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

theory List-Index imports Main begin
This theory collects functions for index-based manipulation of lists.

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

\text{find-index} P \text{ xs} finds the index of the first element in \text{xs} that satisfies \text{P}.

\text{index} \text{ xs} \text{ x} finds the index of the first occurrence of \text{x} in \text{xs}.

\text{last-index} \text{ xs} \text{ x} finds the index of the last occurrence of \text{x} in \text{xs}.

All functions return \text{length \text{xs}} if \text{xs} does not contain a suitable element.

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

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

\textbf{definition} \text{index} :: \text{'a list} \Rightarrow \text{'a} \Rightarrow \text{nat where}
\text{index} \text{ xs} = (\lambda \text{x}. \text{find-index} (\lambda \text{x}. \text{x}=\text{a}) \text{ xs})

\textbf{definition} \text{last-index} :: \text{'a list} \Rightarrow \text{'a} \Rightarrow \text{nat where}
\text{last-index} \text{ xs} \text{ x} =
(\text{let} \text{i} = \text{index} (\text{rev} \text{ xs}) \text{ x}; \text{n} = \text{size} \text{ xs}
\text{ in if} \text{i} = \text{n} \text{ then} \text{i} \text{ else} \text{n} - (\text{i} + 1))
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) = (ALL 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) = (ALL 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 (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
apply (metis last-index-size-conv)
done

lemma size-last-index-conv:
size xs = n ⇒ (n = last-index xs x) = (x /∈ set xs)
by (metis last-index-size-conv)

lemma find-index-less-size-conv:
\[(\text{find-index } P \text{ } xs < \text{ size } xs) = (\exists x : \text{ set } xs. \text{ P } x)\]

by (induct xs) auto

lemma index-less-size-conv:
\[(\text{index } xs \text{ } x < \text{ size } xs) = (x \in \text{ set } xs)\]

by(auto simp: index-def find-index-less-size-conv)

lemma last-index-less-size-conv:
\[(\text{last-index } xs \text{ } x < \text{ size } xs) = (x : \text{ set } xs)\]

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]:
\[x : \text{ set } xs \implies \text{ size } xs <= n \implies \text{ index } xs \text{ } x < n\]

apply(induct xs) apply auto

apply (metis index-less-size-conv less-eq-Suc-le less-trans-Suc)
done

lemma last-index-less[simp]:
\[x : \text{ set } xs \implies \text{ size } xs <= n \implies \text{ last-index } xs \text{ } x < n\]

by(simp add: last-index-less-size-conv[symmetric])

lemma last-index-Cons: last-index \(x \# xs\) \(y\) =
\[(\text{if } x = y \text{ then}
\begin{align*}
\text{if } x & \in \text{ set } xs \text{ then last-index } xs \text{ } y + 1 \text{ else } 0 \\
\text{else } \text{ last-index } xs \text{ } y + 1
\end{align*}\)

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 \((xs \@ ys)\) \(x\) =
\[(\text{if } x : \text{ set } ys \text{ then size } xs + \text{ last-index } ys \text{ } x \text{ \begin{align*}
\text{else if } x : \text{ set } xs \text{ then last-index } xs \text{ } x \text{ else size } xs + \text{ size } ys
\end{align*}})\]

by (induct xs) (simp-all add: last-index-Cons last-index-size-conv)

lemma last-index-Snoc[simp]:
\[\text{last-index } (xs \@ [x]) \text{ } y =\]
\[(\text{if } x = y \text{ then size } xs \text{ \begin{align*}
\text{else if } y : \text{ set } xs \text{ then last-index } xs \text{ } y \text{ else size } xs + 1
\end{align*}})\]

by(simp add: last-index-append last-index-Cons)

lemma nth-find-index: find-index \(P \text{ } xs < \text{ size } xs \implies P(xs ! \text{ find-index } P \text{ } xs)\)

by (induct xs) auto

lemma nth-index[simp]: \(x \in \text{ set } xs \implies xs ! \text{ index } xs \text{ } x = x\)

by (induct xs) auto

lemma nth-last-index[simp]: \(x \in \text{ set } xs \implies xs ! \text{ last-index } xs \text{ } x = x\)

3
by (simp add: last-index-def index-size-conv Let-def rev-nth [symmetric])

lemma index-nth-id:
[ \text{distinct } xs; \ n < \text{length } xs ] \implies index xs (xs ! n) = n
by (metis in-set-cong-nth index-less-size-conv nth-eq-iff-index-eq nth-index)

lemma index-upt [simp]: \text{m} \leq i \implies i < n \implies index [m..<n] i = i - \text{m}
by (induction n) (auto simp add: index-append)

