Polynomial Factorization*

René Thiemann and Akihisa Yamada

March 17, 2025

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

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

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 $p_1^1 \cdot \ldots \cdot p_n^n$ 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 $p_1 \cdot \ldots \cdot p_n$, 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 $n \ mod \ 30 = 11$, 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* ⇒ *'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* ∧ (∀ *i* < *length xs*. *ys* ! *i* ∈ *xs* ! *i*)}
⟨*proof*⟩

lemma *set-concat-lists*[*simp*]: *set* (*concat-lists xs*) = {*as*. *length as* = *length xs* ∧ (∀ *i*<*length xs*. *as* ! *i* ∈ *set* (*xs* ! *i*))}
  ⟨*proof*⟩

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

fun *find-map-filter* :: (*'a* ⇒ *'b*) ⇒ (*'b* ⇒ *bool*) ⇒ *'a list* ⇒ *'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*)

[1]The 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*: *find-map-filter f p as* = *Some b* $\Longrightarrow$ *p b* $\wedge$ *b* $\in$ *f* ' *set as*
⟨*proof*⟩

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

lemma *remdups-adj-sorted-distinct*[*simp*]: *sorted xs* $\Longrightarrow$ *distinct* (*remdups-adj xs*)
⟨*proof*⟩

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

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

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

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

⟨*proof*⟩

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

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

lemma *list-all2-map-map*: ($\bigwedge$ *x*. *x* $\in$ *set xs* $\Longrightarrow$ *R* (*f x*) (*g x*)) $\Longrightarrow$ *list-all2 R* (*map f xs*) (*map g 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 set*) *list* $\Rightarrow$ *bool* **where**
is-partition cs $\longleftrightarrow$ ($\forall$ *j*<*length cs*. $\forall$ *i*<*j*. *cs* ! *i* $\cap$ *cs* ! *j* = {})

definition *is-partition-alt* :: ('*a set*) *list* ⇒ *bool* **where**
is-partition-alt cs ⟷ (∀ *i j*. *i* < *length cs* ∧ *j* < *length cs* ∧ *i* ≠ *j* ⟶ *cs*!*i* ∩ *cs*!*j* = {})

lemma *is-partition-alt*: *is-partition* = *is-partition-alt*
⟨*proof*⟩

lemma *is-partition-Nil*:
is-partition [] = *True* ⟨*proof*⟩

lemma *is-partition-Cons*:
is-partition (*x*#*xs*) ⟷ *is-partition xs* ∧ *x* ∩ ⋃(*set xs*) = {} (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *is-partition-sublist*:
assumes *is-partition* (*us* @ *xs* @ *ys* @ *zs* @ *vs*)
shows *is-partition* (*xs* @ *zs*)
⟨*proof*⟩

lemma *is-partition-inj-map*:
assumes *is-partition xs*
and *inj-on f* (⋃*x* ∈ *set xs*. *x*)
shows *is-partition* (*map* ((‘) *f*) *xs*)
⟨*proof*⟩

context
begin
private fun *is-partition-impl* :: '*a set list* ⇒ '*a set option* **where**
is-partition-impl [] = *Some* {}
| *is-partition-impl* (*as* # *rest*) = *do* {
all ← *is-partition-impl rest*;
if as ∩ *all* = {} *then Some* (*all* ∪ *as*) *else None*
}

lemma *is-partition-code*[*code*]: *is-partition as* = (*is-partition-impl as* ≠ *None*)
⟨*proof*⟩
end

lemma *case-prod-partition*:
case-prod f (*partition p xs*) = *f* (*filter p xs*) (*filter* (*Not* ∘ *p*) *xs*)
⟨*proof*⟩

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

1.3 merging functions

definition *fun-merge* :: ('*a* ⇒ '*b*)*list* ⇒ '*a set list* ⇒ '*a* ⇒ '*b*
where *fun-merge fs as a* ≡ (*fs* ! (*LEAST i*. *i* < *length as* ∧ *a* ∈ *as* ! *i*)) *a*

lemma *fun-merge*: **assumes**
i: $i < length\ as$
and *a*: $a \in as\ !\ i$
and *ident*: $\bigwedge\ i\ j\ a.\ i < length\ as \Longrightarrow j < length\ as \Longrightarrow a \in as\ !\ i \Longrightarrow a \in as\ !\ j \Longrightarrow (fs\ !\ i)\ a = (fs\ !\ j)\ a$
shows $fun\text{-}merge\ fs\ as\ a = (fs\ !\ i)\ a$
⟨*proof*⟩

lemma *fun-merge-part*: **assumes**
part: *is-partition as*
and *i*: $i < length\ as$
and *a*: $a \in as\ !\ i$
shows $fun\text{-}merge\ fs\ as\ a = (fs\ !\ i)\ a$
⟨*proof*⟩

lemma *map-nth-conv*: $map\ f\ ss = map\ g\ ts \Longrightarrow \forall\ i < length\ ss.\ f(ss!i) = g(ts!i)$
⟨*proof*⟩

lemma *distinct-take-drop*:
assumes *dist*: *distinct vs* **and** *len*: $i < length\ vs$ **shows** $distinct(take\ i\ vs\ @\ drop\ (Suc\ i)\ vs)$ (**is** $distinct(?xs@?ys)$)
⟨*proof*⟩

lemma *map-nth-eq-conv*:
assumes *len*: $length\ xs = length\ ys$
shows $(map\ f\ xs = ys) = (\forall\ i{<}length\ ys.\ f\ (xs\ !\ i) = ys\ !\ i)$ (**is** $?l = ?r$)
⟨*proof*⟩

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

lemma *map-upt-add'*:
$map\ f\ [a..{<}a+b] = map\ (\lambda\ i.\ f\ (a + i))\ [0..{<}b]$
⟨*proof*⟩

definition *generate-lists* :: $nat \Rightarrow 'a\ list \Rightarrow 'a\ list\ list$
where $generate\text{-}lists\ n\ xs \equiv concat\text{-}lists\ (map\ (\lambda\ \text{-}.\ xs)\ [0\ ..{<}\ n])$

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

lemma *nth-append-take*:
assumes $i \leq length\ xs$ **shows** $(take\ i\ xs\ @\ y\#ys)!i = y$
⟨*proof*⟩

lemma *nth-append-take-is-nth-conv*:
assumes $i < j$ **and** $j \leq$ *length xs* **shows** $(take\ j\ xs\ @\ ys)!i = xs!i$
⟨*proof*⟩

lemma *nth-append-drop-is-nth-conv*:
assumes $j < i$ **and** $j \leq$ *length xs* **and** $i \leq$ *length xs*
shows $(take\ j\ xs\ @\ y\ \#\ drop\ (Suc\ j)\ xs)!i = xs!i$
⟨*proof*⟩

lemma *nth-append-take-drop-is-nth-conv*:
assumes $i \leq$ *length xs* **and** $j \leq$ *length xs* **and** $i \neq j$
shows $(take\ j\ xs\ @\ y\ \#\ drop\ (Suc\ j)\ xs)!i = xs!i$
⟨*proof*⟩

lemma *take-drop-imp-nth*: $[\![take\ i\ ss\ @\ x\ \#\ drop\ (Suc\ i)\ ss = ss]\!] \Longrightarrow x = ss!i$
⟨*proof*⟩

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

lemma *take-drop-update-second*: **assumes** $j <$ *length ds* **and** *length cs* = *length ds*
shows $(take\ j\ ds\ @\ drop\ j\ cs)[j := cs\ !\ j] = take\ j\ ds\ @\ drop\ j\ cs$
⟨*proof*⟩

lemma *nth-take-prefix*:
$length\ ys \leq length\ xs \Longrightarrow \forall\, i < length\ ys.\ xs!i = ys!i \Longrightarrow take\ (length\ ys)\ xs = ys$
⟨*proof*⟩

lemma *take-upt-idx*:
assumes i: $i <$ *length ls*
shows $take\ i\ ls = [\ ls\ !\ j\ .\ j \leftarrow [0..{<}i]]$
⟨*proof*⟩

fun *distinct-eq* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$ **where**
distinct-eq - [] = *True*
| $distinct\text{-}eq\ eq\ (x\ \#\ xs) = ((\forall\ y \in set\ xs.\ \neg\ (eq\ y\ x)) \wedge distinct\text{-}eq\ eq\ xs)$

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

lemma *append-Cons-nth-left*:
assumes $i <$ *length xs*
shows $(xs\ @\ u\ \#\ ys)\ !\ i = xs\ !\ i$

⟨*proof*⟩

lemma *append-Cons-nth-middle*:
assumes $i = length\ xs$
shows $(xs\ @\ y\ \#\ zs)\ !\ i = y$
⟨*proof*⟩

lemma *append-Cons-nth-right*:
assumes $i > length\ xs$
shows $(xs\ @\ u\ \#\ ys)\ !\ i = (xs\ @\ z\ \#\ ys)\ !\ i$
⟨*proof*⟩

lemma *append-Cons-nth-not-middle*:
assumes $i \neq length\ xs$
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 $length\ xs = length\ ys$
and $\bigwedge i.\ i < length\ xs \Longrightarrow length\ (xs\ !\ i) = length\ (ys\ !\ i)$
and $\bigwedge i\ j.\ i < length\ xs \Longrightarrow j < length\ (xs\ !\ i) \Longrightarrow P\ (xs\ !\ i\ !\ j)\ (ys\ !\ i\ !\ j)$
shows $\forall k{<}length\ (concat\ xs).\ P\ (concat\ xs\ !\ k)\ (concat\ ys\ !\ k)$
⟨*proof*⟩

lemma *eq-length-concat-nth*:
assumes $length\ xs = length\ ys$
and $\bigwedge i.\ i < length\ xs \Longrightarrow length\ (xs\ !\ i) = length\ (ys\ !\ i)$
shows $length\ (concat\ xs) = length\ (concat\ ys)$
⟨*proof*⟩

