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 \textit{A}\textsubscript{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
assumes unique-oid: \((oid, op1) \in opset \Rightarrow (oid, op2) \in opset \Rightarrow op1 = op2\)
and ref-older: \((oid, oper) \in opset \Rightarrow ref \in deps oper \Rightarrow ref < oid\)
and finite-opset: finite opset

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

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

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

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

**1.2 Helper lemmas about lists**

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

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

lemma distinct-fst-append:
assumes \(x \in \text{set } (\text{map } \text{fst } xs)\)
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 # ya
thus xa = xb \land ya = yb
by auto

next
fix a xs xa xb x
assume IH: \(\forall xa xb x. \text{distinct } xs \implies xs = xa @ x # ya \implies xs = xb @ x # yb\)
\(\implies xa = xb \land ya = yb\)
and distinct \((a @ x)\) and \(a @ x = xa @ x # ya\) and \(a @ x = xb @ x # yb\)
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 \textit{append-set-rem-last}:\n\begin{itemize}
  \item \textbf{assumes} \texttt{set (xs @ [x]) = set (ys @ [x] @ zs)}
  \item \textbf{and} \texttt{distinct (xs @ [x]) \textbf{and} distinct (ys @ [x] @ zs)}
\end{itemize}
\textbf{shows} \texttt{set xs = set (ys @ zs)}
\textbf{proof –}
\begin{itemize}
  \item \texttt{have distinct xs}
  \begin{itemize}
    \item \textbf{using} \textbf{assms} \textbf{distinct-append} \textbf{by} \textbf{blast}
  \end{itemize}
  \item \texttt{moreover from this have set xs = set (xs @ [x]) – {x}}
  \begin{itemize}
    \item \textbf{by} \textbf{(meson} \textbf{assms} \textbf{distinct-set-remove-last})
  \end{itemize}
  \item \texttt{moreover have distinct (ys @ zs)}
  \begin{itemize}
    \item \textbf{using} \textbf{assms} \textbf{distinct-rem-mid} \textbf{by} \textbf{simp}
  \end{itemize}
  \item \texttt{ultimately show set xs = set (ys @ zs)}
  \begin{itemize}
    \item \textbf{using} \textbf{assms} \textbf{distinct-set-remove-mid} \textbf{by} \textbf{metis}
  \end{itemize}
\end{itemize}
\textbf{qed}

Lemma \textit{distinct-map-fst-remove1}:\n\begin{itemize}
  \item \textbf{assumes} \texttt{distinct (map fst xs)}
  \item \textbf{shows} \texttt{distinct (map fst (remove1 x xs))}
  \item \textbf{using} \textbf{assms} \textbf{proof(induction xs)}
\end{itemize}
\textbf{case Nil}
\item \texttt{then show} \texttt{distinct (map fst (remove1 x []))}
  \begin{itemize}
    \item \textbf{by} \textbf{simp}
  \end{itemize}
\textbf{next}
\item \texttt{case (Cons a xs)}
\item \texttt{hence IH: distinct (map fst (remove1 x xs))}
  \begin{itemize}
    \item \textbf{by} \textbf{simp}
  \end{itemize}
\item \texttt{then show} \texttt{distinct (map fst (remove1 x (a # xs)))}
  \begin{itemize}
    \item \textbf{proof(cases} \texttt{a = x)}
    \begin{itemize}
      \item \texttt{case True}
      \begin{itemize}
        \item \texttt{then show} \texttt{?thesis}
        \begin{itemize}
          \item \texttt{using} \texttt{Cons.prems} \textbf{by} \textbf{auto}
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \end{itemize}
\item \texttt{next}
\item \texttt{case False}
\item \texttt{moreover have} \texttt{fst a \notin fst ‘ set (remove1 x xs)}
  \begin{itemize}
    \item \texttt{by} \texttt{(metis (no-types, lifting) Cons.prems distinct.simps(2) image-iff list.simps(9) notin-set-remove1 set-map)}
  \end{itemize}
\item \texttt{ultimately show} \texttt{?thesis}
  \begin{itemize}
    \item \texttt{using} \texttt{IH} \textbf{by} \textbf{auto}
  \end{itemize}
\end{itemize}
\textbf{qed}

1.3 The \textit{spec-ops} predicate

The \textit{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 \textit{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}} :: (\textit{oid}::{linorder} × \textit{\textquotesingle oper}) list \Rightarrow (\textit{\textquotesingle oper} \Rightarrow \textit{\textquotesingle oid} set) \Rightarrow \textit{bool}

where

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

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

\textit{shows \textit{spec-ops [\textit{[]} \textit{deps}}

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

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

\textit{assumes \textit{spec-ops \textit{ops} \textit{deps}}

\textit{shows \textit{distinct \textit{ops}}

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

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

\textit{assumes \textit{spec-ops \textit{ops} \textit{deps}}

\textit{shows \textit{distinct (map \textit{fst \textit{ops}})}

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

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

\textit{assumes \textit{spec-ops \textit{ops} \textit{deps}}

\textit{shows \textit{sorted (map \textit{fst \textit{ops}})}

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

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

\textit{assumes \textit{spec-ops \textit{(\textit{x \# \textit{xs}}) \textit{deps}}

\textit{shows \textit{spec-ops \textit{xs \textit{deps}}

\textbf{proof –

\textit{have \textit{sorted (map \textit{fst (\textit{x \# \textit{xs}})) and distinct (map \textit{fst (\textit{x \# \textit{xs}}))

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

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

\textit{by simp

\textit{moreover have \textit{\forall \textit{oid oper ref} \cdot (\textit{oid}, \textit{oper}) \in \textit{set \textit{xs} \land \textit{ref} \in \textit{deps oper} \rightarrow \textit{ref} < \textit{oid

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

\textit{ultimately show \textit{spec-ops \textit{xs \textit{deps}

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

\textit{qed}}

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

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

\textit{shows \textit{spec-ops \textit{xs \textit{deps}}

\textbf{proof –

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

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

\textit{moreover from this have \textit{sorted (map \textit{fst \textit{xs}) and distinct \textit{xs

6
by (auto simp add: sorted-append distinct-butlast distinct-map)
moreover have ∀ oid oper ref. (oid, oper) ∈ set xs ∧ ref ∈ deps oper → ref < oid
  by (metis assms butlast-snoc in-set-butlastD spec-ops-def)
ultimately show spec-ops xs deps
  by (simp add: spec-ops-def)
qed

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

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

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

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

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

lemma spec-ops-add-any:
  assumes spec-ops (xs @ ys) deps
  and ∀ i ∈ set (map fst xs). i < oid
  and ∀ i ∈ set (map fst ys). oid < i
  and ∀ ref ∈ deps oper. ref < oid
  shows spec-ops (xs @ [(oid, oper)] @ ys) deps
  using assms proof (induction ys rule: rev-induct)
  case Nil
  then show spec-ops (xs @ [(oid, oper)] @ []) deps
    by (simp add: spec-ops-add-last)
next
case (snoc y ys)
  have IH: spec-ops (xs @ [(oid, oper)] @ ys) deps
    proof
      from snoc have spec-ops (xs @ ys) deps
        by (metis append-assoc spec-ops-rem-last)
      thus spec-ops (xs @ [(oid, oper)] @ ys) deps
        using assms(2) snoc by auto
    qed
  obtain yi yo where y-pair: y = (yi, yo)
    by force
  have oid-yi: oid < yi
    by (simp add: snoc.prems(3) y-pair)
  have yi-biggest: ∀ i ∈ set (map fst (xs @ [(oid, oper)] @ ys)). i < yi
    proof
      have ∀ i ∈ set (map fst xs). i < yi
        using oid-yi assms(2) less-trans by blast
      moreover have ∀ i ∈ set (map fst ys). i < yi
        by (metis UnCI append-assoc map-append set-append snoc.prems(1) spec-ops-id-inc y-pair)
      ultimately show ?thesis
        using oid-yi by auto
    qed
  have sorted (map fst (xs @ [(oid, oper)] @ ys @ [y]))
    proof
from IH have sorted (map fst (xs @ [(oid, oper)] @ ys))
  using spec-ops-def by blast
hence sorted (map fst (xs @ [(oid, oper)] @ ys) @ [yi])
  using yi-biggest
by (simp add: sorted-append dual-order.strict-implies-order)
thus sorted (map fst (xs @ [(oid, oper)] @ ys @ [yi]))
  by (simp add: y-pair)
qed
moreover have distinct (map fst (xs @ [(oid, oper)] @ ys @ [yi]))
proof —
  have distinct (map fst (xs @ [(oid, oper)] @ ys) @ [yi])
  using IH yi-biggest spec-ops-def
  by (metis distinct.list1_rotate order-less-irrefl rotate1.simps(2))
thus distinct (map fst (xs @ [(oid, oper)] @ ys @ [yi]))
  by (simp add: y-pair)
qed
moreover have \( \forall i \text{ opr } r. (i, \text{ opr}) \in \text{ set } (xs @ [(oid, oper)] @ ys @ [yi]) \land r \in \text{ deps opr} \implies r < i \)
proof —
  have \( \forall i \text{ opr } r. (i, \text{ opr}) \in \text{ set } (xs @ [(oid, oper)] @ ys) \land r \in \text{ deps opr} \implies r < i \)
  by (meson IH spec-ops-def)
moreover have \( \forall \text{ ref}. \text{ ref} \in \text{ deps yo} \implies \text{ ref} < yi \)
  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 @ [yi]) 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 \( \exists \text{ pre suf}. xs = \text{ pre} @ \text{ suf} \land \)
  \( \forall i \in \text{ set } (\text{ map fst pre}). i < \text{ oid} \land \)
  \( \forall i \in \text{ set } (\text{ map fst suf}). \text{ oid} < i \)
using assms proof(induction xs rule: rev-induct)
case Nil
then show \(?case by force \)
next
case (snoc x xs)
obtain xi xr where y-pair: \( x = (xi, xr) \)
  by force
obtain pre suf where IH: \( xs = \text{ pre} @ \text{ suf} \land \)
  \( \forall a \in \text{ set } (\text{ map fst pre}). a < \text{ oid} \land \)
  \( \forall a \in \text{ set } (\text{ map fst suf}). \text{ oid} < a \)
by (metis UnCI map-append set-append snoc spec-ops-rem-last)
then show \( \forall \alpha \in \text{set} (\text{map} \text{fst} (\text{pre} \# \text{suf})). x < \alpha \)

- show \( \forall x \in \text{set} \text{fst} \text{prems}(1) \) \( x < \alpha \) by \text{metis}

- hence \( \forall x \in \text{set} \text{prems}(1) \) \( x < \alpha \) by \text{simp}

- hence \( \forall x \in \text{set} \text{prems}(1) \) \( x < \alpha \) by \text{auto}

- hence \( \forall x \in \text{set} \text{prems}(1) \) \( x < \alpha \) by \text{auto}

- hence \( \forall x \in \text{set} \text{prems}(1) \) \( x < \alpha \) by \text{auto}

- hence \( \forall x \in \text{set} \text{prems}(1) \) \( x < \alpha \) by \text{auto}

- hence \( \forall x \in \text{set} \text{prems}(1) \) \( x < \alpha \) by \text{auto}

then show \( \forall \beta \in \text{set} \text{prems}(1) \) \( \beta < \alpha \) by \text{auto}

qed

qed

\textbf{lemma} \text{spec-ops-exists-base}:

\textbf{assumes} \text{finite ops}

\textbf{and} \( \forall \alpha, \beta \in \text{ops} \Rightarrow \alpha = \beta \)

\textbf{and} \( \forall \alpha, \beta \in \text{ops} \Rightarrow \alpha = \beta \)

\textbf{shows} \( \exists \text{op-list}. \text{set} \text{op-list} = \text{ops} \land \text{spec-ops} \text{op-list} \text{deps} \)

\textbf{using} \text{assms proof(induct ops rule: Finite-Set.finite-induct-select)}

\textbf{case} \text{empty}

then show \( \exists \text{op-list}. \text{set} \text{op-list} = \emptyset \land \text{spec-ops} \text{op-list} \text{deps} \)

by \( \text{simp add: spec-ops-empty} \)

\textbf{next}

\textbf{case} \text{select subset}

from \textbf{this} obtain \text{op-list} where \text{set} \text{op-list} = \text{subset} \text{and} \text{spec-ops} \text{op-list} \text{deps}

using \text{assms by blast}

moreover obtain \( \exists \text{oid oper ref}. \text{oid oper ref} \in \text{ops} \Rightarrow \text{ref \in deps oper ref} \Rightarrow \text{ref < \alpha} \)

using \text{select.hyps(1) by auto}

moreover from \textbf{this} have \( \exists \text{oid oper ref}. \text{oid oper ref} \in \text{ops} \Rightarrow \text{ref \in deps oper ref} \Rightarrow \text{ref < \alpha} \)

using \text{assms(2) by auto}

hence \( \exists \text{oid oper ref}. \text{oid oper ref} \in \text{ops} \Rightarrow \text{ref \in deps oper ref} \Rightarrow \text{ref < \alpha} \)

by \( \text{metis (no-types, lifting) DiffD2 select image-iff prod.collapse psubsetD select.hyps(1)} \)

from \textbf{this} obtain \text{pre suf}

where \text{op-list} = \text{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 set $(\text{pre} \ominus \{(\text{oid}, \text{oper})\} \ominus \text{suf}) = \text{insert} (\text{oid}, \text{oper}) \ominus \text{suf}$
using calculation by auto

moreover have spec-ops $(\text{pre} \ominus \{(\text{oid}, \text{oper})\} \ominus \text{suf}) \ominus \text{deps}$
using calculation spec-ops-add-any assms (3) by (metis DiffD1)

ultimately show ?case by blast

qed

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

\textbf{lemma spec-ops-exists:}
assumes opset ops deps
shows $\exists \text{op-list}. \text{set op-list} = \text{ops} \land \text{spec-ops op-list deps}$
proof
  
  have finite ops
  using assms opset.finite-opset by force

  moreover have $\forall \text{oid op1 op2}. (\text{oid}, \text{op1}) \in \text{ops} \Rightarrow (\text{oid}, \text{op2}) \in \text{ops} \Rightarrow \text{op1} = \text{op2}$
  using assms opset.unique-oid by force

  moreover have $\forall \text{oid oper ref}. (\text{oid}, \text{oper}) \in \text{ops} \Rightarrow \text{ref} \in \text{deps oper} \Rightarrow \text{ref} < \text{oid}$
  using assms opset.ref-older by force

  ultimately show $\exists \text{op-list}. \text{set op-list} = \text{ops} \land \text{spec-ops op-list deps}$
  by (simp add: spec-ops-exists-base)

qed

\textbf{lemma spec-ops-oid-unique:}
assumes spec-ops op-list deps
  and $(\text{oid}, \text{op1}) \in \text{set op-list}$
  and $(\text{oid}, \text{op2}) \in \text{set op-list}$
shows $\text{op1} = \text{op2}$
using assms proof(induction op-list, simp)

\textbf{case (Cons x op-list)}
have distinct $(\text{map} \text{fst} (x \# \text{op-list}))$
  using Cons.prems(1) spec-ops-def by blast

\textbf{hence notin:} $\text{fst} x \notin \text{set} (\text{map} \text{fst} \text{op-list})$
  by simp

\textbf{then show} $\text{op1} = \text{op2}$
\textbf{proof(cases }$\text{fst} x = \text{oid}$
  \textbf{case True}
  \textbf{then show} $\text{op1} = \text{op2}$
  using Cons.prems notin by (metis Pair-inject in-set-zipE set-ConsD zip-map-fst-snd)

\textbf{next}
  \textbf{case False}
  \textbf{then have} $(\text{oid}, \text{op1}) \in \text{set op-list} \land (\text{oid}, \text{op2}) \in \text{set op-list}$
  using Cons.prems by auto

  \textbf{then show} $\text{op1} = \text{op2}$
  using Cons.IH Cons.prems(1) spec-ops-rem-cons by blast
Conversely, for any given \textit{spec-ops} list, the set of pairs in the list is an OpSet:

\begin{lemma}
\textit{spec-ops-is-opset}:
\begin{itemize}
\item \textit{assumes} \textit{spec-ops op-list deps}
\item \textit{shows} \textit{opset (set op-list) deps}
\end{itemize}
\begin{proof}
\item \textit{have} \((\forall \text{oid op1 op2}. (\text{oid}, \text{op1}) \in \text{set op-list} \implies (\text{oid}, \text{op2}) \in \text{set op-list} \implies \text{op1} = \text{op2})\)
\item \textit{using} \textit{assms spec-ops-oid-unique} \textit{by fastforce}
\item \textit{moreover have} \((\forall \text{oid oper ref}. (\text{oid}, \text{oper}) \in \text{set op-list} \implies \text{ref} \in \text{deps oper} \implies \text{ref} < \text{oid})\)
\item \textit{by} \((\text{meson assms spec-ops-ref-less})\)
\item \textit{moreover have} \textit{finite (set op-list)}
\item \textit{by} \textit{simp}
\item \textit{ultimately show} \textit{opset (set op-list) deps}
\item \textit{by} \((\text{simp add: opset-def})\)
\end{proof}
\end{lemma}

1.4 The \textit{crdt-ops} predicate

Like \textit{spec-ops}, the \textit{crdt-ops} predicate describes the linearisation of an OpSet into a list. Like \textit{spec-ops}, it requires IDs to be unique. However, its other properties are different: \textit{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 \textit{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 \textit{crdt-ops} requires that concurrent operations commute. For any \textit{crdt-ops} operation sequence, there is a permutation that satisfies the \textit{spec-ops} predicate. Thus, to check whether a CRDT satisfies its sequential specification, we can prove that interpreting any \textit{crdt-ops} operation sequence with the commutative operation interpretation results in the same end result as interpreting the \textit{spec-ops} permutation of that operation sequence with the sequential operation interpretation.

