Propagators: An Introduction

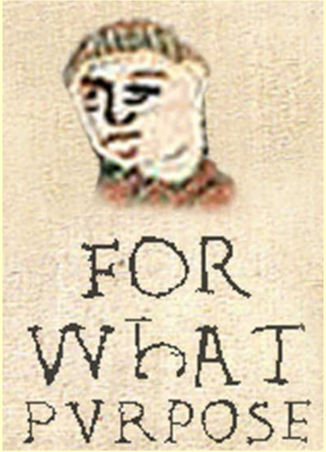
George Wilson

Data61/CSIRO

george.wilson@data61.csiro.au

November 14, 2017

What?



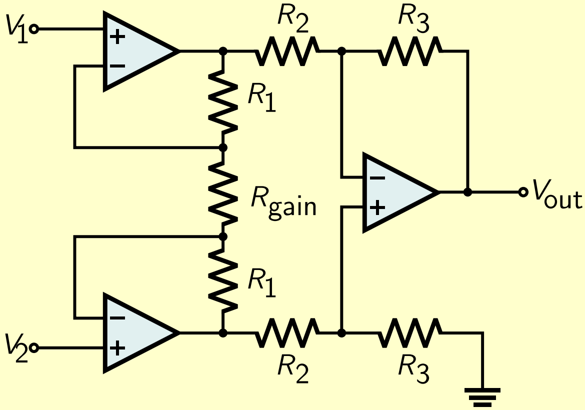
Why?

Beginnings as early as the 1970's at MIT

- Guy L. Steele Jr.
- Gerald J. Sussman
- Richard Stallman

More recently:

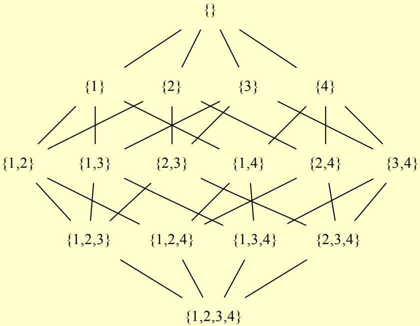
- Alexey Radul



```
(define (map f xs)
  (cond ((null? xs) '())
        (else (cons (f (car xs))
                    (map f (cdr xs))))))
```

And then

- Edward Kmett



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

They're related to many areas of research, including:

- Logic programming (particularly Datalog)
- Constraint solvers
- Conflict-Free Replicated Datatypes
- LVars
- Programming language theory
- And Spreadsheets!

They have advantages:

- are extremely expressive
- lend themselves to parallel and distributed evaluation
- allow different strategies of problem-solving to cooperate

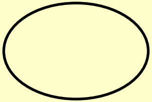
Propagators

The *propagator model* is a model of computation
We model computations as *propagator networks*

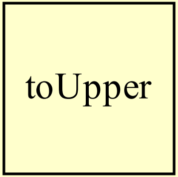
The *propagator model* is a model of computation
We model computations as *propagator networks*

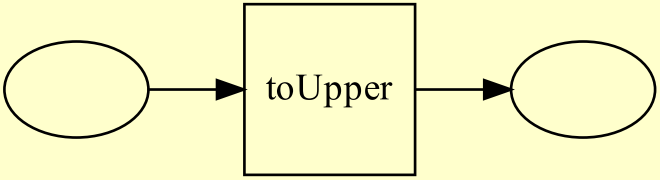
A propagator network comprises

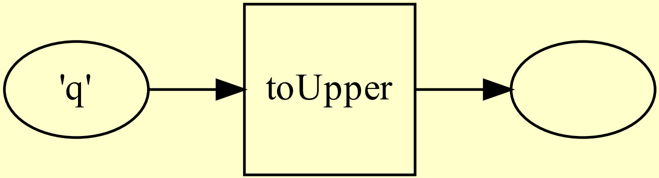
- cells
- propagators
- connections between cells and propagators

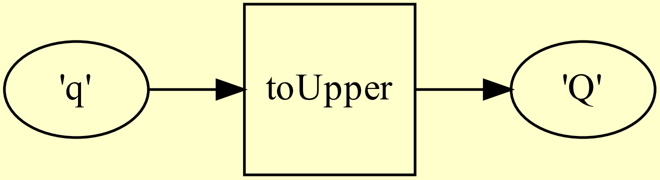


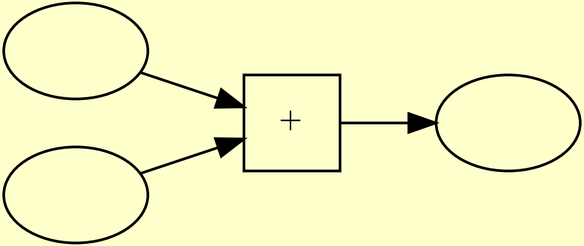
3

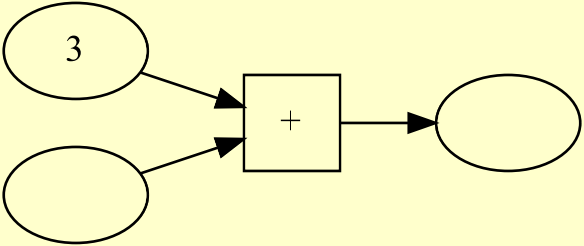

toUpper

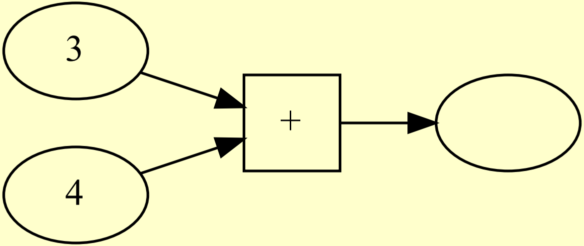

toUpper

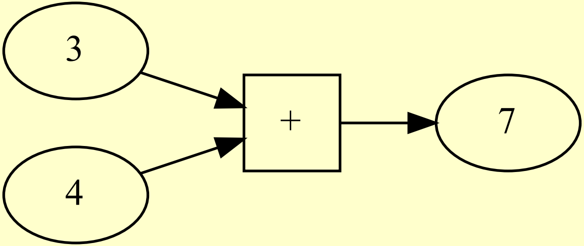

'q'
toUpper


'q'
toUpper
'Q'




