“TOWARDS” HOMOMORPHIC COMPUTATION FOR DISTRIBUTED COMPUTING

Christopher S. Meiklejohn
Kevin Clancy
Heather Miller

Université catholique de Louvain
Instituto Superior Técnico
Northeastern University

Northeastern Universit

WHY COORDINATION-FREE PROGRAMMING?

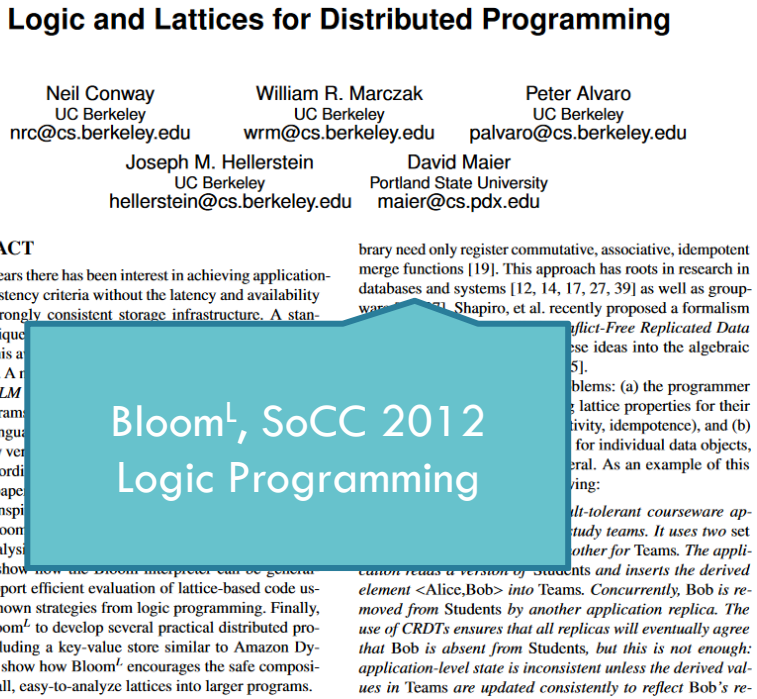
1. Cost of coordination
   - Increasing latency in geo-replicated applications

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:
   - v1 $\sqsubseteq$ v2 means that v1 approximates v2 (v2 contains everything in v1 and possibly more)
   - "Once something has happened, it continues to have happened…"
     - Bloom's protocol programming, LVars "Freeze After Writing"

PROGRAMMING M

Data Structures: Lattices
Programming Model: Lattice Combinators



Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
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 criteria without the latency and availability costs of strongly consistent storage infrastructure. A stan-

In this pape

that takes inspi

eralizes Bloom

CALM analysi

tices. We show

ized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

brary need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as group-

Shapiro, et al. recently proposed a formalism

Conflict-Free Replicated Data

ese ideas into the algebraic

5].

blems: (a) the programmer

lattice properties for their

tivity, idempotence), and (b)

for individual data objects,

eral. As an example of this

ving:

