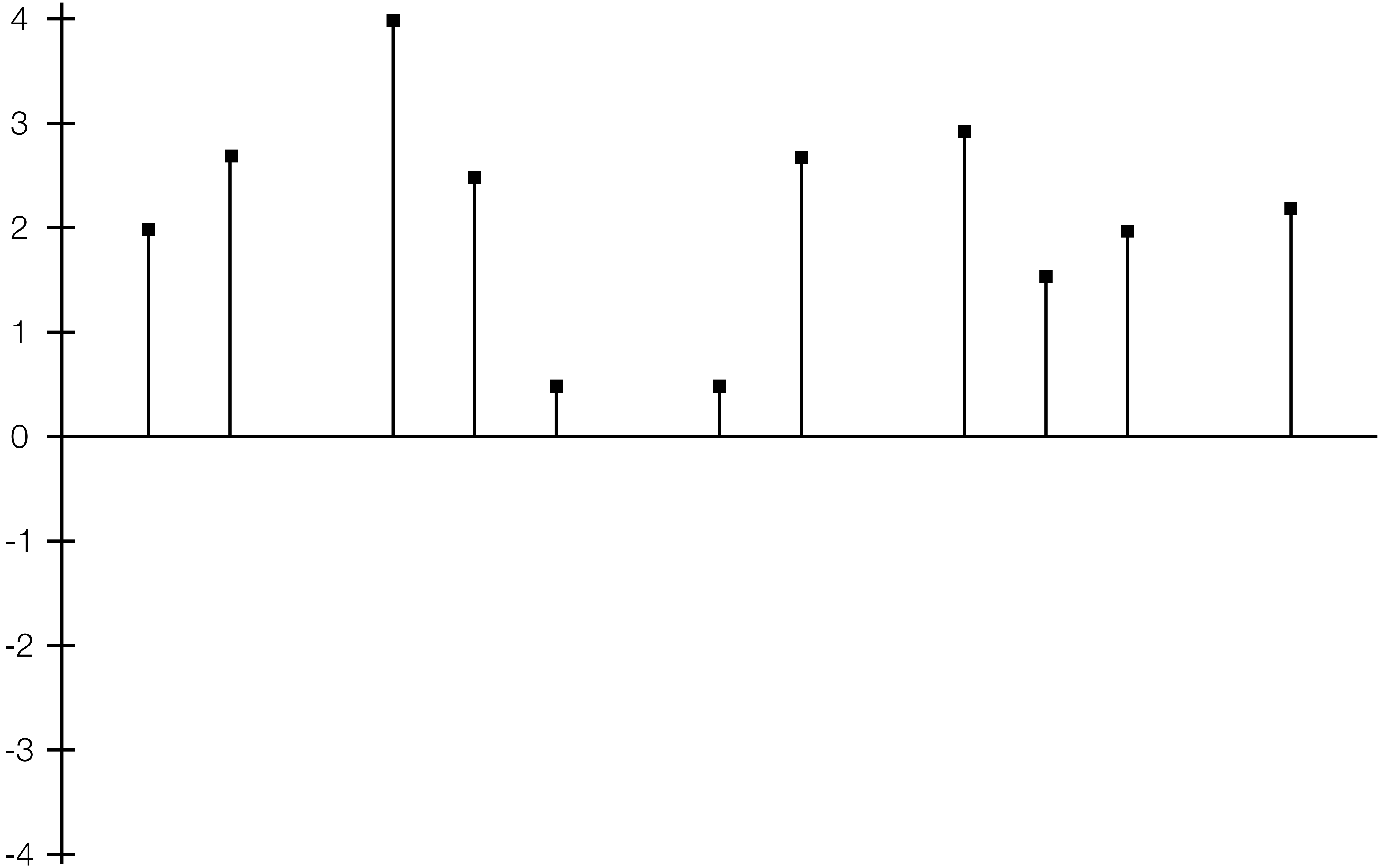
FilterMax



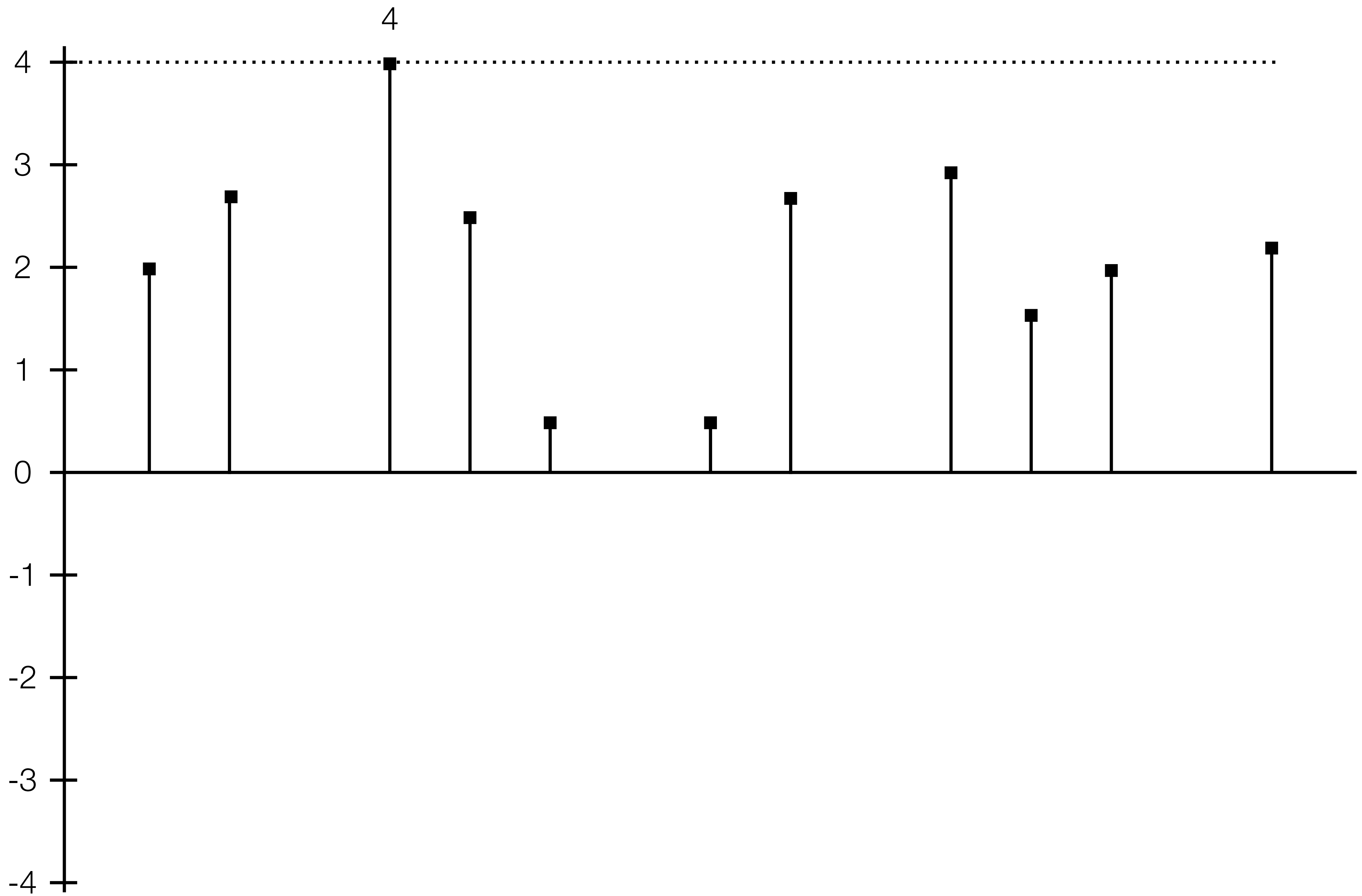
FilterMax



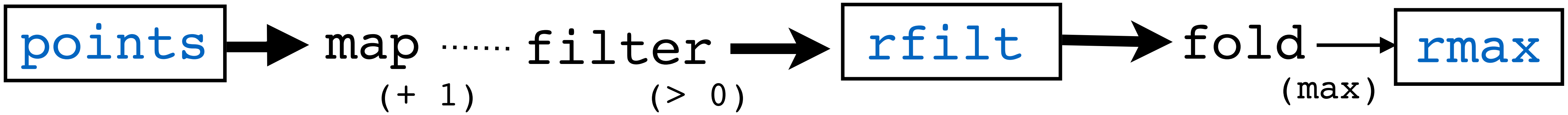
FilterMax



FilterMax

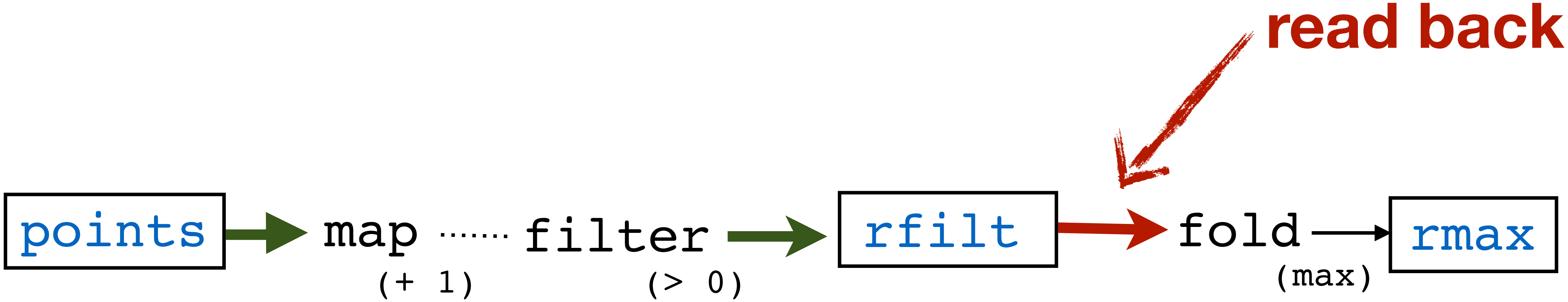


```
create table rfilt as
select y + 1 as h
from   points
where  h > 0

create table rmax as
select max(h) from rfilt
```



points → map (+ 1) ······· filter (> 0) → rfilt → fold (max) → rmax

```
create table rfilt as
select y + 1 as h
from   points
where  h > 0

create table rmax as
select max(h) from rfilt
```



read back

points

map (+ 1)

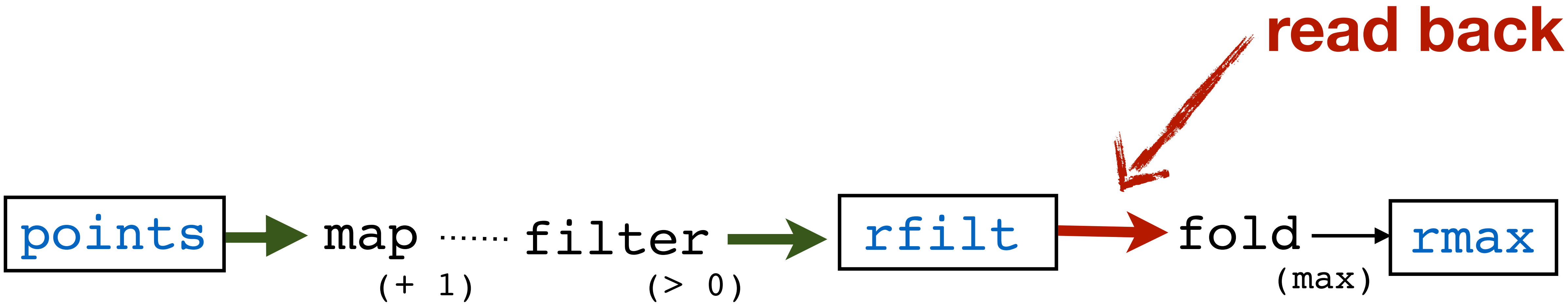
filter (> 0)

rfilt

fold (max)

rmax

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax points
 = let t2    = map (+ 1) points
       rfilt = filter (> 0) t2
       rmax  = fold max 0 rfilt
   in  (rfilt, rmax)
```



read back

points map (+ 1) filter (> 0) rfilt fold (max) rmax

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax points
 = let t2    = map (+ 1) points
       rfilt = filter (> 0) t2
       rmax  = fold max 0 rfilt
   in  (rfilt, rmax)
```

```
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 points
 = let t2    = unstream (mapS (+ 1) (stream points))
       rfilt = unstream (filterS (> 0) (stream t2))
       rmax  = foldS max 0 (stream rfilt)
   in  (rfilt, rmax)
```

```
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 points
 = let
      rfilt = unstream (filterS (> 0)
                 (stream (unstream (mapS (+ 1)
                              (stream points)))))
      rmax  = foldS max 0 (stream rfilt)
   in  (rfilt, rmax)
```

```
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 points
 = let
       rfilt = unstream (filterS (> 0)
                 (stream (unstream (mapS (+ 1)
                               (stream points)))))
       rmax  = foldS max 0 (stream rfilt)
   in  (rfilt, rmax)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax points
 = let
       rfilt = unstream (filterS (> 0)
                   (stream (unstream (mapS (+ 1)
                                (stream points)))))
       rmax  = foldS max 0 (stream rfilt)
   in  (rfilt, rmax)
```

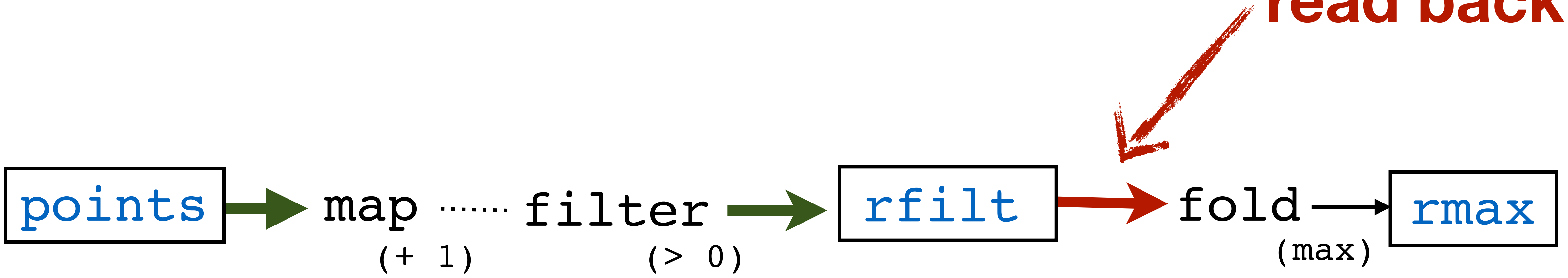
```
RULE “stream/unstream”
     forall xs. stream (unstream xs) = xs
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax points
 = let
       rfilt = unstream (filterS (> 0)
                 ~~(stream (unstream~~ (mapS (+ 1)
                               (stream points)))))
       rmax  = foldS max 0 (stream rfilt)
   in  (rfilt, rmax)
```

```
    RULE “stream/unstream”
         forall xs. stream (unstream xs) = xs
```

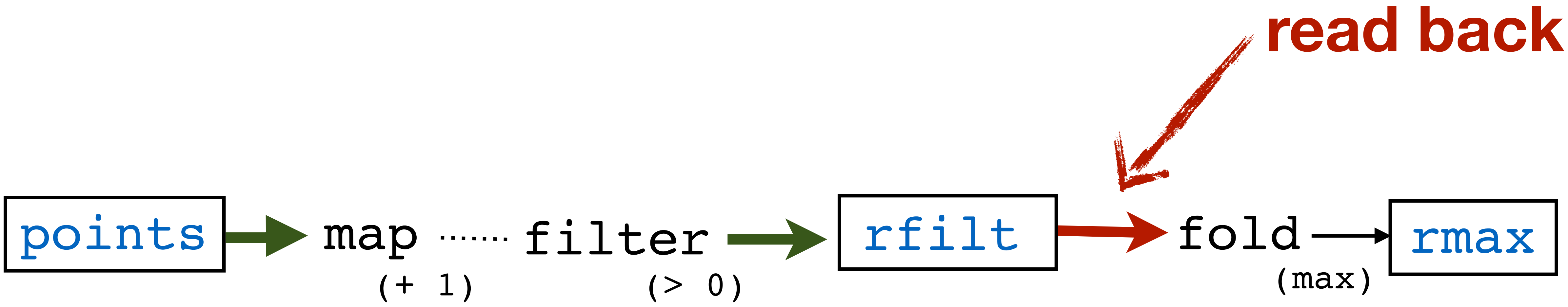
```haskell
filterMax :: Vector Int -> (Vector Int, Int)
filterMax points
 = let
       rfilt = unstream (filterS (> 0) (mapS (+ 1)
                                (stream points)))
       rmax  = foldS max 0 (stream rfilt)
   in  (rfilt, rmax)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax points
 = let
       rfilt = unstream (filterS (> 0) (mapS (+ 1)
                               (stream points)))
       rmax  = foldS max 0 (stream rfilt)
   in  (rfilt, rmax)
```



Back to SQL

```
create table rfilt as
select y + 1 as h
from   points
where  h > 0

create table rmax as
select max(h) from rfilt
```



read back

points

map

(+ 1)

filter

(> 0)

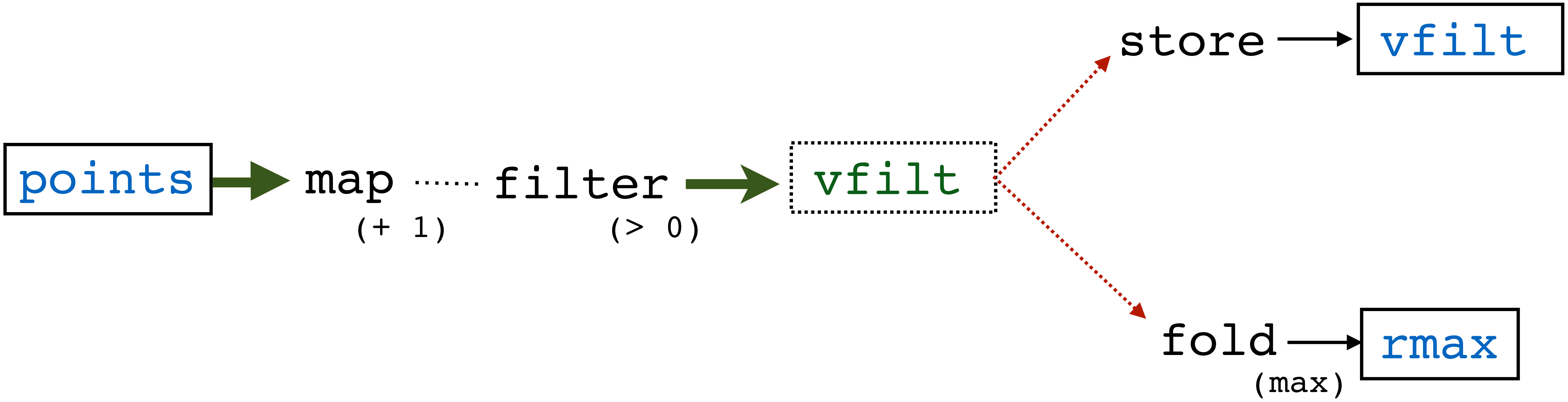
rfilt

fold

(max)

rmax

```
create view vfilt as
select y + 1 as h
from   points
where  h > 0

create table rfilt as
select * from vfilt

create table rmax as
select max(h) from vfilt
```



points

map (+ 1)

filter (> 0)

vfilt

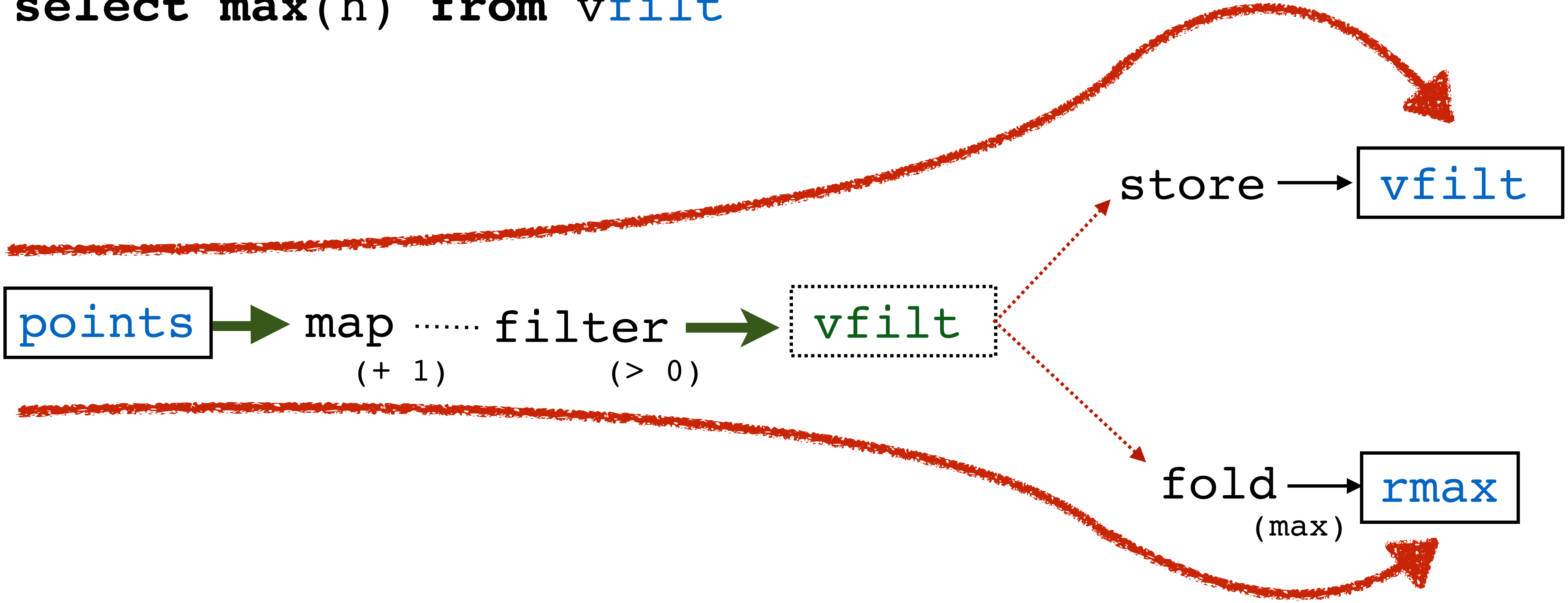
store

vfilt

fold (max)

rmax

```
create view vfilt as
select y + 1 as h
from   points
where  h > 0

create table rfilt as
select * from vfilt

create table rmax as
select max(h) from vfilt
```



points

map

(+ 1)

filter

(> 0)