\begin{inductive}
\textit{crdt-ops :: ('oid::linorder) \times 'oper} list \Rightarrow ('oper \Rightarrow 'oid set) \Rightarrow bool
\end{inductive}
\begin{where}
\textit{crdt-ops [] deps }
\textit{crdt-ops xs deps;}
\textit{oid \notin set (map fst xs);}
\textit{\forall ref \in deps oper. ref \in set (map fst xs) \land ref < oid}
\textit{\implies crdt-ops (xs @ [(oid, oper)]) deps}
\end{where}
inductive-cases crdt-ops-last: crdt-ops (xs @ [x]) deps

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

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

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

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

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

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

lemma crdt-ops-unique-last:
assumes crdt-ops (xs @ [(oid, oper)]) deps
shows oid \notin set (map fst xs)
using assms crdt-ops.cases by blast
lemma crdt-ops-unique-mid:
  assumes crdt-ops (xs @ [(oid, oper)] @ ys) deps
  shows oid \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 = (yi, yr)
    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)
  have (xs @ (oid, oper) # ys) @ [(yi, yr)] = xs @ (oid, oper) # ys @ [(yi, yr)]
    by simp
  hence yi \notin set (map fst (xs @ (oid, oper) # ys))
    using crdt-ops-unique-last by (metis append-Cons append-self-conv2 snoc.prems y-pair)
  thus oid \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 @ (oid, oper) # 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 @ [(oid, oper)] @ ys) deps
  shows \\land ref. ref \in deps oper \implies ref \notin set ys
  proof
    from assms(1) have \\land ref. ref \in deps oper \implies ref \in set (map fst xs)
      by (simp add: crdt-ops-ref-exists)
    moreover have distinct (map fst (xs @ [(oid, oper)] @ ys))
      using assms crdt-ops-distinct-fst by blast
    ultimately have \\land ref. ref \in deps oper \implies ref \notin set (map fst ([(oid, oper)] @ ys))
      using distinct-fst-append by metis
    thus \\land ref. ref \in deps oper \implies ref \notin set ys
      by simp
  qed
lemma crdt-ops-reorder:
assumes crdt-ops \((xs \ @ \ [(oid, oper)] \ @ \ ys)\) deps
and \(\bigwedge op2 \ r. op2 \in \text{snd} \ i \set ys \Longrightarrow r \in \text{deps} \ op2 \Longrightarrow r \neq oid\)
shows crdt-ops \((xs \ @ \ ys \ @ [(oid, oper)])\) deps
using assms proof (induction ys rule: rev-induct)
case Nil
then show crdt-ops \((xs \ @ \ [] \ @ [(oid, oper)])\) deps
using crdt-ops-rem-last by auto
next
case \((snoc \ y \ ys)\)
then obtain \(yi \ yo\) where y-pair: \(y = (yi, yo)\)
by fastforce
have IH: crdt-ops \((xs \ @ \ ys \ @ [(oid, oper)])\) deps
proof –
have crdt-ops \((xs \ @ [(oid, oper)] \ @ \ ys)\) deps
by (metis snoc append assoc crdt-ops-rem-last)
thus crdt-ops \((xs \ @ \ ys \ @ [(oid, oper)])\) deps
using snoc.IH snoc.prems(2) by auto
qed
have crdt-ops \((xs \ @ \ ys \ @ [y])\) deps
proof –
have \(yi \notin \text{fst} \ i \set (xs \ @ [(oid, oper)] \ @ \ ys)\)
by (metis y-pair append-assoc crdt-ops-unique-last set-map snoc.prems(1))
hence \(yi \notin \text{fst} \ i \set (xs \ @ \ ys)\)
by auto
moreover have \(\bigwedge r. r \in \text{deps} \ yo \Longrightarrow r \in \text{fst} \ i \set (xs \ @ \ ys) \land r < yi\)
proof –
have \(\bigwedge r. r \in \text{deps} \ yo \Longrightarrow r \neq oid\)
using snoc.prems(2) y-pair by fastforce
moreover have \(\bigwedge r. r \in \text{deps} \ yo \Longrightarrow r \in \text{fst} \ i \set (xs \ @ [(oid, oper)] \ @ \ ys)\)
by (metis y-pair append-assoc snoc.prems(1) crdt-ops-ref-exists)
moreover have \(\bigwedge r. r \in \text{deps} \ yo \Longrightarrow r < yi\)
using crdt-ops-ref-less snoc.prems(1) y-pair by fastforce
ultimately show \(\bigwedge r. r \in \text{deps} \ yo \Longrightarrow r \in \text{fst} \ i \set (xs \ @ \ ys) \land r < yi\)
by simp
qed
moreover from IH have crdt-ops \((xs \ @ \ ys)\) deps
using crdt-ops-rem-last by force
ultimately show crdt-ops \((xs \ @ \ ys \ @ [y])\) deps
using y-pair crdt-ops-intro by (metis append assoc)
qed
moreover have \(oid \notin \text{fst} \ i \set (xs \ @ \ ys \ @ [y])\)
using crdt-ops-unique-mid by (metis (no-types, lifting) UnE image-Un image-set append snoc.prems(1))
morerover have \(\bigwedge r. r \in \text{deps} \ oper \Longrightarrow r \in \text{fst} \ i \set (xs \ @ \ ys \ @ [y])\)
using crdt-ops-ref-exists
by (metis UnCI append-Cons image-Un set-append snoc.prems(1))
morerover have \(\bigwedge r. r \in \text{deps} \ oper \Longrightarrow r < oid\)
using IH crdt-ops-ref-less by fastforce
ultimately show crdt-ops \((xs \oplus (ys \ominus [y]) \oplus [(oid, oper)])\) \(\text{deps}\)
using crdt-ops-intro by (metis append-assoc)
qed

lemma crdt-ops-rem-middle:
assumes crdt-ops \((xs \oplus [(oid, ref)] \oplus ys)\) \(\text{deps}\)
and \(\forall op2. op2 \in \text{snd } \text{set } ys \rightarrow r \in \text{deps } op2 \rightarrow r \neq oid\)
shows crdt-ops \((xs \oplus ys)\) \(\text{deps}\)
using assms crdt-ops-rem-last crdt-ops-reorder append-assoc by metis

lemma crdt-ops-independent-suf:
assumes spec-ops \((xs \oplus [(oid, oper)])\) \(\text{deps}\)
and crdt-ops \((ys \oplus [(oid, oper)] \oplus zs)\) \(\text{deps}\)
and \(\text{set } (xs \oplus [(oid, oper)]) = \text{set } (ys \oplus [(oid, oper)] \oplus zs)\)
shows \(\forall op2. op2 \in \text{snd } \text{set } zs \rightarrow r \in \text{deps } op2 \rightarrow r \neq oid\)
proof
  have \(\forall op2. op2 \in \text{snd } \text{set } xs \rightarrow r \in \text{deps } op2 \rightarrow r < oid\)
  using spec-ops-id-inc by fastforce
moreover have \(\forall i2. op2 \in \text{set } xs \rightarrow r \in \text{deps } op2 \rightarrow r < i2\)
  using assms spec-ops-ref-less spec-ops-rem-last by fastforce
ultimately show \(\forall op2. op2 \in \text{snd } \text{set } zs \rightarrow r \in \text{deps } op2 \rightarrow r < oid\)
  by fastforce
qed
moreover have \(\text{set } zs \subseteq \text{set } xs\)
proof
  have distinct \((xs \oplus [(oid, oper)])\) and distinct \((ys \oplus [(oid, oper)] \oplus zs)\)
  using assms spec-ops-distinct crdt-ops-distinct by blast+
hence \(\text{set } xs = \text{set } (ys \oplus zs)\)
  by (meson append-set-rem-last assms(3))
then show \(\text{set } zs \subseteq \text{set } xs\)
  using append-subset(2) by simp
qed
ultimately show \(\forall op2. op2 \in \text{snd } \text{set } zs \rightarrow r \in \text{deps } op2 \rightarrow r \neq oid\)
  by fastforce
qed

lemma crdt-ops-reorder-spec:
assumes spec-ops \((xs \oplus [x])\) \(\text{deps}\)
and crdt-ops \((ys \oplus [x] \oplus zs)\) \(\text{deps}\)
and \(\text{set } (xs \oplus [x]) = \text{set } (ys \oplus [x] \oplus zs)\)
shows crdt-ops \((ys \oplus zs \oplus [x])\) \(\text{deps}\)
using assms proof
  obtain oid oper where \(x\)-pair: \(x = (oid, oper)\) by force
hence \(\forall op2. op2 \in \text{snd } \text{set } zs \rightarrow r \in \text{deps } op2 \rightarrow r \neq oid\)
  using assms crdt-ops-independent-suf by fastforce
thus crdt-ops \((ys \oplus zs \oplus [x])\) \(\text{deps}\)
using assms(2) crdt-ops-reorder x-pair by metis
qed

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

lemma crdt-ops-rem-penultimate:
  assumes crdt-ops \{xs @ [(i1, r1)] @ [(i2, r2)]\} deps
  and \(\forall r \in \text{deps} \quad r2 \implies r \neq i1\)
  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 \(\forall r \in \text{deps} \quad r2 \implies r \in \text{fst set} \{\text{map fst xs}\}\)
    using crdt-ops-ref-exists by metis
  hence \(\forall r \in \text{deps} \quad r2 \implies r \in \text{set} \{\text{map fst xs}\}\)
    using assms(2) by auto
  moreover have \(\forall r \in \text{deps} r2 \implies r < i2\)
    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 \(\exists ys. \quad \text{set} \{xs\} = \text{set} \{ys\} \land \text{spec-ops} \{ys\} \text{ deps}\)
  using assms proof(induction xs rule: List.rev-induct)
  case Nil
  then show \(\exists ys. \quad \text{set} [] = \text{set} \{ys\} \land \text{spec-ops} \{ys\} \text{ deps}\)
    by (simp add: spec-ops-empty)
next
  case (snoc x xs)
  hence IH: \(\exists ys. \quad \text{set} \{xs\} = \text{set} \{ys\} \land \text{spec-ops} \{ys\} \text{ deps}\)
    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 \( \exists \text{pre suf}. \, \text{ys} = \text{pre@\text{suf}} \land \)
\( \forall i \in \text{set (map fst pre)}. \, i < \text{oid} \land \)
\( \forall i \in \text{set (map fst suf)}. \, \text{oid} < i \)

proof

have \( \text{oid} \notin \text{set (map fst \text{xs})} \)
using calculation(3) crdt-ops-unique-last snoc.prems by force
hence \( \text{oid} \notin \text{set (map fst \text{ys})} \)
by (simp add: calculation(1))
thus \( \text{?thesis} \)
using spec-ops-split (spec-ops \text{ys} \text{deps}) by blast

qed

from this obtain \( \text{pre suf} \) where \( \text{ys} = \text{pre @ suf} \) and
\( \forall i \in \text{set (map fst pre)}. \, i < \text{oid} \land \)
\( \forall i \in \text{set (map fst suf)}. \, \text{oid} < i \) by force
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 \( \text{spec-ops (pre @ [(\text{oid}, \text{ref})] @ suf) \text{deps}} \)
proof

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

qed

ultimately show \( \exists \text{ys. set (xs @ [x])} = \text{set \text{ys} \land spec-ops \text{ys} \text{deps}} \)
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 \text{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} :: \(\text{'oid list} \Rightarrow (\text{'oid} \times \text{'oid option}) \Rightarrow \text{'oid list}\) where
fun insert-alt :: ('oid × 'oid option) set ⇒ ('oid × 'oid) ⇒ ('oid × 'oid option)
set where
insert-alt list-rel (oid, ref) = 
  (if ∃ n. (ref, n) ∈ list-rel
   then {(p, n) ∈ list-rel. p ≠ ref} ∪ {(ref, Some oid)} ∪
      {(i, n). i = oid ∧ (ref, n) ∈ list-rel}
   else list-rel)

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

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

2.1 The insert-ops predicate

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

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

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

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

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

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

lemma insert-ops-appendD:
  assumes insert-ops (xs @ ys)
shows \text{insert-ops} \; \text{xs}
\textbf{using} \; \text{assms by} \; (\text{induction} \; \text{ys rule: List.rev-induct},
\text{auto, \text{metis insert-ops-rem-last append-assoc})}

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

\textbf{lemma} \; \text{insert-ops-remove1:}
\textbf{assumes} \; \text{insert-ops} \; \text{xs}
\textbf{shows} \; \text{insert-ops} \; (\text{remove1} \; x \; \text{xs})
\textbf{using} \; \text{assms \text{insert-ops-def spec-ops-remove1 by blast}}

\textbf{lemma} \; \text{last-op-greatest:}
\textbf{assumes} \; \text{insert-ops} \; (\text{op-list} \; @ \; [(\text{oid}, \; \text{oper})])
\textbf{and} \; x \in \text{set} \; (\text{map \; \text{fst} \; \text{op-list}})
\textbf{shows} \; x < \text{oid}
\textbf{using} \; \text{assms \text{spec-ops-id-inc insert-ops-def by metis}}

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

\textbf{lemma} \; \text{insert-ops-memb-ref-older:}
\textbf{assumes} \; \text{insert-ops \; op-list}
\textbf{and} \; (\text{oid, Some \; ref}) \in \text{set \; op-list}
\textbf{shows} \; \text{ref} < \text{oid}
\textbf{using} \; \text{assms \text{insert-ops-ref-older split-list-first by fastforce}}

\textbf{2.2 Properties of the \text{insert-spec function}}

\textbf{lemma} \; \text{insert-spec-none [simp]:}
\textbf{shows} \; \text{set} \; (\text{insert-spec} \; \text{xs} \; (\text{oid, None})) = \text{set} \; \text{xs} \cup \{\text{oid}\}
\textbf{by} \; (\text{induction \; xs}, \text{auto simp add: insert-commute sup-commute})
lemma insert-spec-set [simp]:
  assumes ref ∈ set xs
  shows set (insert-spec xs (oid, Some ref)) = set xs ∪ {oid}
  using assms proof (induction xs)
  assume ref ∈ set []
  thus set (insert-spec [] (oid, Some ref)) = set [] ∪ {oid}
    by auto
next
  fix a xs
  assume ref ∈ set xs ⟹ set (insert-spec xs (oid, Some ref)) = set xs ∪ {oid}
    and ref ∈ set (a#xs)
  thus set (insert-spec (a#xs) (oid, Some ref)) = set (a#xs) ∪ {oid}
    by (cases a = ref, auto simp add: insert-commute sup-commute)
qed

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

lemma list-greater-non-memb:
  fixes oid :: 'oid :: linorder
  assumes ⟨∀x. x ∈ set xs ⟹ x < oid ⟩
    and oid ∈ set xs
  shows False
  using assms by blast

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

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

lemma insert-spec-distinct [intro]:
  fixes oid :: 'oid::{linorder}
  assumes distinct xs
    and $\forall x. x \in \text{set} \; \Rightarrow \; x < \text{oid}$
    and ref = Some r $\Rightarrow$ 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 $\Rightarrow$ ($\forall x. x \in \text{set} \; \Rightarrow \; x < \text{oid}$) $\Rightarrow$ distinct (insert-spec xs (oid, ref))
      and D: distinct (a#xs)
      and L: $\forall x. x \in \text{set} \; (a#xs) \; \Rightarrow \; x < \text{oid}$
    show distinct (insert-spec (a#xs) (oid, ref))
      proof(cases ref)
        assume ref = None
        thus distinct (insert-spec (a#xs) (oid, ref))
          using D L by auto
      next
        fix id
        assume S: ref = Some id
        {
          assume EQ: $a = \text{id}$
          hence id $\neq$ oid
            using D L by auto
          moreover have id $\notin$ set xs
            using D EQ by auto
          moreover have oid $\notin$ set xs
            using L by auto
          ultimately have id $\neq$ oid $\land$ id $\notin$ set xs $\land$ oid $\notin$ set xs $\land$ 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(2) insert-spec-none insert-spec-none)
          have 1: distinct xs
            using D by auto
        }
  qed
have $\bigwedge_x. x \in \text{set } xs \implies x < \text{oid}$
using L by auto
hence distinct (insert-spec $xs$ (oid, Some id))
using $S \text{IH}[OF 1]$ by blast
hence $a \notin \text{set } (\text{insert-spec } xs \text{ (oid, Some id)}) \land \text{distinct } (\text{insert-spec } xs \text{ (oid, Some id)})$
using $0$ by auto
}
from this $S \text{ T}$ show distinct (insert-spec ($a \# xs$) (oid, ref))
by clarsimp
qed
qed

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

lemma insert-somewhere:
assumes ref = None $\lor$ (ref = Some r $\land$ r $\in$ set list)
shows $\exists xs ys. list = xs @ ys \land \text{insert-spec list (oid, ref)} = xs @ oid \neq ys$
using assms proof(induction list)
assume ref = None $\lor$ ref = Some r $\land$ r $\in$ set []
thus $\exists xs ys. [] = xs @ ys \land \text{insert-spec [] (oid, ref)} = xs @ oid \neq ys$
proof
assume ref = None
thus $\exists xs ys. [] = xs @ ys \land \text{insert-spec [] (oid, ref)} = xs @ oid \neq ys$
by auto
next
assume ref = Some r $\land$ r $\in$ set []
thus $\exists xs ys. [] = xs @ ys \land \text{insert-spec [] (oid, ref)} = xs @ oid \neq ys$
by auto
qed
next
fix a list
assume 1: ref = None $\lor$ ref = Some r $\land$ r $\in$ set (a#list)
and IH: ref = None $\lor$ ref = Some r $\land$ r $\in$ set list $\implies$
$\exists xs ys. list = xs @ ys \land \text{insert-spec list (oid, ref)} = xs @ oid \neq ys$
show $\exists xs ys. a \# list = xs @ ys \land \text{insert-spec (a \# list) (oid, ref)} = xs @ oid$
# ys
proof(rule disjE[OF 1])
assume ref = None
thus $\exists xs ys. a \# list = xs @ ys \land \text{insert-spec (a \# list) (oid, ref)} = xs @ oid$
# ys
by force
next
assume ref = Some r $\land$ r $\in$ set (a \# list)
hence 2: r = a $\lor$ r $\in$ set list and 3: ref = Some r
by auto

