

# OpSets: Sequential Specifications for Replicated Datatypes

## Proof Document

Martin Kleppmann<sup>1</sup>, Victor B. F. Gomes<sup>1</sup>, Dominic P. Mulligan<sup>2</sup>, and Alastair R. Beresford<sup>1</sup>

<sup>1</sup>Department of Computer Science and Technology, University of Cambridge, UK

<sup>2</sup>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

<b>1</b>	<b>Abstract OpSet</b>	<b>2</b>
1.1	OpSet definition . . . . .	2
1.2	Helper lemmas about lists . . . . .	3
1.3	The <i>spec-ops</i> predicate . . . . .	5
1.4	The <i>crdt-ops</i> predicate . . . . .	12
<b>2</b>	<b>Specifying list insertion</b>	<b>18</b>
2.1	The <i>insert-ops</i> predicate . . . . .	19
2.2	Properties of the <i>insert-spec</i> function . . . . .	20
2.3	Properties of the <i>interp-ins</i> function . . . . .	25
2.4	Equivalence of the two definitions of insertion . . . . .	26
2.5	The <i>list-order</i> predicate . . . . .	32
<b>3</b>	<b>Relationship to Strong List Specification</b>	<b>40</b>
3.1	Lemmas about insertion and deletion . . . . .	42
3.2	Lemmas about interpreting operations . . . . .	47
3.3	Satisfying all conditions of $\mathcal{A}_{\text{strong}}$ . . . . .	51

<b>4</b>	<b>Interleaving of concurrent insertions</b>	<b>54</b>
4.1	Lemmas about <i>insert-ops</i> . . . . .	54
4.2	Lemmas about <i>interp-ins</i> . . . . .	58
4.3	Lemmas about <i>list-order</i> . . . . .	62
4.4	The <i>insert-seq</i> predicate . . . . .	63
4.5	The proof of no interleaving . . . . .	68
<b>5</b>	<b>The Replicated Growable Array (RGA)</b>	<b>76</b>
5.1	Commutativity of <i>insert-rga</i> . . . . .	77
5.2	Lemmas about the <i>rga-ops</i> predicate . . . . .	80
5.3	Lemmas about the <i>interp-rga</i> function . . . . .	80
5.4	Proof that RGA satisfies the list specification . . . . .	82

## 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:  $(oid, op1) \in opset \implies (oid, op2) \in opset \implies op1 = op2$ 
and ref-older:  $(oid, oper) \in opset \implies ref \in deps\ oper \implies ref < oid$ 
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  $set\ (xs\ @\ zs) \subseteq set\ (xs\ @\ ys\ @\ zs)$ 

```

```

by auto

```

```

thus opset (set (xs @ zs)) deps

```

```

using assms opset-subset by blast

```

**qed**

## 1.2 Helper lemmas about lists

Some general-purpose lemmas 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**  $xa = xb \wedge ya = yb$   
**using assms proof** (*induction xs arbitrary: xa xb x*)  
**fix**  $xa xb x$   
**assume**  $\square = xa @ x \# ya$   
**thus**  $xa = xb \wedge ya = yb$   
**by** *auto*

**next**  
**fix**  $a xs xa xb x$   
**assume**  $IH: \bigwedge xa xb x. distinct xs \implies xs = xa @ x \# ya \implies xs = xb @ x \# yb$   
 $\implies xa = xb \wedge ya = yb$   
**and**  $distinct (a \# xs)$  **and**  $a \# xs = xa @ x \# ya$  **and**  $a \# xs = xb @ x \# yb$   
**thus**  $xa = xb \wedge ya = yb$   
**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**  $set ys \subseteq set xs$  **and**  $set zs \subseteq set xs$   
**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**

$$\begin{aligned} \text{spec-ops ops deps} \equiv & (\text{sorted } (\text{map fst ops}) \wedge \text{distinct } (\text{map fst ops}) \wedge \\ & (\forall \text{ oid oper ref. } (\text{oid}, \text{oper}) \in \text{set ops} \wedge \text{ref} \in \text{deps oper} \longrightarrow \text{ref} < \text{oid})) \end{aligned}$$

**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**  $\forall \text{ oid oper ref. } (\text{oid}, \text{oper}) \in \text{set xs} \wedge \text{ref} \in \text{deps oper} \longrightarrow \text{ref} < \text{oid}$

**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  $\forall oid\ oper\ ref. (oid, oper) \in set\ xs \wedge ref \in deps\ oper \longrightarrow ref < oid$   
 using *assms(1) spec-ops-def* **by** *fastforce*  
 hence  $\forall i\ opr\ r. (i, opr) \in set\ (xs\ @\ [(oid, oper)]) \wedge r \in deps\ opr \longrightarrow r < i$   
 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  $\forall i \in set\ (map\ fst\ xs). i < oid$   
 and  $\forall i \in set\ (map\ fst\ ys). oid < i$   
 and  $\forall ref \in deps\ oper. ref < oid$   
 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*)  
 then *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.prem(3) y-pair*)  
 have *yi-biggest:  $\forall i \in set\ (map\ fst\ (xs\ @\ [(oid, oper)]\ @\ ys)). i < yi$*   
**proof** –  
 have  $\forall i \in set\ (map\ fst\ xs). i < yi$   
 using *oid-yi assms(2) less-trans* **by** *blast*  
 moreover have  $\forall i \in set\ (map\ fst\ ys). i < yi$   
 by (*metis UnCI append-assoc map-append set-append snoc.prem(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  $\forall i \text{ opr } r. (i, \text{opr}) \in \text{set } (xs @ [(oid, oper)] @ ys @ [y])$ 
   $\wedge r \in \text{deps opr} \longrightarrow r < i$ 
proof –
  have  $\forall i \text{ opr } r. (i, \text{opr}) \in \text{set } (xs @ [(oid, oper)] @ ys) \wedge r \in \text{deps opr} \longrightarrow r <$ 
i
  by (meson IH spec-ops-def)
moreover have  $\forall \text{ref}. \text{ref} \in \text{deps } yo \longrightarrow \text{ref} < yi$ 
  by (metis spec-ops-ref-less append-is-Nil-conv last-appendR last-in-set last-snoc
list.simps(3) snoc.premis(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  $\notin \text{set } (\text{map fst } xs)$ 
shows  $\exists \text{pre suf}. xs = \text{pre} @ \text{suf} \wedge$ 
   $(\forall i \in \text{set } (\text{map fst } \text{pre}). i < \text{oid}) \wedge$ 
   $(\forall i \in \text{set } (\text{map fst } \text{suf}). \text{oid} < i)$ 
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:  $xs = \text{pre} @ \text{suf} \wedge$ 
   $(\forall a \in \text{set } (\text{map fst } \text{pre}). a < \text{oid}) \wedge$ 
   $(\forall a \in \text{set } (\text{map fst } \text{suf}). \text{oid} < a)$ 
  by (metis UnCI map-append set-append snoc spec-ops-rem-last)

```

```

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

lemma spec-ops-exists-base:
  assumes finite ops
    and  $\bigwedge \text{oid } \text{op1 } \text{op2}. (\text{oid}, \text{op1}) \in \text{ops} \implies (\text{oid}, \text{op2}) \in \text{ops} \implies \text{op1} = \text{op2}$ 
    and  $\bigwedge \text{oid } \text{oper } \text{ref}. (\text{oid}, \text{oper}) \in \text{ops} \implies \text{ref} \in \text{deps } \text{oper} \implies \text{ref} < \text{oid}$ 
  shows  $\exists \text{op-list}. \text{set } \text{op-list} = \text{ops} \wedge \text{spec-ops } \text{op-list } \text{deps}$ 
  using assms proof(induct ops rule: Finite-Set.finite-induct-select)
  case empty
  then show  $\exists \text{op-list}. \text{set } \text{op-list} = \{\} \wedge \text{spec-ops } \text{op-list } \text{deps}$ 
    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)  $\in$  ops - subset
      using select.hyps(1) by auto
    moreover from this have  $\bigwedge \text{op2}. (\text{oid}, \text{op2}) \in \text{ops} \implies \text{op2} = \text{oper}$ 
      using assms(2) by auto
    hence oid  $\notin$  fst ‘ subset
      by (metis (no-types, lifting) DiffD2 select image-iff prod.collapse psubsetD select.hyps(1))
    from this obtain pre suf
      where op-list = pre @ suf

```

```

    and  $\forall i \in \text{set } (\text{map fst pre}). i < \text{oid}$ 
    and  $\forall i \in \text{set } (\text{map fst suf}). \text{oid} < i$ 
    using spec-ops-split calculation by (metis (no-types, lifting) set-map)
    moreover have  $\text{set } (\text{pre } @ \text{[(oid, oper)] } @ \text{suf}) = \text{insert } (\text{oid, oper}) \text{ subset}$ 
    using calculation by auto
    moreover have  $\text{spec-ops } (\text{pre } @ \text{[(oid, oper)] } @ \text{suf}) \text{ deps}$ 
    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  $\exists \text{op-list}. \text{set op-list} = \text{ops} \wedge \text{spec-ops op-list deps}$ 
proof -
  have finite ops
  using assms opset.finite-opset by force
  moreover have  $\bigwedge \text{oid op1 op2}. (\text{oid, op1}) \in \text{ops} \implies (\text{oid, op2}) \in \text{ops} \implies \text{op1} = \text{op2}$ 
  using assms opset.unique-oid by force
  moreover have  $\bigwedge \text{oid oper ref}. (\text{oid, oper}) \in \text{ops} \implies \text{ref} \in \text{deps oper} \implies \text{ref} < \text{oid}$ 
  using assms opset.ref-older by force
  ultimately show  $\exists \text{op-list}. \text{set op-list} = \text{ops} \wedge \text{spec-ops op-list deps}$ 
  by (simp add: spec-ops-exists-base)
qed

```

```

lemma spec-ops-oid-unique:
  assumes spec-ops op-list deps
  and  $(\text{oid, op1}) \in \text{set op-list}$ 
  and  $(\text{oid, op2}) \in \text{set op-list}$ 
  shows  $\text{op1} = \text{op2}$ 
  using assms proof(induction op-list, simp)
  case (Cons x op-list)
  have distinct (map fst (x # op-list))
  using Cons.prem1 spec-ops-def by blast
  hence notin: fst x  $\notin$  set (map fst op-list)
  by simp
  then show  $\text{op1} = \text{op2}$ 
  proof(cases fst x = oid)
    case True
    then show  $\text{op1} = \text{op2}$ 
    using Cons.prem1 notin by (metis Pair-inject in-set-zipE set-ConsD zip-map-fst-snd)
  next
    case False
    then have  $(\text{oid, op1}) \in \text{set op-list}$  and  $(\text{oid, op2}) \in \text{set op-list}$ 
    using Cons.prem1 by auto
    then show  $\text{op1} = \text{op2}$ 
    using Cons.IH Cons.prem1 spec-ops-rem-cons by blast
  qed

```

**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**  $\bigwedge oid\ op1\ op2. (oid, op1) \in set\ op-list \implies (oid, op2) \in set\ op-list \implies op1 = op2$

**using** *assms spec-ops-oid-unique* **by** *fastforce*

**moreover have**  $\bigwedge oid\ oper\ ref. (oid, oper) \in set\ op-list \implies ref \in deps\ oper \implies ref < oid$

**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**  $\bigwedge r. r \in \text{deps } \text{oper} \implies r \in \text{fst } \text{'set } xs \wedge r < \text{oid}$   
**and** *oid*  $\notin \text{fst } \text{'set } xs$   
**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*)  $\in \text{set } xs$   
**and**  $r \in \text{deps } \text{oper}$   
**shows**  $r < \text{oid}$   
**using** *assms* **by** (*induction rule: crdt-ops.induct, auto*)

