An Intuition for Propagators

George Wilson

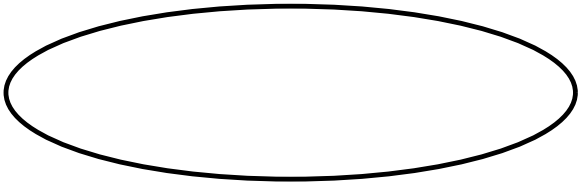
CSIRO's Data61

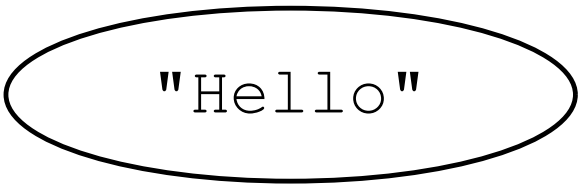
george.wilson@data61.csiro.au

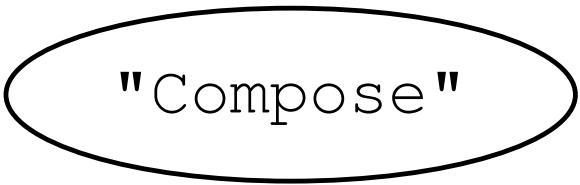
2nd September 2019

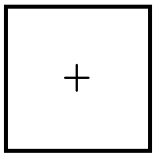
1970s, MIT

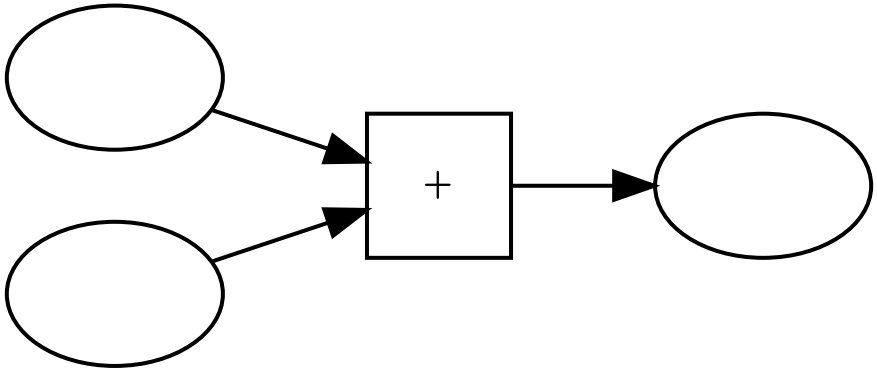
a model of computation for **highly parallel** machines

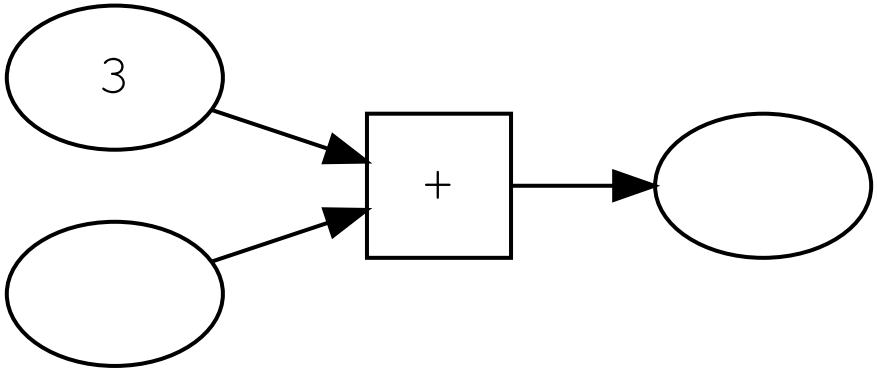


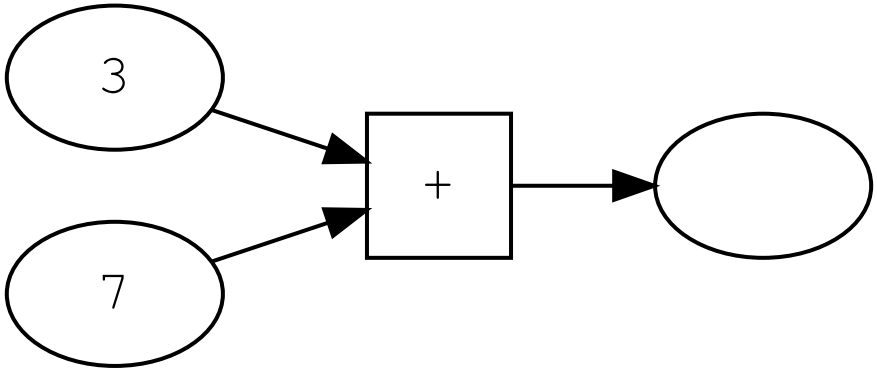

"Hello"

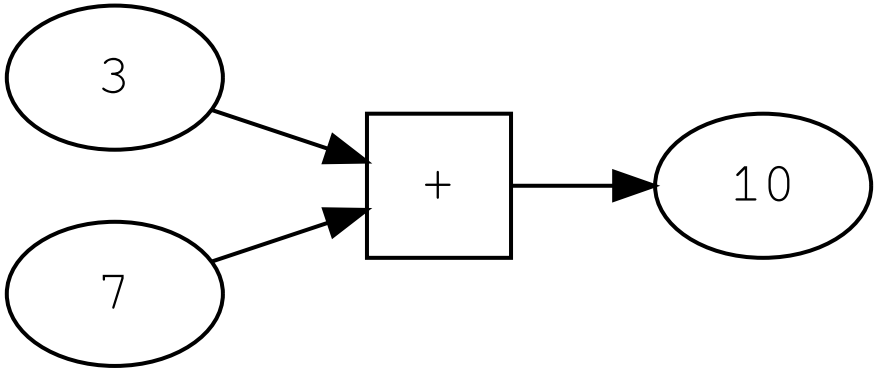

"Compose"

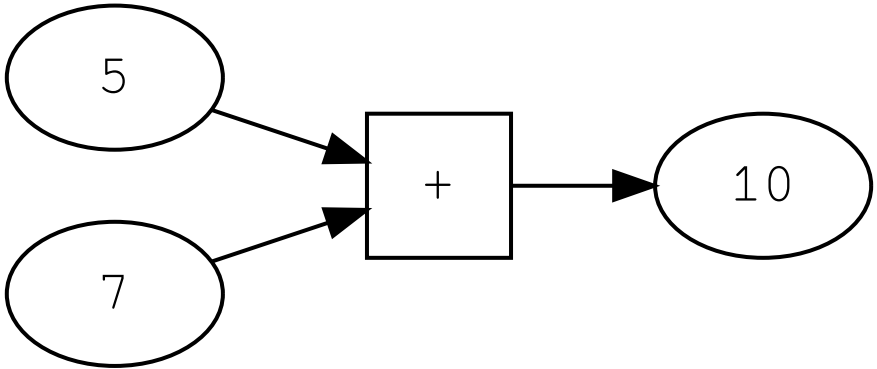


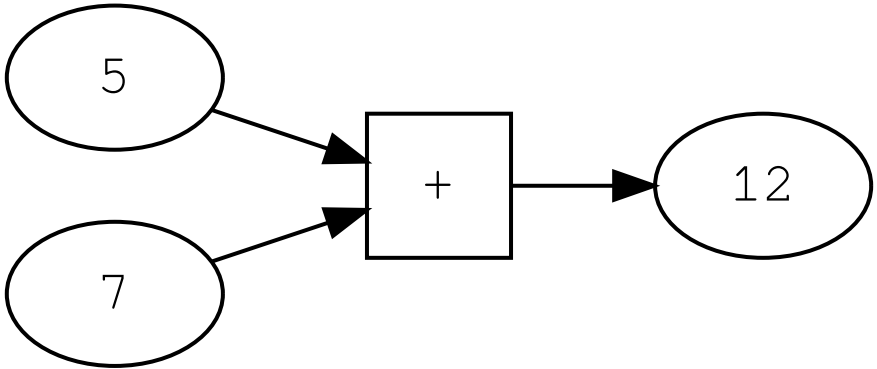



3
+


3
7
+


3
7
+
10


5
7
+
10

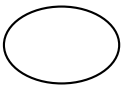

5
7
+
12

```haskell
-- types
data Par a
instance Monad Par

data Cell a
```

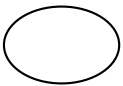
```haskell
-- types
data Par a
instance Monad Par

data Cell a


-- Creating a cell
cell    :: Par (Cell a)
```

