Subresultants*

Sebastiaan Joosten, René Thiemann and Akihisa Yamada

December 7, 2022

Abstract

We formalize the theory of subresultants and the subresultant polynomial remainder sequence as described by Brown and Traub. As a result, we obtain efficient certified algorithms for computing the resultant and the greatest common divisor of polynomials.

Contents

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

1 Introduction

Computing the gcd of two polynomials can be done via the Euclidean algorithm, if the domain of the polynomials is a field. For non-field polynomials, one has to replace the modulo operation by the pseudo-modulo operation, which results in the exponential growth of coefficients in the gcd algorithm. To counter this problem, one may divide the intermediate polynomials by their contents in every iteration of the gcd algorithm. This is precisely the way how currently resultants and gcds are computed in Isabelle.

Computing contents in every iteration is a costly operation, and therefore Brown and Traub have developed the subresultant PRS (polynomial remainder sequence) algorithm [1, 2]. It avoids intermediate content computation and at the same time keeps the coefficients small, i.e., the coefficients grow at most polynomially.

The soundness of the subresultant PRS gcd algorithm is in principle similar to the Euclidean algorithm, i.e., the intermediate polynomials that are computed in both algorithms differ only by a constant factor. The major problem is to prove that all the performed divisions are indeed exact divisions. To this end, we formalize the fundamental theorem of Brown and Traub as well as the resulting algorithms by following the original (condensed) proofs. This is in contrast to a similar Coq formalization by Mahboubi [4], which follows another proof based on polynomial determinants.

As a consequence of the new algorithms, we significantly increased the speed of the algebraic number implementation [5] which heavily relies upon the computation of resultants of bivariate polynomials.

2 Resultants

This theory defines the Sylvester matrix and the resultant and contains basic facts about these notions. After the connection between resultants and subresultants has been established, we then use properties of subresultants to transfer them to resultants. Remark: these properties have previously been proven separately for both resultants and subresultants; and this is the reason for splitting the theory of resultants in two parts, namely "Resultant-Prelim" and "Resultant" which is located in the Algebraic-Number AFP-entry.

theory *Resultant-Prelim*
imports
Jordan-Normal-Form.Determinant
Polynomial-Interpolation.Ring-Hom-Poly
begin

Sylvester matrix

definition *sylvester-mat-sub* :: *nat* ⇒ *nat* ⇒ *'a poly* ⇒ *'a poly* ⇒ *'a* :: *zero mat*
where

```
  sylvester-mat-sub m n p q ≡
   mat (m+n) (m+n) (λ (i,j).
     if i < n then
       if i ≤ j ∧ j − i ≤ m then coeff p (m + i − j) else 0
     else if i − n ≤ j ∧ j ≤ i then coeff q (i−j) else 0)

definition sylvester-mat :: 'a poly ⇒ 'a poly ⇒ 'a :: zero mat where
  sylvester-mat p q ≡ sylvester-mat-sub (degree p) (degree q) p q

lemma sylvester-mat-sub-dim[simp]:
  fixes m n p q
  defines S ≡ sylvester-mat-sub m n p q
  shows dim-row S = m+n and dim-col S = m+n
  unfolding S-def sylvester-mat-sub-def by auto

lemma sylvester-mat-sub-carrier:
  shows sylvester-mat-sub m n p q ∈ carrier-mat (m+n) (m+n) by auto

lemma sylvester-mat-dim[simp]:
  fixes p q
  defines d ≡ degree p + degree q
  shows dim-row (sylvester-mat p q) = d dim-col (sylvester-mat p q) = d
  unfolding sylvester-mat-def d-def by auto

lemma sylvester-carrier-mat:
  fixes p q
  defines d ≡ degree p + degree q
  shows sylvester-mat p q ∈ carrier-mat d d unfolding d-def by auto

lemma sylvester-mat-sub-index:
  fixes p q
  assumes i: i < m+n and j: j < m+n
  shows sylvester-mat-sub m n p q $$ (i,j) =
    (if i < n then
       if i ≤ j ∧ j − i ≤ m then coeff p (m + i − j) else 0
     else if i − n ≤ j ∧ j ≤ i then coeff q (i−j) else 0)
  unfolding sylvester-mat-sub-def
  unfolding index-mat(1)[OF i j] by auto

lemma sylvester-index-mat:
  fixes p q
  defines m ≡ degree p and n ≡ degree q
  assumes i: i < m+n and j: j < m+n
  shows sylvester-mat p q $$ (i,j) =
    (if i < n then
       if i ≤ j ∧ j − i ≤ m then coeff p (m + i − j) else 0
     else if i − n ≤ j ∧ j ≤ i then coeff q (i − j) else 0)
  unfolding sylvester-mat-def
  using sylvester-mat-sub-index[OF i j] unfolding m-def n-def.
```

lemma *sylvester-index-mat2*:
fixes $p\ q :: 'a :: comm\text{-}semiring\text{-}1\ poly$
defines $m \equiv degree\ p$ **and** $n \equiv degree\ q$
assumes *i*: $i < m+n$ **and** *j*: $j < m+n$
shows $sylvester\text{-}mat\ p\ q\ \$\$\ (i,j) =$
$(if\ i < n\ then\ coeff\ (monom\ 1\ (n - i) * p)\ (m+n-j)$
$else\ coeff\ (monom\ 1\ (m + n - i) * q)\ (m+n-j))$
apply(*subst sylvester-index-mat*)
unfolding *m-def*[*symmetric*] *n-def*[*symmetric*]
using *i j* **apply** (*simp*,*simp*)
unfolding *coeff-monom-mult*
apply(*cases* $i < n$)
apply (*cases* $i \le j \wedge j - i \le m$)
using *j m-def* **apply** (*force*, *force simp*: *coeff-eq-0*)
apply (*cases* $i - n \le j \wedge j \le i$)
using *i j coeff-eq-0*[*of q*] *n-def* **by** *auto*

lemma *sylvester-mat-sub-0*[*simp*]: $sylvester\text{-}mat\text{-}sub\ 0\ n\ 0\ q = 0_m\ n\ n$
unfolding *sylvester-mat-sub-def* **by** *auto*

lemma *sylvester-mat-0*[*simp*]: $sylvester\text{-}mat\ 0\ q = 0_m\ (degree\ q)\ (degree\ q)$
unfolding *sylvester-mat-def* **by** *simp*

lemma *sylvester-mat-const*[*simp*]:
fixes $a :: 'a :: semiring\text{-}1$
shows $sylvester\text{-}mat\ [:a:]\ q = a \cdot_m 1_m\ (degree\ q)$
and $sylvester\text{-}mat\ p\ [:a:] = a \cdot_m 1_m\ (degree\ p)$
by(*auto simp*: *sylvester-index-mat*)

lemma *sylvester-mat-sub-map*:
assumes *f0*: $f\ 0 = 0$
shows $map\text{-}mat\ f\ (sylvester\text{-}mat\text{-}sub\ m\ n\ p\ q) = sylvester\text{-}mat\text{-}sub\ m\ n\ (map\text{-}poly\ f\ p)\ (map\text{-}poly\ f\ q)$
(**is** $?l = ?r$)
proof(*rule eq-matI*)
note [*simp*] = *coeff-map-poly*[*of f*, *OF f0*]
show *dim*: $dim\text{-}row\ ?l = dim\text{-}row\ ?r\ dim\text{-}col\ ?l = dim\text{-}col\ ?r$ **by** *auto*
fix *i j*
assume *ij*: $i < dim\text{-}row\ ?r\ j < dim\text{-}col\ ?r$
note $ij' = this$[*unfolded sylvester-mat-sub-dim*]
note $ij'' = ij$[*unfolded dim*[*symmetric*] *index-map-mat*]
show $?l\ \$\$\ (i, j) = ?r\ \$\$\ (i,j)$
unfolding *index-map-mat*(*1*)[*OF ij''*]
unfolding *sylvester-mat-sub-index*[*OF ij'*]
unfolding *Let-def*
using *f0* **by** *auto*
qed

definition *resultant* :: *'a poly* ⇒ *'a poly* ⇒ *'a* :: *comm-ring-1* **where**
resultant p q = *det* (*sylvester-mat p q*)

Resultant, but the size of the base Sylvester matrix is given.

definition *resultant-sub m n p q* = *det* (*sylvester-mat-sub m n p q*)

lemma *resultant-sub*: *resultant p q* = *resultant-sub* (*degree p*) (*degree q*) *p q*
unfolding *resultant-def sylvester-mat-def resultant-sub-def* **by** *auto*

lemma *resultant-const*[*simp*]:
fixes *a* :: *'a* :: *comm-ring-1*
shows *resultant* [:*a*:] *q* = *a* ^ (*degree q*)
and *resultant p* [:*a*:] = *a* ^ (*degree p*)
unfolding *resultant-def* **unfolding** *sylvester-mat-const* **by** *simp-all*

lemma *resultant-1*[*simp*]:
fixes *p* :: *'a* :: *comm-ring-1 poly*
shows *resultant 1 p* = *1 resultant p 1* = *1*
using *resultant-const*(*1*) [*of 1 p*] *resultant-const*(*2*) [*of p 1*]
by (*auto simp add*: *pCons-one*)

lemma *resultant-0*[*simp*]:
fixes *p* :: *'a* :: *comm-ring-1 poly*
assumes *degree p* > *0*
shows *resultant 0 p* = *0 resultant p 0* = *0*
using *resultant-const*(*1*)[*of 0 p*] *resultant-const*(*2*)[*of p 0*]
using *zero-power assms* **by** *auto*

lemma (**in** *comm-ring-hom*) *resultant-map-poly*: *degree* (*map-poly hom p*) = *degree p* ⟹
degree (*map-poly hom q*) = *degree q* ⟹ *resultant* (*map-poly hom p*) (*map-poly hom q*) = *hom* (*resultant p q*)
unfolding *resultant-def sylvester-mat-def sylvester-mat-sub-def hom-det*[*symmetric*]
by (*rule arg-cong*[*of - - det*], *auto*)

lemma (**in** *inj-comm-ring-hom*) *resultant-hom*: *resultant* (*map-poly hom p*) (*map-poly hom q*) = *hom* (*resultant p q*)
by (*rule resultant-map-poly*, *auto*)

end

3 Dichotomous Lazard

This theory contains Lazard's optimization in the computation of the subresultant PRS as described by Ducos [3, Section 2].

