Polynomial Factorization*

René Thiemann and Akihisa Yamada

February 6, 2026

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 . . . 7

2 Preliminaries **39**
- 2.1 Missing Multiset . . . 39
- 2.2 Precomputation . . . 41
- 2.3 Order of Polynomial Roots . . . 43

3 Explicit Formulas for Roots **46**

4 Division of Polynomials over Integers **50**

5 More on Polynomials **57**

6 Gauss Lemma **64**

7 Prime Factorization **75**
- 7.1 Definitions . . . 75
- 7.2 Proofs . . . 77

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

8 Rational Root Test **92**

9 Kronecker Factorization **96**
9.1 Definitions 96
9.2 Code setup for divisors 97
9.3 Proofs 97

10 Polynomial Divisibility **105**
10.1 Fundamental Theorem of Algebra for Factorizations 106

11 Square Free Factorization **108**
11.1 Yun's factorization algorithm 113
11.2 Yun factorization and homomorphisms 138

12 GCD of rational polynomials via GCD for integer polynomials **140**

13 Rational Factorization **142**

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*)
  by (*induct xs*, *auto simp*: *set-Cons-def*)

lemma *sum-list-concat*: *sum-list* (*concat ls*) = *sum-list* (*map sum-list ls*)
  by (*induct ls*, *auto*)

lemma *listset*: *listset xs* = { *ys*. *length ys* = *length xs* ∧ (∀ *i* < *length xs*. *ys* ! *i* ∈ *xs* ! *i*)}
proof (*induct xs*)
  case (*Cons x xs*)
  let *?n* = *length xs*
  from *Cons*
  have *?case* = (*set-Cons x* {*ys*. *length ys* = *?n* ∧ (∀ *i* < *?n*. *ys* ! *i* ∈ *xs* ! *i*)} =
    {*ys*. *length ys* = *Suc ?n* ∧ *ys* ! *0* ∈ *x* ∧ (∀ *i* < *?n*. *ys* ! *Suc i* ∈ *xs* ! *i*)})
    (**is** - = (*?L* = *?R*))
    by (*auto simp*: *all-Suc-conv*)
  also have *?L* = *?R*
    by (*auto simp*: *set-Cons-def*, *case-tac xa*, *auto*)
  finally show *?case* **by** *simp*

[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.

qed *auto*

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

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*)

lemma *find-map-filter-Some*: *find-map-filter f p as* = *Some b* ⟹ *p b* ∧ *b* ∈ *f* ‘ *set as*
by (*induct f p as rule*: *find-map-filter.induct*, *auto simp*: *Let-def split*: *if-splits*)

lemma *find-map-filter-None*: *find-map-filter f p as* = *None* ⟹ ∀ *b* ∈ *f* ‘ *set as*. ¬ *p b*
by (*induct f p as rule*: *find-map-filter.induct*, *auto simp*: *Let-def split*: *if-splits*)

lemma *remdups-adj-sorted-distinct*[*simp*]: *sorted xs* ⟹ *distinct* (*remdups-adj xs*)
by (*induct xs rule*: *remdups-adj.induct*) (*auto*)

lemma *subseqs-length-simple*:
assumes *b* ∈ *set* (*subseqs xs*) **shows** *length b* ≤ *length xs*
using *assms* **by**(*induct xs arbitrary*:*b*;*auto simp*:*Let-def Suc-leD*)

lemma *subseqs-length-simple-False*:
assumes *b* ∈ *set* (*subseqs xs*) *length xs* < *length b* **shows** *False*
using *assms subseqs-length-simple* **by** *fastforce*

lemma *empty-subseqs*[*simp*]: [] ∈ *set* (*subseqs xs*) **by** (*induct xs*, *auto simp*: *Let-def*)

lemma *full-list-subseqs*: {*ys*. *ys* ∈ *set* (*subseqs xs*) ∧ *length ys* = *length xs*} = {*xs*}

proof (*induct xs*)
case (*Cons x xs*)
have *?case* = ({*ys* ∈ (#) *x* ‘ *set* (*subseqs xs*) ∪ *set* (*subseqs xs*).
length ys = *Suc* (*length xs*)} = (#) *x* ‘ {*xs*}) (**is** - = (*?l* = *?r*))
by (*auto simp*: *Let-def*)
also have *?l* = {*ys* ∈ (#) *x* ‘ *set* (*subseqs xs*). *length ys* = *Suc* (*length xs*)}
using *length-subseqs*[*of xs*]
using *subseqs-length-simple-False* **by** *force*
also have ... = (#) *x* ‘ {*ys* ∈ *set* (*subseqs xs*). *length ys* = *length xs*}
by *auto*
also have ... = (#) *x* ‘ {*xs*} **unfolding** *Cons* **by** *auto*
finally show *?case* **by** *simp*
qed *simp*

```
lemma nth-concat-split: assumes i < length (concat xs)
  shows ∃ j k. j < length xs ∧ k < length (xs ! j) ∧ concat xs ! i = xs ! j ! k
  using assms
proof (induct xs arbitrary: i)
  case (Cons x xs i)
  define I where I = i − length x
  show ?case
  proof (cases i < length x)
    case True note l = this
    hence i: concat (Cons x xs) ! i = x ! i by (auto simp: nth-append)
    show ?thesis unfolding i
      by (rule exI[of - 0], rule exI[of - i], insert Cons l, auto)
  next
    case False note l = this
    from l Cons(2) have i: i = length x + I I < length (concat xs) unfolding
I-def by auto
    hence iI: concat (Cons x xs) ! i = concat xs ! I by (auto simp: nth-append)
    from Cons(1)[OF i(2)] obtain j k where
      IH: j < length xs ∧ k < length (xs ! j) ∧ concat xs ! I = xs ! j ! k by auto
    show ?thesis unfolding iI
      by (rule exI[of - Suc j], rule exI[of - k], insert IH, auto)
  qed
qed simp

lemma nth-concat-diff: assumes i1 < length (concat xs) i2 < length (concat xs)
i1 ≠ i2
  shows ∃ j1 k1 j2 k2. (j1,k1) ≠ (j2,k2) ∧ j1 < length xs ∧ j2 < length xs
    ∧ k1 < length (xs ! j1) ∧ k2 < length (xs ! j2)
    ∧ concat xs ! i1 = xs ! j1 ! k1 ∧ concat xs ! i2 = xs ! j2 ! k2
  using assms
proof (induct xs arbitrary: i1 i2)
  case (Cons x xs)
  define I1 where I1 = i1 − length x
  define I2 where I2 = i2 − length x
  show ?case
  proof (cases i1 < length x)
    case True note l1 = this
    hence i1: concat (Cons x xs) ! i1 = x ! i1 by (auto simp: nth-append)
    show ?thesis
    proof (cases i2 < length x)
      case True note l2 = this
      hence i2: concat (Cons x xs) ! i2 = x ! i2 by (auto simp: nth-append)
      show ?thesis unfolding i1 i2
        by (rule exI[of - 0], rule exI[of - i1], rule exI[of - 0], rule exI[of - i2],
          insert Cons(4) l1 l2, auto)
    next
      case False note l2 = this
      from l2 Cons(3) have i22: i2 = length x + I2 I2 < length (concat xs)
```

unfolding *I2-def* **by** *auto*
    hence *i2*: *concat* (*Cons x xs*) ! *i2* = *concat xs* ! *I2* **by** (*auto simp*: *nth-append*)
    from *nth-concat-split*[*OF i22*(*2*)] **obtain** *j2 k2* **where**
      ∗: *j2* < *length xs* ∧ *k2* < *length* (*xs* ! *j2*) ∧ *concat xs* ! *I2* = *xs* ! *j2* ! *k2* **by** *auto*
    show *?thesis* **unfolding** *i1 i2*
      by (*rule exI*[*of - 0*], *rule exI*[*of - i1*], *rule exI*[*of - Suc j2*], *rule exI*[*of - k2*], *insert* ∗ *l1*, *auto*)
  qed
 next
  case *False* **note** *l1* = *this*
 from *l1 Cons*(*2*) **have** *i11*: *i1* = *length x* + *I1 I1* < *length* (*concat xs*) **unfolding** *I1-def* **by** *auto*
  hence *i1*: *concat* (*Cons x xs*) ! *i1* = *concat xs* ! *I1* **by** (*auto simp*: *nth-append*)
  show *?thesis*
  proof (*cases i2* < *length x*)
    case *False* **note** *l2* = *this*
     from *l2 Cons*(*3*) **have** *i22*: *i2* = *length x* + *I2 I2* < *length* (*concat xs*) **unfolding** *I2-def* **by** *auto*
    hence *i2*: *concat* (*Cons x xs*) ! *i2* = *concat xs* ! *I2* **by** (*auto simp*: *nth-append*)
    from *Cons*(*4*) *i11 i22* **have** *diff*: *I1* ≠ *I2* **by** *auto*
    from *Cons*(*1*)[*OF i11*(*2*) *i22*(*2*) *diff*] **obtain** *j1 k1 j2 k2*
      where *IH*: (*j1*,*k1*) ≠ (*j2*,*k2*) ∧ *j1* < *length xs* ∧ *j2* < *length xs*
      ∧ *k1* < *length* (*xs* ! *j1*) ∧ *k2* < *length* (*xs* ! *j2*)
      ∧ *concat xs* ! *I1* = *xs* ! *j1* ! *k1* ∧ *concat xs* ! *I2* = *xs* ! *j2* ! *k2* **by** *auto*
    show *?thesis* **unfolding** *i1 i2*
      by (*rule exI*[*of - Suc j1*], *rule exI*[*of - k1*], *rule exI*[*of - Suc j2*], *rule exI*[*of - k2*],
       insert IH, *auto*)
  next
    case *True* **note** *l2* = *this*
    hence *i2*: *concat* (*Cons x xs*) ! *i2* = *x* ! *i2* **by** (*auto simp*: *nth-append*)
    from *nth-concat-split*[*OF i11*(*2*)] **obtain** *j1 k1* **where**
      ∗: *j1* < *length xs* ∧ *k1* < *length* (*xs* ! *j1*) ∧ *concat xs* ! *I1* = *xs* ! *j1* ! *k1* **by** *auto*
    show *?thesis* **unfolding** *i1 i2*
      by (*rule exI*[*of - Suc j1*], *rule exI*[*of - k1*], *rule exI*[*of - 0*], *rule exI*[*of - i2*],
       insert ∗ *l2*, *auto*)
  qed
 qed
qed *auto*

lemma *list-all2-map-map*: (⋀ *x*. *x* ∈ *set xs* ⟹ *R* (*f x*) (*g x*)) ⟹ *list-all2 R* (*map f xs*) (*map g xs*)
 by (*induct xs*, *auto*)

lemma *in-set-idx*: *a* ∈ *set as* ⟹ ∃ *i*. *i* < *length as* ∧ *a* = *as* ! *i*
 unfolding *set-conv-nth* **by** *auto*

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* ⇒ *bool* **where**
is-partition cs ⟷ (∀ *j*<*length cs*. ∀ *i*<*j*. *cs* ! *i* ∩ *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 (*intro ext*)
fix *cs* :: *'a set list*
{
assume *is-partition-alt cs*
hence *is-partition cs* **unfolding** *is-partition-def is-partition-alt-def* **by** *auto*
}
moreover
{
assume *part*: *is-partition cs*
have *is-partition-alt cs* **unfolding** *is-partition-alt-def*
proof (*intro allI impI*)
fix *i j*
assume *i* < *length cs* ∧ *j* < *length cs* ∧ *i* ≠ *j*
with *part* **show** *cs* ! *i* ∩ *cs* ! *j* = {}
unfolding *is-partition-def*
by (*cases i* < *j*, *simp*, *cases j* < *i*, *force*, *simp*)
qed
}
ultimately
show *is-partition cs* = *is-partition-alt cs* **by** *auto*
qed

lemma *is-partition-Nil*:
is-partition [] = *True* **unfolding** *is-partition-def* **by** *auto*

lemma *is-partition-Cons*:
is-partition (*x*#*xs*) ⟷ *is-partition xs* ∧ *x* ∩ ⋃(*set xs*) = {} (**is** *?l* = *?r*)
proof
assume *?l*
have *one*: *is-partition xs*
proof (*unfold is-partition-def*, *intro allI impI*)
fix *j i* **assume** *j* < *length xs* **and** *i* < *j*
hence *Suc j* < *length*(*x*#*xs*) **and** *Suc i* < *Suc j* **by** *auto*
from ‹*?l*›[*unfolded is-partition-def*,*THEN spec*,*THEN mp*,*THEN spec*,*THEN mp*,*OF this*]
have (*x*#*xs*)!(*Suc i*) ∩ (*x*#*xs*)!(*Suc j*) = {} **.**

```
    thus xs!i ∩ xs!j = {} by simp
  qed
  have two: x ∩ ⋃(set xs) = {}
  proof (rule ccontr)
    assume x ∩ ⋃(set xs) ≠ {}
    then obtain y where y ∈ x and y ∈ ⋃(set xs) by auto
    then obtain z where z ∈ set xs and y ∈ z by auto
    then obtain i where i < length xs and xs!i = z using in-set-conv-nth[of z
xs] by auto
    with ‹y ∈ z› have y ∈ (x#xs)!Suc i by auto
    moreover with ‹y ∈ x› have y ∈ (x#xs)!0 by simp
    ultimately have (x#xs)!0 ∩ (x#xs)!Suc i ≠ {} by auto
    moreover from ‹i < length xs› have Suc i < length(x#xs) by simp
    ultimately show False using ‹?l›[unfolded is-partition-def] by best
  qed
  from one two show ?r ..
next
  assume ?r
  show ?l
  proof (unfold is-partition-def, intro allI impI)
    fix j i
    assume j: j < length (x # xs)
    assume i: i < j
    from i obtain j' where j': j = Suc j' by (cases j, auto)
    with j have j'len: j' < length xs and j'elem: (x # xs) ! j = xs ! j' by auto
    show (x # xs) ! i ∩ (x # xs) ! j = {}
    proof (cases i)
      case 0
      with j'elem have (x # xs) ! i ∩ (x # xs) ! j = x ∩ xs ! j' by auto
      also have ... ⊆ x ∩ ⋃(set xs) using j'len by force
      finally show ?thesis using ‹?r› by auto
    next
      case (Suc i')
      with i j' have i'j': i' < j' by auto
      from Suc j' have (x # xs) ! i ∩ (x # xs) ! j = xs ! i' ∩ xs ! j' by auto
      with ‹?r› i'j' j'len show ?thesis unfolding is-partition-def by auto
    qed
  qed
qed

lemma is-partition-sublist:
  assumes is-partition (us @ xs @ ys @ zs @ vs)
  shows is-partition (xs @ zs)
proof (rule ccontr)
  assume ¬ is-partition (xs @ zs)
  then obtain i j where j:j < length (xs @ zs) and i:i < j and ∗:(xs @ zs) ! i ∩
(xs @ zs) ! j ≠ {}
    unfolding is-partition-def by blast
  then show False
```

```
  proof (cases j < length xs)
    case True
    let ?m = j + length us
    let ?n = i + length us
    from True have ?m < length (us @ xs @ ys @ zs @ vs) by auto
    moreover from i have ?n < ?m by auto
    moreover have (us @ xs @ ys @ zs @ vs) ! ?n ∩ (us @ xs @ ys @ zs @ vs) !
?m ≠ {}
      using i True ∗ nth-append
    by (metis (no-types, lifting) add-diff-cancel-right' not-add-less2 order.strict-trans)
    ultimately show False using assms unfolding is-partition-def by auto
  next
    case False
    let ?m = j + length us + length ys
    from j have m:?m < length (us @ xs @ ys @ zs @ vs) by auto
   have mj:(us @ (xs @ ys @ zs @ vs)) ! ?m = (xs @ zs) ! j unfolding nth-append
using False j by auto
    show False
    proof (cases i < length xs)
      case True
      let ?n = i + length us
      from i have ?n < ?m by auto
     moreover have (us @ xs @ ys @ zs @ vs) ! ?n = (xs @ zs) ! i by (simp add:
True nth-append)
      ultimately show False using ∗ m assms mj unfolding is-partition-def by
blast
    next
      case False
      let ?n = i + length us + length ys
      from i have i:?n < ?m by auto
      moreover have (us @ xs @ ys @ zs @ vs) ! ?n = (xs @ zs) ! i
        unfolding nth-append using False i j less-diff-conv2 by auto
      ultimately show False using ∗ m assms mj unfolding is-partition-def by
blast
    qed
  qed
qed

lemma is-partition-inj-map:
  assumes is-partition xs
  and inj-on f (⋃x ∈ set xs. x)
  shows is-partition (map ((‘) f) xs)
proof (rule ccontr)
  assume ¬ is-partition (map ((‘) f) xs)
  then obtain i j where neq:i ≠ j
    and i:i < length (map ((‘) f) xs) and j:j < length (map ((‘) f) xs)
    and map ((‘) f) xs ! i ∩ map ((‘) f) xs ! j ≠ {}
    unfolding is-partition-alt is-partition-alt-def by auto
  then obtain x where x ∈ map ((‘) f) xs ! i and x ∈ map ((‘) f) xs ! j by auto
```

```
  then obtain y z where yi:y ∈ xs ! i and yx:f y = x and zj:z ∈ xs ! j and zx:f
z = x
    using i j by auto
  show False
  proof (cases y = z)
    case True
      with zj yi neq assms(1) i j show ?thesis by (auto simp: is-partition-alt
is-partition-alt-def)
  next
    case False
    have y ∈ (⋃ x ∈ set xs. x) using yi i by force
    moreover have z ∈ (⋃ x ∈ set xs. x) using zj j by force
   ultimately show ?thesis using assms(2) inj-on-def[of f (⋃ x∈set xs. x)] False
zx yx by blast
  qed
qed

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 −
  note [simp] = is-partition-Cons is-partition-Nil
  have ⋀ bs. (is-partition as = (is-partition-impl as ≠ None)) ∧
    (is-partition-impl as = Some bs ⟶ bs = ⋃ (set as))
  proof (induct as)
    case (Cons as rest bs)
    show ?case
    proof (cases is-partition rest)
      case False
      thus ?thesis using Cons by auto
    next
      case True
      with Cons obtain c where rest: is-partition-impl rest = Some c
        by (cases is-partition-impl rest, auto)
      with Cons True show ?thesis by auto
    qed
  qed auto
  thus ?thesis by blast
qed
end

lemma case-prod-partition:
```

case-prod f (*partition p xs*) = *f* (*filter p xs*) (*filter* (*Not* ∘ *p*) *xs*)
by *simp*

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

lemma *map-nth-conv*: *map f ss* = *map g ts* ⟹ ∀ *i* < *length ss*. *f*(*ss*!*i*) = *g*(*ts*!*i*)
proof (*intro allI impI*)
fix *i* **show** *map f ss* = *map g ts* ⟹ *i* < *length ss* ⟹ *f*(*ss*!*i*) = *g*(*ts*!*i*)
proof (*induct ss arbitrary*: *i ts*)
case *Nil* **thus** *?case* **by** (*induct ts*) *auto*
next
case (*Cons s ss*) **thus** *?case*
by (*induct ts*, *simp*, (*cases i*, *auto*))
qed
qed

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 −
from *id-take-nth-drop*[*OF len*] **have** *vs*[*symmetric*]: *vs* = *?xs* @ *vs*!*i* # *?ys* **.**
with *dist* **have** *distinct ?xs* **and** *distinct*(*vs*!*i*#*?ys*) **and** *set ?xs* ∩ *set*(*vs*!*i*#*?ys*) = {} **using** *distinct-append*[*of ?xs vs*!*i*#*?ys*] **by** *auto*
hence *distinct ?ys* **and** *set ?xs* ∩ *set ?ys* = {} **by** *auto*
with ‹*distinct ?xs*› **show** *?thesis* **using** *distinct-append*[*of ?xs ?ys*] *vs* **by** *simp*
qed

lemma *distinct-alt*:
assumes ∀ *x*. *length* (*filter* ((=) *x*) *xs*) ≤ *1*
shows *distinct xs*
using *assms* **proof**(*induct xs*)
case (*Cons x xs*)
then have *IH*:*distinct xs*
by (*metis dual-order.trans filter.simps*(*2*) *impossible-Cons nle-le*)
from *Cons*(*2*) **have** *length* (*filter* ((=) *x*) *xs*) = *0*
by (*metis* (*mono-tags*) *One-nat-def add.right-neutral add-Suc-right filter.simps*(*2*) *le-less length-0-conv less-Suc0 list.simps*(*3*) *list.size*(*4*) *nat.inject*)
then have *x* ∉ *set* (*xs*)
by (*metis* (*full-types*) *filter-empty-conv length-0-conv*)
with *IH* **show** *?case*
by *simp*
qed *simp*

lemma *distinct-filter2*:
assumes ∀ *i* < *size xs*. ∀ *j* < *size xs*. *i* ≠ *j* ∧ *f* (*xs*!*i*) ∧ *f* (*xs*!*j*) ⟶ *xs*!*i* ≠ *xs*!*j*
shows *distinct* (*filter f xs*)
using *assms* **proof**(*induct xs*)

case (*Cons x xs*)
{fix *i j* **assume** *i* < *length xs j* < *length xs i* ≠ *j f* (*xs*!*i*) *f* (*xs*!*j*)
with *Cons*(*2*) **have** *xs*!*i* ≠ *xs*!*j*
by (*metis not-less-eq nth-Cons-Suc Suc-inject length-Cons*)
}
with *Cons*(*1*) **have** *IH*:*distinct* (*filter f xs*)
by *presburger*
show *?case* **proof**(*cases f x*)
case *True*
with *Cons*(*2*) **have** ∀ *j*<*length xs*. *f* (*xs* ! *j*) ⟶ *x* ≠ *xs* ! *j* **by** *fastforce*
then have *x* ∉ *set* (*filter f xs*)
using *in-set-idx* [*of x xs*] **by** *auto*
then show *?thesis* **unfolding** *filter.simps* **using** *True IH* **by** *simp*
next
case *False*
then show *?thesis* **unfolding** *filter.simps* **using** *IH* **by** *presburger*
qed
qed *simp*

lemma *distinct-is-partition*:
assumes *distinct xs*
shows *is-partition* (*map* (λ*x*. {*x*}) *xs*)
using *assms* **proof**(*induct xs*)
case (*Cons x xs*)
then show *?case* **unfolding** *list.map*(*2*) *is-partition-Cons* **by** *force*
qed (*simp add*: *is-partition-Nil*)

lemma *is-partition-append*:
assumes *is-partition xs* **and** *is-partition zs*
and ∀ *i* < *length xs*. *xs*!*i* ∩ ⋃ (*set zs*) = {}
shows *is-partition* (*xs*@*zs*)
by (*smt* (*verit*, *del-insts*) *add-diff-inverse-nat assms*(*1*) *assms*(*2*) *assms*(*3*) *disjoint-iff is-partition-alt is-partition-alt-def length-append mem-simps*(*9*) *nat-add-left-cancel-less nth-append nth-mem*)

lemma *distinct-is-partitition-sets*:
assumes *distinct xs*
and *xs* = *concat ys*
shows *is-partition* (*map set ys*)
using *assms* **proof**(*induct ys arbitrary*:*xs*)
case (*Cons y ys*)
have *is-partition* (*map set ys*) **proof**−
from *Cons*(*2*,*3*) **have** *distinct* (*concat ys*)
unfolding *concat.simps* **by** *simp*
with *Cons*(*1*) **show** *?thesis* **by** *simp*
qed
moreover from *Cons*(*2*,*3*) **have** *set y* ∩ ⋃ (*set* (*map set ys*)) = {}
using *distinct-append*[*of y concat ys*]**by** *simp*
ultimately show *?case*

```
    unfolding is-partition-Cons list.map by simp
qed (simp add: is-partition-Nil)


lemma map-nth-eq-conv:
  assumes len: length xs = length ys
  shows (map f xs = ys) = (∀ i<length ys. f (xs ! i) = ys ! i) (is ?l = ?r)
proof −
  have (map f xs = ys) = (map f xs = map id ys) by auto
  also have ... = (∀ i < length ys. f (xs ! i) = id (ys ! i))
    using map-nth-conv[of f xs id ys] nth-map-conv[OF len, of f id] unfolding len
    by blast
  finally show ?thesis by auto
qed

lemma map-upt-len-conv:
  map (λ i . f (xs!i)) [0..<length xs] = map f xs
  by (rule nth-equalityI, auto)

lemma map-upt-add':
  map f [a..<a+b] = map (λ i. f (a + i)) [0..<b]
  by (induct b, auto)


definition generate-lists :: nat ⇒ 'a list ⇒ 'a list list
  where generate-lists n xs ≡ concat-lists (map (λ -. xs) [0 ..< n])

lemma set-generate-lists[simp]: set (generate-lists n xs) = {as. length as = n ∧
set as ⊆ set xs}
proof −
  {
    fix as
    have (length as = n ∧ (∀ i<n. as ! i ∈ set xs)) = (length as = n ∧ set as ⊆
set xs)
    proof −
      {
        assume length as = n
        hence n: n = length as by auto
        have (∀ i<n. as ! i ∈ set xs) = (set as ⊆ set xs) unfolding n
          unfolding all-set-conv-all-nth[of as λ x. x ∈ set xs, symmetric] by auto
      }
      thus ?thesis by auto
    qed
  }
  thus ?thesis unfolding generate-lists-def unfolding set-concat-lists by auto
qed

lemma nth-append-take:
  assumes i ≤ length xs shows (take i xs @ y#ys)!i = y
```

proof −
from *assms* **have** *a*: *length*(*take i xs*) = *i* **by** *simp*
have (*take i xs* @ *y*#*ys*)!(*length*(*take i xs*)) = *y* **by** (*rule nth-append-length*)
thus *?thesis* **unfolding** *a* **.**
qed

lemma *nth-append-take-is-nth-conv*:
assumes *i* < *j* **and** *j* ≤ *length xs* **shows** (*take j xs* @ *ys*)!*i* = *xs*!*i*
proof −
from *assms* **have** *i* < *length*(*take j xs*) **by** *simp*
hence (*take j xs* @ *ys*)!*i* = *take j xs* ! *i* **unfolding** *nth-append* **by** *simp*
thus *?thesis* **unfolding** *nth-take*[*OF assms*(*1*)] **.**
qed

lemma *nth-append-drop-is-nth-conv*:
assumes *j* < *i* **and** *j* ≤ *length xs* **and** *i* ≤ *length xs*
shows (*take j xs* @ *y* # *drop* (*Suc j*) *xs*)!*i* = *xs*!*i*
proof −
from ‹*j* < *i*› **obtain** *n* **where** *ij*: *Suc*(*j* + *n*) = *i* **using** *less-imp-Suc-add* **by** *auto*
with *assms* **have** *i*: *i* = *length*(*take j xs*) + *Suc n* **by** *auto*
have *len*: *Suc j* + *n* ≤ *length xs* **using** *assms i* **by** *auto*
have (*take j xs* @ *y* # *drop* (*Suc j*) *xs*)!*i* =
(*y* # *drop* (*Suc j*) *xs*)!(*i* − *length*(*take j xs*)) **unfolding** *nth-append i* **by** *auto*
also have ... = (*y* # *drop* (*Suc j*) *xs*)!(*Suc n*) **unfolding** *i* **by** *simp*
also have ... = (*drop* (*Suc j*) *xs*)!*n* **by** *simp*
finally show *?thesis* **using** *ij len* **by** *simp*
qed

lemma *nth-append-take-drop-is-nth-conv*:
assumes *i* ≤ *length xs* **and** *j* ≤ *length xs* **and** *i* ≠ *j*
shows (*take j xs* @ *y* # *drop* (*Suc j*) *xs*)!*i* = *xs*!*i*
proof −
from *assms* **have** *i* < *j* ∨ *i* > *j* **by** *auto*
thus *?thesis* **using** *assms*
by (*auto simp*: *nth-append-take-is-nth-conv nth-append-drop-is-nth-conv*)
qed

lemma *take-drop-imp-nth*: ⟦*take i ss* @ *x* # *drop* (*Suc i*) *ss* = *ss*⟧ ⟹ *x* = *ss*!*i*
proof (*induct ss arbitrary*: *i*)
case (*Cons s ss*)
from ‹*take i* (*s*#*ss*) @ *x* # *drop* (*Suc i*) (*s*#*ss*) = (*s*#*ss*)› **show** *?case*
proof (*induct i*)
case (*Suc i*)
from *Cons* **have** *IH*: *take i ss* @ *x* # *drop* (*Suc i*) *ss* = *ss* ⟹ *x* = *ss*!*i* **by** *auto*
from *Suc* **have** *take i ss* @ *x* # *drop* (*Suc i*) *ss* = *ss* **by** *auto*
with *IH* **show** *?case* **by** *auto*
qed *auto*
qed *auto*

lemma *take-drop-update-first*: **assumes** $j < length\ ds$ **and** $length\ cs = length\ ds$
shows $(take\ j\ ds\ @\ drop\ j\ cs)[j := ds\ !\ j] = take\ (Suc\ j)\ ds\ @\ drop\ (Suc\ j)\ cs$
using *assms*
proof (*induct j arbitrary*: *ds cs*)
case *0*
then obtain *d dds c ccs* **where** *ds*: $ds = d\ \#\ dds$ **and** *cs*: $cs = c\ \#\ ccs$ **by** (*cases ds*, *simp*, *cases cs*, *auto*)
show *?case* **unfolding** *ds cs* **by** *auto*
next
case (*Suc j*)
then obtain *d dds c ccs* **where** *ds*: $ds = d\ \#\ dds$ **and** *cs*: $cs = c\ \#\ ccs$ **by** (*cases ds*, *simp*, *cases cs*, *auto*)
from *Suc*(*1*)[*of dds ccs*] *Suc*(*2*) *Suc*(*3*) **show** *?case* **unfolding** *ds cs* **by** *auto*
qed

lemma *take-drop-update-second*: **assumes** $j < length\ ds$ **and** $length\ cs = length\ ds$
shows $(take\ j\ ds\ @\ drop\ j\ cs)[j := cs\ !\ j] = take\ j\ ds\ @\ drop\ j\ cs$
using *assms*
proof (*induct j arbitrary*: *ds cs*)
case *0*
then obtain *d dds c ccs* **where** *ds*: $ds = d\ \#\ dds$ **and** *cs*: $cs = c\ \#\ ccs$ **by** (*cases ds*, *simp*, *cases cs*, *auto*)
show *?case* **unfolding** *ds cs* **by** *auto*
next
case (*Suc j*)
then obtain *d dds c ccs* **where** *ds*: $ds = d\ \#\ dds$ **and** *cs*: $cs = c\ \#\ ccs$ **by** (*cases ds*, *simp*, *cases cs*, *auto*)
from *Suc*(*1*)[*of dds ccs*] *Suc*(*2*) *Suc*(*3*) **show** *?case* **unfolding** *ds cs* **by** *auto*
qed

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 (*induct xs ys rule*: *list-induct2'*)
case (*4 x xs y ys*)
have $take\ (length\ ys)\ xs = ys$
by (*rule 4*(*1*), *insert 4*(*2*−*3*), *auto*)
moreover from *4*(*3*) **have** $x = y$ **by** *auto*
ultimately show *?case* **by** *auto*
qed *auto*

lemma *take-upt-idx*:
assumes *i*: $i < length\ ls$
shows $take\ i\ ls = [\ ls\ !\ j\ .\ j \leftarrow [0..<i]]$
proof −
have *e*: $0 + i \leq i$ **by** *auto*

show *?thesis*
using *take-upt*[*OF e*] *take-map map-nth*
by (*metis* (*opaque-lifting*, *no-types*) *add.left-neutral i nat-less-le take-upt*)
qed

fun *distinct-eq* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool* **where**
distinct-eq - [] = True
| *distinct-eq eq* (*x # xs*) = ((∀ *y* ∈ *set xs*. ¬ (*eq y x*)) ∧ *distinct-eq eq xs*)

lemma *distinct-eq-append*: *distinct-eq eq* (*xs* @ *ys*) = (*distinct-eq eq xs* ∧ *distinct-eq eq ys* ∧ (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. ¬ (*eq y x*)))
by (*induct xs*, *auto*)

lemma *append-Cons-nth-left*:
assumes *i* < *length xs*
shows (*xs* @ *u # ys*) ! *i* = *xs* ! *i*
using *assms nth-append*[*of xs - i*] **by** *simp*

lemma *append-Cons-nth-middle*:
assumes *i* = *length xs*
shows (*xs* @ *y # zs*) ! *i* = *y*
using *assms* **by** *auto*

lemma *append-Cons-nth-right*:
assumes *i* > *length xs*
shows (*xs* @ *u # ys*) ! *i* = (*xs* @ *z # ys*) ! *i*
by (*simp add*: *assms nth-append*)

lemma *append-Cons-nth-not-middle*:
assumes *i* ≠ *length xs*
shows (*xs* @ *u # ys*) ! *i* = (*xs* @ *z # ys*) ! *i*
by (*metis assms list-update-length nth-list-update-neq*)

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

lemma *concat-all-nth*:
assumes *length xs* = *length ys*
and ⋀*i*. *i* < *length xs* ⟹ *length* (*xs* ! *i*) = *length* (*ys* ! *i*)
and ⋀*i j*. *i* < *length xs* ⟹ *j* < *length* (*xs* ! *i*) ⟹ *P* (*xs* ! *i* ! *j*) (*ys* ! *i* ! *j*)
shows ∀ *k*<*length* (*concat xs*). *P* (*concat xs* ! *k*) (*concat ys* ! *k*)
using *assms*
proof (*induct xs ys rule*: *list-induct2*)
case (*Cons x xs y ys*)
from *Cons*(*3*)[*of 0*] **have** *xy*: *length x* = *length y* **by** *simp*
from *Cons*(*4*)[*of 0*] *xy* **have** *pxy*: ⋀ *j*. *j* < *length x* ⟹ *P* (*x* ! *j*) (*y* ! *j*) **by** *auto*
{
fix *i*
assume *i*: *i* < *length xs*

```
    with Cons(3)[of Suc i]
    have len: length (xs ! i) = length (ys ! i) by simp
    from Cons(4)[of Suc i] i have ⋀ j. j < length (xs ! i) ⟹ P (xs ! i ! j) (ys !
i ! j)
      by auto
    note len and this
  }
  from Cons(2)[OF this] have ind: ⋀ k. k < length (concat xs) ⟹ P (concat xs
! k) (concat ys ! k)
    by auto
  show ?case unfolding concat.simps
  proof (intro allI impI)
    fix k
    assume k: k < length (x @ concat xs)
    show P ((x @ concat xs) ! k) ((y @ concat ys) ! k)
    proof (cases k < length x)
      case True
      show ?thesis unfolding nth-append using True xy pxy[OF True]
        by simp
    next
      case False
      with k have k − (length x) < length (concat xs) by auto
      then obtain n where n: k − length x = n and nxs: n < length (concat xs)
by auto
      show ?thesis unfolding nth-append n n[unfolded xy] using False xy ind[OF
nxs]
        by auto
    qed
  qed
qed auto

lemma eq-length-concat-nth:
  assumes length xs = length ys
    and ⋀i. i < length xs ⟹ length (xs ! i) = length (ys ! i)
  shows length (concat xs) = length (concat ys)
using assms
proof (induct xs ys rule: list-induct2)
  case (Cons x xs y ys)
  from Cons(3)[of 0] have xy: length x = length y by simp
  {
    fix i
    assume i < length xs
    with Cons(3)[of Suc i]
    have length (xs ! i) = length (ys ! i) by simp
  }
  from Cons(2)[OF this] have ind: length (concat xs) = length (concat ys) by
simp
  show ?case using xy ind by auto
qed auto
```

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 (*induct xs*)
  case (*Cons x xs*) **thus** *?case* **by** (*cases x* ∈ *set* (*list-union xs ys*)) (*auto*)
qed *simp*

declare *list-union.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 (*induct xs*)
  case (*Cons x xs*) **thus** *?case* **by** (*cases x* ∈ *set ys*) (*auto*)
qed *simp*

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

lemma *nth-drop-0*: *0* < *length ss* ⟹ (*ss*!*0*)#*drop* (*Suc 0*) *ss* = *ss*
  by (*simp add*: *Cons-nth-drop-Suc*)

lemma *set-foldr-remdups-set-map-conv*[*simp*]:
  set (*foldr* (λ*x xs*. *remdups* (*f x* @ *xs*)) *xs* []) = ⋃(*set* (*map* (*set* ∘ *f*) *xs*))
  by (*induct xs*) *auto*

lemma *subset-set-code*[*code-unfold*]: *set xs* ⊆ *set ys* ⟷ *list-all* (λ*x*. *x* ∈ *set ys*) *xs*
  unfolding *list-all-iff* **by** *auto*

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*

by (*induct xs ys rule*: *union-list-sorted.induct*, *auto*)

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* ⟹ *sorted ys* ⟹
set (*subtract-list-sorted xs ys*) = *set xs* − *set ys*
proof (*induct xs ys rule*: *subtract-list-sorted.induct*)
case (*1 x xs y ys*)
have *xxs*: *sorted* (*x* # *xs*) **by** *fact*
have *yys*: *sorted* (*y* # *ys*) **by** *fact*
have *xs*: *sorted xs* **using** *xxs* **by** (*simp*)
show *?case*
proof (*cases x* = *y*)
case *True*
thus *?thesis* **using** *1*(*1*)[*OF True xs yys*] **by** *auto*
next
case *False* **note** *neq* = *this*
note *IH* = *1*(*2*−*3*)[*OF this*]
show *?thesis*
by (*cases x* < *y*, *insert IH xxs yys False*, *auto*)
qed
qed *auto*

lemma *subset-subtract-listed-sorted*: *set* (*subtract-list-sorted xs ys*) ⊆ *set xs*
by (*induct xs ys rule*: *subtract-list-sorted.induct*, *auto*)

lemma *set-subtract-list-distinct*[*simp*]: *distinct xs* ⟹ *distinct* (*subtract-list-sorted xs ys*)
by (*induct xs ys rule*: *subtract-list-sorted.induct*, *insert subset-subtract-listed-sorted*, *auto*)

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*)
by (*simp-all add*: *remdups-sort-def*)

maximum and minimum

lemma *max-list-mono*: **assumes** ⋀ *x*. *x* ∈ *set xs* − *set ys* ⟹ ∃ *y*. *y* ∈ *set ys* ∧ *x* ≤ *y*
shows *max-list xs* ≤ *max-list ys*
using *assms*

```
proof (induct xs)
  case (Cons x xs)
  have x ≤ max-list ys
  proof (cases x ∈ set ys)
    case True
    from max-list[OF this] show ?thesis .
  next
    case False
    with Cons(2)[of x] obtain y where y: y ∈ set ys
      and xy: x ≤ y by auto
    from xy max-list[OF y] show ?thesis by arith
  qed
  moreover have max-list xs ≤ max-list ys
    by (rule Cons(1)[OF Cons(2)], auto)
  ultimately show ?case by auto
qed auto

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 (induct xs)
  case oCons : (Cons y ys)
  show ?case
  proof (cases ys)
    case Nil
    thus ?thesis using oCons by auto
  next
    case (Cons z zs)
    hence min-list (y # ys) = min y (min-list ys)
      by auto
    then show ?thesis
      using min-le-iff-disj oCons.hyps oCons.prems by auto
  qed
qed simp

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)
  by (metis min-list.simps len length-greater-0-conv min.mono nth-drop-0 xsys xy)

lemma min-list-nth:
  assumes length xs = length ys
    and ⋀i. i < length ys ⟹ xs ! i ≤ ys ! i
  shows min-list xs ≤ min-list ys
using assms
```

```
proof (induct xs arbitrary: ys)
  case (Cons x xs zs)
  from Cons(2) obtain y ys where zs: zs = y # ys by (cases zs, auto)
  note Cons = Cons[unfolded zs]
  from Cons(2) have len: length xs = length ys by simp
  from Cons(3)[of 0] have xy: x ≤ y by simp
  {
    fix i
    assume i < length xs
    with Cons(3)[of Suc i] Cons(2)
    have xs ! i ≤ ys ! i by simp
  }
  from Cons(1)[OF len this] Cons(2) have ind: min-list xs ≤ min-list ys by simp
  show ?case unfolding zs
    by (rule min-list-Cons[OF xy len ind])
qed auto

lemma min-list-ex:
  assumes xs ≠ [] shows ∃ x∈set xs. min-list xs = x
  using assms
proof (induct xs)
  case oCons : (Cons x xs)
  show ?case
  proof (cases xs)
    case (Cons y ys)
    hence id: min-list (x # xs) = min x (min-list xs) and nNil: xs ≠ [] by auto
    show ?thesis
    proof (cases x ≤ min-list xs)
      case True
      show ?thesis unfolding id
        by (rule bexI[of - x], insert True, auto simp: min-def)
    next
      case False
      show ?thesis unfolding id min-def
        using oCons(1)[OF nNil] False by auto
    qed
  qed auto
qed auto

lemma min-list-subset:
  assumes subset: set ys ⊆ set xs and mem: min-list xs ∈ set ys
  shows min-list xs = min-list ys
  by (metis antisym empty-iff empty-set mem min-list min-list-ex subset subsetD)
```