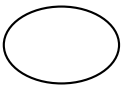
```haskell
-- types
data Par a
instance Monad Par

data Cell a


-- Creating a cell
cell    :: Par (Cell a)


-- Working with Cells
content :: Cell a -> Par (Maybe a)
write   :: Cell a -> a -> Par ()
```

```haskell
-- types
data Par a
instance Monad Par

data Cell a


-- Creating a cell
cell    :: Par (Cell a)


-- Working with Cells
content :: Cell a -> Par (Maybe a)
write   :: Cell a -> a -> Par ()


-- Creating a propagator
watch   :: Cell a -> (a -> Par ()) -> Par ()
```

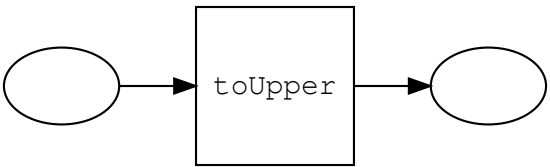


```
do
  input  <- cell
```

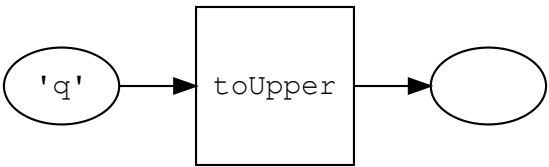




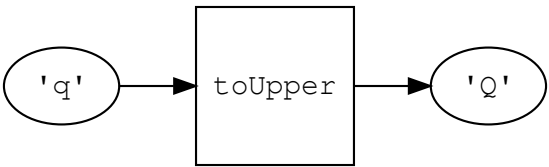
```
do
  input  <- cell
  output <- cell
```



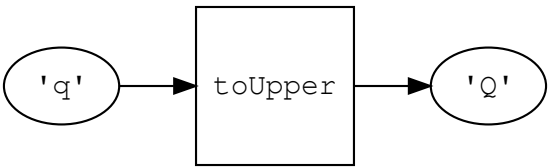

```
do
  input  <- cell
  output <- cell
  watch input (\c ->
    write output (toUpper c))
```



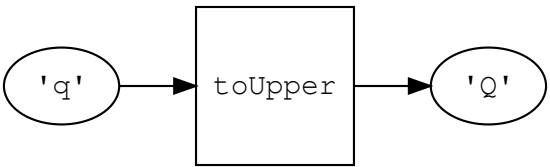

```
do
  input  <- cell
  output <- cell
  watch input (\c ->
    write output (toUpper c))
  -- run the network
  write input 'q'
  content output    -- Just 'Q'
```




```
do
  input  <- cell
  output <- cell
  watch input (\c ->
    write output (toUpper c))
  -- run the network
  write input 'q'
  content output    -- Just 'Q'
```

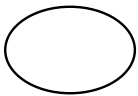


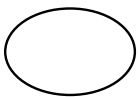

```
do
  input  <- cell
  output <- cell
  watch input (\c ->
    write output (toUpper c))
  -- run the network
  write input 'q'
  content output    -- Just 'Q'
```

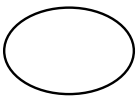


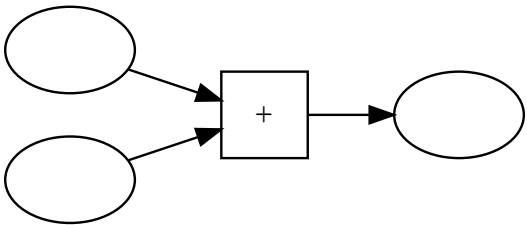

```
do
  input  <- cell
  output <- cell
  lift toUpper input output

  -- run the network
  write input 'q'
  content output    -- Just 'Q'
```

```
do
  inL  <- cell
  inR  <- cell
  out  <- cell
```

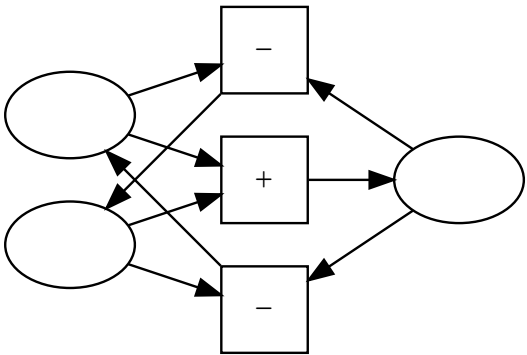




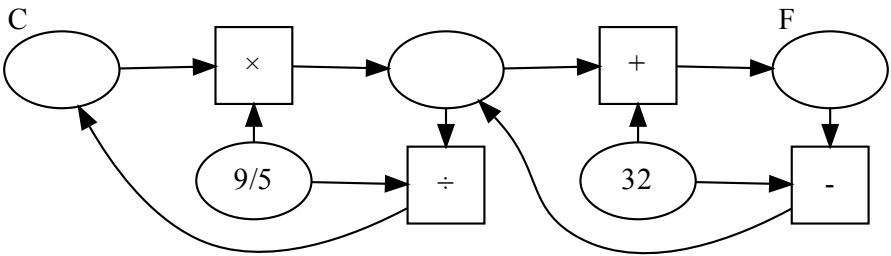


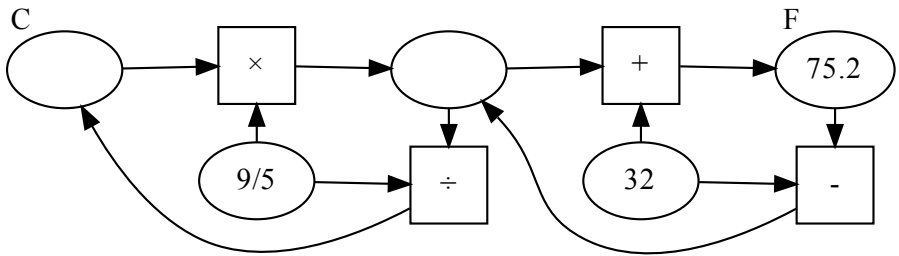


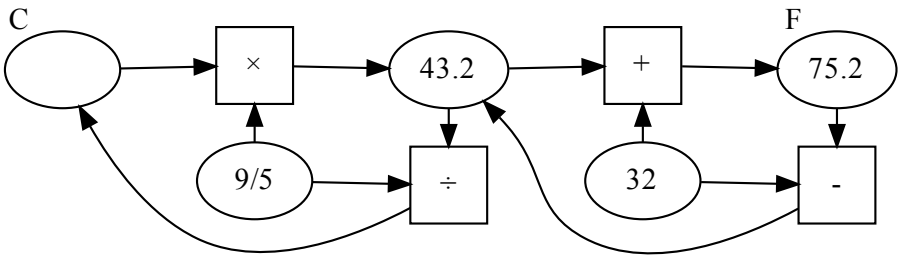

```
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  adder inL inR out

    where
      adder l r o = do
        lift2 (+) l r o
```

