List Index

Tobias Nipkow

September 13, 2023

Abstract

This theory provides functions for finding the index of an element in a list, by predicate and by value.

1 Index-based manipulation of lists

theory *List-Index* **imports** *Main* **begin**

This theory collects functions for index-based manipulation of lists.

1.1 Finding an index

This subsection defines three functions for finding the index of items in a list:

find-index P xs finds the index of the first element in *xs* that satisfies *P*.

index xs x finds the index of the first occurrence of *x* in *xs*.

last-index xs x finds the index of the last occurrence of *x* in *xs*.

All functions return *length xs* if *xs* does not contain a suitable element.

The argument order of *find-index* follows the function of the same name in the Haskell standard library. For *index* (and *last-index*) the order is intentionally reversed: *index* maps lists to a mapping from elements to their indices, almost the inverse of function *nth*.

primrec *find-index* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow nat$ **where**
find-index - [] = *0* |
find-index P (*x*#*xs*) = (*if P x then 0 else find-index P xs* + *1*)

definition *index* :: $'a\ list \Rightarrow 'a \Rightarrow nat$ **where**
index xs = (λa. *find-index* (λx. *x*=*a*) *xs*)

definition *last-index* :: $'a\ list \Rightarrow 'a \Rightarrow nat$ **where**
last-index xs x =
(*let i* = *index* (*rev xs*) *x*; *n* = *size xs*

in if i = n then i else n − (i+1))

lemma *find-index-append*: *find-index P (xs @ ys) =*
(if ∃x∈set xs. P x then find-index P xs else size xs + find-index P ys)
by (*induct xs*) *simp-all*

lemma *find-index-le-size*: *find-index P xs <= size xs*
by(*induct xs*) *simp-all*

lemma *index-le-size*: *index xs x <= size xs*
by(*simp add*: *index-def find-index-le-size*)

lemma *last-index-le-size*: *last-index xs x <= size xs*
by(*simp add*: *last-index-def Let-def index-le-size*)

lemma *index-Nil*[*simp*]: *index* [] *a = 0*
by(*simp add*: *index-def*)

lemma *index-Cons*[*simp*]: *index (x#xs) a = (if x=a then 0 else index xs a + 1)*
by(*simp add*: *index-def*)

lemma *index-append*: *index (xs @ ys) x =*
(if x : set xs then index xs x else size xs + index ys x)
by (*induct xs*) *simp-all*

lemma *index-conv-size-if-notin*[*simp*]: *x ∉ set xs ⟹ index xs x = size xs*
by (*induct xs*) *auto*

lemma *find-index-eq-size-conv*:
size xs = n ⟹ (find-index P xs = n) = (∀x ∈ set xs. ~ P x)
by(*induct xs arbitrary*: *n*) *auto*

lemma *size-eq-find-index-conv*:
size xs = n ⟹ (n = find-index P xs) = (∀x ∈ set xs. ~ P x)
by(*metis find-index-eq-size-conv*)

lemma *index-size-conv*: *size xs = n ⟹ (index xs x = n) = (x ∉ set xs)*
by(*auto simp*: *index-def find-index-eq-size-conv*)

lemma *size-index-conv*: *size xs = n ⟹ (n = index xs x) = (x ∉ set xs)*
by (*metis index-size-conv*)

lemma *last-index-size-conv*:
size xs = n ⟹ (last-index xs x = n) = (x ∉ set xs)
apply(*auto simp*: *last-index-def index-size-conv*)
apply(*drule length-pos-if-in-set*)
apply *arith*
done

lemma *size-last-index-conv*:
  size xs = *n* $\Longrightarrow$ (*n* = *last-index xs x*) = (*x* $\notin$ *set xs*)
by (*metis last-index-size-conv*)

lemma *find-index-less-size-conv*:
  (*find-index P xs* < *size xs*) = ($\exists$ *x* $\in$ *set xs*. *P x*)
by (*induct xs*) *auto*

lemma *index-less-size-conv*:
  (*index xs x* < *size xs*) = (*x* $\in$ *set xs*)
by(*auto simp*: *index-def find-index-less-size-conv*)

lemma *last-index-less-size-conv*:
  (*last-index xs x* < *size xs*) = (*x* : *set xs*)
by(*simp add*: *last-index-def Let-def index-size-conv length-pos-if-in-set*
    del:*length-greater-0-conv*)

