Mergeable Types

ANONYMOUS AUTHOR(S)

Distributed applications often replicate data to improve availability and fault tolerance. But, the use of replication leads to a significantly more complex and onerous programming model. To ensure a sensible measure of consistency among different replicas of an object, applications are typically structured so that the operations applied to a replicated object on one node are subsequently communicated to other nodes where they can be re-applied locally. Oftentimes, the degree of restructuring and care necessary to transform a sequential or shared-memory concurrent application to a meaningful (replicated) distributed one can be substantial.

In this paper, we present a new interpretation of replicated types that overcome these concerns. Our declarative programming model called VML treats replication as a compositional form of versioning, in which consistency is enforced through programmer-supplied *merge* functions. Distributed state is now viewed in terms of a tree of immutable object versions, with merge actions used explicitly by the application or implicitly by the underlying runtime system to periodically reconcile local versions to yield a consistent global state. Our model frees the programmer from having to think of low-level operational notions related to distribution and replication, and therefore does not entail any restructuring of application logic to enable distributed programming.

We show how any ML datatype can be enriched to support mergeability, formalize a concurrency semantics based on such types, and describe an instantiation of OCaml equipped with these features. Our implementation is integrated with a content-addressable distributed storage layer that allows any OCaml program to be easily transformed into a composable, efficient, and highly-available distributed application.

CCS Concepts: •**Software and its engineering** → **Distributed programming languages; Data types and structures;** *Semantics;* •**Theory of computation** → *Type structures;*

Additional Key Words and Phrases: Replicated Datatypes, Eventual Consistency, Availability, CRDTs, Versions, ML

ACM Reference format:
Anonymous Author(s). 2017. Mergeable Types. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 25 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Real-world distributed programs are challenging to write and maintain because they often conflate two distinct mechanisms. The first concerns the expression of application logic - how do we define computations that are robust in the presence of distributed communication among concurrently executing threads of control? The second deals with system concerns - how do we express notions of visibility, replication, and consistency when the nodes participating in such computations may be geographically distributed and the networks that connect them unreliable? To simplify reasoning in such complicated environments, programming models often make strong assumptions on the guarantees provided by an implementation (such as serializability (Papadimitriou 1979) or strong consistency (Corbett et al. 2012; Dragojević et al. 2015)) that may inadvertently mask, restrict, or ignore important albeit unpleasant realities (e.g., network partitions (Brewer 2000; Gilbert and Lynch 2002)). When unwarranted assumptions are made, program behavior is often difficult to predict and verify. On the other hand, when these features are explicitly exposed to the programmer, e.g., by requiring that applications be written in terms of specialized distributed data structures (Burckhardt et al. 2012, 2014; Shapiro et al. 2011a) or control primitives (Alvaro et al. 2011a), simplicity, composability, and ease-of-reasoning can suffer.

In this paper, we consider how declarative abstractions can be used to overcome this tension to provide the best of both worlds: functional programs whose logical structure can be seamlessly transplanted to a distributed setting, while nonetheless being resilient to system realities such as network failure and latency. Our key insight is the development of a principled methodology that allows us to transform low-level operational details such as replication and consistency into high-level compositional abstractions over language-level data types.

To illustrate how the kinds of problems described above manifest in practice, consider how we might write a simple counter library (see Fig. 1a). A `Counter` supports two (update) operations - `add` and `mult` - that lets a non-negative integer value be added or multiplied to the counter, resp. Observe that the library is written in an

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

```
module Counter: sig
  type t
  val add: int -> t -> t
  val mult: int -> t -> t
  val read: t -> int
end = struct
  type t = int
  let add x v = v + (abs x)
  let mult x v = v * (abs x)
  let read v = v
end
```

(a) Counter library in OCaml

```
module Counter: sig
  type t
  type eff
  val add: int -> t-> eff
  val mult: int -> t -> eff
  val apply: eff -> t -> t
  val read: t -> int
end = struct
  type t = int
  type eff = Add of int
  let add x v = Add (abs x)
  let mult x v = Add (v * (abs x - 1))
  let apply (Add x) v = x + v
  let read v = v
end
```

(b) Counter library re-engineered for effect-based replication

idiomatic functional style, with no special reasoning principles needed to realize desired functionality. As long as applications use the library on a single machine, this implementation behaves as expected. However, if the library is used in the context of a more sophisticated application, say one whose computation is distributed among a collection of machines, its behavior can become significantly harder to understand. In particular, a distributed implementation might wish to *replicate* the counter state on each node to improve response time or fault tolerance. Unfortunately, adding replication doesn't come for free. Attempting to update every replicated copy atomically is problematic in the absence of sophisticated transaction support, which impose significant performance penalties. But, without such heavyweight mechanisms, applying an `Add` operation on one node may not be instantaneously witnessed on another, which may be in the process of simultaneously attempting to perform its own `Add` or `Mult` action. While synchronizing the activities of all nodes to ensure at most one such operation is performed at a time is impractical, designating a single node to hold the counter state eschewing replication altogether (as in a typical client-server configuration), is also an undesirable solution, given the sensitivity of such architectures to network partitions and server failures, and the negative performance impact it incurs in geo-distributed environments (Sovran et al. 2011). Removing coordination altogether by simply replicating counter state without having any supporting consistency protocol is an equally infeasible approach.

A method often adopted to address these concerns is to re-define a datatype's operations to return *effects* instead of values (Burckhardt et al. 2014; Shapiro et al. 2011a). An *effect* is a tag that identifies the operation to be applied uniformly at all replicas to incorporate the effects of the original operation. Fig. 1b shows the `Counter` library with operations re-defined to returns effects. Observe that the `Counter.add x v` operation now returns an `Add x` effect, which, when applied at a replica (see `apply` in the figure), adds `x` to the local counter value. Note, however, that `add` and `mult` are not commutative operations - assuming two replicas have the same initial counter value of 0, applying the effect of adding 3 and then multiplying 5 on one replica yields 15, while applying the effect of first multiplying 5 and then adding 3 yields 3 on the other. Such scenarios are possible in a distributed system because there are no coherence guarantees on the order in which effects are received by different nodes. Thus, implementations must be carefully written to take the lack of commutativity into account when defining how effects are applied at a replica; in the figure, multiplication is expressed in terms of addition to avoid the kind of undesirable behavior described above.

In this implementation, all replicas will eventually contain the same counter value, assuming updates to the counter eventually quiesces. While transmitting effects provide a low-level operational basis for handling replicated state, the *ad hoc* nature of the solution confounds desirable high-level reasoning principles. Indeed, the semantic gap between the two versions of the counter, one cognizant of replication and the other not, breaks backward compatibility with the original state-based implementation. Just as significantly, its non-trivial construction must be developed in different guises for every distributed data structure used by the application. The lack of composability is yet another important downside of this approach. Consider an application that uses two replicated counters, c_1 and c_2, bound by the invariant $c_2 \geq c_1$, duly enforced by the application when updating c_2 or c_1. An execution may nonetheless witness anomalous states that violate the invariant because updates to c_1 and c_2 may be applied independently in any

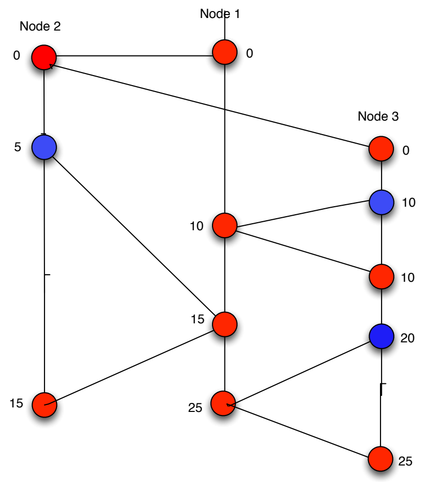


Fig. 2. A replicated counter can be expressed as a sequence of versions managed by logically-distributed concurrently executing threads. Local versions produced by these threads periodically merge their state with one another. In the figure, when the local version of Node 3, labeled 20, merges with the local version of Node 1, whose counter state is 15 at the time, a new counter value is produced. This value takes into account the previous merged state (10) from which the two versions were derived to yield a new merged state of 25. There is no other form of synchronization or coordination among versions except through merging. In the figure, red circles represent counter state produced through merging, and blue circles represent state produced by applying counter operations on a node-local version.

order on any replica. For example, if a replica increments both counters, a `read` operation performed at another replica may witness the increment to c_1, but not c_2, thus violating the invariant. The intention to atomically apply both effects cannot be captured in the absence of external mechanisms to compose effects (such as effect-based transactions (Sivaramakrishnan et al. 2015)). Rather than viewing a distributed computation in terms of `Add` effects produced by operations, can we formulate a more declarative interpretation, directly in terms of the counter value maintained by each node? To do so, we first observe that each replica essentially operates over its own version of a counter. Thus, local operations on a replicated object can be thought of as yielding new versions, collectively producing a version tree, with one branch for each replica. Every branch represents different (immutable) versions maintained by different nodes, with the state produced by the computation performed over a counter on a node recorded along the node's local branch for the counter. Now, to generate a globally consistent view of a counter, we only need to define a merge operation that explains how to combine two local versions to produce a new version that reflects both their states. This operation is defined not in terms of replicas or other system-specific artifacts, but in terms of the semantics of the datatype itself.

Framing replication as merging leads to a counter implementation that bears strong similarity to the original sequential one:

```
module Replicated_Counter = struct
  include Counter
  let merge lca v1 v2 =
      lca + (v1 - lca) + (v2 - lca)
end
```

The role of `lca` (lowest common ancestor) here captures salient history - the state resulting from the merge of two versions derived from the same ancestor state should not unwittingly duplicate the contributions of the ancestor. This interpretation of a replicated datatype is thus given in terms of the evolution of a program state implicitly associated with the different nodes that comprise a distributed application with merge operations serving to communicate and reconcile different local states.

Our main innovation is thus the development of a programming model that realizes a monadic version control system centered around data, rather than file, coherence. The VML (Versioned ML) programming model comprises

```
module type CANVAS = sig
  type pixel = {r:char; g:char; b:char}
  type tree =
    | N of pixel
    | B of {tl: tree; tr: tree; bl: tree; br: tree}
  type t = {max_x:int; max_y:int; canvas:tree}
  type loc = {x:int; y:int}

  val new_canvas: int -> int -> t
  val set_px: t -> loc -> pixel -> t
  val get_px: t -> loc -> pixel
  val merge: (* lca *)t -> (* v1 *)t -> (* v2 *)t -> t
end
```

Fig. 3. Canvas: a sample VML application

the VST monad which lets programmers write and compose concurrent computations around multiple (implicit) versions of a mergeable datatype, an ordinary ML datatype additionally equipped with a *merge* function responsible for deriving a consistent global state from a collection of local versions of that state. A computation progresses by forking (i.e., replicating) new versions of existing versions, allowing synchronization-free local computation to proceed on those nodes, creating new versions along an existing *branch* that represents updated local instances, and by merging branches to realize global consistency.

