OpSets: Sequential Specifications for Replicated Datatypes
Proof Document

Martin Kleppmann\textsuperscript{1}, Victor B. F. Gomes\textsuperscript{1}, Dominic P. Mulligan\textsuperscript{2}, and Alastair R. Beresford\textsuperscript{1}

\textsuperscript{1}Department of Computer Science and Technology, University of Cambridge, UK
\textsuperscript{2}Security Research Group, Arm Research, Cambridge, UK

Abstract

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

Contents

1 Abstract OpSet ...................................................... 2
1.1 OpSet definition .................................................. 2
1.2 Helper lemmas about lists .......................................... 3
1.3 The \textit{spec-ops} predicate ........................................ 5
1.4 The \textit{crdt-ops} predicate ........................................ 12

2 Specifying list insertion ............................................. 18
2.1 The \textit{insert-ops} predicate ....................................... 19
2.2 Properties of the \textit{insert-spec} function ......................... 20
2.3 Properties of the \textit{interp-ins} function ......................... 25
2.4 Equivalence of the two definitions of insertion .................. 26
2.5 The \textit{list-order} predicate ....................................... 32

3 Relationship to Strong List Specification .......................... 40
3.1 Lemmas about insertion and deletion ............................. 42
3.2 Lemmas about interpreting operations ............................. 48
3.3 Satisfying all conditions of $A_{\text{strong}}$ ....................... 52
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
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 ⊆ Y
**shows** opset X deps
**proof**
fix oid op1 op2
assume (oid, op1) ∈ X and (oid, op2) ∈ X
thus op1 = op2
using assms by (meson opset.unique-oid set-mp)
next
fix oid oper ref
assume (oid, oper) ∈ X and ref ∈ deps oper
thus ref < oid
using assms by (meson opset.ref-older set-rev-mp)
next
show finite X
using assms opset.finite-opset finite-subset by blast
qed

**lemma opset-insert:**

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

**lemma opset-sublist:**

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

1.2 Helper lemmas about lists

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

**lemma distinct-rem-mid:**

assumes unique-oid: (oid, op1) ∈ opset ⇒ (oid, op2) ∈ opset ⇒ op1 = op2
and ref-older: (oid, oper) ∈ opset ⇒ ref ∈ deps oper ⇒ ref < oid
and finite-opset: finite opset
assumes distinct \((xs @ [x] @ ys)\)
shows distinct \((xs @ ys)\)
using assms by (induction ys rule: rev-induct, simp-all)

lemma distinct-fst-append:
assumes \(x \in \text{set} (\text{map \text{fst} \ xs})\)
and distinct \((\text{map \text{fst} (xs @ ys)})\)
shows \(x \notin \text{set} (\text{map \text{fst} \ ys})\)
using assms by (induction ys rule: rev-induct, simp-all)

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

lemma distinct-set-remove-mid:
assumes distinct \((xs @ x @ ys)\)
shows \(\text{set (xs @ ys)} = \text{set (xs @ x @ ys)} - \{x\}\)
using assms by force

lemma distinct-list-split:
assumes distinct \(xs\)
and \(xs = xa @ x # ya\)
and \(xs = xb @ x # yb\)
shows \(xa = xb \land ya = yb\)
using assms proof (induction xs arbitrary: xa xb x)
fix \(xa xb x\)
assume \([] = xa @ x \neq ya\)
thus \(xa = xb \land ya = yb\)
by auto
next
fix a xs xa xb x
assume IH: \(\forall xa \ \text{distinct \ xs} \implies xs = xa @ x \neq ya \implies\)
\(xs = xb @ x \neq yb\)
\(\implies xa = xb \land ya = yb\)
and distinct (a \# xs) and a \# xs = xa @ x \# ya and a \# xs = xb @ x \# yb
thus \(xa = xb \land ya = yb\)
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

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 \textit{spec-ops}.

\textbf{definition} \textit{spec-ops} :: \(\langle \text{oid} :: \text{linorder} \rangle \times \text{'oper} \rangle \text{ list} \Rightarrow \langle \text{'oper} \Rightarrow \text{oid set} \rangle \Rightarrow \text{bool}\)

\textbf{where}

\[
\text{spec-ops ops deps} \equiv (\text{sorted (map fst ops)} \land \text{distinct (map fst ops)} \land (\forall \text{oid oper ref. (oid, oper)} \in \text{set ops} \land \text{ref} \in \text{deps oper} \rightarrow \text{ref} < \text{oid}))
\]

\textbf{lemma} \textit{spec-ops-empty}:

\textbf{shows} \text{spec-ops} [] deps

\textbf{by} (simp add: \textit{spec-ops-def})

\textbf{lemma} \textit{spec-ops-distinct}:

\textbf{assumes} \text{spec-ops ops deps}

\textbf{shows} \text{distinct ops}

\textbf{using} \textit{assms distinct-map spec-ops-def} \textbf{by} blast

\textbf{lemma} \textit{spec-ops-distinct-fst}:

\textbf{assumes} \text{spec-ops ops deps}

\textbf{shows} \text{distinct (map fst ops)}

\textbf{using} \textit{assms} \textbf{by} (simp add: \textit{spec-ops-def})

\textbf{lemma} \textit{spec-ops-sorted}:

\textbf{assumes} \text{spec-ops ops deps}

\textbf{shows} \text{sorted (map fst ops)}

\textbf{using} \textit{assms} \textbf{by} (simp add: \textit{spec-ops-def})

\textbf{lemma} \textit{spec-ops-rem-cons}:

\textbf{assumes} \text{spec-ops (xs @ [x]) deps}

\textbf{shows} \text{spec-ops xs deps}

\textbf{proof} –

\textbf{have} \text{sorted (map fst (xs @ [x])) and distinct (map fst (xs @ [x]))}

\textbf{using} \textit{assms spec-ops-def} \textbf{by} blast+

\textbf{moreover from this have} \text{sorted (map fst xs)}

\textbf{by} (simp add: \textit{sorted-Cons})

\textbf{moreover have} \(\forall \text{oid oper ref. (oid, oper)} \in \text{set xs} \land \text{ref} \in \text{deps oper} \rightarrow \text{ref} < \text{oid}\)

\textbf{by} (meson \textit{assms set-subset-Cons spec-ops-def subsetCE})

\textbf{ultimately show} \text{spec-ops xs deps}

\textbf{by} (simp add: \textit{spec-ops-def})

\textbf{qed}

\textbf{lemma} \textit{spec-ops-rem-last}:

\textbf{assumes} \text{spec-ops (xs @ [x]) deps}

\textbf{shows} \text{spec-ops xs deps}

\textbf{proof} –

\textbf{have} \text{sorted (map fst (xs @ [x])) and distinct (map fst (xs @ [x]))}

\textbf{using} \textit{assms spec-ops-def} \textbf{by} blast+

\textbf{moreover from this have} \text{sorted (map fst xs) and distinct xs}
by (auto simp add: sorted-append distinct-butlast distinct-map)
moreover have \( \forall \text{oid oper ref. (oid, oper)} \in \text{set} \, \text{xs} \land \text{ref} \in \text{deps} \, \text{oper} \rightarrow \text{ref} < \text{oid} \)
  by (metis assms butlast-snoc in-set-butlastD spec-ops-def)
ultimately show spec-ops \( \text{xs} \) \( \text{deps} \)
  by (simp add: spec-ops-def)
qed

lemma spec-ops-remove1:
  assumes spec-ops \( \text{xs} \) \( \text{deps} \)
  shows spec-ops \( \text{remove1} \, \text{x} \, \text{xs} \) \( \text{deps} \)
  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 \( \text{xs} \) \( \text{deps} \)
  and \( \text{oid} \) \( \text{oper} \) \( \in \) set \( \text{xs} \)
  and \( \text{r} \) \( \in \) deps oper
  shows \( \text{r} < \text{oid} \)
  using assms spec-ops-def
  by force

lemma spec-ops-ref-less-last:
  assumes spec-ops \( \text{xs} \) \( \omega \) \( \text{oper} \) \( \in \) set \( \text{xs} \)
  and \( \text{r} \) \( \in \) deps oper
  shows \( \text{r} < \text{oid} \)
  using assms spec-ops-ref-less
  by fastforce

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

lemma spec-ops-add-last:
  assumes spec-ops \( \text{xs} \) \( \text{deps} \)
  and \( \forall \, \text{i} \in \) set \( \text{map} \, \text{fst} \, \text{xs} \).
    \( \text{i} < \text{oid} \)
  and \( \forall \, \text{ref} \in \) deps oper.
    \( \text{ref} < \text{oid} \)
  shows spec-ops \( \text{xs} \) \( \omega \) \( \text{oper} \) \( \in \) set \( \text{xs} \)

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

lemma spec-ops-add-any:
  assumes spec-ops (xs @ ys) deps
  and ∀ i ∈ set (map fst xs). i < oid
  and ∀ i ∈ set (map fst ys). oid < i
  and ∀ ref ∈ deps oper. ref < oid
  shows spec-ops (xs @ [(oid, oper)] @ ys) deps
  using assms proof(induction ys rule: rev-induct)
  case Nil
  then show spec-ops (xs @ [(oid, oper)] @ []) deps
    by (simp add: spec-ops-add-last)
next
  case (snoc y ys)
  have IH: spec-ops (xs @ [(oid, oper)] @ ys) deps
    proof
      from snoc have spec-ops (xs @ ys) deps
        by (metis append-assoc spec-ops-rem-last)
      thus spec-ops (xs @ [(oid, oper)] @ ys) deps
        using assms(2) snoc by auto
    qed
  obtain yi yo where y-pair: y = (yi, yo)
    by force
  have oid-yi: oid < yi
    by (simp add: snoc.prems(3) y-pair)
  have yi-biggest: ∀ i ∈ set (map fst (xs @ [(oid, oper)] @ ys)). i < yi
    proof
      have ∀ i ∈ set (map fst xs). i < yi
        using oid-yi assms(2) less-trans by blast
      moreover have ∀ i ∈ set (map fst ys). i < yi
        by (metis UnCI append-assoc map-append set-append snoc.prems(1) spec-ops-id-inc y-pair)
      ultimately show ?thesis
        using oid-yi by auto
    qed
  have sorted (map fst (xs @ [(oid, oper)] @ ys @ [y]))
    proof
from IH have sorted (map fst (xs @ [(oid, oper)] @ ys))
    using spec-ops-def by blast
hence sorted (map fst (xs @ [(oid, oper)] @ ys) @ [yi])
    using yi-biggest sorted-append
by (metis (no-types, lifting) append-nil2 order-less-imp-le set-consD sorted-single)
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 \( \forall i \ opr r. \ (i, \ opr) \in \set (xs @ [(oid, oper)] @ ys @ [y]) \land r \in \deps opr \rightarrow r < i \)
proof −
    have \( \forall i \ opr r. \ (i, \ opr) \in \set (xs @ [(oid, oper)] @ ys) \land r \in \deps opr \rightarrow r < i \)
    by (meson IH spec-ops-def)
moreover have \( \forall ref. \ ref \in \deps yo \rightarrow ref < yi \)
    by (metis spec-ops-ref-less append-is-nil-conv last-appendR last-in-set last-snoc list.simps(3) snoc.prems(1) y-pair)
ultimately show \(?thesis\)
    using y-pair by auto
qed
ultimately show spec-ops (xs @ [(oid, oper)] @ ys @ [y]) deps
    using spec-ops-def by blast
qed

lemma spec-ops-split:
    assumes spec-ops xs deps
    and oid \notin set (map fst xs)
shows \( \exists pre suf. \ xs = pre @ suf \land \)
    \( \forall i \in \set (map fst pre). \ i < oid \) \land
    \( \forall i \in \set (map fst suf). \ oid < i \)
using assms proof(induction xs rule: rev-induct)
case Nil
then show \(?case\)
next
case (snoc x xs)
obtain xi xr where y-pair: \( x = (xi, xr) \)
    by force
obtain pre suf where IH: \( xs = pre @ suf \land \)
    \( \forall a \in \set (map fst pre). \ a < oid \) \land
    \( \forall a \in \set (map fst suf). \ oid < a \)
    by (metis UnCI map-append set-append snoc spec-ops-rem-last)
then show ?case
proof(cases xi < oid)
  case xi-less: True
  have \( \forall x \in \text{set} (\text{map fst (pre @ suf)}). \, x < xi \)
  using IH spec-ops-id-inc snoc.prems(1) y-pair by metis
  hence \( \forall x \in \text{set} \, . \, \text{fst} \, x < xi \)
  by simp
  hence \( \forall x \in \text{set} \, . \, \text{fst} \, x < oid \)
  using x-less by auto
  hence suf = []
  using IH last-in-set by fastforce
  hence xs @ [x] = (pre @ ([(xi, xr)])) @ [] ∧
  \( \forall i \in \text{set} (\text{map fst (pre @ ([(xi, xr)]))). a < oid) \) ∧
  \( \forall i \in \text{set} (\text{map fst []). oid < a) \)
  by (simp add: IH x-less y-pair)
  then show ?thesis by force
next
  case False
  hence oid < xi using snoc.prems(2) y-pair by auto
  hence xs @ [x] = (pre @ (suf @ ([(xi, xr)]))) ∧
  \( \forall i \in \text{set} (\text{map fst pre). i < oid) \) ∧
  \( \forall i \in \text{set} (\text{map fst (suf @ ([(xi, xr)]))). oid < i) \)
  by (simp add: IH y-pair)
  then show ?thesis by blast
qed
qed

lemma spec-ops-exists-base:
  assumes finite ops
  and \( \forall \text{oid} \, \text{op1} \, \text{op2}. (\text{oid}, \text{op1}) \in \text{ops} \implies (\text{oid}, \text{op2}) \in \text{ops} \implies \text{op1} = \text{op2} \)
  and \( \forall \text{oid} \, \text{oper} \, \text{ref}. (\text{oid}, \text{oper}) \in \text{ops} \implies \text{ref} \in \text{deps oper} \implies \text{ref} < \text{oid} \)
  shows \( \exists \text{op-list. set op-list = ops} \land \text{spec-ops op-list deps} \)
  using assms proof(induct ops rule: Finite-Set.finite-induct-select)
  case empty
  then show \( \exists \text{op-list. set op-list = {}} \land \text{spec-ops op-list deps} \)
  by (simp add: spec-ops-empty)
next
  case (select subset)
  from this obtain op-list where set op-list = subset and spec-ops op-list deps
  using assms by blast
  moreover obtain oid oper where select: (oid, oper) \in \text{ops} - \text{subset}
  using select.hyps(1) by auto
  moreover from this have \( \forall \text{op2. (oid, op2)} \in \text{ops} \implies \text{op2} = \text{oper} \)
  using assms(2) by auto
  hence oid \notin \text{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 $\forall i \in \text{set} (\text{map} \, \text{fst} \, \text{pre}). \ i < \text{oid}$
and $\forall i \in \text{set} (\text{map} \, \text{fst} \, \text{suf}). \ \text{oid} < i$
using spec-ops-split calculation by (metis (no-types, lifting) set-map)
moreover have $\text{set} (\text{pre} \, @ \ [(\text{oid}, \text{oper})] @ \text{suf}) = \text{insert} (\text{oid}, \text{oper}) \, \text{subset}$
using calculation by auto
moreover have $\text{spec-ops} (\text{pre} \, @ \ [(\text{oid}, \text{oper})] @ \text{suf}) \, \text{deps}$
using calculation spec-ops-add-any assms(3) by (metis DiffD1)
ultimately show $\, ?\text{case}$ by blast
qed

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

**lemma** spec-ops-exists:
  assumes opset ops deps
  shows $\exists \, \text{op-list}. \ \text{set} \, \text{op-list} = \text{ops} \land \text{spec-ops} \, \text{op-list} \, \text{deps}$
proof  
  have finite ops
  using assms opset.finite-opset by force
moreover have $\bigwedge \text{oid} \, \text{op1} \, \text{op2}. \ (\text{oid}, \text{op1}) \in \text{ops} \Rightarrow (\text{oid}, \text{op2}) \in \text{ops} \Rightarrow \text{op1} = \text{op2}$
  using assms opset.unique-oid by force
moreover have $\bigwedge \text{oid} \, \text{oper} \, \text{ref}. \ (\text{oid}, \text{oper}) \in \text{ops} \Rightarrow \text{ref} \in \text{deps} \, \text{oper} \Rightarrow \text{ref} < \text{oid}$
  using assms opset.ref-older by force
ultimately show $\exists \, \text{op-list}. \ \text{set} \, \text{op-list} = \text{ops} \land \text{spec-ops} \, \text{op-list} \, \text{deps}$
  by (simp add: spec-ops-exists-base)
qed

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

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

*Moreover have* \( \forall oid \ oper \ ref. \ (oid, oper) \in \text{set op-list} \implies ref \in \text{deps oper} \implies ref < oid \)

*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 |

\[ \text{crdt-ops } xs \text{ deps; } \]

oid \notin \text{set (map fst } xs); \]

\( \forall \text{ref} \in \text{deps oper. ref} \in \text{set (map fst } xs) \land \text{ref} < \text{oid} \]

\[ \implies \text{crdt-ops } (xs \ominus [(\text{oid, oper})]) \text{ deps} \]

**QED**
inductive-cases \text{crdt-ops-last}: \text{crdt-ops} (xs @ [x]) \text{deps}

\textbf{lemma crdt-ops-intro:}
\begin{itemize}
\item \textbf{assumes} \( \forall r. r \in \text{deps} \text{ oper} \implies r \in \text{fst ' set xs} \land r < \text{oid} \)
\item \textbf{and} \( \text{oid} \notin \text{fst ' set xs} \)
\item \textbf{and} \( \text{crdt-ops} \text{ xs} \text{ deps} \)
\item \textbf{shows} \( \text{crdt-ops} (xs @ [(\text{oid, oper})]) \text{ deps} \)
\item \textbf{using} \( \text{assms} \text{ crdt-ops}.\text{simp} \text{ by force} \)
\end{itemize}

