# lib/algebra/function.ath

```
# Theory of functions with axioms for defining application and composition, and
# theorems about surjective, injective, and bijective properties.

module Function {
domain (Fun Domain Codomain)
declare at: (C, D) [(Fun D C) D] -> C
declare identity: (D) [] -> (Fun D D)
declare o: (D, C, B) [(Fun C B) (Fun D C)] -> (Fun D B)

set-precedence o (plus 10 (get-precedence at))

define [f g h x x' y] := [?f ?g ?h ?x ?x' ?y]

define identity-definition := (forall x . identity at x = x)

define compose-definition := (forall f g x . (f o g) at x = f at (g at x))

define function-equality :=
  (forall f g . f = g <==> forall x . f at x = g at x)

define theory :=
  (make-theory [] [identity-definition compose-definition function-equality])

define associative := (forall f g h . (f o g) o h = f o (g o h))
define right-identity :=  (forall f . f o identity = f)
define left-identity :=  (forall f . identity o f = f)
define Monoid-theorems := [associative right-identity left-identity]

define proofs :=
   method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [at identity o] := (adapt [at identity o]);
         [cd id] := [compose-definition identity-definition]}
      match theorem {
       (val-of associative) =>
         pick-any f g h
           let {all-x := pick-any x
                          (!chain
                             [((f o g) o h at x)
                         --> ((f o g) at h at x)     [cd]
                         --> (f at g at h at x)      [cd]
                         <-- (f at (g o h) at x)     [cd]
                         <-- ((f o (g o h)) at x)    [cd]])}
             (!chain-> [all-x
                    ==> ((f o g) o h = f o (g o h))   [function-equality]])
    | (val-of right-identity) =>
      pick-any f
        let {all-x := pick-any x
                        (!chain
                         [((f o identity) at x)
                          --> (f at (identity at x))  [cd]
                          --> (f at x)                [id]])}
        (!chain->
         [all-x ==> (f o identity = f) [function-equality]])
    | (val-of left-identity) =>
       pick-any f
         let {all-x := pick-any x
                         (!chain
                          [((identity o f) at x)
                           --> (identity at (f at x)) [cd]
                           --> (f at x)               [id]])}
         (!chain->
          [all-x ==> (identity o f = f) [function-equality]])
    }

(add-theorems theory |{Monoid-theorems := proofs}|)
```

```
#.................................................................

declare surjective, injective, bijective: (D, C) [(Fun D C)] -> Boolean

  define surjective-definition :=
    (forall f . surjective f <==> forall y . exists x . f at x = y)

  define injective-definition :=
    (forall f . injective f <==> forall x y . f at x = f at y ==> x = y)

  define bijective-definition :=
    (forall f . bijective f <==> surjective f & injective f)

  (add-axioms theory [surjective-definition injective-definition
                          bijective-definition])

define identity-surjective := (surjective identity)

define identity-injective := (injective identity)

define identity-bijective := (bijective identity)

define compose-surjective-preserving :=
  (forall f g . surjective f & surjective g ==> surjective f o g)

define compose-injective-preserving :=
  (forall f g . injective f & injective g ==> injective f o g)

define compose-bijective-preserving :=
  (forall f g . bijective f & bijective g ==> bijective f o g)

define Inverse-theorems :=
 [identity-surjective identity-injective identity-bijective
  compose-surjective-preserving compose-injective-preserving
  compose-bijective-preserving]

# Proofs of first and fourth:

define proofs-1 :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [at identity o] := (adapt [at identity o]);
         [cd id] := [compose-definition identity-definition]}
    match theorem {
      (val-of identity-surjective) =>
        let {SDI := (!instance surjective-definition [identity]);
             all-y :=
              pick-any y
                 (!chain->
                 [(identity at y) --> y          [id]
              ==> (exists x . identity at x = y)  [existence]])}
        (!chain-> [all-y ==>
                   (surjective identity)        [SDI]])
    | (val-of compose-surjective-preserving) =>
      pick-any f g
        assume (surjective f & surjective g)
          let {f-case :=
                (!chain->
                 [(surjective f)
              ==> (forall y .
                     exists x . f at x = y)      [surjective-definition]]);
               g-case :=
                (!chain->
                 [(surjective g)
                ==> (forall y .
                     exists x . g at x = y)      [surjective-definition]]);
               all-y :=
                 pick-any y
                   let {f-case-y :=
                          (!chain->
```

```
                           [true
                        ==> (exists y' .
