A Formalization of Elementary Linear Algebra: Part III

A Formal Theory of Algebraic Field Extensions: Part I

David M. Russinoff
david@russinoff.com

November 17, 2026

Formalization of Algebra in ACL2

Finite Group Theory: `books/projects/groups/`

- Groups, subgroups, quotient groups
- Homomorphisms, direct products, Abelian groups
- Symmetric groups, Sylow theorems

Linear Algebra: `books/projects/linear/`

- Matrix algebra, determinants
- Row reduction, matrix inverse, systems of linear equations
- **Vector spaces, linear transformations**

Field Theory: `books/projects/fields/`

- **Algebraic field extensions**
- Galois theory

THE FIELD F

```
(encapsulate (((fp *) => *)                          ;element recognizer
              ((f+ * *) => *) ((f* * *) => *)        ;operations
              ((f0) => *) ((f1) => *)                ;identities
              ((f- *) => *) ((f/ *) => *))           ;inverses
  ;; Local definitions:
  (local (defun fp (x) (rationalp x)))
  (local (defun f+ (x y) (+ x y)))
  (local (defun f* (x y) (* x y)))
  (local (defun f0 () 0))
  (local (defun f1 () 1))
  (local (defun f- (x) (- x)))
  (local (defun f/ (x) (/ x)))
  ;; Field axioms:
  (defthm f+closed (implies (and (fp x) (fp y)) (fp (f+ x y))))
  ...
  (defthm fdist
    (implies (and (fp x) (fp y) (fp z))
             (equal (f* x (f+ y z)) (f+ (f* x y) (f* x z))))))
)
```

The Vector Space V

```
(encapsulate (((vp *) => *)          ;vector recognizer
              ((v+ * *) => *)        ;vector addition
              ((v0) => *)            ;zero vector
              ((v- *) => *)          ;additive inverse
              ((v* * *) => *))       ;scalar multiplication


  ;; Local definitions:
  (local (defun vp (x) (fp x)))
  (local (defun v+ (x y) (f+ x y)))
  ...


  ;; Vector space axioms:
  (defthm vp-v0 (vp (v0)))
  (defthm v+closed
    (implies (and (vp x) (vp y)) (vp (v+ x y))))
  ...
  (defthmd vdistv
    (implies (and (fp c) (vp x) (vp y)) (equal (v+ (v* c x) (v* c y))
)
```

The Vector Space V

```
(encapsulate (((vp *) => *)          ;vector recognizer
              ((v+ * *) => *)        ;vector addition
              ((v0) => *)            ;zero vector
              ((v- *) => *)          ;additive inverse
              ((v* * *) => *)        ;scalar multiplication
              ((vbasis0) => *)       ;canonical basis
              ((vcoords0 *) => *)) ;coordinates relative to vbasis0
  ;; Local definitions:
  (local (defun vp (x) (fp x)))
  (local (defun v+ (x y) (f+ x y)))
  ...
  (local (defun vbasis0 () (list (f1))))
  (local (defun vcoords0 (x) (list x)))
  ;; Vector space axioms:
  (defthm vp-v0 (vp (v0)))
  (defthm v+closed
    (implies (and (vp x) (vp y)) (vp (v+ x y))))
  ...
  (defthmd vdistv
    (implies (and (fp c) (vp x) (vp y)) (equal (v+ (v* c x) (v* c y))
   ...
```

THE VECTOR SPACE V (CONT.)

```
...
;; List of vectors
(defun vlistnp (x n)
  (if (zp n)
      (null x)
    (and (consp x)
         (vp (car x))
         (vlistnp (cdr x) (1- n)))))
;; Linear combination of a list of vectors:
(defun vcomb (flist vlist)
  (if (consp flist)
      (v+ (v* (car flist) (car vlist))
          (vcomb (cdr flist) (cdr vlist)))
    (v0)))
;; Dimension of the space
(defun vdim () (len (vbasis0)))
...
```

THE VECTOR SPACE V (CONT.)

```
  ...
  (defthm vlistnp-basis0
    (vlistnp (vbasis0) (vdim)))
  (defthm flistnp-vcoords0
    (implies (vp x)
             (flistnp (vcoords0 x) (vdim))))
  (defthm vbasis0-spans
    (implies (vp x)
             (equal (vcomb (vcoords0 x) (vbasis0))
                    x)))
  (defthmd vbasis0-lin-indep
    (implies (and (flistnp c (vdim))
                  (equal (vcomb c (vbasis0)) (v0)))
             (equal (flistn0 (vdim)) c)))
)
```