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 (λn. 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*)
using *assms* **proof** (*induct as arbitrary*: *i f bs*)
case (*Cons x xs*)
show *?case*
proof (*cases i*)
case (*Suc j*)
with *Cons*(*2*) **have** *j* < *length xs* **by** *simp*
from *Cons*(*1*)[*OF this*] **and** *Suc* **show** *?thesis* **by** *simp*
qed *simp*
qed *simp*

lemma *permut-sound*:
assumes *i* < *length as*
shows *permut as f* ! *i* = *as* ! (*f i*)
using *assms* **and** *permut-aux-sound* **by** *simp*

lemma *permut-aux-length*:
assumes *bij-betw f* {..<*length as*} {..<*length bs*}
shows *length* (*permut-aux as f bs*) = *length as*
by (*induct as arbitrary*: *f bs*, *simp-all*)

lemma *permut-length*:
assumes *bij-betw f* {..< *length as*} {..< *length as*}
shows *length* (*permut as f*) = *length as*
using *permut-aux-length*[*OF assms*] **by** *simp*

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*
using *assms*[*symmetric*] **by** (*induct zs arbitrary*: *y*) *simp-all*

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*
using *assms* **by** (*induct xs*) *simp-all*

lemma *foldl-foldr-o-id*:
foldl (∘) *id fs* = *foldr* (∘) *fs id*
proof (*induct fs*)
case (*Cons f fs*)

```
  have id ∘ f = f ∘ id by simp
  with Cons [symmetric] show ?case
    by (simp only: foldl-Cons foldr-Cons o-apply [of - - id] foldl-assoc o-assoc)
qed simp

lemma foldr-o-o-id[simp]:
  foldr ((∘) ∘ f) xs id a = foldr f xs a
  by (induct xs) simp-all

lemma Ex-list-of-length-P:
  assumes ∀ i<n. ∃ x. P x i
  shows ∃ xs. length xs = n ∧ (∀ i<n. P (xs ! i) i)
proof −
  from assms have ∀ i. ∃ x. i < n ⟶ P x i by simp
  from choice[OF this] obtain xs where xs: ⋀ i. i < n ⟹ P (xs i) i by auto
  show ?thesis
    by (rule exI[of - map xs [0 ..< n]], insert xs, auto)
qed

lemma ex-set-conv-ex-nth: (∃ x∈set xs. P x) = (∃ i<length xs. P (xs ! i))
  using in-set-conv-nth[of - xs] by force

lemma map-eq-set-zipD [dest]:
  assumes map f xs = map f ys
    and (x, y) ∈ set (zip xs ys)
  shows f x = f y
using assms
proof (induct xs arbitrary: ys)
  case (Cons x xs)
  then show ?case by (cases ys) auto
qed simp

fun span :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list × '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)
  by (induct xs, auto)

declare span.simps[simp del]

lemma parallel-list-update: assumes
  one-update: ⋀ xs i y. length xs = n ⟹ i < n ⟹ r (xs ! i) y ⟹ p xs ⟹ p
(xs[i := y])
  and init: length xs = n p xs
  and rel: length ys = n ⋀ i. i < n ⟹ r (xs ! i) (ys ! i)
  shows p ys
```

```
proof −
  note len = rel(1) init(1)
  {
    fix i
    assume i ≤ n
    hence p (take i ys @ drop i xs)
    proof (induct i)
      case 0 with init show ?case by simp
    next
      case (Suc i)
      hence IH: p (take i ys @ drop i xs) by simp
      from Suc have i: i < n by simp
      let ?xs = (take i ys @ drop i xs)
      have length ?xs = n using i len by simp
      from one-update[OF this i - IH, of ys ! i] rel(2)[OF i] i len
      show ?case by (simp add: nth-append take-drop-update-first)
    qed
  }
  from this[of n] show ?thesis using len by auto
qed

lemma nth-concat-two-lists:
  i < length (concat (xs :: 'a list list)) ⟹ length (ys :: 'b list list) = length xs
  ⟹ (⋀ i. i < length xs ⟹ length (ys ! i) = length (xs ! i))
  ⟹ ∃ j k. j < length xs ∧ k < length (xs ! j) ∧ (concat xs) ! i = xs ! j ! k ∧
     (concat ys) ! i = ys ! j ! k
proof (induct xs arbitrary: i ys)
  case (Cons x xs i yys)
  then obtain y ys where yys: yys = y # ys by (cases yys, auto)
  note Cons = Cons[unfolded yys]
  from Cons(4)[of 0] have [simp]: length y = length x by simp
  show ?case
  proof (cases i < length x)
    case True
    show ?thesis unfolding yys
        by (rule exI[of - 0], rule exI[of - i], insert True Cons(2−4), auto simp:
nth-append)
  next
    case False
    let ?i = i − length x
    from False Cons(2−3) have ?i < length (concat xs) length ys = length xs by
auto
    note IH = Cons(1)[OF this]
    {
      fix i
      assume i < length xs
      with Cons(4)[of Suc i] have length (ys ! i) = length (xs ! i) by simp
    }
    from IH[OF this]
```

```
    obtain j k where IH1: j < length xs k < length (xs ! j)
      concat xs ! ?i = xs ! j ! k
      concat ys ! ?i = ys ! j ! k by auto
    show ?thesis unfolding yys
     by (rule exI[of - Suc j], rule exI[of - k], insert IH1 False, auto simp: nth-append)
  qed
qed simp
```

Removing duplicates w.r.t. some function.

```
fun remdups-gen :: ('a ⇒ 'b) ⇒ 'a list ⇒ 'a list where
  remdups-gen f [] = []
| remdups-gen f (x # xs) = x # remdups-gen f [y <− xs. ¬ f x = f y]

lemma remdups-gen-subset: set (remdups-gen f xs) ⊆ set xs
  by (induct f xs rule: remdups-gen.induct, auto)

lemma remdups-gen-elem-imp-elem: x ∈ set (remdups-gen f xs) ⟹ x ∈ set xs
  using remdups-gen-subset[of f xs] by blast

lemma elem-imp-remdups-gen-elem: x ∈ set xs ⟹ ∃ y ∈ set (remdups-gen f xs).
f x = f y
proof (induct f xs rule: remdups-gen.induct)
  case (2 f z zs)
  show ?case
  proof (cases f x = f z)
    case False
    with 2(2) have x ∈ set [y←zs . f z ≠ f y] by auto
    from 2(1)[OF this] show ?thesis by auto
  qed auto
qed auto


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)
using assms(1, 2)
proof (induct xss arbitrary: i rule: List.rev-induct)
  case (snoc xs xss)
  then show ?case using assms by (cases i < length xss) (auto simp: nth-append)
qed simp

lemma concat-map-empty [simp]:
  concat (map (λ-. []) xs) = []
  by simp
```

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*
unfolding *assms* [*symmetric*] **by** (*rule map-upt-len-conv*)

lemma *concat-map-concat* [*simp*]:
concat (*map concat xs*) = *concat* (*concat xs*)
by (*induct xs*) *simp-all*

lemma *concat-concat-map*:
concat (*concat* (*map f xs*)) = *concat* (*map* (*concat* ∘ *f*) *xs*)
by (*induct xs*) *simp-all*

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

lemma *Ball-at-Least0LessThan-conv* [*simp*]:
length xs = *n* ⟹
(∀ *i* ∈ {*0* ..< *n*}. *P* (*xs* ! *i*)) ⟷ (∀ *x* ∈ *set xs*. *P x*)
by (*metis atLeast0LessThan in-set-conv-nth lessThan-iff*)

lemma *sum-list-replicate-length* [*simp*]:
sum-list (*replicate* (*length xs*) (*Suc 0*)) = *length xs*
by (*induct xs*) *simp-all*

lemma *list-all2-in-set2*:
assumes *list-all2 P xs ys* **and** *y* ∈ *set ys*
obtains *x* **where** *x* ∈ *set xs* **and** *P x y*
using *assms* **by** (*induct*) *auto*

lemma *map-eq-conv'*:
map f xs = *map g ys* ⟷ *length xs* = *length ys* ∧ (∀ *i* < *length xs*. *f* (*xs* ! *i*) = *g* (*ys* ! *i*))
using *map-equality-iff map-equality-iff nth-map-conv* **by** *auto*

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*
using *assms* **by** (*rule remdups-adj.cases*)

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*
using *assms* **by** (*cases xs*; *cases tl xs*; *cases tl* (*tl xs*), *auto*)

lemma *foldr-append2* [*simp*]:
foldr ((@) ∘ *f*) *xs* (*ys* @ *zs*) = *foldr* ((@) ∘ *f*) *xs ys* @ *zs*
by (*induct xs*) *simp-all*

lemma *foldr-append2-Nil* [*simp*]:
foldr ((@) ∘ *f*) *xs* [] @ *zs* = *foldr* ((@) ∘ *f*) *xs zs*
unfolding *foldr-append2* [*symmetric*] **by** *simp*

lemma *UNION-set-zip*:
(⋃ *x* ∈ *set* (*zip* [*0*..<*length xs*] (*map f xs*)). *g x*) = (⋃ *i* < *length xs*. *g* (*i*, *f* (*xs* ! *i*)))
by (*auto simp*: *set-conv-nth*)

lemma *zip-fst*: *p* ∈ *set* (*zip as bs*) ⟹ *fst p* ∈ *set as*
by (*metis in-set-zipE prod.collapse*)

lemma *zip-snd*: *p* ∈ *set* (*zip as bs*) ⟹ *snd p* ∈ *set bs*
by (*metis in-set-zipE prod.collapse*)

lemma *zip-size-aux*: *size-list* (*size o snd*) (*zip ts ls*) ≤ (*size-list size ls*)
proof (*induct ls arbitrary*: *ts*)
case (*Cons l ls ts*)
thus *?case* **by** (*cases ts*, *auto*)
qed *auto*

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 −
show *?thesis* **unfolding** *arg-cong*[**where** *f* = *length*, *OF id-take-nth-drop*[*OF i*]]
unfolding *remove-nth-def* **by** *simp*
qed

lemma *remove-nth-length* :
assumes *n-bd*: *n* < *length xs*
shows *length* (*remove-nth n xs*) = *length xs* − *1*
using *n-bd* **by** *force*

lemma *remove-nth-id* : *length xs* ≤ *n* ⟹ *remove-nth n xs* = *xs*
by *simp*

lemma *remove-nth-sound-l* :
assumes *p-ub*: *p* < *n*
shows (*remove-nth n xs*) ! *p* = *xs* ! *p*
proof (*cases n* < *length xs*)
case *True*
from *length-take* **and** *True* **have** *ltk*: *length* (*take n xs*) = *n* **by** *simp*
{
assume *pltn*: *p* < *n*
from *this* **and** *ltk* **have** *plttk*: *p* < *length* (*take n xs*) **by** *simp*
with *nth-append*[*of take n xs - p*]
have ((*take n xs*) @ (*drop* (*Suc n*) *xs*)) ! *p* = *take n xs* ! *p* **by** *auto*
with *pltn* **and** *nth-take* **have** ((*take n xs*) @ (*drop* (*Suc n*) *xs*)) ! *p* = *xs* ! *p*
by *simp*
}
from *this* **and** *ltk* **and** *p-ub* **show** *?thesis* **by** *simp*
next
case *False*
hence *length xs* ≤ *n* **by** *arith*
with *remove-nth-id* **show** *?thesis* **by** *force*
qed

lemma *remove-nth-sound-r* :
assumes *n* ≤ *p* **and** *p* < *length xs*
shows (*remove-nth n xs*) ! *p* = *xs* ! (*Suc p*)
proof−
from ‹*n* ≤ *p*› **and** ‹*p* < *length xs*› **have** *n-ub*: *n* < *length xs* **by** *arith*
from *length-take* **and** *n-ub* **have** *ltk*: *length* (*take n xs*) = *n* **by** *simp*
from ‹*n* ≤ *p*› **and** *ltk* **and** *nth-append*[*of take n xs - p*]
have *Hrew*: ((*take n xs*) @ (*drop* (*Suc n*) *xs*)) ! *p* = *drop* (*Suc n*) *xs* ! (*p* − *n*) **by** *auto*
from ‹*n* ≤ *p*› **have** *idx*: *Suc n* + (*p* − *n*) = *Suc p* **by** *arith*
from ‹*p* < *length xs*› **have** *Sp-ub*: *Suc p* ≤ *length xs* **by** *arith*
from *idx* **and** *Sp-ub* **and** *nth-drop* **have** *Hrew′*: *drop* (*Suc n*) *xs* ! (*p* − *n*) = *xs* ! (*Suc p*) **by** *simp*
from *Hrew* **and** *Hrew′* **show** *?thesis* **by** *simp*
qed

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*)
using *assms remove-nth-sound-l remove-nth-sound-r*[*of n i xs*] **by** *auto*

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 (cases p < length as)
 case True
  hence p-ub: p < length as by assumption
  with remove-nth-length have lr-ub: length (remove-nth p as) = length as − 1 by
auto
  {
   fix i assume i-ub: i < length (remove-nth p as)
   have P (remove-nth p as ! i) (remove-nth p bs ! i)
    proof (cases i < p)
    case True
     from i-ub and lr-ub have i-ub2: i < length as by arith
     from i-ub2 and Pab have P: P (as ! i) (bs ! i) by blast
     from P and remove-nth-sound-l[OF True, of as] and remove-nth-sound-l[OF
True, of bs]
      show ?thesis by simp
     next
     case False
      hence p-ub2: p ≤ i by arith
      from i-ub and lr-ub have Si-ub: Suc i < length as by arith
      with Pab have P: P (as ! Suc i) (bs ! Suc i) by blast
      from i-ub and lr-ub have i-uba: i < length as by arith
      from i-uba and aslbs have i-ubb: i < length bs by simp
      from P and p-ub and aslbs and remove-nth-sound-r[OF p-ub2 i-uba]
       and remove-nth-sound-r[OF p-ub2 i-ubb]
       show ?thesis by auto
     qed
  }
  thus ?thesis by simp
 next
 case False
  hence p-lba: length as ≤ p by arith
  with aslbs have p-lbb: length bs ≤ p by simp
  from remove-nth-id[OF p-lba] and remove-nth-id[OF p-lbb] and Pab
  show ?thesis by simp
qed

declare remove-nth-def[simp del]

lemma concat-nth:
  assumes m < length xs and n < length (xs ! m)
    and i = sum-list (map length (take m xs)) + n
  shows concat xs ! i = xs ! m ! n
  using assms
proof (induct xs arbitrary: m n i)
  case (Cons x xs)
  show ?case
  proof (cases m)
    case 0
```

```
    then show ?thesis using Cons by (simp add: nth-append)
  next
    case (Suc k)
    with Cons(1) [of k n i − length x] and Cons(2−)
    show ?thesis by (simp-all add: nth-append)
  qed
qed simp

lemma concat-nth-length:
  i < length uss ⟹ j < length (uss ! i) ⟹
    sum-list (map length (take i uss)) + j < length (concat uss)
proof (induct uss arbitrary: i j)
  case (Cons u uss i j)
  thus ?case by (cases i, auto)
qed auto

lemma less-length-concat:
  assumes i < length (concat xs)
  shows ∃ m n.
    i = sum-list (map length (take m xs)) + n ∧
    m < length xs ∧ n < length (xs ! m) ∧ concat xs ! i = xs ! m ! n
  using assms
proof (induct xs arbitrary: i rule: length-induct)
  case (1 xs)
  then show ?case
  proof (cases xs)
    case (Cons y ys)
    note [simp] = this
    { assume ∗: i < length y
      with 1 obtain n where i = n and n < length y
        and y ! i = y ! n by simp
      then have i = sum-list (map length (take 0 xs)) + n
        and 0 < length xs and n < length (xs ! 0)
        and concat xs ! i = xs ! 0 ! n
        using ∗ by (auto simp: nth-append)
      then have ?thesis by blast }
    moreover
    { assume ∗: i ≥ length y
      define j where j = i − length y
      then have length ys < length xs j < length (concat ys)
        using ∗ and 1.prems by auto
      with 1 obtain m n where j = sum-list (map length (take m ys)) + n
        and m < length ys and n < length (ys ! m)
        and concat ys ! j = ys ! m ! n by blast
      then have i = sum-list (map length (take (Suc m) xs)) + n
        and Suc m < length xs and n < length (xs ! Suc m)
        and concat xs ! i = xs ! Suc m ! n
        using ∗ by (simp-all add: j-def nth-append)
      then have ?thesis by blast }
```

```
    ultimately show ?thesis by force
  qed simp
qed

lemma concat-remove-nth:
  assumes i < length sss
    and j < length (sss ! i)
  defines k ≡ sum-list (map length (take i sss)) + j
  shows concat (take i sss @ remove-nth j (sss ! i) # drop (Suc i) sss) = remove-nth
k (concat sss)
  using assms
  unfolding remove-nth-def
proof (induct sss rule: List.rev-induct)
  case Nil then show ?case by auto
next
  case (snoc ss sss)
  then have i = length sss ∨ i < length sss by auto
  then show ?case
  proof
    assume i:i = length sss
   have sum-list (map length sss) = length (concat sss) by (simp add: length-concat)
    with snoc i show ?thesis by simp
  next
    assume i:i < length sss
    then have nth:(sss @ [ss]) ! i = sss ! i by (simp add: nth-append)
    from i have drop:drop (Suc i) (sss @ [ss]) = drop (Suc i) sss @ [ss] by auto
    from i have take:take i (sss @ [ss]) = take i sss by auto
    from snoc(1)[OF i] snoc(2−) have 1:concat (take i (sss @ [ss]) @
      (take j ((sss @ [ss]) ! i) @ drop (Suc j) ((sss @ [ss]) ! i)) # drop (Suc i) (sss
@ [ss])) =
      take k (concat sss) @ drop (Suc k) (concat sss) @ ss unfolding take nth drop
by simp
    from snoc(4) take have k:k = sum-list (map length (take i sss)) + j by auto
    from nth snoc(3) have j: j < length (sss ! i) by auto
    have takek:take (sum-list (map length (take i sss)) + j) (concat (sss @ [ss])) =

      take (sum-list (map length (take i sss)) + j) (concat sss)
      using concat-nth-length[OF i j] by auto
    have dropk:drop (Suc (sum-list (map length (take i sss)) + j)) (concat sss) @
ss =
      drop (Suc (sum-list (map length (take i sss)) + j)) (concat (sss @ [ss]))
      using concat-nth-length[OF i j] by auto
    have take k (concat sss) @ drop (Suc k) (concat sss) @ ss =
      take k (concat (sss @ [ss])) @ drop (Suc k) (concat (sss @ [ss]))
      unfolding k takek dropk ..
    with 1 show ?thesis by auto
  qed
qed
```

lemma *nth-append-Cons*: $(xs \mathbin{@} y \# zs) ! i =$
$(\mathit{if}\ i < \mathit{length}\ xs\ \mathit{then}\ xs ! i\ \mathit{else}\ \mathit{if}\ i = \mathit{length}\ xs\ \mathit{then}\ y\ \mathit{else}\ zs ! (i - \mathit{Suc}\ (\mathit{length}\ xs)))$
by (*cases i length xs rule*: *linorder-cases*, *auto simp*: *nth-append*)

lemma *sum-list-take-eq*:
fixes *xs* :: *nat list*
shows $k < i \Longrightarrow i < \mathit{length}\ xs \Longrightarrow \mathit{sum\text{-}list}\ (\mathit{take}\ i\ xs) =$
$\mathit{sum\text{-}list}\ (\mathit{take}\ k\ xs) + xs ! k + \mathit{sum\text{-}list}\ (\mathit{take}\ (i - \mathit{Suc}\ k)\ (\mathit{drop}\ (\mathit{Suc}\ k)\ xs))$
by (*subst id-take-nth-drop* [*of k*]) (*auto simp*: *min-def drop-take*)

lemma *nth-equalityE*:
$xs = ys \Longrightarrow (\mathit{length}\ xs = \mathit{length}\ ys \Longrightarrow (\bigwedge i.\ i < \mathit{length}\ xs \Longrightarrow xs ! i = ys ! i) \Longrightarrow P) \Longrightarrow P$
by *simp*

lemma *not-Nil-imp-last*: $xs \neq [] \Longrightarrow \exists\, ys\ y.\ xs = ys @ [y]$
by (*meson rev-exhaust*)

lemma *Nil-or-last*: $xs = [] \vee (\exists\, ys\ y.\ xs = ys @ [y])$
using *not-Nil-imp-last* **by** *blast*

fun *fold-map* :: $('a \Rightarrow 'b \Rightarrow 'c \times 'b) \Rightarrow 'a\ list \Rightarrow 'b \Rightarrow 'c\ list \times 'b$ **where**
fold-map f [] *y* = ([], *y*)
| *fold-map f* (*x*#*xs*) *y* = (*case f x y of*
(x', y') ⇒ *case fold-map f xs* y' *of*
(xs', y'') ⇒ ($x' \#\ xs'$, y''))

lemma *fold-map-cong* [*fundef-cong*]:
assumes $a = b$ **and** $xs = ys$
and $\bigwedge x.\ x \in \mathit{set}\ xs \Longrightarrow f\ x = g\ x$
shows *fold-map f xs a* = *fold-map g ys b*
using *assms* **by** (*induct ys arbitrary*: *a b xs*) *simp-all*

lemma *fold-map-map-conv*:
assumes $\bigwedge x\ ys.\ x \in \mathit{set}\ xs \Longrightarrow f\ (g\ x)\ (g'\ x \mathbin{@} ys) = (h\ x,\ ys)$
shows *fold-map f* (*map g xs*) (*concat* (*map* g' *xs*) @ *ys*) = (*map h xs*, *ys*)
using *assms* **by** (*induct xs*) *simp-all*

lemma *map-fst-fold-map*:
map f (*fst* (*fold-map g xs y*)) = *fst* (*fold-map* (λ*a b*. *apfst f* (*g a b*)) *xs y*)
by (*induct xs arbitrary*: *y*) (*auto split*: *prod.splits*, *metis fst-conv*)

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*
using *adjust-idx-def adjust-idx-rev-def* **by** *auto*

lemma *adjust-idx-rev2*:
assumes *j* ≠ *i* **shows** *adjust-idx i* (*adjust-idx-rev i j*) = *j*
using *adjust-idx-def adjust-idx-rev-def assms* **by** *auto*

lemma *adjust-idx-i*:
adjust-idx i j ≠ *i*
using *adjust-idx-def lessI less-irrefl-nat* **by** *auto*

lemma *adjust-idx-nth*:
assumes *i*: *i* < *length xs*
shows *remove-nth i xs* ! *j* = *xs* ! *adjust-idx i j* (**is** *?l* = *?r*)
proof −
let *?j* = *adjust-idx i j*
from *i* **have** *ltake*: *length* (*take i xs*) = *i* **by** *simp*
note *nth-xs* = *arg-cong*[**where** *f* = λ *xs*. *xs* ! *?j*, *OF id-take-nth-drop*[*OF i*],
unfolded nth-append ltake]
show *?thesis*
proof (*cases j* < *i*)
case *True*
hence *j*: *?j* = *j* **unfolding** *adjust-idx-def* **by** *simp*
show *?thesis* **unfolding** *nth-xs* **unfolding** *j remove-nth-def nth-append ltake*
using *True* **by** *simp*
next
case *False*
hence *j*: *?j* = *Suc j* **unfolding** *adjust-idx-def* **by** *simp*
from *i* **have** *lxs*: *min* (*length xs*) *i* = *i* **by** *simp*
show *?thesis* **unfolding** *nth-xs* **unfolding** *j remove-nth-def nth-append*
using *False* **by** (*simp add*: *lxs*)
qed
qed

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*)
by (*simp add*: *adjust-idx-nth adjust-idx-rev2 i ji*)

lemma *adjust-idx-length*:
assumes *i*: *i* < *length xs*
and *j*: *j* < *length* (*remove-nth i xs*)
shows *adjust-idx i j* < *length xs*
using *adjust-idx-def i j remove-nth-len* **by** *fastforce*

lemma *adjust-idx-rev-length*:
assumes *i* < *length xs*

and $j < length\ xs$
and $j \neq i$
shows *adjust-idx-rev i j* $<$ *length* (*remove-nth i xs*)
by (*metis adjust-idx-def adjust-idx-rev2 assms not-less-eq remove-nth-len*)

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*: $\forall i.\ i < length\ bs \longrightarrow P\ (as\ !\ i)\ (bs\ !\ i)$
and *nscd*: $\forall i.\ i < length\ ds \longrightarrow P\ (cs\ !\ i)\ (ds\ !\ i)$
shows $\forall i.\ i < length\ (bs\ @\ ds) \longrightarrow P\ ((as\ @\ cs)\ !\ i)\ ((bs\ @\ ds)\ !\ i)$
by (*simp add*: *lab nsab nscd nth-append*)

Extension of filter and partition to binary relations.

fun *filter2* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow ('a\ list \times 'b\ list)$ **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 (*induct as arbitrary*: *bs*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons a as*) **note** *IH* = *this*
thus *?case* **proof** (*cases bs*)
case *Nil*
thus *?thesis* **by** *simp*
next
case (*Cons b bs*)
thus *?thesis* **proof** (*cases P a b*)
case *True*
with *Cons* **and** *IH* **show** *?thesis* **by** *simp*
next
case *False*
with *Cons* **and** *IH* **show** *?thesis* **by** *simp*
qed
qed
qed

lemma *filter2-sound*: $\forall i.\ i < length\ (fst\ (filter2\ P\ as\ bs)) \longrightarrow P\ (fst\ (filter2\ P\ as\ bs)\ !\ i)\ (snd\ (filter2\ P\ as\ bs)\ !\ i)$
proof (*induct as arbitrary*: *bs*)
case *Nil*

```
 thus ?case by simp
next
case (Cons a as) note IH = this
 thus ?case proof (cases bs)
  case Nil
   thus ?thesis by simp
  next
  case (Cons b bs)
   thus ?thesis proof (cases P a b)
    case False
     with Cons and IH show ?thesis by simp
    next
    case True
     {
      fix i
      assume i-bd: i < length (fst (filter2 P (a # as) (b # bs)))
       have P (fst (filter2 P (a # as) (b # bs)) ! i) (snd (filter2 P (a # as) (b
# bs)) ! i)          proof (cases i)
        case 0
         with True show ?thesis by simp
        next
        case (Suc j)
         with i-bd and True have j < length (fst (filter2 P as bs)) by auto
         with Suc and IH and True show ?thesis by simp
       qed
     }
     with Cons show ?thesis by simp
   qed
qed
qed

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 (λa 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)
  by (simp add: filter2-sound partition2-def)

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)
  by (metis filter2-sound partition2-def snd-conv)
```

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
using assms proof (induct as arbitrary: i)
  case Nil thus ?case by simp
  next
  case (Cons a as) note IH = this
   thus ?case proof (cases x = a)
     case True with IH(2) show ?thesis by simp
     next
     case False note neq-x-a = this
      show ?thesis proof (cases mem-idx x as)
       case None with IH(2) and neq-x-a show ?thesis by simp
       next
       case (Some j)
        with IH(2) and neq-x-a have i = Suc j by simp
        with IH(1) and Some show ?thesis by simp
      qed
   qed
qed

lemma mem-idx-sound-output2:
  assumes mem-idx x as = Some i
  shows ∀ j. j < i ⟶ as ! j ≠ x
using assms proof (induct as arbitrary: i)
  case Nil thus ?case by simp
  next
  case (Cons a as) note IH = this
   thus ?case proof (cases x = a)
     case True with IH show ?thesis by simp
     next
     case False note neq-x-a = this
      show ?thesis proof (cases mem-idx x as)
       case None with IH(2) and neq-x-a show ?thesis by simp
       next
       case (Some j)
        with IH(2) and neq-x-a have eq-i-Sj: i = Suc j by simp
        {
          fix k assume k-bd: k < i
          have (a # as) ! k ≠ x
          proof (cases k)
          case 0 with neq-x-a show ?thesis by simp
          next
          case (Suc l)
            with k-bd and eq-i-Sj have l-bd: l < j by arith
            with IH(1) and Some have as ! l ≠ x by simp
            with Suc show ?thesis by simp
          qed
```

```
      }
      thus ?thesis by simp
    qed
  qed
qed

lemma mem-idx-sound:
 (x ∈ set as) = (∃ i. mem-idx x as = Some i)
proof (induct as)
 case Nil thus ?case by simp
 next
 case (Cons a as) note IH = this
  show ?case proof (cases x = a)
   case True thus ?thesis by simp
   next
   case False
    {
     assume x ∈ set (a # as)
      with False have x ∈ set as by simp
      with IH obtain i where Some-i: mem-idx x as = Some i by auto
      with False have mem-idx x (a # as) = Some (Suc i) by simp
     hence ∃ i. mem-idx x (a # as) = Some i by simp
    }
    moreover
    {
     assume ∃ i. mem-idx x (a # as) = Some i
      then obtain i where Some-i: mem-idx x (a # as) = Some i by fast
      have x ∈ set as proof (cases i)
        case 0 with mem-idx-sound-output[OF Some-i] and False show ?thesis
by simp
       next
       case (Suc j)
        with Some-i and False have mem-idx x as = Some j by simp
        hence ∃ i. mem-idx x as = Some i by simp
        with IH show ?thesis by simp
      qed
      hence x ∈ set (a # as) by simp
    }
    ultimately show ?thesis by fast
  qed
qed

lemma mem-idx-sound2:
  (x ∉ set as) = (mem-idx x as = None)
  unfolding mem-idx-sound by auto

lemma sum-list-replicate-mono: assumes w1 ≤ (w2 :: nat)
  shows sum-list (replicate n w1) ≤ sum-list (replicate n w2)
proof (induct n)
```

```
  case (Suc n)
  thus ?case using ‹w1 ≤ w2› by auto
qed simp

lemma all-gt-0-sum-list-map:
  assumes ∗: ⋀x. f x > (0::nat)
    and x: x ∈ set xs and len: 1 < length xs
  shows f x < (∑ x←xs. f x)
  using x len
proof (induct xs)
  case (Cons y xs)
  show ?case
  proof (cases y = x)
    case True
    with ∗[of hd xs] Cons(3) show ?thesis by (cases xs, auto)
  next
    case False
    with Cons(2) have x: x ∈ set xs by auto
    then obtain z zs where xs: xs = z # zs by (cases xs, auto)
    show ?thesis
    proof (cases length zs)
      case 0
      with x xs ∗[of y] show ?thesis by auto
    next
      case (Suc n)
      with xs have 1 < length xs by auto
      from Cons(1)[OF x this] show ?thesis by simp
    qed
  qed
qed simp

lemma map-of-filter:
  assumes P x
  shows map-of [(x',y) ← ys. P x'] x = map-of ys x
proof (induct ys)
  case (Cons xy ys)
  obtain x' y where xy: xy = (x',y) by force
  show ?case
    using assms local.Cons by auto
qed simp

lemma set-subset-insertI: set xs ⊆ set (List.insert x xs)
  by auto

lemma set-removeAll-subset: set (removeAll x xs) ⊆ set xs
  by auto

lemma map-of-append-Some:
  map-of xs y = Some z ⟹ map-of (xs @ ys) y = Some z
```

by *simp*

lemma *map-of-append-None*:
map-of xs y = *None* ⟹ *map-of* (*xs* @ *ys*) *y* = *map-of ys y*
by (*simp add*: *map-add-def*)

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*)#}
using *assms*
proof (*induct as arbitrary*: *n*)
case (*Cons a as*)
note [*simp*] = *remove-nth-def*
show *?case*
proof (*cases n*)
case (*Suc n*)
with *Cons* **have** *n-bd*: *n* < *length as* **by** *auto*
with *Cons* **have** *mset* (*remove-nth n as*) = *mset as* − {#*as* ! *n*#} **by** *auto*
hence *G*: *mset* (*remove-nth* (*Suc n*) (*a* # *as*)) = *mset as* − {#*as* ! *n*#} + {#*a*#}
by *simp*
thus *?thesis*
proof (*cases a* = *as*!*n*)
case *True*
with *G* **and** *Suc* **and** *insert-DiffM2*[*symmetric*]
and *insert-DiffM2*[*of* - {#*as* ! *n*#}]
and *nth-mem-mset*[*of n as*] **and** *n-bd*
show *?thesis* **by** *auto*
next
case *False*
from *G* **and** *Suc* **and** *diff-union-swap*[*OF this*[*symmetric*], *symmetric*] **show** *?thesis* **by** *simp*
qed

```
  qed auto
qed auto


lemma multiset-subset-insert: {ps. ps ⊆# add-mset x xs} =
    {ps. ps ⊆# xs} ∪ add-mset x ` {ps. ps ⊆# xs} (is ?l = ?r)
proof −
  {
    fix ps
    have (ps ∈ ?l) = (ps ⊆# xs + {# x #}) by auto
    also have ... = (ps ∈ ?r)
    proof (cases x ∈# ps)
      case True
      then obtain qs where ps: ps = qs + {#x#} by (metis insert-DiffM2)
      show ?thesis unfolding ps mset-subset-eq-mono-add-right-cancel
        by (auto dest: mset-subset-eq-insertD)
    next
      case False
      hence id: (ps ⊆# xs + {#x#}) = (ps ⊆# xs)
        by (simp add: subset-mset.inf.absorb-iff2 inter-add-left1)
      show ?thesis unfolding id using False by auto
    qed
    finally have (ps ∈ ?l) = (ps ∈ ?r) .
  }
  thus ?thesis by auto
qed

lemma multiset-of-subseqs: mset ` set (subseqs xs) = { ps. ps ⊆# mset xs}
proof (induct xs)
  case (Cons x xs)
  show ?case (is ?l = ?r)
  proof −
    have id: ?r = {ps. ps ⊆# mset xs} ∪ (add-mset x) ` {ps. ps ⊆# mset xs}
      by (simp add: multiset-subset-insert)
    show ?thesis unfolding id Cons[symmetric]
      by (auto simp add: Let-def) (metis UnCI image-iff mset.simps(2))
  qed
qed simp

lemma remove1-mset: w ∈ set vs ⟹ mset (remove1 w vs) + {#w#} = mset vs
  by (induct vs) auto

lemma fold-remove1-mset: mset ws ⊆# mset vs ⟹ mset (fold remove1 ws vs) +
mset ws = mset vs
proof (induct ws arbitrary: vs)
  case (Cons w ws vs)
  from Cons(2) have w ∈ set vs using set-mset-mono by force
  from remove1-mset[OF this] have vs: mset vs = mset (remove1 w vs) + {#w#}
by simp
```

from *Cons(2)*[*unfolded vs*] **have** *mset ws* $\subseteq\#$ *mset* (*remove1 w vs*) **by** *auto*
from *Cons(1)*[*OF this*,*symmetric*]
show *?case* **unfolding** *vs* **by** (*simp add*: *ac-simps*)
qed *simp*

lemma *subseqs-sub-mset*: *ws* $\in$ *set* (*subseqs vs*) $\Longrightarrow$ *mset ws* $\subseteq\#$ *mset vs*
proof (*induct vs arbitrary*: *ws*)
case (*Cons v vs Ws*)
note *mem* = *Cons(2)*
note *IH* = *Cons(1)*
show *?case*
proof (*cases Ws*)
case (*Cons w ws*)
show *?thesis*
proof (*cases v* = *w*)
case *True*
from *mem Cons* **have** *ws* $\in$ *set* (*subseqs vs*) **by** (*auto simp*: *Let-def Cons-in-subseqsD*[*of* - *ws vs*])
from *IH*[*OF this*]
show *?thesis* **unfolding** *Cons True* **by** *simp*
next
case *False*
with *mem Cons* **have** *Ws* $\in$ *set* (*subseqs vs*) **by** (*auto simp*: *Let-def Cons-in-subseqsD*[*of* - *ws vs*])
note *IH* = *mset-subset-eq-count*[*OF IH*[*OF this*]]
with *IH*[*of v*] **show** *?thesis* **by** (*intro mset-subset-eqI*, *auto*, *linarith*)
qed
qed *simp*
qed *simp*

lemma *filter-mset-inequality*: *filter-mset f xs* $\neq$ *xs* $\Longrightarrow$ $\exists$ *x* $\in\#$ *xs*. $\neg$ *f x*
by (*induct xs*, *auto*)

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* $\in$ *set xs* $\Longrightarrow$ *Mapping.lookup* (*Mapping.tabulate xs f*) *x* = *Some* (*f x*)

by (*transfer*, *simp add*: *map-of-map-Pair-key*)

lemma *lookup-tabulate2*: *Mapping.lookup* (*Mapping.tabulate xs f*) *x* = *Some y* ⟹ *y* = *f x*
by *transfer* (*metis map-of-map-Pair-key option.distinct*(*1*) *option.sel*)

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 (*intro ext*)
fix *x*
show *memo-int low up f x* = *f x*
proof (*cases x* ≥ *low* ∧ *x* ≤ *up*)
case *False*
thus *?thesis* **unfolding** *memo-int-def* **by** *auto*
next
case *True*
from *True* **have** *x*: *x* ∈ *set* [*low* .. *up*] **by** *auto*
with *True lookup-tabulate*[*OF this*, *of f*]
show *?thesis* **unfolding** *memo-int-def* **by** *auto*
qed
qed

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 (*intro ext*)
fix *x*
show *memo-nat low up f x* = *f x*
proof (*cases x* ≥ *low* ∧ *x* < *up*)
case *False*
thus *?thesis* **unfolding** *memo-nat-def* **by** *auto*
next
case *True*
from *True* **have** *x*: *x* ∈ *set* [*low* ..< *up*] **by** *auto*
with *True lookup-tabulate*[*OF this*, *of f*]
show *?thesis* **unfolding** *memo-nat-def* **by** *auto*
qed
qed

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 (*intro ext*)
fix *x*
show *memo xs f x = f x*
proof (*cases Mapping.lookup* (*Mapping.tabulate xs f*) *x*)
case *None*
thus *?thesis* **unfolding** *memo-def* **by** *auto*
next
case (*Some y*)
with *lookup-tabulate2*[*OF this*]
show *?thesis* **unfolding** *memo-def* **by** *auto*
qed
qed

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* **unfolding** *order-def*
proof (*rule Least-equality*, *intro notI*)
assume [:− *a*, *1*:] ^ *Suc* (*Suc 0*) *dvd* [:− *a*, *1*:]
from *dvd-imp-degree-le*[*OF this*] **show** *False* **by** *auto*
next
fix *n*
assume ∗: ¬ [:− *a*, *1*:] ^ *Suc n dvd* [:− *a*, *1*:]
thus *Suc 0* ≤ *n*
by (*cases n*, *auto*)
qed

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

lemma *linear-power-nonzero*: [: *a*, *1* :] ^ *n* ≠ *0*
proof
assume [: *a*, *1* :]^*n* = *0*
with *arg-cong*[*OF this*, *of degree*, *unfolded degree-linear-power*]
show *False* **by** *auto*
qed

