## Abstract algebraic theories: Semigroup, Identity, Monoid, Group

module Semigroup {
    declare +: (S) [S S] -> S [200]
    define associative := (forall x y z . (x + y) + z = x + (y + z))
    define theory := (make-theory [ ] [associative])
}

module Identity {
    open Semigroup
    declare <0>: (S) [ ] -> S
    define left-identity := (forall x . <0> + x = x)
    define right-identity := (forall x . x + <0> = x)
    define theory := (make-theory [ ] [left-identity right-identity])
}

module Monoid {
    open Identity
    define theory := (make-theory [Semigroup Identity] [ ])
}

module Group {
    open Monoid
    declare U-: (S) [S] -> S
    declare -: (S) [S S] -> S
    define right-inverse := (forall x . x + U- x = <0>)
    define minus-definition := (forall x y . x - y = x + U- y)
    define theory := (make-theory [Monoid] [right-inverse minus-definition])
}

extend-module Group {
    define left-inverse := (forall x . (U- x) + x = <0>)
    define double-negation := (forall x . U- U- x = x)
    define unique-negation := (forall x y . x + y = <0> ==> U- x = y)
    define neg-plus := (forall x y . U- (x + y) = (U- y) + (U- x))
    define left-inverse-proof :=
        method (theorem adapt)
        let {{ _ _ chain _ _ } := (proof-tools adapt theory); [+ U- <0>]} := (adapt [+ U- <0>])
        conclude (adapt theorem)
        pick-any x
        (chain
        [(U- x) + x]
        <- (((U- x) + x) + <0>)
        [right-identity]
        ---> ((U- x) + (x + <0>))
        [associative]
        <- (U- x) + (x + ((U- x) + U- U- x))
        [right-inverse]
        ---> ((U- x) + U- U- x)
        [associative]
        <- (((U- x) + <0>) + U- U- x)
        [associative]
        ---> ((U- x) + U- U- x)
        [right-identity]
        ---> <0>
        [right-inverse])
    (add-theorems 'Group |{ left-inverse := left-inverse-proof |})
}

extend-module Group {
    define proofs :=
        method (theorem adapt)
let { [get prove chain chain-> chain<->] := (proof-tools adapt theory); [+ U- <0>] := (adapt [+ U- <0>]) }

match theorem {
(val-of double-negation) =>
  conclude [adapt theorem]
  pick-any x:(sort-of <0>)
  { [left-identity] [right-inverse] [associative] [right-inverse] [right-identity] } <- (x + (U- x))<-> (x + (U- x))
  pick-any x:(sort-of <0>)
  !chain [{LI := (!prove left-inverse)}]
| (val-of unique-negation) =>
  conclude [adapt theorem]
  pick-any x y:(sort-of <0>)
  let { LI := (!prove unique-negation) }
  assume A := (x + y = <0>)
  { [left-identity] [A] [associative] [left-identity] [left-identity] } <- ((U- x) + x) <- ((U- x) + x) + y
  --> (<0> + y)
  --> (x + (<0> + (U- x)))
  --> (<0>)
  --> <0>
  --> <0>
  {[ A == (U- x) + y = (U- x) + (U- x) ] [UN]}

(add-theorems theory |{left-inverse := left-inverse-proof}|)

module Abelian-Monoid {
  open Monoid
  define commutative := (forall x y . x + y = y + x)
  define theory := (make-theory [Monoid] [commutative])
}

module Abelian-Group {
  open Group
  define commutative := (forall x y . x + y = y + x)
  define theory := (make-theory [Group] [commutative])
}

# Commutativity allows a shorter proof for Left-Inverse and
# a more natural statement of Neg-Plus:

extend-module Abelian-Group {
  define left-inverse-proof :=
    method (theorem adapt)
    let { [get prove chain chain-> chain<-] := (proof-tools adapt theory); [+ U- <0>] := (adapt [+ U- <0>]) }
    conclude [adapt theorem]
    pick-any x
    { [associative] [right-inverse] [right-inverse] [right-identity] } <- ((U- x) + x)
    pick-any x y
    let { UN := (!prove unique-negation); }
    assume A := (x + y = <0>)
    { [left-identity] [A] [left-identity] [left-identity] } <- ((x + y) + (U- x)) <- ((U- x) + (U- x))
    --> (x + (<0> + (U- x)))
    --> (x + (U- x))
    --> <0>
    {[ A == (U- x) + y = (U- x) + (U- x) ] [UN]}

    (add-theorems theory |{left-inverse := left-inverse-proof}|)

  define neg-plus := (forall x y . U- (x + y) = (U- x) + (U- y))
define neg-plus-proof :=
  method (theorem adapt)
    let ((get prove chain chain-> chain<-) := (proof-tools adapt theory);
        [+ U- <0>] := (adapt [+ U- <0>]))
    conclude (adapt theorem)
  pick-any x y
    let {Group-version := (!prove-property Group.neg-plus
        adapt Group.theory)}
    (!chain {((U- (x + y))
        --> ((U- y) + (U- x)) [Group-version]
        --> ((U- x) + (U- y)) [commutative]})
    (add-theorems theory (add-theorems theory (neg-plus := neg-plus-proof)))
} # close module Abelian-Group

# Multiplicative-Semigroup, Monoid, and Group theories
module MSG { # Multiplicative-Semigroup
    declare *: (S) [S S] -> S [300]
    define theory := (adapt-theory 'Semigroup [Semigroup.+ := *])
}

module MM { # Multiplicative-Monoid
    declare <1>: (S) [] -> S
    define theory := (adapt-theory 'Monoid [Semigroup.+ := MSG.*, Monoid.<0> := <1>])
}

module MAM { # Multiplicative-Abelian-Monoid
    open MM
    define theory := (adapt-theory 'Abelian-Monoid [Monoid.<0> := <1>])
}

module MG { # Multiplicative-Group
    declare inv: (T) [T] -> T
    declare /: (T) [T T] -> T
    define theory := (adapt-theory 'Group [Monoid.<0> := MM.<1>,
                                           Group.U- := inv, Group.- := /])
}