

Polynomial Factorization*

René Thiemann and Akihisa Yamada

December 14, 2021

Abstract

Based on existing libraries for polynomial interpolation and matrices, we formalized several factorization algorithms for polynomials, including Kronecker's algorithm for integer polynomials, Yun's square-free factorization algorithm for field polynomials, and a factorization algorithm which delivers root-free polynomials.

As side products, we developed division algorithms for polynomials over integral domains, as well as primality-testing and prime-factorization algorithms for integers.

Contents

1	Introduction	2
1.1	Missing List	3
1.2	Partitions	4
1.3	merging functions	5
2	Preliminaries	18
2.1	Missing Multiset	18
2.2	Precomputation	19
2.3	Order of Polynomial Roots	20
3	Explicit Formulas for Roots	21
4	Division of Polynomials over Integers	23
5	More on Polynomials	25
6	Gauss Lemma	28
7	Prime Factorization	32
7.1	Definitions	32
7.2	Proofs	33

*Supported by FWF (Austrian Science Fund) project Y757.

8	Rational Root Test	36
9	Kronecker Factorization	37
9.1	Definitions	37
9.2	Code setup for divisors	38
9.3	Proofs	38
10	Polynomial Divisibility	40
10.1	Fundamental Theorem of Algebra for Factorizations	41
11	Square Free Factorization	42
11.1	Yun’s factorization algorithm	44
11.2	Yun factorization and homomorphisms	51
12	GCD of rational polynomials via GCD for integer polynomials	53
13	Rational Factorization	53

1 Introduction

The details of the factorization algorithms have mostly been extracted from Knuth’s Art of Computer Programming [1]. Also Wikipedia provided valuable help.

As a first fast preprocessing for factorization we integrated Yun’s factorization algorithm which identifies duplicate factors [2]. In contrast to the existing formalized result that the GCD of p and p' has no duplicate factors (and the same roots as p), Yun’s algorithm decomposes a polynomial p into $p_1^1 \cdot \dots \cdot p_n^n$ such that no p_i has a duplicate factor and there is no common factor of p_i and p_j for $i \neq j$. As a comparison, the GCD of p and p' is exactly $p_1 \cdot \dots \cdot p_n$, but without decomposing this product into the list of p_i ’s.

Factorization over \mathbb{Q} is reduced to factorization over \mathbb{Z} with the help of Gauss’ Lemma.

Kronecker’s algorithm for factorization over \mathbb{Z} requires both polynomial interpolation over \mathbb{Z} and prime factorization over \mathbb{N} . Whereas the former is available as a separate AFP-entry, for prime factorization we mechanized a simple algorithm depicted in [1]: For a given number n , the algorithm iteratively checks divisibility by numbers until \sqrt{n} , with some optimizations: it uses a precomputed set of small primes (all primes up to 1000), and if $n \bmod 30 = 11$, the next test candidates in the range $[n, n + 30)$ are only the 8 numbers $n, n + 2, n + 6, n + 8, n + 12, n + 18, n + 20, n + 26$.

However, in theory and praxis it turned out that Kronecker's algorithm is too inefficient. Therefore, in a separate AFP-entry we formalized the Berlekamp-Zassenhaus factorization.¹

There also is a combined factorization algorithm: For polynomials of degree 2, the closed form for the roots of quadratic polynomials is applied. For polynomials of degree 3, the rational root test determines whether the polynomial is irreducible or not, and finally for degree 4 and higher, Kronecker's factorization algorithm is applied.

1.1 Missing List

The provides some standard algorithms and lemmas on lists.

theory *Missing-List*

imports

Matrix.Utility

HOL-Library.Monad-Syntax

begin

fun *concat-lists* :: 'a list list \Rightarrow 'a list list **where**

concat-lists [] = [[]]

| *concat-lists* (as # xs) = *concat* (*map* (λ vec. *map* (λ a. a # vec) as) (*concat-lists* xs))

lemma *concat-lists-listset*: *set* (*concat-lists* xs) = *listset* (*map set* xs)

<proof>

lemma *sum-list-concat*: *sum-list* (*concat* ls) = *sum-list* (*map sum-list* ls)

<proof>

lemma *listset*: *listset* xs = { ys. *length* ys = *length* xs \wedge (\forall i < *length* xs. ys ! i \in xs ! i)}

<proof>

lemma *set-concat-lists[simp]*: *set* (*concat-lists* xs) = {as. *length* as = *length* xs \wedge (\forall i < *length* xs. as ! i \in *set* (xs ! i))}

<proof>

declare *concat-lists.simps[simp del]*

fun *find-map-filter* :: ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a list \Rightarrow 'b option **where**

find-map-filter f p [] = None

| *find-map-filter* f p (a # as) = (let b = f a in if p b then Some b else *find-map-filter* f p as)

¹The Berlekamp-Zassenhaus AFP-entry was originally not present and at that time, this AFP-entry contained an implementation of Berlekamp-Zassenhaus as a non-certified function.

lemma *find-map-filter-Some*: $\text{find-map-filter } f \text{ } p \text{ } as = \text{Some } b \implies p \text{ } b \wedge b \in f \text{ ' } \text{set } as$
 ⟨proof⟩

lemma *find-map-filter-None*: $\text{find-map-filter } f \text{ } p \text{ } as = \text{None} \implies \forall b \in f \text{ ' } \text{set } as. \neg p \text{ } b$
 ⟨proof⟩

lemma *remdups-adj-sorted-distinct[simp]*: $\text{sorted } xs \implies \text{distinct } (\text{remdups-adj } xs)$
 ⟨proof⟩

lemma *subseqs-length-simple*:
 assumes $b \in \text{set } (\text{subseqs } xs)$ shows $\text{length } b \leq \text{length } xs$
 ⟨proof⟩

lemma *subseqs-length-simple-False*:
 assumes $b \in \text{set } (\text{subseqs } xs)$ $\text{length } xs < \text{length } b$ shows *False*
 ⟨proof⟩

lemma *empty-subseqs[simp]*: $[] \in \text{set } (\text{subseqs } xs)$ ⟨proof⟩

lemma *full-list-subseqs*: $\{ys. ys \in \text{set } (\text{subseqs } xs) \wedge \text{length } ys = \text{length } xs\} = \{xs\}$
 ⟨proof⟩

lemma *nth-concat-split*: assumes $i < \text{length } (\text{concat } xs)$
 shows $\exists j \ k. j < \text{length } xs \wedge k < \text{length } (xs ! j) \wedge \text{concat } xs ! i = xs ! j ! k$
 ⟨proof⟩

lemma *nth-concat-diff*: assumes $i1 < \text{length } (\text{concat } xs)$ $i2 < \text{length } (\text{concat } xs)$
 $i1 \neq i2$
 shows $\exists j1 \ k1 \ j2 \ k2. (j1, k1) \neq (j2, k2) \wedge j1 < \text{length } xs \wedge j2 < \text{length } xs$
 $\wedge k1 < \text{length } (xs ! j1) \wedge k2 < \text{length } (xs ! j2)$
 $\wedge \text{concat } xs ! i1 = xs ! j1 ! k1 \wedge \text{concat } xs ! i2 = xs ! j2 ! k2$
 ⟨proof⟩

lemma *list-all2-map-map*: $(\bigwedge x. x \in \text{set } xs \implies R \text{ } (f \text{ } x) \text{ } (g \text{ } x)) \implies \text{list-all2 } R \text{ } (\text{map } f \text{ } xs) \text{ } (\text{map } g \text{ } xs)$
 ⟨proof⟩

1.2 Partitions

Check whether a list of sets forms a partition, i.e., whether the sets are pairwise disjoint.

definition *is-partition* :: $('a \text{ set}) \text{ list} \Rightarrow \text{bool}$ where
 $\text{is-partition } cs \longleftrightarrow (\forall j < \text{length } cs. \forall i < j. cs ! i \cap cs ! j = \{\})$

definition *is-partition-alt* :: ('a set) list \Rightarrow bool **where**
is-partition-alt cs \longleftrightarrow (\forall i j. i < length cs \wedge j < length cs \wedge i \neq j \longrightarrow cs!i \cap cs!j = {})

lemma *is-partition-alt*: *is-partition* = *is-partition-alt*
 \langle proof \rangle

lemma *is-partition-Nil*:
is-partition [] = True \langle proof \rangle

lemma *is-partition-Cons*:
is-partition (x#xs) \longleftrightarrow *is-partition* xs \wedge x \cap \bigcup (set xs) = {} (**is** ?l = ?r)
 \langle proof \rangle

lemma *is-partition-sublist*:
assumes *is-partition* (us @ xs @ ys @ zs @ vs)
shows *is-partition* (xs @ zs)
 \langle proof \rangle

lemma *is-partition-inj-map*:
assumes *is-partition* xs
and *inj-on* f (\bigcup x \in set xs. x)
shows *is-partition* (map ((\cdot) f) xs)
 \langle proof \rangle

context
begin

private fun *is-partition-impl* :: 'a set list \Rightarrow 'a set option **where**
is-partition-impl [] = Some {}
| *is-partition-impl* (as # rest) = do {
 all \leftarrow *is-partition-impl* rest;
 if as \cap all = {} then Some (all \cup as) else None
}

lemma *is-partition-code*[code]: *is-partition* as = (*is-partition-impl* as \neq None)
 \langle proof \rangle
end

lemma *case-prod-partition*:
case-prod f (partition p xs) = f (filter p xs) (filter (Not \circ p) xs)
 \langle proof \rangle

lemmas *map-id*[simp] = *list.map-id*

1.3 merging functions

definition *fun-merge* :: ('a \Rightarrow 'b)list \Rightarrow 'a set list \Rightarrow 'a \Rightarrow 'b
where *fun-merge* fs as a \equiv (fs ! (LEAST i. i < length as \wedge a \in as ! i)) a

lemma *fun-merge*: **assumes**

i: $i < \text{length } as$

and *a*: $a \in as ! i$

and *ident*: $\bigwedge i j a. i < \text{length } as \implies j < \text{length } as \implies a \in as ! i \implies a \in as ! j$
 $\implies (fs ! i) a = (fs ! j) a$

shows *fun-merge* $fs \ as \ a = (fs ! i) \ a$

<proof>

lemma *fun-merge-part*: **assumes**

part: *is-partition* as

and *i*: $i < \text{length } as$

and *a*: $a \in as ! i$

shows *fun-merge* $fs \ as \ a = (fs ! i) \ a$

<proof>

lemma *map-nth-conv*: $\text{map } f \ ss = \text{map } g \ ts \implies \forall i < \text{length } ss. f(ss!i) = g(ts!i)$

<proof>

lemma *distinct-take-drop*:

assumes *dist*: *distinct* vs **and** *len*: $i < \text{length } vs$ **shows** *distinct*(*take* $i \ vs \ @ \ \text{drop}$
 $(\text{Suc } i) \ vs$) (**is** *distinct*($?xs @ ?ys$))

<proof>

lemma *map-nth-eq-conv*:

assumes *len*: $\text{length } xs = \text{length } ys$

shows $(\text{map } f \ xs = \text{map } g \ ys) = (\forall i < \text{length } ys. f(xs ! i) = g(ys ! i))$ (**is** $?l = ?r$)

<proof>

lemma *map-upt-len-conv*:

$\text{map } (\lambda i . f (xs!i)) [0..<\text{length } xs] = \text{map } f \ xs$

<proof>

lemma *map-upt-add'*:

$\text{map } f [a..<a+b] = \text{map } (\lambda i . f (a + i)) [0..<b]$

<proof>

definition *generate-lists* :: $\text{nat} \Rightarrow 'a \ \text{list} \Rightarrow 'a \ \text{list} \ \text{list}$

where *generate-lists* $n \ xs \equiv \text{concat-lists } (\text{map } (\lambda -. \ xs) [0 ..<n])$

lemma *set-generate-lists[simp]*: $\text{set } (\text{generate-lists } n \ xs) = \{as. \text{length } as = n \wedge \text{set } as \subseteq \text{set } xs\}$

<proof>

lemma *nth-append-take*:

assumes $i \leq \text{length } xs$ **shows** $(\text{take } i \ xs \ @ \ y \ # \ ys) ! i = y$

<proof>

lemma *nth-append-take-is-nth-conv*:

assumes $i < j$ **and** $j \leq \text{length } xs$ **shows** $(\text{take } j \text{ } xs @ ys)!i = xs!i$
(proof)

lemma *nth-append-drop-is-nth-conv*:

assumes $j < i$ **and** $j \leq \text{length } xs$ **and** $i \leq \text{length } xs$
shows $(\text{take } j \text{ } xs @ y \# \text{drop } (Suc \ j) \ xs)!i = xs!i$
(proof)

lemma *nth-append-take-drop-is-nth-conv*:

assumes $i \leq \text{length } xs$ **and** $j \leq \text{length } xs$ **and** $i \neq j$
shows $(\text{take } j \text{ } xs @ y \# \text{drop } (Suc \ j) \ xs)!i = xs!i$
(proof)

lemma *take-drop-imp-nth*: $\llbracket \text{take } i \text{ } ss @ x \# \text{drop } (Suc \ i) \ ss = ss \rrbracket \implies x = ss!i$
(proof)

lemma *take-drop-update-first*: **assumes** $j < \text{length } ds$ **and** $\text{length } cs = \text{length } ds$
shows $(\text{take } j \text{ } ds @ \text{drop } j \text{ } cs)[j := ds \ ! \ j] = \text{take } (Suc \ j) \ ds @ \text{drop } (Suc \ j) \ cs$
(proof)

lemma *take-drop-update-second*: **assumes** $j < \text{length } ds$ **and** $\text{length } cs = \text{length } ds$

shows $(\text{take } j \text{ } ds @ \text{drop } j \text{ } cs)[j := cs \ ! \ j] = \text{take } j \text{ } ds @ \text{drop } j \text{ } cs$
(proof)

lemma *nth-take-prefix*:

$\text{length } ys \leq \text{length } xs \implies \forall i < \text{length } ys. xs!i = ys!i \implies \text{take } (\text{length } ys) \ xs = ys$
(proof)

lemma *take-upt-idx*:

assumes $i < \text{length } ls$
shows $\text{take } i \text{ } ls = [ls \ ! \ j . j \leftarrow [0..<i]]$
(proof)

fun *distinct-eq* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$ **where**

$\text{distinct-eq} - [] = \text{True}$
 $|\ \text{distinct-eq } eq \ (x \# xs) = ((\forall y \in \text{set } xs. \neg (eq \ y \ x)) \wedge \text{distinct-eq } eq \ xs)$

lemma *distinct-eq-append*: $\text{distinct-eq } eq \ (xs @ ys) = (\text{distinct-eq } eq \ xs \wedge \text{distinct-eq } eq \ ys \wedge (\forall x \in \text{set } xs. \forall y \in \text{set } ys. \neg (eq \ y \ x)))$
(proof)

lemma *append-Cons-nth-left*:

assumes $i < \text{length } xs$
shows $(xs @ u \# ys) \ ! \ i = xs \ ! \ i$

<proof>

lemma *append-Cons-nth-middle*:

assumes $i = \text{length } xs$

shows $(xs @ y \# zs) ! i = y$

<proof>

lemma *append-Cons-nth-right*:

assumes $i > \text{length } xs$

shows $(xs @ u \# ys) ! i = (xs @ z \# ys) ! i$

<proof>

lemma *append-Cons-nth-not-middle*:

assumes $i \neq \text{length } xs$

shows $(xs @ u \# ys) ! i = (xs @ z \# ys) ! i$

<proof>

lemmas *append-Cons-nth = append-Cons-nth-middle append-Cons-nth-not-middle*

lemma *concat-all-nth*:

assumes $\text{length } xs = \text{length } ys$

and $\bigwedge i. i < \text{length } xs \implies \text{length } (xs ! i) = \text{length } (ys ! i)$

and $\bigwedge i j. i < \text{length } xs \implies j < \text{length } (xs ! i) \implies P (xs ! i ! j) (ys ! i ! j)$

shows $\forall k < \text{length } (\text{concat } xs). P (\text{concat } xs ! k) (\text{concat } ys ! k)$

<proof>

lemma *eq-length-concat-nth*:

assumes $\text{length } xs = \text{length } ys$

and $\bigwedge i. i < \text{length } xs \implies \text{length } (xs ! i) = \text{length } (ys ! i)$

shows $\text{length } (\text{concat } xs) = \text{length } (\text{concat } ys)$

<proof>

primrec

$\text{list-union} :: 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$

where

$\text{list-union } [] \text{ } ys = ys$

$| \text{list-union } (x \# xs) \text{ } ys = (\text{let } zs = \text{list-union } xs \text{ } ys \text{ in if } x \in \text{set } zs \text{ then } zs \text{ else } x \# zs)$

lemma *set-list-union[simp]*: $\text{set } (\text{list-union } xs \text{ } ys) = \text{set } xs \cup \text{set } ys$

