Theory development leading to the Fundamental Theorem of Algebra

Herman Geuvers, Randy Pollack, Freek Wiedijk, Jan Zwanenburg

November 10, 2000

Contents

1 About this document

1.1 Why this document

To have an actual overview of the concrete Coq proof-development of FTA, we need to have a document in which all lemmas and definition can be found, with the actual names (in Coq), together with (syntactic) conventions, design decisions and explanations. We produce this document in LaTeX, to be able to use ordinary mathematical notation for explanation.

This document should always reflect the actual state of affairs of the Coq development. Therefore it is best to extract this LaTeX code form the Coq code. Coq already provides one way of doing so, by the so called 'gallina' files. These remove from a Coq file all proofs, yielding a `.g` file. However, we want more: notably we want to add *mathematical* comments. Hence we introduce a different set-up, where we add comments to the Coq files. A small compiler script then translates the Coq files to LaTeX source code (taking the comments into account, comaparable with compiler-directives in programs). Note that there are two other possibilities: (1) Use LaTeX files with comments that facilitate the extraction of Coq code; (2) Define a new file format from which both the Coq code and the LaTeX code can be extracted. As we are working with Coq most of the time, these two options seem less convenient. So, taking the option of using the Coq files as basic, the comments will have to be written as Coq-comments to be able to use the files as valid Coq proof scripts.

1.2 How to make this document

In the `algebra` main directory, type `maketex` (or `bin/maketex` if you get the reply `maketex: command not found`). This command concatenates the file `body/body_pre.tex`, the tex comments in the Coq script files (see below), and the file `body/body_post.tex` into the file `body/body.tex`. The file `body/body.tex` should not be edited manually! Go to the `body` subdirectory, and latex the file `body.tex`.

1.3 Tex comments in script files

- To put some piece of the script file verbatim in the tex file (e.g. for lemmas and definitions), insert

  ```
  (* Begin_Tex_Verb *)
  ```

  before this piece, and

  ```
  (* End_Tex_Verb *)
  ```

  just after this piece.

- In order to put some prose in the tex file, insert a line

  ```
  (* Tex_Prose
  ```

  then insert the text, and then insert a line

  ```
  *)
  ```

1.4 Guidelines for producing this document

We have made the following LaTeX environments:

- The convention, numconvention environments for unnumbered and numbered conventions, respectively.
- The nameconvention environment.
- The notation, numnotation environments.

- The shortcoming environment.

What *should* be included and how:

- Lemmas: verbatim.
- Definitions: verbatim. (It is not clear how interactively constructed definitions should be included; see below.)
- Variable declarations: typically with the LaTeX convention environment.
- Syntactic definitions: with the LaTeX notation environment.
- Sections: Since every script file corresponds with one section in the LaTeX document, a Coq-section should usually be included as a LaTeX-subsection, and a nested Coq-section should be included as a LaTeX-subsubsection. However, the end of one Coq-section does not always coincide with the start of a next Coq-section (in contrast to LaTeX). In these cases, additional LaTeX ((sub)sub)sections are introduced to avoid confusion (e.g. see `CSetoids.v`).

  If a Coq-section is introduced only for a few definitions/lemmas, it is sometimes simpler not to include this structure in the LaTeX file.
- Coercions: as conventions.
- Comments: usually as LaTeX prose.

What *should not* be included:

- Proofs.
- Tactic definitions.
- Import lists (`Require`).
- Commands for setting implicit arguments on/off.
- Hints.

2 General notes

2.1 Notes about the development

- We would like to avoid giving explicit proof terms in definitions and lemmas as much as possible. (In mathematics, the existence (or construction) of an object may depend on the *truth* (or *provability*) of a statement, but not on its actual proof. In Coq, we cannot always avoid this, in particular, when making definitions.
- In Coq, a definition can be given in two ways. Either it must be completely written out, or it is completely constructed interactively; there is no in-between. The latter option has as disadvantage that we cannot really see how something is defined. (We do not display the proof script, but even a proof script says little about the actual constructed object). The former has as disadvantage that the subterms corresponding with proofs may be hard to write. Currently, we don't have any guideline which of the two options to choose. (One would like to pack all proof information in lemmas, so that each proof occurring in a definition could be denoted by just a variable.) What soemtimes works well for objects that are hard to define is to define the object interactively, then to print it (using the `Print` command) and then to redefine the printed thing (possibly after some editing).
- The word `Axiom` occurring in proof scripts usually means we just did not prove the property yet.

- Many operations `op` have a syntactically sugared version, e.g. for which some arguments are implicit (e.g. defined by the "Syntactic Definition" construction in combination with "?"). This sugared version is always denoted by capitalizing the first letter of `op`.

- In Coq, there are two ways to provide implicit arguments. First the `Implicit Arguments` mechanism. Second, the `Syntactic Definition` mechanism in combination with `?`. We need a guideline when to use which one.

2.2 Notes about Coq (shortcomings)

1. The Coq syntax is not particularly perfect. All kinds of operators can not be written down in the way one would want to. A mature parser with overloading (where the + in $x + y$ depends on the types of x and y) would help.

2. Proof obligations can be very cumbersome. If you write down something like `3^(1-k)`, you have to think about whether it is *allowed* to user a power to the $1 - k$th. That's ok to a certain extent (it's in the mathematics), *but* you will have to add the *reason* why you're allowed to do it as an argument. This makes expressions very heavy.

3. Continuing forward and then 'backwards' constructing what you need can be inefficient, because what you are going to *use* are the things that are already there, instead of *extending* the things that are already there. So, you will take detours, do things in an unnecesarily ugly way and do things in a too restricted way (where you could prove a more general result), just to prevent you from having to add a whole list of elementary lemmas. (You have the feeling that 'that level of generality is not the issue right now'.) Moreover, re-coqing the whole development, taking into account all the new 'basic generalities' will take a lot of time, which discourages you even stronger to prove the add the generalities.

   On the other hand, starting from the basics only works (and even then only to a certain extent) if you know what the basic results are that you will be needing. This is the case for established theories, like groups (but even then you will notice that when you're in Coq, you will need a lot more then you had imagined), but it does not work if you want to set up, say, a small library for dealing with finite sequences: you will reamin working on the basics for a long time, to find out later that you needed different definitions and properties about them.

4. 'Efficiency' makes that you will write down the development less clearly then possible and desirable. For example Freek uses `Half` instead of `One[/]Two` and then chooses a derivation that uses only multiplication, because division is so complicated. That is not what you want, because the development gets less perspicuous.

General namegiving conventions:

- `imp` stands for implies.

- "left" and "right" are abbreviated to `lft` and `rht`, respectively.

preserve	$\mapsto$	`pres`
predicate	$\mapsto$	`pred`
relation	$\mapsto$	`rel`
respects	$\mapsto$	`resp`

3 Formalization of the algebraic hierarchy in Coq

We extend structures according to the following abstract pattern (the reader might find it easier to consider a specific example, e.g. see Section 9).

Let us say we extend structure A (with name $AName$) to structure B (with name $BName$, abbreviated to b), with operations $opName_1 : \sigma_1, \ldots, opName_n : \sigma_n$, and with properties $propName_1 : P_1, \ldots, propName_m : P_m$. We do this as follows.

First we define a record is_*Bname* in which all properties are grouped:

```
Record is_BName [A:AName; opName₁ : σ₁; ... ; opNameₙ : σₙ] : Prop :=
  {ax_propName₁ : P₁;
   ⋮
   ax_propNameₘ : Pₘ
  }.
```

Then we define the actual structure B:

```
Record BName : Type :=
  {b_crr :> AName;
   b_opName₁ : σ₁;
   ⋮
   b_opNameₙ : σₙ;
   b_proof : (is_BName b_crr b_opName₁ ...b_opNameₙ)
  }.
```

And then we prove the basic lemmas directly corresponding to the properties (maybe after introducing some notational conveniences).

```
Section BName_axioms.
Variable B : BName.
Lemma BName_is_BName : is_BName B b_opName₁...b_opNameₙ.
Lemma propName₁ : P₁.
⋮
Lemma propNameₘ : Pₘ.
End BName_axioms.
```

The reason for introducing the basic lemmas is stated in note 3 below.

Notes.

1. The properties P_1 through P_m are always written using auxiliary definitions, often defined directly before is_*Bname*. E.g. in the definition of monoids (Section 9) P_1 is `is_rht_unit G Csg_op Zero`, and `is_rht_unit` is defined directly before `is_Monoid`.

2. If there is just one property (i.e. $m = 1$), then we do not use the record mechanism to define is_*Bname*, but use an ordinary definition instead.

3. Each field in the record is_*Bname* has an associated function ax_*propName*$_i$, with the following type:

   ```
   (A:AName; opName₁ : σ₁; ... ; opNameₙ : σₙ) is_BName -> Pᵢ.
   ```

   This type is very cumbersome to use. Therefore we introduced the basic lemmas. The natural names $propName_1 \ldots propName_m$ are reserved for these basic lemmas, and to avoid name clashes the field names are prefixed with "ax_".

4. The operations may be either functions that deliver a mathematical object (e.g. addition), or predicates (e.g. less than).

Alternatives and design decisions

- Alternative: more structuring, with sig_*bName*.
- Alternative: less structuring, without is_*bName*.

4 New tactics

There are two groups of new tactics, each treated in a separate subsection. To understand their description the reader should have basic knowledge of our implementation of the algebraic hierarchy.

4.1 Proving equalities

There are two new tactics that prove `CSetoid` equalities, i.e. goals of the form a `[=]` b. First, we have the `Algebra` tactic, which is implemented with Coq's Auto mechanism. It can be used to prove simple equalities, typically only those that are just one application of a lemma.

Second, we have the `Rational` tactic. This tactic works only from `CRings` onwards, so for `CRings`, `CFields`, `COrdFields` and so on, but not for `CGroups`, `CMonoids`, etc. The tactic `Rational` is much more clever than `Algebra`: it rewrites both sides of the equality to some canonical form and compares these. It automatically uses commutativity, associativity and so on.

4.2 Using equalities

Consider the following situation: the goal is of the form a `[#]` b and the user wants to prove this by showing a' `[#]` b, since a `[=]` a'. In this situation, the user can type `Step_ap_lft` a'. This generates the goal a' `[#]` b, and a goal a `[=]` a', which the `Algebra` tactic tries to solve.

Similarly, the right hand side b of an apartness can be replaced by an equal b' by typing `Step_ap_rht` b'. Similar to `Step_ap_lft` and `Step_ap_rht` are the `Step_Rat_ap_lft` and `Step_Rat_ap_rht` tactics, which use `Rational` instead of `Algebra` to prove the equality.

These four tactic are for the apartness relation. Similar tactics exist for `[=]`, `[:<]`, `[:<=]`, and `AbsSmall`, as shown below.

Relation	equality proved with `Algebra`	equality proved with `Rational`
`[#]`	`Step_ap_lft`	`Step_Rat_ap_lft`
	`Step_ap_rht`	`Step_Rat_ap_rht`
`[=]`	`Step_lft` (or `Step`)	`Step_Rat_lft`
	`Step_rht` (or `Stepr`)	`Step_Rat_rht`
`[:<]`	`Step_less_lft`	`Step_Rat_less_lft`
	`Step_less_rht`	`Step_Rat_less_rht`
`[:<=]`	`Step_leEq_lft`	`Step_Rat_leEq_lft`
	`Step_leEq_rht`	`Step_Rat_leEq_rht`
`AbsSmall`	`Step_AbsSmall_lft`	`Step_Rat_AbsSmall_lft`
	`Step_AbsSmall_rht`	`Step_Rat_AbsSmall_rht`

Finally, the `Step_final` c tactic tries to prove the equality a `[=]` b by applying `Algebra` to a `[=]` c and c `[=]` b.

5 Basics

This is random stuff that should be in the Coq basic library. This is the axiom of choice. It currently is the one 'real' axiom. We need it because we write our logic in Prop.

```
Axiom ex_informative : (A:Set;P:A->Prop)(Ex [x:A](P x))->{x:A|(P x)}.
```

The next is needed to define the Max function. We have to be able to go from a 'non-informative' disjunction to an 'informative' one

```
Lemma or_informative : (A,B:Prop)A\/B->{A}+{B}.

Lemma lt_z_two: (lt O (2)).

Lemma le_pred : (n,m:nat)(le n m)->(le (pred n)(pred m)).

Lemma lt_mult_right :
  (x,y,z:nat) (lt x y) -> (lt (O) z) -> (lt (mult x z) (mult y z)).

Lemma le_mult_left :
  (x,y,z:nat) (le x y) -> (le (mult z x) (mult z y)).
```

Faculty function. Does not exist in Arith. (?) Needed for special operations on polynomials.

```
Fixpoint fac [n:nat] :nat :=
Cases n of
   O  => (S O)
   | (S m) => (mult (S m)(fac m))
end.

Lemma nat_fac_gtzero:(n:nat)(lt O (fac n)).

Lemma not_eq_sym: (A:Set; a,b:A) ~a=b -> ~b=a.

Lemma contrapos: (A,B:Prop)(A->B)->(X:Prop)(B->X)->A->X.

Lemma deMorgan_or_and: (A,B,X:Prop)((A\/B)->X) -> (A->X)/\(B->X).

Lemma deMorgan_and_or: (A,B,X:Prop)(A->X)/\(B->X) -> (A\/B->X).

Lemma deMorgan_ex_all:
  (A:Set)(P:A->Prop)(X:Prop)((Ex P)->X) -> (a:A)(P a)->X.

Lemma deMorgan_all_ex:
  (A:Set)(P:A->Prop)(X:Prop)((a:A)(P a)->X) -> (Ex P)->X.

Inductive Ex2 [A,B:Set; P:A->B->Prop] : Prop :=
   Ex2_intro: (a:A;b:B)(P a b) -> (Ex2 P).
```

Three lemmas for proving properties about definitions made with case distinction to a sumbool, i.e. `{A} + {B}`.

```
Lemma sumbool_rec_or : (A,B:Prop)(S:Set)(l,r:S)(s:{A}+{B})
                  (sumbool_rec A B [_:{A}+{B}]S [x:A]l [x:B]r s) = l \/
                  (sumbool_rec A B [_:{A}+{B}]S [x:A]l [x:B]r s) = r.

Lemma not_r_sumbool_rec : (A,B:Prop)(S:Set)(l,r:S)~B->(H:{A}+{B})
                  (sumbool_rec A B [_:{A}+{B}]S [x:A]l [x:B]r H) = l.

Lemma not_l_sumbool_rec : (A,B:Prop)(S:Set)(l,r:S)~A->(H:{A}+{B})
                          (sumbool_rec A B [_:{A}+{B}]S [x:A]l [x:B]r H) = r.

Definition max : nat->nat->nat.

Lemma le_max1 : (m,n:nat)(le m (max m n)).

Lemma le_max2 : (m,n:nat)(le n (max m n)).
```

5.1 Some results about Z

We consider the injection `inject_nat` from `nat` to !Z! as a coercion.

```
Lemma POS_anti_convert :
              (n:nat)`(S n) = (POS (anti_convert n))`.
Induction n.
Simpl.
Reflexivity.
Intros.
Simpl.
Reflexivity.
Qed.


Lemma NEG_anti_convert :
              (n:nat)`-(S n) = (NEG (anti_convert n))`.


Lemma lt_O_positive_to_nat :
                     (p:positive)(m:nat)(lt O m)->(lt O (positive_to_nat p m)).


Lemma anti_convert_pred_convert :
                          (p:positive)(p = (anti_convert (pred (convert p)))).


Lemma p_is_some_anti_convert : (p:positive)(EX n:nat | p = (anti_convert n)).


Lemma convert_is_POS :
              (p:positive)`(convert p) = (POS p)`.


Lemma min_convert_is_NEG :
              (p:positive)`-(convert p) = (NEG p)`.


Lemma Z_exh : (z:Z)(EX n:nat | z=n) \/ (EX n:nat | z=`-n`).


Lemma nats_Z_ind :
                 (P:Z->Prop)((n:nat)(P n)) -> ((n:nat)(P `-n`)) -> (z:Z)(P z).


Lemma pred_succ_Z_ind : (P:Z->Prop)(P `0`) ->
                                   ((n:Z)(P n)->(P `n+1`)) ->
                                   ((n:Z)(P n)->(P `n-1`)) ->
                                   (z:Z)(P z).


Lemma Zmult_minus_distr_r : (n,m,p:Z)`p*(n-m) = p*n-p*m`.


Lemma Zodd_Zeven_min1 : (x:Z)(Zodd x) -> (Zeven `x-1`).
```

```
Definition caseZ_diff := [A:Set][z:Z][f:nat->nat->A]
  Cases z of
    ZERO => (f O O)
  | (POS m) => (f (convert m) O)
  | (NEG m) => (f O (convert m))
  end.
```

```
Lemma caseZ_diff_O : (A:Set)(f:nat->nat->A)(caseZ_diff `0` f) = (f O O).
```

```
Lemma caseZ_diff_Pos : (A:Set)(f:nat->nat->A)(n:nat)
                       (caseZ_diff `n` f) = (f n O).
```

```
Lemma caseZ_diff_Neg : (A:Set)(f:nat->nat->A)(n:nat)
                       (caseZ_diff `-n` f) = (f O n).
```

```
Lemma proper_caseZ_diff : (A:Set)(f:nat->nat->A)
              ((m,n,p,q:nat)(plus m q) = (plus n p) -> (f m n) = (f p q)) ->
              (m,n:nat)(caseZ_diff `m-n` f) = (f m n).
```

```
Lemma diff_Z_ind : (P:Z->Prop)((m,n:nat)(P `m-n`)) -> ((z:Z)(P z)).
```

6 Setoids

Definition of a constructive setoid, i.e. a set with an equivalence relation and an apartness relation.

6.1 Relations necessary for Setoids

Convention: Let `A : Set`.

```
Definition irreflexive [R:(relation A)] : Prop :=
        (x:A)(not(R x x)).
```

```
Definition cotransitive [R:(relation A)] : Prop :=
        (x,y:A)(R x y) -> (z:A)(R x z) \/ (R z y).
```

```
Definition tight_apart [eq,R:(relation A)]: Prop :=
        (x,y:A)(not(R x y)) <-> (eq x y).
```

```
Definition antisymmetric [R:(relation A)] : Prop :=
        (x,y: A)(R x y) -> (not(R y x)).
```

6.2 Definition of Setoid

```
Record is_CSetoid [A:Set; eq,ap:(relation A)] : Prop :=
  { ax_ap_irreflexive  : (irreflexive ap);
    ax_ap_symmetric    : (Symmetric ap);
    ax_ap_cotransitive : (cotransitive ap);
    ax_ap_tight        : (tight_apart eq ap)
  }.
```

```
Record CSetoid : Type :=
  { cs_crr   :> Set;
    cs_eq    :  (relation cs_crr);
    cs_ap    :  (relation cs_crr);
    cs_proof :  (is_CSetoid cs_crr cs_eq cs_ap)
  }.
```

Notation:

- `cs_eq ? x y` is denoted by `x [=] y`.
- `cs_ap ? x y` is denoted by `x [#] y`.

```
Definition cs_neq [S: CSetoid] : (relation S) := [x,y:S](not(x [=] y)).
```

Notation: `cs_neq ? x y` is denoted by `x [~=] y`.

Name convention: In the names of lemmas, we refer to `[=]` by `eq`, `[~=]` by `neq`, and `[#]` by `ap`.

6.3 Setoid axioms

We want concrete lemmas that state the axiomatic properties of a setoid.

Convention: Let `S` be a setoid.

```
Lemma CSetoid_is_CSetoid : (is_CSetoid S (cs_eq S) (cs_ap S)).
```

```
Lemma ap_irreflexive : (irreflexive (cs_ap S)).
```

```
Lemma ap_symmetric : (Symmetric (cs_ap S)).
```

```
Lemma ap_cotransitive : (cotransitive (cs_ap S)).
```

```
Lemma ap_tight : (tight_apart (cs_eq S) (cs_ap S)).
```

6.4 Setoid basics

Convention: Let `S` be a setoid.

In 'there exists a *unique* $a : S$ such that ...', we now mean unique with respect to the setoid equality. We use `ex_unq` to denote unique existence

```
Definition ex_unq := [P:S->Prop]
(EX x:S | (P x) /\ (y:S)(P y)-> x[=]y).

Lemma eq_reflexive: (Reflexive (cs_eq S)).

Lemma eq_symmetric: (Symmetric (cs_eq S)).

Lemma eq_transitive: (Transitive (cs_eq S)).
```

Coq shortcoming: The lemma `eq_reflexive` above is convertible to `eq_reflexive_unfolded` below. We need the second version too, because the first cannot be applied when an instance of reflexivity is needed. ("I have complained bitterly about this." RP)

Name convention: If lemma a is just an unfolding of lemma b, the name of a is the name b with the suffix "`_unfolded`".

```
Lemma eq_reflexive_unfolded : (x:S)(x [=] x).

Lemma eq_symmetric_unfolded : (x,y:S)(x [=] y)->(y [=] x).

Lemma eq_transitive_unfolded : (x,y,z:S)(x [=] y)->(y [=] z)->(x [=] z).

Lemma ap_irreflexive_unfolded : (x:S)(~(x[#]x)).

Lemma ap_cotransitive_unfolded : (x,y:S)((x[#]y)->
                                        (z:S)((x[#]z) \/ (z[#]y))).

Lemma ap_symmetric_unfolded : (x,y:S)((x[#]y) -> (y[#]x)).
```

Coq shortcoming: We would like to write

```
Lemma eq_equiv_not_ap : (x,y:S)(x [=] y) <-> ~(x [#] y).
```

In Coq, however, this lemma cannot be easily applied. Therefore we have to split the lemma into the following two lemmas `eq_imp_not_ap` and `not_ap_imp_eq`.

```
Lemma eq_imp_not_ap : (x,y:S)(x [=] y) -> ~(x [#] y).

Lemma not_ap_imp_eq : (x,y:S)~(x [#] y) -> (x [=] y).

Lemma neq_imp_notnot_ap : (x,y:S)(x [~=] y) -> ~~(x [#] y).

Lemma notnot_ap_imp_neq :  (x,y:S)~~(x [#] y) -> (x [~=] y).

Lemma ap_imp_neq: (x,y:S)(x [#] y)->(x [~=] y).

Lemma not_neq_imp_eq : (x,y:S)~(x[~=]y) -> x [=] y.

Lemma eq_imp_not_neq : (x,y:S)(x [=] y) -> ~(x[~=]y).
```

Notation: `Ex_unq` stands for `ex_unq ?`

6.5 Relations and predicates

Here we define the notions of well-definedness and strong extensionality on predicates and relations.

Convention: Let `S` be a setoid.

Name convention:

- "well-defined" is abbreviated to `well_def` (or `wd`?).
- "strongly extensional" is abbreviated to `strong_ext` (or `strext`?).

6.5.1 Predicates

Convention: Let `P` be a predicate on (the carrier of) `S`.

```
Definition pred_strong_ext: Prop := (x,y: S)(P x) -> (P y) \/ (x [#] y).
Definition pred_well_def: Prop :=   (x,y: S)(P x) -> (x [=] y) -> (P y).
```

6.5.2 Definition of a setoid predicate

```
Record CSetoid_predicate : Type :=
  { csp_pred   :> S -> Prop;
    csp_wd     :  (pred_well_def csp_pred);
    csp_strext :  (pred_strong_ext csp_pred)
  }.
```

6.5.3 Relation

Convention: Let `R` be a relation on (the carrier of) `S`.

```
Definition rel_well_def_rht: Prop :=
  (x,y,z: S)(R x y) -> (y [=] z) -> (R x z).
Definition rel_well_def_lft: Prop :=
  (x,y,z: S)(R x y) -> (x [=] z) -> (R z y).
Definition rel_strong_ext : Prop :=
  (x1,x2,y1,y2 : S)(R x1 y1) -> (R x2 y2) \/ (x1 [#] x2) \/ (y1 [#] y2).
```

```
Definition rel_strong_ext_lft : Prop :=
  (x1,x2,y : S)(R x1 y) -> (R x2 y) \/ (x1 [#] x2).
Definition rel_strong_ext_rht : Prop :=
  (x,y1,y2 : S)(R x y1) -> (R x y2) \/ (y1 [#] y2).
```

```
Lemma rel_strext_imp_lftarg: rel_strong_ext -> rel_strong_ext_lft.
```

```
Lemma rel_strext_imp_rhtarg: rel_strong_ext -> rel_strong_ext_rht.
```

```
Lemma rel_strextarg_imp_strext:
   rel_strong_ext_rht -> rel_strong_ext_lft -> rel_strong_ext.
```

6.5.4 Definition of a setoid relation

The type of relations over a setoid.

```
Record CSetoid_relation : Type :=
  { csr_rel    :> S -> S -> Prop;
    csr_wdr    : (rel_well_def_rht csr_rel);
    csr_wdl    : (rel_well_def_lft csr_rel);
    csr_strext : (rel_strong_ext csr_rel)
  }.
```

```
Lemma ap_well_def_rht: (rel_well_def_rht (cs_ap S)).
```

```
Lemma ap_well_def_lft: (rel_well_def_lft (cs_ap S)).
```

```
Lemma ap_well_def_rht_unfolded:
   (x,y,z:S)(x [#] y) -> (y [=] z) -> (x [#] z).
```

```
Lemma ap_well_def_lft_unfolded:
   (x,y,z:S)(x [#] y) -> (x [=] z) -> (z [#] y).
```

```
Lemma ap_strong_ext: (rel_strong_ext (cs_ap S)).
```

```
Definition predS_well_def [P:S->Set]: Set := (x,y:S)(P x) -> (x [=] y) -> (P y).
```

6.6 Functions between setoids

Such functions must preserve the setoid equality and be strongly extensional w.r.t. the apartness, i.e. e.g. if `f(x,y) apart f(x1,y1)`, then `x apart x1 \/ y apart y1`. For every arity this has to be defined separately.

Convention: Let `S1`, `S2` and `S3` be setoids.

First we consider unary functions. In the following two definitions, `f` is a function from (the carrier of) `S1` to (the carrier of) `S2`.

```
Definition fun_well_def : Prop := (x, y: S1)(x [=] y) -> (f x) [=] (f y).
Definition fun_strong_ext : Prop := (x, y: S1)((f x) [#] (f y)) -> x [#] y.
```

```
Lemma fun_strong_ext_imp_well_def : fun_strong_ext -> fun_well_def.
```

```
Record CSetoid_fun : Set :=
  { csf_fun    :> S1 -> S2;
    csf_wd     :  (fun_well_def csf_fun);
    csf_strext :  (fun_strong_ext csf_fun)
  }.
```

```
Definition Const_CSetoid_fun : S2 -> CSetoid_fun.
```

Now we consider binary functions. In the following two definitions, `f` is a function from `S1` and `S2` to `S3`.

```
Definition bin_fun_well_def : Prop :=
  (x1, x2: S1)(y1, y2: S2)
   (x1 [=] x2) -> (y1 [=] y2) -> (f x1 y1) [=] (f x2 y2).

Definition bin_fun_strong_ext : Prop :=
  (x1, x2: S1)(y1, y2: S2)
   ((f x1 y1) [#] (f x2 y2)) -> (x1 [#] x2) \/ (y1 [#] y2).


Lemma bin_fun_strong_ext_imp_well_def : bin_fun_strong_ext -> bin_fun_well_def.


Record CSetoid_bin_fun : Set :=
  { csbf_fun    :> S1 -> S2 -> S3;
    csbf_wd     :  (bin_fun_well_def csbf_fun);
    csbf_strext :  (bin_fun_strong_ext csbf_fun)
  }.
```

Let S1, S2 and S3 be setoids, f a setoid function from S1 to S2, and g from S2 to S3 in the following definition of composition.

```
Definition compose_CSetoid_fun: (CSetoid_fun S1 S3).
```

6.7 The unary and binary (inner) operations on a csetoid.

An operation is a function with domain(s) and co-domain equal.

Name convention: The word "unary operation" is abbreviated to un_op; "binary operation" is abbreviated to bin_op.

Convention: Let S be a setoid.