3
+


3
4
+


3
4
+
7

$$z \leftarrow x + y$$

$$z = x + y$$

$$7 = x + 4$$

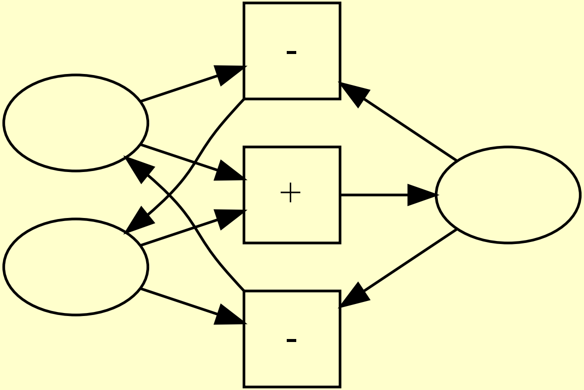
$$7 = 3 + 4$$

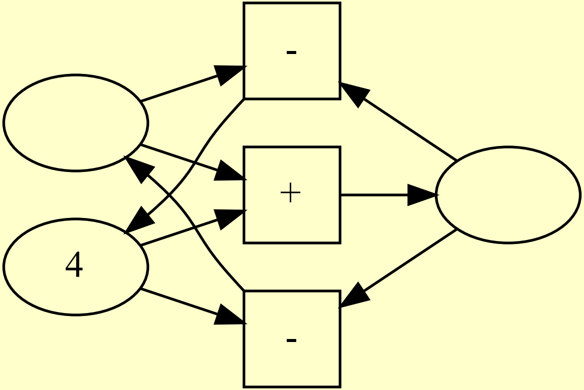
$$z = x + y$$

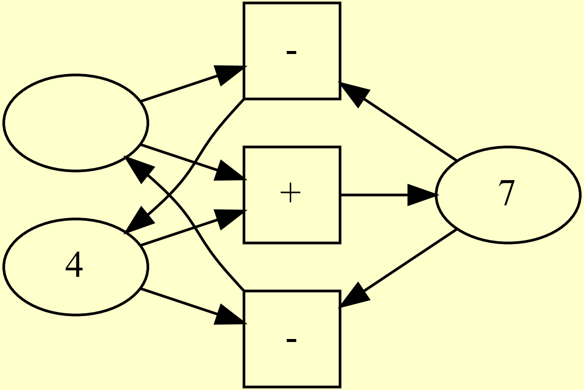
$$z \leftarrow x + y$$

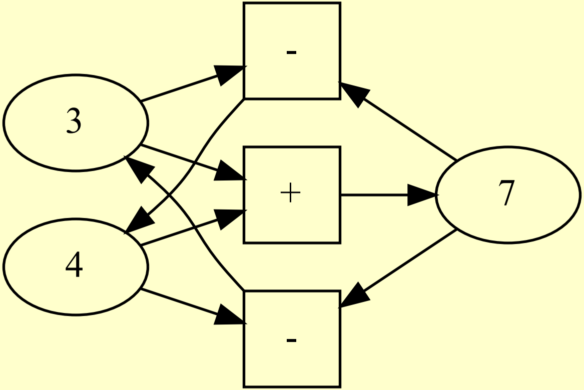
$$x \leftarrow z - y$$

$$y \leftarrow z - x$$



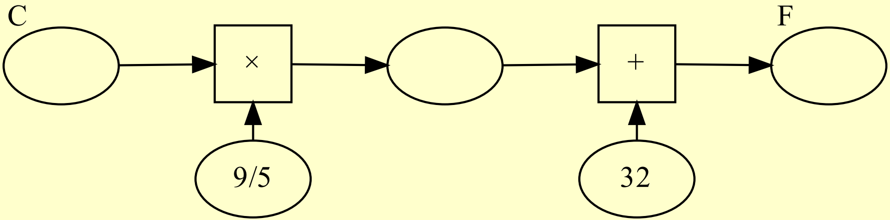

-
+
-
4


-
+
-
4
7

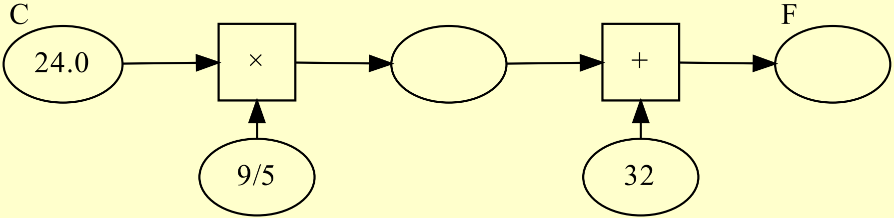

3
4
-
+
-
7

Propagators let us express multi-directional relationships!

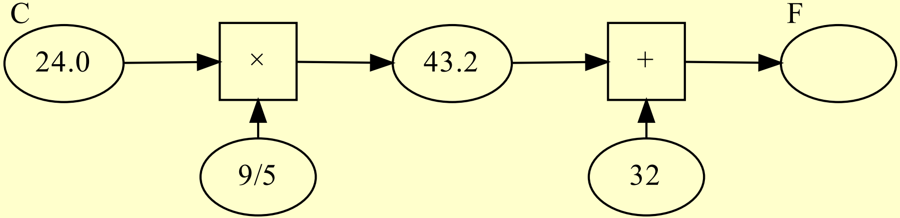
$$^\circ F = {}^\circ C \times \frac{9}{5} + 32$$



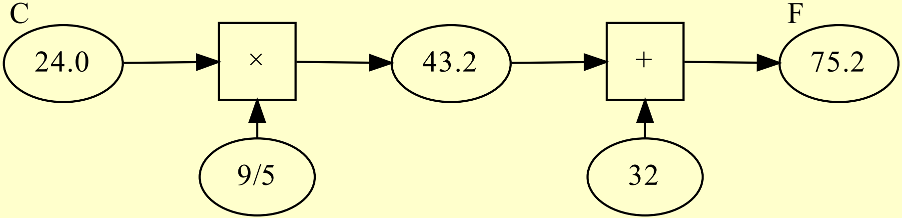
$$^\circ F = {}^\circ C \times \frac{9}{5} + 32$$



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

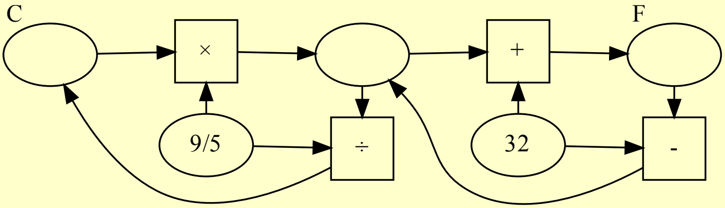


$$^\circ F = {}^\circ C \times \frac{9}{5} + 32$$



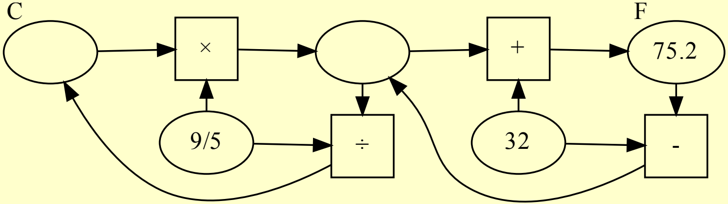
$$°F = °C \times \frac{9}{5} + 32$$

$$°C = (°F - 32) \div \frac{9}{5}$$



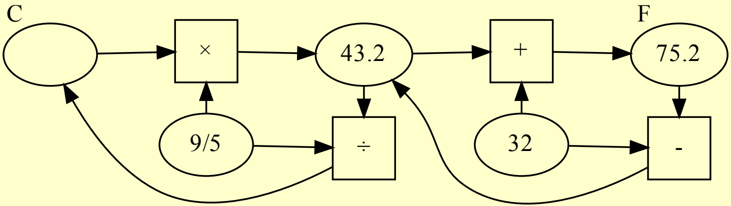
$$°F = °C \times \frac{9}{5} + 32$$

$$°C = (°F - 32) \div \frac{9}{5}$$



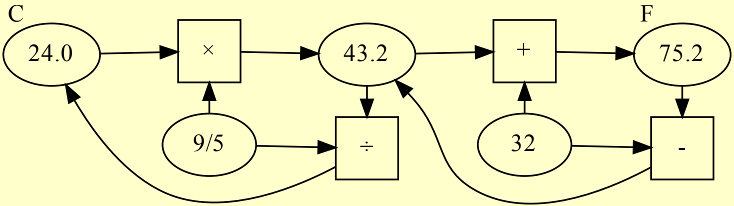
$$°F = °C \times \frac{9}{5} + 32$$

$$°C = (°F - 32) \div \frac{9}{5}$$



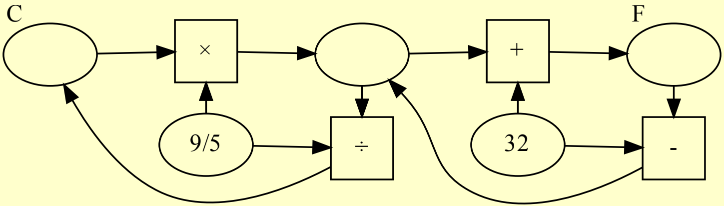
$$°F = °C \times \frac{9}{5} + 32$$

$$°C = (°F - 32) \div \frac{9}{5}$$



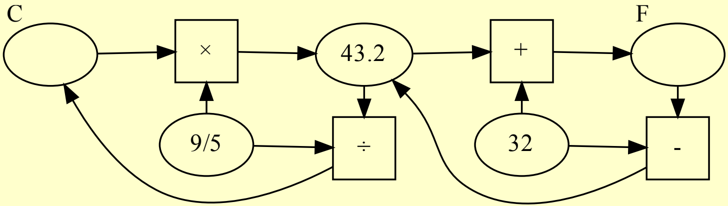
$$^\circ F = {^\circ C} \times \frac{9}{5} + 32$$

