Cloud Types:
Robust Abstractions for Replicated Shared State

March 27, 2014

Technical Report
MSR-TR-2014-43

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Important

This document is *work in progress*. Feel free to cite, but note that we will update the contents without warning (pages are timestamped at the bottom right), and that we are likely going to publish the content in some future venue, at which point we will update this paragraph.

Cloud Types: Robust Abstractions for Replicated Shared State

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

Daan Leijen
Microsoft Research
daan@microsoft.com

Manuel Fähndrich
Google
fahndrich@google.com

Abstract

In the age of cloud-connected mobile devices, users want responsive apps that read and write shared data everywhere, at all times, even if network connections are slow or unavailable. Cloud types [6] have been proposed as a solution that lets programmers declare, read, and update shared structured data while hiding tricky consistency issues related to update propagation and conflict resolution. However, previous work on cloud types does not satisfactorily address (1) how to best understand the weak consistency model, (2) where consistency pitfalls continue to lurk, and (3) how to implement the system efficiently and reliably.

We address these questions by (1) introducing the GLUT model (global log of update transactions), suitable as a mental reference model that helps programmers to visualize executions and reason about correctness, (2) describing typical consistency pitfalls (anti-patterns) and how to avoid them using cloud types, and (3) an efficient client-server implementation of GLUT that uses robust streaming and optimal delta reduction.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Distributed programming

Keywords Concurrent revisions, Distributed applications, Eventual consistency.

1. Introduction

Many applications can benefit from replicating shared data across devices, because it is often desirable to keep applications responsive even if network connections are slow or unavailable. Unfortunately, the CAP theorem [1, 4, 14] shows that strong consistency (such as linearizability or sequential consistency) requires communication with a reliable server or with a majority partition on each update, which becomes slow or impossible if network connections are slow or unavailable. Since responsiveness is often more important than strong consistency, researchers and practitioners have proposed the use of various forms of eventual consistency [5, 11, 17, 26]. In such systems, update propagation and conflict resolution is lazy, proceeding only when network conditions permit, and replicas may temporarily differ, while converging to the same state eventually.

Although eventual consistency offers clear benefits, it is also more difficult to understand, both for system implementors and client programmers, motivating the need for simple programming models for weakly consistent shared data. One recently proposed approach in this area is to provide (1) special data types called *cloud types* [6] to declare cloud-shared data, which implicitly specify how conflicts are resolved, and (2) providing *eventually consistent transactions* [5], which allow clients to group multiple updates into packets that are interleaved and propagated atomically. We call these packets *update transactions*.

Our experience with implementing cloud types and update transactions in a scripting language for mobile devices suggests that they provide significant benefits. In particular, the elimination of all error handling code and almost all synchronization code substantially simplifies the app development. However, we also discovered that there remains room for improvement in several aspects:

- *Reasoning.* Client programmers often misunderstand where exactly they risk consistency errors, and require an operational system model to understand how to write correct programs. The existing consistency models for cloud types and update transactions are either too abstract for non-experts in memory consistency (e.g. the axiomatic model in [5]), or too complicated and overly general for the situation at hand (e.g. the revision diagram model in [5, 6]).
- *Implementation and Performance.* The client-server system model proposed in [6] transmits the entire state in each message, which is impractical unless the amount

of data shared is very small. Moreover, the pushing and pulling of updates between client and server cannot proceed concurrently but is forced to alternate, which introduces significant delays.

In this paper, we describe several improvements in these areas. Specifically, we make the following contributions:

- We introduce the GLUT reference model (global log of update transactions), an operational model describing the system behavior precisely, yet abstractly (Section 2). It achieves strong eventual consistency by constructing both a global log of update transactions as well as keeping local update logs. In related work, we compare it to the TSO (total store order) memory model.
- We show in Section 3 how this model can be used effectively within a programming language, offering high level abstractions to program with persistent shared state almost as conveniently as with regular global variables.
- Using the programming model, we explain typical consistency errors (anti-patterns) by walking through several examples, and show how to avoid them. We conclude with a precise data model describing all the primitive read and update operations (Section 3).
- We present a detailed system implementation model of GLUT that provides significant performance and reliability advantages:

  Robust Streaming. Updates are streamed continuously in both directions between server and client. The model shows precisely how connections are established, how they can fail, and how they can be reconnected, without disrupting the execution of the client program at any point (Section 4).

  Optimal Delta Reduction. Update sequences often exhibit redundancy. If a variable is assigned several times, only the last update matters. We show how to implement update reduction, and prove that our implementation is correct and optimal (Section 5). This means that we store and propagate a minimal amount of information only. When clients accumulate updates while offline, they need not store every single update, and they transmit only a minimal sequence of updates to the server when reconnecting.
- We have implemented the ideas presented in this paper in the TouchDevelop programming language and development environment. Thus, we have made the cloud types programming model publicly available online for inspection and experimentation, and we provide links to a dozen example applications.

Overall, our work marks a big step forward in advancing cloud types and update transactions as a programming model for eventual consistency, by providing both an understandable high-level system and data model, and a detailed implementation containing powerful and interesting optimizations.

2. Reference Model

The first contribution of this paper is to give a simple consistency model that programmers can use in practice to reason about their programs at a high abstraction level. We call this model GLUT, or global log of update transactions. The model has the following ingredients:

- A shared global log of all update instructions which represents our persistent data. Section 3 explains how the data is defined using *cloud types*;
- Each client issues read- and update instructions on the data concurrently;
- Each client regularly calls yield to indicate transaction boundaries. Yielding is non-blocking and is only used to delineate transactions. In between yields, all updates are atomic.

Informally, the GLUT model operates as follows:

- Any update operation by a client is first only locally visible and confined to a local buffer for the current transaction. On yield these update operations become committed.
- The shared global log of update transactions is built by interleaving the committed updates of all clients.
- Clients are aware only of some prefix of the shared log. When a client performs a read, it see the a value that is consistent with a composite of log pieces, in the following order:
  1. uncommitted updates in the current transaction,
  2. local committed updates that are not yet part of the known prefix,
  3. the known prefix of the shared global log.
- During yield, clients may become aware of a larger prefix of the shared log. This means that the state observed by the client program may change. It also guarantees eventual consistency, since all clients (except crashed ones) eventually learn about the entire shared log.

Using this reference model we can precisely specify the guarantees and behavior of our system, yet it abstracts low-level system behaviors like client, server, and network failures and/or delays. Moreover, it allows for large transactions which make it very easy for programmers to reason about most code.

Automatic Transactions. For event-based programs, rather than relying on the programmer to sprinkle the code with yield instructions, it is easy and convenient to call yield automatically between event handlers. Thus the user is guaranteed that all updates within an event handler are atomic, without writing a single extra line of code.

Flush. A client can issue the flush instruction which blocks until all previous updates have propagated to the global shared log. This is needed when a client requires stronger consistency than what is afforded by lazy update propagation through yield. In our experience, few client applications require it, and if so, only in specific situations where the problem domain requires true arbitration (such as finalizing a reservation, ending an auction, or joining a game with an upper limit on the number of players). A flush instruction necessarily involves communication and may block if the network is partitioned. Thus, it should thus be used sparingly.

2.1 The Formal Model

We will now give a formal reference model, which precisely specifies the guarantees, yet abstracts low-level system behaviors. It is deliberately unrealistic in several aspects:

- The model keeps a globally shared log of updates, and local logs per client, which grow without bounds.
- There is no communication. Instead, we specify the behavior using atomic transitions that modify the shared system state.
- There are no failures of any kind.

These deliberate simplifications allow us to reason about program behavior at a high abstraction level (Section 3), which is the key challenge when programming with replicated shared state. Yet, all of these simplifications are justifiable: In Section 4 we present a realistic client-server implementation model that conforms fully to the reference model, but is very efficient. In particular, it avoids storing a log on the server and guarantees that only the minimal amount of data is stored (using log reduction, Section 5), and it completely hides client, server, and network failures.

Labeled Transition System. The formal model is specified as a labeled transition system, consisting of shared state and nondeterministic atomic transitions. This type of specification is well suited for operational specifications of consistency protocols [13, 19] as it enables both rigorous reasoning about invariants and model checking [16].

We represent data using abstract types for updates, reads, and values. The semantic meaning is represented by a single function rvalue that takes a read operation and a sequence of updates, and returns the value that results from applying all the updates in the sequence to the initial state of the data:

```
type Update; type Read; type Value;
function rvalue: Read × Update* → Value
```

Update sequences are stored in *rounds*. Each round represents updates that were issued by a client inside one or more update transactions. Rounds by each client are numbered consecutively, starting with 1.

```
struct Round { client: Client; number: ℕ;
updates: Update*; }
```

```
// system state
log: Round* := []; // global shared log
class Client {
    known : ℕ:= 0; // prefix of log known to this client
    current : Round := new Round(this,1,[]);
    committed: Round* := []; // committed rounds of this client
    flushing: ℕ:= 0; //nonzero if a flush is in progress
}
// externally visible system transitions
external update(c:Client; u:Update) {
    requires c.flushing = 0;
    c.current.updates.append(u);
}
external read(c:Client, r:Read) : Value {
    requires c.flushing = 0;
    var compositelog := knownlogprefix(c)
          · c.committed[numrounds(c, knownlogprefix(c))..]
          · c.current;
    return rvalue(r, updates(compositelog));
}
external yield(c:Client) {
    requires c.flushing = 0;
    if (*) commit_current(c);
    while (*) learn_next_logentry(c);
}
// internal system transitions
internal grow_log(c:Client) {
    requires numrounds(c,log) < c.committed.length;
    log = log  · c.committed[numrounds(c,log)];
}
```

Figure 1. States and transitions of the GLUT system model.

```
procedure commit_current(c:Client) {
    c.committed := c.committed  · c.current;
    c.current = new Round(this, c.current.number + 1, []);
}
procedure learn_next_logentry(c:Client) {
    requires c.known < log.length;
    c.known := c.known + 1;
}
function knownlogprefix(c: client): Round* {
  return log[0..c.known];
}
function numrounds(c: client, seq: Round*): ℕ{
  return |{ r | r in seq and r.client = c }|;
}
function updates(seq: Round*): Update* {
  return seq[0].updates ··· seq[seq.length1].updates;
}
```

Figure 2. Auxiliary procedures and functions for the GLUT system model.

