An Intuition for Propagators

George Wilson

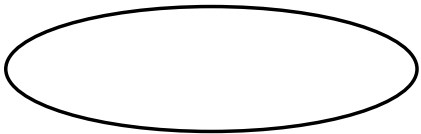
CSIRO's Data61

george.wilson@data61.csiro.au

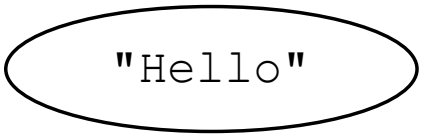
2nd September 2019

1970s, MIT

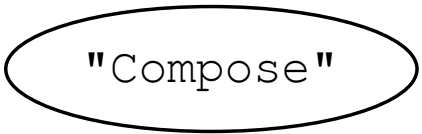
a model of computation for **highly parallel** machines



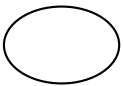
```
do
  c <- cell
```

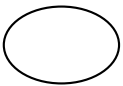



```
do
  c <- cell
  write c "Hello"
```

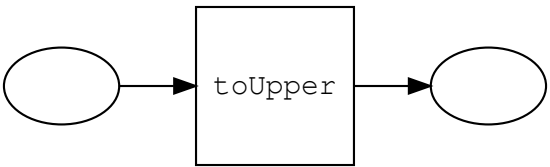



```
do
  c <- cell
  write c "Hello"
  write c "Compose"
```

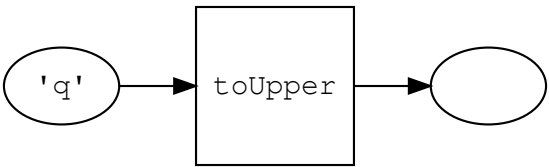




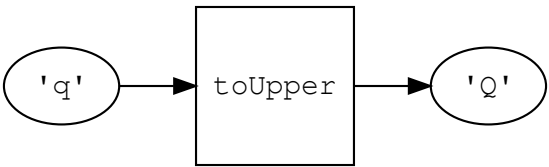
```
do
  input  <- cell
  output <- cell
```



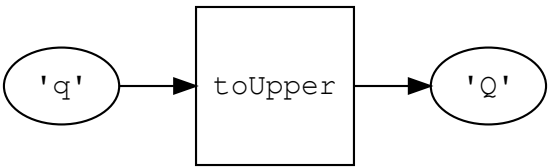

```
do
  input  <- cell
  output <- cell
  lift toUpper input output
```




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

  -- run the network
  write input 'q'
```

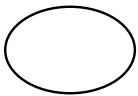


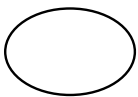

```
do
  input  <- cell
  output <- cell
  lift toUpper input output

  -- run the network
  write input 'q'
```

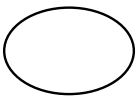


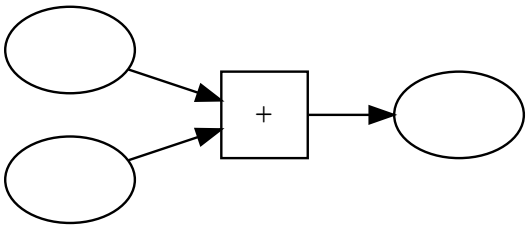

```
do
  input  <- cell
  output <- cell
  lift toUpper input output

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

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

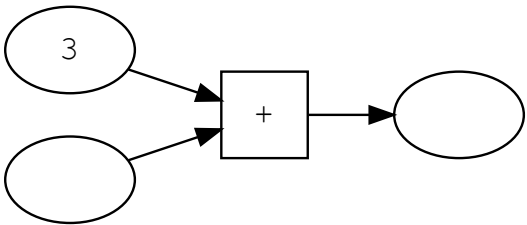




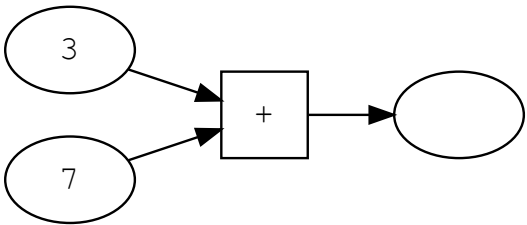




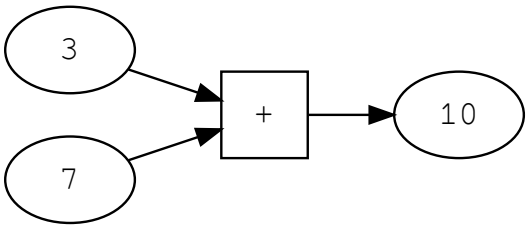

```
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  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
```




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

  adder inL inR out

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

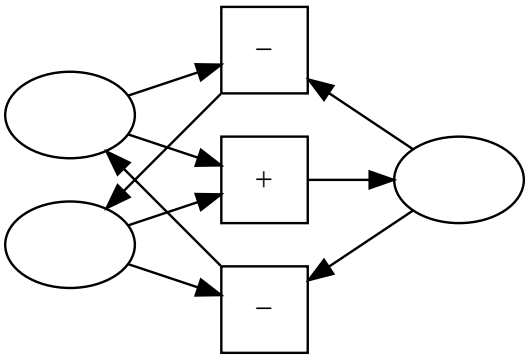



```
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  adder inL inR out

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

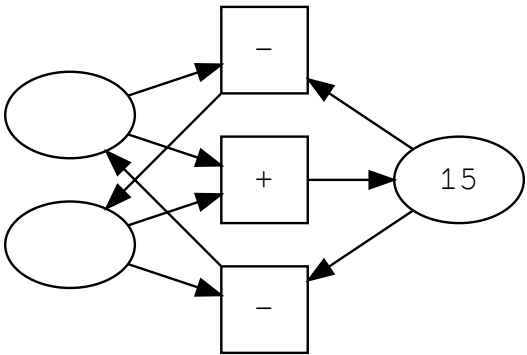
$$z = x + y$$

$$z \leftarrow x + y$$

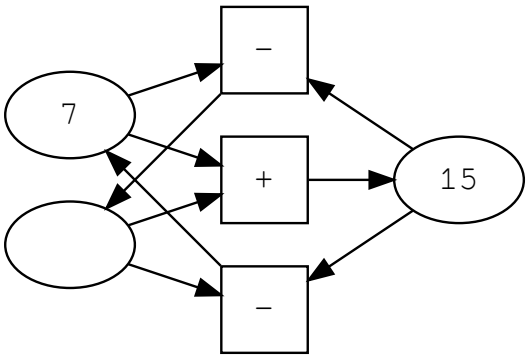
$$
\begin{aligned}
z &\leftarrow x + y \\
x &\leftarrow z - y \\
y &\leftarrow z - x
\end{aligned}
$$



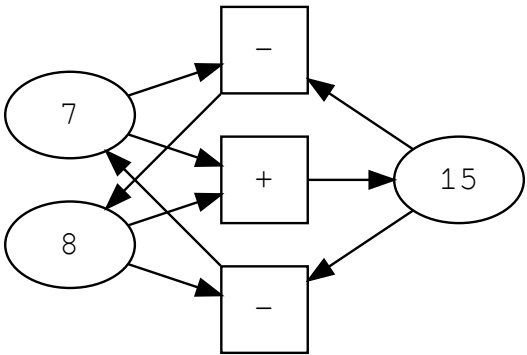

