Cloud Types for Eventual Consistency

Problem

Applications keep local replicas of shared data.

Accessibility is paramount

How do we program around eventually consistent replicas?

Solution

A layer of abstraction.

Specialized cloud data types.

Automatically shared and persisted.

Revision Consistency

Revision diagrams guarantee eventual consistency

Grocery List

Display, Buy (add), and Bought(remove)

Cloud Integer (CInt)

Cloud Array

Yield statement

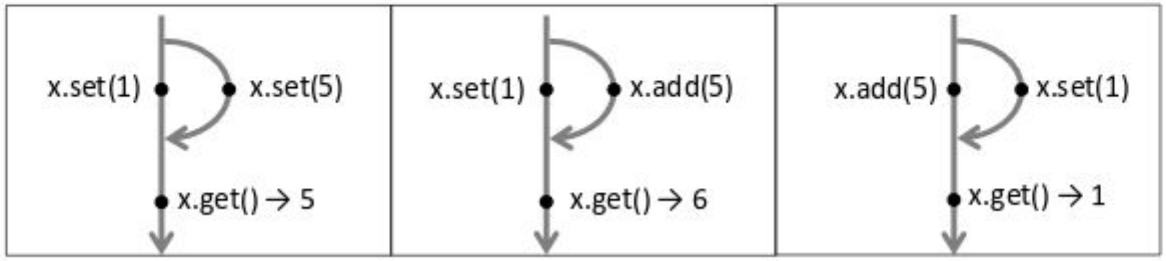
Gives permission to propagate changes, apply external changes

Revision Diagrams

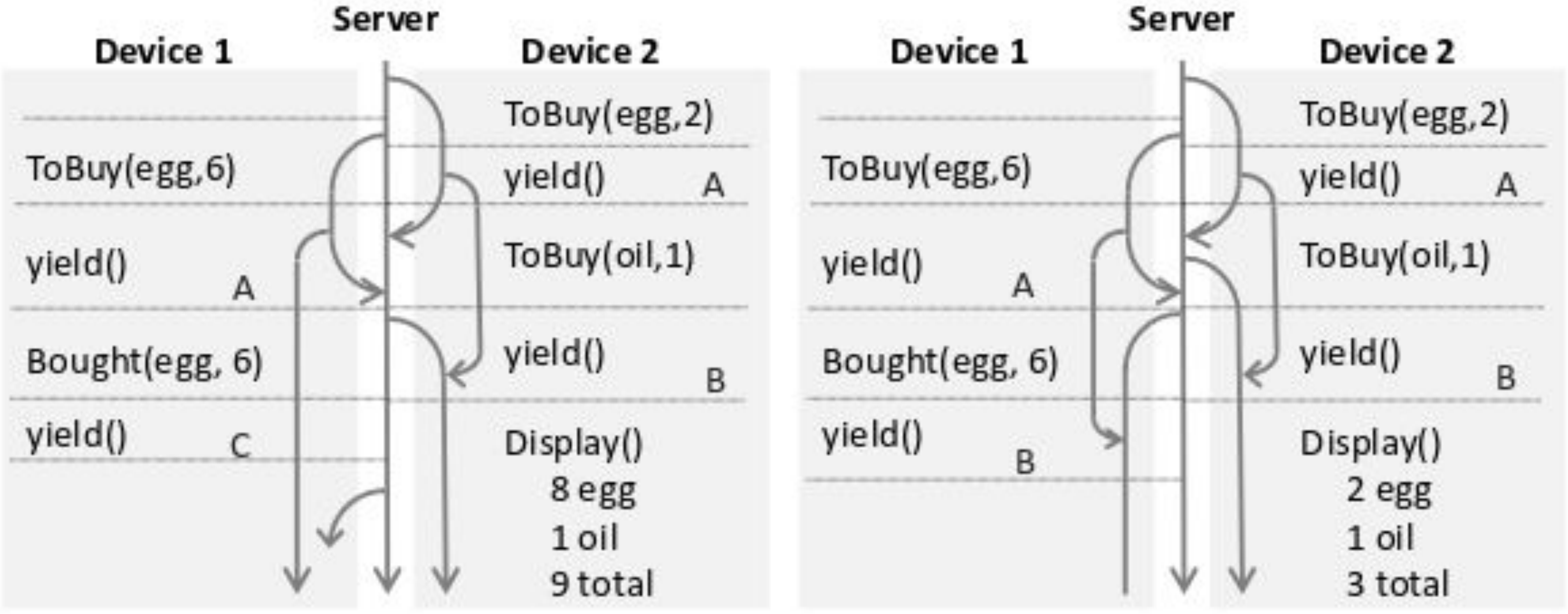
Similar to source control systems, show how versions are joined and forked.

