Formalized Burrows-Wheeler Transform

Louis Cheung and Christine Rizkallah

January 17, 2025

Abstract

The Burrows-Wheeler transform (BWT) [2] is an invertible lossless transformation that permutes input sequences into alternate sequences of the same length that frequently contain long localized regions that involve clusters consisting of just a few distinct symbols, and sometimes also include long runs of same-symbol repetitions. Moreover, there is a one-to-one correspondence between the BWT and suffix arrays [7]. As a consequence, the BWT is widely used in data compression and as an indexing data structure for pattern search. In this formalization [4], we present the formal verification of both the BWT and its inverse, building on a formalization of suffix arrays [5]. This is the artefact of our CPP paper [3].

Contents

theory *Nat-Mod-Helper*
imports *Main*
begin

1 Nat Modulo Helper

lemma *nat-mod-add-neq-self*:
$[\![a < (n :: nat);\ b < n;\ b \neq 0]\!] \Longrightarrow (a + b)\ mod\ n \neq a$
⟨*proof*⟩

lemma *nat-mod-a-pl-b-eq1*:
$[\![n + b \leq a;\ a < (n :: nat)]\!] \Longrightarrow (a + b)\ mod\ n = b - (n - a)$
⟨*proof*⟩

lemma *not-mod-a-pl-b-eq2*:
$[\![n - a \leq b;\ a < n;\ b < (n :: nat)]\!] \Longrightarrow (a + b)\ mod\ n = b - (n - a)$
⟨*proof*⟩

end
theory *Rotated-Substring*
imports *Nat-Mod-Helper*
begin

2 Rotated Sublists

definition *is-sublist* :: $'a\ list \Rightarrow\ 'a\ list \Rightarrow bool$
where
$is\text{-}sublist\ xs\ ys = (\exists\, as\ bs.\ xs = as\ @\ ys\ @\ bs)$

definition *is-rot-sublist* :: $'a\ list \Rightarrow\ 'a\ list \Rightarrow bool$
where
$is\text{-}rot\text{-}sublist\ xs\ ys = (\exists\, n.\ is\text{-}sublist\ (rotate\ n\ xs)\ ys)$

definition *inc-one-bounded* :: $nat \Rightarrow nat\ list \Rightarrow bool$
where
$inc\text{-}one\text{-}bounded\ n\ xs \equiv$
$(\forall\, i.\ Suc\ i < length\ xs \longrightarrow xs\ !\ Suc\ i = Suc\ (xs\ !\ i)\ mod\ n) \land$
$(\forall\, i < length\ xs.\ xs\ !\ i < n)$

lemma *inc-one-boundedD*:

$[\![inc\text{-}one\text{-}bounded\ n\ xs;\ Suc\ i < length\ xs]\!] \Longrightarrow xs\ !\ Suc\ i = Suc\ (xs\ !\ i)\ mod\ n$
$[\![inc\text{-}one\text{-}bounded\ n\ xs;\ i < length\ xs]\!] \Longrightarrow xs\ !\ i < n$
⟨*proof*⟩

lemma *inc-one-bounded-nth-plus*:
$[\![inc\text{-}one\text{-}bounded\ n\ xs;\ i + k < length\ xs]\!] \Longrightarrow xs\ !\ (i + k) = (xs\ !\ i + k)\ mod\ n$
⟨*proof*⟩

lemma *inc-one-bounded-neq*:
$[\![inc\text{-}one\text{-}bounded\ n\ xs;\ length\ xs \le n;\ i + k < length\ xs;\ k \ne 0]\!] \Longrightarrow xs\ !\ (i + k) \ne xs\ !\ i$
⟨*proof*⟩

corollary *inc-one-bounded-neq-nth*:
assumes *inc-one-bounded n xs*
and $length\ xs \le n$
and $i < length\ xs$
and $j < length\ xs$
and $i \ne j$
shows $xs\ !\ i \ne xs\ !\ j$
⟨*proof*⟩

lemma *inc-one-bounded-distinct*:
$[\![inc\text{-}one\text{-}bounded\ n\ xs;\ length\ xs \le n]\!] \Longrightarrow distinct\ xs$
⟨*proof*⟩

lemma *inc-one-bounded-subset-upt*:
$[\![inc\text{-}one\text{-}bounded\ n\ xs;\ length\ xs \le n]\!] \Longrightarrow set\ xs \subseteq \{0..<n\}$
⟨*proof*⟩

lemma *inc-one-bounded-consD*:
$inc\text{-}one\text{-}bounded\ n\ (x\ \#\ xs) \Longrightarrow inc\text{-}one\text{-}bounded\ n\ xs$
⟨*proof*⟩

lemma *inc-one-bounded-nth*:
$[\![inc\text{-}one\text{-}bounded\ n\ xs;\ i < length\ xs]\!] \Longrightarrow xs\ !\ i = ((\lambda x.\ Suc\ x\ mod\ n)^{\wedge\wedge}i)\ (xs\ !\ 0)$
⟨*proof*⟩

lemma *inc-one-bounded-nth-le*:
$[\![inc\text{-}one\text{-}bounded\ n\ xs;\ i < length\ xs;\ (xs\ !\ 0) + i < n]\!] \Longrightarrow$
$xs\ !\ i = (xs\ !\ 0) + i$
⟨*proof*⟩

lemma *inc-one-bounded-upt1*:
assumes *inc-one-bounded n xs*
and $length\ xs = Suc\ k$
and $Suc\ k \le n$
and $(xs\ !\ 0) + k < n$

shows $xs = [xs\ !\ 0..<(xs\ !\ 0) + Suc\ k]$
⟨*proof*⟩

lemma *inc-one-bounded-upt2*:
assumes *inc-one-bounded n xs*
and $length\ xs = Suc\ k$
and $Suc\ k \leq n$
and $n \leq (xs\ !\ 0) + k$
shows $xs = [xs\ !\ 0..<n]\ @\ [0..<(xs\ !\ 0) + Suc\ k - n]$
⟨*proof*⟩

lemmas *inc-one-bounded-upt* = *inc-one-bounded-upt1 inc-one-bounded-upt2*

lemma *is-rot-sublist-nil*:
is-rot-sublist xs []
⟨*proof*⟩

lemma *rotate-upt*:
$m \leq n \Longrightarrow rotate\ m\ [0..<n] = [m..<n]\ @\ [0..<m]$
⟨*proof*⟩

lemma *inc-one-bounded-is-rot-sublist*:
assumes *inc-one-bounded n xs* $length\ xs \leq n$
shows *is-rot-sublist* $[0..<n]$ *xs*
⟨*proof*⟩

lemma *is-rot-sublist-idx*:
is-rot-sublist $[0..<length\ xs]$ $ys \Longrightarrow$ *is-rot-sublist xs* (*map* ((!) *xs*) *ys*)
⟨*proof*⟩

lemma *is-rot-sublist-upt-eq-upt-hd*:
$[\![$*is-rot-sublist* $[0..<Suc\ n]$ *ys*; $length\ ys = Suc\ n$; $ys\ !\ 0 = 0]\!] \Longrightarrow ys = [0..<Suc\ n]$
⟨*proof*⟩

lemma *is-rot-sublist-upt-eq-upt-last*:
$[\![$*is-rot-sublist* $[0..<Suc\ n]$ *ys*; $length\ ys = Suc\ n$; $ys\ !\ n = n]\!] \Longrightarrow ys = [0..<Suc\ n]$
⟨*proof*⟩

end
theory *Count-Util*
imports *HOL−Library.Multiset*
HOL−Combinatorics.List-Permutation
SuffixArray.List-Util
SuffixArray.List-Slice
begin