theory *Dichotomous-Lazard*
imports

HOL−Computational-Algebra.Polynomial-Factorial
begin

lemma *power-fract*[*simp*]: (*Fract a b*)^*n* = *Fract* (*a*^*n*) (*b*^*n*)
by (*induct n*, *auto simp*: *fract-collapse*)

lemma *range-to-fract-dvd-iff*: **assumes** *b*: *b* ≠ *0*
shows *Fract a b* ∈ *range to-fract* ⟷ *b dvd a*
proof
assume *b dvd a* **then obtain** *c* **where** *a*: *a* = *b* ∗ *c* **unfolding** *dvd-def* **by** *auto*
have *Fract a b* = *Fract c 1* **using** *b* **unfolding** *a* **by** (*simp add*: *eq-fract*)
thus *Fract a b* ∈ *range to-fract* **unfolding** *to-fract-def* **by** *auto*
next
assume *Fract a b* ∈ *range to-fract*
then obtain *c* **where** *Fract a b* = *Fract c 1* **unfolding** *to-fract-def* **by** *auto*
hence *a* = *b* ∗ *c* **using** *b* **by** (*simp add*: *eq-fract*)
thus *b dvd a* **..**
qed

lemma *Fract-cases-coprime* [*cases type*: *fract*]:
fixes *q* :: *'a* :: *factorial-ring-gcd fract*
obtains (*Fract*) *a b* **where** *q* = *Fract a b b* ≠ *0 coprime a b*
proof −
obtain *a b* **where** *q*: *q* = *Fract a b* **and** *b0*: *b* ≠ *0* **by** (*cases q*, *auto*)
define *g* **where** *g*: *g* = *gcd a b*
define *A* **where** *A*: *A* = *a div g*
define *B* **where** *B*: *B* = *b div g*
have *a*: *a* = *A* ∗ *g* **unfolding** *A g* **by** *simp*
have *b*: *b* = *B* ∗ *g* **unfolding** *B g* **by** *simp*
from *b0 b* **have** *0*: *B* ≠ *0* **by** *auto*
have *q*: *q* = *Fract A B* **unfolding** *q a b*
by (*subst eq-fract*, *auto simp*: *b0 0 g*)
have *cop*: *coprime A B* **unfolding** *A B g* **using** *b0*
by (*simp add*: *div-gcd-coprime*)
assume ⋀*a b*. *q* = *Fract a b* ⟹ *b* ≠ *0* ⟹ *coprime a b* ⟹ *thesis*
from *this*[*OF q 0 cop*] **show** *?thesis* **.**
qed

lemma *to-fract-power-le*: **fixes** *a* :: *'a* :: *factorial-ring-gcd fract*
assumes *no-fract*: *a* ∗ *b* ^ *e* ∈ *range to-fract*
and *a*: *a* ∈ *range to-fract*
and *le*: *f* ≤ *e*
shows *a* ∗ *b* ^ *f* ∈ *range to-fract*
proof −
obtain *bn bd* **where** *b*: *b* = *Fract bn bd* **and** *bd*: *bd* ≠ *0* **and** *copb*: *coprime bn bd* **by** (*cases b*, *auto*)
obtain *an* **where** *a*: *a* = *Fract an 1* **using** *a* **unfolding** *to-fract-def* **by** *auto*
have *id*: *a* ∗ *b* ^ *e* = *Fract* (*an* ∗ *bn*^*e*) (*bd*^*e*) **unfolding** *a b power-fract mult-fract* **by** *simp*

```
  have 0: bd^e ≠ 0 for e using bd by auto
  from no-fract[unfolded id range-to-fract-dvd-iff[OF 0]] have dvd: bd ^ e dvd an
* bn ^ e .
  from copb have copb: coprime (bd ^ e) (bn ^ e) for e
    by (simp add: ac-simps)
  from dvd copb [of e] bd have bd ^ e dvd an
    by (simp add: coprime-dvd-mult-left-iff)
  hence bd ^ f dvd an using le by (rule power-le-dvd)
  hence dvd: bd ^ f dvd an * bn ^ f by simp
  from le obtain g where e: e = f + g using le-Suc-ex by blast
   have id': a * b ^ f = Fract (an * bn^f) (bd^f) unfolding a b power-fract
mult-fract by simp
  show ?thesis unfolding id' range-to-fract-dvd-iff[OF 0] by (rule dvd)
qed

lemma div-divide-to-fract: assumes x ∈ range to-fract
  and x = (y :: 'a :: idom-divide fract) / z
  and x' = y' div z'
  and y = to-fract y' z = to-fract z'
  shows x = to-fract x'
proof (cases z' = 0)
  case True
  thus ?thesis using assms by auto
next
  case False
  from assms obtain r where to-fract y' / to-fract z' = to-fract r by auto
  thus ?thesis using False assms
    by (simp add: eq-fract(1) to-fract-def)
qed

declare Euclidean-Division.divmod-nat-def [termination-simp]

fun dichotomous-Lazard :: 'a :: idom-divide ⇒ 'a ⇒ nat ⇒ 'a where
  dichotomous-Lazard x y n = (if n ≤ 1 then if n = 1 then x else 1 else
    let (d,r) = Euclidean-Division.divmod-nat n 2;
      rec = dichotomous-Lazard x y d;
      recsq = rec * rec div y in
    if r = 0 then recsq else recsq * x div y)

lemma dichotomous-Lazard-main: fixes x :: 'a :: idom-divide
  assumes ⋀ i. i ≤ n ⟹ (to-fract x)^i / (to-fract y)^(i - 1) ∈ range to-fract
  shows to-fract (dichotomous-Lazard x y n) = (to-fract x)^n / (to-fract y)^(n-1)

  using assms
proof (induct x y n rule: dichotomous-Lazard.induct)
  case (1 x y n)
  let ?f = to-fract
  consider (0) n = 0 | (1) n = 1 | (n) ¬ n ≤ 1 by linarith
  thus ?case
```

```
  proof cases
    case n
    obtain d r where n2: Euclidean-Division.divmod-nat n 2 = (d,r) by force
    from Euclidean-Division.divmod-nat-def[of n 2] n2 have dr: d = n div 2 r =
n mod 2 by auto
    hence r: r = 0 ∨ r = 1 by auto
    define rec where rec = dichotomous-Lazard x y d
    let ?sq = rec * rec div y
    have res: dichotomous-Lazard x y n = (if r = 0 then ?sq else ?sq * x div y)
      unfolding dichotomous-Lazard.simps[of x y n] n2 Let-def rec-def using n by
auto
    have ndr: n = d + d + r unfolding dr by presburger
    from ndr r n have d0: d ≠ 0 by auto
    have IH: ?f rec = ?f x ^ d / ?f y ^ (d - 1)
      using 1(1)[OF n refl n2[symmetric] 1(2), folded rec-def] ndr by auto
    have ?f (rec * rec) = ?f x ^ d / ?f y ^ (d - 1) * ?f x ^ d / ?f y ^ (d - 1)
using IH by simp
   also have ... = ?f x ^ (d + d) / ?f y ^ (d - 1 + (d - 1)) unfolding power-add
by simp
    also have d - 1 + (d - 1) = d + d - 2 using d0 by simp
    finally have id: ?f (rec * rec) = ?f x ^ (d + d) / ?f y ^ (d + d - 2) .
    let ?dd = (?f x ^ (d + d) / ?f y ^ (d + d - 2)) / ?f y
    let ?d = ?f x ^ (d + d) / ?f y ^ (d + d - 1)
    have dd: ?dd = ?d using d0 by (cases d, auto)
    have sq: ?f ?sq = ?d unfolding dd[symmetric]
   proof (rule sym, rule div-divide-to-fract[OF - refl refl id[symmetric] refl], unfold
dd)
      show ?d ∈ range ?f by (rule 1(2), insert ndr, auto)
    qed
    show ?thesis
    proof (cases r = 0)
      case True
      with res sq show ?thesis unfolding ndr by auto
    next
      case False
      with r have r: r = 1 by auto
      let ?sq' = ?sq * x div y
      from False res have res: dichotomous-Lazard x y n = ?sq' by simp
      from sq have id: ?f (?sq * x) = ?f x ^ (d + d + r) / ?f y ^ (d + d - 1)
        unfolding r by simp
      let ?dd = (?f x ^ (d + d + r) / ?f y ^ (d + d - 1)) / ?f y
      let ?d = ?f x ^ (d + d + r) / ?f y ^ (d + d + r - 1)
      have dd: ?dd = ?d using d0 unfolding r by (cases d, auto)
      have sq': ?f ?sq' = ?d unfolding dd[symmetric]
        proof (rule sym, rule div-divide-to-fract[OF - refl refl id[symmetric] refl],
unfold dd)
        show ?d ∈ range ?f by (rule 1(2), unfold ndr, auto)
      qed
      show ?thesis unfolding res sq' unfolding ndr by simp
```

```
    qed
  qed auto
qed

lemma dichotomous-Lazard: fixes x :: 'a :: factorial-ring-gcd
  assumes (to-fract x)^n / (to-fract y)^(n−1) ∈ range to-fract
  shows to-fract (dichotomous-Lazard x y n) = (to-fract x)^n / (to-fract y)^(n−1)

proof (rule dichotomous-Lazard-main)
  fix i
  assume i: i ≤ n
  show to-fract x ^ i / to-fract y ^ (i − 1) ∈ range to-fract
  proof (cases i)
    case (Suc j)
    have id: to-fract x ^ i / to-fract y ^ (i − 1) = to-fract x * (to-fract x / to-fract
y) ^ j
      unfolding Suc by (simp add: power-divide)
    from Suc i have n ≠ 0 and j: j ≤ n − 1 by auto
    hence idd: to-fract x * (to-fract x / to-fract y) ^ (n − 1) = (to-fract x)^n /
(to-fract y)^(n−1)
      by (cases n, auto simp: power-divide)
    show ?thesis unfolding id
      by (rule to-fract-power-le[OF - - j], unfold idd, insert assms, auto)
  next
    case 0
    have 1 = to-fract 1 by simp
    hence 1 ∈ range to-fract by blast
    thus ?thesis using 0 by auto
  qed
qed

declare dichotomous-Lazard.simps[simp del]

end
```