lemma *index-less*[*simp*]:
  x : *set xs* $\Longrightarrow$ *size xs* <= *n* $\Longrightarrow$ *index xs x* < *n*
apply(*induct xs*) **apply** *auto*
apply (*metis index-less-size-conv less-eq-Suc-le less-trans-Suc*)
done

lemma *last-index-less*[*simp*]:
  x : *set xs* $\Longrightarrow$ *size xs* <= *n* $\Longrightarrow$ *last-index xs x* < *n*
by(*simp add*: *last-index-less-size-conv*[*symmetric*])

lemma *last-index-Cons*: *last-index* (*x*#*xs*) *y* =
  (*if x*=*y then*
    if x $\in$ *set xs then last-index xs y* + *1 else 0*
   else last-index xs y + *1*)
using *index-le-size*[*of rev xs y*]
apply(*auto simp add*: *last-index-def index-append Let-def*)
apply(*simp add*: *index-size-conv*)
done

lemma *last-index-append*: *last-index* (*xs* @ *ys*) *x* =
  (*if x* : *set ys then size xs* + *last-index ys x*
   else if x : *set xs then last-index xs x else size xs* + *size ys*)
by (*induct xs*) (*simp-all add*: *last-index-Cons last-index-size-conv*)

lemma *last-index-Snoc*[*simp*]:
  last-index (*xs* @ [*x*]) *y* =
  (*if x*=*y then size xs*
   else if y : *set xs then last-index xs y else size xs* + *1*)
by(*simp add*: *last-index-append last-index-Cons*)

lemma *nth-find-index*: *find-index P xs* < *size xs* $\Longrightarrow$ *P*(*xs* ! *find-index P xs*)
by (*induct xs*) *auto*

lemma *nth-index*[*simp*]: $x \in set\ xs \Longrightarrow xs\ !\ index\ xs\ x = x$
by (*induct xs*) *auto*

lemma *nth-last-index*[*simp*]: $x \in set\ xs \Longrightarrow xs\ !\ last\text{-}index\ xs\ x = x$
by(*simp add*:*last-index-def index-size-conv Let-def rev-nth*[*symmetric*])

lemma *index-rev*: $[\![\ distinct\ xs;\ x \in set\ xs\ ]\!] \Longrightarrow$
$index\ (rev\ xs)\ x = length\ xs - index\ xs\ x - 1$
by (*induct xs*) (*auto simp*: *index-append*)

lemma *index-nth-id*:
$[\![\ distinct\ xs;\ \ n < length\ xs\ ]\!] \Longrightarrow index\ xs\ (xs\ !\ n) = n$
by (*metis in-set-conv-nth index-less-size-conv nth-eq-iff-index-eq nth-index*)

lemma *index-upt*[*simp*]: $m \le i \Longrightarrow i < n \Longrightarrow index\ [m..<n]\ i = i-m$
by (*induction n*) (*auto simp add*: *index-append*)

lemma *index-eq-index-conv*[*simp*]: $x \in set\ xs \lor y \in set\ xs \Longrightarrow$
$(index\ xs\ x = index\ xs\ y) = (x = y)$
by (*induct xs*) *auto*

lemma *last-index-eq-index-conv*[*simp*]: $x \in set\ xs \lor y \in set\ xs \Longrightarrow$
$(last\text{-}index\ xs\ x = last\text{-}index\ xs\ y) = (x = y)$
by (*induct xs*) (*auto simp*:*last-index-Cons*)

lemma *inj-on-index*: *inj-on* (*index xs*) (*set xs*)
by (*simp add*:*inj-on-def*)

lemma *inj-on-index2*: $I \subseteq set\ xs \Longrightarrow inj\text{-}on\ (index\ xs)\ I$
by (*rule inj-onI*) *auto*

lemma *inj-on-last-index*: *inj-on* (*last-index xs*) (*set xs*)
by (*simp add*:*inj-on-def*)

lemma *find-index-conv-takeWhile*:
$find\text{-}index\ P\ xs = size(takeWhile\ (Not\ o\ P)\ xs)$
by(*induct xs*) *auto*