Our model allows ML programmers to get the benefits of achieving highly available (low-latency) distributed computation, while continuing to enjoy the comforts of high-level reasoning and the familiarity of using standard libraries already provided by the language. Notably, while VML makes no explicit reference to any specific operational manifestation of a distributed system (e.g., programmers do not need to explicitly manage replicas), we demonstrate that it can be nonetheless efficiently realized on existing real-world geo-replicated distributed systems.

Our contributions are summarized below:

- We formally introduce the concept of a *mergeable* datatype to admit high-level declarative reasoning about distributed computation and replicated state in ML programs.
- We describe VML, a programming model that extends ML to brings to bear the power and flexibility of a version control system to the administration of replicated data. VML hides the complexity of version control behind a monad, and exposes its functionality via a simple API that lets programmers define and compose distributed ML computations around such data. Issues related to replication only manifest implicitly in the definition of a merge operation that defines coherence among different instances of (presumably) replicated state.
- We present a formalization of the VML programming model, and establish the conditions under which eventual convergence of concurrent version state can be guaranteed.
- We describe an implementation of the VML library that transparently adds persistence and replication features to a mergeable type, and which sits atop the Irmin persistent store (Irmin 2016), a content-addressable storage library for OCaml. We also present case studies and experimental results that justify the practical utility of our approach.

The remainder of the paper is organized as follows. We present the VML programming model informally in the next section. Sec. 3 develops a series of examples that illustrate how merge operations can be defined and used. Sec. 4 defines an operational semantics, and formalizes our intuition of a mergeable type in terms of morphisms over datatype operations. We also present soundness (consistency) and progress guarantees enjoyed by the semantics. Sec. 5 presents an instantiation of the model suitable for highly-available distributed environments with network partitions and failures. Experimental results and benchmarks are given in Sec. 6. Secs. 7 and 8 describes related work and conclusions.

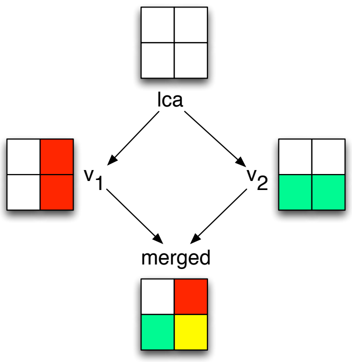


Fig. 4. Merging concurrent versions (`v1` and `v2`) of a drawing canvas. `lca` is their common ancestor.

2 PROGRAMMING MODEL

In this section, we describe the VML programming model through the example of a collaborative drawing application we call Canvas. Fig. 3 shows the signature of the Canvas application. Canvas represents a free-hand drawing canvas in terms of a tree of quadrants. Each quadrant is simply a leaf node containing a single pixel (`r` , `g` , or `b`) or a tree, if the quadrant contains multiple pixels of different colors. Quadrants are expanded into a tree structures as and when pixels are colored. The representation is thus optimized for sparse canvases, such as whiteboards. The application supports three simple operations: creating a new canvas, setting the pixel at a specified coordinate, and returning the pixel at a given coordinate. The `merge` function is explained below. Canvas lets multiple users collaborate on a canvas that is conceptually shared among them. Under a shared-memory abstraction, there would be a single copy of the canvas that is updated concurrently by multiple clients; from the perspective of any single client, the canvas could change without any explicit intervention. VML ascribes functional semantics to sharing by letting each client work on its own version of the state (the tree data structure in this example), later merging concurrent versions on-demand. Thus, the primary artifact of the VML programming model is a versioned data structure in which different versions are managed by different clients.

VML requires a three-way `merge` function to merge concurrent versions of a drawing canvas (see Fig. 5) that includes two concurrent versions (`v1` and `v2`), and their lowest common ancestor (`lca`) - the version from which the two concurrent versions evolved independently. The merge function can make use of the pixel values of the common ancestor to merge the pixel values on both the canvases. For instance, if the color of a pixel in `v1` is white, and in `v2` it is green, and its color in `lca` is white, then it means that only `v2` modified the color. Hence the pixel is colored green in the merged canvas. On the other hand, if the pixel is red in `v1` , then it means that both `v1` and `v2` have modified the color. In such case, an appropriate color-mixing algorithm can be used to determine the color of pixel. For instance, the pixel can be colored yellow - an additive combination of red and green. The logic is illustrated in Fig. 4.

VML lets programmers define and compose concurrent computations around versioned data structures. Fig. 6 shows the signature of the VST module that implements the programming model along the lines of the well-known `State` monad (Wadler 1995). The monad encapsulates a versioned functional state (`'a`) and the type `('a, 'b) t` represents a monadic computation that returns a `'b` result. Functions `return` and `bind` have their usual monadic interpretation. `get_current_version` is like the `State` monad's `get` ; it returns the versioned state encapsulated by the monad. To initiate a computation, we use `with_init_version_do` which runs a monadic computation against a given initial version and returns the result. The `fork_version` operation returns a computation that forks a new concurrent version from the current version, and runs the given monadic computation asynchronously against the forked version. `sync_next_version` (simply called `sync`) accepts a *proposal* for the next version of the state; this proposal is the current local version of the state that reflects local modifications not yet witnessed by any other concurrently executing computation. The operation returns (via a monad) the actual next version, which becomes the current version for the rest of the computation. This version is created by merging the proposal with a subset of concurrent versions that have become available since the last merge or fork. Thus, `sync` effectively lets a computation synchronize with a subset of concurrent computations to obtain their latest updates.

```
let color_mix px1 px2 : pixel =
let f = Char.code in
let h x y = Char.chr @@ (x + y)/ 2 in
let (r1,g1,b1) = (f px1.r, f px1.g, f px1.b) in
let (r2,g2,b2) = (f px2.r, f px2.g, f px2.b) in
let (r,g,b) = (h r1 r2, h g1 g2, h b1 b2) in {r=r; g=g; b=b}

let b_of_n px = B {tl_t=N px; tr_t=N px; bl_t=N px; br_t=N px}

let rec merge lca v1 v2 =
  if v1=v2 then v1
  else if v1=lca then v2
  else if v2=lca then v1
  else match (lca,v1,v2) with
    | (_, B _, N px2) -> merge lca v1 @@ b_of_n px2
    | (_, N px1, B _) -> merge lca (b_of_n px1) v2
    | (N px, B _, B _) -> merge (b_of_n px) v1 v2
    | (B x, B x1, B x2) ->
        let tl_t' = merge x.tl_t x1.tl_t x2.tl_t in
        let tr_t' = merge x.tr_t x1.tr_t x2.tr_t in
        let bl_t' = merge x.bl_t x1.bl_t x2.bl_t in
        let br_t' = merge x.br_t x1.br_t x2.br_t in
          B {tl_t=tl_t'; tr_t=tr_t'; bl_t=bl_t'; br_t=br_t'}
    | (_, N px1, N px2) ->
        (* pixels are merged by mixing colors *)
        let px' = color_mix px1 px2 in N px'
```

Fig. 5. Merging different versions of a canvas.

```
  module type VML = sig
    type ('a, 'b) t
    val return : 'b -> ('a, 'b) t
    val bind : ('a, 'b) t -> ('b -> ('a, 'c) t) -> ('a, 'c) t
    val get_current_version: unit -> ('a, 'a) t
    val with_init_version_do: 'a -> ('a, 'b) t -> 'b
    val fork_version : ('a, 'b) t -> ('a, unit) t
    val sync_next_version: unit -> 'a -> ('a, 'a) t
  end
```

Fig. 6. Signature of the VML monad

Fig. 7 demonstrates how a collaborative drawing session between Alice, Bob and Cheryl can be composed using VML. A possible execution of the session is visualized in Fig. 8. Assume that the session starts with Alice on a 5×5 blank canvas, as shown below:

```
  module C = Canvas;;
  with_init_version_do (C.new_blank 5 5) alice_f
```

Alice starts by reading the current version of the canvas, which is blank. She then invites Bob for collaboration by forking a new concurrent version for Bob. Bob, in turn, invites Cheryl for collaboration. All three of them start working on the blank canvas. Alice draws a red horizontal line from $(0,0)$ (top-left) to $(4,0)$ (top-right) using `C.draw_line`.[1] Meanwhile, Bob draws a green vertical line from $(0,0)$ to $(0,4)$, and Cheryl draws a similar line from $(4,0)$ to $(4,4)$. All three of them call `sync` with their respective proposals (C_0'). While any partial ordering of

[1]Its definition is not shown, but can be constructed using `set_px`.

```
let alice_f : C.t unit t =
  get () >>= fun c0 ->
  fork bob_f >>= fun () ->
  let c0' = C.draw_line c0
    {x=0;y=0}
    {x=4;y=0} in
  sync () ~v:c0' >>=
  fun c1 ->
  let c1' = C.draw_line c1
    {x=0;y=4}
    {x=4;y=4} in
  sync () ~v:c1' >>=
  fun c2 -> return ()
```

```
let bob_f : C.t unit t =
  get () >>= fun c0 ->
  fork cheryl_f >>=
  fun () ->
  let c0' = C.draw_line c0
    {x=0;y=0}
    {x=0;y=4} in
  sync () ~v:c0' >>=
  fun c1 -> sync () >>=
  fun c2 -> return ()
```

```
let cheryl_f : C.t unit t =
  get () >>= fun c0 ->
  let c0' = C.draw_line c0
    {x=4;y=0}
    {x=4;y=0} in
  sync () ~v:c0' >>=
  fun c1 -> sync () >>=
  fun c2 -> return ()
```

Fig. 7. Canvas: A collaborative drawing session between Alice, Bob, and Cheryl

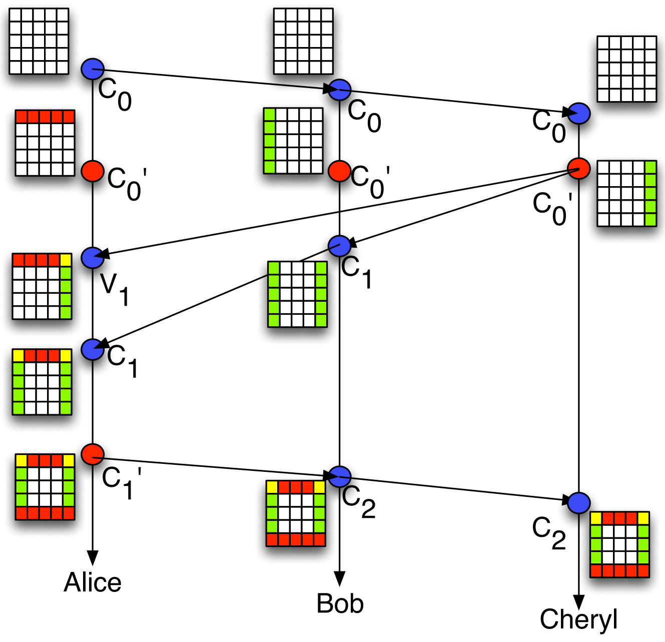


Fig. 8. Canvas: Collaborative drawing session visualized

concurrent `syncs` is valid, we consider a linear order where Cheryl's `sync` happens first, followed by Bob's and then Alice's. Cheryl's `sync` does not find any concurrent versions, hence installs the proposed version (C'_0) as the next version on Cheryl's branch. Bob's `sync` finds Cheryl's C'_0 as a concurrent version, and merges it with its proposal to produce the next version C_1, which is then installed on Bob's branch. The lowest common ancestor (LCA) for this merge is the initial version (C_0), and the two concurrent versions are Bob's C'_0 and Cheryl's C'_0. Next, Alice's `sync` finds Cheryl's C'_0 and Bob's C_1 as concurrent versions, and merges them successively with Alice's proposal. For the first merge, the two concurrent versions are Alice's C'_0 and Cheryl's C'_0, and the LCA is the initial version (C_0). The result of this merge is installed as the next version (V_1) on Alice's branch. For the next merge, the two concurrent versions are Alice's V_1 and Bob's C_1 and the LCA is Cheryl's C'_0[2]. The result (C_1) becomes the next version on Alice's branch, and the return value of Alice's `sync`. Next, Alice draws a red horizontal line from $(0, 4)$ to $(4, 4)$, proposes this canvas (`c1'` in the first column of Fig. 7) as the next version to `sync`. Since there are no concurrent versions, C'_1