System state. The system state is defined at the top of Fig. 1. It stores (1) a global shared log consisting of rounds, and (2) a client object for each client. Each client object stores (1) an index known into the global shared log, indicating the prefix of the shared log that this client knows about, (2) the current round, containing a sequence number and uncommitted updates, (3) a sequence committed of all the rounds that this client has committed, and (4) a number indicating if a flush is in progress on this client.

System transitions. There are seven system transitions overall (broken up into Fig. 1 and Fig. 3). Six of them model client instructions and are considered externally visible (this distinction matters later when we consider refinement), and two are internal. All of them execute atomically, and have one or more precondition, meaning that they fire only in certain states. For example, the read, update, and yield transition all have a precondition stating that clients may not invoke them while waiting for a flush to complete on this client. Some of them (e.g yield) contain nondeterministic choices, shown as * expressions.

Updates and Reads. If a client c performs an update, it is appended to the updates in the current round of c. If it performs a read, it first constructs the visible sequence of updates by concatenating (1) all updates in the log prefix known to c computed by the helper function knownlogprefix(c) in Fig. 2, (2) all the committed rounds by *c with the exception* of the ones that are already in the known log prefix, and (3) the uncommitted updates in the current round. Note that the visible server and committed rounds can change during a yield or flush only. Thus, the user code does not need to worry about the stability of reads — consecutive reads without a yield or flush in between see a consistent state.

Yield. As expected, a yield may commit the current round. However, it is not forced to do so (the commit_current call is guarded by a nondeterministic choice). This allows implementations to skip some commits, and thus to aggregate several transactions into a single round, which is important to keep storage and network requirements under control. Also during yield, the client may become aware of a longer prefix of server rounds by calling learn_next_logentry one or more times, which increments the value of the confirmed field.

Flush. (Fig. 3) The flush operation is nonatomic, thus there are two externally visible transitions that model its beginning and its end. The flush_begin stores the current round number in the c.flushing field and commits the current round. While the flush is in progress, the flush_yield can fire at any time - it is equivalent to yield called by the client, but the client is blocked at this point so we need an alternative internal transition to allow the client to become aware of longer prefixes. Once the known prefix includes the round that was flushed (second precondition of flush_end), the flush may end. Flushes block only the client that issues them and have no effect or interaction with other clients.

```
external flush_begin(c:Client) {
    requires c.flushing = 0;
    c.flushing := c.current.number;
}
internal flush_yield(c:Client) {
    requires c.flushing > 0;
    if (*) commit_current(c);
    while (*) learn_next_logentry(c);
}
external flush_end(c:Client) {
    requires c.flushing > 0;
    requires c.flushing = numrounds(c, knownlogprefix(c));
    c.flushing := 0;
}
```

Figure 3. Flush transitions for the GLUT system model.

2.2 Guarantees

The GLUT model is quiescently consistent and eventually consistent (under the usual fairness assumptions on the nondeterministic choices) and causally consistent. It is quiescently consistent, because after a sufficient number of yields without further updates, the server will incorporate all client rounds in its log, and all clients end up with confirmed being equal to the length of the server log. At that point, each client state is equivalent because no unconfirmed commits, nor current round updates are present.

The model is also eventually consistent in the sense of visibility/arbitration-based axiomatic definitions [5, 7, 8]. In particular, the model provides eventually consistent transactions according to the axiomatic definition in [5]. The arbitration order is simply the order in the server log, and visibility is determined as spelled out in the definition of reads.

The model is causally consistent because an update U by some client C cannot become visible to other clients before all of the updates (let's call them V) that are visible to client C at the time it performs U. The reason is that the updates V consist of (1) the common server prefix, or (2) local unconfirmed or uncommitted updates, which are all guaranteed to become visible to other clients no later than U.

When clients issue updates, those updates are stored in the current round. They are isolated until that round is committed, i.e. the updates are not visible to the server or other clients. They are, however, visible to subsequent operations by the same client (a property which is sometimes called Read-my-Writes [25]).

Our update transactions are different from conventional transactions (read-committed, serializable, snapshot isolation, or parallel snapshot isolation) since they do not check for any read or write conflicts. In particular, they never fail. The advantage is that they are highly available [2], *i.e.* progress is not hampered by network partitions. The disadvantage is that unlike serializable transactions (but like read-committed, snapshot, or parallel snapshot transactions), they

are not always sufficient to maintain all data invariants. We discuss the impact of this fact in Section 5.

2.3 Comparison to TSO

GLUT appears superficially similar to TSO[28] (total store order), a widely used relaxed memory model. However, even if we ignore the obvious differences (support for transactions and cloud types) for a moment, they are not equivalent, because TSO reads always read the latest shared state, while GLUT may see a stale prefix of the shared log. In particular, the following stale-read litmus test shows an outcome that is not possible on TSO, but is possible on GLUT.

write A, 2 ; yield read B, 0	write B, 1 ; yield write A, 1 ; yield read A, 2

Under TSO, seeing read A, 2 implies that the write A, 1 was overwritten by write A, 2, thus already drained to memory. Therefore write B, 1 has moved to memory thus the left program must read B as 1. Not so under GLUT: even if write B, 1 is in the server log, the left program may not yet see it.

3. Data Model

In general, programmers choose data models appropriate for location, size, and lifetime of the data. For example, persistent data is often modelled using relational databases or scalable key-value stores, while memory-resident data is typically represented by variables, arrays, and records, sometimes using a garbage-collected heap.

Choosing the right data model for persistent, replicated, shared data is paramount to keep the complexity of conflict resolution and garbage collection under control. Replicated data types [7, 20, 21] encapsulate those challenges within simple abstract data types such as counters, sets, or lists. Cloud types [6] go one step further, allowing programmers to express interrelated data structures. To illustrate cloud types in general, and how they integrate with the GLUT model, we now walk through a series of examples, demonstrating common consistency errors (anti-patterns) and how they can be solved.

3.1 Bird Watch Example

Let's write a program that keeps track of bird sightings. To start, we will just count the overall bird sightings using a global cloud number (nr):

```
cloud nr birdcount ;

function sighting() {
  birdCount.set(birdCount.get() + 1)
}
```

In the example, we assume there is a UI that invokes the sighting function. Moreover, we assume that the outer event loop calls yield after each event is handled. This ensures that all operations done in an event are part of one update transaction, and that individual operations are never interleaved with other distributed updates. This ensures that the programmer can always reason sequentially over the (cloud) state within each event handler. Also, every cloud value comes with a *default value* and they never have to be initialized. For numbers, the default is 0.

Nevertheless, the example is still wrong as it fails to count bird sightings reliably. Suppose two clients both have a sighting at the same time and both increment the count to, say, 1. After yielding, both update transactions, set(1), are appended to the global log and the final value of the bird count will be just 1. The anti-pattern here is that updates to a cloud value must make sense even if some 'earlier' updates are not yet visible to the local client.

To address this issue, cloud types generally come with a richer set of operations than just set. In particular, the nr type has an add operation which works incrementally:

```
cloud nr birdCount ;

function sighting() {
  birdCount.add( 1 )
}
```

In this case, a concurrent sighting appends two add(1) update transactions to the log, resulting in a correct global count of sightings.

We now extend our application to keep track of both the name and count of each bird sighting. Our data model supports *cloud tables* to maintain rows of cloud values:

```
cloud table Birds {
  name : str ;
  count: nr ;
}

function sighting(name: string) {
  var bird = find(name) ;
  bird.count.add(1) ;
}

function find( name : string ) {
  var bird ;
  foreach( bird in Birds ) {
    if (bird.name.get() == name)
      return bird ;
  }
  bird = Birds.new() ;
  bird.name.set( name ) ;
  return bird ;
}
```

On a sighting, we first call the find function to see if there already exists an entry for the particular bird in the cloud table. If no such entry is found, we create a new row using the new function and return that instead. Just like before, we then invoke add(1) to add reliably to the bird count.

Again though, there is a problem with our example. It is possible that two concurrent clients both create a new row for a new sighting of the same bird name because they cannot not see each others' updates yet. The anti-pattern here is the test if (not exist) create which is generally a problem because the element may already exist but is not yet visible

to the local client. For this situation, our data model provides *cloud indices* which are conceptually infinite key-value dictionaries where all entries are pre-initialized. In our case, the Birds cloud index is keyed by the bird name:

```
cloud index Birds[name : string] {
  count: nr;
}

function sighting( name : string ) {
  Birds[name].count.add(1);
}
```

Now the example works as expected. In particular, as shown in Figure 4, each update transaction will consist of the entire Birds[name].count.add(1) expression. The server atomically resolves the first Birds[name] part, potentially creating a fresh entry, and then perform the field operation count.add(1).