lemma *index-conv-takeWhile*: $index\ xs\ x = size(takeWhile\ (\lambda y.\ x \ne y)\ xs)$
by(*induct xs*) *auto*

lemma *find-index-first*: $i < find\text{-}index\ P\ xs \Longrightarrow \neg P\ (xs!i)$
unfolding *find-index-conv-takeWhile*
by (*metis comp-apply nth-mem set-takeWhileD takeWhile-nth*)

lemma *index-first*: $i < index\ xs\ x \Longrightarrow x \ne xs!i$
using *find-index-first* **unfolding** *index-def* **by** *blast*

lemma *find-index-eqI*:
  assumes $i \leq length\ xs$
  assumes $\forall j<i.\ \neg P\ (xs!j)$
  assumes $i<length\ xs \Longrightarrow P\ (xs!i)$
  shows $find\text{-}index\ P\ xs = i$
by (*metis* (*mono-tags*, *lifting*) *antisym-conv2 assms find-index-eq-size-conv find-index-first find-index-less-size-conv linorder-neqE-nat nth-find-index*)

lemma *find-index-eq-iff*:
  $find\text{-}index\ P\ xs = i$
  $\longleftrightarrow (i \leq length\ xs \wedge (\forall j<i.\ \neg P\ (xs!j)) \wedge (i<length\ xs \longrightarrow P\ (xs!i)))$
by (*auto intro*: *find-index-eqI*
    simp: *nth-find-index find-index-le-size find-index-first*)

lemma *index-eqI*:
  assumes $i \leq length\ xs$
  assumes $\forall j<i.\ xs!j \neq x$
  assumes $i<length\ xs \Longrightarrow xs!i = x$
  shows $index\ xs\ x = i$
unfolding *index-def* **by** (*simp add*: *find-index-eqI assms*)

lemma *index-eq-iff*:
  $index\ xs\ x = i$
  $\longleftrightarrow (i \leq length\ xs \wedge (\forall j<i.\ xs!j \neq x) \wedge (i<length\ xs \longrightarrow xs!i = x))$
by (*auto intro*: *index-eqI*
    simp: *index-le-size index-less-size-conv*
    dest: *index-first*)

lemma *index-take*: $index\ xs\ x >= i \Longrightarrow x \notin set(take\ i\ xs)$
apply(*subst* (*asm*) *index-conv-takeWhile*)
apply(*subgoal-tac* $set(take\ i\ xs) <= set(takeWhile\ ((\neq)\ x)\ xs)$)
 apply(*blast dest*: *set-takeWhileD*)
apply(*metis set-take-subset-set-take takeWhile-eq-take*)
done

lemma *last-index-drop*:
  $last\text{-}index\ xs\ x < i \Longrightarrow x \notin set(drop\ i\ xs)$
apply(*subgoal-tac* $set(drop\ i\ xs) = set(take\ (size\ xs - i)\ (rev\ xs))$)
 apply(*simp add*: *last-index-def index-take Let-def split*:*if-split-asm*)
apply (*metis rev-drop set-rev*)
done

lemma *set-take-if-index*: **assumes** $index\ xs\ x < i$ **and** $i \leq length\ xs$
shows $x \in set\ (take\ i\ xs)$
proof −
  have $index\ (take\ i\ xs\ @\ drop\ i\ xs)\ x < i$
    using *append-take-drop-id*[*of i xs*] *assms*(*1*) **by** *simp*
  thus *?thesis* **using** *assms*(*2*)
    by(*simp add*:*index-append del*:*append-take-drop-id split*: *if-splits*)

qed

lemma *index-take-if-index*:
assumes $index\ xs\ x \leq n$ **shows** $index\ (take\ n\ xs)\ x = index\ xs\ x$
proof *cases*
  assume $x : set(take\ n\ xs)$ **with** *assms* **show** *?thesis*
    by (*metis append-take-drop-id index-append*)
next
  assume $x \notin set(take\ n\ xs)$ **with** *assms* **show** *?thesis*
  by (*metis order-le-less set-take-if-index le-cases length-take min-def size-index-conv take-all*)
qed

lemma *index-take-if-set*:
  $x : set(take\ n\ xs) \Longrightarrow index\ (take\ n\ xs)\ x = index\ xs\ x$
by (*metis index-take index-take-if-index linear*)

