Bar$_{\mathcal{QL}}$: Collaborating Through Change

Oliver Kennedy and Lukasz Ziarek
SUNY Buffalo
{okennedy, lziarek}@buffalo.edu

Abstract

Applications such as Google Docs, Office 365, and Dropbox show a growing trend towards incorporating multi-user *collaboration* functionality into web applications. These collaborative applications share a need to efficiently express shared state, typically through a shared log abstraction. Extensive research efforts on log abstractions by the database, programming languages, and distributed systems communities have identified a variety of analysis techniques based on the algebraic properties of updates (*i.e.*, pairwise commutativity, subsumption, and idempotence). Although these techniques have been applied to specific application domains, to the best of our knowledge, no attempt has been made to create a general framework for such analyses in the context of a non-trivial update language. In this paper, we introduce *monadic logs*, a semantically rich state abstraction that provides a powerful, expressive framework for reasoning about a variety of application state properties. We also define Bar$_{\mathcal{QL}}$, a general purpose state-update language, and show how the monadic log abstraction allows us to reason about the properties of updates expressed in Bar$_{\mathcal{QL}}$. Finally, we show how such analyses can be expressed declaratively using the SPARQL graph query language.

1. Introduction

Many web applications have been released that improve on the functionality of desktop applications (*e.g.* Google Docs). Collaborative functionality is a natural consequence of this shift from the desktop to the web. Fully featured word processors, presentation editors, spreadsheets, and drawing programs now exist that allow users to simultaneously edit, view, and annotate documents in "real-time." Although these collaborative applications are structured using a client/server model, the core functionality of the application is typically built into the client. The server's primary role is to relay state updates between clients. In spite of this apparent structural simplicity, developers continue to expend considerable effort to achieve infrastructure scalability through specialization.

To address this concern, we present the theoretical foundations for a generalized, yet performant server infrastructure for collaborative applications: Laasie[1]. Laasie clients are application front-ends, each maintaining and mediating access to a local replica of the global application state. Clients post state updates to the Laasie infrastructure, which defines a canonical order over the updates that it receives, and ensures delivery of updates to all participating clients.

A primary goal for this infrastructure is to give application developers a way to easily formulate and assert properties over both global and client state. For instance, our infrastructure can assert that clients provably recover from link failures (*e.g.*, when the host platform changes networks or after it wakes from sleep mode). Recoverability is expressed as a "replay," or re-execution of the updates stored within the log, and parameterized by the state from which recovery is initiated.

To achieve generality in reasoning about such properties, our formal model expresses updates in terms of *intent* rather than *effect*. The resulting log encodes application state updates in a functional form, storing the *computation* itself rather than its effects. We call such logs *monadic*, as they precisely encode a sequence of global state transformation monads. Properties are inferred via program analysis of the computation encoded within the log. This structure allows properties like recoverability to be stated declaratively, and asserted over the log through automated reasoning techniques.

One use of such properties is to provide a general equivalence class of monadic logs over which that property holds. We observe that meaning-preserving equivalence classes of *transforms* of monadic logs define the possible structural representations of a computation's state. An implementation would, therefore, be free to pick the representation that most adequately satisfies its performance, security, and/or behavioral requirements. We introduce a general framework for reasoning about transforms over monadic logs and consider two properties for each transform: (1) *Tail-Correctness*, or whether the transformed log generates a state identical to the original sequence of updates, and (2) *Recoverability*, or whether the transformed log can synthesize the most recent state from a given previous state. In addition to reasoning about these properties in the abstract, we show a concrete expression of both properties via SPARQL queries issued over a graphical representation of the log and its contents.

To drive our discussion of monadic logs, we introduce an example monad language: Bar$_{\mathcal{QL}}$, an update language that can express conditionals and iteration over complex hierarchical datatypes. Updates expressed in Bar$_{\mathcal{QL}}$ are not evaluated, but rather appended to a monadic log. The consequence of this is a simplified semantics for out-of-order appends and encoding of updates as increments (*i.e.*, deltas) rather than fixed writes (*e.g.*, $var := 3$). In short, Bar$_{\mathcal{QL}}$ allows the operational semantics of updates to be managed functionally as first class monads.

By showing that we can assert a range of properties over monadic logs built on a non-trivial language like Bar$_{\mathcal{QL}}$, we show the potential for monadic logs in real world systems.

[1] Log-As-Application-State InfrastructurE

The contributions of this paper are as follows:

1. The formal definition of monadic logs, a novel monad-based representation of application state, along with provable safety properties for log transforms.
2. An exploration of monadic logs, where transforms necessary for log compaction are identified, necessary conditions for their tail-correctness and recoverability are proven, and program analysis is used to efficiently identify viable log transforms.
3. An extension of preliminary work on Laasie [6], with a precise specification of $\text{Bar}_{\mathcal{QL}}$: a language for transforming weakly typed complex data, and a reduction to a fine-grained incremental representation of $\text{Bar}_{\mathcal{QL}}$.
4. The use of graph queries, formulated in SPARQL, to assert global properties about the log with respect to local properties of each log entry, obtained through program analysis. Thus, log properties are expressed declaritvely and asserted structurally.

1.1 Roadmap

We begin in Section 2 with a formal definition of monadic logs, and define the fundamental semantics of monadic log transforms. We demonstrate the benefits of the monadic log representation by example in Section 3, through a nontrivial state management task: log compaction with recoverability. We define two specific log transforms for compaction, two safety properties for log transforms: tail-correctness and recoverability, and two abstract relations over log entries: pairwise commutativity and subsumption. We show how these two relations can be used to assert the safety properties of the two log compaction transforms.

To make these properties concrete, we introduce a simple, but nontrivial monad language named $\text{Bar}_{\mathcal{QL}}$ in Section 4. This language, loosely based on the Monad Algebra, is amenable to the sorts of program analysis required to assert the safety properties presented in Section 3. In Section 5, we reduce $\text{Bar}_{\mathcal{QL}}$ to a simplified incremental form that provides a fine-grained view of the effects of each monad, and show how this fine-grained view can be used for program analysis.

Finally, in Section 6, we return to the log compaction problem and show how the safety properties for recoverability can be asserted over logs of $\text{Bar}_{\mathcal{QL}}$ monads through properties obtained by program analysis. We then show graph based definitions of commutativity and subsumption that can be derived structurally from the log. Lastly we show the assertion of safety of log compaction transforms via simple queries over these graph based representations.

2. Monadic Logs

A popular strategy for state replication and persistence [10, 13, 32, 36] is to log a history of operations applied to the application's state. Logs present a convenient abstraction for state replication, as the primary challenge of coordination is reduced to establishing a canonical order over state updates. Each client can update its local state replica by replaying operations as they are appended to the log. Logs are also extremely appealing for the purpose of analytics. A log illustrates the full evolution of an application's state as it is used. Developers can examine how users interact with their applications, users can revisit past versions of the state, and temporal relationships can be used to define access policies.

To maximize the analytical potential of a log, care must be taken to record not just the effects of an update, but also any metadata relevant to the analysis. For example, building a policy based on provenance (*e.g.*, to implement a taint-tracking system) requires logging policy-specific provenance metadata in the log. A more complex analysis may require additional metadata, and some analyses may not be possible if they require metadata that has not already been collected. Notably, current systems *specialize* the log and metadata structure to the application domain supported by the system. Expanding systems to support additional analyses or application domains is often difficult if not outright impossible.

Instead, we propose a generalized logging strategy that provides an expressive framework for analysis of an application's execution, while being agnostic to the application domain. To achieve generality, our proposed framework logs each update as a function, rather than as the effects of applying that function to the state. That is, each update is recorded as a state transformation monad. We refer to the resulting structure as a monadic log. Monadic logs also give developers an additional degree of flexibility in manipulating application state. Consider the following potential applications:

Compaction with Provable Recovery. Although an append-only log is a useful high-level abstraction, in practice it is necessary to compact the log to bound its size. This is typically done via a snapshot of the application state that is substituted for all log entries that precede it. Unfortunately, eliminating all log entries preceding the snapshot also invalidates all clients who's local views correspond to a state prior to the snapshot. Such clients must be restarted from scratch, negating the benefits of a log. Conversely, monadic logs explicitly contain all information necessary to *merge* log entries. For example, an overwriting update to an object that has not been read since its last update can be merged with its last update, such that only the most recent update is kept.