<proof>

declare *list-union.simps[simp del]*

fun *list-inter* :: $'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**

$\text{list-inter } [] \text{ } bs = []$

$| \text{list-inter } (a \# as) \text{ } bs =$

$(\text{if } a \in \text{set } bs \text{ then } a \# \text{list-inter } as \text{ } bs \text{ else } \text{list-inter } as \text{ } bs)$

lemma *set-list-inter*[simp]:
 $set (list-inter\ xs\ ys) = set\ xs \cap set\ ys$
 ⟨proof⟩

declare *list-inter.simps*[simp del]

primrec *list-diff* :: 'a list ⇒ 'a list ⇒ 'a list **where**
 $list-diff\ []\ ys = []$
 $| list-diff\ (x\ \#\ xs)\ ys = (let\ zs = list-diff\ xs\ ys\ in\ if\ x \in set\ ys\ then\ zs\ else\ x\ \#\ zs)$

lemma *set-list-diff*[simp]:
 $set (list-diff\ xs\ ys) = set\ xs - set\ ys$
 ⟨proof⟩

declare *list-diff.simps*[simp del]

lemma *nth-drop-0*: $0 < length\ ss \implies (ss!0)\#\ drop\ (Suc\ 0)\ ss = ss$ ⟨proof⟩

lemma *set-foldr-remdups-set-map-conv*[simp]:
 $set (foldr\ (\lambda x\ xs.\ remdups\ (f\ x\ @\ xs))\ xs\ []) = \bigcup (set\ (map\ (set\ \circ\ f)\ xs))$
 ⟨proof⟩

lemma *subset-set-code*[code-unfold]: $set\ xs \subseteq set\ ys \longleftrightarrow list-all\ (\lambda x.\ x \in set\ ys)\ xs$
 ⟨proof⟩

fun *union-list-sorted* **where**
 $union-list-sorted\ (x\ \#\ xs)\ (y\ \#\ ys) =$
 $(if\ x = y\ then\ x\ \#\ union-list-sorted\ xs\ ys$
 $else\ if\ x < y\ then\ x\ \#\ union-list-sorted\ xs\ (y\ \#\ ys)$
 $else\ y\ \#\ union-list-sorted\ (x\ \#\ xs)\ ys)$
 $| union-list-sorted\ []\ ys = ys$
 $| union-list-sorted\ xs\ [] = xs$

lemma [simp]: $set (union-list-sorted\ xs\ ys) = set\ xs \cup set\ ys$
 ⟨proof⟩

fun *subtract-list-sorted* :: ('a :: linorder) list ⇒ 'a list ⇒ 'a list **where**
 $subtract-list-sorted\ (x\ \#\ xs)\ (y\ \#\ ys) =$
 $(if\ x = y\ then\ subtract-list-sorted\ xs\ (y\ \#\ ys)$
 $else\ if\ x < y\ then\ x\ \#\ subtract-list-sorted\ xs\ (y\ \#\ ys)$
 $else\ subtract-list-sorted\ (x\ \#\ xs)\ ys)$
 $| subtract-list-sorted\ []\ ys = []$
 $| subtract-list-sorted\ xs\ [] = xs$

lemma *set-subtract-list-sorted*[simp]: $sorted\ xs \implies sorted\ ys \implies$

set (subtract-list-sorted xs ys) = set xs - set ys
<proof>

lemma *subset-subtract-listed-sorted*: *set (subtract-list-sorted xs ys) ⊆ set xs*
<proof>

lemma *set-subtract-list-distinct[simp]*: *distinct xs ⇒ distinct (subtract-list-sorted xs ys)*
<proof>

definition *remdups-sort xs = remdups-adj (sort xs)*

lemma *remdups-sort[simp]*: *sorted (remdups-sort xs) set (remdups-sort xs) = set xs*
distinct (remdups-sort xs)
<proof>

maximum and minimum

lemma *max-list-mono*: **assumes** $\bigwedge x. x \in \text{set } xs - \text{set } ys \implies \exists y. y \in \text{set } ys \wedge x \leq y$
shows *max-list xs ≤ max-list ys*
<proof>

fun *min-list* :: ('a :: linorder) list ⇒ 'a **where**
min-list [x] = x
| min-list (x # xs) = min x (min-list xs)

lemma *min-list*: *(x :: 'a :: linorder) ∈ set xs ⇒ min-list xs ≤ x*
<proof>

lemma *min-list-Cons*:
assumes *xy*: *x ≤ y*
and *len*: *length xs = length ys*
and *xsys*: *min-list xs ≤ min-list ys*
shows *min-list (x # xs) ≤ min-list (y # ys)*
<proof>

lemma *min-list-nth*:
assumes *length xs = length ys*
and $\bigwedge i. i < \text{length } ys \implies xs ! i \leq ys ! i$
shows *min-list xs ≤ min-list ys*
<proof>

lemma *min-list-ex*:
assumes *xs ≠ []* **shows** $\exists x \in \text{set } xs. \text{min-list } xs = x$
<proof>

lemma *min-list-subset*:
assumes *subset*: *set ys ⊆ set xs* **and** *mem*: *min-list xs ∈ set ys*

shows $\text{min-list } xs = \text{min-list } ys$
 $\langle \text{proof} \rangle$

Apply a permutation to a list.

primrec $\text{permut-aux} :: 'a \text{ list} \Rightarrow (\text{nat} \Rightarrow \text{nat}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
 $\text{permut-aux } [] \ - \ - = [] \ |$
 $\text{permut-aux } (a \ \# \ as) \ f \ bs = (bs \ ! \ f \ 0) \ \# \ (\text{permut-aux } as \ (\lambda n. f \ (Suc \ n))) \ bs$

definition $\text{permut} :: 'a \text{ list} \Rightarrow (\text{nat} \Rightarrow \text{nat}) \Rightarrow 'a \text{ list}$ **where**
 $\text{permut } as \ f = \text{permut-aux } as \ f \ as$
declare $\text{permut-def}[\text{simp}]$

lemma permut-aux-sound :
assumes $i < \text{length } as$
shows $\text{permut-aux } as \ f \ bs \ ! \ i = bs \ ! \ (f \ i)$
 $\langle \text{proof} \rangle$

lemma permut-sound :
assumes $i < \text{length } as$
shows $\text{permut } as \ f \ ! \ i = as \ ! \ (f \ i)$
 $\langle \text{proof} \rangle$

lemma permut-aux-length :
assumes $\text{bij-betw } f \ \{.. < \text{length } as\} \ \{.. < \text{length } bs\}$
shows $\text{length } (\text{permut-aux } as \ f \ bs) = \text{length } as$
 $\langle \text{proof} \rangle$

lemma permut-length :
assumes $\text{bij-betw } f \ \{.. < \text{length } as\} \ \{.. < \text{length } as\}$
shows $\text{length } (\text{permut } as \ f) = \text{length } as$
 $\langle \text{proof} \rangle$

declare $\text{permut-def}[\text{simp del}]$

lemma foldl-assoc :
fixes $b :: ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$ (**infixl** \cdot 55)
assumes $\bigwedge f \ g \ h. f \cdot (g \cdot h) = f \cdot g \cdot h$
shows $\text{foldl } (\cdot) \ (x \cdot y) \ zs = x \cdot \text{foldl } (\cdot) \ y \ zs$
 $\langle \text{proof} \rangle$

lemma foldr-assoc :
assumes $\bigwedge f \ g \ h. b \ (b \ f \ g) \ h = b \ f \ (b \ g \ h)$
shows $\text{foldr } b \ xs \ (b \ y \ z) = b \ (\text{foldr } b \ xs \ y) \ z$
 $\langle \text{proof} \rangle$

lemma foldl-foldr-o-id :
 $\text{foldl } (\circ) \ id \ fs = \text{foldr } (\circ) \ fs \ id$
 $\langle \text{proof} \rangle$

lemma *foldr-o-o-id*[simp]:
 $foldr ((\circ) \circ f) xs id a = foldr f xs a$
 ⟨proof⟩

lemma *Ex-list-of-length-P*:
assumes $\forall i < n. \exists x. P x i$
shows $\exists xs. length\ xs = n \wedge (\forall i < n. P (xs\ !\ i)\ i)$
 ⟨proof⟩

lemma *ex-set-conv-ex-nth*: $(\exists x \in set\ xs. P\ x) = (\exists i < length\ xs. P (xs\ !\ i))$
 ⟨proof⟩

lemma *map-eq-set-zipD* [dest]:
assumes $map\ f\ xs = map\ f\ ys$
and $(x, y) \in set\ (zip\ xs\ ys)$
shows $f\ x = f\ y$
 ⟨proof⟩

fun *span* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \times 'a\ list$ **where**
 $span\ P\ (x\ \#\ xs) =$
 $(if\ P\ x\ then\ let\ (ys,\ zs) = span\ P\ xs\ in\ (x\ \#\ ys,\ zs))$
 $else\ ([],\ x\ \#\ xs) \mid$
 $span\ -\ [] = ([], [])$

lemma *span*[simp]: $span\ P\ xs = (takeWhile\ P\ xs,\ dropWhile\ P\ xs)$
 ⟨proof⟩

declare *span.simps*[simp del]

lemma *parallel-list-update*: **assumes**
one-update: $\bigwedge xs\ i\ y. length\ xs = n \Longrightarrow i < n \Longrightarrow r\ (xs\ !\ i)\ y \Longrightarrow p\ xs \Longrightarrow p\ (xs[i := y])$
and *init*: $length\ xs = n \wedge p\ xs$
and *rel*: $length\ ys = n \wedge i. i < n \Longrightarrow r\ (xs\ !\ i)\ (ys\ !\ i)$
shows $p\ ys$
 ⟨proof⟩

lemma *nth-concat-two-lists*:
 $i < length\ (concat\ (xs :: 'a\ list\ list)) \Longrightarrow length\ (ys :: 'b\ list\ list) = length\ xs$
 $\Longrightarrow (\bigwedge i. i < length\ xs \Longrightarrow length\ (ys\ !\ i) = length\ (xs\ !\ i))$
 $\Longrightarrow \exists j\ k. j < length\ xs \wedge k < length\ (xs\ !\ j) \wedge (concat\ xs)\ !\ i = xs\ !\ j\ !\ k \wedge$
 $(concat\ ys)\ !\ i = ys\ !\ j\ !\ k$
 ⟨proof⟩

Removing duplicates w.r.t. some function.

fun *remdups-gen* :: $('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $remdups-gen\ f\ [] = []$
 $\mid remdups-gen\ f\ (x\ \#\ xs) = x\ \#\ remdups-gen\ f\ [y <- xs. \neg f\ x = f\ y]$

lemma *remdups-gen-subset*: $set (remdups-gen f xs) \subseteq set xs$
 ⟨proof⟩

lemma *remdups-gen-elim-imp-elim*: $x \in set (remdups-gen f xs) \implies x \in set xs$
 ⟨proof⟩

lemma *elim-imp-remdups-gen-elim*: $x \in set xs \implies \exists y \in set (remdups-gen f xs).$
 $f x = f y$
 ⟨proof⟩

lemma *take-nth-drop-concat*:
 assumes $i < length xss$ and $xss ! i = ys$
 and $j < length ys$ and $ys ! j = z$
 shows $\exists k < length (concat xss).$
 $take k (concat xss) = concat (take i xss) @ take j ys \wedge$
 $concat xss ! k = xss ! i ! j \wedge$
 $drop (Suc k) (concat xss) = drop (Suc j) ys @ concat (drop (Suc i) xss)$
 ⟨proof⟩

lemma *concat-map-empty* [simp]:
 $concat (map (_. []) xs) = []$
 ⟨proof⟩

lemma *map-upt-len-same-len-conv*:
 assumes $length xs = length ys$
 shows $map (\lambda i. f (xs ! i)) [0 ..< length ys] = map f xs$
 ⟨proof⟩

lemma *concat-map-concat* [simp]:
 $concat (map concat xs) = concat (concat xs)$
 ⟨proof⟩

lemma *concat-concat-map*:
 $concat (concat (map f xs)) = concat (map (concat \circ f) xs)$
 ⟨proof⟩

lemma *UN-upt-len-conv* [simp]:
 $length xs = n \implies (\bigcup i \in \{0 ..< n\}. f (xs ! i)) = \bigcup (set (map f xs))$
 ⟨proof⟩

lemma *Ball-at-Least0LessThan-conv* [simp]:
 $length xs = n \implies$
 $(\forall i \in \{0 ..< n\}. P (xs ! i)) \longleftrightarrow (\forall x \in set xs. P x)$
 ⟨proof⟩

lemma *sum-list-replicate-length* [simp]:
 $sum-list (replicate (length xs) (Suc 0)) = length xs$
 ⟨proof⟩

lemma *list-all2-in-set2*:

assumes *list-all2* P xs ys **and** $y \in \text{set } ys$
obtains x **where** $x \in \text{set } xs$ **and** P x y
<proof>

lemma *map-eq-conv'*:

$\text{map } f$ $xs = \text{map } g$ $ys \iff \text{length } xs = \text{length } ys \wedge (\forall i < \text{length } xs. f (xs ! i) = g (ys ! i))$
<proof>

lemma *list-3-cases*[*case-names Nil 1 2*]:

assumes $xs = [] \implies P$
and $\bigwedge x. xs = [x] \implies P$
and $\bigwedge x y ys. xs = x \# y \# ys \implies P$
shows P
<proof>

lemma *list-4-cases*[*case-names Nil 1 2 3*]:

assumes $xs = [] \implies P$
and $\bigwedge x. xs = [x] \implies P$
and $\bigwedge x y. xs = [x, y] \implies P$
and $\bigwedge x y z zs. xs = x \# y \# z \# zs \implies P$
shows P
<proof>

lemma *foldr-append2* [*simp*]:

$\text{foldr } ((@) \circ f) xs (ys @ zs) = \text{foldr } ((@) \circ f) xs ys @ zs$
<proof>

lemma *foldr-append2-Nil* [*simp*]:

$\text{foldr } ((@) \circ f) xs [] @ zs = \text{foldr } ((@) \circ f) xs zs$
<proof>

lemma *UNION-set-zip*:

$(\bigcup x \in \text{set } (\text{zip } [0..<\text{length } xs] (\text{map } f xs)). g x) = (\bigcup i < \text{length } xs. g (i, f (xs ! i)))$
<proof>

lemma *zip-fst*: $p \in \text{set } (\text{zip } as bs) \implies \text{fst } p \in \text{set } as$

<proof>

lemma *zip-snd*: $p \in \text{set } (\text{zip } as bs) \implies \text{snd } p \in \text{set } bs$

<proof>

lemma *zip-size-aux*: $\text{size-list } (\text{size } o \text{snd}) (\text{zip } ts ls) \leq (\text{size-list } \text{size } ls)$

<proof>

We define the function that remove the nth element of a list. It uses

take and drop and the soundness is therefore not too hard to prove thanks to the already existing lemmas.

