Power Operator for Lists

Štěpán Holub, Martin Raška, Štěpán Starosta and Tobias Nipkow

March 17, 2025

Abstract

This entry defines the power operator `xs ^^ n`, the `n`-fold concatenation of `xs` with itself.

Much of the theory is taken from the AFP entry Combinatorics on Words Basics where the operator is called `^@`. This new entry uses the standard overloaded `^^` syntax and is aimed at becoming the central theory of the power operator for lists that can be extended easily.

1 The Power Operator $\frown\frown$ on Lists

theory *List-Power*
imports *Main*
begin

overloading *pow-list* == *compow* :: *nat* ⇒ *'a list* ⇒ *'a list*
begin

primrec *pow-list* :: *nat* ⇒ *'a list* ⇒ *'a list* **where**
pow-list 0 xs = [] |
pow-list (*Suc n*) *xs* = *xs* @ *pow-list n xs*

end

context
begin

interpretation *monoid-mult* [] *append*
  rewrites *power u n* = *u* $\frown\frown$ *n*
proof−
  show *class.monoid-mult* [] (@)
    by (*unfold-locales*, *simp-all*)
  show *power.power* [] (@) *u n* = *u* $\frown\frown$ *n*
    by(*induction n*) (*auto simp add*: *power.power.simps*)
qed

— inherited power properties

lemmas *pow-list-zero* = *power.power-0* **and**
pow-list-one = *power-Suc0-right* **and**
pow-list-1 = *power-one-right* **and**
pow-list-Nil = *power-one* **and**
pow-list-2 = *power2-eq-square* **and**
pow-list-Suc = *power-Suc* **and**
pow-list-Suc2 = *power-Suc2* **and**
pow-list-comm = *power-commutes* **and**
pow-list-add = *power-add* **and**
pow-list-eq-if = *power-eq-if* **and**
pow-list-mult = *power-mult* **and**
pow-list-commuting-commutes = *power-commuting-commutes*

end

lemma *pow-list-alt*: $xs^{\frown\frown}n = concat\ (replicate\ n\ xs)$
by (*induct n*) *auto*

lemma *pow-list-single*: $[a]\ ^{\frown\frown}\ m = replicate\ m\ a$
by(*simp add*: *pow-list-alt*)

lemma *length-pow-list-single* [*simp*]: $length([a]\ ^{\frown\frown}\ n) = n$
by (*simp add*: *pow-list-single*)

lemma *nth-pow-list-single*: $i < m \Longrightarrow ([a]\ ^{\frown\frown}\ m)\ !\ i = a$
by (*simp add*: *pow-list-single*)

lemma *pow-list-not-NilD*: $xs\ ^{\frown\frown}\ m \neq [] \Longrightarrow 0 < m$
by (*cases m*) *auto*

lemma *length-pow-list*: $length(xs\ ^{\frown\frown}\ k) = k * length\ xs$
by (*induction k*) *simp+*

lemma *pow-list-set*: $set\ (w\ ^{\frown\frown}\ Suc\ k) = set\ w$
by (*induction k*)(*simp-all*)

lemma *pow-list-slide*: $xs\ @\ (ys\ @\ xs)\ ^{\frown\frown}\ n\ @\ ys = (xs\ @\ ys)^{\frown\frown}(Suc\ n)$
by (*induction n*) *simp+*

lemma *hd-pow-list*: $0 < n \Longrightarrow hd(xs\ ^{\frown\frown}\ n) = hd\ xs$
by(*auto simp*: *pow-list-alt hd-append gr0-conv-Suc*)

lemma *rev-pow-list*: $rev\ (xs\ ^{\frown\frown}\ m) = (rev\ xs)\ ^{\frown\frown}\ m$
by (*induction m*)(*auto simp*: *pow-list-comm*)

lemma *eq-pow-list-iff-eq-exp*[*simp*]: **assumes** $xs \neq []$ **shows** $xs\ ^{\frown\frown}\ k = xs\ ^{\frown\frown}\ m$
$\longleftrightarrow k = m$
proof
assume $k = m$ **thus** $xs\ ^{\frown\frown}\ k = xs\ ^{\frown\frown}\ m$ **by** *simp*

next
  assume $xs \frown\frown k = xs \frown\frown m$
  thus $k = m$ **using** ‹$xs \neq []$›[*folded length-0-conv*]
    by (*metis length-pow-list mult-cancel2*)
qed

lemma *pow-list-Nil-iff-0*: $xs \neq [] \Longrightarrow xs \frown\frown m = [] \longleftrightarrow m = 0$
by (*simp add*: *pow-list-eq-if*)