```
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  adder inL inR out

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




```
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
```



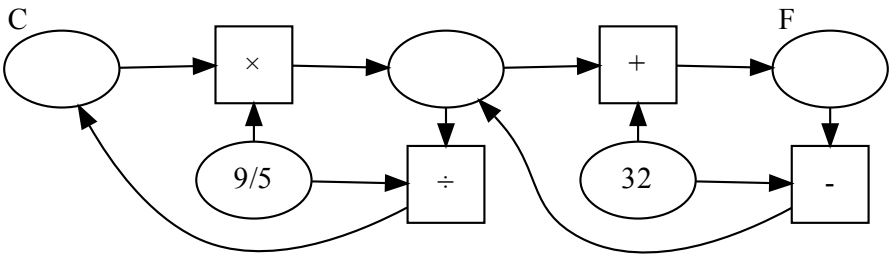

```
do
  inL  <- cell
  inR  <- cell
  out  <- cell

  adder inL inR out

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




```
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
```

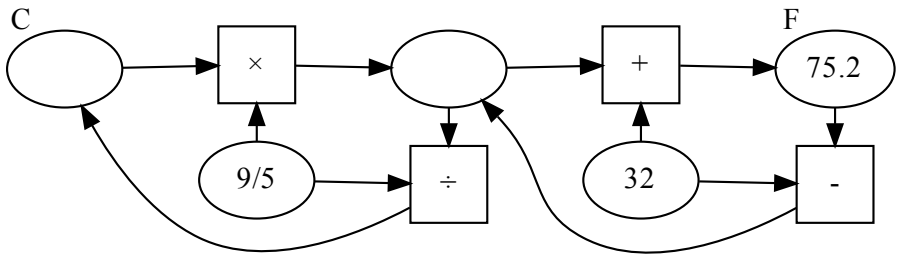
$$^\circ F = {}^\circ C \times \frac{9}{5} + 32$$

$$^\circ C = (^\circ F - 32) \div \frac{9}{5}$$



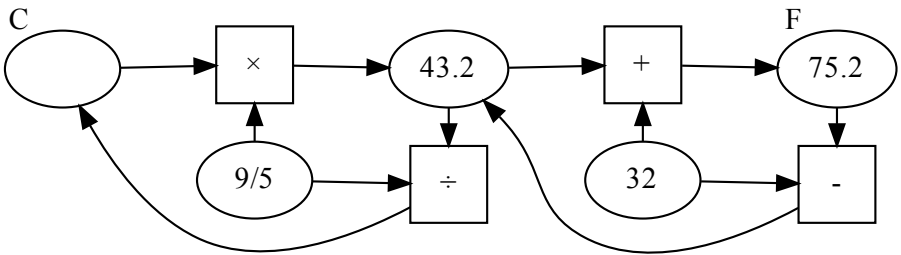
$$^\circ F = {}^\circ C \times \frac{9}{5} + 32$$

$$^\circ C = (^\circ F - 32) \div \frac{9}{5}$$



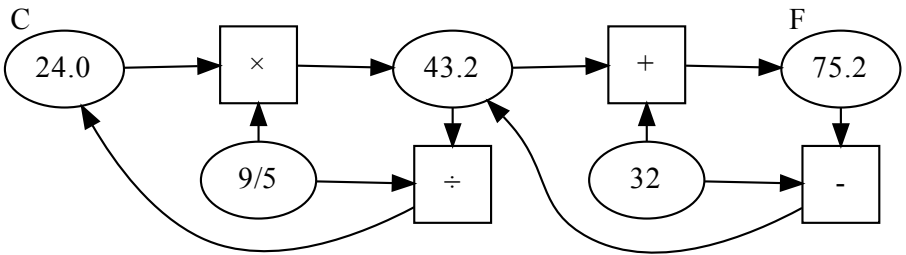
$$^\circ F = {}^\circ C \times \tfrac{9}{5} + 32$$

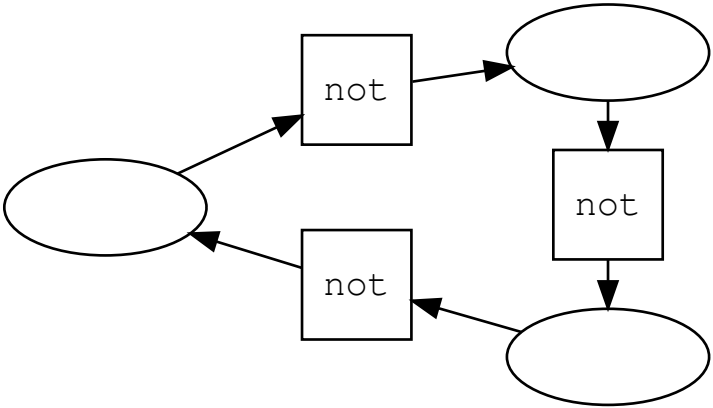
$$^\circ C = (^\circ F - 32) \div \tfrac{9}{5}$$

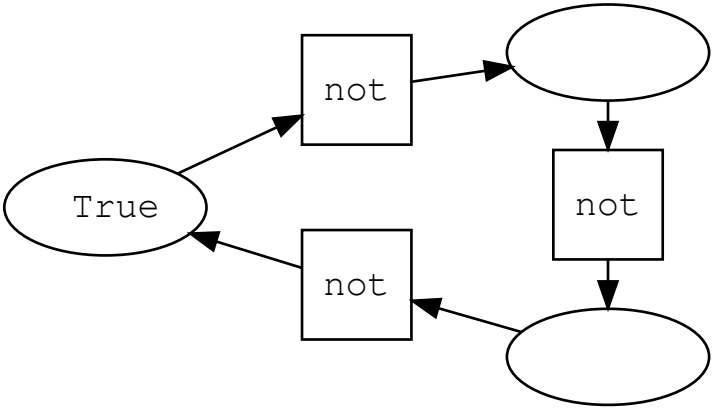


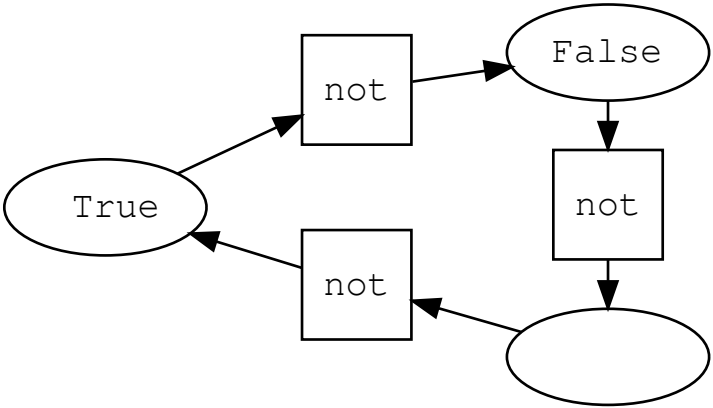
