“TOWARDS” HOMOMORPHIC COMPUTATION FOR DISTRIBUTED COMPUTING

Christopher S. Meiklejohn
Kevin Clancy
Heather Miller

Université catholique de Louvain
Instituto Superior Técnico
Northeastern University
WHY COORDINATION-FREE PROGRAMMING?

1. Cost of coordination
   - Increasing latency in geo-replicated applications

2. CALM result and beyond [CIDR 2011]
   - Convergence guaranteed with a combination of lattice-based programming and monotone logic
   - Regardless of network anomalies: message duplication and/or message reordering

3. Key insights:
   - $v_1 \sqsubseteq v_2$ means that $v_1$ approximates $v_2$ ($v_2$ contains everything in $v_1$ and possibly more)
   - “Once something has happened, it continues to have happened…”
   - Bloom’s protocol programming, LVars “Freeze After Writing"
Logic and Lattices for Distributed Programming

Neil Conway  
William R. Marczak  
Peter Alvaro  
Joseph M. Hellerstein  
UC Berkeley  
nc@cs.berkeley.edu  
wmr@cs.berkeley.edu  
palvaro@cs.berkeley.edu  
hellerstein@cs.berkeley.edu

David Maier  
Portland State University  
maier@cs.pdx.edu

ABSTRACT
In recent years there has been interest in achieving application-level consistency without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to implement consistency using a distributed storage service. This paper considers a CALM (Code and Logic as a Module) distributed storage service that takes advantage of lattices to achieve strong consistency automatically without coordination.

In this paper we take advantage of lattices to implement a distributed storage service. This service is CALM-based and is designed to support efficient evaluation of lattice-based code using well-defined strategies for logic programming. Finally, we use Bloom filters to develop a new distributed storage service that is able to support efficient evaluation of lattice-based code using well-defined strategies for logic programming. Finally, we use Bloom filters to develop a new distributed storage service that is able to support efficient evaluation of lattice-based code using well-defined strategies for logic programming. Finally, we use Bloom filters to develop a new distributed storage service that is able to support efficient evaluation of lattice-based code using well-defined strategies for logic programming. Finally, we use Bloom filters to develop a new distributed storage service that is able to support efficient evaluation of lattice-based code using well-defined strategies for logic programming. Finally, we use Bloom filters to develop a new distributed storage service that is able to support efficient evaluation of lattice-based code using well-defined strategies for logic programming.
PROGRAMMING MODEL:
EXAMPLE

A :: Set
B :: Set
B = map \lambda x. x A

Whenever A changes, B is recomputed from A.

A :: OR-Set
B :: OR-Set
B = fold \lambda x \lambda y. x + y \perp A

Combinators can be user defined.

Custom, non-trivial lattices.

Higher-order programming with functions.
CHALLENGES: STATE-OF-THE-ART

1. Combinators are lattice-specific. 
   Existing solutions are either limited to particular lattices or place the onus on the developer.

2. Monotone functions are required for correctness.
   Assumes the developer will implement monotone functions correctly.

3. Incrementality of computations
   Homomorphisms are a special case of monotone functions where function application distributes over join.
   Developers must understand how to express computation as a homomorphism and correctly implement it.

If you follow the rules, you have “correct” programs.

If you don’t…?

Lasp’s proof wasn’t comprehensive enough to catch composition failures.

No formal specification of either systems execution model.

Without monotone checks, program behavior is nondeterministic.
   (i.e. function reads clock, function is antitone, etc.)
CHALLENGES: LATTICES

Bloom

User-provided ADTs

Combinators are user-provided

Combinators must be labeled as either monotone or as a homomorphism

Lasp

Any CRDT implementing a “CRDT” interface is supported

- ordering relation, merge, inflation, etc.