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



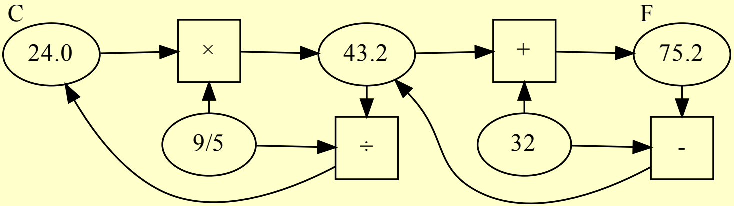
$$^{\circ}F = {^{\circ}C} \times \frac{9}{5} + 32$$

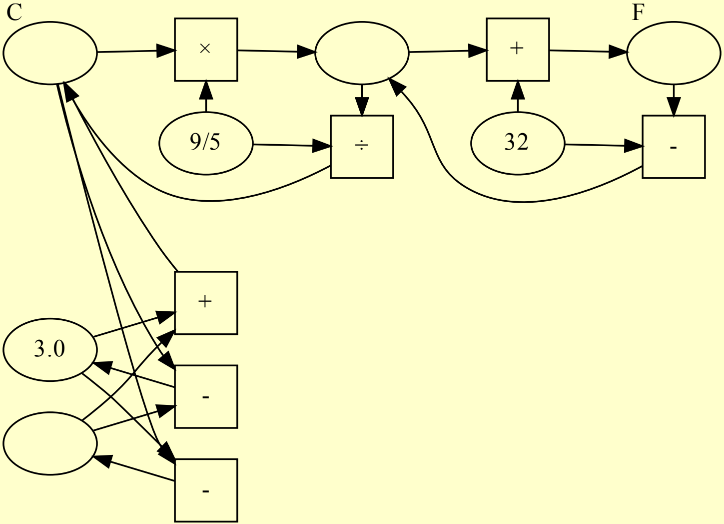
$$^{\circ}C = ({^{\circ}F} - 32) \div \frac{9}{5}$$



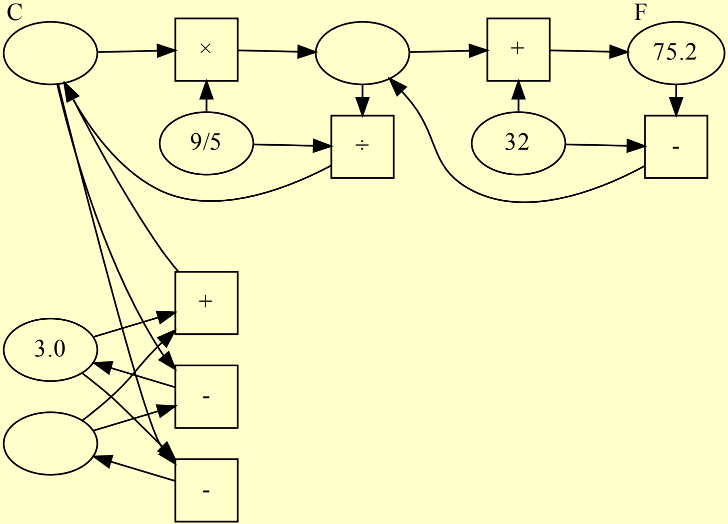
$$°F = °C \times \frac{9}{5} + 32$$

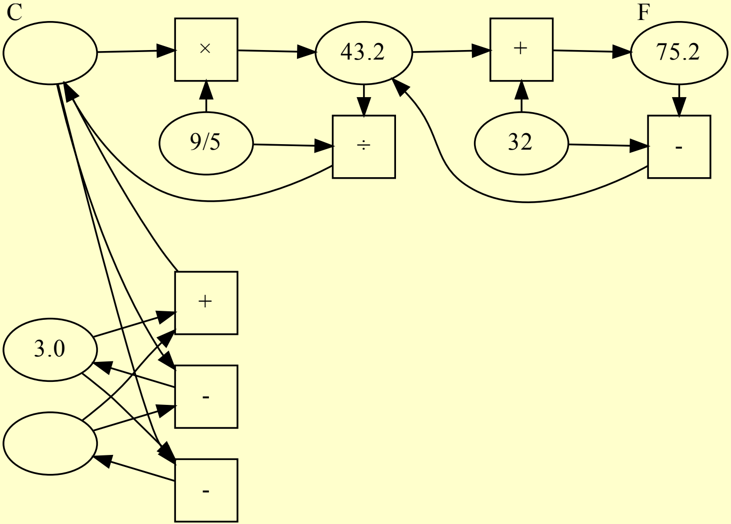
$$°C = (°F - 32) \div \frac{9}{5}$$

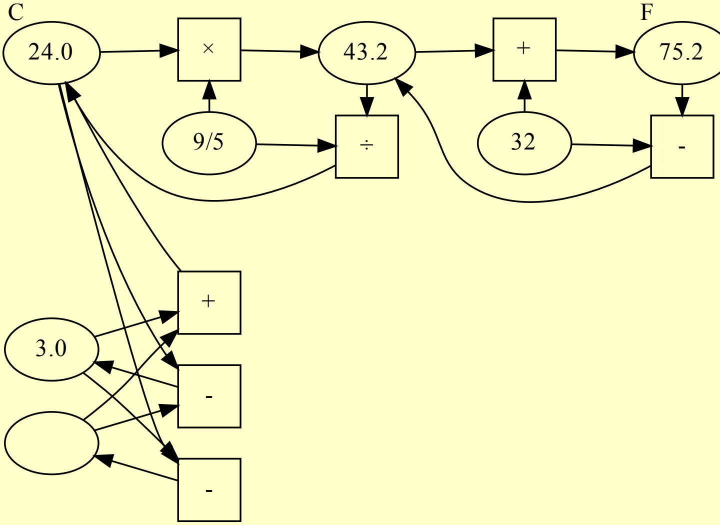


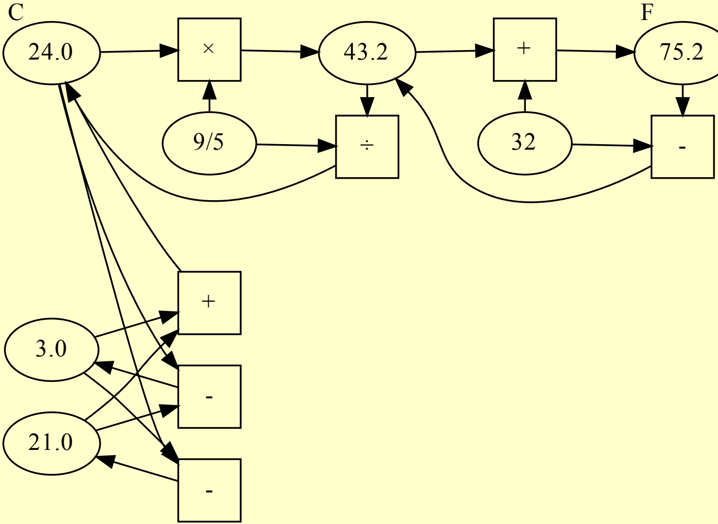


C
×
+
F
9/5
÷
32
-
+
3.0
-
-


C
×
+
F
75.2
9/5
÷
32
-
+
3.0
-
-


C
×
43.2
+
F
75.2
9/5
÷
32
-
+
3.0
-
-


C
24.0
×
43.2
+
F
75.2
9/5
÷
32
-
+
3.0
-
-



C
24.0
×
43.2
+
F
75.2
9/5
÷
32
-
+
3.0
-
21.0
-

We can combine networks into larger networks!

?

