Relational Reasoning for Mergeable Replicated Data Types

KC Sivaramakrishnan
joint work with
Gowtham Kaki, Swarn Priya, Suresh Jagannathan
• Weak Consistency & Isolation

• Serializability
• Linearizability
When *system-level concerns* like replication & availability affect *application-level design* decisions, programming becomes *complicated*. 
Seems like Twitter wants me to follow this guy.

Who to follow

Doug Woos  
@dougwoos  
PhD student, joining @BrownCSDept as a lecturer in Fall 2019. Into programming languages, distributed systems, baseball, and other stuff. he/him

Doug Woos  
@dougwoos  
PhD student, joining @BrownCSDept as a lecturer in Fall 2019. Into programming languages, distributed systems, baseball, and other stuff. he/him

Doug Woos  
@dougwoos  
PhD student, joining @BrownCSDept as a lecturer in Fall 2019. Into programming languages, distributed systems, baseball, and other stuff. he/him
Sequential Counter

module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
end
Sequential Counter

module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
end

- Written in idiomatic style
- Composable

  type counter_list = Counter.t list
Replicated Counter
Replicated Counter
Replicated Counter
Replicated Counter
Replicated Counter
Replicated Counter

+2

INTERNET

+3

+5

+5

+5

+5

+5
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
module Counter : sig
    type t
    val read : t -> int
    val add : t -> int -> t
    val sub : t -> int -> t
    val mult : t -> int -> t
end = struct
    type t = int
    let read x = x
    let add x d = x + d
    let sub x d = x - d
    let mult x n = x * n
end
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end

Diverges
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end

Addition and multiplication do not commute
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end
• Capture the effect of multiplication through the commutative addition operation
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end

• Capture the effect of multiplication through the commutative addition operation
module Counter : sig
    type t
    val read : t -> int
    val add : t -> int -> t
    val sub : t -> int -> t
    val mult : t -> int -> t
end = struct
    type t = int
    let read x = x
    let add x d = x + d
    let sub x d = x - d
    let mult x n = x * n
end

• Capture the effect of multiplication through the commutative addition operation
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end

• Capture the effect of multiplication through the commutative addition operation
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end

• Capture the effect of multiplication through the commutative addition operation
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end

• Capture the effect of multiplication through the commutative addition operation
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
end

- Capture the effect of multiplication through the commutative addition operation
Capture the effect of multiplication through the commutative addition operation

CRDTs
Conflict-free Replicated Data Types (CRDT)
Conflict-free Replicated Data Types (CRDT)

- CRDT is guaranteed to ensure *strong eventual consistency (SEC)*
  - G-counters, PN-counters, OR-Sets, Graphs, Ropes, docs, sheets
  - Simple interface for the clients of CRDTs
Conflict-free Replicated Data Types (CRDT)

• CRDT is guaranteed to ensure *strong eventual consistency (SEC)*
  ★ G-counters, PN-counters, OR-Sets, Graphs, Ropes, docs, sheets
  ★ Simple interface for the clients of CRDTs

• Need to reengineer every datatype to ensure SEC (commutativity)
  ★ Do not mirror sequential counter parts => implementation & proof burden
  ★ Do not compose!
    ✦ counter set is not a composition of counter and set CRDTs
Can we *program & reason about* replicated data types as an extension of their sequential counterparts?
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
module Counter : sig
    type t
    val read : t -> int
    val add : t -> int -> t
    val sub : t -> int -> t
    val mult : t -> int -> t
    val merge : lca:t -> v1:t -> v2:t -> t
end

= struct
    type t = int
    let read x = x
    let add x d = x + d
    let sub x d = x - d
    let mult x n = x * n
    let merge ~lca ~v1 ~v2 =
        lca + (v1 - lca) + (v2 - lca)
end
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
module Counter : sig
    type t
    val read : t -> int
    val add : t -> int -> t
    val sub : t -> int -> t
    val mult : t -> int -> t
    val merge : lca:t -> v1:t -> v2:t -> t
end = struct
    type t = int
    let read x = x
    let add x d = x + d
    let sub x d = x - d
    let mult x n = x * n
    let merge ~lca ~v1 ~v2 =
        lca + (v1 - lca) + (v2 - lca)
end
module Counter : sig
    type t
    val read : t -> int
    val add : t -> int -> t
    val sub : t -> int -> t
    val mult : t -> int -> t
    val merge : lca:t -> v1:t -> v2:t -> t
end = struct
    type t = int
    let read x = x
    let add x d = x + d
    let sub x d = x - d
    let mult x n = x * n
    let merge ~lca ~v1 ~v2 =
        lca + (v1 - lca) + (v2 - lca)
end
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 =
    lca + (v1 - lca) + (v2 - lca)
end

• 3-way merge function makes the counter suitable for distribution
module Counter : sig
  type t
  val read : t -> int
  val add : t -> int -> t
  val sub : t -> int -> t
  val mult : t -> int -> t
  val merge : lca:t -> v1:t -> v2:t -> t
end = struct
  type t = int
  let read x = x
  let add x d = x + d
  let sub x d = x - d
  let mult x n = x * n
  let merge ~lca ~v1 ~v2 = lca + (v1 - lca) + (v2 - lca)
end