4 Binary Exponentiation

This theory defines the standard algorithm for binary exponentiation, or exponentiation by squaring.

```
theory Binary-Exponentiation
imports
  Main
begin

declare Euclidean-Division.divmod-nat-def[termination-simp]

context monoid-mult
begin
```

```
fun binary-power :: 'a ⇒ nat ⇒ 'a where
  binary-power x n = (if n = 0 then 1 else
    let (d,r) = Euclidean-Division.divmod-nat n 2;
       rec = binary-power (x * x) d in
    if r = 0 then rec else rec * x)

lemma binary-power[simp]: binary-power = (^)
proof (intro ext)
  fix x n
  show binary-power x n = x ^ n
  proof (induct x n rule: binary-power.induct)
    case (1 x n)
    show ?case
    proof (cases n = 0)
      case False
      note IH = 1[OF False]
      obtain d r where n2: Euclidean-Division.divmod-nat n 2 = (d,r) by force
       from Euclidean-Division.divmod-nat-def[of n 2] n2 have dr: d = n div 2 r
= n mod 2 by auto
      hence r: r = 0 ∨ r = 1 by auto
      let ?rec = binary-power (x * x) d
      have binary-power x n = (if r = 0 then ?rec else ?rec * x)
        unfolding binary-power.simps[of x n] n2 using False by auto
      also have ... = ?rec * x ^ r using r by (cases r = 0, auto)
      also have ?rec = (x * x) ^ d
        by (rule IH[OF - refl], simp add: n2)
      also have ... = x ^ (d + d) unfolding power-add
        using power2-eq-square power-even-eq power-mult by auto
      also have ... * x ^ r = x ^ (d + d + r)
        by (simp add: power-add)
      also have d + d + r = n unfolding dr by presburger
      finally show ?thesis .
    qed auto
  qed
qed

lemma binary-power-code-unfold[code-unfold]: (^) = binary-power
  by simp

declare binary-power.simps[simp del]
end
end
```

5 Homomorphisms

We register two homomorphism, namely lifting constants to polynomials, and lifting elements of some domain into their fraction field.

```
theory More-Homomorphisms
```

imports *Polynomial-Interpolation.Ring-Hom-Poly*
Jordan-Normal-Form.Determinant
begin

abbreviation (*input*) *coeff-lift* == $\lambda a.$ [: *a* :]

interpretation *coeff-lift-hom*: *inj-comm-monoid-add-hom coeff-lift* **by** (*unfold-locales*, *auto*)
interpretation *coeff-lift-hom*: *inj-ab-group-add-hom coeff-lift*..
interpretation *coeff-lift-hom*: *inj-comm-semiring-hom coeff-lift*
by *standard* (*simp-all add*: *ac-simps*)
interpretation *coeff-lift-hom*: *inj-comm-ring-hom coeff-lift*..
interpretation *coeff-lift-hom*: *inj-idom-hom coeff-lift*..

The following rule is incompatible with existing simp rules.

declare *coeff-lift-hom.hom-mult*[*simp del*]
declare *coeff-lift-hom.hom-add*[*simp del*]
declare *coeff-lift-hom.hom-uminus*[*simp del*]

interpretation *to-fract-hom*: *inj-comm-ring-hom to-fract* **by** (*unfold-locales*, *auto*)
interpretation *to-fract-hom*: *idom-hom to-fract*..
interpretation *to-fract-hom*: *inj-idom-hom to-fract*..

end

6 Polynomial coefficients with integer index

We provide a function to access the coefficients of a polynomial via an integer index. Then index-shifting becomes more convenient, e.g., compare in the lemmas for accessing the coeffiencent of a product with a monomial there is no special case for integer coefficients, whereas for natural number coefficients there is a case-distinction.

theory *Coeff-Int*
imports
HOL−Combinatorics.Permutations
Polynomial-Interpolation.Missing-Polynomial
begin

definition *coeff-int* :: *'a* :: *zero poly* ⇒ *int* ⇒ *'a* **where**
coeff-int p i = (*if i* < *0 then 0 else coeff p* (*nat i*))

lemma *coeff-int-eq-0*: *i* < *0* ∨ *i* > *int* (*degree p*) ⟹ *coeff-int p i* = *0*
unfolding *coeff-int-def*
by (*cases i* < *0*, *auto intro*: *coeff-eq-0*)

lemma *coeff-int-smult*[*simp*]: *coeff-int* (*smult c p*) *i* = *c* ∗ *coeff-int p i*
unfolding *coeff-int-def* **by** *simp*

lemma *coeff-int-signof-mult*: *coeff-int* (*of-int* (*sign x*) ∗ *f*) *i* = *of-int* (*sign x*) ∗ *coeff-int f i*
by (*auto simp*: *coeff-int-def sign-def*)

lemma *coeff-int-sum*: *coeff-int* (*sum p A*) *i* = (∑ *x*∈*A*. *coeff-int* (*p x*) *i*)
using *coeff-sum*[*of p A nat i*] **unfolding** *coeff-int-def*
by (*cases i* < *0*, *auto*)

lemma *coeff-int-0*[*simp*]: *coeff-int f 0* = *coeff f 0* **unfolding** *coeff-int-def* **by** *simp*

lemma *coeff-int-monom-mult*: *coeff-int* (*monom a d* ∗ *f*) *i* = (*a* ∗ *coeff-int f* (*i* − *d*))
proof (*cases i* < *0*)
case *True*
thus *?thesis* **unfolding** *coeff-int-def* **by** *simp*
next
case *False*
hence *i* ≥ *0* **by** *auto*
then obtain *j* **where** *i*: *i* = *int j* **by** (*rule nonneg-eq-int*)
show *?thesis*
proof (*cases i* ≥ *d*)
case *True*
with *i* **have** *nat* (*int j* − *int d*) = *j* − *d* **by** *auto*
with *coeff-monom-mult*[*of a*] **show** *?thesis* **unfolding** *coeff-int-def i*
by *simp*
next
case *False*
thus *?thesis* **unfolding** *i* **by** (*simp add*: *coeff-int-def coeff-monom-mult*)
qed
qed

lemma *coeff-prod-const*: **assumes** *finite xs* **and** *y* ∉ *xs*
and ⋀ *x*. *x* ∈ *xs* ⟹ *degree* (*f x*) = *0*
shows *coeff* (*prod f* (*insert y xs*)) *i* = *prod* (λ *x*. *coeff* (*f x*) *0*) *xs* ∗ *coeff* (*f y*) *i*
using *assms*
proof (*induct xs rule*: *finite-induct*)
case (*insert x xs*)
from *insert*(*2*,*4*) **have** *id*: *insert y* (*insert x xs*) − {*x*} = *insert y xs* **by** *auto*
have *prod f* (*insert y* (*insert x xs*)) = *f x* ∗ *prod f* (*insert y xs*)
by (*subst prod.remove*[*of - x*], *insert insert*(*1*,*2*) *id*, *auto*)
hence *coeff* (*prod f* (*insert y* (*insert x xs*))) *i* = *coeff* (*f x* ∗ *prod f* (*insert y xs*)) *i* **by** *simp*
also have ... = *coeff* (*f x*) *0* ∗ (*coeff* (*prod f* (*insert y xs*)) *i*)
proof −
from *insert*(*5*)[*of x*] *degree0-coeffs*[*of f x*] **obtain** *c* **where** *fx*: *f x* = [: *c* :] **by** *auto*
show *?thesis* **unfolding** *fx* **by** *auto*
qed
also have (*coeff* (*prod f* (*insert y xs*)) *i*) = (∏ *x*∈*xs*. *coeff* (*f x*) *0*) ∗ *coeff* (*f y*)