**lemma** *crdt-ops-ref-less-last*:

**assumes** *crdt-ops* (*xs* @ [(*oid*, *oper*)]) *deps*  
**and**  $r \in \text{deps } \text{oper}$   
**shows**  $r < \text{oid}$   
**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*  $\notin \text{set } (\text{map fst } xs)$   
**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*  $\notin \text{set } (\text{map fst } xs)$   
**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.prem
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  $\bigwedge op2 r. op2 \in snd \text{ ' set } ys \implies r \in deps \text{ } op2 \implies r \neq oid$ 
  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.prem(2) by auto
qed
have crdt-ops (xs @ ys @ [y]) deps
proof -
  have yi  $\notin fst \text{ ' set } (xs @ [(oid, oper)] @ ys)$ 
    by (metis y-pair append-assoc crdt-ops-unique-last set-map snoc.prem(1))
  hence yi  $\notin fst \text{ ' set } (xs @ ys)$ 
    by auto
  moreover have  $\bigwedge r. r \in deps \text{ } yo \implies r \in fst \text{ ' set } (xs @ ys) \wedge r < yi$ 
  proof -
    have  $\bigwedge r. r \in deps \text{ } yo \implies r \neq oid$ 
      using snoc.prem(2) y-pair by fastforce
    moreover have  $\bigwedge r. r \in deps \text{ } yo \implies r \in fst \text{ ' set } (xs @ [(oid, oper)] @ ys)$ 
      by (metis y-pair append-assoc snoc.prem(1) crdt-ops-ref-exists)
    moreover have  $\bigwedge r. r \in deps \text{ } yo \implies r < yi$ 
      using crdt-ops-ref-less snoc.prem(1) y-pair by fastforce
    ultimately show  $\bigwedge r. r \in deps \text{ } yo \implies r \in fst \text{ ' set } (xs @ ys) \wedge r < yi$ 
      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  $\notin fst \text{ ' set } (xs @ ys @ [y])$ 
  using crdt-ops-unique-mid by (metis (no-types, lifting) UnE image-Un image-set set-append snoc.prem(1))
moreover have  $\bigwedge r. r \in deps \text{ } oper \implies r \in fst \text{ ' set } (xs @ ys @ [y])$ 
  using crdt-ops-ref-exists
  by (metis UnCI append-Cons image-Un set-append snoc.prem(1))
moreover have  $\bigwedge r. r \in deps \text{ } oper \implies r < oid$ 

```

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  $\bigwedge op2 r. op2 \in snd \text{ ' set } ys \implies r \in deps\ op2 \implies r \neq oid$   
 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  $\bigwedge op2 r. op2 \in snd \text{ ' set } zs \implies r \in deps\ op2 \implies r \neq oid$   
**proof** –  
 have  $\bigwedge op2 r. op2 \in snd \text{ ' set } xs \implies r \in deps\ op2 \implies r < oid$   
**proof** –  
 from *assms(1)* have  $\bigwedge i. i \in fst \text{ ' set } xs \implies i < oid$   
 using *spec-ops-id-inc* **by** *fastforce*  
 moreover have  $\bigwedge i2 op2 r. (i2, op2) \in set\ xs \implies r \in deps\ op2 \implies r < i2$   
 using *assms(1) spec-ops-ref-less spec-ops-rem-last* **by** *fastforce*  
 ultimately show  $\bigwedge op2 r. op2 \in snd \text{ ' set } xs \implies r \in deps\ op2 \implies r < oid$   
**by** *fastforce*

**qed**

moreover have  $set\ zs \subseteq set\ xs$

**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  $set\ zs \subseteq set\ xs$   
 using *append-subset(2)* **by** *simp*

**qed**

ultimately show  $\bigwedge op2 r. op2 \in snd \text{ ' set } zs \implies r \in deps\ op2 \implies r \neq oid$   
**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  $\bigwedge op2 r. op2 \in snd \text{ ' set } zs \implies r \in deps\ op2 \implies r \neq oid$   
 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  $\bigwedge r. r \in \text{deps } r2 \implies r \neq i1$   
 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*  $\notin$  *set* (*map fst xs*)  
 by *auto*  
 moreover have *crdt-ops* ((*xs* @ [(*i1*, *r1*)] @ [(*i2*, *r2*)] @ *xs*) *deps*  
 using *assms*(1) by *auto*  
 hence  $\bigwedge r. r \in \text{deps } r2 \implies r \in \text{fst } \text{'set } (xs @ [(i1, r1)])$   
 using *crdt-ops-ref-exists* by *metis*  
 hence  $\bigwedge r. r \in \text{deps } r2 \implies r \in \text{set } (map fst xs)$   
 using *assms*(2) by *auto*  
 moreover have  $\bigwedge r. r \in \text{deps } r2 \implies r < i2$   
 using *assms*(1) *crdt-ops-ref-less* by *fastforce*  
 ultimately show *crdt-ops* (*xs* @ [(*i2*, *r2*)] @ *xs*) *deps*  
 by (*simp add: crdt-ops-intro*)  
 qed

**lemma** *crdt-ops-spec-ops-exist*:  
 assumes *crdt-ops xs deps*  
 shows  $\exists ys. \text{set } xs = \text{set } ys \wedge \text{spec-ops } ys \text{ deps}$   
 using *assms* **proof**(*induction xs rule: List.rev-induct*)  
 case *Nil*  
 then show  $\exists ys. \text{set } [] = \text{set } ys \wedge \text{spec-ops } ys \text{ deps}$   
 by (*simp add: spec-ops-empty*)  
**next**  
 case (*snoc x xs*)  
 hence *IH*:  $\exists ys. \text{set } xs = \text{set } ys \wedge \text{spec-ops } ys \text{ deps}$   
 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  $\exists pre\ suf. ys = pre@ suf \wedge$ 
                 $(\forall i \in set\ (map\ fst\ pre). i < oid) \wedge$ 
                 $(\forall i \in set\ (map\ fst\ suf). oid < i)$ 
proof –
  have  $oid \notin set\ (map\ fst\ xs)$ 
    using calculation(3) crdt-ops-unique-last snoc.prem by force
  hence  $oid \notin set\ (map\ fst\ ys)$ 
    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
   $\forall i \in set\ (map\ fst\ pre). i < oid$  and
   $\forall i \in set\ (map\ fst\ suf). oid < i$  by force
moreover have  $set\ (xs @ [(oid, ref)]) = set\ (pre @ [(oid, ref)] @ suf)$ 
  using crdt-ops-distinct calculation snoc.prem by simp
moreover have spec-ops (pre @ [(oid, ref)] @ suf) deps
proof –
  have  $\forall r \in deps\ ref. r < oid$ 
    using calculation(3) crdt-ops-ref-less-last snoc.prem 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  $\exists ys. set\ (xs @ [x]) = set\ ys \wedge spec-ops\ ys\ deps$ 
  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  $\Rightarrow$  ('oid  $\times$  'oid option)  $\Rightarrow$  '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)  $\implies$  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)  $\implies$  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  $\in$  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)  $\in$  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  $\cup$  {oid}
by (induction xs, auto simp add: insert-commute sup-commute)

```

```

lemma insert-spec-set [simp]:
  assumes  $ref \in set\ xs$ 
  shows  $set\ (insert-spec\ xs\ (oid,\ Some\ ref)) = set\ xs \cup \{oid\}$ 
  using assms proof(induction xs)
  assume  $ref \in set\ []$ 
  thus  $set\ (insert-spec\ []\ (oid,\ Some\ ref)) = set\ [] \cup \{oid\}$ 
    by auto
next
  fix  $a\ xs$ 
  assume  $ref \in set\ xs \implies set\ (insert-spec\ xs\ (oid,\ Some\ ref)) = set\ xs \cup \{oid\}$ 
    and  $ref \in set\ (a\#\!xs)$ 
  thus  $set\ (insert-spec\ (a\#\!xs)\ (oid,\ Some\ ref)) = set\ (a\#\!xs) \cup \{oid\}$ 
    by(cases a = ref, auto simp add: insert-commute sup-commute)
qed

```

```

lemma insert-spec-none [simp]:
  assumes  $ref \notin set\ xs$ 
  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  $ref \notin set\ xs \implies insert-spec\ xs\ (oid,\ Some\ ref) = xs$ 
    and  $ref \notin set\ (a\#\!xs)$ 
  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  $\bigwedge x. x \in set\ xs \implies x < oid$ 
    and  $oid \in set\ xs$ 
  shows False
  using assms by blast

```

```

lemma inserted-item-ident:
  assumes  $a \in set\ (insert-spec\ xs\ (e,\ i))$ 
    and  $a \notin set\ xs$ 
  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  $\bigwedge x. x \in \text{set } xs \implies x < \text{oid}$ 
    and  $\text{ref} = \text{Some } r \implies r < \text{oid}$ 
  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  $\implies (\bigwedge x. x \in \text{set } xs \implies x < \text{oid}) \implies \text{distinct (insert-spec xs (oid, ref))}$ 
  and D: distinct (a#xs)
  and L:  $\bigwedge x. x \in \text{set } (a\#xs) \implies x < \text{oid}$ 
  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  $\neq$  oid
    using D L by auto
  moreover have id  $\notin$  set xs
    using D EQ by auto
  moreover have oid  $\notin$  set xs
    using L by auto
  ultimately have id  $\neq$  oid  $\wedge$  id  $\notin$  set xs  $\wedge$  oid  $\notin$  set xs  $\wedge$  distinct xs
    using D by auto
  }
  note T = this
  {
  assume NEQ: a  $\neq$  id
  have 0: a  $\notin$  set (insert-spec xs (oid, Some id))
    using D L by(metis distinct.simps(2) insert-spec.simps(1) insert-spec-none
insert-spec-none)
  insert-spec-set insert-iff list.set(2) not-less-iff-gr-or-eq)
  have 1: distinct xs
    using D by auto
  }

```

```

    have  $\bigwedge x. x \in \text{set } xs \implies x < oid$ 
      using  $L$  by auto
    hence  $\text{distinct } (\text{insert-spec } xs \text{ } (oid, \text{Some } id))$ 
      using  $S \text{ IH}[OF 1]$  by blast
    hence  $a \notin \text{set } (\text{insert-spec } xs \text{ } (oid, \text{Some } id)) \wedge \text{distinct } (\text{insert-spec } xs \text{ } (oid, \text{Some } id))$ 
      using  $0$  by auto
  }
  from  $this \ S \ T$  show  $\text{distinct } (\text{insert-spec } (a \# xs) \text{ } (oid, ref))$ 
    by  $\text{clarsimp}$ 
qed
qed

```

```

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

```

```

lemma insert-somewhere:
  assumes  $ref = \text{None} \vee (ref = \text{Some } r \wedge r \in \text{set } list)$ 
  shows  $\exists xs \ ys. list = xs \ @ \ ys \wedge \text{insert-spec } list \text{ } (oid, ref) = xs \ @ \ oid \ # \ ys$ 
  using assms proof(induction list)
  assume  $ref = \text{None} \vee ref = \text{Some } r \wedge r \in \text{set } []$ 
  thus  $\exists xs \ ys. [] = xs \ @ \ ys \wedge \text{insert-spec } [] \text{ } (oid, ref) = xs \ @ \ oid \ # \ ys$ 
  proof
    assume  $ref = \text{None}$ 
    thus  $\exists xs \ ys. [] = xs \ @ \ ys \wedge \text{insert-spec } [] \text{ } (oid, ref) = xs \ @ \ oid \ # \ ys$ 
      by auto
  next
    assume  $ref = \text{Some } r \wedge r \in \text{set } []$ 
    thus  $\exists xs \ ys. [] = xs \ @ \ ys \wedge \text{insert-spec } [] \text{ } (oid, ref) = xs \ @ \ oid \ # \ ys$ 
      by auto
  qed

```

```

next
  fix  $a \ list$ 
  assume  $1: ref = \text{None} \vee ref = \text{Some } r \wedge r \in \text{set } (a \# list)$ 
  and  $IH: ref = \text{None} \vee ref = \text{Some } r \wedge r \in \text{set } list \implies$ 
     $\exists xs \ ys. list = xs \ @ \ ys \wedge \text{insert-spec } list \text{ } (oid, ref) = xs \ @ \ oid \ # \ ys$ 
  show  $\exists xs \ ys. a \ # \ list = xs \ @ \ ys \wedge \text{insert-spec } (a \ # \ list) \text{ } (oid, ref) = xs \ @ \ oid \ # \ ys$ 
  proof(rule disjE[OF 1])
    assume  $ref = \text{None}$ 
    thus  $\exists xs \ ys. a \ # \ list = xs \ @ \ ys \wedge \text{insert-spec } (a \ # \ list) \text{ } (oid, ref) = xs \ @ \ oid \ # \ ys$ 
      by force
  next
    assume  $ref = \text{Some } r \wedge r \in \text{set } (a \ # \ list)$ 
    hence  $2: r = a \vee r \in \text{set } list$  and  $3: ref = \text{Some } r$ 
      by auto

```

```

show  $\exists xs\ ys.\ a \# list = xs @ ys \wedge insert-spec\ (a \# list)\ (oid, ref) = xs @ oid$ 
# ys
proof(rule disjE[OF 2])
  assume  $r = a$ 
  thus  $\exists xs\ ys.\ a \# list = xs @ ys \wedge insert-spec\ (a \# list)\ (oid, ref) = xs @ oid$ 
# ys
  using  $\exists$  by(metis append-Cons append-Nil insert-spec.simps( $\exists$ ))
next
  assume  $r \in set\ list$ 
  from this obtain  $xs\ ys$ 
  where  $list = xs @ ys \wedge insert-spec\ list\ (oid, ref) = xs @ oid \# ys$ 
  using IH  $\exists$  by auto
  thus  $\exists xs\ ys.\ a \# list = xs @ ys \wedge insert-spec\ (a \# list)\ (oid, ref) = xs @ oid$ 
# ys
  using  $\exists$  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  $insert-spec\ (xs @ ys)\ (oid, ref) = (insert-spec\ xs\ (oid, ref)) @ ys$ 
using assms proof(induction ys rule: rev-induct)
assume  $ref = None \vee ref = Some\ r \wedge r \in set\ xs$ 
thus  $insert-spec\ (xs @ [])\ (oid, ref) = insert-spec\ xs\ (oid, ref) @ []$ 
  by auto
next
  fix  $x\ xsa$ 
  assume IH:  $ref = None \vee ref = Some\ r \wedge r \in set\ xs \implies insert-spec\ (xs @ xsa)$ 
 $(oid, ref) = insert-spec\ xs\ (oid, ref) @ xsa$ 
  and  $ref = None \vee ref = Some\ r \wedge r \in set\ xs$ 
  thus  $insert-spec\ (xs @ xsa @ [x])\ (oid, ref) = insert-spec\ xs\ (oid, ref) @ xsa @$ 
 $[x]$ 
proof(induction xs)
  assume  $ref = None \vee ref = Some\ r \wedge r \in set\ []$ 
  thus  $insert-spec\ ([] @ xsa @ [x])\ (oid, ref) = insert-spec\ []\ (oid, ref) @ xsa @$ 
 $[x]$ 
  by auto
next
  fix  $a\ xs$ 
  assume 1:  $ref = None \vee ref = Some\ r \wedge r \in set\ (a \# xs)$ 
  and 2:  $((ref = None \vee ref = Some\ r \wedge r \in set\ xs \implies insert-spec\ (xs @ xsa)$ 
 $(oid, ref) = insert-spec\ xs\ (oid, ref) @ xsa) \implies$ 
 $ref = None \vee ref = Some\ r \wedge r \in set\ xs \implies insert-spec\ (xs @ xsa @$ 
 $[x])\ (oid, ref) = insert-spec\ xs\ (oid, ref) @ xsa @ [x])$ 
  and 3:  $(ref = None \vee ref = Some\ r \wedge r \in set\ (a \# xs) \implies insert-spec\ ((a$ 
 $\# xs) @ xsa)\ (oid, ref) = insert-spec\ (a \# xs)\ (oid, ref) @ xsa)$ 
  show  $insert-spec\ ((a \# xs) @ xsa @ [x])\ (oid, ref) = insert-spec\ (a \# xs)\ (oid,$ 
 $ref) @ xsa @ [x]$ 

```



```

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)  $\subseteq$  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]))  $\subseteq$  set (map fst (xs @ [x]))
proof(cases ref)
  case None
    then show set (interp-ins (xs @ [x]))  $\subseteq$  set (map fst (xs @ [x]))
      using IH spec x-pair by auto
  next
    case (Some a)
      then show set (interp-ins (xs @ [x]))  $\subseteq$  set (map fst (xs @ [x]))
        using IH spec x-pair by (cases a  $\in$  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  $\forall x \in$  set (map fst xs).  $x <$  oid
  using last-op-greatest snoc.premis by blast
hence  $\forall x \in$  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  $\Rightarrow$  ('oid  $\times$  'oid option) set where
  succ-rel [] = {} |
  succ-rel [head] = {(head, None)} |
  succ-rel (head#x#xs) = {(head, Some x)}  $\cup$  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  $\Rightarrow$  ('oid  $\times$  'oid option) list  $\Rightarrow$  ('oid  $\times$  'oid option) set
where