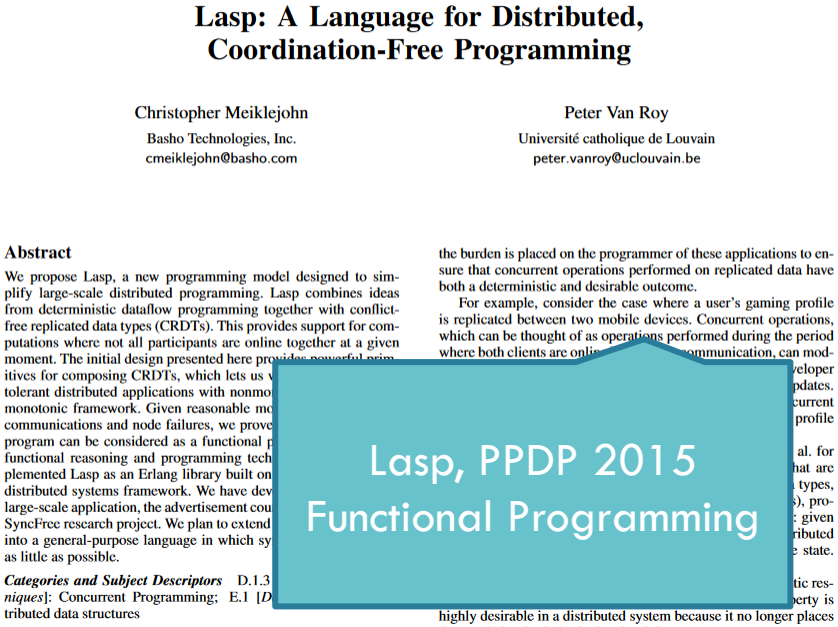
lt-tolerant courseware ap-

tudy teams. It uses two set

other for Teams. *The appli-*

cation reads a version of Students *and inserts the derived element* <Alice,Bob> *into* Teams. *Concurrently,* Bob *is removed from* Students *by another application replica. The use of CRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students, *but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s re-*

Bloom^L, SoCC 2012
Logic Programming



Lasp: A Language for Distributed, Coordination-Free Programming

Christopher Meiklejohn
Basho Technologies, Inc.
cmeiklejohn@basho.com

Peter Van Roy
Université catholique de Louvain
peter.vanroy@uclouvain.be

Abstract

We propose Lasp, a new programming model designed to simplify large-scale distributed programming. Lasp combines ideas from deterministic dataflow programming together with conflict-free replicated data types (CRDTs). This provides support for computations where not all participants are online together at a given moment. The initial design presented here pro

itives for composing CRDTs, which lets us

tolerant distributed applications with nonmo

monotonic framework. Given reasonable mo

communications and node failures, we prove

program can be considered as a functional

functional reasoning and programming tech

plemented Lasp as an Erlang library built on

distributed systems framework. We have dev

large-scale application, the advertisement cou

SyncFree research project. We plan to extend

into a general-purpose language in which sy

as little as possible.

Categories and Subject Descriptors D.1.3

niques]: Concurrent Programming; E.1 [*D*

tributed data structures

Keywords Eventual Consistency, Commutative Operations, Erlang

the burden is placed on the programmer of these applications to ensure that concurrent operations performed on replicated data have both a deterministic and desirable outcome.

For example, consider the case where a user's gaming profile is replicated between two mobile devices. Concurrent operations, which can be thought of as operations performed during the period where both clients are onli

ommunication, can mod-

veloper

pdates.

current

profile

al. for

hat are

types,

), pro-

given

ributed

state.

tic res-

perty is

highly desirable in a distributed system because it no longer places the resolution logic in the hands of the programmer; programmers are able to use replicated data types that function as if they were their sequential counterparts. However, it has been shown that arbi-

Lasp, PPDP 2015
Functional Programming

PROGRAMMING MODEL: EXAMPLE

A :: Set

B :: Set

B = map λx.x A

Whenever A changes, B is recomputed from A.

Combinators can be user defined.

A :: OR-Set

B :: OR-Set

B = fold λx λy. x + y ⊥ A

Custom, non-trivial lattices.

Higher-order programming with functions.

CHALLENGES: STATE-OF-THE-ART

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

1. attices or place the onus on the developer

2. No formal specification of either systems execution model.

If you follow the rules, you have "**correct" programs.**

If you don't… ?

3. Incrementality of computations

Without monotone checks, program behavior is nondeterministic.
(ie. function reads clock, function is antitone, etc.)

functions where function application distributes

putation as a homomorphism and correctly

CHALLENGES: LATTICES

Bloom^L	Lasp
User-provided ADTs	Any CRDT implementing a “CRDT” interface is supported - ordering relation, merge, inflation, etc.
Combinators are user-provided	Combinators are built in for one CRDT: the Observed-Remove Set
Combinators must be labeled as either monotone or as a homomorphism	Higher-order combinator, fold, provided but requires user to ensure monotonicity

CHALLENGES: COMBINATORS

Bloom^L

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

Bloom^L

Unchecked annotations on homomorphic functions, and are incrementally evaluated

Lasp

(Some) built-in combinators are homomorphic, but are not evaluated as so

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 Bloom^L and Lasp: previously not formalized

DATA TYPES & COMBINATORS: BLOOM^L

Specified by the user

Must have:

- Values that form a lattice
- Least-upper-bound function
- Combinators

Homomorphism

Annotations are unchecked

```
1  class Bud::SetLattice < Bud::Lattice
2    wrapper_name :lset

4    def initialize(x=[])
5      # Input validation removed for brevity
6      @v = x.uniq # Remove duplicates from i
7    end

9    def merge(i)
10     self.class.new(@v | i.reveal)
11   end

13   morph :intersect do |i|
14     self.class.new(@v & i.reveal)
15   end

17   morph :contains? do |i|
18     Bud::BoolLattice.new(@v.member? i)
19   end

21   monotone :size do
22     Bud::MaxLattice.new(@v.size)
23   end
24 end
```

Least-upper-bound

Monotone Function

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

DATA TYPES & COMBINATORS: LASP

Implemented through a “CRDT” interface

Combinators are built into runtime:

- Functional: map, filter
- Set-theoretic: union, product, intersec
- Higher-order: fold

Homomorphism

Fold must be monotonic, unchec

Monotone Function

3.5 Functional Programming

We now look at the semantics for functional programming primitives that are lifted to operate over CRDTs: **map**, **filter**, and **fold**. We formalize them as follows:

Definition 3.3. The **map** function defines a process that never terminates, which reads elements of the input stream s and creates elements in the output stream t. For each element, the value v is separated from the metadata, the function f is applied to the value, and new metadata is attached to the resulting value $f(v)$. If two or more values map to the same $f(v)$ (for instance, if the function provided to map is surjective), the metadata is combined into one triple for all values of v.

$$
\begin{aligned}
F(s_i, f) &= \{f(v) \mid (v, \_, \_) \in s_i\} \\
A(s_i, f, w) &= \bigcup \{a \mid (v, a, \_) \in s_i \wedge w = f(v)\} \\
R(s_i, f, w) &= \bigcup \{r \mid (v, \_, r) \in s_i \wedge w = f(v)\} \\
map'(s_i, f) &= \{(w, A(s_i, f, w), R(s_i, f, w)) \mid w \in F(s_i, f)\} \\
map(s, f) &= t = [map'(s_i, f) \mid s_i \in s]
\end{aligned}
\tag{9}
$$

$$
\begin{aligned}
product'(s_i, u_j) = \{&((v, v'), a \times a', a \times r' \cup r \times a') \\
&\mid (v, a, r) \in s_i, (v', a', r') \in u_j\}
\end{aligned}
\tag{12}
$$

$$
product(s, u) = t = [product'(s_i, u_j) \mid s_i \in s, u_j \in u]
$$

RECONCILING THE "DELTA"

Challenges:

1. What is a useful generalization of the data types in Lasp and Bloom^L?
   - 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. Homomorphic functions from lattice A to lattice B
   - Monotone function where the combining function is join (least-upper-bound)

DELTA LATTICES & MULTICHAINS

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 → 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 + OBJECTS

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
ie. {1,2} → {1} ∪ {2} → 1 or 2

A practical instantiation of a CRDT-like lattice-based data type

COMPUTATIONAL DELTA OBJECTS: BLOOM^L

LOrd[T]:

Multichain:

`(1, T)`

Mutation functions:

`set(t, m) = (1, t)`

Query functions:

`value(m) = π₂ m`

Ordered register

LBool = LOrd[2]

2: ordered using the natural's ordering

LMax = LOrd[NMax]

Naturals ordered with bottom at zero or infinity

LMin = LOrd[NMin]

COMPUTATIONAL DELTA OBJECTS: CRDTS

G-Counter:

Multichain:

`(R, NMax)`

Mutation functions:

`increment(r, m) = (r, m(r) + 1)`

Query functions:

`value(m) = fold (π2 , +) a`

Grow-only counter is replica to count mappings

G-Set[A]:

Multichain:

`(A, 2)`

Mutation functions:

`add(a, m) = (a, 1)`

Query functions:

`value(m) = map π1 m`

Grow-only set is element to 2

COMPUTATIONAL DELTA OBJECTS: LASP

Observed-Remove Set[A]:

Multichain:

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

Mutation functions:

$add(a, t, m) = ((a, t, \bot), 1)$

$rem(a, t, m) = ((a, t, \top), 1)$

Query functions:

$value(m) = map\ \pi_1\ \{(a, t) \mid (a, t, \bot) \in m\} \setminus \{(a, t) \mid (a, t, \top) \in m\}$

IMPLEMENTING COMBINATORS: CHALLENGES

1. Combinators must be monotone, to ensure convergence
   - Alvaro et al., "Consistency Analysis in Bloom: a CALM and Collected Approach", CIDR 2011