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

lemma *set-list-union*[*simp*]: *set* (*list-union xs ys*) = *set xs* ∪ *set ys*
⟨*proof*⟩

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

fun *list-inter* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
list-inter [] *bs* = []
| *list-inter* (*a*#*as*) *bs* =
(*if a* ∈ *set bs then a* # *list-inter as bs else list-inter as bs*)

lemma *set-list-inter*[*simp*]:
set (*list-inter xs ys*) = *set xs* ∩ *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* ∈ *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* ⟹ (*ss*!*0*)#*drop* (*Suc 0*) *ss* = *ss*
⟨*proof*⟩

lemma *set-foldr-remdups-set-map-conv*[*simp*]:
set (*foldr* (*λx xs. remdups* (*f x* @ *xs*)) *xs* []) = ⋃(*set* (*map* (*set* ∘ *f*) *xs*))
⟨*proof*⟩

lemma *subset-set-code*[*code-unfold*]: *set xs* ⊆ *set ys* ⟷ *list-all* (*λx. x* ∈ *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* ∪ *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* $\Longrightarrow$ *sorted ys* $\Longrightarrow$
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* $\Longrightarrow$ *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* ∈ *set xs* − *set ys* $\Longrightarrow$ ∃ *y*. *y* ∈ *set ys* ∧ *x* ≤ *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* $\Longrightarrow$ *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* < *length ys* $\Longrightarrow$ *xs* ! *i* ≤ *ys* ! *i*
shows *min-list xs* ≤ *min-list ys*
⟨*proof*⟩

lemma *min-list-ex*:
assumes *xs* ≠ [] **shows** ∃ *x*∈*set xs*. *min-list xs* = *x*
⟨*proof*⟩

lemma *min-list-subset*:

assumes *subset*: *set ys* ⊆ *set xs* **and** *mem*: *min-list xs* ∈ *set ys*
shows *min-list xs* = *min-list ys*
⟨*proof*⟩

Apply a permutation to a list.

primrec *permut-aux* :: *'a list* ⇒ (*nat* ⇒ *nat*) ⇒ *'a list* ⇒ *'a list* **where**
permut-aux [] - - = [] |
permut-aux (*a* # *as*) *f bs* = (*bs* ! *f 0*) # (*permut-aux as* (λ*n*. *f* (*Suc n*)) *bs*)

definition *permut* :: *'a list* ⇒ (*nat* ⇒ *nat*) ⇒ *'a list* **where**
permut as f = *permut-aux as f as*
declare *permut-def*[*simp*]

lemma *permut-aux-sound*:
assumes *i* < *length as*
shows *permut-aux as f bs* ! *i* = *bs* ! (*f i*)
⟨*proof*⟩

lemma *permut-sound*:
assumes *i* < *length as*
shows *permut as f* ! *i* = *as* ! (*f i*)
⟨*proof*⟩

lemma *permut-aux-length*:
assumes *bij-betw f* {..<*length as*} {..<*length bs*}
shows *length* (*permut-aux as f bs*) = *length as*
⟨*proof*⟩

lemma *permut-length*:
assumes *bij-betw f* {..< *length as*} {..< *length as*}
shows *length* (*permut as f*) = *length as*
⟨*proof*⟩

declare *permut-def*[*simp del*]

lemma *foldl-assoc*:
fixes *b* :: (*'a* ⇒ *'a*) ⇒ (*'a* ⇒ *'a*) ⇒ *'a* ⇒ *'a* (**infixl** ‹·› *55*)
assumes ⋀*f g h*. *f* · (*g* · *h*) = *f* · *g* · *h*
shows *foldl* (·) (*x* · *y*) *zs* = *x* · *foldl* (·) *y zs*
⟨*proof*⟩

lemma *foldr-assoc*:
assumes ⋀*f g h*. *b* (*b f g*) *h* = *b f* (*b g h*)
shows *foldr b xs* (*b y z*) = *b* (*foldr b xs y*) *z*
⟨*proof*⟩

lemma *foldl-foldr-o-id*:
foldl (∘) *id fs* = *foldr* (∘) *fs id*
⟨*proof*⟩

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**
  $span\ P\ (x\ \#\ xs) =$
    $(if\ P\ x\ then\ let\ (ys, zs) = span\ P\ xs\ in\ (x\ \#\ ys, zs)$
    $else\ ([], x\ \#\ xs))$ |
  $span$ - $[] = ([], [])$

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*: $length\ xs = n\ p\ xs$
  and *rel*: $length\ ys = n \bigwedge i.\ i < n \Longrightarrow r\ (xs\ !\ i)\ (ys\ !\ i)$
  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**
  $remdups\text{-}gen\ f\ [] = []$
| $remdups\text{-}gen\ f\ (x\ \#\ xs) = x\ \#\ remdups\text{-}gen\ f\ [y <- xs.\ \neg\ f\ x = f\ y]$

lemma *remdups-gen-subset*: *set* (*remdups-gen f xs*) ⊆ *set xs*
⟨*proof*⟩

lemma *remdups-gen-elem-imp-elem*: *x* ∈ *set* (*remdups-gen f xs*) ⟹ *x* ∈ *set xs*
⟨*proof*⟩

lemma *elem-imp-remdups-gen-elem*: *x* ∈ *set xs* ⟹ ∃ *y* ∈ *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 ∃ *k* < *length* (*concat xss*).
take k (*concat xss*) = *concat* (*take i xss*) @ *take j ys* ∧
concat xss ! *k* = *xss* ! *i* ! *j* ∧
drop (*Suc k*) (*concat xss*) = *drop* (*Suc j*) *ys* @ *concat* (*drop* (*Suc i*) *xss*)
⟨*proof*⟩

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

lemma *map-upt-len-same-len-conv*:
assumes *length xs* = *length ys*
shows *map* (λ*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* ∘ *f*) *xs*)
⟨*proof*⟩

lemma *UN-upt-len-conv* [*simp*]:
length xs = *n* ⟹ (⋃ *i* ∈ {*0* ..< *n*}. *f* (*xs* ! *i*)) = ⋃(*set* (*map f xs*))
⟨*proof*⟩

lemma *Ball-at-Least0LessThan-conv* [*simp*]:
length xs = *n* ⟹
(∀ *i* ∈ {*0* ..< *n*}. *P* (*xs* ! *i*)) ⟷ (∀ *x* ∈ *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** $y \in$ *set ys*
obtains x **where** $x \in$ *set xs* **and** *P x y*
⟨*proof*⟩

lemma *map-eq-conv′*:
map f xs = *map g ys* ⟷ *length xs* = *length ys* ∧ (∀ *i* < *length xs*. *f* (*xs* ! *i*) = *g* (*ys* ! *i*))
⟨*proof*⟩

lemma *list-3-cases*[*case-names Nil 1 2*]:
assumes *xs* = [] ⟹ *P*
and ⋀*x*. *xs* = [*x*] ⟹ *P*
and ⋀*x y ys*. *xs* = *x*#*y*#*ys* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *list-4-cases*[*case-names Nil 1 2 3*]:
assumes *xs* = [] ⟹ *P*
and ⋀*x*. *xs* = [*x*] ⟹ *P*
and ⋀*x y*. *xs* = [*x*,*y*] ⟹ *P*
and ⋀*x y z zs*. *xs* = *x* # *y* # *z* # *zs* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *foldr-append2* [*simp*]:
foldr ((@) ∘ *f*) *xs* (*ys* @ *zs*) = *foldr* ((@) ∘ *f*) *xs ys* @ *zs*
⟨*proof*⟩

lemma *foldr-append2-Nil* [*simp*]:
foldr ((@) ∘ *f*) *xs* [] @ *zs* = *foldr* ((@) ∘ *f*) *xs zs*
⟨*proof*⟩

lemma *UNION-set-zip*:
(⋃*x* ∈ *set* (*zip* [*0*..<*length xs*] (*map f xs*)). *g x*) = (⋃*i* < *length xs*. *g* (*i*, *f* (*xs* ! *i*)))
⟨*proof*⟩

lemma *zip-fst*: *p* ∈ *set* (*zip as bs*) ⟹ *fst p* ∈ *set as*
⟨*proof*⟩

lemma *zip-snd*: *p* ∈ *set* (*zip as bs*) ⟹ *snd p* ∈ *set bs*
⟨*proof*⟩

lemma *zip-size-aux*: *size-list* (*size o snd*) (*zip ts ls*) ≤ (*size-list size ls*)
⟨*proof*⟩

We definie 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* ⇒ *'a list* ⇒ *'a list* **where**
remove-nth n xs ≡ (*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* ≤ *n* ⟹ *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* ≤ *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*: ∀ *i*. *i* < *length as* ⟶ *P* (*as* ! *i*) (*bs* ! *i*)
shows ∀ *i*. *i* < *length* (*remove-nth p as*) ⟶ *P* (*remove-nth p as* ! *i*) (*remove-nth p bs* ! *i*)
⟨*proof*⟩

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

definition *adjust-idx* :: *nat* ⇒ *nat* ⇒ *nat* **where**
adjust-idx i j ≡ (*if j* < *i then j else* (*Suc j*))

definition *adjust-idx-rev* :: *nat* ⇒ *nat* ⇒ *nat* **where**

adjust-idx-rev i j ≡ (*if j* < *i then j else j* − *Suc 0*)

lemma *adjust-idx-rev1*: *adjust-idx-rev i* (*adjust-idx i j*) = *j*
⟨*proof*⟩

lemma *adjust-idx-rev2*:
assumes *j* ≠ *i* **shows** *adjust-idx i* (*adjust-idx-rev i j*) = *j*
⟨*proof*⟩

lemma *adjust-idx-i*:
adjust-idx i j ≠ *i*
⟨*proof*⟩

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

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

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

lemma *adjust-idx-rev-length*:
assumes *i* < *length xs*
and *j* < *length xs*
and *j* ≠ *i*
shows *adjust-idx-rev i j* < *length* (*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*: *length as* = *length bs*
and *lcd*: *length cs* = *length ds*
and *nsab*: ∀ *i*. *i* < *length bs* ⟶ *P* (*as* ! *i*) (*bs* ! *i*)
and *nscd*: ∀ *i*. *i* < *length ds* ⟶ *P* (*cs* ! *i*) (*ds* ! *i*)
shows ∀ *i*. *i* < *length* (*bs* @ *ds*) ⟶ *P* ((*as* @ *cs*) ! *i*) ((*bs* @ *ds*) ! *i*)
⟨*proof*⟩

