OpSets: Sequential Specifications for Replicated Datatypes
Proof Document

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

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

Abstract

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

Contents

1 Abstract OpSet

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

theory *OpSet*
  imports *Main*
begin

1.1 OpSet definition

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

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

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

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

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

lemma *opset-subset*:
assumes *opset Y deps*
and $X \subseteq Y$
shows *opset X deps*
⟨*proof*⟩

lemma *opset-insert*:
assumes *opset* (*insert x ops*) *deps*
shows *opset ops deps*
⟨*proof*⟩

lemma *opset-sublist*:
assumes *opset* (*set* (*xs* @ *ys* @ *zs*)) *deps*
shows *opset* (*set* (*xs* @ *zs*)) *deps*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *distinct-fst-append*:
assumes $x \in$ *set* (*map fst xs*)
and *distinct* (*map fst* (*xs* @ *ys*))
shows $x \notin$ *set* (*map fst ys*)
⟨*proof*⟩

lemma *distinct-set-remove-last*:
assumes *distinct* (*xs* @ [*x*])
shows *set xs* = *set* (*xs* @ [*x*]) − {*x*}
⟨*proof*⟩

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

⟨*proof*⟩

lemma *distinct-list-split*:
assumes *distinct xs*
and $xs = xa \mathbin{@} x \# ya$
and $xs = xb \mathbin{@} x \# yb$
shows $xa = xb \wedge ya = yb$
⟨*proof*⟩

lemma *distinct-append-swap*:
assumes *distinct* (*xs* @ *ys*)
shows *distinct* (*ys* @ *xs*)
⟨*proof*⟩

lemma *append-subset*:
assumes *set xs* = *set* (*ys* @ *zs*)
shows $set\ ys \subseteq set\ xs$ **and** $set\ zs \subseteq set\ xs$
⟨*proof*⟩

lemma *append-set-rem-last*:
assumes *set* (*xs* @ [*x*]) = *set* (*ys* @ [*x*] @ *zs*)
and *distinct* (*xs* @ [*x*]) **and** *distinct* (*ys* @ [*x*] @ *zs*)
shows *set xs* = *set* (*ys* @ *zs*)
⟨*proof*⟩

lemma *distinct-map-fst-remove1*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*remove1 x xs*))
⟨*proof*⟩

1.3 The *spec-ops* predicate

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

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

lemma *spec-ops-empty*:
shows *spec-ops* [] *deps*
⟨*proof*⟩

lemma *spec-ops-distinct*:
  assumes *spec-ops ops deps*
  shows *distinct ops*
  ⟨*proof*⟩

lemma *spec-ops-distinct-fst*:
  assumes *spec-ops ops deps*
  shows *distinct* (*map fst ops*)
  ⟨*proof*⟩

lemma *spec-ops-sorted*:
  assumes *spec-ops ops deps*
  shows *sorted* (*map fst ops*)
  ⟨*proof*⟩

lemma *spec-ops-rem-cons*:
  assumes *spec-ops* (*x* # *xs*) *deps*
  shows *spec-ops xs deps*
⟨*proof*⟩

lemma *spec-ops-rem-last*:
  assumes *spec-ops* (*xs* @ [*x*]) *deps*
  shows *spec-ops xs deps*
⟨*proof*⟩

lemma *spec-ops-remove1*:
  assumes *spec-ops xs deps*
  shows *spec-ops* (*remove1 x xs*) *deps*
  ⟨*proof*⟩

lemma *spec-ops-ref-less*:
  assumes *spec-ops xs deps*
    and (*oid*, *oper*) ∈ *set xs*
    and *r* ∈ *deps oper*
  shows *r* < *oid*
  ⟨*proof*⟩

lemma *spec-ops-ref-less-last*:
  assumes *spec-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
    and *r* ∈ *deps oper*
  shows *r* < *oid*
  ⟨*proof*⟩

lemma *spec-ops-id-inc*:
  assumes *spec-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
    and *x* ∈ *set* (*map fst xs*)
  shows *x* < *oid*
⟨*proof*⟩

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

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

lemma *spec-ops-split*:
assumes *spec-ops xs deps*
and $oid \notin set\ (map\ fst\ xs)$
shows $\exists\, pre\ suf.\ xs = pre\ @\ suf\ \wedge$
$(\forall\, i \in set\ (map\ fst\ pre).\ i < oid)\ \wedge$
$(\forall\, i \in set\ (map\ fst\ suf).\ oid < i)$
⟨*proof*⟩

lemma *spec-ops-exists-base*:
assumes *finite ops*
and $\bigwedge oid\ op1\ op2.\ (oid,\ op1) \in ops \Longrightarrow (oid,\ op2) \in ops \Longrightarrow op1 = op2$
and $\bigwedge oid\ oper\ ref.\ (oid,\ oper) \in ops \Longrightarrow ref \in deps\ oper \Longrightarrow ref < oid$
shows $\exists\, op\text{-}list.\ set\ op\text{-}list = ops \wedge spec\text{-}ops\ op\text{-}list\ deps$
⟨*proof*⟩

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

lemma *spec-ops-exists*:
assumes *opset ops deps*
shows $\exists\, op\text{-}list.\ set\ op\text{-}list = ops \wedge spec\text{-}ops\ op\text{-}list\ deps$
⟨*proof*⟩

lemma *spec-ops-oid-unique*:
assumes *spec-ops op-list deps*
and $(oid,\ op1) \in set\ op\text{-}list$
and $(oid,\ op2) \in set\ op\text{-}list$
shows $op1 = op2$
⟨*proof*⟩