2. Combinators must be from lattice to lattice, to ensure composition
   - Conway et al., "Logic and Lattices for Distributed Programming", SoCC 2012
   - Meiklejohn et al., "Lasp: A Language for Distributed, Coordination-Free Programming", PPDP 2015

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 opera

Important points:

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

Combine scalars for multiple occurrences in expression

Application joins and composes disjoint typing contexts and coeffects

$\Gamma @ R \vdash e : \tau$

(CONST) $\dfrac{}{\cdot @ \langle\rangle \vdash c : ty(c)}$

(VAR) $\dfrac{}{x : \tau @ \langle use \rangle \vdash x : \tau}$

(ABS) $\dfrac{\Gamma, x : \sigma @ \langle \ldots, s \rangle \vdash e : \tau}{\Gamma @ \langle \ldots \rangle \vdash \lambda x.e : \sigma \xrightarrow{s} \tau}$

(APP) $\dfrac{\Gamma_1 @ R \vdash e_1 : \sigma \xrightarrow{t} \tau \qquad \Gamma_2 @ S \vdash e_2 : \sigma}{\Gamma_1, \Gamma_2 @ R \times t \circledast S \vdash e_1\ e_2 : \tau}$

(CTXT) $\dfrac{\Gamma @ R \vdash e : \tau \qquad \Gamma' @ R' \rightsquigarrow \Gamma @ R, \theta}{\Gamma' @ R' \vdash \theta e : \tau}$

$\Gamma', R' \rightsquigarrow \Gamma, R, \theta$

(WEAK) $\ldots, ign \rangle \rightsquigarrow \Gamma @ \langle \ldots \rangle, \emptyset$

$\ldots \rangle \rightsquigarrow \Gamma_1, x : \tau, y : \sigma, \Gamma_2 @ \langle \ldots t, s \ldots \rangle, \emptyset$

$\Gamma_1, y : \tau, z : \tau, \Gamma_2 @ \langle \ldots s, t \ldots \rangle, [y, z \mapsto x]$

(SUB) $\Gamma_1, x : \tau, \Gamma_2 @ \langle \ldots s' \ldots \rangle \rightsquigarrow \Gamma_1, x : \tau, \Gamma_2 @ \langle \ldots s \ldots \rangle, \emptyset \quad (s \leq s')$

