Manual alignment of MathComp and WikiData Concepts

Authors: Talia Ringer & Stefania Dumbrava

Concept	MathComp	WikiData	WikiData Identifier	Definition
Ring	ssralg.ringType	Ring	Q534381	non-commutative rings (semi rings with an opposite) The HB class is called Ring.
Semiring	ssralg.semiRingType	Semiring	Q1333055	non-commutative semirings (NModule with a multiplication) The HB class is called SemiRing.
Field	ssralg.fieldType	field	Q190109	commutative fields The HB class is called Field.
Finite field	algebra.finalg,finFieldType	finite field	Q603880	The finite counterpart of fieldType
Monoid			Q208237	
Commutative Ring	algebra.ssralg.comRingType	commutative ring	Q858656	commutative rings The HB class is called ComRing.
Non-commutative Semiring	algebra.ssralg.semiRingType	N/A	N/A	non-commutative semirings (NModule with a multiplication) The HB class is called SemiRing.
Commutative Semiring	algebra.ssralg.comSemiringType	N/A	N/A	commutative semirings The HB class is called ComSemiRing

Commutative/Abelian Monoid	algebra.ssralg.nmodType	abelian monoid	Q19934355	additive abelian monoid The HB class is called Nmodule.
Additive Abelian Group	algebra.ssralg.zmodType	N/A	N/A	additive abelian group (Nmodule with an opposite) The HB class is called Zmodule.
Left Algebra	algebra.ssralg.lalgType	N/A	N/A	left algebra, ring with scaling that associates on the left The HB class is called Lalgebra.
Algebra	algebra.ssralg.algType	algebra	Q1000660	ring with scaling that associates both left and right The HB class is called Algebra.
Commutative Algebra	algebra.ssralg.comAlgType	commutative algebra	Q727659	commutative algType The HB class is called ComAlgebra.
Unit Ring	algebra.ssralg.unitRingType	unit ring	Q118084	Rings whose units have computable inverses The HB class is called UnitRing.
Commutative Unit Ring	algebra.ssralg.comUnitRingType	commutative unit ring	N/A	commutative UnitRing The HB class is called ComUnitRing.
Unit Algebra	algebra.ssralg.unitAlgType	unit algebra	Q2621172	algebra with computable inverses The HB class is called UnitAlgebra.

Integral Domain	algebra.ssralg.idomainType	integral domain	Q628792	integral, commutative, ring with partial inverses The HB class is called IntegralDomain.
Closed Fields	algebra.ssralg.closedFieldType	algebraically closed field	Q1047547	algebraically closed fields The HB class is called ClosedField.
Subalgebra	algebra.ssralg.subAlgType	subalgebra	Q629933	join of algType and subType (P : pred V) such that val is linear The HB class is called SubAlgebra.
Subring	algebra.ssralg.subRingType	subring	Q929536	join of ringType and subType (P : pred R) such that val is a morphism The HB class is called SubRing.
Subfield	algebra.ssralg.subField	subfield	Q91327913	join of fieldType and subType (P : pred R) such that val is a ring morphism The HB class is called SubField.
Linear function	algebra.ssralg.Linear.type	linear function	Q15854269	linear functions : U -> V
Additive	algebra.ssralg.additive	additive function	Q95744479	f of type U -> V is additive, i.e., f maps the Zmodule structure of U to that of V, 0 to 0 - to - and + to + (equivalently,

				binary - to -) := {morph f : u v / u - v}
Multiplicative	algebra.ssralg.multiplicative	multiplicative function	Q1048447	f of type R -> S is multiplicative, i.e., f maps 1 and * in R to 1 and * in S, respectively R and S must have canonical semiRingType instances
Scalar	algebra.ssralg.scalar	scalar function	Q91108373	scalar f <-> f of type U -> R is a scalar function, i.e., f (a *: u + v) = a * f u + f v
Finite Ring	algebra.finalg.finRingType	finite ring	Q2354159	The finite counterpart of ringType
Finite Algebra	algebra.finalg.finAlgType	finite algebra	Q85760939	The finite counterpart of algType
Finite Unit Algebra	algebra.finalg.finUnitAlgType	N/A	N/A	The finite counterpart of unitAlgType
Rational Number	algebra.rat.rat	rational number	Q1244890	the type of rational number, with single constructor Rat
Numerator	algebra.rat.numq	numerator	Q2279584	numerator of (r : rat)
Denominator	algebra.rat.denq	denominator	Q3044574	denominator of (r : rat)
Finite Abelian Group		Finite_abelian_group	Q181296	

Finite Group	fingroup.fingroup.finGroupType	Finite_group	Q1057968	the structure for finite types with a group law The HB class is called FinGroup.