```

```

  interp-alt head ops  $\equiv$  foldl insert-alt {(head, None)}
    (map ( $\lambda$ x. case x of
      (oid, None)  $\Rightarrow$  (oid, head) |
      (oid, Some ref)  $\Rightarrow$  (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)  $\in$  succ-rel xs
and (a, b2)  $\in$  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  $\notin$  set (x#xs)
using 3 by auto
hence a  $\notin$  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)}  $\subseteq$  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**  $\{(p, n) \in \text{succ-rel } (x \# xs). p \neq x\} = \text{succ-rel } xs$   
**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**  $\text{succ-rel } (x \# xs) = \{(x, y)\} \cup \text{succ-rel } xs$   
**using** *calculation head-notin image-iff by (cases xs, fastforce+)*  
**moreover from this have**  $\bigwedge n. (x, n) \in \text{succ-rel } (x \# xs) \implies n = y$   
**by** *(metis Pair-inject fst-conv head-notin image-eqI insertE insert-is-Un)*  
**hence**  $\{(p, n) \in \text{succ-rel } (x \# xs). p \neq x\} = \text{succ-rel } (x \# xs) - \{(x, y)\}$   
**by** *blast*  
**moreover have**  $\text{succ-rel } (x \# xs) - \{(x, y)\} = \text{succ-rel } xs$   
**using** *image-iff calculation by fastforce*  
**ultimately show**  $\{(p, n) \in \text{succ-rel } (x \# xs). p \neq x\} = \text{succ-rel } xs$   
**by** *simp*  
**qed**

**lemma** *succ-rel-swap-head*:  
**assumes** *distinct (ref # list)*  
**and**  $(ref, n) \in \text{succ-rel } (ref \# list)$   
**shows**  $\text{succ-rel } (oid \# list) = \{(oid, n)\} \cup \text{succ-rel } list$   
**proof**(*cases list*)  
**case** *Nil*  
**then show** *?thesis using assms by auto*  
**next**  
**case** *(Cons a list)*  
**moreover from this have**  $n = \text{Some } a$   
**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**  $a \neq ref$   
**and** *distinct (oid # a # b # list)*  
**shows**  $\text{insert-alt } (\text{succ-rel } (a \# b \# list)) (oid, ref) =$   
 $\{(a, \text{Some } b)\} \cup \text{insert-alt } (\text{succ-rel } (b \# list)) (oid, ref)$   
**proof**(*cases ∃ n. (ref, n) ∈ succ-rel (a # b # list)*)  
**case** *True*  
**hence**  $\text{insert-alt } (\text{succ-rel } (a \# b \# list)) (oid, ref) =$   
 $\{(p, n) \in \text{succ-rel } (a \# b \# list). p \neq ref\} \cup \{(ref, \text{Some } oid)\} \cup$   
 $\{(i, n). i = oid \wedge (ref, n) \in \text{succ-rel } (a \# b \# list)\}$   
**by** *simp*

**moreover have**  $\{(p, n) \in \text{succ-rel } (a \# b \# \text{list}). p \neq \text{ref}\} =$   
 $\{(a, \text{Some } b)\} \cup \{(p, n) \in \text{succ-rel } (b \# \text{list}). p \neq \text{ref}\}$   
**using** *assms(1)* **by auto**  
**moreover have**  $\text{insert-alt } (\text{succ-rel } (b \# \text{list})) (oid, \text{ref}) =$   
 $\{(p, n) \in \text{succ-rel } (b \# \text{list}). p \neq \text{ref}\} \cup \{(\text{ref}, \text{Some } oid)\} \cup$   
 $\{(i, n). i = oid \wedge (\text{ref}, n) \in \text{succ-rel } (b \# \text{list})\}$   
**proof** –  
**have**  $\exists n. (\text{ref}, n) \in \text{succ-rel } (b \# \text{list})$   
**using** *assms(1)* **True by auto**  
**thus** *?thesis* **by simp**  
**qed**  
**moreover have**  $\{(i, n). i = oid \wedge (\text{ref}, n) \in \text{succ-rel } (a \# b \# \text{list})\} =$   
 $\{(i, n). i = oid \wedge (\text{ref}, n) \in \text{succ-rel } (b \# \text{list})\}$   
**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**  $\text{succ-rel } (\text{insert-spec } (\text{ref} \# \text{list}) (oid, \text{Some } \text{ref})) =$   
 $\text{insert-alt } (\text{succ-rel } (\text{ref} \# \text{list})) (oid, \text{ref})$   
**proof** –  
**obtain** *n* **where** *ref-in-rel*:  $(\text{ref}, n) \in \text{succ-rel } (\text{ref} \# \text{list})$   
**by** (*cases list, auto*)  
**moreover from this have**  $\{(p, n) \in \text{succ-rel } (\text{ref} \# \text{list}). p \neq \text{ref}\} = \text{succ-rel } \text{list}$   
**using** *assms succ-rel-rem-head by (metis (mono-tags, lifting))*  
**moreover have**  $\{(i, n). i = oid \wedge (\text{ref}, n) \in \text{succ-rel } (\text{ref} \# \text{list})\} = \{(oid, n)\}$   
**proof** –  
**have**  $\bigwedge nx. (\text{ref}, nx) \in \text{succ-rel } (\text{ref} \# \text{list}) \implies nx = n$   
**using** *assms by (simp add: succ-rel-functional ref-in-rel)*  
**hence**  $\{(i, n) \in \text{succ-rel } (\text{ref} \# \text{list}). i = \text{ref}\} \subseteq \{(\text{ref}, n)\}$   
**by** *blast*  
**moreover have**  $\{(\text{ref}, n)\} \subseteq \{(i, n) \in \text{succ-rel } (\text{ref} \# \text{list}). i = \text{ref}\}$   
**by** (*simp add: ref-in-rel*)  
**ultimately show** *?thesis* **by blast**  
**qed**  
**moreover have**  $\text{insert-alt } (\text{succ-rel } (\text{ref} \# \text{list})) (oid, \text{ref}) =$   
 $\{(p, n) \in \text{succ-rel } (\text{ref} \# \text{list}). p \neq \text{ref}\} \cup \{(\text{ref}, \text{Some } oid)\} \cup$   
 $\{(i, n). i = oid \wedge (\text{ref}, n) \in \text{succ-rel } (\text{ref} \# \text{list})\}$   
**proof** –  
**have**  $\exists n. (\text{ref}, n) \in \text{succ-rel } (\text{ref} \# \text{list})$   
**using** *ref-in-rel* **by blast**  
**thus** *?thesis* **by simp**  
**qed**  
**ultimately have**  $\text{insert-alt } (\text{succ-rel } (\text{ref} \# \text{list})) (oid, \text{ref}) =$   
 $\text{succ-rel } \text{list} \cup \{(\text{ref}, \text{Some } oid)\} \cup \{(oid, n)\}$

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

**lemma** *succ-rel-insert-later:*

**assumes**  $\text{succ-rel } (\text{insert-spec } (b \# list) (oid, \text{Some } ref)) =$   
 $\text{insert-alt } (\text{succ-rel } (b \# list)) (oid, ref)$   
**and**  $a \neq ref$   
**and**  $\text{distinct } (a \# b \# list)$   
**shows**  $\text{succ-rel } (\text{insert-spec } (a \# b \# list) (oid, \text{Some } ref)) =$   
 $\text{insert-alt } (\text{succ-rel } (a \# b \# list)) (oid, ref)$   
**proof** –  
**have**  $\text{succ-rel } (a \# b \# list) = \{(a, \text{Some } b)\} \cup \text{succ-rel } (b \# list)$   
**by simp**  
**moreover have**  $\text{insert-spec } (a \# b \# list) (oid, \text{Some } ref) =$   
 $a \# (\text{insert-spec } (b \# list) (oid, \text{Some } ref))$   
**using** *assms(2)* **by simp**  
**hence**  $\text{succ-rel } (\text{insert-spec } (a \# b \# list) (oid, \text{Some } ref)) =$   
 $\{(a, \text{Some } b)\} \cup \text{succ-rel } (\text{insert-spec } (b \# list) (oid, \text{Some } ref))$   
**by auto**  
**hence**  $\text{succ-rel } (\text{insert-spec } (a \# b \# list) (oid, \text{Some } ref)) =$   
 $\{(a, \text{Some } b)\} \cup \text{insert-alt } (\text{succ-rel } (b \# list)) (oid, ref)$   
**using** *assms(1)* **by auto**  
**moreover have**  $\text{insert-alt } (\text{succ-rel } (a \# b \# list)) (oid, ref) =$   
 $\{(a, \text{Some } b)\} \cup \text{insert-alt } (\text{succ-rel } (b \# list)) (oid, ref)$   
**using** *succ-rel-insert-alt assms(2)* **by auto**  
**ultimately show** *?thesis* **by blast**  
**qed**

**lemma** *succ-rel-insert-Some:*

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

```

      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.prem1 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.prem1 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  $\bigwedge r. \text{Some } r \in \text{snd ' set ops} \implies r \neq \text{head}$ 
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.prem1 by blast
  moreover have set (interp-ins xs)  $\subseteq$  fst ' set xs
    using interp-ins-subset snoc.prem1 by fastforce
  ultimately show distinct (head # interp-ins xs)
    using snoc.prem2 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.prem3 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**

$$\textit{list-order ops } x \ y \equiv \exists \textit{xs ys zs. interp-ins ops} = \textit{xs} @ [x] @ \textit{ys} @ [y] @ \textit{zs}$$

**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  $\exists \textit{xs ys zs. interp-ins ops} = \textit{xs} @ [x] @ \textit{ys} @ [y] @ \textit{zs}$ 
using assms by (auto simp add: list-order-def)

```



```

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

lemma list-order-memb2:
  assumes list-order ops x y
  shows  $y \in \text{set } (\text{interp-ins ops})$ 
  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  $1: \text{interp-ins op-list} = (\text{xxs}@[x]@\text{xys})@(\text{y}\#\text{xzs})$ 
    using assms by (auto simp add: list-order-def interp-ins-def)
  obtain yxs yys yzs where  $2: \text{interp-ins op-list} = \text{yxs}@[y]\#(\text{yys}@[z]@\text{yzs})$ 
    using assms by (auto simp add: list-order-def interp-ins-def)
  have  $3: \text{distinct } (\text{interp-ins op-list})$ 
    using assms interp-ins-distinct by blast
  hence  $\text{xzs} = \text{yys}@[z]@\text{yzs}$ 
    using distinct-list-split[OF 3, OF 2, OF 1] by auto
  hence  $\text{interp-ins op-list} = \text{xxs}@[x]@\text{xys}@[y]@\text{yys}@[z]@\text{yzs}$ 
    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  $\text{rest} = \text{before } @ \text{after}$ 
    and  $\text{ops} = \text{before } @ (\text{oid}, \text{ref}) \# \text{after}$ 
  shows  $\exists \text{xs ys zs. } \text{interp-ins rest} = \text{xs } @ \text{zs} \wedge \text{interp-ins ops} = \text{xs } @ \text{ys } @ \text{zs}$ 
  using assms proof(induction after arbitrary: rest ops rule: List.rev-induct)
  case Nil
  then have  $1: \text{interp-ins ops} = \text{insert-spec } (\text{interp-ins before}) (\text{oid}, \text{ref})$ 
    by (simp add: interp-ins-tail-unfold)
  then show  $\exists \text{xs ys zs. } \text{interp-ins rest} = \text{xs } @ \text{zs} \wedge \text{interp-ins ops} = \text{xs } @ \text{ys } @ \text{zs}$ 
  proof(cases ref)
  case None
  hence  $\text{interp-ins rest} = [] @ (\text{interp-ins before}) \wedge$ 
     $\text{interp-ins ops} = [] @ [\text{oid}] @ (\text{interp-ins before})$ 
    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.premis(3) by auto
  then show ?thesis by blast
next
  case False
  hence interp-ins ops = (interp-ins rest) @ [] @ []
  using insert-spec-nonex 1 Nil.premis(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.premis(3) snoc.premis(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  $\exists xs\ ys\ zs.\ interp-ins\ rest = xs @ zs \wedge interp-ins\ ops = xs @ ys @ zs$ 
  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.premis by auto
  next
  case r-ys
  hence r  $\notin$  set xs and r  $\notin$  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.premis by auto
  next
  case r-zs
  hence r  $\notin$  set xs and r  $\notin$  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.premis 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.premis 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  $\subseteq$  set B and distinct A and distinct B*  
**shows** *length A  $\leq$  length B*  
**using** *assms proof(induction B arbitrary: A)*  
**case** *Nil*  
**then show** *length A  $\leq$  length [] by simp*

**next**

**case** *(Cons a B)*  
**then show** *length A  $\leq$  length (a # B)*  
**proof**(*cases a  $\in$  set A*)  
**case** *True*  
**have** *set (remove1 a A)  $\subseteq$  set B*  
**using** *Cons.prem1 by auto*  
**hence** *length (remove1 a A)  $\leq$  length B*  
**using** *Cons.IH Cons.prem1 by auto*  
**then show** *length A  $\leq$  length (a # B)*  
**by** (*simp add: True length-remove1*)

**next**

**case** *False*  
**hence** *set A  $\subseteq$  set B*  
**using** *Cons.prem1 by auto*  
**hence** *length A  $\leq$  length B*  
**using** *Cons.IH Cons.prem1 by auto*  
**then show** *length A  $\leq$  length (a # B)*  
**by** *simp*

**qed**

**qed**

**lemma** *set-subset-length-eq:*

**assumes** *set A  $\subseteq$  set B and length B  $\leq$  length A*  
**and** *distinct A and distinct B*  
**shows** *set A = set B*

**proof** –

**have** *length A  $\leq$  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  $\subseteq$  set xs*  
**and** *distinct ys and distinct xs*

```

shows  $\exists e. e \in \text{set } xs \wedge e \notin \text{set } ys$ 
using assms proof(induction xs arbitrary: ys)
case Nil
then show  $\exists e. e \in \text{set } [] \wedge e \notin \text{set } ys$ 
  by simp
next
case (Cons a xs)
then show  $\exists e. e \in \text{set } (a \# xs) \wedge e \notin \text{set } ys$ 
proof(cases a \in set ys)
  case True
  have IH:  $\exists e. e \in \text{set } xs \wedge e \notin \text{set } (\text{remove1 } a \text{ } ys)$ 
  proof –
    have  $\text{length } xs - \text{length } (\text{remove1 } a \text{ } ys) = \text{Suc } m$ 
    by (metis Cons.prem1 One-nat-def Suc-pred True diff-Suc-Suc length-Cons
      length-pos-if-in-set length-remove1)
    moreover have  $\text{set } (\text{remove1 } a \text{ } ys) \subseteq \text{set } xs$ 
    using Cons.prem2 by auto
    ultimately show ?thesis
    by (meson Cons.IH Cons.prem3 distinct.simps2 distinct-remove1)
  qed
  moreover have  $\text{set } ys - \{a\} \subseteq \text{set } xs$ 
  using Cons.prem4 by auto
  ultimately show  $\exists e. e \in \text{set } (a \# xs) \wedge e \notin \text{set } ys$ 
  by (metis Cons.prem5 distinct.simps2 in-set-remove1 set-subset-Cons
subsetCE)
  next
  case False
  then show  $\exists e. e \in \text{set } (a \# xs) \wedge e \notin \text{set } ys$ 
  by auto
qed
qed

