An Intuition for Propagators

George Wilson

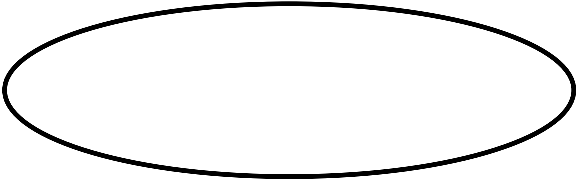
CSIRO's Data61

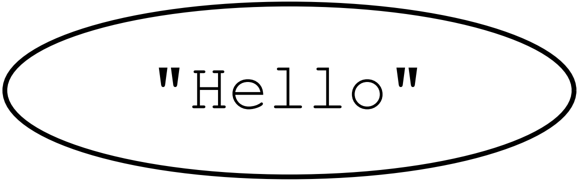
george.wilson@data61.csiro.au

2nd September 2019

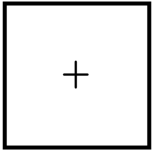
1970s, MIT

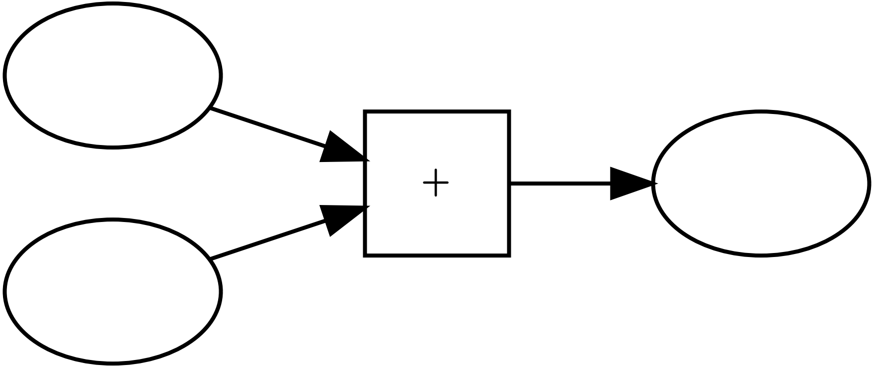
a model of computation for **highly concurrent** machines

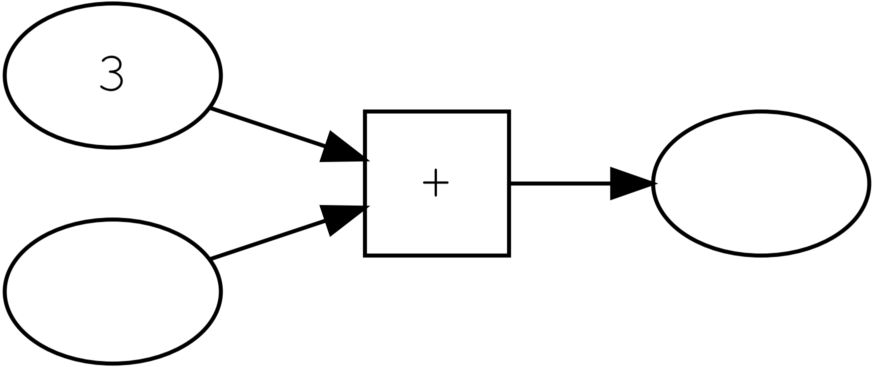


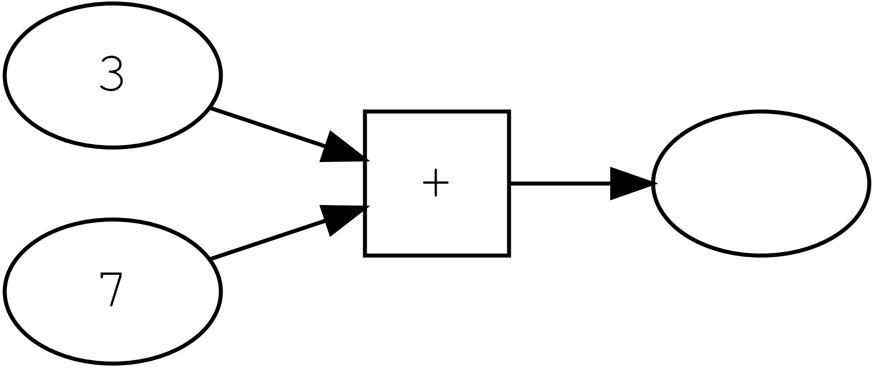

"Hello"

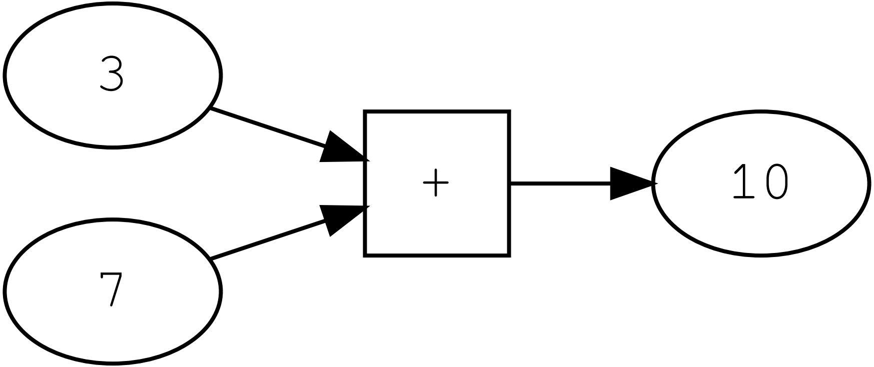
"Compose"

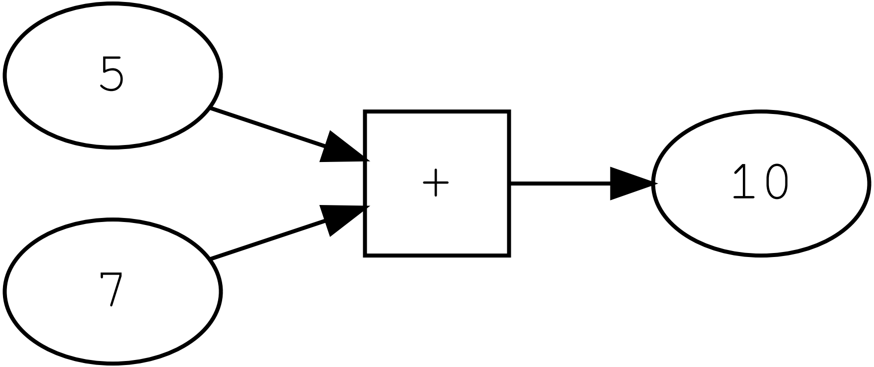


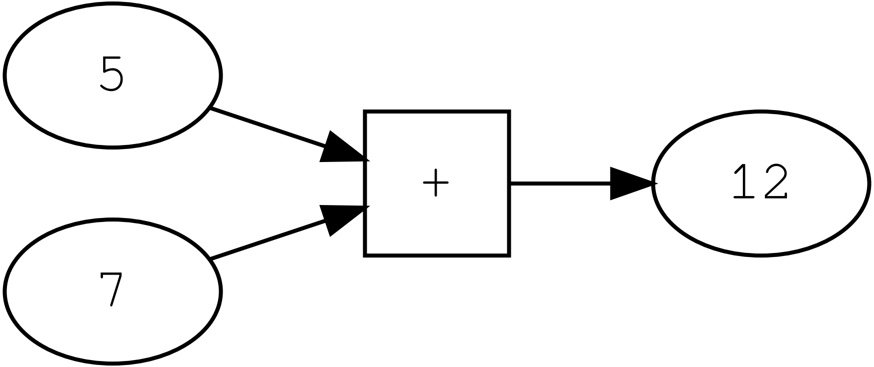



3
+


3
7
+


3
7
+
10


5
7
+
10

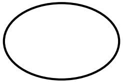

5
7
+
12

```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)
```

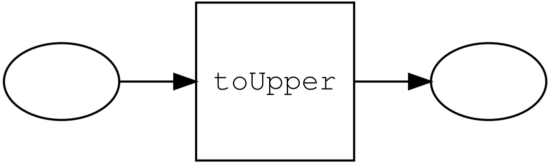
```haskell
-- types
data Par a
instance Monad Par

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 ()
```