Extension of filter and partition to binary relations.

fun *filter2* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a list* ⇒ *'b list* ⇒ (*'a list* × *'b 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*: ∀ *i*. *i* < *length* (*fst* (*filter2 P as bs*)) ⟶ *P* (*fst* (*filter2 P as bs*) ! *i*) (*snd* (*filter2 P as bs*) ! *i*)
⟨*proof*⟩

definition *partition2* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a list* ⇒ '*b list* ⇒ ('*a list* × '*b list*) × ('*a list* × '*b list*) **where**
partition2 P as bs ≡ ((*filter2 P as bs*) , (*filter2* (λ*a b*. ¬ (*P a b*)) *as bs*))

lemma *partition2-sound-P*: ∀ *i*. *i* < *length* (*fst* (*fst* (*partition2 P as bs*))) ⟶
P (*fst* (*fst* (*partition2 P as bs*)) ! *i*) (*snd* (*fst* (*partition2 P as bs*)) ! *i*)
⟨*proof*⟩

lemma *partition2-sound-nP*: ∀ *i*. *i* < *length* (*fst* (*snd* (*partition2 P as bs*))) ⟶
¬ *P* (*fst* (*snd* (*partition2 P as bs*)) ! *i*) (*snd* (*snd* (*partition2 P 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 ∀ *j*. *j* < *i* ⟶ *as* ! *j* ≠ *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-list* (*replicate n w1*) ≤ *sum-list* (*replicate n w2*)
⟨*proof*⟩

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 $f\ x < (\sum x{\leftarrow}xs.\ f\ x)$
⟨*proof*⟩

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

lemma *set-subset-insertI*: *set xs* ⊆ *set* (*List.insert x xs*)
⟨*proof*⟩

lemma *set-removeAll-subset*: *set* (*removeAll x xs*) ⊆ *set xs*
⟨*proof*⟩

lemma *map-of-append-Some*:
map-of xs y = *Some z* ⟹ *map-of* (*xs* @ *ys*) *y* = *Some z*
⟨*proof*⟩

lemma *map-of-append-None*:
map-of xs y = *None* ⟹ *map-of* (*xs* @ *ys*) *y* = *map-of ys y*
⟨*proof*⟩

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* < *length as*

shows *mset* (*remove-nth n as*) = *mset as* − {#(*as*!*n*)#}
⟨*proof*⟩

lemma *multiset-subset-insert*: {*ps*. *ps* ⊆# *add-mset x xs*} =
{*ps*. *ps* ⊆# *xs*} ∪ *add-mset x* ‘ {*ps*. *ps* ⊆# *xs*} (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *multiset-of-subseqs*: *mset* ‘ *set* (*subseqs xs*) = { *ps*. *ps* ⊆# *mset xs*}
⟨*proof*⟩

lemma *remove1-mset*: *w* ∈ *set vs* ⟹ *mset* (*remove1 w vs*) + {#*w*#} = *mset vs*
⟨*proof*⟩

lemma *fold-remove1-mset*: *mset ws* ⊆# *mset vs* ⟹ *mset* (*fold remove1 ws vs*) + *mset ws* = *mset vs*
⟨*proof*⟩

lemma *subseqs-sub-mset*: *ws* ∈ *set* (*subseqs vs*) ⟹ *mset ws* ⊆# *mset vs*
⟨*proof*⟩

lemma *filter-mset-inequality*: *filter-mset f xs* ≠ *xs* ⟹ ∃ *x* ∈# *xs*. ¬ *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* ∈ *set xs* ⟹ *Mapping.lookup* (*Mapping.tabulate xs f*) *x* = *Some* (*f x*)
⟨*proof*⟩

lemma *lookup-tabulate2*: *Mapping.lookup* (*Mapping.tabulate xs f*) *x* = *Some y* ⟹ *y* = *f x*
⟨*proof*⟩

definition *memo-int* :: *int* ⇒ *int* ⇒ (*int* ⇒ *'a*) ⇒ (*int* ⇒ *'a*) **where**
memo-int low up f ≡ *let m* = *Mapping.tabulate* [*low* .. *up*] *f*
in (λ *x*. *if x* ≥ *low* ∧ *x* ≤ *up then the* (*Mapping.lookup m x*) *else f x*)

lemma *memo-int*[*simp*]: *memo-int low up f* = *f*
⟨*proof*⟩

definition *memo-nat* :: *nat* ⇒ *nat* ⇒ (*nat* ⇒ ′*a*) ⇒ (*nat* ⇒ ′*a*) **where**
memo-nat low up f ≡ *let m* = *Mapping.tabulate* [*low* ..< *up*] *f*
in (λ *x*. *if x* ≥ *low* ∧ *x* < *up then the* (*Mapping.lookup m x*) *else f x*)

lemma *memo-nat*[*simp*]: *memo-nat low up f* = *f*
⟨*proof*⟩

definition *memo* :: ′*a list* ⇒ (′*a* ⇒ ′*b*) ⇒ (′*a* ⇒ ′*b*) **where**
memo xs f ≡ *let m* = *Mapping.tabulate xs f*
in (λ *x*. *case Mapping.lookup m x of None* ⇒ *f x* | *Some y* ⇒ *y*)

lemma *memo*[*simp*]: *memo xs f* = *f*
⟨*proof*⟩

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*:] = *Suc 0* ⟨*proof*⟩

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

lemma *linear-power-nonzero*: [: *a*, *1* :] ^ *n* ≠ *0*
⟨*proof*⟩

lemma *order-linear-power*′: *order a* ([: *b*, *1*:]^*Suc n*) = (*if b* = −*a then Suc n else 0*)
⟨*proof*⟩

lemma *order-linear-power*: *order a* ([: *b*, *1*:]^*n*) = (*if b* = −*a then n else 0*)
⟨*proof*⟩

lemma *order-linear*′: *order a* [: *b*, *1*:] = (*if b* = −*a then 1 else 0*)
⟨*proof*⟩

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

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

lemma *order-code*[*code*]: *order* (*a*::*'a*::*idom-divide*) *p* =
  (*if p* = *0 then Code.abort* (*STR ''order of polynomial 0 undefined''*) (λ -. *order a p*)
    else if poly p a $\neq$ *0 then 0 else Suc* (*order a* (*p 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*: *degree p* = *0*
  shows {*x*. *poly p x* = *0*} = {}
  ⟨*proof*⟩

definition *roots1* :: *'a* :: *field poly* ⇒ *'a* **where**
  roots1 p = (− *coeff p 0* / *coeff p 1*)

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

lemma *roots2*: **fixes** *p* :: *'a* :: *field-char-0 poly*
  assumes *p2*: *p* = [: *c*, *b*, *a* :] **and** *a*: *a* $\neq$ *0*
  shows {*x*. *poly p x* = *0*} = { − (*b* / (*2* ∗ *a*)) + *e* | *e*. *e*^*2* = (*b* / (*2* ∗ *a*))^*2* − *c*/*a*} (**is** *?l* = *?r*)
⟨*proof*⟩

definition *croots2* :: *complex poly* ⇒ *complex list* **where**

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

definition *complex-rat* :: *complex* ⇒ *bool* **where**
complex-rat x = (*Re x* ∈ ℚ ∧ *Im x* ∈ ℚ)

lemma *croots2*: **assumes** *degree p* = *2*
shows {*x*. *poly p x* = *0*} = *set* (*croots2 p*)
⟨*proof*⟩

definition *rroots2* :: *real poly* ⇒ *real list* **where**
rroots2 p = (*let a* = *coeff p 2*; *b* = *coeff p 1*; *c* = *coeff p 0*; *b2a* = *b* / (*2* ∗ *a*);
bac = *b2a*^*2* − *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* ⇒ *rat list* **where**
rat-roots2 p = (*let a* = *coeff p 2*; *b* = *coeff p 1*; *c* = *coeff p 0*; *b2a* = *b* / (*2* ∗ *a*);
bac = *b2a*^*2* − *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*⟩

Determinining roots of complex polynomials of degree up to 2.

definition *croots* :: *complex poly* ⇒ *complex list* **where**
croots p = (*if p* = *0* ∨ *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* ≠ *0 degree p* ≤ *2*
shows *set* (*croots p*) = {*x*. *poly p x* = *0*}
⟨*proof*⟩

Determinining roots of real polynomials of degree up to 2.

definition *rroots* :: *real poly* ⇒ *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 *set* (*rroots* p) = $\{x.$ *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* ⇒ *int* ⇒ (*int poly* × *int poly*) *option* ⇒ (*int poly* × *int poly*) *option* **where**
 div-int-poly-step q = (λ*a sro. case sro of Some* (*s*, *r*) ⇒
   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* ⇒ *None*)