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

lemma *spec-ops-is-opset*:
assumes *spec-ops op-list deps*
shows *opset* $(set\ op\text{-}list)$ *deps*
⟨*proof*⟩

1.4 The *crdt-ops* predicate

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

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

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

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

lemma *crdt-ops-intro*:
  assumes ⋀*r*. *r* ∈ *deps oper* ⟹ *r* ∈ *fst* ‘ *set xs* ∧ *r* < *oid*
    and *oid* ∉ *fst* ‘ *set xs*
    and *crdt-ops xs deps*
  shows *crdt-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
  ⟨*proof*⟩

lemma *crdt-ops-rem-last*:
  assumes *crdt-ops* (*xs* @ [*x*]) *deps*
  shows *crdt-ops xs deps*
  ⟨*proof*⟩

lemma *crdt-ops-ref-less*:
  assumes *crdt-ops xs deps*
    and (*oid*, *oper*) ∈ *set xs*
    and *r* ∈ *deps oper*
  shows *r* < *oid*
  ⟨*proof*⟩

lemma *crdt-ops-ref-less-last*:
  assumes *crdt-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
    and $r \in$ *deps oper*
  shows $r <$ *oid*
  ⟨*proof*⟩

lemma *crdt-ops-distinct-fst*:
  assumes *crdt-ops xs deps*
  shows *distinct* (*map fst xs*)
  ⟨*proof*⟩

lemma *crdt-ops-distinct*:
  assumes *crdt-ops xs deps*
  shows *distinct xs*
  ⟨*proof*⟩

lemma *crdt-ops-unique-last*:
  assumes *crdt-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
  shows *oid* $\notin$ *set* (*map fst xs*)
  ⟨*proof*⟩

lemma *crdt-ops-unique-mid*:
  assumes *crdt-ops* (*xs* @ [(*oid*, *oper*)] @ *ys*) *deps*
  shows *oid* $\notin$ *set* (*map fst xs*) $\wedge$ *oid* $\notin$ *set* (*map fst ys*)
  ⟨*proof*⟩

lemma *crdt-ops-ref-exists*:
  assumes *crdt-ops* (*pre* @ (*oid*, *oper*) # *suf*) *deps*
    and *ref* $\in$ *deps oper*
  shows *ref* $\in$ *fst* ‘ *set pre*
  ⟨*proof*⟩

lemma *crdt-ops-no-future-ref*:
  assumes *crdt-ops* (*xs* @ [(*oid*, *oper*)] @ *ys*) *deps*
  shows $\bigwedge$*ref*. *ref* $\in$ *deps oper* $\Longrightarrow$ *ref* $\notin$ *fst* ‘ *set ys*
⟨*proof*⟩

lemma *crdt-ops-reorder*:
  assumes *crdt-ops* (*xs* @ [(*oid*, *oper*)] @ *ys*) *deps*
    and $\bigwedge$*op2 r*. *op2* $\in$ *snd* ‘ *set ys* $\Longrightarrow$ $r \in$ *deps op2* $\Longrightarrow$ $r \neq$ *oid*
  shows *crdt-ops* (*xs* @ *ys* @ [(*oid*, *oper*)]) *deps*
  ⟨*proof*⟩

lemma *crdt-ops-rem-middle*:
  assumes *crdt-ops* (*xs* @ [(*oid*, *ref*)] @ *ys*) *deps*
    and $\bigwedge$*op2 r*. *op2* $\in$ *snd* ‘ *set ys* $\Longrightarrow$ $r \in$ *deps op2* $\Longrightarrow$ $r \neq$ *oid*
  shows *crdt-ops* (*xs* @ *ys*) *deps*
  ⟨*proof*⟩

lemma *crdt-ops-independent-suf*:
  assumes *spec-ops* (*xs* @ [(*oid*, *oper*)]) *deps*
    and *crdt-ops* (*ys* @ [(*oid*, *oper*)] @ *zs*) *deps*
    and *set* (*xs* @ [(*oid*, *oper*)]) = *set* (*ys* @ [(*oid*, *oper*)] @ *zs*)
  shows $\bigwedge$*op2 r*. *op2* ∈ *snd* ‘ *set zs* ⟹ *r* ∈ *deps op2* ⟹ *r* ≠ *oid*
⟨*proof*⟩

lemma *crdt-ops-reorder-spec*:
  assumes *spec-ops* (*xs* @ [*x*]) *deps*
    and *crdt-ops* (*ys* @ [*x*] @ *zs*) *deps*
    and *set* (*xs* @ [*x*]) = *set* (*ys* @ [*x*] @ *zs*)
  shows *crdt-ops* (*ys* @ *zs* @ [*x*]) *deps*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *crdt-ops-rem-penultimate*:
  assumes *crdt-ops* (*xs* @ [(*i1*, *r1*)] @ [(*i2*, *r2*)]) *deps*
    and $\bigwedge$*r*. *r* ∈ *deps r2* ⟹ *r* ≠ *i1*
  shows *crdt-ops* (*xs* @ [(*i2*, *r2*)]) *deps*
⟨*proof*⟩

lemma *crdt-ops-spec-ops-exist*:
  assumes *crdt-ops xs deps*
  shows ∃ *ys*. *set xs* = *set ys* ∧ *spec-ops ys deps*
  ⟨*proof*⟩

end

2 Specifying list insertion

theory *Insert-Spec*
  imports *OpSet*
begin

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

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

fun *insert-spec* :: *'oid list* ⇒ (*'oid* × *'oid option*) ⇒ *'oid list* **where**
  insert-spec xs (*oid*, *None*) = *oid#xs* |
  insert-spec [] (*oid*, *-*) = [] |
  insert-spec (*x#xs*) (*oid*, *Some ref*) =
    (*if x* = *ref then x* # *oid* # *xs*
        else x # (*insert-spec xs* (*oid*, *Some ref*)))