Note the contrast between cloud tables and indices: the cloud tables give the ablity to create (globally) *fresh* and *ordered* rows addressed by unique identities, while cloud indices provide unordered records of fields addressed by one or more key values. Global cloud variables as in our first example are internally implemented using a designated cloud index without keys, thus with one entry containing all globals as fields.

Even though a cloud table was not the right data structure for tracking the bird count, it is useful in other cases. If the bird log were to also keep track of individual sightings of birds, a table would be the correct structure for keeping track of rows of sightings containing the bird name, the person doing the sighting, the place, date and time.

This concludes our birding example. Note how concise and robust the final example is. Even though this code offers full distributed operation with eventual consistency and robustness under disconnected operation, there is almost no noise: no special error handling code, retrying of transactions, checking of connections, special server code, etc. All of this is taken care of by the implementation of the GLUT model. As we saw with the anti-patterns, we still need to carefully consider the implications of those issues that are intrinsic to distributed operation, but the GLUT model gives the programmer a robust mental model to think about these.

3.2 The data model syntax

Formally, the complete syntax of our data model is defined in Figure 4. For our purposes, we define three basic cloud types: numbers, booleans, and strings. The first entry of each $\mathsf{Val}_{\mathsf{type}}$ declaration defines the default value for each type. Also, each type comes with a set of valid operations $\mathsf{Fop}_{\mathsf{type}}$ that can be performed on values of that type.

The Uid type is for unique identifiers that are used to identify table rows. We use a single Rid type to identify records of cloud values which can be either a table name indexed by a uid (as tname(uid)), or an index name indexed by keys.

Finally the various Update and Read operations are defined. Note how fields are indexed uniformly in updates and reads, using the syntax rid.fname.ftype. In particular, the field type is included. We do this because we want to avoid needing an explicit schema for the server data, but we do want to all operations to be type safe. We can allow for every field name to be indexed by its type because it incurs no performance or storage overhead. This becomes apparent when we define the exact semantics of the data model in next section.

3.3 Semantics

We formally define the semantics of our data model by defining read and update operations on state objects (Fig. 5). State objects store the current state, represented by (1) the current (i.e. created and not deleted) row identifiers, which are stored in rows, separately for each table, and sorted by creation order, and (2) all non-default field values, which are stored in the map fields. State objects implement three methods, read for performing read operations on the current state, update for performing updates, and targets_deleted_data which can check if an update is redundant (i.e. targets data that is already deleted and thus has no effect).

State objects are of minimal size, i.e. they do not hold on to irrelevant information: they do not store fields that have the default value, and when a row is deleted, its field values are also removed. In particular, the state object does not contain any tombstones. The only exception is the used field which does indeed store all previously used identifiers. However, this field is a *ghost* field, used for proof purposes only: it does not affect control flow (other than assertions) and is not present in the physical implementation.

We now discuss the state object implementation in Figure 5 in some more detail. The state contains two properties, namely rows and fields. The rows is used for tables and maps table names to an ordered sequence of Uid's (and an empty sequence by default). The fields stores all the cloud values for both tables and indices, and maps any triple of a record identifier, field name, and field type ($\mathsf{Rid} \times \mathsf{Fname} \times \mathsf{Ftype}$) to either undefined or a cloud value.

The read method defines how Read operations are handled. If it is a field read (fread(rid,fname,ftype)) the value is read from the fields. If the value is undefined, we return the default value of that cloud type. This is very important: we never store default values and this allows us to efficiently represent for example 'infinite' indices in bounded storage.

The update method defines Update operations. There are two assertions that require update sequences to be *well-formed*. In particular:

A1. When creating a new row using new(uid,tname), the uid must be fresh in the sense that it has not been used before in the update sequence.

A2. For any field update update(rid,fname,ftype,fop), the operation fop must be a valid operation for the field type ftype.

Set	Variable		Definition	Meaning
Val	v	=	$n \mid b \mid s$	value
$\mathsf{Val}_{\mathsf{nr}}$	n	=	$0 \mid \ldots$	number
$\mathsf{Val}_{\mathsf{bool}}$	b	=	false | true	boolean
$\mathsf{Val}_{\mathsf{str}}$	s	=	"" $\mid \ldots$	string
Uid	*uid*	=	$\ldots$	unique identifier
Fname	*fname*	=	$\ldots$	field name
Tname	*tname*	=	$\ldots$	table name
Iname	*iname*	=	$\ldots$	index name
Ftype	*ftype*	=	nr | bool | str	field types
$\mathsf{Fop}_{\mathsf{nr}}$	fop_{nr}	=	$\mathsf{set}(n) \mid \mathsf{add}(n)$	number field updates
$\mathsf{Fop}_{\mathsf{str}}$	fop_{str}	=	$\mathsf{set}(s) \mid \mathsf{setifempty}(s)$	string field updates
$\mathsf{Fop}_{\mathsf{bool}}$	fop_{bool}	=	$\mathsf{set}(b)$	boolean field updates
Rid	*rid*	=		record identifier
			$\mid$ *tname*(*uid*)	table row
			$\mid$ *iname*$[key_1, \ldots, key_n]$	index entry
Key	*key*	=	*uid* $\mid s \mid n \mid b$	index key
Update	*upd*	=		update operation
			| clr	clear all data
			| new(*uid*, *tname*)	create table row
			| del *uid*	delete table row
			| *rid*.*fname*.*ftype*.fop_{ftype}	field update
Read	*rd*	=		read operation
			| rows *tname*	enumerate rows
			| fread *rid*.*fname*.*ftype*	field read

Figure 4. The syntax of the data model.

Generally, it is straightforward to ensure that clients can only generate well-formed update sequences.

In the update method, the clr operations simply resets the rows and fields. The new operation is interesting since it takes a unique identifier as an argument where it is asserted that this uid is indeed not used already. Having a uid as an argument allows each client to generate unique identifiers locally without synchronization with the server which is crucial for disconnected operation for example.

The del(uid) operation deletes a particular row in a table and the corresponding fields in that table. Note though that the expression key.contains(uid) deletes both fields in the table row (indexed by tname(uid)) *and* any fields that happen to be indexed by that uid of the form iname[...,uid,...].

The update(rid,fname,ftype,fop) operation is the most interesting and performs an update operation on a particular field. First there is a check that the particular rid does not refer to a record that has been deleted already (perhaps by some other concurrent client). If there is any uid in the rid, either of the form tname(uid) or iname[...,uid,...], where the uid is not in the rows, then we return immediately as this update is now a no-op. This is important for the optimality of storage: if we allow the update to happen on a deleted entry, this may result in a non-default value which would take up storage space. After this check, we simply read the current value, apply the operation, and write back the new value. Again, if the new value happens to be the default value of that type, we remove the field to minimize storage requirements.

4. Robust Streaming

We now describe a streaming server-client implementation model. It is observationally equivalent to the abstract model from § 2. However, it models communication and failures realistically, and eliminates the unbounded logs present in the abstract model.

- Connections are represented as stream pairs (sockets). Clients stream their updates to the server, and the server streams updates back to all connected clients.
- Channels can fail at any time, and on any end, without disrupting the execution of the server or clients. In particular, user code can always read, update, and yield, without blocking, regardless of connectivity. The flush operation may of course block if disconnected.
- The server may crash and recover, losing soft state in the process, but preserving persistent state. The persistent server state contains a snapshot of the current state of the

```
class State
{
  rows : Tname → Uid* = {};
  fields : Rid × Fname × Ftype ⇀ Val = {};
  function size() { return rows.count + fields.count; }
  method read(r : Read) {
    match(r) with {
      fread(rid,fname,ftype) → {
        var val = fields[(rid,fname,ftype)];
        return (val == undefined) ? defaultval(ftype) : val;
      }
      rows(tname) → return rows[tname];
  } }
  method update(u: Update) {
    match(u) with {
      clr() → rows = fields = {};
      new(uid,tname) → {
        assert(! used.contains(uid)); //A1
        rows[tname].append(uid);
      }
      del(uid) → {
        foreach (tn in rows.keys)
          if (rows[tn].contains(uid)) rows[tn].remove(uid);
        foreach (key in fields.keys)
          if (key.contains(uid)) fields.remove(key);
      }
      update(rid, fname, ftype, fop) → {
        assert(fop in Fop_tname); //A2
        if (exists uid in rid : !rows.contains(uid))
          return; // update on nonexisting record is noop
        var curval = read(rid,fname,ftype);
        var newval = match (fop) with {
          set(v) → v
          add(n) → curval + n
          setifempty(s) → (curval = "" ? s : curval)
        }
        if (newval = defaultval(ftype))
          fields.remove((rid,fname,ftype));
        else
          fields[(rid,fname,ftype)] = newval;
    } }
    foreach(uid in u)
      used.append(uid); //track uids to detect freshness violations
  }
  method targets_deleted_data(u : Update): boolean {
    return exists uid in u : !rows.contains(uid);
  }
}
```

Figure 5. The semantics of the data model.

data and the number of the last round committed by each client. Noteably, it does *not* store a log of operations as in the reference model.