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

```
  case True
  thus ?thesis unfolding True order-power-n-n by simp
next
  case False
  let ?p = [: b, 1 :]^Suc n
  from linear-power-nonzero have ?p ≠ 0 .
  have p: ?p = (∏ a← replicate (Suc n) b. [:a, 1:]) by auto
  {
    assume order a ?p ≠ 0
    then obtain m where ord: order a ?p = Suc m by (cases order a ?p, auto)
    from order[OF ‹?p ≠ 0›, of a, unfolded ord] have dvd: [:− a, 1:] ^ Suc m dvd
?p by auto
     from poly-linear-exp-linear-factors[OF dvd[unfolded p]] False have False by
auto
  }
  hence order a ?p = 0 by auto
  with False show ?thesis by simp
qed

lemma order-linear-power: order a ([: b, 1 :]^n) = (if b = −a then n else 0)
proof (cases n)
  case (Suc m)
  show ?thesis unfolding Suc order-linear-power' by simp
qed simp


lemma order-linear': order a [: b, 1 :] = (if b = −a then 1 else 0)
  using order-linear-power'[of a b 0] by simp

lemma degree-div-less:
  assumes p: (p::'a::field poly) ≠ 0 and dvd: r dvd p and deg: degree r ≠ 0
  shows degree (p div r) < degree p
proof −
  from dvd obtain q where prq: p = r * q unfolding dvd-def by auto
  have degree p = degree r + degree q
    unfolding prq
    by (rule degree-mult-eq, insert p prq, auto)
  with deg have deg: degree q < degree p by auto
  from prq have q = p div r
    using deg p by auto
  with deg show ?thesis by auto
qed


lemma order-sum-degree: assumes p ≠ 0
  shows sum (λ a. order a p) { a. poly p a = 0 } ≤ degree p
proof −
  define n where n = degree p
  have degree p ≤ n unfolding n-def by auto
```

```
  thus ?thesis using ‹p ≠ 0›
  proof (induct n arbitrary: p)
    case (0 p)
    define a where a = coeff p 0
    from 0 have degree p = 0 by auto
    hence p: p = [: a :] unfolding a-def
      by (metis degree-0-id)
    with 0 have a ≠ 0 by auto
    thus ?case unfolding p by auto
  next
    case (Suc m p)
    note order = order[OF ‹p ≠ 0›]
    show ?case
    proof (cases ∃ a. poly p a = 0)
      case True
      then obtain a where root: poly p a = 0 by auto
      with order-root[of p a] Suc obtain n where orda: order a p = Suc n
        by (cases order a p, auto)
      let ?a = [: −a, 1 :] ^ Suc n
      from order-decomp[OF ‹p ≠ 0›, of a, unfolded orda]
        obtain q where p: p = ?a * q and ndvd: ¬ [:− a, 1:] dvd q by auto
      from ‹p ≠ 0›[unfolded p] have nz: ?a ≠ 0 q ≠ 0 by auto
      hence deg: degree p = degree ?a + degree q unfolding p
        by (subst degree-mult-eq, auto)
      have ord: ⋀ a. order a p = order a ?a + order a q
        unfolding p
        by (subst order-mult, insert nz, auto)
      have roots: { a. poly p a = 0 } = insert a ({ a. poly q a = 0} − {a}) using
root
        unfolding p poly-mult by auto
      have fin: finite {a. poly q a = 0} by (rule poly-roots-finite[OF ‹q ≠ 0›])
      have Suc n = order a p using orda by simp
       also have ... = Suc n + order a q unfolding ord order-linear-power' by
simp
      finally have order a q = 0 by auto
      with order-root[of q a] ‹q ≠ 0› have qa: poly q a ≠ 0 by auto
      have (∑ a∈{a. poly q a = 0} − {a}. order a p) = (∑ a∈{a. poly q a = 0}
− {a}. order a q)
      proof (rule sum.cong[OF refl])
        fix b
        assume b ∈ {a. poly q a = 0} − {a}
        hence b ≠ a by auto
        hence order b ?a = 0 unfolding order-linear-power' by simp
        thus order b p = order b q unfolding ord by simp
      qed
      also have ... = (∑ a∈{a. poly q a = 0}. order a q) using qa by auto
      also have ... ≤ degree q
        by (rule Suc(1)[OF - ‹q ≠ 0›],
        insert deg[unfolded degree-linear-power] Suc(2), auto)
```

```
    finally have (∑ a∈{a. poly q a = 0} − {a}. order a p) ≤ degree q .
    thus ?thesis unfolding roots deg using fin
      by (subst sum.insert, simp-all only: degree-linear-power, auto simp: orda)
  qed auto
 qed
qed

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 ≠ 0 then 0 else Suc (order a (p div [: −a, 1 :]))))
proof (cases p = 0)
 case False note p = this
 note order = order[OF p]
 show ?thesis
 proof (cases poly p a = 0)
  case True
  with order-root[of p a] p obtain n where ord: order a p = Suc n
   by (cases order a p, auto)
  from this(1) have [: −a, 1 :] dvd p
   using True poly-eq-0-iff-dvd by blast
  then obtain q where p: p = [: −a, 1 :] ∗ q unfolding dvd-def by auto
  have ord: order a p = order a [: −a, 1 :] + order a q
   using p False order-mult[of [: −a, 1 :] q] by auto
  have q: p div [: −a, 1 :] = q using False p
   by (metis mult-zero-left nonzero-mult-div-cancel-left)
  show ?thesis unfolding ord q using False True by auto
 next
  case False
  with order-root[of p a] p show ?thesis by auto
 qed
qed auto

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 ≠ 0 and p0: degree p = 0
 shows {x. poly p x = 0} = {}
```

```
  using degree0-coeffs[OF p0] p by auto

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 −
  obtain a b where p = [: b, a :] a ≠ 0
    by (meson degree1-coeffs p1)
  then show ?thesis unfolding roots1-def
    by (auto simp: add-eq-0-iff nonzero-neg-divide-eq-eq2)
qed

lemma roots2: fixes p :: 'a :: field-char-0 poly
  assumes p2: p = [: c, b, a :] and a: a ≠ 0
  shows {x. poly p x = 0} = { − ( b / (2 * a)) + e | e. e^2 = ( b / (2 * a))^2
− c/a} (is ?l = ?r)
proof −
  define b2a where b2a = b / (2 * a)
  {
    fix x
     have (x ∈ ?l) = (x * x * a + x * b + c = 0) unfolding p2 by (simp add:
field-simps)
    also have ... = ((x * x + 2 * x * b2a) + c/a = 0) using a by (auto simp:
b2a-def field-simps)
   also have x * x + 2 * x * b2a = (x * x + 2 * x * b2a + b2a^2) − b2a^2 by
simp
    also have ... = (x + b2a) ^ 2 − b2a ^ 2
      by (simp add: field-simps power2-eq-square)
    also have (... + c / a = 0) = ((x + b2a) ^ 2 = b2a^2 − c/a) by algebra
     also have ... = (x ∈ ?r) unfolding b2a-def[symmetric] by (auto simp:
field-simps)
    finally have (x ∈ ?l) = (x ∈ ?r) .
  }
  thus ?thesis by auto
qed

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 −
from *degree2-coeffs*[*OF assms*] **obtain** *a b c*
where *p*: *p* = [:*c*, *b*, *a*:] **and** *a*: *a* ≠ *0* **by** *metis*
note *main* = *roots2*[*OF p a*]
have *2*: *2* = *Suc* (*Suc 0*) **by** *simp*
have *coeff*: *coeff p 2* = *a coeff p 1* = *b coeff p 0* = *c* **unfolding** *p* **by** (*auto simp*: *2*)
let *?b2a* = *b* / (*2* ∗ *a*)
define *b2a* **where** *b2a* = *?b2a*
let *?bac* = *b2a*^*2* − *c*/*a*
define *bac* **where** *bac* = *?bac*
have *roots*: *set* (*croots2 p*) = {− *b2a* + *csqrt bac*, − *b2a* − *csqrt bac*}
unfolding *croots2-def Let-def coeff b2a-def*[*symmetric*] *bac-def*[*symmetric*]
by (*auto split*: *if-splits*)
show *?thesis* **unfolding** *roots main b2a-def*[*symmetric*] *bac-def*[*symmetric*]
using *power2-eq-iff* **by** *fastforce*
qed

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 −
from *degree2-coeffs*[*OF assms*] **obtain** *a b c*
where *p*: *p* = [:*c*, *b*, *a*:] **and** *a*: *a* ≠ *0* **by** *metis*
note *main* = *roots2*[*OF p a*]
have *2*: *2* = *Suc* (*Suc 0*) **by** *simp*
have *coeff*: *coeff p 2* = *a coeff p 1* = *b coeff p 0* = *c* **unfolding** *p* **by** (*auto simp*: *2*)
let *?b2a* = *b* / (*2* ∗ *a*)
define *b2a* **where** *b2a* = *?b2a*
let *?bac* = *b2a*^*2* − *c*/*a*
define *bac* **where** *bac* = *?bac*
have *roots*: *set* (*rroots2 p*) = (*if bac* < *0 then* {} *else* {− *b2a* + *sqrt bac*, − *b2a* − *sqrt bac*})
unfolding *rroots2-def Let-def coeff b2a-def*[*symmetric*] *bac-def*[*symmetric*]

```
    by (auto split: if-splits)
  show ?thesis unfolding roots main b2a-def[symmetric] bac-def[symmetric]
    by auto
qed

lemma rat-roots2: assumes degree p = 2
  shows {x. poly p x = 0} = set (rat-roots2 p)
proof −
  from degree2-coeffs[OF assms] obtain a b c
  where p: p = [:c, b, a:] and a: a ≠ 0 by metis
  note main = roots2[OF p a]
  have 2: 2 = Suc (Suc 0) by simp
  have coeff: coeff p 2 = a coeff p 1 = b coeff p 0 = c unfolding p by (auto simp:
2)
  let ?b2a = b / (2 * a)
  define b2a where b2a = ?b2a
  let ?bac = b2a^2 − c/a
  define bac where bac = ?bac
  have roots: (rat-roots2 p) = (map (λ e. −b2a + e) (sqrt-rat bac))
    unfolding rat-roots2-def Let-def coeff b2a-def[symmetric] bac-def[symmetric]
by auto
  show ?thesis unfolding roots main b2a-def[symmetric] bac-def[symmetric]
    by (auto simp: power2-eq-square)
qed
```

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}
  using assms unfolding croots-def
  using roots0[of p] roots1[of p] croots2[of p]
  by (auto split: if-splits)
```

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 ≠ 0 degree p ≤ 2
  shows set (rroots p) = {x. poly p x = 0}
  using assms unfolding rroots-def
  using roots0[of p] roots1[of p] rroots2[of p]
  by (auto split: if-splits)

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* : ⋀ *a b c*. *b* ∈ *s* ⟹ *f'* *a* = *f b* (*f'* *c*) ⟹ *g'* *a* = *g b* (*g'* *c*)
and *f'-inj* : ⋀ *a b*. *f'* *a* = *f'* *b* ⟹ *a* = *b*
and *f-bit-sur* : ⋀ *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*)
using *lst-in-s*
proof (*induct lst arbitrary*: *a*)
case (*Cons x xs*)
have *prems*: *f'* *a* = (*f x* ∘ *foldr f xs*) (*f'* *b*) **using** *Cons.prems* **unfolding** *foldr-Cons* **by** *auto*
hence ∃ *c'*. *f'* *c'* = *foldr f xs* (*f'* *b*) **using** *f-bit-sur* **by** *fastforce*

then obtain c' **where** c'*-def*: $f'\ c' = foldr\ f\ xs\ (f'\ b)$ **by** *blast*
hence $f'\ a = f\ x\ (f'\ c')$ **using** *prems* **by** *simp*
hence $g'\ a = g\ x\ (g'\ c')$ **using** *f-is-g Cons.prems*(2) **by** *simp*
also have $g'\ c' = foldr\ g\ xs\ (g'\ b)$ **using** *Cons.hyps*[*of* c'] c'*-def Cons.prems*(2)
by *auto*
finally have $g'\ a = (g\ x \circ foldr\ g\ xs)\ (g'\ b)$ **by** *simp*
thus *?case* **using** *foldr-Cons* **by** *simp*
qed (*insert* f'*-inj*, *auto*)

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

lemma *rat-int-poly-step-agree* :
assumes $coeff\ (pCons\ b\ c2)\ (degree\ q)\ mod\ coeff\ q\ (degree\ q) = 0$
shows $(rp\ a1, rp\ a2) = (div\text{-}rat\text{-}poly\text{-}step\ (rp\ q) \circ rat\text{-}of\text{-}int)\ b\ (rp\ c1, rp\ c2)$
$\longleftrightarrow Some\ (a1, a2) = div\text{-}int\text{-}poly\text{-}step\ q\ b\ (Some\ (c1, c2))$
proof −
have *coeffs*: $coeff\ (pCons\ b\ c2)\ (degree\ q)\ mod\ coeff\ q\ (degree\ q) = 0$ **using** *assms* **by** *auto*
let $?ri = rat\text{-}of\text{-}int$
let $?withDiv1 = pCons\ (?ri\ (coeff\ (pCons\ b\ c2)\ (degree\ q)\ div\ coeff\ q\ (degree\ q)))\ (rp\ c1)$
let $?withSls1 = pCons\ (coeff\ (pCons\ (?ri\ b)\ (rp\ c2))\ (degree\ q)\ /\ coeff\ (rp\ q)\ (degree\ q))\ (rp\ c1)$
let $?ident1 = ?withDiv1 = ?withSls1$
let $?withDiv2 = rp\ (pCons\ b\ c2 - smult\ (coeff\ (pCons\ b\ c2)\ (degree\ q)\ div\ coeff\ q\ (degree\ q))\ q)$
let $?withSls2 = pCons\ (?ri\ b)\ (rp\ c2) - smult\ (coeff\ (pCons\ (?ri\ b)\ (rp\ c2))\ (degree\ q)\ /\ coeff\ (rp\ q)\ (degree\ q))\ (rp\ q)$
let $?ident2 = ?withDiv2 = ?withSls2$
note *simps* = *div-int-poly-step-def option.simps Let-def prod.simps*
have *id1*:$?ri\ (coeff\ (pCons\ b\ c2)\ (degree\ q)\ div\ coeff\ q\ (degree\ q)) =$
$?ri\ (coeff\ (pCons\ b\ c2)\ (degree\ q))\ /\ ?ri\ (coeff\ q\ (degree\ q))$ **using** *coeffs*
by *auto*
have *id2*:*?ident1* **unfolding** *id1*
by (*simp*, *fold of-int-hom.coeff-map-poly-hom of-int-hom.map-poly-pCons-hom*, *simp*)
hence *id3*:*?ident2* **using** *id2* **by** (*auto simp*: *hom-distribs*)

have *c1*:$((rp\ (pCons\ (coeff\ (pCons\ b\ c2)\ (degree\ q)\ div\ coeff\ q\ (degree\ q))\ c1)$
$,rp\ (pCons\ b\ c2 - smult\ (coeff\ (pCons\ b\ c2)\ (degree\ q)\ div\ coeff\ q\ (degree\ q))\ q))$
$= div\text{-}rat\text{-}poly\text{-}step\ (rp\ q)\ (?ri\ b)\ (rp\ c1, rp\ c2)) \longleftrightarrow (?ident1 \wedge ?ident2)$
unfolding *div-rat-poly-step-def simps*
by (*simp add*: *hom-distribs*)
have $((rp\ a1,\ rp\ a2) = (div\text{-}rat\text{-}poly\text{-}step\ (rp\ q) \circ rat\text{-}of\text{-}int)\ b\ (rp\ c1,\ rp\ c2))$
$\longleftrightarrow$
$(rp\ a1 = ?withSls1 \wedge rp\ a2 = ?withSls2)$

unfolding *div-rat-poly-step-def simps* **by** *simp*
also have ... ⟷
((*a1* = *pCons* (*coeff* (*pCons b c2*) (*degree q*) *div coeff q* (*degree q*)) *c1*) ∧
(*a2* = *pCons b c2* − *smult* (*coeff* (*pCons b c2*) (*degree q*) *div coeff q* (*degree q*)) *q*))
by (*fold id2 id3 of-int-hom.map-poly-pCons-hom*, *unfold of-int-poly-hom.eq-iff*, *auto*)
also have *c0*:... ⟷ *Some* (*a1*,*a2*) = *div-int-poly-step q b* (*Some* (*c1*,*c2*))
unfolding *divmod-int-def div-int-poly-step-def option.simps Let-def prod.simps*
using *coeffs* **by** (*auto split*: *option.splits prod.splits if-splits*)
finally show *?thesis* **.**
qed

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 −
note *simps* = *div-int-poly-step-def option.simps Let-def divmod-int-def prod.simps*
from *Some*[*unfolded simps*] **have** *mod0*: *coeff* (*pCons b c2*) (*degree q*) *mod coeff q* (*degree q*) = *0*
by (*auto split*: *option.splits prod.splits if-splits*)
thus *?thesis* **using** *assms rat-int-poly-step-agree* **by** *auto*
qed

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
assume *is-int-rat* (*rat-of-int x* / *rat-of-int y*)
then obtain *v* **where** *v-def*:*rat-of-int v* = *rat-of-int x* / *rat-of-int y*
using *int-of-rat*(*2*) *is-int-rat* **by** *fastforce*
hence *v* = ⌊*rat-of-int x* / *rat-of-int y*⌋ **by** *linarith*
hence ∗: *v* ∗ *y* = *x* − *x mod y* **using** *div-is-floor-divide-rat mod-div-equality-int* **by** *simp*
have *rat-of-int v* ∗ *rat-of-int y* = *rat-of-int x* − *rat-of-int* (*x mod y*)
unfolding *of-int-mult* [*symmetric*] ∗ **by** *simp*
hence (*rat-of-int x* / *rat-of-int y*) ∗ *rat-of-int y* = *rat-of-int x* − *rat-of-int* (*x mod y*)
using *v-def* **by** *simp*
hence *rat-of-int x* = *rat-of-int x* − *rat-of-int* (*x mod y*) **by** (*simp add*: *assms*)
thus *x mod y* = *0* **by** *simp*
qed (*force*)

lemma *pCons-of-rp-contains-ints* :
assumes *rp a* = *pCons b c*
shows *is-int-rat b*
proof −
have ⋀ *b n*. *rp a* = *b* ⟹ *is-int-rat* (*coeff b n*) **by** *auto*
hence *rp a* = *pCons b c* ⟹ *is-int-rat* (*coeff* (*pCons b c*) *0*)**.**

thus *?thesis* **using** *assms* **by** *auto*
qed

lemma *rat-step-then-int-poly-step* :
assumes $q \neq 0$
and $(rp\ a1, rp\ a2) = (div\text{-}rat\text{-}poly\text{-}step\ (rp\ q) \circ rat\text{-}of\text{-}int)\ b2\ (rp\ c1, rp\ c2)$
shows $Some\ (a1, a2) = div\text{-}int\text{-}poly\text{-}step\ q\ b2\ (Some\ (c1, c2))$
proof −
let *?mustbeint* = $rat\text{-}of\text{-}int\ (coeff\ (pCons\ b2\ c2)\ (degree\ q))\ /\ rat\text{-}of\text{-}int\ (coeff\ q\ (degree\ q))$
let *?mustbeint2* = $coeff\ (pCons\ (rat\text{-}of\text{-}int\ b2)\ (rp\ c2))\ (degree\ (rp\ q))\ /\ coeff\ (rp\ q)\ (degree\ (rp\ q))$
have *mustbeint* : *?mustbeint* = *?mustbeint2* **by** (*fold hom-distribs of-int-hom.coeff-map-poly-hom, simp*)
note *simps* = *div-int-poly-step-def option.simps Let-def divmod-int-def prod.simps*
from *assms leading-coeff-neq-0*[*of q*] **have** *q0*:$coeff\ q\ (degree\ q) \neq 0$ **by** *simp*

have $rp\ a1 = pCons$ *?mustbeint2* $(rp\ c1)$
using *assms*(*2*) **unfolding** *div-rat-poly-step-def* **by** (*simp add*:*div-int-poly-step-def Let-def*)
hence *is-int-rat ?mustbeint2*
unfolding *div-rat-poly-step-def* **using** *pCons-of-rp-contains-ints* **by** *simp*
hence *is-int-rat ?mustbeint* **unfolding** *mustbeint* **by** *simp*
hence $coeff\ (pCons\ b2\ c2)\ (degree\ q)\ mod\ coeff\ q\ (degree\ q) = 0$
using *is-int-rat-division q0* **by** *simp*
thus *?thesis* **using** *rat-int-poly-step-agree assms* **by** *simp*
qed

lemma *div-int-poly-step-surjective* : $Some\ a = div\text{-}int\text{-}poly\text{-}step\ q\ b\ c \Longrightarrow \exists\ c'.\ c = Some\ c'$
unfolding *div-int-poly-step-def* **by**(*cases c, simp-all*)

lemma *div-mod-int-poly-then-pdivmod*:
assumes $div\text{-}mod\text{-}int\text{-}poly\ p\ q = Some\ (r, m)$
shows $(rp\ p\ div\ rp\ q,\ rp\ p\ mod\ rp\ q) = (rp\ r,\ rp\ m)$
and $q \neq 0$
proof −
let *?rpp* = $(\lambda\ (a, b).\ (rp\ a, rp\ b))$
let *?p* = $rp\ p$
let *?q* = $rp\ q$
let *?r* = $rp\ r$
let *?m* = $rp\ m$
let *?div-rat-step* = *div-rat-poly-step ?q*
let *?div-int-step* = *div-int-poly-step q*
from *assms* **show** *q0* : $q \neq 0$ **using** *div-mod-int-poly-def* **by** *auto*
hence $div\text{-}mod\text{-}int\text{-}poly\ p\ q = Some\ (r, m) \longleftrightarrow Some\ (r, m) = foldr\ (div\text{-}int\text{-}poly\text{-}step\ q)\ (coeffs\ p)\ (Some\ (0, 0))$
unfolding *div-mod-int-poly-def fold-coeffs-def* **by** (*auto split*: *option.splits prod.splits if-splits*)

```
  hence innerRes: Some (r,m) = foldr (?div-int-step) (coeffs p) (Some (0, 0))
using assms by simp
  { fix oldRes res :: int poly × int poly
    fix lst :: int list
    have Some res = foldr ?div-int-step lst (Some oldRes) ⟹
          ?rpp res = foldr (?div-rat-step ∘ rat-of-int) lst (?rpp oldRes)
     using foldr-cong-plus[of set lst Some ?div-int-step ?rpp ?div-rat-step ∘ rat-of-int

        lst res oldRes] int-step-then-rat-poly-step div-int-poly-step-surjective by auto
    hence Some res = foldr ?div-int-step lst (Some oldRes)
      ⟹ ?rpp res = foldr ?div-rat-step (map rat-of-int lst) (?rpp oldRes)
      using foldr-map[of ?div-rat-step rat-of-int lst] by simp
  }
  hence equal-foldr : Some (r,m) = foldr (?div-int-step) (coeffs p) (Some (0,0))
    ⟹ ?rpp (r,m) = foldr (?div-rat-step) (map rat-of-int (coeffs p)) (?rpp (0,0)).
  have (map rat-of-int (coeffs p) = coeffs ?p) by simp
  hence (?r,?m) = (foldr (?div-rat-step) (coeffs ?p) (0,0)) using equal-foldr in-
nerRes by simp
  thus (?p div ?q, ?p mod ?q) = (?r,?m)
    using fold-coeffs-def [of ?div-rat-step ?p] q0
      div-mod-fold-coeffs [of ?p ?q]
    unfolding div-rat-poly-step-def by auto
qed

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 −
  obtain b1 b2 where pair: pair = (b1, b2) by (cases pair) simp
  define p12 where p12 = coeff (pCons (rat-of-int x) b2) (degree (rp q)) / coeff
(rp q) (degree (rp q))
  obtain a1 a2 where a = (a1, a2) by (cases a) simp
  with assms pair have (rp a1, rp a2) = div-rat-poly-step (rp q) (rat-of-int x) (b1,
b2)
    by simp
  then have a1: rp a1 = pCons p12 b1
    and rp a2 = pCons (rat-of-int x) b2 − smult p12 (rp q)
    by (auto split: prod.splits simp add: Let-def div-rat-poly-step-def p12-def)
  then obtain p21 p22 where rp p21 = pCons p22 b2
    apply (simp add: field-simps)
   apply (metis coeff-pCons-0 of-int-hom.map-poly-hom-add of-int-hom.map-poly-hom-smult
of-int-hom.coeff-map-poly-hom)
    done
  moreover obtain p21' p21q where p21 = pCons p21' p21q
    by (rule pCons-cases)
  ultimately obtain p2 where b2 = rp p2
    by (auto simp: hom-distribs)
  moreover obtain a1' a1q where a1 = pCons a1' a1q
```

```
    by (rule pCons-cases)
  with a1 obtain p1 where b1 = rp p1
    by (auto simp: hom-distribs)
  ultimately have pair = (rp p1, rp p2) using pair by simp
  then show ?thesis by auto
qed

lemma pdivmod-then-div-mod-int-poly:
  assumes q0: q ≠ 0 and (rp p div rp q, rp p mod rp q) = (rp r, rp m)
  shows   div-mod-int-poly p q = Some (r,m)
proof −
  let ?rpp = (λ (a,b). (rp a,rp b))
  let ?p = rp p
  let ?q = rp q
  let ?r = rp r
  let ?m = rp m
  let ?div-rat-step = div-rat-poly-step ?q
  let ?div-int-step = div-int-poly-step q
  { fix oldRes res :: int poly × int poly
    fix lst :: int list
    have inj: (⋀a b. (case a of (a, b) ⇒ (rp a, rp b)) = (case b of (a, b) ⇒ (rp a,
rp b)) ⟹ a = b)
      by auto
    have (⋀a b c. b ∈ set lst ⟹
            (case a of (a, b) ⇒ (map-poly rat-of-int a, map-poly rat-of-int b)) =
            (div-rat-poly-step (map-poly rat-of-int q) ∘ rat-of-int) b
             (case c of (a, b) ⇒ (map-poly rat-of-int a, map-poly rat-of-int b)) ⟹
            Some a = div-int-poly-step q b (Some c))
      using rat-step-then-int-poly-step[OF q0] by auto
    hence ?rpp res = foldr (?div-rat-step ∘ rat-of-int) lst (?rpp oldRes)
        ⟹ Some res = foldr ?div-int-step lst (Some oldRes)
     using foldr-cong-plus[of set lst ?rpp ?div-rat-step ∘ rat-of-int Some ?div-int-step
lst res oldRes]
          div-rat-poly-step-sur inj by simp
    hence ?rpp res = foldr ?div-rat-step (map rat-of-int lst) (?rpp oldRes)
      ⟹ Some res = foldr ?div-int-step lst (Some oldRes)
      using foldr-map[of ?div-rat-step rat-of-int lst] by auto
  }
  hence equal-foldr : ?rpp (r,m) = foldr (?div-rat-step) (map rat-of-int (coeffs p))
(?rpp (0,0))
                ⟹ Some (r,m) = foldr (?div-int-step) (coeffs p) (Some (0,0))
    by simp
  have (?r,?m) = (foldr (?div-rat-step) (coeffs ?p) (0,0))
    using fold-coeffs-def[of ?div-rat-step ?p] assms
      div-mod-fold-coeffs [of ?p ?q]
    unfolding div-rat-poly-step-def by auto
  hence Some (r,m) = foldr (?div-int-step) (coeffs p) (Some (0,0))
    using equal-foldr by simp
 thus ?thesis using q0 unfolding div-mod-int-poly-def by (simp add: fold-coeffs-def)
```

qed

lemma *div-int-then-rqp*:
assumes *div-int-poly p q = Some r*
shows *r * q = p*
and *q ≠ 0*
proof −
let *?rpp = (λ (a,b). (rp a,rp b))*
let *?p = rp p*
let *?q = rp q*
let *?r = rp r*
have *Some (r,0) = div-mod-int-poly p q* **using** *assms* **unfolding** *div-int-poly-def*
by (*auto split*: *option.splits prod.splits if-splits*)
with *div-mod-int-poly-then-pdivmod*[*of p q r 0*]
have *?p div ?q = ?r ∧ ?p mod ?q = 0* **by** *simp*
with *div-mult-mod-eq*[*of ?p ?q*]
have *?p = ?r * ?q* **by** *auto*
also have *... = rp (r * q)* **by** (*simp add*: *hom-distribs*)
finally have *?p = rp (r * q)*.
thus *r * q = p* **by** *simp*
show *q ≠ 0* **using** *assms* **unfolding** *div-int-poly-def div-mod-int-poly-def*
by (*auto split*: *option.splits prod.splits if-splits*)
qed

lemma *rqp-then-div-int*:
assumes *r * q = p*
and *q0*:*q ≠ 0*
shows *div-int-poly p q = Some r*
proof −
let *?rpp = (λ (a,b). (rp a,rp b))*
let *?p = rp p*
let *?q = rp q*
let *?r = rp r*
have *?p = ?r * ?q* **using** *assms*(*1*) **by** (*auto simp*: *hom-distribs*)
hence *?p div ?q = ?r* **and** *?p mod ?q = 0*
using *q0* **by** *simp-all*
hence (*rp p div rp q*, *rp p mod rp q*) = (*rp r*, *0*) **by** (*auto split*: *prod.splits*)
hence (*rp p div rp q*, *rp p mod rp q*) = (*rp r*, *rp 0*) **by** *simp*
hence *Some (r,0) = div-mod-int-poly p q*
using *pdivmod-then-div-mod-int-poly*[*OF q0*,*of p r 0*] **by** *simp*
thus *?thesis* **unfolding** *div-mod-int-poly-def div-int-poly-def* **using** *q0*
by (*metis* (*mono-tags*, *lifting*) *option.simps*(*5*) *split-conv*)
qed

lemma *div-int-poly*: (*div-int-poly p q = Some r*) ⟷ (*q ≠ 0 ∧ p = r * q*)
using *div-int-then-rqp rqp-then-div-int* **by** *blast*

definition *dvd-int-poly* :: *int poly ⇒ int poly ⇒ bool* **where**
dvd-int-poly q p = (if q = 0 then p = 0 else div-int-poly p q ≠ None)

lemma *dvd-int-poly*[*simp*]: *dvd-int-poly q p* = (*q dvd p*)
unfolding *dvd-def dvd-int-poly-def* **using** *div-int-poly*[*of p q*]
by (*cases q* = *0*, *auto*)

definition *dvd-int-poly-non-0* :: *int poly* ⇒ *int poly* ⇒ *bool* **where**
dvd-int-poly-non-0 q p = (*div-int-poly p q* ≠ *None*)

lemma *dvd-int-poly-non-0*[*simp*]: *q* ≠ *0* ⟹ *dvd-int-poly-non-0 q p* = (*q dvd p*)
unfolding *dvd-def dvd-int-poly-non-0-def* **using** *div-int-poly*[*of p q*] **by** *auto*

lemma [*code-unfold*]: *p dvd q* ⟷ *dvd-int-poly p q* **by** *simp*

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′*))
unfolding *gcd-poly-code-def Let-def primitive-part-def* **by** *simp*

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 (*cases a* = *0* ∨ *f* = *0*)
case *False*
let *?c* = *content*
let *?pp* = *primitive-part*
let *?ua* = *unit-factor a*
let *?na* = *normalize a*

define *H* **where** *H* = *gcd* (*?c f*) (*?c g*)
have *H dvd ?c f* **unfolding** *H-def* **by** *auto*
then obtain *F* **where** *fh*: *?c f* = *H* ∗ *F* **unfolding** *dvd-def* **by** *blast*
from *False* **have** *cf0*: *?c f* ≠ *0* **by** *auto*
hence *H*: *H* ≠ *0* **unfolding** *H-def* **by** *auto*
from *arg-cong*[*OF fh*, *of* λ *f*. *f div H*] *H* **have** *F*: *F* = *?c f div H* **by** *auto*
have *H dvd ?c g* **unfolding** *H-def* **by** *auto*
then obtain *G* **where** *gh*: *?c g* = *H* ∗ *G* **unfolding** *dvd-def* **by** *blast*
from *arg-cong*[*OF gh*, *of* λ *f*. *f div H*] *H* **have** *G*: *G* = *?c g div H* **by** *auto*
have *coprime F G* **using** *H* **unfolding** *F G H-def*
using *cf0 div-gcd-coprime* **by** *blast*
have *is-unit ?ua* **using** *False* **by** *simp*
then have *ua*: *is-unit* [: *?ua* :]
by (*simp add*: *is-unit-const-poly-iff*)
have *gcd* (*smult a f*) *g* = *smult* (*gcd* (*?na* ∗ *?c f*) (*?c g*))
(*gcd* (*smult ?ua* (*?pp f*)) (*?pp g*))
unfolding *gcd-poly-decompose*[*of smult a f*]
content-smult primitive-part-smult **by** *simp*
also have *smult ?ua* (*?pp f*) = *?pp f* ∗ [: *?ua* :] **by** *simp*
also have *gcd* ... (*?pp g*) = *gcd* (*?pp f*) (*?pp g*)
unfolding *gcd-mult-unit1*[*OF ua*] **..**
also have *gcd* (*?na* ∗ *?c f*) (*?c g*) = *gcd* ((*?na* ∗ *F*) ∗ *H*) (*G* ∗ *H*)
unfolding *fh gh* **by** (*simp add*: *ac-simps*)
also have ... = *gcd* (*?na* ∗ *F*) *G* ∗ *normalize H* **unfolding** *gcd-mult-right gcd.commute*[*of G*]
by (*simp add*: *normalize-mult*)
also have *normalize H* = *H* **by** (*metis H-def normalize-gcd*)
finally
have *gcd* (*smult a f*) *g* = *smult* (*gcd* (*?na* ∗ *F*) *G*) (*smult H* (*gcd* (*?pp f*) (*?pp g*))) **by** *simp*
also have *smult H* (*gcd* (*?pp f*) (*?pp g*)) = *gcd f g* **unfolding** *H-def*
by (*rule gcd-poly-decompose*[*symmetric*])
also have *gcd* (*?na* ∗ *F*) *G* = *gcd* (*F* ∗ *?na*) *G* **by** (*simp add*: *ac-simps*)
also have ... = *gcd ?na G*
using ‹*coprime F G*› **by** (*simp add*: *gcd-mult-right-left-cancel ac-simps*)
finally show *?thesis* **unfolding** *G H-def cg cf* **using** *False* **by** *simp*
next
case *True*
hence *gcd* (*smult a f*) *g* = *normalize g* **by** (*cases a* = *0*, *auto*)
thus *?thesis* **using** *True* **by** *simp*
qed

lemma *gcd-smult-ex*: **assumes** *a* ≠ *0*
shows ∃ *b*. *gcd* (*smult a f*) *g* = *smult b* (*gcd f g*) ∧ *b* ≠ *0*
proof (*cases f* = *0*)
case *True*
thus *?thesis* **by** (*intro exI*[*of - 1*], *auto*)
next
case *False*

hence *id*: $(a = 0 \vee f = 0) = False$ **using** *assms* **by** *auto*
show *?thesis* **unfolding** *gcd-smult id if-False*
by (*intro exI conjI*, *rule refl*, *insert assms*, *auto*)
qed

lemma *primitive-part-idemp*[*simp*]:
fixes $f :: 'a :: \{semiring\text{-}gcd,normalization\text{-}semidom\text{-}multiplicative\}$ *poly*
shows *primitive-part* (*primitive-part f*) = *primitive-part f*
by (*metis content-primitive-part*[*of f*] *primitive-part-eq-0-iff primitive-part-prim*)

lemma *content-gcd-primitive*:
$f \neq 0 \Longrightarrow$ *content* (*gcd* (*primitive-part f*) *g*) = *1*
$f \neq 0 \Longrightarrow$ *content* (*gcd* (*primitive-part f*) (*primitive-part g*)) = *1*
by (*metis* (*no-types*, *lifting*) *content-dvd-contentI content-primitive-part gcd-dvd1 is-unit-content-iff*)+

lemma *content-gcd-content*: *content* (*gcd f g*) = *gcd* (*content f*) (*content g*)
(**is** *?l* = *?r*)
proof −
let *?c* = *content*
have *?l* = *normalize* (*gcd* (*?c f*) (*?c g*)) ∗
?c (*gcd* (*primitive-part f*) (*primitive-part g*))
unfolding *gcd-poly-decompose*[*of f g*] *content-smult* **..**
also have ... = *gcd* (*?c f*) (*?c g*) ∗
?c (*gcd* (*primitive-part f*) (*primitive-part g*)) **by** *simp*
also have ... = *?r* **using** *content-gcd-primitive*[*of f g*]
by (*metis* (*no-types*, *lifting*) *content-dvd-contentI content-eq-zero-iff content-primitive-part gcd-dvd2 gcd-eq-0-iff is-unit-content-iff mult-cancel-left1*)
finally show *?thesis* **.**
qed

lemma *gcd-primitive-part*:
gcd (*primitive-part f*) (*primitive-part g*) = *normalize* (*primitive-part* (*gcd f g*))
proof(*cases f* = *0*)
case *True*
show *?thesis* **unfolding** *gcd-poly-decompose*[*of f g*] *gcd-0-left primitive-part-0 True*
by (*simp add*: *associatedI primitive-part-dvd-primitive-partI*)
next
case *False*
have *normalize 1* = *normalize* (*unit-factor* (*gcd* (*content f*) (*content g*)))
by (*simp add*: *False*)
then show *?thesis* **unfolding** *gcd-poly-decompose*[*of f g*]
by (*metis* (*no-types*) *Polynomial.normalize-smult content-gcd-primitive*(*1*)[*OF False*] *content-times-primitive-part normalize-gcd primitive-part-smult*)
qed

lemma *primitive-part-gcd*: *primitive-part* (*gcd f g*)
= *unit-factor* (*gcd f g*) ∗ *gcd* (*primitive-part f*) (*primitive-part g*)

unfolding *gcd-primitive-part*
by (*metis* (*no-types*, *lifting*)
content-times-primitive-part gcd.normalize-idem mult-cancel-left2 mult-smult-left
normalize-eq-0-iff normalize-mult-unit-factor primitive-part-eq-0-iff
smult-content-normalize-primitive-part unit-factor-mult-normalize)

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 (*cases f* = *0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
have *normalize* (*content* (*normalize* (*primitive-part f*))) = *1*
using *content-primitive-part*[*OF False*] *content-dvd content-const*
content-dvd-contentI dvd-normalize-iff is-unit-content-iff **by** (*metis* (*no-types*))
then have *content* (*normalize* (*primitive-part f*)) = *1* **by** *fastforce*
then have *content* (*normalize f*) = *1* ∗ *content f*
by (*metis* (*no-types*) *content-smult mult.commute normalize-content*
smult-content-normalize-primitive-part)
then have *content f* = *content* (*normalize f*)
by *simp*
then show *?thesis* **unfolding** *smult-content-normalize-primitive-part*[*of f*,*symmetric*]
by (*metis* (*no-types*) *False content-times-primitive-part mult.commute mult-cancel-left*
mult-smult-right smult-content-normalize-primitive-part)
qed

lemma *length-coeffs-primitive-part*[*simp*]: *length* (*coeffs* (*primitive-part f*)) = *length*
(*coeffs f*)
proof (*cases f* = *0*)
case *False*
hence *length* (*coeffs f*) ≠ *0 length* (*coeffs* (*primitive-part f*)) ≠ *0* **by** *auto*
thus *?thesis* **using** *degree-primitive-part*[*of f*, *unfolded degree-eq-length-coeffs*] **by**
linarith
qed *simp*

lemma *degree-unit-factor*[*simp*]: *degree* (*unit-factor f*) = *0*
by (*simp add*: *monom-0 unit-factor-poly-def*)

lemma *degree-normalize*[*simp*]: *degree* (*normalize f*) = *degree f*
proof (*cases f* = *0*)
case *False*
have *degree f* = *degree* (*unit-factor f* ∗ *normalize f*) **by** *simp*
also have ... = *degree* (*unit-factor f*) + *degree* (*normalize f*)
by (*rule degree-mult-eq*, *insert False*, *auto*)
finally show *?thesis* **by** *simp*
qed *simp*

lemma *content-iff*: *x dvd content p* $\longleftrightarrow$ ($\forall$ *c* $\in$ *set* (*coeffs p*). *x dvd c*)
by (*simp add*: *content-def dvd-gcd-list-iff*)

lemma *is-unit-field-poly*[*simp*]: (*p*::'*a*::*field poly*) *dvd 1* $\longleftrightarrow$ *p* $\neq$ *0* $\wedge$ *degree p* = *0*
proof(*intro iffI conjI*, *unfold conj-imp-eq-imp-imp*)
assume *is-unit p*
then obtain *q* **where** $*$: *p* $*$ *q* = *1* **by** (*elim dvdE*, *auto*)
from $*$ **show** *p0*: *p* $\neq$ *0* **by** *auto*
from $*$ **have** *q0*: *q* $\neq$ *0* **by** *auto*
from $*$ *degree-mult-eq*[*OF p0 q0*]
show *degree p* = *0* **by** *auto*
next
assume *degree p* = *0*
from *degree0-coeffs*[*OF this*]
obtain *c* **where** *c*: *p* = [:*c*:] **by** *auto*
assume *p* $\neq$ *0*
with *c* **have** *c* $\neq$ *0* **by** *auto*
with *c* **have** *1* = *p* $*$ [:*1*/*c*:] **by** *auto*
from *dvdI*[*OF this*] **show** *is-unit p*.
qed