23
show ∃xs ys. a # list = xs @ ys ∧ insert-spec (a # list) (oid, ref) = xs @ oid # ys
proof(rule disjE[OF 2])
  assume r = a
  thus ∃xs ys. a # list = xs @ ys ∧ insert-spec (a # list) (oid, ref) = xs @ oid # ys
  using 3 by (metis append-Cons append-Nil insert-spec.simps(3))
next
  assume r ∈ set list
  from this obtain xs ys
    where list = xs @ ys ∧ insert-spec list (oid, ref) = xs @ oid # ys
    using IH 3 by auto
  thus ∃xs ys. a # list = xs @ ys ∧ insert-spec (a # list) (oid, ref) = xs @ oid # ys
  using 3 by clarsimp (metis append-Cons append-Nil)
qed
qed
qed

lemma insert-first-part:
assumes ref = None ∨ (ref = Some r ∧ r ∈ set xs)
shows insert-spec (xs @ ys) (oid, ref) = (insert-spec xs (oid, ref)) @ ys
using assms proof(induction ys rule: rev-induct)
  assume ref = None ∨ ref = Some r ∧ r ∈ set xs
  thus insert-spec (xs @ xsa @ []) (oid, ref) = insert-spec xs (oid, ref) @ []
    by auto
next
  fix x xsa
  assume IH: ref = None ∨ ref = Some r ∧ r ∈ set xs ==> insert-spec (xs @ xsa)
      (oid, ref) = insert-spec xs (oid, ref) @ xsa
  and ref = None ∨ ref = Some r ∧ r ∈ set xs
  thus insert-spec (xs @ xsa @ [x]) (oid, ref) = insert-spec xs (oid, ref) @ xsa @ [x]
    proof(induction xs)
      assume ref = None ∨ ref = Some r ∧ r ∈ set []
      thus insert-spec ([] @ xsa @ [x]) (oid, ref) = insert-spec [] (oid, ref) @ xsa @ [x]
        by auto
next
  fix a xs
  assume 1: ref = None ∨ ref = Some r ∧ r ∈ set (a # xs)
  and 2: ((ref = None ∨ ref = Some r ∧ r ∈ set xs ==\implies insert-spec (xs @ xsa)
   (oid, ref) = insert-spec xs (oid, ref) @ xsa) ==\implies
    ref = None ∨ ref = Some r ∧ r ∈ set xs ==\implies insert-spec (xs @ xsa @ [x])
      (oid, ref) = insert-spec xs (oid, ref) @ xsa @ [x])
  and 3: (ref = None ∨ ref = Some r ∧ r ∈ set (a # xs) ==\implies insert-spec ((a
  # xs) @ xsa) (oid, ref) = insert-spec (a # xs) (oid, ref) @ xsa)
  show insert-spec ((a # xs) @ xsa @ [x]) (oid, ref) = insert-spec (a # xs) (oid,
      ref) @ xsa @ [x]
proof (rule disjE[OF 1])
  assume ref = None
  thus insert-spec ((a # xs) @ xsa @ [x]) (oid, ref) = insert-spec (a # xs) (oid, ref) @ xsa @ [x]
    by auto
next
  assume ref = Some r ∧ r ∈ set (a # xs)
  thus insert-spec ((a # xs) @ xsa @ [x]) (oid, ref) = insert-spec (a # xs) (oid, ref) @ xsa @ [x]
    using 2 3 by auto
qed
qed

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

2.3 Properties of the interp-ins function

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

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

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

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

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

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

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 3.1H by auto

qed

lemma succ-rel-rem-head:
assumes distinct (x # xs)
shows \{(p, n) \in succ-rel (x # xs). p \neq x\} = succ-rel xs

proof
have head-notin: x \notin fst \cdot succ-rel xs
  using assms by (simp add: succ-rel-set-fst)
moreover obtain y where (x, y) \in succ-rel (x # xs)
  by (cases xs, auto)
moreover have succ-rel (x # xs) = \{(x, y)\} \cup succ-rel xs
  using calculation head-notin image-iff by (cases xs, fastforce+)
moreover from this have \forall n. (x, n) \in succ-rel (x # xs) \Rightarrow n = y
  by (metis Pair-inject fst-conv head-notin image-eqI insertE insert-is-Un)
hence \{(p, n) \in succ-rel (x # xs). p \neq x\} = succ-rel (x # xs) - \{(x, y)\}
  by blast
moreover have succ-rel (x # xs) - \{(x, y)\} = succ-rel xs
  using image-iff calculation by fastforce
ultimately show \{(p, n) \in succ-rel (x # xs). p \neq x\} = succ-rel xs
  by simp
qed

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

proof(cases list)
case Nil
then show ?thesis using assms by auto
next
case (Cons a list)
moreover from this have n = Some a
  by (metis Un-iff assms singletonI succ-rel.functional)
ultimately show ?thesis by simp
qed

lemma succ-rel-insert-alt:
assumes a \neq ref
  and distinct (oid # a # b # list)
shows insert-alt (succ-rel (a # b # list)) (oid, ref) = 
  \{(a, Some b)\} \cup insert-alt (succ-rel (b # list)) (oid, ref)

proof(cases \exists n. (ref, n) \in succ-rel (a # b # list))
case True
hence insert-alt (succ-rel (a # b # list)) (oid, ref) = 
  \{(p, n) \in succ-rel (a # b # list). p \neq ref\} \cup \{(ref, Some oid)\} \cup 
  \{(i, n). i = oid \land (ref, n) \in succ-rel (a # b # list)\}
  by simp

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

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

moreover have \text{insert-alt} (\text{succ-rel} (b \# \text{list})) (\text{oid}, \text{ref}) = \\
\{(p, n) \in \text{succ-rel} (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} (b \# \text{list})\}

proof –

have \exists n. (\text{ref}, n) \in \text{succ-rel} (b \# \text{list})

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

thus \text{thesis} by \text{simp}

qed

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

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

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

next

case \text{False}

then show \text{thesis} by \text{auto}

qed

lemma \text{succ-rel-insert-head}:

assumes \text{distinct} (\text{ref} \# \text{list})

shows \text{succ-rel} (\text{insert-spec} (\text{ref} \# \text{list}) (\text{oid}, \text{Some } \text{ref})) = \\
\text{insert-alt} (\text{succ-rel} (\text{ref} \# \text{list})) (\text{oid}, \text{ref})

proof –

obtain n where \text{ref-in-rel} (\text{ref}, n) \in \text{succ-rel} (\text{ref} \# \text{list})

by (cases \text{list}, \text{auto})

moreover from \text{this} have \{(p, n) \in \text{succ-rel} (\text{ref} \# \text{list}). p \neq \text{ref}\} = \text{succ-rel} \text{list}

using \text{assms succ-rel-rem-head by (metis (mono-tags, lifting))}

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

proof –

have \land nx. (\text{ref}, nx) \in \text{succ-rel} (\text{ref} \# \text{list}) \Rightarrow nx = n

using \text{assms by (simp add: succ-rel-functional ref-in-rel)}

hence \{(i, n) \in \text{succ-rel} (\text{ref} \# \text{list}). i = \text{ref}\} \subseteq \{(\text{ref}, n)\}

by \text{blast}

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

by (simp add: ref-in-rel)

ultimately show \text{thesis} by \text{blast}

qed

moreover have \text{insert-alt} (\text{succ-rel} (\text{ref} \# \text{list})) (\text{oid}, \text{ref}) = \\
\{(p, n) \in \text{succ-rel} (\text{ref} \# \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} (\text{ref} \# \text{list})\}

proof –

have \exists n. (\text{ref}, n) \in \text{succ-rel} (\text{ref} \# \text{list})

using \text{ref-in-rel} by \text{blast}

thus \text{thesis} by \text{simp}

qed

ultimately have \text{insert-alt} (\text{succ-rel} (\text{ref} \# \text{list})) (\text{oid}, \text{ref}) =
\[ \text{succ-rel list } \cup \{(\text{ref, Some oid})\} \cup \{(\text{oid, n})\} \]

by simp

moreover have \(\text{succ-rel (oid }\#\text{ list)} = \{(\text{oid, n})\} \cup \text{succ-rel list}\)

using assms \text{ref-in-rel succ-rel-swap-head by metis}

hence \(\text{succ-rel (ref }\#\text{ oid }\#\text{ list)} = \{(\text{ref, Some oid}), (\text{oid, n})\} \cup \text{succ-rel list}\)

by auto

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

by auto

qed

lemma \text{succ-rel-insert-later}:

assumes \(\text{succ-rel (insert-spec (b }\#\text{ list)} (\text{oid, Some ref})) = \text{insert-alt (succ-rel (b }\#\text{ list)} (\text{oid, ref)}\)

and \(a \neq \text{ref}\)

and distinct \((a \neq b \neq \text{list})\)

shows \(\text{succ-rel (insert-spec (a }\#\text{ b }\#\text{ list)} (\text{oid, Some ref})) = \text{insert-alt (succ-rel (a }\#\text{ b }\#\text{ list)} (\text{oid, ref)}\)

proof –

have \(\text{succ-rel (a }\#\text{ b }\#\text{ list)} = \{(a, \text{Some b})\} \cup \text{succ-rel (b }\#\text{ list)}\)

by simp

moreover have \(\text{insert-spec (a }\#\text{ b }\#\text{ list)} (\text{oid, Some ref}) = a \# (\text{insert-spec (b }\#\text{ list)} (\text{oid, Some ref}))\)

using assms(2) by simp

hence \(\text{succ-rel (insert-spec (a }\#\text{ b }\#\text{ list)} (\text{oid, Some ref})) = \{(a, \text{Some b})\} \cup \text{succ-rel (insert-spec (b }\#\text{ list)} (\text{oid, Some ref}))\)

by auto

hence \(\text{succ-rel (insert-spec (a }\#\text{ b }\#\text{ list)} (\text{oid, Some ref})) = \{(a, \text{Some b})\} \cup \text{insert-alt (succ-rel (b }\#\text{ list)} (\text{oid, ref)}\)

using assms(1) by auto

moreover have \(\text{insert-alt (succ-rel (a }\#\text{ b }\#\text{ list)} (\text{oid, ref}) = \{(a, \text{Some b})\} \cup \text{insert-alt (succ-rel (b }\#\text{ list)} (\text{oid, ref)}\)

using \text{succ-rel-insert-alt assms(2)} by auto

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

qed

lemma \text{succ-rel-insert-Some}:

assumes \(\text{distinct list}\)

shows \(\text{succ-rel (insert-spec list (oid, Some ref)) = \text{insert-alt (succ-rel list) (oid, ref)}\)

using assms \text{proof(induction list)}

case \text{Nil} 

then show \(\text{succ-rel (insert-spec [] (oid, Some ref)) = \text{insert-alt (succ-rel [])} (oid, ref)}\)

by simp

next 

case \((\text{Cons a list})\)

hence \(\text{distinct (a }\#\text{ list)}\)

by simp
then show \( \text{succ-rel} (\text{insert-spec} (a \# \text{list}) (\text{oid}, \text{Some ref})) = \text{insert-alt} (\text{succ-rel} (a \# \text{list}) (\text{oid}, \text{ref})) \)

proof (cases \( a = \text{ref} \))

\begin{itemize}
  \item case True
    then show \( \text{thesis} \)
      using succ-rel-insert-head \( \langle \text{distinct} (a \# \text{list}) \rangle \) by metis
  \item case False
    hence \( a \neq \text{ref} \)
      by simp
    moreover have \( \text{succ-rel} (\text{insert-spec} \text{list} (\text{oid}, \text{Some ref})) = \text{insert-alt} (\text{succ-rel} (a \# \text{list}) (\text{oid}, \text{ref})) \)
      using Cons.IH Cons.prems by auto
    ultimately show \( \text{succ-rel} (\text{insert-spec} (a \# \text{list}) (\text{oid}, \text{Some ref})) = \text{insert-alt} (\text{succ-rel} (a \# \text{list}) (\text{oid}, \text{ref})) \)
      by (cases list, force, metis Cons.prems succ-rel-insert-later)
\end{itemize}

qed

qed

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

\textbf{theorem} \text{insert-alt-equivalent}:

\begin{itemize}
  \item assumes \text{insert-ops ops}
  \item and \text{head} \notin \text{fst} \cdot \text{set ops}
  \item and \( \forall r. \text{Some } r \in \text{snd} \cdot \text{set ops} \Rightarrow r \neq \text{head} \)
\end{itemize}

\text{shows} \( \text{succ-rel} (\text{head} \# \text{interp-ins ops}) = \text{interp-alt} \text{head ops} \)

\textbf{using} \text{assms} \textbf{proof} (induction \text{ops} rule: \text{List.rev-induct})

\begin{itemize}
  \item case \text{Nil}
    then show \( \text{succ-rel} (\text{head} \# \text{interp-ins []}) = \text{interp-alt} \text{head []} \)
      by (simp add: \text{interp-ins-def interp-alt-def})
  \item \textbf{next}
  \item case \text{IH: } \text{succ-rel} (\text{head} \# \text{interp-ins xs}) = \text{interp-alt} \text{head xs}
    using \text{snoc by auto}
  \item have \text{distinct-list: distinct} (\text{head} \# \text{interp-ins xs})
    \textbf{proof}
    \begin{itemize}
        \item have \text{distinct} (\text{interp-ins xs})
        \textbf{using} \text{interp-ins-distinct snoc.prems(1) by blast}
        \item moreover have \text{set} (\text{interp-ins xs}) \subseteq \text{fst} \cdot \text{set xs}
        \textbf{using} \text{interp-ins-subset snoc.prems(1) by fastforce}
        \item ultimately show \text{distinct} (\text{head} \# \text{interp-ins xs})
        \textbf{using} \text{snoc.prems(2) by auto}
    \end{itemize}
\end{itemize}

\textbf{qed}

\textbf{obtain} \( \text{oid } r \text{ where x-pair: } x = (\text{oid}, \text{r}) \text{ by force} \)

then show \( \text{succ-rel} (\text{head} \# \text{interp-ins} (\text{xs } @ [x])) = \text{interp-alt} \text{head} (\text{xs } @ [x]) \)
\textbf{proof} (cases \( r \))

\begin{itemize}
  \item case \text{None}
    have \text{interp-alt head} (\text{xs } @ [x]) = \text{insert-alt} (\text{interp-alt head} \text{xs}) (\text{oid}, \text{head})
      by (simp add: \text{interp-alt-def None x-pair})
    moreover have \( \ldots = \text{insert-alt} (\text{succ-rel} (\text{head} \# \text{interp-ins xs})) (\text{oid}, \text{head}) \)
      by (simp add: IH)
\end{itemize}

\textbf{qed}
moreover have ... = succ-rel (insert-spec (head # interp-ins xs) (oid, Some head))
  using distinct-list succ-rel-insert-\textit{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 \textit{thesis} by simp

next
  case (Some ref)
  have ref \neq head
    by (simp add: Some snoc.prems(3) x-pair)
  have interp-alt head (xs @ [x]) = insert-alt (interp-alt head xs) (oid, ref)
    by (simp add: interp-alt-def Some x-pair)
  moreover have ... = insert-alt (succ-rel (head # interp-ins xs)) (oid, ref)
    by (simp add: IH)
  moreover have ... = succ-rel (insert-spec (head # interp-ins xs) (oid, Some ref))
    using distinct-list succ-rel-insert-\textit{Some} by metis
  moreover have ... = succ-rel (head # (insert-spec (interp-ins xs) (oid, Some ref)))
    using ref \neq head by auto
  moreover have ... = succ-rel (head # (interp-ins (xs @ [x])))
    by (simp add: interp-ins-tail-unfold Some x-pair)
ultimately show \textit{thesis} by simp
qed

2.5 The list-order predicate

\textit{list-order} \textit{ops} \textit{x y} holds iff, after interpreting the list of insertion operations \textit{ops}, the list element with ID \textit{x} appears before the list element with ID \textit{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 \textit{ops} \textit{x y} ≡ \exists \textit{xs} \textit{ys} \textit{zs}. interp-ins \textit{ops} = \textit{xs} @ [x] @ \textit{ys} @ [y] @ \textit{zs}

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

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

32
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

thus $\text{list-order op-list x z}$
using assms by (metis append.assoc list-orderI)
qed

lemma insert-preserves-order:
assumes insert-ops ops and insert-ops rest
and $\text{rest} = \text{before @} \text{after}$
and $\text{ops} = \text{before } (\text{oid}, \text{ref}) \# \text{after}$
shows $\exists \text{xys zs. interp-ins rest} = \text{xzs @} \text{zs} \land \text{interp-ins ops} = \text{xz @} \text{ys @} \text{zs}$
using assms proof(induction after arbitrary: rest ops rule: List.rev-induct)
case Nil
then have 1: interp-ins rest = insert-spec (interp-ins before) (oid, ref)
by (simp add: interp-ins-tail-unfold)
then show $\exists \text{xys zs. interp-ins rest} = \text{xzs @} \text{zs} \land \text{interp-ins ops} = \text{xz @} \text{ys @} \text{zs}$
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 $\text{thesis}$ by blast
next
case (Some a)
then show \( \text{thesis} \)

proof (cases \( a \in \text{set} \) (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 \( \text{thesis} \) 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 \( \text{thesis} \) by blast

qed

qed

next

  case (snoc oper op-list)
  then have insert-ops ((before @ (oid, ref)) @ [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)) @ [oper]

using insert-ops-appendD by blast+

then obtain \( zs \) \( ys \) \( zs \) where IH1: interp-ins (before @ op-list) = \( xs @ zs \)
  
and IH2: interp-ins (before @ (oid, ref)) @ [oper] = \( xs @ ys @ zs \)

using snoc.IH by blast

obtain \( i2 \) \( r2 \) where oper = (\( i2 \), \( r2 \)) by force

then show \( \exists xs \) \( ys \) \( zs \). interp-ins rest = \( xs @ zs \) \& interp-ins ops = \( xs @ ys @ zs \)

proof (cases \( r2 \))

  case None
  hence interp-ins (before @ op-list @ [oper]) = \( i2 \) @ [i2 @ xs]
    
by (metis IH1 \( \langle oper = (i2, r2) \rangle \) append.assoc append.Cons insert-spec.simps(1)

interp-ins-tail-unfold)

moreover have interp-ins (before @ (oid, ref)) @ [op-list] @ [oper]) = \( i2 \) @ [i2 @ xs]