$$^\circ F = {}^\circ C \times \frac{9}{5} + 32$$

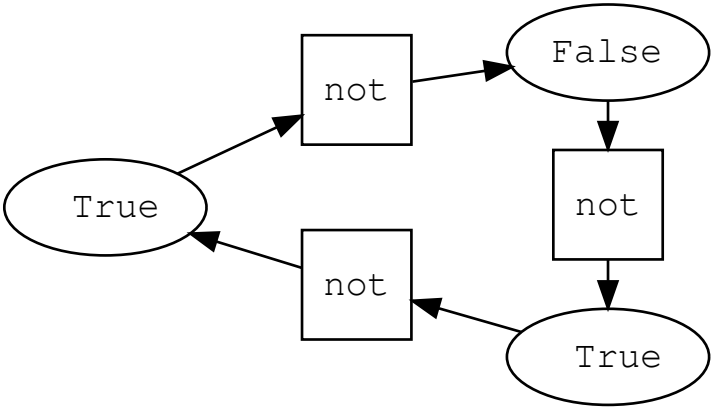
$$^\circ C = (^\circ F - 32) \div \frac{9}{5}$$

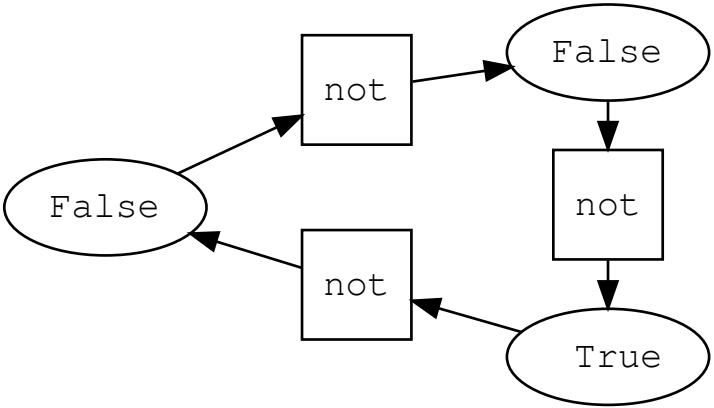


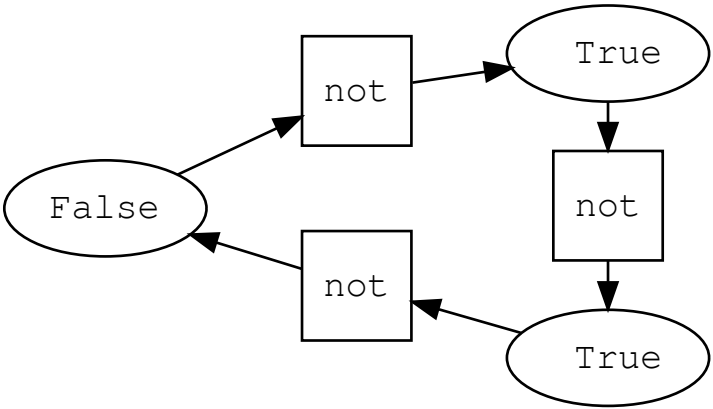

not
not
not

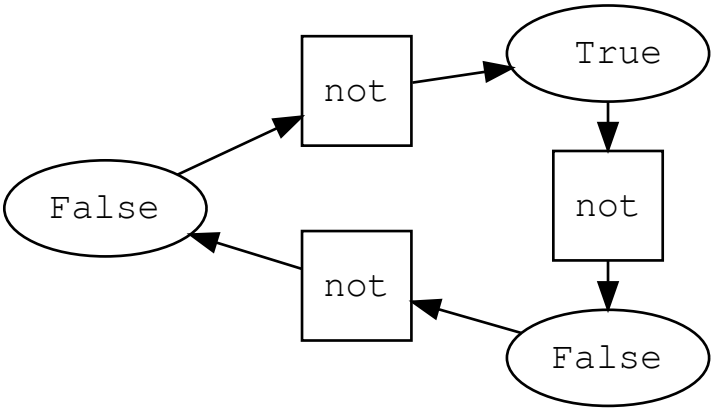

not
not
True
not

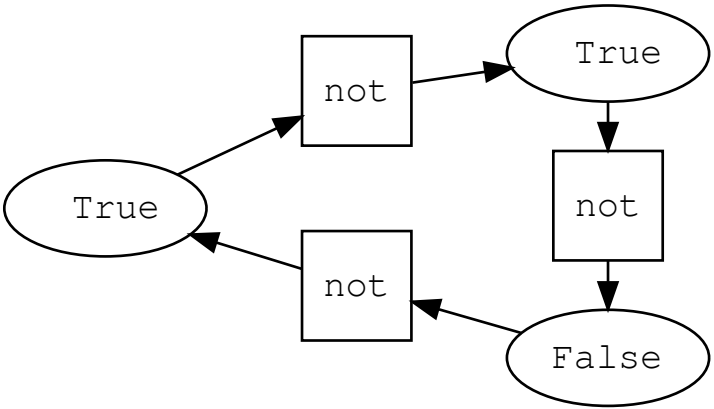

not
False
True
not
not

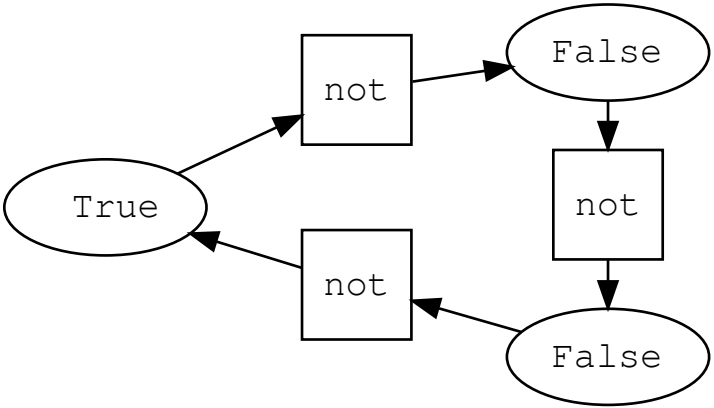

not
False
True
not
not
True


not
False
False
not
not
True


not
True
False
not
not
True


not
True
False
not
not
False


True
not
True
not
not
False

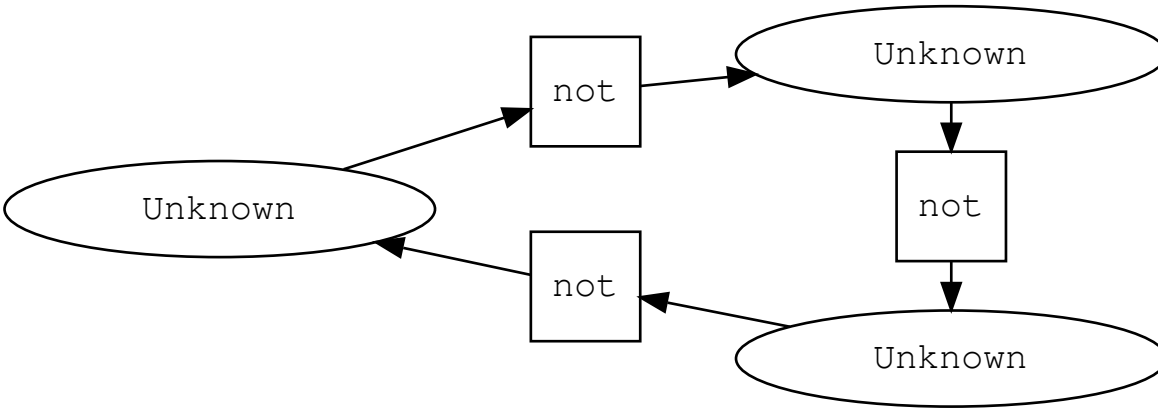

not
False
True
not
not
False

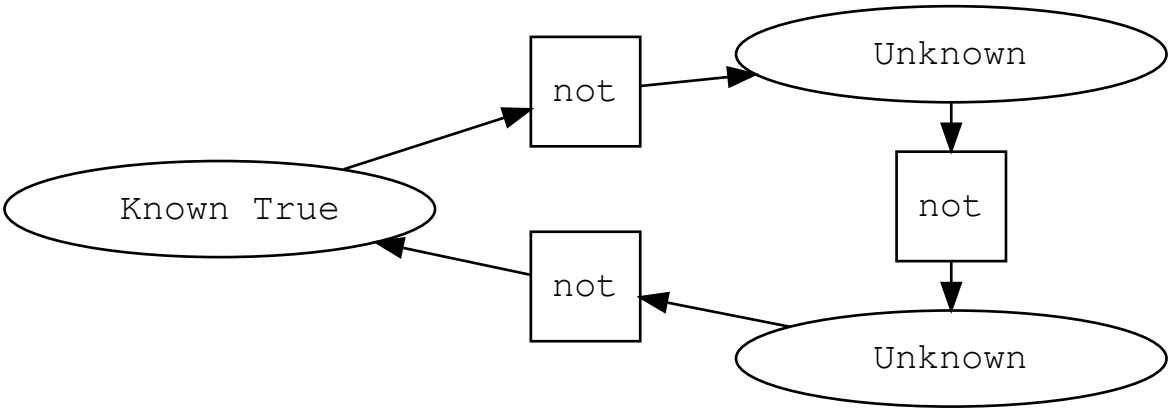
?!

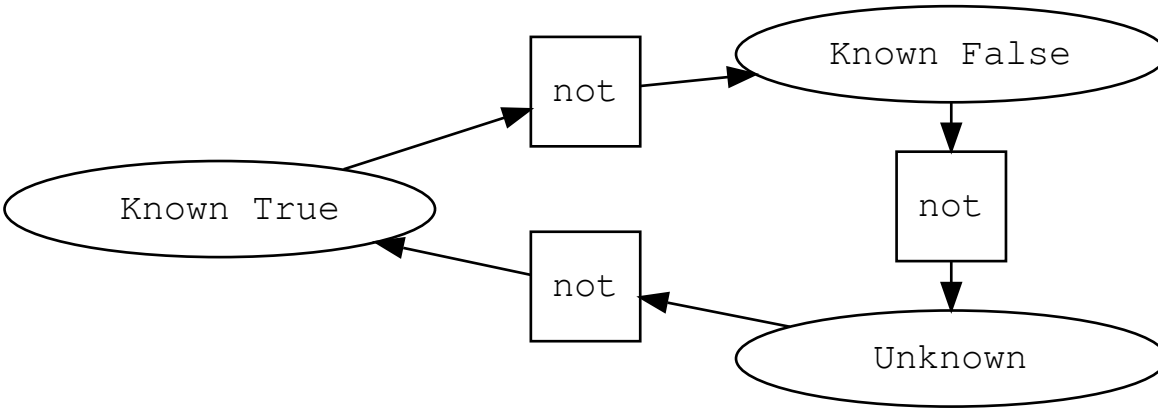
How can we fix this?

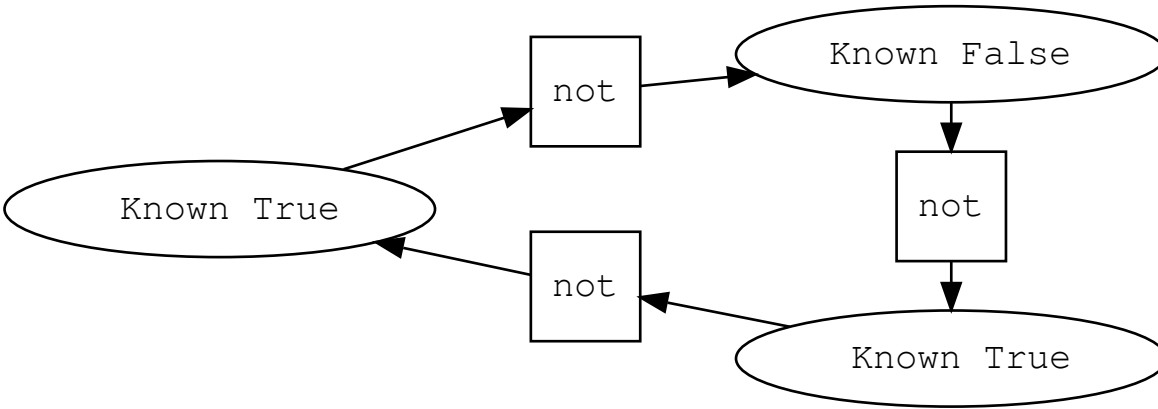
```haskell
data Perhaps a
  = Unknown
  | Known a
  | Contradiction
```

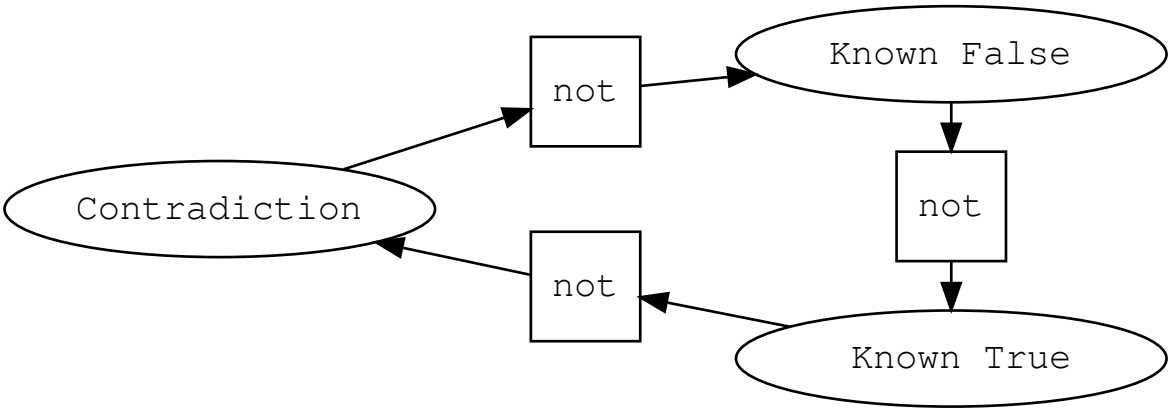
```haskell
data Perhaps a
  = Unknown
  | Known a
  | Contradiction


