

Fusing filters with Integer Linear Programming

Amos Robinson (that's me!)

Gabriele Keller

Ben Lippmeier

I don't want to write *this*

```
DO 10 I = 1, SIZE(XS)
    SUM1 = SUM1 + XS(I)
    IF (XS(I) .GT. 0) THEN
        SUM2 = SUM2 + XS(I)
    END IF
10 CONTINUE
```

```
DO 20 I = 1, SIZE(XS)
    NOR1(I) = XS(I) / SUM1
    NOR2(I) = XS(I) / SUM2
20 CONTINUE
```

I'd rather write this

```
sum1 = fold (+) 0 xs
```

```
nor1 = map (/ sum1) xs
```

```
ys = filter (> 0) xs
```

```
sum2 = fold (+) 0 ys
```

```
nor2 = map (/ sum2) xs
```

But I also want speed

- Naive compilation: one loop for each combinator
- We need fusion!

Vertical fusion

```
sum1 = fold (+) 0 xs -- loop 1
```

```
nor1 = map (/ sum1) xs -- loop 2
```

```
ys = filter (> 0) xs -- loop 3
```

```
sum2 = fold (+) 0 ys -- loop 4
```

```
nor2 = map (/ sum2) xs -- loop 5
```

Vertical fusion

```
sum1 = fold (+) 0 xs -- loop 1
```

```
nor1 = map (/ sum1) xs -- loop 2
```

```
sum2 = filterFold
```

```
      (> 0) (+) 0 xs -- loop 3
```

```
nor2 = map (/ sum2) xs -- loop 4
```

Horizontal fusion

```
sum1 = fold (+) 0 xs -- loop 1
```

```
nor1 = map (/ sum1) xs -- loop 2
```

```
sum2 = filterFold
```

```
      (> 0) (+) 0 xs -- loop 3
```

```
nor2 = map (/ sum2) xs -- loop 4
```

Horizontal fusion

```
sum1 = fold (+) 0 xs -- loop 1
```

```
(nor1, sum2)
```

```
= mapFilterFold (/ sum1)
```

```
(> 0) (+) 0 xs -- loop 2
```

```
nor2 = map (/ sum2) xs -- loop 3
```


Finished

```
sum1 = fold (+) 0 xs -- loop 1
```

```
(nor1, sum2)
```

```
= mapFilterFold (/ sum1)
```

```
(> 0) (+) 0 xs -- loop 2
```

```
nor2 = map (/ sum2) xs -- loop 3
```

Multiple choices

- What if we applied the fusion rules in a different order?
- There are far too many to try all of them, but...

Order matters

```
sum1 = fold (+) 0 xs -- loop 1
```

```
nor1 = map (/ sum1) xs -- loop 2
```

```
ys = filter (> 0) xs -- loop 3
```

```
sum2 = fold (+) 0 ys -- loop 4
```

```
nor2 = map (/ sum2) xs -- loop 5
```

Order matters

```
sum1 = fold (+) 0 xs -- loop 1
```

```
nor1 = map (/ sum1) xs -- loop 2
```

```
sum2 = filterFold
```

```
(> 0) (+) 0 xs -- loop 3
```

```
nor2 = map (/ sum2) xs -- loop 5
```

Order matters

```
(sum1, sum2) = foldFilterFold
```

```
    (+) 0
```

```
    (> 0) (+) 0 xs -- loop 1
```

```
nor1 = map (/ sum1) xs -- loop 2
```

```
nor2 = map (/ sum2) xs -- loop 3
```

Order matters

```
(sum1, sum2) = foldFilterFold
```

```
(+) 0
```

```
(> 0) (+) 0 xs -- loop 1
```

```
nor1 = map (/ sum1) xs -- loop 2
```

```
nor2 = map (/ sum2) xs -- loop 3
```

Order matters

```
(sum1, sum2) = foldFilterFold
```

```
(+) 0
```

```
(> 0) (+) 0 xs -- loop 1
```

```
(nor1, nor2) = mapMap
```

```
(/ sum1) (/ sum2) xs -- loop 2
```

Order matters

```
(sum1, sum2) = foldFilterFold
```

```
    (+) 0
```

```
    (> 0) (+) 0 xs -- loop 1
```

```
(nor1, nor2) = mapMap
```

```
    (/ sum1) (/ sum2) xs -- loop 2
```


Which order?

- Finding the *best* order is the hard part.
- That's why we use...

Integer Linear Programming!

Minimise $y - x$ *Objective*

Subject to $0 \leq x \leq 2$ *Constraints*

$$0 \leq y \leq 2$$

$$x + 2y \geq 3$$

Where $x : \mathbb{Z}$ *Variables*

$$y : \mathbb{Z}$$

Integer Linear Programming!

Minimise $y - x$ *Objective*

Subject to $0 \leq x \leq 2$ *Constraints*

$$0 \leq y \leq 2$$

$$x + 2y \geq 3$$

Where $x : \mathbb{Z} = 2$ *Variables*

$$y : \mathbb{Z} = 1$$

Create a graph

XS

```
sum1 = fold (+) 0 xs
```

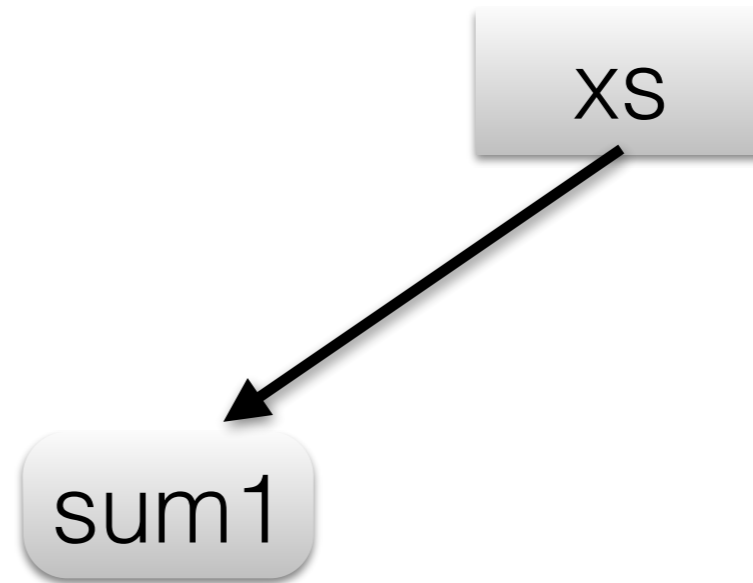
```
nor1 = map (/ sum1) xs
```

```
ys = filter (> 0) xs
```

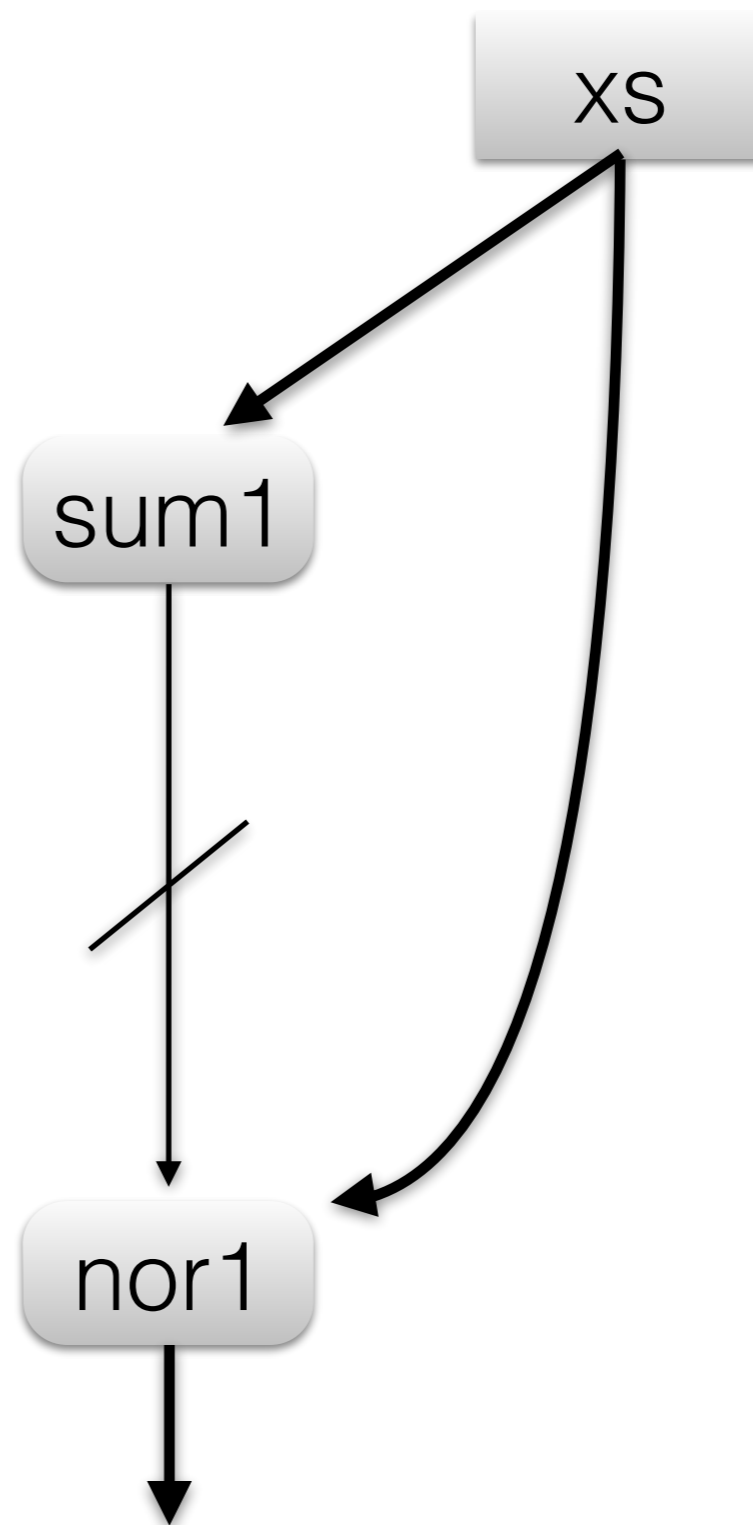
```
sum2 = fold (+) 0 ys
```

```
nor2 = map (/ sum2) xs
```