vfilt

store

vfilt

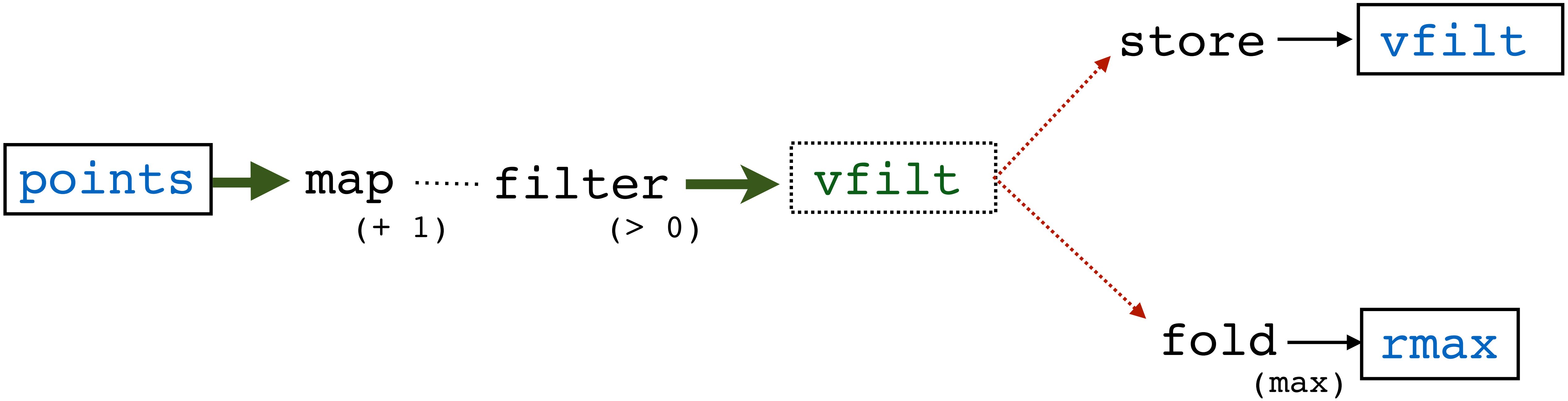
fold

(max)

rmax

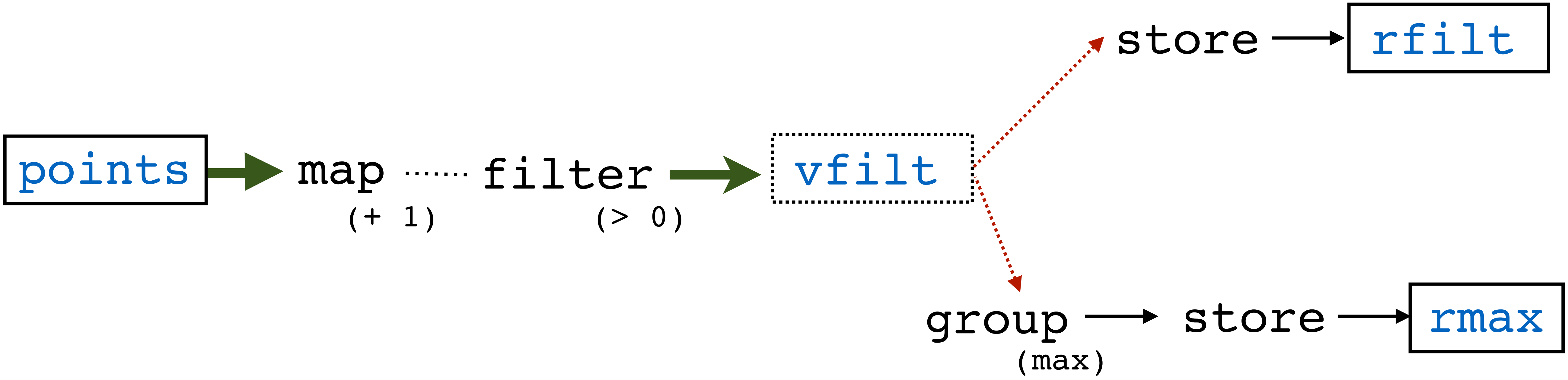
Apache Pig

```
create view vfilt as
select y + 1 as h
from   points
where  h > 0

create table rfilt as
select * from vfilt

create table rmax as
select max(h) from vfilt
```

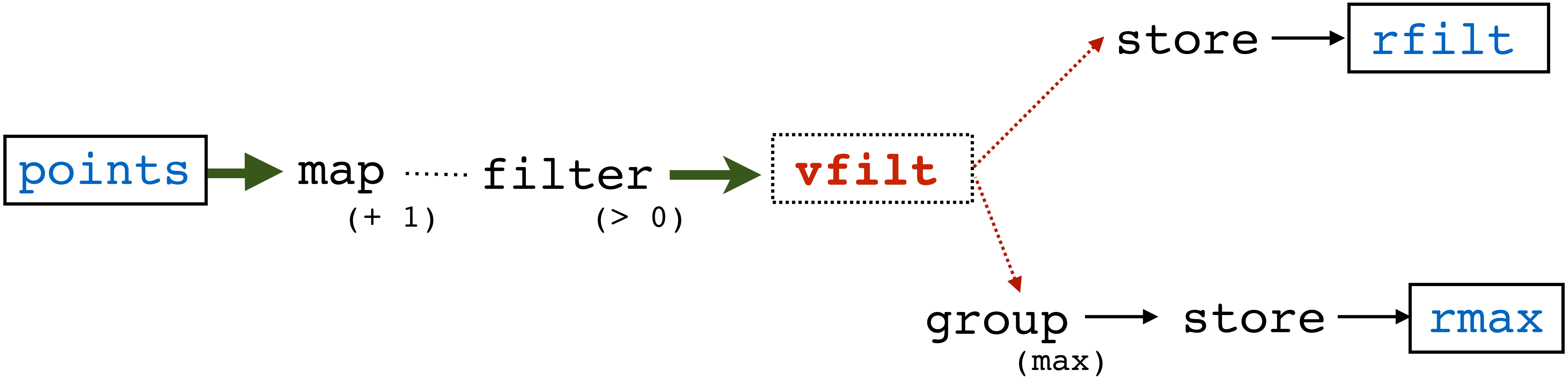


points
map
(+ 1)
filter
(> 0)
vfilt
store
vfilt
fold
(max)
rmax

```
points = load ‘points.txt' as (h:int);
ps1    = foreach points generate g + 1 as h1;
vfilt  = filter ps1 by h1 > 0;
vfiltg = group vfilt all;
vmax   = foreach vfiltg generate MAX(vfilt.h1);
store vfilt into ‘rfilt.txt’
store vmax  into ‘rmax.txt’
```



```
points = load ‘points.txt' as (h:int);
ps1    = foreach points generate g + 1 as h1;
vfilt  = filter ps1 by h1 > 0;
vfiltg = group vfilt all;
vmax   = foreach vfiltg generate MAX(vfilt.h1);
store vfilt into ‘rfilt.txt’
store vmax  into ‘rmax.txt’
```



points

map
(+ 1)

filter
(> 0)

vfilt

store

rfilt

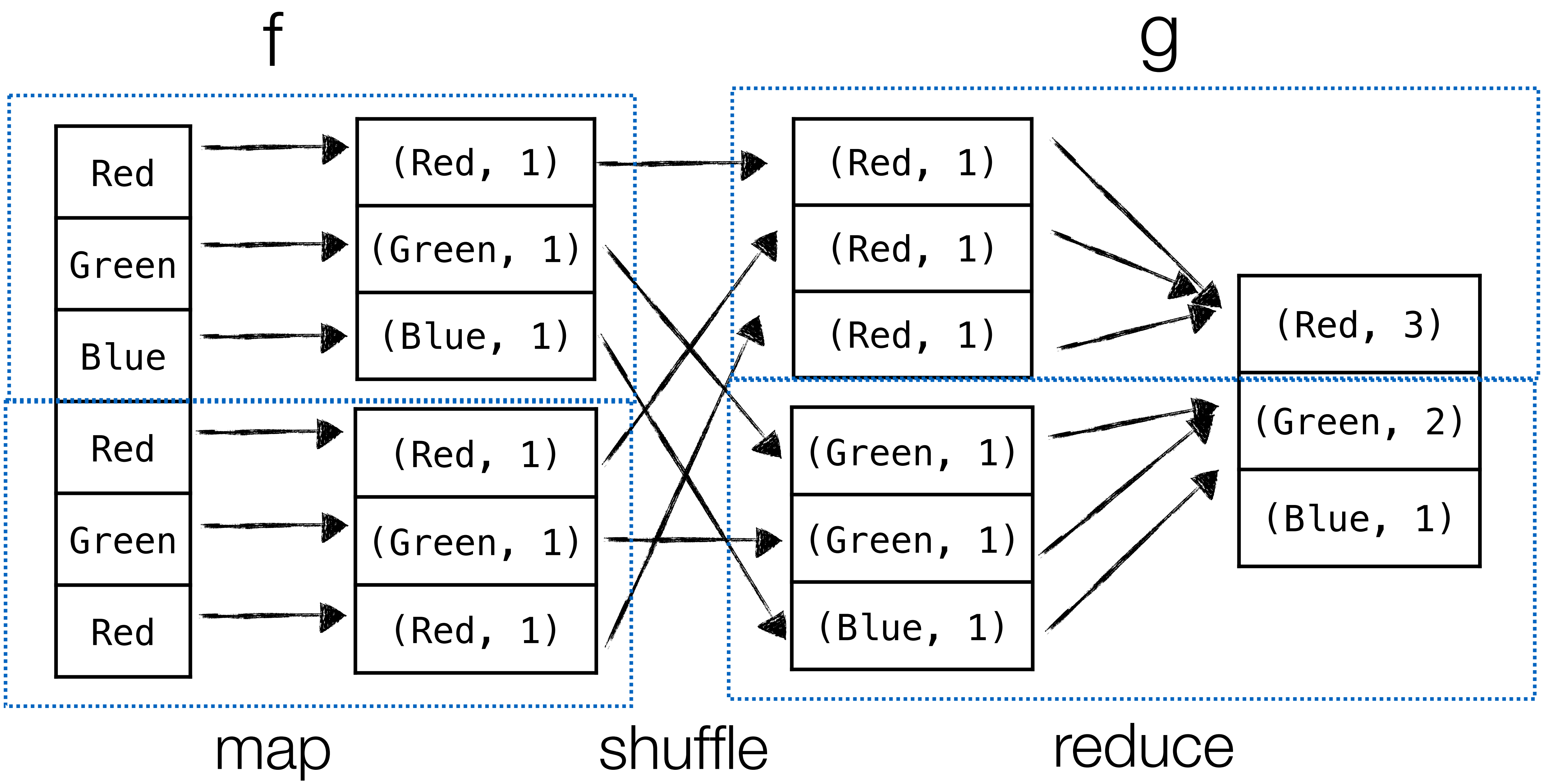
group
(max)

store

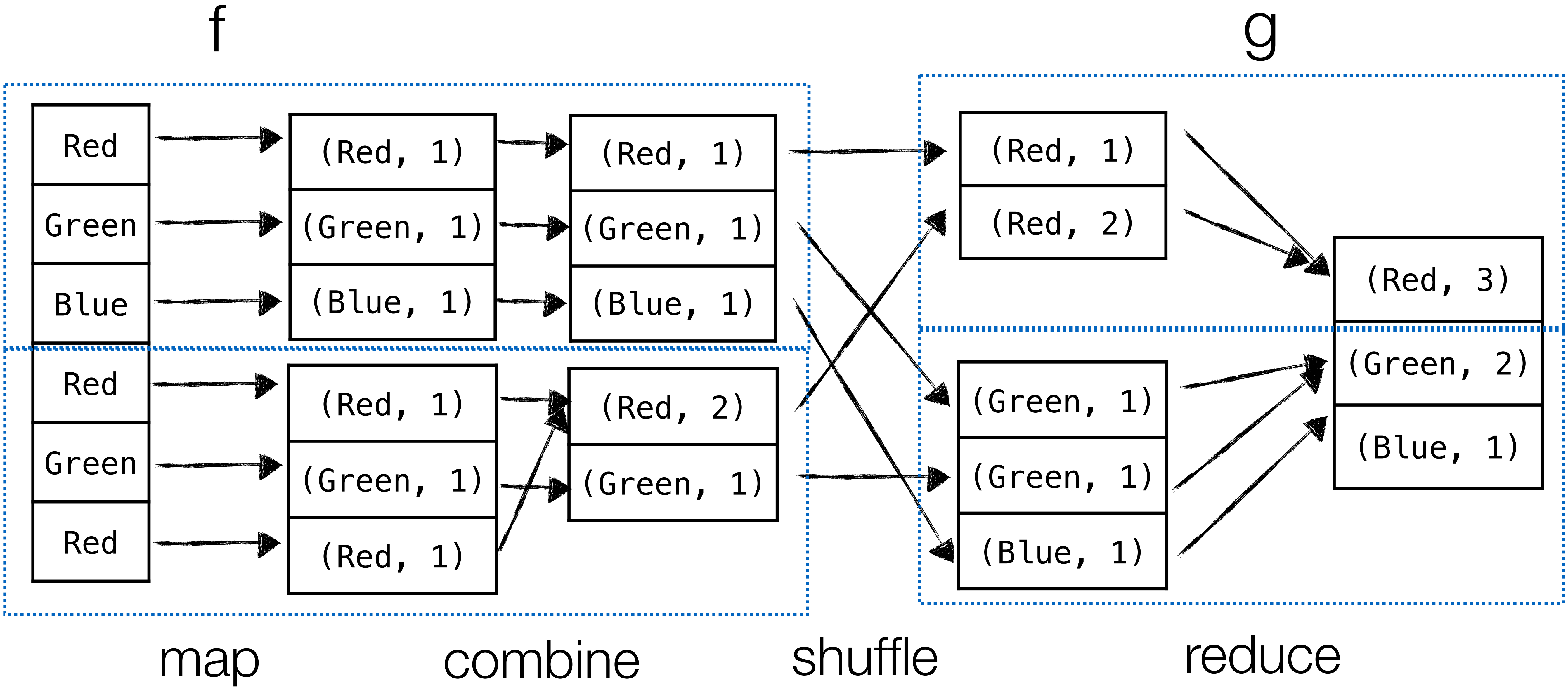
rmax

```
Map Plan
Split - scope-52
|   rfilt: Store(file:///Users/majestic/rfilt.txt:org.apache.pig.builtin.PigStorage) - scope-14
|   rfiltg: Local Rearrange[tuple]{chararray}(false) - scope-45
|   |   Project[chararray][0] - scope-46
|   |---rmax: New For Each(false,false)[bag] - scope-33
|       |   Project[chararray][0] - scope-34
|       |   POUserFunc(org.apache.pig.builtin.AlgebraicMathBase$Initial)[tuple] - scope-35
|       |   |---Project[bag][0] - scope-36
|       |       |---Project[bag][1] - scope-37
|       |---Pre Combiner Local Rearrange[tuple]{Unknown} - scope-47
|---rfilt: Filter[bag] - scope-7
   |   Greater Than[boolean] - scope-10
   |   |---Project[int][0] - scope-8
   |   |---Constant(0) - scope-9
   |---ps1: New For Each(false)[bag] - scope-6
       |   Add[int] - scope-4
       |   |---Cast[int] - scope-2
       |   |   |---Project[bytearray][0] - scope-1
       |   |---Constant(1) - scope-3
       |---points: Load(file:///Users/majestic/points.txt:org.apache.pig.builtin.PigStorage) - scope-0--------

Combine Plan
rfiltg: Local Rearrange[tuple]{chararray}(false) - scope-49
|   Project[chararray][0] - scope-50
|---rmax: New For Each(false,false)[bag] - scope-38
   |   Project[chararray][0] - scope-39
   |   POUserFunc(org.apache.pig.builtin.LongSum$Intermediate)[tuple] - scope-40
   |   |---Project[bag][1] - scope-41
   |---POCombinerPackage[tuple]{chararray} - scope-43--------

Reduce Plan
rmax: Store(file:///Users/majestic/rmax.txt:org.apache.pig.builtin.PigStorage) - scope-26
|---rmax: New For Each(false)[bag] - scope-25
   |   POUserFunc(org.apache.pig.builtin.LongSum$Final)[long] - scope-23
   |   |---Project[bag][1] - scope-42
   |---POCombinerPackage[tuple]{chararray} - scope-51--------
```

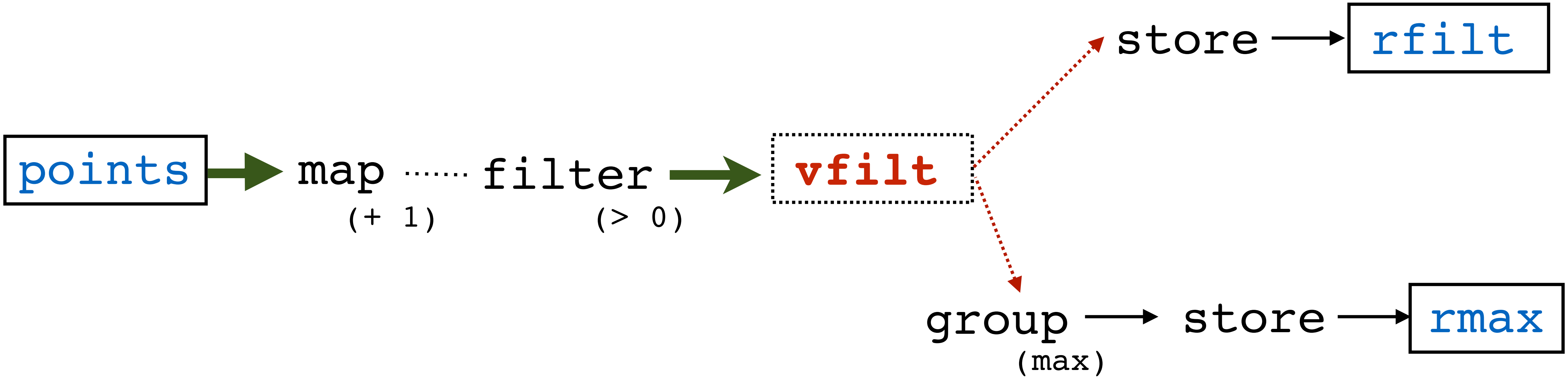
```
mapReduce :: Ord k
          => (k -> Array (Pair k a)))  - f
          -> (a -> a -> a)             - g
          -> Array k
          -> Array (Pair k a)
```



```
mapReduce :: Ord k
          => (k -> Array (Pair k a)))  - f
          -> (a -> a -> a)             - g
          -> Array k
          -> Array (Pair k a)
```



```
points = load ‘points.txt' as (h:int);
ps1    = foreach points generate g + 1 as h1;
vfilt  = filter ps1 by h1 > 0;
vfiltg = group vfilt all;
vmax   = foreach vfiltg generate MAX(vfilt.h1);
store vfilt into ‘rfilt.txt’
store vmax  into ‘rmax.txt’
```

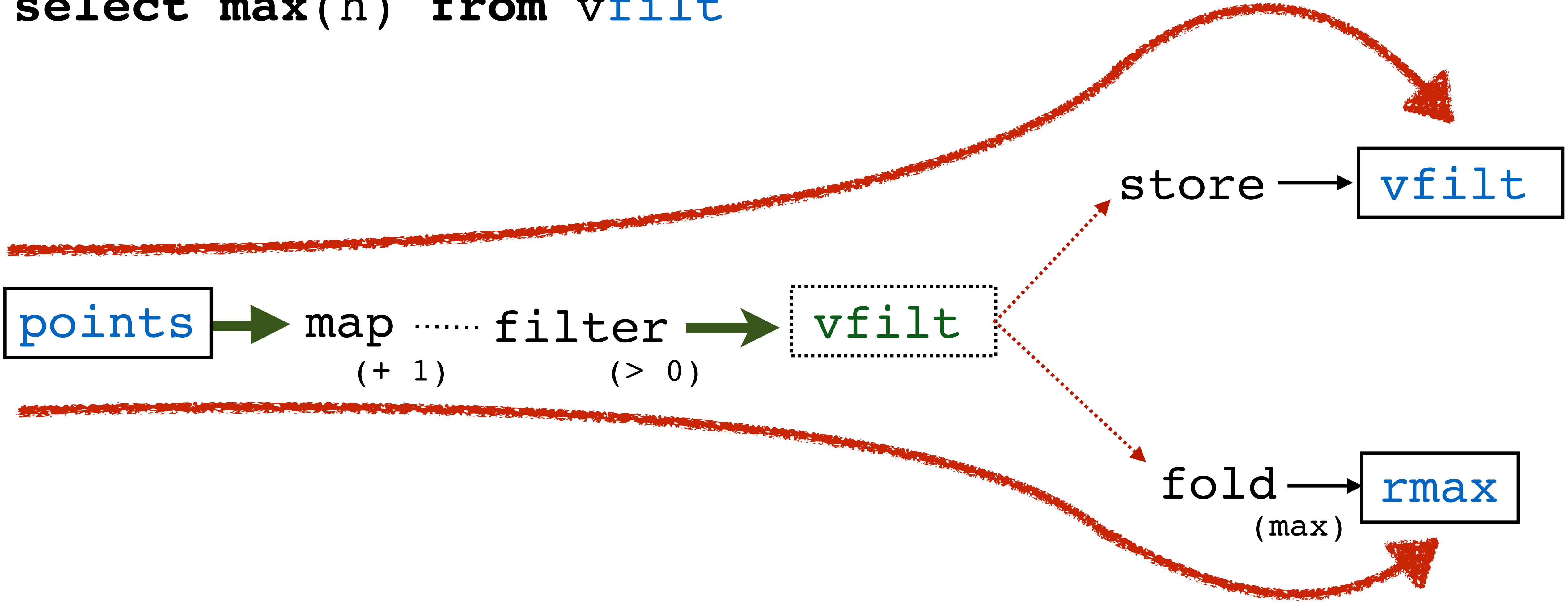


- Characterisation of queries / data flow graphs that can be evaluated with single MapReduce jobs?

- Related work says: we convert queries to MR jobs like this, and here are some optimisations that sometimes apply.