tryWrite :: (Eq a) => a -> Perhaps a -> Perhaps a
tryWrite a p = case p of
  Unknown -> Known a
  Known b -> if a == b then Known b else Contradiction
  Contradiction -> Contradiction
```

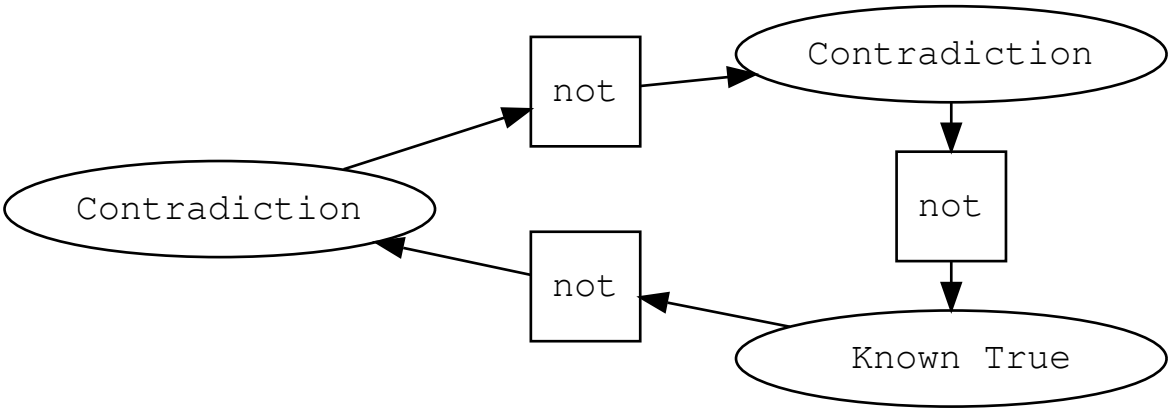

not
Unknown
not
Unknown
not
Unknown

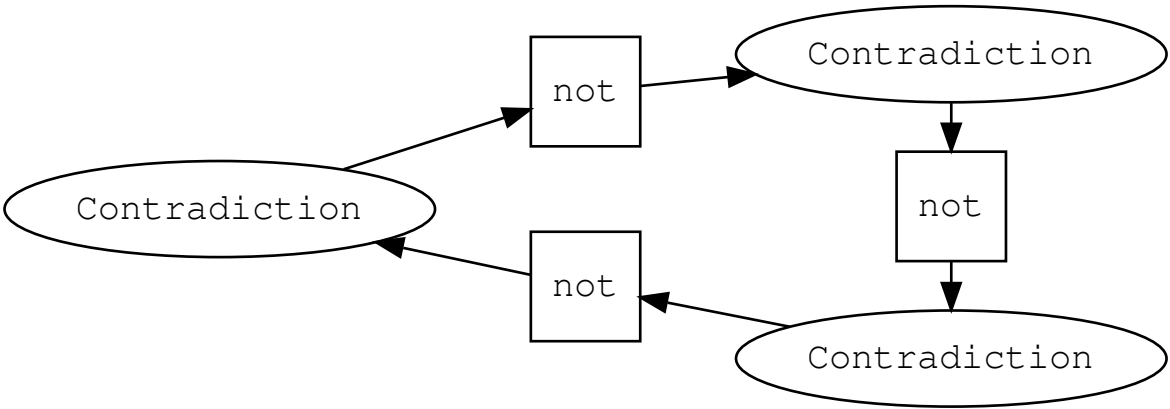

Known True
not
Unknown
not
Unknown
not


Known False
not
not
Known True
not
Unknown

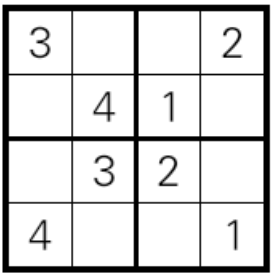

not
Known False
Known True
not
not
Known True

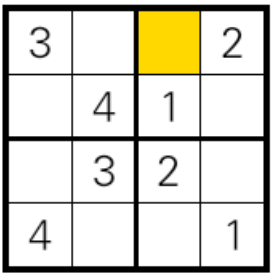

not
Known False
Contradiction
not
not
Known True

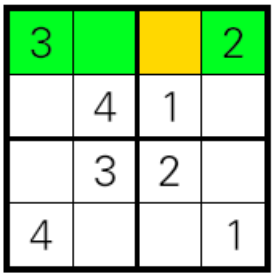

not
Contradiction
Contradiction
not
not
Known True

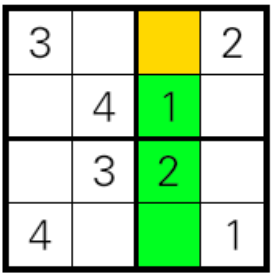

Contradiction
not
Contradiction
not
Contradiction
not

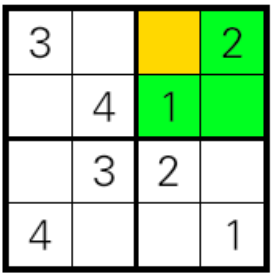
Accumulate information about a value

