Coordination-free Collaborative Replication

based on Operational Transformation

Masato Takeichi

takeichi@acm.org

December 16, 2024

Abstract

- Coordination-free Collaborative Replication (CCR) maintains consistency across replicas in distributed systems without explicit coordination messages.
- CCR uses Operational Transformation (OT) to ensure consistency by transforming operations for data integrity across replicas.
- Unlike Conflict-free Replicated Data Type (CRDT) often resulting in counterintuitive behaviors, CCR employs intuitive conflict-free replication.
- CCR addresses inefficiencies in messaging by developing a versatile Coordination-free Protocol for Data Confluence.

Introduction

- A 2P-Set CRDT shopping cart (A, R) to which item x is added as $(A \cup \{x\}, R)$ and from which x is removed as $(A, R \cup \{x\})$ so that A and R are always increasing, with the cart contents $A \setminus R$.
    - We cannot add the item x again when x has once been added and then removed.
- A CCR shopping cart C to which x is added as $C \cup \{x\}$ if $x \notin C$, and from which x is removed as $C \setminus \{x\}$ if $x \in C$.
    - This cart enables us to add x once having been removed.
- CCR provides an intuitive and more efficient practical solution for collaborative data sharing in distributed systems.

Collaborative Replication

Shattering the CALM for Collaborative Data

- The CALM (Consistency As Logical Monotonicity) theorem gives a solution to the question: "What is the family of problems that can be consistently computed without coordination, and what lies outside that family?" as "A program has a consistent, coordination-free distributed implementation if and only if it is monotonic."
- CALM leverages static analysis to certify the state-based convergence properties provided by CRDTs which provide a framework for monotonic programming patterns.

Monotonicity is not the only golden rule!

- An operation is confluent if it produces the same outputs for any nondeterministic ordering of a set of inputs.
- If the outputs of one confluent operation are consumed by another confluent operator as inputs, the resulting composite operation is confluent.
- Hence, if we restrict ourselves to build programs by composing confluent operations, our programs are confluent by construction, despite orderings of messages or execution steps within and across sites of distributed systems.

OT is Disliked in Replication, but ...

* OT originally proposed in 1989 has been kept away from replicated data sharing since many algorithms were proved to be wrong.
* To make matters worse, most of OT algorithms assume dedicated servers for conflict resolution of the clients and not suitable for P2P replication.

Insight into OT-based collaborative replication:

* Every updating process is a sequence of operations transmitted between P2P sites.
* Received operations are transformed to include the local operations done since the last common data.
* They are then applied locally to get the new data.

OT-based Collaborative Replication

Updating Operations

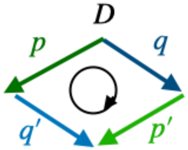
- The set O of operations on data $D \in \mathcal{D}$ is a *monoid* $(O, !, \odot)$ with the identity operation "!" with several basic operations as generators and $\odot :: O \times O \to O$.
- Updating operator $p \in O$ applied to D is generically written as $D \odot p$ using a symbol $\odot :: \mathcal{D} \times O \to \mathcal{D}$. Thus, $D \odot p \odot q = (D \odot p) \odot q = D \odot (p \odot q)$
- The sequence of updating operators $p_1, p_2, \cdots$ called *patch* is of course an operator in O.
- The equality of $D \odot p$ and $D \odot q$ defines the equivalence relation $p \leftrightarrow_D q$ on O and the equality of operators $p =_D q$.

Example of Updating Operations

- Basic operators for text editing are $(\text{Ins } k\ t) \in O$ for inserting string t at position k into the replicated text, and $(\text{Del } k\ n) \in O$ for deleting n characters at k from the replica.
- These operations are "generators" of the monoid.
- Successive application of these generators to the data is expressed as $\langle \text{Ins } k_1\ t_1, \text{Del } k_2\ n_2, \text{Ins } k_3\ t_3 \rangle$.
- It is a representation of the element $(\text{Ins } k_1\ t_1 \odot \text{Del } k_2\ n_2 \odot \text{Ins } k_3\ t_3) \in O$ of the monoid $(O, !, \odot)$.
- Note that the element of the monoid may have several equivalent representations.

Operational Transformation

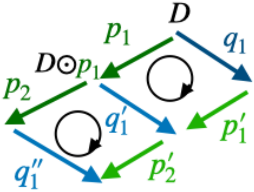
- Operational Transformation $T_D :: O \times O \to O \times O$ takes operations p and q on D and gives a pair of operations p' and q' by $(p', q') = T_D(p, q)$.



$D \odot p \odot q_p = D \odot q \odot p_q$

$(p', q') = T_D(p, q)$

TP1 Confluence Property



$(p'_1 \odot p'_2, q''_1) = T_D(p_1 \odot p_2, q_1)$

Compositional Property

- Local TP-1 Confluence Property T_D takes p and q on D and produces $(p', q') = T_D(p, q)$ which makes $D \odot p \odot q' = D \odot q \odot p'$.
  - * p and q on the same D are made into the common confluent state by T_D.
- Minimal Property If $(p', q') = T_D(p, q)$ holds, there is no p'', q'', $r_p \neq !$, $r_q \neq !$ satisfying $(p'', q'') = T_D(p, q)$ with $p' = p'' \odot r_p$, $q' = q'' \odot r_q$.
  - * No extra operations are introduced by T_D.
- Symmetric Property If $(p', q') = T_D(p, q)$ and $(q'', p'') = T_D(q, p)$, then $p' = p''$ and $q' = q''$ hold.
- Compositional Property If $(p_1', q_1') = T_D(p_1, q_1)$ and $(p_2', q_1'') = T_{D \odot p_1}(p_2, q_1')$, $T_D(p_1 \odot p_2, q_1)$ gives $(p_1' \odot p_2', q_1'')$.