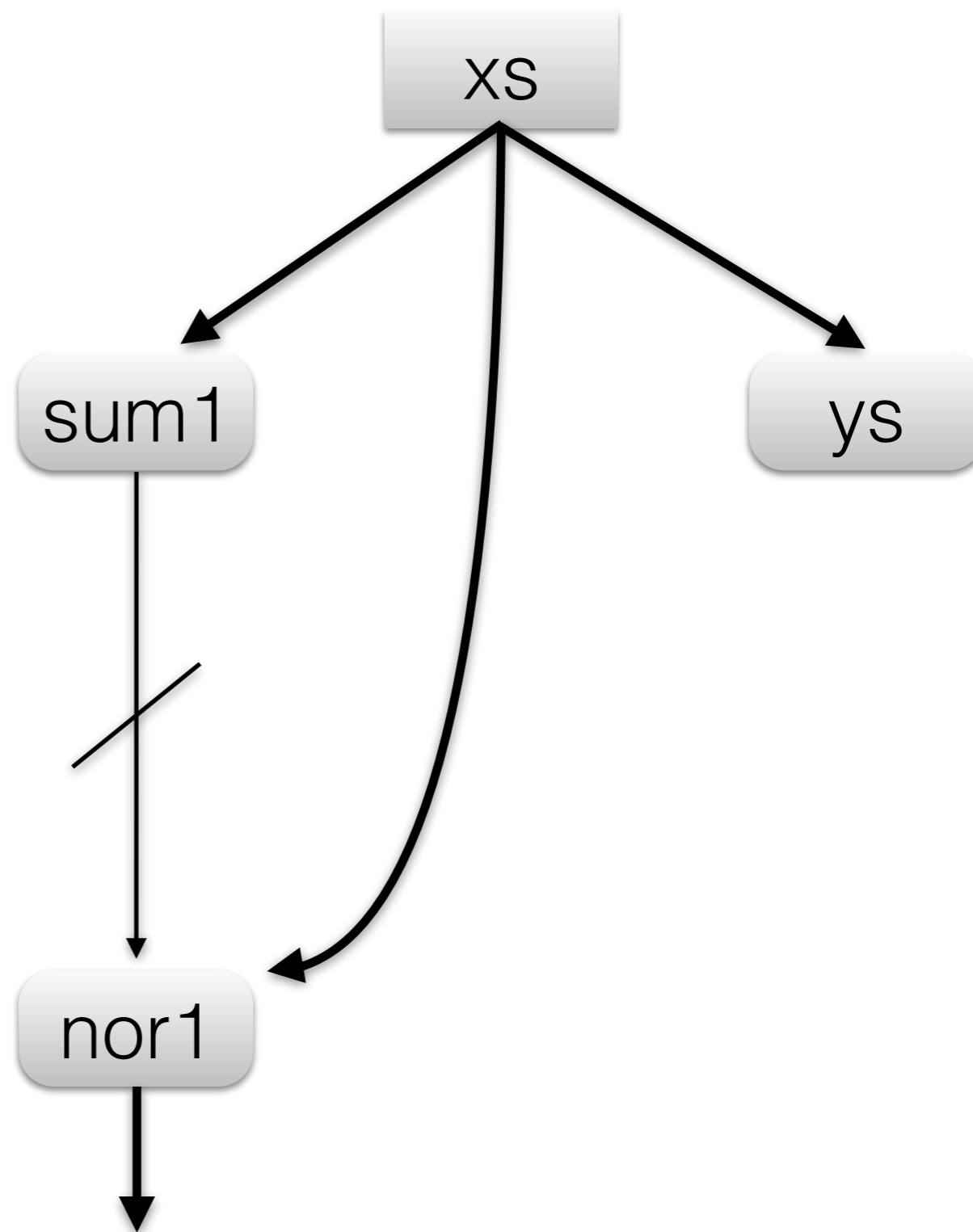
```
sum1 = fold (+) 0 xs
nor1 = map (/ sum1) xs
ys = filter (> 0) xs
sum2 = fold (+) 0 ys
nor2 = map (/ sum2) xs
```



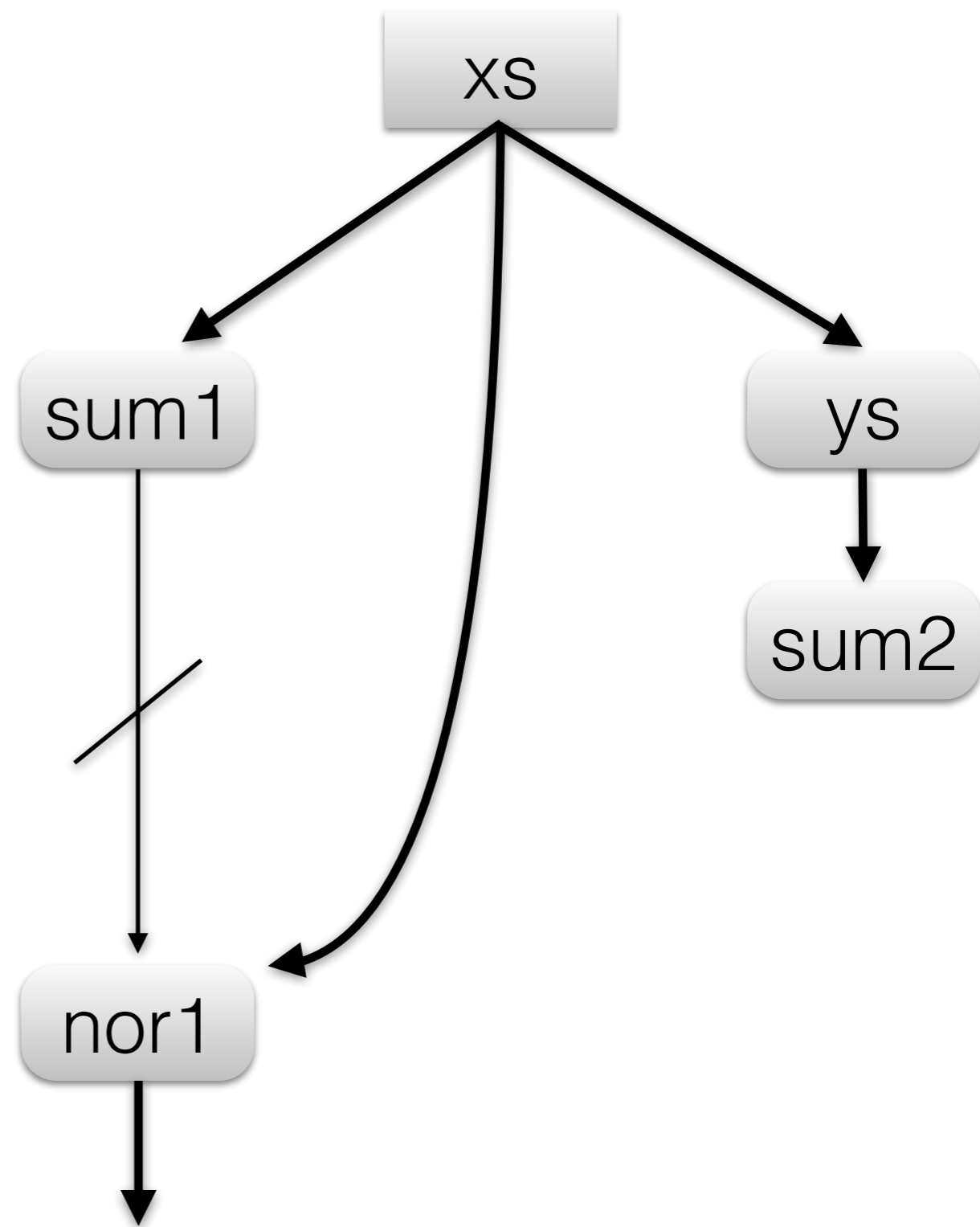
```
sum1 = fold (+) 0 xs
nor1 = map (/ sum1) xs
ys = filter (> 0) xs
sum2 = fold (+) 0 ys
nor2 = map (/ sum2) xs
```



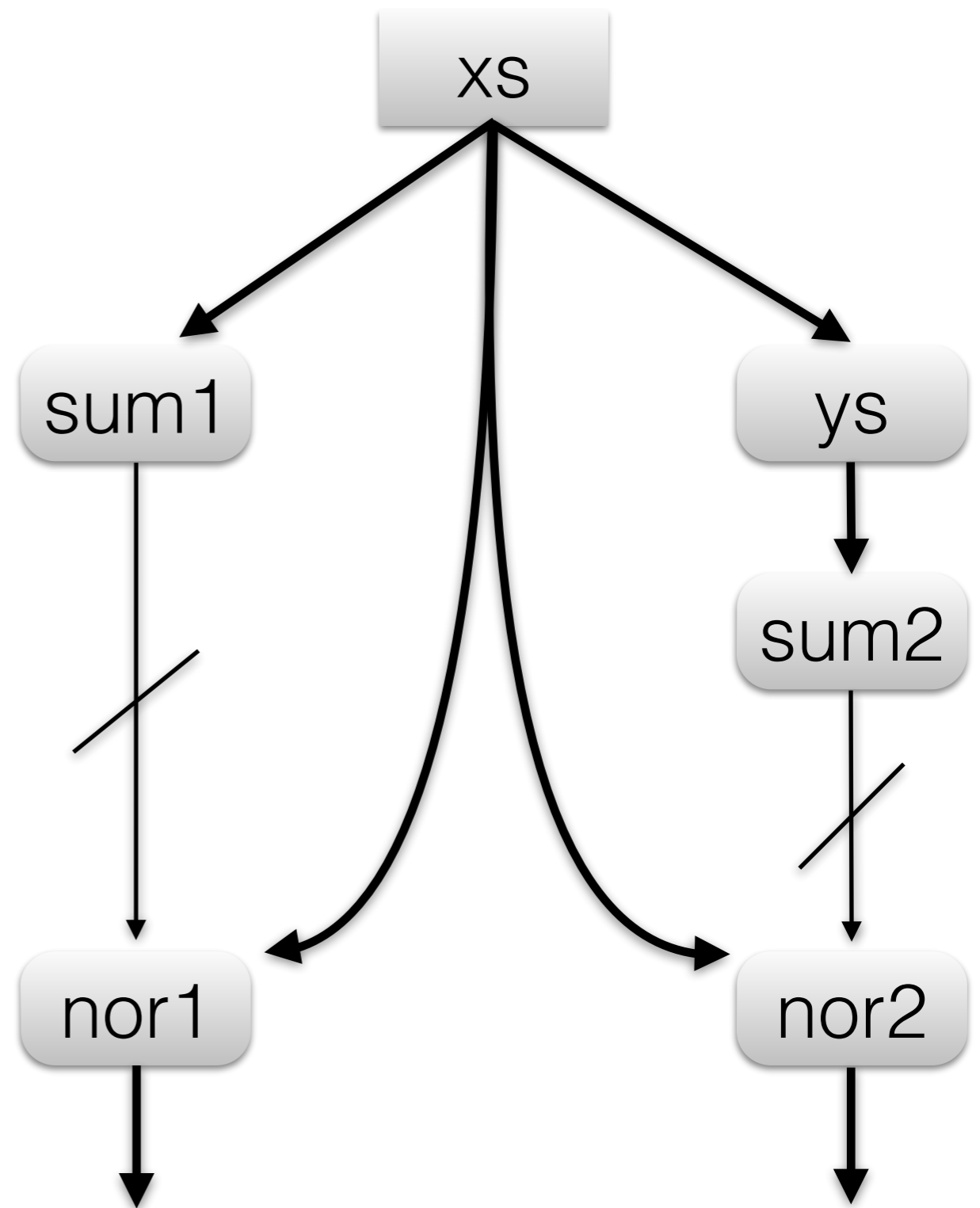
```
sum1 = fold (+) 0 xs
nor1 = map (/ sum1) xs
ys = filter (> 0) xs
sum2 = fold (+) 0 ys
nor2 = map (/ sum2) xs
```



