

lib/algebra/function.ath

```

1 # Theory of functions with axioms for defining application and composition, and
2 # theorems about surjective, injective, and bijective properties.
3
4 module Function {
5   domain (Fun Domain Codomain)
6   declare at: (C, D) [(Fun D C) D] -> C
7   declare identity: (D) [] -> (Fun D D)
8   declare o: (D, C, B) [(Fun C B) (Fun D C)] -> (Fun D B)
9
10  set-precedence o (plus 10 (get-precedence at))
11
12  define [f g h x x' y] := [?f ?g ?h ?x ?x' ?y]
13
14  define identity-definition := (forall x . identity at x = x)
15
16  define compose-definition := (forall f g x . (f o g) at x = f at (g at x))
17
18  define function-equality :=
19    (forall f g . f = g <==> forall x . f at x = g at x)
20
21  define theory :=
22    (make-theory [] [identity-definition compose-definition function-equality])
23
24  define associative := (forall f g h . (f o g) o h = f o (g o h))
25  define right-identity := (forall f . f o identity = f)
26  define left-identity := (forall f . identity o f = f)
27  define Monoid-theorems := [associative right-identity left-identity]
28
29  define proofs :=
30    method (theorem adapt)
31      let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
32          [at identity o] := (adapt [at identity o]);
33          [cd id] := [compose-definition identity-definition]}
34      match theorem {
35        (val-of associative) =>
36          pick-any f g h
37            let {all-x := pick-any x
38                (!chain
39                  [((f o g) o h at x)
40                   --> ((f o g) at h at x) [cd]
41                   --> (f at g at h at x) [cd]
42                   <-- (f at (g o h) at x) [cd]
43                   <-- ((f o (g o h)) at x) [cd])]}
44              (!chain-> [all-x
45                        ==> ((f o g) o h = f o (g o h)) [function-equality]])
46        | (val-of right-identity) =>
47          pick-any f
48            let {all-x := pick-any x
49                (!chain
50                  [((f o identity) at x)
51                   --> (f at (identity at x)) [cd]
52                   --> (f at x) [id])]}
53              (!chain->
54                [all-x ==> (f o identity = f) [function-equality]])
55        | (val-of left-identity) =>
56          pick-any f
57            let {all-x := pick-any x
58                (!chain
59                  [((identity o f) at x)
60                   --> (identity at (f at x)) [cd]
61                   --> (f at x) [id])]}
62              (!chain->
63                [all-x ==> (identity o f = f) [function-equality]])
64      }
65
66  (add-theorems theory |[Monoid-theorems := proofs]|)
67

```

```

68 #.....
69
70 declare surjective, injective, bijective: (D, C) [(Fun D C)] -> Boolean
71
72 define surjective-definition :=
73   (forall f . surjective f <==> forall y . exists x . f at x = y)
74
75 define injective-definition :=
76   (forall f . injective f <==> forall x y . f at x = f at y ==> x = y)
77
78 define bijective-definition :=
79   (forall f . bijective f <==> surjective f & injective f)
80
81 (add-axioms theory [surjective-definition injective-definition
82                   bijective-definition])
83
84 define identity-surjective := (surjective identity)
85
86 define identity-injective := (injective identity)
87
88 define identity-bijective := (bijective identity)
89
90 define compose-surjective-preserving :=
91   (forall f g . surjective f & surjective g ==> surjective f o g)
92
93 define compose-injective-preserving :=
94   (forall f g . injective f & injective g ==> injective f o g)
95
96 define compose-bijective-preserving :=
97   (forall f g . bijective f & bijective g ==> bijective f o g)
98
99 define Inverse-theorems :=
100 [identity-surjective identity-injective identity-bijective
101  compose-surjective-preserving compose-injective-preserving
102  compose-bijective-preserving]
103
104 # Proofs of first and fourth:
105
106 define proofs-1 :=
107   method (theorem adapt)
108     let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
109         [at identity o] := (adapt [at identity o]);
110         [cd id] := [compose-definition identity-definition]}
111     match theorem {
112       (val-of identity-surjective) =>
113         let {SDI := (!instance surjective-definition [identity]);
114             all-y :=
115               pick-any y
116                 (!chain->
117                   [(identity at y) --> y           [id]
118                    ==> (exists x . identity at x = y) [existence]])}
119             (!chain-> [all-y ==>
120                       (surjective identity)         [SDI]])
121       | (val-of compose-surjective-preserving) =>
122         pick-any f g
123         assume (surjective f & surjective g)
124         let {f-case :=
125             (!chain->
126               [(surjective f)
127                ==> (forall y .
128                    exists x . f at x = y)           [surjective-definition]])}
129             g-case :=
130             (!chain->
131               [(surjective g)
132                ==> (forall y .
133                    exists x . g at x = y)           [surjective-definition]])}
134             all-y :=
135               pick-any y
136                 let {f-case-y :=
137                     (!chain->