Implication of Basic Properties

- In case $(O, !, \odot)$ is a semigroup and p and q have inverses p^{-1} and q^{-1}, $T_D(p, q) = (q^{-1}, p^{-1})$ always makes D into the confluent data $\dot{D}$.
  - * However, no one would like to use this transformation.
- Symmetric Property is essential for server-less P2P replication.
  - * No dedicated site for the server to control the update propagation to clients.
  - * Most OT for text editing are non-symmetric because it is aimed solely at the server-clients system.
- Compositional Property assures the transformation defined for the basic operations can be extended to accept any operations of the monoid $(O, !, \odot)$.

Building up Confluence Property of OT

$T_D :: O \times O \to O \times O$, $(p_q, q_p) = T_D(p, q)$ for p in site P and q in Q fulfills *TP1-Confluence* $p \odot q_p \leftrightarrow_D q \odot p_q$, which is written as $p \#_D q$.

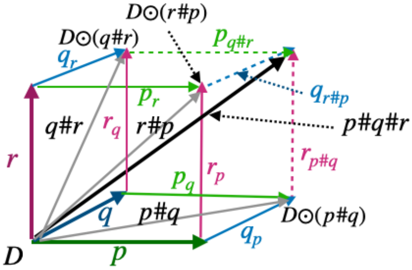
- Idempotence When P and Q share p applied to D to get the confluent state $D \odot p$ and hence $p \#_D p = p$.
- Commutativity It is straightforward that $p \#_D q = q \#_D p$ for p and q on D.
- Associativity To establish the confluence property of T_D applied in two steps for updates on D by three sites, the relations should hold regardless of the application order:
  $(p \#_D q) \#_D r = (p \#_D r) \#_D q$, $(q \#_D r) \#_D p = (q \#_D p) \#_D r$, and $(r \#_D p) \#_D q = (r \#_D q) \#_D p$.

More on Idempotence Property

- It is subtle to define $T_D(p, p) = (!, !)$ for $p \#_D p = p$.
- In the text editing application, a conflict may occur by operations $p = (\text{Ins } k\ x)$ and $q = (\text{Ins } k\ y)$.
- We may make both effective by shifting q's k to $k' = k + \mathit{length}(x)$ to define $T_D(\text{Ins } k\ x, \text{Ins } k\ y) = (\text{Ins } k\ x, \text{Ins } k'\ y)$. However, if x and y are equal, this contradicts $T_D(p, p) = (!, !)$ and $p \#_D p = p$.
- What happens there? Above p and q look the same for $x = y$ in spite of their independence.
- Hereafter to make the independence explicit by attaching the site ID $\$i$ as $(\text{Ins } \$i\ k\ x)$ for the internal representation, while the external representation remains as $(\text{Ins } k\ x)$.

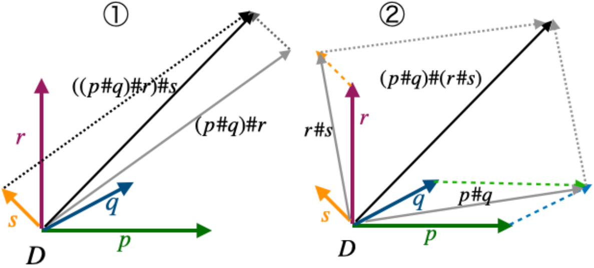
Associativity of Confluence Relation

- Along with commutativity, these are equivalent to $(p\#_D q)\#r = p\#_D(q\#_D r)$.
- Hence, these relations are written without parentheses as $p\#_D q\#_D r$.



TP2-Confluence Property

- The above confluence property is a case of *TP2-Confluence Property* which requires a confluence for more than three sites in general.
- Updates of four sites by three applications of T_D; the last one may take an operation s of S with the confluent operation $p\#_D q\#_D r$ of the three, or take the confluent operations $p\#_D q$ and $r\#_D s$ of the two.



Algebraic Structure of Confluence Property

- **Identity element** $p \#_D ! = \ ! \#_D p =_D p$.
- **Idempotence** $p \#_D p = p$, which comes from the Minimal Property of T_D
- **Commutativity** $p \#_D q = q \#_D p$, which comes from the TP1-Property of T_D
- **Associativity** $(p \#_D q) \#_D r = p \#_D (q \#_D r)$, which is required for the TP2-Property of T_D

The Algebraic Structure of Confluence Property suggests us to put the TP2-Confluence Property into practical coordination-free replication by sending updated operations on the common replicated data in arbitrary order to others.

Coordination-free Replication Protocol

Replication Protocol of Operation-based CRDT

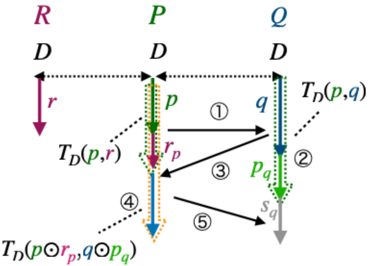
- Operation-based CRDT (CmRDT) exchanges updated operations by Reliable Causal Broadcast (RCB), while State-based CRDT (CvRDT) merges data states.
- RCB sends updates to the other site again in case of disconnection after a local update, and makes every local update at each site be eventually reflected at other sites.
- CmRDT does not confirm arrivals of messages sent from other sites, but RCB resends messages which have been arrived under the hood of CmRDT.

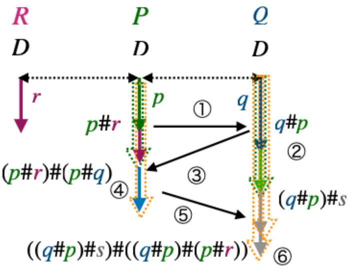
Replication Protocol of OT-based Collaboration

- OT-based Collaborative Replication shares similarities with CmRDT, but does not follow RCB.
- OT-based replication also realizes that all the local updates eventually arrive on all sites.
- In OT-based replication, each site receives a patch (sequence of operations) to be transformed and applied its result to the local replica in collaboration.
- Unlike CmRDT, operations in OT-based replication are not necessarily commutative, so the result of replication depends on the application order.
- The OT's Compositionality guarantees replication of more than three sites by applying OTs for two, which realizes the Coordination-free Collaboration.