Linear Independence

Constructive definition:

```
(defun vcoord-mat (l)
  (if (consp l)
      (cons (vcoords0 (car l)) (vcoord-mat (cdr l)))
    ()))

(defund vindepp (l)
  (or (null l)
      (equal (row-rank (vcoord-mat l))
             (len l))))

(defund vdepp (l) (not (vindepp l)))
```

Trivial consequences:

```
(defthm vindepp-vbasis0 (vindepp (vbasis0)))

(defthmd vdepp-if->-vdim
  (implies (and (natp m) (> m (vdim)) (vlistnp l m))
           (vdepp l)))
```

Equivalent Characterization

A list `l` is linearly dependent iff the zero vector is a (constructively defined) nontrivial linear combination of `l`:

```
(defthm vindepp-vcomb-v0
  (implies (and (natp m) (vlistnp l m)
                (vindepp l) (flistnp c m)
                (equal (vcomb c l) (v0)))
           (equal c (flistn0 m))))

(defun vdep-coeffs (l)
  (nth (1- (len l)) (row-reduce-mat (vcoord-mat l))))

(defthmd vdepp-vcomb-v0
  (implies (and (posp m) (vlistnp l m) (vdepp l))
           (let ((c (vdep-coeffs l)))
             (and (flistnp c m)
                  (not (equal c (flistn0 m)))
                  (equal (vcomb c l) (v0))))))
```

Linear Transformations

We introduce a second vector space, W:

```
(encapsulate (((wp *) => *) ((w+ * *) => *) ((w0) => *) ... ) ... )
```

All definitions and lemmas pertaining to `V` are repeated for `W`.

A generic linear transformation from `V` to `W`:

```
(encapsulate (((lin *) => *))
  (local (defun lin (x) (declare (ignore x)) (w0)))
  (defthm lin-val (implies (vp x) (wp (lin x))))
  (defthm lin-v0 (equal (lin (v0)) (w0)))
  (defthm lin-v+
    (implies (and (vp x) (vp y))
             (equal (lin (v+ x y))
                    (w+ (lin x) (lin y)))))
  (defthm lin-v*
    (implies (and (fp c) (vp x))
             (equal (lin (v* c x))
                    (w* c (lin x))))))
```

Matrix of a Linear Transformation

Constructive definition:

```
(defun lin-list (l)
  (if (consp l)
      (cons (lin (car l)) (lin-list (cdr l)))
    ()))

(defund lin-mat () (wcoord-mat (lin-list (vbasis0))))
```

Properties:

```
(defthm fmatp-lin-mat (fmatp (lin-mat) (vdim) (wdim)))

(defthmd lin-mat-lin
  (implies (vp x)
           (equal (wcoords0 (lin x))
                  (car (fmat* (list (vcoords0 x)) (lin-mat))))))
```

Injections and Surjections

```
(defun-sk lin-injective-p ()
  (forall (x)
    (implies (and (vp x) (equal (lin x) (w0)))
             (equal x (v0)))))

(defchoose lin-preimage x (y)
  (and (vp x) (equal (lin x) y)))

(defun-sk lin-surjective-p ()
  (forall (y)
    (implies (wp y)
             (and (vp (lin-preimage y))
                  (equal (lin (lin-preimage y)) y)))))

(defthmd injection-dim-<=
  (implies (lin-injective-p)
           (and (<= (vdim) (wdim))
                (iff (lin-surjective-p) (equal (vdim) (wdim))))
```

Inverse of a Linear Transformation

Constructive definition, valid if `lin` is injective and surjective:

```
(defund lin-inv (y)
  (vcomb (car (fmat* (list (wcoords0 y)) (inverse-mat (lin-mat))))
         (vbasis0)))
```

Properties:

```
(defthm vp-lin-inv
  (implies (and (lin-injective-p) (= (vdim) (wdim)) (wp y))
           (vp (lin-inv y))))

(defthmd lin-inv-linear
  (implies (and (lin-injective-p) (= (vdim) (wdim))
                (wp x) (wp y) (fp c))
           (equal (lin-inv (w+ (w* c x) y))
                  (v+ (v* c (lin-inv x)) (lin-inv y)))))

(defthmd lin-lin-inv
  (implies (and (lin-injective-p) (= (vdim) (wdim)) (wp y))
           (let ((x (lin-inv y)))
             (and (vp x) (equal (lin x) y)))))
```

Fields as Logical Objects (Overview)

We shall define a predicate `fieldp` that recognizes a class of *fields*. A field is either a *base field* or a *simple extension* of a field.

