Generalizing CRDTs as Uniquely-Indexed Foldable Data Structures over Semilattices

Aesa Kamar

October 2021

Abstract

Conflict-free replicated data types are used ubiquitously throughout distributed systems as a means of sharing data. Development of CRDTs requires mathematical rigor and testing to demonstrate correctness. This paper provides a way to use type-theoretic programming which allows the use of more complex data types as CRDTs. Because we are able to encode proofs into they type system directly, we can decouple the complexity of building provably correct CRDTs into orthogonal pieces which can individually be proved for correctness, then composed together. We provide some examples in the Scala programming language for how to encode these concepts.

1 Introduction

In distributed applications where many decentralized compute nodes communicate via networks, it is useful to have copies of a common data-structure present on each node. In peer-to-peer decentralized applications, each node has a local copy of its own data-structure; it may also make local edits which we would like to make available on all other nodes in the cluster. In order to make local edits available to other nodes, each node must broadcast edits to other members of the cluster, as well as receive edits from other members. A common technique is to use a Gossip-style protocols running on each node to broadcast updates to a a random set of cluster members. Over several rounds, it is statistically likely that all nodes will receive a copy of edits from any given node. (TODO cite) Once edits from other nodes are received, a combining operation must be performed to merge them into the local copy such that the local copy will be consistent with other copies once all messages have been applied and received.

Because peer-to-peer decentralized applications are asynchronous and separated by networks, they must contend with out-of-order message delivery as well as at-least-once message delivery patterns. In order to guarantee convergence of the merge operation given out-of-order message delivery patterns, it is necessary to prove the commutativity law holds on the merge(message: M). (TODO Why do I need to do this?) Similarly, in order to guarantee that state

does not diverge in repeated application of a message, it is necessary to prove that the idemopotency law holds on the ‘merge(message: M)‘