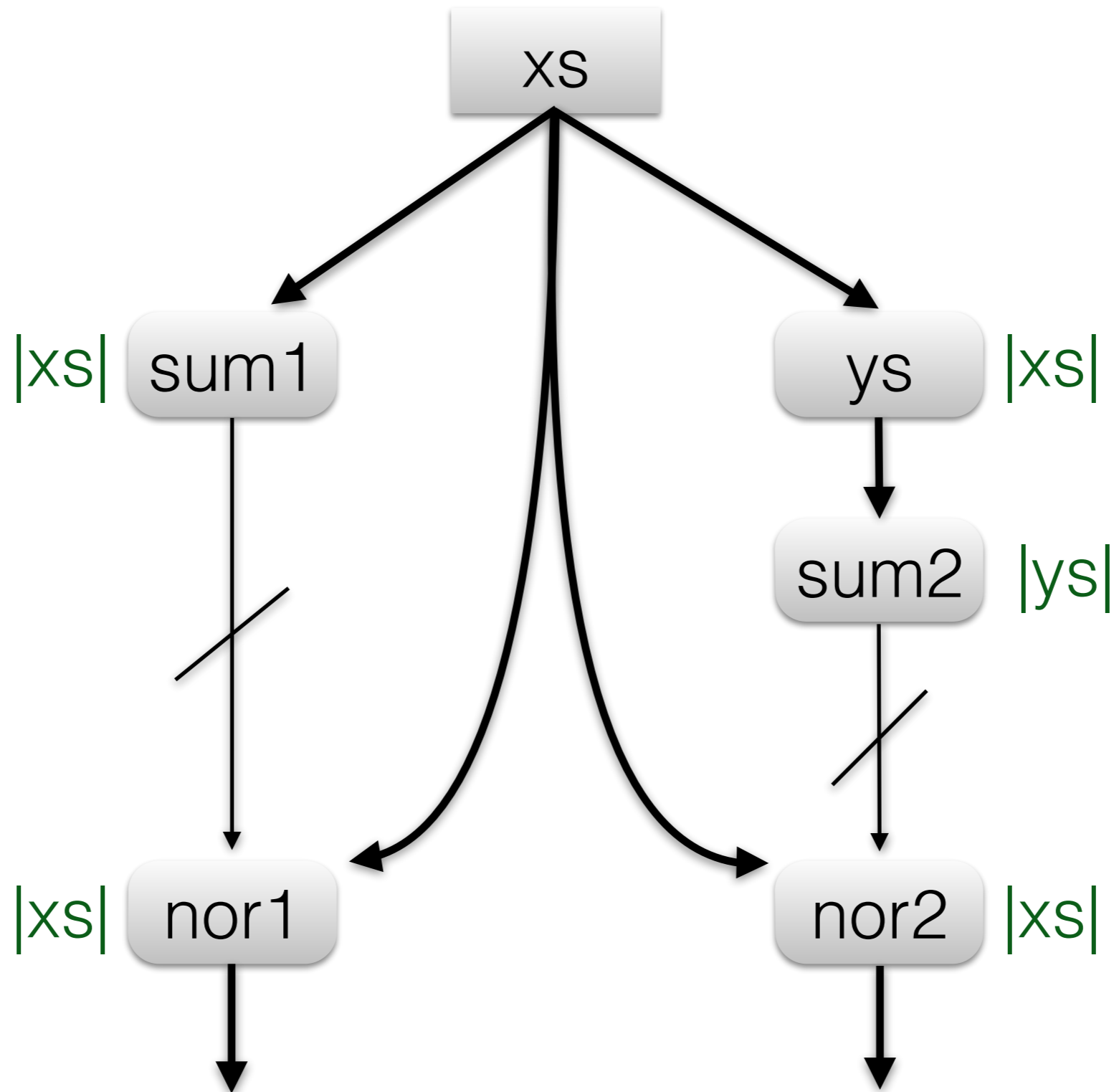

```
sum1 = fold (+) 0 xs
nor1 = map (/ sum1) xs
ys = filter (> 0) xs
sum2 = fold (+) 0 ys
nor2 = map (/ sum2) xs
```



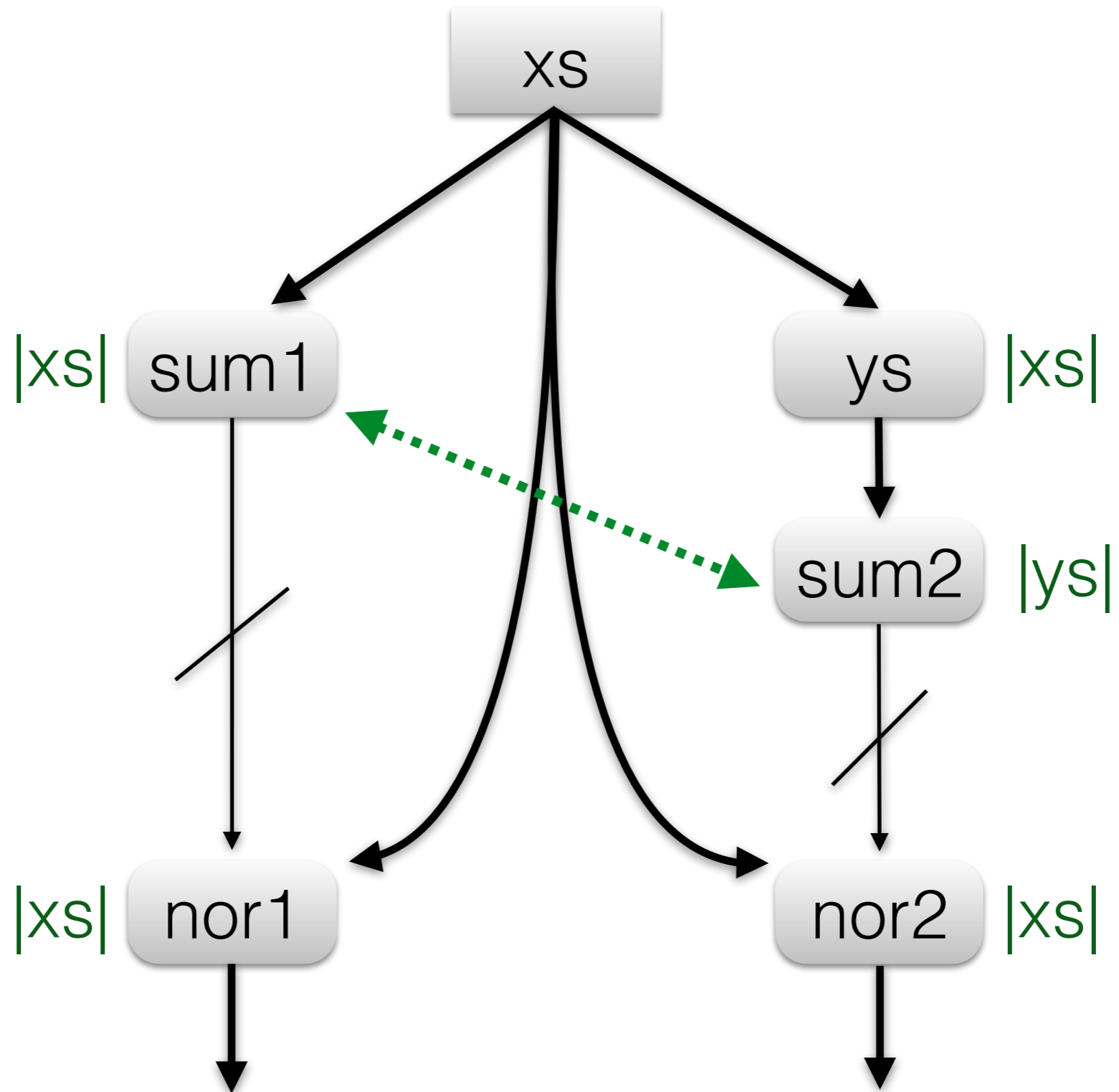
```
sum1 = fold (+) 0 xs
nor1 = map (/ sum1) xs
ys = filter (> 0) xs
sum2 = fold (+) 0 ys
nor2 = map (/ sum2) xs
```



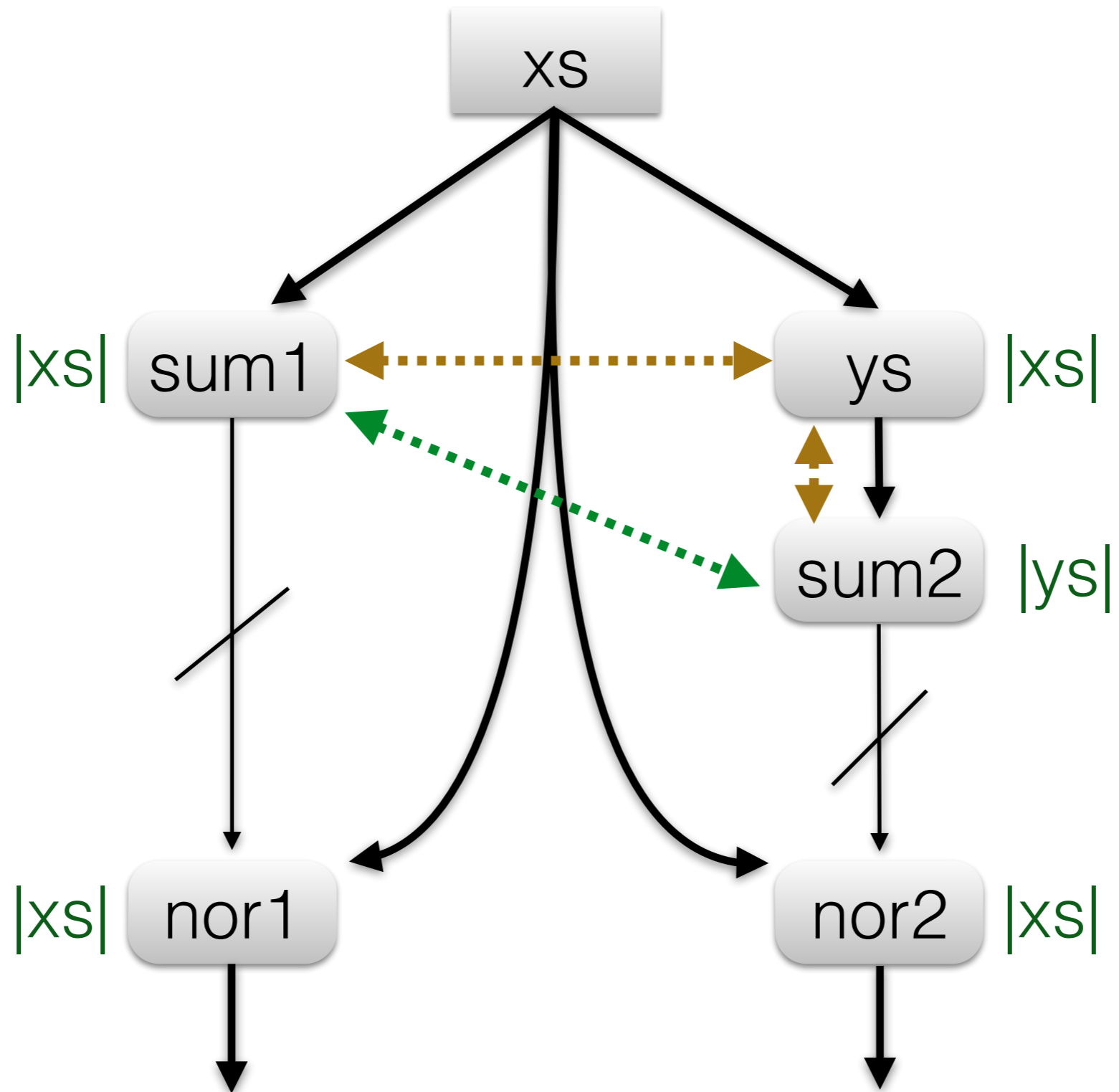
Different size loops



Different size loops



Different size loops



Filter constraint

Minimise ...