Cells *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

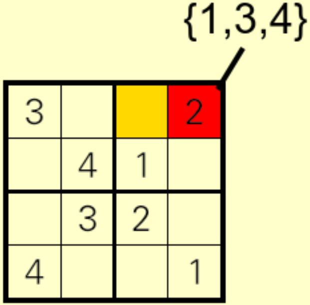
3			2
	4	1	
	3	2	
4			1

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	
	3	2	
4			1

{2,3,4}

{1,2,4}

3			2
	4	1	
	3	2	
4			1

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

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

Cells accumulate information in a *bounded join-semilattice*

Cells accumulate information in a *bounded join-semilattice*

A bounded join-semilattice is:

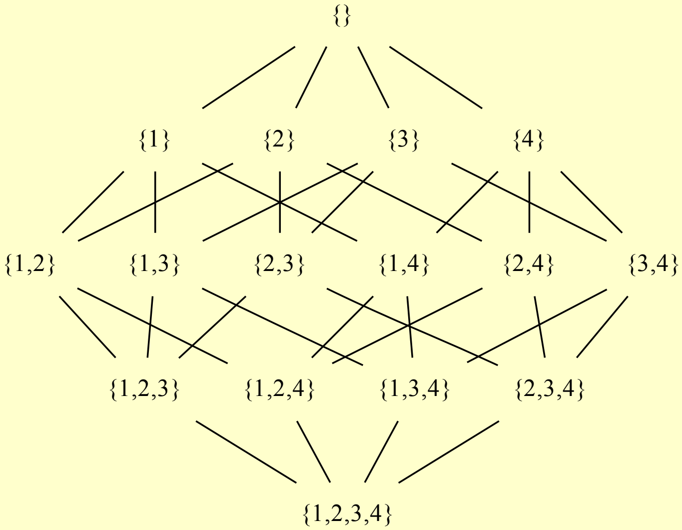
- A *partially ordered set*
- with a least element
- such that any set of elements has a *least upper bound*

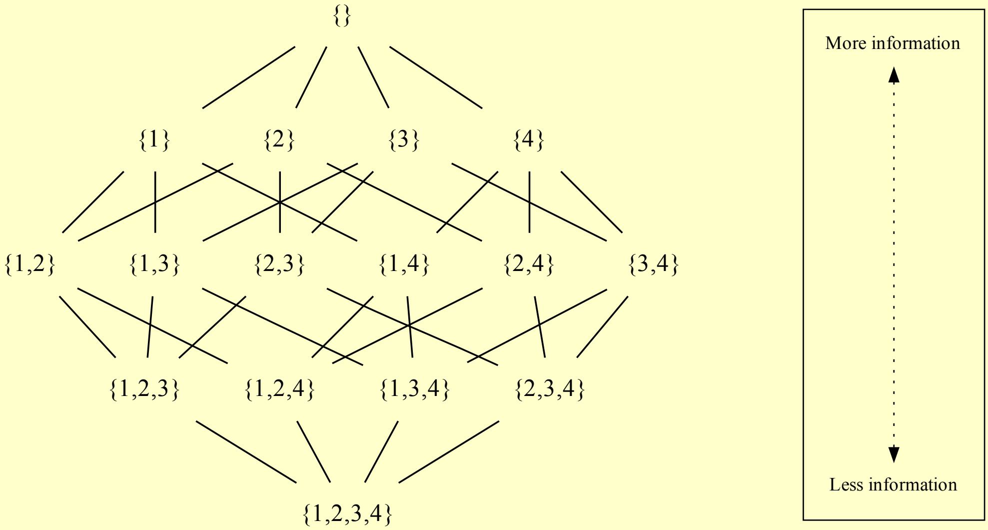
Cells accumulate information in a *bounded join-semilattice*

A bounded join-semilattice is:

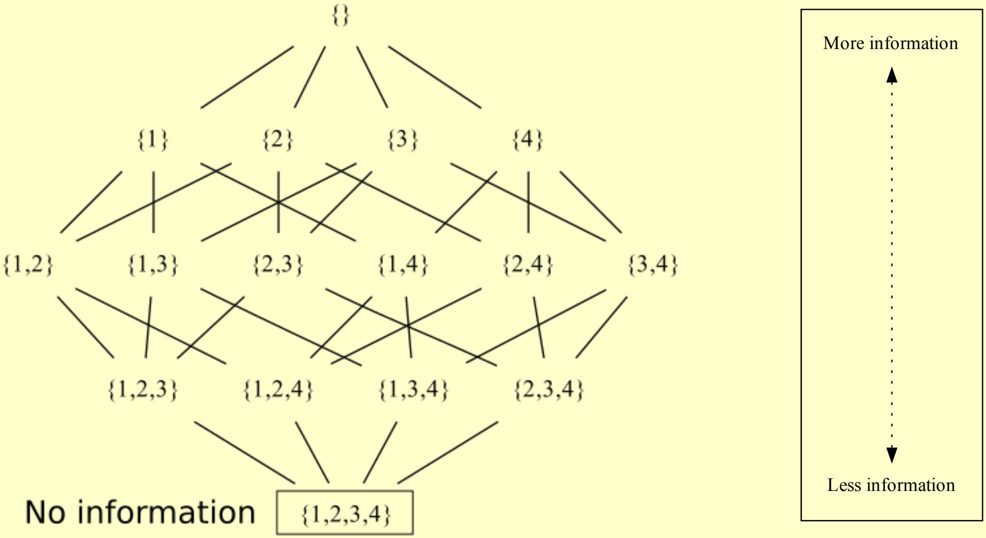
- A *partially ordered set*
- with a least element
- such that any set of elements has a *least upper bound*

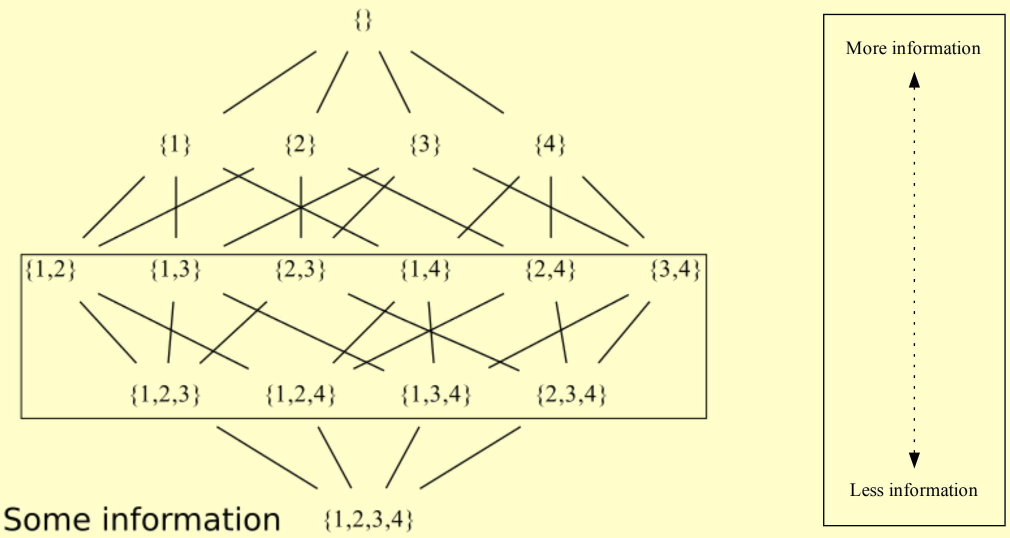
"Least upper bound" is denoted as $\vee$ and is usually pronounced "join"

