

# An Intuition for Propagators

George Wilson

CSIRO's Data61

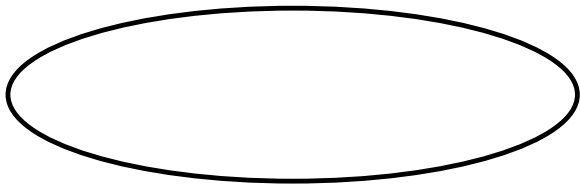
[george.wilson@data61.csiro.au](mailto:george.wilson@data61.csiro.au)

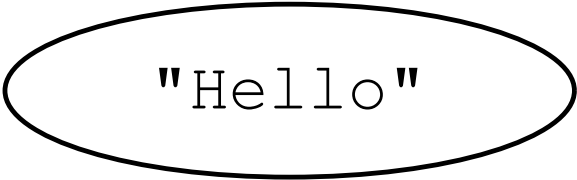
2nd September 2019



1970s, MIT

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

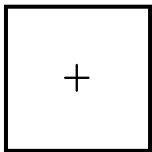


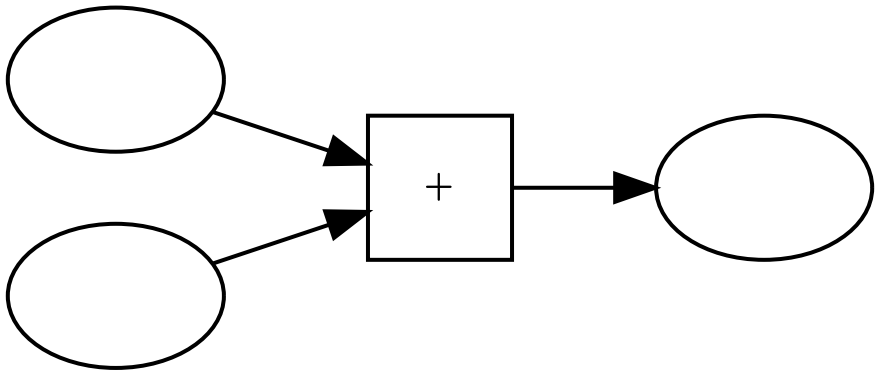


"Hello"

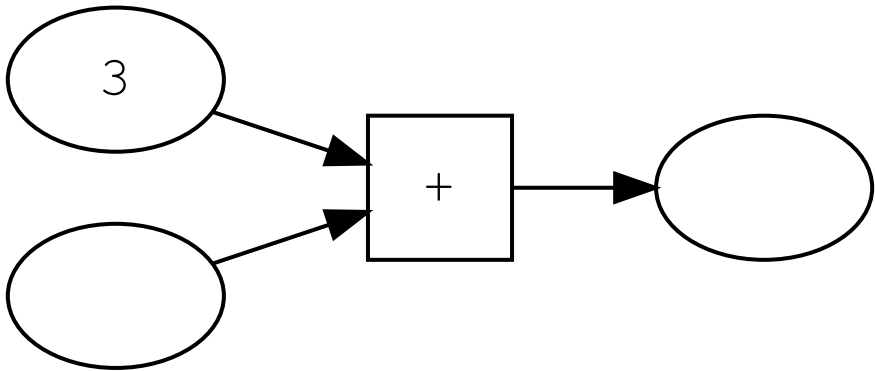


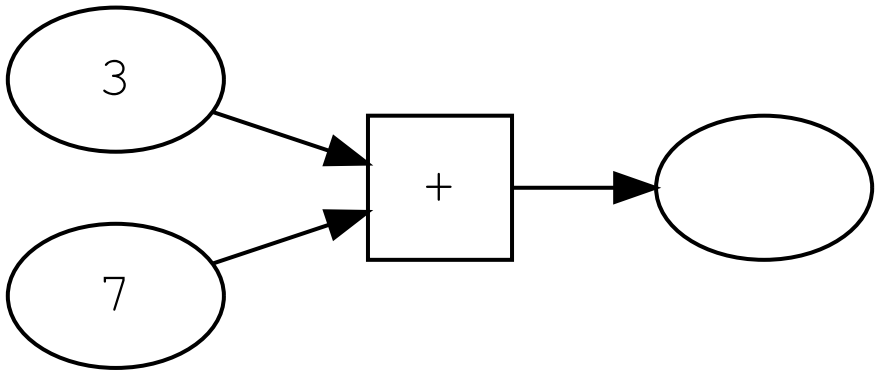
"Compose"