- Clients may crash silently or temporarily stop executing for an unbounded amount of time, yet are always able to reconnect. In particular, there are no timeouts. Permanent failures of clients cannot disrupt the server, other clients, or the consistency guarantees.
- Even when clients operate offline forever, their operation logs do not grow without bounds, but use optimal delta reduction to minimize storage requirements. In particular, regardless of the operations performed on the client, the stored delta is no larger than the sum of the current data size and the data size at the time they were last connected.

States and Deltas. The streaming model does not store any update sequences. Instead, it eagerly reduces such sequences, and stores them in a special reduced data format, either as *state objects* (if the sequence represents the tail of the log) or as *delta objects* (if the sequence represents some segment of the log). We present the delta object implementation (and show correctness and optimality) in Section 5.1. For now, we just use the following abstract functions:

```
const initialstate : State
function read: Read x State → Value
function apply : State x Delta* ⇀ State
const emptydelta: Delta
function append : Delta x Update ⇀ Delta
function reduce : Delta* ⇀ Delta
```

Note that some of these functions are partial because they require freshness of unique identifiers and correctly typed operations on fields.

Rounds. As before, we use rounds to encapsulate update sequences. This time, however, we store the updates in reduced form, using delta objects.

```
struct Round {client: Client; number: ℕ;delta: Delta; }
```

Segments. We store pieces of the log in objects called *segments*. They store a state or delta object, and a partial map maxround that records the maximal client round of each client that is represented in segment.

```
class LogTail {
   maxround: (Client ⇀ ℕ) := {};
   state: State := initialstate;
   method apply(s: LogSegment) {
      foreach((c,r) in s.maxround)
         maxround[c] := r;
      state := apply(state, s.delta);
} }
class LogSegment {
   maxround: (Client ⇀ ℕ) := {};
   delta: Delta := emptydelta;
   method append(r: Round):void {
```

```
requests: set of Channel := {};
class Channel {
    client: Client;
    serverstream: (LogTail | LogSegment)* = [];
    clientstream: Round* = [];
    Channel(c: Client) { client = c; }
}
internal request_is_lost(ch: Channel) {
    requests.remove(ch);
}
internal channel_fails_at_server(ch: Channel) {
    connections.remove(ch);
}
internal channel_fails_at_client(c:Client) {
    c.channel = null;
}
```

Figure 6. State and Transitions of Network.

```
        maxround[r.client] := r.number;
        delta := reduce(delta  · r.delta);
} }
```

Network (Fig. 6). We model the network state as a set request of pending connection requests, and channel objects representing connections. Channels are created by some specific client, and contain two streams, one for each direction. The client sends its sequence of rounds, and the server sends the log in reduced pieces (always starting with the tail, followed by consecutive segments). Server and clients can only access their respective ends of each stream, and do not communicate in any other form. We explicitly model network failures using the three transitions shown. Channels guarantee reliable in-order delivery (as provided by TCP sockets, for example), but can fail themselves, and do so non-atomically at either end.

Server (Fig. 7). The server state is separated into persistent state (serverstate), which stores the current state and the number of the last round of each client that has been incorporated into the state, and soft state (connections) which stores currently active connections and which gets obliterated during the crash_recovery transition. A connection is started by the accept_connection transition, which removes a channel from the set of requests and adds it to the active connections connections, replacing any prior connection by the same client. It then sends the current state on the channel.

During normal operation, the server repeatedly performs the processbatch operation. It combines a nondeterministic number of rounds from each active connection into a single segment (which stores all updates in reduced form as a delta object). It then appends this segment to the persistent state (which applies the delta to the current state, and updates the maximum round number per client), and sends it out on all active channels.

```
// persistent state
serverstate: LogTail := new LogTail();
// soft state
connections: (Client ⇀ Channel) := {};
internal crash_recovery() { connections := {}; }
internal accept_connection(ch: Channel) {
  requires ch in requests;
  requests := requests.remove(ch);
  connections[ch.client] := ch; // may replace prior connection
  ch.serverstream.append(serverstate); // send current state
}
internal processbatch() {
    var s = new LogSegment();
    // append incoming segments to s
    foreach((c,ch) in connections)
        receive(s, ch, *);
    // atomically commit changes to persistent state
    serverstate.apply(s);
    // notify connected clients
    foreach((c,ch) in connections)
        ch.serverstream.append(s);
}
procedure receive(s: LogSegment, ch:Channel, count: int) {
    requires count <= ch.clientstream.length;
    foreach(r in ch.clientstream[0..count])
        s.append(r);
    ch.clientstream := ch.clientstream[count..];
}
```

Figure 7. State and Transitions of Server.

Client (Fig. 8). Clients store the known prefix of the server log as a state object in base. Of the local rounds, it stores only the unconfirmed ones and the current one. Just as in GLUT, (1) updates by the user code are added to the current round, (2) reads by the user code see a composite of the last known prefix, uncommitted rounds, and the current round, (3) yields commit the current round and grow the known prefix. However, we now model asynchronous communication explicitly.

Channel setup happens in stages. send_connection_request can fire at any point, independent of the execution of the client program, and creates a new channel (replacing the previously stored c.channel), and sends a connection request to the server (modeled by adding the channel to the requests set). When the server accepts, it sends its current state, containing a snapshot of the data and the number of the last round committed. During a yield or flush_yield, clients may receive this information (first branch of conditional in receive). The client then replaces its base state with the sent server state, and determines which of their uncommitted rounds have made it to the server by inspecting the maxround entry, and removes those. Then, the client resends all the remaining unconfirmed rounds, which is important to recover correctly from failure of earlier channels. Now, the channel is established.

```
class Client {
    base: State := emptystate;
    unconfirmed: Round* := [];
    current: Round := new Round(this, 1, emptydelta);
    channel: Channel := null;
    received1stpacket: bool := false;
    flushing: bool := false;
}
function curstate(c:Client): State {
    return apply(base, deltas(unconfirmed · current)) }
function deltas(seq: Round*): Delta {
    return seq[0].delta ··· seq[seq.length1].delta; }
function connection_established() {
    return c.channel != null ∧ c.received1stpacket; }
external client_update(c: Client, u: Update) {
    requires !c.flushing;
    if (! curstate(c).targets_deleted_data(u))
        c.current.delta := c.current.delta.append(u);
}
external client_read(c: Client, r: Read) {
    requires !c.flushing;
    return read(r, curstate(c));
}
external yield(c: Client) {
    requires !c.flushing;
    if (*) commit_current(c);
    while(*) receive(c);
}
internal send_connection_request(c: Client) {
    c.channel := new Channel(c); // may replace prior channel
    c.received1stpacket := false;
    requests := requests + c.channel;
}
procedure commit_current(c:Client) {
    requires connection_established(c);
    c.channel.clientstream.append(c.current);
    c.unconfirmed.append(c.current);
    c.current := new Round(c, c.current.number + 1, emptydelta);
}
procedure receive(c: Client) {
    requires c != null ∧ c.channel.serverstream.length > 0;
    var s := c.channel.serverstream[0];
    c.channel.serverstream := c.channel.serverstream[1..];
    if (! c.received1stpacket) {
          c.received1stpacket := true;
          assert(s instanceof LogTail);
          c.base := s.state; // replace base state
          adjust_confirmed(c, s.maxround[this]);
          // resubmit unconfirmed rounds
          c.channel.clientstream.append(c.unconfirmed);
    } else {
        assert(s instanceof LogSegment);
        c.base.append(s);
        adjust_confirmed(c, s.maxround[this]);
} }
procedure adjust_confirmed(c: Client, upto: ℕ) {
    while (upto ⩾ c.unconfirmed[0].number)
        c.unconfirmed := c.unconfirmed[1..];
}
```

Figure 8. State and transitions of Client.

```
external flush_begin(c: Client)
{
    requires !c.flushing;
    c.flushing := true;
}
internal flush_yield(c: Client)
    requires c.flushing;
    if (*) commit_current(c);
    while(*) receive(c);
}
external flush_end(c: Client)
{
    requires c.flushing;
    requires c.committed = [];
    c.flushing := false;
}
```

Figure 9. Flush transitions on streaming client.

Once the channel is established, and the client receives deltas (second branch of conditional in receive), it applies those to the base state, and drains the unconfirmed updates based on the information received from the server. To commit the current round in commit_current, the precondition states that there must be an established channel — otherwise we keep appending updates to the current round. This precondition is important to ensure that clients resume sending the correct round after a previous channel failed: only after establishing a new channel can we be sure exactly which updates made it all the way into the server state. The freshly committed round is immediately sent on the channel.

The flush transitions (Fig.9) use a slightly different (but equivalent) condition for ending the flush than the GLUT ones (Fig. 3): it ends when there are no unconfirmed rounds left. To indicate that a flush is in progress, it stores a boolean, not a number.

In the update transition, we perform an extra check to see if the update targets data that is already deleted, and skip it if that is the case. This check is not needed for correctness (the update will have no observable effect either way), but is needed to ensure optimal delta reduction, as we will explain in Section 5.2.

Omitted optimizations. In our prototype we implemented a few additional optimizations left out here for simplicity. They include:

- The server caches recent deltas. When clients reconnect, and the server still has the relevant deltas in its cache, the server sends only the deltas needed instead of the whole state.
- The server, when sending segments to a client c, includes not the whole maxround, but only maxround[c].
- As written, reads are highly inefficient, thus some optimizations are required. Our implementation stores an object for each field and row, and keeps relevant updates for

that object stored in the object (to avoid traversing large logs just to look up the current state). Also, we cache the result of expensive reads, such as table enumerations.