[2]Thus, the LCA of versions on two branches can lie outside both the branches.

```
module type MRope = sig
  module A: sig
    type t
    val concat: t -> t -> t
    val split: t -> t*t
    val merge: t -> t -> t -> t
  end
  type t =
  | Leaf of A.t
  | Node of t * int * t
  ...
  (* Standard rope functions *)
  val merge: t -> t -> t -> t
end
```

(a) Signature of Mergeable Ropes

```
let rec flatten = function
  | Leaf x -> x
  | Node (l,_,r) -> A.concat (flatten l) (flatten r)

let rec merge old l r =
  if l = r then l else if old = l then r
  else if old = r then l else merge_rec old l r

and merge_rec old l r = match (old,l,r) with
  | Leaf _, _, _ | _, Leaf _, _
  | _, _, Leaf _ ->
      rope_of @@ A.merge (flatten old)
                         (flatten l) (flatten r)
  | Node (oldl,_,oldr), Node (ll,_,lr),
    Node (rl,_,rr) ->
      let newl = merge oldl ll rl in
      let newr = merge oldr lr rr in
        Node (newl, create_index newl, newr)
```

(b) Mergeable rope implementation

Fig. 9. Signature and implementation for mergeable ropes

becomes her next version. The subsequent `sync` operations from Bob and Cheryl propose no new versions, hence simply obtain Alice's C_1' as next versions.

3 MERGEABLE TYPES

The Canvas example demonstrates the VML's utility in building concurrent applications with conceptual state sharing that make them suitable for transparent deployment in a distributed setting. The model allows concurrent computations to be composed around any mergeable type. We now present a series of examples that demonstrate how mergeable types are built - either by defining merge functions for ML data types, or by composing existing mergeable types.

3.1 Ropes

A rope is a data structure that facilitates efficient implementations of operations such as concatenation, splitting, lookup and insertion at a random position etc., for a list of values in a distributed setting. The values are usually characters, with their lists being strings. The data structure is a binary tree, whose leaf nodes store substrings of a larger string such that their left-to-right (in-order) concatenation yields the larger string. The idea is to facilitate efficient lookups and insertions at random positions in the string by storing position indexes in the internal nodes of the tree.

Fig. 9a shows a rope interface in OCaml. The type of the list of values is left abstract (`A.t`); it can be instantiated with any module that supports `concat` and `split` operations. The interface shown in Fig. 9a is that of a mergeable rope (`MRope`), whose `merge` implementation is shown in Fig. 9b. The function returns one of its versions (`l` or `r`), if the other one is the same as the common ancestor. Otherwise, it calls `merge_rec`, which recursively compares the structures of merging ropes, and their common ancestor. The function recurses until one of the ropes differs from the other two, or all of them are leaf nodes, at which point it relies on the (abstract) list merge function (`A.merge`) to merge the lists represented by the ropes. `merge_rec` uses auxiliary functions, `mk_rope` and `create_index`, whose definitions are not shown to construct a rope given a list (`A.t`), and to compute the index at a node, given its left sub-tree, resp.

One feature of the mergeable rope implementation that is characteristic of mergeable types in general is compositionality; a rope is a polymorphic mergeable data type that composes mergeability over its polymorphic constituents. Compositionality is concretized by the `MRope` signature , which requires the type of items in the list (`A.t`) to also be mergeable (`A.merge`).

```
module type MList = sig
  module A: sig
    type t
    val merge: t -> t -> t -> t
  end
  type t = A.t list
  ... (* All the standard list functions *)
  val merge: t -> t -> t -> t (* and the merge function *)
  (* The following are exposed only for this presentation *)
  type edit =
      I of A.t * int
    | D of int
    | S of int * A.t * A.t
  val edit_seq: t -> t -> edit list option
  val op_transform: edit list -> edit list -> edit list
end
```

Fig. 10. Signature of Mergeable Lists

3.2 Lists

As an abstract data type, lists support three operations: `insert x i l`, to insert an element `x` at position `i` in the list `l`, `delete i l`, to delete the element at `i`'th position, and `subst i x l`, to substitute the element at the `i`'th position with `x`. Lists of mergeable items are mergeable if they preserve the following properties:

- Elements present in both the merging lists will be present in the final list. No element is deleted unless it is deleted in at least one list.
- An element deleted in one list will not reappear in the merged list (unlike, for example, Amazon's replicated shopping cart (DeCandia et al. 2007)).
- The order of elements is retained. That is, if x occurs before y in one of the lists, and if both elements are present in the merged list, then x occurs before y in the merged list.
- If an element y is substituted for an element x in one list, then y is also substituted for x in the merged list (i.e., x is absent and y is present), unless x is deleted or substituted in the other list. In the former case, deletion wins, and neither x nor y occur in the merged list. In the latter case, if x is substituted by a different element z in the other list, then the merged list substitutes x by a merge of y and z, defined as per the merge semantics of the list element type.

The merge operation for lists is composed of two separate functions, `edit_seq` and `op_transform`. Both these functions have been implemented in less than 50 lines of OCaml. The functions are described below.

- `edit_seq` takes a pair of lists, `v` and `v'`, and computes the shortest sequence of list operations that need to be applied on `v` to obtain `v'`. Such a sequence is called an *edit sequence*. The length of the sequence corresponds to the standard notion of the *edit distance* between the two lists, which can be computed in polynomial time (Wagner and Fischer 1974). `edit_seq` modifies one such algorithm to also compute the edit sequence (an `edit list`). The signature specified in Fig. 10 represents edits using the type `edit`. Constructors `I`, `D`, and `S` stand for `insert`, `delete`, and `subst`, respectively.[3] The subst constructor also carries the `A.t` element that was substituted. The element serves as the `lca` argument to `A.merge` in case of concurrent conflicting substitutions. Fig. 11 illustrates edit sequences for a sample list. The sequence `[I(c,0); S(3,c,s)]` maps the list `[a;b;c]` to `[c;a;b;s]` because `subst 3 s (insert c 0 [a;b;c]) = [c;a;b;s]`.
- `op_transform` takes a pair of edit sequences, s_1 and s_2, that map a list v to two different lists, v_1 and v_2 (e.g., Fig. 11), and transforms s_1 to s_1' such that s_1' has the same effect on v_2 as s_1 had on v. For instance, in Fig. 11, let s_1 be the edit sequence `[D(1); S(1,c,d)]`, which maps the list `[a;b;c]` (v) to `[a;d]` (v_1). The

[3] Note that `I`, `D` and `S` are *not* effects; no list function generates them, and they are not exchanged between concurrent threads. They are simply tags used by the list merge operation for convenience. Indeed, to better appreciate the virtues of mergeable types, we encourage the reader to think about how one might implement a replicated list with this functionality using an explicit effect representation as described in Sec. 1.

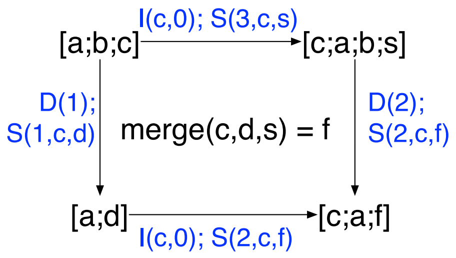


Fig. 11. Lists of mergeable values are mergeable.

D edit deletes the first element of v. However, if s_1 is to be applied on the list `[c;a;b;s]` (v_2), which has a new element at position 0, the D edit must delete the second element from v_2, if it were to have the same effect as it did on v. We say `D(1)` in s_1 has to be transformed w.r.t the sequence s_2 = `[I(c,0);S(3,c,s)]`. The function `op_transform` (shown in the Appendix) computes such *operational transformations* for an edit sequence s_1 w.r.t another edit sequence s_2. A notable transformation rule is for a substitution w.r.t another substitution at the same position. Since the substituting items could be different, the function relies on the item merge function (`A.merge`) to merge them into a single item. For instance, in Fig. 11, there are simultaneous substitutions to the item `c` in s_1 and s_2; while s_1 substitutes it with `d`, s_2 does it with `s`. The merged list therefore substitutes `c` with the merge of `d` and `s`, defined as per the merge semantics of `A.t`.

The `MList.merge` can now be defined as a function that first computes edit sequences, s_1 and s_2, for the two concurrent lists, v_1 and v_2 w.r.t their common ancestor v, transforms one of them (say, s_1) w.r.t s_2 to obtain s_1', and finally applies (the operations defined by) s_1' to v_2 to obtain the merged list.

3.3 Lists of Integers and Quantities

A mergeable list of integers can be composed using mergeable lists and mergeable integers. Concretely, it is a composition of `MList` and `Int` types (i.e., `MList(Int)`), where `Int.merge` is the same as `Counter.merge` described in the introduction. An alternative instantiation of `MList` is for a quantity type (`Qty`) that supports `add` and `subtract` operations, and defines its `merge` operation a little differently:

```
let merge lca v1 v2 = if v1 = lca then v2
    else (if v2 = lca then v1 else max v1 v2)
```

`MList(Qty)`, like `MList(Int)`, is a list of integers, and has the same semantics under a sequential execution. Their differences however manifest under a concurrent execution, where both adopt different methods to reconcile concurrent updates.

If efficient insertions and lookups (by position) are desired over a large list of integers, while admitting concurrent updates, we require a mergeable rope of integers, which can be obtained by simply composing `MList(Int)` with `MRope` (i.e., `MRope(MList(Int))`).