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

lemma last-index-eq-index-conv [simp]: \text{x} \in \text{set } xs \lor \text{y} \in \text{set } xs = \implies (last-index xs \text{x} = last-index xs \text{y}) = (\text{x} = \text{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 \text{set } xs \implies inj-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 index-conv-takeWhile: index xs \text{x} = \text{size} (\text{takeWhile } (\lambda y. \text{x} \neq y) \text{xs})
by (induct xs) auto

lemma index-take: index xs \text{x} \geq i \implies \text{x} \notin \text{set} (\text{take } i \text{xs})
apply (subst (asm) index-conv-takeWhile)
apply (subgoal-tac set (\text{take } i \text{xs}) \leq set (\text{takeWhile } (op \neq x) \text{xs}))
apply (blast dest: set-takeWhileD)
apply (metis set-take-subset-set-take takeWhile-eq-take)
done

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

lemma set-take-if-index: \textbf{assumes} index xs \text{x} < i \textbf{ and } i \leq \text{length } xs
shows \text{x} \in \text{set} (\text{take } i \text{xs})
proof 
  have index (\text{take } i \text{xs} @ \text{drop } i \text{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 ≤ 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 /∈ 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) =⇒ index (take n xs) x = index xs x
by (metis index-take index-take-if-index linear)

lemma index-last [simp]:
x ≠ [] =⇒ distinct xs =⇒ index xs (last xs) = length xs - 1
by (induction xs) auto

lemma index-update-if-diff2:
  n < length xs =⇒ x ≠ xs!n =⇒ x ≠ y =⇒ 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 =⇒ index xs x < i =⇒ x /∈ 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: swap-def simp del: distinct-swap)
  apply (metis index-nth-id list-update-same-conv)
  apply (metis (erased, hide-lams) 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 =⇒ X = set xs =⇒ l = size xs =⇒ bij-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 → set xs = X → index xs · X = {0..<size xs} 
by (simp add: bij-betw-imageE bij-betw-index)

0.2 Map with index

primrec map-index' :: nat ⇒ (nat ⇒ 'a ⇒ 'b) ⇒ 'a list ⇒ '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 (split 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 (split f) (zip [Suc n..< n + length (x # xs)] xs) by simp
also have ... = map (split f) (zip (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 ∈ set (map-index f xs) = (∃ i < length xs. f i (xs ! i) = x)
unfolding map-index by (auto simp: set-zip intro!: image-eqI[of - split f])
lemma set-map-index[simp]: \( x \in set \ (map-index' \ n \ f \ xs) \)
\( \iff (\exists i < length xs. f (n+i) \ (xs!i) = x) \)
unfolding map-index'-map-zip
by (auto simp: set-zip intro!: image-eqI[of - split f])

lemma nth-map-index[simp]: \( p < \text{length } xs \implies \text{map-index } f \ xs \ ! \ p = f \ p \ (xs \ ! p) \)
unfolding map-index by auto

lemma map-index-cong:
\( \forall p < \text{length } xs. f \ p \ (xs \ ! p) = g \ p \ (xs \ ! p) \implies \text{map-index } f \ xs = \text{map-index } g \ xs \)
unfolding map-index by (auto simp: set-zip)

lemma map-index-id: \( \text{map-index} \ (\text{curry } snd) \ xs = xs \)
unfolding map-index by auto

lemma map-index-no-index[simp]: \( \text{map-index} \ (\lambda n \ x. f \ x) \ xs = \text{map } f \ xs \)
unfolding map-index by (induct xs rule: rev-induct) auto

lemma map-index-congL:
\( \forall p < \text{length } xs. f \ p \ (xs \ ! p) = xs \ ! p \implies \text{map-index } f \ xs = xs \)
by (rule trans[of map-index-cong map-index-id]) auto

lemma map-index'-is-NilD: \( \text{map-index}' \ n \ f \ xs = [] \implies xs = [] \)
by (induct xs) auto

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

lemma map-index'-is-ConsD:
\( \text{map-index}' \ n \ f \ zs = y \ # \ ys \implies \exists z \ zs. xs = z \ # \ zs \land f \ n \ z = y \land \text{map-index}' \)
\( (n+1) \ f \ zs = ys \)
by (induct xs arbitrary: n) auto

lemma map-index'-eq-imp-length-eq:
\( \text{map-index}' \ n \ f \ xs = \text{map-index}' \ n \ g \ ys \implies \text{length } xs = \text{length } ys \)
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]: \( \text{map-index}' \ n \ f \ (\text{map-index}' \ n \ g \ xs) = \text{map-index}' \n (\lambda n. f \ n \ o \ g \ n) \ xs \)
by (induct xs arbitrary: n) auto

lemma map-index'-append[simp]: \( \text{map-index}' \ n \ f \ (a @ b) = \text{map-index}' \ n \ f \ a @ \text{map-index}' \ (n + \text{length } a) \ f \ b \)
by (induct a arbitrary: n) auto

7
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)

0.3 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

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 ∀ p ∈ fst ' set pxs. p < p' p' ≤ Suc (length xs + length pxs)
by (auto simp: image_iff sorted-Cons sorted-append le_eq_less_or_eq)
with snoc.prems show ?case by (intro snoc(1)) (auto simp: sorted-Cons sorted-append)
qed auto

8
with 0 Cons.prems show ?thesis unfolding fold.simps o-apply
by (intro invar-fold-insert-nth) (auto simp: sorted-Cons 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) simp: sorted-Cons)

qed simp

end