3 Counting

3.1 Count List

lemma *count-in*:
$x \in set\ xs \Longrightarrow count\text{-}list\ xs\ x > 0$
⟨*proof*⟩

lemma *in-count*:
$count\text{-}list\ xs\ x > 0 \Longrightarrow x \in set\ xs$
⟨*proof*⟩

lemma *notin-count*:
$count\text{-}list\ xs\ x = 0 \Longrightarrow x \notin set\ xs$
⟨*proof*⟩

lemma *count-list-eq-count*:
$count\text{-}list\ xs\ x = count\ (mset\ xs)\ x$
⟨*proof*⟩

lemma *count-list-perm*:
$xs <^{\sim\sim}> ys \Longrightarrow count\text{-}list\ xs\ x = count\text{-}list\ ys\ x$
⟨*proof*⟩

lemma *in-count-nth-ex*:
$count\text{-}list\ xs\ x > 0 \Longrightarrow \exists\, i < length\ xs.\ xs\ !\ i = x$
⟨*proof*⟩

lemma *in-count-list-slice-nth-ex*:
$count\text{-}list\ (list\text{-}slice\ xs\ i\ j)\ x > 0 \Longrightarrow \exists\, k < length\ xs.\ i \leq k \wedge k < j \wedge xs\ !\ k = x$
⟨*proof*⟩

3.2 Cardinality

lemma *count-list-card*:
$count\text{-}list\ xs\ x = card\ \{j.\ j < length\ xs \wedge xs\ !\ j = x\}$
⟨*proof*⟩

lemma *card-le-eq-card-less-pl-count-list*:
fixes $s :: 'a :: linorder\ list$
shows $card\ \{k.\ k < length\ s \wedge s\ !\ k \leq a\} = card\ \{k.\ k < length\ s \wedge s\ !\ k < a\}$
$+\ count\text{-}list\ s\ a$
⟨*proof*⟩

lemma *card-less-idx-upper-strict*:
fixes $s :: 'a :: linorder\ list$
assumes $a \in set\ s$
shows $card\ \{k.\ k < length\ s \wedge s\ !\ k < a\} < length\ s$
⟨*proof*⟩

lemma *card-less-idx-upper*:
  shows $card\ \{k.\ k < length\ s \wedge s\ !\ k < a\} \leq length\ s$
  ⟨*proof*⟩

lemma *card-pl-count-list-strict-upper*:
  fixes $s :: 'a :: linorder\ list$
  shows $card\ \{i.\ i < length\ s \wedge s\ !\ i < a\} + count\text{-}list\ s\ a \leq length\ s$
⟨*proof*⟩

3.3 Sorting

lemma *sorted-nth-le*:
  assumes $sorted\ xs$
  and $card\ \{k.\ k < length\ xs \wedge xs\ !\ k < c\} < length\ xs$
shows $c \leq xs\ !\ card\ \{k.\ k < length\ xs \wedge xs\ !\ k < c\}$
  ⟨*proof*⟩

lemma *sorted-nth-le-gen*:
  assumes $sorted\ xs$
  and $card\ \{k.\ k < length\ xs \wedge xs\ !\ k < c\} + i < length\ xs$
shows $c \leq xs\ !\ (card\ \{k.\ k < length\ xs \wedge xs\ !\ k < c\} + i)$
⟨*proof*⟩

lemma *sorted-nth-less-gen*:
  assumes $sorted\ xs$
  and $i < card\ \{k.\ k < length\ xs \wedge xs\ !\ k < c\}$
shows $xs\ !\ i < c$
⟨*proof*⟩

lemma *sorted-nth-gr-gen*:
  assumes $sorted\ xs$
  and $card\ \{k.\ k < length\ xs \wedge xs\ !\ k < c\} + i < length\ xs$
  and $count\text{-}list\ xs\ c \leq i$
shows $xs\ !\ (card\ \{k.\ k < length\ xs \wedge xs\ !\ k < c\} + i) > c$
⟨*proof*⟩

end
theory *Rank-Util*
  imports *HOL−Library.Multiset*
    Count-Util
    SuffixArray.Prefix
begin

4 Rank Definition

Count how many occurrences of an element are in a certain index in the list

Definition 3.7 from [3]: Rank

definition $rank :: 'a\ list \Rightarrow 'a \Rightarrow nat \Rightarrow nat$

where
$rank\ s\ x\ i \equiv count\text{-}list\ (take\ i\ s)\ x$

5 Rank Properties

5.1 List Properties

lemma *rank-cons-same*:
$rank\ (x\ \#\ xs)\ x\ (Suc\ i) = Suc\ (rank\ xs\ x\ i)$
$\langle proof \rangle$

lemma *rank-cons-diff*:
$a \neq x \Longrightarrow rank\ (a\ \#\ xs)\ x\ (Suc\ i) = rank\ xs\ x\ i$
$\langle proof \rangle$

5.2 Counting Properties

lemma *rank-length*:
$rank\ xs\ x\ (length\ xs) = count\text{-}list\ xs\ x$
$\langle proof \rangle$

lemma *rank-gre-length*:
$length\ xs \leq n \Longrightarrow rank\ xs\ x\ n = count\text{-}list\ xs\ x$
$\langle proof \rangle$

lemma *rank-not-in*:
$x \notin set\ xs \Longrightarrow rank\ xs\ x\ i = 0$
$\langle proof \rangle$

lemma *rank-0*:
$rank\ xs\ x\ 0 = 0$
$\langle proof \rangle$

Theorem 3.11 from [3]: Rank Equivalence

lemma *rank-card-spec*:
$rank\ xs\ x\ i = card\ \{j.\ j < length\ xs \land j < i \land xs\ !\ j = x\}$
$\langle proof \rangle$

lemma *le-rank-plus-card*:
$i \leq j \Longrightarrow$
$rank\ xs\ x\ j = rank\ xs\ x\ i + card\ \{k.\ k < length\ xs \land i \leq k \land k < j \land xs\ !\ k = x\}$
$\langle proof \rangle$

5.3 Bound Properties

lemma *rank-lower-bound*:
assumes $k < rank\ xs\ x\ i$

shows $k < i$
⟨*proof*⟩

corollary *rank-Suc-ex*:
assumes $k < rank\ xs\ x\ i$
shows $\exists\, l.\ i = Suc\ l$
⟨*proof*⟩

lemma *rank-upper-bound*:
$[\![i < length\ xs;\ xs\ !\ i = x]\!] \Longrightarrow rank\ xs\ x\ i < count\text{-}list\ xs\ x$
⟨*proof*⟩

lemma *rank-idx-mono*:
$i \leq j \Longrightarrow rank\ xs\ x\ i \leq rank\ xs\ x\ j$
⟨*proof*⟩

lemma *rank-less*:
$[\![i < length\ xs;\ i < j;\ xs\ !\ i = x]\!] \Longrightarrow rank\ xs\ x\ i < rank\ xs\ x\ j$
⟨*proof*⟩

lemma *rank-upper-bound-gen*:
$rank\ xs\ x\ i \leq count\text{-}list\ xs\ x$
⟨*proof*⟩