Apache Hive

```
create view vfilt as
select y + 1 as h
from   points
where  h > 0

create table rfilt as
select * from vfilt

create table rmax as
select max(h) from vfilt
```



points

map

(+ 1)

filter

(> 0)

vfilt

store

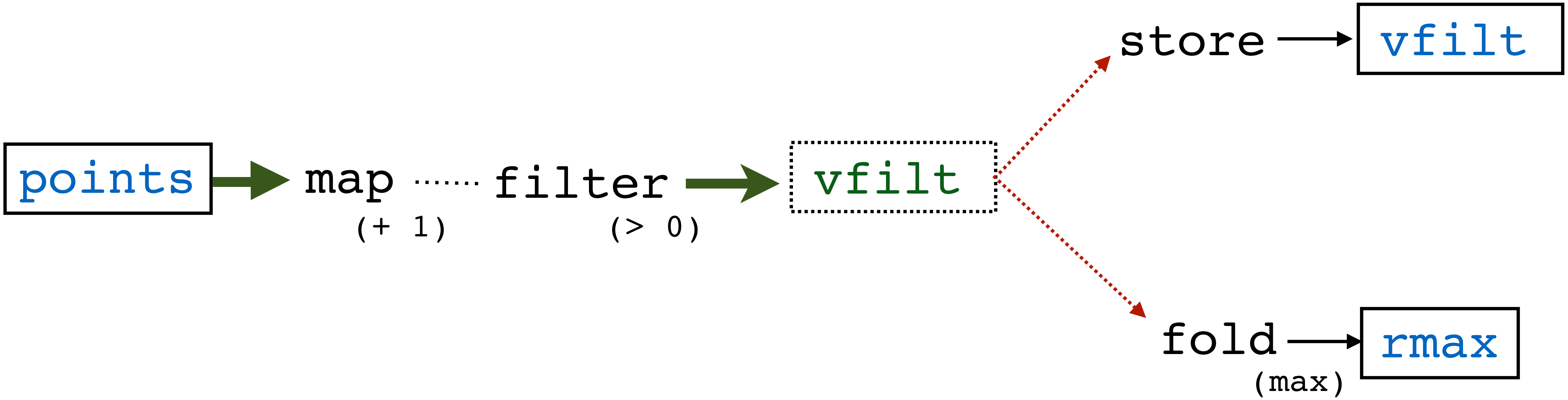
vfilt

fold

(max)

rmax

```
from (select y + 1 as h
      from   points
      where  h > 0) vfilt
insert overwrite table rfilt
select vfilt.h
insert overwrite table rmax
select max(vfilt.h)
```



Data Flow Fusion

Data Flow Fusion

- Guaranteed fusion for a particular class of queries.
- Can evaluate these in constant space with single imperative loops.
- Straightforward to do native code generation.

Data Flow Fusion

1. **Refactor** to expose desired data flow.

(currently working to make this automatic)

2. **Slurp** out a data flow graph from the source.

3. **Schedule** the graph into an abstract loop nest.

4. **Extract** implementation code from the nest.

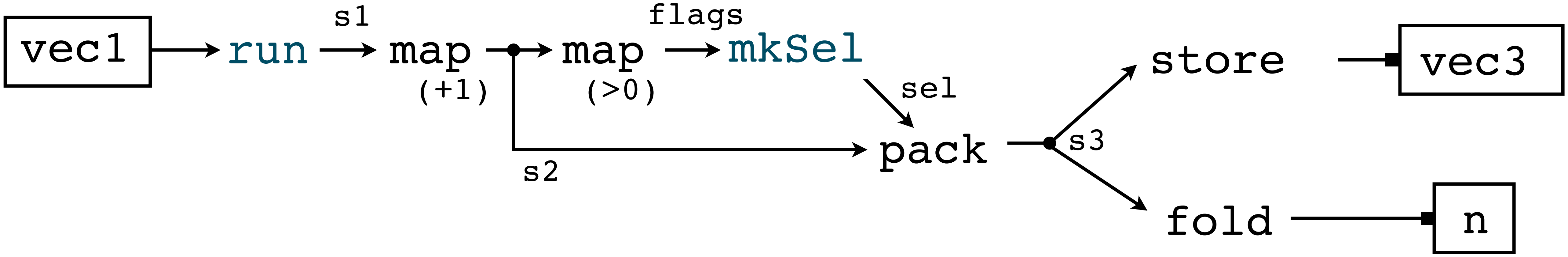
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax points
 = let t2    = map (+ 1) points
       rfilt = filter (> 0) t2
       rmax  = fold max 0 rfilt
   in  (rfilt, rmax)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = store s3
             n    = fold max 0 s3
         in (vec3, n)))
```

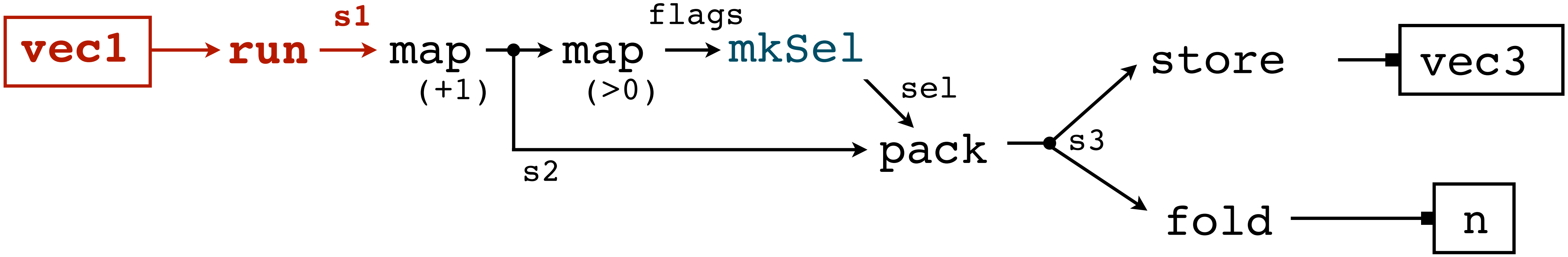
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = store s3
             n    = fold max 0 s3
         in (vec3, n)))
```

```
pack (Sel [T F F T F]) [1 2 3 4 5] = [1 4]
```

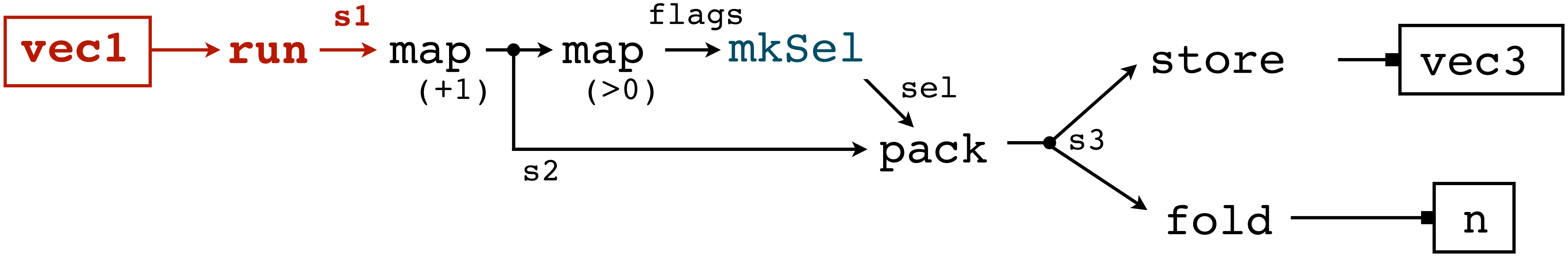
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = store s3
             n    = fold max 0 s3
         in (vec3, n)))
```



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = store s3
             n    = fold max 0 s3
         in (vec3, n)))
```



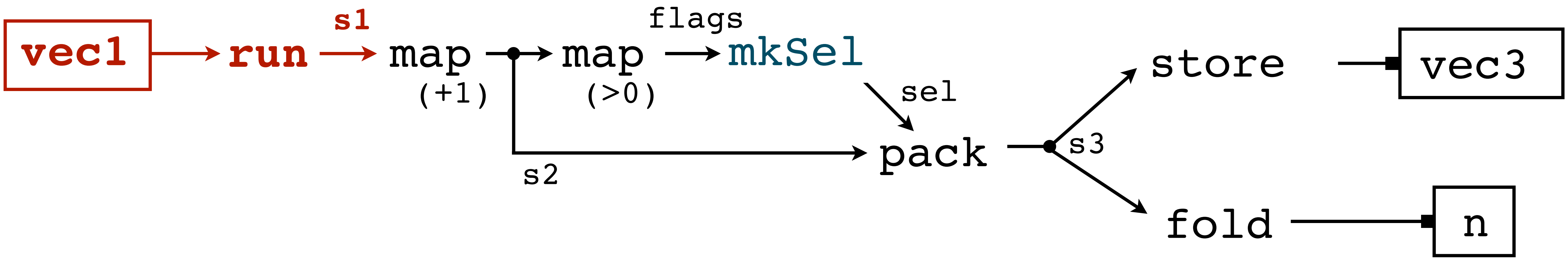
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                  s1 :: Series k1 Int
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = store s3
             n    = fold max 0 s3
         in (vec3, n)))
```



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = store s3
             n    = fold max 0 s3
         in (vec3, n)))
```

`s1 :: Series k1 Int`

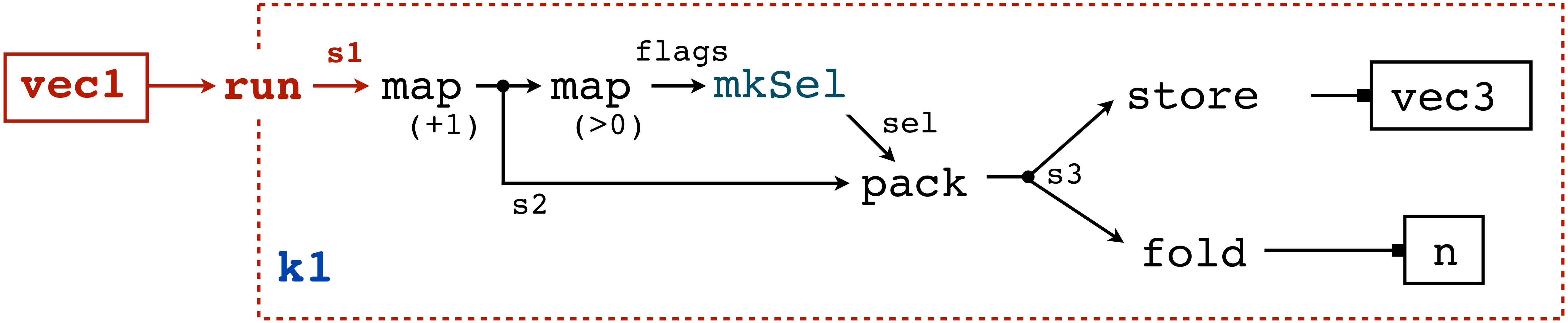
Rate Variable



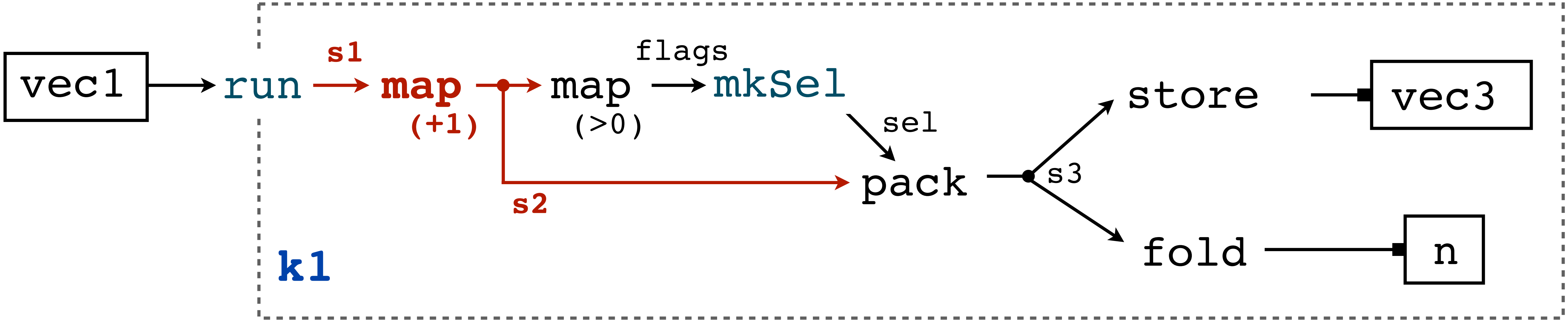
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->
    let s2    = map (+ 1) s1
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = store s3
             n    = fold max 0 s3
         in (vec3, n)))
```

`s1 :: Series k1 Int`

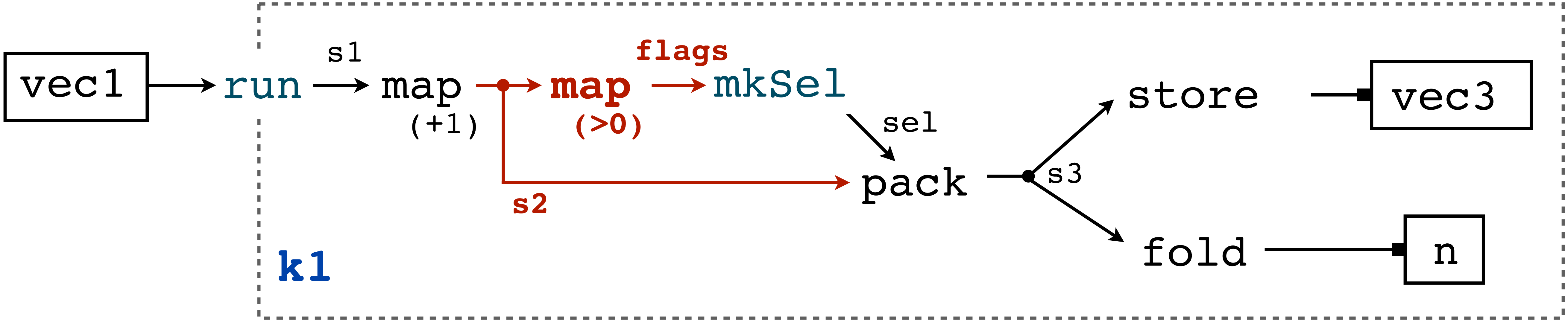
Rate Variable



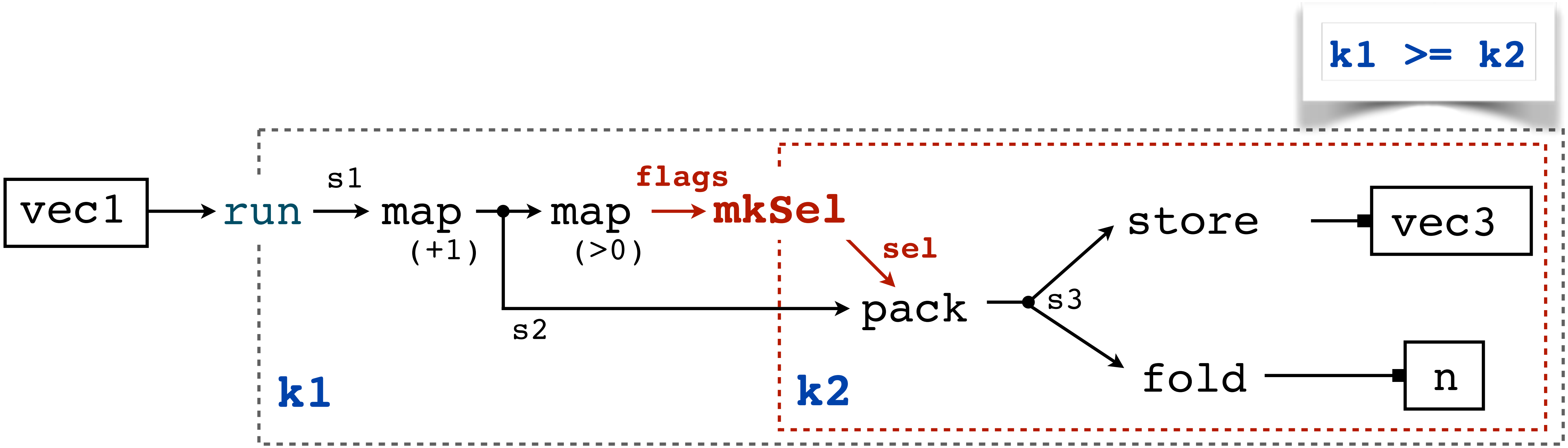
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                      s1 :: Series k1 Int
    let s2     = map (+ 1) s1            s2 :: Series k1 Int
        flags = map (> 0) s2
    in  mkSel flags (\sel ->
         let s3    = pack sel s2
             vec3 = store s3
             n     = fold max 0 s3
         in (vec3, n)))
```



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                  s1 :: Series k1 Int
    let s2    = map (+ 1) s1         s2 :: Series k1 Int
        flags = map (> 0) s2      flags :: Series k1 Bool
    in  mkSel flags (\sel ->
         let s3   = pack sel s2
             vec3 = store s3
             n    = fold max 0 s3
         in (vec3, n)))
```

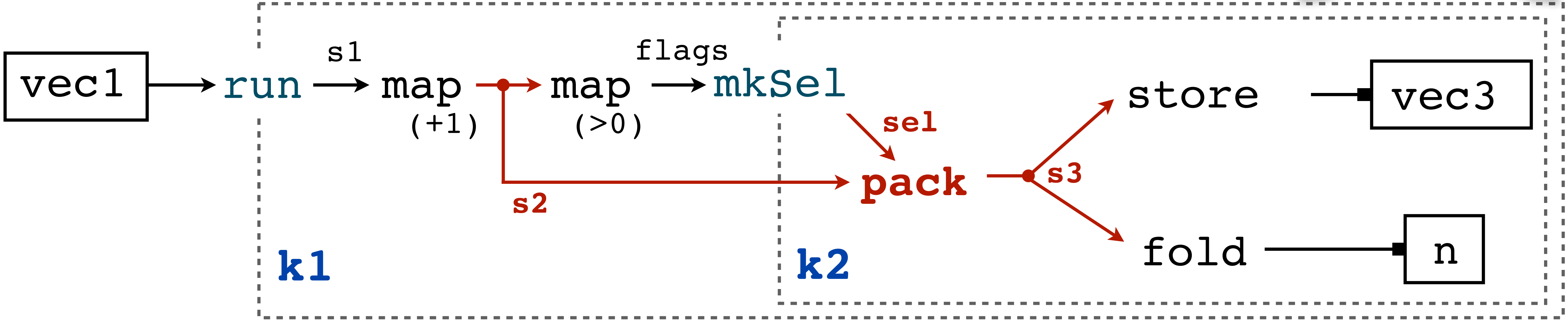


```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                  s1 :: Series k1 Int
    let s2    = map (+ 1) s1         s2 :: Series k1 Int
        flags = map (> 0) s2      flags :: Series k1 Bool
    in  mkSel flags (\sel ->        sel :: Sel k1 k2
         let s3   = pack sel s2
             vec3 = store s3
             n    = fold max 0 s3
         in (vec3, n)))
```



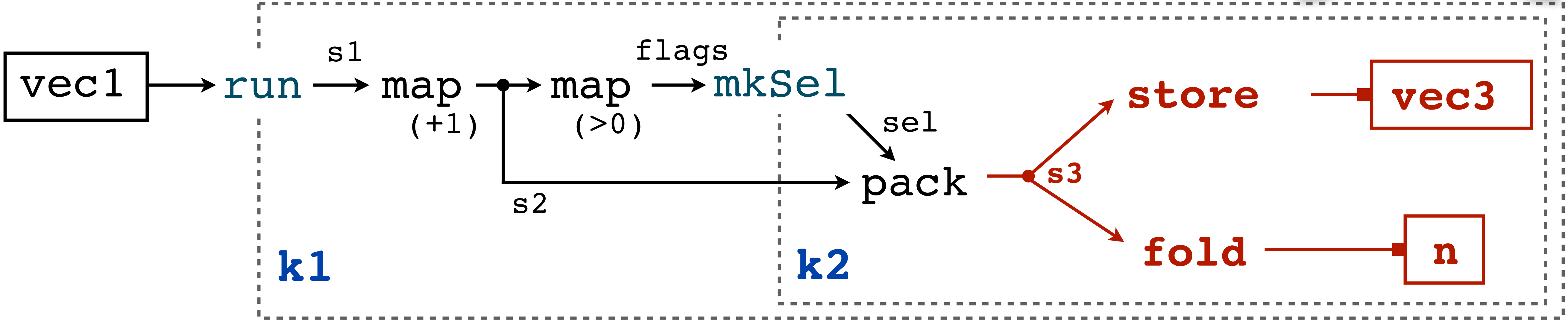
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                 s1 :: Series k1 Int
    let s2    = map (+ 1) s1         s2 :: Series k1 Int
        flags = map (> 0) s2      flags :: Series k1 Bool
    in  mkSel flags (\sel ->        sel :: Sel k1 k2
         let s3    = pack sel s2     s3 :: Series k2 Int
             vec3 = store s3
             n    = fold max 0 s3
         in (vec3, n)))
```

k1 >= k2

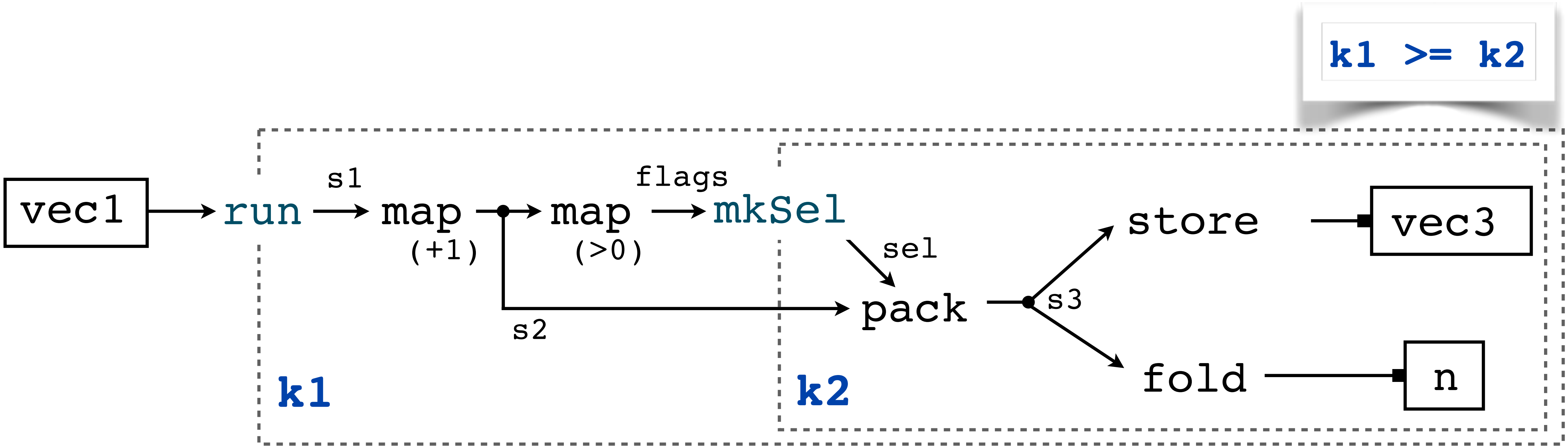


```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                  s1 :: Series k1 Int
    let s2     = map (+ 1) s1         s2 :: Series k1 Int
        flags = map (> 0) s2      flags :: Series k1 Bool
    in  mkSel flags (\sel ->        sel :: Sel k1 k2
         let s3   = pack sel s2       s3 :: Series k2 Int
             vec3 = store s3        vec3 :: Vector Int
             n    = fold max 0 s3      n :: Int
         in (vec3, n)))
```