Logging similar to a database

Cloud types can provide optimized operations and bounded logs.



Execution Model



Entities

Allows for dynamic creation or deletion of entries in an array.

Can be created without specifying an index.

Can be explicitly deleted.

Stronger Consistency

One must establish a server connection and wait for a response.

```
array Seat [
  row : int,
  letter : string ]
{
  assignedTo : CString;
}
```

```
function NaiveReserve(seat: Seat, customer : string)
{
  if (seat.assignedTo.get() == "")
    seat.assignedTo.set(customer);
  else
    print("reservation failed");
}
```

Flush and setIfEmpty()

```
seat.assignedTo.setIfEmpty(customer);
flush;
if (seat.assignedTo.get() ≠ customer) print("reservation failed");
```

Syntax

Index Types: Int, String, E, A (entity or array ID)

Cloud Types: CInt, CString, CSet

Schema: sequence of declarations

Properties: map an index to a cloud type

Operations do not return cloud types.

entity names	$Ent \ni$	E	$::= \ldots$
array names	$Arr \ni$	A	$::= \ldots$
index types		ι	$::= \mathsf{Int} \mid \mathsf{String} \mid E \mid A$
cloud types		ω	$::= \mathsf{CInt} \mid \mathsf{CString} \mid \mathsf{CSet}\langle\iota\rangle \mid \ldots$
expression types		τ	$::= \iota \mid \mathsf{Set}\langle\tau\rangle \mid \tau \rightarrow \tau \mid (\tau_1, \ldots, \tau_n)$
key names		k	$::= \ldots$
property names		p	$::= \ldots$
declarations		$decl$	$::= \mathsf{entity}\ E(k_1 : \iota_1, \ldots, k_n : \iota_n)$
			$\mid \mathsf{array}\ A[k_1 : \iota_1, \ldots, k_n : \iota_n]$
			$\mid \mathsf{property}\ p\ :\ \iota \rightarrow \omega$
schema		$\mathcal{S}$	$::= decl_1; \ldots; decl_n$
unique id's	$Uid \ni$	uid	$::= \ldots$ (abstract)
constants	$Con \ni$	c	$::= \ldots$ (integer and string literals)
updates		op_u	$::= \ldots$ (predefined)
queries		op_q	$::= \ldots$ (predefined)
operations		op	$::= op_u \mid op_q$
values	$Val \ni$	v	$::= A[v_1, \ldots, v_n] \mid E[uid, v_1, \ldots, v_n]$
			$\mid c \mid x \mid (v_1, \ldots, v_n) \mid \lambda(x : \tau). e$
expressions		e	$::= \mathsf{new}\ E(e_1, \ldots, e_n)$
			$\mid \mathsf{delete}\ e$
			$\mid A[e_1, \ldots, e_n]$
			$\mid e.p.op(e_1, .., e_n)$
			$\mid e.k$
			$\mid \mathsf{all}\ E$
			$\mid \mathsf{entries}\ p$
			$\mid \mathsf{yield} \mid \mathsf{flush}$
			$\mid v \mid e_1\, e_2 \mid e_1; e_2 \mid (e_1, \ldots, e_n)$
program		$program$	$::= \mathcal{S}; e$