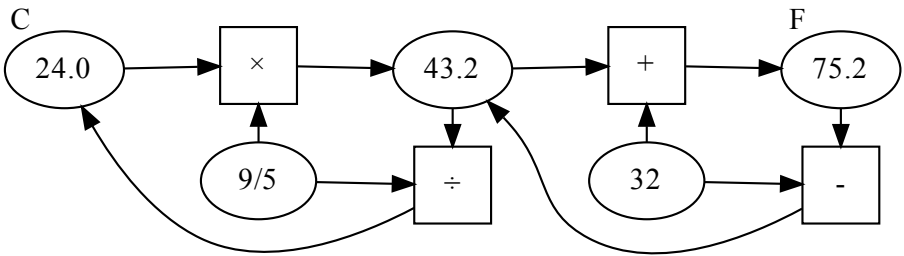


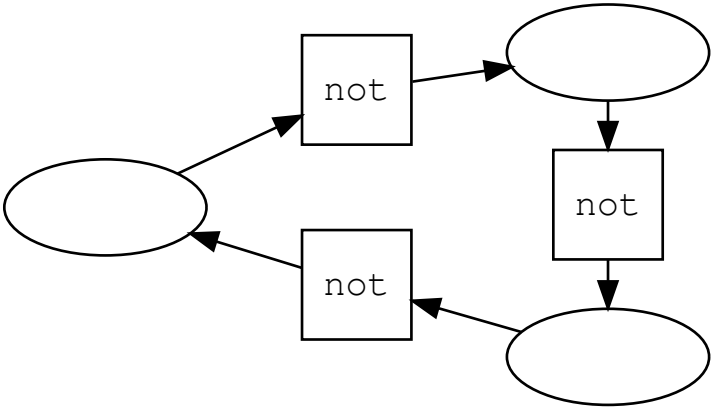

```
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  adder inL inR out

    where
      adder l r o = do
        lift2 (+) l r o
        lift2 (−) o l r
        lift2 (−) o r l
```