Combinators are built in for one CRDT: the Observed-Remove Set

Higher-order combinator, fold, provided but requires user to ensure monotonicity
CHALLENGES: COMBINATORS

Bloom\textsuperscript{1}

Must be labeled as either monotone or homomorphic

Functions are unchecked as to whether they are monotone, or homomorphic

Lasp

Built in combinators designed to provide monotonicity

Higher-order programming with fold requires unchecked user-implemented function is both monotone and has inverse function
## CHALLENGES: INCREMENTALITY

<table>
<thead>
<tr>
<th>Bloom</th>
<th>Lasp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unchecked annotations on homomorphic functions, and are incrementally evaluated</td>
<td>(Some) built-in combinators are homomorphic, but are not evaluated as so</td>
</tr>
</tbody>
</table>
CONTRIBUTIONS

1. Computational delta objects
   - Generalization of CRDTs to objects have a unified format enabling easier computation
   - Changes expressed as “deltas” that can be derived through object decomposition
   - Combinators are implemented in terms of monotone functions on deltas

2. Monotonicity typing
   - Instantiation of Petricek’s structural coeffects system to detect monotonicity violations

3. Operational semantics for an incremental calculus with lattices
   - Expresses computations as functions from lattice to lattice
   - Incremental evaluation when lattice computations form homomorphisms
   - Generalized semantics for Bloom⁵ and Lasp: previously not formalized
DATA TYPES & COMBINATORS: BLOOM̂

Specified by the user

Must have:
- Values that form a lattice
- Least-upper-bound function
- Combinators

Annotations are unchecked

Figure 4: Example implementation of the `lset` lattice.

```
class Bud::SetLattice < Bud::Lattice
  wrapper_name :lset
  def initialize(x=[]) # Input validation removed for brevity
    @v = x.uniq # Remove duplicates from i
  end
  def merge(i)
    self.class.new(@v | i.reveal)
  end
  morph :intersect do |i|
    self.class.new(@v & i.reveal)
  end
  morph :contains? do |i|
    Bud::BoolLattice.new(@v.member? i)
  end
  monotone :size do
    Bud::MaxLattice.new(@v.size)
  end
end
```
DATA TYPES & COMBINATORS: LASP

Implemented through a “CRDT” interface

Combinators are built into runtime:
- Functional: map, filter
- Set-theoretic: union, product, intersection
- Higher-order: fold

Fold must be monotonic, unchecked
RECONCILING THE “DELTA”

Challenges:

1. What is a useful generalization of the data types in Lasp and Bloom\(^1\)?
   - Generalization to delta lattices, and a special case for non-compositional types called multichains

2. How do we encode a notion of incrementality into the programming model?
   - Each delta lattice can be decomposed into a set of maximal deltas

Key Insights:

1. Monotone functions from lattice A to lattice B
   - \((A_\delta \rightarrow B)\) on deltas decomposed from A, combined using an arbitrary function \((B \rightarrow B \rightarrow B)\)

2. Homomorphism functions from lattice A to lattice B
   - Monotone function where the combining function is \texttt{join} (least-upper-bound)
DELTA LATTICES & MULTICHAI N S

Delta lattices
- Bounded join-semilattices, S
- \( J(S) \), the set of join irreducible elements taken from S, are finitely join-dense
- Maximal chains from \( J(S) \) are mutually exclusive

Multichains \( M(L, T) \)
- Special case of delta lattices
- \( L \rightarrow T \)
- \( L \): a set of unordered labels
- \( T \): a totally ordered set

Any element from the delta lattice can be decomposed into a join of deltas
Deltas are maximal, removing redundancy
Multichain deltas are \( (L, \max(T)) \)
Computational delta lattices $(S, A_δ, F, G)$
- $S$ is a delta lattice
- $A_δ$ is a poset
- $F$ is a function from $S$ to $A_δ$ only defined where $S$ is a join-irreducible element take from $S$
- $G$ is a function from $A_δ$ to $S$ where $G; F$ for some $S$ is the identity function

Computational delta objects $(A, M, Q)$
- $A$ is a computational delta lattice
- $M$ is a set of monotone mutation functions that return deltas that will be joined into the store
- $Q$ is a set of query functions over $A$

For type safety on monotone functions on $A_δ$
Convert from join-irreducible singleton set to $π_1$