Finite Subset		Finite_subset	Q36161	
Finite Union		Finite_union	Q185359	
		First_Isomorphism_Theorem	Q1065966	

https://github.com/math-comp/math-comp/blob/master/mathcomp/algebra/ssralg.v
https://github.com/math-comp/math-comp/blob/master/mathcomp/algebra/finalg.v
https://github.com/math-comp/math-comp/blob/master/mathcomp/algebra/rat.v
https://github.com/math-comp/math-comp/blob/master/mathcomp/fingroup/fingroup.v

Q603880

```
(******************************************************************************)
(*         {group gT}  == type of groups with elements of type gT           *)
(*    baseFinGroupType == the structure for finite types with a monoid law  *)
(*                        and an involutive antimorphism; finGroupType is   *)
(*                        derived from baseFinGroupType                     *)
(*                        The HB class is called BaseFinGroup.              *)
(*   FinGroupType mulVg == the finGroupType structure for an existing       *)
(*                        baseFinGroupType structure, built from a proof of *)
(*                        the left inverse group axiom for that structure's *)
(*                        operations                                        *)
(*        [group of G] == a clone for an existing {group gT} structure on   *)
(*                        G : {set gT} (the existing structure might be for *)
(*                        some delta-expansion of G)                        *)
(* If gT implements finGroupType, then we can form {set gT}, the type of      *)
(* finite sets with elements of type gT (as finGroupType extends finType).    *)
(* The group law extends pointwise to {set gT}, which thus implements a sub-  *)
(* interface baseFinGroupType of finGroupType. To be consistent with the      *)
(* predType interface, this is done by coercion to FinGroup.arg_sort, an      *)
(* alias for FinGroup.sort. Accordingly, all pointwise group operations below *)
(* have arguments of type (FinGroup.arg_sort) gT and return results of type   *)
(* FinGroup.sort gT.                                                          *)
(*   The notations below are declared in two scopes:                          *)
(*     group_scope (delimiter %g) for point operations and set constructs.    *)
(*     Group_scope (delimiter %G) for explicit {group gT} structures.         *)
```

```
(* These scopes should not be opened globally, although group_scope is often  *)
(* opened locally in group-theory files (via Import GroupScope).              *)
(*   As {group gT} is both a subtype and an interface structure for {set gT}, *)
(* the fact that a given G : {set gT} is a group can (and usually should) be  *)
(* inferred by type inference with canonical structures. This means that all  *)
(* `group' constructions (e.g., the normaliser 'N_G(H)) actually define sets  *)
(* with a canonical {group gT} structure; the %G delimiter can be used to     *)
(* specify the actual {group gT} structure (e.g., 'N_G(H)%G).                 *)
(*  Operations on elements of a group:                                        *)
(*               x * y == the group product of x and y                        *)
(*              x ^+ n == the nth power of x, i.e., x * ... * x (n times)    *)
(*                x^-1 == the group inverse of x                              *)
(*              x ^- n == the inverse of x ^+ n (notation for (x ^+ n)^-1)   *)
(*                   1 == the unit element                                    *)
(*               x ^ y == the conjugate of x by y (i.e., y^-1 * (x * y))     *)
(*            [~ x, y]  == the commutator of x and y (i.e., x^-1 * x ^ y)     *)
(*     [~ x1, ..., xn]  == the commutator of x1, ..., xn (associating left)   *)
(*    \prod_(i ...) x i == the product of the x i (order-sensitive)           *)
(*         commute x y  <-> x and y commute                                   *)
(*      centralises x A <-> x centralises A                                   *)
(*               'C[x] == the set of elements that commute with x             *)
(*             'C_G[x] == the set of elements of G that commute with x        *)
(*               <[x]> == the cyclic subgroup generated by the element x      *)
(*                #[x] == the order of the element x, i.e., #|<[x]>|          *)
(*  Operations on subsets/subgroups of a finite group:                        *)
(*               H * G == {xy | x \in H, y \in G}                             *)
(*   1 or [1] or [1 gT] == the unit group                                     *)
(*         [set: gT]%G == the group of all x : gT (in Group_scope)            *)
(*         group_set G == G contains 1 and is closed under binary product;    *)
(*                        this is the characteristic property of the          *)
(*                        {group gT} subtype of {set gT}                      *)
(*            [subg G] == the subtype, set, or group of all x \in G: this     *)
(*                        notation is defined simultaneously in %type, %g     *)
(*                        and %G scopes, and G must denote a {group gT}       *)
(*                        structure (G is in the %G scope)                    *)
(*         subg, sgval == the projection into and injection from [subg G]     *)
(*                 H^# == the set H minus the unit element                    *)
(*              repr H == some element of H if 1 \notin H != set0, else 1     *)
(*                        (repr is defined over sets of a baseFinGroupType,   *)
(*                        so it can be used, e.g., to pick right cosets.)     *)
(*              x *: H == left coset of H by x                                *)
(*         lcosets H G == the set of the left cosets of H by elements of G    *)
(*              H :* x == right coset of H by x                               *)
(*         rcosets H G == the set of the right cosets of H by elements of G   *)
(*            #|G : H| == the index of H in G, i.e., #|rcosets G H|           *)
(*              H :^ x == the conjugate of H by x                             *)
(*              x ^: H == the conjugate class of x in H                       *)
(*           classes G == the set of all conjugate classes of G               *)
(*             G :^: H == {G :^ x | x \in H}                                  *)
```

```
(*    class_support G H == {x ^ y | x \in G, y \in H}                        *)
(*        commg_set G H == {[~ x, y] | x \in G, y \in H}; NOT the commutator! *)
(*                <<H>> == the subgroup generated by the set H               *)
(*            [~: G, H] == the commmutator subgroup of G and H, i.e.,        *)
(*                         <<commg_set G H>>>                                *)
(*     [~: H1, ..., Hn] == commutator subgroup of H1, ..., Hn (left assoc.)   *)
(*             H <*> G == the subgroup generated by sets H and G (H join G)  *)
(*           (H * G)%G == the join of G H : {group gT} (convertible, but not *)