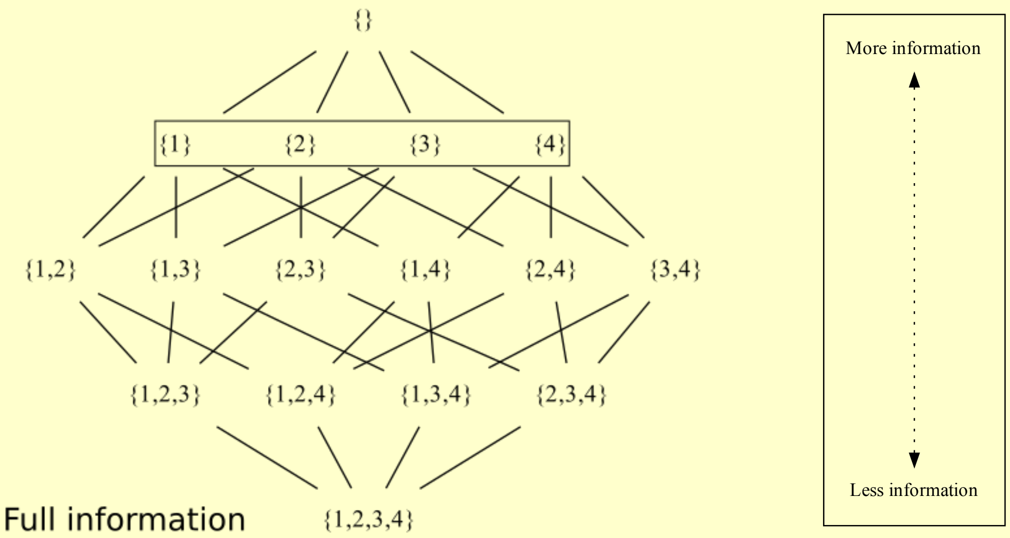

{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
{1,2,3,4}

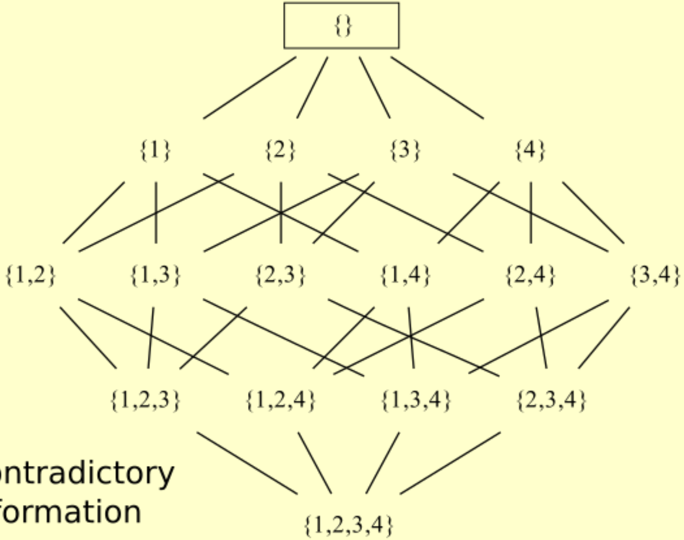


{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
{1,2,3,4}
More information
Less information

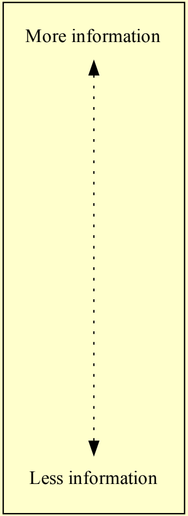

{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
No information
{1,2,3,4}
More information
Less information

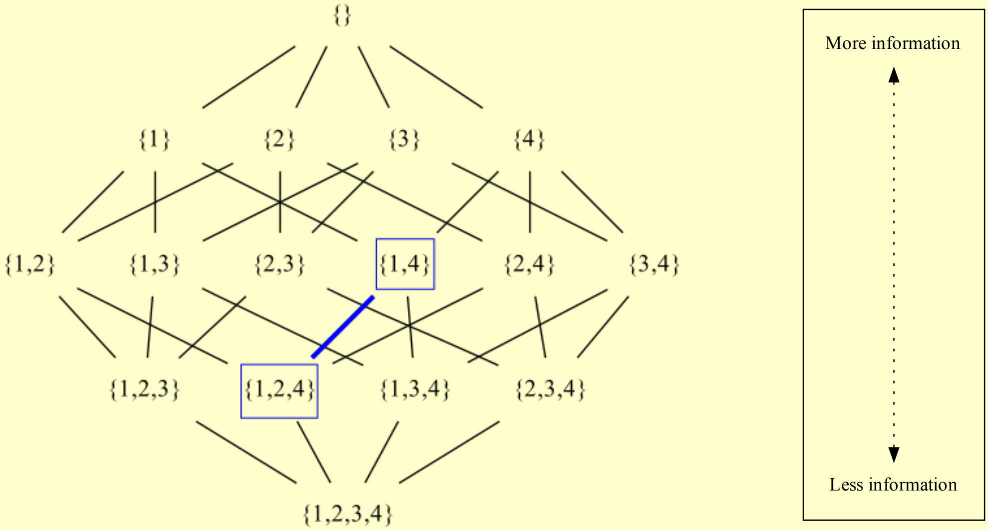

{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
Some information
{1,2,3,4}
More information
Less information


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

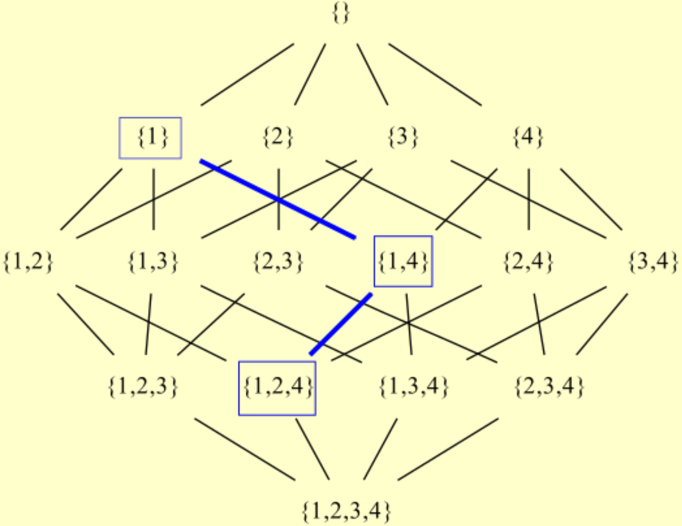


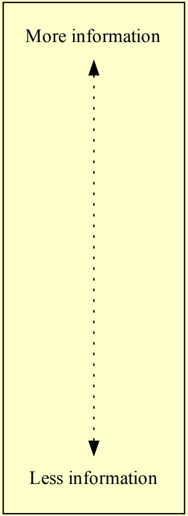
Contradictory information



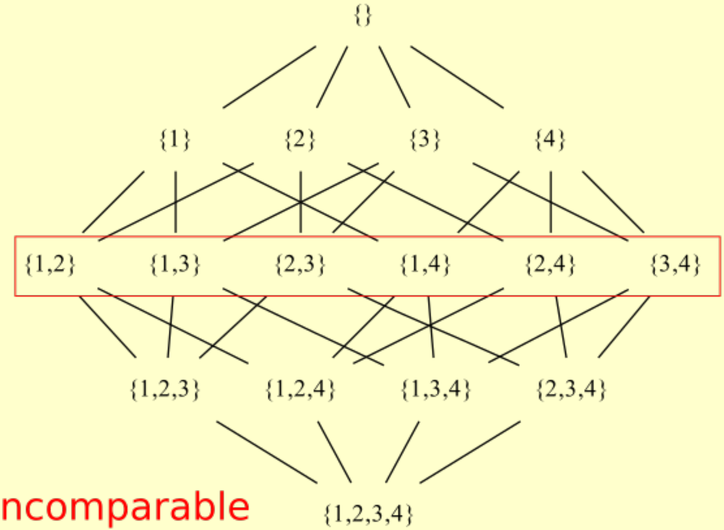

{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
{1,2,3,4}
More information
Less information

