```
_{\rm I} # fast-power, a function that computes x +* n with lg n
2 # + operations, optimized to avoid the last doubling done by
  # a simpler algorithm when it's unnecessary. Based upon the power
4 # function developed in Stepanov & McJones, Elements of Programming,
5 # pp. 41-42.
8 load "nat-half"
9 load "power"
10 load "strong-induction"
11 #-----
12 extend-module Monoid {
14 declare pap_1, pap_2: (T) [T T N] -> T
15 # Together these functions implement a recursive version of the
16 # power-accumulate-positive iterative function in Elements of
17 # Programming, p. 41.
19 declare fpp_1, fpp_2: (T) [T N] -> T
20 # Together these functions implement a recursive version of the
_{21} # iterative function power (first overloading) in Elements of
22 # Programming, p. 41.
24 declare fast-power: (T) [T N] -> T
^{25} # fast-power is the same as the function power (second overloading) in
26 # Elements of Programming, p. 42.
27
28
   module fast-power {
   define [+' * half even odd one two] :=
29
           [N.+ N.* N.half N.even N.odd N.one N.two]
31
    define [r a x n] := [?r:'S ?a:'S ?x:'S ?n:N]
32
33
    define axioms :=
34
    (fun
35
      [(fast-power a n) =
36
          <0>]
37
                                        when (n = zero)
                                                            # right-zero
                                       when (n =/= zero)] # right-nonzero
            (fpp_1 a n)
38
40
      (fpp_1 a n) =
           [(fpp_1 (a + a) (half n))
                                       when (even n)
                                                           # fpp-even
41
            (fpp_2 a (half n))
                                       when (\sim even n)]
                                                           # fpp-odd
      (fpp_2 a n) =
43
                                        when (n = zero)
                                                            # fpp-zero
                                       when (n =/= zero)] # fpp-nonzero
45
            (pap_1 a (a + a) n)
46
      (pap_1 r a n) =
47
           [(pap_2 (r + a) a n)]
                                      when (odd n)
                                                            # pap-odd
48
            (pap_1 r (a + a) (half n)) when (\sim odd n)
                                                            # pap-even
      (pap_2 r a n) =
50
                                        when (n = one)
                                                         # pap-one
51
           ſr
            (pap_1 r (a + a) (half n)) when (n = /= one)]]) # pap-not-zone
52
53
    define [right-zero right-nonzero fpp-even fpp-odd fpp-zero fpp-nonzero
55
            pap-odd pap-even pap-one pap-not-one] := axioms
56
57
    (add-axioms theory axioms)
58
  define pap_1-correctness0 :=
     (forall n . n =/= zero ==> (forall x r . (pap_1 r x n) = r + x +* n))
62
63 define pap_1-correctness :=
64
    (forall n \times r . n = /= zero ==> (pap_1 r \times n) = r + x + * n)
65
66 define fpp_1-correctness0 :=
   (forall n . n =/= zero ==> (forall x . (fpp_1 x n) = x + * n)
```

2

```
define fpp_1-correctness :=
69
     (forall n x . n =/= zero ==> (fpp_1 x n) = x + n)
71
   define fpp_2-correctness :=
72
73
     (forall n x . n =/= zero ==> (fpp_2 x n) = x +* (two * n +' one))
74
  define correctness := (forall n x . (fast-power x n) = x + * n)
76
   define theorems := [pap_1-correctness0 pap_1-correctness fpp_2-correctness
77
78
                       fpp_1-correctness0 fpp_1-correctness correctness]
   #......