- 3-way merge function makes the counter suitable for distribution
- Does not appeal to individual operations => independently extend data-type
Systems $\rightarrow$ PL
• CRDTs need to take care of systems level concerns such as exactly once delivery
• CRDTs need to take care of systems level concerns such as exactly once delivery

• 3-way merge handles it automatically
• CRDTs need to take care of systems level concerns such as exactly once delivery
• 3-way merge handles it automatically
• CRDTs need to take care of systems level concerns such as exactly once delivery
• 3-way merge handles it automatically
• CRDTs need to take care of systems level concerns such as exactly once delivery
• 3-way merge handles it automatically

22 = 21 + (21 - 21) + (22 - 21)
Does the 3-way merge idea generalise?
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
            (* at-least once semantics *)
end
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end

• Try replicating queues by asynchronously transmitting operations
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop : 'a t -> ('a * 'a t) option
    (* at-least once semantics *)
end

• Try replicating queues by asynchronously transmitting operations
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop : 'a t -> ('a * 'a t) option
    (* at-least once semantics *)
end

• Try replicating queues by asynchronously transmitting operations
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop : 'a t -> ('a * 'a t) option
    (* at-least once semantics *)
end

• Try replicating queues by asynchronously transmitting operations

[1,2]  

pop() → 1  pop() → 1

[2]  [2]
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop : 'a t -> ('a * 'a t) option
    (* at-least once semantics *)
end

• Try replicating queues by asynchronously transmitting operations
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end

• Try replicating queues by asynchronously transmitting operations
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop : 'a t -> ('a * 'a t) option
    (* at-least once semantics *)
end

• Try replicating queues by asynchronously transmitting operations

• Convergence is not sufficient; \textit{Intent} is not preserved
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop : 'a t -> ('a * 'a t) option
    (* at-least once semantics *)
end

- Try replicating queues by asynchronously transmitting operations

- Convergence is not sufficient; *Intent* is not preserved
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop : 'a t -> ('a * 'a t) option
    (* at-least once semantics *)
end

• Try replicating queues by asynchronously transmitting operations

• Convergence is not sufficient; Intent is not preserved
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop : 'a t -> ('a * 'a t) option
  (* at-least once semantics *)
end

- Try replicating queues by asynchronously transmitting operations

- Convergence is not sufficient; Intent is not preserved
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop : 'a t -> ('a * 'a t) option (* at-least once semantics *)
end

- Try replicating queues by asynchronously transmitting operations

- Convergence is not sufficient; **Intent** is not preserved
module type Queue = sig
  type 'a t
  val push : 'a t -> 'a -> 'a t
  val pop : 'a t -> ('a * 'a t) option
    (* at-least once semantics *)
end

• Try replicating queues by asynchronously transmitting operations

• Convergence is not sufficient; Intent is not preserved
Concretising Intent
Concretising Intent

- Intent is a woolly term
  - How can we formalise the intent of operations on a data structure?
Concretising Intent

• Intent is a woolly term
  ★ How can we formalise the intent of operations on a data structure?
Concretising Intent

• Intent is a woolly term
  ★ How can we formalise the intent of operations on a data structure?

• For a replicated queue,
Concretising Intent

• Intent is a woolly term
  ★ How can we formalise the intent of operations on a data structure?

• For a replicated queue,
  1. Any element popped in either v1 or v2 does not remain in v
Concretising Intent

• Intent is a woolly term
  ★ How can we formalise the intent of operations on a data structure?

• For a replicated queue,
  1. Any element popped in either v1 or v2 does not remain in v
  2. Any element pushed into either v1 or v2 appears in v
Concretising Intent

• Intent is a woolly term
  ★ *How can we formalise the intent of operations on a data structure?*

• For a replicated queue,
  1. Any element popped in either v₁ or v₂ does not remain in v
  2. Any element pushed into either v₁ or v₂ appears in v
  3. An element that remains untouched in l, v₁, v₂ remains in v
Concretising Intent

• Intent is a woolly term
  ★ How can we formalise the intent of operations on a data structure?

• For a replicated queue,
  1. Any element popped in either v1 or v2 does not remain in v
  2. Any element pushed into either v1 or v2 appears in v
  3. An element that remains untouched in l, v1, v2 remains in v
  4. Order of pairs of elements in l, v1, v2 must be preserved in m, if those elements are present in v.
Relational Specification
Relational Specification

Let’s define relations $R_{\text{mem}}$ and $R_{\text{ob}}$ to capture membership and ordering

- $R_{\text{mem}}[1,2,3] = \{1,2,3\}$
- $R_{\text{ob}}[1,2,3] = \{(1,2), (1,3), (2,3)\}$
• Let’s define relations $R_{\text{mem}}$ and $R_{\text{ob}}$ to capture membership and ordering

★ $R_{\text{mem}} [1,2,3] = \{1,2,3\}$
★ $R_{\text{ob}} [1,2,3] = \{ (1,2), (1,3), (2,3) \}$
Relational Specification

Let’s define relations $R_{\text{mem}}$ and $R_{\text{ob}}$ to capture membership and ordering