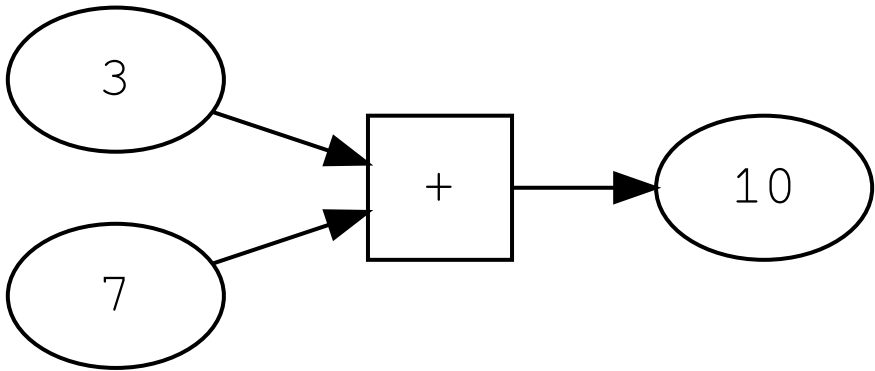


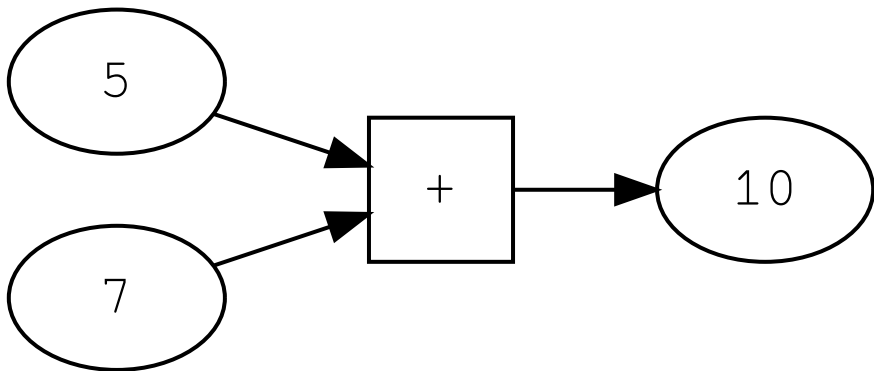


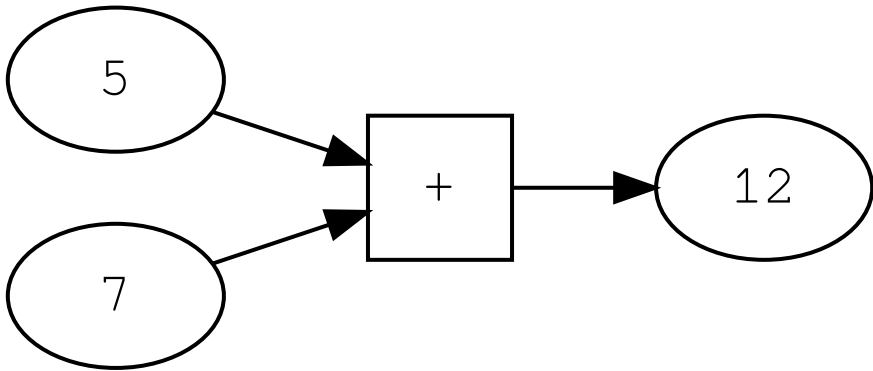












```
-- types
```

```
data Cell a
```

```
data Par a
```

```
instance Monad Par
```

```
-- types
```

```
data Cell a
```

```
data Par a
```

```
instance Monad Par
```

```
-- Creating a cell
```

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

```
-- types
```

```
data Cell a
```

```
data Par a
```

```
instance Monad Par
```

```
-- Creating a cell
```

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

```
-- Working with Cells
```

```
content  :: Cell a -> Par (Maybe a)
```

```
write    :: Cell a -> a -> Par ()
```



*-- types*

**data Cell** a

**data Par** a

**instance Monad Par**

*-- Creating a cell*

**cell** :: **Par** (**Cell** a)

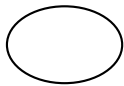
*-- Working with Cells*

**content** :: **Cell** a -> **Par** (**Maybe** a)

**write** :: **Cell** a -> a -> **Par** ()

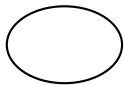
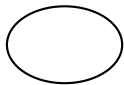
*-- Creating a propagator*