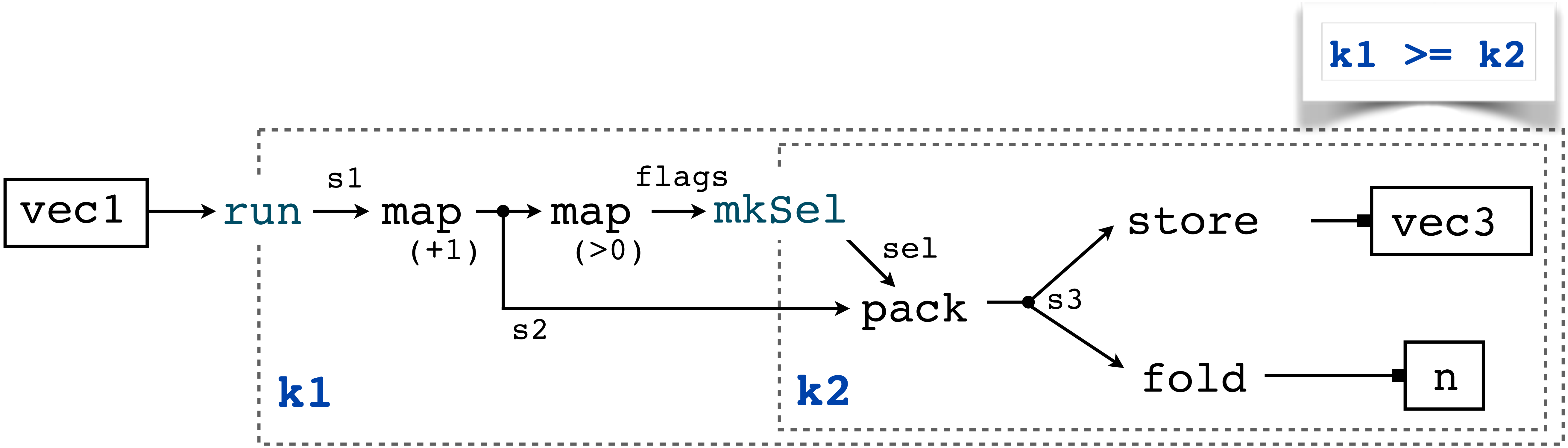
`k1 >= k2`



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                  s1 :: Series k1 Int
    let s2     = map (+ 1) s1         s2 :: Series k1 Int
        flags = map (> 0) s2      flags :: Series k1 Bool
    in  mkSel flags (\sel ->        sel :: Sel k1 k2
         let s3   = pack sel s2       s3 :: Series k2 Int
             vec3 = store s3        vec3 :: Vector Int
             n    = fold max 0 s3      n :: Int
         in (vec3, n)))
```

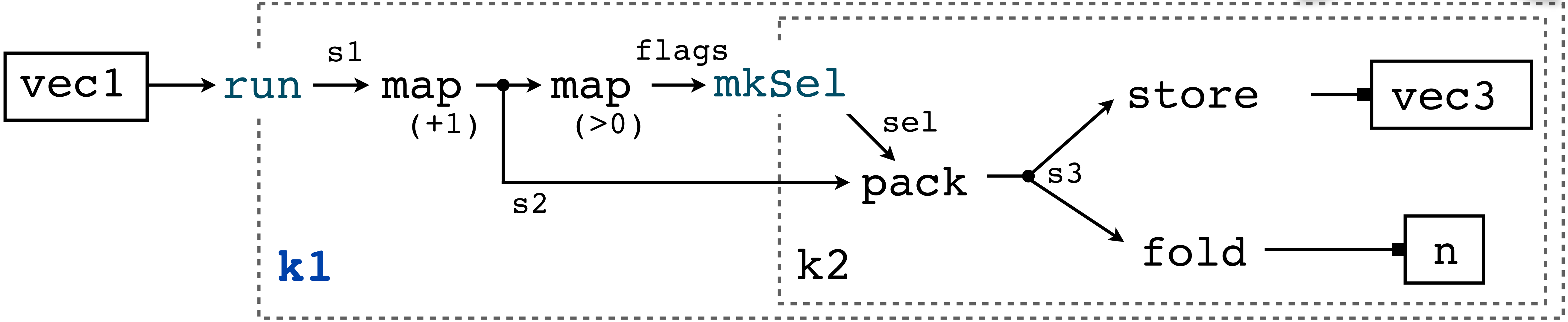


```
run    :: Vector a
       -> (forall k1. Series k1 a -> b) -> b

mkSel  :: Series k1 Bool
       -> (forall k2. Sel k1 k2   -> b) -> b
```



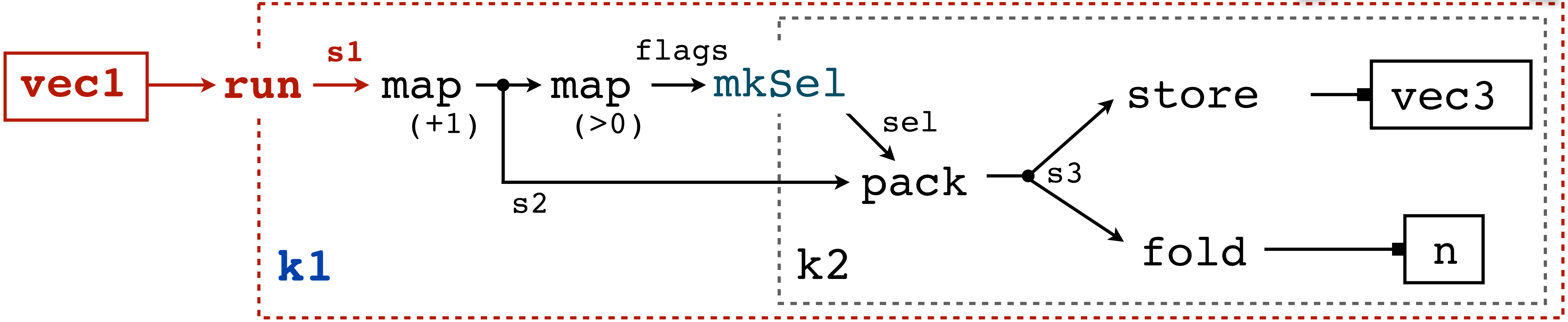
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: ...

     body:  ...


     inner: ...




     end:   ...

   } yields ...
```

k1 >= k2



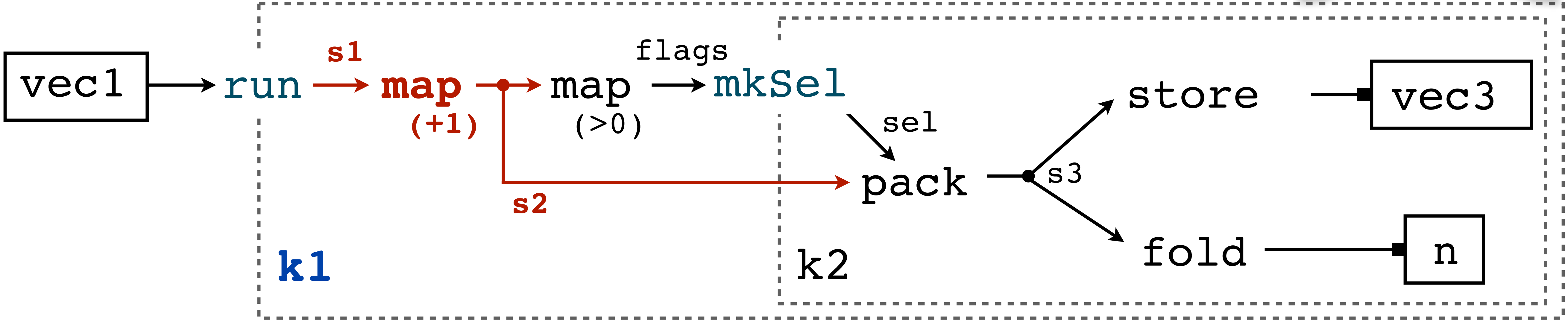
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: ...

     body:  x1   = next k1 vec1


     inner: ...




     end:   ...

   } yields ...
```

k1 >= k2



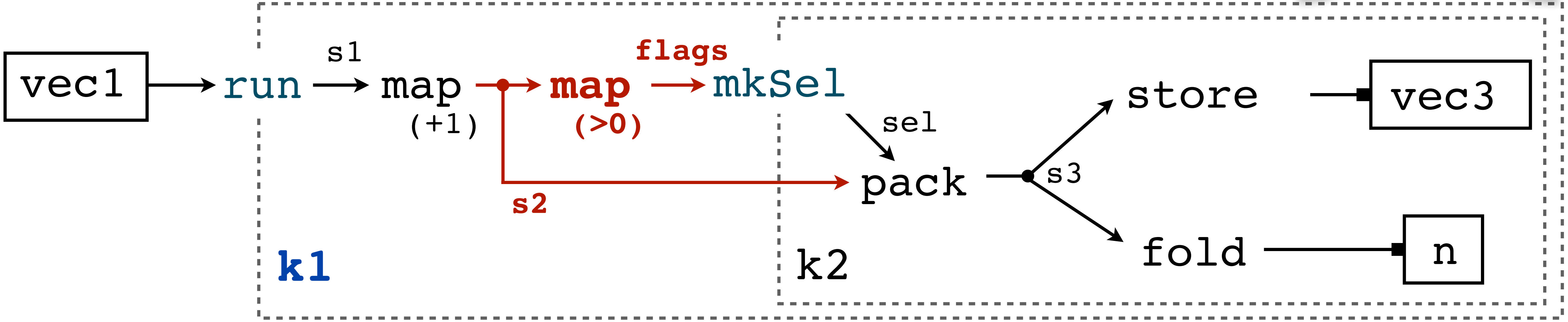
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: ...

     body:  x1   = next k1 vec1
            x2   = (+ 1) x1

     inner: ...




     end:   ...

   } yields ...
```

`k1 >= k2`



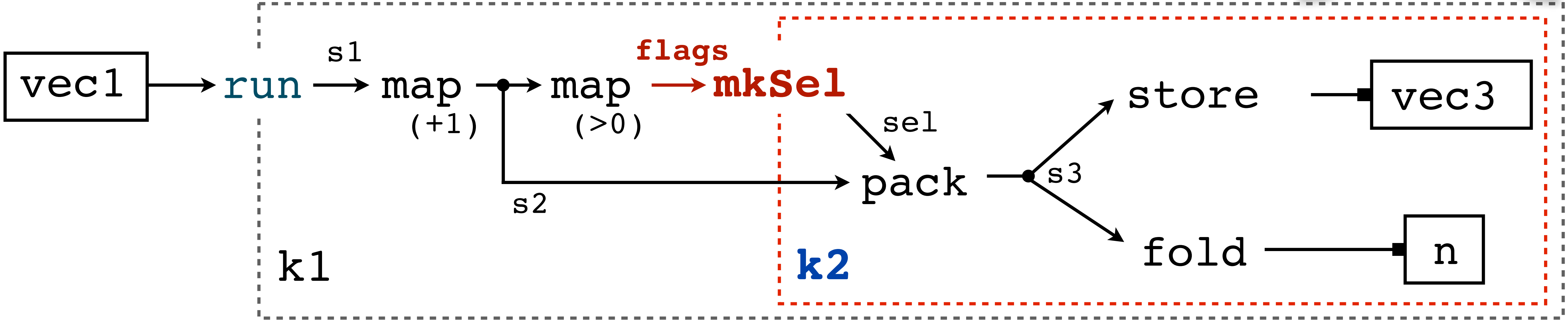
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: ...

     body:  x1   = next k1 vec1
            x2   = (+ 1) x1
            xf   = (> 0) x1
     inner: ...



     end:   ...

   } yields ...
```

k1 >= k2



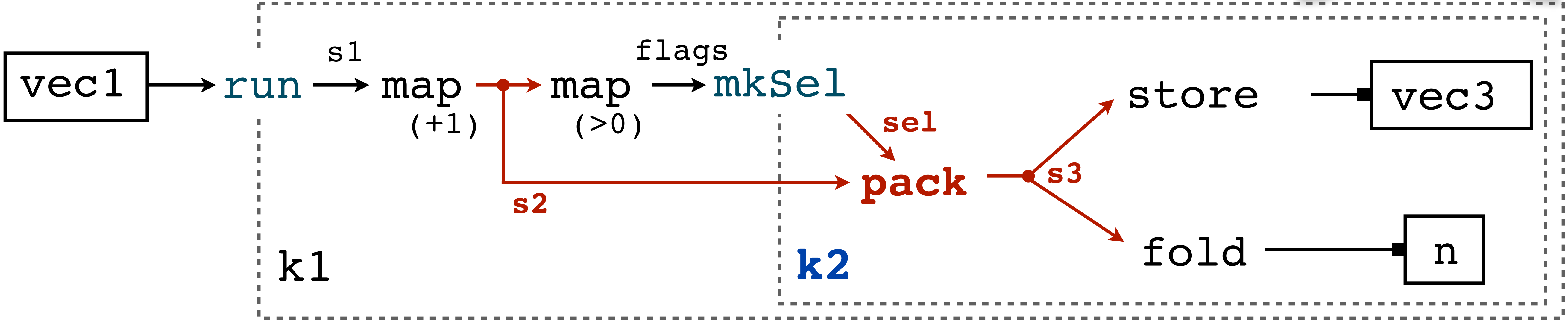
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: ...

     body:  x1   = next k1 vec1
            x2   = (+ 1) x1
            xf   = (> 0) x1
     inner: guard k2 xf
            { body: ...



            }
     end:   ...

   } yields ...
```

k1 >= k2



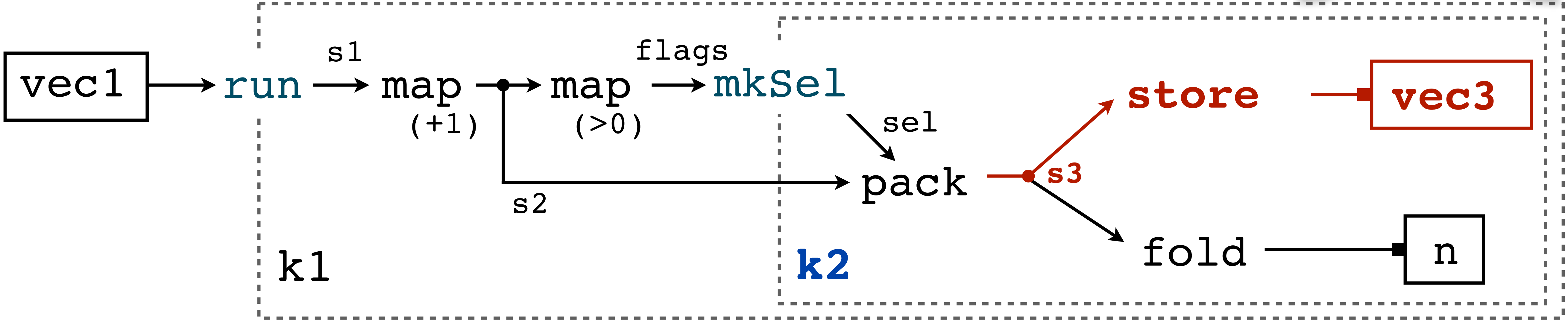
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: ...

     body:  x1   = next k1 vec1
            x2   = (+ 1) x1
            xf   = (> 0) x1
     inner: guard k2 xf
            { body: x3    = x2



            }
     end:   ...

   } yields ...
```

k1 >= k2



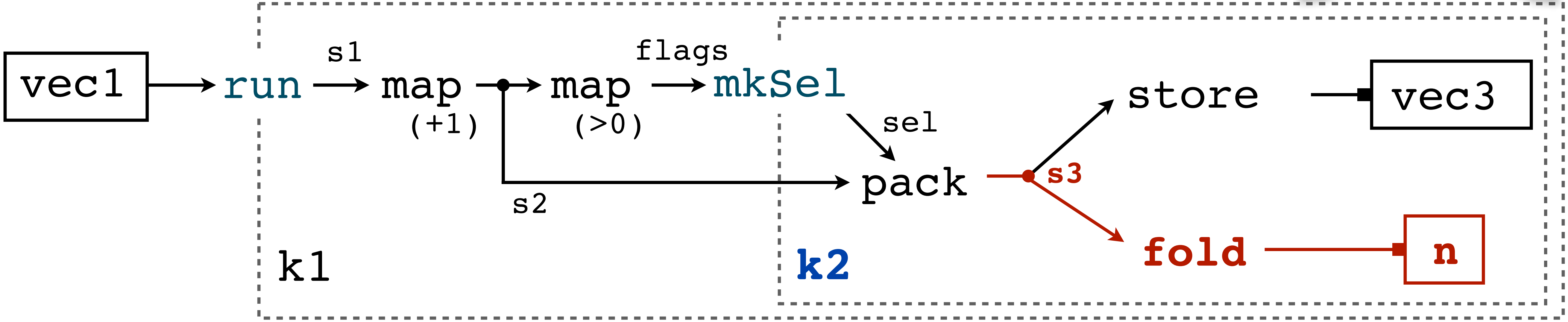
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: vec3 = newVec k2

     body:  x1   = next k1 vec1
            x2   = (+ 1) x1
            xf   = (> 0) x1
     inner: guard k2 xf
            { body: x3    = x2
                    write k2 vec3 x3


            }
     end:   slice k2 vec3

   } yields ...
```

`k1 >= k2`



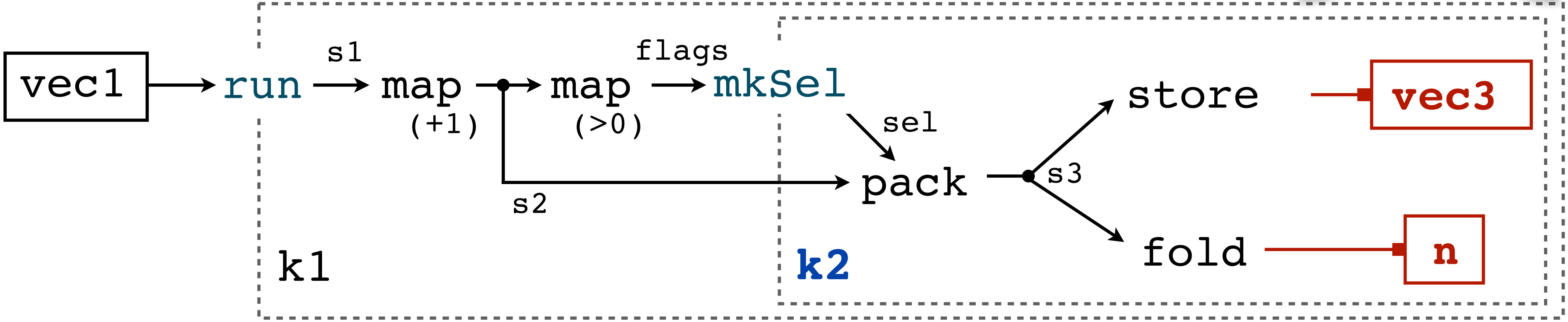
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: vec3 = newVec k2
            nAcc = newAcc 0
     body:  x1   = next k1 vec1
            x2   = (+ 1) x1
            xf   = (> 0) x1
     inner: guard k2 xf
            { body: x3    = x2
                    write k2 vec3 x3
                    nAcc := (+) nAcc x3
            }
     end:   slice k2 vec3
            n    = readAcc nAcc
   } yields ...
```

`k1 >= k2`



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: vec3 = newVec k2
            nAcc = newAcc 0
     body:  x1   = next k1 vec1
            x2   = (+ 1) x1
            xf   = (> 0) x1
     inner: guard k2 xf
            { body: x3    = x2
                    write k2 vec3 x3
                    nAcc := (+) nAcc x3
            }
     end:   slice k2 vec3
            n    = readAcc nAcc
   } yields (vec3, n)
```

k1 >= k2



```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: vec3 = newVec k2
            nAcc = newAcc 0
     body:  x1   = next k1 vec1
            x2   = (+ 1) x1
            xf   = (> 0) x1
     inner: guard k2 xf
            { body: x3    = x2
                    write k2 vec3 x3
                    nAcc := (+) nAcc x3
            }
     end:   slice k2 vec3
            n    = readAcc nAcc
   } yields (vec3, n)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: vec3 = newVec k2
            nAcc = newAcc 0
     body:  x1   = next k1 vec1
            x2   = (+ 1) x1
            xf   = (> 0) x1
     inner: guard k2 xf
            { body: x3    = x2
                    write k2 vec3 x3
                    nAcc := (+) nAcc x3
            }
     end:   slice k2 vec3
            n    = readAcc nAcc
   } yields (vec3, n)
```