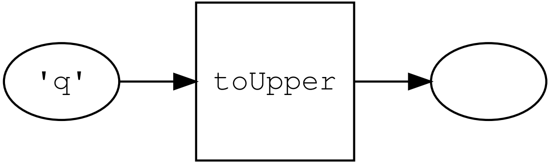


```
do
  input <- cell
```

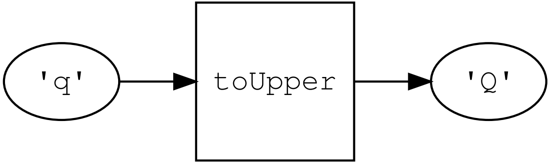
```
do
  input  <- cell
  output <- cell
```



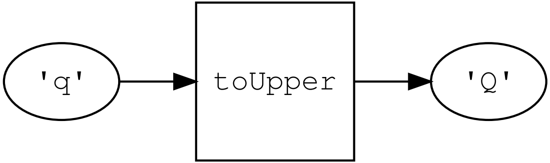

```
do
  input  <- cell
  output <- cell
  watch input (\c ->
    write output (toUpper c))
```




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

  write input 'q'
  content output   -- Just 'Q'
```

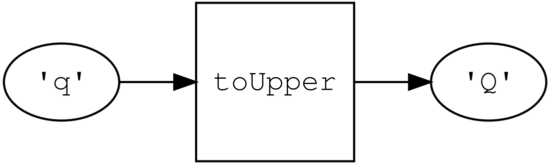



```haskell
do
  input  <- cell
  output <- cell
  watch input (\c ->
    write output (toUpper c))

  write input 'q'
  content output    -- Just 'Q'
```



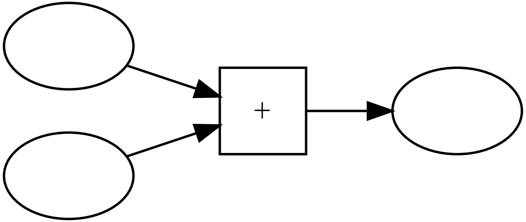

```haskell
do
  input  <- cell
  output <- cell
  watch input (\c ->
    write output (toUpper c))

  write input 'q'
  content output   -- Just 'Q'
```

```haskell
lift :: (a -> b) -> Cell a -> Cell b -> Par ()
lift f input output =
  watch input (\a ->
    write output (f a))
```

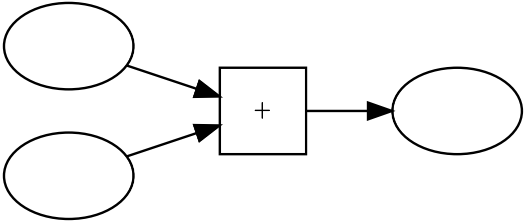



```haskell
do
  input  <- cell
  output <- cell
  lift toUpper input output

  write input 'q'
  content output   -- Just 'Q'
```

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

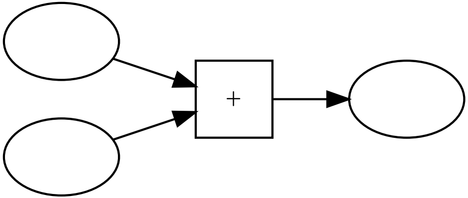



```haskell
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  watch inL (\x -> do
    maybeY <- content inR
    case maybeY of
      Nothing -> pure ()
      Just y  -> write out (x+y)
```

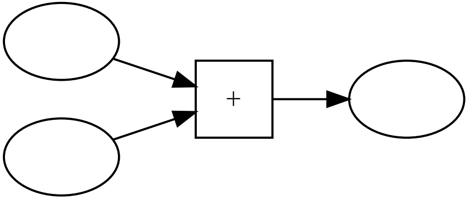



```haskell
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  watch inL (\x -> do
    maybeY <- content inR
    case maybeY of
      Nothing -> pure ()
      Just y  -> write out (x+y)

  watch inR (\y -> do
    maybeX <- content inL
    case maybeX of
      Nothing -> pure ()
      Just x  -> write out (x+y)
```

```haskell
with :: Cell a -> (a -> Par ()) -> Par ()
with theCell callback = do
  maybeA <- content theCell
    case maybeA of
      Nothing -> pure ()
      Just a  -> callback a
```



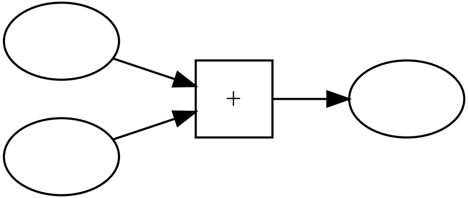

```
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  watch inL (\x ->
    with inR (\y ->
      write out (x+y)
```



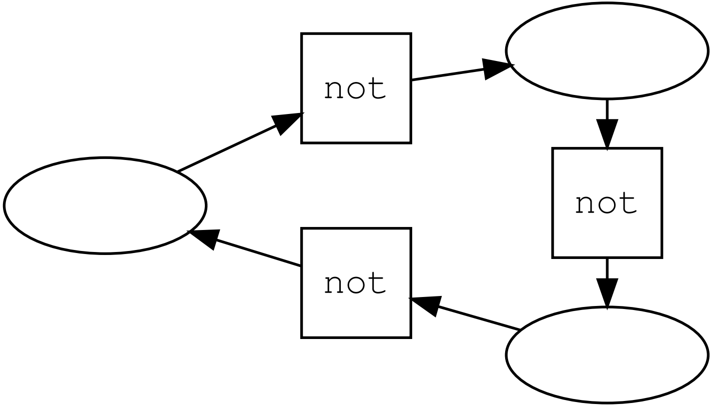

```
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  watch inL (\x ->
    with inR (\y ->
      write out (x+y)

  watch inR (\y ->
    with inL (\x ->
      write out (x+y)
```

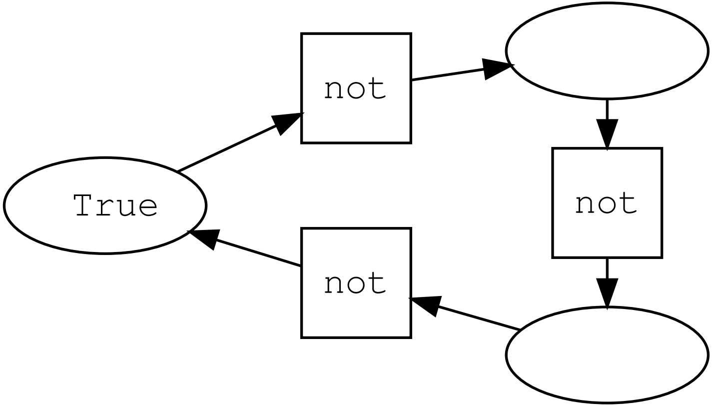
```haskell
lift2 :: (a -> b -> c)
      -> Cell a -> Cell b -> Cell c
      -> Par ()
```