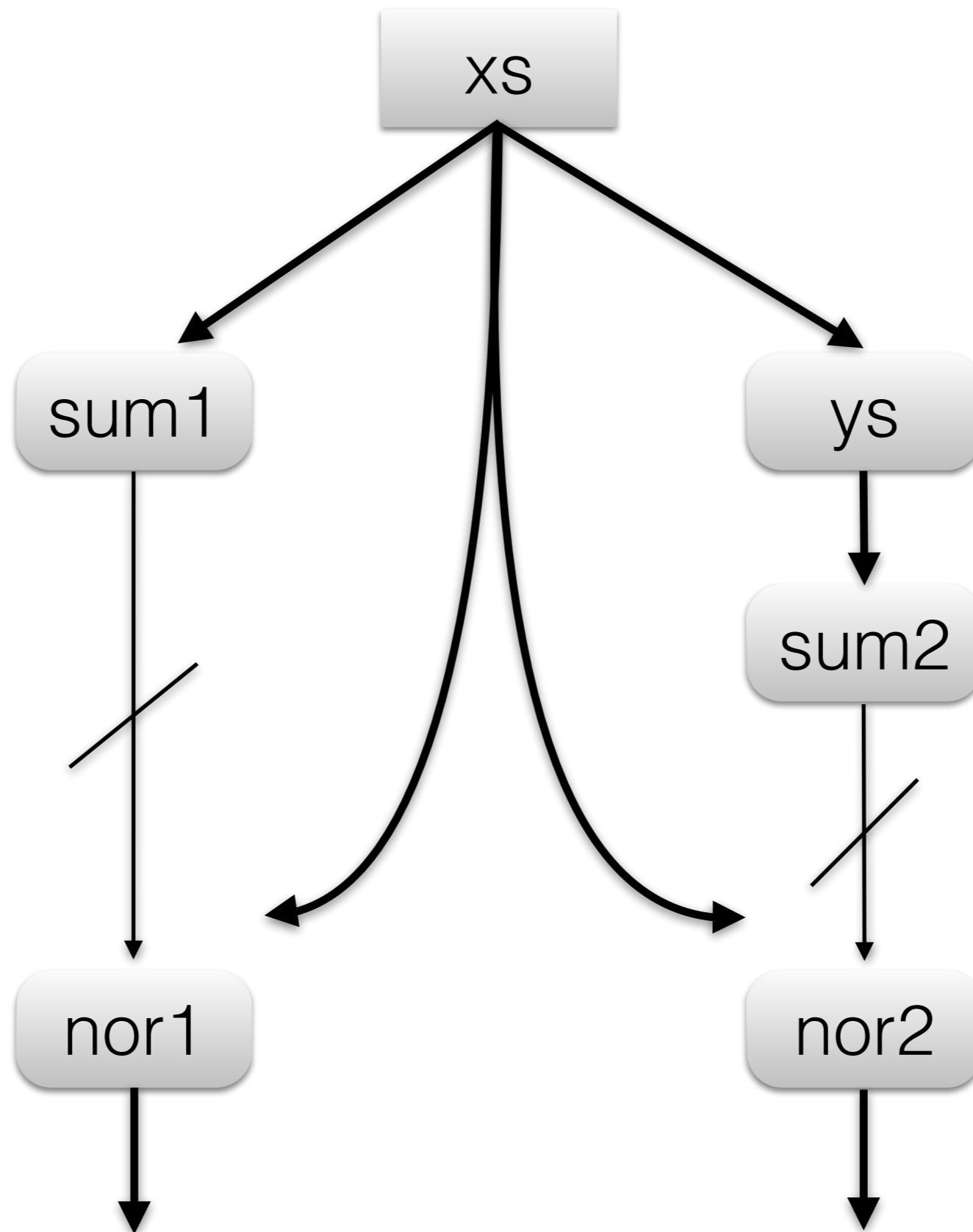
Subject to ...

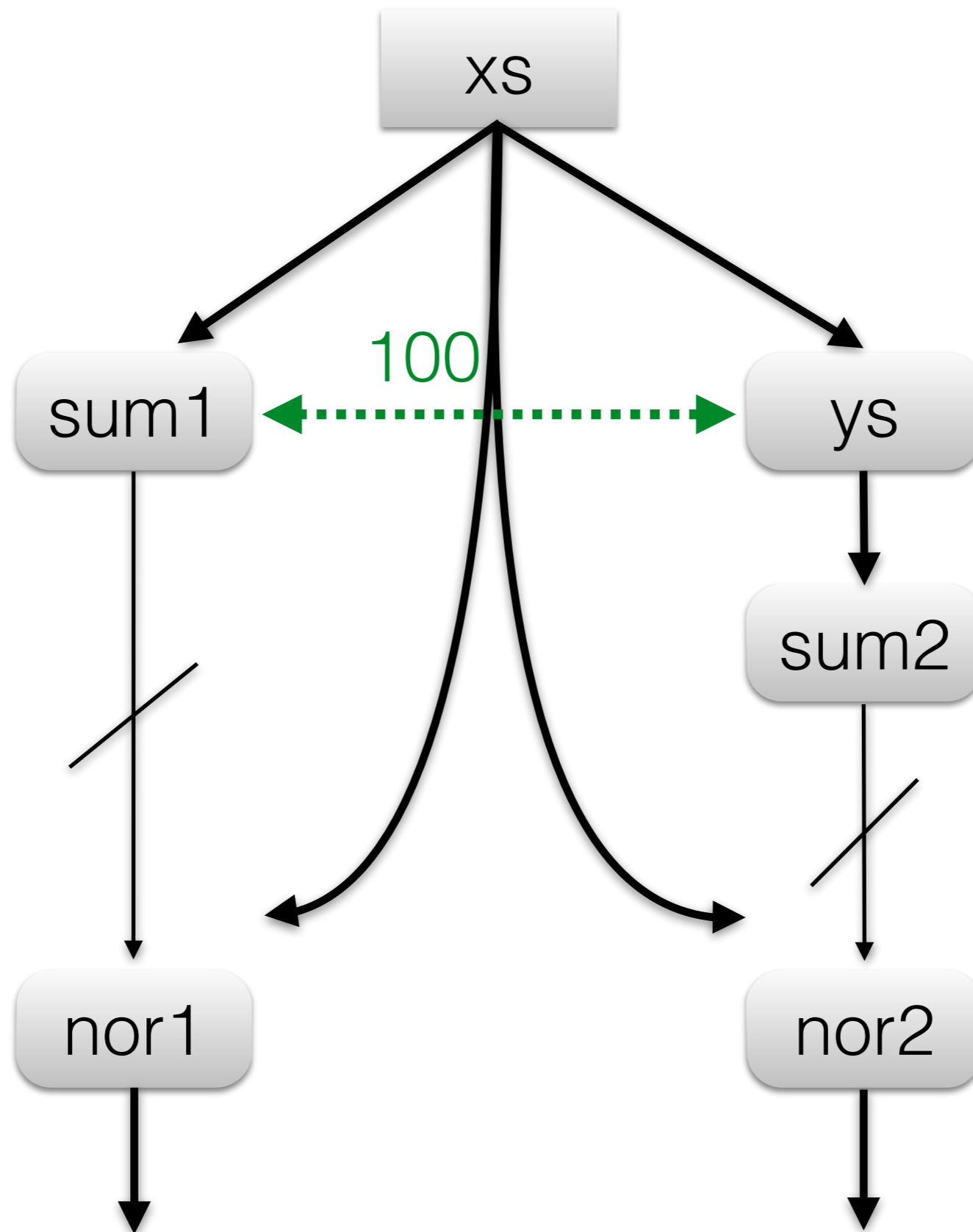
$$f(\text{sum1}, ys) \leq f(\text{sum1}, \text{sum2})$$

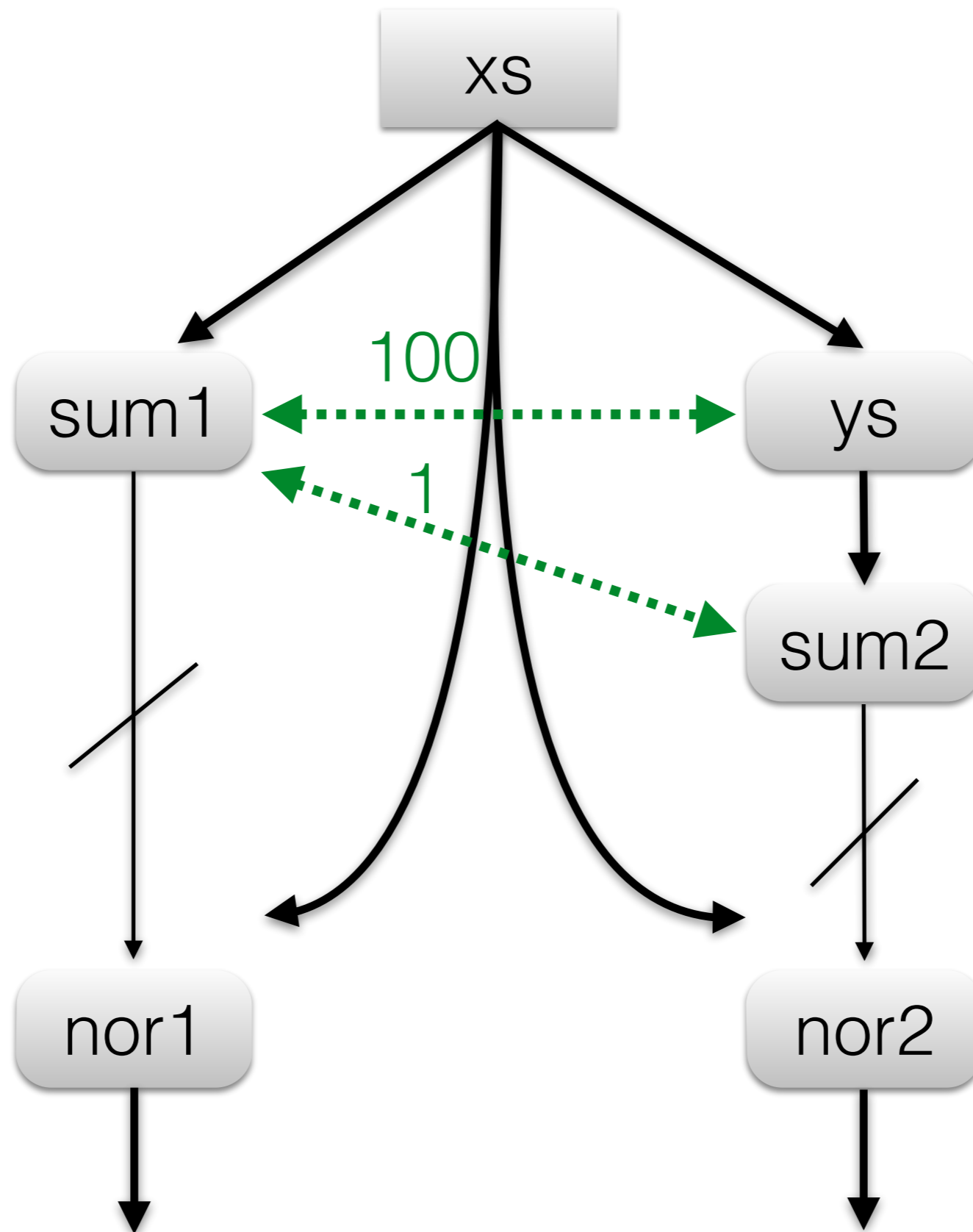
$$f(\text{sum2}, ys) \leq f(\text{sum1}, \text{sum2})$$