i **using** *insert* **by** *auto*
also have *coeff* (*f x*) *0* ∗ ... = *prod* (λ *x*. *coeff* (*f x*) *0*) (*insert x xs*) ∗ *coeff* (*f y*) *i*
by (*subst prod.insert-remove*, *insert insert*(*1*,*2*,*4*), *auto simp*: *ac-simps*)
finally show *?case* **.**
qed *simp*

lemma *coeff-int-prod-const*: **assumes** *finite xs* **and** *y* ∉ *xs*
and ⋀ *x*. *x* ∈ *xs* ⟹ *degree* (*f x*) = *0*
shows *coeff-int* (*prod f* (*insert y xs*)) *i* = *prod* (λ *x*. *coeff-int* (*f x*) *0*) *xs* ∗ *coeff-int* (*f y*) *i*
using *coeff-prod-const*[*OF assms*] **unfolding** *coeff-int-def* **by** (*cases i* < *0*, *auto*)

lemma *coeff-int*[*simp*]: *coeff-int p n* = *coeff p n* **unfolding** *coeff-int-def* **by** *auto*

lemma *coeff-int-minus*[*simp*]:
coeff-int (*a* − *b*) *i* = *coeff-int a i* − *coeff-int b i*
by (*auto simp*: *coeff-int-def*)

lemma *coeff-int-pCons-0*[*simp*]: *coeff-int* (*pCons 0 b*) *i* = *coeff-int b* (*i* − *1*)
by (*auto simp*: *Nitpick.case-nat-unfold coeff-int-def coeff-pCons nat-diff-distrib'*)

end

7 Subresultants and the subresultant PRS

This theory contains most of the soundness proofs of the subresultant PRS algorithm, where we closely follow the papers of Brown [1] and Brown and Traub [2]. This is in contrast to a similar Coq formalization of Mahboubi [4] which is based on polynomial determinants.

Whereas the current file only contains an algorithm to compute the resultant of two polynomials efficiently, there is another theory "Subresultant-Gcd" which also contains the algorithm to compute the GCD of two polynomials via the subresultant algorithm. In both algorithms we integrate Lazard's optimization in the dichotomous version, but not the second optmization described by Ducos [3].

theory *Subresultant*
imports
Resultant-Prelim
Dichotomous-Lazard
Binary-Exponentiation
More-Homomorphisms
Coeff-Int
begin

7.1 Algorithm

locale *div-exp-param* =
  fixes *div-exp* :: *'a* :: *idom-divide* ⇒ *'a* ⇒ *nat* ⇒ *'a*
begin
partial-function(*tailrec*) *subresultant-prs-main* **where**
  subresultant-prs-main f g c = (*let*
   m = *degree f*;
   n = *degree g*;
   lf = *lead-coeff f*;
   lg = *lead-coeff g*;
   δ = *m* − *n*;
   d = *div-exp lg c δ*;
   h = *pseudo-mod f g*
  in if h = *0 then* (*g*,*d*)
     else subresultant-prs-main g (*sdiv-poly h* ((−*1*) ^(*δ* + *1*) ∗ *lf* ∗ (*c* ^*δ*))) *d*)

definition *subresultant-prs* **where**
  subresultant-prs f g = (*let*
    h = *pseudo-mod f g*;
    δ = (*degree f* − *degree g*);
    d = *lead-coeff g* ^*δ*
    in if h = *0 then* (*g*,*d*)
       else subresultant-prs-main g ((− *1*) ^(*δ* + *1*) ∗ *h*) *d*)

definition *resultant-impl-main* **where**
  resultant-impl-main G1 G2 = (*if G2* = *0 then* (*if degree G1* = *0 then 1 else 0*)
else
     case subresultant-prs G1 G2 of
     (*Gk*,*hk*) ⇒ (*if degree Gk* = *0 then hk else 0*))

definition *resultant-impl* **where**
  resultant-impl f g =
     (*if length* (*coeffs f*) ≥ *length* (*coeffs g*) *then resultant-impl-main f g*
     else let res = *resultant-impl-main g f in*
      if even (*degree f*) ∨ *even* (*degree g*) *then res else* − *res*)
end

locale *div-exp-sound* = *div-exp-param* +
  assumes *div-exp*: ⋀ *x y n*.
     (*to-fract x*)^*n* / (*to-fract y*)^(*n*−*1*) ∈ *range to-fract*
     ⟹ *to-fract* (*div-exp x y n*) = (*to-fract x*)^*n* / (*to-fract y*)^(*n*−*1*)

definition *basic-div-exp* :: *'a* :: *idom-divide* ⇒ *'a* ⇒ *nat* ⇒ *'a* **where**
  basic-div-exp x y n = *x*^*n div y*^(*n*−*1*)

We have an instance for arbitrary integral domains.

lemma *basic-div-exp*: *div-exp-sound basic-div-exp*
 by (*unfold-locales*, *unfold basic-div-exp-def*, *rule sym*, *rule div-divide-to-fract*, *auto simp*: *hom-distribs*)

Lazard's optimization is only proven for factorial rings.

lemma *dichotomous-Lazard*: *div-exp-sound* (*dichotomous-Lazard* :: *'a* :: *factorial-ring-gcd* ⇒ -)
by (*unfold-locales*, *rule dichotomous-Lazard*)

7.2 Soundness Proof for *div-exp-param.resultant-impl div-exp = resultant*

abbreviation *pdivmod* :: *'a*::*field poly* ⇒ *'a poly* ⇒ *'a poly* × *'a poly*
where
pdivmod p q ≡ (*p div q*, *p mod q*)

lemma *even-sum-list*: **assumes** ⋀ *x*. *x* ∈ *set xs* ⟹ *even* (*f x*) = *even* (*g x*)
shows *even* (*sum-list* (*map f xs*)) = *even* (*sum-list* (*map g xs*))
using *assms* **by** (*induct xs*, *auto*)

lemma *for-all-Suc*: *P i* ⟹ (∀ *j* ≥ *Suc i*. *P j*) = (∀ *j* ≥ *i*. *P j*) **for** *P*
by (*metis* (*full-types*) *Suc-le-eq less-le*)

lemma *pseudo-mod-left-0*[*simp*]: *pseudo-mod 0 x* = *0*
unfolding *pseudo-mod-def pseudo-divmod-def*
by (*cases x* = *0*; *cases length* (*coeffs x*), *auto*)

lemma *pseudo-mod-right-0*[*simp*]: *pseudo-mod x 0* = *x*
unfolding *pseudo-mod-def pseudo-divmod-def* **by** *simp*

lemma *snd-pseudo-divmod-main-cong*:
assumes *a1* = *b1 a3* = *b3 a4* = *b4 a5* = *b5 a6* = *b6*
shows *snd* (*pseudo-divmod-main a1 a2 a3 a4 a5 a6*) = *snd* (*pseudo-divmod-main b1 b2 b3 b4 b5 b6*)
using *assms snd-pseudo-divmod-main* **by** *metis*

lemma *snd-pseudo-mod-smult-invar-right*:
shows (*snd* (*pseudo-divmod-main* (*x* ∗ *lc*) *q r* (*smult x d*) *dr n*))
= *snd* (*pseudo-divmod-main lc q'* (*smult* (*x*^*n*) *r*) *d dr n*)
proof(*induct n arbitrary*: *q q' r dr*)
case (*Suc n*)
let *?q* = *smult* (*x* ∗ *lc*) *q* + *monom* (*coeff r dr*) *n*
let *?r* = *smult* (*x* ∗ *lc*) *r* − (*smult x* (*monom* (*coeff r dr*) *n* ∗ *d*))
let *?dr* = *dr* − *1*
let *?rec-lhs* = *pseudo-divmod-main* (*x* ∗ *lc*) *?q ?r* (*smult x d*) *?dr n*
let *?rec-rhs* = *pseudo-divmod-main lc q'* (*smult* (*x*^*n*) *?r*) *d ?dr n*
have [*simp*]: ⋀ *n*. *x* ^ *n* ∗ (*x* ∗ *lc*) = *lc* ∗ (*x* ∗ *x* ^ *n*)
⋀ *n c*. *x* ^ *n* ∗ (*x* ∗ *c*) = *x* ∗ *x* ^ *n* ∗ *c*
⋀ *n*. *x* ∗ *x* ^ *n* ∗ *lc* = *lc* ∗ (*x* ∗ *x* ^ *n*)
by (*auto simp*: *ac-simps*)
have *snd* (*pseudo-divmod-main* (*x* ∗ *lc*) *q r* (*smult x d*) *dr* (*Suc n*)) = *snd ?rec-lhs*
by (*auto simp*:*Let-def*)

also have ... = *snd ?rec-rhs* **using** *Suc* **by** *auto*
also have ... = *snd* (*pseudo-divmod-main lc q'* (*smult* (*x*^*Suc n*) *r*) *d dr* (*Suc n*))
unfolding *pseudo-divmod-main.simps Let-def*
proof(*rule snd-pseudo-divmod-main-cong*,*goal-cases*)
case *2*
show *?case* **by** (*auto simp*:*smult-add-right smult-diff-right smult-monom smult-monom-mult*)
qed *auto*
finally show *?case* **by** *auto*
qed *auto*