definition *remove-nth* :: *nat* \Rightarrow 'a list \Rightarrow 'a list **where**
remove-nth *n xs* \equiv (*take* *n xs*) @ (*drop* (*Suc* *n*) *xs*)

declare *remove-nth-def*[*simp*]

lemma *remove-nth-len*:
assumes *i*: *i* < *length xs*
shows *length xs* = *Suc* (*length* (*remove-nth* *i xs*))
<proof>

lemma *remove-nth-length* :
assumes *n-bd*: *n* < *length xs*
shows *length* (*remove-nth* *n xs*) = *length xs* - 1
<proof>

lemma *remove-nth-id* : *length xs* \leq *n* \implies *remove-nth* *n xs* = *xs*
<proof>

lemma *remove-nth-sound-l* :
assumes *p-ub*: *p* < *n*
shows (*remove-nth* *n xs*) ! *p* = *xs* ! *p*
<proof>

lemma *remove-nth-sound-r* :
assumes *n* \leq *p* **and** *p* < *length xs*
shows (*remove-nth* *n xs*) ! *p* = *xs* ! (*Suc* *p*)
<proof>

lemma *nth-remove-nth-conv*:
assumes *i* < *length* (*remove-nth* *n xs*)
shows *remove-nth* *n xs* ! *i* = *xs* ! (*if* *i* < *n* *then* *i* *else* *Suc* *i*)
<proof>

lemma *remove-nth-P-compat* :
assumes *aslbs*: *length as* = *length bs*
and *Pab*: $\forall i. i < \text{length } as \longrightarrow P (as ! i) (bs ! i)$
shows $\forall i. i < \text{length} (remove_nth\ p\ as) \longrightarrow P (remove_nth\ p\ as ! i) (remove_nth\ p\ bs ! i)$
<proof>

declare *remove-nth-def*[*simp del*]

definition *adjust-idx* :: *nat* \Rightarrow *nat* \Rightarrow *nat* **where**
adjust-idx *i j* \equiv (*if* *j* < *i* *then* *j* *else* (*Suc* *j*))

definition *adjust-idx-rev* :: *nat* \Rightarrow *nat* \Rightarrow *nat* **where**
adjust-idx-rev *i j* \equiv (*if* *j* < *i* *then* *j* *else* *j* - *Suc* 0)

lemma *adjust-idx-rev1*: $\text{adjust-idx-rev } i \ (\text{adjust-idx } i \ j) = j$
 ⟨*proof*⟩

lemma *adjust-idx-rev2*:
assumes $j \neq i$ **shows** $\text{adjust-idx } i \ (\text{adjust-idx-rev } i \ j) = j$
 ⟨*proof*⟩

lemma *adjust-idx-i*:
 $\text{adjust-idx } i \ j \neq i$
 ⟨*proof*⟩

lemma *adjust-idx-nth*:
assumes $i < \text{length } xs$
shows $\text{remove-nth } i \ xs \ ! \ j = xs \ ! \ \text{adjust-idx } i \ j$ (**is** $?l = ?r$)
 ⟨*proof*⟩

lemma *adjust-idx-rev-nth*:
assumes $i < \text{length } xs$
and $ji: j \neq i$
shows $\text{remove-nth } i \ xs \ ! \ \text{adjust-idx-rev } i \ j = xs \ ! \ j$ (**is** $?l = ?r$)
 ⟨*proof*⟩

lemma *adjust-idx-length*:
assumes $i < \text{length } xs$
and $j: j < \text{length } (\text{remove-nth } i \ xs)$
shows $\text{adjust-idx } i \ j < \text{length } xs$
 ⟨*proof*⟩

lemma *adjust-idx-rev-length*:
assumes $i < \text{length } xs$
and $j < \text{length } xs$
and $j \neq i$
shows $\text{adjust-idx-rev } i \ j < \text{length } (\text{remove-nth } i \ xs)$
 ⟨*proof*⟩

If a binary relation holds on two couples of lists, then it holds on the concatenation of the two couples.

lemma *P-as-bs-extend*:
assumes $lab: \text{length } as = \text{length } bs$
and $lcd: \text{length } cs = \text{length } ds$
and $nsab: \forall i. i < \text{length } bs \longrightarrow P \ (as \ ! \ i) \ (bs \ ! \ i)$
and $nscd: \forall i. i < \text{length } ds \longrightarrow P \ (cs \ ! \ i) \ (ds \ ! \ i)$
shows $\forall i. i < \text{length } (bs \ @ \ ds) \longrightarrow P \ ((as \ @ \ cs) \ ! \ i) \ ((bs \ @ \ ds) \ ! \ i)$
 ⟨*proof*⟩

Extension of filter and partition to binary relations.

fun *filter2* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \ \text{list} \Rightarrow 'b \ \text{list} \Rightarrow ('a \ \text{list} \times 'b \ \text{list})$ **where**
filter2 $P \ [] \ - = ([], []) \ |$


```

filter2 P - [] = ([], []) |
filter2 P (a # as) (b # bs) = (if P a b
  then (a # fst (filter2 P as bs), b # snd (filter2 P as bs))
  else filter2 P as bs)

```

lemma *filter2-length*:

```

length (fst (filter2 P as bs)) ≡ length (snd (filter2 P as bs))
⟨proof⟩

```

lemma *filter2-sound*: $\forall i. i < \text{length} (\text{fst} (\text{filter2 } P \text{ as } bs)) \longrightarrow P (\text{fst} (\text{filter2 } P \text{ as } bs) ! i) (\text{snd} (\text{filter2 } P \text{ as } bs) ! i)$
⟨proof⟩

definition *partition2* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow ('a \text{ list} \times 'b \text{ list}) \times ('a \text{ list} \times 'b \text{ list})$ **where**
partition2 P as bs ≡ ((filter2 P as bs) , (filter2 ($\lambda a b. \neg (P a b)$) as bs))

lemma *partition2-sound-P*: $\forall i. i < \text{length} (\text{fst} (\text{fst} (\text{partition2 } P \text{ as } bs))) \longrightarrow P (\text{fst} (\text{fst} (\text{partition2 } P \text{ as } bs)) ! i) (\text{snd} (\text{fst} (\text{partition2 } P \text{ as } bs)) ! i)$
⟨proof⟩

lemma *partition2-sound-nP*: $\forall i. i < \text{length} (\text{fst} (\text{snd} (\text{partition2 } P \text{ as } bs))) \longrightarrow \neg P (\text{fst} (\text{snd} (\text{partition2 } P \text{ as } bs)) ! i) (\text{snd} (\text{snd} (\text{partition2 } P \text{ as } bs)) ! i)$
⟨proof⟩

Membership decision function that actually returns the value of the index where the value can be found.

```

fun mem-idx :: 'a ⇒ 'a list ⇒ nat Option.option where
  mem-idx - [] = None |
  mem-idx x (a # as) = (if x = a then Some 0 else map-option Suc (mem-idx x as))

```

lemma *mem-idx-sound-output*:

```

assumes mem-idx x as = Some i
shows i < length as ∧ as ! i = x
⟨proof⟩

```

lemma *mem-idx-sound-output2*:

```

assumes mem-idx x as = Some i
shows  $\forall j. j < i \longrightarrow as ! j \neq x$ 
⟨proof⟩

```

lemma *mem-idx-sound*:

```

(x ∈ set as) = (∃ i. mem-idx x as = Some i)
⟨proof⟩

```

lemma *mem-idx-sound2*:

```

(x ∉ set as) = (mem-idx x as = None)
⟨proof⟩

```

lemma *sum-list-replicate-mono*: **assumes** $w1 \leq (w2 :: nat)$
shows $sum\text{-list } (replicate\ n\ w1) \leq sum\text{-list } (replicate\ n\ w2)$
 $\langle proof \rangle$

lemma *all-gt-0-sum-list-map*:
assumes $*$: $\bigwedge x. f\ x > (0 :: nat)$
and $x: x \in set\ xs$ **and** $len: 1 < length\ xs$
shows $f\ x < (\sum x \leftarrow xs. f\ x)$
 $\langle proof \rangle$

lemma *finite-distinct*: $finite\ \{ xs . distinct\ xs \wedge set\ xs = X \}$ **(is finite** $(?S\ X)$
 $\langle proof \rangle$

lemma *finite-distinct-subset*:
assumes $finite\ X$
shows $finite\ \{ xs . distinct\ xs \wedge set\ xs \subseteq X \}$ **(is finite** $(?S\ X)$
 $\langle proof \rangle$

lemma *map-of-filter*:
assumes $P\ x$
shows $map\text{-of } [(x',y) \leftarrow ys. P\ x']\ x = map\text{-of } ys\ x$
 $\langle proof \rangle$

lemma *set-subset-insertI*: $set\ xs \subseteq set\ (List.insert\ x\ xs)$ $\langle proof \rangle$
lemma *set-removeAll-subset*: $set\ (removeAll\ x\ xs) \subseteq set\ xs$ $\langle proof \rangle$

lemma *map-of-append-Some*:
 $map\text{-of } xs\ y = Some\ z \implies map\text{-of } (xs\ @\ ys)\ y = Some\ z$
 $\langle proof \rangle$

lemma *map-of-append-None*:
 $map\text{-of } xs\ y = None \implies map\text{-of } (xs\ @\ ys)\ y = map\text{-of } ys\ y$
 $\langle proof \rangle$

end

2 Preliminaries

2.1 Missing Multiset

This theory provides some definitions and lemmas on multisets which we did not find in the Isabelle distribution.

theory *Missing-Multiset*
imports
HOL-Library.Multiset
Missing-List

begin

lemma *remove-nth-soundness*:

assumes $n < \text{length } as$

shows $mset (\text{remove-nth } n \ as) = mset \ as - \{\#(as!n)\# \}$

<proof>

lemma *multiset-subset-insert*: $\{ps. ps \subseteq\# \text{add-mset } x \ xs\} =$

$\{ps. ps \subseteq\# xs\} \cup \text{add-mset } x \ \{ps. ps \subseteq\# xs\} \ (\text{is } ?l = ?r)$

<proof>

lemma *multiset-of-subseqs*: $mset \ ' \ set \ (\text{subseqs } xs) = \{ps. ps \subseteq\# mset \ xs\}$

<proof>

lemma *remove1-mset*: $w \in \text{set } vs \implies mset (\text{remove1 } w \ vs) + \{\#w\# \} = mset \ vs$

<proof>

lemma *fold-remove1-mset*: $mset \ ws \subseteq\# mset \ vs \implies mset (\text{fold } \text{remove1 } \ ws \ vs) + mset \ ws = mset \ vs$

<proof>

lemma *subseqs-sub-mset*: $ws \in \text{set } (\text{subseqs } vs) \implies mset \ ws \subseteq\# mset \ vs$

<proof>

lemma *filter-mset-inequality*: $\text{filter-mset } f \ xs \neq xs \implies \exists x \in\# xs. \neg f \ x$

<proof>

end

2.2 Precomputation

This theory contains precomputation functions, which take another function f and a finite set of inputs, and provide the same function f as output, except that now all values $f \ i$ are precomputed if i is contained in the set of finite inputs.

theory *Precomputation*

imports

Containers.RBT-Set2

HOL-Library.RBT-Mapping

begin

lemma *lookup-tabulate*: $x \in \text{set } xs \implies \text{Mapping.lookup } (\text{Mapping.tabulate } xs \ f) \ x = \text{Some } (f \ x)$

<proof>

lemma *lookup-tabulate2*: $\text{Mapping.lookup } (\text{Mapping.tabulate } xs \ f) \ x = \text{Some } y \implies y = f \ x$

<proof>

definition *memo-int* :: $int \Rightarrow int \Rightarrow (int \Rightarrow 'a) \Rightarrow (int \Rightarrow 'a)$ **where**
memo-int low up f \equiv *let m = Mapping.tabulate [low .. up] f*
in ($\lambda x.$ *if* $x \geq low \wedge x \leq up$ *then the* (*Mapping.lookup m x*) *else f x*)

lemma *memo-int[simp]*: *memo-int low up f = f*
 \langle *proof* \rangle

definition *memo-nat* :: $nat \Rightarrow nat \Rightarrow (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a)$ **where**
memo-nat low up f \equiv *let m = Mapping.tabulate [low ..< up] f*
in ($\lambda x.$ *if* $x \geq low \wedge x < up$ *then the* (*Mapping.lookup m x*) *else f x*)

lemma *memo-nat[simp]*: *memo-nat low up f = f*
 \langle *proof* \rangle

definition *memo* :: $'a\ list \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$ **where**
memo xs f \equiv *let m = Mapping.tabulate xs f*
in ($\lambda x.$ *case* *Mapping.lookup m x of None* \Rightarrow *f x* | *Some y* \Rightarrow *y*)

lemma *memo[simp]*: *memo xs f = f*
 \langle *proof* \rangle

end

2.3 Order of Polynomial Roots

We extend the collection of results on the order of roots of polynomials. Moreover, we provide code-equations to compute the order for a given root and polynomial.

theory *Order-Polynomial*

imports

Polynomial-Interpolation.Missing-Polynomial

begin

lemma *order-linear[simp]*: *order a* $[: - a, 1:] = \text{Suc } 0$ \langle *proof* \rangle

declare *order-power-n-n[simp]*

lemma *linear-power-nonzero*: $[: a, 1:] \wedge n \neq 0$
 \langle *proof* \rangle

lemma *order-linear-power'*: *order a* $([: b, 1:] \wedge \text{Suc } n) = (\text{if } b = -a \text{ then } \text{Suc } n \text{ else } 0)$
 \langle *proof* \rangle

lemma *order-linear-power*: *order a* $([: b, 1:] \wedge n) = (\text{if } b = -a \text{ then } n \text{ else } 0)$
 \langle *proof* \rangle

lemma *order-linear'*: $\text{order } a \text{ } [: b, 1:] = (\text{if } b = -a \text{ then } 1 \text{ else } 0)$
 ⟨proof⟩

lemma *degree-div-less*:
assumes $p: (p::'a::\text{field poly}) \neq 0$ **and** $\text{dvd}: r \text{ dvd } p$ **and** $\text{deg}: \text{degree } r \neq 0$
shows $\text{degree } (p \text{ div } r) < \text{degree } p$
 ⟨proof⟩

lemma *order-sum-degree*: **assumes** $p \neq 0$
shows $\text{sum } (\lambda a. \text{order } a \text{ } p) \{ a. \text{poly } p \text{ } a = 0 \} \leq \text{degree } p$
 ⟨proof⟩

lemma *order-code*[code]: $\text{order } (a::'a::\text{idom-divide}) \text{ } p =$
 (if $p = 0$ then $\text{Code.abort } (\text{STR } \text{"order of polynomial 0 undefined"}) (\lambda _ . \text{order } a \text{ } p)$
 else if $\text{poly } p \text{ } a \neq 0$ then 0 else $\text{Suc } (\text{order } a \text{ } (p \text{ div } [: -a, 1 :]))$)
 ⟨proof⟩

end

3 Explicit Formulas for Roots

We provide algorithms which use the explicit formulas to compute the roots of polynomials of degree up to 2. For polynomials of degree 3 and 4 have a look at the AFP entry "Cubic-Quartic-Equations".

theory *Explicit-Roots*

imports

Polynomial-Interpolation.Missing-Polynomial

Sqrt-Babylonian.Sqrt-Babylonian

begin

lemma *roots0*: **assumes** $p: p \neq 0$ **and** $p0: \text{degree } p = 0$
shows $\{x. \text{poly } p \text{ } x = 0\} = \{\}$
 ⟨proof⟩

definition *roots1* :: $'a :: \text{field poly} \Rightarrow 'a$ **where**
 $\text{roots1 } p = (- \text{coeff } p \text{ } 0 / \text{coeff } p \text{ } 1)$

lemma *roots1*: **fixes** $p :: 'a :: \text{field poly}$
assumes $p1: \text{degree } p = 1$
shows $\{x. \text{poly } p \text{ } x = 0\} = \{\text{roots1 } p\}$
 ⟨proof⟩

lemma *roots2*: **fixes** $p :: 'a :: \text{field-char-0 poly}$
assumes $p2: p = [: c, b, a :]$ **and** $a: a \neq 0$
shows $\{x. \text{poly } p \text{ } x = 0\} = \{ - (b / (2 * a)) + e \mid e. e^2 = (b / (2 * a))^2$

$- c/a$ } (is ?l = ?r)
 <proof>

definition *croots2* :: complex poly \Rightarrow complex list **where**

croots2 p = (let a = coeff p 2; b = coeff p 1; c = coeff p 0; b2a = b / (2 * a);
 bac = b2a² - c/a;
 e = csqrt bac
 in
 remdups [- b2a + e, - b2a - e])

definition *complex-rat* :: complex \Rightarrow bool **where**

complex-rat x = (Re x \in \mathbf{Q} \wedge Im x \in \mathbf{Q})

lemma *croots2*: **assumes** degree p = 2

shows {x. poly p x = 0} = set (croots2 p)

<proof>

definition *rroots2* :: real poly \Rightarrow real list **where**

rroots2 p = (let a = coeff p 2; b = coeff p 1; c = coeff p 0; b2a = b / (2 * a);
 bac = b2a² - c/a
 in if bac = 0 then [- b2a] else if bac < 0 then []
 else let e = sqrt bac
 in
 [- b2a + e, - b2a - e])

definition *rat-roots2* :: rat poly \Rightarrow rat list **where**

rat-roots2 p = (let a = coeff p 2; b = coeff p 1; c = coeff p 0; b2a = b / (2 * a);
 bac = b2a² - c/a
 in map (λ e. - b2a + e) (sqrt-rat bac))

lemma *rroots2*: **assumes** degree p = 2

shows {x. poly p x = 0} = set (rroots2 p)

<proof>

lemma *rat-roots2*: **assumes** degree p = 2

shows {x. poly p x = 0} = set (rat-roots2 p)

<proof>

Determining roots of complex polynomials of degree up to 2.