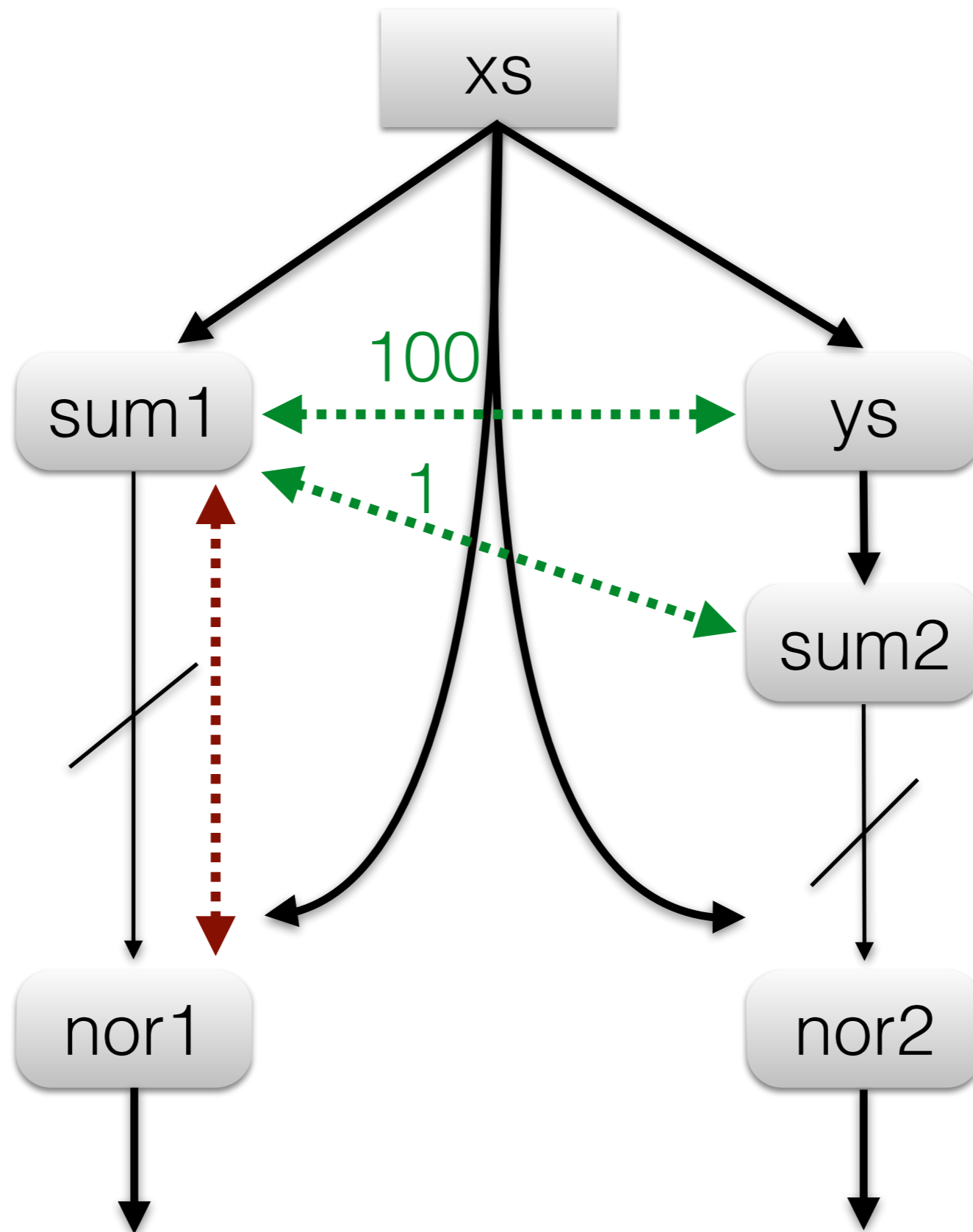
$f(a,b) = 0$ iff a and b are fused together