{1,2,4} < {1,4}

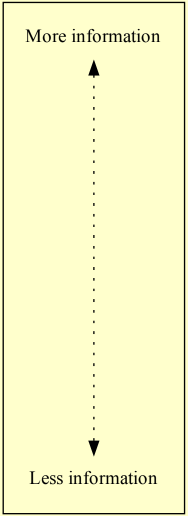

{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
{1,2,3,4}

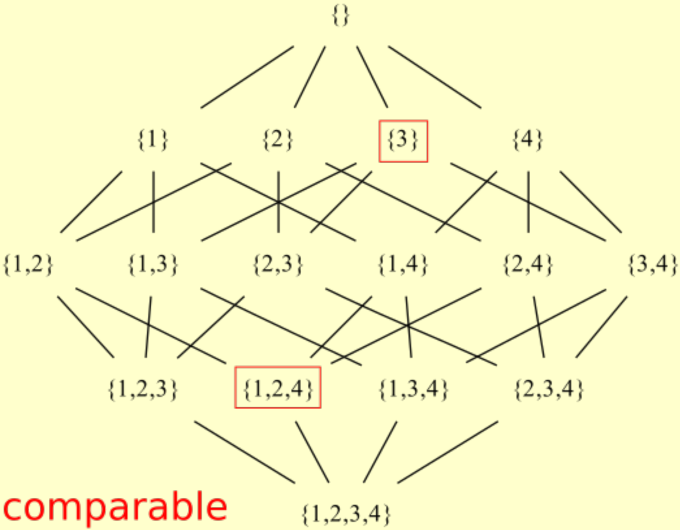

More information
Less information

{1,2,4} < {1,4} < {1}

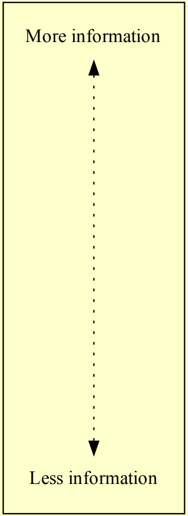


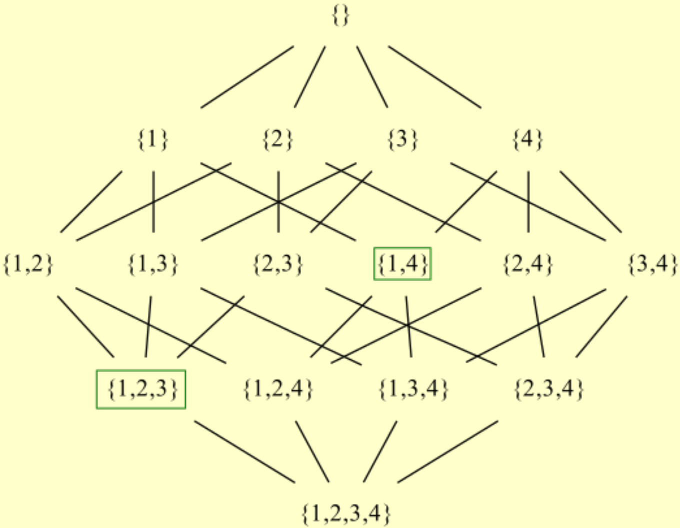
Incomparable

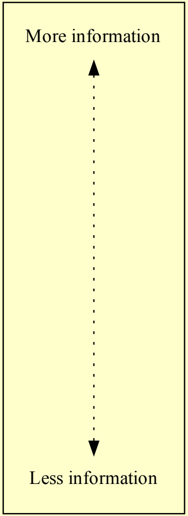




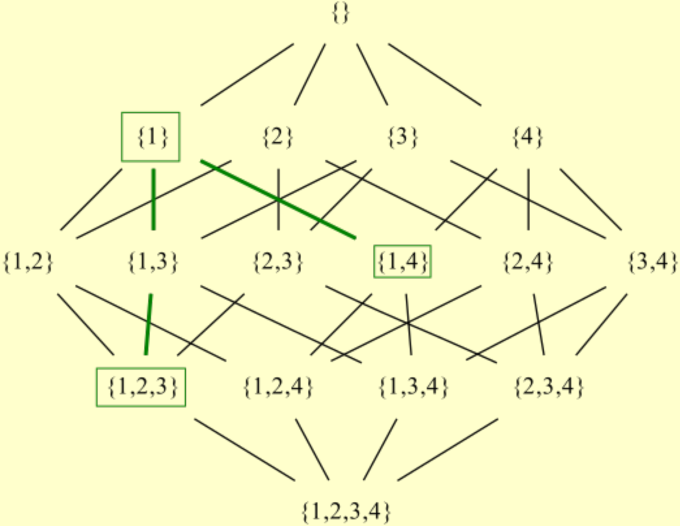
Incomparable

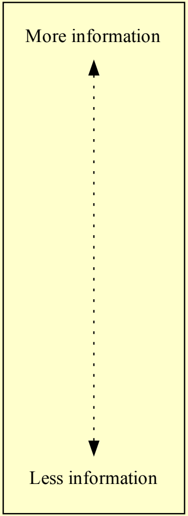



{}
{1}
{2}
{3}
{4}
{1,2}
{1,3}
{2,3}
{1,4}
{2,4}
{3,4}
{1,2,3}
{1,2,4}
{1,3,4}
{2,3,4}
{1,2,3,4}


More information
Less information

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


{}
{1}
{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

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

$\vee$ has useful algebraic properties. It is:

- A monoid
- that's commutative
- and idempotent

Left identity

$$\epsilon \vee x = x$$

Right identity

$$x \vee \epsilon = x$$

Associativity

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

Commutative

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

Idempotent

$$x \vee x = x$$

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

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

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

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


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

newtype SudokuSet = S (Set SudokuVal)
```

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


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

newtype SudokuSet = S (Set SudokuVal)


instance BoundedJoinSemilattice SudokuSet where
  bottom     = S (Set.fromList [One, Two, Three, Four])
  S a \/ S b = S (Set.intersection a b)
```

We don't write values directly to cells

Instead we *join information in*

We don't write values directly to cells
Instead we *join information in*

This makes our propagators *monotone*, meaning that as the input cells gain information, the output cells gain information (or don't change)

We don't write values directly to cells
Instead we *join information in*

This makes our propagators *monotone*, meaning that as the input cells gain information, the output cells gain information (or don't change)

A function $f : A \to B$ where A and B are partially ordered sets is **monotone** if and only if, for all $x, y \in A.\ x \leq y \implies f(x) \leq f(y)$

The bounded join-semilattice laws and monotonicity, combined with the finiteness of our lattices, make propagator networks deterministic,
even in the face of parallelism and distribution

The bounded join-semilattice laws and monotonicity, combined with the finiteness of our lattices, make propagator networks deterministic,
even in the face of parallelism and distribution

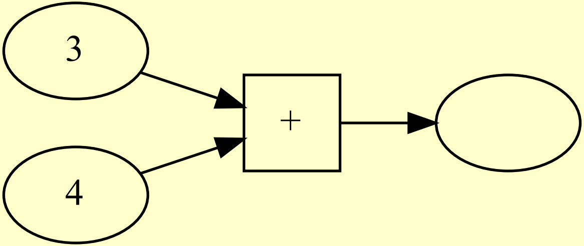
Bounded join-semilattices are already popular in the distributed systems world
See: Conflict Free Replicated Datatypes

The bounded join-semilattice laws and monotonicity, combined with the finiteness of our lattices, make propagator networks deterministic,
even in the face of parallelism and distribution

Bounded join-semilattices are already popular in the distributed systems world
See: Conflict Free Replicated Datatypes

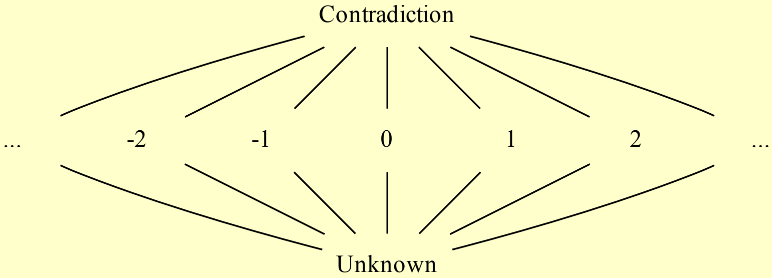
We can relax these constraints in a few different directions