Expressions

$$\frac{\textsf{entity } E(k_1 : \iota_1, \ldots, k_n : \iota_n) \in \mathcal{S} \quad \mathcal{S}, \Gamma \vdash e_i : \iota_i}{\mathcal{S}, \Gamma \vdash \textsf{new } E(e_1, \ldots, e_n) : E}$$

$$\frac{\mathcal{S}, \Gamma \vdash e : E}{\mathcal{S}, \Gamma \vdash \textsf{delete } e : \textsf{Unit}}$$

$$\frac{\textsf{array } A[k_1 : \iota_1, \ldots, k_n : \iota_n] \in \mathcal{S} \quad \mathcal{S}, \Gamma \vdash e_i : \iota_i}{\mathcal{S}, \Gamma \vdash A[e_1, \ldots, e_n] : A}$$

$$\frac{\textsf{entity } E(\ldots) \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash E[uid, v_1, \ldots, v_n] : E}$$

$$\frac{\mathcal{S}, \Gamma \vdash e : \iota \quad \textsf{property } p : \iota \rightarrow \omega \in \mathcal{S} \quad \omega.op : (\tau_1, \ldots, \tau_n) \rightarrow \tau \in \Gamma \quad \mathcal{S}, \Gamma \vdash e_i : \tau_i}{\mathcal{S}, \Gamma \vdash e.p.op(e_1, \ldots, e_n) : \tau}$$

$$\frac{\textsf{entity } E(\ldots) \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash \textsf{all } E : \textsf{Set}\langle E \rangle}$$

$$\frac{\mathcal{S}, \Gamma \vdash e : E \quad \textsf{entity } E(\ldots, k : \iota, \ldots) \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash e.k : \iota}$$

$$\frac{\textsf{property } p : \iota \rightarrow \omega \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash \textsf{entries } p : \textsf{Set}\langle \iota \rangle}$$

$$\frac{\mathcal{S}, \Gamma \vdash e : A \quad \textsf{array } A[\ldots, k : \iota, \ldots] \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash e.k : \iota}$$

$$\frac{x : \tau \in \Gamma}{\mathcal{S}, \Gamma \vdash x : \tau}$$

$$\frac{\mathcal{S}, (\Gamma, x : \tau_1) \vdash e : \tau_2}{\mathcal{S}, \Gamma \vdash \lambda(x : \tau_1).\, e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\mathcal{S}, \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \mathcal{S}, \Gamma \vdash e_2 : \tau_2}{\mathcal{S}, \Gamma \vdash e_1\, e_2 : \tau}$$

$$\frac{\mathcal{S}, \Gamma \vdash e_i : \tau_i}{\mathcal{S}, \Gamma \vdash (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n)}$$

$$\frac{\mathcal{S}, \Gamma \vdash e_1 : \tau_1 \quad \mathcal{S}, \Gamma \vdash e_2 : \tau_2}{\mathcal{S}, \Gamma \vdash e_1 ; e_2 : \tau_2}$$