**watch** :: **Cell** a -> (a -> **Par** ()) -> **Par** ()



do

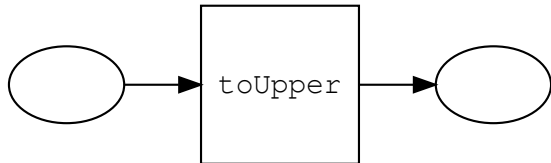
input <- cell



do

input <- cell

output <- cell



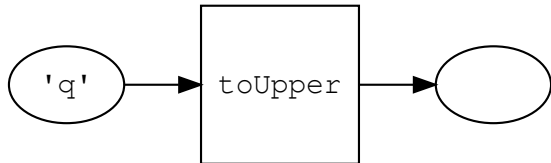
do

```
input  <- cell
```

```
output <- cell
```

```
watch input (\c ->
```

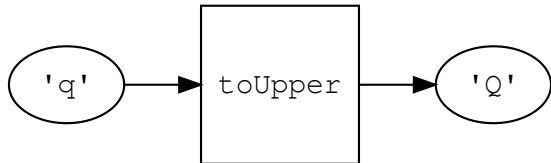
```
  write output (toUpper c))
```



do

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

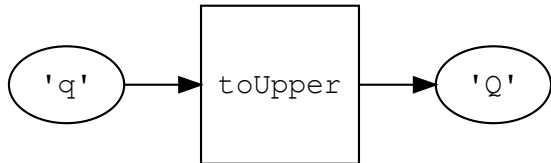
```
write input 'q'
content output  -- Just 'Q'
```



do

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

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



do

```
input  <- cell
```

```
output <- cell
```

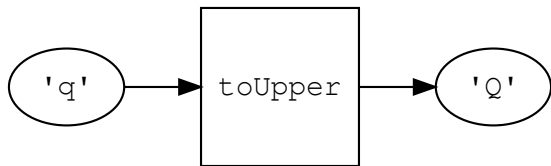
```
watch input (\c ->  
  write output (toUpper c))
```

```
write input 'q'
```

```
content output  -- Just 'Q'
```

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





do

```
input  <- cell
```

```
output <- cell
```

```
lift toUpper input output
```

```
write input 'q'
```

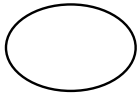
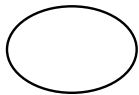
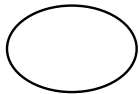
```
content output  -- Just 'Q'
```

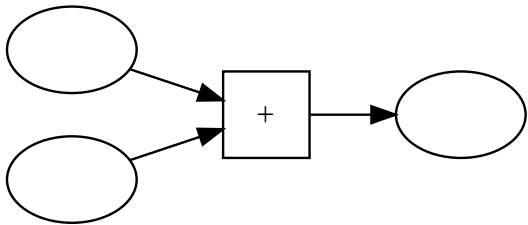
do

inL <- cell

inR <- cell

out <- cell





do

inL <- cell

inR <- cell

out <- cell

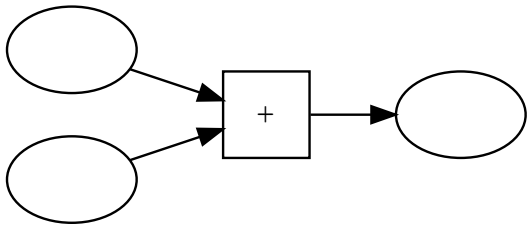
watch inL (\x -> do

maybeY <- content inR

case maybeY of

Nothing -> pure ()

Just y -> write out (x+y)



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
```

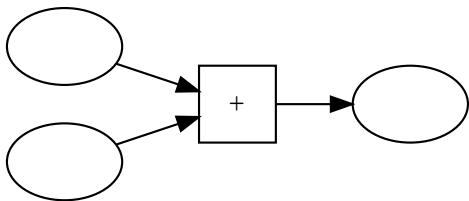
```
  maybeX <- content inL
```

```
  case maybeX of
```

```
    Nothing -> pure ()
```

```
    Just x   -> write out (x+y)
```

```
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

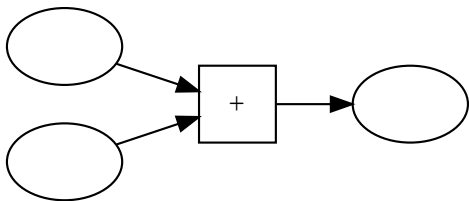
inR <- cell

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)
```

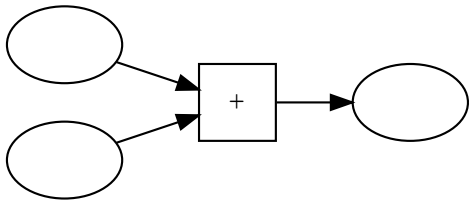
```
watch inR (\y ->  
  with inL (\x ->  
    write out (x+y)
```

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



```
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)))
```



**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

TODO: scary giant propagator network

?!

How can we fix this?