3.4 Shopping List

A non-trivial demonstration illustrating the compositional power of mergeable types is a shopping list application that allows its users to collaboratively build a shopping list (listing items in the decreasing order of their priority), by adding new items with a specified quantity, deleting existing items, and increasing or decreasing their quantity. An item (`Item.t`) is represented as a tuple of its name (a string) and the quantity (A `Qty.t`). The merge function for `Item.t` merges the quantities of items (using `Qty.merge`) having the same names. Items with different names are considered distinct. A shopping list is a mergeable list of items, i.e., `MList(Item)` (although a grocery shopping list may not be large enough to justify using ropes, `MRope(MList(Item))` is certainly allowed). `MList(Item).merge` automatically lifts the merge semantics for items to a merge semantics for lists of items. Fig. 12 illustrates its behavior. Here, two users, Alice and Bob, concurrently edit a shopping list whose initial contents are `milk` and `eggs`. While both simultaneously update the quantity of eggs, Bob also removes `milk` and inserts `candy`. Fig. 13 shows the VML code for two concurrent sessions. The merged shopping list contains `eggs` and `candy`, where the quantity of eggs is obtained by merging Alice's and Bob's quantity as per the definition of `Qty.merge`. The examples described above

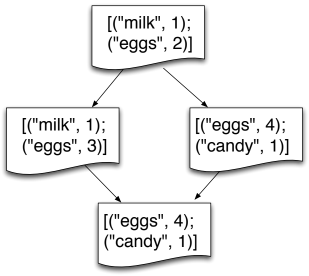

[("milk", 1); ("eggs", 2)]
[("milk", 1); ("eggs", 3)]
[("eggs", 4); ("candy", 1)]
[("eggs", 4); ("candy", 1)]

Fig. 12. Merging concurrent shopping lists

```
let alice_f : L.t unit t =
  fork bob_f >>= fun () -> (* Invite Bob *)
  get () >>= fun c0 ->
  let qty = L.assoc "eggs" in
  let c0' = L.subst 1 ("eggs",qty+1) c0 in
  sync () ~v:c0' >>=
  fun c1 -> return ()
```

```
let bob_f : L.t unit t =
  get () >>= fun c0 ->
  let c0' = L.insert ("candy",Qty.of_int 1) 1 @@
            L.delete @@ L.lookup "milk" in
  sync () ~v:c0' >>=
  fun c1 -> return ()
```

Fig. 13. Bob and Alice collaboratively build a grocery shopping list

make certain assumptions about the concurrency model and the underlying system, such as the existence of a single lowest common ancestor (LCA) for any pair of versions, the ability to access a previous version on any branch, and mergeability of any two concurrent versions. Enforcing these guarantees in a fully decentralized distributed setting requires addressing non-trivial theoretical challenges. These, and other details that contribute to the practicality of our model, such as containing the complexity of merges, are discussed in subsequent sections.

4 FORMALIZATION

4.1 Operational Semantics

We formalize our ideas in the context of a lambda calculus (λ_b) shown in Fig. 14. Expressions of λ_b are variables, constants, and VML primitives composed using the lambda combinator. For brevity, we use short names for VML primitives: `run` for `with_init_version_do`, and `fork` for `fork_version`. To simplify the technical development, VML's `sync_next_version` operation is broken down into two primitives - `push` and `pull`, which can be composed to get the desired effect of the original: $\texttt{sync}\ x = (\lambda y.\texttt{pull})\,(\texttt{push}\ x)$. The semantics of `get_current_version` is subsumed by `pull`, hence elided. Values (v) are constants and lambda abstractions. A program (p) is a parallel composition of threads, where each thread is an expression (s) indexed by the corresponding thread identifier (t).

Fig. 14 also shows the syntax of *branches*, which are artifacts of evaluation and only appear during run-time. A branch is a non-empty sequence of tagged values, where the tag captures the abstract run-time operation that led to the creation of the value. It is implicitly assumed that each value added to a branch is uniquely identifiable, and hence no two values on a branch are equal. The uniqueness assumption is later extended to a collection of branches that constitute a branching structure. A real implementation meets this assumption by versioning values across the branches. Thus, in reality, branches contain *versions* which denote values. To simplify the presentation, the semantics, does not make this distinction, and uses values and versions interchangeably.

The small-step operational semantics of λ_b is defined via a reduction relation ($\longrightarrow$) that relates *program states.* A program state (p; H) consists of a program p and a *branch history* H that maps thread identifiers to corresponding branches; each thread is associated with a branch during evaluation. Evaluation contexts have been defined separately for expressions (E) and programs (P), with the latter subsuming the former.

E is defined to evaluate the first argument of a `run` expression to a value that constitutes an initial version (recall that `run` models VST's `with_init_version_do`). A program evaluation context non-deterministically picks an extant thread to evaluate. The administrative rule that relates transitions of holes to transitions of expressions and programs is straightforward, and hence elided. The remaining core reduction rules are presented in Fig. 14. For brevity, we write

Syntax

$$\begin{array}{lcllll}
 & & t \in \mathtt{Thread\ Ids} & & x, y \in \mathtt{Variables} & c \in \{()\} \cup \mathbb{N} \\
v & \in & \mathtt{Values} & ::= & c \mid \lambda x.\, s & \\
s & \in & \mathtt{Expressions} & ::= & v \mid s\, s \mid \mathtt{run}\ s\, s \mid \mathtt{fork}\ s \mid \mathtt{pull} \mid \mathtt{push}\ s & \\
p & \in & \mathtt{Programs} & ::= & s_t \mid p \,||\, p & \\
f & \in & \mathtt{Tags} & ::= & \mathtt{INIT} \mid \mathtt{FORK}\ b \mid \mathtt{PUSH} \mid \mathtt{MERGE}\ b & \\
b & \in & \mathtt{Branches} & ::= & [(v, f)] \mid (v, f) :: b &
\end{array}$$

Artifacts of Evaluation

$$\begin{array}{lcll}
E & \in & \mathtt{Eval.\ Contexts}(s) & ::= \bullet \mid \bullet\, s \mid v\, \bullet \mid \mathtt{run}\ \bullet\ s \\
P & \in & \mathtt{Eval.\ Contexts}(p) & ::= E_t \mid \bullet \,||\, p \mid p \,||\, \bullet \\
H & \in & \mathtt{Branch\ Histories} & ::= t \mapsto b \\
\mathtt{lca} & \in & \mathtt{LCA\ function} & ::= b \times b \rightarrow v
\end{array}$$

Reduction Relation $\boxed{p;\ H \longrightarrow p';\ H'}$

$$\begin{array}{lcll}
((\lambda x.s)\ v)_t; H & \longrightarrow & ([v/x]\,s)_t; H & \text{[E-App]} \\
(\mathtt{run}\ v\ s)_t; \cdot & \longrightarrow & s_t; \cdot[t_\top \mapsto [(v, \mathtt{INIT})]][t \mapsto [(v, \mathtt{FORK}\ [(v, \mathtt{INIT})])]] & \text{[E-Run]} \\
(\mathtt{fork}\ s)_t; H(t \mapsto (v, \_) :: b) & \longrightarrow & ()_t \,||\, s_{t'}; H[t' \mapsto [(v, \mathtt{FORK}\, H(t))]] \quad \mathtt{where}\ t' \notin dom(H) & \text{[E-Fork]} \\
(\mathtt{push}\ v)_t; H & \longrightarrow & ()_t; H[t \mapsto (v, \mathtt{PUSH}) :: H(t)] & \text{[E-Push]} \\
(\mathtt{pull})_t; H(t \mapsto (v, \_) :: m) & \longrightarrow & v_t; H & \text{[E-Pull]}
\end{array}$$

[E-Pull-Wait]

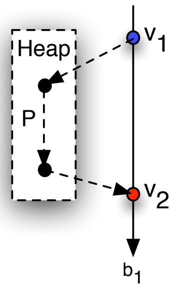
$$\frac{t \neq t' \quad H \vdash v' \rightsquigarrow_m v \quad v' \npreceq v \quad v_m = \mathtt{merge}\,(\mathtt{lca}\,(H(t), H(t')), v, v')}{(\mathtt{pull})_t; H(t \mapsto (v, f) :: m)(t' \mapsto (v', \_) :: \_) \longrightarrow (\mathtt{pull})_t; H[t \mapsto (v_m, \mathtt{MERGE}\ H(t')) :: (v, f) :: m]}$$

Fig. 14. vml: Syntax and Operational Semantics

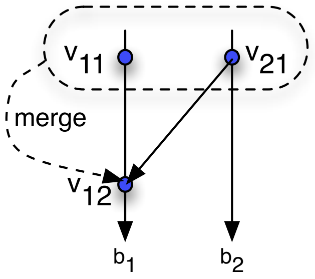
$H(t \mapsto (v, f))$ to denote the proposition that H maps t to (v, f). The notation $H[t \mapsto (v, f)]$ denotes the extension of H with the binding $t \mapsto (v, f)$, as usual.

Reduction rules let expression evaluation take a step by rewriting the expression and suitably updating the branch history (H). E-App is the standard beta reduction rule. The E-Run rule is applicable only when H is empty, i.e., when no prior branching structure exists, and commences a distributed computation with a corresponding version tree. The rule rewrites the `run` $v\ s$ expression to s, while creating a new branching structure with two branches: a *top* branch that has just the initial version (tagged with `INIT`), and a branch for the current thread (t) forked-off from the top branch. The first version on the current branch ($H(t)$) denotes the same value (v) as the initial version on the top branch, although versions themselves are deemed distinct. The new version is tagged with a `FORK` tag that keeps the record of its origin, namely the `fork` operation and the branch from which the current branch is forked. The E-Fork rule forks a new thread with a fresh id (t') and adds it to the thread pool. The corresponding branch ($H(t')$) is forked from the parent thread's branch ($H(t)$). The semantics of branch forking is the same as described above. The `fork` expression in the parent thread evaluates to `()`. The E-Push rule creates a new version on the current branch ($H(t)$) using the pushed value (v). Although our semantics does not directly expose heap-allocated values, the intention is that v is a replicated object, manipulated on the local heap that, after push, now becomes subject to merging and coordination with other replicas.

The semantics non-deterministically chooses E-Pull or E-Pull-Wait rules to reduce a `pull` expression. The E-Pull rule reduces `pull` to `()`, and returns the latest version on the current branch. The E-Pull-Wait rule can be thought of as a stutter step; it doesn't reduce `pull`, but updates the branching structure by merging (the latest version of) a concurrent branch ($H(t')$) into (the latest version of) the current branch ($H(t)$), and extending the current branch with the merged version (v_m). The versions are merged only if they are *safely mergeable* (denoted



(a) When a thread `push`es a new version, it is presumably a semantic successor of the version it last `pull`ed into the heap.



(b) Versions created by the `merge` operation are syntactic successors of merged versions, but need not necessarily be semantic successors.

Fig. 15. New versions are created from existing versions either through `push` or `merge`.

$H \vdash v' \rightsquigarrow_m v$) - a notion that we will explain shorty. The new version is tagged with a `MERGE` tag that, like a `FORK` tag, records its origin. The rule assumes a function `lca` that computes the *lowest common ancestor* (LCA) for the latest versions on the given pair of branches. The formal definition of LCA is given below. The E-Pull-Wait and E-Pull rules thus let a thread synchronize with other distributed threads manipulating versions of replicas in multiple steps before returning the result of the `pull`. Since `sync` is a composition of `push` and `pull`, its behavior can be explained thus: `sync` pushes the given value onto the current (local) branch, merges a (possibly empty) subset of concurrent branches into the local branch, and returns the result. (The operation is not guaranteed to synchronize with all concurrent branches because not all such branches may be available, because of e.g., network partitions.)

