Mergeable Replicated Data Types

GOWTHAM KAKI, Purdue University, USA
SWARN PRIYA, Purdue University, USA
KC SIVARAMAKRISHNAN, IIT Madras, India
SURESH JAGANNATHAN, Purdue University, USA

Programming geo-replicated distributed systems is challenging given the complexity of reasoning about different evolving states on different replicas. Existing approaches to this problem impose significant burden on application developers to consider the effect of how operations performed on one replica are witnessed and applied on others. To alleviate these challenges, we present a fundamentally different approach to programming in the presence of replicated state. Our insight is based on the use of *invertible relational specifications* of an inductively-defined data type as a mechanism to capture salient aspects of the data type relevant to how its different instances can be safely merged in a replicated environment. Importantly, because these specifications only address a data type's (static) structural properties, their formulation does not require exposing low-level system-level details concerning asynchrony, replication, visibility, etc. As a consequence, our framework enables the correct-by-construction synthesis of rich merge functions over arbitrarily complex (i.e., composable) data types. We show that the use of a rich relational specification language allows us to extract sufficient conditions to automatically derive merge functions that have meaningful non-trivial convergence properties. We incorporate these ideas in a tool called Quark, and demonstrate its utility via a detailed evaluation study on real-world benchmarks.

1 INTRODUCTION

Modern distributed data-intensive applications often replicate data across geographically diverse locations to (a) enable trust decentralization, (b) guarantee low-latency access to application state, and (c) provide high availability even in the face of node and network failures. There are three basic approaches that have been proposed to program and reason about applications in this setting. The first re-engineers algorithms to be cognizant of replicated behavior. This strategy yields *Replicated Data Types* (RDTs) [Burckhardt et al. 2014; Shapiro et al. 2011a], abstractions that expose the same interface as ordinary (sequential) data types, but whose implementations are aware of replicated state. In some cases, the data type's underlying representation can be defined to guarantee the absence of conflicting updates (e.g., by ensuring its operations are commutative). Otherwise, ensuring convergence of all replicas can be enforced by preemptively avoiding conflicts through selective consistency strengthening [Li et al. 2014a, 2012a]. Correct RDT implementations guarantee that all executions correspond to some linearization of the operations performed on them. A second approach, captured by abstractions like *concurrent revisions* [Burckhardt et al. 2010], admit richer semantics by permitting executions that are not linearizable; these abstractions explicitly expose replicated behavior to clients by defining operations that create and synchronize different versions of object state, where each version captures the evolution of a replicated object as it executes on a different replica. Finally, there have been recent attempts to equip specifications, rather than applications, with mechanisms that characterize notions of correctness in the presence of replication [Houshmand and Lesani 2019; Sivaramakrishnan et al. 2015], using these specifications to guide implementations on when and how different global coordination and synchronization mechanisms should be applied. In all three cases, developers must grapple with various operational

Authors' addresses: Gowtham Kaki, Purdue University, USA; Swarn Priya, Purdue University, USA; KC Sivaramakrishnan, IIT Madras, India; Suresh Jagannathan, Purdue University, USA.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

nuances of replication, either in the way objects are defined, abstractions used, or specifications written. As a result, all three approaches impose significant cognitive burden that complicates reasoning and hinders adoption.

In this paper, we propose a fundamentally different approach to programming with replicated state that enables the *automatic* derivation of correct distributed (replicated) variants of ordinary data types. Key to our approach is the use of *invertible relational specifications* of an inductive data type definition. These specifications capture salient aspects of the data type that are independent of its execution under any system model, thus freeing the programmers from having to explicitly reason about low-level operational issues related to replication, asynchrony, visibility, etc. Their relational structure, however, provides sufficient guidance on structural properties maintained by the type (e.g., element ordering) critical to how we might correctly *merge* multiple instances in a replicated setting.

Thus, like the version-based schemes mentioned above, our approach is also based on a model of replication centered around *versioned states* and explicit *merges*. In particular, we model replicated state in terms of concurrently evolving *versions* of a data type that trace their origin to a common ancestor version. We assume implementations synchronize pairs of replicas by merging concurrent versions into a single convergent version that captures salient characteristics of its parents. The merge operation is further aided by context information provided by the *lowest common ancestor* (LCA) version of the merging versions.

Because the exact semantics of merging depends on the type and structure of replicated state, data types define merge semantics via an explicit merge function. The merge function performs a three-way merge involving a pair of concurrent versions and their LCA version that constitutes the context for the merge. The version control model of replication, therefore, allows any ordinary data type equipped with a three-way merge function to become a distributed data type. The full expressivity of merge functions can be exploited to define bespoke distributed semantics for data types that need not necessarily mirror their sequential behavior (i.e., distributed objects that are not linearizable or serializable), but which are nonetheless well-defined (i.e., convergent) and have clear utility.

Unlike prior approaches, however, which neither provide any guarantees on the correctness of merge operations as they relate to the semantics of the data type over which they are defined nor define a principled methodology for defining such operations over arbitrary types, our focus in this paper is on deriving such correct merge functions automatically over arbitrarily complex (i.e, composable) data type definitions, and in the process, ascribe to them a meaningful and useful distributed semantics. By doing so, we eliminate the need to reason about low-level operational or axiomatic details of replication when transforming sequential data types to their replicated equivalents.

```
module Counter: COUNTER =
struct
  type t = int
  let zero = 0
  let add x v = v + x
  let sub  x v = v - x
  let mult x v = x * v
  let read v = v
end
```

Fig. 1. A Counter data type in OCaml

Our approach towards deriving data type-specific merge functions is informed by two fundamental observations about replicated data type state and its type. First, we note that it is possible to define an intuitive notion of a merge operation on concurrent versions of an abstract object state *regardless* of its type. We illustrate this notion in the context of a simple integer counter, whose OCaml implementation is shown in Fig. 1. Suppose we wish to replicate the state of the counter across multiple machines, each of which is allowed to perform concurrent conflicting updates to its local instance. As long as clients just use the counter's add and sub operations, conflicts are benign -

since integer addition and subtraction commute, `add` and `sub` operations can be asynchronously propagated and applied in any order on all replicas, with the resulting final state guaranteed to be the result of a linearization of all concurrently generated operations[1]. However, since integer multiplication does not commute with addition and subtraction, we cannot simply apply `mult` on various replicas asynchronously, and expect the state to converge. Global synchronization for every multiplication is certainly helpful, but is typically too expensive to be practical [Bailis et al. 2013a,b] at scale. Under such circumstances, it is not readily apparent if we can define replicated counters that support multiplication and yet still have a well-defined semantics that guarantees all replicas will converge to the same counter state.

Fortunately, a state- and merge-centric view of replication lets us arrive at such a semantics naturally. In the current example, we view the replicated counter state as progressing linearly in terms of versions on different replicas. Synchronization between replicas merges their respective (latest) versions into a new version in the context of their lowest common ancestor (LCA) version. We can define the merge operation by focusing on the *difference* between the LCA version and the state on each replica. Fig. 2 illustrates this intuition through an example. Here, two concurrent versions of a counter, 10 and 4, emerge on different replicas starting from a common ancestor (LCA) version 5. The first version 10 is a result of applying `mult 2` to LCA 5, whereas the second version 4 is a result of performing `sub 1`. To merge these concurrent versions, we ignore the operations and instead focus on the difference between each version and the LCA. Here, the differences (literally) are +5 and −1, respectively. The merged version can now be obtained by *composing* the differences and applying the composition on the LCA. Here, composing +5 and −1 gives +4, and applying it to the LCA 5 gives us 9 as the merged version. In general, the merge strategy for an integer counter can be defined in terms of a three-way merge function as follows:

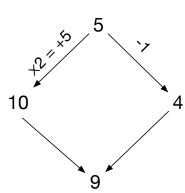


Fig. 2. Counter merge visualized

```
let merge l v1 v2 = l + (v1 - l) + (v2 - l)
```

In the above definition, l is the common ancestor version, whereas v_1 and v_2 are the concurrent versions. Note that the *mergeable* counter described above does not guarantee linearizability (for instance, if the concurrent operations in Fig. 2 are `mult 2` and `mult 3`, then the merge result would be 25 and not 30). Nonetheless, it guarantees convergence, and has a meaningful semantics in the sense that the *effect* of each operation is preserved in the final state. Indeed, such a counter type would be useful in practice, for instance, to record the balance in a banking application, which might use `mult` to compute an account's interest.[2]

The `Counter` example demonstrates the utility of a state- and merge-centric view of replication, and the benefit of using *differences* as a means of reasoning about merge semantics. Indeed, the abstract notion of a difference is general enough that it would appear to make sense (intuitively) to apply a similar approach for other data types. However, this notion does not easily generalize because data types often have complex inductive definitions built using other data types, making it hard to uniformly define concepts involving differences, their application, and their composition. It is in this context that we find our second observation useful. While data types are by themselves quite diverse, we note that they can nonetheless be mapped losslessly to the rich domain of relations over sets, wherein relations so merged can be mapped back to the concrete domain

[1] Implicit here is the assumption of an operation-centric model of replication, where an operation is immediately applied at one replica, and lazily propagated to other replicas [Burckhardt et al. 2014; Li et al. 2012a; Shapiro et al. 2011a; Sivaramakrishnan et al. 2015].

[2] Contrary to popular belief, real-world banking applications are weakly consistent [Brewer 2013]

to yield consistent and useful definitions of these aforementioned concepts. The semantics of a merge in the relational set domain, albeit non-trivial, is nonetheless standard in the sense that it is independent of the concrete interpretations (in the data type domain) of the merging relations, and hence can be defined once and for all. This suggests that the merge semantics for arbitrary data types can be automatically derived, given a pair of abstraction (α) and concretization (γ) functions for each data type that map the values of that type to and from the relational domain (the pair (α, γ) is an *invertible relational specification* of the type). The approach, summarized in Fig. 3, is indeed the one we use to automatically derive merges in this paper. The resultant *mergeable replicated data types* (MRDTs or *mergeable types*, for short) have well-defined distributed semantics in the same sense as the mergeable counter (i.e., a merge operation applied at each replica results in the same state that preserves the effects of all operations performed on all replicas).

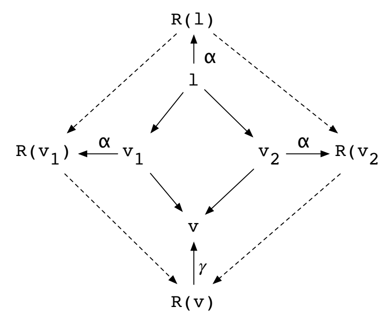


Fig. 3. Merging values in relational domain with help of abstraction (α) and concretization (γ) functions. Solid (resp. dashed) unlabeled arrows represent a merge in the concrete (resp. abstract) domain.

To make MRDTs an effective component of a distributed programming model that yield tangible benefits to programmers, they must be supported by an underlying runtime system that facilitates efficient three-way merges and state replication. Such a system would have to track the provenance (i.e., full history) of concurrently evolving versions, facilitate detection and sharing of common sub-structure across multiple versions, allow efficient computation and propagation of succinct “diffs” between versions, and ideally also support persistence of replicated state. Fortunately, these demands can be readily met by a content-addressable storage abstraction underlying modern version control systems such as Git. Indeed, we have successfully implemented a range of MRDTs, including mergeable variants of lists, queues, trees, maps and heaps, as well as realistic applications composed of such data types, including standard database benchmarks such as TPC-C and TPC-E, on top of the content-addressable storage abstraction underlying Git, and have evaluated them with encouraging results.

In summary, the contributions of this paper are the following:

(1) We introduce the notion of a *mergeable* data type, a high-level abstraction equipped with a three-way merge operation to allow different replica-local states of its instances to be sensibly merged.
(2) We formalize well-definedness conditions for mergeable types by interpreting the behavior of merge actions in a relational set-theoretic framework and show that such an interpretation allows the expression of a rich class of merge functions with intuitive semantics that is significantly more expressive than CRDTs and related mechanisms. More importantly, we show that declarative specifications defining the correctness conditions for merge operations provide sufficient structure to enable automated synthesis of correct merges.
(3) We describe Quark, an implementation of mergeable data types in OCaml built on top of a distributed, content-addressable, version-based, persistent storage abstraction that enables highly efficient merge operations.
(4) A detailed experimental study over a collection of data structure benchmarks as well as well-studied large-scale applications justify the merits of our approach.

The remainder of the paper is structured as follows. In the next section, we provide a more detailed motivating example to illustrate our ideas. Sec. 3 formalizes the concept of relational abstraction for data structures. Sec. 4 defines the rules to derive merge specifications for data structures given their relational abstractions. Sec. 5 provides details on how to automatically derive well-formed merge functions from these specifications. Sec. 6 presents details about Quark's implementation. Sec. 7 discusses experimental results. Related work and conclusions are given in Sec. 8.