```haskell
lift2 :: (a -> b -> c)
      -> Cell a -> Cell b -> Cell c
      -> Par ()
lift2 f inL inR out = do
  watch inL (\a ->
    with inR (\b ->
      write out (f a b)))
  watch inR (\b ->
    with inL (\a ->
      write out (f a b)))
```

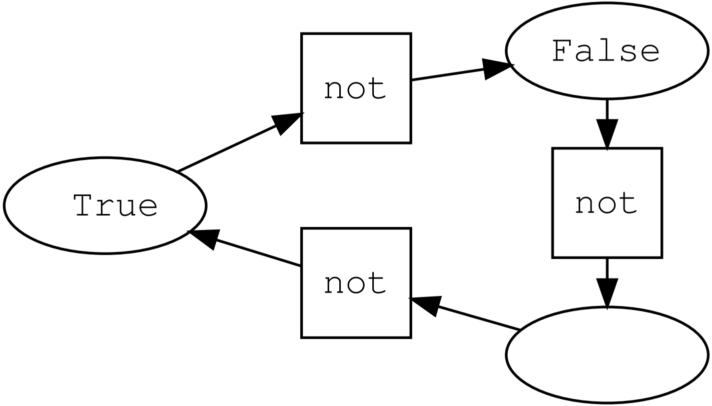



```haskell
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  adder inL inR out