5. Optimal Delta Reduction

In any system that tracks sequences of updates, it is important to keep the length of such sequences under control. In particular, update sequences often exhibit redundancy; for example, if the same field is assigned a new value several times, it is sufficient to store and propagate the last update only. In this section, we show how to optimally reduce update sequences. This is not an easy problem, since it requires us to thoroughly remove deleted data without compromising the semantics. Many systems in practice take the easy route and use tombstones for deleted data, and are thus not optimal.

We formally express the log reduction concept by defining a reduction relation $w \rhd w'$ on update sequences; it captures whether we can replace an update sequence w with another sequence w' without any observable effect.

DEFINITION 1. *Given an update sequence* $w_1 \in \mathsf{Update}^*$ *and an update sequence or undefined value* $w_2 \in (\mathsf{Update}^* \cup \bot)$*, we say that* $w_1 \rhd w_2$ *(read:* w_1 *may reduce to* w_2*) iff for all* $r \in \mathsf{Read}$ *and* $a, b \in \mathsf{Update}^*$ *such that* $\mathsf{rvalue}(a \cdot w_1 \cdot b, r) \neq \bot$*, we have* $\mathsf{rvalue}(a \cdot w_1 \cdot b, r) = \mathsf{rvalue}(a \cdot w_2 \cdot b, r)$*. For notational convenience, we use* $w_2 \lhd w_1$ *(read:* w_2 *may replace* w_1*) interchangeably with* $w_1 \rhd w_2$*.*

For example, we can prove that deletion is idempotent, i.e. for any $i \in \mathsf{Uid}$, we have $\mathsf{del}\ i \cdot \mathsf{del}\ i \rhd \mathsf{del}\ i$. As another example, it is easy to see that $\mathsf{new}(i, a) \cdot \mathsf{new}(i, a) \rhd \bot$ because $\mathsf{rvalue}(a \cdot \mathsf{new}(i, a) \cdot \mathsf{new}(i, a) \cdot b, r) = \bot$ for all r, a, b (because the update sequence is not well-formed).

Note that the reduction relation is reflexive ($w \rhd w$), transitive ($w_1 \rhd w_2 \wedge w_2 \rhd w_3 \Rightarrow w_1 \rhd w_3$), and congruent ($w_1 \rhd w_2 \Rightarrow \forall a, b : a w_1 b \rhd a w_2 b$). However, it is not symmetric, because the right-hand side may cause fewer well-formedness violations than the left-hand side. For example, although $\mathsf{new}(i, a) \cdot \mathsf{del}\ i \rhd []$ (we prove this below), the converse $[] \rhd \mathsf{new}(i, a) \cdot \mathsf{del}\ i$ is not true: $\mathsf{rvalue}(\mathsf{new}(i, a), \mathsf{rows}\ a) = i$ but $\mathsf{rvalue}(\mathsf{new}(i, a) \cdot \mathsf{new}(i, a) \cdot \mathsf{del}\ i, \mathsf{rows}\ a) = \bot$.

Although reduction presents a great opportunity for saving space and reducing network consumption, many such reductions are possible, and it is not immediately clear what reductions to apply, in what order, and to what effect. The following definition sheds light on what we expect from a good reduction function.

DEFINITION 2. *Let* $\mathsf{reduce} : \mathsf{Update}^* \rightharpoonup \mathsf{Update}^*$ *be a partial function on update sequences. For a a subset* $W \subseteq \mathsf{Update}^*$ *of update sequences, we say*

- reduce *is a* correct reduction function *on* W *if, for all* $w \in W$*, we have* $w \rhd \mathsf{reduce}(w)$*.*
- reduce *is* optimal reduction function *on* W *if, for all* $w_1 \in W$ *and* $w_1 \in \mathsf{Update}^*$ *such that* $w_1 \rhd w_2$*, we have* $|\mathsf{reduce}(w_1)| \leq |w_2|$*.*

Optimality implies that when clients operate offline, the delta grows only as much as needed to accommodate the data. For example, even if clients repeatedly update the same location, or create and then delete many objects, the delta stays the same size. We can understand this phenomenon as follows: since the minimal number of updates needed to get from database state last to database state current is bounded by last.size() + current.size(), we know that the delta is never larger than the size of the snapshot of the database when last connected, plus the current size of the database.

In the remainder of this section, we describe our reduction implementation, and then prove that it is (1) correct on all update sequences, and (2) optimal for NDU-free sequences (defined below), which include all sequences occuring in executions of our streaming model.

DEFINITION 3. *A sequence of updates* $w \in \mathsf{Update}^*$ *is called NDU-free if it does not contain a subsequence of the form* $\mathsf{new}(i, a) \cdot \mathsf{del}\ i \cdot m$ *where* $i \in \mathsf{Uid}$ *and* $a \in \mathsf{Tname}$*, and where* $m \in \mathsf{Update}$ *is an update that contains* i*.*

5.1 Delta Objects

Our reduction implementation consists of a Delta class that stores update sequences in reduced form and allows us to efficiently append and reduce updates individually. It has the following abstract signature:

```
type Delta = ...
const emptydelta : Delta
function append : Delta × Update ⇀ Delta
function seq : Delta → Update*
```

The implementation is shown in Fig. 10. Note that the append function is partial because some updates may cause an assertion violation (creating a row with a *uid* that is already in use triggers assertion A3, and using an operation that is of the wrong type triggers assertion A4). The last function above, seq, reads back the reduced sequence from the delta.

5.2 Correctness and Optimality

THEOREM 4. *[Optimal Delta Reduction] The following reduction function is correct for all* w*, and is optimal for all NDU-free well-formed* w*:*

```
function reduce(w: Delta*): Delta* {
  var d = new Delta();
  foreach(u in w)
    d.append(u);
  return d.seq();
}
```

A complete proof is included in appendix A. Note that without NDU-freedom, reduce is not guaranteed to be opti-

```
class Delta {
  cleared : boolean;
  deleted : Uid*;
  created : (Uid × Tname)*;
  updated : (Rid × Fname × Ftype) ⇀ Fop;
  Delta() { cleared := false; deleted := [];
            created := []; updated := {}; }
  function seq(): Update* {
    return (cleared ? [clr] : [])
           · deleted.map((uid) ⇒ del(uid))
           · created.map((uid,tname) ⇒ new(uid,tname))
           · updated.map(((r,f,t),o) ⇒ fupd(r,f,t,o));
  }
  method append(u: Update) {
    match(u) with {
      clr() → { deleted := created := [];
                updated := {}; cleared := true; }
      new(uid,tname) → {
        assert(uid does not appear in
                deleted, created, or updated); //A3
        created.append((uid,tname));
      }
      del(uid) → {
        if (!deleted.contains(uid)) {
          if (exists tname : created.contains(uid,tname))
            created.remove((uid,tname));
          else if (!cleared)
            deleted.add(uid);
          foreach (key in updated.keys)
            if (key.contains(uid))
              updated.remove(key);
      } }
      update(rid, fname, ftype, fop) → {
        assert(fop in Fop_tname);//A4
        if (fop = add(0) ∨ fop = setifempty(0)
              ∨ exists uid in deleted s.t. uid occurs in rid)
          return; // update has no effect
        var op := match (fop) with {
          set(v) → set(v); // last writer wins
          add(n)) → {
            match updated[rid,fname,ftype] with {
              undefined → add(n);
              add(m) → add(m+n);
              set(m) → set(m+n);
          } }
          setifempty(s)) {
            match updated[rid,fname,ftype] with {
              undefined → setifempty(s);
              set("") → set(s); // succeed
              set(v) → set(v); // fail
              setifempty(v) → setifempty(v); // fail
        } } }
        updated[rid,fname,ftype] :=
          (op = add(0) ∨ op = setifempty(0)) ? undefined : op;
} } } }
```

Figure 10. Delta implementation.

mal. For example,

$$\mathsf{new}(i,a)\cdot\mathsf{del}\ i\cdot\mathsf{del}\ i \quad \triangleright \quad \epsilon,$$

but because reduce always proceeds from left to right, it does not produce the optimal result:

$$\begin{aligned}&\mathsf{reduce}(\mathsf{new}(i,a)\cdot\mathsf{del}\ i\cdot\mathsf{del}\ i)\\ =\ &\mathsf{reduce}(\mathsf{reduce}(\mathsf{new}(i,a)\cdot\mathsf{del}\ i)\cdot\mathsf{del}\ i)\\ =\ &\mathsf{reduce}(\epsilon\cdot\mathsf{del}\ i)=\mathsf{del}\ i\neq\epsilon\end{aligned}$$

We now show that all delta sequences appearing in executions of the streaming model are NDU-free, which concludes our optimality proof.

Proof of NDU-freedom. We need to examine the two places where the Delta.append method is called: (1) when processing an update u by the user code. In this case, the transition calls targets_deleted_data(u) and skips the update if it contains an update that contains a deleted uid. Thus current.delta is always NDU-free, and therefore all the deltas in all the rounds. Note that once a client deletes a row, it will forever appear deleted to this client; therefore, this check enforces that a delete of some row by some client can never be followed by an update by the same client that contains that same row. (2) When the server creates a new segment and adds multiple client rounds. If one of the rounds contains a $\mathsf{new}(i,a)$, then there cannot be any $\mathsf{del}\ i$ except by the same client, because the new is not visible to other clients yet. However, then there cannot be an update containing i by the same client, for the reason just explained.

