Mergeable Types

Gowtham Kaki
Purdue University

KC Sivaramakrishnan
University of Cambridge

Samodya Abeysiriwardane
Purdue University

Suresh Jagannathan
Purdue University

Distributed applications often eschew strong consistency and replicate data asynchronously to improve availability and fault tolerance. However, programming under eventual consistency is significantly more complex and often leads to onerous programming model where inconsistencies must be handled explicitly. We introduce VML, *a programming model that extends ML datatypes with mergeability à la version control systems with the ability to define and compose distributed ML computations around such data. Our OCaml implementation instantiates mergeable types on Irmin, a distributed content-addressible store to enable composable and highly-available distributed applications.*

1 A Replicated Counter

Consider a monotonic counter data type:

```
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
```

Observe that the library is written in an 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 replica 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 distributed transaction support, which impose significant performance penalties. But, without such heavyweight mechanisms, applying an `Add` operation on one replica may not be instantaneously witnessed on another, which may be in the process of simultaneously attempting to perform its own `Add` or `Mult` action. Since `Add` and `Mult` do not commute, this may result in divergence of the counter state across various replicas.

Rather than viewing each operation in terms its effect on the global state, can we formulate a more declarative interpretation, directly in terms of the counter value maintained by each replica? Since the counter is replicated, each local operation can be thought of as yielding a new local version, collectively producing a version tree, with one branch for each replica. Every branch represents different (immutable) versions maintained by different replicas, with the state produced by the computation performed over a counter on a replica recorded along the replica'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 replicas that comprise a distributed application with merge operations serving to communicate and reconcile different local states.

2 Collaborative drawing

VML not only supports primitive data types but also algebraic data types. This code snippet:

```
module type CANVAS = sig
  type pixel = {r:char; g:char; b:char}
  type tree =
  | N of pixel
  | B of {tl_t:tree; tr_t:tree; bl_t:tree; br_t:
     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
```

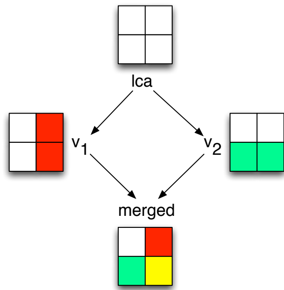
shows the signature of the Canvas application. Canvas represents a free-hand drawing canvas in terms of a tree of quadrants. A quadrant is either a leaf replica containing a single pixel (an `r-g-b` tuple), or a tree of sub-quadrants, if the quadrant contains multiple pixels of different colors. Quadrants 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.

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.

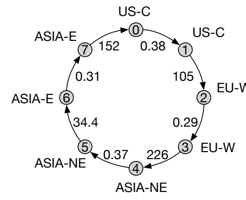
VML requires a three-way `merge` function to merge concurrent versions of a drawing canvas that includes two concurrent versions (`v1` and `v2`), and their lowest common ancestor (`lca`) - the version from which the two concurrent versions evolved independently.

```
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; tr_t; bl_t; br_t}
  | (_, N px1, N px2) →
    (* pixels are merged by mixing colors *)
    let px' = color_mix px1 px2 in N px'
```

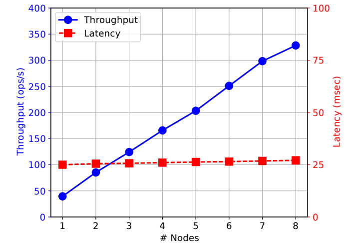
The merge function can make use of the pixel values of the common ancestor to merge the pixel values on both the canvases. For instance, if the color of a pixel in `v1` is white, and in `v2` it is green, and its color in `lca` is white, then it means that only `v2` modified the color. Hence the pixel 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 below.



We have built several mergeable datatypes including lists, ropes, etc, which can be freely composed together. That is, a list of counters behaves like a mergeable list for append and remove operations, with updates reconciled through counter merge semantics.



(a) Our experimental configuration consists of an 8-node ring cluster executing on Google Cloud Platform. Edge labels are inter-node latencies in milliseconds.



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

Figure 1: VML performance evaluation.

3 Distributed Instantiation

The VML programming model is realized on top of Irmin [2], an OCaml library database implementation that is part of the MirageOS project [3]. 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, among others.

Irmin provides support for distribution, fault-tolerance and concurrency control by incorporating the Git distributed version control [1] 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.

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 [4] to derive bi-directional transformations between objects on OCaml and Irmin heaps. We also derive the various serialization functions required by Irmin

We evaluated the performance of the system on a collaborative application that simulates concurrent editing of the same document by several authors. The benchmark itself was constructed with a list of ropes. The workload consists of 4000 edit operations at random indices with 85% insertions and 15% deletions. 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 1b. 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.

References

[1] Git: a free and open source distributed version control system, 2017. Accessed: 2017-01-04 10:12:00.

[2] 2016. Irmin: https://mirage.io/blog/introducing-irmin.

[3] A programming framework for building type-safe, modular systems, 2013. Accessed: 2017-01-03 12:21:00.

[4] PPX extension points, 2017. Accessed: 2017-01-04 10:12:00.