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

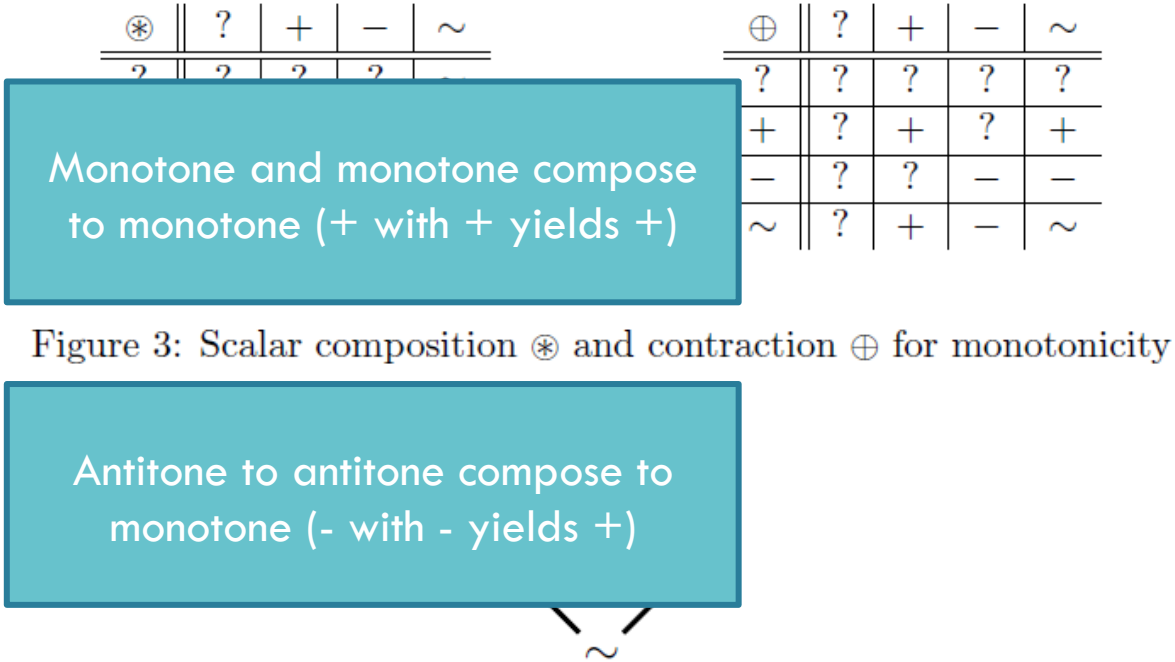


Figure 3: Scalar composition ⊛ and contraction ⊕ for monotonicity

Figure 4: Hasse diagram for the partial order ≤ on monotonicity scala

LIFTED MONOTONE FUNCTIONS

Given we want a monotone function from delta lattice A → delta lattice B…

We write:

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

Ca

- W as all combining functions may not be commutative
- C se

Apply the transformation and combine

Execution:

- $\mathtt{fold}\ (g \circ f)\ \bot_B\ \mathbf{deltas}(A)$

Decompose A into deltas

LIFTED HOMOMORPHIC FUNCTIONS

Given we want a homomorphic function from delta lattice A → delta lattice B…

We write:

- Mapping from A delta to B. `f : (A`$_{\delta}$ →

Execution:

- `fold (⊔ ∘ f)` $\bot_B$ **`deltas`**`(A)`

Deltas taken from A

Apply the transformation and combine using the join

MONOTONE FUNCTIONS: BLOOM^L

size on LSet

- size :: LSet → LMax
- f : λ((a, 1)). {(1, 1)}
- g : +

Produce a set of deltas for B

sum on LSet

- sum :: LSet → LMax
- f : λ((a, 1)). {(1, a)}
- g : +

Combining function is applied coordinate-wise

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

HOMOMORPHIC FUNCTIONS: BLOOM^L

+(y) on LMax
- + :: N → LMax → LMax
- f : λ((1, X)). {(1, x + y)}

Produce a set of deltas for B

-(y) on LMax
- - :: N → LMax → LMax
- f : λ((1, X)). {(1, x - y)}

map(g) on LSet
- map :: (A → B) → LSet → LSet
- f : λ((a, 1)). {(g(a), 1)}

filter(g) on LSet
- filter :: (A → Bool) → LSet → LSet
- f : λ((a, 1)). {}

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

OPERATIONAL SEMANTICS

$\mu; \pi; \rho; \phi \hookrightarrow \mu'; \pi'; \rho'; \phi'$

R-Mon-E

$$\frac{e \to e'}{\mu; \pi; \rho; \phi_1[l_1, l_2, v, D, e]\phi_2 \hookrightarrow \mu; \pi; \rho; \phi_1[l_1, l_2, v, D, e']\phi_2}$$

R-Mon-V-Next

$$\frac{}{\mu; \pi; \rho; \phi_1[l, v, d :: D, a]\phi_2 \hookrightarrow \mu; \pi; \rho; \phi_1[l, v, D, (v\ d)\ a]\phi_2}$$

R-Mon-V-Fin

$$\frac{getMonUpdates(l, \bot \boxplus a, \pi) = \phi' \qquad updateHom(l, \mu(l) \boxplus a, \rho) = \rho'}{\mu; \pi; \rho; \phi_1[l, v, \emptyset, a]\phi_2 \hookrightarrow \mu[l \mapsto a]; \pi; \rho'; \phi_1\phi_2\phi'}$$

R-Hom-E

$$\frac{e \to e'}{\mu; \pi; \rho_1[l_1, l_2, v, D, e]\rho_2; \phi \hookrightarrow \mu; \pi; \rho_1[l_1, l_2, v, D, e']\rho_2; \phi}$$

R-Hom-V-Next

$$\frac{\begin{array}{c} getMonUpdates(l_2, \bot \boxplus (\mu(l_2) \sqcup a), \pi) = \phi' \\ updateHom(l_2, \mu(l_2) \boxplus a, \rho_1[l_1, l_2, v, D, v\ d]\rho_2) = \rho' \end{array}}{\mu; \pi; \rho_1[l_1, l_2, v, d :: D, a]\rho_2; \phi \hookrightarrow \mu[l_2 \mapsto \mu(l_2) \sqcup a]; \pi; \rho'; \phi}$$