6. Implementation in TouchDevelop

We have realized the ideas presented in this paper and their implementation in a web-based IDE and runtime system called TouchDevelop [27]. The language supports tables, indices, and all the operations described in this paper. Additionally, there is support for 1) linking rows to other table rows, thus enabling cascading deletes, 3) data session management, and 4) automatic UI updates when shared data changes. We implemented the streaming model using a Azure cloud service backed by Azure table storage (for the persistent state).

The three versions of the bird log example of Section 3 are available within TouchDevelop: Version 1 (`http://tdev.ly/ubbh`) is the initial bird log version with both the read-increment-set, and the double-create problem. Version 2 (`http://tdev.ly//zalu`) fixes first problem by using the add operation on cloud numbers, whereas Version 3 (`http://tdev.ly/ljjb`) additionally fixes the double-create problem by using an index instead of a table. The reader can experiment with the implementation from any modern web browser, on any device (touchscreen or not). To see and edit the code, follow the "more info / tweak it" link.

For illustration, we give a few examples of apps that use cloud types below: feel free to run them, inspect and edit their code, and create your own variations! The first two examples below have been contributed by our user community. In all of these examples, the use of cloud types is very simple, with the exception of the Cloud Game Selector which involves tricky synchronization and a flush operation.

- **Relatd** [sic] (`http://tdev.ly/ruef`) Lets users enter their qualities (either from a list, or freely entered) and finds and displays other users that share them.
- **Chatter Box** (`http://tdev.ly/spji`) A chat application.
- **TouchDevelop Jr.** (`http://tdev.ly/vkrpa`) Program a tiny robot using a simple language, then share your scripts with other users.
- **Instant Poll** (`http://tdev.ly/nggfa`) An app for quickly polling an audience and displaying the responses as a grid of colors.
- **Expense Recorder** (`http://tdev.ly/nvoha`) Allows easy recording of expenses in a table.
- **Contest Voting** (`http://tdev.ly/etww`) Used to determine the winner of the "Touch of Summer" coding contest.
- **Cloud List** (`http://tdev.ly/blqz`) A general-purpose list that can be concurrently edited.
- **Cloud Game Selector** (`http://tdev.ly/nvjh`) A library for matching multiple players to play games together.
- **Cloud Paper Scissors** (`http://tdev.ly/sxjua`) A simple rock/paper/scissors game that uses the cloud game selector library.

We have also experimented with refactoring non-cloud data in TD scripts into cloud types. Our formative study shows that refactoring is applicable, relevant, and saves human effort [15].

7. Related Work

As discussed in the introduction, we build on previous work on cloud types [6] by giving an operational reference model, a data model for optimal delta reduction, and a robust streaming implementation. Our types and primitives are almost identical (except that we have omitted cascading deletes for now).

Replicated Data Types. Most closely related are replicated data types [7, 21, 21] and Bayou's weakly consistent replication [26]. Replicated data types are similar to our cloud types, but do not easily compose (because the consistency protocols are defined per-object, not per-database), and generally provide weaker guarantees (in particular, no update transactions), and no way to recover sequential consistency (as with flush). In Bayou [26], and in the original Concurrent Revisions work [9], conflict resolution is achieved by explicit merge functions written by the user. In contrast, cloud types use conflict resolution that is *type-directed*.

Operational Transformations. The Jupiter system [18] has a similar system structure (client-server with bidirectional streaming). However, it uses *operational transformations* (OT) to reconcile conflicting updates, instead of the simple sequentialization in our data model. Unlike cloud types, OT specializes on collaborative editing of text sequences, and conflict resolution that is appropriate for that context, rather than on structured application data organized as tables, indexes, or sets. OT algorithms [10, 22–24] require more computation (quadratic in the number of concurrent updates) and are less robust (clients cannot disconnect for unbounded periods, then reconnect) than GLUT. Also, there is no optimal reduction in OT, thus operation logs accumulate.

Eventual consistency. EC is motivated by the impossibility of achieving strong consistency, availability, and partition tolerance at the same time, as stated by the CAP theorem [14]. EC across the literature uses a variety of techniques to propagate updates (e.g. general causally-ordered broadcast [21, 23]). For a general high-level comparison of eventual consistency notions appearing in the literature, see [3, 8]. The GLUT system can also be understood as an example of a reliable total order broadcast [12]. However, it goes beyond broadcast because it includes a data model and optimal reduction for the updates.

8. Conclusion

Our work shows that cloud types and update transactions can be both (1) abstractly understood and programmed against, and (2) efficiently and robustly implemented. We hope it provides a path for simpler development of distributed applications, in particular for social or collaborative scenarios. In the future, we would like to add more data types (such as lists), gather more experience and feedback using our deployed prototype, and provide mechanically verified correctness proofs for our models.

References

[1] IEEE Computer CAP retrospective edition. *Computer*, 45(2), 2012.

[2] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. In *International Conference on Very Large Databases (VLDB)*, 2014.

[3] P. Bernstein and S. Das. Rethinking eventual consistency. In *SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 923–928. ACM, 2013.

[4] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC'00*.

[5] S. Burckhardt, M. Fähndrich, D. Leijen, and M. Sagiv. Eventually Consistent Transactions. In *European Symposium on Programming (ESOP), (extended version available as Microsoft Tech Report MSR-TR-2011-117), LNCS*, volume 7211, pages 64–83, 2012.

[6] S. Burckhardt, M. Fähndrich, D. Leijen, and B. Wood. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 7313 of *LNCS*, pages 283–307. Springer, 2012.

[7] S. Burckhardt, A. Gotsman, H.Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *Principles of Programming Languages (POPL)*, 2014.

[8] S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, Microsoft, 2013.

[9] S. Burckhardt and D. Leijen. Semantics of Concurrent Revisions. In *European Symposium on Programming (ESOP), LNCS*, volume 6602, pages 116–135, 2011.

[10] M. Cart and J. Ferrie. Asynchronous reconciliation based on operational transformation for p2p collaborative environments. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2007)*, pages 127–138, Nov 2007.

[11] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Symposium on Operating Systems Principles*, pages 205–220, 2007.

[12] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, Dec. 2004.

[13] D. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

[14] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[15] M. Hilton, A. Christi, D. Dig, M. Moskal, S. Burckhardt, and N. Tillmann. Refactoring local to cloud data types for mobile apps. In *MobileSoft '14*. ACM, 2014.

[16] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. 2002.

[17] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP'11*.

[18] D. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *User interface and software technology (UIST)*, 1995.

[19] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 34–41, 1995.

[20] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report Rapport de recherche 7506, INRIA, 2011.

[21] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Grenoble, France, Oct. 2011.

[22] M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Conference on Supporting Group Work*, GROUP '97, pages 435–445. ACM.

[23] C. Sun and C. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Computer Supported Cooperative Work*, CSCW '98, pages 59–68. ACM, 1998.

[24] D. Sun and C. Sun. Operation context and context-based operational transformation. In *Conference on Computer Supported Cooperative Work*, CSCW '06, pages 279–288. ACM, 2006.

[25] D. Terry, A. Demers, K. Petersen, M. S. M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[26] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29:172–182, December 1995.

[27] N. Tillmann, M. Moskal, J. de Halleux, and M. Fähndrich. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *ONWARD '11 at SPLASH (also available as Microsoft TechReport MSR-TR-2011-49)*, 2011.

[28] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

A. Proof of Theorem 4

We now prove this theorem. We start with a few basic invariants that are useful later on. Then we show optimality (§A.1), and finally correctness (§A.2).

LEMMA 5 (Delta invariants). *All $d \in \Delta$ satisfy the following conditions:*

(Δ1) *Each $i \in$ Uid occurs at most once within d.*created *and d.*deleted.

(Δ2) *If i appears in d.*deleted*, then it does not appear in d.*updated.

(Δ3) *If $d.\mathsf{updated}(k, f, t) = o$ with $o \neq \bot$, then $o \in \mathsf{Fop}_t$.*

(Δ4) *If $d.\mathsf{updated}(k, f, t) = o$, then $o \neq \mathit{add}(0)$ and $o \neq \mathit{setifempty}(\text{""})$.*

(Δ5) *If $d.$*cleared *then $d.$*deleted $= []$.

PROOF. By induction. It is easy to verify that all conditions are true on emptydelta, and remain true under any append. □

A.1 Optimality Proof

To prove optimality, we need to show that if w_1 is well-formed and NDU-free, and if $w_1 \rhd w_2$, then $\mathsf{reduce}(w_1)$ is defined and $|\mathsf{reduce}(w_1)| \leq |w_2|$. First, we argue that $\mathsf{reduce}(w_1) \neq \bot$, because otherwise it would have to trigger an assertion (A3 / A4 in Fig. 10). But if w_1 triggered A3 (A4) it would have to also trigger A1 / A2 in Fig. 5, contradicting well-formedness. Second, we prove two lemmas (6, 7) that together imply that w_2 can be no shorter than $\mathsf{reduce}(w_1)$.

LEMMA 6. *Let $w_1 \in \mathsf{Update}^*$ be an update sequence. Then:*

(i) $\mathsf{reduce}(w_1)$ *contains at most one occurrence of* clr.