5.4 Sorted Properties

lemma *sorted-card-rank-idx*:
assumes $sorted\ xs$
and $i < length\ xs$
shows $i = card\ \{j.\ j < length\ xs \wedge xs\ !\ j < xs\ !\ i\} + rank\ xs\ (xs\ !\ i)\ i$
⟨*proof*⟩

lemma *sorted-rank*:
assumes $sorted\ xs$
and $i < length\ xs$
and $xs\ !\ i = a$
shows $rank\ xs\ a\ i = i - card\ \{k.\ k < length\ xs \wedge xs\ !\ k < a\}$
⟨*proof*⟩

lemma *sorted-rank-less*:
assumes $sorted\ xs$
and $i < length\ xs$
and $xs\ !\ i < a$
shows $rank\ xs\ a\ i = 0$
⟨*proof*⟩

lemma *sorted-rank-greater*:
assumes $sorted\ xs$
and $i < length\ xs$

and $xs ! i > a$
shows $rank\ xs\ a\ i = count\text{-}list\ xs\ a$
⟨*proof*⟩

end
theory *Select-Util*
imports *Count-Util*
SuffixArray.Sorting-Util
begin

6 Select Definition

Find nth occurrence of an element in a list

Definition 3.8 from [3]: Select

fun $select :: 'a\ list \Rightarrow 'a \Rightarrow nat \Rightarrow nat$
where
$select\ []\ \text{-}\ \text{-} = 0\ |$
$select\ (a\#xs)\ x\ 0 = (if\ x = a\ then\ 0\ else\ Suc\ (select\ xs\ x\ 0))\ |$
$select\ (a\#xs)\ x\ (Suc\ i) = (if\ x = a\ then\ Suc\ (select\ xs\ x\ i)\ else\ Suc\ (select\ xs\ x\ (Suc\ i)))$

7 Select Properties

7.1 Length Properties

lemma *notin-imp-select-length*:
$x \notin set\ xs \Longrightarrow select\ xs\ x\ i = length\ xs$
⟨*proof*⟩

lemma *select-length-imp-count-list-less*:
$select\ xs\ x\ i = length\ xs \Longrightarrow count\text{-}list\ xs\ x \leq i$
⟨*proof*⟩

lemma *select-Suc-length*:
$select\ xs\ x\ i = length\ xs \Longrightarrow select\ xs\ x\ (Suc\ i) = length\ xs$
⟨*proof*⟩

7.2 List Properties

lemma *select-cons-neq*:
$[\![select\ xs\ x\ i = j;\ x \neq a]\!] \Longrightarrow select\ (a\ \#\ xs)\ x\ i = Suc\ j$
⟨*proof*⟩

lemma *cons-neq-select*:
$[\![select\ (a\ \#\ xs)\ x\ i = Suc\ j;\ x \neq a]\!] \Longrightarrow select\ xs\ x\ i = j$
⟨*proof*⟩

lemma *cons-eq-select*:
$select\ (x\ \#\ xs)\ x\ (Suc\ i) = Suc\ j \Longrightarrow select\ xs\ x\ i = j$
$\langle proof \rangle$

lemma *select-cons-eq*:
$select\ xs\ x\ i = j \Longrightarrow select\ (x\ \#\ xs)\ x\ (Suc\ i) = Suc\ j$
$\langle proof \rangle$

7.3 Bound Properties

lemma *select-max*:
$select\ xs\ x\ i \leq length\ xs$
$\langle proof \rangle$

7.4 Nth Properties

lemma *nth-select*:
$[\![j < length\ xs;\ count\text{-}list\ (take\ (Suc\ j)\ xs)\ x = Suc\ i;\ xs\ !\ j = x]\!]$
$\Longrightarrow select\ xs\ x\ i = j$
$\langle proof \rangle$

lemma *nth-select-alt*:
$[\![j < length\ xs;\ count\text{-}list\ (take\ j\ xs)\ x = i;\ xs\ !\ j = x]\!]$
$\Longrightarrow select\ xs\ x\ i = j$
$\langle proof \rangle$

lemma *select-nth*:
$[\![select\ xs\ x\ i = j;\ j < length\ xs]\!]$
$\Longrightarrow count\text{-}list\ (take\ (Suc\ j)\ xs)\ x = Suc\ i \land xs\ !\ j = x$
$\langle proof \rangle$

lemma *select-nth-alt*:
$[\![select\ xs\ x\ i = j;\ j < length\ xs]\!]$
$\Longrightarrow count\text{-}list\ (take\ j\ xs)\ x = i \land xs\ !\ j = x$
$\langle proof \rangle$

lemma *select-less-0-nth*:
assumes $i < length\ xs$
and $i < select\ xs\ x\ 0$
shows $xs\ !\ i \neq x$
$\langle proof \rangle$

7.5 Sorted Properties

Theorem 3.10 from [3]: Select Sorted Equivalence

lemma *sorted-select*:
assumes $sorted\ xs$
and $i < count\text{-}list\ xs\ x$
shows $select\ xs\ x\ i = card\ \{j.\ j < length\ xs \land xs\ !\ j < x\} + i$

⟨*proof*⟩

corollary *sorted-select-0-plus*:
assumes *sorted xs*
and $i < \textit{count-list } xs\ x$
shows $\textit{select } xs\ x\ i = \textit{select } xs\ x\ 0 + i$
⟨*proof*⟩

corollary *select-sorted-0*:
assumes *sorted xs*
and $0 < \textit{count-list } xs\ x$
shows $\textit{select } xs\ x\ 0 = \textit{card } \{j.\ j < \textit{length } xs \wedge xs\ !\ j < x\}$
⟨*proof*⟩

end
theory *Rank-Select*
imports *Main*
Rank-Util
Select-Util
begin

8 Rank and Select Properties

8.1 Correctness of Rank and Select

Correctness theorem statements based on [1].

8.1.1 Rank Correctness

lemma *rank-spec*:
$\textit{rank } s\ x\ i = \textit{count } (\textit{mset } (\textit{take } i\ s))\ x$
⟨*proof*⟩

8.1.2 Select Correctness

lemma *select-spec*:
$\textit{select } s\ x\ i = j$
$\Longrightarrow (j < \textit{length } s \wedge \textit{rank } s\ x\ j = i) \vee (j = \textit{length } s \wedge \textit{count-list } s\ x \leq i\ )$
⟨*proof*⟩

Theorem 3.9 from [3]: Correctness of Select

lemma *select-correct*:
$\textit{select } s\ x\ i \leq \textit{length } s\ \wedge$
$(\textit{select } s\ x\ i < \textit{length } s \longrightarrow \textit{rank } s\ x\ (\textit{select } s\ x\ i) = i) \wedge$
$(\textit{select } s\ x\ i = \textit{length } s \longrightarrow \textit{count-list } s\ x \leq i)$
⟨*proof*⟩

8.2 Rank and Select

lemma *rank-select*:
  $select\ xs\ x\ i < length\ xs \Longrightarrow rank\ xs\ x\ (select\ xs\ x\ i) = i$
⟨*proof*⟩

lemma *select-upper-bound*:
  $i < rank\ xs\ x\ j \Longrightarrow select\ xs\ x\ i < length\ xs$
⟨*proof*⟩

lemma *select-out-of-range*:
  assumes $count\text{-}list\ xs\ a \le i$
  and $mset\ xs = mset\ ys$
shows $select\ ys\ a\ i = length\ ys$
  ⟨*proof*⟩

8.3 Sorted Properties

lemma *sorted-nth-gen*:
  assumes *sorted xs*
  and $card\ \{k.\ k < length\ xs \land xs\ !\ k < c\} < length\ xs$
  and $count\text{-}list\ xs\ c > i$
shows $xs\ !\ (card\ \{k.\ k < length\ xs \land xs\ !\ k < c\} + i) = c$
⟨*proof*⟩

lemma *sorted-nth-gen-alt*:
  assumes *sorted xs*
  and $card\ \{k.\ k < length\ xs \land xs\ !\ k < a\} \le i$
  and $i < card\ \{k.\ k < length\ xs \land xs\ !\ k < a\} + card\ \{k.\ k < length\ xs \land xs$
$!\ k = a\}$
shows $xs\ !\ i = a$
⟨*proof*⟩

end
theory *SA-Util*
  imports *SuffixArray.Suffix-Array-Properties*
    SuffixArray.Simple-SACA-Verification
    ../counting/Rank-Select
begin