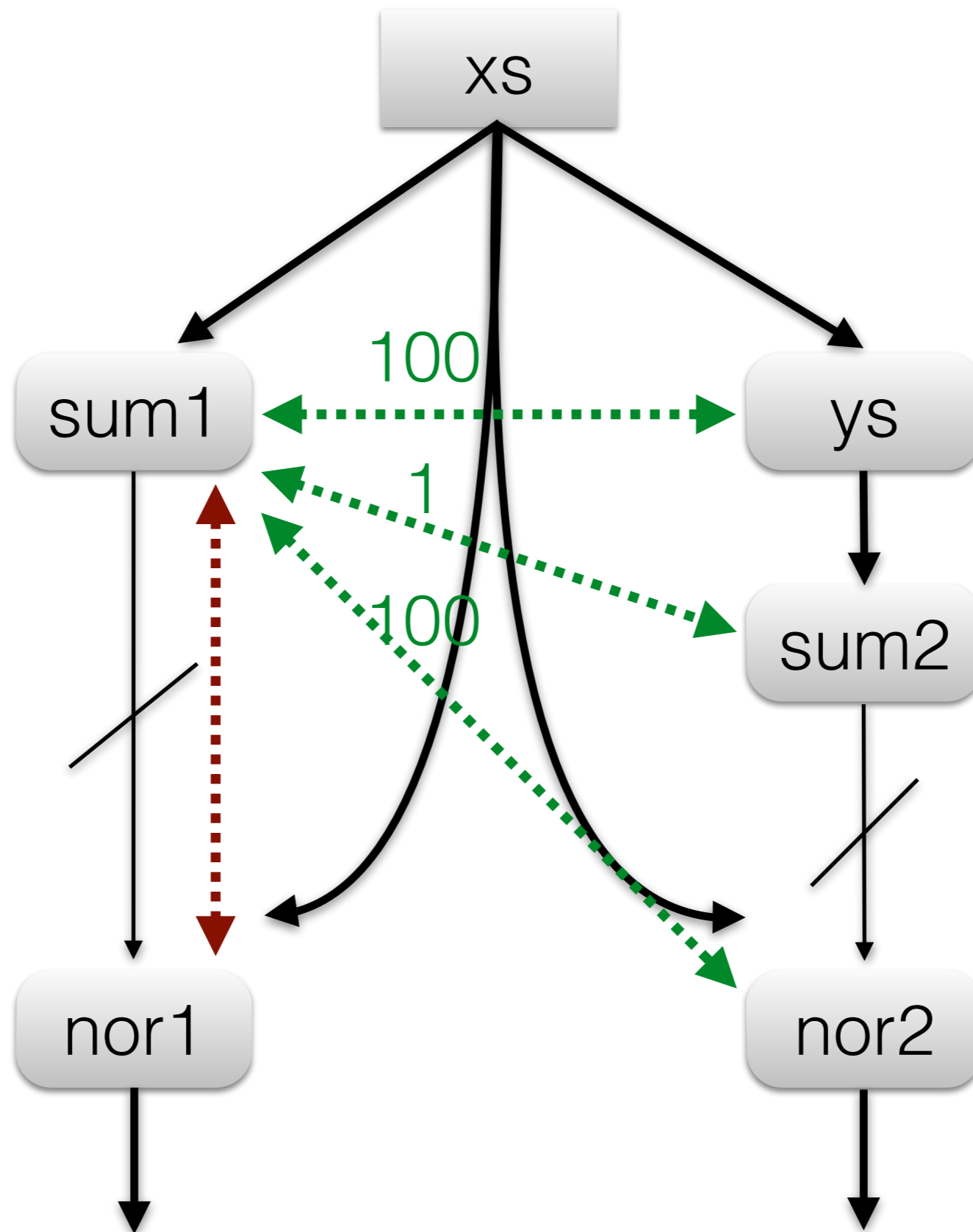
Objective function

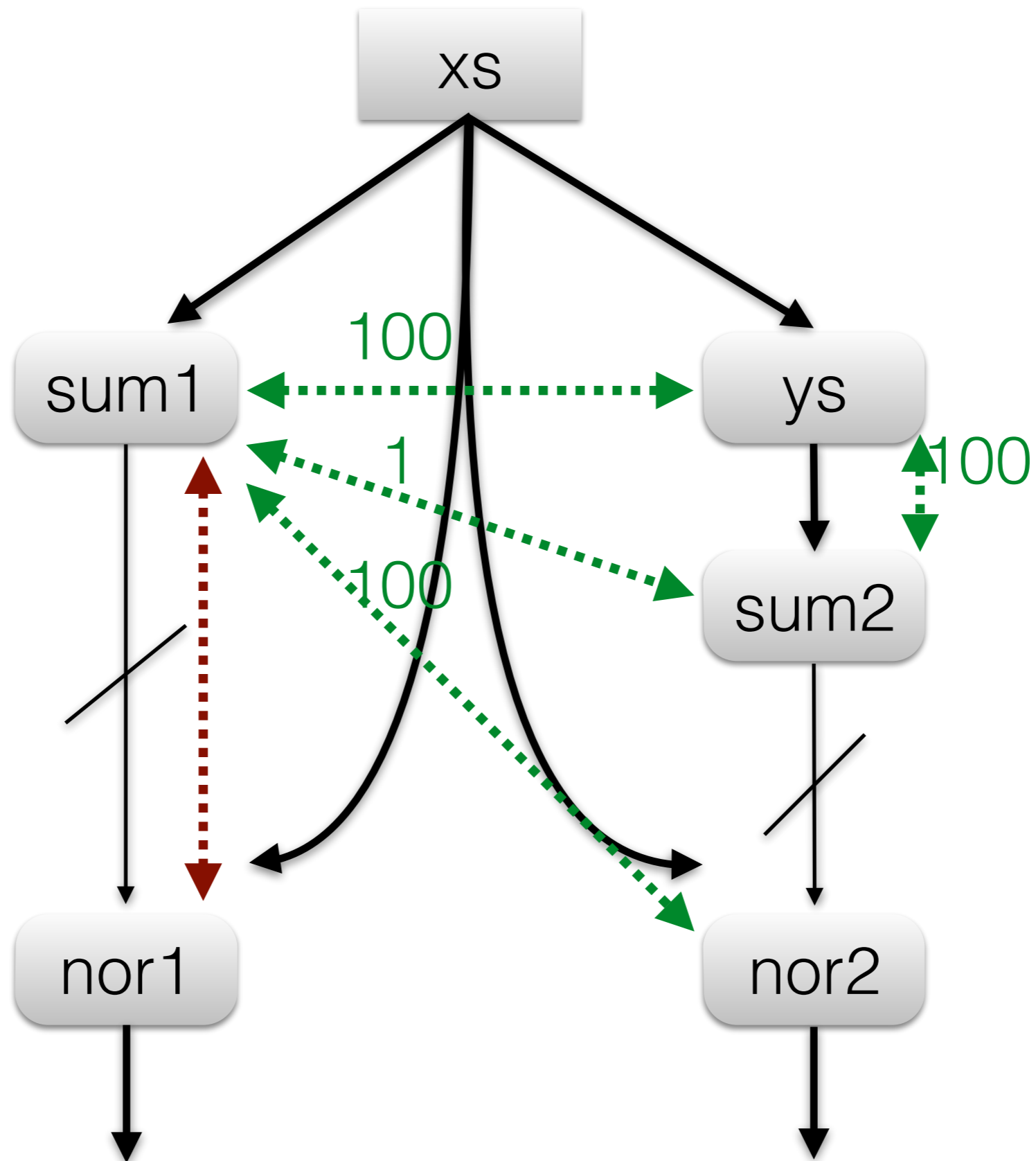


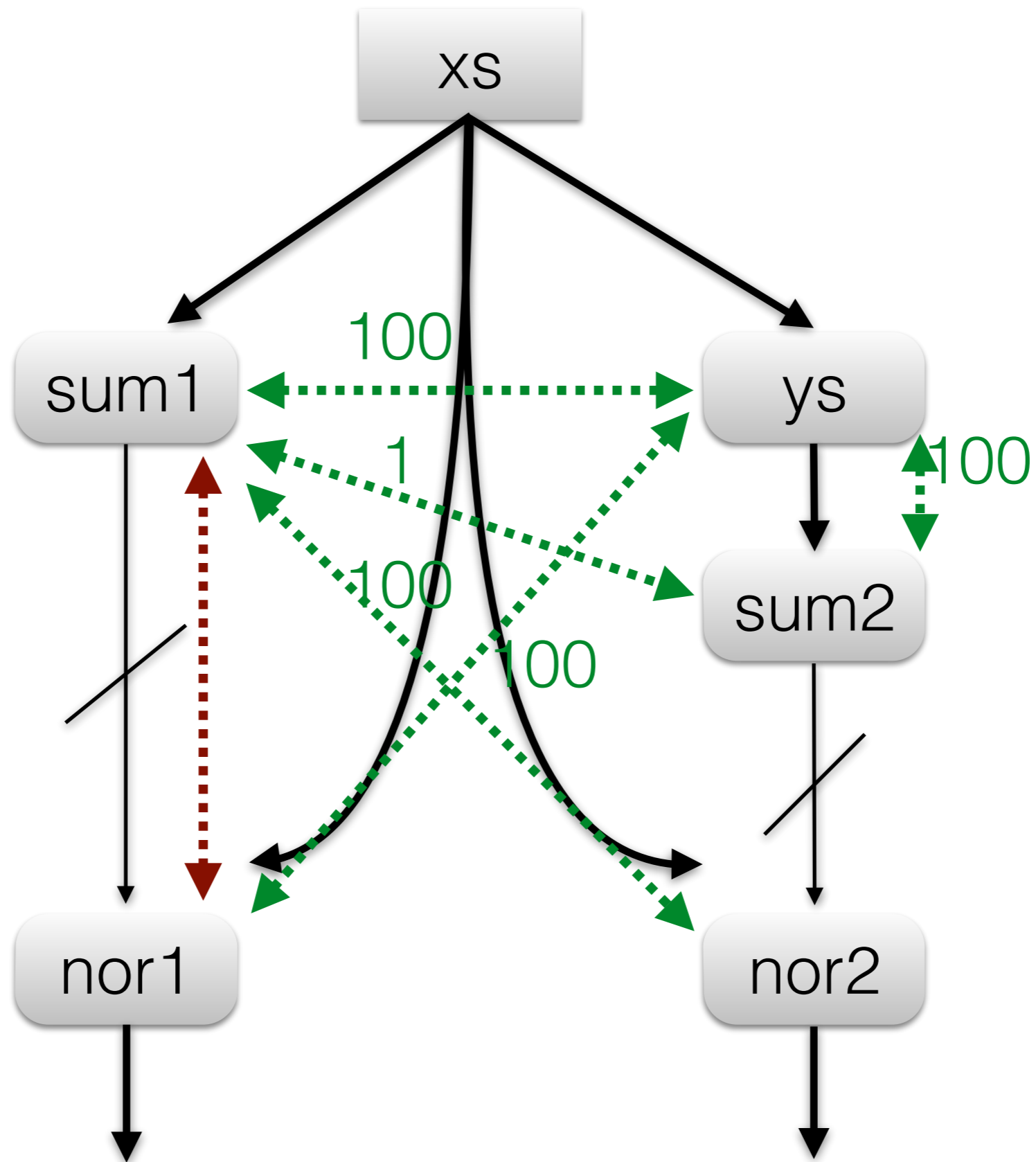


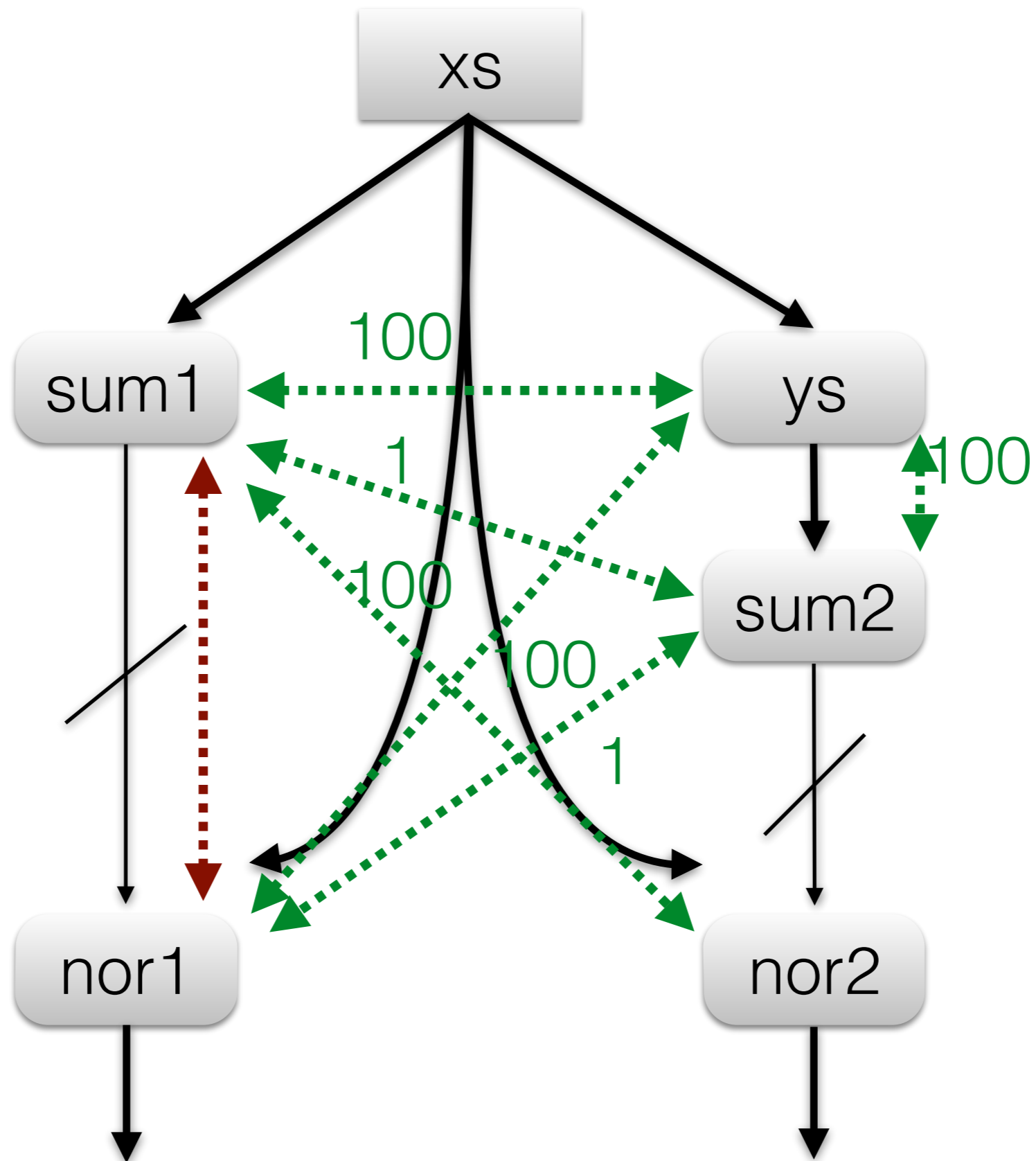


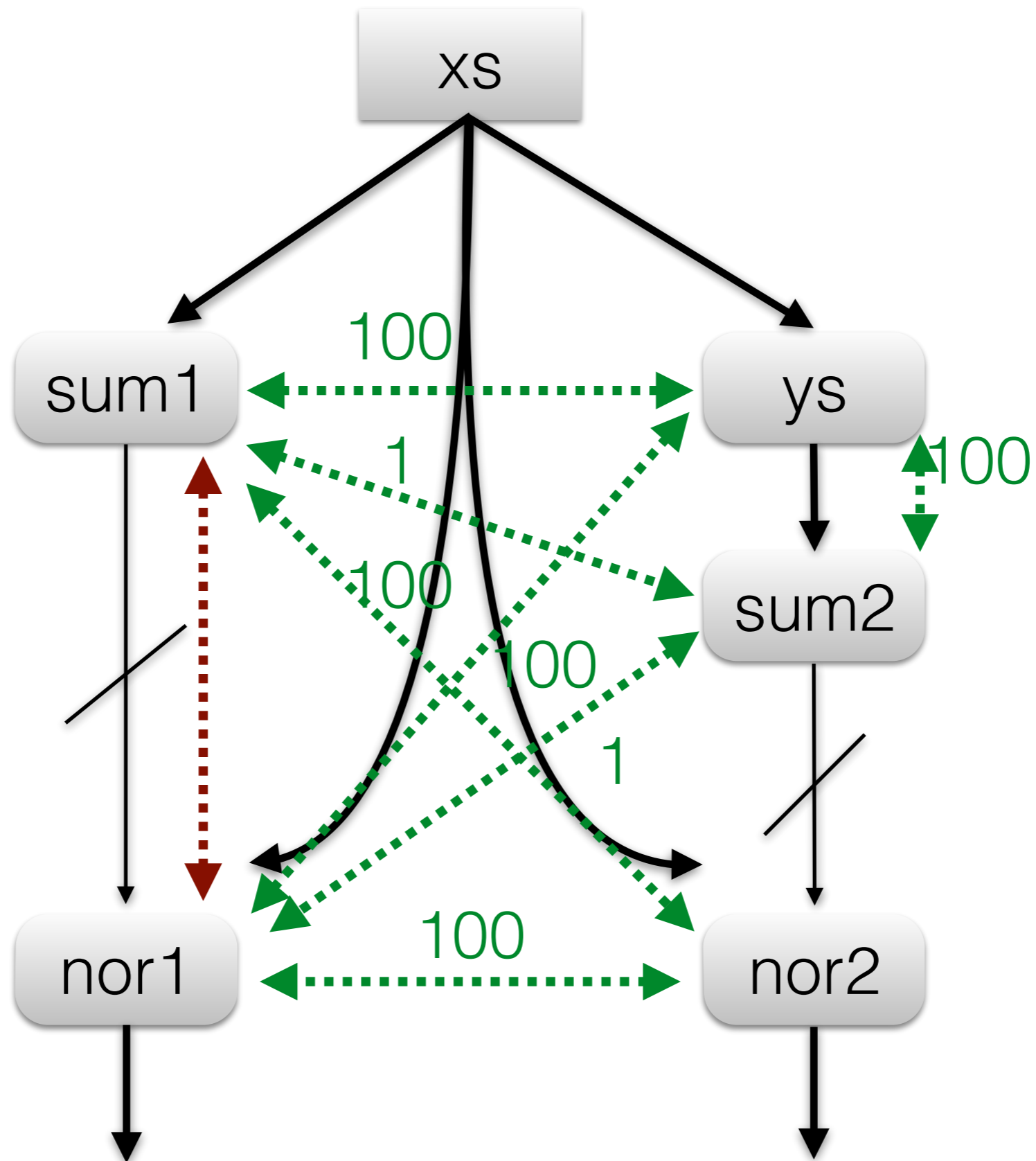








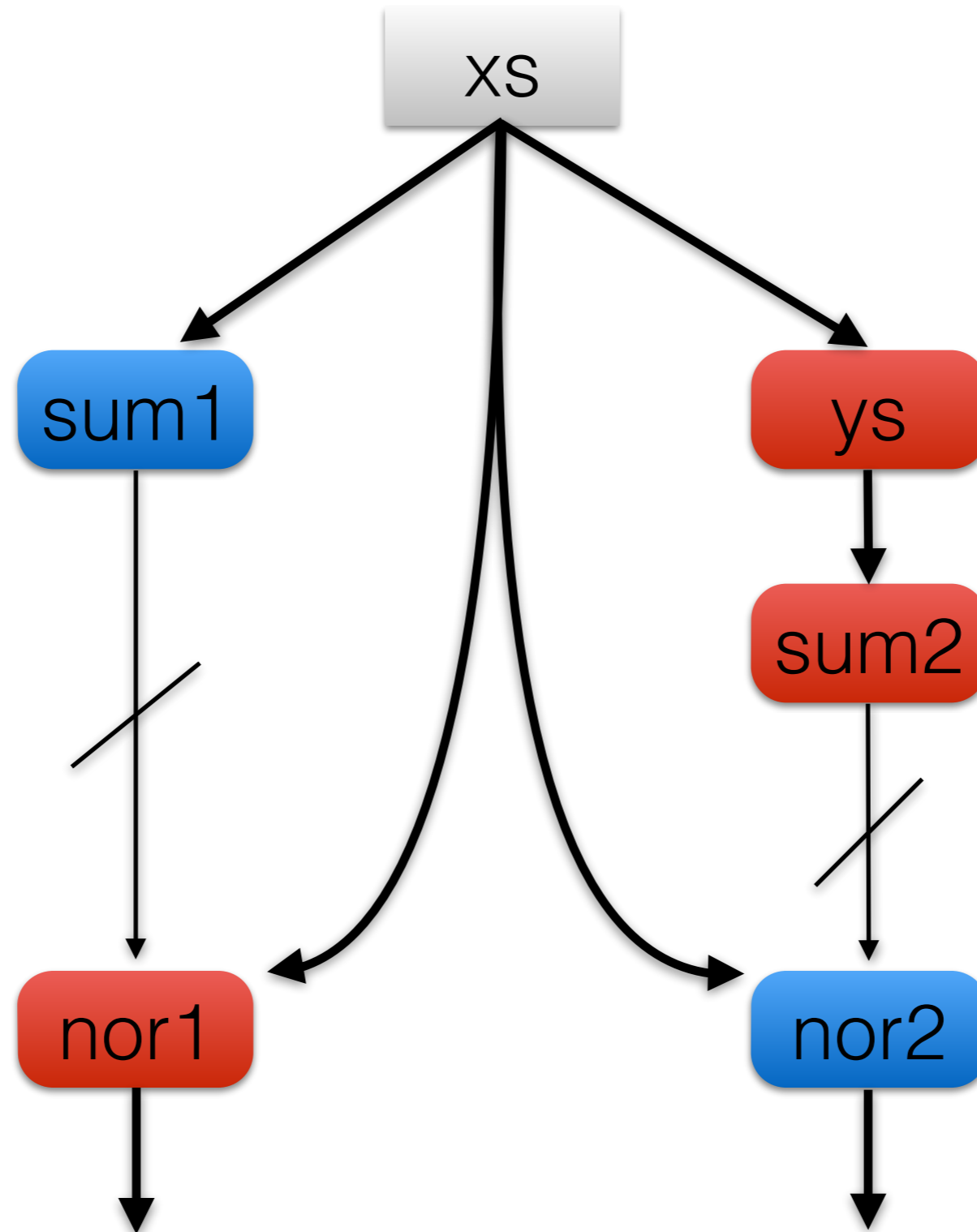




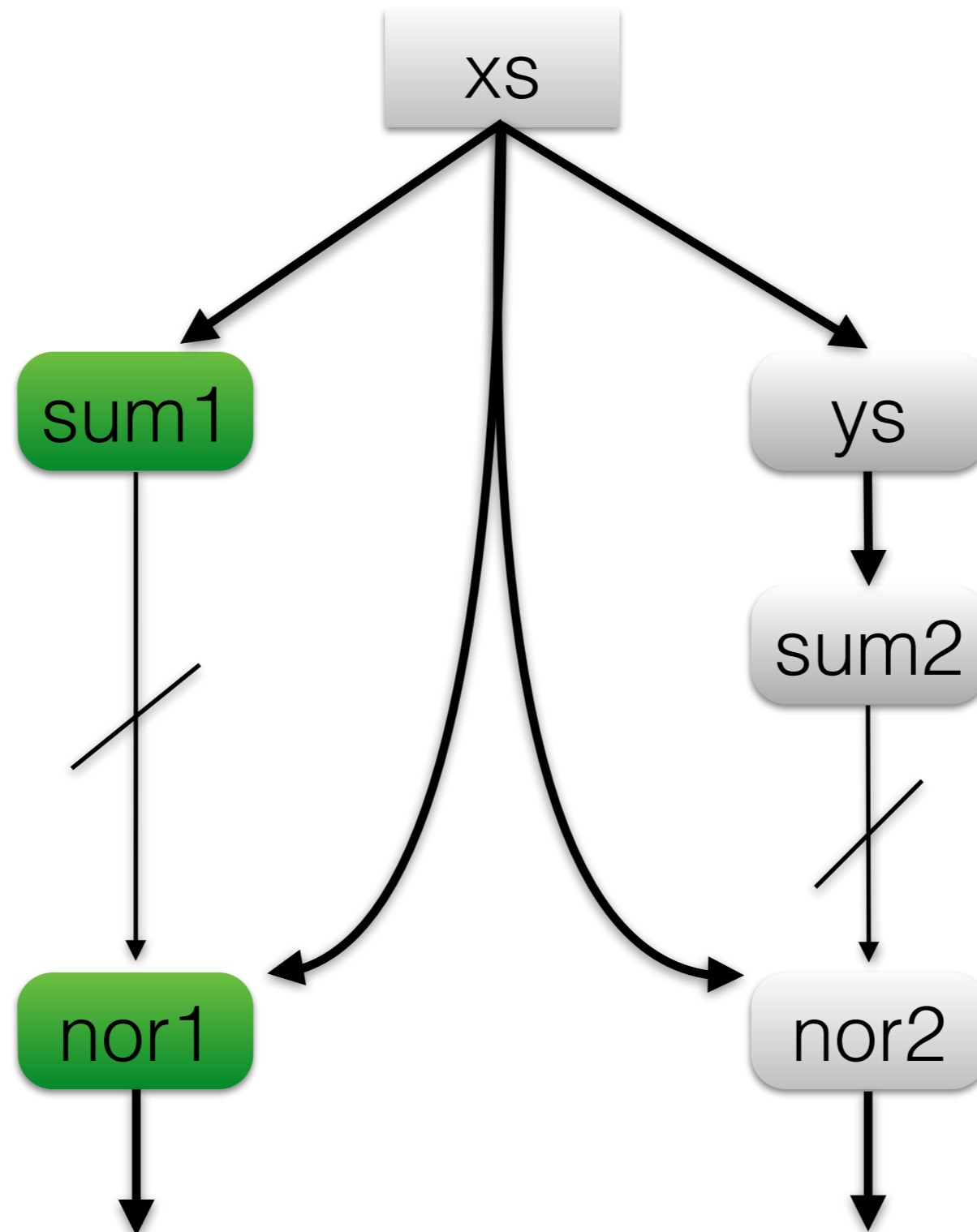
Objective function

$$\begin{aligned} \text{Minimise} \quad & 100f(\text{sum1}, \text{ys}) & + & \quad 1f(\text{sum1}, \text{sum2}) \\ & + 100f(\text{sum1}, \text{nor2}) & + & \quad 