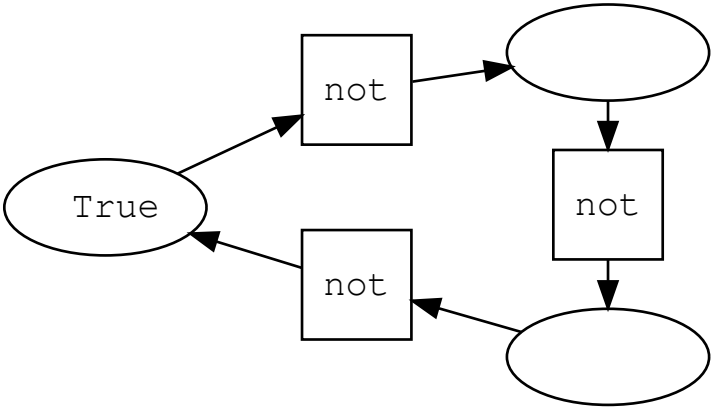

C
×
+
F
9/5
÷
32
-

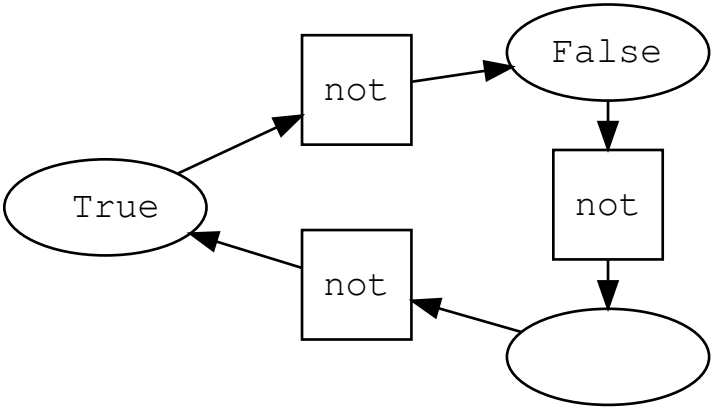

C
×
+
F
75.2
9/5
÷
32
-

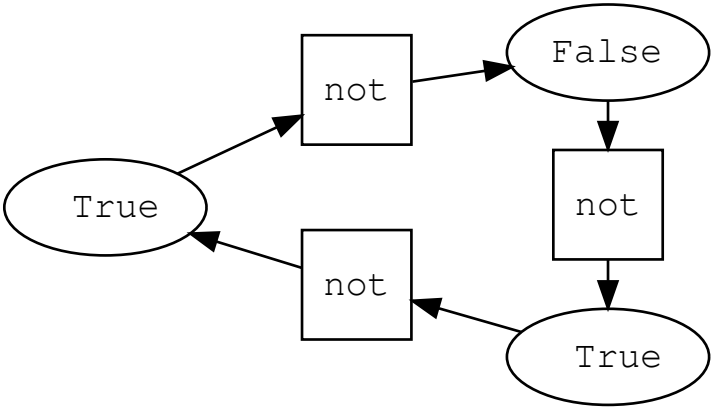

C
×
43.2
+
F
75.2
9/5
÷
32
-

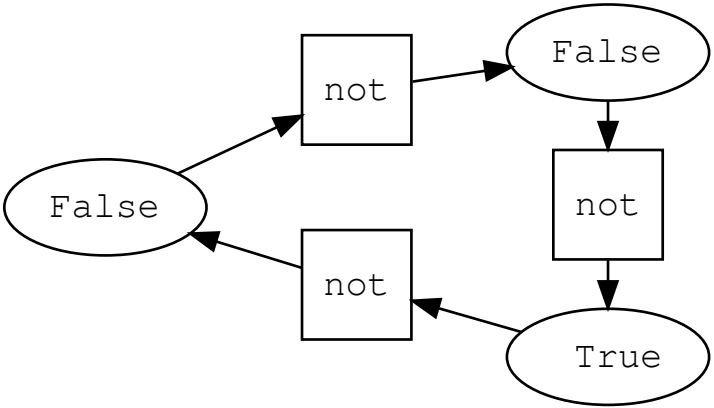

C
24.0
×
43.2
+
F
75.2
9/5
÷
32
-

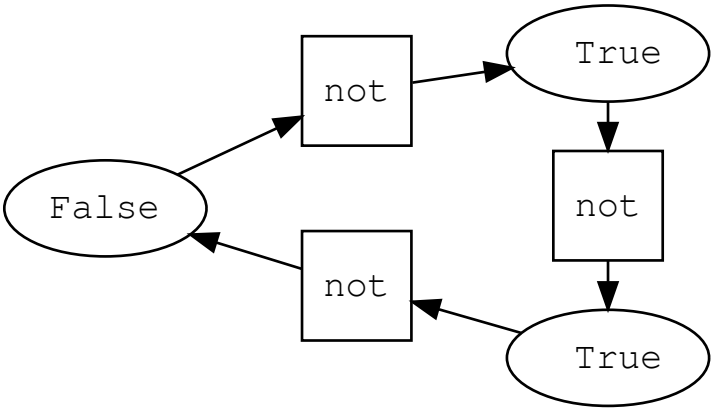

not
not
not

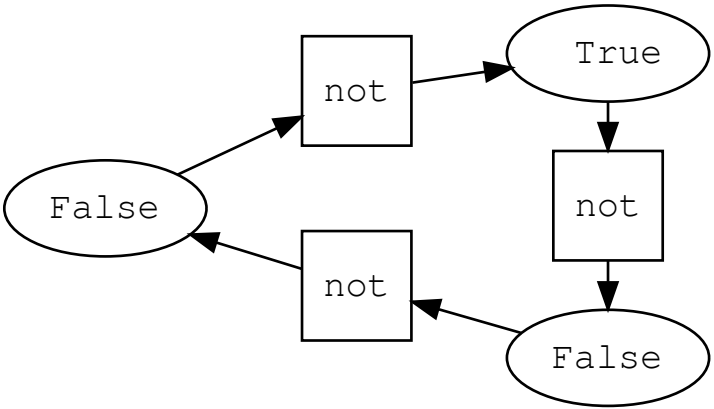

not
not
True
not

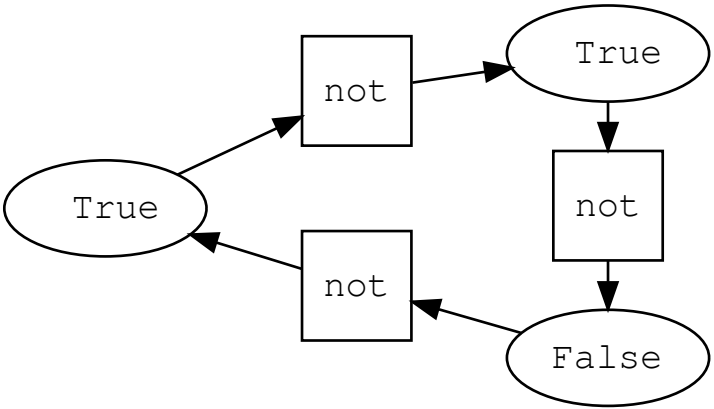

not
False
True
not
not

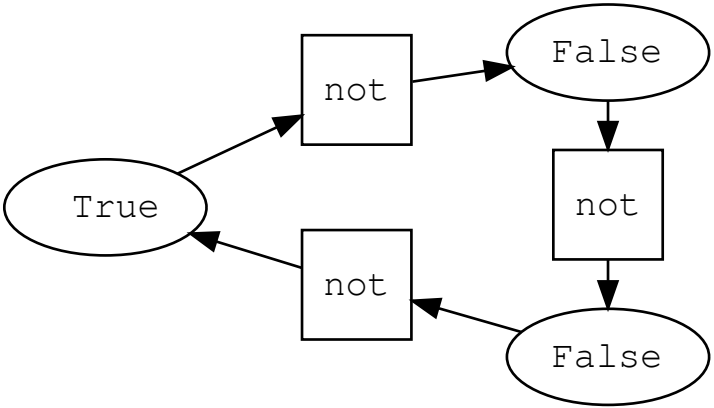

not
False
True
not
not
True


not
False
False
not
not
True


not
True
False
not
not
True


not
True
False
not
not
False


True
not
True
not
not
False


not
False
True
not
not
False

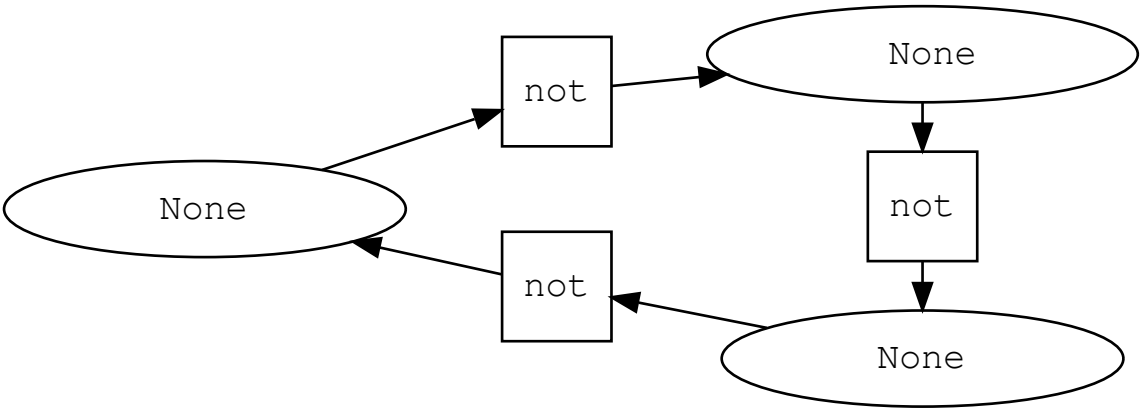
?!

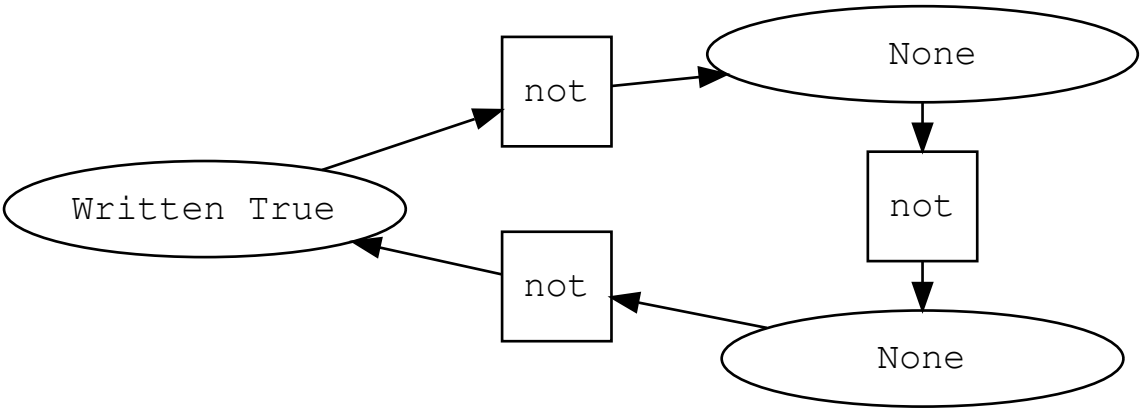
How can we fix this?

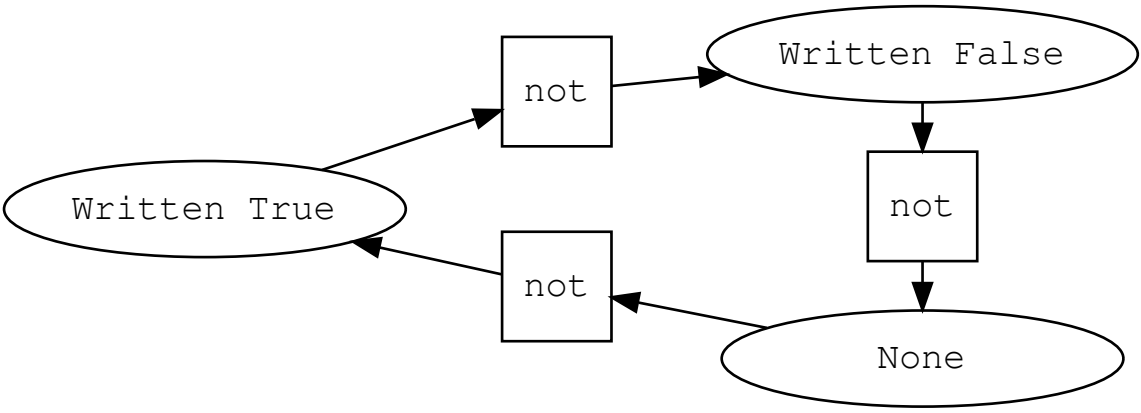
```haskell
data WriteOnce a
  = None
  | Written a
  | TooMany
```