definition *croots* :: complex poly \Rightarrow complex list **where**

croots p = (if p = 0 \vee degree p > 2 then []
 else (if degree p = 0 then [] else if degree p = 1 then [roots1 p]
 else croots2 p))

lemma *croots*: **assumes** p \neq 0 degree p \leq 2

shows set (croots p) = {x. poly p x = 0}

<proof>

Determining roots of real polynomials of degree up to 2.

definition *rroots* :: real poly \Rightarrow real list **where**

```

rroots p = (if p = 0 ∨ degree p > 2 then []
  else (if degree p = 0 then [] else if degree p = 1 then [roots1 p]
  else rroots2 p))

```

lemma *rroots*: **assumes** $p \neq 0$ *degree* $p \leq 2$
shows $\text{set } (rroots\ p) = \{x. \text{poly } p\ x = 0\}$
<proof>

end

4 Division of Polynomials over Integers

This theory contains an algorithm to efficiently compute divisibility of two integer polynomials.

theory *Dvd-Int-Poly*

imports

Polynomial-Interpolation.Ring-Hom-Poly

Polynomial-Interpolation.Divmod-Int

Polynomial-Interpolation.Is-Rat-To-Rat

begin

definition *div-int-poly-step* :: *int poly* \Rightarrow *int* \Rightarrow (*int poly* \times *int poly*) *option* \Rightarrow (*int poly* \times *int poly*) *option* **where**

```

  div-int-poly-step q = ( $\lambda a\ s\ r.$  case sro of Some (s, r)  $\Rightarrow$ 
    let ar = pCons a r; (b,m) = divmod-int (coeff ar (degree q)) (coeff q (degree q))
    in if m = 0 then Some (pCons b s, ar - smult b q) else None | None  $\Rightarrow$  None)

```

declare *div-int-poly-step-def*[code-unfold]

definition *div-mod-int-poly* :: *int poly* \Rightarrow *int poly* \Rightarrow (*int poly* \times *int poly*) *option* **where**

```

div-mod-int-poly p q = (if q = 0 then None
  else (let n = degree q; qn = coeff q n
  in fold-coeffs (div-int-poly-step q) p (Some (0, 0))))

```

definition *div-int-poly* :: *int poly* \Rightarrow *int poly* \Rightarrow *int poly option* **where**

```

div-int-poly p q =
  (case div-mod-int-poly p q of None  $\Rightarrow$  None | Some (d,m)  $\Rightarrow$  if m = 0 then
  Some d else None)

```

definition *div-rat-poly-step* :: '*a*::*field* *poly* \Rightarrow '*a* \Rightarrow '*a poly* \times '*a poly* \Rightarrow '*a poly* \times '*a poly* **where**

```

div-rat-poly-step q = ( $\lambda a\ (s, r).$ 
  let b = coeff (pCons a r) (degree q) / coeff q (degree q)
  in (pCons b s, pCons a r - smult b q))

```

lemma *foldr-cong-plus*:

assumes $f\text{-is-}g : \bigwedge a\ b\ c. b \in s \implies f' a = f b (f' c) \implies g' a = g b (g' c)$
and $f'\text{-inj} : \bigwedge a\ b. f' a = f' b \implies a = b$
and $f\text{-bit-sur} : \bigwedge a\ b\ c. f' a = f b c \implies \exists c'. c = f' c'$
and $\text{lst-in-}s : \text{set } \text{lst} \subseteq s$
shows $f' a = \text{foldr } f\ \text{lst} (f' b) \implies g' a = \text{foldr } g\ \text{lst} (g' b)$
 $\langle \text{proof} \rangle$

abbreviation (*input*) $rp :: \text{int poly} \Rightarrow \text{rat poly}$ **where**
 $rp \equiv \text{map-poly rat-of-int}$

lemma $\text{rat-int-poly-step-agree} :$
assumes $\text{coeff } (p\text{Cons } b\ c2) (\text{degree } q) \bmod \text{coeff } q (\text{degree } q) = 0$
shows $(rp\ a1, rp\ a2) = (\text{div-rat-poly-step } (rp\ q) \circ \text{rat-of-int})\ b (rp\ c1, rp\ c2)$
 $\longleftrightarrow \text{Some } (a1, a2) = \text{div-int-poly-step } q\ b (\text{Some } (c1, c2))$
 $\langle \text{proof} \rangle$

lemma $\text{int-step-then-rat-poly-step} :$
assumes $\text{Some} : \text{Some } (a1, a2) = \text{div-int-poly-step } q\ b (\text{Some } (c1, c2))$
shows $(rp\ a1, rp\ a2) = (\text{div-rat-poly-step } (rp\ q) \circ \text{rat-of-int})\ b (rp\ c1, rp\ c2)$
 $\langle \text{proof} \rangle$

lemma $\text{is-int-rat-division} :$
assumes $y \neq 0$
shows $\text{is-int-rat } (\text{rat-of-int } x / \text{rat-of-int } y) \longleftrightarrow x \bmod y = 0$
 $\langle \text{proof} \rangle$

lemma $p\text{Cons-of-rp-contains-ints} :$
assumes $rp\ a = p\text{Cons } b\ c$
shows $\text{is-int-rat } b$
 $\langle \text{proof} \rangle$

lemma $\text{rat-step-then-int-poly-step} :$
assumes $q \neq 0$
and $(rp\ a1, rp\ a2) = (\text{div-rat-poly-step } (rp\ q) \circ \text{rat-of-int})\ b2 (rp\ c1, rp\ c2)$
shows $\text{Some } (a1, a2) = \text{div-int-poly-step } q\ b2 (\text{Some } (c1, c2))$
 $\langle \text{proof} \rangle$

lemma $\text{div-int-poly-step-surjective} : \text{Some } a = \text{div-int-poly-step } q\ b\ c \implies \exists c'. c = \text{Some } c'$
 $\langle \text{proof} \rangle$

lemma $\text{div-mod-int-poly-then-pdivmod} :$
assumes $\text{div-mod-int-poly } p\ q = \text{Some } (r, m)$
shows $(rp\ p\ \text{div } rp\ q, rp\ p\ \text{mod } rp\ q) = (rp\ r, rp\ m)$
and $q \neq 0$
 $\langle \text{proof} \rangle$

lemma $\text{div-rat-poly-step-sur} :$

assumes $(\text{case } a \text{ of } (a, b) \Rightarrow (rp\ a, rp\ b)) = (\text{div-rat-poly-step } (rp\ q) \circ \text{rat-of-int})$
 $x\ \text{pair}$

shows $\exists c'. \text{pair} = (\text{case } c' \text{ of } (a, b) \Rightarrow (rp\ a, rp\ b))$
 $\langle \text{proof} \rangle$

lemma *pdivmod-then-div-mod-int-poly*:

assumes $q0: q \neq 0$ **and** $(rp\ p\ \text{div}\ rp\ q, rp\ p\ \text{mod}\ rp\ q) = (rp\ r, rp\ m)$
shows $\text{div-mod-int-poly } p\ q = \text{Some } (r, m)$
 $\langle \text{proof} \rangle$

lemma *div-int-then-rqp*:

assumes $\text{div-int-poly } p\ q = \text{Some } r$
shows $r * q = p$
and $q \neq 0$
 $\langle \text{proof} \rangle$

lemma *rqp-then-div-int*:

assumes $r * q = p$
and $q0: q \neq 0$
shows $\text{div-int-poly } p\ q = \text{Some } r$
 $\langle \text{proof} \rangle$

lemma *div-int-poly*: $(\text{div-int-poly } p\ q = \text{Some } r) \longleftrightarrow (q \neq 0 \wedge p = r * q)$
 $\langle \text{proof} \rangle$

definition *dvd-int-poly* :: $\text{int poly} \Rightarrow \text{int poly} \Rightarrow \text{bool}$ **where**

$\text{dvd-int-poly } q\ p = (\text{if } q = 0 \text{ then } p = 0 \text{ else } \text{div-int-poly } p\ q \neq \text{None})$

lemma *dvd-int-poly[simp]*: $\text{dvd-int-poly } q\ p = (q\ \text{dvd}\ p)$
 $\langle \text{proof} \rangle$

definition *dvd-int-poly-non-0* :: $\text{int poly} \Rightarrow \text{int poly} \Rightarrow \text{bool}$ **where**

$\text{dvd-int-poly-non-0 } q\ p = (\text{div-int-poly } p\ q \neq \text{None})$

lemma *dvd-int-poly-non-0[simp]*: $q \neq 0 \implies \text{dvd-int-poly-non-0 } q\ p = (q\ \text{dvd}\ p)$
 $\langle \text{proof} \rangle$

lemma *[code-unfold]*: $p\ \text{dvd}\ q \longleftrightarrow \text{dvd-int-poly } p\ q$ $\langle \text{proof} \rangle$

hide-const *rp*
end

5 More on Polynomials

This theory contains several results on content, gcd, primitive part, etc.. Moreover, there is a slightly improved code-equation for computing the gcd.

theory *Missing-Polynomial-Factorial*

imports *HOL-Computational-Algebra.Polynomial-Factorial*

begin

Improved code equation for *gcd-poly-code* which avoids computing the content twice.

lemma *gcd-poly-code-code*[code]: *gcd-poly-code* p q =
 (if p = 0 then normalize q else if q = 0 then normalize p else let
 c1 = content p;
 c2 = content q;
 p' = map-poly (λ x. x div c1) p;
 q' = map-poly (λ x. x div c2) q
 in smult (gcd c1 c2) (gcd-poly-code-aux p' q'))
 ⟨proof⟩

lemma *gcd-smult*: **fixes** f g :: 'a :: {factorial-ring-gcd,semiring-gcd-mult-normalize}
poly
defines cf: cf ≡ content f
and cg: cg ≡ content g
shows gcd (smult a f) g = (if a = 0 ∨ f = 0 then normalize g else
 smult (gcd a (cg div (gcd cf cg))) (gcd f g))
 ⟨proof⟩

lemma *gcd-smult-ex*: **assumes** a ≠ 0
shows ∃ b. gcd (smult a f) g = smult b (gcd f g) ∧ b ≠ 0
 ⟨proof⟩

lemma *primitive-part-idemp*[simp]:
fixes f :: 'a :: {semiring-gcd,normalization-semidom-multiplicative} *poly*
shows primitive-part (primitive-part f) = primitive-part f
 ⟨proof⟩

lemma *content-gcd-primitive*:
 f ≠ 0 ⇒ content (gcd (primitive-part f) g) = 1
 f ≠ 0 ⇒ content (gcd (primitive-part f) (primitive-part g)) = 1
 ⟨proof⟩

lemma *content-gcd-content*: content (gcd f g) = gcd (content f) (content g)
 (is ?l = ?r)
 ⟨proof⟩

lemma *gcd-primitive-part*:
 gcd (primitive-part f) (primitive-part g) = normalize (primitive-part (gcd f g))
 ⟨proof⟩

lemma *primitive-part-gcd*: primitive-part (gcd f g)
 = unit-factor (gcd f g) * gcd (primitive-part f) (primitive-part g)
 ⟨proof⟩

lemma *primitive-part-normalize*:

fixes $f :: 'a :: \{\text{semiring-gcd, idom-divide, normalization-semidom-multiplicative}\}$
poly
shows $\text{primitive-part} (\text{normalize } f) = \text{normalize} (\text{primitive-part } f)$
 $\langle \text{proof} \rangle$

lemma $\text{length-coeffs-primitive-part}[simp]: \text{length} (\text{coeffs} (\text{primitive-part } f)) = \text{length} (\text{coeffs } f)$
 $\langle \text{proof} \rangle$

lemma $\text{degree-unit-factor}[simp]: \text{degree} (\text{unit-factor } f) = 0$
 $\langle \text{proof} \rangle$

lemma $\text{degree-normalize}[simp]: \text{degree} (\text{normalize } f) = \text{degree } f$
 $\langle \text{proof} \rangle$

lemma $\text{content-iff}: x \text{ dvd content } p \longleftrightarrow (\forall c \in \text{set} (\text{coeffs } p). x \text{ dvd } c)$
 $\langle \text{proof} \rangle$

lemma $\text{is-unit-field-poly}[simp]: (p :: 'a :: \text{field poly}) \text{ dvd } 1 \longleftrightarrow p \neq 0 \wedge \text{degree } p = 0$
 $\langle \text{proof} \rangle$

definition *primitive where*
 $\text{primitive } f \longleftrightarrow (\forall x. (\forall y \in \text{set} (\text{coeffs } f). x \text{ dvd } y) \longrightarrow x \text{ dvd } 1)$

lemma *primitiveI:*
assumes $(\bigwedge x. (\bigwedge y. y \in \text{set} (\text{coeffs } f) \implies x \text{ dvd } y) \implies x \text{ dvd } 1)$
shows $\text{primitive } f$ $\langle \text{proof} \rangle$

lemma *primitiveD:*
assumes $\text{primitive } f$
shows $(\bigwedge y. y \in \text{set} (\text{coeffs } f) \implies x \text{ dvd } y) \implies x \text{ dvd } 1$
 $\langle \text{proof} \rangle$

lemma *not-primitiveE:*
assumes $\neg \text{primitive } f$
and $\bigwedge x. (\bigwedge y. y \in \text{set} (\text{coeffs } f) \implies x \text{ dvd } y) \implies \neg x \text{ dvd } 1 \implies \text{thesis}$
shows thesis $\langle \text{proof} \rangle$

lemma $\text{primitive-iff-content-eq-1}[simp]:$
fixes $f :: 'a :: \text{semiring-gcd poly}$
shows $\text{primitive } f \longleftrightarrow \text{content } f = 1$
 $\langle \text{proof} \rangle$

lemma *primitive-prod-list:*
fixes $fs :: 'a :: \{\text{factorial-semiring, semiring-Gcd, normalization-semidom-multiplicative}\}$
poly list
assumes $\text{primitive} (\text{prod-list } fs)$ **and** $f \in \text{set } fs$ **shows** $\text{primitive } f$
 $\langle \text{proof} \rangle$

```

lemma irreducible-imp-primitive:
  fixes  $f :: 'a :: \{idom, semiring-gcd\}$  poly
  assumes irr: irreducible f and deg: degree f  $\neq$  0 shows primitive f
  <proof>

lemma irreducible-primitive-connect:
  fixes  $f :: 'a :: \{idom, semiring-gcd\}$  poly
  assumes cf: primitive f shows irreducibled f  $\longleftrightarrow$  irreducible f (is ?l  $\longleftrightarrow$  ?r)
  <proof>

lemma deg-not-zero-imp-not-unit:
  fixes  $f :: 'a :: \{idom-divide, semidom-divide-unit-factor\}$  poly
  assumes deg-f: degree f > 0
  shows  $\neg$  is-unit f
  <proof>

lemma content-pCons[simp]: content (pCons a p) = gcd a (content p)
  <proof>

lemma content-field-poly:
  fixes  $f :: 'a :: \{field, semiring-gcd\}$  poly
  shows content f = (if f = 0 then 0 else 1)
  <proof>

```

end

6 Gauss Lemma

We formalized Gauss Lemma, that the content of a product of two polynomials p and q is the product of the contents of p and q . As a corollary we provide an algorithm to convert a rational factor of an integer polynomial into an integer factor.