R-Hom-V-Fin

$$\frac{\begin{array}{c} getMonUpdates(l_2, \bot \boxplus (\mu(l_2) \sqcup a), \pi) = \phi' \\ updateHom(l_2, \mu(l_2) \boxplus a, \rho_1[l_1, l_2, v, \emptyset, \bot]\rho_2) = \rho' \qquad a \neq \bot \end{array}}{\mu; \pi; \rho_1[l_1, l_2, v, \emptyset, a]\rho_2; \phi \hookrightarrow \mu[l_2 \mapsto \mu(l_2) \sqcup a]; \pi; \rho'; \phi\phi'}$$

$a \in A$ (Lattice elements)
$d \in A_\delta$ (Deltas)
$m \in M$ (Delta mutators)

Monotone functions: decompose lattice into deltas and apply/combine

$\mu ::= \cdot \mid \mu[k \mapsto a]$ (Stores)
$\pi ::= \cdot \mid \pi[l_1, l_2, \lambda x.e]$ (Monotone link environment)
$\rho ::= \cdot \mid \rho[l_1, l_2, \lambda x.e, D, e]$ (Homomorphic link environmnet)
$\phi ::= \cdot \mid \phi[l, v, \langle d_1, \ldots, d_n \rangle, e]$ (Pending monotone link updates)

Homomorphisms: apply one delta at a time

$, v)$ is defined
$v \to \delta(c, v)$

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. Bloom^L 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"

$$\text{APP}\ \frac{\text{VAR}\ \dfrac{}{pre : B \xrightarrow{1} B@\langle 0\rangle \vdash pre : B \xrightarrow{1} B} \qquad \dfrac{}{x : B@\langle 0\rangle \vdash x : B}\ \text{VAR}}{pre : B \xrightarrow{1} B, x : B@\langle 0, 1\rangle \vdash pre\ x : B}$$

Typing context and coeffects are joined / composed.

OPERATIONAL SEMANTICS: SYNTAX/EXPRESSIONS

Homomorphic and monotone links between store locations

$a \in A$ (Lattice elements)
$d \in A_\delta$ (Deltas)
$m \in M$ (Delta mutators)
$q \in Q$ (Query functions)
$x \in X$ = Set of variables
$c \in C$ = Set of constants
$e ::= \lambda x.e \mid e_1\ e_2 \mid c \mid a \mid d \mid l \mid \mathbf{hlink}\ e_1\ e_2\ e_3 \mid \mathbf{mlink}\ e_1\ e_2\ e_3 \mid \mathbf{new} \mid \mathbf{update}\ e_1\ e_2$
$v ::= c \mid a \mid d \mid m \mid q \mid \lambda x.e \mid l \mid ()$

Interaction with the store

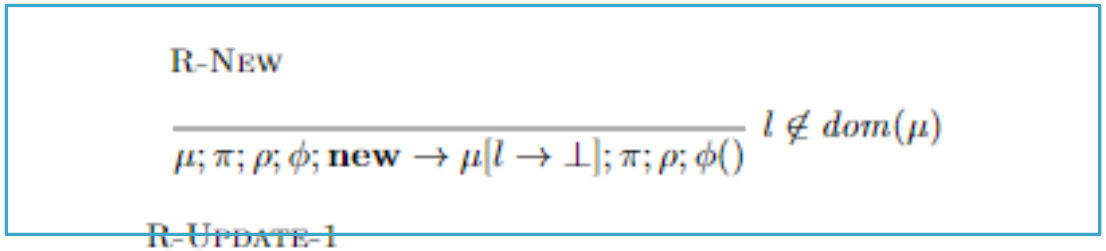
$l \in \mathcal{L}$ = Finite set of locations
$\mu ::= \cdot \mid \mu[k \mapsto a]$ (Stores)
$\pi ::= \cdot \mid \pi[l_1, l_2, \lambda x.e]$ (Monotone link environment)
$\rho ::= \cdot \mid \rho[l_1, l_2, \lambda x.e, D, e]$ (Homomorphic link environmnet)
$\phi ::= \cdot \mid \phi[l, v, \langle d_1, \ldots, d_n \rangle, e]$ (Pending monotone link updates)

$e \to e'$

$$\textsc{App-L}\ \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \textsc{App-R}\ \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'} \qquad \textsc{Beta}\ \frac{}{(\lambda x.e)\ v \to e[x := v]} \qquad \textsc{Delta}\ \frac{\delta(c, v) \text{ is defined}}{c\ v \to \delta(c, v)}$$