lemma *snd-pseudo-mod-smult-invar-left*:
shows *snd* (*pseudo-divmod-main lc q* (*smult x r*) *d dr n*)
= *smult x* (*snd* (*pseudo-divmod-main lc q' r d dr n*))
proof(*induct n arbitrary*:*x lc q q' r d dr*)
case (*Suc n*)
have *sm*:*smult lc* (*smult x r*) − *monom* (*coeff* (*smult x r*) *dr*) *n* ∗ *d*
=*smult x* (*smult lc r* − *monom* (*coeff r dr*) *n* ∗ *d*)
by (*auto simp*:*smult-diff-right smult-monom smult-monom-mult mult.commute*[*of lc x*])
let *?q'* = *smult lc q'* + *monom* (*coeff r dr*) *n*
show *?case* **unfolding** *pseudo-divmod-main.simps Let-def Suc*(*1*)[*of lc - - - - - ?q'*] *sm* **by** *auto*
qed *auto*

lemma *snd-pseudo-mod-smult-left*[*simp*]:
shows *snd* (*pseudo-divmod* (*smult* (*x*::'*a*::*idom*) *p*) *q*) = (*smult x* (*snd* (*pseudo-divmod p q*)))
unfolding *pseudo-divmod-def*
by (*auto simp*:*snd-pseudo-mod-smult-invar-left*[*of - - - - - - - - 0*] *Polynomial.coeffs-smult*)

lemma *pseudo-mod-smult-right*:
assumes (*x*::'*a*::*idom*)≠*0 q*≠*0*
shows (*pseudo-mod p* (*smult* (*x*::'*a*::*idom*) *q*)) = (*smult* (*x*^(*Suc* (*length* (*coeffs p*)) − *length* (*coeffs q*))) (*pseudo-mod p q*))
unfolding *pseudo-divmod-def pseudo-mod-def*
by (*auto simp*:*snd-pseudo-mod-smult-invar-right*[*of - - - - - - - 0*]
snd-pseudo-mod-smult-invar-left[*of - - - - - - - 0*] *Polynomial.coeffs-smult assms*)

lemma *pseudo-mod-zero*[*simp*]:
pseudo-mod 0 f = (*0*::'*a* :: {*idom*} *poly*)
pseudo-mod f 0 = *f*
unfolding *pseudo-mod-def snd-pseudo-mod-smult-left*[*of 0 - f*,*simplified*]
unfolding *pseudo-divmod-def* **by** *auto*

lemma *prod-combine*:
assumes $j \leq i$
shows $f\ i * (\prod l{\leftarrow}[j..{<}i].\ (f\ l :: 'a{::}comm\text{-}monoid\text{-}mult)) = prod\text{-}list\ (map\ f\ [j..{<}Suc\ i])$
proof(*subst prod-list-map-remove1*[*of i* [*j..<Suc i*] *f*],*goal-cases*)
case *2*
have *remove1 i* ([*j..<i*] @ [*i*]) = [*j..<i*] **by** (*simp add*: *remove1-append*)
thus *?case* **by** *auto*
qed (*insert assms*, *auto*)

lemma *prod-list-minus-1-exp*: *prod-list* (*map* (λ *i*. (−*1*)^(*f i*)) *xs*)
= (−*1*)^(*sum-list* (*map f xs*))
by (*induct xs*, *auto simp*: *power-add*)

lemma *minus-1-power-even*: (− (*1* :: *'b* :: *comm-ring-1*))^ *k* = (*if even k then 1 else* (−*1*))
by *auto*

lemma *minus-1-even-eqI*: **assumes** *even k* = *even l* **shows**
(− (*1* :: *'b* :: *comm-ring-1*))^*k* = (− *1*)^*l*
unfolding *minus-1-power-even assms* **by** *auto*

lemma (**in** *comm-monoid-mult*) *prod-list-multf*:
$(\prod x{\leftarrow}xs.\ f\ x * g\ x) = prod\text{-}list\ (map\ f\ xs) * prod\text{-}list\ (map\ g\ xs)$
by (*induct xs*) (*simp-all add*: *algebra-simps*)

lemma *inverse-prod-list*: *inverse* (*prod-list xs*) = *prod-list* (*map inverse* (*xs* :: *'a* :: *field list*))
by (*induct xs*, *auto*)

definition *pow-int* :: *'a* :: *field* ⇒ *int* ⇒ *'a* **where**
pow-int x e = (*if e < 0 then 1 / (x* ^ (*nat* (−*e*))) *else x* ^ (*nat e*))

lemma *pow-int-0*[*simp*]: *pow-int x 0* = *1* **unfolding** *pow-int-def* **by** *auto*

lemma *pow-int-1*[*simp*]: *pow-int x 1* = *x* **unfolding** *pow-int-def* **by** *auto*

lemma *exp-pow-int*: *x* ^ *n* = *pow-int x n*
unfolding *pow-int-def* **by** *auto*

lemma *pow-int-add*: **assumes** *x*: *x* ≠ *0* **shows** *pow-int x* (*a* + *b*) = *pow-int x a* ∗ *pow-int x b*
proof −
have ∗:
¬ *a* + *b* < *0* ⟹ *a* < *0* ⟹ *nat b* = *nat* (*a* + *b*) + *nat* (−*a*)
¬ *a* + *b* < *0* ⟹ *b* < *0* ⟹ *nat a* = *nat* (*a* + *b*) + *nat* (−*b*)
a + *b* < *0* ⟹ ¬ *a* < *0* ⟹ *nat* (−*b*) = *nat a* + *nat* (−*a* −*b*)

$a + b < 0 \Longrightarrow \neg\ b < 0 \Longrightarrow nat\ (-a) = nat\ b + nat\ (-a - b)$
by *auto*
have *pow-eq*: $l = m \Longrightarrow (x \hat{}\ l = x \hat{}\ m)$ **for** *l m* **by** *auto*
from *x* **show** *?thesis* **unfolding** *pow-int-def*
by (*auto split*: *if-splits simp*: *power-add*[*symmetric*] *simp*: $*$ *intro*!: *pow-eq*, *auto simp*: *power-add*)
qed

lemma *pow-int-mult*: *pow-int* $(x * y)$ *a* = *pow-int x a* $*$ *pow-int y a*
unfolding *pow-int-def* **by** (*cases a* < 0, *auto simp*: *power-mult-distrib*)

lemma *pow-int-base-1*[*simp*]: *pow-int 1 a* = *1*
unfolding *pow-int-def* **by** (*cases a* < 0, *auto*)

lemma *pow-int-divide*: *a* / *pow-int x b* = *a* $*$ *pow-int x* $(-b)$
unfolding *pow-int-def* **by** (*cases b rule*: *linorder-cases*[*of - 0*], *auto*)

lemma *divide-prod-assoc*: *x* / $(y * z ::$ *'a* :: *field*$)$ = *x* / *y* / *z* **by** (*simp add*: *field-simps*)

lemma *minus-1-inverse-pow*[*simp*]: *x* / $(-1)\hat{}n = (x ::$ *'a* :: *field*$) * (-1)\hat{}n$
by (*simp add*: *minus-1-power-even*)

definition *subresultant-mat* :: *nat* $\Rightarrow$ *'a* :: *comm-ring-1 poly* $\Rightarrow$ *'a poly* $\Rightarrow$ *'a poly mat* **where**
subresultant-mat J F G = (*let*
dg = *degree G*; *df* = *degree F*; *f* = *coeff-int F*; *g* = *coeff-int G*; *n* = (*df* $-$ *J*) + (*dg* $-$ *J*)
in mat n n (λ (*i*,*j*). *if j* $<$ *dg* $-$ *J then*
if i = *n* $-$ *1 then monom 1* (*dg* $-$ *J* $-$ *1* $-$ *j*) $*$ *F else* [: *f* (*df* $-$ *int i* + *int j*) :]
else let jj = *j* $-$ (*dg* $-$ *J*) *in*
if i = *n* $-$ *1 then monom 1* (*df* $-$ *J* $-$ *1* $-$ *jj*) $*$ *G else* [: *g* (*dg* $-$ *int i* + *int jj*) :]))

lemma *subresultant-mat-dim*[*simp*]:
fixes *j p q*
defines *S* $\equiv$ *subresultant-mat j p q*
shows *dim-row S* = (*degree p* $-$ *j*) + (*degree q* $-$ *j*) **and** *dim-col S* = (*degree p* $-$ *j*) + (*degree q* $-$ *j*)
unfolding *S-def subresultant-mat-def Let-def* **by** *auto*