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

definition *div-mod-int-poly* :: *int poly* ⇒ *int poly* ⇒ (*int poly* × *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* ⇒ *int poly* ⇒ *int poly option* **where**
  div-int-poly p q =
   (*case div-mod-int-poly p q of None* ⇒ *None* | *Some* (*d*,*m*) ⇒ *if m* = *0 then Some d else None*)

definition *div-rat-poly-step* :: *'a::field poly* ⇒ *'a* ⇒ *'a poly* × *'a poly* ⇒ *'a poly* × *'a poly* **where**
  div-rat-poly-step q = (λ*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-is-g* : $\bigwedge$ *a b c*. *b* ∈ *s* ⟹ *f' a* = *f b* (*f' c*) ⟹ *g' a* = *g b* (*g' c*)
   and *f'-inj* : $\bigwedge$ *a b*. *f' a* = *f' b* ⟹ *a* = *b*
   and *f-bit-sur* : $\bigwedge$ *a b c*. *f' a* = *f b c* ⟹ ∃ *c'*. *c* = *f' c'*

and *lst-in-s* : *set lst* ⊆ *s*
shows f' *a* = *foldr f lst* (f' *b*) ⟹ g' *a* = *foldr g lst* (g' *b*)
⟨*proof*⟩

abbreviation (*input*) *rp* :: *int poly* ⇒ *rat poly* **where**
rp ≡ *map-poly rat-of-int*

lemma *rat-int-poly-step-agree* :
assumes *coeff* (*pCons b c2*) (*degree q*) *mod coeff q* (*degree q*) = *0*
shows (*rp a1*,*rp a2*) = (*div-rat-poly-step* (*rp q*) ∘ *rat-of-int*) *b* (*rp c1*,*rp c2*)
⟷ *Some* (*a1*,*a2*) = *div-int-poly-step q b* (*Some* (*c1*,*c2*))
⟨*proof*⟩

lemma *int-step-then-rat-poly-step* :
assumes *Some*:*Some* (*a1*,*a2*) = *div-int-poly-step q b* (*Some* (*c1*,*c2*))
shows (*rp a1*,*rp a2*) = (*div-rat-poly-step* (*rp q*) ∘ *rat-of-int*) *b* (*rp c1*,*rp c2*)
⟨*proof*⟩

lemma *is-int-rat-division* :
assumes *y* ≠ *0*
shows *is-int-rat* (*rat-of-int x* / *rat-of-int y*) ⟷ *x mod y* = *0*
⟨*proof*⟩

lemma *pCons-of-rp-contains-ints* :
assumes *rp a* = *pCons b c*
shows *is-int-rat b*
⟨*proof*⟩

lemma *rat-step-then-int-poly-step* :
assumes *q* ≠ *0*
and (*rp a1*,*rp a2*) = (*div-rat-poly-step* (*rp q*) ∘ *rat-of-int*) *b2* (*rp c1*,*rp c2*)
shows *Some* (*a1*,*a2*) = *div-int-poly-step q b2* (*Some* (*c1*,*c2*))
⟨*proof*⟩

lemma *div-int-poly-step-surjective* : *Some a* = *div-int-poly-step q b c* ⟹ ∃ c'. *c* = *Some* c'
⟨*proof*⟩

lemma *div-mod-int-poly-then-pdivmod*:
assumes *div-mod-int-poly p q* = *Some* (*r*,*m*)
shows (*rp p div rp q*, *rp p mod rp q*) = (*rp r*, *rp m*)
and *q* ≠ *0*
⟨*proof*⟩

lemma *div-rat-poly-step-sur*:
assumes (*case a of* (*a*, *b*) ⇒ (*rp a*, *rp b*)) = (*div-rat-poly-step* (*rp q*) ∘ *rat-of-int*) *x pair*
shows ∃ c'. *pair* = (*case* c' *of* (*a*, *b*) ⇒ (*rp a*, *rp b*))

⟨*proof*⟩

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

lemma *div-int-then-rqp*:
assumes $div\text{-}int\text{-}poly\ p\ q = Some\ r$
shows $r * q = p$
and $q \neq 0$
⟨*proof*⟩

lemma *rqp-then-div-int*:
assumes $r * q = p$
and *q0*:$q \neq 0$
shows $div\text{-}int\text{-}poly\ p\ q = Some\ r$
⟨*proof*⟩

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

definition *dvd-int-poly* :: *int poly* ⇒ *int poly* ⇒ *bool* **where**
$dvd\text{-}int\text{-}poly\ q\ p = (if\ q = 0\ then\ p = 0\ else\ div\text{-}int\text{-}poly\ p\ q \neq None)$

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

definition *dvd-int-poly-non-0* :: *int poly* ⇒ *int poly* ⇒ *bool* **where**
$dvd\text{-}int\text{-}poly\text{-}non\text{-}0\ q\ p = (div\text{-}int\text{-}poly\ p\ q \neq None)$

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

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

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*
Polynomial-Interpolation.Missing-Polynomial
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* :: {*semiring-gcd*,*idom-divide*,*normalization-semidom-multiplicative*} *poly*
shows *primitive-part* (*normalize f*) = *normalize* (*primitive-part f*)
⟨*proof*⟩

lemma *length-coeffs-primitive-part*[*simp*]: *length* (*coeffs* (*primitive-part f*)) = *length* (*coeffs f*)
⟨*proof*⟩

lemma *degree-unit-factor*[*simp*]: *degree* (*unit-factor f*) = *0*
⟨*proof*⟩

lemma *degree-normalize*[*simp*]: *degree* (*normalize f*) = *degree f*
⟨*proof*⟩

lemma *content-iff*: *x dvd content p* ⟷ (∀ *c* ∈ *set* (*coeffs p*). *x dvd c*)
⟨*proof*⟩

lemma *is-unit-field-poly*[*simp*]: (*p*::′*a*::*field poly*) *dvd 1* ⟷ *p* ≠ *0* ∧ *degree p* = *0*
⟨*proof*⟩

definition *primitive* **where**
primitive f ⟷ (∀ *x*. (∀ *y* ∈ *set* (*coeffs f*). *x dvd y*) ⟶ *x dvd 1*)

lemma *primitiveI*:
assumes (⋀*x*. (⋀*y*. *y* ∈ *set* (*coeffs f*) ⟹ *x dvd y*) ⟹ *x dvd 1*)
shows *primitive f* ⟨*proof*⟩

lemma *primitiveD*:
assumes *primitive f*
shows (⋀*y*. *y* ∈ *set* (*coeffs f*) ⟹ *x dvd y*) ⟹ *x dvd 1*
⟨*proof*⟩

lemma *not-primitiveE*:
assumes ¬ *primitive f*
and ⋀*x*. (⋀*y*. *y* ∈ *set* (*coeffs f*) ⟹ *x dvd y*) ⟹ ¬ *x dvd 1* ⟹ *thesis*
shows *thesis* ⟨*proof*⟩

lemma *primitive-iff-content-eq-1*[*simp*]:
fixes *f* :: ′*a* :: *semiring-gcd poly*
shows *primitive f* ⟷ *content f* = *1*
⟨*proof*⟩

lemma *primitive-prod-list*:
fixes *fs* :: ′*a* :: {*factorial-semiring*,*semiring-Gcd*,*normalization-semidom-multiplicative*} *poly list*
assumes *primitive* (*prod-list fs*) **and** *f* ∈ *set fs* **shows** *primitive f*
⟨*proof*⟩

lemma *irreducible-imp-primitive*:
fixes *f* :: ′*a* :: {*idom*,*semiring-gcd*} *poly*
assumes *irr*: *irreducible f* **and** *deg*: *degree f* ≠ *0* **shows** *primitive f*
⟨*proof*⟩

lemma *irreducible-primitive-connect*:
fixes *f* :: *'a* :: {*idom*,*semiring-gcd*} *poly*
assumes *cf*: *primitive f* **shows** $irreducible_d$ *f* ⟷ *irreducible f* (**is** *?l* ⟷ *?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 ¬ *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 = *sdiv-poly p* (*content p*)
⟨*proof*⟩

definition *common-denom* :: *rat list* ⇒ *int* × *int list* **where**
common-denom xs ≡ *let*
nds = *map quotient-of xs*;
denom = *list-lcm* (*map snd nds*);
ints = *map* (λ (*n*,*d*). *n* ∗ *denom div d*) *nds*
in (*denom*, *ints*)

definition *rat-to-int-poly* :: *rat poly* ⇒ *int* × *int poly* **where**
rat-to-int-poly p ≡ *let*
ais = *coeffs p*;
d = *fst* (*common-denom ais*)
in (*d*, *map-poly* (λ *x*. *case quotient-of x of* (*p*,*q*) ⇒ *p* ∗ *d div q*) *p*)

definition *rat-to-normalized-int-poly* :: *rat poly* ⇒ *rat* × *int poly* **where**
rat-to-normalized-int-poly p ≡ *if p* = *0 then* (*1*,*0*) *else case rat-to-int-poly p of* (*s*,*q*)
⇒ (*of-int* (*content q*) / *of-int s*, *primitive-part q*)

lemma *rat-to-normalized-int-poly-code*[*code*]:
rat-to-normalized-int-poly p = (*if p* = *0 then* (*1*,*0*) *else case rat-to-int-poly p of* (*s*,*q*)
⇒ *let c* = *content q in* (*of-int c* / *of-int s*, *sdiv-poly q c*))
⟨*proof*⟩

lemma *common-denom*: **assumes** *cd*: *common-denom xs* = (*dd*,*ys*)
shows *xs* = *map* (λ *i*. *of-int i* / *of-int dd*) *ys dd* > *0*
⋀*x*. *x* ∈ *set xs* ⟹ *rat-of-int* (*case quotient-of x of* (*n*, *x*) ⇒ *n* ∗ *dd div x*) / *rat-of-int dd* = *x*
⟨*proof*⟩

lemma *rat-to-int-poly*: **assumes** *rat-to-int-poly p* = (*d*,*q*)
shows *p* = *smult* (*inverse* (*of-int d*)) (*map-poly of-int q*) *d* > *0*
⟨*proof*⟩

lemma *content-ge-0-int*: *content p* ≥ (*0* :: *int*)
⟨*proof*⟩

lemma *abs-content-int*[*simp*]: **fixes** *p* :: *int poly*
shows *abs* (*content p*) = *content p* ⟨*proof*⟩

lemma *content-smult-int*: **fixes** *p* :: *int poly*
shows *content* (*smult a p*) = *abs a* ∗ *content p* ⟨*proof*⟩

lemma *normalize-non-0-smult*: ∃ *a*. (*a* :: ′*a* :: *semiring-gcd*) ≠ *0* ∧ *smult a* (*primitive-part p*) = *p*
⟨*proof*⟩

lemma *rat-to-normalized-int-poly*: **assumes** *rat-to-normalized-int-poly p* = (*d*,*q*)
shows *p* = *smult d* (*map-poly of-int q*) *d* > *0 p* ≠ *0* ⟹ *content q* = *1 degree q*

