# OpSets: Sequential Specifications for Replicated Datatypes Proof Document

Martin Kleppmann[1], Victor B. F. Gomes[1], Dominic P. Mulligan[2], and Alastair R. Beresford[1]

[1]Department of Computer Science and Technology, University of Cambridge, UK
[2]Security Research Group, Arm Research, Cambridge, UK

**Abstract**

We introduce OpSets, an executable framework for specifying and reasoning about the semantics of replicated datatypes that provide eventual consistency in a distributed system, and for mechanically verifying algorithms that implement these datatypes. Our approach is simple but expressive, allowing us to succinctly specify a variety of abstract datatypes, including maps, sets, lists, text, graphs, trees, and registers. Our datatypes are also composable, enabling the construction of complex data structures. To demonstrate the utility of OpSets for analysing replication algorithms, we highlight an important correctness property for collaborative text editing that has traditionally been overlooked; algorithms that do not satisfy this property can exhibit awkward interleaving of text. We use OpSets to specify this correctness property and prove that although one existing replication algorithm satisfies this property, several other published algorithms do not.

## Contents

# 1 Abstract OpSet

In this section, we define a general-purpose OpSet abstraction that is not specific to any one particular datatype. We develop a library of useful lemmas that we can build upon later when reasoning about a specific datatype.

**theory** *OpSet*
**imports** *Main*
**begin**

## 1.1 OpSet definition

An OpSet is a set of (ID, operation) pairs with an associated total order on IDs (represented here with the *linorder* typeclass), and satisfying the following properties:

1. The ID is unique (that is, if any two pairs in the set have the same ID, then their operation is also the same).
2. If the operation references the IDs of any other operations, those referenced IDs are less than that of the operation itself, according to the total order on IDs. To avoid assuming anything about the structure of operations here, we use a function *deps* that returns the set of dependent IDs for a given operation. This requirement is a weak expression of causality: an operation can only depend on causally prior operations, and by making the total order on IDs a linear extension of the causal order, we can easily ensure that any referenced IDs are less than that of the operation itself.
3. The OpSet is finite (but we do not assume any particular maximum size).

**locale** *opset* =
**fixes** *opset* :: ('*oid*::{*linorder*} × '*oper*) *set*
**and** *deps* :: '*oper* ⇒ '*oid set*

**assumes** *unique-oid*: $(oid, op1) \in opset \Longrightarrow (oid, op2) \in opset \Longrightarrow op1 = op2$
**and** *ref-older*: $(oid, oper) \in opset \Longrightarrow ref \in deps\ oper \Longrightarrow ref < oid$
**and** *finite-opset*: *finite opset*

We prove that any subset of an OpSet is also a valid OpSet. This is the case because, although an operation can depend on causally prior operations, the OpSet does not require those prior operations to actually exist. This weak assumption makes the OpSet model more general and simplifies reasoning about OpSets.

**lemma** *opset-subset*:
**assumes** *opset Y deps*
**and** $X \subseteq Y$
**shows** *opset X deps*
**proof**
**fix** *oid op1 op2*
**assume** $(oid, op1) \in X$ **and** $(oid, op2) \in X$
**thus** *op1 = op2*
**using** *assms* **by** (*meson opset.unique-oid subsetD*)
**next**
**fix** *oid oper ref*
**assume** $(oid, oper) \in X$ **and** $ref \in deps\ oper$
**thus** $ref < oid$
**using** *assms* **by** (*meson opset.ref-older rev-subsetD*)
**next**
**show** *finite X*
**using** *assms opset.finite-opset finite-subset* **by** *blast*
**qed**

**lemma** *opset-insert*:
**assumes** *opset* (*insert x ops*) *deps*
**shows** *opset ops deps*
**using** *assms opset-subset* **by** *blast*

**lemma** *opset-sublist*:
**assumes** *opset* (*set* (*xs* @ *ys* @ *zs*)) *deps*
**shows** *opset* (*set* (*xs* @ *zs*)) *deps*
**proof** −
**have** *set* (*xs* @ *zs*) ⊆ *set* (*xs* @ *ys* @ *zs*)
**by** *auto*
**thus** *opset* (*set* (*xs* @ *zs*)) *deps*
**using** *assms opset-subset* **by** *blast*
**qed**

## 1.2 Helper lemmas about lists

Some general-purpose lemas about lists and sets that are helpful for subsequent proofs.

**lemma** *distinct-rem-mid*:

**assumes** *distinct* (*xs* @ [*x*] @ *ys*)
**shows** *distinct* (*xs* @ *ys*)
**using** *assms* **by** (*induction ys rule*: *rev-induct*, *simp-all*)

**lemma** *distinct-fst-append*:
**assumes** $x \in$ *set* (*map fst xs*)
**and** *distinct* (*map fst* (*xs* @ *ys*))
**shows** $x \notin$ *set* (*map fst ys*)
**using** *assms* **by** (*induction ys*, *force*+)

**lemma** *distinct-set-remove-last*:
**assumes** *distinct* (*xs* @ [*x*])
**shows** *set xs* = *set* (*xs* @ [*x*]) − {*x*}
**using** *assms* **by** *force*

**lemma** *distinct-set-remove-mid*:
**assumes** *distinct* (*xs* @ [*x*] @ *ys*)
**shows** *set* (*xs* @ *ys*) = *set* (*xs* @ [*x*] @ *ys*) − {*x*}
**using** *assms* **by** *force*

**lemma** *distinct-list-split*:
**assumes** *distinct xs*
**and** *xs* = *xa* @ *x* # *ya*
**and** *xs* = *xb* @ *x* # *yb*
**shows** *xa* = *xb* ∧ *ya* = *yb*
**using** *assms* **proof**(*induction xs arbitrary*: *xa xb x*)
**fix** *xa xb x*
**assume** [] = *xa* @ *x* # *ya*
**thus** *xa* = *xb* ∧ *ya* = *yb*
**by** *auto*
**next**
**fix** *a xs xa xb x*
**assume** *IH*: ⋀*xa xb x*. *distinct xs* ⟹ *xs* = *xa* @ *x* # *ya* ⟹ *xs* = *xb* @ *x* # *yb*
⟹ *xa* = *xb* ∧ *ya* = *yb*
**and** *distinct* (*a* # *xs*) **and** *a* # *xs* = *xa* @ *x* # *ya* **and** *a* # *xs* = *xb* @ *x* # *yb*
**thus** *xa* = *xb* ∧ *ya* = *yb*
**by**(*case-tac xa*; *case-tac xb*) *auto*
**qed**

**lemma** *distinct-append-swap*:
**assumes** *distinct* (*xs* @ *ys*)
**shows** *distinct* (*ys* @ *xs*)
**using** *assms* **by** (*induction ys*, *auto*)

**lemma** *append-subset*:
**assumes** *set xs* = *set* (*ys* @ *zs*)
**shows** *set ys* ⊆ *set xs* **and** *set zs* ⊆ *set xs*
**by** (*metis Un-iff assms set-append subsetI*)+

```
lemma append-set-rem-last:
  assumes set (xs @ [x]) = set (ys @ [x] @ zs)
    and distinct (xs @ [x]) and distinct (ys @ [x] @ zs)
  shows set xs = set (ys @ zs)
proof −
  have distinct xs
    using assms distinct-append by blast
  moreover from this have set xs = set (xs @ [x]) − {x}
    by (meson assms distinct-set-remove-last)
  moreover have distinct (ys @ zs)
    using assms distinct-rem-mid by simp
  ultimately show set xs = set (ys @ zs)
    using assms distinct-set-remove-mid by metis
qed

lemma distinct-map-fst-remove1:
  assumes distinct (map fst xs)
  shows distinct (map fst (remove1 x xs))
  using assms proof(induction xs)
  case Nil
  then show distinct (map fst (remove1 x []))
    by simp
next
  case (Cons a xs)
  hence IH: distinct (map fst (remove1 x xs))
    by simp
  then show distinct (map fst (remove1 x (a # xs)))
  proof(cases a = x)
    case True
    then show ?thesis
      using Cons.prems by auto
  next
    case False
    moreover have fst a ∉ fst ‘ set (remove1 x xs)
      by (metis (no-types, lifting) Cons.prems distinct.simps(2) image-iff
          list.simps(9) notin-set-remove1 set-map)
    ultimately show ?thesis
      using IH by auto
  qed
qed
```

## 1.3 The *spec-ops* predicate

The *spec-ops* predicate describes a list of (ID, operation) pairs that corresponds to the linearisation of an OpSet, and which we use for sequentially interpreting the OpSet. A list satisfies *spec-ops* iff it is sorted in ascending order of IDs, if the IDs are unique, and if every operation's dependencies have lower IDs than the operation itself. A list is implicitly finite in Isabelle/HOL.

These requirements correspond to the OpSet definition above, and indeed we prove later that every OpSet has a linearisation that satisfies *spec-ops*.

**definition** *spec-ops* :: ('*oid*::{*linorder*} × '*oper*) *list* ⇒ ('*oper* ⇒ '*oid set*) ⇒ *bool*
**where**
  *spec-ops ops deps* ≡ (*sorted* (*map fst ops*) ∧ *distinct* (*map fst ops*) ∧
        (∀ *oid oper ref*. (*oid*, *oper*) ∈ *set ops* ∧ *ref* ∈ *deps oper* ⟶ *ref* < *oid*))

**lemma** *spec-ops-empty*:
  **shows** *spec-ops* [] *deps*
  **by** (*simp add*: *spec-ops-def*)

**lemma** *spec-ops-distinct*:
  **assumes** *spec-ops ops deps*
  **shows** *distinct ops*
  **using** *assms distinct-map spec-ops-def* **by** *blast*

**lemma** *spec-ops-distinct-fst*:
  **assumes** *spec-ops ops deps*
  **shows** *distinct* (*map fst ops*)
  **using** *assms* **by** (*simp add*: *spec-ops-def*)

**lemma** *spec-ops-sorted*:
  **assumes** *spec-ops ops deps*
  **shows** *sorted* (*map fst ops*)
  **using** *assms* **by** (*simp add*: *spec-ops-def*)

**lemma** *spec-ops-rem-cons*:
  **assumes** *spec-ops* (*x* # *xs*) *deps*
  **shows** *spec-ops xs deps*
**proof** −
  **have** *sorted* (*map fst* (*x* # *xs*)) **and** *distinct* (*map fst* (*x* # *xs*))
    **using** *assms spec-ops-def* **by** *blast*+
  **moreover from** *this* **have** *sorted* (*map fst xs*)
    **by** *simp*
  **moreover have** ∀ *oid oper ref*. (*oid*, *oper*) ∈ *set xs* ∧ *ref* ∈ *deps oper* ⟶ *ref* < *oid*
    **by** (*meson assms set-subset-Cons spec-ops-def subsetCE*)
  **ultimately show** *spec-ops xs deps*
    **by** (*simp add*: *spec-ops-def*)
**qed**

**lemma** *spec-ops-rem-last*:
  **assumes** *spec-ops* (*xs* @ [*x*]) *deps*
  **shows** *spec-ops xs deps*
**proof** −
  **have** *sorted* (*map fst* (*xs* @ [*x*])) **and** *distinct* (*map fst* (*xs* @ [*x*]))
    **using** *assms spec-ops-def* **by** *blast*+
  **moreover from** *this* **have** *sorted* (*map fst xs*) **and** *distinct xs*

**by** (*auto simp add*: *sorted-append distinct-butlast distinct-map*)
**moreover have** $\forall$ *oid oper ref*. (*oid*, *oper*) $\in$ *set xs* $\wedge$ *ref* $\in$ *deps oper* $\longrightarrow$ *ref* $<$ *oid*
**by** (*metis assms butlast-snoc in-set-butlastD spec-ops-def*)
**ultimately show** *spec-ops xs deps*
**by** (*simp add*: *spec-ops-def*)
**qed**

**lemma** *spec-ops-remove1*:
**assumes** *spec-ops xs deps*
**shows** *spec-ops* (*remove1 x xs*) *deps*
**using** *assms distinct-map-fst-remove1 spec-ops-def*
**by** (*metis notin-set-remove1 sorted-map-remove1 spec-ops-def*)

**lemma** *spec-ops-ref-less*:
**assumes** *spec-ops xs deps*
**and** (*oid*, *oper*) $\in$ *set xs*
**and** *r* $\in$ *deps oper*
**shows** *r* $<$ *oid*
**using** *assms spec-ops-def* **by** *force*

**lemma** *spec-ops-ref-less-last*:
**assumes** *spec-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
**and** *r* $\in$ *deps oper*
**shows** *r* $<$ *oid*
**using** *assms spec-ops-ref-less* **by** *fastforce*

**lemma** *spec-ops-id-inc*:
**assumes** *spec-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
**and** *x* $\in$ *set* (*map fst xs*)
**shows** *x* $<$ *oid*
**proof** $-$
**have** *sorted* ((*map fst xs*) @ (*map fst* [(*oid*, *oper*)]))
**using** *assms*(*1*) **by** (*simp add*: *spec-ops-def*)
**hence** $\forall$ *i* $\in$ *set* (*map fst xs*). *i* $\leq$ *oid*
**by** (*simp add*: *sorted-append*)
**moreover have** *distinct* ((*map fst xs*) @ (*map fst* [(*oid*, *oper*)]))
**using** *assms*(*1*) **by** (*simp add*: *spec-ops-def*)
**hence** $\forall$ *i* $\in$ *set* (*map fst xs*). *i* $\neq$ *oid*
**by** *auto*
**ultimately show** *x* $<$ *oid*
**using** *assms*(*2*) *le-neq-trans* **by** *auto*
**qed**

**lemma** *spec-ops-add-last*:
**assumes** *spec-ops xs deps*
**and** $\forall$ *i* $\in$ *set* (*map fst xs*). *i* $<$ *oid*
**and** $\forall$ *ref* $\in$ *deps oper*. *ref* $<$ *oid*
**shows** *spec-ops* (*xs* @ [(*oid*, *oper*)]) *deps*

**proof** −
**have** *sorted* ((*map fst xs*) @ [*oid*])
**using** *assms sorted-append spec-ops-sorted* **by** *fastforce*
**moreover have** *distinct* ((*map fst xs*) @ [*oid*])
**using** *assms spec-ops-distinct-fst* **by** *fastforce*
**moreover have** ∀ *oid oper ref*. (*oid*, *oper*) ∈ *set xs* ∧ *ref* ∈ *deps oper* ⟶ *ref* < *oid*
**using** *assms*(*1*) *spec-ops-def* **by** *fastforce*
**hence** ∀ *i opr r*. (*i*, *opr*) ∈ *set* (*xs* @ [(*oid*, *oper*)]) ∧ *r* ∈ *deps opr* ⟶ *r* < *i*
**using** *assms*(*3*) **by** *auto*
**ultimately show** *spec-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
**by** (*simp add*: *spec-ops-def*)
**qed**

**lemma** *spec-ops-add-any*:
**assumes** *spec-ops* (*xs* @ *ys*) *deps*
**and** ∀ *i* ∈ *set* (*map fst xs*). *i* < *oid*
**and** ∀ *i* ∈ *set* (*map fst ys*). *oid* < *i*
**and** ∀ *ref* ∈ *deps oper*. *ref* < *oid*
**shows** *spec-ops* (*xs* @ [(*oid*, *oper*)] @ *ys*) *deps*
**using** *assms* **proof**(*induction ys rule*: *rev-induct*)
**case** *Nil*
**then show** *spec-ops* (*xs* @ [(*oid*, *oper*)] @ []) *deps*
**by** (*simp add*: *spec-ops-add-last*)
**next**
**case** (*snoc y ys*)
**have** *IH*: *spec-ops* (*xs* @ [(*oid*, *oper*)] @ *ys*) *deps*
**proof** −
**from** *snoc* **have** *spec-ops* (*xs* @ *ys*) *deps*
**by** (*metis append-assoc spec-ops-rem-last*)
**thus** *spec-ops* (*xs* @ [(*oid*, *oper*)] @ *ys*) *deps*
**using** *assms*(*2*) *snoc* **by** *auto*
**qed**
**obtain** *yi yo* **where** *y-pair*: *y* = (*yi*, *yo*)
**by** *force*
**have** *oid-yi*: *oid* < *yi*
**by** (*simp add*: *snoc.prems*(*3*) *y-pair*)
**have** *yi-biggest*: ∀ *i* ∈ *set* (*map fst* (*xs* @ [(*oid*, *oper*)] @ *ys*)). *i* < *yi*
**proof** −
**have** ∀ *i* ∈ *set* (*map fst xs*). *i* < *yi*
**using** *oid-yi assms*(*2*) *less-trans* **by** *blast*
**moreover have** ∀ *i* ∈ *set* (*map fst ys*). *i* < *yi*
**by** (*metis UnCI append-assoc map-append set-append snoc.prems*(*1*) *spec-ops-id-inc y-pair*)
**ultimately show** *?thesis*
**using** *oid-yi* **by** *auto*
**qed**
**have** *sorted* (*map fst* (*xs* @ [(*oid*, *oper*)] @ *ys* @ [*y*]))
**proof** −

```
    from IH have sorted (map fst (xs @ [(oid, oper)] @ ys))
      using spec-ops-def by blast
    hence sorted (map fst (xs @ [(oid, oper)] @ ys) @ [yi])
      using yi-biggest
      by (simp add: sorted-append dual-order.strict-implies-order)
    thus sorted (map fst (xs @ [(oid, oper)] @ ys @ [y]))
      by (simp add: y-pair)
  qed
  moreover have distinct (map fst (xs @ [(oid, oper)] @ ys @ [y]))
  proof −
    have distinct (map fst (xs @ [(oid, oper)] @ ys) @ [yi])
      using IH yi-biggest spec-ops-def
      by (metis distinct.simps(2) distinct1-rotate order-less-irrefl rotate1.simps(2))
    thus distinct (map fst (xs @ [(oid, oper)] @ ys @ [y]))
      by (simp add: y-pair)
  qed
  moreover have ∀ i opr r. (i, opr) ∈ set (xs @ [(oid, oper)] @ ys @ [y])
                  ∧ r ∈ deps opr ⟶ r < i
  proof −
    have ∀ i opr r. (i, opr) ∈ set (xs @ [(oid, oper)] @ ys) ∧ r ∈ deps opr ⟶ r <
i
      by (meson IH spec-ops-def)
    moreover have ∀ ref. ref ∈ deps yo ⟶ ref < yi
     by (metis spec-ops-ref-less append-is-Nil-conv last-appendR last-in-set last-snoc
         list.simps(3) snoc.prems(1) y-pair)
    ultimately show ?thesis
      using y-pair by auto
  qed
  ultimately show spec-ops (xs @ [(oid, oper)] @ ys @ [y]) deps
    using spec-ops-def by blast
qed

lemma spec-ops-split:
  assumes spec-ops xs deps
    and oid ∉ set (map fst xs)
  shows ∃ pre suf. xs = pre @ suf ∧
          (∀ i ∈ set (map fst pre). i < oid) ∧
          (∀ i ∈ set (map fst suf). oid < i)
  using assms proof(induction xs rule: rev-induct)
  case Nil
  then show ?case by force
next
  case (snoc x xs)
  obtain xi xr where y-pair: x = (xi, xr)
    by force
  obtain pre suf where IH: xs = pre @ suf ∧
            (∀ a∈set (map fst pre). a < oid) ∧
            (∀ a∈set (map fst suf). oid < a)
    by (metis UnCI map-append set-append snoc spec-ops-rem-last)
```

```
  then show ?case
  proof(cases xi < oid)
    case xi-less: True
    have ∀ x ∈ set (map fst (pre @ suf)). x < xi
      using IH spec-ops-id-inc snoc.prems(1) y-pair by metis
    hence ∀ x ∈ set suf. fst x < xi
      by simp
    hence ∀ x ∈ set suf. fst x < oid
      using xi-less by auto
    hence suf = []
      using IH last-in-set by fastforce
    hence xs @ [x] = (pre @ [(xi, xr)]) @ [] ∧
            (∀ a∈set (map fst ((pre @ [(xi, xr)]))). a < oid) ∧
            (∀ a∈set (map fst []). oid < a)
      by (simp add: IH xi-less y-pair)
    then show ?thesis by force
  next
    case False
    hence oid < xi using snoc.prems(2) y-pair by auto
    hence xs @ [x] = pre @ (suf @ [(xi, xr)]) ∧
            (∀ i ∈ set (map fst pre). i < oid) ∧
            (∀ i ∈ set (map fst (suf @ [(xi, xr)])). oid < i)
      by (simp add: IH y-pair)
    then show ?thesis by blast
  qed
qed

lemma spec-ops-exists-base:
  assumes finite ops
    and ⋀oid op1 op2. (oid, op1) ∈ ops ⟹ (oid, op2) ∈ ops ⟹ op1 = op2
    and ⋀oid oper ref. (oid, oper) ∈ ops ⟹ ref ∈ deps oper ⟹ ref < oid
  shows ∃ op-list. set op-list = ops ∧ spec-ops op-list deps
  using assms proof(induct ops rule: Finite-Set.finite-induct-select)
  case empty
  then show ∃ op-list. set op-list = {} ∧ spec-ops op-list deps
    by (simp add: spec-ops-empty)
next
  case (select subset)
  from this obtain op-list where set op-list = subset and spec-ops op-list deps
    using assms by blast
  moreover obtain oid oper where select: (oid, oper) ∈ ops − subset
    using select.hyps(1) by auto
  moreover from this have ⋀op2. (oid, op2) ∈ ops ⟹ op2 = oper
    using assms(2) by auto
  hence oid ∉ fst ‘ subset
    by (metis (no-types, lifting) DiffD2 select image-iff prod.collapse psubsetD se-
lect.hyps(1))
  from this obtain pre suf
    where op-list = pre @ suf
```

```
    and ∀ i ∈ set (map fst pre). i < oid
    and ∀ i ∈ set (map fst suf). oid < i
  using spec-ops-split calculation by (metis (no-types, lifting) set-map)
 moreover have set (pre @ [(oid, oper)] @ suf) = insert (oid, oper) subset
  using calculation by auto
 moreover have spec-ops (pre @ [(oid, oper)] @ suf) deps
  using calculation spec-ops-add-any assms(3) by (metis DiffD1)
 ultimately show ?case by blast
qed
```

We prove that for any given OpSet, a *spec-ops* linearisation exists:

```
lemma spec-ops-exists:
 assumes opset ops deps
 shows ∃ op-list. set op-list = ops ∧ spec-ops op-list deps
proof −
 have finite ops
  using assms opset.finite-opset by force
 moreover have ⋀oid op1 op2. (oid, op1) ∈ ops ⟹ (oid, op2) ∈ ops ⟹ op1
= op2
  using assms opset.unique-oid by force
 moreover have ⋀oid oper ref. (oid, oper) ∈ ops ⟹ ref ∈ deps oper ⟹ ref <
oid
  using assms opset.ref-older by force
 ultimately show ∃ op-list. set op-list = ops ∧ spec-ops op-list deps
  by (simp add: spec-ops-exists-base)
qed

lemma spec-ops-oid-unique:
 assumes spec-ops op-list deps
  and (oid, op1) ∈ set op-list
  and (oid, op2) ∈ set op-list
 shows op1 = op2
 using assms proof(induction op-list, simp)
 case (Cons x op-list)
 have distinct (map fst (x # op-list))
  using Cons.prems(1) spec-ops-def by blast
 hence notin: fst x ∉ set (map fst op-list)
  by simp
 then show op1 = op2
 proof(cases fst x = oid)
  case True
  then show op1 = op2
   using Cons.prems notin by (metis Pair-inject in-set-zipE set-ConsD zip-map-fst-snd)
 next
  case False
  then have (oid, op1) ∈ set op-list and (oid, op2) ∈ set op-list
   using Cons.prems by auto
  then show op1 = op2
   using Cons.IH Cons.prems(1) spec-ops-rem-cons by blast
```

**qed**
**qed**

Conversely, for any given *spec-ops* list, the set of pairs in the list is an OpSet:

**lemma** *spec-ops-is-opset*:
**assumes** *spec-ops op-list deps*
**shows** *opset* (*set op-list*) *deps*
**proof** −
**have** $\bigwedge$*oid op1 op2*. (*oid*, *op1*) ∈ *set op-list* ⟹ (*oid*, *op2*) ∈ *set op-list* ⟹ *op1* = *op2*
**using** *assms spec-ops-oid-unique* **by** *fastforce*
**moreover have** $\bigwedge$*oid oper ref*. (*oid*, *oper*) ∈ *set op-list* ⟹ *ref* ∈ *deps oper* ⟹ *ref* < *oid*
**by** (*meson assms spec-ops-ref-less*)
**moreover have** *finite* (*set op-list*)
**by** *simp*
**ultimately show** *opset* (*set op-list*) *deps*
**by** (*simp add*: *opset-def*)
**qed**

## 1.4 The *crdt-ops* predicate

Like *spec-ops*, the *crdt-ops* predicate describes the linearisation of an OpSet into a list. Like *spec-ops*, it requires IDs to be unique. However, its other properties are different: *crdt-ops* does not require operations to appear in sorted order, but instead, whenever any operation references the ID of a prior operation, that prior operation must appear previously in the *crdt-ops* list. Thus, the order of operations is partially constrained: operations must appear in causal order, but concurrent operations can be ordered arbitrarily.

This list describes the operation sequence in the order it is typically applied to an operation-based CRDT. Applying operations in the order they appear in *crdt-ops* requires that concurrent operations commute. For any *crdt-ops* operation sequence, there is a permutation that satisfies the *spec-ops* predicate. Thus, to check whether a CRDT satisfies its sequential specification, we can prove that interpreting any *crdt-ops* operation sequence with the commutative operation interpretation results in the same end result as interpreting the *spec-ops* permutation of that operation sequence with the sequential operation interpretation.