```

**lemma** *app-length-lt-exists*:

```

assumes  $xsa @ zsa = xs @ ys$ 
and  $\text{length } xsa \leq \text{length } xs$ 
shows  $xsa @ (\text{drop } (\text{length } xsa) \text{ } xs) = xs$ 
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  $\text{set } A \subseteq \text{set } B$ 
and list-order A x y
shows list-order B x y
using assms proof(induction rule: measure-induct-rule[where f=\lambda x. (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.prem1 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.prem3 <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.prem3 subset-Un-eq)
      then show list-order xa x y
      using less.prem4 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.prem3 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.prem2)
        moreover have set A ⊆ set (remove1 e xa)
          by (metis (no-types, lifting) <e ∉ set A> in-set-remove1 less.prem3)
      rev-subsetD subsetI)
      ultimately show ?thesis
      by (simp add: Suc less.IH less.prem1 less.prem4)
    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.prem2 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  $\mathcal{A}_{\text{strong}}$  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  $\mathcal{A}_{\text{strong}}$  specification.

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

An abstract execution  $A = (H, \text{vis})$  belongs to the *strong list specification*  $\mathcal{A}_{\text{strong}}$  if and only if there is a relation  $\text{lo} \subseteq \text{elems}(A) \times \text{elems}(A)$ , called the *list order*, such that:

1. Each event  $e = \text{do}(op, w) \in H$  returns a sequence of elements  $w = a_0 \dots a_{n-1}$ , where  $a_i \in \text{elems}(A)$ , such that
  - (a)  $w$  contains exactly the elements visible to  $e$  that have been inserted, but not deleted:

$$\forall a. a \in w \iff (\text{do}(\text{ins}(a, \_), \_) \leq_{\text{vis}} e) \wedge \neg(\text{do}(\text{del}(a), \_) \leq_{\text{vis}} e).$$

- (b) The order of the elements is consistent with the list order:

$$\forall i, j. (i < j) \implies (a_i, a_j) \in \text{lo}.$$

- (c) Elements are inserted at the specified position: if  $op = \text{ins}(a, k)$ , then  $a = a_{\min\{k, n-1\}}$ .

2. The list order  $\text{lo}$  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  $\mathcal{A}_{\text{strong}}$  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* (λoper.  
 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* (λoper.  
 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* (λoper.  
 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  $\text{set} (\text{map fst} (\text{insertions ops})) \subseteq \text{set} (\text{map fst ops})$ 
proof(induction ops rule: List.rev-induct)
  case Nil
  then show  $\text{set} (\text{map fst} (\text{insertions } [])) \subseteq \text{set} (\text{map fst } [])$ 
    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  $\text{set} (\text{map fst} (\text{insertions} (\text{ops} @ [a]))) \subseteq \text{set} (\text{map fst} (\text{ops} @ [a]))$ 
proof(cases oper)
  case (Insert ref val)
  hence  $\text{insertions} (\text{ops} @ [a]) = \text{insertions ops} @ [(oid, ref)]$ 
    by (simp add: a-pair insertions-last-ins)
  then show ?thesis using snoc.IH a-pair by auto
next
  case (Delete ref)
  hence  $\text{insertions} (\text{ops} @ [a]) = \text{insertions ops}$ 
    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  $\text{set } A \subseteq \text{set } B$ 
  shows  $\text{set} (\text{insertions } A) \subseteq \text{set} (\text{insertions } B)$ 
  using assms proof(induction B arbitrary: A rule: List.rev-induct)
  case Nil
  then show  $\text{set} (\text{insertions } A) \subseteq \text{set} (\text{insertions } [])$ 
    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.prem(2) by blast
  then show  $\text{set} (\text{insertions } A) \subseteq \text{set} (\text{insertions} (\text{ops} @ [a]))$ 
proof(cases a ∈ set A)
  case True
  then obtain as bs where A-split: A = as @ a # bs  $\wedge a \notin \text{set } as$ 
    by (meson split-list-first)
  hence  $\text{remove1 } a A = as @ bs$ 
    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))  $\subseteq$  set (insertions ops)  
 proof –  
   have list-ops (remove1 a A)  
     using snoc.premis(1) list-ops-def spec-ops-remove1 by blast  
   moreover have set (remove1 a A)  $\subseteq$  set ops  
   proof –  
     have distinct A  
       using snoc.premis(1) list-ops-def spec-ops-distinct by blast  
     hence a  $\notin$  set (remove1 a A)  
     by auto  
     moreover have set (ops @ [a]) = set ops  $\cup$  {a}  
     by auto  
     moreover have set (remove1 a A)  $\subseteq$  set A  
     by (simp add: set-remove1-subset)  
     ultimately show set (remove1 a A)  $\subseteq$  set ops  
     using snoc.premis(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))  $\cup$  {(oid, ref)}  
   using as-a-bs as-bs by auto  
   moreover have set (insertions (ops @ [a])) = set (insertions ops)  $\cup$  {(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  $\subseteq$  set ops

```

    using DiffE snoc.prem1 by auto
  hence set (insertions A) ⊆ set (insertions ops)
    using snoc.IH snoc.prem1 ⟨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.prem1 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.prem1 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**  $\text{inserted-ids } (xs @ [(oid, Delete ref)]) = \text{inserted-ids } xs$   
**by** (simp add: inserted-ids-def map-filter-None map-filter-append)

**lemma** *inserted-ids-exist*:  
**shows**  $oid \in \text{set } (\text{inserted-ids } ops) \longleftrightarrow (\exists \text{ ref val. } (oid, Insert \text{ ref val}) \in \text{set } ops)$   
**proof**(*induction ops rule: List.rev-induct*)  
**case Nil**  
**then show**  $oid \in \text{set } (\text{inserted-ids } []) \longleftrightarrow (\exists \text{ ref val. } (oid, Insert \text{ ref val}) \in \text{set } [])$   
**by** (simp add: inserted-ids-def List.map-filter-def)  
**next**  
**case (snoc a ops)**  
**obtain**  $i \text{ oper where } a\text{-pair: } a = (i, \text{oper})$   
**by** *fastforce*  
**then show**  $oid \in \text{set } (\text{inserted-ids } (ops @ [a])) \longleftrightarrow$   
 $(\exists \text{ ref val. } (oid, Insert \text{ ref val}) \in \text{set } (ops @ [a]))$   
**proof**(*cases oper*)  
**case (Insert r v)**  
**moreover from this have**  $\text{inserted-ids } (ops @ [a]) = \text{inserted-ids } ops @ [i]$   
**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**  $\text{inserted-ids } (ops @ [a]) = \text{inserted-ids } ops$   
**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**  $\text{deleted-ids } (xs @ [(oid, Insert \text{ ref val})]) = \text{deleted-ids } xs$   
**by** (simp add: deleted-ids-def map-filter-None map-filter-append)

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

**lemma** *deleted-ids-exist*:  
**shows**  $\text{ref} \in \text{set } (\text{deleted-ids } ops) \longleftrightarrow (\exists i. (i, Delete \text{ ref}) \in \text{set } ops)$   
**proof**(*induction ops rule: List.rev-induct*)  
**case Nil**  
**then show**  $\text{ref} \in \text{set } (\text{deleted-ids } []) \longleftrightarrow (\exists i. (i, Delete \text{ ref}) \in \text{set } [])$   
**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  $ref \in set (deleted-ids (ops @ [a])) \longleftrightarrow (\exists i. (i, Delete\ ref) \in set (ops$ 
 $@ [a]))$ 
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  $\bigwedge ref. ref \in set (deleted-ids\ ops) \implies ref < oid$ 
proof -
  fix ref
  assume  $ref \in set (deleted-ids\ ops)$ 
  then obtain i where in-ops:  $(i, Delete\ ref) \in set\ ops$ 
  using deleted-ids-exist by blast
  have  $ref < i$ 
  proof -
    have  $\bigwedge i\ oper\ r. (i, oper) \in set\ ops \implies r \in list-op-deps\ oper \implies r < i$ 
    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  $\bigwedge i. i \in set (map\ fst\ ops) \implies i < oid$ 
    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  $\longleftrightarrow$  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.premis 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**  $set (inserted-ids (xs @ [x])) - set (deleted-ids (xs @ [x])) =$   
 $set (inserted-ids xs) - (set (deleted-ids xs) \cup \{ref\})$   
**using** *calculation*( $\beta$ ) **by** *auto*  
**moreover have**  $\dots = set (inserted-ids xs) - set (deleted-ids xs) - \{ref\}$   
**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.prem*s(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**  $\langle distinct\ (a\ \#\ xs) \rangle$  *Cons.prem*s(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**  $\langle distinct\ (a\ \#\ xs) \rangle$  *Cons.prem*s(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.prem*s **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.prem*s(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.prem(2) by auto
      hence length (insert-spec xs (oid, Some (xs ! nat))) = Suc (length xs)
        using Cons.IH Cons.prem(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 $\mathcal{A}_{\text{strong}}$

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.  $\mathcal{A}_{\text{strong}}$  uses the visibility relation  $\leq_{\text{vis}}$  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**  $a \in \text{dom } (vals) \longleftrightarrow (\exists \text{ ref val. } (a, \text{Insert ref val}) \in \text{set ops}) \wedge$   
 $(\nexists i. (i, \text{Delete } a) \in \text{set ops})$   
**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  $\subseteq$  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  $\vee$  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  $\text{fst } (\text{interp-ops } (\text{ops } @ [(\text{oid}, \text{ins}]))) = \text{oid} \# \text{fst } (\text{interp-ops } \text{ops})$ 
using  $\text{assms}(2)$   $\text{interp-ops-unfold-last}$ 
by  $(\text{metis } \text{fst-conv } \text{insert-spec.simps}(1) \text{interp-op.simps}(1) \text{prod.collapse})$ 
ultimately show  $?thesis$ 
by  $(\text{simp add: min-k assms}(3))$ 
next
case  $\text{False}$ 
moreover from this have  $k > 0$  and  $\text{fst } (\text{interp-ops } \text{ops}) \neq []$ 
using  $\text{neq0-conv}$  by  $\text{blast+}$ 
from this obtain  $\text{nat}$  where  $k = \text{Suc } \text{nat}$ 
using  $\text{gr0-implies-Suc}$  by  $\text{blast}$ 
hence  $\text{make-insert } (\text{fst } (\text{interp-ops } \text{ops})) \text{ val } k =$ 
 $\text{Insert } (\text{Some } ((\text{fst } (\text{interp-ops } \text{ops})) ! (\text{min } \text{nat } (\text{length } (\text{fst } (\text{interp-ops } \text{ops}))$ 
 $- 1)))) \text{ val}$ 
using  $\text{False}$  by  $(\text{cases } \text{fst } (\text{interp-ops } \text{ops}), \text{auto})$ 
hence  $\text{fst } (\text{interp-ops } (\text{ops } @ [(\text{oid}, \text{ins}]))) =$ 
 $\text{insert-spec } (\text{fst } (\text{interp-ops } \text{ops})) (\text{oid}, \text{Some } ((\text{fst } (\text{interp-ops } \text{ops})) ! (\text{min}$ 
 $\text{nat } (\text{length } (\text{fst } (\text{interp-ops } \text{ops})) - 1))))$ 
by  $(\text{metis } \text{assms}(2) \text{fst-conv } \text{interp-op.simps}(1) \text{interp-ops-unfold-last } \text{prod.collapse})$ 
moreover have  $\text{min } \text{nat } (\text{length } (\text{fst } (\text{interp-ops } \text{ops})) - 1) < \text{length } (\text{fst } (\text{interp-ops}$ 
 $\text{ops}))$ 
by  $(\text{simp add: } \langle \text{fst } (\text{interp-ops } \text{ops}) \neq [] \rangle \text{min.strict-coboundedI2})$ 
moreover have  $\text{distinct } (\text{fst } (\text{interp-ops } \text{ops}))$ 
using  $\text{interp-ops-distinct } \text{list-ops-def } \text{spec-ops-rem-last } \text{assms}(1)$  by  $\text{blast}$ 
moreover have  $\text{length } \text{list} = \text{Suc } (\text{length } (\text{fst } (\text{interp-ops } \text{ops})))$ 
using  $\text{assms}(3)$   $\text{calculation}$  by  $(\text{simp add: } \text{insert-spec-inc-length})$ 
ultimately show  $?thesis$ 
using  $\text{assms } \text{insert-spec-nth-oid}$ 
by  $(\text{metis } \text{Suc-diff-1 } \langle k = \text{Suc } \text{nat} \rangle \text{diff-Suc-1 } \text{length-greater-0-conv } \text{min-Suc-Suc})$ 
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  $\text{list-ops } \text{ops}$ 
and  $\text{list-order } \text{ops } x \ y$ 
and  $\text{list-order } \text{ops } y \ z$ 
shows  $\text{list-order } \text{ops } x \ z$ 
using  $\text{assms } \text{list-order-trans } \text{list-ops-insertions } \text{list-order-equiv}$  by  $\text{blast}$ 

```

**theorem** *list-order-irrefl:*

```

assumes  $\text{list-ops } \text{ops}$ 
shows  $\neg \text{list-order } \text{ops } x \ x$ 
proof  $-$ 
have  $\text{list-order } \text{ops } x \ x \implies \text{False}$ 
proof  $-$ 
assume  $\text{list-order } \text{ops } x \ x$ 
then obtain  $xs \ ys \ zs$  where  $\text{split: } \text{fst } (\text{interp-ops } \text{ops}) = xs @ [x] @ ys @ [x] @$ 

```