lemma *index-last*[*simp*]:
  $xs \neq [] \Longrightarrow distinct\ xs \Longrightarrow index\ xs\ (last\ xs) = length\ xs - 1$
by (*induction xs*) *auto*

lemma *index-update-if-diff2*:
  $n < length\ xs \Longrightarrow x \neq xs!n \Longrightarrow x \neq y \Longrightarrow index\ (xs[n := y])\ x = index\ xs\ x$
by(*subst* (*2*) *id-take-nth-drop*[*of n xs*])
  (*auto simp*: *upd-conv-take-nth-drop index-append min-def*)

lemma *set-drop-if-index*: $distinct\ xs \Longrightarrow index\ xs\ x < i \Longrightarrow x \notin set(drop\ i\ xs)$
by (*metis in-set-dropD index-nth-id last-index-drop last-index-less-size-conv nth-last-index*)

lemma *index-swap-if-distinct*: **assumes** $distinct\ xs\ i < size\ xs\ j < size\ xs$
shows $index\ (xs[i := xs!j,\ j := xs!i])\ x =$
  $(if\ x = xs!i\ then\ j\ else\ if\ x = xs!j\ then\ i\ else\ index\ xs\ x)$
proof−
  have $distinct(xs[i := xs!j,\ j := xs!i])$ **using** *assms* **by** *simp*
  with *assms* **show** *?thesis*
    apply (*auto simp*: *simp del*: *distinct-swap*)
    apply (*metis index-nth-id list-update-same-conv*)
   apply (*metis* (*erased*, *opaque-lifting*) *index-nth-id length-list-update list-update-swap nth-list-update-eq*)
    apply (*metis index-nth-id length-list-update nth-list-update-eq*)
    by (*metis index-update-if-diff2 length-list-update nth-list-update*)
qed

lemma *bij-betw-index*:
  $distinct\ xs \Longrightarrow X = set\ xs \Longrightarrow l = size\ xs \Longrightarrow bij\text{-}betw\ (index\ xs)\ X\ \{0..<l\}$
apply *simp*
apply(*rule bij-betw-imageI*[*OF inj-on-index*])
by (*auto simp*: *image-def*) (*metis index-nth-id nth-mem*)

lemma *index-image*: $distinct\ xs \Longrightarrow set\ xs = X \Longrightarrow index\ xs\ `\ X = \{0..<size\ xs\}$
by (*simp add*: *bij-betw-imp-surj-on bij-betw-index*)

lemma *index-map-inj-on*:
$[\![\ inj\text{-}on\ f\ S;\ y \in S;\ set\ xs \subseteq S\ ]\!] \Longrightarrow index\ (map\ f\ xs)\ (f\ y) = index\ xs\ y$
by (*induct xs*) (*auto simp*: *inj-on-eq-iff*)

lemma *index-map-inj*: $inj\ f \Longrightarrow index\ (map\ f\ xs)\ (f\ y) = index\ xs\ y$
by (*simp add*: *index-map-inj-on*[**where** *S=UNIV*])

1.2 Map with index

primrec *map-index'* :: $nat \Rightarrow (nat \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list$ **where**
map-index' *n f* [] = []
| *map-index'* *n f* (*x#xs*) = *f n x* # *map-index'* (*Suc n*) *f xs*

lemma *length-map-index'*[*simp*]: *length* (*map-index'* *n f xs*) = *length xs*
by (*induct xs arbitrary*: *n*) *auto*

lemma *map-index'-map-zip*: *map-index'* *n f xs* = *map* (*case-prod f*) (*zip* [*n* ..< *n* + *length xs*] *xs*)
proof (*induct xs arbitrary*: *n*)
case (*Cons x xs*)
hence *map-index'* *n f* (*x#xs*) = *f n x* # *map* (*case-prod f*) (*zip* [*Suc n* ..< *n* + *length* (*x* # *xs*)] *xs*) **by** *simp*
also have ... = *map* (*case-prod f*) (*zip* (*n* # [*Suc n* ..< *n* + *length* (*x* # *xs*)]) (*x* # *xs*)) **by** *simp*
also have (*n* # [*Suc n* ..< *n* + *length* (*x* # *xs*)]) = [*n* ..< *n* + *length* (*x* # *xs*)] **by** (*induct xs*) *auto*
finally show *?case* **by** *simp*
qed *simp*