\textbf{lemma crdt-ops-rem-last:}
\begin{itemize}
\item \textbf{assumes} \( \text{crdt-ops} (xs @ [x]) \text{ deps} \)
\item \textbf{shows} \( \text{crdt-ops} \text{ xs} \text{ deps} \)
\item \textbf{using} \( \text{assms} \text{ crdt-ops}.\text{cases snoc-eq-iff-butlast by blast} \)
\end{itemize}

\textbf{lemma crdt-ops-ref-less:}
\begin{itemize}
\item \textbf{assumes} \( \text{crdt-ops} \text{ xs} \text{ deps} \)
\item \textbf{and} \( (\text{oid, oper}) \in \text{set xs} \)
\item \textbf{and} \( r \in \text{deps} \text{ oper} \)
\item \textbf{shows} \( r < \text{oid} \)
\item \textbf{using} \( \text{assms by (induction rule: crdt-ops.induct, auto)} \)
\end{itemize}

\textbf{lemma crdt-ops-ref-less-last:}
\begin{itemize}
\item \textbf{assumes} \( \text{crdt-ops} (xs @ [(\text{oid, oper})]) \text{ deps} \)
\item \textbf{and} \( r \in \text{deps} \text{ oper} \)
\item \textbf{shows} \( r < \text{oid} \)
\item \textbf{using} \( \text{assms crdt-ops-ref-less by fastforce} \)
\end{itemize}

\textbf{lemma crdt-ops-distinct-fst:}
\begin{itemize}
\item \textbf{assumes} \( \text{crdt-ops} \text{ xs} \text{ deps} \)
\item \textbf{shows} \( \text{distinct} \left( \text{map} \text{ fst} \text{ xs} \right) \)
\item \textbf{using} \( \text{assms proof (induction xs rule: List.rev-induct, simp)} \)
\item \textbf{case} \( \text{snoc x xs} \)
\item \textbf{hence} \( \text{distinct} \left( \text{map} \text{ fst} \text{ xs} \right) \)
\item \textbf{using} \( \text{crdt-ops-last by blast} \)
\item \textbf{moreover have} \( \text{fst} \text{ x} \notin \text{set} \left( \text{map} \text{ fst} \text{ xs} \right) \)
\item \textbf{using} \( \text{snoc by (metis crdt-ops-last fstI image-set)} \)
\item \textbf{ultimately show} \( \text{distinct} \left( \text{map} \text{ fst} \left( \text{xs} @ [x] \right) \right) \)
\item \textbf{by simp} \)
\end{itemize}
\end{document}
lemma crdt-ops-unique-mid:
assumes crdt-ops \((xs @ [\langle oid, oper \rangle]) @ ys)\) deps
shows oid \notin set (map fst xs) \land oid \notin set (map fst ys)
using assms proof (induction ys rule: rev-induct)
case Nil
then show oid \notin set (map fst xs) \land oid \notin 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: \(y = \langle yi, yr \rangle\)
  by fastforce
have IH: oid \notin set (map fst xs) \land oid \notin set (map fst ys)
  using crdt-ops-rem-last snoc by (metis append-assoc)
hence \(yi \notin set (map fst (xs @ [\langle oid, oper \rangle]) # ys)\)
  using crdt-ops-unique-last by (metis append-Cons append-self-conv2 snoc.prems y-pair)
thus oid \notin set (map fst xs) \land oid \notin set (map fst (ys @ [\(y\)])
  using IH y-pair by auto
qed

lemma crdt-ops-ref-exists:
assumes crdt-ops \((pre @ \langle oid, oper \rangle) # suf\)\) deps
  and ref \in deps oper
shows ref \in \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 \((xs @ [\langle oid, oper \rangle]) @ ys\)\) deps
shows \(\forall \text{ ref. } \text{ ref } \in \text{ deps oper } \Rightarrow \text{ ref } \notin \text{ fst } \set \text{ ys}\)
proof
  from assms(1) have \(\forall \text{ ref. } \text{ ref } \in \text{ deps oper } \Rightarrow \text{ ref } \in \text{ set } (map \text{ fst } xs)\)
    by (simp add: crdt-ops-ref-exists)
  moreover have distinct \((map \text{ fst } (xs @ [\langle oid, oper \rangle]) @ ys)\)
    using assms crdt-ops-distinct-fst by blast
  ultimately have \(\forall \text{ ref. } \text{ ref } \in \text{ deps oper } \Rightarrow \text{ ref } \notin \text{ set } (map \text{ fst } (\langle [\langle oid, oper \rangle] @ ys)\))
    using distinct-fst-append by metis
  thus \(\forall \text{ ref. } \text{ ref } \in \text{ deps oper } \Rightarrow \text{ ref } \notin \text{ fst } \set \text{ ys}\)
    by simp
qed
lemma crdt-ops-reorder:
  assumes crdt-ops (xs @ [(oid, oper)] @ ys) deps
  and ∃op2 r. op2 ∈ snd ' set ys ⇒ r ∈ deps op2 ⇒ r ≠ oid
  shows crdt-ops (xs @ ys @ [(oid, oper)]) deps
  using assms proof (induction ys rule: rev-induct)
  case Nil
  then show crdt-ops (xs @ [] @ [(oid, oper)]) deps
  using crdt-ops-rem-last by auto
  next
  case (snoc y ys)
  then obtain yi yo where y-pair: y = (yi, yo)
  by fastforce
  have IH: crdt-ops (xs @ ys @ [(oid, oper)]) deps
  proof
    have crdt-ops (xs @ [(oid, oper)] @ ys) deps
    by (metis snoc append assoc crdt-ops-rem-last)
    thus crdt-ops (xs @ ys @ [(oid, oper)]) deps
    using snoc.IH snoc.prems(2) by auto
  qed
  have crdt-ops (xs @ ys @ [y]) deps
  proof
    have y ∈ fst ' set (xs @ [(oid, oper)] @ ys)
    by (metis y-pair append assoc crdt-ops-unique-last set-map snoc.prems(1))
    hence y ∈ fst ' set (xs @ ys)
    by auto
    moreover have ∃r. r ∈ deps yo ⇒ r ∈ fst ' set (xs @ ys) ∧ r < yi
    proof
      have ∃r. r ∈ deps yo ⇒ r ≠ oid
      using snoc.prems(2) y-pair by fastforce
      moreover have ∃r. r ∈ deps yo ⇒ r ∈ fst ' set (xs @ [(oid, oper)] @ ys)
      by (metis y-pair append assoc snoc.prems(1) crdt-ops-ref-exists)
      moreover have ∃r. r ∈ deps yo ⇒ r < yi
      using crdt-ops-ref-less snoc.prems(1) y-pair by fastforce
      ultimately show ∃r. r ∈ deps yo ⇒ r ∈ fst ' set (xs @ ys) ∧ r < yi
      by simp
    qed
    moreover from IH have crdt-ops (xs @ ys) deps
    using crdt-ops-rem-last by force
    ultimately show crdt-ops (xs @ ys @ [y]) deps
    using y-pair crdt-ops-intro by (metis append assoc)
  qed
  moreover have oid ∉ fst ' set (xs @ ys @ [y])
  using crdt-ops-unique-mid by (metis no-types lifting UnE image-Un
  image-set append snoc.prems(1))
  moreover have ∃r. r ∈ deps oper ⇒ r ∈ fst ' set (xs @ ys @ [y])
  using crdt-ops-ref-exists
  by (metis UnCI append Cons image-Un set-append snoc.prems(1))
  moreover have ∃r. r ∈ deps oper ⇒ r < oid

15
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 \( \forall op2. op2 \in \text{snd } \set ys \implies r \in \text{deps op2} \implies r \neq \text{oid} \)
  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 \( \forall op2. op2 \in \text{snd } \set zs \implies r \in \text{deps op2} \implies r \neq \text{oid} \)
  proof
    have \( \forall op2. op2 \in \text{snd } \set xs \implies r \in \text{deps op2} \implies r < \text{oid} \)
    proof
      from assms(1) have \( \forall i. i \in \text{fst } \set xs \implies i < \text{oid} \)
        using spec-ops-id-inc by fastforce
      moreover have \( \forall i2 op2. (i2, op2) \in \set xs \implies r \in \text{deps op2} \implies r < i2 \)
        using assms(1) spec-ops-ref-less spec-ops-rem-last by fastforce
      ultimately show \( \forall op2. op2 \in \text{snd } \set xs \implies r \in \text{deps op2} \implies r < \text{oid} \)
        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 \( set zs \subseteq \set zs \)
        using append-subset(2) by simp
    qed
    ultimately show \( \forall op2. op2 \in \text{snd } \set zs \implies r \in \text{deps op2} \implies r \neq \text{oid} \)
      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 \( \forall op2. op2 \in \text{snd } \set zs \implies r \in \text{deps op2} \implies r \neq \text{oid} \)
      using assms crdt-ops-independent-suf by fastforce
    thus crdt-ops \( (ys @ zs @ [x]) \) deps
  qed

16
lemma crdt-ops-rem-spec:

assumes spec-ops (xs @ [x]) deps

and crdt-ops (ys @ [x] @ zs) deps

and set (xs @ [x]) = set (ys @ [x] @ zs)

shows crdt-ops (ys @ zs) deps

using assms crdt-ops-rem-last crdt-ops-reorder-spec append-assoc by metis

lemma crdt-ops-rem-penultimate:

assumes crdt-ops (xs @ [(i1, r1)] @ [(i2, r2)]) deps

and \( r. r \in \text{deps} r2 \implies r \neq i1 \)

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 \text{set} (\text{map fst} xs)

by auto

moreover have crdt-ops ((xs @ [(i1, r1)]) @ [(i2, r2)]) deps

using assms(1) by auto

hence \( \forall r. r \in \text{deps} r2 \implies r \in \text{fst} \cdot \text{set} (xs @ [(i1, r1)]) \)

using crdt-ops-ref-exists by metis

hence \( \forall r. r \in \text{deps} r2 \implies r \in \text{set} (\text{map fst} xs) \)

using assms(2) by auto

moreover have \( \forall r. r \in \text{deps} r2 \implies r < i2 \)

using assms(1) crdt-ops-ref-less by fastforce

ultimately show crdt-ops (xs @ [(i2, r2)]) deps

by (simp add: crdt-ops-intro)

qed

lemma crdt-ops-spec-ops-exist:

assumes crdt-ops xs deps

shows \( \exists ys. \text{set} xs = \text{set} ys \land \text{spec-ops} ys \text{ deps} \)

using assms proof(induction xs rule: List.rev-induct)

case Nil

then show \( \exists ys. \text{set} [] = \text{set} ys \land \text{spec-ops} ys \text{ deps} \)

by (simp add: spec-ops-empty)

next
case (snoc x xs)

hence IH: \( \exists ys. \text{set} xs = \text{set} ys \land \text{spec-ops} ys \text{ deps} \)

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 \( \exists \text{pre suf}, \ \text{ys} = \text{pre} \circ \text{suf} \land \)
\[
(\forall i \in \text{set (map fst pre)}. i < \text{oid}) \land \\
(\forall i \in \text{set (map fst suf)}. \text{oid} < i)
\]

proof
- 
  have \( \text{oid} \notin \text{set (map fst xs)} \)
  using calculation(3) crdt-ops-unique-last snoc.prems by force
  hence \( \text{oid} \notin \text{set (map fst ys)} \)
  by (simp add: calculation(3))
  thus \( ?\text{thesis} \)
  using spec-ops-split (spec-ops ys deps) by blast

qed

from this obtain \( \text{pre suf} \) where \( \text{ys} = \text{pre} @ \text{suf} \land \)
\[
(\forall i \in \text{set (map fst pre)}. i < \text{oid}) \land \\
(\forall i \in \text{set (map fst suf)}. \text{oid} < i)
\]

moreover have \( \text{set (xs @ [(\text{oid}, \text{ref})])} = \text{set (pre @ [(\text{oid}, \text{ref})] @ suf)} \)
  using crdt-ops-distinct calculation snoc.prems by simp

moreover have \( \text{spec-ops (pre @ [(\text{oid}, \text{ref})] @ suf)} \) deps
  proof
    - 
      have \( \forall r \in \text{deps ref}. r < \text{oid} \)
      using calculation(3) crdt-ops-ref-less-last snoc.prems by fastforce
      hence \( \text{spec-ops (pre @ [(\text{oid}, \text{ref})] @ suf)} \) deps
        using spec-ops-add-any calculation by metis
      thus \( ?\text{thesis} \) by simp

  qed

ultimately show \( \exists \text{ys}. \text{set (xs @ [x])} = \text{set ys} \land \text{spec-ops ys deps} \)
  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 \((\text{ID}, \text{ref})\), where \text{ref} is either \text{None} (signifying an insertion at the head of the list) or \text{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: \text{insert-spec} and \text{insert-alt}. The \text{insert-alt} definition matches the paper, while \text{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 \text{insert-spec} :: \'oid list \Rightarrow (\'oid \times \'oid option) \Rightarrow \'oid list where
\[
\begin{align*}
\text{insert-spec } xs & \ (oid, \text{None}) = oid \# xs \\
\text{insert-spec } [] & \ (oid, -) = [] \\
\text{insert-spec } (x \# xs) & \ (oid, \text{Some ref}) = \\
& \quad (\text{if } x = \text{ref} \text{ then } x \# oid \# xs \\
& \quad \quad \text{else } x \# (\text{insert-spec } xs \ (oid, \text{Some ref}))
\end{align*}
\]

fun insert-alt \:: ('oid × 'oid option) set ⇒ ('oid × 'oid) ⇒ ('oid × 'oid option) set

where

\[
\text{insert-alt } \text{list-rel} \ (oid, \text{ref}) = \\
\quad (\text{if } \exists n. \ (\text{ref}, n) \in \text{list-rel} \\
\quad \quad \text{then } \{(p, n) \in \text{list-rel}. \ p \neq \text{ref}\} \cup \{(\text{ref}, \text{Some oid})\} \cup \\
\quad \quad \{(i, n). \ i = \text{oid} \land (\text{ref}, n) \in \text{list-rel} \}
\quad \text{else } \text{list-rel})
\]

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

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

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

lemma list-greater-non-memb:
  fixes oid :: 'oid::{linorder}
  assumes \( \forall x.\ x \in \text{set}\ \text{xs} \Rightarrow x < \text{oid} \)
  and oid ∈ set xs
  shows False
  using assms by blast

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

21
next
  case (Some ref)
  then show \( a = e \) using Cons by (case-tac \( x = \text{ref} \), auto)
qed
qed

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

22
have \( \land x. x \in \text{set } xs \implies x < oid \)

using L by auto

hence distinct (insert-spec xs (oid, Some id))

using S IH[OF 1] by blast

hence \( a \notin \text{set } \) (insert-spec xs (oid, Some id)) \( \land \) distinct (insert-spec xs (oid, Some id))

using 0 by auto

} from this S T show distinct (insert-spec \( a \# xs \) (oid, ref))

by clarsimp

qed

qed

lemma insert-after-ref:

assumes distinct (xs @ ref \# ys)

shows insert-spec (xs @ ref \# ys) (oid, Some ref) = xs @ ref \# oid \# ys

using assms by (induction xs, auto)

lemma insert-somewhere:

assumes \( \text{ref} = \text{None} \lor (\text{ref} = \text{Some } r \land r \in \text{set list}) \)

shows \( \exists xs \ ys. \text{list} = xs \@ ys \land \text{insert-spec list (oid, ref)} = xs \@ oid \# ys \)

using assms proof(induction list)

assume \( \text{ref} = \text{None} \lor \text{ref} = \text{Some } r \land r \in \text{set } [] \)

thus \( \exists xs \ ys. \text{[]} = xs \@ ys \land \text{insert-spec } [] (\text{oid, ref}) = xs \@ oid \# ys \)

proof

assume \( \text{ref} = \text{None} \)

thus \( \exists xs \ ys. \text{[]} = xs \@ ys \land \text{insert-spec } [] (\text{oid, ref}) = xs \@ oid \# ys \)

by auto

next

assume \( \text{ref} = \text{Some } r \land r \in \text{set } [] \)

thus \( \exists xs \ ys. \text{[]} = xs \@ ys \land \text{insert-spec } [] (\text{oid, ref}) = xs \@ oid \# ys \)

by auto

qed

next

fix a list

assume 1: \( \text{ref} = \text{None} \lor \text{ref} = \text{Some } r \land r \in \text{set } (a\#\text{list}) \)

and IH: \( \text{ref} = \text{None} \lor \text{ref} = \text{Some } r \land r \in \text{set list} \implies \exists xs \ ys. \text{list} = xs \@ ys \land \text{insert-spec list (oid, ref)} = xs \@ oid \# ys \)

show \( \exists xs \ ys. a \# \text{list} = xs \@ ys \land \text{insert-spec } (a \# \text{list}) (\text{oid, ref}) = xs \@ oid \# ys \)

proof(rule disjE[OF 1])

assume \( \text{ref} = \text{None} \)

thus \( \exists xs \ ys. a \# \text{list} = xs \@ ys \land \text{insert-spec } (a \# \text{list}) (\text{oid, ref}) = xs \@ oid \# ys \)

by force

next

assume \( \text{ref} = \text{Some } r \land r \in \text{set } (a \# \text{list}) \)

hence 2: \( r = a \lor r \in \text{set list} \) and 3: \( \text{ref} = \text{Some } r \)

by auto
show \( \exists x \, y \, s. \ a \ # \ list = x @ y \ # \ insert-spec (a \ # \ list) (oid, ref) = x @ y \ # \ ys \)

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

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

24
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

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)
  thus insert-spec ((a ≠ xs) @ ys) (oid, ref) = (a ≠ xs) @ insert-spec ys (oid, ref)
    by auto
qed