To define LCA, we formalize the intuitive notation of the ancestor relationship between versions of a legal branching history (i.e., a branching history generated by the rules in Fig. 14):

*Definition 4.1 (**Ancestor**).* Version v_1 is a ancestor of version v_2 under a history H (written $H \vdash v_1 \leq v_2$) if and only if one of the following is true:

- There exists a branch b in H (i.e., $\exists(t \in dom(H)).\, H(t) = b$) in which v_2 immediately succeeds v_1,
- There exists a branch b in H that contains (v_2, `FORK` $(v_1, f_1) :: b_1$), for some f_1 and b_1,
- There exists a branch b in H that contains (v_2, `MERGE` $(v_1, f_1) :: b_1$), for some f_1 and b_1,
- $v_1 = v_2$, or v_1 is transitively an ancestor of v_2, i.e., $\exists v.\, H \vdash v_1 \leq v \wedge H \vdash v \leq v$

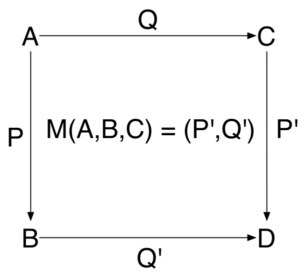
The ancestor relation is therefore a partial order with a greatest lower bound (the initial version). Thus, for any two versions in a legal history, there exists at least one common ancestor. The ancestor relationship among all common ancestors lets us define the notion of a lowest common ancestor (LCA):

*Definition 4.2 (**Lowest Common Ancestor**).* Version v is said to be a common ancestor of versions v_1 and v_2 under a history H if and only if $H \vdash v \leq v_1$ and $H \vdash v \leq v_2$. It is said to be the lowest common ancestor (LCA) of v_1 and v_2, iff there does not exist a v' such that $H \vdash v' \leq v_1$ and $H \vdash v' \leq v_2$ and $H \vdash v \leq v'$.

The function `lca` used in the premise of E-Pull-Wait computes the LCA of two branches, and assumes it to be unique. But, observe that the definition of LCA doesn't guarantee its uniqueness; it has to be enforced explicitly. The semantics enforces the uniqueness of LCA by constraining the branching structure via the E-Pull-Wait rule. The rule merges version v' into version v only if they are safely mergeable ($H \vdash v' \rightsquigarrow_m v$), i.e., they have a single LCA, and merging both versions does not lead to a case where a branch (i.e., its latest version) has multiple LCAs with remaining branches, andd hence gets *stuck*. Such nuances of LCA, and the guarantees provided by the system notwithstanding these nuances, are discussed in Sec. 4.3.

4.2 Mergeable Data Type

We now formally define a mergeable type. A functional data type library is a tuple consisting of a type (t), and a collection of functions (f, g, etc.) of type $t \rightarrow t$ that are *primitive morphisms*. A morphism (P, Q, etc.) is either a primitive morphism, or an associative composition of morphisms ($P \circ Q \circ R$). An object $B : t$ is called a *semantic*



$$\begin{tikzcd} A \arrow[r, "Q"] \arrow[d, "P"'] \arrow[dr, phantom, "M(A,B,C) = (P',Q')"] & C \arrow[d, "P'"] \\ B \arrow[r, "Q'"'] & D \end{tikzcd}$$

Fig. 16. Commutative diagram representing the merge operation.

successor of $A : t$ (conversely, A is called a *semantic ancestor* of B) if and only if there exists a morphism P such that $P(A) = B$.

The aim of a three-way merge function over a type t is to merge a pair of semantic successors, $B : t$ and $C : t$, of an object $A : t$, into another object $D : t$ such that the relationship between the semantic successors and D satisfies certain conditions. These conditions can be understood by observing the E-Pull-Wait rule of Fig. 14, which applies the `merge` function to a pair of concurrent versions (v_1 and v_2) and their lowest common ancestor (v). Thus, the only relationship that exists between v and v_1, and v and v_2 is the syntactic ancestor relationship that follows from the computation's branching structure. The merge function, as described above, however assumes that concurrent versions are semantic successors of their LCA. It is therefore essential to maintain coherence between the syntactic and semantic ancestor relations.

The ways in which syntactic ancestor relationships are created among versions is captured in Fig. 15. Whenever an object is pushed onto the branch, an ancestor relationship is created between the previous version v_1, and the newly created version v_2 (Fig. 15a). However, since v_2 is pushed by the thread after reading v_1 into the heap, it is reasonable to assume that v_2 is a result of applying a morphism P to v_1 (i.e., $\exists P.\ v_2 = P(v_1)$). Hence, v_1 is a semantic ancestor of v_2. Fig. 15b captures another way of establishing an ancestor relationship, namely by merging branches. Version v_{21} on branch b_2 is merged into v_{11} on b_1 to create v_{12} on b_1. Versions v_{11} and v_{21} are now syntactic ancestors of v_{12}, but for them to be semantic ancestors, the merge function has to enforce the relationship explicitly. In other words, the result of merging a pair of semantic successors, B and C, of an A, has to be an object D that is a semantic successor of both B and C. We now formalize this intuition to define a mergeable type.

*Definition 4.3 (**Mergeable Type**).* A type t is said to be mergeable, if and only if there exists a function M of type $t \times t \times t \to (t \to t) \times (t \to t)$ that satisfies the following property: $\forall (A, B, C : t).\ (\exists (P, Q : t \to t).\ B = P(A) \wedge C = Q(A)) \Rightarrow (\exists (P', Q' : t \to t).\ M(A, B, C) = (P', Q') \wedge Q'(B) = P'(C))$

Intuitively, a type t is a mergeable type if, whenever there exist two morphisms P and Q that map $A : t$ to $B : t$ and $C : t$, there also exists morphisms P' and Q', that map both B and C to $D : t$. This intuition is expressed visually in Fig. 16. For a mergeable type t, we have the guarantee that a sequence of values read by a thread from its branch is sensible, as per the data type semantics:

Theorem 4.4 (**Branch-local consistency**). *Let t be a mergeable type, and let H be a legal branching history over values of type t. For every pair of values $v_1 : t$ and $v_2 : t$ along a branch b in H, if $H \vdash v_1 \preceq v_2$, then there exists a morphism $P : t \to t$ such that $v_2 = P(v_1)$.*

The proof follows directly from the fact that the values on branches are computed from previous values either by applying morphisms locally, or by merging from the remote branches. In the latter case, Def. 4.3 guarantees that the existence of a morphism from the merging value to the merged value.

Rather than specifying merge (M) in terms of an LCA as we did in describing our programming model, Def. 4.3 specifies M in terms of a pair of morphisms. This specification is useful to avoid the expression of certain nonsensical merge functions, and thus enforce a useful branch-local consistency property (Theorem 4.4). For instance, consider the counter data type from Sec. 1. A three-way merge function can be defined to return a constant regardless of its arguments:

```
let merge lca a b = 0
```

$$\begin{tikzcd}
A \arrow[r, "{[[Q^*]]}"] \arrow[d, "{[[P^*]]}"'] \arrow[dr, phantom, "{M(P^*,Q^*) = (P',Q')}"] & C \arrow[d, "P'"] \\
B \arrow[r, "Q'"'] & D
\end{tikzcd}$$

Fig. 17. Commutative diagram for Def. 4.5.

Such a merge function allows the client to witness a violation of branch-local consistency. For example, the client may `sync` the counter, whose current local value is 10, and obtain 0 as a result, thus witnessing a violation of a counter's monotonicity property. Fortunately, Def. 4.3 disallows this semantics. It is impossible to define the function M for the counter data type that violates the monotonicity invariant, because no counter morphism violates the invariant. In general, the specification of M guarantees that the merge operation for a type t preserves any invariants (e.g., balancedness of a tree, sortedness of a list, non-negativeness of an integer etc) that are preserved by all of t's morphisms. The commutativity of the diagram in Fig. 16 need not be enforced statically. It can equally be verified at run-time by checking that the morphisms returned by M map respective concurrent versions to the same value. The E-Pull-Wait rule of the operational semantics (Fig. 14) can be extended to this effect:

$$\frac{t \neq t' \quad H \vdash v' \rightsquigarrow_m v \quad v' \not\leq v \quad (P', Q') = M(\texttt{lca}\,(H(t), H(t')), v, v') \quad Q'(v) = P'(v')}{(\texttt{pull})_t; H(t \mapsto (v, f) :: m)(t' \mapsto (v', \_) :: \_) \longrightarrow (\texttt{pull})_t; H[t \mapsto (v_m, \texttt{MERGE}\ H(t')) :: (v, f) :: m]}$$

Def. 4.3 is immediately applicable to simple data types like counter to explain why they are mergeable. The merge logic for counter is simple enough that it does not require great effort to identify the P and Q that lead to the concurrent counter versions, and transform them into P' and Q' such that the diagram in Fig. 16 commutes. For more sophisticated data types, such as list, this process is more involved, and may require the programmer to perform non-trivial reasoning. We now present an alternative definition of a mergeable type that is stronger than Def. 4.3, but also serves as a recipe to build non-trivial merge functions, like that of a list.

Notation. As mentioned earlier, we distinguish between the named functions defined by a data type library (primitive morphisms), and their compositions (morphisms). A sequence of primitive morphisms is written P^* or Q^*. The composition of a sequence of primitive morphisms (written $[\![P^*]\!]$) is their right-to-left composition, i.e., `fold_left` $(\lambda f.\lambda g.f \circ g)$ id P^*, where `fold_left` has its usual OCaml type. The length of a sequence S is written $|S|$.

*Definition 4.5 (**Mergeable Type (Strengthened)**).* A type t is said to be mergeable if and only if the following conditions hold:

- There exists a function $W : t \times t \to (t \to t)^*$ that accepts an object $A : t$ and its semantic successor $B : t$, and returns a minimal sequence of primitive morphisms P^*, whose composition maps A to B. That is, $W(A, B) = P^*$, where $[\![P^*]\!](A) = B$, and there does not exist an R^* such that $[\![R^*]\!](A) = B$ and $|R^*| < |P^*|$
- There exists a function $M : (t \to t)^* \times (t \to t)^* \to (t \to t) \times (t \to t)$ that accepts a pair of minimal sequences of primitive morphisms, P^* and Q^*, and returns a pair of morphisms, P' and Q', such that for any object $A : t$, $(Q' \circ [\![P^*]\!])(A) = (P' \circ [\![Q^*]\!])(A)$.

The commutativity diagram of Fig. 17 visualizes this definition.

The definition asserts the existence of two functions for a mergeable type t. First, the function W, which (implicitly) computes the edit distance between an object $A : t$ and its semantic successor $B : t$. The sequence of primitive morphisms it returns (P^*) is called an edit sequence. Applying the edit sequence on A produces B (i.e., $[\![P^*]\!](A) = B$). Second, the definition asserts the existence of a function M, which performs an operational transformation of the primitive morphisms in an edit script. An operational transformation of a primitive morphism P w.r.t a morphism Q is another primitive morphism P' such that for any object $A : t$, P' has the same *effect* on $Q(A)$ as P has on A.

Def. 4.5 is immediately applicable to the list type, and lets us explain why it is a mergeable type. `MList.edit_seq` and `MList.op_transform`, respectively, provide evidences for the existence of W and M.