abbreviation *map-index* ≡ *map-index'* *0*

lemmas *map-index* = *map-index'-map-zip*[*of 0*, *simplified*]

lemma *take-map-index*: *take p* (*map-index f xs*) = *map-index f* (*take p xs*)
unfolding *map-index* **by** (*auto simp*: *min-def take-map take-zip*)

lemma *drop-map-index*: *drop p* (*map-index f xs*) = *map-index'* *p f* (*drop p xs*)
unfolding *map-index'-map-zip* **by** (*cases p* < *length xs*) (*auto simp*: *drop-map drop-zip*)

lemma *map-map-index*[*simp*]: *map g* (*map-index f xs*) = *map-index* (*λn x. g* (*f n x*)) *xs*
unfolding *map-index* **by** *auto*

lemma *map-index-map*[*simp*]: *map-index f* (*map g xs*) = *map-index* (*λn x. f n* (*g x*)) *xs*

unfolding *map-index* **by** (*auto simp*: *map-zip-map2*)

lemma *set-map-index*[*simp*]: $x \in$ *set* (*map-index f xs*) = ($\exists$ *i* < *length xs*. *f i* (*xs* ! *i*) = *x*)
unfolding *map-index* **by** (*auto simp*: *set-zip intro*!: *image-eqI*[*of - case-prod f*])

lemma *set-map-index′*[*simp*]: *x*∈*set* (*map-index′ n f xs*)
⟷ (∃ *i*<*length xs*. *f* (*n*+*i*) (*xs*!*i*) = *x*)
unfolding *map-index′-map-zip*
by (*auto simp*: *set-zip intro*!: *image-eqI*[*of - case-prod f*])

lemma *nth-map-index*[*simp*]: *p* < *length xs* ⟹ *map-index f xs* ! *p* = *f p* (*xs* ! *p*)
unfolding *map-index* **by** *auto*

lemma *map-index-cong*:
∀ *p* < *length xs*. *f p* (*xs* ! *p*) = *g p* (*xs* ! *p*) ⟹ *map-index f xs* = *map-index g xs*
unfolding *map-index* **by** (*auto simp*: *set-zip*)

lemma *map-index-id*: *map-index* (*curry snd*) *xs* = *xs*
unfolding *map-index* **by** *auto*

lemma *map-index-no-index*[*simp*]: *map-index* (λ*n x*. *f x*) *xs* = *map f xs*
unfolding *map-index* **by** (*induct xs rule*: *rev-induct*) *auto*

lemma *map-index-congL*:
∀ *p* < *length xs*. *f p* (*xs* ! *p*) = *xs* ! *p* ⟹ *map-index f xs* = *xs*
by (*rule trans*[*OF map-index-cong map-index-id*]) *auto*

lemma *map-index′-is-NilD*: *map-index′ n f xs* = [] ⟹ *xs* = []
by (*induct xs*) *auto*

declare *map-index′-is-NilD*[*of 0*, *dest*!]

lemma *map-index′-is-ConsD*:
map-index′ n f xs = *y* # *ys* ⟹ ∃ *z zs*. *xs* = *z* # *zs* ∧ *f n z* = *y* ∧ *map-index′* (*n* + *1*) *f zs* = *ys*
by (*induct xs arbitrary*: *n*) *auto*

lemma *map-index′-eq-imp-length-eq*: *map-index′ n f xs* = *map-index′ n g ys* ⟹ *length xs* = *length ys*
proof (*induct ys arbitrary*: *xs n*)
case (*Cons y ys*) **thus** *?case* **by** (*cases xs*) *auto*
qed (*auto dest*!: *map-index′-is-NilD*)

lemmas *map-index-eq-imp-length-eq* = *map-index′-eq-imp-length-eq*[*of 0*]

lemma *map-index′-comp*[*simp*]: *map-index′ n f* (*map-index′ n g xs*) = *map-index′ n* (λ*n*. *f n o g n*) *xs*
by (*induct xs arbitrary*: *n*) *auto*

lemma *map-index′-append*[*simp*]: *map-index′ n f* (*a* @ *b*)
  = *map-index′ n f a* @ *map-index′* (*n* + *length a*) *f b*
  by (*induct a arbitrary*: *n*) *auto*

lemma *map-index-append*[*simp*]: *map-index f* (*a* @ *b*)
  = *map-index f a* @ *map-index′* (*length a*) *f b*
  using *map-index′-append*[**where** *n=0*]
  by (*simp del*: *map-index′-append*)