2.3 Properties of the interp-ins function

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

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

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

hence IH: set (interp-ins xs) ⊆ set (map fst xs)
  using interp-ins-def by blast

obtain oid ref where x-pair: x = (oid, ref)
  by fastforce

hence spec: interp-ins (xs @ [x]) = insert-spec (interp-ins xs) (oid, ref)
  by (simp add: interp-ins-def)

then show set (interp-ins (xs @ [x])) ⊆ set (map fst (xs @ [x]))
proof (cases ref)
  case None
  then show set (interp-ins (xs @ [x])) ⊆ set (map fst (xs @ [x]))
    using IH spec x-pair by auto

next
  case (Some a)
  then show set (interp-ins (xs @ [x])) ⊆ set (map fst (xs @ [x]))
    using IH spec x-pair by (cases a ∈ set (interp-ins xs), auto)

qed

qed

lemma interp-ins-distinct:
  assumes insert-ops op-list
  shows distinct (interp-ins op-list)
  using assms proof (induction op-list rule: rev-induct)
  case Nil
  then show distinct (interp-ins []) by (simp add: interp-ins-def)

next
  case (snoc x xs)
  hence IH: distinct (interp-ins xs) by blast
  obtain oid ref where x-pair: x = (oid, ref) by force
  hence ∀x ∈ set (map fst xs). x < oid
    using last-op-greatest snoc.prems by blast
  hence ∀x ∈ set (interp-ins xs). x < oid
    using interp-ins-subset by fastforce
  hence distinct (insert-spec (interp-ins xs) (oid, ref))
    using IH insert-spec-distinct insert-spec-none by metis
  then show distinct (interp-ins (xs @ [x]))
    by (simp add: x-pair interp-ins-tail-unfold)

qed

2.4 Equivalence of the two definitions of insertion

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

We first define how to derive the successor relation from an Isabelle list. This relation contains (id, None) if id is the last element of the list, and (id1, id2) if id1 is immediately followed by id2 in the list.
fun succ-rel :: 'oid list ⇒ ('oid × 'oid option) set where
  succ-rel [] = {}
  succ-rel [head] = {(head, None)}
  succ-rel (head#x#xs) = {(head, Some x)} ∪ succ-rel (x#xs)

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

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

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

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

27
using \( \exists \).IH by auto

qed

qed

\textbf{lemma \textit{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: \(x \notin \text{fst} \cdot \text{succ-rel} xs\)
    using assms by (simp add: succ-rel-set-fst)
moreover obtain \(y\) where \((x, y) \in \text{succ-rel} (x \# xs)\)
  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 \(\{n. (x, n) \in \text{succ-rel} (x \# xs) \Rightarrow n = y\}\)
  by (metis Pair-inject fst-cone 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

\textbf{lemma \textit{succ-rel-swap-head}}:
assumes \((\text{ref} \# \text{list})\) and \((\text{ref}, n) \in \text{ succ-rel} (\text{ref} \# \text{list})\)
shows \(\text{succ-rel} (\text{oid} \# \text{list}) = \{(\text{oid}, n)\} \cup \text{succ-rel} \text{list}\)
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

\textbf{lemma \textit{succ-rel-insert-alt}}:
assumes \(a \neq \text{ref}\)
and distinct \((\text{oid} \# a \# b \# \text{list})\)
shows \(\text{insert-alt} (\text{succ-rel} (a \# b \# \text{list})) (\text{oid}, \text{ref}) =\)
  \(\{(a, \text{Some} b)\} \cup \text{insert-alt} (\text{succ-rel} (b \# \text{list})) (\text{oid}, \text{ref})\)
proof(cases \(\exists n. (\text{ref}, n) \in \text{succ-rel} (a \# b \# \text{list})\))
case True
  hence \(\text{insert-alt} (\text{succ-rel} (a \# b \# \text{list})) (\text{oid}, \text{ref}) =\)
    \(\{(p, n) \in \text{succ-rel} (a \# b \# \text{list}). p \neq \text{ref}\} \cup \{(\text{ref}, \text{Some} \text{oid})\} \cup\)
    \(\{(i, n). i = \text{oid} \land (\text{ref}, n) \in \text{succ-rel} (a \# b \# \text{list})\}\) 
  by simp

28
moreover have \((p, n) \in \text{succ-rel} \ (a \ # b \ # \ list). \ p \neq \text{ref}\) =
\\{ \{a, \text{Some } b\} \cup \{(p, n) \in \text{succ-rel} \ (b \ # \ list). \ p \neq \text{ref}\} \}

using \text{assms}(1) \text{ by auto}

moreover have \(\text{insert-alt} \ (\text{succ-rel} \ (b \ # \ list)) \ (\text{oid}, \text{ref})\) =
\\{ \{(p, n) \in \text{succ-rel} \ (b \ # \ list). \ p \neq \text{ref}\} \cup \{(\text{ref}, \text{Some } \text{oid})\} \cup
\{(i, n). \ i = \text{oid} \land (\text{ref}, n) \in \text{succ-rel} \ (b \ # \ list)\} \}

proof -
\begin{align*}
& \text{have } \exists n. \ (\text{ref}, n) \in \text{succ-rel} \ (b \ # \ list) \\
& \quad \text{using } \text{assms}(1) \text{ True by auto} \\
& \quad \text{thus } \text{thesis by simp} \\
& \quad \text{qed}
\end{align*}

moreover have \((i, n). \ i = \text{oid} \land (\text{ref}, n) \in \text{succ-rel} \ (a \ # b \ # \ list)\) =
\\{ \{(i, n). \ i = \text{oid} \land (\text{ref}, n) \in \text{succ-rel} \ (b \ # \ list)\} \}

using \text{assms}(1) \text{ by auto}

ultimately show \(\text{thesis by simp}\)

next
\begin{align*}
& \text{case } \text{False} \\
& \quad \text{then show } \text{thesis by auto} \\
& \quad \text{qed}
\end{align*}

\begin{align*}
\text{lemma } \text{succ-rel-insert-head}: \\
& \text{assumes } \text{distinct} \ (\text{ref} \ # \ list) \\
& \text{shows } \text{succ-rel} \ (\text{insert-spec} \ (\text{ref} \ # \ list) \ (\text{oid}, \text{Some } \text{ref}))) = \\
& \quad \text{insert-alt} \ (\text{succ-rel} \ (\text{ref} \ # \ list)) \ (\text{oid}, \text{ref}) \\
& \text{proof -} \\
& \quad \text{obtain } n \text{ where } \text{ref-in-rel} : (\text{ref}, n) \in \text{succ-rel} \ (\text{ref} \ # \ list) \\
& \quad \quad \text{by } (\text{cases list, auto}) \\
& \quad \quad \text{moreover from } \text{this have } \{(p, n) \in \text{succ-rel} \ (\text{ref} \ # \ list). \ p \neq \text{ref}\} = \text{succ-rel list} \\
& \quad \quad \text{using } \text{assms } \text{succ-rel-rem-head by } (\text{metis } (\text{mono-tags, lifting})) \\
& \quad \quad \text{moreover have } \{(i, n). \ i = \text{oid} \land (\text{ref}, n) \in \text{succ-rel} \ (\text{ref} \ # \ list)\} = \{(\text{oid}, n)\} \\
& \quad \quad \text{proof -} \\
& \quad \quad \quad \text{have } \land nx. \ (\text{ref}, nx) \in \text{succ-rel} \ (\text{ref} \ # \ list) \Rightarrow nx = n \\
& \quad \quad \quad \text{using } \text{assms } \text{by } (\text{simp add: succ-rel-functional ref-in-rel}) \\
& \quad \quad \quad \text{hence } \{(i, n) \in \text{succ-rel} \ (\text{ref} \ # \ list). \ i = \text{ref}\} \subseteq \{(\text{ref}, n)\} \\
& \quad \quad \quad \text{by } \text{blast} \\
& \quad \quad \quad \text{moreover have } \{(\text{ref}, n)\} \subseteq \{(i, n) \in \text{succ-rel} \ (\text{ref} \ # \ list)\}. \ i = \text{ref} \\
& \quad \quad \quad \text{by } (\text{simp add: ref-in-rel}) \\
& \quad \quad \quad \text{ultimately show } \text{thesis by } \text{blast} \\
& \quad \quad \quad \text{qed}
\end{align*}

moreover have \(\text{insert-alt} \ (\text{succ-rel} \ (\text{ref} \ # \ list)) \ (\text{oid}, \text{ref})\) =
\\{ \{(p, n) \in \text{succ-rel} \ (\text{ref} \ # \ list). \ p \neq \text{ref}\} \cup \{(\text{ref}, \text{Some } \text{oid})\} \cup \{(i, n). \ i = \text{oid} \land (\text{ref}, n) \in \text{succ-rel} \ (\text{ref} \ # \ list)\} \}

proof -
\begin{align*}
& \text{have } \exists n. \ (\text{ref}, n) \in \text{succ-rel} \ (\text{ref} \ # \ list) \\
& \quad \text{using } \text{ref-in-rel} \text{ by } \text{blast} \\
& \quad \text{thus } \text{thesis by simp} \\
& \quad \text{qed}
\end{align*}

ultimately have \(\text{insert-alt} \ (\text{succ-rel} \ (\text{ref} \ # \ list)) \ (\text{oid}, \text{ref})\) =

\ldots
succ-rel list \cup \{(ref, Some oid)\} \cup \{(oid, n)\}

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

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

lemma succ-rel-insert-Some:
assumes distinct list
shows succ-rel (insert-spec list (oid, Some ref)) = insert-alt (succ-rel list) (oid, ref)
using assms proof(induction list)
case Nil
then show succ-rel (insert-spec [] (oid, Some ref)) = insert-alt (succ-rel []) (oid, ref)
by simp
next
case (Cons a list)
hence distinct (a \# list)
by simp
then show \( \text{succ-rel } (\text{insert-spec } (a \# \text{list}) (\text{oid}, \text{Some ref})) = \text{insert-alt } (\text{succ-rel } (a \# \text{list}) ) (\text{oid}, \text{ref}) \)
\[
\text{proof}\left(\text{cases } a = \text{ref}\right) \\
\text{case } \text{True} \\
\text{then show } \text{thesis} \\
\text{using } \text{succ-rel-insert-head}\langle \text{distinct } (a \# \text{list}) \rangle \text{ by } \text{metis} \\
\text{next} \\
\text{case } \text{False} \\
\text{hence } a \neq \text{ref} \text{ by } \text{simp} \\
\text{moreover have } \text{succ-rel } (\text{insert-spec list } (\text{oid}, \text{Some ref})) = \\
\text{insert-alt } (\text{succ-rel } (a \# \text{list} )) (\text{oid}, \text{ref}) \\
\text{using } \text{Cons.IH Cons.prems } \text{by } \text{auto} \\
\text{ultimately show } \text{succ-rel } (\text{insert-spec } (a \# \text{list}) ) (\text{oid}, \text{ref}) = \\
\text{insert-alt } (\text{succ-rel } (a \# \text{list} )) (\text{oid}, \text{ref}) \\
\text{by } \text{(cases \text{list}, \text{force}, \text{metis Cons.prems succ-rel-insert-later})} \\
\text{qed} \\
\text{qed}
\]

The main result of this section, that \text{insert-spec} and \text{insert-alt} are equivalent.

\textbf{theorem} \text{insert-alt-equivalent}: \\
\text{assumes } \text{insert-ops \text{ops}} \\
\text{and } \text{head } \notin \text{fst } \text{\text{set } \text{ops}} \\
\text{and } \text{\( \forall \text{r. Some r } \in \text{snd } \text{\text{set } \text{ops}} \Rightarrow r \neq \text{head} \)} \\
\text{shows } \text{succ-rel } (\text{head } \# \text{ interp-ins \text{ops}}) = \text{interp-alt head \text{ops}} \\
\text{using } \text{assms } \text{proof} (\text{induction \text{ops } rule: List.rev-induct}) \\
\text{case } \text{Nil} \\
\text{then show } \text{succ-rel } (\text{head } \# \text{ interp-ins } []) = \text{interp-alt head []} \\
\text{by } \text{(simp add: interp-ins-def interp-alt-def)} \\
\text{next} \\
\text{case } (\text{snoc x xs}) \\
\text{have } \text{IH}: \text{succ-rel } (\text{head } \# \text{ interp-ins xs}) = \text{interp-alt head xs} \\
\text{using } \text{snoc } \text{by } \text{auto} \\
\text{have } \text{distinct-list: distinct } (\text{head } \# \text{ interp-ins xs}) \\
\text{proof } - \\
\text{have } \text{distinct } (\text{interp-ins xs}) \\
\text{using } \text{interp-ins-distinct snoc.prems(1) by blast} \\
\text{moreover have } \text{set } (\text{interp-ins xs}) \subseteq \text{fst } \text{\text{set } xs} \\
\text{using } \text{interp-ins-subset snoc.prems(1) by fastforce} \\
\text{ultimately show } \text{distinct } (\text{head } \# \text{ interp-ins xs}) \\
\text{using } \text{snoc.prems(2) by auto} \\
\text{qed} \\
\text{obtain } \text{oid r where } \text{x-pair: } x = (\text{oid}, r) \text{ by } \text{force} \\
\text{then show } \text{succ-rel } (\text{head } \# \text{ interp-ins } (\text{xs } @ [x])) = \text{interp-alt head } (\text{xs } @ [x]) \\
\text{proof}\left(\text{cases } r\right) \\
\text{case } \text{None} \\
\text{have } \text{interp-alt head } (\text{xs } @ [x]) = \text{insert-alt } (\text{interp-alt head xs} ) (\text{oid}, \text{head}) \\
\text{by } \text{(simp add: interp-alt-def None x-pair)} \\
\text{moreover have } \text{... } = \text{insert-alt } (\text{succ-rel } (\text{head } \# \text{ interp-ins xs}) ) (\text{oid}, \text{head}) \\
\text{by } \text{(simp add: IH)} \\
\text{qed} \\
\text{qed}
\]

31
moreover have ... = succ-rel (insert-spec (head # interp-ins xs) (oid, Some head))
using distinct-list succ-rel-insert-Some by metis
moreover have ... = succ-rel (head # (insert-spec (interp-ins xs) (oid, None)))
by auto
moreover have ... = succ-rel (head # (interp-ins (xs @ [x])))
by (simp add: interp-ins-tail-unfold None x-pair)
ultimately show ?thesis by simp
next
case (Some ref)
have ref ≠ head
by (simp add: Some snoc.prems(3) x-pair)
have interp-alt head (xs @ [x]) = insert-alt (interp-alt head xs) (oid, ref)
by (simp add: interp-alt-def Some x-pair)
moreover have ... = insert-alt (succ-rel (head # interp-ins xs)) (oid, ref)
by (simp add: IH)
mindlest have ... = succ-rel (insert-spec (head # interp-ins xs) (oid, Some ref))
using distinct-list succ-rel-insert-Some by metis
moreover have ... = succ-rel (head # (insert-spec (interp-ins xs) (oid, Some ref)))
using ref ≠ head by auto
moreover have ... = succ-rel (head # (interp-ins (xs @ [x])))
by (simp add: interp-ins-tail-unfold Some x-pair)
ultimately show ?thesis by simp
qed
qed