9 Suffix Array Properties

9.1 Bijections

lemma *bij-betw-empty*:
  bij-betw f {} {}
  ⟨*proof*⟩

lemma *bij-betw-sort-idx-ex*:
  assumes $xs = sort\ ys$

shows $\exists f.$ *bij-betw f* $\{j.\ j <$ *length ys* $\wedge$ *ys* ! $j < x\}$ $\{j.\ j <$ *length xs* $\wedge$ *xs* ! $j < x\}$
⟨*proof*⟩

9.2 Suffix Properties

lemma *suffix-hd-set-eq*:
$\{k.\ k <$ *length s* $\wedge$ *s* ! $k = c\ \} = \{k.\ k <$ *length s* $\wedge$ $(\exists$ *xs. suffix s k* $= c$ # *xs*$)\}$
⟨*proof*⟩

lemma *suffix-hd-set-less*:
$\{k.\ k <$ *length s* $\wedge$ *s* ! $k < c\ \} = \{k.\ k <$ *length s* $\wedge$ *suffix s k* $< [c]\}$
⟨*proof*⟩

lemma *select-nth-suffix-start1*:
assumes $i <$ *card* $\{k.\ k <$ *length s* $\wedge$ $(\exists$ *as. suffix s k* $= a$ # *as*$)\}$
and *xs* = *sort s*
shows *select xs a i* = *card* $\{k.\ k <$ *length s* $\wedge$ *suffix s k* $< [a]\} + i$
⟨*proof*⟩

lemma *select-nth-suffix-start2*:
assumes *card* $\{k.\ k <$ *length s* $\wedge$ $(\exists$ *as. suffix s k* $= a$ # *as*$)\} \leq i$
and *xs* = *sort s*
shows *select xs a i* = *length xs*
⟨*proof*⟩

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

9.3 General Properties

lemma *sa-subset-upt*:
set (*sa s*) $\subseteq$ $\{0..<$ *length s*$\}$
⟨*proof*⟩

lemma *sa-suffix-sorted*:
sorted (*map* (*suffix s*) (*sa s*))
⟨*proof*⟩

9.4 Nth Properties

lemma *sa-nth-suc-le*:
assumes $j <$ *length s*
and $i < j$
and *s* ! (*sa s* ! *i*) = *s* ! (*sa s* ! *j*)
and *Suc* (*sa s* ! *i*) < *length s*
and *Suc* (*sa s* ! *j*) < *length s*
shows *s* ! *Suc* (*sa s* ! *i*) $\leq$ *s* ! (*Suc* (*sa s* ! *j*))
⟨*proof*⟩

lemma *sa-nth-suc-le-ex*:

assumes $j < \mathit{length}\ s$
and $i < j$
and $s\ !\ (\mathit{sa}\ s\ !\ i) = s\ !\ (\mathit{sa}\ s\ !\ j)$
and $\mathit{Suc}\ (\mathit{sa}\ s\ !\ i) < \mathit{length}\ s$
and $\mathit{Suc}\ (\mathit{sa}\ s\ !\ j) < \mathit{length}\ s$
shows $\exists\, k\ l.\ k < l \wedge \mathit{sa}\ s\ !\ k = \mathit{Suc}\ (\mathit{sa}\ s\ !\ i) \wedge \mathit{sa}\ s\ !\ l = \mathit{Suc}\ (\mathit{sa}\ s\ !\ j)$
$\langle \mathit{proof} \rangle$

lemma *sorted-map-nths-sa*:
$\mathit{sorted}\ (\mathit{map}\ (\mathit{nth}\ s)\ (\mathit{sa}\ s))$
$\langle \mathit{proof} \rangle$

lemma *perm-map-nths-sa*:
$s <^{\sim\sim}> \mathit{map}\ (\mathit{nth}\ s)\ (\mathit{sa}\ s)$
$\langle \mathit{proof} \rangle$

lemma *sort-eq-map-nths-sa*:
$\mathit{sort}\ s = \mathit{map}\ (\mathit{nth}\ s)\ (\mathit{sa}\ s)$
$\langle \mathit{proof} \rangle$

lemma *sort-sa-nth*:
$i < \mathit{length}\ s \Longrightarrow \mathit{sort}\ s\ !\ i = s\ !\ (\mathit{sa}\ s\ !\ i)$
$\langle \mathit{proof} \rangle$

lemma *inj-on-nth-sa-upt*:
assumes $j \le \mathit{length}\ s\ l \le \mathit{length}\ s$
shows $\mathit{inj\text{-}on}\ (\mathit{nth}\ (\mathit{sa}\ s))\ (\{i..{<}j\} \cup \{k..{<}l\})$
$\langle \mathit{proof} \rangle$

lemma *nth-sa-upt-set*:
$\mathit{nth}\ (\mathit{sa}\ s)\ `\ \{0..{<}\mathit{length}\ s\} = \{0..{<}\mathit{length}\ s\}$
$\langle \mathit{proof} \rangle$

9.5 Valid List Properties

lemma *valid-list-sa-hd*:
assumes *valid-list s*
shows $\exists\, n.\ \mathit{length}\ s = \mathit{Suc}\ n \wedge \mathit{sa}\ s\ !\ 0 = n$
$\langle \mathit{proof} \rangle$

lemma *valid-list-not-last*:
assumes *valid-list s*
and $i < \mathit{length}\ s$
and $j < \mathit{length}\ s$
and $i \neq j$
and $s\ !\ i = s\ !\ j$
shows $i < \mathit{length}\ s - 1 \wedge j < \mathit{length}\ s - 1$
$\langle \mathit{proof} \rangle$

end

lemma *Suffix-Array-General-ex*:
  $\exists$ *sa. Suffix-Array-General sa*
  ⟨*proof*⟩

end
theory *SA-Count*
  imports *Rank-Select*
          ../util/SA-Util
begin

10 Counting Properties on Suffix Arays

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

10.1 Counting Properties

lemma *sa-card-index*:
  assumes $i < length\ s$
  shows $i = card\ \{j.\ j < length\ s \wedge suffix\ s\ (sa\ s\ !\ j) < suffix\ s\ (sa\ s\ !\ i)\}$
    (**is** $i = card\ ?A$)
⟨*proof*⟩

corollary *sa-card-s-index*:
  assumes $i < length\ s$
  shows $i = card\ \{j.\ j < length\ s \wedge suffix\ s\ j < suffix\ s\ (sa\ s\ !\ i)\}$
    (**is** $i = card\ ?A$)
⟨*proof*⟩

lemma *sa-card-s-idx*:
  assumes $i < length\ s$
  shows $i = card\ \{j.\ j < length\ s \wedge s\ !\ j < s\ !\ (sa\ s\ !\ i)\}$ +
      $card\ \{j.\ j < length\ s \wedge s\ !\ j = s\ !\ (sa\ s\ !\ i) \wedge suffix\ s\ j < suffix\ s\ (sa\ s\ !$
$i)\}$
⟨*proof*⟩

lemma *sa-card-index-lower-bound*:
  assumes $i < length\ s$
  shows $card\ \{j.\ j < length\ s \wedge s\ !\ (sa\ s\ !\ j) < s\ !\ (sa\ s\ !\ i)\} \leq i$
  (**is** $card\ ?A \leq i$)
⟨*proof*⟩

lemma *sa-card-rank-idx*:
  assumes $i < length\ s$
  shows $i = card\ \{j.\ j < length\ s \wedge s\ !\ (sa\ s\ !\ j) < s\ !\ (sa\ s\ !\ i)\}$
      $+\ rank\ (sort\ s)\ (s\ !\ (sa\ s\ !\ i))\ i$
⟨*proof*⟩

corollary *sa-card-rank-s-idx*:
  assumes $i < length\ s$
  shows $i = card\ \{j.\ j < length\ s \wedge s\ !\ j < s\ !\ (sa\ s\ !\ i)\}$
    $+ rank\ (sort\ s)\ (s\ !\ (sa\ s\ !\ i))\ i$
⟨*proof*⟩

lemma *sa-rank-nth*:
  assumes $i < length\ s$
  shows $rank\ (sort\ s)\ (s\ !\ (sa\ s\ !\ i))\ i =$
    $card\ \{j.\ j < length\ s \wedge s\ !\ j = s\ !\ (sa\ s\ !\ i) \wedge$
      $suffix\ s\ j < suffix\ s\ (sa\ s\ !\ i)\}$
⟨*proof*⟩

lemma *sa-suffix-nth*:
  assumes $card\ \{k.\ k < length\ s \wedge s\ !\ k < c\ \} + i < length\ s$
  and $i < count\text{-}list\ s\ c$
shows $\exists\, as.\ suffix\ s\ (sa\ s\ !\ (card\ \{k.\ k < length\ s \wedge s\ !\ k < c\} + i)) = c\ \#\ as$
⟨*proof*⟩