definition *subresultant'-mat* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'a* :: *comm-ring-1 poly* $\Rightarrow$ *'a poly* $\Rightarrow$ *'a mat* **where**
subresultant'-mat J l F G = (*let*
γ = *degree G*; φ = *degree F*; *f* = *coeff-int F*; *g* = *coeff-int G*; *n* = (φ $-$ *J*) + (γ $-$ *J*)
in mat n n (λ (*i*,*j*). *if j* $<$ γ $-$ *J then*

```
      if i = n − 1 then (f (l − int (γ − J − 1) + int j)) else (f (φ − int i + int
j))
      else let jj = j − (γ − J) in
      if i = n − 1 then (g (l − int (φ − J − 1) + int jj)) else (g (γ − int i + int
jj))))

lemma subresultant-index-mat:
  fixes F G
  assumes i: i < (degree F − J) + (degree G − J) and j: j < (degree F − J) +
(degree G − J)
  shows subresultant-mat J F G $$ (i,j) =
    (if j < degree G − J then
       if i = (degree F − J) + (degree G − J) − 1 then monom 1 (degree G − J
− 1 − j) ∗ F else ([: coeff-int F ( degree F − int i + int j) :])
     else let jj = j − (degree G − J) in
      if i = (degree F − J) + (degree G − J) − 1 then monom 1 ( degree F − J
− 1 − jj) ∗ G else ([: coeff-int G (degree G − int i + int jj) :]))
  unfolding subresultant-mat-def Let-def
  unfolding index-mat(1)[OF i j] split by auto

definition subresultant :: nat ⇒ 'a :: comm-ring-1 poly ⇒ 'a poly ⇒ 'a poly where
  subresultant J F G = det (subresultant-mat J F G)


lemma subresultant-smult-left: assumes (c :: 'a :: {comm-ring-1, semiring-no-zero-divisors})
≠ 0
  shows subresultant J (smult c f) g = smult (c ^(degree g − J)) (subresultant J
f g)
proof −
  let ?df = degree f
  let ?dg = degree g
  let ?n = (?df − J) + (?dg − J)
  let ?m = ?dg − J
  let ?M = mat ?n ?n (λ (i,j). if i = j then if i < ?m then [:c:] else 1 else 0)
  from ‹c ≠ 0› have deg: degree (smult c f) = ?df by simp
  let ?S = subresultant-mat J f g
  let ?cS = subresultant-mat J (smult c f) g
  have dim: dim-row ?S = ?n dim-col ?S = ?n  dim-row ?cS = ?n dim-col ?cS =
?n using deg by auto
  hence C: ?S ∈ carrier-mat ?n ?n ?cS ∈ carrier-mat ?n ?n ?M ∈ carrier-mat
?n ?n by auto
  have dim': dim-row (?S ∗ ?M) = ?n dim-col (?S ∗ ?M) = ?n using dim (1,2)
by simp-all
  define S where S = ?S
  have ?cS = ?S ∗ ?M
  proof (rule eq-matI, unfold dim' dim)
    fix i j
    assume ij: i < ?n j < ?n
    have (?S ∗ ?M) $$ (i,j) = row ?S i · col ?M j
```

```
    by (rule index-mult-mat, insert ij dim, auto)
   also have ... = (∑ k = 0..<?n. row S i $ k * col ?M j $ k) unfolding
scalar-prod-def S-def[symmetric]
    by simp
  also have ... = (∑ k = 0..<?n. S $$ (i,k) * ?M $$ (k,j))
    by (rule sum.cong, insert ij, auto simp: S-def)
  also have ... = S $$ (i,j) * ?M $$ (j,j) + sum (λ k. S $$ (i,k) * ?M $$ (k,j))
({0..<?n} - {j})
    by (rule sum.remove, insert ij, auto)
  also have ... = S $$ (i,j) * ?M $$ (j,j)
    by (subst sum.neutral, insert ij, auto)
  also have ... = ?cS $$ (i,j) unfolding subresultant-index-mat[OF ij] S-def
    by (subst subresultant-index-mat, unfold deg, insert ij, auto)
  finally show ?cS $$ (i,j) = (?S * ?M) $$ (i,j) by simp
 qed auto
 from arg-cong[OF this, of det] det-mult[OF C(1) C(3)]
 have subresultant J (smult c f) g = subresultant J f g * det ?M
   unfolding subresultant-def by auto
 also have det ?M = [:c ^ ?m :]
 proof (subst det-upper-triangular[OF - C(3)])
   show upper-triangular ?M
     by (rule upper-triangularI, auto)
   have prod-list (diag-mat ?M) = (∏ k = 0..<?n. (?M $$ (k,k)))
     unfolding prod-list-diag-prod by simp
   also have ... = (∏ k = 0..<?m. ?M $$ (k,k)) * (∏ k = ?m..<?n. ?M $$
(k,k))
     by (subst prod.union-disjoint[symmetric], (auto)[3], rule prod.cong, auto)
   also have (∏ k = 0..<?m. ?M $$ (k,k)) = (∏ k = 0..<?m. [: c :])
     by (rule prod.cong, auto)
   also have (∏ k = 0..<?m. [: c :]) = [: c :] ^ ?m by simp
   also have (∏ k = ?m..<?n. ?M $$ (k,k)) = (∏ k = ?m..<?n. 1)
     by (rule prod.cong, auto)
   finally show prod-list (diag-mat ?M) = [: c^?m :] unfolding poly-const-pow
by simp
 qed
 finally show ?thesis by simp
qed

lemma subresultant-swap:
 shows subresultant J f g = smult ((- 1) ^ ((degree f - J) * (degree g - J)))
(subresultant J g f)
proof -
 let ?A = subresultant-mat J f g
 let ?k = degree f - J
 let ?n = degree g - J
 have nk: ?n + ?k = ?k + ?n by simp
 have change: j < ?k + ?n ⟹ ((if j < ?k then j + ?n else j - ?k) < ?n)
   = (¬ (j < ?k)) for j by auto
 have subresultant J f g = det ?A unfolding subresultant-def by simp
```

```
  also have ... = (−1)^(?k * ?n) * det (mat (?k + ?n) (?k + ?n) (λ (i,j).
    ?A $$ (i,(if j < ?k then j + ?n else j − ?k)))) (is - = - * det ?B)
    by (rule det-swap-cols, auto simp: subresultant-mat-def Let-def)
  also have ?B = subresultant-mat J g f
    unfolding subresultant-mat-def Let-def
    by (rule eq-matI, unfold dim-row-mat dim-col-mat nk index-mat split,
      subst index-mat, (auto)[2], unfold split, subst change, force,
      unfold if-conn, rule if-cong[OF refl if-cong if-cong], auto)
  also have det ... = subresultant J g f unfolding subresultant-def ..
   also have (−1)^(?k * ?n) * ... = [: (−1)^(?k * ?n) :] * ...  by (unfold
hom-distribs, simp)
  also have ... = smult ((−1)^(?k * ?n)) (subresultant J g f) by simp
  finally show ?thesis .
qed

lemma subresultant-smult-right:assumes (c :: 'a :: {comm-ring-1, semiring-no-zero-divisors})
≠ 0
  shows subresultant J f (smult c g) = smult (c ^(degree f − J)) (subresultant J
f g)
  unfolding subresultant-swap[of - f] subresultant-smult-left[OF assms]
    degree-smult-eq using assms by (simp add: ac-simps)

lemma coeff-subresultant: coeff (subresultant J F G) l =
  (if degree F − J + (degree G − J) = 0 ∧ l ≠ 0 then 0 else det (subresultant'-mat
J l F G))
proof (cases degree F − J + (degree G − J) = 0)
  case True
 show ?thesis unfolding subresultant-def subresultant-mat-def subresultant'-mat-def
Let-def True
    by simp
next
  case False
  let ?n = degree F − J + (degree G − J)
  define n where n = ?n
  from False have n: n ≠ 0 unfolding n-def by auto
  hence id: {0..<n} = insert (n − 1) {0..< (n − 1)} by (cases n, auto)
  have idn: (x = x) = True for x :: nat by simp
  let ?M = subresultant-mat J F G
  define M where M = ?M
  let ?L = subresultant'-mat J l F G
  define L where L = ?L
  {
    fix p
    assume p: p permutes {0..<n}
    from n p have n1: n − 1 < n p (n − 1) < n by auto
    have coeff-int (∏ i = 0..<n. M $$ (i, p i)) l =
      (∏ i = 0 ..< (n − 1). coeff-int (M $$ (i, p i)) 0) * coeff-int (M $$ (n − 1,
p (n − 1))) l
      unfolding id
```