**inductive** *crdt-ops* :: ('*oid*::{*linorder*} × '*oper*) *list* ⇒ ('*oper* ⇒ '*oid set*) ⇒ *bool*
**where**
*crdt-ops* [] *deps* |
⟦*crdt-ops xs deps*;
*oid* ∉ *set* (*map fst xs*);
∀ *ref* ∈ *deps oper*. *ref* ∈ *set* (*map fst xs*) ∧ *ref* < *oid*
⟧ ⟹ *crdt-ops* (*xs* @ [(*oid*, *oper*)]) *deps*

**inductive-cases** *crdt-ops-last*: *crdt-ops* (*xs* @ [*x*]) *deps*

**lemma** *crdt-ops-intro*:
**assumes** $\bigwedge r.\ r \in deps\ oper \Longrightarrow r \in fst\ `\ set\ xs \wedge r < oid$
**and** $oid \notin fst\ `\ set\ xs$
**and** *crdt-ops xs deps*
**shows** *crdt-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
**using** *assms crdt-ops.simps* **by** *force*

**lemma** *crdt-ops-rem-last*:
**assumes** *crdt-ops* (*xs* @ [*x*]) *deps*
**shows** *crdt-ops xs deps*
**using** *assms crdt-ops.cases snoc-eq-iff-butlast* **by** *blast*

**lemma** *crdt-ops-ref-less*:
**assumes** *crdt-ops xs deps*
**and** (*oid*, *oper*) ∈ *set xs*
**and** *r* ∈ *deps oper*
**shows** *r* < *oid*
**using** *assms* **by** (*induction rule*: *crdt-ops.induct*, *auto*)

**lemma** *crdt-ops-ref-less-last*:
**assumes** *crdt-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
**and** *r* ∈ *deps oper*
**shows** *r* < *oid*
**using** *assms crdt-ops-ref-less* **by** *fastforce*

**lemma** *crdt-ops-distinct-fst*:
**assumes** *crdt-ops xs deps*
**shows** *distinct* (*map fst xs*)
**using** *assms* **proof** (*induction xs rule*: *List.rev-induct*, *simp*)
**case** (*snoc x xs*)
**hence** *distinct* (*map fst xs*)
**using** *crdt-ops-last* **by** *blast*
**moreover have** *fst x* ∉ *set* (*map fst xs*)
**using** *snoc* **by** (*metis crdt-ops-last fstI image-set*)
**ultimately show** *distinct* (*map fst* (*xs* @ [*x*]))
**by** *simp*
**qed**

**lemma** *crdt-ops-distinct*:
**assumes** *crdt-ops xs deps*
**shows** *distinct xs*
**using** *assms crdt-ops-distinct-fst distinct-map* **by** *blast*

**lemma** *crdt-ops-unique-last*:
**assumes** *crdt-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
**shows** *oid* ∉ *set* (*map fst xs*)
**using** *assms crdt-ops.cases* **by** *blast*

**lemma** *crdt-ops-unique-mid*:
**assumes** *crdt-ops* (*xs* @ [(*oid*, *oper*)] @ *ys*) *deps*
**shows** *oid* ∉ *set* (*map fst xs*) ∧ *oid* ∉ *set* (*map fst ys*)
**using** *assms* **proof**(*induction ys rule*: *rev-induct*)
**case** *Nil*
**then show** *oid* ∉ *set* (*map fst xs*) ∧ *oid* ∉ *set* (*map fst* [])
**by** (*metis crdt-ops-unique-last Nil-is-map-conv append-Nil2 empty-iff empty-set*)
**next**
**case** (*snoc y ys*)
**obtain** *yi yr* **where** *y-pair*: *y* = (*yi*, *yr*)
**by** *fastforce*
**have** *IH*: *oid* ∉ *set* (*map fst xs*) ∧ *oid* ∉ *set* (*map fst ys*)
**using** *crdt-ops-rem-last snoc* **by** (*metis append-assoc*)
**have** (*xs* @ (*oid*, *oper*) # *ys*) @ [(*yi*, *yr*)] = *xs* @ (*oid*, *oper*) # *ys* @ [(*yi*, *yr*)]
**by** *simp*
**hence** *yi* ∉ *set* (*map fst* (*xs* @ (*oid*, *oper*) # *ys*))
**using** *crdt-ops-unique-last* **by** (*metis append-Cons append-self-conv2 snoc.prems y-pair*)
**thus** *oid* ∉ *set* (*map fst xs*) ∧ *oid* ∉ *set* (*map fst* (*ys* @ [*y*]))
**using** *IH y-pair* **by** *auto*
**qed**

**lemma** *crdt-ops-ref-exists*:
**assumes** *crdt-ops* (*pre* @ (*oid*, *oper*) # *suf*) *deps*
**and** *ref* ∈ *deps oper*
**shows** *ref* ∈ *fst* ‘ *set pre*
**using** *assms* **proof**(*induction suf rule*: *List.rev-induct*)
**case** *Nil* **thus** *?case*
**by** (*metis crdt-ops-last prod.sel*(*2*))
**next**
**case** (*snoc x xs*) **thus** *?case*
**using** *crdt-ops.cases* **by** *force*
**qed**

**lemma** *crdt-ops-no-future-ref*:
**assumes** *crdt-ops* (*xs* @ [(*oid*, *oper*)] @ *ys*) *deps*
**shows** ⋀*ref*. *ref* ∈ *deps oper* ⟹ *ref* ∉ *fst* ‘ *set ys*
**proof** −
**from** *assms*(*1*) **have** ⋀*ref*. *ref* ∈ *deps oper* ⟹ *ref* ∈ *set* (*map fst xs*)
**by** (*simp add*: *crdt-ops-ref-exists*)
**moreover have** *distinct* (*map fst* (*xs* @ [(*oid*, *oper*)] @ *ys*))
**using** *assms crdt-ops-distinct-fst* **by** *blast*
**ultimately have** ⋀*ref*. *ref* ∈ *deps oper* ⟹ *ref* ∉ *set* (*map fst* ([(*oid*, *oper*)] @ *ys*))
**using** *distinct-fst-append* **by** *metis*
**thus** ⋀*ref*. *ref* ∈ *deps oper* ⟹ *ref* ∉ *fst* ‘ *set ys*
**by** *simp*
**qed**

**lemma** *crdt-ops-reorder*:
**assumes** *crdt-ops* (*xs* @ [(*oid*, *oper*)] @ *ys*) *deps*
**and** $\bigwedge$*op2 r*. *op2* ∈ *snd* ‘ *set ys* ⟹ *r* ∈ *deps op2* ⟹ *r* ≠ *oid*
**shows** *crdt-ops* (*xs* @ *ys* @ [(*oid*, *oper*)]) *deps*
**using** *assms* **proof**(*induction ys rule*: *rev-induct*)
**case** *Nil*
**then show** *crdt-ops* (*xs* @ [] @ [(*oid*, *oper*)]) *deps*
**using** *crdt-ops-rem-last* **by** *auto*
**next**
**case** (*snoc y ys*)
**then obtain** *yi yo* **where** *y-pair*: *y* = (*yi*, *yo*)
**by** *fastforce*
**have** *IH*: *crdt-ops* (*xs* @ *ys* @ [(*oid*, *oper*)]) *deps*
**proof** −
**have** *crdt-ops* (*xs* @ [(*oid*, *oper*)] @ *ys*) *deps*
**by** (*metis snoc*(*2*) *append.assoc crdt-ops-rem-last*)
**thus** *crdt-ops* (*xs* @ *ys* @ [(*oid*, *oper*)]) *deps*
**using** *snoc.IH snoc.prems*(*2*) **by** *auto*
**qed**
**have** *crdt-ops* (*xs* @ *ys* @ [*y*]) *deps*
**proof** −
**have** *yi* ∉ *fst* ‘ *set* (*xs* @ [(*oid*, *oper*)] @ *ys*)
**by** (*metis y-pair append-assoc crdt-ops-unique-last set-map snoc.prems*(*1*))
**hence** *yi* ∉ *fst* ‘ *set* (*xs* @ *ys*)
**by** *auto*
**moreover have** $\bigwedge$*r*. *r* ∈ *deps yo* ⟹ *r* ∈ *fst* ‘ *set* (*xs* @ *ys*) ∧ *r* < *yi*
**proof** −
**have** $\bigwedge$*r*. *r* ∈ *deps yo* ⟹ *r* ≠ *oid*
**using** *snoc.prems*(*2*) *y-pair* **by** *fastforce*
**moreover have** $\bigwedge$*r*. *r* ∈ *deps yo* ⟹ *r* ∈ *fst* ‘ *set* (*xs* @ [(*oid*, *oper*)] @ *ys*)
**by** (*metis y-pair append-assoc snoc.prems*(*1*) *crdt-ops-ref-exists*)
**moreover have** $\bigwedge$*r*. *r* ∈ *deps yo* ⟹ *r* < *yi*
**using** *crdt-ops-ref-less snoc.prems*(*1*) *y-pair* **by** *fastforce*
**ultimately show** $\bigwedge$*r*. *r* ∈ *deps yo* ⟹ *r* ∈ *fst* ‘ *set* (*xs* @ *ys*) ∧ *r* < *yi*
**by** *simp*
**qed**
**moreover from** *IH* **have** *crdt-ops* (*xs* @ *ys*) *deps*
**using** *crdt-ops-rem-last* **by** *force*
**ultimately show** *crdt-ops* (*xs* @ *ys* @ [*y*]) *deps*
**using** *y-pair crdt-ops-intro* **by** (*metis append.assoc*)
**qed**
**moreover have** *oid* ∉ *fst* ‘ *set* (*xs* @ *ys* @ [*y*])
**using** *crdt-ops-unique-mid* **by** (*metis* (*no-types*, *lifting*) *UnE image-Un*
*image-set set-append snoc.prems*(*1*))
**moreover have** $\bigwedge$*r*. *r* ∈ *deps oper* ⟹ *r* ∈ *fst* ‘ *set* (*xs* @ *ys* @ [*y*])
**using** *crdt-ops-ref-exists*
**by** (*metis UnCI append-Cons image-Un set-append snoc.prems*(*1*))
**moreover have** $\bigwedge$*r*. *r* ∈ *deps oper* ⟹ *r* < *oid*

**using** *IH crdt-ops-ref-less* **by** *fastforce*
**ultimately show** *crdt-ops* (*xs* @ (*ys* @ [*y*]) @ [(*oid*, *oper*)]) *deps*
**using** *crdt-ops-intro* **by** (*metis append-assoc*)
**qed**

**lemma** *crdt-ops-rem-middle*:
**assumes** *crdt-ops* (*xs* @ [(*oid*, *ref*)] @ *ys*) *deps*
**and** ⋀*op2 r*. *op2* ∈ *snd* ‘ *set ys* ⟹ *r* ∈ *deps op2* ⟹ *r* ≠ *oid*
**shows** *crdt-ops* (*xs* @ *ys*) *deps*
**using** *assms crdt-ops-rem-last crdt-ops-reorder append-assoc* **by** *metis*

**lemma** *crdt-ops-independent-suf*:
**assumes** *spec-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
**and** *crdt-ops* (*ys* @ [(*oid*, *oper*)] @ *zs*) *deps*
**and** *set* (*xs* @ [(*oid*, *oper*)]) = *set* (*ys* @ [(*oid*, *oper*)] @ *zs*)
**shows** ⋀*op2 r*. *op2* ∈ *snd* ‘ *set zs* ⟹ *r* ∈ *deps op2* ⟹ *r* ≠ *oid*
**proof** −
**have** ⋀*op2 r*. *op2* ∈ *snd* ‘ *set xs* ⟹ *r* ∈ *deps op2* ⟹ *r* < *oid*
**proof** −
**from** *assms*(*1*) **have** ⋀*i*. *i* ∈ *fst* ‘ *set xs* ⟹ *i* < *oid*
**using** *spec-ops-id-inc* **by** *fastforce*
**moreover have** ⋀*i2 op2 r*. (*i2*, *op2*) ∈ *set xs* ⟹ *r* ∈ *deps op2* ⟹ *r* < *i2*
**using** *assms*(*1*) *spec-ops-ref-less spec-ops-rem-last* **by** *fastforce*
**ultimately show** ⋀*op2 r*. *op2* ∈ *snd* ‘ *set xs* ⟹ *r* ∈ *deps op2* ⟹ *r* < *oid*
**by** *fastforce*
**qed**
**moreover have** *set zs* ⊆ *set xs*
**proof** −
**have** *distinct* (*xs* @ [(*oid*, *oper*)]) **and** *distinct* (*ys* @ [(*oid*, *oper*)] @ *zs*)
**using** *assms spec-ops-distinct crdt-ops-distinct* **by** *blast+*
**hence** *set xs* = *set* (*ys* @ *zs*)
**by** (*meson append-set-rem-last assms*(*3*))
**then show** *set zs* ⊆ *set xs*
**using** *append-subset*(*2*) **by** *simp*
**qed**
**ultimately show** ⋀*op2 r*. *op2* ∈ *snd* ‘ *set zs* ⟹ *r* ∈ *deps op2* ⟹ *r* ≠ *oid*
**by** *fastforce*
**qed**

**lemma** *crdt-ops-reorder-spec*:
**assumes** *spec-ops* (*xs* @ [*x*]) *deps*
**and** *crdt-ops* (*ys* @ [*x*] @ *zs*) *deps*
**and** *set* (*xs* @ [*x*]) = *set* (*ys* @ [*x*] @ *zs*)
**shows** *crdt-ops* (*ys* @ *zs* @ [*x*]) *deps*
**using** *assms* **proof** −
**obtain** *oid oper* **where** *x-pair*: *x* = (*oid*, *oper*) **by** *force*
**hence** ⋀*op2 r*. *op2* ∈ *snd* ‘ *set zs* ⟹ *r* ∈ *deps op2* ⟹ *r* ≠ *oid*
**using** *assms crdt-ops-independent-suf* **by** *fastforce*
**thus** *crdt-ops* (*ys* @ *zs* @ [*x*]) *deps*

**using** *assms*(*2*) *crdt-ops-reorder x-pair* **by** *metis*
**qed**

**lemma** *crdt-ops-rem-spec*:
**assumes** *spec-ops* (*xs* @ [*x*]) *deps*
**and** *crdt-ops* (*ys* @ [*x*] @ *zs*) *deps*
**and** *set* (*xs* @ [*x*]) = *set* (*ys* @ [*x*] @ *zs*)
**shows** *crdt-ops* (*ys* @ *zs*) *deps*
**using** *assms crdt-ops-rem-last crdt-ops-reorder-spec append-assoc* **by** *metis*

**lemma** *crdt-ops-rem-penultimate*:
**assumes** *crdt-ops* (*xs* @ [(*i1*, *r1*)] @ [(*i2*, *r2*)]) *deps*
**and** ⋀*r*. *r* ∈ *deps r2* ⟹ *r* ≠ *i1*
**shows** *crdt-ops* (*xs* @ [(*i2*, *r2*)]) *deps*
**proof** −
**have** *crdt-ops* (*xs* @ [(*i1*, *r1*)]) *deps*
**using** *assms*(*1*) *crdt-ops-rem-last* **by** *force*
**hence** *crdt-ops xs deps*
**using** *crdt-ops-rem-last* **by** *force*
**moreover have** *distinct* (*map fst* (*xs* @ [(*i1*, *r1*)] @ [(*i2*, *r2*)]))
**using** *assms*(*1*) *crdt-ops-distinct-fst* **by** *blast*
**hence** *i2* ∉ *set* (*map fst xs*)
**by** *auto*
**moreover have** *crdt-ops* ((*xs* @ [(*i1*, *r1*)]) @ [(*i2*, *r2*)]) *deps*
**using** *assms*(*1*) **by** *auto*
**hence** ⋀*r*. *r* ∈ *deps r2* ⟹ *r* ∈ *fst* ‘ *set* (*xs* @ [(*i1*, *r1*)])
**using** *crdt-ops-ref-exists* **by** *metis*
**hence** ⋀*r*. *r* ∈ *deps r2* ⟹ *r* ∈ *set* (*map fst xs*)
**using** *assms*(*2*) **by** *auto*
**moreover have** ⋀*r*. *r* ∈ *deps r2* ⟹ *r* < *i2*
**using** *assms*(*1*) *crdt-ops-ref-less* **by** *fastforce*
**ultimately show** *crdt-ops* (*xs* @ [(*i2*, *r2*)]) *deps*
**by** (*simp add*: *crdt-ops-intro*)
**qed**

**lemma** *crdt-ops-spec-ops-exist*:
**assumes** *crdt-ops xs deps*
**shows** ∃ *ys*. *set xs* = *set ys* ∧ *spec-ops ys deps*
**using** *assms* **proof**(*induction xs rule*: *List.rev-induct*)
**case** *Nil*
**then show** ∃ *ys*. *set* [] = *set ys* ∧ *spec-ops ys deps*
**by** (*simp add*: *spec-ops-empty*)
**next**
**case** (*snoc x xs*)
**hence** *IH*: ∃ *ys*. *set xs* = *set ys* ∧ *spec-ops ys deps*
**using** *crdt-ops-rem-last* **by** *blast*
**then obtain** *ys oid ref*
**where** *set xs* = *set ys* **and** *spec-ops ys deps* **and** *x* = (*oid*, *ref*)
**by** *force*

```
  moreover have ∃ pre suf. ys = pre@suf ∧
                    (∀ i ∈ set (map fst pre). i < oid) ∧
                    (∀ i ∈ set (map fst suf). oid < i)
  proof −
    have oid ∉ set (map fst xs)
      using calculation(3) crdt-ops-unique-last snoc.prems by force
    hence oid ∉ set (map fst ys)
      by (simp add: calculation(1))
    thus ?thesis
      using spec-ops-split ‹spec-ops ys deps› by blast
  qed
  from this obtain pre suf where ys = pre @ suf and
    ∀ i ∈ set (map fst pre). i < oid and
    ∀ i ∈ set (map fst suf). oid < i by force
  moreover have set (xs @ [(oid, ref)]) = set (pre @ [(oid, ref)] @ suf)
    using crdt-ops-distinct calculation snoc.prems by simp
  moreover have spec-ops (pre @ [(oid, ref)] @ suf) deps
  proof −
    have ∀ r ∈ deps ref. r < oid
      using calculation(3) crdt-ops-ref-less-last snoc.prems by fastforce
    hence spec-ops (pre @ [(oid, ref)] @ suf) deps
      using spec-ops-add-any calculation by metis
    thus ?thesis by simp
  qed
  ultimately show ∃ ys. set (xs @ [x]) = set ys ∧ spec-ops ys deps
    by blast
qed

end
```

# 2 Specifying list insertion

```
theory Insert-Spec
  imports OpSet
begin
```

In this section we consider only list insertion. We model an insertion operation as a pair (*ID, ref*), where *ref* is either *None* (signifying an insertion at the head of the list) or *Some r* (an insertion immediately after a reference element with ID *r*). If the reference element does not exist, the operation does nothing.

We provide two different definitions of the interpretation function for list insertion: *insert-spec* and *insert-alt*. The *insert-alt* definition matches the paper, while *insert-spec* uses the Isabelle/HOL list datatype, making it more suitable for formal reasoning. In a later subsection we prove that the two definitions are in fact equivalent.

```
fun insert-spec :: 'oid list ⇒ ('oid × 'oid option) ⇒ 'oid list where
```

```
  insert-spec xs     (oid, None)     = oid#xs |
  insert-spec []     (oid, -)        = [] |
  insert-spec (x#xs) (oid, Some ref) =
    (if x = ref then x # oid # xs
                else x # (insert-spec xs (oid, Some ref)))
```

```
fun insert-alt :: ('oid × 'oid option) set ⇒ ('oid × 'oid) ⇒ ('oid × 'oid option) set
where
  insert-alt list-rel (oid, ref) = (
      if ∃ n. (ref, n) ∈ list-rel
      then {(p, n) ∈ list-rel. p ≠ ref} ∪ {(ref, Some oid)} ∪
           {(i, n). i = oid ∧ (ref, n) ∈ list-rel}
      else list-rel)
```

*interp-ins* is the sequential interpretation of a set of insertion operations. It starts with an empty list as initial state, and then applies the operations from left to right.

```
definition interp-ins :: ('oid × 'oid option) list ⇒ 'oid list where
  interp-ins ops ≡ foldl insert-spec [] ops
```

## 2.1 The *insert-ops* predicate

We now specialise the definitions from the abstract OpSet section for list insertion. *insert-opset* is an opset consisting only of insertion operations, and *insert-ops* is the specialisation of the *spec-ops* predicate for insertion operations. We prove several useful lemmas about *insert-ops*.

```
locale insert-opset = opset opset set-option
  for opset :: ('oid::{linorder} × 'oid option) set

definition insert-ops :: ('oid::{linorder} × 'oid option) list ⇒ bool where
  insert-ops list ≡ spec-ops list set-option

lemma insert-ops-NilI [intro!]:
  shows insert-ops []
  by (auto simp add: insert-ops-def spec-ops-def)

lemma insert-ops-rem-last [dest]:
  assumes insert-ops (xs @ [x])
  shows insert-ops xs
  using assms insert-ops-def spec-ops-rem-last by blast

lemma insert-ops-rem-cons:
  assumes insert-ops (x # xs)
  shows insert-ops xs
  using assms insert-ops-def spec-ops-rem-cons by blast

lemma insert-ops-appendD:
  assumes insert-ops (xs @ ys)
```

**shows** *insert-ops xs*
**using** *assms* **by** (*induction ys rule*: *List.rev-induct*,
*auto*, *metis insert-ops-rem-last append-assoc*)

**lemma** *insert-ops-rem-prefix*:
**assumes** *insert-ops* (*pre* @ *suf*)
**shows** *insert-ops suf*
**using** *assms* **proof**(*induction pre*)
**case** *Nil*
**then show** *insert-ops* ([] @ *suf*) ⟹ *insert-ops suf*
**by** *auto*
**next**
**case** (*Cons a pre*)
**have** *sorted* (*map fst suf*)
**using** *assms* **by** (*simp add*: *insert-ops-def sorted-append spec-ops-def*)
**moreover have** *distinct* (*map fst suf*)
**using** *assms* **by** (*simp add*: *insert-ops-def spec-ops-def*)
**ultimately show** *insert-ops* ((*a* # *pre*) @ *suf*) ⟹ *insert-ops suf*
**by** (*simp add*: *insert-ops-def spec-ops-def*)
**qed**

**lemma** *insert-ops-remove1*:
**assumes** *insert-ops xs*
**shows** *insert-ops* (*remove1 x xs*)
**using** *assms insert-ops-def spec-ops-remove1* **by** *blast*

**lemma** *last-op-greatest*:
**assumes** *insert-ops* (*op-list* @ [(*oid*, *oper*)])
**and** *x* ∈ *set* (*map fst op-list*)
**shows** *x* < *oid*
**using** *assms spec-ops-id-inc insert-ops-def* **by** *metis*

**lemma** *insert-ops-ref-older*:
**assumes** *insert-ops* (*pre* @ [(*oid*, *Some ref*)] @ *suf*)
**shows** *ref* < *oid*
**using** *assms* **by** (*auto simp add*: *insert-ops-def spec-ops-def*)

**lemma** *insert-ops-memb-ref-older*:
**assumes** *insert-ops op-list*
**and** (*oid*, *Some ref*) ∈ *set op-list*
**shows** *ref* < *oid*
**using** *assms insert-ops-ref-older split-list-first* **by** *fastforce*

## 2.2 Properties of the *insert-spec* function

**lemma** *insert-spec-none* [*simp*]:
**shows** *set* (*insert-spec xs* (*oid*, *None*)) = *set xs* ∪ {*oid*}
**by** (*induction xs*, *auto simp add*: *insert-commute sup-commute*)

**lemma** *insert-spec-set* [*simp*]:
**assumes** *ref* ∈ *set xs*
**shows** *set* (*insert-spec xs* (*oid*, *Some ref*)) = *set xs* ∪ {*oid*}
**using** *assms* **proof**(*induction xs*)
**assume** *ref* ∈ *set* []
**thus** *set* (*insert-spec* [] (*oid*, *Some ref*)) = *set* [] ∪ {*oid*}
**by** *auto*
**next**
**fix** *a xs*
**assume** *ref* ∈ *set xs* ⟹ *set* (*insert-spec xs* (*oid*, *Some ref*)) = *set xs* ∪ {*oid*}
**and** *ref* ∈ *set* (*a*#*xs*)
**thus** *set* (*insert-spec* (*a*#*xs*) (*oid*, *Some ref*)) = *set* (*a*#*xs*) ∪ {*oid*}
**by**(*cases a* = *ref*, *auto simp add*: *insert-commute sup-commute*)
**qed**

**lemma** *insert-spec-nonex* [*simp*]:
**assumes** *ref* ∉ *set xs*
**shows** *insert-spec xs* (*oid*, *Some ref*) = *xs*
**using** *assms* **proof**(*induction xs*)
**show** *insert-spec* [] (*oid*, *Some ref*) = []
**by** *simp*
**next**
**fix** *a xs*
**assume** *ref* ∉ *set xs* ⟹ *insert-spec xs* (*oid*, *Some ref*) = *xs*
**and** *ref* ∉ *set* (*a*#*xs*)
**thus** *insert-spec* (*a*#*xs*) (*oid*, *Some ref*) = *a*#*xs*
**by**(*cases a* = *ref*, *auto simp add*: *insert-commute sup-commute*)
**qed**

**lemma** *list-greater-non-memb*:
**fixes** *oid* :: *'oid*::{*linorder*}
**assumes** ⋀*x*. *x* ∈ *set xs* ⟹ *x* < *oid*
**and** *oid* ∈ *set xs*
**shows** *False*
**using** *assms* **by** *blast*

**lemma** *inserted-item-ident*:
**assumes** *a* ∈ *set* (*insert-spec xs* (*e*, *i*))
**and** *a* ∉ *set xs*
**shows** *a* = *e*
**using** *assms* **proof**(*induction xs*)
**case** *Nil*
**then show** *a* = *e* **by** (*cases i*, *auto*)
**next**
**case** (*Cons x xs*)
**then show** *a* = *e*
**proof**(*cases i*)
**case** *None*
**then show** *a* = *e* **using** *assms* **by** *auto*

```
  next
    case (Some ref)
    then show a = e using Cons by (case-tac x = ref, auto)
  qed
qed

lemma insert-spec-distinct [intro]:
  fixes oid :: 'oid::{linorder}
  assumes distinct xs
    and ⋀x. x ∈ set xs ⟹ x < oid
    and ref = Some r ⟶ r < oid
  shows distinct (insert-spec xs (oid, ref))
  using assms(1) assms(2) proof(induction xs)
  show distinct (insert-spec [] (oid, ref))
    by(cases ref, auto)
next
  fix a xs
  assume IH: distinct xs ⟹ (⋀x. x ∈ set xs ⟹ x < oid) ⟹ distinct (insert-spec
xs (oid, ref))
    and D: distinct (a#xs)
    and L: ⋀x. x ∈ set (a#xs) ⟹ x < oid
  show distinct (insert-spec (a#xs) (oid, ref))
  proof(cases ref)
    assume ref = None
    thus distinct (insert-spec (a#xs) (oid, ref))
      using D L by auto
  next
    fix id
    assume S: ref = Some id
    {
      assume EQ: a = id
      hence id ≠ oid
        using D L by auto
      moreover have id ∉ set xs
        using D EQ by auto
      moreover have oid ∉ set xs
        using L by auto
      ultimately have id ≠ oid ∧ id ∉ set xs ∧ oid ∉ set xs ∧ distinct xs
        using D by auto
    }
    note T = this
    {
      assume NEQ: a ≠ id
      have 0: a ∉ set (insert-spec xs (oid, Some id))
        using D L by(metis distinct.simps(2) insert-spec.simps(1) insert-spec-none
insert-spec-nonex
            insert-spec-set insert-iff list.set(2) not-less-iff-gr-or-eq)
      have 1: distinct xs
        using D by auto
```

```
      have ⋀x. x ∈ set xs ⟹ x < oid
        using L by auto
      hence distinct (insert-spec xs (oid, Some id))
        using S IH[OF 1] by blast
      hence a ∉ set (insert-spec xs (oid, Some id)) ∧ distinct (insert-spec xs (oid,
Some id))
        using 0 by auto
    }
    from this S T show distinct (insert-spec (a # xs) (oid, ref))
      by clarsimp
  qed
qed

lemma insert-after-ref:
  assumes distinct (xs @ ref # ys)
  shows insert-spec (xs @ ref # ys) (oid, Some ref) = xs @ ref # oid # ys
  using assms by (induction xs, auto)

lemma insert-somewhere:
  assumes ref = None ∨ (ref = Some r ∧ r ∈ set list)
  shows ∃ xs ys. list = xs @ ys ∧ insert-spec list (oid, ref) = xs @ oid # ys
  using assms proof(induction list)
  assume ref = None ∨ ref = Some r ∧ r ∈ set []
  thus ∃ xs ys. [] = xs @ ys ∧ insert-spec [] (oid, ref) = xs @ oid # ys
  proof
    assume ref = None
    thus ∃ xs ys. [] = xs @ ys ∧ insert-spec [] (oid, ref) = xs @ oid # ys
      by auto
  next
    assume ref = Some r ∧ r ∈ set []
    thus ∃ xs ys. [] = xs @ ys ∧ insert-spec [] (oid, ref) = xs @ oid # ys
      by auto
  qed
next
  fix a list
  assume 1: ref = None ∨ ref = Some r ∧ r ∈ set (a#list)
    and IH: ref = None ∨ ref = Some r ∧ r ∈ set list ⟹
        ∃ xs ys. list = xs @ ys ∧ insert-spec list (oid, ref) = xs @ oid # ys
  show ∃ xs ys. a # list = xs @ ys ∧ insert-spec (a # list) (oid, ref) = xs @ oid
# ys
  proof(rule disjE[OF 1])
    assume ref = None
    thus ∃ xs ys. a # list = xs @ ys ∧ insert-spec (a # list) (oid, ref) = xs @ oid
# ys
      by force
  next
    assume ref = Some r ∧ r ∈ set (a # list)
    hence 2: r = a ∨ r ∈ set list and 3: ref = Some r
      by auto
```