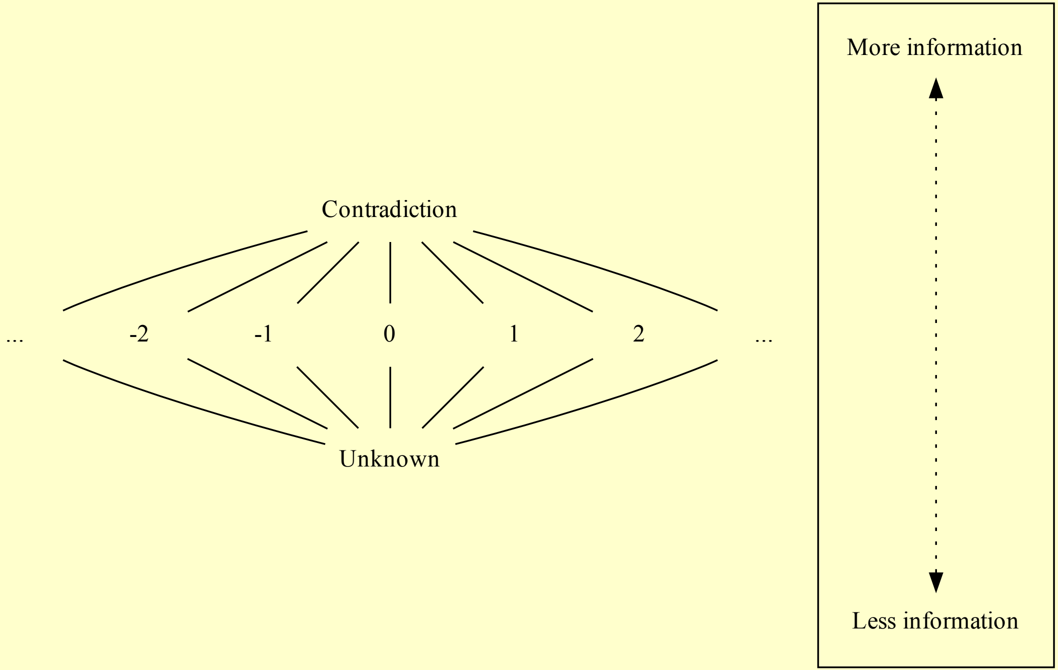
?

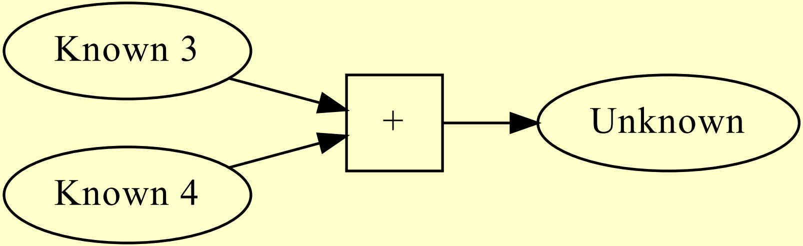

3
4
+

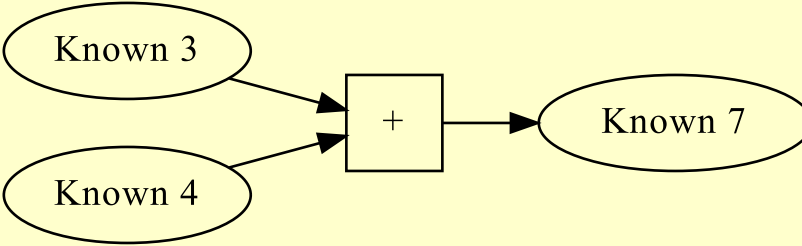
```haskell
data Perhaps a = Unknown | Known a | Contradiction
```

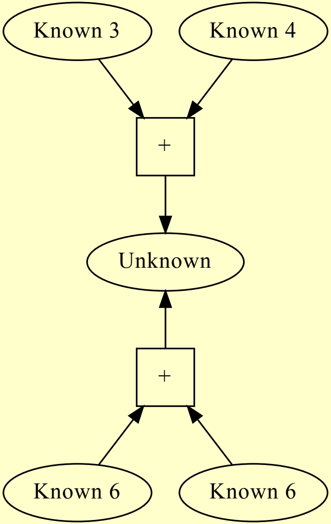
```haskell
data Perhaps a = Unknown | Known a | Contradiction

instance Eq a => BoundedJoinSemiLattice (Perhaps a) where
  bottom = Unknown
  (\/) Unknown x             = x
  (\/) x       Unknown       = x
  (\/) Contradiction _       = Contradiction
  (\/) _      Contradiction = Contradiction
  (\/) (Known a) (Known b) =
    if a == b
      then Known a
      else Contradiction
```

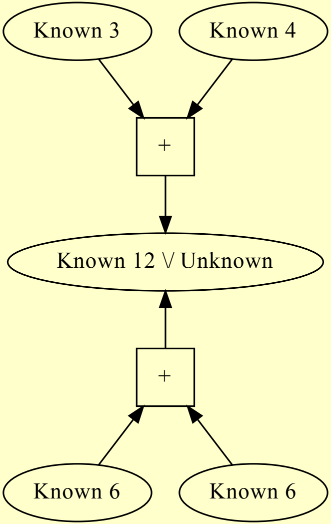

Contradiction
...
-2
-1
0
1
2
...
Unknown

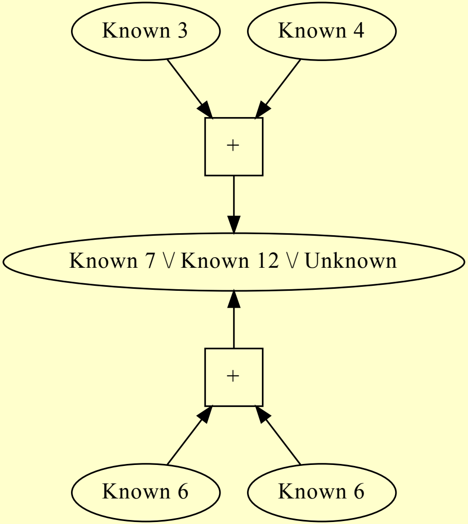

Contradiction
...
-2
-1
0
1
2
...
Unknown
More information
Less information

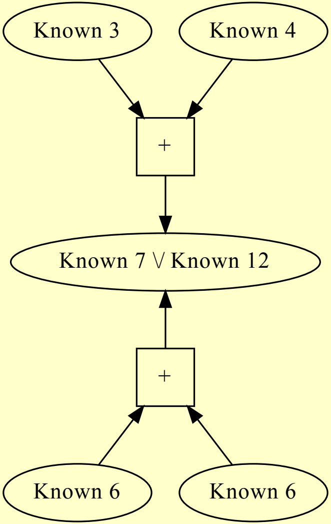

Known 3
Known 4
+
Unknown

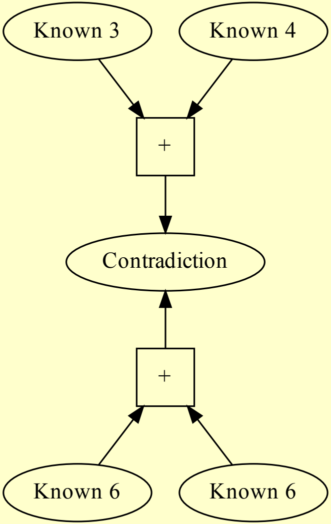

Known 3
Known 4
+
Known 7


Known 3
Known 4
+
Unknown
+
Known 6
Known 6


Known 3
Known 4
+
Known 12 ∨ Unknown
+
Known 6
Known 6


Known 3
Known 4
+
Known 7 ∨ Known 12 ∨ Unknown
+
Known 6
Known 6


Known 3
Known 4
+
Known 7 \/ Known 12
+
Known 6
Known 6



Known 3
Known 4
+
Contradiction
+
Known 6
Known 6

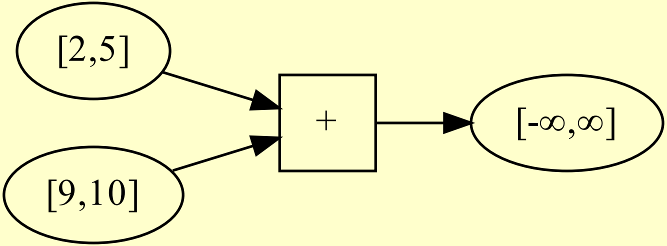
There are loads of other bounded join-semilattices too!