- $R_{\text{mem}}[1,2,3] = \{1,2,3\}$
- $R_{\text{ob}}[1,2,3] = \{(1,2), (1,3), (2,3)\}$

$$R_{\text{mem}}(v) = R_{\text{mem}}(l) \cap R_{\text{mem}}(v_1) \cap R_{\text{mem}}(v_2)$$

$$\cup R_{\text{mem}}(v_1) - R_{\text{mem}}(l) \cup R_{\text{mem}}(v_2) - R_{\text{mem}}(l)$$
Relational Specification

- Let’s define relations $R_{\text{mem}}$ and $R_{\text{ob}}$ to capture membership and ordering
  
  \[ R_{\text{mem}}[1,2,3] = \{1,2,3\} \]
  
  \[ R_{\text{ob}}[1,2,3] = \{ (1,2), (1,3), (2,3) \} \]
  
  \[
  R_{\text{mem}}(v) = R_{\text{mem}}(l) \cap R_{\text{mem}}(v_1) \cap R_{\text{mem}}(v_2) \sqcup R_{\text{mem}}(v_1) - R_{\text{mem}}(l) \sqcup R_{\text{mem}}(v_2) - R_{\text{mem}}(l)
  \]

1. Any element popped in either $v_1$ or $v_2$ does not remain in $v$
Relational Specification

- Let’s define relations $R_{mem}$ and $R_{ob}$ to capture membership and ordering
  - $R_{mem} \{1,2,3\} = \{1,2,3\}$
  - $R_{ob} \{1,2,3\} = \{(1,2), (1,3), (2,3)\}$

$$R_{mem}(v) = R_{mem}(l) \cap R_{mem}(v_1) \cap R_{mem}(v_2) \cup R_{mem}(v_1) - R_{mem}(l) \cup R_{mem}(v_2) - R_{mem}(l)$$

1. Any element popped in either $v_1$ or $v_2$ does not remain in $v$
2. Any element pushed into either $v_1$ or $v_2$ appears in $v$
Relational Specification

• Let’s define relations $R_{\text{mem}}$ and $R_{\text{ob}}$ to capture membership and ordering
  - $R_{\text{mem}}[1,2,3] = \{1,2,3\}$
  - $R_{\text{ob}}[1,2,3] = \{(1,2), (1,3), (2,3)\}$

$$R_{\text{mem}}(v) = R_{\text{mem}}(l) \cap R_{\text{mem}}(v_1) \cap R_{\text{mem}}(v_2) \cup R_{\text{mem}}(v_1) - R_{\text{mem}}(l) \cup R_{\text{mem}}(v_2) - R_{\text{mem}}(l)$$

1. Any element popped in either $v_1$ or $v_2$ does not remain in $v$
2. Any element pushed into either $v_1$ or $v_2$ appears in $v$
3. An element that remains untouched in $l$, $v_1$, $v_2$ remains in $v$
Relational Specification

\[ R_{ob}(\nu) \supseteq (R_{ob}(l) \cap R_{ob}(\nu_1) \cap R_{ob}(\nu_2) \]
\[ \cup R_{ob}(\nu_1) - R_{ob}(l) \cup R_{ob}(\nu_2) - R_{ob}(l)) \]
\[ \cap (R_{mem}(\nu) \times R_{mem}(\nu)) \]
Relational Specification

\[ R_{ob}(v) \supseteq (R_{ob}(l) \cap R_{ob}(v_1) \cap R_{ob}(v_2) \cup R_{ob}(v_1) - R_{ob}(l) \cup R_{ob}(v_2) - R_{ob}(l)) \cap (R_{mem}(v) \times R_{mem}(v)) \]

- RHS has to be confined to \( R_{mem}(v) \times R_{mem}(v) \) since certain orders might be missing
  - Consider \( l = [0], v_1 = [0,1], v_2 = [1], v = [1] \)
Relational Specification

\[ R_{ob}(v) \supseteq (R_{ob}(l) \cap R_{ob}(v_1) \cap R_{ob}(v_2) \]
\[ \cup R_{ob}(v_1) - R_{ob}(l) \cup R_{ob}(v_2) - R_{ob}(l)) \]
\[ \cap (R_{mem}(v) \times R_{mem}(v)) \]

- RHS has to be confined to \( R_{mem}(v) \times R_{mem}(v) \) since certain orders might be missing
  
  ★ Consider \( l = [0], v1= [0,1], v2 = [ ], v = [1] \)

- RHS is an underspecification since orders between concurrent insertions will only be present in \( R_{ob}(v) \)
  
  ★ Consider \( l = [ ], v1= [0], v2 = [1], v = [0,1] \)
\[ R_{\text{mem}} = \{1,2\} \]
\[ R_{\text{ob}} = \{ (1,2) \} \]
$R_{\text{mem}} = \{1,2\}$

$R_{\text{ob}} = \{(1,2)\}$

pop() $\rightarrow$ 1

$R_{\text{mem}} = \{2\}$

$R_{\text{ob}} = \{\}$. 
\[ R_{\text{mem}} = \{1,2\} \]
\[ R_{\text{ob}} = \{ (1,2) \} \]

\[ [1,2] \]

pop() \rightarrow 1

\[ R_{\text{mem}} = \{2\} \]
\[ R_{\text{ob}} = \{ \} \]

pop() \rightarrow 1

\[ R_{\text{mem}} = \{2\} \]
\[ R_{\text{ob}} = \{ \} \]
$R_{\text{mem}} = \{1,2\}$

