Contents

Chapter 1

Library functional-algebra.base

This module defines functions and notations shared by all of the modules in this package.

The following notations are introduced here to simplify sequences of algebraic rewrites which would otherwise be expressed as long sequences of eq_ind*.

`Notation` "A || B @ X 'by' E"
  := (eq_ind_r (`fun` $X \Rightarrow B$) A E) (`at level` 40, `left associativity`).

`Notation` "A || B @ X 'by' <- H"
  := (eq_ind_r (`fun` $X \Rightarrow B$) A (eq_sym H)) (`at level` 40, `left associativity`).

The following notation can be used to define equality assertions. These are like unittests in that they check that a given expression reduces to a given value. `Notation` "A =:= B"
  := (eq_refl A : $A = B$) (`at level` 90).

Chapter 2

Library functional-algebra.abelian_group

This module defines the Abelian Group record type which can be used to represent abelian groups and provides a collection of axioms and theorems describing them.

`Require Import` *Description*.
`Require Import` *base*.
`Require Import` *function*.
`Require Import` *monoid*.
`Require Import` *group*.

`Module` *Abelian_Group*.

Accepts one argument: f, a binary function; and asserts that f is commutative. `Definition` *is_comm* (T : `Type`) (f : $T \rightarrow T \rightarrow T$)
: `Prop`
:= $\forall\ x\ y : T, f\ x\ y = f\ y\ x$.

Represents algebraic abelian groups. `Structure` *Abelian_Group* : `Type` := *abelian_group*
{

Represents the set of group elements. E : `Set`;

Represents the identity element. *E_0* : E;

Represents the group operation. *op* : $E \rightarrow E \rightarrow E$;

Asserts that the group operator is associative. *op_is_assoc* : *Monoid.is_assoc* E *op*;

Asserts that the group operator is commutative. *op_is_comm* : *is_comm* E *op*;

Asserts that E_0 is the left identity element. *op_id_l* : *Monoid.is_id_l* E *op* *E_0*;

Asserts that every element has a left inverse.

Strictly speaking, this axiom should be:

forall x : E, exists y : E, Monoid.is_inv_l E op E_0 op_id x y

which asserts and verifies that op y x equals the identity element. Technically, we haven't shown that E_0 is the identity element yet, so we're being a bit presumptuous defining inverses in this way. While we could prove op_id in this structure definition, we prefer not to to improve readability and instead use the form given below, which Monoid.is_inv_l reduces to anyway. *op_inv_l_ex* : $\forall$ *x* : *E*, $\exists$ *y* : *E*, *op y x* = *E_0*

}.

Enable implicit arguments for group properties.

Arguments E_0 {*a*}.

Arguments op {*a*} *x y*.

Arguments op_is_assoc {*a*} *x y z*.

Arguments op_is_comm {*a*} *x y*.

Arguments op_id_l {*a*} *x*.

Arguments op_inv_l_ex {*a*} *x*.

Define notations for group properties.

`Notation` "0" := *E_0* : *abelian_group_scope*.

`Notation` "x + y" := (*op x y*) (`at level` 50, `left associativity`) : *abelian_group_scope*.

`Notation` "{+}" := *op* : *abelian_group_scope*.

`Open Scope` *abelian_group_scope*.

`Section` *Theorems*.

Represents an arbitrary abelian group.

Note: we use Variable rather than Parameter to ensure that the following theorems are generalized w.r.t ag. `Variable` *g* : *Abelian_Group*.

Represents the set of group elements.

Note: We use Let to define E as a local abbreviation. `Let` *E* := *E g*.

Accepts one group element, x, and asserts that x is the left identity element. `Definition` *op_is_id_l* := *Monoid.is_id_l E* {+}.

Accepts one group element, x, and asserts that x is the right identity element. `Definition` *op_is_id_r* := *Monoid.is_id_r E* {+}.

Accepts one group element, x, and asserts that x is the identity element. `Definition` *op_is_id* := *Monoid.is_id E* {+}.

Proves that every left identity must also be a right identity. `Definition` *op_is_id_lr*

: $\forall$ *x* : *E*, *op_is_id_l x* $\rightarrow$ *op_is_id_r x*

:= `fun` *x H y*

$\Rightarrow$ *H y*

$|| \; a = y$ @a `by` $op\_is\_comm\ y\ x$.

Proves that every left identity is an identity. `Definition` $op\_is\_id\_lid$
$: \forall\ x : E,\ op\_is\_id\_l\ x \rightarrow op\_is\_id\ x$
:= `fun` $x\ H$
$\Rightarrow conj\ H\ (op\_is\_id\_lr\ x\ H)$.

Proves that 0 is the right identity element. `Definition` op_id_r
$: op\_is\_id\_r\ 0$
$:= op\_is\_id\_lr\ 0\ op\_id\_l$.

Proves that 0 is the identity element. `Definition` op_id
$: op\_is\_id\ 0$
$:= conj\ op\_id\_l\ op\_id\_r$.

Accepts two group elements, x and y, and asserts that y is x's left inverse. `Definition` $op\_is\_inv\_l := Monoid.is\_inv\_l\ E\ \{+\}\ 0\ op\_id$.

Accepts two group elements, x and y, and asserts that y is x's right inverse. `Definition` $op\_is\_inv\_r := Monoid.is\_inv\_r\ E\ \{+\}\ 0\ op\_id$.

Accepts two group elements, x and y, and asserts that y is x's inverse. `Definition` $op\_is\_inv := Monoid.is\_inv\ E\ \{+\}\ 0\ op\_id$.

Proves that every element has a right inverse. `Definition` $op\_inv\_r\_ex$
$: \forall\ x : E, \exists\ y : E,\ op\_is\_inv\_r\ x\ y$
:= `fun` x
$\Rightarrow ex\_ind$
(`fun` $(y : E)\ (H : op\_is\_inv\_l\ x\ y)$
$\Rightarrow ex\_intro$
$(op\_is\_inv\_r\ x)$
y
$(H \;||\; a = 0$ @a `by` $op\_is\_comm\ x\ y))$
$(op\_inv\_l\_ex\ x)$.

Represents the group structure formed by op over E. `Definition` $op\_group := Group.group\ E\ 0\ \{+\}\ op\_is\_assoc\ op\_id\_l\ op\_id\_r\ op\_inv\_l\_ex\ op\_inv\_r\_ex$.

Represents the monoid formed by op over E. `Definition` $op\_monoid := Group.op\_monoid\ op\_group$.

Proves that the left identity element is unique. `Definition` $op\_id\_l\_uniq$
$: \forall\ x : E,\ Monoid.is\_id\_l\ E\ \{+\}\ x \rightarrow x = 0$
$:= Group.op\_id\_l\_uniq\ op\_group$.

Proves that the right identity element is unique. `Definition` $op\_id\_r\_uniq$
$: \forall\ x : E,\ Monoid.is\_id\_r\ E\ \{+\}\ x \rightarrow x = 0$
$:= Group.op\_id\_r\_uniq\ op\_group$.

Proves that the identity element is unique. `Definition` op_id_uniq
$: \forall\ x : E,\ Monoid.is\_id\ E\ \{+\}\ x \rightarrow x = 0$

$:=$ *Group.op_id_uniq op_group.*

Proves that for every group element, x, its left and right inverses are equal. `Definition` *op_inv_l_r_eq*
$: \forall\ x\ y : E,\ op\_is\_inv\_l\ x\ y \rightarrow \forall\ z : E,\ op\_is\_inv\_r\ x\ z \rightarrow y = z$
$:=$ *Group.op_inv_l_r_eq op_group.*

Proves that the inverse relation is symmetrical. `Definition` *op_inv_sym*
$: \forall\ x\ y : E,\ op\_is\_inv\ x\ y \leftrightarrow op\_is\_inv\ y\ x$
$:=$ *Group.op_inv_sym op_group.*

Proves that every group element has an inverse. `Definition` *op_inv_ex*
$: \forall\ x : E, \exists\ y : E,\ op\_is\_inv\ x\ y$
$:=$ *Group.op_inv_ex op_group.*

Proves the left introduction rule. `Definition` *op_intro_l*
$: \forall\ x\ y\ z : E,\ x = y \rightarrow z + x = z + y$
$:=$ *Group.op_intro_l op_group.*

Proves the right introduction rule. `Definition` *op_intro_r*
$: \forall\ x\ y\ z : E,\ x = y \rightarrow x + z = y + z$
$:=$ *Group.op_intro_r op_group.*

Proves the left cancellation rule. `Definition` *op_cancel_l*
$: \forall\ x\ y\ z : E,\ z + x = z + y \rightarrow x = y$
$:=$ *Group.op_cancel_l op_group.*

Proves the right cancellation rule. `Definition` *op_cancel_r*
$: \forall\ x\ y\ z : E,\ x + z = y + z \rightarrow x = y$
$:=$ *Group.op_cancel_r op_group.*

Proves that an element's left inverse is unique. `Definition` *op_inv_l_uniq*
$: \forall\ x\ y\ z : E,\ op\_is\_inv\_l\ x\ y \rightarrow op\_is\_inv\_l\ x\ z \rightarrow z = y$
$:=$ *Group.op_inv_l_uniq op_group.*

Proves that an element's right inverse is unique. `Definition` *op_inv_r_uniq*
$: \forall\ x\ y\ z : E,\ op\_is\_inv\_r\ x\ y \rightarrow op\_is\_inv\_r\ x\ z \rightarrow z = y$
$:=$ *Group.op_inv_r_uniq op_group.*

Proves that an element's inverse is unique. `Definition` *op_inv_uniq*
$: \forall\ x\ y\ z : E,\ op\_is\_inv\ x\ y \rightarrow op\_is\_inv\ x\ z \rightarrow z = y$
$:=$ *Group.op_inv_uniq op_group.*

Proves explicitly that every element has a unique inverse. `Definition` *op_inv_uniq_ex*
$: \forall\ x : E, \exists!\ y : E,\ op\_is\_inv\ x\ y$
$:=$ *Group.op_inv_uniq_ex op_group.*

Represents strongly-specified negation. `Definition` *op_neg_strong*
$: \forall\ x : E, \{\ y \mid op\_is\_inv\ x\ y\ \}$
$:=$ *Group.op_neg_strong op_group.*

Represents negation. `Definition` *op_neg*

$: E \to E$
$:= Group.op\_neg\ op\_group.$

Close Scope *nat_scope.*

Notation "{-}" := (op_neg) : *abelian_group_scope.*

Notation "- x" := $(op\_neg\ x)$: *abelian_group_scope.*

Asserts that the negation returns the inverse of its argument. Definition *op_neg_def*
$: \forall\ x : E,\ op\_is\_inv\ x$ (- x)
$:= Group.op\_neg\_def\ op\_group.$

Proves that negation is one-to-one Definition *op_neg_inj*
$: is\_injective\ E\ E\ op\_neg$
$:= Group.op\_neg\_inj\ op\_group.$

Proves the cancellation property for negation. Definition *op_cancel_neg*
$: \forall\ x : E,\ op\_neg$ (- x) $= x$
$:= Group.op\_cancel\_neg\ op\_group.$

Proves that negation is onto Definition *op_neg_onto*
$: is\_onto\ E\ E\ op\_neg$
$:= Group.op\_neg\_onto\ op\_group.$

Proves that negation is surjective Definition *op_neg_bijective*
$: is\_bijective\ E\ E\ op\_neg$
$:= Group.op\_neg\_bijective\ op\_group.$

Proves that neg x = y -> neg y = x Definition *op_neg_rev*
$: \forall\ x\ y : E$, - $x = y \to$ - $y = x$
$:= Group.op\_neg\_rev\ op\_group.$

End *Theorems.*

End *Abelian_Group.*

Chapter 3

Library functional-algebra.commutative_ring

This module defines the commutative_ring record type which represents algebraic commutative rings and provides a collection of axioms and theorems describing them.

```
Require Import Description.
Require Import FunctionalExtensionality.
Require Import base.
Require Import function.
Require Import monoid.
Require Import group.
Require Import abelian_group.
Require Import ring.
```

```
Module Commutative_Ring.
```

Represents algebraic commutative rings. `Structure` *Commutative_Ring* : `Type` := *commutative_ring* {

Represents the set of ring elements. *E* : `Set`;

Represents 0 - the additive identity. *E_0* : *E*;

Represents 1 - the multiplicative identity. *E_1* : *E*;

Represents addition. *sum* : $E \to E \to E$;

Represents multiplication. *prod* : $E \to E \to E$;

Asserts that 0 /= 1. *distinct_0_1*: $E\_0 \neq E\_1$;

Asserts that addition is associative. *sum_is_assoc* : *Monoid.is_assoc E sum*;

Asserts that addition is commutative. $sum\_is\_comm : Abelian\_Group.is\_comm\ E\ sum$;

Asserts that 0 is the left identity element. $sum\_id\_l : Monoid.is\_id\_l\ E\ sum\ E\_0$;

Asserts that every element has an additive inverse. $sum\_inv\_l\_ex : \forall\ x : E, \exists\ y : E, sum\ y\ x = E\_0$;

Asserts that multiplication is associative. $prod\_is\_assoc : Monoid.is\_assoc\ E\ prod$;

Asserts that multiplication is commutative. $prod\_is\_comm : Abelian\_Group.is\_comm\ E\ prod$;

Asserts that 1 is the left identity element. $prod\_id\_l : Monoid.is\_id\_l\ E\ prod\ E\_1$;

Asserts that multiplication is left distributive over addition. $prod\_sum\_distrib\_l : Ring.is\_distrib\_l\ E\ prod\ sum$
}.

Enable implicit arguments for commutative ring properties.

Arguments E_0 {c}.

Arguments E_1 {c}.

Arguments sum {c} x y.

Arguments prod {c} x y.

Arguments distinct_0_1 {c} _.

Arguments sum_is_assoc {c} x y z.

Arguments sum_is_comm {c} x y.

Arguments sum_id_l {c} x.

Arguments sum_inv_l_ex {c} x.

Arguments prod_is_assoc {c} x y z.

Arguments prod_id_l {c} x.

Arguments prod_sum_distrib_l {c} x y z.

Arguments prod_is_comm {c} x y.

Define notations for ring properties.

`Notation` "0" := *E_0* : *commutative_ring_scope.*

`Notation` "1" := *E_1* : *commutative_ring_scope.*

`Notation` "x + y" := (*sum x y*) (`at level` 50, `left associativity`) : *commutative_ring_scope.*

`Notation "{+}" := sum : commutative_ring_scope.`

`Notation "x # y" := (prod x y) (at level 50, left associativity) : commutative_ring_scope.`

`Notation "{#}" := prod : commutative_ring_scope.`

`Open Scope commutative_ring_scope.`

`Section Theorems.`

Represents an arbitrary commutative ring.

Note: we use Variable rather than Parameter to ensure that the following theorems are generalized w.r.t r. `Variable r : Commutative_Ring.`

Represents the set of group elements.

Note: We use Let to define E as a local abbreviation. `Let E := E r.`

Accepts one ring element, x, and asserts that x is the left identity element. `Definition sum_is_id_l := Monoid.is_id_l E {+}.`

Accepts one ring element, x, and asserts that x is the right identity element. `Definition sum_is_id_r := Monoid.is_id_r E {+}.`

Accepts one ring element, x, and asserts that x is the identity element. `Definition sum_is_id := Monoid.is_id E {+}.`

Accepts one ring element, x, and asserts that x is the left identity element. `Definition prod_is_id_l := Monoid.is_id_l E {#}.`

Accepts one ring element, x, and asserts that x is the right identity element. `Definition prod_is_id_r := Monoid.is_id_r E {#}.`

Accepts one ring element, x, and asserts that x is the identity element. `Definition prod_is_id := Monoid.is_id E {#}.`

Proves that 1 is the right identity element.

```
Definition prod_id_r
  : prod_is_id_r 1
  := fun x : E
       ⇒ eq_ind_r
            (fun a ⇒ a = x)
            (prod_id_l x)
            (prod_is_comm x 1).
```

Proves that multiplication is right distributive over addition.

```
Definition prod_sum_distrib_r
  : Ring.is_distrib_r E {#} {+}
  := fun x y z : E
       ⇒ prod_sum_distrib_l x y z
       || x # (y + z) = a + (x # z) @a by ← prod_is_comm x y
       || x # (y + z) = (y # x) + a @a by ← prod_is_comm x z
       || a = (y # x) + (z # x) @a by ← prod_is_comm x (y + z).
```

Represents the non-commutative ring formed by addition and multiplication over E. `Definition ring := Ring.ring E 0 1 {+} {#} distinct_0_1 sum_is_assoc sum_is_comm sum_id_l sum_inv_l_ex prod_is_assoc prod_id_l prod_id_r prod_sum_distrib_l prod_sum_distrib_r.`

Represents the abelian group formed by addition over E. `Definition` *sum_abelian_group* := *Ring.sum_abelian_group* `ring`.

Represents the group formed by addition over E. `Definition` *sum_group* := *Ring.sum_group* `ring`.

Represents the monoid formed by addition over E. `Definition` *sum_monoid* := *Ring.sum_monoid* `ring`.

Represents the monoid formed by multiplication over E. `Definition` *prod_monoid* := *Ring.prod_monoid* `ring`.

Proves that 1 <> 0. `Definition` *distinct_1_0*
: $E\_1\ (c := r) \neq E\_0\ (c := r)$
:= `fun` $H : E\_1 = E\_0$
$\Rightarrow$ *distinct_0_1* (*eq_sym* H).

A predicate that accepts one element, x, and asserts that x is nonzero. `Definition` *nonzero*
: $E \rightarrow$ `Prop`
:= *Ring.nonzero* `ring`.

Proves that 0 is the right identity element. `Definition` *sum_id_r*
: *sum_is_id_r* 0
:= *Ring.sum_id_r* `ring`.

Proves that 0 is the identity element. `Definition` *sum_id*
: *sum_is_id* 0
:= *Ring.sum_id* `ring`.

Accepts two elements, x and y, and asserts that y is x's left inverse. `Definition` *sum_is_inv_l* := *Monoid.is_inv_l* E {+} 0 *sum_id*.

Accepts two elements, x and y, and asserts that y is x's right inverse. `Definition` *sum_is_inv_r* := *Monoid.is_inv_r* E {+} 0 *sum_id*.

Accepts two elements, x and y, and asserts that y is x's inverse. `Definition` *sum_is_inv* := *Monoid.is_inv* E {+} 0 *sum_id*.

Asserts that every element has a right inverse. `Definition` *sum_inv_r_ex*
: $\forall\ x : E, \exists\ y : E$, *sum_is_inv_r* $x\ y$
:= *Ring.sum_inv_r_ex* `ring`.

Proves that the left identity element is unique. `Definition` *sum_id_l_uniq*
: $\forall\ x : E$, *Monoid.is_id_l* E {+} $x \rightarrow x = 0$
:= *Ring.sum_id_l_uniq* `ring`.

Proves that the right identity element is unique. `Definition` *sum_id_r_uniq*
: $\forall\ x : E$, *Monoid.is_id_r* E {+} $x \rightarrow x = 0$
:= *Ring.sum_id_r_uniq* `ring`.

Proves that the identity element is unique. `Definition` *sum_id_uniq*

: $\forall\ x\ :\ E,\ Monoid.is\_id\ E\ \{+\}\ x \rightarrow x = 0$
:= *Ring.sum_id_uniq* `ring`.

Proves that for every group element, x, its left and right inverses are equal. `Definition` *sum_inv_l_r_eq*
: $\forall\ x\ y\ :\ E,\ sum\_is\_inv\_l\ x\ y \rightarrow \forall\ z\ :\ E,\ sum\_is\_inv\_r\ x\ z \rightarrow y = z$
:= *Ring.sum_inv_l_r_eq* `ring`.

Proves that the inverse relation is symmetrical. `Definition` *sum_inv_sym*
: $\forall\ x\ y\ :\ E,\ sum\_is\_inv\ x\ y \leftrightarrow sum\_is\_inv\ y\ x$
:= *Ring.sum_inv_sym* `ring`.

Proves that an element's inverse is unique. `Definition` *sum_inv_uniq*
: $\forall\ x\ y\ z\ :\ E,\ sum\_is\_inv\ x\ y \rightarrow sum\_is\_inv\ x\ z \rightarrow z = y$
:= *Ring.sum_inv_uniq* `ring`.

Proves that every element has an inverse. `Definition` *sum_inv_ex*
: $\forall\ x\ :\ E,\ \exists\ y\ :\ E,\ sum\_is\_inv\ x\ y$
:= *Ring.sum_inv_ex* `ring`.

Proves explicitly that every element has a unique inverse. `Definition` *sum_inv_uniq_ex*
: $\forall\ x\ :\ E,\ \exists!\ y\ :\ E,\ sum\_is\_inv\ x\ y$
:= *Ring.sum_inv_uniq_ex* `ring`.

Proves the left introduction rule. `Definition` *sum_intro_l*
: $\forall\ x\ y\ z\ :\ E,\ x = y \rightarrow z + x = z + y$
:= *Ring.sum_intro_l* `ring`.

Proves the right introduction rule. `Definition` *sum_intro_r*
: $\forall\ x\ y\ z\ :\ E,\ x = y \rightarrow x + z = y + z$
:= *Ring.sum_intro_r* `ring`.

Proves the left cancellation rule. `Definition` *sum_cancel_l*
: $\forall\ x\ y\ z\ :\ E,\ z + x = z + y \rightarrow x = y$
:= *Ring.sum_cancel_l* `ring`.

Proves the right cancellation rule. `Definition` *sum_cancel_r*
: $\forall\ x\ y\ z\ :\ E,\ x + z = y + z \rightarrow x = y$
:= *Ring.sum_cancel_r* `ring`.

Proves that an element's left inverse is unique. `Definition` *sum_inv_l_uniq*
: $\forall\ x\ y\ z\ :\ E,\ sum\_is\_inv\_l\ x\ y \rightarrow sum\_is\_inv\_l\ x\ z \rightarrow z = y$
:= *Ring.sum_inv_l_uniq* `ring`.