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

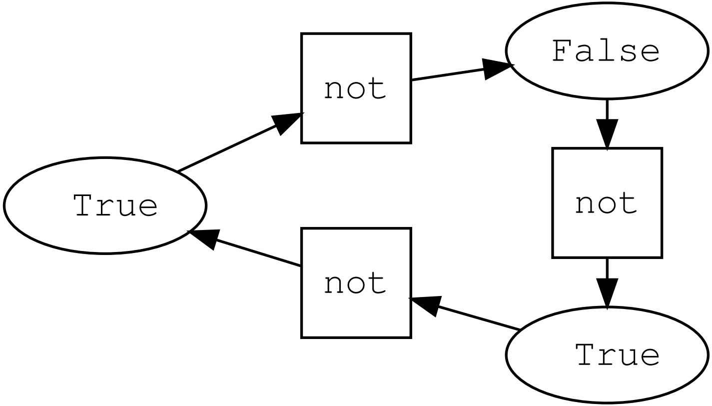

not
not
not

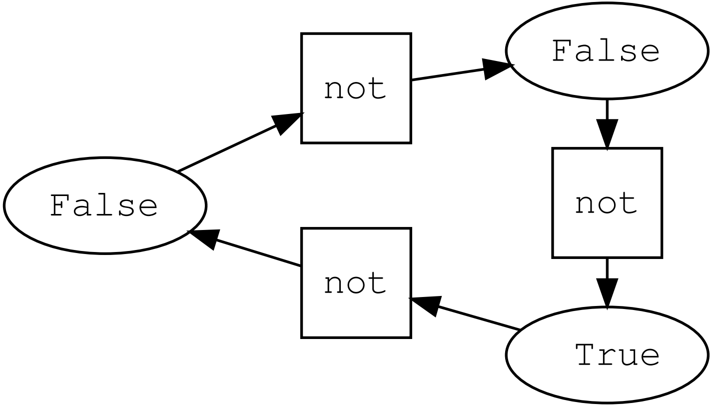


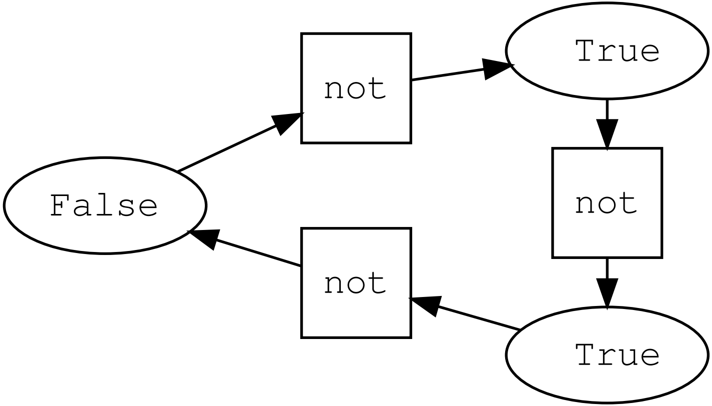
True
not
not
not

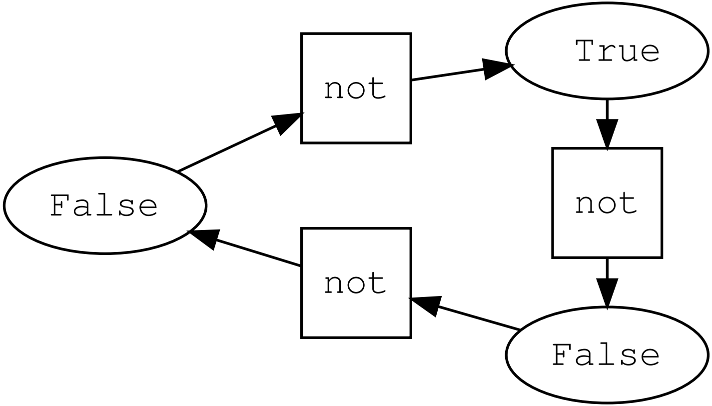


True
not
False
not
not

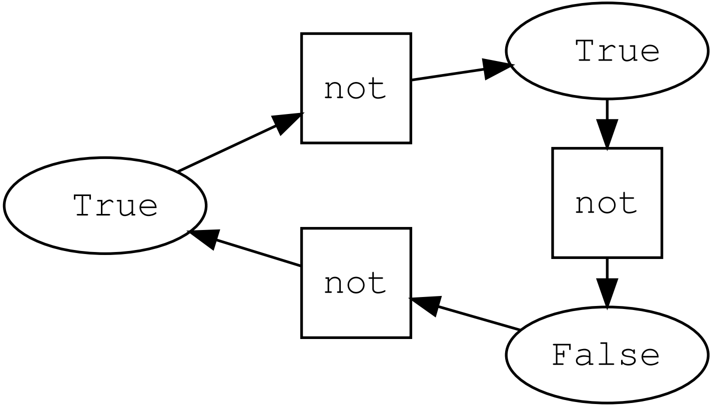

not
False
True
not
not
True

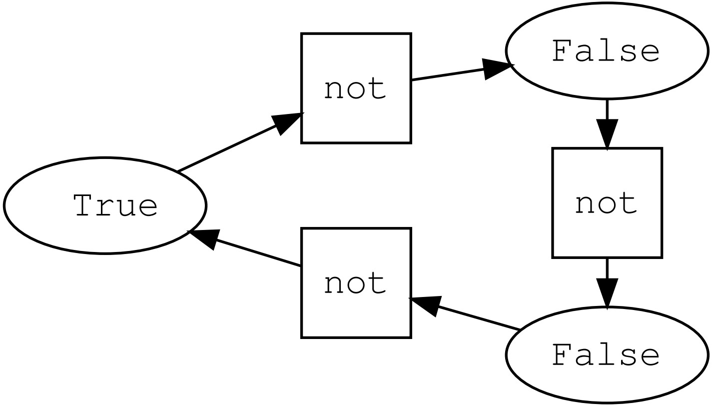

not
False
False
not
not
True


True
not
False
not
not
True



not
True
False
not
not
False


True
not
True
not
False
not


True
not
False
not
False
not

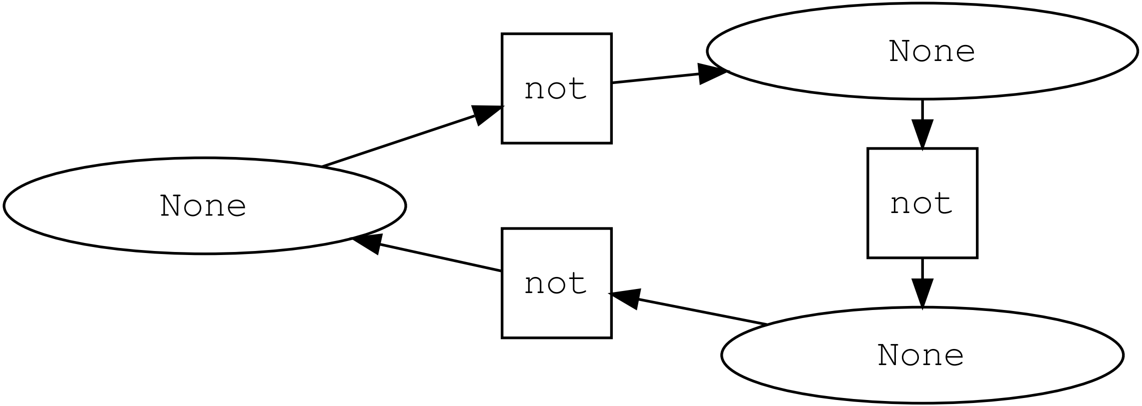
?!

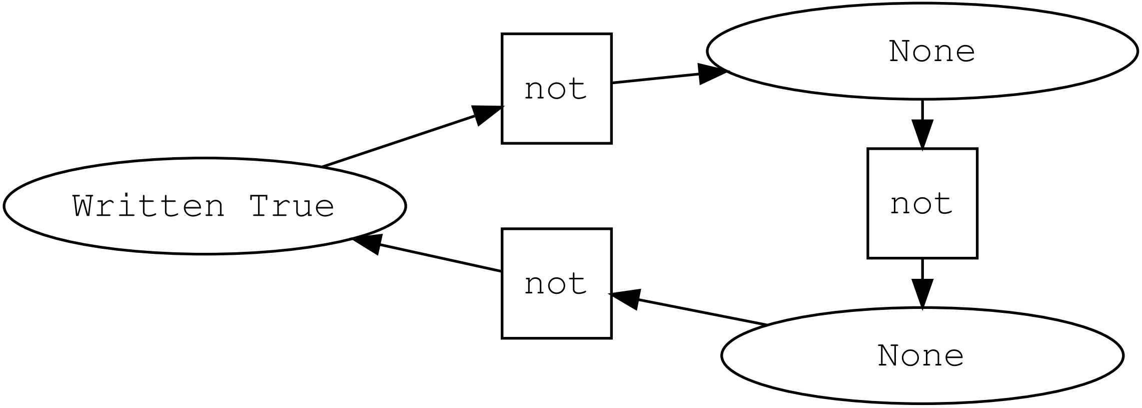
How can we fix this?

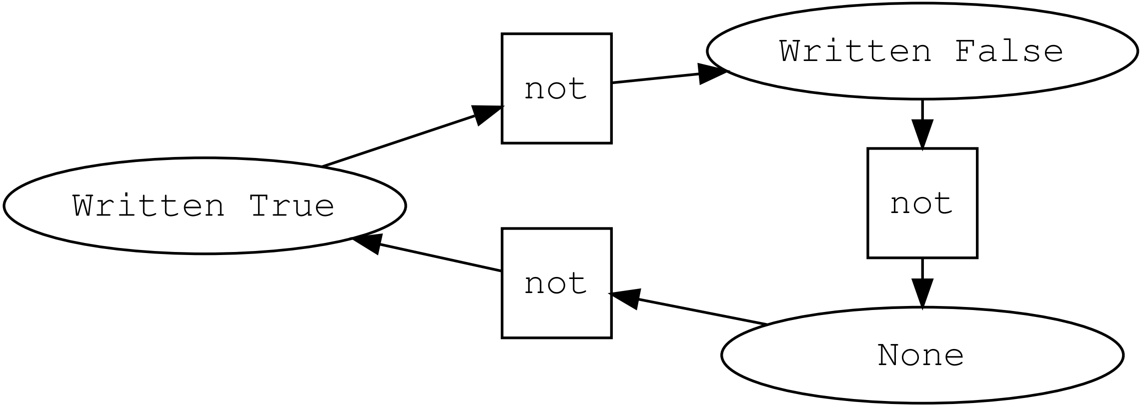
```haskell
data WriteOnce a
  = None
  | Written a
  | TooMany
```

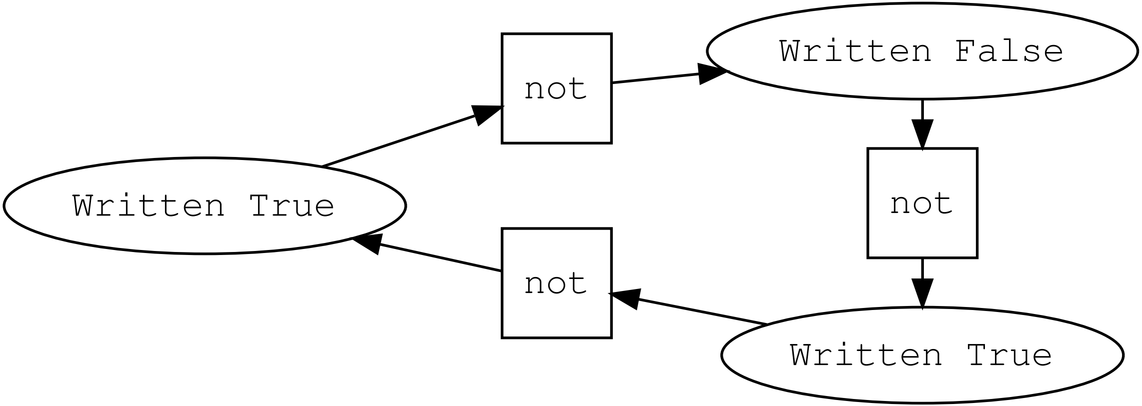
```haskell
data WriteOnce a
  = None
  | Written a
  | TooMany

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

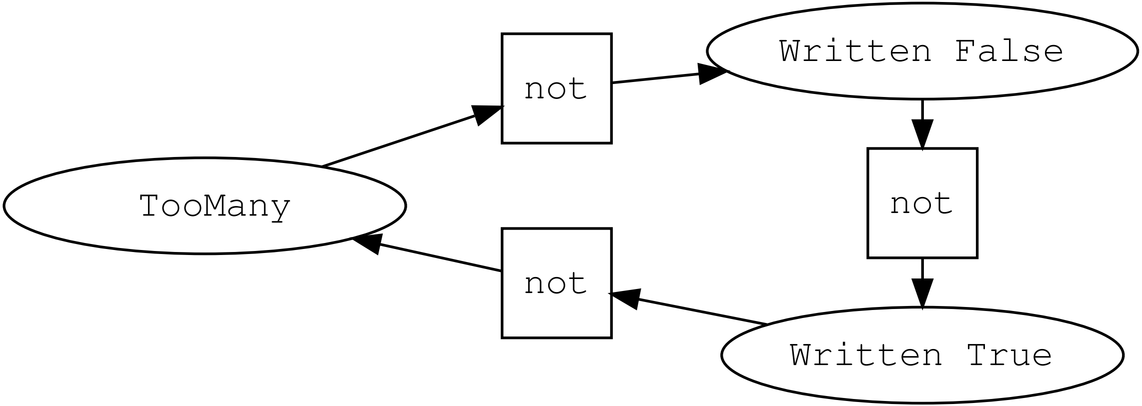
```haskell
data WriteOnce a
  = None
  | Written a
  | TooMany