(*                         identical to (G <*> H)%G)                         *)
(* (\prod_(i ...) H i)%G == the group generated by the H i                   *)
(* {in G, centralised H} <-> G centralises H                                 *)
(* {in G, normalised H} <-> G normalises H                                   *)
(*                    <-> forall x, x \in G -> H :^ x = H                    *)
(*                'N(H) == the normaliser of H                               *)
(*              'N_G(H) == the normaliser of H in G                          *)
(*              H <| G <=> H is a normal subgroup of G                       *)
(*                'C(H) == the centraliser of H                              *)
(*              'C_G(H) == the centraliser of H in G                         *)
(*            gcore H G == the largest subgroup of H normalised by G         *)
(*                         If H is a subgroup of G, this is the largest      *)
(*                         normal subgroup of G contained in H).             *)
(*            abelian H <=> H is abelian                                     *)
(*          subgroups G == the set of subgroups of G, i.e., the set of all   *)
(*                         H : {group gT} such that H \subset G              *)
(* In the notation below G is a variable that is bound in P.                 *)
(*          [max G | P] <=> G is the largest group such that P holds         *)
(*     [max H of G | P] <=> H is the largest group G such that P holds       *)
(*      [max G | P & Q] := [max G | P && Q], likewise [max H of G | P & Q]   *)
(*          [min G | P] <=> G is the smallest group such that P holds        *)
(*      [min G | P & Q] := [min G | P && Q], likewise [min H of G | P & Q]   *)
(*     [min H of G | P] <=> H is the smallest group G such that P holds      *)
(* In addition to the generic suffixes described in ssrbool.v and finset.v,  *)
(* we associate the following suffixes to group operations:                  *)
(*   1 - identity element, as in group1 : 1 \in G                            *)
(*   M - multiplication, as is invMg : (x * y)^-1 = y^-1 * x^-1              *)
(*       Also nat multiplication, for expgM : x ^+ (m * n) = x ^+ m ^+ n     *)
(*   D - (nat) addition, for expgD : x ^+ (m + n) = x ^+ m * x ^+ n          *)
(*   V - inverse, as in mulgV : x * x^-1 = 1                                 *)
(*   X - exponentiation, as in conjXg : (x ^+ n) ^ y = (x ^ y) ^+ n          *)
(*   J - conjugation, as in orderJ : #[x ^ y] = #[x]                         *)
(*   R - commutator, as in conjRg : [~ x, y] ^ z = [~ x ^ z, y ^ z]          *)
(*   Y - join, as in centY : 'C(G <*> H) = 'C(G) :&: 'C(H)                   *)
(* We sometimes prefix these with an `s' to indicate a set-lifted operation, *)
(* e.g., conjsMg : (A * B) :^ x = A :^ x * B :^ x.                           *)
(******************************************************************************)
```

```
(******************************************************************************)
(* This file defines a datatype for rational numbers and equips it with a     *)
(* structure of archimedean, real field, with int and nat declared as closed  *)
(* subrings.                                                                  *)
(*        n%:Q == explicit cast from int to rat, ie. the specialization to   *)
(*                rationals of the generic ring morphism n%:~R               *)
(*      numq r == numerator of (r : rat)                                      *)
(*      denq r == denominator of (r : rat)                                    *)
(*      ratr r == generic embedding of (r : rat) into an arbitrary unit ring.*)
(* [rat x // y] == smart constructor for rationals, definitionally equal      *)
(*                to x / y for concrete values, intended for printing only   *)
(*                of normal forms. The parsable notation is for debugging.   *)
(******************************************************************************)
```

```
(******************************************************************************)
(*      The algebraic part of the algebraic hierarchy for finite types       *)
(*                                                                            *)
(* This file clones the entire ssralg hierarchy for finite types; this       *)
(* allows type inference to function properly on expressions that mix        *)
(* combinatorial and algebraic operators, e.g., [set x + y | x in A, y in A].*)
(*                                                                            *)
(*   finNmodType, finZmodType, finSemiRingType, finComRingType, *)
(*   finComSemiRingType, finUnitRingType, finComUnitRingType, finIdomType,   *)
(*   finLmodType, finLalgType     *)
(*      == the finite counterparts of nmodType, etc                          *)
(* Note that a finFieldType is canonically decidable.                        *)
(*   This file also provides direct tie-ins with finite group theory:        *)
(*     [finGroupMixin of R for +%R] == structures for R                      *)
(*                     {unit R} == the type of units of R, which has a   *)
(*                                canonical group structure            *)
(*                FinRing.unit R Ux == the element of {unit R} corresponding *)
(*                                to x, where Ux : x \in GRing.unit     *)
(*                       'U%act == the action by right multiplication of *)
(*                                {unit R} on R, via FinRing.unit_act   *)
(*                                (This is also a group action.)       *)
(******************************************************************************)
```

Not done yet:

```
(******************************************************************************)
(*      lmodType R == module with left multiplication by external scalars     *)
(*                  in the ring R                                *)
(*                  The HB class is called Lmodule.                  *)                  *)
(*comUnitAlgType R == commutative UnitAlgebra                              *)
(*                  The HB class is called ComUnitAlgebra.               *)
(*    decFieldType == fields with a decidable first order theory             *)
```

```
(*                  The HB class is called DecidableField.                *)
(*                                                                        *)
(* and their joins with subType:                                          *)
(*                                                                        *)
(*        subNmodType V P == join of nmodType and subType (P : pred V) such  *)
(*                          that val is semi_additive                     *)
(*                          The HB class is called SubNmodule.            *)
(*        subZmodType V P == join of zmodType and subType (P : pred V)       *)