2 MOTIVATION

Consider a queue data structure whose OCaml interface is shown in Fig. 4. `Queue` supports two operations: `push` a that adds an element a to the tail end of the queue, and `pop` that removes and returns the element at the head of the queue (or returns `None` if the queue is empty). We say the client that performed `pop` has *consumed* the popped element. For simplicity, we realize queue as a list of elements, i.e., we concretize the type `'a Queue.t` as `'a list` for this discussion. Like `Counter` with `mult`, `Queue`'s implementation does not qualify it as a CRDT, since `push` and `pop` do not commute. Hence, its semantics under (operation-centric) asynchronous replication is ill-founded as illustrated in Fig. 5.

The execution shown in Fig. 5a starts with two replicas, R_1 and R_2, of a queue containing the elements 1 followed by 2. Two distinct clients connect to each of the replicas and concurrently perform `pop` operations, simultaneously consuming 1. The `pop`s are then propagated over the network and applied at the respective remote replicas to keep them consistent with the origin. However, due to a concurrent `pop` already being applied at the remote replica, the subsequently arriving `pop` operation pops a different and yet-to-be-consumed element 2 in each case. The result is a convergent yet incorrect final state, where the element 2 vanishes without ever being consumed. Fig. 5b shows a very similar execution that involves `push`es instead of `pop`s. Starting from a singleton queue containing 1, two concurrent `push` operations push elements 2 and 3 resp. on different replicas. When these operations are eventually applied at the remotes, they are applied in different orders, resulting in the divergence of replica states. Fig. 5c shows another example of divergence, this time involving both `push`es and `pop`s. The execution starts with two replicas, R_1 and R_2, of a singleton queue containing the 1. Two `pop` operations are concurrently issued by clients, both (independently) consuming 1. The `pop`s are then applied at the respective remotes after a delay. During this delay, R_1 sees no activity, leaving the queue empty for R_2's `pop`, which effectively becomes a `Nop`. On R_2 however, a `push 2` operation is performed meanwhile, so when R_1's `pop` is subsequently applied, it pops the (yet unconsumed) element 2. As a result, the final state of the queue on R_2 is empty. Like the `pop`s, the `push 2` operation is also propagated and eventually applied on R_1, resulting in the final state on R_1 being a singleton queue. Thus the replicas R_1 and R_2 of the final state of the queue diverge, which preempts any consistent semantics of the queue operations from being applied to explain the execution.

```
module Queue: sig
  type 'a t
  val push: 'a -> 'a t -> 'a t
  val pop: 'a t -> 'a option * 'a t
end = ...
```

Fig. 4. The signature of a queue in OCaml

Bad executions such as those in Fig. 5 can be avoided if every queue operation globally synchronized. However, as explained before, enforcing global synchronization requires sacrificing availability (i.e., latency), an undesirable tradeoff for most applications [Brewer 2000]. It may therefore seem impossible to replicate queues with meaningful and useful semantics without losing availability. Fortunately, this turns out not to be the case. In the context of real applications, there exist implementations of highly available replicated queues whose semantics, albeit non-standard, i.e., not linearizable or serializable, have nonetheless proven to be useful. Amazon's Simple Queue

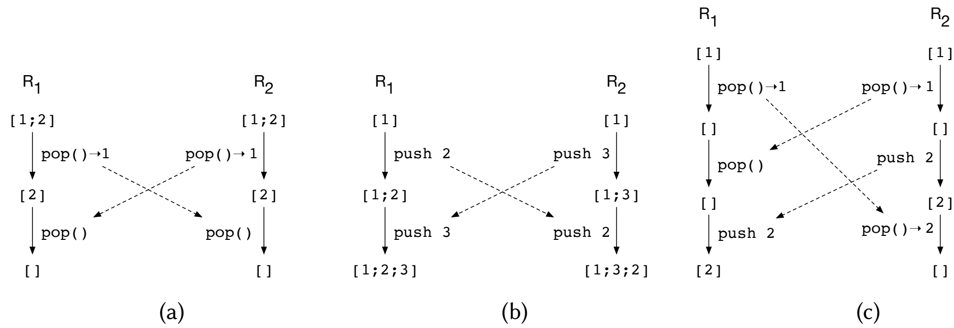


Fig. 5. Ill-formed queue executions

Service (SQS) [Amazon SQS [n. d.]] is one such queue implementation with a non-standard *at-least-once delivery* semantics, which guarantees, among other things, that a queued message is delivered to a client for consumption at least once. Devoid of a formal context, such semantics may seem *ad hoc*; however, casting the Queue data type as a mergeable type would let us *derive* such semantics from first principles, thus giving us a formal basis to reason about its correctness.

Recall that our underlying execution model is based on state-centric model of replication with versioned state and explicit three-way merges (which we show how to synthesize). Under this model, two concurrent versions v_1 and v_2 of a queue can independently evolve from a common ancestor (LCA) version l. The semantics of the queue under replication depends on how these versions are merged into a single version v (Fig. 3). The concurrent versions v_1 and v_2 would have evolved from l through several push and pop applications, however let us ignore the operations for a while and focus on the relationship between the queue states l, v_1, and v_2. Intuitively, the following relationships must hold among the three queues:

(1) For every element $x \in l$, if $x \in v_1$ and $x \in v_2$, i.e., if x is not popped in either of the concurrent versions, then $x \in v$, i.e., x must be in the merged version. In other words, a queue element that was never consumed should *not* be deleted.
(2) For every $x \in l$ if $x \notin v_1$ or $x \notin v_2$, i.e., if x is popped in either v_1 or v_2, then $x \notin v$. That is, a consumed element (regardless of how many times it was consumed) should never reappear in the queue.
(3) For every $x \in v_1$ (resp. v_2), if $x \notin l$, that is x is newly pushed into v_1 (resp. v_2), then $x \in v$. That is, an element that is newly added in either concurrent versions must be present in the merged version.
(4) For every $x, y \in l$ (resp. v_1 and v_2), if x occurs before y in l (resp. v_1 and v_2), and if $x, y \in v$, i.e., x and y are not deleted, then x also occurs before y in v. In other words, the order of elements in each queue must be preserved in the merged queue.

To formalize these properties more succinctly, we define two relations on lists: (1). A *membership* relation on a list l (written $R_{mem}(l)$) is a unary relation, i.e., a set, containing all the elements in l, and (2). An *occurs-before* relation on l (written $R_{ob}(l)$) is a binary relation relating every pair of elements x and y in l, such that x occurs before y in l. For a concrete list $l = [1; 2; 3]$, $R_{mem}(l)$ is the set $\{1, 2, 3\}$, and $R_{ob}(l)$ is the set $\{(1, 2), (1, 3), (2, 3)\}$. Note that for any list l $R_{ob}(l) \subseteq R_{mem}(l) \times R_{mem}(l)$, i.e., $R_{ob}(l)$ is only defined for the elements in $R_{mem}(l)$. Using R_{mem}, we can succinctly specify the relationship among the members of l, v_1, v_2, and v, where $v = \texttt{merge}\ l\ v_1\ v_2$, as follows[3]:

$$\begin{aligned} R_{mem}(v) \quad = \quad & R_{mem}(l) \cap R_{mem}(v_1) \cap R_{mem}(v_2) \\ & \cup \quad R_{mem}(v_1) - R_{mem}(l) \quad \cup \quad R_{mem}(v_2) - R_{mem}(l) \end{aligned} \tag{1}$$

The left hand side denotes the set of elements in the merged version v. The right hand side is a union of three components: (1). The elements common among three versions l, v_1, and v_2, (2). The elements in v_1 not in l, i.e., newly added in v_1, and (3). The elements in v_2 not in l, i.e., newly added in v_2. Observe that we applied the same intuitions as the counter merge from Sec. 1 to arrive at the above specification, namely merging concurrent versions by computing, composing and applying their respective *differences* to the common ancestor. However, we have interpreted the *difference* through the means of a relation over sets that abstracts the structure of a queue and captures only its membership property. Another important point to note is that the specification does not appeal to any operational characteristics of queues, either sequentially or in the context of replication.

Similar intuitions can be applied to manage the structural aspects of merging queues by capturing their respective *orders* via the *occurs-before* relation (R_{ob}) over lists, but after accounting for a couple of caveats. First, since $R_{ob} \subseteq R_{mem} \times R_{mem}$, $R_{ob}(v)$ has to be confined to the the domain of $R_{mem}(v) \times R_{mem}(v)$. Second, the order between a pair of elements where each comes from a distinct concurrent version is indeterminate, thus $R_{ob}(v)$ can only be underspecified. Taking these caveats into account, $R_{ob}(v)$ of the merged version v can be specified thus:

$$\begin{aligned} R_{ob}(v) \quad \supseteq \quad & (R_{ob}(l) \cap R_{ob}(v_1) \cap R_{ob}(v_2) \\ & \quad \cup \quad R_{ob}(v_1) - R_{ob}(l) \quad \cup \quad R_{ob}(v_2) - R_{ob}(l)) \\ & \cap \quad (R_{mem}(v) \times R_{mem}(v)) \end{aligned} \tag{2}$$

Note the $\supseteq$ capturing the underspecification. The right hand side is essentially same as the right hand side of the R_{mem} equation (above), except that R_{ob} replaces R_{mem}, and we compute an intersection with $R_{mem}(v) \times R_{mem}(v)$ at the top level to confine $R_{ob}(v)$ to the elements in v. As mentioned earlier, the specification does not induce a fixed order among elements coming from different queues. To recover convergence, a merge function on queues can choose to order such elements through a consistent ordering relation, such as a lexicographic order.

The *membership* and *occurs-before* specifications together characterize the merge semantics of the queue data type that we derived from basic principles we enumerated above. We shall now reconsider the executions from Fig. 5, this time under a state-centric model of replication, and demonstrate how our merge specification leads us to a consistent distributed semantics for queue, which subsumes a *at-least-once delivery* semantics. The corresponding executions under this model are shown in Fig. 6.

Fig. 6a is the same execution in Fig. 5a with the dashed line representing a version propagation followed by a merge, rather than an operation propagation followed by an application. For each version, the R_{mem} and R_{ob} relations are shown below its actual value. If the version is a result of a merge, then we compute its R_{mem} and R_{ob} sets using equations 1 and 2 of the merge specification above. For both the merges shown in the figure, the concurrent versions (v_1 and v_2) are the same: the singleton queue `[2]`, and their LCA version (l) is the initial queue `[1;2]`. Thus each concurrent version is a result of popping 1 from the LCA (which is consumed/delivered twice as acceptable under *at-least-once delivery* semantics). Intuitively, the result of the merge should be a version that incorporates the effect of popping 1, while leaving the rest of the queue unchanged from the LCA. This leaves the queue `[2]` as the only possible result of the merge (and the execution). Indeed, this is the result we would obtain if reconstruct the queue from the merged R_{mem} and R_{ob} relations

[3]We elide parentheses for perspicuity. Any ambiguity in parsing should be resolved by assuming that $\cap$ and $-$ bind tighter than $\cup$

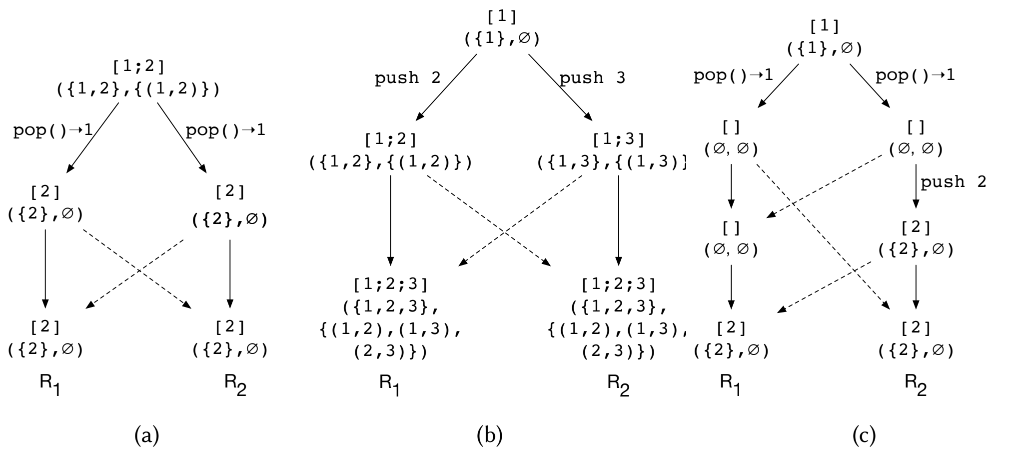


Fig. 6. State-centric view of queue replication aided by context-aware merges (shown in dashed lines)