$R_{\text{ob}} = \{ (1,2) \}$

Diagram:

- Initial state: $R_{\text{mem}} = \{1,2\}$, $R_{\text{ob}} = \{ (1,2) \}$
- After pop(): $R_{\text{mem}} = \{2\}$, $R_{\text{ob}} = \{ \}$
- After pop(): $R_{\text{mem}} = \{2\}$, $R_{\text{ob}} = \{ \}$
$R_{\text{mem}} = \{1\}$

$R_{\text{ob}} = \{\} \quad \text{[1]}$
push(2)

$R_{\text{mem}} = \{1\}$

$R_{\text{ob}} = \{\}\}$

$R_{\text{mem}} = \{1,2\}$

$R_{\text{ob}} = \{(1,2)\}$
\[ R_{\text{mem}} = \{1\} \]

\[ R_{\text{ob}} = \{\} \]

\[ \text{push}(2) \]

\[ R_{\text{mem}} = \{1,2\} \]

\[ R_{\text{ob}} = \{ (1,2) \} \]

\[ \text{push}(3) \]

\[ R_{\text{mem}} = \{1,3\} \]

\[ R_{\text{ob}} = \{ (1,3) \} \]
Use < as an arbitration function between concurrent insertions
\[ \text{R}_{\text{mem}} = \{1\} \]
\[ \text{R}_{\text{ob}} = \{\} \]

Use < as an arbitration function between concurrent insertions
Characteristic Relations

A sequence of relations $\overline{R}_T$ is called a characteristic relation of a data type $T$, if for every $x : T$ and $y : T$, $\overline{R}_T(x) = \overline{R}_T(y)$ iff $x$ and $y$ are extensionally equal as interpreted under $T$. 
Characteristic Relations

A sequence of relations $\overline{R}_T$ is called a characteristic relation of a data type $T$, if for every $x : T$ and $y : T$, $\overline{R}_T(x) = \overline{R}_T(y)$ iff $x$ and $y$ are extensionally equal as interpreted under $T$.

- $R_{\text{mem}}$ and $R_{\text{ob}}$ are the characteristic relations of queue
Characteristic Relations

A sequence of relations $\overline{R}_T$ is called a characteristic relation of a data type $T$, if for every $x : T$ and $y : T$, $\overline{R}_T(x) = \overline{R}_T(y)$ iff $x$ and $y$ are extensionally equal as interpreted under $T$.

- $R_{\text{mem}}$ and $R_{\text{ob}}$ are the characteristic relations of queue
- Appeals only to the sequential properties of the data type
  - Ignore distribution when defining characteristic relations.
Synthesizing Merge
Synthesizing Merge

• Semantics of merge in relational domain is quite standard across data types
Synthesizing Merge

- Semantics of merge in relational domain is quite standard across data types
  - Can we synthesise merge functions for arbitrary data type?
Synthesizing Merge

- Semantics of merge in relational domain is quite standard across data types

  ★ Can we synthesise merge functions for arbitrary data type?
Synthesizing Merge

- Semantics of merge in relational domain is quite standard across data types.

- Can we synthesise merge functions for arbitrary data type?

```
\begin{center}
\begin{tikzpicture}
  \node (l) at (0,0) {l};
  \node (v1) at (-2,-2) {R(v_1)};
  \node (v2) at (2,-2) {R(v_2)};
  \node (v) at (0,-4) {v};
  \node (v') at (0,-2) {v'};

  \draw[->] (l) to node {$\alpha$} (v1);
  \draw[->] (l) to node {$\alpha$} (v2);
  \draw[->] (v1) to node {$\alpha$} (v);
  \draw[->] (v2) to node {$\alpha$} (v);
  \draw[->] (v) to node {$\gamma$} (v');

  \node (l') at (0,-6) {R(l)};
  \draw[->] (l') to (l);

  \node (l'' at (0,-8) {R(v)};
  \draw[->] (l'') to (v');

\end{tikzpicture}
\end{center}
```

Abstraction function
Synthesizing Merge

- Semantics of merge in relational domain is quite standard across data types
  - Can we synthesise merge functions for arbitrary data type?

```
    R(l)
   /   \
 v   α   v
 /     \ \
| R(v₁)  |  v₂   | R(v₂)
|        |   α   |        \
|  v₁    |  v₂   |  v    |
|        |        |        |
| R(γ)   |        | R(v)  |
```

Abstraction function
Concretisation function
Synthesizing Merge

• Semantics of merge in relational domain is quite standard across data types

★ Can we synthesise merge functions for arbitrary data type?

- Abstraction function
- Concretisation function
- Synthesize (goal)
Synthesizing Merge

- Semantics of merge in relational domain is quite standard across data types
  - Can we synthesise merge functions for arbitrary data type?