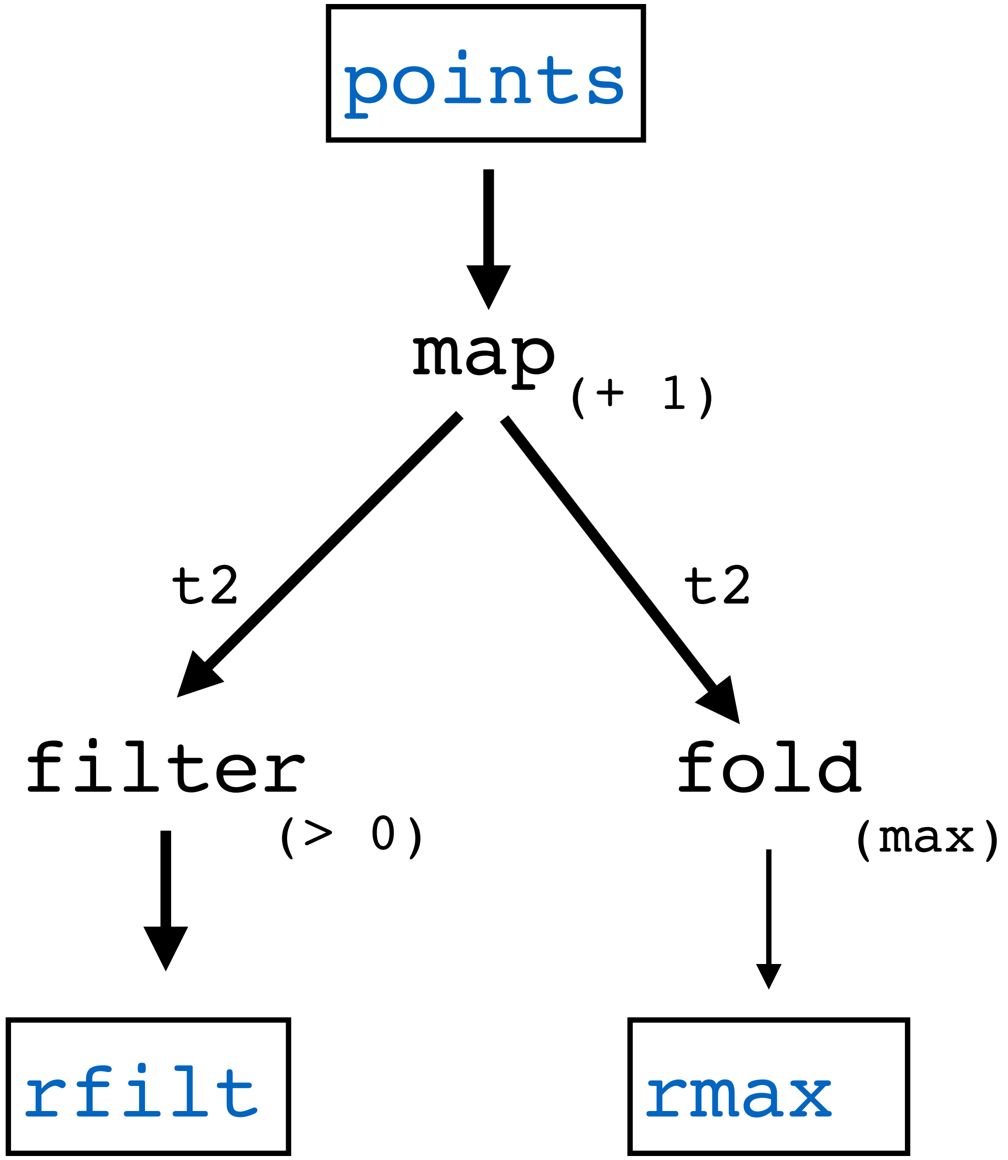
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = loop k1
   { start: vec3 = newVec k2
            nAcc = newAcc 0
     body:  x1   = next k1 vec1
            x2   = (+ 1) x1
            xf   = (> 0) x1
     inner: guard k2 xf
            { body: x3    = x2
                    write k2 vec3 x3
                    nAcc := (+) nAcc x3
            }
     end:   slice k2 vec3
            n    = readAcc nAcc
   } yields (vec3, n)
```

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = do vec3  <- newVec (length vec1)
      nAcc  <- newAcc 0
      k2Acc <- newAcc 0
      loopM (length vec1)
       (\ix1 ->
         do x1      <- next ix1 vec1
            let x2 = (+ 1) x1
            let xf = (> 0) x1
            guardM k2Acc xf (\ix2 ->
             do let x3 = x2
                write ix2 vec3 x3
                modifyAcc nAcc (\x -> (+) x x3))
     sliceVec k2Acc vec3
     n      <- readAcc nAcc
     return (vec3, n)
```

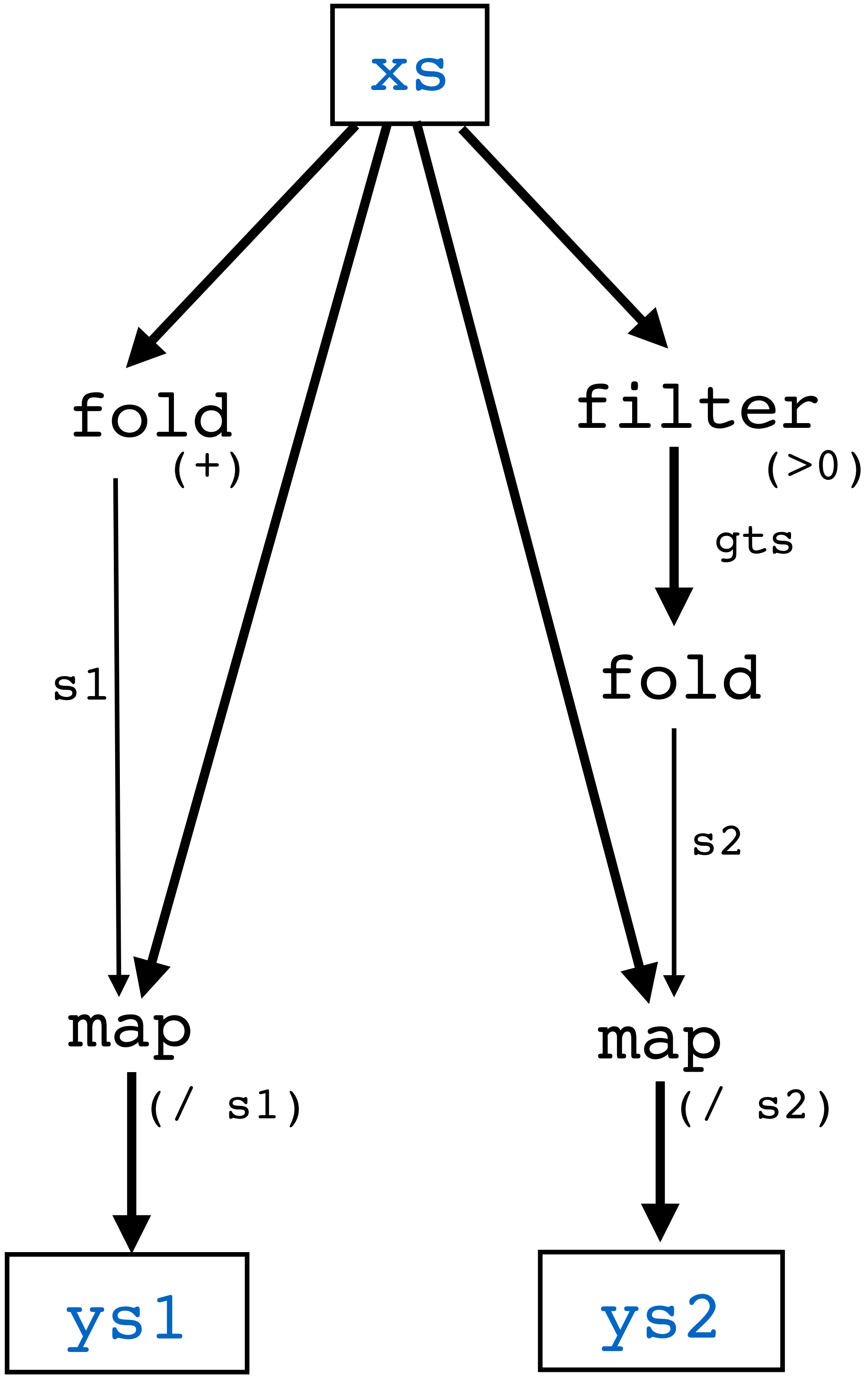
Operator Clustering

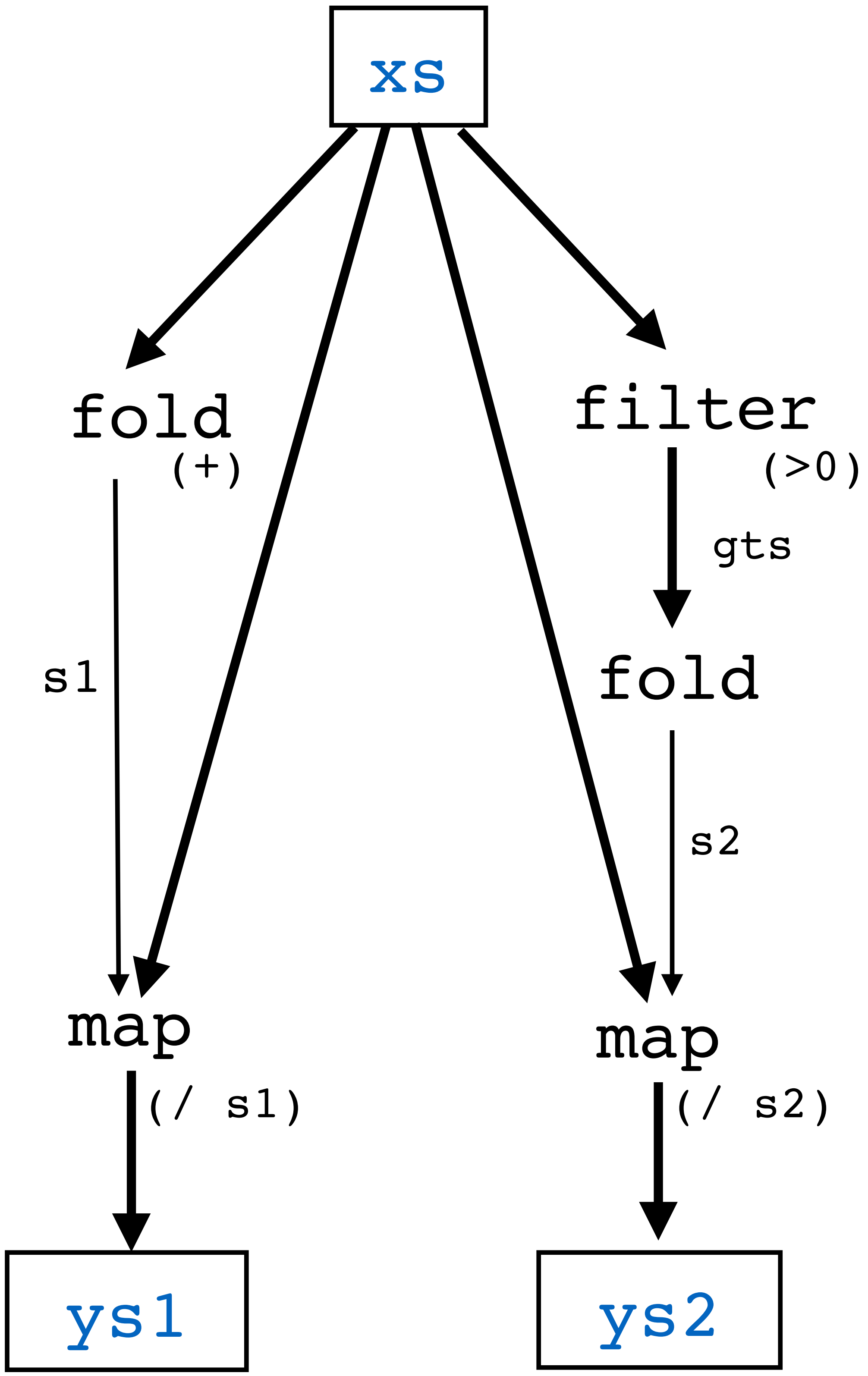
```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax points
 = let t2    = map (+ 1) points
       rfilt = filter (> 0) t2
       rmax  = fold max 0 rfilt
   in  (rfilt, rmax)
```

```
t2    = map (+ 1) points
rfilt = filter (> 0) t2
rmax  = fold max 0 rfilt
```



```
s1   = fold   (+) 0 xs
gts  = filter (> 0) xs
s2   = fold   (+) 0 gts
ys1  = map    (/ sum1) xs
ys2  = map    (/ sum2) xs
```





xs

fold

(+)

s1

map

(/ s1)

ys1

filter

(>0)

gts

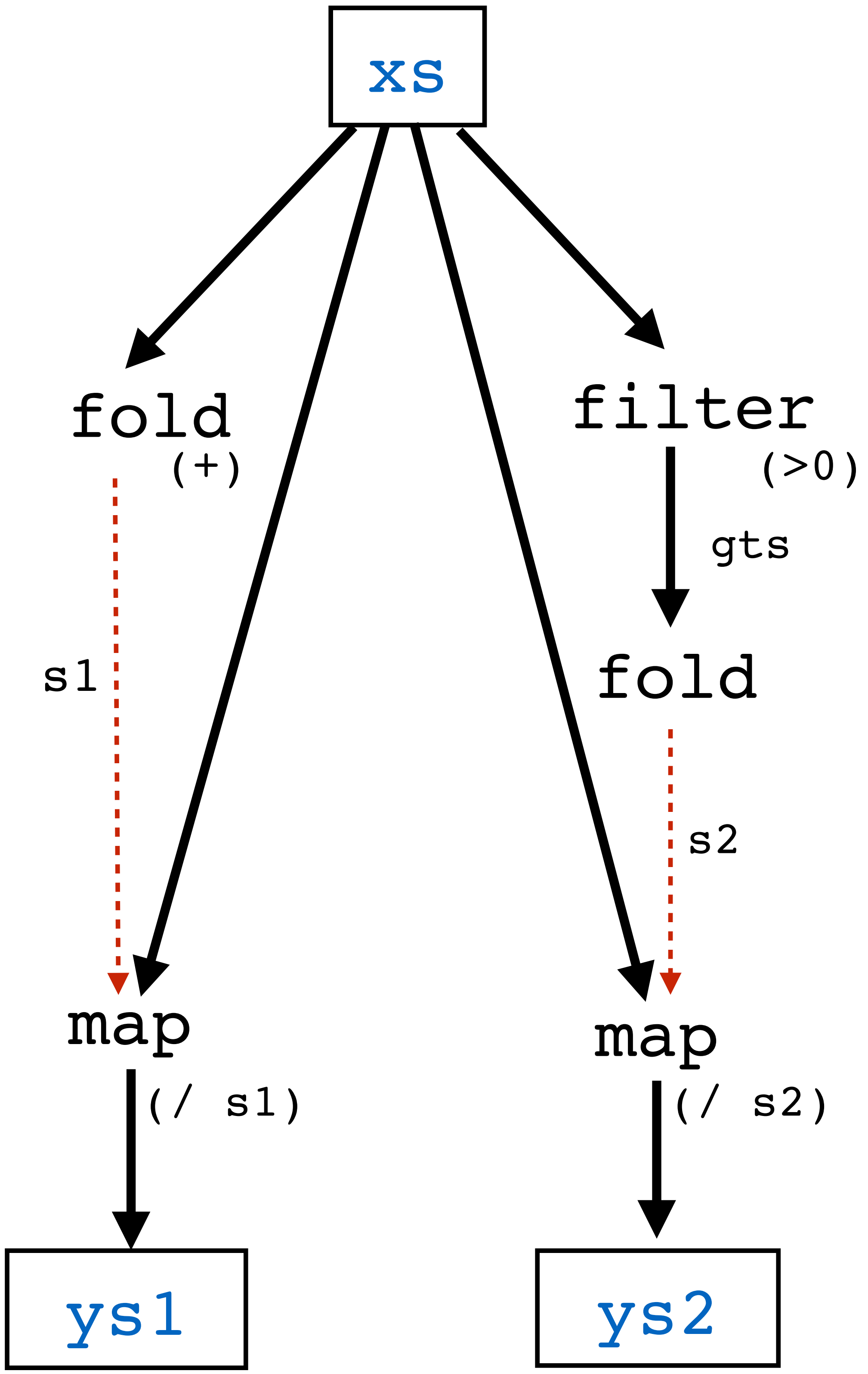
fold

s2

map

(/ s2)

ys2



xs

fold

(+)

s1

filter

(>0)

gts

fold

s2

map

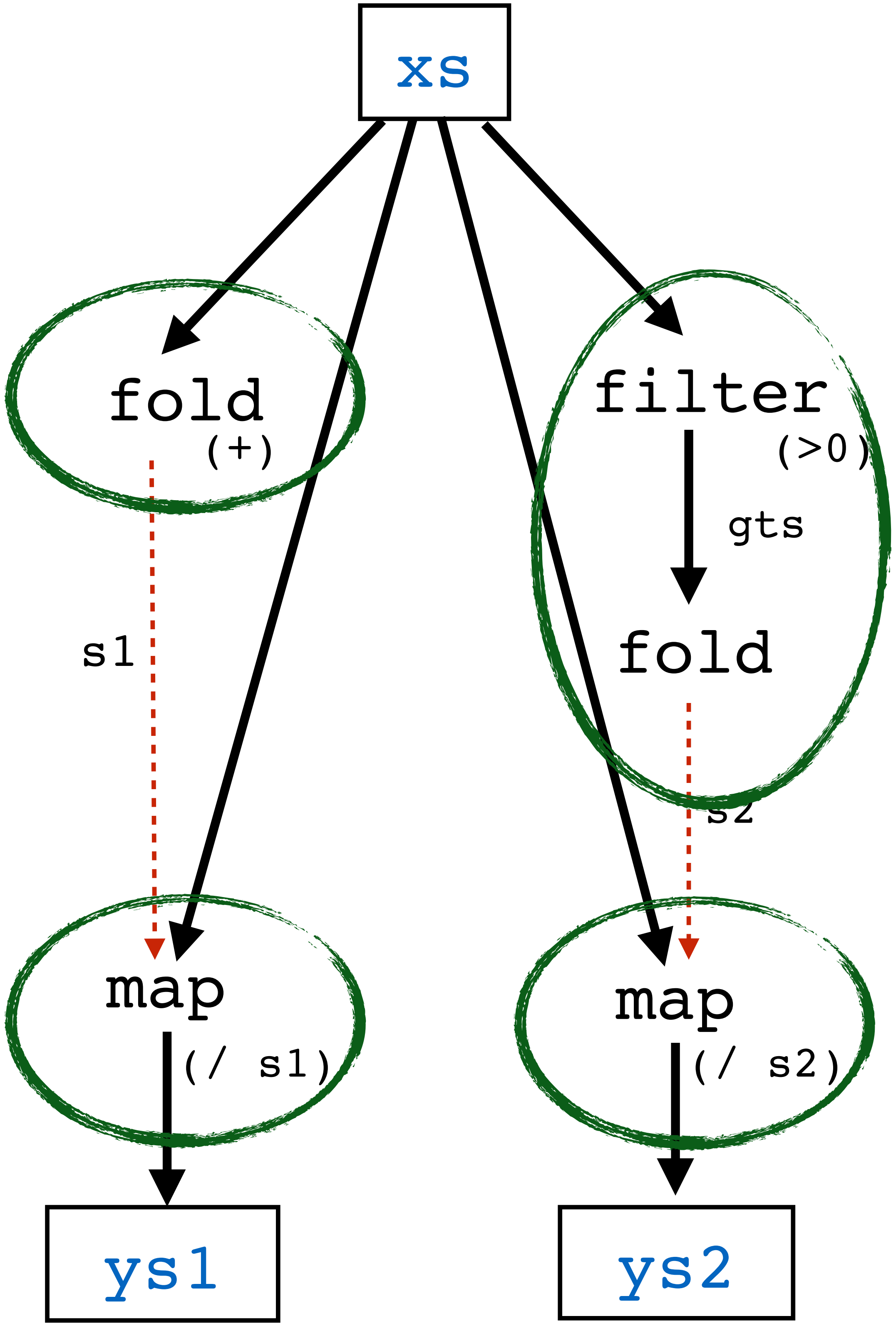
(/ s1)

map

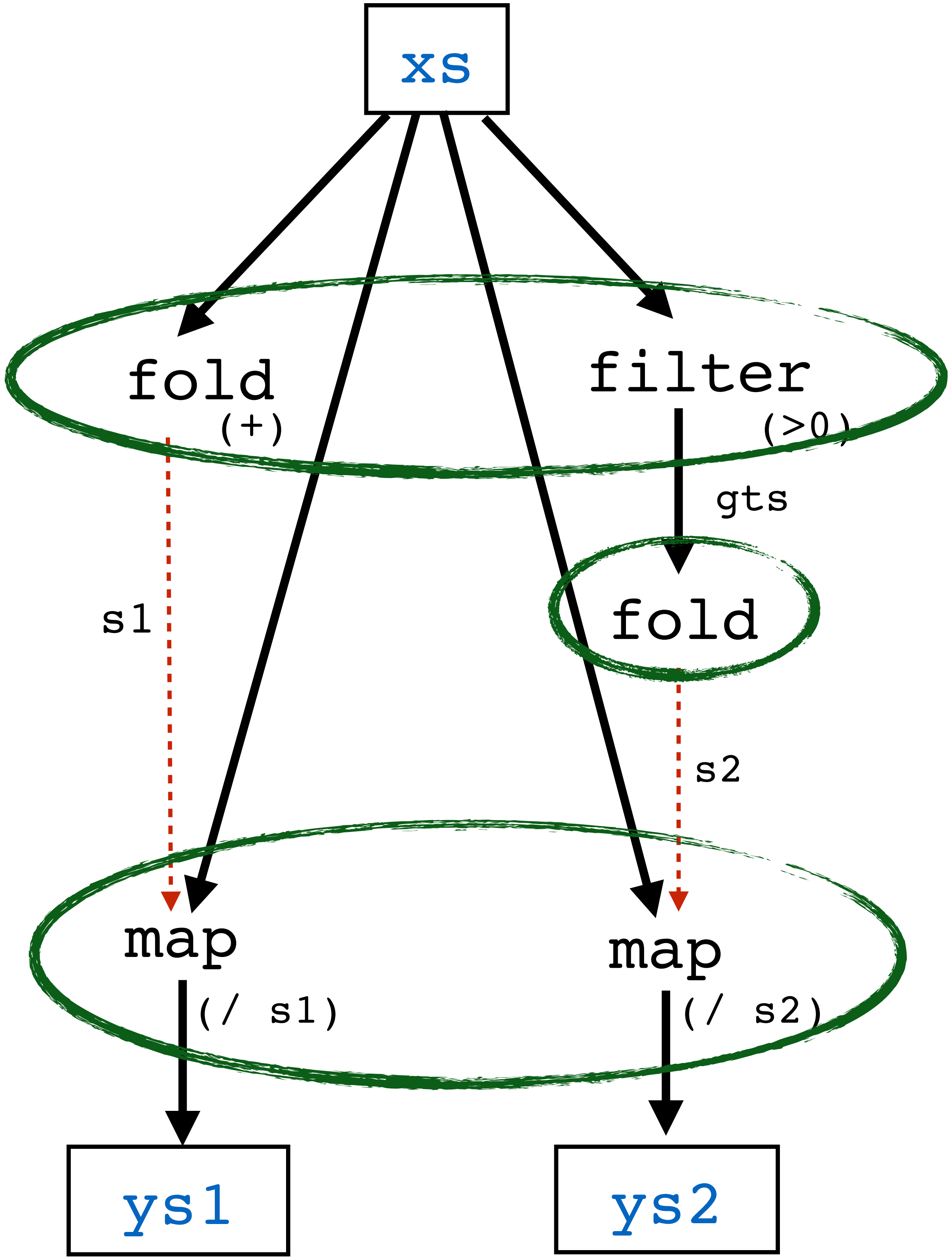
(/ s2)

