# fast-power, a function that computes \( x \times n \) with \( \log n \) operations, optimized to avoid the last doubling done by a simpler algorithm when it's unnecessary. Based upon the power function developed in Stepanov & McJones, Elements of Programming, pp. 41-42.

---

```plaintext
load "nat-half"
load "power"
load "strong-induction"

---

# fast-power

```
define fpp_1-correctness :=
(forall n x . n /= zero => (fpp_1 x n) = x +* n)

define fpp_2-correctness :=
(forall n x . n /= zero => (fpp_2 x n) = x +* (two * n + ^ one))

define correctness := (forall n x . (fast-power x n) = x +* n)

define theorems := [pap_1-correctness0 pap_1-correctness fpp_2-correctness
fpp_1-correctness0 fpp_1-correctness correctness]

#..........................................................................

define theorems := [pap_1-correctness0 pap_1-correctness fpp_2-correctness
fpp_1-correctness0 fpp_1-correctness correctness]

#..........................................................................

define proofs :=
method (theorem adapt)
let {[get prove chain chain-> chain<->] := (proof-tools adapt theory);
[pap_1 pap_2 fpp_1 fpp_2 fast-power] :=
(adapt [pap_1 pap_2 fpp_1 fpp_2 fast-power]);
PR1 := (!prove Power.right-one);
PR2 := (!prove Power.right-two);
PRT := (!prove Power.right-times);
PRP := (!prove Power.right-plus))
match theorem {
(val-of pap_1-correctness0) =>
let {theorem := (adapt theorem)}
(!strong-induction.principle theorem'
method (n)
assume ind-hyp := (strong-induction.hypothesis theorem' n)
conclude (strong-induction.conclusion theorem' n)
assume (n /= zero)
pick-any x
(!two-cases
assume (n = one)
pick-any r
let [C := (!chain->
[(odd ($ zero))
<==> (odd n) N.one-definition]
=> ((pap_1 r x n) = (pap_2 (r + x) x n)) [pap-odd]])
(!combine-equations
(!chain
[(pap_1 r x n) = (pap_2 (r + x) x n) [C]
= (r + x) [pap-one (n = one)]])
(!chain
[(r + x ++ one) = (r + x) [PR1]])
assume (n /= one)
let (fact1a := ((half n) /= zero =>
(forall ?a ?r . (pap_1 ?r ?a (half n)) = (?r + ?a ++ (half n))));
fact1b := (forall ?a ?r . (pap_1 ?r ?a (half n)) =
(?r + ?a ++ (half n)));
fact2 := (forall ?r . (pap_1 ?r (x + x) (half n)) =
(?r + x ++ (two * (half n))));
_ := (!chain->
[(n /= zero)
==> ((half n) < n) [N.half.less]
==> fact1a [ind-hyp]]);
D := (!by-contradiction ((half n) /= zero)
assume ((half n) = zero)
let (E := (!chain->
[(half n) = zero]
==> (n = zero | n = one)
[N.half.equal-zero]])
(!cases E
assume (n = zero)
(!absurd (n = zero) (n /= zero))
assume \( n = \text{one} \)

\[
\text{absurd} (n = \text{one}) (n =/\ = \text{one}))\)
\]

\_ := \(!\text{chain}\rightarrow \text{D} \Rightarrow \text{fact1b} \ [\text{fact1a}]\))

\_ := \text{conclude} \text{fact2}

\text{pick-any} \ r

\[
\text{(!chain}
\]

\[
\text{[(pap}_1 \ r \ (x + x) \ (\text{half } n))
\]

\[
= (r + (x + x) (\text{half } n)) \ [\text{fact1b}]
\]

\[
= (r + (x + \text{two} \ (\text{half } n)) \ [\text{PR2}]
\]

\[
= (r + x + (\text{two} + (\text{half } n))) \ [\text{PR3}]\)
\]

\text{(!two-cases}

\text{assume} (\text{even } n)

\text{pick-any} \ r

\text{let} (F := \(!\text{chain}\rightarrow

\text{[(even } n) \Rightarrow (\sim \text{odd } n) \ [\text{N.EO.not-odd-if-even}]\))

\text{(!chain}

\[
\text{[(pap}_1 \ r \ x \ n)
\]

\[
= (pap}_1 \ r \ (x + x) \ (\text{half } n)) \ [\text{pap-even } F]
\]

\[
= (r + x + \text{two} \ (\text{half } n)) \ [\text{fact2}]
\]

\[
= (r + x + \text{one} + \text{two} \ (\text{half } n)) \ [\text{N.EO.even-definition}]\)
\]

\text{assume} (\sim \text{even } n)

\text{pick-any} \ r

\text{let} (G := \(!\text{chain}\rightarrow

\text{[(\sim even } n) \Rightarrow (\text{odd } n) \ [\text{N.EO.odd-if-not-even}]\))

\text{(!chain}

\[
\text{[(pap}_1 \ r \ x \ n)
\]

\[
= (pap}_2 (r + x) x \ n) \ [\text{pap-odd } G]
\]

\[
= (pap}_1 \ r \ ((x + x) (\text{half } n)) \ [\text{pap-not-one}]
\]

\[
= ((r + x) + \text{one} + \text{two} \ (\text{half } n)) \ [\text{fact2}]\]

\[
= (r + x + \text{one} + \text{two} \ (\text{half } n))) \ [\text{associative}]
\]

\[
= (r + (x + \text{one} + \text{two} \ (\text{half } n))) \ [\text{PR1}]
\]

\[
= (r + (x + \text{one} + \text{two} \ (\text{half } n))) \ [\text{PRP}]
\]

\[
= (r + x + \text{two} \ (\text{half } n) + \text{one})) \ [\text{N.Plus.commutative}]
\]

\[
= (r + x + \text{one} + \text{two} \ (\text{half } n)) \ [\text{N.Plus.odd-definition}]\)
\]