Proves that an element's right inverse is unique. `Definition` *sum_inv_r_uniq*
: $\forall\ x\ y\ z\ :\ E,\ sum\_is\_inv\_r\ x\ y \rightarrow sum\_is\_inv\_r\ x\ z \rightarrow z = y$
:= *Ring.sum_inv_r_uniq* `ring`.

Proves that 0 is its own left additive inverse. `Definition` *sum_0_inv_l*
: $sum\_is\_inv\_l\ 0\ 0$
:= *Ring.sum_0_inv_l* `ring`.

Proves that 0 is its own right additive inverse. `Definition` *sum_0_inv_r*
: *sum_is_inv_r* 0 0
:= *Ring.sum_0_inv_r* `ring`.

Proves that 0 is it's own additive inverse. `Definition` *sum_0_inv*
: *sum_is_inv* 0 0
:= *Ring.sum_0_inv* `ring`.

Represents strongly-specified negation. `Definition` *sum_neg_strong*
: ∀ *x* : *E*, { *y* | *sum_is_inv* *x* *y* }
:= *Ring.sum_neg_strong* `ring`.

Represents negation. `Definition` *sum_neg*
: *E* → *E*
:= *Ring.sum_neg* `ring`.

`Notation` "{-}" := (*sum_neg*) : *commutative_ring_scope.*

`Notation` "- x" := (*sum_neg* *x*) : *commutative_ring_scope.*

Asserts that the negation returns the inverse of its argument. `Definition` *sum_neg_def*

: ∀ *x* : *E*, *sum_is_inv* *x* (- *x*)
:= *Ring.sum_neg_def* `ring`.

Proves that negation is one-to-one `Definition` *sum_neg_inj*
: *is_injective* *E* *E* *sum_neg*
:= *Ring.sum_neg_inj* `ring`.

Proves the cancellation property for negation. `Definition` *sum_cancel_neg*
: ∀ *x* : *E*, *sum_neg* (- *x*) = *x*
:= *Ring.sum_cancel_neg* `ring`.

Proves that negation is onto `Definition` *sum_neg_onto*
: *is_onto* *E* *E* *sum_neg*
:= *Ring.sum_neg_onto* `ring`.

Proves that negation is surjective `Definition` *sum_neg_bijective*
: *is_bijective* *E* *E* *sum_neg*
:= *Ring.sum_neg_bijective* `ring`.

Proves that 0's negation is 0. `Definition` *sum_0_neg*
: - 0 = 0
:= *proj2* (*sum_neg_def* 0)
|| *a* = 0 @*a* `by` ← *sum_id_l* (- 0).

Proves that if an element's, x, negation equals 0, x must equal 0. `Definition` *sum_neg_0*
: ∀ *x* : *E*, - *x* = 0 → *x* = 0
:= `fun` *x* *H*
⇒ *proj2* (*sum_neg_def* *x*)
|| *x* + *a* = 0 @*a* `by` ← *H*

|| $a = 0$ @a by ← *sum_id_r* x.

Prove that 0 is the only element whose additive inverse (negation) equals 0. `Definition` *sum_neg_0_uniq*
: *unique* (`fun` $x \Rightarrow$ - $x = 0$) 0
:= *conj sum_0_neg*
(`fun` x $H \Rightarrow$ *eq_sym* (*sum_neg_0* x H)).

Accepts one element, x, and asserts that x is the identity element. `Definition` *prod_id*
: *prod_is_id* 1
:= *Ring.prod_id* `ring`.

Proves that the left identity element is unique. `Definition` *prod_id_l_uniq*
: $\forall$ x : E, (*Monoid.is_id_l* E {#} x) $\rightarrow x = 1$
:= *Ring.prod_id_l_uniq* `ring`.

Proves that the right identity element is unique. `Definition` *prod_id_r_uniq*
: $\forall$ x : E, (*Monoid.is_id_r* E {#} x) $\rightarrow x = 1$
:= *Ring.prod_id_r_uniq* `ring`.

Proves that the identity element is unique. `Definition` *prod_id_uniq*
: $\forall$ x : E, (*Monoid.is_id* E {#} x) $\rightarrow x = 1$
:= *Ring.prod_id_uniq* `ring`.

Proves the left introduction rule. `Definition` *prod_intro_l*
: $\forall$ x y z : E, $x = y \rightarrow z \# x = z \# y$
:= *Ring.prod_intro_l* `ring`.

Proves the right introduction rule. `Definition` *prod_intro_r*
: $\forall$ x y z : E, $x = y \rightarrow x \# z = y \# z$
:= *Ring.prod_intro_r* `ring`.

Accepts two elements, x and y, and asserts that y is x's left inverse. `Definition` *prod_is_inv_l* := *Ring.prod_is_inv_l* `ring`.

Accepts two elements, x and y, and asserts that y is x's right inverse. `Definition` *prod_is_inv_r* := *Ring.prod_is_inv_r* `ring`.

Accepts two elements, x and y, and asserts that y is x's inverse. `Definition` *prod_is_inv* := *Ring.prod_is_inv* `ring`.

Accepts one argument, x, and asserts that x has a left inverse. `Definition` *prod_has_inv_l* := *Ring.prod_has_inv_l* `ring`.

Accepts one argument, x, and asserts that x has a right inverse. `Definition` *prod_has_inv_r* := *Ring.prod_has_inv_r* `ring`.

Accepts one argument, x, and asserts that x has an inverse. `Definition` *prod_has_inv* := *Ring.prod_has_inv* `ring`.

Proves that every left inverse must also be a right inverse. `Definition` *prod_is_inv_lr*
: $\forall$ x y : E, *prod_is_inv_l* x y $\rightarrow$ *prod_is_inv_r* x y

$:=$ `fun` $x\ y\ H$
$\Rightarrow H \mid\mid a = 1$ $@a$ `by` *prod_is_comm* $x\ y$.

Proves that the left and right inverses of an element must be equal. `Definition` *prod_inv_l_r_eq* := *Ring.prod_inv_l_r_eq* `ring`.

Proves that the inverse relationship is symmetric. `Definition` *prod_inv_sym* := *Ring.prod_inv_sym* `ring`.

Proves the left cancellation law for elements possessing a left inverse. `Definition` *prod_cancel_l* := *Ring.prod_cancel_l* `ring`.

Proves the right cancellation law for elements possessing a right inverse. `Definition` *prod_cancel_r* := *Ring.prod_cancel_r* `ring`.

Proves that an element's left inverse is unique. `Definition` *prod_inv_l_uniq* := *Ring.prod_inv_l_uniq* `ring`.

Proves that an element's right inverse is unique. `Definition` *prod_inv_r_uniq* := *Ring.prod_inv_r_uniq* `ring`.

Proves that an element's inverse is unique. `Definition` *prod_inv_uniq* := *Ring.prod_inv_uniq* `ring`.

Proves that 1 is its own left multiplicative inverse. `Definition` *recipr_1_l*
: *prod_is_inv_l* 1 1
:= *Ring.recipr_1_l* `ring`.

Proves that 1 is its own right multiplicative inverse. `Definition` *recipr_1_r*
: *prod_is_inv_r* 1 1
:= *Ring.recipr_1_r* `ring`.

Proves that 1 is its own recriprical. `Definition` *recipr_1*
: *prod_is_inv* 1 1
:= *Ring.recipr_1* `ring`.

Proves that 1 has a left multiplicative inverse. `Definition` *prod_has_inv_l_1*
: *prod_has_inv_l* 1
:= *Ring.prod_has_inv_l_1* `ring`.

Proves that 1 has a right multiplicative inverse. `Definition` *prod_has_inv_r_1*
: *prod_has_inv_r* 1
:= *Ring.prod_has_inv_r_1* `ring`.

Proves that 1 has a reciprical `Definition` *prod_has_inv_1*
: *prod_has_inv* 1
:= *Ring.prod_has_inv_1* `ring`.

Asserts that multiplication is distributive over addition. `Definition` *prod_sum_distrib*
: *Ring.is_distrib* E {#} {+}
:= *Ring.prod_sum_distrib* `ring`.

Proves that 0 times every number equals 0.

0 x = 0 x (0 + 0) x = 0 x 0 x + 0 x = 0 x 0 x = 0 `Definition` *prod_0_l*
: $\forall\ x\ :\ E,\ 0\ \#\ x = 0$
:= *Ring.prod_0_l* `ring`.

Proves that 0 times every number equals 0. `Definition` *prod_0_r*
: $\forall\ x\ :\ E,\ x\ \#\ 0 = 0$
:= *Ring.prod_0_r* `ring`.

Proves that 0 does not have a left multiplicative inverse. `Definition` *prod_0_inv_l*
: ¬ *prod_has_inv_l* 0
:= *Ring.prod_0_inv_l* `ring`.

Proves that 0 does not have a right multiplicative inverse. `Definition` *prod_0_inv_r*
: ¬ *prod_has_inv_r* 0
:= *Ring.prod_0_inv_r* `ring`.

Proves that 0 does not have a multiplicative inverse - I.E. 0 does not have a reciprocal. `Definition` *prod_0_inv*
: ¬ *prod_has_inv* 0
:= *Ring.prod_0_inv* `ring`.

Proves that multiplicative inverses, when they exist are always nonzero. `Definition` *prod_inv_0*
: $\forall\ x\ y\ :\ E,$ *prod_is_inv* $x\ y \rightarrow$ *nonzero* y
:= *Ring.prod_inv_0* `ring`.

Represents -1 and proves that it exists. `Definition` *E_n1_strong*
: { x : E | *sum_is_inv* 1 x }
:= *Ring.E_n1_strong* `ring`.

Represents -1. `Definition` *E_n1* : *E* := *Ring.E_n1* `ring`.

Defines a symbolic representation for -1

Note: here we represent the inverse of 1 rather than the negation of 1. Letter we prove that the negation equals the inverse.

Note: brackets are needed to ensure Coq parses the symbol as a single token instead of a prefixed function call. `Notation "{-1}"` := *E_n1* : *commutative_ring_scope*.

Asserts that -1 is the additive inverse of 1. `Definition` *E_n1_def*
: *sum_is_inv* 1 {-1}
:= *Ring.E_n1_def* `ring`.

Asserts that -1 is the left inverse of 1. `Definition` *E_n1_inv_l*
: *sum_is_inv_l* 1 {-1}
:= *Ring.E_n1_inv_l* `ring`.

Asserts that -1 is the right inverse of 1. `Definition` *E_n1_inv_r*
: *sum_is_inv_r* 1 {-1}
:= *Ring.E_n1_inv_r* `ring`.

Asserts that every additive inverse of 1 must be equal to -1. `Definition` *E_n1_uniq*

: $\forall$ *x* : *E*, *sum_is_inv* 1 *x* $\rightarrow$ *x* = {-1}
:= *Ring.E_n1_uniq* `ring`.

Proves that -1 * x equals the multiplicative inverse of x.

- 1 x + x = 0

- 1 x + 1 x = 0

(-1 + 1) x = 0 0 x = 0 0 = 0 `Definition` *prod_n1_x_inv_l*
: $\forall$ *x* : *E*, *sum_is_inv_l* *x* ({-1} # *x*)
:= *Ring.prod_n1_x_inv_l* `ring`.

Proves that x * -1 equals the multiplicative inverse of x.
x -1 + x = 0 `Definition` *prod_x_n1_inv_l*
: $\forall$ *x* : *E*, *sum_is_inv_l* *x* (*x* # {-1})
:= *Ring.prod_x_n1_inv_l* `ring`.

Proves that x + -1 x = 0. `Definition` *prod_n1_x_inv_r*
: $\forall$ *x* : *E*, *sum_is_inv_r* *x* ({-1} # *x*)
:= *Ring.prod_n1_x_inv_r* `ring`.

Proves that x + x -1 = 0. `Definition` *prod_x_n1_inv_r*
: $\forall$ *x* : *E*, *sum_is_inv_r* *x* (*x* # {-1})
:= *Ring.prod_x_n1_inv_r* `ring`.

Proves that -1 x is the additive inverse of x. `Definition` *prod_n1_x_inv*
: $\forall$ *x* : *E*, *sum_is_inv* *x* ({-1} # *x*)
:= *Ring.prod_n1_x_inv* `ring`.

Proves that x -1 is the additive inverse of x. `Definition` *prod_x_n1_inv*
: $\forall$ *x* : *E*, *sum_is_inv* *x* (*x* # {-1})
:= *Ring.prod_x_n1_inv* `ring`.

Proves that multiplying by -1 is equivalent to negation. `Definition` *prod_n1_neg*
: {#} {-1} = *sum_neg*
:= *Ring.prod_n1_neg* `ring`.

Accepts one element, x, and proves that x -1 equals the additive negation of x. `Definition` *prod_x_n1_neg*
: $\forall$ *x* : *E*, *x* # {-1} = - *x*
:= *Ring.prod_x_n1_neg* `ring`.

Accepts one element, x, and proves that

- 1 x equals the additive negation of x.

`Definition` *prod_n1_x_neg*
: $\forall$ *x* : *E*, {-1} # *x* = - *x*
:= *Ring.prod_n1_x_neg* `ring`.

Proves that -1 x = x -1. `Definition` *prod_n1_eq*
: $\forall$ *x* : *E*, {-1} # *x* = *x* # {-1}
:= *Ring.prod_n1_eq* `ring`.

Proves that the additive negation of 1 equals -1. `Definition` *neg_1*
: {-} 1 = {-1}
:= *Ring.neg_1* `ring`.

Proves that the additive negation of -1 equals 1. `Definition` *neg_n1*
: *sum_neg* {-1} = 1
:= *Ring.neg_n1* `ring`.

Proves that -1 * -1 = 1.

- 1 * -1 = -1 * -1
- 1 * -1 = prod -1 -1
- 1 * -1 = sum_neg -1
- 1 * -1 = 1

`Definition` *prod_n1_n1*
: {-1} # {-1} = 1
:= *Ring.prod_n1_n1* `ring`.

Proves that -1 is its own multiplicative inverse. `Definition` *E_n1_inv*
: *prod_is_inv* {-1} {-1}
:= *Ring.E_n1_inv* `ring`.

`End` *Theorems*.

`End` *Commutative_Ring*.

Chapter 4

Library functional-algebra.ring

This module defines the Ring record type which can be used to represent algebraic rings and provides a collection of axioms and theorems describing them.

```
Require Import Description.
Require Import FunctionalExtensionality.
Require Import base.
Require Import function.
Require Import monoid.
Require Import group.
Require Import abelian_group.

Module Ring.

Close Scope nat_scope.
```

Accepts two binary functions, f and g, and asserts that f is left distributive over g.

```
Definition is_distrib_l (T : Type) (f g : T → T → T)
  : Prop
  := ∀ x y z : T, f x (g y z) = g (f x y) (f x z).
```

Accepts two binary functions, f and g, and asserts that f is right distributive over g.

```
Definition is_distrib_r (T : Type) (f g : T → T → T)
  : Prop
  := ∀ x y z : T, f (g y z) x = g (f y x) (f z x).
```

Accepts two binary functions, f and g, and asserts that f is distributive over g.

```
Definition is_distrib (T : Type) (f g : T → T → T)
  : Prop
  := is_distrib_l T f g ∧ is_distrib_r T f g.
```

Represents algebraic rings `Structure Ring : Type := ring {`

Represents the set of ring elements. `E : Set;`

Represents 0 - the additive identity. `E_0 : E;`

Represents 1 - the multiplicative identity. $E\_1 : E$;

Represents addition. $sum : E \to E \to E$;

Represents multiplication. $prod : E \to E \to E$;

Asserts that 0 /= 1. $distinct\_0\_1 : E\_0 \neq E\_1$;

Asserts that addition is associative. $sum\_is\_assoc : Monoid.is\_assoc\ E\ sum$;

Asserts that addition is commutative. $sum\_is\_comm : Abelian\_Group.is\_comm\ E\ sum$;

Asserts that E_0 is the left identity element. $sum\_id\_l : Monoid.is\_id\_l\ E\ sum\ E\_0$;

Asserts that every element has an additive inverse. $sum\_inv\_l\_ex : \forall\ x : E, \exists\ y : E, sum\ y\ x = E\_0$;

Asserts that multiplication is associative. $prod\_is\_assoc : Monoid.is\_assoc\ E\ prod$;

Asserts that 1 is the left identity element. $prod\_id\_l : Monoid.is\_id\_l\ E\ prod\ E\_1$;

Asserts that 1 is the right identity element. $prod\_id\_r : Monoid.is\_id\_r\ E\ prod\ E\_1$;

Asserts that multiplication is left distributive over addition. $prod\_sum\_distrib\_l : is\_distrib\_l\ E\ prod\ sum$;

Asserts that multiplication is right distributive over addition. $prod\_sum\_distrib\_r : is\_distrib\_r\ E\ prod\ sum$
}.

Enable implicit arguments for ring properties.

Arguments E_0 {r}.

Arguments E_1 {r}.

Arguments sum {r} x y.

Arguments prod {r} x y.

Arguments distinct_0_1 {r} _.

Arguments sum_is_assoc {r} x y z.

Arguments sum_is_comm {r} x y.

Arguments sum_id_l {r} x.

Arguments sum_inv_l_ex {*r*} *x.*

Arguments prod_is_assoc {*r*} *x y z.*

Arguments prod_id_l {*r*} *x.*

Arguments prod_id_r {*r*} *x.*

Arguments prod_sum_distrib_l {*r*} *x y z.*

Arguments prod_sum_distrib_r {*r*} *x y z.*

Define notations for ring properties.

`Notation` "0" := *E_0* : *ring_scope.*

`Notation` "1" := *E_1* : *ring_scope.*

`Notation` "x + y" := (*sum x y*) (`at level` 50, `left associativity`) : *ring_scope.*

`Notation` "{+}" := *sum* : *ring_scope.*

`Notation` "x # y" := (*prod x y*) (`at level` 50, `left associativity`) : *ring_scope.*

`Notation` "{#}" := *prod* : *ring_scope.*

`Open Scope` *ring_scope.*

`Section` *Theorems.*

Represents an arbitrary ring.
Note: we use Variable rather than Parameter to ensure that the following theorems are generalized w.r.t r. `Variable` *r* : *Ring.*

Represents the set of group elements.
Note: We use Let to define E as a local abbreviation. `Let` *E* := *E r.*

A predicate that accepts one element, x, and asserts that x is nonzero. `Definition` *nonzero* (*x* : *E*) : `Prop` := $x \neq 0$.

Accepts one ring element, x, and asserts that x is the left identity element. `Definition` *sum_is_id_l* := *Monoid.is_id_l E sum.*

Accepts one ring element, x, and asserts that x is the right identity element. `Definition` *sum_is_id_r* := *Monoid.is_id_r E sum.*

Accepts one ring element, x, and asserts that x is the identity element. `Definition` *sum_is_id* := *Monoid.is_id E sum.*

Represents the abelian group formed by addition over E. `Definition` *sum_abelian_group* := *Abelian_Group.abelian_group E* 0 {+} *sum_is_assoc sum_is_comm sum_id_l sum_inv_l_ex.*

Represents the group formed by addition over E. `Definition` *sum_group* := *Abelian_Group.op_group sum_abelian_group.*

Represents the monoid formed by addition over E. `Definition` *sum_monoid* := *Abelian_Group.op_monoid sum_abelian_group.*

Proves that 0 is the right identity element. `Definition` *sum_id_r*

: $sum\_is\_id\_r\ 0$
:= $Abelian\_Group.op\_id\_r\ sum\_abelian\_group.$

Proves that 0 is the identity element. `Definition` sum_id
: $sum\_is\_id\ 0$
:= $Abelian\_Group.op\_id\ sum\_abelian\_group.$

Accepts two elements, x and y, and asserts that y is x's left inverse. `Definition` $sum\_is\_inv\_l$
:= $Abelian\_Group.op\_is\_inv\_l\ sum\_abelian\_group.$

Accepts two elements, x and y, and asserts that y is x's right inverse. `Definition` $sum\_is\_inv\_r$
:= $Abelian\_Group.op\_is\_inv\_r\ sum\_abelian\_group.$

Accepts two elements, x and y, and asserts that y is x's inverse. `Definition` sum_is_inv
:= $Abelian\_Group.op\_is\_inv\ sum\_abelian\_group.$

Asserts that every element has a right inverse. `Definition` $sum\_inv\_r\_ex$
: $\forall\ x : E, \exists\ y : E,\ sum\_is\_inv\_r\ x\ y$
:= $Abelian\_Group.op\_inv\_r\_ex\ sum\_abelian\_group.$

Proves that the left identity element is unique. `Definition` $sum\_id\_l\_uniq$
: $\forall\ x : E,\ Monoid.is\_id\_l\ E\ \{+\}\ x \rightarrow x = 0$
:= $Abelian\_Group.op\_id\_l\_uniq\ sum\_abelian\_group.$

Proves that the right identity element is unique. `Definition` $sum\_id\_r\_uniq$
: $\forall\ x : E,\ Monoid.is\_id\_r\ E\ \{+\}\ x \rightarrow x = 0$
:= $Abelian\_Group.op\_id\_r\_uniq\ sum\_abelian\_group.$

Proves that the identity element is unique. `Definition` sum_id_uniq
: $\forall\ x : E,\ Monoid.is\_id\ E\ \{+\}\ x \rightarrow x = 0$
:= $Abelian\_Group.op\_id\_uniq\ sum\_abelian\_group.$

Proves that for every group element, x, its left and right inverses are equal. `Definition` $sum\_inv\_l\_r\_eq$
: $\forall\ x\ y : E,\ sum\_is\_inv\_l\ x\ y \rightarrow \forall\ z : E,\ sum\_is\_inv\_r\ x\ z \rightarrow y = z$
:= $Abelian\_Group.op\_inv\_l\_r\_eq\ sum\_abelian\_group.$