= *degree p*
⟨*proof*⟩

lemma *content-dvd-1*:
content g = *1* **if** *content f* = (*1* :: *'a* :: *semiring-gcd*) *g dvd f*
⟨*proof*⟩

lemma *dvd-smult-int*: **fixes** *c* :: *int* **assumes** *c*: *c* ≠ *0*
and *dvd*: *q dvd* (*smult c p*)
shows *primitive-part q dvd p*
⟨*proof*⟩

lemma $irreducible_d$*-primitive-part*:
fixes *p* :: *int poly*
shows $irreducible_d$ (*primitive-part p*) ⟷ $irreducible_d$ *p* (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

lemma $irreducible_d$*-smult-int*:
fixes *c* :: *int* **assumes** *c*: *c* ≠ *0*
shows $irreducible_d$ (*smult c p*) = $irreducible_d$ *p* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma $irreducible_d$*-as-irreducible*:
fixes *p* :: *int poly*
shows $irreducible_d$ *p* ⟷ *irreducible* (*primitive-part p*)
⟨*proof*⟩

lemma *rat-to-int-factor-content-1*: **fixes** *p* :: *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* ≠ *0*
shows *p* = *rg* ∗ *sh*
⟨*proof*⟩

lemma *rat-to-int-factor-explicit*: **fixes** *p* :: *int poly*
assumes *pgh*: *map-poly rat-of-int p* = *g* ∗ *h*
and *g*: *rat-to-normalized-int-poly g* = (*r*,*rg*)
shows ∃ *r*. *p* = *rg* ∗ *smult* (*content p*) *r*
⟨*proof*⟩

lemma *rat-to-int-factor*: **fixes** *p* :: *int poly*
assumes *pgh*: *map-poly rat-of-int p* = *g* ∗ *h*
shows ∃ *g' h'*. *p* = *g'* ∗ *h'* ∧ *degree g'* = *degree g* ∧ *degree h'* = *degree h*
⟨*proof*⟩

lemma *rat-to-int-factor-normalized-int-poly*: **fixes** *p* :: *rat poly*

assumes *pgh*: $p = g * h$
and *p*: *rat-to-normalized-int-poly* $p = (i, ip)$
shows $\exists\ g'\ h'.\ ip = g' * h' \wedge degree\ g' = degree\ g$
⟨*proof*⟩

lemma *irreducible-smult* [*simp*]:
fixes *c* :: *'a* :: *field*
shows *irreducible* (*smult c p*) $\longleftrightarrow$ *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* :: *int poly*
assumes *p*: *irreducible*$_d$ *p*
shows *irreducible*$_d$ (*map-poly rat-of-int p*)
⟨*proof*⟩

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

lemma *dvd-content-dvd*: **assumes** *dvd*: *content f dvd content g primitive-part f dvd primitive-part g*
shows *f dvd g*
⟨*proof*⟩

lemma *sdiv-poly-smult*: $c \neq 0 \Longrightarrow$ *sdiv-poly* (*smult c f*) *c* = *f*
⟨*proof*⟩

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

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

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

lemma *gcd-rat-to-gcd-int*: *gcd* (*of-int-poly f* :: *rat poly*) (*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 Euclidean-Rings.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 Euclidean-Rings.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* ≥ *2*
and *small*: ⋀ *j*. *2* ≤ *j* ⟹ *j* < *i* ⟹ ¬ *j dvd n*
and *i*: ¬ *i* ∗ *i* ≤ *n*
shows *prime* (*n*::*nat*) ⟨*proof*⟩

lemma *candidate-invariant-0*: *candidate-invariant 0*
⟨*proof*⟩

lemma *next-candidates*: **assumes** *res*: *next-candidates n* = (*m*,*ps*)
and *n*: *candidate-invariant n*
shows *candidate-invariant m sorted ps* {*i*. *prime i* ∧ *n* ≤ *i* ∧ *i* < *m*} ⊆ *set ps*
set ps ⊆ {*2*..} ∩ {*n*..<*m*} *distinct ps ps* ≠ [] *n* < *m*
⟨*proof*⟩

lemma *prime-test-iterate2*: **assumes** *small*: ⋀ *j*. *2* ≤ *j* ⟹ *j* < (*i* :: *nat*) ⟹ ¬ *j dvd n*
and *odd*: *odd n*
and *n*: *n* ≥ *3*
and *i*: *i* ≥ *3 odd i*
and *mod*: ¬ *i dvd n*
and *j*: *2* ≤ *j j* < *i* + *2*
shows ¬ *j dvd n*
⟨*proof*⟩

lemma *prime-divisor*: **assumes** *j* ≥ *2* **and** *j dvd n* **shows**
∃ *p* :: *nat*. *prime p* ∧ *p dvd j* ∧ *p dvd n*
⟨*proof*⟩

lemma *prime-nat-main*: *ni* = (*n*,*i*,*is*) ⟹ *i* ≥ *2* ⟹ *n* ≥ *2* ⟹
(⋀ *j*. *2* ≤ *j* ⟹ *j* < *i* ⟹ ¬ (*j dvd n*)) ⟹
(⋀ *j*. *i* ≤ *j* ⟹ *j* < *jj* ⟹ *prime j* ⟹ *j* ∈ *set is*) ⟹ *i* ≤ *jj* ⟹
sorted is ⟹ *distinct is* ⟹ *candidate-invariant jj* ⟹ *set is* ⊆ {*i*..<*jj*} ⟹
res = *prime-nat-main n jj is* ⟹
res = *prime n*
⟨*proof*⟩

lemma *prime-factorization-nat-main*: *ni* = (*n*,*i*,*is*) ⟹ *i* ≥ *2* ⟹ *n* ≥ *2* ⟹
(⋀ *j*. *2* ≤ *j* ⟹ *j* < *i* ⟹ ¬ (*j dvd n*)) ⟹
(⋀ *j*. *i* ≤ *j* ⟹ *j* < *jj* ⟹ *prime j* ⟹ *j* ∈ *set is*) ⟹ *i* ≤ *jj* ⟹
sorted is ⟹ *distinct is* ⟹ *candidate-invariant jj* ⟹ *set is* ⊆ {*i*..<*jj*} ⟹
res = *prime-factorization-nat-main n jj is ps* ⟹
∃ *qs*. *res* = *qs* @ *ps* ∧ *Ball* (*set qs*) *prime* ∧ *n* = *prod-list qs*
⟨*proof*⟩

lemma *prime-nat*[*simp*]: *prime-nat n* = *prime n*
⟨*proof*⟩

lemma *prime-factorization-nat*: **fixes** *n* :: *nat*
defines *pf* ≡ *prime-factorization-nat n*
shows *Ball* (*set pf*) *prime*
and *n* ≠ *0* ⟹ *prod-list pf* = *n*
and *n* = *0* ⟹ *pf* = []
⟨*proof*⟩

lemma *prod-mset-multiset-prime-factorization-nat* [*simp*]:
$(x::nat) \neq 0 \Longrightarrow$ *prod-mset* (*prime-factorization* x) = x
⟨*proof*⟩

lemma *prime-factorization-unique''*:
fixes A :: $'a$ :: {*factorial-semiring-multiplicative*} *multiset*
assumes $\bigwedge p.\ p \in\#\ A \Longrightarrow$ *prime* p
assumes *prod-mset* A = *normalize* x
shows *prime-factorization* $x = A$
⟨*proof*⟩

lemma *multiset-prime-factorization-nat-correct*:
prime-factorization n = *mset* (*prime-factorization-nat* n)
⟨*proof*⟩

lemma *multiset-prime-factorization-code*[*code-unfold*]:
prime-factorization = (λn. *mset* (*prime-factorization-nat* n))
⟨*proof*⟩

lemma *divisors-nat*:
$n \neq 0 \Longrightarrow$ *set* (*divisors-nat* n) = {p. p *dvd* n} *distinct* (*divisors-nat* n) *divisors-nat* 0 = []
⟨*proof*⟩

lemma *divisors-int-pos*: $x \neq 0 \Longrightarrow$ *set* (*divisors-int-pos* x) = {i. i *dvd* $x \wedge i > 0$} *distinct* (*divisors-int-pos* x)
divisors-int-pos 0 = []
⟨*proof*⟩

lemma *divisors-int*: $x \neq 0 \Longrightarrow$ *set* (*divisors-int* x) = {i. i *dvd* x} *distinct* (*divisors-int* x)
divisors-int 0 = []
⟨*proof*⟩

definition *divisors-fun* :: ($'a \Rightarrow$ ($'a$ :: {*comm-monoid-mult*,*zero*}) *list*) $\Rightarrow$ *bool* **where**
divisors-fun df $\equiv$ ($\forall$ x. $x \neq 0 \longrightarrow$ *set* (*df* x) = { d. d *dvd* x }) $\wedge$ ($\forall$ x. *distinct* (*df* x))

lemma *divisors-funD*: *divisors-fun df* $\Longrightarrow x \neq 0 \Longrightarrow d$ *dvd* $x \Longrightarrow d \in$ *set* (*df* x)
⟨*proof*⟩

definition *divisors-pos-fun* :: ($'a \Rightarrow$ ($'a$ :: {*comm-monoid-mult*,*zero*,*ord*}) *list*) $\Rightarrow$ *bool* **where**
divisors-pos-fun df $\equiv$ ($\forall$ x. $x \neq 0 \longrightarrow$ *set* (*df* x) = { d. d *dvd* $x \wedge d > 0$}) $\wedge$ ($\forall$ x. *distinct* (*df* x))

lemma *divisors-pos-funD*: *divisors-pos-fun df* $\Longrightarrow$ *x* $\neq$ *0* $\Longrightarrow$ *d dvd x* $\Longrightarrow$ *d* $>$ *0* $\Longrightarrow$ *d* $\in$ *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* $\Longrightarrow$ *poly p x* = *0*
divisors-fun df $\Longrightarrow$ *divisors-pos-fun dp* $\Longrightarrow$ *rational-root-test-main df dp p* = *None* $\Longrightarrow$ ¬ (∃ *x*. *poly p x* = *0*)
⟨*proof*⟩

lemma *rational-root-test*:
rational-root-test p = *Some x* ⟹ *poly p x* = *0*
rational-root-test p = *None* ⟹ ¬ (∃ *x*. *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* ⇒ *int list*
and *dp* :: *int* ⇒ *int list*
and *bnd* :: *nat*
begin

definition *kronecker-samples* :: *nat* ⇒ *int list* **where**
kronecker-samples n ≡ *let min* = − *int* (*n div 2*) *in* [*min* .. *min* + *int n*]

lemma *kronecker-samples-0*: *0* ∈ *set* (*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* ⇒ *int poly option* **where**
kronecker-factorization-main p ≡ *if degree p* ≤ *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* ∈ 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 dvd y*) = (*map-poly rat-of-int x dvd map-poly of-int y*)
⟨*proof*⟩

lemma *content-x-minus-const-int*[*simp*]: *content* [: *c*, *1* :] = (*1* :: *int*)

