Formally Verified Suffix Array Construction

Louis Cheung and Christine Rizkallah

October 3, 2024

Abstract

A suffix array [2] is a data structure that is extensively used in text retrieval and data compression applications, including query suggestion mechanisms in web search, and in bioinformatics tools for DNA sequencing and matching. This wide applicability means that algorithms for constructing suffix arrays are of great practical importance. The Suffix Array by Induced Sorting (SA-IS) algorithm [3] is a conceptually complex yet highly efficient suffix array construction technique, based on an earlier algorithm [1].

As part of this formalization, we have developed the SA-IS algorithm in Isabelle/HOL and formally verified that it is equivalent to a mathematical functional specification of suffix arrays. This required verifying a wide range of underlying properties of lists and suffixes, that could be reused in other contexts. We also used Isabelle's code extraction facilities to extract an executable Haskell implementation of SAIS. In particular, this entry includes the following: an axiomatic characterisation of suffix array construction; a formally verified encoding of a straightforward but inefficient suffix array construction algorithm (validating the specification); and a formally verified encoding of the linear time SA-IS algorithm.

Contents

theory *Nat-Util*
  imports *Main*
begin

1 HOL

lemma *duplicate-assms*:
  $([\![P; P]\!] \Longrightarrow Q) \equiv (P \Longrightarrow Q)$
  ⟨*proof*⟩

2 Natural Number Arithmetic

lemma *div-2-eq-Suc*:
  $[\![x \ div \ 2 = y \ div \ 2; x \neq y]\!] \Longrightarrow (y = Suc \ x) \lor (x = Suc \ y)$
  ⟨*proof*⟩

lemma *Suc-m-sub-n-div-2*:
  $Suc \ ((m - n) \ div \ 2) > (m - Suc \ n) \ div \ 2$
  ⟨*proof*⟩

lemma *Suc-div-2-less-Suc*:
  $Suc \ x \ div \ 2 < Suc \ x$
  ⟨*proof*⟩

lemma *nat-x-less-y-le-Suc-x*:

$[\![x < y;\ y \leq Suc\ x]\!] \Longrightarrow y = Suc\ x$
⟨*proof*⟩

lemma *nat-sub-eq-add*:
$[\![(a :: nat) - b = c - d;\ b < a]\!] \Longrightarrow a + d = c + b$
⟨*proof*⟩

end
theory *Fun-Util*
imports *Main*
begin

3 Monotonic Functions

lemma *strict-mono-leD*: $strict\text{-}mono\ r \Longrightarrow m \leq n \Longrightarrow r\ m \leq r\ n$
⟨*proof*⟩

definition *map-to-nat* :: $('a :: linorder\ list) \Rightarrow ('a \Rightarrow nat)$
where
$map\text{-}to\text{-}nat\ xs = (\lambda x.\ card\ \{y|y.\ y \in set\ xs \wedge y < x\})$

lemma *map-to-nat-strict-mono-on*:
strict-mono-on (*set xs*) (*map-to-nat xs*)
⟨*proof*⟩

lemma *strict-mono-on-map-set-ex*:
$\exists (f :: ('a :: linorder \Rightarrow nat)).\ strict\text{-}mono\text{-}on\ (set\ xs)\ f$
⟨*proof*⟩

locale *Linorder-to-Nat-List* =
fixes *map-to-nat* :: $'a :: linorder\ list \Rightarrow 'a \Rightarrow nat$
and *xs* :: $'a :: linorder\ list$
assumes *map-to-nat-strict-mono-on*: *strict-mono-on* (*set xs*) (*map-to-nat xs*)

context *Linorder-to-Nat-List* **begin**

lemma *strict-mono-on-Suc-map-to-nat*:
strict-mono-on (*set xs*) (λx. *Suc* (*map-to-nat xs x*))
⟨*proof*⟩

end

lemma *Linorder-to-Nat-List-ex*:
$\exists \alpha.\ Linorder\text{-}to\text{-}Nat\text{-}List\ \alpha\ xs$
⟨*proof*⟩

end
theory *Set-Util*

imports *Main*
begin

4 Sets

lemma *pigeonhole-principle-advanced*:
assumes *finite A*
and *finite B*
and $A \cap B = \{\}$
and *card A* $>$ *card B*
and *bij-betw f* $(A \cup B)$ $(A \cup B)$
shows $\exists\, a \in A.\ f\ a \in A$
⟨*proof*⟩

lemma *Suc-mod-n-bij-betw*:
bij-betw $(\lambda x.\ Suc\ x\ mod\ n)$ $\{0..<n\}$ $\{0..<n\}$
⟨*proof*⟩

lemma *subset-upt-no-Suc*:
assumes $A \subseteq \{1..<n\}$
and $\forall\, x \in A.\ Suc\ x \notin A$
shows *card A* $\leq$ *n div 2*
⟨*proof*⟩

lemma *in-set-mapD*:
$x \in set\ (map\ f\ xs) \Longrightarrow \exists\, y \in set\ xs.\ x = f\ y$
⟨*proof*⟩

4.1 From AutoCorres

lemma *disjointI'*:
assumes $\bigwedge x\ y.\ [\![\ x \in A;\ y \in B\ ]\!] \Longrightarrow x \neq y$
shows $A \cap B = \{\}$
⟨*proof*⟩

lemma *disjoint-subset2*:
assumes $B' \subseteq B$ **and** $A \cap B = \{\}$
shows $A \cap B' = \{\}$
⟨*proof*⟩

end
theory *List-Util*
imports *Main*
begin

5 General Lists

lemma *list-cases-3*:
$T = [] \vee (\exists x.\ T = [x]) \vee (\exists a\ b\ xs.\ T = a \# b \# xs)$
⟨*proof*⟩

lemma *length-cons-cons*:
$T = a \# b \# xs \Longrightarrow \exists n.\ length\ T = Suc\ (Suc\ n)$
⟨*proof*⟩

lemma *length-Suc-Suc*:
$length\ T = Suc\ (Suc\ n) \Longrightarrow \exists a\ b\ xs.\ T = a \# b \# xs$
⟨*proof*⟩

lemma *length-Suc-0*:
$length\ xs = Suc\ 0 \Longrightarrow \exists x.\ xs = [x]$
⟨*proof*⟩

lemma *map-eq-replicate*:
$\forall x \in set\ xs.\ f\ x = k \Longrightarrow map\ f\ xs = replicate\ (length\ xs)\ k$
⟨*proof*⟩

lemma *map-upt-eq-replicate*:
$\forall x \in set\ [i..<j].\ f\ x = k \Longrightarrow map\ f\ [i..<j] = replicate\ (j - i)\ k$
⟨*proof*⟩

lemma *in-set-list-update*:
$[\![x \in set\ xs;\ xs\ !\ k \neq x]\!] \Longrightarrow x \in set\ (xs[k := y])$
⟨*proof*⟩

lemma *Max-greD*:
$i < length\ s \Longrightarrow Max\ (set\ s) \geq s\ !\ i$
⟨*proof*⟩

lemma *list-neq-rc1*:
$(\exists z\ zs.\ xs = ys\ @\ z \# zs) \Longrightarrow xs \neq ys$
⟨*proof*⟩

lemma *list-neq-rc2*:
$(\exists z\ zs.\ ys = xs\ @\ z \# zs) \Longrightarrow xs \neq ys$
⟨*proof*⟩

lemma *list-neq-rc3*:
$(\exists x\ y\ as\ bs\ cs.\ xs = as\ @\ x \# bs \wedge ys = as\ @\ y \# cs \wedge x \neq y) \Longrightarrow xs \neq ys$
⟨*proof*⟩

lemma *list-neq-rc*:
$(\exists z\ zs.\ xs = ys\ @\ z \# zs) \vee$

$(\exists z\ zs.\ ys = xs\ @\ z\ \#\ zs) \vee$
$(\exists x\ y\ as\ bs\ cs.\ xs = as\ @\ x\ \#\ bs \wedge ys = as\ @\ y\ \#\ cs \wedge x \neq y) \Longrightarrow$
$xs \neq ys$
⟨*proof*⟩

lemma *list-neq-fc*:
$xs \neq ys \Longrightarrow$
$(\exists z\ zs.\ xs = ys\ @\ z\ \#\ zs) \vee$
$(\exists z\ zs.\ ys = xs\ @\ z\ \#\ zs) \vee$
$(\exists x\ y\ as\ bs\ cs.\ xs = as\ @\ x\ \#\ bs \wedge ys = as\ @\ y\ \#\ cs \wedge x \neq y)$
⟨*proof*⟩

lemma *list-neq-cases*:
$xs \neq ys \longleftrightarrow$
$(\exists z\ zs.\ xs = ys\ @\ z\ \#\ zs) \vee$
$(\exists z\ zs.\ ys = xs\ @\ z\ \#\ zs) \vee$
$(\exists x\ y\ as\ bs\ cs.\ xs = as\ @\ x\ \#\ bs \wedge ys = as\ @\ y\ \#\ cs \wedge x \neq y)$
⟨*proof*⟩

6 Find

lemma *findSomeD*:
$find\ P\ xs = Some\ x \Longrightarrow P\ x \wedge x \in set\ xs$
⟨*proof*⟩

lemma *findNoneD*:
$find\ P\ xs = None \Longrightarrow \forall x \in set\ xs.\ \neg P\ x$
⟨*proof*⟩

7 Filter

lemma *filter-update-nth-success*:
$[\![P\ v;\ i < length\ xs]\!] \Longrightarrow$
$filter\ P\ (xs[i := v]) = (filter\ P\ (take\ i\ xs))\ @\ [v]\ @\ (filter\ P\ (drop\ (Suc\ i)\ xs))$
⟨*proof*⟩

lemma *filter-update-nth-fail*:
$[\![\neg P\ v;\ i < length\ xs]\!] \Longrightarrow$
$filter\ P\ (xs[i := v]) = (filter\ P\ (take\ i\ xs))\ @\ (filter\ P\ (drop\ (Suc\ i)\ xs))$
⟨*proof*⟩

lemma *filter-take-nth-drop-success*:
$[\![i < length\ xs;\ P\ (xs\ !\ i)]\!] \Longrightarrow$
$filter\ P\ xs = (filter\ P\ (take\ i\ xs))\ @\ [xs\ !\ i]\ @\ (filter\ P\ (drop\ (Suc\ i)\ xs))$
⟨*proof*⟩

lemma *filter-take-nth-drop-fail*:
$[\![i < length\ xs;\ \neg P\ (xs\ !\ i)]\!] \Longrightarrow$

$filter\ P\ xs = (filter\ P\ (take\ i\ xs))\ @\ (filter\ P\ (drop\ (Suc\ i)\ xs))$
⟨proof⟩

lemma *filter-nth-1*:
$[\![i < length\ xs;\ P\ (xs\ !\ i)]\!] \Longrightarrow$
$\exists\, i'.\ i' < length\ (filter\ P\ xs) \wedge (filter\ P\ xs)\ !\ i' = xs\ !\ i$
⟨proof⟩

lemma *filter-nth-2*:
$[\![i < length\ (filter\ P\ xs)]\!] \Longrightarrow$
$\exists\, i'.\ i' < length\ xs \wedge (filter\ P\ xs)\ !\ i = xs\ !\ i'$
⟨proof⟩

lemma *filter-nth-relative-1*:
$[\![i < length\ xs;\ P\ (xs\ !\ i);\ j < i;\ P\ (xs\ !\ j)]\!] \Longrightarrow$
$\exists\, i'\ j'.\ i' < length\ (filter\ P\ xs) \wedge j' < i' \wedge (filter\ P\ xs)\ !\ i' = xs\ !\ i\ \wedge$
$(filter\ P\ xs)\ !\ j' = xs\ !\ j$
⟨proof⟩

lemma *filter-nth-relative-neq-1*:
assumes $i < length\ xs\ P\ (xs\ !\ i)\ j < length\ xs\ P\ (xs\ !\ j)\ i \neq j$
shows $\exists\, i'\ j'.\ i' < length\ (filter\ P\ xs) \wedge j' < length\ (filter\ P\ xs) \wedge (filter\ P\ xs)\ !$
$i' = xs\ !\ i\ \wedge$
$(filter\ P\ xs)\ !\ j' = xs\ !\ j \wedge i' \neq j'$
⟨proof⟩

lemma *filter-nth-relative-2*:
$[\![i < length\ (filter\ P\ xs);\ j < i]\!] \Longrightarrow$
$\exists\, i'\ j'.\ i' < length\ xs \wedge j' < i' \wedge (filter\ P\ xs)\ !\ i = xs\ !\ i' \wedge (filter\ P\ xs)\ !\ j = xs$
$!\ j'$
⟨proof⟩

lemma *filter-nth-relative-neq-2*:
assumes $i < length\ (filter\ P\ xs)\ j < length\ (filter\ P\ xs)\ i \neq j$
shows $\exists\, i'\ j'.\ i' < length\ xs \wedge j' < length\ xs \wedge xs\ !\ i' = (filter\ P\ xs)\ !\ i\ \wedge$
$xs\ !\ j' = (filter\ P\ xs)\ !\ j \wedge i' \neq j'$
⟨proof⟩

lemma *filter-find*:
$filter\ P\ xs \neq [] \Longrightarrow find\ P\ xs = Some\ ((filter\ P\ xs)\ !\ 0)$
⟨proof⟩

lemma *filter-nth-update-subset*:
$set\ (filter\ P\ (xs[i := v])) \subseteq \{v\} \cup set\ (filter\ P\ xs)$
⟨proof⟩

8 Upt

lemma *card-upt*:

card $\{0..<n\} = n$
⟨*proof*⟩

lemma *bounded-distinct-subset-upt-length*:
$[\![distinct\ xs;\ \forall\, i<length\ xs.\ xs\ !\ i < length\ xs]\!] \Longrightarrow set\ xs \subseteq \{0..<length\ xs\}$
⟨*proof*⟩

lemma *bounded-distinct-eq-upt-length*:
assumes *distinct xs*
assumes $\forall\, i < length\ xs.\ xs\ !\ i < length\ xs$
shows $set\ xs = \{0..<length\ xs\}$
⟨*proof*⟩

lemma *set-map-nth-subset*:
assumes $n \leq length\ xs$
shows $set\ (map\ (nth\ xs)\ [0..<n]) \subseteq set\ xs$
⟨*proof*⟩

lemma *set-map-nth-eq*:
$set\ (map\ (nth\ xs)\ [0..<length\ xs]) = set\ xs$
⟨*proof*⟩

lemma *distinct-map-nth*:
assumes *distinct xs*
assumes $n \leq length\ xs$
shows $distinct\ (map\ (nth\ xs)\ [0..<n])$
⟨*proof*⟩
end
theory *Sorting-Util*
imports *Main*
begin

9 Lemmas about bijections

A convenient definition of an inverses between two sets

definition
inverses-on ::
$('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow bool$
where
$inverses\text{-}on\ f\ g\ A\ B \longleftrightarrow$
$(\forall\, x \in A.\ g\ (f\ x) = x) \wedge$
$(\forall\, x \in B.\ f\ (g\ x) = x)$

lemmas *inverses-onD1* = *inverses-on-def*[*THEN iffD1*, *THEN conjunct1*]
lemmas *inverses-onD2* = *inverses-on-def*[*THEN iffD1*, *THEN conjunct2*]

The inverses relation over maps

lemma *inverses-on-mapD*:

assumes *inverses-on* (*map f*) (*map g*) {*xs*. *set xs* ⊆ *A*} {*xs*. *set xs* ⊆ *B*}
shows *inverses-on f g A B*
⟨*proof*⟩

lemma *inverses-on-map*:
assumes *inverses-on f g A B*
shows *inverses-on* (*map f*) (*map g*) {*xs*. *set xs* ⊆ *A*} {*xs*. *set xs* ⊆ *B*}
⟨*proof*⟩

Inverses are symmetric

lemma *inverses-on-sym*:
inverses-on f g A B = *inverses-on g f B A*
⟨*proof*⟩

Convenient theorem to obtain the inverse of a bijection between two sets

lemma *bij-betw-inv-alt*:
assumes *bij-betw f A B*
shows ∃ *g*. *bij-betw g B A* ∧ *inverses-on f g A B*
⟨*proof*⟩

Bijections over maps

lemma *bij-betw-map*:
assumes *bij-betw f A B*
shows *bij-betw* (*map f*) {*xs*. *set xs* ⊆ *A*} {*xs*. *set xs* ⊆ *B*}
⟨*proof*⟩

Eliminating the map from a bijection relation

lemma *bij-betw-mapD*:
assumes *bij-betw* (*map f*) {*xs*. *set xs* ⊆ *A*} {*xs*. *set xs* ⊆ *B*}
shows *bij-betw f A B*
⟨*proof*⟩

Obtaining the inverse over map

lemma *bij-betw-inv-map*:
assumes *bij-betw f A B*
shows ∃ *g*. *bij-betw* (*map g*) {*xs*. *set xs* ⊆ *B*} {*xs*. *set xs* ⊆ *A*} ∧
inverses-on (*map f*) (*map g*) {*xs*. *set xs* ⊆ *A*} {*xs*. *set xs* ⊆ *B*}
⟨*proof*⟩

10 Lemmas about monotone functions

Note that the base version of monotone is used as the sorts cause some issues with the types

Essentially a general version of *strict-mono ?f* ⟹ (*?f ?x* < *?f ?y*) = (*?x* < *?y*)

lemma *monotone-on-iff*:
assumes *monotone-on A orda ordb f*

and *asymp-on A orda*
and *totalp-on A orda*
and *asymp-on* (*f* ' *A*) *ordb*
and *totalp-on* (*f* ' *A*) *ordb*
and $x \in A$
and $y \in A$
shows *orda x y* $\longleftrightarrow$ *ordb* (*f x*) (*f y*)
⟨*proof*⟩

The inverse of a monotonic function is also monotonic

lemma *monotone-on-bij-betw-inv*:
assumes *monotone-on A orda ordb f*
and *asymp-on A orda*
and *totalp-on A orda*
and *asymp-on B ordb*
and *totalp-on B ordb*
and *bij-betw f A B*
and *bij-betw g B A*
and *inverses-on f g A B*
shows *monotone-on B ordb orda g*
⟨*proof*⟩

lemma *monotone-on-bij-betw*:
assumes *monotone-on A orda ordb f*
and *asymp-on A orda*
and *totalp-on A orda*
and *asymp-on B ordb*
and *totalp-on B ordb*
and *bij-betw f A B*
shows $\exists$ *g. bij-betw g B A* $\wedge$ *inverses-on f g A B* $\wedge$ *monotone-on B ordb orda g*
⟨*proof*⟩

11 Sorting

11.1 General sorting

Intro for *sorted-wrt*

lemmas *sorted-wrtI* = *sorted-wrt-iff-nth-less*[*THEN iffD2*, *OF allI*, *OF allI*, *OF impI*, *OF impI*]

lemma *sorted-wrt-mapI*:
($\bigwedge i\ j.$ ⟦$i < j$; $j <$ *length xs*⟧ $\Longrightarrow$ *P* (*f* (*xs* ! *i*)) (*f* (*xs* ! *j*))) $\Longrightarrow$
sorted-wrt P (*map f xs*)
⟨*proof*⟩

lemma *sorted-wrt-mapD*:
($\bigwedge i\ j.$ ⟦*sorted-wrt P* (*map f xs*); $i < j$; $j <$ *length xs*⟧ $\Longrightarrow$ *P* (*f* (*xs* ! *i*)) (*f* (*xs* ! *j*)))
⟨*proof*⟩

lemma *monotone-on-sorted-wrt-map*:
assumes *monotone-on A orda ordb f*
and *sorted-wrt orda xs*
and *set xs* $\subseteq$ *A*
shows *sorted-wrt ordb* (*map f xs*)
⟨*proof*⟩

lemma *monotone-on-map-sorted-wrt*:
assumes *monotone-on A orda ordb f*
and *asymp-on A orda*
and *totalp-on A orda*
and *asymp-on* (*f* ‘ *A*) *ordb*
and *totalp-on* (*f* ‘ *A*) *ordb*
and *sorted-wrt ordb* (*map f xs*)
and *set xs* $\subseteq$ *A*
shows *sorted-wrt orda xs*
⟨*proof*⟩

11.2 Sorting on linear orders

context *linorder* **begin**

abbreviation *strict-sorted xs* $\equiv$ *sorted-wrt* (<) *xs*

lemma *sorted-nth-less-mono*:
⟦*sorted xs*; *i* < *length xs*; *j* < *length xs*; *i* $\neq$ *j*; *xs* ! *i* < *xs* ! *j*⟧ $\Longrightarrow$ *i* < *j*
⟨*proof*⟩

lemma *strict-sorted-nth-less-mono*:
⟦*strict-sorted xs*; *i* < *length xs*; *j* < *length xs*; *i* $\neq$ *j*; *xs* ! *i* < *xs* ! *j*⟧ $\Longrightarrow$ *i* < *j*
⟨*proof*⟩

lemma *strict-sorted-Min*:
⟦*strict-sorted xs*; *xs* $\neq$ []⟧ $\Longrightarrow$ *xs* ! *0* = *Min* (*set xs*)
⟨*proof*⟩

lemma *strict-sorted-take*:
assumes *strict-sorted xs*
and *i* < *length xs*
shows *set* (*take i xs*) = {*x*. *x* $\in$ *set xs* $\wedge$ *x* < *xs* ! *i*}
⟨*proof*⟩

lemma *strict-sorted-card-idx*:
⟦*strict-sorted xs*; *i* < *length xs*⟧ $\Longrightarrow$ *card* {*x*. *x* $\in$ *set xs* $\wedge$ *x* < *xs* ! *i*} = *i*
⟨*proof*⟩

lemmas *strict-sorted-distinct-set-unique* =
sorted-distinct-set-unique[*OF strict-sorted-imp-sorted - strict-sorted-imp-sorted*]

lemma *sorted-and-distinct-imp-strict-sorted*:
$[\![sorted\ xs;\ distinct\ xs]\!] \Longrightarrow strict\text{-}sorted\ xs$
⟨*proof*⟩

lemma *filter-sorted*:
$sorted\ xs \Longrightarrow sorted\ (filter\ P\ xs)$
⟨*proof*⟩

lemma *sorted-nth-eq*:
assumes $sorted\ xs$
and $j < length\ xs$
and $xs\ !\ i = xs\ !\ j$
and $i \leq k$
and $k \leq j$
shows $xs\ !\ k = xs\ !\ i$
⟨*proof*⟩

lemma *sorted-find-Min*:
$sorted\ xs \Longrightarrow \exists\, x \in set\ xs.\ P\ x \Longrightarrow List.find\ P\ xs = Some\ (Min\ \{x{\in}set\ xs.\ P\ x\})$
⟨*proof*⟩

lemma *sorted-cons-nil*:
$xs = [] \Longrightarrow sorted\ (x\ \#\ xs)$
⟨*proof*⟩

lemma *sorted-consI*:
$[\![xs \neq [];\ sorted\ xs;\ x \leq xs\ !\ 0]\!] \Longrightarrow sorted\ (x\ \#\ xs)$
⟨*proof*⟩

end

11.3 Sorting on orders

context *order* **begin**

lemma *strict-mono-strict-sorted-map-1*:
assumes $strict\text{-}mono\ \alpha$
and $strict\text{-}sorted\ xs$
shows $strict\text{-}sorted\ (map\ \alpha\ xs)$
⟨*proof*⟩

lemma *strict-mono-sorted-map-2*:
assumes $strict\text{-}mono\ \alpha$
and $strict\text{-}sorted\ (map\ \alpha\ xs)$
shows $strict\text{-}sorted\ xs$
⟨*proof*⟩

end

12 Mapping elements to natural numbers

This section contains a mapping from elements to natural numbers that maintains ordering.

definition *elm-rank* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow nat$
  where
elm-rank ord A x = *card* $\{y.\ y \in A \wedge ord\ y\ x\}$

lemma *monotone-on-elm-rank*:
  assumes *finite A*
  and *transp-on A ord*
  and *irreflp-on A ord*
  shows *monotone-on A ord* (<) (*elm-rank ord A*)
⟨*proof*⟩

lemma *elm-rank-insert-min*:
  assumes *finite A*
  and $x \notin A$
  and $\forall y \in A.\ ord\ x\ y$
  and $z \in A$
shows *elm-rank ord* (*insert x A*) *z* = *Suc* (*elm-rank ord A z*)
  ⟨*proof*⟩

definition (**in** *order*) *elem-rank* :: $'a\ set \Rightarrow 'a \Rightarrow nat$
  where
elem-rank = *elm-rank* (<)

lemma (**in** *order*) *strict-mono-on-elem-rank*:
  assumes *finite A*
  shows *strict-mono-on A* (*elem-rank A*)
  ⟨*proof*⟩

lemma (**in** *linorder*) *bij-betw-elem-rank-upt*:
  assumes *finite A*
  shows *bij-betw* (*elem-rank A*) *A* $\{0..<card\ A\}$
⟨*proof*⟩

lemma (**in** *order*) *elem-rank-insert-min*:
  $[\![finite\ A;\ x \notin A;\ \forall y{\in}A.\ x < y;\ z \in A]\!] \Longrightarrow$ *elem-rank* (*insert x A*) *z* = *Suc* (*elem-rank A z*)
  ⟨*proof*⟩
end
theory *Repeat*
  imports *Main*
begin

13 Repeat Function At Most N Times

fun *repeatatm* :: *nat* ⇒ (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'a* ⇒ *'b* ⇒ *'a*) ⇒ *'a* ⇒ *'b* ⇒ *'a*
  where
repeatatm 0 - - acc - = acc |
repeatatm (Suc n) f g acc obsv = (if f acc obsv then acc else repeatatm n f g (g acc obsv) obsv)

declare *repeatatm.simps*[*simp del*]

13.1 Step and early termination lemmas

lemma *repeatatm-step-stop-Suc*:
  f (repeatatm n f g a b) b
    ⟹ *repeatatm (Suc n) f g a b = repeatatm n f g a b*
⟨*proof*⟩

lemma *repeatatm-step*:
  ¬*f (repeatatm n f g a b) b*
    ⟹ *repeatatm (Suc n) f g a b = g (repeatatm n f g a b) b*
⟨*proof*⟩

lemma *repeatatm-step-forward*:
  ¬*f a b* ⟹ *repeatatm (Suc n) f g a b = repeatatm n f g (g a b) b*
  ⟨*proof*⟩

lemma *repeatatm-stop-Suc*:
  ⟦*f (repeatatm n f g a b) b*⟧ ⟹ *f (repeatatm (Suc n) f g a b) b*
⟨*proof*⟩

lemma *repeatatm-stop*:
  ⟦*f (repeatatm n f g a b) b*; *n* ≤ *m*⟧ ⟹ *f (repeatatm m f g a b) b*
⟨*proof*⟩

lemma *repeatatm-step-stop*:
  ⟦*f (repeatatm n f g a b) b*; *n* ≤ *m*⟧ ⟹ *repeatatm m f g a b = repeatatm n f g a b*
⟨*proof*⟩

lemma *repeatatm-not-stop-Suc*:
  ¬*f (repeatatm (Suc n) f g a b) b* ⟹ ¬*f (repeatatm n f g a b) b*
  ⟨*proof*⟩

lemma *repeatatm-maintain-inv*:
  assumes ⋀*a*. *P a* ⟹ *P (g a b)*
  shows *P a* ⟹ *P (repeatatm n f g a b)*
⟨*proof*⟩

14 Repeat Function N Times

definition *repeat* :: *nat* ⇒ (*'a* ⇒ *'b* ⇒ *'a*) ⇒ *'a* ⇒ *'b* ⇒ *'a*
where
repeat n f a b = *repeatatm n* (λ*x y*. *False*) *f a b*

lemma *repeat-0*:
repeat 0 f a b = *a*
⟨*proof*⟩

lemma *repeat-step*:
repeat (*Suc n*) *f a b* = *f* (*repeat n f a b*) *b*
⟨*proof*⟩

lemma *repeat-step-forward*:
repeat (*Suc n*) *f a b* = *repeat n f* (*f a b*) *b*
⟨*proof*⟩

lemma *repeat-maintain-inv*:
assumes ⋀*a*. *P a* ⟹ *P* (*f a b*)
shows *P a* ⟹ *P* (*repeat n f a b*)
⟨*proof*⟩

lemma *repeat-eq-fold*:
repeat n f a b = *fold* (λ- *a*. *f a b*) [*0*..<*n*] *a*
⟨*proof*⟩

end
theory *Continuous-Interval*
imports *Main*
begin

15 Continuous Intervals

definition
continuous-list :: (*nat* × *nat*) *list* ⇒ *bool*
where
continuous-list xs =
(∀ *i*. *Suc i* < *length xs* ⟶ *fst* (*xs* ! *Suc i*) = *snd* (*xs* ! *i*))

lemma *continuous-list-nil*:
continuous-list []
⟨*proof*⟩

lemma *continuous-list-singleton*:
continuous-list [*x*]
⟨*proof*⟩

lemma *continuous-list-cons*:

continuous-list (*x* # *xs*) $\Longrightarrow$ *continuous-list xs*
⟨*proof*⟩

lemma *continuous-list-app*:
continuous-list (*xs* @ *ys*) $\Longrightarrow$ *continuous-list xs* $\wedge$ *continuous-list ys*
⟨*proof*⟩

lemma *continuous-list-interval-1*:
assumes *continuous-list xs*
and *xs* $\neq$ []
and *fst* (*hd xs*) $\leq$ *i*
and *i* < *snd* (*last xs*)
shows $\exists$ *j* < *length xs*. *fst* (*xs* ! *j*) $\leq$ *i* $\wedge$ *i* < *snd* (*xs* ! *j*)
⟨*proof*⟩

lemma *continuous-list-interval-2*:
assumes *continuous-list xs*
and *length xs* = *Suc n*
and *fst* (*xs* ! *0*) $\leq$ *i*
and *i* < *snd* (*xs* ! *n*)
shows $\exists$ *j* < *length xs*. *fst* (*xs* ! *j*) $\leq$ *i* $\wedge$ *i* < *snd* (*xs* ! *j*)
⟨*proof*⟩

end
theory *List-Slice*
imports *Main*
begin

16 List Slices

fun *list-slice* ::
'a list $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'a list*
where
list-slice xs i j = *drop i* (*take j xs*)

lemma *length-list-slice*[*simp add*]:
length (*list-slice xs i j*) = (*min j* (*length xs*)) − *i*
⟨*proof*⟩

lemma *list-slice-cons*:
fixes *i j* :: *nat*
assumes *i* $\leq$ *j*
assumes *i* > *0*
shows *list-slice* (*x* # *xs*) *i j* = *list-slice xs* (*i* − *1*) (*j* − *1*)
⟨*proof*⟩

lemma *list-slice-append*:
fixes *i j k* :: *nat*
assumes *i* $\leq$ *j*

assumes $j \leq k$
shows *list-slice xs i k* = *list-slice xs i j* @ *list-slice xs j k*
⟨*proof*⟩

lemma *list-slice-0-length*:
fixes *xs* :: *'a list*
fixes *n* :: *nat*
assumes *length xs* ≤ *n*
shows *list-slice xs 0 n* = *xs*
⟨*proof*⟩

lemma *list-slice-n-n*[*simp add*]:
fixes *xs* :: *'a list*
fixes *n* :: *nat*
shows *list-slice xs n n* = []
⟨*proof*⟩

lemma *list-slice-nth*:
fixes *i s e* :: *nat*
fixes *xs* :: *'a list*
assumes *i* < *length xs*
assumes $s \leq i$
assumes $i < e$
shows (*list-slice xs s e*) ! ($i - s$) = *xs* ! *i*
⟨*proof*⟩

lemma *list-slice-start-gre-length*:
fixes *xs* :: *'a list*
fixes *s* :: *nat*
assumes *length xs* ≤ *s*
shows *list-slice xs s e* = []
⟨*proof*⟩

lemma *list-slice-end-gre-length*:
fixes *xs* :: *'a list*
fixes *e* :: *nat*
assumes *length xs* ≤ *e*
shows *list-slice xs s e* = *list-slice xs s* (*length xs*)
⟨*proof*⟩

lemma *fold-list-slice*:
fixes *i j* :: *nat*
fixes *B* :: *nat list*
assumes $i \leq j$
and *j* < *length B*
and *sorted B*
fixes *T zs* :: *'a list*
shows
fold (λ*x xs*. *xs* @ *list-slice T* (*B* ! *x*) (*B* ! *Suc x*)) [*i*..<*j*] *zs*

$= zs$ @ (*list-slice* T (B ! i) (B ! j))
⟨*proof*⟩

lemma *nth-list-slice*:
fixes $i\ s\ e$ :: *nat*
fixes *xs* :: *'a list*
assumes $i <$ *length* (*list-slice xs s e*)
shows (*list-slice xs s e*) ! i = *xs* ! ($s + i$)
⟨*proof*⟩

lemma *list-slice-nth-eq-iff-index-eq*:
fixes $i\ s\ e\ j$ :: *nat*
fixes *xs* :: *'a list*
assumes *distinct* (*list-slice xs s e*)
assumes $e \leq$ *length xs*
assumes $s \leq i$ **and** $i < e$
and $s \leq j$ **and** $j < e$
shows (*xs* ! i = *xs* ! j) ⟷ ($i = j$)
⟨*proof*⟩

lemma *distinct-list-slice*:
fixes $i\ j$ :: *nat*
fixes *xs* :: *'a list*
assumes *distinct xs*
shows *distinct* (*list-slice xs i j*)
⟨*proof*⟩

lemma *list-slice-nth-mem*:
fixes e :: *nat*
fixes *xs* :: *'a list*
fixes $s\ i$ :: *nat*
assumes $s \leq i$ **and** $i < e$
assumes $e \leq$ *length xs*
shows *xs* ! $i \in$ *set* (*list-slice xs s e*)
⟨*proof*⟩

lemma *nth-mem-list-slice*:
fixes x :: *'a*
fixes *xs* :: *'a list*
fixes $s\ e$ :: *nat*
assumes $x \in$ *set* (*list-slice xs s e*)
shows $\exists\, i <$ *length xs*.
$s \leq i \wedge$
$i < e \wedge$
xs ! $i = x$
⟨*proof*⟩

lemma *list-slice-subset*:
fixes $i\ j$ :: *nat*

fixes *xs* :: *'a list*
shows *set* (*list-slice xs i j*) ⊆ *set xs*
⟨*proof*⟩

lemma *list-slice-Suc*:
fixes *i j* :: *nat*
fixes *xs* :: *'a list*
assumes $i <$ *length xs*
assumes $i < j$
shows *list-slice xs i j* = *xs* ! *i* # *list-slice xs* (*Suc i*) *j*
⟨*proof*⟩

lemma *list-slice-update-unchanged-1*:
fixes *xs* :: *'a list*
fixes *i j k* :: *nat*
assumes $i < j$
shows *list-slice* (*xs*[*i* := *x*]) *j k* = *list-slice xs j k*
⟨*proof*⟩

lemma *list-slice-update-unchanged-2*:
fixes *i j k* :: *nat*
fixes *xs* :: *'a list*
assumes $k \leq i$
shows *list-slice* (*xs*[*i* := *x*]) *j k* = *list-slice xs j k*
⟨*proof*⟩

lemma *list-slice-update-changed*:
assumes $i <$ *length xs*
assumes $j \leq i$
assumes $i < k$
shows *list-slice* (*xs*[*i* := *x*]) *j k* = (*list-slice xs j k*)[*i* − *j* := *x*]
⟨*proof*⟩

lemma *list-slice-map-nth-upt*:
assumes $j <$ *length xs*
shows *list-slice xs i j* = *map* (*nth xs*) [*i*..<*j*]
⟨*proof*⟩

lemma *map-list-slice*:
map f (*list-slice xs i j*) = *list-slice* (*map f xs*) *i j*
⟨*proof*⟩

17 Sorted List Slice

lemma (**in** *linorder*) *sorted-list-slice*:
assumes *sorted xs*
shows *sorted* (*list-slice xs i j*)
⟨*proof*⟩

lemma (**in** *linorder*) *sorted-map-list-slice*:
  assumes *sorted* (*map f xs*)
  shows *sorted* (*map f* (*list-slice xs i j*))
  ⟨*proof*⟩

lemma (**in** *linorder*) *sorted-map-filter-list-slice*:
  assumes *sorted* (*map f* (*filter P xs*))
  shows *sorted* (*map f* (*filter P* (*list-slice xs i j*)))
⟨*proof*⟩

lemma (**in** *linorder*) *list-slice-sorted-nth-mono*:
  assumes *sorted* (*list-slice xs s e*)
  and $s \leq i$
  and $i \leq j$
  and $j < e$
  and $j <$ *length xs*
shows *xs* ! $i \leq$ *xs* ! j
⟨*proof*⟩
end
theory *List-Lexorder-Util*
  imports
    HOL−Library.List-Lexorder
begin

lemma *same-equiv-def*:
  $(\forall j<n.\ s\ !\ (i + j) = s\ !\ Suc\ (i + j)) = (\forall j\leq n.\ s\ !\ (i + j) = s\ !\ i)$
⟨*proof*⟩

lemma *list-less-ex*:
  $xs < ys \longleftrightarrow$
   $(\exists b\ c\ as\ bs\ cs.\ xs = as\ @\ b\ \#\ bs \wedge ys = as\ @\ c\ \#\ cs \wedge b < c) \vee$
   $(\exists c\ cs.\ ys = xs\ @\ c\ \#\ cs)$
  ⟨*proof*⟩

end
theory *List-Permutation-Util*
  imports *HOL−Combinatorics.List-Permutation ../util/List-Util*
begin

lemma *perm-distinct-set-of-upt-iff*:
  xs <~~> [*0..<n*] ⟷ *distinct xs* ∧ *set xs* = {*0..<n*}
  ⟨*proof*⟩

lemma *distinct-set-of-upto-length*:
  ⟦*distinct xs*; *set xs* = {*0..<n*}⟧ ⟹ *length xs* = *n*
  ⟨*proof*⟩

lemma *set-perm-upt*:

xs <~~> [0..<n] ⟹ set xs = {0..<n}
⟨proof⟩

lemma *perm-upt-length*:
xs <~~> [0..<n] ⟹ length xs = n
⟨proof⟩

lemma *perm-nth-ex*:
⟦xs <~~> [0..<n]; i < n⟧ ⟹ ∃ k < n. xs ! i = k
⟨proof⟩

lemma *ex-perm-nth*:
⟦xs <~~> [0..<n]; k < n⟧ ⟹ ∃ i < n. xs ! i = k
⟨proof⟩

lemma *set-map-nth-perm-subset*:
⟦ys <~~> [0..<n]; n ≤ length xs⟧ ⟹ set (map (nth xs) ys) ⊆ set xs
⟨proof⟩

lemma *set-map-nth-perm-eq*:
ys <~~> [0..<length xs] ⟹ set (map (nth xs) ys) = set xs
⟨proof⟩

lemma *distinct-map-nth-perm*:
⟦distinct xs; n ≤ length xs; ys <~~> [0..<n]⟧ ⟹ distinct (map (nth xs) ys)
⟨proof⟩

theorem *distinct-set-imp-perm*:
assumes distinct xs
and distinct ys
and set xs = set ys
shows xs <~~> ys
⟨proof⟩

theorem *perm-nth*:
assumes xs <~~> ys
and i < length xs
shows ∃ j < length ys. ys ! j = xs ! i
⟨proof⟩

lemma *sort-perm*:
xs <~~> sort xs
⟨proof⟩

end
theory *List-Lexorder-NS*
imports
../util/Sorting-Util
../util/List-Slice

../order/List-Permutation-Util

begin

18 General Non-standard Lexicographical Comparison

This section is based on the *lexord* classical lexicographical definition in the the List library but accounts for a variant of lexicographic order defined below that we rely on for verifying sais. The main difference is that this ordering preferences the original string over its prefix. For example, "aaa" is less than "aa", which in turn is less than "a".

definition *nslexord* :: ('a × 'a) *set* ⇒ ('a *list* × 'a *list*) *set* **where**
nslexord r = {(x,y). (∃ a v. x = y @ a # v) ∨
(∃ u a b v w. (a, b) ∈ r ∧ x = u @ a # v ∧ y = u @ b # w)}

definition *nslexordp* :: ('a ⇒ 'a ⇒ *bool*) ⇒ 'a *list* ⇒ 'a *list* ⇒ *bool*
where
nslexordp cmp xs ys ⟷
(∃ b c as bs cs. xs = as @ b # bs ∧ ys = as @ c # cs ∧ cmp b c) ∨
(∃ c cs. xs = ys @ c # cs)

lemma *nslexord-eq-nslexordp*:
(xs, ys) ∈ nslexord {(x, y). cmp x y} ⟷ nslexordp cmp xs ys
(xs, ys) ∈ nslexord r ⟷ nslexordp (λx y. (x, y) ∈ r) xs ys
⟨*proof*⟩

definition *nslexordeqp* :: ('a ⇒ 'a ⇒ *bool*) ⇒ 'a *list* ⇒ 'a *list* ⇒ *bool*
where
nslexordeqp cmp xs ys ⟷ *nslexordp cmp xs ys* ∨ (xs = ys)

18.1 Intro and Elimination

lemma *nslexordpI1*:
∃ b c as bs cs. xs = as @ b # bs ∧ ys = as @ c # cs ∧ cmp b c ⟹ nslexordp cmp xs ys
⟨*proof*⟩

lemma *nslexordpI2*:
∃ c cs. xs = ys @ c # cs ⟹ nslexordp cmp xs ys
⟨*proof*⟩

lemma *nslexordpE*:
nslexordp cmp xs ys ⟹
(∃ b c as bs cs. xs = as @ b # bs ∧ ys = as @ c # cs ∧ cmp b c) ∨
(∃ c cs. xs = ys @ c # cs)
⟨*proof*⟩

lemma *nslexordp-imp-eq*:
nslexordp cmp xs ys $\Longrightarrow$ *nslexordeqp cmp xs ys*
⟨*proof*⟩

lemma *nslexordeqp-imp-eq-or-less*:
nslexordeqp cmp xs ys $\Longrightarrow$ *xs* = *ys* ∨ *nslexordp cmp xs ys*
⟨*proof*⟩

18.2 Simplification

lemma *nslexord-Nil-left*[*simp*]: ([], *y*) ∉ *nslexord r*
⟨*proof*⟩

lemma *nslexord-Nil-right*[*simp*]: (*y*, []) ∈ *nslexord r* = (∃ *a x*. *y* = *a* # *x*)
⟨*proof*⟩

lemma *nslexord-cons-cons*[*simp*]:
(*a* # *x*, *b* # *y*) ∈ *nslexord r* ⟷ (*a*, *b*) ∈ *r* ∨ (*a* = *b* ∧ (*x*, *y*) ∈ *nslexord r*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *nslexordp-cons-cons*[*simp*]:
nslexordp r (*a* # *x*) (*b* # *y*) ⟷ *r a b* ∨ (*a* = *b* ∧ *nslexordp r x y*)
⟨*proof*⟩

lemmas *nslexord-simps* = *nslexord-Nil-left nslexord-Nil-right nslexord-cons-cons*

lemma *nslexord-same-pref-iff*:
(*xs* @ *ys*, *xs* @ *zs*) ∈ *nslexord r* ⟷ (∃ *x* ∈ *set xs*. (*x*, *x*) ∈ *r*) ∨ (*ys*, *zs*) ∈ *nslexord r*
⟨*proof*⟩

lemma *nslexord-same-pref-if-irrefl*[*simp*]:
irrefl r $\Longrightarrow$ (*xs* @ *ys*, *xs* @ *zs*) ∈ *nslexord r* ⟷ (*ys*, *zs*) ∈ *nslexord r*
⟨*proof*⟩

lemma *nslexord-append-leftI*:
∃ *b z*. *y* = *b* # *z* $\Longrightarrow$ (*x* @ *y*, *x*) ∈ *nslexord r*
⟨*proof*⟩

lemma *nslexord-append-left-rightI*:
(*a* ,*b*) ∈ *r* $\Longrightarrow$ (*u* @ *a* # *x*, *u* @ *b* # *y*) ∈ *nslexord r*
⟨*proof*⟩

lemma *nslexord-append-rightI*:
(*u*, *v*) ∈ *nslexord r* $\Longrightarrow$ (*x* @ *u*, *x* @ *v*) ∈ *nslexord r*
⟨*proof*⟩

lemma *nslexord-append-rightD*:
$[\![(x @ u, x @ v) \in nslexord\ r; (\forall a. (a,a) \notin r)]\!] \Longrightarrow (u,v) \in nslexord\ r$
⟨*proof*⟩
lemma *nslexord-lex*:
$(x,y) \in lex\ r = ((x,y) \in nslexord\ r \wedge length\ x = length\ y)$
⟨*proof*⟩

18.3 Recursive version

fun *nslexordrec* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$
where
nslexordrec P [] - = False |
nslexordrec P - [] = True |
nslexordrec P (x#xs) (y#ys) = (if P x y then True else if x = y then nslexordrec P xs ys else False)

lemma *nslexordp-eq-nslexordrec*:
nslexordp cmp xs ys $\longleftrightarrow$ *nslexordrec cmp xs ys*
⟨*proof*⟩

lemmas *nslexordp-induct = nslexordrec.induct*

18.4 Properties

Useful properties for proving things about relations, such as what type of order is satisfied

lemma *nslexord-total-on*:
assumes *total-on A R*
shows *total-on* $\{xs.\ set\ xs \subseteq A\}$ *(nslexord R)*
⟨*proof*⟩

lemma *total-on-totalp-on-eq*:
total-on A $\{(x, y).\ R\ x\ y\}$ *= totalp-on A R*
⟨*proof*⟩

lemmas *nslexordp-totalp-on =*
nslexord-total-on[OF total-on-totalp-on-eq[THEN iffD2],
simplified nslexord-eq-nslexordp(1) totalp-on-total-on-eq[symmetric]]

lemma *nslexord-total*:
total r $\Longrightarrow$ *total (nslexord r)*
⟨*proof*⟩

lemma *nslexordp-totalp*:
totalp r $\Longrightarrow$ *totalp (nslexordp r)*
⟨*proof*⟩

corollary *nslexord-linear*:

(∀ *a b*. (*a*,*b*) ∈ *r* ∨ *a* = *b* ∨ (*b*,*a*) ∈ *r*) ⟹ (*x*,*y*) ∈ *nslexord r* ∨ *x* = *y* ∨ (*y*,*x*) ∈ *nslexord r*
⟨*proof*⟩

lemma *nslexord-irrefl-on*:
assumes *irrefl-on A R*
shows *irrefl-on* {*xs*. *set xs* ⊆ *A*} (*nslexord R*)
⟨*proof*⟩

lemma *irrefl-on-irreflp-on-eq*:
irrefl-on A {(*x*, *y*). *R x y*} = *irreflp-on A R*
⟨*proof*⟩

lemmas *nslexordp-irreflp-on* =
nslexord-irrefl-on[*OF irrefl-on-irreflp-on-eq*[*THEN iffD2*],
simplified nslexord-eq-nslexordp(*1*) *irreflp-on-irrefl-on-eq*[*symmetric*]]

lemma *nslexord-irreflexive*:
∀ *x*. (*x*,*x*) ∉ *r* ⟹ (*xs*,*xs*) ∉ *nslexord r*
⟨*proof*⟩

lemma *nslexord-irrefl*:
irrefl R ⟹ *irrefl* (*nslexord R*)
⟨*proof*⟩

lemma *nslexordp-irreflp*:
assumes *irreflp R*
shows *irreflp* (*nslexordp R*)
⟨*proof*⟩

lemma *asym-on-asymp-on-eq*:
asym-on A {(*x*, *y*). *R x y*} = *asymp-on A R*
⟨*proof*⟩

lemma *nslexord-asym-on*:
assumes *asym-on A R*
shows *asym-on* {*xs*. *set xs* ⊆ *A*} (*nslexord R*)
⟨*proof*⟩

lemmas *nslexordp-asymp-on* =
nslexord-asym-on[*OF asym-on-asymp-on-eq*[*THEN iffD2*],
simplified nslexord-eq-nslexordp(*1*) *asymp-on-asym-on-eq*[*symmetric*]]

lemma *nslexord-asym*:
assumes *asym R*
shows *asym* (*nslexord R*)
⟨*proof*⟩

lemma *nslexordp-asymp*:

assumes *asymp R*
shows *asymp* (*nslexordp R*)
⟨*proof*⟩

lemma *nslexord-asymmetric*:
assumes *asym R* (*a*, *b*) ∈ *nslexord R*
shows (*b*, *a*) ∉ *nslexord R*
⟨*proof*⟩

lemma *trans-on-transp-on-eq*:
trans-on A {(*x*, *y*). *R x y*} = *transp-on A R*
⟨*proof*⟩

lemma *nslexord-trans-on*:
assumes *trans-on A R*
shows *trans-on* {*xs*. *set xs* ⊆ *A*} (*nslexord R*)
⟨*proof*⟩

lemmas *nslexordp-transp-on* =
nslexord-trans-on[*OF trans-on-transp-on-eq*[*THEN iffD2*],
simplified nslexord-eq-nslexordp(*1*) *transp-on-trans-on-eq*[*symmetric*]]

lemma *nslexord-trans*:
assumes *trans R*
shows *trans* (*nslexord R*)
⟨*proof*⟩

lemma *nslexordp-transp*:
assumes *transp R*
shows *transp* (*nslexordp R*)
⟨*proof*⟩