proof (*rule coeff-int-prod-const*, (*auto*)[*2*])
fix *i*
assume *i* ∈ {*0* ..< *n* − *1*}
with *p* **have** *i*: *i* ≠ *n* − *1* **and** *i* < *n p i* < *n* **by** (*auto simp*: *n-def*)
note *id* = *subresultant-index-mat*[*OF this*(*2*−*3*)[*unfolded n-def*], *folded M-def n-def*]
show *degree* (*M* $$ (*i*, *p i*)) = *0* **unfolding** *id Let-def* **using** *i*
by (*simp split*: *if-splits*)
qed
also have (∏ *i* = *0* ..< (*n* − *1*). *coeff-int* (*M* $$ (*i*, *p i*)) *0*)
= (∏ *i* = *0* ..< (*n* − *1*). *L* $$ (*i*, *p i*))
proof (*rule prod.cong*[*OF refl*])
fix *i*
assume *i* ∈ {*0* ..< *n* − *1*}
with *p* **have** *i*: *i* ≠ *n* − *1* **and** *ii*: *i* < *n p i* < *n* **by** (*auto simp*: *n-def*)
note *id* = *subresultant-index-mat*[*OF this*(*2*−*3*)[*unfolded n-def*], *folded M-def n-def*]
note *id'* = *L-def*[*unfolded subresultant'-mat-def Let-def*, *folded n-def*] *index-mat*[*OF ii*]
show *coeff-int* (*M* $$ (*i*, *p i*)) *0* = *L* $$ (*i*, *p i*)
unfolding *id id' split* **using** *i* **proof** (*simp add*: *if-splits Let-def*)
qed
qed
also have *coeff-int* (*M* $$ (*n* − *1*, *p* (*n* − *1*))) *l* =
(*if p* (*n* − *1*) < *degree G* − *J then*
coeff-int (*monom 1* (*degree G* − *J* − *1* − *p* (*n* − *1*)) ∗ *F*) *l*
else coeff-int (*monom 1* (*degree F* − *J* − *1* − (*p* (*n* − *1*) − (*degree G* − *J*))) ∗ *G*) *l*)
using *subresultant-index-mat*[*OF n1*[*unfolded n-def*], *folded M-def n-def*, *unfolded idn if-True Let-def*]
by *simp*
also have ... = (*if p* (*n* − *1*) < *degree G* − *J*
then coeff-int F (*int l* − *int* (*degree G* − *J* − *1* − *p* (*n* − *1*)))
else coeff-int G (*int l* − *int* (*degree F* − *J* − *1* − (*p* (*n* − *1*) − (*degree G* − *J*)))))
unfolding *coeff-int-monom-mult* **by** *simp*
also have ... = (*if p* (*n* − *1*) < *degree G* − *J*
then coeff-int F (*int l* − *int* (*degree G* − *J* − *1*) + *p* (*n* − *1*))
else coeff-int G (*int l* − *int* (*degree F* − *J* − *1*) + (*p* (*n* − *1*) − (*degree G* − *J*))))
proof (*cases p* (*n* − *1*) < *degree G* − *J*)
case *True*
hence *int* (*degree G* − *J* − *1* − *p* (*n* − *1*)) = *int* (*degree G* − *J* − *1*) − *p* (*n* − *1*) **by** *simp*
hence *id*: *int l* − *int* (*degree G* − *J* − *1* − *p* (*n* − *1*)) = *int l* − *int* (*degree G* − *J* − *1*) + *p* (*n* − *1*) **by** *simp*
show *?thesis* **using** *True* **unfolding** *id* **by** *simp*
next
case *False*

```
    from n1 False have degree F − J − 1 ≥ p (n − 1) − (degree G − J)
      unfolding n-def by linarith
    hence int (degree F − J − 1 − (p (n − 1) − (degree G − J))) = int (degree
F − J − 1) − (p (n − 1) − (degree G − J))
      by linarith
    hence id: int l − int (degree F − J − 1 − (p (n − 1) − (degree G − J))) =
      int l − int (degree F − J − 1) + (p (n − 1) − (degree G − J)) by simp
    show ?thesis unfolding id using False by simp
  qed
  also have ... = L $$ (n − 1, p (n − 1))
    unfolding L-def subresultant'-mat-def Let-def n-def[symmetric] using n1 by
simp
  also have (∏ i = 0..<n − 1. L $$ (i, p i)) ∗ ... = (∏ i = 0..<n. L $$ (i, p
i))
    unfolding id by simp
  finally have coeff-int (∏ i = 0..<n. M $$ (i, p i)) (int l) = (∏ i = 0..<n. L
$$ (i, p i)) .
 } note ∗ = this
 have coeff-int (subresultant J F G) l =
  (∑ p∈{p. p permutes {0..<n}}. signof p ∗ coeff-int (∏ i = 0..<n. M $$ (i, p
i)) l)
  unfolding subresultant-def det-def subresultant-mat-dim idn if-True n-def[symmetric]
M-def
    coeff-int-sum coeff-int-signof-mult by simp
 also have ... = (∑ p∈{p. p permutes {0..<n}}. signof p ∗ (∏ i = 0..<n. L $$
(i, p i)))
  by (rule sum.cong[OF refl], insert ∗, simp)
 also have ... = det L
 proof −
  have id: dim-row (subresultant'-mat J l F G) = n
    dim-col (subresultant'-mat J l F G) = n unfolding subresultant'-mat-def
Let-def n-def
    by auto
  show ?thesis unfolding det-def L-def id by simp
 qed
 finally show ?thesis unfolding L-def coeff-int-def using False by auto
qed

lemma subresultant'-zero-ge: assumes (degree f − J) + (degree g − J) ≠ 0 and
k ≥ degree f + (degree g − J)
 shows det (subresultant'-mat J k f g) = 0
proof −
 obtain dg where dg: degree g − J = dg by simp
 obtain df where df: degree f − J = df by simp
 obtain ddf where ddf: degree f = ddf by simp
 note ∗ = assms(2)[unfolded ddf dg] assms(1)
 define M where M = (λ i j. if j < dg
          then coeff-int f (degree f − int i + int j)
          else coeff-int g (degree g − int i + int (j − dg)))
```

```
  let ?M = subresultant'-mat J k f g
  have M: det ?M = det (mat (df + dg) (df + dg)
    (λ(i, j).
        if i = df + dg − 1 then
          if j < dg
            then coeff-int f (int k − int (dg − 1) + int j)
            else coeff-int g (int k − int (df − 1) + int (j − dg))
        else M i j)) (is - = det ?N)
    unfolding subresultant'-mat-def Let-def M-def
    by (rule arg-cong[of - - det], rule eq-matI, auto simp: df dg)
  also have ?N = mat (df + dg) (df + dg)
    (λ(i, j).
        if i = df + dg − 1 then 0
        else M i j)
    by (rule cong-mat[OF refl refl], unfold split, rule if-cong[OF refl - refl],
      auto simp add: coeff-int-def df dg ddf intro!: coeff-eq-0, insert ∗(1),
      unfold ddf[symmetric] dg[symmetric] df[symmetric], linarith+)
  also have ... = mat_r (df + dg) (df + dg) (λi. if i = df + dg − 1 then 0_v (df
+ dg) else
    vec (df + dg) (λ j. M i j))
    by (rule eq-matI, auto)
  also have det ... = 0
   by (rule det-row-0, insert ∗, auto simp: df[symmetric] dg[symmetric] ddf[symmetric])
  finally show ?thesis .
qed

lemma subresultant'-zero-lt: assumes
  J: J ≤ degree f J ≤ degree g J < k
  and k: k < degree f + (degree g − J)
  shows det (subresultant'-mat J k f g) = 0
proof −
  obtain dg where dg: dg = degree g − J by simp
  obtain df where df: df = degree f − J by simp
  note ∗ = assms[folded df dg]
  define M where M = (λ i j. if j < dg
          then coeff-int f (degree f − int i + int j)
          else coeff-int g (degree g − int i + int (j − dg)))
  define N where N = (λ j. if j < dg
          then coeff-int f (int k − int (dg − 1) + int j)
          else coeff-int g (int k − int (df − 1) + int (j − dg)))
  let ?M = subresultant'-mat J k f g
  have M: ?M = mat (df + dg) (df + dg)
    (λ(i, j).
        if i = df + dg − 1 then N j
        else M i j)
    unfolding subresultant'-mat-def Let-def
    by (rule eq-matI, auto simp: df dg M-def N-def)
  also have ... = mat (df + dg) (df + dg)
    (λ(i, j).
```

```
      if i = df + dg − 1 then N j
      else if i = degree f + dg − 1 − k then N j else M i j) (is - = ?N)
   unfolding N-def
   by (rule cong-mat[OF refl refl], unfold split, rule if-cong[OF refl refl], unfold
M-def N-def,
     insert J k, auto simp: df dg intro!: arg-cong[of - - coeff-int -])
 finally have id: ?M = ?N .
 have deg: degree f + dg − 1 − k < df + dg df + dg − 1 < df + dg
   using k J unfolding df dg by auto
 have id: row ?M (degree f + dg − 1 − k) = row ?M (df + dg − 1)
   unfolding arg-cong[OF id, of row]
   by (rule eq-vecI, insert deg, auto)
 show ?thesis
   by (rule det-identical-rows[OF - - - - id, of df + dg], insert deg assms,
     auto simp: subresultant'-mat-def Let-def df dg)
qed

lemma subresultant'-mat-sylvester-mat: transpose-mat (subresultant'-mat 0 0 f g)
= sylvester-mat f g
proof −
 obtain dg where dg: degree g = dg by simp
 obtain df where df: degree f = df by simp
 let ?M = transpose-mat (subresultant'-mat 0 0 f g)
 let ?n = degree f + degree g
 have dim: dim-row ?M = ?n dim-col ?M = ?n by (auto simp: subresultant'-mat-def
Let-def)
 show ?thesis
 proof (rule eq-matI, unfold sylvester-mat-dim dim df dg, goal-cases)
   case ij: (1 i j)
   have ?M $$ (i,j) = (if i < dg
       then if j = df + dg − 1
           then coeff-int f (− int (dg − 1) + int i)
           else coeff-int f (int df − int j + int i)
       else if j = df + dg − 1
           then coeff-int g (− int (df − 1) + int (i − dg))
           else coeff-int g (int dg − int j + int (i − dg)))
     using ij unfolding subresultant'-mat-def Let-def by (simp add: if-splits df
dg)
   also have ... = (if i < dg
       then coeff-int f (int df − int j + int i)
       else coeff-int g (int dg − int j + int (i − dg)))
   proof −
     have cong: (b ⟹ x = z) ⟹ (¬ b ⟹ y = z) ⟹ (if b then coeff-int f x else
coeff-int f y) = coeff-int f z
       for b x y z and f :: 'a poly by auto
     show ?thesis
       by (rule if-cong[OF refl cong cong], insert ij, auto)
   qed
   also have ... = sylvester-mat f g $$ (i,j)
```

```
    proof −
      have ∗: i ≤ j ⟹ j − i ≤ df ⟹ nat (int df − int j + int i) = df − (j −
i) for j i df
        by simp
      show ?thesis unfolding sylvester-index-mat[OF ij[folded df dg]] df dg
      proof (rule if-cong[OF refl])
        assume i: i < dg
        have int df − int j + int i < 0 ⟶ ¬ j − i ≤ df by auto
        thus coeff-int f (int df − int j + int i) = (if i ≤ j ∧ j − i ≤ df then coeff f
(df + i − j) else 0)
          using i ij by (simp add: coeff-int-def ∗, intro impI  coeff-eq-0[of f, unfolded
df], linarith)
      next
        assume i: ¬ i < dg
        hence ∗∗: i − dg ≤ j ⟹ dg − (j + dg − i) = i − j using ij by linarith
        have int dg − int j + int (i − dg) < 0 ⟶ ¬ j ≤ i by auto
        thus coeff-int g (int dg − int j + int (i − dg)) = (if i − dg ≤ j ∧ j ≤ i
then coeff g (i − j) else 0)
          using ij i by (simp add: coeff-int-def ∗ ∗∗, intro impI  coeff-eq-0[of g,
unfolded dg], linarith)
      qed
    qed
    finally show ?case .
  qed auto
qed

lemma coeff-subresultant-0-0-resultant: coeff (subresultant 0 f g) 0 = resultant f g
proof −
  let ?M = transpose-mat (subresultant'-mat 0 0 f g)
  have det (subresultant'-mat 0 0 f g) = det ?M
    by (subst det-transpose, auto simp: subresultant'-mat-def Let-def)
  also have ?M = sylvester-mat f g
    by (rule subresultant'-mat-sylvester-mat)
  finally show ?thesis by (simp add: coeff-subresultant resultant-def)
qed

lemma subresultant-zero-ge: assumes k ≥ degree f + (degree g − J)
  and (degree f − J) + (degree g − J) ≠ 0
  shows coeff (subresultant J f g) k = 0
  unfolding coeff-subresultant
  by (subst subresultant'-zero-ge[OF assms(2,1)], simp)

lemma subresultant-zero-lt: assumes k < degree f + (degree g − J)
  and J ≤ degree f J ≤ degree g J < k
  shows coeff (subresultant J f g) k = 0
  unfolding coeff-subresultant
  by (subst subresultant'-zero-lt[OF assms(2,3,4,1)], simp)

lemma subresultant-resultant: subresultant 0 f g = [: resultant f g :]
```

```
proof (cases degree f + degree g = 0)
  case True
 thus ?thesis unfolding subresultant-def subresultant-mat-def resultant-def Let-def
      sylvester-mat-def sylvester-mat-sub-def
    by simp
next
  case 0: False
  show ?thesis
  proof (rule poly-eqI)
    fix k
    show coeff (subresultant 0 f g) k = coeff [:resultant f g:] k
    proof (cases k = 0)
      case True
      thus ?thesis using coeff-subresultant-0-0-resultant[of f g] by auto
    next
      case False
      hence 0 < k ∧ k < degree f + degree g ∨ k ≥ degree f + degree g by auto
      thus ?thesis using subresultant-zero-ge[of f g 0 k] 0
        subresultant-zero-lt[of k f g 0] 0 False by (cases k, auto)
    qed
  qed
qed

lemma (in inj-comm-ring-hom) subresultant-hom:
  map-poly hom (subresultant J f g) = subresultant J (map-poly hom f) (map-poly
hom g)
proof −
  note d = subresultant-mat-def Let-def
  interpret p: map-poly-inj-comm-ring-hom hom..
  show ?thesis unfolding subresultant-def unfolding p.hom-det[symmetric]
  proof (rule arg-cong[of - - det])
    show p.mat-hom (subresultant-mat J f g) =
      subresultant-mat J (map-poly hom f) (map-poly hom g)
    proof (rule eq-matI, goal-cases)
      case (1 i j)
      hence ij: i < degree f − J + (degree g − J) j < degree f − J + (degree g −
J)
        unfolding d degree-map-poly by auto
      show ?case
      by (auto simp add: coeff-int-def d map-mat-def index-mat(1)[OF ij] hom-distribs)
    qed (auto simp: d)
  qed
qed
```

