lib/algebra/permutation.ath

```
# Permutation theory, basis for showing that permutations under composition
# form a group (for which see lib/algebra/permutation_unittest.ath).

load "algebra/function"

module Permutation {
  open Function
  domain (Perm D)

  declare perm->fun: (D) [(Perm D)] -> (Fun D D)
  declare fun->perm: (D) [(Fun D D)] -> (Perm D)

  set-precedence (perm->fun fun->perm) 350

  define [p q r f x y] := [?p:(Perm 'D1) ?q:(Perm 'D2) ?r:(Perm 'D3)
                           ?f:(Fun 'D4 'D5) ?x ?y]

  define is-bijective := (forall p . bijective perm->fun p)

  define fun->fun := (forall p . fun->perm perm->fun p = p)

  define perm->perm :=
    (forall f . bijective f ==> perm->fun fun->perm f = f)

  declare o: (D) [(Perm D) (Perm D)] -> (Perm D)
  declare identity: (D) [] -> (Perm D)

  define o' := Function.o
  set-precedence o' (plus 10 (get-precedence perm->fun))

  define identity' := Function.identity

  define compose-definition :=
    (forall p q . p o q = fun->perm (perm->fun p o' perm->fun q))

  define identity-definition := (identity = fun->perm identity')

  define theory :=
    (make-theory ['Function]
                 [is-bijective fun->fun perm->perm
                  compose-definition identity-definition])

  define associative := (forall p q r . (p o q) o r = p o (q o r))
  define right-identity := (forall p . p o identity = p)
  define left-identity := (forall p . identity o p = p)

  define Monoid-theorems := [associative right-identity left-identity]

define [f->p p->f] := [fun->perm perm->fun]

define proofs :=
  method (theorem adapt)
    let {[_ prove chain chain-> _] := (proof-tools adapt theory);
         [identity' o' at identity o fun->perm perm->fun] :=
           (adapt [identity' o' at identity o fun->perm perm->fun]);
         [id cd] := [identity-definition compose-definition]}
    match theorem {
      (val-of associative) =>
         let {CA := (!prove Function.associative);
              CBP := (!prove compose-bijective-preserving)}
         pick-any p:(Perm 'S) q:(Perm 'S) r:(Perm 'S)
           let {_ := (!chain-> [true
                           ==> (bijective perm->fun q) [is-bijective]]);
                _ := (!chain->
                        [true
                     ==> (bijective perm->fun p)           [is-bijective]
                     ==> (bijective perm->fun p &
```

```
                        bijective perm->fun q)          [augment]
                   ==> (bijective
                         (perm->fun p o' perm->fun q))  [CBP]]);
                _ := (!chain->
                      [true
                   ==> (bijective perm->fun r)          [is-bijective]
                   ==> (bijective perm->fun q &
                        bijective perm->fun r)          [augment]
                   ==> (bijective
                         (perm->fun q o' perm->fun r))  [CBP]])}
           (!combine-equations
            (!chain
             [((p o q) o r)
          --> ((fun->perm (perm->fun p o' perm->fun q)) o r)    [cd]
          --> (fun->perm
                ((perm->fun fun->perm
                  (perm->fun p o' perm->fun q)) o' perm->fun r)) [cd]
          --> (fun->perm
                 ((perm->fun p o' perm->fun q) o' perm->fun r))  [perm->perm]
          --> (fun->perm
                (perm->fun p o' (perm->fun q o' perm->fun r)))   [CA]])
            (!chain
             [(p o (q o r))
          --> (p o (fun->perm (perm->fun q o' perm->fun r)))    [cd]
          --> (fun->perm
                (perm->fun p o'
                 (perm->fun
                   fun->perm
                    (perm->fun q o' perm->fun r))))             [cd]
          --> (fun->perm
                 (perm->fun p o' (perm->fun q o' perm->fun r)))  [perm->perm]
                 ]))
    | (val-of right-identity) =>
        let {RI := (!prove Function.right-identity);
             IB := (!prove identity-bijective)}
        pick-any p:(Perm 'S)
          (!chain
           [(p o identity)
            --> (p o (fun->perm identity'))                [IB id]
            --> (fun->perm (perm->fun p o'
                            perm->fun fun->perm identity')) [cd]
            --> (fun->perm (perm->fun p o' identity'))      [perm->perm]
            --> (fun->perm (perm->fun p))                   [RI]
            --> p                                           [fun->fun]])
    | (val-of left-identity) =>
      let {LI := (!prove Function.left-identity);
           IB := (!prove identity-bijective)}
       pick-any p:(Perm 'S)
          (!chain
           [(identity o p)
            --> ((fun->perm identity') o p)                [IB id]