18.5 Monotonicity

Properties about monotonicity

lemma *monotone-on-nslexordp*:
assumes *monotone-on A orda ordb f*
shows *monotone-on* {*xs*. *set xs* ⊆ *A*} (*nslexordp orda*) (*nslexordp ordb*) (*map f*)
⟨*proof*⟩

lemma *monotone-on-bij-betw-inv-nslexordp*:
assumes *monotone-on A orda ordb f*
and *asymp-on A orda*
and *totalp-on A orda*
and *asymp-on B ordb*
and *totalp-on B ordb*
and *bij-betw f A B*
and *bij-betw g B A*
and *inverses-on f g A B*

shows *monotone-on* $\{xs.\ set\ xs \subseteq B\}$ (*nslexordp ordb*) (*nslexordp orda*) (*map g*)
⟨*proof*⟩

lemma *monotone-on-bij-betw-nslexordp*:
assumes *monotone-on A orda ordb f*
and *asymp-on A orda*
and *totalp-on A orda*
and *asymp-on B ordb*
and *totalp-on B ordb*
and *bij-betw f A B*
shows $\exists g.$ *bij-betw* (*map g*) $\{xs.\ set\ xs \subseteq B\}$ $\{xs.\ set\ xs \subseteq A\}$ $\wedge$
inverses-on (*map f*) (*map g*) $\{xs.\ set\ xs \subseteq A\}$ $\{xs.\ set\ xs \subseteq B\}$ $\wedge$
monotone-on $\{xs.\ set\ xs \subseteq B\}$ (*nslexordp ordb*) (*nslexordp orda*) (*map g*)
⟨*proof*⟩

lemma *monotone-on-iff-nslexordp*:
assumes *monotone-on A orda ordb f*
and *asymp-on A orda*
and *totalp-on A orda*
and *asymp-on B ordb*
and *totalp-on B ordb*
and *bij-betw f A B*
and *set xs* $\subseteq A$
and *set ys* $\subseteq A$
shows *nslexordp orda xs ys* $\longleftrightarrow$ *nslexordp ordb* (*map f xs*) (*map f ys*)
⟨*proof*⟩

18.6 Other

lemma *nslexordp-cons1-exE*:
assumes *nslexordp cmp xs* (x # xs)
shows $\exists a\ as\ bs.\ x \# xs = as\ @\ x \# a \# bs \wedge cmp\ a\ x \wedge (\forall b \in set\ as.\ b = x)$
⟨*proof*⟩

lemma *nslexordp-cons2-exE*:
assumes *nslexordp cmp* (x # xs) *xs*
shows $(\forall k \in set\ xs.\ k = x) \vee (\exists a\ as\ bs.\ x \# xs = as\ @\ x \# a \# bs \wedge cmp\ x\ a$
$\wedge (\forall b \in set\ as.\ b = x))$
⟨*proof*⟩

19 Order definitions on lists of linorder elements

definition *list-less-ns* :: ($'a$ :: *linorder*) *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ *bool*
where
list-less-ns xs ys =
$(\exists n.\ n \le length\ xs \wedge n \le length\ ys \wedge$
$(\forall i < n.\ xs\ !\ i = ys\ !\ i) \wedge$
$(length\ ys = n \longrightarrow n < length\ xs) \wedge$
$(length\ ys \neq n \longrightarrow length\ xs \neq n \wedge xs\ !\ n < ys\ !\ n))$

definition *list-less-eq-ns* :: ('*a* :: *linorder*) *list* ⇒ '*a list* ⇒ *bool*
where
list-less-eq-ns xs ys =
(∃ *n*. *n* ≤ *length xs* ∧ *n* ≤ *length ys* ∧
(∀ *i* < *n*. *xs* ! *i* = *ys* ! *i*) ∧
(*length ys* ≠ *n* ⟶ *length xs* ≠ *n* ∧ *xs* ! *n* < *ys* ! *n*))

— Alternative definition

definition *list-less-ns-ex* :: ('*a* :: *linorder*) *list* ⇒ ('*a* :: *linorder*) *list* ⇒ *bool*
where
list-less-ns-ex xs ys ⟷
(∃ *b c as bs cs*. *xs* = *as* @ *b* # *bs* ∧ *ys* = *as* @ *c* # *cs* ∧ *b* < *c*) ∨
(∃ *c cs*. *xs* = *ys* @ *c* # *cs*)

20 Helper list comparison theorems

lemma *list-less-ns-alt-def*:
list-less-ns xs ys = *list-less-ns-ex xs ys*
⟨*proof*⟩

lemma *nslexordp-eq-list-less-ns-ex*:
nslexordp (<) = *list-less-ns-ex*
⟨*proof*⟩

lemma *nslexordp-eq-list-less-ns-ex-apply*:
nslexordp (<) *x y* = *list-less-ns-ex x y*
⟨*proof*⟩

lemma *nslexordp-eq-list-less-ns*:
nslexordp (<) = *list-less-ns*
⟨*proof*⟩

lemma *nslexordp-eq-list-less-ns-app*:
nslexordp (<) *x y* = *list-less-ns x y*
⟨*proof*⟩

lemma *nslexordeqp-eq-list-less-eq-ns-apply*:
nslexordeqp (<) *x y* = *list-less-eq-ns x y*
⟨*proof*⟩

21 *list-less-ns* helpers

lemma *list-less-ns-cons-same*:
list-less-ns (*a* # *xs*) (*a* # *ys*) = *list-less-ns xs ys*
⟨*proof*⟩

lemma *list-less-ns-cons-diff*:
$a < b \Longrightarrow$ *list-less-ns* $(a \# xs)$ $(b \# ys)$
⟨*proof*⟩

lemma *list-less-ns-cons*:
list-less-ns $(a \# xs)$ $(b \# ys) = (a \leq b \wedge (a = b \longrightarrow$ *list-less-ns xs ys*$))$
⟨*proof*⟩

lemma *list-less-eq-ns-cons-same*:
list-less-eq-ns $(a \# xs)$ $(a \# ys) =$ *list-less-eq-ns xs ys*
⟨*proof*⟩

lemma *list-less-eq-ns-cons*:
list-less-eq-ns $(a \# xs)$ $(b \# ys) = (a \leq b \wedge (a = b \longrightarrow$ *list-less-eq-ns xs ys*$))$
⟨*proof*⟩

lemma *list-less-ns-hd-same*:
$[\![hd\ xs = hd\ ys;\ xs \neq [\,];\ ys \neq [\,]]\!] \Longrightarrow$ *list-less-ns xs ys* $=$ *list-less-ns* $(tl\ xs)$ $(tl\ ys)$
⟨*proof*⟩

lemma *list-less-ns-recurse*:
$[\![xs \neq [\,];\ ys \neq [\,]]\!] \Longrightarrow$
$(hd\ xs = hd\ ys \longrightarrow$ *list-less-ns xs ys* $=$ *list-less-ns* $(tl\ xs)$ $(tl\ ys)) \wedge$
$(hd\ xs \neq hd\ ys \longrightarrow$ *list-less-ns xs ys* $= (hd\ xs < hd\ ys))$
⟨*proof*⟩

lemma *list-less-ns-nil*:
$xs \neq [\,] \Longrightarrow$ *list-less-ns xs* $[\,]$
⟨*proof*⟩

lemma *list-less-ns-app*:
$bs \neq [\,] \Longrightarrow$ *list-less-ns* $(as\ @\ bs)$ as
⟨*proof*⟩

22 Lists of linorder elements are linorders with a bottom element

lemma *list-less-ns-imp-less-eq-not-less-eq*:
list-less-ns x y $\Longrightarrow$ (*list-less-eq-ns x y* $\wedge \neg$ *list-less-eq-ns y x*)
⟨*proof*⟩

lemma *list-less-eq-ns-not-less-eq-imp-less*:
list-less-eq-ns x y $\wedge \neg$ *list-less-eq-ns y x* $\Longrightarrow$ *list-less-ns x y*
⟨*proof*⟩

lemma *list-less-eq-ns-trans*:

$[\![$*list-less-eq-ns x y*; *list-less-eq-ns y z*$]\!] \Longrightarrow$ *list-less-eq-ns x z*
⟨*proof*⟩

lemma *list-less-eq-ns-anti-sym*:
$[\![$*list-less-eq-ns x y*; *list-less-eq-ns y x*$]\!] \Longrightarrow x = y$
⟨*proof*⟩

lemma *list-less-eq-ns-linear*:
list-less-eq-ns x y ∨ *list-less-eq-ns y x*
⟨*proof*⟩

interpretation *ordlistns*: *linorder list-less-eq-ns list-less-ns*
⟨*proof*⟩

interpretation *ordlistns*: *order-top list-less-eq-ns list-less-ns* []
⟨*proof*⟩

23 Recursive Definition

fun *lt-ns* :: (*'a* :: *linorder*) *list* ⇒ *'a list* ⇒ *bool*
where
lt-ns [] [] = *False* |
lt-ns [] - = *False* |
lt-ns - [] = *True* |
lt-ns (*a* # *as*) (*b* # *bs*) =
(*if a* < *b then True*
else if a > *b then False*
else lt-ns as bs)

lemma *list-less-ns-lt-ns*:
list-less-ns xs ys = *lt-ns xs ys*
⟨*proof*⟩

24 *list-less-ns-ex* helpers

lemma *list-less-ns-exI1*:
∃ *b c as bs cs*. *xs* = *as* @ *b* # *bs* ∧ *ys* = *as* @ *c* # *cs* ∧ *b* < *c* ⟹ *list-less-ns-ex xs ys*
⟨*proof*⟩

lemma *list-less-ns-exI2*:
∃ *c cs*. *xs* = *ys* @ *c* # *cs* ⟹ *list-less-ns-ex xs ys*
⟨*proof*⟩

lemma *list-less-ns-exE*:
list-less-ns-ex xs ys ⟹
(∃ *b c as bs cs*. *xs* = *as* @ *b* # *bs* ∧ *ys* = *as* @ *c* # *cs* ∧ *b* < *c*) ∨
(∃ *c cs*. *xs* = *ys* @ *c* # *cs*)

⟨proof⟩

lemma *list-less-ns-app-same*:
list-less-ns (*as* @ *xs*) (*as* @ *ys*) = *list-less-ns xs ys*
⟨proof⟩

lemma *list-less-eq-ns-app-same*:
list-less-eq-ns (*as* @ *xs*) (*as* @ *ys*) = *list-less-eq-ns xs ys*
⟨proof⟩

lemma *list-less-ns-cons1-exE*:
assumes *list-less-ns xs* (*x* # *xs*)
shows $\exists$ *a as bs*. *x* # *xs* = *as* @ *x* # *a* # *bs* $\wedge$ *x* > *a* $\wedge$ ($\forall$ *b* $\in$ *set as*. *b* = *x*)
⟨proof⟩

lemma *list-less-ns-cons1-exI*:
assumes $\exists$ *a as bs*. *x* # *xs* = *as* @ *x* # *a* # *bs* $\wedge$ *x* > *a* $\wedge$ ($\forall$ *b* $\in$ *set as*. *b* = *x*)
shows *list-less-ns-ex xs* (*x* # *xs*)
⟨proof⟩

lemma *list-less-ns-cons2-ex*:
assumes *list-less-ns* (*x* # *xs*) *xs*
shows ($\forall$ *k* $\in$ *set xs*. *k* = *x*) $\vee$ ($\exists$ *a as bs*. *x* # *xs* = *as* @ *x* # *a* # *bs* $\wedge$ *x* < *a* $\wedge$ ($\forall$ *b* $\in$ *set as*. *b* = *x*))
⟨proof⟩

end
theory *Valid-List*
imports *Main ../util/List-Util*
begin

25 Valid List

definition
valid-list :: ('*a* :: {*linorder*, *order-bot*}) *list* $\Rightarrow$ *bool*
where
valid-list s = (*length s* > *0* $\wedge$ ($\forall$ *i* < *length s* − *1*. *s* ! *i* $\neq$ *bot*) $\wedge$ *last s* = *bot*)

lemma *valid-list-ex-def*:
fixes *s* ::('*a* :: {*linorder*, *order-bot*}) *list*
shows (*valid-list s*) =
($\exists$ *xs*. *s* = *xs* @ [*bot*] $\wedge$
($\forall$ *i* < *length xs*. *xs* ! *i* $\neq$ *bot*))
⟨proof⟩

lemma *valid-list-iff-butlast-app-last*:
fixes *s* :: ('*a* :: {*linorder*, *order-bot*}) *list*
shows *valid-list s* $\longleftrightarrow$
s $\neq$ [] $\wedge$

$(\forall x \in set\ (butlast\ s).\ x \neq bot) \wedge$
$last\ s = bot$
⟨*proof*⟩

lemma *valid-list-consI*:
fixes $s :: ('a :: \{linorder,\ order\text{-}bot\})\ list$
fixes $a :: 'a$
assumes *valid-list s*
and $a \neq bot$
shows *valid-list* $(a \# s)$
⟨*proof*⟩

lemma *valid-list-consD*:
fixes $s :: ('a :: \{linorder,\ order\text{-}bot\})\ list$
fixes $a :: 'a$
assumes *valid-list* $(a \# s)$
assumes $s \neq []$
shows *valid-list s*
⟨*proof*⟩

lemma *Min-valid-list*:
fixes $s :: ('a :: \{linorder,\ order\text{-}bot\})\ list$
assumes *valid-list s*
shows $Min\ (set\ s) = bot$
⟨*proof*⟩

lemma *valid-list-length*:
fixes $s :: ('a :: \{linorder,\ order\text{-}bot\})\ list$
assumes *valid-list s*
shows $length\ s > 0$
⟨*proof*⟩

lemma *valid-list-length-ex*:
fixes $s :: ('a :: \{linorder,\ order\text{-}bot\})\ list$
assumes *valid-list s*
shows $\exists n.\ length\ s = Suc\ n$
⟨*proof*⟩

lemma *valid-list-not-nil*:
fixes $s :: ('a :: \{linorder,\ order\text{-}bot\})\ list$
assumes *valid-list s*
shows $s \neq []$
⟨*proof*⟩

lemma *valid-list-Suc-mapping*:
fixes $f :: 'a \Rightarrow nat$
fixes $s :: 'a\ list$
shows *valid-list* $((map\ (\lambda x.\ Suc\ (f\ x))\ s)\ @\ [bot])$
⟨*proof*⟩

lemma *valid-list-app*:
  assumes *valid-list* (*xs* @ *y* # *ys*)
  shows *valid-list* (*y* # *ys*)
  ⟨*proof*⟩

lemma *not-valid-list-app*:
  assumes *valid-list* (*xs* @ *y* # *ys*)
  shows ¬*valid-list xs*
  ⟨*proof*⟩

lemma *valid-list-neqE*:
  assumes *valid-list xs valid-list ys xs* ≠ *ys*
  shows ∃ *x y as bs cs*. *xs* = *as* @ *x* # *bs* ∧ *ys* = *as* @ *y* # *cs* ∧ *x* ≠ *y*
⟨*proof*⟩

end
theory *Valid-List-Util*
  imports *List-Lexorder-Util List-Lexorder-NS Valid-List*
begin

26 Order Equivalence

lemma *valid-list-list-less-equiv-list-less-ns*:
  assumes *valid-list s1*
  and *valid-list s2*
shows *s1* < *s2* = *list-less-ns s1 s2*
⟨*proof*⟩

lemma *valid-list-list-less-eq-equiv-list-less-eq-ns*:
  assumes *valid-list s1*
  and *valid-list s2*
shows *s1* ≤ *s2* = *list-less-eq-ns s1 s2*
  ⟨*proof*⟩

27 Classical Lexicographical Order

lemma *valid-list-list-less-imp*:
  assumes *valid-list* (*xs* @ [*bot*])
  and *valid-list* (*ys* @ [*bot*])
  and (*xs* @ [*bot*]) < (*ys* @ [*bot*])
shows *xs* < *ys*
⟨*proof*⟩

lemma *strict-mono-on-list-less-map*:
  fixes α :: ′*a* :: *preorder* ⇒ ′*b* :: *ord*
  assumes *strict-mono-on A* α
  and *set xs* ⊆ *A*

and *set ys* $\subseteq A$
and $xs < ys$
shows $(map\ \alpha\ xs) < (map\ \alpha\ ys)$
$\langle proof \rangle$

lemma *strict-mono-list-less-map*:
assumes *strict-mono* α
and $xs < ys$
shows $map\ \alpha\ xs < map\ \alpha\ ys$
$\langle proof \rangle$

lemma *strict-mono-on-map-list-less*:
fixes $\alpha :: 'a :: linorder \Rightarrow 'b :: order$
assumes *strict-mono-on* $A\ \alpha$
and *set xs* $\subseteq A$
and *set ys* $\subseteq A$
and $(map\ \alpha\ xs) < (map\ \alpha\ ys)$
shows $xs < ys$
$\langle proof \rangle$

lemma *strict-mono-map-list-less*:
fixes $\alpha :: 'a :: linorder \Rightarrow 'b :: order$
assumes *strict-mono* α
and $(map\ \alpha\ xs) < (map\ \alpha\ ys)$
shows $xs < ys$
$\langle proof \rangle$

28 Non-standard Lexicographical Ordering

lemma *sorted-list-less-ns*:
assumes *sorted* $(a \# bs @ [c])$
and $c < d$
shows *list-less-ns* $(a \# bs @ [c, d] @ xs)$ $(bs @ [c, d] @ ys)$
$\langle proof \rangle$

lemma *rev-sorted-list-less-ns*:
assumes *sorted* $(rev\ (a \# bs @ [c]))$
and $c > d$
shows *list-less-ns* $(bs @ [c, d] @ xs)$ $(a \# bs @ [c, d] @ ys)$
$\langle proof \rangle$

lemma *sorted-cons-list-less-ns*:
assumes *sorted* $(a \# bs)$
shows *list-less-ns* $(a \# bs)$ bs
$\langle proof \rangle$

end
theory *Suffix*
imports *Main*

begin

29 Suffix

abbreviation *suffix* :: $'a$ *list* ⇒ *nat* ⇒ $'a$ *list*
  where
suffix xs i ≡ *drop i xs*

lemma *suffixes-neq*:
  ⟦*i* < *length s*; *j* < *length s*; *i* ≠ *j*⟧ ⟹ *suffix s i* ≠ *suffix s j*
  ⟨*proof*⟩

lemma *distinct-suffixes*:
  ⟦*distinct xs*; ∀ *x* ∈ *set xs*. *x* < *length s*⟧ ⟹ *distinct* (*map* (*suffix s*) *xs*)
  ⟨*proof*⟩

lemma *suffix-eq-index*:
  ⟦*i* < *length xs*; *j* < *length xs*; *suffix xs i* = *suffix xs j*⟧ ⟹ *i* = *j*
  ⟨*proof*⟩

lemma *suffix-neq-nil*:
  i < *length s* ⟹ *suffix s i* ≠ []
  ⟨*proof*⟩

lemma *suffix-map*:
  suffix (*map f xs*) *i* = *map f* (*suffix xs i*)
  ⟨*proof*⟩

lemma *set-suffix-subset*:
  set (*suffix s i*) ⊆ *set s*
  ⟨*proof*⟩
lemma *suffix-cons-suc*:
  suffix (*a* # *xs*) (*Suc i*) = *suffix xs i*
  ⟨*proof*⟩

lemma *suffix-app*:
  i < *length xs* ⟹ *suffix* (*xs* @ *ys*) *i* = *suffix xs i* @ *ys*
  ⟨*proof*⟩

lemma *suffix-cons-ex*:
  i < *length T* ⟹ ∃ *x xs*. *suffix T i* = *x* # *xs* ∧ *x* = *T* ! *i*
  ⟨*proof*⟩

lemma *suffix-cons-Suc*:
  i < *length T* ⟹ *suffix T i* = *T* ! *i* # *suffix T* (*Suc i*)
  ⟨*proof*⟩

lemma *suffix-cons-app*:
  suffix T i = *as* @ *bs* ⟹ *suffix T* (*i* + *length as*) = *bs*

⟨*proof*⟩

lemma *suffix-0*:
suffix T 0 = T
⟨*proof*⟩

end
theory *Suffix-Util*
imports
../util/List-Slice
Suffix
Valid-List
Valid-List-Util

begin

30 Valid Lists and Suffixes

lemma *valid-suffix*:
$[\![$*valid-list s*; $i <$ *length s*$]\!] \Longrightarrow$ *valid-list* (*suffix s i*)
⟨*proof*⟩

lemma *last-suffix-index*:
assumes *valid-list s*
and $i <$ *length s*
shows *hd* (*suffix s i*) = *bot* $\longleftrightarrow$ $i =$ *length s* $- 1$
⟨*proof*⟩

31 Prefixes and Suffixes

lemma *suffix-has-no-prefix-suffix*:
assumes *valid-list*: *valid-list s*
and *i-less-len-s*: $i <$ *length s*
and *j-less-len-s*: $j <$ *length s*
and *i-neq-j*: $i \neq j$
shows $\neg$ ($\exists s'$. *suffix s i* = (*suffix s j*) @ s')
⟨*proof*⟩

32 Suffix Comparisons

32.1 Lexicographical Ordering

lemma *suffix-less-ex*:
fixes *s* :: (*'a* :: {*linorder*, *order-bot*}) *list*
assumes *valid-list s*
and $i <$ *length s*
and $j <$ *length s*
and *suffix s i* $<$ *suffix s j*

shows $\exists\, b\ c\ as\ bs\ cs.\ suffix\ s\ i = as\ @\ b\ \#\ bs\ \wedge$
$suffix\ s\ j = as\ @\ c\ \#\ cs \wedge b < c$
⟨*proof*⟩

lemma *suffix-less-nth*:
assumes *valid-list s*
and $i < length\ s$
and $j < length\ s$
and $suffix\ s\ i < suffix\ s\ j$
shows
$\exists\, n.\ n < length\ (suffix\ s\ i) \wedge$
$n < length\ (suffix\ s\ j) \wedge$
$(\forall\, k < n.\ (suffix\ s\ i)\ !\ k = (suffix\ s\ j)\ !\ k) \wedge$
$(suffix\ s\ i)\ !\ n < (suffix\ s\ j)\ !\ n$
⟨*proof*⟩

lemma *suffix-less-butlast*:
assumes *valid-list s*
and $i < length\ s$
and $j < length\ s$
and $suffix\ s\ i < suffix\ s\ j$
shows $butlast\ (suffix\ s\ i) < butlast\ (suffix\ s\ j)$
⟨*proof*⟩

32.2 Non-standard List Ordering

lemma *suffix-less-ns-ex*:
assumes *valid-list s*
and $i < length\ s$
and $j < length\ s$
and *list-less-ns* (*suffix s i*) (*suffix s j*)
shows $\exists\, b\ c\ as\ bs\ cs.$
$suffix\ s\ i = as\ @\ b\ \#\ bs\ \wedge$
$suffix\ s\ j = as\ @\ c\ \#\ cs \wedge b < c$
⟨*proof*⟩

lemma *suffix-less-ns-nth*:
assumes *valid-list s*
and $i < length\ s$
and $j < length\ s$
and *list-less-ns* (*suffix s i*) (*suffix s j*)
shows
$\exists\, n.\ n < length\ (suffix\ s\ i) \wedge$
$n < length\ (suffix\ s\ j) \wedge$
$(\forall\, k < n.\ (suffix\ s\ i)\ !\ k = (suffix\ s\ j)\ !\ k) \wedge$
$(suffix\ s\ i)\ !\ n < (suffix\ s\ j)\ !\ n$
⟨*proof*⟩

33 List Slice

declare *list-slice.simps*[*simp del*]

lemma *list-slice-to-suffix*:
list-slice T i j = *take* (*j* − *i*) (*suffix T i*)
⟨*proof*⟩

lemma *suffix-eq-list-slice*:
suffix T i = *list-slice T i* (*length T*)
⟨*proof*⟩

lemma *list-slice-suffix*:
list-slice T i j = *list-slice* (*suffix T i*) *0* (*j* − *i*)
⟨*proof*⟩

lemma *suffix-to-list-slice-app*:
i ≤ *j* ⟹ *suffix T i* = (*list-slice T i j*) @ (*list-slice T j* (*length T*))
⟨*proof*⟩

34 Sorting

lemma *ordlist-strict-mono-strict-sorted-1*:
assumes *strict-mono* α
and *strict-sorted* (*map* (*suffix* (*map* α *s*)) *xs*)
shows *strict-sorted* (*map* (*suffix s*) *xs*)
⟨*proof*⟩

lemma *ordlist-strict-mono-on-strict-sorted-1*:
assumes *strict-mono-on A* α
and *set s* ⊆ *A*
and *strict-sorted* (*map* (*suffix* (*map* α *s*)) *xs*)
shows *strict-sorted* (*map* (*suffix s*) *xs*)
⟨*proof*⟩

lemma *ordlist-strict-mono-strict-sorted-2*:
assumes *strict-mono* α
and *strict-sorted* (*map* (*suffix s*) *xs*)
shows *strict-sorted* (*map* (*suffix* (*map* α *s*)) *xs*)
⟨*proof*⟩

lemma *ordlist-strict-mono-on-strict-sorted-2*:
assumes *strict-mono-on A* α
and *set s* ⊆ *A*
and *strict-sorted* (*map* (*suffix s*) *xs*)
shows *strict-sorted* (*map* (*suffix* (*map* α *s*)) *xs*)
⟨*proof*⟩

lemma *valid-list-ordlist-ordlistns-strict-sorted-eq*:
assumes *valid-list T*

and *set xs* ⊆ {*0*..<*length T*}
shows *ordlistns.strict-sorted* (*map* (*suffix T*) *xs*) ⟷
strict-sorted (*map* (*suffix T*) *xs*)
⟨*proof*⟩

lemma *Min-valid-suffix*:
assumes *valid-list T*
and *length T* = *Suc n*
shows *ordlistns.Min* {*suffix T i* |*i*. *i* < *length T*} = *suffix T n*
⟨*proof*⟩

end
theory *Prefix*
imports *Main*
begin

35 Prefix Definition

abbreviation *prefix* :: *'a list* ⇒ *nat* ⇒ *'a list*
where
prefix xs i ≡ *take i xs*

lemma *prefix-neq*:
assumes *i* < *length s*
and *j* < *length s*
and *i* ≠ *j*
shows *prefix s i* ≠ *prefix s j*
⟨*proof*⟩

lemma *not-prefix-app*:
(∀ *k*. *s1* ≠ *prefix s2 k*) ⟷ (∀ *xs*. *s2* ≠ *s1* @ *xs*)
⟨*proof*⟩

lemma *not-prefix-imp-not-nil*:
∀ *k*. *s1* ≠ *prefix s2 k* ⟹ *s1* ≠ []
⟨*proof*⟩

end
theory *Prefix-Util*
imports *Prefix ../order/Suffix-Util*
begin

lemma *prefix-suffix-not-suffix*:
assumes *valid-list s*
and *i* < *length s*
and *j* < *length s*
and *i* ≠ *j*
shows ¬(∃ *k*. *prefix* (*suffix s i*) *k* = *suffix s j*)
⟨*proof*⟩

end
theory *Suffix-Array*
imports
../util/Sorting-Util
../order/List-Lexorder-Util
../order/Suffix
../order/Valid-List
../order/List-Permutation-Util
begin

36 Axiomatic Suffix Array Specification

locale *Suffix-Array-General* =
fixes *sa* :: (*'a* :: {*linorder*, *order-bot*}) *list* ⇒ *nat list*
assumes *sa-g-permutation*: *sa s* <~~> [*0*..<*length s*]
and *sa-g-sorted*: *strict-sorted* (*map* (*suffix s*) (*sa s*))

locale *Suffix-Array-Restricted* =
fixes *sa* :: *nat list* ⇒ *nat list*
assumes *sa-r-permutation*: *valid-list s* ⟹ *sa s* <~~> [*0*..<*length s*]
and *sa-r-sorted*: *valid-list s* ⟹ *strict-sorted* (*map* (*suffix s*) (*sa s*))

37 Wrapper for Natural Number String only Algorithm

definition *sa-nat-wrapper* ::
(*'a* :: *linorder list* ⇒ *'a* ⇒ *nat*) ⇒ (*nat list* ⇒ *nat list*) ⇒ *'a* :: *linorder list* ⇒ *nat list*
where
sa-nat-wrapper α *sa xs* =
tl (*sa* ((*map* (λ*x*. *Suc* (α *xs x*)) *xs*) @ [*bot*]))

end
theory *Suffix-Array-Properties*
imports
../util/Fun-Util
../order/Suffix-Util
Suffix-Array

begin

38 General Suffix Array Properties

context *Suffix-Array-General* **begin**

lemma *sa-length*:

length (*sa s*) = *length s*
⟨*proof*⟩

lemma *sa-distinct*:
distinct (*sa s*)
⟨*proof*⟩

lemma *sa-set-upt*:
set (*sa s*) = {*0*..<*length s*}
⟨*proof*⟩

lemma *sa-nth-ex*:
i < *length s* ⟹ ∃ *k* < *length s*. *sa s* ! *i* = *k*
⟨*proof*⟩

lemma *ex-sa-nth*:
k < *length s* ⟹ ∃ *i* < *length s*. *sa s* ! *i* = *k*
⟨*proof*⟩

end

lemma *Suffix-Array-General-determinism*:
assumes *Suffix-Array-General f*
and *Suffix-Array-General g*
shows *f* = *g*
⟨*proof*⟩

39 Properties of Suffix Arrays on Valid Lists

lemma *valid-list-bot-min*:
assumes *valid-list* (*s* @ [*bot*])
and *sa* (*s* @ [*bot*]) <~~> [*0*..<*length* (*s* @ [*bot*])]
and *strict-sorted* (*map* (*suffix* (*s* @ [*bot*])) (*sa* (*s* @ [*bot*])))
shows ∃ *xs*. *sa* (*s* @ [*bot*]) = *length s* # *xs*
⟨*proof*⟩

lemma *valid-list-bot-perm*:
assumes *valid-list* (*s* @ [*bot*])
and *sa* (*s* @ [*bot*]) <~~> [*0*..<*length* (*s* @ [*bot*])]
and *strict-sorted* (*map* (*suffix* (*s* @ [*bot*])) (*sa* (*s* @ [*bot*])))
shows ∃ *xs*. *sa* (*s* @ [*bot*]) = *length s* # *xs* ∧ *xs* <~~> [*0*..<*length s*]
⟨*proof*⟩

lemma *valid-list-bot-perm-sort*:
assumes *valid-list* (*s* @ [*bot*])
and *sa* (*s* @ [*bot*]) <~~> [*0*..<*length* (*s* @ [*bot*])]
and *strict-sorted* (*map* (*suffix* (*s* @ [*bot*])) (*sa* (*s* @ [*bot*])))
shows ∃ *xs*. *sa* (*s* @ [*bot*]) = *length s* # *xs* ∧ *xs* <~~> [*0*..<*length s*] ∧
strict-sorted (*map* (*suffix s*) *xs*)

⟨*proof*⟩

theorem *Suffix-Array-Restricted-valid-list-bot-perm-sort*:
assumes *valid-list* (*s* @ [*bot*])
and *Suffix-Array-Restricted sa*
shows ∃ *xs*. *sa* (*s* @ [*bot*]) = *length s* # *xs* ∧ *xs* <~~> [*0*..<*length s*] ∧
strict-sorted (*map* (*suffix s*) *xs*)
⟨*proof*⟩

lemma *Suffix-Array-Restricted-wrapper-permutation*:
assumes *Linorder-to-Nat-List* α *s*
and *Suffix-Array-Restricted sa*
shows *sa-nat-wrapper* α *sa s* <~~> [*0*..<*length s*]
⟨*proof*⟩

lemma *Suffix-Array-Restricted-wrapper-sorted*:
assumes *Linorder-to-Nat-List* α *s*
and *Suffix-Array-Restricted sa*
shows *strict-sorted* (*map* (*suffix s*) (*sa-nat-wrapper* α *sa s*))
⟨*proof*⟩

40 Equivalence

lemma *Suffix-Array-General-imp-Restrict*:
Suffix-Array-General sa-nat ⟹ *Suffix-Array-Restricted sa-nat*
⟨*proof*⟩

interpretation *Linorder-to-Nat-List map-to-nat*
⟨*proof*⟩

lemma *Suffix-Array-Restricted-imp-General*:
Suffix-Array-Restricted sa ⟹ *Suffix-Array-General* (*sa-nat-wrapper map-to-nat sa*)
⟨*proof*⟩

lemma *Suffix-Array-General-Restrict-determinism*:
assumes *Suffix-Array-Restricted f*
and *Suffix-Array-General g*
shows *sa-nat-wrapper map-to-nat f* = *g*
⟨*proof*⟩

end
theory *Simple-SACA*
imports
../order/Suffix
../order/List-Lexorder-Util
begin

fun *gen-suffixes* :: (′*a* :: {*linorder*,*order-bot*}) *list* ⇒ ′*a list list*

where
gen-suffixes s = *map* (*suffix s*) [*0*..<(*length s*)]

fun *suffix-ids* :: ($'a$ :: {*linorder*,*order-bot*}) *list* ⇒ $'a$ *list list* ⇒ *nat list*
where
suffix-ids s ss = *map* (λ*x*. *length s* − *length x*) *ss*

fun *simple-saca* :: ($'a$ :: {*linorder*,*order-bot*}) *list* ⇒ *nat list*
where
simple-saca s = *suffix-ids s* (*sort* (*gen-suffixes s*))

end
theory *Simple-SACA-Verification*
imports
Simple-SACA
../spec/Suffix-Array
begin

lemma *suf-length-app*:
i < *length xs* ⟹ *length* (*suffix* (*xs* @ *ys*) *i*) = *length* (*suffix xs i*) + *length ys*
⟨*proof*⟩

lemma *distinct-natlist-add*:
distinct (*xs* :: *nat list*) ⟹ *distinct* (*map* ((+) *n*) *xs*)
⟨*proof*⟩

lemma *nat-minus-cancel-right*:
⟦(*x*::*nat*) ≤ *n*; *y* ≤ *n*; *n* − *x* = *n* − *y*⟧ ⟹ *x* = *y*
⟨*proof*⟩

lemma *distinct-natlist-sub*:
⟦*distinct* (*xs* :: *nat list*); ∀ *x* ∈ *set xs*. *x* ≤ *n*⟧ ⟹ *distinct* (*map* ((−) *n*) *xs*)
⟨*proof*⟩

lemma *map-suf-app*:
n ≤ *length xs* ⟹
map (*length* ∘ *suffix* (*xs* @ *ys*)) [*0*..<*n*] = *map* ((+) (*length ys*)) (*map* (*length* ∘ (*suffix xs*)) [*0*..<*n*])
⟨*proof*⟩

lemma *distinct-map-length-gen-suffixes*:
distinct (*map length* (*gen-suffixes s*))
⟨*proof*⟩

lemma *different-length-different-list*:
length a ∉ *length* ‘ *set xs* ⟹ *a* ∉ *set xs*
⟨*proof*⟩

lemma *distinct-map-length-sort*:

distinct (*map length xs*) $\Longrightarrow$ *distinct* (*map length* (*sort xs*))
⟨*proof*⟩

lemma *suffix-ids-def'*:
suffix-ids s xs = *map* (((−) (*length s*)) ∘ *length*) *xs*
⟨*proof*⟩

lemma *distinct-simple-saca*:
distinct (*simple-saca s*)
⟨*proof*⟩

lemma *suf-suffix-id-suf*:
i < *length s* $\Longrightarrow$ *suffix s* (*length s* − *length* (*suffix s i*)) = *suffix s i*
⟨*proof*⟩

lemma *in-set-ordlist-sort*:
(*x* ∈ *set xs*) = (*x* ∈ *set* (*sort xs*))
⟨*proof*⟩

lemma *ordlist-sort-conv-nth*:
(∃ *i*<*length xs*. *xs* ! *i* = *x*) = (∃ *i*<*length xs*. (*sort xs*) ! *i* = *x*)
⟨*proof*⟩

lemma *ordlist-sort-nth-before*:
⟦*i* < *length xs*; (*sort xs*) ! *i* = *x*⟧ $\Longrightarrow$
∃ *j*<*length xs*. *xs* ! *j* = *x*
⟨*proof*⟩

lemma *suf-sort-suf-nth*:
i < *length s* $\Longrightarrow$
suffix s (*length s* − *length* ((*sort* (*gen-suffixes s*)) ! *i*)) =
sort (*gen-suffixes s*) ! *i*
⟨*proof*⟩

lemma *map-suf-simple-saca*:
map (*suffix s*) (*simple-saca s*) = *sort* (*gen-suffixes s*)
⟨*proof*⟩

interpretation *simple-saca*: *Suffix-Array-General simple-saca*
⟨*proof*⟩

end
theory *List-Type*
imports
../../util/Nat-Util
../../util/Set-Util
../../util/Fun-Util
../../util/List-Util
../../order/Suffix-Util

../../order/Valid-List-Util
../../spec/Suffix-Array-Properties
begin

This theory file contains the background theory for the SAIS algorithm (Nong et al., DCC 2009), which is essentially an optimisation of the KA algorithm (Ko et al, JDA 2005).

41 Small and Large List Types

datatype *SL-types* = *S-type* | *L-type*

This section contains a generalisation of the suffix types to sequences of any type and any element comparison function that satisfies certain properties given the theorem. Typical constraints involve either one or a combination of *totalp-on*, *irreflp-on*, *transp-on* and *asymp-on*.

definition
list-type :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ SL-types
where
list-type cmp xs =
(*if nslexordp cmp xs* (*suffix xs* (*Suc 0*))
then S-type
else L-type)

lemma *list-type-cons-same*:
⟦*irreflp-on A cmp*; *x* ∈ *A*⟧ ⟹ *list-type cmp* (*x* # *x* # *xs*) = *list-type cmp* (*x* # *xs*)
⟨*proof*⟩

lemma *list-type-nil*:
list-type cmp [] = *L-type*
⟨*proof*⟩

lemma *list-type-singleton*:
list-type cmp [*x*] = *S-type*
⟨*proof*⟩

lemma *list-type-s-type-eq*:
list-type cmp xs = *S-type* ⟷ *nslexordp cmp xs* (*suffix xs* (*Suc 0*))
⟨*proof*⟩

lemma *list-type-l-type-eq*:
list-type cmp xs = *L-type* ⟷ ¬*nslexordp cmp xs* (*suffix xs* (*Suc 0*))
⟨*proof*⟩

lemma *list-type-cons-diff1*:
cmp x y ⟹ *list-type cmp* (*x* # *y* # *xs*) = *S-type*
⟨*proof*⟩

lemma *list-type-cons-diff2*:
  $[\![\neg cmp\ x\ y;\ x \neq y]\!] \Longrightarrow list\text{-}type\ cmp\ (x\ \#\ y\ \#\ xs) = L\text{-}type$
  ⟨*proof*⟩

lemma *list-type-s-neq-nil*:
  $list\text{-}type\ cmp\ xs = S\text{-}type \Longrightarrow xs \neq []$
  ⟨*proof*⟩

lemma *list-type-s-hd-cmp*:
  $list\text{-}type\ cmp\ (x\ \#\ y\ \#\ xs) = S\text{-}type \Longrightarrow cmp\ x\ y \vee x = y$
  ⟨*proof*⟩

lemma *list-type-l-hd-cmp*:
  $list\text{-}type\ cmp\ (x\ \#\ y\ \#\ xs) = L\text{-}type \Longrightarrow \neg cmp\ x\ y \vee x = y$
  ⟨*proof*⟩

lemma *list-type-repl*:
  $[\![irreflp\text{-}on\ A\ cmp;\ x \in A;\ set\ xs = \{x\}]\!] \Longrightarrow list\text{-}type\ cmp\ (x\ \#\ xs) = S\text{-}type$
  ⟨*proof*⟩

lemma *list-type-s-ex*:
  assumes $list\text{-}type\ cmp\ (x\ \#\ xs) = S\text{-}type$
  shows $(\forall a \in set\ xs.\ a = x) \vee (\exists b\ as\ bs.\ x\ \#\ xs = as\ @\ x\ \#\ b\ \#\ bs \wedge cmp\ x\ b$
$\wedge (\forall k \in set\ as.\ k = x))$
⟨*proof*⟩

lemma *list-type-l-type-ex*:
  assumes $list\text{-}type\ cmp\ (x\ \#\ xs) = L\text{-}type$
  and $totalp\text{-}on\ A\ cmp$
  and $x \in A$
  and $set\ xs \subseteq A$
  shows $\exists b\ as\ bs.\ x\ \#\ xs = as\ @\ x\ \#\ b\ \#\ bs \wedge cmp\ b\ x \wedge (\forall k \in set\ as.\ k = x)$
⟨*proof*⟩

theorem *l-less-than-s-type-list-type*:
  assumes $list\text{-}type\ cmp\ (a\ \#\ s1) = S\text{-}type$
  and $list\text{-}type\ cmp\ (a\ \#\ s2) = L\text{-}type$
  and $totalp\text{-}on\ A\ cmp$
  and $transp\text{-}on\ A\ cmp$
  and $a \in A$
  and $set\ s1 \subseteq A$
  and $set\ s2 \subseteq A$
shows $nslexordp\ cmp\ (a\ \#\ s2)\ (a\ \#\ s1)$
⟨*proof*⟩

lemma *list-type-cons-diff-type1*:
  $[\![list\text{-}type\ cmp\ (a\ \#\ b\ \#\ xs) = S\text{-}type;\ list\text{-}type\ cmp\ (b\ \#\ xs) = L\text{-}type]\!] \Longrightarrow$
    $cmp\ a\ b$

⟨*proof*⟩

lemma *list-type-cons-diff-type2*:
⟦*list-type cmp* (*a* # *b* # *xs*) = *L-type*; *list-type cmp* (*b* # *xs*) = *S-type*⟧ ⟹
¬*cmp a b* ∧ *a* ≠ *b*
⟨*proof*⟩

42 Suffix Type

This section contains the suffix type definition.

definition *suffix-type* :: (′*a* :: {*linorder*, *order-bot*}) *list* ⇒ *nat* ⇒ *SL-types*
where
suffix-type s i ≡
(*if list-less-ns* (*suffix s i*) (*suffix s* (*Suc i*)) *then S-type*
else L-type)

lemma *suffix-type-list-type-eq*:
suffix-type xs i = *list-type* (<) (*suffix xs i*)
⟨*proof*⟩

There are two types of suffixes (*SL-types*): *S-type* and *L-type*. An S-type suffix is a suffix that is strictly less than the suffix that occurs immediately after it, and an L-type suffix is a suffix that is strictly greater than the suffix that occurs immediately after it. The definition of less than used here is *list-less-ns*. Note that this definition of less than differs from lexicographical order(*list-less*, i.e. dictionary order, but it is equivalent when the both lists are valid (*valid-list*) as shown in ⟦*valid-list ?s1.0*; *valid-list ?s2.0*⟧ ⟹ (*?s1.0* < *?s2.0*) = *list-less-ns ?s1.0 ?s2.0*. There are three reasons for using the *list-less-ns* definition, and we explain in order of importance.

The first reason is that the original suffix types definition required a special case for the singleton suffix that only contains the sentinel symbol. While this special case makes sense in regards to the algorithms, i.e. it is necessary for the correctness of the algorithms, it does not naturally follow from the intuition of suffix types. In fact, it contradicts the intuitive definition that follows from the lexicographical order *list-less*. That is, a list that only consists of one element is always strictly greater than the empty list. With the alternate definition of less than *list-less-ns*, a proper prefix is always strictly greater, and so, a singleton list will always be strictly less than the empty list. Therefore, there is no need to have a special case for the singleton suffix that only contains the sentinel.

The second reason is that the SAIS algorithm uses a sublist order that depends on the suffix type definition (see Section S̈AIS Sublist Order)̈. This definition is perfectly valid for the algorithm, since the ordering is only used for sublist of the same list. However, the ordering is not easily understandable when applied to arbitrary list, even though it is equivalent to *list-less-ns*, which we prove in a later section. As an ordering, *list-less-ns* is much easier to understand. It is also used within the definition of *suffix-type*. Therefore, it makes more sense to reuse *list-less-ns*, rather than having multiple definitions of the same thing.

The third reason is that the original suffix types definition does not handle the case where the suffix is not terminated by sentinel symbol. The reason for this is that it is assumed that all lists are terminated by the sentinel. This assumption is very important to the SAIS algorithm as it is central to its correctness argument. That being said, in terms of elegance and consistency, using *list-less-ns* requires the least amount of special cases.

42.1 General Suffix Type Simplifications

This section contains theorems that simplify the use of the definition *suffix-type*.

lemma *suffix-type-cons-suc*:
suffix-type (*a* # *s*) (*Suc i*) = *suffix-type s i*
⟨*proof*⟩

lemma *suffix-type-cons-same*:
suffix-type (*x* # *x* # *xs*) *0* = *suffix-type* (*x* # *xs*) *0*
⟨*proof*⟩

lemma *suffix-type-suffix*:
suffix-type s i = *suffix-type* (*suffix s i*) *0*
⟨*proof*⟩

lemma *suffix-type-suffix-gen*:
suffix-type (*suffix s n*) *i* = *suffix-type s* (*i* + *n*)
⟨*proof*⟩

lemma *suffix-type-eq-Suc*:
suffix-type xs n = *suffix-type xs* (*Suc n*) ⟹
suffix-type xs n = *S-type* ∨ *suffix-type xs* (*Suc n*) = *L-type*
⟨*proof*⟩

42.2 S-Type Simplifications

This subsection contains theorems about facts that can be derived S-type suffixes and vice versa.

lemma *suffix-is-bot*:
suffix s i = [*bot*] ⟹ *suffix-type s i* = *S-type*
⟨*proof*⟩

lemma *suffix-is-singleton*:
suffix s i = [*x*] ⟹ *suffix-type s i* = *S-type*
⟨*proof*⟩

lemma *suffix-type-last*:
length xs = *Suc n* ⟹ *suffix-type xs n* = *S-type*
⟨*proof*⟩

lemma *s-type-list-less-ns*:
suffix-type s i = *S-type* ⟷ *list-less-ns* (*suffix s i*) (*suffix s* (*Suc i*))
⟨*proof*⟩

lemma *nth-less-imp-s-type*:
⟦*Suc i* < *length s*; *s* ! *i* < *s* ! *Suc i*⟧ ⟹ *suffix-type s i* = *S-type*
⟨*proof*⟩

lemma *sl-type-hd-less*:
⟦*Suc i* < *length s*; *hd* (*suffix s i*) < *hd* (*suffix s* (*Suc i*))⟧ ⟹
suffix-type s i = *S-type*
⟨*proof*⟩

lemma *suffix-type-cons-less*:
x < *y* ⟹ *suffix-type* (*x* # *y* # *xs*) *0* = *S-type*
⟨*proof*⟩

lemma *suffix-type-s-bound*:
suffix-type s i = *S-type* ⟹ *i* < *length s*
⟨*proof*⟩

lemma *s-type-letter-le-Suc*:
⟦*Suc i* < *length T*; *suffix-type T i* = *S-type*⟧ ⟹
T ! *i* ≤ *T* ! (*Suc i*)
⟨*proof*⟩

lemma *s-type-ex*:
assumes *suffix-type* (*x* # *xs*) *0* = *S-type*
shows (∀ *a* ∈ *set xs*. *a* = *x*) ∨ (∃ *b as bs*. *x* # *xs* = *as* @ *x* # *b* # *bs* ∧ *x* < *b* ∧ (∀ *k* ∈ *set as*. *k* = *x*))
⟨*proof*⟩

42.3 L-Type Simplifications

This subsection contains theorems about facts that can be derived from L-type suffixes and vice versa.

lemma *suffix-is-nil*:
suffix s i = [] ⟹ *suffix-type s i* = *L-type*
⟨*proof*⟩

lemma *l-type-list-less-ns*:
suffix-type s i = *L-type* ⟷ *list-less-ns* (*suffix s* (*Suc i*)) (*suffix s i*) ∨ *suffix s i* = []
⟨*proof*⟩

lemma *nth-gr-imp-l-type*:
$[\![Suc\ i < length\ s;\ s\ !\ i > s\ !\ Suc\ i]\!] \Longrightarrow suffix\text{-}type\ s\ i = L\text{-}type$
⟨*proof*⟩

lemma *sl-type-hd-greater*:
$[\![Suc\ i < length\ s;\ hd\ (suffix\ s\ i) > hd\ (suffix\ s\ (Suc\ i))]\!] \Longrightarrow$
$suffix\text{-}type\ s\ i = L\text{-}type$
⟨*proof*⟩

lemma *suffix-type-cons-greater*:
$x > y \Longrightarrow\ suffix\text{-}type\ (x\ \#\ y\ \#\ xs)\ 0 = L\text{-}type$
⟨*proof*⟩

lemma *l-type-letter-gre-Suc*:
$[\![i < length\ T;\ suffix\text{-}type\ T\ i = L\text{-}type]\!] \Longrightarrow$
$T\ !\ (Suc\ i) \leq T\ !\ i$
⟨*proof*⟩

lemma *l-type-ex*:
assumes $suffix\text{-}type\ (x\ \#\ xs)\ 0 = L\text{-}type$
shows $\exists\, b\ as\ bs.\ x\ \#\ xs = as\ @\ x\ \#\ b\ \#\ bs \wedge x > b \wedge (\forall\, k \in set\ as.\ k = x)$
⟨*proof*⟩

An overlooked property, but one that is crucial for completeness of the SAIS algorithm

lemma *suffix-max-hd-is-l-type*:
assumes $valid\text{-}list\ s$
and $i < length\ s$
and $length\ s > Suc\ 0$
and $hd\ (suffix\ s\ i) = Max\ (set\ s)$
shows $suffix\text{-}type\ s\ i = L\text{-}type$
⟨*proof*⟩

42.4 General Suffix Type Theories

This subsection contains the background theory needed to prove that computing the suffix types of a list can be achieved in linear time by starting from the end of the list (lemma 1, Ko et al., JDA 2005).