1.3 Insert at position

primrec *insert-nth* :: *nat* ⇒ *′a* ⇒ *′a list* ⇒ *′a list* **where**
  insert-nth 0 x xs = *x* # *xs*
| *insert-nth* (*Suc n*) *x xs* = (*case xs of* [] ⇒ [*x*] | *y* # *ys* ⇒ *y* # *insert-nth n x ys*)

lemma *insert-nth-take-drop*[*simp*]: *insert-nth n x xs* = *take n xs* @ [*x*] @ *drop n xs*
proof (*induct n arbitrary*: *xs*)
  case *Suc* **thus** *?case* **by** (*cases xs*) *auto*
qed *simp*

lemma *length-insert-nth*: *length* (*insert-nth n x xs*) = *Suc* (*length xs*)
  by (*induct xs*) *auto*

lemma *set-insert-nth*:
  set (*insert-nth i x xs*) = *insert x* (*set xs*)
by (*simp add*: *set-append*[*symmetric*])

lemma *distinct-insert-nth*:
  assumes *distinct xs*
  assumes *x* ∉ *set xs*
  shows *distinct* (*insert-nth i x xs*)
using *assms* **proof** (*induct xs arbitrary*: *i*)
  case *Nil*
  then show *?case* **by** (*cases i*) *auto*
next
  case (*Cons a xs*)
  then show *?case*
    by (*cases i*) (*auto simp add*: *set-insert-nth simp del*: *insert-nth-take-drop*)
qed

lemma *nth-insert-nth-front*:
  assumes *i* < *j j* ≤ *length xs*
  shows *insert-nth j x xs* ! *i* = *xs* ! *i*
using *assms* **by** (*simp add*: *nth-append*)

lemma *nth-insert-nth-index-eq*:
  assumes *i* ≤ *length xs*
  shows *insert-nth i x xs* ! *i* = *x*

using *assms* **by** (*simp add*: *nth-append*)

lemma *nth-insert-nth-back*:
  assumes $j < i$ $i \leq$ *length xs*
  shows *insert-nth j x xs* ! $i =$ *xs* ! $(i - 1)$
using *assms* **by** (*cases i*) (*auto simp add*: *nth-append min-def*)

lemma *nth-insert-nth*:
  assumes $i \leq$ *length xs* $j \leq$ *length xs*
  shows *insert-nth j x xs* ! $i =$ (*if* $i = j$ *then* x *else if* $i < j$ *then xs* ! i *else xs* ! $(i - 1)$)
using *assms* **by** (*simp add*: *nth-insert-nth-front nth-insert-nth-index-eq nth-insert-nth-back del*: *insert-nth-take-drop*)

lemma *insert-nth-inverse*:
  assumes $j \leq$ *length xs* $j' \leq$ *length xs'*
  assumes $x \notin$ *set xs* $x \notin$ *set xs'*
  assumes *insert-nth j x xs* = *insert-nth j' x xs'*
  shows $j = j'$
proof −
  from *assms*(*1*,*3*) **have** $\forall$ *i*≤*length xs*. *insert-nth j x xs* ! $i = x \longleftrightarrow i = j$
    by (*auto simp add*: *nth-insert-nth simp del*: *insert-nth-take-drop*)
  moreover from *assms*(*2*,*4*) **have** $\forall$ *i*≤*length xs'*. *insert-nth j' x xs'* ! $i = x \longleftrightarrow i = j'$
    by (*auto simp add*: *nth-insert-nth simp del*: *insert-nth-take-drop*)
  ultimately show $j = j'$
    using *assms*(*1*,*2*,*5*) **by** (*metis dual-order.trans nat-le-linear*)
qed