$$\frac{}{\mathcal{S}, \Gamma \vdash \textsf{yield} : \textsf{Unit}}$$

$$\frac{}{\mathcal{S}, \Gamma \vdash \textsf{flush} : \textsf{Unit}}$$

Semantics

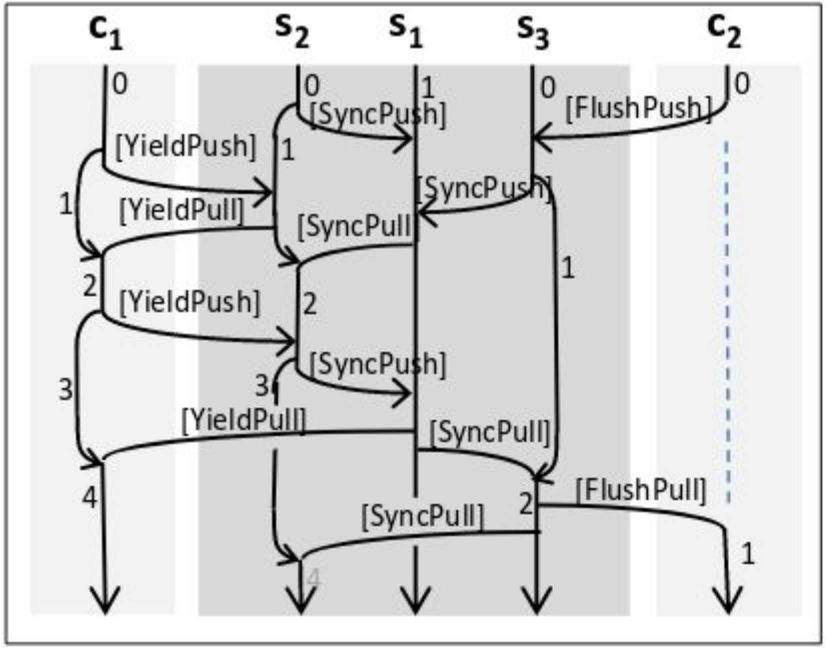
Evaluation context is an abstraction of a program counter.

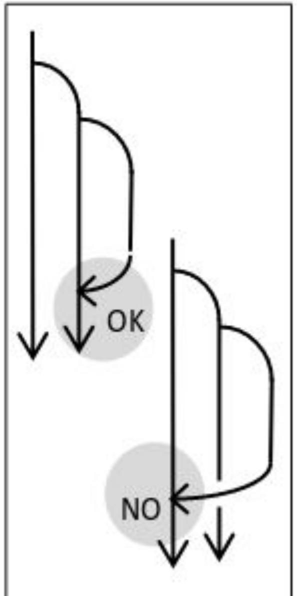
Yield, Flush, and Barrier (block) can't be described as local operations.

$$
\begin{array}{rl}
\mathcal{E} ::= & \square \\
| & \textsf{new}\ E(v_1, ..., v_i, \mathcal{E}, e_j, ..., e_n) \\
| & \textsf{delete}\ \mathcal{E} \\
| & A[v_1, ..., v_i, \mathcal{E}, e_j, ..., e_n] \\
| & \mathcal{E}.p.op(e_1, ..., e_n) \\
| & v.p.op(v_1, ..., v_i, \mathcal{E}, e_j, ..., e_n) \\
| & \mathcal{E}.k \\
| & \mathcal{E}\, e \mid v\, \mathcal{E} \mid \mathcal{E}; e \\
| & (v_1, ..., v_i, \mathcal{E}, e_j, ..., e_n)
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{E}[\![\textsf{new}\ E(v_1, ..., v_n)]\!]; \sigma & \rightarrow & \mathcal{E}[\![E[uid, v_1, ..., v_n]]\!]; \sigma.\textsf{create}_E(E[uid, v_1, ..., v_n]) \quad (\text{fresh } uid) \\
\mathcal{E}[\![\textsf{delete}\ E[uid, ...]]\!]; \sigma & \rightarrow & \mathcal{E}[\![()]\!]; \sigma.\textsf{delete}_E(uid) \\
\mathcal{E}[\![v.p.op_u(v_1, ..., v_n)]\!]; \sigma & \rightarrow & \mathcal{E}[\![()]\!]; \sigma.\textsf{update}_p(v, op_u(v_1, ..., v_n)) \\
 & & \\
\mathcal{E}[\![v.p.op_q(v_1, ..., v_n)]\!]; \sigma & \rightarrow & \mathcal{E}[\![\sigma.\textsf{query}_p(v, op_q(v_1, ..., v_n))]\!]; \sigma \\
\mathcal{E}[\![\textsf{all}\ E]\!]; \sigma & \rightarrow & \mathcal{E}[\![\sigma.\textsf{all}_E]\!]; \sigma \\
\mathcal{E}[\![\textsf{entries}\ p]\!]; \sigma & \rightarrow & \mathcal{E}[\![\sigma.\textsf{entries}_p]\!]; \sigma \\
 & & \\
\mathcal{E}[\![A[v_1, ..., v_n].k_i]\!]; \sigma & \rightarrow & \mathcal{E}[\![v_i]\!]; \sigma \\
\mathcal{E}[\![E[uid, v_1, ..., v_n].k_i]\!]; \sigma & \rightarrow & \mathcal{E}[\![v_i]\!]; \sigma \\
\mathcal{E}[\![(\lambda(x:\tau).e)\, v]\!]; \sigma & \rightarrow & \mathcal{E}[\![e[v/x]]\!]; \sigma \\
\mathcal{E}[\![v; e]\!]; \sigma & \rightarrow & \mathcal{E}[\![e]\!]; \sigma
\end{array}
$$