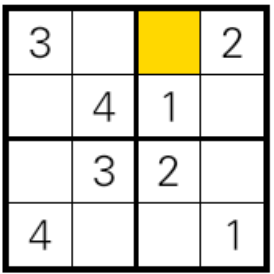

3 2
4 1
3 2
4 1

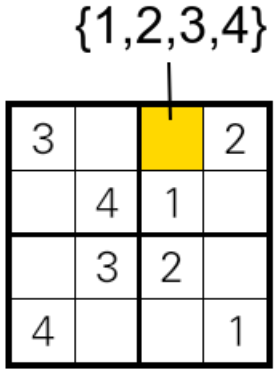

3
2
4
1
3
2
4
1

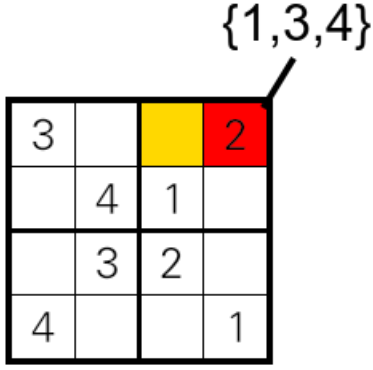

3
2
4
1
3
2
4
1

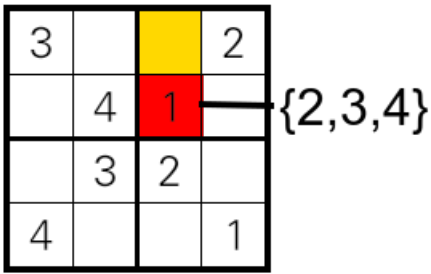

3
2
4
1
3
2
4
1

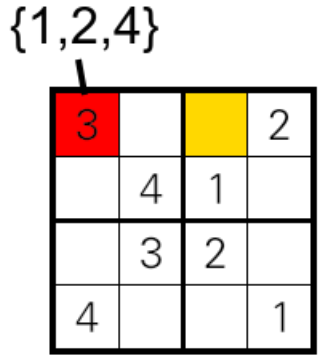

3
2
4
1
3
2
4
1


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

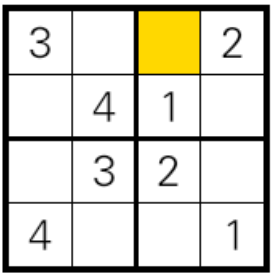

3
2
4
1
{2,3,4}
3
2
4
1

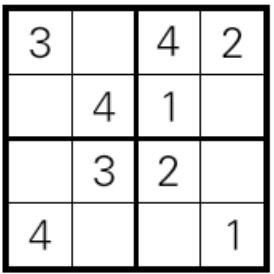

{1,2,4}
3
2
4
1
3
2
4
1

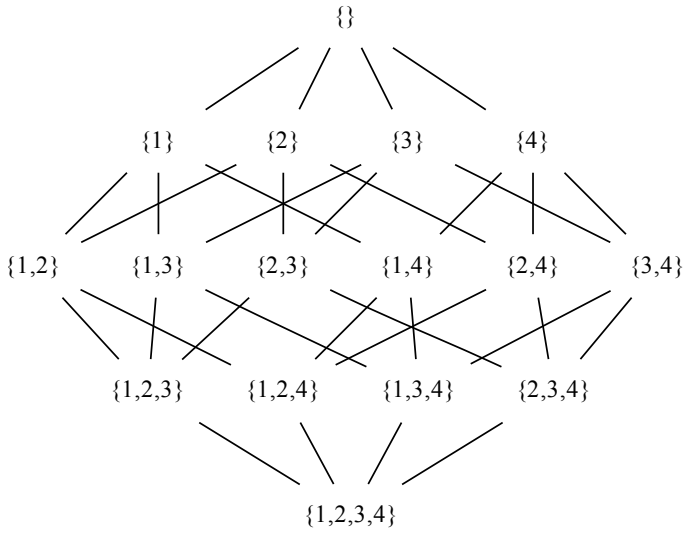
$$\{2,3,4\} \cap \{1,3,4\} \cap \{1,2,4\} \cap \{1,2,3,4\}$$

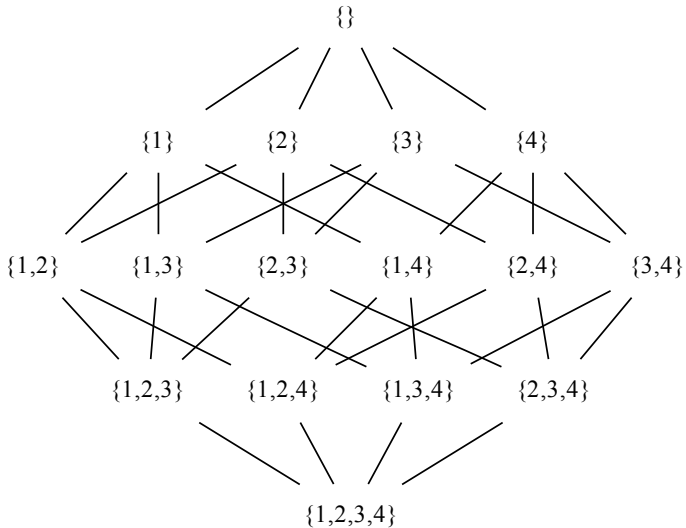
3			2
	4	1	
	3	2	
4			1

{4}



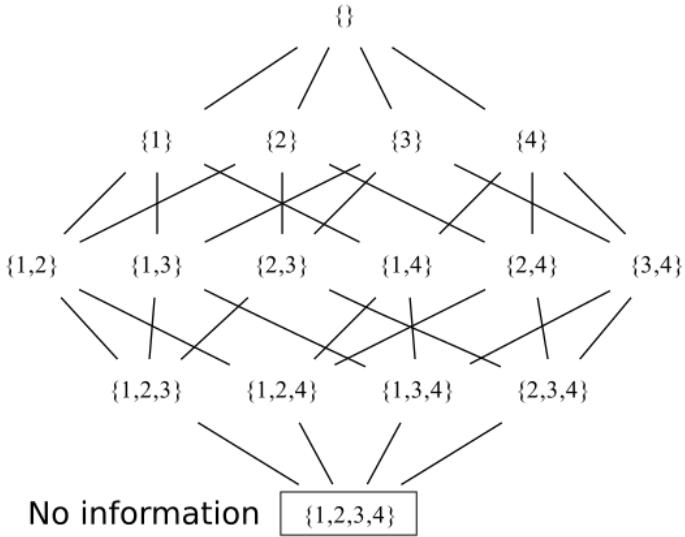

3 4 2
4 1
3 2
4 1


{}
{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}



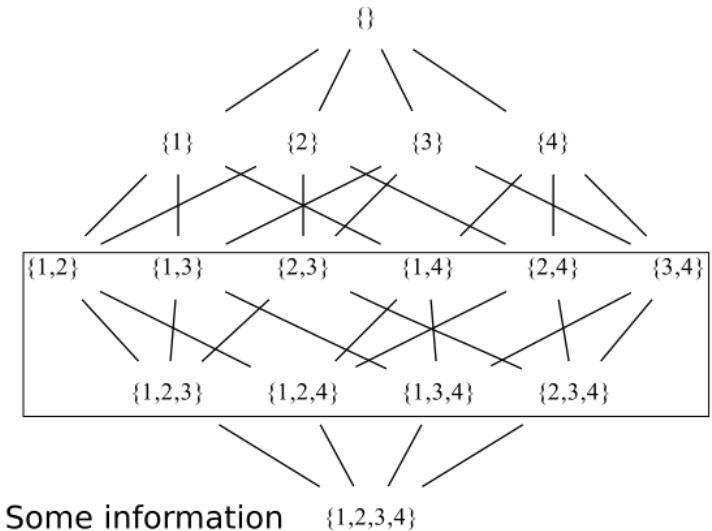