                                f at y' = y)      [f-case]])}
                   pick-witness y' for f-case-y
                      let {g-case-y' :=
                            (!chain->
                             [true
                          ==> (exists x .
                                 g at x = y')     [g-case]])}
                        pick-witness x for g-case-y'
                        (!chain->
                         [(f o g at x)
                      --> (f at g at x)           [cd]
                      --> (f at y')               [(g at x = y')]
                      --> y                       [(f at y' = y)]
                      ==> (exists x .
                             f o g at x = y)      [existence]])}

          (!chain-> [all-y
                 ==> (surjective f o g)           [surjective-definition]])

    }

(add-theorems theory |{[identity-surjective
                        compose-surjective-preserving] := proofs-1}|)

define proofs :=
  method (theorem adapt)
   let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
        [at identity o] := (adapt [at identity o]);
        [cd id] := [compose-definition identity-definition]}
    match theorem {
      (val-of identity-injective) =>
         let {IDI := (!instance injective-definition [identity]);
              all-xx' :=
               pick-any x x'
                 assume A := ((identity at x) = (identity at x'))
                   (!chain
                    [x <-- (identity at x)           [id]
                       --> (identity at x')          [A]
                       --> x'                        [id]])}
          (!chain-> [all-xx' ==> (injective identity) [IDI]])
    | (val-of identity-bijective) =>
        let {BDI := (!instance bijective-definition [identity]);
             s-and-i := (!both (!prove identity-surjective)
                               (!prove identity-injective))}
         (!chain->
            [s-and-i ==> (bijective identity)          [BDI]])
    }

 (add-theorems theory |{[identity-injective identity-bijective] := proofs}|)

define proof :=
  method (theorem adapt)
   let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
        [at identity o] := (adapt [at identity o]);
        [cd id] := [compose-definition identity-definition]}
    match theorem {
      (val-of compose-injective-preserving) =>
       let {indef := injective-definition}
       pick-any f g
         assume (injective f & injective g)
           let {f-case := (!chain->
                           [(injective f)
                            ==> (forall x x' . f at x = f at x'
                                                 ==> x = x') [indef]]);
                g-case := (!chain->
                           [(injective g)
                            ==> (forall x x' . g at x = g at x'
                                                 ==> x = x') [indef]]);
```

```
                all-xx' :=
                 pick-any x x'
                   assume A := ((f o g) at x = (f o g) at x')
                     let {B := conclude (f at (g at x) =
                                          f at (g at x'))
                                 (!chain
                                  [(f at (g at x))
                                   <-- ((f o g) at x)   [cd]
                                   --> ((f o g) at x')  [A]
                                   --> (f at (g at x')) [cd]])}
                      (!chain->
                       [B ==> (g at x = g at x')          [f-case]
                          ==> (x = x')                    [g-case]])}

           (!chain-> [all-xx' ==> (injective f o g)     [indef]])
    }

(add-theorems theory |{compose-injective-preserving := proof}|)

define proof :=
  method (theorem adapt)
   let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
        [at identity o] := (adapt [at identity o]);
        [cd id] := [compose-definition identity-definition]}
    match theorem {
      (val-of compose-bijective-preserving) =>
      pick-any f:(Fun 'S 'T) g:(Fun 'U 'S)
        assume bfg := (bijective f & bijective g)
          let {f-s&i := (!chain-> [(bijective f) ==>
                                   (surjective f & injective f)
                                        [bijective-definition]]);
               g-s&i := (!chain-> [(bijective g) ==>
                                   (surjective g & injective g)
                                        [bijective-definition]]);
               f&g-s := (!both (!left-and f-s&i) (!left-and g-s&i));
               f&g-i := (!both (!right-and f-s&i) (!right-and g-s&i));
               csp := (!prove compose-surjective-preserving);
               cip := (!prove compose-injective-preserving);
               cs&i :=
               (!both
                (!chain-> [f&g-s ==> (surjective f o g) [csp]])
                (!chain-> [f&g-i ==> (injective f o g)  [cip]]))}
          (!chain-> [cs&i ==> (bijective f o g)
                                     [bijective-definition]])
   }
  (add-theorems theory |{compose-bijective-preserving := proof}|)
} # close module Function
```