Properties of binary operations

```
Definition commutes [f:S->S->S] : Prop := (x,y: S)(f x y) [=] (f y x).
Definition associative [f:S->S->S]: Prop :=
    (x,y,z:S)(f x (f y z)) [=] (f (f x y) z).
```

Well-defined unary operations on a setoid.

```
Definition un_op_well_def := (fun_well_def S S).
Definition un_op_strong_ext := (fun_strong_ext S S).


Definition CSetoid_un_op : Set := (CSetoid_fun S S).
Definition Build_CSetoid_un_op := (Build_CSetoid_fun S S).


Lemma id_strext : (un_op_strong_ext [x:S]x).


Lemma id_pres_eq : (un_op_well_def [x:S]x).


Definition id_un_op := (Build_CSetoid_un_op [x:S]x id_pres_eq id_strext).


Definition un_op_strext := (csf_strext S S).
```

```
Lemma un_op_wd_unfolded:
  (op:CSetoid_un_op)(x,y:S)(x [=] y) -> (op x) [=] (op y).


Lemma un_op_strext_unfolded:
  (op:CSetoid_un_op)(x,y:S)((op x) [#] (op y)) -> x [#] y.
```

Well-defined binary operations on a setoid.

```
Definition bin_op_well_def := (bin_fun_well_def S S S).
Definition bin_op_strong_ext := (bin_fun_strong_ext S S S).


Definition CSetoid_bin_op : Set := (CSetoid_bin_fun S S S).
Definition Build_CSetoid_bin_op := (Build_CSetoid_bin_fun S S S).


Definition bin_op_wd := (csbf_wd S S S).
Definition bin_op_strext := (csbf_strext S S S).


Lemma bin_op_wd_unfolded:
    (op:CSetoid_bin_op)(x1, x2, y1, y2: S)
      (x1 [=] x2) -> (y1 [=] y2) -> (op x1 y1) [=] (op x2 y2).


Lemma bin_op_strext_unfolded:
    (op:CSetoid_bin_op)(x1, x2, y1, y2: S)
      ((op x1 y1) [#] (op x2 y2)) -> (x1 [#] x2) \/ (y1 [#] y2).


Lemma bin_op_is_wd_un_op_lft:
    (op:CSetoid_bin_op)(c: S)(un_op_well_def [x:S](op x c)).


Lemma bin_op_is_wd_un_op_rht:
    (op:CSetoid_bin_op)(c: S)(un_op_well_def [x:S](op c x)).


Lemma bin_op_is_strext_un_op_lft:
    (op:CSetoid_bin_op)(c: S)(un_op_strong_ext [x:S](op x c)).


Lemma bin_op_is_strext_un_op_rht:
    (op:CSetoid_bin_op)(c: S)(un_op_strong_ext [x:S](op c x)).


Definition bin_op2un_op_rht : CSetoid_bin_op->(S)->CSetoid_un_op :=
[op:CSetoid_bin_op; c:(S)]
 (Build_CSetoid_un_op [x:(S)](op c x)
   (bin_op_is_wd_un_op_rht op c) (bin_op_is_strext_un_op_rht op c)).


Definition bin_op2un_op_lft : CSetoid_bin_op->(S)->CSetoid_un_op :=
[op:CSetoid_bin_op; c:(S)]
 (Build_CSetoid_un_op [x:(S)](op x c)
   (bin_op_is_wd_un_op_lft op c) (bin_op_is_strext_un_op_lft op c)).
```

6.8 The binary outer operations on a csetoid.

Convention: Let S1 and S2 be setoids.

Well-defined outer operations on a setoid.

```
Definition outer_op_well_def := (bin_fun_well_def S1 S2 S2).
Definition outer_op_strong_ext := (bin_fun_strong_ext S1 S2 S2).

Definition CSetoid_outer_op : Set := (CSetoid_bin_fun S1 S2 S2).
Definition Build_CSetoid_outer_op := (Build_CSetoid_bin_fun S1 S2 S2).
Definition csoo_wd := (csbf_wd S1 S2 S2).
Definition csoo_strext := (csbf_strext S1 S2 S2).

Lemma csoo_wd_unfolded:
    (op:CSetoid_outer_op)(x1, x2: S1)(y1, y2: S2)
      (x1 [=] x2) -> (y1 [=] y2) -> (op x1 y1) [=] (op x2 y2).
```

6.9 Combining operations

Convention: Let S1, S2 and S3 be setoids.

In the following definition, we assume f is a setoid function from S1 to S2, and op is an unary operation on S2. opOnFun is the composition op after f.

```
Definition opOnFun: (CSetoid_fun S1 S2).
```

6.10 Subsetoids

Convention: Let S be a setoid, and P a predicate on the carrier of S.

```
Record subcsetoid_crr : Set :=
  { scs_elem :> S;
    scs_prf : (P scs_elem)
  }.

Definition restrict_relation [R:(relation S)]: (relation subcsetoid_crr) :=
  [a, b: subcsetoid_crr]
  Cases a b of
   (Build_subcsetoid_crr x _) (Build_subcsetoid_crr y _) => (R x y)
  end.

Definition subcsetoid_eq: (relation subcsetoid_crr) :=
  (restrict_relation (cs_eq S)).

Definition subcsetoid_ap: (relation subcsetoid_crr) :=
  (restrict_relation (cs_ap S)).

Remark subcsetoid_equiv : (equiv ? subcsetoid_eq).

Lemma subcsetoid_is_CSetoid : (is_CSetoid ? subcsetoid_eq subcsetoid_ap).

Definition Build_SubCSetoid : CSetoid :=
  (Build_CSetoid subcsetoid_crr subcsetoid_eq subcsetoid_ap
            subcsetoid_is_CSetoid).
```

6.10.1 Subsetoid unary operations

Convention: Let `f` be a unary setoid operation on `S`.

```
Definition un_op_pres_pred : Prop := (x:S)(P x) -> (P (f x)).
```

Convention: Assume `pr : un_op_pres_pred`.

```
Definition restr_un_op : subcsetoid_crr -> subcsetoid_crr :=
  [a: subcsetoid_crr]
  Cases a of (Build_subcsetoid_crr x p) =>
             (Build_subcsetoid_crr (f x) (pr x p))
  end.
```

```
Lemma restr_un_op_well_def: (un_op_well_def Build_SubCSetoid restr_un_op).
```

```
Lemma restr_un_op_strong_ext: (un_op_strong_ext Build_SubCSetoid restr_un_op).
```

```
Definition Build_SubCSetoid_un_op: (CSetoid_un_op Build_SubCSetoid) :=
  (Build_CSetoid_un_op Build_SubCSetoid restr_un_op restr_un_op_well_def
                                                   restr_un_op_strong_ext).
```

6.10.2 Subsetoid binary operations

Convention: Let `f` be a binary setoid operation on `S`.

```
Definition bin_op_pres_pred : Prop := (x,y:S)(P x) -> (P y) -> (P (f x y)).
```

Convention: Assume `un_op_pres_pred`.

```
Definition restr_bin_op [a,b:subcsetoid_crr]: subcsetoid_crr :=
  Cases a b of (Build_subcsetoid_crr x p) (Build_subcsetoid_crr y q) =>
                 (Build_subcsetoid_crr (f x y) (pr x y p q))
  end.
```

```
Lemma restr_bin_op_well_def: (bin_op_well_def Build_SubCSetoid restr_bin_op).
```

```
Lemma restr_bin_op_strong_ext:
                         (bin_op_strong_ext Build_SubCSetoid restr_bin_op).
```

```
Definition Build_SubCSetoid_bin_op : (CSetoid_bin_op Build_SubCSetoid) :=
  (Build_CSetoid_bin_op Build_SubCSetoid restr_bin_op restr_bin_op_well_def
                                                 restr_bin_op_strong_ext).
```

```
Lemma restr_f_assoc: (Associative f) ->
                     (Associative (Build_SubCSetoid_bin_op)).
```

6.10.3 Miscalleneous

```
Lemma proper_caseZ_diff_CS : (S:CSetoid)(f:nat->nat->S)
              ((m,n,p,q:nat)(plus m q) = (plus n p) -> (f m n) [=] (f p q)) ->
              (m,n:nat)(caseZ_diff `m-n` f) [=] (f m n).
```

7 Vector Basics

7.1 Some List Basics

Convention: Let `A` be a Set and `X` be a Setoid.

```
Fixpoint NotIn  [x:X;l:(list X)] : Prop :=
      Cases l of
nil => True
| (cons y k) => (x[#]y)/\(NotIn x k) end.
```

```
Fixpoint distinct [l:(list X)] : Prop :=
Cases l of nil => True
| (cons x k) => (NotIn x k)/\(distinct k)
end.
```

```
Fixpoint take [i:nat;l:(list A)] : (list A) :=
Cases l of
  nil  => (nil A)
  | (cons x k) => (Cases i of
O => (nil A)
| (S j) => (cons x (take j k))
   end)
end.
```

```
Fixpoint drop [i:nat;l:(list A)] : (list A) :=
Cases l of
  nil  => (nil A)
  | (cons x k) => (Cases i of
O => l
| (S j) =>  (drop j k)
   end)
end.
```

Some preminiaries about list insertion. We insert an element at the end if the index is too high, so `ins_list`$ri(l_0,\ldots,l_{n-1}) = (l_0,\ldots,l_{i-1}rl_i,\ldots,l_{n-1})$

```
Fixpoint ins_list [r:A;i:nat;l:(list A)] : (list A) :=
Cases l of
  nil  => (cons r (nil A))
  | (cons x k) => (Cases i of
O => (cons r (cons x k))
| (S j) => (cons x (ins_list r j k))
   end)
end.
```

```
Fixpoint ins_end [r:A;l:(list A)] : (list A) :=
Cases l of
  nil  => (cons r (nil A))
  | (cons x k) => (cons x (ins_end r k))
end.
```

```
Lemma ins_lth: (r:A; i:nat; l: (list A))
((length (ins_list r i l)) = (S(length l))).
```

```
Lemma nthd_inslist_comm : (i:nat; l:(list A);r:A)
((nth i (ins_list r i l)r) = r).
```

Taking the ith element of a list l, given that l has length n and $i < n$.

```
Fixpoint nth_list [l:(list A);n:nat] :
                              (i:nat)(lt i n)->(length l)=n ->A :=
<[n:nat;l:(list A)](i:nat)(lt i n)->((length l)=n)->A>
Cases n l of
   O   _           => [i:nat;H:(lt i O);Hl :(length _)=O]
                                (False_rec A (lt_n_O i H))
|(S n0) nil        => [i:nat;H:(lt i (S n0));Hl :(length (nil A))=(S n0)]
                                (False_rec A (O_S n0 Hl))
|(S n0)(cons a l0) => [i:nat;H:(lt i (S n0));Hl :(length  (cons a l0))=(S n0)]
(* Distiguish (lt (S i) (S n0)) \/ ((S i)=(S n0)) *)
     (Cases (le_lt_eq_dec (S i) (S n0) H) of
(* x: (lt (S i) (S n0)) *)
     (left x) => (nth_list (lt_S_n i n0 x) (eq_add_S (length l0) n0 Hl))
(* x : ((S i)=(S n0)) *)
     | (right x) => a
     end)
end.
```

8 Semi Groups

8.1 Definition of the notion semi group

```
Definition is_CSemi_grp [A:CSetoid; unit: A; op:(CSetoid_bin_op A)] :=
  (Associative op).
```

```
Record CSemi_grp : Type :=
  { csg_crr   :> CSetoid;
    csg_unit  :  csg_crr;                      (* non-empty *)
    csg_op    :  (CSetoid_bin_op csg_crr);
    csg_proof :  (is_CSemi_grp csg_crr csg_unit csg_op)
  }.
```

Notation: `csg_unit ?` is denoted with `Zero`.

Notation: `csg_op ? x y` is denoted with `x [+] y`.

Name convention: In the *names* of lemmas, we will denote `Zero` with `zero`, and `[+]` with `plus`. We denote `[#] Zero` in the names of lemmas by `ap_zero` (and not, e.g. `nonzero`).

8.2 Semi group axioms

The axiomatic properties of a semi group.

Convention: Let `G` be a semi-group.

```
Lemma CSemi_grp_is_CSemi_grp : (is_CSemi_grp G Zero Csg_op).
```

```
Lemma plus_assoc : (Associative (csg_op G)).
```

8.3 Semi group basics

Convention: Let `G` be a semi-group.

```
Lemma plus_assoc_unfolded : (G:CSemi_grp)(x,y,z:G)
                            (x [+] (y [+] z)) [=] ((x [+] y) [+] z).
```

The predicate "non-zero" is defined as follows. In lemmas we will continue to write `x [#] Zero`, rather than `nonZeroP x`, but the predicate is useful for some high-level definitions, e.g. for the setoid of non-zeros.

```
Definition nonZeroP [x:G] : Prop := (x [#] Zero).
```

8.4 Sub-semi-groups

Convention: Let `csg` a semi-group and `P` a predicate on the semi-group with `P Zero` and which is preserved by `[+]`.

```
Local subcrr : CSetoid := (Build_SubCSetoid ? P).
Definition Build_SubCSemi_grp : CSemi_grp :=
  (Build_CSemi_grp subcrr
                   (Build_subcsetoid_crr ??? Punit)
                   (Build_SubCSetoid_bin_op ??? op_pres_P)
                   (restr_f_assoc ??? op_pres_P (plus_assoc csg))).
```

9 Monoids

We deal only with commutative algebraic structures.

9.1 Definition of monoids

```
Definition is_rht_unit
 [S: CSetoid; op: (CSetoid_bin_op S); one: S] : Prop :=
    (x: S)((op x one) [=] x).
```

```
Definition is_lft_unit
 [S: CSetoid; op: (CSetoid_bin_op S); one: S] : Prop :=
    (x: S)((op one x) [=] x).
```

```
Record is_CMonoid [G: CSemi_grp]: Prop :=
  { runit : (is_rht_unit G Csg_op Zero);
    commut: (commutes G Csg_op)
  }.
```

```
Record CMonoid : Type :=
  { cm_crr   :> CSemi_grp;
    cm_proof :  (is_CMonoid cm_crr)
  }.
```

9.2 Monoid axioms

Convention: Let `M` be a monoid.

```
Lemma CMonoid_is_CMonoid : (is_CMonoid M).
```

```
Lemma cm_rht_unit: (Is_rht_unit Csg_op Zero::M).
```

```
Lemma cm_commutes: (Commutes (csg_op M)).
```

9.3 Monoid basics

Convention: Let `G` be a monoid.

```
Lemma cm_commutes_unfolded: (x, y: M)((x [+] y) [=] (y [+] x)).
```

```
Lemma cm_rht_unit_unfolded : (x:M)((x [+] Zero) [=] x).
```

```
Lemma lftUnit_iff_rhtUnit:
  (x:M)(Is_lft_unit Csg_op x) <-> (Is_rht_unit Csg_op x).
```

```
Lemma cm_lft_unit: (Is_lft_unit Csg_op Zero::M).
```

```
Lemma cm_lft_unit_unfolded: (x:M)((Zero [+] x) [=] x).
```

```
Lemma cm_unit_unique_lft:
    (x:M)(Is_lft_unit Csg_op x)->(x [=] Zero).
```

```
Lemma cm_unit_unique_rht:
    (x:M)(Is_rht_unit Csg_op x)->(x [=] Zero).
```

TO DO: Say something about the reason for the following lemma.

```
Lemma is_CMonoid_proof_irr :
  (S:CSetoid)(one:S)(mult:(CSetoid_bin_op S))(p,q:(Associative mult))
                (is_CMonoid (Build_CSemi_grp S one mult p)) ->
                (is_CMonoid (Build_CSemi_grp S one mult q)).
```

9.4 Submonoids

Convention: Let `P` a predicate on `M` with `P Zero` and which is preserved by `[+]`.

```
Local subcrr : CSemi_grp := (Build_SubCSemi_grp ?? Punit op_pres_P).
```

```
Lemma ismon_scrr: (is_CMonoid subcrr).
```

```
Definition Build_SubCMonoid : CMonoid := (Build_CMonoid subcrr ismon_scrr).
```

10 Groups

10.1 Definition of the notion of Group

```
Definition is_inverse
 [S:CSetoid; op:(CSetoid_bin_op S); one:S; x:S; x_inv:S] : Prop :=
  ((op x x_inv) [=] one).
```

```
Definition is_CGroup [G: CMonoid; inv: (CSetoid_un_op G)] :=
     (x: G)(Is_inverse Csg_op Zero x (inv x)).
```

```
Record CGroup : Type :=
  { cg_crr   :> CMonoid;
    cg_inv   :  (CSetoid_un_op cg_crr);
    cg_proof :  (is_CGroup cg_crr cg_inv)
  }.
```

Notation: `cg_inv ? a` is denoted by `[--]a`.

```
Definition cg_minus := [G: CGroup][x, y: G](x [+] [--]y).
```

Notation: `cg_minus ? a b` is denoted by `a[-]b`.

Name convention: In the *names* of lemmas, we will denote `[--]` with `min`, and `[-]` with `minus`.

10.2 Group axioms

Convention: Let `G` be a group.

```
Lemma cg_rht_inv: (x: G)(Is_inverse Csg_op Zero x (Cg_inv x)).
```

10.3 Group basics

General properties of groups.

Convention: Let `G` be a group.

```
Lemma cg_rht_inv_unfolded: (x:G)((x [+] [--]x) [=] Zero).
```

```
Lemma cg_minus_correct: (x: G)(x [-] x) [=] Zero.
```

```
Lemma cg_lft_inv: (x: G)(Is_inverse Csg_op Zero ([--]x) x).
```

```
Lemma cg_lft_inv_unfolded: (x:G) ([--]x [+] x) [=] Zero.
```

```
Lemma cg_minus_unfolded: (x,y:G) (x [-] y) [=] (x [+] [--]y).
```

```
Lemma cg_minus_wd: (x,x',y,y':G)
  (x [=] x')->(y [=] y')->(x[-]y [=] x'[-]y').
```

```
Lemma cg_minus_strext: (x,x',y,y':G)
  (x[-]y [#] x'[-]y') -> (x [#] x')\/(y [#] y').
```

```
Definition cg_minus_is_csetoid_bin_op : (CSetoid_bin_op G):=
(Build_CSetoid_bin_op G (cg_minus G)
   (cg_minus_wd) (cg_minus_strext)).
```

```
Lemma grp_inv_assoc: (x,y:G) (x[+]y)[-]y [=] x.
```

```
Lemma cg_inv_unique: (x,y:G) ((x [+] y) [=] Zero) -> y [=] [--]x.
```

```
Lemma cg_inv_inv: (x:G)[--][--]x [=] x.
```

```
Lemma cg_cancel_lft: (a, b, c:G)((a [+] b) [=] (a [+] c)) -> b [=] c.
```

```
Lemma cg_cancel_rht: (a, b, c: G)((b [+] a) [=] (c [+] a)) -> b [=] c.
```

```
Lemma cg_inv_unique': (x,y:G) ((x [+] y) [=] Zero) -> x [=] [--]y.
```

```
Lemma cg_inv_unique_2 : (x,y:G)(x[-]y [=] Zero)->(x [=] y).
```

```
Lemma cg_zero_inv: [--]Zero::G [=] Zero.
```

```
Lemma cg_inv_zero : (x:G)x[-]Zero [=] x.
```

```
Lemma cg_inv_op: (x,y:G) [--](x[+]y) [=] [--]y [+] [--]x.
```

```
Lemma csg_op_inv: (x,y:G) [--](x[+]y) [=] [--]x [+] [--]y.
```

10.4 Sub-groups

Convention: Let `P` a predicate on `G` with `P Zero` and which is preserved by `[+]` and `[--]`.

```
Local subcrr : CMonoid := (Build_SubCMonoid ?? Punit op_pres_P).
Local subinv : (CSetoid_un_op subcrr) :=
  (Build_SubCSetoid_un_op ??? inv_pres_P).
```

```
Lemma isgrp_scrr: (is_CGroup subcrr subinv).
```

```
Definition Build_SubCGroup : CGroup := (Build_CGroup subcrr ? isgrp_scrr).
```

10.5 Associative properties of groups

Convention: Let `G` be a group.

```
Lemma assoc_1 : (x,y,z:G)(x[-](y[-]z) [=] (x[-]y)[+]z).

Lemma assoc_2 : (x,y,z:G)(x[+](y[-]z) [=] (x[+]y)[-]z).

Lemma minus_plus : (x,y,z:G)((x[-](y[+]z)) [=] ((x[-]y)[-]z)).

Lemma zero_minus : (x:G)((Zero [-] x) [=] ([--] x)).

Lemma cg_cancel_mixed:(x,y:G)(x[=]x[-]y[+]y).

Lemma plus_resp_eq:(a,b,c:G)(b[=]c)->(a[+]b[=]a[+]c).
```

10.6 Apartness in Constructive Groups

Specific properties of *constructive* groups.

Convention: Let `G` be a group.

```
Lemma cg_add_ap_zero:
  (x,y:G)((x [+] y) [#] Zero) -> (x [#] Zero) \/ (y [#] Zero).

Lemma op_rht_resp_ap: (x,y,z:G)(x [#] y) -> (x [+] z) [#] (y [+] z).

Lemma op_lft_resp_ap: (x,y,z:G)(y [#] z) -> (x [+] y) [#] (x [+] z).

Lemma cg_ap_cancel_rht: (x,y,z:G) ((x [+] z) [#] (y [+] z)) -> x [#] y.

Lemma plus_cancel_ap_rht : (x,y,z:G)((x[+]z) [#] (y[+]z)) -> x[#]y.

Lemma cg_ap_cancel_lft: (x,y,z:G) ((x [+] y) [#] (x [+] z)) -> y [#] z.

Lemma plus_cancel_ap_lft : (x,y,z:G)((z[+]x) [#] (z[+]y)) -> x[#]y.

Lemma minus_ap_zero: (x,y:G)(x [#] y) -> (x[-]y) [#] Zero.

Lemma zero_minus_apart: (x,y:G)((x[-]y) [#] Zero) -> x [#] y.

Lemma min_resp_ap_zero: (x:G)(x [#] Zero) -> [--]x [#] Zero.

Lemma min_resp_ap : (x,y:G)(x[#]y) -> ([--]x) [#] ([--]y).

Lemma minus_resp_ap_lft : (x,y,z:G)(x[#]y) -> (z[-]x) [#] (z[-]y).

Lemma minus_resp_ap_rht : (x,y,z:G)(x[#]y) -> (x[-]z) [#] (y[-]z).

Lemma minus_cancel_ap_rht : (x,y,z:G)((x[-]z) [#] (y[-]z)) -> x[#]y.
```

11 Sums

Convention: Let G be a group.

```
Fixpoint sumlist [l:(list G)] : G :=
  Cases l of
    nil => Zero::G
  | (cons x k) => x[+](sumlist k)
  end.
```

```
Fixpoint sumx [n:nat] : ((i:nat)(lt i n)->G)->G :=
  <[n:nat](((i:nat)(lt i n)->G)->G)>
  Cases n of
    O => [_:(i:nat)(lt i O)->G]Zero::G
  | (S m) => [f:(i:nat)(lt i (S m))->G]
      (sumx m [i:nat][l:(lt i m)](f i (lt_S ?? l)))
        [+](f m (lt_n_Sn m))
  end.
```

```
Fixpoint sum0 [n:nat] : (nat->G)->G := [f:nat->G]
(* sum i=0..(n-1) *)
  Cases n of
    O => Zero::G
  | (S m) => (sum0 m f)[+](f m)
  end.
```

```
Definition sum1 : nat->nat->(nat->G)->G :=
(* sum i=m..(n-1) *)
  [m,n:nat][f:nat->G]((sum0 n f)[-](sum0 m f)).
```

```
Definition sum : nat->nat->(nat->G)->G :=
(* sum i=m..n *)
  [m,n:nat](sum1 m (S n)).
```

```
Lemma sum_one : (n:nat)(f:nat->G)(sum n n f)[=](f n).
```

```
Lemma sum_sum : (l,m,n:nat)(f:nat->G)
  (sum l m f)[+](sum (S m) n f)[=](sum l n f).
```

```
Lemma sum_first : (m,n:nat)(f:nat->G)
  (sum m n f) [=] (f m)[+](sum (S m) n f).
```

```
Lemma sum_last : (m,n:nat)(f:nat->G)
  (sum m (S n) f) [=] (sum m n f)[+](f (S n)).
```

```
Lemma sum_last' : (m,n:nat)(f:nat->G)(lt O n) ->
  (sum m n f) [=] (sum m (pred n) f)[+](f n).
```

```
Lemma sum0_wd : (m:nat)(f,f':nat->G)((i:nat)(f i)[=](f' i)) ->
                (sum0 m f)[=](sum0 m f').
```

```
Lemma sum_wd : (m,n:nat)(f,f':nat->G)((i:nat)(f i)[=](f' i)) ->
               (sum m n f)[=](sum m n f').
```

```
Lemma sum0_plus_sum0 : (f,g:nat->G)(m:nat)
                     (sum0 m [i:nat]((f i)[+](g i))) [=]
                     ((sum0 m f) [+] (sum0 m g)).
```

```
Lemma sum_plus_sum : (f,g:nat->G)(a,b:nat)
                     (sum a b [i:nat]((f i)[+](g i))) [=]
                     ((sum a b f) [+] (sum a b g)).
```

```
Lemma sum_apzero : (a:nat->G)(k,l:nat)(le k l) ->
  ((sum k l a) [#] Zero) ->
    (EX i | (le k i) /\ (le i l) /\ ((a i) [#] Zero)).
```

```
Lemma sum_zero : (a:nat->G)(k,l:nat)(le k (S l)) ->
  ((i:nat)(le k i) -> (le i l) -> (a i) [=] Zero) ->
    (sum k l a) [=] Zero.
```

```
Lemma sum_wd' :
  (k,l:nat)(le k (S l)) -> (a,a':nat->G)
    ((i:nat)(le k i) -> (le i l) -> ((a i)[=](a' i))) ->
      (sum k l a)[=](sum k l a').
```

```
Lemma sum_term : (a:nat->G)(k,i0,l:nat)(le k i0) -> (le i0 l) ->
  ((i:nat)(le k i) -> (~i=i0) -> (le i l) -> (a i) [=] Zero) ->
    (sum k l a) [=] (a i0).
```

```
Lemma sum0_shift : (a,b:nat->G)(l:nat)
  ((i:nat)((a i) [=] (b (S i)))) ->
    (b (O))[+](sum0 l a) [=] (sum0 (S l) b).
```

```
Lemma sum_shift : (a,b:nat->G)(k,l:nat)
  ((i:nat)((a i) [=] (b (S i)))) ->
    (sum k l a) [=] (sum (S k) (S l) b).
```

12 Rings

We actually define commutative rings with 1.

12.1 Definition of the notion of Ring

```
Definition distributive :=
  [S:CSetoid; mult,plus:(CSetoid_bin_op S)]
  (x,y,z:S) (mult x (plus y z)) [=] (plus (mult x y) (mult x z)).
```

```
Record is_CRing [G : CGroup; one : G; mult : (CSetoid_bin_op G)] : Prop :=
  { ax_mult_assoc : (Associative mult);
    ax_mult_mon : (is_CMonoid (Build_CSemi_grp G one mult ax_mult_assoc));
    ax_dist : (Distributive mult (csg_op G));
    ax_non_triv : (one [#] Zero)
  }.
```

```
Record CRing : Type :=
  { cr_crr   :> CGroup;
    cr_one   :  cr_crr;
    cr_mult  :  (CSetoid_bin_op cr_crr);
    cr_proof :  (is_CRing cr_crr cr_one cr_mult)
  }.
```

```
Definition cr_plus  :=  csg_op.
Definition cr_inv   :=  cg_inv.
Definition cr_minus :=  cg_minus.
```

Notation: The term `cr_one` with its first argument left implicit is denoted by `One`.

Notation: The multiplication `cr_mult x y` is denoted by `x [*] y`.

Name convention: In the *names* of lemmas, we will denote `One` with `one`, and `[*]` with `mult`.

12.2 Ring axioms

Convention: Let `R` be a ring.

```
Lemma CRing_is_CRing : (is_CRing R One Cr_mult).
```

```
Lemma mult_assoc : (Associative (cr_mult R)).
```

```
Lemma mult_mon : (is_CMonoid (Build_CSemi_grp R One Cr_mult mult_assoc)).
```

```
Lemma dist : (Distributive Cr_mult (cr_plus R)).
```

```
Lemma ring_non_triv : (One::R [#] Zero).
```

```
Lemma mult_wd : (x1,x2,y1,y2:R)(x1[=]x2)-> (y1[=]y2)-> (x1[*]y1) [=] (x2[*]y2).
```

```
Lemma mult_wd_lft : (x1,x2,y:R)(x1[=]x2)-> (x1[*]y) [=] (x2[*]y).
```

```
Lemma mult_wd_rht : (x,y1,y2:R)(y1[=]y2)-> (x[*]y1) [=] (x[*]y2).
```

12.3 Ring constructions

Convention: Let `R` be a ring.

The multiplicative monoid of a ring is defined as follows.

```
Definition Build_multCMonoid : CMonoid := (Build_CMonoid ? (mult_mon R)).
```

The csetoid of non-zeros of a ring is defined as follows.

```
Definition NonZeros : CSetoid := (Build_SubCSetoid R NonZeroP).
Definition nzinj : NonZeros -> R := (scs_elem R NonZeroP).

Definition nzpro : (x:R)(x [#] Zero)->NonZeros :=
                           (Build_subcsetoid_crr R NonZeroP).

Lemma inj_pro: (x:R; nz:(x[#]Zero)) (nzinj (nzpro x nz)) [=] x.

Lemma pro_inj: (x:NonZeros) (nzpro (nzinj x) (scs_prf ?? x)) [=] x.

Lemma nonz_eq:(x,y:NonZeros)(((nzinj x)[=](nzinj y))-> (x[=]y)).

Lemma nzinj_strong_ext : (x,y:NonZeros)((nzinj x) [#] (nzinj y)) -> (x[#]y).

Lemma nzinj_wd : (x,y:NonZeros)(x[=]y) -> (nzinj x) [=] (nzinj y).

Lemma nzpro_strong_ext : (x,y:R)(Hx:x[#]Zero)(Hy:y[#]Zero)
                        (((nzpro x Hx) [#] (nzpro y Hy)) -> (x[#]y)).

Lemma nzpro_wd : (x,y:R)(Hx:x[#]Zero)(Hy:y[#]Zero) (x [=] y) ->
                 (nzpro x Hx) [=] (nzpro y Hy).
```

Notation 1 *We will write* `nzpro ? x p` *as* `x [//] p`.