```
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
```

```
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
```

Now every network will give a **deterministic answer** in **finite time**



Mutability is **chaos**

WriteOnce is **rigid**

Accumulate information about a value

Accumulate information about a value

`merge :: a -> a -> a`

```
data WriteOnce a
```

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

```
= TooMany
```

```
  -- I know the answer exactly
```

```
| Written a
```

```
  -- I don't know anything
```

```
= None
```

```
data WriteOnce a
```

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

```
  = TooMany
```

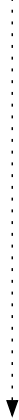
```
  -- I know the answer exactly
```

```
  | Written a
```

```
  -- I don't know anything
```

```
  = None
```

More information



Less information

```
mergeWrites :: WriteOnce a -> WriteOnce a -> WriteOnce a
mergeWrites a      None      = a
mergeWrites None   b         = b
mergeWrites TooMany b         = TooMany
mergeWrites a      TooMany    = TooMany
mergeWrites (Written a) (Written b) =
    if a == b then Written a else TooMany
```

```
mergeWrites :: WriteOnce a -> WriteOnce a -> WriteOnce a
```

```
mergeWrites a      None      = a
```

```
mergeWrites None    b        = b
```

```
mergeWrites TooMany b        = TooMany
```

```
mergeWrites a      TooMany    = TooMany
```

```
mergeWrites (Written a) (Written b) =
```

```
    if a == b then Written a else TooMany
```

```
mergeWrites :: WriteOnce a -> WriteOnce a -> WriteOnce a
mergeWrites a      None      = a
mergeWrites None    b        = b
mergeWrites TooMany b        = TooMany
mergeWrites a      TooMany    = TooMany
mergeWrites (Written a) (Written b) =
    if a == b then Written a else TooMany
```



```
mergeWrites :: WriteOnce a -> WriteOnce a -> WriteOnce a
mergeWrites a      None      = a
mergeWrites None   b         = b
mergeWrites TooMany b         = TooMany
mergeWrites a      TooMany    = TooMany
mergeWrites (Written a) (Written b) =
    if a == b then Written a else TooMany
```

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



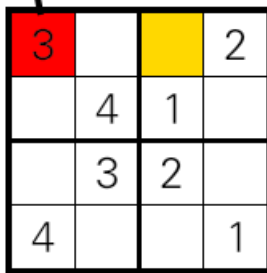
$\{1,3,4\}$

3			2
	4	1	
	3	2	
4			1

3			2
	4	1	
	3	2	
4			1

$\{2,3,4\}$

$\{1,2,4\}$



3		2	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

The accumulation must:

- tolerate **reordering** of information
- tolerate **grouping** of information
- ignore **redundancy** of information

# Join semilattice

Commutative:

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

Associative:

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

Idempotent:

$$x \vee x = x$$