(*                          such that val is additive                     *)
(*                          The HB class is called SubZmodule.            *)
(*     subSemiRingType R P == join of semiRingType and subType (P : pred R)   *)
(*                          such that val is a semiring morphism          *)
(*                          The HB class is called SubSemiRing.           *)
(*  subComSemiRingType R P == join of comSemiRingType and subType (P : pred R)*)
(*                          such that val is a morphism                   *)
(*                          The HB class is called SubComSemiRing.        *)
(*      subComRingType R P == join of comRingType and subType (P : pred R)    *)
(*                          such that val is a morphism                   *)
(*                          The HB class is called SubComRing.            *)
(*       subLmodType R V P == join of lmodType and subType (P : pred V)       *)
(*                          such that val is scalable                     *)
(*                          The HB class is called SubLmodule.            *)
(*       subLalgType R V P == join of lalgType and subType (P : pred V)       *)
(*                          such that val is linear                       *)
(*                          The HB class is called SubLalgebra.           *)
(*     subUnitRingType R P == join of unitRingType and subType (P : pred R)   *)
(*                          such that val is a ring morphism              *)
(*                          The HB class is called SubUnitRing.           *)
(*  subComUnitRingType R P == join of comUnitRingType and subType (P : pred R)*)
(*                          such that val is a ring morphism              *)
(*                          The HB class is called SubComUnitRing.        *)
(*      subIdomainType R P == join of idomainType and subType (P : pred R)    *)
(*                          such that val is a ring morphism              *)
(*                          The HB class is called SubIntegralDomain.     *)
(*                                                                        *)
(* Morphisms between the above structures:                                *)
(*                                                                        *)
(*     Additive.type U V == semi additive (resp. additive) functions between  *)
(*                        nmodType (resp. zmodType) instances U and V       *)
(*    RMorphism.type R S == semi ring (resp. ring) morphism between           *)
(*                        semiRingType (resp. ringType) instances R and S   *)
(*   GRing.Scale.law R V == scaling morphism : R -> V -> V                   *)
(*                        The HB class is called GRing.Scale.Law.          *)
(* LRMorphism.type R A B == linear ring morphisms, i.e., algebra morphisms    *)
(*                                                                        *)
(* Closedness predicates for the algebraic structures:                    *)
(*                                                                        *)
(*  opprClosed V == predicate closed under opposite on V : zmodType          *)
(*                The HB class is called OppClosed.                        *)
```

```
(*  addrClosed V == predicate closed under addition on V : nmodType          *)
(*               The HB class is called AddClosed.                 *)
(*  zmodClosed V == predicate closed under opposite and addition on V        *)
(*               The HB class is called ZmodClosed.                   *)
(* mulr2Closed R == predicate closed under multiplication on R : semiRingType *)
(*               The HB class is called Mul2Closed.                  *)
(*  mulrClosed R == predicate closed under multiplication and for 1          *)
(*               The HB class is called MulClosed.                  *)
(*  smulClosed R == predicate closed under multiplication and for -1         *)
(*               The HB class is called SmulClosed.                  *)
(* semiring2Closed R == predicate closed under addition and multiplication    *)
(*               The HB class is called Semiring2Closed.               *)
(* semiringClosed R == predicate closed under semiring operations            *)
(*               The HB class is called SemiringClosed.                *)
(* subringClosed R == predicate closed under ring operations                *)
(*               The HB class is called SubringClosed.                 *)
(*   divClosed R == predicate closed under division                         *)
(*               The HB class is called DivClosed.                   *)
(*  sdivClosed R == predicate closed under division and opposite             *)
(*               The HB class is called SdivClosed.                  *)
(* submodClosed R == predicate closed under lmodType operations              *)
(*               The HB class is called SubmodClosed.                 *)
(* subalgClosed R == predicate closed under lalgType operations              *)
(*               The HB class is called SubalgClosed.                 *)
(* divringClosed R == predicate closed under unitRing operations             *)
(*               The HB class is called DivringClosed.                *)
(* divalgClosed R S == predicate closed under (S : unitAlg R) operations      *)
(*               The HB class is called DivalgClosed.                 *)
(*                                                                 *)
(* Canonical properties of the algebraic structures:                      *)
(*  * nmodType (additive abelian monoids):                               *)
(*                  0 == the zero (additive identity) of a Nmodule         *)
(*              x + y == the sum of x and y (in a Nmodule)                *)
(*             x *+ n == n times x, with n in nat (non-negative), i.e.,    *)
(*                       x + (x + .. (x + x)..) (n terms); x *+ 1 is thus  *)
(*                       convertible to x, and x *+ 2 to x + x            *)
(*       \sum_<range> e == iterated sum for a Zmodule (cf bigop.v)          *)
(*               e`_i == nth 0 e i, when e : seq M and M has a zmodType    *)
(*                       structure                                 *)
(*           support f == 0.-support f, i.e., [pred x | f x != 0]          *)
(*        addr_closed S <-> collective predicate S is closed under finite    *)
(*                        sums (0 and x + y in S, for x, y in S)          *)
(* [SubChoice_isSubNmodule of U by <:] == nmodType mixin for a subType whose  *)
(*                       base type is a nmodType and whose predicate's is  *)
(*                       a nmodClosed                              *)
(*                                                                 *)
(*  * zmodType (additive abelian groups):                               *)
(*                - x == the opposite (additive inverse) of x             *)
(*              x - y == the difference of x and y; this is only notation  *)