12.4 Ring unfolded

Convention: Let `R` be a ring.

```
Lemma mult_assoc_unfolded : (x,y,z:R)
                            (x [*] (y [*] z)) [=] ((x [*] y) [*] z).

Lemma mult_commutes: (x,y:R) (x[*]y) [=] (y[*]x).

Lemma mult_one: (x:R)(x [*] One) [=] x.

Lemma one_mult: (x:R)(One [*] x) [=] x.

Lemma ring_dist_unfolded:
  (x,y,z:R) x [*] (y [+] z) [=] (x [*] y) [+] (x [*] z).

Lemma ring_distl_unfolded:
  (x,y,z:R) (y [+] z) [*] x [=] (y [*] x) [+] (z [*] x).
```

12.5 Ring basics

Convention: Let R be a ring.

```
Lemma one_ap_zero : (One::R [#] Zero).
```

NonZeroP is well-defined and strongly-extensional, but we don't use this.

```
Definition is_zero_rht [S:CSetoid; op:(CSetoid_bin_op S); zero:S]: Prop :=
    (x:S)((op x zero) [=] zero).

Definition is_zero_lft [S:CSetoid; op:(CSetoid_bin_op S); zero:S]: Prop :=
    (x:S)((op zero x) [=] zero).

Lemma cring_mult_zero: (x:R) (x [*] Zero) [=] Zero.

Lemma cring_mult_zero_op: (x:R) Zero[*]x [=] Zero.

Lemma cring_minus_mult_lft: (x,y:R) x [*] ([--]y) [=] [--](x [*] y).

Lemma cring_minus_mult_rht: (x,y:R) ([--]x) [*] y [=] [--](x [*] y).

Lemma cring_mult_ap_zero: (x,y:R)((x [*] y) [#] Zero) -> (x [#] Zero).

Lemma cring_mult_ap_zero_op: (x,y:R)((x [*] y) [#] Zero) -> (y [#] Zero).

Lemma inv_mult_invol: (x,y:R)([--]x[*][--]y [=] x[*]y).

Lemma ring_dist_minus :
  (x,y,z:R) x [*] (y [-] z) [=] (x [*] y) [-] (x [*] z).

Lemma ring_distl_minus:
  (x,y,z:R) (y [-] z) [*] x [=] (y [*] x) [-] (z [*] x).
```

12.6 Ring Auxiliary

Some auxiliary functions and operations over a ring; especially geared towards CReals.

12.6.1 Exponentiation

Convention: Let R be a ring.

```
Fixpoint nexp [x:R;m:nat]: R :=
  Cases m of O => One | (S n) => ((nexp x n) [*] x)
  end.

Lemma nexp_well_def: (n:nat)(Fun_well_def [x:R](nexp x n)).

Lemma nexp_strong_ext: (n:nat)(Fun_strong_ext [x:R](nexp x n)).

Definition nexp_op :=
  [n:nat](Build_CSetoid_un_op R [x:R](nexp x n)
    (nexp_well_def n) (nexp_strong_ext n)).
```

Notation: a [^] b stands for nexp_op ? a b.

12.6.2 The injection of natural numbers into a ring

Convention: Let `R` be a ring.

The injection of Coq natural numbers into a ring is called `nring`.

```
Fixpoint nring [m:nat]: R :=
  Cases m of O => Zero | (S n) => ((nring n) [+] One)
  end.
```

```
Lemma nring_comm_plus:
  (n,m:nat)(nring (plus n m)) [=] ((nring n) [+] (nring m)).
```

```
Lemma nring_comm_mult :
  (n,m:nat)(nring (mult n m)) [=] ((nring n) [*] (nring m)).
```

Notation: `Two` denotes `Nring (S (S O))`, `Three` denotes `Nring (S (S (S O)))` and `Four` denotes `Nring (S (S (S (S O))))`.

```
Lemma one_plus_one : (R:CRing)((One::R) [+] One) [=] Two.
```

```
Lemma x_plus_x : (R:CRing)(x:R)(x[+]x) [=] Two[*]x.
```

12.6.3 The injection of integers into a ring

Convention: Let `R` be a ring.

The injection of Coq integers into a ring is called `zring`.

```
Definition zring [k:Z] : R := (caseZ_diff k [m,n:nat](Nring m) [-] (Nring n)).
```

```
Lemma zring_zero : (zring `0`) [=] Zero.
```

```
Lemma zring_diff : (m,n:nat)(zring `m-n`) [=] (Nring m) [-] (Nring n).
```

```
Lemma zring_plus_nat : (n:nat)(zring `n`) [=] (Nring n).
```

```
Lemma zring_min_nat : (n:nat)(zring `-n`) [=] [--](Nring n).
```

```
Lemma zring_plus : (i,j:Z)(zring `i+j`) [=] (zring i)[+](zring j).
```

```
Lemma zring_min : (i:Z)(zring `-i`) [=] [--](zring i).
```

```
Lemma zring_minus : (i,j:Z)(zring `i-j`) [=] (zring i)[-](zring j).
```

```
Lemma zring_mult : (i,j:Z)(zring `i*j`) [=] (zring i)[*](zring j).
```

```
Lemma zring_one : (zring `1`) [=] One.
```

```
Lemma zring_inv : (x:R)((zring `-1`)[*]x [=] [--]x).
```

12.7 Ring sums

Convention: Let R be a ring.

12.7.1 Infinite Ring sums

```
Fixpoint Sum_upto [r:nat->R; n:nat] : R :=
  Cases n of O     => Zero
   | (S x) => (r x) [+] (Sum_upto r x)
  end.

Lemma sum_upto_O : (r:nat->R)(Sum_upto r O) [=] Zero.

Definition Sum_from_upto [r:nat->R; p,n:nat] : R :=
  (Sum_upto r n) [-] (Sum_upto r p).
```

Here's an alternative def of `Sum_from_upto`, with lemma that it's equivalent to the original.

```
Definition seq_from [r:nat->R; p:nat] : nat->R := [i:nat](r (plus p i)).

Definition Sum_from_upto_alt [r:nat->R; p,n:nat] : R :=
  (Sum_upto (seq_from r p) (minus n p)).

Ax_iom sum_from_upto_eq:
  (r:nat->R)(n,p:nat)(le p n) ->
    (Sum_from_upto_alt r p n) [=] (Sum_from_upto r p n).
```

12.8 Ring Sums over Lists

```
Fixpoint RList_Mem [l:(list R); n:nat] : R :=
  Cases l n of nil _       => Zero
             | (cons a _) O => a
             | (cons _ k) (S y) => (RList_Mem k y)
  end.

Fixpoint List_Sum_upto [l:(list R); n:nat] : R :=
  Cases n of O     => Zero
   | (S x) => (RList_Mem l x) [+] (List_Sum_upto l x)
  end.

Lemma list_sum_upto_O : (l:(list R))(List_Sum_upto l O) [=] Zero.

Definition List_Sum_from_upto [l:(list R); p,n:nat] : R :=
   (List_Sum_upto l n) [-] (List_Sum_upto l p).
```

Here's an alternative def of `List_Sum_from_upto`, with lemma that it's equivalent to the original.

```
Fixpoint List_End [l:(list R); p:nat] : (list R) :=
   Cases p of O     => l
            | (S x) => (List_End (tail l) x)
   end.

Definition List_Sum_from_upto_alt [l:(list R); p,n:nat] : R :=
   (List_Sum_upto (List_End l p) (minus n p)).

Ax_iom list_sum_from_upto_eq :
  (l:(list R))(n,p:nat)(le p n) ->
      (List_Sum_from_upto_alt l p n) [=] (List_Sum_from_upto l p n).
```

12.9 Distribution properties

Convention: Let `R` be a ring.

```
Lemma dist_1b : (x,y,z:R)((x[+]y)[*]z [=] x[*]z[+]y[*]z).

Lemma dist_2a : (x,y,z:R)(z[*](x[-]y) [=] z[*]x[-]z[*]y).

Lemma dist_2b : (x,y,z:R)((x[-]y)[*]z [=] x[*]z[-]y[*]z).

Lemma mult_distr_sum0_lft : (f:nat->R)(x:R)(a:nat)
                            (Sum0 a [i:nat](x [*] (f i))) [=]
                            (x [*] (Sum0 a f)).

Lemma mult_distr_sum_lft : (f:nat->R)(x:R)(a,b:nat)
                           (Sum a b [i:nat](x [*] (f i))) [=]
                           (x [*] (Sum a b f)).

Lemma mult_distr_sum_rht : (f:nat->R)(x:R)(a,b:nat)
                           (Sum a b [i:nat]((f i) [*] x)) [=]
                           ((Sum a b f) [*] x).
```

12.10 Properties of exponenation over N

Convention: Let `R` be a ring.

```
Lemma nexp_wd :(x,y:R)(n:nat)(x[=]y)->(x[^]n [=] y[^]n).

Lemma nexp_Sn : (x:R)(n:nat)(x[*]x[^]n [=] x[^](S n)).

Lemma nexp_plus : (x:R)(m,n:nat)(x[^]m[*]x[^]n [=] x[^](plus m n)).

Lemma one_nexp : (n:nat)((One::R)[^]n [=] One).

Lemma mult_nexp : (x,y:R)(n:nat)((x[*]y)[^]n [=] x[^]n[*]y[^]n).

Lemma nexp_mult : (x:R)(m,n:nat)((x[^]m)[^]n [=] x[^](mult m n)).

Lemma zero_nexp : (x:R)(n:nat)(lt O n)->((Zero::R)[^]n [=] Zero).

Lemma min_nexp_even : (x:R)(n:nat)(even n)->(([--]x)[^]n [=] x[^]n).

Lemma min_nexp_two : (x:R)(([--]x)[^](2) [=] x[^](2)).

Lemma min_nexp_odd : (x:R)(n:nat)(odd n)->(([--]x)[^]n [=] [--](x[^]n)).

Lemma nexp_one : (x:R)(x[^](1) [=] x).

Lemma nexp_two : (x:R)(x[^](2) [=] x[*]x).

Lemma nexp_funny : (x,y:R)((x[+]y)[*](x[-]y) [=] x[^](2)[-]y[^](2)).

Lemma nexp_funny' : (x,y:R)((x[-]y)[*](x[+]y) [=] x[^](2)[-]y[^](2)).
```

13 The Step2 tactic

Some definition and lemmas used by a tactic for equational reasoning.

```
Definition implies := [p,q:Prop]p->q : Prop->Prop->Prop.
```

```
Lemma fold_implies : (p,q:Prop)(p->q)==(implies p q).
```

```
Lemma eq_wd_implies :
  (S:CSetoid)(x,y,x',y':S)(x[=]x')->(y[=]y')->(implies x'[=]y' x[=]y).
```

```
Lemma ap_wd_implies :
  (S:CSetoid)(x,y,x',y':S)(x[=]x')->(y[=]y')->(implies x'[#]y' x[#]y).
```

14 Polynomials

The first section only proves the polynomials form a ring, and nothing more interesting. Section 14.2 gives some basic properties of equality and induction of polynomials.

14.1 Definition of polynomials; they form a ring

Convention: Let CR be a ring.

(cpoly CR) is $R[X]$; cpoly_zero is the 'empty' polynomial with no coefficients; (cpoly_linear c p) is $c + X * p$

```
Inductive cpoly : Set :=
    cpoly_zero : cpoly
  | cpoly_linear : (s:CR)(c:cpoly)cpoly.
```

```
Definition cpoly_constant [c:CR]: cpoly := (cpoly_linear c cpoly_zero).
Definition cpoly_one: cpoly := (cpoly_constant One).
```

Some useful induction lemmas for doubly quantified propositions.

```
Lemma cpoly_double_ind0 : (P:cpoly->cpoly->Prop)
   ((p:cpoly)(P p cpoly_zero)) ->
   ((p:cpoly)(P cpoly_zero p)) ->
   ((p,q:cpoly)(c,d:CR)(P p q)->(P (cpoly_linear c p) (cpoly_linear d q))) ->
   (p,q:cpoly)(P p q).
```

```
Lemma cpoly_double_sym_ind0 : (P:cpoly->cpoly->Prop)
   (Symmetric P) ->
   ((p:cpoly)(P p cpoly_zero)) ->
   ((p,q:cpoly)(c,d:CR)(P p q)->(P (cpoly_linear c p) (cpoly_linear d q))) ->
   (p,q:cpoly)(P p q).
```

```
Lemma cpoly_double_ind0' : (P:cpoly->cpoly->Prop)
   ((p:cpoly)(P cpoly_zero p)) ->
   ((p:cpoly)(c:CR)(P (cpoly_linear c p) cpoly_zero)) ->
   ((p,q:cpoly)(c,d:CR)(P p q)->(P (cpoly_linear c p) (cpoly_linear d q))) ->
   (p,q:cpoly)(P p q).
```

14.1.1 The polynomials form a setoid

```
Fixpoint cpoly_eq_zero [p:cpoly] : Prop :=
  Cases p of
    cpoly_zero => True
  | (cpoly_linear c p1) => (c [=] Zero) /\ (cpoly_eq_zero p1)
  end.


Fixpoint cpoly_eq [p:cpoly] : cpoly->Prop := [q:cpoly]
  Cases p of
    cpoly_zero => (cpoly_eq_zero q)
  | (cpoly_linear c p1) =>
      Cases q of
        cpoly_zero => (cpoly_eq_zero p)
      | (cpoly_linear d q1) => (c [=] d) /\ (cpoly_eq p1 q1)
      end
  end.


Lemma cpoly_eq_p_zero : (p:cpoly)(cpoly_eq p cpoly_zero) == (cpoly_eq_zero p).
Induction p; Auto.
Qed.


Fixpoint cpoly_ap_zero [p:cpoly] : Prop :=
  Cases p of
    cpoly_zero => False
  | (cpoly_linear c p1) => (c [#] Zero) \/ (cpoly_ap_zero p1)
  end.


Fixpoint cpoly_ap [p:cpoly] : cpoly->Prop := [q:cpoly]
  Cases p of
    cpoly_zero => (cpoly_ap_zero q)
  | (cpoly_linear c p1) =>
      Cases q of
        cpoly_zero => (cpoly_ap_zero p)
      | (cpoly_linear d q1) => (c [#] d) \/ (cpoly_ap p1 q1)
      end
  end.


Lemma cpoly_ap_p_zero : (p:cpoly)(cpoly_ap_zero p) == (cpoly_ap p cpoly_zero).


Lemma irreflexive_cpoly_ap : (irreflexive cpoly_ap).


Lemma symmetric_cpoly_ap : (symmetric ? cpoly_ap).


Lemma cotransitive_cpoly_ap : (cotransitive cpoly_ap).


Lemma tight_apart_cpoly_ap : (tight_apart cpoly_eq cpoly_ap).


Lemma cpoly_is_CSetoid : (is_CSetoid ? cpoly_eq cpoly_ap).
```

```
Definition cpoly_csetoid :=
                    (Build_CSetoid cpoly cpoly_eq cpoly_ap cpoly_is_CSetoid).
```

Now that we know that the polynomials form a setoid, we can use the notation with `[#]` and `[=]`. In order to use this notation, we introduce `cpoly_zero_cs` and `cpoly_linear_cs`, so that Coq recognizes we are talking about a setoid. We formulate the induction properties and the most basic properties of equality and apartness in terms of these generators.

```
Local cpoly_zero_cs := cpoly_zero : cpoly_csetoid.

Local cpoly_linear_cs [c:CR; p:cpoly_csetoid] : cpoly_csetoid :=
                      (cpoly_linear c p).


Lemma cpoly_ind_cs : (P:cpoly_csetoid->Prop)
   (P cpoly_zero_cs) ->
   ((p:cpoly_csetoid)(c:CR)(P p)->(P (cpoly_linear_cs c p))) ->
   (p:cpoly_csetoid)(P p).


Lemma cpoly_double_ind0_cs : (P:cpoly_csetoid->cpoly_csetoid->Prop)
   ((p:cpoly_csetoid)(P p cpoly_zero_cs)) ->
   ((p:cpoly_csetoid)(P cpoly_zero_cs p)) ->
   ((p,q:cpoly_csetoid)(c,d:CR)
                 (P p q)->(P (cpoly_linear_cs c p) (cpoly_linear_cs d q))) ->
   (p,q:cpoly_csetoid)(P p q).


Lemma cpoly_double_sym_ind0_cs : (P:cpoly_csetoid->cpoly_csetoid->Prop)
   (Symmetric P) ->
   ((p:cpoly_csetoid)(P p cpoly_zero_cs)) ->
   ((p,q:cpoly_csetoid)(c,d:CR)(P p q)->
                       (P (cpoly_linear_cs c p) (cpoly_linear_cs d q))) ->
   (p,q:cpoly_csetoid)(P p q).


Lemma cpoly_lin_eq_zero : (p:cpoly_csetoid)(c:CR)
     ((cpoly_linear_cs c p) [=] cpoly_zero_cs) ==
     ((c [=] Zero) /\ (p [=] (cpoly_zero_cs))).


Lemma cpoly_zero_eq_lin : (p:cpoly_csetoid)(c:CR)
     (cpoly_zero_cs [=] (cpoly_linear_cs c p)) ==
     ((c [=] Zero) /\ ((cpoly_zero_cs) [=] p)).


Lemma cpoly_lin_eq_lin : (p,q:cpoly_csetoid)(c,d:CR)
     ((cpoly_linear_cs c p) [=] (cpoly_linear_cs d q)) ==
     ((c [=] d) /\ (p [=] q)).


Lemma cpoly_lin_ap_zero : (p:cpoly_csetoid)(c:CR)
     ((cpoly_linear_cs c p) [#] cpoly_zero_cs) ==
     ((c [#] Zero) \/ (p [#] (cpoly_zero_cs))).


Lemma cpoly_zero_ap_lin : (p:cpoly_csetoid)(c:CR)
     (cpoly_zero_cs [#] (cpoly_linear_cs c p)) ==
     ((c [#] Zero) \/ ((cpoly_zero_cs) [#] p)).
```

```
Lemma cpoly_lin_ap_lin : (p,q:cpoly_csetoid)(c,d:CR)
     ((cpoly_linear_cs c p) [#] (cpoly_linear_cs d q)) ==
     ((c [#] d) \/ (p [#] q)).
```

```
Lemma cpoly_linear_strext : (bin_fun_strong_ext ??? cpoly_linear_cs).
```

```
Lemma cpoly_linear_wd : (bin_fun_well_def ??? cpoly_linear_cs).
```

```
Definition cpoly_linear_fun :=
  (Build_CSetoid_bin_fun ???? cpoly_linear_wd cpoly_linear_strext).
```

```
Lemma cpoly_double_comp_ind : (P : cpoly_csetoid -> cpoly_csetoid -> Prop)
 ((p1,p2,q1,q2:cpoly_csetoid)((p1[=]p2)->(q1[=]q2)->(P p1 q1)->(P p2 q2))) ->
 (P cpoly_zero_cs cpoly_zero_cs) ->
 ((p,q:cpoly_csetoid)(c,d:CR)(P p q)->
                           (P (cpoly_linear_cs c p) (cpoly_linear_cs d q))) ->
 (p,q:cpoly_csetoid)(P p q).
```

```
Lemma cpoly_triple_comp_ind :
 (P : cpoly_csetoid -> cpoly_csetoid -> cpoly_csetoid -> Prop)
 ((p1,p2,q1,q2,r1,r2:cpoly_csetoid)((p1[=]p2)->(q1[=]q2)->(r1[=]r2)->
                   (P p1 q1 r1)->(P p2 q2 r2))) ->
 (P cpoly_zero_cs cpoly_zero_cs cpoly_zero_cs) ->
 ((p,q,r:cpoly_csetoid)(c,d,e:CR)(P p q r)->
     (P (cpoly_linear_cs c p) (cpoly_linear_cs d q) (cpoly_linear_cs e r))) ->
 (p,q,r:cpoly_csetoid)(P p q r).
```

14.1.2 The polynomials form a semi-group and a monoid

```
Fixpoint cpoly_plus [p:cpoly] : cpoly -> cpoly := [q:cpoly]
  Cases p of
    cpoly_zero => q
  | (cpoly_linear c p1) =>
      Cases q of
        cpoly_zero => p
      | (cpoly_linear d q1) => (cpoly_linear c[+]d (cpoly_plus p1 q1))
      end
  end.
```

```
Definition cpoly_plus_cs [p,q:cpoly_csetoid] : cpoly_csetoid
                                             := (cpoly_plus p q).
```

```
Lemma cpoly_zero_plus : (p:cpoly_csetoid)(cpoly_plus_cs cpoly_zero_cs p) = p.
```

```
Lemma cpoly_plus_zero : (p:cpoly_csetoid)(cpoly_plus_cs p cpoly_zero_cs) = p.
```

```
Lemma cpoly_lin_plus_lin : (p,q:cpoly_csetoid)(c,d:CR)
             (cpoly_plus_cs (cpoly_linear_cs c p) (cpoly_linear_cs d q)) =
             (cpoly_linear_cs (c[+]d) (cpoly_plus_cs p q)).
```

```
Lemma cpoly_plus_commutative :
               (p,q:cpoly_csetoid)(cpoly_plus_cs p q) [=] (cpoly_plus_cs q p).

Lemma cpoly_plus_q_ap_q : (p,q:cpoly_csetoid)
              ((cpoly_plus_cs p q) [#] q) ->
              (p [#] cpoly_zero_cs).

Lemma cpoly_p_plus_ap_p : (p,q:cpoly_csetoid)
              ((cpoly_plus_cs p q) [#] p) -> (q [#] cpoly_zero).

Lemma cpoly_ap_zero_plus : (p,q:cpoly_csetoid)
              ((cpoly_plus_cs p q) [#] cpoly_zero_cs) ->
              (p [#] cpoly_zero_cs) \/ (q [#] cpoly_zero_cs).

Lemma cpoly_plus_op_strext : (bin_op_strong_ext cpoly_csetoid cpoly_plus_cs).

Lemma cpoly_plus_op_proof : (bin_op_well_def cpoly_csetoid cpoly_plus_cs).

Definition cpoly_plus_op :=
  (Build_CSetoid_bin_op ?? cpoly_plus_op_proof cpoly_plus_op_strext).

Lemma cpoly_plus_associative : (Associative cpoly_plus_op).

Definition cpoly_csemi_grp :=
       (Build_CSemi_grp cpoly_csetoid cpoly_zero cpoly_plus_op
                        cpoly_plus_associative).

Lemma cpoly_cm_proof : (is_CMonoid cpoly_csemi_grp).

Definition cpoly_cmonoid := (Build_CMonoid ? cpoly_cm_proof).
```

14.1.3 The polynomials form a group

```
Fixpoint cpoly_min [p:cpoly] : cpoly :=
  Cases p of
    cpoly_zero => cpoly_zero
  | (cpoly_linear c p1) => (cpoly_linear [--]c (cpoly_min p1))
  end.

Definition cpoly_min_cs [p:cpoly_csetoid] : cpoly_csetoid := (cpoly_min p).

Lemma cpoly_min_zero : (cpoly_min_cs cpoly_zero_cs) = cpoly_zero_cs.

Lemma cpoly_min_lin : (p:cpoly_csetoid)(c:CR)
    (cpoly_min_cs (cpoly_linear_cs c p)) =
    (cpoly_linear_cs ([--]c) (cpoly_min_cs p)).

Lemma cpoly_min_op_strext : (un_op_strong_ext cpoly_csetoid cpoly_min_cs).

Lemma cpoly_min_op_proof : (un_op_well_def cpoly_csetoid cpoly_min_cs).

Definition cpoly_min_op :=
  (Build_CSetoid_un_op ?? cpoly_min_op_proof cpoly_min_op_strext).

Lemma cpoly_cg_proof : (is_CGroup cpoly_cmonoid cpoly_min_op).

Definition cpoly_cgroup := (Build_CGroup ?? cpoly_cg_proof).
```

14.1.4 The polynomials form a ring

```
Fixpoint cpoly_mult_cr [q:cpoly] : CR -> cpoly := [c:CR]
  Cases q of
    cpoly_zero => cpoly_zero
  | (cpoly_linear d q1) => (cpoly_linear c[*]d (cpoly_mult_cr q1 c))
  end.
```

```
Fixpoint cpoly_mult [p:cpoly] : cpoly -> cpoly := [q:cpoly]
  Cases p of
    cpoly_zero => cpoly_zero
  | (cpoly_linear c p1) =>
      (cpoly_plus (cpoly_mult_cr q c)
                  (cpoly_linear Zero (cpoly_mult p1 q)))
  end.
```

```
Definition cpoly_mult_cr_cs [p:cpoly_csetoid; c:CR] : cpoly_csetoid
                                                    := (cpoly_mult_cr p c).
```

```
Lemma cpoly_zero_mult_cr : (c:CR)
                            (cpoly_mult_cr_cs cpoly_zero_cs c)=cpoly_zero_cs.
```

```
Lemma cpoly_lin_mult_cr : (c,d:CR)(q:cpoly_csetoid)
                              (cpoly_mult_cr_cs (cpoly_linear_cs d q) c) =
                              (cpoly_linear_cs c[*]d (cpoly_mult_cr_cs q c)).
```

```
Lemma cpoly_mult_cr_zero : (p:cpoly_csetoid)
                           (cpoly_mult_cr_cs p Zero) [=] cpoly_zero_cs.
```

```
Lemma cpoly_mult_cr_strext :
          (bin_fun_strong_ext cpoly_cgroup CR cpoly_cgroup cpoly_mult_cr_cs).
```

```
Lemma cpoly_mult_cr_wd :
           (bin_fun_well_def cpoly_cgroup CR cpoly_cgroup cpoly_mult_cr_cs).
```

```
Definition cpoly_mult_cs [p,q:cpoly_csetoid] : cpoly_csetoid
                                             := (cpoly_mult p q).
```

```
Lemma cpoly_zero_mult : (q:cpoly_csetoid)
                        (cpoly_mult_cs cpoly_zero_cs q)=cpoly_zero_cs.
```

```
Lemma cpoly_lin_mult : (c:CR)(p,q:cpoly_csetoid)
                         (cpoly_mult_cs (cpoly_linear_cs c p) q) =
                         (cpoly_plus_cs (cpoly_mult_cr_cs q c)
                             (cpoly_linear_cs Zero (cpoly_mult_cs p q))).
```

```
Lemma cpoly_mult_op_strext : (bin_op_strong_ext cpoly_csetoid cpoly_mult_cs).
```

```
Lemma cpoly_mult_op_proof : (bin_op_well_def cpoly_csetoid cpoly_mult).

Definition cpoly_mult_op :=
  (Build_CSetoid_bin_op ?? cpoly_mult_op_proof cpoly_mult_op_strext).

Lemma cpoly_mult_cr_dist : (p,q:cpoly_csetoid)(c:CR)
               (cpoly_mult_cr_cs (cpoly_plus_cs p q) c) [=]
               (cpoly_plus_cs (cpoly_mult_cr_cs p c) (cpoly_mult_cr_cs q c)).

Lemma cpoly_cr_dist : (Distributive cpoly_mult_op cpoly_plus_op).

Lemma cpoly_mult_cr_assoc_mult_cr : (p:cpoly_csetoid)(c,d:CR)
   (cpoly_mult_cr_cs (cpoly_mult_cr_cs p c) d)
      [=] (cpoly_mult_cr_cs p d[*]c).

Lemma cpoly_mult_cr_assoc_mult : (p,q:cpoly_csetoid)(c:CR)
          (cpoly_mult_cr_cs (cpoly_mult_cs p q) c)
      [=] (cpoly_mult_cs (cpoly_mult_cr_cs p c) q).

Lemma cpoly_mult_zero :
   (p:cpoly_csetoid)(cpoly_mult_cs p cpoly_zero_cs) [=] (cpoly_zero_cs).

Lemma cpoly_mult_lin : (c:CR)(p,q:cpoly_csetoid)
                         (cpoly_mult_cs p (cpoly_linear_cs c q)) [=]
                         (cpoly_plus_cs (cpoly_mult_cr_cs p c)
                             (cpoly_linear_cs Zero (cpoly_mult_cs p q))).

Lemma cpoly_mult_commutative :
   (p,q:cpoly_csetoid)(cpoly_mult_cs p q) [=] (cpoly_mult_cs q p).

Lemma cpoly_mult_dist_rht :
   (p,q,r:cpoly_csetoid)
   (cpoly_mult_cs (cpoly_plus_cs p q) r) [=]
   (cpoly_plus_cs (cpoly_mult_cs p r) (cpoly_mult_cs q r)).

Lemma cpoly_mult_assoc : (Associative cpoly_mult_op).

Lemma cpoly_mult_cr_one : (p:cpoly_csetoid)(cpoly_mult_cr_cs p One)[=]p.

Lemma cpoly_one_mult : (p:cpoly_csetoid)(cpoly_mult_cs cpoly_one p)[=]p.

Lemma cpoly_mult_one : (p:cpoly_csetoid)(cpoly_mult_cs p cpoly_one)[=]p.

Lemma cpoly_mult_monoid: (is_CMonoid
   (Build_CSemi_grp cpoly_csetoid cpoly_one cpoly_mult_op cpoly_mult_assoc)).
```

```
Lemma cpoly_cr_non_triv : (cpoly_ap cpoly_one cpoly_zero).
```

```
Lemma cpoly_is_CRing : (is_CRing cpoly_cgroup cpoly_one cpoly_mult_op).
```

```
Definition cpoly_cring :=
  (Build_CRing cpoly_cgroup cpoly_one cpoly_mult_op cpoly_is_CRing) : CRing.
```

```
Lemma cpoly_constant_strext : (fun_strong_ext CR cpoly_cring  cpoly_constant).
```

```
Lemma cpoly_constant_wd : (fun_well_def CR cpoly_cring cpoly_constant).
```

```
Definition _c_ :=
  (Build_CSetoid_fun CR cpoly_cring cpoly_constant
                     cpoly_constant_wd cpoly_constant_strext).
```

```
Definition _x_ : cpoly_cring := (cpoly_linear_cs Zero One::cpoly_cring).
```

```
Definition cpoly_x_minus_c : CR -> cpoly_cring
                           := [c:CR](cpoly_linear_cs ([--]c) One::cpoly_cring).
```

```
Lemma cpoly_x_minus_c_strext : (fun_strong_ext CR cpoly_cring cpoly_x_minus_c).
```

```
Lemma cpoly_x_minus_c_wd : (fun_well_def CR cpoly_cring cpoly_x_minus_c).
```

Notation: `cpoly_linear_fun ? c p` is denoted with `c [+X*] p`.