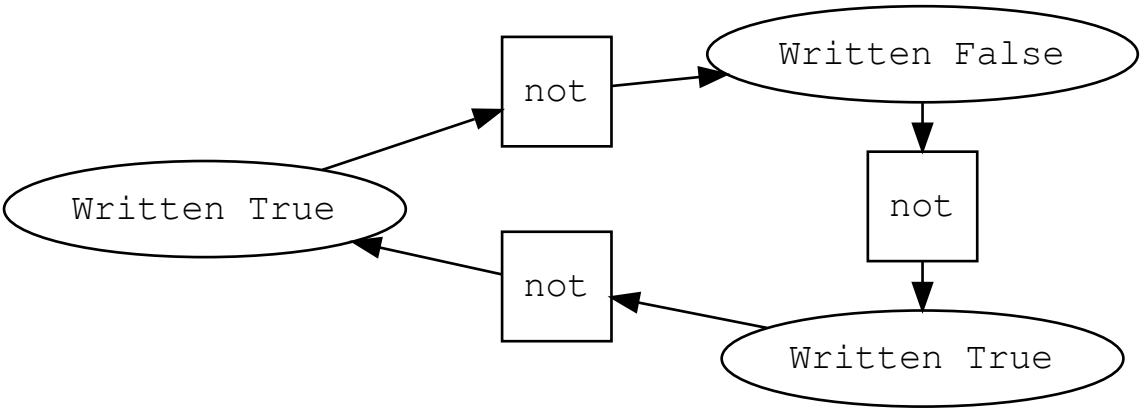
```haskell
data WriteOnce a
  = None
  | Written a
  | TooMany

tryWrite ::           a -> WriteOnce a -> WriteOnce a
tryWrite a w = case w of
  None      -> Written a
  Written b -> TooMany
  TooMany   -> TooMany
```

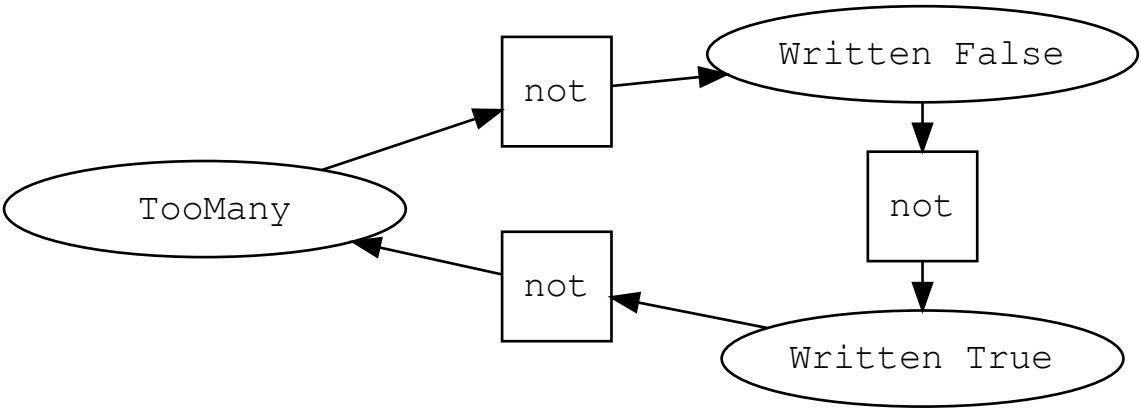
```haskell
data WriteOnce a
  = None
  | Written a
  | TooMany

tryWrite :: (Eq a) => a -> WriteOnce a -> WriteOnce a
tryWrite a w = case w of
  None      -> Written a
  Written b -> if a == b then Written b else TooMany
  TooMany   -> TooMany
```