```

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 ∈ set (fst (interp-ops ops))
  and y ∈ set (fst (interp-ops ops))
  and x ≠ 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  $(oid, ref) \in set\ ops$ 
shows  $\exists pre\ suf. ops = pre @ [(oid, ref)] @ suf \wedge$ 
 $(\forall i \in set\ (map\ fst\ pre). i < oid) \wedge$ 
 $(\forall i \in set\ (map\ fst\ suf). oid < i)$ 
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  $\forall i \in set\ (map\ fst\ xs). i < oid$ 
using last-op-greatest snoc.prem(1) by blast
ultimately have  $xs @ [x] = xs @ [(oid, ref)] @ [] \wedge$ 
 $(\forall i \in set\ (map\ fst\ xs). i < oid) \wedge$ 
 $(\forall i \in set\ (map\ fst\ []). oid < i)$ 
by auto
then show ?thesis by force
next
case False
hence  $(oid, ref) \in set\ xs$ 
using snoc.prem(2) by auto
from this obtain pre suf where IH: xs = pre @ [(oid, ref)] @ suf  $\wedge$ 
 $(\forall i \in set\ (map\ fst\ pre). i < oid) \wedge$ 
 $(\forall i \in set\ (map\ fst\ suf). oid < i)$ 
using snoc.IH snoc.prem(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.prem(1))
hence  $xi \neq oid$ 
by auto
have xi-max:  $\forall x \in set\ (map\ fst\ (pre @ [(oid, ref)] @ suf)). x < xi$ 
using IH last-op-greatest snoc.prem(1) x-pair by blast
then show ?thesis
proof(cases xi < oid)
case True
moreover from this have  $\forall x \in set\ suf. fst\ x < oid$ 
using xi-max by auto
hence  $suf = []$ 
using IH last-in-set by fastforce
ultimately have  $xs @ [x] = (pre @ [(xi, xr)]) @ [] \wedge$ 
 $(\forall i \in set\ (map\ fst\ ((pre @ [(xi, xr)]))). i < oid) \wedge$ 
 $(\forall i \in set\ (map\ fst\ []). oid < i)$ 
using dual-order.asym xi-max by auto
then show ?thesis by (simp add: IH)
next
case False

```

**hence**  $oid < xi$   
**using**  $\langle xi \neq oid \rangle$  **by** *auto*  
**hence**  $\forall i \in set (map\ fst\ (suf\ @\ [(xi,\ xr)]))$ .  $oid < i$   
**using** *IH map-fst-append1* **by** *auto*  
**hence**  $xs\ @\ [x] = pre\ @\ [(oid,\ ref)]\ @\ (suf\ @\ [(xi,\ xr)]) \wedge$   
 $(\forall i \in set (map\ fst\ pre). i < oid) \wedge$   
 $(\forall i \in set (map\ fst\ (suf\ @\ [(xi,\ xr)]))$ .  $oid < i$   
**by** (*simp add: IH x-pair*)  
**then show** *?thesis* **by** *blast*  
**qed**  
**qed**  
**qed**

**lemma** *insert-ops-split-2*:  
**assumes** *insert-ops ops*  
**and**  $(xid,\ xr) \in set\ ops$   
**and**  $(yid,\ yr) \in set\ ops$   
**and**  $xid < yid$   
**shows**  $\exists as\ bs\ cs. ops = as\ @\ [(xid,\ xr)]\ @\ bs\ @\ [(yid,\ yr)]\ @\ cs \wedge$   
 $(\forall i \in set (map\ fst\ as). i < xid) \wedge$   
 $(\forall i \in set (map\ fst\ bs). xid < i \wedge i < yid) \wedge$   
 $(\forall i \in set (map\ fst\ cs). yid < i)$

**proof** –  
**obtain** *as as1* **where** *x-split*:  $ops = as\ @\ [(xid,\ xr)]\ @\ as1 \wedge$   
 $(\forall i \in set (map\ fst\ as). i < xid) \wedge (\forall i \in set (map\ fst\ as1). xid < i)$   
**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**  $(yid,\ yr) \in set\ as1$   
**using** *x-split assms* **by** *auto*  
**from this obtain** *bs cs* **where** *y-split*:  $as1 = bs\ @\ [(yid,\ yr)]\ @\ cs \wedge$   
 $(\forall i \in set (map\ fst\ bs). i < yid) \wedge (\forall i \in set (map\ fst\ cs). yid < i)$   
**using** *assms insert-ops-split*  $\langle insert-ops as1 \rangle$  **by** *blast*  
**hence**  $ops = as\ @\ [(xid,\ xr)]\ @\ bs\ @\ [(yid,\ yr)]\ @\ cs$   
**using** *x-split* **by** *blast*  
**moreover have**  $\forall i \in set (map\ fst\ bs). xid < i \wedge i < yid$   
**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**  $\bigwedge i. i \in set (map\ fst\ (xs\ @\ [(i1,\ r1)]\ @\ ys)) \implies i < i2$   
**by** (*metis append.assoc assms last-op-greatest*)



**moreover have**  $i1 \in \text{set } (\text{map } \text{fst } (xs @ [(i1, r1)] @ ys))$   
**by** *auto*  
**ultimately show**  $i1 < i2$   
**by** *blast*  
**qed**

**lemma** *insert-ops-subset-last*:  
**assumes** *insert-ops*  $(xs @ [x])$   
**and** *insert-ops*  $ys$   
**and**  $\text{set } ys \subseteq \text{set } (xs @ [x])$   
**and**  $x \in \text{set } ys$   
**shows**  $x = \text{last } ys$   
**using** *assms* **proof** (*induction*  $ys$ , *simp*)  
**case** (*Cons*  $y ys$ )  
**then show**  $x = \text{last } (y \# ys)$   
**proof** (*cases*  $ys = []$ )  
**case** *True*  
**then show**  $x = \text{last } (y \# ys)$   
**using** *Cons.prem* $s(4)$  **by** *auto*  
**next**  
**case** *ys-nonempty*: *False*  
**have**  $x \neq y$   
**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**  $\bigwedge i. i \in \text{set } (\text{map } \text{fst } (y \# mid)) \implies i < li$   
**by** (*metis* *last-op-greatest* *Cons.prem* $s(2)$  *calculation* *append-Cons*)  
**hence**  $\text{fst } y < li$   
**by** *simp*  
**moreover have**  $\bigwedge i. i \in \text{set } (\text{map } \text{fst } xs) \implies i < \text{fst } x$   
**using** *assms(1)* *last-op-greatest* **by** (*metis* *prod.collapse*)  
**hence**  $\bigwedge i. i \in \text{set } (\text{map } \text{fst } (y \# ys)) \implies i \leq \text{fst } x$   
**using** *Cons.prem* $s(3)$  **by** *fastforce*  
**ultimately show**  $x \neq y$   
**by** *fastforce*  
**qed**  
**then show**  $x = \text{last } (y \# ys)$   
**using** *Cons.IH* *Cons.prem* $s$  *insert-ops-rem-cons* *ys-nonempty*  
**by** (*metis* *dual-order.trans* *last-ConsR* *set-ConsD* *set-subset-Cons*)  
**qed**  
**qed**

**lemma** *subset-butlast*:  
**assumes**  $\text{set } xs \subseteq \text{set } (ys @ [y])$   
**and**  $\text{last } xs = y$   
**and** *distinct*  $xs$   
**shows**  $\text{set } (\text{butlast } xs) \subseteq \text{set } ys$

```

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  $\notin$  set (map fst xs @ map fst ys)
  using Cons.prems by auto
moreover have set (map fst (butlast xs))  $\subseteq$  set (map fst xs)
  by (metis in-set-butlastD map-butlast subsetI)
hence set (map fst (butlast xs) @ map fst ys)  $\subseteq$  set (map fst xs @ map fst ys)
  by auto
ultimately have fst a  $\notin$  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  $\notin$  set (map fst xs @ map fst ys)
  using Cons.prems by auto
moreover have set (map fst (butlast ys))  $\subseteq$  set (map fst ys)
  by (metis in-set-butlastD map-butlast subsetI)
hence set (map fst xs @ map fst (butlast ys))  $\subseteq$  set (map fst xs @ map fst ys)
  by auto
ultimately have fst a  $\notin$  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)  $\vee$ 
  set (interp-ins (xs @ [(oid, ref)])) = (set (interp-ins xs)  $\cup$  {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  $set (interp-ins (xs @ [x])) = set (interp-ins xs) \vee$ 
     $set (interp-ins (xs @ [x])) = (set (interp-ins xs) \cup \{fst\ x\})$ 
  using assms interp-ins-maybe-grow by (cases x, auto)

lemma interp-ins-maybe-grow3:
  assumes insert-ops (xs @ ys)
  shows  $\exists A. A \subseteq set (map\ fst\ ys) \wedge set (interp-ins (xs @ ys)) = set (interp-ins$ 
xs)  $\cup A$ 
  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:  $A \subseteq set (map\ fst\ ys) \wedge$ 
     $set (interp-ins (xs @ ys)) = set (interp-ins xs) \cup A$ 
    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  $A \subseteq set (map\ fst (ys @ [x]))$ 
      using IH by auto
    ultimately show ?thesis
      using IH by auto
    next
    case False
    then have  $set (interp-ins (xs @ ys @ [x])) = set (interp-ins (xs @ ys)) \cup \{fst$ 
x $\}$ 
      by (metis append-assoc interp-ins-maybe-grow2 snoc.prems)
    moreover have  $A \cup \{fst\ x\} \subseteq set (map\ fst (ys @ [x]))$ 
      using IH by auto
    ultimately show ?thesis
      using IH Un-assoc by metis
  qed
qed

lemma interp-ins-ref-none:
  assumes insert-ops ops
    and  $ops = xs @ [(oid, Some\ ref)] @ ys$ 
    and  $ref \notin set (interp-ins xs)$ 
  shows  $oid \notin set (interp-ins ops)$ 
  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  $\bigwedge i. i \in set (map\ fst\ xs) \implies i < oid$ 

```

**using** *Nil.prem*s last-op-greatest **by** fastforce  
**hence**  $\bigwedge i. i \in \text{set } (\text{interp-ins } xs) \implies i < oid$   
**by** (*meson interp-ins-subset subsetCE*)  
**ultimately show**  $oid \notin \text{set } (\text{interp-ins } ops)$   
**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*:  $oid \notin \text{set } (\text{interp-ins } (xs @ (oid, Some ref) \# ys))$   
**by** (*simp add: snoc.IH snoc.prem*s(3))  
**moreover have** *distinct* (*map fst* (*xs* @ (*oid*, *Some ref*) # *ys* @ [*x*]))  
**using** *snoc.prem*s **by** (*metis append-Cons append-self-conv2 insert-ops-def spec-ops-def*)  
**hence** *fst* *x*  $\neq oid$   
**using** *empty-iff* **by** auto  
**moreover have** *insert-ops* ((*xs* @ (*oid*, *Some ref*) # *ys*) @ [*x*])  
**using** *snoc.prem*s **by** auto  
**hence**  $\text{set } (\text{interp-ins } ((xs @ (oid, Some ref) \# ys) @ [x])) =$   
 $\text{set } (\text{interp-ins } (xs @ (oid, Some ref) \# ys)) \vee$   
 $\text{set } (\text{interp-ins } ((xs @ (oid, Some ref) \# ys) @ [x])) =$   
 $\text{set } (\text{interp-ins } (xs @ (oid, Some ref) \# ys)) \cup \{fst\ x\}$   
**using** *interp-ins-maybe-grow2* **by** blast  
**ultimately show**  $oid \notin \text{set } (\text{interp-ins } ops)$   
**using** *snoc.prem*s(2) **by** auto  
**qed**

**lemma** *interp-ins-last-None*:  
**shows**  $oid \in \text{set } (\text{interp-ins } (ops @ [(oid, None)]))$   
**by** (*simp add: interp-ins-tail-unfold*)

**lemma** *interp-ins-monotonic*:  
**assumes** *insert-ops* (*pre* @ *suf*)  
**and**  $oid \in \text{set } (\text{interp-ins } pre)$   
**shows**  $oid \in \text{set } (\text{interp-ins } (pre @ suf))$   
**using** *assms interp-ins-maybe-grow3* **by** auto

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

**using** *insert-ops-ref-older snoc.premis(1)* **by** *auto*  
**moreover have**  $oid < fst\ x$   
**using** *insert-ops-sorted-oids* **by** (*metis prod.collapse snoc.premis(1)*)  
**have**  $set\ (interp-ins\ ((pre\ @\ [(oid,\ Some\ ref)])\ @\ xs)\ @\ [x]) =$   
 $set\ (interp-ins\ (pre\ @\ [(oid,\ Some\ ref)])\ @\ xs) \vee$   
 $set\ (interp-ins\ ((pre\ @\ [(oid,\ Some\ ref)])\ @\ xs)\ @\ [x]) =$   
 $set\ (interp-ins\ (pre\ @\ [(oid,\ Some\ ref)])\ @\ xs) \cup \{fst\ x\}$   
**by** (*metis (full-types) append.assoc interp-ins-maybe-grow2 snoc.premis(1)*)  
**ultimately show**  $ref \notin set\ (interp-ins\ (pre\ @\ [(oid,\ Some\ ref)])\ @\ xs\ @\ [x])$   
**using**  $\langle oid < fst\ x \rangle$  **by** *auto*  
**qed**

**lemma** *interp-ins-append-memb*:  
**assumes** *insert-ops (pre @ [(oid, Some ref)] @ suf)*  
**and**  $ref \in set\ (interp-ins\ pre)$   
**shows**  $oid \in set\ (interp-ins\ (pre\ @\ [(oid,\ Some\ ref)])\ @\ suf)$   
**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**  $oid \in set\ (interp-ins\ (xs\ @\ ys))$   
**and**  $oid \in set\ (map\ fst\ xs)$   
**shows**  $oid \in set\ (interp-ins\ xs)$   
**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.premis(3) split-list-last*)  
**hence** *insert-ops (cs @ [(oid, ref)] @ (ds @ ys) @ [y])*  
**using** *snoc.premis(1)* **by** *auto*  
**hence**  $oid < fst\ y$   
**using** *insert-ops-sorted-oids* **by** (*metis prod.collapse*)  
**hence**  $oid \neq fst\ y$   
**by** *blast*  
**then show** *?case*  
**using** *snoc.IH snoc.premis(1) snoc.premis(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**  $ref \in set\ (interp-ins\ (xs\ @\ [(oid,\ Some\ ref)])\ @\ ys)$   
**shows**  $\exists r. (ref,\ r) \in set\ xs$   
**proof** –  
**have**  $ref < oid$   
**using** *assms(1) insert-ops-ref-older* **by** *blast*  
**have**  $ref \in set\ (map\ fst\ (xs\ @\ [(oid,\ Some\ ref)])\ @\ ys)$   
**by** (*meson assms(2) interp-ins-subset subsetCE*)  
**then obtain**  $x$  **where**  $x-prop: x \in set\ (xs\ @\ [(oid,\ Some\ ref)])\ @\ ys \wedge\ fst\ x = ref$

```

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

```

### 4.3 Lemmas about *list-order*

lemma *list-order-append*:

```

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

```

lemma *list-order-insert-ref*:

```

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

```

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

**lemma**  $list\text{-}order\text{-}insert\text{-}none$ :

**assumes**  $insert\text{-}ops (ops @ [(oid, None)])$   
**and**  $x \in set (interp\text{-}ins\ ops)$   
**shows**  $list\text{-}order (ops @ [(oid, None)])\ oid\ x$

**proof** –

**have**  $interp\text{-}ins (ops @ [(oid, None)]) = insert\text{-}spec (interp\text{-}ins\ ops) (oid, None)$   
**by**  $(simp\ add: interp\text{-}ins\text{-}tail\text{-}unfold)$

**moreover obtain**  $xs\ ys$  **where**  $interp\text{-}ins\ ops = xs @ [x] @ ys$

**using**  $assms(2)\ split\text{-}list\text{-}first$  **by**  $fastforce$

**hence**  $insert\text{-}spec (interp\text{-}ins\ ops) (oid, None) = [] @ [oid] @ xs @ [x] @ ys$

**by**  $simp$

**ultimately show**  $list\text{-}order (ops @ [(oid, None)])\ oid\ x$

**using**  $assms(1)\ list\text{-}orderI$  **by**  $metis$

**qed**

**lemma**  $list\text{-}order\text{-}insert\text{-}between$ :

**assumes**  $insert\text{-}ops (ops @ [(oid, Some\ ref)])$

**and**  $list\text{-}order\ ops\ ref\ x$

**shows**  $list\text{-}order (ops @ [(oid, Some\ ref)])\ oid\ x$

**proof** –

**have**  $interp\text{-}ins (ops @ [(oid, Some\ ref)]) = insert\text{-}spec (interp\text{-}ins\ ops) (oid, Some\ ref)$

**by**  $(simp\ add: interp\text{-}ins\text{-}tail\text{-}unfold)$

**moreover obtain**  $xs\ ys\ zs$  **where**  $interp\text{-}ins\ ops = xs @ [ref] @ ys @ [x] @ zs$

**using**  $assms\ list\text{-}orderE$  **by**  $blast$

**moreover have**  $\dots = xs @ ref \# (ys @ [x] @ zs)$

**by**  $simp$

**moreover have**  $distinct (xs @ ref \# (ys @ [x] @ zs))$

**using**  $assms(1)\ calculation$  **by**  $(metis\ interp\text{-}ins\text{-}distinct\ insert\text{-}ops\text{-}rem\text{-}last)$

**hence**  $insert\text{-}spec (xs @ ref \# (ys @ [x] @ zs)) (oid, Some\ ref) = xs @ ref \# oid \# (ys @ [x] @ zs)$

**using**  $assms(1)\ calculation$  **by**  $(simp\ add: insert\text{-}after\text{-}ref)$

**moreover have**  $\dots = (xs @ [ref]) @ [oid] @ ys @ [x] @ zs$

**by**  $simp$

**ultimately show**  $list\text{-}order (ops @ [(oid, Some\ ref)])\ oid\ x$

**using**  $assms(1)\ list\text{-}orderI$  **by**  $metis$

**qed**

#### 4.4 The $insert\text{-}seq$ predicate

The predicate  $insert\text{-}seq\ start\ ops$  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  $\Rightarrow$  ('oid  $\times$  'oid option) list  $\Rightarrow$  bool **where**  
*insert-seq start* [(oid, start)] |  
 [[*insert-seq start* (list @ [(prev, ref)])]]  
 $\Rightarrow$  *insert-seq start* (list @ [(prev, ref), (oid, Some prev)])

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

**lemma** *insert-seq-hd*:  
**assumes** *insert-seq start xs*  
**shows**  $\exists$  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**  $xs \neq []$   
**shows** *insert-seq start xs*  
**using** *assms* *insert-seq.cases* **by** *fastforce*

**lemma** *insert-seq-butlast*:  
**assumes** *insert-seq start xs*  
**and**  $xs \neq []$  **and**  $xs \neq [last\ xs]$   
**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*  $\neq []$   
**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**  $set\ xs \subseteq set\ ops$   
**shows**  $\forall i \in set\ (map\ fst\ xs). i \in set\ (interp-ins\ ops)$   
**using** *assms* **proof**(induction *xs* rule: *List.rev-induct*, *simp*)  
**case** (*snoc x xs*)  
**then have** *IH*:  $\forall i \in set\ (map\ fst\ xs). i \in set\ (interp-ins\ ops)$



**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**  $\forall i \in \text{set } (\text{map fst } (xs @ [x])). i \in \text{set } (\text{interp-ins ops})$

**proof**(*cases xs = []*)

**case** *True*

**then obtain** *oid* **where**  $xs @ [x] = [(oid, None)]$

**using** *insert-seq-hd snoc.prem(2)* **by** *fastforce*

**hence**  $(oid, None) \in \text{set ops}$

**using** *snoc.prem(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**  $oid \in \text{set } (\text{interp-ins } (as @ [(oid, None)]))$

**by** (*simp add: interp-ins-last-None*)

**ultimately have**  $oid \in \text{set } (\text{interp-ins ops})$

**using** *interp-ins-monotonic snoc.prem(1)* **by** *blast*

**then show**  $\forall i \in \text{set } (\text{map fst } (xs @ [x])). i \in \text{set } (\text{interp-ins ops})$

**using**  $\langle xs @ [x] = [(oid, None)] \rangle$  **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**  $yx\text{-pairs}: y = (yi, yr) \wedge x = (xi, xr)$

**by** *force*

**then have**  $xr = \text{Some } yi$

**using** *insert-seq-last-ref snoc.prem(2) snoc-y* **by** *fastforce*

**have**  $yi < xi$

**using** *insert-ops-sorted-oids snoc-y yx-pairs snoc.prem(3)*

**by** (*metis (no-types, lifting) append-eq-append-conv2*)

**have**  $(yi, yr) \in \text{set ops}$  **and**  $(xi, \text{Some } yi) \in \text{set ops}$

**using** *snoc.prem(4) snoc-y yx-pairs <xr = Some yi>* **by** *auto*

**then obtain** *as bs cs* **where**  $ops\text{-split}: ops = as @ [(yi, yr)] @ bs @ [(xi, \text{Some } yi)] @ cs$

**using** *insert-ops-split-2 <yi < xi> snoc.prem(1)* **by** *blast*

**hence**  $yi \in \text{set } (\text{interp-ins } (as @ [(yi, yr)] @ bs))$

**proof** –

**have**  $yi \in \text{set } (\text{interp-ins ops})$

**by** (*simp add: IH snoc-y yx-pairs*)

**moreover have**  $ops = (as @ [(yi, yr)] @ bs) @ [(xi, \text{Some } yi)] @ cs$

**using** *ops-split* **by** *simp*

**moreover have**  $yi \in \text{set } (\text{map fst } (as @ [(yi, yr)] @ bs))$

**by** *simp*

**ultimately show** *?thesis*

**using** *snoc.prem(1) interp-ins-append-forward* **by** *blast*

**qed**

**hence**  $xi \in \text{set } (\text{interp-ins } ((as @ [(yi, yr)] @ bs) @ [(xi, \text{Some } yi)] @ cs))$

**using** *snoc.prem(1) interp-ins-append-memb ops-split* **by** *force*

**hence**  $xi \in \text{set } (\text{interp-ins ops})$

```

    by (simp add: ops-split)
  then show  $\forall i \in \text{set } (\text{map fst } (xs @ [x])). i \in \text{set } (\text{interp-ins ops})$ 
    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  $\subseteq$  set ops
    and ref  $\in$  set (interp-ins ops)
  shows  $\forall i \in \text{set } (\text{map fst } xs). \text{list-order ops ref } i$ 
  using assms proof (induction xs rule: List.rev-induct, simp)
  case (snoc x xs)
  have IH:  $\forall i \in \text{set } (\text{map fst } xs). \text{list-order ops ref } i$ 
    using snoc.IH snoc.prem1 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.prem2 by fastforce
    moreover have x  $\in$  set ops
      using snoc.prem4 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  $\langle \text{snd } x = \text{Some ref} \rangle$ 
        by (metis append-Cons append-self-conv2 prod.collapse snoc.prem1)
      moreover from this obtain rr where (ref, rr)  $\in$  set cs
      using interp-ins-find-ref x-split  $\langle \text{snd } x = \text{Some ref} \rangle$  assms(5)
      by (metis (no-types, lifting) append-Cons append-self-conv2 prod.collapse)
      hence ref  $\in$  set (interp-ins cs)
    proof -
      have ops = cs @ [(fst x, Some ref)] @ ds
        by (metis x-split  $\langle \text{snd } x = \text{Some ref} \rangle$  append-Cons append-self-conv2
        prod.collapse)
      thus ref  $\in$  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.prem(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.prem(2) snoc-y by fastforce
have yi < xi
  using insert-ops-sorted-oids snoc-y yx-pairs snoc.prem(3)
  by (metis (no-types, lifting) append-eq-append-conv2)
have (yi, yr) ∈ set ops and (xi, Some yi) ∈ set ops
  using snoc.prem(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.prem(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.prem(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.prem(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.prem(1))
ultimately show list-order ops ref (fst x)
  using list-order-trans snoc.prem(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:  $\forall i \in \text{set } (\text{map fst } xs). i \notin \text{set } (\text{interp-ins } ops)$ 
  using snoc.IH snoc.prem1 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.prem4 by (metis split-list last-in-set snoc-eq-iff-butlast subset-code(1))
have fst x  $\notin \text{set } (\text{interp-ins } ops)$ 
proof(cases xs = [])
  case True
    then obtain xi where x = (xi, Some ref)
      using insert-seq-hd snoc.prem2 by force
      moreover have ref  $\notin \text{set } (\text{interp-ins } as)$ 
        using interp-ins-monotonic snoc.prem1 snoc.prem5  $\langle ops = as @ x \# bs \rangle$ 
by blast
      ultimately have xi  $\notin \text{set } (\text{interp-ins } (as @ [x] @ bs))$ 
        using snoc.prem1 by (simp add: interp-ins-ref-nonex  $\langle ops = as @ x \# bs \rangle$ )
      then show fst x  $\notin \text{set } (\text{interp-ins } ops)$ 
        by (simp add:  $\langle ops = as @ x \# bs \rangle \langle x = (xi, Some ref) \rangle$ )
    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)  $\wedge$  x = (xi, xr)
        by fastforce
      moreover from this have xr = Some yi
        using insert-seq.cases snoc.prem2 calculation by fastforce
      moreover have yi  $\notin \text{set } (\text{interp-ins } ops)$ 
        using IH calculation
        by (metis Nil-is-map-conv fst-conv last-in-set last-map snoc-eq-iff-butlast)
      hence yi  $\notin \text{set } (\text{interp-ins } as)$ 
        using  $\langle ops = as @ x \# bs \rangle$  interp-ins-monotonic snoc.prem1 by blast
      ultimately have xi  $\notin \text{set } (\text{interp-ins } (as @ [x] @ bs))$ 
        using interp-ins-ref-nonex snoc.prem1  $\langle ops = as @ x \# bs \rangle$  by fastforce
      then show fst x  $\notin \text{set } (\text{interp-ins } ops)$ 
        by (simp add:  $\langle ops = as @ x \# bs \rangle \langle y = (yi, yr) \wedge x = (xi, xr) \rangle$ )
    qed
  then show  $\forall i \in \text{set } (\text{map fst } (xs @ [x])). i \notin \text{set } (\text{interp-ins } ops)$ 
    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  $\subseteq$  set ops and set ys  $\subseteq$  set ops
  and distinct (map fst xs @ map fst ys)

```