@ [ys @ zs]

by (metis IH2 None \( \langle oper = (i2, r2) \rangle \) append.assoc append.Cons insert-spec.simps(1)

interp-ins-tail-unfold)

ultimately show \( \text{thesis} \)

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) \( r \in \text{set} \) xs | (r-ys) \( r \in \text{set} \) ys | (r-zs) \( r \in \text{set} \) zs |
(r-nonex) \( r \notin \text{set} (xs @ ys @ zs) \)
by auto
then show \( \exists xs \ ys \ zs. \ \text{interp-ins rest} = xs @ zs \land \text{interp-ins ops} = xs @ ys @ zs \)
proof(cases)
case r-xs
from this have \( \text{insert-spec} (xs @ ys @ zs) (i2, \text{Some} \ r) = \)
(\( \text{insert-spec} xs (i2, \text{Some} \ r) \)) @ ys @ zs
by (meson insert-first-part)
moreover have \( \text{insert-spec} (xs @ zs) (i2, \text{Some} \ r) = (\text{insert-spec} xs (i2, \text{Some} \ r)) @ zs \)
by (meson r-xs insert-first-part)
ultimately show ?thesis
using 1 2 ⟨oper = (i2, r2)⟩ snoc.prems by auto
next
case r-ys
hence \( r \notin \text{set} xs \) and \( r \notin \text{set} zs \)
using IH2_3 ops2 interp-ins-distinct by force+
moreover from this have \( \text{insert-spec} (xs @ ys @ zs) (i2, \text{Some} \ r) = \)
\( xs @ (\text{insert-spec} ys (i2, \text{Some} \ r)) @ zs \)
using insert-first-part insert-second-part insert-spec-nonex
by (metis Some UnE r-ys set-append)
moreover have \( \text{insert-spec} (xs @ zs) (i2, \text{Some} \ r) = xs @ zs \)
by (simp add: Some calculation(1) calculation(2))
ultimately show ?thesis
using 1 2 ⟨oper = (i2, r2)⟩ snoc.prems by auto
next
case r-zs
hence \( r \notin \text{set} xs \) and \( r \notin \text{set} ys \)
using IH2_3 ops2 interp-ins-distinct by force+
moreover from this have \( \text{insert-spec} (xs @ ys @ zs) (i2, \text{Some} \ r) = \)
\( xs @ ys @ (\text{insert-spec} zs (i2, \text{Some} \ r)) \)
by (metis Some UnE insert-second-part insert-spec-nonex set-append)
moreover have \( \text{insert-spec} (xs @ zs) (i2, \text{Some} \ r) = xs @ (\text{insert-spec} zs (i2, \text{Some} \ r)) \)
by (simp add: r-zs calculation(1) insert-second-part)
ultimately show ?thesis
using 1 2 ⟨oper = (i2, r2)⟩ snoc.prems by auto
next
case r-nonex
then have \( \text{insert-spec} (xs @ ys @ zs) (i2, \text{Some} \ r) = xs @ ys @ zs \)
by simp
moreover have \( \text{insert-spec} (xs @ zs) (i2, \text{Some} \ r) = xs @ zs \)
using r-nonex by simp
ultimately show ?thesis
using 1 2 ⟨oper = (i2, r2)⟩ snoc.prems by auto
qed
qed
qed
lemma distinct-fst:
  assumes distinct (map fst A)
  shows distinct A
  using assms by (induction A) auto

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

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

lemma length-diff-Suc-exists:
  assumes length xs − length ys = Suc m

36
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 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

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

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

37
have distinct (map fst A) and distinct (map fst xa) and
sorted (map fst A) and sorted (map fst xa)
using less.prems by (auto simp add: insert-ops-def spec-ops-def)
hence distinct A and distinct xa
by (auto simp add: distinct-fst)
then show list-order xa x y
proof (cases length xa = length A)
case 0
hence set A = set xa
using set-subset-length-eq less.prems (3) ⟨distinct A⟩ ⟨distinct xa⟩ diff-is-0-eq
by blast
hence A = xa
using ⟨distinct (map fst A)⟩ ⟨distinct (map fst xa)⟩
⟨sorted (map fst A)⟩ ⟨sorted (map fst xa)⟩ map-sorted-distinct-set-unique
by (metis distinct-map less.prems (3) subset-Un-eq)

case (Suc nat)
then obtain e where e ∈ set xa and e /∈ set A
using length-diffSuc-exists less.prems (3) ⟨distinct A⟩ ⟨distinct xa⟩
diff-Suc-1 diff-commute length-remove1 lessSuc-eq Suc ⟨e ∈ set xa⟩
by (simp add: remove1-append)
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) subsetD)
ultimately show ?thesis
by (simp add: Suc less.IH less.prems (4))
qed
then obtain xs ys zs where interp-ins (remove1 e xa) = xs @ x # ys @ y #
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 zsa ysa zsa where interp-ins (ps @ ss) = xsa @ zsa
and interp-xa: interp-ins (ps @ (oid, ref) # ss) = xsa @ ysa @ zsa
using insert-preserves-order e-pair by metis
moreover have \texttt{zsa-zsa1}: \texttt{zsa} @ \texttt{zsa} = \texttt{xs} @ \texttt{x} @ \texttt{ys} @ \texttt{y} @ \texttt{zs}

using \texttt{interp-ins-def remove1-append calculation xa-split by auto}

then show \texttt{list-order xa x y}

proof\((\text{cases length zsa} \leq \text{length xs})\)

case \texttt{True}

then obtain \texttt{ts} where \texttt{zsa@ts} = \texttt{xs}

using \texttt{app-length-lt-exists zsa-zsa by blast}

hence \texttt{interp-ins xa} = (\texttt{zsa} @ \texttt{ysa} @ \texttt{ts}) @ \texttt{[x]} @ \texttt{ys} @ \texttt{[y]} @ \texttt{zs}

using \texttt{calculation xa-split by auto}

then show \texttt{list-order xa x y}

using \texttt{list-order-def by blast}

next

case \texttt{False}

then show \texttt{list-order xa x y}

proof\((\text{cases length } \texttt{xs} \leq \text{length } (\texttt{xs} @ \texttt{x} # \texttt{ys}))\)

case \texttt{True}

have \texttt{zsa-zsa1}: \texttt{zsa} @ \texttt{zsa} = (\texttt{xs} @ \texttt{x} # \texttt{ys}) @ (\texttt{y} # \texttt{zs})

by \((\text{simp add: zsa-zsa})\)

then obtain \texttt{us} where \texttt{zsa} @ \texttt{us} = \texttt{xs} @ \texttt{x} # \texttt{ys}

using \texttt{app-length-lt-exists True by blast}

moreover from this have \texttt{hs} \texttt{xs} @ \texttt{x} # (drop \((\text{Suc (length xs)})\) \texttt{xs}) = \texttt{xsa}

using \texttt{append-eq-append-conv-if id-take-nth-drop linorder-not-less nth-append nth-append-length False by metis}

moreover have \texttt{us} @ \texttt{y} # \texttt{zs} = \texttt{zsa}

by \((\text{metis True zsa-zsa1 append-eq-append-conv-if append-eq-conv-conj calculation(1))}\)

ultimately have \texttt{interp-ins za} = \texttt{xs} @ \texttt{[x]} @

((drop \((\text{Suc (length xs)})\) \texttt{xs}) @ \texttt{ysa} @ \texttt{us}) @ \texttt{[y]} @ \texttt{zs}

by \((\text{simp add: e-pair interp-za za-split})\)

then show \texttt{list-order xa x y}

using \texttt{list-order-def by blast}

next

case \texttt{False}

hence \texttt{length } (\texttt{xs} @ \texttt{x} # \texttt{ys}) < \texttt{length zsa}

using \texttt{not-less by blast}

hence \texttt{length } (\texttt{xs} @ \texttt{x} # \texttt{ys} @ \texttt{[y]}) \leq \texttt{length zsa}

by \texttt{simp}

moreover have \texttt{(xs @ x # ys @ [y]) @ zs} = \texttt{zsa} @ \texttt{zsa}

by \((\text{simp add: zsa-zsa})\)

ultimately obtain \texttt{vs} where \texttt{(xs @ x # ys @ [y]) @ vs} = \texttt{zsa}

using \texttt{app-length-lt-exists by blast}

hence \texttt{hsa @ ysa @ zsa} = \texttt{xs} @ \texttt{[x]} @ \texttt{ys} @ \texttt{[y]} @ \texttt{vs} @ \texttt{ysa} @ \texttt{zsa}

by \texttt{simp}

hence \texttt{interp-ins za} = \texttt{xs} @ \texttt{[x]} @ \texttt{ys} @ \texttt{[y]} @ (\texttt{vs} @ \texttt{ysa} @ \texttt{zsa})

using \texttt{e-pair interp-za za-split by auto}

then show \texttt{list-order xa x y}

using \texttt{list-order-def by blast}

qed

qed
In this section we show that our list specification is stronger than the \text{\textit{A\textsubscript{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 \text{\textit{A\textsubscript{strong}}} specification.

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

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

1. Each event $e = \text{\textit{do}}(\text{\textit{op}}, \text{\textit{w}}) \in H$ returns a sequence of elements $w = a_0 \ldots a_{n-1}$, where $a_i \in \text{\textit{elems}}(A)$, such that
   
   (a) $w$ contains exactly the elements visible to $e$ that have been inserted, but not deleted:
   
   $$\forall a. a \in w \iff (\text{\textit{do}}(\text{\textit{ins}}(a, \_), \_) \leq \text{\textit{vis}} e) \wedge \neg (\text{\textit{do}}(\text{\textit{del}}(a, \_), \_) \leq \text{\textit{vis}} e).$$

   (b) The order of the elements is consistent with the list order:
   
   $$\forall i, j. (i < j) \implies (a_i, a_j) \in \text{\textit{lo}}.$$

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

2. The list order $\text{\textit{lo}}$ 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 $\text{\textit{A\textsubscript{strong}}}$ using a simplified interpretation function that defines only insertion and deletion on a single list.

theory List-Spec
  imports Insert-Spec
begin

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

```plaintext
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_.

```plaintext
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))
```

```plaintext
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.
```

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

The _make-insert_ function generates a new operation for insertion into a given index in a given list. The exclamation mark is Isabelle’s list subscript operator.