```

```
(*                    for x + (- y)                            *)
(*            x *- n == notation for - (x *+ n), the opposite of x *+ n   *)
(*      oppr_closed S <-> collective predicate S is closed under opposite  *)
(*      zmod_closed S <-> collective predicate S is closed under zmodType  *)
(*                    operations (0 and x - y in S, for x, y in S)      *)
(*                    This property coerces to oppr_pred and addr_pred. *)
(* [SubChoice_isSubZmodule of U by <:] == zmodType mixin for a subType whose  *)
(*                    base type is a zmodType and whose predicate's     *)
(*                    is a zmodClosed                             *)
(*                                                          *)
(*  * SemiRing (non-commutative semirings):                            *)
(*               R^c == the converse Ring for R: R^c is convertible to R *)
(*                    but when R has a canonical ringType structure    *)
(*                    R^c has the converse one: if x y : R^c, then     *)
(*                    x * y = (y : R) * (x : R)                     *)
(*                 1 == the multiplicative identity element of a Ring    *)
(*              n%:R == the ring image of an n in nat; this is just      *)
(*                    notation for 1 *+ n, so 1%:R is convertible to 1 *)
(*                    and 2%:R to 1 + 1                             *)
(*          <number> == <number>%:R with <number> a sequence of digits   *)
(*             x * y == the ring product of x and y                    *)
(*     \prod_<range> e == iterated product for a ring (cf bigop.v)        *)
(*            x ^+ n == x to the nth power with n in nat (non-negative), *)
(*                    i.e., x * (x * .. (x * x)..) (n factors); x ^+ 1 *)
(*                    is thus convertible to x, and x ^+ 2 to x * x    *)
(*      GRing.comm x y <-> x and y commute, i.e., x * y = y * x           *)
(*        GRing.lreg x <-> x if left-regular, i.e., *%R x is injective     *)
(*        GRing.rreg x <-> x if right-regular, i.e., *%R x is injective    *)
(*          [char R] == the characteristic of R, defined as the set of   *)
(*                    prime numbers p such that p%:R = 0 in R          *)
(*                    The set [char R] has at most one element, and is *)
(*                    implemented as a pred_nat collective predicate   *)
(*                    (see prime.v); thus the statement p \in [char R] *)
(*                    can be read as `R has characteristic p', while   *)
(*                    [char R] =i pred0 means `R has characteristic 0' *)
(*                    when R is a field.                            *)
(*    Frobenius_aut chRp == the Frobenius automorphism mapping x in R to     *)
(*                    x ^+ p, where chRp : p \in [char R] is a proof   *)
(*                    that R has (non-zero) characteristic p          *)
(*       mulr_closed S <-> collective predicate S is closed under finite   *)
(*                    products (1 and x * y in S for x, y in S)       *)
(*    semiring_closed S <-> collective predicate S is closed under semiring *)
(*                    operations (0, 1, x + y and x * y in S)         *)
(* [SubNmodule_isSubSemiRing of R by <:] ==                             *)
(* [SubChoice_isSubSemiRing of R by <:] == semiRingType mixin for a          *)
(*                    subType whose base type is a semiRingType and    *)
(*                    whose predicate's is a semiringClosed           *)
(*                                                          *)
(*  * Ring (non-commutative rings):                                 *)
```

```
(*        GRing.sign R b := (-1) ^+ b in R : ringType, with b : bool        *)
(*                     This is a parsing-only helper notation, to be    *)
(*                     used for defining more specific instances.       *)
(*        smulr_closed S <-> collective predicate S is closed under products *)
(*                     and opposite (-1 and x * y in S for x, y in S)   *)
(*      subring_closed S <-> collective predicate S is closed under ring     *)
(*                     operations (1, x - y and x * y in S)            *)
(* [SubZmodule_isSubRing of R by <:] ==                                  *)
(* [SubChoice_isSubRing of R by <:] == ringType mixin for a subType whose base*)
(*                     type is a ringType and whose predicate's is a    *)
(*                     subringClosed                                    *)
(*                                                                      *)
(*  * ComSemiRing (commutative SemiRings):                              *)
(* [SubNmodule_isSubComSemiRing of R by <:] ==                          *)
(* [SubChoice_isSubComSemiRing of R by <:] == comSemiRingType mixin for a     *)
(*                     subType whose base type is a comSemiRingType and *)
(*                     whose predicate's is a semiringClosed            *)
(*                                                                      *)
(*  * ComRing (commutative Rings):                                      *)
(* [SubZmodule_isSubComRing of R by <:] ==                              *)
(* [SubChoice_isSubComRing of R by <:] == comRingType mixin for a             *)
(*                     subType whose base type is a comRingType and     *)
(*                     whose predicate's is a subringClosed             *)
(*                                                                      *)
(*  * UnitRing (Rings whose units have computable inverses):            *)
(*     x \is a GRing.unit <=> x is a unit (i.e., has an inverse)         *)
(*                x^-1 == the ring inverse of x, if x is a unit, else x    *)
(*               x / y == x divided by y (notation for x * y^-1)          *)
(*              x ^- n := notation for (x ^+ n)^-1, the inverse of x ^+ n  *)
(*        invr_closed S <-> collective predicate S is closed under inverse   *)
(*        divr_closed S <-> collective predicate S is closed under division  *)
(*                     (1 and x / y in S)                               *)
(*       sdivr_closed S <-> collective predicate S is closed under division  *)
(*                     and opposite (-1 and x / y in S, for x, y in S)  *)
(*      divring_closed S <-> collective predicate S is closed under unitRing  *)
(*                     operations (1, x - y and x / y in S)            *)
(* [SubRing_isSubUnitRing of R by <:] ==                                *)
(* [SubChoice_isSubUnitRing of R by <:] == unitRingType mixin for a subType   *)
(*                     whose base type is a unitRingType and whose      *)
(*                     predicate's is a divringClosed and whose ring    *)
(*                     structure is compatible with the base type's     *)
(*                                                                      *)