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
```

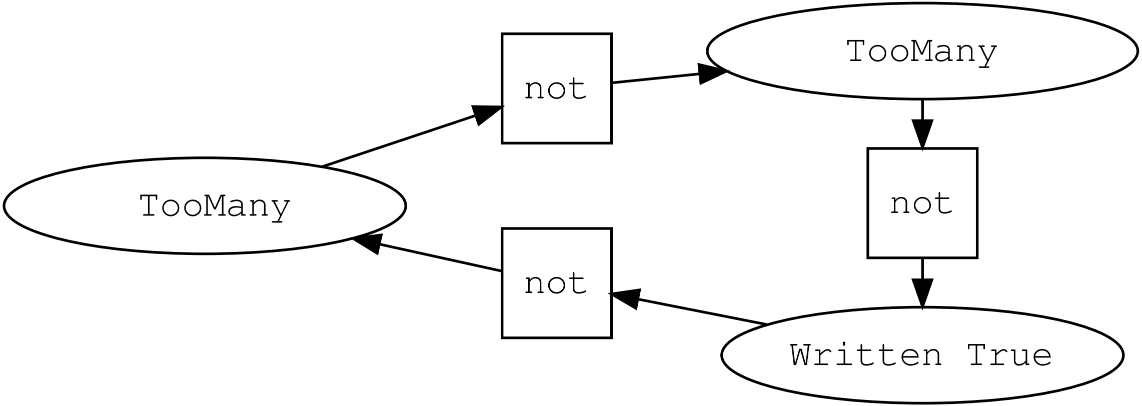

None
not
None
not
not
None

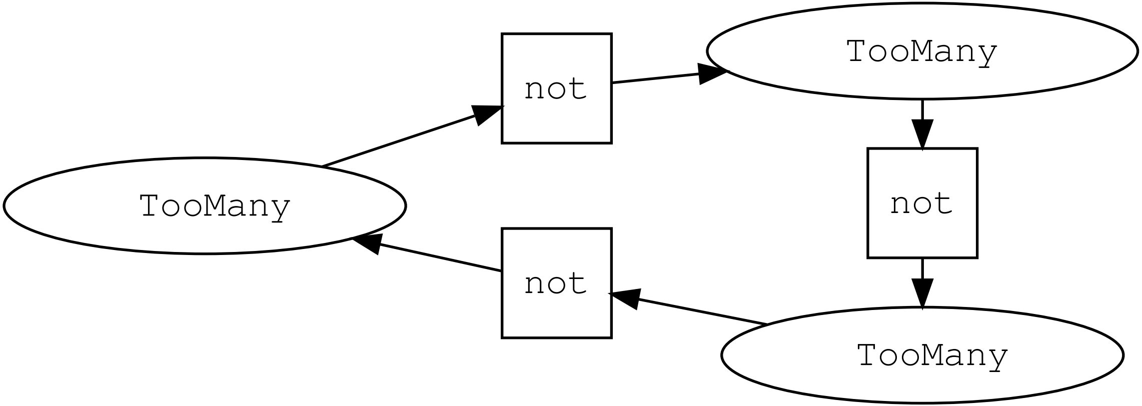

Written True
not
None
not
None
not


Written True
not
Written False
not
None
not


Written True
not
Written False
not
not
Written True


Written False
not
not
TooMany
not
Written True


TooMany
not
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

```haskell
data WriteOnce a

  -- 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

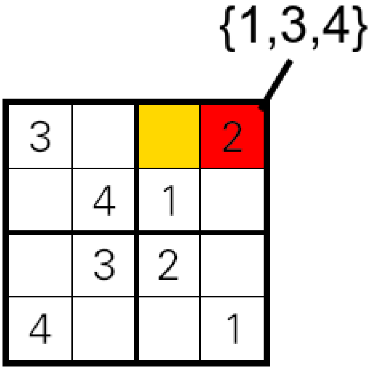
3			2
	4	1	
	3	2	
4			1

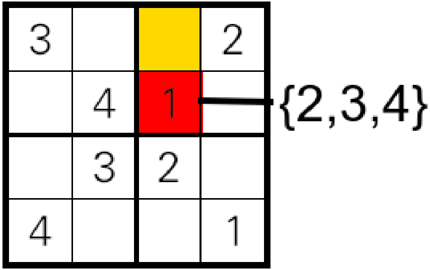
3			2
	4	1	
	3	2	
4			1

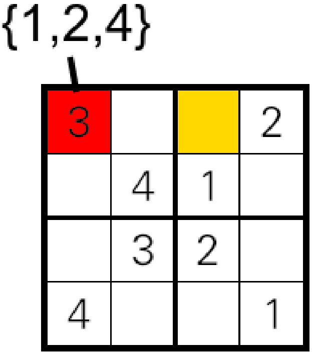
3			2
	4	1	
	3	2	
4			1

{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

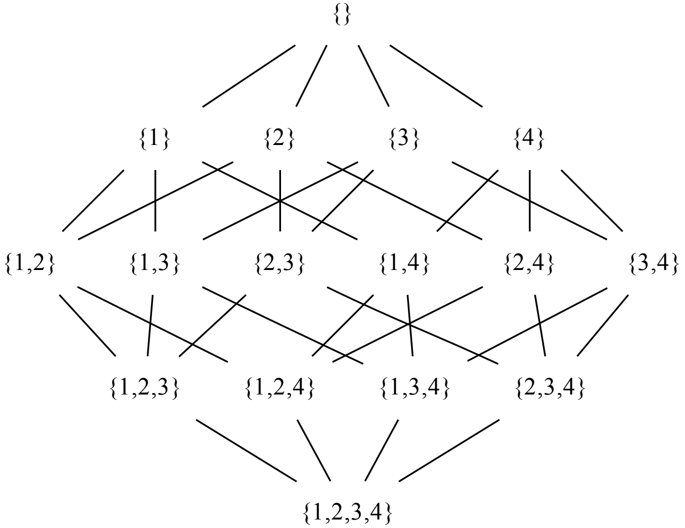
$$\{2,3,4\} \cap \{1,3,4\} \cap \{1,2,4\} \cap \{1,2,3,4\}$$