A practical instantiation of a CRDT-like lattice-based data type
COMPUTATIONAL DELTA OBJECTS: BLOOM\(^L\)

\(\text{LOrd}[T]\):

- **Multichain:**
  
  \((1, T)\)

- **Mutation functions:**
  
  \(\text{set}(t, m) = (1, t)\)

- **Query functions:**
  
  \(\text{value}(m) = \pi_2 m\)

\(\text{LBool} = \text{LOrd}[2]\)

\(\text{LMax} = \text{LOrd}[\text{NMax}]\)

\(\text{LMin} = \text{LOrd}[\text{NMin}]\)

- **Ordered register**

- **2: ordered using the natural's ordering**

- **Naturals ordered with bottom at zero or infinity**
Computational Delta Objects: CRDTs

**G-Counter:**
*Multichain:*
\((R, \text{NMax})\)

*Mutation functions:*
\(\text{increment}(r, m) = (r, m(r) + 1)\)

*Query functions:*
\(\text{value}(m) = \text{fold}(\pi_2, +)\ a\)

**G-Set[A]:**
*Multichain:*
\((A, 2)\)

*Mutation functions:*
\(\text{add}(a, m) = (a, 1)\)

*Query functions:*
\(\text{value}(m) = \text{map} \pi_1 \ m\)

Grow-only counter is replica to count mappings

Grow-only set is element to 2
COMPUTATIONAL DELTA OBJECTS: LASP

Observed-Remove Set[A]:

Multichain:

\(((A \times T \times D), 2)\)

Mutation functions:

\[
\begin{align*}
\text{add}(a, t, m) &= ((a, t, \bot), 1) \\
\text{rem}(a, t, m) &= ((a, t, \top), 1)
\end{align*}
\]

Query functions:

\[
\text{value}(m) = \text{map } \pi_1 \{ (a, t) \mid (a, t, \bot) \in m \} \setminus \{ (a, t) \mid (a, t, \top) \in m \}
\]
IMPLEMENTING COMBINATORS: CHALLENGES

1. Combinators must be monotone, to ensure convergence

2. Combinators must be from lattice to lattice, to ensure composition
   - Conway et al., “Logic and Lattices for Distributed Programming”, SoCC 2012

3. Combinators must be a lattice homomorphism, to enable incremental evaluation
   - Conway et al., “Logic and Lattices for Distributed Programming”, SoCC 2012
COEFFECTS

Petricek’s structural coeffects system

Extend context with a “coeffect”
- Vector, entry per item in the typing context
- “Scalar” value with an ordering relation
- Composition and a contraction operation

Important points:
- Contraction used to when variables occur multiple times in the body (join)
- Composition used for function application (sum)
MONOTONICITY TYPING

Instantiation of structural coeffects system

Labels represent:
- Arbitrary (?)
- Monotone (+)
- Antitone (-)
- Unknown (~)

Key insights:
- Variable occurrence is monotone
- Function application “composes” monotonicity
- Multiple occurrences are joined with contraction

Figure 3: Scalar composition $\otimes$ and contraction $\oplus$ for monotonicity

Figure 4: Hasse diagram for the partial order $\leq$ on monotonicity scalars
LIFTED MONOTONE FUNCTIONS

Given we want a monotone function from delta lattice $A \rightarrow$ delta lattice $B$...

We write:
- Mapping from $A$ delta to $B$. $f : (A_\delta \rightarrow B)$
- Summing / combining function for $B$’s. $g : (B \rightarrow B \rightarrow B)$

Caveats:
- We assume that there’s a total order on $A_\delta$ as all combining functions may not be commutative.
- Combining function will be applied pairwise.

Execution:
- $\text{fold } (g \circ f) \bot_B \text{ deltas}(A)$
- Decompose $A$ into deltas.
LIFTED HOMOMORPHIC FUNCTIONS

Given we want a homomorphic function from delta lattice $A \rightarrow$ delta lattice $B$...