```
class SemiLattice a where
```

```
  (\\)      :: a -> a -> a
```

```
  bottom :: a
```

```
instance (Eq a) => SemiLattice (WriteOnce a) where
```

```
  None      \\ b      = b
```

```
  TooMany   \\ x      = TooMany
```

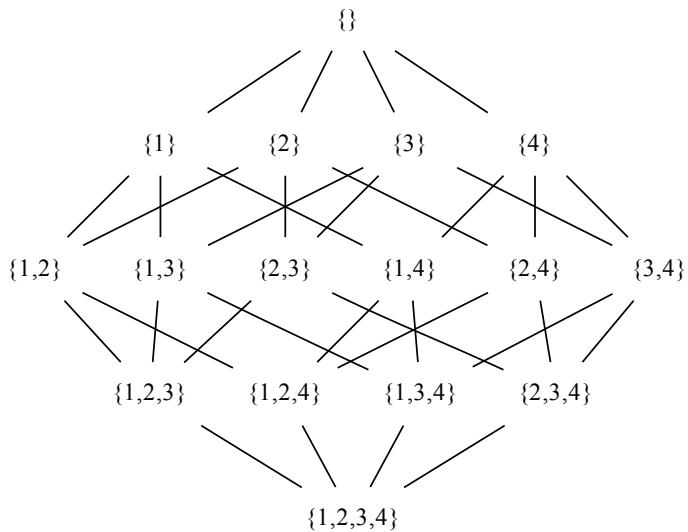
```
  Written a \\ None    = Written a
```

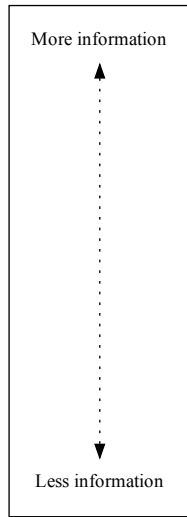
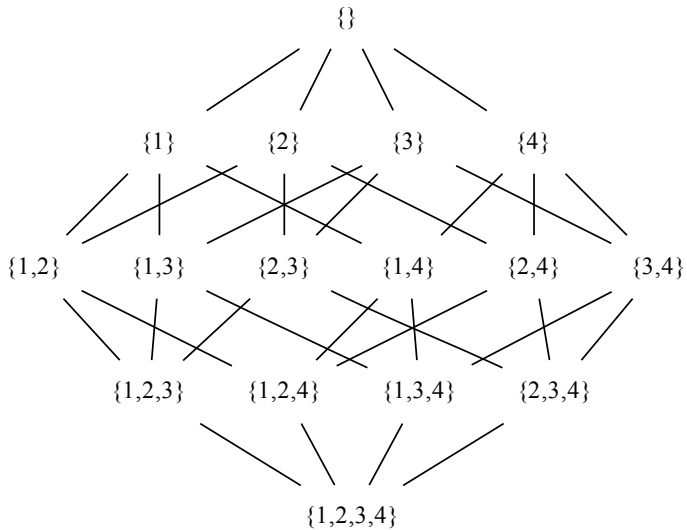
```