⟨*proof*⟩

lemma *length-upto-add-nat*[*simp*]: *length* [*a* .. *a* + *int n*] = *Suc n*
⟨*proof*⟩

lemma *kronecker-samples*: *distinct* (*kronecker-samples n*) *length* (*kronecker-samples n*) = *Suc n*
⟨*proof*⟩

lemma *dvd-int-poly-non-0-degree-1*[*simp*]: *degree q* ≥ *1* ⟹ *dvd-int-poly-non-0 q p* = (*q dvd p*)
⟨*proof*⟩

context fixes *df dp* :: *int* ⇒ *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* ≥ *2* ⟹ *bnd* ≥ *1*
shows *degree q* ≥ *1 degree q* ≤ *bnd q dvd p*
⟨*proof*⟩

lemma *kronecker-factorization-rat-main-sound*: **assumes**
some: *kronecker-factorization-rat-main df dp bnd p* = *Some q*
and *bnd*: *degree p* ≥ *2* ⟹ *bnd* ≥ *1*
shows *degree q* ≥ *1 degree q* ≤ *bnd q dvd p*
⟨*proof*⟩

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 ¬ (∃ *q*. *1* ≤ *degree q* ∧ *degree q* ≤ *bnd* ∧ *q dvd p*)
⟨*proof*⟩

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* $\Longrightarrow$
degree q $\geq$ *1* $\wedge$ *degree q* $<$ *degree p* $\wedge$ *q dvd p*
kronecker-factorization p = *None* $\Longrightarrow$ *degree p* $\geq$ *1* $\Longrightarrow$ $irreducible_d$ *p*
⟨*proof*⟩

lemma *kronecker-factorization-rat*:
kronecker-factorization-rat p = *Some q* $\Longrightarrow$
degree q $\geq$ *1* $\wedge$ *degree q* $<$ *degree p* $\wedge$ *q dvd p*
kronecker-factorization-rat p = *None* $\Longrightarrow$ *degree p* $\geq$ *1* $\Longrightarrow$ $irreducible_d$ *p*
⟨*proof*⟩

end
end

10 Polynomial Divisibility

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

theory *Polynomial-Irreducibility*
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 $\Longrightarrow$ *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* $\longleftrightarrow$ *p dvd q* $\vee$ *p dvd r*
⟨*proof*⟩

lemma *irreducible-dvd-pow*:
fixes *p* :: *'a poly*
assumes *irr*: *irreducible p*
shows *p dvd q* ^ *n* $\Longrightarrow$ *p dvd q*
⟨*proof*⟩

lemma *irreducible-dvd-prod*: **fixes** *p* :: *'a poly*
assumes *irr*: *irreducible p*
and *dvd*: *p dvd prod f as*
shows ∃ *a* ∈ *as. p dvd f a*
⟨*proof*⟩

lemma *irreducible-dvd-prod-list*: **fixes** *p* :: *'a poly*
assumes *irr*: *irreducible p*
and *dvd*: *p dvd prod-list as*
shows ∃ *a* ∈ *set as. p dvd a*
⟨*proof*⟩

lemma *dvd-mult-imp-degree*: **fixes** *p* :: *'a poly*
assumes *p dvd q* ∗ *r*
and *degree p* > *0*
shows ∃ *s t. irreducible s* ∧ *p* = *s* ∗ *t* ∧ (*s dvd q* ∨ *s dvd r*)
⟨*proof*⟩

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* :: *complex poly*
shows ∃ *as. smult* (*coeff p* (*degree p*)) (∏ *a* ← *as.* [:− *a*, *1*:]) = *p* ∧ *length as* = *degree p*
⟨*proof*⟩

lemma *rsquarefree-card-degree*: **assumes** *p0*: (*p* :: *complex poly*) ≠ *0*
shows *rsquarefree p* = (*card* {*x. poly p x* = *0*} = *degree p*)
⟨*proof*⟩

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-Irreducibility
  Order-Polynomial
  Fundamental-Theorem-Algebra-Factorized
  Polynomial-Interpolation.Ring-Hom-Poly
begin

definition *square-free* :: *'a* :: *comm-semiring-1 poly* ⇒ *bool* **where**
  square-free p = (*p* ≠ *0* ∧ (∀ *q*. *degree q* > *0* ⟶ ¬ (*q* ∗ *q dvd p*)))

lemma *square-freeI*:
  assumes ⋀ *q*. *degree q* > *0* ⟹ *q* ≠ *0* ⟹ *q* ∗ *q dvd p* ⟹ *False*
  and *p*: *p* ≠ *0*
  shows *square-free p* ⟨*proof*⟩

lemma *square-free-multD*:
  assumes *sf*: *square-free* (*f* ∗ *g*)
  shows *h dvd f* ⟹ *h dvd g* ⟹ *degree h* = *0 square-free f square-free g*
⟨*proof*⟩

lemma *irreducible$_d$-square-free*:
  fixes *p* :: *'a* :: {*comm-semiring-1*, *semiring-no-zero-divisors*} *poly*
  shows *irreducible$_d$ p* ⟹ *square-free p*
  ⟨*proof*⟩

lemma *square-free-factor*: **assumes** *dvd*: *a dvd p*
  and *sf*: *square-free p*
  shows *square-free a*
⟨*proof*⟩

lemma *square-free-prod-list-distinct*:
  assumes *sf*: *square-free* (*prod-list us* :: *'a* :: *idom poly*)
  and *us*: ⋀ *u*. *u* ∈ *set us* ⟹ *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* (∏ *fs*)
and *fin*: *finite fs*
and *f*: *f* ∈ *fs*
and *g*: *g* ∈ *fs*
and *fg*: *f* ≠ *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*: ¬ *separable f*
shows ∃ *g k*. *f* = *g* ∗ *k* ∧ *degree g* ≠ *0* ∧ *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* ≠ *0* ⟹ *square-free* (*f* :: *'a poly*) ⟹ *square-free* (*smult c f*)

⟨*proof*⟩

lemma *square-free-smult-iff* [*simp*]: $c \neq 0 \Longrightarrow$ *square-free* (*smult c f*) = *square-free* (*f* :: *'a poly*)
⟨*proof*⟩
end

context
assumes *SORT-CONSTRAINT*(*'a*::*factorial-ring-gcd*)
begin
definition *square-free-factorization* :: *'a poly* ⇒ *'a* × (*'a poly* × *nat*) *list* ⇒ *bool*
where
square-free-factorization p cbs ≡ *case cbs of* (*c*,*bs*) ⇒
(*p* = *smult c* ($\prod$(*a*, *i*)∈ *set bs*. *a* ^ *i*))
∧ (*p* = *0* ⟶ *c* = *0* ∧ *bs* = [])
∧ (∀ *a i*. (*a*,*i*) ∈ *set bs* ⟶ *square-free a* ∧ *degree a* > *0* ∧ *i* > *0*)
∧ (∀ *a i b j*. (*a*,*i*) ∈ *set bs* ⟶ (*b*,*j*) ∈ *set bs* ⟶ (*a*,*i*) ≠ (*b*,*j*) ⟶ *coprime a b*)
∧ *distinct bs*

lemma *square-free-factorizationD*: **assumes** *square-free-factorization p* (*c*,*bs*)
shows *p* = *smult c* ($\prod$(*a*, *i*)∈ *set bs*. *a* ^ *i*)
(*a*,*i*) ∈ *set bs* ⟹ *square-free a* ∧ *degree a* ≠ *0* ∧ *i* > *0*
(*a*,*i*) ∈ *set bs* ⟹ (*b*,*j*) ∈ *set bs* ⟹ (*a*,*i*) ≠ (*b*,*j*) ⟹ *coprime a b*
p = *0* ⟹ *c* = *0* ∧ *bs* = []
distinct bs
⟨*proof*⟩

lemma *square-free-factorization-prod-list*: **assumes** *square-free-factorization p* (*c*,*bs*)
shows *p* = *smult c* (*prod-list* (*map* (λ (*a*,*i*). *a* ^ *i*) *bs*))
⟨*proof*⟩
end