A base field is either the rational field, a prime field, or the generic field.

A *polynomial* `p` of *degree* `d` over a field `f` is a list of elements of `f` of length `d` + 1 with `(car p)` $\neq 0$ unless `d` = 0.

An irreducible monic polynomial `p` over `f` determines a simple extension of `f`. Its elements are the polynomials over `f` of degree less than `(degree p)`. Its operations are polynomial addition and multiplication modulo `p`.

A simple extension is represented as a cons `(p . f)`.
A field is an improper list `(p1 ... pk . f)`.

Base Fields

Definition:

```
(defund fbasep (f)
  (or (zerop f)        ;the rational field is encoded as 0
      (primep f)       ;the prime field of order p is encoded as p
      (equal f 'f0))) ;the generic field is encoded as the symbol f0
```

Definitions of basic functions are straightforward:

- `(beltp x f)`
- `(badd x y f)` and `(bmul x y f)`
- `(bzero f)` and `(bone f)`
- `(bneg x f)` and `(brecip x f)`

For example,

```
(defund bmul (x y f)
  (cond ((zerop f) (* x y))
        ((primep f) (mod (* x y) f))
        (t (f* x y))))
```

Mutually Recursive Functions

Functions that characterize the structure of a field:

- `(feltp x f)`
- `(fadd x y f)` and `(fmul x y f)`
- `(fzero f)` and `(fone f)`
- `(fneg x f)` and `(frecip x f)`

Functions pertaining to polynomials over a field:

- `(polyp p f)`
- `(padd p q f)` and `(pmul p q f)`
- `(pzero f)` and `(pone f)`
- `(pneg p f)`
- `(pquot p q f)`, `(prem p q f)`, and `(pdivides p q f)`
- `(pgcd p q f)`, `(r$ p q f)`, and `(s$ p q f)`

IRREDUCIBILITY

```
(defchoose pfactor q (p f)
  (and (polyp q f) (monicp q f)
       (> (degree q) 0) (< (degree q) (degree p))
       (pdivides q p f)))

(defund reduciblep (p f)
  (let ((q (pfactor p f)))
    (and (polyp q f) (monicp q f)
         (> (degree q) 0) (< (degree q) (degree p))
         (pdivides q p f))))

(defund irreduciblep (p f) (not (reduciblep p f)))
```

Fields and Extensions

```
(defund fieldp (f)
  (if (consp f)
      (and (fieldp (cdr f))
           (polyp (car f) (cdr f)) (monicp (car f) (cdr f))
           (irreduciblep (car f) (cdr f)) (>= (degree (car f)) 2))
    (fbasep f)))

(defun extends (e f)
  (or (equal e f) (and (consp e) (extends (cdr e) f))))

(defun extensionp (e f) (and (fieldp f) (fieldp e) (extends e f)))

(defun edegree (e f)
  (if (equal e f)
      1
    (and (consp e)
         (* (degree (car e)) (edegree (cdr e) f)))))
```

Inductive Proof of Field Axioms

To enable the induction, we define a predicate that collectively asserts the axioms:

```
(defun-sk fadd-closed-p (f)
  (forall (x y)
    (implies (and (feltp x f) (feltp y f))
             (feltp (fadd x y f) f))))
...
(defund field-axioms-p (f)
  (and (fadd-closed-p f) (fmul-closed-p f) ... (fdistrib-p f)))
```

Base step:

```
(defthmd field-axioms-p-fbasep
  (implies (fbasep f) (field-axioms-p f)))
```

Inductive step:

```
(defthmd field-axioms-p-step
  (implies (and (fieldp f) (consp f) (field-axioms-p (cdr f)))
           (field-axioms-p f)))
```

Inductive Proof of Field Axioms (cont.)

The main theorem follows by induction:

```
(defthmd field-axioms-p-fieldp
  (implies (fieldp f) (field-axioms-p f)))
```

Many consequences of this theorem have been proved along the way, including the field axioms, e.g.,

```
(defthm fmul-frecip
  (implies (and (fieldp f) (feltp x f)  (not (equal x (fzero f))))
           (equal (fmul x (frecip x f) f)
                  (fone f))))
```

the polynomial ring axioms, e.g.,

```
(defthmd padd-comm
  (implies (and (fieldp f)
                (polyp x f) (polyp y f))
           (equal (padd x y f) (padd y x f))))
```

and countless other properties of fields, polynomials, divisibility, irreducibility, etc.