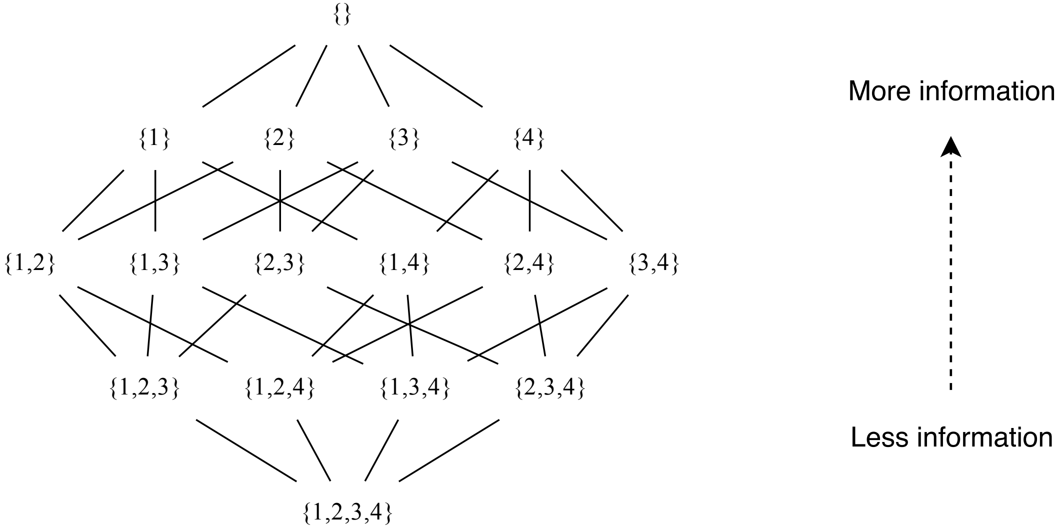
3			2
	4	1	
	3	2	
4			1

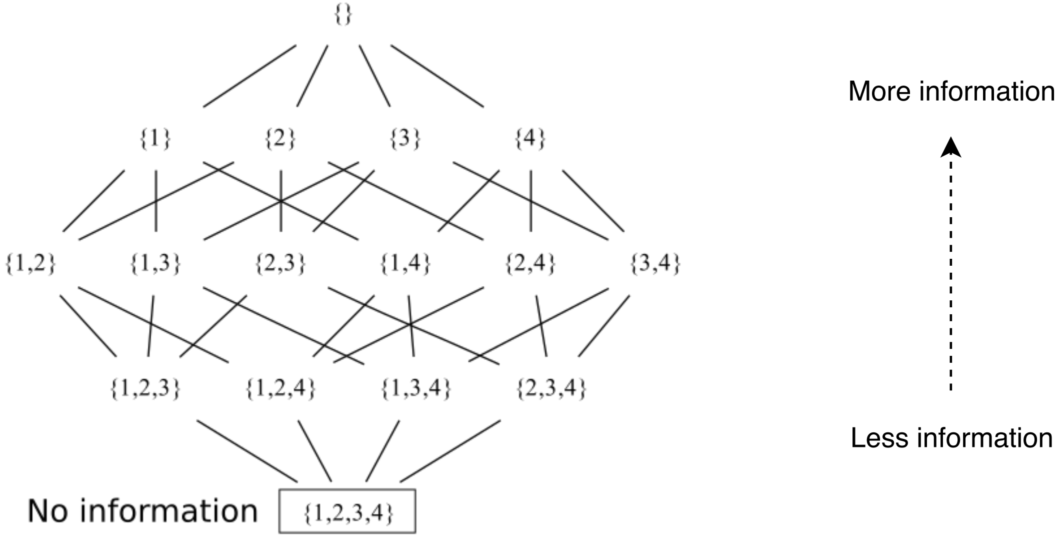
{4}

3			2
	4	1	
	3	2	
4			1

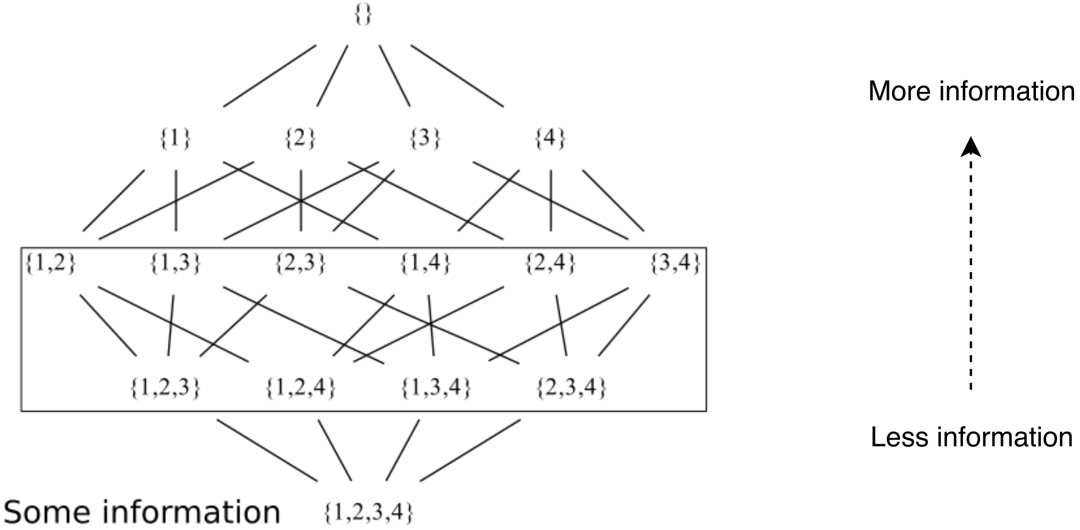
3		4	2
	4	1	
	3	2	
4			1

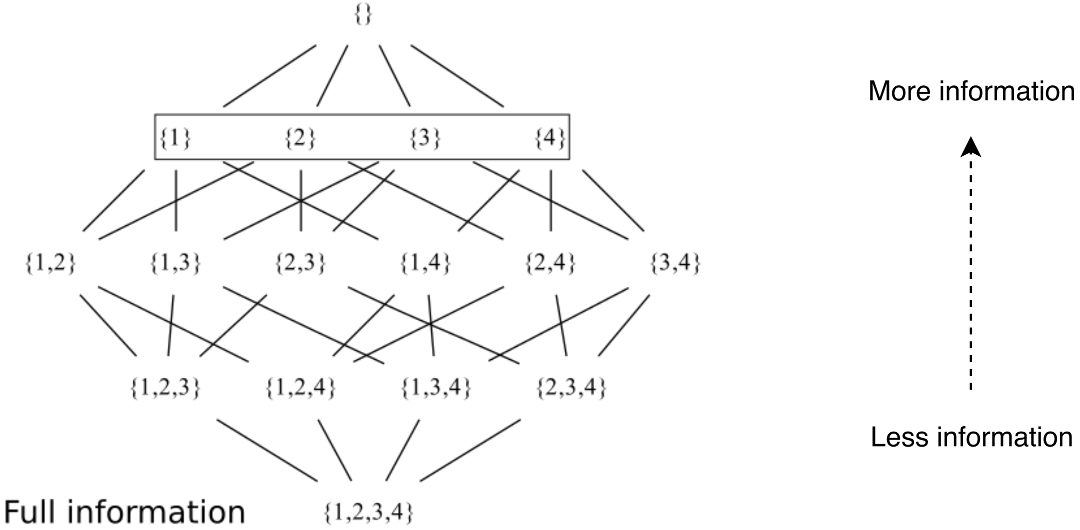

{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
{1,2,3,4}

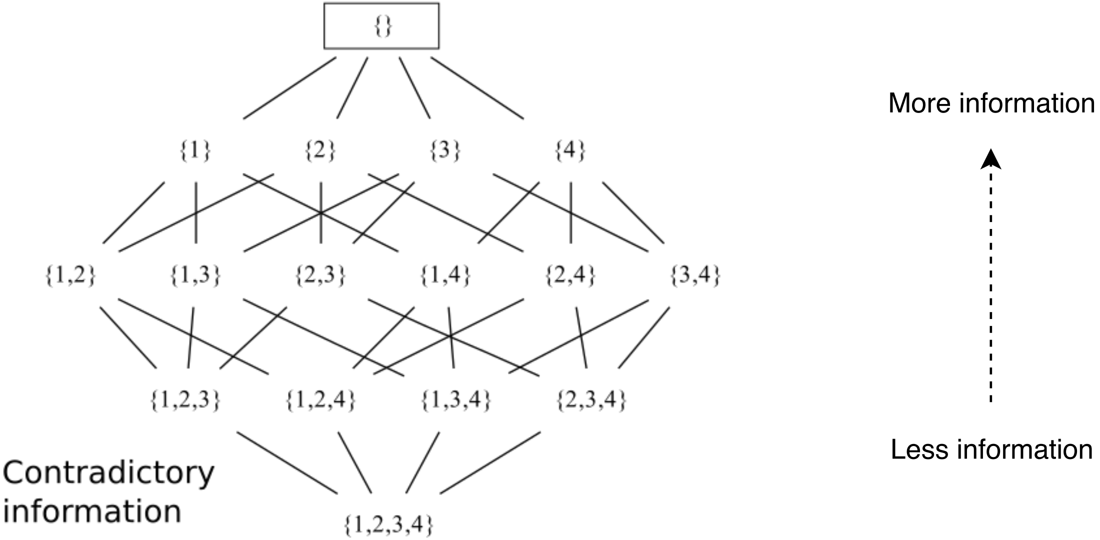

{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
{1,2,3,4}
More information
Less information

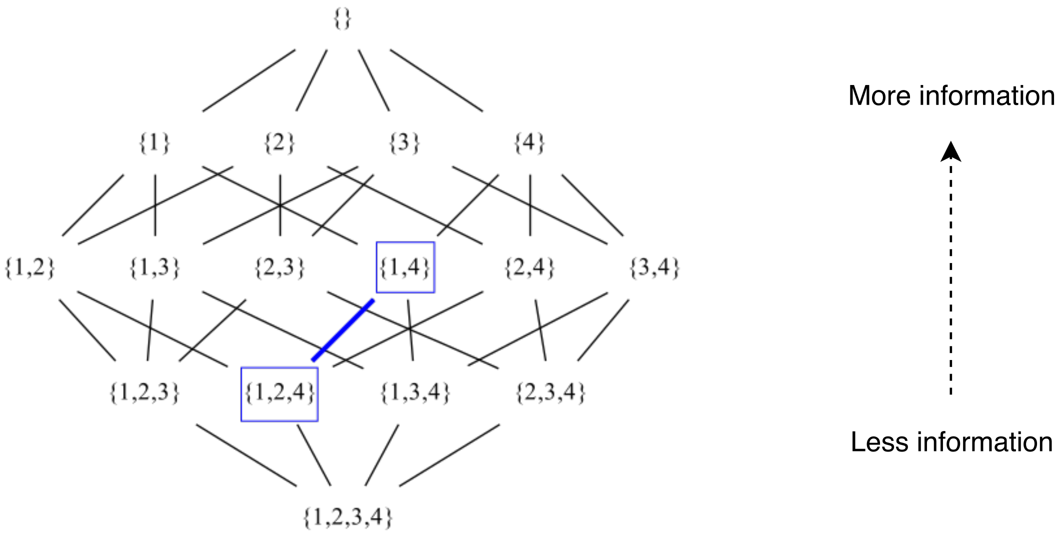


{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