```plaintext
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.

```plaintext
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 ops :: ("oid::{linorder} × ("oid, 'val) list-op) set

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

3.1 Lemmas about insertion and deletion

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

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

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

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

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

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

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

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

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

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

lemma \text{insertions-subset:}
assumes \( \text{list-ops} A \text{ and list-ops} B \)
and \( A \subseteq B \)
shows \( \text{set} (\text{insertions} \text{A}) \subseteq \text{set} (\text{insertions} \text{B}) \)
using assms
proof (induction \text{B arbitrary: A rule: List.rev-induct})
case Nil
then show \( \text{set} (\text{insertions} \text{A}) \subseteq \text{set} (\text{insertions} []) \)
by (simp add: \text{insertions-def map-filter-simps(2)})
next
case (snoc a \text{ops})
obtain \text{oid oper} where \text{a-pair: } a = (\text{oid, oper})
by fastforce
have \( \text{list-ops} \text{ops} \)
using list-ops-def spec-ops-rem-last snoc.prems(2) by blast
then show \( \text{set} (\text{insertions} \text{A}) \subseteq \text{set} (\text{insertions} (\text{ops} @ [a])) \)
proof (cases \text{a} \in \text{set} \text{A})
case True
then obtain \( \text{as bs} \) where \text{A-split: } A = \text{as} @ \text{a} \# \text{bs} \wedge \text{a} \notin \text{set as}
by (meson split-list-first)
hence \text{remove1 a A} = \text{as} @ \text{bs}
by (simp add: remove1-append)
hence \text{as-bs: } \text{insertions} (\text{remove1 a A}) = \text{insertions} \text{as} @ \text{insertions} \text{bs}
by (simp add: insertions-def map-filter-append)
moreover have \( A = \text{as} \otimes [a] \otimes \text{bs} \)
by (simp add: A-split)
hence \( \text{as-a-bs: insertions A = insertions as \otimes insertions [a] \otimes insertions bs} \)
by (metis insertions-def map-filter-append)
moreover have \( \text{IH}: \text{set} \ (\text{insertions (remove1 a A)}) \subseteq \text{set} \ (\text{insertions ops}) \)
proof –
  have \( \text{list-ops (remove1 a A)} \)
    using \( \text{snoc.prems(1) list-ops-def spec-ops-remove1 by blast} \)
moreover have \( \text{set (remove1 a A)} \subseteq \text{set} \ \text{ops} \)
proof –
  have \( \text{distinct A} \)
    using \( \text{snoc.prems(1) list-ops-def spec-ops-distinct by blast} \)
  hence \( a \notin \text{set (remove1 a A)} \)
  by \( \text{auto} \)
moreover have \( \text{set (ops @ [a]) = set ops \cup \{a\}} \)
by \( \text{auto} \)
moreover have \( \text{set (remove1 a A)} \subseteq \text{set} \ \text{A} \)
by (simp add: set-remove1-subset)
ultimately show \( \text{set (remove1 a A)} \subseteq \text{set} \ \text{ops} \)
  using \( \text{snoc.prems(3) by blast} \)
qed
ultimately show \( \text{?thesis} \)
  by (simp add: (list-ops ops) snoc.IH)
qed
ultimately show \( \text{?thesis} \)
proof (cases oper)
case \( \text{(Insert ref val)} \)
hence \( \text{insertions [a] = \{(oid, ref)\}} \)
  by (simp add: insertions-def map-filter-Some a-pair)
  hence \( \text{set (insertions A) = set (insertions (remove1 a A)) \cup \{(oid, ref)\}} \)
  using \( \text{as-a-bs as-bs by auto} \)
moreover have \( \text{set (insertions (ops @ [a])) = set (insertions ops) \cup \{(oid, ref)\}} \)
by (simp add: Insert a-pair insertions-last-ins)
ultimately show \( \text{?thesis} \)
  using \( \text{IH by auto} \)
next
case \( \text{(Delete ref)} \)
hence \( \text{insertions [a] = \[]} \)
  by (simp add: insertions-def map-filter-None a-pair)
  hence \( \text{set (insertions A) = set (insertions (remove1 a A))} \)
  using \( \text{as-a-bs as-bs by auto} \)
moreover have \( \text{set (insertions (ops @ [a])) = set (insertions ops)} \)
by (simp add: Delete a-pair insertions-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
  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 \( \forall i. i \in \text{set (map fst ops)} \Longrightarrow i < \text{oid} \)
      using a-pair list-ops-def snoc.prems spec-ops-id-inc by fastforce
    hence \( \forall i. i \in \text{set (map fst (insertions ops))} \Longrightarrow i < \text{oid} \)
      using insertions-fst-subset by blast
    moreover have list-op-deps oper = set-option ref
      using Insert by (cases ref, auto)
    hence \( \forall r. r \in \text{set-option ref} \Longrightarrow r < \text{oid} \)
      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
lemma inserted-ids-last-ins:
  shows \(\text{inserted-ids } (xs @ [(\text{oid}, \text{Insert ref val})]) = \text{inserted-ids } xs @ [\text{oid}]\)
  by (simp add: inserted-ids-def map-filter-Some map-filter-append)

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

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

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

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

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

next 

case (snoc a ops)

obtain oid oper where a-pair: \( a = (\text{oid}, \text{oper}) \)

by fastforce

then show ref \( \in \) set (deleted-ids (ops @ [a])) \( \iff \exists i. \ (i, \text{Delete ref}) \in \) set (ops @ [a])

proof (cases oper)

case (Insert r v)

moreover from this have deleted-ids (ops @ [a]) = deleted-ids ops

by (simp add: a-pair deleted-ids-last-ins)

ultimately show ?thesis

using a-pair snoc. IH by auto

next 

case (Delete r)

moreover from this have deleted-ids (ops @ [a]) = deleted-ids ops @ [r]

by (simp add: a-pair deleted-ids-last-del)

ultimately show ?thesis

using a-pair snoc. IH by auto

qed 

qed 

lemma deleted-ids-refs-older:

assumes list-ops (ops @ [(oid, oper)])

shows \( \forall \text{ref. ref} \in \) set (deleted-ids ops) \( \implies \) ref < oid

proof --

fix ref

assume ref \( \in \) set (deleted-ids ops)

then obtain i where in-ops: \( (i, \text{Delete ref}) \in \) set ops

using deleted-ids-exist by blast

have ref < i

proof --

have \( \forall \text{r oper r.} \ (i, \text{oper}) \in \) set ops \( \implies \) r \( \in \) list-op-deps oper \( \implies \) r < i

by (meson assms list-ops-def spec-ops-ref-less spec-ops-rem-last)

thus ref < i

using in-ops by auto

qed 

moreover have i < oid

proof --

have \( \forall i. \ i \in \) set (map fst ops) \( \implies \) i < oid

using assms by (simp add: list-ops-def spec-ops-id-inc)

thus ?thesis

by (metis in-ops in-set-zipE zip-map-fst-snd)

qed

ultimately show ref < oid 

using order.strict-trans by blast

qed
3.2 Lemmas about interpreting operations

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

**Lemma interp-ops-distinct:**
- **assumes** \(\text{list-ops } \text{ops}\)
- **shows** \(\text{distinct} (\text{fst} (\text{interp-ops } \text{ops}))\)
  - by (simp add: assms interp-ins-distinct interp-ops-list-equiv list-ops-insertions)

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

hence \(\set{\text{inserted-ids \((xs \# [x])\)}} = \set{\text{inserted-ids \((xs \# [x])\)}}\)
using calculation\((3)\) by auto
moreover have ... = \(\set{\text{inserted-ids \((xs \# [x])\)}} = \set{\text{inserted-ids \((xs \# [x])\)}}\)
by blast
ultimately show thesis by auto
qed

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

lemma insert-spec-inc-length:
assumes distinct xs and \(n < \text{length} \(xs\)\)
shows length \((insert-spec \((oid, \text{Some} \((xs ! n)\)))\) = Suc (length \(xs\))
using assms proof(induction xs arbitrary: \(n, simp\))
case (Cons a xs)
  have distinct (a ≠ xs)
    using Cons.prems(1) by auto
  then show length (insert-spec (a ≠ xs) (oid, Some ((a ≠ xs) ! n))) = Suc (length (a ≠ xs))
    proof (cases n)
      case 0
      hence insert-spec (a ≠ xs) (oid, Some ((a ≠ xs) ! n)) = a ≠ oid # xs
        by simp
      then show ?thesis
        by simp
    next
      case (Suc nat)
      hence nat < length xs
        using Cons.prems(2) by auto
      hence length (insert-spec xs (oid, Some (xs ! nat))) = Suc (length xs)
        using Cons.IH Cons.prems(1) by auto
      then show ?thesis
        by (simp add: Suc)
    qed
  qed
lemma list-split-two-elems:
  assumes distinct xs
    and x ∈ set xs and y ∈ set xs
    and x ≠ y
  shows ∃ pre mid suf. xs = pre @ x ≠ mid @ y ≠ suf ∨ xs = pre @ y ≠ mid @ x ≠ suf
proof −
  obtain as bs where as-bs: xs = as @ [x] @ bs
    using assms(2) split-list-first by fastforce
  show ?thesis
    proof (cases y ∈ set as)
      case True
      then obtain cs ds where as = cs @ [y] @ ds
        using assms(3) split-list-first by fastforce
      then show ?thesis
        by (auto simp add: as-bs)
    next
      case False
      then have y ∈ set bs
        using as-bs assms(3) assms(4) by auto
      then obtain cs ds where bs = cs @ [y] @ ds
        using assms(3) split-list-first by fastforce
      then show ?thesis
        by (auto simp add: as-bs)
    qed
  qed
3.3 Satisfying all conditions of $A_{\text{strong}}$

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. $A_{\text{strong}}$ uses the visibility relation $\leq_{\text{vis}}$ to capture the operations known to a node at some arbitrary point in the execution; in the OpSet model, we can simply prove the theorem for an arbitrary OpSet, since the contents of the OpSet at a particular time on a particular node correspond exactly to the set of operations known to that node at that time.

**Theorem** inserted-but-not-deleted:
- **Assumes** list-ops ops
  - and interp-ops ops = (list, vals)
- **Shows** $a \in \text{dom}(vals) \iff (\exists \text{ref val. } (a, \text{Insert ref val}) \in \text{set ops}) \land (\nexists i. (i, \text{Delete } a) \in \text{set ops})$
- **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 $@$ [(oid, ins)])
  - and ins = make-insert (fst (interp-ops ops)) val k
  - and list = fst (interp-ops (ops $@$ [(oid, ins)]))
- **Shows** list $(\text{min } k \text{ (length list } - 1)) = \text{oid}$
- **Proof** (cases $k = 0 \lor \text{fst (interp-ops ops) } = []$)
  - case True
moreover from this
have make-insert (fst (interp-ops ops)) val k = Insert None val
and min-k: min k (length (fst (interp-ops ops))) = 0
by (cases k, auto)
hence fst (interp-ops (ops @ [(oid, ins)])) = oid ≠ fst (interp-ops ops)
using assms(2) interp-ops-unfold-last
by (metis fst-conv insert-spec.simps(1) interp-op.simps(1) prod.collapse)
ultimately show ?thesis
by (simp add: min-k assms(3))
next
  case False
moreover from this have k > 0 and fst (interp-ops ops) ≠ []
  using neq0-conv by blast+
from this obtain nat where k = Suc nat
  using gr0-implies-Suc by blast
hence make-insert (fst (interp-ops ops)) val k =
  Insert (Some ((fst (interp-ops ops)) ! (min nat (length (fst (interp-ops ops)))
  − 1))) val
  using False by (cases fst (interp-ops ops), auto)
hence fst (interp-ops (ops @ [(oid, ins)])) =
  insert-spec (fst (interp-ops ops)) (oid, Some ((fst (interp-ops ops)) ! (min
  nat (length (fst (interp-ops ops))) − 1))))
  by (metis assms(2) fst-conv interp-op.simps(1) interp-ops-unfold-last prod.collapse)
moreover have min nat (length (fst (interp-ops ops))) − 1 < length (fst (interp-ops
  ops))
  by (simp add: (fst (interp-ops ops) ≠ []); min_strict-coboundedI2)
moreover have distinct (fst (interp-ops ops))
  using interp-ops-distinct list-ops-def spec-ops-rem-last assms(1) by blast
moreover have length list = Suc (length (fst (interp-ops ops))
  using assms(3) calculation by (simp add: insert-spec-inc-length)
ultimately show ?thesis
  using assms insert-spec-nth-oid
  by (metis Suc-diff-1 (k = Suc nat) diff-Suc-1 length-greater-0-conv min-Suc-Suc)
qed

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

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

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

53
have list-order ops x x ⇒ 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 ¬ list-order ops x x
  by blast
qed

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

end

4 Interleaving of concurrent insertions

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

theory Interleaving
  imports Insert-Spec
begin

4.1 Lemmas about insert-ops

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

lemma insert-ops-split:
  assumes insert-ops ops
  and (oid, ref) ∈ 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.asgn xi-max by auto
then show \( ?\)thesis by (simp add: IH)
next
case False
  hence oid < xi
  using \( xi \neq \) oid by auto
  hence \( \forall i \in \text{set} (\text{map \textsc{fst} (\text{suf} \circ ([xi, xr])))}. \) oid < \( i \)
  using IH map-fst-append1 by auto
hence \( xs \circ [x] = \text{pre} \circ [(\text{oid, ref)]) \circ (\text{suf} \circ ([xi, xr]))] \) \( \land \)
  \( \forall i \in \text{set} (\text{map \textsc{fst} \text{pre}).} i < \text{oid} \) \( \land \)
  \( \forall i \in \text{set} (\text{map \textsc{fst} (\text{suf} \circ ([xi, xr]))}. \) oid < \( i \)
  by (simp add: IH \( x\)-pair)
then show \( ?\)thesis by blast
qed
qed
qed

lemma insert-ops-split-2:
assumes insert-ops \( as \) \( \circ \text{ops} \)
and \( (\text{clid}, \ (x)) \in \text{set} \) \( \text{ops} \)
and \( (\text{clid}, \ (y)) \in \text{set} \) \( \text{ops} \)
and \( \text{clid} \ < \ y \)
shows \( \exists bs \) \( \) \( cs \) \( . \) \( \text{ops} = as \circ ([\text{clid}, \ x]) \circ bs \circ ([\text{clid}, \ y]) \circ cs \) \( \land \)
  \( \forall i \in \text{set} (\text{map \textsc{fst} as).} i < \text{clid} \) \( \land \)
  \( \forall i \in \text{set} (\text{map \textsc{fst} bs).} \text{clid} < i \land i < \text{clid} \) \( \land \)
  \( \forall i \in \text{set} (\text{map \textsc{fst} cs).} \text{clid} < i \)\nproof
  obtain as \( \) as1 where \( \text{x-split:} \) \( \text{ops} = as \circ ([\text{clid}, \ x]) \circ as1 \) \( \land \)
  \( \forall i \in \text{set} (\text{map \textsc{fst} as).} i < \text{clid} \) \( \land \) \( \forall i \in \text{set} (\text{map \textsc{fst} as1).} \text{clid} < i \)
  using assms insert-ops-split by blast
  hence insert-ops \( (as \circ ([\text{clid}, \ x])) \circ as1 \) \( \land \)
  using assms(1) by auto
  hence insert-ops as1 \( \land \)
  using assms(1) insert-ops-rem-prefix by blast
have \( (\text{clid}, \ (y)) \in \text{set} as1 \)
  using \( x\)-split assmss by auto
from this obtain \( bs \) \( \) \( cs \) \( where \) \( y\)-split: \( as1 = bs \circ ([\text{clid}, \ (y)]) \circ cs \) \( \land \)
  \( \forall i \in \text{set} (\text{map \textsc{fst} bs).} i < \text{clid} \) \( \land \) \( \forall i \in \text{set} (\text{map \textsc{fst} cs).} \text{clid} < i \)
  using assms insert-ops-split \( \circ \text{ops} \) \( \circ \text{as1} \) by blast
  hence ops = as \circ ([\text{clid}, \ x]) \circ bs \circ ([\text{clid}, \ y]) \circ cs \)
  using \( x\)-split by blast
moreover have \( \forall i \in \text{set} (\text{map \textsc{fst} bs).} \text{clid} < i \land i < \text{clid} \)
  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 \( (xs \circ ([i1, r1]) \circ ys \circ ([i2, r2])) \)

shows \( i_1 < i_2 \)

proof –

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

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

ultimately show \( i_1 < i_2 \)
    by blast

qed

lemma insert-ops-subset-last:

assumes insert-ops \((xs @ [x])\)
and insert-ops \(ys\)
and \(\text{set \ ys} \subseteq \text{set} \ (xs @ [x])\)
and \(x \in \text{set \ ys}\)
shows \(x = \text{last \ ys}\)
using assms proof (induction \(ys\), simp)

next

case \(ys\)-nonempty: False
have \(x \neq y\)
proof –
    obtain \(mid \ l\) where \(ys = mid @ [l]\)
    using append-butlast-last-id \(ys\)-nonempty by metis

moreover obtain \(li \ lr\) where \(l = (li, lr)\)
    by force

moreover have \(\forall i. \ i \in \text{set} \ (\text{map \ fst \ (y \# \ mid)}) \implies i < li\)
    by (metis last-op-greatest assms(2) calculation append-Cons)

hence \(\text{fst \ y} < li\)
    by simp

moreover have \(\forall i. \ i \in \text{set} \ (\text{map \ fst \ xs}) \implies i < \text{fst} \ x\)
    using assms(1) last-op-greatest by (metis prod.collapse)

hence \(\forall i. \ i \in \text{set} \ (\text{map \ fst \ (y \# \ ys)}) \implies i \leq \text{fst} \ x\)
    using assms(3) by fastforce

ultimately show \(x \neq y\)
    by fastforce

qed

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

qed

qed

lemma subset-butlast:
\textbf{assumes} \( \text{set } xs \subseteq \text{set } (ys @ [y]) \)

\text{and} \ last \ xs = y