Proves that the inverse relation is symmetrical. `Definition` sum_inv_sym
: $\forall\ x\ y : E,\ sum\_is\_inv\ x\ y \leftrightarrow sum\_is\_inv\ y\ x$
:= $Abelian\_Group.op\_inv\_sym\ sum\_abelian\_group.$

Proves that an element's inverse is unique. `Definition` sum_inv_uniq
: $\forall\ x\ y\ z : E,\ sum\_is\_inv\ x\ y \rightarrow sum\_is\_inv\ x\ z \rightarrow z = y$
:= $Abelian\_Group.op\_inv\_uniq\ sum\_abelian\_group.$

Proves that every group element has an inverse. `Definition` sum_inv_ex
: $\forall\ x : E, \exists\ y : E,\ sum\_is\_inv\ x\ y$
:= $Abelian\_Group.op\_inv\_ex\ sum\_abelian\_group.$

Proves explicitly that every element has a unique inverse. `Definition` $sum\_inv\_uniq\_ex$

: $\forall\ x\ :\ E, \exists!\ y\ :\ E,\ sum\_is\_inv\ x\ y$
$:= Abelian\_Group.op\_inv\_uniq\_ex\ sum\_abelian\_group.$

Proves the left introduction rule. `Definition` *sum_intro_l*
: $\forall\ x\ y\ z\ :\ E,\ x = y \rightarrow z + x = z + y$
$:= Abelian\_Group.op\_intro\_l\ sum\_abelian\_group.$

Proves the right introduction rule. `Definition` *sum_intro_r*
: $\forall\ x\ y\ z\ :\ E,\ x = y \rightarrow x + z = y + z$
$:= Abelian\_Group.op\_intro\_r\ sum\_abelian\_group.$

Proves the left cancellation rule. `Definition` *sum_cancel_l*
: $\forall\ x\ y\ z\ :\ E,\ z + x = z + y \rightarrow x = y$
$:= Abelian\_Group.op\_cancel\_l\ sum\_abelian\_group.$

Proves the right cancellation rule. `Definition` *sum_cancel_r*
: $\forall\ x\ y\ z\ :\ E,\ x + z = y + z \rightarrow x = y$
$:= Abelian\_Group.op\_cancel\_r\ sum\_abelian\_group.$

Proves that an element's left inverse is unique. `Definition` *sum_inv_l_uniq*
: $\forall\ x\ y\ z\ :\ E,\ sum\_is\_inv\_l\ x\ y \rightarrow sum\_is\_inv\_l\ x\ z \rightarrow z = y$
$:= Abelian\_Group.op\_inv\_l\_uniq\ sum\_abelian\_group.$

Proves that an element's right inverse is unique. `Definition` *sum_inv_r_uniq*
: $\forall\ x\ y\ z\ :\ E,\ sum\_is\_inv\_r\ x\ y \rightarrow sum\_is\_inv\_r\ x\ z \rightarrow z = y$
$:= Abelian\_Group.op\_inv\_r\_uniq\ sum\_abelian\_group.$

TODO: move the following theorems about 0 to monoid.
Proves that 0 is its own left additive inverse. `Definition` *sum_0_inv_l*
: $sum\_is\_inv\_l\ 0\ 0$
$:= sum\_id\_l\ 0.$

Proves that 0 is its own right additive inverse. `Definition` *sum_0_inv_r*
: $sum\_is\_inv\_r\ 0\ 0$
$:= sum\_id\_r\ 0.$

Proves that 0 is it's own additive inverse. `Definition` *sum_0_inv*
: $sum\_is\_inv\ 0\ 0$
$:= conj\ sum\_0\_inv\_l\ sum\_0\_inv\_r.$

Proves that if an element's, x, additive inverse equals 0, x equals 0. `Definition` *sum_inv_0*
: $\forall\ x\ :\ E,\ sum\_is\_inv\ x\ 0 \rightarrow x = 0$
:= `fun` $x\ H$
$\Rightarrow proj1\ H$
$||\ a = 0$ @a `by` $\leftarrow sum\_id\_l\ x.$

Proves that 0 is the only element whose additive inverse is 0. `Definition` *sum_inv_0_uniq*
: $unique$ (`fun` $x \Rightarrow sum\_is\_inv\ x\ 0)\ 0$
$:= conj\ sum\_0\_inv$

(`fun` x H $\Rightarrow$ *eq_sym* (*sum_inv_0* x H)).

Represents strongly-specified negation. `Definition` *sum_neg_strong*
: $\forall$ x : E, { y | *sum_is_inv* x y }
:= *Abelian_Group.op_neg_strong* *sum_abelian_group*.

Represents negation. `Definition` *sum_neg*
: $E \rightarrow E$
:= *Abelian_Group.op_neg* *sum_abelian_group*.

`Notation` "{-}" := (*sum_neg*) : *ring_scope*.

`Notation` "- x" := (*sum_neg* x) : *ring_scope*.

Asserts that the negation returns the inverse of its argument. `Definition` *sum_neg_def*

: $\forall$ x : E, *sum_is_inv* x (- x)
:= *Abelian_Group.op_neg_def* *sum_abelian_group*.

Proves that negation is one-to-one `Definition` *sum_neg_inj*
: *is_injective* E E {-}
:= *Abelian_Group.op_neg_inj* *sum_abelian_group*.

Proves the cancellation property for negation. `Definition` *sum_cancel_neg*
: $\forall$ x : E, - (- x) = x
:= *Abelian_Group.op_cancel_neg* *sum_abelian_group*.

Proves that negation is onto `Definition` *sum_neg_onto*
: *is_onto* E E {-}
:= *Abelian_Group.op_neg_onto* *sum_abelian_group*.

Proves that negation is surjective `Definition` *sum_neg_bijective*
: *is_bijective* E E {-}
:= *Abelian_Group.op_neg_bijective* *sum_abelian_group*.

Proves that neg x = y -> neg y = x `Definition` *sum_neg_rev*
: $\forall$ x y : E, - x = y $\rightarrow$ - y = x
:= *Abelian_Group.op_neg_rev* *sum_abelian_group*.

Accepts one element, x, and asserts that x is the left identity element. `Definition` *prod_is_id_l* := *Monoid.is_id_l* E *prod*.

Accepts one element, x, and asserts that x is the right identity element. `Definition` *prod_is_id_r* := *Monoid.is_id_r* E *prod*.

Accepts one element, x, and asserts that x is the identity element. `Definition` *prod_is_id* := *Monoid.is_id* E *prod*.

Represents the monoid formed by op over E. `Definition` *prod_monoid* := *Monoid.monoid* E 1 {#} *prod_is_assoc* *prod_id_l* *prod_id_r*.

Proves that 1 is the identity element. `Definition` *prod_id*
: *prod_is_id* 1

$:= Monoid.op\_id\ prod\_monoid.$

Proves that the left identity element is unique. `Definition` $prod\_id\_l\_uniq$
$: \forall\ x\ :\ E, (Monoid.is\_id\_l\ E\ prod\ x) \to x = 1$
$:= Monoid.op\_id\_l\_uniq\ prod\_monoid.$

Proves that the right identity element is unique. `Definition` $prod\_id\_r\_uniq$
$: \forall\ x\ :\ E, (Monoid.is\_id\_r\ E\ prod\ x) \to x = 1$
$:= Monoid.op\_id\_r\_uniq\ prod\_monoid.$

Proves that the identity element is unique. `Definition` $prod\_id\_uniq$
$: \forall\ x\ :\ E, (Monoid.is\_id\ E\ prod\ x) \to x = 1$
$:= Monoid.op\_id\_uniq\ prod\_monoid.$

Proves the left introduction rule. `Definition` $prod\_intro\_l$
$: \forall\ x\ y\ z\ :\ E, x = y \to z \# x = z \# y$
$:= Monoid.op\_intro\_l\ prod\_monoid.$

Proves the right introduction rule. `Definition` $prod\_intro\_r$
$: \forall\ x\ y\ z\ :\ E, x = y \to x \# z = y \# z$
$:= Monoid.op\_intro\_r\ prod\_monoid.$

Accepts two elements, x and y, and asserts that y is x's left inverse. `Definition` $prod\_is\_inv\_l := Monoid.op\_is\_inv\_l\ prod\_monoid.$

Accepts two elements, x and y, and asserts that y is x's right inverse. `Definition` $prod\_is\_inv\_r := Monoid.op\_is\_inv\_r\ prod\_monoid.$

Accepts two elements, x and y, and asserts that y is x's inverse. `Definition` $prod\_is\_inv := Monoid.op\_is\_inv\ prod\_monoid.$

Accepts one argument, x, and asserts that x has a left inverse. `Definition` $prod\_has\_inv\_l := Monoid.has\_inv\_l\ prod\_monoid.$

Accepts one argument, x, and asserts that x has a right inverse. `Definition` $prod\_has\_inv\_r := Monoid.has\_inv\_r\ prod\_monoid.$

Accepts one argument, x, and asserts that x has an inverse. `Definition` $prod\_has\_inv := Monoid.has\_inv\ prod\_monoid.$

Proves that the left and right inverses of an element must be equal. `Definition` $prod\_inv\_l\_r\_eq := Monoid.op\_inv\_l\_r\_eq\ prod\_monoid.$

Proves that the inverse relationship is symmetric. `Definition` $prod\_inv\_sym := Monoid.op\_inv\_sym\ prod\_monoid.$

Proves the left cancellation law for elements possessing a left inverse. `Definition` $prod\_cancel\_l := Monoid.op\_cancel\_l\ prod\_monoid.$

Proves the right cancellation law for elements possessing a right inverse. `Definition` $prod\_cancel\_r := Monoid.op\_cancel\_r\ prod\_monoid.$

Proves that an element's left inverse is unique. `Definition` $prod\_inv\_l\_uniq := Monoid.op\_inv\_l\_uniq\ prod\_monoid.$

Proves that an element's right inverse is unique. `Definition` *prod_inv_r_uniq* := *Monoid.op_inv_r_uniq prod_monoid*.

Proves that an element's inverse is unique. `Definition` *prod_inv_uniq* := *Monoid.op_inv_uniq prod_monoid*.

Proves that 1 is its own left multiplicative inverse. `Definition` *recipr_1_l*
: *prod_is_inv_l* 1 1
:= *Monoid.op_inv_0_l prod_monoid*.

Proves that 1 is its own right multiplicative inverse. `Definition` *recipr_1_r*
: *prod_is_inv_r* 1 1
:= *Monoid.op_inv_0_r prod_monoid*.

Proves that 1 is its own recriprical. `Definition` *recipr_1*
: *prod_is_inv* 1 1
:= *Monoid.op_inv_0 prod_monoid*.

Proves that 1 has a left multiplicative inverse. `Definition` *prod_has_inv_l_1*
: *prod_has_inv_l* 1
:= *Monoid.op_has_inv_l_0 prod_monoid*.

Proves that 1 has a right multiplicative inverse. `Definition` *prod_has_inv_r_1*
: *prod_has_inv_r* 1
:= *Monoid.op_has_inv_r_0 prod_monoid*.

Proves that 1 has a reciprical `Definition` *prod_has_inv_1*
: *prod_has_inv* 1
:= *Monoid.op_has_inv_0 prod_monoid*.

TODO Reciprical functions (op_neg) from Monoid.

Asserts that multiplication is distributive over addition. `Definition` *prod_sum_distrib*
: *is_distrib E prod sum*
:= *conj prod_sum_distrib_l prod_sum_distrib_r*.

Proves that 0 times every number equals 0.

0 x = 0 x (0 + 0) x = 0 x 0 x + 0 x = 0 x 0 x = 0 `Definition` *prod_0_l*
: $\forall\ x : E,\ 0 \# x = 0$
:= `fun` x
  $\Rightarrow$ `let` H
    : $(0 \# x) + (0 \# x) = (0 \# x) + 0$
    := *eq_refl* $(0 \# x)$
      $||\ a \# x = 0 \# x$ @a `by` (*sum_id_l* 0)
      $||\ a = 0 \# x$ @a `by` $\leftarrow$ *prod_sum_distrib_r* x 0 0
      $||\ (0 \# x) + (0 \# x) = a$ @a `by` *sum_id_r* $(0 \# x)$
  `in` *sum_cancel_l* $(0 \# x)$ 0 $(0 \# x)$ H.

Proves that 0 times every number equals 0. `Definition` *prod_0_r*
: $\forall\ x : E,\ x \# 0 = 0$
:= `fun` x

```
  ⇒ let H
      : (x # 0) + (x # 0) = 0 + (x # 0)
      := eq_refl (x # 0)
        || x # a = x # 0 @a by sum_id_r 0
        || a = x # 0 @a by ← prod_sum_distrib_l x 0 0
        || (x # 0) + (x # 0) = a @a by sum_id_l (x # 0)
    in sum_cancel_r (x # 0) 0 (x # 0) H.
```

Proves that 0 does not have a left multiplicative inverse. `Definition` *prod_0_inv_l*

```
: ¬ prod_has_inv_l 0
:= ex_ind
     (fun x (H : x # 0 = 1)
       ⇒ distinct_0_1 (H || a = 1 @a by ← prod_0_r x)).
```

Proves that 0 does not have a right multiplicative inverse. `Definition` *prod_0_inv_r*

```
: ¬ prod_has_inv_r 0
:= ex_ind
     (fun x (H : 0 # x = 1)
       ⇒ distinct_0_1 (H || a = 1 @a by ← prod_0_l x)).
```

Proves that 0 does not have a multiplicative inverse - I.E. 0 does not have a reciprocal. `Definition` *prod_0_inv*

```
: ¬ prod_has_inv 0
:= ex_ind
     (fun x H ⇒ prod_0_inv_l (ex_intro (fun x ⇒ prod_is_inv_l 0 x) x (proj1 H))).
```

Proves that multiplicative inverses, when they exist are always nonzero. `Definition` *prod_inv_0*

```
: ∀ x y : E, prod_is_inv x y → nonzero y
:= fun x y H (H0 : y = 0)
     ⇒ distinct_0_1
         (proj1 H
           || a # x = 1 @a by ← H0
           || a = 1 @a by ← prod_0_l x).
```

Represents -1 and proves that it exists. `Definition` *E_n1_strong*

```
: { x : E | sum_is_inv 1 x }
:= constructive_definite_description (sum_is_inv 1) (sum_inv_uniq_ex 1).
```

Represents -1. `Definition` *E_n1 : E := proj1_sig E_n1_strong.*

Defines a symbolic representation for -1

Note: here we represent the inverse of 1 rather than the negation of 1. Letter we prove that the negation equals the inverse.

Note: brackets are needed to ensure Coq parses the symbol as a single token instead of a prefixed function call. `Notation "{-1}" :=` *E_n1 : ring_scope.*

Asserts that -1 is the additive inverse of 1. `Definition` *E_n1_def*

: *sum_is_inv* 1 {-1}
:= *proj2_sig E_n1_strong.*

Asserts that -1 is the left inverse of 1. `Definition` *E_n1_inv_l*
: *sum_is_inv_l* 1 {-1}
:= *proj1 E_n1_def.*

Asserts that -1 is the right inverse of 1. `Definition` *E_n1_inv_r*
: *sum_is_inv_r* 1 {-1}
:= *proj2 E_n1_def.*

Asserts that every additive inverse of 1 must be equal to -1. `Definition` *E_n1_uniq*
: $\forall$ *x* : *E*, *sum_is_inv* 1 *x* $\rightarrow$ *x* = {-1}
:= `fun` *x* $\Rightarrow$ *sum_inv_uniq* 1 {-1} *x E_n1_def.*

Proves that -1 * x equals the multiplicative inverse of x.

- 1 x + x = 0
- 1 x + 1 x = 0

(-1 + 1) x = 0 0 x = 0 0 = 0 `Definition` *prod_n1_x_inv_l*
: $\forall$ *x* : *E*, *sum_is_inv_l x* ({-1} # *x*)
:= `fun` *x*
$\Rightarrow$ *prod_0_l x*
|| *a* # *x* = 0 @*a* `by` *E_n1_inv_l*
|| *a* = 0 @*a* `by` $\leftarrow$ *prod_sum_distrib_r x* {-1} 1
|| ({-1} # *x*) + *a* = 0 @*a* `by` $\leftarrow$ *prod_id_l x.*

Proves that x * -1 equals the multiplicative inverse of x.
x -1 + x = 0 `Definition` *prod_x_n1_inv_l*
: $\forall$ *x* : *E*, *sum_is_inv_l x* (*x* # {-1})
:= `fun` *x*
$\Rightarrow$ *prod_0_r x*
|| *x* # *a* = 0 @*a* `by` *E_n1_inv_l*
|| *a* = 0 @*a* `by` $\leftarrow$ *prod_sum_distrib_l x* {-1} 1
|| (*x* # {-1}) + *a* = 0 @*a* `by` $\leftarrow$ *prod_id_r x.*

Proves that x + -1 x = 0. `Definition` *prod_n1_x_inv_r*
: $\forall$ *x* : *E*, *sum_is_inv_r x* ({-1} # *x*)
:= `fun` *x*
$\Rightarrow$ *prod_0_l x*
|| *a* # *x* = 0 @*a* `by` *E_n1_inv_r*
|| *a* = 0 @*a* `by` $\leftarrow$ *prod_sum_distrib_r x* 1 {-1}
|| *a* + ({-1} # *x*) = 0 @*a* `by` $\leftarrow$ *prod_id_l x.*

Proves that x + x -1 = 0. `Definition` *prod_x_n1_inv_r*
: $\forall$ *x* : *E*, *sum_is_inv_r x* (*x* # {-1})

$:=$ `fun` x
$\Rightarrow$ *prod_0_r x*
|| x # $a = 0$ @a `by` *E_n1_inv_r*
|| $a = 0$ @a `by` $\leftarrow$ *prod_sum_distrib_l* x 1 {-1}
|| a + (x # {-1}) = 0 @a `by` $\leftarrow$ *prod_id_r x.*

Proves that -1 x is the additive inverse of x. `Definition` *prod_n1_x_inv*
: $\forall$ x : E, *sum_is_inv* x ({-1} # x)
:= `fun` $x \Rightarrow$ *conj* (*prod_n1_x_inv_l* x) (*prod_n1_x_inv_r* x).

Proves that x -1 is the additive inverse of x. `Definition` *prod_x_n1_inv*
: $\forall$ x : E, *sum_is_inv* x (x # {-1})
:= `fun` $x \Rightarrow$ *conj* (*prod_x_n1_inv_l* x) (*prod_x_n1_inv_r* x).

Proves that multiplying by -1 is equivalent to negation. `Definition` *prod_n1_neg*
: *prod* {-1} = {-}
:= *functional_extensionality*
(*prod* {-1}) {-}
(`fun` x
$\Rightarrow$ *sum_inv_uniq* x (- x) ({-1} # x)
(*sum_neg_def* x)
(*prod_n1_x_inv* x)).

Accepts one element, x, and proves that x -1 equals the additive negation of x. `Definition` *prod_x_n1_neg*
: $\forall$ x : E, x # {-1} = - x
:= `fun` x
$\Rightarrow$ *sum_inv_uniq* x (- x) (x # {-1})
(*sum_neg_def* x)
(*prod_x_n1_inv* x).

Accepts one element, x, and proves that

- 1 x equals the additive negation of x.

`Definition` *prod_n1_x_neg*
: $\forall$ x : E, {-1} # x = - x
:= `fun` x
$\Rightarrow$ *sum_inv_uniq* x (- x) ({-1} # x)
(*sum_neg_def* x)
(*prod_n1_x_inv* x).

Proves that -1 x = x -1. `Definition` *prod_n1_eq*
: $\forall$ x : E, {-1} # x = x # {-1}
:= `fun` x
$\Rightarrow$ *sum_inv_uniq* x (x # {-1}) ({-1} # x)
(*prod_x_n1_inv* x)

(*prod_n1_x_inv* x).

Proves that the additive negation of 1 equals -1. `Definition` *neg_1*
: {-} 1 = {-1}
:= *eq_refl* ({-} 1)
|| {-} 1 = a @a `by` *prod_x_n1_neg* 1
|| {-} 1 = a @a `by` ← *prod_id_l* {-1}.

Proves that the additive negation of -1 equals 1. `Definition` *neg_n1*
: {-} {-1} = 1
:= *sum_neg_rev* 1 {-1} *neg_1*.

Proves that -1 * -1 = 1.

- 1 * -1 = -1 * -1
- 1 * -1 = prod -1 -1
- 1 * -1 = - -1
- 1 * -1 = 1

`Definition` *prod_n1_n1*
: {-1} # {-1} = 1
:= *eq_refl* ({-1} # {-1})
|| {-1} # {-1} = a @a `by` ← *prod_n1_x_neg* {-1}
|| {-1} # {-1} = a @a `by` ← *neg_n1*.

Proves that -1 is its own multiplicative inverse. `Definition` *E_n1_inv*
: *prod_is_inv* {-1} {-1}
:= *conj prod_n1_n1 prod_n1_n1*.

`End` *Theorems*.

`End` *Ring*.

Chapter 5

Library functional-algebra.field

This module defines the Field record type which can be used to represent algebraic fields and provides a collection of axioms and theorems describing them.