11.1 Yun's factorization algorithm

locale *yun-gcd* =
fixes *Gcd* :: *'a* :: *factorial-ring-gcd poly* ⇒ *'a poly* ⇒ *'a poly*
begin

partial-function (*tailrec*) *yun-factorization-main* ::
'a poly ⇒ *'a poly* ⇒
nat ⇒ (*'a poly* × *nat*)*list* ⇒ (*'a poly* × *nat*)*list* **where**
[*code*]: *yun-factorization-main bn cn i 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*,*Suc i*) # *sqr*)))

definition *yun-monic-factorization* :: *'a poly* ⇒ (*'a poly* × *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* (λ (*a*,*i*). *a* ≠ *1*) (*yun-factorization-main b0 c0 0* [])))

definition *square-free-monic-poly* :: *'a poly* ⇒ *'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* ⇒ *'a poly* ⇒ *'a poly*
begin
interpretation *yun-gcd Gcd* ⟨*proof*⟩

definition *square-free-poly* :: *'a poly* ⇒ *'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* ⇒ *'a* × (*'a poly* × *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* × *nat*) *set*
and *p* :: *'a poly*
assumes *p*: *p* = *prod* (λ (*a*,*i*). *a* ^ *Suc i*) *as*
and *fin*: *finite as*
assumes *as-distinct*: ⋀ *a i b j*. (*a*,*i*) ∈ *as* ⟹ (*b*,*j*) ∈ *as* ⟹ (*a*,*i*) ≠ (*b*,*j*) ⟹ *a* ≠ *b*
and *as-irred*: ⋀ *a i*. (*a*,*i*) ∈ *as* ⟹ $irreducible_d$ *a*
and *as-monic*: ⋀ *a i*. (*a*,*i*) ∈ *as* ⟹ *monic a*
begin

lemma *poly-exp-expand*:
  $p = (prod\ (\lambda\ (a,i).\ a\ \widehat{}\ i)\ as) * prod\ (\lambda\ (a,i).\ a)\ as$
  ⟨*proof*⟩

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

lemma *monic-gen*: **assumes** $bs \subseteq as$
  shows $monic\ (\prod\ (a,\ i) \in bs.\ a)$
  ⟨*proof*⟩

lemma *nonzero-gen*: **assumes** $bs \subseteq as$
  shows $(\prod\ (a,\ i) \in bs.\ a) \neq 0$
  ⟨*proof*⟩

lemma *monic-Prod*: $monic\ ((\prod (a,\ i){\in}as.\ a\ \widehat{}\ i))$
  ⟨*proof*⟩

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

lemma *pderiv-exp-gcd*:
  $gcd\ p\ (pderiv\ p) = (\prod (a,\ i){\in}as.\ a\ \widehat{}\ i)$ (**is** - = *?prod*)
⟨*proof*⟩

lemma *p-div-gcd-p-pderiv*: $p\ div\ (gcd\ p\ (pderiv\ p)) = (\prod (a,\ i){\in}as.\ a)$
  ⟨*proof*⟩

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

lemma *A-B-C-D*: $A\ n = (\prod\ (a,\ i) \in as \cap UNIV \times \{n\}.\ a)$
  $B\ n = (\prod\ (a,\ i) \in as - UNIV \times \{0\ ..{<}\ n\}.\ a)$
  $C\ n = (\sum (a,\ i){\in}as - UNIV \times \{0\ ..{<}\ n\}.$
    $(\prod (b,\ j){\in}as - UNIV \times \{0\ ..{<}\ n\} - \{(a,\ i)\}.\ b) * smult\ (of\text{-}nat\ (Suc\ i - n))$
$(pderiv\ a))$

$D\ n = (\prod\ (a,\ i) \in as \cap UNIV \times \{n\}.\ a) *$
$(\sum\ (a,i) \in as - UNIV \times \{0\ ..< Suc\ n\}.$
$(\prod (b,\ j) \in as - UNIV \times \{0\ ..< Suc\ n\} - \{(a,\ i)\}.\ b) * (smult\ (of\text{-}nat\ (i -$
$n))\ (pderiv\ a)))$
⟨*proof*⟩

lemmas $A = A\text{-}B\text{-}C\text{-}D(1)$
lemmas $B = A\text{-}B\text{-}C\text{-}D(2)$

lemmas *ABCD-simps* = *A.simps B.simps C.simps D.simps*
declare *ABCD-simps*[*simp del*]

lemma *prod-A*:
$(\prod i = 0..< n.\ A\ i\ \hat{}\ Suc\ i) = (\prod (a,\ i) \in as \cap UNIV \times \{0\ ..< n\}.\ a\ \hat{}\ Suc\ i)$
⟨*proof*⟩

lemma *prod-A-is-p-unknown*: **assumes** $\bigwedge a\ i.\ (a,i) \in as \Longrightarrow i < n$
shows $p = (\prod i = 0..< n.\ A\ i\ \hat{}\ Suc\ i)$
⟨*proof*⟩

definition *bound* :: *nat* **where**
bound = *Suc* (*Max* (*snd* ‘ *as*))

lemma *bound*: **assumes** *m*: $m \geq bound$
shows $B\ m = 1$
⟨*proof*⟩

lemma *coprime-A-A*: **assumes** $i \neq j$
shows *coprime* (*A i*) (*A j*)
⟨*proof*⟩

lemma *A-monic*: *monic* (*A i*)
⟨*proof*⟩

lemma *A-square-free*: *square-free* (*A i*)
⟨*proof*⟩

lemma *prod-A-is-p-B-bound*: **assumes** $B\ n = 1$
shows $p = (\prod i = 0..< n.\ A\ i\ \hat{}\ Suc\ i)$
⟨*proof*⟩

interpretation *yun-gcd gcd* ⟨*proof*⟩

lemma *square-free-monic-poly*: (*poly* (*square-free-monic-poly p*) $x = 0$) = (*poly p* $x = 0$)
⟨*proof*⟩

lemma *yun-factorization-induct*: **assumes** *base*: $\bigwedge bn\ cn.\ bn = 1 \Longrightarrow P\ bn\ cn$

and *step*: $\bigwedge$ *bn cn*. *bn* ≠ *1* ⟹ *P* (*bn div* (*gcd bn* (*cn* − *pderiv bn*)))
((*cn* − *pderiv bn*) *div* (*gcd bn* (*cn* − *pderiv bn*))) ⟹ *P bn cn*
and *id*: *bn* = *p div gcd p* (*pderiv p*) *cn* = *pderiv p div gcd p* (*pderiv p*)
shows *P bn cn*
⟨*proof*⟩

lemma *yun-factorization-main*: **assumes** *yun-factorization-main* (*B n*) (*C n*) *n bs* = *cs*
set bs = {(*A i*, *Suc i*) | *i*. *i* < *n*} *distinct* (*map snd bs*)
shows ∃ *m*. *set cs* = {(*A i*, *Suc i*) | *i*. *i* < *m*} ∧ *B m* = *1* ∧ *distinct* (*map snd cs*)
⟨*proof*⟩

lemma *yun-monic-factorization-res*: **assumes** *res*: *yun-monic-factorization p* = *bs*
shows ∃ *m*. *set bs* = {(*A i*, *Suc i*) | *i*. *i* < *m* ∧ *A i* ≠ *1*} ∧ *B m* = *1* ∧ *distinct* (*map snd bs*)
⟨*proof*⟩

lemma *yun-monic-factorization*: **assumes** *yun*: *yun-monic-factorization p* = *bs*
shows *square-free-factorization p* (*1*,*bs*) (*b*,*i*) ∈ *set bs* ⟹ *monic b distinct* (*map snd bs*)
⟨*proof*⟩
end

lemma *monic-factorization*: **assumes** *monic p*
shows ∃ *as*. *monic-factorization as p*
⟨*proof*⟩

lemma *square-free-monic-poly*:
assumes *monic* (*p* :: ′*a* :: {*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*)
⟨*proof*⟩

lemma *yun-factorization-induct*:
assumes *base*: $\bigwedge$ *bn cn*. *bn* = *1* ⟹ *P bn cn*
and *step*: $\bigwedge$ *bn cn*. *bn* ≠ *1* ⟹ *P* (*bn div* (*gcd bn* (*cn* − *pderiv bn*)))
((*cn* − *pderiv bn*) *div* (*gcd bn* (*cn* − *pderiv bn*))) ⟹ *P bn cn*
and *id*: *bn* = *p div gcd p* (*pderiv p*) *cn* = *pderiv p div gcd p* (*pderiv p*)
and *monic*: *monic* (*p* :: ′*a* :: {*field-char-0*,*euclidean-ring-gcd*,*semiring-gcd-mult-normalize*} *poly*)
shows *P bn cn*
⟨*proof*⟩

lemma *square-free-poly*:
(*poly* (*square-free-poly gcd p*) *x* = *0*) = (*poly p x* = *0*)
⟨*proof*⟩

lemma *yun-monic-factorization*:
fixes *p* :: *'a* :: {*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* ≠ *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*: (∏ *x*←*bs*. (*x* :: *'a* :: *comm-monoid-mult*) ^ *i*)
= *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* ≠ (*0* :: *'a poly*)
and *ai*: (*a*,*i*) ∈ *set bs* **and** *rt*: *poly a x* = *0*
shows *order x p* = *i*
⟨*proof*⟩

lemma *square-free-factorization-order-root-no-mem*:
assumes *sff*: *square-free-factorization p* (*c*,*bs*)
and *p*: *p* ≠ (*0* :: *'a poly*)
and *no-root*: ⋀ *a i*. (*a*,*i*) ∈ *set bs* ⟹ *poly a x* ≠ *0*
shows *order x p* = *0*
⟨*proof*⟩

lemma *square-free-factorization-order-root*:
assumes *sff*: *square-free-factorization p* (*c*,*bs*)
and *p*: *p* ≠ (*0* :: *'a poly*)
shows *order x p* = *i* ⟷ (*i* = *0* ∧ (∀ *a j*. (*a*,*j*) ∈ *set bs* ⟶ *poly a x* ≠ *0*)
∨ (∃ *a j*. (*a*,*j*) ∈ *set bs* ∧ *poly a x* = *0* ∧ *i* = *j*)) (**is** *?l* = (*?r1* ∨ *?r2*))
⟨*proof*⟩