\text{and} \ distinct \ xs

\textbf{shows} \( \text{set } (\text{butlast } xs) \subseteq \text{set } ys \)

\textbf{using} \assms \textbf{by} (\text{induction } xs, \text{auto})

\textbf{lemma} \ distinct-append-butlast1:

\textbf{assumes} \ distinct \ (map \ fst \ xs @ map \ fst \ ys)

\textbf{shows} \ distinct \ (map \ fst \ (butlast \ xs) @ map \ fst \ ys)

\textbf{using} \assms \textbf{proof}(\text{induction } xs, \text{simp})

\textbf{case} \ (Cons \ a \ xs)

\textbf{have} \ \( \text{fst } a \notin \text{set } (map \ fst \ xs @ map \ fst \ ys) \)

\textbf{using} \ Cons.\assms \textbf{by} \text{auto}

\textbf{moreover have} \ \( \text{set } (map \ fst \ (\text{butlast } xs)) \subseteq \text{set } (map \ fst \ xs) \)

\textbf{by} (metis \ in-set-butlastD \ map-butlast \ subsetI)

\textbf{hence} \ \( \text{set } (map \ fst \ (\text{butlast } xs) @ map \ fst \ ys) \subseteq \text{set } (map \ fst \ xs @ map \ fst \ ys) \)

\textbf{by} \text{auto}

\textbf{ultimately have} \ \( \text{fst } a \notin \text{set } (map \ fst \ (butlast \ xs) @ map \ fst \ ys) \)

\textbf{by} \text{blast}

\textbf{then show} \ distinct \ (map \ fst \ (\text{butlast } (a \# xs)) @ map \ fst \ ys)

\textbf{using} \ Cons.\IH \assms \textbf{by} \text{auto}

\textbf{qed}

\textbf{lemma} \ distinct-append-butlast2:

\textbf{assumes} \ distinct \ (map \ fst \ xs @ map \ fst \ ys)

\textbf{shows} \ distinct \ (map \ fst \ xs @ map \ fst \ (butlast \ ys))

\textbf{using} \assms \textbf{proof}(\text{induction } xs)

\textbf{case} \ Nil

\textbf{then show} \ distinct \ (map \ fst \ [] @ map \ fst \ (\text{butlast } ys))

\textbf{by} \text{simp add: distinguished-butlast map-butlast}

\textbf{next}

\textbf{case} \ (Cons \ a \ xs)

\textbf{have} \ \( \text{fst } a \notin \text{set } (map \ fst \ xs @ map \ fst \ ys) \)

\textbf{using} \ Cons.\assms \textbf{by} \text{auto}

\textbf{moreover have} \ \( \text{set } (map \ fst \ (\text{butlast } ys)) \subseteq \text{set } (map \ fst \ ys) \)

\textbf{by} (metis \ in-set-butlastD \ map-butlast \ subsetI)

\textbf{hence} \ \( \text{set } (map \ fst \ xs @ map \ fst \ (\text{butlast } ys)) \subseteq \text{set } (map \ fst \ xs @ map \ fst \ ys) \)

\textbf{by} \text{auto}

\textbf{ultimately have} \ \( \text{fst } a \notin \text{set } (map \ fst \ xs @ map \ fst \ (\text{butlast } ys)) \)

\textbf{by} \text{blast}

\textbf{then show} \ ?case

\textbf{using} \ Cons.\IH \assms \textbf{by} \text{auto}

\textbf{qed}

\textbf{4.2 Lemmas about interp-ins}

\textbf{lemma} \ interp-ins-maybe-grow:

\textbf{assumes} \ insert-ops \ (xs @ [(oid, ref)])

\textbf{shows} \ \( \text{set } (\text{interp-ins } (\text{xs } @ [(\text{oid, ref)}])) = \text{set } (\text{interp-ins } \text{xs}) \lor \)

58
set (interp-ins (xs @ [(oid, ref)])) = (set (interp-ins xs) ∪ {oid})

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

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

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

lemma interp-ins-ref-nonex:
assumes insert-ops ops
and ops = xs @ [(oid, Some ref)] @ ys
and ref /∈ set (interp-ins xs)
shows oid /∈ set (interp-ins ops)
using assms proof (induction ys arbitrary: ops rule: List.rev-induct)
case Nil
then have interp-ins ops = insert-spec (interp-ins xs) (oid, Some ref)
  by (simp add: interp-ins-tail-unfold)
moreover have \( \forall i. \ i \in \text{set (map fst xs)} \Rightarrow i < \text{oid} \)
  using Nil.prems last-op-greatest by fastforce
hence \( \forall i. \ i \in \text{set (interp-ins xs)} \Rightarrow i < \text{oid} \)
  by (meson interp-ins-subset subsetCE)
ultimately show \( \text{oid} \notin \text{set (interp-ins ops)} \)
  using assms(3) by auto

next
case (snoc x ys)
then have insert-ops (xs @ (oid, Some ref) \# ys)
  by (metis append.assoc append.simps(1) append-Cons insert-ops-appendD)
hence IH: \( \text{oid} \notin \text{set (interp-ins (xs @ (oid, Some ref) \# ys))} \)
  by (simp add: snoc.IH snoc.prems)
moreover have distinct (map fst (xs @ (oid, Some ref) \# ys @ [x]))
  using snoc.prems by (metis append-Cons append-self-conv2 insert-ops-def spec-ops-def)
hence \( \text{fst x} \neq \text{oid} \)
  using empty-iff by auto
moreover have insert-ops ((xs @ (oid, Some ref) \# ys) @ [x])
  using snoc.prems by auto
hence set (interp-ins ((xs @ (oid, Some ref) \# ys) @ [x])) =
  set (interp-ins (xs @ (oid, Some ref) \# ys)) \cup \{\text{fst x}\}
  using interp-ins-maybe-grow2 by blast
ultimately show \( \text{oid} \notin \text{set (interp-ins ops)} \)
  using snoc.prems(2) by auto
qed

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

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

lemma interp-ins-append-non-memb:
  assumes insert-ops (pre \# [(\text{oid}, \text{Some ref})] \@ suf)
  and ref \notin \text{set (interp-ins pre)}
  shows \( \text{ref} \notin \text{set (interp-ins (pre @ [(\text{oid}, \text{Some ref})]) \@ suf)} \)
  using assms proof(induction suf rule: List.rev-induct)
case Nil
then show \( \text{ref} \notin \text{set (interp-ins (pre @ [(\text{oid}, \text{Some ref})]) \@ [])} \)
  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])
  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 \in set (interp-ins pre)
  shows oid \in set (interp-ins (pre @ [(oid, Some ref)] @ suf))
  using assms by (metis UnCI append-assoc insert-spec-set interp-ins-monotonic interp-ins-tail-unfold singletonI)

lemma interp-ins-append-forward:
  assumes insert-ops (xs @ ys)
  and oid \in set (interp-ins (xs @ ys))
  and oid \in set (map fst xs)
  shows oid \in set (interp-ins xs)
  using assms proof (induction ys rule: List.rev-induct, simp)
  case (snoc y ys)
    obtain cs ds ref where xs = cs @ (oid, ref) \# ds
    by (metis (no-types, lifting) imageE prod.collapse set-map snoc.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 \neq 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 \in set (interp-ins (xs @ [(oid, Some ref)] @ ys))
  shows \exists r. (ref, r) \in set xs
  proof
    have ref < oid
using assms(1) insert-ops-ref-older by blast
have ref ∈ set (map fst (xs @ [(oid, Some ref)] @ ys))
  by (meson assms(2) interp-ins-subset subsetCE)
then obtain x where x-prop: x ∈ set (xs @ [(oid, Some ref)] @ ys) ∧ fst x = ref
  by fastforce
obtain xr where x-pair: x = (ref, xr)
  using prod.exhaust-set x-prop by blast
show (∃r. (ref, r) ∈ set xs)
  case True
  then show (∃r. (ref, r) ∈ set xs)
  by (metis x-prop prod.collapse)
next
  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 (meson 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
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 \( \text{interp-ins \ ops} = xs @ [\text{ref}] @ ys \)
using assms(2) split-list-first by fastforce
hence \( \text{insert-spec (interp-ins \ ops)} (\text{oid}, \text{Some \ ref}) = xs @ [\text{ref}] @ [] @ [\text{oid}] @ ys \)
using assms(1) insert-after-ref interp-ins-distinct by fastforce
ultimately show \( \text{list-order} \ (\text{ops} @ [(\text{oid}, \text{Some \ ref})]) \ \text{ref} \ \text{oid} \)
using assms(1) list-orderI by metis
qed

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

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

63
4.4 The insert-seq predicate

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

\[
\text{inductive insert-seq :: } ('oid option \Rightarrow ('oid \times 'oid option) \text{ list } \Rightarrow \text{ bool where }}
\]

\[
\begin{align*}
\text{insert-seq start } & \text{[[oid, start]]} \mid \\
\text{insert-seq start } & \text{((list \@ [[prev, ref]])\text{)}} \\
\implies & \text{insert-seq start } \text{((list \@ [[prev, ref]], (oid, Some prev)))}
\end{align*}
\]

lemma insert-seq-nonempty:

assumes insert-seq start xs

shows \(xs \neq []\)

using assms by (induction rule: insert-seq.induct, auto)

lemma insert-seq-hd:

assumes insert-seq start xs

shows \(\exists oid. \text{hd } xs = (oid, start)\)

using assms by (induction rule: insert-seq.induct, simp, metis append-self-conv2 hd-append2 list.sel(1))

lemma insert-seq-rem-last:

assumes insert-seq start (xs @ [x])

and \(xs \neq []\)

shows insert-seq start xs

using assms insert-seq.cases by fastforce

lemma insert-seq-butlast:

assumes insert-seq start xs

and \(xs \neq []\) and \(xs \neq [\text{last } xs]\)

shows insert-seq start (butlast xs)

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

lemma insert-seq-last-ref:

assumes insert-seq start (xs @ [[xi, xr], (yi, yr)])

shows yr = Some xi

using assms insert-seq.cases by fastforce

lemma insert-seq-start-none:
assumes \( \text{insert-ops} \ \text{ops} \)
\( \text{and} \ \text{insert-seq} \ None \ \text{xs} \ \text{and} \ \text{insert-ops} \ \text{xs} \)
\( \text{and} \ \text{set} \ \text{xs} \subseteq \text{set} \ \text{ops} \)
shows \( \forall i \in \text{set} \ (\text{map} \ \text{fst} \ \text{xs}). \ i \in \text{set} \ (\text{interp-ins} \ \text{ops}) \)
using \( \text{assms} \ \text{proof}(\text{induction} \ \text{xs} \ \text{rule}: \text{List.rev-induct}, \ \text{simp}) \)
case \( \text{snoc} \ \text{x} \ \text{xs} \)
then have \( \text{IH}: \forall i \in \text{set} \ (\text{map} \ \text{fst} \ \text{xs}), \ i \in \text{set} \ (\text{interp-ins} \ \text{ops}) \)
by \( (\text{metis} \ \text{Nil-is-map-conv} \ \text{append-is-Nil-conv} \ \text{insert-ops-append} \ \text{insert-seq-rem-last} \ \text{le-supE} \ \text{list.simps}(3) \ \text{set-append} \ \text{split-list}) \)
then show \( \forall i \in \text{set} \ (\text{map} \ \text{fst} \ (\text{xs} @ [x])), \ i \in \text{set} \ (\text{interp-ins} \ \text{ops}) \)
proof \( (\text{cases} \ \text{xs} = []) \)
case \( \text{True} \)
then obtain \( \text{oid} \ \text{where} \ \text{xs} @ [x] = [(\text{oid}, \ \text{None})] \)
using \( \text{insert-seq-hd} \ \text{snoc.prems}(2) \ \text{by fastforce} \)
hence \( (\text{oid}, \ \text{None}) \in \text{set} \ \text{ops} \)
using \( \text{snoc.prems}(4) \ \text{by auto} \)
then obtain \( \text{as} \ \text{bs} \ \text{where} \ \text{ops} = \text{as} @ (\text{oid}, \ \text{None}) \neq \text{bs} \)
by \( (\text{meson} \ \text{split-list}) \)
hence \( \text{ops} = (\text{as} @ [(\text{oid}, \ \text{None})]) @ \text{bs} \)
by \( (\text{simp add:} \ \text{ops} = \text{as} @ (\text{oid}, \ \text{None}) \neq \text{bs}) \)
moreover have \( \text{oid} \in \text{set} \ (\text{interp-ins} \ (\text{as} @ [(\text{oid}, \ \text{None})])) \)
by \( (\text{simp add:} \ \text{interp-ins-last-None}) \)
ultimately have \( \text{oid} \in \text{set} \ (\text{interp-ins} \ \text{ops}) \)
using \( \text{interp-ins-monotonic} \ \text{snoc.prems}(1) \ \text{by blast} \)
then show \( \forall i \in \text{set} \ (\text{map} \ \text{fst} \ (\text{xs} @ [x])), \ i \in \text{set} \ (\text{interp-ins} \ \text{ops}) \)
using \( \text{xs} @ [x] = [(\text{oid}, \ \text{None})] \ \text{by auto} \)
next
case \( \text{False} \)
then obtain \( \text{rest} \ \text{y} \ \text{where} \ \text{snoc-y:} \ \text{xs} = \text{rest} @ [y] \)
using \( \text{append-buttlast-last-id} \ \text{by metis} \)
obtain \( \text{yi} \ \text{yr} \ \text{xi} \ \text{xr} \ \text{where} \ \text{yx-pairs:} \ \text{y} = (\text{yi}, \ \text{yr}) \land \text{x} = (\text{xi}, \ \text{xr}) \)
by \( \text{force} \)
then have \( \text{xr} = \text{Some} \ \text{yi} \)
using \( \text{insert-seq-last-ref} \ \text{snoc.prems}(2) \ \text{snoc-y} \ \text{by fastforce} \)
have \( \text{yi} < \text{xi} \)
using \( \text{insert-ops-sorted-oids} \ \text{snoc-y} \ \text{yx-pairs} \ \text{snoc.prems}(3) \)
by \( (\text{metis} \ (\text{no-types}, \ \text{lifting}) \ \text{append-eq-append-conv2}) \)
have \( (\text{yi}, \ \text{yr}) \in \text{set} \ \text{ops} \ \text{and} \ (\text{xi}, \ \text{Some} \ \text{yi}) \in \text{set} \ \text{ops} \)
using \( \text{snoc.prems}(4) \ \text{snoc-y} \ \text{yx-pairs} \ \text{xr} = \text{Some} \ \text{yi} \ \text{by auto} \)
then obtain \( \text{as} \ \text{bs} \ \text{cs} \ \text{where} \ \text{ops-split:} \ \text{ops} = \text{as} @ [(\text{yi}, \ \text{yr})] @ \text{bs} @ [(\text{xi}, \ \text{Some} \ \text{yi})] @ \text{cs} \)
using \( \text{insert-ops-split-2} \ (\text{yi} < \text{xi}) \ \text{snoc.prems}(1) \ \text{by blast} \)
hence \( \text{yi} \in \text{set} \ (\text{interp-ins} \ (\text{as} @ [(\text{yi}, \ \text{yr})] @ \text{cs})) \)
proof –
have \( \text{yi} \in \text{set} \ (\text{interp-ins} \ \text{ops}) \)
by \( (\text{simp add:} \ \text{IH} \ \text{snoc-y} \ \text{yx-pairs}) \)
moreover have \( \text{ops} = (\text{as} @ [(\text{yi}, \ \text{yr})] @ \text{bs}) @ [(\text{xi}, \ \text{Some} \ \text{yi})] @ \text{cs} \)
using \( \text{ops-split} \ \text{by simp} \)
moreover have \( \text{yi} \in \text{set} \ (\text{map} \ \text{fst} \ (\text{as} @ [(\text{yi}, \ \text{yr})] @ \text{bs})) \)

65
by simp
ultimately show \(?thesis
using snoc.prems(1) interp-ins-append-forward by blast
qed

hence \(\xi \in \text{set (interp-ins \(((a@[[y_1, y_r]]@bs)@[\langle x_i, \text{Some } y_i \rangle]@cs))\})\)
using snoc.prems(1) interp-ins-append-memb ops-split by force
hence \(\xi \in \text{set (interp-ins ops)}\)
by (simp add: ops-split)
then show \(\forall i \in \text{set (map \text{fst } (xs@[x]))}. \ i \in \text{set (interp-ins ops)}\)
using IH yx-pairs by auto
qed
qed

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

case (snoc x xs)

have IH: \(\forall i \in \text{set (map \text{fst } xs)}. \ \text{list-order ops ref i}\)
using snoc.IH snoc.prems insert-seq-rem-last insert-ops-appendD
by (metis Nil-is-map-conv Un-subset-iff empty-set equals0D set-append)

moreover have list-order ops ref (fst x)
proof (cases xs = [])

case True

using insert-seq-hd snoc.prems(2) by fastforce
moreover have x \(\in\) set ops
using snoc.prems(4) by auto
then obtain cs ds where x-split: ops = cs@x#ds
by (meson split-list)

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

have insert-ops (cs@[\langle \text{fst } x, \text{Some ref} \rangle]@ds)
using x-split (snd x = Some ref)
by (metis append-Cons append-self-conv2 prod.collapse snoc.prems(1))

moreover from this obtain rr where (ref, rr) \(\in\) set cs
using interp-ins-find-ref x-split (snd x = Some ref) assms(5)
by (metis (no-types, lifting) append-Cons append-self-conv2 prod.collapse)

hence ref \(\in\) set (interp-ins cs)

proof –

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

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

thus ref \(\in\) set (interp-ins cs)

using assms(5) calculation interp-ins-append-forward interp-ins-append-non-memb
by blast
ultimately show list-order (cs @ [(fst x, Some ref)]) ref (fst x)
using list-order-insert-ref by (metis append.assoc insert-ops-appendD)

moreover have ops = (cs @ [(fst x, Some ref)]) @ ds
using calculation x-split
by (metis append-eq-Cons-conv append-eq-append-conv2 append-self-conv2 prod.collapse)

ultimately show list-order ops ref (fst x).
using list-order-append snoc.prems(1) by blast

next

case False
then obtain rest y where snoc-y: xs = rest @ [y]
using append-butlast-last-id by metis

obtain yi yr xi xr where yx-pairs: y = (yi, yr) ∧ x = (xi, xr)
by force

then have xr = Some yi
using insert-seq-last-ref snoc.prems(2) snoc-y by fastforce

have yi < xi
using insert-ops-sorted-oids snoc-y yx-pairs snoc.prems(3)
by (metis (no-types, lifting) append-eq-append-conv2)

have (yi, yr) ∈ set ops and (xi, Some yi) ∈ set ops
using snoc.prems(4) snoc-y yx-pairs (xr = Some yi) by auto
then obtain as bs cs where ops-split: ops = as @ [(yi, yr)] @ bs @ [(xi, Some yi)] @ cs
using insert-ops-split-2 (yi < xi) snoc.prems(1) by blast

have list-order ops ref yi
by (simp add: IH snoc-y yx-pairs)

moreover have list-order (as @ [(yi, yr)] @ bs @ [(xi, Some yi)]) yi xi
proof −

have insert-ops ((as @ [(yi, yr)] @ bs @ [(xi, Some yi)]) @ cs)
using ops-split snoc.prems(1) by auto
hence insert-ops ((as @ [(yi, yr)] @ bs) @ [(xi, Some yi)])
using insert-ops-appendD by fastforce

moreover have yi ∈ set (interp-ins ops)
using /list-order ops ref yi: list-order-memb2 by auto
hence yi ∈ set (interp-ins (as @ [(yi, yr)] @ bs))
using interp-ins-append-non-memb ops-split snoc.prems(1) by force

ultimately show ?thesis
using list-order-insert-ref by force

qed

hence list-order ops yi xi
by (metis append.assoc list-order-append ops-split snoc.prems(1))

ultimately show list-order ops ref (fst x)
using list-order-trans snoc.prems(1) yx-pairs by auto

qed

ultimately show ∀ i ∈ set (map fst (xs @ [x])). list-order ops ref i
by auto

qed
lemma insert-seq-no-start:

assumes insert-ops ops
and insert-seq (Some ref) xs and insert-ops xs
and set xs ⊆ set ops
and ref /∈ set (interp-ins ops)
shows ∀ i ∈ set (map fst xs), i /∈ set (interp-ins ops)