(ii) $\mathsf{reduce}(w_1)$ *contains at most one occurrence of* del *i for each i.*

(iii) $\mathsf{reduce}(w_1)$ *contains at most one occurrence of* new$(i, _)$ *for each i.*

(iv) $\mathsf{reduce}(w_1)$ *contains at most one occurrence of $r.f.t.\_$ for each (r, f, t)*

PROOF. (i) is obvious since we store a simple boolean to record whether clr happened. (ii) is guaranteed because we explicitly check for duplicates before recording deletions. (iii) is guaranteed by assertion A3. (iv) is guaranteed because we store field updates in a partial map, keyed by (r, f, t). □

LEMMA 7. *Let w_1 be well-formed and NDU-free, and let $w_1 \rhd w_2$. Then:*

(i) If $\mathsf{reduce}(w_1)$ *contains* clr*, then w_2 contains* clr.

(ii) If $\mathsf{reduce}(w_1)$ *contains* del *i, then w_2 contains* del *i.*

(iii) If $\mathsf{reduce}(w_1)$ *contains* new(i, a)*, then w_2 contains* new(i, a).

(iv) If $\mathsf{reduce}(w_1)$ *contains $r.f.t.o$, then w_2 contains $r.f.t.o'$ for some o'.*

PROOF. For each claim, we proceed indirectly: assuming the claim is false, we find r and w such that $w \cdot w_1$ is well-formed (which implies $\mathsf{rvalue}(w \cdot w_1, r) = \mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), r)$) but for which $\mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), r) \neq \mathsf{rvalue}(w \cdot w_2, r)$ which then contradicts $w_1 \rhd w_2$.

(i) Let $w = \mathsf{new}(i_1, a) \cdots \mathsf{new}(i_n, a)$ where the i_j are pairwise distinct and do not appear in w_1, and where $n > |w_2| + |\mathsf{reduce}(w_1)|$. Since clr is the only update that can remove more than one row at a time, and is not contained in w_2, $|\mathsf{rvalue}(w \cdot w_2, \mathsf{rows}\ a)| \geq n - |w_2| > |\mathsf{reduce}(w_1)|$. Since clr is contained in $\mathsf{reduce}(w_1)$ and no update can add more than one row at a time, $|\mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{rows}\ a)| < |\mathsf{reduce}(w_1)|$. Thus, $\mathsf{rvalue}(w \cdot w_2, \mathsf{rows}\ a) \neq \mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{rows}\ a)$.

(ii) Since del $i \in \mathsf{reduce}(w_1)$, we must have del $i \in w_1$. Distinguish cases (using the first matching case below):

- [clr $\in w_1$ or clr $\in w_2$]. Then clr $\in w_1$ and thus clr $\in \mathsf{reduce}(w_1)$. But by (Δ5) this implies del $i \notin \mathsf{reduce}(w_1)$, contradicting the assumption.
- [$\mathsf{new}(i, a) \in w_1$ for some a]. Then it must occur only once, and before del i in w_1 (otherwise w_1 is not well-formed). Also, there cannot occur a second del i in w_1, because w_1 is assumed NDU-free, nor can there occur an intervening clr in w_1(first case). But this implies that $\mathsf{reduce}(w_1)$ does not contain del i (since $\mathsf{new}(i, a)$ and del i cancel out during reduction), contradicting the assumption.
- [otherwise]. Let $w = \mathsf{new}(i, a)$. Then $w \cdot w_1$ is well-formed (because $\mathsf{new}(i, a) \notin w_1$, which would have been the preceding case). Since del $i \in \mathsf{reduce}(w_1)$, $\mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{rows}\ a) \not\ni i$. But $\mathsf{rvalue}(w \cdot w_2, \mathsf{rows}\ a) \ni i$ because w_2 contains neither clr (first case) nor del i (which we assumed for the purposes of deriving a contradiction to the claim). Thus, $\mathsf{rvalue}(w \cdot w_2, \mathsf{rows}\ a) \neq \mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{rows}\ a)$.

(iii) Let $w = []$. If $\mathsf{new}(i, a) \notin w_2$, $\mathsf{rvalue}(w \cdot w_2, \mathsf{rows}\ a) \not\ni i$, but since $\mathsf{new}(i, a) \in \mathsf{reduce}(w_1)$, $\mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{rows}\ a) \ni i$. Thus, $\mathsf{rvalue}(w \cdot w_2, \mathsf{rows}\ a) \neq \mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{rows}\ a)$.

(iv) Pick w based on the value of o:

1. if $o = \mathsf{set}(v)$ for some v, pick $w = r.f.t.\mathsf{set}(x)$ for some $x \neq v$.
2. if $o = \mathsf{add}(n)$ for some $n \neq 0$, pick $w = r.f.t.\mathsf{set}(0)$
3. if $o = \mathsf{setifempty}(v)$ for some $v \neq \text{""}$, pick $w = r.f.t.\mathsf{set}(\text{""})$

Note that these cases are exhaustive, because $\mathsf{reduce}(w_1)$ contains no other field updates by (Δ4). Also, $w \cdot w_1$ must be well-formed, otherwise w_1 must create an identifier used in r which would contradict the assumption that w_1 is NDU-free. Now, if w_2 contains no updates for (r, f, t), then $\mathsf{rvalue}(w \cdot w_2, \mathsf{fread}\ r.f.t)$ is x or 0 or "", respectively. But on the other hand, $\mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{fread}\ r.f.t)$ is v or n or v, respectively. Thus, $\mathsf{rvalue}(w \cdot w_2, \mathsf{fread}\ r.f.t) \neq \mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{fread}\ r.f.t)$.

□

A.2 Correctness Proof

To get started with proving the correctness of the reduction function as claimed in Thm. 4, we first assemble all the reductions that are needed for the proof, collected in the Lemma below. None of these reductions increase the length of the sequence, and most of them decrease it.

LEMMA 8. *For all* $w \in \mathsf{Update}^*$, $u \in \mathsf{Update}$, $i \in \mathsf{Uid}$, $a \in \mathsf{Tname}$, $k, k' \in \mathsf{Key}$, $f, f' \in \mathsf{Fname}$, $t, t' \in \mathsf{Ftype}$, $o \in \mathsf{Fop}$, $s, s' \in \mathsf{Val}_{str}$, $n, m \in \mathsf{Val}_{nr}$, *the following are true:*

(i). $w \cdot \mathsf{clr} \rhd \mathsf{clr}$
(ii). $w \cdot \mathsf{new}(i,a) \rhd \mathsf{new}(i,a) \cdot w$
(iii). *if* i *occurs in* u *then* $(u \cdot \mathsf{del}\ i \rhd \mathsf{del}\ i)$ *else* $(u \cdot \mathsf{del}\ i \rhd \mathsf{del}\ i \cdot u)$
(iv). $\mathsf{new}(i,a) \cdot \mathsf{del}\ i \rhd []$
(v). $\mathsf{del}\ i \cdot \mathsf{del}\ i \rhd \mathsf{del}\ i$ *and* $\mathsf{clr} \cdot \mathsf{del}\ i \rhd \mathsf{clr}$
(vi). $(i \text{ occurs in } u) \Rightarrow (u \cdot \mathsf{new}(i,a) \rhd \bot)$
(vii). $((k,f,t) \neq (k',f',t')) \Rightarrow (k.f.t.o \cdot k'.f'.t'.o' \rhd k'.f'.t'.o' \cdot k.f.t.o)$
(viii). $k.f.t.o \cdot k.f.t.\mathsf{set}(v) \rhd k.f.t.\mathsf{set}(v)$
(ix). $k.f.t.\mathsf{add}(m) \cdot k.f.t.\mathsf{add}(n) \rhd k.f.t.\mathsf{add}(m+n)$
(x). $k.f.t.\mathsf{set}(m) \cdot k.f.t.\mathsf{add}(n) \rhd k.f.t.\mathsf{set}(m+n)$
(xi). $k.f.t.\mathsf{set}(\text{""}) \cdot k.f.t.\mathsf{setifempty}(s) \rhd k.f.t.\mathsf{set}(s)$
(xii). $(s \neq \text{""}) \Rightarrow (k.f.t.\mathsf{set}(\mathsf{s}) \cdot k.f.t.\mathsf{setifempty}(s') \rhd k.f.t.\mathsf{set}(s))$
(xiii). $(s \neq \text{""}) \Rightarrow (k.f.t.\mathsf{setifempty}(\mathsf{s}) \cdot k.f.t.\mathsf{setifempty}(s') \rhd k.f.t.\mathsf{setifempty}(s))$
(xiv). $k.f.t.\mathsf{add}(0) \rhd []$ *and* $k.f.t.\mathsf{setifempty}(\text{""}) \rhd []$
(xv). $(o \notin \mathsf{Fop}_t) \Rightarrow (k.f.t.o \rhd \bot)$

PROOF. Using Def.1 directly to prove claims of the form $w_1 \rhd w_2$ is unwieldy because of the large number of quantifiers. Instead we use Lemma 9 below, which allows us to check a condition that quantifies over states only. The proofs of the claims are straightforward, we show the first four only.

(i) Let $apply(s, w \cdot \mathsf{clr}) \neq \bot$. Note that (1) clr clears rows and fields, and (2) since used can only grow, $apply(s, w \cdot \mathsf{clr}).\mathsf{used} \supseteq apply(s, \mathsf{clr}).\mathsf{used}$. Thus $apply(s, w \cdot \mathsf{clr}) \sqsupseteq apply(s, \mathsf{clr})$.