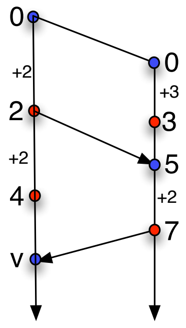
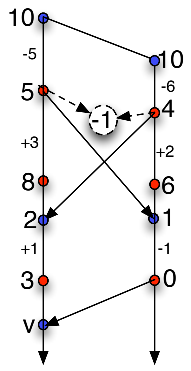
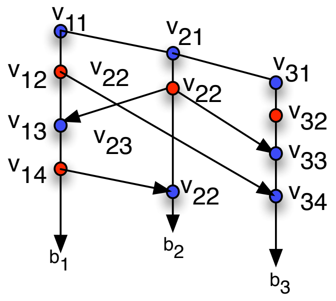


Fig. 18. This example of a grow-only counter illustrate why `merge` needs a lowest common ancestor, and not just a common ancestor. Both 0 and 2 are common ancestors of 4 and 7, while 2 is their lowest common ancestor (since $0 \leq 2$). The result (v) of merging 4 and 7 is 11 (incorrect) if 0 is used as the common ancestor for merge, and 9 (correct, because 2+2+3+2 = 9) if 2 is used.



(a) In this example, 1 and 3 have two LCAs (3 and 4) a result of previous merges. The dotted circle denotes a virtual ancestor obtained by merging the two LCAs.



(b) In this example, latest versions on b_1 and b_3, and b_2 and b_3 have two LCAs each, hence are unmergeable.

Fig. 19. Examples where two branches have more than one LCA, hence cannot merge.

4.3 Properties of the LCA

When merging two concurrent versions v_1 and v_2, the common ancestor argument for `merge` must be the LCA of v_1 and v_2, without which `merge` may yield unexpected results. This is demonstrated for the grow-only counter in Fig. 18, where an incorrect count is obtained if a common ancestor that is not an LCA is used to merge 4 and 7. While in this example there is a unique LCA for 4 and 7, in general this may not be the case. With unrestrained branching and merging, there is no bound on the number of LCAs a pair of versions can have. For example, in Fig. 19a, the merge of 0 with 3 is preceded by two "criss-cross" merges between their respective branches[4] resulting in there being two LCAs (5 and 4) for 0 and 3. Multiple LCAs can occur even without criss-cross merges, as demonstrated by Fig. 19b. Concurrent versions with multiple LCAs do not lend themselves to three-way merging. If such versions are latest on their respective branches, they render the branches unmergeable (since `lca` is no longer a function) as demonstrated by examples in Fig. 19. Note that for both the examples in Fig. 19, no extension of the branching structure can make the branches merge again. Thus the system is effectively *partitioned* permanently. This is clearly a problem.

The problem of multiple LCAs also arises in the context of source control systems, which employ *ad hoc* mechanisms to pave the way for three-way merging. Git (Git 2017), for instance, recursively merges LCAs by default to compute a virtual ancestor, which then serves as the LCA for merging concurrent versions. This method is demonstrated in the branching structure for a mergeable, replicated counter as shown in Fig. 19a, where LCAs 5 and 4 of 0 and 3 are

[4] When discussing merges and LCAs, we often attribute the properties of latest versions on branches to the branches themselves. For instance, when we say two branches merge, in fact their latest versions merge. Likewise, LCA of two branches means the LCA of their latest versions.

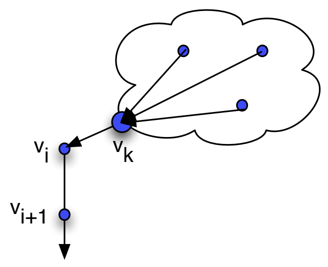


Fig. 20. The cloud represents the set of all ancestors of versions v_i and v_{i+1}, i.e., their causal history. The locus v_k of both is the version that succeeds every other ancestor in the causal history.

merged (with their LCA being 10) to generate -1 as the virtual LCA to merge 5 and 4. A major downside with this approach is that it makes no guarantees on the relationship between the virtual ancestor and its concurrent versions; the former may not even be a legal ancestor of the latter as per the semantics of the data type. For instance, suppose the integer type in Fig. 19a represents a bank account balance, which is expected to disallow any activity on the account if the balance is ever known to be less than zero. From the perspective of the library designer and its clients, there is no meaningful scenario in which versions 3 and 0 can emerge from -1, since the only transition allowed by the semantics from -1 is to itself. Clearly, *ad hoc* mechanisms like this are error-prone and difficult to apply in general.

Fortunately, unlike source control systems where branching structure is entirely dictated by the user, vml abstracts away branching structure from the programmer, and hence retains the ability to manifest it in a way that it deems fit. In particular, vml solves the problem of multiple LCAs by suitably constraining the branching structure such that the problem never arises. The constraints are imposed either implicitly, as a result of how the operational semantics defines an atomic step, or explicitly, by insisting that certain conditions be met before merging a pair of versions (E-Pull-Wait). First, the operational semantics already disables criss-cross merges since it only ever merges versions that are latest on their respective branches. Second, we impose certain pre-conditions on the merging branches to preempt the structure shown in Fig. 19b. The intuition is as follows: consider the branch b_3 at the instance of merging v_{12}. Since it has already merged v_{22}, v_{22} could be a common ancestor for b_3 and some other branch (call it b). Now, if b_3 merges v_{12}, the same could be true of v_{12}. Since v_{12} and v_{22} are not ordered by the ancestor relation, both become LCAs of b_3 and b. We observe that this scenario can be prevented if, when merging b_1, b_3 insists on an ancestor relation between the last merged version (v_{22}) and the currently merging version. We call the last merged version of a branch its *locus*. By maintaining a total order of among loci of a branch b as new versions are created, we effectively guarantee the presence of a version (the locus) that is lower than all the ancestors of the latest version on b. Next, by requiring the locus to be an ancestor of the merging version, we ensure that all the common ancestors have a single lowest element, thus enforcing the uniqueness of the LCA.

We now formalize the intuitions described above via a series of definitions that help us state the guarantees offered by the system.

*Definition 4.6 (**Internal and External Ancestors**).* Given a branch b and a version $v \in b$, an internal ancestor ($\preceq_i$) of v is an ancestor from the same branch b. An external ancestor ($\preceq_o$) of v is an ancestor from a different branch $b' \neq b$.

*Definition 4.7 (**Locus**).* Given a branch b and a version $v \in b$, the locus (v_o) of v is an ancestor that is not an ancestor of any other external ancestor of v. That is, $H \vdash v_o \preceq v$, and there does not exist a $v'_o \notin b$ such that $H \vdash v'_o \preceq v$ and $H \vdash v_o \preceq v'_o$. We lift the notion of locus to the level of branches by defining the locus of a branch as the locus of its latest version.

In a legal branching history, every version has a unique locus (property follows from Def. 4.8 below). We define the *causal history* of a version as the set of all ancestors of that version. A locus (v_o) of a version (v) is therefore the maximum element (lowest element in the ancestor relation) of the causal history of v's internal ancestor that was the result of the last merge. This is illustrated in Fig. 20. We make use of this observation while defining mergeability.

*Definition 4.8 (**Mergeability**).* Given a history H, a version v_1 and a version v_2 that is not an ancestor of v_1 under H, v_2 is mergeable into v_1 (denoted $H \vdash v_2 \rightsquigarrow_m v_1$) if and only if the locus (v_o) of v_1 is an ancestor of v_2, and no internal ancestor of v_1 that is not also an ancestor of v_o is an ancestor of v_2.

If v_1 and v_2 referred in the above definition are the latest versions on their respective branches b_1 and b_2, we say b_2 is mergeable into b_1, or more generally, b_1 and b_2 are mergeable. The definition essentially requires all of v_1's causal history that precedes its locus v_o to be included in v_2's causal history, and none of v_1's history that succeeds v_o to be included in v_2's history. This allows us to view versions v_1 and v_2 as having been independently evolved from a common causal history whose maximum (w.r.t the ancestor relation) is the version v_o. Consequently, v_o becomes the LCA for v_2's merge into v_1. Generalizing this observation for the latest versions on any two branches, we state the following theorem:

Theorem 4.9 (**Unique LCA**). *Every pair of branches in a legal history H, if mergeable as per Def. 4.8, have a unique lowest common ancestor.*

While the definition of mergeability is sufficient to enforce uniqueness of LCAs, it may hinder progress in the sense that there may not exist a pair of branches in a legal history satisfying the definition, thus making the progress impossible. The following theorem asserts that this is not the case:

Theorem 4.10 (**Progress**). *In a legal branching history H produced by the operational semantics, if two branches, b_i and b_j are not mergeable (as per Def. 4.8), then there exists a sequence of fork and merge operations (between mergeable versions) that can be performed on H to yield a new history H', where b_i and b_j are mergeable.*

The proof is by the induction on the size of branching history. The key observation is that the smallest history that is the union of causal histories of two unmergeable versions (call them v_i and v_j) is smaller than the history that includes the unmergeable versions. Since causal histories have maximum versions (respective loci), which, by the inductive hypothesis, can be made mergeable, we merge them to yield a new version v that is the maximum of the causal histories of both v_i and v_j. If v_i and v_j are the latest versions on their respective branches, the branches are now mergeable.

5 DISTRIBUTED INSTANTIATION

We describe the realization[5] of vml on top of a persistent distributed store and the language extensions to enable and enforce the constraints of the vml.

The vml programming model is realized on top of Irmin (Irmin 2016), an OCaml library database implementation that is part of the MirageOS project (Mirage 2013). Irmin provides a persistent multi-versioned store with a content-addressable heap abstraction. Simply put, content-addressability means that the address of a data block is determined by its content. If the content changes, then so does the address. Old content continues to be available from the old address. Content-addressability also results in constant time structural equality checks, which we exploit in our mergeable rope implementation (Section 3.1), among others.

Irmin provides support for distribution, fault-tolerance and concurrency control by incorporating the Git distributed version control (Git 2017) protocol over its object model. Indeed, Irmin is fully compatible with Git command line tools. Distributed replicas in vml are created by cloning a vml repository. Due to vml's support for mergeable types, each replica can operate completely independently, accepting client requests, even when disconnected from other replicas, resulting in a highly available distributed system.

Concurrent operations in Irmin are tracked using the notion of branches, allowing the programmer to explicitly merge branches on demand. vml's concurrency and distribution model is also realized in terms of branches; each replica operates on its own branch, and thus is isolated from the actions on other replicas. In the presence of network partitions, the nodes in one partition may not receive updates from the other partition, preventing merges from happening across a partition. This can be a problem if a pair of branches in one partition (the current partition) have multiple LCAs in the other partition (the remote partition), and hence cannot merge. Fig. 21 illustrate this problem for the branching structure of Fig. 19b.