Directed synthesis using the type of the characteristic relations
Abstraction Function: Queue
Abstraction Function: Queue

\[
\text{let rec } R_{\text{mem}} = \text{function} \\
| \text{[]} \rightarrow \emptyset \\
| \text{x::xs} \rightarrow \{x\} \cup R_{\text{mem}}(\text{xs})
\]
Abstraction Function: Queue

let rec \( R_{mem} \) = function
  | [] -> \( \emptyset \)
  | x::xs -> \{ x \} \cup R_{mem}(xs)

\( R_{mem} : \{ v : \text{int list} \} \rightarrow \mathcal{P}(\text{int}) \)
Abstraction Function: Queue

let rec \( R_{\text{mem}} \) = function
\[
\begin{align*}
& | \ [] \rightarrow \emptyset \\
& | \ x::xs \rightarrow \{x\} \cup R_{\text{mem}}(xs)
\end{align*}
\]

\( R_{\text{mem}} : \{v : \text{int list}\} \rightarrow \mathcal{P}(\text{int}) \)

let rec \( R_{\text{ob}} \) = function
\[
\begin{align*}
& | \ [] \rightarrow \emptyset \\
& | \ x::xs \rightarrow (\{x\} \times R_{\text{mem}}(xs)) \cup R_{\text{ob}}(xs)
\end{align*}
\]
Abstraction Function: Queue

let rec $R_{mem}$ = function
    | [] -> ∅
    | x::xs -> \{x\} $\cup$ $R_{mem}(xs)$

$R_{mem}$ : \{v : int list\} $\rightarrow$ $\mathcal{P}$(int)

let rec $R_{ob}$ = function
    | [] -> ∅
    | x::xs -> \{x\} $\times$ $R_{mem}(xs)$ $\cup$ $R_{ob}(xs)$

$R_{ob}$ : \{v : int list\} $\rightarrow$ $\mathcal{P}(R_{mem}(v) \times R_{mem}(v))$
Abstraction Function: Binary Tree

type 'a tree = | E
| N of 'a tree * 'a * 'a tree
Abstraction Function: Binary Tree

type 'a tree = | E
| N of 'a tree * 'a * 'a tree

let rec $R_{mem}$ = function
| E -> ∅
| N(l,x,r) -> $R_{mem}(l) \cup \{x\} \cup R_{mem}(r)$
Abstraction Function: Binary Tree

```ocaml
type 'a tree = | E
| N of 'a tree * 'a * 'a tree

let rec Rmem = function
| E -> Ø
| N(l,x,r) -> Rmem(l) ∪ {x} ∪ Rmem(r)

type label = L | R
let rec Rto = function
| E -> Ø
| N(l,x,r) ->
  let l_des = {x} × {L} × Rmem(l) in
  let r_des = {x} × {R} × Rmem(r) in
  Rto(l) ∪ l_des ∪ r_des ∪ Rto(r)
```

24
Abstraction Function: Binary Heap

type 'a tree = | E
          | N of 'a tree * 'a * 'a tree

let rec \( R_{mem} \) = function
    | E -> ∅
    | N(l,x,r) -> \( R_{mem}(l) \cup \{x\} \cup R_{mem}(r) \)
Abstraction Function: Binary Heap

\[
\text{type } 'a \text{ tree} = | E \\
| N \text{ of } 'a \text{ tree} * 'a * 'a \text{ tree}
\]

\[
\begin{align*}
\text{let rec } R_{\text{mem}} &= \text{function} \\
| E &\to \emptyset \\
| N(l,x,r) &\to R_{\text{mem}}(l) \cup \{x\} \cup R_{\text{mem}}(r)
\end{align*}
\]

\[
\begin{align*}
\text{let rec } R_{\text{ans}} &= \text{function} \\
| E &\to \emptyset \\
| N(l,x,r) &\to \\
&\quad \text{let } \text{des}_x = R_{\text{mem}}(l) \cup R_{\text{mem}}(r) \text{ in} \\
&\quad \text{let } r_{\text{ans}} = \{x\} \times \text{des}_x \text{ in} \\
&\quad R_{\text{ans}}(l) \cup r_{\text{ans}} \cup R_{\text{ans}}(r)
\end{align*}
\]
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Characteristic Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Heap</td>
<td>Membership ( (R_{\text{mem}}) ), Ancestor ( (R_{\text{ans}} \subseteq R_{\text{mem}} \times R_{\text{mem}}) )</td>
</tr>
<tr>
<td>Priority Queue</td>
<td>Membership ( (R_{\text{mem}}) )</td>
</tr>
<tr>
<td>Set</td>
<td>Membership ( (R_{\text{mem}}) )</td>
</tr>
<tr>
<td>Graph</td>
<td>Vertex ( (R_V) ), Edge ( (R_E) )</td>
</tr>
<tr>
<td>Functional Map</td>
<td>Key-Value ( (R_{k,v}) )</td>
</tr>
<tr>
<td>List</td>
<td>Membership ( (R_{\text{mem}}) ), Order ( (R_{ob}) )</td>
</tr>
<tr>
<td>Binary Tree</td>
<td>Membership ( (R_{\text{mem}}) ), Tree-order ( (R_{to} \subseteq R_{\text{mem}} \times \text{label} \times R_{\text{mem}}) )</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>Membership ( (R_{\text{mem}}) )</td>
</tr>
</tbody>
</table>

