Propagators: An Introduction

George Wilson

Data61/CSIRO
george.wilson@data61.csiro.au

14th November 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
toUpper
toUpper
3 + 3
3 + 4
3 + 4 = 7
\[ z \leftarrow x + y \]
\[ z = x + y \]
7 = x + 4
7 = 3 + 4
\[ z = x + y \]
\[ z \leftarrow x + y \]
\[ x \leftarrow z - y \]
\[ y \leftarrow z - x \]
Propagators let us express bidirectional relationships!
\[ ^\circ F = ^\circ C \times \frac{9}{5} + 32 \]
\[ ^\circ F = ^\circ C \times \frac{9}{5} + 32 \]
\[ ^\circ F = ^\circ C \times \frac{9}{5} + 32 \]
\[ ^\circ F = ^\circ C \times \frac{9}{5} + 32 \]
\[ ^\circ F = ^\circ C \times \frac{9}{5} + 32 \]

\[ ^\circ C = (^\circ F - 32) \div \frac{9}{5} \]
\[ ^\circ F = ^\circ C \times \frac{9}{5} + 32 \]

\[ ^\circ C = (^\circ F - 32) \div \frac{9}{5} \]
\[ ^\circ F = ^\circ C \times \frac{9}{5} + 32 \]

\[ ^\circ C = (^\circ F - 32) \div \frac{9}{5} \]
\[ ^\circ F = ^\circ C \times \frac{9}{5} + 32 \]

\[ ^\circ C = ( ^\circ F - 32 ) \div \frac{9}{5} \]
\[ ^\circ F = ^\circ C \times \frac{9}{5} + 32 \]

\[ ^\circ C = (^\circ F - 32) \div \frac{9}{5} \]
\[ ^\circ F = \ ^\circ C \times \frac{9}{5} + 32 \]

\[ ^\circ C = (\ ^\circ F - 32) \div \frac{9}{5} \]
°F = °C × $\frac{9}{5} + 32$

°C = ($°F - 32$) ÷ $\frac{9}{5}$
\[ C \times \frac{9}{5} \div 32 + F \]
C × \left(\frac{9}{5}\right) \div 32 + 75.2 = 3.0
\[ C \times \frac{9}{5} \div 32 + 32 = F \]

\[ 75.2 \]
\[ C \times \left( \frac{9}{5} \right) - 43.2 + 32 = F \]

\[ 24.0 \times \frac{9}{5} - 43.2 + 32 = 75.2 \]
We can combine networks into larger networks!
Cells *accumulate information* about a value
\{2,3,4\} \cap \{1,3,4\} \cap \\
\{1,2,4\} \cap \{1,2,3,4\}
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 subset 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 subset of elements has a *least upper bound*

“Least upper bound” is denoted as $\vee$ and is usually pronounced “join”
Some information

{} 

{} 

{} 

{} 

\{1\} \quad \{2\} \quad \{3\} \quad \{4\} 

\{1,2\} \quad \{1,3\} \quad \{2,3\} \quad \{1,4\} \quad \{2,4\} \quad \{3,4\} 

\{1,2,3\} \quad \{1,2,4\} \quad \{1,3,4\} \quad \{2,3,4\} 

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

Less information

Full information

\{1,2,3,4\}
Contradictory information

{}
\{1,2,4\} < \{1,4\}
\{1,2,4\} < \{1,4\} < \{1\}
\{1,2,3\} \lor \{1,4\}
\{1, 2, 3\} \vee \{1, 4\} = \{1\}
\( \lor \) has useful algebraic properties. It is:

- A monoid
- that’s commutative
- and idempotent
Left identity
\[ \epsilon \lor x = x \]

Right identity
\[ x \lor \epsilon = x \]

Associativity
\[ (x \lor y) \lor z = x \lor (y \lor z) \]

Commutative
\[ x \lor y = y \lor x \]

Idempotent
\[ x \lor x = x \]
class BoundedJoinSemilattice a where
    bottom :: a
    (\/) :: a -> a -> a
class BoundedJoinSemilattice a where
  bottom :: a
  (\/) :: a -> a -> a

data SudokuVal = One | Two | Three | Four
  deriving (Eq, Ord, Show)
class BoundedJoinSemilattice a where
    bottom :: a
    (\/) :: a -> a -> a

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

newtype SudokuSet = S (Set SudokuVal)
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 $x, y \in A$. $x \leq y \implies f(x) \leq f(y)$
All our lattices so far have been finite.
Thanks to these properties:

- the bounded join-semilattice laws
- the finiteness of our lattice
- the monotonicity of our propagators

our propagator networks will yield with a deterministic answer, in finite time, regardless of parallelism and distribution.
Thanks to these properties:

- the bounded join-semilattice laws
- the finiteness of our lattice
- the monotonicity of our propagators

our propagator networks will yield with a deterministic answer, in finite time, regardless of parallelism and distribution.