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

lemma *primitiveI*:
assumes ($\bigwedge$*x*. ($\bigwedge$*y*. *y* $\in$ *set* (*coeffs f*) $\Longrightarrow$ *x dvd y*) $\Longrightarrow$ *x dvd 1*)
shows *primitive f* **by** (*insert assms*, *auto simp*: *primitive-def*)

lemma *primitiveD*:
assumes *primitive f*
shows ($\bigwedge$*y*. *y* $\in$ *set* (*coeffs f*) $\Longrightarrow$ *x dvd y*) $\Longrightarrow$ *x dvd 1*
by (*insert assms*, *auto simp*: *primitive-def*)

lemma *not-primitiveE*:
assumes $\neg$ *primitive f*
and $\bigwedge$*x*. ($\bigwedge$*y*. *y* $\in$ *set* (*coeffs f*) $\Longrightarrow$ *x dvd y*) $\Longrightarrow$ $\neg$ *x dvd 1* $\Longrightarrow$ *thesis*
shows *thesis* **by** (*insert assms*, *auto simp*: *primitive-def*)

lemma *primitive-iff-content-eq-1*[*simp*]:
fixes *f* :: '*a* :: *semiring-gcd poly*
shows *primitive f* $\longleftrightarrow$ *content f* = *1*
proof(*intro iffI primitiveI*)
fix *x*
assume ($\bigwedge$*y*. *y* $\in$ *set* (*coeffs f*) $\Longrightarrow$ *x dvd y*)
from *gcd-list-greatest*[*of coeffs f*, *OF this*]
have *x dvd content f* **by** (*simp add*: *content-def*)
also assume *content f* = *1*
finally show *x dvd 1*.

next
assume *primitive f*
from *primitiveD*[*OF this list-gcd*[*of - coeffs f*], *folded content-def*]
show *content f = 1* **by** *simp*
qed

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 (*insert assms*, *induct fs arbitrary*: *f*)
case (*Cons f' fs*)
from *Cons.prems*
have *is-unit* (*content f'* ∗ *content* (*prod-list fs*)) **by** (*auto simp*: *content-mult*)
from *this*[*unfolded is-unit-mult-iff*]
have *content f' = 1* **and** *content* (*prod-list fs*) *= 1* **by** *auto*
moreover from *Cons.prems* **have** *f = f'* ∨ *f* ∈ *set fs* **by** *auto*
ultimately show *?case* **using** *Cons.hyps*[*of f*] **by** *auto*
qed *auto*

lemma *irreducible-imp-primitive*:
fixes *f* :: *'a* :: {*idom*,*semiring-gcd*} *poly*
assumes *irr*: *irreducible f* **and** *deg*: *degree f* ≠ *0* **shows** *primitive f*
proof (*rule ccontr*)
assume *not*: ¬ *?thesis*
then have ¬ [:*content f*:] *dvd 1* **by** *simp*
moreover have *f* = [:*content f*:] ∗ *primitive-part f* **by** *simp*
note *Factorial-Ring.irreducibleD*[*OF irr this*]
ultimately
have *primitive-part f dvd 1* **by** *auto*
from *this*[*unfolded poly-dvd-1*] **have** *degree f = 0* **by** *auto*
with *deg* **show** *False* **by** *auto*
qed

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
assume *l*: *?l* **show** *?r*
proof(*rule ccontr*, *elim not-irreducibleE*)
from *l* **have** *deg*: *degree f > 0* **by** (*auto dest*: *irreducible*$_d$*D*)
from *cf* **have** *f0*: *f* ≠ *0* **by** *auto*
then show *f = 0* ⟹ *False* **by** *auto*
show *f dvd 1* ⟹ *False* **using** *deg* **by** (*auto simp*:*poly-dvd-1*)
fix *a b* **assume** *fab*: *f = a* ∗ *b* **and** *a1*: ¬ *a dvd 1* **and** *b1*: ¬ *b dvd 1*
then have *af*: *a dvd f* **and** *bf*: *b dvd f* **by** *auto*
with *f0* **have** *a0*: *a* ≠ *0* **and** *b0*: *b* ≠ *0* **by** *auto*
from *irreducible*$_d$*D*(*2*)[*OF l*, *of a*] *af dvd-imp-degree-le*[*OF af f0*]
have *degree a = 0* ∨ *degree a = degree f*

```
    by (metis degree-smult-le irreducible_d-dvd-smult l le-antisym Nat.neq0-conv)
   then show False
   proof(elim disjE)
    assume degree a = 0
    then obtain c where ac: a = [:c:] by (auto dest: degree0-coeffs)
     from fab[unfolded ac] have c dvd content f by (simp add: content-iff co-
effs-smult)
    with cf have c dvd 1 by simp
    then have a dvd 1 by (auto simp: ac)
    with a1 show False by auto
   next
    assume dega: degree a = degree f
     with f0 degree-mult-eq[OF a0 b0] fab have degree b = 0 by (auto simp:
ac-simps)
    then obtain c where bc: b = [:c:] by (auto dest: degree0-coeffs)
     from fab[unfolded bc] have c dvd content f by (simp add: content-iff co-
effs-smult)
    with cf have c dvd 1 by simp
    then have b dvd 1 by (auto simp: bc)
    with b1 show False by auto
   qed
  qed
next
  assume r: ?r
  show ?l
  proof(intro irreducible_dI)
   show degree f > 0
   proof (rule ccontr)
    assume ¬degree f > 0
    then obtain f0 where f: f = [:f0:] by (auto dest: degree0-coeffs)
    from cf[unfolded this] have normalize f0 = 1 by auto
    then have f0 dvd 1 by (unfold normalize-1-iff)
    with r[unfolded f irreducible-const-poly-iff] show False by auto
   qed
  next
   fix g h assume deg-g: degree g > 0 and deg-gf: degree g < degree f and fgh: f
= g * h
   with r have g dvd 1 ∨ h dvd 1 by auto
   with deg-g have degree h = 0 by (auto simp: poly-dvd-1)
   with deg-gf[unfolded fgh] degree-mult-eq[of g h] show False by (cases g = 0 ∨
h = 0, auto)
  qed
qed

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 −
```

have *degree* (*normalize f*) > *0*
using *deg-f degree-normalize* **by** *auto*
hence *normalize f* ≠ *1*
by *fastforce*
thus ¬ *is-unit f* **using** *normalize-1-iff* **by** *auto*
qed

lemma *content-pCons*[*simp*]: *content* (*pCons a p*) = *gcd a* (*content p*)
proof(*induct p arbitrary*: *a*)
case *0* **show** *?case* **by** *simp*
next
case (*pCons c p*)
then show *?case* **by** (*cases p* = *0*, *auto simp*: *content-def cCons-def*)
qed

lemma *content-field-poly*:
fixes *f* :: *'a* :: {*field*,*semiring-gcd*} *poly*
shows *content f* = (*if f* = *0 then 0 else 1*)
by(*induct f*, *auto simp*: *dvd-field-iff is-unit-normalize*)

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*)
by (*simp add*: *primitive-part-def sdiv-poly-def*)

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*))
unfolding *Let-def rat-to-normalized-int-poly-def primitive-part-alt-def* **..**

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 −
let *?nds* = *map quotient-of xs*
define *nds* **where** *nds* = *?nds*
let *?denom* = *list-lcm* (*map snd nds*)
let *?ints* = *map* (λ (*n*,*d*). *n* ∗ *dd div d*) *nds*
from *cd*[*unfolded common-denom-def Let-def*]
have *dd*: *dd* = *?denom* **and** *ys*: *ys* = *?ints* **unfolding** *nds-def* **by** *auto*
show *dd0*: *dd* > *0* **unfolding** *dd*
by (*intro list-lcm-pos*(*3*), *auto simp*: *nds-def quotient-of-nonzero*)
{
fix *x*
assume *x*: *x* ∈ *set xs*
obtain *p q* **where** *quot*: *quotient-of x* = (*p*,*q*) **by** *force*
from *x* **have** (*p*,*q*) ∈ *set nds* **unfolding** *nds-def* **using** *quot* **by** *force*
hence *q* ∈ *set* (*map snd nds*) **by** *force*
from *list-lcm*[*OF this*] **have** *q*: *q dvd dd* **unfolding** *dd* **.**
show *rat-of-int* (*case quotient-of x of* (*n*, *x*) ⇒ *n* ∗ *dd div x*) / *rat-of-int dd* = *x*
unfolding *quot split* **unfolding** *quotient-of-div*[*OF quot*]
proof −
have *f1*: *q* ∗ (*dd div q*) = *dd*

```
    using dvd-mult-div-cancel q by blast
   have rat-of-int (dd div q) ≠ 0
    using dd0 dvd-mult-div-cancel q by fastforce
   thus rat-of-int (p * dd div q) / rat-of-int dd = rat-of-int p / rat-of-int q
    using f1 by (metis (no-types) div-mult-swap mult-divide-mult-cancel-right
of-int-mult q)
  qed
 } note main = this
 show xs = map (λ i. of-int i / of-int dd) ys unfolding ys map-map o-def nds-def
  by (rule sym, rule map-idI, rule main)
qed

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 −
 let ?f = λ x. case quotient-of x of (pa, x) ⇒ pa * d div x
 define f where f = ?f
 from assms[unfolded rat-to-int-poly-def Let-def]
  obtain xs where cd: common-denom (coeffs p) = (d,xs)
  and q: q = map-poly f p unfolding f-def by (cases common-denom (coeffs p),
auto)
 from common-denom[OF cd] have d: d > 0 and
  id: ⋀ x. x ∈ set (coeffs p) ⟹ rat-of-int (f x) / rat-of-int d = x
  unfolding f-def by auto
 have f0: f 0 = 0 unfolding f-def by auto
 have id: rat-of-int (f (coeff p n)) / rat-of-int d = coeff p n for n
  using id[of coeff p n] f0 range-coeff by (cases coeff p n = 0, auto)
 show d > 0 by fact
 show p = smult (inverse (of-int d)) (map-poly of-int q)
  unfolding q smult-as-map-poly using id f0
  by (intro poly-eqI, auto simp: field-simps coeff-map-poly)
qed

lemma content-ge-0-int: content p ≥ (0 :: int)
 unfolding content-def
 by (cases coeffs p, auto)

lemma abs-content-int[simp]: fixes p :: int poly
 shows abs (content p) = content p using content-ge-0-int[of p] by auto

lemma content-smult-int: fixes p :: int poly
 shows content (smult a p) = abs a * content p by simp

lemma normalize-non-0-smult: ∃ a. (a :: 'a :: semiring-gcd) ≠ 0 ∧ smult a
(primitive-part p) = p
 by (cases p = 0, rule exI[of - 1], simp, rule exI[of - content p], auto)

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 −
  have p = smult d (map-poly of-int q) ∧ d > 0 ∧ (p ≠ 0 ⟶ content q = 1)
  proof (cases p = 0)
    case True
    thus ?thesis using assms unfolding rat-to-normalized-int-poly-def
      by (auto simp: eval-poly-def)
  next
    case False
    hence p0: p ≠ 0 by auto
    obtain s r where id: rat-to-int-poly p = (s,r) by force
    let ?cr = rat-of-int (content r)
    let ?s = rat-of-int s
    let ?q = map-poly rat-of-int q
    from rat-to-int-poly[OF id] have p: p = smult (inverse ?s) (map-poly of-int r)
    and s: s > 0 by auto
    let ?q = map-poly rat-of-int q
    from p0 assms[unfolded rat-to-normalized-int-poly-def id split]
    have d: d = ?cr / ?s and q: q = primitive-part r by auto
    from content-times-primitive-part[of r, folded q] have qr: smult (content r) q
= r .
    have smult d ?q = smult (?cr / ?s) ?q
      unfolding d by simp
    also have ?cr / ?s = ?cr * inverse ?s by (rule divide-inverse)
    also have ... = inverse ?s * ?cr by simp
    also have smult (inverse ?s * ?cr) ?q = smult (inverse ?s) (smult ?cr ?q) by
simp
    also have smult ?cr ?q = map-poly of-int (smult (content r) q) by (simp add:
hom-distribs)
    also have ... = map-poly of-int r unfolding qr ..
    finally have pq: p = smult d ?q unfolding p by simp
    from p p0 have r0: r ≠ 0 by auto
    from content-eq-zero-iff[of r] content-ge-0-int[of r] r0 have cr: ?cr > 0 by
linarith
    with s have d0: d > 0 unfolding d by auto
    from content-primitive-part[OF r0] have cq: content q = 1 unfolding q .
    from pq d0 cq show ?thesis by auto
  qed
  thus p: p = smult d (map-poly of-int q) and d: d > 0 and p ≠ 0 ⟹ content
q = 1 by auto
  show degree q = degree p unfolding p smult-as-map-poly
    by (rule sym, subst map-poly-map-poly, force+, rule degree-map-poly, insert d,
auto)
qed

lemma content-dvd-1:
  content g = 1 if content f = (1 :: 'a :: semiring-gcd) g dvd f
proof −
  from ‹g dvd f› have content g dvd content f
```

by (*rule content-dvd-contentI*)
with ‹*content f = 1*› **show** *?thesis*
by *simp*
qed

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 (*cases p = 0*)
case *True* **thus** *?thesis* **by** *auto*
next
case *False* **note** *p0* = *this*
let *?cp* = *smult c p*
from *p0 c* **have** *cp0*: *?cp* ≠ *0* **by** *auto*
from *dvd* **obtain** *r* **where** *prod*: *?cp* = *q* ∗ *r* **unfolding** *dvd-def* **by** *auto*
from *prod cp0* **have** *q0*: *q* ≠ *0* **and** *r0*: *r* ≠ *0* **by** *auto*
let *?c* = *content* :: *int poly* ⇒ *int*
let *?n* = *primitive-part* :: *int poly* ⇒ *int poly*
let *?pn* = λ *p*. *smult* (*?c p*) (*?n p*)
have *cq*: (*?c q* = *0*) = *False* **using** *content-eq-zero-iff q0* **by** *auto*
from *prod* **have** *id1*: *?cp* = *?pn q* ∗ *?pn r* **unfolding** *content-times-primitive-part*
by *simp*
from *arg-cong*[*OF this*, *of content*, *unfolded content-smult-int content-mult*
content-primitive-part[*OF r0*] *content-primitive-part*[*OF q0*], *symmetric*]
p0[*folded content-eq-zero-iff*] *c*
have *abs c dvd ?c q* ∗ *?c r* **unfolding** *dvd-def* **by** *auto*
hence *c dvd ?c q* ∗ *?c r* **by** *auto*
then obtain *d* **where** *id*: *?c q* ∗ *?c r* = *c* ∗ *d* **unfolding** *dvd-def* **by** *auto*
have *?cp* = *?pn q* ∗ *?pn r* **by** *fact*
also have ... = *smult* (*c* ∗ *d*) (*?n q* ∗ *?n r*) **unfolding** *id* [*symmetric*]
by (*metis content-mult content-times-primitive-part primitive-part-mult*)
finally have *id*: *?cp* = *smult c* (*?n q* ∗ *smult d* (*?n r*)) **by** (*simp add*: *mult.commute*)
interpret *map-poly-inj-zero-hom* (∗) *c* **using** *c* **by** (*unfold-locales*, *auto*)
have *p* = *?n q* ∗ *smult d* (*?n r*) **using** *id*[*unfolded smult-as-map-poly*[*of c*]] **by**
auto
thus *dvd*: *?n q dvd p* **unfolding** *dvd-def* **by** *blast*
qed

lemma *irreducible*$_d$-*primitive-part*:
fixes *p* :: *int poly*
shows *irreducible*$_d$ (*primitive-part p*) ⟷ *irreducible*$_d$ *p* (**is** *?l* ⟷ *?r*)
proof (*rule iffI*, *rule irreducible*$_d$*I*)
assume *l*: *?l*
show *degree p* > *0* **using** *l* **by** *auto*
have *dpp*: *degree* (*primitive-part p*) = *degree p* **by** *simp*
fix *q r*
assume *deg*: *degree q* < *degree p degree r* < *degree p* **and** *p* = *q* ∗ *r*
then have *pp*: *primitive-part p* = *primitive-part q* ∗ *primitive-part r* **by** (*simp*
add: *primitive-part-mult*)

have ¬ *irreducible$_d$* (*primitive-part p*)
apply (*intro reducible$_d$I*, *rule exI*[*of - primitive-part q*], *rule exI*[*of - primitive-part r*], *unfold dpp*)
using *deg pp* **by** *auto*
with *l* **show** *False* **by** *auto*
next
show *?r* ⟹ *?l* **by** (*metis irreducible$_d$-smultI normalize-non-0-smult*)
qed

lemma *irreducible$_d$-smult-int*:
fixes *c* :: *int* **assumes** *c*: *c* ≠ *0*
shows *irreducible$_d$* (*smult c p*) = *irreducible$_d$ p* (**is** *?l* = *?r*)
using *irreducible$_d$-primitive-part*[*of smult c p*, *unfolded primitive-part-smult*] *c*
apply (*cases c* < *0*, *simp*)
apply (*metis add.inverse-inverse add.inverse-neutral c irreducible$_d$-smultI normalize-non-0-smult smult-1-left smult-minus-left*)
apply (*simp add*: *irreducible$_d$-primitive-part*)
done

lemma *irreducible$_d$-as-irreducible*:
fixes *p* :: *int poly*
shows *irreducible$_d$ p* ⟷ *irreducible* (*primitive-part p*)
using *irreducible-primitive-connect*[*of primitive-part p*]
by (*cases p* = *0*, *auto simp*: *irreducible$_d$-primitive-part*)

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 −
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
from *p* **have** *rp0*: *?rp p* ≠ *0* **by** *simp*
with *pgh* **have** *g0*: *g* ≠ *0* **and** *h0*: *h* ≠ *0* **by** *auto*
from *rat-to-normalized-int-poly*[*OF g*] *g0*
have *r*: *r* > *0 r* ≠ *0* **and** *g*: *g* = *smult r* (*?rp rg*) **and** *crg*: *content rg* = *1* **by** *auto*
from *rat-to-normalized-int-poly*[*OF h*] *h0*
have *s*: *s* > *0 s* ≠ *0* **and** *h*: *h* = *smult s* (*?rp sh*) **and** *csh*: *content sh* = *1* **by** *auto*
let *?irs* = *inverse* (*r* ∗ *s*)
from *r s* **have** *irs0*: *?irs* ≠ *0* **by** (*auto simp*: *field-simps*)
have *?rp* (*rg* ∗ *sh*) = *?rp rg* ∗ *?rp sh* **by** (*simp add*: *hom-distribs*)
also have ... = *smult ?irs* (*?rp p*) **unfolding** *pgh g h* **using** *r s*
by (*simp add*: *field-simps*)

```
  finally have id: ?rp (rg * sh) = smult ?irs (?rp p) by auto
  have rsZ: ?irs ∈ ℤ
  proof (rule ccontr)
    assume not: ¬ ?irs ∈ ℤ
    obtain n d where irs': quotient-of ?irs = (n,d) by force
    from quotient-of-denom-pos[OF irs'] have d > 0 .
    from not quotient-of-div[OF irs'] have d ≠ 1 d ≠ 0 and irs: ?irs = ?r n / ?r
d by auto
    with irs0 have n0: n ≠ 0 by auto
    from ‹d > 0› ‹d ≠ 1› have d ≥ 2 and ¬ d dvd 1 by auto
    with content-iff[of d p, unfolded cp] obtain c where
      c: c ∈ set (coeffs p) and dc: ¬ d dvd c
      by auto
    from c range-coeff[of p] obtain i where c = coeff p i by auto
    from arg-cong[OF id, of λ p. coeff p i,
      unfolded coeff-smult of-int-hom.coeff-map-poly-hom this[symmetric] irs]
    have ?r n / ?r d * ?r c ∈ ℤ by (metis Ints-of-int)
    also have ?r n / ?r d * ?r c = ?r (n * c) / ?r d by simp
    finally have inZ: ?r (n * c) / ?r d ∈ ℤ .
    have cop: coprime n d by (rule quotient-of-coprime[OF irs'])

    define prod where prod = ?r (n * c) / ?r d
    obtain n' d' where quot: quotient-of prod = (n',d') by force
    have qr: ⋀ x. quotient-of (?r x) = (x, 1)
      using Rat.of-int-def quotient-of-int by auto
    from quotient-of-denom-pos[OF quot] have d' > 0 .
    with quotient-of-div[OF quot] inZ[folded prod-def] have d' = 1
      by (metis Ints-cases Rat.of-int-def old.prod.inject quot quotient-of-int)
    with quotient-of-div[OF quot] have prod = ?r n' by auto
     from arg-cong[OF this, of quotient-of, unfolded prod-def rat-divide-code qr
Let-def split]
    have Rat.normalize (n * c, d) = (n',1) by simp
    from normalize-crossproduct[OF ‹d ≠ 0›, of 1 n * c n', unfolded this]
    have id: n * c = n' * d by auto
    from quotient-of-coprime[OF irs'] have coprime n d .
    with id have d dvd c
      by (metis coprime-commute coprime-dvd-mult-right-iff dvd-triv-right)
    with dc show False ..
  qed
  then obtain irs where irs: ?irs = ?r irs unfolding Ints-def by blast
  from id[unfolded irs, folded hom-distribs, unfolded of-int-poly-hom.eq-iff]
  have p: rg * sh = smult irs p by auto
  have content (rg * sh) = 1 unfolding content-mult crg csh by auto
  from this[unfolded p content-smult-int cp] have abs irs = 1 by simp
  hence abs ?irs = 1 using irs by auto
  with r s have ?irs = 1 by auto
  with irs have irs = 1 by auto
  with p show p: p = rg * sh by auto
qed
```

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 $\exists\ r.\ p = rg * smult\ (content\ p)\ r$
proof −
show *?thesis*
proof (*cases* $p = 0$)
case *True*
show *?thesis* **unfolding** *True*
by (*rule exI*[*of* - *0*], *auto simp*: *degree-monom-eq*)
next
case *False*
hence *p*: $p \neq 0$ **by** *auto*
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
define *q* **where** $q = primitive\text{-}part\ p$
from *content-times-primitive-part*[*of p*, *folded q-def*] *content-eq-zero-iff*[*of p*] *p*
obtain *a* **where** *a*: $a \neq 0$ **and** *pq*: $p = smult\ a\ q$ **and** *acp*: $content\ p = a$ **by** *metis*
from *a pq p* **have** *ra*: $?r\ a \neq 0$ **and** *q0*: $q \neq 0$ **by** *auto*
from *content-primitive-part*[*OF p*, *folded q-def*] **have** *cq*: $content\ q = 1$ **by** *auto*
obtain *s sh* **where** *h*: *rat-to-normalized-int-poly* (*smult* (*inverse* (*?r a*)) *h*) = (*s*,*sh*) **by** *force*
from *arg-cong*[*OF pgh*[*unfolded pq*], *of smult* (*inverse* (*?r a*))] *ra*
have *?rp q* = *g* ∗ *smult* (*inverse* (*?r a*)) *h* **by** (*auto simp*: *hom-distribs*)
from *rat-to-int-factor-content-1*[*OF cq this g h q0*]
have *qrs*: $q = rg * sh$ **.**
show *?thesis* **unfolding** *acp* **unfolding** *pq qrs*
by (*rule exI*[*of* - *sh*], *auto*)
qed
qed

lemma *rat-to-int-factor*: **fixes** p :: *int poly*
assumes *pgh*: *map-poly rat-of-int* $p = g * h$
shows $\exists\ g'\ h'.\ p = g' * h' \wedge degree\ g' = degree\ g \wedge degree\ h' = degree\ h$
proof(*cases* $p = 0$)
case *True*
with *pgh* **have** $g = 0 \vee h = 0$ **by** *auto*
then show *?thesis*
by (*metis True degree-0 mult-hom.hom-zero mult-zero-left rat-to-normalized-int-poly*(*4*) *surj-pair*)
next
case *False*
obtain *r rg* **where** *ri*: *rat-to-normalized-int-poly* (*smult* (*1 / of-int* (*content p*)) *g*) = (*r*,*rg*) **by** *force*
obtain *q qh* **where** *ri2*: *rat-to-normalized-int-poly h* = (*q*,*qh*) **by** *force*
show *?thesis*
proof (*intro exI conjI*)

have *of-int-poly* (*primitive-part p*) = *smult* (*1 / of-int* (*content p*)) (*g * h*)
apply (*auto simp*: *primitive-part-def pgh*[*symmetric*] *smult-map-poly map-poly-map-poly o-def intro*!: *map-poly-cong*)
by (*metis* (*no-types*, *lifting*) *content-dvd-coeffs div-by-0 dvd-mult-div-cancel floor-of-int nonzero-mult-div-cancel-left of-int-hom.hom-zero of-int-mult*)
also have ... = *smult* (*1 / of-int* (*content p*)) *g * h* **by** *simp*
finally have *of-int-poly* (*primitive-part p*) = ...
note *main* = *rat-to-int-factor-content-1*[*OF - this ri ri2*, *simplified*, *OF False*]
show *p* = *smult* (*content p*) *rg * qh* **by** (*simp add*: *main*[*symmetric*])
from *ri2* **show** *degree qh* = *degree h* **by** (*fact rat-to-normalized-int-poly*)
from *rat-to-normalized-int-poly*(*4*)[*OF ri*] *False*
show *degree* (*smult* (*content p*) *rg*) = *degree g* **by** *auto*
qed
qed

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 ∃ *g' h'*. *ip* = *g' * h'* ∧ *degree g'* = *degree g*
proof −
from *rat-to-normalized-int-poly*[*OF p*]
have *p*: *p* = *smult i* (*map-poly rat-of-int ip*) **and** *i*: *i* ≠ *0* **by** *auto*
from *arg-cong*[*OF p*, *of smult* (*inverse i*), *unfolded pgh*] *i*
have *map-poly rat-of-int ip* = *g * smult* (*inverse i*) *h* **by** *auto*
from *rat-to-int-factor*[*OF this*] **show** *?thesis* **by** *auto*
qed

lemma *irreducible-smult* [*simp*]:
fixes *c* :: *'a* :: *field*
shows *irreducible* (*smult c p*) ⟷ *irreducible p* ∧ *c* ≠ *0*
using *irreducible-mult-unit-left*[*of* [:*c*:], *simplified*] **by** *force*

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 (*rule irreducible*$_d$*I*)
from *irreducible*$_d$*D*[*OF p*]
have *p*: *degree p* ≠ *0* **and** *irr*: ⋀ *q r*. *degree q* < *degree p* ⟹ *degree r* < *degree p* ⟹ *p* ≠ *q * r* **by** *auto*
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
from *p* **show** *rp*: *degree* (*?rp p*) > *0* **by** *auto*
from *p* **have** *p0*: *p* ≠ *0* **by** *auto*
fix *g h* :: *rat poly*
assume *deg*: *degree g* > *0 degree g* < *degree* (*?rp p*) *degree h* > *0 degree h* < *degree* (*?rp p*) **and** *pgh*: *?rp p* = *g * h*

from *rat-to-int-factor*[*OF pgh*] **obtain** g' h' **where** *p*: $p = g' * h'$ **and** *dg*: *degree* $g' =$ *degree g degree* $h' =$ *degree h*
by *auto*
from *irr*[*of* g' h'] *deg*[*unfolded dg*]
show *False* **using** *degree-mult-eq*[*of* g' h'] **by** (*auto simp*: *p dg*)
qed

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 −
from *rat-to-normalized-int-poly*[*OF rp*]
have *rp*: *rp* = *smult a* (*map-poly rat-of-int ip*) **and** *a*: $a \neq 0$ **by** *auto*
with *irreducible*$_d$*-int-rat*[*OF ip*] **show** *?thesis* **by** *auto*
qed

lemma *dvd-content-dvd*: **assumes** *dvd*: *content f dvd content g primitive-part f dvd primitive-part g*
shows *f dvd g*
proof −
let *?cf* = *content f* **let** *?nf* = *primitive-part f*
let *?cg* = *content g* **let** *?ng* = *primitive-part g*
have *f dvd g* = (*smult ?cf ?nf dvd smult ?cg ?ng*)
unfolding *content-times-primitive-part* **by** *auto*
from *dvd*(*1*) **obtain** *ch* **where** *cg*: *?cg* = *?cf* ∗ *ch* **unfolding** *dvd-def* **by** *auto*
from *dvd*(*2*) **obtain** *nh* **where** *ng*: *?ng* = *?nf* ∗ *nh* **unfolding** *dvd-def* **by** *auto*
have *f dvd g* = (*smult ?cf ?nf dvd smult ?cg ?ng*)
unfolding *content-times-primitive-part*[*of f*] *content-times-primitive-part*[*of g*]
by *auto*
also have ... = (*smult ?cf ?nf dvd smult ?cf ?nf* ∗ *smult ch nh*) **unfolding** *cg ng*
by (*metis mult.commute mult-smult-right smult-smult*)
also have ... **by** (*rule dvd-triv-left*)
finally show *?thesis* **.**
qed

lemma *sdiv-poly-smult*: $c \neq 0 \Longrightarrow$ *sdiv-poly* (*smult c f*) *c* = *f*
by (*intro poly-eqI*, *unfold coeff-sdiv-poly coeff-smult*, *auto*)

lemma *primitive-part-smult-int*: **fixes** *f* :: *int poly* **shows**
primitive-part (*smult d f*) = *smult* (*sgn d*) (*primitive-part f*)
proof (*cases* $d = 0 \vee f = 0$)
case *False*
obtain *cf* **where** *cf*: *content f* = *cf* **by** *auto*
with *False* **have** *0*: $d \neq 0$ $f \neq 0$ $cf \neq 0$ **by** *auto*
show *?thesis*
proof (*rule poly-eqI*, *unfold primitive-part-alt-def coeff-sdiv-poly content-smult-int coeff-smult cf*)

```
    fix n
    consider (pos) d > 0 | (neg) d < 0 using 0(1) by linarith
    thus d * coeff f n div (|d| * cf) = sgn d * (coeff f n div cf)
    proof cases
      case neg
      hence ?thesis = (d * coeff f n div - (d * cf) = - (coeff f n div cf)) by auto
      also have d * coeff f n div - (d * cf) = - (d * coeff f n div (d * cf))
        by (subst dvd-div-neg, insert 0(1), auto simp: cf[symmetric])
      also have d * coeff f n div (d * cf) = coeff f n div cf using 0(1) by auto
      finally show ?thesis by simp
    qed auto
  qed
qed auto

lemma gcd-smult-left: assumes c ≠ 0
  shows gcd (smult c f) g = gcd f (g :: 'b :: {field-gcd} poly)
proof -
  from assms have normalize c = 1
    by (meson dvd-field-iff is-unit-normalize)
  then show ?thesis
   by (metis (no-types) Polynomial.normalize-smult gcd.commute gcd.left-commute
gcd-left-idem gcd-self smult-1-left)
qed

lemma gcd-smult-right: c ≠ 0 ⟹ gcd f (smult c g) = gcd f (g :: 'b :: {field-gcd}
poly)
  using gcd-smult-left[of c g f] by (simp add: gcd.commute)

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 (cases f = 0 ∧ g = 0)
  case True
  thus ?thesis by simp
next
  case False
  let ?r = rat-of-int
  let ?rp = map-poly ?r
  from False have gcd0: gcd f g ≠ 0 by auto
  hence lc0: lead-coeff (gcd f g) ≠ 0 by auto
  hence inv: inverse (?r (lead-coeff (gcd f g))) ≠ 0 by auto
  show ?thesis
  proof (rule sym, rule gcdI, goal-cases)
    case 1
    have gcd f g dvd f by auto
    then obtain h where f: f = gcd f g * h unfolding dvd-def by auto
    show ?case by (rule smult-dvd[OF - inv], insert arg-cong[OF f, of ?rp], simp
add: hom-distribs)
  next
    case 2
```

have *gcd f g dvd g* **by** *auto*
then obtain *h* **where** *g*: *g* = *gcd f g* ∗ *h* **unfolding** *dvd-def* **by** *auto*
show *?case* **by** (*rule smult-dvd*[*OF - inv*], *insert arg-cong*[*OF g*, *of ?rp*], *simp add*: *hom-distribs*)
next
case (*3 h*)
show *?case*
proof (*rule dvd-smult*)
obtain *ch ph* **where** *h*: *rat-to-normalized-int-poly h* = (*ch*, *ph*) **by** *force*
from *3* **obtain** *ff* **where** *f*: *?rp f* = *h* ∗ *ff* **unfolding** *dvd-def* **by** *auto*
from *3* **obtain** *gg* **where** *g*: *?rp g* = *h* ∗ *gg* **unfolding** *dvd-def* **by** *auto*
from *rat-to-int-factor-explicit*[*OF f h*] **obtain** *f′* **where** *f*: *f* = *ph* ∗ *f′* **by** *blast*
from *rat-to-int-factor-explicit*[*OF g h*] **obtain** *g′* **where** *g*: *g* = *ph* ∗ *g′* **by** *blast*
from *f g* **have** *ph dvd gcd f g* **by** *auto*
then obtain *gg* **where** *gcd*: *gcd f g* = *ph* ∗ *gg* **unfolding** *dvd-def* **by** *auto*
note ∗ = *rat-to-normalized-int-poly*[*OF h*]
show *h dvd ?rp* (*gcd f g*) **unfolding** *gcd* ∗(*1*)
by (*rule smult-dvd*, *insert* ∗(*2*), *auto*)
qed
next
case *4*
have [*simp*]: [:*1*:] = *1* **by** *simp*
show *?case* **unfolding** *normalize-poly-def*
by (*rule poly-eqI*, *simp*)
qed
qed