No information
{1,2,3,4}
More information
Less information

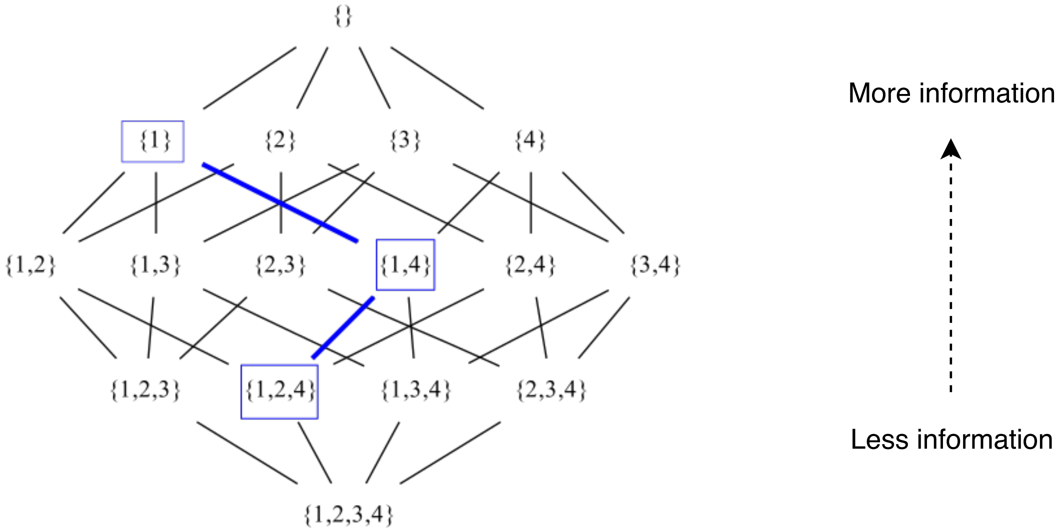

{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
{1,2,3,4}
Some information
More information
Less information


{}
{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}
Full information
More information
Less information

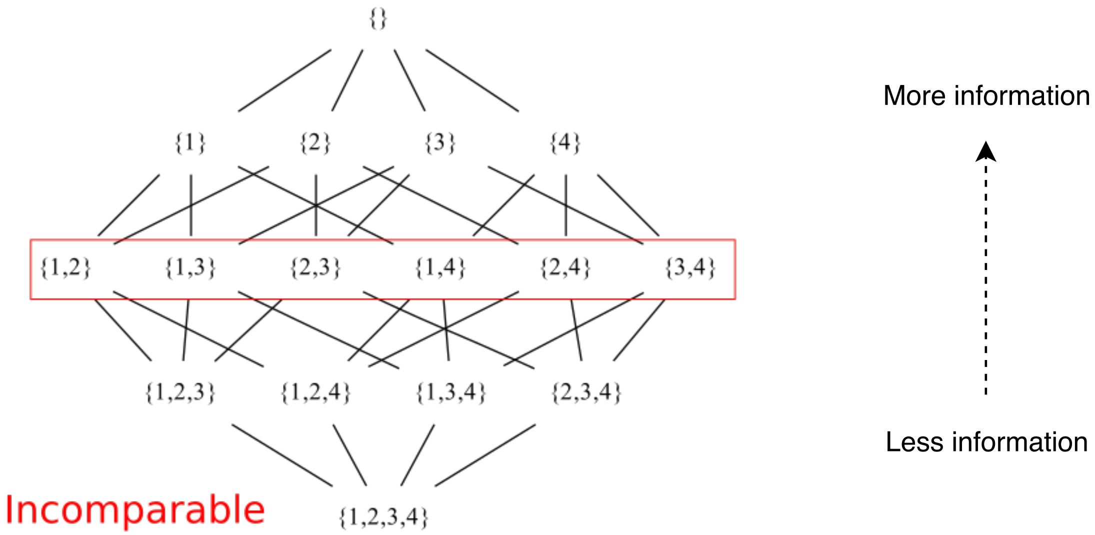

{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
{1,2,3,4}
Contradictory information
More information
Less information