2.5 The list-order predicate

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

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

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

lemma list-orderE:
assumes list-order ops x y
shows ∃ xs ys zs. interp-ins ops = xs @ [x] @ ys @ [y] @ zs
using assms by (auto simp add: list-order-def)
lemma list-order-memb1:
assumes list-order ops x y
shows \( x \in \text{set} (\text{interp-ins ops}) \)
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: \( \text{interp-ins op-list} = (xxs@x@xys)@(y#xzs) \)
using assms by (auto simp add: list-order-def interp-ins-def)
obtain yxs yys yzs where 2: \( \text{interp-ins op-list} = yxs@y#(yys@z@yzs) \)
using assms by (auto simp add: list-order-def interp-ins-def)
have 3: distinct (\text{interp-ins op-list})
using assms interp-ins-distinct by blast
hence xzs = yys@z@yzs using distinct-list-split [OF 3, OF 2, OF 1]
by auto
hence \( \text{interp-ins op-list} = xxs@x@xys@y@yys@z@yzs \)
using 1 2 3 by clarsimp
thus list-order op-list x z
using assms by (metis append.assoc list-orderI)
qed

lemma insert-preserves-order:
assumes insert-ops ops and insert-ops rest and rest = before @ after and ops = before @ (oid, ref) @ after
shows \( \exists \text{xs ys zs. interp-ins rest} = \text{xs @ zs} \land \text{interp-ins ops} = \text{xs @ ys @ zs} \)
using assms proof (induction after arbitrary: rest ops rule: List.rev-induct)
case Nil
then have 1: \( \text{interp-ins ops} = \text{insert-spec (interp-ins before) (oid, ref)} \)
by (simp add: interp-ins-tail-unfold)
then show \( \exists \text{xs ys zs. interp-ins rest} = \text{xs @ zs} \land \text{interp-ins ops} = \text{xs @ ys @ zs} \)
proof (cases ref)
case None
hence interp-ins rest = [] @ (interp-ins before) \land
    interp-ins ops = [] @ [oid] @ (interp-ins before)
using 1 Nil.prems(3) by simp
then show \?thesis by blast
next
case (Some a)
then show \(?thesis\)
proof(cases a ∈ set (interp-ins before))
  case True
  then obtain xs ys where interp-ins before = xs @ ys ∧
                   insert-spec (interp-ins before) (oid, ref) = xs @ oid # ys
     using insert-somewhere Some by metis
  hence interp-ins rest = xs @ ys ∧ interp-ins ops = xs @ [oid] @ ys
     using 1 Nil.prems(3) by auto
  then show \(?thesis\) by blast
next
  case False
  hence interp-ins ops = (interp-ins rest) @ [] @ []
     using insert-spec-nonex 1 Nil.prems(3) Some by simp
  then show \(?thesis\) by blast
qed
qed

next
  case (snoc oper op-list)
  then have insert-ops ((before @ (oid, ref) # op-list) @ [ oper])
      and insert-ops ((before @ op-list) @ [ oper])
     by auto
  then have ops1: insert-ops (before @ op-list)
      and ops2: insert-ops (before @ (oid, ref) # op-list)
     using insert-ops-appendD by blast+
  then obtain xs ys zs where IH1: interp-ins (before @ op-list) = xs @ zs
      and IH2: interp-ins (before @ (oid, ref) # op-list) = xs @ ys @ zs
     using snoc.IH by blast
  obtain i2 r2 where oper = (i2, r2) by force
  then show \(∃\) xs ys zs. interp-ins rest = xs @ zs ∧ interp-ins ops = xs @ ys @ zs
proof(cases r2)
  case None
  hence interp-ins (before @ op-list @ [ oper]) = (i2 # xs) @ zs
     by (metis IH1 oper = (i2, r2): append.assoc append-Cons insert-spec.simps(1)
         interp-ins-tail-unfold)
  moreover have interp-ins (before @ (oid, ref) # op-list @ [ oper]) = (i2 # xs)
      @ ys @ zs
     by (metis IH2 None oper = (i2, r2): append.assoc append-Cons insert-spec.simps(1)
         interp-ins-tail-unfold)
  ultimately show \(?thesis\)
     using snoc.prems(3) snoc.prems(4) by blast
next
  case (Some r)
  then have 1: interp-ins (before @ (oid, ref) # op-list @ [(i2, r2)]) =
           insert-spec (xs @ ys @ zs) (i2, Some r)
     by (metis IH2 append.assoc append-Cons interp-ins-tail-unfold)
  have 2: interp-ins (before @ op-list @ [(i2, r2)]) = insert-spec (xs @ zs) (i2,
     Some r)
     by (metis IH1 append.assoc interp-ins-tail-unfold Some)
  consider (r-xs) r ∈ set xs | (r-ys) r ∈ set ys | (r-zA r ∈ set zs |
(r-nonex) \( r \notin \text{set } (xs \cdot ys \cdot zs) \)
by auto
then show \( \exists xs \ ys \ zs. \text{interp-ins rest } = xs \cdot zs \land \text{interp-ins ops } = xs \cdot ys \cdot zs \)

proof (cases)
case r-xs
from this have insert-spec \((xs \cdot ys \cdot zs) \ (i2, \text{Some } r) = (\text{insert-spec } xs \ (i2, \text{Some } r)) \cdot ys \cdot zs \)
by (meson insert-first-part)
moreover have insert-spec \((xs \cdot zs) \ (i2, \text{Some } r) = (\text{insert-spec } xs \ (i2, \text{Some } r)) \cdot zs \)
by (meson r-xs insert-first-part)
ultimately show ?thesis
using 1 2 \( \langle \text{oper } = (i2, \text{r2}) \rangle \) \( \text{snoc.prems by auto} \)
next
case r-ys
hence \( r \notin \text{set } xs \land r \notin \text{set } zs \)
using IH2 \( \text{ops2 interp-ins-distinct by force+} \)
moreover from this have insert-spec \((xs \cdot ys \cdot zs) \ (i2, \text{Some } r) = (xs \cdot (\text{insert-spec } ys \ (i2, \text{Some } r)) \cdot zs \)
using insert-first-part insert-second-part insert-spec-nonex
by (metis Some UnE \( r-ys \) \( \text{set-append} \))
moreover have insert-spec \((xs \cdot zs) \ (i2, \text{Some } r) = xs \cdot zs \)
by (simp add: Some \( \text{calculation(1) calculation(2)} \))
ultimately show ?thesis
using 1 2 \( \langle \text{oper } = (i2, \text{r2}) \rangle \) \( \text{snoc.prems by auto} \)
next
case r-zs
hence \( r \notin \text{set } xs \land r \notin \text{set } ys \)
using IH2 \( \text{ops2 interp-ins-distinct by force+} \)
moreover from this have insert-spec \((xs \cdot ys \cdot zs) \ (i2, \text{Some } r) = (xs \cdot ys \cdot (\text{insert-spec } zs \ (i2, \text{Some } r)) \)
by (metis Some \( \text{UnE insert-second-part insert-spec-nonex set-append} \))
moreover have insert-spec \((xs \cdot zs) \ (i2, \text{Some } r) = xs \cdot (\text{insert-spec } zs \ (i2, \text{Some } r)) \)
by (simp add: \( r-zs \) \( \text{calculation(1)} \) \( \text{insert-second-part} \))
ultimately show ?thesis
using 1 2 \( \langle \text{oper } = (i2, \text{r2}) \rangle \) \( \text{snoc.prems by auto} \)
next
case r-nonex
then have insert-spec \((xs \cdot ys \cdot zs) \ (i2, \text{Some } r) = xs \cdot ys \cdot zs \)
by simp
moreover have insert-spec \((xs \cdot zs) \ (i2, \text{Some } r) = xs \cdot zs \)
using r-nonex by simp
ultimately show ?thesis
using 1 2 \( \langle \text{oper } = (i2, \text{r2}) \rangle \) \( \text{snoc.prems by auto} \)
qed
qed
qed
lemma distinct-fst:
assumes distinct (map fst A)
shows distinct A
using assms by (induction A) auto

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

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

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

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

lemma list-order-monotonic:
assumes insert-ops A and insert-ops B
  and set A ⊆ set B
  and list-order A x y
shows list-order B x y
using assms proof (induction rule: measure-induct-rule
  where f = λx. (length x − length A))
case (less xa)
have distinct (map fst A) and distinct (map fst xa) and
    sorted (map fst A) and sorted (map fst xa)
using less.prems by (auto simp add: insert-ops-def spec-ops-def)
hence distinct A and distinct xa
by (auto simp add: distinct-fst)
then show list-order xa x y
proof (cases length xa - length A)
case 0
hence set A = set xa
    using set-subset-length-eq less.prems ⟨distinct A⟩ ⟨distinct xa⟩ diff-is-0-eq
by blast
hence A = xa
    using ⟨distinct (map fst A)⟩ ⟨distinct (map fst xa)⟩
    ⟨sorted (map fst A)⟩ ⟨sorted (map fst xa)⟩ map-sorted-distinct-set-unique
by (metis distinct-map less.prems ⟨distinct (map fst A)⟩ ⟨distinct (map fst xa)⟩
    subset-Un-eq)
then show list-order xa x y
    using less.prems (4) by blast
next
case (Suc nat)
then obtain e where e ∈ set xa and e /∈ set A
    using length-diff-Suc-exists ⟨distinct A⟩ ⟨distinct xa⟩ less.prems (3)
    by blast
proof -
  have length (remove1 e xa) - length A < Suc nat
    using diff-Suc-1 diff-commute length-remove1 less-Suc-eq Suc ⟨e ∈ set xa⟩
    by (metis)
moreover have insert-ops (remove1 e xa)
  by (simp add: insert-ops-remove1 less.prems (2))
moreover have set A ⊆ set (remove1 e xa)
  by (metis (no-types, lifting) ⟨e /∈ set A⟩ in-set-remove1 less.prems (3)
    set-rev-mp subsetI)
ultimately show ?thesis
  by (simp add: Suc less.IH less.prems (1) less.prems (4))
qed
then obtain xs ys zs where interp-ins (remove1 e xa) = xs @ x # ys @ y # zs
    using list-order-def by fastforce
moreover obtain oid ref where e-pair: e = (oid, ref)
  by fastforce
moreover obtain ps ss where xa-split: xa = ps @ [e] @ ss and e /∈ set ps
    using split-list-first (e ∈ set xa) by fastforce
hence remove1 e (ps @ e # ss) = ps @ ss
  by (simp add: remove1-append)
moreover from this have insert-ops (ps @ ss) and insert-ops (ps @ e # ss)
    using xa-split less.prems (2) by (metis append-Cons append-Nil insert-ops-remove1,
    auto)
then obtain PSA YSA ZSA where interp-ins (ps @ ss) = PSA @ ZSA
    and interp-xa: interp-ins (ps @ (oid, ref) # ss) = PSA @ 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 zsa \leq length xs)
case True
then obtain ts where \(xsa@ts = xs\)
using app-length-lt-exists zsa-zsa by blast
hence interp-ins za = (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 xs \leq 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 za = xs @ [x] @
\((\text{drop } (Suc \text{ (length } xs)) xsa) @ ysa @ us) @ [y] @ zs\)
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 \((xs @ x \# ys @ [y])\) \leq 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 za = 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
3 Relationship to Strong List Specification

In this section we show that our list specification is stronger than the $A_{\text{strong}}$ 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 $A_{\text{strong}}$ specification.

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

An abstract execution $A = (H, \text{vis})$ belongs to the strong list specification $A_{\text{strong}}$ 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 $e = \text{do}(\text{op}, w) \in H$ returns a sequence of elements $w = a_0 \ldots a_{n-1}$, 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:
   \[
   \forall a. a \in w \iff (\text{do}(\text{ins}(a), \_), \_) \leq_{\text{vis}} e \land \neg(\text{do}(\text{del}(a), \_), \_) \leq_{\text{vis}} e).
   \]
   (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 $\text{op} = \text{ins}(a, k)$, then $a = a_{\text{min}\{k, n-1\}}$.

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 $A_{\text{strong}}$ using a simplified interpretation function that defines only insertion and deletion on a single list.

decision theory List-Spec
  imports Insert-Spec
begin

We first define a datatype for list operations, with two constructors: $\text{Insert \ ref \ val}$, and $\text{Delete \ ref}$. For insertion, the $\text{ref}$ 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)) 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 ≡ \exists 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}

41
locale list-opset = opset opset list-op-deps

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

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

3.1 Lemmas about insertion and deletion

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

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

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

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

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

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

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

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

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

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

lemma insertions-subset:
assumes list-ops A and list-ops B and set A ⊆ set B
shows set (insertions A) ⊆ set (insertions B)
using assms proof (induction B arbitrary: A rule: List.rev-induct)
case Nil
then show set (insertions A) ⊆ set (insertions [])
  by (simp add: insertions-def map-filter-simps)
next
case (snoc a ops)
obtain oid oper where a-pair: a = (oid, oper)
  by fastforce
have list-ops ops
  using list-ops-def spec-ops-rem-last snoc.prems(2) by blast
then show set (insertions A) ⊆ set (insertions (ops @ [a]))
proof (cases a ∈ set A)
case True
then obtain as bs where A-split: A = as @ a # bs ∧ a /∈ set as
  by (meson split-list-first)
hence remove1 a A = as @ bs
  by (simp add: remove1-append)
hence as-bs: insertions (remove1 a A) = insertions as @ insertions bs
43
by (simp add: insertions-def map-filter-append)
moreover have \( A = \text{as} \# [a] \# \text{bs} \)
by (simp add: A-split)
hence \( \text{as-a-bs: insertions } A = \text{insertions } \text{as} \# \text{insertions } [a] \# \text{insertions } \text{bs} \)
by (metis insertions-def map-filter-append)
moreover have \( \text{IH}: \text{set } (\text{insertions } (\text{remove1 } a \ A)) \subseteq \text{set } (\text{insertions } \text{ops}) \)
proof –
have \( \text{list-ops } (\text{remove1 } a \ A) \)
  using \( \text{snoc} \).prems(1) list-ops-def spec-ops-remove1 by blast
moreover have \( \text{set } (\text{remove1 } a \ A) \subseteq \text{set } \text{ops} \)
proof –
have \( \text{distinct } A \)
  using \( \text{snoc} \).prems(1) list-ops-def spec-ops-distinct by blast
hence \( a \notin \text{set } (\text{remove1 } a \ A) \)
by auto
moreover have \( \text{set } ([\text{ops} \# [a]]) = \text{set } \text{ops} \cup \{a\} \)
by auto
moreover have \( \text{set } (\text{remove1 } a \ A) \subseteq \text{set } A \)
by (simp add: set-remove1-subset)
ultimately show \( \text{set } (\text{remove1 } a \ A) \subseteq \text{set } \text{ops} \)
  using \( \text{snoc}. \text{prems}(3) \) by blast
qed
ultimately show \( ?\text{thesis} \)
  by (simp add: \( \langle \text{list-ops } \text{ops} \rangle \) \text{snoc}. \text{IH})
qed
ultimately show \( ?\text{thesis} \)
proof(cases oper)
case (Insert \text{ref} \text{val})
hence \( \text{insertions } [a] = [(\text{oid}, \text{ref})] \)
  by (simp add: insertions-def map-filter-\text{Some} \text{a-pair})
hence \( \text{set } (\text{insertions } A) = \text{set } (\text{insertions } (\text{remove1 } a \ A)) \cup \{(\text{oid}, \text{ref})\} \)
  using \( \text{as-a-bs} \) \( \text{as-bs} \) by auto
moreover have \( \text{set } (\text{insertions } ([\text{ops} \# [a]])) = \text{set } (\text{insertions } \text{ops}) \cup \{(\text{oid}, \text{ref})\} \)
  by (simp add: \( \langle \text{insertions } \text{ops} \rangle \text{a-pair} \text{insertions}-\text{last-\text{ins}} \))
ultimately show \( ?\text{thesis} \)
  using \( \text{IH} \) by auto
next
case (Delete \text{ref})
hence \( \text{insertions } [a] = [] \)
  by (simp add: insertions-def map-filter-\text{None} \text{a-pair})
hence \( \text{set } (\text{insertions } A) = \text{set } (\text{insertions } (\text{remove1 } a \ A)) \)
  using \( \text{as-a-bs} \) \( \text{as-bs} \) by auto
moreover have \( \text{set } (\text{insertions } ([\text{ops} \# [a]])) = \text{set } (\text{insertions } \text{ops}) \)
  by (simp add: Delete \text{a-pair} \text{insertions}-\text{last-del})
ultimately show \( ?\text{thesis} \)
  using \( \text{IH} \) by auto
qed
next
case False
hence set A ⊆ set ops
using DiffE snoc.prems by auto
hence set (insertions A) ⊆ set (insertions ops)
using snoc.IH snoc.prems(1) (list-ops ops) by blast
moreover have set (insertions ops) ⊆ set (insertions (ops @ [a]))
  by (simp add: insertions-def map-filter-append)
ultimately show ?thesis
  by blast
qed

lemma list-ops-insertions:
  assumes list-ops ops
  shows insert-ops (insertions ops)
  using assms proof(induction ops rule: List.rev-induct)
  case Nil
  then show insert-ops (insertions [])
    by (simp add: insert-ops-def spec-ops-def insertions-def map-filter-def)
next
  case (snoc a ops)
  hence IH: insert-ops (insertions ops)
  using list-ops-def spec-ops-rem-last by blast
  obtain oid oper where a-pair: a = (oid, oper)
    by fastforce
  then show insert-ops (insertions (ops @ [a]))
    proof(cases oper)
      case (Insert ref val)
      hence insertions (ops @ [a]) = insertions ops @ [(oid, ref)]
        by (simp add: a-pair insertions-last-ins)
      moreover have ∃i. i ∈ set (map fst ops) ⟹ i < oid
        using a-pair list-ops-def snoc.prems spec-ops-id-inc by fastforce
      hence ∃i. i ∈ set (map fst (insertions ops)) ⟹ i < oid
        using insertions-fst-subset by blast
      moreover have list-op-deps oper = set-option ref
        using Insert by (cases ref, auto)
      hence ∃r. r ∈ set-option ref ⟹ r < oid
        using list-ops-def spec-ops-ref-less
        by (metis a-pair last-in-set snoc.prems snoc-eq-iff-butlast)
      ultimately show ?thesis
        using IH insert-ops-def spec-ops-add-last by metis
    next
      case (Delete ref)
      hence insertions (ops @ [a]) = insertions ops
        by (simp add: a-pair insertions-last-del)
      then show ?thesis by (simp add: IH)
    qed
qed
lemma inserted-ids-last-ins:
  shows inserted-ids \((xs @ [(oid, Insert ref val)])\) = inserted-ids \(xs @ [oid]\)
by (simp add: inserted-ids-def map-filter-None map-filter-append)

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

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

qed

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

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

lemma deleted-ids-exist:
  shows ref \(\in\) set \((deleted-ids ops)\) \iff \((\exists i. (i, Delete ref) \in set ops)\)
proof (induction \(ops\) rule: List.rev-induct)
case Nil
  then show ref \(\in\) set \((deleted-ids [])\) \iff \((\exists i. (i, Delete ref) \in set [])\)
by (simp add: deleted-ids-def List.map-filter-def)

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

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

**Lemma interp-ops-list-equiv:**

**shows** \( \text{fst (interp-ops ops)} = \text{interp-ins (insertions ops)} \)

**proof**

*(induction ops rule: List.rev-induct)*

*case Nil*

*have 1: \( \text{fst (interp-ops [])} = [] \)*

*by (simp add: interp-ops-def)*

*have 2: \( \text{interp-ins (insertions [])} = [] \)*

*by (simp add: insertions-def map-filter-def interp-ins-def)*

*show \( \text{fst (interp-ops [])} = \text{interp-ins (insertions [])} \)*

*by (simp add: 1 2)*

*next*

*case (snoc a ops)*

*obtain oid oper where a-pair: \( a = (oid, oper) \)*

*by fastforce*

*then show \( \text{fst (interp-ops (ops @ [a])]} = \text{interp-ins (ops @ [a])} \)*

**proof**

*(cases oper)*

*case (Insert ref val)*

*hence \( \text{insertions (ops @ [a])} = \text{insertions ops @ [(oid, ref)]} \)*

*by (simp add: a-pair insertions-last-ins)*

*hence \( \text{interp-ins (insertions (ops @ [a])]} = \text{insert-spec (interp-ins (insertions ops)) (oid, ref)} \)*