Log Views. An application might have a secondary client: a toolbar widget that displays aggregate values (*e.g.*, an unread message count). This application requires only a simplified aggregate view of the application state. A log encoding application state can be transformed into a log manipulating the view by a delta transform.

Retroactive Policy Enforcement. Write access policies can be enforced by deleting log entries that violate the access policy. Such policies can be applied *retroactively* with minimal performance overhead, because each log entry encodes the full semantics of its update. The effects of deleting or modifying an log entry are immediately propagated through all subsequent log entries.

Obfuscated Collaboration. A policy that obfuscates private data (*e.g.*, by adding a random factor) for unauthorized users does not prevent authorized and unauthorized users from collaborating. Although each user sees a distinct view of an application's state, updates applied to the anonymized view can be *seamlessly applied to the original data*, making them visible to authorized users as well.

Each of these examples can be implemented by applying a log-based transform to the application's state. Such transforms can be applied directly by Laasie, and do not require fundamental changes to the application's code. Prior to introducing log transforms, we first describe the log itself.

2.1 Formalism

We begin by formally defining a monadic log as a sequence of state transformation functions, or monads.

Definition 1. *Let the type τ represent an application's state, and $\mathcal{M}$ be a family of monads over τ, $\mathcal{M} \subseteq 2^{\tau \to \tau}$. A* **monadic log** *$\ell(\mathcal{M}, \tau)$ is defined by a sequence of monads $m_i \in \mathcal{M}$. A monadic log may be represented in three ways:*

1. As the raw sequence of monads $[m_1, \ldots, m_n]$.

2. *As the composition of these monads (by convention from left to right)* $m_1 \circ \ldots \circ m_n \equiv \lambda x.m_n(\ldots(m_1(x)))$.
3. *As the state obtained by evaluating the composition on a default initial value* v_0*:* $m_n(\ldots(m_1(v_0)))$.

Because these representations have clearly distinct types and can be easily generated from the initial list representation, for prospecuity we use them interchangeably. The appropriate representation will be clear from the context in which it is used. The size of a monadic log ($|\ell|$) is the arity of the monad sequence.

This relatively straightforward state representation can provide a wealth of information: provenance, intermediate states, and privacy-loss (e.g., as in differential privacy) can all be computed from the log through program analysis. Furthermore, because the full application semantics are embedded in the log, such analyses can be performed without needing to pre-provision analysis-specific metadata. Although in this paper we deal with the theoretical underpinnings of monadic logs, a preliminary exploration of their feasibility and performance characteristics has shown that they are a viable replacement for standard log-based state replication mechanisms in real-world systems [6].

A monadic log provides an added degree of freedom for application developers to mutate state. Instead of modifying the state directly (i.e., by applying a new update to the tail of the log), state changes can be expressed as direct transforms of the log itself.

Definition 2. *A* **log transform***:* $T : \ell(\mathcal{M}, \tau) \rightarrow \ell(\mathcal{M}, \tau)$ *maps one monadic log* $[m_1, \ldots, m_n]$ *into another* $[m'_1, \ldots, m'_{n'}]$

For simplicity, we will consider only size-preserving transforms (i.e., where $|\ell| = |T(\ell)|$). If we assume closure of the monad language $\mathcal{M}$ over composition and the presence of an identity operation, this can be done without loss of generality. Using a size-preserving transform allows us to establish a correspondence between state versions, both before and after the transform.

Definition 3. *Let the* **timestamp** i *of a monad* $m_i \in \ell$ *be defined by the order of monads in the log* ℓ*.*

Definition 4. *An* **intermediate state** ℓ_k *is the log consisting of the first* k *monads in* ℓ*:* $\ell_k = [m_1, \ldots, m_k]$

3. Provable Recovery

Throughout the rest of the paper, we will use log compaction as a simple, high-level example to illustrate the power of log transforms. Our theory of log transforms allow us to easily express the notion of compaction in a more general form than typical checkpoint-based. We start with the building blocks: two primitive log transforms, *delete* and *compose*, and properties that define when it is safe to apply these primitive transforms.

Delete. A size-preserving deletion transform is effected by replacing the deleted monad with the identity monad **id**. The transform $\mathcal{R}_{\texttt{del}}(x)$, which deletes monad m_x is defined as

$$m'_i = \begin{cases} m_i & \ldots \quad i \neq x \\ \mathbf{id} & \ldots \quad i = x \end{cases}$$

Compose. For a monad language closed over composition, monads may be merged into a single log entry. The log size is preserved by inserting an **id** monad in place of one of the merged monads. By convention, the composed monad replaces the monad with the higher timestamp. The transform $\mathcal{R}_{\texttt{cmp}}(x, y)$, which merges monads m_x and m_y is defined as

$$m'_i = \begin{cases} m_i & \ldots \quad i \notin \{x, y\} \\ \mathbf{id} & \ldots \quad i = x \\ m_x \circ m_y & \ldots \quad i = y \end{cases}$$

These primitives can be used to implement a variety of different log compaction strategies. As a trivial example, a checkpoint at timestamp k can be implemented by applying the transform $\mathcal{R}_{\texttt{cmp}}(i, i+1)$ for all $i < k$.

3.1 Safety

Although compaction may change the log itself, it is crucial that it not change the application state encoded in the log. We now propose three properties of log transforms that can be used to assert this safety condition: tail-correctness, recoverability, and $\vec{t}$-recoverability. The simplest of these properties, tail-correctness takes a holistic view of the log.

Definition 5. *Given a default initial value* v_0*, a log transform* T *is tail-correct if, when applied to an arbitrary log* ℓ *of size* n*, the resulting log* $\ell' = T(\ell)$ *evaluates to the same value* $\ell_n(v_0) = \ell'_n(v_0)$*.*

In a practical setting, tail-correctness is insufficient. Consider a state replica currently at state ℓ_i. To bring this client up to state ℓ_n, it should be sufficient to send log entries $m_{i+1}, \ldots, m_n$. However, even if we restrict ourselves to tail-correct log transforms, an error may still occur. Consider a simple monad language over integers with a single operation: increment by A ($[+ = A]$), and a log containing two entries.

$$\ell = m_1 : [+ = 1], m_2 : [+ = 1]$$

The transform $T_{\texttt{cmp}}(1, 2)$ produces the log

$$\ell' = m'_1 : \mathbf{id}, m'_2 : [+ = 2]$$

This transform is tail correct ($\ell_2 = \ell'_2$). However, a client at state 1 ($\ell_1 = 1$) before the transform using ℓ' to obtain the 'current' state (i.e., applying m'_2 to its local state) will obtain the incorrect state ($m'_2(\ell_1) = 3$). To address this limitation, we define a stronger correctness property: recoverability.

Definition 6. *Given a default initial value* v_0*, a log transform is* **recoverable from timestamp** i *(or equivalently state* ℓ_i*) if the final state* ℓ_n *of the original log can be obtained by applying the sequence of rewritten monads following timestamp* i *to the state* ℓ_i*, taken from the original log.*

$$(m_1 \circ \ldots \circ m_n)(v_0) = (m'_{i+1} \circ \ldots \circ m'_n)(v_i)$$

Or equivalently (because v_i *is defined by the original log)*

$$(m_1 \circ \ldots \circ m_n)(v_0) = (m_1 \circ \ldots \circ m_i \circ m'_{i+1} \circ \ldots \circ m'_n)(v_0)$$

Definition 7. *A log transform is* **recoverable** *if it is recoverable from all timestamps in the log (*i.e.*,* $i \in [0, n]$*))*

Note that tail-correctness is the special case of recoverability from timestamp 0. The intent of recoverability is to protect disconnected clients from reaching an inconsistent state when replaying log entries on reconnection. However, to guarantee full recoverability, we must disregard many useful log transforms. In a practical setting, a server will not need to guarantee recoverability for all timestamps, and the notion of recoverability can be relaxed.