The main intuition is that the suffix type of the (i+1)th suffix is known and the ith suffix starts with same symbol of the (i+1)th suffix, then the ith suffix will have the same type.

theorem *sl-type-hd-equal*:
$[\![Suc\ i < length\ s;\ hd\ (suffix\ s\ i) = hd\ (suffix\ s\ (Suc\ i))]\!] \Longrightarrow$
$suffix\text{-}type\ s\ i = suffix\text{-}type\ s\ (Suc\ i)$
⟨*proof*⟩

corollary *sl-type-prefix-equal*:

$\llbracket i + n \leq \mathit{length}\ s;\ \forall j < n.\ \mathit{hd}\ (\mathit{suffix}\ s\ (i + j)) = \mathit{hd}\ (\mathit{suffix}\ s\ i)\rrbracket \Longrightarrow$
$\forall j < n.\ \mathit{suffix\text{-}type}\ s\ (i + j) = \mathit{suffix\text{-}type}\ s\ i$
⟨*proof*⟩

corollary *sl-type-prefix-equal-nth*:
$\llbracket i + n \leq \mathit{length}\ s;\ \forall j < n.\ (\mathit{suffix}\ s\ i)\ !\ j = (\mathit{suffix}\ s\ i)\ !\ 0\rrbracket \Longrightarrow$
$\forall j < n.\ \mathit{suffix\text{-}type}\ s\ (i + j) = \mathit{suffix\text{-}type}\ s\ i$
⟨*proof*⟩

corollary *sl-type-prefix-replicate*:
$\forall i < n.\ \mathit{suffix\text{-}type}\ (\mathit{replicate}\ n\ a\ @\ \mathit{as})\ i = \mathit{suffix\text{-}type}\ (\mathit{replicate}\ n\ a\ @\ \mathit{as})\ 0$
⟨*proof*⟩

lemma *suffix-type-neq*:
$\llbracket \mathit{suffix\text{-}type}\ T\ j \neq \mathit{suffix\text{-}type}\ T\ (\mathit{Suc}\ j);\ \mathit{Suc}\ j < \mathit{length}\ T\rrbracket \Longrightarrow T\ !\ j \neq T\ !\ \mathit{Suc}\ j$
⟨*proof*⟩

42.5 S/L-Type Ordering

This section contains the crucial theorem that L-type suffixes are always less than S-type suffixes if they start with the same symbol (lemma 2, Ko et al., JDA 2005).

theorem *l-less-than-s-type-general*:
assumes $\mathit{suffix\text{-}type}\ (a\ \#\ s1)\ 0 = \mathit{S\text{-}type}$
and $\mathit{suffix\text{-}type}\ (a\ \#\ s2)\ 0 = \mathit{L\text{-}type}$
shows $\mathit{list\text{-}less\text{-}ns}\ (a\ \#\ s2)\ (a\ \#\ s1)$
⟨*proof*⟩

corollary *l-less-than-s-type-suffix*:
assumes $i < \mathit{length}\ s$
and $j < \mathit{length}\ s$
and $s\ !\ i = s\ !\ j$
and $\mathit{suffix\text{-}type}\ s\ i = \mathit{S\text{-}type}$
and $\mathit{suffix\text{-}type}\ s\ j = \mathit{L\text{-}type}$
shows $\mathit{list\text{-}less\text{-}ns}\ (\mathit{suffix}\ s\ j)\ (\mathit{suffix}\ s\ i)$
⟨*proof*⟩

theorem *l-less-than-s-type*:
assumes *valid-list s*
and $i < \mathit{length}\ s$
and $j < \mathit{length}\ s$
and $\mathit{hd}\ (\mathit{suffix}\ s\ i) = \mathit{hd}\ (\mathit{suffix}\ s\ j)$
and $\mathit{suffix\text{-}type}\ s\ i = \mathit{S\text{-}type}$
and $\mathit{suffix\text{-}type}\ s\ j = \mathit{L\text{-}type}$
shows $\mathit{list\text{-}less\text{-}ns}\ (\mathit{suffix}\ s\ j)\ (\mathit{suffix}\ s\ i)$
⟨*proof*⟩

corollary (**in** *Suffix-Array-General*) *same-hd-s-after-l*:
assumes *valid-list*: *valid-list s*

and *i-less-len-s*: $i < length\ s$
and *j-less-len-s*: $j < length\ s$
and *i-neq-j*: $i \neq j$
and *suf-i-type*: *suffix-type s ((sa s)! i) = L-type*
and *suf-j-type*: *suffix-type s ((sa s)! j) = S-type*
and *hd-eq*: *hd (suffix s ((sa s) ! i)) = hd (suffix s ((sa s) ! j))*
shows $i < j$
⟨*proof*⟩

42.6 Implementation of Suffix Type Computation

This subsection contain a shallow embedding of a function that would compute the suffix types for a list.

fun *abs-get-suffix-types* :: (*'a* :: {*linorder*, *order-bot*}) *list* ⇒ *SL-types list*
where
abs-get-suffix-types [] = [] |
abs-get-suffix-types ([-]) = [*S-type*] |
abs-get-suffix-types (*a* # *b* # *xs*) =
(*let ys* = *abs-get-suffix-types* (*b* # *xs*)
in
(*if a* < *b then S-type* # *ys*
else if a > *b then L-type* # *ys*
else hd (*ys*) # *ys*))

lemma *length-abs-get-suffix-types*:
length (*abs-get-suffix-types s*) = *length s*
⟨*proof*⟩

lemma *abs-get-suffix-types-correct-nth*:
i < *length s* ⟹ *abs-get-suffix-types s* ! *i* = *suffix-type s i*
⟨*proof*⟩

lemma *get-suffix-types-correct*:
∀ *i* < *length s*. (*abs-get-suffix-types s*) ! *i* = *suffix-type s i*
⟨*proof*⟩

43 SAIS Sublist Order

This section contains the sublist ordering used in SAIS (definition 2.3, Nong et al., DCC 2009). Note that this generalised so that it is not a ternary relation but a binary relation.

fun *ss-order-less* :: (*'a* :: {*linorder*, *order-bot*}) *list* ⇒ *'a list* ⇒ *bool*
where
ss-order-less [] - = *False* |
ss-order-less - [] = *True* |
ss-order-less (*a* # *as*) (*b* # *bs*) =
(*if a* < *b then True*

else if a > b then False
else if suffix-type (a # as) 0 = suffix-type (b # bs) 0 then ss-order-less as bs
else if suffix-type (a # as) 0 = L-type then True
else False)

As described in section "Suffix Type", the SAIS sublist ordering (*ss-order-less*) is equivalent to *list-less-ns*.

lemma *ss-order-less-equiv-list-less-ns*:
ss-order-less s1 s2 = list-less-ns s1 s2
⟨*proof*⟩

44 Sorting

lemma *sorted-letters-s-types*:
assumes $\forall k \geq i.\ k < j \longrightarrow$ *suffix-type T k = S-type*
and $j \leq$ *length T*
shows *sorted* (*list-slice T i j*)
⟨*proof*⟩

lemma *sorted-letters-l-types*:
assumes $\forall k \geq i.\ k < j \longrightarrow$ *suffix-type T k = L-type*
and $j \leq$ *length T*
shows *sorted* ((*rev* (*list-slice T i j*)))
⟨*proof*⟩

45 LMS-Types

This section contains the definition of an LMS-type; standing for large, middle and small. It also contains lemmas pertaining to these types.

definition
abs-is-lms :: ('*a* :: {*linorder*, *order-bot*}) *list* ⇒ *nat* ⇒ *bool*
where
abs-is-lms s i ≡
(*suffix-type s i = S-type*) ∧
(∃ *j*. *i = Suc j* ∧
suffix-type s j = L-type)

LMS-types are subtypes of *S-type*. This is because these are *S-type*, but they are also immediately succeed *L-type*.

45.1 LMS-Type Simplifications

This subsection contains theorems about facts that can be derived from the *abs-is-lms* definition and vice versa.

lemma *lms-type-list-less-ns*:
abs-is-lms s i = (∃ *j*. *i = Suc j* ∧ *list-less-ns* (*suffix s i*) (*suffix s j*) ∧

list-less-ns (*suffix s i*) (*suffix s* (*Suc i*)))
⟨*proof*⟩

lemma *abs-is-lms-0*:
¬*abs-is-lms s 0*
⟨*proof*⟩

lemma *abs-is-lms-cons-suc*:
$i > 0 \Longrightarrow$ *abs-is-lms* (*a* # *s*) (*Suc i*) = *abs-is-lms s i*
⟨*proof*⟩

lemma *i-s-type-imp-Suc-i-not-lms*:
suffix-type s i = *S-type* $\Longrightarrow$ ¬*abs-is-lms s* (*Suc i*)
⟨*proof*⟩

lemma *suffix-type-same-imp-not-lms*:
suffix-type s i = *suffix-type s* (*Suc i*) $\Longrightarrow$ ¬*abs-is-lms s* (*Suc i*)
⟨*proof*⟩

lemma *abs-is-lms-consec*:
abs-is-lms xs i $\Longrightarrow$ ¬*abs-is-lms xs* (*Suc i*)
abs-is-lms xs (*Suc i*) $\Longrightarrow$ ¬*abs-is-lms xs i*
⟨*proof*⟩

lemma *abs-is-lms-gre-length*:
$n \geq$ *length xs* $\Longrightarrow$ ¬*abs-is-lms xs n*
⟨*proof*⟩

lemma *abs-is-lms-suffix*:
abs-is-lms (*suffix s n*) *i* $\Longrightarrow$ *abs-is-lms s* ($i + n$)
⟨*proof*⟩

lemma *abs-is-lms-i-gr-0*:
$i > 0 \Longrightarrow$ *abs-is-lms* (*suffix s n*) *i* = *abs-is-lms s* ($i + n$)
⟨*proof*⟩

lemma *set-abs-is-lms-suffix*:
{*i*. *abs-is-lms* (*suffix s n*) ($i - n$)} = {*i*. *abs-is-lms s i* ∧ $i > n$}
⟨*proof*⟩

lemma *abs-is-lms-set-less-length*:
$n \geq$ *length xs* $\Longrightarrow$ {*i*. *abs-is-lms xs i* ∧ $i < n$} = {*i*. *abs-is-lms xs i*}
⟨*proof*⟩

lemma *abs-is-lms-suffix-Suc*:
abs-is-lms (*suffix s n*) (*Suc i*) = *abs-is-lms s* (*Suc* ($i + n$))
⟨*proof*⟩

45.2 LMS-Type Sets and Subsets

This subsection contains lemmas about sets and subsets of LMS-types.

lemma *set-lms-gr-0*:
$\{i.\ abs\text{-}is\text{-}lms\ xs\ i \wedge 0 < i\} = \{i.\ abs\text{-}is\text{-}lms\ xs\ i\}$
⟨*proof*⟩

lemma *set-lms-n-subset*:
$\{i.\ abs\text{-}is\text{-}lms\ xs\ i \wedge i > n\} \subseteq \{i.\ abs\text{-}is\text{-}lms\ xs\ i\}$
⟨*proof*⟩

lemma *set-lms-Suc-subset*:
$\{i.\ abs\text{-}is\text{-}lms\ xs\ i \wedge i > Suc\ n\} \subseteq \{i.\ abs\text{-}is\text{-}lms\ xs\ i \wedge i > n\}$
⟨*proof*⟩

lemma *set-lms-Suc-insert*:
$abs\text{-}is\text{-}lms\ xs\ (Suc\ n) \Longrightarrow \{i.\ abs\text{-}is\text{-}lms\ xs\ i \wedge i > n\} = insert\ (Suc\ n)\ \{i.\ abs\text{-}is\text{-}lms\ xs\ i \wedge i > Suc\ n\}$
⟨*proof*⟩

lemma *lms-finite*:
$finite\ \{i.\ abs\text{-}is\text{-}lms\ xs\ i\}$
⟨*proof*⟩

lemma *lms-set-empty*:
$[\![length\ xs = Suc\ n;\ m \geq n]\!] \Longrightarrow \{i.\ abs\text{-}is\text{-}lms\ xs\ i \wedge i > m\ \} = \{\}$
⟨*proof*⟩

45.3 Implementation of LMS-Types Computation

This section contains a shallow embedding of a function that would compute all the LMS-types of an ordered list.

fun *get-lms* :: $('a :: \{linorder,\ order\text{-}bot\})\ list \Rightarrow nat \Rightarrow nat\ list$
where
$get\text{-}lms\ xs\ 0 = []\ |$
$get\text{-}lms\ xs\ (Suc\ n) = (if\ abs\text{-}is\text{-}lms\ xs\ n\ then\ n\ \#\ get\text{-}lms\ xs\ n\ else\ get\text{-}lms\ xs\ n)$

lemma *get-lms-correct*:
$get\text{-}lms\ xs\ n = rev\ (filter\ (abs\text{-}is\text{-}lms\ xs)\ [0..<n])$
⟨*proof*⟩

45.3.1 Properties

This subsection contains miscellaneous lemmas about facts that can be derived from the shallow embedding and vice versa.

lemma *get-lms-element-bound*:
$x \in set\ (get\text{-}lms\ xs\ n) \Longrightarrow x < n \wedge x > 0$
⟨*proof*⟩

lemma *distinct-get-lms*:
distinct (*get-lms xs n*)
⟨*proof*⟩

lemma *get-lms-abs-is-lms*:
$x \in set\ (get\text{-}lms\ xs\ n) \longleftrightarrow abs\text{-}is\text{-}lms\ xs\ x \wedge x < n$
⟨*proof*⟩

lemma *lms-le-length*:
$x \in set\ (get\text{-}lms\ xs\ n) \Longrightarrow x < length\ xs$
⟨*proof*⟩

lemma *get-lms-set*:
$set\ (get\text{-}lms\ xs\ n) = \{i.\ abs\text{-}is\text{-}lms\ xs\ i \wedge i < n\}$
⟨*proof*⟩

lemma *get-lms-set-n-gre-length*:
$n \geq length\ xs \Longrightarrow set\ (get\text{-}lms\ xs\ n) = \{i.\ abs\text{-}is\text{-}lms\ xs\ i\}$
⟨*proof*⟩

45.4 Cardinality LMS-Types

This section contains lemmas about how many LMS-types exist (lemma 2.1, Nonge et al., DCC2009). These lemmas are particularly important when proving that the SAIS is O(n) in space (bytes) and time complexity (lemma 3.1, Nong et al., DCC 2009).

lemma *num-lms-bound-1*:
$length\ (get\text{-}lms\ xs\ n) \leq n\ div\ 2$
⟨*proof*⟩

lemma *num-lms-bound-2*:
$length\ (get\text{-}lms\ xs\ n) \leq length\ xs\ div\ 2$
⟨*proof*⟩

lemma *card-abs-is-lms-bound*:
$xs \neq [] \Longrightarrow card\ \{i.\ abs\text{-}is\text{-}lms\ xs\ i\} < length\ xs$
⟨*proof*⟩

lemma *card-abs-is-lms-bound-length-div-2*:
$card\ \{i.\ abs\text{-}is\text{-}lms\ xs\ i\} \leq length\ xs\ div\ 2$
⟨*proof*⟩

lemma *length-filter-lms*:
$T \neq [] \Longrightarrow length\ (filter\ (abs\text{-}is\text{-}lms\ T)\ [0..<length\ T]) < length\ T$
⟨*proof*⟩

45.5 General Properties about LMS-types

lemma *abs-is-lms-imp-le-nth*:
$\llbracket$*abs-is-lms T i*; *Suc i* < *length T*$\rrbracket \Longrightarrow$ *T* ! *i* ≤ *T* ! *Suc i*
⟨*proof*⟩

lemma *abs-is-lms-neq*:
abs-is-lms T (*Suc i*) $\Longrightarrow$ *T* ! *Suc i* < *T* ! *i*
⟨*proof*⟩

lemma *abs-is-lms-last*:
$\llbracket$*valid-list T*; *length T* = *Suc* (*Suc n*)$\rrbracket \Longrightarrow$ *abs-is-lms T* (*Suc n*)
⟨*proof*⟩

lemma *abs-is-lms-imp-less-length*:
abs-is-lms T i $\Longrightarrow$ *i* < *length T*
⟨*proof*⟩

lemma *s-type-and-not-lms-Suc*:
$\llbracket$¬*abs-is-lms T* (*Suc i*); *suffix-type T* (*Suc i*) = *S-type*$\rrbracket \Longrightarrow$ *suffix-type T i* = *S-type*
⟨*proof*⟩

lemma *no-lms-imp-all-s-type*:
assumes *j* < *length T*
and *i* ≤ *j*
and ∀ *k*>*i*. *k* ≤ *j* ⟶ ¬*abs-is-lms T k*
and *suffix-type T j* = *S-type*
and *i* ≤ *k*
and *k* ≤ *j*
shows *suffix-type T k* = *S-type*
⟨*proof*⟩

lemma *first-l-type-after-s-type*:
assumes *j* < *length T*
and *i* ≤ *j*
and ∀ *k*>*i*. *k* ≤ *j* ⟶ ¬*abs-is-lms T k*
and *suffix-type T j* = *L-type*
and *suffix-type T i* = *S-type*
shows ∃ *l*≥*i*. *l* ≤ *j* ∧ (∀ *k*<*l*. *i* ≤ *k* ⟶ *suffix-type T k* = *S-type*) ∧ *suffix-type T l* = *L-type*
⟨*proof*⟩

lemma *no-lms-imp-and-s-imp-all-s-below*:
assumes ∀ *k*. *i* ≤ *k* ∧ *k* < *j* ⟶ ¬*abs-is-lms T k*
and *suffix-type T k* = *S-type*
and *i* ≤ *k*
and *k* < *j*
shows $\llbracket$*i* ≤ *k'*; *k'* ≤ *k*$\rrbracket \Longrightarrow$ *suffix-type T k'* = *S-type*
⟨*proof*⟩

lemma *no-lms-imp-and-l-imp-all-l-above*:
  assumes $\forall k.\ i \leq k \wedge k < j \longrightarrow \neg abs\text{-}is\text{-}lms\ T\ k$
  and $suffix\text{-}type\ T\ k = L\text{-}type$
  and $i \leq k$
  and $k < j$
shows $[\![k \leq k';\ k' < j]\!] \Longrightarrow suffix\text{-}type\ T\ k' = L\text{-}type$
⟨*proof*⟩

lemma *lms-sublist-helper*:
  assumes $\forall k.\ suffix\text{-}type\ T\ k = S\text{-}type \longrightarrow Suc\ k < n \longrightarrow i \leq k \longrightarrow suffix\text{-}type\ T\ (Suc\ k) \neq L\text{-}type$
  and $suffix\text{-}type\ T\ i = S\text{-}type$
shows $[\![i \leq k;\ k < n]\!] \Longrightarrow suffix\text{-}type\ T\ k = S\text{-}type$
⟨*proof*⟩

end
theory *Buckets*
  imports
    ../../util/Continuous-Interval
    List-Type
begin

46 Buckets

46.1 Entire Bucket

definition $bucket :: ('a :: \{linorder,order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow nat\ set$
  where
$bucket\ \alpha\ T\ b \equiv \{k\ |k.\ k < length\ T \wedge \alpha\ (T\ !\ k) = b\}$

definition $bucket\text{-}size :: ('a :: \{linorder,order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow nat$
  where
$bucket\text{-}size\ \alpha\ T\ b \equiv card\ (bucket\ \alpha\ T\ b)$

definition $bucket\text{-}upt :: ('a :: \{linorder,order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow nat\ set$
  where
$bucket\text{-}upt\ \alpha\ T\ b = \{k\ |k.\ k < length\ T \wedge \alpha\ (T\ !\ k) < b\}$

definition $bucket\text{-}start :: ('a :: \{linorder,order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow nat$
  where
$bucket\text{-}start\ \alpha\ T\ b \equiv card\ (bucket\text{-}upt\ \alpha\ T\ b)$

definition $bucket\text{-}end :: ('a :: \{linorder,order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow nat$
  where
$bucket\text{-}end\ \alpha\ T\ b \equiv card\ (bucket\text{-}upt\ \alpha\ T\ (Suc\ b))$

lemma *bucket-upt-subset*:
$bucket\text{-}upt\ \alpha\ T\ b \subseteq \{0..<length\ T\}$
⟨*proof*⟩

lemma *bucket-upt-subset-Suc*:
$bucket\text{-}upt\ \alpha\ T\ b \subseteq bucket\text{-}upt\ \alpha\ T\ (Suc\ b)$
⟨*proof*⟩

lemma *bucket-upt-un-bucket*:
$bucket\text{-}upt\ \alpha\ T\ b \cup bucket\ \alpha\ T\ b = bucket\text{-}upt\ \alpha\ T\ (Suc\ b)$
⟨*proof*⟩

lemma *bucket-0*:
assumes $valid\text{-}list\ T\ \alpha\ bot = 0\ strict\text{-}mono\ \alpha\ length\ T = Suc\ k$
shows $bucket\ \alpha\ T\ 0 = \{k\}$
⟨*proof*⟩

lemma *finite-bucket*:
$finite\ (bucket\ \alpha\ T\ x)$
⟨*proof*⟩

lemma *finite-bucket-upt*:
$finite\ (bucket\text{-}upt\ \alpha\ T\ b)$
⟨*proof*⟩

lemma *bucket-start-Suc*:
$bucket\text{-}start\ \alpha\ T\ (Suc\ b) = bucket\text{-}start\ \alpha\ T\ b + bucket\text{-}size\ \alpha\ T\ b$
⟨*proof*⟩

lemma *bucket-start-le*:
$b \leq b' \Longrightarrow bucket\text{-}start\ \alpha\ T\ b \leq bucket\text{-}start\ \alpha\ T\ b'$
⟨*proof*⟩

lemma *bucket-start-Suc-eq-bucket-end*:
$bucket\text{-}start\ \alpha\ T\ (Suc\ b) = bucket\text{-}end\ \alpha\ T\ b$
⟨*proof*⟩

lemma *bucket-end-le-length*:
$bucket\text{-}end\ \alpha\ T\ b \leq length\ T$
⟨*proof*⟩

lemma *bucket-start-le-end*:
$bucket\text{-}start\ \alpha\ T\ b \leq bucket\text{-}end\ \alpha\ T\ b$
⟨*proof*⟩

lemma *le-bucket-start-le-end*:
$b \leq b' \Longrightarrow bucket\text{-}start\ \alpha\ T\ b \leq bucket\text{-}end\ \alpha\ T\ b'$
⟨*proof*⟩

lemma *bucket-end-le*:
$b \leq b' \Longrightarrow$ *bucket-end* α T $b \leq$ *bucket-end* α T b'
⟨*proof*⟩

lemma *less-bucket-end-le-start*:
$b < b' \Longrightarrow$ *bucket-end* α T $b \leq$ *bucket-start* α T b'
⟨*proof*⟩

lemma *bucket-end-def'*:
bucket-end α T b = *bucket-start* α T b + *bucket-size* α T b
⟨*proof*⟩

lemma *valid-list-bucket-start-0*:
⟦*valid-list* T; *strict-mono* α; α *bot* = *0*⟧ $\Longrightarrow$
bucket-start α T *0* = *0*
⟨*proof*⟩

lemma *bucket-upt-0*:
bucket-upt α T *0* = {}
⟨*proof*⟩

lemma *bucket-start-0*:
bucket-start α T *0* = *0*
⟨*proof*⟩

lemma *valid-list-bucket-upt-Suc-0*:
⟦*valid-list* T; *strict-mono* α; α *bot* = *0*; *length* T = *Suc* n⟧ $\Longrightarrow$
bucket-upt α T (*Suc* *0*) = {n}
⟨*proof*⟩

lemma *valid-list-bucket-end-0*:
⟦*valid-list* T; *strict-mono* α; α *bot* = *0*⟧ $\Longrightarrow$
bucket-end α T *0* = *1*
⟨*proof*⟩

lemma *nth-Max*:
$T \neq [] \Longrightarrow \exists\, i <$ *length* T. T ! i = *Max* (*set* T)
⟨*proof*⟩

lemma *bucket-upt-Suc-Max*:
strict-mono $\alpha \Longrightarrow$ *bucket-upt* α T (*Suc* (α (*Max* (*set* T)))) = {*0*..<*length* T}
⟨*proof*⟩

lemma *bucket-end-Max*:
strict-mono $\alpha \Longrightarrow$ *bucket-end* α T (α (*Max* (*set* T))) = *length* T
⟨*proof*⟩

lemma *bucket-end-eq-length*:
⟦*strict-mono* α; $b \leq \alpha$ (*Max* (*set* T)); $T \neq []$; *bucket-end* α T b = *length* T⟧ $\Longrightarrow$

$b = \alpha$ (*Max* (*set* *T*))
⟨*proof*⟩

46.2 L-types

definition *l-bucket* :: ($'a$ :: {*linorder*,*order-bot*} ⇒ *nat*) ⇒ $'a$ *list* ⇒ *nat* ⇒ *nat set*
where
l-bucket α *T* *b* = {*k* |*k*. *k* ∈ *bucket* α *T* *b* ∧ *suffix-type* *T* *k* = *L-type*}

definition *l-bucket-size* :: ($'a$ :: {*linorder*,*order-bot*} ⇒ *nat*) ⇒ $'a$ *list* ⇒ *nat* ⇒ *nat*
where
l-bucket-size α *T* *b* ≡ *card* (*l-bucket* α *T* *b*)

definition *l-bucket-end* :: ($'a$ :: {*linorder*,*order-bot*} ⇒ *nat*) ⇒ $'a$ *list* ⇒ *nat* ⇒ *nat*
where
l-bucket-end α *T* *b* = *bucket-start* α *T* *b* + *l-bucket-size* α *T* *b*

lemma *l-bucket-subset-bucket*:
l-bucket α *T* *b* ⊆ *bucket* α *T* *b*
⟨*proof*⟩

lemma *bucket-upt-int-l-bucket*:
strict-mono α ⟹ *bucket-upt* α *T* *b* ∩ *l-bucket* α *T* *b* = {}
⟨*proof*⟩

lemma *subset-l-bucket*:
⟦∀ *k* < *length ls*. *ls* ! *k* < *length T* ∧ *suffix-type* *T* (*ls* ! *k*) = *L-type* ∧ α (*T* ! (*ls* ! *k*)) = *x*;
distinct ls⟧ ⟹
set ls ⊆ *l-bucket* α *T* *x*
⟨*proof*⟩

lemma *finite-l-bucket*:
finite (*l-bucket* α *T* *x*)
⟨*proof*⟩

lemma *l-bucket-list-eq*:
⟦∀ *k* < *length ls*. *ls* ! *k* < *length T* ∧ *suffix-type* *T* (*ls* ! *k*) = *L-type* ∧ α (*T* ! (*ls* ! *k*)) = *x*;
distinct ls; *length ls* = *l-bucket-size* α *T* *x*⟧ ⟹
set ls = *l-bucket* α *T* *x*
⟨*proof*⟩

lemma *l-bucket-le-bucket-size*:
l-bucket-size α *T* *b* ≤ *bucket-size* α *T* *b*
⟨*proof*⟩

lemma *l-bucket-not-empty*:
$[\![i < \mathit{length}\ T;\ \mathit{suffix\text{-}type}\ T\ i = \mathit{L\text{-}type}]\!] \Longrightarrow 0 < \mathit{l\text{-}bucket\text{-}size}\ \alpha\ T\ (\alpha\ (T\ !\ i))$
⟨*proof*⟩

lemma *l-bucket-end-le-bucket-end*:
$\mathit{l\text{-}bucket\text{-}end}\ \alpha\ T\ b \leq \mathit{bucket\text{-}end}\ \alpha\ T\ b$
⟨*proof*⟩

lemma *l-bucket-Max*:
assumes *valid-list T*
and $\mathit{Suc}\ 0 < \mathit{length}\ T$
and *strict-mono* α
shows $\mathit{l\text{-}bucket}\ \alpha\ T\ (\alpha\ (\mathit{Max}\ (\mathit{set}\ T))) = \mathit{bucket}\ \alpha\ T\ (\alpha\ (\mathit{Max}\ (\mathit{set}\ T)))$
⟨*proof*⟩

46.3 LMS-types

definition *lms-bucket* :: $('a :: \{\mathit{linorder},\mathit{order\text{-}bot}\} \Rightarrow \mathit{nat}) \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}\ \mathit{set}$
where
$\mathit{lms\text{-}bucket}\ \alpha\ T\ b = \{k\ |k.\ k \in \mathit{bucket}\ \alpha\ T\ b \wedge \mathit{abs\text{-}is\text{-}lms}\ T\ k\}$

definition *lms-bucket-size* :: $('a :: \{\mathit{linorder},\mathit{order\text{-}bot}\} \Rightarrow \mathit{nat}) \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}$
where
$\mathit{lms\text{-}bucket\text{-}size}\ \alpha\ T\ b \equiv \mathit{card}\ (\mathit{lms\text{-}bucket}\ \alpha\ T\ b)$

lemma *lms-bucket-subset-bucket*:
$\mathit{lms\text{-}bucket}\ \alpha\ T\ b \subseteq \mathit{bucket}\ \alpha\ T\ b$
⟨*proof*⟩

lemma *finite-lms-bucket*:
$\mathit{finite}\ (\mathit{lms\text{-}bucket}\ \alpha\ T\ b)$
⟨*proof*⟩

lemma *disjoint-l-lms-bucket*:
$\mathit{l\text{-}bucket}\ \alpha\ T\ b \cap \mathit{lms\text{-}bucket}\ \alpha\ T\ b = \{\}$
⟨*proof*⟩

46.4 S-types

definition *s-bucket* :: $('a :: \{\mathit{linorder},\mathit{order\text{-}bot}\} \Rightarrow \mathit{nat}) \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}\ \mathit{set}$
where
$\mathit{s\text{-}bucket}\ \alpha\ T\ b = \{k\ |k.\ k \in \mathit{bucket}\ \alpha\ T\ b \wedge \mathit{suffix\text{-}type}\ T\ k = \mathit{S\text{-}type}\}$

definition *s-bucket-size* :: $('a :: \{\mathit{linorder},\mathit{order\text{-}bot}\} \Rightarrow \mathit{nat}) \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}$
where
$\mathit{s\text{-}bucket\text{-}size}\ \alpha\ T\ b \equiv \mathit{card}\ (\mathit{s\text{-}bucket}\ \alpha\ T\ b)$

definition *s-bucket-start* :: ($'a$ :: {*linorder*,*order-bot*} ⇒ *nat*) ⇒ $'a$ *list* ⇒ *nat* ⇒ *nat*
where
s-bucket-start α *T b* ≡ *bucket-start* α *T b* + *l-bucket-size* α *T b*

lemma *finite-s-bucket*:
finite (*s-bucket* α *T b*)
⟨*proof*⟩

lemma *disjoint-l-s-bucket*:
l-bucket α *T b* ∩ *s-bucket* α *T b* = {}
⟨*proof*⟩

lemma *lms-subset-s-bucket*:
lms-bucket α *T b* ⊆ *s-bucket* α *T b*
⟨*proof*⟩

lemma *l-un-s-bucket*:
bucket α *T b* = *l-bucket* α *T b* ∪ *s-bucket* α *T b*
⟨*proof*⟩

lemma *s-bucket-Max*:
assumes *valid-list T*
and *length T* > *Suc 0*
and *strict-mono* α
shows *s-bucket* α *T* (α (*Max* (*set T*))) = {}
⟨*proof*⟩

lemma *s-bucket-0*:
assumes *valid-list T*
and *strict-mono* α
and α *bot* = *0*
and *length T* = *Suc n*
shows *s-bucket* α *T 0* = {*n*}
⟨*proof*⟩

lemma *s-bucket-successor*:
⟦*valid-list T*; *strict-mono* α; α *bot* = *0*; *b* ≠ *0*; *x* ∈ *s-bucket* α *T b*⟧ ⟹
Suc x ∈ *s-bucket* α *T b* ∨ (∃ *b'*. *b* < *b'* ∧ *Suc x* ∈ *bucket* α *T b'*)
⟨*proof*⟩

lemma *subset-s-bucket-successor*:
⟦*valid-list T*; *strict-mono* α; α *bot* = *0*; *b* ≠ *0*; *A* ⊆ *s-bucket* α *T b*; *A* ≠ {}⟧ ⟹
∃ *x* ∈ *A*. *Suc x* ∈ *s-bucket* α *T b* − *A* ∨ (∃ *b'*. *b* < *b'* ∧ *Suc x* ∈ *bucket* α *T b'*)
⟨*proof*⟩

lemma *valid-list-s-bucket-start-0*:
⟦*valid-list T*; *strict-mono* α; α *bot* = *0*⟧ ⟹

s-bucket-start α T 0 = 0
⟨*proof*⟩

definition *pure-s-bucket* :: (*'a* :: {*linorder*,*order-bot*} ⇒ *nat*) ⇒ *'a list* ⇒ *nat* ⇒ *nat set*
where
pure-s-bucket α T b = {*k* |*k*. *k* ∈ *s-bucket α T b* ∧ *k* ∉ *lms-bucket α T b*}

definition *pure-s-bucket-size* :: (*'a* :: {*linorder*,*order-bot*} ⇒ *nat*) ⇒ *'a list* ⇒ *nat* ⇒ *nat*
where
pure-s-bucket-size α T b ≡ *card* (*pure-s-bucket α T b*)

lemma *finite-pure-s-bucket*:
finite (*pure-s-bucket α T b*)
⟨*proof*⟩

lemma *pure-s-subset-s-bucket*:
pure-s-bucket α T b ⊆ *s-bucket α T b*
⟨*proof*⟩

lemma *disjoint-lms-pure-s-bucket*:
lms-bucket α T b ∩ *pure-s-bucket α T b* = {}
⟨*proof*⟩

lemma *disjoint-pure-s-lms-bucket*:
pure-s-bucket α T b ∩ *lms-bucket α T b* = {}
⟨*proof*⟩

lemma *s-eq-pure-s-un-lms-bucket*:
s-bucket α T b = *pure-s-bucket α T b* ∪ *lms-bucket α T b*
⟨*proof*⟩

lemma *l-pl-pure-s-pl-lms-size*:
bucket-size α T b = *l-bucket-size α T b* + *pure-s-bucket-size α T b* + *lms-bucket-size α T b*
⟨*proof*⟩

lemma *s-bucket-start-eq-l-bucket-end*:
s-bucket-start α T b = *l-bucket-end α T b*
⟨*proof*⟩

lemma *s-eq-pure-pl-lms-size*:
s-bucket-size α T b = *pure-s-bucket-size α T b* + *lms-bucket-size α T b*
⟨*proof*⟩

lemma *bucket-end-eq-s-start-pl-size*:
bucket-end α T b = *s-bucket-start α T b* + *s-bucket-size α T b*
⟨*proof*⟩

lemma *bucket-start-le-s-bucket-start*:
  bucket-start α *T b* $\leq$ *s-bucket-start* α *T b*
  ⟨*proof*⟩

lemma *bucket-0-size1*:
  assumes *valid-list T*
  and *strict-mono* α
  and α *bot* = *0*
shows *bucket-size* α *T 0* = *Suc 0* $\wedge$ *l-bucket-size* α *T 0* = *0*
⟨*proof*⟩

lemma *bucket-0-size2*:
  assumes *valid-list T*
  and *strict-mono* α
  and α *bot* = *0*
  and *length T* = *Suc* (*Suc n*)
shows *bucket-size* α *T 0* = *Suc 0* $\wedge$ *l-bucket-size* α *T 0* = *0* $\wedge$ *lms-bucket-size* α *T 0* = *Suc 0* $\wedge$
    pure-s-bucket-size α *T 0* = *0*
⟨*proof*⟩

definition *lms-bucket-start* :: (*'a* :: {*linorder*,*order-bot*} $\Rightarrow$ *nat*) $\Rightarrow$ *'a list* $\Rightarrow$ *nat* $\Rightarrow$ *nat*
  where
lms-bucket-start α *T b* = *bucket-start* α *T b* + *l-bucket-size* α *T b* + *pure-s-bucket-size* α *T b*

lemma *l-bucket-end-le-lms-bucket-start*:
  l-bucket-end α *T b* $\leq$ *lms-bucket-start* α *T b*
  ⟨*proof*⟩

lemma *lms-bucket-start-le-bucket-end*:
  lms-bucket-start α *T b* $\leq$ *bucket-end* α *T b*
  ⟨*proof*⟩

lemma *lms-bucket-pl-size-eq-end*:
  lms-bucket-start α *T b* + *lms-bucket-size* α *T b* = *bucket-end* α *T b*
  ⟨*proof*⟩

47 Continuous Buckets

lemma *continuous-buckets*:
  continuous-list (*map* (λb. (*bucket-start* α *T b*, *bucket-end* α *T b*)) [*i*..<*j*])
  ⟨*proof*⟩

lemma *index-in-bucket-interval-gen*:
  ⟦*i* < *length T*; *strict-mono* α⟧ $\Longrightarrow$
    $\exists$ *b* $\leq$ α (*Max* (*set T*)). *bucket-start* α *T b* $\leq$ *i* $\wedge$ *i* < *bucket-end* α *T b*

⟨*proof*⟩

lemma *index-in-bucket-interval*:
  $[\![i < \mathit{length}\ T;\ \mathit{valid\text{-}list}\ T;\ \alpha\ \mathit{bot} = 0;\ \mathit{strict\text{-}mono}\ \alpha]\!] \Longrightarrow$
    $\exists\, b \leq \alpha\ (\mathit{Max}\ (\mathit{set}\ T)).\ \mathit{bucket\text{-}start}\ \alpha\ T\ b \leq i \wedge i < \mathit{bucket\text{-}end}\ \alpha\ T\ b$
  ⟨*proof*⟩

48 Bucket Initialisation

definition *lms-bucket-init* :: $('a :: \{\mathit{linorder},\mathit{order\text{-}bot}\} \Rightarrow \mathit{nat}) \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{nat}\ \mathit{list} \Rightarrow \mathit{bool}$
  where
$\mathit{lms\text{-}bucket\text{-}init}\ \alpha\ T\ B =$
  $(\alpha\ (\mathit{Max}\ (\mathit{set}\ T)) < \mathit{length}\ B\ \wedge$
   $(\forall\, b \leq \alpha\ (\mathit{Max}\ (\mathit{set}\ T)).\ B\ !\ b = \mathit{bucket\text{-}end}\ \alpha\ T\ b))$

lemma *lms-bucket-init-length*:
  $\mathit{lms\text{-}bucket\text{-}init}\ \alpha\ T\ B \Longrightarrow \alpha\ (\mathit{Max}\ (\mathit{set}\ T)) < \mathit{length}\ B$
  ⟨*proof*⟩

lemma *lms-bucket-initD*:
  $[\![\mathit{lms\text{-}bucket\text{-}init}\ \alpha\ T\ B;\ b \leq \alpha\ (\mathit{Max}\ (\mathit{set}\ T))]\!] \Longrightarrow B\ !\ b = \mathit{bucket\text{-}end}\ \alpha\ T\ b$
  ⟨*proof*⟩

definition *l-bucket-init* :: $('a :: \{\mathit{linorder},\mathit{order\text{-}bot}\} \Rightarrow \mathit{nat}) \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{nat}\ \mathit{list} \Rightarrow \mathit{bool}$
  where
$\mathit{l\text{-}bucket\text{-}init}\ \alpha\ T\ B =$
  $(\alpha\ (\mathit{Max}\ (\mathit{set}\ T)) < \mathit{length}\ B\ \wedge$
   $(\forall\, b \leq \alpha\ (\mathit{Max}\ (\mathit{set}\ T)).\ B\ !\ b = \mathit{bucket\text{-}start}\ \alpha\ T\ b))$

lemma *l-bucket-init-length*:
  $\mathit{l\text{-}bucket\text{-}init}\ \alpha\ T\ B \Longrightarrow \alpha\ (\mathit{Max}\ (\mathit{set}\ T)) < \mathit{length}\ B$
  ⟨*proof*⟩

lemma *l-bucket-initD*:
  $[\![\mathit{l\text{-}bucket\text{-}init}\ \alpha\ T\ B;\ b \leq \alpha\ (\mathit{Max}\ (\mathit{set}\ T))]\!] \Longrightarrow B\ !\ b = \mathit{bucket\text{-}start}\ \alpha\ T\ b$
  ⟨*proof*⟩

definition *s-bucket-init*
  where
$\mathit{s\text{-}bucket\text{-}init}\ \alpha\ T\ B =$
  $(\alpha\ (\mathit{Max}\ (\mathit{set}\ T)) < \mathit{length}\ B\ \wedge$
   $(\forall\, b{\leq}\alpha\ (\mathit{Max}\ (\mathit{set}\ T)).$
    $(b > 0 \longrightarrow B\ !\ b = \mathit{bucket\text{-}end}\ \alpha\ T\ b)\ \wedge$
    $(b = 0 \longrightarrow B\ !\ b = 0)$
   $)$
  $)$

lemma *s-bucket-init-length*:
s-bucket-init α *T B* $\Longrightarrow$ α (*Max* (*set T*)) < *length B*
⟨*proof*⟩

lemma *s-bucket-initD*:
⟦*s-bucket-init* α *T B*; *b* $\leq$ α (*Max* (*set T*)); *b* > *0*⟧ $\Longrightarrow$ *B* ! *b* = *bucket-end* α *T b*
⟦*s-bucket-init* α *T B*; *b* $\leq$ α (*Max* (*set T*)); *b* = *0*⟧ $\Longrightarrow$ *B* ! *b* = *0*
⟨*proof*⟩

49 Bucket Range

definition *in-s-current-bucket*
where
in-s-current-bucket α *T B b i* $\equiv$ (*b* $\leq$ α (*Max* (*set T*)) $\wedge$ *B* ! *b* $\leq$ *i* $\wedge$ *i* < *bucket-end* α *T b*)

lemma *in-s-current-bucketD*:
in-s-current-bucket α *T B b i* $\Longrightarrow$ *b* $\leq$ α (*Max* (*set T*))
in-s-current-bucket α *T B b i* $\Longrightarrow$ *B* ! *b* $\leq$ *i*
in-s-current-bucket α *T B b i* $\Longrightarrow$ *i* < *bucket-end* α *T b*
⟨*proof*⟩

definition *in-s-current-buckets*
where
in-s-current-buckets α *T B i* $\equiv$ ($\exists$ *b*. *in-s-current-bucket* α *T B b i*)

lemma *in-s-current-bucket-list-slice*:
assumes *length SA* = *length T*
and *in-s-current-bucket* α *T B b i*
and *SA* ! *i* = *x*
shows *x* $\in$ *set* (*list-slice SA* (*B* ! *b*) (*bucket-end* α *T b*))
⟨*proof*⟩

definition *in-l-bucket*
where
in-l-bucket α *T b i* $\equiv$ (*b* $\leq$ α (*Max* (*set T*)) $\wedge$ *bucket-start* α *T b* $\leq$ *i* $\wedge$ *i* < *l-bucket-end* α *T b*)

end
theory *LMS-List-Slice-Util*
imports *List-Type*
begin

50 Helpers

lemma *filter-abs-is-lms-upt-0*:
filter (*abs-is-lms xs*) [*0*..<*n*] = *filter* (*abs-is-lms xs*) [*Suc 0*..<*n*]

⟨*proof*⟩

lemma *filter-abs-is-lms-upt-hd*:
⟦*abs-is-lms xs i*; *i* < *n*⟧ ⟹
filter (*abs-is-lms xs*) [*i*..<*n*] = *i* # *filter* (*abs-is-lms xs*) [*Suc i*..<*n*]
⟨*proof*⟩

51 LMS Slice

51.1 Find the next LMS position

fun
abs-find-index' :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *nat* ⇒ *nat*
where
abs-find-index' P xs i =
(*case xs of*
[] ⇒ *i*
| *x#xs'* ⇒
(*if P x*
then i
else abs-find-index' P xs' (*Suc i*)))

definition
abs-find-next-lms :: (*'a* :: {*linorder*, *order-bot*}) *list* ⇒ *nat* ⇒ *nat*
where
abs-find-next-lms T i =
(*case find* (λ*j*. *abs-is-lms T j*) [*Suc i*..<*length T*] *of*
Some j ⇒ *j*
| - ⇒ *length T*)

lemma *abs-find-next-lms-le-length*:
abs-find-next-lms T i ≤ *length T*
⟨*proof*⟩

lemma *abs-find-next-lms-abs-is-lms*:
abs-is-lms T (*Suc i*) ⟹ *abs-find-next-lms T i* = *Suc i*
⟨*proof*⟩

lemma *Suc-not-lms-imp-abs-find-next-eq-Suc*:
¬ *abs-is-lms T* (*Suc i*) ⟹ *abs-find-next-lms T i* = *abs-find-next-lms T* (*Suc i*)
⟨*proof*⟩

lemma *abs-find-next-lms-lower-bound-1*:
i < *length T* ⟹ *i* < *abs-find-next-lms T i*
⟨*proof*⟩

lemma *abs-find-next-lms-lower-bound-2*:
length T ≤ *i* ⟹ *length T* ≤ *abs-find-next-lms T i*
⟨*proof*⟩

lemma *abs-find-next-lms-le-Suc*:
abs-find-next-lms T i ≤ *abs-find-next-lms T* (*Suc i*)
⟨*proof*⟩

lemma *no-lms-between-i-and-next*:
⟦*i* < *k*; *k* < *abs-find-next-lms T i*⟧ ⟹ ¬*abs-is-lms T k*
⟨*proof*⟩

lemma *abs-find-next-lms-less-length-abs-is-lms*:
abs-find-next-lms T i < *length T* ⟹
abs-is-lms T (*abs-find-next-lms T i*)
⟨*proof*⟩

lemma *abs-find-next-lms-strict-upper-imp-lower-bound*:
abs-find-next-lms T i < *length T* ⟹
i < *abs-find-next-lms T i*
⟨*proof*⟩

lemma *abs-find-next-lms-suffix*:
assumes *i* ≤ *length T*
shows *abs-find-next-lms T i* =
i + *abs-find-next-lms* (*suffix T i*) *0*
⟨*proof*⟩

lemma *abs-find-next-lms-cons-Suc*:
assumes *i* ≤ *length xs*
shows *abs-find-next-lms* (*x* # *xs*) (*Suc i*) =
Suc (*abs-find-next-lms xs i*)
⟨*proof*⟩

lemma *abs-find-next-lms-funpow-Suc*:
((*abs-find-next-lms T*)^^(*Suc k*)) *i* =
abs-find-next-lms T (((*abs-find-next-lms T*)^^*k*) *i*)
⟨*proof*⟩

lemma *abs-find-next-lms-funpow-le*:
i < *length T* ⟹
((*abs-find-next-lms T*)^^*k*) *i* ≤
((*abs-find-next-lms T*)^^(*Suc k*)) *i*
⟨*proof*⟩

lemma *no-lms-between-i-and-next-funpow*:
⟦((*abs-find-next-lms T*)^^*k*) *i* <
((*abs-find-next-lms T*)^^(*Suc k*)) *i*;
((*abs-find-next-lms T*)^^*k*) *i* < *j*;
j < ((*abs-find-next-lms T*)^^(*Suc k*)) *i*⟧ ⟹
¬ *abs-is-lms T j*
⟨*proof*⟩

lemma *abs-find-next-lms-eq-Suc*:
  $xs \neq [] \Longrightarrow \exists k.\ abs\text{-}find\text{-}next\text{-}lms\ xs\ i = Suc\ k$
  ⟨*proof*⟩

lemma *filter-no-lms1*:
  $[\![abs\text{-}is\text{-}lms\ xs\ i;\ i < k;\ k \leq abs\text{-}find\text{-}next\text{-}lms\ xs\ i]\!] \Longrightarrow$
  $filter\ (abs\text{-}is\text{-}lms\ xs)\ [Suc\ i..{<}k] = []$
⟨*proof*⟩

lemma *filter-no-lms2*:
  $[\![\neg abs\text{-}is\text{-}lms\ xs\ i;\ i < k;\ k \leq abs\text{-}find\text{-}next\text{-}lms\ xs\ i]\!] \Longrightarrow$
  $filter\ (abs\text{-}is\text{-}lms\ xs)\ [i..{<}k] = []$
⟨*proof*⟩

51.2 LMS Prefix

fun
  closest-lms ::
    $('a :: \{linorder,\ order\text{-}bot\})\ list \Rightarrow nat \Rightarrow nat$
where
  $closest\text{-}lms\ T\ i =$
    $(if\ abs\text{-}is\text{-}lms\ T\ i$
     $then\ i$
     $else\ abs\text{-}find\text{-}next\text{-}lms\ T\ i)$

definition
  lms-prefix ::
    $('a :: \{linorder,\ order\text{-}bot\})\ list \Rightarrow nat \Rightarrow 'a\ list$
where
  $lms\text{-}prefix\ T\ i =$
    $list\text{-}slice\ T\ i\ (Suc\ (closest\text{-}lms\ T\ i))$

lemma *lms-lms-prefix*:
  $abs\text{-}is\text{-}lms\ T\ i \Longrightarrow lms\text{-}prefix\ T\ i = [T\ !\ i]$
  ⟨*proof*⟩

lemma *suffix-to-lms-prefix*:
  $i < length\ T \Longrightarrow$
    $suffix\ T\ i =$
    $lms\text{-}prefix\ T\ i\ @$
      $(list\text{-}slice\ T\ (Suc\ (closest\text{-}lms\ T\ i))\ (length\ T))$
  ⟨*proof*⟩

lemma *abs-find-next-lms-funpow-all-lms*:
  $[\![abs\text{-}is\text{-}lms\ xs\ ((abs\text{-}find\text{-}next\text{-}lms\ xs\ \widehat{\ }\widehat{\ }\ Suc\ k)\ x);$
     $i \leq k]\!] \Longrightarrow$
   $abs\text{-}is\text{-}lms\ xs\ ((abs\text{-}find\text{-}next\text{-}lms\ xs\ \widehat{\ }\widehat{\ }\ Suc\ i)\ x)$

⟨*proof*⟩

51.3 LMS Slice

definition
lms-slice :: (*'a* :: {*linorder*, *order-bot*}) *list* ⇒ *nat* ⇒ *'a list*
where
lms-slice T i =
list-slice T i (*Suc* (*abs-find-next-lms T i*))

lemma *suffix-to-lms-slice*:
i < *length T* ⟹
suffix T i =
lms-slice T i @
(*list-slice T* (*Suc* (*abs-find-next-lms T i*)) (*length T*))
⟨*proof*⟩

lemma *suffix-to-lms-slice-app-suffix*:
i < *length T* ⟹
suffix T i =
lms-slice T i @
(*suffix T* (*Suc* (*abs-find-next-lms T i*)))
⟨*proof*⟩

lemma *lms-slice-cons*:
⟦*i* < *length T*; *suffix-type T i* = *S-type*⟧ ⟹
lms-slice T i =
T ! *i* # *lms-slice T* (*Suc i*)
⟨*proof*⟩

lemma *lms-slice-hd*:
i < *length T* ⟹
∃ *xs*. *lms-slice T i* = *T* ! *i* # *xs*
⟨*proof*⟩

lemma *lms-slice-suffix*:
assumes *i* ≤ *length T*
shows *lms-slice* (*suffix T i*) *0* =
lms-slice T i
⟨*proof*⟩

lemma *lms-slice-suffix-gen*:
assumes *i* ≤ *length T*
and *j* ≤ *length T* − *i*
shows *lms-slice* (*suffix T i*) *j* =
lms-slice T (*i* + *j*)
⟨*proof*⟩

lemma *lms-slice-cons-Suc*:

$i \leq \mathit{length}\ xs \Longrightarrow \mathit{lms\text{-}slice}\ (x\ \#\ xs)\ (\mathit{Suc}\ i) = \mathit{lms\text{-}slice}\ xs\ i$
⟨proof⟩

51.4 LMS Substring butlast

definition *lms-slice-butlast* :: ($'a$ :: {*linorder*, *order-bot*}) *list* ⇒ *nat* ⇒ $'a$ *list*
where
lms-slice-butlast T i = *list-slice T i* (*abs-find-next-lms T i*)

lemma *lms-slice-to-butlast-app*:
abs-find-next-lms T i < *length T* ⟹
lms-slice T i = *lms-slice-butlast T i* @ [*T* ! *abs-find-next-lms T i*]
⟨proof⟩

lemma *lms-slice-eq-butlast*:
length T ≤ *abs-find-next-lms T i* ⟹
lms-slice T i = *lms-slice-butlast T i*
⟨proof⟩

lemma *lms-slice-eq-suffix*:
length T ≤ *abs-find-next-lms T i* ⟹
lms-slice T i = *suffix T i*
⟨proof⟩

lemma *suffix-abs-find-next-lms*:
abs-find-next-lms T i < *length T* ⟹
suffix T i = *lms-slice-butlast T i* @ *suffix T* (*abs-find-next-lms T i*)
⟨proof⟩

51.5 Suffix Types

lemma *suffix-type-lms-slice-l-s*:
assumes *suffix-type T i* = *L-type*
and *suffix-type T* (*Suc i*) = *S-type*
shows *suffix-type* (*lms-slice T i*) *0* = *suffix-type T i*
⟨proof⟩

lemma *abs-find-next-lms-same-types*:
assumes ∀ *k*. *i* ≤ *k* ∧ *k* < *length T* ⟶ *suffix-type T k* = *suffix-type T i*
and *i* ≤ *j*
shows *abs-find-next-lms T j* = *length T*
⟨proof⟩

lemma *lms-slice-same-types*:
assumes ∀ *k*. *i* ≤ *k* ∧ *k* < *length T* ⟶ *suffix-type T k* = *suffix-type T i*
and *i* ≤ *j*
shows *lms-slice T j* = *suffix T j*
⟨proof⟩

lemma *all-l-types-up-to-next-lms*:

⟦$i \leq k$; $k <$ *abs-find-next-lms T i*; *suffix-type T i = L-type*⟧ $\Longrightarrow$ *suffix-type T k = L-type*
⟨*proof*⟩

lemma *abs-find-next-lms-eq-length*:
assumes *abs-find-next-lms T i = length T*
and $i <$ *length T*
shows *suffix-type T i = S-type*
⟨*proof*⟩

lemma *abs-find-next-lms-eq-length-all-s-types*:
assumes *abs-find-next-lms T i = length T*
and $i \leq j$
and $j <$ *length T*
shows *suffix-type T j = S-type*
⟨*proof*⟩

lemma *abs-find-next-lms-first-l-after-s-type*:
assumes *abs-find-next-lms T i* $<$ *length T*
and *suffix-type T i = S-type*
shows $\exists j>i.$ $j <$ *abs-find-next-lms T i* $\wedge$ ($\forall k<j.$ $i \leq k \longrightarrow$ *suffix-type T k = S-type*) $\wedge$
suffix-type T j = L-type
⟨*proof*⟩

lemma *lms-slice-type*:
assumes $i <$ *length T*
shows *suffix-type* (*lms-slice T i*) *0* = *suffix-type T i*
⟨*proof*⟩

lemma *lms-slice-l-less-than-s-type-gen*:
assumes *suffix-type* (*a # as*) *0* = *L-type*
and *suffix-type* (*a # bs*) *0* = *S-type*
shows *list-less-ns* (*lms-slice* (*a # as*) *0*) (*lms-slice* (*a # bs*) *0*)
⟨*proof*⟩

lemma *lms-slice-l-less-than-s-type*:
assumes $i <$ *length T*
and $j <$ *length T*
and *T ! i = T ! j*
and *suffix-type T i = L-type*
and *suffix-type T j = S-type*
shows *list-less-ns* (*lms-slice T i*) (*lms-slice T j*)
⟨*proof*⟩

lemma *lms-prefix-type*:
assumes $i <$ *length T*
shows *suffix-type* (*lms-prefix T i*) *0* = *suffix-type T i*
⟨*proof*⟩

lemma *lms-prefix-l-less-than-s-type-gen*:
assumes *suffix-type* (*a* # *as*) *0* = *L-type*
and *suffix-type* (*a* # *bs*) *0* = *S-type*
shows *list-less-ns* (*lms-prefix* (*a* # *as*) *0*) (*lms-prefix* (*a* # *bs*) *0*)
⟨*proof*⟩

lemma *lms-prefix-l-less-than-s-type*:
assumes *i* < *length T*
and *j* < *length T*
and *T* ! *i* = *T* ! *j*
and *suffix-type T i* = *L-type*
and *suffix-type T j* = *S-type*
shows *list-less-ns* (*lms-prefix T i*) (*lms-prefix T j*)
⟨*proof*⟩

lemma *l-type-lms-prefix-cons*:
assumes *suffix-type T i* = *L-type*
and *i* < *length T*
shows *lms-prefix T i* = *T* ! *i* # *lms-prefix T* (*Suc i*)
⟨*proof*⟩

52 Ordering LMS-substrings

This section contains theorems about how LMS-substrings and suffixes are ordered.

lemma *lms-slice-eq-suffix-less*:
assumes *lms-slice T i* = *lms-slice T j*
shows *list-less-ns* (*suffix T i*) (*suffix T j*) ⟷
list-less-ns (*suffix T* (*abs-find-next-lms T i*)) (*suffix T* (*abs-find-next-lms T j*))
⟨*proof*⟩

lemma *lms-slice-eq-suffix-less-funpow′*:
assumes ∀ *k* < *n*. *lms-slice T* (((*abs-find-next-lms T*)^^*k*) *i*) =
lms-slice T (((*abs-find-next-lms T*)^^*k*) *j*)
and *k* < *n*
shows *list-less-ns* (*suffix T i*) (*suffix T j*) ⟷
list-less-ns (*suffix T* (((*abs-find-next-lms T*)^^*k*) *i*)) (*suffix T* (((*abs-find-next-lms T*)^^*k*) *j*))
⟨*proof*⟩

lemma *lms-slice-eq-suffix-less-funpow*:
assumes ∀ *k* < *n*. *lms-slice T* (((*abs-find-next-lms T*)^^*k*) *i*) =
lms-slice T (((*abs-find-next-lms T*)^^*k*) *j*)
shows *list-less-ns* (*suffix T i*) (*suffix T j*) ⟷
list-less-ns (*suffix T* (((*abs-find-next-lms T*)^^*n*) *i*)) (*suffix T* (((*abs-find-next-lms T*)^^*n*) *j*))

⟨*proof*⟩

lemma *list-slice-single*:
  $i < length\ xs \Longrightarrow list\text{-}slice\ xs\ i\ (Suc\ i) = [xs\ !\ i]$
  ⟨*proof*⟩

lemma *less-lms-slice-imp-suffix*:
  assumes $i < length\ T$
  and $j < length\ T$
  and $list\text{-}less\text{-}ns\ (lms\text{-}slice\ T\ i)\ (lms\text{-}slice\ T\ j)$
  shows $list\text{-}less\text{-}ns\ (suffix\ T\ i)\ (suffix\ T\ j)$
⟨*proof*⟩

lemma *lms-slice-list-less-ns-suffix*:
  assumes $abs\text{-}is\text{-}lms\ T\ i$
  and $abs\text{-}is\text{-}lms\ T\ j$
  and $list\text{-}less\text{-}ns\ (lms\text{-}slice\ T\ i)\ (lms\text{-}slice\ T\ j)$
shows $list\text{-}less\text{-}ns\ (suffix\ T\ i)\ (suffix\ T\ j)$
  ⟨*proof*⟩

lemma *less-suffix-imp-lms-slice*:
  assumes $i < length\ T$
  and $j < length\ T$
  and $lms\text{-}slice\ T\ i \neq lms\text{-}slice\ T\ j$
  and $list\text{-}less\text{-}ns\ (suffix\ T\ i)\ (suffix\ T\ j)$
shows $list\text{-}less\text{-}ns\ (lms\text{-}slice\ T\ i)\ (lms\text{-}slice\ T\ j)$
  ⟨*proof*⟩

lemma *not-lms-imp-next-eq-lms-prefix*:
  $\neg abs\text{-}is\text{-}lms\ T\ i \Longrightarrow lms\text{-}slice\ T\ i = lms\text{-}prefix\ T\ i$
  ⟨*proof*⟩

lemma *lms-slice-last*:
  assumes $valid\text{-}list\ T$
  and $length\ T = Suc\ n$
shows $lms\text{-}slice\ T\ n = [bot]$
  ⟨*proof*⟩

lemma *Min-valid-lms-slice*:
  assumes $valid\text{-}list\ T$
  and $length\ T = Suc\ n$
shows $ordlistns.Min\ \{lms\text{-}slice\ T\ i\ |i.\ i < length\ T\} = lms\text{-}slice\ T\ n$
⟨*proof*⟩

lemma *unique-valid-lms-slice*:
  assumes $valid\text{-}list\ T$
  and $length\ T = Suc\ n$
shows $\forall\, i < n.\ lms\text{-}slice\ T\ i \neq lms\text{-}slice\ T\ n$
⟨*proof*⟩

lemma *strict-Min-valid-lms-slice*:
assumes *valid-list T*
and *length T = Suc n*
shows ∀ *i* < *n*. *list-less-ns* (*lms-slice T n*) (*lms-slice T i*)
⟨*proof*⟩

lemma *ordlistns-lms-slice-imp-suffix-strict-sorted*:
assumes *set xs* ⊆ {*i*. *abs-is-lms T i*} *ordlistns.strict-sorted* (*map* (*lms-slice T*) *xs*)
shows *ordlistns.strict-sorted* (*map* (*suffix T*) *xs*)
⟨*proof*⟩

53 Mapping from suffix to lists of LMS-Substrings

This section contains the mapping from LMS-type suffixes to suffixes of the reduced sequence. The mapping is constructed in 3 major steps. 1) From suffix ID to a sequence of LMS-type suffix IDs 2) From a sequence of LMS-type suffix IDs to a sequence of LMS-substrings 3) From a LMS-type suffix to a reduced suffix using the mappings 1, 2 and *ordlistns.elem-rank* The mapping is also shown to be monotonic.

abbreviation *lms-substrs xs* ≡ *lms-slice xs* ‘ {*i*. *abs-is-lms xs i*}
abbreviation *lms-suffixes xs* ≡ *suffix xs* ‘ {*i*. *abs-is-lms xs i*}

abbreviation *nth-lms xs i* ≡ (*abs-find-next-lms xs* ^^ *Suc i*) *0*

abbreviation *lms0 xs* ≡ *abs-find-next-lms xs 0*
abbreviation *lms0-suffix xs* ≡ *suffix xs* (*lms0 xs*)
abbreviation *lms0-substr xs* ≡ *lms-slice xs* (*lms0 xs*)

53.1 LMS Sequence

definition *lms-seq* :: *'a* :: {*linorder*,*order-bot*} *list* ⇒ *nat* ⇒ *nat list*
where
lms-seq xs i = *filter* (*abs-is-lms xs*) [*i*..<*length xs*]

lemma *lms-seq-distinct*:
distinct (*lms-seq xs i*)
⟨*proof*⟩

lemma *lms-seq-sorted*:
sorted (*lms-seq xs i*)
⟨*proof*⟩

lemma *lms-seq-strict-sorted*:
strict-sorted (*lms-seq xs i*)
⟨*proof*⟩

lemma *lms-seq-abs-is-lms-hd*:
  abs-is-lms xs i $\Longrightarrow$ $\exists$ *ys*. *lms-seq xs i* = *i* # *ys*
  ⟨*proof*⟩

lemma *length-lms-seq*:
  assumes *abs-is-lms xs i*
  shows *length* (*lms-seq xs i*) = *card* {*j*. *abs-is-lms xs j* $\wedge$ *i* $\leq$ *j*}
⟨*proof*⟩

lemma *length-lms-seq-less*:
  assumes *abs-is-lms xs i*
  and *abs-is-lms xs j*
  and *i* < *j*
shows *length* (*lms-seq xs j*) < *length* (*lms-seq xs i*)
⟨*proof*⟩

lemma *lms-seq-nth-0*:
  lms-seq xs (*Suc k*) $\neq$ [] $\Longrightarrow$ *lms-seq xs* (*Suc k*) ! *0* = *abs-find-next-lms xs k*
  ⟨*proof*⟩

lemma *lms-seq-eq-cons-lms*:
  assumes *abs-is-lms xs i i* < *k k* $\leq$ *abs-find-next-lms xs i*
  shows *lms-seq xs i* = *i* # *lms-seq xs k*
⟨*proof*⟩

lemma *lms-seq-not-lms*:
  assumes ¬*abs-is-lms xs i i* < *k k* $\leq$ *abs-find-next-lms xs i*
  shows *lms-seq xs i* = *lms-seq xs k*
⟨*proof*⟩

lemma *lms-seq-eq-cons*:
  assumes *lms-seq xs* (*Suc i*) $\neq$ []
  shows *lms-seq xs* (*Suc i*) = *abs-find-next-lms xs i* # *lms-seq xs* (*Suc* (*abs-find-next-lms xs i*))
⟨*proof*⟩

lemma *lms-seq-nth-abs-is-lms*:
  i < *length* (*lms-seq xs k*) $\Longrightarrow$ *abs-is-lms xs* ((*lms-seq xs k*) ! *i*)
  ⟨*proof*⟩

lemma *lms-seq-0*:
  lms-seq xs 0 = *lms-seq xs* (*Suc 0*)
  ⟨*proof*⟩

lemma *lms-seq-nth*:
  i < *length* (*lms-seq xs* (*Suc k*)) $\Longrightarrow$ *lms-seq xs* (*Suc k*) ! *i* = ((*abs-find-next-lms xs*)$\widehat{\ }\widehat{\ }$(*Suc i*)) *k*
⟨*proof*⟩

lemma *inj-on-lms-seq*:
  inj-on (*lms-seq xs*) {*i. abs-is-lms xs i*}
  ⟨*proof*⟩

lemma *list-app-imp-suffix*:
  xs = *ys* @ *zs* ⟹ *suffix xs* (*length ys*) = *zs*
  ⟨*proof*⟩

abbreviation *nth-lms-seq xs i* ≡ *lms-seq xs* (*nth-lms xs i*)

abbreviation *lms0-seq xs* ≡ *lms-seq xs* (*lms0 xs*)

lemma *lms-seq-0-zeroth-lms*:
  lms-seq xs 0 = *lms0-seq xs*
  ⟨*proof*⟩

lemma *lms-seq-set*:
  set (*lms-seq xs i*) = {*k. abs-is-lms xs k* ∧ *i* ≤ *k*}
  ⟨*proof*⟩

lemma *lms-seq-last-eq-length*:
  length (*lms-seq xs i*) = *Suc n* ⟹
    abs-find-next-lms xs ((*lms-seq xs i*) ! *n*) = *length xs*
⟨*proof*⟩

lemma *lms0-seq-has-all-lms*:
  set (*lms0-seq xs*) = {*i. abs-is-lms xs i*}
  ⟨*proof*⟩

lemma *lms0-seq-length*:
  length (*lms0-seq xs*) = *card* {*i. abs-is-lms xs i*}
  ⟨*proof*⟩

lemma *lms0-seq-nth*:
  i < *card* {*i. abs-is-lms xs i*} ⟹ *lms0-seq xs* ! *i* = *nth-lms xs i*
  ⟨*proof*⟩

lemma *lms-seq-Suc1*:
  assumes *abs-is-lms xs i*
  shows *lms-seq xs i* = *i* # *lms-seq xs* (*Suc i*)
  ⟨*proof*⟩

lemma *lms-seq-Suc2*:
  assumes ¬*abs-is-lms xs i*
  shows *lms-seq xs i* = *lms-seq xs* (*Suc i*)
  ⟨*proof*⟩

lemma *lms-seq-suf*:
  i ≤ *j* ⟹ ∃ *ys. lms-seq xs i* = *ys* @ *lms-seq xs j*

⟨*proof*⟩

lemma *lms-lms-seq-is-suffix*:
assumes *abs-is-lms xs i*
shows $\exists k <$ *length* (*lms0-seq xs*).
suffix (*lms0-seq xs*) $k =$ *lms-seq xs i*
⟨*proof*⟩

lemma *nth-lms*:
$i <$ *card* {*i. abs-is-lms xs i*} $\Longrightarrow$
abs-is-lms xs (*nth-lms xs i*)
⟨*proof*⟩

lemma *card-abs-find-next-lms-funpow*:
$i <$ *card* {*k. abs-is-lms xs k*} $\Longrightarrow$
card {*k. abs-is-lms xs k* $\wedge$ $k <$ *nth-lms xs i*} $= i$
⟨*proof*⟩

lemma *lms-seq-nth-suffix*:
$i <$ *card* {*i. abs-is-lms xs i*} $\Longrightarrow$
suffix (*lms0-seq xs*) $i =$ *nth-lms-seq xs i*
⟨*proof*⟩

53.2 LMS-Substring Sequence

definition *lms-substr-seq* :: *'a* :: {*linorder*,*order-bot*} *list* $\Rightarrow$ *nat* $\Rightarrow$ *'a list list*
where
lms-substr-seq xs i = *map* (*lms-slice xs*) (*lms-seq xs i*)

lemma *lms-substr-seq-length*:
length (*lms-substr-seq xs i*) = *length* (*lms-seq xs i*)
⟨*proof*⟩

lemma *inj-on-map-lms-slice-lms-seq*:
inj-on (*map* (*lms-slice xs*)) (*lms-seq xs* ‘ {*i. abs-is-lms xs i*})
⟨*proof*⟩

lemma *inj-on-lms-substr-seq*:
inj-on (*lms-substr-seq xs*) {*i. abs-is-lms xs i*}
⟨*proof*⟩

lemma *lms-substr-seq-nth*:
$i <$ *length* (*lms-substr-seq xs* (*Suc k*)) $\Longrightarrow$
lms-substr-seq xs (*Suc k*) ! $i =$ *lms-slice xs* ((*abs-find-next-lms xs* ^^ *Suc i*) *k*)
⟨*proof*⟩

lemma *lms-substr-seq-nth-abs-is-lms*:
$i <$ *length* (*lms-substr-seq xs k*) $\Longrightarrow$
(*lms-substr-seq xs k*) ! $i \in$ *lms-substrs xs*

⟨proof⟩

definition *suffix-to-id*
where
suffix-to-id xs ys = *length xs* − *length ys*

lemma *suffix-lengths-neq*:
⟦*i* < *j*; *j* < *length xs*⟧ ⟹ *length* (*suffix xs i*) > *length* (*suffix xs j*)
⟨proof⟩

lemma *inj-on-suffix-to-id*:
inj-on (*suffix-to-id xs*) (*suffix xs* ‘ {*i*. *abs-is-lms xs i*})
⟨proof⟩

lemma *suffix-id-suffix*:
i < *length xs* ⟹ *suffix-to-id xs* (*suffix xs i*) = *i*
⟨proof⟩

lemma *suffix-to-id-image*:
suffix-to-id xs ‘ *suffix xs* ‘ {*i*. *abs-is-lms xs i*} = {*i*. *abs-is-lms xs i*}
⟨proof⟩

abbreviation *lms-substr-seq-id xs* ≡ (*lms-substr-seq xs*) ∘ (*suffix-to-id xs*)

lemma *lms-subtrs-seq-id-suffix*:
lms-substr-seq-id xs (*suffix xs i*) = *lms-substr-seq xs i*
⟨proof⟩

lemma *lms-substr-seq-id-nth-abs-is-lms*:
i < *length* (*lms-substr-seq-id xs* (*suffix xs k*)) ⟹
(*lms-substr-seq-id xs* (*suffix xs k*)) ! *i* ∈ *lms-substrs xs*
⟨proof⟩

lemma *inj-on-lms-substr-seq-o-suffix-to-id*:
inj-on (*lms-substr-seq-id xs*) (*lms-suffixes xs*)
⟨proof⟩

lemma *list-less-ns-lms-substr-seq-suffix*:
assumes *abs-is-lms xs i*
and *abs-is-lms xs j*
and *nslexordp list-less-ns* (*lms-substr-seq xs i*) (*lms-substr-seq xs j*)
shows *list-less-ns* (*suffix xs i*) (*suffix xs j*)
⟨proof⟩

lemma *monotone-on-lms-substr-seq-id*:
monotone-on (*lms-suffixes xs*) *list-less-ns* (*nslexordp list-less-ns*) (*lms-substr-seq-id xs*)
(**is** *monotone-on ?A ?orda ?ordb ?f*)
⟨proof⟩

lemma *list-less-ns-suffix-lms-substr-seq*:
assumes *abs-is-lms xs i*
and *abs-is-lms xs j*
and *list-less-ns* (*suffix xs i*) (*suffix xs j*)
shows *nslexordp list-less-ns* (*lms-substr-seq xs i*) (*lms-substr-seq xs j*)
⟨*proof*⟩

lemma *lms-substr-seq-suf*:
$i \leq j \Longrightarrow \exists ys.$ *lms-substr-seq xs i* = *ys* @ *lms-substr-seq xs j*
⟨*proof*⟩

lemma *lms-lms-substr-seq-is-suffix*:
assumes *abs-is-lms xs i*
shows ∃ *k* < *length* (*lms-substr-seq xs* (*abs-find-next-lms xs 0*)).
suffix (*lms-substr-seq xs* (*abs-find-next-lms xs 0*)) *k* = *lms-substr-seq xs i*
⟨*proof*⟩

lemma *lms-substr-seq-nth-suffix*:
i < *card* {*i. abs-is-lms xs i*} ⟹
suffix (*lms-substr-seq xs* (*abs-find-next-lms xs 0*)) *i* =
lms-substr-seq xs ((*abs-find-next-lms xs* ^^ *Suc i*) *0*)
⟨*proof*⟩

53.3 LMS Map

lemma *finite-lms-substrs*:
finite (*lms-substrs xs*)
⟨*proof*⟩

definition *lms-map* :: ('*a* :: {*linorder*, *order-bot*}) *list* ⇒ '*a list* ⇒ *nat list*
where
lms-map xs ≡ (*map* (*ordlistns.elem-rank* (*lms-substrs xs*))) ∘ (*lms-substr-seq-id xs*)

lemma *lms-substr-seq-o-suffix-to-id-range*:
(*lms-substr-seq xs* ∘ *suffix-to-id xs*) ‘ *lms-suffixes xs* ⊆ {*ys. set ys* ⊆ *lms-substrs xs*}
⟨*proof*⟩

lemma *lms-map-o-def*:
lms-map xs ys = *map* (*ordlistns.elem-rank* (*lms-substrs xs*)) (*lms-substr-seq-id xs ys*)
⟨*proof*⟩

lemma *inj-on-lms-map*:
inj-on (*lms-map xs*) (*lms-suffixes xs*)
⟨*proof*⟩

lemma *lms-map-length*:

length (*lms-map xs ys*) = *length* (*lms-substr-seq xs* (*suffix-to-id xs ys*))
⟨*proof*⟩

lemma *lms-map-nth-suffix*:
i < *card* {*i*. *abs-is-lms xs i*} ⟹
suffix (*lms-map xs* (*suffix xs* (*abs-find-next-lms xs 0*))) *i* =
lms-map xs (*suffix xs* ((*abs-find-next-lms xs* ^^ *Suc i*) *0*))
⟨*proof*⟩

lemma *lms-lms-map-is-suffix*:
assumes *abs-is-lms xs i*
shows ∃ *k* < *length* (*lms-map xs* (*suffix xs* (*abs-find-next-lms xs 0*))).
suffix (*lms-map xs* (*suffix xs* (*abs-find-next-lms xs 0*))) *k* = *lms-map xs* (*suffix xs i*)

⟨*proof*⟩

lemma *length-reduced-seq*:
length (*lms-map xs* (*suffix xs* (*abs-find-next-lms xs 0*))) = *card* (*lms-suffixes xs*)
⟨*proof*⟩

corollary *lms-lms-map-in-suffixes*:
abs-is-lms xs i ⟹
lms-map xs (*suffix xs i*) ∈
suffix (*lms-map xs* (*suffix xs* (*abs-find-next-lms xs 0*))) ‘ {*0*..<*card* (*lms-suffixes xs*)}
⟨*proof*⟩

lemma *card-lms-suffixes*:
card (*lms-suffixes xs*) = *card* {*i*. *abs-is-lms xs i*}
⟨*proof*⟩

lemma *lms-map-image*:
lms-map xs ‘ *lms-suffixes xs* =
suffix (*lms-map xs* (*suffix xs* (*abs-find-next-lms xs 0*))) ‘ {*0*..<*card* (*lms-suffixes xs*)}
⟨*proof*⟩

lemma *monotone-on-lms-map*:
monotone-on (*lms-suffixes xs*) *list-less-ns list-less-ns* (*lms-map xs*)
⟨*proof*⟩

lemma *list-less-ns-lms-map-suffix*:
assumes *abs-is-lms xs i*
and *abs-is-lms xs j*
and *list-less-ns* (*lms-map xs* (*suffix xs i*)) (*lms-map xs* (*suffix xs j*))
shows *list-less-ns* (*suffix xs i*) (*suffix xs j*)
⟨*proof*⟩

abbreviation
lms0-map xs ≡
lms-map xs (*lms0-suffix xs*)

lemma *sorted-reduced-seq-imp-lms*:
assumes *ordlistns.strict-sorted* (*map* (*suffix* (*lms0-map xs*)) *ys*)
and $\forall y \in$ *set ys*. $y <$ *card* {*i*. *abs-is-lms xs i*}
shows *ordlistns.strict-sorted* (*map* (*suffix xs*) (*map* ((!) (*lms0-seq xs*)) *ys*))
⟨*proof*⟩

lemma *sorted-distinct-lms-substr*:
assumes *ordlistns.sorted* (*map* (*lms-slice xs*) *ys*)
and *distinct* (*map* (*lms-slice xs*) *ys*)
and $\forall y \in$ *set ys*. $y <$ *length xs*
shows *ordlistns.sorted* (*map* (*suffix xs*) *ys*)
⟨*proof*⟩

lemma *distinct-lms0-map*:
assumes *distinct* (*lms0-map xs*)
shows *distinct* (*map* (*lms-slice xs*) (*lms0-seq xs*))
⟨*proof*⟩

lemma *sorted-distinct-lms-substr-perm*:
assumes *ordlistns.sorted* (*map* (*lms-slice xs*) *ys*)
and *distinct* (*lms0-map xs*)
and *ys* <~~> *lms0-seq xs*
shows *ordlistns.sorted* (*map* (*suffix xs*) *ys*)
⟨*proof*⟩

lemma *list-less-ns-suffix-lms-map*:
assumes *abs-is-lms xs i*
and *abs-is-lms xs j*
and *list-less-ns* (*suffix xs i*) (*suffix xs j*)
shows *list-less-ns* (*lms-map xs* (*suffix xs i*)) (*lms-map xs* (*suffix xs j*))
⟨*proof*⟩

lemma *valid-list-lms-map*:
assumes *valid-list* (*a* # *b* # *xs*)
and *abs-is-lms* (*a* # *b* # *xs*) *i*
shows *valid-list* (*lms-map* (*a* # *b* # *xs*) (*suffix* (*a* # *b* # *xs*) *i*))
⟨*proof*⟩

end
theory *Abs-SAIS*
imports *../prop/Buckets*
../prop/LMS-List-Slice-Util
../../util/Repeat
begin

54 Induce Sorting

54.1 Bucket Insert

```
fun abs-bucket-insert ::
  (('a :: {linorder, order-bot}) ⇒ nat) ⇒
   'a list ⇒
   nat list ⇒
   nat list ⇒
   nat list ⇒
   nat list
  where
abs-bucket-insert α T - SA [] = SA |
abs-bucket-insert α T B SA (x # xs) =
  (let b = α (T ! x);
       k = B ! b − Suc 0;
       SA' = SA[k := x];
       B' = B[b := k]
   in abs-bucket-insert α T B' SA' xs)
```

54.2 Induce L-types

```
fun abs-induce-l-step ::
  nat list × nat list × nat ⇒
   (('a :: {linorder, order-bot}) ⇒ nat) × 'a list ⇒
   nat list × nat list × nat
  where
abs-induce-l-step (B, SA, i) (α, T) =
  (if i < length SA ∧ SA ! i < length T
   then
     (case SA ! i of
        Suc j ⇒
          (case suffix-type T j of
             L-type ⇒
               (let k = α (T ! j);
                    l = B ! k
                in (B[k := Suc l], SA[l := j], Suc i))
           | - ⇒ (B, SA, Suc i))
        | - ⇒ (B, SA, Suc i))
   else (B, SA, Suc i))
```

```
definition abs-induce-l-base ::
  (('a :: {linorder, order-bot}) ⇒ nat) ⇒
   'a list ⇒
   nat list ⇒
   nat list ⇒
   nat list × nat list × nat
  where
abs-induce-l-base α T B SA = repeat (length T) abs-induce-l-step (B, SA, 0) (α,
T)
```

definition *abs-induce-l* ::
  ((*'a* :: {*linorder*, *order-bot*}) ⇒ *nat*) ⇒
   'a list ⇒
   nat list ⇒
   nat list ⇒
   nat list
  where
abs-induce-l α *T B SA* =
  (*let* (*B'*, *SA'*, *i*) = *abs-induce-l-base* α *T B SA*
   in SA')

54.3 Induce S-types

fun *abs-induce-s-step* ::
  nat list × *nat list* × *nat* ⇒
   ((*'a* :: {*linorder*, *order-bot*}) ⇒ *nat*) × *'a list* ⇒
   nat list × *nat list* × *nat*
  where
abs-induce-s-step (*B*, *SA*, *i*) (α, *T*) =
  (*case i of*
    Suc n ⇒
      (*if Suc n* < *length SA* ∧ *SA* ! *Suc n* < *length T then*
        (*case SA* ! *Suc n of*
          Suc j ⇒
            (*case suffix-type T j of*
              S-type ⇒
                (*let b* = α (*T* ! *j*);
                      k = *B* ! *b* − *Suc 0*
                 in (*B*[*b* := *k*], *SA*[*k* := *j*], *n*)
)
              | - ⇒ (*B*, *SA*, *n*)
)
          | - ⇒ (*B*, *SA*, *n*)
)
        else
          (*B*, *SA*, *n*)
)
    | -   ⇒ (*B*, *SA*, *0*)
)

definition *abs-induce-s-base* ::
  ((*'a* :: {*linorder*, *order-bot*}) ⇒ *nat*) ⇒
   'a list ⇒
   nat list ⇒
   nat list ⇒
   nat list × *nat list* × *nat*
  where
abs-induce-s-base α *T B SA* = *repeat* (*length T*) *abs-induce-s-step* (*B*, *SA*, *length*

T) (*α*, *T*)

definition *abs-induce-s* ::
  ((*'a* :: {*linorder*, *order-bot*}) ⇒ *nat*) ⇒
   'a list ⇒
   nat list ⇒
   nat list ⇒
   nat list
  where
abs-induce-s α T B SA =
  (*let* (*B'*, *SA'*, *i*) = *abs-induce-s-base α T B SA*
   in SA')

54.4 Induce Sorting

definition *abs-sa-induce* ::
  ((*'a* :: {*linorder*, *order-bot*}) ⇒ *nat*) ⇒
   'a list ⇒
   nat list ⇒
   nat list
  where
abs-sa-induce α T LMS =
  (*let*
      B0 = *map* (*bucket-end α T*) [*0*..<*Suc* (*α* (*Max* (*set T*)))];
      B1 = *map* (*bucket-start α T*) [*0*..<*Suc* (*α* (*Max* (*set T*)))];

      — Initialise SA
      SA = *replicate* (*length T*) (*length T*);

      — Insert the LMS types into the suffix array
      SA = *abs-bucket-insert α T B0 SA* (*rev LMS*);

      — Insert the L types into the suffix array
      SA = *abs-induce-l α T B1 SA*

   — Insert the S types into the suffix array
   in abs-induce-s α T (*B0*[*0* := *0*]) *SA*)

55 Rename Mapping

fun *abs-rename-mapping'* ::
  (*'a* :: {*linorder*, *order-bot*}) *list* ⇒
    nat list ⇒
    nat list ⇒
    nat ⇒
    nat list
  where
abs-rename-mapping' - [] *names* - = *names* |
abs-rename-mapping' - [*x*] *names i* = *names*[*x* := *i*] |

abs-rename-mapping′ T (*a* # *b* # *xs*) *names i* =
  (*if lms-slice T a* = *lms-slice T b*
    then abs-rename-mapping′ T (*b* # *xs*) (*names*[*a* := *i*]) *i*
   else abs-rename-mapping′ T (*b* # *xs*) (*names*[*a* := *i*]) (*Suc i*))

definition *abs-rename-mapping* :: (*′a* :: {*linorder*, *order-bot*}) *list* ⇒ *nat list* ⇒ *nat list*
  where
abs-rename-mapping T LMS = *abs-rename-mapping′ T LMS* (*replicate* (*length T*) (*length T*)) *0*

56 Rename String

fun *rename-string* :: *nat list* ⇒ *nat list* ⇒ *nat list*
  where
rename-string [] - = [] |
rename-string (*x*#*xs*) *names* = (*names* ! *x*) # *rename-string xs names*

57 Order LMS

fun *order-lms* :: *nat list* ⇒ *nat list* ⇒ *nat list*
  where
order-lms LMS [] = [] |
order-lms LMS (*x* # *xs*) = *LMS* ! *x* # *order-lms LMS xs*

58 Extract LMS

abbreviation *abs-extract-lms* :: (*′a* :: {*linorder*, *order-bot*}) *list* ⇒ *nat list* ⇒ *nat list*
  where
abs-extract-lms ≡ *filter* ∘ *abs-is-lms*

59 SAIS Definition

function *abs-sais* ::
  nat list ⇒
   nat list
where
  abs-sais [] = [] |
  abs-sais [*x*] = [*0*] |
  abs-sais (*a* # *b* # *xs*) =
   (*let*
      T = *a* # *b* # *xs*;

      — Extract the LMS types
      LMS0 = *abs-extract-lms T* [*0*..<*length T*];

— Induce the prefix ordering based on LMS
SA = *abs-sa-induce id T LMS0*;

— Extract the LMS types
LMS = *abs-extract-lms T SA*;

— Create a new alphabet
names = *abs-rename-mapping T LMS*;

— Make a reduced string
T′ = *rename-string LMS0 names*;

— Obtain the correct ordering of LMS-types
LMS = (*if distinct T′ then LMS else order-lms LMS0* (*abs-sais T′*))

— Induce the suffix ordering based of LMS
in abs-sa-induce id T LMS)
⟨*proof*⟩

end
theory *Abs-Bucket-Insert-Verification*
imports
../abs−def/Abs-SAIS
../../util/List-Util
../../util/List-Slice

begin

60 Bucket Insert with Ghost State

fun *bucket-insert-abs′* ::
((*′a* :: {*linorder*, *order-bot*}) ⇒ *nat*) ⇒
′a list ⇒
nat list ⇒
nat list ⇒
nat list ⇒
nat list ⇒
nat list × *nat list* × *nat list*
where
bucket-insert-abs′ α *T B SA gs* [] = (*SA*, *B*, *gs*) |
bucket-insert-abs′ α *T B SA gs* (*x* # *xs*) =
(*let b* = α (*T* ! *x*);
k = *B* ! *b* − *Suc 0*;
SA′ = *SA*[*k* := *x*];
B′ = *B*[*b* := *k*];
gs′ = *gs* @ [*x*]
in bucket-insert-abs′ α *T B′ SA′ gs′ xs*)

61 Simple Properties

lemma *abs-bucket-insert-length*:
length (*abs-bucket-insert* α *T B SA xs*) = *length SA*
⟨*proof*⟩

lemma *abs-bucket-insert-equiv*:
abs-bucket-insert α *T B SA xs* = *fst* (*bucket-insert-abs′* α *T B SA gs xs*)
⟨*proof*⟩

62 Invariants

62.1 Defintions and Simple Helper Lemmas

62.1.1 Distinctness

definition *lms-distinct-inv* ::
(*′a* :: {*linorder*, *order-bot*}) *list* ⇒ *nat list* ⇒ *nat list* ⇒ *bool*
where
lms-distinct-inv T SA LMS =
distinct ((*filter* ($\lambda x.$ x < *length T*) *SA*) @ *LMS*)

lemma *lms-inv-distinct-inv-helper*:
assumes *lms-distinct-inv T SA LMS*
shows *distinct* (*filter* ($\lambda x.$ x < *length T*) *SA*) ∧
distinct LMS ∧
set (*filter* ($\lambda x.$ x < *length T*) *SA*) ∩ *set LMS* = {}
⟨*proof*⟩

62.1.2 LMS Bucket Ptr

definition *cur-lms-types* ::
(*′a* :: {*linorder*, *order-bot*} ⇒ *nat*) ⇒ *′a list* ⇒ *nat list* ⇒ *nat* ⇒ *nat set*
where
cur-lms-types α *T SA b* =
{*i*|*i*. *i* ∈ *set SA* ∧
i ∈ *lms-bucket* α *T b* }

lemma *cur-lms-subset-SA*:
cur-lms-types α *T SA b* ⊆ *set SA*
⟨*proof*⟩

lemma *cur-lms-subset-lms-bucket*:
cur-lms-types α *T SA b* ⊆ *lms-bucket* α *T b*
⟨*proof*⟩

definition *num-lms-types* ::
(*′a* :: {*linorder*, *order-bot*} ⇒ *nat*) ⇒ *′a list* ⇒ *nat list* ⇒ *nat* ⇒ *nat*
where
num-lms-types α *T SA b* =

card (*cur-lms-types* α *T SA b*)

lemma *num-lms-types-upper-bound*:
num-lms-types α *T SA b* $\leq$ *lms-bucket-size* α *T b*
⟨*proof*⟩

definition *lms-bucket-ptr-inv* ::
('*a* :: {*linorder*, *order-bot*} $\Rightarrow$ *nat*) $\Rightarrow$
'*a list* $\Rightarrow$ *nat list* $\Rightarrow$ *nat list* $\Rightarrow$ *bool*
where
lms-bucket-ptr-inv α *T B SA* $\equiv$
($\forall$ *b* $\leq$ α (*Max* (*set T*)).
B ! *b* + *num-lms-types* α *T SA b* = *bucket-end* α *T b*)

lemma *lms-bucket-ptr-invD*:
assumes *lms-bucket-ptr-inv* α *T B SA*
and *b* $\leq$ α (*Max* (*set T*))
shows *B* ! *b* + *num-lms-types* α *T SA b* = *bucket-end* α *T b*
⟨*proof*⟩

lemma *lms-bucket-ptr-lower-bound*:
assumes *lms-bucket-ptr-inv* α *T B SA*
and *b* $\leq$ α (*Max* (*set T*))
shows *lms-bucket-start* α *T b* $\leq$ *B* ! *b*
⟨*proof*⟩

lemma *lms-bucket-ptr-upper-bound*:
assumes *lms-bucket-ptr-inv* α *T B SA*
and *b* $\leq$ α (*Max* (*set T*))
shows *B* ! *b* $\leq$ *bucket-end* α *T b*
⟨*proof*⟩

62.1.3 Unknowns

definition *lms-unknowns-inv* ::
('*a* :: {*linorder*, *order-bot*} $\Rightarrow$ *nat*) $\Rightarrow$
'*a list* $\Rightarrow$ *nat list* $\Rightarrow$ *nat list* $\Rightarrow$ *bool*
where
lms-unknowns-inv α *T B SA* $\equiv$
($\forall$ *b* $\leq$ α (*Max* (*set T*)).
($\forall$ *i*. *lms-bucket-start* α *T b* $\leq$ *i* $\wedge$
i < *B* ! *b* $\longrightarrow$ *SA* ! *i* = *length T*))

lemma *lms-unknowns-invD*:
assumes *lms-unknowns-inv* α *T B SA*
and *b* $\leq$ α (*Max* (*set T*))
and *lms-bucket-start* α *T b* $\leq$ *i*
and *i* < *B* ! *b*
shows *SA* ! *i* = *length T*

⟨*proof*⟩

62.1.4 Locations

definition *lms-locations-inv* ::
  $('a :: \{linorder, order\text{-}bot\} \Rightarrow nat) \Rightarrow$
   $'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
  lms-locations-inv α T B $SA \equiv$
    $(\forall b \leq \alpha\ (Max\ (set\ T)).$
      $(\forall i.\ B\ !\ b \leq i\ \wedge$
        $i < bucket\text{-}end\ \alpha\ T\ b \longrightarrow SA\ !\ i \in lms\text{-}bucket\ \alpha\ T\ b))$

lemma *lms-locations-invD*:
  assumes *lms-locations-inv* α T B SA
  and $b \leq \alpha\ (Max\ (set\ T))$
  and $B\ !\ b \leq i$
  and $i < bucket\text{-}end\ \alpha\ T\ b$
  shows $SA\ !\ i \in lms\text{-}bucket\ \alpha\ T\ b$
  ⟨*proof*⟩

62.1.5 Unchanged

definition *lms-unchanged-inv* ::
  $('a :: \{linorder, order\text{-}bot\} \Rightarrow nat) \Rightarrow$
   $'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
  lms-unchanged-inv α T B SA $SA' \equiv$
   $(\forall b \leq \alpha\ (Max\ (set\ T)).$
     $(\forall i.\ bucket\text{-}start\ \alpha\ T\ b \leq i\ \wedge$
       $i < B\ !\ b \longrightarrow SA'\ !\ i = SA\ !\ i))$

lemma *lms-unchanged-invD*:
  assumes *lms-unchanged-inv* α T B SA SA'
  and $b \leq \alpha\ (Max\ (set\ T))$
  and $bucket\text{-}start\ \alpha\ T\ b \leq i$
  and $i < B\ !\ b$
  shows $SA'\ !\ i = SA\ !\ i$
  ⟨*proof*⟩

62.1.6 Inserted

definition *lms-inserted-inv* ::
  $nat\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
  lms-inserted-inv LMS SA $LMSa$ $LMSb \equiv$
    $LMS = LMSa\ @\ LMSb\ \wedge$
    $set\ LMSa \subseteq set\ SA$

lemma *lms-inserted-invD*:

$\bigwedge LMS\ SA\ LMSa\ LMSb.\ lms\text{-}inserted\text{-}inv\ LMS\ SA\ LMSa\ LMSb \Longrightarrow LMS = LMSa\ @\ LMSb$

$\bigwedge LMS\ SA\ LMSa\ LMSb.\ lms\text{-}inserted\text{-}inv\ LMS\ SA\ LMSa\ LMSb \Longrightarrow set\ LMSa \subseteq set\ SA$

⟨*proof*⟩

62.1.7 Sorted

definition $lms\text{-}sorted\text{-}inv :: ('a :: \{linorder,\ order\text{-}bot\})\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$

where

$lms\text{-}sorted\text{-}inv\ T\ LMS\ SA \equiv$

$(\forall j < length\ SA.$

$\forall i < j.$

$SA\ !\ i \in set\ LMS \land SA\ !\ j \in set\ LMS \longrightarrow$

$(T\ !\ (SA\ !\ i) \neq T\ !\ (SA\ !\ j) \longrightarrow T\ !\ (SA\ !\ i) < T\ !\ (SA\ !\ j)) \land$

$(T\ !\ (SA\ !\ i) = T\ !\ (SA\ !\ j) \longrightarrow$

$(\exists j' < length\ LMS.\ \exists i' < j'.\ LMS\ !\ i' = SA\ !\ j \land LMS\ !\ j' = SA\ !\ i))$

$)$

lemma *lms-sorted-invD*:

$[\![lms\text{-}sorted\text{-}inv\ T\ LMS\ SA;\ j < length\ SA;\ i < j;\ SA\ !\ i \in set\ LMS;\ SA\ !\ j \in set\ LMS]\!] \Longrightarrow$

$(T\ !\ (SA\ !\ i) \neq T\ !\ (SA\ !\ j) \longrightarrow T\ !\ (SA\ !\ i) < T\ !\ (SA\ !\ j)) \land$

$(T\ !\ (SA\ !\ i) = T\ !\ (SA\ !\ j) \longrightarrow$

$(\exists j' < length\ LMS.\ \exists i' < j'.\ LMS\ !\ i' = SA\ !\ j \land LMS\ !\ j' = SA\ !\ i))$

⟨*proof*⟩

lemma *lms-sorted-invD1*:

$[\![lms\text{-}sorted\text{-}inv\ T\ LMS\ SA;\ j < length\ SA;\ i < j;$

$SA\ !\ i \in set\ LMS;\ SA\ !\ j \in set\ LMS;$

$T\ !\ (SA\ !\ i) \neq T\ !\ (SA\ !\ j)]\!] \Longrightarrow$

$T\ !\ (SA\ !\ i) < T\ !\ (SA\ !\ j)$

⟨*proof*⟩

lemma *lms-sorted-invD2*:

$[\![lms\text{-}sorted\text{-}inv\ T\ LMS\ SA;\ j < length\ SA;\ i < j;\ SA\ !\ i \in set\ LMS;\ SA\ !\ j \in set\ LMS;$

$T\ !\ (SA\ !\ i) = T\ !\ (SA\ !\ j)]\!] \Longrightarrow$

$\exists j' < length\ LMS.\ \exists i' < j'.\ LMS\ !\ i' = SA\ !\ j \land LMS\ !\ j' = SA\ !\ i$

⟨*proof*⟩

62.2 Combined Invariant

definition $lms\text{-}inv ::$

$('a :: \{linorder,\ order\text{-}bot\} \Rightarrow nat) \Rightarrow$

$'a\ list \Rightarrow$

$nat\ list \Rightarrow$

$nat\ list \Rightarrow$

$nat\ list \Rightarrow$

nat list ⇒
nat list ⇒
nat list ⇒
bool

where

lms-inv α *T B LMS LMSa LMSb SA0 SA* ≡
lms-distinct-inv T SA LMSb ∧
lms-bucket-ptr-inv α *T B SA* ∧
lms-unknowns-inv α *T B SA* ∧
lms-locations-inv α *T B SA* ∧
lms-unchanged-inv α *T B SA0 SA* ∧
lms-inserted-inv LMS SA LMSa LMSb ∧
lms-sorted-inv T LMS SA ∧
strict-mono α ∧
α (*Max* (*set T*)) < *length B* ∧
set LMS ⊆ {*i. abs-is-lms T i*} ∧
length SA0 = *length T* ∧
length SA = *length T* ∧
(∀ *i* < *length T*. *SA0* ! *i* = *length T*)

lemma *lms-invD*:
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ *lms-distinct-inv T SA LMSb*
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ *lms-bucket-ptr-inv* α *T B SA*
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ *lms-unknowns-inv* α *T B SA*
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ *lms-locations-inv* α *T B SA*
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ *lms-unchanged-inv* α *T B SA0 SA*
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ *lms-inserted-inv LMS SA LMSa LMSb*
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ *lms-sorted-inv T LMS SA*
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ *strict-mono* α
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ α (*Max* (*set T*)) < *length B*
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ *set LMS* ⊆ {*i. abs-is-lms T i*}
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ *length SA0* = *length T*
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ *length SA* = *length T*
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ ∀ *i* < *length T*. *SA0* ! *i* = *length T*
⟨*proof*⟩

lemma *lms-inv-lms-helper*:
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ ∀ *x* ∈ *set LMS*. *abs-is-lms T x*
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ ∀ *x* ∈ *set LMSa*. *abs-is-lms T x*
lms-inv α *T B LMS LMSa LMSb SA0 SA* ⟹ ∀ *x* ∈ *set LMSb*. *abs-is-lms T x*
⟨*proof*⟩