**show** $\exists xs\ ys.\ a \# list = xs @ ys \wedge insert\text{-}spec\ (a \# list)\ (oid, ref) = xs @ oid \# ys$
**proof**(*rule disjE*[*OF 2*])
**assume** $r = a$
**thus** $\exists xs\ ys.\ a \# list = xs @ ys \wedge insert\text{-}spec\ (a \# list)\ (oid, ref) = xs @ oid \# ys$
**using** *3* **by**(*metis append-Cons append-Nil insert-spec.simps*(*3*))
**next**
**assume** $r \in set\ list$
**from** *this* **obtain** *xs ys*
**where** $list = xs @ ys \wedge insert\text{-}spec\ list\ (oid, ref) = xs @ oid \# ys$
**using** *IH 3* **by** *auto*
**thus** $\exists xs\ ys.\ a \# list = xs @ ys \wedge insert\text{-}spec\ (a \# list)\ (oid, ref) = xs @ oid \# ys$
**using** *3* **by** *clarsimp* (*metis append-Cons append-Nil*)
**qed**
**qed**
**qed**

**lemma** *insert-first-part*:
**assumes** $ref = None \vee (ref = Some\ r \wedge r \in set\ xs)$
**shows** $insert\text{-}spec\ (xs @ ys)\ (oid, ref) = (insert\text{-}spec\ xs\ (oid, ref)) @ ys$
**using** *assms* **proof**(*induction ys rule*: *rev-induct*)
**assume** $ref = None \vee ref = Some\ r \wedge r \in set\ xs$
**thus** $insert\text{-}spec\ (xs @ [])\ (oid, ref) = insert\text{-}spec\ xs\ (oid, ref) @ []$
**by** *auto*
**next**
**fix** *x xsa*
**assume** *IH*: $ref = None \vee ref = Some\ r \wedge r \in set\ xs \Longrightarrow insert\text{-}spec\ (xs @ xsa)\ (oid, ref) = insert\text{-}spec\ xs\ (oid, ref) @ xsa$
**and** $ref = None \vee ref = Some\ r \wedge r \in set\ xs$
**thus** $insert\text{-}spec\ (xs @ xsa @ [x])\ (oid, ref) = insert\text{-}spec\ xs\ (oid, ref) @ xsa @ [x]$
**proof**(*induction xs*)
**assume** $ref = None \vee ref = Some\ r \wedge r \in set\ []$
**thus** $insert\text{-}spec\ ([] @ xsa @ [x])\ (oid, ref) = insert\text{-}spec\ []\ (oid, ref) @ xsa @ [x]$
**by** *auto*
**next**
**fix** *a xs*
**assume** *1*: $ref = None \vee ref = Some\ r \wedge r \in set\ (a \# xs)$
**and** *2*: $((ref = None \vee ref = Some\ r \wedge r \in set\ xs \Longrightarrow insert\text{-}spec\ (xs @ xsa)\ (oid, ref) = insert\text{-}spec\ xs\ (oid, ref) @ xsa) \Longrightarrow$
$ref = None \vee ref = Some\ r \wedge r \in set\ xs \Longrightarrow insert\text{-}spec\ (xs @ xsa @ [x])\ (oid, ref) = insert\text{-}spec\ xs\ (oid, ref) @ xsa @ [x])$
**and** *3*: $(ref = None \vee ref = Some\ r \wedge r \in set\ (a \# xs) \Longrightarrow insert\text{-}spec\ ((a \# xs) @ xsa)\ (oid, ref) = insert\text{-}spec\ (a \# xs)\ (oid, ref) @ xsa)$
**show** $insert\text{-}spec\ ((a \# xs) @ xsa @ [x])\ (oid, ref) = insert\text{-}spec\ (a \# xs)\ (oid, ref) @ xsa @ [x]$

**proof**(*rule disjE*[*OF 1*])
**assume** *ref = None*
**thus** *insert-spec* ((*a # xs*) @ *xsa* @ [*x*]) (*oid*, *ref*) = *insert-spec* (*a # xs*) (*oid*, *ref*) @ *xsa* @ [*x*]
**by** *auto*
**next**
**assume** *ref = Some r* ∧ *r* ∈ *set* (*a # xs*)
**thus** *insert-spec* ((*a # xs*) @ *xsa* @ [*x*]) (*oid*, *ref*) = *insert-spec* (*a # xs*) (*oid*, *ref*) @ *xsa* @ [*x*]
**using** *2 3* **by** *auto*
**qed**
**qed**
**qed**

**lemma** *insert-second-part*:
**assumes** *ref = Some r*
**and** *r* ∉ *set xs*
**and** *r* ∈ *set ys*
**shows** *insert-spec* (*xs* @ *ys*) (*oid*, *ref*) = *xs* @ (*insert-spec ys* (*oid*, *ref*))
**using** *assms* **proof**(*induction xs*)
**assume** *ref = Some r*
**thus** *insert-spec* ([] @ *ys*) (*oid*, *ref*) = [] @ *insert-spec ys* (*oid*, *ref*)
**by** *auto*
**next**
**fix** *a xs*
**assume** *ref = Some r* **and** *r* ∉ *set* (*a # xs*) **and** *r* ∈ *set ys*
**and** *ref = Some r* ⟹ *r* ∉ *set xs* ⟹ *r* ∈ *set ys* ⟹ *insert-spec* (*xs* @ *ys*) (*oid*, *ref*) = *xs* @ *insert-spec ys* (*oid*, *ref*)
**thus** *insert-spec* ((*a # xs*) @ *ys*) (*oid*, *ref*) = (*a # xs*) @ *insert-spec ys* (*oid*, *ref*)
**by** *auto*
**qed**

## 2.3 Properties of the *interp-ins* function

**lemma** *interp-ins-empty* [*simp*]:
**shows** *interp-ins* [] = []
**by** (*simp add*: *interp-ins-def*)

**lemma** *interp-ins-tail-unfold*:
**shows** *interp-ins* (*xs* @ [*x*]) = *insert-spec* (*interp-ins xs*) *x*
**by** (*clarsimp simp add*: *interp-ins-def*)

**lemma** *interp-ins-subset* [*simp*]:
**shows** *set* (*interp-ins op-list*) ⊆ *set* (*map fst op-list*)
**proof**(*induction op-list rule*: *List.rev-induct*)
**case** *Nil*
**then show** *set* (*interp-ins* []) ⊆ *set* (*map fst* [])
**by** (*simp add*: *interp-ins-def*)
**next**

```
  case (snoc x xs)
  hence IH: set (interp-ins xs) ⊆ set (map fst xs)
    using interp-ins-def by blast
  obtain oid ref where x-pair: x = (oid, ref)
    by fastforce
  hence spec: interp-ins (xs @ [x]) = insert-spec (interp-ins xs) (oid, ref)
    by (simp add: interp-ins-def)
  then show set (interp-ins (xs @ [x])) ⊆ set (map fst (xs @ [x]))
  proof(cases ref)
    case None
    then show set (interp-ins (xs @ [x])) ⊆ set (map fst (xs @ [x]))
      using IH spec x-pair by auto
  next
    case (Some a)
    then show set (interp-ins (xs @ [x])) ⊆ set (map fst (xs @ [x]))
      using IH spec x-pair by (cases a ∈ set (interp-ins xs), auto)
  qed
qed

lemma interp-ins-distinct:
  assumes insert-ops op-list
  shows distinct (interp-ins op-list)
  using assms proof(induction op-list rule: rev-induct)
  case Nil
  then show distinct (interp-ins [])
    by (simp add: interp-ins-def)
next
  case (snoc x xs)
  hence IH: distinct (interp-ins xs) by blast
  obtain oid ref where x-pair: x = (oid, ref) by force
  hence ∀ x ∈ set (map fst xs). x < oid
    using last-op-greatest snoc.prems by blast
  hence ∀ x ∈ set (interp-ins xs). x < oid
    using interp-ins-subset by fastforce
  hence distinct (insert-spec (interp-ins xs) (oid, ref))
    using IH insert-spec-distinct insert-spec-nonex by metis
  then show distinct (interp-ins (xs @ [x]))
    by (simp add: x-pair interp-ins-tail-unfold)
qed
```

## 2.4 Equivalence of the two definitions of insertion

At the beginning of this section we gave two different definitions of interpretation functions for list insertion: *insert-spec* and *insert-alt*. In this section we prove that the two are equivalent.

We first define how to derive the successor relation from an Isabelle list. This relation contains (*id*, *None*) if *id* is the last element of the list, and (*id1*, *id2*) if *id1* is immediately followed by *id2* in the list.

```
fun succ-rel :: 'oid list ⇒ ('oid × 'oid option) set where
  succ-rel [] = {} |
  succ-rel [head] = {(head, None)} |
  succ-rel (head#x#xs) = {(head, Some x)} ∪ succ-rel (x#xs)
```

*interp-alt* is the equivalent of *interp-ins*, but using *insert-alt* instead of *insert-spec*. To match the paper, it uses a distinct head element to refer to the beginning of the list.

```
definition interp-alt :: 'oid ⇒ ('oid × 'oid option) list ⇒ ('oid × 'oid option) set
where
  interp-alt head ops ≡ foldl insert-alt {(head, None)}
    (map (λx. case x of
          (oid, None)     ⇒ (oid, head) |
          (oid, Some ref) ⇒ (oid, ref))
     ops)

lemma succ-rel-set-fst:
  shows fst ` (succ-rel xs) = set xs
  by (induction xs rule: succ-rel.induct, auto)

lemma succ-rel-functional:
  assumes (a, b1) ∈ succ-rel xs
    and (a, b2) ∈ succ-rel xs
    and distinct xs
  shows b1 = b2
  using assms proof(induction xs rule: succ-rel.induct)
  case 1
  then show ?case by simp
next
  case (2 head)
  then show ?case by simp
next
  case (3 head x xs)
  then show ?case
  proof(cases a = head)
    case True
    hence a ∉ set (x#xs)
      using 3 by auto
    hence a ∉ fst ` (succ-rel (x#xs))
      using succ-rel-set-fst by metis
    then show b1 = b2
      using 3 image-iff by fastforce
  next
    case False
    hence {(a, b1), (a, b2)} ⊆ succ-rel (x#xs)
      using 3 by auto
    moreover have distinct (x#xs)
      using 3 by auto
    ultimately show b1 = b2
```

```
    using 3.IH by auto
  qed
qed

lemma succ-rel-rem-head:
  assumes distinct (x # xs)
  shows {(p, n) ∈ succ-rel (x # xs). p ≠ x} = succ-rel xs
proof −
  have head-notin: x ∉ fst ‘ succ-rel xs
    using assms by (simp add: succ-rel-set-fst)
  moreover obtain y where (x, y) ∈ succ-rel (x # xs)
    by (cases xs, auto)
  moreover have succ-rel (x # xs) = {(x, y)} ∪ succ-rel xs
    using calculation head-notin image-iff by (cases xs, fastforce+)
  moreover from this have ⋀n. (x, n) ∈ succ-rel (x # xs) ⟹ n = y
    by (metis Pair-inject fst-conv head-notin image-eqI insertE insert-is-Un)
  hence {(p, n) ∈ succ-rel (x # xs). p ≠ x} = succ-rel (x # xs) − {(x, y)}
    by blast
  moreover have succ-rel (x # xs) − {(x, y)} = succ-rel xs
    using image-iff calculation by fastforce
  ultimately show {(p, n) ∈ succ-rel (x # xs). p ≠ x} = succ-rel xs
    by simp
qed

lemma succ-rel-swap-head:
  assumes distinct (ref # list)
    and (ref, n) ∈ succ-rel (ref # list)
  shows succ-rel (oid # list) = {(oid, n)} ∪ succ-rel list
proof(cases list)
  case Nil
  then show ?thesis using assms by auto
next
  case (Cons a list)
  moreover from this have n = Some a
    by (metis Un-iff assms singletonI succ-rel.simps(3) succ-rel-functional)
  ultimately show ?thesis by simp
qed

lemma succ-rel-insert-alt:
  assumes a ≠ ref
    and distinct (oid # a # b # list)
  shows insert-alt (succ-rel (a # b # list)) (oid, ref) =
        {(a, Some b)} ∪ insert-alt (succ-rel (b # list)) (oid, ref)
proof(cases ∃ n. (ref, n) ∈ succ-rel (a # b # list))
  case True
  hence insert-alt (succ-rel (a # b # list)) (oid, ref) =
          {(p, n) ∈ succ-rel (a # b # list). p ≠ ref} ∪ {(ref, Some oid)} ∪
          {(i, n). i = oid ∧ (ref, n) ∈ succ-rel (a # b # list)}
    by simp
```

```
  moreover have {(p, n) ∈ succ-rel (a # b # list). p ≠ ref} =
                {(a, Some b)} ∪ {(p, n) ∈ succ-rel (b # list). p ≠ ref}
    using assms(1) by auto
  moreover have insert-alt (succ-rel (b # list)) (oid, ref) =
          {(p, n) ∈ succ-rel (b # list). p ≠ ref} ∪ {(ref, Some oid)} ∪
          {(i, n). i = oid ∧ (ref, n) ∈ succ-rel (b # list)}
  proof −
    have ∃ n. (ref, n) ∈ succ-rel (b # list)
      using assms(1) True by auto
    thus ?thesis by simp
  qed
  moreover have {(i, n). i = oid ∧ (ref, n) ∈ succ-rel (a # b # list)} =
                {(i, n). i = oid ∧ (ref, n) ∈ succ-rel (b # list)}
    using assms(1) by auto
  ultimately show ?thesis by simp
next
  case False
  then show ?thesis by auto
qed

lemma succ-rel-insert-head:
  assumes distinct (ref # list)
  shows succ-rel (insert-spec (ref # list) (oid, Some ref)) =
        insert-alt (succ-rel (ref # list)) (oid, ref)
proof −
  obtain n where ref-in-rel: (ref, n) ∈ succ-rel (ref # list)
    by (cases list, auto)
  moreover from this have {(p, n) ∈ succ-rel (ref # list). p ≠ ref} = succ-rel list
    using assms succ-rel-rem-head by (metis (mono-tags, lifting))
  moreover have {(i, n). i = oid ∧ (ref, n) ∈ succ-rel (ref # list)} = {(oid, n)}
  proof −
    have ⋀nx. (ref, nx) ∈ succ-rel (ref # list) ⟹ nx = n
      using assms by (simp add: succ-rel-functional ref-in-rel)
    hence {(i, n) ∈ succ-rel (ref # list). i = ref} ⊆ {(ref, n)}
      by blast
    moreover have {(ref, n)} ⊆ {(i, n) ∈ succ-rel (ref # list). i = ref}
      by (simp add: ref-in-rel)
    ultimately show ?thesis by blast
  qed
  moreover have insert-alt (succ-rel (ref # list)) (oid, ref) =
                  {(p, n) ∈ succ-rel (ref # list). p ≠ ref} ∪ {(ref, Some oid)} ∪
                  {(i, n). i = oid ∧ (ref, n) ∈ succ-rel (ref # list)}
  proof −
    have ∃ n. (ref, n) ∈ succ-rel (ref # list)
      using ref-in-rel by blast
    thus ?thesis by simp
  qed
  ultimately have insert-alt (succ-rel (ref # list)) (oid, ref) =
                  succ-rel list ∪ {(ref, Some oid)} ∪ {(oid, n)}
```

**by** *simp*
**moreover have** *succ-rel* (*oid* # *list*) = {(*oid*, *n*)} ∪ *succ-rel list*
**using** *assms ref-in-rel succ-rel-swap-head* **by** *metis*
**hence** *succ-rel* (*ref* # *oid* # *list*) = {(*ref*, *Some oid*), (*oid*, *n*)} ∪ *succ-rel list*
**by** *auto*
**ultimately show** *succ-rel* (*insert-spec* (*ref* # *list*) (*oid*, *Some ref*)) =
*insert-alt* (*succ-rel* (*ref* # *list*)) (*oid*, *ref*)
**by** *auto*
**qed**

**lemma** *succ-rel-insert-later*:
**assumes** *succ-rel* (*insert-spec* (*b* # *list*) (*oid*, *Some ref*)) =
*insert-alt* (*succ-rel* (*b* # *list*)) (*oid*, *ref*)
**and** *a* ≠ *ref*
**and** *distinct* (*a* # *b* # *list*)
**shows** *succ-rel* (*insert-spec* (*a* # *b* # *list*) (*oid*, *Some ref*)) =
*insert-alt* (*succ-rel* (*a* # *b* # *list*)) (*oid*, *ref*)
**proof** −
**have** *succ-rel* (*a* # *b* # *list*) = {(*a*, *Some b*)} ∪ *succ-rel* (*b* # *list*)
**by** *simp*
**moreover have** *insert-spec* (*a* # *b* # *list*) (*oid*, *Some ref*) =
*a* # (*insert-spec* (*b* # *list*) (*oid*, *Some ref*))
**using** *assms*(*2*) **by** *simp*
**hence** *succ-rel* (*insert-spec* (*a* # *b* # *list*) (*oid*, *Some ref*)) =
{(*a*, *Some b*)} ∪ *succ-rel* (*insert-spec* (*b* # *list*) (*oid*, *Some ref*))
**by** *auto*
**hence** *succ-rel* (*insert-spec* (*a* # *b* # *list*) (*oid*, *Some ref*)) =
{(*a*, *Some b*)} ∪ *insert-alt* (*succ-rel* (*b* # *list*)) (*oid*, *ref*)
**using** *assms*(*1*) **by** *auto*
**moreover have** *insert-alt* (*succ-rel* (*a* # *b* # *list*)) (*oid*, *ref*) =
{(*a*, *Some b*)} ∪ *insert-alt* (*succ-rel* (*b* # *list*)) (*oid*, *ref*)
**using** *succ-rel-insert-alt assms*(*2*) **by** *auto*
**ultimately show** *?thesis* **by** *blast*
**qed**

**lemma** *succ-rel-insert-Some*:
**assumes** *distinct list*
**shows** *succ-rel* (*insert-spec list* (*oid*, *Some ref*)) = *insert-alt* (*succ-rel list*) (*oid*, *ref*)
**using** *assms* **proof**(*induction list*)
**case** *Nil*
**then show** *succ-rel* (*insert-spec* [] (*oid*, *Some ref*)) = *insert-alt* (*succ-rel* []) (*oid*, *ref*)
**by** *simp*
**next**
**case** (*Cons a list*)
**hence** *distinct* (*a* # *list*)
**by** *simp*
**then show** *succ-rel* (*insert-spec* (*a* # *list*) (*oid*, *Some ref*)) =

```
          insert-alt (succ-rel (a # list)) (oid, ref)
  proof(cases a = ref)
    case True
    then show ?thesis
      using succ-rel-insert-head ‹distinct (a # list)› by metis
  next
    case False
    hence a ≠ ref by simp
    moreover have succ-rel (insert-spec list (oid, Some ref)) =
                  insert-alt (succ-rel list) (oid, ref)
      using Cons.IH Cons.prems by auto
    ultimately show succ-rel (insert-spec (a # list) (oid, Some ref)) =
                    insert-alt (succ-rel (a # list)) (oid, ref)
      by (cases list, force, metis Cons.prems succ-rel-insert-later)
  qed
qed
```

The main result of this section, that *insert-spec* and *insert-alt* are equivalent.

```
theorem insert-alt-equivalent:
  assumes insert-ops ops
    and head ∉ fst ‘ set ops
    and ⋀r. Some r ∈ snd ‘ set ops ⟹ r ≠ head
  shows succ-rel (head # interp-ins ops) = interp-alt head ops
  using assms proof(induction ops rule: List.rev-induct)
  case Nil
  then show succ-rel (head # interp-ins []) = interp-alt head []
    by (simp add: interp-ins-def interp-alt-def)
next
  case (snoc x xs)
  have IH: succ-rel (head # interp-ins xs) = interp-alt head xs
    using snoc by auto
  have distinct-list: distinct (head # interp-ins xs)
  proof −
    have distinct (interp-ins xs)
      using interp-ins-distinct snoc.prems(1) by blast
    moreover have set (interp-ins xs) ⊆ fst ‘ set xs
      using interp-ins-subset snoc.prems(1) by fastforce
    ultimately show distinct (head # interp-ins xs)
      using snoc.prems(2) by auto
  qed
  obtain oid r where x-pair: x = (oid, r) by force
  then show succ-rel (head # interp-ins (xs @ [x])) = interp-alt head (xs @ [x])
  proof(cases r)
    case None
    have interp-alt head (xs @ [x]) = insert-alt (interp-alt head xs) (oid, head)
      by (simp add: interp-alt-def None x-pair)
    moreover have ... = insert-alt (succ-rel (head # interp-ins xs)) (oid, head)
      by (simp add: IH)
    moreover have ... = succ-rel (insert-spec (head # interp-ins xs) (oid, Some
```

*head*))
**using** *distinct-list succ-rel-insert-Some* **by** *metis*
**moreover have** ... = *succ-rel* (*head* # (*insert-spec* (*interp-ins xs*) (*oid*, *None*)))
**by** *auto*
**moreover have** ... = *succ-rel* (*head* # (*interp-ins* (*xs* @ [*x*])))
**by** (*simp add*: *interp-ins-tail-unfold None x-pair*)
**ultimately show** *?thesis* **by** *simp*
**next**
**case** (*Some ref*)
**have** *ref* ≠ *head*
**by** (*simp add*: *Some snoc.prems*(*3*) *x-pair*)
**have** *interp-alt head* (*xs* @ [*x*]) = *insert-alt* (*interp-alt head xs*) (*oid*, *ref*)
**by** (*simp add*: *interp-alt-def Some x-pair*)
**moreover have** ... = *insert-alt* (*succ-rel* (*head* # *interp-ins xs*)) (*oid*, *ref*)
**by** (*simp add*: *IH*)
**moreover have** ... = *succ-rel* (*insert-spec* (*head* # *interp-ins xs*) (*oid*, *Some ref*))
**using** *distinct-list succ-rel-insert-Some* **by** *metis*
**moreover have** ... = *succ-rel* (*head* # (*insert-spec* (*interp-ins xs*) (*oid*, *Some ref*)))
**using** ‹*ref* ≠ *head*› **by** *auto*
**moreover have** ... = *succ-rel* (*head* # (*interp-ins* (*xs* @ [*x*])))
**by** (*simp add*: *interp-ins-tail-unfold Some x-pair*)
**ultimately show** *?thesis* **by** *simp*
**qed**
**qed**

## 2.5 The *list-order* predicate

*list-order ops x y* holds iff, after interpreting the list of insertion operations *ops*, the list element with ID *x* appears before the list element with ID *y* in the resulting list. We prove several lemmas about this predicate; in particular, that executing additional insertion operations does not change the relative ordering of existing list elements.