(ii) Let $apply(s, w \cdot \mathsf{new}(i,a)) \neq \bot$. Then i cannot appear in neither $s.\mathsf{used}$ nor w. Thus all updates in w commute with $\mathsf{new}(i,a)$.

(iii) If u does not contain i, then u is either clr, del i' with $i' \neq i$, $\mathsf{new}(i',a)$ with $i' \neq i$ or $r.f.t.o$ with $i \notin r$. In all of those cases the delete operation commutes. If u does contain i, it is either (1) del i : see claim (v), or (2) $\mathsf{new}(i,a)$: deleting a nonexisting uid is the same as creating then deleting it, or (3) $r.f.t.o$ with $i \in r$: then the deletion means the update turns into a no-op since it targets a nonexisting record.

(iv) Let $apply(s, \mathsf{new}(i,a) \cdot \mathsf{del}\ i) \neq \bot$. Since $\mathsf{new}(i,a)$ adds a row that gets immediately removed again by del i, rows and fields are the same as if nothing was done at all, but used contains one more element. Thus $apply(s, \mathsf{new}(i,a) \cdot \mathsf{del}\ i) \sqsupseteq apply(s, [])$.

□

LEMMA 9. *Define a binary relation* $\sqsupseteq$ *on states as*

$$(s_1 \sqsupseteq s_2) \Leftrightarrow ((s_1.\mathsf{rows} = s_2.\mathsf{rows}) \wedge (s_1.\mathsf{fields} = s_2.\mathsf{fields}) \wedge (s_1.\mathsf{used} \supseteq s_2.\mathsf{used}))$$

Then the following condition is sufficient to imply $w_1 \rhd w_2$*:*

$$\forall s \in \mathsf{State}: \quad apply(s, w_1) \neq \bot \Rightarrow apply(s, w_1) \sqsupseteq apply(s, w_2).$$

PROOF. It is easy to see that $s_1 \sqsupseteq s_2$ implies both

$$\forall r \in \mathsf{Read}: \quad \mathsf{read}(s_1, r) \neq \bot \Rightarrow \mathsf{read}(s_1, r) = \mathsf{read}(s_2, r)$$

$$\forall b \in \mathsf{Update}^*: \quad \mathsf{apply}(s_1, b) \neq \bot \Rightarrow \mathsf{apply}(s_1, b) \sqsupseteq \mathsf{apply}(s_2, b)$$

from which it is easy to deduce the claim. □

We now proceed to prove the correctness claim in Thm. 4. We show that $\mathsf{reduce}(w) \lhd w$. by induction over the number of elements in w. For $|w| = 0$, the claim is trivially satisfied ($[] \lhd []$). For the induction step, we can assume $\mathsf{reduce}(w) \lhd w$, and we let d be the state of d at the end of the execution of $\mathsf{reduce}(w)$. Then, using (a) Lemma 10 below and (b) the induction hypothesis, we get that for all $u \in \mathsf{Update}$,

$$\mathsf{reduce}(w \cdot u) = d.\mathsf{append}(u).\mathsf{seq}() \lhd^{(a)} d.\mathsf{seq}() \cdot u = \mathsf{reduce}(w) \cdot u \lhd^{(b)} w \cdot u,$$

which concludes the correctness proof.

LEMMA 10. *For all* $d \in \Delta$ *and* $u \in \mathsf{Update}$*, we have* $d.\mathsf{append}(u).\mathsf{seq}() \lhd d.\mathsf{seq}() \cdot u$.

PROOF. We let *lhs* $= d.\mathsf{append}(u).\mathsf{seq}$ and *rhs* $= d.\mathsf{seq} \cdot u$, and thus need to show that *lhs* $\lhd$ *rhs*. We write $d.\mathsf{seq} = C \cdot D \cdot N \cdot U$ where C, D, N, U are sequences of clear, delete, new, and field updates. Now, we do a case distinction (case conditions shown in brackets, applying the first case that matches), and make use of the reductions proved in Lemma 8, labelling $\lhd$ with a subscript indicating the clause used.

- $[u = \mathsf{clr}]$. Then *lhs* $= \mathsf{clr} \lhd_{(i)} C \cdot D \cdot N \cdot U \cdot \mathsf{clr} =$ *rhs*.
- $[u = \mathsf{new}(i,a)]$.

- [i occurs in $D \cdot N \cdot U$]. Then *lhs* $= \bot \lhd_{(vi)} C \cdot D \cdot N \cdot U \cdot \mathsf{new}(i, a) =$ *rhs*.
- [otherwise]. *lhs* $= C \cdot D \cdot N \cdot \mathsf{new}(i, a) \cdot U \lhd_{(ii)} C \cdot D \cdot N \cdot U \cdot \mathsf{new}(i, a) =$ *rhs*.

- [$u = \mathsf{del}\ i$]. Let U', N' be the subsequences of U, N obtained by removing updates containing i.
  - [$D = D_1 \cdot \mathsf{del}\ i \cdot D_2$]. By $\Delta 5$, $C = []$. Then *lhs* $= D \cdot N \cdot U \lhd_{(v)} \cdot D_1 \cdot \mathsf{del}\ i \cdot \mathsf{del}\ i \cdot D_2 \cdot N \cdot U \lhd_{(iii),(\Delta 1),(\Delta 2)} D_1 \cdot \mathsf{del}\ i \cdot D_2 \cdot N \cdot U \cdot \mathsf{del}\ i =$ *rhs*.
  - [$N = N_1 \cdot \mathsf{new}(i, a) \cdot N_2$ for some a]. Then *lhs* $= C \cdot D \cdot N_1 \cdot N_2 \cdot U' \lhd_{(iv)} C \cdot D \cdot N_1 \cdot \mathsf{new}(i, a) \cdot \mathsf{del}\ i \cdot N_2 \cdot U' \lhd_{(iii)} C \cdot D \cdot N_1 \cdot \mathsf{new}(i, a) \cdot N_2 \cdot U \cdot \mathsf{del}\ i =$ *rhs*.
  - [$C = \mathsf{clr}$]. Then *lhs* $= \mathsf{clr} \cdot D \cdot N \cdot U' \lhd_{(v)} \mathsf{clr} \cdot \mathsf{del}\ i \cdot D \cdot N \cdot U' \lhd_{(iii)} C \cdot D \cdot N \cdot U \cdot \mathsf{del}\ i =$ *rhs*.
  - [otherwise]. Then *lhs* $= C \cdot D \cdot \mathsf{del}\ i \cdot N \cdot U' \lhd_{(iii)} C \cdot D \cdot N \cdot U \cdot \mathsf{del}\ i =$ *rhs*.
- [$u = k.f.t.o$]. For convenience, we define a function Φ on field updates as: $\Phi((k'.f'.t'.o')) =$

$$\begin{cases} [] & \text{if } o' = \mathsf{add}(0) \text{ or } o' = \mathsf{setifempty}(0) \\ k'.f'.t'.o' & \text{otherwise} \end{cases}$$

  - [$o \notin \mathsf{Fop}_t$]. Then *lhs* $= \bot \lhd_{(xv)} C \cdot D \cdot N \cdot U \cdot k.f.t.o =$ *rhs*.
  - [$(k, f, t) \notin U$]. Then *lhs* $= C \cdot D \cdot N \cdot U_1 \cdot \Phi(u) \cdot U_2 \lhd_{(vii)} C \cdot D \cdot N \cdot U \cdot \Phi(u) \lhd_{(xiv)}$ *rhs*.
  - [$U = U_1 \cdot u' \cdot U_2$ where $u' = k.f.t.o'$]. Then
    - [$o = \mathsf{add}(0)$ or $o = \mathsf{setifempty}(\text{""})$]. Then *lhs* $= C \cdot D \cdot N \cdot U \lhd_{(xiv)}$ *rhs*.
    - [$o = \mathsf{set}(v)$]. Then *lhs* $= C \cdot D \cdot N \cdot U_1 \cdot u \cdot U_2 \lhd_{(viii)} C \cdot D \cdot N \cdot U_1 \cdot u' \cdot u \cdot U_2 \lhd_{(vii)} C \cdot D \cdot N \cdot U_1 \cdot u' \cdot U_2 \cdot u =$ *rhs*.
    - [$o' = \mathsf{add}(m), o = \mathsf{add}(n)$]. Then *lhs* $= C \cdot D \cdot N \cdot U_1 \cdot \Phi(k.f.t.\mathsf{add}(m+n)) \cdot U_2 \lhd_{(xv)} C \cdot D \cdot N \cdot U_1 \cdot k.f.t.\mathsf{add}(m+n) \cdot U_2 \lhd_{(ix)} C \cdot D \cdot N \cdot U_1 \cdot u' \cdot u \cdot U_2 \lhd_{(vii)} C \cdot D \cdot N \cdot U_1 \cdot u' \cdot U_2 \cdot u =$ *rhs*.
    - [$o' = \mathsf{set}(m), o = \mathsf{add}(n)$]. Analogously, using (x) instead of (ix).
    - [$o' = \mathsf{set}(\text{""})), o = \mathsf{setifempty}(s)$]. Analogously, using (xi).
    - [$o' = \mathsf{set}(v), v \neq \text{""}, o = \mathsf{setifempty}(s)$]. Analogously, using (xii).
    - [$o' = \mathsf{setifempty}(v), o = \mathsf{setifempty}(s)$]. Analogously, using (xiii) and $(\Delta 4)$ (the latter implies $v \neq \text{""}$).

□