Standard expression reduction rules extended with constants and functional constants

OPERATIONAL SEMANTICS: STORES



$$\textsc{R-New}\quad \frac{}{\mu;\pi;\rho;\phi;\mathbf{new} \rightarrow \mu[l \rightarrow \bot];\pi;\rho;\phi()}\ l \notin dom(\mu)$$

Create locations in the store

$$\textsc{R-Update-1}\quad \frac{e_1 \rightarrow e_1'}{\quad \textbf{pdate}\ e_1'\ e_2}$$

Monotonic links forward location state decomposed into deltas

$$\textsc{R-Update-2}\quad \frac{\quad \textbf{pdate}\ l\ e_2'}{}$$

$$\textsc{R-Update-V}\quad \frac{a = \mu(l) \qquad getMonUpdates(l, \bot \boxtimes (a' \sqcup a)) = \phi' \qquad updateHom(l, a \boxtimes a', \rho) = \rho'}{\mu;\pi;\rho;\phi;\textbf{update}\ l\ a' \rightarrow \mu[l \mapsto a' \sqcup a];\pi;\rho';\phi'()}$$

Updates store, and propagates deltas forward using links

Homomorphic links propagate forward only the deltas representing the change

OPERATIONAL SEMANTICS: LINKS

$\boxed{\pi; \rho \to \pi'; \rho'}$

HLINK-1

$$\frac{e_1 \to e_1'}{\pi; \rho; \textbf{hlink}\ e_1\ e_2\ e_3 \to \pi; \rho; \textbf{hlink}\ e_1'\ e_2\ e_3}$$

HLINK-2

$$\frac{e_2 \to e_2'}{\pi; \rho; \textbf{hlink}\ l_1\ e_2\ e_3 \to \pi; \rho; \textbf{hlink}\ l_1\ e_2'\ e_3}$$

HLINK-3

$$\frac{e_3 \to e_3'}{\pi; \rho; \textbf{hlink}\ l_1\ l_2\ e_3 \to \pi; \rho; \textbf{hlink}\ l_1\ l_2\ e_3'}$$

HLINK

$$\frac{}{\pi; \rho; \textbf{hlink}\ l_1\ l_2\ \lambda x.e \to \pi; \rho[l_1, l_2, \lambda x.e, \emptyset, \bot]; ()}$$

Homomorphic links store pendin deltas and a monotone functio between store locations.

MLINK-1

$$\frac{e_1 \to e_1'}{\pi; \rho; \textbf{mlink}\ e_1\ e_2\ e_3 \to \pi; \rho; \textbf{mlink}\ e_1'\ e_2\ e_3}$$

MLINK-2

$$\frac{e_2 \to e_2'}{\pi; \rho; \textbf{mlink}\ l_1\ e_2\ e_3 \to \pi; \rho; \textbf{mlink}\ l_1\ e_2'\ e_3}$$

MLINK-3

$$\frac{e_3 \to e_3'}{\pi; \rho; \textbf{mlink}\ l_1\ l_2\ e_3 \to \pi; \rho; \textbf{mlink}\ l_1\ l_2\ e_3'}$$

MLINK

$$\frac{}{\pi; \rho; \textbf{mlink}\ l_1\ l_2\ \lambda x.e \to \pi[l_1, l_2, \lambda x.e]; \rho; ()}$$

Monotone function track a monoto function between store location

OPERATIONAL SEMANTICS: MLINKS

$$\mu; \pi; \rho; \phi \hookrightarrow \mu'; \pi'; \rho'; \phi'$$

R-Mon-E

$$\frac{e \to e'}{\mu; \pi; \rho; \phi_1[l_1, l_2, v, D, e]\phi_2 \hookrightarrow \mu; \pi; \rho; \phi_1[l_1, l_2, v, D, e']\phi_2}$$

R-Mon-V-Next

$$\frac{}{\mu; \pi; \rho; \phi_1[l, v, d :: D, a]\phi_2 \hookrightarrow \mu; \pi; \rho; \phi_1[l, v, D, (v\ d)\ a]\phi_2}$$