Asynchronous Messaging for Collaboration - 1

1. P sends p to Q and other sites at its convenience.
2. Q computes $(p_q, q_p) = T_D(p, q)$ to get $q \odot p_q$ that makes D into the confluent state $D \odot q \odot p_q$.
   - * Note that $q \odot p_q$ is a representation of $q \#_D p = p \#_D q$.
   - * Q keeps $q \odot p_q$ for its local representation of $q \#_D p$.
3. Then, Q sends $q \#_D p$ to other sites including P.
   - * Q broadcasts the representation $q \odot p_q$ of $q \#_D p$.



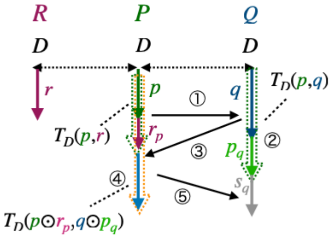


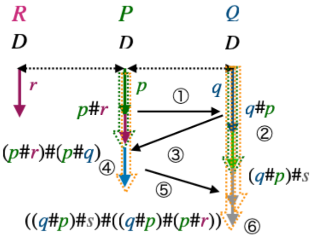
Asynchronous Messaging for Collaboration - 2

④ When P receives $q \odot p_q$ sent from Q after its state D has been updated to $D \odot p \odot r_p$ by some r from R, P makes $p \odot r_p$ and $q \odot p_q$ confluent into $(p \odot r_p) \#_D (q \odot p_q)$ by $T_D(p \odot r_p, q \odot p_q)$.

⑤ P broadcasts this to other sites.

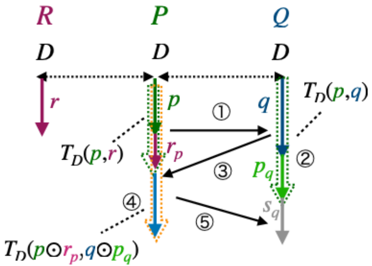
- $p \odot r_p$ is a rep of $p \#_D r$ and $q \odot p_q$ is of $q \#_D p = p \#_D q$
- $(p \odot r_p) \#_D (q \odot p_q) \leftrightarrow_D$ $(p \#_D r) \#_D (p \#_D q) = p \#_D (r \#_D q)$.

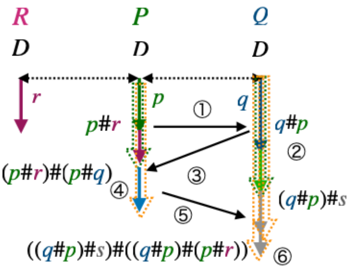




Asynchronous Messaging for Collaboration - 3

⑥ Q receives $(p\#_D r)\#_D(p\#_D q)$ after having reflected S's s as $(q\#_D p)\#_D s$ and computes a representation of operation $((q\#_D p)\#_D s)\#_D((p\#_D r)\#_D(p\#_D q))$ $= (q\#_D p)\#_D(s\#_D(p\#_D r))$.
This can be read as "Q's operation q is first made confluent with p and then with $s\#_D(p\#_D r)$, which has been made confluent with s and $p\#_D r$", and ...



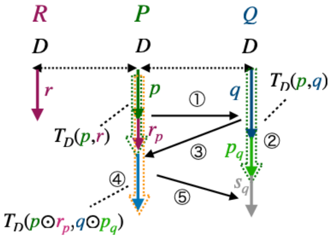


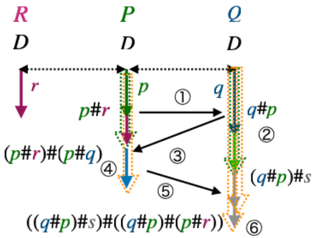
Eventual Consistency of Collaborative Replication

- CCR assures that every update on the local replica of a site is sent to other sites to produce the new confluent operations in each site.
- Every time updates occur in the system, they continue the replication process of making a new confluent state.
- When no updates are in the system, the replicated data in the sites is the same across the system. Thus, the confluent replication leads to the eventual consistency.

When does the replication process terminate?

- If R has not operated r in step ④, P makes $p\#_D(q \cdot p_q)$ or $p\#_D(q\#_D p)$ which is reduced to $p\#_D q$.
- Moreover, if S has neither done s in ⑥ then Q computes $T_D(q \cdot p_q, p \cdot q_p) = (!, !)$.
- Thus, the process should terminate when OT gives the identity op, even over the circular connection.



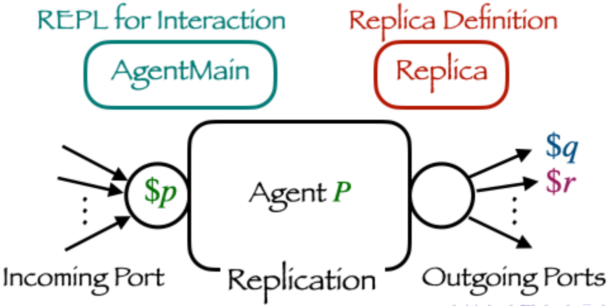


Incremental Messaging in CCR

- Up to now, we have not been concerned about the problem of efficiency about the size of messages.
- It costs too much if updated operations of each site are always sent from the beginning of replication.
- We can reduce the message size between connected sites by keeping the indices of the last message of the partner site.
- Note that the indices differ partner-wise of the connection. It is an easy way of reducing the cost of messaging.

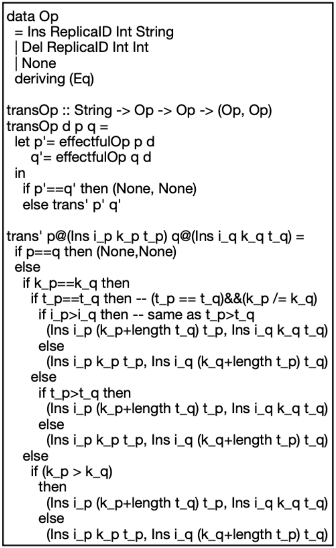
Coordination-free Replication in Practice

- The CCR Agent consists of three modules, *Agent*, *AgentMain*, and *Replica*.
  - *Agent* does Collaborative Replication for the replica defined in *Replica* with other Agents.
  - *AgentMain* provides a REPL for the user.
- The *Replica* module defines the type of the replica data, the operations and the transformation.