```

and  $\text{fst } (\text{hd } xs) < \text{fst } (\text{hd } ys)$ 
and  $\bigwedge r. \text{start} = \text{Some } r \implies r \in \text{set } (\text{interp-ins } ops)$ 
shows  $(\forall x \in \text{set } (\text{map } \text{fst } xs). \forall y \in \text{set } (\text{map } \text{fst } ys). \text{list-order } ops \ y \ x) \wedge$ 
 $(\forall r. \text{start} = \text{Some } r \longrightarrow (\forall x \in \text{set } (\text{map } \text{fst } xs). \text{list-order } ops \ r \ x) \wedge$ 
 $(\forall y \in \text{set } (\text{map } \text{fst } ys). \text{list-order } ops \ r \ y))$ 
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)  $a \in \text{set } xs \mid$  (a-in-ys)  $a \in \text{set } ys \mid$  (neither)  $a \notin \text{set } xs \wedge a \notin$ 
set ys
  by blast
then show ?case
proof(cases)
  case a-in-xs
    then have  $a \notin \text{set } ys$ 
      using snoc.prems(8) by auto
    hence  $\text{set } ys \subseteq \text{set } ops$ 
      using snoc.prems(7) DiffE by auto
    from a-in-xs have  $a = \text{last } xs$ 
      using insert-ops-subset-last snoc.prems by blast
    have IH:  $(\forall x \in \text{set } (\text{map } \text{fst } (\text{butlast } xs)). \forall y \in \text{set } (\text{map } \text{fst } ys). \text{list-order } ops$ 
 $y \ x) \wedge$ 
 $(\forall r. \text{start} = \text{Some } r \longrightarrow (\forall x \in \text{set } (\text{map } \text{fst } (\text{butlast } xs)). \text{list-order } ops$ 
 $r \ x) \wedge$ 
 $(\forall y \in \text{set } (\text{map } \text{fst } \quad \quad \quad ys). \text{list-order } ops \ r \ y))$ 
      proof(cases xs = [a])
        case True
          moreover have  $\forall r. \text{start} = \text{Some } r \longrightarrow (\forall y \in \text{set } (\text{map } \text{fst } ys). \text{list-order}$ 
 $ops \ r \ y)$ 
            using insert-seq-after-start  $\langle \text{insert-ops } ops \rangle \langle \text{set } ys \subseteq \text{set } ops \rangle$  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  $\langle a = \text{last } xs \rangle$  have  $\text{set } (\text{butlast } xs) \subseteq \text{set } ops$ 
              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  $\langle a = \text{last } xs \rangle$  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  $\text{set } ys \subseteq \text{set } ops$ 
            using  $\langle a \notin \text{set } ys \rangle \langle \text{set } ys \subseteq \text{set } ops \rangle$  by blast
          moreover have  $\text{hd } (\text{butlast } xs) = \text{hd } xs$ 

```

**by** (*metis append-butlast-last-id calculation(2) hd-append2 insert-seq-nonempty snoc.premis(2)*)  
**hence**  $\text{fst } (\text{hd } (\text{butlast } xs)) < \text{fst } (\text{hd } ys)$   
**by** (*simp add: snoc.premis(9)*)  
**moreover have**  $\bigwedge r. \text{start} = \text{Some } r \implies r \in \text{set } (\text{interp-ins ops})$   
**proof** –  
**fix**  $r$   
**assume**  $\text{start} = \text{Some } r$   
**then obtain**  $xid$  **where**  $\text{hd } xs = (xid, \text{Some } r)$   
**using** *insert-seq-hd snoc.premis(2)* **by** *auto*  
**hence**  $r < xid$   
**by** (*metis hd-in-set insert-ops-memb-ref-older insert-seq-nonempty snoc.premis(2) snoc.premis(3)*)  
**moreover have**  $xid < \text{fst } a$   
**proof** –  
**have**  $xs = (\text{butlast } xs) @ [a]$   
**using** *snoc.premis(2) insert-seq-nonempty*  $\langle a = \text{last } xs \rangle$  **by** *fastforce*  
**moreover have**  $(xid, \text{Some } r) \in \text{set } (\text{butlast } xs)$   
**using**  $\langle \text{hd } xs = (xid, \text{Some } r) \rangle$  *insert-seq-nonempty list.set-sel(1) snoc.premis(2)*  
**by** (*metis*  $\langle \text{hd } (\text{butlast } xs) = \text{hd } xs \rangle \langle \text{insert-seq start } (\text{butlast } xs) \rangle$ )  
**hence**  $xid \in \text{set } (\text{map } \text{fst } (\text{butlast } xs))$   
**by** (*metis in-set-zipE zip-map-fst-snd*)  
**ultimately show** *?thesis*  
**using** *snoc.premis(3) last-op-greatest* **by** (*metis prod.collapse*)  
**qed**  
**ultimately have**  $r \neq \text{fst } a$   
**using** *dual-order.asym* **by** *blast*  
**thus**  $r \in \text{set } (\text{interp-ins ops})$   
**using** *snoc.premis(1) snoc.premis(10) interp-ins-maybe-grow2*  $\langle \text{start} = \text{Some } r \rangle$  **by** *blast*  
**qed**  
**ultimately show** *?thesis*  
**using**  $\langle \text{insert-ops ops} \rangle$  *snoc.IH snoc.premis(4) snoc.premis(5)* **by** *blast*  
**qed**  
**moreover have**  $x\text{-exists}: \forall x \in \text{set } (\text{map } \text{fst } (\text{butlast } xs)). x \in \text{set } (\text{interp-ins ops})$   
**proof**(*cases start*)  
**case** *None*  
**moreover have**  $\text{set } (\text{butlast } xs) \subseteq \text{set } ops$   
**using**  $\langle a = \text{last } xs \rangle$  *distinct-map snoc.premis(6) snoc.premis(8) subset-butlast*  
**by** *fastforce*  
**ultimately show** *?thesis*  
**using** *insert-seq-start-none*  $\langle \text{insert-ops ops} \rangle$  *snoc.premis*  
**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**  $\forall y \in \text{set } (\text{map fst } ys). \text{list-order } (\text{ops } @ [a]) y (\text{fst } a)$   
**proof** (*cases*  $xs = [a]$ )  
   **case** *xs-a: True*  
   **have**  $ys \neq [] \implies \text{False}$   
   **proof** –  
     **assume**  $ys \neq []$   
     **then obtain** *yi* **where** *yi-def: ys = (yi, start) # (tl ys)*  
       **by** (*metis hd-Cons-tl insert-seq-hd snoc.premis(4)*)  
     **hence**  $(yi, \text{start}) \in \text{set ops}$   
       **by** (*metis <set ys ⊆ set ops> list.set-intros(1) subsetCE*)  
     **hence**  $yi \in \text{set } (\text{map fst ops})$   
       **by force**  
     **hence**  $yi < \text{fst } a$   
       **using** *snoc.premis(1) last-op-greatest* **by** (*metis prod.collapse*)  
     **moreover have**  $\text{fst } a < yi$   
       **by** (*metis yi-def xs-a fst-conv list.sel(1) snoc.premis(9)*)  
     **ultimately show** *False*  
       **using** *less-not-sym* **by** *blast*  
   **qed**  
   **then show**  $\forall y \in \text{set } (\text{map fst } ys). \text{list-order } (\text{ops } @ [a]) y (\text{fst } a)$   
     **using** *insert-seq-nonempty snoc.premis(4)* **by** *blast*  
**next**  
   **case** *xs-longer: False*  
   **hence** *butlast-split: butlast xs = (butlast (butlast xs)) @ [last (butlast xs)]*  
     **using**  $\langle a = \text{last } xs \rangle$  *insert-seq-butlast insert-seq-nonempty snoc.premis(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**  $xs = (\text{butlast } (\text{butlast } xs)) @ [\text{last } (\text{butlast } xs), a]$   
     **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.premis(2)*)  
   **hence**  $\text{snd } a = \text{Some } (\text{fst } (\text{last } (\text{butlast } xs)))$   
     **using** *snoc.premis(2) insert-seq-last-ref* **by** (*metis prod.collapse*)  
   **hence**  $\text{list-order } (\text{ops } @ [a]) (\text{fst } (\text{last } (\text{butlast } xs))) (\text{fst } a)$   
     **using** *list-order-insert-ref ref-exists snoc.premis(1)* **by** (*metis prod.collapse*)  
   **moreover have**  $\forall y \in \text{set } (\text{map fst } ys). \text{list-order } \text{ops } y (\text{fst } (\text{last } (\text{butlast } xs)))$   
   **by** (*metis IH butlast-split last-in-set last-map map-is-Nil-conv snoc-eq-iff-butlast*)  
   **hence**  $\forall y \in \text{set } (\text{map fst } ys). \text{list-order } (\text{ops } @ [a]) y (\text{fst } (\text{last } (\text{butlast } xs)))$   
     **using** *list-order-append snoc.premis(1)* **by** *blast*  
   **ultimately show**  $\forall y \in \text{set } (\text{map fst } ys). \text{list-order } (\text{ops } @ [a]) y (\text{fst } a)$   
     **using** *list-order-trans snoc.premis(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.premis(2)*)  
   **hence**  $\text{set } (\text{map fst } xs) = \text{set } (\text{map fst } (\text{butlast } xs)) \cup \{\text{fst } a\}$

by (simp add: ‹a = last xs›)  
 moreover have  $\forall r. \text{start} = \text{Some } r \longrightarrow \text{list-order } (\text{ops } @ [a]) r (\text{fst } a)$   
 using snoc.prem1 by (cases start, auto simp add: insert-seq-after-start ‹a = last xs› map-fst-xs)  
 ultimately show  $(\forall x \in \text{set } (\text{map } \text{fst } xs). \forall y \in \text{set } (\text{map } \text{fst } ys). \text{list-order } (\text{ops } @ [a]) y x) \wedge$   
 $(\forall r. \text{start} = \text{Some } r \longrightarrow (\forall x \in \text{set } (\text{map } \text{fst } xs). \text{list-order } (\text{ops } @ [a]) r x))$   
 $\wedge$   
 $(\forall y \in \text{set } (\text{map } \text{fst } ys). \text{list-order } (\text{ops } @ [a]) r y)$   
 using snoc.prem1 by (simp add: list-order-append)  
 next  
 case a-in-ys  
 then have  $a \notin \text{set } xs$   
 using snoc.prem8 by auto  
 hence  $\text{set } xs \subseteq \text{set } \text{ops}$   
 using snoc.prem6 DiffE by auto  
 from a-in-ys have  $a = \text{last } ys$   
 using insert-ops-subset-last snoc.prem by blast  
 have IH:  $(\forall x \in \text{set } (\text{map } \text{fst } xs). \forall y \in \text{set } (\text{map } \text{fst } (\text{butlast } ys)). \text{list-order } \text{ops } y x) \wedge$   
 $(\forall r. \text{start} = \text{Some } r \longrightarrow (\forall x \in \text{set } (\text{map } \text{fst } xs). \text{list-order } \text{ops } r x) \wedge$   
 $(\forall y \in \text{set } (\text{map } \text{fst } (\text{butlast } ys)). \text{list-order } \text{ops } r y))$   
 proof (cases ys = [a])  
 case True  
 moreover have  $\forall r. \text{start} = \text{Some } r \longrightarrow (\forall y \in \text{set } (\text{map } \text{fst } xs). \text{list-order } \text{ops } r y)$   
 using insert-seq-after-start ‹insert-ops ops› ‹set xs  $\subseteq$  set ops› snoc.prem  
 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  $\text{set } (\text{butlast } ys) \subseteq \text{set } \text{ops}$   
 using snoc.prem 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.prem(4)  
 by blast  
 moreover have insert-ops (butlast ys)  
 using snoc.prem(4) snoc.prem(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.prem(8) by blast  
 moreover have  $\text{set } xs \subseteq \text{set } \text{ops}$   
 using ‹a  $\notin$  set xs› ‹set xs  $\subseteq$  set ops› by blast  
 moreover have  $\text{hd } (\text{butlast } ys) = \text{hd } ys$   
 by (metis append-butlast-last-id calculation(2) hd-append2 insert-seq-nonempty snoc.prem(4))  
 hence  $\text{fst } (\text{hd } xs) < \text{fst } (\text{hd } (\text{butlast } ys))$



```

    by (simp add: snoc.premis(9))
  moreover have  $\bigwedge r. \text{start} = \text{Some } r \implies r \in \text{set } (\text{interp-ins ops})$ 
  proof -
    fix r
    assume start = Some r
    then obtain yid where hd ys = (yid, Some r)
      using insert-seq-hd snoc.premis(4) by auto
    hence r < yid
  by (metis hd-in-set insert-ops-memb-ref-older insert-seq-nonempty snoc.premis(4)
snoc.premis(5))
  moreover have yid < fst a
  proof -
    have ys = (butlast ys) @ [a]
      using snoc.premis(4) insert-seq-nonempty  $\langle a = \text{last } ys \rangle$  by fastforce
    moreover have (yid, Some r)  $\in \text{set } (\text{butlast } ys)$ 
      using  $\langle \text{hd } ys = (yid, \text{Some } r) \rangle$  insert-seq-nonempty list.sel(1)
snoc.premis(2)
    by (metis  $\langle \text{hd } (\text{butlast } ys) = \text{hd } ys \rangle$   $\langle \text{insert-seq start } (\text{butlast } ys) \rangle$ )
    hence yid  $\in \text{set } (\text{map fst } (\text{butlast } ys))$ 
    by (metis in-set-zipE zip-map-fst-snd)
    ultimately show ?thesis
      using snoc.premis(5) last-op-greatest by (metis prod.collapse)
  qed
  ultimately have r  $\neq$  fst a
    using dual-order.asym by blast
  thus r  $\in \text{set } (\text{interp-ins ops})$ 
    using snoc.premis(1) snoc.premis(10) interp-ins-maybe-grow2  $\langle \text{start} = \text{Some } r \rangle$  by blast
  qed
  ultimately show ?thesis
    using  $\langle \text{insert-ops ops} \rangle$  snoc.IH snoc.premis(2) snoc.premis(3) by blast
  qed
  moreover have  $\forall x \in \text{set } (\text{map fst } xs). \text{list-order } (\text{ops } @ [a]) (\text{fst } a) x$ 
  proof (cases ys = [a])
  case ys-a: True
  then show  $\forall x \in \text{set } (\text{map fst } xs). \text{list-order } (\text{ops } @ [a]) (\text{fst } a) x$ 
  proof (cases start)
  case None
  then show ?thesis
    using insert-seq-start-none list-order-insert-none snoc.premis
  by (metis  $\langle \text{insert-ops ops} \rangle$   $\langle \text{set } xs \subseteq \text{set } ops \rangle$  fst-conv insert-seq-hd list.sel(1)
ys-a)
  next
  case (Some r)
  moreover from this have  $\forall x \in \text{set } (\text{map fst } xs). \text{list-order } ops r x$ 
    using IH by blast
  ultimately show ?thesis
    using snoc.premis(1) snoc.premis(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**  $\langle a = \text{last } ys \rangle$  *insert-seq-butlast insert-seq-nonempty snoc.premis(4)* **by**  
*fastforce*  
**moreover from** *this* **have** *ys = (butlast (butlast ys)) @ [last (butlast ys), a]*  
**by** (*metis*  $\langle a = \text{last } ys \rangle$  *append.assoc append-butlast-last-id butlast.simps(2)*  
*insert-seq-nonempty last-ConsL last-ConsR list.simps(3) snoc.premis(4)*)  
**hence** *snd a = Some (fst (last (butlast ys)))*  
**using** *snoc.premis(4) insert-seq-last-ref* **by** (*metis prod.collapse*)  
**moreover have**  $\forall x \in \text{set } (\text{map } \text{fst } xs). \text{list-order ops } (\text{fst } (\text{last } (\text{butlast } ys))) x$   
**by** (*metis IH butlast-split last-in-set last-map map-is-Nil-conv snoc-eq-iff-butlast*)  
**ultimately show**  $\forall x \in \text{set } (\text{map } \text{fst } xs). \text{list-order } (\text{ops } @ [a]) (\text{fst } a) x$   
**using** *list-order-insert-between snoc.premis(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.premis(4)*)  
**hence** *set (map fst ys) = set (map fst (butlast ys))  $\cup$  {fst a}*  
**by** (*simp add:  $\langle a = \text{last } ys \rangle$* )  
**moreover have**  $\forall r. \text{start} = \text{Some } r \longrightarrow \text{list-order } (\text{ops } @ [a]) r (\text{fst } a)$   
**using** *snoc.premis* **by** (*cases start, auto simp add: insert-seq-after-start  $\langle a =$*   
*last ys  $\rangle$  map-fst-xs*)  
**ultimately show**  $(\forall x \in \text{set } (\text{map } \text{fst } xs). \forall y \in \text{set } (\text{map } \text{fst } ys). \text{list-order } (\text{ops}$   
 $@ [a]) y x) \wedge$   
 $(\forall r. \text{start} = \text{Some } r \longrightarrow (\forall x \in \text{set } (\text{map } \text{fst } xs). \text{list-order } (\text{ops } @ [a]) r x)$   
 $\wedge$   
 $(\forall y \in \text{set } (\text{map } \text{fst } ys). \text{list-order } (\text{ops } @ [a]) r y))$   
**using** *snoc.premis(1)* **by** (*simp add: list-order-append*)  
**next**  
**case** *neither*  
**hence** *set xs  $\subseteq$  set ops and set ys  $\subseteq$  set ops*  
**using** *snoc.premis(6) snoc.premis(7) DiffE* **by** *auto*  
**have**  $\forall r. \text{start} = \text{Some } r \longrightarrow r \in \text{set } (\text{interp-ins ops}) \vee (xs = [] \wedge ys = [])$   
**proof** (*cases xs*)  
**case** *Nil*  
**then show** *?thesis* **using** *insert-seq-nonempty snoc.premis(2)* **by** *blast*  
**next**  
**case** *xs-nonempty*: (*Cons x xsa*)  
**have**  $\bigwedge r. \text{start} = \text{Some } r \implies r \in \text{set } (\text{interp-ins ops})$   
**proof** –  
**fix** *r*  
**assume** *start = Some r*  
**then obtain** *xi* **where** *x = (xi, Some r)*  
**using** *insert-seq-hd xs-nonempty snoc.premis(2)* **by** *fastforce*  
**hence**  $(xi, \text{Some } r) \in \text{set ops}$   
**using**  $\langle \text{set } xs \subseteq \text{set ops} \rangle$  *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.prem(1) by (metis prod.collapse)
  ultimately have fst a ≠ r
    using order.asym by blast
  then show r ∈ set (interp-ins ops)
    using snoc.prem(1) snoc.prem(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.prem 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.prem(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  $\bigwedge r. \text{start} = \text{Some } r \implies r \in \text{set } (\text{interp-ins ops})$ 
  using assms(9) by blast
ultimately have  $\forall x \in \text{set } (\text{map fst ys}). \forall y \in \text{set } (\text{map fst xs}). \text{list-order ops } y \ x$ 
  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  $\subseteq$  set ops and set ys  $\subseteq$  set ops
  and start  $\notin$  set (interp-ins ops)
shows  $\forall x \in \text{set } (\text{map fst xs}) \cup \text{set } (\text{map fst ys}). x \notin \text{set } (\text{interp-ins ops})$ 
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* ::  $'oid::\{\text{linorder}\}$  list  $\Rightarrow 'oid \Rightarrow 'oid$  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* ::  $'oid::\{\text{linorder}\}$  list  $\Rightarrow ('oid \times 'oid \text{ option}) \Rightarrow 'oid$  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  
 $x \# \text{insert-body } xs \ e$   
else  
 $x \# \text{insert-rga } xs \ (e, \text{Some } i)$ )