Bounded join-semilattices are already popular in the distributed systems world.
See: Conflict Free Replicated Datatypes
Thanks to these properties:

- the bounded join-semilattice laws
- the finiteness of our lattice
- the monotonicity of our propagators

our propagator networks will yield with a deterministic answer, in finite time, regardless 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.
Our lattices only need the *ascending chain condition*
data Perhaps a = Unknown | Known a | Contradiction
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
Known 3 + Known 4

Contradiction

Known 6 + Known 6
There are loads of other bounded join-semilattices too!
[1, 5]
\[ [1, 5] \cap [2, 7] = [2, 5] \]
\[ [1, 5] \cap [2, 7] = [2, 5] \]

\[ [2, 5] + [9, 10] = [11, 15] \]
\([ [2,5] ]\) and \([ [9,10] ]\) are added together to result in \([ -\infty, \infty ]\).
We can use this to combine multiple imprecise measurements
What other bounded join-semilattices are there?
\{1,2,3,4\}

\{1,2,3\} \quad \{1,2,4\} \quad \{1,3,4\} \quad \{2,3,4\}

\{1,2\} \quad \{1,3\} \quad \{2,3\} \quad \{1,4\} \quad \{2,4\} \quad \{3,4\}

\{1\} \quad \{2\} \quad \{3\} \quad \{4\}

\{\}
• Set intersection or union
• Interval intersection
• Perhaps

And so many more!
• Set intersection or union
• Interval intersection
• Perhaps

And so many more!
What happens when we hit contradiction?
What happens when we hit contradiction?

:(

(:(
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.
\[ [2, 5] \cap [3, 7] \cap [6, 9] = [] \]
\[ [2, 5] \cap [3, 7] \cap [6, 9] = [] \]
\[ [2, 5] \cap [3, 7] = [3, 5] \]
\[ [2, 5] \cap [3, 7] \cap [6, 9] = [] \]

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

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

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

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

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

Consistent subsets:
\[
\begin{align*}
\{ & \} \\
\{ [2, 5] \} \\
\{ [3, 7] \} \\
\{ [6, 9] \} \\
\{ [2, 5], [3, 7] \} \\
\{ [3, 7], [6, 9] \}
\end{align*}
\]
\[ [2, 5] \cap [3, 7] \cap [6, 9] = \emptyset \]
\[ [2, 5] \cap [3, 7] = [3, 5] \]
\[ [3, 7] \cap [6, 9] = [6, 7] \]
\[ [2, 5] \cap [6, 9] = \emptyset \]

Consistent subsets:
\[
\emptyset \\
\{[2, 5]\} \\
\{[3, 7]\} \\
\{[6, 9]\} \\
\{[2, 5],[3, 7]\} \\
\{[3, 7],[6, 9]\} \\
\]

Maximal consistent subsets:
\[
\{[2, 5],[3, 7]\} \\
\{[3, 7],[6, 9]\} \\
\]

Inconsistent subsets:
\[
\{[2, 5],[6, 9]\} \\
\{[2, 5],[3, 7],[6, 9]\} \\
\{[3, 7],[6, 9]\} \\
\]

Minimal inconsistent subsets:
\[
\{[2, 5],[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, 9]\}  
- \{[2, 5], [3, 7]\}  
- \{[3, 7], [6, 9]\}

Inconsistent subsets:
- \{[2, 5], [6, 9]\}  
- \{[2, 5], [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, 9]\}  
- \{[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]\}

Maximal consistent subsets:
- \{[2, 5], [3, 7]\}  
- \{[3, 7], [6, 9]\}
This concept is something called a *Truth Management System*
Now that we can handle contradiction, we can make guesses!
Now that we can handle contradiction, we can make guesses!

This lets us encode search problems easily
We can relax some of our conditions in certain circumstances
We can turn any expression tree into a propagator network.
There will only ever be one writer to a cell.

$$(5 + 2) \times (x + y)$$
Wrapping up
Alexey Radul’s work on propagators:

- **Art of the Propagator**
  

- **Propagation Networks: A Flexible and Expressive Substrate for Computation**
  
Lindsey Kuper’s work on LVars is closely related, and works today:

- Lattice-Based Data Structures for Deterministic Parallel and Distributed Programming

- 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/propagators)
- [http://github.com/ekmett/concurrent](http://github.com/ekmett/concurrent)
- Lambda Jam talk (Normal mode):
  [https://www.youtube.com/watch?v=acZkF6Q2XKs](https://www.youtube.com/watch?v=acZkF6Q2XKs)
- Boston Haskell talk (Hard mode):
  [https://www.youtube.com/watch?v=DyPzPeOPgUE](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
- mix all this stuff together
- parallelise and distribute
Thanks for listening!