Algebraic fields are rings in which every *non-zero* element has a multiplicative inverse. The subset of elements that have inverses form a group w.r.t multiplication.

```
Require Import Eqdep.
Require Import Description.

Require Import base.
Require Import function.
Require Import monoid.
Require Import monoid_group.
Require Import group.
Require Import abelian_group.
Require Import ring.
Require Import commutative_ring.

Module Field.
```

Represents algebraic fields. `Structure Field : Type := field {`

Represents the set of elements. `E: Set;`

Represents 0 - the additive identity. `E_0: E;`

Represents 1 - the multiplicative identity. `E_1: E;`

Represents addition. `sum: E → E → E;`

Represents multiplication. `prod: E → E → E;`

Asserts that 0 <> 1. `distinct_0_1: E_0 ≠ E_1;`

Asserts that addition is associative. $sum\_is\_assoc : Monoid.is\_assoc\ E\ sum$;

Asserts that addition is commutative. $sum\_is\_comm : Abelian\_Group.is\_comm\ E\ sum$;

Asserts that 0 is the left identity element. $sum\_id\_l : Monoid.is\_id\_l\ E\ sum\ E\_0$;

Asserts that every element has an additive inverse. $sum\_inv\_l\_ex : \forall\ x : E, \exists\ y : E, sum\ y\ x = E\_0$;

Asserts that multiplication is associative. $prod\_is\_assoc : Monoid.is\_assoc\ E\ prod$;

Asserts that multiplication is commutative. $prod\_is\_comm : Abelian\_Group.is\_comm\ E\ prod$;

Asserts that 1 is the left identity element. $prod\_id\_l : Monoid.is\_id\_l\ E\ prod\ E\_1$;

Asserts that every *non-zero* element has a multiplicative inverse.
Note: this is the property that distinguishes fields from commutative rings. $prod\_inv\_l\_ex : \forall\ x : E, x \neq E\_0 \rightarrow \exists\ y : E, prod\ y\ x = E\_1$;

Asserts that multiplication is left distributive over addition. $prod\_sum\_distrib\_l : Ring.is\_distrib\_l\ E\ prod\ sum$
}.

Enable implicit arguments for field properties.

Arguments E_0 {f}.

Arguments E_1 {f}.

Arguments sum {f} x y.

Arguments prod {f} x y.

Arguments distinct_0_1 {f} _.

Arguments sum_is_assoc {f} x y z.

Arguments sum_is_comm {f} x y.

Arguments sum_id_l {f} x.

Arguments sum_inv_l_ex {f} x.

Arguments prod_is_assoc {f} x y z.

Arguments prod_is_comm {f} x y.

Arguments prod_id_l {f} x.

Arguments prod_inv_l_ex {f} x _.

Arguments prod_sum_distrib_l {*f*} *x y z.*

Define notations for field properties.

`Notation` "0" := *E_0* : *field_scope.*

`Notation` "1" := *E_1* : *field_scope.*

`Notation` "x + y" := (*sum x y*) (`at level` 50, `left associativity`) : *field_scope.*

`Notation` "{+}" := *sum* : *field_scope.*

`Notation` "x # y" := (*prod x y*) (`at level` 50, `left associativity`) : *field_scope.*

`Notation` "{#}" := *prod* : *field_scope.*

`Open Scope` *field_scope.*

`Section` *Theorems.*

Represents an arbitrary commutative ring.
Note: we use Variable rather than Parameter to ensure that the following theorems are generalized w.r.t r. `Variable` *f* : *Field.*

Represents the set of group elements.
Note: We use Let to define E as a local abbreviation. `Let` *E* := *E f.*

Accepts one ring element, x, and asserts that x is the left identity element. `Definition` *sum_is_id_l* := *Monoid.is_id_l E* {+}.

Accepts one ring element, x, and asserts that x is the right identity element. `Definition` *sum_is_id_r* := *Monoid.is_id_r E* {+}.

Accepts one ring element, x, and asserts that x is the identity element. `Definition` *sum_is_id* := *Monoid.is_id E* {+}.

Accepts one ring element, x, and asserts that x is the left identity element. `Definition` *prod_is_id_l* := *Monoid.is_id_l E* {#}.

Accepts one ring element, x, and asserts that x is the right identity element. `Definition` *prod_is_id_r* := *Monoid.is_id_r E* {#}.

Accepts one ring element, x, and asserts that x is the identity element. `Definition` *prod_is_id* := *Monoid.is_id E* {#}.

Represents the commutative ring that addition and multiplication form over E. `Definition` *commutative_ring*
:= *Commutative_Ring.commutative_ring E* 0 1 {+} {#}
distinct_0_1 sum_is_assoc sum_is_comm sum_id_l sum_inv_l_ex
prod_is_assoc prod_is_comm prod_id_l prod_sum_distrib_l.

Represents the non-commutative ring formed by addition and multiplication over E. `Definition ring` := *Commutative_Ring.ring commutative_ring.*

Represents the abelian group formed by addition over E. `Definition` *sum_abelian_group* := *Commutative_Ring.sum_abelian_group commutative_ring.*

Represents the abelian group formed by addition over E. `Definition` *sum_group* := *Commutative_Ring.sum_group commutative_ring.*

Represents the monoid formed by addition over E. `Definition` *sum_monoid* := *Commutative_Ring.sum_monoid commutative_ring.*

Represents the monoid formed by multiplication over E. `Definition` *prod_monoid* := *Commutative_Ring.prod_monoid commutative_ring.*

Proves that 1 <> 0. `Definition` *distinct_1_0*
: $1 \neq 0$
:= *Commutative_Ring.distinct_1_0 commutative_ring.*

A predicate that accepts one element, x, and asserts that x is nonzero. `Definition` *nonzero*
: $E \rightarrow$ `Prop`
:= *Commutative_Ring.nonzero commutative_ring.*

Proves that 0 is the right identity element. `Definition` *sum_id_r*
: *sum_is_id_r* 0
:= *Commutative_Ring.sum_id_r commutative_ring.*

Proves that 0 is the identity element. `Definition` *sum_id* := *Commutative_Ring.sum_id commutative_ring.*

Accepts two elements, x and y, and asserts that y is x's left inverse. `Definition` *sum_is_inv_l* := *Monoid.is_inv_l E* {+} 0 *sum_id.*

Accepts two elements, x and y, and asserts that y is x's right inverse. `Definition` *sum_is_inv_r* := *Monoid.is_inv_r E* {+} 0 *sum_id.*

Accepts two elements, x and y, and asserts that y is x's inverse. `Definition` *sum_is_inv* := *Monoid.is_inv E* {+} 0 *sum_id.*

Asserts that every element has a right inverse. `Definition` *sum_inv_r_ex*
: $\forall\ x : E, \exists\ y : E,$ *sum_is_inv_r x y*
:= *Commutative_Ring.sum_inv_r_ex commutative_ring.*

Proves that the left identity element is unique. `Definition` *sum_id_l_uniq*
: $\forall\ x : E,$ *Monoid.is_id_l E* {+} $x \rightarrow x = 0$
:= *Commutative_Ring.sum_id_l_uniq commutative_ring.*

Proves that the right identity element is unique. `Definition` *sum_id_r_uniq*
: $\forall\ x : E,$ *Monoid.is_id_r E* {+} $x \rightarrow x = 0$
:= *Commutative_Ring.sum_id_r_uniq commutative_ring.*

Proves that the identity element is unique. `Definition` *sum_id_uniq*
: $\forall\ x : E,$ *Monoid.is_id E* {+} $x \rightarrow x = 0$
:= *Commutative_Ring.sum_id_uniq commutative_ring.*

Proves that for every group element, x, its left and right inverses are equal. `Definition` *sum_inv_l_r_eq*

$: \forall\ x\ y\ :\ E,\ sum\_is\_inv\_l\ x\ y \rightarrow \forall\ z\ :\ E,\ sum\_is\_inv\_r\ x\ z \rightarrow y = z$
$:= Commutative\_Ring.sum\_inv\_l\_r\_eq\ commutative\_ring.$

Proves that the inverse relation is symmetrical. `Definition` *sum_inv_sym*
$: \forall\ x\ y\ :\ E,\ sum\_is\_inv\ x\ y \leftrightarrow sum\_is\_inv\ y\ x$
$:= Commutative\_Ring.sum\_inv\_sym\ commutative\_ring.$

Proves that an element's inverse is unique. `Definition` *sum_inv_uniq*
$: \forall\ x\ y\ z\ :\ E,\ sum\_is\_inv\ x\ y \rightarrow sum\_is\_inv\ x\ z \rightarrow z = y$
$:= Commutative\_Ring.sum\_inv\_uniq\ commutative\_ring.$

Proves that every element has an inverse. `Definition` *sum_inv_ex*
$: \forall\ x\ :\ E,\ \exists\ y\ :\ E,\ sum\_is\_inv\ x\ y$
$:= Commutative\_Ring.sum\_inv\_ex\ commutative\_ring.$

Proves explicitly that every element has a unique inverse. `Definition` *sum_inv_uniq_ex*
$: \forall\ x\ :\ E,\ \exists!\ y\ :\ E,\ sum\_is\_inv\ x\ y$
$:= Commutative\_Ring.sum\_inv\_uniq\_ex\ commutative\_ring.$

Proves the left introduction rule. `Definition` *sum_intro_l*
$: \forall\ x\ y\ z\ :\ E,\ x = y \rightarrow z + x = z + y$
$:= Commutative\_Ring.sum\_intro\_l\ commutative\_ring.$

Proves the right introduction rule. `Definition` *sum_intro_r*
$: \forall\ x\ y\ z\ :\ E,\ x = y \rightarrow x + z = y + z$
$:= Commutative\_Ring.sum\_intro\_r\ commutative\_ring.$

Proves the left cancellation rule. `Definition` *sum_cancel_l*
$: \forall\ x\ y\ z\ :\ E,\ z + x = z + y \rightarrow x = y$
$:= Commutative\_Ring.sum\_cancel\_l\ commutative\_ring.$

Proves the right cancellation rule. `Definition` *sum_cancel_r*
$: \forall\ x\ y\ z\ :\ E,\ x + z = y + z \rightarrow x = y$
$:= Commutative\_Ring.sum\_cancel\_r\ commutative\_ring.$

Proves that an element's left inverse is unique. `Definition` *sum_inv_l_uniq*
$: \forall\ x\ y\ z\ :\ E,\ sum\_is\_inv\_l\ x\ y \rightarrow sum\_is\_inv\_l\ x\ z \rightarrow z = y$
$:= Commutative\_Ring.sum\_inv\_l\_uniq\ commutative\_ring.$

Proves that an element's right inverse is unique. `Definition` *sum_inv_r_uniq*
$: \forall\ x\ y\ z\ :\ E,\ sum\_is\_inv\_r\ x\ y \rightarrow sum\_is\_inv\_r\ x\ z \rightarrow z = y$
$:= Commutative\_Ring.sum\_inv\_r\_uniq\ commutative\_ring.$

Represents strongly-specified negation. `Definition` *sum_neg_strong*
$: \forall\ x\ :\ E,\ \{\ y\ |\ sum\_is\_inv\ x\ y\ \}$
$:= Commutative\_Ring.sum\_neg\_strong\ commutative\_ring.$

Represents negation. `Definition` *sum_neg*
$:\ E \rightarrow E$
$:= Commutative\_Ring.sum\_neg\ commutative\_ring.$

`Notation "{-}"` $:= (sum\_neg) : field\_scope.$

Notation "- x" := ($sum\_neg\ x$) : *field_scope*.

Asserts that the negation returns the inverse of its argument. Definition *sum_neg_def*
: $\forall\ x : E$, *sum_is_inv* x (- x)
:= *Commutative_Ring.sum_neg_def commutative_ring*.

Proves that negation is one-to-one Definition *sum_neg_inj*
: *is_injective* $E\ E$ {-}
:= *Commutative_Ring.sum_neg_inj commutative_ring*.

Proves the cancellation property for negation. Definition *sum_cancel_neg*
: $\forall\ x : E$, {-} (- x) = x
:= *Commutative_Ring.sum_cancel_neg commutative_ring*.

Proves that negation is onto Definition *sum_neg_onto*
: *is_onto* $E\ E$ {-}
:= *Commutative_Ring.sum_neg_onto commutative_ring*.

Proves that negation is surjective Definition *sum_neg_bijective*
: *is_bijective* $E\ E$ {-}
:= *Commutative_Ring.sum_neg_bijective commutative_ring*.

Proves that 1 is the right identity element. Definition *prod_id_r*
: *prod_is_id_r* 1
:= *Commutative_Ring.prod_id_r commutative_ring*.

Accepts one element, x, and asserts that x is the identity element. Definition *prod_id*
: *prod_is_id* 1
:= *Commutative_Ring.prod_id commutative_ring*.

Proves that the left identity element is unique. Definition *prod_id_l_uniq*
: $\forall\ x : E$, (*Monoid.is_id_l* E {#} x) $\rightarrow x = 1$
:= *Commutative_Ring.prod_id_l_uniq commutative_ring*.

Proves that the right identity element is unique. Definition *prod_id_r_uniq*
: $\forall\ x : E$, (*Monoid.is_id_r* E {#} x) $\rightarrow x = 1$
:= *Commutative_Ring.prod_id_r_uniq commutative_ring*.

Proves that the right identity element is unique. Definition *prod_id_uniq*
: $\forall\ x : E$, (*Monoid.is_id* E {#} x) $\rightarrow x = 1$
:= *Commutative_Ring.prod_id_uniq commutative_ring*.

Proves the left introduction rule. Definition *prod_intro_l*
: $\forall\ x\ y\ z : E$, $x = y \rightarrow z \# x = z \# y$
:= *Commutative_Ring.prod_intro_l commutative_ring*.

Proves the right introduction rule. Definition *prod_intro_r*
: $\forall\ x\ y\ z : E$, $x = y \rightarrow x \# z = y \# z$
:= *Commutative_Ring.prod_intro_r commutative_ring*.

Accepts two elements, x and y, and asserts that y is x's left inverse. Definition *prod_is_inv_l* := *Commutative_Ring.prod_is_inv_l commutative_ring*.

Accepts two elements, x and y, and asserts that y is x's right inverse. `Definition` *prod_is_inv_r* := *Commutative_Ring.prod_is_inv_r commutative_ring.*

Accepts two elements, x and y, and asserts that y is x's inverse. `Definition` *prod_is_inv* := *Commutative_Ring.prod_is_inv commutative_ring.*

Accepts one argument, x, and asserts that x has a left inverse. `Definition` *prod_has_inv_l* := *Commutative_Ring.prod_has_inv_l commutative_ring.*

Accepts one argument, x, and asserts that x has a right inverse. `Definition` *prod_has_inv_r* := *Commutative_Ring.prod_has_inv_r commutative_ring.*

Accepts one argument, x, and asserts that x has an inverse. `Definition` *prod_has_inv* := *Commutative_Ring.prod_has_inv commutative_ring.*

Proves that every left inverse must also be a right inverse. `Definition` *prod_is_inv_lr* := *Commutative_Ring.prod_is_inv_lr commutative_ring.*

Proves that every non-zero element has a right multiplicative inverse. `Definition` *prod_inv_r_ex*

```
  : ∀ x : E, x ≠ 0 → prod_has_inv_r x
  := fun x H
       ⇒ ex_ind
           (fun y H0
              ⇒ ex_intro (prod_is_inv_r x) y
                   (prod_is_inv_lr x y H0))
           (prod_inv_l_ex x H).
```

Proves that every non-zero element has a multiplicative inverse. `Definition` *prod_inv_ex*

```
  : ∀ x : E, nonzero x → prod_has_inv x
  := fun x H
       ⇒ ex_ind
           (fun y H0
              ⇒ ex_intro (prod_is_inv x) y
                   (conj H0
                      (prod_is_inv_lr x y H0)))
           (prod_inv_l_ex x H).
```

Proves that the left and right inverses of an element must be equal. `Definition` *prod_inv_l_r_eq*

```
  : ∀ x y : E, prod_is_inv_l x y → ∀ z : E, prod_is_inv_r x z → y = z
  := Commutative_Ring.prod_inv_l_r_eq commutative_ring.
```

Proves that the inverse relationship is symmetric. `Definition` *prod_inv_sym*

```
  : ∀ x y : E, prod_is_inv x y ↔ prod_is_inv y x
  := Commutative_Ring.prod_inv_sym commutative_ring.
```

Proves the left cancellation law for elements possessing a left inverse. `Definition` *prod_cancel_l*

$: \forall\ x\ y\ z : E,\ nonzero\ z \rightarrow z \ \#\ x = z \ \#\ y \rightarrow x = y$
$:=$ `fun` $x\ y\ z\ H$
$\Rightarrow$ *Commutative_Ring.prod_cancel_l commutative_ring x y z (prod_inv_l_ex z H).*

Proves the right cancellation law for elements possessing a right inverse. `Definition` *prod_cancel_r*
$: \forall\ x\ y\ z : E,\ nonzero\ z \rightarrow x \ \#\ z = y \ \#\ z \rightarrow x = y$
$:=$ `fun` $x\ y\ z\ H$
$\Rightarrow$ *Commutative_Ring.prod_cancel_r commutative_ring x y z (prod_inv_r_ex z H).*

Proves that an element's left inverse is unique. `Definition` *prod_inv_l_uniq*
$: \forall\ x : E,\ nonzero\ x \rightarrow \forall\ y\ z : E,\ prod\_is\_inv\_l\ x\ y \rightarrow prod\_is\_inv\_l\ x\ z \rightarrow z = y$
$:=$ `fun` $x\ H$
$\Rightarrow$ *Commutative_Ring.prod_inv_l_uniq commutative_ring x (prod_inv_r_ex x H).*

Proves that an element's right inverse is unique. `Definition` *prod_inv_r_uniq*
$: \forall\ x : E,\ nonzero\ x \rightarrow \forall\ y\ z : E,\ prod\_is\_inv\_r\ x\ y \rightarrow prod\_is\_inv\_r\ x\ z \rightarrow z = y$
$:=$ `fun` $x\ H$
$\Rightarrow$ *Commutative_Ring.prod_inv_r_uniq commutative_ring x (prod_inv_l_ex x H).*

Proves that an element's inverse is unique. `Definition` *prod_inv_uniq*
$: \forall\ x\ y\ z : E,\ prod\_is\_inv\ x\ y \rightarrow prod\_is\_inv\ x\ z \rightarrow z = y$
$:=$ *Commutative_Ring.prod_inv_uniq commutative_ring.*

Proves that every nonzero element has a unique inverse. `Definition` *prod_uniq_inv_ex*
$: \forall\ x : E,\ nonzero\ x \rightarrow \exists!\ y : E,\ prod\_is\_inv\ x\ y$
$:=$ `fun` $x\ H$
$\Rightarrow$ *ex_ind*
(`fun` y ($H0$: *prod_is_inv x y*)
$\Rightarrow$ *ex_intro*
(*unique* (*prod_is_inv x*))
y
(*conj H0* (`fun` $z\ H1 \Rightarrow$ *eq_sym* (*prod_inv_uniq x y z H0 H1*))))
(*prod_inv_ex x H*).

Proves that 1 is its own left multiplicative inverse. `Definition` *recipr_1_l*
: *prod_is_inv_l* 1 1
$:=$ *Commutative_Ring.recipr_1_l commutative_ring.*

Proves that 1 is its own right multiplicative inverse. `Definition` *recipr_1_r*
: *prod_is_inv_r* 1 1
$:=$ *Commutative_Ring.recipr_1_r commutative_ring.*

Proves that 1 is its own recriprical. `Definition` *recipr_1*
: *prod_is_inv* 1 1
$:=$ *Commutative_Ring.recipr_1 commutative_ring.*

Proves that 1 has a left multiplicative inverse. `Definition` *prod_has_inv_l_1*
: *prod_has_inv_l* 1

$:=$ *Commutative_Ring.prod_has_inv_l_1 commutative_ring.*

Proves that 1 has a right multiplicative inverse. `Definition` *prod_has_inv_r_1*
: *prod_has_inv_r* 1
$:=$ *Commutative_Ring.prod_has_inv_r_1 commutative_ring.*

Proves that 1 has a reciprical `Definition` *prod_has_inv_1*
: *prod_has_inv* 1
$:=$ *Commutative_Ring.prod_has_inv_1 commutative_ring.*

Proves that multiplication is right distributive over addition. `Definition` *prod_sum_distrib_r*
: *Ring.is_distrib_r E* {#} {+}
$:=$ *Commutative_Ring.prod_sum_distrib_r commutative_ring.*

Asserts that multiplication is distributive over addition. `Definition` *prod_sum_distrib*
: *Ring.is_distrib E* {#} {+}
$:=$ *Commutative_Ring.prod_sum_distrib commutative_ring.*

Proves that 0 times every number equals 0.
0 x = 0 x (0 + 0) x = 0 x 0 x + 0 x = 0 x 0 x = 0 `Definition` *prod_0_l*
: $\forall\ x : E$, 0 # $x = 0$
$:=$ *Commutative_Ring.prod_0_l commutative_ring.*

Proves that 0 times every number equals 0. `Definition` *prod_0_r*
: $\forall\ x : E$, x # $0 = 0$
$:=$ *Commutative_Ring.prod_0_r commutative_ring.*

Proves that 0 does not have a left multiplicative inverse. `Definition` *prod_0_inv_l*
: $\neg$ *prod_has_inv_l* 0
$:=$ *Commutative_Ring.prod_0_inv_l commutative_ring.*

Proves that 0 does not have a right multiplicative inverse. `Definition` *prod_0_inv_r*
: $\neg$ *prod_has_inv_r* 0
$:=$ *Commutative_Ring.prod_0_inv_r commutative_ring.*

Proves that 0 does not have a multiplicative inverse - I.E. 0 does not have a reciprocal. `Definition` *prod_0_inv*
: $\neg$ *prod_has_inv* 0
$:=$ *Commutative_Ring.prod_0_inv commutative_ring.*

Proves that multiplicative inverses, when they exist are always nonzero. `Definition` *prod_inv_0*
: $\forall\ x\ y : E$, *prod_is_inv* $x\ y \rightarrow$ *nonzero* y
$:=$ *Commutative_Ring.prod_inv_0 commutative_ring.*

Proves that the product of two non-zero values is non-zero.
x * y <> 0 x * y = 0 -> False
assume x * y = 0 1/x * x * y = 1/x * 0 y = 0 which is a contradiction. `Definition` *prod_nonzero_closed*
: $\forall\ x : E$, *nonzero* $x \rightarrow \forall\ y : E$, *nonzero* $y \rightarrow$ *nonzero* (x # y)

$:=$ `fun` x H y $H0$ $(H1 : x \# y = 0)$
$\Rightarrow$ ex_ind
(`fun` z $(H2 : prod\_is\_inv\_l\ x\ z)$
$\Rightarrow$ $H0$ $(prod\_intro\_l\ (x \# y)\ 0\ z\ H1$
$||\ z \# (x \# y) = a$ @a `by` $\leftarrow$ $prod\_0\_r\ z$
$||\ a = 0$ @a `by` $\leftarrow$ $prod\_is\_assoc\ z\ x\ y$
$||\ a \# y = 0$ @a `by` $\leftarrow$ $H2$
$||\ a = 0$ @a `by` $\leftarrow$ $prod\_id\_l\ y))$
$(prod\_inv\_l\_ex\ x\ H)$.