In contrast to the theory on unique factorization domains – where Gauss Lemma is also proven in a more generic setting – we are here in an executable setting and do not use the unspecified *some* – *gcd* function. Moreover, there is a slight difference in the definition of content: in this theory it is only defined for integer-polynomials, whereas in the UFD theory, the content is defined for polynomials in the fraction field.

```

theory Gauss-Lemma
imports
  HOL-Computational-Algebra.Primes
  HOL-Computational-Algebra.Field-as-Ring
  Polynomial-Interpolation.Ring-Hom-Poly
  Missing-Polynomial-Factorial
begin

```

lemma *primitive-part-alt-def*:
primitive-part $p = \text{sdiv-poly } p (\text{content } p)$
 $\langle \text{proof} \rangle$

definition *common-denom* :: *rat list* \Rightarrow *int* \times *int list* **where**
common-denom $xs \equiv \text{let}$
 $nds = \text{map quotient-of } xs;$
 $denom = \text{list-lcm } (\text{map snd } nds);$
 $ints = \text{map } (\lambda (n,d). n * denom \text{ div } d) nds$
 $\text{in } (denom, ints)$

definition *rat-to-int-poly* :: *rat poly* \Rightarrow *int* \times *int poly* **where**
rat-to-int-poly $p \equiv \text{let}$
 $ais = \text{coeffs } p;$
 $d = \text{fst } (\text{common-denom } ais)$
 $\text{in } (d, \text{map-poly } (\lambda x. \text{case quotient-of } x \text{ of } (p,q) \Rightarrow p * d \text{ div } q) p)$

definition *rat-to-normalized-int-poly* :: *rat poly* \Rightarrow *rat* \times *int poly* **where**
rat-to-normalized-int-poly $p \equiv \text{if } p = 0 \text{ then } (1,0) \text{ else case rat-to-int-poly } p \text{ of}$
 (s,q)
 $\Rightarrow (\text{of-int } (\text{content } q) / \text{of-int } s, \text{primitive-part } q)$

lemma *rat-to-normalized-int-poly-code*[code]:
rat-to-normalized-int-poly $p = (\text{if } p = 0 \text{ then } (1,0) \text{ else case rat-to-int-poly } p \text{ of}$
 (s,q)
 $\Rightarrow \text{let } c = \text{content } q \text{ in } (\text{of-int } c / \text{of-int } s, \text{sdiv-poly } q c))$
 $\langle \text{proof} \rangle$

lemma *common-denom: assumes* $cd: \text{common-denom } xs = (dd,ys)$
shows $xs = \text{map } (\lambda i. \text{of-int } i / \text{of-int } dd) ys \text{ dd} > 0$
 $\bigwedge x. x \in \text{set } xs \Longrightarrow \text{rat-of-int } (\text{case quotient-of } x \text{ of } (n, x) \Rightarrow n * dd \text{ div } x) /$
 $\text{rat-of-int } dd = x$
 $\langle \text{proof} \rangle$

lemma *rat-to-int-poly: assumes* $\text{rat-to-int-poly } p = (d,q)$
shows $p = \text{smult } (\text{inverse } (\text{of-int } d)) (\text{map-poly } \text{of-int } q) \text{ d} > 0$
 $\langle \text{proof} \rangle$

lemma *content-ge-0-int: content* $p \geq (0 :: \text{int})$
 $\langle \text{proof} \rangle$

lemma *abs-content-int*[simp]: **fixes** $p :: \text{int poly}$
shows $\text{abs } (\text{content } p) = \text{content } p \langle \text{proof} \rangle$

lemma *content-smult-int: fixes* $p :: \text{int poly}$
shows $\text{content } (\text{smult } a p) = \text{abs } a * \text{content } p \langle \text{proof} \rangle$

lemma *normalize-non-0-smult: $\exists a. (a :: 'a :: \text{semiring-gcd}) \neq 0 \wedge \text{smult } a$*
 $(\text{primitive-part } p) = p$

<proof>

lemma *rat-to-normalized-int-poly*: **assumes** *rat-to-normalized-int-poly* $p = (d, q)$
shows $p = \text{smult } d \ (\text{map-poly of-int } q) \ d > 0 \ p \neq 0 \implies \text{content } q = 1 \ \text{degree } q = \text{degree } p$
<proof>

lemma *content-dvd-1*:
content $g = 1$ **if** **content** $f = (1 :: 'a :: \text{semiring-gcd}) \ g \ \text{dvd } f$
<proof>

lemma *dvd-smult-int*: **fixes** $c :: \text{int}$ **assumes** $c: c \neq 0$
and *dvd*: $q \ \text{dvd} \ (\text{smult } c \ p)$
shows *primitive-part* $q \ \text{dvd} \ p$
<proof>

lemma *irreducible_a-primitive-part*:
fixes $p :: \text{int poly}$
shows *irreducible_a* (*primitive-part* p) \longleftrightarrow *irreducible_a* p (**is** $?l \longleftrightarrow ?r$)
<proof>

lemma *irreducible_a-smult-int*:
fixes $c :: \text{int}$ **assumes** $c: c \neq 0$
shows *irreducible_a* (*smult* $c \ p$) = *irreducible_a* p (**is** $?l = ?r$)
<proof>

lemma *irreducible_a-as-irreducible*:
fixes $p :: \text{int poly}$
shows *irreducible_a* $p \longleftrightarrow$ *irreducible* (*primitive-part* p)
<proof>

lemma *rat-to-int-factor-content-1*: **fixes** $p :: \text{int poly}$
assumes *cp*: **content** $p = 1$
and *pgh*: *map-poly rat-of-int* $p = g * h$
and *g*: *rat-to-normalized-int-poly* $g = (r, rg)$
and *h*: *rat-to-normalized-int-poly* $h = (s, sh)$
and *p*: $p \neq 0$
shows $p = rg * sh$
<proof>

lemma *rat-to-int-factor-explicit*: **fixes** $p :: \text{int poly}$
assumes *pgh*: *map-poly rat-of-int* $p = g * h$
and *g*: *rat-to-normalized-int-poly* $g = (r, rg)$
shows $\exists r. p = rg * \text{smult} \ (\text{content } p) \ r$
<proof>

lemma *rat-to-int-factor*: **fixes** $p :: \text{int poly}$
assumes *pgh*: *map-poly rat-of-int* $p = g * h$

shows $\exists g' h'. p = g' * h' \wedge \text{degree } g' = \text{degree } g \wedge \text{degree } h' = \text{degree } h$
 ⟨proof⟩

lemma *rat-to-int-factor-normalized-int-poly*: **fixes** $p :: \text{rat poly}$
assumes $pg h: p = g * h$
and $p: \text{rat-to-normalized-int-poly } p = (i, ip)$
shows $\exists g' h'. ip = g' * h' \wedge \text{degree } g' = \text{degree } g$
 ⟨proof⟩

lemma *irreducible-smult* [*simp*]:
fixes $c :: 'a :: \text{field}$
shows $\text{irreducible } (\text{smult } c \ p) \longleftrightarrow \text{irreducible } p \wedge c \neq 0$
 ⟨proof⟩

A polynomial with integer coefficients is irreducible over the rationals, if it is irreducible over the integers.

theorem *irreducible_d-int-rat*: **fixes** $p :: \text{int poly}$
assumes $p: \text{irreducible}_d \ p$
shows $\text{irreducible}_d (\text{map-poly } \text{rat-of-int } p)$
 ⟨proof⟩

corollary *irreducible_d-rat-to-normalized-int-poly*:
assumes $rp: \text{rat-to-normalized-int-poly } rp = (a, ip)$
and $ip: \text{irreducible}_d \ ip$
shows $\text{irreducible}_d \ rp$
 ⟨proof⟩

lemma *dvd-content-dvd*: **assumes** $dvd: \text{content } f \ \text{dvd } \text{content } g \ \text{primitive-part } f \ \text{dvd } \text{primitive-part } g$
shows $f \ \text{dvd } g$
 ⟨proof⟩

lemma *sdiv-poly-smult*: $c \neq 0 \implies \text{sdiv-poly } (\text{smult } c \ f) \ c = f$
 ⟨proof⟩

lemma *primitive-part-smult-int*: **fixes** $f :: \text{int poly}$ **shows**
 $\text{primitive-part } (\text{smult } d \ f) = \text{smult } (\text{sgn } d) (\text{primitive-part } f)$
 ⟨proof⟩

lemma *gcd-smult-left*: **assumes** $c \neq 0$
shows $\text{gcd } (\text{smult } c \ f) \ g = \text{gcd } f \ (g :: 'b :: \{\text{field-gcd}\} \ \text{poly})$
 ⟨proof⟩

lemma *gcd-smult-right*: $c \neq 0 \implies \text{gcd } f \ (\text{smult } c \ g) = \text{gcd } f \ (g :: 'b :: \{\text{field-gcd}\} \ \text{poly})$
 ⟨proof⟩

lemma *gcd-rat-to-gcd-int*: $\text{gcd } (\text{of-int-poly } f :: \text{rat poly}) \ (\text{of-int-poly } g) =$

smult (inverse (of-int (lead-coeff (gcd f g))) (of-int-poly (gcd f g)))
 ⟨proof⟩

end

7 Prime Factorization

This theory contains not-completely naive algorithms to test primality and to perform prime factorization. More precisely, it corresponds to prime factorization algorithm A in Knuth's textbook [1].

theory *Prime-Factorization*

imports

HOL-Computational-Algebra.Primes

Missing-List

Missing-Multiset

begin

7.1 Definitions

definition *primes-1000* :: *nat list* **where**

primes-1000 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997]

lemma *primes-1000*: *primes-1000* = *filter prime [0..<1001]*

⟨proof⟩

definition *next-candidates* :: *nat* ⇒ *nat* × *nat list* **where**

next-candidates *n* = (if *n* = 0 then (1001, *primes-1000*) else (*n* + 30, [*n*, *n* + 2, *n* + 6, *n* + 8, *n* + 12, *n* + 18, *n* + 20, *n* + 26]))

definition *candidate-invariant* *n* = (*n* = 0 ∨ *n mod* 30 = (11 :: *nat*))

partial-function (*tailrec*) *remove-prime-factor* :: *nat* ⇒ *nat* ⇒ *nat list* ⇒ *nat* × *nat list* **where**

[code]: *remove-prime-factor* *p n ps* = (case *Divides.divmod-nat* *n p* of (*n',m*) ⇒ if *m = 0* then *remove-prime-factor* *p n' (p # ps)* else (*n,ps*))

partial-function (*tailrec*) *prime-factorization-nat-main*

:: *nat* ⇒ *nat* ⇒ *nat list* ⇒ *nat list* ⇒ *nat list* **where**

[code]: *prime-factorization-nat-main* *n j is ps* = (case *is* of

[] ⇒

(case *next-candidates* *j* of (*j,is*) ⇒ *prime-factorization-nat-main* *n j is ps*)

| (*i # is*) ⇒ (case *Divides.divmod-nat* *n i* of (*n',m*) ⇒

if *m = 0* then case *remove-prime-factor* *i n' (i # ps)*

of (*n',ps'*) ⇒ if *n' = 1* then *ps'* else

prime-factorization-nat-main *n' j is ps'*

else if *i * i ≤ n* then *prime-factorization-nat-main* *n j is ps*

else (*n # ps*))

partial-function (*tailrec*) *prime-nat-main*

:: *nat* ⇒ *nat* ⇒ *nat list* ⇒ *bool* **where**

[code]: *prime-nat-main* *n j is* = (case *is* of

[] ⇒ (case *next-candidates* *j* of (*j,is*) ⇒ *prime-nat-main* *n j is*)

| (*i # is*) ⇒ (if *i dvd n* then *i ≥ n* else if *i * i ≤ n* then *prime-nat-main* *n j is* else *True*))

definition *prime-nat* :: *nat* ⇒ *bool* **where**

prime-nat *n* ≡ if *n < 2* then *False* else — TODO: integrate precomputed map
case *next-candidates* 0 of (*j,is*) ⇒ *prime-nat-main* *n j is*

definition *prime-factorization-nat* :: *nat* ⇒ *nat list* **where**

prime-factorization-nat *n* ≡ rev (if *n < 2* then [] else

case *next-candidates* 0 of (*j,is*) ⇒ *prime-factorization-nat-main* *n j is* [])

definition *divisors-nat* :: *nat* ⇒ *nat list* **where**

divisors-nat *n* ≡ if *n = 0* then [] else

remdups-adj (sort (map *prod-list* (subseqs (*prime-factorization-nat* *n*))))

definition *divisors-int-pos* :: *int* ⇒ *int list* **where**

divisors-int-pos *x* ≡ map *int* (*divisors-nat* (*nat* (*abs* *x*)))

definition *divisors-int* :: *int* ⇒ *int list* **where**

divisors-int *x* ≡ let *xs* = *divisors-int-pos* *x* in *xs* @ (map *uminus* *xs*)

7.2 Proofs

lemma *remove-prime-factor*: **assumes** *res*: *remove-prime-factor* *i n ps* = (*m,qs*)

and *i*: *i > 1*

and *n*: *n ≠ 0*

shows ∃ *rs*. *qs* = *rs* @ *ps* ∧ *n* = *m* * *prod-list* *rs* ∧ ¬ *i dvd m* ∧ set *rs* ⊆ {*i*}

⟨*proof*⟩

lemma prime-sqrtI: **assumes** $n: n \geq 2$
and small: $\bigwedge j. 2 \leq j \implies j < i \implies \neg j \text{ dvd } n$
and i: $\neg i * i \leq n$
shows $\text{prime } (n::\text{nat})$ *<proof>*

lemma candidate-invariant-0: *candidate-invariant 0*
<proof>

lemma next-candidates: **assumes** $\text{res}: \text{next-candidates } n = (m, ps)$
and n: *candidate-invariant n*
shows $\text{candidate-invariant } m \text{ sorted } ps \{i. \text{prime } i \wedge n \leq i \wedge i < m\} \subseteq \text{set } ps$
 $\text{set } ps \subseteq \{2..\} \cap \{n..<m\}$ *distinct ps ps $\neq []$ $n < m$*
<proof>

lemma prime-test-iterate2: **assumes** $\text{small}: \bigwedge j. 2 \leq j \implies j < (i :: \text{nat}) \implies \neg j \text{ dvd } n$
and odd: *odd n*
and n: $n \geq 3$
and i: $i \geq 3$ *odd i*
and mod: $\neg i \text{ dvd } n$
and j: $2 \leq j < i + 2$
shows $\neg j \text{ dvd } n$
<proof>

lemma prime-divisor: **assumes** $j \geq 2$ **and** $j \text{ dvd } n$ **shows**
 $\exists p :: \text{nat. prime } p \wedge p \text{ dvd } j \wedge p \text{ dvd } n$
<proof>

lemma prime-nat-main: $ni = (n, i, is) \implies i \geq 2 \implies n \geq 2 \implies$
 $(\bigwedge j. 2 \leq j \implies j < i \implies \neg (j \text{ dvd } n)) \implies$
 $(\bigwedge j. i \leq j \implies j < jj \implies \text{prime } j \implies j \in \text{set } is) \implies i \leq jj \implies$
 $\text{sorted } is \implies \text{distinct } is \implies \text{candidate-invariant } jj \implies \text{set } is \subseteq \{i..<jj\} \implies$
 $\text{res} = \text{prime-nat-main } n \text{ } jj \text{ } is \implies$
 $\text{res} = \text{prime } n$
<proof>

lemma prime-factorization-nat-main: $ni = (n, i, is) \implies i \geq 2 \implies n \geq 2 \implies$
 $(\bigwedge j. 2 \leq j \implies j < i \implies \neg (j \text{ dvd } n)) \implies$
 $(\bigwedge j. i \leq j \implies j < jj \implies \text{prime } j \implies j \in \text{set } is) \implies i \leq jj \implies$
 $\text{sorted } is \implies \text{distinct } is \implies \text{candidate-invariant } jj \implies \text{set } is \subseteq \{i..<jj\} \implies$
 $\text{res} = \text{prime-factorization-nat-main } n \text{ } jj \text{ } is \text{ } ps \implies$
 $\exists qs. \text{res} = qs @ ps \wedge \text{Ball } (\text{set } qs) \text{ prime} \wedge n = \text{prod-list } qs$
<proof>

lemma prime-nat[simp]: $\text{prime-nat } n = \text{prime } n$
<proof>

lemma *prime-factorization-nat*: **fixes** $n :: \text{nat}$

defines $pf \equiv \text{prime-factorization-nat } n$

shows $\text{Ball } (\text{set } pf) \text{ prime}$

and $n \neq 0 \implies \text{prod-list } pf = n$

and $n = 0 \implies pf = []$

$\langle \text{proof} \rangle$

lemma *prod-mset-multiset-prime-factorization-nat* [*simp*]:

$(x :: \text{nat}) \neq 0 \implies \text{prod-mset } (\text{prime-factorization } x) = x$

$\langle \text{proof} \rangle$

lemma *prime-factorization-unique''*:

fixes $A :: 'a :: \{\text{factorial-semiring-multiplicative}\} \text{ multiset}$

assumes $\bigwedge p. p \in \# A \implies \text{prime } p$

assumes $\text{prod-mset } A = \text{normalize } x$

shows $\text{prime-factorization } x = A$

$\langle \text{proof} \rangle$

lemma *multiset-prime-factorization-nat-correct*:

$\text{prime-factorization } n = \text{mset } (\text{prime-factorization-nat } n)$

$\langle \text{proof} \rangle$

lemma *multiset-prime-factorization-code*[*code-unfold*]:

$\text{prime-factorization} = (\lambda n. \text{mset } (\text{prime-factorization-nat } n))$

$\langle \text{proof} \rangle$

lemma *divisors-nat*:

$n \neq 0 \implies \text{set } (\text{divisors-nat } n) = \{p. p \text{ dvd } n\} \text{ distinct } (\text{divisors-nat } n) \text{ divisors-nat } 0 = []$

$\langle \text{proof} \rangle$

lemma *divisors-int-pos*: $x \neq 0 \implies \text{set } (\text{divisors-int-pos } x) = \{i. i \text{ dvd } x \wedge i > 0\} \text{ distinct } (\text{divisors-int-pos } x)$