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

lemma *square-free-factorizationD′*: **fixes** $p :: 'a\ poly$
  assumes *sf*: *square-free-factorization p* (*c*, *bs*)
  shows $p = smult\ c\ (\prod(a,\ i) \leftarrow bs.\ a\ \hat{}\ i)$
    and *square-free* (*prod-list* (*map fst bs*))
    and $\bigwedge\ b\ i.\ (b,i) \in set\ bs \Longrightarrow degree\ b > 0 \wedge i > 0$
    and $p = 0 \Longrightarrow c = 0 \wedge bs = []$
⟨*proof*⟩

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

lemma *square-free-factorization-def′*: **fixes** $p :: 'a\ poly$
  shows *square-free-factorization p* (*c*,*bs*) ⟷
  $(p = smult\ c\ (\prod(a,\ i) \leftarrow bs.\ a\ \hat{}\ i)) \wedge$
  (*square-free* (*prod-list* (*map fst bs*))) ∧
  $(\forall\ b\ i.\ (b,i) \in set\ bs \longrightarrow degree\ b > 0 \wedge i > 0) \wedge$
  $(p = 0 \longrightarrow c = 0 \wedge bs = [])$
  ⟨*proof*⟩

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*: $\bigwedge\ b.\ b \in set\ bs \Longrightarrow degree\ b > 0$
  shows *square-free-factorization p* ($c * b\hat{}i$, *bs1* @ *map* (λ *b*. (*b*,*i*)) *bs* @ *bs2*)
⟨*proof*⟩

lemma *square-free-factorization-further-factorization*: **fixes** $p :: 'a\ poly$
  assumes *sff*: *square-free-factorization p* (*c*, *bs*)
  and *bs*: $\bigwedge\ b\ i\ d\ fs.\ (b,i) \in set\ bs \Longrightarrow f\ b = (d,fs)$
    $\Longrightarrow b = smult\ d\ (prod\text{-}list\ fs) \wedge (\forall\ f \in set\ fs.\ degree\ f > 0)$
  and *h*: $h = (\lambda\ (b,i).\ case\ f\ b\ of\ (d,fs) \Rightarrow (d\hat{}i, map\ (\lambda\ f.\ (f,i))\ fs))$
  and *gs*: *gs* = *map h bs*
  and *d*: $d = c * prod\text{-}list\ (map\ fst\ gs)$
  and *es*: *es* = *concat* (*map snd gs*)
  shows *square-free-factorization p* (*d*, *es*)
⟨*proof*⟩

lemma *square-free-factorization-prod-listI*: **fixes** *p* :: *'a poly*
assumes *sff*: *square-free-factorization p* (*c*, *bs1* @ ((*prod-list bs*),*i*) # *bs2*)
and *bs*: $\bigwedge$ *b*. *b* ∈ *set bs* ⟹ *degree b* > *0*
shows *square-free-factorization p* (*c*, *bs1* @ *map* (λ *b*. (*b*,*i*)) *bs* @ *bs2*)
⟨*proof*⟩

lemma *square-free-factorization-factorI*: **fixes** *p* :: *'a poly*
assumes *sff*: *square-free-factorization p* (*c*, *bs1* @ (*a*,*i*) # *bs2*)
and *r*: *degree r* ≠ *0* **and** *s*: *degree s* ≠ *0*
and *a*: *a* = *r* ∗ *s*
shows *square-free-factorization p* (*c*, *bs1* @ ((*r*,*i*) # (*s*,*i*) # *bs2*))
⟨*proof*⟩

end

lemma *monic-square-free-irreducible-factorization*: **assumes** *mon*: *monic* (*f* :: *'b* :: *field poly*)
and *sf*: *square-free f*
shows ∃ *P*. *finite P* ∧ *f* = $\prod$ *P* ∧ *P* ⊆ {*q*. *irreducible q* ∧ *monic q*}
⟨*proof*⟩

context
assumes *SORT-CONSTRAINT*(*'a* :: {*field*, *factorial-ring-gcd*})
begin
lemma *monic-factorization-uniqueness*:
fixes *P*::*'a poly set*
assumes *finite-P*: *finite P*
and *PQ*: $\prod$ *P* = $\prod$ *Q*
and *P*: *P* ⊆ {*q*. $irreducible_d$ *q* ∧ *monic q*}
and *finite-Q*: *finite Q*
and *Q*: *Q* ⊆ {*q*. $irreducible_d$ *q* ∧ *monic q*}
shows *P* = *Q*
⟨*proof*⟩
end

11.2 Yun factorization and homomorphisms

locale *field-hom-0'* = *field-hom hom*
for *hom* :: *'a* :: {*field-char-0*,*field-gcd*} ⇒
'b :: {*field-char-0*,*field-gcd*}
begin
sublocale *field-hom'* ⟨*proof*⟩
end

lemma (**in** *field-hom-0'*) *yun-factorization-main-hom*:
defines *hp*: *hp* ≡ *map-poly hom*
defines *hpi*: *hpi* ≡ *map* (λ (*f*,*i*). (*hp f*, *i* :: *nat*))
assumes *monic*: *monic p* **and** *f*: *f* = *p 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)*
⟨*proof*⟩

lemma *square-free-square-free-factorization*:
square-free (p :: 'a :: {field,factorial-ring-gcd,semiring-gcd-mult-normalize} poly) ⟹
degree p ≠ 0 ⟹ *square-free-factorization p (1,[(p,1)])*
⟨*proof*⟩

lemma *constant-square-free-factorization*:
degree p = 0 ⟹ *square-free-factorization p (coeff p 0,[])*
⟨*proof*⟩

lemma (**in** *field-hom-0'*) *yun-monic-factorization*:
defines *hp*: *hp ≡ map-poly hom*
defines *hpi*: *hpi ≡ map (λ (f,i). (hp f, i :: nat))*
assumes *monic*: *monic f*
shows *yun-gcd.yun-monic-factorization gcd (hp f) = hpi (yun-gcd.yun-monic-factorization gcd f)*
⟨*proof*⟩

lemma (**in** *field-hom-0'*) *yun-factorization-hom*:
defines *hp*: *hp ≡ map-poly hom*
defines *hpi*: *hpi ≡ map (λ (f,i). (hp f, i :: nat))*
shows *yun-factorization gcd (hp f) = map-prod hom hpi (yun-factorization gcd f)*
⟨*proof*⟩

lemma (**in** *field-hom-0'*) *square-free-map-poly*:
square-free (map-poly hom f) = square-free f
⟨*proof*⟩

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* ⇒ *rat poly* ⇒ *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* ⇒ *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 ⇒ [] | *Some x* ⇒ *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 ⇒ [] | *Some x* ⇒ *x* # *roots-of-rat-poly-main* (*p div* [:−*x*,*1*:]))
⟨*proof*⟩

lemma *roots-of-rat-poly-main*: $p \neq 0 \Longrightarrow set\ (roots\text{-}of\text{-}rat\text{-}poly\text{-}main\ p) = \{x.\ poly\ p\ x = 0\}$
⟨*proof*⟩

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

definition *roots-of-rat-poly* :: *rat poly* ⇒ *rat list* **where**
roots-of-rat-poly p ≡ *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* ⇒ *bool* **where**
root-free p = $(degree\ p = 1 \lor (\forall\ x.\ poly\ p\ x \neq 0))$

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* ⇒ *rat list* ⇒ *rat poly list* ⇒ *rat* × *rat poly list* **where**
[*code*]: *factorize-root-free-main p xs fs* = (*case xs of Nil* ⇒
let l = *coeff p* (*degree p*); *q* = *smult* (*inverse l*) *p in* (*l*, (*if q* = *1 then fs else q* # *fs*))
| *x* # *xs* ⇒
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* ⇒ *rat* × *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* = *smult c* (*prod-list qs*)
$\bigwedge q.\ q \in set\ qs \Longrightarrow root\text{-}free\ q \land monic\ q \land degree\ q \neq 0$
⟨*proof*⟩

definition *rational-proper-factor* :: *rat poly* ⇒ *rat poly option* **where**
rational-proper-factor p = (*if degree p* ≤ *1 then None*
else if degree p = *2 then* (*case rat-roots2 p of Nil* ⇒ *None* | *Cons x xs* ⇒ *Some* [:−*x*,*1* :])

else if degree p = 3 then (case rational-root-test p of None ⇒ None | Some x ⇒ Some [:−x,1:])
else kronecker-factorization-rat p)

lemma *degree-1-dvd-root*: **assumes** *q*: *degree (q :: 'a :: field poly) = 1*
and *rt*: $\bigwedge$ *x. poly p x ≠ 0*
shows ¬ *q dvd p*
⟨*proof*⟩

lemma *rational-proper-factor*:
degree p > 0 ⟹ *rational-proper-factor p = None* ⟹ *irreducible*$_d$ *p*
rational-proper-factor p = Some q ⟹ *q dvd p* ∧ *degree q* ≥ *1* ∧ *degree q < degree p*
⟨*proof*⟩

function *factorize-rat-poly-main :: rat ⇒ rat poly list ⇒ rat poly list ⇒ rat × rat poly list* **where**
factorize-rat-poly-main c irr [] = (c,irr)
| *factorize-rat-poly-main c irr (p # ps) = (if degree p = 0*
then factorize-rat-poly-main (c ∗ coeff p 0) irr ps
else (case rational-proper-factor p of
None ⇒ factorize-rat-poly-main c (p # irr) ps
| *Some q ⇒ factorize-rat-poly-main c irr (q # p div q # ps)))*
⟨*proof*⟩

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

lemma *wf-factorize-rat-poly-main-wf-rel*: *wf factorize-rat-poly-main-wf-rel*
⟨*proof*⟩

lemma *factorize-rat-poly-main-wf-rel-sub*:
((a, b, ps), (c, d, p # ps)) ∈ factorize-rat-poly-main-wf-rel
⟨*proof*⟩

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)) ∈ factorize-rat-poly-main-wf-rel*
⟨*proof*⟩

termination
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ∈ *set qs* ⟹ $irreducible_d$ *q*
⟨*proof*⟩

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.