**definition**  $\text{interp-rga} :: ('oid::\{\text{linorder}\} \times 'oid \text{ option}) \text{ list} \Rightarrow 'oid \text{ list}$  **where**  
 $\text{interp-rga } ops \equiv \text{foldl } \text{insert-rga} \ [] \ ops$

## 5.1 Commutativity of $\text{insert-rga}$

**lemma**  $\text{insert-body-set-ins}$  [*simp*]:  
**shows**  $\text{set } (\text{insert-body } xs \ e) = \text{insert } e \ (\text{set } xs)$   
**by** (*induction xs, auto*)

**lemma**  $\text{insert-rga-set-ins}$ :  
**assumes**  $i \in \text{set } xs$   
**shows**  $\text{set } (\text{insert-rga } xs \ (oid, \text{Some } i)) = \text{insert } oid \ (\text{set } xs)$   
**using** *assms* **by** (*induction xs, auto*)

**lemma**  $\text{insert-body-commutes}$ :  
**shows**  $\text{insert-body } (\text{insert-body } xs \ e1) \ e2 = \text{insert-body } (\text{insert-body } xs \ e2) \ e1$   
**by** (*induction xs, auto*)

**lemma**  $\text{insert-rga-insert-body-commute}$ :  
**assumes**  $i2 \neq \text{Some } e1$   
**shows**  $\text{insert-rga } (\text{insert-body } xs \ e1) \ (e2, i2) = \text{insert-body } (\text{insert-rga } xs \ (e2, i2)) \ e1$   
**using** *assms* **by** (*induction xs; cases i2*) (*auto simp add: insert-body-commutes*)

**lemma**  $\text{insert-rga-None-commutes}$ :  
**assumes**  $i2 \neq \text{Some } e1$   
**shows**  $\text{insert-rga } (\text{insert-rga } xs \ (e1, \text{None})) \ (e2, i2) =$   
 $\text{insert-rga } (\text{insert-rga } xs \ (e2, i2)) \ (e1, \text{None})$   
**using** *assms* **by** (*induction xs; cases i2*) (*auto simp add: insert-body-commutes*)

**lemma**  $\text{insert-rga-nonexistent}$ :  
**assumes**  $i \notin \text{set } xs$   
**shows**  $\text{insert-rga } xs \ (e, \text{Some } i) = xs$   
**using** *assms* **by** (*induction xs, auto*)

**lemma**  $\text{insert-rga-Some-commutes}$ :  
**assumes**  $i1 \in \text{set } xs$  **and**  $i2 \in \text{set } xs$   
**and**  $e1 \neq i2$  **and**  $e2 \neq i1$   
**shows**  $\text{insert-rga } (\text{insert-rga } xs \ (e1, \text{Some } i1)) \ (e2, \text{Some } i2) =$   
 $\text{insert-rga } (\text{insert-rga } xs \ (e2, \text{Some } i2)) \ (e1, \text{Some } i1)$   
**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  $i2 \neq \text{Some } e1$  and  $i1 \neq \text{Some } e2$ 
  shows  $\text{insert-rga } (\text{insert-rga } xs (e1, i1)) (e2, i2) =$ 
     $\text{insert-rga } (\text{insert-rga } xs (e2, i2)) (e1, i1)$ 
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  $r1 \in \text{set } xs \wedge r2 \in \text{set } xs$ )
      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  $\exists p s. xs = p @ s \wedge \text{insert-body } xs e = p @ e \# s$ 
proof (induction  $xs$ , force)
  case (Cons  $a xs$ )
  then obtain  $p s$  where  $IH: xs = p @ s \wedge \text{insert-body } xs e = p @ e \# s$ 
    by blast
  then show  $\exists p s. a \# xs = p @ s \wedge \text{insert-body } (a \# xs) e = p @ e \# s$ 
  proof (cases  $a < e$ )
    case True
    then have  $a \# xs = [] @ (a \# p @ s) \wedge \text{insert-body } (a \# xs) e = [] @ e \# (a$ 
 $\# p @ s)$ 
      by (simp add: IH)
    then show ?thesis by blast
  next
  case False

```

**then have**  $a \# xs = (a \# p) @ s \wedge \text{insert-body } (a \# xs) e = (a \# p) @ e \# s$   
**using** *IH* **by** *auto*  
**then show** *?thesis* **by** *blast*  
**qed**  
**qed**

**lemma** *insert-between-elements*:  
**assumes**  $xs = \text{pre} @ \text{ref} \# \text{suf}$   
**and** *distinct xs*  
**and**  $\bigwedge i. i \in \text{set } xs \implies i < e$   
**shows**  $\text{insert-rga } xs (e, \text{Some } \text{ref}) = \text{pre} @ \text{ref} \# e \# \text{suf}$   
**using** *assms* **proof**(*induction xs arbitrary: pre, force*)  
**case** (*Cons a xs*)  
**then show**  $\text{insert-rga } (a \# xs) (e, \text{Some } \text{ref}) = \text{pre} @ \text{ref} \# e \# \text{suf}$   
**proof**(*cases pre*)  
**case** *pre-nil: Nil*  
**then have**  $a = \text{ref}$   
**using** *Cons.premis(1)* **by** *auto*  
**then show** *?thesis*  
**using** *Cons.premis pre-nil* **by** (*cases suf, auto*)  
**next**  
**case** (*Cons b pre'*)  
**then have**  $\text{insert-rga } xs (e, \text{Some } \text{ref}) = \text{pre}' @ \text{ref} \# e \# \text{suf}$   
**using** *Cons.IH Cons.premis* **by** *auto*  
**then show** *?thesis*  
**using** *Cons.premis(1) Cons.premis(2) local.Cons* **by** *auto*  
**qed**  
**qed**

**lemma** *insert-rga-after-ref*:  
**assumes**  $\forall x \in \text{set } as. a \neq x$   
**and**  $\text{insert-body } (cs @ ds) e = cs @ e \# ds$   
**shows**  $\text{insert-rga } (as @ a \# cs @ ds) (e, \text{Some } a) = as @ a \# cs @ e \# ds$   
**using** *assms* **by** (*induction as; auto*)

**lemma** *insert-rga-preserves-order*:  
**assumes**  $i = \text{None} \vee (\exists i'. i = \text{Some } i' \wedge i' \in \text{set } xs)$   
**and** *distinct xs*  
**shows**  $\exists \text{pre } \text{suf}. xs = \text{pre} @ \text{suf} \wedge \text{insert-rga } xs (e, i) = \text{pre} @ e \# \text{suf}$   
**proof**(*cases i*)  
**case** *None*  
**then show**  $\exists \text{pre } \text{suf}. xs = \text{pre} @ \text{suf} \wedge \text{insert-rga } xs (e, i) = \text{pre} @ e \# \text{suf}$   
**using** *insert-body-split* **by** *auto*  
**next**  
**case** (*Some r*)  
**moreover from** *this* **obtain** *as bs* **where**  $xs = as @ r \# bs \wedge (\forall x \in \text{set } as. x \neq r)$   
**using** *assms(1) split-list-first* **by** *fastforce*  
**moreover have**  $\exists cs ds. bs = cs @ ds \wedge \text{insert-body } bs e = cs @ e \# ds$

**by** (*simp add: insert-body-split*)  
**then obtain**  $cs\ ds$  **where**  $bs = cs @ ds \wedge insert\text{-}body\ bs\ e = cs @ e \# ds$   
**by** *blast*  
**ultimately have**  $xs = (as @ r \# cs) @ ds \wedge insert\text{-}rga\ xs\ (e, i) = (as @ r \# cs) @ e \# ds$   
**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**  $\bigwedge r. r2 = Some\ r \implies r \neq i1$   
**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**  $ref \in fst\ 'set\ pre$   
**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**  $ref \in fst\ 'set\ pre$   
**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.prem)
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.prem)
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.prem x-pair by fastforce
  hence xr = None ∨ (∃ r. xr = Some r ∧ r ∈ set (map fst xs))

```

```

    using option.set-sel by blast
  hence  $xr = \text{None} \vee (\exists r. xr = \text{Some } r \wedge r \in \text{set } (\text{interp-rga } xs))$ 
    using interp-rga-ids rga-ops-rem-last snoc.prem1 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:  $x = (\text{oid}, \text{ref})$  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:  $\bigwedge i. i \in \text{set } (\text{interp-rga } xs) \implies i < \text{oid}$ 
proof -
  have  $\bigwedge i. i \in \text{set } (\text{map fst } ys) \implies i < \text{oid}$ 
    using assms(3) by (simp add: spec-ops-id-inc x-pair insert-ops-def)
  hence  $\bigwedge i. i \in \text{set } (\text{map fst } xs) \implies i < \text{oid}$ 
    using <set xs = set ys> by auto
  thus  $\bigwedge i. i \in \text{set } (\text{interp-rga } xs) \implies i < \text{oid}$ 
    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  $\exists as bs. \text{interp-rga } xs = as @ r \# bs$ 
proof -

```

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

**lemma**  $\text{interp-rga-reorder}$ :

**assumes**  $\text{rga-ops } (pre @ suf @ [(oid, ref)])$   
**and**  $\bigwedge i r. (i, \text{Some } r) \in \text{set } suf \implies r \neq oid$   
**and**  $\bigwedge r. ref = \text{Some } r \implies r \notin \text{fst } ' \text{set } suf$   
**shows**  $\text{interp-rga } (pre @ (oid, ref) \# suf) = \text{interp-rga } (pre @ suf @ [(oid, ref)])$   
**using**  $\text{assms}$  **proof**  $(\text{induction } suf \text{ rule: List.rev-induct})$   
**case Nil**  
**then show**  $?case$  **by simp**  
**next**  
**case**  $(\text{snoc } x \ xs)$   
**have**  $\text{ref-not-}x: \bigwedge r. ref = \text{Some } r \implies r \neq \text{fst } x$  **using**  $\text{snoc.prem}(3)$  **by auto**  
**have**  $IH: \text{interp-rga } (pre @ (oid, ref) \# xs) = \text{interp-rga } (pre @ xs @ [(oid, ref)])$   
**proof** –  
**have**  $\text{rga-ops } ((pre @ xs) @ [x] @ [(oid, ref)])$   
**using**  $\text{snoc.prem}(1)$  **by auto**  
**moreover have**  $\bigwedge r. ref = \text{Some } r \implies r \neq \text{fst } x$   
**by**  $(\text{simp add: ref-not-}x)$   
**ultimately have**  $\text{rga-ops } ((pre @ xs) @ [(oid, ref)])$   
**using**  $\text{rga-ops-rem-penultimate}$   
**by**  $(\text{metis } (no\text{-types, lifting}) \text{ Cons-eq-append-conv prod.collapse})$   
**thus**  $?thesis$  **using**  $\text{snoc}$  **by force**  
**qed**  
**obtain**  $xi$   $xr$  **where**  $x\text{-pair}: x = (xi, xr)$  **by force**  
**have**  $\text{interp-rga } (pre @ (oid, ref) \# xs @ [(xi, xr)]) =$   
 $\text{insert-rga } (\text{interp-rga } (pre @ xs @ [(oid, ref)])) (xi, xr)$   
**using**  $IH$   $\text{interp-rga-tail-unfold}$  **by**  $(\text{metis } \text{append.assoc } \text{append-Cons})$   
**moreover have**  $\dots = \text{insert-rga } (\text{insert-rga } (\text{interp-rga } (pre @ xs)) (oid, ref)) (xi,$   
 $xr)$   
**using**  $\text{interp-rga-tail-unfold}$  **by**  $(\text{metis } \text{append-assoc})$   
**moreover have**  $\dots = \text{insert-rga } (\text{insert-rga } (\text{interp-rga } (pre @ xs)) (xi, xr)) (oid,$

```

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

lemma rga-spec-equal:
assumes set xs = set ys
  and insert-ops xs
  and rga-ops ys
shows  $\text{interp-ins } xs = \text{interp-rga } ys$ 
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  $x \in \text{set } ys$ 
  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.prem(3) ys-split by blast
    thus crdt-ops (pre @ suf) set-option
      using crdt-ops-rem-spec snoc.prem(3) ys-split insert-ops-def by blast
  qed
hence rga-ops (pre @ suf)
  using rga-ops-def by blast
moreover have  $\text{set } xs = \text{set } (pre @ suf)$ 
  by (metis append-set-rem-last crdt-ops-distinct insert-ops-def rga-ops-def
    snoc.prem(3) 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.prem(3) 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.prem(1) ys-split by auto
  thus ?thesis
    using IH snoc.prem(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  $\bigwedge op2 r. op2 \in snd \text{ ' set suf } \implies r \in set-option op2 \implies r \neq oid$ 
    using snoc.prem
  by (simp add: crdt-ops-independent-suf insert-ops-def rga-ops-def x-pair ys-split)
  hence  $\bigwedge i r. (i, Some r) \in set suf \implies r \neq oid$ 
    by fastforce
  moreover have  $\bigwedge r. ref = Some r \implies r \notin fst \text{ ' set suf}$ 
    using crdt-ops-no-future-ref snoc.prem(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  $\exists ys. set xs = set ys \wedge insert-ops ys$ 
  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  $\exists ys. set ys = set xs \wedge insert-ops ys \wedge interp-ins ys = interp-rga xs$ 
  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.