  Written a \\ Written b = if a == b then Written a else TooMany
```

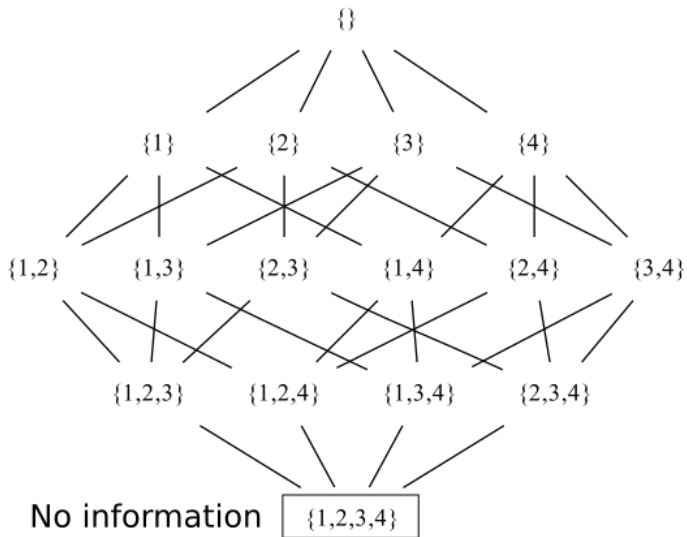
```
  Written a \\ TooMany  = TooMany
```

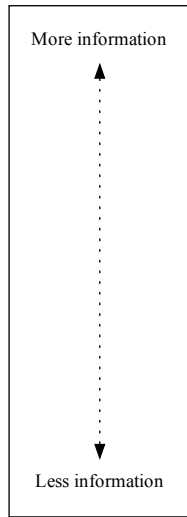
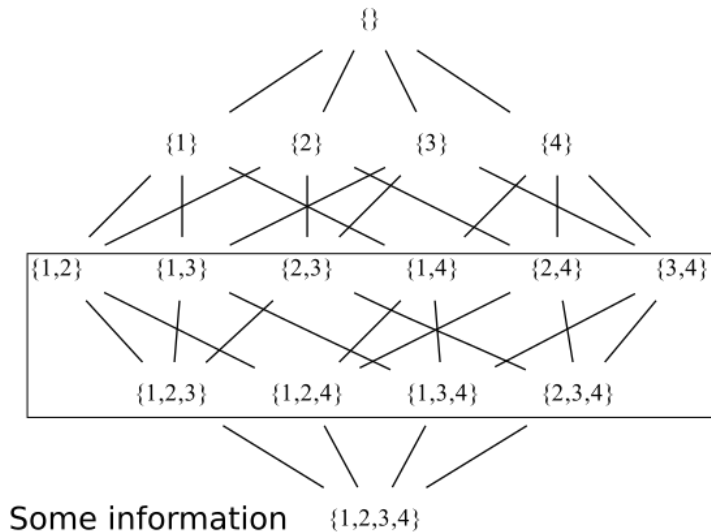
```
instance (Ord a) => Semilattice (Set a) where
  p \/\ q = Set.intersection p q
  bottom = Set.empty
```

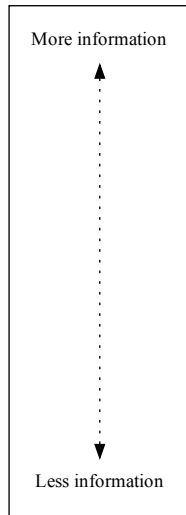
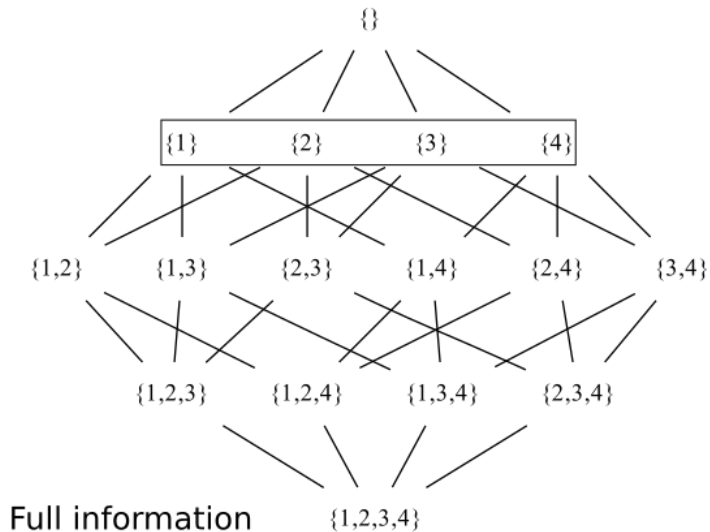
Semilattices are a type of partial order



