Power Operator for Lists

Štěpán Holub, Martin Raška, Štěpán Starosta and Tobias Nipkow

October 13, 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 ^^ 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* ^^ *n*
⟨*proof*⟩

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

lemmas[*simp*] = *pow-list-Nil pow-list-zero pow-list-one pow-list-1 pow-list-Suc pow-list-2*

lemma *pow-list-alt*: *xs*^^*n* = *concat* (*replicate n xs*)
⟨*proof*⟩

lemma *pow-list-single*: [*a*] ^^ *m* = *replicate m a*
⟨*proof*⟩

lemma *length-pow-list-single* [*simp*]: *length*([*a*] ^^ *n*) = *n*
⟨*proof*⟩

lemma *nth-pow-list-single*: *i* < *m* ⟹ ([*a*] ^^ *m*) ! *i* = *a*
⟨*proof*⟩

lemma *pow-list-not-NilD*: *xs* ^^ *m* ≠ [] ⟹ *0* < *m*
⟨*proof*⟩

lemma *length-pow-list*: *length*(*xs* ^^ *k*) = *k* ∗ *length xs*
⟨*proof*⟩

lemma *pow-list-set*: *set* (*w* ^^ *Suc k*) = *set w*
⟨*proof*⟩

lemma *pow-list-set-if*: *set* (*w* ^^ *k*) = (*if k=0 then* {} *else set w*)
⟨*proof*⟩

lemma *in-pow-list-set*[*simp*]: *x* ∈ *set* (*ys* ^^ *m*) ⟷ *x* ∈ *set ys* ∧ *m* ≠ *0*
⟨*proof*⟩

lemma *pow-list-slide*: *xs* @ (*ys* @ *xs*) ^^ *n* @ *ys* = (*xs* @ *ys*)^^(*Suc n*)
⟨*proof*⟩

lemma *hd-pow-list*: *0* < *n* ⟹ *hd*(*xs* ^^ *n*) = *hd xs*
⟨*proof*⟩

lemma *rev-pow-list*: *rev* (*xs* ^^ *m*) = (*rev xs*) ^^ *m*
⟨*proof*⟩

lemma *eq-pow-list-iff-eq-exp*[*simp*]: **assumes** *xs* ≠ [] **shows** *xs* ^^ *k* = *xs* ^^ *m* ⟷ *k* = *m*
⟨*proof*⟩

lemma *pow-list-Nil-iff-0*: $xs \neq [] \Longrightarrow xs$ ^^ $m = [] \longleftrightarrow m = 0$
⟨*proof*⟩

lemma *pow-list-Nil-iff-Nil*: $0 < m \Longrightarrow xs$ ^^ $m = [] \longleftrightarrow xs = []$
⟨*proof*⟩

lemma *pow-eq-eq*:
assumes xs ^^ $k = ys$ ^^ k **and** $0 < k$
shows $(xs::'a\ list) = ys$
⟨*proof*⟩

lemma *pow-list-eq-appends-iff*:
$n \geq m \Longrightarrow xs$^^$n$ @ $ys = xs$^^m @ $zs \longleftrightarrow zs = xs$^^$(n-m)$ @ ys
⟨*proof*⟩

lemmas *pow-list-eq-appends-iff2* = *pow-list-eq-appends-iff*[*THEN eq-iff-swap*]

lemma *pow-list-eq-single-appends-iff*[*simp*]:
$[\![\, x \notin set\ ys;\ x \notin set\ zs \,]\!] \Longrightarrow [x]$^^$m$ @ $ys = [x]$^^n @ $zs \longleftrightarrow m = n \wedge ys = zs$
⟨*proof*⟩

lemma *map-pow-list*[*simp*]: $map\ f\ (xs$ ^^ $k) = (map\ f\ xs)$ ^^ k
⟨*proof*⟩

lemma *concat-pow-list*: $concat\ (xs$ ^^ $k) = (concat\ xs)$ ^^ k
⟨*proof*⟩

lemma *concat-pow-list-single*[*simp*]: $concat\ ([a]$ ^^ $k) = a$ ^^ k
⟨*proof*⟩

lemma *pow-list-single-Nil-iff*: $[a]$ ^^ $n = [] \longleftrightarrow n = 0$
⟨*proof*⟩

lemma *hd-pow-list-single*: $k \neq 0 \Longrightarrow hd\ ([a]$ ^^ $k) = a$
⟨*proof*⟩

lemma *index-pow-mod*: $i < length(xs$ ^^ $k) \Longrightarrow (xs$ ^^ $k)!i = xs!(i\ mod\ length\ xs)$
⟨*proof*⟩

lemma *unique-letter-word*: **assumes** $\bigwedge c.\ c \in set\ w \Longrightarrow c = a$ **shows** $w = [a]$ ^^ $length\ w$
⟨*proof*⟩

lemma *count-list-pow-list*: $count\text{-}list\ (w$ ^^ $k)\ a = k * (count\text{-}list\ w\ a)$
⟨*proof*⟩

lemma *sing-pow-lists*: $a \in A \Longrightarrow [a]$ ^^ $n \in lists\ A$
⟨*proof*⟩

lemma *one-generated-list-power*: $u \in lists\ \{x\} \Longrightarrow \exists\, k.\ concat\ u = x\ \widehat{}\widehat{}\ k$
⟨*proof*⟩

lemma *pow-list-in-lists*: $0 < k \Longrightarrow u\ \widehat{}\widehat{}\ k \in lists\ B \Longrightarrow u \in lists\ B$
⟨*proof*⟩

For code generation.

context
begin

qualified definition *list-pow* :: *nat* ⇒ *'a list* ⇒ *'a list*
where *list-pow-code-def* [*code-abbrev*]: *list-pow* = *compow*

lemma [*code*]:
list-pow 0 u = []
list-pow (*Suc n*) *u* = *u* @ *list-pow n u*
⟨*proof*⟩

end

lemma *pows-list-comm*: $t\widehat{}\widehat{}k$ @ $t\widehat{}\widehat{}m$ = $t\widehat{}\widehat{}m$ @ $t\widehat{}\widehat{}k$
⟨*proof*⟩

lemma *comm-append-pow-list-iff*: u @ $v = v$ @ $u \longleftrightarrow (\exists\ r\ k\ m.\ u = r\widehat{}\widehat{}k \wedge v = r\widehat{}\widehat{}m)$
⟨*proof*⟩

lemma *pow-list-comm-comm*: **assumes** $0 < j$ **and** $x\widehat{}\widehat{}j = y\widehat{}\widehat{}k$ **shows** x @ y = y @ x
⟨*proof*⟩

lemma *comm-common-pow-list-iff*: u @ $v = v$ @ $u \longleftrightarrow u\ \widehat{}\widehat{}\ length\ v = v\ \widehat{}\widehat{}\ length\ u$
⟨*proof*⟩

lemma *comm-pows-list-comm*: **assumes** $0 < k$ $0 < m$
shows $u\ \widehat{}\widehat{}\ k$ @ $v\ \widehat{}\widehat{}\ m = v\ \widehat{}\widehat{}\ m$ @ $u\ \widehat{}\widehat{}\ k \longleftrightarrow u$ @ $v = v$ @ u
⟨*proof*⟩

lemma *rotate1-pow-list-swap*: $rotate1\ (u\ \widehat{}\widehat{}\ k) = (rotate1\ u)\ \widehat{}\widehat{}\ k$
⟨*proof*⟩

lemma *rotate-pow-list-swap*: $rotate\ n\ (u\ \widehat{}\widehat{}\ k) = (rotate\ n\ u)\ \widehat{}\widehat{}\ k$
⟨*proof*⟩

end