Embedding of a Field in an Extension

Although `f` is not a subfield of an extension `e`, there is a natural homomorphism from `f` to `e`:

```
(defund flift (x f e)
  (if (equal f e)
      x
    (and (consp e) (list (flift x f (cdr e))))))
```

This induces a homomorphism of the polynomial rings:

```
(defun plift (p f e)
  (if (consp p)
      (cons (flift (car p) f e) (plift (cdr p) f e))
    ()))
```

Polynomial Evaluation and Roots

```
(defun fpower (x n f)
  (if (zp n) (fone f) (fmul x (fpower x (1- n) f) f)))

(defun peval (p x f)
  (if (consp p)
      (if (consp (cdr p))
          (fadd (fmul (car p) (fpower x (degree p) f) f)
                (peval (pstrip (cdr p) f) x f)
                f)
        (car p))
    ()))

(defund prootp (x p f)
  (and (feltp x f) (equal (peval p x f) (fzero f))))
```

Identity Polynomial and Primitive Element

```
(defund pid (f) (list (fone f) (fzero f)))

(defthmd peval-pid
  (implies (and (fieldp f) (feltp x f))
           (equal (peval (pid f) x f) x)))

(defund primitive (f) (pid (cdr f)))

(defthmd prootp-primitive
  (implies (and (fieldp f) (consp f))
           (prootp (primitive f) (plift (car f) (cdr f) f) f)))
```

Field Extensions as Vector Spaces

Operations:

- Vector addition: `(fadd x y e)`
- Scalar multiplication: `(fmul (flift c f e) x e)`

For a simple extension `e` = `(p . f)`, with `(degree p)` = `d`:

- `(ebasis0 e f)` is the list of the first `d` powers of `(primitive e)`
- `(ecoords0 x e f)` = `(zpad x d f)`

In the general case, the definition of `(ebasis0 e f)` and `(ecoords0 x e f)` are extended recursively.

Dimension of the space: `(edegree e f)`.

Once the vector space axioms are proved, our results on vector spaces may be applied by functional instantiated.

Finite Extensions are Algebraic

If `e` is an extension of `f` of degree `d` and `x` is in `e`, then by functional instantiation of `vdepp-if->-vdim`, the first `d` + 1 powers of `x` are linearly dependent over `f`.

By `vdepp-vcomb-v0`, this provides an executably defined polynomial, `(zero-poly x e f)`, of which `x` is a root:

```
(defthmd prootp-zero-poly
  (implies (and (extensionp e f) (feltp x e))
           (let ((p (zero-poly x e f)))
             (and (polyp p f)
                  (monicp p f)
                  (>= (degree p 1)
                  (<= (degree p) (edegree e f))
                  (prootp x (plift p f e) e)))))
```

EMBEDDINGS (OVERVIEW)

Let `e` and `k` be extensions of `f`. An *embedding* of `e` in `k` over `f` is a field homomorphism of `e` into `k` that fixes `f`, i.e., if `x` is in `f`, then `(flift x f e)` is mapped to `(flift x f k)`.

When `e` = `k`, these embeddings are the *automorphisms* of `e` over `f` and form the *Galois group* of `e` over `f`.

An embedding of a simple extension is determined by the image of its primitive element.

Thus, we define an embedding to be a list of elements of `k` corresponding to the simple extensions in the construction of `e` as an extension of `f`.

EMBEDDINGS

```
(mutual-recursion

  (defund embed (x phi k f)
    (if (consp phi)
        (peval (pembed x (cdr phi) k f) (car phi) k)
      (flift x f k)))

  (defun pembed (p phi k f)
      (if (consp p)
          (cons (embed (car p) phi k f)
                (pembed (cdr p) phi k f))
        ()))
)

(defund embeddingp (phi e k f)
  (if (equal e f)
      (null phi)
    (and (consp phi)
         (embeddingp (cdr phi) (cdr e) k f)
         (prootp (car phi) (pembed (car e) (cdr phi) k f) k))))
```