$\text{divisors-int-pos } 0 = []$

$\langle \text{proof} \rangle$

lemma *divisors-int*: $x \neq 0 \implies \text{set } (\text{divisors-int } x) = \{i. i \text{ dvd } x\} \text{ distinct } (\text{divisors-int } x)$

$\text{divisors-int } 0 = []$

$\langle \text{proof} \rangle$

definition *divisors-fun* :: $('a \Rightarrow ('a :: \{\text{comm-monoid-mult,zero}\}) \text{ list}) \Rightarrow \text{bool}$

where

$\text{divisors-fun } df \equiv (\forall x. x \neq 0 \longrightarrow \text{set } (df x) = \{d. d \text{ dvd } x\}) \wedge (\forall x. \text{distinct } (df x))$

lemma *divisors-funD*: $\text{divisors-fun } df \implies x \neq 0 \implies d \text{ dvd } x \implies d \in \text{set } (df x)$

$\langle \text{proof} \rangle$

definition *divisors-pos-fun* :: ('a ⇒ ('a :: {comm-monoid-mult,zero,ord}) list) ⇒ bool **where**

divisors-pos-fun df ≡ (∀ x. x ≠ 0 → set (df x) = { d. d dvd x ∧ d > 0}) ∧ (∀ x. distinct (df x))

lemma *divisors-pos-funD*: *divisors-pos-fun* df ⇒ x ≠ 0 ⇒ d dvd x ⇒ d > 0 ⇒ d ∈ set (df x)
 ⟨proof⟩

lemma *divisors-fun-nat*: *divisors-fun* *divisors-nat*
 ⟨proof⟩

lemma *divisors-fun-int*: *divisors-fun* *divisors-int*
 ⟨proof⟩

lemma *divisors-pos-fun-int*: *divisors-pos-fun* *divisors-int-pos*
 ⟨proof⟩

end

8 Rational Root Test

This theory contains a formalization of the rational root test, i.e., a decision procedure to test whether a polynomial over the rational numbers has a rational root.

theory *Rational-Root-Test*

imports

Gauss-Lemma

Missing-List

Prime-Factorization

begin

definition *rational-root-test-main* ::

(int ⇒ int list) ⇒ (int ⇒ int list) ⇒ rat poly ⇒ rat option **where**
rational-root-test-main df dp p ≡ let ip = snd (rat-to-normalized-int-poly p);
 a0 = coeff ip 0; an = coeff ip (degree ip)
 in if a0 = 0 then Some 0 else
 let d0 = df a0; dn = dp an
 in map-option fst
 (find-map-filter (λ x. (x,poly p x))
 (λ (-, res). res = 0) [rat-of-int b0 / of-int bn . b0 <- d0, bn <- dn, coprime
 b0 bn])

definition *rational-root-test* :: rat poly ⇒ rat option **where**

rational-root-test p =

rational-root-test-main *divisors-int* *divisors-int-pos* p

lemma *rational-root-test-main*:

rational-root-test-main *df dp p = Some x* \implies *poly p x = 0*
divisors-fun df \implies *divisors-pos-fun dp* \implies *rational-root-test-main df dp p =*
None \implies $\neg (\exists x. \text{poly } p \ x = 0)$
{*proof*}

lemma *rational-root-test*:

rational-root-test p = Some x \implies *poly p x = 0*
rational-root-test p = None \implies $\neg (\exists x. \text{poly } p \ x = 0)$
{*proof*}

end

9 Kronecker Factorization

This theory contains Kronecker's factorization algorithm to factor integer or rational polynomials.

theory *Kronecker-Factorization*

imports

Polynomial-Interpolation.Polynomial-Interpolation
Sqrt-Babylonian.Sqrt-Babylonian-Auxiliary
Missing-List
Prime-Factorization
Precomputation
Gauss-Lemma
Dvd-Int-Poly

begin

9.1 Definitions

context

fixes *df* :: *int* \Rightarrow *int list*
and *dp* :: *int* \Rightarrow *int list*
and *bnd* :: *nat*

begin

definition *kronecker-samples* :: *nat* \Rightarrow *int list* **where**

kronecker-samples n \equiv *let min = - int (n div 2) in [min .. min + int n]*

lemma *kronecker-samples-0*: $0 \in \text{set } (\text{kronecker-samples } n)$ {*proof*}

Since 0 is always a samples value, we make a case analysis: we only take positive divisors of $p(0)$, and consider all divisors for other $p(j)$.

definition *kronecker-factorization-main* :: *int poly* \Rightarrow *int poly option* **where**

kronecker-factorization-main p \equiv *if degree p \leq 1 then None else let*
p = primitive-part p;
js = kronecker-samples bnd;

```

    cjs = map (λ j. (poly p j, j)) js
  in (case map-of cjs 0 of
    Some j ⇒ Some ([: - j, 1 :])
  | None ⇒ let djs = map (λ (v,j). map (Pair j) (if j = 0 then dp v else df v)) cjs
  in
    map-option the (find-map-filter newton-interpolation-poly-int
      (λ go. case go of None ⇒ False | Some g ⇒ dvd-int-poly-non-0 g p ∧ degree g
        ≥ 1)
      (concat-lists djs)))

```

definition *kronecker-factorization-rat-main* :: *rat poly* ⇒ *rat poly option* **where**
kronecker-factorization-rat-main p ≡ *map-option* (*map-poly of-int*)
(*kronecker-factorization-main* (*snd* (*rat-to-normalized-int-poly* p)))
end

definition *kronecker-factorization* :: *int poly* ⇒ *int poly option* **where**
kronecker-factorization p =
kronecker-factorization-main *divisors-int* *divisors-int-pos* (*degree* p *div* 2) p

definition *kronecker-factorization-rat* :: *rat poly* ⇒ *rat poly option* **where**
kronecker-factorization-rat p =
kronecker-factorization-rat-main *divisors-int* *divisors-int-pos* (*degree* p *div* 2) p

9.2 Code setup for divisors

definition *divisors-nat-copy* n ≡ if n = 0 then [] else *remdups-adj* (*sort* (*map*
prod-list (*subseqs* (*prime-factorization-nat* n))))

lemma *divisors-nat-copy[simp]*: *divisors-nat-copy* = *divisors-nat*
⟨*proof*⟩

definition *memo-divisors-nat* ≡ *memo-nat* 0 100 *divisors-nat-copy*

lemma *memo-divisors-nat[code-unfold]*: *divisors-nat* = *memo-divisors-nat*
⟨*proof*⟩

9.3 Proofs

context
begin

lemma *rat-to-int-poly-of-int*: **assumes** *rp*: *rat-to-int-poly* (*map-poly of-int* p) =
(*c*, *q*)
shows *c* = 1 *q* = *p*
⟨*proof*⟩

lemma *rat-to-normalized-int-poly-of-int*: **assumes** *rat-to-normalized-int-poly* (*map-poly*
of-int p) = (*c*, *q*)
shows *c* ∈ \mathbb{Z} *p* ≠ 0 ⇒ *c* = *of-int* (*content* p) ∧ *q* = *primitive-part* p
⟨*proof*⟩

lemma *dvd-poly-int-content-1*: **assumes** *c-x: content x = 1*
shows $(x \text{ dvd } y) = (\text{map-poly rat-of-int } x \text{ dvd map-poly of-int } y)$
 $\langle \text{proof} \rangle$

lemma *content-x-minus-const-int[simp]*: $\text{content } [: c, 1 :] = (1 :: \text{int})$
 $\langle \text{proof} \rangle$

lemma *length-upto-add-nat[simp]*: $\text{length } [a .. a + \text{int } n] = \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *kronecker-samples*: $\text{distinct } (\text{kronecker-samples } n) \text{ length } (\text{kronecker-samples } n) = \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *dvd-int-poly-non-0-degree-1[simp]*: $\text{degree } q \geq 1 \implies \text{dvd-int-poly-non-0 } q$
 $p = (q \text{ dvd } p)$
 $\langle \text{proof} \rangle$

context **fixes** *df dp :: int \Rightarrow int list*
and *bnd :: nat*
begin

lemma *kronecker-factorization-main-sound*: **assumes** *some: kronecker-factorization-main*
df dp bnd p = Some q
and *bnd: degree p $\geq 2 \implies$ bnd ≥ 1*
shows $\text{degree } q \geq 1 \text{ degree } q \leq \text{bnd } q \text{ dvd } p$
 $\langle \text{proof} \rangle$

lemma *kronecker-factorization-rat-main-sound*: **assumes**
some: kronecker-factorization-rat-main df dp bnd p = Some q
and *bnd: degree p $\geq 2 \implies$ bnd ≥ 1*
shows $\text{degree } q \geq 1 \text{ degree } q \leq \text{bnd } q \text{ dvd } p$
 $\langle \text{proof} \rangle$

context
assumes *df: divisors-fun df and dpf: divisors-pos-fun dp*
begin

lemma *kronecker-factorization-main-complete*: **assumes**
none: kronecker-factorization-main df dp bnd p = None
and *dp: degree p ≥ 2*
shows $\neg (\exists q. 1 \leq \text{degree } q \wedge \text{degree } q \leq \text{bnd} \wedge q \text{ dvd } p)$
 $\langle \text{proof} \rangle$

lemma *kronecker-factorization-rat-main-complete*: **assumes**

```

none: kronecker-factorization-rat-main df dp bnd p = None
and dp: degree p ≥ 2
shows ¬ (∃ q. 1 ≤ degree q ∧ degree q ≤ bnd ∧ q dvd p)
⟨proof⟩
end
end

```

```

lemma kronecker-factorization:
kronecker-factorization p = Some q ⇒
degree q ≥ 1 ∧ degree q < degree p ∧ q dvd p
kronecker-factorization p = None ⇒ degree p ≥ 1 ⇒ irreducibled p
⟨proof⟩

```

```

lemma kronecker-factorization-rat:
kronecker-factorization-rat p = Some q ⇒
degree q ≥ 1 ∧ degree q < degree p ∧ q dvd p
kronecker-factorization-rat p = None ⇒ degree p ≥ 1 ⇒ irreducibled p
⟨proof⟩

```

```

end
end

```

10 Polynomial Divisibility

We make a connection between irreducibility of Missing-Polynomial and Factorial-Ring.

```

theory Polynomial-Divisibility
imports
Polynomial-Interpolation.Missing-Polynomial
begin

```

```

lemma dvd-gcd-mult: fixes p :: 'a :: semiring-gcd
assumes dvd: k dvd p * q k dvd p * r
shows k dvd p * gcd q r
⟨proof⟩

```

```

lemma poly-gcd-monic-factor:
monic p ⇒ gcd (p * q) (p * r) = p * gcd q r
⟨proof⟩

```

```

context
assumes SORT-CONSTRAINT('a :: field)
begin

```

```

lemma field-poly-irreducible-dvd-mult[simp]:
assumes irr: irreducible (p :: 'a poly)
shows p dvd q * r ⟷ p dvd q ∨ p dvd r
⟨proof⟩

```


lemma *irreducible-dvd-pow*:
fixes $p :: 'a \text{ poly}$
assumes $irr: \text{irreducible } p$
shows $p \text{ dvd } q \wedge n \implies p \text{ dvd } q$
 $\langle \text{proof} \rangle$

lemma *irreducible-dvd-prod*: **fixes** $p :: 'a \text{ poly}$
assumes $irr: \text{irreducible } p$
and $dvd: p \text{ dvd } \text{prod } f \text{ as}$
shows $\exists a \in \text{as. } p \text{ dvd } f a$
 $\langle \text{proof} \rangle$

lemma *irreducible-dvd-prod-list*: **fixes** $p :: 'a \text{ poly}$
assumes $irr: \text{irreducible } p$
and $dvd: p \text{ dvd } \text{prod-list } \text{as}$
shows $\exists a \in \text{set as. } p \text{ dvd } a$
 $\langle \text{proof} \rangle$

lemma *dvd-mult-imp-degree*: **fixes** $p :: 'a \text{ poly}$
assumes $p \text{ dvd } q * r$
and $\text{degree } p > 0$
shows $\exists s t. \text{irreducible } s \wedge p = s * t \wedge (s \text{ dvd } q \vee s \text{ dvd } r)$
 $\langle \text{proof} \rangle$

end

end

10.1 Fundamental Theorem of Algebra for Factorizations

Via the existing formulation of the fundamental theorem of algebra, we prove that we always get a linear factorization of a complex polynomial. Using this factorization we show that root-square-freeness of complex polynomial is identical to the statement that the cardinality of the set of all roots is equal to the degree of the polynomial.

theory *Fundamental-Theorem-Algebra-Factorized*
imports
Order-Polynomial
HOL-Computational-Algebra.Fundamental-Theorem-Algebra
begin

lemma *fundamental-theorem-algebra-factorized*: **fixes** $p :: \text{complex poly}$
shows $\exists \text{as. } \text{smult } (\text{coeff } p \text{ (degree } p)) \left(\prod a \leftarrow \text{as. } [:- a, 1:] \right) = p \wedge \text{length as} = \text{degree } p$
 $\langle \text{proof} \rangle$

lemma *rsquarefree-card-degree*: **assumes** $p0$: $(p :: \text{complex poly}) \neq 0$
shows $\text{rsquarefree } p = (\text{card } \{x. \text{poly } p \ x = 0\} = \text{degree } p)$
 $\langle \text{proof} \rangle$

end

11 Square Free Factorization

We implemented Yun's algorithm to perform a square-free factorization of a polynomial. We further show properties of a square-free factorization, namely that the exponents in the square-free factorization are exactly the orders of the roots. We also show that factorizing the result of square-free factorization further will again result in a square-free factorization, and that square-free factorizations can be lifted homomorphically.

theory *Square-Free-Factorization*

imports

Matrix.Utility

Polynomial-Divisibility

Order-Polynomial

Fundamental-Theorem-Algebra-Factorized

Polynomial-Interpolation.Ring-Hom-Poly

begin

definition *square-free* :: $'a :: \text{comm-semiring-1 poly} \Rightarrow \text{bool}$ **where**
 $\text{square-free } p = (p \neq 0 \wedge (\forall q. \text{degree } q > 0 \longrightarrow \neg (q * q \text{ dvd } p)))$

lemma *square-freeI*:

assumes $\bigwedge q. \text{degree } q > 0 \Longrightarrow q \neq 0 \Longrightarrow q * q \text{ dvd } p \Longrightarrow \text{False}$

and $p: p \neq 0$

shows $\text{square-free } p$ $\langle \text{proof} \rangle$

lemma *square-free-multD*:

assumes $\text{sf}: \text{square-free } (f * g)$

shows $h \text{ dvd } f \Longrightarrow h \text{ dvd } g \Longrightarrow \text{degree } h = 0 \text{ square-free } f \text{ square-free } g$
 $\langle \text{proof} \rangle$

lemma *irreducible_d-square-free*:

fixes $p :: 'a :: \{\text{comm-semiring-1}, \text{semiring-no-zero-divisors}\} \text{ poly}$

shows $\text{irreducible}_d \ p \Longrightarrow \text{square-free } p$

$\langle \text{proof} \rangle$

lemma *square-free-factor*: **assumes** $\text{dvd}: a \text{ dvd } p$

and $\text{sf}: \text{square-free } p$

shows $\text{square-free } a$

$\langle \text{proof} \rangle$

lemma *square-free-prod-list-distinct*:

assumes *sf*: square-free (prod-list us :: 'a :: idom poly)
and *us*: $\bigwedge u. u \in \text{set } us \implies \text{degree } u > 0$
shows *distinct us*
 <proof>

definition *separable where*
separable f = coprime f (pderiv f)

lemma *separable-imp-square-free*:
assumes *sep*: separable (f :: 'a:: {field, factorial-ring-gcd, semiring-gcd-mult-normalize}
 poly)
shows square-free f
 <proof>

lemma *square-free-rsquarefree*: **assumes** *f*: square-free f
shows rsquarefree f
 <proof>

lemma *square-free-prodD*:
fixes *fs* :: 'a :: {field, euclidean-ring-gcd, semiring-gcd-mult-normalize} poly set
assumes *sf*: square-free ($\prod fs$)
and *fin*: finite fs
and *f*: $f \in fs$
and *g*: $g \in fs$
and *fg*: $f \neq g$
shows coprime f g
 <proof>

lemma *rsquarefree-square-free-complex*: **assumes** *rsquarefree* (p :: complex poly)
shows square-free p
 <proof>

lemma *square-free-separable-main*:
fixes *f* :: 'a :: {field, factorial-ring-gcd, semiring-gcd-mult-normalize} poly
assumes square-free f
and *sep*: $\neg \text{separable } f$
shows $\exists g k. f = g * k \wedge \text{degree } g \neq 0 \wedge \text{pderiv } g = 0$
 <proof>

lemma *square-free-imp-separable*: **fixes** *f* :: 'a :: {field-char-0, factorial-ring-gcd, semiring-gcd-mult-normalize}
 poly
assumes square-free f
shows separable f
 <proof>

lemma *square-free-iff-separable*:
 square-free (f :: 'a :: {field-char-0, factorial-ring-gcd, semiring-gcd-mult-normalize}
 poly) = separable f