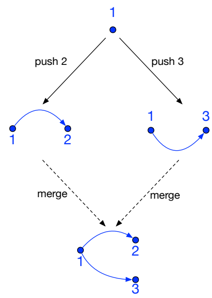
```
let rec R_mem = function
    | [] -> ∅
    | x::xs -> {x} ∪ R_mem(xs)

let rec R_ob = function
    | [] -> ∅
    | x::xs -> ({x} × R_mem(xs)) ∪ R_ob(xs)
```

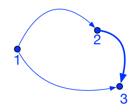
Fig. 7. Functions that compute R_{mem} and R_{ob} relations for a list. Syntax is stylized to aid comprehension.

shown in the figure. Execution in Fig. 6b corresponds to the one in Fig. 5b. Here we have two merges: one into R_1 and other into R_2. The concurrent versions for both the merges are the same: `[1;2]` and `[1;3]`, and their LCA is the queue `[1]`. Each concurrent version pushes a new element (2 and 3, resp.) to the queue *after* the existing element 1. Intuitively, the merged queue should contain both the new elements ordered after 1. Indeed, this is also what the merged R_{mem} and R_{ob} relations suggest. The order between new elements, however, is left unspecified by R_{ob}. As mentioned earlier, a consistent ordering relation has to be used to order such elements. Choosing the less-than relation, we obtain the result of the merge as `[1;2;3]`. In Fig. 6c, there are three merges: two into R_1 and one into R_2. For the first merge into R_1, the concurrent versions are both empty queues, and their LCA is the singleton queue `[1]`. Thus both versions represent a pop of 1, and their merged version, which reconciles both the pops, should be an empty queue, which is also what the merged relations suggest. The second merge into R_1 and the only merge into R_2, both merge an empty queue (`[]`) and a singleton queue `[2]`, with the LCA version being the initial queue `[1]`. While the version `[]` can be understood as resulting from the popping an element from LCA, the concurrent version `[2]` goes one step ahead and pushes a new element 2. Consequently, the merged version should be a queue not containing 1, but containing the new element 2, i.e., `[2]`, which is again consistent with the result obtained by merging R_{mem} and R_{ob} relations. Thus in all three executions discussed above, the relational merge specification (Eqs. 1 and 2) consistently guides us towards a meaningful result, imparting a well-defined distributed semantics to the queue data type in the process.

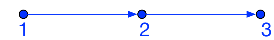
To operationalize the merge specification discussed above, i.e., to derive a merge function that *implements* the specification, we require functions (α and γ resp.) to map a queue to the relational domain and back. The abstraction function α is simply a pair-wise composition of functions that compute R_{mem} and R_{ob} relations for a given list. The eponymous functions are shown in Fig. 7.



(a) R_{ob} relations of merging queues from Fig. 6b visualized as graphs



(b) The incomplete R_{ob} resulting from the merge is completed by adding an *arbitration edge* between 2 and 3 consistent with their domain ordering



(c) The topological ordering of the completed R_{ob} graph yields the result of queue merge

Fig. 8. Concretizing a queue from a subset of its R_{ob} relation resulting from a merge

The R_{mem} function computes the set of elements in a given list l, which is its unary membership relation. The function R_{ob} computes the set of all pairs (x, y) such that x occurs before y in l. The concretization function γ reconstructs a list/queue given a subset of its R_{mem} and R_{ob} relations (The subsets are a consequence of underspecification, e.g., R_{ob} specification in Eq. 2). One way γ can materialize a list from the given subsets of its R_{mem} and R_{ob} relations is by constructing a directed graph G whose vertices are $R_{mem}(v)$, and edges are $R_{ob}(v)$. A topological ordering of vertices in G, where ties are broken as per a consistent *arbitration* order (e.g., lexicographic order) yields the merged list/queue. Fig. 8 demonstrates this approach for the queue merge example in Fig. 6b.

We have generalized the aforementioned graph-based approach for concretizing ordering relations, and abstracted it away as a library function γ_{ord}. Given *ord*, an *arbitration order*, the function γ_{ord} concretizes an ordering relation of a data structure (not necessarily a total order) as a graph isomorphic to that structure, using the arbitration order to break ties (as shown in Fig. 8b). More discussion on γ_{ord} can be found in Sec. 5. Instantiating *ord* with less-than relation (<) on integers, the concretization function of a queue can be written as shown in Fig. 9a. The result of $\gamma_{<}$(`rmem`,`robs`) is a list-like graph as shown in Fig. 8c. The function `mk_list` traverses the graph beginning from its root to construct a list isomorphic with the graph. Standard library function `Set.elements` is returns a list of elements in a set. The `DiGraph` library is assumed to support a function `root` that returns a root (vertex with indegree 0) of a directed graph, and a function `succ` that returns the list of successors of the given vertex in the graph.

The γ_{ord} function thus (mostly) automates the task of concretizing orders, which is usually the non-trivial part of writing γ. Given both α and γ, the merge function for queues (lists, in general) follows straightforwardly from the merge specification as shown in Fig. 9b. For brevity, we write $A \diamond B \diamond C$ to denote the three-way merge of sets A, B, and C, which is defined thus:

$$A \diamond B \diamond C = (A \cap B \cap C) \,\cup\, (B - A) \,\cup\, (C - A)$$

3 ABSTRACTING DATA STRUCTURES AS RELATIONS

The various data structures defined by a program differ in terms of the patterns of data access they choose to support, e.g., value lookups in case of a tree and insertions in case of an unordered list. Nonetheless, regardless of its access pattern priorities, a data structure can be uniquely characterized by the contents it holds, and the structural relationships defined among them. This observation lets

```
let γ (rmem, robs) =
  if robs = ∅
  then Set.elements rmem
  else
    let g = γ< (rmem,robs) in
    let rec mk_list x =
      match DiGraph.succ x with
      | [] -> [x]
      | [y] -> x::(mk_list y)
      | _ -> error()
    mk_list (DiGraph.root g)
```

(a) Queue concretization function in OCaml

```
let merge l v1 v2 =
  let (rmem_l, robs_l) = α(l) in
  let (rmem_v1, robs_v1) = α(v1) in
  let (rmem_v2, robs_v2) = α(v2) in
  let rmem_v = rmem_l ⋄ rmem_v1 ⋄ rmem_v2
    in
  let robs_v = (robs_l ⋄ robs_v1 ⋄
      robs_v2)
                 ∩ (rmem_v × rmem_v) in
  γ(rmem_v, robs_v)
```

(b) Queue merge composed of abstraction (α) and concretization (γ) functions

Fig. 9. (Along with Fig. 7) Relational approach to queue merge materialized in OCaml

us capture salient aspects of an arbitrary data structure using concrete artifacts, such as sets and relations.

The relational encoding of the list data type has already been demonstrated in Sec. 2. As shown, membership and order properties of a list l, represented by relations $R_{mem}(l)$ and $R_{ob}(l)$, *characterize* l in the sense that one can reconstruct the list l given these two relations[4]. We call such relations the *characteristic relations* of a data type, a notion we shall formalize shortly. Note that characteristic relations need not be unique. For instance, we could equivalently have defined an *occurs-after* (R_{oa}) relation - a dual of the occurs-before relation, that relates the list elements in reverse order, and use it in place of R_{ob} as a characteristic relation for lists without any loss of generality.

Relational abstractions can be computed for other data types too, but before describing a general procedure for doing so, we first make explicit certain heretofore implicit conventions we have been using in the presentation thus far. First, we often use a relation name (e.g., R_{mem}) interchangeably to refer to the relation as well as the function that computes that relation. To be precise, $R_{mem}(l)$ is the membership relation for a list l, whereas R_{mem} is a function that computes such a relation for any list l. But we prefer to call them both relations, with the latter being thought of as a relation parameterized on lists. Second, we use relations and sets to characterize data structures in this presentation, when the proper abstraction is multi-sets, i.e., sets where each element carries a unique cardinal number. While using sets leads to a simpler formulation and typically does not result in any loss of generality, we explicitly use multi-sets when they are indeed required.

As another example of a relational specification, consider the characteristic relations that specify a binary tree whose OCaml type signature is given below:

```
type 'a tree = | E
               | N of 'a tree * 'a * 'a tree
```

An R_{mem} function can be defined for trees similar to lists that computes the set of elements in a tree. A tree may denote a binary heap, in which case an *ancestor* relation is enough to capture its

[4]One might think R_{ob} itself is sufficient, but that is not true. R_{ob} is empty for both singleton and empty lists, making it impossible to distinguish between them.

Data Type	Characteristic Relations
Binary Heap	Membership (R_{mem}), Ancestor ($R_{ans} \subseteq R_{mem} \times R_{mem}$)
Priority Queue	Membership (R_{mem})
Set	Membership (R_{mem})
Graph	Vertex (R_V), Edge (R_E)
Functional Map	Key-Value (R_{kv})
List	Membership (R_{mem}), Order (R_{ob})
Binary Tree	Membership (R_{mem}), Tree-order ($R_{to} \subseteq R_{mem} \times \texttt{label} \times R_{mem}$)
Binary Search Tree	Membership (R_{mem})

Table 1. Characteristic relations for various data types

structure (since relative order between siblings does not matter). The definition is shown below:

```
let rec R_ans = function
  | E -> ∅
  | N(l,x,r) ->
    let des_x = R_mem(l) ∪ R_mem(r) in
    let r_ans = {x} × des_x in
    R_ans(l) ∪ r_ans ∪ R_ans(r)
```

```
type label = L | R
let rec R_to = function
  | E -> ∅
  | N(l,x,r) ->
    let l_des = {x} × {L} × R_mem(l) in
    let r_des = {x} × {R} × R_mem(r) in
    R_to(l) ∪ l_des ∪ r_des ∪ R_to(r)
```

The full structure of the tree, including the relative order between siblings, can be captured via as a ternary *tree-order* relation (R_{to} shown above) that extends the ancestor relation with labels denoting whether an element is to the left of its ancestor or to its right.

However, the shape of a data structure may not always be relevant. For instance, given two binary search trees with the same set of elements, it does not matter whether they have the same shape. Their extensional behavior is presumably indistinguishable since they would give the same answers to the same queries. In such cases, a membership relation is enough to completely characterize a tree. Indeed, different data types have different definitions of extensional equality, so we take that into account in formalizing the notion of characteristic relations:

Definition 3.1. A sequence of relations $\overline{R}_T$ is called the characteristic relations of a data type T, if for every $x : T$ and $y : T$, $\overline{R}_T(x) = \overline{R}_T(y)$ implies $x =_T y$, where $=_T$ denotes the extensional equality relation as interpreted by T.

Our formalization requires the type of each characteristic relation to be specified in order to derive a merge function for that relation. This type is not necessarily the same as its OCaml type for we let additional constraints be specified to precisely characterize the relation. The syntax of relation types and other technicalities are discussed in Sec. 4.

The approach of characterizing data structures in terms of relations is applicable to many interesting data types as shown in Table 1. The vertex and edge relations of a graph are essentially its vertex and edge sets respectively. The key-value relation of a functional map is a semantic relation that relates each key to a value. Concretely, it is just a set of key-value pairs.

Basic data types, such as natural numbers and integers, can also be given a relational interpretation in terms of multi-sets, although such an interpretation is not particularly enlightening. For example, a natural number n can be represented as a multi-set $\{1 : n\}$, meaning that it is equal to a set containing n ones. Zero is the empty set $\{\}$. Addition corresponds to multi-set union, subtraction to multi-set difference, and a minimum operation to multi-set intersection.

4 DERIVING RELATIONAL MERGE SPECIFICATIONS

In Sec. 2, we presented a merge specification for queues expressed in terms of the membership (R_{mem}) and order (R_{ob}) relations of the list data type. The specification realizes the abstract idea of merging concurrent versions by computing, composing and applying *differences* to the LCA. Similar specifications can be derived for other inductive data types, such as trees, graphs, etc. in terms of their characteristic relations listed in Table 1. Beyond these data types, however, the approach suggested thus far is presumably hard to generalize as it ignores an important aspect of data type construction, namely composition. In this section, we first demonstrate the challenges posed by data structure composition, and subsequently generalize our approach to include such compositions. We also formalize our approach as a set of (algorithmic) rules to *derive* merge specifications for arbitrary data structures and their compositions, given their characteristic relations, and abstraction/concretization functions.