Definition 8. *Given a set of timestamps* $\vec{t}$*, a log transform is* $\vec{t}$*-recoverable if it is recoverable from every* $t \in \vec{t}$*.*

By tracking when clients disconnect (regardless of whether or not the disconnection is transient), the server can identify ranges of log entries over which non-recoverable log transforms can still be performed safely.

3.2 Language Properties

We now show how these correctness conditions can be asserted through properties of the monad language itself. When the latter properties can be obtained through program analysis, the two primitive transforms can be applied aggressively to the full extent permitted by tail-correctness, recoverability, or $\vec{t}$-recoverability. Concretely, we start with two properties of monads themselves: subsumption and pairwise commutativity.

Definition 9. *Two monads of the same kind m_1 and m_2 are* **pairwise commutative** *if their compositions are equivalent, regardless of the order in which they are composed.*

$$\mathcal{C}(m_1, m_2) \iff \forall x : m_2(m_1(x)) \equiv m_1(m_2(x))$$

Definition 10. *A monad m_2* **subsumes** *a monad of the same kind m_1 if the effects of m_1 are completely screened by m_2.*

$$\mathcal{S}(m_1, m_2) \iff \forall x : m_2(m_1(x)) \equiv m_2(x)$$

Pairwise commutativity and subsumption are relatively straightforward properties, amenable to being asserted by program analysis. A language specifically designed for this purpose is discussed below, in Section 4. We start by showing how to evaluate tail-correctness for deletion and composition transforms.

Lemma 1. *The transform $\mathcal{R}_{\texttt{del}}(x)$ is tail-correct if m_x is subsumed by the aggregate composition of all monads following it: $\mathcal{S}(m_x, (m_{x+1} \circ m_{x+2} \circ \ldots \circ m_n))$.*

Proof. The identity operation has no effect on the state, and can be inserted anywhere. By subsumption, we have that

$$m_x \circ \ldots \circ m_n \equiv m_{x+1} \circ \ldots \circ m_n$$

Thus, $\ell_n = \ell'_n$. □

Lemma 2. *The transform $\mathcal{R}_{\texttt{cmp}}(x, y)$ is tail-correct for any monad language closed over composition if m_x commutes with the aggregate composition of all monads between it and m_y: $\mathcal{C}(m_x, (m_{x+1} \circ \ldots \circ m_{y-1}))$.*

Proof. As before, the identity operation has no effect on state. If $x = y - 1$, then the merged monads are equivalent to the separate monads by the associativity of composition. Otherwise, by commutativity, we have that

$$m_x \circ \ldots \circ m_{y-1} \equiv m_{x+1} \circ \ldots \circ m_{y-1} \circ m_x$$

Once m_x and m_y are adjacent, they can be merged as before. □

We next show how to infer recoverability.

Lemma 3. *If the log transform $\mathcal{R}_{\texttt{del}}(x)$ is tail-correct, it is recoverable.*

Proof. Recoverability from any state v_i s.t. $i < x$ is equivalent to tail-correctness, because these states are unaffected by the transform. Recoverability when $i \geq x$ is guaranteed always: The state v_i being recovered from is taken before the transform, and monads $m'_{x+1}, \ldots, m'_n$ are identical to their pre-transform counterparts. □

This proof shows a tight coupling between correctness and recoverability, and illustrates an intriguing log partitioning. If a transform only modifies monads that fall within a fixed range, recoverability "errors" can only occur at states that fall within that same range.

Proposition 1. *Let $\mathcal{R}$ be a tail-correct log transform, which only alters log entries at timestamps in the range $[x, y]$. Monads outside of this range are unaffected by $\mathcal{R}$.*

$\mathcal{R}$ is recoverable iff it is recoverable from all states $v_i \in [x, y)$

Proof. The proof is identical to that of Lemma 3. □

Lemma 4. *The transform $\mathcal{R}_{\texttt{cmp}}(x, y)$ is recoverable if it is correct, and if m_x is idempotent: $\mathcal{S}(m_x, m_x)$.*

Proof. From the commutativity property required to show correctness, we have that $m_x \circ \ldots \circ m_{y-1} \equiv m_{x+1} \circ \ldots \circ m_{y-1} \circ m_x$. For all $i \geq x$, state $v_i = (m_x \circ \ldots \circ m_i)(v_{x-1})$. Thus, $(m'_{i+1} \circ m'_y)(v_i) \equiv (m_x \circ \ldots \circ m_x \circ m_y)(v_x)$. By commutativity, we can transform this expression as $m_{x+1} \ldots \circ m_x \circ m_x \circ m_y$. By idempotence, this is equivalent to the original rewritten expression, and by Proposition 1 the proof devolves to that of correctness. □

4. Bar$_{\mathcal{QL}}$

In this section we introduce Bar$_{\mathcal{QL}}$, a log-based update language loosely based on the Monad Algebra with unions and aggregates [28], for generating monadic logs. Bar$_{\mathcal{QL}}$ is intended to be simple, expressive, and amenable to program analysis required for log analytics. After laying out Bar$_{\mathcal{QL}}$, we will return to our example, and show how program analysis can be used to assert pairwise commutativity and subsumption in Bar$_{\mathcal{QL}}$.

Unlike the Monad Algebra, which uses sets as the base collection type, Bar$_{\mathcal{QL}}$ uses maps[2] and has weaker type semantics. Furthermore, Bar$_{\mathcal{QL}}$ is intentionally limited to operations with linear computational complexity in the size of the input data; neither the pair-with nor cross-product operations are included. In our domain, this is not a limitation, as the server is acting primarily as a relay for state. Cross-products can be transmitted to clients more efficiently in their factorized form, and clients are expected to be capable of computing cross products locally.

The domains and grammar for Bar$_{\mathcal{QL}}$ are given in Fig. 1. We use c to range over constants, p over primitives (strings, integers, floats, and booleans), k over keys, M over monads (queries), τ over types, v over values of type τ, and θ over binary operations over primitive types. The type τ operated over by Bar$_{\mathcal{QL}}$ monads is equivalent to unstructured XML or JSON. Values are either of primitive type, null, or collections (mappings from k to τ). Note that collections are total mappings; for instances, a singleton can be defined as the collection where all keys except one map to the *null* value. By convention, when referring to collections we will implicitly assume the presence of this mapping for all keys that are not explicitly specified in the rules themselves.

We formalize Bar$_{\mathcal{QL}}$ in Fig. 2 in terms of a big-step operational semantics. Order of evaluation is defined by the structure of the rules. In Bar$_{\mathcal{QL}}$ queries are monads, structures that represent computation. Reducing the query corresponds to evaluating the computation expressed by that query. The rules for *PrimitiveConstant*, *Null* and *EmptySet* all define operations that take an input value and produce a constant value regardless of input. The rule *PrimitiveConstant* produces a primitive constant c, the rule *Null* produces the *null* value, and the rule *EmptySet* produces a empty set. We define an empty set as a collection that is a total mapping, where all keys map to the *null* value, represented as: $\{* \rightarrow null\}$. The

[2] Maps are also referred to as hashes, dictionaries, or lookup tables.

$$
\begin{array}{lcllcl}
c & \in & \mathit{Constant} :\rightarrow p & k & \in & \mathit{Key} \\
p & \in & \mathit{Primitive} & M & \in & \mathit{Monad} : \tau \rightarrow \tau \\
v & \in & \mathit{Value} & \tau & \in & \mathit{Type} : p \mid \{k_i \rightarrow \tau_i\} \mid \mathit{null} \\
\theta & \in & \mathit{BinaryOp} & & & \\
\end{array}
$$

$$
\begin{array}{rcl}
M & := & M.k \mid M \Leftarrow M \mid \mathbf{map}\ M\ \mathbf{using}\ M \mid M\ \mathbf{op}_{[\theta]}\ M \\
 & \mid & \mathbf{agg}_{[\theta]}(M) \mid \mathbf{agg}_{[\Leftarrow]}(M) \mid \mathbf{filter}\ M\ \mathbf{using}\ M \\
 & \mid & \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M \mid M \circ M \mid \mathbf{c} \mid \mathbf{null} \mid \emptyset
\end{array}
$$

Figure 1: Domains and grammar for $\text{Bar}_{\mathcal{QL}}$.

.