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

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

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

2.1 The *insert-ops* predicate

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

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

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

lemma *insert-ops-NilI* [*intro*!]:
  shows *insert-ops* []
  ⟨*proof*⟩

lemma *insert-ops-rem-last* [*dest*]:
  assumes *insert-ops* (*xs* @ [*x*])
  shows *insert-ops xs*
  ⟨*proof*⟩

lemma *insert-ops-rem-cons*:
  assumes *insert-ops* (*x* # *xs*)
  shows *insert-ops xs*

⟨*proof*⟩

lemma *insert-ops-appendD*:
assumes *insert-ops* (*xs* @ *ys*)
shows *insert-ops xs*
⟨*proof*⟩

lemma *insert-ops-rem-prefix*:
assumes *insert-ops* (*pre* @ *suf*)
shows *insert-ops suf*
⟨*proof*⟩

lemma *insert-ops-remove1*:
assumes *insert-ops xs*
shows *insert-ops* (*remove1 x xs*)
⟨*proof*⟩

lemma *last-op-greatest*:
assumes *insert-ops* (*op-list* @ [(*oid*, *oper*)])
and $x \in$ *set* (*map fst op-list*)
shows $x <$ *oid*
⟨*proof*⟩

lemma *insert-ops-ref-older*:
assumes *insert-ops* (*pre* @ [(*oid*, *Some ref*)] @ *suf*)
shows *ref* < *oid*
⟨*proof*⟩

lemma *insert-ops-memb-ref-older*:
assumes *insert-ops op-list*
and (*oid*, *Some ref*) ∈ *set op-list*
shows *ref* < *oid*
⟨*proof*⟩

2.2 Properties of the *insert-spec* function

lemma *insert-spec-none* [*simp*]:
shows *set* (*insert-spec xs* (*oid*, *None*)) = *set xs* ∪ {*oid*}
⟨*proof*⟩

lemma *insert-spec-set* [*simp*]:
assumes *ref* ∈ *set xs*
shows *set* (*insert-spec xs* (*oid*, *Some ref*)) = *set xs* ∪ {*oid*}
⟨*proof*⟩

lemma *insert-spec-nonex* [*simp*]:
assumes *ref* ∉ *set xs*
shows *insert-spec xs* (*oid*, *Some ref*) = *xs*
⟨*proof*⟩

lemma *list-greater-non-memb*:
  fixes *oid* :: *'oid*::{*linorder*}
  assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow x < oid$
    and $oid \in set\ xs$
  shows *False*
  ⟨*proof*⟩

lemma *inserted-item-ident*:
  assumes $a \in set\ (insert\text{-}spec\ xs\ (e,\ i))$
    and $a \notin set\ xs$
  shows $a = e$
  ⟨*proof*⟩

lemma *insert-spec-distinct* [*intro*]:
  fixes *oid* :: *'oid*::{*linorder*}
  assumes *distinct xs*
    and $\bigwedge x.\ x \in set\ xs \Longrightarrow x < oid$
    and $ref = Some\ r \longrightarrow r < oid$
  shows *distinct* (*insert-spec xs* (*oid*, *ref*))
  ⟨*proof*⟩

lemma *insert-after-ref*:
  assumes *distinct* (*xs* @ *ref* # *ys*)
  shows *insert-spec* (*xs* @ *ref* # *ys*) (*oid*, *Some ref*) = *xs* @ *ref* # *oid* # *ys*
  ⟨*proof*⟩

lemma *insert-somewhere*:
  assumes $ref = None \vee (ref = Some\ r \wedge r \in set\ list)$
  shows $\exists xs\ ys.\ list = xs\ @\ ys \wedge insert\text{-}spec\ list\ (oid,\ ref) = xs\ @\ oid\ \#\ ys$
  ⟨*proof*⟩

lemma *insert-first-part*:
  assumes $ref = None \vee (ref = Some\ r \wedge r \in set\ xs)$
  shows *insert-spec* (*xs* @ *ys*) (*oid*, *ref*) = (*insert-spec xs* (*oid*, *ref*)) @ *ys*
  ⟨*proof*⟩

lemma *insert-second-part*:
  assumes $ref = Some\ r$
    and $r \notin set\ xs$
    and $r \in set\ ys$
  shows *insert-spec* (*xs* @ *ys*) (*oid*, *ref*) = *xs* @ (*insert-spec ys* (*oid*, *ref*))
  ⟨*proof*⟩

2.3 Properties of the *interp-ins* function

lemma *interp-ins-empty* [*simp*]:
  shows *interp-ins* [] = []
  ⟨*proof*⟩

lemma *interp-ins-tail-unfold*:
shows *interp-ins* (*xs* @ [*x*]) = *insert-spec* (*interp-ins xs*) *x*
⟨*proof*⟩

lemma *interp-ins-subset* [*simp*]:
shows *set* (*interp-ins op-list*) ⊆ *set* (*map fst op-list*)
⟨*proof*⟩

lemma *interp-ins-distinct*:
assumes *insert-ops op-list*
shows *distinct* (*interp-ins op-list*)
⟨*proof*⟩

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* (λ*x*. *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*
⟨*proof*⟩

lemma *succ-rel-functional*:
assumes (*a*, *b1*) ∈ *succ-rel xs*
and (*a*, *b2*) ∈ *succ-rel xs*
and *distinct xs*
shows *b1* = *b2*