Model and Distribution





Join rule

A more sensical example

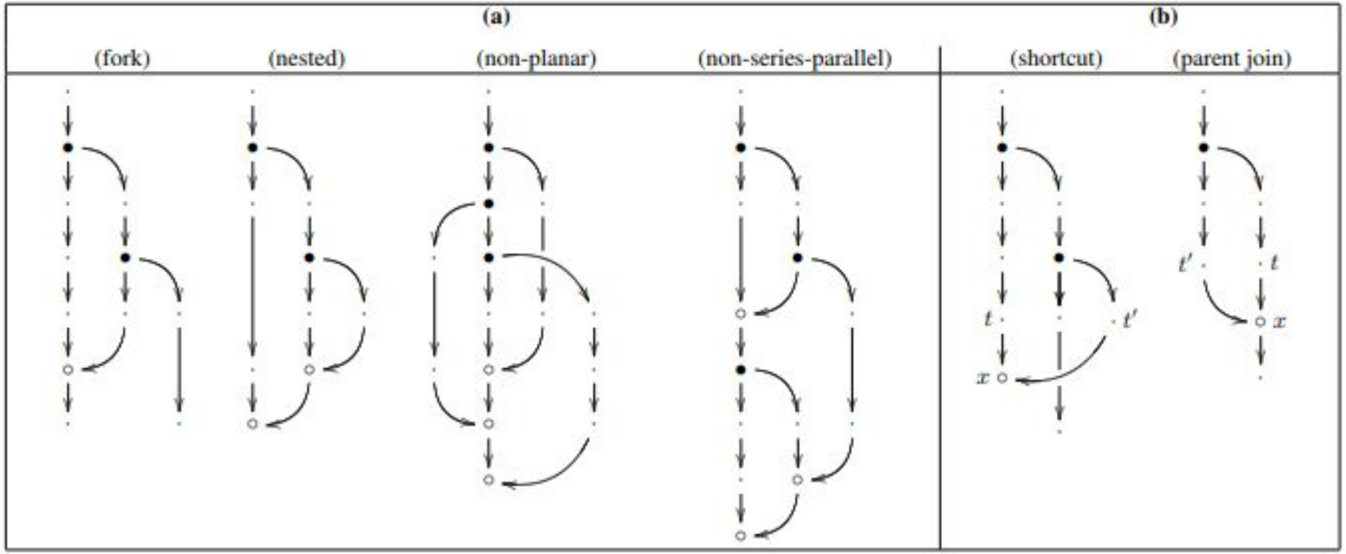


Figure 1. **(a)** Four examples of revision diagrams. **(b)** Two diagrams that are not revision diagrams since they violate the join property at the creation of the join node x. In the rightmost diagram, $F(t')$ is undefined on the main revision and therefore $F(t') \rightarrow^* t$ does not hold.