4.1 Compositionality

Consider an integer pair type - `int*int`. One might define relations R_{fst} and R_{snd} on `int*int` as follows: R_{fst} and R_{snd} comprise the characteristic relations of integer pairs since if the relations

$$\texttt{let}\ R_{fst}\ \texttt{= fun (x,\_) -> \{x\}} \qquad \texttt{let}\ R_{snd}\ \texttt{= fun (\_,y) -> \{y\}}$$

are equal for two integer pairs, then the pairs themselves must be equal. Using these relations, one might try to specify the merge semantics of the pair type by emulating the membership (R_{mem}) specification from the queue example of Sec. 2. Let v_1 and v_2, each an integer pair, denote the merging versions, and let l be their LCA version. Let v be the result of their three-way merge, i.e., $v\ =\ \texttt{merge}\ l\ v_1\ v_2$. Substituting R_{mem} with R_{fst} (resp. R_{snd}) in queue's merge specification leads to the following:

$$\begin{array}{rcl} R_{fst}(v) & = & R_{fst}(l) \cap R_{fst}(v_1) \cap R_{fst}(v_2) \\ & & \quad \cup \quad R_{fst}(v_1) - R_{fst}(l) \quad \cup \quad R_{fst}(v_2) - R_{fst}(l) \\ R_{snd}(v) & = & \ldots\texttt{(respectively for}\ R_{snd}\texttt{)} \end{array}$$

Unfortunately, the specification is meaningless in the context of a pair. Fig. 10 illustrates why. Here, two concurrent `int*int` versions, (3,4) and (5,6), evolve from an initial version (1,2). Their respective R_{fst} and R_{snd} relations are as shown in the figure. Applying the above specification for the `int*int` merge function, we deduce that the R_{fst} and R_{snd} relations for the merged version should be the sets $\{3, 5\}$ and $\{4, 6\}$, respectively. However, the sets do not correspond to any integer pair, since R_{fst} and R_{snd} for any such pair is expected to be a singleton set. Hence the specification is incorrect.

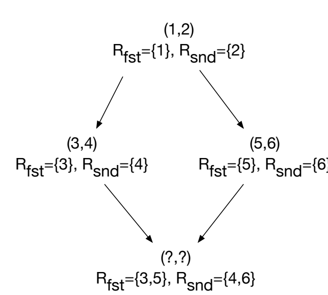


Fig. 10. Incorrect merge of integer pairs

Clearly, the approach we took for queue does not generalize to a pair. The problem lies in how we view these two data structures from the perspective of merging. While the merge specification we wrote for queue treats it as a collection of unmergeable atoms, such an interpretation is not sensible for pairs, as the example in Fig. 10 demonstrates. Unlike a queue, a pair defines a fixed-size container that assigns an ordinal number ("first", "second" etc) to each of its elements. Two versions of a pair are mergeable only if their elements with corresponding ordinals are mergeable. In Fig. 10, if we assume the integers are in fact (mergeable) counters (i.e., `Counter.t` objects), we can use `Counter.merge` to merge the first and second components of the merging pairs independently, composing them into a merged pair as described below:

$$
\begin{array}{lcl}
T, \tau \in \texttt{Data Types} & & R \in \texttt{Relation Names} \\
\rho \in \texttt{Tuple Types} & := & T \mid R(v) \mid \rho \times \rho \\
s \in \texttt{Relation Types} & := & \{v : T\} \rightarrow \mathcal{P}(\rho)
\end{array}
$$

Fig. 11. Type specification syntax for (functions that compute) relations

```
let merge l v1 v2 = (Counter.merge (fst l) (fst v1) (fst v2),
                     Counter.merge (snd l) (snd v1) (snd v2))
```

Recall that the Counter.merge is the following function:

```
let merge l v1 v2 = l + (v1 - l) + (v2 - l)
```

Thus the result of merging the pair of counters and their LCA from Fig. 10 is:

```
(Counter.merge 1 3 5, Counter.merge 2 4 6) = (7,8)
```

The pair example demonstrates the need and opportunity to make merges compositional. The specification of such a composite merge function is invariably compositional in terms of the merge specifications of the types involved. Let $\phi_c(l, v_1, v_2, v)$ denote the counter merge specification defined, for instance, thus:

$$\phi_c(l, v_1, v_2, v) \Leftrightarrow v = l + (v_1 - l) + (v_2 - l)$$

We can now define a merge specification ($\phi_{c \times c}$) for counter pairs in terms of ϕ_c, and the relations R_{fst} and R_{snd} as follows:

$$
\begin{array}{lcl}
\phi_{c\times c}(l, v_1, v_2, v) & \Leftrightarrow & \forall x, y, z, s.\ x \in R_{fst}(l) \ \wedge \ y \in R_{fst}(v_1) \ \wedge \ z \in R_{fst}(v_2) \\
& & \qquad \wedge\ \phi_c(x, y, z, s) \Rightarrow s \in R_{fst}(v) \\
& & \wedge\ \forall s.\ s \in R_{fst}(v) \Rightarrow \exists x, y, z.\ x \in R_{fst}(l) \ \wedge \ y \in R_{fst}(v_1) \\
& & \qquad \wedge\ z \in R_{fst}(v_2) \ \wedge \ \phi_c(x, y, z, s) \\
& & \wedge\ \ldots \texttt{(respectively for } R_{snd}\texttt{)}
\end{array}
$$

The first conjunct on the right hand side essentially says that if (counters) x, y, and z are respectively the first components of the pairs l, v_1 and v_2, and s is the result of merging x, y and z via Counter.merge, then s is the first component of the merged pair v. The second conjunct states the converse. Similar propositions also apply for the second components (accessible via R_{snd}), but elided. Observe that the specification captures the merge semantics of a pair while abstracting away the merge semantics of its component types. In other words, $\phi_{a\times b}$, the merge specification of the type a*b is parametric on the merge specifications ϕ_a and ϕ_b of types a and b respectively. Thus, the merge specification for a pair of queues, i.e., $\phi_{q\times q}$, can be obtained by replacing ϕ_c with ϕ_q, the queue merge specification (Sec. 2) in the above definition. The ability to compose merge specifications in this way is key to deriving a sensible merge semantics for any composition of data structures.

A pair is an example of a composite data structure that assigns implicit ordinals to its constituents. Alternatively, a data structure may assign explicit ordinals or identifiers to its members. For instance, a map abstract data type (implemented using balanced trees or hash tables) identifies its constituent values with explicit keys. In either case, the top-level merge is essentially similar to the one described for pair, and involves merging constituent values that bear corresponding ordinals or identifiers. Note that this assumes that the values are indeed mergeable. Data structures may be composed of types that are not mergeable *by design*, e.g., the keys in a map data type are not mergeable, although they serve to identify the values which are mergeable. Since the merge strategy of a data structure should work differently for its mergeable and non-mergeable constituents, we need a way to identify them as such. This can be done through the type specification of relations, as described below.

4.2 Type Specifications for Characteristic Relations

As mentioned in Sec. 3, characteristic relations of a data type need to be explicitly typed. Fig. 11 shows the syntax of type specifications for such relations. We use both T and τ to refer to data types, with the latter used to highlight that the type being referred to is mergeable. A relation maps a value v of a data type T to a set of tuples each of type ρ. A tuple type is specified in terms of the set from which it is drawn. It could be the set of all values of a (different) type T, or the set defined by a (different) relation R on v, or a cross product of such sets. Note that the cross-product operator is treated as associative in this context, hence for any three sets A, B and C, $A \times (B \times C) = (A \times B) \times C = A \times B \times C$. The syntax allows the type of a relation R on $v : T$ to refer another relation R' on $v : T$ to constrain the domain of its tuples. Some examples of relations with type specifications are given below.

Example 4.1. The characteristic relations of `int list` data type can be specified thus:

$$\begin{aligned} R_{mem} &: \{v : \texttt{int list}\} \rightarrow \mathcal{P}(\texttt{int}), \\ R_{ob} &: \{v : \texttt{int list}\} \rightarrow \mathcal{P}(R_{mem}(v) \times R_{mem}(v)) \end{aligned}$$

Example 4.2. The characteristic relations of a map data type with string keys and counter values can be specified thus:

$$\begin{aligned} R_k &: \{v : \texttt{(string,int) map}\} \rightarrow \mathcal{P}(\texttt{string}), \\ R_{kv} &: \{v : \texttt{(string,int) map}\} \rightarrow \mathcal{P}(R_k(v) \times \texttt{counter}) \end{aligned}$$

Type constraints, as described above, ensure syntactic correctness of relations. However, not all syntactically valid relations lead to semantically meaningful merge specifications. To identify those that do, we define a well-formedness condition on type specifications of relations. Let ρ_R denote the type of tuples in a relation R defined over $v : T$, for some data type T (i.e., $R : v : T \longrightarrow \mathcal{P}(\rho_R)$). Since tuple types can refer to other relations (see ρ in Fig. 11, and the R_{ob} and R_{kv} type definitions above), ρ_R could be composed of $R'(v)$, where R' is another relation on $v : T$. We consider "flattening" such ρ_R by recursively substituting every occurrence of $R'(v)$ with the tuple type $\rho_{R'}$ of R' in ρ_R (i.e., $[\rho'_R/R'(v)]\,\rho_R$). For instance, the flattened tuple types of R_{ob} and R_{kv} are $\texttt{int} \times \texttt{int}$ and $\texttt{string} \times \texttt{int}$, respectively. In general, the flattened tuple type of ρ_R (denoted $\lfloor \rho_R \rfloor$) is a non-empty cross product of the form $T_1 \times T_2 \times \ldots T_n$, which we shorten as $\overline{T}$. We define the well-formedness of a relation's type specification by examining its flattened tuple type as follows.

Definition 4.3. A relation $R : \{v : T\} \longrightarrow \mathcal{P}(\rho)$ is said to have a well-formed type specification if and only if there exists a non-empty $\overline{T}$ and a (possibly empty) $\overline{\tau}$ such that:

- $\lfloor \rho \rfloor = \overline{T} \times \overline{\tau}$, and
- Every $T_i \in \overline{T}$ is *not mergeable*, whereas
- Every $\tau_i \in \overline{\tau}$ is *mergeable*.

Informally, a *mergeable type* is a data type for which a merge specification can be derived, and a merge function that meets the specification exists (e.g., queues and counters). Basic data types, such as strings and floats, are considered not mergeable for the sake of this discussion. The well-formedness definition presented above effectively constrains relations to be one of the following two kinds based on the type of their tuples: (a). those containing tuples composed only of non-mergeable types (i.e., $\overline{\tau} = \emptyset$ and $\lfloor \rho \rfloor = \overline{T}$), and (b). those containing tuples composed of non-mergeable types *followed by* mergeable types (i.e., $\lfloor \rho \rfloor = \overline{T} \times \overline{\tau}$ and $\overline{\tau} \neq \emptyset$). The former are relations that capture the *contents* and the *structural relationships* among the contents in a data structure (e.g., R_{mem}, R_{ob}, and R_k), and the latter are those that capture their *semantic relationships*[5] (e.g., R_{kv} - a relation that identifies key-value relationship latent in each element of a map). Based on this categorization, we can now formalize the rules to derive merge specifications of an arbitrary data type from the well-formed type specification of its characteristic relations.

4.3 Derivation Rules

Fig. 12 shows the derivation rules for merge specifications. The rules define the judgment

$$\phi_T(l, v_1, v_2, v) \supseteq \varphi$$

where ϕ_T is the merge specification for a type T parameterized on the merging versions (v_1 and v_2), their LCA (l), and the merge result (v), and φ is a first-order logic (FOL) formula. The interpretation is that the merge specification ϕ_T should subsume the FOL formula φ. The rules let us derive such constraints for every R on type T with a well-formed type specification $R : T \rightarrow \mathcal{P}(\rho)$. Accumulating the constraints derived over several such applications of the rules (until fixpoint) results in the full merge specification of type T. The rules invoke the definitions of flattening, well-formedness, etc. that we introduced above.