**definition** *list-order* :: ('*oid*::{*linorder*} × '*oid option*) *list* ⇒ '*oid* ⇒ '*oid* ⇒ *bool*
**where**
*list-order ops x y* ≡ ∃ *xs ys zs*. *interp-ins ops* = *xs* @ [*x*] @ *ys* @ [*y*] @ *zs*

**lemma** *list-orderI*:
**assumes** *interp-ins ops* = *xs* @ [*x*] @ *ys* @ [*y*] @ *zs*
**shows** *list-order ops x y*
**using** *assms* **by** (*auto simp add*: *list-order-def*)

**lemma** *list-orderE*:
**assumes** *list-order ops x y*
**shows** ∃ *xs ys zs*. *interp-ins ops* = *xs* @ [*x*] @ *ys* @ [*y*] @ *zs*
**using** *assms* **by** (*auto simp add*: *list-order-def*)

**lemma** *list-order-memb1*:
**assumes** *list-order ops x y*
**shows** $x \in$ *set* (*interp-ins ops*)
**using** *assms* **by** (*auto simp add*: *list-order-def*)

**lemma** *list-order-memb2*:
**assumes** *list-order ops x y*
**shows** $y \in$ *set* (*interp-ins ops*)
**using** *assms* **by** (*auto simp add*: *list-order-def*)

**lemma** *list-order-trans*:
**assumes** *insert-ops op-list*
**and** *list-order op-list x y*
**and** *list-order op-list y z*
**shows** *list-order op-list x z*
**proof** −
**obtain** *xxs xys xzs* **where** *1*: *interp-ins op-list* = (*xxs*@[*x*]@*xys*)@(*y*#*xzs*)
**using** *assms* **by** (*auto simp add*: *list-order-def interp-ins-def*)
**obtain** *yxs yys yzs* **where** *2*: *interp-ins op-list* = *yxs*@*y*#(*yys*@[*z*]@*yzs*)
**using** *assms* **by** (*auto simp add*: *list-order-def interp-ins-def*)
**have** *3*: *distinct* (*interp-ins op-list*)
**using** *assms interp-ins-distinct* **by** *blast*
**hence** *xzs* = *yys*@[*z*]@*yzs*
**using** *distinct-list-split*[*OF 3*, *OF 2*, *OF 1*] **by** *auto*
**hence** *interp-ins op-list* = *xxs*@[*x*]@*xys*@[*y*]@*yys*@[*z*]@*yzs*
**using** *1 2 3* **by** *clarsimp*
**thus** *list-order op-list x z*
**using** *assms* **by** (*metis append.assoc list-orderI*)
**qed**

**lemma** *insert-preserves-order*:
**assumes** *insert-ops ops* **and** *insert-ops rest*
**and** *rest* = *before* @ *after*
**and** *ops* = *before* @ (*oid*, *ref*) # *after*
**shows** ∃ *xs ys zs*. *interp-ins rest* = *xs* @ *zs* ∧ *interp-ins ops* = *xs* @ *ys* @ *zs*
**using** *assms* **proof**(*induction after arbitrary*: *rest ops rule*: *List.rev-induct*)
**case** *Nil*
**then have** *1*: *interp-ins ops* = *insert-spec* (*interp-ins before*) (*oid*, *ref*)
**by** (*simp add*: *interp-ins-tail-unfold*)
**then show** ∃ *xs ys zs*. *interp-ins rest* = *xs* @ *zs* ∧ *interp-ins ops* = *xs* @ *ys* @ *zs*
**proof**(*cases ref*)
**case** *None*
**hence** *interp-ins rest* = [] @ (*interp-ins before*) ∧
*interp-ins ops* = [] @ [*oid*] @ (*interp-ins before*)
**using** *1 Nil.prems*(*3*) **by** *simp*
**then show** *?thesis* **by** *blast*
**next**
**case** (*Some a*)
**then show** *?thesis*

```
    proof(cases a ∈ set (interp-ins before))
      case True
      then obtain xs ys where interp-ins before = xs @ ys ∧
          insert-spec (interp-ins before) (oid, ref) = xs @ oid # ys
        using insert-somewhere Some by metis
      hence interp-ins rest = xs @ ys ∧ interp-ins ops = xs @ [oid] @ ys
        using 1 Nil.prems(3) by auto
      then show ?thesis by blast
    next
      case False
      hence interp-ins ops = (interp-ins rest) @ [] @ []
        using insert-spec-nonex 1 Nil.prems(3) Some by simp
      then show ?thesis by blast
    qed
  qed
next
  case (snoc oper op-list)
  then have insert-ops ((before @ (oid, ref) # op-list) @ [oper])
    and insert-ops ((before @ op-list) @ [oper])
    by auto
  then have ops1: insert-ops (before @ op-list)
    and ops2: insert-ops (before @ (oid, ref) # op-list)
    using insert-ops-appendD by blast+
  then obtain xs ys zs where IH1: interp-ins (before @ op-list) = xs @ zs
    and IH2: interp-ins (before @ (oid, ref) # op-list) = xs @ ys @ zs
    using snoc.IH by blast
  obtain i2 r2 where oper = (i2, r2) by force
  then show ∃xs ys zs. interp-ins rest = xs @ zs ∧ interp-ins ops = xs @ ys @ zs
  proof(cases r2)
    case None
    hence interp-ins (before @ op-list @ [oper]) = (i2 # xs) @ zs
     by (metis IH1 ‹oper = (i2, r2)› append.assoc append-Cons insert-spec.simps(1)
         interp-ins-tail-unfold)
    moreover have interp-ins (before @ (oid, ref) # op-list @ [oper]) = (i2 # xs)
@ ys @ zs
     by (metis IH2 None ‹oper = (i2, r2)› append.assoc append-Cons insert-spec.simps(1)
         interp-ins-tail-unfold)
    ultimately show ?thesis
      using snoc.prems(3) snoc.prems(4) by blast
  next
    case (Some r)
    then have 1: interp-ins (before @ (oid, ref) # op-list @ [(i2, r2)]) =
                 insert-spec (xs @ ys @ zs) (i2, Some r)
      by (metis IH2 append.assoc append-Cons interp-ins-tail-unfold)
     have 2: interp-ins (before @ op-list @ [(i2, r2)]) = insert-spec (xs @ zs) (i2,
Some r)
      by (metis IH1 append.assoc interp-ins-tail-unfold Some)
    consider (r-xs) r ∈ set xs | (r-ys) r ∈ set ys | (r-zs) r ∈ set zs |
      (r-nonex) r ∉ set (xs @ ys @ zs)
```

```
    by auto
  then show ∃xs ys zs. interp-ins rest = xs @ zs ∧ interp-ins ops = xs @ ys @ zs
  proof(cases)
    case r-xs
    from this have insert-spec (xs @ ys @ zs) (i2, Some r) =
                   (insert-spec xs (i2, Some r)) @ ys @ zs
      by (meson insert-first-part)
   moreover have insert-spec (xs @ zs) (i2, Some r) = (insert-spec xs (i2, Some
r)) @ zs
      by (meson r-xs insert-first-part)
    ultimately show ?thesis
      using 1 2 ‹oper = (i2, r2)› snoc.prems by auto
  next
    case r-ys
    hence r ∉ set xs and r ∉ set zs
      using IH2 ops2 interp-ins-distinct by force+
    moreover from this have insert-spec (xs @ ys @ zs) (i2, Some r) =
                            xs @ (insert-spec ys (i2, Some r)) @ zs
      using insert-first-part insert-second-part insert-spec-nonex
      by (metis Some UnE r-ys set-append)
    moreover have insert-spec (xs @ zs) (i2, Some r) = xs @ zs
      by (simp add: Some calculation(1) calculation(2))
    ultimately show ?thesis
      using 1 2 ‹oper = (i2, r2)› snoc.prems by auto
  next
    case r-zs
    hence r ∉ set xs and r ∉ set ys
      using IH2 ops2 interp-ins-distinct by force+
    moreover from this have insert-spec (xs @ ys @ zs) (i2, Some r) =
                            xs @ ys @ (insert-spec zs (i2, Some r))
      by (metis Some UnE insert-second-part insert-spec-nonex set-append)
   moreover have insert-spec (xs @ zs) (i2, Some r) = xs @ (insert-spec zs (i2,
Some r))
      by (simp add: r-zs calculation(1) insert-second-part)
    ultimately show ?thesis
      using 1 2 ‹oper = (i2, r2)› snoc.prems by auto
  next
    case r-nonex
    then have insert-spec (xs @ ys @ zs) (i2, Some r) = xs @ ys @ zs
      by simp
    moreover have insert-spec (xs @ zs) (i2, Some r) = xs @ zs
      using r-nonex by simp
    ultimately show ?thesis
      using 1 2 ‹oper = (i2, r2)› snoc.prems by auto
  qed
 qed
qed

lemma distinct-fst:
```

```
  assumes distinct (map fst A)
  shows distinct A
  using assms by (induction A) auto

lemma subset-distinct-le:
  assumes set A ⊆ set B and distinct A and distinct B
  shows length A ≤ length B
  using assms proof(induction B arbitrary: A)
  case Nil
  then show length A ≤ length [] by simp
next
  case (Cons a B)
  then show length A ≤ length (a # B)
  proof(cases a ∈ set A)
    case True
    have set (remove1 a A) ⊆ set B
      using Cons.prems by auto
    hence length (remove1 a A) ≤ length B
      using Cons.IH Cons.prems by auto
    then show length A ≤ length (a # B)
      by (simp add: True length-remove1)
  next
    case False
    hence set A ⊆ set B
      using Cons.prems by auto
    hence length A ≤ length B
      using Cons.IH Cons.prems by auto
    then show length A ≤ length (a # B)
      by simp
  qed
qed

lemma set-subset-length-eq:
  assumes set A ⊆ set B and length B ≤ length A
    and distinct A and distinct B
  shows set A = set B
proof −
  have length A ≤ length B
    using assms by (simp add: subset-distinct-le)
  moreover from this have card (set A) = card (set B)
    using assms by (simp add: distinct-card le-antisym)
  ultimately show set A = set B
    using assms(1) by (simp add: card-subset-eq)
qed

lemma length-diff-Suc-exists:
  assumes length xs − length ys = Suc m
    and set ys ⊆ set xs
    and distinct ys and distinct xs
```

```
  shows ∃ e. e ∈ set xs ∧ e ∉ set ys
  using assms proof(induction xs arbitrary: ys)
  case Nil
  then show ∃ e. e ∈ set [] ∧ e ∉ set ys
    by simp
next
  case (Cons a xs)
  then show ∃ e. e ∈ set (a # xs) ∧ e ∉ set ys
  proof(cases a ∈ set ys)
    case True
    have IH: ∃ e. e ∈ set xs ∧ e ∉ set (remove1 a ys)
    proof −
      have length xs − length (remove1 a ys) = Suc m
       by (metis Cons.prems(1) One-nat-def Suc-pred True diff-Suc-Suc length-Cons
           length-pos-if-in-set length-remove1)
      moreover have set (remove1 a ys) ⊆ set xs
        using Cons.prems by auto
      ultimately show ?thesis
        by (meson Cons.IH Cons.prems distinct.simps(2) distinct-remove1)
    qed
    moreover have set ys − {a} ⊆ set xs
      using Cons.prems(2) by auto
    ultimately show ∃ e. e ∈ set (a # xs) ∧ e ∉ set ys
       by (metis Cons.prems(4) distinct.simps(2) in-set-remove1 set-subset-Cons
subsetCE)
  next
    case False
    then show ∃ e. e ∈ set (a # xs) ∧ e ∉ set ys
      by auto
  qed
qed

lemma app-length-lt-exists:
  assumes xsa @ zsa = xs @ ys
    and length xsa ≤ length xs
  shows xsa @ (drop (length xsa) xs) = xs
  using assms by (induction xsa arbitrary: xs zsa ys, simp,
      metis append-eq-append-conv-if append-take-drop-id)

lemma list-order-monotonic:
  assumes insert-ops A and insert-ops B
    and set A ⊆ set B
    and list-order A x y
  shows list-order B x y
   using assms proof(induction rule: measure-induct-rule[where f=λx. (length x
− length A)])
  case (less xa)
  have distinct (map fst A) and distinct (map fst xa) and
    sorted (map fst A) and sorted (map fst xa)
```

```
    using less.prems by (auto simp add: insert-ops-def spec-ops-def)
  hence distinct A and distinct xa
    by (auto simp add: distinct-fst)
  then show list-order xa x y
  proof(cases length xa − length A)
    case 0
    hence set A = set xa
      using set-subset-length-eq less.prems(3) ‹distinct A› ‹distinct xa› diff-is-0-eq
by blast
    hence A = xa
      using ‹distinct (map fst A)› ‹distinct (map fst xa)›
        ‹sorted (map fst A)› ‹sorted (map fst xa)› map-sorted-distinct-set-unique
      by (metis distinct-map less.prems(3) subset-Un-eq)
    then show list-order xa x y
      using less.prems(4) by blast
  next
    case (Suc nat)
    then obtain e where e ∈ set xa and e ∉ set A
      using length-diff-Suc-exists ‹distinct A› ‹distinct xa› less.prems(3) by blast
    hence IH: list-order (remove1 e xa) x y
    proof −
      have length (remove1 e xa) − length A < Suc nat
        using diff-Suc-1 diff-commute length-remove1 less-Suc-eq Suc ‹e ∈ set xa›
by metis
      moreover have insert-ops (remove1 e xa)
        by (simp add: insert-ops-remove1 less.prems(2))
      moreover have set A ⊆ set (remove1 e xa)
          by (metis (no-types, lifting) ‹e ∉ set A› in-set-remove1 less.prems(3)
rev-subsetD subsetI)
      ultimately show ?thesis
        by (simp add: Suc less.IH less.prems(1) less.prems(4))
    qed
   then obtain xs ys zs where interp-ins (remove1 e xa) = xs @ x # ys @ y # zs
      using list-order-def by fastforce
    moreover obtain oid ref where e-pair: e = (oid, ref)
      by fastforce
    moreover obtain ps ss where xa-split: xa = ps @ [e] @ ss and e ∉ set ps
      using split-list-first ‹e ∈ set xa› by fastforce
    hence remove1 e (ps @ e # ss) = ps @ ss
      by (simp add: remove1-append)
    moreover from this have insert-ops (ps @ ss) and insert-ops (ps @ e # ss)
     using xa-split less.prems(2) by (metis append-Cons append-Nil insert-ops-remove1,
auto)
    then obtain xsa ysa zsa where interp-ins (ps @ ss) = xsa @ zsa
      and interp-xa: interp-ins (ps @ (oid, ref) # ss) = xsa @ ysa @ zsa
      using insert-preserves-order e-pair by metis
    moreover have xsa-zsa: xsa @ zsa = xs @ x # ys @ y # zs
      using interp-ins-def remove1-append calculation xa-split by auto
    then show list-order xa x y
```

```
    proof(cases length xsa ≤ length xs)
      case True
      then obtain ts where xsa@ts = xs
        using app-length-lt-exists xsa-zsa by blast
      hence interp-ins xa = (xsa @ ysa @ ts) @ [x] @ ys @ [y] @ zs
        using calculation xa-split by auto
      then show list-order xa x y
        using list-order-def by blast
    next
      case False
      then show list-order xa x y
      proof(cases length xsa ≤ length (xs @ x # ys))
        case True
        have xsa-zsa1: xsa @ zsa = (xs @ x # ys) @ (y # zs)
          by (simp add: xsa-zsa)
        then obtain us where xsa @ us = xs @ x # ys
          using app-length-lt-exists True by blast
        moreover from this have xs @ x # (drop (Suc (length xs)) xsa) = xsa
          using append-eq-append-conv-if id-take-nth-drop linorder-not-less
            nth-append nth-append-length False by metis
        moreover have us @ y # zs = zsa
            by (metis True xsa-zsa1 append-eq-append-conv-if append-eq-conv-conj
calculation(1))
        ultimately have interp-ins xa = xs @ [x] @
            ((drop (Suc (length xs)) xsa) @ ysa @ us) @ [y] @ zs
          by (simp add: e-pair interp-xa xa-split)
        then show list-order xa x y
          using list-order-def by blast
      next
        case False
        hence length (xs @ x # ys) < length xsa
          using not-less by blast
        hence length (xs @ x # ys @ [y]) ≤ length xsa
          by simp
        moreover have (xs @ x # ys @ [y]) @ zs = xsa @ zsa
          by (simp add: xsa-zsa)
        ultimately obtain vs where (xs @ x # ys @ [y]) @ vs = xsa
          using app-length-lt-exists by blast
        hence xsa @ ysa @ zsa = xs @ [x] @ ys @ [y] @ vs @ ysa @ zsa
          by simp
        hence interp-ins xa = xs @ [x] @ ys @ [y] @ (vs @ ysa @ zsa)
          using e-pair interp-xa xa-split by auto
        then show list-order xa x y
          using list-order-def by blast
      qed
    qed
  qed
qed
```

**end**

# 3 Relationship to Strong List Specification

In this section we show that our list specification is stronger than the $\mathcal{A}_{\mathsf{strong}}$ specification of collaborative text editing by Attiya et al. [1]. We do this by showing that the OpSet interpretation of any set of insertion and deletion operations satisfies all of the consistency criteria that constitute the $\mathcal{A}_{\mathsf{strong}}$ specification.

Attiya et al.'s specification is as follows [1]:

> An abstract execution $A = (H, \mathsf{vis})$ belongs to the *strong list specification* $\mathcal{A}_{\mathsf{strong}}$ if and only if there is a relation $\mathsf{lo} \subseteq \mathsf{elems}(A) \times \mathsf{elems}(A)$, called the *list order*, such that:
>
> 1. Each event $e = do(op, w) \in H$ returns a sequence of elements $w = a_0 \dots a_{n-1}$, where $a_i \in \mathsf{elems}(A)$, such that
>    (a) $w$ contains exactly the elements visible to $e$ that have been inserted, but not deleted:
>
> $$\forall a.\ a \in w \iff (do(\mathsf{ins}(a, \_), \_) \leq_{\mathsf{vis}} e) \wedge \neg(do(\mathsf{del}(a), \_) \leq_{\mathsf{vis}} e).$$
>
>    (b) The order of the elements is consistent with the list order:
>
> $$\forall i, j.\ (i < j) \implies (a_i, a_j) \in \mathsf{lo}.$$
>
>    (c) Elements are inserted at the specified position: if $op = \mathsf{ins}(a, k)$, then $a = a_{\min\{k,\, n-1\}}$.
> 2. The list order $\mathsf{lo}$ is transitive, irreflexive and total, and thus determines the order of all insert operations in the execution.

This specification considers only insertion and deletion operations, but no assignment. Moreover, it considers only a single list object, not a graph of composable objects like in our paper. Thus, we prove the relationship to $\mathcal{A}_{\mathsf{strong}}$ using a simplified interpretation function that defines only insertion and deletion on a single list.

**theory** *List-Spec*
  **imports** *Insert-Spec*
**begin**

We first define a datatype for list operations, with two constructors: *Insert ref val*, and *Delete ref*. For insertion, the *ref* argument is the ID of the existing element after which we want to insert, or *None* to insert at the head of the list. The *val* argument is an arbitrary value to associate with the list

element. For deletion, the *ref* argument is the ID of the existing list element to delete.

```
datatype ('oid, 'val) list-op =
  Insert 'oid option 'val |
  Delete 'oid
```

When interpreting operations, the result is a pair (*list, vals*). The *list* contains the IDs of list elements in the correct order (equivalent to the list relation in the paper), and *vals* is a mapping from list element IDs to values (equivalent to the element relation in the paper).

Insertion delegates to the previously defined *insert-spec* interpretation function. Deleting a list element removes it from *vals*.

```
fun interp-op :: ('oid list × ('oid ⇀ 'val)) ⇒ ('oid × ('oid, 'val) list-op)
                 ⇒ ('oid list × ('oid ⇀ 'val)) where
  interp-op (list, vals) (oid, Insert ref val) = (insert-spec list (oid, ref), vals(oid ↦
val)) |
  interp-op (list, vals) (oid, Delete ref    ) = (list, vals(ref := None))
```

```
definition interp-ops :: ('oid × ('oid, 'val) list-op) list ⇒ ('oid list × ('oid ⇀ 'val))
where
  interp-ops ops ≡ foldl interp-op ([], Map.empty) ops
```

*list-order ops x y* holds iff, after interpreting the list of operations *ops*, the list element with ID $x$ appears before the list element with ID $y$ in the resulting list.

```
definition list-order :: ('oid × ('oid, 'val) list-op) list ⇒ 'oid ⇒ 'oid ⇒ bool where
  list-order ops x y ≡ ∃ xs ys zs. fst (interp-ops ops) = xs @ [x] @ ys @ [y] @ zs
```

The *make-insert* function generates a new operation for insertion into a given index in a given list. The exclamation mark is Isabelle's list subscript operator.

```
fun make-insert :: 'oid list ⇒ 'val ⇒ nat ⇒ ('oid, 'val) list-op where
  make-insert list val 0       = Insert None val |
  make-insert []   val k       = Insert None val |
  make-insert list val (Suc k) = Insert (Some (list ! (min k (length list − 1)))) val
```

The *list-ops* predicate is a specialisation of *spec-ops* to the *list-op* datatype: it describes a list of (ID, operation) pairs that is sorted by ID, and can thus be used for the sequential interpretation of the OpSet.

```
fun list-op-deps :: ('oid, 'val) list-op ⇒ 'oid set where
  list-op-deps (Insert (Some ref) -) = {ref} |
  list-op-deps (Insert  None      -) = {}    |
  list-op-deps (Delete  ref       ) = {ref}
```

```
locale list-opset = opset opset list-op-deps
  for opset :: ('oid::{linorder} × ('oid, 'val) list-op) set
```

**definition** *list-ops* :: (*'oid*::{*linorder*} × (*'oid*, *'val*) *list-op*) *list* ⇒ *bool* **where**
*list-ops ops* ≡ *spec-ops ops list-op-deps*

## 3.1 Lemmas about insertion and deletion

**definition** *insertions* :: (*'oid*::{*linorder*} × (*'oid*, *'val*) *list-op*) *list* ⇒ (*'oid* × *'oid option*) *list* **where**
*insertions ops* ≡ *List.map-filter* (λ*oper*.
*case oper of* (*oid*, *Insert ref val*) ⇒ *Some* (*oid*, *ref*) |
(*oid*, *Delete ref* ) ⇒ *None*) *ops*

**definition** *inserted-ids* :: (*'oid*::{*linorder*} × (*'oid*, *'val*) *list-op*) *list* ⇒ *'oid list* **where**
*inserted-ids ops* ≡ *List.map-filter* (λ*oper*.
*case oper of* (*oid*, *Insert ref val*) ⇒ *Some oid* |
(*oid*, *Delete ref* ) ⇒ *None*) *ops*

**definition** *deleted-ids* :: (*'oid*::{*linorder*} × (*'oid*, *'val*) *list-op*) *list* ⇒ *'oid list* **where**
*deleted-ids ops* ≡ *List.map-filter* (λ*oper*.
*case oper of* (*oid*, *Insert ref val*) ⇒ *None* |
(*oid*, *Delete ref* ) ⇒ *Some ref*) *ops*

**lemma** *interp-ops-unfold-last*:
**shows** *interp-ops* (*xs* @ [*x*]) = *interp-op* (*interp-ops xs*) *x*
**by** (*simp add*: *interp-ops-def*)

**lemma** *map-filter-append*:
**shows** *List.map-filter P* (*xs* @ *ys*) = *List.map-filter P xs* @ *List.map-filter P ys*
**by** (*auto simp add*: *List.map-filter-def*)

**lemma** *map-filter-Some*:
**assumes** *P x* = *Some y*
**shows** *List.map-filter P* [*x*] = [*y*]
**by** (*simp add*: *assms map-filter-simps*(*1*) *map-filter-simps*(*2*))

**lemma** *map-filter-None*:
**assumes** *P x* = *None*
**shows** *List.map-filter P* [*x*] = []
**by** (*simp add*: *assms map-filter-simps*(*1*) *map-filter-simps*(*2*))

**lemma** *insertions-last-ins*:
**shows** *insertions* (*xs* @ [(*oid*, *Insert ref val*)]) = *insertions xs* @ [(*oid*, *ref*)]
**by** (*simp add*: *insertions-def map-filter-Some map-filter-append*)

**lemma** *insertions-last-del*:
**shows** *insertions* (*xs* @ [(*oid*, *Delete ref*)]) = *insertions xs*
**by** (*simp add*: *insertions-def map-filter-None map-filter-append*)

**lemma** *insertions-fst-subset*:
**shows** *set* (*map fst* (*insertions ops*)) ⊆ *set* (*map fst ops*)
**proof**(*induction ops rule*: *List.rev-induct*)
**case** *Nil*
**then show** *set* (*map fst* (*insertions* [])) ⊆ *set* (*map fst* [])
**by** (*simp add*: *insert-ops-def spec-ops-def insertions-def map-filter-def*)
**next**
**case** (*snoc a ops*)
**obtain** *oid oper* **where** *a-pair*: *a* = (*oid*, *oper*)
**by** *fastforce*
**then show** *set* (*map fst* (*insertions* (*ops* @ [*a*]))) ⊆ *set* (*map fst* (*ops* @ [*a*]))
**proof**(*cases oper*)
**case** (*Insert ref val*)
**hence** *insertions* (*ops* @ [*a*]) = *insertions ops* @ [(*oid*, *ref*)]
**by** (*simp add*: *a-pair insertions-last-ins*)
**then show** *?thesis* **using** *snoc.IH a-pair* **by** *auto*
**next**
**case** (*Delete ref*)
**hence** *insertions* (*ops* @ [*a*]) = *insertions ops*
**by** (*simp add*: *a-pair insertions-last-del*)
**then show** *?thesis* **using** *snoc.IH* **by** *auto*
**qed**
**qed**

**lemma** *insertions-subset*:
**assumes** *list-ops A* **and** *list-ops B*
**and** *set A* ⊆ *set B*
**shows** *set* (*insertions A*) ⊆ *set* (*insertions B*)
**using** *assms* **proof**(*induction B arbitrary*: *A rule*: *List.rev-induct*)
**case** *Nil*
**then show** *set* (*insertions A*) ⊆ *set* (*insertions* [])
**by** (*simp add*: *insertions-def map-filter-simps*(*2*))
**next**
**case** (*snoc a ops*)
**obtain** *oid oper* **where** *a-pair*: *a* = (*oid*, *oper*)
**by** *fastforce*
**have** *list-ops ops*
**using** *list-ops-def spec-ops-rem-last snoc.prems*(*2*) **by** *blast*
**then show** *set* (*insertions A*) ⊆ *set* (*insertions* (*ops* @ [*a*]))
**proof**(*cases a* ∈ *set A*)
**case** *True*
**then obtain** *as bs* **where** *A-split*: *A* = *as* @ *a* # *bs* ∧ *a* ∉ *set as*
**by** (*meson split-list-first*)
**hence** *remove1 a A* = *as* @ *bs*
**by** (*simp add*: *remove1-append*)
**hence** *as-bs*: *insertions* (*remove1 a A*) = *insertions as* @ *insertions bs*
**by** (*simp add*: *insertions-def map-filter-append*)
**moreover have** *A* = *as* @ [*a*] @ *bs*

```
    by (simp add: A-split)
  hence as-a-bs: insertions A = insertions as @ insertions [a] @ insertions bs
    by (metis insertions-def map-filter-append)
  moreover have IH: set (insertions (remove1 a A)) ⊆ set (insertions ops)
  proof −
    have list-ops (remove1 a A)
      using snoc.prems(1) list-ops-def spec-ops-remove1 by blast
    moreover have set (remove1 a A) ⊆ set ops
    proof −
      have distinct A
        using snoc.prems(1) list-ops-def spec-ops-distinct by blast
      hence a ∉ set (remove1 a A)
        by auto
      moreover have set (ops @ [a]) = set ops ∪ {a}
        by auto
      moreover have set (remove1 a A) ⊆ set A
        by (simp add: set-remove1-subset)
      ultimately show set (remove1 a A) ⊆ set ops
        using snoc.prems(3) by blast
    qed
    ultimately show ?thesis
      by (simp add: ‹list-ops ops› snoc.IH)
  qed
  ultimately show ?thesis
  proof(cases oper)
    case (Insert ref val)
    hence insertions [a] = [(oid, ref)]
      by (simp add: insertions-def map-filter-Some a-pair)
    hence set (insertions A) = set (insertions (remove1 a A)) ∪ {(oid, ref)}
      using as-a-bs as-bs by auto
     moreover have set (insertions (ops @ [a])) = set (insertions ops) ∪ {(oid,
ref)}
      by (simp add: Insert a-pair insertions-last-ins)
    ultimately show ?thesis
      using IH by auto
  next
    case (Delete ref)
    hence insertions [a] = []
      by (simp add: insertions-def map-filter-None a-pair)
    hence set (insertions A) = set (insertions (remove1 a A))
      using as-a-bs as-bs by auto
    moreover have set (insertions (ops @ [a])) = set (insertions ops)
      by (simp add: Delete a-pair insertions-last-del)
    ultimately show ?thesis
      using IH by auto
  qed
 next
  case False
  hence set A ⊆ set ops
```

```
    using DiffE snoc.prems by auto
  hence set (insertions A) ⊆ set (insertions ops)
    using snoc.IH snoc.prems(1) ‹list-ops ops› by blast
  moreover have set (insertions ops) ⊆ set (insertions (ops @ [a]))
    by (simp add: insertions-def map-filter-append)
  ultimately show ?thesis
    by blast
 qed
qed

lemma list-ops-insertions:
 assumes list-ops ops
 shows insert-ops (insertions ops)
 using assms proof(induction ops rule: List.rev-induct)
 case Nil
 then show insert-ops (insertions [])
  by (simp add: insert-ops-def spec-ops-def insertions-def map-filter-def)
next
 case (snoc a ops)
 hence IH: insert-ops (insertions ops)
  using list-ops-def spec-ops-rem-last by blast
 obtain oid oper where a-pair: a = (oid, oper)
  by fastforce
 then show insert-ops (insertions (ops @ [a]))
 proof(cases oper)
  case (Insert ref val)
  hence insertions (ops @ [a]) = insertions ops @ [(oid, ref)]
   by (simp add: a-pair insertions-last-ins)
  moreover have ⋀i. i ∈ set (map fst ops) ⟹ i < oid
   using a-pair list-ops-def snoc.prems spec-ops-id-inc by fastforce
  hence ⋀i. i ∈ set (map fst (insertions ops)) ⟹ i < oid
   using insertions-fst-subset by blast
  moreover have list-op-deps oper = set-option ref
   using Insert by (cases ref, auto)
  hence ⋀r. r ∈ set-option ref ⟹ r < oid
   using list-ops-def spec-ops-ref-less
   by (metis a-pair last-in-set snoc.prems snoc-eq-iff-butlast)
  ultimately show ?thesis
   using IH insert-ops-def spec-ops-add-last by metis
 next
  case (Delete ref)
  hence insertions (ops @ [a]) = insertions ops
   by (simp add: a-pair insertions-last-del)
  then show ?thesis by (simp add: IH)
 qed
qed

lemma inserted-ids-last-ins:
 shows inserted-ids (xs @ [(oid, Insert ref val)]) = inserted-ids xs @ [oid]
```