<proof>

context

assumes *SORT-CONSTRAINT*('a::{field,factorial-ring-gcd})

begin

lemma *square-free-smult*: $c \neq 0 \implies \text{square-free } (f :: 'a \text{ poly}) \implies \text{square-free } (\text{smult } c f)$

<proof>

lemma *square-free-smult-iff[simp]*: $c \neq 0 \implies \text{square-free } (\text{smult } c f) = \text{square-free } (f :: 'a \text{ poly})$

<proof>

end

context

assumes *SORT-CONSTRAINT*('a::factorial-ring-gcd)

begin

definition *square-free-factorization* :: $'a \text{ poly} \Rightarrow 'a \times ('a \text{ poly} \times \text{nat}) \text{ list} \Rightarrow \text{bool}$

where

square-free-factorization $p \text{ cbs} \equiv \text{case cbs of } (c, \text{bs}) \Rightarrow$

$(p = \text{smult } c (\prod_{(a, i) \in \text{set bs. } a \wedge \text{Suc } i})$

$\wedge (p = 0 \longrightarrow c = 0 \wedge \text{bs} = [])$

$\wedge (\forall a \ i. (a, i) \in \text{set bs} \longrightarrow \text{square-free } a \wedge \text{degree } a > 0)$

$\wedge (\forall a \ i \ b \ j. (a, i) \in \text{set bs} \longrightarrow (b, j) \in \text{set bs} \longrightarrow (a, i) \neq (b, j) \longrightarrow \text{coprime } a \ b)$

$\wedge \text{distinct } \text{bs}$

lemma *square-free-factorizationD*: **assumes** *square-free-factorization* $p \ (c, \text{bs})$

shows $p = \text{smult } c (\prod_{(a, i) \in \text{set bs. } a \wedge \text{Suc } i})$

$(a, i) \in \text{set bs} \implies \text{square-free } a \wedge \text{degree } a \neq 0$

$(a, i) \in \text{set bs} \implies (b, j) \in \text{set bs} \implies (a, i) \neq (b, j) \implies \text{coprime } a \ b$

$p = 0 \implies c = 0 \wedge \text{bs} = []$

distinct bs

<proof>

lemma *square-free-factorization-prod-list*: **assumes** *square-free-factorization* $p \ (c, \text{bs})$

shows $p = \text{smult } c (\text{prod-list } (\text{map } (\lambda (a, i). a \wedge \text{Suc } i) \text{bs}))$

<proof>

end

11.1 Yun's factorization algorithm

locale *yun-gcd* =

fixes $Gcd :: 'a :: \text{factorial-ring-gcd poly} \Rightarrow 'a \text{ poly} \Rightarrow 'a \text{ poly}$

begin

partial-function (*tailrec*) *yun-factorization-main* ::

$'a \text{ poly} \Rightarrow 'a \text{ poly} \Rightarrow$

$\text{nat} \Rightarrow ('a \text{ poly} \times \text{nat}) \text{list} \Rightarrow ('a \text{ poly} \times \text{nat}) \text{list}$ **where**

[code]: *yun-factorization-main* $bn \ cn \ i \ \text{sqr} = ($

```

    if bn = 1 then sqr
    else (
    let
      dn = cn - pderiv bn;
      an = Gcd bn dn
    in yun-factorization-main (bn div an) (dn div an) (Suc i) ((an,i) # sqr)))

```

definition *yun-monic-factorization* :: 'a poly \Rightarrow ('a poly \times nat)list **where**

```

yun-monic-factorization p = (let
  pp = pderiv p;
  u = Gcd p pp;
  b0 = p div u;
  c0 = pp div u
in
  (filter ( $\lambda$  (a,i). a  $\neq$  1) (yun-factorization-main b0 c0 0 [])))

```

definition *square-free-monic-poly* :: 'a poly \Rightarrow 'a poly **where**

```

square-free-monic-poly p = (p div (Gcd p (pderiv p)))
end

```

```

declare yun-gcd.yun-monic-factorization-def [code]
declare yun-gcd.yun-factorization-main.simps [code]
declare yun-gcd.square-free-monic-poly-def [code]

```

context

```

  fixes Gcd :: 'a :: {field-char-0,euclidean-ring-gcd} poly  $\Rightarrow$  'a poly  $\Rightarrow$  'a poly
begin
interpretation yun-gcd Gcd <proof>

```

definition *square-free-poly* :: 'a poly \Rightarrow 'a poly **where**

```

square-free-poly p = (if p = 0 then 0 else
  square-free-monic-poly (smult (inverse (coeff p (degree p))) p))

```

definition *yun-factorization* :: 'a poly \Rightarrow 'a \times ('a poly \times nat)list **where**

```

yun-factorization p = (if p = 0
  then (0,[]) else (let
  c = coeff p (degree p);
  q = smult (inverse c) p
in (c, yun-monic-factorization q)))

```

lemma *yun-factorization-0*[simp]: *yun-factorization* 0 = (0,[])
<proof>

end

locale *monic-factorization* =

```

  fixes as :: ('a :: {field-char-0,euclidean-ring-gcd,semiring-gcd-mult-normalize}
poly  $\times$  nat) set
  and p :: 'a poly

```

assumes p : $p = \text{prod } (\lambda (a,i). a \wedge \text{Suc } i) \text{ as}$
and fin : finite as
assumes as-distinct : $\bigwedge a i b j. (a,i) \in \text{as} \implies (b,j) \in \text{as} \implies (a,i) \neq (b,j) \implies a \neq b$
and as-irred : $\bigwedge a i. (a,i) \in \text{as} \implies \text{irreducible}_d a$
and as-monic : $\bigwedge a i. (a,i) \in \text{as} \implies \text{monic } a$
begin

lemma poly-exp-expand :
 $p = (\text{prod } (\lambda (a,i). a \wedge i) \text{ as}) * \text{prod } (\lambda (a,i). a) \text{ as}$
 $\langle \text{proof} \rangle$

lemma pderiv-exp-prod :
 $\text{pderiv } p = (\text{prod } (\lambda (a,i). a \wedge i) \text{ as} * \text{sum } (\lambda (a,i). \text{prod } (\lambda (b,j). b) (\text{as} - \{(a,i)\}) * \text{smult } (\text{of-nat } (\text{Suc } i)) (\text{pderiv } a)) \text{ as})$
 $\langle \text{proof} \rangle$

lemma monic-gen : **assumes** $\text{bs} \subseteq \text{as}$
shows $\text{monic } (\prod (a, i) \in \text{bs}. a)$
 $\langle \text{proof} \rangle$

lemma nonzero-gen : **assumes** $\text{bs} \subseteq \text{as}$
shows $(\prod (a, i) \in \text{bs}. a) \neq 0$
 $\langle \text{proof} \rangle$

lemma monic-Prod : $\text{monic } ((\prod (a, i) \in \text{as}. a \wedge i))$
 $\langle \text{proof} \rangle$

lemma coprime-generic :
assumes bs : $\text{bs} \subseteq \text{as}$
and f : $\bigwedge a i. (a,i) \in \text{bs} \implies f i > 0$
shows $\text{coprime } (\prod (a, i) \in \text{bs}. a)$
 $(\sum (a, i) \in \text{bs}. (\prod (b, j) \in \text{bs} - \{(a, i)\}. b) * \text{smult } (\text{of-nat } (f i)) (\text{pderiv } a))$
 $(\text{is coprime } ?\text{single } ?\text{onederiv})$
 $\langle \text{proof} \rangle$

lemma pderiv-exp-gcd :
 $\text{gcd } p (\text{pderiv } p) = (\prod (a, i) \in \text{as}. a \wedge i) (\text{is } - = ?\text{prod})$
 $\langle \text{proof} \rangle$

lemma $\text{p-div-gcd-p-pderiv}$: $p \text{ div } (\text{gcd } p (\text{pderiv } p)) = (\prod (a, i) \in \text{as}. a)$
 $\langle \text{proof} \rangle$

fun $A B C D :: \text{nat} \Rightarrow 'a \text{ poly}$ **where**
 $A n = \text{gcd } (B n) (D n)$
 $| B 0 = p \text{ div } (\text{gcd } p (\text{pderiv } p))$
 $| B (\text{Suc } n) = B n \text{ div } A n$
 $| C 0 = \text{pderiv } p \text{ div } (\text{gcd } p (\text{pderiv } p))$
 $| C (\text{Suc } n) = D n \text{ div } A n$

| $D\ n = C\ n - pderiv\ (B\ n)$

lemma *A-B-C-D*: $A\ n = (\prod (a, i) \in as \cap UNIV \times \{n\}. a)$

$B\ n = (\prod (a, i) \in as - UNIV \times \{0 ..< n\}. a)$

$C\ n = (\sum (a, i) \in as - UNIV \times \{0 ..< n\}.$

$(\prod (b, j) \in as - UNIV \times \{0 ..< n\} - \{(a, i)\}. b) * smult\ (of\ nat\ (Suc\ i - n))$
 $(pderiv\ a))$

$D\ n = (\prod (a, i) \in as \cap UNIV \times \{n\}. a) *$

$(\sum (a, i) \in as - UNIV \times \{0 ..< Suc\ n\}.$

$(\prod (b, j) \in as - UNIV \times \{0 ..< Suc\ n\} - \{(a, i)\}. b) * (smult\ (of\ nat\ (i - n))\ (pderiv\ a)))$
 $\langle proof \rangle$

lemmas $A = A-B-C-D(1)$

lemmas $B = A-B-C-D(2)$

lemmas $ABCD-simps = A.simps\ B.simps\ C.simps\ D.simps$

declare $ABCD-simps[simp\ del]$

lemma *prod-A*:

$(\prod i = 0..< n. A\ i \wedge Suc\ i) = (\prod (a, i) \in as \cap UNIV \times \{0 ..< n\}. a \wedge Suc\ i)$
 $\langle proof \rangle$

lemma *prod-A-is-p-unknown*: **assumes** $\bigwedge a\ i. (a, i) \in as \implies i < n$

shows $p = (\prod i = 0..< n. A\ i \wedge Suc\ i)$
 $\langle proof \rangle$

definition *bound* :: *nat* **where**

$bound = Suc\ (Max\ (snd\ 'as))$

lemma *bound*: **assumes** $m: m \geq bound$

shows $B\ m = 1$
 $\langle proof \rangle$

lemma *coprime-A-A*: **assumes** $i \neq j$

shows $coprime\ (A\ i)\ (A\ j)$
 $\langle proof \rangle$

lemma *A-monic*: $monic\ (A\ i)$

$\langle proof \rangle$

lemma *A-square-free*: $square-free\ (A\ i)$

$\langle proof \rangle$

lemma *prod-A-is-p-B-bound*: **assumes** $B\ n = 1$

shows $p = (\prod i = 0..< n. A\ i \wedge Suc\ i)$
 $\langle proof \rangle$

interpretation *yun-gcd gcd* \langle proof \rangle

lemma *square-free-monic-poly*: $(poly (square-free-monic-poly p) x = 0) = (poly p x = 0)$
 \langle proof \rangle

lemma *yun-factorization-induct*: **assumes** *base*: $\bigwedge bn cn. bn = 1 \implies P bn cn$
and step: $\bigwedge bn cn. bn \neq 1 \implies P (bn \text{ div } (gcd bn (cn - pderiv bn)))$
 $((cn - pderiv bn) \text{ div } (gcd bn (cn - pderiv bn))) \implies P bn cn$
and id: $bn = p \text{ div } gcd p (pderiv p) \quad cn = pderiv p \text{ div } gcd p (pderiv p)$
shows $P bn cn$
 \langle proof \rangle

lemma *yun-factorization-main*: **assumes** *yun-factorization-main* $(B n) (C n) n$
 $bs = cs$
set $bs = \{(A i, i) \mid i. i < n\}$ *distinct* $(map \text{ snd } bs)$
shows $\exists m. \text{ set } cs = \{(A i, i) \mid i. i < m\} \wedge B m = 1 \wedge \text{ distinct } (map \text{ snd } cs)$
 \langle proof \rangle

lemma *yun-monic-factorization-res*: **assumes** *res*: *yun-monic-factorization* $p = bs$
shows $\exists m. \text{ set } bs = \{(A i, i) \mid i. i < m \wedge A i \neq 1\} \wedge B m = 1 \wedge \text{ distinct } (map \text{ snd } bs)$
 \langle proof \rangle

lemma *yun-monic-factorization*: **assumes** *yun*: *yun-monic-factorization* $p = bs$
shows *square-free-factorization* $p (1, bs) (b, i) \in \text{ set } bs \implies \text{ monic } b \text{ distinct } (map \text{ snd } bs)$
 \langle proof \rangle
end

lemma *monic-factorization*: **assumes** *monic* p
shows $\exists as. \text{ monic-factorization } as p$
 \langle proof \rangle

lemma *square-free-monic-poly*:
assumes *monic* $(p :: 'a :: \{\text{field-char-0, euclidean-ring-gcd, semiring-gcd-mult-normalize}\})$
poly
shows $(poly (yun-gcd.square-free-monic-poly gcd p) x = 0) = (poly p x = 0)$
 \langle proof \rangle

lemma *yun-factorization-induct*:
assumes *base*: $\bigwedge bn cn. bn = 1 \implies P bn cn$
and step: $\bigwedge bn cn. bn \neq 1 \implies P (bn \text{ div } (gcd bn (cn - pderiv bn)))$
 $((cn - pderiv bn) \text{ div } (gcd bn (cn - pderiv bn))) \implies P bn cn$
and id: $bn = p \text{ div } gcd p (pderiv p) \quad cn = pderiv p \text{ div } gcd p (pderiv p)$
and monic: *monic* $(p :: 'a :: \{\text{field-char-0, euclidean-ring-gcd, semiring-gcd-mult-normalize}\})$
poly
shows $P bn cn$
 \langle proof \rangle

lemma *square-free-poly*:

(*poly* (*square-free-poly gcd p*) $x = 0$) = (*poly p x = 0*)
<proof>

lemma *yun-monic-factorization*:

fixes $p :: 'a :: \{\text{field-char-0, euclidean-ring-gcd, semiring-gcd-mult-normalize}\}$ *poly*

assumes *res*: *yun-gcd.yun-monic-factorization gcd p = bs*

and *monic*: *monic p*

shows *square-free-factorization p (1,bs) (b,i) ∈ set bs ⇒ monic b distinct (map snd bs)*
<proof>

lemma *square-free-factorization-smult*: **assumes** $c: c \neq 0$

and *sf*: *square-free-factorization p (d,bs)*

shows *square-free-factorization (smult c p) (c * d, bs)*
<proof>

lemma *yun-factorization*: **assumes** *res*: *yun-factorization gcd p = c-bs*

shows *square-free-factorization p c-bs (b,i) ∈ set (snd c-bs) ⇒ monic b*
<proof>

lemma *prod-list-pow-suc*: $(\prod x \leftarrow bs. (x :: 'a :: \text{comm-monoid-mult}) * x^i)$
= *prod-list bs * prod-list bs ^ i*

<proof>

declare *irreducible-linear-field-poly*[intro!]