10.2 Ordering Properties

lemma *sa-suffix-order-le*:
  assumes $card\ \{k.\ k < length\ s \wedge s\ !\ k < c\ \} < length\ s$
  shows $[c] \leq suffix\ s\ (sa\ s\ !\ (card\ \{k.\ k < length\ s \wedge s\ !\ k < c\}))$
⟨*proof*⟩

lemma *sa-suffix-order-le-gen*:
  assumes $card\ \{k.\ k < length\ s \wedge s\ !\ k < c\ \} + i < length\ s$
  shows $[c] \leq suffix\ s\ (sa\ s\ !\ (card\ \{k.\ k < length\ s \wedge s\ !\ k < c\} + i))$
⟨*proof*⟩

lemma *sa-suffix-nth-less*:
  assumes $i < card\ \{k.\ k < length\ s \wedge s\ !\ k < c\}$
  shows $\forall\, as.\ suffix\ s\ (sa\ s\ !\ i) < c\ \#\ as$
⟨*proof*⟩

lemma *sa-suffix-nth-gr*:
  assumes $card\ \{k.\ k < length\ s \wedge s\ !\ k < c\} + i < length\ s$
  and $count\text{-}list\ s\ c \leq i$
shows $\forall\, as.\ c\ \#\ as < suffix\ s\ (sa\ s\ !\ (card\ \{k.\ k < length\ s \wedge s\ !\ k < c\} + i))$
⟨*proof*⟩

end

end
theory *BWT*
  imports *../../util/SA-Util*

begin

11 Burrows-Wheeler Transform

Based on [2]

Definition 3.3 from [3]: Canonical BWT

definition *bwt-canon* :: $('a :: \{linorder, order\text{-}bot\})$ *list* $\Rightarrow$ *'a list*
where
$bwt\text{-}canon\ s = map\ last\ (sort\ (map\ (\lambda x.\ rotate\ x\ s)\ [0..<length\ s]))$

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

Definition 3.4 from [3]: Suffix Array Version of the BWT

definition *bwt-sa* :: $('a :: \{linorder, order\text{-}bot\})$ *list* $\Rightarrow$ *'a list*
where
$bwt\text{-}sa\ s = map\ (\lambda i.\ s\ !\ ((i + length\ s - Suc\ 0)\ mod\ (length\ s)))\ (sa\ s)$

end

12 BWT Verification

12.1 List Rotations

lemma *rotate-suffix-prefix*:
assumes $i < length\ xs$
shows $rotate\ i\ xs = suffix\ xs\ i\ @\ prefix\ xs\ i$
⟨*proof*⟩

lemma *rotate-last*:
assumes $i < length\ xs$
shows $last\ (rotate\ i\ xs) = xs\ !\ ((i + length\ xs - Suc\ 0)\ mod\ (length\ xs))$
⟨*proof*⟩

lemma (**in** *Suffix-Array-General*) *map-last-rotations*:
$map\ last\ (map\ (\lambda i.\ rotate\ i\ s)\ (sa\ s)) = bwt\text{-}sa\ s$
⟨*proof*⟩

lemma *distinct-rotations*:
assumes *valid-list s*
and $i < length\ s$
and $j < length\ s$
and $i \neq j$
shows $rotate\ i\ s \neq rotate\ j\ s$
⟨*proof*⟩

12.2 Ordering

lemma *list-less-suffix-app-prefix-1*:
assumes *valid-list xs*
and $i < length\ xs$

and $j < length\ xs$
and $suffix\ xs\ i < suffix\ xs\ j$
shows $suffix\ xs\ i\ @\ prefix\ xs\ i < suffix\ xs\ j\ @\ prefix\ xs\ j$
⟨*proof*⟩

lemma *list-less-suffix-app-prefix-2*:
assumes *valid-list xs*
and $i < length\ xs$
and $j < length\ xs$
and $suffix\ xs\ i\ @\ prefix\ xs\ i < suffix\ xs\ j\ @\ prefix\ xs\ j$
shows $suffix\ xs\ i < suffix\ xs\ j$
⟨*proof*⟩

corollary *list-less-suffix-app-prefix*:
assumes *valid-list xs*
and $i < length\ xs$
and $j < length\ xs$
shows $suffix\ xs\ i < suffix\ xs\ j \longleftrightarrow$
$suffix\ xs\ i\ @\ prefix\ xs\ i < suffix\ xs\ j\ @\ prefix\ xs\ j$
⟨*proof*⟩

Theorem 3.5 from [3]: Same Suffix and Rotation Order

lemma *list-less-suffix-rotate*:
assumes *valid-list xs*
and $i < length\ xs$
and $j < length\ xs$
shows $suffix\ xs\ i < suffix\ xs\ j \longleftrightarrow rotate\ i\ xs < rotate\ j\ xs$
⟨*proof*⟩

lemma (**in** *Suffix-Array-General*) *sorted-rotations*:
assumes *valid-list s*
shows *strict-sorted* (*map* ($\lambda i.\ rotate\ i\ s$) (*sa s*))
⟨*proof*⟩

12.3 BWT Equivalence

Theorem 3.6 from [3]: BWT and Suffix Array Correspondence Canoncial BWT and BWT via Suffix Array Correspondence

theorem (**in** *Suffix-Array-General*) *bwt-canon-eq-bwt-sa*:
assumes *valid-list s*
shows $bwt\text{-}canon\ s = bwt\text{-}sa\ s$
⟨*proof*⟩

end
theory *BWT-SA-Corres*
imports *BWT*
../../counting/SA-Count
../../util/Rotated-Substring
begin

13 BWT and Suffix Array Correspondence

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

Definition 3.12 from [3]: BWT Permutation

definition *bwt-perm* :: ('*a* :: {*linorder*, *order-bot*}) *list* ⇒ *nat list*
  where
bwt-perm s = *map* (λ*i*. (*i* + *length s* − *Suc 0*) *mod* (*length s*)) (*sa s*)