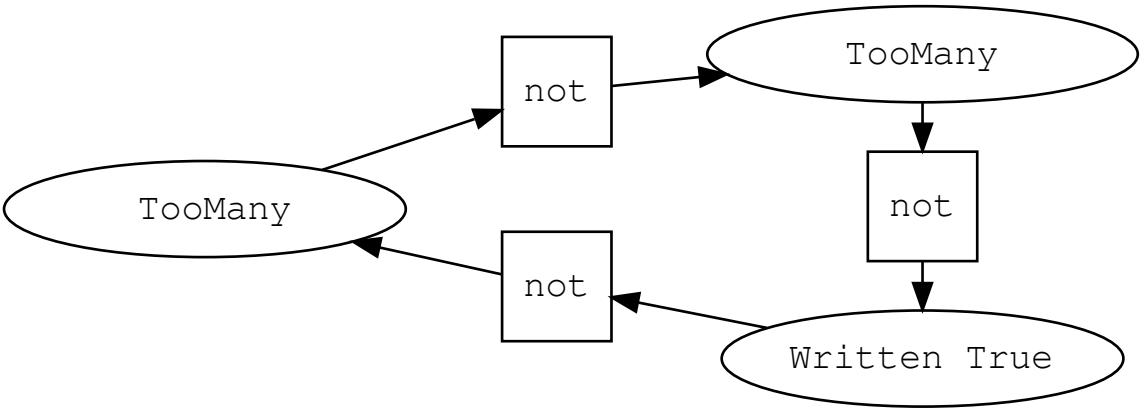

not
None
not
None
not
None

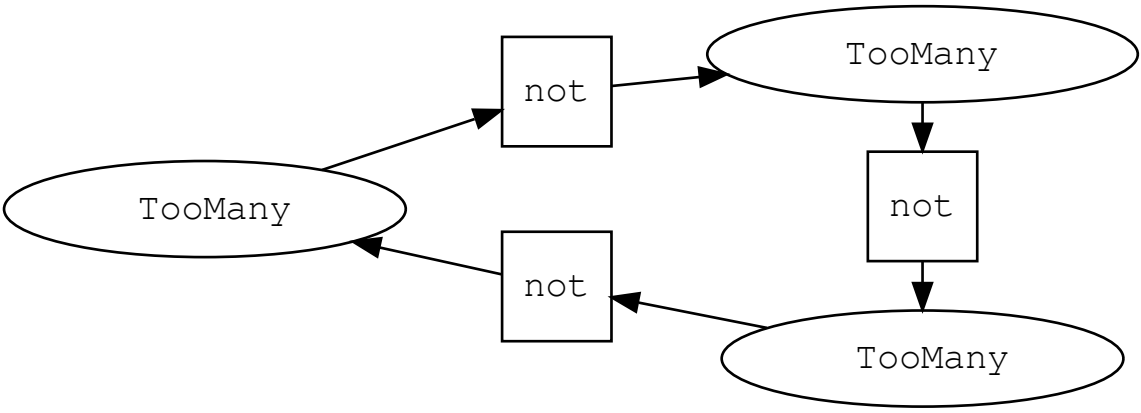

Written True
not
None
not
None
not


Written False
not
not
Written True
not
None


Written False
not
Written True
not
not
Written True


not
Written False
not
TooMany
not
Written True


not
TooMany
TooMany
not
not
Written True


not
TooMany
TooMany
not
not
TooMany

Mutability is **chaos**

WriteOnce is **rigid**

Accumulate information about a value

Accumulate information about a value

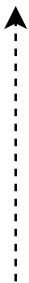
monotonically

Monotonicity

f is monotone if

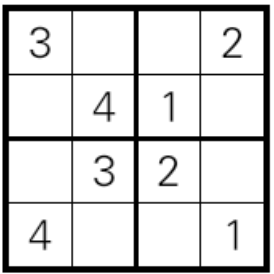
$$x \leq y \implies f(x) \leq f(y)$$

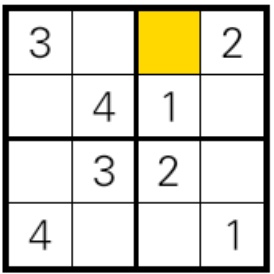
More information

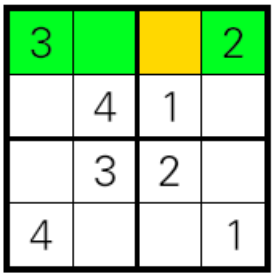


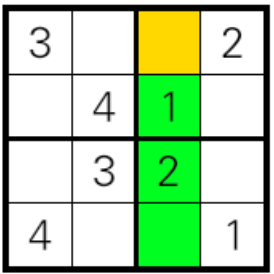
Less information

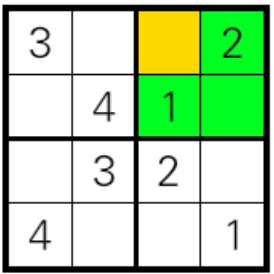
```haskell
-- I have heard contradictory answers!
  TooMany

-- I know the answer exactly
  Written a

-- I don't know anything
  None
```

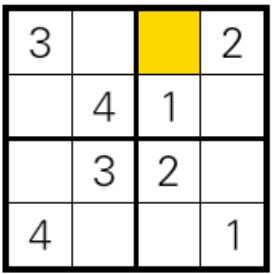

3 2
4 1
3 2
4 1

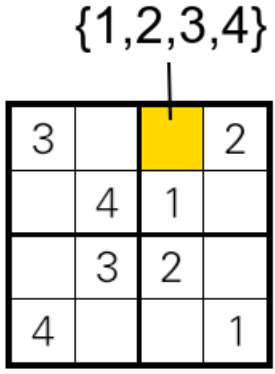

3
2
4
1
3
2
4
1

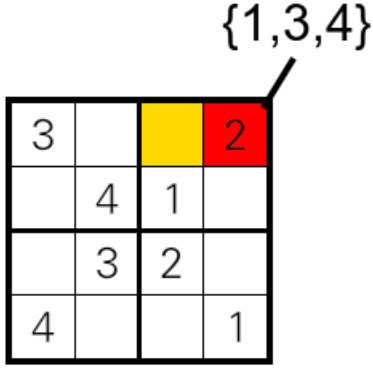

3
2
4
1
3
2
4
1

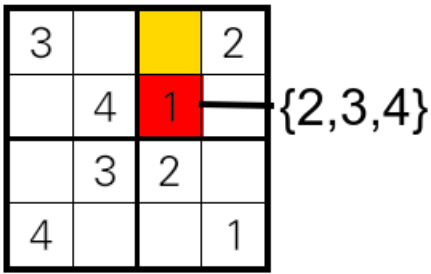

3
2
4
1
3
2
4
1

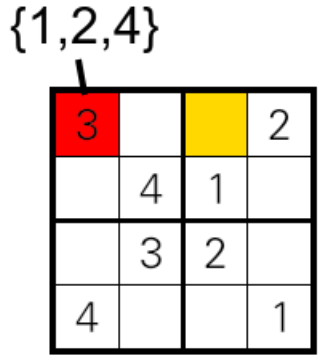

3
2
4
1
3
2
4
1


3
2
4
1
3
2
4
1


{1,2,3,4}
3
2
4
1
3
2
4
1


{1,3,4}
3
2
4
1
3
2
4
1

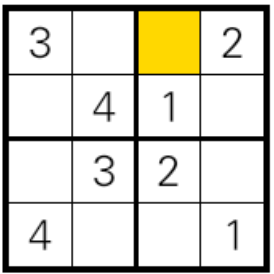

3
2
4
1
{2,3,4}
3
2
4
1

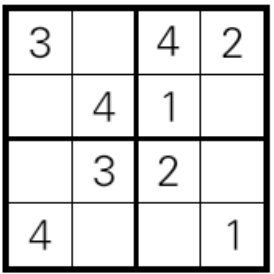

{1,2,4}
3
2
4
1
3
2
4
1

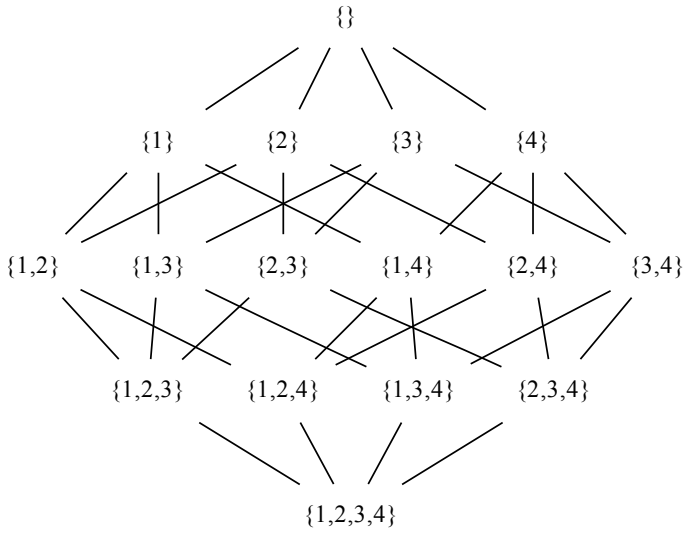
$$\{2,3,4\} \cap \{1,3,4\} \cap \{1,2,4\} \cap \{1,2,3,4\}$$