Identity operation passes through the input value unchanged. *Subscripting* and *Singleton* are standard operations. In comparison to Monad Algebra, these operations correspond to not only the singleton operation over sets, but also the tuple constructor and projection operations. Because collection elements are identified by keys, we can reference specific elements of the collection in much the same way as selection from a tuple.

The most significant way in which $\text{Bar}_{\mathcal{QL}}$ differs from Monad Algebra is its use of the *Merge* operation ($\Leftarrow$) instead of set union ($\cup$). $\Leftarrow$ combines two sets, replacing undefined entries in one collection (keys for which a collection maps to *null*) with their values from the other collection:

$$(\{A := 1\} \Leftarrow \{B := 2\})(\mathit{null}) = \{A \rightarrow 1, B \rightarrow 2\}$$

If a key is defined in both collections, the right input takes precedence:

$$(\{A := 1\} \Leftarrow \{A := 2\})(\mathit{null}) = \{A \rightarrow 2\}$$

The merge operator can be combined with singleton and identity to define updates to collections:

$$(\mathbf{id} \Leftarrow \{A := 3\})(\{A \rightarrow 1, B \rightarrow 2\}) = \{A \rightarrow 3, B \rightarrow 2\}$$

Subscripting can be combined with merge, singleton, and identity to define point modifications to collections:

$$
\begin{aligned}
(\mathbf{id} \Leftarrow \{A := (\mathbf{id}.A \Leftarrow \{B := 2\})\})(\{A \rightarrow \{C \rightarrow 1\}\}) \\
= \{A \rightarrow \{B \rightarrow 2, C \rightarrow 1\}\} \quad (1)
\end{aligned}
$$

Primitive binary operators are defined monadically with operation *PrimBinOp*, and include basic arithmetic, comparisons, and boolean operations. These operations can be combined with identity, singleton, and merge to define updates. For example, to increment A by 1, we write:

$$\{\mathbf{id} \Leftarrow \{A := \mathbf{id}.A + 1\}\}(\{A \rightarrow 2\}) = \{A \rightarrow 3\}$$

$\text{Bar}_{\mathcal{QL}}$ provides constructs for mapping, flattening and aggregation. The *Map* operation is analogous to its definition in Monad Algebra, save that key names are preserved. The *Flatten* operation is also similar, except that is based on $\Leftarrow$, rather than $\cup$ as in Monad Algebra. The *PrimitiveAggregation* class of operators defines aggregation using any closed binary operator θ operating over over primitive type. To increment all children of the root by 1 we write:

$$(\mathbf{map\ id\ using}\ (\mathbf{id} + 1))(\{A \rightarrow 1, B \rightarrow 2\}) = \{A \rightarrow 2, B \rightarrow 3\}$$

To increment the child C of each child of the root by 1, we write:

$$
\begin{aligned}
(\mathbf{map\ id\ using}\ (\mathbf{id} \leftarrow \{C := \mathbf{id}.C + 1\}))(\\
\{A \rightarrow \{C \rightarrow 1\}, B \rightarrow \{C \rightarrow 2, D \rightarrow 1\}\} \\
) = \{A \rightarrow \{C \rightarrow 2\}, B \rightarrow \{C \rightarrow 3, D \rightarrow 1\}\} \quad (2)
\end{aligned}
$$

Finally, $\text{Bar}_{\mathcal{QL}}$ supports *Conditionals* and *Filtering*, as well as *Composition* of queries. The rules for these reductions are standard.

4.1 Read-Normal Form

We now present a normalized form of $\text{Bar}_{\mathcal{QL}}$, in which the read pattern of a $\text{Bar}_{\mathcal{QL}}$ monad (*i.e.*, which collection elements it accesses) are made more explicit.

Definition 11. *A $\text{Bar}_{\mathcal{QL}}$ monad of the form $\mathbf{id}.k_1.k_2.(\ldots).k_n$ is a* **point read** *at path $\phi = k_1.k_2.(\ldots).k_n$. A $\text{Bar}_{\mathcal{QL}}$ monad is in* **read-normal form** *if subscript operators appear exclusively as part of point reads, or the subscripting key is the temporary key 'tmp'.*

A valid $\text{Bar}_{\mathcal{QL}}$ query can be converted into read-normal form by evaluating (i.e., pushing down) every subscript operation. This evaluation process is presented in Algorithm 1. To obtain a $\text{Bar}_{\mathcal{QL}}$ expression in read-normal form, the algorithm is applied repeatedly to reify subscript operators that violate read-normal form. This fixed-point computation is shown in Algorithm 2.

Algorithm 1 Subscript

Require: m, a $\text{Bar}_{\mathcal{QL}}$ monad.
Require: ϕ, a path to subscript. This path may contain wildcard elements $*$, which match all keys.
Ensure: m_ϕ, a $\text{Bar}_{\mathcal{QL}}$ monad such that $(m.\phi) \equiv m_\phi$. For every wildcard element in the path, one level of collection nesting is created for each key matched by the wildcard.

```
if k = [ ] then
    let m_φ ← m
else if m matches (m' ⇐ m'') then
    let m'_φ, m''_φ ← Subscript(m', φ), Subscript(m'', φ)
    let m_φ ← (if m''_φ ≠ null then m''_φ else m'_φ)
else if m matches {k := m'} then
    if φ matches k.φ' then
        let m_φ ← (Subscript(m', φ'))
    else if φ matches *.φ' then
        let m_φ ← ({k := Subscript(m', φ')})
    else
        let m_φ ← (null)
else if m matches ∅ then
    let m_φ ← (null)
else if m matches id.φ' then
    let m_φ ← (id.φ'.φ)
else if m matches map m' using m'' then
    if φ matches *.φ' then
        let m_φ ← (m' ∘ Subscript(m'', φ'))
    else
        let k.φ' ← (φ)
        let m_φ ← (Subscript(m', k) ∘ Subscript(m'', φ'))
else if m matches m' ∘ m'' then
    let m_φ ← (Subscript(m''[id/m'], φ))
else if m matches filter m' using m'' then
    let m_φ ← (if Subscript(m', φ) ∘ m'' then Subscript(m', φ) else null)
else if m matches if m' then m'' else m''' then
    let m_φ ← (if m' then Subscript(m'', φ) else Subscript(m''', φ))
else if m matches agg_[⇐](m') then
    let m_φ ← ((agg_[merge]({tmp := Subscript(m', φ)})).tmp)
else
    error {m does not produce a collection and can not be subscripted}
```

$$\textit{PrimitiveConstant}\ \frac{}{\widehat{c}(v) \mapsto c} \qquad \textit{Null}\ \frac{}{\widehat{null}(v) \mapsto null} \qquad \textit{EmptySet}\ \frac{}{\widehat{\emptyset}(v) \mapsto \{* \to null\}} \qquad \textit{Identity}\ \frac{}{\mathbf{id}(v) \mapsto v}$$

$$\textit{Subscripting}\ \frac{M(v) \mapsto \{..., k \to v_1, ...\}}{(M.k)(v) \mapsto v_1} \qquad \textit{Singleton}\ \frac{M(v) \mapsto v_1}{\{key := M\}(v) \mapsto \{k \to v_1, * \to null\}}$$

$$\textit{Merge}\ \frac{M_1(v) \mapsto \{k_i \to v_i\} \quad M_2(v) \mapsto \{k_j \to v_j\}}{(M_1 \widehat{\Leftarrow} M_2)(v) \mapsto \{k \to v \mid (k = k_i = k_j) \wedge (((v = v_i) \wedge (v_j = null)) \vee ((v = v_j) \wedge (v_j \neq null)))\}}$$

$$\textit{Map}\ \frac{\begin{array}{c} M_{coll}(v) \mapsto \{k_i \to v_i\} \\ M_{map}(v_i) \mapsto v_i' \end{array}}{(\mathbf{map}\ M_{coll}\ \mathbf{using}\ M_{map})(v) \mapsto \{k_i \to v_i' \mid v_i \neq null\}} \qquad \textit{PrimBinOp}\ \frac{\begin{array}{c} M_1(v) \mapsto v_1 : p \quad M_2(v) \mapsto v_2 : p \\ \theta \in \{+, *, -, /, =, \mathbf{AND}, \mathbf{OR}, \neq, <, \leq, >, \geq\} \end{array}}{(M_1 \widehat{\theta} M_2)(v) \mapsto v_1 \theta v_2}$$