Table 1. Characteristic relations for various data types
Compositionality: Pair
Compositionality: Pair

- The merge of a pair is the merge of the corresponding constituents
Compositionality: Pair

- The merge of a pair is the merge of the corresponding constituents

- A pair data type is defined by the relations:

  \[
  \text{let } R_{fst} = \text{fun } (x, \_ ) \to \{x\} \quad \text{let } R_{snd} = \text{fun } (\_, y) \to \{y\}
  \]
Compositionality: Pair

• The merge of a pair is the merge of the corresponding constituents

• A pair data type is defined by the relations:

\[
\begin{align*}
\text{let } R_{fst} & = \text{fun } (x, \_ ) \rightarrow \{ x \} & \text{let } R_{snd} & = \text{fun } (\_, y) \rightarrow \{ y \}
\end{align*}
\]

• Assume that the pair is composed of 2 counters. The counter merge spec is: \( \phi_c(l, v_1, v_2, v) \Leftrightarrow v = l + (v_1 - l) + (v_2 - l) \)
Compositionality: Pair

• The merge of a pair is the merge of the corresponding constituents

• A pair data type is defined by the relations:

\[
\text{let } R_{fst} = \text{fun } (x, -) \rightarrow \{x\} \quad \text{let } R_{snd} = \text{fun } (-, y) \rightarrow \{y\}
\]

• Assume that the pair is composed of 2 counters. The counter merge spec is: \( \phi_c(l, v_1, v_2, v) \iff v = l + (v_1 - l) + (v_2 - l) \)

• Then, pair merge spec is:

\[
\phi_{c\times c}(l, v_1, v_2, v) \iff \forall x, y, z, s. x \in R_{fst}(l) \land y \in R_{fst}(v_1) \land z \in R_{fst}(v_2) \\
\land \phi_c(x, y, z, s) \Rightarrow s \in R_{fst}(v) \\
\land \forall s. s \in R_{fst}(v) \Rightarrow \exists x, y, z. x \in R_{fst}(l) \land y \in R_{fst}(v_1) \\
\land z \in R_{fst}(v_2) \land \phi_c(x, y, z, s) \\
\land \ldots \text{(respectively for } R_{snd})
\]
Generalising Pairs to Ordinates
Generalising Pairs to Ordinates

• An alternative characteristic relation for a pair is:

\[
\text{let } R_{\text{pair}} (x, y) = \{ (1, x), (2, y) \}
\]
Generalising Pairs to Ordinates

• An alternative characteristic relation for a pair is:

\[
\text{let } R_{\text{pair}} (x, y) = \{(1, x), (2, y)\}
\]

\[
R_{\text{pair}} : \{v : \text{counter} \times \text{counter}\} \rightarrow \mathcal{P} (\text{int} \times \text{counter})
\]
Generalising Pairs to Ordinates

• An alternative characteristic relation for a pair is:

\[
\text{let } R_{pair} (x, y) = \{(1, x), (2, y)\}
\]

\[
R_{pair} : \{v : \text{counter} \times \text{counter}\} \rightarrow \mathcal{P}(\text{int} \times \text{counter})
\]

• Corresponding merge specification is:
Generalising Pairs to Ordinates

• An alternative characteristic relation for a pair is:

\[
\text{let } R_{\text{pair}} (x, y) = \{(1, x), (2, y)\}
\]

\[
R_{\text{pair}} : \{\nu : \text{counter} \times \text{counter}\} \to \mathcal{P} (\text{int} \times \text{counter})
\]

• Corresponding merge specification is:

\[
\phi_{c \times c} = \forall (k : \text{int}). \forall (x, y, z, s : \text{counter}). (k, x) \in R_{\text{pair}}(l) \land (k, y) \in R_{\text{pair}}(\nu_1) \\
\land (k, z) \in R_{\text{pair}}(\nu_2) \land \phi_c(x, y, z, s) \Rightarrow (k, s) \in R(\nu) \\
\land \forall (k : \text{int}). \forall (s : \text{counter}). (k, s) \in R_{\text{pair}}(\nu) \Rightarrow \exists (x, y, z : \text{counter}). (k, x) \in R_{\text{pair}}(l) \\
\land (k, y) \in R_{\text{pair}}(\nu_1) \land (k, z) \in R_{\text{pair}}(\nu_2) \land \phi_c(x, y, z, s)
\]
Generalising Pairs to Ordinates

• An alternative characteristic relation for a pair is:

\[
\text{let } R_{\text{pair}} (x, y) = \{(1, x), (2, y)\}
\]

\[R_{\text{pair}} : \{v : \text{counter} \times \text{counter}\} \rightarrow \mathcal{P} (\text{int} \times \text{counter})\]

• Corresponding merge specification is:

\[
\phi_{c \times c} = \forall (k : \text{int}). \forall (x, y, z, s : \text{counter}). (k, x) \in R_{\text{pair}}(l) \land (k, y) \in R_{\text{pair}}(v_1) \\
\land (k, z) \in R_{\text{pair}}(v_2) \land \phi_c(x, y, z, s) \Rightarrow (k, s) \in R(v) \\
\land \forall (k : \text{int}). \forall (s : \text{counter}). (k, s) \in R_{\text{pair}}(v) \Rightarrow \exists (x, y, z : \text{counter}). (k, x) \in R_{\text{pair}}(l) \\
\land (k, y) \in R_{\text{pair}}(v_1) \land (k, z) \in R_{\text{pair}}(v_2) \land \phi_c(x, y, z, s)
\]