*by (simp add: interp-ins-tail-unfold)*

*moreover have \( \text{fst (interp-ops (ops @ [a])]} = \text{insert-spec (fst (interp-ops ops)) (oid, ref)} \)*

*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 \( \text{insertions (ops @ [a])} = \text{insertions ops} \)*

*by (simp add: a-pair insertions-last-del)*

*moreover have \( \text{fst (interp-ops (ops @ [a])]} = \text{fst (interp-ops ops)} \)*

*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

**Lemma interp-ops-distinct:**

**assumes** list-ops ops

**shows** distinct \( \text{fst (interp-ops ops)} \)

*by (simp add: asms interp-ins-distinct interp-ops-list-equiv list-ops-insertions)*

**Lemma list-order-equiv:**

**shows** \( \text{list-order ops x y} \iff \text{Insert-Spec.list-order (insertions ops) x y} \)

*by (simp add: Insert-Spec.list-order-def List-Spec.list-order-def interp-ops-list-equiv)*

48
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)
  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-ref-older snoc.prems x-pair by blast
      ultimately show ?thesis by auto
    next
      case (Delete ref)
      hence interp-ops (xs @ [x]) = (list, vals(ref := None))
        by (simp add: interp-ops-unfold-last interp-xs x-pair)
      hence dom (snd (interp-ops (xs @ [x]))) = (dom vals) − {ref}
        by simp
      moreover have set (inserted-ids xs) − set (deleted-ids xs) = dom vals
        by (simp add: deleted-ids-def map-filter-simps(2))
    qed
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 \( xs @ \) \([\text{ref}] \)
by (simp add: Delete deleted-ids-last-del x-pair)
hence set \( (inserted-ids (xs @ [x])) − set (deleted-ids (xs @ [x]))\) =
set \( (inserted-ids xs) − (set (deleted-ids xs) ∪ \{\text{ref}\})\)
using calculation(3) by auto
moreover have \( \ldots = set (inserted-ids xs) − set (deleted-ids xs) − \{\text{ref}\}\)
by blast
ultimately show \( \text{thesis by auto} \)
qed

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

lemma insert-spec-inc-length:
assumes \( \text{distinct xs} \) and \( n < \text{length xs} \)
shows \( \text{length} (\text{insert-spec} \ (\text{oid}, \text{Some} (\text{xs} ! n))) = \text{Suc} (\text{length} \ (\text{xs})) \)
using assms proof(induction \( xs \) arbitrary: \( n, \text{simp} \))
case (Cons a xs)
have distinct (a ≠ xs)
  using Cons.prems(1) by auto
then show length (insert-spec (a ≠ xs) (oid, Some ((a ≠ xs) ! n))) = Suc (length (a ≠ xs))
proof(cases n)
case 0
  hence insert-spec (a ≠ xs) (oid, Some ((a ≠ xs) ! n)) = a ≠ oid ≠ xs
  by simp
then show ?thesis
  by simp
next
case (Suc nat)
hence nat < length xs
  using Cons.prems(2) by auto
hence length (insert-spec xs (oid, Some (xs ! nat))) = Suc (length xs)
  using Cons.IH Cons.prems(1) by auto
then show ?thesis
  by (simp add: Suc)
qed
qed

lemma list-split-two-elems:
assumes distinct xs
  and x ∈ set xs and y ∈ set xs
  and x ≠ y
shows ∃pre mid suf. xs = pre @ x ≠ mid @ y ≠ suf ∨ xs = pre @ y ≠ mid @ x ≠ suf
proof —
obtain as bs where as-bs: xs = as @ [x] @ bs
  using assms(2) split-list-first by fastforce
show ?thesis
proof(cases y ∈ set as)
case True
  then obtain cs ds where as = cs @ [y] @ ds
    using assms(3) split-list-first by fastforce
  then show ?thesis
    by (auto simp add: as-bs)
next
case False
  then have y ∈ set bs
    using as-bs assms(3) assms(4) by auto
  then obtain cs ds where bs = cs @ [y] @ ds
    using assms(3) split-list-first by fastforce
  then show ?thesis
    by (auto simp add: as-bs)
qed
qed
3.3 Satisfying all conditions of \( \mathcal{A}_{\text{strong}} \)

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** \( a \in \text{dom} \ (\text{vals}) \leftrightarrow (\exists \text{ref val.} \ (a, \text{Insert ref val}) \in \text{set ops}) \land (\nexists \ i. \ (i, \text{Delete a}) \in \text{set ops}) \)
  **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
  **shows** list-order some-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 \( @ \ [(\text{oid, ins})])
  and ins = make-insert (fst (interp-ops ops)) val k
  and list = fst (interp-ops (ops \( @ \ [(\text{oid, ins})])))
  **shows** list ! (\( \min k \ (\text{length list} - 1) \)) = oid
  **proof** (cases k = 0 \or\ fst (interp-ops ops) = [])
  case True
moreover from this
have make-insert (fst (interp-ops ops)) val k = Insert None val
and min-k: min k (length (fst (interp-ops ops))) = 0
by (cases k, auto)
hence fst (interp-ops (ops @ [((oid, ins)]))) = oid ≠ fst (interp-ops ops)
using assms(2) interp-ops-unfold-last
by (metis fst-conv insert-spec.simps(1) interp-op.simps(1) prod.collapse)
ultimately show ?thesis
by (simp add: min-k assms(3))

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

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

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

theorem list-order-irrefl:
assumes list-ops ops
shows ¬ list-order ops x x
proof –

53
have list-order ops x x \implies False
proof –
assume list-order ops x x
then obtain zs ys zs where split: fst (interp-ops ops) = xs @ [x] @ ys @ [x] @ zs
by (meson List-Spec.list-order-def)
moreover have distinct (fst (interp-ops ops))
by (simp add: assms interp-ops-distinct)
ultimately show False
by (simp add: split)
qed
thus \neg list-order ops x x
by blast
qed

theorem list-order-total:
assumes list-ops ops
and x \in set (fst (interp-ops ops))
and y \in set (fst (interp-ops ops))
and x \neq y
shows list-order ops x y \lor 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 \lor
fst (interp-ops ops) = pre @ y @ mid @ x @ suf
using list-split-two-elems assms by metis
then show list-order ops x y \lor list-order ops y x
by (simp add: list-order-def, blast)
qed

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 \forall i \in set (map fst xs). P i
and P x
shows \forall i \in set (map fst (xs @ [(x, y)])). P i
using assms by (induction xs, auto)

lemma insert-ops-split:
assumes insert-ops ops
and (oid, ref) ∈ set ops
shows ∃ pre suf. ops = pre @ [(oid, ref)] @ suf ∧
(∀ i ∈ set (map fst pre). i < oid) ∧
(∀ i ∈ set (map fst suf). oid < i)
using assms proof(induction ops rule: List.rev-induct)
case Nil
then show ?case by auto
next
case (snoc x xs)
then show ?case
proof(cases x = (oid, ref))
  case True
  moreover from this have ∀ i ∈ set (map fst xs). i < oid
  using last-op-greatest snoc.prems(1) by blast
  ultimately have xs @ [x] = xs @ [(oid, ref)] @ [] ∧
  (∀ i ∈ set (map fst xs). i < oid) ∧
  (∀ i ∈ set (map fst []). oid < i)
  by auto
  then show ?thesis by force
next
case False
hence (oid, ref) ∈ set xs
using snoc.prems(2) by auto
from this obtain pre suf where IH: xs = pre @ [(oid, ref)] @ suf ∧
(∀ i ∈ set (map fst pre). i < oid) ∧
(∀ i ∈ set (map fst suf). oid < i)
using snoc.IH snoc.prems(1) by blast
obtain xi xr where x-pair: x = (xi, xr)
by force
hence distinct (map fst (pre @ [(oid, ref)] @ suf @ [(xi, xr)]))
by (metis IH append.assoc insert-ops-def spec.ops-def snoc.prems(1))
hence xi ≠ oid
by auto
have xi-max: ∀ x ∈ set (map fst (pre @ [(oid, ref)] @ suf)). x < xi
using IH last-op-greatest snoc.prems(1) x-pair by blast
then show ?thesis
proof(cases xi < oid)
  case True
  using xi-max by auto
hence suf = []
  using IH last-in-set by fastforce
ultimately have xs @ [x] = (pre @ [(xi, xr)]) @ [] ∧
(∀ i ∈ set (map fst ((pre @ [(xi, xr)]))). i < oid) ∧
(∀ i ∈ set (map fst []). oid < i)
using dual-order.asgm xi-max by auto
then show ?thesis by (simp add: IH)
next
case False
  hence oid < xi
    using ⟨xi ≠ oid⟩ by auto
  hence ∀ i ∈ set (map fst (suf @([(xi, xr)]))). oid < i
    using IH map-fst-append1 by auto
  hence ∃ as bs cs. ops = as @([(xi, xr)]) @ bs @([(yid, yr)]) @ cs ∧
    (∀ i ∈ set (map fst as). i < xid) ∧
    (∀ i ∈ set (map fst bs). xid < i ∧ i < yid) ∧
    (∀ i ∈ set (map fst cs). yid < i)
  by (simp add: IH x-pair)
  then show ?thesis by blast
qed
qed

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

lemma insert-ops-sorted-oids:
  assumes insert-ops (xs @([(i1, r1)]) @ ys @([(i2, r2)]))
shows \( i_1 < i_2 \)

**proof**

- **have** \( \forall i. i \in \text{set} (\text{map} \ \text{fst} (xs @ [(i1, r1)] @ ys)) \implies i < i_2 \)
  by \( \text{metis append.assoc assms last-op-greatest} \)

- **moreover have** \( i_1 \in \text{set} (\text{map} \ \text{fst} (xs @ [(i1, r1)] @ ys)) \)
  by \( \text{auto} \)

- **ultimately show** \( i_1 < i_2 \)
  by \( \text{blast} \)

**qed**

**lemma** \( \text{insert-ops-subset-last} \):

**assumes** \( \text{insert-ops} (xs @ [x]) \)

- **and** \( \text{insert-ops} \ ys \)

- **and** \( \text{set} \ ys \subseteq \text{set} (xs @ [x]) \)

- **and** \( x \in \text{set} \ ys \)

**shows** \( x = \text{last} \ ys \)

**using** \( \text{assms} \)**

**proof**(induction \( ys \), simp)

**case** \( \text{Cons} \ y \ ys \)

- **then show** \( x = \text{last} (y \# ys) \)

**proof**(cases \( ys = [] \))

- **case** \( \text{True} \)
  - **using** \( \text{Cons.prems(4)} \)** by \( \text{auto} \)

- **next**
  - **case** \( ys\text{-nonempty}: \text{False} \)
    - **have** \( x \neq y \)
      **proof**
      - **obtain** \( \text{mid} \ l \ where \ ys = \text{mid} @ [l] \)
        **using** \( \text{append-butlast-last-id ys\text{-nonempty}} \)** by \( \text{metis} \)
      - **moreover obtain** \( \text{li \ lr \ where \ l = (li, lr)} \)
        **by** \( \text{force} \)
      - **moreover have** \( \forall i. i \in \text{set} (\text{map} \ \text{fst} (y \# \text{mid})) \implies i < li \)
        **by** \( \text{metis last-op-greatest Cons.prems(2) calculation append-Cons} \)
      - **hence** \( \text{fst} \ y < li \)
        **by** \( \text{simp} \)
      - **moreover have** \( \forall i. i \in \text{set} (\text{map} \ \text{fst} \ xs) \implies i < \text{fst} \ x \)
        **using** \( \text{assms(1) last-op-greatest by (metis prod.collapse)} \)
      - **hence** \( \forall i. i \in \text{set} (\text{map} \ \text{fst} (y \# \text{ys})) \implies i \leq \text{fst} \ x \)
        **using** \( \text{Cons.prems(3)} \)** by \( \text{fastforce} \)
      - **ultimately show** \( x \neq y \)
        **by** \( \text{fastforce} \)
      **qed**

- **then show** \( x = \text{last} (y \# ys) \)
  **using** \( \text{Cons.IH Cons.prems insert-ops-rem-cons ys\text{-nonempty}} \)
  **by** \( \text{metis dual-order.trans last-ConsR set-ConsD set-subset-Cons} \)

**qed**

**qed**

**lemma** \( \text{subset-butlast} \):

57
assumes \( set \, xs \subseteq set \, (ys \, @ \,[y]) \)
and last \( xs = y \)
and distinct \( xs \)
shows \( set \, (butlast \, xs) \subseteq set \, ys \)
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 \( \text{fst} \, a \notin set \, (map \, fst \, xs \, @ \, map \, fst \, ys) \)
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 \( \text{fst} \, a \notin set \, (map \, fst \, (butlast \, a \, \# \, xs)) \, @ \, map \, fst \, ys \)
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 \( \text{fst} \, a \notin set \, (map \, fst \, xs \, @ \, map \, fst \, ys) \)
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 \( set \, (map \, fst \, (butlast \, ys)) \subseteq set \, (map \, fst \, xs \, @ \, map \, fst \, ys) \)
by auto
ultimately have \( \text{fst} \, a \notin set \, (map \, fst \, xs \, @ \, map \, fst \, (butlast \, ys)) \)
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 \( set \, (interp-ins \, (xs \, @ \, [(oid, \, ref)])) = set \, (interp-ins \, xs) \lor \)

58
\[ \text{set} \left( \text{interp-ins} \left( \left[ \left[ \text{oid}, \text{ref} \right]\right] \right) \right) = \left( \text{set} \left( \text{interp-ins} \left[ \right] \right) \cup \{ \text{oid} \} \right) \]

by (cases \text{ref}, simp add: interp-ins-tail-unfold, metis insert-spec-nonex insert-spec-set interp-ins-tail-unfold)

**lemma interp-ins-maybe-grow2:**

**assumes** \text{insert-ops} \left( \left[ x \right]\right)  
**shows** \text{set} \left( \text{interp-ins} \left( \left[ x \right]\right) \right) = \text{set} \left( \text{interp-ins} \left[ \right] \right) \cup \{ \text{oid} \} 

**using** \text{assms} interp-ins-maybe-grow by (cases \text{x}, auto)

**lemma interp-ins-maybe-grow3:**

**assumes** \text{insert-ops} \left( \left[ y \right]\right)  
**shows** \exists A. A \subseteq \text{set} \left( \text{map} \text{fst} \left[ y \right]\right) \land \text{set} \left( \text{interp-ins} \left[ y \right] \right) = \text{set} \left( \text{interp-ins} \left[ \right] \right) \cup A 

**using** \text{assms} proof(induction \text{ys} rule: List.rev-induct) case \text{Nil} then show \text{?case} by simp next case \text{Nil} then have \text{insert-ops} \left( \left[ y \right]\right) by (metis append-assoc interp-ins-maybe-grow2 snoc.prems) then obtain A where IH: A \subseteq \text{set} \left( \text{map} \text{fst} \left[ y \right]\right) \land 

\text{set} \left( \text{interp-ins} \left[ y \right] \right) = \text{set} \left( \text{interp-ins} \left[ \right] \right) \cup A 

**using** snoc.IH by blast then show \text{?case} proof(cases \text{set} \left( \text{interp-ins} \left( \left[ x \right]\right) \right) = \text{set} \left( \text{interp-ins} \left[ y \right] \right)) case True moreover have A \subseteq \text{set} \left( \text{map} \text{fst} \left[ y \right]\right) using IH by auto ultimately show \text{?thesis} using IH by auto next case False then have \text{set} \left( \text{interp-ins} \left( \left[ x \right]\right) \right) = \text{set} \left( \text{interp-ins} \left[ y \right] \right) \cup \{ \text{fst} \} \left( \right) 

by (metis append-assoc interp-ins-maybe-grow2 snoc.prems) moreover have A \cup \{ \text{fst} \} \subseteq \text{set} \left( \text{map} \text{fst} \left[ y \right]\right) using IH by auto ultimately show \text{?thesis} using IH Un-assoc by metis qed qed

**lemma interp-ins-ref-nonex:**

**assumes** \text{insert-ops} \text{ops}  
and \text{ops} = \left[ \left[ \text{oid}, \text{Some} \text{ref}\right]\right] \@ \text{ys}  
and \text{ref} \notin \text{set} \left( \text{interp-ins} \left[ \right] \right)  
**shows** \text{oid} \notin \text{set} \left( \text{interp-ins} \left[ \right] \right) 

**using** \text{assms} proof(induction \text{ys} arbitrary: \text{ops} rule: List.rev-induct)
case Nil
then have \( \text{interp-ins} \ \text{ops} = \text{insert-spec} \ (\text{interp-ins} \ \text{xs}) \ (\text{oid}, \ \text{Some ref}) \)
by (simp add: interp-ins-tail-unfold)
moreover have \( \bigwedge_i. \ i \in \text{set} \ (\text{map \ fst} \ \text{xs}) \implies i < \text{oid} \)
using Nil.prems last-op-greatest by fastforce
hence \( \bigwedge_i. \ i \in \text{set} \ (\text{interp-ins} \ \text{xs}) \implies i < \text{oid} \)
by (meson interp-ins-subset subsetCE)
ultimately show \( \text{oid} \notin \text{set} \ (\text{interp-ins} \ \text{ops}) \)
using assms(3) by auto
next
\begin{itemize}
\item case (\text{snoc} \ x \ \text{ys})
\item then have \( \text{insert-ops} \ (\text{xs} @ (\text{oid}, \ \text{Some ref}) \ # \ \text{ys}) \)
by (metis append.assoc append.simps(1) append-Cons insert-ops-appendD)
hence \( \text{IH}: \ \text{oid} \notin \text{set} \ (\text{interp-ins} \ (\text{xs} @ (\text{oid}, \ \text{Some ref}) \ # \ \text{ys})) \)
by (simp add: snoc.IH snoc.prems(3))
moreover have \( \text{distinct} \ (\text{map \ fst} \ (\text{xs} @ (\text{oid}, \ \text{Some ref}) \ # \ \text{ys} @ [\text{x}])) \)
using snoc.prems by (metis append-Cons append-self-conv2 insert-ops-def spec-ops-def)
hence \( \text{fst} \ x \neq \text{oid} \)
using empty-iff by auto
moreover have \( \text{insert-ops} \ ((\text{xs} @ (\text{oid}, \ \text{Some ref}) \ # \ \text{ys}) @ [\text{x}]) \)
using snoc.prems by auto
hence set (\text{interp-ins} \ ((\text{xs} @ (\text{oid}, \ \text{Some ref}) \ # \ \text{ys}) @ [\text{x}])) = 
set (\text{interp-ins} \ ((\text{xs} @ (\text{oid}, \ \text{Some ref}) \ # \ \text{ys})) \cup \{\text{fst} \ x\}) = 
set (\text{interp-ins} \ ((\text{xs} @ (\text{oid}, \ \text{Some ref}) \ # \ \text{ys})) \cup \{\text{fst} \ x\}) \cup \{\text{fst} \ x\}
using interp-ins-maybe-grow2 by blast
ultimately show \( \text{oid} \notin \text{set} \ (\text{interp-ins} \ \text{ops}) \)
using snoc.prems(2) by auto
qed
\end{itemize}