using assms proof(induction xs rule: List.rev-induct, simp)
case (snoc x xs)
  have IH: ∀ i ∈ set (map fst xs), i /∈ set (interp-ins ops)
    using snoc.IH snoc.prems insert-seq-rem-last insert-ops-appendD
    by (metis append-is-Nil-conv le-sup-iff list.map-disc-iff set-append split-list-first)
  obtain as bs where ops = as @ x # bs
    using snoc.prems(4) by (metis split-list last-in-set snoc-eq-iff-butlast subset-code(1))
  have fst x /∈ set (interp-ins ops)
    proof(cases xs = [])
      case True
      then obtain xi where x = (xi, Some ref)
        using insert-seq-hd snoc.prems(2) by force
      moreover have ref /∈ set (interp-ins as)
        using interp-ins-monotonic snoc.prems(1) snoc.prems(5) \ops = as @ x # bs
        by blast
      ultimately have xi /∈ set (interp-ins (as @ [x] @ bs))
        using snoc.prems(1) by (simp add: interp-ins-ref-nonex \ops = as @ x # bs)
      then show fst x /∈ set (interp-ins ops)
        by (simp add: \ops = as @ x # bs, x = (xi, Some ref))
    next
    case xs-nonempty: False
    then obtain y where xs = (butlast xs) @ [y]
      by (metis append-butlast-last-id)
    moreover from this obtain yi yr xi xr where y = (yi, yr) ∧ x = (xi, xr)
      by fastforce
    moreover from this have xr = Some yi
      using insert-seq.cases snoc.prems(2) calculation by fastforce
    moreover have yi /∈ set (interp-ins ops)
      using IH calculation
      by (metis Nil-is-map-conv fst-conv last-in-set last-map snoc-eq-iff-butlast)
    hence yi /∈ set (interp-ins as)
      using \ops = as @ x # bs interp-ins-monotonic snoc.prems(1) by blast
    ultimately have xi /∈ set (interp-ins (as @ [x] @ bs))
      using interp-ins-ref-nonex snoc.prems(1) \ops = as @ x # bs
      by fastforce
    then show fst x /∈ set (interp-ins ops)
      by (simp add: \ops = as @ x # bs, y = (yi, yr) ∧ x = (xi, xr))
    qed
  qed
  then show ∀ i ∈ set (map fst (xs @ [x])), i /∈ set (interp-ins ops)
    using IH by auto
qed
4.5 The proof of no interleaving

lemma no-interleaving-ordered:
assumes insert-ops ops
and insert-seq start xs and insert-ops xs
and insert-seq start ys and insert-ops ys
and set xs ⊆ set ops and set ys ⊆ set ops
and distinct (map fst xs @ map fst ys)
and fst (hd xs) < fst (hd ys)
and \( \forall r. \) start = Some r \( \implies r \in \) set (interp-ins ops)
shows \( \forall x \in \) set (map fst xs), \( \forall y \in \) set (map fst ys). list-order ops y x \( \land \)
(\( \forall r. \) start = Some r \( \implies \) \( \forall x \in \) set (map fst xs). list-order ops r x) \( \land \)
(\( \forall y \in \) set (map fst ys). list-order ops r 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 \( \in \) set xs \( \mid \) (a-in-ys) a \( \in \) set ys \( \mid \) (neither) a \( \notin \) set xs \( \land \) a \( \notin \) set ys
by blast
then show ?case
proof (cases)
case a-in-xs
then have a \( \notin \) set ys
using snoc.prems(8) by auto
hence set ys \( \subseteq \) 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: \( \forall x \in \) set (map fst (butlast xs)). \( \forall y \in \) set (map fst ys). list-order ops y x \( \land \)
(\( \forall r. \) start = Some r \( \implies \) \( \forall x \in \) set (map fst (butlast xs)). list-order ops r x) \( \land \)
(\( \forall y \in \) set (map fst ys). list-order ops r y))
proof (cases xs = [a])
case True
moreover have \( \forall r. \) start = Some r \( \implies \) \( \forall y \in \) set (map fst ys). list-order ops r y)
using insert-seq-after-start (insert-ops ops) (set ys \( \subseteq \) set ops) snoc.prems
by (metis append-Nil2 calculation insert-seq-hd interp-ins-append-non-memb list.sel(1))
ultimately show ?thesis by auto
next
case xs-longer: False
from (a = last xs) have set (butlast xs) \( \subseteq \) 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 \text{insert-ops} (butlast \text{xs})
  using \text{snoc.prem}(2) \text{snoc.prem}(3) \text{insert-ops-appendD}
  by (\text{metis append-butlast-last-id insert-seq-nonempty})
moreover have \text{distinct} (\text{map \text{fst} (butlast \text{xs}) \& map \text{fst} \text{ys})
  using \text{distinct-append-butlast1} \text{snoc.prem}(8) \text{by blast}
moreover have \text{set} \text{ys} \subseteq \text{set} \text{ops}
  using \text{distinct-append-butlast1} \text{snoc.prem}(8) \text{by blast}
moreover have hd (butlast \text{xs}) = hd \text{xs}
by (\text{metis hd-append2 insert-seq-nonempty snoc.prem}(2))
hence \text{fst} (hd (butlast \text{xs})) < \text{fst} (hd \text{ys})
by (simp add: \text{snoc.prem}(9))
moreover have \text{⋀} r. \text{start} = \text{Some} r \Rightarrow r \in \text{set} (\text{interp-ins} \text{ops})
proof –
  fix r
  assume \text{start} = \text{Some} r
  then obtain \text{xid} where \text{hd} \text{xs} = (\text{xid}, \text{Some} r)
  using \text{insert-seq-hd} \text{snoc.prem}(2) \text{by auto}
  hence r < \text{xid}
  by (\text{metis hd-in-set insert-ops-memb-ref-older insert-seq-nonempty snoc.prem}(2))
moreover have \text{distinct} (\text{map \text{fst} (butlast \text{xs}) \& map \text{fst} \text{ys})
  using \text{distinct-append-butlast1} \text{snoc.prem}(8) \text{by blast}
moreover have \text{⋀} \text{xid} \in \text{set } (\text{map \text{fst} (butlast \text{xs})})
proof –
  have \text{xs} = (\text{butlast} \text{xs}) \@ \text{[a]}
    using \text{snoc.prem}(2) \text{insert-seq-nonempty} \text{a = last} \text{xs} \text{by fastforce}
  moreover have (\text{xid}, \text{Some} r) \in \text{set} (\text{butlast} \text{xs})
    using (\text{hd} \text{xs} = (\text{xid}, \text{Some} r)) \text{insert-seq-nonempty list.set-sel}(1)
  hence \text{xid} \in \text{set} (\text{map \text{fst} (butlast \text{xs})})
    by (\text{metis \text{hd-in-set} insert-ops-memb-ref-older insert-seq-nonempty snoc.prem}(2))
  ultimately show \text{?thesis}
    using \text{snoc.prem}(3) \text{last-op-greatest} \text{by} (\text{metis \text{prod.collapse}})
  qed
ultimately have \text{r} \neq \text{fst} \text{a}
  using \text{dual-order.asym} \text{by blast}
thus \text{r} \in \text{set} (\text{interp-ins} \text{ops})
using \text{snoc.prem}(1) \text{snoc.prem}(10) \text{interp-ins-maybe-grow2} \text{start = Some} \text{r} \text{by blast}
qed
ultimately show \text{?thesis}
  using (\text{insert-ops} \text{ops}) \text{snoc.IH} \text{snoc.prem}(4) \text{snoc.prem}(5) \text{by blast}
qed
moreover have \text{x-exists}: \forall \text{x} \in \text{set} (\text{map \text{fst} (butlast \text{xs})}) \text{. \text{x} \in \text{set} (\text{interp-ins} \text{ops})}
proof(cases \text{start})
case None
moreover have \text{set} (\text{butlast} \text{xs}) \subseteq \text{set} \text{ops}
  using (\text{a = last} \text{xs}) \text{distinct-map} \text{snoc.prem}(6) \text{snoc.prem}(8) \text{subset-butlast}
ultimately show \( \text{thesis} \)

using \langle \text{insert-ops \ ops} \rangle \ \text{snoc.prems}

by \text{metis append-butlast-last-id butlast.simps(2) empty-iff empty-set}

\langle \text{insert-ops-rem-last insert-seq-butlast insert-seq-nonempty list.simps(8)} \rangle

next

case (Some \( a \))
then show \( \text{thesis} \)

using \( \text{IH list-order-memb2} \) by blast

qed

moreover have \( \forall y \in \text{set (map \ fst \ ys)} . \text{list-order (ops @ [a])} \ y \ (\text{fst} \ a) \)

proof (cases \( xs = [a] \))

case \( xs-a: True \)

have \( ys \neq [] \Rightarrow False \)

proof
−
assume \( ys \neq [] \)

then obtain \( yi \) where \( yi-def: ys = (yi, \text{start}) \neq (tl \ ys) \)

by \text{metis hd-Cons-tl insert-seq-hd snoc.prems(4)}

hence \( (yi, \text{start}) \in \text{set ops} \)

by \text{metis \langle set \ ys \subseteq set \ ops \rangle list.set-intros(1) subsetCE}

hence \( yi \in \text{set (map \ fst \ ops)} \)

by force

hence \( yi < \text{fst} \ a \)

using \text{snoc.prems(1) last-op-greatest by (metis prod.collapse)}

moreover have \( \text{fst} \ a < yi \)

by \text{metis \( yi-def \ \text{xs-a} \ \text{fst-conv list.sel(1) snoc.prems(9)} \)}

ultimately show False

using \text{less-not-sym by blast}

qed

then show \( \forall y \in \text{set (map \ fst \ ys)} . \text{list-order (ops @ [a])} \ y \ (\text{fst} \ a) \)

using \text{insert-seq-nonempty snoc.prems(4) by blast}

next

case \( xs-longer: False \)

hence \text{butlast-split: butlast \( xs = (\text{butlast (butlast} \ xs)) \ @ [last (butlast} \ xs)] \}

using \( a = \text{last} \ xs \) \text{insert-seq-butlast insert-seq-nonempty snoc.prems(2)} by fastforce

hence \text{ref-exists: \( \text{fst} \ (last (butlast} \ xs)) \in \text{set interp-ins ops} \}

using \text{x-exists by (metis last-in-set last-map map-is-Nil-conv snoc-eq-iff-butlast)}

moreover from \text{butlast-split have} \( xs = (\text{butlast (butlast} \ xs)) \ @ [last (butlast \ xs), a] \)

by \text{metis \( a = \text{last} \ xs \) append.assoc append-butlast-last-id butlast.simps(2)}

\langle \text{insert-ops-rem-last insert-seq-butlast insert-seq-nonempty list.simps(2) snoc.prems(2)} \rangle

hence \( \text{snd} \ a = \text{Some (fst (last (butlast} \ xs))} \)

using \text{snoc.prems(2) insert-seq-last-ref by (metis prod.collapse)}

hence \text{list-order (ops @ [a])} \ (\text{fst (last (butlast} \ xs))) \ (\text{fst} \ a)

using \text{list-order-insert-ref ref-exists snoc.prems(1) by (metis prod.collapse)}

moreover have \( \forall y \in \text{set (map \ fst \ ys)} . \text{list-order ops} \ y \ (\text{fst (last (butlast} \ xs)) \)

by \text{metis IH butlast-split last-in-set last-map map-is-Nil-conv snoc-eq-iff-butlast}

hence \( \forall y \in \text{set (map \ fst \ ys)} . \text{list-order (ops @ [a])} \ y \ (\text{fst (last (butlast} \ xs)) \)

71
using list-order-append snoc.prems(1) by blast
ultimately show \( \forall y \in \text{set } (\text{map } \text{fst } \text{ys}). \text{list-order } (\text{ops } @ [a]) y (\text{fst } a) \)
using list-order-trans snoc.prems(1) by blast
qed
moreover have map-fst-xs: map \( \text{fst } \text{xs} = \text{map } \text{fst } (\text{butlast } \text{xs}) @ \text{map } \text{fst } [\text{last } \text{xs}] \)
by (metis append-butlast-last-id insert-seq-nonempty map-append snoc.prems(2))
moreover have \( \forall r. \text{start } = \text{Some } r \longrightarrow \text{list-order } (\text{ops } @ [a]) r (\text{fst } a) \)
using snoc.prems by (cases start, auto simp add: insert-seq-after-start \( a = \text{last } \text{xs}; \text{map-fst-xs } \))
ultimately show \( \forall x \in \text{set } (\text{map } \text{fst } \text{xs}). \forall y \in \text{set } (\text{map } \text{fst } \text{ys}). \text{list-order } (\text{ops } @ [a]) y x \)
by blast

proof (cases ys = [a])
case True
moreover have \( \forall r. \text{start } = \text{Some } r \longrightarrow (\forall y \in \text{set } (\text{map } \text{fst } \text{xs}). \text{list-order } \text{ops } y x) \)
by (metis append-nil2 calculation insert-seq-hd interp-ins-append-non-memb list.sel(1))
ultimately show \( \text{thesis } \)
next
case \( \text{ys-longer}; \text{False} \)
from \( a = \text{last } \text{ys} \) have set (butlast \( \text{ys} \)) \( \subseteq \text{set } \text{ops} \)
using snoc.prems by (simp add: distinct-fst subset-butlast)
moreover have insert-seq start (butlast \( \text{ys} \))
using insert-seq-butlast insert-seq-nonempty \( \text{ys-longer } a = \text{last } \text{ys}; \text{snoc.prems(4)} \)
bystart
moreover have insert-ops (butlast \( \text{ys} \))
using snoc.prems(4) snoc.prems(5) insert-ops-appendD
by (metis append-butlast-last-id insert-seq-nonempty)
moreover have distinct (map fst xs @ map fst (butlast ys))
  using distinct-append-butlast2 snoc.prems(8) by blast
moreover have set xs ⊆ set ops
  using (a ∉ set xs) (set xs ⊆ set ops) by blast
moreover have hd (butlast ys) = hd ys
  by (metis append-butlast-last-id calculation(2) hd-append2 insert-seq-nonempty
       snoc.prems(4))
  hence fst (hd xs) < fst (hd (butlast ys))
  by (simp add: snoc.prems(9))
moreover have \( r. \) start = Some r \( \implies r \in \{ \text{interp-ins ops} \} \)
  proof –
    fix r
    assume start = Some r
    then obtain yid where hd ys = (yid, Some r)
      using insert-seq-hd snoc.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 \( a = \text{last ys} \) by fastforce
    moreover have (yid, Some r) ∈ set (butlast ys)
      using \( \{ \text{hd ys} = (yid, \text{Some r}) \} \), insert-seq-nonempty list.set.sel(1)
      by (metis in-set-zipE zip-map-fst-snd)
    hence yid ∈ set (map fst (butlast ys))
    by (metis hd (butlast ys) = hd ys; \( \{ \text{insert-seq start (butlast ys)} \} \))
    hence \( yid \in \{ \text{set xs} \subseteq \{ \text{set ops} \} \} \)
    by (metis in-set-zipE zip-map-fst-snd)
    ultimately show \( ?\text{thesis} \)
      using snoc.prems(5) last-op-greatest by (metis prod.collapse)
    qed
    ultimately have \( r \neq \text{fst a} \)
    using dual-order.asym by blast
    thus \( r \in \{ \text{interp-ins ops} \} \)
    using snoc.prems(1) snoc.prems(10) interp-ins-maybe-grow2 (start = Some r) by blast
  qed
moreover have \( \forall x \in \{ \text{map fst xs} \}, \text{list-order (ops \( @ \) [a]) (fst a) x} \)
  proof(cases ys = [a])
    case ys-a: True
    then show \( \forall x \in \{ \text{map fst xs} \}, \text{list-order (ops \( @ \) [a]) (fst a) x} \)
      proof(cases start)
        case None
        then show \( ?\text{thesis} \)
          using insert-seq-start-none list-order-insert-none snoc.prems
          by (metis insert-ops ops \( \{ \text{set xs} \subseteq \{ \text{set ops} \} \} \),\ text{fst-conv insert-seq-hd list.sel}(1))
    qed
  qed
moreover have \( \forall x \in \{ \text{map fst xs} \}, \text{list-order (ops \( @ \) [a]) (fst a) x} \)
  proof(cases ys = [a])
    case ys-a: True
    then show \( \forall x \in \{ \text{map fst xs} \}, \text{list-order (ops \( @ \) [a]) (fst a) x} \)
      proof(cases start)
        case None
        then show \( ?\text{thesis} \)
          using insert-seq-start-none list-order-insert-none snoc.prems
          by (metis insert-ops ops \( \{ \text{set xs} \subseteq \{ \text{set ops} \} \} \),\ text{fst-conv insert-seq-hd list.sel}(1))
    qed
moreover have \( \forall x \in \{ \text{map fst xs} \}, \text{list-order (ops \( @ \) [a]) (fst a) x} \)
  proof(cases ys = [a])
    case ys-a: True
    then show \( \forall x \in \{ \text{map fst xs} \}, \text{list-order (ops \( @ \) [a]) (fst a) x} \)
      proof(cases start)
        case None
        then show \( ?\text{thesis} \)
          using insert-seq-start-none list-order-insert-none snoc.prems
          by (metis insert-ops ops \( \{ \text{set xs} \subseteq \{ \text{set ops} \} \} \),\ text{fst-conv insert-seq-hd list.sel}(1))
    qed
ys-a)

next
case (Some r)
  moreover from this have ∀ x ∈ set (map fst xs). list-order ops r x
  using IH by blast
ultimately show \( ?thesis \)
  using snoc.prems(1) snoc.prems(4) list-order-insert-between
  by (metis fst-conv insert-seq-hd list.sel(1) ys-a)
qed
next
case ys-longer: False
  hence butlast-split: butlast ys = (butlast (butlast ys)) @ [last (butlast ys)]
  using (a = last ys) insert-seq-butlast insert-seq-nonempty snoc.prems(4) by
fastforce
moreover from this have ys = (butlast (butlast ys)) @ [last (butlast ys)], a
  by (metis (a = last ys) append.assoc append-butlast-last-id butlast.simps(2)
    insert-seq-nonempty last-ConsL last-ConsR list.simps(3) snoc.prems(4))
  hence snd a = Some (fst (last (butlast ys))))
  using snoc.prems(4) insert-seq-last-ref by (metis prod.collapse)
mOREover have ∀ x ∈ set (map fst xs). list-order ops (fst (last (butlast ys))) x
  by (metis IH butlast-split last-in-set last-map map-is-Nil-conv snoc-eq-iff-butlast)
ultimately show ∀ x ∈ set (map fst xs). list-order (ops @ [a]) (fst a) x
  using list-order-insert-between snoc.prems(1) by (metis prod.collapse)
qed
moreover have map-fst-xs: map fst ys = map fst (butlast ys) @ map fst [last ys]
  by (metis append-butlast-last-id insert-seq-nonempty map-append snoc.prems(4))
  hence set (map fst ys) = set (map fst (butlast ys)) ∪ {fst a}
    by (simp add: a = last ys)
moreover have ∀ r. start = Some r → list-order (ops @ [a]) r (fst a)
  using snoc.prems by (cases start, auto simp add: insert-seq-after-start a =
    last ys: map-fst-xs)
ultimately show (∀ x ∈ set (map fst xs). ∀ y ∈ set (map fst ys). list-order (ops
    @ [a]) y x) ∨
  (∀ r. start = Some r → (∀ x ∈ set (map fst xs). list-order (ops @ [a]) r x) ∧
    (∀ y ∈ set (map fst ys). list-order (ops @ [a]) r y))
  using snoc.prems(1) by (simp add: list-order-append)
next
case neither
  hence set xs ⊆ set ops and set ys ⊆ set ops
    using snoc.prems(6) snoc.prems(7) DiffE by auto
have (∀ r. start = Some r → r ∈ set (interp-ins ops)) ∨ (xs = [] ∧ ys = [])
proof(cases xs)
case Nil