13.1 BWT Using Suffix Arrays

lemma *map-bwt-indexes*:
  fixes *s* :: ('*a* :: {*linorder*, *order-bot*}) *list*
  shows *bwt-sa s* = *map* (λ*i*. *s* ! *i*) (*bwt-perm s*)
  ⟨*proof*⟩

lemma *map-bwt-indexes-perm*:
  fixes *s* :: ('*a* :: {*linorder*, *order-bot*}) *list*
  shows *bwt-perm s* <~~> [*0*..<*length s*]
⟨*proof*⟩

lemma *bwt-sa-perm*:
  fixes *s* :: ('*a* :: {*linorder*, *order-bot*}) *list*
  shows *bwt-sa s* <~~> *s*
  ⟨*proof*⟩

lemma *bwt-sa-nth*:
  fixes *s* :: ('*a* :: {*linorder*, *order-bot*}) *list*
  fixes *i* :: *nat*
  assumes *i* < *length s*
  shows *bwt-sa s* ! *i* = *s* ! (((*sa s* ! *i*) + *length s* − *1*) *mod* (*length s*))
  ⟨*proof*⟩

lemma *bwt-perm-nth*:
  fixes *s* :: ('*a* :: {*linorder*, *order-bot*}) *list*
  fixes *i* :: *nat*
  assumes *i* < *length s*
  shows *bwt-perm s* ! *i* = ((*sa s* ! *i*) + *length s* − *1*) *mod* (*length s*)
  ⟨*proof*⟩

lemma *bwt-perm-s-nth*:
  fixes *s* :: ('*a* :: {*linorder*, *order-bot*}) *list*
  fixes *i* :: *nat*
  assumes *i* < *length s*
  shows *bwt-sa s* ! *i* = *s* ! (*bwt-perm s* ! *i*)
  ⟨*proof*⟩

lemma *bwt-sa-length*:
  fixes *s* :: ('*a* :: {*linorder*, *order-bot*}) *list*
  shows *length* (*bwt-sa s*) = *length s*

⟨*proof*⟩

lemma *bwt-perm-length*:
fixes s :: (′*a* :: {*linorder*, *order-bot*}) *list*
shows *length* (*bwt-perm* s) = *length* s
⟨*proof*⟩

lemma *ex-bwt-perm-nth*:
fixes s :: (′*a* :: {*linorder*, *order-bot*}) *list*
fixes k :: *nat*
assumes $k <$ *length* s
shows $\exists\, i <$ *length* s. *bwt-perm* s ! $i = k$
⟨*proof*⟩

lemma *valid-list-sa-index-helper*:
fixes s :: (′*a* :: {*linorder*, *order-bot*}) *list*
fixes $i\ j$ :: *nat*
assumes *valid-list* s
and $i <$ *length* s
and $j <$ *length* s
and $i \neq j$
and s ! (*bwt-perm* s ! i) = s ! (*bwt-perm* s ! j)

shows *sa* s ! $i \neq 0$
⟨*proof*⟩

Theorem 3.13 from [3]: Suffix Relative Order Preservation Relative order of the suffixes is maintained by the BWT permutation

lemma *bwt-relative-order*:
fixes s :: (′*a* :: {*linorder*, *order-bot*}) *list*
fixes $i\ j$ :: *nat*
assumes *valid-list* s
and $i < j$
and $j <$ *length* s
and s ! (*bwt-perm* s ! i) = s ! (*bwt-perm* s ! j)
shows *suffix* s (*bwt-perm* s ! i) < *suffix* s (*bwt-perm* s ! j)
⟨*proof*⟩

lemma *bwt-sa-card-s-idx*:
fixes s :: (′*a* :: {*linorder*, *order-bot*}) *list*
fixes i :: *nat*
assumes *valid-list* s
and $i <$ *length* s
shows i = *card* {j. $j <$ *length* $s \land j < i \land$ *bwt-sa* s ! $j \neq$ *bwt-sa* s ! i} +
card {j. $j <$ *length* $s \land s$! j = *bwt-sa* s ! $i \land$
suffix s j < *suffix* s (*bwt-perm* s ! i)}
⟨*proof*⟩

lemma *bwt-perm-to-sa-idx*:

assumes *valid-list s*
and $i < length\ s$
shows $\exists\, k < length\ s.\ sa\ s\ !\ k = bwt\text{-}perm\ s\ !\ i \wedge$
$k = card\ \{j.\ j < length\ s \wedge s\ !\ j < bwt\text{-}sa\ s\ !\ i\} +$
$card\ \{j.\ j < length\ s \wedge s\ !\ j = bwt\text{-}sa\ s\ !\ i \wedge$
$suffix\ s\ j < suffix\ s\ (bwt\text{-}perm\ s\ !\ i)\}$
⟨*proof*⟩

corollary *bwt-perm-eq*:
fixes $s :: ('a :: \{linorder,\ order\text{-}bot\})\ list$
fixes $i :: nat$
assumes *valid-list s*
and $i < length\ s$
shows $bwt\text{-}perm\ s\ !\ i =$
$sa\ s\ !\ (card\ \{j.\ j < length\ s \wedge s\ !\ j < bwt\text{-}sa\ s\ !\ i\} +$
$card\ \{j.\ j < length\ s \wedge s\ !\ j = bwt\text{-}sa\ s\ !\ i \wedge$
$suffix\ s\ j < suffix\ s\ (bwt\text{-}perm\ s\ !\ i)\})$
⟨*proof*⟩

13.2 BWT Rank Properties

lemma *bwt-perm-rank-nth*:
fixes $s :: ('a :: \{linorder,\ order\text{-}bot\})\ list$
fixes $i :: nat$
assumes *valid-list s*
and $i < length\ s$
shows $rank\ (bwt\text{-}sa\ s)\ (bwt\text{-}sa\ s\ !\ i)\ i =$
$card\ \{j.\ j < length\ s \wedge s\ !\ j = bwt\text{-}sa\ s\ !\ i \wedge$
$suffix\ s\ j < suffix\ s\ (bwt\text{-}perm\ s\ !\ i)\}$
⟨*proof*⟩

lemma *bwt-sa-card-rank-s-idx*:
fixes $s :: ('a :: \{linorder,\ order\text{-}bot\})\ list$
fixes $i :: nat$
assumes *valid-list s*
and $i < length\ s$
shows $i = card\ \{j.\ j < length\ s \wedge j < i \wedge bwt\text{-}sa\ s\ !\ j \neq bwt\text{-}sa\ s\ !\ i\} +$
$rank\ (bwt\text{-}sa\ s)\ (bwt\text{-}sa\ s\ !\ i)\ i$
⟨*proof*⟩

13.3 Suffix Array and BWT Rank

lemma *sa-bwt-perm-same-rank*:
fixes $s :: ('a :: \{linorder,\ order\text{-}bot\})\ list$
fixes $i\ j :: nat$
assumes *valid-list s*
and $i < length\ s$
and $j < length\ s$
and $sa\ s\ !\ i = bwt\text{-}perm\ s\ !\ j$
shows $rank\ (sort\ s)\ (s\ !\ (sa\ s\ !\ i))\ i = rank\ (bwt\text{-}sa\ s)\ (bwt\text{-}sa\ s\ !\ j)\ j$

⟨*proof*⟩

Theorem 3.17 from [3]: Same Rank Rank for each symbol is the same in the BWT and suffix array

lemma *rank-same-sa-bwt-perm*:
  fixes $s :: ('a :: \{linorder, order\text{-}bot\})\ list$
  fixes $i\ j :: nat$
  fixes $v :: 'a$
  assumes *valid-list s*
  and $i < length\ s$
  and $j < length\ s$
  and $s\ !\ (sa\ s\ !\ i) = v$
  and $bwt\text{-}sa\ s\ !\ j = v$
  and $rank\ (sort\ s)\ v\ i = rank\ (bwt\text{-}sa\ s)\ v\ j$
shows $sa\ s\ !\ i = bwt\text{-}perm\ s\ !\ j$
⟨*proof*⟩

lemma *rank-bwt-card-suffix*:
  fixes $s :: ('a :: \{linorder, order\text{-}bot\})\ list$
  fixes $i :: nat$
  fixes $a :: 'a$
  assumes $i < length\ s$
  shows $rank\ (bwt\text{-}sa\ s)\ a\ i =$
    $card\ \{k.\ k < length\ s \wedge k < i \wedge bwt\text{-}sa\ s\ !\ k = a \wedge$
      $a\ \#\ suffix\ s\ (sa\ s\ !\ k) < a\ \#\ suffix\ s\ (sa\ s\ !\ i)\}$
⟨*proof*⟩

