lib/main/nat-fast-power2.ath

```
# This version of fast-power still uses embedded recursion but
# eliminates one multiplication by inserting a test for n = one.  An
# optimization?  Not if multiplication is a fixed-cost operation, since
# the extra test doubles the number of test instructions.

#--------------------------------------------------------------------------
load "nat-power.ath"
load "nat-half.ath"
load "strong-induction.ath"
#--------------------------------------------------------------------------

extend-module N {
declare fast-power: [N N] -> N
module fast-power {
define [x r m n] := [?x:N ?r:N ?m:N ?n:N]

assert axioms :=
 (fun
  [(fast-power x n) =
    [one                        when (n = zero)
     x                          when (n = one)
     (square (fast-power x (half n)))
                                when (n =/= zero & n =/= one & Even n)
     ((square (fast-power x (half n))) * x)
                                when (n =/= zero & n =/= one & ~ Even n)]])
define [if-zero if-one nonzero-nonone-even nonzero-nonone-odd] := axioms
#--------------------------------------------------------------------------
define nonzero-even :=
   (forall x n .
     (n =/= zero & Even n) ==>
     (fast-power x n) = square (fast-power x (half n)))
define nonzero-odd :=
   (forall x n .
     (n =/= zero & ~ Even n) ==>
     (fast-power x n) = (square (fast-power x (half n))) * x)

conclude nonzero-even
  pick-any x n
    assume (n =/= zero & (Even n))
      (!two-cases
        assume (n = one)
          (!from-complements
           ((fast-power x n) = (square (fast-power x (half n))))
           (Even n)
           (!chain->
            [(Odd (S zero))
             ==> (Odd n)      [(n = one) one-definition]
             ==> (~ (Even n)) [EO.not-Even-if-Odd]]))
       assume (n =/= one)
         (!chain
          [(fast-power x n) = (square (fast-power x (half n)))
               [nonzero-nonone-even]]))

conclude nonzero-odd
  pick-any x n
    assume (n =/= zero & ~ (Even n))
      (!two-cases
        assume (n = one)
         (!combine-equations
          (!chain [(fast-power x n) --> x  [if-one]])
          (!chain [((square (fast-power x (half n))) * x)
                   --> ((square (fast-power x zero)) * x)
                      [(n = one) one-definition half.if-one]
                   --> ((square one) * x)  [if-zero]
                   --> x  [square.definition Times.left-one]]))
        assume (n =/= one)
          (!chain
```

```
           [(fast-power x n) --> ((square (fast-power x (half n))) * x)
              [nonzero-nonone-odd]]))

#.....................................................................
# Now the same proof as given in nat-fast-power.ath works to prove:

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

define step :=
 method (n)
  assume ind-hyp :=
          (forall ?m . ?m < n ==> (forall ?x . (fast-power ?x ?m) = ?x ** ?m))
    conclude (forall ?x . (fast-power ?x n) = ?x ** n)
      pick-any x
        (!two-cases
          assume (n = zero)
            (!chain [(fast-power x n)
                     --> one           [if-zero]
                     <-- (x ** zero)   [Power.if-zero]
                     <-- (x ** n)      [(n = zero)]])
          assume (n =/= zero)
            let {fact1 := conclude goal :=
                              (forall ?x .
                                (fast-power ?x (half n)) = ?x ** (half n))
                             (!chain-> [(n =/= zero)
                                    ==> ((half n) < n) [half.less]
                                    ==> goal           [ind-hyp]]);
                 fact2 := conclude
                             ((square (fast-power x (half n))) =
                              x ** (two * (half n)))
                            (!chain
                             [(square (fast-power x (half n)))
                              --> (square (x ** (half n)))     [fact1]
                              --> (x ** (half n) * x ** (half n))
                                                               [square.definition]
                              <-- (x ** ((half n) + (half n))) [Power.Plus-case]
                              <-- (x ** (two * (half n)))      [Times.two-times]])}
            (!two-cases
              assume (Even n)
                (!chain
                 [(fast-power x n)
                  --> (square (fast-power x (half n)))
                                              [nonzero-even]
                  --> (x ** (two * (half n))) [fact2]
                  --> (x ** n)                [EO.Even-definition]])
              assume (~ (Even n))
                let {_ := (!chain-> [(~ (Even n))
                                 ==> (Odd n)     [EO.Odd-if-not-Even]])}
                (!chain
                 [(fast-power x n)
                  --> ((square (fast-power x (half n))) * x)
                                              [nonzero-odd]
                  --> ((x ** (two * (half n))) * x)  [fact2]
                  <-- ((x ** (two * (half n))) * (x ** one))
                                                   [Power.right-one]
                  <-- (x ** ((two * (half n)) + one))   [Power.Plus-case]
                  --> (x ** n)                 [EO.Odd-definition]]))))

conclude correctness
  (!strong-induction.principle correctness step)
} # fast-power
} # N
```