14.2 Apartness, equality, and induction

```
Lemma linear_eq_zero : (p:Cpoly_cring)(c:CR)
     ((c [+X*] p) [=] Zero) == ((c [=] Zero) /\ (p [=] Zero)).
```

```
Lemma zero_eq_linear : (p:Cpoly_cring)(c:CR)
     (Zero [=] c [+X*] p) == ((c [=] Zero) /\ (Zero [=] p)).
```

```
Lemma linear_eq_linear : (p,q:Cpoly_cring)(c,d:CR)
     (c [+X*] p [=] d [+X*] q) == ((c [=] d) /\ (p [=] q)).
```

```
Lemma linear_ap_zero : (p:Cpoly_cring)(c:CR)
     (c [+X*] p [#] Zero) == ((c [#] Zero) \/ (p [#] Zero)).
```

```
Lemma zero_ap_linear : (p:Cpoly_cring)(c:CR)
     (Zero [#] c [+X*] p) == ((c [#] Zero) \/ (Zero [#] p)).
```

```
Lemma linear_ap_linear : (p,q:Cpoly_cring)(c,d:CR)
     (c [+X*] p [#] d [+X*] q) == ((c [#] d) \/ (p [#] q)).
```

```
Lemma cpoly_induc : (P:Cpoly_cring->Prop)
   (P Zero) ->
   ((p:Cpoly_cring)(c:CR)(P p)->(P (c [+X*] p))) ->
   (p:Cpoly_cring)(P p).
```

```
Lemma cpoly_double_ind : (P:Cpoly_cring->Cpoly_cring->Prop)
   ((p:Cpoly_cring)(P p Zero)) ->
   ((p:Cpoly_cring)(P Zero p)) ->
   ((p,q:Cpoly_cring)(c,d:CR)(P p q)->(P (c [+X*] p) (d [+X*] q))) ->
   (p,q:Cpoly_cring)(P p q).
```

```
Lemma cpoly_double_sym_ind : (P:Cpoly_cring->Cpoly_cring->Prop)
   (Symmetric P) ->
   ((p:Cpoly_cring)(P p Zero)) ->
   ((p,q:Cpoly_cring)(c,d:CR)(P p q)->(P (c [+X*] p) (d [+X*] q))) ->
   (p,q:Cpoly_cring)(P p q).
```

```
Lemma poly_double_comp_ind : (P : Cpoly_cring -> Cpoly_cring -> Prop)
 ((p1,p2,q1,q2:Cpoly_cring)((p1[=]p2)->(q1[=]q2)->(P p1 q1)->(P p2 q2))) ->
 (P Zero Zero) ->
 ((p,q:Cpoly_cring)(c,d:CR)(P p q)->(P (c [+X*] p) (d [+X*] q))) ->
 (p,q:Cpoly_cring)(P p q).
```

```
Lemma poly_triple_comp_ind :
 (P : Cpoly_cring -> Cpoly_cring -> Cpoly_cring -> Prop)
 ((p1,p2,q1,q2,r1,r2:Cpoly_cring)((p1[=]p2)->(q1[=]q2)->(r1[=]r2)->
                   (P p1 q1 r1)->(P p2 q2 r2))) ->
 (P Zero Zero Zero) ->
 ((p,q,r:Cpoly_cring)(c,d,e:CR)(P p q r)->
        (P (c [+X*] p) (d [+X*] q) (e [+X*] r))) ->
 (p,q,r:Cpoly_cring)(P p q r).
```

```
Fixpoint cpoly_apply [p:Cpoly_cring] : CR -> CR := [x:CR]
  Cases p of
    cpoly_zero => Zero
  | (cpoly_linear c p1) =>
      (c [+](x [*] (cpoly_apply p1 x)))
  end.
```

```
Lemma cpoly_apply_strext : (bin_fun_strong_ext ??? cpoly_apply).
```

```
Lemma cpoly_apply_wd : (bin_fun_well_def ??? cpoly_apply).
```

```
Definition cpoly_apply_fun :=
  (Build_CSetoid_bin_fun ???? cpoly_apply_wd cpoly_apply_strext).
```

Notation: `cpoly_apply_fun` is denoted infix by `!`. The first argument is left implicit. In the names of lemmas, we write `apply`.

14.3 Basic properties of polynomials

Convention: Let R be a ring and write RX for the ring of polynomials over R.

14.3.1 Constant and identity

```
(* BELONGS IN CSETOIDS ! *)
Lemma csetoid_fun_wd_unfolded : (S1,S2:CSetoid)(f:(CSetoid_fun S1 S2))
                            (x,x':S1)(x[=]x')->(f x)[=](f x').


(* BELONGS IN CSETOIDS ! *)
Lemma csetoid_bin_fun_wd_unfolded :
       (S1,S2,S3:CSetoid)(f:(CSetoid_bin_fun S1 S2 S3))
              (x,x':S1;y,y':S2)(x[=]x')->(y[=]y')->(f x y)[=](f x' y').


Lemma cpoly_X_ : (_X_ [=] ((Zero::RX) [+X*] One)).


Lemma cpoly_C_ : (c:RX)((_C_ c) [=] (c [+X*] Zero)).


Lemma cpoly_const_eq : (c,d:R)(c[=]d)->(_C_ c)[=](_C_ d).


Lemma _c_zero : Zero [=] (_C_ Zero::R).


Lemma _c_one : One [=] (_C_ One::R).


Lemma _c_mult : (a,b:R)((_C_ a[*]b) [=] (_C_ a)[*](_C_ b)).


Lemma cpoly_lin : (p:RX)(c:R)
                 (c [+X*] p) [=] ((_C_ c) [+] (_X_ [*] p)).


(* SUPERFLUOUS *)
Lemma poly_linear : (c:R)(f:RX)
  ((cpoly_linear ? c f) [=] _X_[*]f[+](_C_ c)).


Lemma poly_c_apzero : (a:R)((_C_ a) [#] Zero) -> (a [#] Zero).


Lemma _c_mult_lin : (p:RX)(c,d:R)
                     (_C_ c) [*] (d [+X*] p) [=] c[*]d [+X*] ((_C_ c) [*] p).


(* SUPERFLUOUS ? *)
Lemma lin_mult : (p,q:RX)(c:R)
 ((c [+X*] p) [*] q) [=] (_C_ c) [*] q [+] _X_ [*] (p [*] q).
```

14.3.2 Application

```
(* !!! SUPERFLUOUS *)
Lemma poly_eq_zero :(p:RX)(p[=](cpoly_zero R)) -> (x:R)(p!x [=] Zero).


Lemma apply_wd : (p,p':RX)(x,x':R)(p [=] p')->(x [=] x')->(p!x [=] p'!x').


Lemma cpolyap_pres_eq :(f:RX;x,y:R)(x[=]y)->(f!x)[=](f!y).


Lemma cpolyap_strext :(f:RX;x,y:R)((f!x)[#](f!y))->(x[#]y).


Definition cpoly_csetoid_op : RX -> (CSetoid_un_op R) :=
[f:RX](Build_CSetoid_fun ??[x:R](f!x) (cpolyap_pres_eq f)(cpolyap_strext f)).


Lemma _c_apply : (c,x:R)((_C_ c)!x [=] c).


Lemma _x_apply : (x:R)(_X_!x [=] x).


Lemma plus_apply : (p,q:RX)(x:R)((p [+] q)!x [=] p!x [+] q!x).


Lemma min_apply : (p:RX)(x:R)(([--]p)!x [=] [--](p!x)).


Lemma minus_apply : (p,q:RX)(x:R)((p [-] q)!x [=] p!x [-] q!x).


Lemma _c_mult_apply : (q:RX)(c,x:R)(((_C_ c) [*] q)!x [=] c [*] q!x).


Lemma mult_apply : (p,q:RX)(x:R)((p [*] q)!x [=] p!x [*] q!x).


Lemma one_apply : (x:R)(One!x [=] One).


Lemma nexp_apply : (p:RX)(n:nat)(x:R)((p[^]n)!x [=] (p!x)[^]n).


(* SUPERFLUOUS *)
Lemma poly_min_apply :
  (p:RX)(x:R)((cpoly_min ? p)!x [=] [--](p!x)).


Lemma sum0_cpoly_ap : (f:nat->RX)(a:R)(k:nat)
  ((Sum0 k f)!a [=] (Sum0 k [i:nat](f i)!a)).


Lemma sum_cpoly_ap : (f:nat->RX)(a:R)(k,l:nat)
  ((Sum k l f)!a [=] (Sum k l [i:nat](f i)!a)).
```

15 Polynomials: NthCoeff

Convention: Let `R` be a ring and write `RX` for the ring of polynomials over `R`.

The n-th coefficient of a polynomial. The default value is `Zero::CR` e.g. if the n is higher than the length. For the polynomial $a_0 + a_1X + a_2X^2 + \cdots + a_nX^n$, the 0-th coefficient is a_0, the 1-th is a_1 etcetera.

```
Fixpoint nth_coeff [n:nat; p:RX] : R :=
  Cases p of
    cpoly_zero => Zero
  | (cpoly_linear c q) =>
      Cases n of
        O => c
      | (S m) => (nth_coeff m q)
      end
  end.
```

```
Lemma Nth_coeff_strext : (n:nat)(p,p':RX)
               ((nth_coeff n p)[#](nth_coeff n p')) -> p[#]p'.
```

```
Lemma Nth_coeff_wd : (n:nat)(p,p':RX)
  (p [=] p') -> ((nth_coeff n p) [=] (nth_coeff n p')).
```

```
Definition nth_coeff_fun[n:nat] :=
  (Build_CSetoid_fun ??? (Nth_coeff_wd n) (Nth_coeff_strext n)).
```

Coq shortcoming: We would like to use `nth_coeff_fun n` all the time. However, Coq's coercion mechanism doesn't support this properly: the term `(nth_coeff_fun n p)` won't get parsed, and has to be written as `((nth_coeff_fun n) p)` instead.

So, in the names of lemmas, we write `(nth_coeff n p)`, which always (e.g. in proofs) can be converted to `((nth_coeff_fun n) p)`.

```
Definition nonConst [p:RX]: Prop :=
   (EX n | (lt O n)/\((nth_coeff n p) [#] Zero)).
```

```
(* Following probably NOT needed *)
(* These functions are NOT extensional, that is *)
(* they are not SETOID functions *)

Fixpoint nth_coeff_ok [n:nat; p:RX] : bool :=
  Cases n p of
     O cpoly_zero => false
  |  O (cpoly_linear c q) => true
  | (S m) cpoly_zero => false
  | (S m) (cpoly_linear c q) => (nth_coeff_ok m q)
  end.

(* The in_coeff predicate*)
Fixpoint in_coeff  [c:R;p:RX] : Prop :=
      Cases p of
  cpoly_zero => False
```

```
| (cpoly_linear d q) => (c [=] d) \/ (in_coeff c q)
      end.
(* cpoly_zero case should be (c [=] Zero) to be extensional *)


Lemma nth_coeff_S :
 (m:nat)(p:RX)(c:R)(in_coeff (nth_coeff m p) p)
   ->(in_coeff (nth_coeff (S m) (c [+X*] p)) (c [+X*] p)).
Simpl; Auto.
Qed.


Lemma Nth_coeff_zero : (n:nat)(Nth_coeff n Zero::RX)[=]Zero.


Lemma coeff_O_lin : (p:RX)(c:R)(Nth_coeff (0) (c[+X*]p)) [=]c.


Lemma coeff_Sm_lin : (p:RX)(c:R)(m:nat)
                     (Nth_coeff (S m) (c[+X*]p)) [=] (Nth_coeff m p).


Lemma coeff_O_c_ : (c:R)(Nth_coeff (0) (_C_ c))[=]c.


Lemma coeff_O_x_mult : (p:RX)(Nth_coeff (0) (_X_ [*] p)) [=] Zero.


Lemma coeff_Sm_x_mult : (p:RX)(m:nat)
             (Nth_coeff (S m) (_X_ [*] p)) [=] (Nth_coeff m p).


Lemma coeff_Sm_mult_x_ : (p:RX)(m:nat)
             (Nth_coeff (S m) (p [*] _X_)) [=] (Nth_coeff m p).


Lemma Nth_coeff_ap_zero_imp : (p:RX)(n:nat)
                             ((Nth_coeff n p) [#] Zero) -> (p [#] Zero).


Lemma Nth_coeff_plus : (p,q:RX)(n:nat)
  (Nth_coeff n (p[+]q)) [=] (Nth_coeff n p)[+](Nth_coeff n q).


Lemma Nth_coeff_min : (p:RX)(n:nat)
  (Nth_coeff n ([--]p)) [=] [--](Nth_coeff n p).


Lemma Nth_coeff_c_mult_p : (p:RX)(c:R)(n:nat)
  ((Nth_coeff n (_C_ c)[*]p) [=] c[*](Nth_coeff n p)).


Lemma Nth_coeff_p_mult_c_ : (p:RX)(c:R)(n:nat)
  ((Nth_coeff n p[*](_C_ c)) [=] (Nth_coeff n p)[*]c).


Lemma Nth_coeff_complicated : (a,b:R)(p:RX)(n:nat)
  (Nth_coeff (S n) ((_C_ a)[*]_X_[+](_C_ b))[*]p) [=]
    a[*](Nth_coeff n p)[+]b[*](Nth_coeff (S n) p).
```

```
Lemma all_Nth_coeff_eq_imp : (p,p':RX)
  ((i:nat)((Nth_coeff i p) [=] (Nth_coeff i p'))) -> (p [=] p').


Lemma poly_at_zero : (p:RX)(p!Zero [=] (Nth_coeff O p)).


Lemma nth_coeff_min' :
  (p:(cpoly R))(i:nat)(Nth_coeff i (cpoly_min ? p)) [=] [--](Nth_coeff i p).


Lemma nth_coeff_minus :
  (p,q:RX)(i:nat)(Nth_coeff i p[-]q) [=] (Nth_coeff i p)[-](Nth_coeff i q).


Lemma nth_coeff_sum0 :
  (p_:nat->RX)(k,n:nat)
    (Nth_coeff k (Sum0 n p_)) [=] (Sum0 n [i:nat](Nth_coeff k (p_ i))).


Lemma nth_coeff_sum :
  (p_:nat->RX)(k,m,n:nat)
    (Nth_coeff k (Sum m n p_)) [=] (Sum m n [i:nat](Nth_coeff k (p_ i))).


Lemma nth_coeff_nexp_eq :
   (i:nat)(Nth_coeff i (_X_ [^]i)) [=] One::R.


Lemma nth_coeff_nexp_neq :
   (i,j:nat)~(i=j)->(Nth_coeff i (_X_[^]j)) [=] Zero::R.


Lemma nth_coeff_mult :
  (p,q:RX)(n:nat)(Nth_coeff n p[*]q) [=]
    (Sum (O) n [i:nat](Nth_coeff i p)[*](Nth_coeff (minus n i) q)).
```

16 Fields

16.1 Definition of the notion Field

```
Definition is_CField
  [R: CRing; cf_rcpcl: (CSetoid_un_op (NonZeros R))] : Prop :=
  (x: (NonZeros R))(Is_inverse (cr_mult R) One
                              (NZinj x) (NZinj (cf_rcpcl x))).

Record CField : Type :=
  { cf_crr   :> CRing;
    cf_rcpcl :  (CSetoid_un_op (NonZeros cf_crr));
    cf_proof :  (is_CField cf_crr cf_rcpcl)
  }.


Definition f_rcpcl [F:CField; x:(NonZeros F)] : F := (NZinj ((cf_rcpcl ?) x)).
```

`cf_div` is the division in a field.

```
Definition cf_div [F:CField; x: F; y: (NonZeros F)]: F := (x [*] (F_rcpcl y)).
```

Notation: The division `cf_div x y` is denoted infix by `x [/] y`.

Recall that `x [//] p` is of type `NonZeros F` if `x : F` and `p : x [#] Zero` (see Notation 1).

Convention 2

1. *We do not use* `NonZeros`*, but write the condition* `[#]Zero` *separately.*
2. *In each lemma, we use only variables for proof objects, and these variables are universally quantified.*

   E.g. the informal lemma

   ```
   (1/x).(1/y) = 1/(x.y) for all x and y
   ```

   is formalized as

   ```
   (x,y:F)(H1:x#0)(H2:y#0)(H3:(x.y)#0) (1/x//H1).(1/y//H2) = 1/(x.y)//H3
   ```

   and not *as*

   ```
   (x,y:F)(H1:x#0)(H2:y#0)(1/x//H1).(1/y//H2) = 1/(x.y)//(prod_nz x y H1 H2)
   ```

   We have made this choice to make it easier to apply lemmas; this can be quite awkward if we would use the last formulation.
3. *So every division occurring in the formulation of a lemma is of the form* `e[/]e'[//]H` *where* `H` *is a variable. Only exceptions: we may write* `e[/](Snring n)` *and* `e[/]TwoNZ`*,* `e[/]ThreeNZ` *and so on. (Constants like* `TwoNZ` *will be defined later on.)*

16.2 Field axioms

Convention: Let `F` be a field.

```
Lemma CField_is_CField : (is_CField F (cf_rcpcl F)).
```

```
Lemma rcpcl_is_inverse :
 (x: (NonZeros F))(Is_inverse (cr_mult F) One
                              (NZinj x) (NZinj ((cf_rcpcl F) x))).
```

16.3 Field basics

Convention: Let `F` be a field.

```
Lemma rcpcl_is_inverse_unfolded :
            (x: (NonZeros F))(NZinj x) [*] (NZinj ((cf_rcpcl F) x)) [=] One.
```

```
Lemma field_mult_inv: (x:F)(H:x[#]Zero)(x [*] (F_rcpcl (x[//]H))) [=] One.
```

```
Lemma field_mult_inv_op : (x:F)(H:x[#]Zero)((F_rcpcl (x[//]H)) [*] x) [=] One.
```

16.4 Properties of multiplication

Convention: Let `F` be a field.

```
Lemma mult_resp_ap_zero:
  (x,y:F)(x [#] Zero) -> (y [#] Zero) -> (x[*]y [#] Zero).
```

```
Lemma mult_lft_resp_ap :
   (x,y,z:F)(x [#] y) -> (z [#] Zero) -> (z [*] x) [#] (z [*] y).
```

```
Lemma mult_rht_resp_ap :
   (x,y,z:F)(x [#] y) -> (z [#] Zero) -> (x [*] z) [#] (y [*] z).
```

```
Lemma mult_resp_neq_zero :
   (x,y:F)(x [~=] Zero)->(y [~=] Zero)->((x [*] y) [~=] Zero).
```

```
Lemma mult_resp_neq :
   (x,y,z:F)(x [~=] y)->(z [~=] Zero)->((x [*] z) [~=] (y [*] z)).
```

```
Lemma mult_eq_zero :
   (x,y:F)(x [~=] Zero)->((x [*] y) [=] Zero)->(y [=] Zero).
```

```
Lemma mult_cancel_lft : (x,y,z:F)(z [#] Zero)->(z[*]x [=] z[*]y)->(x [=] y).
```

```
Lemma mult_cancel_rht : (x,y,z:F)(z [#] Zero)->(x[*]z [=] y[*]z)->(x [=] y).
```

The multiplicative group of nonzeros of a field. The multiplicative monoid of NonZeros.

```
Definition NonZeroMonoid : CMonoid :=
  (Build_SubCMonoid (Build_multCMonoid F)
                    (nonZeroP F) (one_ap_zero F) mult_resp_ap_zero).
```

```
Lemma fmg_cs_inv: (CSetoid_un_op NonZeroMonoid).
```

```
Lemma plus_nonzeros_eq_mult_dom : (x,y:NonZeroMonoid)
            (NZinj (x [+] y)) [=] ((NZinj x) [*] (NZinj y)).
```

```
Lemma cfield_to_mult_cgroup : CGroup.
```

```
Lemma square_eq_aux :
      (x,a:F)(x[^](2) [=] a[^](2))-> (x[+]a)[*](x[-]a) [=] Zero.
```

```
Lemma square_eq_weak :
      (x,a:F)(x[^](2) [=] a[^](2))-> ~((x [#] a) /\ (x [#] [--]a)).
```

```
Lemma conditional_square_eq :
   (x,a:F)(Two::F [#] Zero)->(a [#] Zero)->
   (x[^](2) [=] a[^](2))->((x [=] a) \/ (x [=] [--]a)).
```

16.5 Properties of reciprocal

Convention: Let F be a field.

```
Lemma inv_one : (F_rcpcl (One[//](ring_non_triv F))) [=] One.
```

```
Lemma cf_rcpcl_wd : (F:CField)(x,y:(NonZeros F)) (x [=] y) ->
                                    ((cf_rcpcl F) x) [=] ((cf_rcpcl F) y).
```

```
Lemma F_rcpcl_wd : (F:CField)(x,y:(NonZeros F)) (x [=] y) ->
                                            (F_rcpcl x) [=] (F_rcpcl y).
```

```
Lemma F_rcpcl_mult : (y,z:F)(nzy:y [#] Zero)(nzz:z[#]Zero)(nzyz:y[*]z[#]Zero)
 (F_rcpcl ((y[*]z) [//]nzyz)) [=] (F_rcpcl y [//] nzy) [*](F_rcpcl z [//] nzz).
```

```
Lemma F_rcpcl_resp_ap_zero : (y:F)(nzy:(y [#] Zero))
                                  ((F_rcpcl (y [//]nzy)) [#] Zero).
```

```
Lemma F_rcpcl_F_rcpcl : (x:F)(nzx:x [#] Zero)(nzr:(F_rcpcl x [//] nzx)[#]Zero)
                        (F_rcpcl (F_rcpcl x [//] nzx) [//] nzr) [=] x.
```

16.6 Properties of division

Convention: Let F be a field.

Name convention: In the names of lemmas, we denote [/] by div, and One[/] by recip.

```
Lemma div_prop : (F:CField)(x:F)(H:x[#]Zero)(Zero[/]x[//]H [=] Zero).
```

```
Lemma div_1 : (x,y:F)(y_:y [#] Zero)((x[/](y[//]y_))[*]y [=] x).
```

```
Lemma div_1' : (x,y:F)(y_:y [#] Zero)(y[*](x[/](y[//]y_)) [=] x).
```

```
Lemma div_1'' : (x,y:F)(y_:y [#] Zero)((x[*]y)[/](y[//]y_) [=] x).
```

```
Lemma x_div_x : (x:F)(x_: x [#] Zero)((x[/](x[//]x_)) [=] One).
```

```
Lemma x_div_one : (x:F)((x[/](One[//](ring_non_triv (F)))) [=] x).
```

The next lemma says x.(y/z) = (x.y)/z.

```
Lemma x_mult_y_div_z : (x,y,z:F)(z_: z [#] Zero)
                       ((x[*](y[/](z[//]z_))) [=] ((x[*]y)[/](z[//]z_))).
```

```
Lemma div_wd : (x0,x1,y0,y1:F)(nzy0:y0 [#] Zero)(nzy1:y1[#]Zero)
  (x0 [=] x1)->(y0 [=] y1)->(x0[/]y0[//]nzy0 [=] x1[/]y1[//]nzy1).
```

The next lemma says (x/y)/z = x/(y.z)

```
Lemma div_div : (x,y,z:F)(nzy:y [#] Zero)(nzz:z[#]Zero)
  (nzyz:y[*]z[#]Zero)
    (((x[/]y[//]nzy)[/]z[//]nzz) [=] (x[/](y[*]z)[//]nzyz)).
```

```
Lemma div_resp_ap_zero_rev : (x,y:F)(nzy:(y [#] Zero))(x[#]Zero) ->
                                  (x [/] y [//]nzy [#] Zero).
```

```
Lemma div_resp_ap_zero:
(x,y:F)(nzy:(y [#] Zero))(x [/] y [//]nzy [#] Zero) -> (x[#]Zero).
```

The next lemma says `x/(y/z) = (x.z)/y`

```
Lemma div_div2 : (x,y,z:F)
  (nzy:y [#] Zero)(nzz:z[#]Zero)(nzyz:(y[/]z[//]nzz)[#]Zero)
  ((x[/](y[/](z[//]nzz)) [//]nzyz) [=] ((x[*]z)[/](y[//]nzy))).
```

The next lemma says `(x.p)/(y.q) = (x/y).(p/q)`

```
Lemma mult_of_divs : (x,y,p,q:F)
  (nzy:(y[#]Zero))(nzq:(q[#]Zero))(nzyq:y[*]q[#]Zero)
    ((x[*]p)[/](y[*]q)[//]nzyq [=] (x[/]y[//]nzy)[*](p[/]q[//]nzq)).
```

```
Lemma div_dist : (x,y,z:F)(nzz:z[#]Zero)
  (x[+]y)[/]z[//]nzz [=] x[/]z[//]nzz[+]y[/]z[//]nzz.
```

```
Lemma div_semi_sym : (x,y,z:F)(Hy:y[#]Zero)(Hz:z[#]Zero)
                     (x[/]y[//]Hy)[/]z[//]Hz [=] (x[/]z[//]Hz)[/]y[//]Hy.
```

```
Lemma eq_div : (x,y,u,v:F)(Hy:y[#]Zero)(Hv:v[#]Zero)
               (x[*]v [=] u[*]y) -> (x[/]y[//]Hy [=] u[/]v[//]Hv).
```

```
Lemma div_strong_ext : (x0,x1,y0,y1:F)(Hy0:y0[#]Zero)(Hy1:y1[#]Zero)
                       (x0[/]y0[//]Hy0 [#] x1[/]y1[//]Hy1) ->
                       (x0[#]x1) \/ (y0[#]y1).
```

```
Lemma mult_cancel_ap_zero_lft :
 (x,y:R)(x [*] y [#] Zero) -> x [#] Zero.
```

```
Lemma mult_cancel_ap_zero_rht :
 (x,y:R)(x [*] y [#] Zero) -> y [#] Zero.
```

17 Reflection

18 Polynomials: Degree

Convention: Let `R` be a ring and write `RX` for the ring of polynomials over `R`.