Recall that the tuple type of a relation is a cross product involving data types and other relations. We use its set interpretation in set operations such as intersection. For instance, if the characteristic relation on `int list` has the type ν : `int list` $\rightarrow \mathcal{P}(\texttt{int} \times R_{mem}(\nu))$, then its tuple type $\rho = \texttt{int} \times R_{mem}(\nu)$ has a natural set interpretation as the cross product of the set of all integers and $R_{mem}(\nu)$, and hence can be used in set expressions such as $R_{ob}(\nu) \cap \rho$, as the rules in Fig. 12 do. The notation $A \diamond B \diamond C$ denotes three-way merge of sets A, B, and C, defined formally in Sec. 2. We define an *extension* operation on relations that relate ordinals or identifiers of non-mergeable type(s) $\overline{T}$ with values of mergeable type(s) $\overline{\tau}$. Let R be such a relation on type T, and let 0_i denote the "zero" or "empty" value of type τ_i. We call 0 an empty value of a type if $\overline{R}(0) = \emptyset$ for all characteristic relations $\overline{R}$ on that type (e.g., an empty list for type `list`). An extension of R is a relation R_+ that relates ordinals or identifiers not already related by R to empty or zero values. Formally, we define R_+ by defining its containment relation as follows:

$$\forall(\overline{k} : \overline{T}).\forall(\overline{x} : \overline{\tau}).\ \ (\overline{k}, \overline{x}) \in R_+ \Leftrightarrow (\overline{k}, \overline{x}) \in R \ \vee\ (\nexists(\overline{y} : \overline{\tau}).\ (\overline{k}, \overline{y}) \in R \ \wedge\ \textstyle\bigwedge_i x_i = 0_i)$$

A tuple $(\overline{k}, \overline{x})$ is in R_+ if and only if it is already in R, or R does not relate $\overline{k}$ to anything, and each x_i is an empty value. We also define a *projection* of R, denoted R_k, that is simply the set of ordinals or identifiers in R. The definition is as follows:

$$\forall(\overline{k} : \overline{T}).\ \ \overline{k} \in R_k \Leftrightarrow \exists(\overline{x} : \overline{\tau}).\ (\overline{k}, \overline{x}) \in R$$

Note that R_+ and R_k are merely notations to simplify the rules in Fig. 12, as will be evident shortly.

The rule Set-Merge derives merge constraints for a relation R that is composed of only non-mergeable types ($\overline{T}$), and do not draw on other relations, i.e., its tuple type ρ is not a cross product of other relations. Thus, R capture the elements of T rather than their relative order. Examples include R_{mem} (list) and R_k (map). The consequent of Set-Merge enforces the set merge semantics on R, and is an exact specification of the merge result, leaving no room for the merge function to conjure new elements of its own. As an example, one can apply the Set-Merge rule to the `int list` type to obtain a constraint on R_{mem} as described in Sec. 2.

The rule Order-Merge-1 constrains a relation R whose tuple type ρ involves cross-product of other relations. Thus the relation R can be construed as an ordering relation over tuples captured by other relations over the same data structure. Examples include R_{ob} (binary relation on lists) and

[5] This categorization corresponds exactly to the properties of interest that were said to uniformly characterize all data structures (Sec. 3).

$\boxed{\phi_T(l, v_1, v_2, v) \supseteq \varphi}$

$$\frac{R : \{v : T\} \to \mathcal{P}\left(\overline{T}\right)}{\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{x} : \overline{T}).\ \overline{x} \in (R(l) \diamond R(v_1) \diamond R(v_2)) \Leftrightarrow \overline{x} \in R(v)} \quad [\textsc{Set-Merge}]$$

$$\frac{R : \{v : T\} \to \mathcal{P}(\rho) \quad \lfloor \rho \rfloor = \overline{T}}{\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{x} : \overline{T}).\ \overline{x} \in (R(l) \diamond R(v_1) \diamond R(v_2)\ \cap\ \rho) \Rightarrow \overline{x} \in R(v)} \quad [\textsc{Order-Merge-1}]$$

$$\frac{R : \{v : T\} \to \mathcal{P}(\rho) \quad \lfloor \rho \rfloor = \overline{T}}{\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{x} : \overline{T}).\ \overline{x} \in R(v) \Rightarrow \overline{x} \in \rho} \quad [\textsc{Order-Merge-2}]$$

$$\frac{R : \{v : T\} \to \mathcal{P}(\rho) \quad \lfloor \overline{\rho} \rfloor = \overline{T} \times \overline{\tau} \quad \overline{\tau} \neq \emptyset}{\begin{array}{l}\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{k} : \overline{T}).\forall(\overline{x}, \overline{y}, \overline{z}, \overline{s} : \overline{\tau}).\ (\overline{k}, \overline{x}) \in R_+(l)\ \wedge\ (\overline{k}, \overline{y}) \in R_+(v_1)\ \wedge\ (\overline{k}, \overline{z}) \in R_+(v_2) \\ \quad \wedge\ \overline{k} \in (R_k(l) \diamond R_k(v_1) \diamond R_k(v_2))\ \wedge\ \bigwedge_i \phi_{\tau_i}(x_i, y_i, z_i, s_i)\ \wedge\ (\overline{k}, \overline{s}) \in \rho \Rightarrow (\overline{k}, \overline{s}) \in R(v)\end{array}} \quad [\textsc{Rel-Merge-1}]$$

$$\frac{R : \{v : T\} \to \mathcal{P}(\rho) \quad \lfloor \overline{\rho} \rfloor = \overline{T} \times \overline{\tau} \quad \overline{\tau} \neq \emptyset}{\begin{array}{l}\phi_T(l, v_1, v_2, v) \supseteq \forall(\overline{k} : \overline{T}).\forall(\overline{s} : \overline{\tau}).\ (\overline{k}, \overline{s}) \in R(v) \Rightarrow (\overline{k}, \overline{s}) \in \rho \\ \quad \wedge\ \exists(\overline{x}, \overline{y}, \overline{z} : \overline{\tau}).\ (\overline{k}, \overline{x}) \in R_+(l)\ \wedge\ (\overline{k}, \overline{y}) \in R_+(v_1)\ \wedge\ (\overline{k}, \overline{z}) \in R_+(v_2) \\ \quad\quad \wedge\ \overline{k} \in (R_k(l) \diamond R_k(v_1) \diamond R_k(v_2))\ \wedge\ \bigwedge_i \phi_{\tau_i}(x_i, y_i, z_i, s_i)\end{array}} \quad [\textsc{Rel-Merge-2}]$$

Fig. 12. Rules to derive a merge specification for a data type T

R_{to} (ternary relation on trees). The conclusion of Order-Merge-1 adds a constraint to ϕ_T that merely enforces the set merge semantics over the ordering relation R, while retaining only those tuples that belong to the set ρ. The constraint is only an implication (and not a bi-implication), thereby underspecifying the merge result, and letting the merge function add new orders on existing elements. However, in order to prevent the merge from creating elements out of thin air, we need a constraint in reverse direction, albeit a weaker one. The rule Order-Merge-2 fulfills this need, by restricting the tuples in the merged order relation to be drawn from the cross product of existing relations (ρ). Observe that these two rules together give us the constraints on R_{ob} that we wrote for the queue data structure in Sec. 2.

The rules Rel-Merge-1 and Rel-Merge-2 are concerned with the last category of relations that relate a data structure composed of multiple types to the (mergeable) values of those types through (non-mergeable) ordinals or identifiers. The premise of both rules assert this expectation on R by constraining its tuple type ρ to be of the form $\overline{T} \times \overline{\tau}$, where τ stands for a mergeable type. An example of such an R is the R_{kv} relation over a map v that relates its keys to mergeable values. The Rel-Merge-1 requires a tuple $(\overline{k}, \overline{s})$ to be present in the merged relation if $\overline{k}$ is related to $\overline{x}$, $\overline{y}$, and $\overline{z}$ of type $\overline{\tau}$ respectively by the (extended) relations $R(l)$, $R(v_1)$, and $R(v_2)$, and each s_i is the result of merging x_i, y_i, and z_i as per the merge semantics of τ_i (captured by ϕ_{τ_i}). The rule thus composes the merge specification ϕ_T of T using the merge specifications $\phi_{\overline{\tau}}$ of its constituent mergeable types $\overline{\tau}$. Using the extended relation R_+ instead of R for l, v_1, and v_2 lets us cover the case where $\overline{k}$ is related to something in one (resp. two) of the three versions, but is left unrelated in the remaining two (resp. one) versions. The extended relation R_+ lets us assume a zero value for $\overline{x}$, $\overline{y}$, or $\overline{z}$, whichever is appropriate, in such cases. We also ensure that $\overline{k}$ *needs* to be related to something in the merged version by separately merging the sets of ordinals in each merging relation as captured by the constraint $\overline{k} \in R_k(l) \diamond R_k(v_1) \diamond R_k(v_2)$. The rule Rel-Merge-2 asserts the converse of the constraint added in Rel-Merge-1, effectively making the merge specification an exact specification like in Set-Merge. Thus, for instance, a merge function of a map cannot introduce new key-value pairs that cannot be derived from the existing pairs by merging their values.

Example 4.4. The merge specification presented earlier for a pair of counters can now be formally derived, albeit with a few minor changes: we use the R_{pair} relation instead of R_{fst} and R_{snd}, which assigns an explicit (integer) ordinal to each pair component:

```
let R_pair (x,y) = {(1,x), (2,y)}
```

The type specification is $R_{pair} : \{\nu : \texttt{counter} * \texttt{counter}\} \to \mathcal{P}(\texttt{int} \times \texttt{counter})$. The tuple type is of the form $T \times \tau$, where T is not mergeable and τ is mergeable (an ordinal type can be defined separately from integers to be non-mergeable). Applying Rel-Merge-1 and Rel-Merge-2 rules yields the following merge specification for counter pairs (simplified for presentation):

$$\begin{array}{rcl} \phi_{c\times c} & = & \forall(k:\texttt{int}).\forall(x,y,z,s:\texttt{counter}).\ (k,x) \in R_{pair}(l)\ \wedge\ (k,y) \in R_{pair}(v_1) \\ & & \qquad \wedge\ (k,z) \in R_{pair}(v_2)\ \wedge\ \phi_c(x,y,z,s) \Rightarrow (k,s) \in R(v) \\ & \wedge & \forall(k:\texttt{int}).\forall(s:\texttt{counter}).\ (k,s) \in R_{pair}(v) \Rightarrow \exists(x,y,z:\texttt{counter}).\ (k,x) \in R_{pair}(l) \\ & & \qquad \wedge\ (k,y) \in R_{pair}(v_1)\ \wedge\ (k,z) \in R_{pair}(v_2)\ \wedge\ \phi_c(x,y,z,s) \end{array}$$

To check that the above is indeed a correct merge specification for counter pairs, one can observe that a function that directly implements this specification would correctly merge the example in Fig. 10.

5 DERIVING MERGE FUNCTIONS

We have thus far focused on deriving a merge specification for a data type, given the type specification of its characteristic relations. We now describe how to synthesize a function that operationalizes the specification, given these relation definitions. The synthesis problem is formalized thus:

Definition 5.1 (Merge Synthesis Problem). Given a data type T, a function α that computes the characteristic relations for values of T, a function γ that maps the characteristic relations back to instances of T, and a (derived) merge specification ϕ_T of T expressed in terms of its characteristic relations, synthesize a function F such that for all l, v_1, and v_2 of type T, $\phi_T(l, v_1, v_2, F(l, v_1, v_2))$ holds.

The synthesis process is quite straightforward as the expressive merge specification ϕ_T already describes what the result of a relational merge should be. For each FOL constraint φ in ϕ_T that specifies the necessary tuples in the merged relation (i.e., of the form $\ldots \Rightarrow \overline{x} \in R(v)$ or $\ldots \Leftrightarrow \overline{x} \in R(v)$ in Fig. 12), we describe its operational interpretation $[\![\varphi]\!]$ that *computes* the merged relation in a way that satisfies the constraint. We start with the simplest such φ, which is the constraint added to ϕ_T by Set-Merge. Recall that α is a pair-wise composition of characteristic relations of type T (i.e., $\alpha = \lambda x. \overline{R}(x)$). Let R be a characteristic relation, which we obtain by projecting from α, and let `r_l`, `r_v1`, and `r_v2` be variables denoting the sets $R(l)$, $R(v_1)$, and $R(v_2)$, resp. Using these definitions, we translate the Set-Merge constraint almost identically as shown below:

$$[\![\forall(\overline{x}:\overline{T}).\ \overline{x} \in (R(l) \diamond R(v_1) \diamond R(v_2)) \Leftrightarrow \overline{x} \in R(v)]\!] \quad = \quad \texttt{r\_l} \ \diamond\ \texttt{r\_v1} \ \diamond\ \texttt{r\_v2}$$

Order-Merge-1 can be similarly operationalized. One aspect that needs attention is the intersection with the set ρ denoting the tuple space of R. Since ρ could be composed of an infinite set like `int`, intersection with ρ cannot be naïvely interpreted. Instead, we synthesize a Boolean function $\mathbb{B}_\rho$

that returns `true` for elements present in the set ρ, and implement the intersection in terms of a `Set.filter` operation that filters a set to contain only those elements that satisfy this predicate:

$$[\![\forall(\overline{x} : \overline{T}).\ \overline{x} \in (R(l) \diamond R(v_1) \diamond R(v_2)\ \cap\ \rho) \Rightarrow \overline{x} \in R(v)]\!] \quad = \quad$$

```
let x = r_l ⋄ r_v1 ⋄ r_v2 in
Set.filter 𝔹ρ x
```