```

```

138         [true
139         ==> (exists y' .
140             f at y' = y)      [f-case]]})
141     pick-witness y' for f-case-y
142     let {g-case-y' :=
143         (!chain->
144             [true
145             ==> (exists x .
146                 g at x = y')    [g-case]])}
147     pick-witness x for g-case-y'
148     (!chain->
149         [(f o g at x)
150         --> (f at g at x)        [cd]
151         --> (f at y')           [(g at x = y')]
152         --> y                    [(f at y' = y)]
153         ==> (exists x .
154             f o g at x = y)     [existence]])}
155
156     (!chain-> [all-y
157         ==> (surjective f o g)   [surjective-definition]])
158 }
159
160
161 (add-theorems theory |{[identity-surjective
162     compose-surjective-preserving] := proofs-1}|)
163
164 define proofs :=
165 method (theorem adapt)
166 let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
167     [at identity o] := (adapt [at identity o]);
168     [cd id] := [compose-definition identity-definition]}
169 match theorem {
170     (val-of identity-injective) =>
171     let {IDI := (!instance injective-definition [identity]);
172         all-xx' :=
173         pick-any x x'
174         assume A := ((identity at x) = (identity at x'))
175         (!chain
176             [x <-- (identity at x)      [id]
177             --> (identity at x')      [A]
178             --> x'                     [id]])}
179     (!chain-> [all-xx' ==> (injective identity) [IDI]])
180 | (val-of identity-bijective) =>
181     let {BDI := (!instance bijective-definition [identity]);
182         s-and-i := (!both (!prove identity-surjective)
183                     (!prove identity-injective))}
184     (!chain->
185         [s-and-i ==> (bijective identity)      [BDI]])
186 }
187
188 (add-theorems theory |{[identity-injective identity-bijective] := proofs}|)
189
190 define proof :=
191 method (theorem adapt)
192 let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
193     [at identity o] := (adapt [at identity o]);
194     [cd id] := [compose-definition identity-definition]}
195 match theorem {
196     (val-of compose-injective-preserving) =>
197     let {indef := injective-definition}
198     pick-any f g
199     assume (injective f & injective g)
200     let {f-case := (!chain->
201         [(injective f)
202         ==> (forall x x' . f at x = f at x'
203             ==> x = x') [indef]])}
204     g-case := (!chain->
205         [(injective g)
206         ==> (forall x x' . g at x = g at x'
207             ==> x = x') [indef]])}

```

```

208     all-xx' :=
209     pick-any x x'
210     assume A := ((f o g) at x = (f o g) at x')
211     let {B := conclude (f at (g at x) =
212                       f at (g at x'))
213           (!chain
214             [(f at (g at x))
215              <-- ((f o g) at x) [cd]
216               --> ((f o g) at x') [A]
217               --> (f at (g at x')) [cd]]])
218           (!chain->
219             [B ==> (g at x = g at x') [f-case]
220              ==> (x = x') [g-case]])}
221
222     (!chain-> [all-xx' ==> (injective f o g) [indef]])
223   }
224
225 (add-theorems theory |{compose-injective-preserving := proof}|)
226
227 define proof :=
228   method (theorem adapt)
229     let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
230         [at identity o] := (adapt [at identity o]);
231         [cd id] := [compose-definition identity-definition]}
232     match theorem {
233       (val-of compose-bijective-preserving) =>
234       pick-any f:(Fun 'S 'T) g:(Fun 'U 'S)
235       assume bfg := (bijective f & bijective g)
236       let {f-s&i := (!chain-> [(bijective f) ==>
237                               (surjective f & injective f)
238                               [bijective-definition]]);
239           g-s&i := (!chain-> [(bijective g) ==>
240                               (surjective g & injective g)
241                               [bijective-definition]]);
242           f&g-s := (!both (!left-and f-s&i) (!left-and g-s&i));
243           f&g-i := (!both (!right-and f-s&i) (!right-and g-s&i));
244           csp := (!prove compose-surjective-preserving);
245           cip := (!prove compose-injective-preserving);
246           cs&i :=
247             (!both
248              (!chain-> [f&g-s ==> (surjective f o g) [csp]])
249              (!chain-> [f&g-i ==> (injective f o g) [cip]]))
250           (!chain-> [cs&i ==> (bijective f o g)
251                     [bijective-definition]])
252     }
253 (add-theorems theory |{compose-bijective-preserving := proof}|)
254 } # close module Function

```