⟨*proof*⟩

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

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*⟩

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*⟩

lemma *succ-rel-insert-head*:
assumes *distinct* (*ref* # *list*)
shows *succ-rel* (*insert-spec* (*ref* # *list*) (*oid*, *Some ref*)) =
insert-alt (*succ-rel* (*ref* # *list*)) (*oid*, *ref*)
⟨*proof*⟩

lemma *succ-rel-insert-later*:
assumes *succ-rel* (*insert-spec* (*b* # *list*) (*oid*, *Some ref*)) =
insert-alt (*succ-rel* (*b* # *list*)) (*oid*, *ref*)
and $a \neq$ *ref*
and *distinct* (*a* # *b* # *list*)
shows *succ-rel* (*insert-spec* (*a* # *b* # *list*) (*oid*, *Some ref*)) =
insert-alt (*succ-rel* (*a* # *b* # *list*)) (*oid*, *ref*)
⟨*proof*⟩

lemma *succ-rel-insert-Some*:
assumes *distinct list*
shows *succ-rel* (*insert-spec list* (*oid*, *Some ref*)) = *insert-alt* (*succ-rel list*) (*oid*, *ref*)
⟨*proof*⟩

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

theorem *insert-alt-equivalent*:
assumes *insert-ops ops*
and *head* $\notin$ *fst* ‘ *set ops*
and $\bigwedge r.$ *Some* $r \in$ *snd* ‘ *set ops* $\Longrightarrow r \neq$ *head*
shows *succ-rel* (*head* # *interp-ins ops*) = *interp-alt head ops*
⟨*proof*⟩

2.5 The *list-order* predicate

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

definition *list-order* :: $('oid::\{linorder\} \times 'oid\ option)\ list \Rightarrow 'oid \Rightarrow 'oid \Rightarrow bool$
where
  $list\text{-}order\ ops\ x\ y \equiv \exists\, xs\ ys\ zs.\ interp\text{-}ins\ ops = xs\ @\ [x]\ @\ ys\ @\ [y]\ @\ zs$

lemma *list-orderI*:
  assumes $interp\text{-}ins\ ops = xs\ @\ [x]\ @\ ys\ @\ [y]\ @\ zs$
  shows *list-order ops x y*
  ⟨*proof*⟩

lemma *list-orderE*:
  assumes *list-order ops x y*
  shows $\exists\, xs\ ys\ zs.\ interp\text{-}ins\ ops = xs\ @\ [x]\ @\ ys\ @\ [y]\ @\ zs$
  ⟨*proof*⟩

lemma *list-order-memb1*:
  assumes *list-order ops x y*
  shows $x \in set\ (interp\text{-}ins\ ops)$
  ⟨*proof*⟩

lemma *list-order-memb2*:
  assumes *list-order ops x y*
  shows $y \in set\ (interp\text{-}ins\ ops)$
  ⟨*proof*⟩

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*⟩

lemma *insert-preserves-order*:
  assumes *insert-ops ops* **and** *insert-ops rest*
    and $rest = before\ @\ after$
    and $ops\ = before\ @\ (oid, ref)\ \#\ after$
  shows $\exists\, xs\ ys\ zs.\ interp\text{-}ins\ rest = xs\ @\ zs \wedge interp\text{-}ins\ ops = xs\ @\ ys\ @\ zs$
  ⟨*proof*⟩

lemma *distinct-fst*:
  assumes *distinct* (*map fst A*)
  shows *distinct A*

⟨*proof*⟩

lemma *subset-distinct-le*:
assumes *set* $A \subseteq$ *set* B **and** *distinct* A **and** *distinct* B
shows *length* $A \leq$ *length* B
⟨*proof*⟩

lemma *set-subset-length-eq*:
assumes *set* $A \subseteq$ *set* B **and** *length* $B \leq$ *length* A
and *distinct* A **and** *distinct* B
shows *set* $A =$ *set* B
⟨*proof*⟩

lemma *length-diff-Suc-exists*:
assumes *length xs* $-$ *length ys* $=$ *Suc m*
and *set ys* $\subseteq$ *set xs*
and *distinct ys* **and** *distinct xs*
shows $\exists\, e.\ e \in$ *set xs* $\wedge\ e \notin$ *set ys*
⟨*proof*⟩

lemma *app-length-lt-exists*:
assumes *xsa* @ *zsa* $=$ *xs* @ *ys*
and *length xsa* $\leq$ *length xs*
shows *xsa* @ (*drop* (*length xsa*) *xs*) $=$ *xs*
⟨*proof*⟩

lemma *list-order-monotonic*:
assumes *insert-ops* A **and** *insert-ops* B
and *set* $A \subseteq$ *set* B
and *list-order* $A\ x\ y$
shows *list-order* $B\ x\ y$
⟨*proof*⟩

end

3 Relationship to Strong List Specification

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

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

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

$\mathsf{elems}(A)$, called the *list order*, such that:

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

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

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

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

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

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

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

fun *interp-op* :: (*'oid list* × (*'oid* ⇀ *'val*)) ⇒ (*'oid* × (*'oid*, *'val*) *list-op*)
    ⇒ (*'oid list* × (*'oid* ⇀ *'val*)) **where**

interp-op (*list*, *vals*) (*oid*, *Insert ref val*) = (*insert-spec list* (*oid*, *ref*), *vals*(*oid* ↦ *val*)) |
interp-op (*list*, *vals*) (*oid*, *Delete ref*) = (*list*, *vals*(*ref* := *None*))