(*  * ComUnitRing (commutative rings with computable inverses):         *)
(* [SubChoice_isSubComUnitRing of R by <:] == comUnitRingType mixin for a     *)
(*                     subType whose base type is a comUnitRingType and *)
(*                     whose predicate's is a divringClosed and whose   *)
(*                     ring structure is compatible with the base       *)
(*                     type's                                           *)
(*                                                                      *)
```

```
(*  * IntegralDomain (integral, commutative, ring with partial inverses):     *)
(* [SubComUnitRing_isSubIntegralDomain R by <:] ==                          *)
(* [SubChoice_isSubIntegralDomain R by <:] == mixin axiom for a idomain       *)
(*                        subType                                    *)
(*                                                                   *)
(*  * Field (commutative fields):                                    *)
(*  GRing.Field.axiom inv == field axiom: x != 0 -> inv x * x = 1 for all x   *)
(*                        This is equivalent to the property above, but    *)
(*                        does not require a unitRingType as inv is an     *)
(*                        explicit argument.                         *)
(* [SubIntegralDomain_isSubField of R by <:] == mixin axiom for a field       *)
(*                        subType                                    *)
(*                                                                   *)
(*  * DecidableField (fields with a decidable first order theory):           *)
(*          GRing.term R == the type of formal expressions in a unit ring R  *)
(*                        with formal variables 'X_k, k : nat, and         *)
(*                        manifest constants x%:T, x : R                  *)
(*                        The notation of all the ring operations is       *)
(*                        redefined for terms, in scope %T.               *)
(*       GRing.formula R == the type of first order formulas over R; the %T  *)
(*                        scope binds the logical connectives /\, \/, ~,   *)
(*                        ==>, ==, and != to formulae; GRing.True/False    *)
(*                        and GRing.Bool b denote constant formulae, and   *)
(*                        quantifiers are written 'forall/'exists 'X_k, f  *)
(*                          GRing.Unit x tests for ring units             *)
(*                          GRing.If p_f t_f e_f emulates if-then-else     *)
(*                          GRing.Pick p_f t_f e_f emulates fintype.pick   *)
(*                          foldr GRing.Exists/Forall q_f xs can be used   *)
(*                            to write iterated quantifiers              *)
(*        GRing.eval e t == the value of term t with valuation e : seq R     *)
(*                        (e maps 'X_i to e`_i)                        *)
(*  GRing.same_env e1 e2 <-> environments e1 and e2 are extensionally equal   *)
(*       GRing.qf_form f == f is quantifier-free                           *)
(*       GRing.holds e f == the intuitionistic CiC interpretation of the     *)
(*                        formula f holds with valuation e                *)
(*     GRing.qf_eval e f == the value (in bool) of a quantifier-free f       *)
(*         GRing.sat e f == valuation e satisfies f (only in a decField)     *)
(*         GRing.sol n f == a sequence e of size n such that e satisfies f,  *)
(*                        if one exists, or [::] if there is no such e     *)
(*       'exists 'X_i, u1 == 0 /\ ... /\ u_m == 0 /\ v1 != 0 ... /\ v_n != 0 *)
(*                                                                   *)
(*  * Lmodule (module with left multiplication by external scalars).         *)
(*               a *: v == v scaled by a, when v is in an Lmodule V and a   *)
(*                        is in the scalar Ring of V                     *)
(*       scaler_closed S <-> collective predicate S is closed under scaling  *)
(*       linear_closed S <-> collective predicate S is closed under linear   *)
(*                        combinations (a *: u + v in S when u, v in S)    *)
(*       submod_closed S <-> collective predicate S is closed under lmodType *)
(*                        operations (0 and a *: u + v in S)              *)
```

```
(* [SubZmodule_isSubLmodule of V by <:] ==                          *)
(* [SubChoice_isSubLmodule of V by <:] == mixin axiom for a subType of an     *)
(*                      lmodType                                   *)
(*                                                                 *)
(*  * Lalgebra (left algebra, ring with scaling that associates on the left): *)
(*                R^o == the regular algebra of R: R^o is convertible to  *)
(*                      R, but when R has a ringType structure then R^o  *)
(*                      extends it to an lalgType structure by letting R *)
(*                      act on itself: if x : R and y : R^o then        *)
(*                      x *: y = x * (y : R)                         *)
(*               k%:A == the image of the scalar k in an L-algebra; this  *)
(*                      is simply notation for k *: 1                   *)
(*       subalg_closed S <-> collective predicate S is closed under lalgType *)
(*                      operations (1, a *: u + v and u * v in S)        *)
(* [lalgMixin of V by <:] == mixin axiom for a subType of an lalgType         *)
(* [SubRing_SubLmodule_isSubLalgebra of V by <:] ==                         *)
(* [SubChoice_isSubLalgebra of V by <:] == mixin axiom for a subType of an    *)
(*                      lalgType                                   *)
(*                                                                 *)
(*  * Algebra (ring with scaling that associates both left and right):       *)
(* [SubLalgebra_isSubAlgebra of V by <:] ==                              *)
(* [SubChoice_isSubAlgebra of V by <:] == mixin axiom for a subType of an     *)
(*                      algType                                    *)
(*                                                                 *)
(*  * UnitAlgebra (algebra with computable inverses):                      *)
(*       divalg_closed S <-> collective predicate S is closed under all      *)
(*                      unitAlgType operations (1, a *: u + v and u / v  *)
(*                      are in S fo u, v in S)                        *)
(*                                                                 *)
(*   In addition to this structure hierarchy, we also develop a separate,     *)
(* parallel hierarchy for morphisms linking these structures:               *)
(*                                                                 *)
(* * Additive (semi additive or additive functions):                       *)
(*       semi_additive f <-> f of type U -> V is semi additive, i.e., f maps *)