Rel-Merge-1 covers the interesting case of compositional merges. In this case, the tuples in R have a sequence of ordinals or identifiers ($\overline{k} : \overline{T}$, which we call *keys*) followed by values of mergeable types ($\overline{\tau}$). Each τ_i is required to have a zero value 0_i for which each characteristic relation has to evaluate to $\emptyset$. In practice, this is enforced by requiring the module `M` that defines τ_i (i.e., `M.t` = τ_i) to have a value `empty:t`, and checking if R(`empty`) evaluates to $\emptyset$ for each R. Since τ_i is a mergeable type, its implementation `M` should contain a `merge` function for τ_i. The R_+ definition used by Rel-Merge-1 effectively *homogenizes* the keys of $R(l)$, $R(v_1)$, and $R(v_2)$, mapping new keys to `empty`. The values with the corresponding keys are then merged using `M.merge` to compute the key-value pairs in the merged relation. Fig. 13 shows the operational interpretation. For brevity, we assume R to be a binary relation relating a single key to a value. `Set.map` is the usual map function with type: `'a set` $\rightarrow$ `('a` $\rightarrow$ `'b)` $\rightarrow$ `'b set`.

```
let ks_r_l = Set.map fst r_l in
let ks_r_v1 = Set.map fst r_v1 in
let ks_r_v2 = Set.map fst r_v2 in
let ks = ks_r_l ⋄ ks_r_v1 ⋄ ks_r_v2 in
let zero = M.empty in
let r_l' = r_l ∪
    (ks - ks_r_l) × {zero} in
let r_v1' = r_v1 ∪
    (ks - ks_r_v1) × {zero} in
let r_v2' = r_v2 ∪
    (ks - ks_r_v2) × {zero} in
Set.map (fun (k,x) ->
  let (x,y,z) =
    (r_l(k), r_v1(k), r_v2(k)) in
  let s = M.merge x y z in
  (k,s)) ks
```

Fig. 13. Operational interpretation of the constraint imposed by Rel-Merge-1 rule from Fig. 12

The operational interpretation of derivation rules from Fig. 12 let us merge characteristic relations. Applying the concretization function γ on merged relations maps the relations back to the concrete domain, thus yielding the final merged value. Letting ⧫ denote relational merges as described above, the whole process can be now succinctly described:

```
let merge l v1 v2 = γ(α(l) ⧫ α(v1) ⧫ α(v2))
```

5.1 Concretizing Orders

The concretization function γ_{ord} aids in the process of concretizing orders, such as R_{ob}, into data structures. An inherent assumption behind γ_{ord} is that there is a single ordering relation (e.g., R_{ob} or R_{to}) that guides concretization. This is indeed true for the data structures listed in Table. 1. The ordering relation is required to be ternary, and is naturally interpreted as a directed graph G where each tuple (u, a, v) denotes an edge from u to v with a label a. Binary orders, such as R_{ob}, are a special case where the labels are all same[6] Concretization works in the context G. The first step is transitive reduction, where an edge (u, v) is removed if there exists edges (u, v') and (v', v) for some v'. A transitively reduced graph is said to be *conflict-free* if for every vertex u, there do not exist more than n edges with the same label a, where n is determined uniquely for each data structure. The basis for defining n is the condition that the ordering relation computed by the data structure's abstraction function α has to be conflict-free for any instance of the data structure. For instance, $n = 1$ for lists and trees as, for any list l and tree t, there do not exist two or more adjacent edges

[6]We shorten (u, a, v) in the presentation to (u, v) when appropriate.

with the same label in the transitively-reduced graph of $R_{ob}(l)$ and $R_{to}(t)$, respectively. On the other hand, $n = 2$ for the binary heap data structure as the transitively reduced *ancestor* relation $R_{ans}(h)$of a heap h never contains more than two adjacent edges with the same label.

The second step of concretizing an ordering relation resulting from a merge is to check if the relation is conflict-free. If it indeed is, then its (transitively-reduced) graph G is already isomorphic to the merged data structure, which can be recovered from G by simply traversing the graph starting from its root while applying appropriate data type constructors (as demonstrated for list/queue data type in Fig. 9a). On the other hand, if the graph G has conflicts, i.e., it has more than n adjacent edges with the same label, then the conflicts need to be resolved before G becomes isomorphic to a valid instance of the data structure. Conflicts that may arise due to a merge are resolved by *inducing* an order among the distinct vertices of conflicting edges using a provided *arbitration relation*. For instance, consider a case where there are two conflicting edges, (u, a, v) and (u, a, v'), in the transitively-reduced ordering graph G of a data structure whose $n = 1$. The conflict is resolved by inducing an order between v and v' using the provided arbitration relation *ord*, which adds either a (v, b, v') or (v', b, v) edge for some label b. Transitive reduction at this point removes one of the conflicting edges, thus resolving the conflict. This process is repeated until all conflicts are resolved, at which point the graph is isomorphic to the merged data structure, and the latter can be reconstructed by simply traversing the former. The process is illustrated for the R_{to} relation shown in Fig. 14. On the left hand side of the figure is the graph G of the R_{to} relation that is obtained by merging the R_{to} relations of two trees. Both trees add d and e (resp.) as a right child to b, which results in tuples (b, R, d) and (b, R, e) in R_{to}. The tuples translate into conflicting edges shown (with colored vertices) in G. To resolve conflicts and generate an R_{to} relation consistent with the tree structure, we can invoke γ_{ord} with (for instance) the following definition of `ord`:

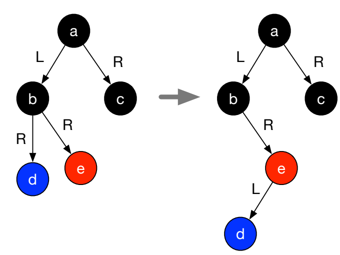


Fig. 14. Resolving conflicts while concretizing R_{to}

```
let ord x y = if x<y then (y,L,x) else (x,L,y)
```

Assuming $d < e$, `ord` adds an edge (e, L, d), which lets (b, R, d) to be removed during transitive reduction, resulting in the graph shown on the right, which is clearly a tree. The γ_{ord} helper function thus aids in reifying ordering relations into concrete data structures. It is available to MRDT developers as a library function named `concretize_order` that takes the set representation of an ordering relation and returns a graph isomorphic to a data structure whose ordering relation subsumes the given set. An arbitration order and an n value (as described above) are also expected to help `concretizes_order` resolve conflicts and extend the ordering relation as necessary.

Having described the abstraction functions (Sec. 3), relational merge derivation (Sec. 4 and Sec. 5), and subsequent concretization (Sec. 5.1), we can now put these together to obtain a complete picture of how MRDTs are *derived* from ordinary OCaml data types. Examples are shown in Figures 15 and 16. The syntax is as close to the real syntax as possible barring minor technical differences.

Fig. 15a shows how `MRBSet.t` – a mergeable replicated variant of a set data type based on a Red-Black binary search tree, is derived from `RBSet.t` – its sequential non-replicated counterpart. MRBSet developer is expected to write the module shown in Fig. 15a. The signature `MERGEABLE` indicates that the module is that of an MRDT, i.e., it defines a data type and a `merge` function on the data type[7]. In Fig. 15a, we reproduce `'a RBSet.t` definition as `'a MRBSet.t` for perspicuity, which otherwise would have followed by including `RBSet` module in `MRBSet`. In general, `MERGEABLE`

[7]There are nuances to `MERGEABLE` signature to accommodate data types with multiple type variables (e.g., `'a t`, `('a,'b) t` etc.), and to expose data type-specific library functions. Such nuances are not discussed here.

```
module G = DiGraph
module E = G.Edge
module MRBSet:MERGEABLE =
struct
  (* 'a RBSet.t reproduced below *)
  type 'a t =
    | Black of 'a t * 'a * 'a t
    | Red of 'a t * 'a * 'a t
    | Empty [@@deriving merge]

  let r_mem t = Set.of_list @@
      RBset.elements t

  let abstract t = r_mem t

  let concretize s_mem = Set.fold
      RBSet.add s_mem Empty
end
```

(a) Mergeable Set

```
module MRBMap(V:MERGEABLE):MERGEABLE =
struct
  (* ('a,V.t) RBMap.t reproduced below *)
  type 'a t =
    | Black of 'a t * 'a * V.t * 'a t
    | Red of 'a t * 'a * V.t * 'a t
    | Empty [@@deriving merge]

  let r_mem t = Set.of_list @@
      RBMap.all_pairs t

  let abstract t = r_mem t

  let concretize s_mem = Set.fold
      (fun (k,v) t -> RBMap.add k v t)
      s_mem Empty
end
```

(b) Mergeable Map

Fig. 15. Red-Black Tree-based Set and Map data structure annotated with their respective abstraction and concretization functions. The corresponding merge functions are derived automatically.

module definitions include the corresponding non-mergeable (ordinary) modules, thus highlighting that mergeable data types are extensions of ordinary OCaml data types with merge logic. The merge logic is derived automatically when prompted by the `@@deriving merge` annotation, which was added to the OCaml syntax with help of a PPX extension [PPX 2017] as a part of our development (Sec. 6 contains details). The merge derivation looks for abstraction and concretization functions in the module definition, and uses them to derive a merge function as described in previous sections and summarized in Fig. 3. The abstraction function is simply is a pair-wise composition of characteristic relations. In Fig. 15a, the abstraction function (named `abstract`) for Red-Black binary search tree is composed of a single characteristic relation[8] – R_{mem}, whose definition (named `r_mem` in Fig. 15a) is is similar to the corresponding definition for list/queue (R_{mem} in Fig. 7), except that it is uses concrete syntax and standard library functions in place of abstract notations. The concretization function `concretize` reconstructs a tree from its membership relation by repeatedly inserting elements starting with an empty tree. Assuming that `RBSet`'s `insert` function is correct, i.e, it returns a valid Red-Black tree, the function `concretize` is also guaranteed to be correct.

Fig. 15b shows a minor variation of `MRBSet` – an implementation of mergeable replicated map, `MRBMap`, based on Red-Black binary search tree. Unlike `MRBSet`, `MRBMap` is a functor parametric on on the type of values (`V.t`), which is also required to be mergeable. The merge function derived in this case would be a composed of `V.merge` as described by the composition rule Rel-Merge-1 from Fig. 12.

In Fig. 16, we show how one would write a mergeable binary tree module, `MBinaryTree`, starting from `BinaryTree`. Like in Fig. 15, we reproduce the data type definition for clarity. Characteristic relations include a *membership* relation `r_mem` and a *tree-order* relation `r_to`, which is essentially the R_{to} relation (Sec. 3) in concrete syntax. The abstraction function `abstract` is simply a pair-wise

[8]Recall from Sec. 3 that *membership* relation is enough to determine the extensional equality of binary search trees, balanced or otherwise.

```
module G = DiGraph
module E = G.Edge
module MBinaryTree:MERGEABLE =
struct
  (* 'a BinaryTree.t reproduced
      below *)
  type 'a t =
    | N of 'a t * 'a * 'a t
    | E [@@deriving merge]

  let r_mem t = Set.of_list @@
      BinaryTree.elements t

  let r_to = function
    | E -> Set.empty
    | N(l, x, r) ->
      let s1 = Set.times3
        (Set.singleton x)
        (Set.singleton `Left)
        (r_mem l) in
      let s2 = Set.times3
        (Set.singleton x)
        (Set.singleton `Right)
        (r_mem r) in
      List.fold_left Set.union
        (r_to l)
        [s1; s2; r_to r]

  let abstract t =
      (r_mem t, r_to t)
```

```
  let concretize (s_mem, s_to) =
    if Set.is_empty s_to
    then
      match Set.elements s_mem with
        | [] -> E
        | [x] -> N(E, x, E)
        | _ -> error()
    else
      let g = concretize_order
                          s_to ~n:1 (<) in
      let rec mk_tree x = match G.succ x with
        | [] -> N(E,x,E)
        | [y] ->
          (match E.label @@ G.find_edge g x y
           with
           | `Left -> N(mk_tree y, x, E)
           | `Right -> N(E, x, mk_tree y))
        | [y;z] ->
          (match (E.label @@ G.find_edge x y,
                  E.label @@ G.find_edge x z)
           with
           | (`Left, `Right) ->
                N(mk_tree y, x, mk_tree z)
           | (`Right, `Left) ->
                N(mk_tree z, x, mk_tree y))
        | _ -> error() in
      mk_tree (G.root g)
end
```

Fig. 16. Binary Tree data structure annotated with abstraction and concretization functions. The merge function is derived automatically.

composition of these relations. Along with the abstraction function, the developer is expected to write the concretization function `concretize`, which in this case uses the aforementioned library function `concretize_order` to concretize the R_{to} ordering relation. If the set representing the tree-order, `s_to` is empty, then there is at most one element in the tree. The function `concretize` returns an appropriate tree in such case. Otherwise it calls `concretize_order`, providing < as the arbitration order, to obtain a graph isomorphic with the final (merged) tree. Note that $n = 1$ in this case as each tree node has no more than one left (‘`Left`) or right (‘`Right`) adjacent nodes in (transitively-reduced) R_{to} relation. The remainder of the `concretize` function following the call to `concretize_order` is the definition of `mk_tree`, which simply traverses the graph g returned by `concretize_order`, and constructs an isomorphic binary tree.