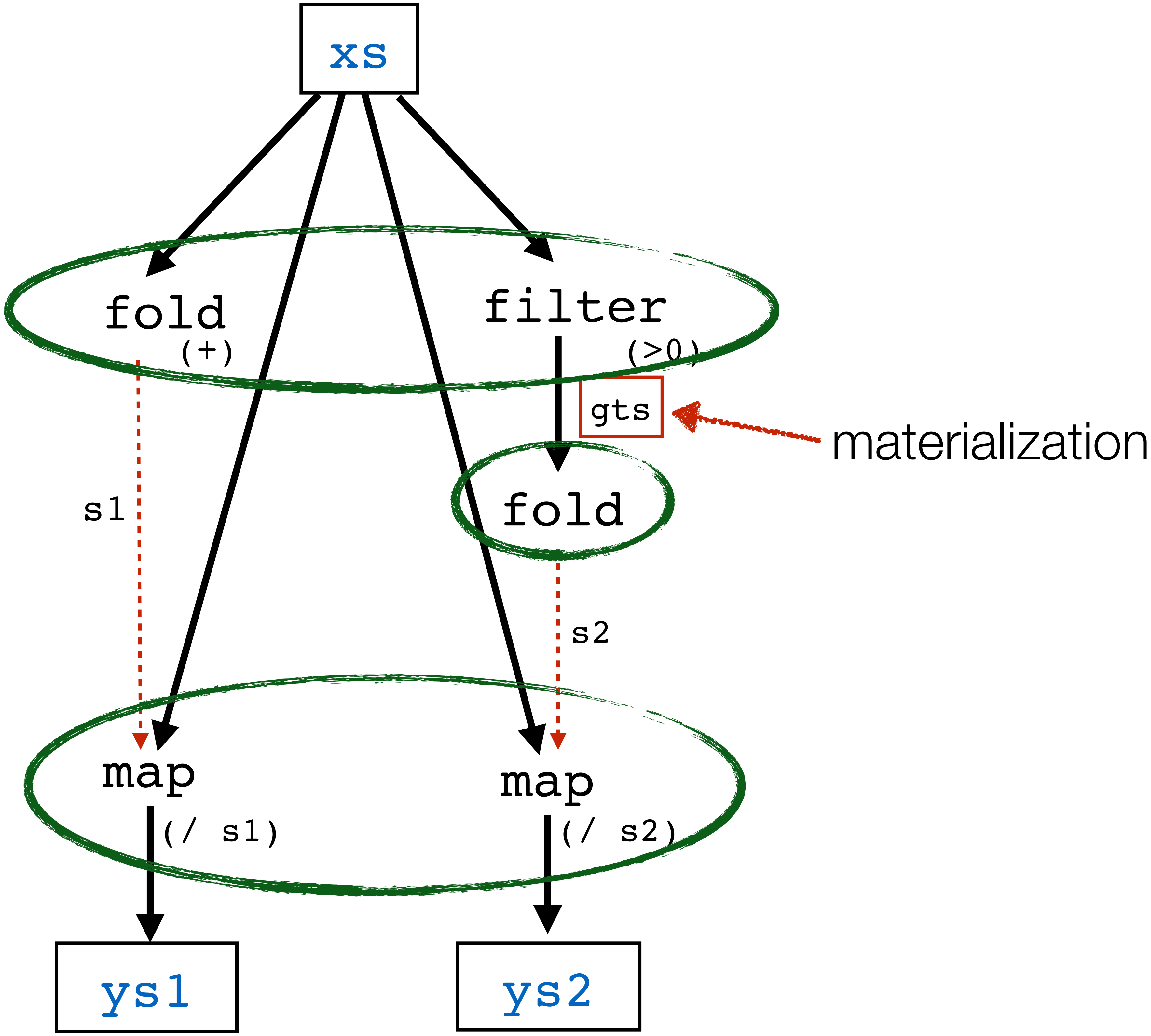
ys1

ys2



(what you'd get with stream fusion)


xs
fold
(+)
filter
(>0)
gts
s1
fold
s2
map
(/ s1)
map
(/ s2)
ys1
ys2



`xs`

`fold` `(+)`

`filter` `(>0)`

`gts`

materialization

`s1`

`fold`

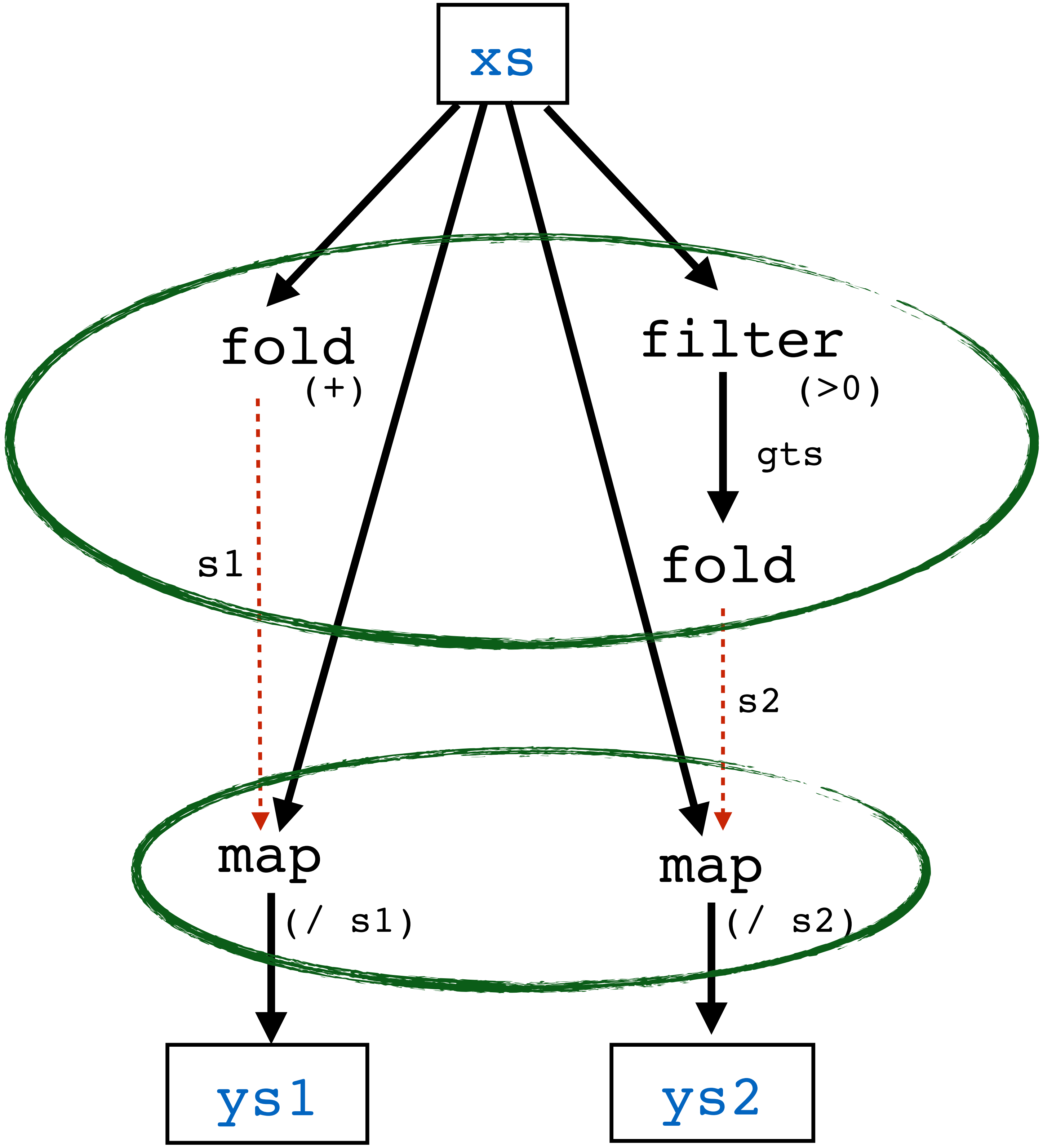
`s2`

`map` `(/ s1)`

`map` `(/ s2)`

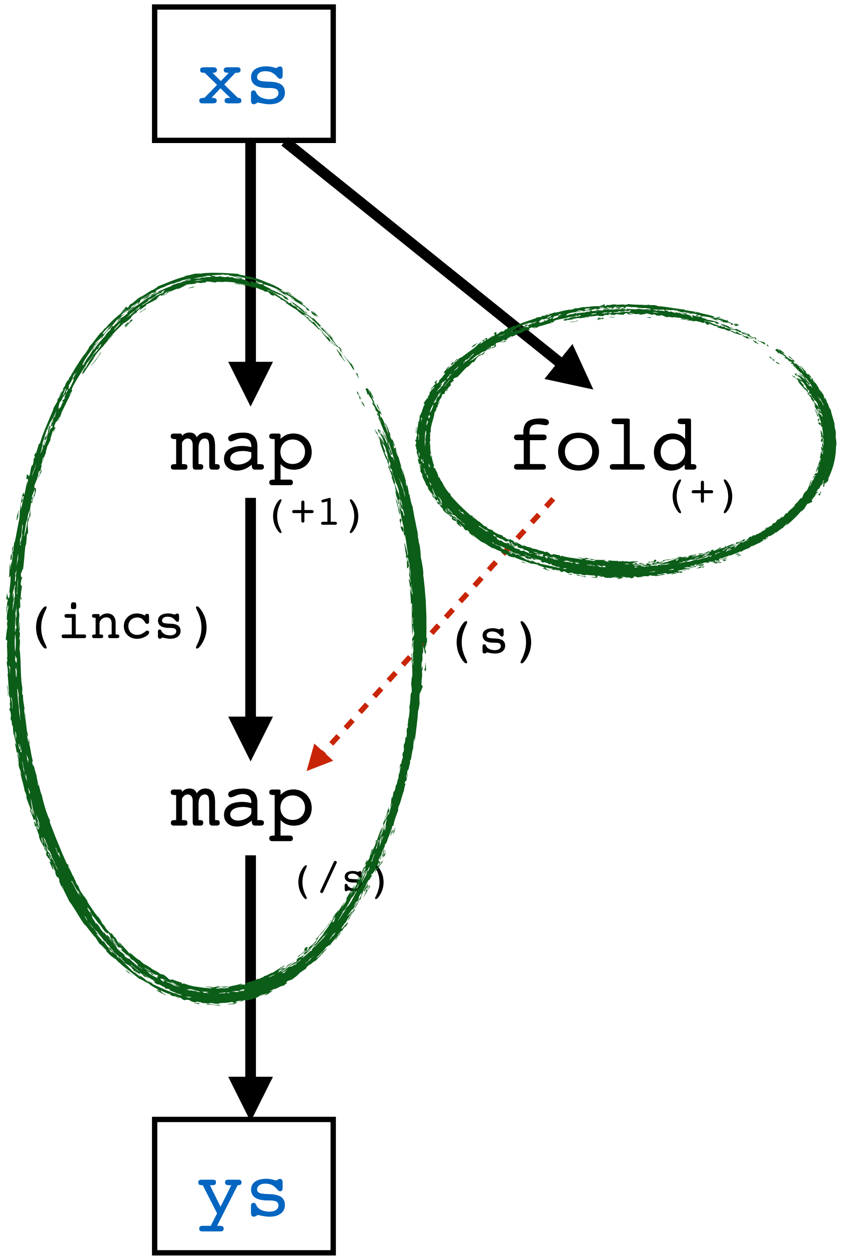
`ys1`

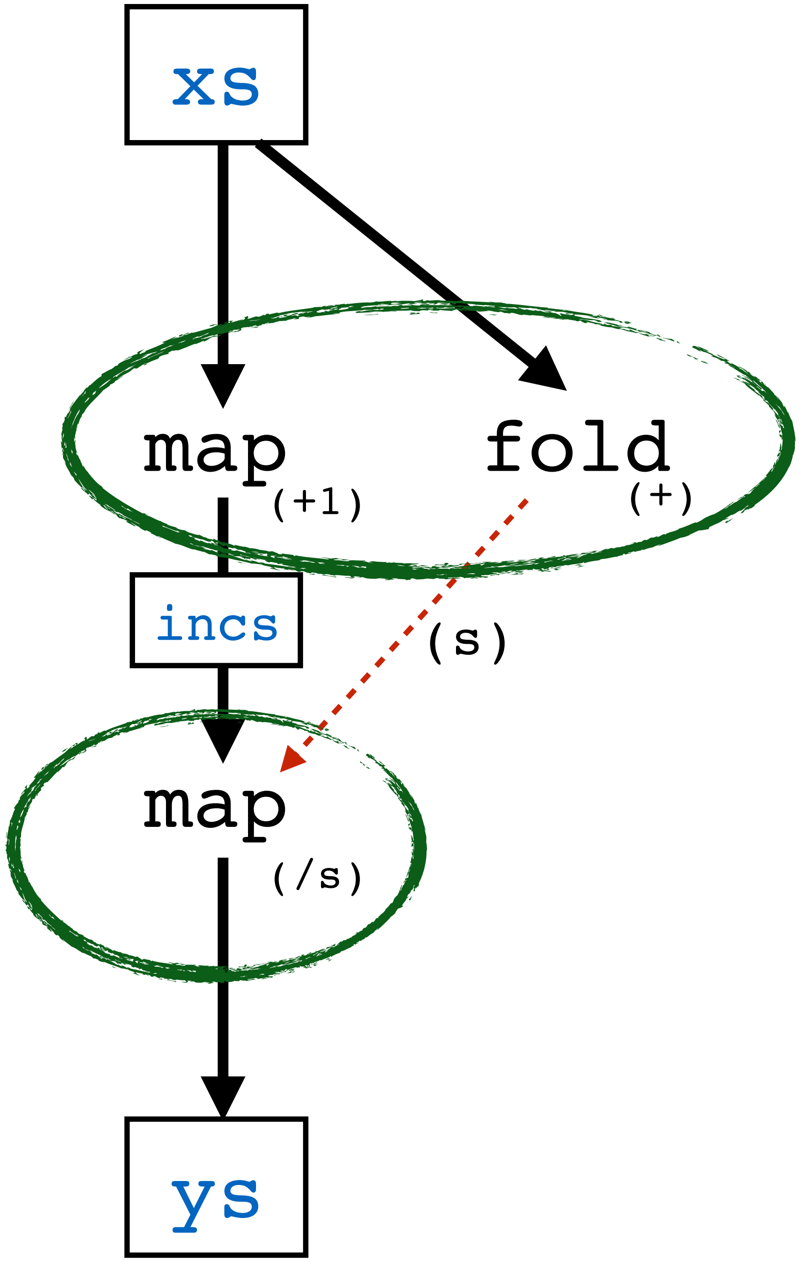
`ys2`


xs
fold
(+)
filter
(>0)
gts
fold
s1
s2
map
(/ s1)
map
(/ s2)
ys1
ys2

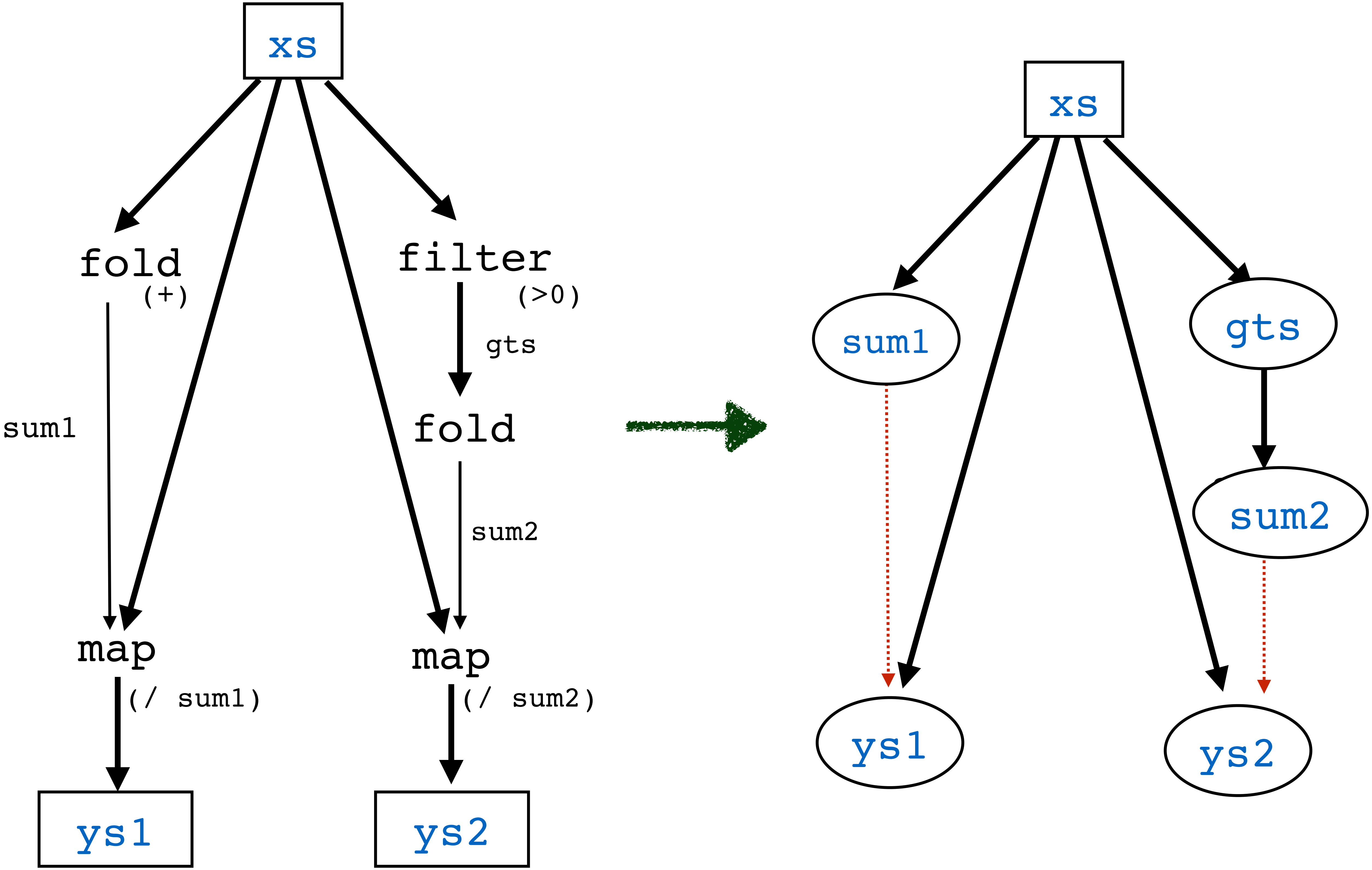
Minimize #access rather than #clusters

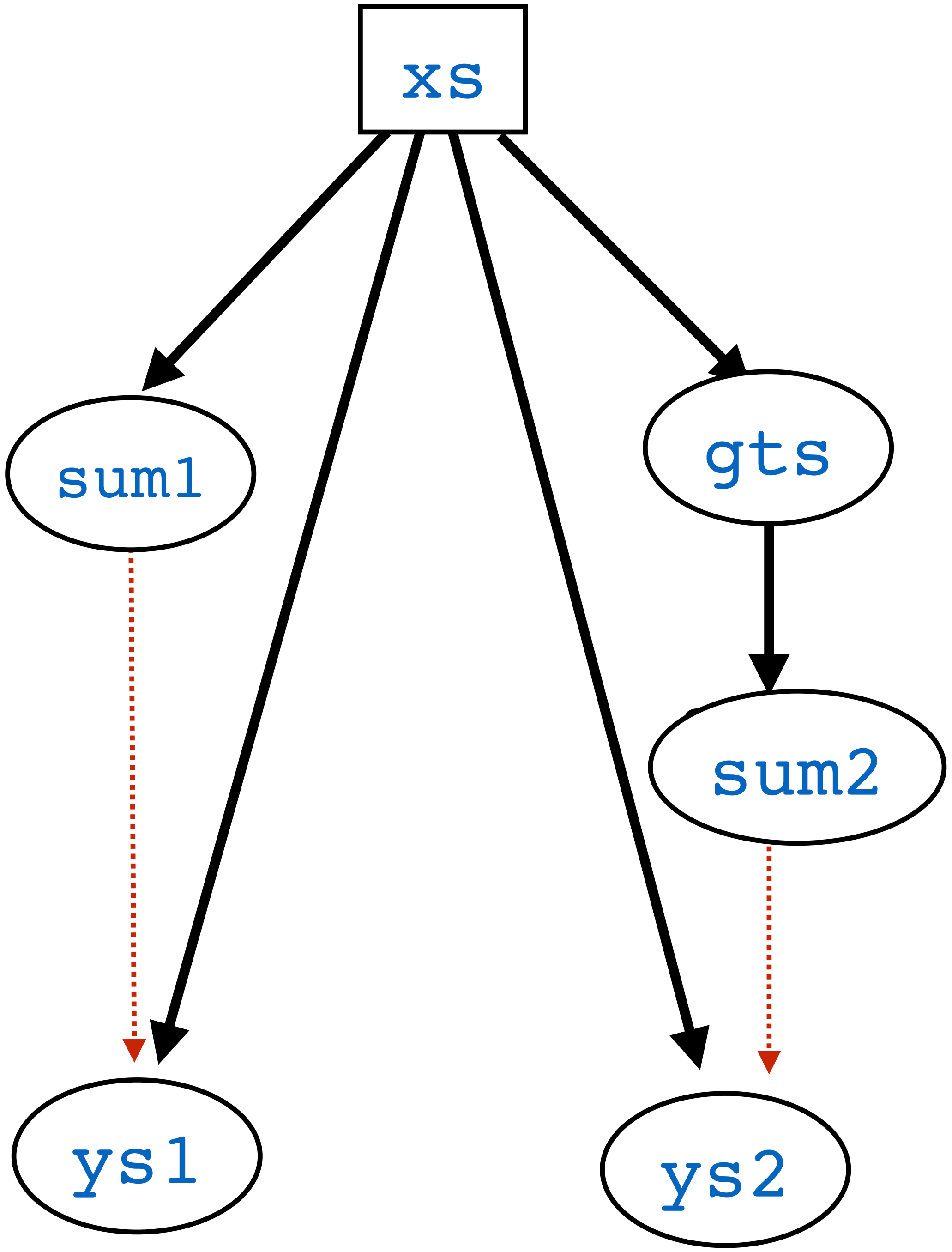
```
normalizeInc :: Vector Int -> Vector Int
normalizeInc xs
  = let incs = map  (+1)    xs
        s    = fold (+) 0   xs
        ys   = map  (/ s) incs
     in ys
```