**by** (*simp add*: *inserted-ids-def map-filter-Some map-filter-append*)

**lemma** *inserted-ids-last-del*:
**shows** *inserted-ids* (*xs* @ [(*oid*, *Delete ref*)]) = *inserted-ids xs*
**by** (*simp add*: *inserted-ids-def map-filter-None map-filter-append*)

**lemma** *inserted-ids-exist*:
**shows** *oid* ∈ *set* (*inserted-ids ops*) ⟷ (∃ *ref val*. (*oid*, *Insert ref val*) ∈ *set ops*)
**proof**(*induction ops rule*: *List.rev-induct*)
**case** *Nil*
**then show** *oid* ∈ *set* (*inserted-ids* []) ⟷ (∃ *ref val*. (*oid*, *Insert ref val*) ∈ *set* [])
**by** (*simp add*: *inserted-ids-def List.map-filter-def*)
**next**
**case** (*snoc a ops*)
**obtain** *i oper* **where** *a-pair*: *a* = (*i*, *oper*)
**by** *fastforce*
**then show** *oid* ∈ *set* (*inserted-ids* (*ops* @ [*a*])) ⟷
(∃ *ref val*. (*oid*, *Insert ref val*) ∈ *set* (*ops* @ [*a*]))
**proof**(*cases oper*)
**case** (*Insert r v*)
**moreover from** *this* **have** *inserted-ids* (*ops* @ [*a*]) = *inserted-ids ops* @ [*i*]
**by** (*simp add*: *a-pair inserted-ids-last-ins*)
**ultimately show** *?thesis*
**using** *snoc.IH a-pair* **by** *auto*
**next**
**case** (*Delete r*)
**moreover from** *this* **have** *inserted-ids* (*ops* @ [*a*]) = *inserted-ids ops*
**by** (*simp add*: *a-pair inserted-ids-last-del*)
**ultimately show** *?thesis*
**by** (*simp add*: *a-pair snoc.IH*)
**qed**
**qed**

**lemma** *deleted-ids-last-ins*:
**shows** *deleted-ids* (*xs* @ [(*oid*, *Insert ref val*)]) = *deleted-ids xs*
**by** (*simp add*: *deleted-ids-def map-filter-None map-filter-append*)

**lemma** *deleted-ids-last-del*:
**shows** *deleted-ids* (*xs* @ [(*oid*, *Delete ref*)]) = *deleted-ids xs* @ [*ref*]
**by** (*simp add*: *deleted-ids-def map-filter-Some map-filter-append*)

**lemma** *deleted-ids-exist*:
**shows** *ref* ∈ *set* (*deleted-ids ops*) ⟷ (∃ *i*. (*i*, *Delete ref*) ∈ *set ops*)
**proof**(*induction ops rule*: *List.rev-induct*)
**case** *Nil*
**then show** *ref* ∈ *set* (*deleted-ids* []) ⟷ (∃ *i*. (*i*, *Delete ref*) ∈ *set* [])
**by** (*simp add*: *deleted-ids-def List.map-filter-def*)
**next**
**case** (*snoc a ops*)

```
  obtain oid oper where a-pair: a = (oid, oper)
    by fastforce
  then show ref ∈ set (deleted-ids (ops @ [a])) ⟷ (∃ i. (i, Delete ref) ∈ set (ops
@ [a]))
  proof(cases oper)
    case (Insert r v)
    moreover from this have deleted-ids (ops @ [a]) = deleted-ids ops
      by (simp add: a-pair deleted-ids-last-ins)
    ultimately show ?thesis
      using a-pair snoc.IH by auto
  next
    case (Delete r)
    moreover from this have deleted-ids (ops @ [a]) = deleted-ids ops @ [r]
      by (simp add: a-pair deleted-ids-last-del)
    ultimately show ?thesis
      using a-pair snoc.IH by auto
  qed
qed

lemma deleted-ids-refs-older:
  assumes list-ops (ops @ [(oid, oper)])
  shows ⋀ref. ref ∈ set (deleted-ids ops) ⟹ ref < oid
proof −
  fix ref
  assume ref ∈ set (deleted-ids ops)
  then obtain i where in-ops: (i, Delete ref) ∈ set ops
    using deleted-ids-exist by blast
  have ref < i
  proof −
    have ⋀i oper r. (i, oper) ∈ set ops ⟹ r ∈ list-op-deps oper ⟹ r < i
      by (meson assms list-ops-def spec-ops-ref-less spec-ops-rem-last)
    thus ref < i
      using in-ops by auto
  qed
  moreover have i < oid
  proof −
    have ⋀i. i ∈ set (map fst ops) ⟹ i < oid
      using assms by (simp add: list-ops-def spec-ops-id-inc)
    thus ?thesis
      by (metis in-ops in-set-zipE zip-map-fst-snd)
  qed
  ultimately show ref < oid
    using order.strict-trans by blast
qed
```

## 3.2 Lemmas about interpreting operations

```
lemma interp-ops-list-equiv:
  shows fst (interp-ops ops) = interp-ins (insertions ops)
```

**proof**(*induction ops rule*: *List.rev-induct*)
**case** *Nil*
**have** *1*: *fst* (*interp-ops* []) = []
**by** (*simp add*: *interp-ops-def*)
**have** *2*: *interp-ins* (*insertions* []) = []
**by** (*simp add*: *insertions-def map-filter-def interp-ins-def*)
**show** *fst* (*interp-ops* []) = *interp-ins* (*insertions* [])
**by** (*simp add*: *1 2*)
**next**
**case** (*snoc a ops*)
**obtain** *oid oper* **where** *a-pair*: *a* = (*oid*, *oper*)
**by** *fastforce*
**then show** *fst* (*interp-ops* (*ops* @ [*a*])) = *interp-ins* (*insertions* (*ops* @ [*a*]))
**proof**(*cases oper*)
**case** (*Insert ref val*)
**hence** *insertions* (*ops* @ [*a*]) = *insertions ops* @ [(*oid*, *ref*)]
**by** (*simp add*: *a-pair insertions-last-ins*)
**hence** *interp-ins* (*insertions* (*ops* @ [*a*])) = *insert-spec* (*interp-ins* (*insertions ops*)) (*oid*, *ref*)
**by** (*simp add*: *interp-ins-tail-unfold*)
**moreover have** *fst* (*interp-ops* (*ops* @ [*a*])) = *insert-spec* (*fst* (*interp-ops ops*)) (*oid*, *ref*)
**by** (*metis Insert a-pair fst-conv interp-op.simps*(*1*) *interp-ops-unfold-last prod.collapse*)
**ultimately show** *?thesis*
**using** *snoc.IH* **by** *auto*
**next**
**case** (*Delete ref*)
**hence** *insertions* (*ops* @ [*a*]) = *insertions ops*
**by** (*simp add*: *a-pair insertions-last-del*)
**moreover have** *fst* (*interp-ops* (*ops* @ [*a*])) = *fst* (*interp-ops ops*)
**by** (*metis Delete a-pair eq-fst-iff interp-op.simps*(*2*) *interp-ops-unfold-last*)
**ultimately show** *?thesis*
**using** *snoc.IH* **by** *auto*
**qed**
**qed**

**lemma** *interp-ops-distinct*:
**assumes** *list-ops ops*
**shows** *distinct* (*fst* (*interp-ops ops*))
**by** (*simp add*: *assms interp-ins-distinct interp-ops-list-equiv list-ops-insertions*)

**lemma** *list-order-equiv*:
**shows** *list-order ops x y* ⟷ *Insert-Spec.list-order* (*insertions ops*) *x y*
**by** (*simp add*: *Insert-Spec.list-order-def List-Spec.list-order-def interp-ops-list-equiv*)

**lemma** *interp-ops-vals-domain*:
**assumes** *list-ops ops*
**shows** *dom* (*snd* (*interp-ops ops*)) = *set* (*inserted-ids ops*) − *set* (*deleted-ids ops*)

```
  using assms proof(induction ops rule: List.rev-induct)
  case Nil
  have 1: interp-ops [] = ([], Map.empty)
    by (simp add: interp-ops-def)
  moreover have 2: inserted-ids [] = [] and deleted-ids [] = []
    by (auto simp add: inserted-ids-def deleted-ids-def map-filter-simps(2))
 ultimately show dom (snd (interp-ops [])) = set (inserted-ids []) − set (deleted-ids
[])
    by (simp add: 1 2)
next
  case (snoc x xs)
  hence IH: dom (snd (interp-ops xs)) = set (inserted-ids xs) − set (deleted-ids xs)
    using list-ops-def spec-ops-rem-last by blast
  obtain oid oper where x-pair: x = (oid, oper)
    by fastforce
  obtain list vals where interp-xs: interp-ops xs = (list, vals)
    by fastforce
  then show dom (snd (interp-ops (xs @ [x]))) =
            set (inserted-ids (xs @ [x])) − set (deleted-ids (xs @ [x]))
  proof(cases oper)
    case (Insert ref val)
    hence interp-ops (xs @ [x]) = (insert-spec list (oid, ref), vals(oid ↦ val))
      by (simp add: interp-ops-unfold-last interp-xs x-pair)
    hence dom (snd (interp-ops (xs @ [x]))) = (dom vals) ∪ {oid}
      by simp
    moreover have set (inserted-ids xs) − set (deleted-ids xs) = dom vals
      using IH interp-xs by auto
    moreover have inserted-ids (xs @ [x]) = inserted-ids xs @ [oid]
      by (simp add: Insert inserted-ids-last-ins x-pair)
    moreover have deleted-ids (xs @ [x]) = deleted-ids xs
      by (simp add: Insert deleted-ids-last-ins x-pair)
    hence set (inserted-ids (xs @ [x])) − set (deleted-ids (xs @ [x])) =
          {oid} ∪ set (inserted-ids xs) − set (deleted-ids xs)
      using calculation(3) by auto
    moreover have ... = {oid} ∪ (set (inserted-ids xs) − set (deleted-ids xs))
      using deleted-ids-refs-older snoc.prems x-pair by blast
    ultimately show ?thesis by auto
  next
    case (Delete ref)
    hence interp-ops (xs @ [x]) = (list, vals(ref := None))
      by (simp add: interp-ops-unfold-last interp-xs x-pair)
    hence dom (snd (interp-ops (xs @ [x]))) = (dom vals) − {ref}
      by simp
    moreover have set (inserted-ids xs) − set (deleted-ids xs) = dom vals
      using IH interp-xs by auto
    moreover have inserted-ids (xs @ [x]) = inserted-ids xs
      by (simp add: Delete inserted-ids-last-del x-pair)
    moreover have deleted-ids (xs @ [x]) = deleted-ids xs @ [ref]
      by (simp add: Delete deleted-ids-last-del x-pair)
```

```
    hence set (inserted-ids (xs @ [x])) − set (deleted-ids (xs @ [x])) =
          set (inserted-ids xs) − (set (deleted-ids xs) ∪ {ref})
      using calculation(3) by auto
    moreover have ... = set (inserted-ids xs) − set (deleted-ids xs) − {ref}
      by blast
    ultimately show ?thesis by auto
  qed
qed

lemma insert-spec-nth-oid:
  assumes distinct xs
    and n < length xs
  shows insert-spec xs (oid, Some (xs ! n)) ! Suc n = oid
  using assms proof(induction xs arbitrary: n)
  case Nil
  then show insert-spec [] (oid, Some ([] ! n)) ! Suc n = oid
    by simp
next
  case (Cons a xs)
  have distinct (a # xs)
    using Cons.prems(1) by auto
  then show insert-spec (a # xs) (oid, Some ((a # xs) ! n)) ! Suc n = oid
  proof(cases a = (a # xs) ! n)
    case True
    then have n = 0
      using ‹distinct (a # xs)› Cons.prems(2) gr-implies-not-zero by force
    then show insert-spec (a # xs) (oid, Some ((a # xs) ! n)) ! Suc n = oid
      by auto
  next
    case False
    then have n > 0
      using ‹distinct (a # xs)› Cons.prems(2) gr-implies-not-zero by force
    then obtain m where n = Suc m
      using Suc-pred′ by blast
    then show insert-spec (a # xs) (oid, Some ((a # xs) ! n)) ! Suc n = oid
      using Cons.IH Cons.prems by auto
  qed
qed

lemma insert-spec-inc-length:
  assumes distinct xs
    and n < length xs
  shows length (insert-spec xs (oid, Some (xs ! n))) = Suc (length xs)
  using assms proof(induction xs arbitrary: n, simp)
  case (Cons a xs)
  have distinct (a # xs)
    using Cons.prems(1) by auto
  then show length (insert-spec (a # xs) (oid, Some ((a # xs) ! n))) = Suc (length
(a # xs))
```

```
  proof(cases n)
    case 0
    hence insert-spec (a # xs) (oid, Some ((a # xs) ! n)) = a # oid # xs
      by simp
    then show ?thesis
      by simp
  next
    case (Suc nat)
    hence nat < length xs
      using Cons.prems(2) by auto
    hence length (insert-spec xs (oid, Some (xs ! nat))) = Suc (length xs)
      using Cons.IH Cons.prems(1) by auto
    then show ?thesis
      by (simp add: Suc)
  qed
qed

lemma list-split-two-elems:
  assumes distinct xs
    and x ∈ set xs and y ∈ set xs
    and x ≠ y
  shows ∃pre mid suf. xs = pre @ x # mid @ y # suf ∨ xs = pre @ y # mid @
x # suf
proof −
  obtain as bs where as-bs: xs = as @ [x] @ bs
    using assms(2) split-list-first by fastforce
  show ?thesis
  proof(cases y ∈ set as)
    case True
    then obtain cs ds where as = cs @ [y] @ ds
      using assms(3) split-list-first by fastforce
    then show ?thesis
      by (auto simp add: as-bs)
  next
    case False
    then have y ∈ set bs
      using as-bs assms(3) assms(4) by auto
    then obtain cs ds where bs = cs @ [y] @ ds
      using assms(3) split-list-first by fastforce
    then show ?thesis
      by (auto simp add: as-bs)
  qed
qed
```

### 3.3 Satisfying all conditions of $\mathcal{A}_{\textsf{strong}}$

Part 1(a) of Attiya et al.'s specification states that whenever the list is observed, the elements of the list are exactly those that have been inserted but

not deleted. $\mathcal{A}_{\mathsf{strong}}$ uses the visibility relation $\leq_{\mathsf{vis}}$ to capture the operations known to a node at some arbitrary point in the execution; in the OpSet model, we can simply prove the theorem for an arbitrary OpSet, since the contents of the OpSet at a particular time on a particular node correspond exactly to the set of operations known to that node at that time.

**theorem** *inserted-but-not-deleted*:
  **assumes** *list-ops ops*
    **and** *interp-ops ops* = (*list*, *vals*)
  **shows** *a* ∈ *dom* (*vals*) ⟷ (∃ *ref val*. (*a*, *Insert ref val*) ∈ *set ops*) ∧
                        (∄ *i*. (*i*, *Delete a*) ∈ *set ops*)
  **using** *assms deleted-ids-exist inserted-ids-exist interp-ops-vals-domain*
  **by** (*metis Diff-iff snd-conv*)

Part 1(b) states that whenever the list is observed, the order of list elements is consistent with the global list order. We can define the global list order simply as the list order that arises from interpreting the OpSet containing all operations in the entire execution. Then, at any point in the execution, the OpSet is some subset of the set of all operations.

We can then rephrase condition 1(b) as follows: whenever list element $x$ appears before list element $y$ in the interpretation of *some-ops*, then for any OpSet *all-ops* that is a superset of *some-ops*, $x$ must also appear before $y$ in the interpretation of *all-ops*. In other words, adding more operations to the OpSet does not change the relative order of any existing list elements.

**theorem** *list-order-consistent*:
  **assumes** *list-ops some-ops* **and** *list-ops all-ops*
    **and** *set some-ops* ⊆ *set all-ops*
    **and** *list-order some-ops x y*
  **shows** *list-order all-ops x y*
 **using** *assms list-order-monotonic list-ops-insertions insertions-subset list-order-equiv*
**by** *metis*

Part 1(c) states that inserted elements appear at the specified position: that is, immediately after an insertion of *oid* at index $k$, the list index $k$ does indeed contain *oid* (provided that $k$ is less than the length of the list). We prove this property below.

**theorem** *correct-position-insert*:
  **assumes** *list-ops* (*ops* @ [(*oid*, *ins*)])
    **and** *ins* = *make-insert* (*fst* (*interp-ops ops*)) *val k*
    **and** *list* = *fst* (*interp-ops* (*ops* @ [(*oid*, *ins*)]))
  **shows** *list* ! (*min k* (*length list* − *1*)) = *oid*
**proof**(*cases k* = *0* ∨ *fst* (*interp-ops ops*) = [])
  **case** *True*
  **moreover from** *this*
  **have** *make-insert* (*fst* (*interp-ops ops*)) *val k* = *Insert None val*
    **and** *min-k*: *min k* (*length* (*fst* (*interp-ops ops*))) = *0*
    **by** (*cases k*, *auto*)

```
  hence fst (interp-ops (ops @ [(oid, ins)])) = oid # fst (interp-ops ops)
    using assms(2) interp-ops-unfold-last
    by (metis fst-conv insert-spec.simps(1) interp-op.simps(1) prod.collapse)
  ultimately show ?thesis
    by (simp add: min-k assms(3))
next
  case False
  moreover from this have k > 0 and fst (interp-ops ops) ≠ []
    using neq0-conv by blast+
  from this obtain nat where k = Suc nat
    using gr0-implies-Suc by blast
  hence make-insert (fst (interp-ops ops)) val k =
      Insert (Some ((fst (interp-ops ops)) ! (min nat (length (fst (interp-ops ops))
− 1)))) val
    using False by (cases fst (interp-ops ops), auto)
  hence fst (interp-ops (ops @ [(oid, ins)])) =
        insert-spec (fst (interp-ops ops)) (oid, Some ((fst (interp-ops ops)) ! (min
nat (length (fst (interp-ops ops)) − 1))))
   by (metis assms(2) fst-conv interp-op.simps(1) interp-ops-unfold-last prod.collapse)
 moreover have min nat (length (fst (interp-ops ops)) − 1) < length (fst (interp-ops
ops))
    by (simp add: ‹fst (interp-ops ops) ≠ []› min.strict-coboundedI2)
  moreover have distinct (fst (interp-ops ops))
    using interp-ops-distinct list-ops-def spec-ops-rem-last assms(1) by blast
  moreover have length list = Suc (length (fst (interp-ops ops)))
    using assms(3) calculation by (simp add: insert-spec-inc-length)
  ultimately show ?thesis
    using assms insert-spec-nth-oid
   by (metis Suc-diff-1 ‹k = Suc nat› diff-Suc-1 length-greater-0-conv min-Suc-Suc)
qed
```

Part 2 states that the list order relation must be transitive, irreflexive, and total. These three properties are straightforward to prove, using our definition of the *list-order* predicate.

```
theorem list-order-trans:
  assumes list-ops ops
    and list-order ops x y
    and list-order ops y z
  shows list-order ops x z
  using assms list-order-trans list-ops-insertions list-order-equiv by blast

theorem list-order-irrefl:
  assumes list-ops ops
  shows ¬ list-order ops x x
proof −
  have list-order ops x x ⟹ False
  proof −
    assume list-order ops x x
    then obtain xs ys zs where split: fst (interp-ops ops) = xs @ [x] @ ys @ [x] @
```

*zs*
    **by** (*meson List-Spec.list-order-def*)
  **moreover have** *distinct* (*fst* (*interp-ops ops*))
    **by** (*simp add*: *assms interp-ops-distinct*)
  **ultimately show** *False*
    **by** (*simp add*: *split*)
 **qed**
 **thus** ¬ *list-order ops x x*
  **by** *blast*
**qed**

**theorem** *list-order-total*:
 **assumes** *list-ops ops*
  **and** $x \in$ *set* (*fst* (*interp-ops ops*))
  **and** $y \in$ *set* (*fst* (*interp-ops ops*))
  **and** $x \neq y$
 **shows** *list-order ops x y* ∨ *list-order ops y x*
**proof** −
 **have** *distinct* (*fst* (*interp-ops ops*))
  **using** *assms*(*1*) **by** (*simp add*: *interp-ops-distinct*)
 **then obtain** *pre mid suf*
  **where** *fst* (*interp-ops ops*) = *pre* @ *x* # *mid* @ *y* # *suf* ∨
      *fst* (*interp-ops ops*) = *pre* @ *y* # *mid* @ *x* # *suf*
  **using** *list-split-two-elems assms* **by** *metis*
 **then show** *list-order ops x y* ∨ *list-order ops y x*
  **by** (*simp add*: *list-order-def*, *blast*)
**qed**

**end**

# 4 Interleaving of concurrent insertions

In this section we prove that our list specification rules out interleaving of concurrent insertion sequences starting at the same position.