62.3 Helpers

lemma *lms-distinct-bucket-ptr-lower-bound*:
assumes *b* = α (*T* ! *x*)
and *lms-distinct-inv T SA* (*x* # *LMS*)

and *lms-bucket-ptr-inv* α *T B SA*
and *strict-mono* α
and $\forall\, i \in$ *set* (*x* # *LMS*). *abs-is-lms T i*
shows *lms-bucket-start* α *T b* < *B* ! *b*
⟨*proof*⟩

lemma *lms-next-insert-at-unknown*:
assumes *b* = α (*T* ! *x*)
and *k* = (*B* ! *b*) − *Suc 0*
and *lms-distinct-inv T SA* (*x* # *LMS*)
and *lms-bucket-ptr-inv* α *T B SA*
and *lms-unknowns-inv* α *T B SA*
and *strict-mono* α
and *length SA* = *length T*
and $\forall\, i \in$ *set* (*x* # *LMS*). *abs-is-lms T i*
shows *k* < *length SA* ∧ *SA* ! *k* = *length T*
⟨*proof*⟩

lemma *lms-distinct-slice*:
assumes *lms-distinct-inv T SA LMS*
and *lms-bucket-ptr-inv* α *T B SA*
and *lms-locations-inv* α *T B SA*
and *length SA* = *length T*
and *b* ≤ α (*Max* (*set T*))
shows *distinct* (*list-slice SA* (*B* ! *b*) (*bucket-end* α *T b*))
⟨*proof*⟩

lemma *lms-slice-subset-lms-bucket*:
assumes *lms-locations-inv* α *T B SA*
and *length SA* = *length T*
and *b* ≤ α (*Max* (*set T*))
shows *set* (*list-slice SA* (*B* ! *b*) (*bucket-end* α *T b*)) ⊆ *lms-bucket* α *T b*
⟨*proof*⟩

lemma *lms-val-location*:
assumes *lms-locations-inv* α *T B SA*
and *lms-unchanged-inv* α *T B SA0 SA*
and *strict-mono* α
and *length SA* = *length T*
and $\forall\, i$ < *length T*. *SA0* ! *i* = *length T*
and *i* < *length SA*
and *SA* ! *i* < *length T*
shows $\exists\, b \leq \alpha$ (*Max* (*set T*)). *B* ! *b* ≤ *i* ∧ *i* < *bucket-end* α *T b*
⟨*proof*⟩

lemma *lms-val-imp-abs-is-lms*:
assumes *lms-locations-inv* α *T B SA*
and *lms-unchanged-inv* α *T B SA0 SA*
and *strict-mono* α

and *length SA* = *length T*
and $\forall\, i < length\ T.\ SA0\ !\ i = length\ T$
and $i < length\ SA$
and $SA\ !\ i < length\ T$
shows *abs-is-lms T* (*SA* ! *i*)
⟨*proof*⟩

lemma *lms-lms-prefix-sorted*:
assumes *lms-bucket-ptr-inv* α *T B SA*
and *lms-locations-inv* α *T B SA*
and *lms-unchanged-inv* α *T B SA0 SA*
and *strict-mono* α
and *length SA* = *length T*
and $\forall\, i < length\ T.\ SA0\ !\ i = length\ T$
and *set LMS* = {*i*. *abs-is-lms T i*}
shows *ordlistns.sorted* (*map* (*lms-prefix T*) (*filter* (λ*x*. *x* < *length T*) *SA*))
⟨*proof*⟩

lemma *lms-suffix-sorted*:
assumes *lms-bucket-ptr-inv* α *T B SA*
and *lms-locations-inv* α *T B SA*
and *lms-unchanged-inv* α *T B SA0 SA*
and *lms-sorted-inv T LMS SA*
and *strict-mono* α
and *length SA* = *length T*
and $\forall\, i < length\ T.\ SA0\ !\ i = length\ T$
and *set LMS* = {*i*. *abs-is-lms T i*}
and *ordlistns.sorted* (*map* (*suffix T*) (*rev LMS*))
shows *ordlistns.sorted* (*map* (*suffix T*) (*filter* (λ*x*. *x* < *length T*) *SA*))
⟨*proof*⟩

lemma *next-index-outside*:
assumes $b = \alpha\ (T\ !\ x)$
and $k = B\ !\ b - Suc\ 0$
and *lms-distinct-inv T SA* (*x* # *LMS*)
and *lms-bucket-ptr-inv* α *T B SA*
and *strict-mono* α
and $\forall\, a \in set\ (x\ \#\ LMS).\ abs\text{-}is\text{-}lms\ T\ a$
and $b' \leq \alpha\ (Max\ (set\ T))$
and $b \neq b'$
shows $k < bucket\text{-}start\ \alpha\ T\ b' \lor bucket\text{-}end\ \alpha\ T\ b' \leq k$
⟨*proof*⟩

62.4 Establishment and Maintenance Steps

62.4.1 Distinctness

lemma *lms-distinct-inv-established*:
assumes *distinct LMS*
and $\forall\, i < length\ SA.\ SA\ !\ i = length\ T$

shows *lms-distinct-inv T SA LMS*
⟨*proof*⟩

lemma *lms-distinct-inv-maintained-step*:
assumes *lms-distinct-inv T SA* (*x* # *LMS*)
shows *lms-distinct-inv T* (*SA*[*k* := *x*]) *LMS*
⟨*proof*⟩

lemma *lms-distinct-inv-maintained*:
assumes *lms-distinct-inv T SA LMS*
shows *lms-distinct-inv T* (*abs-bucket-insert* α *T B SA LMS*) []
⟨*proof*⟩

lemma *abs-bucket-insert-lms-distinct-inv*:
assumes *distinct LMS*
and $\forall$ *i* < *length SA*. *SA* ! *i* = *length T*
shows *lms-distinct-inv T* (*abs-bucket-insert* α *T B SA LMS*) []
⟨*proof*⟩

62.4.2 Bucket Ptr

lemma *lms-bucket-ptr-inv-established*:
assumes *lms-bucket-init* α *T B*
and $\forall$ *i* < *length SA*. *SA* ! *i* = *length T*
shows *lms-bucket-ptr-inv* α *T B SA*
⟨*proof*⟩

lemma *lms-bucket-ptr-inv-maintained-step*:
assumes *b* = α (*T* ! *x*)
and *k* = *B* ! *b* − *Suc 0*
and *lms-distinct-inv T SA* (*x* # *LMS*)
and *lms-bucket-ptr-inv* α *T B SA*
and *lms-unknowns-inv* α *T B SA*
and *strict-mono* α
and α (*Max* (*set T*)) < *length B*
and *length SA* = *length T*
and $\forall$ *a* $\in$ *set* (*x* # *LMS*). *abs-is-lms T a*
shows *lms-bucket-ptr-inv* α *T* (*B*[*b* := *k*]) (*SA*[*k* := *x*])
⟨*proof*⟩

62.4.3 Unknowns

lemma *lms-unknowns-inv-established*:
assumes *lms-bucket-init* α *T B*
and $\forall$ *i* < *length SA*. *SA* ! *i* = *length T*
and *length SA* = *length T*
shows *lms-unknowns-inv* α *T B SA*
⟨*proof*⟩

lemma *lms-unknowns-inv-maintained-step*:

assumes $b = \alpha\ (T\ !\ x)$
and $k = B\ !\ b - Suc\ 0$
and *lms-distinct-inv* $T\ SA\ (x\ \#\ LMS)$
and *lms-bucket-ptr-inv* $\alpha\ T\ B\ SA$
and *lms-unknowns-inv* $\alpha\ T\ B\ SA$
and *strict-mono* α
and $\alpha\ (Max\ (set\ T)) < length\ B$
and $\forall a \in set\ (x\ \#\ LMS).\ abs\text{-}is\text{-}lms\ T\ a$
shows *lms-unknowns-inv* $\alpha\ T\ (B[b := k])\ (SA[k := x])$
⟨*proof*⟩

62.4.4 Locations

lemma *lms-locations-inv-established*:
assumes *lms-bucket-init* $\alpha\ T\ B$
shows *lms-locations-inv* $\alpha\ T\ B\ SA$
⟨*proof*⟩

lemma *lms-locations-inv-maintained-step*:
assumes $b = \alpha\ (T\ !\ x)$
and $k = (B\ !\ b) - Suc\ 0$
and *lms-distinct-inv* $T\ SA\ (x\ \#\ LMS)$
and *lms-bucket-ptr-inv* $\alpha\ T\ B\ SA$
and *lms-locations-inv* $\alpha\ T\ B\ SA$
and *strict-mono* α
and $\alpha\ (Max\ (set\ T)) < length\ B$
and $length\ SA = length\ T$
and $\forall a \in set\ (x\ \#\ LMS).\ abs\text{-}is\text{-}lms\ T\ a$
shows *lms-locations-inv* $\alpha\ T\ (B[b := k])\ (SA[k := x])$
⟨*proof*⟩

62.4.5 Unchanged

lemma *lms-unchanged-inv-established*:
lms-unchanged-inv $\alpha\ T\ B\ SA\ SA$
⟨*proof*⟩

lemma *lms-unchanged-inv-maintained-step*:
assumes $b = \alpha\ (T\ !\ x)$
and $k = (B\ !\ b) - Suc\ 0$
and *lms-distinct-inv* $T\ SA\ (x\ \#\ LMS)$
and *lms-bucket-ptr-inv* $\alpha\ T\ B\ SA$
and *lms-unchanged-inv* $\alpha\ T\ B\ SA0\ SA$
and *strict-mono* α
and $\alpha\ (Max\ (set\ T)) < length\ B$
and $length\ SA = length\ T$
and $\forall a \in set\ (x\ \#\ LMS).\ abs\text{-}is\text{-}lms\ T\ a$
shows *lms-unchanged-inv* $\alpha\ T\ (B[b := k])\ SA0\ (SA[k := x])$
⟨*proof*⟩

62.4.6 Inserted

lemma *lms-inserted-inv-established*:
  shows *lms-inserted-inv LMS SA* [] *LMS*
  ⟨*proof*⟩

lemma *lms-inserted-inv-maintained-step*:
  assumes $b = \alpha\ (T\ !\ x)$
  and $k = (B\ !\ b) - Suc\ 0$
  and *lms-distinct-inv T SA* (*x* # *LMSb*)
  and *lms-bucket-ptr-inv* α *T B SA*
  and *lms-unknowns-inv* α *T B SA*
  and *lms-inserted-inv LMS SA LMSa* (*x* # *LMSb*)
  and *strict-mono* α
  and *length SA* = *length T*
  and $\forall\, a \in$ *set LMS*. *abs-is-lms T a*
shows *lms-inserted-inv LMS* (*SA*[*k* := *x*]) (*LMSa* @ [*x*]) *LMSb*
⟨*proof*⟩

62.4.7 Sorted

lemma *lms-sorted-inv-established*:
  assumes $\forall\, i <$ *length SA*. *SA* ! *i* = *length T*
  and $\forall\, a \in$ *set LMS*. *abs-is-lms T a*
shows *lms-sorted-inv T LMS SA*
  ⟨*proof*⟩

lemma *lms-sorted-inv-maintained-step*:
  assumes $b = \alpha\ (T\ !\ x)$
  and $k = (B\ !\ b) - Suc\ 0$
  and *lms-distinct-inv T SA* (*x* # *LMSb*)
  and *lms-bucket-ptr-inv* α *T B SA*
  and *lms-unknowns-inv* α *T B SA*
  and *lms-locations-inv* α *T B SA*
  and *lms-unchanged-inv* α *T B SA0 SA*
  and *lms-inserted-inv LMS SA LMSa* (*x* # *LMSb*)
  and *lms-sorted-inv T LMS SA*
  and *strict-mono* α
  and *length SA* = *length T*
  and $\forall\, i <$ *length T*. *SA0* ! *i* = *length T*
  and $\forall\, a \in$ *set LMS*. *abs-is-lms T a*
shows *lms-sorted-inv T LMS* (*SA*[*k* := *x*])
  ⟨*proof*⟩

62.5 Combined Establishment and Maintenance

lemma *lms-inv-established*:
  assumes $\forall\, i <$ *length SA*. *SA* ! *i* = *length T*
  and $\forall\, x \in$ *set LMS*. *abs-is-lms T x*
  and *distinct LMS*

and *lms-bucket-init* α *T B*
and *length SA* = *length T*
and *strict-mono* α
shows *lms-inv* α *T B LMS* [] *LMS SA SA*
⟨*proof*⟩

lemma *lms-inv-maintained-step*:
assumes *lms-inv* α *T B LMS LMSa* (*x* # *LMSb*) *SA0 SA*
and $b = \alpha$ (*T* ! *x*)
and k = (*B* ! *b*) − *Suc 0*
shows *lms-inv* α *T* (*B*[*b* := *k*]) *LMS* (*LMSa* @ [*x*]) *LMSb SA0* (*SA*[*k* := *x*])
⟨*proof*⟩

lemma *lms-inv-maintained*:
assumes *bucket-insert-abs'* α *T B SA gs xs* = (*SA'*, *B'*, *gs'*)
and *lms-inv* α *T B LMS gs xs SA0 SA*
shows *lms-inv* α *T B' LMS gs'* [] *SA0 SA'*
⟨*proof*⟩

lemma *lms-inv-holds*:
assumes $\forall i <$ *length SA*. *SA* ! *i* = *length T*
and $\forall x \in$ *set LMS*. *abs-is-lms T x*
and *distinct LMS*
and *lms-bucket-init* α *T B*
and *length SA* = *length T*
and *strict-mono* α
and *bucket-insert-abs'* α *T B SA* [] *LMS* = (*SA'*, *B'*, *gs'*)
shows *lms-inv* α *T B' LMS gs'* [] *SA SA'*
⟨*proof*⟩

63 Exhaustiveness

definition *lms-type-exhaustive* :: (*'a* :: {*linorder*, *order-bot*}) *list* ⇒ *nat list* ⇒ *bool*
where
lms-type-exhaustive T SA = ($\forall i <$ *length T*. *abs-is-lms T i* ⟶ $i \in$ *set SA*)

lemma *lms-type-exhaustiveD*:
⟦*lms-type-exhaustive T SA*; $i <$ *length T*; *abs-is-lms T i*⟧ ⟹ $i \in$ *set SA*
⟨*proof*⟩

lemma *lms-all-inserted-imp-exhaustive*:
assumes *lms-inserted-inv LMS SA LMS* []
and *set LMS* = {*i*. *abs-is-lms T i*}
shows *lms-type-exhaustive T SA*
⟨*proof*⟩

lemma *lms-type-exhaustive-imp-lms-bucket-subset*:
assumes *lms-type-exhaustive T SA*
and $b \leq \alpha$ (*Max* (*set T*))

shows *lms-bucket* α *T b* $\subseteq$ *set SA*
$\langle proof \rangle$

lemma *lms-B-val*:
  assumes $\forall\, i < length\ SA.\ SA\ !\ i = length\ T$
  and *distinct LMS*
  and *lms-bucket-init* α *T B*
  and $length\ SA = length\ T$
  and *strict-mono* α
  and $set\ LMS = \{i.\ abs\text{-}is\text{-}lms\ T\ i\}$
  and $bucket\text{-}insert\text{-}abs'\ \alpha\ T\ B\ SA\ []\ LMS = (SA',\ B',\ gs')$
  and $b \leq \alpha\ (Max\ (set\ T))$
shows $B'\ !\ b = lms\text{-}bucket\text{-}start\ \alpha\ T\ b$
$\langle proof \rangle$

64 Postconditions

definition *lms-vals-post* :: $('a :: \{linorder,\ order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow bool$
where
lms-vals-post α *T SA* =
  $(\forall\, b \leq \alpha\ (Max\ (set\ T)).$
    $lms\text{-}bucket\ \alpha\ T\ b = set\ (list\text{-}slice\ SA\ (lms\text{-}bucket\text{-}start\ \alpha\ T\ b)\ (bucket\text{-}end\ \alpha\ T\ b))$
)

lemma *lms-vals-postD*:
  $[\![lms\text{-}vals\text{-}post\ \alpha\ T\ SA;\ b \leq \alpha\ (Max\ (set\ T))]\!] \Longrightarrow$
   $lms\text{-}bucket\ \alpha\ T\ b = set\ (list\text{-}slice\ SA\ (lms\text{-}bucket\text{-}start\ \alpha\ T\ b)\ (bucket\text{-}end\ \alpha\ T\ b))$
  $\langle proof \rangle$

definition
  lms-pre :: $('a :: \{linorder,\ order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
  lms-pre α *T B SA LMS* $\equiv$
    $(\forall\, i < length\ SA.\ SA\ !\ i = length\ T)\ \wedge$
    $length\ SA = length\ T\ \wedge$
    $lms\text{-}bucket\text{-}init\ \alpha\ T\ B\ \wedge$
    $strict\text{-}mono\ \alpha\ \wedge$
    $distinct\ LMS\ \wedge$
    $set\ LMS = \{i.\ abs\text{-}is\text{-}lms\ T\ i\}$

lemma *lms-pre-elims*:
  $lms\text{-}pre\ \alpha\ T\ B\ SA\ LMS \Longrightarrow \forall\, i < length\ SA.\ SA\ !\ i = length\ T$
  $lms\text{-}pre\ \alpha\ T\ B\ SA\ LMS \Longrightarrow length\ SA = length\ T$
  $lms\text{-}pre\ \alpha\ T\ B\ SA\ LMS \Longrightarrow lms\text{-}bucket\text{-}init\ \alpha\ T\ B$

$lms\text{-}pre\ \alpha\ T\ B\ SA\ LMS \Longrightarrow strict\text{-}mono\ \alpha$
$lms\text{-}pre\ \alpha\ T\ B\ SA\ LMS \Longrightarrow distinct\ LMS$
$lms\text{-}pre\ \alpha\ T\ B\ SA\ LMS \Longrightarrow set\ LMS = \{i.\ abs\text{-}is\text{-}lms\ T\ i\}$
⟨*proof*⟩

lemma *lms-vals-post-holds*:
assumes $\forall i < length\ SA.\ SA\ !\ i = length\ T$
and $distinct\ LMS$
and $lms\text{-}bucket\text{-}init\ \alpha\ T\ B$
and $length\ SA = length\ T$
and $strict\text{-}mono\ \alpha$
and $set\ LMS = \{i.\ abs\text{-}is\text{-}lms\ T\ i\}$
and $bucket\text{-}insert\text{-}abs'\ \alpha\ T\ B\ SA\ []\ LMS = (SA', B', gs')$
shows $lms\text{-}vals\text{-}post\ \alpha\ T\ SA'$
⟨*proof*⟩

corollary *abs-bucket-insert-vals*:
assumes $lms\text{-}pre\ \alpha\ T\ B\ SA\ LMS$
shows $lms\text{-}vals\text{-}post\ \alpha\ T\ (abs\text{-}bucket\text{-}insert\ \alpha\ T\ B\ SA\ LMS)$
⟨*proof*⟩

definition *lms-unknowns-post*
where
$lms\text{-}unknowns\text{-}post\ \alpha\ T\ SA =$
$(\forall b \le \alpha\ (Max\ (set\ T)).$
$(\forall i.\ bucket\text{-}start\ \alpha\ T\ b \le i \wedge i < lms\text{-}bucket\text{-}start\ \alpha\ T\ b \longrightarrow SA\ !\ i = length\ T)$
$)$

lemma *lms-unknowns-postD*:
$[\![lms\text{-}unknowns\text{-}post\ \alpha\ T\ SA;\ b \le \alpha\ (Max\ (set\ T));\ bucket\text{-}start\ \alpha\ T\ b \le i;$
$i < lms\text{-}bucket\text{-}start\ \alpha\ T\ b]\!] \Longrightarrow$
$SA\ !\ i = length\ T$
⟨*proof*⟩

lemma *lms-unknowns-post-holds*:
assumes $\forall i < length\ SA.\ SA\ !\ i = length\ T$
and $distinct\ LMS$
and $lms\text{-}bucket\text{-}init\ \alpha\ T\ B$
and $length\ SA = length\ T$
and $strict\text{-}mono\ \alpha$
and $set\ LMS = \{i.\ abs\text{-}is\text{-}lms\ T\ i\}$
and $bucket\text{-}insert\text{-}abs'\ \alpha\ T\ B\ SA\ []\ LMS = (SA', B', gs')$
shows $lms\text{-}unknowns\text{-}post\ \alpha\ T\ SA'$
⟨*proof*⟩

corollary *abs-bucket-insert-unknowns*:
assumes $lms\text{-}pre\ \alpha\ T\ B\ SA\ LMS$
shows $lms\text{-}unknowns\text{-}post\ \alpha\ T\ (abs\text{-}bucket\text{-}insert\ \alpha\ T\ B\ SA\ LMS)$

⟨*proof*⟩

corollary *abs-bucket-insert-values*:
 assumes *lms-pre* α *T B SA LMS*
 shows $\forall\, b \leq \alpha$ (*Max* (*set T*)).
 ($\forall\, i$. *bucket-start* α *T b* $\leq i \wedge i <$ *lms-bucket-start* α *T b* $\longrightarrow$ (*abs-bucket-insert* α *T B SA LMS*) ! *i* = *length T*) $\wedge$
 lms-bucket α *T b* = *set* (*list-slice* (*abs-bucket-insert* α *T B SA LMS*) (*lms-bucket-start* α *T b*) (*bucket-end* α *T b*))
 ⟨*proof*⟩

lemma *lms-lms-prefix-sorted-holds*:
 assumes $\forall\, i <$ *length SA*. *SA* ! *i* = *length T*
 and *distinct LMS*
 and *lms-bucket-init* α *T B*
 and *length SA* = *length T*
 and *strict-mono* α
 and *set LMS* = {*i*. *abs-is-lms T i*}
 and *bucket-insert-abs′* α *T B SA* [] *LMS* = (*SA′*, *B′*, *gs′*)
shows *ordlistns.sorted* (*map* (*lms-prefix T*) (*filter* (λx. $x <$ *length T*) *SA′*))
⟨*proof*⟩

lemma *lms-suffix-sorted-holds*:
 assumes $\forall\, i <$ *length SA*. *SA* ! *i* = *length T*
 and *distinct LMS*
 and *lms-bucket-init* α *T B*
 and *length SA* = *length T*
 and *strict-mono* α
 and *set LMS* = {*i*. *abs-is-lms T i*}
 and *bucket-insert-abs′* α *T B SA* [] *LMS* = (*SA′*, *B′*, *gs′*)
 and *ordlistns.sorted* (*map* (*suffix T*) (*rev LMS*))
shows *ordlistns.sorted* (*map* (*suffix T*) (*filter* (λx. $x <$ *length T*) *SA′*))
⟨*proof*⟩

lemma *lms-bot-is-first*:
 assumes $\forall\, i <$ *length SA*. *SA* ! *i* = *length T*
 and *distinct LMS*
 and *lms-bucket-init* α *T B*
 and *length SA* = *length T*
 and *strict-mono* α
 and *set LMS* = {*i*. *abs-is-lms T i*}
 and *bucket-insert-abs′* α *T B SA* [] *LMS* = (*SA′*, *B′*, *gs′*)
 and *valid-list T*
 and *length T* = *Suc* (*Suc n*)
 and α *bot* = *0*
shows *SA′* ! *0* = *Suc n*
⟨*proof*⟩

corollary *abs-bucket-insert-bot-first*:

assumes *lms-pre* α *T B SA LMS*
and *valid-list T*
and *length T* = *Suc* (*Suc n*)
and α *bot* = *0*
shows (*abs-bucket-insert* α *T B SA LMS*) ! *0* = *Suc n*
⟨*proof*⟩
theorem *lms-prefix-sorted-bucket*:
assumes *lms-pre* α *T B SA LMS*
and $b \leq \alpha$ (*Max* (*set T*))
shows *ordlistns.sorted* (*map* (*lms-prefix T*)
(*list-slice* (*abs-bucket-insert* α *T B SA LMS*) (*lms-bucket-start* α *T b*)
(*bucket-end* α *T b*)))
(**is** *ordlistns.sorted* (*map ?f ?SA*))
⟨*proof*⟩
theorem *lms-suffix-sorted-bucket*:
assumes *lms-pre* α *T B SA LMS*
and *ordlistns.sorted* (*map* (*suffix T*) (*rev LMS*))
and $b \leq \alpha$ (*Max* (*set T*))
shows *ordlistns.sorted* (*map* (*suffix T*)
(*list-slice* (*abs-bucket-insert* α *T B SA LMS*) (*lms-bucket-start* α *T b*)
(*bucket-end* α *T b*)))
(**is** *ordlistns.sorted* (*map ?f ?SA*))
⟨*proof*⟩

end
theory *Abs-Induce-L-Verification*
imports *../abs−def/Abs-SAIS*
begin

65 Abstract Induce L-types Simple Properties

lemma *abs-induce-l-step-ex*:
$\exists$ *B′ SA′ i′*. *abs-induce-l-step a b* = (*B′*, *SA′*, *i′*)
⟨*proof*⟩

lemma *abs-induce-l-step-B-length*:
abs-induce-l-step (*B*, *SA*, *i*) (α, *T*) = (*B′*, *SA′*, *i′*) $\Longrightarrow$ *length B′* = *length B*
⟨*proof*⟩

lemma *abs-induce-l-step-SA-length*:
abs-induce-l-step (*B*, *SA*, *i*) (α, *T*) = (*B′*, *SA′*, *i′*) $\Longrightarrow$ *length SA′* = *length SA*
⟨*proof*⟩

lemma *abs-induce-l-step-Suc*:
$\exists$ *B′ SA′*. *abs-induce-l-step* (*B*, *SA*, *i*) (α, *T*) = (*B′*, *SA′*, *Suc i*)
⟨*proof*⟩

lemma *abs-induce-l-step-B-val-1*:
⟦*length SA* $\leq$ *i*; *abs-induce-l-step* (*B*, *SA*, *i*) (α, *T*) = (*B′*, *SA′*, *i′*)⟧ $\Longrightarrow$

$B' = B$
$[\![i < \mathit{length}\ SA;\ \mathit{length}\ T \le SA\ !\ i;\ \mathit{abs\text{-}induce\text{-}l\text{-}step}\ (B,\ SA,\ i)\ (\alpha,\ T) = (B', SA',\ i')]\!] \Longrightarrow$
$B' = B$
$[\![i < \mathit{length}\ SA;\ SA\ !\ i < \mathit{length}\ T;\ SA\ !\ i = 0;$
$\mathit{abs\text{-}induce\text{-}l\text{-}step}\ (B,\ SA,\ i)\ (\alpha,\ T) = (B',\ SA',\ i')]\!] \Longrightarrow$
$B' = B$
$[\![i < \mathit{length}\ SA;\ SA\ !\ i < \mathit{length}\ T;\ SA\ !\ i = \mathit{Suc}\ j;\ \mathit{suffix\text{-}type}\ T\ j = \mathit{S\text{-}type};$
$\mathit{abs\text{-}induce\text{-}l\text{-}step}\ (B,\ SA,\ i)\ (\alpha,\ T) = (B',\ SA',\ i')]\!] \Longrightarrow$
$B' = B$
⟨*proof*⟩

lemma *abs-induce-l-step-B-val-2*:
$[\![\mathit{strict\text{-}mono}\ \alpha;$
$\alpha\ (\mathit{Max}\ (\mathit{set}\ T)) < \mathit{length}\ B;$
$i < \mathit{length}\ SA;$
$SA\ !\ i < \mathit{length}\ T;$
$SA\ !\ i = \mathit{Suc}\ j;$
$\mathit{suffix\text{-}type}\ T\ j = \mathit{L\text{-}type};$
$\mathit{abs\text{-}induce\text{-}l\text{-}step}\ (B,\ SA,\ i)\ (\alpha,\ T) = (B',\ SA',\ i')]\!] \Longrightarrow$
$B' = B[\alpha\ (T\ !\ j) := \mathit{Suc}\ (B\ !\ \alpha\ (T\ !\ j))]$
⟨*proof*⟩

lemma *repeat-abs-induce-l-step-index*:
$\exists\, B'\ SA'.\ \mathit{repeat}\ n\ \mathit{abs\text{-}induce\text{-}l\text{-}step}\ (B,\ SA,\ m)\ (\alpha,\ T) = (B',\ SA',\ n + m)$
⟨*proof*⟩

lemma *abs-induce-l-step-lengths*:
$\mathit{abs\text{-}induce\text{-}l\text{-}step}\ (B,\ SA,\ i)\ (\alpha,\ T) = (B',\ SA',\ i') \Longrightarrow$
$\mathit{length}\ B' = \mathit{length}\ B \wedge \mathit{length}\ SA' = \mathit{length}\ SA$
⟨*proof*⟩

lemma *repeat-abs-induce-l-step-lengths*:
$\mathit{repeat}\ n\ \mathit{abs\text{-}induce\text{-}l\text{-}step}\ (B,\ SA,\ i)\ (\alpha,\ T) = (B',\ SA',\ i') \Longrightarrow$
$\mathit{length}\ B' = \mathit{length}\ B \wedge \mathit{length}\ SA' = \mathit{length}\ SA$
⟨*proof*⟩

lemma *abs-induce-l-index*:
$\exists\, B'\ SA'.\ \mathit{abs\text{-}induce\text{-}l\text{-}base}\ \alpha\ T\ B\ SA = (B',\ SA',\ \mathit{length}\ T)$
⟨*proof*⟩

lemma *abs-induce-l-length*:
$\mathit{length}\ (\mathit{abs\text{-}induce\text{-}l}\ \alpha\ T\ B\ SA) = \mathit{length}\ SA$
⟨*proof*⟩

66 Precondition Definitions

definition *lms-init* :: $('a :: \{\mathit{linorder},\mathit{order\text{-}bot}\} \Rightarrow \mathit{nat}) \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{nat}\ \mathit{list} \Rightarrow \mathit{bool}$

where
lms-init α *T SA* =
($\forall b \leq \alpha$ (*Max* (*set T*)).
lms-bucket α *T b* =
set (*list-slice SA* (*lms-bucket-start* α *T b*) (*bucket-end* α *T b*))
)

lemma *lms-init-D*:
⟦*lms-init* α *T SA*; $b \leq \alpha$ (*Max* (*set T*))⟧ $\Longrightarrow$
lms-bucket α *T b* = *set* (*list-slice SA* (*lms-bucket-start* α *T b*) (*bucket-end* α *T b*))
⟨*proof*⟩

lemma *lms-init-nth*:
⟦*lms-init* α *T SA*;
$b \leq \alpha$ (*Max* (*set T*));
lms-bucket-start α *T b* $\leq i$;
$i <$ *bucket-end* α *T b*;
length SA = *length T*⟧ $\Longrightarrow$
abs-is-lms T (*SA* ! *i*) $\wedge$ α (*T* ! (*SA* ! *i*)) = *b*
⟨*proof*⟩

lemma *lms-init-imp-distinct-bucket*:
⟦*lms-init* α *T SA*;
$b \leq \alpha$ (*Max* (*set T*));
length SA = *length T*⟧ $\Longrightarrow$
distinct (*list-slice SA* (*lms-bucket-start* α *T b*) (*bucket-end* α *T b*))
⟨*proof*⟩

lemma *lms-init-imp-all-lms-in-SA*:
assumes *lms-init* α *T SA*
and *strict-mono* α
shows {*k* |*k*. *abs-is-lms T k*} $\subseteq$ *set SA*
⟨*proof*⟩

definition *s-init* :: (*'a* :: {*linorder*,*order-bot*} $\Rightarrow$ *nat*) $\Rightarrow$ *'a list* $\Rightarrow$ *nat list* $\Rightarrow$ *bool*
where
s-init α *T SA* =
($\forall b \leq \alpha$ (*Max* (*set T*)).
$\forall i <$ *length SA*. *l-bucket-end* α *T b* $\leq i \wedge i <$ *lms-bucket-start* α *T b* $\longrightarrow$ *SA* ! *i* = *length T*
)

lemma *s-init-D*:
⟦*s-init* α *T SA*;
$b \leq \alpha$ (*Max* (*set T*));
$i <$ *length SA*;
l-bucket-end α *T b* $\leq i$;

$i < lms\text{-}bucket\text{-}start\ \alpha\ T\ b]\!] \Longrightarrow$
$SA\ !\ i = length\ T$
⟨proof⟩

definition *l-init* :: ('*a* :: {*linorder*,*order-bot*} ⇒ *nat*) ⇒ '*a list* ⇒ *nat list* ⇒ *bool*
where
l-init α *T SA* =
(∀ *b* ≤ α (*Max* (*set T*)).
∀ *i* < *length SA*. *bucket-start* α *T b* ≤ *i* ∧ *i* < *l-bucket-end* α *T b* ⟶ *SA* ! *i* = *length T*
)

lemma *l-init-D*:
[[*l-init* α *T SA*;
b ≤ α (*Max* (*set T*));
i < *length SA*;
bucket-start α *T b* ≤ *i*;
i < *l-bucket-end* α *T b*]] ⟹
SA ! *i* = *length T*
⟨proof⟩

lemma *init-imp-lms-range*:
assumes *lms-init* α *T SA*
and *l-init* α *T SA*
and *s-init* α *T SA*
and *length SA* = *length T*
and *strict-mono* α
and *i* < *length SA*
and *SA* ! *i* = *j*
and *j* < *length T*
shows *lms-bucket-start* α *T* (α (*T* ! *j*)) ≤ *i* ∧ *i* < *bucket-end* α *T* (α (*T* ! *j*))
⟨proof⟩

lemma *init-imp-only-lms-types*:
assumes *lms-init* α *T SA*
and *l-init* α *T SA*
and *s-init* α *T SA*
and *length SA* = *length T*
and *strict-mono* α
shows ∀ *i* < *length SA*. *SA* ! *i* < *length T* ⟶ *abs-is-lms T* (*SA* ! *i*)
⟨proof⟩

lemma *init-imp-only-s-types*:
assumes *lms-init* α *T SA*
and *l-init* α *T SA*
and *s-init* α *T SA*
and *length SA* = *length T*
and *strict-mono* α
shows ∀ *i* < *length SA*. *SA* ! *i* < *length T* ⟶ *suffix-type T* (*SA* ! *i*) = *S-type*

⟨*proof*⟩

definition *lms-sorted-init* ::
$('a :: \{linorder,\ order\text{-}bot\} \Rightarrow nat) \Rightarrow$
$('a\ list \Rightarrow nat \Rightarrow 'a\ list) \Rightarrow$
$'a\ list \Rightarrow$
$nat\ list \Rightarrow$
$bool$
where
lms-sorted-init α f T SA =
$(\forall\, b \le \alpha\ (Max\ (set\ T)).$
ordlistns.sorted (*map* (f T) (*list-slice* SA (*lms-bucket-start* α T b) (*bucket-end* α T b)))
)

lemma *lms-sorted-init-D*:
⟦*lms-sorted-init* α f T SA; $b \le \alpha\ (Max\ (set\ T))$⟧ $\Longrightarrow$
ordlistns.sorted (*map* (f T) (*list-slice* SA (*lms-bucket-start* α T b) (*bucket-end* α T b)))
⟨*proof*⟩

definition *l-suffix-sorted-pre* ::
$('a :: \{linorder,\ order\text{-}bot\} \Rightarrow nat) \Rightarrow\ 'a\ list \Rightarrow nat\ list \Rightarrow bool$
where
l-suffix-sorted-pre α T SA =
$(\forall\, b \le \alpha\ (Max\ (set\ T)).$
ordlistns.sorted (*map* (*suffix* T) (*list-slice* SA (*lms-bucket-start* α T b) (*bucket-end* α T b)))
)

lemma *l-suffix-sorted-preD*:
⟦*l-suffix-sorted-pre* α T SA; $b \le \alpha\ (Max\ (set\ T))$⟧ $\Longrightarrow$
ordlistns.sorted (*map* (*suffix* T) (*list-slice* SA (*lms-bucket-start* α T b) (*bucket-end* α T b)))
⟨*proof*⟩

definition *l-prefix-sorted-pre* ::
$('a :: \{linorder,\ order\text{-}bot\} \Rightarrow nat) \Rightarrow\ 'a\ list \Rightarrow nat\ list \Rightarrow bool$
where
l-prefix-sorted-pre α T SA =
$(\forall\, b \le \alpha\ (Max\ (set\ T)).$
ordlistns.sorted (*map* (*lms-prefix* T) (*list-slice* SA (*lms-bucket-start* α T b) (*bucket-end* α T b)))
)

lemma *l-prefix-sorted-preD*:
⟦*l-prefix-sorted-pre* α T SA; $b \le \alpha\ (Max\ (set\ T))$⟧ $\Longrightarrow$
ordlistns.sorted (*map* (*lms-prefix* T) (*list-slice* SA (*lms-bucket-start* α T b) (*bucket-end* α T b)))

⟨*proof*⟩

definition *l-perm-pre* ::
  $('a :: \{linorder, order\text{-}bot\} \Rightarrow nat) \Rightarrow$
   $'a\ list \Rightarrow$
   $nat\ list \Rightarrow$
   $nat\ list \Rightarrow$
   $bool$
  where
$l\text{-}perm\text{-}pre\ \alpha\ T\ B\ SA =$
  $(lms\text{-}init\ \alpha\ T\ SA\ \wedge$
   $l\text{-}init\ \alpha\ T\ SA\ \wedge$
   $s\text{-}init\ \alpha\ T\ SA\ \wedge$
   $l\text{-}bucket\text{-}init\ \alpha\ T\ B\ \wedge$
   $T \neq []\ \wedge$
   $strict\text{-}mono\ \alpha\ \wedge$
   $length\ SA = length\ T\ \wedge$
   $\alpha\ (Max\ (set\ T)) < length\ B)$

lemma *l-perm-pre-elims*:
  $l\text{-}perm\text{-}pre\ \alpha\ T\ B\ SA \Longrightarrow lms\text{-}init\ \alpha\ T\ SA$
  $l\text{-}perm\text{-}pre\ \alpha\ T\ B\ SA \Longrightarrow l\text{-}init\ \alpha\ T\ SA$
  $l\text{-}perm\text{-}pre\ \alpha\ T\ B\ SA \Longrightarrow s\text{-}init\ \alpha\ T\ SA$
  $l\text{-}perm\text{-}pre\ \alpha\ T\ B\ SA \Longrightarrow l\text{-}bucket\text{-}init\ \alpha\ T\ B$
  $l\text{-}perm\text{-}pre\ \alpha\ T\ B\ SA \Longrightarrow T \neq []$
  $l\text{-}perm\text{-}pre\ \alpha\ T\ B\ SA \Longrightarrow strict\text{-}mono\ \alpha$
  $l\text{-}perm\text{-}pre\ \alpha\ T\ B\ SA \Longrightarrow length\ SA = length\ T$
  $l\text{-}perm\text{-}pre\ \alpha\ T\ B\ SA \Longrightarrow \alpha\ (Max\ (set\ T)) < length\ B$
  ⟨*proof*⟩

67 Invariant Definitions

This section contains all the various invariants that we need for the *abs-induce-l* subroutine.

67.1 Distinctness

definition *l-distinct-inv* :: $('a :: \{linorder, order\text{-}bot\})\ list \Rightarrow nat\ list \Rightarrow bool$
  where
$l\text{-}distinct\text{-}inv\ T\ SA = distinct\ (filter\ (\lambda x.\ x < length\ T)\ SA)$

lemma *l-distinct-inv-D*:
  assumes $l\text{-}distinct\text{-}inv\ T\ SA$
  and $i < length\ SA$
  and $j < length\ SA$
  and $i \neq j$
  and $SA\ !\ i < length\ T$
  and $SA\ !\ j < length\ T$

shows $SA ! i \neq SA ! j$
⟨*proof*⟩

67.2 Predecessor

definition *l-pred-inv* :: $('a :: \{linorder, order\text{-}bot\})\ list \Rightarrow nat\ list \Rightarrow nat \Rightarrow bool$
where
$l\text{-}pred\text{-}inv\ T\ SA\ k =$
$(\forall i < length\ SA.\ SA ! i < length\ T \wedge suffix\text{-}type\ T\ (SA ! i) = L\text{-}type \longrightarrow$
$(\exists j < length\ SA.\ SA ! j = Suc\ (SA ! i) \wedge j < i \wedge j < k))$

lemma *l-pred-inv-D*:
$[\![l\text{-}pred\text{-}inv\ T\ SA\ k;\ i < length\ SA;\ SA ! i < length\ T;\ suffix\text{-}type\ T\ (SA ! i) = L\text{-}type]\!] \Longrightarrow$
$\exists j < length\ SA.\ SA ! j = Suc\ (SA ! i) \wedge SA ! j < length\ T \wedge j < i \wedge j < k$
⟨*proof*⟩

67.3 L Bucket Ptr

We prove that the pointer for each bucket is related to the number of L-types currently in SA. That is, if we subtract the original pointer with the current, we should have the number of L-types currently in SA for each symbol.

definition *cur-l-types* ::
$('a :: \{linorder, order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat \Rightarrow nat\ set$
where
$cur\text{-}l\text{-}types\ \alpha\ T\ SA\ b = \{i|i.\ i \in set\ SA \wedge i \in l\text{-}bucket\ \alpha\ T\ b\ \}$

definition *num-l-types* ::
$('a :: \{linorder, order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat \Rightarrow nat$
where
$num\text{-}l\text{-}types\ \alpha\ T\ SA\ b = card\ (cur\text{-}l\text{-}types\ \alpha\ T\ SA\ b)$

definition *l-bucket-ptr-inv* ::
$('a :: \{linorder, order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
$l\text{-}bucket\text{-}ptr\text{-}inv\ \alpha\ T\ B\ SA \equiv$
$(\forall b \leq \alpha\ (Max\ (set\ T)).\ B ! b = bucket\text{-}start\ \alpha\ T\ b + num\text{-}l\text{-}types\ \alpha\ T\ SA\ b)$

lemma *l-bucket-ptr-inv-D*:
$[\![l\text{-}bucket\text{-}ptr\text{-}inv\ \alpha\ T\ B\ SA;\ b \leq \alpha\ (Max\ (set\ T))]\!] \Longrightarrow$
$B ! b = bucket\text{-}start\ \alpha\ T\ b + num\text{-}l\text{-}types\ \alpha\ T\ SA\ b$
⟨*proof*⟩

67.4 Unknowns

definition *l-unknowns-inv* ::
$('a :: \{linorder, order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
$l\text{-}unknowns\text{-}inv\ \alpha\ T\ B\ SA \equiv$

$(\forall a \leq \alpha\ (Max\ (set\ T)).\ \forall k.\ B\ !\ a \leq k \wedge k < l\text{-}bucket\text{-}end\ \alpha\ T\ a \longrightarrow SA\ !\ k = length\ T)$

lemma *l-unknowns-inv-D*:
$[\![l\text{-}unknowns\text{-}inv\ \alpha\ T\ B\ SA;\ b \leq \alpha\ (Max\ (set\ T));\ B\ !\ b \leq k;\ k < l\text{-}bucket\text{-}end\ \alpha\ T\ b]\!] \Longrightarrow$
$SA\ !\ k = length\ T$
⟨*proof*⟩

67.5 Indexes

definition *l-index-inv* ::
$('a :: \{linorder,\ order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
$l\text{-}index\text{-}inv\ \alpha\ T\ B\ SA \equiv$
$(\forall i < length\ SA.$
$(\forall j.\ SA\ !\ i = Suc\ j \wedge Suc\ j < length\ T \wedge suffix\text{-}type\ T\ j = L\text{-}type \longrightarrow$
$i < B\ !\ (\alpha\ (T\ !\ j))$
$)$
$)$

lemma *l-index-inv-D*:
$[\![l\text{-}index\text{-}inv\ \alpha\ T\ B\ SA;\ i < length\ SA;\ SA\ !\ i = Suc\ j;\ Suc\ j < length\ T;\ suffix\text{-}type\ T\ j = L\text{-}type]\!] \Longrightarrow$
$i < B\ !\ (\alpha\ (T\ !\ j))$
⟨*proof*⟩

67.6 Unchanged

definition *l-unchanged-inv* ::
$('a :: \{linorder,order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
$l\text{-}unchanged\text{-}inv\ \alpha\ T\ SA\ SA' =$
$((length\ SA' = length\ SA) \wedge$
$(\forall b \leq \alpha\ (Max\ (set\ T)).$
$(\forall i < length\ SA.\ l\text{-}bucket\text{-}end\ \alpha\ T\ b \leq i \wedge i < bucket\text{-}end\ \alpha\ T\ b \longrightarrow SA\ !\ i = SA'\ !\ i)$
$))$

lemma *l-unchanged-inv-trans*:
$[\![l\text{-}unchanged\text{-}inv\ \alpha\ T\ SA0\ SA1;\ l\text{-}unchanged\text{-}inv\ \alpha\ T\ SA1\ SA2]\!] \Longrightarrow$
$l\text{-}unchanged\text{-}inv\ \alpha\ T\ SA0\ SA2$
⟨*proof*⟩

lemma *l-unchanged-inv-D*:
$[\![l\text{-}unchanged\text{-}inv\ \alpha\ T\ SA\ SA';\ length\ SA' = length\ SA;\ b \leq \alpha\ (Max\ (set\ T));$
$i < length\ SA;\ l\text{-}bucket\text{-}end\ \alpha\ T\ b \leq i;\ i < bucket\text{-}end\ \alpha\ T\ b]\!] \Longrightarrow$
$SA\ !\ i = SA'\ !\ i$
⟨*proof*⟩

67.7 L Locations

definition *l-locations-inv* ::
  $('a :: \{linorder,order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
  where
l-locations-inv α T B SA =
  $(\forall\, b \leq \alpha\ (Max\ (set\ T)).$
    $(\forall\, i < length\ SA.\ bucket\text{-}start\ \alpha\ T\ b \leq i \wedge i < B\ !\ b \longrightarrow$
      $SA\ !\ i < length\ T \wedge suffix\text{-}type\ T\ (SA\ !\ i) = L\text{-}type \wedge \alpha\ (T\ !\ (SA\ !\ i)) = b$
)
)

lemma *l-locations-inv-D*:
  $[\![l\text{-}locations\text{-}inv\ \alpha\ T\ B\ SA;$
    $b \leq \alpha\ (Max\ (set\ T));$
    $i < length\ SA;$
    $bucket\text{-}start\ \alpha\ T\ b \leq i;$
    $i < B\ !\ b]\!] \Longrightarrow$
    $SA\ !\ i < length\ T \wedge suffix\text{-}type\ T\ (SA\ !\ i) = L\text{-}type \wedge \alpha\ (T\ !\ (SA\ !\ i)) = b$
  ⟨*proof*⟩

lemma *l-locations-list-slice*:
  assumes *l-locations-inv* α T B SA
  and $b \leq \alpha\ (Max\ (set\ T))$
shows $set\ (list\text{-}slice\ SA\ (bucket\text{-}start\ \alpha\ T\ b)\ (B\ !\ b)) \subseteq l\text{-}bucket\ \alpha\ T\ b$
  (**is** $set\ ?xs \subseteq l\text{-}bucket\ \alpha\ T\ b$)
⟨*proof*⟩

67.8 Seen

In this section, we prove that the seen invariant is maintained. In English, this invariant states for all L-type suffixes, excluding the one that starts at position 0, in the suffix array (SA) and that are less than the current index, their left neighbour is also in SA.

definition *l-seen-inv* :: $('a :: \{linorder,\ order\text{-}bot\})\ list \Rightarrow nat\ list \Rightarrow nat \Rightarrow bool$
  where
$l\text{-}seen\text{-}inv\ T\ SA\ n \equiv \forall\, i < n.\ i < length\ SA \wedge SA\ !\ i < length\ T \longrightarrow$
    $(\forall\, j.\ SA\ !\ i = Suc\ j \wedge suffix\text{-}type\ T\ j = L\text{-}type \longrightarrow$
      $(\exists\, k < length\ SA.\ SA\ !\ k = j))$

lemma *l-seen-inv-nth-ex*:
  $[\![l\text{-}seen\text{-}inv\ T\ SA\ n;\ i < n;\ i < length\ SA;\ SA\ !\ i < length\ T;\ SA\ !\ i = Suc\ j;$
    $suffix\text{-}type\ T\ j = L\text{-}type]\!] \Longrightarrow$
  $\exists\, k < length\ SA.\ SA\ !\ k = j$
  ⟨*proof*⟩

67.9 Sortedness

definition *abs-induce-l-sorted* ::

$(('a :: \{linorder, order\text{-}bot\})\ list \Rightarrow nat \Rightarrow 'a\ list) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow bool$
where
abs-induce-l-sorted f T SA = *ordlistns.sorted* (*map* (*f T*) (*filter* (λ*x*. *x* < *length T*) *SA*))

lemma *abs-induce-l-sorted-nth*:
assumes *abs-induce-l-sorted f T SA*
and $i < j$
and $j < length\ SA$
and $SA\ !\ i < length\ T$
and $SA\ !\ j < length\ T$
shows *list-less-eq-ns* (*f T* (*SA* ! *i*)) (*f T* (*SA* ! *j*))
⟨*proof*⟩

definition *l-suffix-sorted-inv* ::
$(('a :: \{linorder, order\text{-}bot\}) \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
l-suffix-sorted-inv α *T B SA* =
(∀ *b* ≤ α (*Max* (*set T*)).
ordlistns.sorted (*map* (*suffix T*) (*list-slice SA* (*bucket-start* α *T b*) (*B* ! *b*))))

lemma *l-suffix-sorted-invD*:
⟦*l-suffix-sorted-inv* α *T B SA*; *b* ≤ α (*Max* (*set T*))⟧ ⟹
ordlistns.sorted (*map* (*suffix T*) (*list-slice SA* (*bucket-start* α *T b*) (*B* ! *b*)))
⟨*proof*⟩

definition *l-prefix-sorted-inv* ::
$(('a :: \{linorder, order\text{-}bot\}) \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
l-prefix-sorted-inv α *T B SA* =
(∀ *b* ≤ α (*Max* (*set T*)).
ordlistns.sorted (*map* (*lms-prefix T*) (*list-slice SA* (*bucket-start* α *T b*) (*B* ! *b*))))

lemma *l-prefix-sorted-invD*:
⟦*l-prefix-sorted-inv* α *T B SA*; *b* ≤ α (*Max* (*set T*))⟧ ⟹
ordlistns.sorted (*map* (*lms-prefix T*) (*list-slice SA* (*bucket-start* α *T b*) (*B* ! *b*)))
⟨*proof*⟩

67.10 Permutation

definition *l-perm-inv* ::
('*a* :: {*linorder*, *order-bot*} ⇒ *nat*) ⇒
'*a list* ⇒
nat list ⇒
nat list ⇒
nat list ⇒
nat ⇒
bool
where