lemma *pow-list-Nil-iff-Nil*: $0 < m \Longrightarrow xs \frown\frown m = [] \longleftrightarrow xs = []$
by (*cases xs*) (*auto simp add*: *pow-list-Nil pow-list-Nil-iff-0*)

lemma *pow-eq-eq*:
  assumes $xs \frown\frown k = ys \frown\frown k$ **and** $0 < k$
  shows $(xs::'a\ list) = ys$
proof−
  have *length xs* = *length ys*
    using *assms*(*1*) *length-pow-list* **by** (*metis nat-mult-eq-cancel1*[*OF* ‹*0* < *k*›])
 thus *?thesis* **by** (*metis Suc-pred append-eq-append-conv assms*(*1*,*2*) *pow-list.simps*(*2*))
qed

lemma *map-pow-list*[*simp*]: *map f* $(xs \frown\frown k)$ = $(map\ f\ xs) \frown\frown k$
by (*induction k*) *simp-all*

lemma *concat-pow-list*: *concat* $(xs \frown\frown k)$ = $(concat\ xs) \frown\frown k$
by (*induction k*) *simp-all*

lemma *concat-pow-list-single*[*simp*]: *concat* $([a] \frown\frown k) = a \frown\frown k$
by (*simp add*: *pow-list-alt*)

lemma *pow-list-single-Nil-iff*: $[a] \frown\frown n = [] \longleftrightarrow n = 0$
by (*simp add*: *pow-list-single*)

lemma *hd-pow-list-single*: $k \neq 0 \Longrightarrow hd\ ([a] \frown\frown k) = a$
by (*cases k*) *simp*+

lemma *index-pow-mod*: $i < length(xs \frown\frown k) \Longrightarrow (xs \frown\frown k)!i = xs!(i\ mod\ length\ xs)$
proof(*induction k*)
  have *aux*: *length*$(xs \frown\frown Suc\ l)$ = *length*$(xs \frown\frown l)$ + *length xs* **for** *l*
    by *simp*
  have *aux1*: *length* $(xs \frown\frown l) \leq i \Longrightarrow i < length(xs \frown\frown l) + length\ xs \Longrightarrow$ *i mod length xs* = *i* − *length*$(xs \frown\frown l)$ **for** *l*
    unfolding *length-pow-list*[*of l xs*]
    using *less-diff-conv2*[*of l* ∗ *length xs i length xs*, *unfolded add.commute*[*of length xs l* ∗ *length xs*]]
      le-add-diff-inverse[*of l*∗*length xs i*]
    by (*simp add*: *mod-nat-eqI*)
  case (*Suc k*)
  show *?case*

unfolding *aux sym*[*OF pow-list-Suc2*[*symmetric*]] *nth-append le-mod-geq*
using *aux1*[*OF - Suc.prems*[*unfolded aux*]]
Suc.IH pow-list-Suc2[*symmetric*] *Suc.prems*[*unfolded aux*] *leI*[*of i length*(*xs* @ *k*)] **by** *presburger*
qed *auto*

lemma *unique-letter-word*: **assumes** $\bigwedge c.\ c \in set\ w \Longrightarrow c = a$ **shows** $w = [a]\ @\ length\ w$
using *assms* **proof** (*induction w*)
case (*Cons b w*)
have $[a]\ @\ length\ w = w$ **using** *Cons.IH*[*OF Cons.prems*[*OF list.set-intros*(*2*)]]..
then show $b\ \#\ w = [a]\ @\ length(b\ \#\ w)$
unfolding *Cons.prems*[*OF list.set-intros*(*1*)] **by** *auto*
qed *simp*

lemma *count-list-pow-list*: *count-list* $(w\ @\ k)\ a = k * (count\text{-}list\ w\ a)$
by (*induction k*) *simp+*

lemma *sing-pow-lists*: $a \in A \Longrightarrow [a]\ @\ n \in lists\ A$
by (*induction n*) *auto*

lemma *one-generated-list-power*: $u \in lists\ \{x\} \Longrightarrow \exists k.\ concat\ u = x\ @\ k$
proof(*induction u rule*: *lists.induct*)
case *Nil*
then show *?case* **by** (*metis concat.simps*(*1*) *pow-list.simps*(*1*))
next
case *Cons*
then show *?case* **by** (*metis concat.simps*(*2*) *pow-list-Suc singletonD*)
qed

lemma *pow-list-in-lists*: $0 < k \Longrightarrow u\ @\ k \in lists\ B \Longrightarrow u \in lists\ B$
by (*metis Suc-pred in-lists-conv-set pow-list-set*)

end