• Given appropriate characteristic relation for a n-tuple \(R_{n\text{-tuple}}\), the same merge specification can be used.
Generalising Pairs to Ordinates

• Similar encoding can be given to maps with non-mergeable types as keys and mergeable types as values

\[ R_k : \{ v : (\text{string}, \text{int}) \text{ map} \} \rightarrow \mathcal{P}(\text{string}), \]
\[ R_{k\nu} : \{ v : (\text{string}, \text{int}) \text{ map} \} \rightarrow \mathcal{P}(R_k(v) \times \text{counter}) \]
Types of Characteristic Relations
Types of Characteristic Relations

- Practical characteristic relations fall into 3 types:
Types of Characteristic Relations

- Practical characteristic relations fall into 3 types:
  - Membership ($R_{\text{mem}}$)

$$R : \{v : T\} \rightarrow \mathcal{P}(\overline{T})$$

where $T$ is a non-mergeable type
Types of Characteristic Relations

- Practical characteristic relations fall into 3 types:
  - Membership ($R_{mem}$)
    \[ R : \{ v : T \} \rightarrow \mathcal{P}(\overline{T}) \], where $T$ is a non-mergeable type
  - Ordering ($R_{ob}$, $R_{ans}$, $R_{to}$)
    \[ R : \{ v : T \} \rightarrow \mathcal{P}(\rho) \]
    - where $\rho$ is a sequence of non-mergeable types and other relations, which flattens to a sequence of non-mergeable types
Types of Characteristic Relations

- Practical characteristic relations fall into 3 types:
  - Membership ($R_{mem}$)
    \[ R : \{v : T\} \rightarrow \mathcal{P}(\bar{T}), \text{ where } T \text{ is a non-mergeable type} \]
  - Ordering ($R_{ob}, R_{ans}, R_{to}$)
    \[ R : \{v : T\} \rightarrow \mathcal{P}(\rho) \]
    where $\rho$ is a sequence of non-mergeable types and other relations, which flattens to a sequence of non-mergeable types
  - Ordinates ($R_{pair}, R_{kv}$)
    \[ R : \{v : T\} \rightarrow \mathcal{P}(\bar{T} \times \bar{\tau}), \text{ where } \tau \text{ is a mergeable type} \]
Deriving merge spec

\[
R: \{ v : T \} \rightarrow \mathcal{P}(\overline{T})
\]
\[
\phi_T(l, v_1, v_2, v) \supseteq \forall (\overline{x} : \overline{T}). \overline{x} \in (R(l) \circ R(v_1) \circ R(v_2)) \leftrightarrow \overline{x} \in R(v)
\]

[Set-Merge]

\[
R: \{ v : T \} \rightarrow \mathcal{P}(\rho) \quad [\rho] = \overline{T}
\]
\[
\phi_T(l, v_1, v_2, v) \supseteq \forall (\overline{x} : \overline{T}). \overline{x} \in (R(l) \circ R(v_1) \circ R(v_2) \cap \rho) \Rightarrow \overline{x} \in R(v)
\]

[Order-Merge-1]

\[
R: \{ v : T \} \rightarrow \mathcal{P}(\rho) \quad [\rho] = \overline{T}
\]
\[
\phi_T(l, v_1, v_2, v) \supseteq \forall (\overline{x} : \overline{T}). \overline{x} \in R(v) \Rightarrow \overline{x} \in \rho
\]

[Order-Merge-2]

\[
R: \{ v : T \} \rightarrow \mathcal{P}(\rho) \quad [\rho] = \overline{T} \times \overline{T} \quad \overline{T} \neq \emptyset
\]
\[
\phi_T(l, v_1, v_2, v) \supseteq \forall (\overline{k} : \overline{T}). \forall (\overline{y}, \overline{z}, \overline{s} : \overline{T}). (\overline{k}, \overline{x}) \in R_{\overline{l}}(l) \land (\overline{k}, \overline{y}) \in R_{\overline{v}_1}(v_1) \land (\overline{k}, \overline{z}) \in R_{\overline{v}_2}(v_2)
\]
\[
\land \overline{k} \in (R_{\overline{k}}(l) \circ R_{\overline{k}}(v_1) \circ R_{\overline{k}}(v_2)) \land \bigwedge \phi_{\overline{t}_i}(x_i, y_i, z_i, s_i) \land (\overline{k}, \overline{s}) \in \rho \Rightarrow (\overline{k}, \overline{s}) \in R(v)
\]