$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \equiv$
$\alpha\ (Max\ (set\ T)) < length\ B\ \wedge$
$length\ SA = length\ T\ \wedge$
$length\ SA' = length\ SA\ \wedge$
$l\text{-}distinct\text{-}inv\ T\ SA'\ \wedge$
$l\text{-}unknowns\text{-}inv\ \alpha\ T\ B\ SA'\ \wedge$
$l\text{-}bucket\text{-}ptr\text{-}inv\ \alpha\ T\ B\ SA'\ \wedge$
$l\text{-}index\text{-}inv\ \alpha\ T\ B\ SA'\ \wedge$
$l\text{-}unchanged\text{-}inv\ \alpha\ T\ SA\ SA'\ \wedge$
$l\text{-}locations\text{-}inv\ \alpha\ T\ B\ SA'\ \wedge$
$l\text{-}pred\text{-}inv\ T\ SA'\ i\ \wedge$
$l\text{-}seen\text{-}inv\ T\ SA'\ i\ \wedge$
$strict\text{-}mono\ \alpha\ \wedge$
$T \neq []\ \wedge$
$lms\text{-}init\ \alpha\ T\ SA\ \wedge$
$s\text{-}init\ \alpha\ T\ SA$

lemma *l-perm-inv-elims*:
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow \alpha\ (Max\ (set\ T)) < length\ B$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow length\ SA = length\ T$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow length\ SA' = length\ SA$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow l\text{-}distinct\text{-}inv\ T\ SA'$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow l\text{-}unknowns\text{-}inv\ \alpha\ T\ B\ SA'$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow l\text{-}bucket\text{-}ptr\text{-}inv\ \alpha\ T\ B\ SA'$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow l\text{-}index\text{-}inv\ \alpha\ T\ B\ SA'$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow l\text{-}unchanged\text{-}inv\ \alpha\ T\ SA\ SA'$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow l\text{-}locations\text{-}inv\ \alpha\ T\ B\ SA'$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow l\text{-}pred\text{-}inv\ T\ SA'\ i$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow l\text{-}seen\text{-}inv\ T\ SA'\ i$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow strict\text{-}mono\ \alpha$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow T \neq []$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow lms\text{-}init\ \alpha\ T\ SA$
$l\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ i \Longrightarrow s\text{-}init\ \alpha\ T\ SA$
⟨*proof*⟩

68 Invariant Helpers

68.1 Distinctness of New Insert

We prove that the next item to be inserted cannot already be in the suffix array.

lemma *l-distinct-pred-inv-helper*:
assumes $i < length\ SA$
and $SA\ !\ i = Suc\ j$
and $Suc\ j < length\ T$
and $suffix\text{-}type\ T\ j = L\text{-}type$
and $l\text{-}distinct\text{-}inv\ T\ SA$
and $l\text{-}pred\text{-}inv\ T\ SA\ i$

shows $j \notin$ *set SA*
⟨*proof*⟩

lemma *l-distinct-slice*:
assumes *l-distinct-inv T SA*
and *l-locations-inv* α *T B SA*
and *length SA* = *length T*
and $b \leq \alpha$ (*Max* (*set T*))
shows *distinct* (*list-slice SA* (*bucket-start* α *T b*) (*B* ! *b*))
(**is** *distinct ?xs*)
⟨*proof*⟩

68.2 Bucket Ranges

lemma *num-l-types-le-l-bucket-size*:
num-l-types α *T SA b* $\leq$ *l-bucket-size* α *T b*
⟨*proof*⟩

lemma *num-l-types-less-l-bucket-size*:
⟦$j \notin$ *set SA*; *suffix-type T j* = *L-type*; α (*T* ! *j*) = *b*; $j <$ *length T*⟧ $\Longrightarrow$
num-l-types α *T SA b* < *l-bucket-size* α *T b*
⟨*proof*⟩

lemma *l-bucket-ptr-inv-imp-le-l-bucket-end*:
⟦*l-bucket-ptr-inv* α *T B SA*; $b \leq \alpha$ (*Max* (*set T*))⟧ $\Longrightarrow$
B ! *b* $\leq$ *l-bucket-end* α *T b*
⟨*proof*⟩

lemma *l-bucket-ptr-inv-imp-less-l-bucket-end*:
⟦*l-bucket-ptr-inv* α *T B SA*; $j <$ *length T*; *suffix-type T j* = *L-type*; $j \notin$ *set SA*;
strict-mono α⟧ $\Longrightarrow$
B ! (α (*T* ! *j*)) < *l-bucket-end* α *T* (α (*T* ! *j*))
⟨*proof*⟩

lemma *bucket-size-imp-less-length*:
⟦*l-bucket-ptr-inv* α *T B SA*; $j <$ *length T*; *suffix-type T j* = *L-type*; $j \notin$ *set SA*;
strict-mono α⟧ $\Longrightarrow$
B ! (α (*T* ! *j*)) < *length T*
⟨*proof*⟩

lemma *l-bucket-ptr-inv-imp-ge-bucket-start*:
⟦*l-bucket-ptr-inv* α *T B SA*; $b \leq \alpha$ (*Max* (*set T*))⟧ $\Longrightarrow$
bucket-start α *T b* $\leq$ *B* ! *b*
⟨*proof*⟩

lemma *l-bucket-ptr-inv-le-bucket-pointers*:
⟦*l-bucket-ptr-inv* α *T B SA*; $a < b$; $b \leq \alpha$ (*Max* (*set T*))⟧ $\Longrightarrow$
B ! *a* $\leq$ *B* ! *b*
⟨*proof*⟩

68.3 No Overwrite

We prove that the next location is set as unknown.

lemma *l-unknowns-l-bucket-ptr-inv-helper*:
$[\![$*l-unknowns-inv* α *T B SA*;
l-bucket-ptr-inv α *T B SA*;
$j <$ *length T*;
suffix-type T j = *L-type*;
$j \notin$ *set SA*;
strict-mono α;
$k = \alpha\ (T\ !\ j)$;
$l = B\ !\ k]\!] \Longrightarrow$
SA ! l = *length T*
⟨*proof*⟩

lemma *unchanged-slice*:
assumes *l-unchanged-inv* α *T SA0 SA*
and *length SA* = *length SA0*
and *length SA* = *length T*
and $b \leq \alpha$ (*Max* (*set T*))
and *l-bucket-end* α *T b* $\leq i$
and $j \leq$ *bucket-end* α *T b*
shows *list-slice SA0 i j* = *list-slice SA i j*
⟨*proof*⟩

lemma *lms-init-unchanged*:
assumes *l-unchanged-inv* α *T SA0 SA*
and *length SA* = *length SA0*
and *length SA* = *length T*
and *lms-init* α *T SA0*
shows *lms-init* α *T SA*
⟨*proof*⟩

lemma *s-init-unchanged*:
assumes *l-unchanged-inv* α *T SA0 SA*
and *length SA* = *length SA0*
and *length SA* = *length T*
and *s-init* α *T SA0*
shows *s-init* α *T SA*
⟨*proof*⟩

lemma *l-suffix-sorted-pre-maintained*:
assumes *l-unchanged-inv* α *T SA0 SA*
and *length SA* = *length SA0*
and *length SA* = *length T*
and *l-suffix-sorted-pre* α *T SA0*
shows *l-suffix-sorted-pre* α *T SA*
⟨*proof*⟩

lemma *l-prefix-sorted-pre-maintained*:
assumes *l-unchanged-inv* α *T SA0 SA*
and *length SA* = *length SA0*
and *length SA* = *length T*
and *l-prefix-sorted-pre* α *T SA0*
shows *l-prefix-sorted-pre* α *T SA*
⟨*proof*⟩

lemma *unknown-range-values*:
assumes *l-unchanged-inv* α *T SA0 SA*
and *l-unknowns-inv* α *T B SA*
and *length SA* = *length SA0*
and *length SA* = *length T*
and *lms-init* α *T SA0*
and *s-init* α *T SA0*
and $b \leq \alpha$ (*Max* (*set T*))
and *B* ! $b \leq i$
and $i <$ *lms-bucket-start* α *T b*
shows *SA* ! i = *length T*
⟨*proof*⟩

68.4 Bucket Values

lemma *same-bucket-same-hd*:
assumes *l-unchanged-inv* α *T SA0 SA*
and *l-locations-inv* α *T B SA*
and *l-bucket-ptr-inv* α *T B SA*
and *l-unknowns-inv* α *T B SA*
and *length SA* = *length T*
and *length SA* = *length SA0*
and *lms-init* α *T SA0*
and *s-init* α *T SA0*
and $b \leq \alpha$ (*Max* (*set T*))
and $i <$ *length SA*
and *SA* ! $i <$ *length T*
and *bucket-start* α *T b* $\leq i$
and $i <$ *bucket-end* α *T b*
shows α (*T* ! (*SA* ! i)) = b
⟨*proof*⟩

lemma *same-hd-same-bucket*:
assumes *l-unchanged-inv* α *T SA0 SA*
and *l-locations-inv* α *T B SA*
and *l-bucket-ptr-inv* α *T B SA*
and *l-unknowns-inv* α *T B SA*
and *strict-mono* α
and *length SA* = *length T*
and *length SA* = *length SA0*

and *lms-init* α *T SA0*
and *s-init* α *T SA0*
and $i <$ *length SA*
and *SA* ! $i <$ *length T*
and $b = \alpha$ (*T* ! (*SA* ! *i*))
shows *bucket-start* α *T b* $\leq i \wedge i <$ *bucket-end* α *T b*
⟨*proof*⟩

lemma *less-bucket-less-hd*:
assumes *l-unchanged-inv* α *T SA0 SA*
and *l-locations-inv* α *T B SA*
and *l-bucket-ptr-inv* α *T B SA*
and *l-unknowns-inv* α *T B SA*
and *strict-mono* α
and *length SA* = *length T*
and *length SA* = *length SA0*
and *lms-init* α *T SA0*
and *s-init* α *T SA0*
and $i <$ *length SA*
and *SA* ! $i <$ *length T*
and $i <$ *bucket-start* α *T b*
shows α (*T* ! (*SA* ! *i*)) $< b$
⟨*proof*⟩

lemma *gr-bucket-gr-hd*:
assumes *l-unchanged-inv* α *T SA0 SA*
and *l-locations-inv* α *T B SA*
and *l-bucket-ptr-inv* α *T B SA*
and *l-unknowns-inv* α *T B SA*
and *strict-mono* α
and *length SA* = *length T*
and *length SA* = *length SA0*
and *lms-init* α *T SA0*
and *s-init* α *T SA0*
and $i <$ *length SA*
and *SA* ! $i <$ *length T*
and *bucket-end* α *T b* $\leq i$
shows $b < \alpha$ (*T* ! (*SA* ! *i*))
⟨*proof*⟩

68.5 Seen

We have two helper lemmas in the case of updating the suffix array SA, and in the case when the current index is incremented. The two lemmas are used in conjunction in the case that the SA is updated and the current index is incremented.

lemma *l-seen-inv-upd*:
assumes *l-seen-inv T SA n* $n \leq k$ *SA* ! k = *length T*

shows *l-seen-inv* T $(SA[k := x])$ n
⟨*proof*⟩

lemma *l-seen-inv-Suc*:
assumes *l-seen-inv* T SA n SA ! n = *Suc* j k < *length* SA SA ! k = j
shows *l-seen-inv* T SA (*Suc* n)
⟨*proof*⟩

69 Distinctness

lemma *distinct-app3*:
distinct (*xs* @ *ys* @ *zs*) ⟷
distinct xs ∧ *distinct ys* ∧ *distinct zs* ∧
set xs ∩ *set ys* = {} ∧ *set xs* ∩ *set zs* = {} ∧ *set ys* ∩ *set zs* = {}
⟨*proof*⟩

69.1 Establishment

lemma *abs-is-lms-imp-in-lms-bucket*:
abs-is-lms T i ⟹ i ∈ *lms-bucket* α T (α (T ! i))
⟨*proof*⟩

lemma *l-distinct-inv-established*:
assumes *lms-init* α T SA
and *l-init* α T SA
and *s-init* α T SA
and *length* SA = *length* T
and *strict-mono* α
and *l-bucket-init* α T B
shows *l-distinct-inv* T SA
⟨*proof*⟩

corollary *l-distinct-inv-perm-established*:
assumes *l-perm-pre* α T B SA
shows *l-distinct-inv* T SA
⟨*proof*⟩

69.2 Maintenance

lemma *l-distinct-inv-maintained*:
assumes i < *length* SA
and SA ! i = *Suc* j
and *Suc* j < *length* T
and *suffix-type* T j = *L-type*
and *l-distinct-inv* T SA
and *l-pred-inv* T SA i
shows *l-distinct-inv* T $(SA[l := j])$
⟨*proof*⟩

corollary *l-distinct-inv-perm-maintained*:
assumes *l-perm-inv* α *T B SA0 SA i*
and $i <$ *length SA*
and *SA* ! $i =$ *Suc j*
and *Suc j* $<$ *length T*
and *suffix-type T j* $=$ *L-type*
shows *l-distinct-inv T* (*SA*[$l := j$])
⟨*proof*⟩

70 Unknowns

70.1 Establishment

lemma *l-unknowns-inv-established*:
assumes *l-init* α *T SA*
l-bucket-init α *T B*
length SA $=$ *length T*
shows *l-unknowns-inv* α *T B SA*
⟨*proof*⟩

corollary *l-unknowns-inv-perm-established*:
assumes *l-perm-pre* α *T B SA*
shows *l-unknowns-inv* α *T B SA*
⟨*proof*⟩

70.2 Maintenance

lemma *l-unknowns-inv-maintained*:
assumes *l-unknowns-inv* α *T B SA*
and *length B* $> \alpha$ (*Max* (*set T*))
and $i <$ *length SA*
and *SA* ! $i =$ *Suc j*
and *Suc j* $<$ *length T*
and *suffix-type T j* $=$ *L-type*
and $k = \alpha$ (*T* ! *j*)
and $l = B$! k
and *strict-mono* α
and *l-distinct-inv T SA*
and *l-pred-inv T SA i*
and *l-bucket-ptr-inv* α *T B SA*
shows *l-unknowns-inv* α *T* (*B*[$k :=$ *Suc* (*B* ! *k*)]) (*SA*[$l := j$])
⟨*proof*⟩

corollary *l-unknowns-inv-perm-maintained*:
assumes *l-perm-inv* α *T B SA0 SA i*
and $i <$ *length SA*
and *SA* ! $i =$ *Suc j*
and *Suc j* $<$ *length T*
and *suffix-type T j* $=$ *L-type*

and $k = \alpha\ (T\ !\ j)$
and $l = B\ !\ k$
shows *l-unknowns-inv* α T $(B[k := Suc\ (B\ !\ k)])$ $(SA[l := j])$
⟨*proof*⟩

71 Number of L-types

71.1 Establishment

We first prove that this invariant is established from the precondition, i.e., that initially, there are only LMS-types, which are just a special type of S-types, and that the initial pointer is the start of the bucket.

lemma *l-bucket-ptr-inv-established*:
assumes *lms-init* α T SA
and *l-init* α T SA
and *s-init* α T SA
and *length* SA = *length* T
and *strict-mono* α
and *l-bucket-init* α T B
shows *l-bucket-ptr-inv* α T B SA
⟨*proof*⟩

corollary *l-bucket-ptr-inv-perm-established*:
assumes *l-perm-pre* α T B SA
shows *l-bucket-ptr-inv* α T B SA
⟨*proof*⟩

71.2 Maintenance

We now prove that the invariant is maintained.

lemma *set-update-mem-neqI*:
$[\![x \in set\ xs;\ xs\ !\ i \neq x]\!] \Longrightarrow x \in set\ (xs[i := y])$
⟨*proof*⟩

lemma *cur-l-types-update-1*:
$[\![SA\ !\ l = length\ T;\ l < length\ SA;\ j \notin set\ SA;\ suffix\text{-}type\ T\ j = L\text{-}type;\ j < length\ T;$
$\alpha\ (T\ !\ j) = b]\!] \Longrightarrow$
cur-l-types α T $(SA[l := j])$ b = *insert* j (*cur-l-types* α T SA b)
⟨*proof*⟩

lemma *cur-l-types-update-2*:
assumes $SA\ !\ l = length\ T$ $\alpha\ (T\ !\ j) \neq b$
shows *cur-l-types* α T $(SA[l := j])$ b = *cur-l-types* α T SA b
⟨*proof*⟩

lemma *num-l-types-update-1*:

$[\![SA\ !\ l = \textit{length}\ T;\ l < \textit{length}\ SA;\ j \notin \textit{set}\ SA;\ \textit{suffix-type}\ T\ j = \textit{L-type};\ j < \textit{length}\ T;$
$\alpha\ (T\ !\ j) = b]\!] \Longrightarrow$
$\textit{num-l-types}\ \alpha\ T\ (SA[l := j])\ b = \textit{Suc}\ (\textit{num-l-types}\ \alpha\ T\ SA\ b)$
⟨*proof*⟩

lemma *num-l-types-update-2*:
$[\![SA\ !\ l = \textit{length}\ T;\ \alpha\ (T\ !\ j) \neq b]\!] \Longrightarrow$
$\textit{num-l-types}\ \alpha\ T\ (SA[l := j])\ b = \textit{num-l-types}\ \alpha\ T\ SA\ b$
⟨*proof*⟩

lemma *l-bucket-ptr-inv-maintained*:
assumes *l-bucket-ptr-inv* α *T B SA*
and *length SA* = *length T*
and *length B* > α (*Max* (*set T*))
and *i* < *length SA*
and *SA* ! *i* = *Suc j*
and *Suc j* < *length T*
and *suffix-type T j* = *L-type*
and *k* = α (*T* ! *j*)
and *l* = *B* ! *k*
and *strict-mono* α
and *l-distinct-inv T SA*
and *l-pred-inv T SA i*
and *l-unknowns-inv* α *T B SA*
shows *l-bucket-ptr-inv* α *T* (*B*[*k* := *Suc* (*B* ! *k*)]) (*SA*[*l* := *j*])
⟨*proof*⟩

corollary *l-bucket-ptr-inv-perm-maintained*:
assumes *l-perm-inv* α *T B SA0 SA i*
and *i* < *length SA*
and *SA* ! *i* = *Suc j*
and *Suc j* < *length T*
and *suffix-type T j* = *L-type*
and *k* = α (*T* ! *j*)
and *l* = *B* ! *k*
shows *l-bucket-ptr-inv* α *T* (*B*[*k* := *Suc* (*B* ! *k*)]) (*SA*[*l* := *j*])
⟨*proof*⟩

72 L Locations

72.1 Establishment

lemma *l-locations-inv-established*:
assumes *l-bucket-init* α *T B*
shows *l-locations-inv* α *T B SA*
⟨*proof*⟩

corollary *l-locations-inv-perm-established*:

assumes *l-perm-pre* α *T B SA*
shows *l-locations-inv* α *T B SA*
⟨*proof*⟩

72.2 Maintenance

lemma *l-locations-inv-maintained*:
assumes *l-locations-inv* α *T B SA*
and *length B* $>$ α (*Max* (*set T*))
and $i <$ *length SA*
and *SA* ! i = *Suc j*
and *Suc j* $<$ *length T*
and *suffix-type T j* = *L-type*
and $k = \alpha$ (T ! j)
and $l = B$! k
and *strict-mono* α
and *l-distinct-inv T SA*
and *l-pred-inv T SA i*
and *l-bucket-ptr-inv* α *T B SA*
shows *l-locations-inv* α *T* ($B[k$:= *Suc* (B ! k)]) (*SA*[l := j])
⟨*proof*⟩

corollary *l-locations-inv-perm-maintained*:
assumes *l-perm-inv* α *T B SA0 SA i*
and $i <$ *length SA*
and *SA* ! i = *Suc j*
and *Suc j* $<$ *length T*
and *suffix-type T j* = *L-type*
and $k = \alpha$ (T ! j)
and $l = B$! k
shows *l-locations-inv* α *T* ($B[k$:= *Suc* (B ! k)]) (*SA*[l := j])
⟨*proof*⟩

73 Unchanged

73.1 Establishment

lemma *l-unchanged-inv-established*:
l-unchanged-inv α *T SA SA*
⟨*proof*⟩

73.2 Maintenance

lemma *l-unchanged-inv-maintained*:
assumes *l-unchanged-inv* α *T SA0 SA*
and *length B* $>$ α (*Max* (*set T*))
and $i <$ *length SA*
and *SA* ! i = *Suc j*
and *Suc j* $<$ *length T*

and *suffix-type* T j = *L-type*
and $k = \alpha\ (T\ !\ j)$
and $l = B\ !\ k$
and *strict-mono* α
and *l-distinct-inv* T *SA*
and *l-pred-inv* T *SA* i
and *l-bucket-ptr-inv* α T B *SA*
shows *l-unchanged-inv* α T *SA0* (*SA*[l := j])
⟨*proof*⟩

corollary *l-unchanged-inv-perm-maintained*:
assumes *l-perm-inv* α T B *SA0* *SA* i
and i < *length* *SA*
and *SA* ! i = *Suc* j
and *Suc* j < *length* T
and *suffix-type* T j = *L-type*
and $k = \alpha\ (T\ !\ j)$
and $l = B\ !\ k$
shows *l-unchanged-inv* α T *SA0* (*SA*[l := j])
⟨*proof*⟩

74 Invariant about the Current Index

74.1 Establishment

The first invariant is that current index is always less than the index where the update will occur.

lemma *l-index-inv-established*:
assumes *lms-init* α T *SA*
and *l-init* α T *SA*
and *s-init* α T *SA*
and *length* *SA* = *length* T
and *strict-mono* α
and *l-bucket-init* α T B
shows *l-index-inv* α T B *SA*
⟨*proof*⟩

corollary *l-index-inv-perm-established*:
assumes *l-perm-pre* α T B *SA*
shows *l-index-inv* α T B *SA*
⟨*proof*⟩

74.2 Maintenance

lemma *l-index-inv-maintained*:
assumes *l-index-inv* α T B *SA*
and *length* $B > \alpha$ (*Max* (*set* T))
and i < *length* *SA*
and *SA* ! i = *Suc* j

and *Suc j < length T*
and *suffix-type T j = L-type*
and $k = \alpha\ (T\ !\ j)$
and $l = B\ !\ k$
and *strict-mono* α
and *l-distinct-inv T SA*
and *l-pred-inv T SA i*
and *l-bucket-ptr-inv* α *T B SA*
and *l-unknowns-inv* α *T B SA*
shows *l-index-inv* α *T* $(B[k := Suc\ (B\ !\ k)])\ (SA[l := j])$
⟨*proof*⟩

corollary *l-index-inv-perm-maintained*:
assumes *l-perm-inv* α *T B SA0 SA i*
and *i < length SA*
and *SA ! i = Suc j*
and *Suc j < length T*
and *suffix-type T j = L-type*
and $k = \alpha\ (T\ !\ j)$
and $l = B\ !\ k$
shows *l-index-inv* α *T* $(B[k := Suc\ (B\ !\ k)])\ (SA[l := j])$
⟨*proof*⟩

75 Predecessor Invariant

75.1 Establishment

The proof for the establishment is simple because initially, SA contains no L-types.

lemma *l-pred-inv-established*:
assumes *lms-init* α *T SA*
and *l-init* α *T SA*
and *s-init* α *T SA*
and *length SA = length T*
and *strict-mono* α
shows *l-pred-inv T SA 0*
⟨*proof*⟩

corollary *l-pred-inv-perm-established*:
assumes *l-perm-pre* α *T B SA*
shows *l-pred-inv T SA 0*
⟨*proof*⟩

75.2 Maintenance

In this section, we prove that the predecessor invariant *l-pred-inv ?T ?SA ?k* = (∀ *i<length ?SA*. *?SA ! i < length ?T* ∧ *suffix-type ?T (?SA ! i) = L-type* ⟶ (∃ *j<length ?SA*. *?SA ! j = Suc (?SA ! i)* ∧ *j < i* ∧ *j < ?k*))is maintained. In English, this invariant states that for all L-type suffixes in the suffix array (SA), their right neighbour is in SA and occurs before them.

We now prove that the invariant is maintained for each branch of the *abs-induce-l-step*

lemma *l-pred-inv-maintained-no-update*:
assumes *l-pred-inv T SA i*
shows *l-pred-inv T SA* (*Suc i*)
⟨*proof*⟩

lemma *l-pred-inv-maintained*:
assumes *l-pred-inv T SA i*
and $i <$ *length SA*
and *SA* ! $i =$ *Suc j*
and *Suc j* $<$ *length T*
and *suffix-type T j* = *L-type*
and $k = \alpha$ (*T* ! *j*)
and $l = B$! *k*
and *strict-mono* α
and *l-distinct-inv T SA*
and *l-bucket-ptr-inv* α *T B SA*
and *l-unknowns-inv* α *T B SA*
and *l-index-inv* α *T B SA*
shows *l-pred-inv T* (*SA*[$l := j$]) (*Suc i*)
⟨*proof*⟩

corollary *l-pred-inv-perm-maintained*:
assumes *l-perm-inv* α *T B SA0 SA i*
and $i <$ *length SA*
and *SA* ! $i =$ *Suc j*
and *Suc j* $<$ *length T*
and *suffix-type T j* = *L-type*
and $k = \alpha$ (*T* ! *j*)
and $l = B$! *k*
shows *l-pred-inv T* (*SA*[$l := j$]) (*Suc i*)
⟨*proof*⟩

76 Seen Invariant

76.1 Establishment

We first show that the invariant is initially true, i.e. *l-seen-inv T SA 0*.

lemma *l-seen-inv-established*:
l-seen-inv T SA 0
⟨*proof*⟩

76.2 Maintenance

We now show that the invariant is maintained after each call of *abs-induce-l-step*.

lemma *l-seen-inv-maintained-no-update*:
⟦*l-seen-inv T SA i*; *length T* ≤ *SA* ! *i*⟧ ⟹ *l-seen-inv T SA* (*Suc i*)
⟦*l-seen-inv T SA i*; *length SA* ≤ *i*⟧ ⟹ *l-seen-inv T SA* (*Suc i*)
⟦*l-seen-inv T SA i*; *SA* ! *i* < *length T*; *SA* ! *i* = *0*⟧ ⟹ *l-seen-inv T SA* (*Suc i*)
⟦*l-seen-inv T SA i*; *SA* ! *i* < *length T*; *SA* ! *i* = *Suc j*; *suffix-type T j* = *S-type*⟧
⟹
l-seen-inv T SA (*Suc i*)
⟨*proof*⟩

lemma *l-seen-inv-maintained*:
assumes *l-seen-inv T SA i*
and *i* < *length SA*
and *SA* ! *i* = *Suc j*
and *Suc j* < *length T*
and *suffix-type T j* = *L-type*
and *k* = *α* (*T* ! *j*)
and *l* = *B* ! *k*
and *length SA* = *length T*
and *strict-mono α*
and *l-distinct-inv T SA*
and *l-pred-inv T SA i*
and *l-unknowns-inv α T B SA*
and *l-bucket-ptr-inv α T B SA*
and *l-index-inv α T B SA*
shows *l-seen-inv T* (*SA*[*l* := *j*]) (*Suc i*)
⟨*proof*⟩

corollary *l-seen-inv-perm-maintained*:
assumes *l-perm-inv α T B SA0 SA i*
and *i* < *length SA*
and *SA* ! *i* = *Suc j*
and *Suc j* < *length T*
and *suffix-type T j* = *L-type*
and *k* = *α* (*T* ! *j*)
and *l* = *B* ! *k*
shows *l-seen-inv T* (*SA*[*l* := *j*]) (*Suc i*)
⟨*proof*⟩

77 Permutation

77.1 Establishment

lemma *l-perm-inv-established*:
assumes *l-perm-pre α T B SA*
shows *l-perm-inv α T B SA SA 0*
⟨*proof*⟩

77.2 Maintenance

lemma *l-perm-inv-maintained*:
assumes *l-perm-inv* α *T B SA0 SA i*
and $i <$ *length SA*
and *SA* ! $i =$ *Suc j*
and *Suc j* $<$ *length T*
and *suffix-type T j* $=$ *L-type*
and $k = \alpha$ (*T* ! *j*)
and $l = B$! k
shows *l-perm-inv* α *T* (*B*[*k* := *Suc* (*B* ! *k*)]) *SA0* (*SA*[*l* := *j*]) (*Suc i*)
⟨*proof*⟩

lemma *l-perm-inv-maintained-no-upd-1*:
assumes *l-perm-inv* α *T B SA0 SA i*
and *length SA* $\leq i$
shows *l-perm-inv* α *T B SA0 SA* (*Suc i*)
⟨*proof*⟩

lemma *l-perm-inv-maintained-no-upd-2*:
assumes *l-perm-inv* α *T B SA0 SA i*
and *length T* $\leq$ *SA* ! *i*
shows *l-perm-inv* α *T B SA0 SA* (*Suc i*)
⟨*proof*⟩

lemma *l-perm-inv-maintained-no-upd-3*:
assumes *l-perm-inv* α *T B SA0 SA i*
and *SA* ! $i <$ *length T*
and *SA* ! $i = 0$
shows *l-perm-inv* α *T B SA0 SA* (*Suc i*)
⟨*proof*⟩

lemma *l-perm-inv-maintained-no-upd-4*:
assumes *l-perm-inv* α *T B SA0 SA i*
and *SA* ! $i <$ *length T*
and *SA* ! $i =$ *Suc j*
and *suffix-type T j* $=$ *S-type*
shows *l-perm-inv* α *T B SA0 SA* (*Suc i*)
⟨*proof*⟩

lemmas *l-perm-inv-maintained-no-update* =
l-perm-inv-maintained-no-upd-1 l-perm-inv-maintained-no-upd-2 l-perm-inv-maintained-no-upd-3 l-perm-inv-maintained-no-upd-4

lemma *abs-induce-l-perm-step*:
assumes *l-perm-inv* α *T B SA0 SA i*
and *abs-induce-l-step* (*B*, *SA*, *i*) (α, *T*) = (*B'*, *SA'*, *i'*)
shows *l-perm-inv* α *T B' SA0 SA' i'*
⟨*proof*⟩

lemma *abs-induce-l-base-perm-inv-maintained*:
assumes *l-perm-inv* α *T B SA0 SA 0*
and *abs-induce-l-base* α *T B SA* = (B', SA', *i*)
shows *l-perm-inv* α *T* B' *SA0* SA' *i*
⟨*proof*⟩

78 Sorted

lemma *l-suffix-sorted-inv-established*:
assumes *l-bucket-init* α *T B*
shows *l-suffix-sorted-inv* α *T B SA*
⟨*proof*⟩

lemma *l-prefix-sorted-inv-established*:
assumes *l-bucket-init* α *T B*
shows *l-prefix-sorted-inv* α *T B SA*
⟨*proof*⟩

lemma *l-sorted-inv-maintained-step*:
assumes *l-perm-inv* α *T B SA0 SA i*
and $i <$ *length SA*
and *SA* ! *i* = *Suc j*
and *Suc j* < *length T*
and *suffix-type T j* = *L-type*
and $k = \alpha$ (*T* ! *j*)
and *l* = *B* ! *k*
and $b \leq \alpha$ (*Max* (*set T*))
and $b \neq k$
and *ordlistns.sorted* (*map f* (*list-slice SA* (*bucket-start* α *T b*) (*B* ! *b*)))
shows *ordlistns.sorted* (*map f* (*list-slice* (*SA*[*l* := *j*]) (*bucket-start* α *T b*) (*B*[*k* := *Suc l*] ! *b*)))
⟨*proof*⟩

lemma *l-suffix-sorted-inv-maintained-step*:
assumes *l-perm-inv* α *T B SA0 SA i*
and *l-suffix-sorted-pre* α *T SA0*
and *l-suffix-sorted-inv* α *T B SA*
and $i <$ *length SA*
and *SA* ! *i* = *Suc j*
and *Suc j* < *length T*
and *suffix-type T j* = *L-type*
and $k = \alpha$ (*T* ! *j*)
and *l* = *B* ! *k*
shows *l-suffix-sorted-inv* α *T* (*B*[*k* := *Suc l*]) (*SA*[*l* := *j*])
⟨*proof*⟩

lemma *l-prefix-sorted-inv-maintained-step*:
assumes *l-perm-inv* α *T B SA0 SA i*

and *l-prefix-sorted-pre* α *T SA0*
and *l-prefix-sorted-inv* α *T B SA*
and $i <$ *length SA*
and *SA* ! i = *Suc j*
and *Suc j* < *length T*
and *suffix-type T j* = *L-type*
and $k = \alpha$ (*T* ! *j*)
and $l = B$! k
shows *l-prefix-sorted-inv* α *T* ($B[k := Suc\ l]$) ($SA[l := j]$)
⟨*proof*⟩

lemma *abs-induce-l-suffix-sorted-step*:
assumes *l-perm-inv* α *T B SA0 SA i*
and *l-suffix-sorted-pre* α *T SA0*
and *l-suffix-sorted-inv* α *T B SA*
and *abs-induce-l-step* (*B*, *SA*, *i*) (α, *T*) = (B', SA', i')
shows *l-suffix-sorted-inv* α *T* B' SA'
⟨*proof*⟩

lemma *abs-induce-l-prefix-sorted-step*:
assumes *l-perm-inv* α *T B SA0 SA i*
and *l-prefix-sorted-pre* α *T SA0*
and *l-prefix-sorted-inv* α *T B SA*
and *abs-induce-l-step* (*B*, *SA*, *i*) (α, *T*) = (B', SA', i')
shows *l-prefix-sorted-inv* α *T* B' SA'
⟨*proof*⟩

lemma *abs-induce-l-base-suffix-sorted-inv-maintained*:
assumes *l-perm-inv* α *T B SA0 SA 0*
and *l-suffix-sorted-pre* α *T SA0*
and *l-suffix-sorted-inv* α *T B SA*
and *abs-induce-l-base* α *T B SA* = (B', SA', *i*)
shows *l-suffix-sorted-inv* α *T* B' SA'
⟨*proof*⟩

lemma *abs-induce-l-base-prefix-sorted-inv-maintained*:
assumes *l-perm-inv* α *T B SA0 SA 0*
and *l-prefix-sorted-pre* α *T SA0*
and *l-prefix-sorted-inv* α *T B SA*
and *abs-induce-l-base* α *T B SA* = (B', SA', *i*)
shows *l-prefix-sorted-inv* α *T* B' SA'
⟨*proof*⟩

79 L-type Exhaustiveness

The *abs-induce-l* function is exhaustive if it has inserted all the L-types

definition *l-type-exhaustive* :: (*'a* :: {*linorder*, *order-bot*}) *list* ⇒ *nat list* ⇒ *bool*
where

$l\text{-}type\text{-}exhaustive\ T\ SA = (\forall\, i < length\ T.\ suffix\text{-}type\ T\ i = L\text{-}type \longrightarrow i \in set\ SA)$

There two cases when the *abs-induce-l* function is not exhaustive: when there is an L-type that is not in SA but its successor (right neighbour) is in SA, and the other is when there is an L-type that is not in SA and its successor is also not in SA. We will show that both cases will be False.

lemma *not-l-type-exhaustive-imp-ex*:
$\neg l\text{-}type\text{-}exhaustive\ T\ SA \Longrightarrow$
$(\exists\, i < length\ T.\ suffix\text{-}type\ T\ i = L\text{-}type \wedge i \notin set\ SA \wedge Suc\ i \in set\ SA) \vee$
$((\exists\, i < length\ T.\ suffix\text{-}type\ T\ i = L\text{-}type \wedge i \notin set\ SA) \wedge$
$\neg(\exists\, i.\ i < length\ T \wedge suffix\text{-}type\ T\ i = L\text{-}type \wedge i \notin set\ SA \wedge Suc\ i \in set\ SA))$
⟨*proof*⟩

lemma *l-type-exhaustive-imp-l-bucket*:
$[\![strict\text{-}mono\ \alpha;\ l\text{-}type\text{-}exhaustive\ T\ SA;\ b \le \alpha\ (Max\ (set\ T))]\!] \Longrightarrow$
$\{i.\ i \in set\ SA \wedge i \in l\text{-}bucket\ \alpha\ T\ b\} = l\text{-}bucket\ \alpha\ T\ b$
⟨*proof*⟩

lemma *l-type-exhaustive-imp-all-l-types*:
$l\text{-}type\text{-}exhaustive\ T\ SA \Longrightarrow$
$\{i.\ i \in set\ SA \wedge i \in l\text{-}bucket\ \alpha\ T\ (\alpha\ (T\ !\ i))\} = \{i.\ i < length\ T \wedge suffix\text{-}type\ T\ i = L\text{-}type\}$
⟨*proof*⟩

79.1 Case 1

In the case 1, we have that $\exists\, k{<}length\ T.\ suffix\text{-}type\ T\ k = L\text{-}type \wedge k \notin set\ SA \wedge Suc\ k \in set\ SA$. From this, we know that $\exists\, j{<}length\ SA.\ SA\ !\ j = Suc\ k$

lemma
$Suc\ k \in set\ SA \Longrightarrow \exists\, j < length\ SA.\ SA\ !\ j = Suc\ k$
⟨*proof*⟩

After executing the *abs-induce-l* function, we know that we have seen

79.2 Case 2

In the case 2, we have that $\exists\, k{<}length\ T.\ suffix\text{-}type\ T\ k = L\text{-}type \wedge k \notin set\ SA \wedge Suc\ k \notin set\ SA$.

lemma *finite-and-Suc-imp-False*:
assumes *finite-A*: $finite\ A$
and *not-empty*: $A \ne \{\}$
and *Suc-A*: $\forall\, a \in A.\ Suc\ a \in A$
shows *False*
⟨*proof*⟩

lemma *not-exhaustive-neighbour-is-l-type*:

assumes *A*: $A = \{k \mid k.\ \textit{suffix-type}\ T\ k = \textit{L-type} \wedge k \notin B \wedge \textit{Suc}\ k \notin B \wedge k < \textit{length}\ T\}$
and *subset-B*: $\{k \mid k.\ \textit{abs-is-lms}\ T\ k\} \subseteq B$
and $k \in A$
shows $\textit{suffix-type}\ T\ (\textit{Suc}\ k) = \textit{L-type}$
⟨*proof*⟩

lemma *no-exhausted-neighbour*:
assumes *A*: $A = \{k \mid k.\ \textit{suffix-type}\ T\ k = \textit{L-type} \wedge k \notin B \wedge \textit{Suc}\ k \notin B \wedge k < \textit{length}\ T\}$
and *B*: $\{k \mid k.\ \textit{abs-is-lms}\ T\ k\} \subseteq B$
and *C*: $\neg(\exists k.\ k < \textit{length}\ T \wedge \textit{suffix-type}\ T\ k = \textit{L-type} \wedge k \notin B \wedge \textit{Suc}\ k \in B)$
and *D*: $\textit{suffix-type}\ T\ i = \textit{L-type}$
and *E*: $i \notin B$
and *F*: $i < \textit{length}\ T$
shows $i \in A$
⟨*proof*⟩

lemma *l-type-less-length-imp-neightbour-less-length*:
$[\![\textit{suffix-type}\ T\ i = \textit{L-type};\ i < \textit{length}\ T]\!] \Longrightarrow \textit{Suc}\ i < \textit{length}\ T$
⟨*proof*⟩

lemma *no-exhausted-neighbour-imp-False*:
assumes *A*: $A = \{k \mid k.\ \textit{suffix-type}\ T\ k = \textit{L-type} \wedge k \notin B \wedge \textit{Suc}\ k \notin B \wedge k < \textit{length}\ T\}$
and *B*: $\{k \mid k.\ \textit{abs-is-lms}\ T\ k\} \subseteq B$
and *C*: $\neg(\exists k.\ k < \textit{length}\ T \wedge \textit{suffix-type}\ T\ k = \textit{L-type} \wedge k \notin B \wedge \textit{Suc}\ k \in B)$
and *nempty*: $A \neq \{\}$
shows *False*
⟨*proof*⟩

79.3 Exhaustiveness Proof

lemma *abs-induce-l-exhaustive*:
assumes *l-seen-inv T SA* (*length SA*)
and *lms-init* α *T SA0*
and *length SA* = *length SA0*
and *length SA* = *length T*
and *strict-mono* α
and *l-unchanged-inv* α *T SA0 SA*
shows *l-type-exhaustive T SA*
⟨*proof*⟩

80 Correctness and Exhaustiveness

lemma *abs-induce-l-perm-inv-imp-exhaustiveness*:
assumes *abs-induce-l-base* α *T B SA* = (B', SA', *i*)
and *l-perm-inv* α *T* B' *SA* SA' *i*
shows *l-type-exhaustive T* SA'

⟨*proof*⟩

lemma *abs-induce-l-perm-inv-B-val*:
assumes *abs-induce-l-base* α *T B SA* = (*B′*, *SA′*, *i*)
and *l-perm-inv* α *T B′ SA SA′ i*
and $b \leq \alpha$ (*Max* (*set T*))
shows *B′* ! *b* = *l-bucket-end* α *T b*
⟨*proof*⟩

theorem *abs-induce-l-distinct-l-bucket*:
assumes *l-perm-pre* α *T B SA*
and $b \leq \alpha$ (*Max* (*set T*))
shows *distinct* (*list-slice* (*abs-induce-l* α *T B SA*) (*bucket-start* α *T b*) (*l-bucket-end* α *T b*))
⟨*proof*⟩

theorem *abs-induce-l-list-slice-l-bucket*:
assumes *l-perm-pre* α *T B SA*
and $b \leq \alpha$ (*Max* (*set T*))
shows *set* (*list-slice* (*abs-induce-l* α *T B SA*) (*bucket-start* α *T b*) (*l-bucket-end* α *T b*)) = *l-bucket* α *T b*
(**is** *set ?xs* = *l-bucket* α *T b*)
⟨*proof*⟩

lemma *abs-induce-l-unchanged*:
assumes *l-perm-pre* α *T B SA*
and $b \leq \alpha$ (*Max* (*set T*))
and *s-bucket-start* α *T b* $\leq i$
and $i <$ *bucket-end* α *T b*
shows (*abs-induce-l* α *T B SA*) ! *i* = *SA* ! *i*
⟨*proof*⟩
theorem *abs-induce-l-suffix-sorted-l-bucket*:
assumes *l-perm-pre* α *T B SA*
and *l-suffix-sorted-pre* α *T SA*
and $b \leq \alpha$ (*Max* (*set T*))
shows *ordlistns.sorted* (*map* (*suffix T*)
(*list-slice* (*abs-induce-l* α *T B SA*) (*bucket-start* α *T b*) (*l-bucket-end* α *T b*)))
⟨*proof*⟩
theorem *abs-induce-l-prefix-sorted-l-bucket*:
assumes *l-perm-pre* α *T B SA*
and *l-prefix-sorted-pre* α *T SA*
and $b \leq \alpha$ (*Max* (*set T*))
shows *ordlistns.sorted* (*map* (*lms-prefix T*)
(*list-slice* (*abs-induce-l* α *T B SA*) (*bucket-start* α *T b*) (*l-bucket-end* α *T b*)))
⟨*proof*⟩

end

theory *Abs-Induce-S-Verification*
imports *../abs−def/Abs-SAIS*
begin

81 Abstract Induce S Simple Properties

lemma *abs-induce-s-step-ex*:
$\exists B'\ SA'\ i'.\ abs\text{-}induce\text{-}s\text{-}step\ a\ b = (B', SA', i')$
⟨*proof*⟩

lemma *abs-induce-s-step-B-length*:
$abs\text{-}induce\text{-}s\text{-}step\ (B, SA, i)\ (\alpha, T) = (B', SA', i') \Longrightarrow length\ B' = length\ B$
⟨*proof*⟩

lemma *abs-induce-s-step-SA-length*:
$abs\text{-}induce\text{-}s\text{-}step\ (B, SA, i)\ (\alpha, T) = (B', SA', i') \Longrightarrow length\ SA' = length\ SA$
⟨*proof*⟩

lemma *abs-induce-s-step-Suc*:
$abs\text{-}induce\text{-}s\text{-}step\ (B, SA, Suc\ i)\ (\alpha, T) = (B', SA', i') \Longrightarrow i' = i$
⟨*proof*⟩

lemma *abs-induce-s-step-0*:
$abs\text{-}induce\text{-}s\text{-}step\ (B, SA, 0)\ (\alpha, T) = (B, SA, 0)$
⟨*proof*⟩

corollary *abs-induce-s-step-0-alt*:
assumes $abs\text{-}induce\text{-}s\text{-}step\ (B, SA, i)\ (\alpha, T) = (B', SA', i')$
and $i = 0$
shows $B = B' \wedge SA = SA' \wedge i' = 0$
⟨*proof*⟩

lemma *repeat-abs-induce-s-step-index*:
$\exists B'\ SA'.\ repeat\ n\ abs\text{-}induce\text{-}s\text{-}step\ (B, SA, m)\ (\alpha, T) = (B', SA', m - n) \wedge$
$length\ SA' = length\ SA \wedge length\ B' = length\ B$
⟨*proof*⟩

lemma *abs-induce-s-base-index*:
$\exists B'\ SA'.\ abs\text{-}induce\text{-}s\text{-}base\ \alpha\ T\ B\ SA = (B', SA', 0)$
⟨*proof*⟩

lemma *abs-induce-s-length*:
$length\ (abs\text{-}induce\text{-}s\ \alpha\ T\ B\ SA) = length\ SA$
⟨*proof*⟩

82 Preconditions

definition *l-types-init*

where
l-types-init α *T SA* $\equiv$
($\forall$ *b* $\leq$ α (*Max* (*set T*)).
set (*list-slice SA* (*bucket-start* α *T b*) (*l-bucket-end* α *T b*)) = *l-bucket* α *T b* $\wedge$
distinct (*list-slice SA* (*bucket-start* α *T b*) (*l-bucket-end* α *T b*))
)

lemma *l-types-initD*:
⟦*l-types-init* α *T SA*; *b* $\leq$ α (*Max* (*set T*))⟧ $\Longrightarrow$
set (*list-slice SA* (*bucket-start* α *T b*) (*l-bucket-end* α *T b*)) = *l-bucket* α *T b*
⟦*l-types-init* α *T SA*; *b* $\leq$ α (*Max* (*set T*))⟧ $\Longrightarrow$
distinct (*list-slice SA* (*bucket-start* α *T b*) (*l-bucket-end* α *T b*))
⟨*proof*⟩

lemma *l-types-init-nth*:
assumes *length SA* = *length T*
and *l-types-init* α *T SA*
and *b* $\leq$ α (*Max* (*set T*))
and *bucket-start* α *T b* $\leq$ *i*
and *i* < *l-bucket-end* α *T b*
shows *SA* ! *i* $\in$ *l-bucket* α *T b*
⟨*proof*⟩

definition *s-type-init*
where
s-type-init T SA $\equiv$ ($\exists$ *n*. *length T* = *Suc n* $\wedge$ *SA* ! *0* = *n*)

definition *s-perm-pre*
where
s-perm-pre α *T B SA n* $\equiv$
s-bucket-init α *T B* $\wedge$
s-type-init T SA $\wedge$
strict-mono α $\wedge$
α (*Max* (*set T*)) < *length B* $\wedge$
length SA = *length T* $\wedge$
l-types-init α *T SA* $\wedge$
valid-list T $\wedge$
α *bot* = *0* $\wedge$
Suc 0 < *length T* $\wedge$
length T $\leq$ *n*

definition *s-sorted-pre*
where
s-sorted-pre α *T SA* $\equiv$
($\forall$ *b* $\leq$ α (*Max* (*set T*)).
ordlistns.sorted (*map* (*suffix T*) (*list-slice SA* (*bucket-start* α *T b*) (*l-bucket-end* α *T b*)))
)

lemma *s-sorted-preD*:
⟦*s-sorted-pre* α *T SA*; $b \leq \alpha$ (*Max* (*set T*))⟧ $\Longrightarrow$
ordlistns.sorted (*map* (*suffix T*) (*list-slice SA* (*bucket-start* α *T b*) (*l-bucket-end* α *T b*)))
⟨*proof*⟩

definition *s-prefix-sorted-pre*
where
s-prefix-sorted-pre α *T SA* $\equiv$
($\forall\, b \leq \alpha$ (*Max* (*set T*)).
ordlistns.sorted (*map* (*lms-slice T*) (*list-slice SA* (*bucket-start* α *T b*) (*l-bucket-end* α *T b*)))
)

lemma *s-prefix-sorted-preD*:
⟦*s-prefix-sorted-pre* α *T SA*; $b \leq \alpha$ (*Max* (*set T*))⟧ $\Longrightarrow$
ordlistns.sorted (*map* (*lms-slice T*) (*list-slice SA* (*bucket-start* α *T b*) (*l-bucket-end* α *T b*)))
⟨*proof*⟩

83 Invariants

83.1 Definitions

83.1.1 Distinctness

definition *s-distinct-inv*
where
s-distinct-inv α *T B SA* $\equiv$
($\forall\, b \leq \alpha$ (*Max* (*set T*)). *distinct* (*list-slice SA* (*B* ! *b*) (*bucket-end* α *T b*)))

lemma *s-distinct-invD*:
⟦*s-distinct-inv* α *T B SA*; $b \leq \alpha$ (*Max* (*set T*))⟧ $\Longrightarrow$
distinct (*list-slice SA* (*B* ! *b*) (*bucket-end* α *T b*))
⟨*proof*⟩

83.1.2 S Bucket Ptr

definition *s-bucket-ptr-inv* ::
('*a* :: {*linorder*, *order-bot*} $\Rightarrow$ *nat*) $\Rightarrow$ '*a list* $\Rightarrow$ *nat list* $\Rightarrow$ *bool*
where
s-bucket-ptr-inv α *T B* $\equiv$
($\forall\, b \leq \alpha$ (*Max* (*set T*)).
s-bucket-start α *T b* $\leq$ *B* ! *b* $\wedge$
B ! *b* $\leq$ *bucket-end* α *T b* $\wedge$
($b = 0 \longrightarrow B \,!\, b = 0$))

lemma *s-bucket-ptr-lower-bound*:
assumes *s-bucket-ptr-inv* α *T B*
and $b \leq \alpha$ (*Max* (*set T*))

shows *s-bucket-start* α *T b* $\leq$ *B* ! *b*
⟨*proof*⟩

lemma *s-bucket-ptr-upper-bound*:
assumes *s-bucket-ptr-inv* α *T B*
and *b* $\leq$ α (*Max* (*set T*))
shows *B* ! *b* $\leq$ *bucket-end* α *T b*
⟨*proof*⟩

lemma *s-bucket-ptr-0*:
assumes *s-bucket-ptr-inv* α *T B*
and *b* = *0*
shows *B* ! *b* = *0*
⟨*proof*⟩