More information

Less information



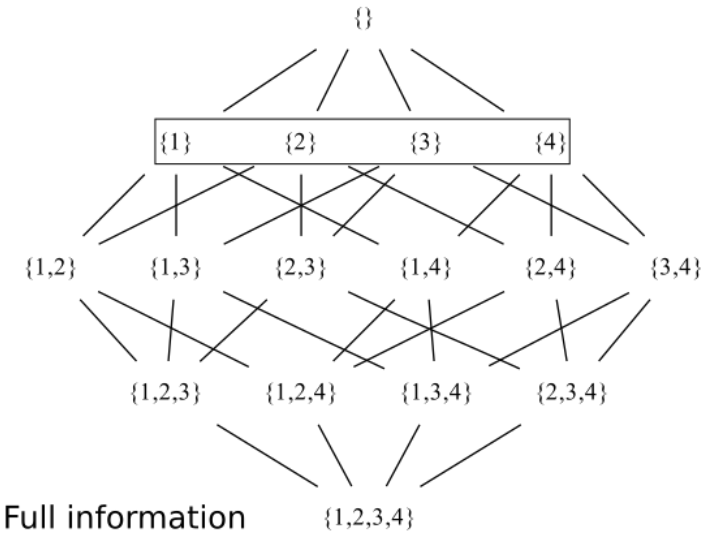
More information

Less information



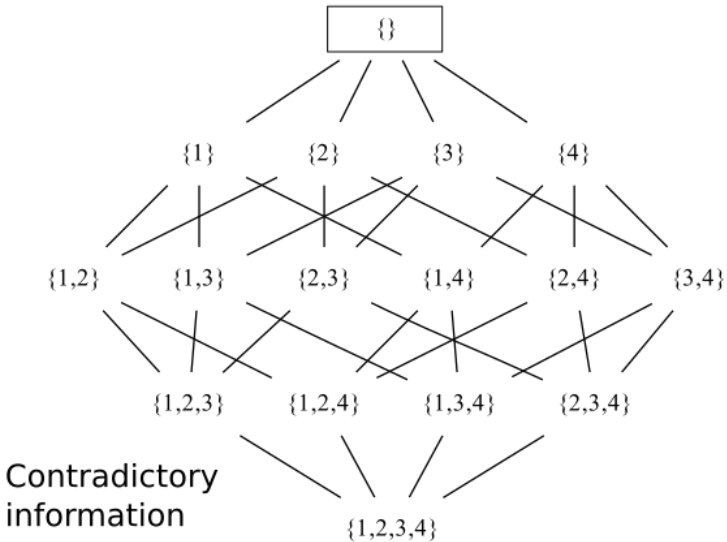
More information

Less information



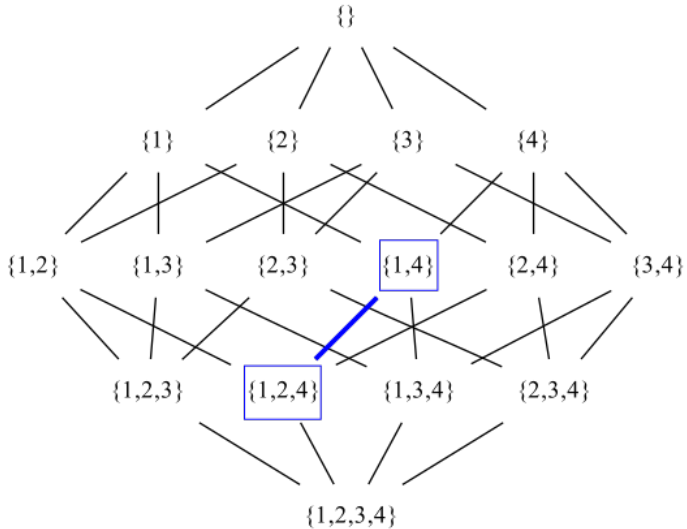
More information

Less information



More information

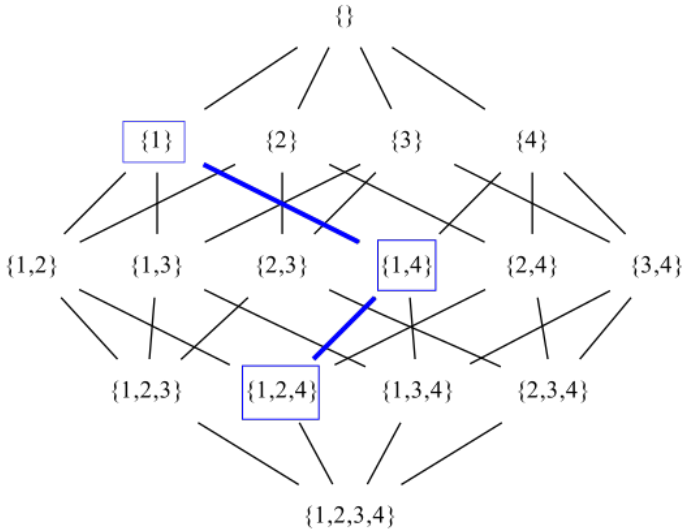
Less information



More information

Less information

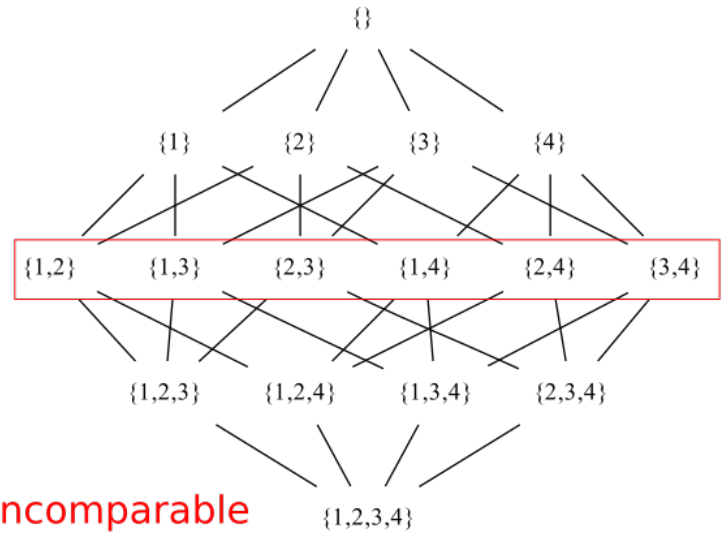
$\{1,2,4\} < \{1,4\}$



More information

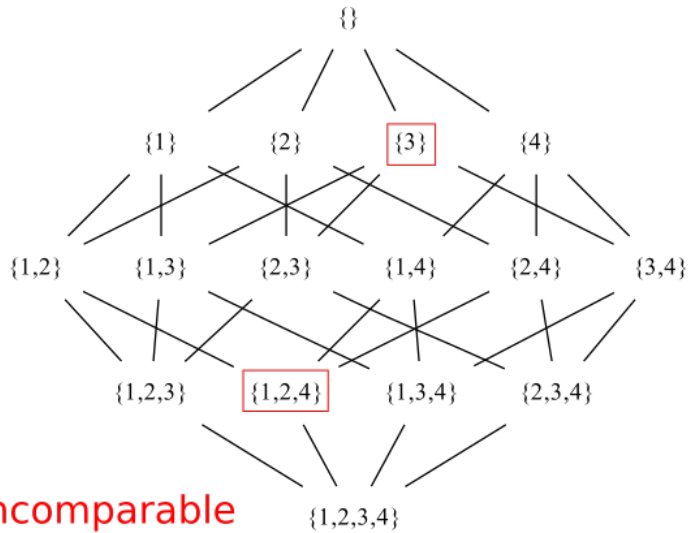
Less information

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



More information

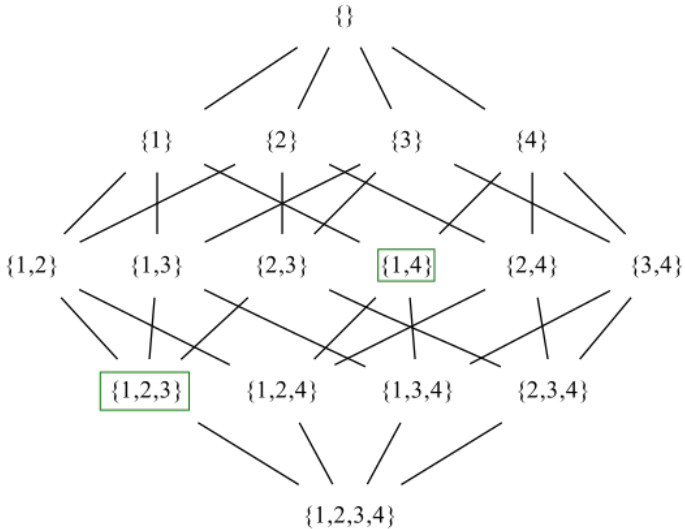
Less information



Incomparable

More information

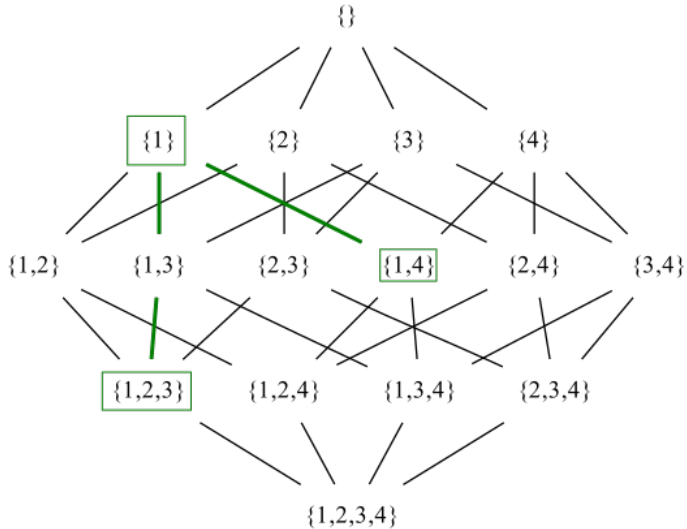
Less information



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)
```