\text{! (\text{val-of pap}_1\text{-correctness}) =>

\text{let} (\text{PC0} := \(!\text{prove pap}_1\text{-correctness}\))

\text{pick-any} \ n \ x \ r

\text{assume} (n =/= \text{zero})

\text{let} (i := \(!\text{chain}\rightarrow \ [(n =/= \text{zero}) \Rightarrow

\text{!(forall } ?x ?r . \ (\text{pap}_1 ?r \ ?x \ n) = ?r + ?x + \text{two} \ n) \ [\text{PC0}]\]))

\text{! (\text{val-of fpp}_2\text{-correctness}) =>

\text{let} (\_ := \(!\text{prove Power.right-two}));

\_ := \(!\text{prove Power.right-times});

\_ := \(!\text{prove pap}_1\text{-correctness})

\text{pick-any} \ n \ x

\text{assume} (n =/= \text{zero})

\text{! (\text{chain} \ [(pap}_1 \ r \ (x + x) \ n)) \ [\text{fpp-nonzero}]

\[
= \text{pap}_1 \ r \ ((x + x) (\text{two} \ n)) \ [\text{Power.right-two}]
\]

\[
= \text{pap}_1 \ r \ ((\text{power-constant}) \ (\text{two} \ n)) \ [\text{Power.right-nonzero}]
\]

\[
= \text{pap}_1 \ r \ ((\text{power-constant}) \ (\text{two} \ n) + \text{one}) \ [\text{N.Plus.right-one}]\)
\]

\text{! (\text{val-of fpp}_1\text{-correctness}) =>

\text{let} (\text{theorem} := \text{(adapt theorem)});

\_ := \(!\text{prove Power.right-times});

\_ := \(!\text{prove Power.right-plus});

\_ := \(!\text{prove Power.right-one});

\_ := \(!\text{prove fpp}_2\text{-correctness})


lib/main/fast-power.ath

(!strong-induction.principle theorem

method (n)
assume ind-hyp := (strong-induction.hypothesis theorem' n)
conclude (strong-induction.conclusion theorem' n)
assume (n /= zero)
pick-any x
(!two-cases
  assume (even n)
  let {fact1 := ((forall ?x . (fpp_1 ?x (half n)) =
      ?x ++ (half n)))
    := (!chain-> [(n /= zero)
      => ((half n) N.< n) [N.half.less]
      => fact1 [ind-hyp]]);
    := (!chain->
      [n /= zero & even n]
      => (half n /= zero)
      [N.EO.half-nonzero-if-nonzero-even]]);
  fact2 := (forall ?x . (fpp_1 ?x (half n)) =
      ?x ++ (half n));
  := (!chain->
    [(half n) /= zero] => fact2 [fact1])}
(!chain
  (fpp_1 x n)
  = (fpp_1 (x + x) half n) [fpp-even]
  = ((x + x) ++ half n) [fact2]
  = ((x + two) ++ half n) [Power.right-two]
  = (x ++ (two * half n)) [Power.right-times]
  = (x ++ n) [N.EO.even-definition]]
assume (~ even n)
let {_: (!chain->
  (~ even n)
  => (odd n) [N.EO.odd-if-not-even]])
(!two-cases
  assume ((half n) = zero)
  let {_: conclude (n = one)
    (chain->
      [((half n) = zero)
      => (n = zero | n = one) [N.half.equal-zero]
      => (n = one) [(dsyl with (n /= zero)])]]
    (chain
      (fpp_1 x n)
      = (fpp_2 x half n) [fpp-odd]
      = x [fpp-zero]
      = (x ++ one) [Power.right-one]
      = (x ++ n) [(n = one)])
  assume ((half n) /= zero)
  (chain
    (fpp_1 x n)
    = (fpp_2 x (half n)) [fpp-odd]
    = (x ++ (two * (half n) +' one)) [fpp_2-correctness]
    = (x ++ n) [N.EO.odd-definition]])}
| (val-of fpp_1-correctness) =>
let (FPC0 := (!prove fpp_1-correctness0))
pick-any n x
assume (n /= zero)
let {C := (chain->
  [(n /= zero)
  => (forall ?x . (fpp_1 ?x n) = ?x ++ n) [FPC0]])
  (chain [(fpp_1 x n) = (x ++ n) [C]])}
| (val-of correctness) =>
let (FPP1 := (!prove fpp_1-correctness))
pick-any n x
(!two-cases
  assume (n = zero)
  (chain [(fast-power x n)
    = <0> [right-zero]
    = (x ++ zero) [Power.right-zero]
    = (x ++ n) [(n = zero)]])
assume (n /= zero)
lib/main/fast-power.ath

(!chain [(fast-power x n)
  = (fpp_1 x n) [right-nonzero]
  = (x ** n) [FPP1]])

} # match

(add-theorems theory |{theorems := proofs}|

| # fast-power
| # Monoid