Fortunately, access to full (causal) branching history assured by Irmin at every node comes to the rescue. Relying on the history, the current partition can fork-off (`fork`) new branches that start where the remote branches left,

[5]https://github.com/icfp2017/vml

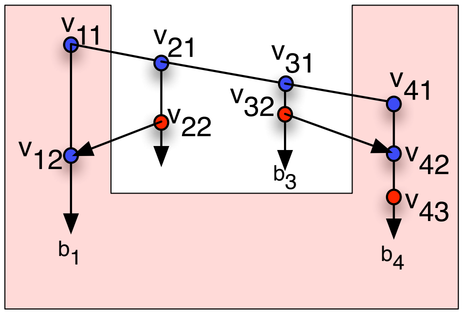


Fig. 21. Network partitions may leave branches b_1 and b_4 in one partition (red background), and branches b_2 and b_3 in the other (yellow background). Access to full history on both sides of the partition lets both sides make progress.

and map them to new (virtual) nodes that emulate the nodes in the remote partition. For the example in Fig. 21, the partition of b_1 and b_4 can fork new branches b'_2 and b'_3 from b_2 and b_3, respectively, and resume the activity as if the partition never happened. After merging the first versions on b'_2 and b'_3, branches b_1 and b_2 will be in the same state as before (when the partition happened), except that the branches are now mergeable because b'_2 and b'_3 can merge. The ability to track full provenance information is thus crucial for VML to overcome network partitions, making it an appropriate programming model for highly-available replicated data types. The VST conceals branching structure, but also transparently performs the necessary merges to obtain a history graph with a unique lowest common ancestor.

Importantly, Irmin supports user-defined three-way merge functions for reconciling concurrent operations. While Irmin's merge functions are defined over objects on Irmin's content-addressable heap, VML's merge functions are defined over OCaml types. We address this representational mismatch with the help of OCaml's PPX metaprogramming support (PPX 2017) to derive bi-directional transformations between objects on OCaml and Irmin heaps. We also derive the various serialization functions required by Irmin[6]. Synchronization between replicas is performed using Git's notion of pushing and pulling updates from remotes (Git Transfer Protocol 2017). As a result, we reap the benefits of efficient delta-transfer (only missing objects are transferred between replicas), compression and end-to-end encrypted communication between the replicas.

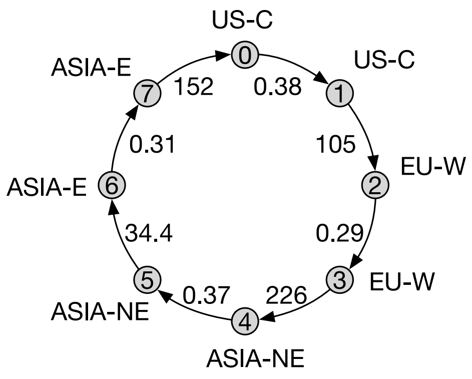
6 EVALUATION

In this section, we present an evaluation of VML programming model with the aim of assessing its practical utility for describing geo-distributed eventually consistent applications. In particular, we are interested in the horizontal scalability of applications built using VML. Typical distributed eventually consistent data stores use custom synchronization and dissemination protocols for transferring updates between replicas. In contrast, VML is realized over Irmin (Irmin 2016) which uses Git transfer protocol (Git Transfer Protocol 2017), a protocol not designed with distributed data stores in mind. Hence, we also evaluate the performance of synchronization and merging.

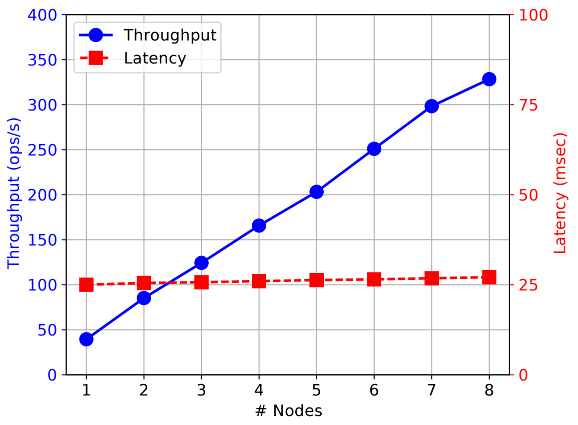
6.1 Benchmark: Collaborative editing

To evaluate the effectiveness of our model, we have implemented a collaborative editing application that simulates concurrent editing of the same document by several authors. This application is not unlike Google Docs, but differs from it by the fact that we do not need a central server that coordinates the edits. Instead, edits from each author is asynchronously sent to other authors. The shared document itself is represented by a mergeable rope of characters (Section 2), and hence, the remote edits can be correctly reconciled into each local document. We resolve conflicts on concurrent substitutions at the same position (such as two authors concurrently editing `"abc"` to `"xbc"` and `"ybc"`, respectively) by choosing the character with the smaller ASCII code. For the benchmarks, the initial document that we use has 1576 characters. The workload consists of 4000 edit operations at random indices with 85% insertions and 15% deletions.

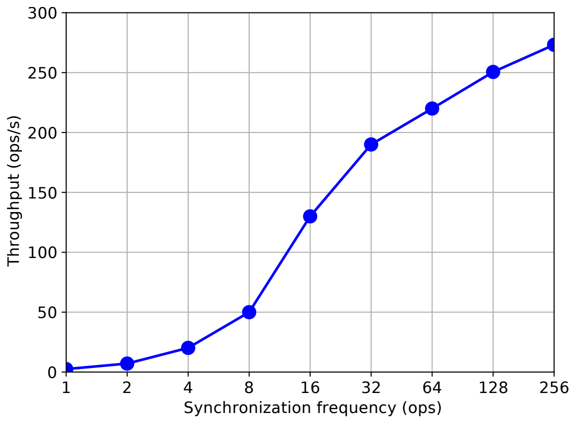
[6] The Since VST monad relies on Irmin for realizing the branch structure, it is in fact parameterized over a mergeable type that implements all these functions. Thus the monad is functorized over the mergeable type, but not polymorphic. The convenience of type parameter inference that polymorphism offers can however be obtained for module parameters using OCaml extensions, such as Modular Implicits (White 2014).



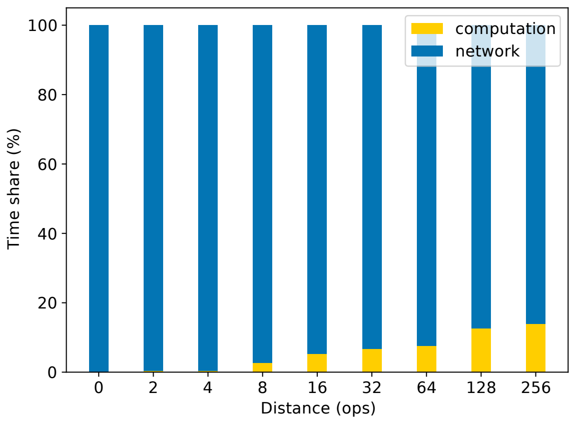
(a) Our experimental configuration consists of an 8-node ring cluster executing on Google Cloud Platform. US-C, EU-W, ASIA-NE and ASIA-E are availability zones us-central1-b, europe-west1-b, asia-northeast1-b and asia-east1-b respectively. Edge labels are inter-node latencies in milliseconds.



(b) Scalability: Overall throughput of the cluster and latency of each operation.



(c) Synchronization: Overall throughput of the cluster while actively synchronizing with the successor node.



(d) Merge performance: Cost of merging concurrent operations across nodes.

Fig. 22. VML performance evaluation.

6.2 Experimental setup

For our experiments, we instantiate an 8-node geo-distributed cluster using the Google Cloud Platform (GCP 2017). Each node is an `n1-standard-4` instance with 2 virtual CPUs and 7.5 GB of memory. The nodes are arranged in a ring as shown in Figure 22a and synchronize (push and pull updates) when necessary with its successor. While VML does not impose any restrictions on synchronizing with other nodes, we choose to only synchronize with the successor since we are guaranteed eventual convergence with a ring cluster. If the successor becomes unavailable, the node synchronizes with other available nodes. Operations are generated and applied locally on each node before they are propagated eventually to other nodes with the help of a background thread that synchronizes with its successor every second.

6.3 Results

We evaluate the scalability of concurrent editing application by increasing the cluster size from 1 to 8 (the 4 node ring cluster consists of nodes numbered 0 to 3), with each node performing concurrent edits to the same document. In each case, we measure the overall cluster throughput and latency of each operation. The results are presented in

Figure 22b. The results show that the cluster throughput increases linearly with the number of concurrent editors, while the latency for each operation remains the same. This is because each operation is performed locally and does not require synchronization with other nodes. The nodes remain available to accept requests even if the node gets disconnected. Since the document type is mergeable, eventually when the node comes back online, the updates are synchronized with the cluster.

While updates are propagated asynchronously, we also measured the impact of propagating updates synchronously by forcing synchronization every few operations. This corresponds to a partially synchronous system, which is also unavailable; if the cluster is unreachable, local operations will not be accepted. While this is not a realistic deployment, the results help us understand the overhead of synchronization. The results are shown in Figure 22c. The result show that frequent blocking synchronization is prohibitively expensive with the throughput dropping to 3 ops/s for synchronizing after every operation. VML uses Git transfer protocol for synchronization which involves multiple round trips between the nodes for synchronization. But as the synchronizations get less frequent, we can see that the throughput asymptotically approaches the asynchronous case (Figure 22b).

Finally, we evaluate the performance of merge functions. For this experiment, we consider the nodes 1 (`us-central1-b`) and 2 (`europe-west1-b`). Both nodes start with the initial document and perform *n* concurrent edits without synchronization. After this, the nodes synchronize pushing the local edits and pulling the remote edits, and merge to produce the final document. In this case, we indicate the distance of synchronization as $2n$. Figure 22d presents the time share between network and computation for performing 3-way merge as we increase the operation distance between the remote branches. We see that the merge time is dominated by the network overhead. This is because of the transfer protocol involving multiple round trips and the need to transfer objects not only for application objects but also Git internal objects for history, directory structure and branches. We observe that we can improve the network performance by implementing a streaming protocol that eagerly transfers objects.

More importantly, the cost of computation grows sub-linearly. This is because of efficient merge function on ropes (Sec. 3.1) enabled by Irmin's content-addressed storage backend, and more expensive operational transformation of merging concurrent edits to leaf nodes (Sec. 3.2) is invoked rarely on small. On average, we measured that with a distance of 256, for merging the two documents with 1743 characters, 50 list merges were performed on strings of average length 13. A purely list based approach would require a polynomial time edit sequence algorithm to be applied on the entire document twice. Compared to this, the rope implementation has much better algorithmic complexity. While not depicted here, we measured that by representing the entire document using a mergeable list type, in addition to slowing down each edit operation, the merge at a distance of 256 was 2.3 times slower than the rope implementation. We conclude that versioning and merging as found in VML can yield substantial throughput and latency gains with low computational cost on modern eventually-consistent cloud systems.

7 RELATED WORK

Our idea of versioning state bears resemblance to Concurrent Revisions (Burckhardt et al. 2010), a programming abstraction that provides deterministic concurrent execution. The idea of using revisions as a means to programming eventually consistent distributed systems was further developed in (Burckhardt et al. 2012). The VML programming model, however, differs substantially from a concurrent revisions model because it imposes no distinction between *servers*, machines that hold global state, and *clients*, devices that operate on local, potentially stale, data - any computation executing in a distributed environment is free to fork new versions, and synchronize against other replicated state. Our model, furthermore, supports fully decentralized operation and is robust to network partitions and failures. Just as significantly, VML allows applications to customize join semantics with programmable merge operations. Indeed, the integration of a version-based mechanism within OCaml allows a degree of type safety, composability, and 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 both approaches, coordination among replicas involves transmitting operations on replicated objects that are performed locally on each replica. In contrast, VML allows programmers to use familiar state-based and functional abstractions when developing distributed applications. As we illustrated in Fig. 1b, using effects and operations to coordinate the activities of replicas may involve addressing subtleties that would not manifest in their absence. Our case studies and experimental results support our contention that using well-understood state (heap)-based abstractions to build distributed applications greatly simplifies program reasoning and eases development, without compromising efficiency.