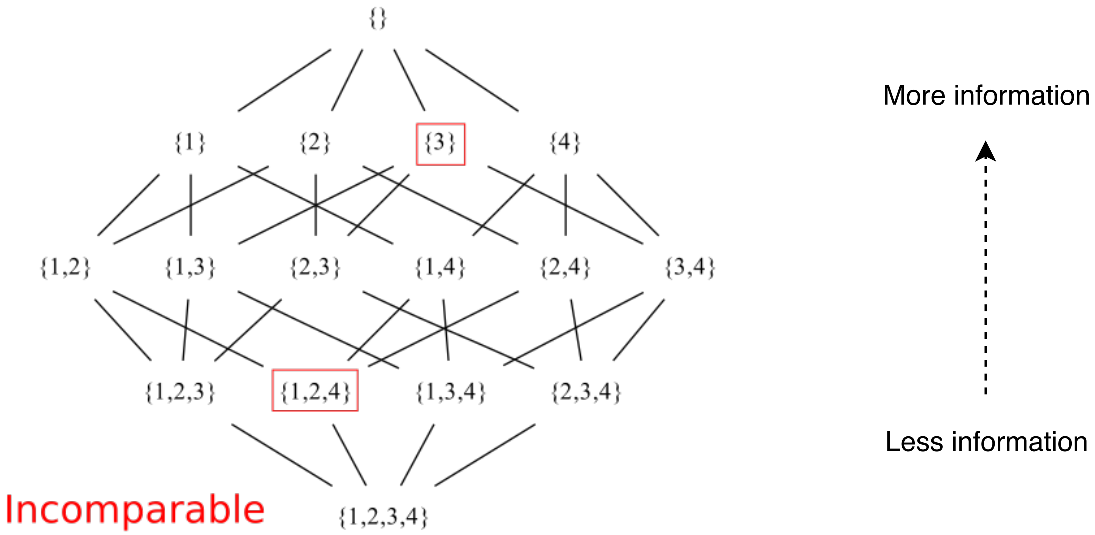

{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
{1,2,3,4}
More information
Less information

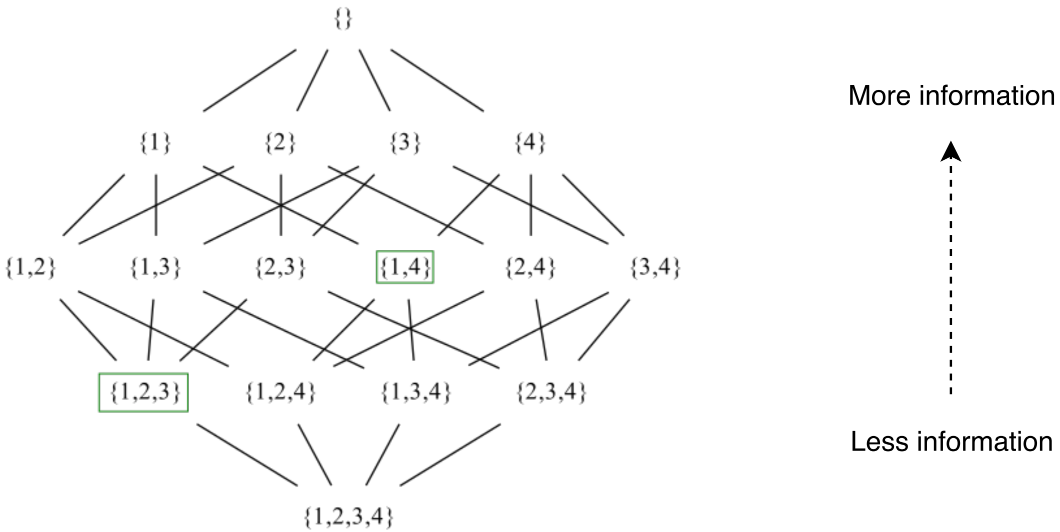
$\{1,2,4\} < \{1,4\}$


{}
{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

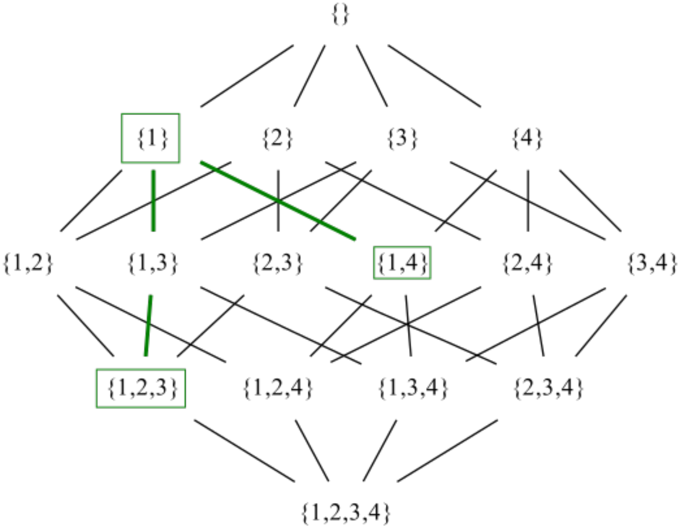
$\{1,2,4\} < \{1,4\} < \{1\}$

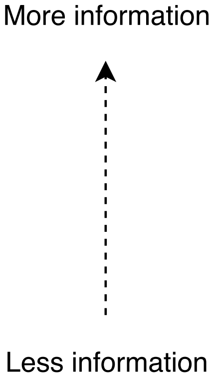

{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
{1,2,3,4}
Incomparable
More information
Less information


{}
{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}
Incomparable
More information
Less information


{}
{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

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


{}
{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

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

Bounded join semilattice

Identity:

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

Associative:

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

Commutative:

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

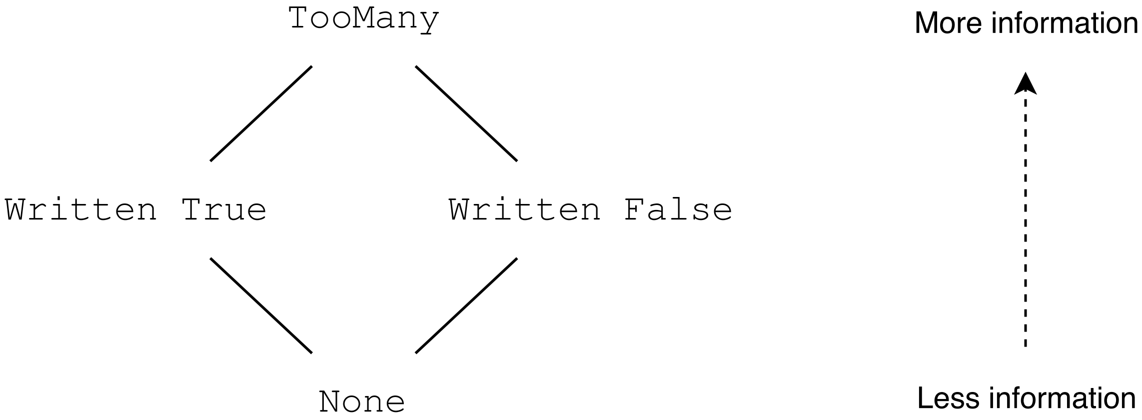
Idempotent:

$$x \vee x = x$$

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

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

instance (Eq a) => SemiLattice (WriteOnce a) where
  None      \/ b         = b
  TooMany   \/ x         = TooMany
  Written a \/ None      = Written a
  Written a \/ TooMany   = TooMany
  Written a \/ Written b = if a == b then Written a else TooMany
```

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

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

data Possibilities = Pos (Set SudokuVal)

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


TooMany
Written True
Written False
None
More information
Less information

Cells hold semilattices

Propagators always join information in

WriteOnce (aka IVar, Promise)
Sets (intersection or union)
Intervals
Search
many many more

Thanks for listening!

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