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

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

lemma interp-ins-append-non-memb:
assumes \( \text{insert-ops} \ (\text{pre} @ [(\text{oid}, \ \text{Some ref})] @ \text{suf}) \)
and \( \text{ref} \notin \text{set} \ (\text{interp-ins} \ \text{pre}) \)
shows \( \text{ref} \notin \text{set} \ (\text{interp-ins} \ (\text{pre} @ [(\text{oid}, \ \text{Some ref})] @ \text{suf})) \)
using assms proof(induction \text{suf} rule: List.rev-induct)
case Nil
then show \( \text{ref} \notin \text{set} \ (\text{interp-ins} \ (\text{pre} @ [(\text{oid}, \ \text{Some ref})] @ [])) \)
by (metis append-Nil2 insert-spec-nonex interp-ins-tail-unfold)
next
case (snoc x xs)
hence IH: ref \notin set (interp-ins (pre @ [(oid, Some ref)] @ xs))
by (metis append-assoc insert-ops-rem-last)
moreover have ref < oid
using insert-ops-ref-older snoc.prems(1) by auto
moreover have oid < fst x
using insert-ops-sorted-oids by (metis prod.collapse snoc.prems(1))
have set (interp-ins ((pre @ [(oid, Some ref)] @ xs) @ [x])) =
  set (interp-ins (pre @ [(oid, Some ref)] @ xs)) ∨
  set (interp-ins ((pre @ [(oid, Some ref)] @ xs) @ [x])) =
  set (interp-ins (pre @ [(oid, Some ref)] @ xs)) ∪ {fst x}
by (metis (full-types) append.assoc interp-ins-maybe-grow2 snoc.prems(1))
ultimately show ref \notin set (interp-ins (pre @ [(oid, Some ref)] @ xs @ [x]))
using ⟨oid < fst x⟩ by auto
qed

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

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

lemma interp-ins-find-ref:
assumes insert-ops (xs @ [(oid, Some ref)] @ ys)
and ref ∈ set (interp-ins (xs @ [(oid, Some ref)] @ ys))
shows \exists r. (ref, r) ∈ set xs
proof –
  have ref < oid

61
using assms(1) insert-ops-ref-older by blast
have ref ∈ set (map fst (xs @ [(oid, Some ref)] @ ys))
  by (meson assms(2) interp-ins-subset subsetCE)
then obtain x where x-prop: x ∈ set (xs @ [(oid, Some ref)] @ ys) ∧ fst x = ref
  by fastforce
obtain x where x-pair: x = (ref, xr)
  using prod.exhaust-set x-prop by blast
show (∃r. (ref, r) ∈ set xs)
next
case True
  then show (∃r. (ref, r) ∈ set xs)
    by (metis x-prop prod.collapse)
case False
hence (ref, xr) ∈ set ([(oid, Some ref)] @ ys)
  using x-prop x-pair by auto
hence (ref, xr) ∈ set ys
  using (ref < oid) x-prop
by (metis append-Cons append-self-cone2 fst-cone min.strict-order_iff set-ConsD)
then obtain as bs where ys = as @ (ref, xr) # bs
  by (meson split-list)
hence insert-ops ([xs @ [(oid, Some ref)] @ as @ [(ref, xr)] @ bs)
  using assms(1) by auto
hence insert-ops (xs @ [(oid, Some ref)] @ as @ [(ref, xr)])
  using insert-ops-appendD by blast
hence oid < ref
  using insert-ops-sorted-oids by auto
then show ?thesis
  using ⟨ref < oid⟩ by force
qed

4.3 Lemmas about list-order

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

lemma list-order-insert-ref:
  assumes insert-ops (ops @ [(oid, Some ref)])
  and ref ∈ set (interp-ins ops)
  shows list-order (ops @ [(oid, Some ref)]) ref oid
proof
  have interp-ins (ops @ [(oid, Some ref)]) = insert-spec (interp-ins ops) (oid, Some ref)
    by (simp add: interp-ins-tail-unfold)
moreover obtain \(xs\ ys\) where \(\text{interp-ins}\ ops = xs \cons \ref \cons ys\)

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

hence \(\text{insert-spec}\ (\text{interp-ins}\ ops)\ (\text{oid}, \text{Some}\ ref) = xs \cons \ref \cons [] \cons \text{oid} \cons ys\)

using \(\text{assms(1)}\) \text{insert-after-ref} \text{interp-ins-distinct} by \text{fastforce}

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

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

qed

\begin{lemma}{list-order-insert-none:}
\begin{assumes}
\begin{align*}
\text{insert-ops} & (\text{ops} \cons [(\text{oid}, \text{None})]) \\
\text{and}\ & x \in \text{set}\ (\text{interp-ins}\ \text{ops}) \\
\text{shows}\ & \text{list-order}\ (\text{ops} @ [(\text{oid}, \text{None})])\ \text{oid}\ x
\end{align*}
\end{assumes}
\begin{proof}
\begin{have}
\begin{align*}
\text{interp-ins} & (\text{ops} @ [(\text{oid}, \text{None})]) = \text{insert-spec} (\text{interp-ins}\ \text{ops})\ (\text{oid}, \text{None}) \\
\text{by}\ & (\text{simp add: interp-ins-tail-unfold})
\end{align*}
\end{have}

moreover obtain \(xs\ ys\) where \(\text{interp-ins}\ \text{ops} = xs \cons [x] \cons ys\)

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

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

by \text{simp}

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

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

qed

\begin{lemma}{list-order-insert-between:}
\begin{assumes}
\begin{align*}
\text{insert-ops} & (\text{ops} @ [(\text{oid}, \text{Some}\ ref)]) \\
\text{and}\ & \text{list-order}\ \text{ops}\ \text{ref}\ \text{x} \\
\text{shows}\ & \text{list-order}\ (\text{ops} @ [(\text{oid}, \text{Some}\ ref)])\ \text{oid}\ \text{x}
\end{align*}
\end{assumes}
\begin{proof}
\begin{have}
\begin{align*}
\text{interp-ins} & (\text{ops} @ [(\text{oid}, \text{Some}\ ref)]) = \text{insert-spec} (\text{interp-ins}\ \text{ops})\ (\text{oid}, \text{Some}\ ref) \\
\text{by}\ & (\text{simp add: interp-ins-tail-unfold})
\end{align*}
\end{have}

moreover obtain \(xs\ ys\ zs\) where \(\text{interp-ins}\ \text{ops} = xs \cons \ref \cons ys \cons [x] \cons zs\)

using \(\text{assms}\) \text{list-orderE} by \text{blast}

moreover have ... = \(xs \cons \ref \cons @ (ys \cons [x] \cons zs)\)

by \text{simp}

moreover have \(\text{distinct}\ (xs @ \ref @ (ys @ [x] @ zs))\)

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

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

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

moreover have ... = \(xs @ [\ref] @ [\text{oid}] @ ys @ [x] @ zs\)

by \text{simp}

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

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

qed
4.4 The insert-seq predicate

The predicate \emph{insert-seq start ops} is true iff \emph{ops} is a list of insertion operations that begins by inserting after \emph{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.

\begin{verbatim}
inductive insert-seq :: 'oid option ⇒ ('oid × 'oid option) list ⇒ bool where
  insert-seq start [(oid, start)] |
  insert-seq start (list @ [(prev, ref)])
  ⇒ insert-seq start (list @ [(prev, ref), (oid, Some prev)])
\end{verbatim}

\textbf{lemma} insert-seq-nonempty:

\textbf{assumes} \(\text{insert-seq start } xs\)
\textbf{shows} \(xs \neq []\)
\textbf{using} \(\text{assms by (induction rule: insert-seq.induct, auto)}\)

\textbf{lemma} insert-seq-hd:

\textbf{assumes} \(\text{insert-seq start } xs\)
\textbf{shows} \(\exists \text{ oid. hd } xs = (\text{oid, start})\)
\textbf{using} \(\text{assms by (induction rule: insert-seq.induct, simp, metis append-self-conv2 hd-append2 list.sel(1))}\)

\textbf{lemma} insert-seq-rem-last:

\textbf{assumes} \(\text{insert-seq start } (xs @ [x])\)
\textbf{and} \(xs \neq []\)
\textbf{shows} \(\text{insert-seq start } xs\)
\textbf{using} \(\text{assms insert-seq.cases by fastforce}\)

\textbf{lemma} insert-seq-butlast:

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

\textbf{lemma} insert-seq-last-ref:

\textbf{assumes} \(\text{insert-seq start } (xs @ [(xi, xr), (yi, yr)])\)
\textbf{shows} \(yr = \text{Some } xi\)
\textbf{using} \(\text{assms insert-seq.cases by fastforce}\)

\textbf{lemma} insert-seq-start-none:
\textbf{assumes} insert-ops \texttt{ops} \\
\textbf{and} insert-seq \texttt{None \textit{x}s} \textbf{and} insert-ops \texttt{x}s \\
\textbf{and} \texttt{set \textit{x}s} \subseteq \texttt{set \textit{ops}} \\
\textbf{shows} \texttt{\forall i \in set (map \textit{fst} \textit{x}s). i \in set (interp-ins \texttt{ops})} \\
\textbf{using} \texttt{assms proof (induction \textit{x}s rule: List.rev-induct, simp)} \\
\textbf{case} (\texttt{snoc x \textit{xs}}) \\
\textbf{then have} \texttt{IH: \forall i \in set (map \textit{fst} \textit{xs}), i \in set (interp-ins \texttt{ops})} \\
\textbf{by} (metis \texttt{Nil-is-map-conv append-is-Nil-conv insert-ops-appendD} insert-seq-rem-last \\
\texttt{le-supE list.simps (3) set-append split-list}) \\
\textbf{then show} \texttt{\forall i \in set (map \textit{fst} (\textit{x}s @ [\textit{x}])). i \in set (interp-ins \texttt{ops})} \\
\textbf{proof} (cases \textit{x}s = []) \\
\textbf{case} \texttt{True} \\
\textbf{then obtain} \texttt{oid where \textit{x}s @ [\textit{x}] = [(oid, None)]} \\
\textbf{using} \texttt{insert-seq-hd snoc.prems (2) by fastforce} \\
\textbf{hence} (oid, None) \in set \texttt{ops} \\
\textbf{using} \texttt{snoc.prems (4) by auto} \\
\textbf{then obtain as \texttt{bs where} \texttt{ops = as @ (oid, None) \# bs}} \\
\textbf{by} (meson \texttt{split-list}) \\
\textbf{hence} \texttt{ops = (as @ [(oid, None)]) @ bs} \\
\textbf{by} (simp add: \texttt{ops = as @ (oid, None) \# bs}) \\
\textbf{moreover have} \texttt{oid \in set (interp-ins (as @ [(oid, None)]))} \\
\textbf{by} (simp add: interp-ins-last-None) \\
\textbf{ultimately have} \texttt{oid \in set (interp-ins \texttt{ops})} \\
\textbf{using} \texttt{interp-ins-monotonic snoc.prems(1) by blast} \\
\textbf{then show} \texttt{\forall i \in set (map \textit{fst} (\textit{x}s @ [\textit{x}])). i \in set (interp-ins \texttt{ops})} \\
\textbf{using} \texttt{(\textit{x}s @ [\textit{x}] = [(oid, None)]) by auto} \\
\textbf{next} \\
\textbf{case} \texttt{False} \\
\textbf{then obtain} \texttt{rest \textit{y} where snoc-y: \textit{x}s = rest @ [\textit{y}]} \\
\textbf{using} \texttt{append-butlast-last-id by metis} \\
\textbf{obtain} \texttt{yi yr xi xr where yx-pairs: y = (yi, yr) \land x = (xi, xr)} \\
\textbf{by} \textbf{force} \\
\textbf{then have} \texttt{xr = Some yi} \\
\textbf{using} \texttt{insert-seq-last-ref snoc.prems (2) snoc-y by fastforce} \\
\textbf{have} \texttt{yi < xi} \\
\textbf{using} \texttt{insert-ops-sorted-oids snoc-y yx-pairs snoc.prems (3)} \\
\textbf{by} (metis \texttt{no-types, lifting append-eq-append-conj2}) \\
\textbf{have} (yi, yr) \in set \texttt{ops} \texttt{and (xi, Some yi)} \in set \texttt{ops} \\
\textbf{using} \texttt{snoc.prems (4) snoc-y yx-pairs (xr = Some yi) by auto} \\
\textbf{then obtain as \texttt{bs \texttt{cs where} ops-split: ops = as @ [(yi, yr)] @ bs @ [(xi, Some yi)] @ cs}} \\
\textbf{using} \texttt{insert-ops-split-2 (yi < xi) snoc.prems (1) by blast} \\
\textbf{hence} \texttt{yi \in set (interp-ins (as @ [(yi, yr)] @ bs))} \\
\textbf{proof --} \\
\textbf{have} \texttt{yi \in set (interp-ins \texttt{ops})} \\
\textbf{by} (simp add: \texttt{IH snoc-y yx-pairs}) \\
\textbf{moreover have} \texttt{ops = (as @ [(yi, yr)] @ bs) @ ([(xi, Some yi)] @ cs)} \\
\textbf{using} \texttt{ops-split by simp} \\
\textbf{moreover have} \texttt{yi \in set (map \textit{fst} (as @ [(yi, yr)] @ bs))} 

65
ultimately show \( \text{thesis} \) using \( \text{snoc.prems}(1) \) interp-ins-append-forward by blast

qed

hence \( x_i \in \text{set} (\text{interp-ins} ((\text{as} @ [(y_i, y_r)] @ bs) @ [\{x_i, \text{Some } y_i\}] @ cs)) \)

using \( \text{snoc.prems}(1) \) interp-ins-append-memb ops-split by force

hence \( x_i \in \text{set} (\text{interp-ins } \text{ops}) \)

by (simp add: ops-split)

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

using \( \text{IH } yx\text{-pairs} \) by auto

qed

lemma insert-seq-after-start:
assumes \( \text{insert-ops } \text{ops} \)
and \( \text{insert-seq } (\text{Some } \text{ref}) \) \( \text{xs} \) and \( \text{insert-ops } \text{xs} \)
and \( \text{set } \text{xs} \subseteq \text{set } \text{ops} \)
and \( \text{ref} \in \text{set} (\text{interp-ins } \text{ops}) \)
shows \( \forall i \in \text{set} (\text{map } \text{fst } \text{xs} @ [x]). \text{i } \in \text{set} (\text{interp-ins } \text{ops}) \)
using assms proof (induction \( \text{xs} \) rule: \( \text{List.rev-induct} \), simp)

case \( (\text{snoc } x \text{xs}) \)

have \( \text{IH}; \forall i \in \text{set} (\text{map } \text{fst } \text{xs}). \text{list-order } \text{ops } \text{ref } i \)

using \( \text{snoc.IH } \text{snoc.prems insert-seq-rem-last insert-ops-appendD} \)
by (metis \( \text{Nil-is-map-conv Un-subset-iff empty-set equals0D set-append} \))

moreover have \( \text{list-order } \text{ops } \text{ref } (\text{fst } x) \)

proof (cases \( \text{xs} = [] \))

case True

have \( \text{sn} \text{d } x = \text{Some } \text{ref} \)

using \( \text{insert-seq-hd } \text{snoc.prems}(2) \) by fastforce

moreover have \( x \in \text{set } \text{ops} \)

using \( \text{snoc.prems}(4) \) by auto

then obtain \( \text{cs } \text{ds} \) where \( \text{x-split: ops = cs @ x # ds} \)

by (meson \( \text{split-list} \))

have \( \text{list-order } (\text{cs } @ [\{\text{fst } x, \text{Some } \text{ref}\}]) \) \( \text{ref } (\text{fst } x) \)

proof –

have \( \text{insert-ops } (\text{cs } @ [\{\text{fst } x, \text{Some } \text{ref}\}]) @ \text{ds} \)

using \( \text{x-split } (\text{snd } x = \text{Some } \text{ref}) \)

by (metis \( \text{append-Cons } \text{append-self-conv2 prod.collapse } \text{snoc.prems}(1) \))

moreover from this obtain \( \text{rr} \) where \( (\text{ref}, \text{rr}) \in \text{set } \text{cs} \)

using interp-ins-find-ref \( \text{x-split } (\text{snd } x = \text{Some } \text{ref}) \) assms(5)

by (metis \( \text{no-types, lifting} \) append-Cons \( \text{append-self-conv2 prod.collapse} \))

hence \( \text{ref} \in \text{set } (\text{interp-ins } \text{cs}) \)

proof –

have \( \text{ops = cs } @ [\{\text{fst } x, \text{Some } \text{ref}\}] @ \text{ds} \)

by (metis \( \text{x-split } (\text{snd } x = \text{Some } \text{ref}) \) append-Cons \( \text{append-self-conv2 prod.collapse} \))

thus \( \text{ref} \in \text{set } (\text{interp-ins } \text{cs}) \)