context

assumes *SORT-CONSTRAINT*('a :: {*field, factorial-ring-gcd, semiring-gcd-mult-normalize*})

begin

lemma *square-free-factorization-order-root-mem*:

assumes *sff*: *square-free-factorization p (c,bs)*

and $p: p \neq (0 :: 'a \text{ poly})$

and *ai*: $(a,i) \in \text{set } bs$ **and** *rt*: *poly a x = 0*

shows *order x p = Suc i*

<proof>

lemma *square-free-factorization-order-root-no-mem*:

assumes *sff*: *square-free-factorization p (c,bs)*

and $p: p \neq (0 :: 'a \text{ poly})$

and *no-root*: $\bigwedge a i. (a,i) \in \text{set } bs \Rightarrow \text{poly } a x \neq 0$

shows *order x p = 0*

<proof>

lemma square-free-factorization-order-root:
assumes *sff*: square-free-factorization p (c, bs)
and $p \neq 0 :: 'a$ poly)
shows $order\ x\ p = i \iff (i = 0 \wedge (\forall\ a\ j. (a,j) \in set\ bs \longrightarrow poly\ a\ x \neq 0) \vee (\exists\ a\ j. (a,j) \in set\ bs \wedge poly\ a\ x = 0 \wedge i = Suc\ j))$ (**is** ?l = (?r1 \vee ?r2))
 $\langle proof \rangle$

lemma square-free-factorization-root:
assumes *sff*: square-free-factorization p (c, bs)
and $p \neq 0 :: 'a$ poly)
shows $\{x. poly\ p\ x = 0\} = \{x. \exists\ a\ i. (a,i) \in set\ bs \wedge poly\ a\ x = 0\}$
 $\langle proof \rangle$

lemma square-free-factorizationD': fixes $p :: 'a$ poly
assumes *sf*: square-free-factorization p (c, bs)
shows $p = smult\ c\ (\prod (a, i) \leftarrow bs. a \wedge Suc\ i)$
and square-free (prod-list (map fst bs))
and $\bigwedge b\ i. (b,i) \in set\ bs \implies degree\ b \neq 0$
and $p = 0 \implies c = 0 \wedge bs = []$
 $\langle proof \rangle$

lemma square-free-factorizationI': fixes $p :: 'a$ poly
assumes *prod*: $p = smult\ c\ (\prod (a, i) \leftarrow bs. a \wedge Suc\ i)$
and *sf*: square-free (prod-list (map fst bs))
and *deg*: $\bigwedge b\ i. (b,i) \in set\ bs \implies degree\ b > 0$
and *0*: $p = 0 \implies c = 0 \wedge bs = []$
shows square-free-factorization p (c, bs)
 $\langle proof \rangle$

lemma square-free-factorization-def': fixes $p :: 'a$ poly
shows square-free-factorization p (c, bs) \iff
 $(p = smult\ c\ (\prod (a, i) \leftarrow bs. a \wedge Suc\ i)) \wedge$
 $(square-free\ (prod-list\ (map\ fst\ bs))) \wedge$
 $(\forall\ b\ i. (b,i) \in set\ bs \longrightarrow degree\ b > 0) \wedge$
 $(p = 0 \longrightarrow c = 0 \wedge bs = [])$
 $\langle proof \rangle$

lemma square-free-factorization-smult-prod-listI: fixes $p :: 'a$ poly
assumes *sff*: square-free-factorization p ($c, bs1 @ (smult\ b\ (prod-list\ bs), i) \# bs2$)
and $bs: \bigwedge b. b \in set\ bs \implies degree\ b > 0$
shows square-free-factorization p ($c * b \wedge (Suc\ i), bs1 @ map\ (\lambda\ b. (b,i))\ bs @ bs2$)
 $\langle proof \rangle$

lemma square-free-factorization-further-factorization: fixes $p :: 'a$ poly
assumes *sff*: square-free-factorization p (c, bs)
and $bs: \bigwedge b\ i\ d\ fs. (b,i) \in set\ bs \implies f\ b = (d, fs)$

$\implies b = \text{smult } d \text{ (prod-list fs)} \wedge (\forall f \in \text{set fs. degree } f > 0)$
and $h: h = (\lambda (b,i). \text{case } f \text{ b of } (d,fs) \Rightarrow (d \hat{\text{Suc}} i, \text{map } (\lambda f. (f,i)) fs))$
and $gs: gs = \text{map } h \text{ bs}$
and $d: d = c * \text{prod-list } (\text{map } \text{fst } gs)$
and $es: es = \text{concat } (\text{map } \text{snd } gs)$
shows $\text{square-free-factorization } p \text{ (d, es)}$
 $\langle \text{proof} \rangle$

lemma $\text{square-free-factorization-prod-listI}$: **fixes** $p :: 'a \text{ poly}$
assumes $\text{sff: square-free-factorization } p \text{ (c, bs1 @ ((prod-list bs),i) \# bs2)}$
and $bs: \bigwedge b. b \in \text{set } bs \implies \text{degree } b > 0$
shows $\text{square-free-factorization } p \text{ (c, bs1 @ map } (\lambda b. (b,i)) \text{ bs @ bs2)}$
 $\langle \text{proof} \rangle$

lemma $\text{square-free-factorization-factorI}$: **fixes** $p :: 'a \text{ poly}$
assumes $\text{sff: square-free-factorization } p \text{ (c, bs1 @ (a,i) \# bs2)}$
and $r: \text{degree } r \neq 0$ **and** $s: \text{degree } s \neq 0$
and $a: a = r * s$
shows $\text{square-free-factorization } p \text{ (c, bs1 @ ((r,i) \# (s,i) \# bs2))}$
 $\langle \text{proof} \rangle$

end

lemma $\text{monic-square-free-irreducible-factorization}$: **assumes** $\text{mon: monic } (f :: 'b$
 $:: \text{field poly})$
and $\text{sf: square-free } f$
shows $\exists P. \text{finite } P \wedge f = \prod P \wedge P \subseteq \{q. \text{irreducible } q \wedge \text{monic } q\}$
 $\langle \text{proof} \rangle$

context

assumes $\text{SORT-CONSTRAINT}('a :: \{\text{field, factorial-ring-gcd}\})$

begin

lemma $\text{monic-factorization-uniqueness}$:

fixes $P :: 'a \text{ poly set}$

assumes $\text{finite-P: finite } P$

and $PQ: \prod P = \prod Q$

and $P: P \subseteq \{q. \text{irreducible}_d q \wedge \text{monic } q\}$

and $\text{finite-Q: finite } Q$

and $Q: Q \subseteq \{q. \text{irreducible}_d q \wedge \text{monic } q\}$

shows $P = Q$

$\langle \text{proof} \rangle$

end

11.2 Yun factorization and homomorphisms

locale $\text{field-hom-0'} = \text{field-hom hom}$

for $\text{hom} :: 'a :: \{\text{field-char-0, field-gcd}\} \Rightarrow$

$'b :: \{\text{field-char-0, field-gcd}\}$

begin

sublocale *field-hom'* \langle proof \rangle
end

lemma (in *field-hom-0'*) *yun-factorization-main-hom*:
defines *hp*: $hp \equiv \text{map-poly hom}$
defines *hpi*: $hpi \equiv \text{map } (\lambda (f,i). (hp f, i :: \text{nat}))$
assumes *monic*: *monic p* **and** *f*: $f = p \text{ div gcd } p (pderiv p)$ **and** *g*: $g = pderiv p$
div gcd p (pderiv p)
shows *yun-gcd.yun-factorization-main gcd (hp f) (hp g) i (hpi as) = hpi (yun-gcd.yun-factorization-main gcd f g i as)*
 \langle proof \rangle

lemma *square-free-square-free-factorization*:
square-free (p :: 'a :: {field,factorial-ring-gcd,semiring-gcd-mult-normalize} poly)
 \implies
degree p \neq 0 \implies square-free-factorization p (1,[(p,0)])
 \langle proof \rangle

lemma *constant-square-free-factorization*:
degree p = 0 \implies square-free-factorization p (coeff p 0,[])
 \langle proof \rangle

lemma (in *field-hom-0'*) *yun-monic-factorization*:
defines *hp*: $hp \equiv \text{map-poly hom}$
defines *hpi*: $hpi \equiv \text{map } (\lambda (f,i). (hp f, i :: \text{nat}))$
assumes *monic*: *monic f*
shows *yun-gcd.yun-monic-factorization gcd (hp f) = hpi (yun-gcd.yun-monic-factorization gcd f)*
 \langle proof \rangle

lemma (in *field-hom-0'*) *yun-factorization-hom*:
defines *hp*: $hp \equiv \text{map-poly hom}$
defines *hpi*: $hpi \equiv \text{map } (\lambda (f,i). (hp f, i :: \text{nat}))$
shows *yun-factorization gcd (hp f) = map-prod hom hpi (yun-factorization gcd f)*
 \langle proof \rangle

lemma (in *field-hom-0'*) *square-free-map-poly*:
square-free (map-poly hom f) = square-free f
 \langle proof \rangle

end

12 GCD of rational polynomials via GCD for integer polynomials

This theory contains an algorithm to compute GCDs of rational polynomials via a conversion to integer polynomials and then invoking the integer polynomial GCD algorithm.

theory *Gcd-Rat-Poly*

imports

Gauss-Lemma

HOL-Computational-Algebra.Field-as-Ring

begin

definition *gcd-rat-poly* :: *rat poly* \Rightarrow *rat poly* \Rightarrow *rat poly* **where**

gcd-rat-poly *f g* = (let

f' = *snd* (*rat-to-int-poly* *f*);

g' = *snd* (*rat-to-int-poly* *g*);

h = *map-poly rat-of-int* (*gcd* *f' g'*)

in *smult* (*inverse* (*lead-coeff* *h*)) *h*)

lemma *gcd-rat-poly[simp]*: *gcd-rat-poly* = *gcd*

<proof>

lemma *gcd-rat-poly-unfold[code-unfold]*: *gcd* = *gcd-rat-poly* *<proof>*

end

13 Rational Factorization

We combine the rational root test, the formulas for explicit roots, and the Kronecker's factorization algorithm to provide a basic factorization algorithm for polynomial over rational numbers. Moreover, also the roots of a rational polynomial can be determined.

theory *Rational-Factorization*

imports

Explicit-Roots

Kronecker-Factorization

Square-Free-Factorization

Rational-Root-Test

Gcd-Rat-Poly

Show.Show-Poly

begin

function *roots-of-rat-poly-main* :: *rat poly* \Rightarrow *rat list* **where**

roots-of-rat-poly-main *p* = (let *n* = *degree* *p* in if *n* = 0 then [] else if *n* = 1 then [*roots1* *p*]

else if *n* = 2 then *rat-roots2* *p* else

case *rational-root-test* *p* of None \Rightarrow [] | Some *x* \Rightarrow *x* # *roots-of-rat-poly-main* (*p* *div* [:-*x*,1:])

<proof>

termination *<proof>*

lemma *roots-of-rat-poly-main-code*[code]: *roots-of-rat-poly-main* p = (let n = degree p in if n = 0 then [] else if n = 1 then [roots1 p] else if n = 2 then rat-roots2 p else case rational-root-test p of None \Rightarrow [] | Some x \Rightarrow x # *roots-of-rat-poly-main* (p div [-x,1:]))
<proof>

lemma *roots-of-rat-poly-main*: p \neq 0 \implies set (*roots-of-rat-poly-main* p) = {x. poly p x = 0}
<proof>

declare *roots-of-rat-poly-main.simps*[simp del]

definition *roots-of-rat-poly* :: rat poly \Rightarrow rat list **where**
roots-of-rat-poly p \equiv let (c,pis) = *yun-factorization gcd-rat-poly* p in concat (map (*roots-of-rat-poly-main* o fst) pis)

lemma *roots-of-rat-poly*: **assumes** p: p \neq 0
shows set (*roots-of-rat-poly* p) = {x. poly p x = 0}
<proof>

definition *root-free* :: 'a :: comm-semiring-0 poly \Rightarrow bool **where**
root-free p = (degree p = 1 \vee (\forall x. poly p x \neq 0))

lemma *irreducible-root-free*:
fixes p :: 'a :: idom poly
assumes *irreducible* p **shows** *root-free* p
<proof>

partial-function (*tailrec*) *factorize-root-free-main* :: rat poly \Rightarrow rat list \Rightarrow rat poly list \Rightarrow rat \times rat poly list **where**
[code]: *factorize-root-free-main* p xs fs = (case xs of Nil \Rightarrow let l = coeff p (degree p); q = smult (inverse l) p in (l, (if q = 1 then fs else q # fs))
| x # xs \Rightarrow if poly p x = 0 then *factorize-root-free-main* (p div [-x,1:]) (x # xs) ([-x,1:] # fs)
else *factorize-root-free-main* p xs fs)

definition *factorize-root-free* :: rat poly \Rightarrow rat \times rat poly list **where**
factorize-root-free p = (if degree p = 0 then (coeff p 0,[]) else *factorize-root-free-main* p (*roots-of-rat-poly* p) [])

lemma *factorize-root-free-0*[simp]: *factorize-root-free* 0 = (0,[])
<proof>

lemma *factorize-root-free*: **assumes** *res*: *factorize-root-free* $p = (c, qs)$
shows $p = \text{smult } c \text{ (prod-list } qs)$
 $\bigwedge q. q \in \text{set } qs \implies \text{root-free } q \wedge \text{monic } q \wedge \text{degree } q \neq 0$
 $\langle \text{proof} \rangle$

definition *rational-proper-factor* :: *rat poly* \Rightarrow *rat poly option* **where**
rational-proper-factor $p =$ (if *degree* $p \leq 1$ then *None*
else if *degree* $p = 2$ then (case *rat-roots2* p of *Nil* \Rightarrow *None* | *Cons* $x \ xs \Rightarrow$ *Some*
 $[-x, 1 \ ::]$)
else if *degree* $p = 3$ then (case *rational-root-test* p of *None* \Rightarrow *None* | *Some* x
 \Rightarrow *Some* $[-x, 1:]$)
else *kronecker-factorization-rat* $p)$

lemma *degree-1-dvd-root*: **assumes** q : *degree* ($q :: 'a :: \text{field poly}$) = 1
and *rt*: $\bigwedge x. \text{poly } p \ x \neq 0$
shows $\neg q \ \text{dvd } p$
 $\langle \text{proof} \rangle$

lemma *rational-proper-factor*:
degree $p > 0 \implies \text{rational-proper-factor } p = \text{None} \implies \text{irreducible}_d \ p$
rational-proper-factor $p = \text{Some } q \implies q \ \text{dvd } p \wedge \text{degree } q \geq 1 \wedge \text{degree } q <$
degree p
 $\langle \text{proof} \rangle$

function *factorize-rat-poly-main* :: *rat* \Rightarrow *rat poly list* \Rightarrow *rat poly list* \Rightarrow *rat* \times *rat*
poly list **where**
factorize-rat-poly-main $c \ \text{irr } [] = (c, \text{irr})$
| *factorize-rat-poly-main* $c \ \text{irr } (p \ \# \ ps) =$ (if *degree* $p = 0$
then *factorize-rat-poly-main* ($c * \text{coeff } p \ 0$) *irr* ps
else (case *rational-proper-factor* p of
None \Rightarrow *factorize-rat-poly-main* $c \ (p \ \# \ \text{irr}) \ ps$
| *Some* $q \Rightarrow$ *factorize-rat-poly-main* $c \ \text{irr } (q \ \# \ p \ \text{div } q \ \# \ ps))$)
 $\langle \text{proof} \rangle$

definition *factorize-rat-poly-main-wf-rel* = *inv-image* (*mult1* $\{(x, y). x < y\}$) ($\lambda(c, \text{irr}, ps). \text{mset } (\text{map } \text{degree } ps)$)

lemma *wf-factorize-rat-poly-main-wf-rel*: *wf* *factorize-rat-poly-main-wf-rel*
 $\langle \text{proof} \rangle$

lemma *factorize-rat-poly-main-wf-rel-sub*:
 $((a, b, ps), (c, d, p \ \# \ ps)) \in \text{factorize-rat-poly-main-wf-rel}$
 $\langle \text{proof} \rangle$

lemma *factorize-rat-poly-main-wf-rel-two*: **assumes** *degree q < degree p degree r < degree p*
shows $((a,b,q \# r \# ps), (c,d,p \# ps)) \in \text{factorize-rat-poly-main-wf-rel}$
 $\langle \text{proof} \rangle$

termination
 $\langle \text{proof} \rangle$

declare *factorize-rat-poly-main.simps*[*simp del*]

lemma *factorize-rat-poly-main*:
assumes *factorize-rat-poly-main c irr ps = (d,qs)*
and *Ball (set irr) irreducible_d*
shows *Ball (set qs) irreducible_d (is ?g1)*
and *smult c (prod-list (irr @ ps)) = smult d (prod-list qs) (is ?g2)*
 $\langle \text{proof} \rangle$

definition *factorize-rat-poly-basic p = factorize-rat-poly-main 1 [] [p]*

lemma *factorize-rat-poly-basic*: **assumes** *res: factorize-rat-poly-basic p = (c,qs)*
shows *p = smult c (prod-list qs)*
 $\bigwedge q. q \in \text{set } qs \implies \text{irreducible}_d q$
 $\langle \text{proof} \rangle$

We removed the *factorize-rat-poly* function from this theory, since the one in Berlekamp-Zassenhaus is easier to use and implements a more efficient algorithm.

end

References

- [1] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.
- [2] D. Yun. On square-free decomposition algorithms. In *Proc. the third ACM symposium on Symbolic and Algebraic Computation*, pages 26–35, 1976.