$$\textit{Flatten}\ \frac{M_{coll}(v) \mapsto \{k_i \to v_i\}}{(\mathbf{agg}_{[\Leftarrow]}(M_{coll}))(v) \mapsto (v_0 \Leftarrow v_1 \Leftarrow \ldots)} \qquad \textit{PrimitiveAggregate}\ \frac{M_{coll}(v) \mapsto \{k_i \to v_i\}}{(\mathbf{agg}_{[\theta]}(M_{coll}))(v) \mapsto (((v_0 \theta v_1) \theta v_2) \theta \ldots)}$$

$$\textit{IfThenElse}\ \frac{M_{cond}(v) \mapsto true \quad M_{then}(v) \mapsto v_1}{(\mathbf{if}\ M_{cond}\ \mathbf{then}\ M_{then}\ \mathbf{else}\ M_{else})(v) \mapsto v_1} \quad \frac{M_{cond}(v) \mapsto false \quad M_{else}(v) \mapsto v_1}{(\mathbf{if}\ M_{cond}\ \mathbf{then}\ M_{then}\ \mathbf{else}\ M_{else})(v) \mapsto v_1}$$

$$\textit{Filter}\ \frac{M_{coll}(v) \mapsto \{k_i \to v_i\} \quad M_{cond}(v_i) \mapsto v_i'}{(\mathbf{filter}\ M_{coll}\ \mathbf{using}\ M_{cond})(v) \mapsto \{k_i \to v_i \mid v_i \neq null, v_i' = true\}} \qquad \textit{Composition}\ \frac{M_1(v) \mapsto v_1 \quad M_2(v_1) \mapsto v_2}{(M_1 \circ M_2)(v) \mapsto v_2}$$

Figure 2: A formal operational semantics for $\text{Bar}_{\mathcal{QL}}$.

Algorithm 2 ReadNormalize

Require: m, a $\text{Bar}_{\mathcal{QL}}$ monad
Ensure: m, a $\text{Bar}_{\mathcal{QL}}$ monad in read-normal form equivalent to m.
while m contains subexpression $m'.\phi$ where $m' \neq \mathbf{id}$, $\phi \neq \mathbf{tmp}$ **do**
replace m' in m with $\texttt{Subscript}(m', \phi)$

Lemma 5. *For any valid $\text{Bar}_{\mathcal{QL}}$ expression,* `ReadNormalize` *reaches a fixed-point that is in read-normal form.*

Proof. The rewrites applied by Algorithm 1 are monotonic. For each recursive step, either ϕ gets shorter, m is simpler (i.e., a sub-AST of the input), or both. Completeness can demonstrated by the existence of a rule for pushing down non-temporary key subscript operator surrounding every other operator, with the following exceptions: (1) $\mathbf{id}$ and other subscript operators are permitted by read-normal form, (2) c, $null$, $\hat{\theta}$, and $\mathbf{agg}_\theta$ each output primitive values, and are not valid inputs to the subscript operator. □

With a monad in read-normal form, we can see what fragment of its input the monad reads from.

Definition 12. *Given a $\text{Bar}_{\mathcal{QL}}$ monad m, its read set $\rho(m)$ is defined as the set of all point-reads not nested within a* **using** *clause.*

Monads appearing in the **using** clause of **map** and **filter** are applied to individual collection elements rather than the entire log as a whole. Consequently, we treat each of these as reading the entire collection.

5. Incremental $\text{Bar}_{\mathcal{QL}}$

We now construct a low-level, incremental semantics for $\text{Bar}_{\mathcal{QL}}$ to aid us in program analysis. These semantics provide a fine-grained representation of the changes applied to a value by a $\text{Bar}_{\mathcal{QL}}$ monad. To do so, we distinguish between monads that affect a limited fragment of their input, from monads that could potentially transform their entire input. We refer to this property as subdivisibility.

Definition 13. *A $\text{Bar}_{\mathcal{QL}}$ monad m is* **subdivisible at path** ϕ *if for all values v in the domain of valid inputs for m it holds that (1) $v.\phi$ and $m(v).\phi$ are both defined and both collections, and (2) the symmetric difference ($\triangle$) between the defined keys of either collection is of bounded size. That is:*

$$\exists \epsilon \forall v : \left| \{k | v.\phi.k \neq null\} \triangle \{k | m(v').\phi.k \neq null\} \right| < \epsilon$$

Subdivisibility effectively allows us to identify a finite subset of the keys of the collection at ϕ that will be modified. If the root path $[\,]$ is subdivisible, we can rewrite the entire monad as a finite set of smaller monads, each updating either one or all of the root's children. This subdivision process can be repeated recursively on all subdivisible children.

Definition 14. *A* point update *is defined by a 3-tuple:*

$$\langle \phi, f, m \rangle : \left(\vec{k} \times ((\tau \times \tau) \to \tau) \times (\tau \to \tau) \right)$$

- *The* target path ϕ *is a sequence of keys that identifies a nested collection element to which this update is being applied. In this sequence, the special wildcard key $*$ corresponds to all (non-null) values at that point in the hierarchy.*
- *The* combiner function f *is a procedure for merging a computed update value into the value at path ϕ in the input.*
- *The* update monad m *computes the update value.*

A point update defines a monad that replaces the value at path ϕ in the input with the result of evaluating $f(\mathbf{id}.\phi, m(\mathbf{id}))$. We call this the **equivalent monad**.

The purpose of the combiner function is to allow us to express point updates incrementally. We consider several different combiner functions below. For now, let us consider only the replacement function $[:=](v_{orig}, v_{upd}) \mapsto v_{upd}$.

Indivisible Point-Updates. Naturally, there are a variety of different ways to encode any given $\text{Bar}_{\mathcal{QL}}$ monad in terms of one or more point-updates.

Definition 15. *Given a set of independent point updates $\vec{U}$, we define the* **equivalent monad** *of $\vec{U}$ as the monad constructed by applying all of the point updates in $\vec{U}$ in parallel to their respective target paths. A set of independent point updates $\vec{U}$* **encodes** *a $\text{Bar}_{\mathcal{QL}}$ monad m if the equivalent monad of $\vec{U}$ is semantically equivalent to m.*

This encoding might be trivial (a replace-by update on the root value) or extremely detailed (add/multiply by a constant). To take full advantage of the point-update encoding, we would like it to be as fine-grained as possible. We capture this desire with the notion of indivisibility.

Definition 16. *A set of paths Φ is* **indivisible** *for a $\text{Bar}_{\mathcal{QL}}$ monad m if for all paths $\phi \in \Phi$, m is not subdivisible at ϕ. We say a point update is indivisible if its equivalent monad is not subdivisible at its target path.*

Let us consider the effect of each $\text{Bar}_{\mathcal{QL}}$ operation on subdivisibility. Let m be a $\text{Bar}_{\mathcal{QL}}$ monad, ϕ be a path, and v be a value for which both m and $\mathbf{id}.\phi$ are guaranteed to output a collection. Let $\texttt{keys}(\texttt{v}')$ be the set of keys in the collection v'. We can classify m based on the keys in its output as follows: (1) finite (F) when $|\texttt{keys}(m(v))|$ is independent of v, (2) finitely different (FD) when the bound on $|\texttt{keys}(m(v)) \bigtriangleup \texttt{keys}(\mathbf{id}.\phi(v))|$ is independent of v, and (3) arbitrary, when there is no independent bound on $|\texttt{keys}(m(v)) \bigtriangleup \texttt{keys}(\mathbf{id}.\phi(v))|$. For convenience, we establish the following ordering over these classes: $F < FD < A$.

Proposition 2. *Given a $\text{Bar}_{\mathcal{QL}}$ monad m and a path ϕ, m is subdivisible if and only if $m.\phi$ is finitely different.*

Given a monad m in read-normal form, we can determine its class recursively as follows. For path ϕ, a point-read at ϕ is in FD, while a point-read at $\phi' \neq \phi$ is in A. Filter and Flatten both restructure their inputs and are in A. The remaining collection-typed leaves are Assignment and EmptySet, as both produce collections of finite size.