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*]
by *eval*

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*}
using *res n*
proof (*induct n arbitrary*: *ps rule*: *less-induct*)
case (*less n ps*)
obtain *n*′ *mo* **where** *dm*: *Euclidean-Rings.divmod-nat n i* = (*n*′,*mo*) **by** *force*
hence *n*′: *n*′ = *n div i* **and** *mo*: *mo* = *n mod i* **by** (*auto simp*: *Euclidean-Rings.divmod-nat-def*)
from *less*(*2*)[*unfolded remove-prime-factor.simps*[*of i n*] *dm*]
have *res*: (*if mo* = *0 then remove-prime-factor i n*′ (*i* # *ps*) *else* (*n*, *ps*)) = (*m*,
qs) **by** *auto*
from *less*(*3*) **have** *n*: *n* ≠ *0* **by** *auto*
with *n*′ *i* **have** *n*′ < *n* **by** *auto*
note *IH* = *less*(*1*)[*OF this*]
show *?case*
proof (*cases mo* = *0*)
case *True*
with *mo n*′ **have** *n*: *n* = *n*′ ∗ *i* **by** *auto*
with ‹*n* ≠ *0*› **have** *n*′: *n*′ ≠ *0* **by** *auto*
from *True res* **have** *remove-prime-factor i n*′ (*i* # *ps*) = (*m*,*qs*) **by** *auto*
from *IH*[*OF this n*′] **obtain** *rs* **where**
qs = *rs* @ *i* # *ps* **and** *n*′ = *m* ∗ *prod-list rs* ∧ ¬ *i dvd m* ∧ *set rs* ⊆ {*i*} **by**
auto
thus *?thesis*
by (*intro exI*[*of* - *rs* @ [*i*]], *unfold n*, *auto*)
next
case *False*

```
    with mo have i-n: ¬ i dvd n by auto
    from False res have id: m = n qs = ps by auto
    show ?thesis unfolding id using i-n by auto
  qed
qed

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) unfolding prime-nat-iff
proof (intro conjI impI allI)
  show 1 < n using n by auto
  fix j
  assume jn: j dvd n
  from jn obtain k where njk: n = j * k unfolding dvd-def by auto
  with ‹1 < n› have jn: j ≤ n by (metis dvd-imp-le jn neq0-conv not-less0)
  show j = 1 ∨ j = n
  proof (rule ccontr)
    assume ¬ ?thesis
    with njk n have j1: j > 1 ∧ j ≠ n by simp
    have ∃ j k. 1 < j ∧ j ≤ k ∧ n = j * k
    proof (cases j ≤ k)
      case True
      thus ?thesis unfolding njk using j1 by blast
    next
      case False
     show ?thesis by (rule exI[of - k], rule exI[of - j], insert ‹1 < n› j1 njk False,
auto)
        (metis Suc-lessI mult-0-right neq0-conv)
    qed
    then obtain j k where j1: 1 < j and jk: j ≤ k and njk: n = j * k by auto
    with small[of j] have ji: j ≥ i unfolding dvd-def by force
    from mult-mono[OF ji ji] have i * i ≤ j * j by auto
    with i have j * j > n by auto
    from this[unfolded njk] have k < j by auto
    with jk show False by auto
  qed
qed

lemma candidate-invariant-0: candidate-invariant 0
  unfolding candidate-invariant-def by auto

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
  unfolding candidate-invariant-def
proof −
  note res = res[unfolded next-candidates-def]
```

```
  note n = n[unfolded candidate-invariant-def]
  show m = 0 ∨ m mod 30 = 11 using res n by (auto split: if-splits)
 show sorted ps using res n by (auto split: if-splits simp: primes-1000-def sorted2-simps
simp del: sorted-wrt.simps(2))
  show set ps ⊆ {2..} ∩ {n..<m} using res n by (auto split: if-splits simp:
primes-1000-def)
  show distinct ps using res n by (auto split: if-splits simp: primes-1000-def)
  show ps ≠ [] using res n by (auto split: if-splits simp: primes-1000-def)
  show n < m using res by (auto split: if-splits)
  show {i. prime i ∧ n ≤ i ∧ i < m} ⊆ set ps
  proof (cases n = 0)
    case True
    hence *: m = 1001 ps = primes-1000 using res by auto
    show ?thesis unfolding * True primes-1000 by auto
  next
    case False
   hence n: n mod 30 = 11 and m: m = n + 30 and ps: ps = [n,n+2,n+6,n+8,n+12,n+18,n+20,n+26]

      using res n by auto
    {
      fix i
      assume *: prime i n ≤ i i < n + 30 i ∉ set ps
      from n * have i11: i ≥ 11 by auto
      define j where j = i − n
      have i: i = n + j using ‹n ≤ i› j-def by auto
      have i mod 30 = (j + n) mod 30 using ‹n ≤ i› unfolding j-def by simp
      also have ... = (j mod 30 + n mod 30) mod 30
        by (simp add: mod-simps)
      also have ... = (j mod 30 + 11) mod 30 unfolding n by simp
      finally have i30: i mod 30 = (j mod 30 + 11) mod 30 by simp
      have 2: 2 dvd (30 :: nat) and 112: 11 mod (2 :: nat) = 1 by simp-all
      have (j + 11) mod 2 = (j + 1) mod 2
        by (rule mod-add-cong) simp-all
      with arg-cong [OF i30, of λj. j mod 2]
      have 2: i mod 2 = (j mod 2 + 1) mod 2
        by (simp add: mod-simps mod-mod-cancel [OF 2])
      have 3: 3 dvd (30 :: nat) and 113: 11 mod (3 :: nat) = 2 by simp-all
      have (j + 11) mod 3 = (j + 2) mod 3
        by (rule mod-add-cong) simp-all
      with arg-cong [OF i30, of λ j. j mod 3] have 3: i mod 3 = (j mod 3 + 2)
mod 3
        by (simp add: mod-simps mod-mod-cancel [OF 3])
      have 5: 5 dvd (30 :: nat) and 115: 11 mod (5 :: nat) = 1 by simp-all
      have (j + 11) mod 5 = (j + 1) mod 5
        by (rule mod-add-cong) simp-all
      with arg-cong [OF i30, of λ j. j mod 5] have 5: i mod 5 = (j mod 5 + 1)
mod 5
        by (simp add: mod-simps mod-mod-cancel [OF 5])
```

```
    from n *(2-)[unfolded ps i, simplified] have
      j ∈ {1,3,5,7,9,11,13,15,17,19,21,23,25,27,29} ∨ j ∈ {4,10,16,22,28} ∨
j ∈ {14,24}
      (is j ∈ ?j2 ∨ j ∈ ?j3 ∨ j ∈ ?j5)
      by simp presburger
    moreover
    {
      assume j ∈ ?j2
      hence j mod 2 = 1 by auto
      with 2 have i mod 2 = 0 by auto
      with i11 have 2 dvd i i ≠ 2 by auto
      with *(1) have False unfolding prime-nat-iff by auto
    }
    moreover
    {
      assume j ∈ ?j3
      hence j mod 3 = 1 by auto
      with 3 have i mod 3 = 0 by auto
      with i11 have 3 dvd i i ≠ 3 by auto
      with *(1) have False unfolding prime-nat-iff by auto
    }
    moreover
    {
      assume j ∈ ?j5
      hence j mod 5 = 4 by auto
      with 5 have i mod 5 = 0 by auto
      with i11 have 5 dvd i i ≠ 5 by auto
      with *(1) have False unfolding prime-nat-iff by auto
    }
    ultimately have False by blast
  }
  thus ?thesis unfolding m ps by auto
 qed
qed


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
 assume dvd: j dvd n
 with small[OF j(1)] have j ≥ i by linarith
 with dvd mod have j > i by (cases i = j, auto)
 with j have j = Suc i by simp
```

with *i* **have** *even j* **by** *auto*
with *dvd j*(*1*) **have** *2 dvd n* **by** (*metis dvd-trans*)
with *odd* **show** *False* **by** *auto*
qed

lemma *prime-divisor*: **assumes** $j \geq 2$ **and** *j dvd n* **shows**
$\exists\ p :: nat.\ prime\ p \wedge p\ dvd\ j \wedge p\ dvd\ n$
proof −
let *?pf* = *prime-factors j*
from *assms* **have** $j > 0$ **by** *auto*
from *prime-factorization-nat*[*OF this*]
have $j = (\prod p \in ?pf.\ p\ \hat{}\ multiplicity\ p\ j)$ **by** *auto*
with ‹$j \geq 2$› **have** $?pf \neq \{\}$ **by** *auto*
then obtain *p* **where** *p*: $p \in ?pf$ **by** *auto*
hence *pr*: *prime p* **by** *auto*
define *rem* **where** $rem = (\prod p \in ?pf - \{p\}.\ p\ \hat{}\ multiplicity\ p\ j)$
from *p* **have** *mult*: $multiplicity\ p\ j \neq 0$
by (*auto simp*: *prime-factors-multiplicity*)
have *finite ?pf* **by** *simp*
have $j = (\prod p \in ?pf.\ p\ \hat{}\ multiplicity\ p\ j)$ **by** *fact*
also have $?pf = (insert\ p\ (?pf - \{p\}))$ **using** *p* **by** *auto*
also have $(\prod p \in insert\ p\ (?pf - \{p\}).\ p\ \hat{}\ multiplicity\ p\ j) =$
$p\ \hat{}\ multiplicity\ p\ j * rem$ **unfolding** *rem-def*
by (*subst prod.insert*, *auto*)
also have $\ldots = p * (p\ \hat{}\ (multiplicity\ p\ j - 1) * rem)$ **using** *mult*
by (*cases multiplicity p j*, *auto*)
finally have *pj*: *p dvd j* **unfolding** *dvd-def* **by** *blast*
with ‹*j dvd n*› **have** *p dvd n* **by** (*metis dvd-trans*)
with *pj pr* **show** *?thesis* **by** *blast*
qed

lemma *prime-nat-main*: $ni = (n,i,is) \Longrightarrow i \geq 2 \Longrightarrow n \geq 2 \Longrightarrow$
$(\bigwedge j.\ 2 \leq j \Longrightarrow j < i \Longrightarrow \neg\ (j\ dvd\ n)) \Longrightarrow$
$(\bigwedge j.\ i \leq j \Longrightarrow j < jj \Longrightarrow prime\ j \Longrightarrow j \in set\ is) \Longrightarrow i \leq jj \Longrightarrow$
$sorted\ is \Longrightarrow distinct\ is \Longrightarrow candidate\text{-}invariant\ jj \Longrightarrow set\ is \subseteq \{i..<jj\} \Longrightarrow$
$res = prime\text{-}nat\text{-}main\ n\ jj\ is \Longrightarrow$
$res = prime\ n$
proof (*induct ni arbitrary*: *n i is jj res rule*: *wf-induct*[*OF*
wf-measures[*of* [λ (*n*,*i*,*is*). *n* − *i*, λ (*n*,*i*,*is*). *if is* = [] *then 1 else 0*]]])
case (*1 ni n i is jj res*)
note *res* = *1*(*12*)
from *1*(*3*−*4*) **have** *i*: $i \geq 2$ **and** *i2*: $Suc\ i \geq 2$ **and** *n*: $n \geq 2$ **by** *auto*
from *1*(*5*) **have** *dvd*: $\bigwedge j.\ 2 \leq j \Longrightarrow j < i \Longrightarrow \neg\ j\ dvd\ n$ **.**
from *1*(*7*) **have** *ijj*: $i \leq jj$ **.**
note *sort-dist* = *1*(*8*−*9*)
have *is*: $\bigwedge j.\ i \leq j \Longrightarrow j < jj \Longrightarrow prime\ j \Longrightarrow j \in set\ is$ **by** (*rule 1*(*6*))
note *simps* = *prime-nat-main.simps*[*of n jj is*]
note *IH* = *1*(*1*)[*rule-format*, *unfolded 1*(*2*), *OF - refl*]
show *?case*

```
  proof (cases is)
    case Nil
    obtain jjj iis where can: next-candidates jj = (jjj,iis) by force
   from res[unfolded simps, unfolded Nil can split] have res: res = prime-nat-main
n jjj iis by auto
    from next-candidates[OF can 1(10)] have can:
      sorted iis distinct iis candidate-invariant jjj
      {i. prime i ∧ jj ≤ i ∧ i < jjj} ⊆ set iis set iis ⊆ {2..} ∩ {jj..<jjj}
      iis ≠ [] jj < jjj by blast+
    from can ijj have i ≤ jjj by auto
    note IH = IH[OF - i n dvd - this can(1−3) - res]
    show ?thesis
    proof (rule IH, force simp: Nil can(6))
      fix x
      assume ix: i ≤ x and xj: x < jjj and px: prime x
      from is[OF ix - px] Nil have jx: jj ≤ x by force
      with can(4) xj px show x ∈ set iis by auto
    qed (insert can(5) ijj, auto)
  next
    case (Cons i' iis)
    with res[unfolded simps]
    have res: res = (if i' dvd n then n ≤ i' else if i' * i' ≤ n then prime-nat-main
n jj iis else True)
      by simp
    from 1(11) Cons have iis: set iis ⊆ {i..<jj} and i': i ≤ i' i' < jj Suc i' ≤ jj
by auto
   from sort-dist have sd-iis: sorted iis distinct iis and i' ∉ set iis by(auto simp:
Cons)
    from sort-dist(1) have set iis ⊆ {i'..} by(auto simp: Cons)
    with iis have set iis ⊆ {i'..<jj} by force
    with ‹i' ∉ set iis›  have iis: set iis ⊆ {Suc i'..<jj}
      by (auto, case-tac x = i', auto)
    {
      fix j
      assume j: 2 ≤ j j < i'
      have ¬ j dvd n
      proof
        assume j dvd n
        from prime-divisor[OF j(1) this] obtain p where
          p: prime p p dvd j p dvd n by auto
        have pj: p ≤ j
          by (rule dvd-imp-le[OF p(2)], insert j, auto)
        have p2: 2 ≤ p using p(1) by (rule prime-ge-2-nat)
        from dvd[OF p2] p(3) have pi: p ≥ i by force
        from pj j(2) i' is[OF pi - p(1)] have p ∈ set is by auto
        with ‹sorted is› have i' ≤ p by(auto simp: Cons)
        with pj j(2) show False by arith
      qed
    } note dvd = this
```

```
from i' i have i'2: 2 ≤ Suc i' by auto
note IH = IH[OF - i'2 n - - i'(3) sd-iis 1(10) iis]
show ?thesis
proof (cases i' dvd n)
  case False note dvdi = this
  {
    fix j
    assume j: 2 ≤ j j < Suc i'
    have ¬ j dvd n
    proof (cases j = i')
      case False
      with j have j < i' by auto
      from dvd[OF j(1) this] show ?thesis .
    qed (insert False, auto)
  } note dvds = this
  show ?thesis
  proof (cases i' * i' ≤ n)
    case True note iin = this
    with res False have res: res = prime-nat-main n jj iis by auto
    from iin have i-n: i' < n
      using dvd dvdi n nat-neq-iff dvd-refl by blast
    {
      fix x
      assume Suc i' ≤ x x < jj prime x
      hence i ≤ x x < jj prime x using i' by auto
      from is[OF this] have x ∈ set is .
      with ‹Suc i' ≤ x› have x ∈ set iis unfolding Cons by auto
    } note iis = this
    show ?thesis
      by (rule IH[OF - dvds iis res], insert i-n i', auto)
  next
    case False
    with res dvdi have res: res = True by auto
    have n: prime n
      by (rule prime-sqrtI[OF n dvd False])
    thus ?thesis unfolding res by auto
  qed
next
  case True
  have i' ≥ 2 using i i' by auto
  from ‹i' dvd n› obtain k where n = i' * k ..
  with n have k ≠ 0 by (cases k = 0, auto)
  with ‹n = i' * k› have *: i' < n ∨ i' = n
    by auto
  with True res have res ⟷ i' = n
    by auto
  also have ... = prime n
  using * proof
    assume i' < n
```

```
      with ‹i' ≥ 2› ‹i' dvd n› have ¬ prime n
        by (auto simp add: prime-nat-iff)
      with ‹i' < n› show ?thesis
        by auto
    next
      assume i' = n
      with dvd n have prime n
        by (simp add: prime-nat-iff')
      with ‹i' = n› show ?thesis
        by auto
    qed
    finally show ?thesis .
  qed
 qed
qed

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 (induct ni arbitrary: n i is jj res ps rule: wf-induct[OF
  wf-measures[of [λ (n,i,is). n - i, λ (n,i,is). if is = [] then 1 else 0]]])
  case (1 ni n i is jj res ps)
  note res = 1(12)
  from 1(3-4) have i: i ≥ 2 and i2: Suc i ≥ 2 and n: n ≥ 2 by auto
  from 1(5) have dvd: ⋀ j. 2 ≤ j ⟹ j < i ⟹ ¬ j dvd n .
  from 1(7) have ijj: i ≤ jj .
  note sort-dist = 1(8-9)
  have is: ⋀ j. i ≤ j ⟹ j < jj ⟹ prime j ⟹ j ∈ set is by (rule 1(6))
  note simps = prime-factorization-nat-main.simps[of n jj is]
  note IH = 1(1)[rule-format, unfolded 1(2), OF - refl]
  show ?case
  proof (cases is)
    case Nil
    obtain jjj iis where can: next-candidates jj = (jjj,iis) by force
   from res[unfolded simps, unfolded Nil can split] have res: res = prime-factorization-nat-main
n jjj iis ps by auto
    from next-candidates[OF can 1(10)] have can:
      sorted iis distinct iis candidate-invariant jjj
      {i. prime i ∧ jj ≤ i ∧ i < jjj} ⊆ set iis set iis ⊆ {2..} ∩ {jj..<jjj}
      iis ≠ [] jj < jjj by blast+
    from can ijj have i ≤ jjj by auto
    note IH = IH[OF - i n dvd - this can(1-3) - res]
    show ?thesis
    proof (rule IH, force simp: Nil can(6))
      fix x
      assume ix: i ≤ x and xj: x < jjj and px: prime x
```

```
    from is[OF ix - px] Nil have jx: jj ≤ x by force
    with can(4) xj px show x ∈ set iis by auto
  qed (insert can(5) ijj, auto)
 next
  case (Cons i' iis)
  obtain n' m where dm: Euclidean-Rings.divmod-nat n i' = (n',m) by force
 hence n': n' = n div i' and m: m = n mod i' by (auto simp: Euclidean-Rings.divmod-nat-def)
  have m: (m = 0) = (i' dvd n) unfolding m by auto
  from Cons res[unfolded simps] dm m n'
  have res: res = (
    if i' dvd n then case remove-prime-factor i' (n div i') (i' # ps) of
        (n', ps') ⇒ if n' = 1 then ps' else prime-factorization-nat-main n' jj iis
ps'
    else if i' * i' ≤ n then prime-factorization-nat-main n jj iis ps else n # ps)
    by simp
  from 1(11) i Cons have iis: set iis ⊆ {i..<jj} and i': i ≤ i' i' < jj Suc i' ≤
jj i' > 1 by auto
  from sort-dist have sd-iis: sorted iis distinct iis and i' ∉ set iis by(auto simp:
Cons)
  from sort-dist(1) Cons have set iis ⊆ {i'..} by(auto)
  with iis have set iis ⊆ {i'..<jj} by force
  with ‹i' ∉ set iis› have iis: set iis ⊆ {Suc i'..<jj}
    by (auto, case-tac x = i', auto)
  {
    fix j
    assume j: 2 ≤ j j < i'
    have ¬ j dvd n
    proof
      assume j dvd n
      from prime-divisor[OF j(1) this] obtain p where
        p: prime p p dvd j p dvd n by auto
      have pj: p ≤ j
        by (rule dvd-imp-le[OF p(2)], insert j, auto)
      have p2: 2 ≤ p using p(1) by (rule prime-ge-2-nat)
      from dvd[OF p2] p(3) have pi: p ≥ i by force
      from pj j(2) i' is[OF pi - p(1)] have p ∈ set is by auto
      with ‹sorted is› have i' ≤ p by (auto simp: Cons)
      with pj j(2) show False by arith
    qed
  } note dvd = this
  from i' i have i'2: 2 ≤ Suc i' by auto
  note IH = IH[OF - i'2 - - - i'(3) sd-iis 1(10) iis]
  {
    fix x
    assume Suc i' ≤ x x < jj prime x
    hence i ≤ x x < jj prime x using i' by auto
    from is[OF this] have x ∈ set is .
    with ‹Suc i' ≤ x› have x ∈ set iis unfolding Cons by auto
  } note iis = this
```

```
    show ?thesis
    proof (cases i' dvd n)
      case False note dvdi = this
      {
        fix j
        assume j: 2 ≤ j j < Suc i'
        have ¬ j dvd n
        proof (cases j = i')
          case False
          with j have j < i' by auto
          from dvd[OF j(1) this] show ?thesis .
        qed (insert False, auto)
      } note dvds = this
      show ?thesis
      proof (cases i' * i' ≤ n)
        case True note iin = this
        with res False have res: res = prime-factorization-nat-main n jj iis ps by
auto
        from iin have i-n: i' < n using dvd dvdi n nat-neq-iff dvd-refl by blast
        show ?thesis
          by (rule IH[OF - n dvds iis res], insert i-n i', auto)
      next
        case False
        with res dvdi have res: res = n # ps by auto
        have n: prime n
          by (rule prime-sqrtI[OF n dvd False])
        thus ?thesis unfolding res by auto
      qed
    next
      case True note i-n = this
     obtain n'' qs where rp: remove-prime-factor i' (n div i') (i' # ps) = (n'',qs)
by force
      with res True
      have res: res = (if n'' = 1 then qs else prime-factorization-nat-main n'' jj iis
qs) by auto
      have pi: prime i' unfolding prime-nat-iff
      proof (intro conjI allI impI)
        show 1 < i' using i' i by auto
        fix j
        assume ji: j dvd i'
        with i' i have j0: j ≠ 0 by (cases j = 0, auto)
        from ji i-n have jn: j dvd n by (metis dvd-trans)
        with dvd[of j] have j: 2 > j ∨ j ≥ i' by linarith
        from ji ‹1 < i'› have j ≤ i' unfolding dvd-def
          by (simp add: dvd-imp-le ji)
        with j j0 show j = 1 ∨ j = i' by linarith
      qed
      from True n' have id: n = n' * i' by auto
      from n id have n' ≠ 0 by (cases n = 0, auto)
```

```
      with id have i' ≤ n by auto
      from remove-prime-factor[OF rp[folded n']  ‹1 < i'› ‹n' ≠ 0›] obtain rs
        where qs: qs = rs @ i' # ps and n': n' = n'' * prod-list rs and i-n'': ¬ i'
dvd n''
        and rs: set rs ⊆ {i'} by auto
      {
        fix j
        assume j dvd n''
        hence j dvd n unfolding id n' by auto
      } note dvd' = this
      show ?thesis
      proof (cases n'' = 1)
        case False
        with res have res: res = prime-factorization-nat-main n'' jj iis qs
          by simp
        from i i' have i' ≥ 2 by simp
        from False n' ‹n' ≠ 0› have n2: n'' ≥ 2 by (cases n'' = 0; auto)
         have lrs: prod-list rs ≠ 0 using n' ‹n' ≠ 0› by (cases prod-list rs = 0,
auto)
        with ‹i' ≥ 2› have prod-list rs * i' ≥ 2 by (cases prod-list rs, auto)
        hence nn'': n > n'' unfolding id n' using n2 by simp
        have i' ≠ n unfolding id n' using pi False by fastforce
        with ‹i' ≤ n› i' have n > i by auto
        with nn'' i i' have less: n - i > n'' - Suc i' by simp
        {
          fix j
          assume 2: 2 ≤ j and ji: j < Suc i'
          have ¬ j dvd n''
          proof (cases j = i')
            case False
            with ji have j < i' by auto
            from dvd' dvd[OF 2 this] show ?thesis by blast
          qed (insert i-n'', auto)
        }
        from IH[OF - n2 this iis res] less obtain ss where
          res: res = ss @ qs ∧ Ball (set ss) prime ∧ n'' = prod-list ss by auto
        thus ?thesis unfolding id n' qs using pi rs by auto
      next
        case True
        with res have res: res = qs by auto
        show ?thesis unfolding id n' res qs True using rs ‹prime i'›
          by (intro exI[of - rs @ [i']], auto)
      qed
    qed
  qed
qed

lemma prime-nat[simp]: prime-nat n = prime n
proof (cases n < 2)
```

```
  case True
  thus ?thesis unfolding prime-nat-def prime-nat-iff by auto
next
  case False
  hence n: n ≥ 2 by auto
  obtain jj is where can: next-candidates 0 = (jj,is) by force
  from next-candidates[OF this candidate-invariant-0]
  have cann: sorted is distinct is candidate-invariant jj
    {i. prime i ∧ 0 ≤ i ∧ i < jj} ⊆ set is
    set is ⊆ {2..} ∩ {0..<jj} distinct is is ≠ [] by auto
  from cann have sub: set is ⊆ {2..<jj} by force
  with ‹is ≠ []› have jj: jj ≥ 2 by (cases is, auto)
  from n can have res: prime-nat n = prime-nat-main n jj is
    unfolding prime-nat-def by auto
  show ?thesis using prime-nat-main[OF refl le-refl n - - jj cann(1−3) sub res]
cann(4) by auto
qed

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 −
  note pf = pf-def[unfolded prime-factorization-nat-def]
  have Ball (set pf) prime ∧ (n ≠ 0 ⟶ prod-list pf = n) ∧ (n = 0 ⟶ pf = [])
  proof (cases n < 2)
    case True
    thus ?thesis using pf by auto
  next
    case False
    hence n: n ≥ 2 by auto
    obtain jj is where can: next-candidates 0 = (jj,is) by force
    from next-candidates[OF this candidate-invariant-0]
    have cann: sorted is distinct is candidate-invariant jj
      {i. prime i ∧ 0 ≤ i ∧ i < jj} ⊆ set is
      set is ⊆ {2..} ∩ {0..<jj} distinct is is ≠ [] by auto
    from cann have sub: set is ⊆ {2..<jj} by force
    with ‹is ≠ []› have jj: jj ≥ 2 by (cases is, auto)
    let ?pfm = prime-factorization-nat-main n jj is []
    from pf[unfolded can] False
    have res: pf = rev ?pfm by simp
    from prime-factorization-nat-main[OF refl le-refl n - - jj cann(1−3) sub refl,
of Nil] cann(4)
    have Ball (set ?pfm) prime n = prod-list ?pfm by auto
    thus ?thesis unfolding res using n by auto
  qed
  thus Ball (set pf) prime n ≠ 0 ⟹ prod-list pf = n n = 0 ⟹ pf = [] by auto
qed
```

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

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 −
  have *prod-mset* $A \neq 0$ **by** (*auto dest*: *assms*(*1*))
  with *assms*(*2*) **have** $x \neq 0$ **by** *simp*
  hence *prod-mset* (*prime-factorization* x) $=$ *prod-mset* A
    by (*simp add*: *assms prod-mset-prime-factorization*)
  with *assms* **show** *?thesis*
    by (*intro prime-factorization-unique'*) *auto*
qed

lemma *multiset-prime-factorization-nat-correct*:
  prime-factorization $n =$ *mset* (*prime-factorization-nat* n)
proof −
  note *pf* $=$ *prime-factorization-nat*[*of* n]
  show *?thesis*
  proof (*cases* $n = 0$)
    case *True*
    thus *?thesis* **using** *pf*(*3*) **by** *simp*
  next
    case *False*
    note *pf* $=$ *pf*(*1*) *pf*(*2*)[*OF False*]
    show *?thesis*
    proof (*rule prime-factorization-unique''*)
      show *prime p* **if** $p \in\#$ *mset* (*prime-factorization-nat* n) **for** p
        using *pf*(*1*) *that* **by** *simp*
      let *?l* $= \prod i \in\#$ *prime-factorization* $n.\ i$
      let *?r* $= \prod i \in\#$ *mset* (*prime-factorization-nat* n). i
      show *prod-mset* (*mset* (*prime-factorization-nat* n)) $=$ *normalize* n
        by (*simp add*: *pf*(*2*) *prod-mset-prod-list*)
    qed
  qed
qed

lemma *multiset-prime-factorization-code*[*code-unfold*]:
  prime-factorization $= (\lambda n.$ *mset* (*prime-factorization-nat* n))
  by (*intro ext multiset-prime-factorization-nat-correct*)

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

```
0 = []
proof −
  show distinct (divisors-nat n) divisors-nat 0 = [] unfolding divisors-nat-def by
auto
  assume n: n ≠ 0
  from n have n > 0 by auto
  {
    fix x
    have (x dvd n) = (x ≠ 0 ∧ (∀ p. multiplicity p x ≤ multiplicity p n))
    proof (cases x = 0)
      case False
      with ‹n > 0› show ?thesis by (auto simp: dvd-multiplicity-eq)
    next
      case True
      with n show ?thesis by auto
    qed
  } note dvd = this
  let ?dn = set (divisors-nat n)
  let ?mf = λ (n :: nat). prime-factorization n
  have ?dn = prod-list ‘ set (subseqs (prime-factorization-nat n)) unfolding divi-
sors-nat-def
    using n by auto
  also have ... = prod-mset ‘ mset ‘ set (subseqs (prime-factorization-nat n))
    by (force simp: prod-mset-prod-list)
  also have mset ‘ set (subseqs (prime-factorization-nat n))
    = {# ps. ps ⊆# mset (prime-factorization-nat n)#}
    unfolding multiset-of-subseqs by simp
  also have ... = {# ps. ps ⊆# ?mf n#}
    thm multiset-prime-factorization-code[symmetric]
    unfolding multiset-prime-factorization-nat-correct[symmetric] by auto
  also have prod-mset ‘ ... = {p. p dvd n} (is ?l = ?r)
  proof −
    {
      fix x
      assume x dvd n
      from this[unfolded dvd] have x: x ≠ 0 by auto
      from ‹x dvd n› ‹x ≠ 0› ‹n ≠ 0› have sub: ?mf x ⊆# ?mf n
        by (subst prime-factorization-subset-iff-dvd) auto
      have prod-mset (?mf x) = x using x
        by (simp add: prime-factorization-nat)
      hence x ∈ ?l using sub by force
    }
    moreover
    {
      fix x
      assume x ∈ ?l
      then obtain ps where x: x = prod-mset ps and sub: ps ⊆# ?mf n by auto
      have x dvd n using prod-mset-subset-imp-dvd[OF sub] n x by simp
    }
```

```
    ultimately show ?thesis by blast
  qed
  finally show set (divisors-nat n) = {p. p dvd n} .
qed

lemma divisors-int-pos: x ≠ 0 ⟹ set (divisors-int-pos x) = {i. i dvd x ∧ i > 0}
distinct (divisors-int-pos x)
  divisors-int-pos 0 = []
proof −
  show divisors-int-pos 0 = [] by code-simp
  show distinct (divisors-int-pos x)
    unfolding divisors-int-pos-def using divisors-nat(2)[of nat (abs x)]
    by (simp add: distinct-map inj-on-def)
  assume x: x ≠ 0
  let ?x = nat (abs x)
  from x have xx: ?x ≠ 0 by auto
  from x have 0: ⋀ y. y dvd x ⟹ y ≠ 0 by auto
  have id: int ' {p. int p dvd x} = {i. i dvd x ∧ 0 < i} (is ?l = ?r)
  proof −
    {
      fix y
      assume y ∈ ?l
      then obtain p where y: y = int p and dvd: int p dvd x by auto
      have y ∈ ?r unfolding y using dvd 0[OF dvd] by auto
    }
    moreover
    {
      fix y
      assume y ∈ ?r
      hence dvd: y dvd x and y0: y > 0 by auto
      define n where n = nat y
      from y0 have y: y = int n unfolding n-def by auto
      with dvd have y ∈ ?l by auto
    }
    ultimately show ?thesis by blast
  qed
  from xx show set (divisors-int-pos x) = {i. i dvd x ∧ i > 0}
    by (simp add: divisors-int-pos-def divisors-nat id)
qed

lemma divisors-int: x ≠ 0 ⟹ set (divisors-int x) = {i. i dvd x} distinct (divisors-int
x)
  divisors-int 0 = []
proof −
  show divisors-int 0 = [] by code-simp
  show distinct (divisors-int x)
  proof (cases x = 0)
    case True
    show ?thesis unfolding True by code-simp
```

next
case *False*
from *divisors-int-pos(1)*[*OF False*] *divisors-int-pos(2)*
show *?thesis* **unfolding** *divisors-int-def Let-def distinct-append distinct-map inj-on-def* **by** *auto*
qed
assume *x*: $x \neq 0$
show *set* (*divisors-int x*) = {*i*. *i dvd x*}
unfolding *divisors-int-def Let-def set-append set-map divisors-int-pos(1)*[*OF x*]
using *x*
by *auto* (*metis* (*no-types*, *lifting*) *dvd-mult-div-cancel image-eqI linorder-neqE-linordered-idom*

mem-Collect-eq minus-dvd-iff minus-minus mult-zero-left neg-less-0-iff-less)
qed

definition *divisors-fun* :: ('*a* ⇒ ('*a* :: {*comm-monoid-mult*,*zero*}) *list*) ⇒ *bool* **where**
divisors-fun df ≡ (∀ *x*. *x* ≠ *0* ⟶ *set* (*df x*) = { *d*. *d dvd x* }) ∧ (∀ *x*. *distinct* (*df x*))

lemma *divisors-funD*: *divisors-fun df* ⟹ *x* ≠ *0* ⟹ *d dvd x* ⟹ *d* ∈ *set* (*df x*)
unfolding *divisors-fun-def* **by** *auto*

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*)
unfolding *divisors-pos-fun-def* **by** *auto*

lemma *divisors-fun-nat*: *divisors-fun divisors-nat*
unfolding *divisors-fun-def* **using** *divisors-nat* **by** *auto*

lemma *divisors-fun-int*: *divisors-fun divisors-int*
unfolding *divisors-fun-def* **using** *divisors-int* **by** *auto*

lemma *divisors-pos-fun-int*: *divisors-pos-fun divisors-int-pos*
unfolding *divisors-pos-fun-def* **using** *divisors-int-pos* **by** *auto*

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* ⟹ *poly p x* = *0*
divisors-fun df ⟹ *divisors-pos-fun dp* ⟹ *rational-root-test-main df dp p* = *None* ⟹ ¬ (∃ *x*. *poly p x* = *0*)
proof −
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
obtain *a ip* **where** *rp*: *rat-to-normalized-int-poly p* = (*a*,*ip*) **by** *force*
from *rat-to-normalized-int-poly*[*OF this*] **have** *p*: *p* = *smult a* (*?rp ip*) **and** *a00*: *a* ≠ *0*
and *cip*: *p* ≠ *0* ⟹ *content ip* = *1* **by** *auto*
let *?a0* = *coeff ip 0*
let *?an* = *coeff ip* (*degree ip*)
let *?d0* = *df ?a0*
let *?dn* = *dp ?an*
let *?ip* = *?rp ip*
define *tests* **where** *tests* = [*rat-of-int b0* / *rat-of-int bn* . *b0* <− *?d0*, *bn* <− *?dn*, *coprime b0 bn*]
let *?f* = (λ *x*. (*x*,*poly p x*))
let *?test* = (λ (-, *res*). *res* = *0*)
define *mo* **where** *mo* = *find-map-filter ?f ?test tests*
note *d* = *rational-root-test-main-def*[*of df dp p*, *unfolded Let-def rp snd-conv mo-def*[*symmetric*] *tests-def*[*symmetric*]]
{
assume *rational-root-test-main df dp p* = *Some x*
from *this*[*unfolded d*] **have** *?a0* = *0* ∧ *x* = *0* ∨ *map-option fst mo* = *Some x* **by** (*auto split*: *if-splits*)

thus *poly p x = 0*
proof
assume ∗: *?a0 = 0 ∧ x = 0*
hence *coeff p 0 = 0* **unfolding** *p coeff-smult* **by** *simp*
hence *poly p 0 = 0* **by** (*cases p*, *auto*)
with ∗ **show** *?thesis* **by** *auto*
next
assume *map-option fst mo = Some x*
then obtain *pair* **where** *find*: *find-map-filter ?f ?test tests = Some pair* **and** *x*: *x = fst pair*
unfolding *mo-def* **by** (*auto split*: *option.splits*)
then obtain *z* **where** *pair*: *pair = (x,z)* **by** (*cases pair*, *auto*)
from *find-map-filter-Some*[*OF find*, *unfolded pair split*] **show** *poly p x = 0* **by** *auto*
qed
}
assume *df*: *divisors-fun df* **and** *dp*: *divisors-pos-fun dp* **and** *res*: *rational-root-test-main df dp p = None*
note *df = divisors-funD*[*OF df*] **note** *dp = divisors-pos-funD*[*OF dp*]
from *res*[*unfolded d*] **have** *a0*: *?a0 ≠ 0* **and** *res*: *map-option fst mo = None* **by** (*auto split*: *if-splits*)
from *res*[*unfolded mo-def*] **have** *find*: *find-map-filter ?f ?test tests = None* **by** *auto*
show ¬ (∃ *x*. *poly p x = 0*)
proof
assume ∃ *x*. *poly p x = 0*
then obtain *x* **where** *poly p x = 0* **by** *auto*
from *this*[*unfolded p*] *a00* **have** *poly* (*?rp ip*) *x = 0* **by** *auto*
from *this*[*unfolded poly-eq-0-iff-dvd*] **have** [: −*x* , *1* :] *dvd ?ip* **by** *auto*
then obtain *q* **where** *ip-id*: *?ip* = [: −*x*, *1* :] ∗ *q* **unfolding** *dvd-def* **by** *auto*
obtain *c q* **where** *x1*: *rat-to-normalized-int-poly* [: −*x*, *1* :] = (*c*, *q*) **by** *force*
from *rat-to-int-factor-explicit*[*OF ip-id x1*] **obtain** *r* **where** *ip*: *ip = q ∗ r* **by** *blast*
from *rat-to-normalized-int-poly*(*4*)[*OF x1*] **have** *deg*: *degree q = 1* **by** *auto*
from *degree1-coeffs*[*OF deg*] **obtain** *a b* **where** *q*: *q* = [: *b*, *a* :] **and** *a*: *a ≠ 0* **by** *metis*
have *ipr*: *ip* = [: *b*, *a* :] ∗ *r* **using** *ip q* **by** *auto*
from *arg-cong*[*OF ipr*, *of* λ *p*. *coeff p 0*] **have** *ba0*: *b dvd ?a0* **by** *auto*
have *rpq*: *?rp q* = [: *?r b*, *?r a* :] **unfolding** *q*
proof (*rule poly-eqI*, *unfold of-int-hom.coeff-map-poly-hom*)
fix *n*
show *?r* (*coeff* [:*b*, *a*:] *n*) = *coeff* [: *?r b*, *?r a*:] *n*
unfolding *coeff-pCons*
by (*cases n*, *force*, *cases n − 1*, *auto*)
qed
from *arg-cong*[*OF ip*, *of ?rp*, *unfolded of-int-poly-hom.hom-mult rpq*] **have** [: *?r b*, *?r a* :] *dvd ?rp ip*
unfolding *dvd-def* **by** *blast*
hence *smult* (*inverse* (*?r a*)) [: *?r b* , *?r a* :] *dvd ?rp ip*

```
    by (rule smult-dvd, insert a, auto)
  also have smult (inverse (?r a)) [: ?r b , ?r a :] = [: ?r b / ?r a, 1 :] using a
    by (simp add: field-simps)
  finally have [: − (− ?r b / ?r a), 1 :] dvd ?rp ip by simp
  from this[unfolded poly-eq-0-iff-dvd[symmetric]]
  have rt: poly (?rp ip) (− ?r b / ?r a) = 0 .
  hence rt: poly p (− ?r b / ?r a) = 0
    unfolding p using a00 by simp
  obtain aa bb where quot: quotient-of (− ?r b / ?r a) = (bb,aa) by force
  hence quotient-of (?r (−b) / ?r a) = (bb, aa) by simp
  from quotient-of-int-div[OF this ‹a ≠ 0›] obtain z where
    z: z ≠ 0 and b: − b = z * bb and a: a = z * aa by auto
  from rt[unfolded quotient-of-div[OF quot]] have rt: poly p (?r bb / ?r aa) = 0
by auto
  from quotient-of-coprime[OF quot] have cop: coprime bb aa coprime (− bb) aa
by auto
  from quotient-of-denom-pos[OF quot] have aa: aa > 0 by auto
  from ba0 arg-cong[OF b, of uminus] z have bba0: bb dvd ?a0 unfolding dvd-def
    by (metis ba0 dvdE dvd-mult-right minus-dvd-iff)
  hence bb0: bb ≠ 0 using a0 by auto
  from df[OF a0 bba0] have bb: bb ∈ set ?d0 by auto
  from a0 have ip0: ip ≠ 0 by auto
  hence an0: ?an ≠ 0 by auto
  from ipr ip0 have r ≠ 0 by auto
  from degree-mult-eq[OF - this, of [:b,a:], folded ipr] ‹a ≠ 0› ipr
  have deg: degree ip = Suc (degree r) by auto
  from arg-cong[OF ipr, of λ p. coeff p (degree ip)] have ba0: a dvd ?an
    unfolding deg by (auto simp: coeff-eq-0)
  hence aa dvd ?an using ‹a ≠ 0› unfolding a by (auto simp: dvd-def)
  from dp[OF an0 this aa] have aa: aa ∈ set ?dn .
  from find-map-filter-None[OF find] rt have (?r bb / ?r aa) ∉ set tests by auto
  note test = this[unfolded tests-def, simplified, rule-format, of - aa]
  from this[of bb] cop bb aa
  show False by auto
 qed
qed

lemma rational-root-test:
  rational-root-test p = Some x ⟹ poly p x = 0
  rational-root-test p = None ⟹ ¬ (∃ x. poly p x = 0)
  using rational-root-test-main(1) rational-root-test-main(2)[OF divisors-fun-int
divisors-pos-fun-int]
  unfolding rational-root-test-def by blast+


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*) **unfolding** *kronecker-samples-def*
by *auto*

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*
unfolding *divisors-nat-def*[*abs-def*] *divisors-nat-copy-def*[*abs-def*] **..**

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

lemma *memo-divisors-nat*[*code-unfold*]: *divisors-nat* = *memo-divisors-nat*
unfolding *memo-divisors-nat-def* **by** *simp*

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 −
define *xs* **where** *xs* = *map* (*snd* ∘ *quotient-of*) (*coeffs* (*map-poly rat-of-int p*))
have *xs*: *set xs* ⊆ {*1*} **unfolding** *xs-def* **by** *auto*
from *assms*[*unfolded rat-to-int-poly-def Let-def*]
have *c*: *c* = *fst* (*common-denom* (*coeffs* (*map-poly rat-of-int p*))) **by** *auto*
also have ... = *list-lcm xs*
unfolding *common-denom-def Let-def xs-def* **by** (*simp add*: *o-assoc*)
also have ... = *1* **using** *xs*
by (*induct xs*, *auto*)
finally show *c*: *c* = *1* **by** *auto*
from *rat-to-int-poly*[*OF rp*, *unfolded c*] **show** *q* = *p* **by** *auto*
qed

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 −
obtain *d r* **where** *ri*: *rat-to-int-poly* (*map-poly rat-of-int p*) = (*d*,*r*) **by** *force*
from *rat-to-int-poly-of-int*[*OF ri*]
assms[*unfolded rat-to-normalized-int-poly-def ri split*]

```
  show c ∈ ℤ p ≠ 0 ⟹ c = of-int (content p) ∧ q = primitive-part p
    by (auto split: if-splits)
qed