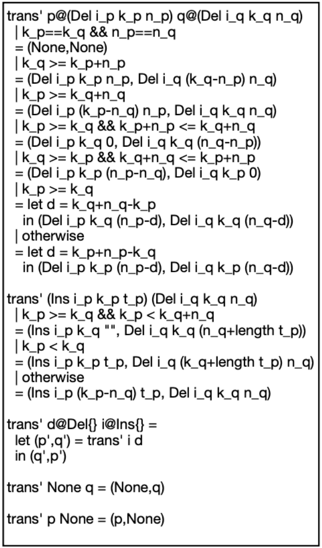


Realtime Collaborative Editing

- OT-based data sharing for server-less collaborative replication.



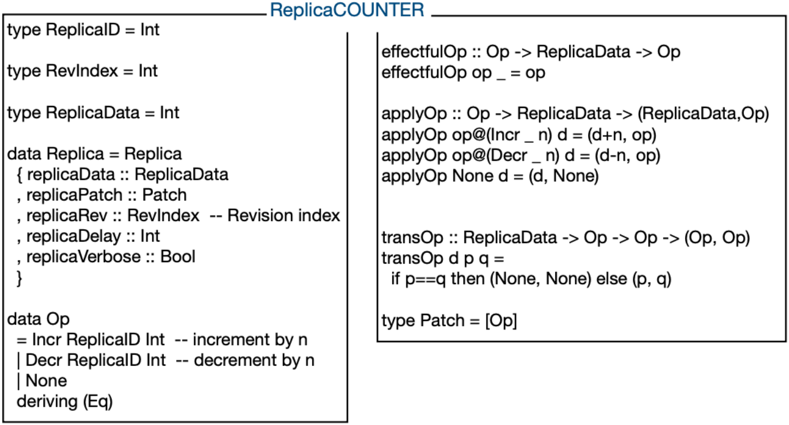
```
data Op
  = Ins ReplicaID Int String
  | Del ReplicaID Int Int
  | None
  deriving (Eq)

transOp :: String -> Op -> Op -> (Op, Op)
transOp d p q =
  let p'= effectfulOp p d
      q'= effectfulOp q d
  in
    if p'==q' then (None, None)
    else trans' p' q'

trans' p@(Ins i_p k_p t_p) q@(Ins i_q k_q t_q) =
  if p==q then (None,None)
  else
    if k_p==k_q then
      if t_p==t_q then -- (t_p == t_q)&&(k_p /= k_q)
        if i_p>i_q then -- same as t_p>t_q
          (Ins i_p (k_p+length t_q) t_p, Ins i_q k_q t_q)
        else
          (Ins i_p k_p t_p, Ins i_q (k_q+length t_p) t_q)
      else
        if t_p>t_q then
          (Ins i_p (k_p+length t_q) t_p, Ins i_q k_q t_q)
        else
          (Ins i_p k_p t_p, Ins i_q (k_q+length t_p) t_q)
    else
      if (k_p > k_q)
        then
          (Ins i_p (k_p+length t_q) t_p, Ins i_q k_q t_q)
        else
          (Ins i_p k_p t_p, Ins i_q (k_q+length t_p) t_q)
```



```
trans' p@(Del i_p k_p n_p) q@(Del i_q k_q n_q)
  | k_p==k_q && n_p==n_q
  = (None,None)
  | k_q >= k_p+n_p
  = (Del i_p k_p n_p, Del i_q (k_q-n_p) n_q)
  | k_p >= k_q+n_q
  = (Del i_p (k_p-n_q) n_p, Del i_q k_q n_q)
  | k_p >= k_q && k_p+n_p <= k_q+n_q
  = (Del i_p k_q 0, Del i_q k_q (n_q-n_p))
  | k_q >= k_p && k_q+n_q <= k_p+n_p
  = (Del i_p k_p (n_p-n_q), Del i_q k_p 0)
  | k_p >= k_q
  = let d = k_q+n_q-k_p
    in (Del i_p k_q (n_p-d), Del i_q k_q (n_q-d))
  | otherwise
  = let d = k_p+n_p-k_q
    in (Del i_p k_p (n_p-d), Del i_q k_p (n_q-d))

trans' (Ins i_p k_p t_p) (Del i_q k_q n_q)
  | k_p >= k_q && k_p < k_q+n_q
  = (Ins i_p k_q "", Del i_q k_q (n_q+length t_p))
  | k_p < k_q
  = (Ins i_p k_p t_p, Del i_q (k_q+length t_p) n_q)
  | otherwise
  = (Ins i_p (k_p-n_q) t_p, Del i_q k_q n_q)

trans' d@Del{} i@Ins{} =
  let (p',q') = trans' i d
  in (q',p')

trans' None q = (None,q)

trans' p None = (p,None)
```

Basic Replicated Data: Counter

- Operations are (Incr $\$i\ n$) for increment $D :: \mathit{Int}$ by n and (Decr $\$i\ x$) for decrement D by n.
- The parameter $\$i$ is the ID ($= 0, 1, 2, \cdots$) of the site in which the operation is issued.