Client Evaluation Rules

Spawn: creates a client.

Yield-Push: Sends revision.

Yield-Pull: Receives revision.

Yield-NOP: disconnected clients can keep executing.

$$\text{[EVAL]}\ \frac{e;\sigma \to e';\sigma'}{\mathcal{C}(c \mapsto (r, e, \sigma)) \Rightarrow \mathcal{C}[c \mapsto (r, e', \sigma')]}$$

$$\text{[SPAWN]}\ \frac{c \notin \mathsf{dom}(\mathcal{C})}{\mathcal{C} \Rightarrow \mathcal{C}[c \mapsto (0, e, \sigma_0)]}$$

[YIELD-PUSH]

$$\frac{R(c) = r \quad R' = R[c \mapsto r+1] \quad \mathit{fork}(\sigma_c) = (\sigma'_c, \sigma''_c) \quad \mathit{join}(\sigma_s, \sigma'_c) = \sigma'_s}{\mathcal{C}(s \mapsto (r_s, R, \sigma_s),\ c \mapsto (r, \mathcal{E}[\![\mathsf{yield}]\!], \sigma_c)) \Rightarrow \mathcal{C}[s \mapsto (r_s, R', \sigma'_s),\ c \mapsto (r+1, \mathcal{E}[\![()]\!], \sigma''_c)]}$$

[YIELD-PULL]

$$\frac{R(c) = r \quad R' = R[c \mapsto r+1] \quad \mathit{fork}(\sigma_s) = (\sigma'_s, \sigma''_s) \quad \mathit{join}(\sigma''_s, \sigma_c) = \sigma'_c}{\mathcal{C}(s \mapsto (r_s, R, \sigma_s),\ c \mapsto (r, \mathcal{E}[\![\mathsf{yield}]\!], \sigma_c)) \Rightarrow \mathcal{C}[s \mapsto (r_s, R', \sigma'_s),\ c \mapsto (r+1, \mathcal{E}[\![()]\!], \sigma'_c)]}$$

$$\text{[YIELD-NOP]}\ \frac{}{\mathcal{C}(c \mapsto (r, \mathcal{E}[\![\mathsf{yield}]\!], \sigma)) \Rightarrow \mathcal{C}[c \mapsto (r, \mathcal{E}[\![()]\!], \sigma)]}$$

Server Evaluation Rules

Sync: Similar to Yield but the round maps are joined.

Servers can be spawned and retired much like clients.

$$[\textsc{Create}]\ \frac{s \notin \mathsf{dom}(\mathcal{C})}{\mathcal{C} \ \Rightarrow\ \mathcal{C}[s \mapsto (0, R_0, \sigma_0)]}$$

$$[\textsc{Sync-Push}]\ \frac{R_s(t) = r_t \qquad R'_s = \mathsf{max}(R_s, R_t) \qquad R''_s = R'_s[t \mapsto r_t + 1] \qquad \mathit{fork}(\sigma_t) = (\sigma'_t, \sigma''_t) \qquad \mathit{join}(\sigma_s, \sigma'_t) = \sigma'_s}{\mathcal{C}(s \mapsto (r_s, R_s, \sigma_s),\, t \mapsto (r_t, R_t, \sigma_t)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R''_s, \sigma'_s),\, t \mapsto (r_t + 1, R_t, \sigma''_t)]}$$

$$[\textsc{Sync-Pull}]\ \frac{R_s(t) = r_t \qquad R'_t = \mathsf{max}(R_s, R_t) \qquad R'_s = R_s[t \mapsto r_t + 1] \qquad \mathit{fork}(\sigma_s) = (\sigma'_s, \sigma''_s) \qquad \mathit{join}(\sigma''_s, \sigma_t) = \sigma'_t}{\mathcal{C}(s \mapsto (r_s, R_s, \sigma_s),\, t \mapsto (r_t, R_t, \sigma_t)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R'_s, \sigma'_s),\, t \mapsto (r_t + 1, R'_t, \sigma'_t)]}$$