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 −
  let ?r = rat-of-int
  let ?rp = map-poly ?r
  show ?thesis
  proof
    assume x dvd y
    then obtain z where y = x * z unfolding dvd-def by auto
    from arg-cong[OF this, of ?rp]
    show ?rp x dvd ?rp y by auto
  next
    assume dvd: ?rp x dvd ?rp y
    show x dvd y
    proof (cases y = 0)
      case True
      thus ?thesis by auto
    next
      case False note y0 = this
      hence ?rp y ≠ 0 by simp
      hence rx0: ?rp x ≠ 0 using dvd by auto
      hence x0: x ≠ 0 by simp
       from dvd obtain z where prod: ?rp y = ?rp x * z unfolding dvd-def by
auto
      obtain cx xx where x: rat-to-normalized-int-poly (?rp x) = (cx, xx) by force
      from rat-to-int-factor-explicit[OF prod x] obtain z where y: y = xx * smult
(content y) z by auto
       from rat-to-normalized-int-poly[OF x] rx0 have xx: ?rp x = smult cx (?rp
xx)
        and cxx: content xx = 1 and cx0: cx > 0 by auto
      obtain cn cd where quot: quotient-of cx = (cn,cd) by force
      from quotient-of-div[OF quot] have cx: cx = ?r cn / ?r cd by auto
      from quotient-of-denom-pos[OF quot] have cd0: cd > 0 by auto
      with cx cx0 have cn0: cn > 0 by (simp add: zero-less-divide-iff)
      from arg-cong[OF xx, of smult (?r cd)] have smult (?r cd) (?rp x) = smult
(?r cn) (?rp xx)
        unfolding cx using cd0 by (auto simp: field-simps)
       from this have id: smult cd x = smult cn xx by (fold hom-distribs, unfold
of-int-poly-hom.eq-iff)
      from arg-cong[OF this, of content, unfolded content-smult-int cxx] cn0 cd0
      have cn: cn = cd * content x by auto
      from quotient-of-coprime[OF quot, unfolded cn] cd0 have cd = 1 by auto
      with cx have cx: cx = ?r cn by auto
      from xx[unfolded this] have x: x = smult cn xx by (fold hom-distribs, simp)
       from arg-cong[OF this, of content, unfolded content-smult-int c-x cxx] cn0
```

have *cn = 1* **by** *auto*
with *x* **have** *xx*: *xx = x* **by** *auto*
show *x dvd y* **using** *y*[*unfolded xx*] **unfolding** *dvd-def* **by** *blast*
qed
qed
qed

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

lemma *length-upto-add-nat*[*simp*]: *length* [*a* .. *a* + *int n*] = *Suc n*
proof (*induct n arbitrary*: *a*)
case (*0 a*)
show *?case* **using** *upto.simps*[*of a a*] **by** *auto*
next
case (*Suc n a*)
from *Suc*[*of a + 1*]
show *?case* **using** *upto.simps*[*of a a + int* (*Suc n*)] **by** (*auto simp*: *ac-simps*)
qed

lemma *kronecker-samples*: *distinct* (*kronecker-samples n*) *length* (*kronecker-samples n*) = *Suc n*
unfolding *kronecker-samples-def Let-def length-upto-add-nat* **by** *auto*

lemma *dvd-int-poly-non-0-degree-1*[*simp*]: *degree q* ≥ *1* ⟹ *dvd-int-poly-non-0 q p* = (*q dvd p*)
by (*intro dvd-int-poly-non-0*, *auto*)

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 −
let *?r = rat-of-int*
let *?rp = map-poly ?r*
note *res = some*[*unfolded kronecker-factorization-main-def Let-def*]
from *res* **have** *dp*: *degree p* ≥ *2* **and** (*degree p* ≤ *1*) = *False* **by** (*auto split*: *if-splits*)
note *res = res*[*unfolded this if-False*]
note *bnd = bnd*[*OF dp*]
define *P* **where** *P = primitive-part p*
have *degP*: *degree P = degree p* **unfolding** *P-def* **by** *simp*
define *js* **where** *js = kronecker-samples bnd*

define *filt* **where** *filt* = (*case-option False* (λ*g*. *dvd-int-poly-non-0 g P* ∧ *1* ≤ *degree g*))
define *tests* **where** *tests* = *concat-lists* (*map* (λ(*v*, *j*). *map* (*Pair j*) (*if j* = *0 then dp v else df v*)) (*map* (λ*j*. (*poly P j*, *j*)) *js*))
note *res* = *res*[*folded P-def*, *folded js-def filt-def*, *folded tests-def*]
let *?zero* = *map* (λ*j*. (*poly P j*, *j*)) *js*
from *res* **have** *res*: (*case map-of ?zero 0 of*
None ⇒ *map-option the* (*find-map-filter newton-interpolation-poly-int filt tests*)
| *Some j* ⇒ *Some* [:− *j*, *1*:]) =
Some q **by** *auto*
have *degree q* ≥ *1* ∧ *degree q* ≤ *bnd* ∧ *q dvd P*
proof (*cases map-of ?zero 0*)
case (*Some j*)
with *res* **have** *q*: *q* = [: − *j*, *1* :] **by** *auto*
from *map-of-SomeD*[*OF Some*] **have** *0*: *poly P j* = *0* **by** *auto*
hence *poly* (*?rp P*) (*?r j*) = *0* **by** *simp*
hence [: − *?r j*, *1* :] *dvd ?rp P* **using** *poly-eq-0-iff-dvd* **by** *blast*
also have [: − *?r j*, *1* :] = *?rp q* **unfolding** *q* **by** *simp*
finally have *dvd*: *?rp q dvd ?rp P* **.**
have *q dvd P*
by (*subst dvd-poly-int-content-1*, *insert dvd q*, *auto*)
with *q dp bnd* **show** *?thesis* **by** *auto*
next
case *None*
from *res*[*unfolded None*]
have *res*: *map-option the* (*find-map-filter newton-interpolation-poly-int filt tests*) = *Some q* **by** *auto*
then obtain *qq* **where**
res: *find-map-filter newton-interpolation-poly-int filt tests* = *Some qq* **and** *q*: *q* = *the qq*
by (*auto split*: *option.splits*)
from *find-map-filter-Some*[*OF res*]
have *filt*: *filt qq* **and** *tests*: *qq* ∈ *newton-interpolation-poly-int* ‘ *set tests* **by** *auto*
from *filt*[*unfolded filt-def*] *q* **obtain** *g* **where** *dvd*: *g dvd P* **and** *dg*: *1* ≤ *degree g* **and** *qq*: *qq* = *Some g*
by (*cases qq*, *auto*)
from *q qq* **have** *gq*: *g* = *q* **by** *auto*
from *tests* **obtain** *t* **where** *t*: *t* ∈ *set tests* **and** *l*: *newton-interpolation-poly-int t* = *Some g* **unfolding** *qq*
by *auto*
from *t*[*unfolded tests-def*]
have *lent*: *length t* = *length js* **and** ⋀ *i*. *i* < *length js* ⟹ *map fst t* ! *i* = *js* ! *i* **by** *auto*
hence *id*: *map fst t* = *js*
by (*intro nth-equalityI*, *auto*)
have *dist*: *distinct js* **and** *lenj*: *length js* = *Suc bnd* **unfolding** *js-def degP*
using *kronecker-samples* **by** *auto*
from *newton-interpolation-poly-int-Some*[*OF dist*[*folded id*] *l*, *unfolded lent lenj*]
have *degree g* ≤ *bnd* **by** *auto*

with *dvd dg* **show** *?thesis* **unfolding** *gq* **by** *auto*
qed note *main* = *this*
thus *degree q* ≥ *1 degree q* ≤ *bnd* **by** *auto*
from *content-times-primitive-part*[*of p*] **have** *p* = *smult* (*content p*) *P* **unfolding** *P-def* **by** *auto*
with *main* **show** *q dvd p* **by** (*metis dvd-smult*)
qed

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 −
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
let *?p* = *rat-to-normalized-int-poly p*
obtain *a P* **where** *rp*: *?p* = (*a*,*P*) **by** *force*
from *rat-to-normalized-int-poly*[*OF this*] **have** *p*: *p* = *smult a* (*?rp P*) **and** *a*: *a* ≠ *0*
and *deg*: *degree P* = *degree p* **by** *auto*
from *some*[*unfolded kronecker-factorization-rat-main-def rp*]
obtain *Q* **where** *some*: *kronecker-factorization-main df dp bnd P* = *Some Q* **and** *q*: *q* = *?rp Q* **by** *auto*
from *kronecker-factorization-main-sound*[*OF some bnd*] **have** *dQ*: *1* ≤ *degree Q*
degree Q ≤ *bnd*
and *dvd*: *Q dvd P* **unfolding** *deg* **by** *auto*
from *dvd* **obtain** *R* **where** *PQR*: *P* = *Q* ∗ *R* **unfolding** *dvd-def* **by** *auto*
from *p*[*unfolded arg-cong*[*OF this*, *of ?rp*]]
have *p* = *q* ∗ *smult a* (*?rp R*) **unfolding** *q* **by** (*auto simp*: *hom-distribs*)
thus *q dvd p* **unfolding** *dvd-def* **by** *blast*
from *q dQ* **show** *degree q* ≥ *1 degree q* ≤ *bnd* **by** *auto*
qed

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 −
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
from *dp* **have** (*degree p* ≤ *1*) = *False* **by** *auto*
note *res* = *none*[*unfolded kronecker-factorization-main-def Let-def this if-False*]
define *P* **where** *P* = *primitive-part p*

have *degP*: *degree P* = *degree p* **unfolding** *P-def* **by** *simp*
define *js* **where** *js* = *kronecker-samples bnd*
define *filt* **where** *filt* = (*case-option False* (λ*g*. *dvd-int-poly-non-0 g P* ∧ 1 ≤ *degree g*))
define *tests* **where** *tests* = *concat-lists* (*map* (λ(*v*, *j*). *map* (*Pair j*) (*if j* = 0 *then dp v else df v*)) (*map* (λ*j*. (*poly P j*, *j*)) *js*))
note *res* = *res*[*folded P-def*, *folded js-def filt-def*, *folded tests-def*]
let *?zero* = *map* (λ*j*. (*poly P j*, *j*)) *js*
from *res* **have** *res*: (*case map-of ?zero 0 of*
None ⇒ *map-option the* (*find-map-filter newton-interpolation-poly-int filt tests*)
| *Some j* ⇒ *Some* [:− *j*, 1:]) =
None **by** *auto*
hence *zero*: *map-of ?zero 0* = *None* **by** (*auto split*: *option.splits*)
with *res* **have** *res*: *find-map-filter newton-interpolation-poly-int filt tests* = *None* **by** *auto*
{
fix *qq*
assume *qq*: 1 ≤ *degree qq degree qq* ≤ *bnd* **and** *dvd*: *qq dvd p*
define *q*′ **where** *q*′ = *primitive-part qq*
define *q* **where** *q* = (*if poly q*′ 0 > 0 *then q*′ *else* −*q*′)
from *qq* **have** *q*′: 1 ≤ *degree q*′ *degree q*′ ≤ *bnd* **unfolding** *q*′*-def* **by** *auto*
hence *q*: 1 ≤ *degree q degree q* ≤ *bnd* **unfolding** *q-def* **by** *auto*
from *dvd* **have** *qq dvd* (*smult* (*content p*) *P*)
using *content-times-primitive-part*[*of p*] **unfolding** *P-def* **by** *simp*
from *dvd-smult-int*[*OF - this*] *dp* **have** *q*′ *dvd P* **unfolding** *q*′*-def*
by *force*
hence *dvd*: *q dvd P* **unfolding** *q-def* **by** *auto*
then obtain *r* **where** *P*: *P* = *q* ∗ *r* **unfolding** *dvd-def* **by** *auto*
{
fix *j*
assume *j*: *j* ∈ *set js*
from *P* **have** *id*: *poly P j* = *poly q j* ∗ *poly r j* **by** *auto*
hence *dvd*: *poly q j dvd poly P j* **by** *auto*
from *j* **have** (*poly P j*, *j*) ∈ *set ?zero* **by** *auto*
with *zero* **have** *zero*: *poly P j* ≠ 0 **unfolding** *map-of-eq-None-iff* **by** *force*
with *id* **have** *poly q j* ≠ 0 **by** *auto*
hence *j* = 0 ⟹ *poly q j* > 0 **unfolding** *q-def* **by** *auto*
from *divisors-funD*[*OF df zero dvd*] *divisors-pos-funD*[*OF dpf zero dvd this*]
have *poly q j* ∈ *set* (*df* (*poly P j*)) *j* = 0 ⟹ *poly q j* ∈ *set* (*dp* (*poly P j*)) **.**
} note *mem1* = *this*
define *t* **where** *t* = *map* (λ *j*. (*j*, *poly q j*)) *js*
have *t*: *t* ∈ *set tests* **unfolding** *tests-def concat-lists-listset listset length-map map-map o-def*
proof (*rule*, *intro conjI allI impI*)
show *length t* = *length js* **unfolding** *t-def* **by** *simp*
fix *i*
assume *i*: *i* < *length js*
hence *jsi*: *js* ! *i* ∈ *set js* **by** *auto*
have *ti*: *t* ! *i* = (*js* ! *i*, *poly q* (*js* ! *i*)) **unfolding** *t-def* **using** *i* **by** *auto*

let *?f* = (λ*x*. *set* (*case* (*poly P x*, *x*) *of* (*v*, *j*) ⇒ *map* (*Pair j*) (*if j* = *0 then dp v else df v*)))
show *t* ! *i* ∈ *map ?f js* ! *i*
unfolding *ti nth-map*[*OF i*] *split* **using** *mem1*[*OF jsi*] **by** *auto*
qed
have *dist*: *distinct js* **and** *lenj*: *length js* = *Suc bnd* **unfolding** *js-def degP*
using *kronecker-samples* **by** *auto*
have *map-fst*: *map fst t* = *js* **unfolding** *t-def*
by (*rule nth-equalityI*, *auto*)
with *dist* **have** *dist*: *distinct* (*map fst t*) **by** *simp*
from *lenj q degP* **have** *degq*: *degree q* < *length t* **unfolding** *t-def* **by** *auto*
from *find-map-filter-None*[*OF res*] *t*
have *nfilt*: ¬ *filt* (*newton-interpolation-poly-int t*) **by** *auto*
have *qt*: ⋀*x y*. (*x*, *y*) ∈ *set t* ⟹ *poly q x* = *y* **unfolding** *t-def* **by** *auto*
from *interpolation-poly-int-None*[*OF dist - qt degq*, *of Newton*] **have**
newton-interpolation-poly-int t ≠ *None* **by** *auto*
then obtain *g* **where** *lt*: *newton-interpolation-poly-int t* = *Some g* **by** *auto*
from *newton-interpolation-poly-int-Some*[*OF dist lt*]
have *gt*: ⋀ *x y*. (*x*, *y*) ∈ *set t* ⟹ *poly g x* = *y* **and** *degg*: *degree g* < *length t*
using *degq* **by** *auto*
from *uniqueness-of-interpolation-point-list*[*OF dist qt degq gt degg*]
have *g*: *g* = *q* **by** *auto*
from *nfilt*[*unfolded lt g*] **have** ¬ *filt* (*Some q*) **.**
from *this*[*unfolded filt-def*] *q dvd* **have** *False* **by** *auto*
} note *main* = *this*
thus *?thesis* **by** *auto*
qed

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
assume ∃ *q*. *1* ≤ *degree q* ∧ *degree q* ≤ *bnd* ∧ *q dvd p*
then obtain *q* **where** *q*: *1* ≤ *degree q degree q* ≤ *bnd* **and** *dvd*: *q dvd p* **by** *auto*
from *dvd* **obtain** *r* **where** *prod*: *p* = *q* ∗ *r* **unfolding** *dvd-def* **by** *auto*
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
let *?p* = *rat-to-normalized-int-poly p*
obtain *a P* **where** *rp*: *?p* = (*a*,*P*) **by** *force*
from *rat-to-normalized-int-poly*[*OF this*] **have** *deg*: *degree P* = *degree p* **by** *auto*
from *rat-to-int-factor-normalized-int-poly*[*OF prod rp*]
obtain *g*′ **where** *dvd*: *g*′ *dvd P* **and** *dg*: *degree g*′ = *degree q* **by** (*auto intro*: *dvdI*)
have *kronecker-factorization-main df dp bnd P* = *None*
using *none*[*unfolded kronecker-factorization-rat-main-def rp*] **by** *auto*
from *kronecker-factorization-main-complete*[*OF this dp*[*folded deg*]] *dg dvd q* **show**
False **by** *auto*
qed

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 ⟹ irreducible$_d$ p
proof −
note *d = kronecker-factorization-def*
{
assume *kronecker-factorization p = Some q*
from *kronecker-factorization-main-sound*[*OF this*[*unfolded d*]]
show *degree q ≥ 1 ∧ degree q < degree p ∧ q dvd p* **by** *auto linarith*
}
assume *kf*: *kronecker-factorization p = None* **and** *deg*: *degree p ≥ 1*
show *irreducible$_d$ p*
proof (*cases degree p = 1*)
case *True*
thus *?thesis* **by** (*rule linear-irreducible$_d$*)
next
case *False*
with *deg* **have** *degree p ≥ 2* **by** *auto*
with *kronecker-factorization-main-complete*[*OF divisors-fun-int divisors-pos-fun-int kf*[*unfolded d*] *this*]
show *?thesis*
by (*intro irreducible$_d$I2*, *auto*)
qed
qed

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 ⟹ irreducible$_d$ p
proof −
note *d = kronecker-factorization-rat-def*
{
assume *kronecker-factorization-rat p = Some q*
from *kronecker-factorization-rat-main-sound*[*OF this*[*unfolded d*]]
show *degree q ≥ 1 ∧ degree q < degree p ∧ q dvd p* **by** *auto linarith*
}
assume *kf*: *kronecker-factorization-rat p = None* **and** *deg*: *degree p ≥ 1*
show *irreducible$_d$ p*
proof (*cases degree p = 1*)
case *True*
thus *?thesis* **by** (*rule linear-irreducible$_d$*)
next
case *False*
with *deg* **have** *degree p ≥ 2* **by** *auto*
with *kronecker-factorization-rat-main-complete*[*OF divisors-fun-int divisors-pos-fun-int*

kf[*unfolded d*] *this*]
show *?thesis*
by (*intro irreducible*$_d$*I2*, *auto*)
qed
qed

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*
by (*rule dvd-trans*, *rule gcd-greatest*[*OF dvd*])
(*auto intro*!: *mult-dvd-mono simp*: *gcd-mult-left*)

lemma *poly-gcd-monic-factor*:
monic p ⟹ *gcd* (*p* ∗ *q*) (*p* ∗ *r*) = *p* ∗ *gcd q r*
by (*rule gcdI* [*symmetric*]) (*simp-all add*: *normalize-mult normalize-monic dvd-gcd-mult*)

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*
using *field-poly-irreducible-imp-prime*[*OF irr*] **by** (*simp add*: *prime-elem-dvd-mult-iff*)

lemma *irreducible-dvd-pow*:
fixes *p* :: *'a poly*
assumes *irr*: *irreducible p*
shows *p dvd q* ^ *n* ⟹ *p dvd q*
using *field-poly-irreducible-imp-prime*[*OF irr*] **by** (*rule prime-elem-dvd-power*)

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*
by (*insert dvd*, *induct as rule*: *infinite-finite-induct*, *insert irr*, *auto*)

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*
by (*insert dvd*, *induct as*, *insert irr*, *auto*)

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 −
from *irreducible*$_d$*-factor*[*OF assms*(*2*)] **obtain** *s t*
where *irred*: *irreducible s* **and** *p*: *p* = *s* ∗ *t* **by** *auto*
from ‹*p dvd q* ∗ *r*› *p* **have** *s*: *s dvd q* ∗ *r* **unfolding** *dvd-def* **by** *auto*
from *s p irred* **show** *?thesis* **by** *auto*
qed

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 −
define *n* **where** *n* = *degree p*
have *degree p* = *n* **unfolding** *n-def* **by** *simp*
thus *?thesis*
proof (*induct n arbitrary*: *p*)
case (*0 p*)
hence ∃ *c*. *p* = [: *c* :] **by** (*cases p*, *auto split*: *if-splits*)
thus *?case* **by** (*intro exI*[*of - Nil*], *auto*)
next

```
    case (Suc n p)
    have dp: degree p = Suc n by fact
    hence ¬ constant (poly p) by (simp add: constant-degree)
    from fundamental-theorem-of-algebra[OF this] obtain c where rt: poly p c =
0 by auto
    hence [:−c,1 :] dvd p by (simp add: dvd-iff-poly-eq-0)
    then obtain q where p: p = q * [: −c,1 :] by (metis dvd-def mult.commute)
    from ‹degree p = Suc n› have dq: degree q = n using p
     by simp (metis add.right-neutral degree-synthetic-div diff-Suc-1 mult.commute
mult-left-cancel p pCons-eq-0-iff rt synthetic-div-correct′ zero-neq-one)
   from Suc(1)[OF this] obtain as where q: [:coeff q (degree q):] * (∏ a←as. [:−
a, 1:]) = q
      and deg: length as = degree q by auto
    have dc: degree p = degree q + degree [: −c, 1 :] unfolding dq dp by simp
     have cq: coeff q (degree q) = coeff p (degree p) unfolding dc unfolding p
coeff-mult-degree-sum unfolding dq by simp
    show ?case using p[unfolded q[unfolded cq, symmetric]]
     by (intro exI[of - c # as], auto simp: ac-simps, insert deg dc, auto)
  qed
qed

lemma rsquarefree-card-degree: assumes p0: (p :: complex poly) ≠ 0
  shows rsquarefree p = (card {x. poly p x = 0} = degree p)
proof −
  from fundamental-theorem-algebra-factorized[of p] obtain c as
    where p: p = smult c (∏ a ← as. [:− a, 1:]) and pas: degree p = length as
    and c: c = coeff p (degree p) by metis
  let ?prod = (∏ a←as. [:− a, 1:])
  from p0 have c: c ≠ 0 unfolding c by auto
 have roots: {x. poly p x = 0} = set as unfolding p poly-smult-zero-iff poly-prod-list
prod-list-zero-iff
    using c by auto
  have idr: (card {x. poly p x = 0} = degree p) = distinct as unfolding roots pas
    using card-distinct distinct-card by blast
  have id: ⋀ q. (p ≠ 0 ∧ q) = q using p0 by simp
  have dist: distinct as = (∀ a. (∑ x←as. if x = a then 1 else 0) ≤ Suc 0) (is ?l
= (∀ a. ?r a))
  proof (cases distinct as)
    case False
    from not-distinct-decomp[OF this] obtain xs ys zs a where as = xs @ [a] @
ys @ [a] @ zs by auto
    hence ¬ ?r a by auto
    thus ?thesis using False by auto
  next
    case True
    {
      fix a
      from True have ?r a
      proof (induct as)
```

```
      case (Cons b bs)
      show ?case
      proof (cases a = b)
        case False
        with Cons show ?thesis by auto
      next
        case True
        with Cons(2) have a ∉ set bs by auto
       hence (∑ x← bs. if x = a then 1 else 0) = (0 :: nat) by (induct bs, auto)
        thus ?thesis unfolding True by auto
      qed
    qed simp
   }
   thus ?thesis using True by auto
  qed
   have rsquarefree p = distinct as unfolding rsquarefree-def' id unfolding p
order-smult[OF c]
    by (subst order-prod-list, auto simp: o-def order-linear' dist)
  thus ?thesis unfolding idr by simp
qed


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 unfolding square-free-def
```

proof (*intro allI conjI*[*OF p*] *impI notI*, *goal-cases*)
case (*1 q*)
from *assms*(*1*)[*OF 1*(*1*) - *1*(*2*)] *1*(*1*) **show** *False* **by** (*cases q = 0*, *auto*)
qed

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 −
from *sf*[*unfolded square-free-def*] **have** *0*: *f* ≠ *0 g* ≠ *0*
and *dvd*: ⋀ *q*. *q* ∗ *q dvd f* ∗ *g* ⟹ *degree q = 0* **by** *auto*
then show *square-free f square-free g* **by** (*auto simp*: *square-free-def*)
assume *h dvd f h dvd g*
then have *h* ∗ *h dvd f* ∗ *g* **by** (*rule mult-dvd-mono*)
from *dvd*[*OF this*] **show** *degree h = 0*.
qed

lemma *irreducible*$_d$*-square-free*:
fixes *p* :: *'a* :: {*comm-semiring-1*, *semiring-no-zero-divisors*} *poly*
shows *irreducible*$_d$ *p* ⟹ *square-free p*
by (*metis degree-0 degree-mult-eq degree-mult-eq-0 irreducible*$_d$*D*(*1*) *irreducible*$_d$*D*(*2*)
irreducible$_d$*-dvd-smult irreducible*$_d$*-smultI less-add-same-cancel2 not-gr-zero square-free-def*)

lemma *square-free-factor*: **assumes** *dvd*: *a dvd p*
and *sf*: *square-free p*
shows *square-free a*
proof (*intro square-freeI*)
fix *q*
assume *q*: *degree q > 0* **and** *q* ∗ *q dvd a*
hence *q* ∗ *q dvd p* **using** *dvd dvd-trans sf square-free-def* **by** *blast*
with *sf*[*unfolded square-free-def*] *q* **show** *False* **by** *auto*
qed (*insert dvd sf*, *auto simp*: *square-free-def*)

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 (*rule ccontr*)
assume ¬ *distinct us*
from *not-distinct-decomp*[*OF this*] **obtain** *xs ys zs u* **where**
us = *xs* @ *u* # *ys* @ *u* # *zs* **by** *auto*
hence *dvd*: *u* ∗ *u dvd prod-list us* **and** *u*: *u* ∈ *set us* **by** *auto*
from *dvd us*[*OF u*] *sf* **have** *prod-list us = 0* **unfolding** *square-free-def* **by** *auto*
hence *0* ∈ *set us* **by** (*simp add*: *prod-list-zero-iff*)
from *us*[*OF this*] **show** *False* **by** *auto*
qed

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 (*rule ccontr*)
note *sep* = *sep*[*unfolded separable-def*]
from *sep* **have** *f0*: *f* ≠ *0* **by** (*cases f*, *auto*)
assume ¬ *square-free f*
then obtain *g* **where** *g*: *degree g* ≠ *0* **and** *g* ∗ *g dvd f* **using** *f0* **unfolding** *square-free-def* **by** *auto*
then obtain *h* **where** *f*: *f* = *g* ∗ (*g* ∗ *h*) **unfolding** *dvd-def* **by** (*auto simp*: *ac-simps*)
have *pderiv f* = *g* ∗ ((*g* ∗ *pderiv h* + *h* ∗ *pderiv g*) + *h* ∗ *pderiv g*)
unfolding *f pderiv-mult*[*of g*] **by** (*simp add*: *field-simps*)
hence *g dvd pderiv f* **unfolding** *dvd-def* **by** *blast*
moreover have *g dvd f* **unfolding** *f dvd-def* **by** *blast*
ultimately have *dvd*: *g dvd* (*gcd f* (*pderiv f*)) **by** *simp*
have *gcd f* (*pderiv f*) ≠ *0* **using** *f0* **by** *simp*
with *g dvd* **have** *degree* (*gcd f* (*pderiv f*)) ≠ *0*
by (*simp add*: *sep poly-dvd-1*)
hence ¬ *coprime f* (*pderiv f*) **by** *auto*
with *sep* **show** *False* **by** *simp*
qed

lemma *square-free-rsquarefree*: **assumes** *f*: *square-free f*
shows *rsquarefree f*
unfolding *rsquarefree-def*
proof (*intro conjI allI*)
fix *x*
show *order x f* = *0* ∨ *order x f* = *1*
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *n* **where** *ord*: *order x f* = *Suc* (*Suc n*)
by (*cases order x f*; *cases order x f* − *1*; *auto*)
define *p* **where** *p* = [:−*x*,*1*:]
from *order-divides*[*of x Suc* (*Suc 0*) *f*, *unfolded ord*]
have *p* ∗ *p dvd f degree p* ≠ *0* **unfolding** *p-def* **by** *auto*
hence ¬ *square-free f* **using** *f*(*1*) **unfolding** *square-free-def* **by** *auto*
with *assms* **show** *False* **by** *auto*
qed
qed (*insert f*, *auto simp*: *square-free-def*)

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*: $f \neq g$
shows *coprime f g*
proof −
have $(\prod\ fs) = f * (\prod\ (fs - \{f\}))$
by (*rule prod.remove*[*OF fin f*])
also have $(\prod\ (fs - \{f\})) = g * (\prod\ (fs - \{f\} - \{g\}))$
by (*rule prod.remove*, *insert fin g fg*, *auto*)
finally obtain *k* **where** *sf*: *square-free* $(f * g * k)$ **using** *sf* **by** (*simp add*: *ac-simps*)
from *sf*[*unfolded square-free-def*] **have** *0*: $f \neq 0\ g \neq 0$
and *dvd*: $\bigwedge q.\ q * q\ dvd\ f * g * k \Longrightarrow degree\ q = 0$
by *auto*
have *gcd f g* ∗ *gcd f g dvd f* ∗ *g* ∗ *k* **by** (*simp add*: *mult-dvd-mono*)
from *dvd*[*OF this*] **have** *degree* (*gcd f g*) = *0* **.**
moreover have *gcd f g* ≠ *0* **using** *0* **by** *auto*
ultimately show *coprime f g* **using** *is-unit-gcd*[*of f g*] *is-unit-iff-degree*[*of gcd f g*] **by** *simp*
qed

lemma *rsquarefree-square-free-complex*: **assumes** *rsquarefree* (*p* :: *complex poly*)
shows *square-free p*
proof (*rule square-freeI*)
fix *q*
assume *d*: *degree q* > *0* **and** *dvd*: *q* ∗ *q dvd p*
from *d* **have** ¬ *constant* (*poly q*) **by** (*simp add*: *constant-degree*)
from *fundamental-theorem-of-algebra*[*OF this*] **obtain** *x* **where** *poly q x* = *0* **by** *auto*
hence [:−*x*,*1*:] *dvd q* **by** (*simp add*: *poly-eq-0-iff-dvd*)
then obtain *k* **where** *q*: *q* = [:−*x*,*1*:] ∗ *k* **unfolding** *dvd-def* **by** *auto*
from *dvd* **obtain** *l* **where** *p*: *p* = *q* ∗ *q* ∗ *l* **unfolding** *dvd-def* **by** *auto*
from *p*[*unfolded q*] **have** *p* = [:−*x*,*1*:]^*2* ∗ (*k* ∗ *k* ∗ *l*) **by** *algebra*
hence [:−*x*,*1*:]^*2 dvd p* **unfolding** *dvd-def* **by** *blast*
from *this*[*unfolded order-divides*] **have** *p* = *0* ∨ ¬ *order x p* ≤ *1* **by** *auto*
thus *False* **using** *assms* **unfolding** *rsquarefree-def'* **by** *auto*
qed (*insert assms*, *auto simp*: *rsquarefree-def*)

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 −
note *cop* = *sep*[*unfolded separable-def*]
from *assms* **have** *f*: *f* ≠ *0* **unfolding** *square-free-def* **by** *auto*
let *?g* = *gcd f* (*pderiv f*)
define *G* **where** *G* = *?g*
from *poly-gcd-monic*[*of f pderiv f*] *f* **have** *mon*: *monic ?g*
by *auto*
have *deg*: *degree G* > *0*

proof (*cases degree G*)
case *0*
from *degree0-coeffs*[*OF this*] *cop mon* **show** *?thesis*
by (*auto simp*: *G-def coprime-iff-gcd-eq-1*)
qed *auto*
have *gf*: *G dvd f* **unfolding** *G-def* **by** *auto*
have *gf'*: *G dvd pderiv f* **unfolding** *G-def* **by** *auto*
from *irreducible*$_d$*-factor*[*OF deg*] **obtain** *g r* **where** *g*: *irreducible g* **and** *G*: *G* = *g* ∗ *r* **by** *auto*
from *gf* **have** *gf*: *g dvd f* **unfolding** *G* **by** (*rule dvd-mult-left*)
from *gf'* **have** *gf'*: *g dvd pderiv f* **unfolding** *G* **by** (*rule dvd-mult-left*)
have *g0*: *degree g* ≠ *0* **using** *g* **unfolding** *irreducible*$_d$*-def* **by** *auto*
from *gf* **obtain** *k* **where** *fgk*: *f* = *g* ∗ *k* **unfolding** *dvd-def* **by** *auto*
have *id1*: *pderiv f* = *g* ∗ *pderiv k* + *k* ∗ *pderiv g* **unfolding** *fgk pderiv-mult* **by** *simp*
from *gf'* **obtain** *h* **where** *pderiv f* = *g* ∗ *h* **unfolding** *dvd-def* **by** *auto*
from *id1*[*unfolded this*] **have** *k* ∗ *pderiv g* = *g* ∗ (*h* − *pderiv k*) **by** (*simp add*: *field-simps*)
hence *dvd*: *g dvd k* ∗ *pderiv g* **unfolding** *dvd-def* **by** *auto*
{
assume *g dvd k*
then obtain *h* **where** *k*: *k* = *g* ∗ *h* **unfolding** *dvd-def* **by** *auto*
with *fgk* **have** *g* ∗ *g dvd f* **by** *auto*
with *g0* **have** ¬ *square-free f* **unfolding** *square-free-def* **using** *f* **by** *auto*
with *assms* **have** *False* **by** *simp*
}
with *g dvd*
have *g dvd pderiv g* **by** *auto*
from *divides-degree*[*OF this*] *degree-pderiv-le*[*of g*] *g0*
have *pderiv g* = *0* **by** *linarith*
with *fgk g0* **show** *?thesis* **by** *auto*
qed

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 (*rule ccontr*)
assume ¬ *separable f*
from *square-free-separable-main*[*OF assms this*]
obtain *g k* **where** ∗: *f* = *g* ∗ *k degree g* ≠ *0 pderiv g* = *0* **by** *auto*
hence *g dvd pderiv g* **by** *auto*
thus *False* **unfolding** *dvd-pderiv-iff* **using** ∗ **by** *auto*
qed

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

using *separable-imp-square-free*[*of f*] *square-free-imp-separable*[*of f*] **by** *auto*

context
assumes *SORT-CONSTRAINT*('*a*::{*field*,*factorial-ring-gcd*})
begin
lemma *square-free-smult*: $c \neq 0 \Longrightarrow$ *square-free* (f :: '*a poly*) $\Longrightarrow$ *square-free* (*smult c f*)
by (*unfold square-free-def*, *insert dvd-smult-cancel*[*of* - *c*], *auto*)

lemma *square-free-smult-iff*[*simp*]: $c \neq 0 \Longrightarrow$ *square-free* (*smult c f*) = *square-free* (f :: '*a poly*)
using *square-free-smult*[*of c f*] *square-free-smult*[*of inverse c smult c f*] **by** *auto*
end

context
assumes *SORT-CONSTRAINT*('*a*::*factorial-ring-gcd*)
begin
definition *square-free-factorization* :: '*a poly* $\Rightarrow$ '*a* $\times$ ('*a poly* $\times$ *nat*) *list* $\Rightarrow$ *bool*
where
square-free-factorization p cbs $\equiv$ *case cbs of* (*c*,*bs*) $\Rightarrow$
(p = *smult c* ($\prod$(*a*, *i*)$\in$ *set bs*. *a* ^ *i*))
$\wedge$ (p = *0* $\longrightarrow$ *c* = *0* $\wedge$ *bs* = [])
$\wedge$ ($\forall$ *a i*. (*a*,*i*) $\in$ *set bs* $\longrightarrow$ *square-free a* $\wedge$ *degree a* > *0* $\wedge$ *i* > *0*)
$\wedge$ ($\forall$ *a i b j*. (*a*,*i*) $\in$ *set bs* $\longrightarrow$ (*b*,*j*) $\in$ *set bs* $\longrightarrow$ (*a*,*i*) $\neq$ (*b*,*j*) $\longrightarrow$ *coprime a b*)
$\wedge$ *distinct bs*

lemma *square-free-factorizationD*: **assumes** *square-free-factorization p* (*c*,*bs*)
shows p = *smult c* ($\prod$(*a*, *i*)$\in$ *set bs*. *a* ^ *i*)
(*a*,*i*) $\in$ *set bs* $\Longrightarrow$ *square-free a* $\wedge$ *degree a* $\neq$ *0* $\wedge$ *i* > *0*
(*a*,*i*) $\in$ *set bs* $\Longrightarrow$ (*b*,*j*) $\in$ *set bs* $\Longrightarrow$ (*a*,*i*) $\neq$ (*b*,*j*) $\Longrightarrow$ *coprime a b*
p = *0* $\Longrightarrow$ *c* = *0* $\wedge$ *bs* = []
distinct bs
using *assms* **unfolding** *square-free-factorization-def split* **by** *blast*+

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 −
note *sff* = *square-free-factorizationD*[*OF assms*]
show *?thesis* **unfolding** *sff*(*1*)
by (*simp add*: *prod.distinct-set-conv-list*[*OF sff*(*5*)])
qed
end

11.1 Yun's factorization algorithm

locale *yun-gcd* =
fixes *Gcd* :: '*a* :: *factorial-ring-gcd poly* $\Rightarrow$ '*a poly* $\Rightarrow$ '*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 .

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,[])
  unfolding yun-factorization-def by simp
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* (λ (*a*,*i*). *a* ^ *i*) *as*) ∗ *prod* (λ (*a*,*i*). *a*) *as*
  unfolding *p prod.distrib*[*symmetric*]
  by (*rule prod.cong*, *auto*)

lemma *pderiv-exp-prod*:
  pderiv p = (*prod* (λ (*a*,*i*). *a* ^ *i*) *as* ∗ *sum* (λ (*a*,*i*).
    prod (λ (*b*,*j*). *b*) (*as* − {(*a*,*i*)}) ∗ *smult* (*of-nat* (*Suc i*)) (*pderiv a*)) *as*)
  unfolding *p pderiv-prod sum-distrib-left*
proof (*rule sum.cong*[*OF refl*])
  fix *x*
  assume *x* ∈ *as*
  then obtain *a i* **where** *x*: *x* = (*a*,*i*) **and** *mem*: (*a*,*i*) ∈ *as* **by** (*cases x*, *auto*)
  let *?si* = *smult* (*of-nat* (*Suc i*)) :: '*a poly* ⇒ '*a poly*
  show (∏ (*a*, *i*)∈*as* − {*x*}. *a* ^ *Suc i*) ∗ *pderiv* (*case x of* (*a*, *i*) ⇒ *a* ^ *Suc i*) =
      (∏ (*a*, *i*)∈*as*. *a* ^ *i*) ∗
       (*case x of* (*a*, *i*) ⇒ (∏ (*a*, *i*)∈*as* − {(*a*, *i*)}. *a*) ∗ *smult* (*of-nat* (*Suc i*)) (*pderiv a*))
    unfolding *x split pderiv-power-Suc*
  proof −
    let *?prod* = ∏ (*a*, *i*)∈*as* − {(*a*, *i*)}. *a* ^ *Suc i*
    let *?l* = *?prod* ∗ (*?si* (*a* ^ *i*) ∗ *pderiv a*)
    let *?r* = (∏ (*a*, *i*)∈*as*. *a* ^ *i*) ∗ ((∏ (*a*, *i*)∈*as* − {(*a*, *i*)}. *a*) ∗ *?si* (*pderiv a*))
    have *?r* = *a* ^ *i* ∗ ((∏ (*a*, *i*)∈*as* − {(*a*, *i*)}. *a* ^ *i*) ∗ (∏ (*a*, *i*)∈*as* − {(*a*, *i*)}. *a*) ∗ *?si* (*pderiv a*))
      unfolding *prod.remove*[*OF fin mem*] **by** (*simp add*: *ac-simps*)
    also have (∏ (*a*, *i*)∈*as* − {(*a*, *i*)}. *a* ^ *i*) ∗ (∏ (*a*, *i*)∈*as* − {(*a*, *i*)}. *a*)
      = *?prod* **unfolding** *prod.distrib*[*symmetric*]
      by (*rule prod.cong*[*OF refl*], *auto*)
    finally show *?l* = *?r*
      by (*simp add*: *ac-simps*)
  qed
qed

lemma *monic-gen*: **assumes** *bs* ⊆ *as*
  shows *monic* (∏ (*a*, *i*) ∈ *bs*. *a*)
  by (*rule monic-prod*, *insert assms as-monic*, *auto*)

lemma *nonzero-gen*: **assumes** $bs \subseteq as$
shows $(\prod\ (a,\ i) \in bs.\ a) \neq 0$
using *monic-gen*[*OF assms*] **by** *auto*

lemma *monic-Prod*: *monic* $((\prod (a,\ i) \in as.\ a\ \hat{}\ i))$
by (*rule monic-prod*, *insert as-monic*, *auto intro*: *monic-power*)

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 −
have *single*: *?single* $\neq 0$ **by** (*rule nonzero-gen*[*OF bs*])
show *?thesis*
proof (*rule gcd-eq-1-imp-coprime*, *rule gcdI* [*symmetric*])
fix *k*
assume *dvd*: *k dvd ?single k dvd ?onederiv*
note *bs-monic* = *as-monic*[*OF subsetD*[*OF bs*]]
from *dvd*(*1*) *single* **have** *k*: $k \neq 0$ **by** *auto*
show *k dvd 1*
proof (*cases degree k* > 0)
case *False*
with *k* **obtain** *c* **where** $k = [:c:]$
by (*auto dest*: *degree0-coeffs*)
with *k* **have** $c \neq 0$
by *auto*
with ‹$k = [:c:]$› **show** *is-unit k*
using *dvdI* [*of 1* [:*c*:] [:*1 / c*:]] **by** *auto*
next
case *True*
from *irreducible*$_d$-*factor*[*OF this*]
obtain *p q* **where** *k*: $k = p * q$ **and** *p*: *irreducible p* **by** *auto*
from *k dvd* **have** *dvd*: *p dvd ?single p dvd ?onederiv* **unfolding** *dvd-def* **by** *auto*
from *irreducible-dvd-prod*[*OF p dvd*(*1*)] **obtain** *a i* **where** *ai*: $(a,i) \in bs$ **and** *pa*: *p dvd a*
by *force*
then obtain *q* **where** *a*: $a = p * q$ **unfolding** *dvd-def* **by** *auto*
from *p*[*unfolded irreducible*$_d$-*def*] **have** *p0*: *degree* $p > 0$ **by** *auto*
from *irreducible*$_d$-*dvd-smult*[*OF p0 as-irred pa*] *ai bs*
obtain *c* **where** *c*: $c \neq 0$ **and** *ap*: $a = smult\ c\ p$ **by** *auto*
hence *ap'*: $p = smult\ (1/c)\ a$ **by** *auto*
let *?prod* $= \lambda\ a\ i.\ (\prod (b,\ j) \in bs - \{(a,\ i)\}.\ b) * smult\ (of\text{-}nat\ (f\ i))\ (pderiv\ a)$
let *?prod'* $= \lambda\ aa\ ii\ a\ i.\ (\prod (b,\ j) \in bs - \{(a,\ i),(aa,ii)\}.\ b) * smult\ (of\text{-}nat\ (f\ i))\ (pderiv\ a)$
define *factor* **where** *factor* $= sum\ (\lambda\ (b,j).\ ?prod'\ a\ i\ b\ j\ )\ (bs - \{(a,i)\})$

define *fac* **where** *fac* = *q* ∗ *factor*
from *fin finite-subset*[*OF bs*] **have** *fin*: *finite bs* **by** *auto*
have *?onederiv* = *?prod a i* + *sum* (λ (*b*,*j*). *?prod b j*) (*bs* − {(*a*,*i*)})
by (*subst sum.remove*[*OF fin ai*], *auto*)
also have *sum* (λ (*b*,*j*). *?prod b j*) (*bs* − {(*a*,*i*)})
= *a* ∗ *factor*
unfolding *factor-def sum-distrib-left*
proof (*rule sum.cong*[*OF refl*])
fix *bj*
assume *mem*: *bj* ∈ *bs* − {(*a*,*i*)}
obtain *b j* **where** *bj*: *bj* = (*b*,*j*) **by** *force*
from *mem bj ai* **have** *ai*: (*a*,*i*) ∈ *bs* − {(*b*,*j*)} **by** *auto*
have *id*: *bs* − {(*b*, *j*)} − {(*a*, *i*)} = *bs* − {(*b*,*j*),(*a*,*i*)} **by** *auto*
show (λ (*b*,*j*). *?prod b j*) *bj* = *a* ∗ (λ (*b*,*j*). *?prod′ a i b j*) *bj*
unfolding *bj split*
by (*subst prod.remove*[*OF - ai*], *insert fin*, *auto simp*: *id ac-simps*)
qed
finally have *?onederiv* = *?prod a i* + *p* ∗ *fac* **unfolding** *fac-def a* **by** *simp*
from *dvd*(*2*)[*unfolded this*] **have** *p dvd ?prod a i* **by** *algebra*
from *this*[*unfolded field-poly-irreducible-dvd-mult*[*OF p*]]
have *False*
proof
assume *p dvd* (∏ (*b*, *j*)∈*bs* − {(*a*, *i*)}. *b*)
from *irreducible-dvd-prod*[*OF p this*] **obtain** *b j* **where** *bj′*: (*b*,*j*) ∈ *bs* − {(*a*,*i*)}
and *pb*: *p dvd b* **by** *auto*
hence *bj*: (*b*,*j*) ∈ *bs* **by** *auto*
from *as-irred bj bs* **have** *irreducible*$_d$ *b* **by** *auto*
from *irreducible*$_d$*-dvd-smult*[*OF p0 this pb*] **obtain** *d* **where** *d*: *d* ≠ *0*
and *b*: *b* = *smult d p* **by** *auto*
with *ap c* **have** *id*: *smult* (*c*/*d*) *b* = *a* **and** *deg*: *degree a* = *degree b* **by** *auto*
from *coeff-smult*[*of c*/*d b degree b*, *unfolded id*] *deg bs-monic*[*OF ai*] *bs-monic*[*OF bj*]
have *c* / *d* = *1* **by** *simp*
from *id*[*unfolded this*] **have** *a* = *b* **by** *simp*
with *as-distinct*[*OF subsetD*[*OF bs ai*] *subsetD*[*OF bs bj*]] *bj′*
show *False* **by** *auto*
next
from *f*[*OF ai*] **obtain** *k* **where** *fi*: *f i* = *Suc k* **by** (*cases f i*, *auto*)
assume *p dvd smult* (*of-nat* (*f i*)) (*pderiv a*)
hence *p dvd* (*pderiv a*) **unfolding** *fi* **using** *dvd-smult-cancel of-nat-eq-0-iff* **by** *blast*
from *this*[*unfolded ap*] **have** *p dvd pderiv p* **using** *c*
by (*metis* ‹*p dvd pderiv a*› *ap′ dvd-trans dvd-triv-right mult.left-neutral pderiv-smult smult-dvd-cancel*)
with *not-dvd-pderiv p0* **show** *False* **by** *auto*
qed
thus *k dvd 1* **by** *simp*
qed

qed (*insert* ‹*?single* ≠ *0*›, *auto*)
qed

lemma *pderiv-exp-gcd*:
gcd p (*pderiv p*) = (∏ (*a*, *i*)∈*as*. *a* ^ *i*) (**is** - = *?prod*)
proof −
let *?sum* = (∑ (*a*, *i*)∈*as*. (∏ (*b*, *j*)∈*as* − {(*a*, *i*)}. *b*) ∗ *smult* (*of-nat* (*Suc i*)) (*pderiv a*))
let *?single* = (∏ (*a*, *i*)∈*as*. *a*)
let *?prd* = λ *a i*. (∏ (*b*, *j*)∈*as* − {(*a*, *i*)}. *b*) ∗ *smult* (*of-nat* (*Suc i*)) (*pderiv a*)
let *?onederiv* = ∑ (*a*, *i*)∈*as*. *?prd a i*
have *pp*: *pderiv p* = *?prod* ∗ *?sum* **by** (*rule pderiv-exp-prod*)
have *p*: *p* = *?prod* ∗ *?single* **by** (*rule poly-exp-expand*)
have *monic*: *monic ?prod* **by** (*rule monic-Prod*)
have *gcd*: *coprime ?single ?onederiv*
by (*rule coprime-generic*, *auto*)
then have *gcd*: *gcd ?single ?onederiv* = *1*
by *simp*
show *?thesis* **unfolding** *pp* **unfolding** *p poly-gcd-monic-factor* [*OF monic*] *gcd*
by *simp*
qed

lemma *p-div-gcd-p-pderiv*: *p div* (*gcd p* (*pderiv p*)) = (∏ (*a*, *i*)∈*as*. *a*)
unfolding *pderiv-exp-gcd* **unfolding** *poly-exp-expand*
by (*rule nonzero-mult-div-cancel-left*, *insert monic-Prod*, *auto*)

fun *A B C D* :: *nat* ⇒ *'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* = (∏ (*a*, *i*) ∈ *as* ∩ *UNIV* × {*n*}. *a*)
B n = (∏ (*a*, *i*) ∈ *as* − *UNIV* × {*0* ..< *n*}. *a*)
C n = (∑ (*a*, *i*)∈*as* − *UNIV* × {*0* ..< *n*}.
(∏ (*b*, *j*)∈*as* − *UNIV* × {*0* ..< *n*} − {(*a*, *i*)}. *b*) ∗ *smult* (*of-nat* (*Suc i* − *n*)) (*pderiv a*))
D n = (∏ (*a*, *i*) ∈ *as* ∩ *UNIV* × {*n*}. *a*) ∗
(∑ (*a*,*i*)∈*as* − *UNIV* × {*0* ..< *Suc n*}.
(∏ (*b*, *j*)∈ *as* − *UNIV* × {*0* ..< *Suc n*} − {(*a*, *i*)}. *b*) ∗ (*smult* (*of-nat* (*i* − *n*)) (*pderiv a*)))
proof (*induct n* **and** *n* **and** *n* **and** *n rule*: *A-B-C-D.induct*)
case (*1 n*)
note *Bn* = *1*(*1*)
note *Dn* = *1*(*2*)
have (∏ (*a*, *i*)∈*as* − *UNIV* × {*0*..< *n*}. *a*) = (∏ (*a*, *i*)∈*as* ∩ *UNIV* × {*n*}. *a*) ∗ (∏ (*a*, *i*)∈*as* − *UNIV* × {*0*..<*Suc n*}. *a*)