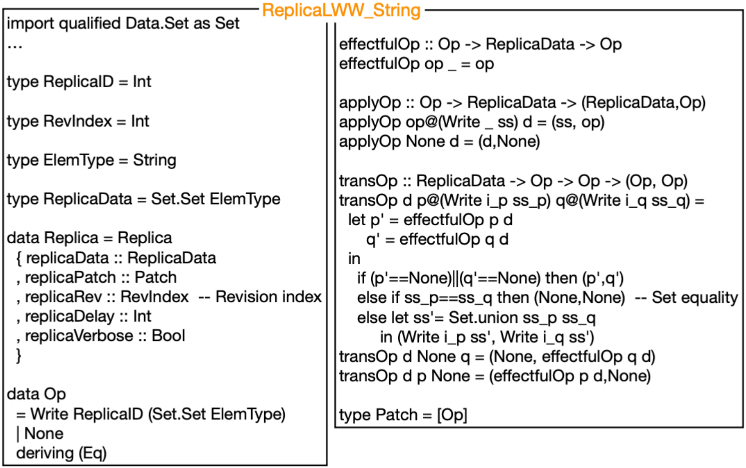
ReplicaCOUNTER

```
type ReplicaID = Int

type RevIndex = Int

type ReplicaData = Int

data Replica = Replica
  { replicaData :: ReplicaData
  , replicaPatch :: Patch
  , replicaRev :: RevIndex  -- Revision index
  , replicaDelay :: Int
  , replicaVerbose :: Bool
  }

data Op
  = Incr ReplicaID Int  -- increment by n
  | Decr ReplicaID Int  -- decrement by n
  | None
  deriving (Eq)
```

```
effectfulOp :: Op -> ReplicaData -> Op
effectfulOp op _ = op

applyOp :: Op -> ReplicaData -> (ReplicaData,Op)
applyOp op@(Incr _ n) d = (d+n, op)
applyOp op@(Decr _ n) d = (d-n, op)
applyOp None d = (d, None)


transOp :: ReplicaData -> Op -> Op -> (Op, Op)
transOp d p q =
  if p==q then (None, None) else (p, q)

type Patch = [Op]
```

Basic Replicated Data: Last-Write-Win Register

- LWW Register maintains the causality of events by concurrent updates, kept as a set of candidates of the winner.
- Operations are (Write $\$i\ x$) for replacing D by string x.



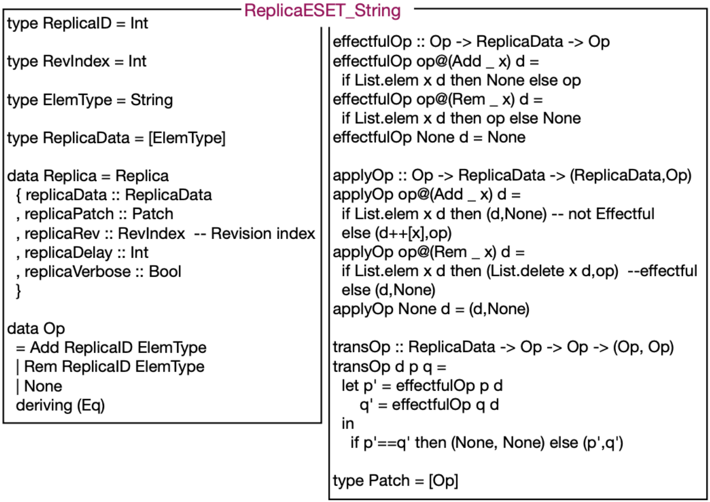
ReplicaLWW_String

```
import qualified Data.Set as Set
...

type ReplicaID = Int

type RevIndex = Int

type ElemType = String

type ReplicaData = Set.Set ElemType

data Replica = Replica
  { replicaData :: ReplicaData
  , replicaPatch :: Patch
  , replicaRev :: RevIndex  -- Revision index
  , replicaDelay :: Int
  , replicaVerbose :: Bool
  }

data Op
  = Write ReplicaID (Set.Set ElemType)
  | None
  deriving (Eq)
```

```
effectfulOp :: Op -> ReplicaData -> Op
effectfulOp op _ = op

applyOp :: Op -> ReplicaData -> (ReplicaData,Op)
applyOp op@(Write _ ss) d = (ss, op)
applyOp None d = (d,None)

transOp :: ReplicaData -> Op -> Op -> (Op, Op)
transOp d p@(Write i_p ss_p) q@(Write i_q ss_q) =
  let p' = effectfulOp p d
      q' = effectfulOp q d
  in
    if (p'==None)||(q'==None) then (p',q')
    else if ss_p==ss_q then (None,None)  -- Set equality
    else let ss'= Set.union ss_p ss_q
         in (Write i_p ss', Write i_q ss')
transOp d None q = (None, effectfulOp q d)
transOp d p None = (effectfulOp p d,None)

type Patch = [Op]
```

Basic Replicated Data: ESET - Set with Effectful Operations

- Operations are (Add $\$i\ x$) for adding x into D if $x \notin D$ and (Rem $\$i\ x$) for removing x from D if $x \in D$.



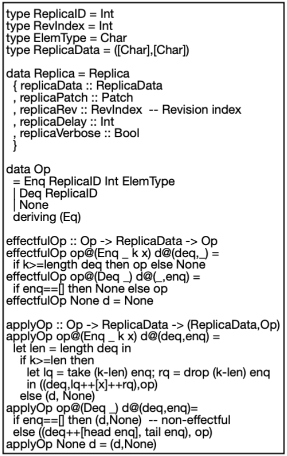
ReplicaESET_String

```
type ReplicaID = Int

type RevIndex = Int

type ElemType = String

type ReplicaData = [ElemType]

data Replica = Replica
  { replicaData :: ReplicaData
  , replicaPatch :: Patch
  , replicaRev :: RevIndex  -- Revision index
  , replicaDelay :: Int
  , replicaVerbose :: Bool
  }

data Op
  = Add ReplicaID ElemType
  | Rem ReplicaID ElemType
  | None
  deriving (Eq)
```