The map operation does not alter the key-set of its input, and thus $\mathbf{map}\ m'\ \mathbf{using}\ m''$ has the same class as m'. Because classes are defined in terms of upper bounds, the **if** construct takes the worse (greater) class from its two clauses. Finally, the merge operation combines keysets similarly: a monad in F merged with a monad in FD produces a finite number of changes to the FD monad's keyset. A monad in A already has an unconstrained keyset. merging it with a monad in F or FD will not introduce new constraints, leaving the result in A.

Subdividing a monad. $\text{Bar}_{\mathcal{QL}}$ monads are converted into incremental $\text{Bar}_{\mathcal{QL}}$ by a simple subdivision process, where we first identify a set of indivisible paths to divide it into. This set must be sufficient to properly convey the semantics of the complete monad.

Definition 17. *A set of paths Φ is a* **complete mask** *for a $\text{Bar}_{\mathcal{QL}}$ monad m if for any value v in the domain of valid inputs to m with $v' = m(v)$, for every path ϕ defined in v or v, either ϕ or one of its ancestors is in Φ (i.e., $\exists \phi'.\phi'' = \phi : \phi' \in \Phi$), or $v.\phi = v'.\phi$.*

The first step in the process of subdividing a $\text{Bar}_{\mathcal{QL}}$ monad m is to obtain a set of paths that are both a complete mask and indivisible for m. This process is shown in Algorithm 3.

Algorithm 3 IndivisibleKeys

Require: m, a $\text{Bar}_{\mathcal{QL}}$ monad in read-normal form.
Require: ϕ, the path prefix being explored (default empty).
Require: `returnAdjustedPathset` **false** if the caller needs detailed information about Φ and will distinguish based on `class`, **true** if Φ should be adjusted accordingly instead (default **true**).
Ensure: $\texttt{class} \in F, FD, A$
Ensure: Φ, a set of paths that is a complete mask and indivisible for m.

```
let (m_1 ⇐ ... ⇐ m_n) ← m
let class ← F
for i ∈ [1, n] do
  if m_i matches id.φ then
    let Φ ← {}; class ← FD
  else if m_i matches id.φ' s.t. (φ ≠ φ') or agg_[⇐|θ](m') then
    let Φ ← {φ}; class ← A; return {Skip indivisible leaves.}
  else if m_i matches k := m' then
    let Φ ← Φ ∪ IndivisibleKeys(m', φ.k, true)
  else if m_i matches map m' using m'' then
    let class', Φ' = IndivisibleKeys(m', φ, false)
    class = max(class, class')
    for all φ' ∈ Φ' do
      for all φ'' ∈ IndivisibleKeys(m'', [], true) do
        let Φ ← Φ ∪ {φ'.φ''}
  else if m_i matches if m' then m'' else m''' then
    let class'', Φ'' = IndivisibleKeys(m'', [], false)
    let class''', Φ''' = IndivisibleKeys(m''', φ, false)
    class = max(class, class'', class''')
    Φ ← Φ ∪ Φ'' ∪ Φ'''
if returnAdjustedPathset ∧ (class ∈ {F, A}) then
  Φ ← {φ}
```

Theorem 1. *The output of Algorithm 3 is a complete mask and indivisible for its input.*

Proof. `IndivisibleKeys` mirrors the classification process outlined above. Each monad is split on the (associative) merge operator, and each child is tested for class. Recursion into the $\mathbf{map}$ and $\mathbf{if}$ operators is special-cased. If the path is identified as being subdivisible (i.e., in class FD), the subdivided expression is returned. Otherwise the path currently being explored is returned. The one exception is $\mathbf{id}.\phi$, which is the identity for the path currently being explored. Consequently, no paths are modified, and an empty ϕ is returned. For any path ϕ that is subdivisible, ϕ is never returned.

Completeness can be shown by exclusion. If ϕ is returned, the current path is completely covered. There are five classes of expression that can produce a value other than ϕ: Assignment, $\mathbf{id}.\phi$, $\mathbf{map}$, and $\mathbf{if}$, or a merge of any of the above. $\mathbf{if}$ statements are handled by trivial recursion. $\mathbf{map}$ produces a wildcard key for all of its children, and thus trivially covers the entire input.

Thus, the only non-trivial case is that of Assignments and $\mathbf{id}.\phi$ combined recursively with merge and is $\mathbf{if}$. Assignment operators modify at most a single key via the merge operation, and that key (or its subdivisible descendents) is guaranteed to be returned by recursion. $\mathbf{id}.\phi$ is the identity for ϕ, and modifies nothing. Consequently, the algorithm produces a complete mask. □

Now that we have a fine-grained set of paths containing the fragments its input that a $\text{Bar}_{\mathcal{QL}}$ monad affects, we can compute the details of these effects. We use a simple form of delta computation (a.k.a., program slicing) to identify the computation that produces the new value at a given path. This process is identical to that of Algorithm 1.

Theorem 2. *Given a $\text{Bar}_{\mathcal{QL}}$ monad m, and the set of point-updates $\vec{U}$ defined as below, $\vec{U}$ encodes m.*

$$\vec{U} = \{\langle \phi, [:=], \texttt{Subscript}(m, \phi)\rangle \mid \phi \in \texttt{IndivisibleKeys}(m)\}$$

Proof. By Theorem 1 $\texttt{IndivisibleKeys}(m)$ is a complete mask for m. By Lemma 5 $\texttt{Subscript}(m, \phi) \equiv m.\phi$, or the updated value at path ϕ. Evaluating everymonad so generated in parallel provides new values for every path ϕ in the complete mask. By Definition 17, paths not in a complete mask of m are not modified by m. Consequently, every path is either updated by a point update in $\vec{U}$ or not modified by m. □

Finally, we extract incrementality out of a given monad using Algorithm 4. This process identifies a combiner function that can be used to further generalize the point update. We consider the following three additional combiner functions:

$$[\Leftarrow|+=|\times=](v_{orig}, v_{upd}) \mapsto v_{orig}\ [\Leftarrow | + |\times]\ v_{upd}$$

Algorithm 4 Incrementalize

Require: m, a $\text{Bar}_{\mathcal{QL}}$ monad.
Require: ϕ, a path.
Ensure: f, a combiner function.
Ensure: Δm, a $\text{Bar}_{\mathcal{QL}}$ monad such that $m \equiv \lambda x.f(x.\phi, \Delta m(x))$.
  if m **matches** $\mathbf{id}.\phi \Leftarrow m'$ **then**
    let $\Delta m \leftarrow m'$
    let $f \leftarrow [\Leftarrow]$
  else if m **matches** $(m_{left})\theta(\mathbf{id}.\phi)\theta(m_{right})$ **and** $\theta \in \{+, \times\}$ **then**
    let $\Delta m \leftarrow (m_{left})\ \theta\ (m_{right})$
    let $f \leftarrow [\theta=]$

6. Log Compaction Revisited

Armed with a suitable monad language, we now return to our log compaction example. We begin at the end, with the subsumption and pairwise commutativity properties. The fine-grained detail of each update exposed by incremental $\text{Bar}_{\mathcal{QL}}$ is ideal for this purpose. We analyze each property, first in the context of individual point updates, and subsequently in the context of sets of point updates. We begin with a simple property of paths and read sets.

Definition 18. *Two paths ϕ and ϕ' are defined as* **ancestor-exclusive** *if $\phi \neq \phi'$ and neither ϕ nor ϕ' is an ancestor (prefix) of the other.*

Definition 19. *A monad m is called* **read-independent** *of a path ϕ if every path in the read-set $\rho(m)$ is ancestor-exclusive with ϕ. A point update u is read-independent of ϕ if its equivalent monad is read-independent.*

Subsumption. We start with the subsumption property. Recall that this property is defined over two monads m, m' where m' subsumes m when $m \circ m' \equiv m'$.

Lemma 6. *Consider two indivisible point updates $u = \langle \phi, f, m \rangle$ and $u' = \langle \phi', f', m' \rangle$ where $\phi = \phi'$ or one of its ancestors. The equivalent monad of u' subsumes that of u if and only if (1) m' is read-independent of ϕ and $f'\mathbf{is}{:=}$, or (2) u is the identity (i.e., u is $\phi{+}{=}0$, $\phi{\times}{=}1$, $\phi{\Leftarrow}\emptyset$, etc...).*

Proof. Subsumption of an identity operation is trivially guaranteed.