**theory** *Interleaving*
 **imports** *Insert-Spec*
**begin**

## 4.1 Lemmas about *insert-ops*

**lemma** *map-fst-append1*:
 **assumes** ∀ *i* ∈ *set* (*map fst xs*). *P i*
  **and** *P x*
 **shows** ∀ *i* ∈ *set* (*map fst* (*xs* @ [(*x*, *y*)])). *P i*
 **using** *assms* **by** (*induction xs*, *auto*)

**lemma** *insert-ops-split*:
 **assumes** *insert-ops ops*

```
    and (oid, ref) ∈ set ops
  shows ∃ pre suf. ops = pre @ [(oid, ref)] @ suf ∧
          (∀ i ∈ set (map fst pre). i < oid) ∧
          (∀ i ∈ set (map fst suf). oid < i)
  using assms proof(induction ops rule: List.rev-induct)
  case Nil
  then show ?case by auto
next
  case (snoc x xs)
  then show ?case
  proof(cases x = (oid, ref))
    case True
    moreover from this have ∀ i ∈ set (map fst xs). i < oid
      using last-op-greatest snoc.prems(1) by blast
    ultimately have xs @ [x] = xs @ [(oid, ref)] @ [] ∧
          (∀ i ∈ set (map fst xs). i < oid) ∧
          (∀ i ∈ set (map fst []). oid < i)
      by auto
    then show ?thesis by force
  next
    case False
    hence (oid, ref) ∈ set xs
      using snoc.prems(2) by auto
    from this obtain pre suf where IH: xs = pre @ [(oid, ref)] @ suf ∧
        (∀ i ∈ set (map fst pre). i < oid) ∧
        (∀ i ∈ set (map fst suf). oid < i)
      using snoc.IH snoc.prems(1) by blast
    obtain xi xr where x-pair: x = (xi, xr)
      by force
    hence distinct (map fst (pre @ [(oid, ref)] @ suf @ [(xi, xr)]))
      by (metis IH append.assoc insert-ops-def spec-ops-def snoc.prems(1))
    hence xi ≠ oid
      by auto
    have xi-max: ∀ x ∈ set (map fst (pre @ [(oid, ref)] @ suf)). x < xi
      using IH last-op-greatest snoc.prems(1) x-pair by blast
    then show ?thesis
    proof(cases xi < oid)
      case True
      moreover from this have ∀ x ∈ set suf. fst x < oid
        using xi-max by auto
      hence suf = []
        using IH last-in-set by fastforce
      ultimately have xs @ [x] = (pre @ [(xi, xr)]) @ [] ∧
          (∀ i ∈ set (map fst ((pre @ [(xi, xr)]))). i < oid) ∧
          (∀ i ∈ set (map fst []). oid < i)
        using dual-order.asym xi-max by auto
      then show ?thesis by (simp add: IH)
    next
      case False
```

```
    hence oid < xi
      using ‹xi ≠ oid› by auto
    hence ∀ i ∈ set (map fst (suf @ [(xi, xr)])). oid < i
      using IH map-fst-append1 by auto
    hence xs @ [x] = pre @ [(oid, ref)] @ (suf @ [(xi, xr)]) ∧
          (∀ i ∈ set (map fst pre). i < oid) ∧
          (∀ i ∈ set (map fst (suf @ [(xi, xr)])). oid < i)
      by (simp add: IH x-pair)
    then show ?thesis by blast
  qed
 qed
qed

lemma insert-ops-split-2:
 assumes insert-ops ops
   and (xid, xr) ∈ set ops
   and (yid, yr) ∈ set ops
   and xid < yid
 shows ∃ as bs cs. ops = as @ [(xid, xr)] @ bs @ [(yid, yr)] @ cs ∧
         (∀ i ∈ set (map fst as). i < xid) ∧
         (∀ i ∈ set (map fst bs). xid < i ∧ i < yid) ∧
         (∀ i ∈ set (map fst cs). yid < i)
proof −
 obtain as as1 where x-split: ops = as @ [(xid, xr)] @ as1 ∧
    (∀ i ∈ set (map fst as). i < xid) ∧ (∀ i ∈ set (map fst as1). xid < i)
   using assms insert-ops-split by blast
 hence insert-ops ((as @ [(xid, xr)]) @ as1)
   using assms(1) by auto
 hence insert-ops as1
   using assms(1) insert-ops-rem-prefix by blast
 have (yid, yr) ∈ set as1
   using x-split assms by auto
 from this obtain bs cs where y-split: as1 = bs @ [(yid, yr)] @ cs ∧
    (∀ i ∈ set (map fst bs). i < yid) ∧ (∀ i ∈ set (map fst cs). yid < i)
   using assms insert-ops-split ‹insert-ops as1› by blast
 hence ops = as @ [(xid, xr)] @ bs @ [(yid, yr)] @ cs
   using x-split by blast
 moreover have ∀ i ∈ set (map fst bs). xid < i ∧ i < yid
   by (simp add: x-split y-split)
 ultimately show ?thesis
   using x-split y-split by blast
qed

lemma insert-ops-sorted-oids:
 assumes insert-ops (xs @ [(i1, r1)] @ ys @ [(i2, r2)])
 shows i1 < i2
proof −
 have ⋀i. i ∈ set (map fst (xs @ [(i1, r1)] @ ys)) ⟹ i < i2
   by (metis append.assoc assms last-op-greatest)
```

```
  moreover have i1 ∈ set (map fst (xs @ [(i1, r1)] @ ys))
    by auto
  ultimately show i1 < i2
    by blast
qed

lemma insert-ops-subset-last:
  assumes insert-ops (xs @ [x])
    and insert-ops ys
    and set ys ⊆ set (xs @ [x])
    and x ∈ set ys
  shows x = last ys
  using assms proof(induction ys, simp)
  case (Cons y ys)
  then show x = last (y # ys)
  proof(cases ys = [])
    case True
    then show x = last (y # ys)
      using Cons.prems(4) by auto
  next
    case ys-nonempty: False
    have x ≠ y
    proof −
      obtain mid l where ys = mid @ [l]
        using append-butlast-last-id ys-nonempty by metis
      moreover obtain li lr where l = (li, lr)
        by force
      moreover have ⋀i. i ∈ set (map fst (y # mid)) ⟹ i < li
        by (metis last-op-greatest Cons.prems(2) calculation append-Cons)
      hence fst y < li
        by simp
      moreover have ⋀i. i ∈ set (map fst xs) ⟹ i < fst x
        using assms(1) last-op-greatest by (metis prod.collapse)
      hence ⋀i. i ∈ set (map fst (y # ys)) ⟹ i ≤ fst x
        using Cons.prems(3) by fastforce
      ultimately show x ≠ y
        by fastforce
    qed
    then show x = last (y # ys)
      using Cons.IH Cons.prems insert-ops-rem-cons ys-nonempty
      by (metis dual-order.trans last-ConsR set-ConsD set-subset-Cons)
  qed
qed

lemma subset-butlast:
  assumes set xs ⊆ set (ys @ [y])
    and last xs = y
    and distinct xs
  shows set (butlast xs) ⊆ set ys
```

**using** *assms* **by** (*induction xs*, *auto*)

**lemma** *distinct-append-butlast1*:
**assumes** *distinct* (*map fst xs* @ *map fst ys*)
**shows** *distinct* (*map fst* (*butlast xs*) @ *map fst ys*)
**using** *assms* **proof**(*induction xs*, *simp*)
**case** (*Cons a xs*)
**have** *fst a* ∉ *set* (*map fst xs* @ *map fst ys*)
**using** *Cons.prems* **by** *auto*
**moreover have** *set* (*map fst* (*butlast xs*)) ⊆ *set* (*map fst xs*)
**by** (*metis in-set-butlastD map-butlast subsetI*)
**hence** *set* (*map fst* (*butlast xs*) @ *map fst ys*) ⊆ *set* (*map fst xs* @ *map fst ys*)
**by** *auto*
**ultimately have** *fst a* ∉ *set* (*map fst* (*butlast xs*) @ *map fst ys*)
**by** *blast*
**then show** *distinct* (*map fst* (*butlast* (*a* # *xs*)) @ *map fst ys*)
**using** *Cons.IH Cons.prems* **by** *auto*
**qed**

**lemma** *distinct-append-butlast2*:
**assumes** *distinct* (*map fst xs* @ *map fst ys*)
**shows** *distinct* (*map fst xs* @ *map fst* (*butlast ys*))
**using** *assms* **proof**(*induction xs*)
**case** *Nil*
**then show** *distinct* (*map fst* [] @ *map fst* (*butlast ys*))
**by** (*simp add*: *distinct-butlast map-butlast*)
**next**
**case** (*Cons a xs*)
**have** *fst a* ∉ *set* (*map fst xs* @ *map fst ys*)
**using** *Cons.prems* **by** *auto*
**moreover have** *set* (*map fst* (*butlast ys*)) ⊆ *set* (*map fst ys*)
**by** (*metis in-set-butlastD map-butlast subsetI*)
**hence** *set* (*map fst xs* @ *map fst* (*butlast ys*)) ⊆ *set* (*map fst xs* @ *map fst ys*)
**by** *auto*
**ultimately have** *fst a* ∉ *set* (*map fst xs* @ *map fst* (*butlast ys*))
**by** *blast*
**then show** *?case*
**using** *Cons.IH Cons.prems* **by** *auto*
**qed**

## 4.2 Lemmas about *interp-ins*

**lemma** *interp-ins-maybe-grow*:
**assumes** *insert-ops* (*xs* @ [(*oid*, *ref*)])
**shows** *set* (*interp-ins* (*xs* @ [(*oid*, *ref*)])) = *set* (*interp-ins xs*) ∨
*set* (*interp-ins* (*xs* @ [(*oid*, *ref*)])) = (*set* (*interp-ins xs*) ∪ {*oid*})
**by** (*cases ref*, *simp add*: *interp-ins-tail-unfold*,
*metis insert-spec-nonex insert-spec-set interp-ins-tail-unfold*)

**lemma** *interp-ins-maybe-grow2*:
**assumes** *insert-ops* (*xs* @ [*x*])
**shows** *set* (*interp-ins* (*xs* @ [*x*])) = *set* (*interp-ins xs*) ∨
*set* (*interp-ins* (*xs* @ [*x*])) = (*set* (*interp-ins xs*) ∪ {*fst x*})
**using** *assms interp-ins-maybe-grow* **by** (*cases x*, *auto*)

**lemma** *interp-ins-maybe-grow3*:
**assumes** *insert-ops* (*xs* @ *ys*)
**shows** ∃ *A*. *A* ⊆ *set* (*map fst ys*) ∧ *set* (*interp-ins* (*xs* @ *ys*)) = *set* (*interp-ins xs*) ∪ *A*
**using** *assms* **proof**(*induction ys rule*: *List.rev-induct*)
**case** *Nil*
**then show** *?case* **by** *simp*
**next**
**case** (*snoc x ys*)
**then have** *insert-ops* (*xs* @ *ys*)
**by** (*metis append-assoc insert-ops-rem-last*)
**then obtain** *A* **where** *IH*: *A* ⊆ *set* (*map fst ys*) ∧
*set* (*interp-ins* (*xs* @ *ys*)) = *set* (*interp-ins xs*) ∪ *A*
**using** *snoc.IH* **by** *blast*
**then show** *?case*
**proof**(*cases set* (*interp-ins* (*xs* @ *ys* @ [*x*])) = *set* (*interp-ins* (*xs* @ *ys*)))
**case** *True*
**moreover have** *A* ⊆ *set* (*map fst* (*ys* @ [*x*]))
**using** *IH* **by** *auto*
**ultimately show** *?thesis*
**using** *IH* **by** *auto*
**next**
**case** *False*
**then have** *set* (*interp-ins* (*xs* @ *ys* @ [*x*])) = *set* (*interp-ins* (*xs* @ *ys*)) ∪ {*fst x*}
**by** (*metis append-assoc interp-ins-maybe-grow2 snoc.prems*)
**moreover have** *A* ∪ {*fst x*} ⊆ *set* (*map fst* (*ys* @ [*x*]))
**using** *IH* **by** *auto*
**ultimately show** *?thesis*
**using** *IH Un-assoc* **by** *metis*
**qed**
**qed**

**lemma** *interp-ins-ref-nonex*:
**assumes** *insert-ops ops*
**and** *ops* = *xs* @ [(*oid*, *Some ref*)] @ *ys*
**and** *ref* ∉ *set* (*interp-ins xs*)
**shows** *oid* ∉ *set* (*interp-ins ops*)
**using** *assms* **proof**(*induction ys arbitrary*: *ops rule*: *List.rev-induct*)
**case** *Nil*
**then have** *interp-ins ops* = *insert-spec* (*interp-ins xs*) (*oid*, *Some ref*)
**by** (*simp add*: *interp-ins-tail-unfold*)
**moreover have** ⋀*i*. *i* ∈ *set* (*map fst xs*) ⟹ *i* < *oid*

```
    using Nil.prems last-op-greatest by fastforce
  hence ⋀i. i ∈ set (interp-ins xs) ⟹ i < oid
    by (meson interp-ins-subset subsetCE)
  ultimately show oid ∉ set (interp-ins ops)
    using assms(3) by auto
next
  case (snoc x ys)
  then have insert-ops (xs @ (oid, Some ref) # ys)
    by (metis append.assoc append.simps(1) append-Cons insert-ops-appendD)
  hence IH: oid ∉ set (interp-ins (xs @ (oid, Some ref) # ys))
    by (simp add: snoc.IH snoc.prems(3))
  moreover have distinct (map fst (xs @ (oid, Some ref) # ys @ [x]))
   using snoc.prems by (metis append-Cons append-self-conv2 insert-ops-def spec-ops-def)
  hence fst x ≠ oid
    using empty-iff by auto
  moreover have insert-ops ((xs @ (oid, Some ref) # ys) @ [x])
    using snoc.prems by auto
  hence set (interp-ins ((xs @ (oid, Some ref) # ys) @ [x])) =
        set (interp-ins (xs @ (oid, Some ref) # ys)) ∨
        set (interp-ins ((xs @ (oid, Some ref) # ys) @ [x])) =
        set (interp-ins (xs @ (oid, Some ref) # ys)) ∪ {fst x}
    using interp-ins-maybe-grow2 by blast
  ultimately show oid ∉ set (interp-ins ops)
    using snoc.prems(2) by auto
qed

lemma interp-ins-last-None:
  shows oid ∈ set (interp-ins (ops @ [(oid, None)]))
  by (simp add: interp-ins-tail-unfold)

lemma interp-ins-monotonic:
  assumes insert-ops (pre @ suf)
    and oid ∈ set (interp-ins pre)
  shows oid ∈ set (interp-ins (pre @ suf))
  using assms interp-ins-maybe-grow3 by auto

lemma interp-ins-append-non-memb:
  assumes insert-ops (pre @ [(oid, Some ref)] @ suf)
    and ref ∉ set (interp-ins pre)
  shows ref ∉ set (interp-ins (pre @ [(oid, Some ref)] @ suf))
  using assms proof(induction suf rule: List.rev-induct)
  case Nil
  then show ref ∉ set (interp-ins (pre @ [(oid, Some ref)] @ []))
    by (metis append-Nil2 insert-spec-nonex interp-ins-tail-unfold)
next
  case (snoc x xs)
  hence IH: ref ∉ set (interp-ins (pre @ [(oid, Some ref)] @ xs))
    by (metis append-assoc insert-ops-rem-last)
  moreover have ref < oid
```

```
    using insert-ops-ref-older snoc.prems(1) by auto
  moreover have oid < fst x
    using insert-ops-sorted-oids by (metis prod.collapse snoc.prems(1))
  have set (interp-ins ((pre @ [(oid, Some ref)] @ xs) @ [x])) =
        set (interp-ins (pre @ [(oid, Some ref)] @ xs)) ∨
        set (interp-ins ((pre @ [(oid, Some ref)] @ xs) @ [x])) =
        set (interp-ins (pre @ [(oid, Some ref)] @ xs)) ∪ {fst x}
    by (metis (full-types) append.assoc interp-ins-maybe-grow2 snoc.prems(1))
  ultimately show ref ∉ set (interp-ins (pre @ [(oid, Some ref)] @ xs @ [x]))
    using ‹oid < fst x› by auto
qed

lemma interp-ins-append-memb:
  assumes insert-ops (pre @ [(oid, Some ref)] @ suf)
    and ref ∈ set (interp-ins pre)
  shows oid ∈ set (interp-ins (pre @ [(oid, Some ref)] @ suf))
  using assms by (metis UnCI append-assoc insert-spec-set interp-ins-monotonic
      interp-ins-tail-unfold singletonI)

lemma interp-ins-append-forward:
  assumes insert-ops (xs @ ys)
    and oid ∈ set (interp-ins (xs @ ys))
    and oid ∈ set (map fst xs)
  shows oid ∈ set (interp-ins xs)
  using assms proof(induction ys rule: List.rev-induct, simp)
  case (snoc y ys)
  obtain cs ds ref where xs = cs @ (oid, ref) # ds
   by (metis (no-types, lifting) imageE prod.collapse set-map snoc.prems(3) split-list-last)
  hence insert-ops (cs @ [(oid, ref)] @ (ds @ ys) @ [y])
    using snoc.prems(1) by auto
  hence oid < fst y
    using insert-ops-sorted-oids by (metis prod.collapse)
  hence oid ≠ fst y
    by blast
  then show ?case
    using snoc.IH snoc.prems(1) snoc.prems(2) assms(3) inserted-item-ident
    by (metis append-assoc insert-ops-appendD interp-ins-tail-unfold prod.collapse)
qed

lemma interp-ins-find-ref:
  assumes insert-ops (xs @ [(oid, Some ref)] @ ys)
    and ref ∈ set (interp-ins (xs @ [(oid, Some ref)] @ ys))
  shows ∃ r. (ref, r) ∈ set xs
proof −
  have ref < oid
    using assms(1) insert-ops-ref-older by blast
  have ref ∈ set (map fst (xs @ [(oid, Some ref)] @ ys))
    by (meson assms(2) interp-ins-subset subsetCE)
  then obtain x where x-prop: x ∈ set (xs @ [(oid, Some ref)] @ ys) ∧ fst x = ref
```

**by** *fastforce*
**obtain** *xr* **where** *x-pair*: $x = (ref, xr)$
**using** *prod.exhaust-sel x-prop* **by** *blast*
**show** $\exists r.\ (ref, r) \in set\ xs$
**proof**(*cases* $x \in set\ xs$)
**case** *True*
**then show** $\exists r.\ (ref, r) \in set\ xs$
**by** (*metis x-prop prod.collapse*)
**next**
**case** *False*
**hence** $(ref, xr) \in set\ ([(oid, Some\ ref)] @ ys)$
**using** *x-prop x-pair* **by** *auto*
**hence** $(ref, xr) \in set\ ys$
**using** ‹$ref < oid$› *x-prop*
**by** (*metis append-Cons append-self-conv2 fst-conv min.strict-order-iff set-ConsD*)
**then obtain** *as bs* **where** $ys = as @ (ref, xr) \# bs$
**by** (*meson split-list*)
**hence** *insert-ops* $((xs @ [(oid, Some\ ref)] @ as @ [(ref, xr)]) @ bs)$
**using** *assms*(*1*) **by** *auto*
**hence** *insert-ops* $(xs @ [(oid, Some\ ref)] @ as @ [(ref, xr)])$
**using** *insert-ops-appendD* **by** *blast*
**hence** $oid < ref$
**using** *insert-ops-sorted-oids* **by** *auto*
**then show** *?thesis*
**using** ‹$ref < oid$› **by** *force*
**qed**
**qed**

## 4.3 Lemmas about *list-order*

**lemma** *list-order-append*:
**assumes** *insert-ops* $(pre @ suf)$
**and** *list-order pre x y*
**shows** *list-order* $(pre @ suf)\ x\ y$
**by** (*metis Un-iff assms list-order-monotonic insert-ops-appendD set-append subset-code*(*1*))

**lemma** *list-order-insert-ref*:
**assumes** *insert-ops* $(ops @ [(oid, Some\ ref)])$
**and** $ref \in set\ (interp\text{-}ins\ ops)$
**shows** *list-order* $(ops @ [(oid, Some\ ref)])\ ref\ oid$
**proof** −
**have** *interp-ins* $(ops @ [(oid, Some\ ref)]) = insert\text{-}spec\ (interp\text{-}ins\ ops)\ (oid, Some\ ref)$
**by** (*simp add*: *interp-ins-tail-unfold*)
**moreover obtain** *xs ys* **where** *interp-ins ops* $= xs @ [ref] @ ys$
**using** *assms*(*2*) *split-list-first* **by** *fastforce*
**hence** *insert-spec* $(interp\text{-}ins\ ops)\ (oid, Some\ ref) = xs @ [ref] @ [] @ [oid] @ ys$
**using** *assms*(*1*) *insert-after-ref interp-ins-distinct* **by** *fastforce*

**ultimately show** *list-order* (*ops* @ [(*oid*, *Some ref*)]) *ref oid*
**using** *assms*(*1*) *list-orderI* **by** *metis*
**qed**

**lemma** *list-order-insert-none*:
**assumes** *insert-ops* (*ops* @ [(*oid*, *None*)])
**and** $x \in$ *set* (*interp-ins ops*)
**shows** *list-order* (*ops* @ [(*oid*, *None*)]) *oid x*
**proof** −
**have** *interp-ins* (*ops* @ [(*oid*, *None*)]) = *insert-spec* (*interp-ins ops*) (*oid*, *None*)
**by** (*simp add*: *interp-ins-tail-unfold*)
**moreover obtain** *xs ys* **where** *interp-ins ops* = *xs* @ [*x*] @ *ys*
**using** *assms*(*2*) *split-list-first* **by** *fastforce*
**hence** *insert-spec* (*interp-ins ops*) (*oid*, *None*) = [] @ [*oid*] @ *xs* @ [*x*] @ *ys*
**by** *simp*
**ultimately show** *list-order* (*ops* @ [(*oid*, *None*)]) *oid x*
**using** *assms*(*1*) *list-orderI* **by** *metis*
**qed**

**lemma** *list-order-insert-between*:
**assumes** *insert-ops* (*ops* @ [(*oid*, *Some ref*)])
**and** *list-order ops ref x*
**shows** *list-order* (*ops* @ [(*oid*, *Some ref*)]) *oid x*
**proof** −
**have** *interp-ins* (*ops* @ [(*oid*, *Some ref*)]) = *insert-spec* (*interp-ins ops*) (*oid*, *Some ref*)
**by** (*simp add*: *interp-ins-tail-unfold*)
**moreover obtain** *xs ys zs* **where** *interp-ins ops* = *xs* @ [*ref*] @ *ys* @ [*x*] @ *zs*
**using** *assms list-orderE* **by** *blast*
**moreover have** ... = *xs* @ *ref* # (*ys* @ [*x*] @ *zs*)
**by** *simp*
**moreover have** *distinct* (*xs* @ *ref* # (*ys* @ [*x*] @ *zs*))
**using** *assms*(*1*) *calculation* **by** (*metis interp-ins-distinct insert-ops-rem-last*)
**hence** *insert-spec* (*xs* @ *ref* # (*ys* @ [*x*] @ *zs*)) (*oid*, *Some ref*) = *xs* @ *ref* # *oid* # (*ys* @ [*x*] @ *zs*)
**using** *assms*(*1*) *calculation* **by** (*simp add*: *insert-after-ref*)
**moreover have** ... = (*xs* @ [*ref*]) @ [*oid*] @ *ys* @ [*x*] @ *zs*
**by** *simp*
**ultimately show** *list-order* (*ops* @ [(*oid*, *Some ref*)]) *oid x*
**using** *assms*(*1*) *list-orderI* **by** *metis*
**qed**

## 4.4 The *insert-seq* predicate

The predicate *insert-seq start ops* is true iff *ops* is a list of insertion operations that begins by inserting after *start*, and then continues by placing each subsequent insertion directly after its predecessor. This definition models the sequential insertion of text at a particular place in a text document.

**inductive** *insert-seq* :: *'oid option* ⇒ (*'oid* × *'oid option*) *list* ⇒ *bool* **where**
*insert-seq start* [(*oid*, *start*)] |
⟦*insert-seq start* (*list* @ [(*prev*, *ref*)])⟧
⟹ *insert-seq start* (*list* @ [(*prev*, *ref*), (*oid*, *Some prev*)])

**lemma** *insert-seq-nonempty*:
**assumes** *insert-seq start xs*
**shows** *xs* ≠ []
**using** *assms* **by** (*induction rule*: *insert-seq.induct*, *auto*)

**lemma** *insert-seq-hd*:
**assumes** *insert-seq start xs*
**shows** ∃ *oid*. *hd xs* = (*oid*, *start*)
**using** *assms* **by** (*induction rule*: *insert-seq.induct*, *simp*,
*metis append-self-conv2 hd-append2 list.sel*(*1*))

**lemma** *insert-seq-rem-last*:
**assumes** *insert-seq start* (*xs* @ [*x*])
**and** *xs* ≠ []
**shows** *insert-seq start xs*
**using** *assms insert-seq.cases* **by** *fastforce*

**lemma** *insert-seq-butlast*:
**assumes** *insert-seq start xs*
**and** *xs* ≠ [] **and** *xs* ≠ [*last xs*]
**shows** *insert-seq start* (*butlast xs*)
**proof** −
**have** *length xs* > *1*
**by** (*metis One-nat-def Suc-lessI add-0-left append-butlast-last-id append-eq-append-conv*
*append-self-conv2 assms*(*2*) *assms*(*3*) *length-greater-0-conv list.size*(*3*) *list.size*(*4*))
**hence** *butlast xs* ≠ []
**by** (*metis length-butlast less-numeral-extra*(*3*) *list.size*(*3*) *zero-less-diff*)
**then show** *insert-seq start* (*butlast xs*)
**using** *assms* **by** (*metis append-butlast-last-id insert-seq-rem-last*)
**qed**

**lemma** *insert-seq-last-ref*:
**assumes** *insert-seq start* (*xs* @ [(*xi*, *xr*), (*yi*, *yr*)])
**shows** *yr* = *Some xi*
**using** *assms insert-seq.cases* **by** *fastforce*

**lemma** *insert-seq-start-none*:
**assumes** *insert-ops ops*
**and** *insert-seq None xs* **and** *insert-ops xs*
**and** *set xs* ⊆ *set ops*
**shows** ∀ *i* ∈ *set* (*map fst xs*). *i* ∈ *set* (*interp-ins ops*)
**using** *assms* **proof**(*induction xs rule*: *List.rev-induct*, *simp*)
**case** (*snoc x xs*)
**then have** *IH*: ∀ *i* ∈ *set* (*map fst xs*). *i* ∈ *set* (*interp-ins ops*)

**by** (*metis Nil-is-map-conv append-is-Nil-conv insert-ops-appendD insert-seq-rem-last le-supE list.simps*(*3*) *set-append split-list*)
**then show** $\forall\, i \in set\ (map\ fst\ (xs\ @\ [x]))$. $i \in set\ (interp\text{-}ins\ ops)$
**proof**(*cases xs* = []
**case** *True*
**then obtain** *oid* **where** *xs* @ [*x*] = [(*oid*, *None*)]
**using** *insert-seq-hd snoc.prems*(*2*) **by** *fastforce*
**hence** (*oid*, *None*) ∈ *set ops*
**using** *snoc.prems*(*4*) **by** *auto*
**then obtain** *as bs* **where** *ops* = *as* @ (*oid*, *None*) # *bs*
**by** (*meson split-list*)
**hence** *ops* = (*as* @ [(*oid*, *None*)]) @ *bs*
**by** (*simp add*: ‹*ops* = *as* @ (*oid*, *None*) # *bs*›)
**moreover have** *oid* ∈ *set* (*interp-ins* (*as* @ [(*oid*, *None*)]))
**by** (*simp add*: *interp-ins-last-None*)
**ultimately have** *oid* ∈ *set* (*interp-ins ops*)
**using** *interp-ins-monotonic snoc.prems*(*1*) **by** *blast*
**then show** $\forall\, i \in set\ (map\ fst\ (xs\ @\ [x]))$. $i \in set\ (interp\text{-}ins\ ops)$
**using** ‹*xs* @ [*x*] = [(*oid*, *None*)]› **by** *auto*
**next**
**case** *False*
**then obtain** *rest y* **where** *snoc-y*: *xs* = *rest* @ [*y*]
**using** *append-butlast-last-id* **by** *metis*
**obtain** *yi yr xi xr* **where** *yx-pairs*: *y* = (*yi*, *yr*) ∧ *x* = (*xi*, *xr*)
**by** *force*
**then have** *xr* = *Some yi*
**using** *insert-seq-last-ref snoc.prems*(*2*) *snoc-y* **by** *fastforce*
**have** *yi* < *xi*
**using** *insert-ops-sorted-oids snoc-y yx-pairs snoc.prems*(*3*)
**by** (*metis* (*no-types*, *lifting*) *append-eq-append-conv2*)
**have** (*yi*, *yr*) ∈ *set ops* **and** (*xi*, *Some yi*) ∈ *set ops*
**using** *snoc.prems*(*4*) *snoc-y yx-pairs* ‹*xr* = *Some yi*› **by** *auto*
**then obtain** *as bs cs* **where** *ops-split*: *ops* = *as* @ [(*yi*, *yr*)] @ *bs* @ [(*xi*, *Some yi*)] @ *cs*
**using** *insert-ops-split-2* ‹*yi* < *xi*› *snoc.prems*(*1*) **by** *blast*
**hence** *yi* ∈ *set* (*interp-ins* (*as* @ [(*yi*, *yr*)] @ *bs*))
**proof** −
**have** *yi* ∈ *set* (*interp-ins ops*)
**by** (*simp add*: *IH snoc-y yx-pairs*)
**moreover have** *ops* = (*as* @ [(*yi*, *yr*)] @ *bs*) @ ([(*xi*, *Some yi*)] @ *cs*)
**using** *ops-split* **by** *simp*
**moreover have** *yi* ∈ *set* (*map fst* (*as* @ [(*yi*, *yr*)] @ *bs*))
**by** *simp*
**ultimately show** *?thesis*
**using** *snoc.prems*(*1*) *interp-ins-append-forward* **by** *blast*
**qed**
**hence** *xi* ∈ *set* (*interp-ins* ((*as* @ [(*yi*, *yr*)] @ *bs*) @ [(*xi*, *Some yi*)] @ *cs*))
**using** *snoc.prems*(*1*) *interp-ins-append-memb ops-split* **by** *force*
**hence** *xi* ∈ *set* (*interp-ins ops*)

```
    by (simp add: ops-split)
  then show ∀ i ∈ set (map fst (xs @ [x])). i ∈ set (interp-ins ops)
    using IH yx-pairs by auto
 qed
qed

lemma insert-seq-after-start:
 assumes insert-ops ops
   and insert-seq (Some ref) xs and insert-ops xs
   and set xs ⊆ set ops
   and ref ∈ set (interp-ins ops)
 shows ∀ i ∈ set (map fst xs). list-order ops ref i
 using assms proof(induction xs rule: List.rev-induct, simp)
 case (snoc x xs)
 have IH: ∀ i ∈ set (map fst xs). list-order ops ref i
   using snoc.IH snoc.prems insert-seq-rem-last insert-ops-appendD
   by (metis Nil-is-map-conv Un-subset-iff empty-set equals0D set-append)
 moreover have list-order ops ref (fst x)
 proof(cases xs = [])
   case True
   hence snd x = Some ref
     using insert-seq-hd snoc.prems(2) by fastforce
   moreover have x ∈ set ops
     using snoc.prems(4) by auto
   then obtain cs ds where x-split: ops = cs @ x # ds
     by (meson split-list)
   have list-order (cs @ [(fst x, Some ref)]) ref (fst x)
   proof −
     have insert-ops (cs @ [(fst x, Some ref)] @ ds)
       using x-split ‹snd x = Some ref›
       by (metis append-Cons append-self-conv2 prod.collapse snoc.prems(1))
     moreover from this obtain rr where (ref, rr) ∈ set cs
       using interp-ins-find-ref x-split ‹snd x = Some ref› assms(5)
       by (metis (no-types, lifting) append-Cons append-self-conv2 prod.collapse)
     hence ref ∈ set (interp-ins cs)
     proof −
       have ops = cs @ ([(fst x, Some ref)] @ ds)
           by (metis x-split ‹snd x = Some ref› append-Cons append-self-conv2
prod.collapse)
       thus ref ∈ set (interp-ins cs)
      using assms(5) calculation interp-ins-append-forward interp-ins-append-non-memb
by blast
     qed
     ultimately show list-order (cs @ [(fst x, Some ref)]) ref (fst x)
       using list-order-insert-ref by (metis append.assoc insert-ops-appendD)
   qed
   moreover have ops = (cs @ [(fst x, Some ref)]) @ ds
     using calculation x-split
       by (metis append-eq-Cons-conv append-eq-append-conv2 append-self-conv2
```

