lib/main/nat-fast-power3.ath

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

extend-module N {
declare fast-power-accumulate: [N N N] -> N
module fast-power-accumulate {
define fpa := fast-power-accumulate
define [r x n] := [?r ?x ?n]
assert axioms :=
  (fun
    [(fpa r x n) =
      [r                              when (n = zero)
       (fpa r (x * x) (half n))       when (n =/= zero & Even n)
       (fpa (r * x) (x * x) (half n))  when (n =/= zero & ~ Even n)]])
define [if-zero nonzero-even nonzero-odd] := axioms

define correctness := (forall n r x . (fpa r x n) = r * x ** n)

define fpa-step :=
 method (n)
  assume ind-hyp :=
          (forall ?m . ?m < n ==>
                       (forall ?r ?x . (fpa ?r ?x ?m) = ?r * ?x ** ?m))
    conclude (forall ?r ?x . (fpa ?r ?x n) = ?r * ?x ** n)
      pick-any r:N x:N
        (!two-cases
          assume (n = zero)
           (!combine-equations
            (!chain [(fpa r x n)
                     --> r                  [if-zero]])
            (!chain [(r * x ** n)
                     --> (r * x ** zero)    [(n = zero)]
                     --> (r * one)          [Power.if-zero]
                     --> r                  [Times.right-one]]))
          assume (n =/= zero)
            let {fact := conclude goal :=
                           (forall ?r ?x .
                             (fpa ?r ?x (half n)) = ?r * ?x ** half n)
                          (!chain-> [(n =/= zero)
                                     ==> (half n < n) [half.less]
                                     ==> goal         [ind-hyp]])}
            (!two-cases
              assume (Even n)
                (!combine-equations
                 (!chain
                  [(fpa r x n)
                   --> (fpa r (x * x) (half n))     [nonzero-even]
                   --> (r * (x * x) ** half n)      [fact]])
                 (!chain
                  [(r * x ** n)
                   <-- (r * x ** (two * half n))  [EO.Even-definition]
                   --> (r * (x ** two) ** half n) [Power.right-times]
                   --> (r * (x * x) ** half n)    [Power.right-two]]))
              assume (~ (Even n))
                let {_ := (!chain-> [(~ (Even n))
                                     ==> (Odd n) [EO.Odd-if-not-Even]])}
                (!combine-equations
                 (!chain
                  [(fpa r x n)
                   --> (fpa (r * x) (x * x) half n)   [nonzero-odd]
                   --> ((r * x) * (x * x) ** half n)  [fact]
                   --> (r * x * (x * x) ** half n)  [Times.associative]])
                 (!chain
                  [(r * x ** n)
```

```
                    <-- (r * x ** (two * (half n) + one))
                                                        [EO.Odd-definition]
                    --> (r * x ** (S (two * half n)))   [Plus.right-one]
                    --> (r * x * (x ** (two * half n))) [Power.if-nonzero]
                    --> (r * x * (x ** two) ** half n) [Power.right-times]
                    --> (r * x * (x * x) ** half n)  [Power.right-two]]))
            )

conclude correctness
  (!strong-induction.principle correctness fpa-step)
} # close module fast-power-accumulate

declare fast-power: [N N] -> N
module fast-power {
define fpa := fast-power-accumulate
assert definition := (fun [(fast-power x n) = (fpa one x n)])

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

conclude correctness
  pick-any n:N x:N
    (!chain [(fast-power x n)
             = (fpa one x n)   [definition]
             = (one * x ** n)  [fast-power-accumulate.correctness]
             = (x ** n)        [Times.left-one]])


} # close module fast-power


} # close module N
```