Represents -1 and proves that it exists. `Definition` E_n1_strong
: { $x : E$ | $sum\_is\_inv\ 1\ x$ }
$:=$ $Commutative\_Ring.E\_n1\_strong\ commutative\_ring$.

Represents -1. `Definition` $E\_n1 : E :=$ $Commutative\_Ring.E\_n1\ commutative\_ring$.

Defines a symbolic representation for -1

Note: here we represent the inverse of 1 rather than the negation of 1. Letter we prove that the negation equals the inverse.

Note: brackets are needed to ensure Coq parses the symbol as a single token instead of a prefixed function call. `Notation "{-1}"` $:= E\_n1 : field\_scope$.

Asserts that -1 is the additive inverse of 1. `Definition` E_n1_def
: sum_is_inv 1 {-1}
$:=$ $Commutative\_Ring.E\_n1\_def\ commutative\_ring$.

Asserts that -1 is the left inverse of 1. `Definition` $E\_n1\_inv\_l$
: $sum\_is\_inv\_l$ 1 {-1}
$:=$ $Commutative\_Ring.E\_n1\_inv\_l\ commutative\_ring$.

Asserts that -1 is the right inverse of 1. `Definition` $E\_n1\_inv\_r$
: $sum\_is\_inv\_r$ 1 {-1}
$:=$ $Commutative\_Ring.E\_n1\_inv\_r\ commutative\_ring$.

Asserts that every additive inverse of 1 must be equal to -1. `Definition` E_n1_uniq
: $\forall\ x : E$, $sum\_is\_inv\ 1\ x \rightarrow x =$ {-1}
$:=$ $Commutative\_Ring.E\_n1\_uniq\ commutative\_ring$.

Proves that -1 * x equals the multiplicative inverse of x.

- 1 x + x = 0
- 1 x + 1 x = 0

(-1 + 1) x = 0 0 x = 0 0 = 0 `Definition` $prod\_n1\_x\_inv\_l$
: $\forall\ x : E$, $sum\_is\_inv\_l\ x$ ({-1} $\#\ x)$
$:=$ $Commutative\_Ring.prod\_n1\_x\_inv\_l\ commutative\_ring$.

Proves that x * -1 equals the multiplicative inverse of x.

x -1 + x = 0 `Definition` $prod\_x\_n1\_inv\_l$

: $\forall$ *x* : *E*, *sum_is_inv_l x* (*x* # {-1})
:= *Commutative_Ring.prod_x_n1_inv_l commutative_ring.*

Proves that x + -1 x = 0. `Definition` *prod_n1_x_inv_r*
: $\forall$ *x* : *E*, *sum_is_inv_r x* ({-1} # *x*)
:= *Commutative_Ring.prod_n1_x_inv_r commutative_ring.*

Proves that x + x -1 = 0. `Definition` *prod_x_n1_inv_r*
: $\forall$ *x* : *E*, *sum_is_inv_r x* (*x* # {-1})
:= *Commutative_Ring.prod_x_n1_inv_r commutative_ring.*

Proves that -1 x is the additive inverse of x. `Definition` *prod_n1_x_inv*
: $\forall$ *x* : *E*, *sum_is_inv x* ({-1} # *x*)
:= *Commutative_Ring.prod_n1_x_inv commutative_ring.*

Proves that x -1 is the additive inverse of x. `Definition` *prod_x_n1_inv*
: $\forall$ *x* : *E*, *sum_is_inv x* (*x* # {-1})
:= *Commutative_Ring.prod_x_n1_inv commutative_ring.*

Proves that multiplying by -1 is equivalent to negation. `Definition` *prod_n1_neg*
: {#} {-1} = {-}
:= *Commutative_Ring.prod_n1_neg commutative_ring.*

Accepts one element, x, and proves that x -1 equals the additive negation of x. `Definition` *prod_x_n1_neg*
: $\forall$ *x* : *E*, *x* # {-1} = - *x*
:= *Commutative_Ring.prod_x_n1_neg commutative_ring.*

Accepts one element, x, and proves that

- 1 x equals the additive negation of x.

`Definition` *prod_n1_x_neg*
: $\forall$ *x* : *E*, {-1} # *x* = - *x*
:= *Commutative_Ring.prod_n1_x_neg commutative_ring.*

Proves that -1 x = x -1. `Definition` *prod_n1_eq*
: $\forall$ *x* : *E*, {-1} # *x* = *x* # {-1}
:= *Commutative_Ring.prod_n1_eq commutative_ring.*

Proves that the additive negation of 1 equals -1. `Definition` *neg_1*
: {-} 1 = {-1}
:= *Commutative_Ring.neg_1 commutative_ring.*

Proves that the additive negation of -1 equals 1. `Definition` *neg_n1*
: - {-1} = 1
:= *Commutative_Ring.neg_n1 commutative_ring.*

Proves that -1 * -1 = 1.

- 1 * -1 = -1 * -1

- 1 * -1 = prod -1 -1
- 1 * -1 = {-} -1
- 1 * -1 = 1

`Definition` *prod_n1_n1*
: {-1} # {-1} = 1
:= *Commutative_Ring.prod_n1_n1 commutative_ring*.

Proves that -1 is its own multiplicative inverse. `Definition` *E_n1_inv*
: *prod_is_inv* {-1} {-1}
:= *Commutative_Ring.E_n1_inv commutative_ring*.

Proves that -1 is nonzero. `Definition` *nonzero_n1*
: *nonzero* {-1}
:= `fun` *H* : {-1} = 0
⇒ *distinct_1_0*
(*prod_intro_l* {-1} 0 {-1} *H*
|| *a* = {-1} # 0 @*a* `by` ← *prod_n1_n1*
|| 1 = *a* @*a* `by` ← *prod_0_r* {-1}).

Represents the reciprical operation. `Definition` *recipr_strong*
: $\forall$ *x* : *E*, *nonzero x* → {*y* | *prod_is_inv x y*}
:= `fun` *x H*
⇒ *constructive_definite_description* (*prod_is_inv x*)
(*prod_uniq_inv_ex x H*).

Represents the reciprical operation. `Definition` *recipr*
: $\forall$ *x* : *E*, *nonzero x* → *E*
:= `fun` *x H*
⇒ *proj1_sig* (*recipr_strong x H*).

`Notation` "{1/ x }" := (*recipr x*) : *field_scope*.

Proves that the reciprical operation correctly returns the inverse of the given element. `Definition` *recipr_def*
: $\forall$ (*x* : *E*) (*H* : *nonzero x*), *prod_is_inv x* ({1/*x*} *H*)
:= `fun` *x H*
⇒ *proj2_sig* (*recipr_strong x H*).

Proves that (1/-1) = -1. `Definition` *recipr_n1*
: ({1/{-1}} *nonzero_n1*) = {-1}
:= *prod_inv_uniq* {-1} {-1} ({1/{-1}} *nonzero_n1*)
E_n1_inv
(*recipr_def* {-1} *nonzero_n1*).

Proves that recipricals are nonzero. `Definition` *recipr_nonzero*
: $\forall$ (*x* : *E*) (*H* : *nonzero x*), *nonzero* ({1/*x*} *H*)

$:=$ `fun` x H

$\Rightarrow$ *prod_inv_0* x ($\{1/x\}$ H) (*recipr_def* x H).

Proves that 1/(1/x) = x `Definition` *recipr_cancel*

: $\forall$ (x : E) (H : *nonzero* x), ($\{1/(\{1/x\}$ $H)\}$ (*recipr_nonzero* x H)) = x

$:=$ `fun` x H

$\Rightarrow$ *Monoid.op_cancel_neg_gen prod_monoid* x

(*prod_inv_ex* x H)

(*prod_inv_ex* ($\{1/x\}$ H) (*recipr_nonzero* x H)).

Represents division. `Definition` *div*

: $E \rightarrow \forall$ x : E, *nonzero* $x \rightarrow E$

$:=$ `fun` x y H

$\Rightarrow$ x # ($\{1/y\}$ H).

`Notation` "x / y" := (*div* x y) : *field_scope*.

Proves that x y/x = y. `Definition` *div_cancel_l*

: $\forall$ (x : E) (H : *nonzero* x) (y : E), x # ((y/x) H) = y

$:=$ `fun` x H y

$\Rightarrow$ *eq_refl* (x # ((y/x) H))

|| x # ((y/x) H) = x # a @a `by` $\leftarrow$ *prod_is_comm* y ($\{1/x\}$ H)

|| x # ((y/x) H) = a @a `by` $\leftarrow$ *prod_is_assoc* x ($\{1/x\}$ H) y

|| x # ((y/x) H) = a # y @a `by` $\leftarrow$ *proj2* (*recipr_def* x H)

|| x # ((y/x) H) = a @a `by` $\leftarrow$ *prod_id_l* y.

Proves that x/y y = x. `Definition` *div_cancel_r*

: $\forall$ (x : E) (H : *nonzero* x) (y : E), ((y/x) H) # x = y

$:=$ `fun` x H y

$\Rightarrow$ *div_cancel_l* x H y

|| a = y @a `by` $\leftarrow$ *prod_is_comm* x ((y/x) H).

The following section proves that the set of nonzero elements forms an algebraic group over multiplication with 1 as the identity.

To show this, we map every nonzero field element, x, onto a dependent product, (x, H), where H represents a proof that x is nonzero.

We then define equality over these products such that two pair, (x, H) and (y, H0), are equal whenever x and y are.

Continuing, we define multiplication reasonably so that (x, H) denotes multiplication over pairs.

With these definitions in hand, we show that the resulting elements form a group and that this group is isomorphic with the set of nonzero field elements.

Represents those field elements that are nonzero.

Note: each value can be seen intuitively as a pair, (x, H), where x is a monoid element and H is a proof that x is invertable. `Definition` D : `Set` := $\{x : E \mid$ *nonzero* $x\}$.

Accepts a field element and a proof that it is nonzero and returns its projection in D. `Definition` *D_cons*

: $\forall$ *x* : *E*, *nonzero x* $\rightarrow$ *D*
:= *exist nonzero*.

Asserts that any two equal non-zero elements, x and y, are equivalent (using dependent equality).

Note: to compare sig elements that differ only in their proof terms, such as (x, H) and (x, H0), we must introduce a new notion of equality called "dependent equality". This relationship is defined in the Eqdep module. `Axiom` *D_eq_dep*
: $\forall$ (*x* : *E*) (*H* : *nonzero x*) (*y* : *E*) (*H0* : *nonzero y*), *y* = *x* $\rightarrow$ *eq_dep E nonzero y H0 x H*.

Given that two invertable monoid elements x and y are equal (using dependent equality), this lemma proves that their projections into D are equal.

Note: this proof is equivalent to:

eq_dep_eq_sig E (Monoid.has_inv m) y x H0 H (D_eq_dep x H y H0 H1).

The definition for eq_dep_eq_sig has been expanded however for compatability with Coq v8.4. `Definition` *D_eq*
: $\forall$ (*x* : *E*) (*H* : *nonzero x*) (*y* : *E*) (*H0* : *nonzero y*), *y* = *x* $\rightarrow$ *D_cons y H0* = *D_cons x H*
:= `fun` *x H y H0 H1*
  $\Rightarrow$ *eq_dep_ind E nonzero y H0*
    (`fun` (*z* : *E*) (*H2* : *nonzero z*)
      $\Rightarrow$ *D_cons y H0* = *D_cons z H2*)
    (*eq_refl* (*D_cons y H0*)) *x H* (*D_eq_dep x H y H0 H1*).

Represents the group identity element. `Definition` *D_1* := *D_cons* 1 *distinct_1_0*.

Represents the group operation.

Note: intuitively this function accepts two invertable monoid elements, (x, H) and (y, H0), and returns (x + y, H1), where H, H0, and H1 are generalized invertability proofs. `Definition` *D_prod*
: *D* $\rightarrow$ *D* $\rightarrow$ *D*
:= *sig_rec*
  (`fun` _ $\Rightarrow$ *D* $\rightarrow$ *D*)
  (`fun` (*u* : *E*) (*H* : *nonzero u*)
    $\Rightarrow$ *sig_rec*
      (`fun` _ $\Rightarrow$ *D*)
      (`fun` (*v* : *E*) (*H0* : *nonzero v*)
        $\Rightarrow$ *D_cons*
          (*u* # *v*)
          (*prod_nonzero_closed u H v H0*))).

TODO

Proves that D and D_prod are isomorphic with the set of nonzero field elements. Definition D_iso

Accepts a group element, x, and asserts that x is a left identity element. `Definition` *D_prod_is_id_l* := *Monoid.is_id_l D D_prod.*

Accepts a group element, x, and asserts that x is a right identity element. `Definition` *D_prod_is_id_r* := *Monoid.is_id_r D D_prod.*

Accepts a group element, x, and asserts that x is an/the identity element. `Definition` *D_prod_is_id* := *Monoid.is_id D D_prod.*

Proves that D_1 is a left identity element. `Definition` *D_prod_id_l*

```
  : D_prod_is_id_l D_1
  := sig_ind
       (fun x ⇒ D_prod D_1 x = x)
       (fun (u : E) (H : nonzero u)
          ⇒ D_eq u H (1 # u) (prod_nonzero_closed 1 distinct_1_0 u H)
               (prod_id_l u)).
```

Proves that D_1 is a right identity element. `Definition` *D_prod_id_r*

```
  : D_prod_is_id_r D_1
  := sig_ind
       (fun x ⇒ D_prod x D_1 = x)
       (fun (u : E) (H : nonzero u)
          ⇒ D_eq u H (u # 1) (prod_nonzero_closed u H 1 distinct_1_0)
               (prod_id_r u)).
```

Proves that D_1 is the identity element. `Definition` *D_prod_id*

```
  : D_prod_is_id D_1
  := conj D_prod_id_l D_prod_id_r.
```

Proves that the group operation is associative. `Definition` *D_prod_assoc*

```
  : Monoid.is_assoc D D_prod
  := sig_ind
       (fun x ⇒ ∀ y z : D, D_prod x (D_prod y z) = D_prod (D_prod x y) z)
       (fun (u : E) (H : nonzero u)
         ⇒ sig_ind
              (fun y ⇒ ∀ z : D, D_prod (D_cons u H) (D_prod y z) = D_prod (D_prod
(D_cons u H) y) z)
              (fun (v : E) (H0 : nonzero v)
                ⇒ sig_ind
                     (fun z ⇒ D_prod (D_cons u H) (D_prod (D_cons v H0) z) =
D_prod (D_prod (D_cons u H) (D_cons v H0)) z)
                     (fun (w : E) (H1 : nonzero w)
                       ⇒ let a
                            : E
                            := u # (v # w) in
                          let H2
                            : nonzero a
```

```
                    := prod_nonzero_closed u H (v # w) (prod_nonzero_closed
v H0 w H1) in
                  let b
                    : E
                    := prod (u # v) w in
                  let H3
                    : nonzero b
                    := prod_nonzero_closed (u # v) (prod_nonzero_closed u H
v H0) w H1 in
                  let X
                    : D
                    := D_cons a H2 in
                  let Y
                    : D
                    := D_cons b H3 in
                  D_eq b H3 a H2
                    (prod_is_assoc u v w)
          ))).
```

Accepts two values, x and y, and asserts that y is a left inverse of x. `Definition` *D_prod_is_inv_l* := *Monoid.is_inv_l D D_prod D_1 D_prod_id.*

Accepts two values, x and y, and asserts that y is a right inverse of x. `Definition` *D_prod_is_inv_r* := *Monoid.is_inv_r D D_prod D_1 D_prod_id.*

Accepts two values, x and y, and asserts that y is an inverse of x. `Definition` *D_prod_is_inv* := *Monoid.is_inv D D_prod D_1 D_prod_id.*

Accepts two nonzero elements, x and y, where y is a left inverse of x and proves that y's projection into D is the left inverse of x's. `Definition` *D_prod_inv_l*

```
  : ∀ (u : E) (H : nonzero u) (v : E) (H0 : nonzero v),
      prod_is_inv_l u v →
      D_prod_is_inv_l (D_cons u H) (D_cons v H0)
  := fun (u : E) (H : nonzero u) (v : E) (H0 : nonzero v)
      ⇒ D_eq 1 distinct_1_0 (v # u) (prod_nonzero_closed v H0 u H).
```

Accepts two invertable monoid elements, x and y, where y is a right inverse of x and proves that y's projection into D is the right inverse of x's. `Definition` *D_prod_inv_r*

```
  : ∀ (u : E) (H : nonzero u) (v : E) (H0 : nonzero v),
      prod_is_inv_r u v →
      D_prod_is_inv_r (D_cons u H) (D_cons v H0)
  := fun (u : E) (H : nonzero u) (v : E) (H0 : nonzero v)
      ⇒ D_eq 1 distinct_1_0 (u # v) (prod_nonzero_closed u H v H0).
```

Accepts two invertable monoid elements, x and y, where y is the inverse of x and proves that y's projection into D is the inverse of x's. `Definition` *D_prod_inv*

```
  : ∀ (u : E) (H : nonzero u) (v : E) (H0 : nonzero v),
```

```
      prod_is_inv u v →
      D_prod_is_inv (D_cons u H) (D_cons v H0)
  := fun (u : E) (H : nonzero u) (v : E) (H0 : nonzero v) (H1 : prod_is_inv u v)
      ⇒ conj (D_prod_inv_l u H v H0 (proj1 H1))
              (D_prod_inv_r u H v H0 (proj2 H1)).
```

Accepts a nonzero element and returns its inverse, y, along with a proof that y is x's inverse. `Definition` *D_prod_neg_strong*

```
  : ∀ x : D, { y : D | D_prod_is_inv x y }
  := sig_rec
      (fun x ⇒ { y : D | D_prod_is_inv x y })
      (fun (u : E) (H : nonzero u)
        ⇒ let v
              : E
              := Monoid.op_neg prod_monoid u (prod_inv_ex u H) in
           let H0
              : prod_is_inv u v
              := Monoid.op_neg_def prod_monoid u (prod_inv_ex u H) in
           let H1
              : nonzero v
              := prod_inv_0 u v H0 in
           exist
              (fun y : D ⇒ D_prod_is_inv (D_cons u H) y)
              (D_cons v H1)
              (D_prod_inv u H v H1 H0)).
```

Proves that every group element has an inverse. `Definition` *D_prod_inv_ex*

```
  : ∀ x : D, ∃ y : D, D_prod_is_inv x y
  := fun x
      ⇒ let (y, H) := D_prod_neg_strong x in
          ex_intro
            (fun y ⇒ D_prod_is_inv x y)
            y H.
```

Proves that every group element has a left inverse. `Definition` *D_prod_inv_l_ex*

```
  : ∀ x : D, ∃ y : D, D_prod_is_inv_l x y
  := fun x
      ⇒ ex_ind
            (fun y (H : D_prod_is_inv x y)
              ⇒ ex_intro (fun z ⇒ D_prod_is_inv_l x z) y (proj1 H))
            (D_prod_inv_ex x).
```

Proves that every group element has a right inverse. `Definition` *D_prod_inv_r_ex*

```
  : ∀ x : D, ∃ y : D, D_prod_is_inv_r x y
  := fun x
```

```
    ⇒ ex_ind
        (fun y (H : D_prod_is_inv x y)
          ⇒ ex_intro (fun z ⇒ D_prod_is_inv_r x z) y (proj2 H))
        (D_prod_inv_ex x).
```

Proves that the set of nonzero elements form a group over multiplication.

```
Definition
nonzero_group := Group.group D D_1 D_prod D_prod_assoc
  D_prod_id_l D_prod_id_r D_prod_inv_l_ex
  D_prod_inv_r_ex.

End Theorems.

End Field.
```

Chapter 6

Library functional-algebra.group

This module defines the Group record type which can be used to represent algebraic groups and provides a collection of theorems and axioms describing them.

`Require Import` *ProofIrrelevance*.
`Require Import` *Description*.
`Require Import` *base*.
`Require Import` *function*.
`Require Import` *monoid*.

`Module` *Group*.

Represents algebraic groups. `Structure` *Group* : `Type` := *group* {

Represents the set of group elements. *E*: `Set`;

Represents the identity element. *E_0*: *E*;

Represents the group operation. *op*: $E \to E \to E$;

Asserts that the group operator is associative. *op_is_assoc* : *Monoid.is_assoc E op*;

Asserts that E_0 is the left identity element. *op_id_l* : *Monoid.is_id_l E op E_0*;

Asserts that E_0 is the right identity element. *op_id_r* : *Monoid.is_id_r E op E_0*;

Asserts that every group element has a left inverse. *op_inv_l_ex* : ∀ *x* : *E*, ∃ *y* : *E*, *Monoid.is_inv_l E op E_0* (*conj op_id_l op_id_r*) *x y*;

Asserts that every group element has a right inverse. *op_inv_r_ex* : ∀ *x* : *E*, ∃ *y* : *E*, *Monoid.is_inv_r E op E_0* (*conj op_id_l op_id_r*) *x y*
}.

Enable implicit arguments for group properties.

Arguments E_0 {g}.

Arguments op {g} x y.

Arguments op_is_assoc {g} x y z.

Arguments op_id_l {g} x.

Arguments op_id_r {g} x.

Arguments op_inv_l_ex {g} x.

Arguments op_inv_r_ex {g} x.

Define notations for group properties.

