

lib/main/nat-fast-power2.ath

```

1  # This version of fast-power still uses embedded recursion but
2  # eliminates one multiplication by inserting a test for n = one. An
3  # optimization? Not if multiplication is a fixed-cost operation, since
4  # the extra test doubles the number of test instructions.
5
6  #-----
7  load "nat-power.ath"
8  load "nat-half.ath"
9  load "strong-induction.ath"
10 #-----
11
12 extend-module N {
13   declare fast-power: [N N] -> N
14   module fast-power {
15     define [x r m n] := [?x:N ?r:N ?m:N ?n:N]
16
17     assert axioms :=
18       (fun
19         [(fast-power x n) =
20           [one
21             x
22             (square (fast-power x (half n)))
23             when (n = zero)
24             when (n = one)
25             when (n /= zero & n /= one & Even n)
26             ((square (fast-power x (half n))) * x)
27             when (n /= zero & n /= one & ~ Even n)]])
28     define [if-zero if-one nonzero-nonone-even nonzero-nonone-odd] := axioms
29     #-----
30     define nonzero-even :=
31       (forall x n .
32         (n /= zero & Even n) ==>
33         (fast-power x n) = square (fast-power x (half n)))
34     define nonzero-odd :=
35       (forall x n .
36         (n /= zero & ~ Even n) ==>
37         (fast-power x n) = (square (fast-power x (half n))) * x)
38
39     conclude nonzero-even
40     pick-any x n
41     assume (n /= zero & (Even n))
42     (!two-cases
43       assume (n = one)
44       (!from-complements
45         ((fast-power x n) = (square (fast-power x (half n))))
46         (Even n)
47         (!chain->
48           [(Odd (S zero))
49            ==> (Odd n)
50            ==> (n = one) one-definition]
51           [(~ (Even n)) [EO.not-Even-if-Odd]]))
52       assume (n /= one)
53       (!chain
54         [(fast-power x n) = (square (fast-power x (half n)))
55          [nonzero-nonone-even]]))
56     conclude nonzero-odd
57     pick-any x n
58     assume (n /= zero & ~ (Even n))
59     (!two-cases
60       assume (n = one)
61       (!combine-equations
62         (!chain [(fast-power x n) --> x [if-one]])
63         (!chain [((square (fast-power x (half n))) * x)
64                  --> ((square (fast-power x zero)) * x)
65                  [(n = one) one-definition half.if-one]
66                  --> ((square one) * x) [if-zero]
67                  --> x [square.definition Times.left-one]]))
68       assume (n /= one)
69       (!chain

```

```

68      [(fast-power x n) --> ((square (fast-power x (half n))) * x)
69      [nonzero-nonone-odd]])
70
71 #.....
72 # Now the same proof as given in nat-fast-power.ath works to prove:
73
74 define correctness := (forall n x . (fast-power x n) = x ** n)
75
76 define step :=
77   method (n)
78     assume ind-hyp :=
79       (forall ?m . ?m < n ==> (forall ?x . (fast-power ?x ?m) = ?x ** ?m))
80     conclude (forall ?x . (fast-power ?x n) = ?x ** n)
81     pick-any x
82     (!two-cases
83       assume (n = zero)
84         (!chain [(fast-power x n)
85           --> one [if-zero]
86           <-- (x ** zero) [Power.if-zero]
87           <-- (x ** n) [(n = zero)]]
88       assume (n != zero)
89         let {fact1 := conclude goal :=
90           (forall ?x .
91             (fast-power ?x (half n)) = ?x ** (half n))
92           (!chain-> [(n != zero)
93             ==> ((half n) < n) [half.less]
94             ==> goal [ind-hyp]]);
95         fact2 := conclude
96           ((square (fast-power x (half n))) =
97            x ** (two * (half n)))
98           (!chain
99             [(square (fast-power x (half n)))
100              --> (square (x ** (half n))) [fact1]
101              --> (x ** (half n) * x ** (half n))
102               [square.definition]
103              <-- (x ** ((half n) + (half n))) [Power.Plus-case]
104              <-- (x ** (two * (half n))) [Times.two-times]]})
105         (!two-cases
106           assume (Even n)
107             (!chain
108               [(fast-power x n)
109                --> (square (fast-power x (half n)))
110                 [nonzero-even]
111                --> (x ** (two * (half n))) [fact2]
112                --> (x ** n) [EO.Even-definition]])
113           assume (~ (Even n))
114             let {_ := (!chain-> [(~ (Even n))
115               ==> (Odd n) [EO.Odd-if-not-Even]]})
116             (!chain
117               [(fast-power x n)
118                --> ((square (fast-power x (half n))) * x)
119                 [nonzero-odd]
120                --> ((x ** (two * (half n))) * x) [fact2]
121                <-- ((x ** (two * (half n))) * (x ** one))
122                 [Power.right-one]
123                <-- (x ** ((two * (half n)) + one)) [Power.Plus-case]
124                --> (x ** n) [EO.Odd-definition]]))
125         conclude correctness
126       (!strong-induction.principle correctness step)
127     } # fast-power
128 } # N
129 } # N

```