(*                      the Nmodule structure of U to that of V, 0 to 0  *)
(*                      and + to +                                   *)
(*                   := (f 0 = 0) * {morph f : x y / x + y}              *)
(*      {additive U -> V} == the interface type for a Structure (keyed on     *)
(*                      a function f : U -> V) that encapsulates the      *)
(*                      semi_additive property; both U and V must have   *)
(*                      canonical nmodType instances                    *)
(*                      When both U and V have zmodType instances, it is *)
(*                      an additive function.                         *)
(*                                                                 *)
(* * RMorphism (semiring or ring morphisms):                          *)                              *)
(*     {rmorphism R -> S} == the interface type for semiring morphisms; both  *)
(*                      R and S must have semiRingType instances         *)
(*                      When both R and S have ringType instances, it is *)
(*                      a ring morphism.                              *)
```

```
(*                                                                          *)
(*  -> If R and S are UnitRings the f also maps units to units and inverses   *)
(*     of units to inverses; if R is a field then f is a field isomorphism    *)
(*     between R and its image.                                             *)
(*  -> Additive properties (raddf_suffix, see below) are duplicated and       *)
(*     specialised for RMorphism (as rmorph_suffix). This allows more         *)
(*     precise rewriting and cleaner chaining: although raddf lemmas will     *)
(*     recognize RMorphism functions, the converse will not hold (we cannot   *)
(*     add reverse inheritance rules because of incomplete backtracking in    *)
(*     the Canonical Projection unification), so one would have to insert a  *)
(*     /= every time one switched from additive to multiplicative rules.      *)
(*                                                                          *)
(* * Linear (linear functions):                                             *)
(*           scalable f <-> f of type U -> V is scalable, i.e., f morphs    *)
(*                          scaling on U to scaling on V, a *: _ to a *: _   *)
(*                          U and V must both have lmodType R structures,    *)
(*                          for the same ringType R.                        *)
(*       scalable_for s f <-> f is scalable for scaling operator s, i.e.,     *)
(*                          f morphs a *: _ to s a _; the range of f only    *)
(*                          need to be a zmodType                           *)
(*                          The scaling operator s should be one of *:%R     *)
(*                          (see scalable, above), *%R or a combination      *)
(*                          nu \; *%R or nu \; *:%R with nu : {rmorphism _}; *)
(*                          otherwise some of the theory (e.g., the linearZ  *)
(*                          rule) will not apply.                           *)
(*             linear f <-> f of type U -> V is linear, i.e., f morphs      *)
(*                          linear combinations a *: u + v in U to similar   *)
(*                          linear combinations in V; U and V must both have *)
(*                          lmodType R structures, for the same ringType R   *)
(*                       := forall a, {morph f: u v / a *: u + v}           *)
(*         linear_for s f <-> f is linear for the scaling operator s, i.e.,   *)
(*                          f (a *: u + v) = s a (f u) + f v                *)
(*                          The range of f only needs to be a zmodType, but  *)
(*                          s MUST be of the form described in the          *)
(*                          scalable_for paragraph above for this predicate  *)
(*                          to type check.                                  *)
(*          lmorphism f <-> f is both additive and scalable                  *)
(*                          This is in fact equivalent to linear f, although *)
(*                          somewhat less convenient to prove.              *)
(*      lmorphism_for s f <-> f is both additive and scalable for s           *)
(*       {linear U -> V} == the interface type for linear functions, i.e., a *)
(*                          Structure that encapsulates the linear property  *)
(*                          for functions f : U -> V; both U and V must have *)
(*                          lmodType R structures, for the same R           *)
(*           {scalar U} == the interface type for scalar functions, of type *)
(*                          U -> R where U has an lmodType R structure       *)
(*    {linear U -> V | s} == the interface type for functions linear for s    *)
(*        (a *: u)%Rlin == transient forms that simplify to a *: u, a * u,  *)
(*         (a * u)%Rlin    nu a *: u, and nu a * u, respectively, and are   *)
```

```
(*       (a *:^nu u)%Rlin    created by rewriting with the linearZ lemma      *)
(*        (a *^nu u)%Rlin    The forms allows the RHS of linearZ to be matched*)
(*                           reliably, using the GRing.Scale.law structure.   *)
(* -> Similarly to Ring morphisms, additive properties are specialized for    *)
(*    linear functions.                                                       *)
(* -> Although {scalar U} is convertible to {linear U -> R^o}, it does not    *)
(*    actually use R^o, so that rewriting preserves the canonical structure   *)
(*    of the range of scalar functions.                                       *)
(* -> The generic linearZ lemma uses a set of bespoke interface structures to *)
(*    ensure that both left-to-right and right-to-left rewriting work even in *)
(*    the presence of scaling functions that simplify non-trivially (e.g.,    *)
(*    idfun \; *%R). Because most of the canonical instances and projections  *)
(*    are coercions the machinery will be mostly invisible (with only the     *)
(*    {linear ...} structure and %Rlin notations showing), but users should   *)
(*    beware that in (a *: f u)%Rlin, a actually occurs in the f u subterm.   *)
(* -> The simpler linear_LR, or more specialized linearZZ and scalarZ rules   *)
(*    should be used instead of linearZ if there are complexity issues, as    *)
(*    well as for explicit forward and backward application, as the main      *)
(*    parameter of linearZ is a proper sub-interface of {linear fUV | s}.     *)
(*                                                                            *)
(* * LRMorphism (linear ring morphisms, i.e., algebra morphisms):             *)
(*          lrmorphism f <-> f of type A -> B is a linear Ring (Algebra)      *)
(*                           morphism: f is both additive, multiplicative and *)