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

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

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

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

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

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

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

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

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

3.1 Lemmas about insertion and deletion

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

definition *inserted-ids* :: (*'oid*::{*linorder*} × (*'oid*, *'val*) *list-op*) *list* ⇒ *'oid list* **where**
inserted-ids ops ≡ *List.map-filter* (λ*oper*.
case oper of (*oid*, *Insert ref val*) ⇒ *Some oid* |

(*oid*, *Delete ref*) ⇒ *None*) *ops*

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

lemma *interp-ops-unfold-last*:
shows *interp-ops* (*xs* @ [*x*]) = *interp-op* (*interp-ops xs*) *x*
⟨*proof*⟩

lemma *map-filter-append*:
shows *List.map-filter P* (*xs* @ *ys*) = *List.map-filter P xs* @ *List.map-filter P ys*
⟨*proof*⟩

lemma *map-filter-Some*:
assumes *P x* = *Some y*
shows *List.map-filter P* [*x*] = [*y*]
⟨*proof*⟩

lemma *map-filter-None*:
assumes *P x* = *None*
shows *List.map-filter P* [*x*] = []
⟨*proof*⟩

lemma *insertions-last-ins*:
shows *insertions* (*xs* @ [(*oid*, *Insert ref val*)]) = *insertions xs* @ [(*oid*, *ref*)]
⟨*proof*⟩

lemma *insertions-last-del*:
shows *insertions* (*xs* @ [(*oid*, *Delete ref*)]) = *insertions xs*
⟨*proof*⟩

lemma *insertions-fst-subset*:
shows *set* (*map fst* (*insertions ops*)) ⊆ *set* (*map fst ops*)
⟨*proof*⟩

lemma *insertions-subset*:
assumes *list-ops A* **and** *list-ops B*
and *set A* ⊆ *set B*
shows *set* (*insertions A*) ⊆ *set* (*insertions B*)
⟨*proof*⟩

lemma *list-ops-insertions*:
assumes *list-ops ops*
shows *insert-ops* (*insertions ops*)
⟨*proof*⟩

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

lemma *inserted-ids-last-del*:
shows *inserted-ids* (*xs* @ [(*oid*, *Delete ref*)]) = *inserted-ids xs*
⟨*proof*⟩

lemma *inserted-ids-exist*:
shows *oid* ∈ *set* (*inserted-ids ops*) ⟷ (∃ *ref val*. (*oid*, *Insert ref val*) ∈ *set ops*)
⟨*proof*⟩

lemma *deleted-ids-last-ins*:
shows *deleted-ids* (*xs* @ [(*oid*, *Insert ref val*)]) = *deleted-ids xs*
⟨*proof*⟩

lemma *deleted-ids-last-del*:
shows *deleted-ids* (*xs* @ [(*oid*, *Delete ref*)]) = *deleted-ids xs* @ [*ref*]
⟨*proof*⟩

lemma *deleted-ids-exist*:
shows *ref* ∈ *set* (*deleted-ids ops*) ⟷ (∃ *i*. (*i*, *Delete ref*) ∈ *set ops*)
⟨*proof*⟩

lemma *deleted-ids-refs-older*:
assumes *list-ops* (*ops* @ [(*oid*, *oper*)])
shows ⋀*ref*. *ref* ∈ *set* (*deleted-ids ops*) ⟹ *ref* < *oid*
⟨*proof*⟩

3.2 Lemmas about interpreting operations

lemma *interp-ops-list-equiv*:
shows *fst* (*interp-ops ops*) = *interp-ins* (*insertions ops*)
⟨*proof*⟩

lemma *interp-ops-distinct*:
assumes *list-ops ops*
shows *distinct* (*fst* (*interp-ops ops*))
⟨*proof*⟩

lemma *list-order-equiv*:
shows *list-order ops x y* ⟷ *Insert-Spec.list-order* (*insertions ops*) *x y*
⟨*proof*⟩

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

lemma *insert-spec-nth-oid*:
  assumes *distinct xs*
    and $n <$ *length xs*
  shows *insert-spec xs* (*oid*, *Some* (*xs* ! *n*)) ! *Suc n* = *oid*
  ⟨*proof*⟩

lemma *insert-spec-inc-length*:
  assumes *distinct xs*
    and $n <$ *length xs*
  shows *length* (*insert-spec xs* (*oid*, *Some* (*xs* ! *n*))) = *Suc* (*length xs*)
  ⟨*proof*⟩

lemma *list-split-two-elems*:
  assumes *distinct xs*
    and $x \in$ *set xs* **and** $y \in$ *set xs*
    and $x \neq y$
  shows ∃ *pre mid suf*. *xs* = *pre* @ *x* # *mid* @ *y* # *suf* ∨ *xs* = *pre* @ *y* # *mid* @ *x* # *suf*
⟨*proof*⟩

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

Part 1(a) of Attiya et al.'s specification states that whenever the list is observed, the elements of the list are exactly those that have been inserted but not deleted. $\mathcal{A}_{\textsf{strong}}$ uses the visibility relation $\leq_{\textsf{vis}}$ 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 $a \in$ *dom* (*vals*) ⟷ (∃ *ref val*. (*a*, *Insert ref val*) ∈ *set ops*) ∧
                (∄ *i*. (*i*, *Delete a*) ∈ *set ops*)
  ⟨*proof*⟩

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

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

theorem *list-order-consistent*:
  assumes *list-ops some-ops* **and** *list-ops all-ops*
    and *set some-ops* $\subseteq$ *set all-ops*
    and *list-order some-ops x y*
  shows *list-order all-ops x y*
  ⟨*proof*⟩

