

lib/algebra/permutation.ath

```

1 # Permutation theory, basis for showing that permutations under composition
2 # form a group (for which see lib/algebra/permutation_unittest.ath).
3
4 load "algebra/function"
5
6 module Permutation {
7   open Function
8   domain (Perm D)
9
10  declare perm->fun: (D) [(Perm D)] -> (Fun D D)
11  declare fun->perm: (D) [(Fun D D)] -> (Perm D)
12
13  set-precedence (perm->fun fun->perm) 350
14
15  define [p q r f x y] := [?p:(Perm 'D1) ?q:(Perm 'D2) ?r:(Perm 'D3)
16                        ?f:(Fun 'D4 'D5) ?x ?y]
17
18  define is-bijective := (forall p . bijective perm->fun p)
19
20  define fun->fun := (forall p . fun->perm perm->fun p = p)
21
22  define perm->perm :=
23    (forall f . bijective f ==> perm->fun fun->perm f = f)
24
25  declare o: (D) [(Perm D) (Perm D)] -> (Perm D)
26  declare identity: (D) [] -> (Perm D)
27
28  define o' := Function.o
29  set-precedence o' (plus 10 (get-precedence perm->fun))
30
31  define identity' := Function.identity
32
33  define compose-definition :=
34    (forall p q . p o q = fun->perm (perm->fun p o' perm->fun q))
35
36  define identity-definition := (identity = fun->perm identity')
37
38  define theory :=
39    (make-theory ['Function]
40               [is-bijective fun->fun perm->perm
41                compose-definition identity-definition])
42
43  define associative := (forall p q r . (p o q) o r = p o (q o r))
44  define right-identity := (forall p . p o identity = p)
45  define left-identity := (forall p . identity o p = p)
46
47  define Monoid-theorems := [associative right-identity left-identity]
48
49  define [f->p p->f] := [fun->perm perm->fun]
50
51  define proofs :=
52    method (theorem adapt)
53      let {[_ prove chain chain-> _] := (proof-tools adapt theory);
54          [identity' o' at identity o fun->perm perm->fun] :=
55            (adapt [identity' o' at identity o fun->perm perm->fun]);
56          [id cd] := [identity-definition compose-definition]}
57      match theorem {
58        (val-of associative) =>
59          let {CA := (!prove Function.associative);
60              CBP := (!prove compose-bijective-preserving)}
61          pick-any p:(Perm 'S) q:(Perm 'S) r:(Perm 'S)
62            let {_ := (!chain-> [true
63                              ==> (bijective perm->fun q) [is-bijective]]);
64                _ := (!chain->
65                      [true
66                       ==> (bijective perm->fun p)           [is-bijective]
67                       ==> (bijective perm->fun p &

```

```

68         bijective perm->fun q)          [augment]
69     ==> (bijective
70         (perm->fun p o' perm->fun q)) [CBP]);
71     _ := (!chain->
72         [true
73         ==> (bijective perm->fun r)      [is-bijective]
74         ==> (bijective perm->fun q &
75             bijective perm->fun r)      [augment]
76         ==> (bijective
77             (perm->fun q o' perm->fun r)) [CBP]]})
78     (!combine-equations
79     (!chain
80     [(p o q) o r)
81     --> ((fun->perm (perm->fun p o' perm->fun q)) o r)    [cd]
82     --> (fun->perm
83         ((perm->fun fun->perm
84             (perm->fun p o' perm->fun q)) o' perm->fun r)) [cd]
85     --> (fun->perm
86         ((perm->fun p o' perm->fun q) o' perm->fun r)) [perm->perm]
87     --> (fun->perm
88         (perm->fun p o' (perm->fun q o' perm->fun r))) [CA]))
89     (!chain
90     [(p o (q o r))
91     --> (p o (fun->perm (perm->fun q o' perm->fun r)))    [cd]
92     --> (fun->perm
93         (perm->fun p o'
94             (perm->fun
95                 fun->perm
96                 (perm->fun q o' perm->fun r))))          [cd]
97     --> (fun->perm
98         (perm->fun p o' (perm->fun q o' perm->fun r)))    [perm->perm]
99     ]))
100 | (val-of right-identity) =>
101     let {RI := (!prove Function.right-identity);
102         IB := (!prove identity-bijective)}
103     pick-any p:(Perm 'S)
104     (!chain
105     [(p o identity)
106     --> (p o (fun->perm identity'))                    [IB id]
107     --> (fun->perm (perm->fun p o'
108                 perm->fun fun->perm identity')) [cd]
109     --> (fun->perm (perm->fun p o' identity'))          [perm->perm]
110     --> (fun->perm (perm->fun p))                      [RI]
111     --> p                                             [fun->fun]])
112 | (val-of left-identity) =>
113     let {LI := (!prove Function.left-identity);
114         IB := (!prove identity-bijective)}
115     pick-any p:(Perm 'S)
116     (!chain
117     [(identity o p)
118     --> ((fun->perm identity') o p)                    [IB id]
119     --> (fun->perm ((perm->fun fun->perm identity')
120                 o' (perm->fun p)))                    [cd]
121     --> (fun->perm (identity' o' (perm->fun p)))        [perm->perm]
122     --> (fun->perm (perm->fun p))                      [LI]
123     --> p                                             [fun->fun]])
124 }
125
126 (add-theorems theory |{Monoid-theorems := proofs}|)
127 } # close module Permutation
128
129 extend-module Permutation {
130     declare at: (D) [(Perm D) D] -> D
131
132     define at' := Function.at
133
134     define at-definition := (forall p x . p at x = (perm->fun p) at' x)
135
136     declare inverse: (D) [(Perm D)] -> (Perm D)
137