`Notation` "0" := *E_0* : *group_scope.*

`Notation` "x + y" := (*op x y*) (`at level` 50, `left associativity`) : *group_scope.*

`Notation` "{+}" := *op* : *group_scope.*

`Open Scope` *group_scope.*

`Section` *Theorems.*

Represents an arbitrary group.

Note: we use Variable rather than Parameter to ensure that the following theorems are generalized w.r.t g. `Variable` *g* : *Group.*

Represents the set of group elements. `Let` *E* := *E g.*

Represents the monoid structure formed by op over E. `Definition` *op_monoid* := *Monoid.monoid E* 0 {+} *op_is_assoc op_id_l op_id_r.*

Accepts one group element, x, and asserts that x is the left identity element. `Definition` *op_is_id_l* := *Monoid.op_is_id_l op_monoid.*

Accepts one group element, x, and asserts that x is the right identity element. `Definition` *op_is_id_r* := *Monoid.op_is_id_r op_monoid.*

Accepts one group element, x, and asserts that x is the identity element. `Definition` *op_is_id* := *Monoid.op_is_id op_monoid.*

Proves that 0 is the identity element. `Definition` *op_id* := *Monoid.op_id op_monoid.*

Accepts two group elements, x and y, and asserts that y is x's left inverse. `Definition` *op_is_inv_l* := *Monoid.op_is_inv_l op_monoid.*

Accepts two group elements, x and y, and asserts that y is x's right inverse. `Definition` *op_is_inv_r* := *Monoid.op_is_inv_r op_monoid.*

Proves that the left identity element is unique. `Definition` *op_id_l_uniq*
: $\forall$ *x* : *E*, (*op_is_id_l x*) $\rightarrow$ *x* = 0
:= *Monoid.op_id_l_uniq op_monoid.*

Proves that the right identity element is unique. `Definition` *op_id_r_uniq*
: $\forall$ *x* : *E*, (*op_is_id_r x*) $\rightarrow$ *x* = 0

$:= \mathit{Monoid.op\_id\_r\_uniq}\ \mathit{op\_monoid}.$

Proves that the identity element is unique. `Definition` $\mathit{op\_id\_uniq}$
$: \forall\ x : E, (\mathit{op\_is\_id}\ x) \rightarrow x = 0$
$:= \mathit{Monoid.op\_id\_uniq}\ \mathit{op\_monoid}.$

Proves the left introduction rule. `Definition` $\mathit{op\_intro\_l}$
$: \forall\ x\ y\ z : E, x = y \rightarrow z + x = z + y$
$:= \mathit{Monoid.op\_intro\_l}\ \mathit{op\_monoid}.$

Proves the right introduction rule. `Definition` $\mathit{op\_intro\_r}$
$: \forall\ x\ y\ z : E, x = y \rightarrow x + z = y + z$
$:= \mathit{Monoid.op\_intro\_r}\ \mathit{op\_monoid}.$

Accepts two group elements, x and y, and asserts that y is x's inverse. `Definition` $\mathit{op\_is\_inv} := \mathit{Monoid.op\_is\_inv}\ \mathit{op\_monoid}.$

Proves that for every group element, x, its left and right inverses are equal. `Definition` $\mathit{op\_inv\_l\_r\_eq}$
$: \forall\ x\ y : E, \mathit{op\_is\_inv\_l}\ x\ y \rightarrow \forall\ z : E, \mathit{op\_is\_inv\_r}\ x\ z \rightarrow y = z$
$:= \mathit{Monoid.op\_inv\_l\_r\_eq}\ \mathit{op\_monoid}.$

Proves that the inverse relation is symmetrical. `Definition` $\mathit{op\_inv\_sym}$
$: \forall\ x\ y : E, \mathit{op\_is\_inv}\ x\ y \leftrightarrow \mathit{op\_is\_inv}\ y\ x$
$:= \mathit{Monoid.op\_inv\_sym}\ \mathit{op\_monoid}.$

Proves that every group element has an inverse. `Definition` $\mathit{op\_inv\_ex}$
$: \forall\ x : E, \exists\ y : E, \mathit{op\_is\_inv}\ x\ y$
$:= \texttt{fun}\ x : E$
$\quad \Rightarrow \mathit{ex\_ind}$
$\qquad (\texttt{fun}\ y\ H$
$\qquad\ \ \Rightarrow \mathit{ex\_ind}$
$\qquad\qquad (\texttt{fun}\ z\ \mathit{H0}$
$\qquad\qquad\ \ \Rightarrow \texttt{let}\ \mathit{H1}$
$\qquad\qquad\qquad\quad : \mathit{op\_is\_inv\_r}\ x\ y$
$\qquad\qquad\qquad\quad := \mathit{H0}$
$\qquad\qquad\qquad\quad \|\ \mathit{op\_is\_inv\_r}\ x\ a\ @a$
$\qquad\qquad\qquad\qquad \texttt{by}\ \mathit{op\_inv\_l\_r\_eq}\ x\ y\ H\ z\ \mathit{H0}\ \texttt{in}$
$\qquad\qquad\qquad \mathit{ex\_intro}$
$\qquad\qquad\qquad\quad (\texttt{fun}\ a \Rightarrow \mathit{op\_is\_inv}\ x\ a)$
$\qquad\qquad\qquad\quad y$
$\qquad\qquad\qquad\quad (\mathit{conj}\ H\ \mathit{H1}))$
$\qquad\qquad (\mathit{op\_inv\_r\_ex}\ x))$
$\qquad (\mathit{op\_inv\_l\_ex}\ x).$

Proves the left cancellation rule. `Definition` $\mathit{op\_cancel\_l}$
$: \forall\ x\ y\ z : E, z + x = z + y \rightarrow x = y$
$:= \texttt{fun}\ x\ y\ z\ H$

$\Rightarrow$ *Monoid.op_cancel_l op_monoid x y z* (*op_inv_l_ex z*) *H*.

Proves the right cancellation rule. `Definition` *op_cancel_r*
: $\forall$ *x y z* : *E*, $x + z = y + z \rightarrow x = y$
:= `fun` *x y z*
$\Rightarrow$ *Monoid.op_cancel_r op_monoid x y z* (*op_inv_r_ex z*).

Proves that an element's left inverse is unique. `Definition` *op_inv_l_uniq*
: $\forall$ *x y z* : *E*, *op_is_inv_l x y* $\rightarrow$ *op_is_inv_l x z* $\rightarrow$ $z = y$
:= `fun` *x*
$\Rightarrow$ *Monoid.op_inv_l_uniq op_monoid x* (*op_inv_r_ex x*).

Proves that an element's right inverse is unique. `Definition` *op_inv_r_uniq*
: $\forall$ *x y z* : *E*, *op_is_inv_r x y* $\rightarrow$ *op_is_inv_r x z* $\rightarrow$ $z = y$
:= `fun` *x*
$\Rightarrow$ *Monoid.op_inv_r_uniq op_monoid x* (*op_inv_l_ex x*).

Proves that an element's inverse is unique. `Definition` *op_inv_uniq*
: $\forall$ *x y z* : *E*, *op_is_inv x y* $\rightarrow$ *op_is_inv x z* $\rightarrow$ $z = y$
:= *Monoid.op_inv_uniq op_monoid*.

Proves explicitly that every element has a unique inverse. `Definition` *op_inv_uniq_ex*
: $\forall$ *x* : *E*, $\exists!$ *y* : *E*, *op_is_inv x y*
:= `fun` *x*
$\Rightarrow$ *ex_ind*
(`fun` *y* (*H* : *op_is_inv x y*)
$\Rightarrow$ *ex_intro*
(`fun` *y* $\Rightarrow$ *op_is_inv x y* $\land$ $\forall$ *z*, *op_is_inv x z* $\rightarrow$ $y = z$)
y
(*conj H* (`fun` *z H0* $\Rightarrow$ *eq_sym* (*op_inv_uniq x y z H H0*))))
(*op_inv_ex x*).

Represents strongly-specified negation. `Definition` *op_neg_strong*
: $\forall$ *x* : *E*, { *y* | *op_is_inv x y* }
:= `fun` *x* $\Rightarrow$ *Monoid.op_neg_strong op_monoid x* (*op_inv_ex x*).

Represents negation. `Definition` *op_neg*
: $E \rightarrow E$
:= `fun` *x* $\Rightarrow$ *Monoid.op_neg op_monoid x* (*op_inv_ex x*).

`Notation "{-}"` := (*op_neg*) : *group_scope*.

Asserts that the negation returns the inverse of its argument `Definition` *op_neg_def*
: $\forall$ *x* : *E*, *op_is_inv x* ({-} *x*)
:= `fun` *x* $\Rightarrow$ *Monoid.op_neg_def op_monoid x* (*op_inv_ex x*).

Proves that negation is one-to-one 0 = 0 x + -x = 0 x + -x = y + -y x + -x = y + -x x = y `Definition` *op_neg_inj*
: *is_injective E E op_neg*
:= `fun` *x y*

$\Rightarrow$ *Monoid.op_neg_inj op_monoid x (op_inv_ex x) y (op_inv_ex y).*

Proves the cancellation property for negation. `Definition` *op_cancel_neg*
: $\forall\ x : E$, {-} ({-} $x) = x$
:= `fun` x
$\Rightarrow$ *Monoid.op_cancel_neg_gen op_monoid x (op_inv_ex x) (op_inv_ex* ({-} x)).

Proves that negation is surjective - onto `Definition` *op_neg_onto*
: *is_onto E E* {-}
:= `fun` $x \Rightarrow$ *ex_intro* (`fun` $y \Rightarrow$ {-} $y = x$) ({-} x) (*op_cancel_neg x*).

Proves that negation is bijective. `Definition` *op_neg_bijective*
: *is_bijective E E* {-}
:= *conj op_neg_inj op_neg_onto.*

Proves that neg x = y -> neg y = x `Definition` *op_neg_rev*
: $\forall\ x\ y : E$, {-} $x = y \rightarrow$ {-} $y = x$
:= `fun` $x\ y\ H$
$\Rightarrow$ *eq_sym*
(`f_equal` {-} H
|| $a =$ {-} y @a `by` $\leftarrow$ *op_cancel_neg x*).

`End` *Theorems.*

`End` *Group.*

`Notation` "{-}" := (*Group.op_neg* _) : *group_scope.*

Chapter 7

Library functional-algebra.function

This module defines basic properties for functions.

Defines the injective predicate. `Definition` *is_injective* $(A\ B : $ `Type`$)$ $(f\ :\ A \rightarrow B)$: `Prop`
$:= \forall\ x\ y\ :\ A,\ f\ x = f\ y \rightarrow x = y.$

Defines the onto predicate. `Definition` *is_onto* $(A\ B : $ `Type`$)$ $(f\ :\ A \rightarrow B)$: `Prop`
$:= \forall\ y\ :\ B,\ \exists\ x\ :\ A,\ f\ x = y.$

Defines the bijective predicate. `Definition` *is_bijective* $(A\ B : $ `Type`$)$ $(f\ :\ A \rightarrow B)$: `Prop`
$:= is\_injective\ A\ B\ f \land is\_onto\ A\ B\ f.$

Chapter 8

Library functional-algebra.monoid

This module defines the Monoid record type which represents algebraic structures called Monoids and provides a collection of theorems and axioms describing them.

`Require Import` *Description.*
`Require Import` *base.*
`Require Import` *function.*
`Require Import` *ProofIrrelevance.*
`Require Import` *Bool.*

`Require Import` *Arith.*
`Require Import` *Wf.*
`Require Import` *Wellfounded.*
`Require Import` *Wf_nat.*

`Module` *Monoid.*

Accepts a function, f, and asserts that f is associative. `Definition` *is_assoc* $(T : \texttt{Type})$ $(f : T \rightarrow T \rightarrow T) : \texttt{Prop} := \forall\ x\ y\ z : T, f\ x\ (f\ y\ z) = f\ (f\ x\ y)\ z.$

Accepts two arguments, f and x, and asserts that x is a left identity element w.r.t. f. `Definition` *is_id_l* $(T : \texttt{Type})\ (f : T \rightarrow T \rightarrow T)\ (E : T) : \texttt{Prop} := \forall\ x : T, f\ E\ x = x.$

Accepts two arguments, f and x, and asserts that x is a right identity element w.r.t. f. `Definition` *is_id_r* $(T : \texttt{Type})\ (f : T \rightarrow T \rightarrow T)\ (E : T) : \texttt{Prop} := \forall\ x : T, f\ x\ E = x.$

Accepts two arguments, f and x, and asserts that x is an identity element w.r.t. f. `Definition` *is_id* $(T : \texttt{Type})\ (f : T \rightarrow T \rightarrow T)\ (E : T) : \texttt{Prop} := is\_id\_l\ T\ f\ E \wedge is\_id\_r\ T\ f\ E.$

Accepts three arguments, f, e, and H, where H proves that e is the identity element w.r.t. f, and returns a function that accepts two arguments, x and y, and asserts that y is x's left inverse. `Definition` *is_inv_l* $(T : \texttt{Type})\ (f : T \rightarrow T \rightarrow T)\ (E : T)\ (\_ : is\_id\ T\ f\ E)\ (x\ y : T) : \texttt{Prop} := f\ y\ x = E.$

Accepts three arguments, f, e, and H, where H proves that e is the identity element w.r.t. f, and returns a function that accepts two arguments, x and y, and asserts that y is x's right inverse. `Definition` is_inv_r $(T : \texttt{Type})$ $(f : T \to T \to T)$ $(E : T)$ $(\_ : is\_id\ T\ f\ E)$ $(x\ y : T)$: `Prop` $:= f\ x\ y = E.$

Accepts three arguments, f, e, and H, where H proves that e is the identity element w.r.t. f, and returns a function that accepts two arguments, x and y, and asserts that y is x's inverse. `Definition` is_inv $(T : \texttt{Type})$ $(f : T \to T \to T)$ $(E : T)$ $(H : is\_id\ T\ f\ E)$ $(x\ y : T)$: `Prop` $:= is\_inv\_l\ T\ f\ E\ H\ x\ y \wedge is\_inv\_r\ T\ f\ E\ H\ x\ y.$

Represents algebraic monoids. `Structure` $Monoid$: `Type` $:= monoid$ $\{$

Represents the set of monoid elements. E: `Set`;

Represents the identity element. E_0: E;

Represents the monoid operation. op: $E \to E \to E$;

Asserts that the monoid operator is associative. $op\_is\_assoc : is\_assoc\ E\ op$;

Asserts that E_0 is the left identity element. $op\_id\_l : is\_id\_l\ E\ op\ E\_0$;

Asserts that E_0 is the right identity element. $op\_id\_r : is\_id\_r\ E\ op\ E\_0$

$\}$.

Enable implicit arguments for monoid properties.

Arguments E_0 $\{m\}$.

Arguments op $\{m\}$ x y.

Arguments op_is_assoc $\{m\}$ x y z.

Arguments op_id_l $\{m\}$ x.

Arguments op_id_r $\{m\}$ x.

Define notations for monoid properties.

`Notation` "0" $:= E\_0 : monoid\_scope.$

`Notation` "x + y" $:= (op\ x\ y)$ (`at level` 50, `left associativity`) $: monoid\_scope.$

`Notation` "{+}" $:= op : monoid\_scope.$

`Open Scope` $monoid\_scope.$

`Section` *Theorems.*

Represents an arbitrary monoid.

Note: we use Variable rather than Parameter to ensure that the following theorems are generalized w.r.t m. `Variable` $m : Monoid.$

Represents the set of monoid elements. `Definition` $M := E\ m.$

Accepts one monoid element, x, and asserts that x is the left identity element. `Definition` $op\_is\_id\_l := is\_id\_l\ M\ \{+\}.$

Accepts one monoid element, x, and asserts that x is the right identity element. `Definition` $op\_is\_id\_r := is\_id\_r\ M\ \{+\}$.

Accepts one monoid element, x, and asserts that x is the identity element. `Definition` $op\_is\_id := is\_id\ M\ \{+\}$.

Proves that 0 is the identity element. `Definition` op_id
: $is\_id\ M\ \{+\}\ 0$
:= $conj\ op\_id\_l\ op\_id\_r$.

Proves that the left identity element is unique. `Definition` $op\_id\_l\_uniq$
: $\forall\ x : M, (op\_is\_id\_l\ x) \rightarrow x = 0$
:= `fun` $x\ H$
$\Rightarrow H\ 0\ ||\ a = 0$ @a `by` $\leftarrow op\_id\_r\ x$.

Proves that the right identity element is unique. `Definition` $op\_id\_r\_uniq$
: $\forall\ x : M, (op\_is\_id\_r\ x) \rightarrow x = 0$
:= `fun` $x\ H$
$\Rightarrow H\ 0\ ||\ a = 0$ @a `by` $\leftarrow op\_id\_l\ x$.

Proves that the identity element is unique. `Definition` op_id_uniq
: $\forall\ x : M, (op\_is\_id\ x) \rightarrow x = 0$
:= `fun` x
$\Rightarrow and\_ind$ (`fun` H _ $\Rightarrow op\_id\_l\_uniq\ x\ H$).

Proves the left introduction rule. `Definition` op_intro_l
: $\forall\ x\ y\ z : M, x = y \rightarrow z + x = z + y$
:= `fun` $x\ y\ z\ H$
$\Rightarrow$ `f_equal` $(\{+\}\ z)\ H$.

Proves the right introduction rule. `Definition` op_intro_r
: $\forall\ x\ y\ z : M, x = y \rightarrow x + z = y + z$
:= `fun` $x\ y\ z\ H$
$\Rightarrow eq\_refl\ (x + z)$
$||\ x + z = a + z$ @a `by` $\leftarrow H$.

Accepts two monoid elements, x and y, and asserts that y is x's left inverse. `Definition` $op\_is\_inv\_l := is\_inv\_l\ M\ \{+\}\ 0\ op\_id$.

Accepts two monoid elements, x and y, and asserts that y is x's right inverse. `Definition` $op\_is\_inv\_r := is\_inv\_r\ M\ \{+\}\ 0\ op\_id$.

Accepts two monoid elements, x and y, and asserts that y is x's inverse. `Definition` $op\_is\_inv := is\_inv\ M\ \{+\}\ 0\ op\_id$.

Accepts one argument, x, and asserts that x has a left inverse. `Definition` has_inv_l
:= `fun` $x \Rightarrow \exists\ y : M,\ op\_is\_inv\_l\ x\ y$.

Accepts one argument, x, and asserts that x has a right inverse. `Definition` has_inv_r
:= `fun` $x \Rightarrow \exists\ y : M,\ op\_is\_inv\_r\ x\ y$.

Accepts one argument, x, and asserts that x has an inverse. `Definition` has_inv

$:=$ `fun` $x \Rightarrow \exists\ y : M,\ op\_is\_inv\ x\ y.$

Proves that the left and right inverses of an element must be equal. `Definition` *op_inv_l_r_eq*

$: \forall\ x\ y : M,\ op\_is\_inv\_l\ x\ y \rightarrow \forall\ z : M,\ op\_is\_inv\_r\ x\ z \rightarrow y = z$

$:=$ `fun` $x\ y\ H1\ z\ H2$

$\Rightarrow op\_is\_assoc\ y\ x\ z$

$||\ y + a = (y + x) + z$ @a `by` $\leftarrow H2$

$||\ a = (y + x) + z$ @a `by` $\leftarrow op\_id\_r\ y$

$||\ y = a + z$ @a `by` $\leftarrow H1$

$||\ y = a$ @a `by` $\leftarrow op\_id\_l\ z.$

Proves that the inverse relationship is symmetric. `Definition` *op_inv_sym*

$: \forall\ x\ y : M,\ op\_is\_inv\ x\ y \leftrightarrow op\_is\_inv\ y\ x$

$:=$ `fun` $x\ y$

$\Rightarrow conj$

(`fun` $H : op\_is\_inv\ x\ y$

$\Rightarrow conj\ (proj2\ H)\ (proj1\ H))$

(`fun` $H : op\_is\_inv\ y\ x$

$\Rightarrow conj\ (proj2\ H)\ (proj1\ H)).$

The next few lemmas define special cases where cancellation holds and culminate in the Unique Inverse theorem which asserts that, in a Monoid, every value has at most one inverse.

Proves the left cancellation law for elements possessing a left inverse. `Definition` *op_cancel_l*

$: \forall\ x\ y\ z : M,\ has\_inv\_l\ z \rightarrow z + x = z + y \rightarrow x = y$

$:=$ `fun` $x\ y\ z\ H\ H0$

$\Rightarrow ex\_ind$

(`fun` $u\ H1$

$\Rightarrow op\_intro\_l\ (z + x)\ (z + y)\ u\ H0$

$||\ a = u + (z + y)$ @a `by` $\leftarrow op\_is\_assoc\ u\ z\ x$

$||\ (u + z) + x = a$ @a `by` $\leftarrow op\_is\_assoc\ u\ z\ y$

$||\ a + x = a + y$ @a `by` $\leftarrow H1$

$||\ a = 0 + y$ @a `by` $\leftarrow op\_id\_l\ x$

$||\ x = a$ @a `by` $\leftarrow op\_id\_l\ y)$

$H.$

Proves the right cancellation law for elements possessing a right inverse. `Definition` *op_cancel_r*

$: \forall\ x\ y\ z : M,\ has\_inv\_r\ z \rightarrow x + z = y + z \rightarrow x = y$

$:=$ `fun` $x\ y\ z\ H\ H0$

$\Rightarrow ex\_ind$

(`fun` $u\ H1$

$\Rightarrow op\_intro\_r\ (x + z)\ (y + z)\ u\ H0$

$||\ a = (y + z) + u$ @a `by` $op\_is\_assoc\ x\ z\ u$

$\|\ x + (z + u) = a\ @a$ `by` $op\_is\_assoc\ y\ z\ u$
$\|\ x + a = y + a\ @a$ `by` $\leftarrow H1$
$\|\ a = y + 0\ @a$ `by` $\leftarrow op\_id\_r\ x$
$\|\ x = a\ @a$ `by` $\leftarrow op\_id\_r\ y)$
H.