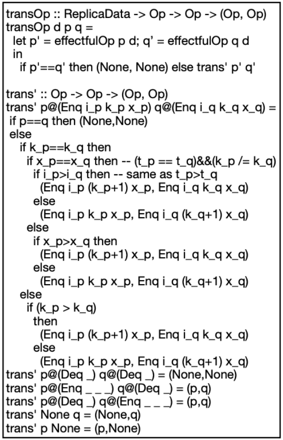
```
effectfulOp :: Op -> ReplicaData -> Op
effectfulOp op@(Add _ x) d =
  if List.elem x d then None else op
effectfulOp op@(Rem _ x) d =
  if List.elem x d then op else None
effectfulOp None d = None

applyOp :: Op -> ReplicaData -> (ReplicaData,Op)
applyOp op@(Add _ x) d =
  if List.elem x d then (d,None) -- not Effectful
  else (d++[x],op)
applyOp op@(Rem _ x) d =
  if List.elem x d then (List.delete x d,op)  --effectful
  else (d,None)
applyOp None d = (d,None)

transOp :: ReplicaData -> Op -> Op -> (Op, Op)
transOp d p q =
  let p' = effectfulOp p d
      q' = effectfulOp q d
  in
    if p'==q' then (None, None) else (p',q')

type Patch = [Op]
```

Basic Replicated Data: Mergeable Replicated Queue

- Operations are (Enq $\$i$ x) for enqueuing x into the queue D and (Deq $\$i$) for dequeuing x from D if it exists.
- $D = (deq, enq)$ for the enqued and dequeued elements.



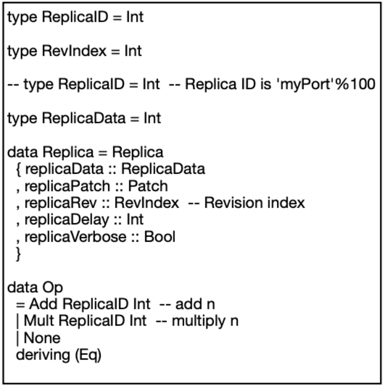
```
type ReplicaID = Int
type RevIndex = Int
type ElemType = Char
type ReplicaData = ([Char],[Char])

data Replica = Replica
  { replicaData :: ReplicaData
  , replicaPatch :: Patch
  , replicaRev :: RevIndex  -- Revision index
  , replicaDelay :: Int
  , replicaVerbose :: Bool
  }

data Op
  = Enq ReplicaID Int ElemType
  | Deq ReplicaID
  | None
  deriving (Eq)

effectfulOp :: Op -> ReplicaData -> Op
effectfulOp op@(Enq _ k x) d@(deq,_) =
  if k>=length deq then op else None
effectfulOp op@(Deq _) d@(_,enq) =
  if enq==[] then None else op
effectfulOp None d = None

applyOp :: Op -> ReplicaData -> (ReplicaData,Op)
applyOp op@(Enq _ k x) d@(deq,enq) =
  let len = length deq in
    if k>=len then
      let lq = take (k-len) enq; rq = drop (k-len) enq
      in ((deq,lq++[x]++rq),op)
    else (d, None)
applyOp op@(Deq _) d@(deq,enq)=
  if enq==[] then (d,None)  -- non-effectful
  else ((deq++[head enq], tail enq), op)
applyOp None d = (d,None)
```



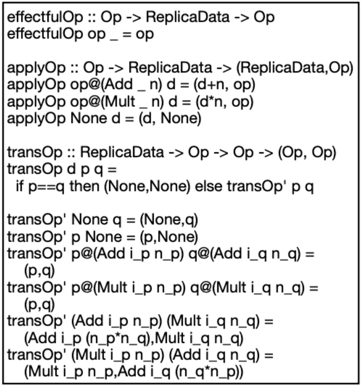
```
transOp :: ReplicaData -> Op -> Op -> (Op, Op)
transOp d p q =
  let p' = effectfulOp p d; q’ = effectfulOp q d
  in
    if p'==q' then (None, None) else trans' p' q'

trans' :: Op -> Op -> (Op, Op)
trans' p@(Enq i_p k_p x_p) q@(Enq i_q k_q x_q) =
 if p==q then (None,None)
 else
    if k_p==k_q then
      if x_p==x_q then -- (t_p == t_q)&&(k_p /= k_q)
        if i_p>i_q then -- same as t_p>t_q
          (Enq i_p (k_p+1) x_p, Enq i_q k_q x_q)
        else
          (Enq i_p k_p x_p, Enq i_q (k_q+1) x_q)
      else
        if x_p>x_q then
          (Enq i_p (k_p+1) x_p, Enq i_q k_q x_q)
        else
          (Enq i_p k_p x_p, Enq i_q (k_q+1) x_q)
    else
      if (k_p > k_q)
        then
          (Enq i_p (k_p+1) x_p, Enq i_q k_q x_q)
        else
          (Enq i_p k_p x_p, Enq i_q (k_q+1) x_q)
trans' p@(Deq _) q@(Deq _) = (None,None)
trans' p@(Enq _ _ _) q@(Deq _) = (p,q)
trans' p@(Deq _) q@(Enq _ _ _) = (p,q)
trans' None q = (None,q)
trans' p None = (p,None)
```

Basic Replicated Data: Integer register with add and mult operators

- M. Weidner proposes *Semidirect Product* for CRDT that makes concurrent updates by Add and Mult confluent.
- CCR transforms (Add m) and (Mult n) into (Add $m \times n$, Mult n) to make D be $(D + m) \times n = (D \times n) + (m \times n)$, which illustrates the semidirect product law using OT.