Modern distributed systems are often equipped with only parsimonious data models (e.g., key-value model) and poorly understood low-level consistency guarantees[7] that complicate program reasoning, and make it hard to enforce application integrity. Some authors (Bailis et al. 2013) 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.

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. 2011b; Bailis et al. 2014a; Balegas et al. 2015a; Gotsman et al. 2016; Li et al. 2014a, 2012). 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 (Lesani et al. 2016; Wilcox et al. 2015). vml seeks to provide a rich high-level programming model, built on rigorous foundations, that can facilitate program reasoning and verification. An important by-product of the programming model is that it 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 a merge operator, a function that can be often easily written for many common datatypes.

Several conditions have been proposed to judge whether an operation on a replicated data object needs coordination or not. (Alvaro et al. 2011a) defines *logical monotonicity* as a sufficient condition for coordination freedom, and proposes a consistency analysis that marks code regions performing non-monotonic reasoning (eg: aggregations, such as `COUNT`) as potential coordination points. (Bailis et al. 2014b) and (Li et al. 2014b) define *invariant confluence* and *invariant safety*, respectively, as conditions for safely executing an operation without coordination. (Balegas et al. 2015b) requires programmers to declare application semantics, and the desired application-specific invariants as formulas in first-order logic. It performs static analysis on these formulas to determine I-offender sets - sets of operations, which, when performed concurrently, result in violation of one or more of the stated invariants. For each offending set of operations, if the programmer chooses invariant-violation avoidance over violation repair, the system employs various techniques, such as escrow reservation, to ensure that the offending set is effectively serialized. All of these approaches differ significantly from the core goals of vml, which is to enable seamless application-driven techniques for programming geo-replicated systems. The vml programmer is not concerned with different system-specific consistency or isolation levels, or invariants that are sensitive to these levels. Instead, the only requirements demanded of the developer is the need to reason about a sensible merge semantics for data structures. Such reasoning can be applied without consideration of system- or architecture-specific details. This reasoning phase implicitly expresses salient semantic invariants without having to be exposed to low-level operational details.

There have been numerous proposals over the years for realizing distributed programming functionality in a functional programming context. Erlang (Armstrong 2007), which is the most well-known, is based on a "shared-nothing" message-passing programming methodology. Poly/ML (Matthews 1997) and Facile (Thomsen et al. 1993) extend Standard ML, while Acute (Sewell et al. 2005) and HashCaml (Billings et al. 2006) extend OCaml, with distributed programming abstractions and support, with particular focus on type-safe marshalling and mobility. Kali-Scheme (Cejtin et al. 1995) defines a distributed extension for Scheme-48, and Cloud Haskell (Epstein et al. 2011) describes a domain-specific language that enables development of distributed Haskell programs. Obliq (Cardelli 1995) explores issues related to mobility, lexical scoping, and network references in an object-oriented framework that enables safe (i.e., lexical) transmission of closures. Orleans (Bykov et al. 2011) provides persistent actor-based programming model for elastic cloud application, but the focus is on strong consistency, mirroring a single-threaded execution.

However, unlike vml, none of these systems consider issues of object replication or consistency as central to their programming models. The realities of modern-day scalable cloud-based distributed systems, especially those that are constructed from geographically distributed participants, dictate that we must necessarily grapple with consistency issues to achieve any kind of reasonable performance; it is noteworthy that replication is inherent in

[7]Cassandra (Lakshman and Malik 2010), a popular NoSQL data store, comes with various consistency enforcement mechanisms, such as anti-entropy protocols, QUORUM and LOCAL_QUORUM reads and writes, and light-weight transactions, each of which can be controlled via configuration knobs or runtime parameters.

many widely-used cloud-based distributed system implementations today (e.g., Cassandra (Lakshman and Malik 2010) or Dynamo (DeCandia et al. 2007)), which provide only weak consistency guarantees. Rather than exposing a low-level operational treatment of replication, foisting the burden of reasoning about replicated state onto the programmer, our primary contribution is the development of a principled high-level approach to framing these issues that abstracts these kinds of system-level details. Our goal is to leverage functional programming principles to minimize disruption to the way programmers structure and reason about their applications, without compromising efficiency or performance. Indeed, the semantics of mergeable types and versioning has utility in any concurrent setting that must deal with non-trivial coordination and synchronization costs, even without taking distribution or replication issues into account.

Finally, VML shares some resemblance to conflict-free replicated data types (CRDT) (Shapiro et al. 2011b). 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 VML need not commute and the reconciliation protocol is defined by user-defined merge functions. Moreover, CRDTs are not composable and each CRDT tends to be built from scratch over the network protocol. Compared to this, VML types are composable; mergeable ropes are polymorphic over its mergeable content. VML uses 3-way merges using the lowest common ancestor, which is critical for all of our user-defined merges. However, CRDTs do not have the benefit of lowest common ancestor for merges and are only presented with the two concurrent versions. If a 3-way merge is desired, then the causal history has to be explicitly encoded in the data type. As a result, constructing even simple data types like counters are more complicated using CRDTs (Shapiro et al. 2011b) compared to their implementation in VML.

VML uses 3-way merges using the lowest common ancestor, which is critical for all of our user-defined merges. However, CRDTs do not have the benefit of lowest common ancestor for merges and are only presented with the two concurrent versions. If a 3-way merge is desired, then the causal history has to be explicitly encoded in the data type. As a result, constructing even simple data types like counters are more complicated using CRDTs (Shapiro et al. 2011b) compared to their implementation in VML. CRDTs also tend to be implemented directly over the network protocols. Hence, low-level concerns such as duplicate delivery, lost messages, message reordering are explicitly handled in the data type definition. In VML, the programmer does not have to deal with underlying network issues since VML is realized on Irmin, a high-level branch-consistent distributed store.

8 CONCLUSIONS

Replication is a critical feature in geo-distributed systems used to improve latency and availability. Existing approaches to dealing with replicated data often involve complex and subtle code restructuring, and the use of specialized datatypes that interact poorly with other language structures. VML is a programming model that extends ML with implicit support for replication that overcomes these concerns. Its notion of concurrency and synchronization is encapsulated within a monad that considers distrbuted computation in terms of typed versions of replicated state. Notably, any ML data structure can participate in a distributed computation using this monad by providing a merge function that dictates how different versions of that structure can be reconciled. Our versioning semantics enjoys strong consistency and progress guarantees that make it useful in a fully decentralized (and unreliable) cloud environment. Furthermore, our experimental results demonstrate that using such high-level abstractions need not come at the expense of efficient implementations.

REFERENCES

Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. 2011a. 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. http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf

Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. 2011b. 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.

Joe Armstrong. 2007. *Programmoing Erlang: Software for a Concurrent World.* Pragmatic Bookshelf.

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

Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014b. Coordination-Avoiding Database Systems. *CoRR* abs/1402.2237 (2014). http://arxiv.org/abs/1402.2237

Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. 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. DOI:http://dx.doi.org/10.1145/2463676.2465279

Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2015a. 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

Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2015b. 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

John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. 2006. Type-safe Distributed Programming for OCaml. In *Proceedings of the 2006 Workshop on ML (ML '06)*. ACM, New York, NY, USA, 20–31. DOI: http://dx.doi.org/10.1145/1159876.1159881

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. DOI: http://dx.doi.org/10.1145/3009837.3009888

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

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. DOI: http://dx.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. DOI: http://dx.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. DOI: http://dx.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

Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*. ACM, New York, NY, USA, Article 16, 14 pages. DOI: http://dx.doi.org/10.1145/2038916.2038932

Luca Cardelli. 1995. A Language with Distributed Scope. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 286–297. DOI: http://dx.doi.org/10.1145/199448.199516

Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. 1995. Higher-order Distributed Objects. *ACM Trans. Program. Lang. Syst.* 17, 5 (Sept. 1995), 704–739. DOI: http://dx.doi.org/10.1145/213978.213986

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 251–264. http://dl.acm.org/citation.cfm?id=2387880.2387905

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 205–220. DOI: http://dx.doi.org/10.1145/1294261.1294281

Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 54–70. DOI: http://dx.doi.org/10.1145/2815400.2815425

Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. ACM, New York, NY, USA, 118–129. DOI: http://dx.doi.org/10.1145/2034675.2034690

GCP 2017. Google Cloud Platform. (2017). Accessed: 2017-01-04 10:12:00.

Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59. DOI: http://dx.doi.org/10.1145/564585.564601

Git 2017. Git: a free and open source distributed version control system. (2017). Accessed: 2017-01-04 10:12:00.

Git Transfer Protocol 2017. Git Internals – Transfer Protocols. (2017). Accessed: 2017-01-04 10:12:00.

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. DOI: http://dx.doi.org/10.1145/2837614.2837625

Irmin 2016. (2016). Irmin: https://mirage.io/blog/introducing-irmin.

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review* 44, 2 (April 2010), 35–40. DOI: http://dx.doi.org/10.1145/1773912.1773922

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. DOI: http://dx.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. 2012. 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. DOI: http://dx.doi.org/10.1145/2043556.2043593

David Matthews. 1997. *Concurrency in Poly/ML*. Springer New York, New York, NY, 31–58. DOI: http://dx.doi.org/10.1007/978-1-4612-2274-3_3

Mirage 2013. A programming framework for building type-safe, modular systems. (2013). https://mirage.io Accessed: 2017-01-03 12:21:00.

Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *Journal of the ACM* 26, 4 (Oct. 1979), 631–653. DOI: http://dx.doi.org/10.1145/322154.322158

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

Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. 2005. Acute: High-level Programming Language Design for Distributed Computation. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, New York, NY, USA, 15–26. DOI: http://dx.doi.org/10.1145/1086365.1086370

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*. Springer-Verlag, Berlin, Heidelberg, 386–400. http://dl.acm.org/citation.cfm?id=2050613.2050642

Marc Shapiro, Nuno Preguia, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Dfago, Franck Petit, and Vincent Villain (Eds.). Lecture Notes in Computer Science, Vol. 6976. Springer Berlin Heidelberg, 386–400. DOI: http://dx.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. DOI: http://dx.doi.org/10.1145/2737924.2737981

Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 385–400. DOI: http://dx.doi.org/10.1145/2043556.2043592

Bent Thomsen, Lone Leth, and Alessandro Giacalone. 1993. *Some Issues in the Semantics of Facile Distributed Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 563–593. DOI: http://dx.doi.org/10.1007/3-540-56596-5_47

Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, London, UK, UK, 24–52. http://dl.acm.org/citation.cfm?id=647698.734146

Robert A. Wagner and Michael J. Fischer. 1974. The String-to-String Correction Problem. *J. ACM* 21, 1 (Jan. 1974), 168–173. DOI: http://dx.doi.org/10.1145/321796.321811

White 2014. Modular Implicits. In *ML Workshop*.

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. DOI: http://dx.doi.org/10.1145/2737924.2737958