Proves that an element's left inverse is unique. `Definition` $op\_inv\_l\_uniq$
$: \forall\ x : M,\ has\_inv\_r\ x \rightarrow \forall\ y\ z : M,\ op\_is\_inv\_l\ x\ y \rightarrow op\_is\_inv\_l\ x\ z \rightarrow z = y$
$:=$ `fun` $x\ H\ y\ z\ H0\ H1$
$\Rightarrow$ `let` $H2$
$: z + x = y + x$
$:= H1\ \|\ z + x = a\ @a$ `by` $H0$ `in`
`let` $H3$
$: z = y$
$:= op\_cancel\_r\ z\ y\ x\ H\ H2$ `in`
$H3$.

Proves that an element's right inverse is unique. `Definition` $op\_inv\_r\_uniq$
$: \forall\ x : M,\ has\_inv\_l\ x \rightarrow \forall\ y\ z : M,\ op\_is\_inv\_r\ x\ y \rightarrow op\_is\_inv\_r\ x\ z \rightarrow z = y$
$:=$ `fun` $x\ H\ y\ z\ H0\ H1$
$\Rightarrow$ `let` $H2$
$: x + z = x + y$
$:= H1\ \|\ x + z = a\ @a$ `by` $H0$ `in`
`let` $H3$
$: z = y$
$:= op\_cancel\_l\ z\ y\ x\ H\ H2$ `in`
$H3$.

Proves that an element's inverse is unique. `Definition` op_inv_uniq
$: \forall\ x\ y\ z : M,\ op\_is\_inv\ x\ y \rightarrow op\_is\_inv\ x\ z \rightarrow z = y$
$:=$ `fun` $x\ y\ z\ H\ H0$
$\Rightarrow op\_inv\_l\_uniq\ x$
$(ex\_intro$ (`fun` $y \Rightarrow op\_is\_inv\_r\ x\ y)\ y\ (proj2\ H))$
$y\ z\ (proj1\ H)\ (proj1\ H0)$.

Proves that the identity element is its own left inverse. `Definition` $op\_inv\_0\_l$
$: op\_is\_inv\_l\ 0\ 0$
$:= op\_id\_l\ 0 : 0 + 0 = 0.$

Proves that the identity element is its own right inverse. `Definition` $op\_inv\_0\_r$
$: op\_is\_inv\_r\ 0\ 0$
$:= op\_id\_r\ 0 : 0 + 0 = 0.$

Proves that the identity element is its own inverse. `Definition` op_inv_0
$: op\_is\_inv\ 0\ 0$
$:= conj\ op\_inv\_0\_l\ op\_inv\_0\_r.$

Proves that the identity element has a left inverse. `Definition` $op\_has\_inv\_l\_0$

: *has_inv_l* 0
:= *ex_intro* (*op_is_inv_l* 0) 0 *op_inv_0_l*.

Proves that the identity element has a right inverse. `Definition` *op_has_inv_r_0*
: *has_inv_r* 0
:= *ex_intro* (*op_is_inv_r* 0) 0 *op_inv_0_r*.

Proves that the identity element has an inverse. `Definition` *op_has_inv_0*
: *has_inv* 0
:= *ex_intro* (*op_is_inv* 0) 0 *op_inv_0*.

Every monoid has a subset of elements that possess inverses. This can be seen by noting that, by definition, every monoid has an identity element (this is what distinguishes a monoid from a semigroup) and the identity element is its own inverse. The following theorems explore the behavior of those elements that possess inverses. They will prove especially ueseful when we consider groups where every element possess an inverse.

Accepts two arguments: x; and H, a proof that x has an inverse; and returns x's inverse, y, along with a proof that y is x's inverse. `Definition` *op_neg_strong*

```
  : ∀ x : M, has_inv x → { y | op_is_inv x y }
  := fun x H
       ⇒ constructive_definite_description (op_is_inv x)
            (ex_ind
              (fun y (H0 : op_is_inv x y)
                ⇒ ex_intro
                    (fun y ⇒ op_is_inv x y ∧ ∀ z, op_is_inv x z → y = z)
                    y
                    (conj H0 (fun z H1 ⇒ eq_sym (op_inv_uniq x y z H0 H1))))
              H).
```

Accepts two arguments: x; and H, a proof that x has an inverse. and returns x's inverse. `Definition` *op_neg*
: ∀ *x* : *M*, *has_inv x* → *M*
:= `fun` *x H* ⇒ *proj1_sig* (*op_neg_strong x H*).

`Notation "{-}"` := (*op_neg*) : *monoid_scope*.

Proves that, forall x and H, where H is a proof that x has an inverse, ‘op_neg x H‘ is x's inverse. `Definition` *op_neg_def*
: ∀ (*x* : *M*) (*H* : *has_inv x*), *op_is_inv x* ({-} *x H*)
:= `fun` *x H* ⇒ *proj2_sig* (*op_neg_strong x H*).

Proves that, forall x and H, where H is a proof that x has an inverse, x is the inverse of ‘- x H‘.

Note: this lemma is an immediate consequence of the symmetry of the inverse predicate. `Definition` *op_neg_inv*
: ∀ (*x* : *M*) (*H* : *has_inv x*), *op_is_inv* ({-} *x H*) *x*
:= `fun` *x H*
⇒ (*proj1* (*op_inv_sym x* ({-} *x H*))) (*op_neg_def x H*).

Proves that, forall x and H, where H is a proof that x has an inverse, '- x H' has an inverse.

Note: this lemma is a weakening of 'op_neg_inv' and is used to apply the lemmas and theorems in this section to expressions involving 'op_neg'. `Definition` *op_neg_inv_ex*

```
  : ∀ (x : M) (H : has_inv x), has_inv ({-} x H)
  := fun x H
       ⇒ ex_intro
            (op_is_inv ({-} x H))
            x
            (op_neg_inv x H).
```

Proves that negation is injective over the set of invertible elements - I.E. forall x and y if the negation of x equals the negation of y then x and y must be equal. `Definition` *op_neg_inj*

```
  : ∀ (x : M) (H : has_inv x) (y : M) (H0 : has_inv y),
      {-} x H = {-} y H0 →
      x = y
  := fun x H y H0 H1
       ⇒ let H2
            : x + ({-} x H) = y + ({-} x H)
            := (proj2 (op_neg_def x H) : x + ({-} x H) = 0)
                 || x + ({-} x H) = a @a by proj2 (op_neg_def y H0)
                 || x + ({-} x H) = y + a @a by H1 in
          let H3
            : x = y
            := op_cancel_r x y ({-} x H)
                 (ex_intro
                    (op_is_inv_r ({-} x H))
                    x
                    (proj2 (proj1 (op_inv_sym x ({-} x H)) (op_neg_def x H))))
                 H2 in
          H3.
```

Proves that double negation is equivalent to the identity function for all invertible values.

Note: the monoid negation operation requires a proof that the value passed to it is invertible. The form of this theorem explicitly asserts that double negation is equivalent to the identity operation for any proof that the negative has an inverse. `Definition` *op_cancel_neg_gen*

```
  : ∀ (x : M) (H : has_inv x) (H0 : has_inv ({-} x H)), {-} ({-} x H) H0 = x
  := fun x H H0
       ⇒ let H1
            : op_is_inv ({-} x H) ({-} ({-} x H) H0)
            := op_neg_def ({-} x H) H0 in
          let H3
```

```
        : op_is_inv ({-} x H) x
        := op_neg_inv x H in
      op_inv_uniq ({-} x H) x ({-} ({-} x H) H0) H3 H1.
```

Proves that double negation is equivalent to the identity function for all invertible values.

```
Definition op_cancel_neg
  : ∀ (x : M) (H : has_inv x), {-} ({-} x H) (op_neg_inv_ex x H) = x
  := fun x H
       ⇒ op_cancel_neg_gen x H (op_neg_inv_ex x H).
```

Proves that negation is onto over the subset of invertable values.

```
Definition op_neg_onto
  : ∀ (y : M) (H : has_inv y), ∃ (x : M) (H0 : has_inv x), {-} x H0 = y
  := fun y H
       ⇒ ex_intro
           (fun x ⇒ ∃ H0 : has_inv x, {-} x H0 = y)
           ({-} y H)
           (ex_intro
             (fun H0 ⇒ {-} ({-} y H) H0 = y)
             (op_neg_inv_ex y H)
             (op_cancel_neg y H)).
```

Proves that invertability is closed over the monoid operation.

Note: given this theorem, we can conclude that the set of invertible elements within a monoid, which must be nonempty, forms a group.

```
Definition op_inv_closed
  : ∀ (x : M) (H : has_inv x) (y : M) (H0 : has_inv y), has_inv (x + y)
  := fun x H y H0
       ⇒ ex_ind
           (fun u (H1 : op_is_inv x u)
             ⇒ ex_ind
                 (fun v (H2 : op_is_inv y v)
                   ⇒ ex_intro
                       (op_is_inv (x + y))
                       (v + u)
                       (conj
                         (op_is_assoc (v + u) x y
                           || (v + u) + (x + y) = a + y @a by op_is_assoc v u x
                           || (v + u) + (x + y) = (v + a) + y @a by ← proj1 H1
                           || (v + u) + (x + y) = a + y @a by ← op_id_r v
                           || (v + u) + (x + y) = a @a by ← proj1 H2
                         )
                         (op_is_assoc (x + y) v u
                           || (x + y) + (v + u) = a + u @a by op_is_assoc x y v
                           || (x + y) + (v + u) = (x + a) + u @a by ← proj2 H2
                           || (x + y) + (v + u) = a + u @a by ← op_id_r x
                           || (x + y) + (v + u) = a @a by ← proj2 H1
```

$)))$

H0)

H.

Proves that every boolean value is either true or false. `Definition` *bool_dec0*
: ∀ *b* : *bool*, {*b* = *true*}+{*b* = *false*}
:= *bool_rect*
(`fun` *b* ⇒ {*b* = *true*}+{*b* = *false*})
(`left` (*true* = *false*) (*eq_refl* *true*))
(`right` (*false* = *true*) (*eq_refl* *false*)).

`End` *Theorems*.

`End` *Monoid*.

Coq does not export notations outside of sections. Consequently the notationsdefined above are not visible to other modules. To fix this, we gather all the notations here for export.

`Notation` "0" := (*Monoid.E_0*) : *monoid_scope*.

`Notation` "x + y" := (*Monoid.op* *x* *y*) (`at level` 50, `left associativity`) : *monoid_scope*.

`Notation` "{+}" := (*Monoid.op*) : *monoid_scope*.

`Notation` "{-}" := (*Monoid.op_neg* _) : *monoid_scope*.

Chapter 9

Library functional-algebra.monoid_expr

This module defines concrete expressions that can be used to represent monoid values and operations, and includes a collection of functions that can be used to manipulate these expressions, and a set of theorems describing these functions.

```
Require Import Description.
Require Import base.
Require Import function.
Require Import ProofIrrelevance.
Require Import Bool.
Require Import List.
Require Import monoid.
Import Monoid.
```

```
Module MonoidExpr.
```

```
Open Scope monoid_scope.
```

```
Section Definitions.
```

Represents the values stored in binary trees. `Variable Term : Set.`

Represents binary trees.

Binary trees can be used to represent many different types of algebraic expressions. Importantly, when flattened, they are isomorphic with lists. Flattening, projecting onto lists, sorting, and folding may be used normalize ("simplify") algebraic expressions.

```
Inductive BTree : Set
  := leaf : Term → BTree
  | node : BTree → BTree → BTree.
```

Accepts a binary tree and returns true iff the tree is a node term.

```
Definition BTree_is_node
  : BTree → bool
  := BTree_rec
```

(fun _ ⇒ *bool*)
(fun _ ⇒ *false*)
(fun _ _ _ _ ⇒ *true*).

Accepts a binary tree and returns true iff the tree is a leaf term. `Definition` *BTree_is_leaf*
: *BTree* → *bool*
:= *BTree_rec*
(fun _ ⇒ *bool*)
(fun _ ⇒ *true*)
(fun _ _ _ _ ⇒ *false*).

Accepts a binary tree and returns true iff the tree is right associative.
Note: right associative trees are isomorphic to lists. `Definition` *BTree_is_rassoc*
: *BTree* → *bool*
:= *BTree_rec*
(fun _ ⇒ *bool*)
(fun _ ⇒ *true*)
(fun *t* _ _ *f*
⇒ *BTree_is_leaf t* && *f*).

Proves that the right subtree in a right associative binary tree is also right associative. `Definition` *BTree_rassoc_thm*
: ∀ *t u* : *BTree*, *BTree_is_rassoc* (*node t u*) = *true* → *BTree_is_rassoc u* = *true*
:= fun *t u H*
⇒ *proj2* (
andb_prop
(*BTree_is_leaf t*)
(*BTree_is_rassoc u*)
H).

`End` *Definitions*.

Arguments leaf {*Term*} *x*.

Arguments node {*Term*} *t u*.

Arguments BTree_is_leaf {*Term*} *t*.

Arguments BTree_is_node {*Term*} *t*.

Arguments BTree_is_rassoc {*Term*} *t*.

Arguments BTree_rassoc_thm {*Term*} *t u H*.

Represents a mapping from abstract terms to monoid set elements. `Structure` *Term_map*
: `Type` := *term_map* {
Represents the monoid set that terms will be projected onto. *term_map_m*: *Monoid*;

Represents the set of terms that will be used to represent monoid values. *term_map_term*
: `Set`;

Accepts a term and returns its projection in E. $term\_map\_eval$: $term\_map\_term \to E\ term\_map\_m$;

Accepts a term and returns true iff the term represents the monoid identity element (0). $term\_map\_is\_zero$: $term\_map\_term \to bool$;

Accepts a term and proves that zero terms evaluate to 0. $term\_map\_is\_zero\_thm$: $\forall\ t,\ term\_map\_is\_zero\ t = true \to term\_map\_eval\ t = 0$
}.

Arguments term_map_eval {t} x.

Arguments term_map_is_zero {t} x.

Arguments term_map_is_zero_thm {t} t0 H.

`Section` *Functions.*

Represents an arbitrary homomorphism mapping binary trees onto some set. `Variable` *map* : *Term_map.*

Represents the set of monoid values. `Let` $E := E\ (term\_map\_m\ map)$.

Represents the set of terms. `Let` $Term := term\_map\_term\ map$.

Accepts a term and returns true iff it is not a zero constant term. `Definition` *Term_is_nonzero*

```
  : Term → bool
  := fun t ⇒ negb (term_map_is_zero t).
```

Maps binary trees onto monoid expressions. `Definition` *BTree_eval*

```
  : BTree Term → E
  := BTree_rec Term
       (fun _ ⇒ E)
       (fun t ⇒ term_map_eval t)
       (fun _ f _ g ⇒ f + g).
```

Accepts two monoid expressions and returns true iff they are denotationally equivalent - I.E. represent the same monoid value. `Definition` *BTree_eq*

```
  : BTree Term → BTree Term → Prop
  := fun t u ⇒ BTree_eval t = BTree_eval u.
```

Accepts two binary trees, t and u, where u is right associative, prepends t onto u in a way that produces a flat list.

```
* * / \ / \ * v => (t) * / \ / \ t u (u) v
```

`Definition` *BTree_shift*

```
  : ∀ (t u : BTree Term), BTree_is_rassoc u = true → { v : BTree Term | BTree_is_rassoc v = true ∧ BTree_eq (node t u) v }
  := let P t u v
```

```
      := BTree_is_rassoc v = true ∧ BTree_eq (node t u) v in
    let T t u
      := BTree_is_rassoc u = true → { v | P t u v } in
    BTree_rec Term
      (fun t ⇒ ∀ u, T t u)
      (fun x u H
        ⇒ let v := node (leaf x) u in
           exist
             (P (leaf x) u)
             v
             (conj
               (andb_true_intro
                 (conj
                   (eq_refl true : BTree_is_leaf (leaf x) = true)
                   H))
               (eq_refl (BTree_eval v))))
      (fun t f u g v H
        ⇒ let (w, H0) := g v H in
           let (x, H1) := f w (proj1 H0) in
           exist
             (P (node t u) v)
             x
             (conj
               (proj1 H1)
               (proj2 H1
                 || BTree_eval t + a = BTree_eval x @a by proj2 H0
                 || a = BTree_eval x @a by ← op_is_assoc (BTree_eval t) (BTree_eval
u) (BTree_eval v)))).
```

Accepts a binary tree and returns an equivalent tree that is right associative. `Definition` *BTree_rassoc*

```
  : ∀ t : BTree Term, { u : BTree Term | BTree_is_rassoc u = true ∧ BTree_eq t u }
  := let P t u
      := BTree_is_rassoc u = true ∧ BTree_eq t u in
    let T t
      := { u | P t u } in
    BTree_rec Term
      (fun t ⇒ T t)
      (fun x
        ⇒ let t := leaf x in
           exist
             (P t)
             t
```

```
          (conj
            (eq_refl true : BTree_is_leaf t = true)
            (eq_refl (BTree_eval t))))
    (fun t _ u g
      ⇒ let (v, H) := g in
        let (w, H0) := BTree_shift t v (proj1 H) in
        exist
          (P (node t u))
          w
          (conj
            (proj1 H0)
            (proj2 H0
              || BTree_eval t + a = BTree_eval w @a by (proj2 H)))).
```

In the following section, we use the isomorphism between right associative binary trees and lists to represent monoid expressions as lists and to use list filtering to eleminate identity elements. This is part of a larger effort to "simplify" momoid expressions.

Accepts a list of monoid elements and computes their sum.

```
Definition list_eval
  : ∀ xs : list Term, E
  := list_rec
      (fun _ ⇒ E)
      0
      (fun x _ f ⇒ (term_map_eval x) + f).
```

Accepts two term lists and asserts that they are equivalent.

```
Definition list_eq : list Term → list Term → Prop
  := fun xs ys : list Term
      ⇒ list_eval xs = list_eval ys.
```

Accepts a right associative binary tree and returns an equivalent list.

```
Definition RABTree_list
  : ∀ t : BTree Term, BTree_is_rassoc t = true → { xs : list Term | BTree_eval t = list_eval xs }
  := let P t xs := BTree_eval t = list_eval xs in
     let T t := BTree_is_rassoc t = true → { xs | P t xs } in
     BTree_rect Term
       (fun t ⇒ T t)
       (fun x _
         ⇒ let xs := cons x nil in
           exist
             (P (leaf x))
             xs
             (eq_sym (op_id_r (term_map_eval x))))
       (BTree_rect Term
         (fun t ⇒ T t → ∀ u, T u → T (node t u))
```

```
          (fun x _ u (g : T u) H
            ⇒ let H0
                  : BTree_is_rassoc u = true
                  := BTree_rassoc_thm (leaf x) u H in
                let (ys, H1) := g H0 in
                let xs := cons x ys in
                exist
                  (P (node (leaf x) u))
                  xs
                  (eq_refl ((term_map_eval x) + (BTree_eval u))
                    || (term_map_eval x) + (BTree_eval u) = (term_map_eval x) + a @a
by ← H1))
          (fun t _ u _ _ v _ H
            ⇒ False_rec
                  { xs | P (node (node t u) v) xs }
                  (diff_false_true H))).
```

Accepts a list of monoid elements and filters out the 0 (identity) elements.

Note: to define this function we must have a way to recognize identity elements. The original definition for monoids did not declare 0 to be a distinguished element. In part this followed from the fact that the set of monoid elements was not declared inductively.

While we cannot assume that models of monoids will define their element sets inductively (for example, note that reals are not defined inductively), we can reasonably expect these models to define 0 as a distinguished element.

As this is somewhat conjectural however, we do not add this as a requirement to the monoid specification, but instead accept the decision procedure here. Definition list_filter_0

```
  : ∀ xs : list Term, { ys : list Term | list_eq xs ys ∧ Is_true (forallb Term_is_nonzero ys)
}
  := let P xs ys := list_eq xs ys ∧ Is_true (forallb Term_is_nonzero ys) in
      let T xs := { ys | P xs ys } in
      list_rec
        T
        (exist
          (P nil)
          nil
          (conj
            (eq_refl E_0)
            I))
        (fun x
          ⇒ (sumbool_rec
                (fun _ ⇒ ∀ xs : list Term, T xs → T (cons x xs))
                (fun (H : term_map_is_zero x = true) xs f
                  ⇒ let H0
```

```
                  : term_map_eval x = 0
                  := term_map_is_zero_thm x H in
                let (ys, H1) := f in
                exist
                  (P (cons x xs))
                  ys
                  (conj
                    (op_id_l (list_eval xs)
                      || 0 + (list_eval xs) = a @a by ← (proj1 H1)
                      || a + (list_eval xs) = list_eval ys @a by H0)
                    (proj2 H1)))
          (fun (H : term_map_is_zero x = false) xs f
            ⇒ let (ys, H0) := f in
                let zs := cons x ys in
                exist
                  (P (cons x xs))
                  zs
                  (conj
                    (eq_refl (list_eval (cons x xs))
                      || term_map_eval x + (list_eval xs) = term_map_eval x + a
@a by ← (proj1 H0))
                    (Is_true_eq_left
                      (forallb Term_is_nonzero zs)
                      (andb_true_intro
                        (conj
                          (eq_refl (Term_is_nonzero x)
                            || Term_is_nonzero x = negb a @a by ← H)
                          (Is_true_eq_true
                            (forallb Term_is_nonzero ys)
                            (proj2 H0)))))))
          (bool_dec0 (term_map_is_zero x)))).
```

Accepts a binary tree and returns an equivalent terms list in which all identity elements have been eliminated.

```
Definition reduce
  : ∀ t : BTree Term, { xs : list Term | BTree_eval t = list_eval xs }
  := fun t
       ⇒ let (u, H) := BTree_rassoc t in
           let (xs, H0) := RABTree_list u (proj1 H) in
           let (ys, H1) := list_filter_0 xs in
           exist
             (fun ys ⇒ BTree_eval t = list_eval ys)
             ys
             ((proj2 H)
```

$\| BTree\_eval\ t = a\ @a$ by $\leftarrow H0$
$\| BTree\_eval\ t = a\ @a$ by $\leftarrow (proj1\ \ H1))$.