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

theorem *correct-position-insert*:
  assumes *list-ops* (*ops* @ [(*oid*, *ins*)])
    and *ins* = *make-insert* (*fst* (*interp-ops ops*)) *val k*
    and *list* = *fst* (*interp-ops* (*ops* @ [(*oid*, *ins*)]))
  shows *list* ! (*min k* (*length list* − *1*)) = *oid*
⟨*proof*⟩

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*
  ⟨*proof*⟩

theorem *list-order-irrefl*:
  assumes *list-ops ops*
  shows ¬ *list-order ops x x*
⟨*proof*⟩

theorem *list-order-total*:
  assumes *list-ops ops*
    and $x \in$ *set* (*fst* (*interp-ops ops*))
    and $y \in$ *set* (*fst* (*interp-ops ops*))
    and $x \neq y$
  shows *list-order ops x y* ∨ *list-order ops y x*
⟨*proof*⟩

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 $\forall\, i \in set\ (map\ fst\ xs).\ P\ i$
    and $P\ x$
  shows $\forall\, i \in set\ (map\ fst\ (xs\ @\ [(x,\ y)])).\ P\ i$
  ⟨*proof*⟩

lemma *insert-ops-split*:
  assumes *insert-ops ops*
    and $(oid,\ ref) \in set\ ops$
  shows $\exists\, pre\ suf.\ ops = pre\ @\ [(oid,\ ref)]\ @\ suf\ \wedge$
    $(\forall\, i \in set\ (map\ fst\ pre).\ i < oid)\ \wedge$
    $(\forall\, i \in set\ (map\ fst\ suf).\ oid < i)$
  ⟨*proof*⟩

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

lemma *insert-ops-sorted-oids*:
  assumes $insert\text{-}ops\ (xs\ @\ [(i1,\ r1)]\ @\ ys\ @\ [(i2,\ r2)])$
  shows $i1 < i2$
⟨*proof*⟩

lemma *insert-ops-subset-last*:
  assumes $insert\text{-}ops\ (xs\ @\ [x])$
    and *insert-ops ys*
    and $set\ ys \subseteq set\ (xs\ @\ [x])$
    and $x \in set\ ys$
  shows $x = last\ ys$
  ⟨*proof*⟩

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

⟨*proof*⟩

lemma *distinct-append-butlast1*:
assumes *distinct* (*map fst xs* @ *map fst ys*)
shows *distinct* (*map fst* (*butlast xs*) @ *map fst ys*)
⟨*proof*⟩

lemma *distinct-append-butlast2*:
assumes *distinct* (*map fst xs* @ *map fst ys*)
shows *distinct* (*map fst xs* @ *map fst* (*butlast ys*))
⟨*proof*⟩

4.2 Lemmas about *interp-ins*

lemma *interp-ins-maybe-grow*:
assumes *insert-ops* (*xs* @ [(*oid*, *ref*)])
shows *set* (*interp-ins* (*xs* @ [(*oid*, *ref*)])) = *set* (*interp-ins xs*) ∨
set (*interp-ins* (*xs* @ [(*oid*, *ref*)])) = (*set* (*interp-ins xs*) ∪ {*oid*})
⟨*proof*⟩

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*})
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *interp-ins-last-None*:
shows *oid* ∈ *set* (*interp-ins* (*ops* @ [(*oid*, *None*)]))
⟨*proof*⟩

lemma *interp-ins-monotonic*:
assumes *insert-ops* (*pre* @ *suf*)
and *oid* ∈ *set* (*interp-ins pre*)
shows *oid* ∈ *set* (*interp-ins* (*pre* @ *suf*))
⟨*proof*⟩

lemma *interp-ins-append-non-memb*:
  assumes *insert-ops* (*pre* @ [(*oid*, *Some ref*)] @ *suf*)
    and *ref* ∉ *set* (*interp-ins pre*)
  shows *ref* ∉ *set* (*interp-ins* (*pre* @ [(*oid*, *Some ref*)] @ *suf*))
  ⟨*proof*⟩

lemma *interp-ins-append-memb*:
  assumes *insert-ops* (*pre* @ [(*oid*, *Some ref*)] @ *suf*)
    and *ref* ∈ *set* (*interp-ins pre*)
  shows *oid* ∈ *set* (*interp-ins* (*pre* @ [(*oid*, *Some ref*)] @ *suf*))
  ⟨*proof*⟩

lemma *interp-ins-append-forward*:
  assumes *insert-ops* (*xs* @ *ys*)
    and *oid* ∈ *set* (*interp-ins* (*xs* @ *ys*))
    and *oid* ∈ *set* (*map fst xs*)
  shows *oid* ∈ *set* (*interp-ins xs*)
  ⟨*proof*⟩

lemma *interp-ins-find-ref*:
  assumes *insert-ops* (*xs* @ [(*oid*, *Some ref*)] @ *ys*)
    and *ref* ∈ *set* (*interp-ins* (*xs* @ [(*oid*, *Some ref*)] @ *ys*))
  shows ∃ *r*. (*ref*, *r*) ∈ *set xs*
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

lemma *list-order-insert-none*:
  assumes *insert-ops* (*ops* @ [(*oid*, *None*)])
    and *x* ∈ *set* (*interp-ins ops*)
  shows *list-order* (*ops* @ [(*oid*, *None*)]) *oid x*
⟨*proof*⟩

lemma *list-order-insert-between*:
  assumes *insert-ops* (*ops* @ [(*oid*, *Some ref*)])
    and *list-order ops ref x*
  shows *list-order* (*ops* @ [(*oid*, *Some ref*)]) *oid x*