100f(\text{ys}, \text{sum2}) \\ & + 100f(\text{ys}, \text{nor1}) & + & \quad 1f(\text{sum2}, \text{nor1}) \\ & + 100f(\text{nor1}, \text{nor2}) \end{aligned}$$

Cyclic clusterings cannot be executed



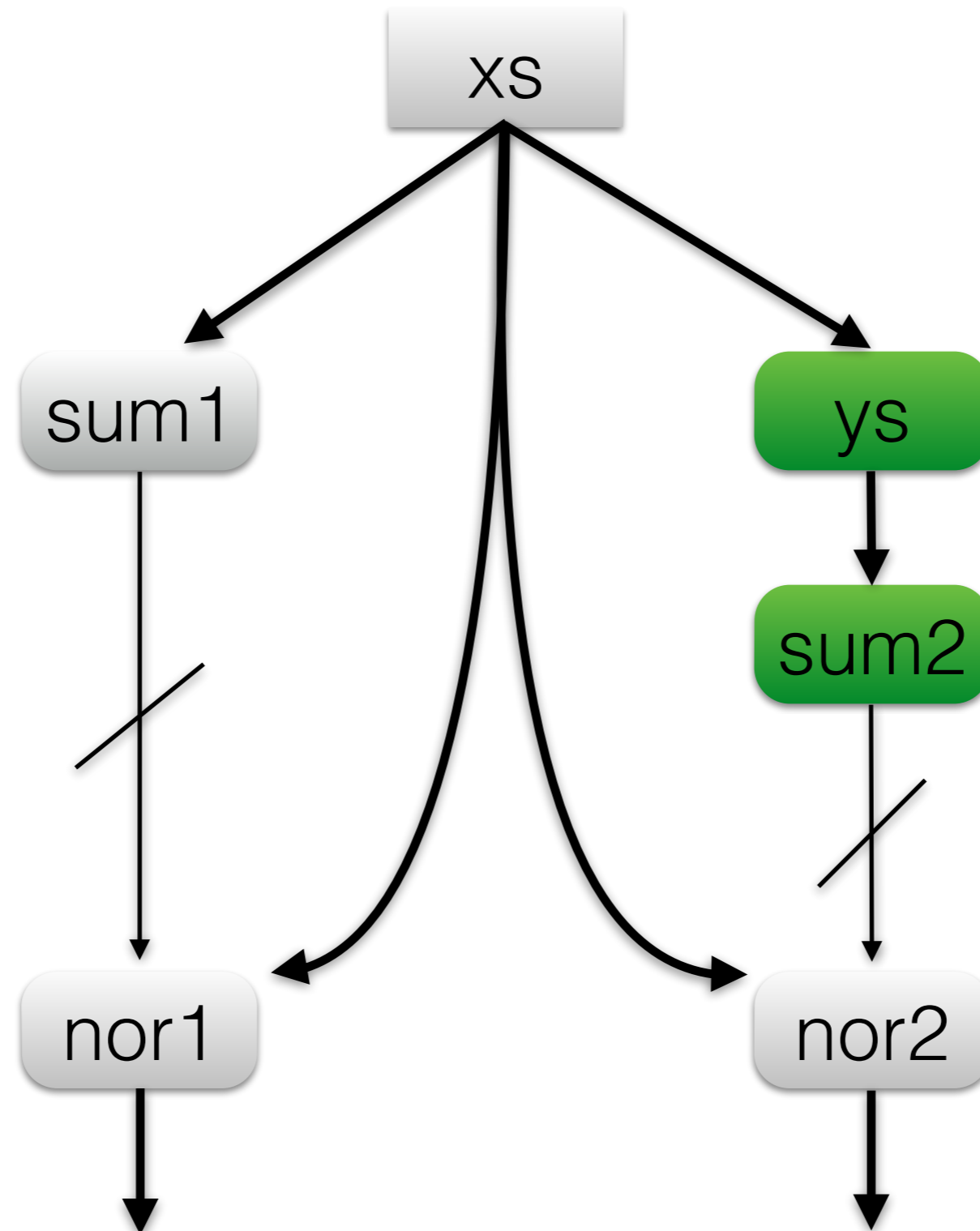
Non-fusible edge



Non-fusible edge

$$o(\text{sum1}) < o(\text{nor1})$$

Fusible edge



Fusible edge

if $f(ys, \text{sum2}) = 0$

then $o(ys) = o(\text{sum2})$

else $o(ys) < o(\text{sum2})$

Fusible edge

$$1 f(y_s, \text{sum2}) \leq o(\text{sum2}) - o(y_s) \leq 100 f(y_s, \text{sum2})$$