To re-cap, the replacement combiner is $[:=](v_{orig}, v_{upd}) \mapsto v_{upd}$. The output of this function is independent of its first input, guaranteeing that any replacement update subsumes all prior updates to the same path or one of its ancestors. The only other possible dependency is in the update computation m', which is asserted to be read-independent of ϕ. The converse can be shown trivially for the numerical combiners, as the output of each is guaranteed to be linearly or multiplicatively correlated with the input.

For the merge combiner ($\Leftarrow$), by indivisibility, the set of keys created or deleted by u is determined by its input. Consequently, it is always possible to identify some input to u that produces an output that will not be altered by u', and subsumption does not hold. □

Theorem 3. *Consider two sets of individisible point updates $\vec{U}$ and $\vec{U}'$ that encode $\text{Bar}_{\mathcal{QL}}$ monads m and m' respectively. m' subsumes m if every point update in $\vec{U}$ is subsumed by a point update in $\vec{U}'$ and if every point update in $\vec{U}'$ is read-independent of every target path in $\vec{U}$.*

Proof. If every effect (i.e., point update) of $\vec{U}$ is subsumed by an operation in $\vec{U}'$, the only other possible way for m to influence the output of m' is via one of the update computations. Read-independence guarantees that this is not possible. □

Commutativity. The other property of interest is pairwise commutativity over functional composition.

Lemma 7. *Consider any two indivisible point updates $u = \langle \phi, f, m \rangle$ and $u' = \langle \phi', f', m' \rangle$. The equivalent monads of u and u' commute if m and m' are read-independent of ϕ' and ϕ respectively, and either (1) ϕ and ϕ' are ancestor-exclusive, or (2) $\phi = \phi'$ and f and f' commute.*

Proof. Read independence guarantees that $m'(\mathbf{id}) = m'(\mathbf{id} \circ u)$ and visa versa. That is, the update value computed for each point update is identical, regardless of which point update is applied first.

If ϕ and ϕ' are ancestor-exclusive, then the effects of both updates are disjoint. If the update values are also order-independent, $u' \circ u \equiv u \circ u'$.

If $\phi = \phi'$, then precisely the same value is being modified. The resulting structure can be rewritten as

$$f'(f(\mathbf{id}, m(\mathbf{id})), m'(\mathbf{id} \circ u)) = f'(f(\mathbf{id}, m(\mathbf{id})), m'(\mathbf{id}))$$

If f and f' commute, then

$$= f(f'(\mathbf{id}, m'(\mathbf{id})), m(\mathbf{id})) = f(f'(\mathbf{id}, m'(\mathbf{id})), m(\mathbf{id} \circ u'))$$

And by read independence, we have commutativity. □

Theorem 4. *Consider two sets of indivisible point updates $\vec{U}$ and $\vec{U}'$ that encode $\text{Bar}_{\mathcal{QL}}$ monads m and m' respectively. m and m' commute if and only if every point update in $\vec{U}$ commutes with every point update $\vec{U}'$.*

Proof. By Definition 15 $\vec{U}$ and m (resp. $\vec{U}'$ and m') are semantically equivalent. If the components of m can be commuted individually across m', then m can be safely decomposed, commuted, and reconstructed on the other side. Conversely, if there is at least one point update that can not be commuted, then the corresponding path in the output of m or m' will change depending on their relative order. □

6.1 Log Patterns

We now turn our attention to recoverability and the two log transforms of interest. Concretely, we will demonstrate that by exposing the semantic properties of individual log entries, the necessary conditions for recoverability of both deletion and composition transforms can be expressed in terms of simple graph patterns over the log, or *log queries* that can be efficiently evaluated incrementally.

Information obtained through program analysis is encoded in a simple graph structure $G_{log} = \langle V, E \rangle$ with labeled edges, defined

as follows. V include one vertex for every monad m in the log, one vertex for point update u in the encoding of each m, one vertex for every type of combiner function f, and one vertex for every possible path ϕ. E includes labeled edges as follows:

- $\langle m, \texttt{prev}, m' \rangle, \langle m, \texttt{next}, m' \rangle$ for all adjacent log entries.
- $\langle m, \texttt{composedOf}, u \rangle$ for every point update m in the set of indivisible point updates that encodes m.
- $\langle u, \texttt{target}, \phi \rangle, \langle u, \texttt{readsFrom}, \phi \rangle$ for the target path of u and the read set $\rho(u)$ respectively. relationship.
- $\langle u, \texttt{combiner}, f \rangle$ for the combiner function f of u.

We can define several useful transforms over this graph using SPARQL, a standard graph query and manipulation language.

Composition. For any two monads m and m', a subgraph describing $m'' = m \circ m'$ can be generated. We begin by generating a set of nodes u'' for each indivisible point update in the encoding of m. Point updates in m and m' operating on disjoint paths are passed through unchanged. If one update modifies a subset of the other however, the two must be combined. Since the outermost path is, by definition, indivisible, the merged point-update must modify the outermost path. The following SPARQL updates produce the relevant edges; the symmetry on m and m' allows only the outermost path to pass through.

```
CONSTRUCT { m'' composedOf u''. u'' basedOn ?u. }
WHERE { m composedOf ?u. ?u target ?phi.
        NOT EXISTS { m' composedOf ?u'.
         ?u' target ?phi'. prefixOf(?phi, ?phi') }.}
CONSTRUCT { ... as above, swapping m and m' ... }
```

All that remains is to create the target and readsets for each u''. The target is defined as the outermost path, and the readset is defined by the union of the two readsets, less any values overwritten by m'.

```
CONSTRUCT { u'' target ?phi. }
WHERE { u'' basedOn ?u; target ?phi.
        u'' basedOn ?u'; target ?phi'.
        prefixOf(?phi, ?phi') }
CONSTRUCT { u'' readsFrom ?phi. }
WHERE { u'' basedOn ?u; readsFrom ?phi.
       (( m' composedOf ?u ) OR
        NOT EXISTS { m' composedOf ?u'; target ?phi'
         prefixOf(?phi', ?phi) } ) }
```

Subsumption. The necessary conditions (Lemma 6) for subsumption $\mathcal{S}(m, m')$ can be phrased as a condition in SPARQL. Every path modified by m must be overwritten by m', and and m' may not read from an affected path.

```
NOT EXISTS {
  m composedOf ?u; target ?phi.
  (NOT EXISTS { m' composedOf ?u; target ?phi'.
                prefixOf(?phi, ?phi') }
   OR (m' composedOf ?u; readsFrom ?phi'.
       NOT ancestorExclusive(?phi, ?phi'))) }
```

Commutativity. The necessary conditions (Lemma 7) for pairwise commutativity $\mathcal{C}(m, m')$ can phrased as a condition in SPARQL. The read set of m may not intersect with the targets of m' and visa versa. Similarly, their respective point update targets must be ancestor exclusive, unless the targets are identical and the combiner functions are commutative.

```
NOT EXISTS { m composedOf ?u; target ?phi
             m' composedOf ?u'; readsFrom ?phi' }.
NOT EXISTS { m composedOf ?u; readsFrom ?phi
             m' composedOf ?u'; target ?phi' }.
NOT EXISTS { m composedOf ?u; target ?phi
             m' composedOf ?u'; target ?phi'
             (NOT ancestorExclusive(?phi,?phi')
              OR (?u combiner ?f. ?u' combiner ?f'.
                  ?f commutesWith ?f') ) }
```

Recoverability. The necessary conditions for recoverability of deletion and composition transforms (Lemmas 4 and 2) can both be expressed in terms of commutativity and subsumption tests over compositions of monads. The above graph-query-based tests, and the definition of a graph view for composed monads, are thus sufficient to assert recoverability of a deletion or composition transform.

Specifying these properties declaratively allows a typical graph database system to use an assortment of indexing, query optimization, and incremental evaluation strategies to evaluate these properties efficiently. Such optimizations are future work.