83.1.3 Locations

definition *s-locations-inv* ::
(*'a* :: {*linorder*, *order-bot*} ⇒ *nat*) ⇒ *'a list* ⇒ *nat list* ⇒ *nat list* ⇒ *bool*
where
s-locations-inv α *T B SA* ≡
(∀ *b* $\leq$ α (*Max* (*set T*)).
(∀ *i*. *B* ! *b* $\leq$ *i* ∧ *i* < *bucket-end* α *T b* ⟶ *SA* ! *i* ∈ *s-bucket* α *T b*))

lemma *s-locations-invD*:
⟦*s-locations-inv* α *T B SA*; *b* $\leq$ α (*Max* (*set T*)); *B* ! *b* $\leq$ *i*; *i* < *bucket-end* α *T b*⟧ ⟹
SA ! *i* ∈ *s-bucket* α *T b*
⟨*proof*⟩

lemma *s-locations-inv-in-list-slice*:
assumes *s-locations-inv* α *T B SA*
and *b* $\leq$ α (*Max* (*set T*))
and *x* ∈ *set* (*list-slice SA* (*B* ! *b*) (*bucket-end* α *T b*))
shows *x* ∈ *s-bucket* α *T b*
⟨*proof*⟩

lemma *s-locations-inv-subset-s-bucket*:
assumes *s-locations-inv* α *T B SA*
and *b* $\leq$ α (*Max* (*set T*))
shows *set* (*list-slice SA* (*B* ! *b*) (*bucket-end* α *T b*)) ⊆ *s-bucket* α *T b*
⟨*proof*⟩

83.1.4 Unchanged

definition *s-unchanged-inv* ::
(*'a* :: {*linorder*, *order-bot*} ⇒ *nat*) ⇒ *'a list* ⇒ *nat list* ⇒ *nat list* ⇒ *nat list* ⇒ *bool*
where
s-unchanged-inv α *T B SA SA'* ≡

$(\forall b \leq \alpha\ (Max\ (set\ T)).\ (\forall i.\ bucket\text{-}start\ \alpha\ T\ b \leq i \wedge i < B\ !\ b \longrightarrow\ SA'\ !\ i = SA\ !\ i))$

lemma *s-unchanged-invD*:
$[\![s\text{-}unchanged\text{-}inv\ \alpha\ T\ B\ SA\ SA';\ b \leq \alpha\ (Max\ (set\ T));\ bucket\text{-}start\ \alpha\ T\ b \leq i;\ i < B\ !\ b]\!] \Longrightarrow$
$SA'\ !\ i = SA\ !\ i$
⟨*proof*⟩

83.1.5 Seen

definition *s-seen-inv* ::
$('a :: \{linorder,\ order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow nat \Rightarrow bool$
where
$s\text{-}seen\text{-}inv\ \alpha\ T\ B\ SA\ n \equiv$
$\forall i < length\ SA.\ n \leq i \longrightarrow$
$(suffix\text{-}type\ T\ (SA\ !\ i) = S\text{-}type \longrightarrow in\text{-}s\text{-}current\text{-}bucket\ \alpha\ T\ B\ (\alpha\ (T\ !\ (SA\ !\ i)))\ i) \wedge$
$(suffix\text{-}type\ T\ (SA\ !\ i) = L\text{-}type \longrightarrow in\text{-}l\text{-}bucket\ \alpha\ T\ (\alpha\ (T\ !\ (SA\ !\ i)))\ i) \wedge$
$SA\ !\ i < length\ T$

lemma *s-seen-invD*:
$[\![s\text{-}seen\text{-}inv\ \alpha\ T\ B\ SA\ n;\ i < length\ SA;\ n \leq i]\!] \Longrightarrow SA\ !\ i < length\ T$
$[\![s\text{-}seen\text{-}inv\ \alpha\ T\ B\ SA\ n;\ i < length\ SA;\ n \leq i;\ suffix\text{-}type\ T\ (SA\ !\ i) = L\text{-}type]\!] \Longrightarrow$
$in\text{-}l\text{-}bucket\ \alpha\ T\ (\alpha\ (T\ !\ (SA\ !\ i)))\ i$
$[\![s\text{-}seen\text{-}inv\ \alpha\ T\ B\ SA\ n;\ i < length\ SA;\ n \leq i;\ suffix\text{-}type\ T\ (SA\ !\ i) = S\text{-}type]\!] \Longrightarrow$
$in\text{-}s\text{-}current\text{-}bucket\ \alpha\ T\ B\ (\alpha\ (T\ !\ (SA\ !\ i)))\ i$
⟨*proof*⟩

83.1.6 Predecessor

definition *s-pred-inv* ::
$(('a :: \{linorder,\ order\text{-}bot\}) \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow nat \Rightarrow bool$
where
$s\text{-}pred\text{-}inv\ \alpha\ T\ B\ SA\ n =$
$(\forall b\ i.\ in\text{-}s\text{-}current\text{-}bucket\ \alpha\ T\ B\ b\ i \wedge b \neq 0 \longrightarrow$
$(\exists j < length\ SA.\ SA\ !\ j = Suc\ (SA\ !\ i) \wedge i < j \wedge n < j)$
$)$

lemma *s-pred-invD*:
$[\![s\text{-}pred\text{-}inv\ \alpha\ T\ B\ SA\ k;\ in\text{-}s\text{-}current\text{-}bucket\ \alpha\ T\ B\ b\ i;\ b \neq 0]\!] \Longrightarrow$
$\exists j < length\ SA.\ SA\ !\ j = Suc\ (SA\ !\ i) \wedge i < j \wedge k < j$
⟨*proof*⟩

83.1.7 Successor

definition *s-suc-inv* ::

$('a :: \{linorder, order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow nat \Rightarrow bool$
where
$s\text{-}suc\text{-}inv\ \alpha\ T\ B\ SA\ n \equiv$
$\forall i < length\ SA.\ n < i \longrightarrow$
$(\forall j.\ SA\ !\ i = Suc\ j \wedge suffix\text{-}type\ T\ j = S\text{-}type \longrightarrow$
$(\exists k.\ in\text{-}s\text{-}current\text{-}bucket\ \alpha\ T\ B\ (\alpha\ (T\ !\ j))\ k \wedge SA\ !\ k = j \wedge k < i))$

lemma *s-suc-invD*:
$[\![s\text{-}suc\text{-}inv\ \alpha\ T\ B\ SA\ n;\ i < length\ SA;\ n < i;\ SA\ !\ i = Suc\ j;\ suffix\text{-}type\ T\ j = S\text{-}type]\!] \Longrightarrow$
$\exists k.\ in\text{-}s\text{-}current\text{-}bucket\ \alpha\ T\ B\ (\alpha\ (T\ !\ j))\ k \wedge SA\ !\ k = j \wedge k < i$
⟨*proof*⟩

83.1.8 Combined Permutation Invariant

definition *s-perm-inv* ::
$('a :: \{linorder, order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow nat \Rightarrow bool$
where
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \equiv$
$s\text{-}distinct\text{-}inv\ \alpha\ T\ B\ SA' \wedge$
$s\text{-}bucket\text{-}ptr\text{-}inv\ \alpha\ T\ B \wedge$
$s\text{-}locations\text{-}inv\ \alpha\ T\ B\ SA' \wedge$
$s\text{-}unchanged\text{-}inv\ \alpha\ T\ B\ SA\ SA' \wedge$
$s\text{-}seen\text{-}inv\ \alpha\ T\ B\ SA'\ n \wedge$
$s\text{-}pred\text{-}inv\ \alpha\ T\ B\ SA'\ n \wedge$
$s\text{-}suc\text{-}inv\ \alpha\ T\ B\ SA'\ n \wedge$
$strict\text{-}mono\ \alpha \wedge$
$\alpha\ (Max\ (set\ T)) < length\ B \wedge$
$length\ SA = length\ T \wedge$
$length\ SA' = length\ T \wedge$
$l\text{-}types\text{-}init\ \alpha\ T\ SA \wedge$
$valid\text{-}list\ T \wedge$
$\alpha\ bot = 0 \wedge$
$Suc\ 0 < length\ T$

lemma *s-perm-inv-elims*:
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow s\text{-}distinct\text{-}inv\ \alpha\ T\ B\ SA'$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow s\text{-}bucket\text{-}ptr\text{-}inv\ \alpha\ T\ B$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow s\text{-}locations\text{-}inv\ \alpha\ T\ B\ SA'$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow s\text{-}unchanged\text{-}inv\ \alpha\ T\ B\ SA\ SA'$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow s\text{-}seen\text{-}inv\ \alpha\ T\ B\ SA'\ n$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow s\text{-}pred\text{-}inv\ \alpha\ T\ B\ SA'\ n$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow s\text{-}suc\text{-}inv\ \alpha\ T\ B\ SA'\ n$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow strict\text{-}mono\ \alpha$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow \alpha\ (Max\ (set\ T)) < length\ B$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow length\ SA = length\ T$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow length\ SA' = length\ T$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow l\text{-}types\text{-}init\ \alpha\ T\ SA$

$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow valid\text{-}list\ T$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow \alpha\ bot = 0$
$s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \Longrightarrow Suc\ 0 < length\ T$
⟨proof⟩

fun *s-perm-inv-alt* ::
$('a :: \{linorder, order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \times nat\ list \times nat \Rightarrow bool$
where
$s\text{-}perm\text{-}inv\text{-}alt\ \alpha\ T\ SA\ (B, SA', n) = s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n$

83.1.9 Sorted

definition *s-sorted-inv*
where
$s\text{-}sorted\text{-}inv\ \alpha\ T\ B\ SA \equiv$
$(\forall b \le \alpha\ (Max\ (set\ T)).$
$ordlistns.sorted\ (map\ (suffix\ T)\ (list\text{-}slice\ SA\ (B\ !\ b)\ (bucket\text{-}end\ \alpha\ T\ b)))$
$)$

lemma *s-sorted-invD*:
$[\![s\text{-}sorted\text{-}inv\ \alpha\ T\ B\ SA;\ b \le \alpha\ (Max\ (set\ T))]\!] \Longrightarrow$
$ordlistns.sorted\ (map\ (suffix\ T)\ (list\text{-}slice\ SA\ (B\ !\ b)\ (bucket\text{-}end\ \alpha\ T\ b)))$
⟨proof⟩

fun *s-sorted-inv-alt* ::
$('a :: \{linorder, order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \times nat\ list \times nat \Rightarrow bool$
where
$s\text{-}sorted\text{-}inv\text{-}alt\ \alpha\ T\ SA\ (B, SA', n) =$
$(s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \land s\text{-}sorted\text{-}pre\ \alpha\ T\ SA \land s\text{-}sorted\text{-}inv\ \alpha\ T\ B\ SA')$

definition *s-prefix-sorted-inv*
where
$s\text{-}prefix\text{-}sorted\text{-}inv\ \alpha\ T\ B\ SA \equiv$
$(\forall b \le \alpha\ (Max\ (set\ T)).$
$ordlistns.sorted\ (map\ (lms\text{-}slice\ T)\ (list\text{-}slice\ SA\ (B\ !\ b)\ (bucket\text{-}end\ \alpha\ T\ b)))$
$)$

lemma *s-prefix-sorted-invD*:
$[\![s\text{-}prefix\text{-}sorted\text{-}inv\ \alpha\ T\ B\ SA;\ b \le \alpha\ (Max\ (set\ T))]\!] \Longrightarrow$
$ordlistns.sorted\ (map\ (lms\text{-}slice\ T)\ (list\text{-}slice\ SA\ (B\ !\ b)\ (bucket\text{-}end\ \alpha\ T\ b)))$
⟨proof⟩

fun *s-prefix-sorted-inv-alt* ::
$('a :: \{linorder, order\text{-}bot\} \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat\ list \Rightarrow nat\ list \times nat\ list \times nat \Rightarrow bool$
where
$s\text{-}prefix\text{-}sorted\text{-}inv\text{-}alt\ \alpha\ T\ SA\ (B, SA', n) =$

($s\text{-}perm\text{-}inv\ \alpha\ T\ B\ SA\ SA'\ n \wedge s\text{-}prefix\text{-}sorted\text{-}pre\ \alpha\ T\ SA \wedge s\text{-}prefix\text{-}sorted\text{-}inv\ \alpha\ T\ B\ SA'$)

83.2 Helpers

lemma *s-current-bucket-pairwise-distinct*:
assumes $s\text{-}distinct\text{-}inv\ \alpha\ T\ B\ SA$
and $s\text{-}locations\text{-}inv\ \alpha\ T\ B\ SA$
and $b \leq \alpha\ (Max\ (set\ T))$
and $b' \leq \alpha\ (Max\ (set\ T))$
and $b \neq b'$
shows $distinct\ (list\text{-}slice\ SA\ (B\ !\ b)\ (bucket\text{-}end\ \alpha\ T\ b)\ @\ list\text{-}slice\ SA\ (B\ !\ b')\ (bucket\text{-}end\ \alpha\ T\ b'))$
⟨*proof*⟩

lemma *s-unchanged-list-slice*:
assumes $s\text{-}unchanged\text{-}inv\ \alpha\ T\ B\ SA0\ SA$
and $length\ SA0 = length\ T$
and $length\ SA = length\ T$
and $b \leq \alpha\ (Max\ (set\ T))$
and $bucket\text{-}start\ \alpha\ T\ b \leq i$
and $j \leq B\ !\ b$
shows $list\text{-}slice\ SA\ i\ j = list\text{-}slice\ SA0\ i\ j$
⟨*proof*⟩

lemma *l-types-init-maintained*:
assumes $s\text{-}bucket\text{-}ptr\text{-}inv\ \alpha\ T\ B$
and $s\text{-}unchanged\text{-}inv\ \alpha\ T\ B\ SA0\ SA$
and $length\ SA0 = length\ T$
and $length\ SA = length\ T$
and $l\text{-}types\text{-}init\ \alpha\ T\ SA0$
shows $l\text{-}types\text{-}init\ \alpha\ T\ SA$
⟨*proof*⟩

lemma *s-sorted-pre-maintained*:
assumes $s\text{-}bucket\text{-}ptr\text{-}inv\ \alpha\ T\ B$
and $s\text{-}unchanged\text{-}inv\ \alpha\ T\ B\ SA0\ SA$
and $length\ SA0 = length\ T$
and $length\ SA = length\ T$
and $s\text{-}sorted\text{-}pre\ \alpha\ T\ SA0$
shows $s\text{-}sorted\text{-}pre\ \alpha\ T\ SA$
⟨*proof*⟩

lemma *s-prefix-sorted-pre-maintained*:
assumes $s\text{-}bucket\text{-}ptr\text{-}inv\ \alpha\ T\ B$
and $s\text{-}unchanged\text{-}inv\ \alpha\ T\ B\ SA0\ SA$
and $length\ SA0 = length\ T$
and $length\ SA = length\ T$
and $s\text{-}prefix\text{-}sorted\text{-}pre\ \alpha\ T\ SA0$

shows *s-prefix-sorted-pre* α *T SA*
⟨*proof*⟩

lemma *s-next-item-not-seen*:
assumes *s-distinct-inv* α *T B SA*
and *s-bucket-ptr-inv* α *T B*
and *s-locations-inv* α *T B SA*
and *s-unchanged-inv* α *T B SA0 SA*
and *s-seen-inv* α *T B SA i*
and *s-pred-inv* α *T B SA i*
and *strict-mono* α
and *length SA0* = *length T*
and *length SA* = *length T*
and *l-types-init* α *T SA0*
and *valid-list T*
and α *bot* = *0*
and *i* = *Suc n*
and *Suc n* < *length SA*
and *SA* ! *Suc n* = *Suc j*
and *suffix-type T j* = *S-type*
and *b* = α (*T* ! *j*)
shows *j* ∉ *set* (*list-slice SA* (*B* ! *b*) (*bucket-end* α *T b*))
⟨*proof*⟩

lemma *s-bucket-ptr-strict-lower-bound*:
assumes *s-distinct-inv* α *T B SA*
and *s-bucket-ptr-inv* α *T B*
and *s-locations-inv* α *T B SA*
and *s-unchanged-inv* α *T B SA0 SA*
and *s-seen-inv* α *T B SA i*
and *s-pred-inv* α *T B SA i*
and *strict-mono* α
and *length SA0* = *length T*
and *length SA* = *length T*
and *l-types-init* α *T SA0*
and *valid-list T*
and α *bot* = *0*
and *i* = *Suc n*
and *Suc n* < *length SA*
and *SA* ! *Suc n* = *Suc j*
and *suffix-type T j* = *S-type*
and *b* = α (*T* ! *j*)
shows *s-bucket-start* α *T b* < *B* ! *b*
⟨*proof*⟩

lemma *outside-another-bucket*:
assumes $b \neq b'$
and *bucket-start* α *T b* ≤ *i*
and *i* < *bucket-end* α *T b*

shows ¬(*bucket-start* α *T* $b' \leq i \wedge i <$ *bucket-end* α *T* b')
⟨*proof*⟩

lemma *s-B-val*:
assumes *s-distinct-inv* α *T B SA*
and *s-bucket-ptr-inv* α *T B*
and *s-locations-inv* α *T B SA*
and *s-unchanged-inv* α *T B SA0 SA*
and *s-seen-inv* α *T B SA i*
and *s-pred-inv* α *T B SA i*
and *strict-mono* α
and *length SA0* = *length T*
and *length SA* = *length T*
and *l-types-init* α *T SA0*
and *valid-list T*
and *length T* > *Suc 0*
and $b \leq \alpha$ (*Max* (*set T*))
and $i < B$! b
shows *B* ! *b* = *s-bucket-start* α *T b*
⟨*proof*⟩

lemma *s-bucket-eq-list-slice*:
assumes *s-distinct-inv* α *T B SA*
and *s-locations-inv* α *T B SA*
and *length SA* = *length T*
and $b \leq \alpha$ (*Max* (*set T*))
and *B* ! *b* = *s-bucket-start* α *T b*
shows *set* (*list-slice SA* (*s-bucket-start* α *T b*) (*bucket-end* α *T b*)) = *s-bucket* α *T b*
(**is** *set ?xs* = *s-bucket* α *T b*)
⟨*proof*⟩

lemma *bucket-eq-list-slice*:
assumes *s-distinct-inv* α *T B SA*
and *s-bucket-ptr-inv* α *T B*
and *s-locations-inv* α *T B SA*
and *s-unchanged-inv* α *T B SA0 SA*
and *length SA0* = *length T*
and *length SA* = *length T*
and *l-types-init* α *T SA0*
and $b \leq \alpha$ (*Max* (*set T*))
and *B* ! *b* = *s-bucket-start* α *T b*
shows *set* (*list-slice SA* (*bucket-start* α *T b*) (*bucket-end* α *T b*)) = *bucket* α *T b*
(**is** *set ?xs* = *bucket* α *T b*)
⟨*proof*⟩

lemma *s-index-lower-bound*:
assumes *s-bucket-ptr-inv* α *T B*
and *s-seen-inv* α *T B SA n*

and $i < \mathit{length}\ SA$
and $n \leq i$
shows *bucket-start* $\alpha\ T\ (\alpha\ (T\ !\ (SA\ !\ i))) \leq i$
(**is** *bucket-start* $\alpha\ T\ ?b \leq i$)
⟨*proof*⟩

lemma *s-index-upper-bound*:
assumes *s-bucket-ptr-inv* $\alpha\ T\ B$
and *s-seen-inv* $\alpha\ T\ B\ SA\ n$
and $i < \mathit{length}\ SA$
and $n \leq i$
shows $i <$ *bucket-end* $\alpha\ T\ (\alpha\ (T\ !\ (SA\ !\ i)))$
(**is** $i <$ *bucket-end* $\alpha\ T\ ?b$)
⟨*proof*⟩

83.3 Establishment and Maintenance Steps

83.3.1 Distinctness

lemma *s-distinct-inv-established*:
assumes *s-bucket-init* $\alpha\ T\ B$
and *valid-list* T
and *strict-mono* α
and $\alpha\ \mathit{bot} = 0$
shows *s-distinct-inv* $\alpha\ T\ B\ SA$
⟨*proof*⟩

lemma *s-distinct-inv-maintained-step*:
assumes *s-distinct-inv* $\alpha\ T\ B\ SA$
and *s-bucket-ptr-inv* $\alpha\ T\ B$
and *s-locations-inv* $\alpha\ T\ B\ SA$
and *s-unchanged-inv* $\alpha\ T\ B\ SA0\ SA$
and *s-seen-inv* $\alpha\ T\ B\ SA\ i$
and *s-pred-inv* $\alpha\ T\ B\ SA\ i$
and *strict-mono* α
and $\alpha\ (\mathit{Max}\ (\mathit{set}\ T)) < \mathit{length}\ B$
and $\mathit{length}\ SA0 = \mathit{length}\ T$
and $\mathit{length}\ SA = \mathit{length}\ T$
and *l-types-init* $\alpha\ T\ SA0$
and *valid-list* T
and $\alpha\ \mathit{bot} = 0$
and $i = \mathit{Suc}\ n$
and $\mathit{Suc}\ n < \mathit{length}\ SA$
and $SA\ !\ \mathit{Suc}\ n = \mathit{Suc}\ j$
and *suffix-type* $T\ j =$ *S-type*
and $b = \alpha\ (T\ !\ j)$
and $k = B\ !\ b - \mathit{Suc}\ 0$
shows *s-distinct-inv* $\alpha\ T\ (B[b := k])\ (SA[k := j])$
⟨*proof*⟩

corollary *s-distinct-inv-maintained-perm-step*:
assumes *s-perm-inv* α *T B SA0 SA i*
and $i = Suc\ n$
and $Suc\ n < length\ SA$
and $SA\ !\ Suc\ n = Suc\ j$
and $suffix\text{-}type\ T\ j = S\text{-}type$
and $b = \alpha\ (T\ !\ j)$
and $k = B\ !\ b - Suc\ 0$
shows *s-distinct-inv* α T $(B[b := k])$ $(SA[k := j])$
⟨*proof*⟩

83.3.2 Bucket Pointer

lemma *s-bucket-ptr-inv-established*:
assumes *s-bucket-init* α *T B*
and *valid-list T*
and *strict-mono* α
and $\alpha\ bot = 0$
shows *s-bucket-ptr-inv* α *T B*
⟨*proof*⟩

lemma *s-bucket-ptr-inv-maintained-step*:
assumes *s-distinct-inv* α *T B SA*
and *s-bucket-ptr-inv* α *T B*
and *s-locations-inv* α *T B SA*
and *s-unchanged-inv* α *T B SA0 SA*
and *s-seen-inv* α *T B SA i*
and *s-pred-inv* α *T B SA i*
and *strict-mono* α
and $\alpha\ (Max\ (set\ T)) < length\ B$
and $length\ SA0 = length\ T$
and $length\ SA = length\ T$
and *l-types-init* α *T SA0*
and *valid-list T*
and $\alpha\ bot = 0$
and $i = Suc\ n$
and $Suc\ n < length\ SA$
and $SA\ !\ Suc\ n = Suc\ j$
and $suffix\text{-}type\ T\ j = S\text{-}type$
and $b = \alpha\ (T\ !\ j)$
and $k = B\ !\ b - Suc\ 0$
shows *s-bucket-ptr-inv* α T $(B[b := k])$
⟨*proof*⟩

corollary *s-bucket-ptr-inv-maintained-perm-step*:
assumes *s-perm-inv* α *T B SA0 SA i*
and $i = Suc\ n$
and $Suc\ n < length\ SA$
and $SA\ !\ Suc\ n = Suc\ j$

and *suffix-type T j* = *S-type*
and *b* = *α* (*T* ! *j*)
and *k* = *B* ! *b* − *Suc 0*
shows *s-bucket-ptr-inv α T* (*B*[*b* := *k*])
⟨*proof*⟩

83.3.3 Locations

lemma *s-locations-inv-established*:
assumes *s-bucket-init α T B*
and *s-type-init T SA*
and *valid-list T*
and *strict-mono α*
and *α bot* = *0*
shows *s-locations-inv α T B SA*
⟨*proof*⟩

lemma *s-locations-inv-maintained-step*:
assumes *s-distinct-inv α T B SA*
and *s-bucket-ptr-inv α T B*
and *s-locations-inv α T B SA*
and *s-unchanged-inv α T B SA0 SA*
and *s-seen-inv α T B SA i*
and *s-pred-inv α T B SA i*
and *strict-mono α*
and *α* (*Max* (*set T*)) < *length B*
and *length SA0* = *length T*
and *length SA* = *length T*
and *l-types-init α T SA0*
and *valid-list T*
and *α bot* = *0*
and *i* = *Suc n*
and *Suc n* < *length SA*
and *SA* ! *Suc n* = *Suc j*
and *suffix-type T j* = *S-type*
and *b* = *α* (*T* ! *j*)
and *k* = *B* ! *b* − *Suc 0*
shows *s-locations-inv α T* (*B*[*b* := *k*]) (*SA*[*k* := *j*])
⟨*proof*⟩

corollary *s-locations-inv-maintained-perm-step*:
assumes *s-perm-inv α T B SA0 SA i*
and *i* = *Suc n*
and *Suc n* < *length SA*
and *SA* ! *Suc n* = *Suc j*
and *suffix-type T j* = *S-type*
and *b* = *α* (*T* ! *j*)
and *k* = *B* ! *b* − *Suc 0*
shows *s-locations-inv α T* (*B*[*b* := *k*]) (*SA*[*k* := *j*])

⟨*proof*⟩

83.3.4 Unchanged

lemma *s-unchanged-inv-established*:
shows *s-unchanged-inv* α *T B SA SA*
⟨*proof*⟩

lemma *s-unchanged-inv-maintained-step*:
assumes *s-distinct-inv* α *T B SA*
and *s-bucket-ptr-inv* α *T B*
and *s-locations-inv* α *T B SA*
and *s-unchanged-inv* α *T B SA0 SA*
and *s-seen-inv* α *T B SA i*
and *s-pred-inv* α *T B SA i*
and *strict-mono* α
and α (*Max* (*set T*)) < *length B*
and *length SA0* = *length T*
and *length SA* = *length T*
and *l-types-init* α *T SA0*
and *valid-list T*
and α *bot* = *0*
and *i* = *Suc n*
and *Suc n* < *length SA*
and *SA* ! *Suc n* = *Suc j*
and *suffix-type T j* = *S-type*
and *b* = α (*T* ! *j*)
and *k* = *B* ! *b* − *Suc 0*
shows *s-unchanged-inv* α *T* (*B*[*b* := *k*]) *SA0* (*SA*[*k* := *j*])
⟨*proof*⟩

corollary *s-unchanged-inv-maintained-perm-step*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *i* = *Suc n*
and *Suc n* < *length SA*
and *SA* ! *Suc n* = *Suc j*
and *suffix-type T j* = *S-type*
and *b* = α (*T* ! *j*)
and *k* = *B* ! *b* − *Suc 0*
shows *s-unchanged-inv* α *T* (*B*[*b* := *k*]) *SA0* (*SA*[*k* := *j*])
⟨*proof*⟩

83.3.5 Seen

lemma *s-seen-inv-established*:
assumes *length SA* = *length T*
and *length T* ≤ *n*
shows *s-seen-inv* α *T B SA n*
⟨*proof*⟩

lemma *s-seen-inv-maintained-step-c1*:
assumes *s-bucket-ptr-inv* α *T B*
and *s-unchanged-inv* α *T B SA0 SA*
and *s-seen-inv* α *T B SA i*
and *strict-mono* α
and *length SA0* = *length T*
and *length SA* = *length T*
and *l-types-init* α *T SA0*
and *valid-list T*
and *Suc 0* < *length T*
and *i* = *Suc n*
and *length SA* ≤ *Suc n*
shows *s-seen-inv* α *T B SA n*
⟨*proof*⟩

corollary *s-seen-inv-maintained-perm-step-c1*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *i* = *Suc n*
and *length SA* ≤ *Suc n*
shows *s-seen-inv* α *T B SA n*
⟨*proof*⟩

lemma *s-seen-inv-maintained-step-c1-alt*:
assumes *s-bucket-ptr-inv* α *T B*
and *s-unchanged-inv* α *T B SA0 SA*
and *s-seen-inv* α *T B SA i*
and *strict-mono* α
and *length SA0* = *length T*
and *length SA* = *length T*
and *l-types-init* α *T SA0*
and *valid-list T*
and *Suc 0* < *length T*
and *i* = *Suc n*
and *length T* ≤ *SA* ! *Suc n*
shows *s-seen-inv* α *T B SA n*
⟨*proof*⟩

corollary *s-seen-inv-maintained-perm-step-c1-alt*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *i* = *Suc n*
and *length T* ≤ *SA* ! *Suc n*
shows *s-seen-inv* α *T B SA n*
⟨*proof*⟩

lemma *s-seen-inv-maintained-step-c2*:
assumes *s-distinct-inv* α *T B SA*
and *s-bucket-ptr-inv* α *T B*
and *s-locations-inv* α *T B SA*
and *s-unchanged-inv* α *T B SA0 SA*

and *s-seen-inv* α *T B SA i*
and *s-pred-inv* α *T B SA i*
and *s-suc-inv* α *T B SA i*
and *strict-mono* α
and α (*Max* (*set T*)) < *length B*
and *length SA0* = *length T*
and *length SA* = *length T*
and *l-types-init* α *T SA0*
and *valid-list T*
and α *bot* = *0*
and *Suc 0* < *length T*
and *i* = *Suc n*
and *Suc n* < *length SA*
and *SA* ! *Suc n* = *0*
shows *s-seen-inv* α *T B SA n*
⟨*proof*⟩

corollary *s-seen-inv-maintained-perm-step-c2*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *i* = *Suc n*
and *Suc n* < *length SA*
and *SA* ! *Suc n* = *0*
shows *s-seen-inv* α *T B SA n*
⟨*proof*⟩

lemma *s-seen-inv-maintained-step-c3*:
assumes *s-distinct-inv* α *T B SA*
and *s-bucket-ptr-inv* α *T B*
and *s-locations-inv* α *T B SA*
and *s-unchanged-inv* α *T B SA0 SA*
and *s-seen-inv* α *T B SA i*
and *s-pred-inv* α *T B SA i*
and *s-suc-inv* α *T B SA i*
and *strict-mono* α
and α (*Max* (*set T*)) < *length B*
and *length SA0* = *length T*
and *length SA* = *length T*
and *l-types-init* α *T SA0*
and *valid-list T*
and α *bot* = *0*
and *Suc 0* < *length T*
and *i* = *Suc n*
and *Suc n* < *length SA*
and *SA* ! *Suc n* = *Suc j*
and *suffix-type T j* = *L-type*
shows *s-seen-inv* α *T B SA n*
⟨*proof*⟩

corollary *s-seen-inv-maintained-perm-step-c3*:

assumes *s-perm-inv* α *T B SA0 SA i*
and $i = Suc\ n$
and $Suc\ n < length\ SA$
and $SA\ !\ Suc\ n = Suc\ j$
and *suffix-type T j* = *L-type*
shows *s-seen-inv* α *T B SA n*
⟨*proof*⟩

lemma *s-seen-inv-maintained-step-c4*:
assumes *s-distinct-inv* α *T B SA*
and *s-bucket-ptr-inv* α *T B*
and *s-locations-inv* α *T B SA*
and *s-unchanged-inv* α *T B SA0 SA*
and *s-seen-inv* α *T B SA i*
and *s-pred-inv* α *T B SA i*
and *s-suc-inv* α *T B SA i*
and *strict-mono* α
and $\alpha\ (Max\ (set\ T)) < length\ B$
and $length\ SA0 = length\ T$
and $length\ SA = length\ T$
and *l-types-init* α *T SA0*
and *valid-list T*
and $\alpha\ bot = 0$
and $Suc\ 0 < length\ T$
and $i = Suc\ n$
and $Suc\ n < length\ SA$
and $SA\ !\ Suc\ n = Suc\ j$
and *suffix-type T j* = *S-type*
and $b = \alpha\ (T\ !\ j)$
and $k = B\ !\ b - Suc\ 0$
shows *s-seen-inv* α $T\ (B[b := k])\ (SA[k := j])\ n$
⟨*proof*⟩

corollary *s-seen-inv-maintained-perm-step-c4*:
assumes *s-perm-inv* α *T B SA0 SA i*
and $i = Suc\ n$
and $Suc\ n < length\ SA$
and $SA\ !\ Suc\ n = Suc\ j$
and *suffix-type T j* = *S-type*
and $b = \alpha\ (T\ !\ j)$
and $k = B\ !\ b - Suc\ 0$
shows *s-seen-inv* α $T\ (B[b := k])\ (SA[k := j])\ n$
⟨*proof*⟩

lemmas *s-seen-inv-maintained-perm-step* =
s-seen-inv-maintained-perm-step-c1
s-seen-inv-maintained-perm-step-c2
s-seen-inv-maintained-perm-step-c3
s-seen-inv-maintained-perm-step-c4

83.3.6 Predecessor

lemma *s-pred-inv-established*:
assumes *s-bucket-init* α *T B*
shows *s-pred-inv* α *T B SA n*
⟨*proof*⟩

lemma *s-pred-inv-maintained-step-alt*:
assumes *s-pred-inv* α *T B SA i*
and $i = Suc\ n$
shows *s-pred-inv* α *T B SA n*
⟨*proof*⟩

corollary *s-pred-inv-maintained-perm-step-alt*:
assumes *s-perm-inv* α *T B SA0 SA i*
and $i = Suc\ n$
shows *s-pred-inv* α *T B SA n*
⟨*proof*⟩

lemma *s-pred-inv-maintained-step*:
assumes *s-distinct-inv* α *T B SA*
and *s-bucket-ptr-inv* α *T B*
and *s-locations-inv* α *T B SA*
and *s-unchanged-inv* α *T B SA0 SA*
and *s-seen-inv* α *T B SA i*
and *s-pred-inv* α *T B SA i*
and *s-suc-inv* α *T B SA i*
and *strict-mono* α
and $\alpha\ (Max\ (set\ T)) < length\ B$
and $length\ SA0 = length\ T$
and $length\ SA = length\ T$
and *l-types-init* α *T SA0*
and *valid-list T*
and $\alpha\ bot = 0$
and $Suc\ 0 < length\ T$
and $i = Suc\ n$
and $Suc\ n < length\ SA$
and $SA\ !\ Suc\ n = Suc\ j$
and $suffix\text{-}type\ T\ j = S\text{-}type$
and $b = \alpha\ (T\ !\ j)$
and $k = B\ !\ b - Suc\ 0$
shows *s-pred-inv* α $T\ (B[b := k])\ (SA[k := j])\ n$
⟨*proof*⟩

corollary *s-pred-inv-maintained-perm-step*:
assumes *s-perm-inv* α *T B SA0 SA i*
and $i = Suc\ n$
and $Suc\ n < length\ SA$
and $SA\ !\ Suc\ n = Suc\ j$
and $suffix\text{-}type\ T\ j = S\text{-}type$

and $b = \alpha\ (T\ !\ j)$
and $k = B\ !\ b - Suc\ 0$
shows *s-pred-inv* $\alpha\ T\ (B[b := k])\ (SA[k := j])\ n$
⟨*proof*⟩

83.3.7 Successor

lemma *s-suc-inv-established*:
assumes *length SA* = *length T*
and *length T* ≤ *n*
shows *s-suc-inv* α *T B SA n*
⟨*proof*⟩

lemma *s-suc-inv-maintained-step-c1*:
assumes *length SA* ≤ *Suc n*
shows *s-suc-inv* α *T B SA n*
⟨*proof*⟩

corollary *s-suc-inv-maintained-perm-step-c1*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *i* = *Suc n*
and *length SA* ≤ *Suc n*
shows *s-suc-inv* α *T B SA n*
⟨*proof*⟩

lemma *s-suc-inv-maintained-step-c1-alt*:
assumes *s-suc-inv* α *T B SA i*
and *s-bucket-ptr-inv* α *T B*
and *s-locations-inv* α *T B SA*
and *strict-mono* α
and α (*Max* (*set T*)) < *length B*
and *valid-list T*
and α *bot* = *0*
and *i* = *Suc n*
and *length T* ≤ *SA* ! *Suc n*
shows *s-suc-inv* α *T B SA n*
⟨*proof*⟩

corollary *s-suc-inv-maintained-perm-step-c1-alt*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *i* = *Suc n*
and *length T* ≤ *SA* ! *Suc n*
shows *s-suc-inv* α *T B SA n*
⟨*proof*⟩

lemma *s-suc-inv-maintained-step-c2*:
assumes *s-suc-inv* α *T B SA i*
and *i* = *Suc n*
and *Suc n* < *length SA*

and *SA ! Suc n = 0*
shows *s-suc-inv α T B SA n*
⟨*proof*⟩

corollary *s-suc-inv-maintained-perm-step-c2*:
assumes *s-perm-inv α T B SA0 SA i*
and *i = Suc n*
and *Suc n < length SA*
and *SA ! Suc n = 0*
shows *s-suc-inv α T B SA n*
⟨*proof*⟩

lemma *s-suc-inv-maintained-step-c3*:
assumes *s-suc-inv α T B SA i*
and *i = Suc n*
and *Suc n < length SA*
and *SA ! Suc n = Suc j*
and *suffix-type T j = L-type*
shows *s-suc-inv α T B SA n*
⟨*proof*⟩

corollary *s-suc-inv-maintained-perm-step-c3*:
assumes *s-perm-inv α T B SA0 SA i*
and *i = Suc n*
and *Suc n < length SA*
and *SA ! Suc n = Suc j*
and *suffix-type T j = L-type*
shows *s-suc-inv α T B SA n*
⟨*proof*⟩

lemma *s-suc-inv-maintained-step-c4*:
assumes *s-distinct-inv α T B SA*
and *s-bucket-ptr-inv α T B*
and *s-locations-inv α T B SA*
and *s-unchanged-inv α T B SA0 SA*
and *s-seen-inv α T B SA i*
and *s-pred-inv α T B SA i*
and *s-suc-inv α T B SA i*
and *strict-mono α*
and *α (Max (set T)) < length B*
and *length SA0 = length T*
and *length SA = length T*
and *l-types-init α T SA0*
and *valid-list T*
and *α bot = 0*
and *Suc 0 < length T*
and *i = Suc n*
and *Suc n < length SA*
and *SA ! Suc n = Suc j*

and *suffix-type T j = S-type*
and $b = \alpha$ (*T* ! *j*)
and *k* = *B* ! *b* − *Suc 0*
shows *s-suc-inv* α *T* (*B*[*b* := *k*]) (*SA*[*k* := *j*]) *n*
⟨*proof*⟩

corollary *s-suc-inv-maintained-perm-step-c4*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *i* = *Suc n*
and *Suc n* < *length SA*
and *SA* ! *Suc n* = *Suc j*
and *suffix-type T j = S-type*
and $b = \alpha$ (*T* ! *j*)
and *k* = *B* ! *b* − *Suc 0*
shows *s-suc-inv* α *T* (*B*[*b* := *k*]) (*SA*[*k* := *j*]) *n*
⟨*proof*⟩

lemmas *s-suc-inv-maintained-perm-step* =
s-suc-inv-maintained-step-c1
s-suc-inv-maintained-perm-step-c2
s-suc-inv-maintained-perm-step-c3
s-suc-inv-maintained-perm-step-c4

83.3.8 Combined Permutation Invariant

lemma *s-perm-inv-established*:
assumes *s-bucket-init* α *T B*
and *s-type-init T SA*
and *strict-mono* α
and α (*Max* (*set T*)) < *length B*
and *length SA* = *length T*
and *l-types-init* α *T SA*
and *valid-list T*
and α *bot* = *0*
and *Suc 0* < *length T*
and *length T* ≤ *n*
shows *s-perm-inv* α *T B SA SA n*
⟨*proof*⟩

lemma *s-perm-inv-maintained-step-c1*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *i* = *Suc n*
and *length SA* ≤ *Suc n*
shows *s-perm-inv* α *T B SA0 SA n*
⟨*proof*⟩

lemma *s-perm-inv-maintained-step-c1-alt*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *i* = *Suc n*

and *length* $T \leq SA$! *Suc n*
shows *s-perm-inv* α *T B SA0 SA n*
⟨*proof*⟩

lemma *s-perm-inv-maintained-step-c2*:
assumes *s-perm-inv* α *T B SA0 SA i*
and $i = Suc\ n$
and $Suc\ n < length\ SA$
and $SA\ !\ Suc\ n = 0$
shows *s-perm-inv* α *T B SA0 SA n*
⟨*proof*⟩

lemma *s-perm-inv-maintained-step-c3*:
assumes *s-perm-inv* α *T B SA0 SA i*
and $i = Suc\ n$
and $Suc\ n < length\ SA$
and $SA\ !\ Suc\ n = Suc\ j$
and *suffix-type T j* = *L-type*
shows *s-perm-inv* α *T B SA0 SA n*
⟨*proof*⟩

lemma *s-perm-inv-maintained-step-c4*:
assumes *s-perm-inv* α *T B SA0 SA i*
and $i = Suc\ n$
and $Suc\ n < length\ SA$
and $SA\ !\ Suc\ n = Suc\ j$
and *suffix-type T j* = *S-type*
and $b = \alpha\ (T\ !\ j)$
and $k = B\ !\ b - Suc\ 0$
shows *s-perm-inv* α T $(B[b := k])$ *SA0* $(SA[k := j])$ n
⟨*proof*⟩

theorem *abs-induce-s-perm-step*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *abs-induce-s-step* (B, SA, i) $(\alpha, T) = (B', SA', i')$
shows *s-perm-inv* α T B' *SA0* SA' i'
⟨*proof*⟩

corollary *abs-induce-s-perm-step-alt*:
$\bigwedge a$. *s-perm-inv-alt* α *T SA0 a* $\Longrightarrow$ *s-perm-inv-alt* α *T SA0* (*abs-induce-s-step a* (α, T))
⟨*proof*⟩

theorem *abs-induce-s-perm-alt-maintained*:
assumes *s-perm-inv-alt* α *T SA0* $(B, SA, length\ T)$
shows *s-perm-inv-alt* α *T SA0* (*abs-induce-s-base* α *T B SA*)
⟨*proof*⟩

corollary *abs-induce-s-perm-maintained*:

assumes *abs-induce-s-base* α *T B SA* = (*B*′, *SA*′, *n*)
and *s-perm-inv* α *T B SA0 SA* (*length T*)
shows *s-perm-inv* α *T B*′ *SA0 SA*′ *n*
⟨*proof*⟩

lemma *s-perm-inv-0-B-val*:
assumes *s-perm-inv* α *T B SA SA*′ *0*
and $b \leq \alpha$ (*Max* (*set T*))
shows *B* ! *b* = *s-bucket-start* α *T b*
⟨*proof*⟩

lemma *s-perm-inv-0-list-slice-bucket*:
assumes *s-perm-inv* α *T B SA SA*′ *0*
and $b \leq \alpha$ (*Max* (*set T*))
shows *set* (*list-slice SA*′ (*bucket-start* α *T b*) (*bucket-end* α *T b*)) = *bucket* α *T b*
⟨*proof*⟩

lemma *s-perm-inv-0-distinct-list-slice*:
assumes *s-perm-inv* α *T B SA SA*′ *0*
and $b \leq \alpha$ (*Max* (*set T*))
shows *distinct* (*list-slice SA*′ (*bucket-start* α *T b*) (*bucket-end* α *T b*))
(**is** *distinct ?xs*)
⟨*proof*⟩

lemma *abs-induce-s-base-distinct*:
assumes *abs-induce-s-base* α *T B SA* = (*B*′, *SA*′, *n*)
and *s-perm-inv* α *T B*′ *SA SA*′ *n*
shows *distinct SA*′
⟨*proof*⟩

lemma *abs-induce-s-base-subset-upt*:
assumes *abs-induce-s-base* α *T B SA* = (*B*′, *SA*′, *n*)
and *s-perm-inv* α *T B*′ *SA SA*′ *n*
shows *set SA*′ ⊆ {*0*..<*length T*}
⟨*proof*⟩

corollary *abs-induce-s-base-eq-upt*:
assumes *abs-induce-s-base* α *T B SA* = (*B*′, *SA*′, *n*)
and *s-perm-inv* α *T B*′ *SA SA*′ *n*
shows *set SA*′ = {*0*..<*length T*}
⟨*proof*⟩

theorem *abs-induce-s-base-perm*:
assumes *abs-induce-s-base* α *T B SA* = (*B*′, *SA*′, *n*)
and *s-perm-inv* α *T B*′ *SA SA*′ *n*
shows *SA*′ <~~> [*0*..< *length T*]
⟨*proof*⟩

83.3.9 Sorted

lemma *s-sorted-established*:
assumes *s-bucket-init* α *T B*
and *strict-mono* α
and *valid-list T*
and α *bot* = *0*
and $b \leq \alpha$ (*Max* (*set T*))
shows *sorted-wrt R* (*list-slice SA* (*B* ! *b*) (*bucket-end* α *T b*))
(**is** *sorted-wrt R ?xs*)
⟨*proof*⟩

lemma *s-sorted-inv-established*:
assumes *s-bucket-init* α *T B*
and *strict-mono* α
and *valid-list T*
and α *bot* = *0*
shows *s-sorted-inv* α *T B SA*
⟨*proof*⟩

lemma *s-prefix-sorted-inv-established*:
assumes *s-bucket-init* α *T B*
and *strict-mono* α
and *valid-list T*
and α *bot* = *0*
shows *s-prefix-sorted-inv* α *T B SA*
⟨*proof*⟩

lemma *s-sorted-maintained-unchanged-step*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *i* = *Suc n*
and *Suc n* < *length SA*
and *SA* ! *Suc n* = *Suc j*
and *suffix-type T j* = *S-type*
and $b = \alpha$ (*T* ! *j*)
and *k* = *B* ! *b* − *Suc 0*
and $b' \leq \alpha$ (*Max* (*set T*))
and *sorted-wrt R* (*list-slice SA* (*B* ! *b'*) (*bucket-end* α *T b'*))
and $b \neq b'$
shows *sorted-wrt R* (*list-slice* (*SA*[*k* := *j*]) ((*B*[*b* := *k*]) ! *b'*) (*bucket-end* α *T b'*))
⟨*proof*⟩

lemma *s-sorted-inv-maintained-step*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *s-sorted-pre* α *T SA0*
and *s-sorted-inv* α *T B SA*
and *i* = *Suc n*
and *Suc n* < *length SA*
and *SA* ! *Suc n* = *Suc j*
and *suffix-type T j* = *S-type*

and $b = \alpha\ (T\ !\ j)$
and $k = B\ !\ b - Suc\ 0$
shows *s-sorted-inv* $\alpha\ T\ (B[b := k])\ (SA[k := j])$
⟨*proof*⟩

lemma *s-prefix-sorted-inv-maintained-step*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *s-prefix-sorted-pre* α *T SA0*
and *s-prefix-sorted-inv* α *T B SA*
and $i = Suc\ n$
and $Suc\ n < length\ SA$
and $SA\ !\ Suc\ n = Suc\ j$
and *suffix-type T j* = *S-type*
and $b = \alpha\ (T\ !\ j)$
and $k = B\ !\ b - Suc\ 0$
shows *s-prefix-sorted-inv* $\alpha\ T\ (B[b := k])\ (SA[k := j])$
⟨*proof*⟩

theorem *abs-induce-s-sorted-step*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *s-sorted-pre* α *T SA0*
and *s-sorted-inv* α *T B SA*
and *abs-induce-s-step* $(B, SA, i)\ (\alpha, T) = (B', SA', i')$
shows *s-sorted-inv* $\alpha\ T\ B'\ SA'$
⟨*proof*⟩

corollary *abs-induce-s-sorted-step-alt*:
$\bigwedge a.$ *s-sorted-inv-alt* α *T SA0 a* $\Longrightarrow$ *s-sorted-inv-alt* α *T SA0* (*abs-induce-s-step a* (α, T))
⟨*proof*⟩

theorem *abs-induce-s-sorted-alt-maintained*:
assumes *s-sorted-inv-alt* α *T SA0* (*B*, *SA*, *length T*)
shows *s-sorted-inv-alt* α *T SA0* (*abs-induce-s-base* α *T B SA*)
⟨*proof*⟩

corollary *abs-induce-s-sorted-maintained*:
assumes *abs-induce-s-base* α *T B SA* $= (B', SA', n)$
and *s-perm-inv* α *T B SA0 SA* (*length T*)
and *s-sorted-pre* α *T SA0*
and *s-sorted-inv* α *T B SA*
shows *s-sorted-inv* $\alpha\ T\ B'\ SA'$
⟨*proof*⟩

theorem *abs-induce-s-prefix-sorted-step*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *s-prefix-sorted-pre* α *T SA0*
and *s-prefix-sorted-inv* α *T B SA*
and *abs-induce-s-step* $(B, SA, i)\ (\alpha, T) = (B', SA', i')$

shows *s-prefix-sorted-inv* α *T B′ SA′*
⟨*proof*⟩

corollary *abs-induce-s-prefix-sorted-step-alt*:
$\bigwedge$*a*. *s-prefix-sorted-inv-alt* α *T SA0 a* $\Longrightarrow$
s-prefix-sorted-inv-alt α *T SA0* (*abs-induce-s-step a* (α, *T*))
⟨*proof*⟩

theorem *abs-induce-s-prefix-sorted-alt-maintained*:
assumes *s-prefix-sorted-inv-alt* α *T SA0* (*B*, *SA*, *length T*)
shows *s-prefix-sorted-inv-alt* α *T SA0* (*abs-induce-s-base* α *T B SA*)
⟨*proof*⟩

corollary *abs-induce-s-prefix-sorted-maintained*:
assumes *abs-induce-s-base* α *T B SA* = (*B′*, *SA′*, *n*)
and *s-perm-inv* α *T B SA0 SA* (*length T*)
and *s-prefix-sorted-pre* α *T SA0*
and *s-prefix-sorted-inv* α *T B SA*
shows *s-prefix-sorted-inv* α *T B′ SA′*
⟨*proof*⟩

theorem *s-sorted-bucket*:
assumes *s-perm-inv* α *T B SA0 SA 0*
and *s-sorted-pre* α *T SA0*
and *s-sorted-inv* α *T B SA*
and $b \leq \alpha$ (*Max* (*set T*))
shows *ordlistns.sorted* (*map* (*suffix T*) (*list-slice SA* (*bucket-start* α *T b*) (*bucket-end* α *T b*)))
(**is** *ordlistns.sorted* (*map* (*suffix T*) *?xs*))
⟨*proof*⟩

theorem *abs-induce-s-base-sorted*:
assumes *abs-induce-s-base* α *T B SA* = (*B′*, *SA′*, *n*)
and *s-perm-inv* α *T B SA0 SA* (*length T*)
and *s-sorted-pre* α *T SA0*
and *s-sorted-inv* α *T B SA*
shows *ordlistns.sorted* (*map* (*suffix T*) *SA′*)
⟨*proof*⟩

theorem *s-prefix-sorted-bucket*:
assumes *s-perm-inv* α *T B SA0 SA 0*
and *s-prefix-sorted-pre* α *T SA0*
and *s-prefix-sorted-inv* α *T B SA*
and $b \leq \alpha$ (*Max* (*set T*))
shows *ordlistns.sorted* (*map* (*lms-slice T*) (*list-slice SA* (*bucket-start* α *T b*) (*bucket-end* α *T b*)))
(**is** *ordlistns.sorted* (*map* (*lms-slice T*) *?xs*))
⟨*proof*⟩

theorem *abs-induce-s-base-prefix-sorted*:
  assumes *abs-induce-s-base α T B SA* = (*B′*, *SA′*, *n*)
  and *s-perm-inv α T B SA0 SA* (*length T*)
  and *s-prefix-sorted-pre α T SA0*
  and *s-prefix-sorted-inv α T B SA*
shows *ordlistns.sorted* (*map* (*lms-slice T*) *SA′*)
⟨*proof*⟩