79
  define proofs :=
   method (theorem adapt)
81
82
     let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);</pre>
          [+ <0> +*] := (adapt [+ <0> +*]);
83
          [pap_1 pap_2 fpp_1 fpp_2 fast-power] :=
84
            (adapt [pap_1 pap_2 fpp_1 fpp_2 fast-power]);
          PR1 := (!prove Power.right-one);
86
          PR2 := (!prove Power.right-two);
87
          PRT := (!prove Power.right-times);
88
          PRP := (!prove Power.right-plus) }
89
     match theorem {
       (val-of pap_1-correctness0) =>
91
92
       let {theorem' := (adapt theorem) }
       (!strong-induction.principle theorem'
93
        method (n)
94
95
         assume ind-hyp := (strong-induction.hypothesis theorem' n)
          conclude (strong-induction.conclusion theorem' n)
96
           assume (n =/= zero)
97
             pick-any x
98
               (!two-cases
                 assume (n = one)
100
                   pick-any r
101
                     let {C := (!chain->
102
                                [(odd (S zero))
103
                             <==> (odd n)
                                                        [(n = one)]
                                                        N.one-definition]
105
                             ==> ((pap_1 r x n) =
106
107
                                  (pap_2 (r + x) x n)) [pap-odd]])}
                      (!combine-equations
108
                       (!chain
110
                       [(pap_1 r x n)
111
                         = (pap_2 (r + x) x n)
                        = (r + x)
112
                                                        [pap-one (n = one)])
                       (!chain
113
                        [(r + x + * n)
                         = (r + x +* one)
                                                        [(n = one)]
115
116
                        = (r + x)
                                                        [PR1]]))
                 assume (n =/= one)
117
                   let {fact1a := ((half n) =/= zero ==>
118
                                    (forall ?a ?r . (pap_1 ?r ?a (half n)) =
119
                                                    ?r + ?a +* (half n));
120
121
                         fact1b := (forall ?a ?r . (pap_1 ?r ?a (half n)) =
                                                    (?r + ?a +* (half n)));
122
                         fact2 := (forall ?r . (pap_1 ?r (x + x) (half n)) =
123
124
                                                (?r + x + * (two * (half n))));
                         _ := (!chain->
125
                               [(n = /= zero)]
126
                            ==> ((half n) N.< n)
                                                       [N.half.less]
127
                            ==> fact1a
                                                        [ind-hyp]]);
                         D := (!by-contradiction ((half n) =/= zero)
129
                               assume ((half n) = zero)
130
131
                                 let {E := (!chain->
                                            [((half n) = zero)]
132
133
                                         ==> (n = zero | n = one)
                                                        [N.half.equal-zero]])}
134
135
                                 (!cases E
                                  assume (n = zero)
136
                                    (!absurd (n = zero) (n =/= zero))
137
```

```
assume (n = one)
                                      (!absurd (n = one) (n =/= one))));
139
                          _ := (!chain-> [D ==> fact1b [fact1a]]);
                          _ := conclude fact2
141
                                 pick-any r
142
143
                                    (!chain
                                     [(pap_1 r (x + x) (half n))]
144
                                      = (r + (x + x) + \star (half n))
                                                                        [fact1b]
                                      = (r + (x +* two) +* (half n)) [PR2]
146
                                      = (r + x + * (two * (half n))) [PRT]])}
147
148
                     (!two-cases
                       assume (even n)
149
                         pick-any r
150
                           let {F := (!chain->
151
152
                                       [(even n)
                                    ==> (~ odd n)
                                                    [N.EO.not-odd-if-even]])}
153
                           (!chain
154
                            [(pap_1 r x n)
                              = (pap_1 r (x + x) (half n))
                                                               [pap-even F]
156
                             = (r + x + * (two * (half n)))
157
                                                               [fact2]
                             = (r + x + * n)
                                                   [N.EO.even-definition]])
158
159
                       assume (~ even n)
                         pick-any r
                           let {G := (!chain->
161
162
                                       [(\sim even n) ==> (odd n)
                                                     [N.EO.odd-if-not-even]])}
163
                           (!chain
164
165
                            [(pap_1 r x n)
                              = (pap_2 (r + x) x n) [pap-odd G]
166
                             = (pap_1 (r + x) (x + x) (half n))
167
                                                     [pap-not-one]
168
169
                             = ((r + x) + x + * (two * (half n)))
170
                                                     [fact2]
                             = (r + (x + x +* (two * (half n))))
171
172
                                                      [associative]
                             = (r + (x +* one + x +* (two * (half n))))
173
                                                      [PR1]
                              = (r + (x + * (one + 'two * (half n))))
175
                                                      [PRP]
176
                              = (r + (x +* (two * (half n) +' one)))
177
                                                      [N.Plus.commutative]
178
                             = (r + x + * n)
                                                      [N.EO.odd-definition]]))))
179
180
      | (val-of pap_1-correctness) =>
         let {PC0 := (!prove pap_1-correctness0)}
181
182
        pick-any n x r
          assume (n =/= zero)
183
184
            let {i := (!chain-> [(n =/= zero) ==>
                                   (forall ?x ?r .