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-cone append-eq-append-conv2 append-self-cone2
prod-collapse)
ultimately show list-order ops ref (fst x)
  using list-order-append snoc.
prems (1) by blast
next
case False
  then obtain rest y where snoc-y: xs = rest @ [y]
  using append-butlast-last-id by metis
  obtain yi yr xi xr where yx-pairs: y = (yi, yr) ∧ x = (xi, xr)
  by force
  then have xr = Some yi
  using insert-seq-last-ref snoc.
  prems (2) snoc-y by fastforce
  have yi < xi
  using insert-ops-sorted-oids snoc-y yx-pairs snoc.prems (1)
  by (metis (no-types, lifting) append-eq-append-conv2)
  have (yi, yr) ∈ set ops and (xi, Some yi) ∈ set ops
  using snoc.prems (4) snoc-y yx-pairs (xr = Some yi) by auto
  then obtain as bs cs where ops-split: ops = as @ [(yi, yr)] @ bs @ [(xi, Some
yi)] @ cs
  using insert-ops-split-2 (yi < xi) snoc.prems (1) by blast
  have list-order ops ref yi
  by (simp add: IH snoc-y yx-pairs)
moreover have list-order (as @ [(yi, yr)] @ bs @ [(xi, Some yi)]) yi xi
proof
  have insert-ops ((as @ [(yi, yr)] @ bs @ [(xi, Some yi)]) @ cs)
  using ops-split snoc.prems (1) by auto
  hence insert-ops ((as @ [(yi, yr)] @ bs) @ [(xi, Some yi)])
  using insert-ops-appendD by fastforce
  moreover have yi ∈ set (interp-ins ops)
  using /list-order ops ref yi: list-order-memb2 by auto
  hence yi ∈ set (interp-ins (as @ [(yi, yr)] @ bs))
  using interp-ins-append-non-memb ops-split snoc.prems (1) by force
ultimately show ?thesis
  using list-order-insert-ref by force
qed
hence list-order ops yi xi
  by (metis append-assoc list-order-append ops-split snoc.prems (1))
ultimately show list-order ops ref (fst x)
  using list-order-trans snoc.prems (1) yx-pairs by auto
qed
ultimately show ∀ i ∈ set (map fst (xs @ [x])). list-order ops ref i
  by auto
qed
lemma insert-seq-no-start:
 assumes insert-ops ops
 and insert-seq (Some ref) xs and insert-ops xs
 and set xs ⊆ set ops
 and ref ∉ set (interp-ins ops)
 shows ∀ i ∈ set (map fst xs), i ∉ set (interp-ins ops)
 using assms proof (induction xs rule: List.rev-induct, simp)
 case (snoc x xs)
 have IH: ∀ i ∈ set (map fst xs), i ∉ set (interp-ins ops)
 using snoc.IH snoc.prems insert-seq-rem-last insert-ops-appendD
 by (metis append-is-Nil-conv le-sup-iff list.map-disc-iff set-append split-list-first)
 obtain as bs where ops = as @ x # bs
 using snoc.prems(4) by (metis split-list last-in-set snoc-eq-iff-butlast subset-code(1))
 have fst x ∉ set (interp-ins ops)
 proof (cases xs = [])
 case True
 then obtain xi where x = (xi, Some ref)
 using insert-seq-hd snoc.prems(2) by force
 moreover have ref ∉ set (interp-ins as)
 using interp-ins-monotonic snoc.prems(1) snoc.prems(5) : ops = as @ x # bs
 by blast
 ultimately have xi ∉ set (interp-ins (as @ [x] @ bs))
 using snoc.prems(1) by (simp add: interp-ins-ref-nonex : ops = as @ x # bs)
 then show fst x ∉ set (interp-ins ops)
 by (simp add: : ops = as @ x # bs) (x = (xi, Some ref))
 next
 case xs-nonempty: False
 then obtain y where xs = (butlast xs) @ [y]
 by (metis append-butlast-last-id)
 moreover from this obtain yi yr xi xr where y = (yi, yr) ∧ x = (xi, xr)
 by fastforce
 moreover from this have xr = Some yi
 using insert-seq.cases snoc.prems(2) calculation by fastforce
 moreover have yi ∉ set (interp-ins ops)
 using IH calculation
 by (metis Nil-is-map-conv fst-conv last-in-set last-map snoc-eq-iff-butlast)
 hence yi ∉ set (interp-ins as)
 using : ops = as @ x # bs interp-ins-monotonic snoc.prems(1) by blast
 ultimately have xi ∉ set (interp-ins (as @ [x] @ bs))
 using interp-ins-ref-nonex snoc.prems(1) : ops = as @ x # bs
 by fastforce
 then show fst x ∉ set (interp-ins ops)
 by (simp add: : ops = as @ x # bs) (y = (yi, yr) ∧ x = (xi, xr))
 qed
 then show ∀ i ∈ set (map fst (xs @ [x])). i ∉ set (interp-ins ops)
 using IH by auto
 qed

68
4.5 The proof of no interleaving

**lemma** no-interleaving-ordered:

**assumes** insert-ops ops
and insert-seq start xs and insert-ops xs
and insert-seq start ys and insert-ops ys
and set xs ⊆ set ops and set ys ⊆ set ops
and distinct (map fst xs @ map fst ys)
and fst (hd xs) < fst (hd ys)
and \( \forall r. \) start = Some r \( \Rightarrow \) r ∈ set (interp-ins ops)

**shows** \( \forall x \in \text{set } \text{map fst xs}. \forall y \in \text{set } \text{map fst ys}. \text{list-order ops } y \text{ x} \) ∧
(\( \forall r. \) start = Some r \( \Rightarrow \) \( \forall x \in \text{set } \text{map fst xs}.\) \text{list-order ops } r \text{x} \) ∧
(\( \forall y \in \text{set } \text{map fst ys}.\) \text{list-order ops } r \text{y} ))

**using** asms proof(induction ops arbitrary: xs ys rule: List.rev-induct, simp)

**case** (snoc a ops)
then have insert-ops ops
using insert-ops-rem-last by auto

**consider** (a-in-xs) a ∈ set xs | (a-in-ys) a ∈ set ys | (neither) a ∉ set xs ∧ a ∉ set ys
by blast
then show ?case

**proof** (cases)
case a-in-xs
then have a ∉ set ys
using snoc.prems(8) by auto
hence set ys ⊆ set ops
using snoc.prems(7) DiffE by auto
from a-in-xs have a = last xs
using insert-ops-subset-last snoc.prems by blast
have IH: \( \forall x \in \text{set } \text{map fst } \text{butlast xs}. \forall y \in \text{set } \text{map fst ys}. \text{list-order ops } y \text{x} \) ∧
\( \forall r. \) start = Some r \( \Rightarrow \) \( \forall x \in \text{set } \text{map fst } \text{butlast xs}.\) \text{list-order ops } r \text{x} \) ∧
(\( \forall y \in \text{set } \text{map fst } \text{ys}.\) \text{list-order ops } r \text{y} )

**proof** (cases xs = [a])
case True
moreover have \( \forall r. \) start = Some r \( \Rightarrow \) \( \forall y \in \text{set } \text{map fst } \text{ys}.\) \text{list-order ops } r \text{y} )
using insert-seq-after-start (insert-ops ops) (set ys ⊆ set ops) snoc.prems
by (metis append-Nil2 calculation insert-seq-hd interp-ins-append-non-memb
list.sel(1))
ultimately show ?thesis by auto

next
case xs-longer: False
from \( a = \text{last xs} \) have set (butlast xs) ⊆ set ops
using snoc.prems by simp add: distinct-fst subset-butlast
moreover have insert-seq start (butlast xs)
using insert-seq-butlast insert-seq-nonempty xs-longer (a = last xs) snoc.prems(2)
by blast
moreover have insert-ops (butlast xs)
using snoc.prems(2) snoc.prems(3) insert-ops-appendD
by (metis append-butlast-last-id insert-seq-nonempty)
moreover have distinct (map fst (butlast xs) @ map fst ys)
using distinct-append-butlast1 snoc.prems(8) by blast
moreover have set ys ⊆ set ops
using ⟨a ∈ set ys⟩ ⟨set ys ⊆ set ops⟩ by blast
moreover have hd (butlast xs) = hd xs
by (metis hd-in-set insert-ops-memb-ref-older insert-seq-nonempty snoc.prems(2))
hence fst (hd (butlast xs)) < fst (hd ys)
by (simp add: snoc.prems(9))
moreover have \(\forall r. \text{start} = \text{Some } r \implies r \in \text{set } (\text{interp-ins } \text{ops})\)
proof
fix r
assume start = Some r
then obtain xid where hd xs = (xid, Some r)
using insert-seq-hd snoc.prems(2) by auto
hence r < xid
by (metis hd-in-set insert-ops-memb-ref-older insert-seq-nonempty snoc.prems(2) snoc.prems(3))
moreover have xid < fst a
proof
have xs = (butlast xs) @ [a]
using snoc.prems(2) insert-seq-nonempty \(\langle a = \text{last } xs \rangle\) 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)
by (metis hd-in-set insert-ops-memb-ref-older insert-seq-nonempty snoc.prems(2))
hence xid ∈ set (butlast xs)
by (metis in-set-zipE zip-map-fst-snd)
ultimately show \(?thesis\)
using snoc.prems(3) last-op-greatest by (metis prod.collapse)
qed
ultimately have \(r \neq \text{fst } a\)
using dual-order.asym by blast
thus \(r \in \text{set } (\text{interp-ins } \text{ops})\)
using snoc.prems(1) snoc.prems(10) interp-ins-maybe-grow2 \(\langle \text{start} = \text{Some } r \rangle\) by blast
qed
moreover have \(\forall x \in \text{set } (\text{map fst } (\text{butlast } xs)). x \in \text{set } (\text{interp-ins } \text{ops})\)
proof(cases start)
  case None
  moreover have set (butlast xs) ⊆ set ops
  using \(\langle a = \text{last } xs \rangle\) distinct-map snoc.prems(6) snoc.prems(8) subset-butlast
  ultimately have x-exists: \(\forall x \in \text{set } (\text{map fst } (\text{butlast } xs)). x \in \text{set } (\text{interp-ins } \text{ops})\)
  proof
  qed
ultimately show \( \text{thesis} \)
  using insert-seq-start-none ⟨insert-ops ops⟩ snoc.prems
  by (metis append-butlast-last-id butlast.simps(2) empty-iiff empty-set
  insert-ops-rem-last insert-seq-butlast insert-seq-nonempty list.simps(8))

next

  case (Some a)
  then show \( \text{thesis} \)
    using IH list-order-memb2 by blast
  qed

moreover have \( \forall y \in \text{set} (\text{map } \text{fst } \text{ys}). \text{list-order} (\text{ops } @ [a]) y (\text{fst } a) \)
proof (cases \( \text{xs} = [a] \))
  case \( \text{xs-a} : \text{True} \)
  have \( \text{ys} \neq [] \implies \text{False} \)
    proof (−)
      assume \( \text{ys} \neq [] \)
      then obtain \( y_i \) where \( y_i \text{-def } : \text{ys} = (y_i, \text{start}) \# (\text{tl } \text{ys}) \)
        by (metis hd-Cons-tl insert-seq-hd snoc.prems(4))
      hence \( (y_i, \text{start}) \in \text{set } \text{ops} \)
        by (metis ⟨\text{set } \text{ys} \subseteq \text{set } \text{ops}⟩ list.set-intros(1) subsetCE)
      hence \( y_i \in \text{set} (\text{map } \text{fst } \text{ops}) \)
        by force
      hence \( y_i < \text{fst } a \)
        using snoc.prems(1) last-op-greatest by (metis prod.collapse)
      moreover have \( \text{fst } a < y_i \)
        by (metis \( y_i \text{-def } \text{xs-a } \text{fst-conv } \text{list.sel}(1) \text{snoc.prems}(9) \))
      ultimately show \( \text{False} \)
        using less-not-sym by blast
    qed

then show \( \forall y \in \text{set} (\text{map } \text{fst } \text{ys}). \text{list-order} (\text{ops } @ [a]) y (\text{fst } a) \)
  using insert-seq-nonempty snoc.prems(4) by blast
next

  case \( \text{xs-longer} : \text{False} \)
  hence butlast-split: butlast \( \text{xs} = (\text{butlast } (\text{butlast } \text{xs})) @ [\text{last } (\text{butlast } \text{xs})] \)
    using (a = last xs) insert-seq-butlast insert-seq-nonempty snoc.prems(2) by fastforce
  hence ref-exists: \( \text{fst } (\text{last } (\text{butlast } \text{xs})) \in \text{set } (\text{interp-ins } \text{ops}) \)
    using x-exists by (metis last-in-set last-map map-is-Nil-conv snoc-eq-iff-butlast)
  moreover from butlast-split have \( \text{xs} = (\text{butlast } (\text{butlast } \text{xs})) @ [\text{last } (\text{butlast } \text{xs}), a] \)
    by (metis (a = last xs) append.assoc append-butlast-last-id butlast.simps(2)
     insert-seq-nonempty last-ConsL last-ConsR list.simps(3) snoc.prems(2))
  hence snd a = Some (fst (last (butlast xs)))
    using snoc.prems(2) insert-seq-last-ref by (metis prod.collapse)
  hence list-order (ops @ [a]) (fst (last (butlast xs))) (fst a)
    using list-order-insert-ref ref-exists snoc.prems(1) by (metis prod.collapse)
  moreover have \( \forall y \in \text{set} (\text{map } \text{fst } \text{ys}). \text{list-order} (\text{ops } @ [a]) y (\text{fst } (\text{last } (\text{butlast } \text{xs}))) \)
    by (metis IH butlast-split last-in-set last-map map-is-Nil-conv snoc-eq-iff-butlast)
  hence \( \forall y \in \text{set} (\text{map } \text{fst } \text{ys}). \text{list-order} (\text{ops } @ [a]) y (\text{fst } (\text{last } (\text{butlast } \text{xs}))) \)
ultimately show $\forall y \in \text{set (map fst ys)}. \text{list-order (ops @ [a]) y (fst a)}$

using \text{snoc.prems(1)} by blast

moreover have $\text{map-fst-xs: map fst xs} = \text{map fst (butlast xs)} @ \text{map fst [last xs]}$

by (metis \text{append-butlast-last-id} \text{insert-seq-nonempty} \text{map-append} \text{snoc.prems(2)})

hence set (map fst xs) = set (map fst (butlast xs)) \cup \{\text{fst a}\}$

by (simp add: \{a = last xs\})

moreover have $\forall r. \text{start} = \text{Some r} \rightarrow \text{list-order (ops @ [a]) r (fst a)}$

using \text{snoc.prems} by (cases start, auto simp add: \text{insert-seq-after-start} \{a = last xs; map-fst-xs\})

ultimately show $(\forall x \in \text{set (map fst xs)}). \forall y \in \text{set (map fst ys)}, \text{list-order (ops @ [a]) x y})$

using \text{snoc.prems(1)} by (simp add: \text{list-order-append})

next

case a-in-ys

then have a \notin \text{set xs}

using \text{snoc.prems(8)} by auto

hence set xs \subseteq set ops

using \text{snoc.prems(6)} \text{DiffE} byauto

from a-in-ys have a = last ys

using \text{insert-ops-subset-last snoc.prems} by blast

have \text{IH: (\forall x \in \text{set (map fst xs)}). \forall y \in \text{set (map fst (butlast ys))}. \text{list-order ops y x})} \land

$(\forall r. \text{start} = \text{Some r} \rightarrow (\forall x \in \text{set (map fst xs)}). \text{list-order ops r x}) \land

(\forall y \in \text{set (map fst (butlast ys))}. \text{list-order ops r y}))$

proof (cases ys = [a])

case True

moreover have $\forall r. \text{start} = \text{Some r} \rightarrow (\forall y \in \text{set (map fst xs)}). \text{list-order ops r y})$

using \text{insert-seq-after-start} (\text{insert-ops ops}) (\text{set xs} \subseteq \text{set ops} \text{snoc.prems}

by (metis \text{append-Nil2 calculation insert-seq-hd interp-ins-append-non-memb list.sel(1)})

ultimately show \text{thesis} by auto

next

case \text{ys-longer: False}

from (a = last ys) have set (butlast ys) \subseteq set ops

using \text{snoc.prems} by (simp add: \text{distinct-fst subset-butlast})

moreover have insert-seq start (butlast ys)

using \text{insert-seq-butlast insert-seq-nonempty ys-longer (a = last ys; snoc.prems(4))}

by blast

moreover have insert-ops (butlast ys)

using \text{snoc.prems(4)} \text{snoc.prems(5)} \text{insert-ops-appendD}

by (metis \text{append-butlast-last-id} \text{insert-seq-nonempty})