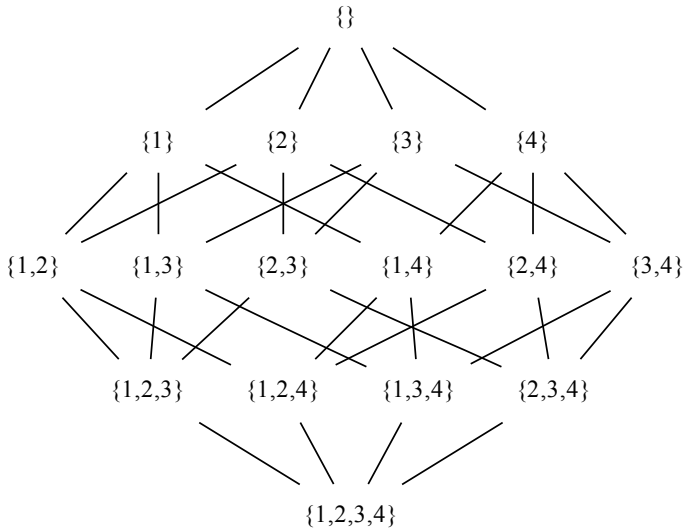
3			2
	4	1	
	3	2	
4			1

{4}



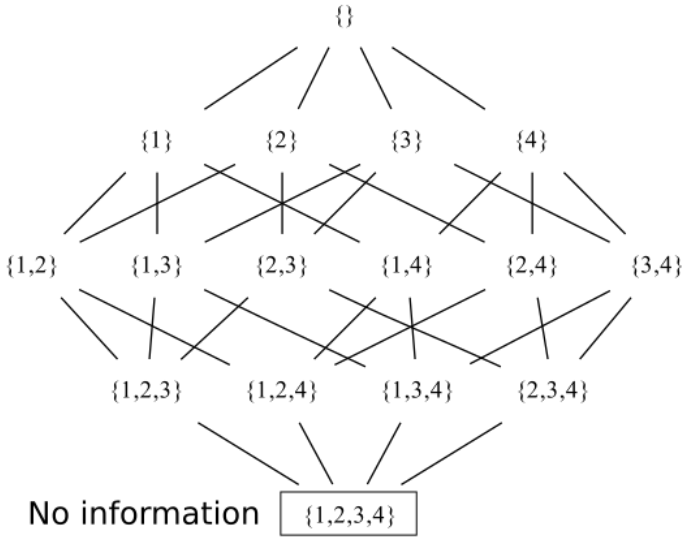

3 4 2
4 1
3 2
4 1


{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
{1,2,3,4}



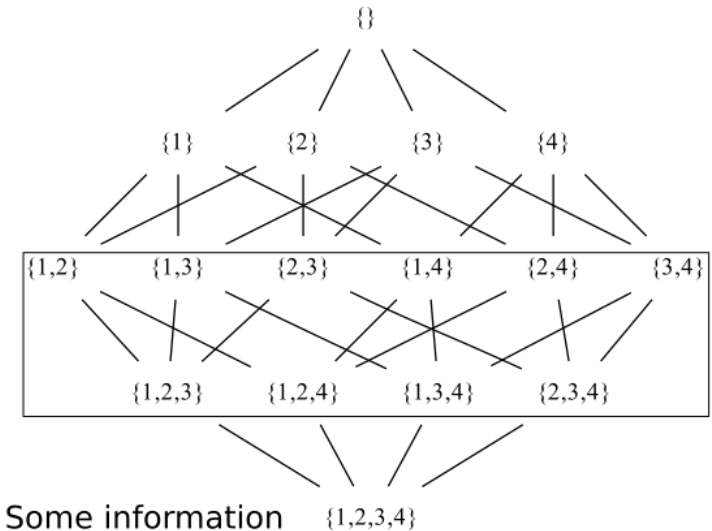
More information

Less information



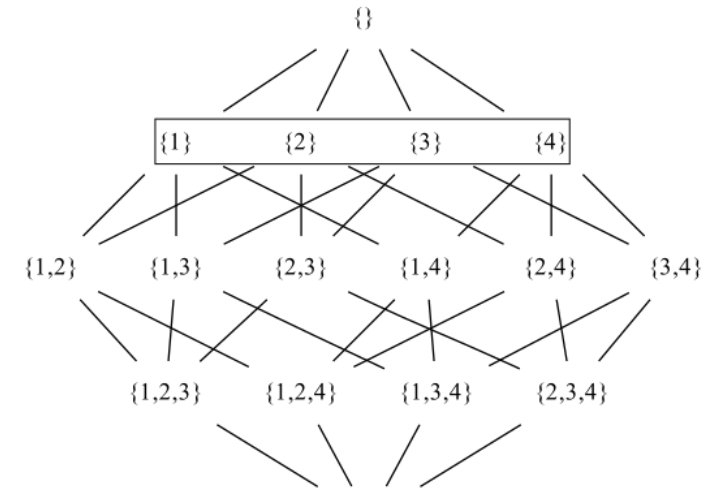
More information

Less information



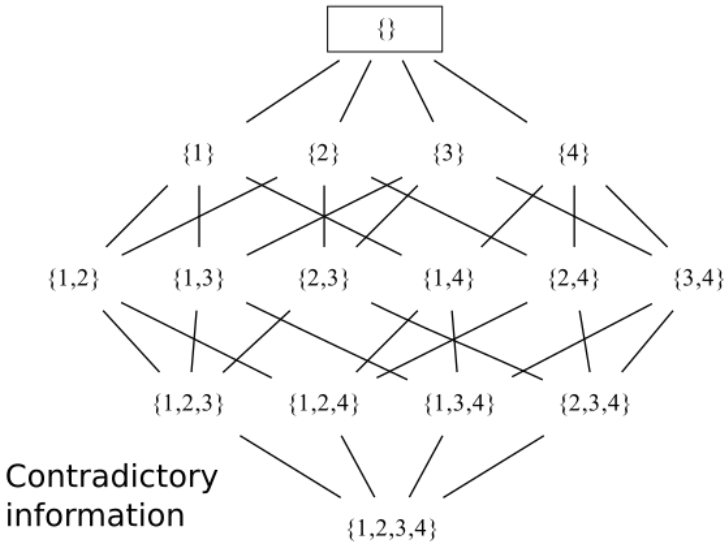
More information

Less information



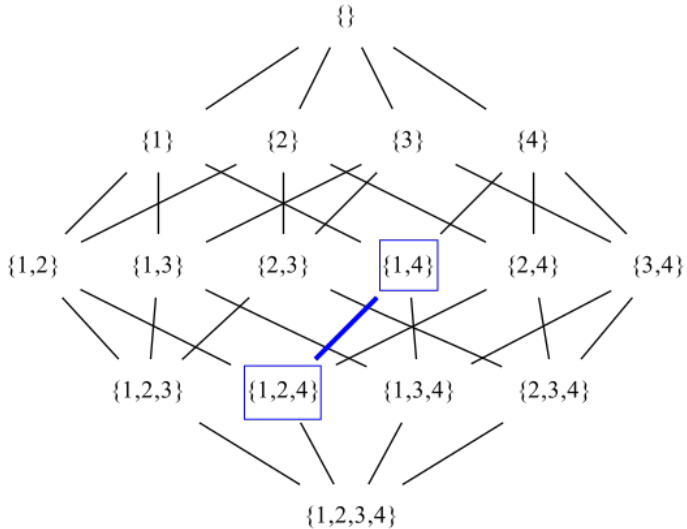
More information

Less information



More information

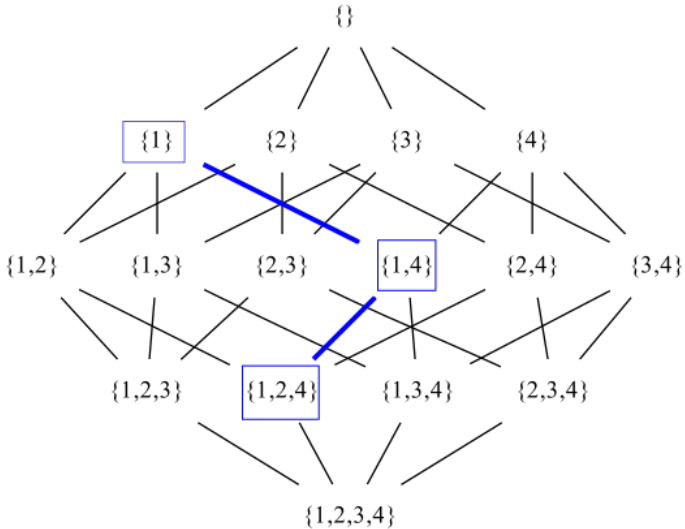
Less information



More information

Less information

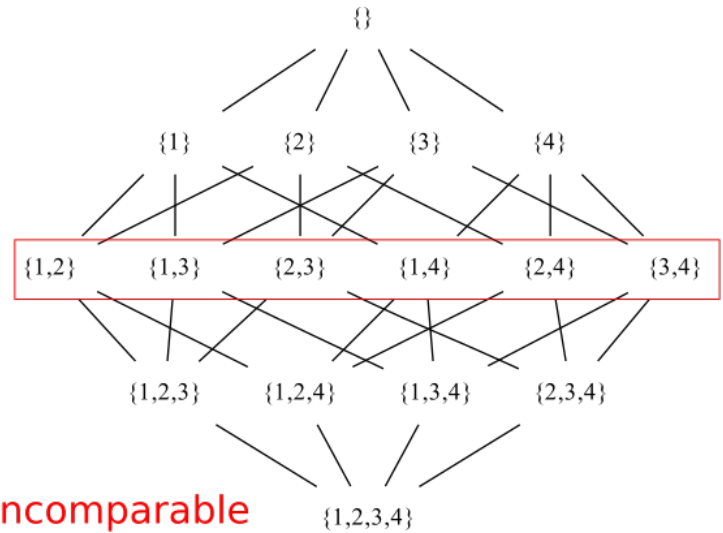
$\{1,2,4\} < \{1,4\}$



More information

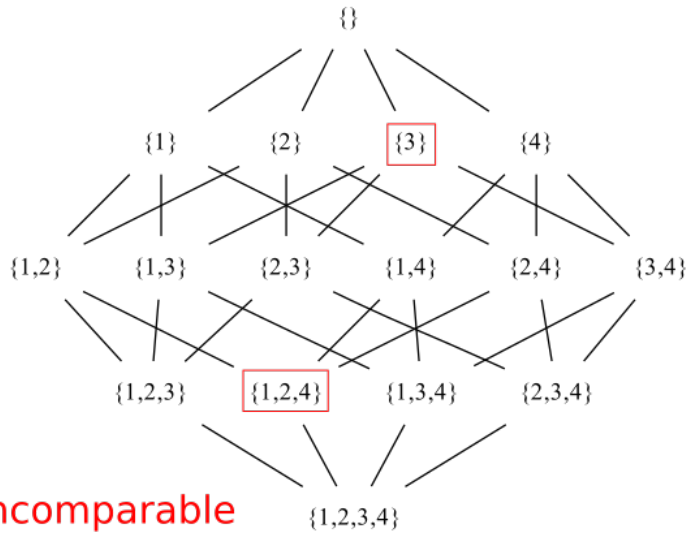
Less information

$\{1,2,4\} < \{1,4\} < \{1\}$



More information

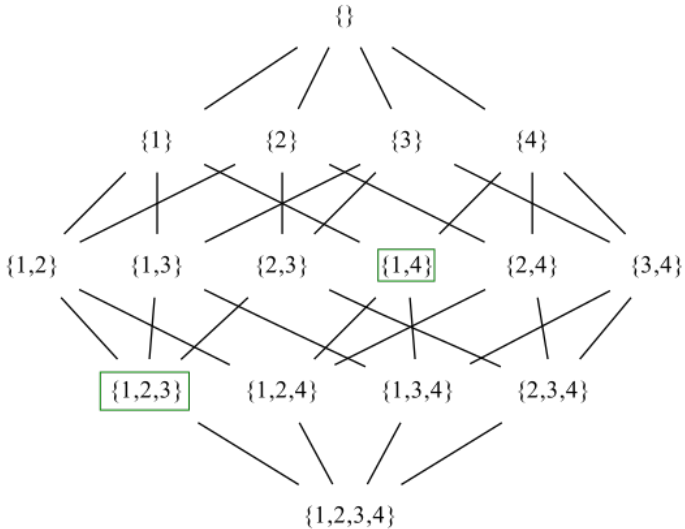
Less information



Incomparable

More information

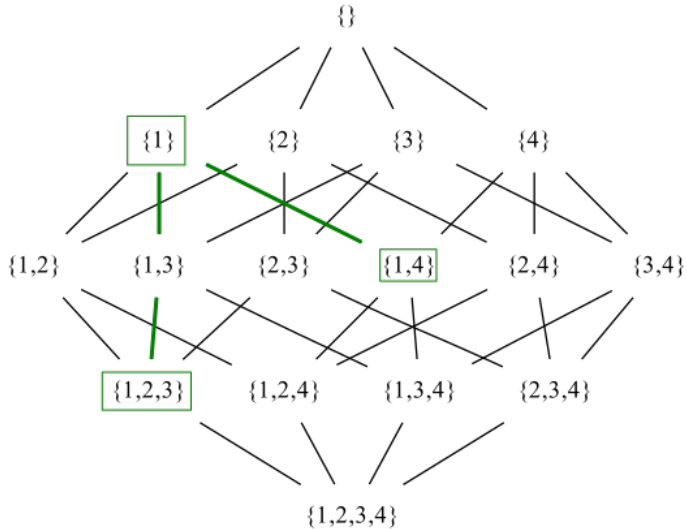
Less information



More information

Less information

$\{1,2,3\} \vee \{1,4\}$



More information

Less information

$\{1,2,3\} \vee \{1,4\} = \{1\}$

Bounded join semilattice

Identity:

$$x \vee bottom = bottom = bottom \vee x$$

Associative:

$$x \vee (y \vee z) = (x \vee y) \vee z$$

Commutative:

$$x \vee y = y \vee x$$

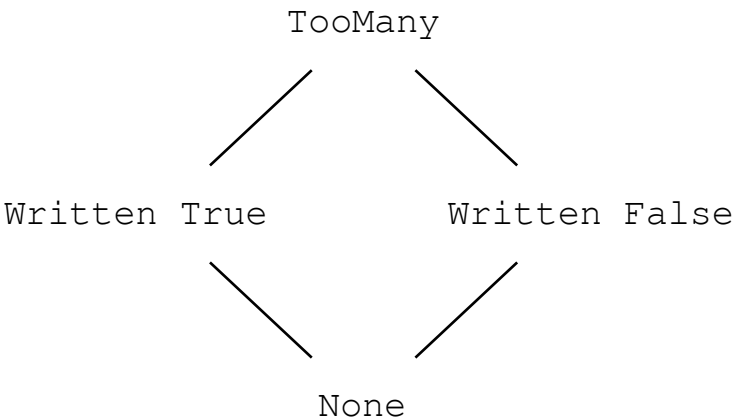
Idempotent:

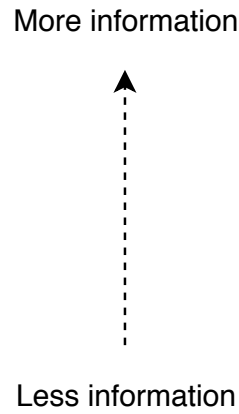
$$x \vee x = x$$

```haskell
class SemiLattice a where
  (\/)    :: a -> a -> a
  bottom :: a
```

```haskell
class SemiLattice a where
  (\/)    :: a -> a -> a
  bottom :: a


data SudokuVal = One | Two | Three | Four
  deriving (Eq, Ord)

data Possibilities = P (Set SudokuVal)
```

```haskell
class SemiLattice a where
  (\/)   :: a -> a -> a
  bottom :: a

data SudokuVal = One | Two | Three | Four
  deriving (Eq, Ord)

data Possibilities = P (Set SudokuVal)

instance Semilattice Possibilities where
  P p \/ P q = P (Set.intersection p q)
  bottom = P (Set.fromList [One,Two,Three,Four])
```


TooMany
Written True
Written False
None


More information
Less information

Cells hold semilattices
Propagators join information in

WriteOnce
Sets (intersection or union)
Intervals
Search
Unification
many more

Thanks for listening!

(Real) code for all these examples and more:
`https://github.com/qfpl/propagator-examples`