lemma *sa-to-bwt-perm-idx*:
  fixes $s :: ('a :: \{linorder, order\text{-}bot\})\ list$
  fixes $i :: nat$
  assumes *valid-list s*
  and $i < length\ s$
shows $sa\ s\ !\ i =$
  $bwt\text{-}perm\ s\ !\ (select\ (bwt\text{-}sa\ s)\ (s\ !\ (sa\ s\ !\ i))\ (rank\ (sort\ s)\ (s\ !\ (sa\ s\ !\ i))\ i))$
⟨*proof*⟩

lemma *suffix-bwt-perm-sa*:
  fixes $s :: ('a :: \{linorder, order\text{-}bot\})\ list$
  fixes $i :: nat$
  assumes *valid-list s*
  and $i < length\ s$
  and $bwt\text{-}sa\ s\ !\ i \neq bot$
shows $suffix\ s\ (bwt\text{-}perm\ s\ !\ i) = bwt\text{-}sa\ s\ !\ i\ \#\ suffix\ s\ (sa\ s\ !\ i)$
⟨*proof*⟩

end

end
theory *IBWT*

imports *BWT-SA-Corres*
begin

14 Inverse Burrows-Wheeler Transform

Inverse BWT algorithm obtained from [6]

14.1 Abstract Versions

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

These are abstract because they use additional information about the original string and its suffix array.

Definition 3.15 from [3]: Abstract LF-Mapping

fun *lf-map-abs* :: *'a list* ⇒ *nat* ⇒ *nat*
where
lf-map-abs s i = *select* (*sort s*) (*bwt-sa s* ! *i*) (*rank* (*bwt-sa s*) (*bwt-sa s* ! *i*) *i*)

Definition 3.16 from [3]: Inverse BWT Permutation

fun *ibwt-perm-abs* :: *nat* ⇒ *'a list* ⇒ *nat* ⇒ *nat list*
where
ibwt-perm-abs 0 - - = [] |
ibwt-perm-abs (*Suc n*) *s i* = *ibwt-perm-abs n s* (*lf-map-abs s i*) @ [*i*]

end

14.2 Concrete Versions

These are concrete because they only rely on the BWT-transformed sequence without any additional information.

Definition 3.14 from [3]: Inverse BWT - LF-mapping

fun *lf-map-conc* :: (*'a* :: {*linorder*, *order-bot*}) *list* ⇒ *'a list* ⇒ *nat* ⇒ *nat*
where
lf-map-conc ss bs i = (*select ss* (*bs* ! *i*) *0*) + (*rank bs* (*bs* ! *i*) *i*)

fun *ibwt-perm-conc* :: *nat* ⇒ (*'a* :: {*linorder*, *order-bot*}) *list* ⇒ *'a list* ⇒ *nat* ⇒ *nat list*
where
ibwt-perm-conc 0 - - - = [] |
ibwt-perm-conc (*Suc n*) *ss bs i* = *ibwt-perm-conc n ss bs* (*lf-map-conc ss bs i*) @ [*i*]

Definition 3.14 from [3]: Inverse BWT - Inverse BWT Rotated Subsequence

fun *ibwtn* :: *nat* ⇒ (*'a* :: {*linorder*, *order-bot*}) *list* ⇒ *'a list* ⇒ *nat* ⇒ *'a list*

where
ibwtn 0 - - - = [] |
ibwtn (Suc n) ss bs i = ibwtn n ss bs (lf-map-conc ss bs i) @ [bs ! i]

Definition 3.14 from [3]: Inverse BWT

fun *ibwt* :: ('*a* :: {*linorder*, *order-bot*}) *list* ⇒ '*a list*
where
ibwt bs = ibwtn (length bs) (sort bs) bs (select bs bot 0)

15 List Filter

lemma *filter-nth-app-upt*:
filter (λ*i*. *P* (*xs* ! *i*)) [*0*..<*length xs*] = *filter* (λ*i*. *P* ((*xs* @ *ys*) ! *i*)) [*0*..<*length xs*]
⟨*proof*⟩

lemma *filter-eq-nth-upt*:
filter P xs = *map* (λ*i*. *xs* ! *i*) (*filter* (λ*i*. *P* (*xs* ! *i*)) [*0*..<*length xs*])
⟨*proof*⟩

lemma *distinct-filter-nth-upt*:
distinct (*filter* (λ*i*. *P* (*xs* ! *i*)) [*0*..<*length xs*])
⟨*proof*⟩

lemma *filter-nth-upt-set*:
set (*filter* (λ*i*. *P* (*xs* ! *i*)) [*0*..<*length xs*]) = {*i*. *i* < *length xs* ∧ *P* (*xs* ! *i*)}
⟨*proof*⟩

lemma *filter-length-upt*:
length (*filter* (λ*i*. *P* (*xs* ! *i*)) [*0*..<*length xs*]) = *card* {*i*. *i* < *length xs* ∧ *P* (*xs* ! *i*)}
⟨*proof*⟩

lemma *perm-filter-length*:
xs <~~> *ys* ⟹
length (*filter* (λ*i*. *P* (*xs* ! *i*)) [*0*..<*length xs*])
= *length* (*filter* (λ*i*. *P* (*ys* ! *i*)) [*0*..<*length ys*])
⟨*proof*⟩

16 Verification of the Inverse Burrows-Wheeler Transform

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

16.1 LF-Mapping Simple Properties

lemma *lf-map-abs-less-length*:
fixes *s* :: '*a list*

fixes $i\ j$ :: *nat*
assumes $i <$ *length s*
shows *lf-map-abs s i* $<$ *length s*
⟨*proof*⟩

corollary *lf-map-abs-less-length-funpow*:
fixes *s* :: *'a list*
fixes $i\ j$ :: *nat*
assumes $i <$ *length s*
shows ((*lf-map-abs s*)^^*k*) $i <$ *length s*
⟨*proof*⟩

lemma *lf-map-abs-equiv*:
fixes *s* :: (*'a* :: {*linorder*, *order-bot*}) *list*
fixes $i\ r$ :: *nat*
fixes v :: *'a*
assumes $i <$ *length* (*bwt-sa s*)
and $v =$ *bwt-sa s* ! *i*
and $r =$ *rank* (*bwt-sa s*) *v i*
shows *lf-map-abs s i* $=$ *card* {*j*. $j <$ *length* (*bwt-sa s*) ∧ *bwt-sa s* ! $j < v$} $+ r$
⟨*proof*⟩

16.2 LF-Mapping Correctness

lemma *sa-lf-map-abs*:
assumes *valid-list s*
and $i <$ *length s*
shows *sa s* ! (*lf-map-abs s i*) $=$ (*sa s* ! i $+$ *length s* $-$ *Suc 0*) *mod* (*length s*)
⟨*proof*⟩

Theorem 3.18 from [3]: Abstract LF-Mapping Correctness

corollary *bwt-perm-lf-map-abs*:
fixes *s* :: (*'a* :: {*linorder*, *order-bot*}) *list*
fixes i :: *nat*
assumes *valid-list s*
and $i <$ *length s*
shows *bwt-perm s* ! (*lf-map-abs s i*) $=$ (*bwt-perm s* ! i $+$ *length s* $-$ *Suc 0*) *mod* (*length s*)
⟨*proof*⟩