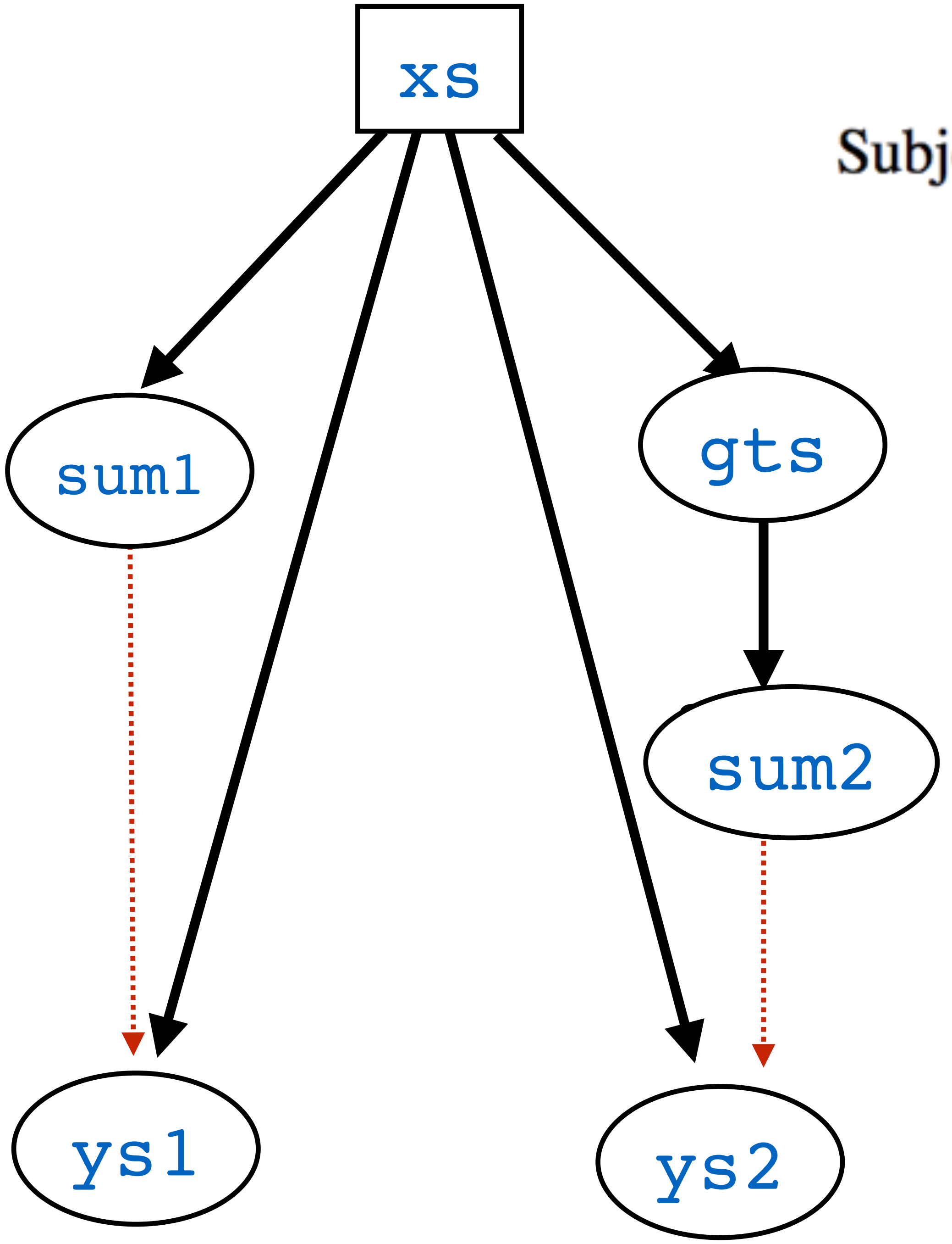


Integer Linear Programming (ILP) Formulation


xs
fold
(+)
sum1
map
(/ sum1)
ys1
filter
(>0)
gts
fold
sum2
map
(/ sum2)
ys2
xs
sum1
gts
sum2
ys1
ys2

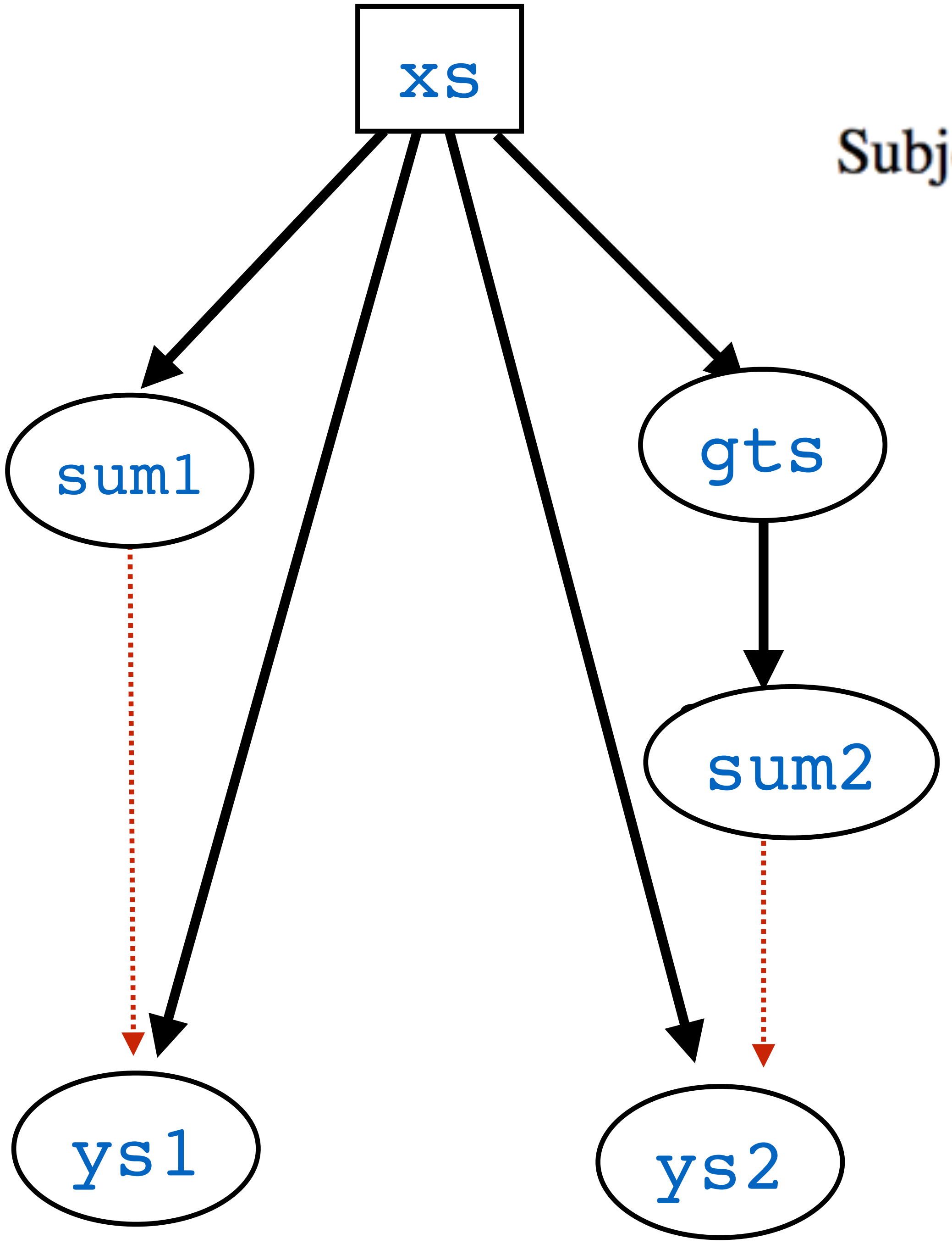


$$\text{Minimise}\quad \begin{aligned}&25\cdot x_{sum1,gts}+1\cdot x_{sum1,sum2}+25\cdot x_{sum1,ys2}+\\&25\cdot x_{gts,sum2}+25\cdot x_{gts,ys1}+1\cdot x_{sum2,ys1}+\\&25\cdot x_{ys1,ys2}+5\cdot c_{gts}+5\cdot c_{ys1}+5\cdot c_{ys2}\end{aligned}$$



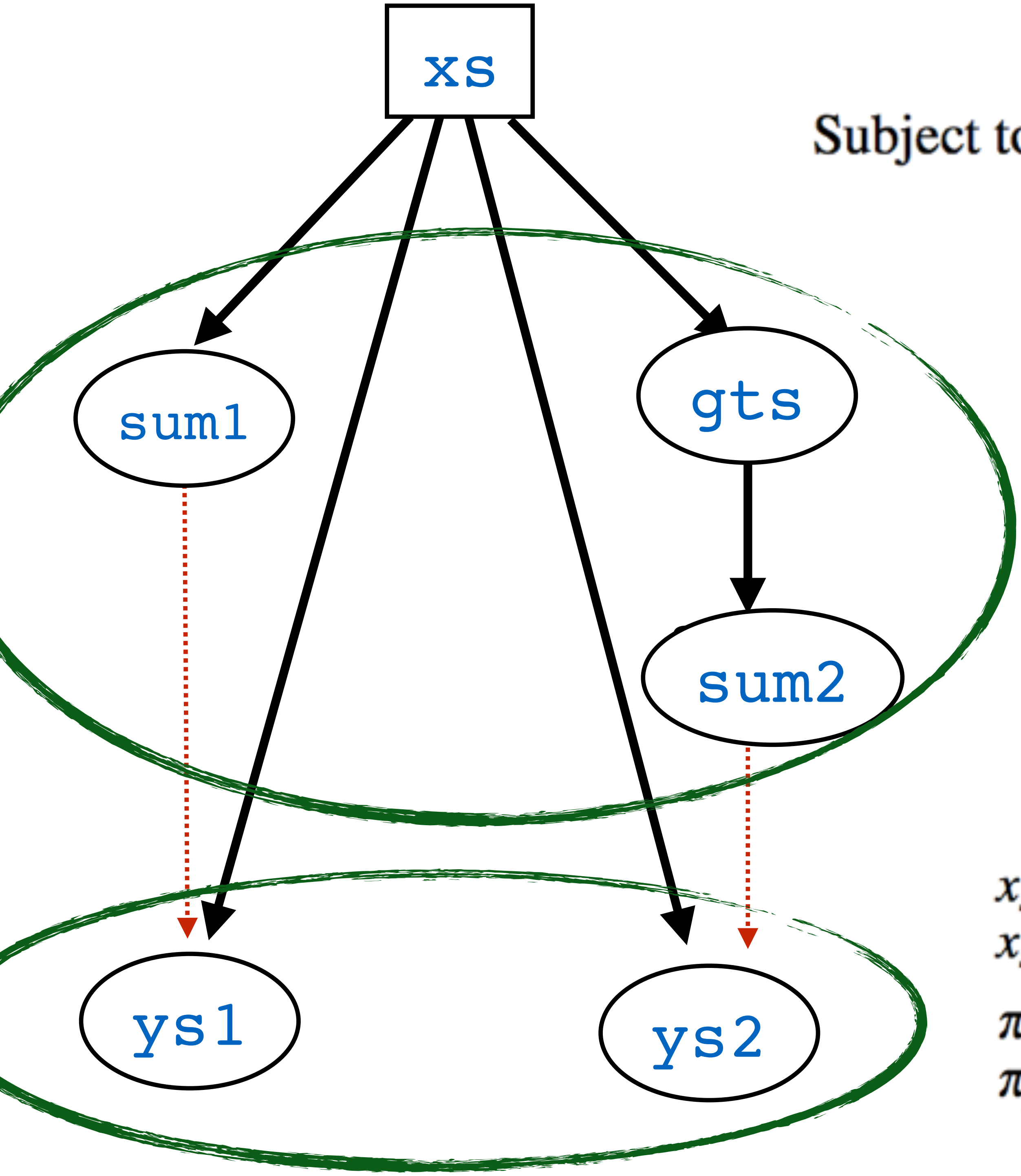
$$
\begin{array}{ll}
\text{Minimise} & 25 \cdot x_{sum1,gts} + 1 \cdot x_{sum1,sum2} + 25 \cdot x_{sum1,ys2} + \\
 & 25 \cdot x_{gts,sum2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} + \\
 & 25 \cdot x_{ys1,ys2} + 5 \cdot c_{gts} + 5 \cdot c_{ys1} + 5 \cdot c_{ys2}
\end{array}
$$

$$
\begin{array}{lrcccl}
\text{Subject to} & -5 \cdot x_{sum1,gts} & \leq & \pi_{gts} - \pi_{sum1} & \leq & 5 \cdot x_{sum1,gts} \\
 & -5 \cdot x_{sum1,sum2} & \leq & \pi_{sum2} - \pi_{sum1} & \leq & 5 \cdot x_{sum1,sum2} \\
 & -5 \cdot x_{sum1,ys2} & \leq & \pi_{ys2} - \pi_{sum1} & \leq & 5 \cdot x_{sum1,ys2} \\
 & -5 \cdot x_{gts,ys1} & \leq & \pi_{ys1} - \pi_{gts} & \leq & 5 \cdot x_{gts,ys1} \\
 & -5 \cdot x_{sum2,ys1} & \leq & \pi_{ys1} - \pi_{sum2} & \leq & 5 \cdot x_{sum2,ys1} \\
 & -5 \cdot x_{ys1,ys2} & \leq & \pi_{ys2} - \pi_{ys1} & \leq & 5 \cdot x_{ys1,ys2}
\end{array}
$$



$$
\begin{array}{ll}
\text{Minimise} & 25 \cdot x_{sum1,gts} + 1 \cdot x_{sum1,sum2} + 25 \cdot x_{sum1,ys2} + \\
& 25 \cdot x_{gts,sum2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} + \\
& 25 \cdot x_{ys1,ys2} + 5 \cdot c_{gts} + 5 \cdot c_{ys1} + 5 \cdot c_{ys2}
\end{array}
$$

$$
\begin{array}{llll}
\text{Subject to} & -5 \cdot x_{sum1,gts} & \leq \pi_{gts} - \pi_{sum1} & \leq 5 \cdot x_{sum1,gts} \\
& -5 \cdot x_{sum1,sum2} & \leq \pi_{sum2} - \pi_{sum1} & \leq 5 \cdot x_{sum1,sum2} \\
& -5 \cdot x_{sum1,ys2} & \leq \pi_{ys2} - \pi_{sum1} & \leq 5 \cdot x_{sum1,ys2} \\
& -5 \cdot x_{gts,ys1} & \leq \pi_{ys1} - \pi_{gts} & \leq 5 \cdot x_{gts,ys1} \\
& -5 \cdot x_{sum2,ys1} & \leq \pi_{ys1} - \pi_{sum2} & \leq 5 \cdot x_{sum2,ys1} \\
& -5 \cdot x_{ys1,ys2} & \leq \pi_{ys2} - \pi_{ys1} & \leq 5 \cdot x_{ys1,ys2} \\
& x_{gts,sum2} & \leq \pi_{sum2} - \pi_{gts} & \leq 5 \cdot x_{gts,sum2} \\
& & \pi_{sum1} < \pi_{ys1} & \\
& & \pi_{sum2} < \pi_{ys2} &
\end{array}
$$

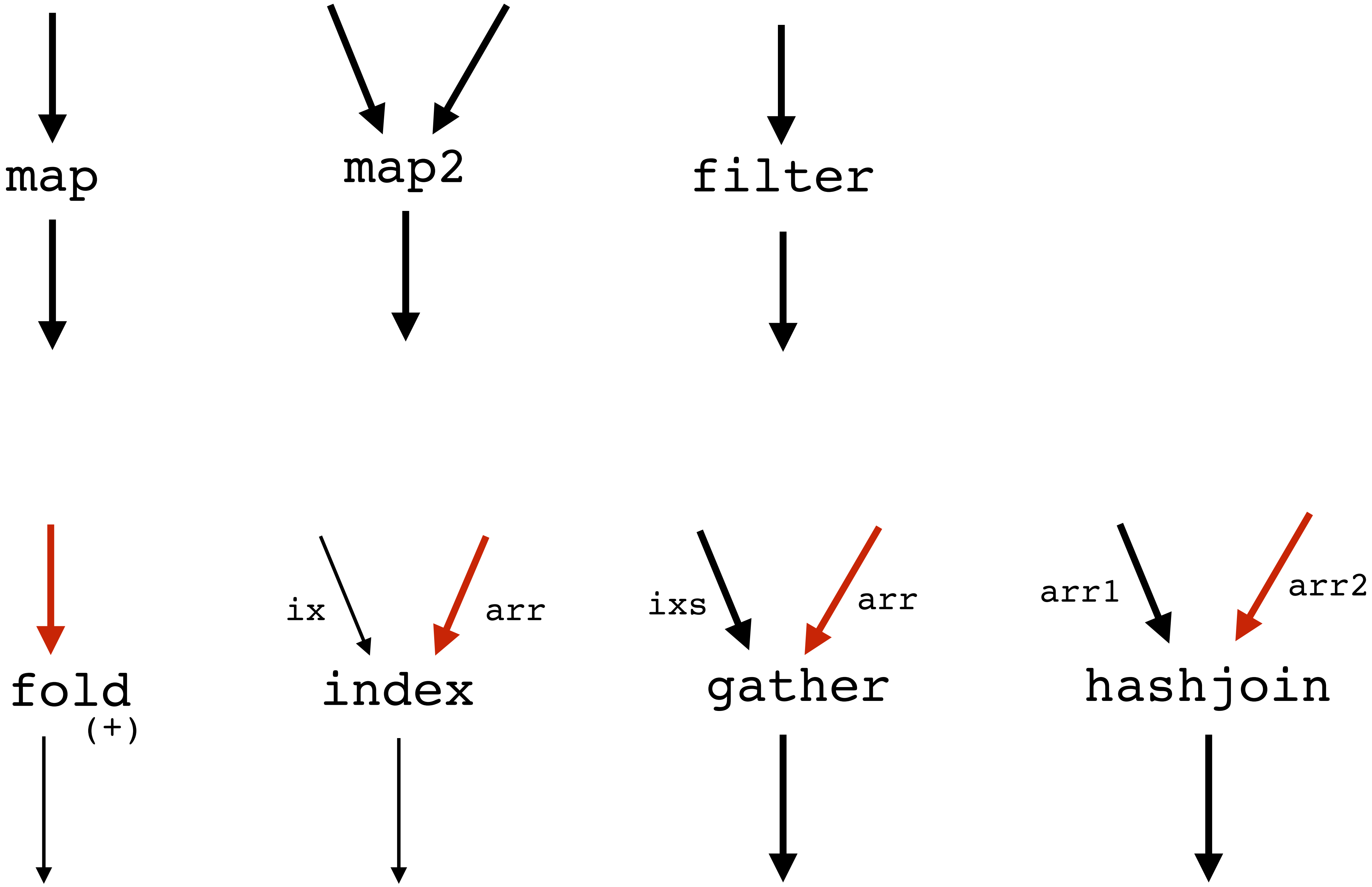


$$\begin{array}{ll}
\text{Minimise} & 25 \cdot x_{sum1,gts} + 1 \cdot x_{sum1,sum2} + 25 \cdot x_{sum1,ys2} + \\
 & 25 \cdot x_{gts,sum2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} + \\
 & 25 \cdot x_{ys1,ys2} + 5 \cdot c_{gts} + 5 \cdot c_{ys1} + 5 \cdot c_{ys2}
\end{array}$$

$$\begin{array}{lllll}
\text{Subject to} & -5 \cdot x_{sum1,gts} & \leq \pi_{gts} - \pi_{sum1} & \leq 5 \cdot x_{sum1,gts} \\
 & -5 \cdot x_{sum1,sum2} & \leq \pi_{sum2} - \pi_{sum1} & \leq 5 \cdot x_{sum1,sum2} \\
 & -5 \cdot x_{sum1,ys2} & \leq \pi_{ys2} - \pi_{sum1} & \leq 5 \cdot x_{sum1,ys2} \\
 & -5 \cdot x_{gts,ys1} & \leq \pi_{ys1} - \pi_{gts} & \leq 5 \cdot x_{gts,ys1} \\
 & -5 \cdot x_{sum2,ys1} & \leq \pi_{ys1} - \pi_{sum2} & \leq 5 \cdot x_{sum2,ys1} \\
 & -5 \cdot x_{ys1,ys2} & \leq \pi_{ys2} - \pi_{ys1} & \leq 5 \cdot x_{ys1,ys2} \\
 & x_{gts,sum2} & \leq \pi_{sum2} - \pi_{gts} & \leq 5 \cdot x_{gts,sum2} \\
 & & \pi_{sum1} < \pi_{ys1} & \\
 & & \pi_{sum2} < \pi_{ys2} &
\end{array}$$

$$\begin{array}{ll}
x_{sum1,gts},\ x_{sum1,sum1},\ x_{sum1,sum2},\ x_{gts,sum2},\ x_{ys1,ys2} & = 0 \\
x_{sum1,ys2},\ x_{gts,ys1},\ x_{sum2,ys1} & = 1 \\
 & = 0 \\
\pi_{sum1},\ \pi_{gts},\ \pi_{sum2} & = 1 \\
\pi_{ys1},\ \pi_{ys2} & = 0
\end{array}$$