```
  by (subst prod.union-disjoint[symmetric], auto, insert fin, auto intro: prod.cong)
 note Bn' = Bn[unfolded this]
 let ?an = (∏ (a, i) ∈ as ∩ UNIV × {n}. a)
 let ?bn = (∏(a, i)∈as − UNIV × {0..<Suc n}. a)
 show A n = ?an unfolding A.simps
 proof (rule gcdI[symmetric, OF - - - normalize-monic[OF monic-gen]])
  have monB1: monic (B n) unfolding Bn by (rule monic-gen, auto)
  hence B n ≠ 0 by auto
  let ?dn = (∑ (a,i)∈as − UNIV × {0 ..< Suc n}.
      (∏(b, j)∈ as − UNIV × {0 ..< Suc n} − {(a, i)}. b) * (smult (of-nat (i
− n)) (pderiv a)))
  have Dn: D n = ?an * ?dn unfolding Dn by auto
  show dvd1: ?an dvd B n unfolding Bn' dvd-def by blast
  show dvd2: ?an dvd D n unfolding Dn dvd-def by blast
  {
   fix k
   assume k dvd B n k dvd D n
   from dvd-gcd-mult[OF this[unfolded Bn' Dn]]
   have k dvd ?an * (gcd ?bn ?dn) .
   moreover have coprime ?bn ?dn
    by (rule coprime-generic, auto)
   ultimately show k dvd ?an by simp
  }
 qed auto
next
 case 2
 have as: as − UNIV × {0..<0} = as by auto
 show ?case unfolding B.simps as p-div-gcd-p-pderiv by auto
next
 case (3 n)
  have id: (∏(a, i)∈as − UNIV × {0..< n}. a) = (∏(a, i)∈as − UNIV ×
{0..<Suc n}. a) * (∏(a, i)∈as ∩ UNIV × {n}. a)
   by (subst prod.union-disjoint[symmetric], auto, insert fin, auto intro: prod.cong)
 show ?case unfolding B.simps 3 id
   by (subst nonzero-mult-div-cancel-right[OF nonzero-gen], auto)
next
 case 4
 have as: as − UNIV × {0..<0} = as ⋀ i. Suc i − 0 = Suc i by auto
 show ?case unfolding C.simps pderiv-exp-gcd unfolding pderiv-exp-prod as
   by (rule nonzero-mult-div-cancel-left, insert monic-Prod, auto)
next
 case (5 n)
 show ?case unfolding C.simps 5
   by (subst nonzero-mult-div-cancel-left, rule nonzero-gen, auto)
next
 case (6 n)
  let ?f = λ (a,i). (∏(b, j)∈as − UNIV × {0 ..< n} − {(a, i)}. b) * (smult
(of-nat (i − n)) (pderiv a))
 have D n = (∑ (a,i)∈as − UNIV × {0 ..< n}. (∏(b, j)∈as − UNIV × {0
```

```
..< n} − {(a, i)}. b) ∗
    (smult (of-nat (Suc i − n)) (pderiv a) − pderiv a))
    unfolding D.simps 6 pderiv-prod sum-subtractf[symmetric] right-diff-distrib
    by (rule sum.cong, auto)
  also have ... = sum ?f (as − UNIV × {0 ..< n})
  proof (rule sum.cong[OF refl])
    fix x
    assume x ∈ as − UNIV × {0 ..< n}
    then obtain a i where x: x = (a,i) and i: Suc i > n by (cases x, auto)
    hence id: Suc i − n = Suc (i − n) by arith
    have id: of-nat (Suc i − n) = of-nat (i − n) + (1 :: 'a) unfolding id by simp
    have id: smult (of-nat (Suc i − n)) (pderiv a) − pderiv a = smult (of-nat (i
− n)) (pderiv a)
      unfolding id smult-add-left by auto
    have cong: ⋀ x y z :: 'a poly. x = y ⟹ x ∗ z = y ∗ z by auto
    show (case x of
          (a, i) ⇒
            (∏(b, j)∈as − UNIV × {0..<n} − {(a, i)}. b) ∗
            (smult (of-nat (Suc i − n)) (pderiv a) − pderiv a)) =
         (case x of
          (a, i) ⇒ (∏(b, j)∈as − UNIV × {0..<n} − {(a, i)}. b) ∗ smult (of-nat
(i − n)) (pderiv a))
      unfolding x split id
      by (rule cong, auto)
  qed
  also have ... = sum ?f (as − UNIV × {0 ..< Suc n}) + sum ?f (as ∩ UNIV
× {n})
    by (subst sum.union-disjoint[symmetric], insert fin, auto intro: sum.cong)
  also have sum ?f (as ∩ UNIV × {n}) = 0
    by (rule sum.neutral, auto)
  finally have id: D n = sum ?f (as − UNIV × {0 ..< Suc n}) by simp
  show ?case unfolding id sum-distrib-left
  proof (rule sum.cong[OF refl])
    fix x
    assume mem: x ∈ as − UNIV × {0 ..< Suc n}
    obtain a i where x: x = (a,i) by force
    with mem have i: i > n by auto
    have cong: ⋀ x y z v :: 'a poly. x = y ∗ v ⟹ x ∗ z = y ∗ (v ∗ z) by auto
    show (case x of
          (a, i) ⇒ (∏(b, j)∈as − UNIV × {0..<n} − {(a, i)}. b) ∗ smult (of-nat
(i − n)) (pderiv a)) =
         (∏(a, i)∈as ∩ UNIV × {n}. a) ∗
         (case x of (a, i) ⇒
            (∏(b, j)∈as − UNIV × {0..<Suc n} − {(a, i)}. b) ∗ smult (of-nat (i
− n)) (pderiv a))
      unfolding x split
      by (rule cong, subst prod.union-disjoint[symmetric], insert fin, (auto)[3],
        rule prod.cong, insert i, auto)
  qed
```

qed

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*:
(∏ *i* = *0*..< *n*. *A i* ^ *Suc i*) = (∏ (*a*, *i*)∈ *as* ∩ *UNIV* × {*0* ..< *n*}. *a* ^ *Suc i*)
proof (*induct n*)
case (*Suc n*)
have *id*: {*0* ..< *Suc n*} = *insert n* {*0* ..< *n*} **by** *auto*
have *id2*: *as* ∩ *UNIV* × {*0* ..< *Suc n*} = *as* ∩ *UNIV* × {*n*} ∪ *as* ∩ *UNIV* × {*0* ..< *n*} **by** *auto*
have *cong*: ⋀ *x y z*. *x* = *y* ⟹ *x* ∗ *z* = *y* ∗ *z* **by** *auto*
show *?case* **unfolding** *id2* **unfolding** *id*
proof (*subst prod.insert*; (*subst prod.union-disjoint*)*?*; (*unfold Suc*)*?*;
(*unfold A*, *rule cong*)*?*)
show (∏ (*a*, *i*)∈*as* ∩ *UNIV* × {*n*}. *a*) ^ *Suc n* = (∏ (*a*, *i*)∈*as* ∩ *UNIV* × {*n*}. *a* ^ *Suc i*)
unfolding *prod-power-distrib*
by (*rule prod.cong*, *auto*)
qed (*insert fin*, *auto*)
qed *simp*

lemma *prod-A-is-p-unknown*: **assumes** ⋀ *a i*. (*a*,*i*) ∈ *as* ⟹ *i* < *n*
shows *p* = (∏ *i* = *0*..< *n*. *A i* ^ *Suc i*)
proof −
have *p* = (∏ (*a*, *i*)∈*as*. *a* ^ *Suc i*) **by** (*rule p*)
also have ... = (∏ *i* = *0*..< *n*. *A i* ^ *Suc i*) **unfolding** *prod-A*
by (*rule prod.cong*, *insert assms*, *auto*)
finally show *?thesis* **.**
qed

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

lemma *bound*: **assumes** *m*: *m* ≥ *bound*
shows *B m* = *1*
proof −
let *?set* = *as* − *UNIV* × {*0*..<*m*}
{
fix *a i*
assume *ai*: (*a*,*i*) ∈ *?set*
hence *i* ∈ *snd* ‘ *as* **by** *force*
from *Max-ge*[*OF - this*] *fin* **have** *i* ≤ *Max* (*snd* ‘ *as*) **by** *auto*
with *ai m*[*unfolded bound-def*] **have** *False* **by** *auto*
}

```
  hence id: ?set = {} by force
  show B m = 1 unfolding B id by simp
qed

lemma coprime-A-A: assumes i ≠ j
  shows coprime (A i) (A j)
proof (rule coprimeI)
  fix k
  assume dvd: k dvd A i k dvd A j
  have Ai: A i ≠ 0 unfolding A
    by (rule nonzero-gen, auto)
  with dvd have k: k ≠ 0 by auto
  show is-unit k
  proof (cases degree k > 0)
    case False
    then obtain c where kc: k = [: c :] by (auto dest: degree0-coeffs)
    with k have 1 = k * [:1 / c:]
      by simp
    thus ?thesis unfolding dvd-def by blast
  next
    case True
    from irreducible-monic-factor[OF this]
    obtain q r where k: k = q * r and q: irreducible q and mq: monic q by auto
    with dvd have dvd: q dvd A i q dvd A j unfolding dvd-def by auto
    from q have q0: degree q > 0 unfolding irreducible_d-def by auto
    from irreducible-dvd-prod[OF q dvd(1)[unfolded A]]
      obtain a where ai: (a,i) ∈ as and qa: q dvd a by auto
    from irreducible-dvd-prod[OF q dvd(2)[unfolded A]]
      obtain b where bj: (b,j) ∈ as and qb: q dvd b by auto
    from as-distinct[OF ai bj] assms have neq: a ≠ b by auto
    from irreducible_d-dvd-smult[OF q0 as-irred[OF ai] qa]
      irreducible_d-dvd-smult[OF q0 as-irred[OF bj] qb]
    obtain c d where c ≠ 0 d ≠ 0 a = smult c q b = smult d q by auto
    hence ab: a = smult (c / d) b and c / d ≠ 0 by auto
    with as-monic[OF bj] as-monic[OF ai] arg-cong[OF ab, of λ p. coeff p (degree
p)]
    have a = b unfolding coeff-smult degree-smult-eq by auto
    with neq show ?thesis by auto
  qed
qed

lemma A-monic: monic (A i)
  unfolding A by (rule monic-gen, auto)

lemma A-square-free: square-free (A i)
proof (rule square-freeI)
  fix q k
  have mon: monic (A i) by (rule A-monic)
  hence Ai: A i ≠ 0 by auto
```

assume *q*: *degree q > 0* **and** *dvd*: *q * q dvd A i*
from *irreducible-monic-factor*[*OF q*] **obtain** *r s* **where** *q*: *q = r * s* **and**
irr: *irreducible r* **and** *mr*: *monic r* **by** *auto*
from *dvd*[*unfolded q*] **have** *dvd2*: *r * r dvd A i* **and** *dvd1*: *r dvd A i* **unfolding** *dvd-def* **by** *auto*
from *irreducible-dvd-prod*[*OF irr dvd1*[*unfolded A*]]
obtain *a* **where** *ai*: (*a*,*i*) ∈ *as* **and** *ra*: *r dvd a* **by** *auto*
let *?rem* = (∏ (*a*, *i*)∈*as* ∩ *UNIV* × {*i*} − {(*a*,*i*)}. *a*)
have *a*: *irreducible*$_d$ *a* **by** (*rule as-irred*[*OF ai*])
from *irreducible*$_d$*-dvd-smult*[*OF - a ra*] *irr*
obtain *c* **where** *ar*: *a = smult c r* **and** *c ≠ 0* **by** *force*
with *mr as-monic*[*OF ai*] *arg-cong*[*OF ar*, *of* λ *p*. *coeff p* (*degree p*)]
have *a = r* **unfolding** *coeff-smult degree-smult-eq* **by** *auto*
with *dvd2* **have** *dvd*: *a * a dvd A i* **by** *simp*
have *id*: *A i = a * ?rem* **unfolding** *A*
by (*subst prod.remove*[*of - (a,i)*], *insert ai fin*, *auto*)
with *dvd* **have** *a dvd ?rem* **using** *a id Ai* **by** *auto*
from *irreducible-dvd-prod*[*OF - this*] *a* **obtain** *b* **where** *bi*: (*b*,*i*) ∈ *as*
and *neq*: *b ≠ a* **and** *ab*: *a dvd b* **by** *auto*
from *as-irred*[*OF bi*] **have** *b*: *irreducible*$_d$ *b* **.**
from *irreducible*$_d$*-dvd-smult*[*OF - b ab*] *a*[*unfolded irreducible*$_d$*-def*]
obtain *c* **where** *c ≠ 0* **and** *ba*: *b = smult c a* **by** *auto*
with *as-monic*[*OF bi*] *as-monic*[*OF ai*] *arg-cong*[*OF ba*, *of* λ *p*. *coeff p* (*degree p*)]
have *a = b* **unfolding** *coeff-smult degree-smult-eq* **by** *auto*
with *neq* **show** *False* **by** *auto*
qed (*insert A-monic*[*of i*], *auto*)

lemma *prod-A-is-p-B-bound*: **assumes** *B n = 1*
shows *p* = (∏ *i = 0..< n*. *A i* ^ *Suc i*)
proof (*rule prod-A-is-p-unknown*)
fix *a i*
assume *ai*: (*a*,*i*) ∈ *as*
let *?rem* = (∏ (*a*, *i*)∈*as − UNIV* × {*0..<n*} − {(*a*,*i*)}. *a*)
have *rem*: *?rem ≠ 0*
by (*rule nonzero-gen*, *auto*)
have *irreducible*$_d$ *a* **using** *as-irred*[*OF ai*] **.**
hence *a*: *a ≠ 0 degree a ≠ 0* **unfolding** *irreducible*$_d$*-def* **by** *auto*
show *i < n*
proof (*rule ccontr*)
assume ¬ *?thesis*
hence *i ≥ n* **by** *auto*
with *ai* **have** *mem*: (*a*,*i*) ∈ *as − UNIV* × {*0 ..< n*} **by** *auto*
have *0 = degree* (∏ (*a*, *i*)∈*as − UNIV* × {*0..<n*}. *a*) **using** *assms* **unfolding** *B* **by** *simp*
also have ... = *degree* (*a * ?rem*)
by (*subst prod.remove*[*OF - mem*], *insert fin*, *auto*)
also have ... = *degree a* + *degree ?rem*

```
    by (rule degree-mult-eq[OF a(1) rem])
  finally show False using a(2) by auto
 qed
qed

interpretation yun-gcd gcd .

lemma square-free-monic-poly: (poly (square-free-monic-poly p) x = 0) = (poly p
x = 0)
proof −
 show ?thesis unfolding square-free-monic-poly-def unfolding p-div-gcd-p-pderiv
   unfolding p poly-prod prod-zero-iff[OF fin] by force
qed

lemma yun-factorization-induct: assumes base: ⋀ bn cn. bn = 1 ⟹ P bn cn
 and step: ⋀ 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 −
 define n where n = (0 :: nat)
 let ?m = λ n. bound − n
 have P (B n) (C n)
 proof (induct n rule: wf-induct[OF wf-measure[of ?m]])
   case (1 n)
   note IH = 1(1)[rule-format]
   show ?case
   proof (cases B n = 1)
     case True
     with base show ?thesis by auto
   next
     case False note Bn = this
     with bound[of n] have ¬ bound ≤ n by auto
     hence (Suc n, n) ∈ measure ?m by auto
     note IH = IH[OF this]
     show ?thesis
         by (rule step[OF Bn], insert IH, simp add: D.simps C.simps B.simps
A.simps)
   qed
 qed
 thus ?thesis unfolding id n-def B.simps C.simps .
qed

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)
 using assms
```

proof −
let *?m* = λ *n*. *bound* − *n*
show *?thesis* **using** *assms*
proof (*induct n arbitrary*: *bs rule*: *wf-induct*[*OF wf-measure*[*of ?m*]])
case (*1 n*)
note *IH* = *1*(*1*)[*rule-format*]
have *res*: *yun-factorization-main* (*B n*) (*C n*) *n bs* = *cs* **by** *fact*
note *res* = *res*[*unfolded yun-factorization-main.simps*[*of B n*]]
have *bs*: *set bs* = {(*A i*, *Suc i*) |*i*. *i* < *n*} *distinct* (*map snd bs*) **by** *fact*+
show *?case*
proof (*cases B n* = *1*)
case *True*
with *res* **have** *bs* = *cs* **by** *auto*
with *True bs* **show** *?thesis* **by** *auto*
next
case *False* **note** *Bn* = *this*
with *bound*[*of n*] **have** ¬ *bound* ≤ *n* **by** *auto*
hence (*Suc n*, *n*) ∈ *measure ?m* **by** *auto*
note *IH* = *IH*[*OF this*]
from *Bn res*[*unfolded Let-def*, *folded D.simps C.simps B.simps A.simps*]
have *res*: *yun-factorization-main* (*B* (*Suc n*)) (*C* (*Suc n*)) (*Suc n*) ((*A n*, *Suc n*) # *bs*) = *cs*
by *simp*
note *IH* = *IH*[*OF this*]
{
fix *i*
assume *i* < *Suc n* ¬ *i* < *n*
hence *n* = *i* **by** *arith*
} **note** *missing* = *this*
have *set* ((*A n*, *Suc n*) # *bs*) = {(*A i*, *Suc i*) |*i*. *i* < *Suc n*}
unfolding *list.simps bs* **by** (*auto*, *subst missing*, *auto*)
note *IH* = *IH*[*OF this*]
from *bs* **have** *distinct* (*map snd* ((*A n*, *Suc n*) # *bs*)) **by** *auto*
note *IH* = *IH*[*OF this*]
show *?thesis* **by** (*rule IH*)
qed
qed
qed

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 −
from *res*[*unfolded yun-monic-factorization-def Let-def*,
folded B.simps C.simps]
obtain *cs* **where** *yun*: *yun-factorization-main* (*B 0*) (*C 0*) *0* [] = *cs* **and** *bs*: *bs* = *filter* (λ (*a*,*i*). *a* ≠ *1*) *cs*
by *auto*
from *yun-factorization-main*[*OF yun*] **obtain** *m* **where** *set cs* = {(*A i*, *Suc i*)

```
|i. i < m}
    B m = 1 distinct (map snd cs)
    by auto
  thus ?thesis unfolding bs by (auto simp: distinct-map-filter)
qed

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 −
  from yun-monic-factorization-res[OF yun]
  obtain m where bs: set bs = {(A i, Suc i) | i. i < m ∧ A i ≠ 1} and B: B m
= 1
    and dist: distinct (map snd bs) by auto
  have id: {0 ..< m} = {i. i < m ∧ A i = 1} ∪ {i. i < m ∧ A i ≠ 1} (is - =
?ignore ∪ -) by auto
  have p = (∏ i = 0..<m. A i ^ Suc i)
    by (rule prod-A-is-p-B-bound[OF B])
  also have ... = prod (λ i. A i ^ Suc i) {i. i < m ∧ A i ≠ 1}
    unfolding id by (subst prod.union-disjoint, (force+)[3],
      subst prod.neutral[of ?ignore], auto)
  also have ... = (∏(a, i)∈ set bs. a ^ i) unfolding bs
    by (rule prod.reindex-cong[of (λ n. n − 1) o snd], auto simp: inj-on-def, force)
  finally have 1: p = (∏(a, i)∈ set bs. a ^ i) .
  {
    fix a i
    assume (a,i) ∈ set bs
    then obtain j where A: a = A j A j ≠ 1 and i: i ≠ 0 unfolding bs by auto
    with A-square-free[of j] A-monic[of j] have square-free a ∧ degree a ≠ 0 monic
a i ≠ 0
      by (auto simp: monic-degree-0)
  } note 2 = this
  {
    fix a i b j
    assume ai: (a,i) ∈ set bs and bj: (b,j) ∈ set bs and neq: (a,i) ≠ (b,j)
    then obtain i' j' where a: a = A i' and b: b = A j' and ij': i = Suc i' j =
Suc j'
      unfolding bs by auto
    from neq dist ai bj have neq: i' ≠ j' using a b ij' by blast
    from coprime-A-A [OF neq] have coprime a b unfolding a b .
  } note 3 = this
  have monic p unfolding p
    by (rule monic-prod, insert as-monic, auto intro: monic-power monic-mult)
  hence 4: p ≠ 0 by auto
  from dist have 5: distinct bs unfolding distinct-map ..
  show square-free-factorization p (1,bs)
    unfolding square-free-factorization-def using 1 2 3 4 5
    by auto
  show (b,i) ∈ set bs ⟹ monic b using 2 by auto
```

```
  show distinct (map snd bs) by fact
qed
end

lemma monic-factorization: assumes monic p
  shows ∃ as. monic-factorization as p
proof −
  from monic-irreducible-factorization[OF assms]
  obtain as f where fin: finite as and p: p = (∏ a∈as. a ^ Suc (f a))
    and as: as ⊆ {q. irreducible_d q ∧ monic q}
    by auto
  define cs where cs = {(a, f a) | a. a ∈ as}
  show ?thesis
  proof (rule exI, standard)
    show finite cs unfolding cs-def using fin by auto
    {
      fix a i
      assume (a,i) ∈ cs
      thus irreducible_d a monic a unfolding cs-def using as by auto
    } note irr = this
     show ⋀a i b j. (a, i) ∈ cs ⟹ (b, j) ∈ cs ⟹ (a, i) ≠ (b, j) ⟹ a ≠ b
unfolding cs-def by auto
    show p = (∏(a, i)∈cs. a ^ Suc i) unfolding p cs-def
      by (rule prod.reindex-cong, auto, auto simp: inj-on-def)
  qed
qed

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 −
  from monic-factorization[OF assms] obtain as where monic-factorization as p
..
  from monic-factorization.square-free-monic-poly[OF this] show ?thesis .
qed

lemma yun-factorization-induct:
  assumes base: ⋀ bn cn. bn = 1 ⟹ P bn cn
  and step: ⋀ 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 −
  from monic-factorization[OF monic] obtain as where monic-factorization as p
..
   from monic-factorization.yun-factorization-induct[OF this base step id] show
```

?thesis .
qed

lemma *square-free-poly*:
(*poly* (*square-free-poly gcd p*) *x* = *0*) = (*poly p x* = *0*)
proof (*cases p* = *0*)
case *True*
thus *?thesis* **unfolding** *square-free-poly-def* **by** *auto*
next
case *False*
let *?c* = *coeff p* (*degree p*)
let *?ic* = *inverse ?c*
have *id*: *square-free-poly gcd p* = *yun-gcd.square-free-monic-poly gcd* (*smult ?ic p*)
unfolding *square-free-poly-def* **using** *False* **by** *auto*
from *False* **have** *mon*: *monic* (*smult ?ic p*) **and** *ic*: *?ic* ≠ *0* **by** *auto*
show *?thesis* **unfolding** *id square-free-monic-poly*[*OF mon*]
using *ic* **by** *simp*
qed

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 −
from *monic-factorization*[*OF monic*] **obtain** *as* **where** *monic-factorization as p*
..
from *monic-factorization.yun-monic-factorization*[*OF this res*]
show *square-free-factorization p* (*1*,*bs*) (*b*,*i*) ∈ *set bs* ⟹ *monic b distinct* (*map snd bs*)
by *auto*
qed

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 −
from *sf*[*unfolded square-free-factorization-def split*]
have *p*: *p* = *smult d* (∏ (*a*, *i*)∈*set bs*. *a* ^ *i*)
and *eq*: *p* = *0* ⟶ *d* = *0* ∧ *bs* = [] **by** *blast*+
from *eq c* **have** *eq*: *smult c p* = *0* ⟶ *c* ∗ *d* = *0* ∧ *bs* = [] **by** *auto*
from *p* **have** *p*: *smult c p* = *smult* (*c* ∗ *d*) (∏ (*a*, *i*)∈*set bs*. *a* ^ *i*) **by** *auto*
from *eq p sf* **show** *?thesis* **unfolding** *square-free-factorization-def* **by** *blast*
qed

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 −
interpret *yun-gcd gcd* .
note *res* = *res*[*unfolded yun-factorization-def Let-def*]
have *square-free-factorization p c-bs* ∧ ((*b*,*i*) ∈ *set* (*snd c-bs*) ⟶ *monic b*)
proof (*cases p* = *0*)
case *True*
with *res* **have** *c-bs* = (*0*, []) **by** *auto*
thus *?thesis* **unfolding** *True* **by** (*auto simp*: *square-free-factorization-def*)
next
case *False*
let *?c* = *coeff p* (*degree p*)
let *?ic* = *inverse ?c*
obtain *c bs* **where** *cbs*: *c-bs* = (*c*,*bs*) **by** *force*
with *False res*
have *c*: *c* = *?c ?c* ≠ *0* **and** *fact*: *yun-monic-factorization* (*smult ?ic p*) = *bs*
by *auto*
from *False* **have** *mon*: *monic* (*smult ?ic p*) **by** *auto*
from *yun-monic-factorization*[*OF fact mon*]
have *sff*: *square-free-factorization* (*smult ?ic p*) (*1*, *bs*) (*b*, *i*) ∈ *set bs* ⟹ *monic*
b **by** *auto*
have *id*: *smult ?c* (*smult ?ic p*) = *p* **using** *False* **by** *auto*
from *square-free-factorization-smult*[*OF c*(*2*) *sff*(*1*), *unfolded id*] *sff*
show *?thesis* **unfolding** *cbs c* **by** *simp*
qed
thus *square-free-factorization p c-bs* (*b*,*i*) ∈ *set* (*snd c-bs*) ⟹ *monic b* **by** *blast*+
qed

lemma *prod-list-pow*: (∏ *x*←*bs*. (*x* :: *'a* :: *comm-monoid-mult*) ^ *i*)
= *prod-list bs* ^ *i*
by (*induct bs*, *auto simp*: *field-simps*)

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 −
note *sff* = *square-free-factorizationD*[*OF sff*]
let *?prod* = (∏ (*a*, *i*)∈*set bs*. *a* ^ *i*)
from *sff* **have** *pf*: *p* = *smult c ?prod* **by** *blast*

```
  with p have c: c ≠ 0 by auto
  have ord: order x p = order x ?prod unfolding pf
    using order-smult[OF c] by auto
  define q where q = [: −x, 1 :]
  have q0: q ≠ 0 unfolding q-def by auto
  have iq: irreducible q by (auto simp: q-def)
  from rt have qa: q dvd a unfolding q-def poly-eq-0-iff-dvd .
  then obtain b where aqb: a = q * b unfolding dvd-def by auto
  from sff(2)[OF ai] have sq: square-free a and mon: degree a ≠ 0 by auto
  let ?rem = (∏(a, i)∈set bs − {(a,i)}. a ^ i)
  have p0: ?prod ≠ 0 using p pf by auto
  have ?prod = a ^ i * ?rem
    by (subst prod.remove[OF - ai], auto)
  also have a ^ i = q ^ i * b ^ i unfolding aqb by (simp add: field-simps)
  finally have id: ?prod = q ^ i * (b ^ i * ?rem) by simp
  hence dvd: q ^ i dvd ?prod by auto
  {
    assume q ^ Suc i dvd ?prod
    hence q dvd ?prod div q ^ i
      by (metis dvd dvd-0-left-iff dvd-div-iff-mult p0 power-Suc)
    also have ?prod div q ^ i = b ^ i * ?rem
      unfolding id by (rule nonzero-mult-div-cancel-left, insert q0, auto)
    finally have q dvd b ∨ q dvd ?rem
      using iq irreducible-dvd-pow[OF iq] by auto
    hence False
    proof
      assume q dvd b
      with aqb have q * q dvd a by auto
      with sq[unfolded square-free-def] mon iq show False
        unfolding irreducible_d-def by auto
    next
      assume q dvd ?rem
      from irreducible-dvd-prod[OF iq this]
      obtain b j where bj: (b,j) ∈ set bs and neq: (a,i) ≠ (b,j) and dvd: q dvd b
^ j by auto
      from irreducible-dvd-pow[OF iq dvd] have qb: q dvd b .
      from sff(3)[OF ai bj neq] have gcd: coprime a b .
      from qb qa have q dvd gcd a b by simp
      from dvd-imp-degree-le[OF this[unfolded gcd]] iq q0 show False
        using gcd by auto
    qed
  }
  hence ndvd: ¬ q ^ Suc i dvd ?prod by blast
  with dvd have order x ?prod = i unfolding q-def
    by (metis order-unique-lemma)
  thus ?thesis unfolding ord .
qed

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 (rule ccontr)
  assume o0: order x p ≠ 0
  with order-root[of p x] p have 0: poly p x = 0 by auto
  note sff = square-free-factorizationD[OF sff]
  let ?prod = (∏(a, i)∈set bs. a ^ i)
  from sff have pf: p = smult c ?prod by blast
  with p have c: c ≠ 0 by auto
  with 0 have 0: poly ?prod x = 0 unfolding pf by auto
  define q where q = [: -x, 1 :]
  from 0 have dvd: q dvd ?prod unfolding poly-eq-0-iff-dvd by (simp add: q-def)

  have q0: q ≠ 0 unfolding q-def by auto
  have iq: irreducible q by (unfold q-def, auto intro:)
  from irreducible-dvd-prod[OF iq dvd]
  obtain a i where ai: (a,i) ∈ set bs and dvd: q dvd a ^ Suc i by auto
  from irreducible-dvd-pow[OF iq dvd] have dvd: q dvd a .
  hence poly a x = 0 unfolding q-def by (simp add: poly-eq-0-iff-dvd q-def)
  with no-root[OF ai] show False by simp
qed

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 -
  note mem = square-free-factorization-order-root-mem[OF sff p]
  note no-mem = square-free-factorization-order-root-no-mem[OF sff p]
  show ?thesis
  proof
    assume ?r1 ∨ ?r2
    thus ?l
    proof
      assume ?r2
      then obtain a j where aj: (a,j) ∈ set bs poly a x = 0 and i: i = j by auto
      from mem[OF aj] i show ?l by simp
    next
      assume ?r1
      with no-mem[of x] show ?l by auto
    qed
  next
    assume ?l
    show ?r1 ∨ ?r2
    proof (cases ∃ a j. (a, j) ∈ set bs ∧ poly a x = 0)
      case True
```