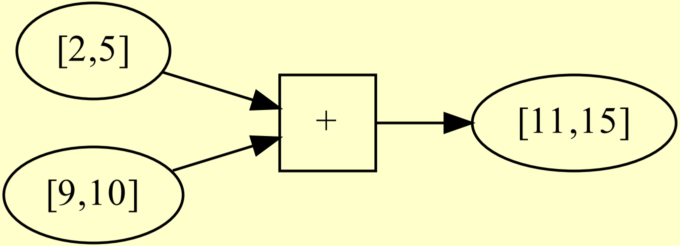
$$[1, 5]$$

$$[1,5] \cap [2,7] = [2,5]$$

$$[1,5] \cap [2,7] = [2,5]$$

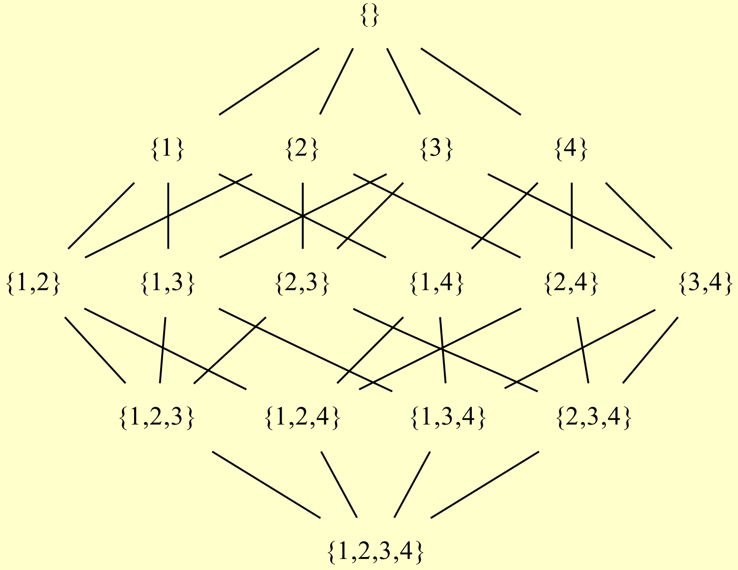
$$[2,5] + [9,10] = [11,15]$$

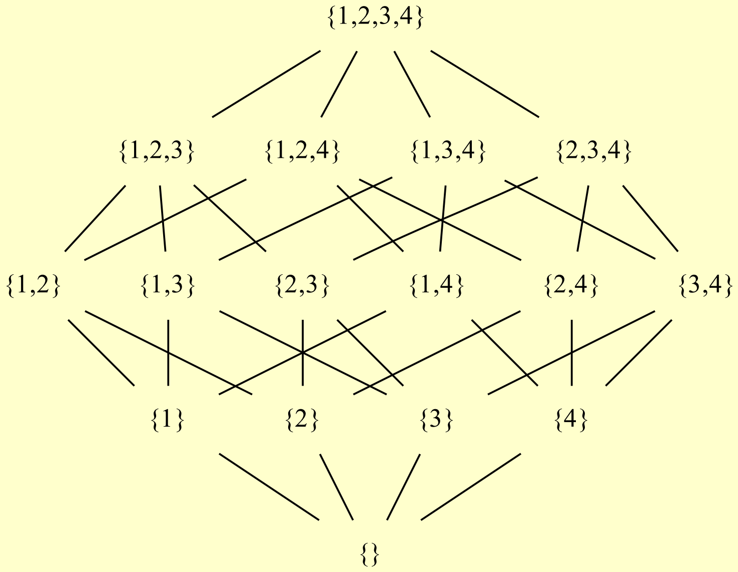

[2,5]
[9,10]
+
[-∞,∞]


[2,5]
[9,10]
+
[11,15]

We can use this to combine multiple imprecise measurements

What other bounded join-semilattices are there?


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

- Set intersection
- Set union
- Interval arithmetic
- `Perhaps`

And so many more!

- Set intersection
- Set union
- Interval arithmetic
- `Perhaps`

And so many more!

?

What happens when we hit contradiction?

What happens when we hit contradiction?

:(

What happens when we hit contradiction?

:(

Monotonicity means that contradiction propagators all over the place

If we track the provenance of information,
we can help identify the source of contradiction

If we track the provenance of information,
we can help identify the source of contradiction

Then we can keep track of which subsets of the information are consistent

and which are inconsistent

$$[2, 5] \cap [3, 7] \cap [6, 9] = []$$

$$[2,5] \cap [3,7] \cap [6,9] = []$$

$$[2,5] \cap [3,7] = [3,5]$$

$$[2,5] \cap [3,7] \cap [6,9] = []$$

$$[2,5] \cap [3,7] = [3,5]$$

$$[3,7] \cap [6,9] = [6,7]$$

$$[2,5] \cap [3,7] \cap [6,9] = []$$

$$[2,5] \cap [3,7] = [3,5]$$

$$[3,7] \cap [6,9] = [6,7]$$

$$[2,5] \cap [6,9] = []$$

$$[2,5] \cap [3,7] \cap [6,9] = []$$

$$[2,5] \cap [3,7] = [3,5]$$

$$[3,7] \cap [6,9] = [6,7]$$

$$[2,5] \cap [6,9] = []$$

Consistent subsets:

$$\{\}$$
$$\{[2,5]\}$$
$$\{[3,7]\}$$
$$\{[6,7]\}$$
$$\{[2,5],[3,7]\}$$
$$\{[3,7],[6,9]\}$$

Maximal consistent subsets:

$$\{[2,5],[3,7]\}$$
$$\{[3,7],[6,9]\}$$

$$[2,5] \cap [3,7] \cap [6,9] = []$$

$$[2,5] \cap [3,7] = [3,5]$$

$$[3,7] \cap [6,9] = [6,7]$$

$$[2,5] \cap [6,9] = []$$

Consistent subsets:

$\{\}$
$\{[2,5]\}$
$\{[3,7]\}$
$\{[6,7]\}$
$\{[2,5],[3,7]\}$
$\{[3,7],[6,9]\}$

Maximal consistent subsets:

$\{[2,5],[3,7]\}$
$\{[3,7],[6,9]\}$

Inconsistent subsets:

$\{[2,5],[6,9]\}$
$\{[2,5],[3,7],[6,9]\}$

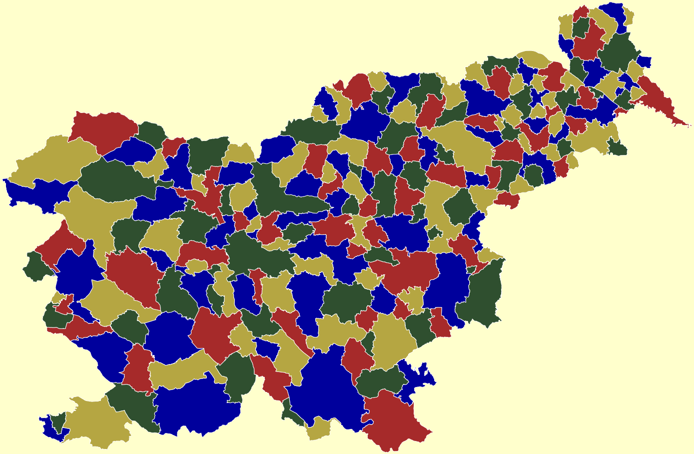
Minimal inconsistent subsets:

$\{[2,5],[6,9]\}$

This concept is something called a *Truth Management System*

Now that we can handle contradiction, we can make guesses!

This lets us encode search problems easily



?

Alexey Radul's work on propagators:

- Art of the Propagator
  `http://web.mit.edu/~axch/www/art.pdf`
- Propagation Networks: A Flexible and Expressive Substrate for Computation
  `http://web.mit.edu/~axch/www/phd-thesis.pdf`

Lindsey Kuper's work on LVars is closely related, and works today:

- Lattice-Based Data Structures for Deterministic Parallel and Distributed Programming
  `https://www.cs.indiana.edu/~lkuper/papers/lindsey-kuper-dissertation.pdf`
- lvish library
  `https://hackage.haskell.org/package/lvish`

Edward Kmett has worked on:

- Making propagators go fast
- Scheduling strategies and garbage collection
- Relaxing requirements (Eg. not requiring a full join-semilattice, admitting non-monotone functions)

Ed's stuff:

- `http://github.com/ekmett/propagators`
- `http://github.com/ekmett/concurrent`
- Lambda Jam talk (Easy mode): `https://www.youtube.com/watch?v=acZkF6Q2XKs`
- Boston Haskell talk (Hard mode): `https://www.youtube.com/watch?v=DyPzPeOPgUE`

In conclusion, propagator networks:

- Admit any Haskell function you can write today ...
- ... and more functions!
- compute bidirectionally
- give us constraint solving and search
- parallelise and distribute

Thanks for listening!