(*                           scalable; A and B must both have lalgType R      *)
(*                           canonical structures, for the same ringType R    *)
(*     lrmorphism_for s f <-> f a linear Ring morphism for the scaling        *)
(*                           operator s: f is additive, multiplicative and    *)
(*                           scalable for s; A must be an lalgType R, but B   *)
(*                           only needs to have a ringType structure          *)
(*    {lrmorphism A -> B} == the interface type for linear morphisms, i.e., a *)
(*                           Structure that encapsulates the lrmorphism       *)
(*                           property for functions f : A -> B; both A and B  *)
(*                           must have lalgType R structures, for the same R  *)
(* {lrmorphism A -> B | s} == the interface type for morphisms linear for s   *)
(*  -> Linear and rmorphism properties do not need to be specialized for      *)
(*     as we supply inheritance join instances in both directions.            *)
(* Finally we supply some helper notation for morphisms:                      *)
(*                    x^f == the image of x under some morphism               *)
(*                           This notation is only reserved (not defined)     *)
(*                           here; it is bound locally in sections where some *)
(*                           morphism is used heavily (e.g., the container    *)
(*                           morphism in the parametricity sections of poly   *)
(*                           and matrix, or the Frobenius section here)       *)
(*                     \0 == the constant null function, which has a          *)
(*                           canonical linear structure, and simplifies on    *)
(*                           application (see ssrfun.v)                       *)
(*                 f \+ g == the additive composition of f and g, i.e., the   *)
(*                           function x |-> f x + g x; f \+ g is canonically  *)
(*                           linear when f and g are, and simplifies on       *)
```

```
(*                        application (see ssrfun.v)                   *)
(*               f \- g == the function x |-> f x - g x, canonically        *)
(*                        linear when f and g are, and simplifies on       *)
(*                        application                                *)
(*                 \- g == the function x |-> - f x, canonically linear     *)
(*                        when f is, and simplifies on application         *)
(*               k \*: f == the function x |-> k *: f x, which is            *)
(*                        canonically linear when f is and simplifies on   *)
(*                        application (this is a shorter alternative to    *)
(*                        *:%R k \o f)                                *)
(*         GRing.in_alg A == the ring morphism that injects R into A, where A *)
(*                        has an lalgType R structure; GRing.in_alg A k    *)
(*                        simplifies to k%:A                           *)
(*               a \*o f == the function x |-> a * f x, canonically linear   *)
(*                        when f is and its codomain is an algType         *)
(*                        and which simplifies on application             *)
(*               a \o* f == the function x |-> f x * a, canonically linear   *)
(*                        when f is and its codomain is an lalgType        *)
(*                        and which simplifies on application             *)
(*               f \* g == the function x |-> f x * g x; f \* g             *)
(*                        simplifies on application                      *)
(* The Lemmas about these structures are contained in both the GRing module   *)
(* and in the submodule GRing.Theory, which can be imported when unqualified  *)
(* access to the theory is needed (GRing.Theory also allows the unqualified   *)
(* use of additive, linear, Linear, etc). The main GRing module should NOT be *)
(* imported.                                                            *)
(*   Notations are defined in scope ring_scope (delimiter %R), except term    *)
(* and formula notations, which are in term_scope (delimiter %T).            *)
(*   This library also extends the conventional suffixes described in library *)
(* ssrbool.v with the following:                                        *)
(*   0 -- ring 0, as in addr0 : x + 0 = x                                *)
(*   1 -- ring 1, as in mulr1 : x * 1 = x                                *)
(*   D -- ring addition, as in linearD : f (u + v) = f u + f v              *)
(*   B -- ring subtraction, as in opprB : - (x - y) = y - x                 *)
(*   M -- ring multiplication, as in invfM : (x * y)^-1 = x^-1 * y^-1         *)
(*  Mn -- ring by nat multiplication, as in raddfMn : f (x *+ n) = f x *+ n   *)
(*   N -- ring opposite, as in mulNr : (- x) * y = - (x * y)                *)
(*   V -- ring inverse, as in mulVr : x^-1 * x = 1                          *)
(*   X -- ring exponentiation, as in rmorphXn : f (x ^+ n) = f x ^+ n         *)
(*   Z -- (left) module scaling, as in linearZ : f (a *: v)  = s *: f v       *)
(* The operator suffixes D, B, M and X are also used for the corresponding    *)
(* operations on nat, as in natrX : (m ^ n)%:R = m%:R ^+ n. For the binary    *)
(* power operator, a trailing "n" suffix is used to indicate the operator     *)
(* suffix applies to the left-hand ring argument, as in                    *)
(*   expr1n : 1 ^+ n = 1 vs. expr1 : x ^+ 1 = x.                          *)
(********************************
```

Comments (Kazuhiko):

- HB means Hierarchy Builder: https://doi.org/10.4230/LIPIcs.FSCD.2020.34

- The header document of each .v file in MathComp is supposed to include the description of any user-facing structure (e.g., rings), data type (e.g., polynomials), and definitions (functions and constants). On the other hand, most theorems and lemmas are not documented.
- There was a proposal to add a docstring feature to Coq:
  https://github.com/coq/coq/wiki/Coq-Call-2021-05-05
  https://github.com/math-comp/hierarchy-builder/issues/230.
  - Without such a feature, syncronizing this database with the latest version of MathComp seems to require a lot of work. In theory, if there is a docstring feature that allows us to embed sufficient metadata in .v files, we should be able to generate this database automatically.
  - This work (MathComp/Wikidata alignment) would be useful to signal the need for a docstring feature to the Coq developers. Some contributors of this work would also be able to contribute to the design of this docstring feature, e.g., a field to write a Wikidata identifier as a part of docstring.
  - Cons: the entire process (adding a docstring feature and then annotating the definitions in MathComp using this feature) may take a long time.