R-Mon-V-Fin

$$\frac{getMonUpdates(l, \bot \boxplus a, \pi) = \phi' \qquad updateHom(l, \mu(l) \boxplus a, \rho) = \rho'}{\mu; \pi; \rho; \phi_1[l, v, \emptyset, a]\phi_2 \hookrightarrow \mu[l \mapsto a]; \pi; \rho'; \phi_1\phi_2\phi'}$$

R-Hom-E

$$\frac{e \to e'}{\mu; \pi; \rho_1[l_1, l_2, v, D, e]\rho_2; \phi \hookrightarrow \mu; \pi; \rho_1[l_1, l_2, v, D, e']\rho_2; \phi}$$

R-Hom-V-Next

$$\frac{getMonUpdates(l_2, \bot \boxplus (\mu(l_2) \sqcup a), \pi) = \phi' \quad updateHom(l_2, \mu(l_2) \boxplus a, \rho_1[l_1, l_2, v, D, v\ d]\rho_2) = \rho'}{\mu; \pi; \rho_1[l_1, l_2, v, d :: D, a]\rho_2; \phi \hookrightarrow \mu[l_2 \mapsto \mu(l_2) \sqcup a]; \pi; \rho'; \phi}$$

R-Hom-V-Fin

$$\frac{getMonUpdates(l_2, \bot \boxplus (\mu(l_2) \sqcup a), \pi) = \phi' \quad updateHom(l_2, \mu(l_2) \boxplus a, \rho_1[l_1, l_2, v, \emptyset, \bot]\rho_2) = \rho' \quad a \neq \bot}{\mu; \pi; \rho_1[l_1, l_2, v, \emptyset, a]\rho_2; \phi \hookrightarrow \mu[l_2 \mapsto \mu(l_2) \sqcup a]; \pi; \rho'; \phi\phi'}$$

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

OPERATIONAL SEMANTICS

$\mu; \pi; \rho; \phi \hookrightarrow \mu'; \pi'; \rho'; \phi'$

R-Mon-E

$$\frac{e \to e'}{\mu; \pi; \rho; \phi_1[l_1, l_2, v, D, e]\phi_2 \hookrightarrow \mu; \pi; \rho; \phi_1[l_1, l_2, v, D, e']\phi_2}$$

R-Mon-V-Next

$$\frac{}{\mu; \pi; \rho; \phi_1[l, v, d :: D, a]\phi_2 \hookrightarrow \mu; \pi; \rho; \phi_1[l, v, D, (v\ d)\ a]\phi_2}$$

R-Mon-V-Fin

$$\frac{getMonUpdates(l, \bot \boxplus a, \pi) = \phi' \qquad updateHom(l, \mu(l) \boxplus a, \rho) = \rho'}{\mu; \pi; \rho; \phi_1[l, v, \emptyset, a]\phi_2 \hookrightarrow \mu[l \mapsto a]; \pi; \rho'; \phi_1\phi_2\phi'}$$

R-Hom-E

$$\frac{e \to e'}{\mu; \pi; \rho_1[l_1, l_2, v, D, e]\rho_2; \phi \hookrightarrow \mu; \pi; \rho_1[l_1, l_2, v, D, e']\rho_2; \phi}$$

R-Hom-V-Next

$$\frac{\begin{array}{c} getMonUpdates(l_2, \bot \boxplus (\mu(l_2) \sqcup a), \pi) = \phi' \\ updateHom(l_2, \mu(l_2) \boxplus a, \rho_1[l_1, l_2, v, D, v\ d]\rho_2) = \rho' \end{array}}{\mu; \pi; \rho_1[l_1, l_2, v, d :: D, a]\rho_2; \phi \hookrightarrow \mu[l_2 \mapsto \mu(l_2) \sqcup a]; \pi; \rho'; \phi}$$

R-Hom-V-Fin

$$\frac{\begin{array}{c} getMonUpdates(l_2, \bot \boxplus (\mu(l_2) \sqcup a), \pi) = \phi' \\ updateHom(l_2, \mu(l_2) \boxplus a, \rho_1[l_1, l_2, v, \emptyset, \bot]\rho_2) = \rho' \qquad a \neq \bot \end{array}}{\mu; \pi; \rho_1[l_1, l_2, v, \emptyset, a]\rho_2; \phi \hookrightarrow \mu[l_2 \mapsto \mu(l_2) \sqcup a]; \pi; \rho'; \phi\phi'}$$

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