The length of a polynomial is the number of its coefficients. This is a syntactical property, as the highest coefficient may be 0. Note that the 'zero' polynomial `cpoly_zero` has length 0, a constant polynomial has length 1 and so forth. So the length is always 1 higher than the 'degree' (assuming that the highest coefficient is # 0)!

```
Fixpoint lth_of_poly [p: RX] : nat :=
  Cases p of
      cpoly_zero => O
| (cpoly_linear d q) => (S(lth_of_poly q))
  end.
```

When dealing with constructive polynomials, notably over the reals or complex numbers, the degree may be unknown, as we can not decide whether the highest coeffiecient is # 0. Hence, degree is a *relation* between polynomials and natural numbers; if the degree is unknown for polynomial p, degree(n, p) doesn't hold for any n. If we don't know the degree of p, we may still know it to be below or above a certain number. E.g. for the polynomial $p_0 + p_1X + \cdots + p_{n-1}X^{n-1}$, if $p_i \# 0$, we can say that the 'degree is at least i' and if $p_{j+1} = \ldots = p_n = 0$ (with n the length of the polynomial), we can say that the 'degree is at most j'.

```
Definition degree_le : nat -> RX -> Prop :=
[n:nat][p:RX] (m:nat)((lt n m)-> ((Nth_coeff m p) [=] Zero)).

Definition degree : nat -> RX -> Prop :=
[n:nat][p:RX] ((Nth_coeff n p) [#] Zero) /\ (degree_le n p).

Definition monic : nat -> RX -> Prop :=
[n:nat][p:RX] ((Nth_coeff n p) [=] One) /\ (degree_le n p).

Definition odd_cpoly : RX -> Prop :=
[p:RX](EX n | (odd n)/\(degree n p)).

Definition even_cpoly : RX -> Prop :=
[p:RX](EX n | (even n)/\(degree n p)).

Definition regular :RX-> Prop :=
[p:RX](EX n |(degree n p)).


Lemma Degree_le_wd : (p,p':RX)(n:nat)
  (p [=] p')->(Degree_le n p)->(Degree_le n p').


Lemma Degree_wd : (p,p':RX)(n:nat)
  (p [=] p')->(Degree n p)->(Degree n p').


Lemma Monic_wd : (p,p':RX)(n:nat)
  (p [=] p')->(Monic n p)->(Monic n p').


Lemma Degree_imp_Degree_le : (p:RX)(n:nat)(Degree n p)->(Degree_le n p).


Lemma Degree_le_c_ : (c:R)(Degree_le O (_C_ c)).


Lemma Degree_c_ : (c:R)(c [#] Zero)->(Degree O (_C_ c)).


Lemma Monic_c_one : (Monic O (_C_ One::R)).


Lemma Degree_le_x_ : (Degree_le (1) _X_::RX).
```

```
Lemma Degree_x_ : (Degree (1) _X_::RX).

Lemma Monic_x_ : (Monic (1) _X_::RX).

Lemma degree_le_mon : (p:RX)(m,n:nat)
  (le m n) -> (Degree_le m p) -> (Degree_le n p).

Lemma Degree_le_min : (p:RX)(n:nat)(Degree_le n p)->(Degree_le n [--]p).

Lemma Degree_le_plus : (p,q:RX)(n:nat)
  (Degree_le n p) -> (Degree_le n q) -> (Degree_le n p[+]q).

Lemma Degree_le_minus : (p,q:RX)(n:nat)
  (Degree_le n p) -> (Degree_le n q) -> (Degree_le n p[-]q).

Lemma sum_degree_le : (f:nat->RX)(n,k,l:nat)(le k (S l)) ->
  ((i:nat)(le k i) -> (le i l) -> (Degree_le n (f i))) ->
    (Degree_le n (Sum k l f)).

Lemma Degree_min : (p:RX)(n:nat)(Degree n p)->(Degree n [--]p).

Lemma Degree_plus_rht : (p,q:RX)(m,n:nat)
             (Degree_le m p)->(Degree n q)->(lt m n)->(Degree n (p [+] q)).

Lemma Degree_minus_lft : (p,q:RX)(m,n:nat)
             (Degree_le m p)->(Degree n q)->(lt m n)->(Degree n (q [-] p)).

Lemma Monic_plus : (p,q:RX)(m,n:nat)
  (Degree_le m p) -> (Monic n q) -> (lt m n) -> (Monic n p[+]q).

Lemma Monic_minus : (p,q:RX)(m,n:nat)
  (Degree_le m p) -> (Monic n q) -> (lt m n) -> (Monic n q[-]p).

Lemma Degree_le_mult : (p,q:RX)(m,n:nat)
             (Degree_le m p)->(Degree_le n q)->(Degree_le (plus m n) p[*]q).

Lemma Degree_mult_aux : (p,q:RX)(m,n:nat)
 (Degree_le m p)->(Degree_le n q)->
             (Nth_coeff (plus m n) p[*]q) [=]
               (Nth_coeff m p)[*](Nth_coeff n q).

Lemma Monic_mult : (p,q:RX)(m,n:nat)
             (Monic m p)->(Monic n q)->(Monic (plus m n) p[*]q).

Lemma Degree_le_nexp : (p:RX)(m,n:nat)
             (Degree_le m p)->(Degree_le (mult m n) p[^]n).
```

```
Lemma Monic_nexp : (p:RX)(m,n:nat)(Monic m p)->(Monic (mult m n) p[^]n).

Lemma lt_i_Lth_of_poly : (i:nat)(p:RX)
  ((Nth_coeff i p) [#] Zero) -> (lt i (Lth_of_poly p)).

Lemma poly_degree_lth : (p:RX)
  (Degree_le (Lth_of_poly p) p).

Lemma poly_ex_degree : (p:RX)(EX n | (Degree_le n p)).

Lemma poly_as_sum'' : (p:RX)(n:nat)(Degree_le n p)->
  p [=] (Sum (0) n [i:nat](_C_ (Nth_coeff i p))[*]_X_[^]i).

Lemma poly_as_sum' : (p:RX)
  p [=] (Sum (0) (Lth_of_poly p) [i:nat](_C_ (Nth_coeff i p))[*]_X_[^]i).

Lemma poly_as_sum : (p:RX)(n:nat)(Degree_le n p)->
  (x:R)(p!x [=] (Sum (0) n [i:nat](Nth_coeff i p)[*]x[^]i)).

Lemma degree_le_zero : (p:RX)
  (Degree_le (0) p) -> (EX a | p [=] (_C_ a)).

Lemma Degree_le_1_imp : (p:RX)
  (Degree_le (1) p) -> (EX a | (EX b | p [=] (_C_ a)[*]_X_[+](_C_ b))).

Lemma degree_le_cpoly_linear :
  (p:(cpoly R))(c:R)(n:nat)
    (Degree_le (S n) (cpoly_linear ? c p)) -> (Degree_le n p).

Lemma monic_cpoly_linear :
  (p:(cpoly R))(c:R)(n:nat)
    (Monic (S n) (cpoly_linear ? c p)) -> (Monic n p).

Lemma monic_one :
  (p:(cpoly R))(c:R)
    (Monic (1) (cpoly_linear ? c p)) -> (x:R)(p!x [=] One).

Lemma monic_apzero :
  (p:RX)(n:nat)(Monic n p) -> (p [#] Zero).

Lemma Degree_mult : (p,q:FX)(m,n:nat)
             (Degree m p)->(Degree n q)->(Degree (plus m n) p[*]q).

Lemma Degree_nexp : (p:FX)(m,n:nat)(Degree m p)->(Degree (mult m n) p[^]n).

Lemma Degree_le_mult_imp : (m,n:nat)(p,q:FX)
  (Degree m p) -> (Degree_le (plus m n) p[*]q) -> (Degree_le n q).

Lemma Degree_mult_imp : (p,q:FX)(m,n:nat)
  (Degree m p) -> (Degree (plus m n) p[*]q) -> (Degree n q).
```

19 Ordered Fields

19.1 Definition of the notion of ordered field

```
Record strictorder [A:Set; R: A->A->Prop] : Prop :=
  { so_trans : (Transitive R);
    so_asym  : (antisymmetric R)
  }.
```

```
Record is_COrdField [F: CField; less : (CSetoid_relation F)] : Prop :=
 {ax_less_strorder : (strictorder less);
  ax_plus_resp_less: (x,y:F)(less x y) -> (z:F)(less (x[+]z) (y[+]z));
  ax_mult_resp_pos : (x,y:F)(less Zero x)->(less Zero y)->(less Zero (x[*]y));
  ax_less_conf_ap  : (x,y:F)(x [#] y) <-> ((less x y) \/ (less y x))
 }.
```

```
Record COrdField : Type :=
  {cof_crr   :> CField;
   cof_less  :  (CSetoid_relation cof_crr);
   cof_proof :  (is_COrdField cof_crr cof_less)
  }.
```

Notation: `cof_less` is denoted infix by `[:<]`. The first argument is left implicit. In the names of lemmas, `[:<]` is written as `less` and `Zero [:<]` is written as `pos`.

```
Definition greater := [F:COrdField; x,y : F] y[:<]x.
```

Notation: `a [:>] b` denotes `greater ? a b`.

Less or equal is defined as “not greater than”.

```
Definition leEq [F:COrdField; x,y:F]: Prop := ~(y[:<]x).
```

Notation: `LeEq` is denoted infix by `[:<=]`. In lemmas, `[:<=]` is written as `leEq`, and `Zero [:<=]` is written as `nonneg`.

19.2 COrdField axioms

Convention: Let `F` be a field.

```
Lemma COrdField_is_COrdField : (is_COrdField F (cof_less F)).
```

```
Lemma less_strorder : (strictorder (cof_less F)).
```

```
Lemma less_transitive_unfolded : (x,y,z:F)(x[:<]y) -> (y[:<]z) -> (x[:<]z).
```

```
Lemma less_antisymmetric_unfolded : (x,y:F)((x[:<]y) -> ~(y[:<]x)).
```

```
Lemma less_irreflexive : (irreflexive (cof_less F)).
```

```
Lemma less_irreflexive_unfolded : (x:F)(~(x[:<]x)).
```

```
Lemma plus_resp_less_rht : (x,y,z:F)(x [:<] y) -> (x[+]z [:<] y[+]z).
```

```
Lemma mult_resp_pos :
                (x,y:F)(Zero [:<] x)->(Zero [:<] y)->(Zero [:<] (x[*]y)).
```

```
Lemma less_conf_ap : (x,y:F)(x [#] y) <-> ((x [:<] y) \/ (y [:<] x)).
```

```
Lemma less_wdr : (x,y,z:F)(x[:<]y) -> (y [=] z) -> (x[:<]z).
```

```
Lemma less_wdl : (x,y,z:F)(x[:<]y) -> (x [=] z) -> (z[:<]y).
```

```
Lemma less_wd_implies :
  (x,y,x',y':F)(x[=]x')->(y[=]y')->(implies x'[:<]y' x[:<]y).
```

19.3 Basics

Convention: Let `R` be a `COrdField`.

```
Lemma less_imp_ap : (x,y:R)((x [:<] y) -> (x [#] y)).
```

```
Lemma Greater_imp_ap : (x,y:R)((y [:<] x) -> (x [#] y)).
```

```
Lemma ap_imp_less : (x,y:R)((x [#] y) -> ((x [:<] y)\/(y [:<] x))).
```

Now properties which can be derived.

```
Lemma cotrans_less : (cotransitive (cof_less R)).
```

```
Lemma cotrans_less_unfolded : (x,y:R)(x [:<] y)->(z:R)(x[:<]z)\/(z[:<]y).
```

```
Lemma pos_ap_zero : (x:R)(Zero [:<] x)-> x [#] Zero.
```

19.4 Infinity of ordered fields

In an ordered field we have that `One [+] One` and `One [+] One [+] One` and so on are all apart from zero. We first show this, so that we can define `TwoNZ`, `ThreeNZ` and so on. These are elements of `NonZeros`, so that we can write e.g. `x[/]TwoNZ`.

Convention: Let `R` be a field.

```
Lemma pos_one : (Zero::R [:<] One).
```

```
Lemma nring_less_succ : (m:nat)((Nring m)::R) [:<] (Nring (S m)).
```

```
Lemma nring_less : (m,n:nat)(lt m n) -> (((Nring m)::R) [:<] (Nring n)).
```

```
Lemma nring_leEq : (m,n:nat)(le m n) -> (((Nring m)::R) [:<=] (Nring n)).
```

```
Lemma nring_apart : (m,n:nat)(~(m=n) -> ((Nring m)::R [#] (Nring n))).

Lemma nring_ap_zero :(n:nat)(~(n=O) ->((nring R n)[#]Zero)).

Lemma nringS_ap_zero:(m:nat)((nring R (S m))[#]Zero).

Lemma nring_ap_zero_imp:(n:nat)((nring R n)[#]Zero)->(~(O=n)).

Definition Snring : nat->(NonZeros R).

Lemma pos_Snring : (n:nat)((Zero::R) [:<] (NZinj (Snring n))).

Lemma nring_fac_ap_zero:(n:nat)((nring R (fac n))[#]Zero).
```

19.4.1 Properties of one up to four

Name convention: In the names of lemmas, we denote the numbers 0,1,2,3,4 and so on, by `zero`, `one`, `two` etc.

```
Lemma less_plusOne: (x:R) x [:<] x[+]One.

Lemma plus_one_ext_less: (x,y:R)(x [:<] y) -> x [:<] y[+]One.

Lemma one_less_two : (One::R [:<] Two).

Lemma two_less_three : (Two::R [:<] Three).

Lemma three_less_four : (Three::R [:<] Four).

Lemma pos_two : (Zero::R [:<] Two).

Lemma one_less_three : (One::R [:<] Three).

Lemma two_less_four : (Two::R [:<] Four).

Lemma pos_three : (Zero::R [:<] Three).

Lemma one_less_four : (One::R [:<] Four).

Lemma pos_four : (Zero::R [:<] Four).

Lemma two_ap_zero : Two [#] Zero::R.

Lemma three_ap_zero : Three [#] Zero::R.

Lemma four_ap_zero : Four [#] Zero::R.
```

Convention: `OneNZ`, `TwoNZ`, `ThreeNZ` and `FourNZ` are 1, 2, 3 and 4 as elements of `NonZeros`.

19.4.2 Consequences of infinity

```
Lemma square_eq :
   (F:COrdField)(x,a:F)(a [#] Zero)->
   (x[^](2) [=] a[^](2))->((x [=] a) \/ (x [=] [--]a)).
```

19.5 Properties of $\leq$ (leEq)

Convention: Let R be a COrdField.

```
Lemma leEq_wdr : (x,y,z:R)(x[:<=]y) -> (y [=] z) -> (x[:<=]z).

Lemma leEq_wdl : (x,y,z:R)(x[:<=]y) -> (x [=] z) -> (z[:<=]y).

Lemma leEq_reflexive : (x:R)(x[:<=]x).

Lemma eq_imp_leEq : (x,y:R)(x[=]y)->(x[:<=]y).

Lemma leEq_imp_eq : (x,y:R)(x [:<=] y) -> (y [:<=] x) -> (x [=] y).

Lemma lt_equiv_imp_eq : (x,x':R)
  ((y:R)(x [:<] y) -> (x' [:<] y)) ->
  ((y:R)(x' [:<] y) -> (x [:<] y)) ->
    x [=] x'.

Lemma less_leEq_trans : (x,y,z:R)(x [:<] y) -> (y [:<=] z) -> (x [:<] z).

Lemma leEq_less_trans : (x,y,z:R)(x [:<=] y) -> (y [:<] z) -> (x [:<] z).

Lemma leEq_transitive:(x,y,z:R)(x[:<=]y)->(y[:<=]z)->(x[:<=]z).

Lemma less_leEq : (x,y:R)(x [:<] y) -> (x [:<=] y).
```

19.6 Properties of $<$ (less)

Convention: Let R be a COrdField.

Convention: We do not use a special predicate for positivity, (e.g. PosP), but just write Zero [:<] x. Reasons: it is more natural; in ordinary mathematics we also write $0 < x$ (or $x > 0$).

19.6.1 Addition and subtraction

$+$ and $-$ respect $<$

```
Lemma plus_resp_less_lft : (x,y,z:R)(x [:<] y) -> (z[+]x) [:<] (z[+]y).

Lemma min_resp_less : (x,y:R)(x [:<] y) -> ([--]y) [:<] ([--]x).

Lemma minus_resp_less : (x,y,z:R)(x [:<] y) -> ((x[-]z) [:<] (y[-]z)).

Lemma minus_resp_less_rht :(x,y,z:R)(y [:<] x) -> ((z[-]x) [:<] (z[-]y)).

Lemma plus_resp_less_both : (a,b,c,d:R)(a [:<] b) -> (c [:<] d) ->
                                       ((a[+]c) [:<] (b[+]d)).
```

For versions of plus_resp_less_both where one [:<] in the assumption is replaced by [:<=] see Section 19.7.1.

Cancellation laws

```
Lemma plus_cancel_less : (x,y,z:R)((x[+]z) [:<] (y[+]z)) -> (x [:<] y).
```

```
Lemma min_cancel_less : (x,y:R)(([--]x) [:<] ([--]y)) -> (y [:<] x).
```

Laws for shifting

Lemmas where an operation is transformed into the inverse operation on the other side of an inequality are called laws for shifting.

Name convention: The names of laws for shifting start with "`shift_`", and then come the operation and the inequality, in the order in which they occur in the conclusion. If the shifted operand changes sides w.r.t. the operation and its inverse, the name gets a prime.

It would be nicer to write the laws for shifting as bi-implications, However, it is impractical to use these in Coq (see the Coq shortcoming in Section 6.4).

```
Lemma shift_less_plus :   (a,b,c:R)(a[-]c [:<] b) -> a [:<] b[+]c.

Lemma shift_less_plus' :  (a,b,c:R)(a[-]b [:<] c) -> a [:<] b[+]c.

Lemma shift_less_minus :  (a,b,c:R)(a[+]c [:<] b)-> (a [:<] b [-] c).

Lemma shift_less_minus' : (a,b,c:R)(c[+]a [:<] b)-> (a [:<] b [-] c).

Lemma shift_plus_less :   (a,b,c:R)(a [:<] c [-] b) -> (a[+]b [:<] c).

Lemma shift_plus_less' :  (a,b,c:R)(b [:<] c [-] a) -> (a[+]b [:<] c).

Lemma shift_minus_less :  (a,b,c:R)(a [:<] c[+]b) -> (a[-]b [:<] c).

Lemma shift_minus_less' : (a,b,c:R)(a [:<] b[+]c) -> (a[-]b [:<] c).
```

Some special cases of laws for shifting.

```
Lemma shift_zero_less_minus : (x,y:R)(x[:<]y) -> (Zero[:<](y[-]x)).

Lemma qltone: (q:R) (q [:<] One) ->(q[-]One [#]Zero).
```

19.6.2 Multiplication and division

By convention 2 in CFields (Section 16), we often have redundant premises in lemmas. E.g. the informal statement

$$\text{for all } x, y \text{ in } R \text{ with } 0 < x \text{ and } 0 < y \text{ we have } 0 < y/x$$

is formalized as follows.

```
(x,y:R)(H: (x [#] Zero))(Zero [:<] x) -> (Zero [:<] y)->
(Zero [:<] y[/]x[//]H)
```

We do this to keep it easy to use such lemmas.

Multiplication and division respect <

```
Lemma mult_resp_less :
 (x,y,z:R)(x [:<] y) -> (Zero [:<] z) -> ((x [*] z) [:<] (y [*] z)).
```

```
Lemma recip_resp_pos :
 (y:R)(Hy:y[#]Zero)(Zero [:<] y) -> Zero [:<] (One [/] y[//]Hy).
```

```
Lemma div_resp_less_rht :
 (x,y,z:R)(H1:z[#]Zero)(x [:<] y) -> (Zero [:<] z) ->
        ((x [/] z[//]H1) [:<] (y [/] z[//]H1)).
```

```
Lemma div_resp_pos :
        (x,y:R)(H: (x [#] Zero))(Zero [:<] x) -> (Zero [:<] y)->
        (Zero [:<] y[/]x[//]H).
```

```
Lemma mult_resp_less_lft :
 (x,y,z:R)(x [:<] y) -> (Zero [:<] z) -> ((z [*] x) [:<] (z [*] y)).
```

```
Lemma mult_resp_less_both :
 (x,y,u,v:R)(Zero [:<=] x) -> (x [:<] y) -> (Zero [:<=] u) -> (u [:<] v) ->
 x[*]u [:<] y[*]v.
```

```
Lemma recip_resp_less :
 (x,y:R)(Hx:x[#]Zero)(Hy:y[#]Zero)(Zero [:<] x) -> (x [:<] y) ->
           (One [/] y[//]Hy) [:<] (One [/] x[//]Hx).
```

Cancellation laws

```
Lemma mult_cancel_less : (x,y,z:R)(Zero [:<] z) ->
                         ((x[*]z) [:<] (y[*]z)) -> (x [:<] y).
```

Laws for shifting

For namegiving, see the Section 19.6.1 on plus and minus.

```
Lemma shift_div_less : (x,y,z:R)(H:y[#]Zero)(Zero [:<] y) ->
                       (x [:<] (z[*]y)) -> ((x[/]y[//]H) [:<] z).
```

```
Lemma shift_div_less' : (x,y,z:R)(H:y[#]Zero)(Zero [:<] y) ->
                        (x [:<] (y [*]z)) -> ((x[/]y[//]H) [:<] z).
```

```
Lemma shift_less_div : (x,y,z:R)(H:y[#]Zero)(Zero [:<] y) ->
                    ((x[*]y) [:<] z) -> (x [:<] z[/]y[//]H).
```

```
Lemma shift_less_mult' : (x,y,z:R)(H:y[#]Zero)(Zero [:<] y) ->
                         ((x[/]y[//]H) [:<] z) -> (x [:<] (y [*]z)).
```

```
Lemma shift_mult_less : (x,y,z:R)(H:y[#]Zero)(Zero [:<] y) ->
                        (x [:<] z[/]y[//]H) -> ((x[*]y) [:<] z).
```

Other properties of multiplication and division

```
Lemma swap_div : (x,y,z:R)(Hy:y[#]Zero)(Hz:z[#]Zero)
                 (Zero [:<] y) -> (Zero [:<] z) ->
                 ((x[/]z[//]Hz)[:<] y) -> ((x[/]y[//]Hy) [:<] z).

Lemma eps_div_less_eps : (eps,d:R)(d_:d [#] Zero)
                         (Zero[:<]eps) -> (One [:<] d) ->
                         ((eps[/](d[//]d_))[:<]eps).

Lemma pos_div_two : (eps:R)(Zero [:<] eps) -> (Zero [:<] (eps[/]TwoNZ)).
```

19.6.3 Miscellaneous

```
Lemma nring_pos : (m:nat)((lt O m) -> (Zero [:<] (nring R m))).

Lemma Smallest_less_Average : (a,b:R)(a[:<]b)->(a[:<]((a[+]b) [/] TwoNZ)).

Lemma Average_less_Greatest : (a,b:R)(a[:<]b)->((a[+]b) [/] TwoNZ) [:<] b.

Lemma sum_resp_less : (f,g:nat->R)(a,b:nat)(le a b)->
                      ((i:nat)(le a i) -> (le i b) -> (f i) [:<] (g i)) ->
                      (Sum a b f) [:<] (Sum a b g).
```

19.7 Properties of $\leq$ (leEq)

Convention: Let R be a COrdField.

19.7.1 Addition and subtraction

+ and - respect $\leq$

```
Lemma plus_resp_leEq : (x,y,z:R)(x [:<=] y) -> (x[+]z) [:<=] (y[+]z).

Lemma plus_resp_leEq_lft : (x,y,z:R)(x [:<=] y) -> (z[+]x) [:<=] (z[+]y).

Lemma minus_resp_leEq : (x,y,z:R)(x [:<=] y) -> (x[-]z) [:<=] (y[-]z).

Lemma min_resp_leEq :  (x,y:R)(x [:<=] y) -> ([--]y) [:<=] ([--]x).

Lemma plus_resp_leEq_both : (x,y,a,b:R)(x [:<=] y)->(a [:<=] b)
->(x [+] a [:<=] y [+] b).

Lemma plus_resp_less_leEq : (a,b,c,d:R)(a [:<] b) -> (c [:<=] d) ->
                                       ((a[+]c) [:<] (b[+]d)).

Lemma plus_resp_leEq_less : (a,b,c,d:R)(a [:<=] b) -> (c [:<] d) ->
                                       ((a[+]c) [:<] (b[+]d)).

Lemma minus_resp_less_leEq :
  (x,y,x',y':R)(x [:<=] y) -> (y' [:<] x') -> (x[-]x' [:<] y[-]y').

Lemma minus_resp_leEq_both :
  (x,y,x',y':R)(x [:<=] y) -> (y' [:<=] x') -> (x[-]x' [:<=] y[-]y').
```

Cancellation properties

```
Lemma plus_cancel_leEq_rht: (x,y,z:R)( (x[+]z) [:<=] (y[+]z) )->
(x[:<=]y).
```

```
Lemma min_cancel_leEq:(x,y:R)([--]y[:<=][--]x)->(x[:<=]y).
```

Laws for shifting

```
Lemma shift_leEq_plus' : (a,b,c:R)(a[-]b [:<=] c) -> (a[:<=] b[+]c).
```

```
Lemma shift_leEq_rht: (a,b:R)(Zero [:<=] b[-]a) -> (a [:<=] b).
```

```
Lemma shift_leEq_lft : (a,b:R)(a [:<=] b) -> (Zero [:<=] b[-]a).
```

19.7.2 Multiplication and division

Multiplication and division respect $\leq$

```
Lemma mult_resp_leEq_rht:
            (x,y,z:R)(x [:<=] y) -> (Zero [:<=] z) -> (x[*]z) [:<=] (y[*]z).
```

```
Lemma mult_resp_leEq_lft:
            (x,y,z:R)(x[:<=]y)->(Zero[:<=]z)-> (z[*]x[:<=]z[*]y).
```

```
Lemma recip_resp_leEq : (x,y:R)(Hx:x[#]Zero)(Hy:y[#]Zero)
                        (Zero[:<]y) -> (y[:<=]x) ->
                        (One[/]x[//]Hx [:<=] One[/]y[//]Hy).
```

Cancellation properties

```
Lemma mult_cancel_leEq : (x,y,z:R)(Zero [:<] z) ->
                         ((x[*]z) [:<=] (y[*]z)) -> (x [:<=] y).
```

Laws for shifting

```
Lemma shift_leEq_mult' :
                (x,y,z:R)(H:y[#]Zero)(Zero[:<]y) ->
                (x[/]y[//]H [:<=] z) -> (x [:<=] y[*]z).
```

```
Lemma shift_leEq_div :
                (x,y,z:R)(H:y[#]Zero)(Zero[:<]y) ->
                (x[*]y [:<=] z) -> (x [:<=] z[/]y[//]H).
```

19.7.3 Miscellaneous

```
Lemma sqr_nonneg: (x:R) Zero [:<=] (x[^](2)).


Lemma cc_abs_aid: (x,y:R)(Zero [:<=] ((x[^](2)) [+] (y[^](2)))).


Lemma nexp_resp_pos : (x:R)(k:nat)(Zero [:<] x)->(Zero [:<] x[^]k).


Lemma power_resp_leEq : (x,y:R)(k:nat)
  (Zero [:<] x)->(x [:<=] y)->(x[^]k [:<=] y[^]k).


Lemma mult_resp_nonneg : (x,y:R)(Zero[:<=]x) -> (Zero[:<=]y) ->
                                (Zero[:<=] x[*]y).


Lemma nexp_resp_nonneg : (x:R)(k:nat)(Zero [:<=] x)->(Zero [:<=] x[^]k).


Lemma nexp_resp_less : (x,y:R)(n:nat)(le (1) n) -> (Zero [:<=] x) ->
                            (x [:<] y) -> x[^]n [:<] y[^]n.


Lemma power_cancel_leEq : (x,y:R)(k:nat)(lt O k) ->
  (Zero [:<=] y)->(x[^]k [:<=] y[^]k)->(x [:<=] y).


Lemma power_cancel_less : (x,y:R)(k:nat)
  (Zero [:<=] y)->(x[^]k [:<] y[^]k)->(x [:<] y).


Lemma sum_resp_leEq : (f,g:nat->R)(a,b:nat)(le a (S b)) ->
                      ((i:nat)(le a i) -> (le i b) -> (f i) [:<=] (g i)) ->
                      (Sum a b f) [:<=] (Sum a b g).


Lemma approach_zero:(x:R)((e:R)(Zero[:<]e)->(x[:<]e))->
~(Zero[:<]x).


Lemma approach_zero_weak:(x:R)((e:R)(Zero [:<] e)-> x [:<=] e)->
                              ~(Zero [:<] x).
```

19.8 Properties of positive numbers

Convention: Let R be a COrdField.

```
Syntactic Definition ZeroR := (Zero::R).
Syntactic Definition OneR := (One::R).


Definition pos_ap_zeros: (x:R)(Zero [:<] x)-> (NonZeros R).


Lemma mult_pos_imp : (a,b:R)(Zero [:<] a[*] b) ->
    ((Zero [:<] a) /\ (Zero [:<] b)) \/ ((a [:<] Zero) /\ (b [:<] Zero)).
```

```
Lemma plus_resp_pos_nonneg :
       (x,y:R)(Zero [:<] x) -> (Zero [:<=] y) -> (Zero [:<] x[+]y).
```

```
Lemma plus_resp_nonneg_pos :
       (x,y:R)(Zero [:<=] x) -> (Zero [:<] y) -> (Zero [:<] x[+]y).
```

```
Lemma pos_square : (x:R)(x [#] Zero)->(Zero [:<] x[^](2)).
```

```
Lemma mult_cancel_pos_rht : (x,y:R)(Zero [:<] x[*]y)->(Zero [:<] x)->(Zero [:<] y).
```

```
Lemma mult_cancel_pos_lft : (x,y:R)(Zero [:<] x[*]y)->(Zero [:<] y)->(Zero [:<] x).
```

```
Lemma pos_wd : (x,y:R)(x [=] y)->(Zero [:<] y)->(Zero [:<] x).
```

```
Lemma even_power_pos : (n:nat)(even n) -> (x:R)(x [#] Zero) -> (Zero [:<] x[^]n).
```

```
Lemma odd_power_cancel_pos : (n:nat)(odd n)->(x:R)(Zero [:<] x[^]n)->(Zero [:<] x).
```

```
Lemma plus_resp_pos : (x,y:R)(Zero [:<] x)->(Zero [:<] y)->(Zero [:<] x[+]y).
```

```
Lemma pos_nring_S: (n:nat)((Zero::R) [:<] (Nring (S n))).
```

19.9 Properties of non-negative numbers

Convention: Let `R` be a `COrdField`.

```
Syntactic Definition ZeroR := (Zero::R).
Syntactic Definition OneR := (One::R).
```

19.10 Properties of `AbsSmall`.

```
Definition absSmall [R:COrdField;e,x:R]: Prop := ([--]e [:<] x) /\ (x [:<] e).
```

Convention: Let `R` be a `COrdField`.

```
Lemma AbsSmall_wd_rht : (rel_well_def_rht R AbsSmall).
```

```
Lemma AbsSmall_wd_rht_unfolded : (x,y,z:R)(AbsSmall x y)->(y[=]z)->
                                 (AbsSmall x z).
```

```
Lemma AbsSmall_wd_lft : (rel_well_def_lft R AbsSmall).
```

```
Lemma AbsSmall_wd_lft_unfolded : (x,y,z:R)(AbsSmall x y)->(x[=]z)->
                                 (AbsSmall z y).
```

```
Syntactic Definition ZeroR := (Zero::R).
```

```
Lemma AbsSmall_leEq_trans :  (e1,e2,d:R)(e1 [:<=] e2) ->
                             (AbsSmall e1 d) -> (AbsSmall e2 d).

Lemma AbsSmall_trans :  (e1,e2,d:R)(e1 [:<] e2) ->
                        (AbsSmall e1 d) -> (AbsSmall e2 d).

Lemma less_imp_AbsSmall : (e,d:R)(Zero [:<=] e) -> (e [:<] d) ->
                           (AbsSmall d e).

Lemma min_resp_AbsSmall: (x,y:R)(AbsSmall x y) -> (AbsSmall x [--]y).

Lemma AbsSmall_minus : (e,x1,x2:R)
                       (AbsSmall e (x1[-]x2)) -> (AbsSmall e (x2[-]x1)).

Lemma AbsSmall_plus : (e1,e2,x1,x2:R)
                      (AbsSmall e1 x1) -> (AbsSmall e2 x2) ->
                      (AbsSmall (e1[+]e2) (x1[+]x2)).

Lemma AbsSmall_eps_div_two : (e,x1,x2:R)
                             (AbsSmall (e[/]TwoNZ) x1) ->
                             (AbsSmall (e[/]TwoNZ) x2) ->
                             (AbsSmall e (x1[+]x2)).

Lemma AbsSmall_x_plus_delta : (x,eps,delta:R)
              (Zero [:<] eps) -> (Zero [:<] delta) -> (delta[:<]eps) ->
              (AbsSmall eps (x[-](x[+]delta))).

Lemma AbsSmall_x_minus_delta : (x,eps,delta:R)
                 (Zero [:<] eps) -> (Zero[:<] delta) -> (delta[:<]eps) ->
                 (AbsSmall eps (x[-](x[-]delta))).

Lemma AbsSmall_x_plus_eps_div2 : (x,eps:R)(Zero [:<] eps)->
                    (AbsSmall eps (x[-](x[+]eps[/]TwoNZ))).

Lemma AbsSmall_x_minus_eps_div2 : (x,eps:R)(Zero [:<] eps)->
                     (AbsSmall eps (x[-](x[-]eps[/]TwoNZ))).

Lemma AbsSmall_intermediate : (x,y,z,eps:R)(x [:<] y)->(y[:<]z)->
                              (AbsSmall eps (z[-]x)) -> (AbsSmall eps (z[-]y)).

Lemma AbsSmall_eps_div2 : (eps:R)(Zero [:<] eps)->
                          (AbsSmall eps (eps[/]TwoNZ)).

Lemma AbsSmall_pos: (e,x:R)(AbsSmall e x)->(Zero[:<]e).

Lemma AbsSmall_mult :(e1,e2,x1,x2: R)(AbsSmall e1 x1)->
(AbsSmall e2 x2)->(AbsSmall Three[*](e1[*]e2) x1[*]x2).

Lemma AbsSmall_cancel_mult:(e,x,z:R)(Zero[:<]z)->
(AbsSmall e[*]z x[*]z)->(AbsSmall e x).
```