Fusion Barriers



map

map2

filter

fold

(+)

ix

index

arr

ixs

gather

arr

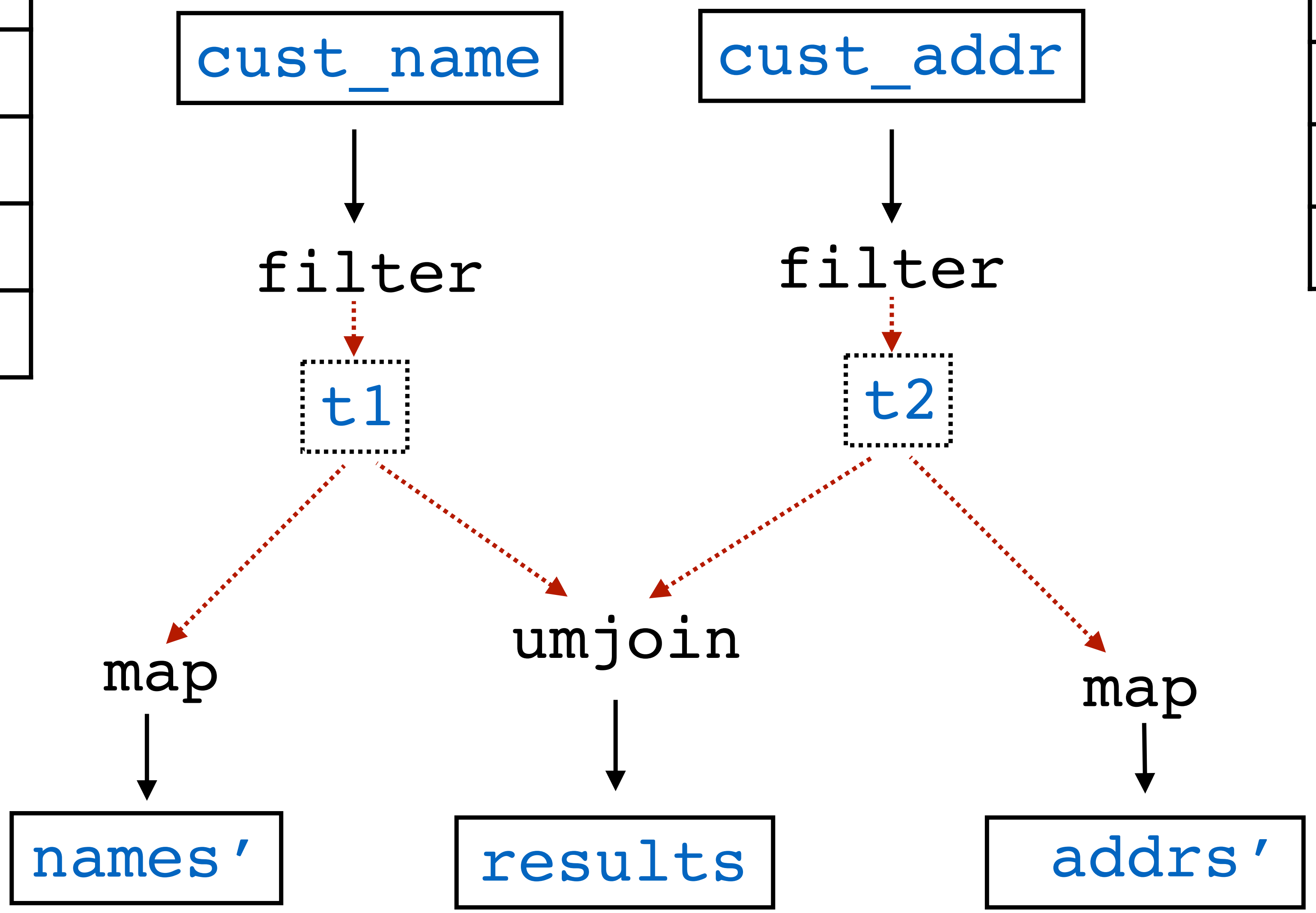
arr1

hashjoin

arr2

Merge Joins

c1	Bob
c4	Alice
c5	John
c6	Rob
c9	Zoe



cust_name

cust_addr

filter

filter

t1

t2

map

umjoin

map

names’

results

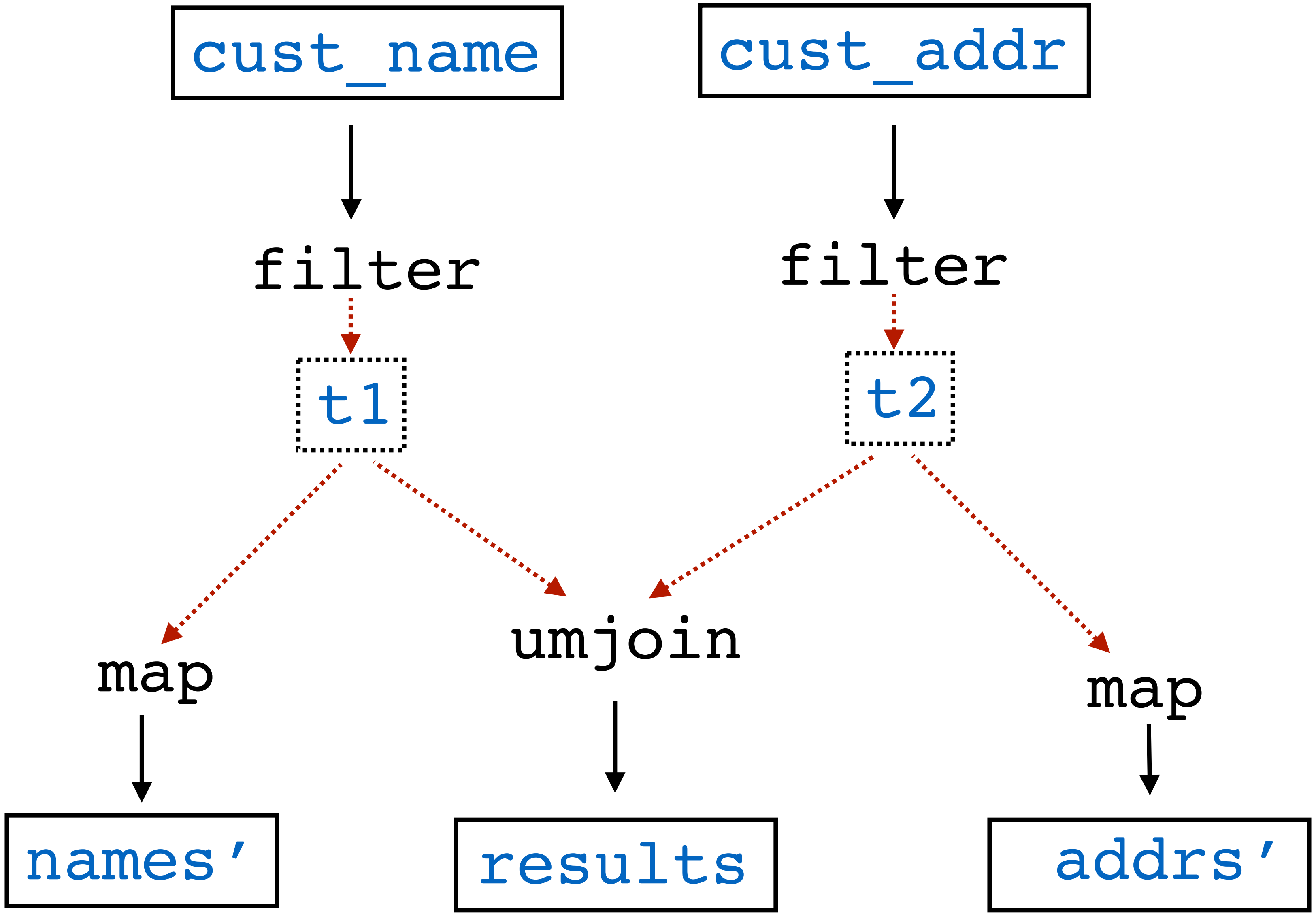
addrs’

c1	Manly
c5	Redfern
c6	Bondi
c9	Penrith

c1	Bob
c5	John
c6	Rob
c9	Zoe

c1	Bob	Manly
c6	Rob	Bondi

c1	Manly
c6	Bondi



cust_name

cust_addr

filter

filter

t1

t2

map

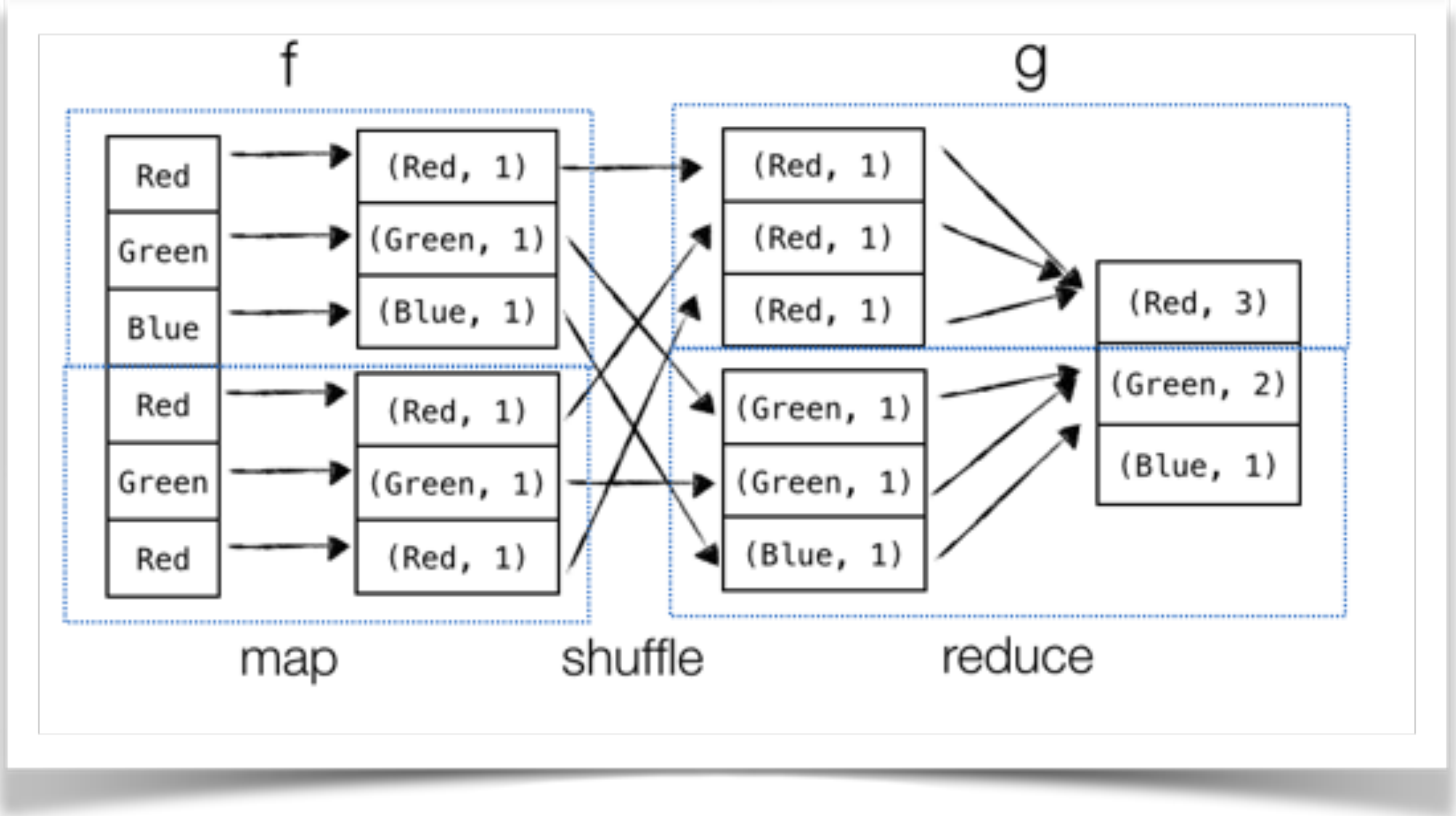
umjoin

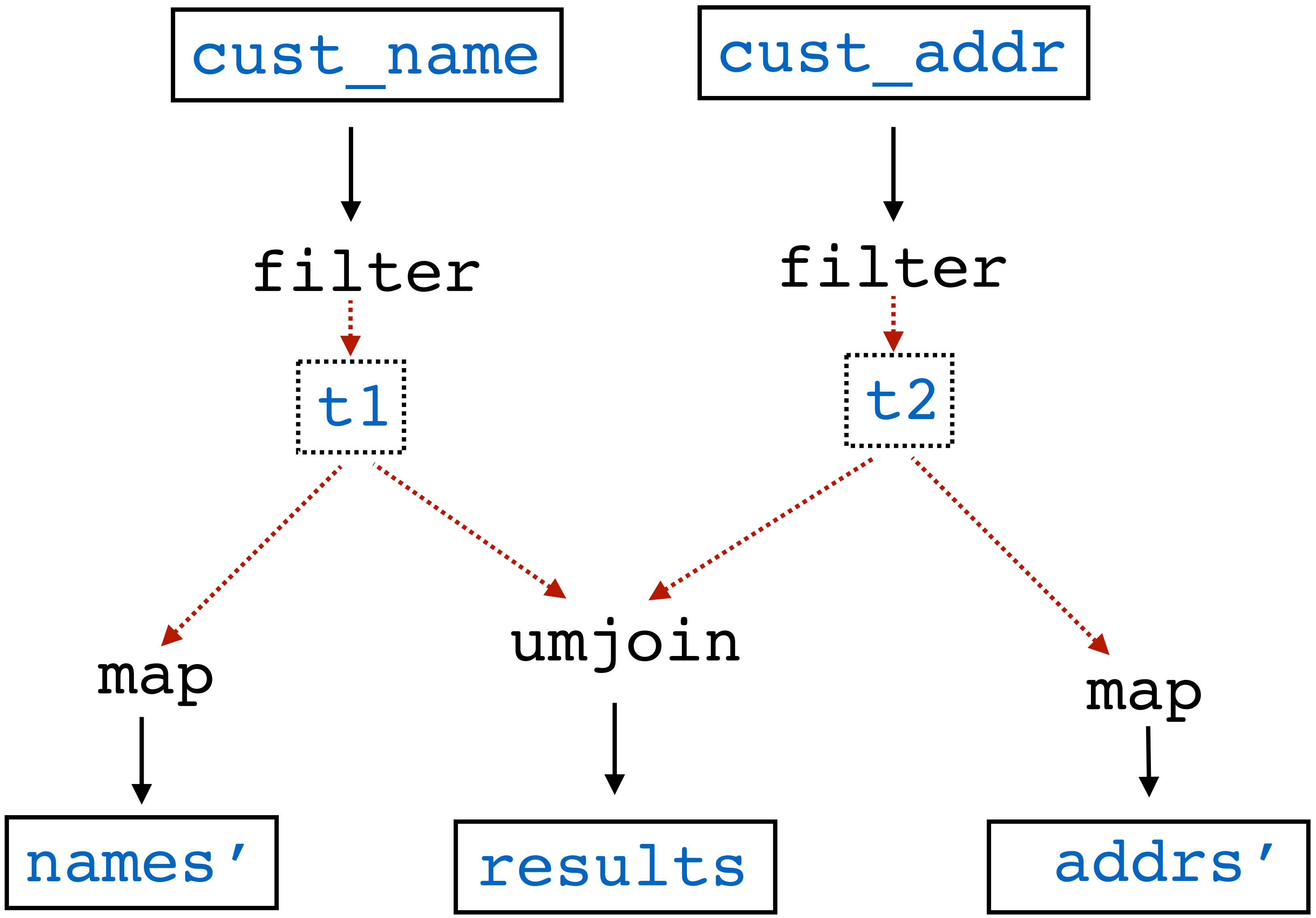
map

names'

results

addrs'





cust_name

cust_addr

filter

filter

t1

t2

map

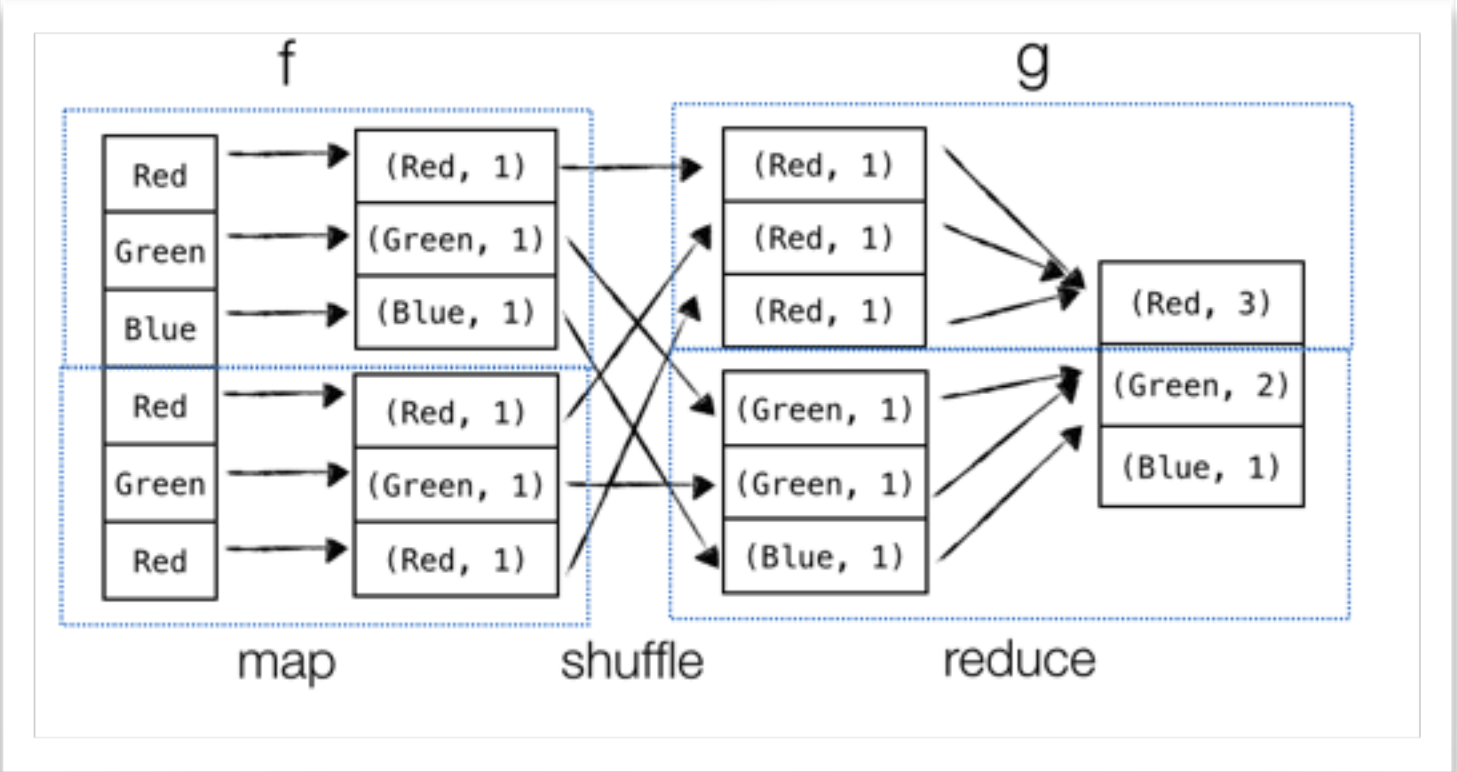
umjoin

map

names'

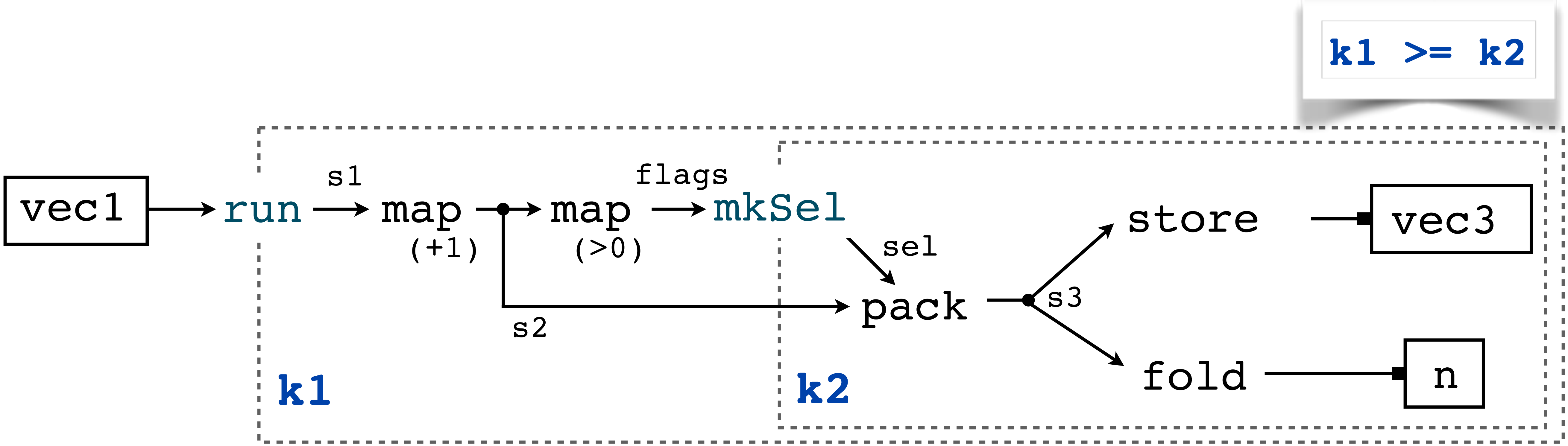
results

addrs'



from … ?

```
filterMax :: Vector Int -> (Vector Int, Int)
filterMax vec1
 = run vec1 (\s1 ->                  s1 :: Series k1 Int
    let s2    = map (+ 1) s1         s2 :: Series k1 Int
        flags = map (> 0) s2      flags :: Series k1 Bool
    in  mkSel flags (\sel ->        sel :: Sel k1 k2
         let s3   = pack sel s2      s3 :: Series k2 Int
             vec3 = store s3       vec3 :: Vector Int
             n    = fold max 0 s3     n :: Int
         in (vec3, n)))
```



Myriad Creatures

Dryad Microsoft Research est. 2007

- MSR project for writing distributed data-flow programs.
- Subsumes MapReduce.
- Channel communication can be via file system or TCP.
- Has low-level graph description language as well has higher-level user facing languages, embedded and otherwise.
- 2011: canned by MSR in favour of Hadoop on Windows.

Spark amplab UC BERKELEY est. 2010

- Data-parallel data-flow graph execution.
- Partial DAG execution, can re-plan a running query based on statistics collected in-flight.
- RDD: Resilient Distributed Data Sets. If a node fails the affected blocks can be re-executed.
- *Anecdotally: runs fine for data sets that fit in-memory (ie, the ones in the papers). Poor performance for larger data sets. Customers going back to plain MapReduce.*

Dremel (Google)

est. 2010

- Distributed query engine for analytic fragment of SQL.
- Analysis of *in-place* data.
- Nested data model based on protocol buffers.
- 2010: supports single-table queries only, no joins.

Tenzing (Google)

est. 2011

- Distributed query engine for analytic fragment of SQL
- Hetrogeneous data-parallel data-flow, can run across multiple back-end data stores.
- Can use remote-lookup reads for joins,
  and do joins via a foreign API rather than on local data.
- 2011: paper mentions experimental LLVM query compiler, but not integrated into the main product.

druid

est. 2011

- Distributed data-store for ingesting time-series data.
- Has own query format packed into JSON.
- Working data is kept in memory.
- Optimised for real-time ingestion.
- No nested data representation.
- No joins.

cloudera IMPALA

est. 2012

- Distributed query engine for analytic fragment of SQL.
- Joins limited by aggregate memory of cluster, (no streaming merge joins from secondary storage)
- “Runtime code generation in Cloudera Impala” discusses LLVM for query compilation, but no mention of fusion.
- 2014: Cloudera now making Hive run on Spark, so maybe not committed to Impala anymore.
- *Anecdotally: implementation is flakey, crashes often.*
- *10-100x faster than Hive for TPC-DS benchmarks.*

Future Work / Challenges

Key Challenges

- Proliferation of distinct systems that want to own the data.
- No overarching framework for comparing the systems.
    - Relational algebra is logical-plan only,
      need to compare physical plans as well.
- Little separation of foundational issues vs implementation.
    - *Don't know how to do it* vs *too expensive with TeraData*.
- Lack of space complexity guarantees for intermediate results.
- Query plans transformed with non-confluent rewrite systems.
- Performance correctness?
- The "cult of optimisation"