```
      then obtain a j where (a, j) ∈ set bs poly a x = 0 by auto
      with mem[OF this] ‹?l›
      have ?r2 by auto
      thus ?thesis ..
    next
      case False
      with no-mem[of x] ‹?l› have ?r1 by auto
      thus ?thesis ..
    qed
  qed
qed

lemma square-free-factorization-root:
  assumes sff: square-free-factorization p (c,bs)
    and p: p ≠ (0 :: 'a poly)
  shows {x. poly p x = 0} = {x. ∃ a i. (a,i) ∈ set bs ∧ poly a x = 0}
  using square-free-factorization-order-root[OF sff p] p
    square-free-factorizationD(2)[OF sff]
  unfolding order-root by auto

lemma square-free-factorizationD': fixes p :: 'a poly
  assumes sf: square-free-factorization p (c, bs)
  shows p = smult c (∏(a, i) ← bs. a ^ i)
    and square-free (prod-list (map fst bs))
    and ⋀ b i. (b,i) ∈ set bs ⟹ degree b > 0 ∧ i > 0
    and p = 0 ⟹ c = 0 ∧ bs = []
proof −
  note sf = square-free-factorizationD[OF sf]
  show p = smult c (∏(a, i) ← bs. a ^ i) unfolding sf(1) using sf(5)
    by (simp add: prod.distinct-set-conv-list)
  show bs: ⋀ b i. (b,i) ∈ set bs ⟹ degree b > 0 ∧ i > 0 using sf(2) by auto
  show p = 0 ⟹ c = 0 ∧ bs = [] using sf(4) .
  show square-free (prod-list (map fst bs))
  proof (rule square-freeI)
    from bs have ⋀ b. b ∈ set (map fst bs) ⟹ b ≠ 0 by fastforce
    thus prod-list (map fst bs) ≠ 0 unfolding prod-list-zero-iff by auto
    fix q
    assume degree q > 0 q * q dvd prod-list (map fst bs)
    from irreducible_d-factor[OF this(1)] this(2) obtain q where
      irr: irreducible q and dvd: q * q dvd prod-list (map fst bs) unfolding dvd-def
by auto
    hence dvd': q dvd prod-list (map fst bs) unfolding dvd-def by auto
    from irreducible-dvd-prod-list[OF irr dvd'] obtain b i where
      mem: (b,i) ∈ set bs and dvd1: q dvd b by auto
    from dvd1 obtain k where b: b = q * k unfolding dvd-def by auto
    from split-list[OF mem] b obtain bs1 bs2 where bs: bs = bs1 @ (b, i) # bs2
by auto
    from irr have q0: q ≠ 0 and dq: degree q > 0 unfolding irreducible_d-def by
auto
```

```
    from sf(2)[OF mem, unfolded b] have square-free (q * k) by auto
    from this[unfolded square-free-def, THEN conjunct2, rule-format, OF dq]
    have qk: ¬ q dvd k by simp
     from dvd[unfolded bs b] have q * q dvd q * (k * prod-list (map fst (bs1 @
bs2)))
      by (auto simp: ac-simps)
    with q0 have q dvd k * prod-list (map fst (bs1 @ bs2)) by auto
    with irr qk have q dvd prod-list (map fst (bs1 @ bs2)) by auto
    from irreducible-dvd-prod-list[OF irr this] obtain b' i' where
      mem': (b',i') ∈ set (bs1 @ bs2) and dvd2: q dvd b' by fastforce
    from dvd1 dvd2 have q dvd gcd b b' by auto
    with dq is-unit-iff-degree[OF q0] have cop: ¬ coprime b b' by force
    from mem' have (b',i') ∈ set bs unfolding bs by auto
    from sf(3)[OF mem this] cop have b': (b',i') = (b,i)
      by (auto simp add: coprime-iff-gcd-eq-1)
    with mem' sf(5)[unfolded bs] show False by auto
  qed
qed


lemma square-free-factorizationI': fixes p :: 'a poly
  assumes prod: p = smult c (∏(a, i) ← bs. a ^ i)
    and sf: square-free (prod-list (map fst bs))
    and deg: ⋀ b i. (b,i) ∈ set bs ⟹ degree b > 0 ∧ i > 0
    and 0: p = 0 ⟹ c = 0 ∧ bs = []
  shows square-free-factorization p (c, bs)
  unfolding square-free-factorization-def split
proof (intro conjI impI allI)
  show p = 0 ⟹ c = 0 p = 0 ⟹ bs = [] using 0 by auto
  {
    fix b i
    assume bi: (b,i) ∈ set bs
    from deg[OF this] show degree b > 0 0 < i by auto
    have b dvd prod-list (map fst bs)
      by (intro prod-list-dvd, insert bi, force)
    from square-free-factor[OF this sf] show square-free b .
  }
  show dist: distinct bs
  proof (rule ccontr)
    assume ¬ ?thesis
    from not-distinct-decomp[OF this] obtain bs1 bs2 bs3 b i where
      bs: bs = bs1 @ [(b,i)] @ bs2 @ [(b,i)] @ bs3 by force
    hence b * b dvd prod-list (map fst bs) by auto
    with sf[unfolded square-free-def, THEN conjunct2, rule-format, of b]
    have db: degree b = 0 by auto
    from bs have (b,i) ∈ set bs by auto
    from deg[OF this] db show False by auto
  qed
  show p = smult c (∏(a, i)∈set bs. a ^ i) unfolding prod using dist
```

```
    by (simp add: prod.distinct-set-conv-list)
  {
    fix a i b j
    assume ai: (a, i) ∈ set bs and bj: (b, j) ∈ set bs and diff: (a, i) ≠ (b, j)
    from split-list[OF ai] obtain bs1 bs2 where bs: bs = bs1 @ (a,i) # bs2 by
auto
    with bj diff have (b,j) ∈ set (bs1 @ bs2) by auto
    from split-list[OF this] obtain cs1 cs2 where cs: bs1 @ bs2 = cs1 @ (b,j) #
cs2 by auto
    have prod-list (map fst bs) = a * prod-list (map fst (bs1 @ bs2)) unfolding bs
by simp
    also have ... = a * b * prod-list (map fst (cs1 @ cs2)) unfolding cs by simp
    finally obtain c where lp: prod-list (map fst bs) = a * b * c by auto
    from deg[OF ai] have 0: gcd a b ≠ 0 by auto
    have gcd: gcd a b * gcd a b dvd prod-list (map fst bs)
      unfolding lp by (simp add: mult-dvd-mono)
    {
      assume degree (gcd a b) > 0
      from sf[unfolded square-free-def, THEN conjunct2, rule-format, OF this] gcd
      have False by simp
    }
    hence degree (gcd a b) = 0 by auto
    with 0 show coprime a b using is-unit-gcd is-unit-iff-degree by blast
  }
qed

lemma square-free-factorization-def': fixes p :: 'a poly
  shows square-free-factorization p (c,bs) ⟷
  (p = smult c (∏(a, i) ← bs. a ^ i)) ∧
  (square-free (prod-list (map fst bs))) ∧
  (∀ b i. (b,i) ∈ set bs ⟶ degree b > 0 ∧ i > 0) ∧
  (p = 0 ⟶ c = 0 ∧ bs = [])
  using square-free-factorizationD'[of p c bs]
  square-free-factorizationI'[of p c bs] by blast

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: ⋀ b. b ∈ set bs ⟹ degree b > 0
  shows square-free-factorization p (c * b^i, bs1 @ map (λ b. (b,i)) bs @ bs2)
proof −
  from square-free-factorizationD'(3)[OF sff, of smult b (prod-list bs) i]
  have b: b ≠ 0 by auto
  note sff = square-free-factorizationD'[OF sff]
  show ?thesis
  proof (intro square-free-factorizationI', goal-cases)
    case 1
   thus ?case unfolding sff(1) by (simp add: o-def ac-simps smult-power prod-list-pow)
  next
```

```
    case 2
   show ?case using sff(2) by (simp add: ac-simps o-def square-free-smult-iff[OF
b])
  next
    case 3
    with sff(3) bs show ?case by auto
  next
    case 4
    from sff(4)[OF this] show ?case by simp
  qed
qed

lemma square-free-factorization-further-factorization: fixes p :: 'a poly
  assumes sff: square-free-factorization p (c, bs)
  and bs: ⋀ b i d fs. (b,i) ∈ set bs ⟹ f b = (d,fs)
    ⟹ b = smult d (prod-list fs) ∧ (∀ f ∈ set fs. degree f > 0)
  and h: h = (λ (b,i). case f b of (d,fs) ⇒ (d^i,map (λ f. (f,i)) fs))
  and gs: gs = map h bs
  and d: d = c * prod-list (map fst gs)
  and es: es = concat (map snd gs)
  shows square-free-factorization p (d, es)
proof −
  note sff = square-free-factorizationD'[OF sff]
  show ?thesis
  proof (rule square-free-factorizationI')
    assume p = 0
    from sff(4)[OF this] show d = 0 ∧ es = [] unfolding d es gs by auto
  next
    have id: (∏ (a, i)←bs. a ^ i) = smult (prod-list (map fst gs)) (∏ (a, i)←es. a
^ i)
      unfolding es gs h map-map o-def using bs
    proof (induct bs)
      case (Cons bi bs)
      obtain b i where bi: bi = (b,i) by force
      obtain d fs where f: f b = (d,fs) by force
      from Cons(2)[OF - f, of i] have b: b = smult d (prod-list fs) unfolding bi
by auto
      note IH = Cons(1)[OF Cons(2), of λ - i - - . i]
      show ?case unfolding bi
        by (simp add: f o-def, simp add: b ac-simps, subst IH,
          auto simp: smult-power prod-list-pow ac-simps)
    qed simp
    show p = smult d (∏ (a, i)←es. a ^ i) unfolding sff(1) using id
      by (simp add: d)
  next
    fix fi i
    assume fi: (fi, i) ∈ set es
    from this[unfolded es] obtain G where G: G ∈ snd ‘ set gs and fi: (fi,i) ∈
set G by auto
```

```
    from G[unfolded gs] fi obtain b where bi: (b,i) ∈ set bs
      and G: G = snd (h (b,i)) by (auto simp: h split: prod.splits)
    from sff(3)[OF bi] have i: i > 0 ..
    obtain d fs where f: f b = (d,fs) by force
    have degree fi > 0
      by (rule bs[THEN conjunct2, rule-format, OF bi f], insert fi G f, unfold h,
auto)
    with i show degree fi > 0 ∧ i > 0 by auto
  next
    have id: ∃ c. prod-list (map fst bs) = smult c (prod-list (map fst es))
      unfolding es gs map-map o-def using bs
    proof (induct bs)
      case (Cons bi bs)
      obtain b i where bi: bi = (b,i) by force
      obtain d fs where f: f b = (d,fs) by force
      from Cons(2)[OF - f, of i] have b: b = smult d (prod-list fs) unfolding bi
by auto
      have ∃ c. prod-list (map fst bs) = smult c (prod-list (map fst (concat (map
(λx. snd (h x)) bs))))
        by (rule Cons(1), rule Cons(2), auto)
      then obtain c where
        IH: prod-list (map fst bs) = smult c (prod-list (map fst (concat (map (λx.
snd (h x)) bs)))) by auto
      show ?case unfolding bi
        by (intro exI[of - c * d], auto simp: b IH, auto simp: h f[unfolded b] o-def)
    qed (intro exI[of - 1], auto)
    then obtain c where prod-list (map fst bs) = smult c (prod-list (map fst es))
by blast
    from sff(2)[unfolded this] show square-free (prod-list (map fst es))
      by (metis smult-eq-0-iff square-free-def square-free-smult-iff)
  qed
qed

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: ⋀ b. b ∈ set bs ⟹ degree b > 0
  shows square-free-factorization p (c, bs1 @ map (λ b. (b,i)) bs @ bs2)
  using square-free-factorization-smult-prod-listI[of p c bs1 1 bs i bs2] sff bs by
auto

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))
  using square-free-factorization-prod-listI[of p c bs1 [r,s] i bs2] sff r s a by auto

end
```

lemma *monic-square-free-irreducible-factorization*: **assumes** *mon*: *monic* (*f* :: *'b* :: *field poly*)
and *sf*: *square-free f*
shows ∃ *P*. *finite P* ∧ *f* = ∏ *P* ∧ *P* ⊆ {*q*. *irreducible q* ∧ *monic q*}
proof −
from *mon* **have** *f0*: *f* ≠ *0* **by** *auto*
from *monic-irreducible-factorization*[*OF assms*(*1*)] **obtain** *P n* **where**
P: *finite P P* ⊆ {*q*. *irreducible*$_d$ *q* ∧ *monic q*} **and** *f*: *f* = (∏ *a*∈*P*. *a* ^ *Suc* (*n a*)) **by** *auto*
have ∗: ∀ *a* ∈ *P*. *n a* = *0*
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *a* **where** *a*: *a* ∈ *P* **and** *n*: *n a* ≠ *0* **by** *auto*
have *f* = *a* ^ (*Suc* (*n a*)) ∗ (∏ *b*∈*P* − {*a*}. *b* ^ *Suc* (*n b*))
unfolding *f* **by** (*rule prod.remove*[*OF P*(*1*) *a*])
with *n* **have** *a* ∗ *a dvd f* **by** (*cases n a*, *auto*)
with *sf*[*unfolded square-free-def*] *f0* **have** *degree a* = *0* **by** *auto*
with *a P*(*2*)[*unfolded irreducible*$_d$*-def*] **show** *False* **by** *auto*
qed
have *f* = ∏ *P* **unfolding** *f*
by (*rule prod.cong*[*OF refl*], *insert* ∗, *auto*)
with *P* **show** *?thesis* **by** *auto*
qed

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*: ∏ *P* = ∏ *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 (*rule*; *rule subsetI*)
fix *x* **assume** *x*: *x* ∈ *P*
have *irr-x*: *irreducible x* **using** *x P* **by** *auto*
then have ∃ *a*∈*Q*. *x dvd id a*
proof (*rule irreducible-dvd-prod*)
show *x dvd prod id Q* **using** *PQ x*
by (*metis dvd-refl dvd-prod finite-P id-apply prod.cong*)
qed
from *this* **obtain** *a* **where** *a*: *a*∈*Q* **and** *x-dvd-a*: *x dvd a* **unfolding** *id-def* **by** *blast*
have *x*=*a* **using** *x P a Q irreducible*$_d$*-dvd-eq*[*OF - - x-dvd-a*] **by** *fast*
thus *x* ∈ *Q* **using** *a* **by** *simp*
next
fix *x* **assume** *x*: *x* ∈ *Q*

have *irr-x*: *irreducible x* **using** *x Q* **by** *auto*
then have $\exists\, a{\in}P.\ x\ dvd\ id\ a$
proof (*rule irreducible-dvd-prod*)
show *x dvd prod id P* **using** *PQ x*
by (*metis dvd-refl dvd-prod finite-Q id-apply prod.cong*)
qed
from *this* **obtain** *a* **where** *a*: $a{\in}P$ **and** *x-dvd-a*: *x dvd a* **unfolding** *id-def* **by** *blast*
have $x{=}a$ **using** *x P a Q irreducible*$_d$*-dvd-eq*[*OF - - x-dvd-a*] **by** *fast*
thus $x \in P$ **using** *a* **by** *simp*
qed
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′* **..**
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 −
let *?P* = λ *f g*. ∀ *i as*. *yun-gcd.yun-factorization-main gcd* (*hp f*) (*hp g*) *i* (*hpi as*) = *hpi* (*yun-gcd.yun-factorization-main gcd f g i as*)
note *ind* = *yun-factorization-induct*[*OF - - f g monic*, *of ?P*, *rule-format*]
interpret *map-poly-hom*: *map-poly-inj-comm-ring-hom*..
interpret *p*: *inj-comm-ring-hom hp* **unfolding** *hp*..
note *homs* = *map-poly-gcd*[*folded hp*]
map-poly-pderiv[*folded hp*]
p.hom-minus
map-poly-div[*folded hp*]
show *?thesis*
proof (*induct rule*: *ind*)
case (*1 f g i as*)
show *?case* **unfolding** *yun-gcd.yun-factorization-main.simps*[*of - hp f*] *yun-gcd.yun-factorization-main.simps*[*of - f*]
unfolding *1* **by** *simp*
next
case (*2 f g i as*)
have *id*: ⋀ *f i fis*. *hpi* ((*f*,*i*) # *fis*) = (*hp f*, *i*) # *hpi fis* **unfolding** *hpi* **by** *auto*
show *?case* **unfolding** *yun-gcd.yun-factorization-main.simps*[*of - hp f*] *yun-gcd.yun-factorization-main.simps*

```
- f]
      unfolding p.hom-1-iff
      unfolding Let-def
      unfolding homs[symmetric] id[symmetric]
      unfolding 2(2) by simp
  qed
qed

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)])
  by (intro square-free-factorizationI', auto)

lemma constant-square-free-factorization:
  degree p = 0 ⟹ square-free-factorization p (coeff p 0,[])
  by (drule degree0-coeffs [of p]) (auto simp: square-free-factorization-def)

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 −
  interpret map-poly-hom: map-poly-inj-comm-ring-hom..
  interpret p: inj-ring-hom hp unfolding hp..
  have hpiN: hpi [] = [] unfolding hpi by simp
  obtain res where res =
     yun-gcd.yun-factorization-main gcd (f div gcd f (pderiv f)) (pderiv f div gcd f
(pderiv f)) 0 [] by auto
  note homs = map-poly-gcd[folded hp]
       map-poly-pderiv[folded hp]
       p.hom-minus
       map-poly-div[folded hp]
        yun-factorization-main-hom[folded hp, folded hpi, symmetric, OF monic refl
refl, of - Nil, unfolded hpiN]
       this
  show ?thesis
    unfolding yun-gcd.yun-monic-factorization-def Let-def
    unfolding homs[symmetric]
    unfolding hpi
      by (induct res, auto)
qed


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*)
using *yun-monic-factorization*[*of smult* (*inverse* (*coeff f* (*degree f*))) *f*]
unfolding *yun-factorization-def Let-def hp hpi*
by (*auto simp*: *hom-distribs*)

lemma (**in** *field-hom-0'*) *square-free-map-poly*:
square-free (*map-poly hom f*) = *square-free f*
proof −
interpret *map-poly-hom*: *map-poly-inj-comm-ring-hom*..
show *?thesis* **unfolding** *square-free-iff-separable separable-def*
by (*simp only*: *hom-distribs* [*symmetric*])
(*simp add*: *coprime-iff-gcd-eq-1 map-poly-gcd* [*symmetric*])
qed

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 (*intro ext*)
fix *f g*
let *?ri* = *map-poly rat-of-int*
obtain *a' f'* **where** *faf'*: *rat-to-int-poly f* = (*a'*,*f'*) **by** *force*
from *rat-to-int-poly*[*OF this*] **obtain** *a* **where**
f: *f* = *smult a* (*?ri f'*) **and** *a*: *a* ≠ *0* **by** *auto*
obtain *b' g'* **where** *gbg'*: *rat-to-int-poly g* = (*b'*,*g'*) **by** *force*
from *rat-to-int-poly*[*OF this*] **obtain** *b* **where**
g: *g* = *smult b* (*?ri g'*) **and** *b*: *b* ≠ *0* **by** *auto*
define *h* **where** *h* = *gcd f' g'*

let *?h = ?ri h*
define *lc* **where** *lc = inverse* (*coeff ?h* (*degree ?h*))
let *?gcd = smult lc ?h*
have *id*: *gcd-rat-poly f g = ?gcd*
unfolding *lc-def h-def gcd-rat-poly-def Let-def faf' gbg' snd-conv* **by** *auto*
show *gcd-rat-poly f g = gcd f g* **unfolding** *id*
proof (*rule gcdI*)
have *h dvd f'* **unfolding** *h-def* **by** *auto*
hence *?h dvd ?ri f'* **unfolding** *dvd-def* **by** (*auto simp*: *hom-distribs*)
hence *?h dvd f* **unfolding** *f* **by** (*rule dvd-smult*)
thus *dvd-f*: *?gcd dvd f*
by (*metis dvdE inverse-zero-imp-zero lc-def leading-coeff-neq-0 mult-eq-0-iff smult-dvd-iff*)
have *h dvd g'* **unfolding** *h-def* **by** *auto*
hence *?h dvd ?ri g'* **unfolding** *dvd-def* **by** (*auto simp*: *hom-distribs*)
hence *?h dvd g* **unfolding** *g* **by** (*rule dvd-smult*)
thus *dvd-g*: *?gcd dvd g*
by (*metis dvdE inverse-zero-imp-zero lc-def leading-coeff-neq-0 mult-eq-0-iff smult-dvd-iff*)
show *normalize ?gcd = ?gcd*
by (*cases lc = 0*)
(*simp-all add*: *normalize-poly-def pCons-one field-simps lc-def*)
fix *k*
assume *dvd*: *k dvd f k dvd g*
obtain *k' c* **where** *kck*: *rat-to-normalized-int-poly k* = (*c*,*k'*) **by** *force*
from *rat-to-normalized-int-poly*[*OF this*] **have** *k*: *k = smult c* (*?ri k'*) **and** *c*: *c* ≠ *0* **by** *auto*
from *dvd*(*1*) **have** *kf*: *k dvd ?ri f'* **unfolding** *f* **using** *a* **by** (*rule dvd-smult-cancel*)
from *dvd*(*2*) **have** *kg*: *k dvd ?ri g'* **unfolding** *g* **using** *b* **by** (*rule dvd-smult-cancel*)
from *kf kg* **obtain** *kf kg* **where** *kf*: *?ri f' = k * kf* **and** *kg*: *?ri g' = k * kg* **unfolding** *dvd-def* **by** *auto*
from *rat-to-int-factor-explicit*[*OF kf kck*] **have** *kf*: *k' dvd f'* **unfolding** *dvd-def* **by** *blast*
from *rat-to-int-factor-explicit*[*OF kg kck*] **have** *kg*: *k' dvd g'* **unfolding** *dvd-def* **by** *blast*
from *kf kg* **have** *k' dvd h* **unfolding** *h-def* **by** *simp*
hence *?ri k' dvd ?ri h* **unfolding** *dvd-def* **by** (*auto simp*: *hom-distribs*)
hence *k dvd ?ri h* **unfolding** *k* **using** *c* **by** (*rule smult-dvd*)
thus *k dvd ?gcd* **by** (*rule dvd-smult*)
qed
qed

lemma *gcd-rat-poly-unfold*[*code-unfold*]: *gcd = gcd-rat-poly* **by** *simp*
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*:]))
by *pat-completeness auto*

termination by (*relation measure degree*,
auto dest: *rational-root-test*(*1*) *intro*!: *degree-div-less simp*: *poly-eq-0-iff-dvd*)

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 −
note *d* = *roots-of-rat-poly-main.simps*[*of p*] *Let-def*
show *?thesis*
proof (*cases rational-root-test p*)
case (*Some x*)
let *?x* = [:−*x*,*1*:]
from *rational-root-test*(*1*)[*OF Some*] **have** *?x dvd p*
by (*simp add*: *poly-eq-0-iff-dvd*)
from *dvd-mult-div-cancel*[*OF this*]
have *pp*: *p div ?x* = *?x* ∗ (*p div ?x*) *div ?x* **by** *simp*
then show *?thesis* **unfolding** *d Some* **by** *auto*
qed (*simp add*: *d*)
qed

lemma *roots-of-rat-poly-main*: *p* ≠ *0* ⟹ *set* (*roots-of-rat-poly-main p*) = {*x*. *poly p x* = *0*}
proof (*induct p rule*: *roots-of-rat-poly-main.induct*)

```
  case (1 p)
  note IH = 1(1)
  note p = 1(2)
  let ?n = degree p
  let ?rr = roots-of-rat-poly-main
  show ?case
  proof (cases ?n = 0)
    case True
    from roots0[OF p True] True show ?thesis by simp
  next
    case False note 0 = this
    show ?thesis
    proof (cases ?n = 1)
      case True
      from roots1[OF True] True show ?thesis by simp
    next
      case False note 1 = this
      show ?thesis
      proof (cases ?n = 2)
        case True
        from rat-roots2[OF True] True show ?thesis by simp
      next
        case False note 2 = this
        from 0 1 2 have id: ?rr p = (case rational-root-test p of None ⇒ [] | Some
x ⇒
          x # ?rr (p div [: −x, 1 :])) by simp
        show ?thesis
        proof (cases rational-root-test p)
          case None
          from rational-root-test(2)[OF None] None id show ?thesis by simp
        next
          case (Some x)
          from rational-root-test(1)[OF Some] have [: −x, 1:] dvd p
            by (simp add: poly-eq-0-iff-dvd)
          from dvd-mult-div-cancel[OF this]
          have pp: p = [: −x, 1:] ∗ (p div [: −x, 1:]) by simp
          with p have p: p div [:− x, 1:] ≠ 0 by auto
          from arg-cong[OF pp, of λ p. {x. poly p x = 0}]
            rational-root-test(1)[OF Some] IH[OF refl 0 1 2 Some p]  show ?thesis
            unfolding id Some by auto
        qed
      qed
    qed
  qed
qed

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 ≠ 0
  shows set (roots-of-rat-poly p) = {x. poly p x = 0}
proof −
  obtain c pis where yun: yun-factorization gcd p = (c,pis) by force
  from yun
  have res: roots-of-rat-poly p = concat (map (roots-of-rat-poly-main ∘ fst) pis)
    by (auto simp: roots-of-rat-poly-def split: if-splits)
  note yun = square-free-factorizationD(1,2,4)[OF yun-factorization(1)[OF yun]]
  from yun(1) p have c: c ≠ 0 by auto
  from yun(1) have p: p = smult c (∏(a, i)∈set pis. a ^ i) .
  have {x. poly p x = 0} = {x. poly (∏(a, i)∈set pis. a ^ i) x = 0}
    unfolding p using c by auto
  also have ... = ⋃ ((λ p. {x. poly p x = 0}) ` fst ` set pis) (is - = ?r)
    using yun(2) by (subst poly-prod-0, force+)
  finally have r: {x. poly p x = 0} = ?r .
  {
    fix p i
    assume p: (p,i) ∈ set pis
    have set (roots-of-rat-poly-main p) = {x. poly p x = 0}
      by (rule roots-of-rat-poly-main, insert yun(2) p, force)
  } note main = this
  have set (roots-of-rat-poly p) = ⋃ ((λ (p, i). set (roots-of-rat-poly-main p)) ` set
pis)
    unfolding res o-def by auto
  also have ... = ?r using main by auto
  finally show ?thesis unfolding r by simp
qed

definition root-free :: 'a :: comm-semiring-0 poly ⇒ bool where
  root-free p = (degree p = 1 ∨ (∀ x. poly p x ≠ 0))

lemma irreducible-root-free:
  fixes p :: 'a :: idom poly
  assumes irreducible p shows root-free p
proof−
  from assms have p0: p ≠ 0 by auto
  {
    fix x
    assume poly p x = 0 and degp: degree p ≠ 1
    hence [:−x,1:] dvd p using poly-eq-0-iff-dvd by blast
    then obtain q where p: p = [:−x,1:] * q by (elim dvdE)
    with p0 have q0: q ≠ 0 by auto
    from irreducibleD[OF assms p]
    have q dvd 1 by (metis one-neq-zero poly-1 poly-eq-0-iff-dvd)
    then have degree q = 0 by (simp add: poly-dvd-1)
    with degree-mult-eq[of [:−x,1:] q, folded p] q0 degp
```

```
    have False by auto
  }
  thus ?thesis unfolding root-free-def by auto
qed

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,[])
  unfolding factorize-root-free-def by simp

lemma factorize-root-free: assumes res: factorize-root-free p = (c,qs)
  shows p = smult c (prod-list qs)
  ⋀ q. q ∈ set qs ⟹ root-free q ∧ monic q ∧ degree q ≠ 0
proof −
  have p = smult c (prod-list qs) ∧ (∀ q ∈ set qs. root-free q ∧ monic q ∧ degree
q ≠ 0)
  proof (cases degree p = 0)
    case True
     thus ?thesis using res unfolding factorize-root-free-def by (auto dest: de-
gree0-coeffs)
  next
    case False
    hence p0: p ≠ 0 by auto
    define fs where fs = ([] :: rat poly list)
    define xs where xs = roots-of-rat-poly p
    define q where q = p
    obtain n  where n: n = degree q + length xs by auto
    have prod: p = q ∗ prod-list fs unfolding q-def fs-def by auto
   have sub: {x. poly q x = 0} ⊆ set xs using roots-of-rat-poly[OF p0] unfolding
q-def xs-def by auto
    have fs: ⋀ q. q ∈ set fs ⟹ root-free q ∧ monic q ∧ degree q ≠ 0 unfolding
fs-def by auto
    have res: factorize-root-free-main q xs fs = (c,qs) using res False
      unfolding xs-def fs-def q-def factorize-root-free-def by auto
    from False have q ≠ 0 unfolding q-def by auto
    from prod sub fs res n this show ?thesis
    proof (induct n arbitrary: q fs xs rule: wf-induct[OF wf-less])
```

case (*1 n q fs xs*)
note *simp* = *factorize-root-free-main.simps*[*of q xs fs*]
note *IH* = *1*(*1*)[*rule-format*]
note *0* = *1*(*2*−)[*unfolded simp*]
show *?case*
proof (*cases xs*)
case *Nil*
note *0* = *0*[*unfolded Nil Let-def*]
hence *no-rt*: ⋀ *x*. *poly q x* ≠ *0* **by** *auto*
hence *q*: *q* ≠ *0* **by** *auto*
let *?r* = *smult* (*inverse c*) *q*
define *r* **where** *r* = *?r*
from *0*(*4*−*5*) **have** *c*: *c* = *coeff q* (*degree q*) **and** *qs*: *qs* = (*if r* = *1 then fs else r* # *fs*) **by** (*auto simp*: *r-def*)
from *q c qs 0*(*1*) **have** *c0*: *c* ≠ *0* **and** *p*: *p* = *smult c* (*prod-list* (*r* # *fs*)) **by** (*auto simp*: *r-def*)
from *p* **have** *p*: *p* = *smult c* (*prod-list qs*) **unfolding** *qs* **by** *auto*
from *0*(*2*,*5*) *c0 c* **have** *root-free ?r monic ?r*
unfolding *root-free-def* **by** *auto*
with *0*(*3*) **have** ⋀ *q*. *q* ∈ *set qs* ⟹ *root-free q* ∧ *monic q* ∧ *degree q* ≠ *0* **unfolding** *qs*
by (*cases degree q* = *0*, *insert degree0-coeffs*[*of q*], *auto split*: *if-splits simp*: *r-def*)
with *p* **show** *?thesis* **by** *auto*
next
case (*Cons x xs*)
note *0* = *0*[*unfolded Cons*]
show *?thesis*
proof (*cases poly q x* = *0*)
case *True*
let *?q* = *q div* [:−*x*,*1*:]
let *?x* = [:−*x*,*1*:]
let *?fs* = *?x* # *fs*
let *?xs* = *x* # *xs*
from *True* **have** *q*: *q* = *?q* ∗ *?x*
by (*metis dvd-mult-div-cancel mult.commute poly-eq-0-iff-dvd*)
with *0*(*6*) **have** *q'*: *?q* ≠ *0* **by** *auto*
have *deg*: *degree q* = *Suc* (*degree ?q*) **unfolding** *arg-cong*[*OF q*, *of degree*]
by (*subst degree-mult-eq*[*OF q'*], *auto*)
hence *n*: *degree ?q* + *length ?xs* < *n* **unfolding** *0*(*5*) **by** *auto*
from *arg-cong*[*OF q*, *of poly*] *0*(*2*) **have** *rt*: {*x*. *poly ?q x* = *0*} ⊆ *set ?xs* **by** *auto*
have *p*: *p* = *?q* ∗ *prod-list ?fs* **unfolding** *prod-list.Cons 0*(*1*) *mult.assoc*[*symmetric*] *q*[*symmetric*] **..**
have *root-free ?x* **unfolding** *root-free-def* **by** *auto*
with *0*(*3*) **have** *rf*: ⋀ *f*. *f* ∈ *set ?fs* ⟹ *root-free f* ∧ *monic f* ∧ *degree f* ≠ *0* **by** *auto*
from *True 0*(*4*) **have** *res*: *factorize-root-free-main ?q ?xs ?fs* = (*c*,*qs*) **by** *simp*

```
        show ?thesis
          by (rule IH[OF - p rt rf res refl q'], insert n, auto)
      next
        case False
        with 0(4) have res: factorize-root-free-main q xs fs = (c,qs) by simp
         from 0(5) obtain m where m: m = degree q + length xs and n: n =
Suc m by auto
        from False 0(2) have rt: {x. poly q x = 0} ⊆ set xs by auto
        show ?thesis by (rule IH[OF - 0(1) rt 0(3) res m 0(6)], unfold n, auto)
      qed
    qed
  qed
  thus p = smult c (prod-list qs)
    ⋀ q. q ∈ set qs ⟹ root-free q ∧ monic q ∧ degree q ≠ 0 by auto
qed


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: ⋀ x. poly p x ≠ 0
  shows ¬ q dvd p
proof −
  from degree1-coeffs[OF q] obtain a b where q: q = [: b, a :] and a: a ≠ 0
    by metis
  have q: q = smult a [: − (− b / a), 1 :] unfolding q
   by (rule poly-eqI, unfold coeff-smult, insert a, auto simp: field-simps coeff-pCons
      split: nat.splits)
 show ?thesis unfolding q smult-dvd-iff poly-eq-0-iff-dvd[symmetric, of - p] using
a rt by auto
qed


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 −
  let ?rp = rational-proper-factor p
  let ?rr = rational-root-test
```

```
  note d = rational-proper-factor-def[of p]
  have (degree p > 0 ⟶ ?rp = None ⟶ irreducible_d p) ∧
       (?rp = Some q ⟶ q dvd p ∧ degree q ≥ 1 ∧ degree q < degree p)
  proof (cases degree p = 0)
    case True
    thus ?thesis unfolding d by auto
  next
    case False note 0 = this
    show ?thesis
    proof (cases degree p = 1)
      case True
      hence ?rp = None unfolding d by auto
      with linear-irreducible_d[OF True] show ?thesis by auto
    next
      case False note 1 = this
      show ?thesis
      proof (cases degree p = 2)
        case True
        hence rp: ?rp = (case rat-roots2 p of Nil ⇒ None | Cons x xs ⇒ Some
[:−x,1 :]) unfolding d by auto
        show ?thesis
        proof (cases rat-roots2 p)
          case Nil
          with rp have rp: ?rp = None by auto
          from Nil rat-roots2[OF True] have nex: ¬ (∃ x. poly p x = 0) by auto
          have irreducible_d p
          proof (rule irreducible_dI)
            fix q r :: rat poly
            assume degree q > 0 degree q < degree p and p: p = q * r
            with True have dq: degree q = 1 by auto
            have ¬ q dvd p by (rule degree-1-dvd-root[OF dq], insert nex, auto)
            with p show False by auto
          qed (insert True, auto)
          with rp show ?thesis by auto
        next
          case (Cons x xs)
          from Cons rat-roots2[OF True] have poly p x = 0 by auto
          from this[unfolded poly-eq-0-iff-dvd] have x: [: −x , 1 :] dvd p by auto
          from Cons rp have rp: ?rp = Some ([: − x, 1 :]) by auto
          show ?thesis using True x unfolding rp by auto
        qed
      next
        case False note 2 = this
        show ?thesis
        proof (cases degree p = 3)
          case True
          hence rp: ?rp = (case ?rr p of None ⇒ None | Some x ⇒ Some [:− x,
1:]) unfolding d by auto
          show ?thesis
```

```
        proof (cases ?rr p)
          case None
           from rational-root-test(2)[OF None] have nex: ¬ (∃ x. poly p x = 0)
by auto
          from rp[unfolded None] have rp: ?rp = None by auto
          have irreducible_d p
          proof (rule irreducible_dI2)
            fix q :: rat poly
            assume degree q > 0 degree q ≤ degree p div 2
            with True have dq: degree q = 1 by auto
            show ¬ q dvd p
              by (rule degree-1-dvd-root[OF dq], insert nex, auto)
          qed (insert True, auto)
          with rp show ?thesis by auto
        next
          case (Some x)
          from rational-root-test(1)[OF Some] have poly p x = 0 .
          from this[unfolded poly-eq-0-iff-dvd] have x: [: −x , 1 :] dvd p by auto
          from Some rp have rp: ?rp = Some ([: − x, 1 :]) by auto
          show ?thesis using True x unfolding rp by auto
        qed
      next
        case False note 3 = this
        let ?kp = kronecker-factorization-rat p
        from 0 1 2 3 have d4: degree p ≥ 4 and d1: degree p ≥ 1 by auto
        hence rp: ?rp = ?kp using d4 d by auto
        show ?thesis
        proof (cases ?kp)
          case None
           with rp kronecker-factorization-rat(2)[OF None d1] show ?thesis by
auto
        next
          case (Some q)
          with rp kronecker-factorization-rat(1)[OF Some] show ?thesis by auto
        qed
      qed
    qed
  qed
  thus 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 by auto
qed

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*)))
by *pat-completeness auto*

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*
unfolding *factorize-rat-poly-main-wf-rel-def* **using** *wf-mult1*[*OF wf-less*] **by** *auto*

lemma *factorize-rat-poly-main-wf-rel-sub*:
((*a*, *b*, *ps*), (*c*, *d*, *p* # *ps*)) ∈ *factorize-rat-poly-main-wf-rel*
unfolding *factorize-rat-poly-main-wf-rel-def*
by (*auto intro*: *mult1I* [*of* - - - - {#}])

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*
unfolding *factorize-rat-poly-main-wf-rel-def mult1-def*
using *add-eq-conv-ex assms ab-semigroup-add-class.add-ac*
by *fastforce*

termination
proof (*relation factorize-rat-poly-main-wf-rel*,
rule wf-factorize-rat-poly-main-wf-rel, *rule factorize-rat-poly-main-wf-rel-sub*,
rule factorize-rat-poly-main-wf-rel-sub, *rule factorize-rat-poly-main-wf-rel-two*)
fix *p q*
assume *rf*: *rational-proper-factor p* = *Some q* **and** *dp*: *degree p* ≠ *0*
from *rational-proper-factor*(*2*)[*OF rf*]
have *dvd*: *q dvd p* **and** *deg*: *1* ≤ *degree q degree q* < *degree p* **by** *auto*
show *degree q* < *degree p* **by** *fact*
from *dvd* **have** *p* = *q* ∗ (*p div q*) **by** *auto*
from *arg-cong*[*OF this*, *of degree*]
have *degree p* = *degree q* + *degree* (*p div q*)
by (*subst degree-mult-eq*[*symmetric*], *insert dp*, *auto*)
with *deg*
show *degree* (*p div q*) < *degree p* **by** *simp*
qed

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 (*atomize*(*full*), *insert assms*, *induct c irr ps rule*: *factorize-rat-poly-main.induct*)
case (*1 c irr*)

```
  thus ?case by (auto simp: factorize-rat-poly-main.simps)
next
  case (2 c irr p ps)
  note IH = 2(1-3)
  note res = 2(4)[unfolded factorize-rat-poly-main.simps(2)[of c irr p ps]]
  note irr = 2(5)
  let ?f = factorize-rat-poly-main
  show ?case
  proof (cases degree p = 0)
    case True
    with res have res: ?f (c * coeff p 0) irr ps = (d,qs) by simp
    from degree0-coeffs[OF True] obtain a where p: p = [: a :] by auto
    from IH(1)[OF True res irr]
    show ?thesis using p by simp
  next
    case False
    note IH = IH(2-)[OF False]
    from False have (degree p = 0) = False by auto
    note res = res[unfolded this if-False]
    let ?rf = rational-proper-factor p
    show ?thesis
    proof (cases ?rf)
      case None
      with res have res: ?f c (p # irr) ps = (d,qs) by auto
      from rational-proper-factor(1)[OF - None] False
      have irp: irreducible_d p by auto
      note IH(1)[OF None res, unfolded atomize-imp imp-conjR, simplified]
      note 1 = conjunct1[OF this, rule-format] conjunct2[OF this, rule-format]
      from irr irp show ?thesis by (auto intro:1 simp: ac-simps)
    next
      case (Some q)
      define pq where pq = p div q
      from Some res have res: ?f c irr (q # pq # ps) = (d,qs) unfolding pq-def
by auto
      from rational-proper-factor(2)[OF Some] have q dvd p by auto
      hence p: p = q * pq unfolding pq-def by auto
      from IH(2)[OF Some, folded pq-def, OF res irr] show ?thesis unfolding p
        by (auto simp: ac-simps)
    qed
  qed
qed

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)
  ⋀ q. q ∈ set qs ⟹ irreducible_d q
   using factorize-rat-poly-main[OF res[unfolded factorize-rat-poly-basic-def]] by
auto
```

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.