# 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.

 confessed

 extend-module N
 declare fast-power: [N N] -> N
 extend-module fast-power { 
 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)
 (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) [~ Even-n-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] 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] ]

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

 conclude correctness

 (!strong-induction.principle correctness
  (step fast-power if-zero nonzero-even nonzero-odd))

 # The proof for fast-power still works:

 conclude correctness

 (!strong-induction.principle correctness
  (step fast-power if-zero nonzero-even nonzero-odd))

 | # fast-power

 | # N