            --> (fun->perm ((perm->fun fun->perm identity')
                            o' (perm->fun p)))              [cd]
            --> (fun->perm (identity' o' (perm->fun p)))    [perm->perm]
            --> (fun->perm (perm->fun p))                   [LI]
            --> p                                           [fun->fun]])
    }

  (add-theorems theory |{Monoid-theorems := proofs}|)
} # close module Permutation

extend-module Permutation {
  declare at: (D) [(Perm D) D] -> D

  define at' := Function.at

  define at-definition := (forall p x . p at x = (perm->fun p) at' x)

  declare inverse: (D) [(Perm D)] -> (Perm D)

```

```
define inverse-definition :=
  (forall p x y . p at x = y ==> (inverse p) at y = x)

declare div: (D) [(Perm D) (Perm D)] -> (Perm D)

define div-definition := (forall p q . p div q = p o inverse q)

(add-axioms theory [at-definition inverse-definition div-definition])

define consistent-inverse :=
  (forall p x x' y . p at x = y & p at x' = y ==> x = x')

define right-inverse-lemma :=
  (forall p . (perm->fun p) o' (perm->fun inverse p) = identity')

define right-inverse := (forall p . p o inverse p = identity)

define Inverse-theorems :=
  [consistent-inverse right-inverse-lemma right-inverse]

define [bij-def inj-def] := [bijective-definition injective-definition]
define at-def := at-definition

define proofs :=
 method (theorem adapt)
   let {[_ prove chain chain-> _] := (proof-tools adapt theory);
        [at' identity' o' at identity o fun->perm perm->fun inverse] :=
          (adapt [at' identity' o' at identity o fun->perm perm->fun
                  inverse]);
       [cd bid] := [compose-definition bijective-definition]}
     match theorem {
       (val-of consistent-inverse) =>
         pick-any p x x' y
           let {inj := (!chain->
                         [true
                     ==> (bijective perm->fun p)        [is-bijective]
                     ==> (injective perm->fun p)        [bij-def]])}
             assume (p at x = y & p at x' = y)
               let {p1 := (!chain->
                             [(p at x) = y              [(p at x = y)]
                                       = (p at x')      [(p at x' = y)]
                         ==> ((perm->fun p) at' x =
                              (perm->fun p) at' x')     [at-def]]);
                    p2 := (!chain->
                             [inj
                         ==> (forall x x' .
                               (perm->fun p) at' x =
                               (perm->fun p) at' x'
                                 ==> x = x')            [inj-def]])}
                 (!chain-> [p1 ==> (x = x')             [p2]])
     | (val-of right-inverse-lemma) =>
       pick-any p
         let {surj :=
                (!chain->
                 [true
              ==> (bijective (perm->fun p))    [is-bijective]
              ==> (surjective (perm->fun p))   [bid]
              ==> (forall y .
                    exists x .
                      (perm->fun p) at' x = y) [surjective-definition]]);
              f := ((perm->fun p) o' (perm->fun inverse p));
              all-y :=
                conclude (forall y . f at' y = identity' at' y)
                  pick-any y
                    pick-witness x for (!instance surj y) witnessed
                         (!chain
                        [(f at' y)
                    <-- (f at' ((perm->fun p) at' x))    [witnessed]
                    --> ((perm->fun p) at'
                          ((perm->fun inverse p) at'
```

```
                                  ((perm->fun p) at' x)))
                                           [Function.compose-definition]
                     <-- (p at ((inverse p) at (p at x))) [at-definition]
                     --> (p at x)                        [inverse-definition]
                     --> ((perm->fun p) at' x)           [at-definition]
                     --> y                               [witnessed]
                     <-- (identity' at' y) [Function.identity-definition]])}
          (!chain-> [all-y ==> (f = identity')        [function-equality]])
      | (val-of right-inverse) =>
        let {RIL := (!prove right-inverse-lemma)}
          pick-any p
              (!chain [(p o inverse p)
                 --> (fun->perm
                       ((perm->fun p) o'
                        (perm->fun inverse p)))      [cd]
                 --> (fun->perm identity')           [RIL]
                 <-- identity                        [identity-definition]])
      }

  (add-theorems theory |{Inverse-theorems := proofs}|)
} # close module Permutation
```