End *Functions.*

Section *Theorems.*

Represents an arbitrary monoid. Variable $m : Monoid$.

Represents the set of monoid elements. Let $E := E\ m$.

Represents monoid values.

Note: In the development that follows, we will use binary trees and lists to represent monoid expressions. We will effectively flatten a tree and filter a list to "simplify" a given expression.

The code that flattens the tree representation does not need to care whether or not the leaves in the tree represent 0 (the monoid identity element), inverses, etc. Accordingly, distinguishing these elements in the definition of BTree would unnecessarily complicate the tree algorithms by adding more recursion cases.

Instead of doing this, we use two types to represent monoid expressions

- trees to represent "terms" (expressions

that are summed together) and Term. Term tracks whether or not a monoid value equals 0 (and later we will use a similar structure to indicate whether or not a given group element is an inverse). This makes this information available when needed (specifically when we eliminate 0s using list filtering) without complicating the tree algorithms. Inductive *Term* : Set

```
  := term_0 : Term
  | term_const : E → Term.
```

Accepts a term and returns the monoid value that it represents. Definition *Term_eval*

```
  : Term → E
  := Term_rec
       (fun _ ⇒ E)
       0
       (fun x ⇒ x).
```

Accepts a term and returns true iff the term is zero. Definition *Term_is_zero*

```
  : Term → bool
  := Term_rec
       (fun _ ⇒ bool)
       true
       (fun _ ⇒ false).
```

Proves that Term_is_zero is correct. Definition *Term_is_zero_thm*

```
  : ∀ t, Term_is_zero t = true → Term_eval t = 0
  := Term_ind
       (fun t ⇒ Term_is_zero t = true → Term_eval t = 0)
```

```
      (fun _ ⇒ eq_refl 0)
      (fun x H
        ⇒ False_ind
            (Term_eval (term_const x) = 0)
            (diff_false_true H)).
```

Defines a map from Term to monoid elements. `Definition` *MTerm_map*
: *Term_map*
:= *term_map m Term Term_eval Term_is_zero Term_is_zero_thm.*

`Section` *Unittests.*

`Let` *map* := *MTerm_map.*

`Variables` *a b c d* : *Term.*

```
Let BFTree_rassoc_test_0
  := proj1_sig (BTree_rassoc map (node (node (leaf a) (leaf b)) (leaf c))) =:=
     node (leaf a) (node (leaf b) (leaf c)).
```

```
Let BTree_rassoc_test_1
  := proj1_sig (BTree_rassoc map (node (leaf a) (node (leaf b) (node (leaf c) (leaf d)))))
=:=
     node (leaf a) (node (leaf b) (node (leaf c) (leaf d))).
```

```
Let BTree_rassoc_test_2
  := proj1_sig (BTree_rassoc map (node (node (leaf a) (leaf b)) (node (leaf c) (leaf d))))
=:=
     node (leaf a) (node (leaf b) (node (leaf c) (leaf d))).
```

`End` *Unittests.*

`End` *Theorems.*

Arguments term_0 {*m*}.

Arguments term_const {*m*} *x*.

`End` *MonoidExpr.*

`Notation` "X # Y" := (*MonoidExpr.node X Y*) (`at level` 60).

`Notation` "X {{+}} Y" := (*MonoidExpr.node X Y*) (`at level` 60).

`Notation` "{{ 0 }}" := (*MonoidExpr.leaf* (*MonoidExpr.term_0*)).

`Notation` "{{ X }}" := (*MonoidExpr.leaf* (*MonoidExpr.term_const X*)).

Defines a notation that can be used to prove that two monoid expressions are equal using prrof by reflection.

We represent both expressions as binary trees and reduce both trees to the same canonical form demonstrating that their associated monoid expressions are equivalent. `Notation` "'reflect' A 'as' B ==> C 'as' D 'using' E"
:= (`let` *x* := *A* `in`

let y := C in
let t := B in
let u := D in
let r := *MonoidExpr.reduce* E t in
let s := *MonoidExpr.reduce* E u in
let v := *proj1_sig* r in
let w := *proj1_sig* s in
let H
: *MonoidExpr.list_eval* E v = *MonoidExpr.list_eval* E w
:= *eq_refl* (*MonoidExpr.list_eval* E v) : *MonoidExpr.list_eval* E v = *MonoidExpr.list_eval* E w in
let $H0$
: *MonoidExpr.BTree_eval* E t = *MonoidExpr.list_eval* E v
:= *proj2_sig* r in
let $H1$
: *MonoidExpr.BTree_eval* E u = *MonoidExpr.list_eval* E w
:= *proj2_sig* s in
let $H2$
: *MonoidExpr.BTree_eval* E t = x
:= *eq_refl* (*MonoidExpr.BTree_eval* E t) : *MonoidExpr.BTree_eval* E t = x in
let $H3$
: *MonoidExpr.BTree_eval* E u = y
:= *eq_refl* (*MonoidExpr.BTree_eval* E u) : *MonoidExpr.BTree_eval* E u = y in
H
|| a = *MonoidExpr.list_eval* E w @a by $H0$
|| a = *MonoidExpr.list_eval* E w @a by $H2$
|| x = a @a by $H1$
|| x = a @a by $H3$)
(at level 40, left associativity).

Section *Unittests*.

Variable m : *Monoid*.

Variables a b c d : E m.

Let *map* := *MonoidExpr.MTerm_map* m.

Let *reflect_test_0*
: $(a + 0) = (0 + a)$
:= *reflect*
$(a + 0)$
as $(\{\{a\}\} \# \{\{0\}\})$
==>
$(0 + a)$
as $(\{\{0\}\} \# \{\{a\}\})$

```
      using map.
Let reflect_test_1
   : (a + 0) + (0 + b) = a + b
   := reflect
         ((a + 0) + (0 + b))
           as (({{a}} # {{0}}) # ({{0}} # {{b}}))
       ==>
         (a + b)
           as ({{a}} # {{b}})
       using map.
Let reflect_test_2
   : (0 + a) + b = (a + b)
   := reflect
         ((0 + a) + b) as (({{0}} # {{a}}) # {{b}})
       ==>
         (a + b) as ({{a}} # {{b}})
       using map.
Let reflect_test_3
   : (a + b) + (c + d) = a + ((b + c) + d)
   := reflect
         (a + b) + (c + d)
           as (({{a}} # {{b}}) # ({{c}} # {{d}}))
         ==>
         a + ((b + c) + d)
           as ({{a}} # (({{b}} # {{c}}) # {{d}}))
         using map.
Let reflect_test_4
   : (a + b) + (0 + c) = (a + 0) + (b + c)
   := reflect
         (a + b) + (0 + c)
           as (({{a}} # {{b}}) # ({{0}} # {{c}}))
         ==>
         (a + 0) + (b + c)
           as (({{a}} # {{0}}) # ({{b}} # {{c}}))
         using map.
Let reflect_test_5
   : (((a + b) + c) + 0) = (((0 + a) + b) + c)
   := reflect
         (((a + b) + c) + 0)
           as ((({{a}} # {{b}}) # {{c}}) # {{0}})
         ==>
```

```
(((0 + a) + b) + c)
  as ((({{0}} # {{a}}) # {{b}}) # {{c}})
using map.
```

End *Unittests*.

Chapter 10

Library functional-algebra.monoid_group

Every monoid has a nonempty subgroup consisting of the monoid's invertible elements.

`Require Import` *base.*
`Require Import` *monoid.*
`Require Import` *group.*
`Require Import` *Eqdep.*

`Module` *Monoid_Group.*

Represents the homomorphic mapping between the set of invertible elements within a monoid and the group formed over them by the monoid operation. `Structure` *Monoid_Group* : `Type` := *monoid_group* {

Represents the set of monoid elements. *E* : `Set`;

Represents the identity monoid element. *E_0* : *E*;

Represents the monoid operation. *monoid_op*: $E \rightarrow E \rightarrow E$;

Asserts that the monoid operator is associative. *monoid_op_is_assoc* : *Monoid.is_assoc E monoid_op*;

Accepts one monoid element, x, and asserts that x is the left identity element. *monoid_op_is_id_l* := *Monoid.is_id_l E monoid_op*;

Accepts one monoid element, x, and asserts that x is the right identity element. *monoid_op_is_id_r* := *Monoid.is_id_r E monoid_op*;

Accepts one monoid element, x, and asserts that x is the identity element. *monoid_op_is_id* := *Monoid.is_id E monoid_op*;

Asserts that 0 is the left identity monoid element. $monoid\_op\_id\_l : Monoid.is\_id\_l\ E\ monoid\_op\ E\_0$;

Asserts that 0 is the right identity monoid element. $monoid\_op\_id\_r : Monoid.is\_id\_r\ E\ monoid\_op\ E\_0$;

Represents the monoid whose invertable elements we are going to map onto a group. $m := Monoid.monoid\ E\ E\_0\ monoid\_op\ monoid\_op\_is\_assoc\ monoid\_op\_id\_l\ monoid\_op\_id\_r$;

Represents those monoid elements that are invertable.

Note: each value can be seen intuitively as a pair, (x, H), where x is a monoid element and H is a proof that x is invertable. D : `Set`

$:= \{\ x : E \mid Monoid.has\_inv\ m\ x\ \}$;

Accepts a monoid element and a proof that it is invertable and returns its projection in D. D_cons

$: \forall\ x : E,\ Monoid.has\_inv\ m\ x \to D$

$:= exist\ (Monoid.has\_inv\ m)$;

Asserts that any two equal invertable monoid elements, x and y, are equivalent (using dependent equality).

Note: to compare sig elements that differ only in their proof terms, such as (x, H) and (x, H0), we must introduce a new notion of equality called "dependent equality". This relationship is defined in the Eqdep module. D_eq_dep

$: \forall\ (x : E)\ (H : Monoid.has\_inv\ m\ x)\ (y : E)\ (H0 : Monoid.has\_inv\ m\ y),\ y = x \to eq\_dep\ E\ (Monoid.has\_inv\ m)\ y\ H0\ x\ H$;

Given that two invertable monoid elements x and y are equal (using dependent equality), this lemma proves that their projections into D are equal.

Note: this proof is equivalent to:

eq_dep_eq_sig E (Monoid.has_inv m) y x H0 H (D_eq_dep x H y H0 H1).

The definition for eq_dep_eq_sig has been expanded however for compatability with Coq v8.4. D_eq

$: \forall\ (x : E)\ (H : Monoid.has\_inv\ m\ x)\ (y : E)\ (H0 : Monoid.has\_inv\ m\ y),\ y = x \to D\_cons\ y\ H0 = D\_cons\ x\ H$

$:=$ `fun` $x\ H\ y\ H0\ H1$

$\Rightarrow eq\_dep\_ind\ E\ (Monoid.has\_inv\ m)\ y\ H0$

(`fun` $(z : E)\ (H2 : Monoid.has\_inv\ m\ z)$

$\Rightarrow D\_cons\ y\ H0 = D\_cons\ z\ H2)$

$(eq\_refl\ (D\_cons\ y\ H0))\ x\ H\ (D\_eq\_dep\ x\ H\ y\ H0\ H1)$;

Represents the group identity element. *D_0* := *D_cons E_0* (*Monoid.op_has_inv_0 m*);

Represents the group operation.

Note: intuitively this function accepts two invertable monoid elements, (x, H) and (y, H0), and returns (x + y, H1), where H, H0, and H1 are generalized invertability proofs.

```
group_op
  : D → D → D
  := sig_rec
       (fun _ ⇒ D → D)
       (fun (u : E) (H : Monoid.has_inv m u)
         ⇒ sig_rec
              (fun _ ⇒ D)
              (fun (v : E) (H0 : Monoid.has_inv m v)
                ⇒ D_cons
                     (monoid_op u v)
                     (Monoid.op_inv_closed m u H v H0)));
```

Accepts a group element, x, and asserts that x is a left identity element. *group_op_is_id_l* := *Monoid.is_id_l D group_op*;

Accepts a group element, x, and asserts that x is a right identity element. *group_op_is_id_r* := *Monoid.is_id_r D group_op*;

Accepts a group element, x, and asserts that x is an/the identity element. *group_op_is_id* := *Monoid.is_id D group_op*;

Proves that D_0 is a left identity element.

```
group_op_id_l
  : group_op_is_id_l D_0
  := sig_ind
       (fun x ⇒ group_op D_0 x = x)
       (fun (u : E) (H : Monoid.has_inv m u)
          ⇒ D_eq u H (monoid_op E_0 u) (Monoid.op_inv_closed m E_0 (Monoid.op_has_inv_0 m) u H)
              (monoid_op_id_l u));
```

Proves that D_0 is a right identity element.

```
group_op_id_r
  : group_op_is_id_r D_0
  := sig_ind
       (fun x ⇒ group_op x D_0 = x)
       (fun (u : E) (H : Monoid.has_inv m u)
```

```
                ⇒ D_eq u H (monoid_op u E_0) (Monoid.op_inv_closed m u H E_0 (Monoid.op_has_inv_0
m))
                      (monoid_op_id_r u));
```

Proves that D_0 is the identity element.

```
group_op_id
  : group_op_is_id D_0
  := conj group_op_id_l group_op_id_r;
```

Proves that the group operation is associative.

```
group_op_assoc
  : Monoid.is_assoc D group_op
  := sig_ind
        (fun x ⇒ ∀ y z : D, group_op x (group_op y z) = group_op (group_op x y) z)
        (fun (u : E) (H : Monoid.has_inv m u)
          ⇒ sig_ind
              (fun y ⇒ ∀ z : D, group_op (D_cons u H) (group_op y z) = group_op
(group_op (D_cons u H) y) z)
              (fun (v : E) (H0 : Monoid.has_inv m v)
                ⇒ sig_ind
                    (fun z ⇒ group_op (D_cons u H) (group_op (D_cons v H0) z)
= group_op (group_op (D_cons u H) (D_cons v H0)) z)
                    (fun (w : E) (H1 : Monoid.has_inv m w)
                      ⇒ let a
                          : E
                          := monoid_op u (monoid_op v w) in
                        let H2
                          : Monoid.has_inv m a
                          := Monoid.op_inv_closed m u H (monoid_op v w) (Monoid.op_inv_close
m v H0 w H1) in
                        let b
                          : E
                          := monoid_op (monoid_op u v) w in
                        let H3
                          : Monoid.has_inv m b
                          := Monoid.op_inv_closed m (monoid_op u v) (Monoid.op_inv_closed
m u H v H0) w H1 in
                        let X
                          : D
                          := D_cons a H2 in
                        let Y
                          : D
                          := D_cons b H3 in
                        D_eq b H3 a H2
```

$(monoid\_op\_is\_assoc\ u\ v\ w)$
$)));$

Accepts two values, x and y, and asserts that y is a left inverse of x. $group\_op\_is\_inv\_l := Monoid.is\_inv\_l\ D\ group\_op\ D\_0\ group\_op\_id;$

Accepts two values, x and y, and asserts that y is a right inverse of x. $group\_op\_is\_inv\_r := Monoid.is\_inv\_r\ D\ group\_op\ D\_0\ group\_op\_id;$

Accepts two values, x and y, and asserts that y is an inverse of x. $group\_op\_is\_inv := Monoid.is\_inv\ D\ group\_op\ D\_0\ group\_op\_id;$

Accepts two invertable monoid elements, x and y, where y is a left inverse of x and proves that y's projection into D is the left inverse of x's. $group\_op\_inv\_l$
$: \forall\ (u : E)\ (H : Monoid.has\_inv\ m\ u)\ (v : E)\ (H0 : Monoid.has\_inv\ m\ v),$
$Monoid.op\_is\_inv\_l\ m\ u\ v \to$
$group\_op\_is\_inv\_l\ (D\_cons\ u\ H)\ (D\_cons\ v\ H0)$
$:=$ `fun` $(u : E)\ (H : Monoid.has\_inv\ m\ u)\ (v : E)\ (H0 : Monoid.has\_inv\ m\ v)$
$\Rightarrow D\_eq\ E\_0\ (Monoid.op\_has\_inv\_0\ m)\ (monoid\_op\ v\ u)\ (Monoid.op\_inv\_closed\ m\ v\ H0\ u\ H);$

Accepts two invertable monoid elements, x and y, where y is a right inverse of x and proves that y's projection into D is the right inverse of x's. $group\_op\_inv\_r$
$: \forall\ (u : E)\ (H : Monoid.has\_inv\ m\ u)\ (v : E)\ (H0 : Monoid.has\_inv\ m\ v),$
$Monoid.op\_is\_inv\_r\ m\ u\ v \to$
$group\_op\_is\_inv\_r\ (D\_cons\ u\ H)\ (D\_cons\ v\ H0)$
$:=$ `fun` $(u : E)\ (H : Monoid.has\_inv\ m\ u)\ (v : E)\ (H0 : Monoid.has\_inv\ m\ v)$
$\Rightarrow D\_eq\ E\_0\ (Monoid.op\_has\_inv\_0\ m)\ (monoid\_op\ u\ v)\ (Monoid.op\_inv\_closed\ m\ u\ H\ v\ H0);$

Accepts two invertable monoid elements, x and y, where y is the inverse of x and proves that y's projection into D is the inverse of x's. $group\_op\_inv$
$: \forall\ (u : E)\ (H : Monoid.has\_inv\ m\ u)\ (v : E)\ (H0 : Monoid.has\_inv\ m\ v),$
$Monoid.op\_is\_inv\ m\ u\ v \to$
$group\_op\_is\_inv\ (D\_cons\ u\ H)\ (D\_cons\ v\ H0)$
$:=$ `fun` $(u : E)\ (H : Monoid.has\_inv\ m\ u)\ (v : E)\ (H0 : Monoid.has\_inv\ m\ v)\ (H1 : Monoid.op\_is\_inv\ m\ u\ v)$
$\Rightarrow conj\ (group\_op\_inv\_l\ u\ H\ v\ H0\ (proj1\ H1))$
$(group\_op\_inv\_r\ u\ H\ v\ H0\ (proj2\ H1));$

Accepts a group element and returns its inverse, y, along with a proof that y is x's inverse. $group\_op\_neg\_strong$

```
: ∀ x : D, { y : D | group_op_is_inv x y }
:= sig_rec
      (fun x ⇒ { y : D | group_op_is_inv x y })
      (fun (u : E) (H : Monoid.has_inv m u)
        ⇒ let v
            : E
            := Monoid.op_neg m u H in
          let H0
            : Monoid.op_is_inv m u v
            := Monoid.op_neg_def m u H in
          let H1
            : Monoid.has_inv m v
            := Monoid.op_neg_inv_ex m u H in
          exist
            (fun y : D ⇒ group_op_is_inv (D_cons u H) y)
            (D_cons v H1)
            (group_op_inv u H v H1 H0));
```

Proves that every group element has an inverse. *group_op_inv_ex*

```
: ∀ x : D, ∃ y : D, group_op_is_inv x y
:= fun x
      ⇒ let (y, H) := group_op_neg_strong x in
        ex_intro
          (fun y ⇒ group_op_is_inv x y)
          y H;
```

Proves that every group element has a left inverse. *group_op_inv_l_ex*

```
: ∀ x : D, ∃ y : D, group_op_is_inv_l x y
:= fun x
      ⇒ ex_ind
          (fun y (H : group_op_is_inv x y)
            ⇒ ex_intro (fun z ⇒ group_op_is_inv_l x z) y (proj1 H))
          (group_op_inv_ex x);
```

Proves that every group element has a right inverse. *group_op_inv_r_ex*

```
: ∀ x : D, ∃ y : D, group_op_is_inv_r x y
:= fun x
      ⇒ ex_ind
          (fun y (H : group_op_is_inv x y)
            ⇒ ex_intro (fun z ⇒ group_op_is_inv_r x z) y (proj2 H))
          (group_op_inv_ex x);
```

Proves that the set of invertable monoid elements form a group over the monoid operation.

$g := Group.group\ D\ D\_0\ group\_op\ group\_op\_assoc$
$\quad group\_op\_id\_l\ group\_op\_id\_r\ group\_op\_inv\_l\_ex$
$\quad group\_op\_inv\_r\_ex$
}.

`End` *Monoid_Group*.

Chapter 11

Library functional-algebra.group_expr

This module can be used to automatically solve equations concerning group expressions.

To do this, we use a technique called reflection. Briefly, we represent group expressions as abstract trees, then we call a function that "simplifies" these trees to some canonical form. We prove that, if two group expressions have the same canonical representation, the expressions are equal.

```
Require Import base.
Require Import monoid.
Require Import monoid_expr.
Require Import group.
Import Group.

Module GroupExpr.

Variable g : Group.

Let E := E g.

Let F := Monoid.E (op_monoid g).

Section Unittests.

Variables a b c d : E.

Let map := MonoidExpr.MTerm_map (op_monoid g).
```

Proves that every group has a monoid set and that group elements can be coerced into monoid elements.

Note: this is significant because it allows us to reduce group expressions using functions provided by MonoidExpr.

```
Let E_test_0
  : F = E
  := eq_refl _.

Let reflect_test_0
  : (a + b) + (c + ({-} d)) = a + ((b + c) + ({-} d))
  := reflect
       (a + b) + (c + ({-} d))
```

```
  as (({{a : F}} # {{b : F}}) # ({{c : F}} # {{({-} d) : F}}))
==>
a + ((b + c) + ({-} d))
  as ({{a : F}} # (({{b : F}} # {{c : F}}) # {{({-} d) : F}}))
using map.
```

```
End Unittests.
```

Note: the unittests given above demonstrate that we use monoid terms to simplify group expressions, but we lose information about negation when we do so. Accordingly, we define an alternate term type that captures this information.

The critical functions are BTree_rassoc and reduce. BTree_rassoc needs the fact that terms encapsulate monoidic values to prove its correctness theorem.

```
End GroupExpr.
```