Insert several elements at given (ascending) positions

lemma *length-fold-insert-nth*:
  length (*fold* (λ(*p*, *b*). *insert-nth p b*) *pxs xs*) = *length xs* + *length pxs*
  by (*induct pxs arbitrary*: *xs*) *auto*

lemma *invar-fold-insert-nth*:
  ⟦∀ *x*∈*set pxs*. $p <$ *fst x*; $p <$ *length xs*; *xs* ! $p = b$⟧ ⟹
    fold (λ(*x*, *y*). *insert-nth x y*) *pxs xs* ! $p = b$
  by (*induct pxs arbitrary*: *xs*) (*auto simp*: *nth-append*)

lemma *nth-fold-insert-nth*:
  ⟦*sorted* (*map fst pxs*); *distinct* (*map fst pxs*); ∀ (*p*, *b*) ∈ *set pxs*. $p <$ *length xs* + *length pxs*;
    $i <$ *length pxs*; *pxs* ! i = (*p*, *b*)⟧ ⟹
  fold (λ(*p*, *b*). *insert-nth p b*) *pxs xs* ! $p = b$
proof (*induct pxs arbitrary*: *xs i p b*)
  case (*Cons pb pxs*)
  show *?case*
  proof (*cases i*)
    case *0*

with *Cons.prems* **have** $p < Suc\ (length\ xs)$
proof (*induct pxs rule*: *rev-induct*)
case (*snoc pb′ pxs*)
then obtain $p'\ b'$ **where** $pb' = (p', b')$ **by** *auto*
with *snoc.prems* **have** $\forall p \in fst\ `\ set\ pxs.\ p < p'\ p' \leq Suc\ (length\ xs + length\ pxs)$
by (*auto simp*: *image-iff sorted-wrt-append le-eq-less-or-eq*)
with *snoc.prems* **show** *?case* **by** (*intro snoc*(*1*)) (*auto simp*: *sorted-append*)
qed *auto*
with *0 Cons.prems* **show** *?thesis* **unfolding** *fold.simps o-apply*
by (*intro invar-fold-insert-nth*) (*auto simp*: *image-iff le-eq-less-or-eq nth-append*)
next
case (*Suc n*) **with** *Cons.prems* **show** *?thesis* **unfolding** *fold.simps*
by (*auto intro*!: *Cons*(*1*))
qed
qed *simp*

1.4 Remove at position

fun *remove-nth* :: $nat \Rightarrow 'a\ list \Rightarrow 'a\ list$
where
remove-nth i [] = []
| *remove-nth 0* (*x* # *xs*) = *xs*
| *remove-nth* (*Suc i*) (*x* # *xs*) = *x* # *remove-nth i xs*

lemma *remove-nth-take-drop*:
remove-nth i xs = *take i xs* @ *drop* (*Suc i*) *xs*
proof (*induct xs arbitrary*: *i*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case* **by** (*cases i*) *auto*
qed

lemma *remove-nth-insert-nth*:
assumes $i \leq length\ xs$
shows *remove-nth i* (*insert-nth i x xs*) = *xs*
using *assms* **proof** (*induct xs arbitrary*: *i*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case* **by** (*cases i*) *auto*
qed

lemma *insert-nth-remove-nth*:
assumes $i < length\ xs$
shows *insert-nth i* (*xs* ! *i*) (*remove-nth i xs*) = *xs*

using *assms* **proof** (*induct xs arbitrary*: *i*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases i*) *auto*
qed

lemma *length-remove-nth*:
  assumes $i < length\ xs$
  shows *length* (*remove-nth i xs*) = *length xs* − *1*
using *assms* **unfolding** *remove-nth-take-drop* **by** *simp*

lemma *set-remove-nth-subset*:
  set (*remove-nth j xs*) ⊆ *set xs*
proof (*induct xs arbitrary*: *j*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases j*) *auto*
qed

lemma *set-remove-nth*:
  assumes *distinct xs* $j < length\ xs$
  shows *set* (*remove-nth j xs*) = *set xs* − {*xs* ! *j*}
using *assms* **proof** (*induct xs arbitrary*: *j*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases j*) *auto*
qed

lemma *distinct-remove-nth*:
  assumes *distinct xs*
  shows *distinct* (*remove-nth i xs*)
using *assms* **proof** (*induct xs arbitrary*: *i*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case*
    by (*cases i*) (*auto simp add*: *set-remove-nth-subset rev-subsetD*)
qed

end