[Rel-Merge-1]

\[
R: \{ v : T \} \rightarrow \mathcal{P}(\rho) \quad [\rho] = \overline{T} \times \overline{T} \quad \overline{T} \neq \emptyset
\]
\[
\phi_T(l, v_1, v_2, v) \supseteq \forall (\overline{k} : \overline{T}). \forall (\overline{s} : \overline{T}). (\overline{k}, \overline{s}) \in R(v) \Rightarrow (\overline{k}, \overline{s}) \in \rho
\]
\[
\land \exists (\overline{x}, \overline{y}, \overline{z} : \overline{T}). (\overline{k}, \overline{x}) \in R_{\overline{l}}(l) \land (\overline{k}, \overline{y}) \in R_{\overline{v}_1}(v_1) \land (\overline{k}, \overline{z}) \in R_{\overline{v}_2}(v_2)
\]
\[
\land \overline{k} \in (R_{\overline{k}}(l) \circ R_{\overline{k}}(v_1) \circ R_{\overline{k}}(v_2)) \land \bigwedge \phi_{\overline{t}_i}(x_i, y_i, z_i, s_i)
\]

[Rel-Merge-2]
Deriving merge spec

\[
R : \{ v : T \} \rightarrow \mathcal{P} (\overline{T}) \\
\phi_T(l, v_1, v_2, v) \supseteq \forall (\overline{x} : \overline{T}), \overline{x} \in (R(l) \circ R(v_1) \circ R(v_2)) \iff \overline{x} \in R(v) 
\]

[Set-Merge]

\[
R : \{ v : T \} \rightarrow \mathcal{P} (\rho) \quad [\rho] = \overline{T} \\
\phi_T(l, v_1, v_2, v) \supseteq \forall (\overline{x} : \overline{T}), \overline{x} \in (R(l) \circ R(v_1) \circ R(v_2) \cap \rho) \Rightarrow \overline{x} \in R(v) 
\]

[Order-Merge-1]

\[
R : \{ v : T \} \rightarrow \mathcal{P} (\rho) \quad [\rho] = \overline{T} \\
\phi_T(l, v_1, v_2, v) \supseteq \forall (\overline{x} : \overline{T}), \overline{x} \in R(v) \Rightarrow \overline{x} \in \rho 
\]

[Order-Merge-2]

\[
R : \{ v : T \} \rightarrow \mathcal{P} (\rho) \quad [\rho] = \overline{T} \times \overline{T} \quad \overline{T} \neq \emptyset \\
\phi_T(l, v_1, v_2, v) \supseteq \forall (\overline{k} : \overline{T}), \forall (\overline{s} : \overline{T}), (\overline{k}, \overline{s}) \in R_k(l) \land (\overline{k}, \overline{y}) \in R_k(v_1) \land (\overline{k}, \overline{z}) \in R_k(v_2) \\
\land \left( \overline{k} \in (R_k(l) \circ R_k(v_1) \circ R_k(v_2)) \land \bigwedge_i \phi_{r_i}(x_i, y_i, z_i, s_i) \land (\overline{k}, \overline{s}) \in \rho \Rightarrow (\overline{k}, \overline{s}) \in R(v) \right) 
\]

[Rel-Merge-1]

\[
R : \{ v : T \} \rightarrow \mathcal{P} (\rho) \quad [\rho] = \overline{T} \times \overline{T} \quad \overline{T} \neq \emptyset \\
\phi_T(l, v_1, v_2, v) \supseteq \forall (\overline{k} : \overline{T}), \forall (\overline{s} : \overline{T}), (\overline{k}, \overline{s}) \in R(v) \Rightarrow (\overline{k}, \overline{s}) \in \rho \\
\land \exists (\overline{x}, \overline{y}, \overline{z} : \overline{T}), (\overline{k}, \overline{x}) \in R_k(l) \land (\overline{k}, \overline{y}) \in R_k(v_1) \land (\overline{k}, \overline{z}) \in R_k(v_2) \\
\land \left( \overline{k} \in (R_k(l) \circ R_k(v_1) \circ R_k(v_2)) \land \bigwedge_i \phi_{r_i}(x_i, y_i, z_i, s_i) \right) 
\]

[Rel-Merge-2]

- Not complete, but practical
- Can derive merge spec for
  - Data structures: Set, Heap, Graph, Queue, TreeDoc
  - Larger apps: TPC-C, TPC-E, Twissandra, Rubis
Distributed Implementation
Distributed Implementation

• For making this programming model practical, we need to:
  ★ Quickly compute LCA
  ★ Optimise storage through sharing
  ★ Optimise network transmissions (state based merge)
Distributed Implementation

• For making this programming model practical, we need to:
  ★ Quickly compute LCA
  ★ Optimise storage through sharing
  ★ Optimise network transmissions (state based merge)

• Irmin
  ★ A reimplementation of Git in pure OCaml
  ★ Arbitrary OCaml objects, not just files + User-defined 3-way merges
  ★ Only transmit diffs over the network
  ★ Multiple storage backends including in-memory, file systems, log-structured-merge database, distributed databases
Performance
Performance

• What is the size of diff compared to the size of data structure?
Performance

• What is the size of diff compared to the size of data structure?

• Setup
  ★ 2 Replicas, fixed number of rounds, each round has N operations
  ★ 75% inserts, 25% deletions
  ★ Synchronise after each round
Performance

• What is the size of diff compared to the size of data structure?

• Setup
  ★ 2 Replicas, fixed number of rounds, each round has N operations
  ★ 75% inserts, 25% deletions
  ★ Synchronise after each round

[Graphs showing performance comparison between Binary Heap and Growable Array]
Thanks for listening!