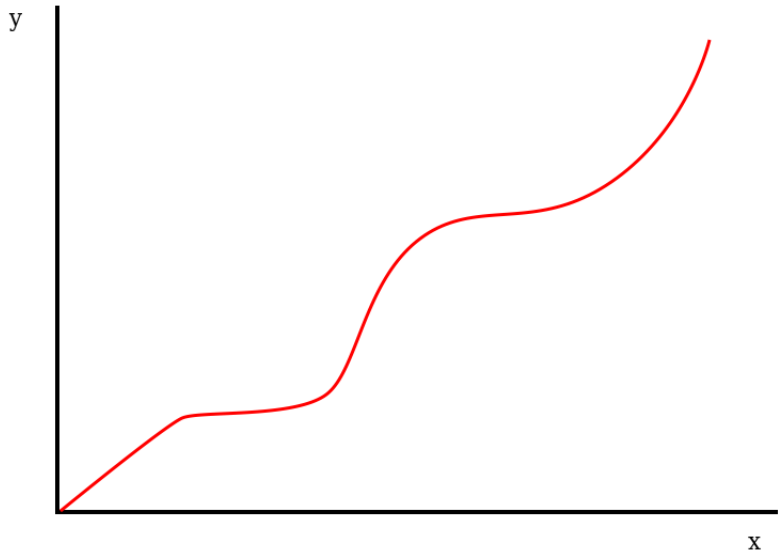
```haskell
class SemiLattice a where
  (\/)   :: a -> a -> a
  bottom :: a

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])
```