EMBEDDINGS AS LINEAR TRANSFORMATIONS

```
(defthm embed-fadd
 (implies (and (extensionp e f) (extensionp k f)
               (embeddingp phi e k f) (feltp x e) (feltp y e))
          (equal (embed (fadd x y e) phi k f)
                 (fadd (embed x phi k f) (embed y phi k f) k))))

(defthm embed-fmul
 (implies (and (extensionp e f) (extensionp k f)
               (embeddingp phi e k f) (feltp x e) (feltp y e))
          (equal (embed (fmul x y e) phi k f)
                 (fmul (embed x phi k f) (embed y phi k f) k))))

(defthm embed-fixes
  (implies (and (extensionp e f) (extensionp k f)
                (embeddingp phi e k f) (feltp x f))
           (equal (embed (flift x f e) phi k f)
                  (flift x f k))))

(defthm embedding-1-1
  (implies (and (extensionp e f) (extensionp k f)
                (embeddingp phi e k f) (feltp x e) (feltp y e)
                (equal (embed x phi k f) (embed y phi k f)))
           (equal x y)))
```

Embeddings and Degrees of Extensions

By functional instantiation of `injection-dim-<=`:

```
(defthmd embedding-degree-<=
  (implies (and (extensionp e f) (extensionp k f)
                (embeddingp phi e k f))
           (<= (edegree e f) (edegree k f))))

(defchoose preembed x (y phi e k f)
  (and (feltp x e) (equal (embed x phi k f) y)))

(defun-sk surjective-embedding-p (phi e k f)
  (forall (y)
    (implies (feltp y k)
             (let ((x (preembed y phi e k f)))
               (and (feltp x e) (equal (embed x phi k f) y))))))

(defthm embedding-surjective
  (implies (and (extensionp e f) (extensionp k f)
                (embeddingp phi e k f))
           (iff (equal (edegree e f) (edegree k f))
                (surjective-embedding-p phi e k f))))
```

IDENTITY EMBEDDING

Trivial embedding of `e` in `k`, when `k` is an extension of `e`:

```
(defun trivial-embedding (e k f)
  (if (and (extends e f) (not (equal e f)))
      (cons (flift (primitive e) e k)
            (trivial-embedding (cdr e) k f))
    ()))
```

Identity embedding of `e` is the case `e` = `k`:

```
(defund id-embedding (e f) (trivial-embedding e e f))

(defthmd id-embedding-id
  (implies (extensionp e f)
           (and (embeddingp (id-embedding e f) e e f)
                (implies (feltp x e)
                         (equal (embed x (id-embedding e f) e f)
                                x)))))
```

Composition of Embeddings

If `phi` embeds `e` in `g` and `psi` embeds `g` in `k`, then the composition embeds `e` in `k`:

```
(defun comp-embedding (psi phi e k f)
  (if (and (extends e f) (not (equal e f)))
      (cons (embed (car phi) psi k f)
            (comp-embedding psi (cdr phi) (cdr e) k f))
    ()))

(defthmd embeddingp-comp-embedding
  (implies
    (and (extensionp e f) (extensionp g f) (extensionp k f)
         (embeddingp phi e g f) (embeddingp psi g k f))
    (and (embeddingp (comp-embedding psi phi e k f) e k f)
         (implies (feltp x e)
                  (equal (embed x (comp-embedding psi phi e k f) k f)
                         (embed (embed x phi g f) psi k f))))))
```

Inverse of an Embedding

If `phi` is an embedding of `e` in `k` and `(edegree e f) = (edegree k f)`, then the inverse of `phi` embeds `k` in `e`.

The definition is based on `lin-inv` and is therefore executable.

```
(defthmd comp-inv-embedding
  (implies (and (extensionp e f) (extensionp k f)
                (equal (edegree e f) (edegree k f)
                (embeddingp phi e k f) )
           (let ((inv (inv-embedding phi e k f)))
             (and (embeddingp inv k e f)
                  (equal (comp-embedding inv phi e e f)
                         (id-embedding e f))
                  (equal (comp-embedding phi inv k k f)
                         (id-embedding k f))))))
```

EXECUTABILITY

Galois goup of a field extension is executabable

- Group operation (`comp-embedding`)
- Inverse operator (`inv-embedding`)

Processes that depend on `pfactor` are not executable

- Polynomial factorization and irreducibility
- Evaluation of `fieldp`
- Computation of minimal polynomials
- Enumeration of Galois group elements

Proposed project: Factorization of polynomials

- Factorization over prime fields
- Factorization over rationals
- Extension to finite fields and number fields