*prod.collapse*)
    **ultimately show** *list-order ops ref* (*fst x*)
      **using** *list-order-append snoc.prems*(*1*) **by** *blast*
  **next**
    **case** *False*
    **then obtain** *rest y* **where** *snoc-y*: *xs* = *rest* @ [*y*]
      **using** *append-butlast-last-id* **by** *metis*
    **obtain** *yi yr xi xr* **where** *yx-pairs*: *y* = (*yi*, *yr*) ∧ *x* = (*xi*, *xr*)
      **by** *force*
    **then have** *xr* = *Some yi*
      **using** *insert-seq-last-ref snoc.prems*(*2*) *snoc-y* **by** *fastforce*
    **have** *yi* < *xi*
      **using** *insert-ops-sorted-oids snoc-y yx-pairs snoc.prems*(*3*)
      **by** (*metis* (*no-types*, *lifting*) *append-eq-append-conv2*)
    **have** (*yi*, *yr*) ∈ *set ops* **and** (*xi*, *Some yi*) ∈ *set ops*
      **using** *snoc.prems*(*4*) *snoc-y yx-pairs* ‹*xr* = *Some yi*› **by** *auto*
    **then obtain** *as bs cs* **where** *ops-split*: *ops* = *as* @ [(*yi*, *yr*)] @ *bs* @ [(*xi*, *Some yi*)] @ *cs*
      **using** *insert-ops-split-2* ‹*yi* < *xi*› *snoc.prems*(*1*) **by** *blast*
    **have** *list-order ops ref yi*
      **by** (*simp add*: *IH snoc-y yx-pairs*)
    **moreover have** *list-order* (*as* @ [(*yi*, *yr*)] @ *bs* @ [(*xi*, *Some yi*)]) *yi xi*
    **proof** −
      **have** *insert-ops* ((*as* @ [(*yi*, *yr*)] @ *bs* @ [(*xi*, *Some yi*)]) @ *cs*)
        **using** *ops-split snoc.prems*(*1*) **by** *auto*
      **hence** *insert-ops* ((*as* @ [(*yi*, *yr*)] @ *bs*) @ [(*xi*, *Some yi*)])
        **using** *insert-ops-appendD* **by** *fastforce*
      **moreover have** *yi* ∈ *set* (*interp-ins ops*)
        **using** ‹*list-order ops ref yi*› *list-order-memb2* **by** *auto*
      **hence** *yi* ∈ *set* (*interp-ins* (*as* @ [(*yi*, *yr*)] @ *bs*))
        **using** *interp-ins-append-non-memb ops-split snoc.prems*(*1*) **by** *force*
      **ultimately show** *?thesis*
        **using** *list-order-insert-ref* **by** *force*
    **qed**
    **hence** *list-order ops yi xi*
      **by** (*metis append-assoc list-order-append ops-split snoc.prems*(*1*))
    **ultimately show** *list-order ops ref* (*fst x*)
      **using** *list-order-trans snoc.prems*(*1*) *yx-pairs* **by** *auto*
  **qed**
  **ultimately show** ∀ *i* ∈ *set* (*map fst* (*xs* @ [*x*])). *list-order ops ref i*
    **by** *auto*
**qed**

**lemma** *insert-seq-no-start*:
  **assumes** *insert-ops ops*
    **and** *insert-seq* (*Some ref*) *xs* **and** *insert-ops xs*
    **and** *set xs* ⊆ *set ops*
    **and** *ref* ∉ *set* (*interp-ins ops*)
  **shows** ∀ *i* ∈ *set* (*map fst xs*). *i* ∉ *set* (*interp-ins ops*)

**using** *assms* **proof**(*induction xs rule*: *List.rev-induct*, *simp*)
**case** (*snoc x xs*)
**have** *IH*: ∀ *i* ∈ *set* (*map fst xs*). *i* ∉ *set* (*interp-ins ops*)
**using** *snoc.IH snoc.prems insert-seq-rem-last insert-ops-appendD*
**by** (*metis append-is-Nil-conv le-sup-iff list.map-disc-iff set-append split-list-first*)
**obtain** *as bs* **where** *ops* = *as* @ *x* # *bs*
**using** *snoc.prems*(*4*) **by** (*metis split-list last-in-set snoc-eq-iff-butlast subset-code*(*1*))
**have** *fst x* ∉ *set* (*interp-ins ops*)
**proof**(*cases xs* = [])
**case** *True*
**then obtain** *xi* **where** *x* = (*xi*, *Some ref*)
**using** *insert-seq-hd snoc.prems*(*2*) **by** *force*
**moreover have** *ref* ∉ *set* (*interp-ins as*)
**using** *interp-ins-monotonic snoc.prems*(*1*) *snoc.prems*(*5*) ‹*ops* = *as* @ *x* # *bs*›
**by** *blast*
**ultimately have** *xi* ∉ *set* (*interp-ins* (*as* @ [*x*] @ *bs*))
**using** *snoc.prems*(*1*) **by** (*simp add*: *interp-ins-ref-nonex* ‹*ops* = *as* @ *x* # *bs*›)
**then show** *fst x* ∉ *set* (*interp-ins ops*)
**by** (*simp add*: ‹*ops* = *as* @ *x* # *bs*› ‹*x* = (*xi*, *Some ref*)›)
**next**
**case** *xs-nonempty*: *False*
**then obtain** *y* **where** *xs* = (*butlast xs*) @ [*y*]
**by** (*metis append-butlast-last-id*)
**moreover from** *this* **obtain** *yi yr xi xr* **where** *y* = (*yi*, *yr*) ∧ *x* = (*xi*, *xr*)
**by** *fastforce*
**moreover from** *this* **have** *xr* = *Some yi*
**using** *insert-seq.cases snoc.prems*(*2*) *calculation* **by** *fastforce*
**moreover have** *yi* ∉ *set* (*interp-ins ops*)
**using** *IH calculation*
**by** (*metis Nil-is-map-conv fst-conv last-in-set last-map snoc-eq-iff-butlast*)
**hence** *yi* ∉ *set* (*interp-ins as*)
**using** ‹*ops* = *as* @ *x* # *bs*› *interp-ins-monotonic snoc.prems*(*1*) **by** *blast*
**ultimately have** *xi* ∉ *set* (*interp-ins* (*as* @ [*x*] @ *bs*))
**using** *interp-ins-ref-nonex snoc.prems*(*1*) ‹*ops* = *as* @ *x* # *bs*› **by** *fastforce*
**then show** *fst x* ∉ *set* (*interp-ins ops*)
**by** (*simp add*: ‹*ops* = *as* @ *x* # *bs*› ‹*y* = (*yi*, *yr*) ∧ *x* = (*xi*, *xr*)›)
**qed**
**then show** ∀ *i* ∈ *set* (*map fst* (*xs* @ [*x*])). *i* ∉ *set* (*interp-ins ops*)
**using** *IH* **by** *auto*
**qed**

## 4.5 The proof of no interleaving

**lemma** *no-interleaving-ordered*:
**assumes** *insert-ops ops*
**and** *insert-seq start xs* **and** *insert-ops xs*
**and** *insert-seq start ys* **and** *insert-ops ys*
**and** *set xs* ⊆ *set ops* **and** *set ys* ⊆ *set ops*
**and** *distinct* (*map fst xs* @ *map fst ys*)

```
  and fst (hd xs) < fst (hd ys)
  and ⋀r. start = Some r ⟹ r ∈ set (interp-ins ops)
 shows (∀ x ∈ set (map fst xs). ∀ y ∈ set (map fst ys). list-order ops y x) ∧
       (∀ r. start = Some r ⟶ (∀ x ∈ set (map fst xs). list-order ops r x) ∧
                              (∀ y ∈ set (map fst ys). list-order ops r y))
 using assms proof(induction ops arbitrary: xs ys rule: List.rev-induct, simp)
 case (snoc a ops)
 then have insert-ops ops
  using insert-ops-rem-last by auto
 consider (a-in-xs) a ∈ set xs | (a-in-ys) a ∈ set ys | (neither) a ∉ set xs ∧ a ∉
set ys
  by blast
 then show ?case
 proof(cases)
  case a-in-xs
  then have a ∉ set ys
   using snoc.prems(8) by auto
  hence set ys ⊆ set ops
   using snoc.prems(7) DiffE by auto
  from a-in-xs have a = last xs
   using insert-ops-subset-last snoc.prems by blast
  have IH: (∀ x ∈ set (map fst (butlast xs)). ∀ y ∈ set (map fst ys). list-order ops
y x) ∧
          (∀ r. start = Some r ⟶ (∀ x ∈ set (map fst (butlast xs)). list-order ops
r x) ∧
                                 (∀ y ∈ set (map fst      ys). list-order ops r y))
  proof(cases xs = [a])
   case True
    moreover have ∀ r. start = Some r ⟶ (∀ y ∈ set (map fst ys). list-order
ops r y)
     using insert-seq-after-start ‹insert-ops ops› ‹set ys ⊆ set ops› snoc.prems
     by (metis append-Nil2 calculation insert-seq-hd interp-ins-append-non-memb
list.sel(1))
   ultimately show ?thesis by auto
  next
   case xs-longer: False
   from ‹a = last xs› have set (butlast xs) ⊆ set ops
    using snoc.prems by (simp add: distinct-fst subset-butlast)
   moreover have insert-seq start (butlast xs)
   using insert-seq-butlast insert-seq-nonempty xs-longer ‹a = last xs› snoc.prems(2)
by blast
   moreover have insert-ops (butlast xs)
    using snoc.prems(2) snoc.prems(3) insert-ops-appendD
    by (metis append-butlast-last-id insert-seq-nonempty)
   moreover have distinct (map fst (butlast xs) @ map fst ys)
    using distinct-append-butlast1 snoc.prems(8) by blast
   moreover have set ys ⊆ set ops
    using ‹a ∉ set ys› ‹set ys ⊆ set ops› by blast
   moreover have hd (butlast xs) = hd xs
```

```
    by (metis append-butlast-last-id calculation(2) hd-append2 insert-seq-nonempty
snoc.prems(2))
    hence fst (hd (butlast xs)) < fst (hd ys)
      by (simp add: snoc.prems(9))
    moreover have ⋀r. start = Some r ⟹ r ∈ set (interp-ins ops)
    proof −
      fix r
      assume start = Some r
      then obtain xid where hd xs = (xid, Some r)
        using insert-seq-hd snoc.prems(2) by auto
      hence r < xid
     by (metis hd-in-set insert-ops-memb-ref-older insert-seq-nonempty snoc.prems(2)
snoc.prems(3))
      moreover have xid < fst a
      proof −
        have xs = (butlast xs) @ [a]
          using snoc.prems(2) insert-seq-nonempty ‹a = last xs› by fastforce
        moreover have (xid, Some r) ∈ set (butlast xs)
              using ‹hd xs = (xid, Some r)› insert-seq-nonempty list.set-sel(1)
snoc.prems(2)
          by (metis ‹hd (butlast xs) = hd xs› ‹insert-seq start (butlast xs)›)
        hence xid ∈ set (map fst (butlast xs))
          by (metis in-set-zipE zip-map-fst-snd)
        ultimately show ?thesis
          using snoc.prems(3) last-op-greatest by (metis prod.collapse)
      qed
      ultimately have r ≠ fst a
        using dual-order.asym by blast
      thus r ∈ set (interp-ins ops)
       using snoc.prems(1) snoc.prems(10) interp-ins-maybe-grow2 ‹start = Some
r› by blast
    qed
    ultimately show ?thesis
      using ‹insert-ops ops› snoc.IH snoc.prems(4) snoc.prems(5) by blast
  qed
   moreover have x-exists: ∀ x ∈ set (map fst (butlast xs)). x ∈ set (interp-ins
ops)
  proof(cases start)
    case None
    moreover have set (butlast xs) ⊆ set ops
      using ‹a = last xs› distinct-map snoc.prems(6) snoc.prems(8) subset-butlast
by fastforce
    ultimately show ?thesis
      using insert-seq-start-none ‹insert-ops ops› snoc.prems
      by (metis append-butlast-last-id butlast.simps(2) empty-iff empty-set
          insert-ops-rem-last insert-seq-butlast insert-seq-nonempty list.simps(8))
  next
    case (Some a)
    then show ?thesis
```

**using** *IH list-order-memb2* **by** *blast*
**qed**
**moreover have** $\forall\, y \in set\ (map\ fst\ ys).\ list\text{-}order\ (ops\ @\ [a])\ y\ (fst\ a)$
**proof**(*cases xs* = [*a*])
**case** *xs-a*: *True*
**have** *ys* ≠ [] ⟹ *False*
**proof** −
**assume** *ys* ≠ []
**then obtain** *yi* **where** *yi-def*: *ys* = (*yi*, *start*) # (*tl ys*)
**by** (*metis hd-Cons-tl insert-seq-hd snoc.prems*(*4*))
**hence** (*yi*, *start*) ∈ *set ops*
**by** (*metis* ‹*set ys* ⊆ *set ops*› *list.set-intros*(*1*) *subsetCE*)
**hence** *yi* ∈ *set* (*map fst ops*)
**by** *force*
**hence** *yi* < *fst a*
**using** *snoc.prems*(*1*) *last-op-greatest* **by** (*metis prod.collapse*)
**moreover have** *fst a* < *yi*
**by** (*metis yi-def xs-a fst-conv list.sel*(*1*) *snoc.prems*(*9*))
**ultimately show** *False*
**using** *less-not-sym* **by** *blast*
**qed**
**then show** $\forall\, y \in set\ (map\ fst\ ys).\ list\text{-}order\ (ops\ @\ [a])\ y\ (fst\ a)$
**using** *insert-seq-nonempty snoc.prems*(*4*) **by** *blast*
**next**
**case** *xs-longer*: *False*
**hence** *butlast-split*: *butlast xs* = (*butlast* (*butlast xs*)) @ [*last* (*butlast xs*)]
**using** ‹*a* = *last xs*› *insert-seq-butlast insert-seq-nonempty snoc.prems*(*2*) **by** *fastforce*
**hence** *ref-exists*: *fst* (*last* (*butlast xs*)) ∈ *set* (*interp-ins ops*)
**using** *x-exists* **by** (*metis last-in-set last-map map-is-Nil-conv snoc-eq-iff-butlast*)
**moreover from** *butlast-split* **have** *xs* = (*butlast* (*butlast xs*)) @ [*last* (*butlast xs*), *a*]
**by** (*metis* ‹*a* = *last xs*› *append.assoc append-butlast-last-id butlast.simps*(*2*) *insert-seq-nonempty last-ConsL last-ConsR list.simps*(*3*) *snoc.prems*(*2*))
**hence** *snd a* = *Some* (*fst* (*last* (*butlast xs*)))
**using** *snoc.prems*(*2*) *insert-seq-last-ref* **by** (*metis prod.collapse*)
**hence** *list-order* (*ops* @ [*a*]) (*fst* (*last* (*butlast xs*))) (*fst a*)
**using** *list-order-insert-ref ref-exists snoc.prems*(*1*) **by** (*metis prod.collapse*)
**moreover have** $\forall\, y \in set\ (map\ fst\ ys).\ list\text{-}order\ ops\ y\ (fst\ (last\ (butlast\ xs)))$
**by** (*metis IH butlast-split last-in-set last-map map-is-Nil-conv snoc-eq-iff-butlast*)
**hence** $\forall\, y \in set\ (map\ fst\ ys).\ list\text{-}order\ (ops\ @\ [a])\ y\ (fst\ (last\ (butlast\ xs)))$
**using** *list-order-append snoc.prems*(*1*) **by** *blast*
**ultimately show** $\forall\, y \in set\ (map\ fst\ ys).\ list\text{-}order\ (ops\ @\ [a])\ y\ (fst\ a)$
**using** *list-order-trans snoc.prems*(*1*) **by** *blast*
**qed**
**moreover have** *map-fst-xs*: *map fst xs* = *map fst* (*butlast xs*) @ *map fst* [*last xs*]
**by** (*metis append-butlast-last-id insert-seq-nonempty map-append snoc.prems*(*2*))
**hence** *set* (*map fst xs*) = *set* (*map fst* (*butlast xs*)) ∪ {*fst a*}

**by** (*simp add*: ‹*a* = *last xs*›)
**moreover have** ∀ *r*. *start* = *Some r* ⟶ *list-order* (*ops* @ [*a*]) *r* (*fst a*)
**using** *snoc.prems* **by** (*cases start*, *auto simp add*: *insert-seq-after-start* ‹*a* = *last xs*› *map-fst-xs*)
**ultimately show** (∀ *x* ∈ *set* (*map fst xs*). ∀ *y* ∈ *set* (*map fst ys*). *list-order* (*ops* @ [*a*]) *y x*) ∧
(∀ *r*. *start* = *Some r* ⟶ (∀ *x* ∈ *set* (*map fst xs*). *list-order* (*ops* @ [*a*]) *r x*) ∧
(∀ *y* ∈ *set* (*map fst ys*). *list-order* (*ops* @ [*a*]) *r y*))
**using** *snoc.prems*(*1*) **by** (*simp add*: *list-order-append*)
**next**
**case** *a-in-ys*
**then have** *a* ∉ *set xs*
**using** *snoc.prems*(*8*) **by** *auto*
**hence** *set xs* ⊆ *set ops*
**using** *snoc.prems*(*6*) *DiffE* **by** *auto*
**from** *a-in-ys* **have** *a* = *last ys*
**using** *insert-ops-subset-last snoc.prems* **by** *blast*
**have** *IH*: (∀ *x* ∈ *set* (*map fst xs*). ∀ *y* ∈ *set* (*map fst* (*butlast ys*)). *list-order ops y x*) ∧
(∀ *r*. *start* = *Some r* ⟶ (∀ *x* ∈ *set* (*map fst xs*). *list-order ops r x*) ∧
(∀ *y* ∈ *set* (*map fst* (*butlast ys*)). *list-order ops r y*))
**proof**(*cases ys* = [*a*])
**case** *True*
**moreover have** ∀ *r*. *start* = *Some r* ⟶ (∀ *y* ∈ *set* (*map fst xs*). *list-order ops r y*)
**using** *insert-seq-after-start* ‹*insert-ops ops*› ‹*set xs* ⊆ *set ops*› *snoc.prems*
**by** (*metis append-Nil2 calculation insert-seq-hd interp-ins-append-non-memb list.sel*(*1*))
**ultimately show** *?thesis* **by** *auto*
**next**
**case** *ys-longer*: *False*
**from** ‹*a* = *last ys*› **have** *set* (*butlast ys*) ⊆ *set ops*
**using** *snoc.prems* **by** (*simp add*: *distinct-fst subset-butlast*)
**moreover have** *insert-seq start* (*butlast ys*)
**using** *insert-seq-butlast insert-seq-nonempty ys-longer* ‹*a* = *last ys*› *snoc.prems*(*4*) **by** *blast*
**moreover have** *insert-ops* (*butlast ys*)
**using** *snoc.prems*(*4*) *snoc.prems*(*5*) *insert-ops-appendD*
**by** (*metis append-butlast-last-id insert-seq-nonempty*)
**moreover have** *distinct* (*map fst xs* @ *map fst* (*butlast ys*))
**using** *distinct-append-butlast2 snoc.prems*(*8*) **by** *blast*
**moreover have** *set xs* ⊆ *set ops*
**using** ‹*a* ∉ *set xs*› ‹*set xs* ⊆ *set ops*› **by** *blast*
**moreover have** *hd* (*butlast ys*) = *hd ys*
**by** (*metis append-butlast-last-id calculation*(*2*) *hd-append2 insert-seq-nonempty snoc.prems*(*4*))
**hence** *fst* (*hd xs*) < *fst* (*hd* (*butlast ys*))

```
      by (simp add: snoc.prems(9))
    moreover have ⋀r. start = Some r ⟹ r ∈ set (interp-ins ops)
    proof −
      fix r
      assume start = Some r
      then obtain yid where hd ys = (yid, Some r)
        using insert-seq-hd snoc.prems(4) by auto
      hence r < yid
     by (metis hd-in-set insert-ops-memb-ref-older insert-seq-nonempty snoc.prems(4)
snoc.prems(5))
      moreover have yid < fst a
      proof −
        have ys = (butlast ys) @ [a]
          using snoc.prems(4) insert-seq-nonempty ‹a = last ys› by fastforce
        moreover have (yid, Some r) ∈ set (butlast ys)
              using ‹hd ys = (yid, Some r)› insert-seq-nonempty list.set-sel(1)
snoc.prems(2)
          by (metis ‹hd (butlast ys) = hd ys› ‹insert-seq start (butlast ys)›)
        hence yid ∈ set (map fst (butlast ys))
          by (metis in-set-zipE zip-map-fst-snd)
        ultimately show ?thesis
          using snoc.prems(5) last-op-greatest by (metis prod.collapse)
      qed
      ultimately have r ≠ fst a
        using dual-order.asym by blast
      thus r ∈ set (interp-ins ops)
       using snoc.prems(1) snoc.prems(10) interp-ins-maybe-grow2 ‹start = Some
r› by blast
    qed
    ultimately show ?thesis
      using ‹insert-ops ops› snoc.IH snoc.prems(2) snoc.prems(3) by blast
  qed
  moreover have ∀ x ∈ set (map fst xs). list-order (ops @ [a]) (fst a) x
  proof(cases ys = [a])
    case ys-a: True
    then show ∀ x ∈ set (map fst xs). list-order (ops @ [a]) (fst a) x
    proof(cases start)
      case None
      then show ?thesis
        using insert-seq-start-none list-order-insert-none snoc.prems
       by (metis ‹insert-ops ops› ‹set xs ⊆ set ops› fst-conv insert-seq-hd list.sel(1)
ys-a)
    next
      case (Some r)
      moreover from this have ∀ x ∈ set (map fst xs). list-order ops r x
        using IH by blast
      ultimately show ?thesis
        using snoc.prems(1) snoc.prems(4) list-order-insert-between
        by (metis fst-conv insert-seq-hd list.sel(1) ys-a)
```

```
    qed
  next
    case ys-longer: False
    hence butlast-split: butlast ys = (butlast (butlast ys)) @ [last (butlast ys)]
      using ‹a = last ys› insert-seq-butlast insert-seq-nonempty snoc.prems(4) by
fastforce
    moreover from this have ys = (butlast (butlast ys)) @ [last (butlast ys), a]
      by (metis ‹a = last ys› append.assoc append-butlast-last-id butlast.simps(2)
          insert-seq-nonempty last-ConsL last-ConsR list.simps(3) snoc.prems(4))
    hence snd a = Some (fst (last (butlast ys)))
      using snoc.prems(4) insert-seq-last-ref by (metis prod.collapse)
    moreover have ∀ x ∈ set (map fst xs). list-order ops (fst (last (butlast ys))) x
    by (metis IH butlast-split last-in-set last-map map-is-Nil-conv snoc-eq-iff-butlast)
    ultimately show ∀ x ∈ set (map fst xs). list-order (ops @ [a]) (fst a) x
      using list-order-insert-between snoc.prems(1) by (metis prod.collapse)
  qed
  moreover have map-fst-xs: map fst ys = map fst (butlast ys) @ map fst [last
ys]
  by (metis append-butlast-last-id insert-seq-nonempty map-append snoc.prems(4))
  hence set (map fst ys) = set (map fst (butlast ys)) ∪ {fst a}
    by (simp add: ‹a = last ys›)
  moreover have ∀ r. start = Some r ⟶ list-order (ops @ [a]) r (fst a)
    using snoc.prems by (cases start, auto simp add: insert-seq-after-start ‹a =
last ys› map-fst-xs)
  ultimately show (∀ x ∈ set (map fst xs). ∀ y ∈ set (map fst ys). list-order (ops
@ [a]) y x) ∧
      (∀ r. start = Some r ⟶ (∀ x ∈ set (map fst xs). list-order (ops @ [a]) r x)
∧
                          (∀ y ∈ set (map fst ys). list-order (ops @ [a]) r y))
    using snoc.prems(1) by (simp add: list-order-append)
next
  case neither
  hence set xs ⊆ set ops and set ys ⊆ set ops
    using snoc.prems(6) snoc.prems(7) DiffE by auto
  have (∀ r. start = Some r ⟶ r ∈ set (interp-ins ops)) ∨ (xs = [] ∧ ys = [])
  proof(cases xs)
    case Nil
    then show ?thesis using insert-seq-nonempty snoc.prems(2) by blast
  next
    case xs-nonempty: (Cons x xsa)
    have ⋀r. start = Some r ⟹ r ∈ set (interp-ins ops)
    proof −
      fix r
      assume start = Some r
      then obtain xi where x = (xi, Some r)
        using insert-seq-hd xs-nonempty snoc.prems(2) by fastforce
      hence (xi, Some r) ∈ set ops
        using ‹set xs ⊆ set ops› xs-nonempty by auto
      hence r < xi
```

```
      using ‹insert-ops ops› insert-ops-memb-ref-older by blast
    moreover have xi ∈ set (map fst ops)
      using ‹(xi, Some r) ∈ set ops› by force
    hence xi < fst a
      using last-op-greatest snoc.prems(1) by (metis prod.collapse)
    ultimately have fst a ≠ r
      using order.asym by blast
    then show r ∈ set (interp-ins ops)
     using snoc.prems(1) snoc.prems(10) interp-ins-maybe-grow2 ‹start = Some
r› by blast
  qed
  then show ?thesis by blast
 qed
 hence (∀ x ∈ set (map fst xs). ∀ y ∈ set (map fst ys). list-order ops y x) ∧
       (∀ r. start = Some r ⟶ (∀ x ∈ set (map fst xs). list-order ops r x) ∧
                              (∀ y ∈ set (map fst ys). list-order ops r y))
  using snoc.prems snoc.IH ‹set xs ⊆ set ops› ‹set ys ⊆ set ops› by blast
 then show (∀ x ∈ set (map fst xs). ∀ y ∈ set (map fst ys). list-order (ops @ [a])
y x) ∧
       (∀ r. start = Some r ⟶ (∀ x ∈ set (map fst xs). list-order (ops @ [a]) r x)
∧
                              (∀ y ∈ set (map fst ys). list-order (ops @ [a]) r y))
  using snoc.prems(1) by (simp add: list-order-append)
qed
qed
```

Consider an execution that contains two distinct insertion sequences, *xs* and *ys*, that both begin at the same initial position *start*. We prove that, provided the starting element exists, the two insertion sequences are not interleaved. That is, in the final list order, either all insertions by *xs* appear before all insertions by *ys*, or vice versa.

```
theorem no-interleaving:
 assumes insert-ops ops
   and insert-seq start xs and insert-ops xs
   and insert-seq start ys and insert-ops ys
   and set xs ⊆ set ops and set ys ⊆ set ops
   and distinct (map fst xs @ map fst ys)
   and start = None ∨ (∃ r. start = Some r ∧ r ∈ set (interp-ins ops))
 shows (∀ x ∈ set (map fst xs). ∀ y ∈ set (map fst ys). list-order ops x y) ∨
       (∀ x ∈ set (map fst xs). ∀ y ∈ set (map fst ys). list-order ops y x)
proof(cases fst (hd xs) < fst (hd ys))
 case True
 moreover have ⋀r. start = Some r ⟹ r ∈ set (interp-ins ops)
   using assms(9) by blast
 ultimately have ∀ x∈set (map fst xs). ∀ y∈set (map fst ys). list-order ops y x
   using assms no-interleaving-ordered by blast
 then show ?thesis by blast
next
```

**case** *False*
**hence** *fst* (*hd ys*) < *fst* (*hd xs*)
**using** *assms*(*2*) *assms*(*4*) *assms*(*8*) *insert-seq-nonempty distinct-fst-append*
**by** (*metis* (*no-types*, *lifting*) *hd-in-set hd-map list.map-disc-iff map-append neqE*)
**moreover have** *distinct* (*map fst ys* @ *map fst xs*)
**using** *assms*(*8*) *distinct-append-swap* **by** *blast*
**moreover have** ⋀*r*. *start* = *Some r* ⟹ *r* ∈ *set* (*interp-ins ops*)
**using** *assms*(*9*) **by** *blast*
**ultimately have** ∀ *x*∈*set* (*map fst ys*). ∀ *y*∈*set* (*map fst xs*). *list-order ops y x*
**using** *assms no-interleaving-ordered* **by** *blast*
**then show** *?thesis* **by** *blast*
**qed**

For completeness, we also prove what happens if there are two insertion sequences, *xs* and *ys*, but their initial position *start* does not exist. In that case, none of the insertions in *xs* or *ys* take effect.