6 IMPLEMENTATION

The infrastructure necessary to implement MRDTs, and execute them in a asynchronously replicated setting has been developed in terms of three major components collectively referred to as Quark. The first component of Quark is a library of MRDT modules corresponding to basic data structures, such as lists and binary trees, along with a collection of signatures (e.g., MERGEABLE) and functions (e.g., concretize_order), that aid in the development of new MRDTs. The second component is an OCaml compiler extension, developed modularly using PPX [PPX 2017], that performs a dual function. Firstly, aided by module signatures and compiler directives (e.g., @@deriving merge), the PPX extension identifies the OCaml type definitions of MRDTs, along with their abstraction and concretization functions, and composes them together, as described in Sections 4 and 5, to generate the corresponding merge functions. While adding a merge function makes an OCaml data type *mergeable*, it is however not *replicated* for replication requires addressing low-level concerns, such as serialization, network fault tolerance etc. The third component of Quark is a content-addressable distributed storage abstraction, called the Quark store, that addresses these concerns, and the secondary function of the PPX extension is to generate the code that translates between the high-level (OCaml) representation of a data type, and its corresponding low-level representation in the Quark store. The schematic diagram of this workflow is shown in Fig. 17. The following sub-section describes the Quark store in detail.

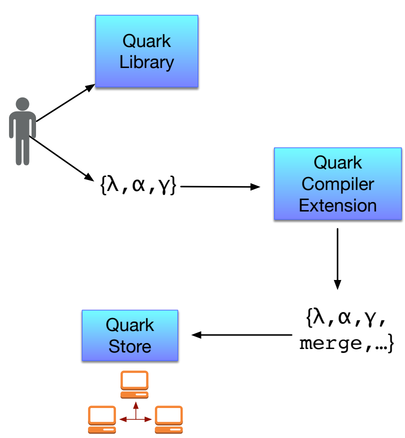


Fig. 17. Quark architecture: Programmer extends OCaml data types (λ) with abstraction (α) and concretization (γ) functions. Quark compiler extension generates merge and low-level code to interface with the Quark store, which handles replication.

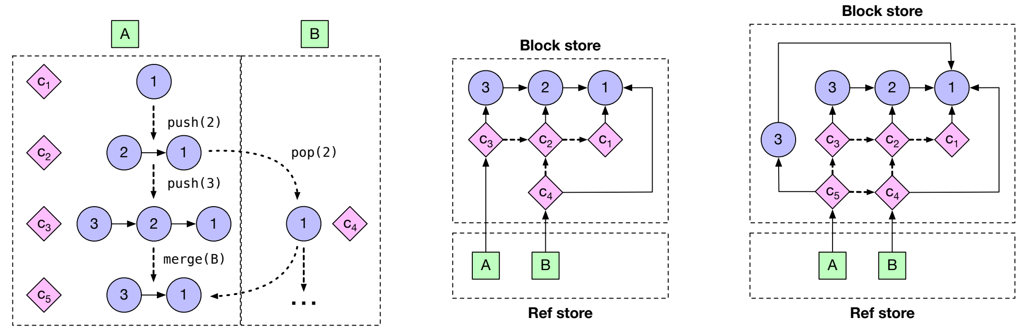
6.1 Quark store

The key innovation of the Quark store is the use of a storage layer that exposes a Git-like API, supporting common Git operations such as cloning a remote repository, forking off branches and merging branches using a three-way merge function. Quark builds on top of these features to achieve a fault-tolerant, highly-available geo-replicated data storage system. For example, creating a new replica is realized by cloning a repository, and remote pushes and pulls are used to achieve inter-replica communication. Quark store also supports a variety of storage backends including in-memory, file systems and fast key-value storage databases, and distributed data stores. We have built a programming model around Quark store's Git-like API to build distributed applications using MRDTs, which is discussed elsewhere [Kaki et al. 2019].

The main challenge in realizing MRDTs as a practical programming model is the need to efficiently store, compute and retrieve the LCA given two concurrent versions. Quark uses a content-addressable block store for storing the data objects corresponding to concurrent versions of the MRDT as well as the history of each of the versions. Given that any data structure is likely to share most of the contents with concurrent and historical versions, content-addressability maximizes sharing between the different versions.

Consider the example presented in Fig. 18a which shows an execution trace on a stack MRDT. There are two versions A and B. Version B is forked off from A and is merged on to A. Since B pops the element 2, it is no longer present in the merged version. B is of course free to further evolve concurrently with respect to A. The diamonds represent the *commits* that correspond to each historical version of the stack and circles represent data objects.

Fig. 18b and Fig. 18c represent the layout of the Quark store before and after the merge. Quark uses a content-addressable append-only *block* store for data and commit information. Objects in the block store are addressed by the content of their hashes. Correspondingly, links between the objects are hashes of the contents of the objects. The reference to the two versions A and B are



(a) An execution trace for stack MRDT. (b) Quark store before merging the commit c4 into c3. (c) Quark store after merging the commit c4 into c3.

Fig. 18. The behavior of Quark content-addressable storage layer for a stack MRDT. A and B are two versions of the stack MRDT. Diamonds represent the commits and circles represent data objects.

Data Structure	Description
Set	From OCaml stdlib. Implemented using AVL Trees.
Heap	Okasaki's Leftist Heap [Okasaki 1998]
RBSet & RBMap	Okasaki's Red-Black Tree with Kahrs's deletion [Kahrs 2001]
Graph	From the Functional Graph Library [Erwig 2001; Functional Graph 2008]
List	Standard implementation of a cons list
Queue	From OCaml stdlib.
Rope	A data structure for fast string concatenation from [Boehm et al. 1995]
TreeDoc	A CRDT for collaborative editing [Preguica et al. 2009] but without replication awareness.
Canvas	A data structure for collaborative freehand drawing

Table 2. A description of data structure benchmarks used in the evaluation.

stored in a mutable *ref* store. The versions point to a particular commit. The commits in turn may point to parent commits (represented by dashed lines between the diamonds), and additionally may point to a single data object. Data objects stored in the block store may only point to other data objects.

Observe that in Fig. 18b, there is only one copy of the stack which is shared among both the concurrent and historical versions. Notice also that the branching structure of the history is apparent in the commit graph. In this example, we are merging the commits c_3 and c_4. Quark traverses the commit graph to identify the lowest common ancestor c_2 and fetches the version of the stack that corresponds to the commit. After the merge, a new commit object c_5 is added along with a new data object for 3 which points to the existing data object 1 in the block store. The version ref for A in the ref store is updated to point to the new commit c_5. As our experimental results indicate, the use of a content-addressable store makes it efficient to implement MRDTs in practice.

7 EVALUATION

We have evaluated our approach implemented in Quark on a collection of data structure and applications.

7.1 Data Structure Benchmarks

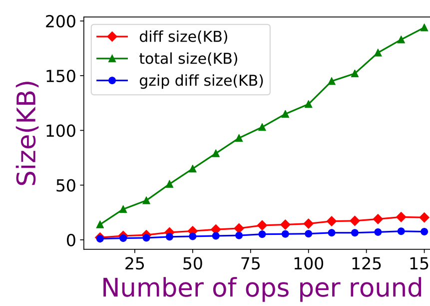
The summary of data structures that we consider is given in Table. 2. Some of these benchmarks are taken directly from the standard library, and span over 500 lines of code defining tens of functions.

Quark lets these data structures be used as MRDTs *as such* with just a few (less than 10) additional lines of code to define a relational specification and derive merges. To evaluate how these MRDTs fare under the version control-inspired asynchronous model of replication that is central to our approach, we constructed experiments that specifically answer two questions:

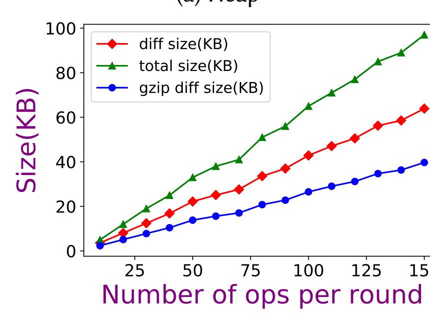
(1) How does the size of the *diff* between versions change relative to the size of the data structure as the latter grows over time, and
(2) How much is the overhead of merge relative to the computational time on the data structure.

As replicas periodically sync, they perform three-way merges to reconcile their versions, which requires both remote and local versions be present. Since transmitting a version in its entirety for each merge operation is redundant and inefficient, Quark computes the diff between the current version and the last version that was merged (using the content-addressable abstraction from Sec. 6), and transmits this diff instead. Smaller diff size (relative to the total size of the data structure) indicates that the data structure is well-suited to be a mergeable type, and the corresponding MRDT can be efficiently realized over Quark.

To measure the diff size relative to the data structure size for each data type, we conduct controlled experiments where a single client performs a series of randomly distributed operations on the data structure and *commits* a version. The exact nature of operations is different for different data types (insertion and deletion for a tree, `remove_min` for a (min) heap etc), but in general the insertion-deletion split is 75%-25%, which lets the data structure grow over time. Since a client can perform any number of operations before synchronizing, we conduct experiments by gradually increasing the number of operations between two successive commits (called a *round*) in steps of 10 from 10 to 150. For every experiment, at the end of each round, we measure the size of the data structure and the diff size between the version being committed and the previous version (computed by Quark's content-addressable abstraction). The experiments were conducted for all the data structures listed in Table. 2, and the results for the best and worst performing ones (in terms of the relative diff size are shown in Fig. 19. The graphs also show the size of the gzipped diff size since this is the actual data transmitted over the network by Quark.



(a) Heap



(b) List

Fig. 19. Diff vs total-size for Heap and List

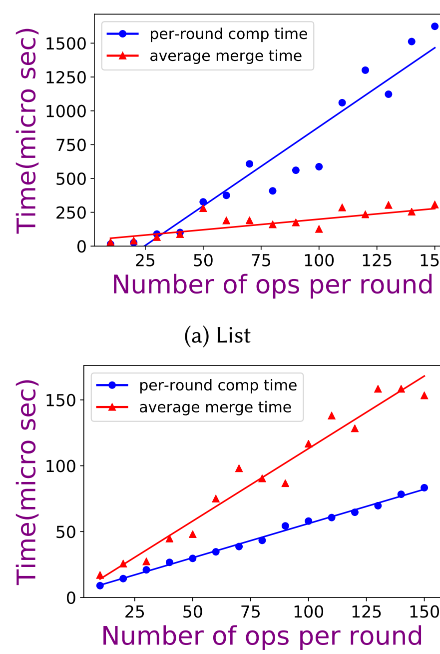
Heap performs the best, which is not surprising considering that its tree-like structure lends itself to natural sharing of objects between successive versions. Inserting a new element into a heap, for instance, creates new objects only along the path from the root to that element, leaving the rest same as the old heap (hence shared). Other tree-like structures, including red-black and AVL trees, ropes, and document trees, also perform similarly, with their results being only slightly worse than heap. List performs the worst, again an unsurprising result considering that its linear structure is not ideal for sharing. For instance, adding (or removing) an element close to the end of a list creates a new list which only shares a small common suffix with the previous list. Nonetheless, as evident from Fig. 19b, its diff size on average is still less than the

Application	SLOC	Types	Txns	DB Size (MB)	Avg. diff size (KB)
TPC-C	1081	9	3	37.9 - 47.19	19.37
TPC-E	1901	19	5	93.3 - 124.30	22.89
RUBiS	998	8	5	9.69 - 11.06	2.62
Twissandra	870	5	4	1.34 - 3.69	4.612

Table 3. Application Benchmarks

total size of the list, and grows sub-linearly relative to the latter. In summary, diff experiments show that version control-inspired replication model can be efficiently supported for common data structures by transmitting succinct diffs over the network rather than entire versions.