⟨*proof*⟩

4.4 The *insert-seq* predicate

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

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

lemma *insert-seq-nonempty*:
  assumes *insert-seq start xs*
  shows *xs* ≠ []
  ⟨*proof*⟩

lemma *insert-seq-hd*:
  assumes *insert-seq start xs*
  shows ∃ *oid*. *hd xs* = (*oid*, *start*)
  ⟨*proof*⟩

lemma *insert-seq-rem-last*:
  assumes *insert-seq start* (*xs* @ [*x*])
    and *xs* ≠ []
  shows *insert-seq start xs*
  ⟨*proof*⟩

lemma *insert-seq-butlast*:
  assumes *insert-seq start xs*
    and *xs* ≠ [] **and** *xs* ≠ [*last xs*]
  shows *insert-seq start* (*butlast xs*)
⟨*proof*⟩

lemma *insert-seq-last-ref*:
  assumes *insert-seq start* (*xs* @ [(*xi*, *xr*), (*yi*, *yr*)])
  shows *yr* = *Some xi*
  ⟨*proof*⟩

lemma *insert-seq-start-none*:
  assumes *insert-ops ops*
    and *insert-seq None xs* **and** *insert-ops xs*
    and *set xs* ⊆ *set ops*
  shows ∀ *i* ∈ *set* (*map fst xs*). *i* ∈ *set* (*interp-ins ops*)
  ⟨*proof*⟩

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 $\forall\, i \in$ *set* (*map fst xs*). *list-order ops ref i*
⟨*proof*⟩

lemma *insert-seq-no-start*:
assumes *insert-ops ops*
and *insert-seq* (*Some ref*) *xs* **and** *insert-ops xs*
and *set xs* $\subseteq$ *set ops*
and *ref* $\notin$ *set* (*interp-ins ops*)
shows $\forall\, i \in$ *set* (*map fst xs*). $i \notin$ *set* (*interp-ins ops*)
⟨*proof*⟩

4.5 The proof of no interleaving

lemma *no-interleaving-ordered*:
assumes *insert-ops ops*
and *insert-seq start xs* **and** *insert-ops xs*
and *insert-seq start ys* **and** *insert-ops ys*
and *set xs* $\subseteq$ *set ops* **and** *set ys* $\subseteq$ *set ops*
and *distinct* (*map fst xs* @ *map fst ys*)
and *fst* (*hd xs*) < *fst* (*hd ys*)
and $\bigwedge r$. *start* = *Some r* $\Longrightarrow$ $r \in$ *set* (*interp-ins ops*)
shows ($\forall\, x \in$ *set* (*map fst xs*). $\forall\, y \in$ *set* (*map fst ys*). *list-order ops y x*) $\wedge$
($\forall\, r$. *start* = *Some r* $\longrightarrow$ ($\forall\, x \in$ *set* (*map fst xs*). *list-order ops r x*) $\wedge$
($\forall\, y \in$ *set* (*map fst ys*). *list-order ops r y*))
⟨*proof*⟩

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* $\subseteq$ *set ops* **and** *set ys* $\subseteq$ *set ops*
and *distinct* (*map fst xs* @ *map fst ys*)
and *start* = *None* $\vee$ ($\exists\, r$. *start* = *Some r* $\wedge$ $r \in$ *set* (*interp-ins ops*))
shows ($\forall\, x \in$ *set* (*map fst xs*). $\forall\, y \in$ *set* (*map fst ys*). *list-order ops x y*) $\vee$
($\forall\, x \in$ *set* (*map fst xs*). $\forall\, y \in$ *set* (*map fst ys*). *list-order ops y x*)
⟨*proof*⟩

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

theorem *missing-start-no-insertion*:
  assumes *insert-ops ops*
    and *insert-seq* (*Some start*) *xs* **and** *insert-ops xs*
    and *insert-seq* (*Some start*) *ys* **and** *insert-ops ys*
    and *set xs* ⊆ *set ops* **and** *set ys* ⊆ *set ops*
    and *start* ∉ *set* (*interp-ins ops*)
  shows ∀ *x* ∈ *set* (*map fst xs*) ∪ *set* (*map fst ys*). *x* ∉ *set* (*interp-ins ops*)
  ⟨*proof*⟩

end

5 The Replicated Growable Array (RGA)

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

theory *RGA*
  imports *Insert-Spec*
begin

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

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

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

5.1 Commutativity of *insert-rga*

lemma *insert-body-set-ins* [*simp*]:
  shows *set* (*insert-body xs e*) = *insert e* (*set xs*)
  ⟨*proof*⟩

lemma *insert-rga-set-ins*:
  assumes *i* ∈ *set xs*

shows *set* (*insert-rga xs* (*oid*, *Some i*)) = *insert oid* (*set xs*)
⟨*proof*⟩

lemma *insert-body-commutes*:
shows *insert-body* (*insert-body xs e1*) *e2* = *insert-body* (*insert-body xs e2*) *e1*
⟨*proof*⟩

lemma *insert-rga-insert-body-commute*:
assumes $i2 \neq Some\ e1$
shows *insert-rga* (*insert-body xs e1*) (*e2*, *i2*) = *insert-body* (*insert-rga xs* (*e2*, *i2*)) *e1*
⟨*proof*⟩

lemma *insert-rga-None-commutes*:
assumes $i2 \neq Some\ e1$
shows *insert-rga* (*insert-rga xs* (*e1*, *None*)) (*e2*, *i2*) =
insert-rga (*insert-rga xs* (*e2*, *i2*)) (*e1*, *None*)
⟨*proof*⟩