$$[\textsc{Retire}]\ \frac{R_s(t) = r_t \qquad R'_s = \mathsf{max}(R_s, R_t) \qquad \mathit{join}(\sigma_s, \sigma_t) = \sigma'_s}{\mathcal{C}(s \mapsto (r_s, R_s, \sigma_s),\, t \mapsto (r_t, R_t, \sigma_t)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R'_s, \sigma'_s),\, t \mapsto \bot]}$$

Flush operation

Ensures that all client updates are pushed to the server.

Client must be able to observe that the updates have arrived.

Flush Push-> Sync -> Commit -> Flush Pull

$$[\textsc{Flush-Push}]\quad \frac{R(c) = r \qquad R' = R[c \mapsto r+1] \qquad join(\sigma_s, \sigma_c) = \sigma'_s}{\mathcal{C}(s \mapsto (r_s, R, \sigma_s),\ c \mapsto (r, \mathcal{E}[\![\mathsf{flush}]\!], \sigma_c)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R', \sigma'_s),\ c \mapsto (r+1, \mathcal{E}[\![\mathsf{block}]\!], \sigma_c)]}$$

$$[\textsc{Flush-Pull}]\quad \frac{R(c^{\mathsf{flush}}) = r \qquad fork(\sigma_s) = (\sigma'_s, \sigma'_c)}{\mathcal{C}(s \mapsto (r_s, R, \sigma_s),\ c \mapsto (r, \mathcal{E}[\![\mathsf{block}]\!], \sigma_c)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R, \sigma'_s),\ c \mapsto (r, \mathcal{E}[\![()]\!], \sigma'_c)]}$$

$$[\textsc{Commit}]\quad \frac{R' = R[\forall c.\ c^{\mathsf{flush}} \mapsto R(c))}{\mathcal{C}(s_{\mathsf{main}} \mapsto (0, R, \sigma)) \ \Rightarrow\ \mathcal{C}[s_{\mathsf{main}} \mapsto (0, R', \sigma)]}$$

Fork-Join Automaton

FJA is defined as the tuple (Q,U,Σ, σ, f, j)

Must track and apply updates when revisions fork and join.

CInt state stores three values a boolean, a base, and an offset

$$
\begin{aligned}
Q^{\mathsf{CInt}} &: \{\mathsf{get}\} \\
U^{\mathsf{CInt}} &: \{\mathsf{set}(n) \mid n \in \mathrm{int}\} \cup \{\mathsf{add}(n) \mid n \in \mathrm{int}\} \\
\Sigma^{\mathsf{CInt}} &: \mathrm{bool} \times \mathrm{int} \times \mathrm{int} \\
\sigma_0^{\mathsf{CInt}} &: (\mathrm{false}, 0, 0) \\
\mathsf{add}(n)^{\#}\ & (r, b, d) = (r, b, d + n) \\
\mathsf{set}(n)^{\#}\ & (r, b, d) = (\mathrm{true}, n, 0) \\
\mathsf{get}^{\#}\ & (r, b, d) = b + d \\
f^{\mathsf{CInt}}\ & (r, b, d) = (r, b, d), (\mathrm{false}, b + d, 0) \\
j^{\mathsf{CInt}}\ & (r_1, b_1, d_1)(r_2, b_2, d_2) = \begin{cases} (\mathrm{true}, b_2, d_2) & \text{if } r_2 = \mathrm{true} \\ (r_1, b_1, d_1 + d_2) & \text{otherwise} \end{cases}
\end{aligned}
$$

CInt example

On fork boolean is reset, base value = current, offset = 0.

Add operations change the offset.

Set changes the boolean, sets base value, resets the offset.

Join assumes the base value or add the offset.