To measure the overhead of merges relative to the computational time, we performed another set of experiments involving three replicas, each serving a client, connected in a ring layout over a (virtual) network with latency distributed uniformly between 10ms and 200ms. Each client behaves the same as with the previous (diff) experiments, except that there is a synchronization that follows the commit at the end of each round that merges the committed version with the remote version and returns the result (remote version comes from the replica upstream in the ring). We record the time spent merging the versions ("merge time"), and also the time spent performing operations in each round. As before, we gradually increase the number of operations per round, which inevitably increases the computational time and *may* increase the merge time depending on the data structure. A better performing data structure is one whose merge time increases sub-linearly, or remains constant, with the increase in computation time. A worse performing one is where merge time increases linearly or more. The results for best and worst performing data structures. in this sense, are shown in Fig. 20. A list performs the best here as its insertion and deletion operations are $O(n)$, making its computational time degrade faster with the increase in number of operations (kn time for computation vs n time for merge in a round of k operations). Red-Black tree (-based set) performs the worst as its $O(\log(n))$ operations are asymptotically faster than $O(n)$ merge. Nonetheless, both metrics are the same order of magnitude, which is several orders of magnitude less than the mean network latency. Moreover, since MRDTs do not require any coordination, synchronization (hence merges) can always be performed off the fast path, thus avoiding any latency overhead due to a merge.



(a) List

(b) Red-Black Tree

Fig. 20. Computation vs merge time for List and Red-Black Tree

7.2 Application Benchmarks

We have also implemented four large application benchmarks by composing several mergeable data types derived from their relational specifications. Table 3 lists their attributes, and the summary of diff experiments we ran on them.

TPC-C and TPC-E are well-known online transaction processing (OLTP) benchmarks in the database community [TPC 2018]. TPC-C emulates a warehouse application consisting of multiple *warehouses* with multiple *districts*, serving *customers* who place *orders* for *items* in *stock*. Each such application type (e.g., `customer`) is implemented as a record with multiple fields, some of which are mergeable. For instance, `c_ytd_payment` field of `customer` record is a mergeable counter recording the customer's year-to-date payment. Such records themselves are made mergeable through a relational specification similar to that of a pair type (Sec. 4). In TPC-C, there are a total of 9 such record types (Types column in Table 3). A mergeable red-black tree-based map ("RBMap") performs the role of a database table in our case. The database, which is otherwise a collection of (named) tables, is simply another mergeable record in our case that relates named fields to RBMaps corresponding to each table. The type design is shown in Fig. 21. TPC-C has 3 transactions that we implemented in our model as functions that map one version of the database to other, returning a result in the process. Concretely:

```
type warehouse = {w_id: id; w_ytd: counter}
type customer =
  {c_w_id: id; c_d_id: id; c_id: id;
   c_name: string; c_bal: counter;
   c_ytd_payment:counter;}
type db =
  {warehouse_table: (id, warehouse) rbmap;
   customer_table: (id*id*id, customer) rbmap;
   ...}
```

Fig. 21. Composition of mergeable data structures in TPC-C (simplified for presentation). Database (db) is composed of mergeable RBMap, which is composed of application-defined types, and ultimately, mergeable counters.

```
type 'a txn = db -> 'a*db
```

Since the database is not in-place updated, transactions are isolated by default. A transaction commit translates to the commit of a new version of type db, which is then merged with concurrent versions of db created by concurrently running transactions. We evaluated our TPC-C application composed of mergeable types by first populating the database (db) as per the TPC-C specification, and then performing the diff experiments as described above with 500 transactions. The database size grew from 37.9MB to 47.19MB during the experiment (DB Size column in Table 3), with the average size of diff due to each transaction being constant around 20KB (Avg. diff size column).

We have implemented three other applications, including the TPC-E and RUBiS [RUBiS 2014] benchmarks, and a twitter-clone called Twissandra [Twissandra 2014]. Our experience of building and experimenting with these applications has been consistent with our earlier observations that (a). complex data models of applications can be realized by composing various mergeable data types (b). the resultant application state lends itself to efficient replication under Quark's replication model with well-defined and useful semantics.

8 RELATED WORK & CONCLUSION

Our idea of versioning state bears resemblance to Concurrent Revisions [Burckhardt et al. 2010, 2012], a programming abstraction that provides deterministic concurrent execution, and Tardis [Crooks et al. 2016], a key-value store that also supports a branch-and-merge concurrency control abstraction. However, unlike these previous efforts which provide no principled methodology for constructing merge functions, or reasoning about their correctness, our primary contribution is in the development of a type-based compositional derivation strategy for merge operations over sophisticated inductive data types. We argue that the formalization provided in this paper significantly alleviates the burden of reasoning about state-based replication. Furthermore, the integration of a version-based mechanism within OCaml allows a degree of type safety and enables profitable use of polymorphism not available in related systems.

[Burckhardt et al. 2015] also presents an operational model of a replicated data store that is based on the abstract system model presented in [Burckhardt et al. 2014]; their design is similar to the model described in [Sivaramakrishnan et al. 2015]. In these approaches, coordination among replicas involves transmitting operations on replicated objects that are performed locally on each replica. In contrast, Quark fully abstracts away such details - while programmers must provide abstraction and concretization functions that map datatype semantics to the language of relations and sets, the reasoning principles involved in performing this mapping are not dependent upon any specific storage or system abstraction, such as eventual consistency [Burckhardt et al. 2014; Shapiro et al. 2011b]. Given a library of predefined functions for common data types, and a methodology for deriving their composition, the burden of migrating sequential data types to a replicated setting is substantially reduced.

Modern distributed systems are often equipped with only parsimonious data models (e.g., key-value model) that complicate program reasoning, and make it hard to enforce application integrity. Some authors [Bailis et al. 2013c] have demonstrated that it is possible to *bolt-on* high-level consistency guarantees (e.g., causal consistency) [Bouajjani et al. 2017; Lloyd et al. 2011] as a *shim layer* service over existing stores, but these approaches do not consider integration of these services within the type abstractions provided by a high-level client-facing language.

A number of verification techniques, programming abstractions, and tools have been proposed to reason about program behavior in a geo-replicated weakly consistent environment. These techniques treat replicated storage as a black box with a fixed pre-defined consistency model [Alvaro et al. 2011; Bailis et al. 2014; Balegas et al. 2015; Gotsman et al. 2016; Li et al. 2014b, 2012b]. On the other hand, compositional proof techniques and mechanized verification frameworks have been developed to rigorously reason about various components of a distributed data store [Kaki et al. 2017; Lesani et al. 2016; Wilcox et al. 2015]. Quark is differentiated from these efforts in its attempt to mask details related to distribution but unnecessary for defining meaningful (convergent) merge operations. An important by-product of this principle is that Quark does not require algorithmic restructuring to transplant a sequential or concurrent program to a distributed, replicated setting; the only additional burden imposed on the developer is the need to provide abstraction and concretization functions for compositional data types that can be used to derive well-formed merge functions, actions that we have demonstrated are significantly simpler than reasoning about weakly-consistent behaviors.

Quark shares some resemblance to conflict-free replicated data types (CRDT) [Shapiro et al. 2011a]. CRDTs define abstract data types such as counters, sets, etc., with commutative operations such that the state of the data type always converges. Unlike CRDTs, the operations on mergeable types in Quark need not commute and the reconciliation protocol is defined by merge functions derived from the semantics of the data types whose instances are intended to be replicated. The lack of composability of CRDTs is a major hindrance to their utility that forms an important point of distinction with the approach presented here. A CRDT's inability to take advantage of provenance information (i.e., LCAs) is another important drawback. As a result, constructing even simple data types like counters are more complicated using CRDTs [Shapiro et al. 2011a] compared to their realization in Quark.

Finally, on the language design front, there have been approaches where relations feature prominently, e.g., Datalog [Maier et al. 2018] and Prolog [Bowen 1979]. In such languages, data is represented as "facts" described by relations, and computation on data is structured as relational queries. In contrast, Quark does not advocate a new style of programming, but rather uses relations to augment capabilities of data structures in an existing model of programming. Relations have been employed to reason about programs and data structures, for example in shape analysis [Chang and

Rival 2008; Jeannet et al. 2010; Kaki and Jagannathan 2014], but the focus is always on using relations to prove correctness of programs, not on using them as convenient run-time representations.

REFERENCES

Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. 249–260.

Amazon SQS [n. d.]. https://aws.amazon.com/sqs Amazon Simple Queue Service.

Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013a. Highly Available Transactions: Virtues and Limitations. *PVLDB* 7, 3 (2013), 181–192.

Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509

Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013b. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 761–772. https://doi.org/10.1145/2463676.2465279

Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013c. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 761–772. https://doi.org/10.1145/2463676.2465279

Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting the Consistency back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer System (EuroSys '15)*. Bordeaux, France. http://lip6.fr/Marc.Shapiro/papers/putting-consistency-back-EuroSys-2015.pdf

Hans-J. Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: An Alternative to Strings. *Softw. Pract. Exper.* 25, 12 (Dec. 1995), 1315–1330. https://doi.org/10.1002/spe.4380251203

Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On Verifying Causal Consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 626–638. https://doi.org/10.1145/3009837.3009888

Kenneth A. Bowen. 1979. Prolog. In *Proceedings of the 1979 Annual Conference (ACM '79)*. ACM, New York, NY, USA, 14–23. https://doi.org/10.1145/800177.810020

Brewer 2013. http://highscalability.com/blog/2013/5/1/myth-eric-brewer-on-why-banks-are-base-not-acid-availability.html Myth: Eric Brewer on Why Banks are BASE Not ACID - Availability Is Revenue.

Eric Brewer. 2000. Towards Robust Distributed Systems (Invited Talk).

Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. 2010. Concurrent Programming with Revisions and Isolation Types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 691–707. https://doi.org/10.1145/1869459.1869515

Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. 2012. Cloud Types for Eventual Consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 283–307. https://doi.org/10.1007/978-3-642-31057-7_14

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 271–284. https://doi.org/10.1145/2535838.2535848

Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP '15)*. Prague, Czech Republic. http://research.microsoft.com/pubs/240462/gsp-tr-2015-2.pdf

Bor-Yuh Evan Chang and Xavier Rival. 2008. Relational Inductive Shape Analysis. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 247–260. https://doi.org/10.1145/1328438.1328469

Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. 2016. TARDiS: A Branch-and-Merge Approach To Weak Consistency. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1615–1628. https://doi.org/10.1145/2882903.2882951

Martin Erwig. 2001. Inductive Graphs and Functional Graph Algorithms. *J. Funct. Program.* 11, 5 (Sept. 2001), 467–492. https://doi.org/10.1017/S0956796801004075

Functional Graph 2008. A Functional Graph Library. http://hackage.haskell.org/package/fgl

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: Replication Coordination Analysis and Synthesis. *PACMPL* 3, POPL (2019), 74:1–74:32. https://dl.acm.org/citation.cfm?id=3290387

Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. 2010. A Relational Approach to Interprocedural Shape Analysis. *ACM Trans. Program. Lang. Syst.* 32, 2, Article 5 (Feb. 2010), 52 pages. https://doi.org/10.1145/1667048.1667050

Stefan Kahrs. 2001. Red-black Trees with Types. *J. Funct. Program.* 11, 4 (July 2001), 425–432. https://doi.org/10.1017/S0956796801004026

Gowtham Kaki and Suresh Jagannathan. 2014. A Relational Framework for Higher-order Shape Analysis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 311–324. https://doi.org/10.1145/2628136.2628159

Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. 2017. Alone Together: Compositional Reasoning and Inference for Weak Isolation. *Proc. ACM Program. Lang.* 2, POPL, Article 27 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158115

Gowtham Kaki, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Version Control Is for Your Data Too. In *3rd Summit on Advances in Programming Languages (SNAPL 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 136. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 8:1–8:18. https://doi.org/10.4230/LIPIcs.SNAPL.2019.8

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 357–370. https://doi.org/10.1145/2837614.2837622

Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014a. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 281–292. http://dl.acm.org/citation.cfm?id=2643634.2643664

Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014b. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 281–292. http://dl.acm.org/citation.cfm?id=2643634.2643664

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012a. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012b. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593

David Maier, K. Tuncay Tekle, Michael Kifer, and David S. Warren. 2018. Declarative Logic Programming. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, Chapter Datalog: Concepts, History, and Outlook, 3–100. https://doi.org/10.1145/3191315.3191317

Chris Okasaki. 1998. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA.

PPX 2017. PPX extension points. Accessed: 2017-01-04 10:12:00.

Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. 2009. A Commutative Replicated Data Type for Cooperative Editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems (ICDCS '09)*. IEEE Computer Society, Washington, DC, USA, 395–403. https://doi.org/10.1109/ICDCS.2009.20

RUBiS 2014. Rice University Bidding System. http://rubis.ow2.org/ Accessed: 2014-11-4 13:21:00.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Lecture Notes in Computer Science, Vol. 6976. Springer Berlin Heidelberg, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Lecture Notes in Computer Science, Vol. 6976. Springer Berlin Heidelberg, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

TPC 2018. http://www.tpc.org/information/benchmarks.asp TPC Benchmarks.

Twissandra 2014. Twitter clone on Cassandra. http://twissandra.com/ Accessed: 2014-11-4 13:21:00.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 357–368. https://doi.org/10.1145/2737924.2737958