lib/main/fast-power.ath

```
# fast-power, a function that computes x +* n with lg 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.

#--------------------------------------------------------------------------
load "nat-half"
load "power"
load "strong-induction"
#--------------------------------------------------------------------------
extend-module Monoid {

declare pap_1, pap_2: (T) [T T N] -> T
# Together these functions implement a recursive version of the
# power-accumulate-positive iterative function in Elements of
# Programming, p. 41.

declare fpp_1, fpp_2: (T) [T N] -> T
# Together these functions implement a recursive version of the
# iterative function power (first overloading) in Elements of
# Programming, p. 41.

declare fast-power: (T) [T N] -> T
# fast-power is the same as the function power (second overloading) in
# Elements of Programming, p. 42.

 module fast-power {
  define [+'   *   half   even   odd   one   two] :=
         [N.+ N.* N.half N.even N.odd N.one N.two]

  define [r a x n] := [?r:'S ?a:'S ?x:'S ?n:N]

  define axioms :=
   (fun
    [(fast-power a n) =
         [<0>                         when (n = zero)      # right-zero
          (fpp_1 a n)                 when (n =/= zero)]   # right-nonzero

     (fpp_1 a n) =
         [(fpp_1 (a + a) (half n))    when (even n)        # fpp-even
          (fpp_2 a (half n))          when (~ even n)]     # fpp-odd
    (fpp_2 a n) =
         [a                           when (n = zero)      # fpp-zero
          (pap_1 a (a + a) n)         when (n =/= zero)]   # fpp-nonzero

    (pap_1 r a n) =
         [(pap_2 (r + a) a n)         when (odd n)         # pap-odd
          (pap_1 r (a + a) (half n))  when (~ odd n)]      # pap-even
    (pap_2 r a n) =
         [r                           when (n = one)       # pap-one
          (pap_1 r (a + a) (half n))  when (n =/= one)]]) # pap-not-zone

  define [right-zero right-nonzero fpp-even fpp-odd fpp-zero fpp-nonzero
          pap-odd pap-even pap-one pap-not-one] := axioms

  (add-axioms theory axioms)

#.........................................................................
define pap_1-correctness0 :=
   (forall n . n =/= zero ==> (forall x r . (pap_1 r x n) = r + x +* n))

define pap_1-correctness :=
   (forall n x r . n =/= zero ==> (pap_1 r x n) = r + x +* n)

define fpp_1-correctness0 :=
  (forall n .  n =/= zero ==> (forall x . (fpp_1 x n) = x +* n))
```

```

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 proofs :=
 method (theorem adapt)
  let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
       [+ <0> +*] := (adapt [+ <0> +*]);
       [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 (S zero))
                         <==> (odd n)              [(n = one)
                                                    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 +* n)
                     = (r + x +* one)              [(n = 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)       [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 = one)
                             (!absurd (n = one) (n =/= one))));
                  _ := (!chain-> [D ==> fact1b  [fact1a]]);
                  _ := conclude fact2
                         pick-any r
                           (!chain
                            [(pap_1 r (x + x) (half n))
                             = (r + (x + x) ++ (half n))     [fact1b]
                             = (r + (x ++ two) ++ (half n)) [PR2]
                             = (r + x ++ (two * (half n)))   [PRT]])}
             (!two-cases
               assume (even n)
                 pick-any r
                   let {F := (!chain->
                              [(even n)
                           ==> (~ odd n)    [N.EO.not-odd-if-even]])}
                   (!chain
                    [(pap_1 r x n)
                     = (pap_1 r (x + x) (half n))    [pap-even F]
                     = (r + x ++ (two * (half n)))   [fact2]
                     = (r + x ++ n)         [N.EO.even-definition]])
               assume (~ even n)
                 pick-any r
                   let {G := (!chain->
                              [(~ even n) ==> (odd n)
                                           [N.EO.odd-if-not-even]])}
                   (!chain
                    [(pap_1 r x n)
                     = (pap_2 (r + x) x n) [pap-odd G]
                     = (pap_1 (r + x) (x + x) (half n))
                                           [pap-not-one]
                     = ((r + x) + x ++ (two * (half n)))
                                           [fact2]
                     = (r + (x + x ++ (two * (half n))))
                                           [associative]
                     = (r + (x ++ one + x ++ (two * (half n))))
                                           [PR1]
                     = (r + (x ++ (one +' two * (half n))))
                                           [PRP]
                     = (r + (x ++ (two * (half n) +' one)))
                                           [N.Plus.commutative]
                     = (r + x ++ n)         [N.EO.odd-definition]])))))
| (val-of pap_1-correctness) =>
  let {PC0 := (!prove pap_1-correctness0)}
  pick-any n x r
   assume (n =/= zero)
     let {i := (!chain-> [(n =/= zero) ==>
                          (forall ?x ?r .
                             (pap_1 ?r ?x n) = ?r + ?x ++ n)
                                           [PC0]])}
     (!chain [(pap_1 r x n) = (r + x ++ n) [i]])
| (val-of fpp_2-correctness) =>
  let {_ := (!prove Power.right-two);
       _ := (!prove Power.right-times);
       _ := (!prove pap_1-correctness)}
  pick-any n x
    assume (n =/= zero)
      (!chain [(fpp_2 x n)
             = (pap_1 x (x + x) n)          [fpp-nonzero]
             = (x + ((x + x) ++ n))         [pap_1-correctness]
             = (x + ((x ++ two) ++ n))      [Power.right-two]
             = (x + (x ++ (two * n)))       [Power.right-times]
             = (x ++ (S (two * n)))         [Power.right-nonzero]
             = (x ++ (two * n +' one))      [N.Plus.right-one]])
 | (val-of fpp_1-correctness0) =>
   let {theorem' := (adapt theorem);
        _ := (!prove Power.right-times);
        _ := (!prove Power.right-plus);
        _ := (!prove Power.right-one);
        _ := (!prove fpp_2-correctness)}
```

```
      (!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 := ((half n) =/= zero ==>
                               (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)
```

```
           (!chain [(fast-power x n)
                    = (fpp_1 x n)                        [right-nonzero]
                    = (x +* n)                           [FPP1]]))
    } # match

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

} # fast-power
} # Monoid
```