72
moreover have distinct (map fst xs @ map fst (butlast ys))
  using distinct-append-butlast2 snoc.prems(8) by blast
moreover have set xs ⊆ set ops
  using (a ∉ set xs) (set xs ⊆ set ops) by blast
moreover have hd (butlast ys) = hd ys
by (metis append-butlast-last-id calculation(2) hd-append2 insert-seq-nonempty
snoc.prems(4))
  hence fst (hd xs) < fst (hd (butlast ys))
  by (simp add: snoc.prems(9))
moreover have \( r \). start = Some r \( r \in \{\text{interp-ins } \text{ops} \}\)
proof –
  fix r
  assume start = Some r
  then obtain yid where hd ys = (yid, Some r)
  using insert-seq-hd snoc.prems(4) by auto
  hence r < yid
  by (metis hd-in-set insert-ops-memb-ref-older insert-seq-nonempty snoc.
prems(4) snoc.prems(5))
moreover have yid < fst a
proof –
  have ys = (butlast ys) @ [a]
  using snoc.prems(4) insert-seq-nonempty (a = last ys) by fastforce
moreover have (yid, Some r) ∈ set (butlast ys)
  using hd ys = (yid, Some r) insert-seq-nonempty list.set.sel(1)
  by (metis :hd (butlast ys) = hd ys; insert-seq start (butlast ys)))
  hence yid ∈ set (map fst (butlast ys))
  by (metis in-set-zipE zip-map-fst-snd)
ultimately show \(?thesis\)
  using snoc.prems(5) last-op-greatest by (metis prod-collapse)
qed
ultimately have r ≠ fst a
  using dual-order.asym by blast
thus r ∈ set (interp-ins ops)
using snoc.prems(1) snoc.prems(10) interp-ins-maybe-grow2 (start = Some
r) by blast
qed
ultimately show \(?thesis\)
  using (insert-ops ops) snoc.IH snoc.prems(2) snoc.prems(3) by blast
qed
moreover have \( \forall x \in \{\text{map fst xs} \}. \text{list-order} \ (\text{ops} @ [a]) \ (\text{fst a}) \ x \)
proof(cases ys = [a])
case ys-a: True
  then show \( \forall x \in \{\text{map fst xs} \}. \text{list-order} \ (\text{ops} @ [a]) \ (\text{fst a}) \ x \)
  proof(cases start)
    case None
    then show \(?thesis\)
      using insert-seq-start-none list-order-insert-none snoc.prems
      by (metis :insert-ops ops) set xs ⊆ set ops fs Conj list.sel(1)
ys-a)

next
  case (Some r)
  moreover from this have ∀ x ∈ set (map fst xs). list-order ops r x
    using IH by blast
  ultimately show ?thesis
    using snoc.prems(1) snoc.prems(4) list-order-insert-between
    by (metis fst-conv insert-seq-hd list.sel(1) ys-a)
qed

next
  case ys-longer: False
  hence butlast-split: butlast ys = (butlast (butlast ys)) @ [last (butlast ys)]
  using (a = last ys) insert-seq-butlast insert-seq-nonempty snoc.prems(4)
  by fastforce
  moreover from this have ys = (butlast (butlast ys)) @ [last (butlast ys), a]
    by (metis (a = last ys) append.assoc append-buttlast-last-id butlast.simps(2)
      insert-seq-nonempty last-ConsL last-ConsR list.simps(3) snoc.prems(4))
  hence snd a = Some (fst (last (butlast ys)))
  using snoc.prems(4) insert-seq-last-ref by (metis prod.collapse)
  moreover have ∀ x ∈ set (map fst xs). list-order ops (fst (last (butlast ys))) x
    by (metis IH butlast-split last-in-set last-map is-Nil-conv snoc-eq-iff-butlast)
  ultimately show ∀ x ∈ set (map fst xs). list-order (ops @ [a]) (fst a) x
    using list-order-insert-between snoc.prems(1) by (metis prod.collapse)
qed

next
  case neither
  hence set xs ⊆ set ops and set ys ⊆ set ops
  using snoc.prems(6) snoc.prems(7) DiffE by auto
  have (∀ r. start = Some r → r ∈ set (interp-ins ops)) ∨ (xs = [] ∧ ys = [])
  proof(cases xs)
    case Nil
    then show ?thesis using insert-seq-nonempty snoc.prems(2) by blast
  next
    case xs-nonempty: (Cons x xs)
    have (r. start = Some r → r ∈ set (interp-ins ops)
proof
  fix r
  assume start = Some r
  then obtain xi where x = (xi, Some r)
    using insert-seq-hd xs-nonempty snoc.prems(2) by fastforce
  hence (xi, Some r) ∈ set ops
    using (set xs ⊆ set ops) xs-nonempty by auto
  hence r < xi
    using last-op-greatest snoc.prems(1) by blast
  moreover have xi ∈ set (map fst ops)
    using ⟨xi, Some r⟩ ∈ set ops by force
  hence xi < fst a
    using list-order ops y x
  ultimately have fst a ≠ r
    using order.asym by blast
  then show r ∈ set (interp-ins ops)
    using snoc.prems(1) snoc.prems(10) interp-ins-maybe-grow2 ⟨start = Some r⟩ by blast
  qed
  then show ?thesis by blast
qed

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\(\text{(cases } \text{fst (hd } \text{xs}) < \text{fst (hd } \text{ys}))\)
case True

moreover have \(\forall r. \text{start = Some } r \implies r \in \text{set (interp-ins ops)}\)
using assms(9) by blast

ultimately have \(\forall x \in \text{set (map } \text{fst } \text{xs}). \forall y \in \text{set (map } \text{fst } \text{ys}). \text{list-order ops } y \text{ } x\)
using assms no-interleaving-ordered by blast
then show \(\text{thesis by blast}\)

next

case False

hence \(\text{fst (hd } \text{ys}) < \text{fst (hd } \text{xs})\)
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 \(\forall r. \text{start = Some } r \implies r \in \text{set (interp-ins ops)}\)
using assms(9) by blast

ultimately have \(\forall x \in \text{set (map } \text{fst } \text{ys}). \forall y \in \text{set (map } \text{fst } \text{xs}). \text{list-order ops } y \text{ } x\)
using assms no-interleaving-ordered by blast
then show \(\text{thesis by blast}\)

qed

For completeness, we also prove what happens if there are two insertion sequences, \(\text{xs}\) and \(\text{ys}\), but their initial position \(\text{start}\) does not exist. In that case, none of the insertions in \(\text{xs}\) or \(\text{ys}\) 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 \(\forall x \in \text{set (map } \text{fst } \text{xs}) \cup \text{set (map } \text{fst } \text{ys}). x \notin \text{set (interp-ins ops)}\)
using assms insert-seq-no-start by (metis UnE)

end

5 The Replicated Growable Array (RGA)

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

theory RGA
  imports Insert-Spec
begin

fun insert-body :: 'oid::{linorder} list \Rightarrow 'oid \Rightarrow 'oid list
where
insert-body [] e = [e]|

76
insert-body \((x \neq xs) e =\)
\[(\text{if } x < e \text{ then } e \neq x \neq xs \\text{ else } x \neq \text{ insert-body } xs e)\]

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

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

5.1 Commutativity of insert-rga

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

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

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

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

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

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

lemma insert-rga-Some-commutes:
assumes $i_1 \in \text{set } xs$ and $i_2 \in \text{set } xs$
and $e_1 \neq i_2$ and $e_2 \neq i_1$
shows $\text{insert-rga} (\text{insert-rga} \ (xs \ e_1, \text{Some } i_1)) \ (e_2, \text{Some } i_2) =
\text{insert-rga} (\text{insert-rga} \ (xs \ e_2, \text{Some } i_2)) \ (e_1, \text{Some } i_1)$
using assms proof (induction $xs$, simp)
case (Cons $a$ $xs$)
then show $?\text{case}$
by (cases $a = i_1$; cases $a = i_2$;
auto simp add: insert-body-commutes insert-rga-insert-body-commute)
qed

lemma insert-rga-commutes:
assumes $i_2 \neq \text{Some } e_1$ and $i_1 \neq \text{Some } e_2$
shows $\text{insert-rga} (\text{insert-rga} \ (xs \ e_1, \text{Some } i_1)) \ (e_2, \text{Some } i_2) =
\text{insert-rga} (\text{insert-rga} \ (xs \ e_2, \text{Some } i_2)) \ (e_1, \text{Some } i_1)$
proof(cases $i_1$
  case None
  then show $?\text{thesis}$
  using assms(1) insert-rga-None-commutes by (cases $i_2$, fastforce, blast)
next
  case some-r1: ($\text{Some } r_1$
  then show $?\text{thesis}$
  proof(cases $i_2$
    case None
    then show $?\text{thesis}$
    using assms(2) insert-rga-None-commutes by fastforce
next
    case some-r2: ($\text{Some } r_2$
    then show $?\text{thesis}$
    proof(cases $r_1 \in \text{set } xs \land r_2 \in \text{set } xs$
      case True
      then show $?\text{thesis}$
      using assms some-r1 some-r2 by (simp add: insert-rga-Some-commutes)
next
      case False
      then show $?\text{thesis}$
      using assms some-r1 some-r2
by (metis insert-iff insert-rga-nonexistent insert-rga-set-ins)
qed
qed

lemma insert-body-split:
shows $\exists p. s. \ xs = p @ s \land \text{insert-body } xs \ e = p @ e \# s$
proof(induction $xs$, force)
case (Cons $a$ $xs$)
then obtain $p$ $s$ where $\text{IH}: \ xs = p @ s \land \text{insert-body } xs \ e = p @ e \# s$
by blast
then show $\exists p. s. a \# xs = p @ s \land \text{insert-body } (a \# xs) \ e = p @ e \# s$

78
proof (cases \( a < e \))
  case True
  then have \( a \# xs = [] @ (a \# p @ s) \land \text{insert-body} (a \# xs) e = [] @ e \# (a \# p @ s) \)
    by (simp add: IH)
  then show \(?thesis\) by blast
next
case False
  then have \( a \# xs = (a \# p) @ s \land \text{insert-body} (a \# xs) e = (a \# p) @ e \# s \)
    using IH by auto
  then show \(?thesis\) by blast
qed
qed

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

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

lemma insert-rga-preserves-order:
assumes \( \text{i = None \lor (\exists i'. \ i = Some i' \land i' \in \text{set xs}} \)
  and \( \text{distinct \( \text{xs} \) \land } \exists \text{pre suf. \( \text{xs = pre @ suf} \land \text{insert-rga \( \text{xs} \) (e, i) = pre @ e \# suf} \)
proof (cases i)
  case None
then show $\exists$ pre suf. $xs = pre @ suf \land insert-rga x$s (e, i) = pre @ e # suf
  using insert-body-split by auto
next
case (Some r)
moreover from this obtain as bs where $xs = as @ r # bs \land (\forall x \in set as. x \neq r)$
  using assms(1) split-list-first by fastforce
moreover have $\exists$ cs ds. $bs = cs @ ds \land insert-body bs e = cs @ e # ds$
  by (simp add: insert-body-split)
then obtain cs ds where $bs = cs @ ds \land insert-body bs e = cs @ e # ds$
  by blast
ultimately have $xs = (as @ r # cs) @ ds \land insert-rga x$s (e, i) = (as @ r # cs) @ e # ds$
  using insert-rga-after-ref by fastforce
then show $\exists$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 $\land. r2 = Some r \Rightarrow r \neq i1$
  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 ∈ fst ' set pre
proof −
  from assms have crdt-ops (pre @ (oid, Some ref) # suf) set-option
    by (simp add: rga-ops-def)
  moreover have set-option (Some ref) = {ref}
    by simp
  ultimately show ref ∈ fst ' set pre
    using crdt-ops-ref-exists by fastforce
qed
5.3 Lemmas about the interp-rga function

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

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

81
by (metis rga-ops-def snoc.prems)

moreover have ∃ pre suf. interp-rga xs = pre@xsuf ∧
  insert-rga (interp-rga xs) (xi, xr) = pre @ xi # suf

proof −

have ∀ r. r ∈ set-option xr → r ∈ set (map fst xs)
  using crdt-ops-ref-exists rga-ops-def snoc.prems x-pair by fastforce

hence xr = None ∨ (∃ r. xr = Some r ∧ r ∈ set (map fst xs))
  using option.set-set by blast

hence interp-rga xs = interp-ins ys
  using interp-rga-ids rga-ops-rem-last snoc.prems by blast

thus ?thesis
  using IH insert-rga-preserves-order by blast

qed

ultimately show distinct (interp-rga (xs @ [x]))
  by (metis Un-iff disjoint-insert (1) distinct.prems(2) distinct-append
      interp-rga-tail-unfold list.prems(15) set-append)

qed

5.4 Proof that RGA satisfies the list specification

lemma final-insert:
  assumes set (xs @ [x]) = set (ys @ [x])
      and rga-ops (xs @ [x])
      and insert-ops (ys @ [x])
      and interp-rga xs = interp-ins ys
  shows interp-rga (xs @ [x]) = interp-ins (ys @ [x])

proof −

obtain oid ref where x-pair: x = (oid, ref) by force

have distinct (xs @ [x]) and distinct (ys @ [x])
  using assms crdt-ops-distinct spec-ops-distinct rga-ops-def insert-ops-def by blast+

then have set xs = set ys
  using assms(1) by force

have oid-greatest: ∃ i. i ∈ set (interp-rga xs) → i < oid

proof −

have ∃ i. i ∈ set (map fst ys) → i < oid
  using assms(3) by (simp add: spec-ops-id-inc x-pair insert-ops-def)

hence ∃ i. i ∈ set (map fst xs) → i < oid
  using (set xs = set ys) by auto

thus ∃ i. i ∈ set (interp-rga xs) → i < oid
  using assms(2) interp-rga-ids rga-ops-rem-last by blast

qed

thus interp-rga (xs @ [x]) = interp-ins (ys @ [x])

proof(cases ref)

  case None

  moreover from this have insert-rga (interp-rga xs) (oid, ref) = oid #
      interp-rga xs

  using oid-greatest hd-in-set insert-body.elims insert-body.prems(1)
      interp-rga.prems(1) list.sel(1) by metis

82
ultimately show \( \text{interp-rga} (xs @ [x]) = \text{interp-ins} (ys @ [x]) \)
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 assms\((2)\) Some by (simp add: rga-ops-ref-exists \(x\)-pair)
  hence \( r \in \text{set} (\text{interp-rga} \ xs) \)
    using assms\((2)\) interp-rga-ids rga-ops-rem-last by blast
thus \( \exists \)thesis by (simp add: split-list)
qed

from this obtain \( as \ bs \) where \( as-bs: \ \text{interp-rga} \ xs = as @ r \# bs \) by force
hence distinct \( (as \ @ r \# bs) \)
  by (metis assms\((2)\) interp-rga-distinct rga-ops-rem-last)
hence \( \text{insert-rga} (as @ r # bs) = \text{interp-rga} (xs @ [x]) = \text{interp-ins} (ys @ [x]) \)
by (metis assms\((4)\) Some as-bs interp-ins-tail-unfold interp-rga-tail-unfold \(x\)-pair)
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 \( \bigwedge r. \ \text{ref} = \text{Some r} \implies r \notin \text{fst} \cdot \text{set} \ suf \)
shows \( \text{interp-rga} (pre @ (oid, ref) # suf) = \text{interp-rga} (pre @ suf @ [(oid, ref)]) \)
using assms proof(induction suf rule: List.rev-induct)
case Nil
then show \( ?\)case by simp
next
case (snoc \( x \) \( xs \))

have \( \text{ref-not-x}: \ \bigwedge r. \ r \neq \text{fst} x \) using snoc.prems\((3)\) 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 snoc.prems\((1)\) by auto
  moreover have \( \bigwedge r. \ r \neq \text{fst} x \)
    by (simp add: ref-not-x)
  ultimately have \( \text{rga-ops} ((pre @ xs) @ [(oid, ref)]) \)
    using rga-ops-rem-penultimate
    by (metis (no-types, lifting) Cons-eq-append-conv prod-collapse)
  thus \( ?\)thesis using snoc by force
qed

obtain \( xi \) \( xr \) where \( x\)-pair: \( x = (xi, xr) \) by force

83
have interp-rga (pre @ (oid, ref) # xs @ [(xi, xr)]) =
insert-rga (interp-rga (pre @ xs @ [(oid, ref)])) (xi, xr)
using IH interp-rga-tail-unfold by (metis append.assoc append-Cons)
moreover have ... = insert-rga (insert-rga (interp-rga (pre @ xs)) (oid, ref))
(xi, xr)
using interp-rga-tail-unfold by (metis append-assoc)
moreover have ...
= insert-rga (insert-rga (interp-rga (pre @ xs)) (xi, xr)) (oid, ref)
proof
have \( \forall xrr. \, xr = \text{Some } xrr \Rightarrow xrr \neq \text{oid} \)
using x-pair snoc.prems(2) by auto
thus ?thesis
using insert-rga-commutes ref-not-x by (metis conv x-pair)
qed
moreover have ...
= interp-rga (pre @ xs @ [x] @ [(oid, ref)])
by (metis append-assoc interp-rga-tail-unfold x-pair)
ultimately show interp-rga (pre @ (oid, ref) # xs @ [x]) =
interp-rga (pre @ (xs @ [x]) @ [(oid, ref)])
by (simp add: x-pair)
qed

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

84
ultimately show ?thesis
using insert-ops-rem-last ys-split snoc by metis
qed
have valid-rga: rga-ops (pre @ suf @ [x])
proof –
  have crdt-ops (pre @ suf @ [x]) set-option
    using snoc.prems ys-split
    by (simp add: crdt-ops-reorder-spec insert-ops-def rga-ops-def)
  thus rga-ops (pre @ suf @ [x])
    by (simp add: rga-ops-def)
qed
have interp-ins (xs @ [x]) = interp-rga (pre @ suf @ [x])
proof –
  have set (xs @ [x]) = set (pre @ suf @ [x])
    using snoc.prems(1) ys-split by auto
  thus ?thesis
    using IH snoc.prems(2) valid-rga final-insert append-assoc by metis
qed
moreover have ...
ultimately show interp-rga (pre @ [x] @ suf)
by (simp add: ys-split)
qed
lemma insert-ops-exist:
assumes rga-ops xs
shows \( \exists \) ys, set xs = set ys \& insert-ops ys
using assms by (simp add: crdt-ops-spec-ops-exist insert-ops-def rga-ops-def)
theorem rga-meets-spec:
assumes rga-ops xs
shows \( \exists \) ys, set ys = set xs \& insert-ops ys \& interp-ins ys = interp-rga xs
using assms rga-spec-equal insert-ops-exist by metis
end

References