16.3 Backwards Inverse BWT Simple Properties

lemma *ibwt-perm-abs-length*:
fixes *s* :: *'a list*
fixes $n\ i$ :: *nat*
shows *length* (*ibwt-perm-abs n s i*) $= n$
⟨*proof*⟩

lemma *ibwt-perm-abs-nth*:
fixes *s* :: *'a list*

fixes $k\ n\ i :: nat$
assumes $k \le n$
shows $(ibwt\text{-}perm\text{-}abs\ (Suc\ n)\ s\ i)\ !\ k = ((lf\text{-}map\text{-}abs\ s)^{\frown\frown}(n{-}k))\ i$
⟨*proof*⟩

corollary *ibwt-perm-abs-alt-nth*:
fixes $s :: 'a\ list$
fixes $n\ i\ k :: nat$
assumes $k < n$
shows $(ibwt\text{-}perm\text{-}abs\ n\ s\ i)\ !\ k = ((lf\text{-}map\text{-}abs\ s)^{\frown\frown}(n - Suc\ k))\ i$
⟨*proof*⟩

lemma *ibwt-perm-abs-nth-le-length*:
fixes $s :: 'a\ list$
fixes $n\ i\ k :: nat$
assumes $i < length\ s$
assumes $k < n$
shows $(ibwt\text{-}perm\text{-}abs\ n\ s\ i)\ !\ k < length\ s$
⟨*proof*⟩

lemma *ibwt-perm-abs-map-ver*:
$ibwt\text{-}perm\text{-}abs\ n\ s\ i = map\ (\lambda x.\ ((lf\text{-}map\text{-}abs\ s)^{\frown\frown}x)\ i)\ (rev\ [0..{<}n])$
⟨*proof*⟩

16.4 Backwards Inverse BWT Correctness

lemma *inc-one-bounded-sa-ibwt-perm-abs*:
fixes $s :: ('a :: \{linorder, order\text{-}bot\})\ list$
fixes $i\ n :: nat$
assumes $valid\text{-}list\ s$
and $i < length\ s$
shows $inc\text{-}one\text{-}bounded\ (length\ s)\ (map\ ((!)\ (sa\ s))\ (ibwt\text{-}perm\text{-}abs\ n\ s\ i))$
(**is** $inc\text{-}one\text{-}bounded\ ?n\ ?xs$)
⟨*proof*⟩

corollary *is-rot-sublist-sa-ibwt-perm-abs*:
fixes $s :: ('a :: \{linorder, order\text{-}bot\})\ list$
fixes $i\ n :: nat$
assumes $valid\text{-}list\ s$
and $i < length\ s$
and $n \le length\ s$
shows $is\text{-}rot\text{-}sublist\ [0..{<}length\ s]\ (map\ ((!)\ (sa\ s))\ (ibwt\text{-}perm\text{-}abs\ n\ s\ i))$
⟨*proof*⟩

lemma *inc-one-bounded-bwt-perm-ibwt-perm-abs*:
fixes $s :: ('a :: \{linorder, order\text{-}bot\})\ list$
fixes $i\ n :: nat$
assumes $valid\text{-}list\ s$
and $i < length\ s$

shows *inc-one-bounded* (*length s*) (*map* ((!) (*bwt-perm s*)) (*ibwt-perm-abs n s i*))
⟨*proof*⟩

Theorem 3.19 from [3]: Abstract Inverse BWT Permutation Rotated Sub-list

corollary *is-rot-sublist-bwt-perm-ibwt-perm-abs*:
fixes *s* :: (′*a* :: {*linorder*, *order-bot*}) *list*
fixes *i n* :: *nat*
assumes *valid-list s*
and $i <$ *length s*
and $n \leq$ *length s*
shows *is-rot-sublist* [*0*..<*length s*] (*map* ((!) (*bwt-perm s*)) (*ibwt-perm-abs n s i*))
⟨*proof*⟩

lemma *bwt-ibwt-perm-sa-lookup-idx*:
assumes *valid-list s*
shows *map* ((!) (*bwt-perm s*)) (*ibwt-perm-abs* (*length s*) *s* (*select* (*bwt-sa s*) *bot* *0*))
= [*0*..<*length s*]
⟨*proof*⟩

lemma *map-bwt-sa-bwt-perm*:
$\forall x \in$ *set xs*. $x <$ *length s* $\Longrightarrow$
map ((!) (*bwt-sa s*)) *xs* = *map* ((!) *s*) (*map* ((!) (*bwt-perm s*)) *xs*)
⟨*proof*⟩

theorem *ibwt-perm-abs-bwt-sa-lookup-correct*:
fixes *s* :: (′*a* :: {*linorder*, *order-bot*}) *list*
assumes *valid-list s*
shows *map* ((!) (*bwt-sa s*)) (*ibwt-perm-abs* (*length s*) *s* (*select* (*bwt-sa s*) *bot 0*)) = *s*
⟨*proof*⟩

16.5 Concretization

lemma *lf-map-abs-eq-conc*:
$i <$ *length s* $\Longrightarrow$ *lf-map-abs s i* = *lf-map-conc* (*sort* (*bwt-sa s*)) (*bwt-sa s*) *i*
⟨*proof*⟩

lemma *ibwt-perm-abs-conc-eq*:
$i <$ *length s* $\Longrightarrow$ *ibwt-perm-abs n s i* = *ibwt-perm-conc n* (*sort* (*bwt-sa s*)) (*bwt-sa s*) *i*
⟨*proof*⟩

theorem *ibwtn-bwt-sa-lookup-correct*:
fixes *s xs ys* :: (′*a* :: {*linorder*, *order-bot*}) *list*
assumes *valid-list s*
and *xs* = *sort* (*bwt-sa s*)
and *ys* = *bwt-sa s*

shows *map* ((!) *ys*) (*ibwt-perm-conc* (*length ys*) *xs ys* (*select ys bot 0*)) = *s*
⟨*proof*⟩

lemma *ibwtn-eq-map-ibwt-perm-conc*:
shows *ibwtn n ss bs i* = *map* ((!) *bs*) (*ibwt-perm-conc n ss bs i*)
⟨*proof*⟩

theorem *ibwtn-correct*:
fixes *s xs ys* :: (*'a* :: {*linorder*, *order-bot*}) *list*
assumes *valid-list s*
and *xs* = *sort* (*bwt-sa s*)
and *ys* = *bwt-sa s*
shows *ibwtn* (*length ys*) *xs ys* (*select ys bot 0*) = *s*
⟨*proof*⟩

16.6 Inverse BWT Correctness

BWT (suffix array version) is invertible

theorem *ibwt-correct*:
fixes *s* :: (*'a* :: {*linorder*, *order-bot*}) *list*
assumes *valid-list s*
shows *ibwt* (*bwt-sa s*) = *s*
⟨*proof*⟩

end

Theorem 3.20 from [3]: Correctness of the Inverse BWT

theorem *ibwt-correct-canon*:
fixes *s* :: (*'a* :: {*linorder*, *order-bot*}) *list*
assumes *valid-list s*
shows *ibwt* (*bwt-canon s*) = *s*
⟨*proof*⟩

end

References

[1] R. Affeldt, J. Garrigue, X. Qi, and K. Tanaka. Proving tree algorithms for succinct data structures. In *Proc. Interactive Theorem Proving*, volume 141 of *LIPIcs*, pages 5:1–5:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[2] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994.

[3] L. Cheung, A. Moffat, and C. Rizkallah. Formalized Burrows-Wheeler Transform. In *Proc. Ceritifed Programs and Proofs*. ACM, 2025. To appear.

[4] L. Cheung and C. Rizkallah. Formalized Burrows-Wheeler Transform (artefact), December 2024.

[5] L. Cheung and C. Rizkallah. Formally verified suffix array construction. *Archive of Formal Proofs*, September 2024. https://isa-afp.org/entries/SuffixArray.html, Formal proof development.

[6] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science*, pages 390–398. IEEE Computer Society, 2000.

[7] 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.