```

```

138 define inverse-definition :=
139   (forall p x y . p at x = y ==> (inverse p) at y = x)
140
141 declare div: (D) [(Perm D) (Perm D)] -> (Perm D)
142
143 define div-definition := (forall p q . p div q = p o inverse q)
144
145 (add-axioms theory [at-definition inverse-definition div-definition])
146
147 define consistent-inverse :=
148   (forall p x x' y . p at x = y & p at x' = y ==> x = x')
149
150 define right-inverse-lemma :=
151   (forall p . (perm->fun p) o' (perm->fun inverse p) = identity')
152
153 define right-inverse := (forall p . p o inverse p = identity)
154
155 define Inverse-theorems :=
156   [consistent-inverse right-inverse-lemma right-inverse]
157
158 define [bij-def inj-def] := [bijective-definition injective-definition]
159 define at-def := at-definition
160
161 define proofs :=
162   method (theorem adapt)
163     let {[_ prove chain chain-> _] := (proof-tools adapt theory);
164         [at' identity' o' at identity o fun->perm perm->fun inverse] :=
165         (adapt [at' identity' o' at identity o fun->perm perm->fun
166               inverse]);
167         [cd bid] := [compose-definition bijective-definition]}
168     match theorem {
169       (val-of consistent-inverse) =>
170         pick-any p x x' y
171         let {inj := (!chain->
172                   [true
173                    ==> (bijective perm->fun p)           [is-bijective]
174                    ==> (injective perm->fun p)           [bij-def]])}
175           assume (p at x = y & p at x' = y)
176           let {p1 := (!chain->
177                     [(p at x) = y                       [(p at x = y)]
178                      = (p at x')                       [(p at x' = y)]
179                     ==> ((perm->fun p) at' x =
180                          (perm->fun p) at' x')         [at-def]]);
181                p2 := (!chain->
182                       [inj
183                        ==> (forall x x' .
184                             (perm->fun p) at' x =
185                             (perm->fun p) at' x'
186                             ==> x = x')                 [inj-def]]]}
187             (!chain-> [p1 ==> (x = x')                    [p2]])
188         | (val-of right-inverse-lemma) =>
189           pick-any p
190           let {surj :=
191                 (!chain->
192                   [true
193                    ==> (bijective (perm->fun p))         [is-bijective]
194                    ==> (surjective (perm->fun p))         [bid]
195                    ==> (forall y .
196                          exists x .
197                          (perm->fun p) at' x = y)       [surjective-definition]]);
198                 f := ((perm->fun p) o' (perm->fun inverse p));
199                 all-y :=
200                 conclude (forall y . f at' y = identity' at' y)
201                 pick-any y
202                 pick-witness x for (!instance surj y) witnessed
203                 (!chain
204                  [(f at' y)
205                   <-- (f at' ((perm->fun p) at' x))      [witnessed]
206                   --> ((perm->fun p) at'
207                       (perm->fun inverse p) at'

```

```

208             ((perm->fun p) at' x)))
209             [Function.compose-definition]
210     <-- (p at ((inverse p) at (p at x))) [at-definition]
211     --> (p at x) [inverse-definition]
212     --> ((perm->fun p) at' x) [at-definition]
213     --> y [witnessed]
214     <-- (identity' at' y) [Function.identity-definition]])}
215     (!chain-> [all-y ==> (f = identity') [function-equality]])
216 | (val-of right-inverse) =>
217 let {RIL := (!prove right-inverse-lemma)}
218     pick-any p
219         (!chain [(p o inverse p)
220                 --> (fun->perm
221                     ((perm->fun p) o'
222                      (perm->fun inverse p))) [cd]
223                 --> (fun->perm identity') [RIL]
224                 <-- identity [identity-definition]])]
225 }
226
227 (add-theorems theory |{Inverse-theorems := proofs}|)
228 } # close module Permutation

```