  then show \( ?thesis \) using insert-seq-nonempty snoc.prems(2) by blast
next
case xs-nonempty: (Cons x 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 (insert-ops ops) insert-ops-memb-ref-older by blast
  moreover have xi ∈ set (map fst ops)
  using ⟨(xi, Some r) ∈ set ops⟩ by force
  hence xi < fst a
  using last-op-greatest snoc.prems(1) by (metis prod.collapse)
  ultimately have fst a ≠ r
  using order.asym by blast
  then show r ∈ set (interp-ins ops)
  using snoc.prems(1) snoc.prems(10) interp-ins-maybe-grow2 (start = Some r)
  by blast
qed
then show ?thesis by blast
qed

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))
sows (∀ 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 \((\text{cases } \text{fst}\ (\text{hd}\ xs) < \text{fst}\ (\text{hd}\ ys))\)

\text{case } \text{True}

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

using \text{assms}(9) \text{ by blast}

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

using \text{assms}\ \text{no-interleaving-ordered}\ \text{by blast}

then show \(\text{?thesis}\ \text{by blast}\)

next

\text{case } \text{False}

hence \(\text{fst}\ (\text{hd}\ ys) < \text{fst}\ (\text{hd}\ xs)\)

using \text{assms}(2) \text{ assms}(4) \text{ insert-seq-nonempty distinct-fst-append}

by (\text{metis}\ \text{no-types},\ \text{lifting}\ \text{hd-in-set}\ \text{hd-map list.map-disc-iff map-append neqE})

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

using \text{assms}(9) \text{ by blast}

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

using \text{assms}\ \text{no-interleaving-ordered}\ \text{by blast}

then show \(\text{?thesis}\ \text{by blast}\)

qed

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

\text{theorem } \text{missing-start-no-insertion}:\

assumes \text{insert-ops ops}\ and \text{insert-seq} (\text{Some start})\ \text{xs}\ and \text{insert-ops xs}\ and \text{insert-seq} (\text{Some start})\ \text{ys}\ and \text{insert-ops ys}\ and \text{set xs} \subseteq \text{set ops}\ and \text{set ys} \subseteq \text{set ops}\ and \text{start} \not\in \text{set}\ (\text{interp-ins ops})\ shows \(\forall x \in \text{set}\ (\text{map } \text{fst}\ xs) \cup \text{set}\ (\text{map } \text{fst}\ ys).\ x \not\in \text{set}\ (\text{interp-ins ops})\)

using \text{assms}\ \text{insert-seq-no-start}\ \text{by} (\text{metis}\ \text{UnE})

\text{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].

\text{theory } \text{RGA}

\text{imports } \text{Insert-Spec}

begin

fun \text{insert-body} :: 'oid::\{linorder\} list \Rightarrow 'oid \Rightarrow 'oid\ \text{list} \text{where}

\text{insert-body} \text{[]} \Rightarrow \text{e} = [\text{e}]

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

fun insert-rga :: 'oid::{linorder} list ⇒ ('oid × 'oid option) ⇒ 'oid list where
insert-rga \(xs\) \((e, \text{None})\) = insert-body \(xs\) \(e\) |

insert-rga \([\]\) \((e, \text{Some } i)\) = \([\]\) |

insert-rga \((x \# xs)\) \((e, \text{Some } i)\) =
\((\text{if } x = i \text{ then } x \# \text{ insert-body } xs \ e \\text{ else } x \# \text{ insert-rga } xs \ (e, \text{Some } i))\)

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 \((\text{insert-body } xs \ e)\) = insert \(e\) \((\text{set } xs)\)
by (induction \(xs\), auto)

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

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

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

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

lemma insert-rga-nonexistent:
assumes \(i \notin \text{set } xs\)
shows insert-rga \((e, \text{Some } i)\) = \(xs\)
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 $\text{insert-rga (insert-rga } xs (e_1, \text{Some } i_1)) (e_2, \text{Some } i_2) =
\text{insert-rga (insert-rga } xs (e_2, \text{Some } i_2)) (e_1, \text{Some } i_1)$
using assms proof (induction $xs$, simp)
case $(\text{Cons } a \ \text{xs})$
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 $\text{insert-rga (insert-rga } xs (e_1, i_1)) (e_2, i_2) =
\text{insert-rga (insert-rga } xs (e_2, i_2)) (e_1, i_1)$
proof (cases $i_1$
  case $\text{None}$
  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 $\text{None}$
    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 $\text{True}$
    then show $?\text{thesis}$
    using assms some-r1 some-r2 by (simp add: insert-rga-Some-commutes)
next
  case $\text{False}$
  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 $\exists p \ s. \ \text{xs} = p @ s \land \text{insert-body } xs e = p @ e \ # \ s$
proof (induction $xs$, force)
case $(\text{Cons } a \ \text{xs})$
then obtain $p \ s$ where $\text{IH}: \ \text{xs} = p @ s \land \text{insert-body } xs e = p @ e \ # \ s$
  by blast
then show $\exists p \ s. \ a \ # \ \text{xs} = p @ s \land \text{insert-body } (a \ # \ \text{xs}) e = p @ e \ # \ s$

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

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

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

lemma insert-rga-preserves-order:
assumes \( i = \text{None } \lor (\exists i', i = \text{Some } i' \land i' \in \text{set } xs) \)
  and distinct \( xs \)
shows \( \exists \text{pre suf. } xs = \text{pre @ suf } \land \text{insert-rga } \text{xs (e, i)} = \text{pre @ e } \# \text{suf}\)
proof (cases \( i \))
case None

then show $\exists \text{ pre suf}. \, \text{xs} = \text{pre} \circ \text{suf} \land \text{insert-rga} \, \text{xs} (e, i) = \text{pre} \circ e \neq \text{suf}$

using $\text{insert-body-split}$ by auto

next

case $(\text{Some } r)$

moreover from this obtain $\text{as bs}$ where $\text{xs} = \text{as} \circ r \neq \text{bs} \land (\forall x \in \text{set as}. \, x \neq r)$

using assms(1) $\text{split-list-first}$ by fastforce

moreover have $\exists \text{ cs ds}. \, \text{bs} = \text{cs} \circ \text{ds} \land \text{insert-body} \, \text{bs} \, e = \text{cs} \circ e \neq \text{ds}$

by $(\text{simp add: insert-body-split})$

then obtain $\text{cs ds}$ where $\text{bs} = \text{cs} \circ \text{ds} \land \text{insert-body} \, \text{bs} \, e = \text{cs} \circ e \neq \text{ds}$

by blast

ultimately have $\text{xs} = (\text{as} \circ r \neq \text{cs}) \circ \text{ds} \land \text{insert-rga} \, \text{xs} (e, i) = (\text{as} \circ r \neq \text{cs}) \circ e \neq \text{ds}$

using $\text{insert-rga-after-ref}$ by fastforce

then show $\text{thesis}$ by blast

qed

5.2 Lemmas about the rga-ops predicate

definition $\text{rga-ops} :: (\text{'oid::{linorder} } \times \text{'oid option}) \text{ list } \Rightarrow \text{ bool}$ where $\text{rga-ops list} \equiv \text{crdt-ops list set-option}$

lemma $\text{rga-ops-rem-last}$:

assumes $\text{rga-ops (xs @ [x])}$

shows $\text{rga-ops xs}$

using assms $\text{crdt-ops-rem-last}$ $\text{rga-ops-def}$ by blast

lemma $\text{rga-ops-rem-penultimate}$:

assumes $\text{rga-ops (xs @ [(i1, r1), (i2, r2)])}$

and $\land r1 = \text{Some } r \Rightarrow r \neq i1$

shows $\text{rga-ops (xs @ [(i2, r2)])}$

using assms proof

have $\text{crdt-ops (xs @ [(i2, r2)])}$ set-option

using assms $\text{crdt-ops-rem-penultimate}$ $\text{rga-ops-def}$ by fastforce

thus $\text{rga-ops (xs @ [(i2, r2)])}$

by $(\text{simp add: rga-ops-def})$

qed

lemma $\text{rga-ops-ref-exists}$:

assumes $\text{rga-ops (pre @ (oid, Some ref) \# suf)}$

shows $\text{ref} \in \text{fst ' set pre}$

proof –

from assms have $\text{crdt-ops (pre @ (oid, Some ref) \# suf)}$ set-option

by $(\text{simp add: rga-ops-def})$

moreover have set-option $(\text{Some ref}) = \{\text{ref}\}$

by simp

ultimately show $\text{ref} \in \text{fst ' set pre}$

using $\text{crdt-ops-ref-exists}$ by fastforce

qed
5.3 Lemmas about the interp-rga function

lemma interp-rga-tail-unfold:
  shows interp-rga \( \langle x \rangle \) = 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 \( r \in \text{set (interp-rga xs)} \)
  using IH rga-ops-ref-exists by (metis x-pair list.set-map snoc.prems)
ultimately have set (interp-rga (xs @ [xi, xr])) = insert xi (set (interp-rga xs))
  by (simp add: insert-rga-set-ins interp-rga-tail-unfold)
then show set (interp-rga (xs @ [x])) = set (map fst (xs @ [x]))
  using IH x-pair by auto
qed

lemma interp-rga-distinct:
  assumes rga-ops xs
  shows distinct (interp-rga xs)
using assms proof (induction xs rule: List.rev-induct)
case Nil
  then show distinct (interp-rga []) by (simp add: interp-rga-def)
next
case (snoc x xs)
hence IH: distinct (interp-rga xs)
  using rga-ops-rem-last by blast
moreover obtain xi xr where x-pair: \( x = (xi, xr) \) by force
moreover from this have xi \( \notin \) set (interp-rga xs)
  using interp-rga-ids crdt-ops-unique-last rga-ops-rem-last

qed
by (metis rga-ops-def snoc.prems)
moreover have \( \exists \) \text{pre suf}. interp-rga \( \text{xs} \) = \text{pre@\text{suf}} \land
interp-rga (interp-rga \( \text{xs} \)) (\( \text{xi}, \text{xr} \)) = \text{pre @ \text{xi} # \text{suf}}
proof -
have \( \bigwedge r. \ r \in \text{set-option} \text{xr} \implies r \in \text{set} (\text{map \text{fst} } \text{xs}) \)
using crdt-ops-ref-exists rga-ops-def snoc.prems \text{x-pair by fastforce}
hence \text{xr} = \text{None } \lor (\exists r. \text{xr} = \text{Some \text{r} } \land r \in \text{set} (\text{map \text{fst} } \text{xs}))
using option.set-set by blast
hence \text{xr} = \text{None } \lor (\exists r. \text{xr} = \text{Some \text{r} } \land r \in \text{set} (\text{interp-rga} \text{xs}))
thus \( \exists \) \text{thesis}
using IH insert-rga-preserves-order by blast
qed
ultimately show distinct (interp-rga (\text{xs}@[\text{x}])))
by (metis Un-iff disjoint-insert(1) distinct.simps(2) distinct-append
interp-rga-tail-unfold list.simps(15) set-append)
qed

5.4 Proof that RGA satisfies the list specification

lemma final-insert:
assumes set (\text{xs}@[\text{x}]) = set (\text{ys}@[\text{x}])
and rga-ops (\text{xs}@[\text{x}])
and insert-ops (\text{ys}@[\text{x}])
and interp-rga \text{xs} = interp-ins \text{ys}
shows interp-rga (\text{xs}@[\text{x}]) = interp-ins (\text{ys}@[\text{x}])
proof -
obtain \text{oid ref} where \text{x-pair}: \text{x} = (\text{oid}, \text{ref}) by force
have distinct (\text{xs}@[\text{x}]) \land distinct (\text{ys}@[\text{x}])
using assms crdt-ops-distinct spec-ops-distinct rga-ops-def insert-ops-def by blast+
then have set \text{xs} = set \text{ys}
using assms(1) by force
have oid-greatest: \( \bigwedge i. \ i \in \text{set} \text{interp-rga } \text{xs} \implies i < \text{oid} \)
proof -
have \( \bigwedge i. \ i \in \text{set} \text{map \text{fst} } \text{ys} \implies i < \text{oid} \)
using assms(3) by (simp add: spec-ops-id-inc x-pair insert-ops-def)
hence \( \bigwedge i. \ i \in \text{set} \text{map \text{fst} } \text{xs} \implies i < \text{oid} \)
using (set \text{xs} = set \text{ys}) by auto
thus \( \bigwedge i. \ i \in \text{set} \text{interp-rga } \text{xs} \implies i < \text{oid} \)
using assms(2) interp-rga-ids rga-ops-rem-last by blast
 qed
thus interp-rga (\text{xs}@[\text{x}]) = interp-ins (\text{ys}@[\text{x}])
proof(cases \text{ref})
case None
moreover from this have interp-rga (interp-rga \text{xs}) (\text{oid}, \text{ref}) = \text{oid } \# \text{interp-rga } \text{xs}
using oid-greatest hd-in-set insert-body.elims insert-body.simps(1)
interp-rga.simps(1) list.sel(1) by metis
82
ultimately show \texttt{interp-rga} (\texttt{xs} @ [\texttt{x}]) = \texttt{interp-ins} (\texttt{ys} @ [\texttt{x}])

using \texttt{assms(4)} by (simp add: interp-ins-tail-unfold interp-rga-tail-unfold \texttt{x-pair})

next

case (Some \texttt{r})

have \exists \texttt{as bs}. \texttt{interp-rga xs} = \texttt{as} @ \texttt{r} # \texttt{bs}

proof –

have \texttt{r} \in \texttt{set} (map \texttt{fst} \texttt{xs})

using \texttt{assms(2)} Some by (simp add: rga-ops-ref-exists \texttt{x-pair})

hence \texttt{r} \in \texttt{set} (\texttt{interp-rga xs})

using \texttt{assms(2)} interp-rga-ids rga-ops-rem-last by blast

thus \texttt{thesis} by (simp add: split-list)

qed

from this obtain \texttt{as bs} where \texttt{as-bs}:

\texttt{interp-rga xs} = \texttt{as} @ \texttt{r} # \texttt{bs}

by force

hence distinct (\texttt{as} @ \texttt{r} # \texttt{bs})

by (metis \texttt{assms(2)} interp-rga-distinct rga-ops-rem-last)

moreover have insert-spec (\texttt{as} @ \texttt{r} # \texttt{bs}) (\texttt{oid}, \texttt{Some \texttt{r}}) = \texttt{as} @ \texttt{r} # \texttt{oid} # \texttt{bs}

by (meson \langle \texttt{distinct (as} @ \texttt{r} # \texttt{bs)} \rangle insert-after-ref)

ultimately show \texttt{interp-rga} (\texttt{xs} @ [\texttt{x}]) = \texttt{interp-ins} (\texttt{ys} @ [\texttt{x}])

by (metis \texttt{assms(4)} Some \texttt{as-bs} interp-ins-tail-unfold interp-rga-tail-unfold \texttt{x-pair})

qed

lemma \texttt{interp-rga-recorder}:

assumes \texttt{rga-ops} (\texttt{pre} @ \texttt{suf} @ [(\texttt{oid}, \texttt{ref})])

and \( \forall i. (i, \texttt{Some \texttt{r}}) \in \texttt{set} \texttt{suf} \implies \texttt{r} \neq \texttt{oid} \)

and \( \forall \texttt{r}. \texttt{ref} = \texttt{Some \texttt{r}} \implies \texttt{r} \neq \texttt{fst} \cdot \texttt{set} \texttt{suf} \)

shows \texttt{interp-rga} (\texttt{pre} @ (\texttt{oid}, \texttt{ref}) # \texttt{suf}) = \texttt{interp-rga} (\texttt{pre} @ \texttt{suf} @ [(\texttt{oid}, \texttt{ref})])

using \texttt{assms} \texttt{proof}(induction \texttt{suf} rule: List.rev-induct)

case \texttt{Nil}

then show \texttt{case} by simp

next

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

have \texttt{ref-not-x}: \( \forall \texttt{r}. \texttt{ref} = \texttt{Some \texttt{r}} \implies \texttt{r} \neq \texttt{fst} \cdot \texttt{x} \) using \texttt{snoc.prems}(3) by auto

have \texttt{IH}: \texttt{interp-rga} (\texttt{pre} @ (\texttt{oid}, \texttt{ref}) # \texttt{xs}) = \texttt{interp-rga} (\texttt{pre} @ \texttt{xs} @ [(\texttt{oid}, \texttt{ref})])

proof –

have \texttt{rga-ops} ((\texttt{pre} @ \texttt{xs}) @ [\texttt{x}] @ [(\texttt{oid}, \texttt{ref})])

using \texttt{snoc.prems(1)} by auto

moreover have \( \forall \texttt{r}. \texttt{ref} = \texttt{Some \texttt{r}} \implies \texttt{r} \neq \texttt{fst} \cdot \texttt{x} \)

by (simp add: \texttt{ref-not-x})

ultimately have \texttt{rga-ops} ((\texttt{pre} @ \texttt{xs}) @ [(\texttt{oid}, \texttt{ref})])

using \texttt{rga-ops-rem-penultimate}

by (metis \langle \texttt{no-types, lifting} \rangle Cons-eq-append-conv \texttt{prod.collapse})

thus \texttt{thesis} using \texttt{snoc} by force

qed

obtain \texttt{xi xrr} where \texttt{x-pair}: \texttt{x} = (\texttt{xi}, \texttt{xrr}) 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 \(\forall \text{xrr. } \text{xr} = \text{Some xrr} \implies \text{xrr} \neq \text{oid}\)
      using x-pair snoc.prems(2) by auto
    thus thesis
      using insert-rga-commutes ref-not-x by (metis-fst-conv x-pair)
qed
moreover have ...
= interp-rga (pre @ (oid, ref) # xs @ [x])
    using interp-rga-def
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 interp-ops-def by blast
qed
hence rga-ops (pre @ suf)
  using rga-ops-def by blast
moreover have set xs = set (pre @ suf)
  by (metis append-set-rem-last crdt-ops-distinct insert-ops-def rga-ops-def
    snoc.prems spec-ops-distinct ys-split)
ultimately show \( \text{thesis} \)

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 \( \text{thesis} \)

using IH snoc.prems(2) valid-rga final-insert append-assoc by metis

qed

moreover have ...

= interp-rga (pre @ [x] @ suf)

proof –

obtain oid ref where x-pair: \( x = (\text{oid}, \text{ref}) \)

by force

have \( \forall op2 r. \text{op2} \in \text{snd} \setminus \text{suf} \Rightarrow \text{r} \in \text{set-option op2} \Rightarrow \text{r} \neq \text{oid} \)

using snoc.prems

by (simp add: crdt-ops-independent-suf insert-ops-def rga-ops-def x-pair ys-split)

hence \( \forall i r. \text{(i, Some r)} \in \text{suf} \Rightarrow \text{r} \neq \text{oid} \)

by fastforce

moreover have \( \forall r. \text{ref} = \text{Some r} \Rightarrow \text{r} \notin \text{fst} \setminus \text{suf} \)

using crdt-ops-no-future-ref snoc.prems(3) x-pair ys-split

by (metis option.set-intros rga-ops-def)

ultimately show interp-rga (pre @ suf @ [x]) = interp-rga (pre @ [x] @ suf)

using interp-rga reorder valid-rga x-pair by force

qed

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

by (simp add: ys-split)

qed

lemma insert-ops-exist:

assumes rga-ops xs

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

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 \( \exists \text{ys. set ys = set xs } \land \text{ insert-ops ys } \land \text{ interp-ins ys } = \text{ interp-rga xs} \)

using assms rga-spec-equal insert-ops-exist by metis

end
References