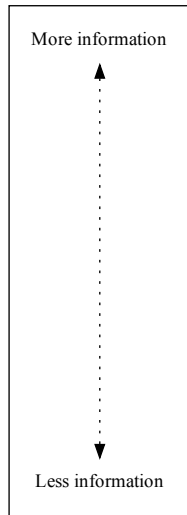
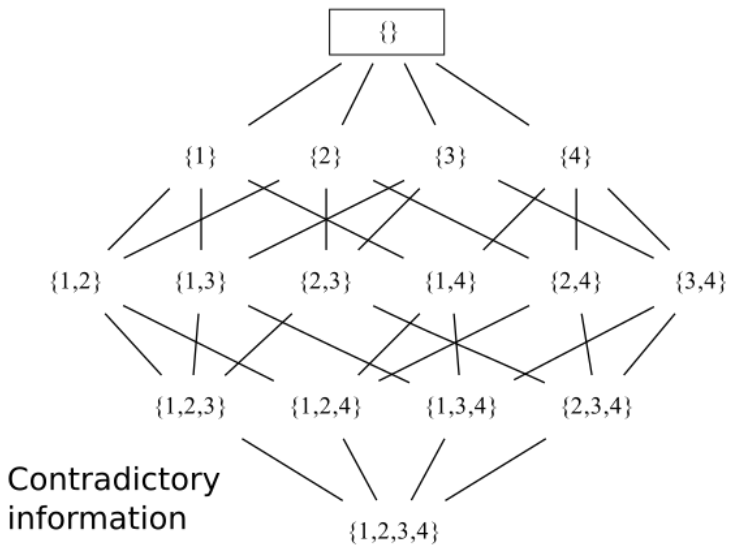


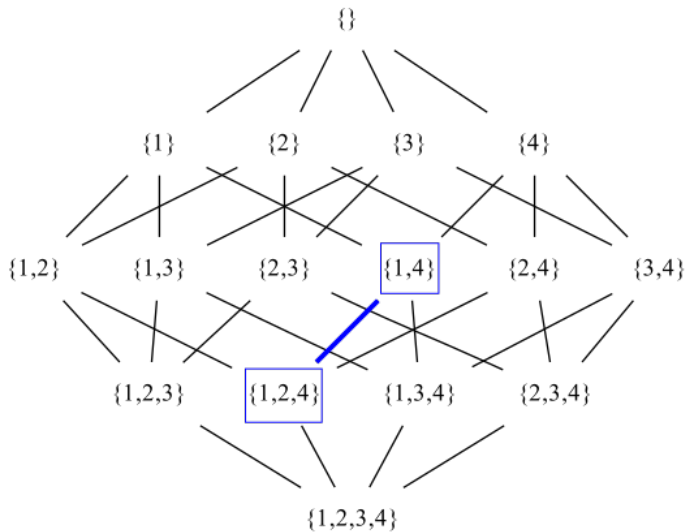










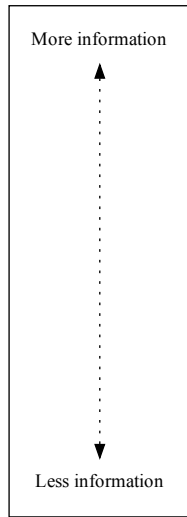
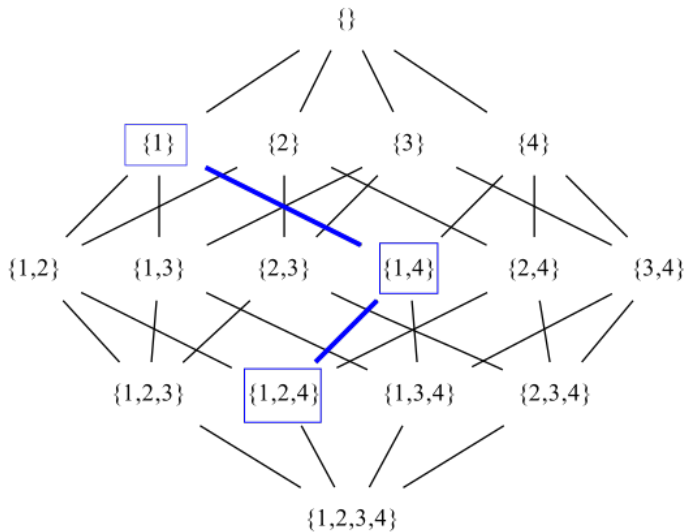


More information

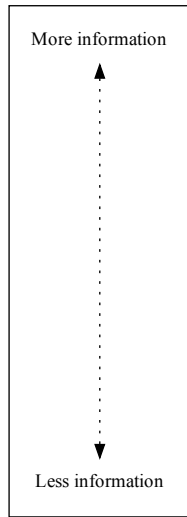
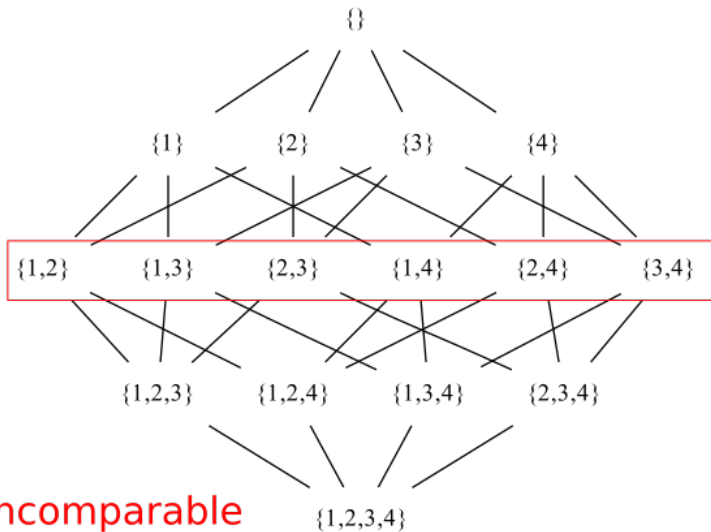


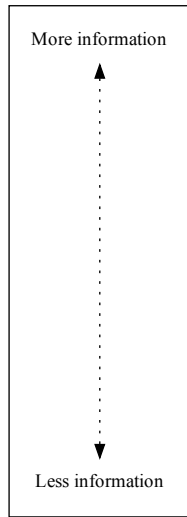
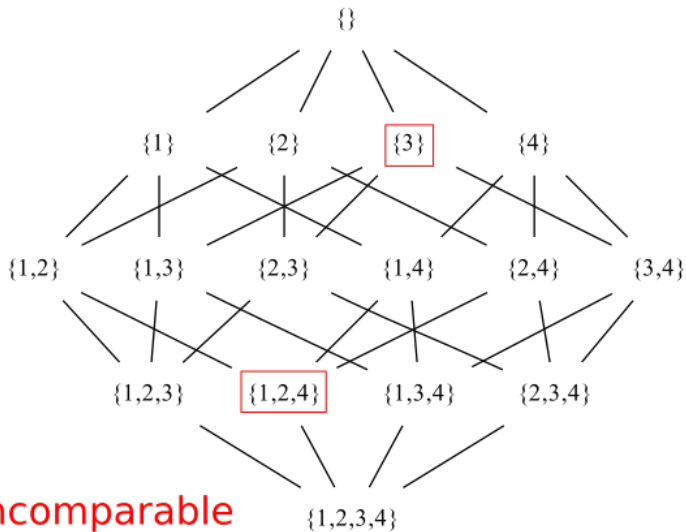