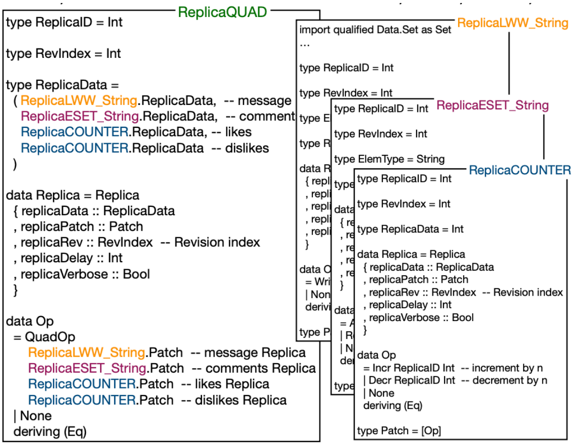
```
type ReplicaID = Int

type RevIndex = Int

-- type ReplicaID = Int  -- Replica ID is 'myPort'%100

type ReplicaData = Int

data Replica = Replica
  { replicaData :: ReplicaData
  , replicaPatch :: Patch
  , replicaRev :: RevIndex  -- Revision index
  , replicaDelay :: Int
  , replicaVerbose :: Bool
  }

data Op
  = Add ReplicaID Int  -- add n
  | Mult ReplicaID Int  -- multiply n
  | None
  deriving (Eq)
```



```
effectfulOp :: Op -> ReplicaData -> Op
effectfulOp op _ = op

applyOp :: Op -> ReplicaData -> (ReplicaData,Op)
applyOp op@(Add _ n) d = (d+n, op)
applyOp op@(Mult _ n) d = (d*n, op)
applyOp None d = (d, None)

transOp :: ReplicaData -> Op -> Op -> (Op, Op)
transOp d p q =
  if p==q then (None,None) else transOp' p q

transOp' None q = (None,q)
transOp' p None = (p,None)
transOp' p@(Add i_p n_p) q@(Add i_q n_q) =
    (p,q)
transOp' p@(Mult i_p n_p) q@(Mult i_q n_q) =
    (p,q)
transOp' (Add i_p n_p) (Mult i_q n_q) =
    (Add i_p (n_p*n_q),Mult i_q n_q)
transOp' (Mult i_p n_p) (Add i_q n_q) =
    (Mult i_p n_p,Add i_q (n_q*n_p))
```

Composite Replicated Data: Social Post Quadruples

- An *Algebraic Replicated Data Types* example of a *SocialMedia* replica for a local-first social media application to share the quadruples *SocialPost* of *messages*, *comments*, *likes*, and *dislikes*.



ReplicaQUAD

```
type ReplicaID = Int

type RevIndex = Int

type ReplicaData =
 ( ReplicaLWW_String.ReplicaData,  -- message
   ReplicaESET_String.ReplicaData,  -- comment
   ReplicaCOUNTER.ReplicaData, -- likes
   ReplicaCOUNTER.ReplicaData  -- dislikes
 )

data Replica = Replica
 { replicaData :: ReplicaData
 , replicaPatch :: Patch
 , replicaRev :: RevIndex  -- Revision index
 , replicaDelay :: Int
 , replicaVerbose :: Bool
 }

data Op
 = QuadOp
    ReplicaLWW_String.Patch  -- message Replica
    ReplicaESET_String.Patch  -- comments Replica
    ReplicaCOUNTER.Patch  -- likes Replica
    ReplicaCOUNTER.Patch  -- dislikes Replica
 | None
 deriving (Eq)
```

ReplicaLWW_String

```
import qualified Data.Set as Set
...

type ReplicaID = Int

type RevIndex = Int

type E

type R

data R
 { repl
 , repl
 , repl
 , repl
 , repl
 }

data O
 = Wri
 | Non
 derivi

type P
```

ReplicaESET_String

```
type ReplicaID = Int

type RevIndex = Int

type ElemType = String

type

data
 { re
 , re
 , re
 , re
 , re
 }

data
 = A
 | R
 | N
 der

type
```

ReplicaCOUNTER

```
type ReplicaID = Int

type RevIndex = Int

type ReplicaData = Int

data Replica = Replica
 { replicaData :: ReplicaData
 , replicaPatch :: Patch
 , replicaRev :: RevIndex  -- Revision index
 , replicaDelay :: Int
 , replicaVerbose :: Bool
 }

data Op
 = Incr ReplicaID Int  -- increment by n
 | Decr ReplicaID Int  -- decrement by n
 | None
 deriving (Eq)

type Patch = [Op]
```

ReplicaQUAD

```
data Op
  = QuadOp
      ReplicaLWW_String.Patch  -- message Replica
      ReplicaESET_String.Patch  -- comments Replica
      ReplicaCOUNTER.Patch  -- likes Replica
      ReplicaCOUNTER.Patch  -- dislikes Replica
  | None
  deriving (Eq)

transOp :: ReplicaData -> Op -> Op -> (Op, Op)
transOp (message,comments,likes,dislikes) p q =
  if p==q then (None, None) else
  if p==None then (None, q) else
  if q==None then (p, None) else
   let (QuadOp ms_p cs_p ls_p ds_p) = p
       (QuadOp ms_q cs_q ls_q ds_q) = q
       (ms_p',ms_q') =
         ReplicaLWW_String.transPatch message ms_p ms_q
       (cs_p',cs_q') =
         ReplicaESET_String.transPatch comments cs_p cs_q
       (ls_p',ls_q') =
         ReplicaCOUNTER.transPatch likes ls_p ls_q
       (ds_p',ds_q') =
         ReplicaCOUNTER.transPatch dislikes ds_p ds_q
   in (QuadOp ms_p' cs_p' ls_p' ds_p',
       QuadOp ms_q' cs_q' ls_q' ds_q')

type Patch = [Op]
```