19.11 Properties of one over successor

Convention: Let R be an ordered field.

```
Definition one_div_succ := [R:COrdField][i:nat](One [/] (Snring R i))
                         :  (R:COrdField)(nat->R).
```

```
Lemma one_div_succ_resp_leEq :
      (m,n:nat)(le m n)->((one_div_succ R n) [:<=] (one_div_succ R m)).
```

```
Lemma one_div_succ_pos : (i:nat)((Zero::R) [:<] (One_div_succ i)).
```

```
Lemma One_div_succ_resp_less : (i,j:nat)(lt i j)->
                              ((cof_less R) (One_div_succ j) (One_div_succ i)).
```

```
Lemma One_div_succ_resp_leEq : (i,j:nat)(le i j)->
                              ((leEq R) (One_div_succ j) (One_div_succ i)).
```

19.12 Properties of Half

```
Definition half := [R:COrdField]One::R[/]TwoNZ.
```

Convention: Let R be a COrdField

```
Lemma half_1 : (Half::R)[*]Two [=] One.
```

```
Lemma pos_half : ((Zero::R) [:<] Half).
```

```
Lemma half_1' : (x:R)((x[*]Half)[*]Two [=] x).
```

```
Lemma half_2 : (Half::R)[+]Half [=] One.
```

```
Lemma half_lt1 : (Half::R)[:<] One.
```

```
Lemma half_3 : (x:R)(Zero [:<] x) -> (Half[*]x [:<] x).
```

19.13 Cauchy

Convention: Let R be a COrdField.

```
Definition Cauchy_prop [g:nat -> R]: Prop :=
   (e:R)(Zero [:<] e) -> (Ex [N:nat](m:nat)(le N m)
   -> (AbsSmall e ((g m)[-](g N)))).
```

```
(* Def. CauchyP, Build_CauchyP *)
(* Should be defined in terms of CauchyP *)
(* Implicit arguments turned off, because Coq makes a mess of it in combination
   with the coercions *)
Record CauchySeq : Set :=
  {CS_seq  :> nat -> R;
   CS_proof: (Cauchy_prop CS_seq)
  }.
```

```
Definition seqLimit [seq:nat->R; lim:R]: Prop :=
   (e:R)(Zero [:<] e) -> (Ex [N:nat](m:nat)(le N m)
   -> (AbsSmall e ((seq m)[-]lim))).
```

```
Lemma estimate_sqrt :
  (a:R)(Zero [:<] a) ->
    (EX e | (Zero [:<] e) /\ (e[^](2) [:<] a)).
```

```
Lemma estimate_sqrt' :
  (a,b:R)(Zero [:<] a) -> (Zero [:<] b) ->
    (EX e | (Zero [:<] e) /\ (a[*]e[^](2) [:<] b)).
```

```
Lemma mult_abssmall'_rht :
  (x,y,C:R)(Zero [:<] C) ->
  ([--]C [:<] x) -> (x [:<] C) -> ([--]C [:<] y) -> (y [:<] C) ->
    (x[*]y [:<] Three[*]C[^](2)).
```

```
Lemma mult_abssmall_rht :
  (x,y,X,Y:R)(Zero [:<] X) -> (Zero [:<] Y) ->
  ([--]X [:<] x) -> (x [:<] X) -> ([--]Y [:<] y) -> (y [:<] Y) ->
    (x[*]y [:<] X[*]Y).
```

```
Lemma mult_abssmall_lft :
  (x,y,X,Y:R)(Zero [:<] X) -> (Zero [:<] Y) ->
  ([--]X [:<] x) -> (x [:<] X) -> ([--]Y [:<] y) -> (y [:<] Y) ->
    ([--](X[*]Y) [:<] x[*]y).
```

```
Lemma mult_abssmall :
  (x,y,X,Y:R)(AbsSmall X x) -> (AbsSmall Y y) -> (AbsSmall X[*]Y x[*]y).
```

20 Exponentiation

20.1 More properties about nexp

Convention: Let R be an ordered field.

```
Lemma nexp_resp_ap_zero : (x:R)(n:nat)(x[#]Zero)->(x[^]n [#] Zero).
```

```
Lemma nexp_distr_div : (x,y:R)(n:nat)(H:y[#]Zero)(H1:y[^]n[#]Zero)
                       (x[/]y[//]H) [^] n [=] (x[^]n) [/] (y[^]n) [//] H1.
```

```
Lemma nexp_distr_div' : (x,y:R)(n:nat)(H:y[#]Zero)
        (x[/]y[//]H) [^] n [=] (x[^]n) [/] (y[^]n) [//] (nexp_resp_ap_zero y n H).
```

```
Lemma small_nexp_resp_lt : (x:R)(m,n:nat)
            (Zero [:<] x) -> (x [:<] One) -> (lt m n) -> x[^]n [:<] x[^]m.
```

```
Lemma great_nexp_resp_lt : (x:R)(m,n:nat)
            (One [:<] x) -> (lt m n) -> x[^]m [:<] x[^]n.
```

```
Lemma small_nexp_resp_le : (x:R)(m,n:nat)
            (Zero [:<] x) -> (x [:<] One) -> (le m n) -> x[^]n [:<=] x[^]m.
```

```
Lemma great_nexp_resp_le : (x:R)(m,n:nat)
            (One [:<] x) -> (le m n) -> x[^]m [:<=] x[^]n.
```

```
Lemma nexp_resp_leEq :
                (x,y:R)(k:nat)(Zero [:<=] x)->(x [:<=] y)->(x[^]k [:<=] y[^]k).
```

```
Lemma nexp_distr_recip : (x:R)(n:nat)(H:x[#]Zero)(H1:x[^]n[#]Zero)
                         (One[/]x[//]H)[^]n [=] One[/](x[^]n)[//]H1.
```

20.2 Definition of `zexp` : integer exponentiation

Convention: Let `R` be an ordered field.

It would be nicer to define `zexp` using `caseZdiff`, but we already have most properties now.

```
Definition zexp[n:Z; x:(NonZeros R)] : R :=
         Cases n of
           ZERO => (One::R)
         | (POS p) => (NZinj x) [^] (convert p)
         | (NEG p) => (One[/]x) [^] (convert p)
         end.
```

Notation: `a [^^] b` stands for `zexp ? a b`.

20.3 Properties of `zexp`

Convention: Let `R` be an ordered field.

```
Lemma zexp_zero : (x:R)(H:x[#]Zero) ((x[//]H) [^^] `0`) [=] One.
```

```
Lemma zexp_nexp : (x:R)(H:x[#]Zero)(n:nat)
                  ((x[//]H) [^^] n) [=] (x [^] n).
```

```
Lemma zexp_min_nexp : (x:R)(H:x[#]Zero)(n:nat)
                      ((x[//]H) [^^] `-n`) [=] ((One[/]x[//]H) [^] n).
```

```
Lemma zexp_min_nexp' : (x:R)(n:nat)(H:x[#]Zero)(H1:x[^]n[#]Zero)
                      ((x[//]H) [^^] `-n`) [=] (One[/](x [^] n)[//]H1).

Lemma zexp_strong_ext : (x,y:R)(m:Z)(Hx:x[#]Zero)(Hy:y[#]Zero)
                      ((x[//]Hx)[^^]m [#] (y[//]Hy)[^^] m) ->
                      x [#] y.

Lemma zexp_well_def : (x,y:R)(m:Z)(Hx:x[#]Zero)(Hy:y[#]Zero)(x [=] y) ->
                      ((x[//]Hx)[^^]m [=] (y[//]Hy)[^^] m).

Lemma zexp_plus1 : (x:R)(H:x[#]Zero)(m:Z)
                   ((x[//]H) [^^] `m+1`) [=] ((x[//]H) [^^] m) [*] x.

Lemma zexp_resp_ap_zero : (x:R)(m:Z)(H:x[#]Zero)((x[//]H) [^^] m) [#] Zero.

Lemma zexp_min : (x:R)(H:x[#]Zero)(m:Z)(Hexp: (x[//]H) [^^] m [#] Zero)
   ((x[//]H) [^^] `-m`) [=] One[/] ((x[//]H) [^^] m) [//] Hexp.

Lemma zexp_min1 : (x:R)(H:x[#]Zero)(m:Z)
                   ((x[//]H) [^^] `m-1`) [=] ((x[//]H) [^^] m) [/] x [//] H.

Lemma zexp_plus : (x:R)(H:x[#]Zero)(m,n:Z)
              ((x[//]H) [^^] `m+n`) [=] ((x[//]H) [^^] m [*] (x[//]H) [^^]n).

Lemma zexp_minus : (x:R)(H:x[#]Zero)(m,n:Z)(Hexp : (x[//]H) [^^] n [#] Zero)
   ((x[//]H) [^^] `m-n`) [=]
   (((x[//]H) [^^] m) [/] ((x[//]H) [^^] n) [//] Hexp).

Lemma one_zexp : (z:Z)(OneNZ [^^] z [=] (One::R)).

Lemma mult_zexp : (x,y:R)(z:Z)(Hx:x[#]Zero)(Hy:y[#]Zero)(Hp:x[*]y[#]Zero)
                  ((x[*]y) [//] Hp) [^^] z [=]
                  (x [//] Hx) [^^] z [*] (y [//] Hy) [^^] z.

Lemma zexp_mult :  (x:R)(m,n:Z)(Hx:x[#]Zero)(He:(x[//]Hx)[^^]m [#] Zero)
                  (x [//] Hx) [^^] `m*n` [=]
                  ((x [//] Hx) [^^] m [//] He) [^^] n.

Lemma zexp_two : (x:R)(Hx:x[#]Zero)(x[//]Hx) [^^] `2` [=] x[*]x.

Lemma min_zexp_even : (x:R)(m:Z)(Zeven m)->(Hx:x[#]Zero)(Hneg:[--]x[#]Zero)
                  ([--]x[//]Hneg) [^^] m [=] (x[//]Hx) [^^] m.

Lemma min_zexp_two : (x:R)(Hx:x[#]Zero)(Hneg:[--]x[#]Zero)
                     ([--]x[//]Hneg) [^^] `2` [=] (x[//]Hx) [^^] `2`.
```

```
Lemma min_zexp_odd : (x:R)(m:Z)(Zodd m)->(Hx:x[#]Zero)(Hneg:[--]x[#]Zero)
                 ([--]x[//]Hneg) [^^] m [=] [--]((x[//]Hx) [^^] m).

Lemma zexp_one : (x:R)(H:x[#]Zero)(x[//]H)[^^]`1` [=] x.

Lemma zexp_funny : (x,y:R)(Hx:x[#]Zero)(Hy:y[#]Zero)
                  (x[+]y) [*] (x[-]y) [=]
                  (x[//]Hx)[^^]`2` [-] (y[//]Hy)[^^]`2`.

Lemma zexp_funny' : (x,y:R)(Hx:x[#]Zero)(Hy:y[#]Zero)
                   (x[-]y) [*] (x[+]y) [=]
                   (x[//]Hx)[^^]`2` [-] (y[//]Hy)[^^]`2`.

Lemma zexp_pos : (x:R)(H:x[#]Zero)(z:Z)(Zero [:<] x) ->
                (Zero [:<] (x[//]H) [^^] z).

Lemma root_unique :
  (x,y:R)(Zero [:<=] x) -> (Zero [:<=] y) -> (n:nat)(lt (0) n) ->
    (x[^]n [=] y[^]n) -> (x [=] y).

Lemma root_one :
  (x:R)(Zero [:<=] x) -> (n:nat)(lt (0) n) -> (x[^]n [=] One) -> (x [=] One).
```

21 Reals

21.1 Definition of the notion of reals

```
Record is_CReals [R: COrdField; lim : (CauchySeq R) -> R] : Prop :=
  {ax_Lim  : (s:(CauchySeq R))(SeqLimit s (lim s));
   ax_Arch : (x:R)(Ex [n:nat](x [:<] (Nring n)))
  }.

Record CReals : Type :=
  {crl_crr   :> COrdField;
   crl_lim   :  (CauchySeq crl_crr) -> crl_crr;
   crl_proof :  (is_CReals crl_crr crl_lim)
  }.

Definition lim := crl_lim : (IR:CReals)(CauchySeq IR) -> IR.
```

Convention: Let IR be a structure for real numbers.

Convention: With ZeroR and OneR we denote the zero and one of this structure.

21.2 CReals axioms

```
Lemma CReals_is_CReals : (is_CReals IR Lim).

Lemma Lim_Cauchy : (s:(CauchySeq IR))(SeqLimit s (Lim s)).

Lemma Archimedes : (x:IR)(Ex [n:nat](x [:<] (Nring n))).

Definition CauchySeqR := (CauchySeq IR).
```

21.3 Cauchy definitions

This section gives several definitions of Cauchy sequences. There are no lemmas in this section.

1. The current definition of Cauchy (`Cauchy_prop`) is:

$$\forall \epsilon > 0. \exists N. \forall m \geq N. |seq_m - seq_N| < \epsilon$$

2. Variant 1 of Cauchy (`Cauchy_prop1`) is:

$$\forall k. \exists N. \forall m \geq N. |seq_m - seq_N| < 1/(k+1)$$

3. In all of the following variants the limit occurs in the definition. Therefore it is useful to define an auxiliary predicate `Cauchy_lim_prop?`, in terms of which `Cauchy_prop?` is defined.

   Variant 2 of Cauchy (`Cauchy_prop2`):

   $\exists y.$`Cauchy_lim_prop2` $seq\ y$ where
   `Cauchy_lim_prop2` $seq\ y := \forall \epsilon > 0. \exists N. \forall m \geq N. |seq_m - y| < \epsilon$

   Note that `Cauchy_lim_prop2` equals `seqLimit`.

4. Variant 3 of Cauchy (`Cauchy_prop3`):

   $\exists y.$`Cauchy_lim_prop3` $seq\ y$ where
   `Cauchy_lim_prop3` $seq\ y := \forall k. \exists N. \forall m \geq N. |seq_m - y| < 1/(k+1)$

5. The following variant is more restricted.
   Variant 4 of Cauchy (`Cauchy_prop4`):

   $\exists y.$`Cauchy_lim_prop4` $seq\ y$ where
   `Cauchy_lim_prop4` $seq\ y := \forall k. |seq_k - y| < 1/(k+1)$

So

`Cauchy_prop4` $\Rightarrow$ `Cauchy_prop3` $\Leftrightarrow$ `Cauchy_prop2` $\Leftrightarrow$ `Cauchy_prop1` $\Leftrightarrow$ `Cauchy_prop`

Note: I don't know which formulations are useful.

```
Definition Cauchy_prop1 := [seq:(nat->IR)]
                           (k:nat)(Ex [N:nat](m:nat)((le N m) ->
                           (AbsSmall (One_div_succ k) ((seq m) [-] (seq N))))).


Definition Cauchy_lim_prop2 := [seq:(nat->IR)][y:IR]
                               (eps:IR)(Zero [:<] eps) ->
                               (Ex [N:nat](m:nat)((le N m) ->
                               (AbsSmall eps ((seq m) [-] y)))).


Definition Cauchy_prop2 := [seq:(nat->IR)]
                           (Ex [y:IR](Cauchy_lim_prop2 seq y)).


Definition Cauchy_lim_prop3 := [seq:(nat->IR)][y:IR]
                               (k:nat)
                               (Ex [N:nat](m:nat)((le N m) ->
                               (AbsSmall (One_div_succ k) ((seq m) [-] y)))).
```

```
Definition Cauchy_prop3 := [seq:(nat->IR)]
                            (Ex [y:IR](Cauchy_lim_prop3 seq y)).


Definition Cauchy_lim_prop4 := [seq:(nat->IR)][y:IR]
                          (((m:nat)(AbsSmall (One_div_succ m) ((seq m)[-]y)))).


Definition Cauchy_prop4 := [seq:(nat->IR)]
                            (Ex [y:IR](Cauchy_lim_prop4 seq y)).
```

21.4 Inequalities of limits

The next lemma is equal to lemma Lim_Cauchy.

```
Lemma Cauchy_complete : (seq:(CauchySeq IR))
                         (Cauchy_lim_prop2 seq (Lim seq)).


Lemma less_Lim_so_less_seq : (seq:(CauchySeq IR))(y:IR)(y [:<] (Lim seq)) ->
             (Ex [N:nat](m:nat)(le N m)->(y [:<] (seq m))).


Lemma Lim_less_so_seq_less : (seq:(CauchySeq IR))(y:IR)((Lim seq) [:<] y) ->
                              (Ex [N:nat](m:nat)(le N m)->((seq m) [:<] y)).


Lemma Lim_less_Lim_so_seq_less_seq :
               (seq1,seq2:(CauchySeq IR))((Lim seq1) [:<] (Lim seq2)) ->
               (Ex [N:nat](m:nat)(le N m)->((seq1 m) [:<] (seq2 m))).
```

The next lemma follows from less_Lim_so_less_seq with $y := (y + (\texttt{Lim}\ seq))/2$.

```
Lemma less_Lim_so : (seq:(CauchySeq IR))(y:IR)(y [:<] (Lim seq)) ->
             (Ex [eps:IR](Zero [:<] eps) /\
                (Ex [N:nat](m:nat)(le N m)->((y[+]eps) [:<] (seq m)))).


Lemma Lim_less_so : (seq:(CauchySeq IR))(y:IR)((Lim seq) [:<] y) ->
             (Ex [eps:IR](Zero [:<] eps) /\
               (Ex [N:nat](m:nat)(le N m)->(((seq m)[+]eps) [:<] y))).


Lemma leEq_seq_so_leEq_lim : (seq:CauchySeqR)(y:IR)((i:nat)y[:<=](seq i)) ->
                                                (y[:<=](Lim seq)).


Lemma Lim_leEq_Lim : (seq1,seq2:CauchySeqR)((i:nat)(seq1 i)[:<=](seq2 i)) ->
                                              ((Lim seq1)[:<=](Lim seq2)).


Lemma seq_leEq_so_lim_leEq : (seq:CauchySeqR)(y:IR)((i:nat)(seq i)[:<=]y) ->
                                       ((Lim seq)[:<=] y).


Lemma limits_unique : (seq:(CauchySeq IR))(y:IR)
                       (Cauchy_lim_prop2 seq y) -> (y [=] (Lim seq)).
```

21.5 Equivalence of formulations of Cauchy

```
Lemma Cauchy_prop1_prop : (seq:(nat->IR))
                          (Cauchy_prop1 seq) -> (Cauchy_prop seq).

Lemma Cauchy_prop2_prop : (seq:(nat->IR))
                          (Cauchy_prop2 seq) -> (Cauchy_prop seq).

Lemma Cauchy_lim_prop3_prop2 : (seq:(nat->IR))(y:IR)
                         (Cauchy_lim_prop3 seq y) -> (Cauchy_lim_prop2 seq y).

Lemma Cauchy_prop3_prop2 : (seq:(nat->IR))
                          (Cauchy_prop3 seq) -> (Cauchy_prop2 seq).

Lemma Cauchy_prop3_prop : (seq:(nat->IR))
                          (Cauchy_prop3 seq) -> (Cauchy_prop seq).

Definition Build_CauchySeq1 : (seq:(nat->IR))
                         (Cauchy_prop1 seq) -> CauchySeqR.

Lemma Cauchy_lim_prop4_prop3 : (seq:(nat->IR))(y:IR)
                        (Cauchy_lim_prop4 seq y) -> (Cauchy_lim_prop3 seq y).

Lemma Cauchy_lim_prop4_prop2 : (seq:(nat->IR))(y:IR)
                        (Cauchy_lim_prop4 seq y) -> (Cauchy_lim_prop2 seq y).

Lemma Cauchy_prop4_prop3 : (seq:(nat->IR))
                        (Cauchy_prop4 seq) -> (Cauchy_prop3 seq).

Lemma Cauchy_prop4_prop : (seq:(nat->IR))
                             (Cauchy_prop4 seq) -> (Cauchy_prop seq).

Definition Build_CauchySeq4 : (seq:(nat->IR))
                         (Cauchy_prop4 seq) -> CauchySeqR.

Definition Build_CauchySeq4_y : (seq:(nat->IR))(y:IR)
                           (Cauchy_lim_prop4 seq y) -> CauchySeqR.

Lemma Lim_CauchySeq4 : (seq:(nat->IR))(y:IR)
                       (H:(Cauchy_lim_prop4 seq y))
                       ((Lim (Build_CauchySeq4_y seq y H)) [=] y).

Definition Build_CauchySeq2 : (seq:(nat->IR))
                         (Cauchy_prop2 seq) -> CauchySeqR.

Definition Build_CauchySeq2_y : (seq:(nat->IR))(y:IR)
                           (Cauchy_lim_prop2 seq y) -> CauchySeqR.

Lemma Lim_CauchySeq2 : (seq:(nat->IR))(y:IR)
                       (H:(Cauchy_lim_prop2 seq y))
                       ((Lim (Build_CauchySeq2_y seq y H)) [=] y).
```

21.6 Properties of Cauchy sequences

```
Lemma Cauchy_lim_sum : (seq1,seq2:(nat->IR))(y1,y2:IR)
                (Cauchy_lim_prop2 seq1 y1) -> (Cauchy_lim_prop2 seq2 y2) ->
                (Cauchy_lim_prop2 ([n:nat]((seq1 n) [+] (seq2 n))) (y1[+]y2)).


Lemma Cauchy_sum : (seq1,seq2:CauchySeqR)
                   (Cauchy_prop ([n:nat]((seq1 n) [+] (seq2 n)))).


Lemma Lim_sum : (seq1,seq2:CauchySeqR)
(Lim (Build_CauchySeq IR ([n:nat]((seq1 n) [+] (seq2 n)))
                         (Cauchy_sum seq1 seq2)))
[=] ((Lim seq1) [+] (Lim seq2)).


Lemma Cauchy_lim_minus : (seq:(nat->IR))(y:IR)
                (Cauchy_lim_prop2 seq y) ->
                (Cauchy_lim_prop2 ([n:nat]([--](seq n))) ([--]y)).


Lemma Cauchy_minus : (seq:CauchySeqR)
                     (Cauchy_prop ([n:nat]([--](seq n)))).


Lemma Lim_minus : (seq:CauchySeqR)
(Lim (Build_CauchySeq IR ([n:nat]([--] (seq n)))
                         (Cauchy_minus seq)))
[=] ([--](Lim seq)).


Lemma Cauchy_lim_min : (seq1,seq2:(nat->IR))(y1,y2:IR)
                (Cauchy_lim_prop2 seq1 y1) -> (Cauchy_lim_prop2 seq2 y2) ->
                (Cauchy_lim_prop2 ([n:nat]((seq1 n) [-] (seq2 n))) (y1[-]y2)).


Lemma Cauchy_min : (seq1,seq2:CauchySeqR)
                   (Cauchy_prop ([n:nat]((seq1 n) [-] (seq2 n)))).


Lemma Lim_min : (seq1,seq2:CauchySeqR)
(Lim (Build_CauchySeq IR ([n:nat]((seq1 n) [-] (seq2 n)))
                         (Cauchy_min seq1 seq2)))
[=] ((Lim seq1) [-] (Lim seq2)).


Lemma Cauchy_lim_mult : (seq1,seq2:(nat->IR))(y1,y2:IR)
                (Cauchy_lim_prop2 seq1 y1) -> (Cauchy_lim_prop2 seq2 y2) ->
                (Cauchy_lim_prop2 ([n:nat]((seq1 n) [*] (seq2 n))) (y1[*]y2)).


Lemma Cauchy_mult : (seq1,seq2:CauchySeqR)
                   (Cauchy_prop ([n:nat]((seq1 n) [*] (seq2 n)))).


Lemma Lim_mult : (seq1,seq2:CauchySeqR)
(Lim (Build_CauchySeq IR ([n:nat]((seq1 n) [*] (seq2 n)))
                         (Cauchy_mult seq1 seq2)))
[=] ((Lim seq1) [*] (Lim seq2)).
```

21.7 Maximum

Let us first consider the informal definition of maximum.

Definition 21.1 *Given $x, y \in \mathbb{R}$, we define the maximum of x and y. First construct a sequence $(y'_i)_{i \in \mathbb{N}}$ such that*

$$\forall i \in \mathbb{N}[-\frac{1}{i} < y - y'_i < \frac{1}{i} \wedge y'_i \# x].$$

This is is possible (Lemma `exis_AFS` *below). Note that $(y'_i)_{i \in \mathbb{N}}$ is a Cauchy sequence and $y = \lim_{i\to\infty} y'_i$. Now define the sequence $(s_i)_{i \in \mathbb{N}}$ by*

$$s_i := \begin{cases} x & \text{if} \quad x > y'_i, \\ y'_i & \text{if} \quad x < y'_i, \end{cases}$$

Now $(s_i)_{i \in \mathbb{N}}$ is a Cauchy sequence and we define

$$max(x, y) := \lim_{i\to\infty} s_i.$$

First we will consider the "apart from" sequence y'_i, then we define the maximum function, and then prove its properties.

21.7.1 The "apart from" sequence

Given an x and y, the "apart from" sequence is a Cauchy sequence y'_i with limit y and with y'_i apart from x for all i. Below, this "apart from sequence" y' is called `AFS`.

Convention: Let `x` and `y` be reals.

```
Lemma Inform_cotrans_less_unfolded
     : (R:COrdField; a,b:R)
        (a [:<] b)->(c:R){a [:<] c}+{c [:<] b}.
```

```
Lemma exis_AFS : (eps:IR)(Zero [:<] eps) ->
                    (sig IR [y':IR]((AbsSmall eps (y' [-] y)) /\ (y' [#] x))).
```

Now we construct the sequence y'_i, with name `AFS`.

```
Definition AFS := [i:nat](proj1_sig ? ? (exis_AFS (One_div_succ i)
                         (one_div_succ_pos ? i)))
               :  nat->IR.
```

```
Lemma AFS_prop :
     (i:nat)((AbsSmall (One_div_succ i) ((AFS i)[-]y)) /\ ((AFS i) [#] x)).
```

```
Lemma AFS_ap : (i:nat)((AFS i) [#] x).
```

`AFS x y` is a Cauchy-sequence, and has limit `y`.

```
Lemma AFS_Cauchy_lim : (Cauchy_lim_prop4 AFS y).
```

```
Lemma Cauchy_prop_AFS : (Cauchy_prop AFS).
```

Now we make `AFCS` which is a *Cauchy* sequence.

```
Definition AFCS : (CauchySeq IR).
```

```
Lemma Lim_AFCS : (Lim AFCS) [=] y.
```

```
Lemma AFCS_ap : (i:nat)((AFCS i) [#] x).
```

21.7.2 The actual max function

Convention: Let `x` and `y` be reals (we will define the maximum of `x` and `y`).

```
Lemma Inform_ap_imp_less : (a,b:IR)((a [#] b) -> {a[:<]b} + {b[:<]a}).
```

`Max_seq i` corresponds with s_i in informal Definition 21.1.

```
Definition Max_seq : nat->IR.
```

```
Lemma Max_is_x : (i:nat)((AFS x y i) [:<]x) -> (Max_seq i) = x.
```

```
Lemma Max_is_AFS_i : (i:nat)(x[:<](AFS x y i)) -> (Max_seq i) = (AFS x y i).
```

```
Lemma Cauchy_Max_seq : (Cauchy_prop Max_seq).
```

```
Definition Max_CauchySeq : CauchySeqR.
```

```
Definition MAX : IR.
```

Constructively, the elementary properties of the maximum function are:

- $x \leq \max(x, y)$,
- $x \leq \max(x, y)$,
- $z < \max(x, y) \Rightarrow z < x \vee z < y$.