Less information

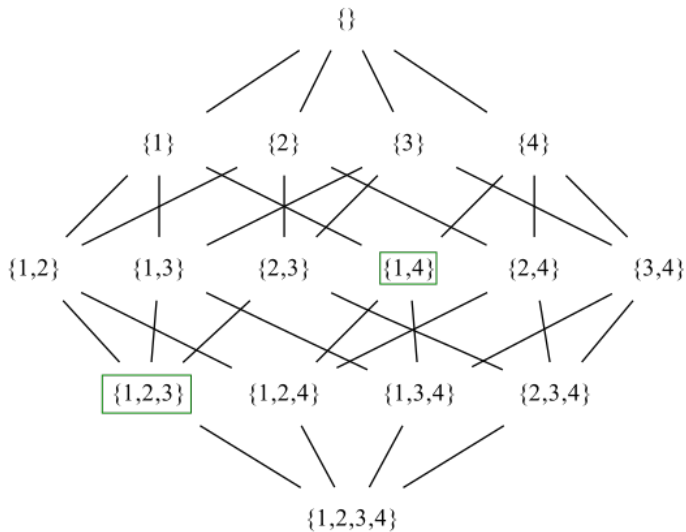
$$\{1,2,4\} < \{1,4\}$$



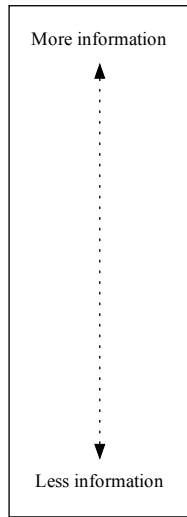
$$\{1, 2, 4\} < \{1, 4\} < \{1\}$$

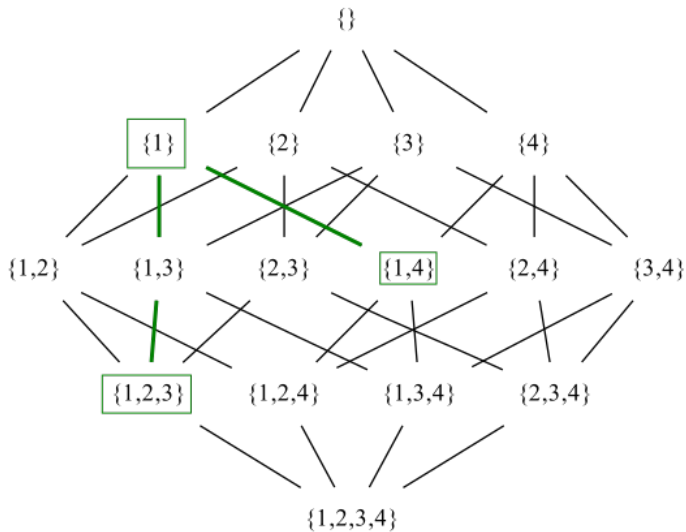




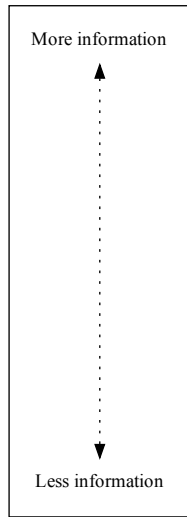


$$\{1,2,3\} \vee \{1,4\}$$





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



# Monotonicity

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



Thanks for listening!