lemma *insert-rga-nonexistent*:
assumes $i \notin set\ xs$
shows *insert-rga xs* (*e*, *Some i*) = *xs*
⟨*proof*⟩

lemma *insert-rga-Some-commutes*:
assumes $i1 \in set\ xs$ **and** $i2 \in set\ xs$
and $e1 \neq i2$ **and** $e2 \neq i1$
shows *insert-rga* (*insert-rga xs* (*e1*, *Some i1*)) (*e2*, *Some i2*) =
insert-rga (*insert-rga xs* (*e2*, *Some i2*)) (*e1*, *Some i1*)
⟨*proof*⟩

lemma *insert-rga-commutes*:
assumes $i2 \neq Some\ e1$ **and** $i1 \neq Some\ e2$
shows *insert-rga* (*insert-rga xs* (*e1*, *i1*)) (*e2*, *i2*) =
insert-rga (*insert-rga xs* (*e2*, *i2*)) (*e1*, *i1*)
⟨*proof*⟩

lemma *insert-body-split*:
shows $\exists p\ s.\ xs = p\ @\ s \wedge insert\text{-}body\ xs\ e = p\ @\ e\ \#\ s$
⟨*proof*⟩

lemma *insert-between-elements*:
assumes $xs = pre\ @\ ref\ \#\ suf$
and *distinct xs*
and $\bigwedge i.\ i \in set\ xs \Longrightarrow i < e$
shows $insert\text{-}rga\ xs\ (e,\ Some\ ref) = pre\ @\ ref\ \#\ e\ \#\ suf$
⟨*proof*⟩

lemma *insert-rga-after-ref*:

assumes $\forall x \in set\ as.\ a \neq x$
and *insert-body* $(cs\ @\ ds)\ e = cs\ @\ e\ \#\ ds$
shows *insert-rga* $(as\ @\ a\ \#\ cs\ @\ ds)\ (e,\ Some\ a) = as\ @\ a\ \#\ cs\ @\ e\ \#\ ds$
⟨*proof*⟩

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

5.2 Lemmas about the *rga-ops* predicate

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

lemma *rga-ops-rem-last*:
assumes *rga-ops* $(xs\ @\ [x])$
shows *rga-ops xs*
⟨*proof*⟩

lemma *rga-ops-rem-penultimate*:
assumes *rga-ops* $(xs\ @\ [(i1,\ r1),\ (i2,\ r2)])$
and $\bigwedge r.\ r2 = Some\ r \Longrightarrow r \neq i1$
shows *rga-ops* $(xs\ @\ [(i2,\ r2)])$
⟨*proof*⟩

lemma *rga-ops-ref-exists*:
assumes *rga-ops* $(pre\ @\ (oid,\ Some\ ref)\ \#\ suf)$
shows $ref \in fst$ ‘ *set pre*
⟨*proof*⟩

5.3 Lemmas about the *interp-rga* function

lemma *interp-rga-tail-unfold*:
shows *interp-rga* $(xs@[x])$ = *insert-rga* (*interp-rga* (xs)) x
⟨*proof*⟩

lemma *interp-rga-ids*:
assumes *rga-ops xs*
shows *set* (*interp-rga xs*) = *set* (*map fst xs*)
⟨*proof*⟩

lemma *interp-rga-distinct*:
assumes *rga-ops xs*
shows *distinct* (*interp-rga xs*)
⟨*proof*⟩

5.4 Proof that RGA satisfies the list specification

lemma *final-insert*:
  assumes $set\ (xs\ @\ [x]) = set\ (ys\ @\ [x])$
    and $rga\text{-}ops\ (xs\ @\ [x])$
    and $insert\text{-}ops\ (ys\ @\ [x])$
    and $interp\text{-}rga\ xs = interp\text{-}ins\ ys$
  shows $interp\text{-}rga\ (xs\ @\ [x]) = interp\text{-}ins\ (ys\ @\ [x])$
⟨*proof*⟩

lemma *interp-rga-reorder*:
  assumes $rga\text{-}ops\ (pre\ @\ suf\ @\ [(oid,\ ref)])$
    and $\bigwedge i\ r.\ (i,\ Some\ r) \in set\ suf \Longrightarrow r \neq oid$
    and $\bigwedge r.\ ref = Some\ r \Longrightarrow r \notin fst\ `\ set\ suf$
  shows $interp\text{-}rga\ (pre\ @\ (oid,\ ref)\ \#\ suf) = interp\text{-}rga\ (pre\ @\ suf\ @\ [(oid,\ ref)])$
  ⟨*proof*⟩

lemma *rga-spec-equal*:
  assumes $set\ xs = set\ ys$
    and $insert\text{-}ops\ xs$
    and $rga\text{-}ops\ ys$
  shows $interp\text{-}ins\ xs = interp\text{-}rga\ ys$
  ⟨*proof*⟩

lemma *insert-ops-exist*:
  assumes $rga\text{-}ops\ xs$
  shows $\exists\, ys.\ set\ xs = set\ ys \wedge insert\text{-}ops\ ys$
  ⟨*proof*⟩

theorem *rga-meets-spec*:
  assumes $rga\text{-}ops\ xs$
  shows $\exists\, ys.\ set\ ys = set\ xs \wedge insert\text{-}ops\ ys \wedge interp\text{-}ins\ ys = interp\text{-}rga\ xs$
  ⟨*proof*⟩

end

References

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

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

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

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