(This can be more concisely expressed as $z < \max(x, y) \Leftrightarrow z < x \vee z < y$). From these elementary properties we can prove all other properties, including strong extensionality. With strong extensionality, we can make the binary *operation* `Max`. (So `Max` is `MAX` coupled with some proofs.)

```
Lemma lft_leEq_MAX : x[:<=]MAX.
```

```
Lemma rht_leEq_MAX : y[:<=]MAX.
```

```
Lemma less_MAX_imp : (z:IR)(z[:<]MAX) -> (z[:<]x) \/ (z[:<]y).
```

```
Lemma MAX_strong_ext : (bin_op_strong_ext ? MAX).
```

```
Lemma MAX_well_def : (bin_op_well_def IR MAX).
```

21.7.3 Properties of Max

```
Definition Max := (Build_CSetoid_bin_op ? MAX MAX_well_def MAX_strong_ext).

Lemma lft_leEq_Max : (x,y:IR)x [:<=] (Max x y).

Lemma rht_leEq_Max : (x,y:IR)y [:<=] (Max x y).

Lemma less_Max_imp : (x,y,z:IR)(z[:<](Max x y)) -> (z[:<]x) \/ (z[:<]y).

Lemma Max_leEq : (x,y,z:IR)(x[:<=]z) -> (y[:<=]z) -> (Max x y)[:<=]z.

Lemma Max_less : (x,y,z:IR)(x [:<] z) -> (y [:<] z) -> (Max x y) [:<] z.

Lemma equiv_imp_eq_max : (x,x',m:IR)
  ((y:IR)(x [:<] y) -> (x' [:<] y) -> (m [:<] y)) ->
  ((y:IR)(m [:<] y) -> (x [:<] y)) ->
  ((y:IR)(m [:<] y) -> (x' [:<] y)) ->
    (Max x x') [=] m.

Lemma Max_id : (x:IR)(Max x x) [=] x.

Lemma Max_comm : (x,y:IR)(Max x y)[=](Max y x).
```

Lemma 21.2 *$\forall x, y \in \mathbb{R}[x \leq y \leftrightarrow max(x, y) = y]$.*

```
Lemma leEq_imp_Max_is_rht : (x,y:IR)(x [:<=] y) -> (Max x y) [=] y.

Lemma Max_is_rht_imp_leEq : (x,y:IR)((Max x y) [=] y) -> (x [:<=] y).
```

21.8 Absolute value

```
Definition ABSIR := [x:IR](Max x ([--]x))
                 :  IR -> IR.

Lemma ABSIR_strong_ext : (un_op_strong_ext ? ABSIR).

Lemma ABSIR_well_def : (un_op_well_def ? ABSIR).

Definition AbsIR := (Build_CSetoid_un_op ? ABSIR ABSIR_well_def ABSIR_strong_ext)
                 :  (CSetoid_un_op IR).

Lemma abs_wdIR: (x,y:IR) (x[=]y)->(AbsIR x) [=] (AbsIR y).

Lemma Abs_wd_lft_unfolded : (x,y,e:IR)(x[=]y) -> ((AbsIR x)[:<]e) ->
((AbsIR y)[:<]e).

Lemma Abs_wd_rht_unfolded : (x,y,e:IR)(x[=]y) -> (e[:<](AbsIR x)) ->
(e[:<](AbsIR y)).

Lemma AbsIRz_isz :(AbsIR Zero)[=]Zero.

Lemma abs_eq_abssmall :(x,e:IR)([--]e [:<] x) -> (x [:<] e) ->
(AbsSmall e x).

Lemma abs_eq_abssmall2 :(x,e:IR)(Zero[:<]e) -> ((AbsIR x) [:<] e) ->
(AbsSmall e x).
```

22 Functions on Reals

22.1 Continuity

Let `f` be a unary setoid operation on `IR` and let `f2` be a binary setoid operation on `IR`.

We use the following notations for intervals. `Intclr a b` for the closed interval $[a, b]$, `Intolr a b` for the open interval $<a, b>$, `Intcl a` for the left-closed interval $[a, \infty>$, `Intol a` for the left-open interval $<a, \infty>$, `Intcr b` for the right-closed interval $< -\infty, b]$.

Intervals like $[a, b]$ are defined for arbitrary reals a, b (being $\emptyset$ for $a > b$).

```
Definition Intclr [a,b:IR; x:IR] : Prop :=
  (a [:<=] x) /\ (x [:<=] b).
Definition Intolr [a,b:IR; x:IR] : Prop :=
  (a [:<] x) /\ (x [:<] b).
Definition Intol [a:IR; x:IR] : Prop :=
  (a [:<] x).
Definition Intcl [a:IR; x:IR] : Prop :=
  (a [:<=] x).
Definition Intcr [b:IR; x:IR] : Prop :=
   (x [:<=] b).
```

The limit of $f(x)$ as x goes to $p = l$, for both unary and binary functions:
The limit of f in p is l if $\forall e \in \mathbb{R}(0 < e) \to \exists d \in \mathbb{R}(0 < d) / \ \forall x \in \mathbb{R}(-d < p - x < d <) \to (AbsSmall - e < l - f(x) < e)$

```
Definition funLim [p,l:IR] : Prop :=
  (e:IR)(Zero [:<] e)->
    (Ex [d:IR] (Zero [:<] d) /\
               (x:IR)(AbsSmall d (p[-]x)) -> (AbsSmall e (l[-](f x)))).


Definition funLim_Cauchy [p,l:IR] : Prop :=
  (s:CauchySeqR)((Lim s)[=] p) ->
     (e:IR)(Zero [:<] e) -> (Ex [N:nat](m:nat)(le N m)
   -> (AbsSmall e ((f(s m))[-]l))).
```

The limit of f in (p, p') is l if $\forall e \in \mathbb{R}(0 < e) \to \exists d \in \mathbb{R}(0 < d) / \ \forall x \in \mathbb{R}(-d < p - x < d <) \to (-d < p' - y < d <) \to (AbsSmall - e < l - f(x, y) < e)$

```
Definition funLim2 [p,p',l:IR] : Prop :=
  (e:IR)(Zero [:<] e)->
    (Ex [d:IR] (Zero [:<] d) /\
               (x,y:IR)(AbsSmall d (p[-]x)) -> (AbsSmall d (p'[-]y))
                        ->(AbsSmall e (l[-](f2 x y)))).
```

The function f is continuous at p if the limit of $f(x)$ as x goes to p is $f(p)$. This is the ϵ/δ definition. We also give the defintion with limits of Cauchy sequences.

```
Definition continAt [p:IR]: Prop := (funLim p (f p)).
Definition continAtCauchy [p:IR]: Prop := (funLim_Cauchy p (f p)).
Definition continAt2 [p,q:IR]: Prop := (funLim2 p q (f2 p q)).


Ax_iom continAt_prop1 :(p:IR)(continAt p)->(continAtCauchy p).
```

```
Definition contin :Prop :=
  (x:IR)(continAt x).
Definition continCauchy :Prop :=
  (x:IR)(continAtCauchy x).
Definition contin2 :Prop :=
  (x,y:IR)(continAt2 x y).
```

Continuous on a closed, resp. open, resp. left open, resp. left closed interval

```
Definition continOnc [a,b:IR]: Prop :=
  (x:IR)(Intclr a b x) -> (continAt x).
Definition continOno [a,b:IR]: Prop :=
  (x:IR)(Intolr a b x) -> (continAt x).
Definition continOnol [a:IR]: Prop :=
  (x:IR)(Intol a x) -> (continAt x).
Definition continOncl [a:IR]: Prop :=
  (x:IR)(Intcl a x) -> (continAt x).
```

If $\lim_{x->p}(fx) = l$, then for every sequence p_n whose limit is p, $\lim_{n->\infty} f(p_n) = l$.

```
Lemma funLim_SeqLimit:
  (p,l:IR)(fl:(funLim p l))
    (pn:nat->IR)(sl:(SeqLimit pn p)) (SeqLimit ([n:nat](f (pn n))) l).
```

Now the same Lemma in terms of Cauchy sequences: if $\lim_{x->p}(fx) = l$, then for every Cauchy sequence s_n whose limit is p, $\lim_{n->\infty} f(s_n) = l$.

```
Ax_iom funLim_isCauchy:
  (p,l:IR)(funLim p l)->(s:CauchySeqR)((Lim s)[=] p) ->
(e:IR)(Zero [:<] e) -> (Ex [N:nat](m:nat)(le N m)
   -> (AbsSmall e ((s m)[-](s N)))).


Definition str_monot  := (x,y:IR)(x[:<]y)->((f x)[:<](f y)).


Definition str_monotOnc  := [a,b:IR]
         (x,y:IR)(Intclr a b x)->(Intclr a b y)
                  ->(x[:<]y)->((f x)[:<](f y)).


Definition str_monotOncl  := [a:IR]
         (x,y:IR)(Intcl a x)->(Intcl a y)
                  ->(x[:<]y)->((f x)[:<](f y)).


Definition str_monotOnol  := [a:IR]
         (x,y:IR)(Intol a x)->(Intol a y)
                  ->(x[:<]y)->((f x)[:<](f y)).
```

$\forall c \in \mathbb{R}(f$ strongly monotonic on $[c, \infty >) \to \forall a, b \in \mathbb{R}(c < a) \to (c < b) \to \ (f(a) < 0) \to \ (0 :< f(b)) \to \forall x, y \in \mathbb{R}(a \leq x \leq b) \to (a \leq y \leq b) \to (x < y) \to \exists z \in \mathbb{R}(x \leq z \leq y) \wedge (f(z) \# 0))$

```
Ax_iom strmon_ivt_prem : (c:IR)(str_monotOncl c) ->
  (a,b:IR)(Intcl c a)->(Intcl c b)-> ((f a)[:<] Zero) -> (Zero [:<](f b))
         -> (x,y:IR)(Intclr a b x)->(Intclr a b y) -> (x[:<]y)
               ->(EX z:IR | (Intclr x y z)/\((f z)[#]Zero)).
```

$\forall c \in \mathbb{R}(f$ strongly monotonic on $[c, \infty >) \to \forall a, b \in \mathbb{R}(a < b) \to (c < a) \to (c < b) \to (f(a) < 0) \to (0 :< f(b)) \to \exists z \in \mathbb{R}(a \leq z \leq b) \wedge (f(z) = 0))$

```
Ax_iom strmoncl_imp_ivt : (c:IR)(str_monotOncl c)
            -> (a,b:IR)(a[:<]b)->(Intcl c a)->(Intcl c b)
                -> ((f a)[:<]Zero) -> (Zero [:<] (f b))
                    -> (EX z:IR | (Intclr a b z)/\ ((f z)[=]Zero)).
```

```
Record is_CMetricField [F: CField; abs : (CSetoid_fun F IR)] : Prop :=
 {ax_abs_gt_zero   : (x:F)(Zero[:<=] (abs x));
  ax_abs_resp_mult : (x,y:F)(abs(x[*]y)) [=](abs x)[*](abs y);
  ax_abs_triangle  : (x,y:F)(abs(x[+]y)) [:<=] (abs x)[+](abs y)
 }.
```

```
Record CMetricField : Type :=
  {cmf_crr   :> CField;
   cmf_abs   :  (CSetoid_fun cmf_crr IR);
   cmf_proof :  (is_CMetricField cmf_crr cmf_abs)
  }.
```

```
Definition MCauchy_prop [g:nat -> F]: Prop :=
   (e:IR)(Zero [:<] e) -> (Ex [N:nat](m:nat)(le N m)
   -> (MAbs((g m)[-](g N)))[:<]e ).
```

```
Record MCauchySeq : Set :=
  {MCS_seq  :> nat -> F;
   MCS_proof: (MCauchy_prop MCS_seq)
  }.
```

```
Definition MseqLimit [seq:nat->F; lim:F]: Prop :=
   (e:IR)(Zero [:<] e) -> (Ex [N:nat](m:nat)(le N m)
-> (MAbs((seq m)[-]lim))[:<]e ).
```

```
Definition is_MCauchyCompl  [lim : MCauchySeq -> F] : Prop :=
  (s:MCauchySeq)(MseqLimit s (lim s)).
```

23 Sums over Reals

Convention: Let c be a real.

Here we prove that

$$\Sigma_{m\leq i\leq n}\ c^k = \frac{c^{n+1}-c^m}{c-1}.$$

```
Lemma sum0_c_exp : (H:c[-]One[#]Zero)(m:nat)
        ((sum0 IR m [i:nat](c[^]i)) [=] ((c[^]m [-] One)[/](c [-] One)[//]H)).
```

```
Lemma sum_c_exp : (H:c[-]One[#]Zero)
                     (m,n:nat)((Sum m n [i:nat](c[^]i))
                     [=]((c[^](S n) [-] c[^]m)[/](c [-] One)[//]H)).
```

The following formulation is often more useful if $c < 1$.

```
Lemma sum_c_exp' : (H:One[-]c[#]Zero)
                      (m,n:nat)((Sum m n [i:nat](c[^]i))
        [=]((c[^]m [-] c[^](S n))[/](One [-] c)[//]H)).
```

24 Roots of Real Numbers

24.1 roots exist

```
Lemma nrootIR : (EX x | (Zero [:<=] x) /\ (x[^]n [=] c)).


(* nth roots exist *)
Lemma nroot': (x:IR)(k:nat)(Zero [:<=] x)->(lt O k)->
  (sig IR [y:IR]((Zero [:<=] y) /\ (y[^]k [=] x))).


Lemma nroot: (x:IR)(k:nat)(Zero [:<] x)->(lt O k)->
  (sig IR [y:IR]((Zero [:<=] y) /\ (y[^]k [=] x))).


Definition nroot_fun [x:IR; k:nat; posx:(Zero [:<] x); posk:(lt O k)] : IR :=
  (proj1_sig ?? (nroot x k posx posk)).


Lemma nroot_power : (x:IR; k:nat; posx:(Zero [:<] x); posk:(lt O k))
  (nroot_fun x k posx posk)[^]k [=] x.


Lemma nroot_pos : (x:IR; k:nat; posx:(Zero [:<] x); posk:(lt O k))
  Zero [:<] (nroot_fun x k posx posk).


Lemma nroot_power' : (x:IR; k:nat; posxk:(Zero [:<] x[^]k); posk:(lt O k))
  (Zero [:<=] x) -> (nroot_fun (x[^]k) k posxk posk) [=] x.


Lemma nroot_pres_lt :
(x,y:IR;k:nat;zltx:(Zero[:<]x);zlty:(Zero[:<]y);lt0k:(lt (0) k))
(x[:<]y)->((nroot_fun x k zltx lt0k) [:<] (nroot_fun y k zlty lt0k)).


Lemma nroot_pres_lt1 : (x:IR;k:nat;zltx:(Zero[:<]x);lt0k:(lt (0) k))
(x[:<]One)->((nroot_fun x k zltx lt0k) [:<] One).
```

24.2 square root

```
Definition sqrt [x:IR; xpos:(Zero [:<=] x)] : IR :=
  (proj1_sig ?? (nroot' x (2) xpos (lt_O_Sn (1)))).


Lemma sqrt_sqr : (x:IR)(xpos:(Zero [:<=] x))((sqrt x xpos)[^](2) [=] x).


Lemma sqrt_nonneg : (x:IR)(xpos:(Zero [:<=] x))(Zero [:<=] (sqrt x xpos)).


Lemma sqrt_wd : (x,y:IR)(xpos:(Zero [:<=] x))(ypos:(Zero [:<=] y))
  (x [=] y)->((sqrt x xpos) [=] (sqrt y ypos)).


Lemma sqrt_to_nonneg : (x:IR)(Zero [:<=] x)->(x2pos:(Zero [:<=] x[^](2)))
  ((sqrt x[^](2) x2pos) [=] x).
```

```
Lemma sqrt_to_nonpos : (x:IR)(x [:<=] Zero)->(x2pos:(Zero [:<=] x[^](2)))
  ((sqrt x[^](2) x2pos) [=] [--]x).


Lemma sqrt_mult : (x,y:IR)
  (xpos:(Zero [:<=] x))(ypos:(Zero [:<=] y))(xypos:(Zero [:<=] x[*]y))
    ((sqrt x[*]y xypos) [=] (sqrt x xpos)[*](sqrt y ypos)).


Lemma sqrt_mult_wd :
  (x,y,z:IR)(xpos:(Zero [:<=] x))(ypos:(Zero [:<=] y))(zpos:(Zero [:<=] z))
    (z [=] x[*]y) -> (sqrt z zpos) [=] (sqrt x xpos)[*](sqrt y ypos).


Lemma sqrt_less : (x,y:IR)(ypos:(Zero [:<=] y))
  (x[^](2) [:<] y)->(x [:<] (sqrt y ypos)).


Lemma sqrt_less' : (x,y:IR)(ypos:(Zero [:<=] y))
  (x[^](2) [:<] y)->([--]x [:<] (sqrt y ypos)).


Lemma AbsIR_sqrt_sqr : (x:IR; xxpos:(Zero [:<=] x[^](2)))
  (AbsIR x) [=] (sqrt x[^](2) xxpos).


Lemma abs_pres_multIR : (x,y:IR) (AbsIR (x[*]y)) [=] ((AbsIR x)[*](AbsIR y)).


(* Cauchy-Schwartz for IR *)
Lemma triangle_IR: (x,y:IR) (AbsIR (x[+]y)) [:<=] ((AbsIR x)[+](AbsIR y)).


Lemma triangle_sumIR : (k,l:nat)(s:nat->IR)(le k (S l)) ->
  (AbsIR (Sum k l s)) [:<=] (Sum k l [i:nat](AbsIR (s i))).
```

25 Complex Numbers

```
Record CC_set : Set :=
      {Re : IR;
       Im : IR}.

Definition cc_ap [x,y:CC_set]: Prop :=
  ((Re x) [#] (Re y)) \/ ((Im x) [#] (Im y)).
Definition cc_eq [x,y:CC_set]: Prop :=
  ((Re x) [=] (Re y)) /\ ((Im x) [=] (Im y)).

Lemma cc_is_CSetoid : (is_CSetoid ? cc_eq cc_ap).


Definition cc_csetoid := (Build_CSetoid CC_set cc_eq cc_ap cc_is_CSetoid).

Definition cc_plus : CC_set -> CC_set -> CC_set :=
  [x,y:CC_set](Build_CC_set ((Re x)[+](Re y)) ((Im x)[+](Im y))).

Definition cc_mult : CC_set -> CC_set -> CC_set :=
  [x,y:CC_set]
```

```
  (Build_CC_set ((Re x)[*](Re y)) [-] ((Im x)[*](Im y))
  ((Re x)[*](Im y)) [+] ((Im x)[*](Re y))).

Definition cc_zero := (Build_CC_set Zero::IR Zero::IR).
Definition cc_one := (Build_CC_set One::IR Zero::IR).
Definition cc_i := (Build_CC_set Zero::IR One::IR).
Definition cc_min [x:CC_set]: CC_set := (Build_CC_set ([--](Re x)) ([--](Im x))).

Lemma cc_plus_op_proof : (bin_op_well_def cc_csetoid cc_plus).


Lemma cc_mult_op_proof : (bin_op_well_def cc_csetoid cc_mult).


Lemma cc_min_op_proof : (un_op_well_def cc_csetoid cc_min).


Lemma cc_min_strext : (un_op_strong_ext cc_csetoid cc_min).


Lemma cc_plus_strext : (bin_op_strong_ext cc_csetoid cc_plus).


Lemma cc_mult_strext : (bin_op_strong_ext cc_csetoid cc_mult).


Definition cc_min_op :=
(Build_CSetoid_un_op cc_csetoid cc_min cc_min_op_proof cc_min_strext).
Definition cc_plus_op :=
(Build_CSetoid_bin_op cc_csetoid cc_plus cc_plus_op_proof cc_plus_strext).
Definition cc_mult_op :=
(Build_CSetoid_bin_op cc_csetoid cc_mult cc_mult_op_proof cc_mult_strext).

Lemma cc_csg_associative : (Associative cc_plus_op).


Lemma cc_cr_mult_associative : (Associative cc_mult_op).


Definition cc_csemi_grp :=
(Build_CSemi_grp cc_csetoid cc_zero cc_plus_op cc_csg_associative).

Lemma cc_cm_proof : (is_CMonoid cc_csemi_grp).


Definition cc_cmonoid :=
(Build_CMonoid ? cc_cm_proof).

Lemma cc_cg_proof : (is_CGroup cc_cmonoid cc_min_op).


Lemma cc_cr_dist : (Distributive cc_mult_op cc_plus_op).


Lemma cc_cr_non_triv : (cc_ap cc_one cc_zero).
```

```
Definition cc_cgroup :=
(Build_CGroup cc_cmonoid cc_min_op cc_cg_proof).

Lemma cc_cr_mult_mon :
(is_CMonoid (Build_CSemi_grp
       (csg_crr cc_cgroup) cc_one cc_mult_op cc_cr_mult_associative)).


Lemma cc_mult_commutes: (Commutes cc_mult_op).


Lemma cc_isCRing : (is_CRing  cc_cgroup cc_one cc_mult_op).


Definition cc_cring :=
(Build_CRing cc_cgroup cc_one cc_mult_op cc_isCRing) : CRing.
```

If $x \# 0$ or $y \# 0$, then $x^2 + y^2 \# 0$.

```
Lemma cc_inv_aid:
(x,y:IR)((x [#] Zero) \/ (y [#] Zero)) ->
(((x[^](2)) [+] (y[^] (2))) [#] Zero).
```

If $x \# 0$ or $y \# 0$, then $\frac{x}{x^2+y^2} \# 0$ or $\frac{-y}{x^2+y^2} \# 0$.

```
Lemma cc_inv_aid2:
(x,y:IR)(h:(x [#] Zero) \/ (y [#] Zero))
  ((x [/] (NZpro (cc_inv_aid ?? h))) [#] Zero) \/
  (([--]y [/] (NZpro (cc_inv_aid ?? h))) [#] Zero).


Definition cc_inv : (NonZeros cc_cring) -> (NonZeros cc_cring).


Lemma cc_Inv : (CSetoid_un_op (NonZeros cc_cring)).


Lemma cc_cfield_proof : (is_CField cc_cring cc_Inv).


Definition cc_cfield := (Build_CField cc_cring cc_Inv cc_cfield_proof).
Definition CC := cc_cfield.

Definition cc_set_CC : IR -> IR -> CC := Build_CC_set.
Definition cc_IR [x:IR]: CC := (cc_set_CC x Zero).

Definition conj : CC -> CC  :=
[z:CC_set](CC_set_rec [_:CC_set]CC_set
[Re0,Im0:IR](Build_CC_set Re0 [--]Im0) z).

Definition AbsCC [z:CC]: IR :=
(sqrt (((Re z)[^](2)) [+] ((Im z)[^](2)))
     (cc_abs_aid ? (Re z) (Im z))).


Lemma TwoCC_ap_zero : (Two::CC [#] Zero).


Lemma I_square : I[*]I [=] [--]One.
```

```
Lemma I_square' : I[^](2) [=] [--]One.

Lemma I_inv_lft : ([--]I)[*]I [=] One.

Lemma I_inv_rht : I[*]([--]I) [=] One.

Lemma mult_I : (x,y:IR)((x[+I*]y)[*]I [=] ([--]y)[+I*]x).

Lemma I_wd : (x,x',y,y':IR)
  (x [=] x')->(y [=] y')->(x[+I*]y [=] x'[+I*]y').

Lemma calculate_norm : (x,y:IR)
  ((x[+I*]y)[*](conj x[+I*]y) [=] (cc_IR x[^](2)[+]y[^](2))).

Lemma calculate_Re: (c:CC)((cc_IR (Re c))[*]Two [=] c[+](conj c)).

Lemma calculate_Im: (c:CC)((cc_IR (Im c))[*](Two[*]I) [=] c[-](conj c)).

Lemma Re_wd : (c,c':CC)(c [=] c')->((Re c) [=] (Re c')).

Lemma Im_wd : (c,c':CC)(c [=] c')->((Im c) [=] (Im c')).

Lemma Re_resp_plus :(x,y:CC)(Re (x[+]y)) [=] (Re x)[+](Re y).

Lemma Re_resp_min :(x,y:CC)(Re (x[-]y)) [=] (Re x)[-](Re y).

Lemma Im_resp_plus :(x,y:CC)(Im (x[+]y)) [=] (Im x)[+](Im y).

Lemma Im_resp_min :(x,y:CC)(Im (x[-]y)) [=] (Im x)[-](Im y).

Lemma cc_calculate_square : (x,y:IR)
  ((x[+I*]y)[^](2) [=] (x[^](2)[-]y[^](2))[+I*]((x[*]y)[*]Two)).

Lemma conj_1 : (c,c':CC)((conj c[+]c') [=] (conj c)[+](conj c')).

Lemma conj_2 : (c,c':CC)((conj c[*]c') [=] (conj c)[*](conj c')).

Lemma conj_4 : (c,c':CC)((conj c) [#] (conj c'))->(c [#] c').

Lemma conj_5 : (c:CC)((conj (conj c)) [=] c).

Lemma conj_6 : (conj Zero) [=] Zero.
```

```
Lemma conj_7 : (conj One) [=] One.

Lemma conj_3 : (c:CC)(n:nat)((conj c[^]n) [=] (conj c)[^]n).

Lemma cc_IR_1 : (x,y:IR) (x [=] y) -> (cc_IR x) [=] (cc_IR y).

Lemma cc_IR_1a : (x,y:IR) (x [#] y) -> (cc_IR x) [#] (cc_IR y).

Lemma cc_IR_2 : (x,y:IR) (cc_IR x)[*](cc_IR y) [=] (cc_IR x[*]y).

Lemma cc_IR_4a : (x,y,z:IR) x[+I*]y[*](cc_IR z) [=] (x[*]z)[+I*](y[*]z).

Lemma cc_IR_4b : (x,y,z:IR) (cc_IR z)[*]x[+I*]y [=] (z[*]x)[+I*](z[*]y).

Lemma cc_IR_5a : (x,y:IR) (cc_IR x)[+](cc_IR y) [=] (cc_IR x[+]y).

Lemma cc_IR_5b : (x,y:IR) (cc_IR x)[-](cc_IR y) [=] (cc_IR x[-]y).

Lemma cc_IR_zero : (cc_IR Zero) [=] Zero.

Lemma cc_IR_one : (cc_IR One) [=] One.

Lemma cc_IR_Nring : (n:nat)((cc_IR (Nring n)) [=] (Nring n)).

Lemma cc_IR_3 : (x:IR)(n:nat) (cc_IR x)[^]n [=] (cc_IR x[^]n).

Lemma AbsCC_wd : (x,y:CC)(x [=] y) -> ((AbsCC x) [=] (AbsCC y)).

Lemma cc_minus_abs :
(x:CC)(AbsCC ([--]x)) [=] (AbsCC x).

Lemma cc_min_abs :
(x,y:CC)(AbsCC (x[-]y)) [=] (AbsCC(y[-]x)).

Lemma cc_mult_abs :
(x,y:CC)(AbsCC (x[*]y)) [=] ((AbsCC x)[*](AbsCC y)).

Lemma AbsCC_minzero : (x:CC)(AbsCC(x[-]Zero))[=] (AbsCC x).

Lemma AbsCC_IR : (x:IR)(Zero [:<=] x) -> (AbsCC (cc_IR x)) [=] x.

Lemma cc_div_abs :
(x,y:CC)(nz:y[#]Zero)(anz:(AbsCC y)[#]Zero)
  (AbsCC (x[/]y[//]nz)) [=] (AbsCC x)[/](AbsCC y)[//]anz.
```

```
Lemma cc_div_abs' :
(x:CC)(y:IR)(nz:y[#]Zero)(cnz:(cc_IR y)[#]Zero)(Zero [:<=] y) ->
  (AbsCC (x[/](cc_IR y)[//]cnz)) [=] (AbsCC x)[/]y[//]nz.
```

```
Lemma AbsCC_zero : (AbsCC Zero) [=] Zero.
```

```
Lemma AbsCC_one : (AbsCC One) [=] One.
```

```
Lemma cc_pow_abs :
(x:CC; n:nat)(AbsCC (x[^]n)) [=] ((AbsCC x)[^]n).
```

```
Lemma AbsCC_pos : (x:CC)(x [#] Zero) -> (Zero [:<] (AbsCC x)).
```

```
Lemma AbsCC_ap_zero : (x:CC)(Zero [#] (AbsCC x)) -> (x [#] Zero).
```

```
Lemma AbsCC_nonneg : (x:CC)(Zero [:<=] (AbsCC x)).
```

```
Lemma AbsCC_small_imp_eq :
  (x:CC)((e:IR)(Zero [:<] e) -> (AbsCC x) [:<] e) -> x [=] Zero.
```

```
Lemma triangle : (x,y:CC)(AbsCC (x[+]y)) [:<=] (AbsCC x)[+](AbsCC y).
```

```
Lemma triangle_sum : (m,n:nat)(z:nat->CC)(le m (S n)) ->
  (AbsCC (Sum m n z)) [:<=] (Sum m n [i:nat](AbsCC (z i))).
```

26 Key Lemma

TO DO: Put next lemma in more appropriate place.

```
Lemma maj_upto_eps : (a:nat->IR; n:nat; eps:IR)
        (lt (0) n) -> (Zero [:<] eps)
        -> (EX k | (le (1) k) /\ (le k n) /\
                (i:nat)(le (1) i) -> (le i n) -> (a i)[-]eps [:<] (a k)).
```