Fusible edge - fused

$$1 f(y_s, \text{sum2}) \leq o(\text{sum2}) - o(y_s) \leq 100 f(y_s, \text{sum2})$$

$$0 \leq o(\text{sum2}) - o(y_s) \leq 0$$

$$o(\text{sum2}) = o(y_s)$$

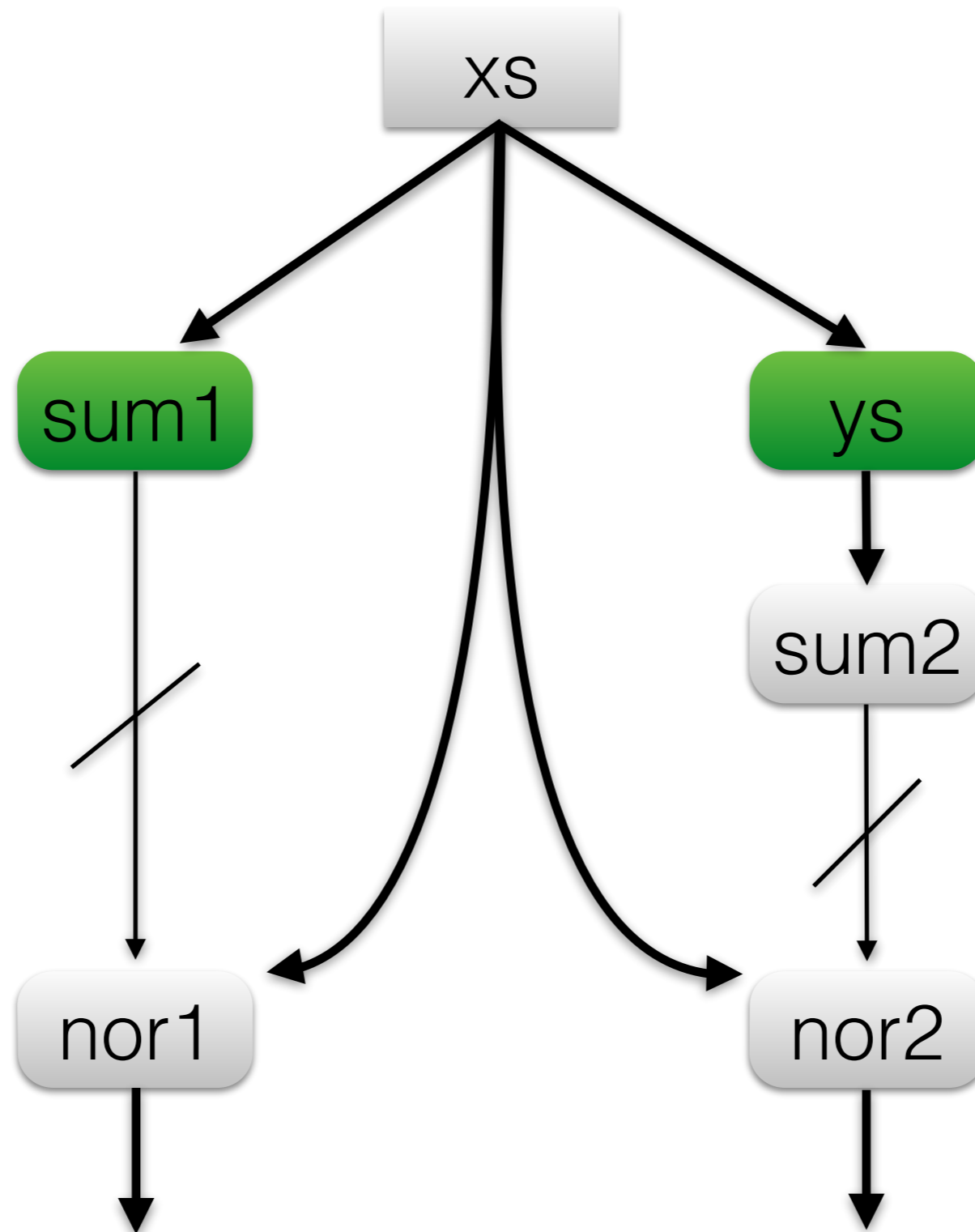
Fusible edge - unfused

$$1 f(y_s, \text{sum2}) \leq o(\text{sum2}) - o(y_s) \leq 100 f(y_s, \text{sum2})$$

$$1 \leq o(\text{sum2}) - o(y_s) \leq 100$$

$$o(\text{sum2}) > o(y_s)$$

No edge



No edge

if $f(\text{sum1}, ys) = 0$

then $o(\text{sum1}) = o(ys)$

No edge

$$-100f(\text{sum1}, ys) \leq o(ys) - o(\text{sum1}) \leq 100f(\text{sum1}, ys)$$

No edge - fused

$$-100f(\text{sum1}, ys) \leq o(ys) - o(\text{sum1}) \leq 100f(\text{sum1}, ys)$$

$$0 \leq o(ys) - o(\text{sum1}) \leq 0$$

$$o(ys) = o(\text{sum1})$$

No edge - unfused

$$-100f(\text{sum1}, ys) \leq o(ys) - o(\text{sum1}) \leq 100f(\text{sum1}, ys)$$

$$-100 \leq o(ys) - o(\text{sum1}) \leq 100$$

All together

$$\begin{aligned} \text{Minimise} \quad & 100f(\text{sum1}, \text{ys}) & + & \quad 1f(\text{sum1}, \text{sum2}) \\ & + 100f(\text{sum1}, \text{nor2}) & + & \quad 100f(\text{ys}, \text{sum2}) \\ & + 100f(\text{ys}, \text{nor1}) & + & \quad 1f(\text{sum2}, \text{nor1}) \\ & + 100f(\text{nor1}, \text{nor2}) \end{aligned}$$

Subject to

$$f(\text{sum1}, \text{ys}) \leq f(\text{sum1}, \text{sum2})$$

$$f(\text{sum2}, \text{ys}) \leq f(\text{sum1}, \text{sum2})$$

$$-100f(\text{sum1}, \text{ys}) \leq o(\text{ys}) \quad -o(\text{sum1}) \leq 100f(\text{sum1}, \text{ys})$$

$$-100f(\text{sum1}, \text{sum2}) \leq o(\text{sum2}) - o(\text{sum1}) \leq 100f(\text{sum1}, \text{sum2})$$

$$1f(\text{ys}, \text{sum2}) \leq o(\text{sum2}) - o(\text{ys}) \leq 100f(\text{ys}, \text{sum2})$$

$$-100f(\text{nor1}, \text{nor2}) \leq o(\text{nor2}) - o(\text{nor1}) \leq 100f(\text{nor1}, \text{nor2})$$

$$o(\text{sum1}) < o(\text{nor1})$$

$$o(\text{sum2}) < o(\text{nor2})$$

Result clustering

$$f(\text{sum1}, \text{ys}) = 0$$

$$f(\text{ys}, \text{sum2}) = 0$$

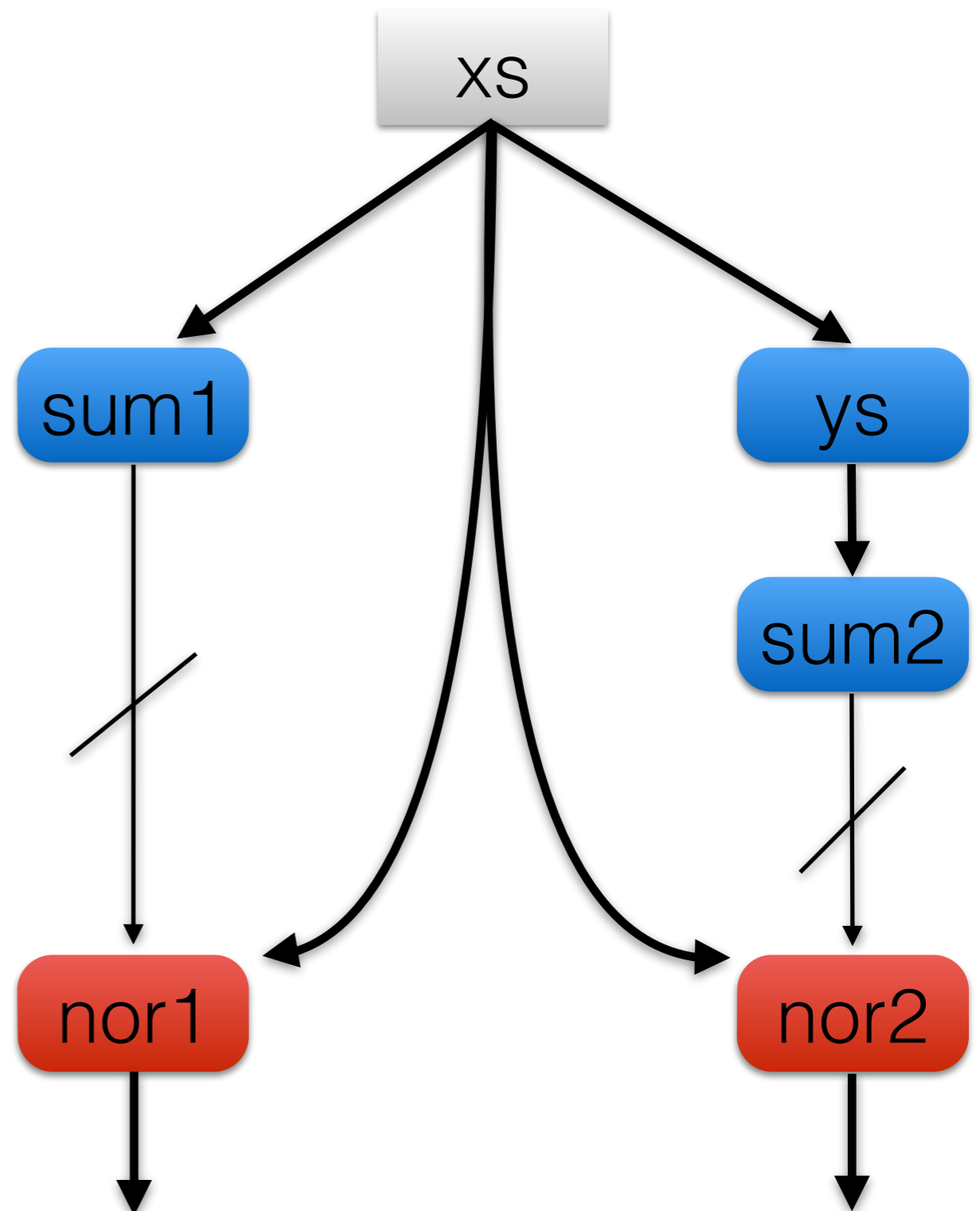
$$f(\text{sum1}, \text{sum2}) = 0$$

$$f(\text{sum1}, \text{nor2}) = 1$$

$$f(\text{ys}, \text{nor1}) = 1$$

$$f(\text{sum2}, \text{nor1}) = 1$$

$$f(\text{nor1}, \text{nor2}) = 0$$



In conclusion

- Integer linear programming isn't as scary as it sounds!
- We can fuse small (<10 combinator) programs in adequate time
- But we still need to look into large programs
- And we need to support more combinators

Timing: small programs

- Quickhull, Normalize2, Closest points, Quad tree and other test cases
- GLPK and CPLEX both took $< 100\text{ms}$.

Timing: large program

- Randomly generated with 24 combinatorics
- GLPK (open source) took $> 20\text{min}$
- COIN/CBC (open source) took 90s
- CPLEX (commercial) took $< 1\text{s}$!

References

- Megiddo 1997: Optimal weighted loop fusion for parallel programs
- Darte 1999: On the complexity of loop fusion
- Lippmeier 2013: Data flow fusion with series expressions in Haskell

Differences from Megiddo

- With *combinators* instead of loops, we have more semantic information about the program.
- Which lets us recognise size-changing operations like filters, and fuse together.

Future work

- Currently only a few combinators: *map*, *map2*, *filter*, *fold*, *gather (bpermute)*, *cross product*
- Need to support: *length*, *reverse*, *append*, *segmented fold*, *segmented map*, *segmented...*