Fork-Join Automaton (CString)

Similar to the rules for CInts but the additional setIfEmpty function.

Conditional Writes, only works if the current value is empty.

$$
\begin{aligned}
Q^{\text{CString}} &: \{\mathsf{get}\}\\
U^{\text{CString}} &: \{\mathsf{set}(s) \mid s \in \text{string}\} \cup \{\mathsf{setIfEmpty}(s) \mid s \in \text{string} \setminus \{""\}\}\\
\Sigma^{\text{CString}} &: \{\bot, \mathsf{wr}, \mathsf{cond}(\text{string})\} \times \text{string}\\
\sigma_0^{\text{CString}} &: (\bot, "")\\
\mathsf{set}(s)^{\#}\ (r,t) &= (\mathsf{wr}, s)\\
\mathsf{setIfEmpty}(s)^{\#}\ (r,t) &= \begin{cases} (\mathsf{wr}, s) & \text{if } r = \mathsf{wr} \wedge t = "" \\ (\mathsf{cond}(s), s) & \text{if } r = \bot \wedge t = "" \\ (\mathsf{cond}(s), t) & \text{if } r = \bot \wedge t \neq "" \\ (r,t) & \text{otherwise} \end{cases}\\
\mathsf{get}^{\#}\ (r,s) &= s\\
f^{\text{CString}}\ (r,s) &= (r,s), (\bot, s)\\
j^{\text{CString}}\ (r_1,s_1)(r_2,s_2) &= \begin{cases} (\mathsf{wr}, s_2) & \text{if } r_2 = \mathsf{wr} \\ (\mathsf{wr}, s) & \text{if } r_1 = \mathsf{wr} \wedge s_1 = "" \wedge r_2 = \mathsf{cond}(s) \\ (\mathsf{cond}(s), s) & \text{if } r_1 = \bot \wedge s_1 = "" \wedge r_2 = \mathsf{cond}(s) \\ (\mathsf{cond}(s), s_1) & \text{if } r_1 = \bot \wedge s_1 \neq "" \wedge r_2 = \mathsf{cond}(s) \\ (r_1, s_1) & \text{otherwise} \end{cases}
\end{aligned}
$$

Complete State FJA’s

Assume a schema S with a state space with separate components for each entity type and property.

Each declaration property stores a total function of keys to values.

Each declaration entity stores a total function from entities to a state.

Complete State FJA operations are commutative.

operation	argument types	return type	entity/property definition
all_E		$\mathsf{Set}\langle E \rangle$	entity $E(k_1 : \iota_1, ..., k_n : \iota_n)$
$\mathsf{create}_E(e)$	E		entity $E(k_1 : \iota_1, ..., k_n : \iota_n)$
$\mathsf{delete}_E(e)$	E		entity $E(k_1 : \iota_1, ..., k_n : \iota_n)$
$\mathsf{entries}_p$		$\mathsf{Set}\langle \iota \rangle$	property $p : \iota \to \omega$
$\mathsf{query}_p(i, q)$	ι, Q^ω	Val	property $p : \iota \to \omega$
$\mathsf{update}_p(i, u)$	ι, U^ω		property $p : \iota \to \omega$

Complete State FJAs

Create: adds an element to an entity,

Delete: Maps an element to $\top$, also deletes dependent entities.

Entries: Returns entries of a property p that map to non-default FJAs

Forking: Point wise forking of all FJA’s for each property.

Joining: Point wise on all properties. For entities compute the maximum order of $\bot < ok < \top$. Also repropagate deletions.

CSets

CSets are built from entities.

Deletion applies to all sets containing an entity

Remove applies to only instances visible before the remove.

x.add(i) $\equiv$ { new $E_p(x, i)$; }
x.contains(i) $\equiv$ { return (all E_p where index == x and element == i).isNotEmpty() ; }
x.remove(i) $\equiv$ { foreach (e in all E_p where index == x and element == i) e.delete() ; }