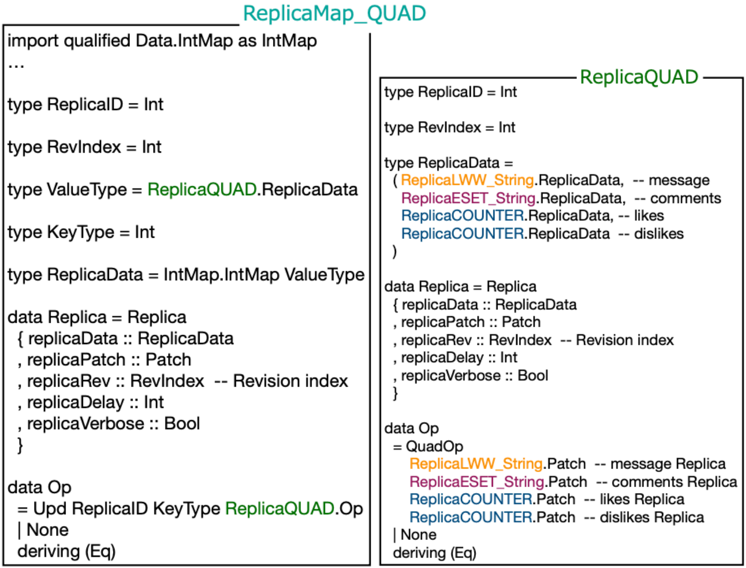
```
transPatch ::
  ReplicaData -> Patch -> Patch -> (Patch, Patch)
transPatch d pa pb = (pa'',pb'')
  where
  (pa',pb') = transPatch' d pa pb
  pa'' = filter (/=None) pa'
  pb'' = filter (/=None) pb'

transPatch' ::
  ReplicaData -> Patch -> Patch -> (Patch, Patch)
transPatch' d pa pb =
  snd (foldl step (d,([], pb)) pa)
  where
  step (d',(pa',pb')) a =
   let (a',pb'') = transOpPatch d' a pb'
       (d'',_) = applyOp a d'
   in (d'',(pa'++[a'],pb''))
transOpPatch ::
  ReplicaData -> Op -> Patch -> (Op, Patch)
transOpPatch d a pb =
  snd (foldl step (d,(a,[])) pb)
  where
  step (d',(a',pb')) b =
   let (a'',b') = transOp d' a' b
       (d'',_) = applyOp b d'
   in (d'',(a'',pb'++[b']))
```

Definition of transPatch is common to Replica modules, and transOp and applyOp refer to those in each module.

Figure: SocialPost Quadruples - Operations and Transformation

Composite Replicated Data: Map of Replica Data

- The *SocialMedia* map of the *SocialPost* quadruples:

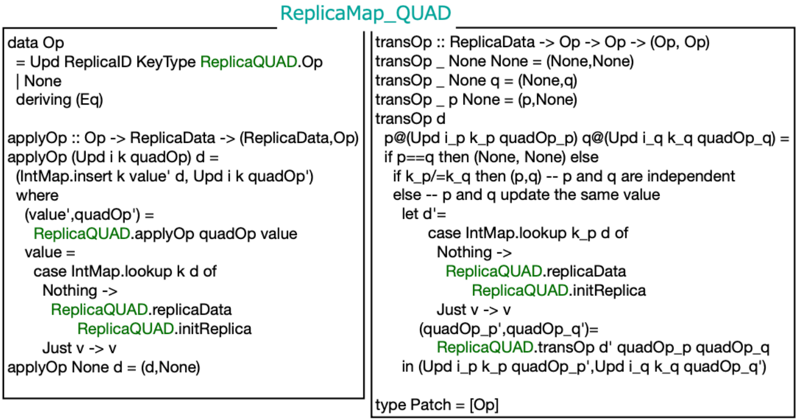


ReplicaMap_QUAD

```
import qualified Data.IntMap as IntMap
...

type ReplicaID = Int

type RevIndex = Int

type ValueType = ReplicaQUAD.ReplicaData

type KeyType = Int

type ReplicaData = IntMap.IntMap ValueType

data Replica = Replica
  { replicaData :: ReplicaData
  , replicaPatch :: Patch
  , replicaRev :: RevIndex  -- Revision index
  , replicaDelay :: Int
  , replicaVerbose :: Bool
  }

data Op
  = Upd ReplicaID KeyType ReplicaQUAD.Op
  | None
  deriving (Eq)
```

ReplicaQUAD

```
type ReplicaID = Int

type RevIndex = Int

type ReplicaData =
  ( ReplicaLWW_String.ReplicaData,  -- message
    ReplicaESET_String.ReplicaData,  -- comments
    ReplicaCOUNTER.ReplicaData, -- likes
    ReplicaCOUNTER.ReplicaData  -- dislikes
  )

data Replica = Replica
  { replicaData :: ReplicaData
  , replicaPatch :: Patch
  , replicaRev :: RevIndex  -- Revision index
  , replicaDelay :: Int
  , replicaVerbose :: Bool
  }

data Op
  = QuadOp
      ReplicaLWW_String.Patch  -- message Replica
      ReplicaESET_String.Patch  -- comments Replica
      ReplicaCOUNTER.Patch  -- likes Replica
      ReplicaCOUNTER.Patch  -- dislikes Replica
  | None
  deriving (Eq)
```



ReplicaMap_QUAD

```
data Op
  = Upd ReplicaID KeyType ReplicaQUAD.Op
  | None
  deriving (Eq)

applyOp :: Op -> ReplicaData -> (ReplicaData,Op)
applyOp (Upd i k quadOp) d =
  (IntMap.insert k value' d, Upd i k quadOp')
  where
    (value',quadOp') =
      ReplicaQUAD.applyOp quadOp value
    value =
      case IntMap.lookup k d of
        Nothing ->
          ReplicaQUAD.replicaData
                ReplicaQUAD.initReplica
        Just v -> v
applyOp None d = (d,None)
```

```
transOp :: ReplicaData -> Op -> Op -> (Op, Op)
transOp _ None None = (None,None)
transOp _ None q = (None,q)
transOp _ p None = (p,None)
transOp d
  p@(Upd i_p k_p quadOp_p) q@(Upd i_q k_q quadOp_q) =
  if p==q then (None, None) else
    if k_p/=k_q then (p,q) -- p and q are independent
    else -- p and q update the same value
      let d'=
            case IntMap.lookup k_p d of
              Nothing ->
                ReplicaQUAD.replicaData
                      ReplicaQUAD.initReplica
              Just v -> v
          (quadOp_p',quadOp_q')=
              ReplicaQUAD.transOp d' quadOp_p quadOp_q
      in (Upd i_p k_p quadOp_p',Upd i_q k_q quadOp_q')

type Patch = [Op]
```