**theorem** *missing-start-no-insertion*:
**assumes** *insert-ops ops*
**and** *insert-seq* (*Some start*) *xs* **and** *insert-ops xs*
**and** *insert-seq* (*Some start*) *ys* **and** *insert-ops ys*
**and** *set xs* ⊆ *set ops* **and** *set ys* ⊆ *set ops*
**and** *start* ∉ *set* (*interp-ins ops*)
**shows** ∀ *x* ∈ *set* (*map fst xs*) ∪ *set* (*map fst ys*). *x* ∉ *set* (*interp-ins ops*)
**using** *assms insert-seq-no-start* **by** (*metis UnE*)

**end**

# 5 The Replicated Growable Array (RGA)

The RGA algorithm [4] is a replicated list (or collaborative text-editing) algorithm. In this section we prove that RGA satisfies our list specification. The Isabelle/HOL definition of RGA in this section is based on our prior work on formally verifying CRDTs [3, 2].

**theory** *RGA*
**imports** *Insert-Spec*
**begin**

**fun** *insert-body* :: *'oid*::{*linorder*} *list* ⇒ *'oid* ⇒ *'oid list* **where**
*insert-body* [] *e* = [*e*] |
*insert-body* (*x* # *xs*) *e* =
(*if x* < *e then e* # *x* # *xs*
*else x* # *insert-body xs e*)

**fun** *insert-rga* :: *'oid*::{*linorder*} *list* ⇒ (*'oid* × *'oid option*) ⇒ *'oid list* **where**
*insert-rga xs* (*e*, *None*) = *insert-body xs e* |
*insert-rga* [] (*e*, *Some i*) = [] |
*insert-rga* (*x* # *xs*) (*e*, *Some i*) =

```
    (if x = i then
       x # insert-body xs e
     else
       x # insert-rga xs (e, Some i))

definition interp-rga :: ('oid::{linorder} × 'oid option) list ⇒ 'oid list where
  interp-rga ops ≡ foldl insert-rga [] ops
```

## 5.1 Commutativity of *insert-rga*

```
lemma insert-body-set-ins [simp]:
  shows  set (insert-body xs e) = insert e (set xs)
  by (induction xs, auto)

lemma insert-rga-set-ins:
  assumes i ∈ set xs
  shows set (insert-rga xs (oid, Some i)) = insert oid (set xs)
  using assms by (induction xs, auto)

lemma insert-body-commutes:
  shows insert-body (insert-body xs e1) e2 = insert-body (insert-body xs e2) e1
  by (induction xs, auto)

lemma insert-rga-insert-body-commute:
  assumes i2 ≠ Some e1
   shows insert-rga (insert-body xs e1) (e2, i2) = insert-body (insert-rga xs (e2,
i2)) e1
  using assms by (induction xs; cases i2) (auto simp add: insert-body-commutes)

lemma insert-rga-None-commutes:
  assumes i2 ≠ Some e1
  shows insert-rga (insert-rga xs (e1, None)) (e2, i2  ) =
        insert-rga (insert-rga xs (e2, i2  )) (e1, None)
  using assms by (induction xs; cases i2) (auto simp add: insert-body-commutes)

lemma insert-rga-nonexistent:
  assumes i ∉ set xs
  shows insert-rga xs (e, Some i) = xs
  using assms by (induction xs, auto)

lemma insert-rga-Some-commutes:
  assumes i1 ∈ set xs and i2 ∈ set xs
    and e1 ≠ i2 and e2 ≠ i1
  shows insert-rga (insert-rga xs (e1, Some i1)) (e2, Some i2) =
        insert-rga (insert-rga xs (e2, Some i2)) (e1, Some i1)
  using assms proof (induction xs, simp)
  case (Cons a xs)
  then show ?case
    by (cases a = i1; cases a = i2;
```

```
      auto simp add: insert-body-commutes insert-rga-insert-body-commute)
qed

lemma insert-rga-commutes:
  assumes i2 ≠ Some e1 and i1 ≠ Some e2
  shows insert-rga (insert-rga xs (e1, i1)) (e2, i2) =
        insert-rga (insert-rga xs (e2, i2)) (e1, i1)
proof(cases i1)
  case None
  then show ?thesis
    using assms(1) insert-rga-None-commutes by (cases i2, fastforce, blast)
next
  case some-r1: (Some r1)
  then show ?thesis
  proof(cases i2)
    case None
    then show ?thesis
      using assms(2) insert-rga-None-commutes by fastforce
  next
    case some-r2: (Some r2)
    then show ?thesis
    proof(cases r1 ∈ set xs ∧ r2 ∈ set xs)
      case True
      then show ?thesis
        using assms some-r1 some-r2 by (simp add: insert-rga-Some-commutes)
    next
      case False
      then show ?thesis
        using assms some-r1 some-r2
        by (metis insert-iff insert-rga-nonexistent insert-rga-set-ins)
    qed
  qed
qed

lemma insert-body-split:
  shows ∃ p s. xs = p @ s ∧ insert-body xs e = p @ e # s
proof(induction xs, force)
  case (Cons a xs)
  then obtain p s where IH: xs = p @ s ∧ insert-body xs e = p @ e # s
    by blast
  then show ∃ p s. a # xs = p @ s ∧ insert-body (a # xs) e = p @ e # s
  proof(cases a < e)
    case True
    then have a # xs = [] @ (a # p @ s) ∧ insert-body (a # xs) e = [] @ e # (a
# p @ s)
      by (simp add: IH)
    then show ?thesis by blast
  next
    case False
```

```
    then have a # xs = (a # p) @ s ∧ insert-body (a # xs) e = (a # p) @ e # s
      using IH by auto
    then show ?thesis by blast
  qed
qed

lemma insert-between-elements:
  assumes xs = pre @ ref # suf
    and distinct xs
    and ⋀i. i ∈ set xs ⟹ i < e
  shows insert-rga xs (e, Some ref) = pre @ ref # e # suf
  using assms proof(induction xs arbitrary: pre, force)
  case (Cons a xs)
  then show insert-rga (a # xs) (e, Some ref) = pre @ ref # e # suf
  proof(cases pre)
    case pre-nil: Nil
    then have a = ref
      using Cons.prems(1) by auto
    then show ?thesis
      using Cons.prems pre-nil by (cases suf, auto)
  next
    case (Cons b pre')
    then have insert-rga xs (e, Some ref) = pre' @ ref # e # suf
      using Cons.IH Cons.prems by auto
    then show ?thesis
      using Cons.prems(1) Cons.prems(2) local.Cons by auto
  qed
qed

lemma insert-rga-after-ref:
  assumes ∀ x∈set as. a ≠ x
    and insert-body (cs @ ds) e = cs @ e # ds
  shows insert-rga (as @ a # cs @ ds) (e, Some a) = as @ a # cs @ e # ds
  using assms by (induction as; auto)

lemma insert-rga-preserves-order:
  assumes i = None ∨ (∃ i'. i = Some i' ∧ i' ∈ set xs)
    and distinct xs
  shows ∃ pre suf. xs = pre @ suf ∧ insert-rga xs (e, i) = pre @ e # suf
proof(cases i)
  case None
  then show ∃ pre suf. xs = pre @ suf ∧ insert-rga xs (e, i) = pre @ e # suf
    using insert-body-split by auto
next
  case (Some r)
  moreover from this obtain as bs where xs = as @ r # bs ∧ (∀ x ∈ set as. x
≠ r)
    using assms(1) split-list-first by fastforce
  moreover have ∃ cs ds. bs = cs @ ds ∧ insert-body bs e = cs @ e # ds
```

**by** (*simp add*: *insert-body-split*)
**then obtain** *cs ds* **where** *bs* = *cs* @ *ds* ∧ *insert-body bs e* = *cs* @ *e* # *ds*
**by** *blast*
**ultimately have** *xs* = (*as* @ *r* # *cs*) @ *ds* ∧ *insert-rga xs* (*e*, *i*) = (*as* @ *r* # *cs*) @ *e* # *ds*
**using** *insert-rga-after-ref* **by** *fastforce*
**then show** *?thesis* **by** *blast*
**qed**

## 5.2 Lemmas about the *rga-ops* predicate

**definition** *rga-ops* :: ('*oid*::{*linorder*} × '*oid option*) *list* ⇒ *bool* **where**
*rga-ops list* ≡ *crdt-ops list set-option*

**lemma** *rga-ops-rem-last*:
**assumes** *rga-ops* (*xs* @ [*x*])
**shows** *rga-ops xs*
**using** *assms crdt-ops-rem-last rga-ops-def* **by** *blast*

**lemma** *rga-ops-rem-penultimate*:
**assumes** *rga-ops* (*xs* @ [(*i1*, *r1*), (*i2*, *r2*)])
**and** ⋀*r*. *r2* = *Some r* ⟹ *r* ≠ *i1*
**shows** *rga-ops* (*xs* @ [(*i2*, *r2*)])
**using** *assms* **proof** −
**have** *crdt-ops* (*xs* @ [(*i2*, *r2*)]) *set-option*
**using** *assms crdt-ops-rem-penultimate rga-ops-def* **by** *fastforce*
**thus** *rga-ops* (*xs* @ [(*i2*, *r2*)])
**by** (*simp add*: *rga-ops-def*)
**qed**

**lemma** *rga-ops-ref-exists*:
**assumes** *rga-ops* (*pre* @ (*oid*, *Some ref*) # *suf*)
**shows** *ref* ∈ *fst* ‘ *set pre*
**proof** −
**from** *assms* **have** *crdt-ops* (*pre* @ (*oid*, *Some ref*) # *suf*) *set-option*
**by** (*simp add*: *rga-ops-def*)
**moreover have** *set-option* (*Some ref*) = {*ref*}
**by** *simp*
**ultimately show** *ref* ∈ *fst* ‘ *set pre*
**using** *crdt-ops-ref-exists* **by** *fastforce*
**qed**

## 5.3 Lemmas about the *interp-rga* function

**lemma** *interp-rga-tail-unfold*:
**shows** *interp-rga* (*xs*@[*x*]) = *insert-rga* (*interp-rga* (*xs*)) *x*
**by** (*clarsimp simp add*: *interp-rga-def*)

**lemma** *interp-rga-ids*:

```
  assumes rga-ops xs
  shows set (interp-rga xs) = set (map fst xs)
  using assms proof(induction xs rule: List.rev-induct)
  case Nil
  then show set (interp-rga []) = set (map fst [])
    by (simp add: interp-rga-def)
next
  case (snoc x xs)
  hence IH: set (interp-rga xs) = set (map fst xs)
    using rga-ops-rem-last by blast
  obtain xi xr where x-pair: x = (xi, xr) by force
  then show set (interp-rga (xs @ [x])) = set (map fst (xs @ [x]))
  proof(cases xr)
    case None
    then show ?thesis
      using IH x-pair by (clarsimp simp add: interp-rga-def)
  next
    case (Some r)
    moreover from this have r ∈ set (interp-rga xs)
      using IH rga-ops-ref-exists by (metis x-pair list.set-map snoc.prems)
    ultimately have set (interp-rga (xs @ [(xi, xr)])) = insert xi (set (interp-rga
xs))
      by (simp add: insert-rga-set-ins interp-rga-tail-unfold)
    then show set (interp-rga (xs @ [x])) = set (map fst (xs @ [x]))
      using IH x-pair by auto
  qed
qed

lemma interp-rga-distinct:
  assumes rga-ops xs
  shows distinct (interp-rga xs)
  using assms proof(induction xs rule: List.rev-induct)
  case Nil
  then show distinct (interp-rga []) by (simp add: interp-rga-def)
next
  case (snoc x xs)
  hence IH: distinct (interp-rga xs)
    using rga-ops-rem-last by blast
  moreover obtain xi xr where x-pair: x = (xi, xr)
    by force
  moreover from this have xi ∉ set (interp-rga xs)
    using interp-rga-ids crdt-ops-unique-last rga-ops-rem-last
    by (metis rga-ops-def snoc.prems)
  moreover have ∃ pre suf. interp-rga xs = pre@suf ∧
        insert-rga (interp-rga xs) (xi, xr) = pre @ xi # suf
  proof −
    have ⋀r. r ∈ set-option xr ⟹ r ∈ set (map fst xs)
      using crdt-ops-ref-exists rga-ops-def snoc.prems x-pair by fastforce
    hence xr = None ∨ (∃ r. xr = Some r ∧ r ∈ set (map fst xs))
```

**using** *option.set-sel* **by** *blast*
**hence** $xr = None \vee (\exists r.\ xr = Some\ r \wedge r \in set\ (interp\text{-}rga\ xs))$
**using** *interp-rga-ids rga-ops-rem-last snoc.prems* **by** *blast*
**thus** *?thesis*
**using** *IH insert-rga-preserves-order* **by** *blast*
**qed**
**ultimately show** *distinct* (*interp-rga* (*xs* @ [*x*]))
**by** (*metis Un-iff disjoint-insert*(*1*) *distinct.simps*(*2*) *distinct-append*
*interp-rga-tail-unfold list.simps*(*15*) *set-append*)
**qed**

## 5.4 Proof that RGA satisfies the list specification

**lemma** *final-insert*:
**assumes** *set* (*xs* @ [*x*]) = *set* (*ys* @ [*x*])
**and** *rga-ops* (*xs* @ [*x*])
**and** *insert-ops* (*ys* @ [*x*])
**and** *interp-rga xs* = *interp-ins ys*
**shows** *interp-rga* (*xs* @ [*x*]) = *interp-ins* (*ys* @ [*x*])
**proof** −
**obtain** *oid ref* **where** *x-pair*: *x* = (*oid*, *ref*) **by** *force*
**have** *distinct* (*xs* @ [*x*]) **and** *distinct* (*ys* @ [*x*])
**using** *assms crdt-ops-distinct spec-ops-distinct rga-ops-def insert-ops-def* **by** *blast*+
**then have** *set xs* = *set ys*
**using** *assms*(*1*) **by** *force*
**have** *oid-greatest*: $\bigwedge i.\ i \in set\ (interp\text{-}rga\ xs) \Longrightarrow i < oid$
**proof** −
**have** $\bigwedge i.\ i \in set\ (map\ fst\ ys) \Longrightarrow i < oid$
**using** *assms*(*3*) **by** (*simp add*: *spec-ops-id-inc x-pair insert-ops-def*)
**hence** $\bigwedge i.\ i \in set\ (map\ fst\ xs) \Longrightarrow i < oid$
**using** ‹*set xs* = *set ys*› **by** *auto*
**thus** $\bigwedge i.\ i \in set\ (interp\text{-}rga\ xs) \Longrightarrow i < oid$
**using** *assms*(*2*) *interp-rga-ids rga-ops-rem-last* **by** *blast*
**qed**
**thus** *interp-rga* (*xs* @ [*x*]) = *interp-ins* (*ys* @ [*x*])
**proof**(*cases ref*)
**case** *None*
**moreover from** *this* **have** *insert-rga* (*interp-rga xs*) (*oid*, *ref*) = *oid* # *interp-rga xs*
**using** *oid-greatest hd-in-set insert-body.elims insert-body.simps*(*1*)
*insert-rga.simps*(*1*) *list.sel*(*1*) **by** *metis*
**ultimately show** *interp-rga* (*xs* @ [*x*]) = *interp-ins* (*ys* @ [*x*])
**using** *assms*(*4*) **by** (*simp add*: *interp-ins-tail-unfold interp-rga-tail-unfold x-pair*)
**next**
**case** (*Some r*)
**have** $\exists as\ bs.$ *interp-rga xs* = *as* @ *r* # *bs*
**proof** −

```
    have r ∈ set (map fst xs)
      using assms(2) Some by (simp add: rga-ops-ref-exists x-pair)
    hence r ∈ set (interp-rga xs)
      using assms(2) interp-rga-ids rga-ops-rem-last by blast
    thus ?thesis by (simp add: split-list)
  qed
  from this obtain as bs where as-bs: interp-rga xs = as @ r # bs by force
  hence distinct (as @ r # bs)
    by (metis assms(2) interp-rga-distinct rga-ops-rem-last)
  hence insert-rga (as @ r # bs) (oid, Some r) = as @ r # oid # bs
    by (metis as-bs insert-between-elements oid-greatest)
  moreover have insert-spec (as @ r # bs) (oid, Some r) = as @ r # oid # bs
    by (meson ‹distinct (as @ r # bs)› insert-after-ref)
  ultimately show interp-rga (xs @ [x]) = interp-ins (ys @ [x])
    by (metis assms(4) Some as-bs interp-ins-tail-unfold interp-rga-tail-unfold
x-pair)
 qed
qed

lemma interp-rga-reorder:
 assumes rga-ops (pre @ suf @ [(oid, ref)])
   and ⋀i r. (i, Some r) ∈ set suf ⟹ r ≠ oid
   and ⋀r. ref = Some r ⟹ r ∉ fst ‘ set suf
 shows interp-rga (pre @ (oid, ref) # suf) = interp-rga (pre @ suf @ [(oid, ref)])
 using assms proof(induction suf rule: List.rev-induct)
 case Nil
 then show ?case by simp
next
 case (snoc x xs)
 have ref-not-x: ⋀r. ref = Some r ⟹ r ≠ fst x using snoc.prems(3) by auto
 have IH: interp-rga (pre @ (oid, ref) # xs) = interp-rga (pre @ xs @ [(oid, ref)])
 proof −
   have rga-ops ((pre @ xs) @ [x] @ [(oid, ref)])
     using snoc.prems(1) by auto
   moreover have ⋀r. ref = Some r ⟹ r ≠ fst x
     by (simp add: ref-not-x)
   ultimately have rga-ops ((pre @ xs) @ [(oid, ref)])
     using rga-ops-rem-penultimate
     by (metis (no-types, lifting) Cons-eq-append-conv prod.collapse)
   thus ?thesis using snoc by force
 qed
 obtain xi xr where x-pair: x = (xi, xr) by force
 have interp-rga (pre @ (oid, ref) # xs @ [(xi, xr)]) =
       insert-rga (interp-rga (pre @ xs @ [(oid, ref)])) (xi, xr)
   using IH interp-rga-tail-unfold by (metis append.assoc append-Cons)
 moreover have ... = insert-rga (insert-rga (interp-rga (pre @ xs)) (oid, ref)) (xi,
xr)
   using interp-rga-tail-unfold by (metis append-assoc)
 moreover have ... = insert-rga (insert-rga (interp-rga (pre @ xs)) (xi, xr)) (oid,
```

*ref*)
**proof** −
**have** $\bigwedge xrr.\ xr = Some\ xrr \Longrightarrow xrr \neq oid$
**using** *x-pair snoc.prems*(*2*) **by** *auto*
**thus** *?thesis*
**using** *insert-rga-commutes ref-not-x* **by** (*metis fst-conv x-pair*)
**qed**
**moreover have** ... = *interp-rga* (*pre* @ *xs* @ [*x*] @ [(*oid*, *ref*)])
**by** (*metis append-assoc interp-rga-tail-unfold x-pair*)
**ultimately show** *interp-rga* (*pre* @ (*oid*, *ref*) # *xs* @ [*x*]) =
*interp-rga* (*pre* @ (*xs* @ [*x*]) @ [(*oid*, *ref*)])
**by** (*simp add*: *x-pair*)
**qed**

**lemma** *rga-spec-equal*:
**assumes** *set xs* = *set ys*
**and** *insert-ops xs*
**and** *rga-ops ys*
**shows** *interp-ins xs* = *interp-rga ys*
**using** *assms* **proof**(*induction xs arbitrary*: *ys rule*: *rev-induct*)
**case** *Nil*
**then show** *?case* **by** (*simp add*: *interp-rga-def interp-ins-def*)
**next**
**case** (*snoc x xs*)
**hence** *x* ∈ *set ys*
**by** (*metis last-in-set snoc-eq-iff-butlast*)
**from** *this* **obtain** *pre suf* **where** *ys-split*: *ys* = *pre* @ [*x*] @ *suf*
**using** *split-list-first* **by** *fastforce*
**have** *IH*: *interp-ins xs* = *interp-rga* (*pre* @ *suf*)
**proof** −
**have** *crdt-ops* (*pre* @ *suf*) *set-option*
**proof** −
**have** *crdt-ops* (*pre* @ [*x*] @ *suf*) *set-option*
**using** *rga-ops-def snoc.prems*(*3*) *ys-split* **by** *blast*
**thus** *crdt-ops* (*pre* @ *suf*) *set-option*
**using** *crdt-ops-rem-spec snoc.prems ys-split insert-ops-def* **by** *blast*
**qed**
**hence** *rga-ops* (*pre* @ *suf*)
**using** *rga-ops-def* **by** *blast*
**moreover have** *set xs* = *set* (*pre* @ *suf*)
**by** (*metis append-set-rem-last crdt-ops-distinct insert-ops-def rga-ops-def*
*snoc.prems spec-ops-distinct ys-split*)
**ultimately show** *?thesis*
**using** *insert-ops-rem-last ys-split snoc* **by** *metis*
**qed**
**have** *valid-rga*: *rga-ops* (*pre* @ *suf* @ [*x*])
**proof** −
**have** *crdt-ops* (*pre* @ *suf* @ [*x*]) *set-option*
**using** *snoc.prems ys-split*

```
    by (simp add: crdt-ops-reorder-spec insert-ops-def rga-ops-def)
  thus rga-ops (pre @ suf @ [x])
    by (simp add: rga-ops-def)
 qed
 have interp-ins (xs @ [x]) = interp-rga (pre @ suf @ [x])
 proof −
  have set (xs @ [x]) = set (pre @ suf @ [x])
    using snoc.prems(1) ys-split by auto
  thus ?thesis
    using IH snoc.prems(2) valid-rga final-insert append-assoc by metis
 qed
 moreover have ... = interp-rga (pre @ [x] @ suf)
 proof −
  obtain oid ref where x-pair: x = (oid, ref)
    by force
  have ⋀op2 r. op2 ∈ snd ‘ set suf ⟹ r ∈ set-option op2 ⟹ r ≠ oid
    using snoc.prems
   by (simp add: crdt-ops-independent-suf insert-ops-def rga-ops-def x-pair ys-split)
  hence ⋀i r. (i, Some r) ∈ set suf ⟹ r ≠ oid
    by fastforce
  moreover have ⋀r. ref = Some r ⟹ r ∉ fst ‘ set suf
    using crdt-ops-no-future-ref snoc.prems(3) x-pair ys-split
    by (metis option.set-intros rga-ops-def)
  ultimately show interp-rga (pre @ suf @ [x]) = interp-rga (pre @ [x] @ suf)
    using interp-rga-reorder valid-rga x-pair by force
 qed
 ultimately show interp-ins (xs @ [x]) = interp-rga ys
  by (simp add: ys-split)
qed

lemma insert-ops-exist:
 assumes rga-ops xs
 shows ∃ ys. set xs = set ys ∧ insert-ops ys
 using assms by (simp add: crdt-ops-spec-ops-exist insert-ops-def rga-ops-def)

theorem rga-meets-spec:
 assumes rga-ops xs
 shows ∃ ys. set ys = set xs ∧ insert-ops ys ∧ interp-ins ys = interp-rga xs
 using assms rga-spec-equal insert-ops-exist by metis

end
```

# References


[1] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski. Specification and complexity of collaborative text editing. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 259–268, July 2016.

[2] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. A framework for establishing strong eventual consistency for conflict-free replicated data types. *Archive of Formal Proofs*, July 2017.

[3] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(OOPSLA), Oct. 2017.

[4] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.