Convention: Let `a : nat->IR`, `n : nat`, `(lt (0) n)`, `eps : IR`, `(Zero [:<] eps)`, `(k:nat)(Zero [:<=] (a k))`, `(a n) [=] One`, `a_0 : IR`, `(eps [:<] a_0)`.

```
Lemma a_0_eps_pos : Zero [:<] a_0 [-] eps.
```

```
Lemma a_0_eps_fuzz : a_0 [-] eps [:<] a_0.
```

```
Lemma lem_1a : (minus n (minus n (1))) = (1).
```

```
Lemma lem_1b : (n,j:nat)(le (minus n (S j)) (minus n j)).
```

```
Lemma lem_1c : (n,j:nat)(le (minus n j) n).
```

```
Lemma lem_1:
  (EX t | (Zero [:<] t) /\ (EX k | (le (1) k) /\ (le k n) /\
        ((a k)[*]t[^]k [=] a_0[-]eps) /\
        (i:nat)(le (1) i) -> (le i n) -> ((a i)[*]t[^]i [:<] a_0))).


Definition p3m := [i:nat](One[/]ThreeNZ)[^]i ::IR.


Lemma p3m_pos : (i:nat)(Zero [:<] (p3m i)).


Lemma p3m_S : (i:nat)((p3m (S i)) [=] (p3m i)[/]ThreeNZ).


Lemma p3m_P : (i:nat)((p3m i) [=] (p3m (S i))[*]Three).


Lemma p3m_aux : (i,j:nat)((p3m (S i))[^]j [=] (p3m j)[*](p3m i)[^]j).


Lemma p3m_pow : (i,j:nat)((p3m i)[^]j [=] (p3m (mult i j))).


Lemma p3m_0 : (p3m (0)) [=] One.


Lemma third_pos : Zero::IR [:<] One[/]ThreeNZ.


Lemma third_less_one : One[/]ThreeNZ [:<] One::IR.


Lemma p3m_mon : (i,j:nat)(lt i j) -> ((p3m j) [:<] (p3m i)).


Lemma p3m_mon' : (i,j:nat)(le i j) -> ((p3m j) [:<=] (p3m i)).


Lemma p3m_small : (i:nat)((p3m i) [:<=] One).


Lemma p3m_smaller : (i:nat)(lt (0) i)->((p3m i) [:<] Half).


Definition chfun [k:nat->nat; a,j:nat] : nat->nat := [i:nat]
  Cases (le_gt_dec i j) of
    (left _)  => (k i)
  | (right _) => a
  end.


Lemma chfun_1 :
  (k:nat->nat; a,j:nat)(i:nat)(le i j) -> (k i) = ((chfun k a j) i).


Lemma chfun_2 :
  (k:nat->nat; j:nat; a:nat)(i:nat)(lt j i) -> a = ((chfun k a j) i).
```

```
Lemma chfun_3 : (k:nat->nat; j:nat; a:nat)
        ((i:nat)(le (1) (k i)) /\ (le (k i) n)) ->
        (le (1) a) -> (le a n) ->
          (i:nat)(le (1) ((chfun k a j) i)) /\ (le ((chfun k a j) i) n).

Lemma chfun_4 : (k:nat->nat; j:nat; a:nat)
        ((i:nat)(le (k (S i)) (k i))) ->
        (le a (k j)) ->
          (i:nat)(le ((chfun k a j) (S i)) ((chfun k a j) i)).

Definition halfeps := Half[*]eps.

Lemma halfeps_pos : Zero [:<] halfeps.

Lemma halfeps_halfeps : (x:IR)((x[-]halfeps)[-]halfeps [=] x[-]eps).

Lemma halfeps_eps : (x,y:IR)(x[-]halfeps [:<] y) -> (x[-]eps [:<] y).

Lemma halfeps_trans :
  (x,y,z:IR)(x[-]halfeps [:<] y) -> (y[-]halfeps [:<] z) -> (x[-]eps [:<] z).

Lemma Key_1a :
(i,j:nat; a,t:IR)
  a[*](t[*](p3m (S j)))[^]i [=] (p3m i)[*](a[*](t[*](p3m j))[^]i).

Lemma Key_1b : (k:nat)(le (1) k) -> ((p3m k)[*]eps [:<] halfeps).

Lemma Key_1 :
(i,k,j:nat; ai,ak,t:IR)
  (le (1) k) -> (lt k i) -> (Zero [:<=] ai) -> (Zero [:<] t) ->
  (ai[*](t[*](p3m j))[^]i[-]eps [:<] ak[*](t[*](p3m j))[^]k) ->
    (ai[*](t[*](p3m (S j)))[^]i[-]halfeps [:<] ak[*](t[*](p3m (S j)))[^]k).

Lemma Key_2 :
(i,k,k',j:nat; ai,ak,ak',t:IR)
  (le (1) k) -> (lt k i) -> (Zero [:<=] ai) -> (Zero [:<] t) ->
  (ak[*](t[*](p3m (S j)))[^]k[-]halfeps [:<] ak'[*](t[*](p3m (S j)))[^]k') ->
  (ai[*](t[*](p3m j))[^]i[-]eps [:<] ak[*](t[*](p3m j))[^]k) ->
        (ai[*](t[*](p3m (S j)))[^]i[-]eps [:<] ak'[*](t[*](p3m (S j)))[^]k').

Lemma Key :
(EX t | (Zero [:<] t) /\
  (J : nat)
  (EX k : nat -> nat |
        ((i:nat)(le (1) (k i)) /\ (le (k i) n)) /\
        ((i:nat)(le (k (S i)) (k i))) /\
        (let k_0 = (k (0)) in
                (a k_0)[*]t[^]k_0 [=] a_0[-]eps) /\
        ((j:nat)(le j J) ->
          let k_j = (k j) in
          let r = t[*](p3m j) in
                (i:nat)(le (1) i) -> (le i n) ->
                        (a i)[*]r[^]i[-]eps [:<] (a k_j)[*]r[^]k_j))).
```

27 Main Lemma

TO DO: Put next lemma in more appropriate place.

```
Lemma kseq_prop :
(k:nat->nat; n:nat)
        ((i:nat)(le (1) (k i)) /\ (le (k i) n)) ->
        ((i:nat)(le (k (S i))(k i))) ->
          (EX j | (lt (S j) (mult (2) n)) /\ (k j) = (k (S j)) /\
                                                 (k (S j)) = (k (S (S j)))).
```

Convention: Let `a : nat->IR`, `n : nat`, `(lt (0) n)`, `eps : IR`, `(Zero [:<] eps)`, `(k:nat)(Zero [:<=] (a k))`, `(a n) [=] One`, `a_0 : IR`, `(eps [:<] a_0)`.

```
Lemma a_0_pos : Zero [:<] a_0.
```

```
Local two_n := (mult (2) n).
Local Small := (p3m n).
Local Smaller := (p3m (mult two_n n)).
```

```
Lemma Main_1a' :
(t:IR; j,k:nat)
  let r' = t[*](p3m (S (S j))) in
  let r = t[*](p3m (S j)) in
    ((i:nat)(le (1) i) -> (le i n) ->
      ((a i)[*]r'[^]i[-]eps [:<] (a k)[*]r'[^]k)) ->
    (i:nat)(le (1) i) -> (le i n) ->
      (a i)[*](r[/]ThreeNZ)[^]i[-]eps [:<] (a k)[*](r[/]ThreeNZ)[^]k.
```

```
Lemma Main_1b' :
(t:IR; j,k:nat)
  let r' = t[*](p3m j) in
  let r = t[*](p3m (S j)) in
    ((i:nat)(le (1) i) -> (le i n) ->
      ((a i)[*]r'[^]i[-]eps [:<] (a k)[*]r'[^]k)) ->
    (i:nat)(le (1) i) -> (le i n) ->
      (a i)[*](r[*]Three)[^]i[-]eps [:<] (a k)[*](r[*]Three)[^]k.
```

```
Lemma Main_1a :
(r:IR; k:nat)
  (Zero [:<] r) -> (le (1) k) -> (le k n) ->
  ((i:nat)(le (1) i) -> (le i n) ->
    ((a i)[*](r[/]ThreeNZ)[^]i[-]eps [:<] (a k)[*](r[/]ThreeNZ)[^]k)) ->
        let p_ = [i:nat](a i)[*]r[^]i in
        let p_k = (a k)[*]r[^]k in
          ((Sum (1) (pred k) p_)
                [:<] Half[*](One[-]Small)[*]p_k[+]Half[*]Three[^]n[*]eps).
```

```
Lemma Main_1b :
(r:IR; k:nat)
  (Zero [:<] r) -> (le (1) k) -> (le k n) ->
  ((i:nat)(le (1) i) -> (le i n) ->
    ((a i)[*](r[*]Three)[^]i[-]eps [:<] (a k)[*](r[*]Three)[^]k)) ->
        let p_ = [i:nat](a i)[*]r[^]i in
        let p_k = (a k)[*]r[^]k in
          ((Sum (S k) n p_)
                [:<] Half[*](One[-]Small)[*]p_k[+]Half[*]Three[^]n[*]eps).


Lemma Main_1 :
(r:IR; k:nat)
  (Zero [:<] r) -> (le (1) k) -> (le k n) ->
  ((i:nat)(le (1) i) -> (le i n) ->
    ((a i)[*](r[/]ThreeNZ)[^]i[-]eps [:<] (a k)[*](r[/]ThreeNZ)[^]k)) ->
  ((i:nat)(le (1) i) -> (le i n) ->
    ((a i)[*](r[*]Three)[^]i[-]eps [:<] (a k)[*](r[*]Three)[^]k)) ->
        let p_ = [i:nat](a i)[*]r[^]i in
        let p_k = (a k)[*]r[^]k in
          ((Sum (1) (pred k) p_) [+] (Sum (S k) n p_)
                [:<] (One[-]Small)[*]p_k[+]Three[^]n[*]eps).


Lemma Main_2' :
(t:IR; i,k:nat)
  ((a i)[*](t[*](p3m (0)))[^]i[-]eps [:<] (a k)[*](t[*](p3m (0)))[^]k) ->
    (a i)[*]t[^]i[-]eps [:<] (a k)[*]t[^]k.


Lemma Main_2 :
(t:IR; j,k:nat)
  let r = t[*](p3m j) in
        (Zero [:<] t) ->
        ((a k)[*]t[^]k [=] a_0[-]eps) ->
        ((i:nat)(le (1) i) -> (le i n) ->
          (a i)[*]t[^]i[-]eps [:<] (a k)[*]t[^]k) ->
        (i:nat)(le (1) i) -> (le i n) -> (a i)[*]r[^]i [:<] a_0.


Lemma Main_3a :
(t:IR; j,k,k_0:nat)
  let r = t[*](p3m j) in
        (le k_0 n) ->
        ((a k_0)[*]t[^]k_0 [=] a_0[-]eps) ->
        ((a k_0)[*]r[^]k_0[-]eps [:<] (a k)[*]r[^]k) ->
        (p3m (mult j n))[*]a_0[-]Two[*]eps [:<] (a k)[*]r[^]k.


Lemma Main_3 :
(t:IR; j,k,k_0:nat)
  let r = t[*](p3m j) in
        (lt j two_n) ->
        (le k_0 n) ->
        ((a k_0)[*]t[^]k_0 [=] a_0[-]eps) ->
        ((a k_0)[*]r[^]k_0[-]eps [:<] (a k)[*]r[^]k) ->
          (Smaller[*]a_0[-]Two[*]eps [:<] (a k)[*]r[^]k).
```

```
Lemma Main :
(EX r | (Zero [:<] r) /\ (EX k | (le (1) k) /\ (le k n) /\
      let p_ = [i:nat](a i)[*]r[^]i in
      let p_k = (a k)[*]r[^]k in
        ((Sum (1) (pred k) p_) [+] (Sum (S k) n p_)
                [:<] (One[-]Small)[*]p_k[+]Three[^]n[*]eps) /\
        (r[^]n [:<] a_0) /\
        (Smaller[*]a_0[-]Two[*]eps [:<] p_k) /\ (p_k [:<] a_0))).
```

28 Roots of Complex Numbers

28.1 CC ap Zero

```
Lemma cc_ap_zero : (P:CC->Prop)
  ((a,b:IR)(a [#] Zero)->(P a[+I*]b))->
  ((a,b:IR)(b [#] Zero)->(P a[+I*]b))->
    (c:CC)(c [#] Zero)->(P c).
```

28.2 Imag to Real

```
Lemma imag_to_real : (a,b,a',b':IR)(a'[+I*]b' [=] (a[+I*]b)[*]I)->
  (a [#] Zero)->(b' [#] Zero).
```

28.3 NRoot I

```
Definition sqrt_Half := (sqrt Half (less_leEq ??? (pos_half IR))).
Definition sqrt_I := sqrt_Half[+I*]sqrt_Half.
```

```
Lemma sqrt_I_nexp : (sqrt_I[^](2) [=] I).
```

```
Lemma nroot_I_nexp_aux : (n:nat)(on:(odd n))
  (Ex [m:nat] (mult n n) = (plus (mult (4) m) (1))).
```

```
Definition nroot_I [n:nat; on:(odd n)] : CC := I[^]n.
```

```
Lemma nroot_I_nexp : (n:nat; on:(odd n))((nroot_I ? on)[^]n [=] I).
```

```
Definition nroot_minus_I [n:nat; on:(odd n)] : CC := [--](nroot_I n on).
```

```
Lemma nroot_minus_I_nexp : (n:nat)(on:(odd n))
  ((nroot_minus_I n on)[^]n [=] [--]I).
```

28.4 NRootCC 1

28.4.1 NRootCC 1 ap real

Convention: Let `a, b : IR` and `bnz : b [#] Zero`.

```
Local c2 := a[^](2)[+]b[^](2).
```

```
Lemma nrCC1_c2pos : (Zero [:<] c2).

Local c := (sqrt c2 (less_leEq ??? nrCC1_c2pos)).

Local a'2 := (c[+]a)[*]Half.

Lemma nrCC1_a'2pos : (Zero [:<] a'2).

Local a' := (sqrt a'2 (less_leEq ??? nrCC1_a'2pos)).

Local b'2 := (c[-]a)[*]Half.

Lemma nrCC1_b'2pos : (Zero [:<] b'2).

Local b' := (sqrt b'2 (less_leEq ??? nrCC1_b'2pos)).

Lemma nrCC1_a3 : a'[^](2)[-]b'[^](2) [=] a.

Lemma nrCC1_a4 : (c[+]a)[*](c[-]a) [=] b[^](2).

Lemma nrCC1_a5 : a'2[*]b'2 [=] (b[*]Half)[^](2).

Lemma nrCC1_a6 : (Zero [:<] a'2[*]b'2).

Lemma nrCC1_a6' : (Zero [:<] (b[*]Half)[^](2)).

Lemma nrCC1_a7_upper : (Zero [:<] b)->(a'[*]b' [=] b[*]Half).

Lemma nrCC1_a7_lower : (b [:<] Zero)->(a'[*][--]b' [=] b[*]Half).

Lemma nrootCC_1_upper : (Zero [:<] b)->((a'[+I*]b')[^](2) [=] a[+I*]b).

Lemma nrootCC_1_lower : (b [:<] Zero)->((a'[+I*][--]b')[^](2) [=] a[+I*]b).

Lemma nrootCC_1_ap_real : (Ex [z:CC](z[^](2) [=] a[+I*]b)).
```

28.4.2 NRootCC 1 ap imag

Convention: Let `a, b : IR` and `anz : a [#] Zero`.

```
Local c' := (a[+I*]b)[*][--]I.
Local a' := (Re c').
Local b' := (Im c').

Lemma nrootCC_1_ap_imag : (Ex [z:CC](z[^](2) [=] a[+I*]b)).

Lemma nrootCC_1 : (c:CC)(c [#] Zero)->(Ex [z:CC](z[^](2) [=] c)).
```

28.5 NRootCC 2

Convention: Let `n : nat` and `c, z : CC` and `cnz : c [#] Zero`.

```
Lemma nrootCC_2' :
  ((z[*](conj z))[^]n [=] c[*](conj c))->
    (z[^]n[*](conj c)[-](conj z)[^]n[*]c [=] Zero)->
      ((z[^]n)[^](2) [=] c[^](2)).
```

```
Lemma nrootCC_2 :
  ((z[*](conj z))[^]n [=] c[*](conj c))->
    (z[^]n[*](conj c)[-](conj z)[^]n[*]c [=] Zero)->
      ((z[^]n [=] c) \/ (z[^]n [=] [--]c)).
```

28.6 NRootCC 3

```
Fixpoint Im_poly [p:(cpoly CC)] : (cpoly IR) :=
  Cases p of
    cpoly_zero => (cpoly_zero IR)
  | (cpoly_linear c p1) => (cpoly_linear IR (Im c) (Im_poly p1))
  end.
```

```
Lemma nrCC3_a1 : (p:(cpoly CC))(r:IR)((Im_poly p)!r [=] (Im p!(cc_IR r))).
```

```
Lemma nrCC3_a2 : (p:(cpoly CC))(n:nat)
  ((Nth_coeff n (Im_poly p)) [=] (Im (Nth_coeff n p))).
```

Convention: Let `a, b : IR` and `bnz : b [#] Zero` and `n : nat`.

```
Definition nrCC3_poly'' := (_X_[+](_C_ I))[^]n.
```

```
Lemma nrCC3_a3 : (r:IR)((nrCC3_poly''!(cc_IR r)) [=] (r[+I*]One)[^]n).
```

```
Lemma nrCC3_a4 : (Degree_le (1) _X_[+](_C_ I)).
```

```
Lemma nrCC3_a5 : (Degree_le n nrCC3_poly'').
```

```
Lemma nrCC3_a6 : (Nth_coeff n nrCC3_poly'') [=] One.
```

```
Definition nrCC3_poly' := nrCC3_poly''[*](_C_ a[+I*][--]b).
```

```
Lemma nrCC3_a7 : (r:IR)
  ((nrCC3_poly'!(cc_IR r)) [=] (r[+I*]One)[^]n[*](a[+I*][--]b)).
```

```
Lemma nrCC3_a8 : (Degree_le n nrCC3_poly').
```

```
Lemma nrCC3_a9 : (Nth_coeff n nrCC3_poly') [=] a[+I*][--]b.
```

```
Definition nrootCC_3_poly := (Im_poly nrCC3_poly').
```

```
Lemma nrootCC_3_ : (r:IR)
  ((nrootCC_3_poly!r) [=] (Im (r[+I*]One)[^]n[*](a[+I*][--]b))).
```

```
Lemma nrootCC_3 : (r:IR)
  ((cc_IR (nrootCC_3_poly!r))[*](Two[*]I) [=]
    ((r[+I*]One)[^]n[*](a[+I*][--]b)[-](r[+I*][--]One)[^]n[*](a[+I*]b))).
```

```
Lemma nrootCC_3_degree : (Degree n nrootCC_3_poly).
```

28.7 NRootCC 3'

Convention: Let `c : IR` and `n : nat` and `nnz : (lt O n)`.

```
Definition nrootCC_3'_poly := _X_[^]n[-](_C_ c).
```

```
Lemma nrootCC_3' : (x:IR)(nrootCC_3'_poly!x [=] x[^]n[-]c).
```

```
Lemma nrootCC_3'_degree : (Degree n nrootCC_3'_poly).
```

28.8 NRootCC 4

28.8.1 NRootCC 4 ap real

Convention: Let `a, b : IR` and `bnz : b [#] Zero` and `n:nat` and `on : (odd n)`.

```
Local c := a[+I*]b.
```

```
Lemma nrCC4_a1 :
  (Ex [r:IR]((r[+I*]One)[^]n[*](conj c)[-](r[+I*][--]One)[^]n[*]c [=] Zero)).
```

Convention: Let `r2', c2 : IR` and `r2'nz : r2' [#] Zero`.

```
Lemma nrCC4_a1' : (Ex [y2:IR]((y2[*]r2')[^]n [=] c2)).
```

Convention: Let `r,y2 : IR` with `(r[+I*]One)[^]n[*](conj c)[-](r[+I*][--]One)[^]n[*]c [=] Zero` and `(y2[*](r[^](2)[+]One))[^]n [=] a[^](2)[+]b[^](2)`.

```
Lemma nrCC4_a2 : (Zero [:<] a[^](2)[+]b[^](2)).
```

```
Lemma nrCC4_a3 : (Zero [:<] r[^](2)[+]One).
```

```
Lemma nrCC4_a4 : (Zero [:<] y2).
```

```
Definition nrCC4_y := (sqrt y2 (less_leEq ??? nrCC4_a4)).
Local y := nrCC4_y.
```

```
Definition nrCC4_x := y[*]r.
Local x := nrCC4_x.

Lemma nrCC4_a5 : x [=] y[*]r.

Lemma nrCC4_a6 : (x[^](2)[+]y[^](2))[^]n [=] a[^](2)[+]b[^](2).

Definition nrCC4_z := x[+I*]y.
Local z := nrCC4_z.

Lemma nrCC4_a7 : (z[^]n[*](conj c)[-](conj z)[^]n[*]c) [=] Zero.

Lemma nrCC4_a8 : (z[*](conj z))[^]n [=] c[*](conj c).

Lemma nrCC4_a9 : (z[^]n [=] c) \/ (z[^]n [=] [--]c).

Lemma nrCC4_a10 : (c:CC)
  (Ex [z:CC]((z[^]n [=] c) \/ (z[^]n [=] [--]c)))->
    (Ex [z:CC](z[^]n [=] c)).

Lemma nrootCC_4_ap_real : (Ex [z:CC](z[^]n [=] c)).
```

28.8.2 NRootCC 4 ap imag

Convention: Let `a, b : IR` and `n : nat` with `a [#] Zero` and `(odd n)`.

```
Local c' := (a[+I*]b)[*]I.
Local a' := (Re c').
Local b' := (Im c').

Lemma nrootCC_4_ap_real' : (Ex [z':CC](z'[^]n [=] a'[+I*]b')).

Lemma nrootCC_4_ap_imag : (Ex [z:CC](z[^]n [=] a[+I*]b)).

Lemma nrootCC_4 : (c:CC)(c [#] Zero)->(n:nat)(odd n)->
  (Ex [z:CC](z[^]n [=] c)).
```

28.9 NRootCC

```
Lemma nrCC_5a1 :
  (P:nat->Prop)
  ((n:nat)(odd n)->(P n))-> ((n:nat)(lt O n)->(P n)->(P (double n)))->
    (n:nat)(lt O n)->(P n).

Lemma nrCC_5a2 : (n:nat)(double n) = (mult (2) n).

Lemma nrCC_5a3 : (n:nat)(z:CC)((z[^](2))[^]n [=] z[^](double n)).
```

Convention: Let `c : CC` with `c [#] Zero`.

```
Lemma nrCC_5a4 : (n:nat)(lt O n)->
  (Ex [z:CC](z[^]n [=] c))->(Ex [z:CC](z[^](double n) [=] c)).

Lemma nrootCC_5 : (n:nat)(lt O n)->(Ex [z:CC](z[^]n [=] c)).
```

28.10 Final def

```
Definition nrootCC :
  (c:CC)(c [#] Zero)->(n:nat)(lt 0 n)->(Ex [z:CC](z[^]n [=] c))
    := nrootCC_5.
```

29 Kneser Lemma

Convention: Let `b : nat->CC`, `n : nat`, `(lt (0) n)`, `(b n) [=] One`, `c : IR`, `(AbsCC b_0) [:<] c`.

```
Local b_0 := (b 0).
Local b_n := (b n).
```

```
Local two_n := (mult (2) n).
Local Small := (p3m n).
Local Smaller := (p3m (mult two_n n)).
Local Smallest := Small[*]Smaller.
Local q := One[-]Smallest.
```

```
Lemma b_0'_exists :
(eta:IR)(Zero [:<] eta) ->
        (EX b_0' | ((AbsCC (b_0'[-]b_0)) [:<] eta) /\ (b_0' [#] Zero)).
```

```
Local eta_0 := (c[-](AbsCC b_0))[/]FourNZ.
```

```
Lemma eta_0_pos : (Zero[:<] eta_0).
```

```
Lemma eta_exists :
        (EX eta | (Zero [:<] eta) /\
          (EX b_0' |
            ((AbsCC (b_0'[-]b_0)) [:<] eta) /\
            (b_0' [#] Zero) /\
            ((AbsCC b_0')[+]Three[*]eta [:<] c))).
```

```
Lemma eps_exists_1 :
  (eps,x,y:IR)(Zero [:<] eps) -> (Zero [:<] x) -> (Zero [:<] y) ->
    (EX eps' | (Zero [:<] eps') /\ (eps' [:<] eps) /\ (x[*]eps' [:<] y)).
```

```
Lemma eps_exists_2 :
  (x1,y1,x2,y2,z:IR)
    (Zero [:<] x1) -> (Zero [:<] y1) ->
    (Zero [:<] x2) -> (Zero [:<] y2) ->
    (Zero [:<] z) ->
      (EX eps | (Zero [:<] eps) /\
        (x1[*]eps [:<] y1) /\ (x2[*]eps [:<] y2) /\ (eps [:<] z)).
```

```
Lemma eps_exists :
(eta,a_0:IR)(Zero [:<] eta) -> (Zero [:<] a_0) ->
        (EX eps | (Zero [:<] eps) /\
          (Two[*](Three[^]n[+]One)[*]eps [:<] eta) /\
          (Two[*]eps [:<] Smaller[*]a_0) /\
          (eps [:<] a_0)).


Local a : nat->IR := [i:nat](AbsCC (b i)).


Lemma z_exists :
(b_0':CC; k:nat; r,eta:IR)
  let a_0 = (AbsCC b_0') in
        (Zero [:<] a_0) -> (Zero [:<] (a k)) ->
        (le (1) k) -> (le k n) ->
        (Zero [:<] r) -> (Zero [:<] eta) ->
        ((AbsCC (b_0'[-]b_0)) [:<] eta) ->
        ((a k)[*]r[^]k [:<] a_0) ->
          (EX z |
            ((AbsCC z) [=] r) /\
            ((AbsCC (b_0[+](b k)[*]z[^]k)) [:<] a_0[-](a k)[*]r[^]k[+]eta)).


Lemma Kneser_1' : Half [:<] q.


Lemma Kneser_1'' : q [:<] One.


Lemma Kneser_1 :
(a_0,eta,eps:IR)(Zero [:<] eta) -> (Zero [:<] eps) ->
        (a_0[+]Three[*]eta [:<] c) ->
        (Two[*](Three[^]n[+]One)[*]eps [:<] eta) ->
          (q[*]a_0[+]Three[^]n[*]eps[+]eps[+]eta [:<] q[*]c).


Lemma Kneser_2a :
(R:CRing)(m,n,i:nat)(f:nat->R)
  (le (1) i) ->
    (Sum m n f) [=] (f m)[+](f i)[+]((Sum (S m) (pred i) f)[+](Sum (S i) n f)).


Lemma Kneser_2b :
(k:nat; z:CC)(le (1) k) ->
  let p_ = [i:nat](b i)[*]z[^]i in
        (Sum (0) n [i:nat](b i)[*]z[^]i) [=]
          (b_0[+](b k)[*]z[^]k)[+]
            ((Sum (1) (pred k) p_)[+](Sum (S k) n p_)).


Lemma Kneser_2c :
(m,n:nat; z:CC)(le m (S n)) ->
  let r = (AbsCC z) in
        (AbsCC (Sum m n [i:nat](b i)[*]z[^]i)) [:<=]
          (Sum m n [i:nat](a i)[*]r[^]i).
```

```
Lemma Kneser_2 :
(k:nat; z:CC)(le (1) k) -> (le k n) ->
  let r = (AbsCC z) in
  let p_ = [i:nat](a i)[*]r[^]i in
        (AbsCC (Sum (0) n [i:nat](b i)[*]z[^]i)) [:<=]
          (AbsCC (b_0[+](b k)[*]z[^]k))[+]
             ((Sum (1) (pred k) p_)[+](Sum (S k) n p_)).
```

```
Lemma Kneser_3 :
        (EX z |
          ((AbsCC z)[^]n [:<] c) /\
          ((AbsCC (Sum (0) n [i:nat](b i)[*]z[^]i)) [:<] q[*]c)).
```

```
Lemma Kneser :
(n:nat)(lt (0) n) ->
  (EX q | (Zero [:<] q) /\ (q [:<] One) /\
    (p:(cpoly CC))(Monic n p) ->
      (c:IR)((AbsCC (p!Zero)) [:<] c) ->
        (EX z |
          ((AbsCC z)[^]n [:<] c) /\
          ((AbsCC (p!z)) [:<] q[*]c))).
```

```
Lemma fta_monic : (p: (cpoly CC);n:nat)(lt (0) n) ->
 (Monic n p)->
 (EX c | ( p ! c) [=] Zero).
```

```
Lemma fta_reg : (p: (cpoly CC);n:nat)(lt (0) n) ->
 (Degree n p)->
 (EX c | ( p ! c) [=] Zero).
```

30 Fundamental Theorem of Algebra

```
Variable f : (cpoly_cring CC).
Variable n : nat.
Hypothesis f_degree : (Degree (S n) f).

Lemma fta_reg' :
  (EX f1 | (Degree (1) f1) /\ (EX f2 | (Degree n f2) /\ (f [=] f1[*]f2))).

Variable f : (cpoly_cring CC).
Variable n : nat.
Hypothesis f_degree : (Degree_le (S n) f).
Variable c : CC.
Hypothesis f_c : f!c [#] Zero.
```

```
Lemma fta_1 :
  (EX f1 | (Degree_le (1) f1) /\ (EX f2 | (Degree_le n f2) /\
    (f [=] f1[*]f2))).
```

```
Lemma fta_1' :
  (EX a | (EX b | (EX g | (Degree_le n g) /\
    (f [=] ((_C_ a)[*]_X_[+](_C_ b))[*]g)))).


Lemma fta' :
  (n:nat)(f:(cpoly_cring CC))
    (Degree_le n f) -> (nonConst ? f) -> (EX z | f!z [=] Zero).


Lemma fta : (f:(cpoly_cring CC))(nonConst ? f) -> (EX z | f!z [=] Zero).
```

Index