We write:
- Mapping from $A$ delta to $B$. $f : (A_\delta \rightarrow B)$

Execution:
- $\text{fold } (\sqcup \circ f) \sqcap_B \text{deltas}(A)$

Apply the transformation and combine using the join

Deltas taken from $A$
MONOTONE FUNCTIONS: BLOOM$^L$

**size on LSet**
- $\text{size} :: \text{LSet} \rightarrow \text{LMax}$
- $f : \lambda((a, 1)). \{(1, 1)\}$
- $g : +$

**sum on LSet**
- $\text{sum} :: \text{LSet} \rightarrow \text{LMax}$
- $f : \lambda((a, 1)). \{(1, a)\}$
- $g : +$

Produce a set of deltas for B

We've completed a full specification of Bloom$^L$ monotone functions, without considering the dictionary.

Combining function is applied coordinate-wise
HOMOMORPHIC FUNCTIONS: BLOOM\textsuperscript{L}

\(+ (y) \) on \text{LMax}
\begin{itemize}
\item \( + : \mathbb{N} \rightarrow \text{LMax} \rightarrow \text{LMax} \)
\item \( f : \lambda(1, X). \{(1, x + y)\} \)
\end{itemize}

\(- (y) \) on \text{LMax}
\begin{itemize}
\item \( - : \mathbb{N} \rightarrow \text{LMax} \rightarrow \text{LMax} \)
\item \( f : \lambda(1, X). \{(1, x - y)\} \)
\end{itemize}

\text{map}(g) \) on \text{LSet}
\begin{itemize}
\item \( \text{map} : (A \rightarrow B) \rightarrow \text{LSet} \rightarrow \text{LSet} \)
\item \( f : \lambda((a, 1)). \{(g(a), 1)\} \)
\end{itemize}

\text{filter}(g) \) on \text{LSet}
\begin{itemize}
\item \( \text{filter} : (A \rightarrow \text{Bool}) \rightarrow \text{LSet} \rightarrow \text{LSet} \)
\item \( f : \lambda((a, 1)). \{} \)
\end{itemize}

We've completed a full specification of Bloom\textsuperscript{L} homomorphisms, without considering the dictionary.
Monotone functions: decompose lattice into deltas and apply/combine

Homomorphisms: apply one delta at a time
FUTURE WORK

1. **Typed Structures**
   - Multichains are only sufficient for non-compositional data types
   - Typed records, lift homomorphisms to operate on multichains embedded in a typed record

2. **Binary Operations w/o Fixed Arguments**
   - ex. Lasp cartesian product vs. Bloom¹ cartesian product

3. **Fixed Point Combinator**
   - Links from locations to themselves – but, how expressive can this be?

4. **Type System**
   - ex. verify that monotone functions on deltas have the correct type signature

5. **Proofs**
   - Progress and preservation
   - Correspondence between monolithic and incremental evaluation
STRUCTURAL COEFFECTS: DATAFLOW EXAMPLE

Dataflow language over streams

“pre” operation for accessing a previous item in the stream

Coeffects are naturals where:

- Ordering is standard order on naturals
- Composition is +
- Contraction is “max”
OPERATIONAL SEMANTICS: SYNTAX/EXPRESSIONS

Interaction with the store

Homomorphic and monotone links between store locations

Standard expression reduction rules extended with constants and functional constants
OPERATIONAL SEMANTICS: STORES

Create locations in the store

Monotonic links forward location state decomposed into deltas

Updates store, and propagates deltas forward using links

Homomorphmic links propagate forward only the deltas representing the change
OPERATIONAL SEMANTICS: LINKS

Homomorphic links store pending deltas and a monotone function between store locations.

Monotone function track a monotone function between store locations.
OPERATIONAL SEMANTICS: MLINKS

Computing pending deltas for each monotonic link and combine the results.

Once completed for all deltas, replace the value in the store and forward on new deltas to out-links.
Compute a single delta from the pending deltas in for each homomorphic link

Once a single delta is computed, join the value with the current value in the store and propagate deltas to out-links