185
186
                                      (pap_1 ?r ?x n) = ?r + ?x + * n)
                                                     [PC011)}
187
            (!chain [(pap_1 r x n) = (r + x + * n) [i]])
188
189
      | (val-of fpp_2-correctness) =>
         let {_ := (!prove Power.right-two);
190
              _ := (!prove Power.right-times);
191
               _ := (!prove pap_1-correctness) }
192
        pick-any n x
193
           assume (n =/= zero)
194
             (!chain [(fpp_2 x n)
195
                     = (pap_1 x (x + x) n)
                                                      [fpp-nonzero]
                     = (x + ((x + x) + * n))
197
                                                      [pap_1-correctness]
198
                     = (x + ((x + \star two) + \star n))
                                                      [Power.right-two]
199
                    = (x + (x + * (two * n)))
                                                      [Power.right-times]
                     = (x + * (S (two * n)))
                                                      [Power.right-nonzero]
200
                     = (x +* (two * n +' one))
201
                                                      [N.Plus.right-one]])
        | (val-of fpp_1-correctness0) =>
202
          let {theorem' := (adapt theorem);
               _ := (!prove Power.right-times);
204
               _ := (!prove Power.right-plus);
205
206
               _ := (!prove Power.right-one);
               _ := (!prove fpp_2-correctness) }
207
```

```
(!strong-induction.principle theorem'
           method (n)
209
            assume ind-hyp := (strong-induction.hypothesis theorem' n)
             conclude (strong-induction.conclusion theorem' n)
211
              assume (n =/= zero)
212
213
               pick-any x
                 (!two-cases
214
                  assume (even n)
                    let {fact1 := ((half n) =/= zero ==>
216
                                     (forall ?x . (fpp_1 ?x (half n)) =
217
                                                   ?x +* (half n));
218
                          _ := (!chain-> [(n =/= zero)
219
                                       ==> ((half n) N.< n)
                                                                    [N.half.less]
220
                                       ==> fact1
                                                                    [ind-hyp]]);
221
                          _ := (!chain->
222
                                 [(n =/= zero & even n)
223
                                  ==> (half n =/= zero)
224
225
                                          [N.EO.half-nonzero-if-nonzero-even]]);
                          fact2 := (forall ?x . (fpp_1 ?x (half n)) =
226
                                                  ?x +* (half n));
227
                          _ := (!chain->
228
                                [((half n) =/= zero) ==> fact2 [fact1]])}
229
                     (!chain
230
231
                      [(fpp_1 x n)
232
                       = (fpp_1 (x + x) half n)
                                                      [fpp-even]
                       = ((x + x) + * half n)
                                                      [fact2]
233
                                                     [Power.right-two]
                       = ((x +* two) +* half n)
234
235
                       = (x +* (two * half n))
                                                      [Power.right-times]
                                                      [N.EO.even-definition]])
                       = (x + * n)
236
                  assume (~ even n)
237
                     let {_ := (!chain->
238
                                [(~ even n)
                                  ==> (odd n)
                                                     [N.EO.odd-if-not-even]])}
240
                     (!two-cases
241
242
                       assume ((half n) = zero)
                         let {_ := conclude (n = one)
243
                                      (!chain->
                                       [((half n) = zero)]
245
                                    ==> (n = zero | n = one) [N.half.equal-zero]
246
                                    ==> (n = one) [(dsyl with (n =/= zero))]])}
247
                         (!chain [(fpp_1 x n)
248
                                   = (fpp_2 x half n)
                                                                [fpp-odd]
249
                                   = x
                                                               [fpp-zero]
250
                                                                [Power.right-one]
251
                                   = (x + * one)
                                   = (x + * n)
252
                                                                [(n = one)])
                       assume ((half n) =/= zero)
253
                         (!chain
                          [(fpp_1 x n)
255
                           = (fpp_2 x (half n))
                                                                [fpp-odd]
                           = (x + * (two * (half n) + ' one))
257
                                                                [fpp_2-correctness]
258
259
                           = (x + * n)
                                                        [N.EO.odd-definition]]))))
        | (val-of fpp_1-correctness) =>
260
261
          let {FPC0 := (!prove fpp_1-correctness0)}
         pick-any n x
262
            assume (n =/= zero)
263
              let {C := (!chain->
264
                          [(n = /= zero)]
265
                       ==> (forall ?x . (fpp_1 ?x n) = ?x +* n)
                                                                          [FPC0]])}
266
              (!chain [(fpp_1 x n) = (x +* n)]
267
                                                                           [C]])
        | (val-of correctness) =>
269
         let {FPP1 := (!prove fpp_1-correctness)}
270
         pick-any n x
271
            (!two-cases
             assume (n = zero)
272
               (!chain [(fast-power x n)
                         = < 0 >
                                                                [right-zero]
274
275
                         = (x + * zero)
                                                                [Power.right-zero]
                                                                [(n = zero)])
                         = (x + * n)
276
             assume (n =/= zero)
277
```