84 Induce S Correctness Theorems

theorem *abs-induce-s-perm*:
  assumes *s-perm-pre α T B SA* (*length T*)
  shows *abs-induce-s α T B SA* <~~> [*0*..< *length T*]
⟨*proof*⟩
theorem *abs-induce-s-sorted*:
  assumes *s-perm-pre α T B SA* (*length T*)
  and *s-sorted-pre α T SA*
shows *ordlistns.sorted* (*map* (*suffix T*) (*abs-induce-s α T B SA*))
⟨*proof*⟩
theorem *abs-induce-s-prefix-sorted*:
  assumes *s-perm-pre α T B SA* (*length T*)
  and *s-prefix-sorted-pre α T SA*
shows *ordlistns.sorted* (*map* (*lms-slice T*) (*abs-induce-s α T B SA*))
⟨*proof*⟩

end
theory *Abs-Induce-Verification*
  imports
    Abs-Induce-L-Verification
    Abs-Induce-S-Verification
    Abs-Bucket-Insert-Verification
begin

85 Bucket Initialisation Properties

lemma *l-bucket-init-map-bucket-start*:
  l-bucket-init α T (*map* (*bucket-start α T*) [*0*..<*Suc* (*α* (*Max* (*set T*)))])
  ⟨*proof*⟩

lemma *lms-bucket-init-map-bucket-end*:
  lms-bucket-init α T (*map* (*bucket-end α T*) [*0*..<*Suc* (*α* (*Max* (*set T*)))])
  ⟨*proof*⟩

lemma *s-bucket-init-map-bucket-end*:
  s-bucket-init α T ((*map* (*bucket-end α T*) [*0*..<*Suc* (*α* (*Max* (*set T*)))])[*0* := *0*])
  ⟨*proof*⟩

abbreviation *bucket-starts α T* ≡ *map* (*bucket-start α T*) [*0*..<*Suc* (*α* (*Max* (*set*

T)))]

abbreviation *bucket-ends* α *T* ≡ *map* (*bucket-end* α *T*) [*0*..<*Suc* (α (*Max* (*set T*)))]

86 Bucket Insert Precondition

lemma *lms-pre-established*:
  assumes *set LMS* = {*i*. *abs-is-lms T i*}
  and *distinct LMS*
  and *strict-mono* α
shows *lms-pre* α *T* (*bucket-ends* α *T*) (*replicate* (*length T*) (*length T*)) (*rev LMS*)
  (**is** *lms-pre* α *T ?B ?SA* (*rev LMS*))
⟨*proof*⟩

87 Induce L Precondition

lemma *l-perm-pre-established*:
  assumes *valid-list T*
  and *strict-mono* α
  and *lms-pre* α *T B SA* (*rev LMS*)
shows *l-perm-pre* α *T* (*bucket-starts* α *T*) (*abs-bucket-insert* α *T B SA* (*rev LMS*))
  (**is** *l-perm-pre* α *T ?B ?SA*)
⟨*proof*⟩

88 Induce S Precondition

lemma *s-perm-pre-established*:
  assumes *valid-list T*
  and *strict-mono* α
  and α *bot* = *0*
  and *Suc 0* < *length T*
  and *lms-pre* α *T B0 SA0* (*rev LMS*)
  and *SA1* = *abs-bucket-insert* α *T B0 SA0* (*rev LMS*)
  and *l-perm-pre* α *T B1 SA1*
shows *s-perm-pre* α *T* ((*bucket-ends* α *T*)[*0* := *0*]) (*abs-induce-l* α *T B1 SA1*) (*length T*)
  (**is** *s-perm-pre* α *T ?B ?SA ?n*)
⟨*proof*⟩

89 Permutation

lemma *abs-sa-induce-permutation*:
  assumes *set LMS* = {*i*. *abs-is-lms T i*}
  and *distinct LMS*
  and *valid-list T*
  and *strict-mono* α

and *α bot = 0*
and *Suc 0 < length T*
shows *abs-sa-induce α T LMS <~~> [0..< length T]*
⟨*proof*⟩

90 Sorting

lemma *abs-sa-induce-suffix-sorted*:
assumes *set LMS = {i. abs-is-lms T i}*
and *distinct LMS*
and *valid-list T*
and *strict-mono α*
and *α bot = 0*
and *Suc 0 < length T*
and *ordlistns.sorted (map (suffix T) LMS)*
shows *ordlistns.sorted (map (suffix T) (abs-sa-induce α T LMS))*
⟨*proof*⟩

theorem *abs-sa-induce-prefix-sorted*:
assumes *set LMS = {i. abs-is-lms T i}*
and *distinct LMS*
and *valid-list T*
and *strict-mono α*
and *α bot = 0*
and *Suc 0 < length T*
shows *ordlistns.sorted (map (lms-slice T) (abs-sa-induce α T LMS))*
⟨*proof*⟩

end
theory *Abs-Extract-LMS-Verification*
imports *../abs−def/Abs-SAIS Abs-Induce-Verification*
begin

91 Extract LMS types Proofs

lemma *abs-extract-lms-correct*:
xs <~~> [0..<length T] ⟹
distinct (abs-extract-lms T xs) ∧ set (abs-extract-lms T xs) = {i. abs-is-lms T i}
⟨*proof*⟩

lemma *set-abs-extract-lms-eq-all-lms*:
set (abs-extract-lms T [0..<length T]) = {i. abs-is-lms T i}
⟨*proof*⟩

lemma *distinct-abs-extract-lms*:

distinct (*abs-extract-lms T* [*0*..<*length T*])
⟨*proof*⟩

lemma *filter-abs-sa-induce-eq-all-lms*:
⟦*set LMS* = {*i*. *abs-is-lms T i*}; *distinct LMS*; *valid-list T*; *strict-mono* α; α *bot* = *0*;
Suc 0 < *length T*⟧ ⟹
set (*abs-extract-lms T* (*abs-sa-induce* α *T LMS*)) = {*i*. *abs-is-lms T i*}
⟨*proof*⟩

lemma *distinct-filter-abs-sa-induce*:
⟦*set LMS* = {*i*. *abs-is-lms T i*}; *distinct LMS*; *valid-list T*; *strict-mono* α; α *bot* = *0*;
Suc 0 < *length T*⟧ ⟹
distinct (*abs-extract-lms T* (*abs-sa-induce* α *T LMS*))
⟨*proof*⟩

end
theory *Abs-Order-LMS-Verification*
imports ../*abs*−*def*/*Abs-SAIS*
begin

92 Order LMS-types Proofs

lemma *abs-order-lms-eq-map-nth*:
order-lms LMS xs = *map* (*nth LMS*) *xs*
⟨*proof*⟩

theorem *distinct-abs-order-lms*:
⟦*xs* <~~> [*0*..<*length LMS*]; *distinct LMS*⟧ ⟹
distinct (*order-lms LMS xs*)
⟨*proof*⟩

theorem *abs-order-lms-eq-all-lms*:
⟦*xs* <~~> [*0*..<*length LMS*]; *set LMS* = *S*⟧ ⟹
set (*order-lms LMS xs*) = *S*
⟨*proof*⟩

end
theory *Abs-Rename-LMS-Verification*
imports ../*abs*−*def*/*Abs-SAIS*

begin

93 Rename Mapping Proofs

lemma *abs-rename-mapping′-length*:
  length (*abs-rename-mapping′ T LMS names i*) = *length names*
  ⟨*proof*⟩

lemma *abs-rename-mapping-length*:
  length (*abs-rename-mapping T LMS*) = *length T*
  ⟨*proof*⟩

lemma *rename-mapping′-unchanged*:
  ⟦*x* ∉ *set LMS*; *x* < *length names*⟧ ⟹
    (*abs-rename-mapping′ T LMS names i*) ! *x* = *names* ! *x*
  ⟨*proof*⟩

lemma *rename-mapping′-lms*:
  assumes *distinct LMS*
  and *ordlistns.sorted* (*map* (*lms-slice T*) *LMS*)
  and *i* ∈ *set LMS*
  and *i* < *length names*
shows (*abs-rename-mapping′ T LMS names j*) ! *i* =
  j + (*ordlistns.elem-rank* ((*lms-slice T*) ‘ *set LMS*) (*lms-slice T i*))
  ⟨*proof*⟩

lemma *abs-rename-mapping-lms*:
  assumes *distinct LMS*
  and *ordlistns.sorted* (*map* (*lms-slice T*) *LMS*)
  and *i* ∈ *set LMS*
  and *i* < *length T*
shows (*abs-rename-mapping T LMS*) ! *i* =
  ordlistns.elem-rank ((*lms-slice T*) ‘ *set LMS*) (*lms-slice T i*)
  ⟨*proof*⟩

lemma *abs-rename-mapping-lms-all*:
  assumes *distinct LMS*
  and *ordlistns.sorted* (*map* (*lms-slice T*) *LMS*)
  and ∀ *x* ∈ *set LMS*. *x* < *length T*
shows ∀ *x* ∈ *set LMS*. (!) (*abs-rename-mapping T LMS*) *x* =
  ordlistns.elem-rank (*lms-slice T* ‘ *set LMS*) (*lms-slice T x*)
  ⟨*proof*⟩

lemma *map-abs-rename-mapping*:
  assumes *distinct LMS*
  and *ordlistns.sorted* (*map* (*lms-slice T*) *LMS*)
  and ∀ *x* ∈ *set LMS*. *x* < *length T*
  and *set xs* ⊆ *set LMS*

shows *map* ((!) (*abs-rename-mapping T LMS*)) *xs* =
map (*ordlistns.elem-rank* (*lms-slice T* ‘ *set LMS*)) (*map* (*lms-slice T*) *xs*)
⟨*proof*⟩

94 Rename String Proofs

lemma *rename-list-length*:
length (*rename-string xs names*) = *length xs*
⟨*proof*⟩

theorem *rename-list-correct*:
rename-string T names = *map* (λ*x*. *names* ! *x*) *T*
⟨*proof*⟩

corollary *rename-list-nth*:
i < *length T* ⟹ (*rename-string T names*) ! *i* = *names* ! (*T* ! *i*)
⟨*proof*⟩

end
theory *Abs-SAIS-Verification-With-Valid-Precondition*
imports
Abs-Induce-Verification
Abs-Rename-LMS-Verification
Abs-Extract-LMS-Verification
Abs-Order-LMS-Verification
begin

95 SAIS General Helpers

termination *abs-sais*
⟨*proof*⟩

lemma *abs-sais-reduced-string*:
assumes *LMS1* = *lms0-seq T*
and *distinct LMS2*
and *set LMS2* = {*i*. *abs-is-lms T i*}
and *ordlistns.sorted* (*map* (*lms-slice T*) *LMS2*)
and *names* = *abs-rename-mapping T LMS2*
and *T*′ = *rename-string LMS1 names*
shows *T*′ = *lms-map T* (*lms0-suffix T*)
⟨*proof*⟩

96 SAIS cases simplifications

lemma *abs-sais-distinct-simp*:

assumes *T = a # b # xs*
and *LMS0 = abs-extract-lms T [0..<length T]*
and *SA = abs-sa-induce id T LMS0*
and *LMS = abs-extract-lms T SA*
and *names = abs-rename-mapping T LMS*
and *T′ = rename-string LMS0 names*
and *distinct T′*
shows *abs-sais T = abs-sa-induce id T LMS*
⟨*proof*⟩

lemma *abs-sais-not-distinct-simp*:
assumes *T = a # b # xs*
and *LMS0 = abs-extract-lms T [0..<length T]*
and *SA = abs-sa-induce id T LMS0*
and *LMS = abs-extract-lms T SA*
and *names = abs-rename-mapping T LMS*
and *T′ = rename-string LMS0 names*
and *LMS1 = order-lms LMS0 (abs-sais T′)*
and *¬ distinct T′*
shows *abs-sais T = abs-sa-induce id T LMS1*
⟨*proof*⟩

97 SAIS returns a permutation

theorem *abs-sais-permutation*:
valid-list T ⟹ abs-sais T <~~> [0..<length T]
⟨*proof*⟩

98 SAIS Sorted Helpers

lemma *abs-sais-subset-idx*:
assumes *valid-list T*
shows *set (abs-sais T) ⊆ {0..<length T}*
⟨*proof*⟩

99 SAIS sorts suffixes

theorem *abs-sais-sorted-alt*:
valid-list T ⟹
ordlistns.strict-sorted (map (suffix T) (abs-sais T))
⟨*proof*⟩

theorem *abs-sais-sorted*:
valid-list T ⟹
strict-sorted (map (suffix T) (abs-sais T))
⟨*proof*⟩

100 Verification of a SAIS construction algorithm

interpretation *abs-sais*: *Suffix-Array-Restricted abs-sais*
⟨*proof*⟩

end
theory *Abs-SAIS-Verification*
imports *Abs-SAIS-Verification-With-Valid-Precondition*
begin

101 Final Theorem: Verification of a generalised SAIS construction algorithm

The @term *abs-sais* implementation produces an output that is equivalent to that of a suffix array construction algorithm for lists of any type that can be linearly ordered. This lifts the restriction that the algorithm only operates on natural numbers terminated by a bottom element.

interpretation *abs-sais-gen*: *Suffix-Array-General sa-nat-wrapper map-to-nat abs-sais*
⟨*proof*⟩

theorem *abs-sais-gen-is-Suffix-Array-General*:
Suffix-Array-General sa ⟷ *sa* = *sa-nat-wrapper map-to-nat abs-sais*
⟨*proof*⟩

end
theory *Bucket-Insert*
imports
../../util/Repeat
begin

102 Bucket Insert

fun *bucket-insert-step* ::
nat list × *nat list* × *nat* ⇒
((*'a* :: {*linorder*, *order-bot*}) ⇒ *nat*) × *'a list* × *nat list* ⇒
nat list × *nat list* × *nat*
where
bucket-insert-step (*B*, *SA*, *i*) (*α*, *T*, *LMS*) =
(*let b* = *α* (*T* ! (*LMS* ! *i*));
k = *B* ! *b* − *Suc 0*
in (*B*[*b* := *k*], *SA*[*k* := *LMS* ! *i*], *Suc i*))

definition *bucket-insert-base* ::
((*'a* :: {*linorder*, *order-bot*}) ⇒ *nat*) ⇒ *'a list* ⇒ *nat list* ⇒ *nat list* ⇒ *nat list* ⇒
nat list × *nat list* × *nat*

where
bucket-insert-base α *T B SA LMS* = *repeat* (*length LMS*) *bucket-insert-step* (*B*, *SA*, *0*) (α, *T*, *LMS*)

definition *bucket-insert* ::
(('*a* :: {*linorder*, *order-bot*}) ⇒ *nat*) ⇒ '*a list* ⇒ *nat list* ⇒ *nat list* ⇒ *nat list* ⇒
nat list
where
bucket-insert α *T B SA LMS* =
(*let* (*B'*, *SA'*, *i*) = *bucket-insert-base* α *T B SA LMS*
in SA')

end
theory *Get-Types*
imports
../prop/List-Type
../prop/LMS-List-Slice-Util
../../util/Repeat
begin

103 Suffix Types

```
fun
  get-suffix-types-step-r0 ::
    SL-types list × nat ⇒ 'a :: {linorder, order-bot} list ⇒ SL-types list × nat
where
  get-suffix-types-step-r0 (xs, i) ys =
    (case i of
      0 ⇒ (xs, 0)
      | Suc j ⇒
        (if Suc j < length xs ∧ Suc j < length ys then
          (if ys ! j < ys ! Suc j then
            (xs[j := S-type], j)
           else if ys ! j > ys ! Suc j then
            (xs[j := L-type], j)
           else
            (xs[j := xs ! Suc j], j))
         else
          (xs, j)))
```

definition *get-suffix-types-base*
where
get-suffix-types-base xs ≡
repeat (*length xs* − *Suc 0*) *get-suffix-types-step-r0*
(*replicate* (*length xs*) *S-type*, *length xs* − *Suc 0*) *xs*

definition *get-suffix-types*
where

get-suffix-types xs ≡ *fst* (*get-suffix-types-base xs*)

104 LMS types

fun *is-lms-ref*
  where
is-lms-ref ST 0 = *False* |
is-lms-ref ST (*Suc i*) =
  (*if Suc i* < *length ST then ST* ! *i* = *L-type* ∧ *ST* ! (*Suc i*) = *S-type else False*)

105 Extracting LMS types

abbreviation *extract-lms ST xs* ≡ *filter* (λ*i. is-lms-ref ST i*) *xs*

106 LMS Substrings

definition *find-next-lms* :: *SL-types list* ⇒ *nat* ⇒ *nat*
  where
find-next-lms ST i =
  (*case find* (λ*j. is-lms-ref ST j*) [*Suc i..<length ST*] *of*
    Some j ⇒ *j*
    | - ⇒ *length ST*)

definition
  lms-slice-ref ::
    ('*a* :: {*linorder*, *order-bot*}) *list* ⇒ *SL-types list* ⇒ *nat* ⇒ '*a list*
where
  lms-slice-ref T ST i =
    list-slice T i (*Suc* (*find-next-lms ST i*))

107 Rename Mapping

fun *rename-mapping'* ::
  ('*a* :: {*linorder*, *order-bot*}) *list* ⇒ *SL-types list* ⇒
   nat list ⇒ *nat list* ⇒ *nat* ⇒ *nat list*
where
  rename-mapping' - - [] *names* - = *names* |
  rename-mapping' - - [*x*] *names i* = *names*[*x* := *i*] |
  rename-mapping' T ST (*a* # *b* # *xs*) *names i* =
    (*if lms-slice-ref T ST a* = *lms-slice-ref T ST b*
      then
        rename-mapping' T ST (*b* # *xs*) (*names*[*a* := *i*]) *i*
     else
        rename-mapping' T ST (*b* # *xs*) (*names*[*a* := *i*]) (*Suc i*))

definition
  rename-mapping ::

('a :: {linorder, order-bot}) list ⇒ SL-types list ⇒ nat list ⇒ nat list
where
rename-mapping T ST LMS =
rename-mapping' T ST LMS (replicate (length T) (length T)) 0

end
theory *Induce-L*
imports
../../util/Repeat
../prop/Buckets
begin

108 Induce L Refinement

```
fun induce-l-step-r0 ::
  nat list × nat list × nat ⇒
   (('a :: {linorder, order-bot}) ⇒ nat) × 'a list ⇒
   nat list × nat list × nat
  where
induce-l-step-r0 (B, SA, i) (α, T) =
  (if SA ! i < length T
   then
     (case SA ! i of
        Suc j ⇒
          (case suffix-type T j of
             L-type ⇒
               (let k = α (T ! j);
                    l = B ! k
                in (B[k := Suc l], SA[l := j], Suc i))
             | - ⇒ (B, SA, Suc i))
        | - ⇒ (B, SA, Suc i))
   else (B, SA, Suc i))

fun induce-l-step ::
  nat list × nat list × nat ⇒
   (('a :: {linorder, order-bot}) ⇒ nat) × 'a list × SL-types list⇒
   nat list × nat list × nat
  where
induce-l-step (B, SA, i) (α, T, ST) =
  (if SA ! i < length T
   then
     (case SA ! i of
        Suc j ⇒
          (case ST ! j of
             L-type ⇒
               (let k = α (T ! j);
                    l = B ! k
                in (B[k := Suc (B ! k)], SA[l := j], Suc i))
             | - ⇒ (B, SA, Suc i))
```

| - ⇒ (*B*, *SA*, *Suc i*))
else (*B*, *SA*, *Suc i*))

definition *induce-l-base* ::
((′*a* :: {*linorder*, *order-bot*}) ⇒ *nat*) ⇒
′*a list* ⇒
SL-types list ⇒
nat list ⇒
nat list ⇒
nat list × *nat list* × *nat*
where
induce-l-base α *T ST B SA* = *repeat* (*length T*) *induce-l-step* (*B*, *SA*, *0*) (α, *T*, *ST*)

definition *induce-l* ::
((′*a* :: {*linorder*, *order-bot*}) ⇒ *nat*) ⇒
′*a list* ⇒
SL-types list ⇒
nat list ⇒
nat list ⇒
nat list
where
induce-l α *T ST B SA* = (*let* (*B*′, *SA*′, *i*) = *induce-l-base* α *T ST B SA in SA*′)

end
theory *Induce-S*
imports *../abs−proof/Abs-Induce-S-Verification*
begin

109 Induce S Refinement

fun *induce-s-step-r0* ::
nat list × *nat list* × *nat* ⇒
((′*a* :: {*linorder*, *order-bot*}) ⇒ *nat*) × ′*a list* ⇒
nat list × *nat list* × *nat*
where
induce-s-step-r0 (*B*, *SA*, *i*) (α, *T*) =
(*case i of*
Suc n ⇒
(*if Suc n* < *length SA* ∧ *SA* ! *Suc n* < *length T then*
(*case SA* ! *Suc n of*
Suc j ⇒
(*case suffix-type T j of*
S-type ⇒
(*let b* = α (*T* ! *j*);
k = *B* ! *b* − *Suc 0*
in (*B*[*b* := *k*], *SA*[*k* := *j*], *n*)
)
| - ⇒ (*B*, *SA*, *n*)

```
          )
        | - ⇒ (B, SA, n)
      )
      else
        (B, SA, n)
     )
  | -   ⇒ (B, SA, 0)
)
```

fun *induce-s-step-r1* ::

```
  nat list × nat list × nat ⇒
   (('a :: {linorder, order-bot}) ⇒ nat) × 'a list × SL-types list ⇒
   nat list × nat list × nat
  where
induce-s-step-r1 (B, SA, i) (α, T, ST) =
  (case i of
    Suc n ⇒
      (if Suc n < length SA ∧ SA ! Suc n < length T then
        (case SA ! Suc n of
          Suc j ⇒
            (case ST ! j of
              S-type ⇒
                (let b = α (T ! j);
                     k = B ! b − Suc 0
                 in (B[b := k], SA[k := j], n)
                )
              | - ⇒ (B, SA, n)
            )
          | - ⇒ (B, SA, n)
        )
        else
          (B, SA, n)
      )
    | -   ⇒ (B, SA, 0)
  )
```

fun *induce-s-step-r2* ::

```
  nat list × nat list × nat ⇒
   (('a :: {linorder, order-bot}) ⇒ nat) × 'a list × SL-types list ⇒
   nat list × nat list × nat
  where
induce-s-step-r2 (B, SA, i) (α, T, ST) =
  (case i of
    Suc n ⇒
      (if Suc n < length SA  then
        (case SA ! Suc n of
          Suc j ⇒
            (case ST ! j of
              S-type ⇒
```

```
                (let b = α (T ! j);
                     k = B ! b − Suc 0
                 in (B[b := k], SA[k := j], n)
                )
              | - ⇒ (B, SA, n)
            )
          | - ⇒ (B, SA, n)
        )
        else
          (B, SA, n)
      )
    | -   ⇒ (B, SA, 0)
  )
```

fun *induce-s-step* ::

```
  nat list × nat list × nat ⇒
   (('a :: {linorder, order-bot}) ⇒ nat) × 'a list × SL-types list ⇒
   nat list × nat list × nat
  where
induce-s-step (B, SA, i) (α, T, ST) =
  (case i of
    Suc n ⇒
    (case SA ! Suc n of
        Suc j ⇒
          (case ST ! j of
            S-type ⇒
              (let b = α (T ! j);
                   k = B ! b − Suc 0
               in (B[b := k], SA[k := j], n)
              )
            | - ⇒ (B, SA, n)
          )
        | - ⇒ (B, SA, n)
      )
    | -   ⇒ (B, SA, 0)
  )
```

definition *induce-s-base* ::

```
  (('a :: {linorder, order-bot}) ⇒ nat) ⇒
   'a list ⇒
   SL-types list ⇒
   nat list ⇒
   nat list ⇒
   nat list × nat list × nat
  where
induce-s-base α T ST B SA = repeat (length T − Suc 0) induce-s-step (B, SA,
length T − Suc 0) (α, T, ST)
```

definition *induce-s* ::

(('a :: {linorder, order-bot}) ⇒ nat) ⇒
'a list ⇒
SL-types list ⇒
nat list ⇒
nat list ⇒
nat list
where
induce-s α *T ST B SA* = (*let* (*B'*, *SA'*, *i*) = *induce-s-base* α *T ST B SA in SA'*)

end
theory *Induce*
imports *Induce-S Induce-L Bucket-Insert*
begin

110 Induce

definition *sa-induce-r0* ::
(('a :: {linorder, order-bot}) ⇒ nat) ⇒
'a list ⇒
nat list ⇒
nat list
where
sa-induce-r0 α *T LMS* =
(*let*
B0 = *map* (*bucket-end* α *T*) [*0..<Suc* (α (*Max* (*set T*)))];
B1 = *map* (*bucket-start* α *T*) [*0..<Suc* (α (*Max* (*set T*)))];

— Initialise SA
SA = *replicate* (*length T*) (*length T*);

— Get the suffix types
ST = *abs-get-suffix-types T*;

— Insert the LMS types into the suffix array
SA = *abs-bucket-insert* α *T B0 SA* (*rev LMS*);

— Insert the L types into the suffix array
SA = *induce-l* α *T ST B1 SA*

— Insert the S types into the suffix array
in induce-s α *T ST* (*B0*[*0* := *0*]) *SA*)

definition *sa-induce-r1* ::
(('a :: {linorder, order-bot}) ⇒ nat) ⇒
'a list ⇒
SL-types list ⇒
nat list ⇒
nat list
where

sa-induce-r1 α *T ST LMS* =
(*let*
B0 = *map* (*bucket-end* α *T*) [*0*..<*Suc* (α (*Max* (*set T*)))];
B1 = *map* (*bucket-start* α *T*) [*0*..<*Suc* (α (*Max* (*set T*)))];

— Initialise SA
SA = *replicate* (*length T*) (*length T*);

— Insert the LMS types into the suffix array
SA = *abs-bucket-insert* α *T B0 SA* (*rev LMS*);

— Insert the L types into the suffix array
SA = *induce-l* α *T ST B1 SA*

— Insert the S types into the suffix array
in induce-s α *T ST* (*B0*[*0* := *0*]) *SA*)

definition *sa-induce-r2* ::
((*'a* :: {*linorder*, *order-bot*}) ⇒ *nat*) ⇒
'a list ⇒
SL-types list ⇒
nat list ⇒
nat list
where
sa-induce-r2 α *T ST LMS* =
(*let*
B0 = *map* (*bucket-end* α *T*) [*0*..<*Suc* (α (*Max* (*set T*)))];
B1 = *map* (*bucket-start* α *T*) [*0*..<*Suc* (α (*Max* (*set T*)))];

— Initialise SA
SA = *replicate* (*length T*) (*length T*);

— Insert the LMS types into the suffix array
SA = *bucket-insert* α *T B0 SA* (*rev LMS*);

— Insert the L types into the suffix array
SA = *induce-l* α *T ST B1 SA*

— Insert the S types into the suffix array
in induce-s α *T ST* (*B0*[*0* := *0*]) *SA*)

abbreviation *sa-induce* ≡ *sa-induce-r2*

end
theory *SAIS*
imports *Induce Get-Types*
begin

111 SAIS

function *sais-r0* ::
  nat list ⇒
   nat list
  where
sais-r0 [] = [] |
sais-r0 [*x*] = [*0*] |
sais-r0 (*a* # *b* # *xs*) =
  (*let*
      T = *a* # *b* # *xs*;

      — Compute the suffix types
      ST = *abs-get-suffix-types T*;

      — Extract the LMS types
      LMS0 = *extract-lms ST* [*0*..<*length T*];

      — Induce the prefix ordering based on LMS
      SA = *sa-induce id T ST LMS0*;

      — Extract the LMS types
      LMS1 = *extract-lms ST SA*;

      — Create a new alphabet
      names = *rename-mapping T ST LMS1*;

      — Make a reduced string (2 lines)
      T′ = *rename-string LMS0 names*;

      — Obtain the correct ordering of LMS-types
      LMS2 = (*if distinct T′ then LMS1 else order-lms LMS0* (*sais-r0 T′*))

   — Induce the suffix ordering based of LMS
   in sa-induce id T ST LMS2)
  ⟨*proof*⟩

function *sais-r1* ::
  nat list ⇒
   nat list
  where
sais-r1 [] = [] |
sais-r1 [*x*] = [*0*] |
sais-r1 (*a* # *b* # *xs*) =
  (*let*
      T = *a* # *b* # *xs*;

      — Compute the suffix types
      ST = *get-suffix-types T*;

— Extract the LMS types
LMS0 = *extract-lms ST* [*0..<length T*];

— Induce the prefix ordering based on LMS
SA = *sa-induce id T ST LMS0*;

— Extract the LMS types
LMS1 = *extract-lms ST SA*;

— Create a new alphabet
names = *rename-mapping T ST LMS1*;

— Make a reduced string
T′ = *rename-string LMS0 names*;

— Obtain the correct ordering of LMS-types
LMS2 = (*if distinct T′ then LMS1 else order-lms LMS0* (*sais-r1 T′*))

— Induce the suffix ordering based of LMS
in sa-induce id T ST LMS2)
⟨*proof*⟩

abbreviation *sais* ≡ *sais-r1*

end
theory *Bucket-Insert-Verification*
imports
../abs−proof/Abs-Bucket-Insert-Verification
../def/Bucket-Insert
begin

112 Bucket Insert

lemma *abs-bucket-insert-step-cons*:
assumes *bucket-insert-step* (*B*, *SA*, *Suc i*) (α, *T*, *a* # *xs*) = (*B1*, *SA1*, *j1*)
and *bucket-insert-step* (*B*, *SA*, *i*) (α, *T*, *xs*) = (*B2*, *SA2*, *j2*)
shows *B1* = *B2* ∧ *SA1* = *SA2*
⟨*proof*⟩

lemma *abs-bucket-insert-base-cons′*:
assumes *repeat n bucket-insert-step* (*B*, *SA*, *Suc i*) (α, *T*, *x* # *xs*) = (*B1*, *SA1*, *j1*)
and *repeat n bucket-insert-step* (*B*, *SA*, *i*) (α, *T*, *xs*) = (*B2*, *SA2*, *j2*)
shows *B1* = *B2* ∧ *SA1* = *SA2*
⟨*proof*⟩

lemma *bucket-insert-base-cons*:
assumes *b* = α (*T* ! *a*)

and *k = B ! b − Suc 0*
and *bucket-insert-base α T B SA (a # xs) = (B1, SA1, j1)*
and *bucket-insert-base α T (B[b := k]) (SA[k := a]) xs = (B2, SA2, j2)*
shows *B1 = B2 ∧ SA1 = SA2*
⟨*proof*⟩

lemma *bucket-insert-cons*:
assumes *b = α (T ! a)*
and *k = B ! b − Suc 0*
shows *bucket-insert α T B SA (a # xs) = bucket-insert α T (B[b := k]) (SA[k := a]) xs*
⟨*proof*⟩

lemma *abs-bucket-insert-eq*:
abs-bucket-insert α T B SA xs = bucket-insert α T B SA xs
⟨*proof*⟩

end
theory *Induce-L-Verification*
imports
../abs−proof/Abs-Induce-L-Verification
../def/Induce-L
begin

113 Induce L Refinement

lemma *abs-induce-l-step-to-r0*:
i < length SA ⟹ abs-induce-l-step (B, SA, i) (α, T) = induce-l-step-r0 (B, SA, i) (α, T)
⟨*proof*⟩

lemma *induce-l-step-r0-to*:
⟦*length ST = length T*; ∀ *k < length ST. ST ! k = suffix-type T k*⟧ ⟹
induce-l-step-r0 (B, SA, i) (α, T) = induce-l-step (B, SA, i) (α, T, ST)
⟨*proof*⟩

lemma *abs-induce-l-step-to*:
assumes *i < length SA*
and *length ST = length T*
and ∀ *k < length ST. ST ! k = suffix-type T k*
shows *abs-induce-l-step (B, SA, i) (α, T) = induce-l-step (B, SA, i) (α, T, ST)*
⟨*proof*⟩

lemma *repeat-abs-induce-l-step-to*:
assumes *n ≤ length SA*
and *length ST = length T*
and ∀ *k < length ST. ST ! k = suffix-type T k*
shows *repeat n abs-induce-l-step (B, SA, 0) (α, T) = repeat n induce-l-step (B, SA, 0) (α, T, ST)*

⟨proof⟩

lemma *abs-induce-l-base-to*:
assumes *length SA = length T*
and *length ST = length T*
and ∀ *i < length ST. ST ! i = suffix-type T i*
shows *abs-induce-l-base α T B SA = induce-l-base α T ST B SA*
⟨proof⟩

lemma *abs-induce-l-eq*:
assumes *length SA = length T*
and *length ST = length T*
and ∀ *i < length ST. ST ! i = suffix-type T i*
shows *abs-induce-l α T B SA = induce-l α T ST B SA*
⟨proof⟩

end
theory *Induce-S-Verification*
imports
../abs−proof/Abs-Induce-S-Verification
../def/Induce-S
begin

114 Induce S Refinement

lemma *abs-induce-s-step-to-r0*:
shows *induce-s-step-r0 (B, SA, i) (α, T) = abs-induce-s-step (B, SA, i) (α, T)*
⟨proof⟩

lemma *induce-s-step-r0-to-r1*:
assumes *length ST = length T*
and ∀ *k < length ST. ST ! k = suffix-type T k*
shows *induce-s-step-r1 (B, SA, i) (α, T, ST) = induce-s-step-r0 (B, SA, i) (α, T)*
⟨proof⟩

lemma *abs-induce-s-step-to-r1*:
assumes *length ST = length T*
and ∀ *k < length ST. ST ! k = suffix-type T k*
shows *induce-s-step-r1 (B, SA, i) (α, T, ST) = abs-induce-s-step (B, SA, i) (α, T)*
⟨proof⟩

lemma *induce-s-step-r1-to-r2*:
assumes *s-perm-inv α T B SA0 SA i*
shows *induce-s-step-r2 (B, SA, i) (α, T, ST) = induce-s-step-r1 (B, SA, i) (α, T, ST)*
⟨proof⟩

lemma *abs-induce-s-step-to-r2*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *length ST* = *length T*
and $\forall$ *k* < *length ST*. *ST* ! *k* = *suffix-type T k*
shows *induce-s-step-r2* (*B*, *SA*, *i*) (α, *T*, *ST*) = *abs-induce-s-step* (*B*, *SA*, *i*) (α, *T*)
⟨*proof*⟩

lemma *induce-s-step-r2-to*:
i < *length SA* ⟹ *induce-s-step* (*B*, *SA*, *i*) (α, *T*, *ST*) = *induce-s-step-r2* (*B*, *SA*, *i*) (α, *T*, *ST*)
⟨*proof*⟩

lemma *abs-induce-s-step-to*:
assumes *s-perm-inv* α *T B SA0 SA i*
and *length ST* = *length T*
and $\forall$ *k* < *length ST*. *ST* ! *k* = *suffix-type T k*
and *i* < *length SA*
shows *induce-s-step* (*B*, *SA*, *i*) (α, *T*, *ST*) = *abs-induce-s-step* (*B*, *SA*, *i*) (α, *T*)
⟨*proof*⟩

lemma *abs-induce-s-base-to'*:
assumes *s-perm-inv* α *T B SA0 SA n*
and *length ST* = *length T*
and $\forall$ *k* < *length ST*. *ST* ! *k* = *suffix-type T k*
and *n* < *length SA*
shows *repeat m induce-s-step* (*B*, *SA*, *n*) (α, *T*, *ST*) = *repeat m abs-induce-s-step* (*B*, *SA*, *n*) (α, *T*)
⟨*proof*⟩

lemma *repeat-abs-induce-step-gre-length*:
assumes *length SA* = *length T*
shows
length T ≤ *Suc n* ⟹
repeat (*Suc m*) *abs-induce-s-step* (*B*, *SA*, *Suc n*) (α, *T*)
= *repeat m abs-induce-s-step* (*B*, *SA*, *n*) (α, *T*)
⟨*proof*⟩

lemma *abs-induce-s-base-to*:
assumes *s-perm-pre* α *T B SA* (*length T*)
and *length ST* = *length T*
and $\forall$ *k* < *length ST*. *ST* ! *k* = *suffix-type T k*
shows *induce-s-base* α *T ST B SA* = *abs-induce-s-base* α *T B SA*
⟨*proof*⟩

lemma *abs-induce-s-eq*:
assumes *s-perm-pre* α *T B SA* (*length T*)
and *length ST* = *length T*
and $\forall$ *k* < *length ST*. *ST* ! *k* = *suffix-type T k*

shows *abs-induce-s* α *T B SA* = *induce-s* α *T ST B SA*
⟨*proof*⟩

end
theory *Induce-Verification*
imports
../abs−proof/Abs-Induce-Verification
../def/Induce
Induce-S-Verification Induce-L-Verification Bucket-Insert-Verification
begin

115 Induce

lemma *sa-induce-to-r0*:
assumes *set LMS* = {*i. abs-is-lms T i*}
and *distinct LMS*
and *valid-list T*
and *strict-mono* α
and α *bot* = *0*
and *Suc 0* < *length T*
shows *abs-sa-induce* α *T LMS* = *sa-induce-r0* α *T LMS*
⟨*proof*⟩

definition *sa-induce-r1* ::
((′*a* :: {*linorder*, *order-bot*}) ⇒ *nat*) ⇒
′*a list* ⇒
SL-types list ⇒
nat list ⇒
nat list
where
sa-induce-r1 α *T ST LMS* =
(*let*
B0 = *map* (*bucket-end* α *T*) [*0*..<*Suc* (α (*Max* (*set T*)))];
B1 = *map* (*bucket-start* α *T*) [*0*..<*Suc* (α (*Max* (*set T*)))];

— Initialise SA
SA = *replicate* (*length T*) (*length T*);

— Insert the LMS types into the suffix array
SA = *abs-bucket-insert* α *T B0 SA* (*rev LMS*);

— Insert the L types into the suffix array
SA = *induce-l* α *T ST B1 SA*

— Insert the S types into the suffix array
in induce-s α *T ST* (*B0*[*0* := *0*]) *SA*)

lemma *sa-induce-r0-to-r1*:
assumes *length ST* = *length T*

and $\forall\, i < length\ ST.\ ST\ !\ i = suffix\text{-}type\ T\ i$
shows *sa-induce-r0* α *T LMS* = *sa-induce-r1* α *T ST LMS*
⟨*proof*⟩

lemma *sa-induce-to-r1*:
assumes *set LMS* = {*i. abs-is-lms T i*}
and *distinct LMS*
and *valid-list T*
and *strict-mono* α
and α *bot* = *0*
and *Suc 0* < *length T*
and *length ST* = *length T*
and $\forall\, i < length\ ST.\ ST\ !\ i = suffix\text{-}type\ T\ i$
shows *abs-sa-induce* α *T LMS* = *sa-induce-r1* α *T ST LMS*
⟨*proof*⟩

lemma *sa-induce-r1-to-r2*:
sa-induce-r1 α *T ST LMS* = *sa-induce-r2* α *T ST LMS*
⟨*proof*⟩

lemma *abs-sa-induce-to-r2*:
assumes *set LMS* = {*i. abs-is-lms T i*}
and *distinct LMS*
and *valid-list T*
and *strict-mono* α
and α *bot* = *0*
and *Suc 0* < *length T*
and *length ST* = *length T*
and $\forall\, i < length\ ST.\ ST\ !\ i = suffix\text{-}type\ T\ i$
shows *abs-sa-induce* α *T LMS* = *sa-induce-r2* α *T ST LMS*
⟨*proof*⟩

end
theory *Get-Types-Verification*
imports
../abs−def/Abs-SAIS
../def/Get-Types
begin

116 Suffix Types

lemma *get-suffix-types-step-r0-ret*:
$\exists\, xs'\ i'.\ get\text{-}suffix\text{-}types\text{-}step\text{-}r0\ (xs,\ i)\ ys = (xs',\ i') \land$
$length\ xs' = length\ xs \land (i = 0 \longrightarrow i' = 0) \land (\exists\, j.\ i = Suc\ j \longrightarrow i' = j)$
⟨*proof*⟩

lemma *get-suffix-types-step-r0-0*:
get-suffix-types-step-r0 (*xs*, *0*) *ys* = (*xs*, *0*)
⟨*proof*⟩

lemma *get-suffix-types-step-r0-Suc*:
$[\![$*Suc i* $<$ *length xs*; *length xs* $=$ *length ys*; $\forall$ *k* $<$ *length xs*. *i* $<$ *k* $\longrightarrow$ *xs* ! *k* $=$ *suffix-type ys k*$]\!] \Longrightarrow$
get-suffix-types-step-r0 (*xs*, *Suc i*) *ys* $=$ (*xs*[*i* := *suffix-type ys i*], *i*)
⟨*proof*⟩

fun *get-suffix-types-inv*
where
get-suffix-types-inv ys (*xs*, *i*) $=$
(*length xs* $=$ *length ys* $\wedge$ *i* $<$ *length xs* $\wedge$ ($\forall$ *k* $<$ *length xs*. *i* $\leq$ *k* $\longrightarrow$ *xs* ! *k* $=$ *suffix-type ys k*))

lemma *get-suffix-types-inv-maintained*:
assumes *get-suffix-types-inv ys* (*xs*, *i*)
shows *get-suffix-types-inv ys* (*get-suffix-types-step-r0* (*xs*, *i*) *ys*)
⟨*proof*⟩

lemma *get-suffix-types-inv-established*:
xs $\neq$ [] $\Longrightarrow$ *get-suffix-types-inv xs* (*replicate* (*length xs*) *S-type*, *length xs* $-$ *Suc 0*)
⟨*proof*⟩

lemma *get-suffix-types-base-prod'*:
$\exists$ *xs'*. *repeat n get-suffix-types-step-r0* (*xs*, *m*) *ys* $=$ (*xs'*, *m* $-$ *n*)
⟨*proof*⟩

lemma *get-suffix-types-inv-holds*:
assumes *xs* $\neq$ []
shows *get-suffix-types-inv xs* (*get-suffix-types-base xs*)
⟨*proof*⟩

lemma *get-suffix-types-base-prod*:
$\exists$ *xs'*. *get-suffix-types-base xs* $=$ (*xs'*, *0*)
⟨*proof*⟩

lemma *get-suffix-types-base-ref*:
get-suffix-types-base xs $=$ (*abs-get-suffix-types xs*, *0*)
⟨*proof*⟩

lemma *get-suffix-types-eq*:
get-suffix-types xs $=$ *abs-get-suffix-types xs*
⟨*proof*⟩

lemmas *length-get-suffix-types* $=$
length-abs-get-suffix-types[*simplified get-suffix-types-eq*]

117 LMS types

lemma *is-lms-refinement*:
assumes *length ST* = *length T* $\forall$ *i* < *length T*. *ST* ! *i* = *suffix-type T i*
shows *is-lms-ref ST* = *abs-is-lms T*
⟨*proof*⟩

118 Extracting LMS types

lemma *extract-lms-eq*:
⟦*length ST* = *length T*; $\forall$ *i* < *length T*. *ST* ! *i* = *suffix-type T i*⟧ $\Longrightarrow$
extract-lms ST = *abs-extract-lms T*
⟨*proof*⟩

119 LMS Substrings

lemma *find-next-lms-refinement*:
⟦*length ST* = *length T*; $\forall$ *i* < *length T*. *ST* ! *i* = *suffix-type T i*⟧ $\Longrightarrow$
find-next-lms ST= *abs-find-next-lms T*
⟨*proof*⟩

lemma *lms-slice-refinement*:
⟦*length ST* = *length T*; $\forall$ *i* < *length T*. *ST* ! *i* = *suffix-type T i*⟧ $\Longrightarrow$
lms-slice-ref T ST = *lms-slice T*
⟨*proof*⟩

120 Rename Mapping

lemma *rename-mapping'-refinement*:
assumes *length ST* = *length T* $\forall$ *i* < *length T*. *ST* ! *i* = *suffix-type T i*
shows *rename-mapping' T ST* = *abs-rename-mapping' T*
⟨*proof*⟩

lemma *rename-mapping-refinement*:
assumes *length ST* = *length T*
assumes $\forall$ *i* < *length T*. *ST* ! *i* = *suffix-type T i*
shows *rename-mapping T ST* = *abs-rename-mapping T*
⟨*proof*⟩

end
theory *SAIS-Verification*
imports
Get-Types-Verification
Induce-Verification
../abs−proof/Abs-SAIS-Verification-With-Valid-Precondition
../def/SAIS

begin

121 SAIS

termination *sais-r0*
⟨*proof*⟩

lemma *abs-sais-r0-distinct-simp*:
assumes $T = a \# b \# xs$
and $ST = abs\text{-}get\text{-}suffix\text{-}types\ T$
and $LMS0 = extract\text{-}lms\ ST\ [0..<length\ T]$
and $SA = sa\text{-}induce\ id\ T\ ST\ LMS0$
and $LMS = extract\text{-}lms\ ST\ SA$
and $names = rename\text{-}mapping\ T\ ST\ LMS$
and $T' = rename\text{-}string\ LMS0\ names$
and $distinct\ T'$
shows $sais\text{-}r0\ T = sa\text{-}induce\ id\ T\ ST\ LMS$
⟨*proof*⟩

lemma *abs-sais-r0-not-distinct-simp*:
assumes $T = a \# b \# xs$
and $ST = abs\text{-}get\text{-}suffix\text{-}types\ T$
and $LMS0 = extract\text{-}lms\ ST\ [0..<length\ T]$
and $SA = sa\text{-}induce\ id\ T\ ST\ LMS0$
and $LMS = extract\text{-}lms\ ST\ SA$
and $names = rename\text{-}mapping\ T\ ST\ LMS$
and $T' = rename\text{-}string\ LMS0\ names$
and $LMS1 = order\text{-}lms\ LMS0\ (sais\text{-}r0\ T')$
and $\neg distinct\ T'$
shows $sais\text{-}r0\ T = sa\text{-}induce\ id\ T\ ST\ LMS1$
⟨*proof*⟩

lemma *abs-sais-to-r0*:
$valid\text{-}list\ T \Longrightarrow abs\text{-}sais\ T = sais\text{-}r0\ T$
⟨*proof*⟩

termination *sais-r1*
⟨*proof*⟩

lemma *abs-sais-r0-to-r1*:
$sais\text{-}r1\ T = sais\text{-}r0\ T$
⟨*proof*⟩

lemma *abs-sais-to-r1*:
$valid\text{-}list\ T \Longrightarrow sais\text{-}r1\ T = abs\text{-}sais\ T$
⟨*proof*⟩

122 Correctness

interpretation *sais*: *Suffix-Array-Restricted sais*
⟨*proof*⟩

interpretation *abs-sais-ref-gen*: *Suffix-Array-General sa-nat-wrapper map-to-nat sais*
⟨*proof*⟩

theorem *sais-gen-is-Suffix-Array-General*:
Suffix-Array-General sa ⟷ *sa* = *sa-nat-wrapper map-to-nat sais*
⟨*proof*⟩

end
theory *Code-Extraction*
imports *../abs−proof/Abs-SAIS-Verification*
../proof/SAIS-Verification
begin

lemma [*code*]:
abs-is-lms T i =
(*if i* > *0 then*
if suffix-type T i = *S-type* ∧ *suffix-type T* (*i* − *1*) = *L-type*
then True
else False
else False)
⟨*proof*⟩

definition
bucket-upt-code :: (′*a* :: {*linorder*,*order-bot*} ⇒ *nat*) ⇒ ′*a list* ⇒ *nat* ⇒ *nat set*
where
bucket-upt-code α *T b* ≡
set (*filter* (λ*x*. α (*T* ! *x*) < *b*) [*0*..<*length T*])

lemma [*code*]:
bucket-upt α *T b* = *bucket-upt-code* α *T b*
⟨*proof*⟩

export-code *abs-sais* **in** *Haskell*
module-name *SAIS* **file-prefix** *abs-sais*

export-code *sais* **in** *Haskell*
module-name *SAIS-REF* **file-prefix** *sais*

end
theory *SACA-Equiv*
imports *sais/abs−proof/Abs-SAIS-Verification*
simple/Simple-SACA-Verification
sais/proof/SAIS-Verification
begin

lemma *Suffix-Array-General-imp-suffix-array*:

Suffix-Array-General sa $\Longrightarrow$
sa s = *simple-saca s*
⟨*proof*⟩

theorem *Suffix-Array-General-equiv-spec*:
Suffix-Array-General sa $\longleftrightarrow$
sa = *simple-saca*
⟨*proof*⟩

corollary *abs-sais-equiv-simple-saca*:
sa-nat-wrapper map-to-nat abs-sais = *simple-saca*
⟨*proof*⟩

corollary *sais-equiv-simple-saca*:
sa-nat-wrapper map-to-nat sais = *simple-saca*
⟨*proof*⟩

end

References

[1] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

[2] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[3] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *Proc. Data Compression Conference*, pages 193–202. IEEE Computing Society, 2009.