7. Related Work

There has been much work focused on the formalization of query languages and database models for complex data [3, 4, 29]. Much of this work is based on monad algebra, Lawvere theories, and universal algebra [5, 8, 25, 26]. Manes *et al.* [30] showed how to implement collection classes using monads. Cluet [20] is an algebra based query language for an object-oriented database system. Our work is based on the same fundamental theories. In the following we compare our work to previous results.

Languages for Transforming Hierarchical Data. There has been considerable work [1–3, 15] on the transformation of hierarchical data. Two approaches have become dominant in this area: Nested Relational Calculus [34] and the Monad Algebra [28]. Our own approach is closely based on the latter, adapted for use with labeled sets, and with the intentional exclusion of the superlinear time complexity pairwith operator (or equivalently, the cartesian cross-product).

Semistructured Data. Also closely related is work on managing semistructured data [14]. The vast majority of recent efforts in this area have been on querying and transforming XML data. One formalization by Koch [27] is also closely based on Monad Algebra. Work by Cheney follows a similar vein, in particular (F)LUX [18, 19], a functional language for XML updates. In [9], Benedikt and Cheney present a formalism for synthesizing the output schema of XML transformations, similar to our notion of the compositional compatibility of mutations. More recently, there has also been interest in querying lighter-weight semistructured data representations like JSON[11, 12].

Algebraic Properties of State Updates. The distributed systems community has identified a number of algebraic properties of state mutations that are useful in distributed concurrency control. Commutativity of updates has been explored extensively [35, 37], but the typical assumption is that a domain-specific commutativity oracle is available, such as for edits to textual data [31, 35]. Our notion of subsumption is quite similar to the Badrinath and Ramamritham [7]'s recoverability property. Unlike subsumption, this property is defined in terms of observable side-effects rather than state, but is otherwise identical. Like prior work on commutativity, they assume that a domain-specific oracle has been provided. Several efforts have been made to understand domain-specific reconciliation strategies. Feldman *et al.*'s Operational Transforms [24] are

analogous to our our mutation languages, but assume that domain-specific operations analogous to our merge operation are available. Perhaps the closest effort to our own has been Preguica *et al.*'s IceCube [33], and Edwards *et al.*'s Bayou [21], each of which exploit a range of specific algebraic properties of updates to distributed state. However, both systems must be explicitly adapted to specific application domains by the construction of domain-specific property oracles, or by mapping the application's behavior down to a trivial update language. To the best of our knowledge, none of these areas have been explored in the context of a non-trivial state update language.

Update Sequencing. The use of distributed logs and publish/subscribe to apply a canonical order to updates has also been explored extensively by the distributed systems and database communities. Ellis *et al.* noted the relevance of sequencing to distributed concurrency control [22]. Eugster *et al.* identified the usefulness of sequencing updates to distributed collection types [23]. Domain specific applications of similar ideas can be found in work by Ostrowski and Birman [32], Weatherspoon *et al.* [36], and others.

Intent-Based Updates. The use of intent-based (*i.e.*, operational) updates appears frequently in database literature, especially in the context of distributed databases, where it is used to reduce communication overhead. Two concrete examples are Ceri and Widom's Starburst [16], and Chang *et al.*'s BigTable [17].

8. Conclusion

We have introduced a formal framework for reasoning about properties over monadic logs, a functional representation of shared state in collaborative web-applications. A theory of log transformation has been presented, showing preservation of safety properties like tail-correctness and recoverability. Overall properties on the log itself can be expressed declaratively through SPARQL queries, leveraging the structure of the log itself to assert those properties.

References

[1] S. Abiteboul and N. Bidoit. Non first normal form relations: An algebra allowing data restructuring. *JCSS*, 33(3):361–393, 1986.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel query language for semistructured data. *JODL*, 1(1):68–88, 1997.

[3] Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *VLDBJ*, 4(4):727–794, October 1995.

[4] Serge Abiteboul and Richard Hull. IFO: a formal semantic database model. *ACM TODS*, 12(4):525–565, November 1987.

[5] Jiří Adámek, Mahdieh Haddadi, and Stefan Milius. From corecursive algebras to corecursive monads. In *CALCO*, pages 55–69, 2011.

[6] Sumit Agarwal, Daniel Bellinger, Oliver Kennedy, Ankur Upadhyay, and Lukasz Ziarek. Monadic logs for collaborative web applications. In *WebDB*, 2013.

[7] B R Badrinath and Krithi Ramamritham. Performance evaluation of semantics-based multilevel concurrency control protocols. In *SIGMOD*, May 1990.

[8] Adriana Balan and Alexander Kurz. On coalgebras over algebras. *Electron. Notes Theor. Comput. Sci.*, 264(2):47–62, August 2010.

[9] M. Benedikt and J. Cheney. Semantics, types and effects for xml updates. *DBPL*, pages 1–17, 2009.

[10] Phillip A Bernstein, CW Reid, and Sudipto Das. Hyder–A Transactional Record Manager for Shared Flash. *CIDR*, 2011.

[11] K. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.C. Kanne, F. Ozcan, and E.J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12), 2011.

[12] K. Beyer, V. Ercegovac, J. Rao, and E. Shekita. Jaql: A json query language. *URL: http://jaql. org*, 2009.

[13] Kenneth Birman and Robert Cooper. The isis project: Real experience with a fault tolerant programming system. In *Proceedings of the 4th workshop on ACM SIGOPS European workshop*, pages 1–5. ACM, 1990.

[14] P. Buneman. Semistructured data. In *PODS*, pages 117–121, 1997.

[15] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[16] Stefano Ceri and Jennifer Widom. Production rules in parallel and distributed database environments. *PVLDB*, 1992.

[17] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM TOCS*, 26(2):4, 2008.

[18] J. Cheney. Lux: A lightweight, statically typed xml update language. *SIGPLAN*, 1060:25–36, 2007.

[19] J. Cheney. Flux: functional updates for xml. *ACM SIGPLAN Notices*, 43(9):3–14, 2008.

[20] S. Cluet, C. Delobel, C. Lécluse, and P. Richard. Reloop, an algebra based query language for an object-oriented database system. *Data Knowl. Eng.*, 5(4):333–352, October 1990.

[21] W Keith Edwards, Elizabeth D Mynatt, and Karin Petersen. Designing and implementing asynchronous collaborative applications with Bayou. In *UIST*, 1997.

[22] C A Ellis and S J Gibbs. Concurrency control in groupware systems. *SIGMOD*, 1989.

[23] Patrick Th Eugster and Rachid Guerraoui. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. *ECOOP*, 2000.

[24] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. SPORC: Group Collaboration using Untrusted Cloud Resources. In *OSDI*, 2010.

[25] Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electron. Notes Theor. Comput. Sci.*, 172:437–458, April 2007.

[26] G. Jaeschke and H. J. Schek. Remarks on the algebra of non first normal form relations. In *PODS*, pages 124–138, 1982.

[27] Christoph Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM TODS*, 31(4):1215–1256, December 2006.

[28] K. Lellahi and V. Tannen. A calculus for collections and aggregates. In *Category Theory and Computer Science*, pages 261–280. Springer, 1997.

[29] Zoran Majkic and Bhanu Prasad. Kleisli category and database mappings. *IJIIDS*, 4(5):509–527, October 2010.

[30] Ernie G. Manes. Implementing collection classes with monads. *Mathematical. Structures in Comp. Sci.*, 8(3):231–276, June 1998.

[31] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data Consistency for P2P Collaborative Editing. In *CSCW*, page 259, 2006.

[32] Krzysztof Ostrowski and Ken Birman. Storing and accessing live mashup content in the cloud. *SIGOPS Review*, 44(2), April 2010.

[33] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. *On The Move to Meaningful Internet . . .* , 2003.

[34] M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM TODS*, 13(4):389–417, 1988.

[35] Marc Shapiro and Nuno Preguiça. Designing a commutative replicated data type. Technical report, CORR, October 2007.

[36] Hakim Weatherspoon, Patrick Eaton, Byung-Gon Chun, and John Kubiatowicz. Antiquity: exploiting a secure log for wide-area distributed storage. In *EuroSys*, 2007.

[37] William E Weihl. Commutativity-based concurrency control for abstract data types. *IEEE TC*, 37(12):1488–1505, 1988.