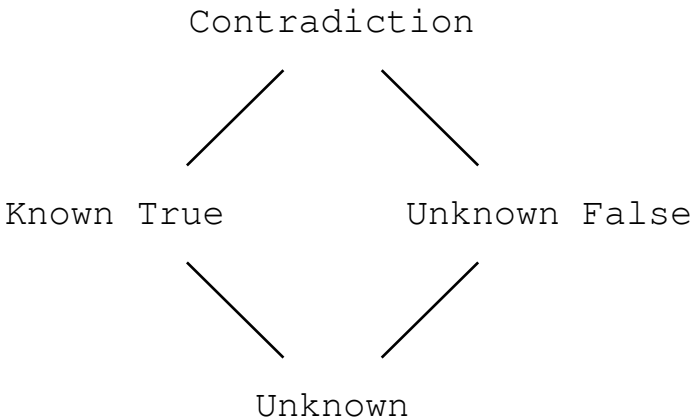
Cells hold semilattices
Propagators join information in

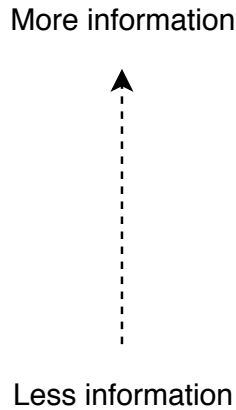
Monotonicity

f is monotone if

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


y
x


Contradiction
Known True
Unknown False
Unknown


More information
Less information

Perhaps
Sets (intersection or union)
Intervals
Bidirectional equations
many more

There's a lot more to say

Even more laziness

Search

Unification

Integer linear programming

SAT solving

many many more

Finding **principled abstractions**
didn't just solve our problems

Thanks for listening!

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

References

Art of the propagator:
https://dspace.mit.edu/handle/1721.1/44215
Alexey Radul's PhD Thesis:
https://dspace.mit.edu/handle/1721.1/54635
Edward Kmett at Boston Haskell:
https://www.youtube.com/watch?v=DyPzPeOPgUE
George Wilson on semi-lattices:
https://www.youtube.com/watch?v=VXl0EEd8IcU

Implementations

Fancy experimental implementation:
https://github.com/ekmett/guanxi
Propagators in Haskell
https://github.com/ekmett/propagators
Propagators in Clojure:
https://github.com/tgk/propaganda