We now derive properties of the resultant via the connection to subresultants.

```
lemma resultant-smult-left: assumes (c :: 'a :: idom) ≠ 0
  shows resultant (smult c f) g = c ^ degree g ∗ resultant f g
  unfolding coeff-subresultant-0-0-resultant[symmetric] subresultant-smult-left[OF
```

assms] *coeff-smult*
by *simp*

lemma *resultant-smult-right*: **assumes** (*c* :: *'a* :: *idom*) ≠ *0*
shows *resultant f* (*smult c g*) = *c* ^ *degree f* ∗ *resultant f g*
unfolding *coeff-subresultant-0-0-resultant*[*symmetric*] *subresultant-smult-right*[*OF assms*] *coeff-smult*
by *simp*

lemma *resultant-swap*: *resultant f g* = (−*1*)^(*degree f* ∗ *degree g*) ∗ (*resultant g f*)
unfolding *coeff-subresultant-0-0-resultant*[*symmetric*]
unfolding *arg-cong*[*OF subresultant-swap*[*of 0 f g*], *of* λ *x. coeff x 0*] *coeff-smult*
by *simp*

The following equations are taken from Brown-Traub "On Euclid's Algorithm and the Theory of Subresultant" (BT)

lemma **fixes** *F B G H* :: *'a* :: *idom poly* **and** *J* :: *nat*
defines *df*: *df* ≡ *degree F*
and *dg*: *dg* ≡ *degree G*
and *dh*: *dh* ≡ *degree H*
and *db*: *db* ≡ *degree B*
defines
n: *n* ≡ (*df* − *J*) + (*dg* − *J*)
and *f*: *f* ≡ *coeff-int F*
and *b*: *b* ≡ *coeff-int B*
and *g*: *g* ≡ *coeff-int G*
and *h*: *h* ≡ *coeff-int H*
assumes *FGH*: *F* + *B* ∗ *G* = *H*
and *dfg*: *df* ≥ *dg*
and *choice*: *dg* > *dh* ∨ *H* = *0* ∧ *F* ≠ *0* ∧ *G* ≠ *0*
shows *BT-eq-18*: *subresultant J F G* = *smult* ((−*1*)^((*df* − *J*) ∗ (*dg* − *J*))) (*det* (*mat n n*
(λ (*i*,*j*).
if j < *df* − *J*
then if i = *n* − *1 then monom 1* ((*df* − *J*) − *1* − *j*) ∗ *G*
else [:*g* (*int dg* − *int i* + *int j*):]
else if i = *n* − *1 then monom 1* ((*dg* − *J*) − *1* − (*j* − (*df* − *J*))) ∗ *H*
else [:*h* (*int df* − *int i* + *int* (*j* − (*df* − *J*))):])))
(**is** - = *smult ?m1 ?right*)
and *BT-eq-19*: *dh* ≤ *J* ⟹ *J* < *dg* ⟹ *subresultant J F G* = *smult* (
(−*1*)^((*df* − *J*) ∗ (*dg* − *J*)) ∗ *lead-coeff G* ^ (*df* − *J*) ∗ *coeff H J* ^ (*dg* − *J* − *1*)) *H*
(**is** - ⟹ - ⟹ - = *smult* (- ∗ *?G* ∗ *?H*) *H*)
and *BT-lemma-1-12*: *J* < *dh* ⟹ *subresultant J F G* = *smult* (
(−*1*)^((*df* − *J*) ∗ (*dg* − *J*)) ∗ *lead-coeff G* ^ (*df* − *dh*)) (*subresultant J G H*)
and *BT-lemma-1-13'*: *J* = *dh* ⟹ *dg* > *dh* ∨ *H* ≠ *0* ⟹ *subresultant dh F G* = *smult* (
(−*1*)^((*df* − *dh*) ∗ (*dg* − *dh*)) ∗ *lead-coeff G* ^ (*df* − *dh*) ∗ *lead-coeff H* ^ (*dg* − *dh* − *1*)) *H*

```
  and BT-lemma-1-14: dh < J ⟹ J < dg − 1 ⟹ subresultant J F G = 0
  and BT-lemma-1-15': J = dg − 1 ⟹ dg > dh ∨ H ≠ 0 ⟹ subresultant (dg
− 1) F G = smult (
    (−1)^(df − dg + 1) * lead-coeff G ^ (df − dg + 1)) H
proof −
  define dfj where dfj = df − J
  define dgj where dgj = dg − J
  note d = df dg dh db
  have F0: F ≠ 0 using dfg choice df by auto
  have G0: G ≠ 0 using choice dg by auto
  have dgh: dg ≥ dh using choice unfolding dh by auto
  have B0: B ≠ 0 using FGH dfg dgh choice F0 G0 unfolding d by auto
  have dfh: df ≥ dh using dfg dgh by auto
  have df = degree (B * G)
  proof (cases H = 0)
    case False
    with choice dfg have dfh: df > dh by auto
    show ?thesis using dfh[folded arg-cong[OF FGH, of degree, folded dh]] choice
      unfolding df by (metis ‹degree (F + B * G) < df› degree-add-eq-left de-
gree-add-eq-right df nat-neq-iff)
  next
    case True
    have F = − B * G using arg-cong[OF FGH[unfolded True], of λ x. x − B *
G] by auto
    thus ?thesis using F0 G0 B0 unfolding df by simp
  qed
  hence dfbg: df = db + dg using degree-mult-eq[OF B0 G0] by (simp add: d)
  hence dbfg: db = df − dg by simp
  let ?dfj = df − J
  let ?dgj = dg − J
  have norm: ?dgj + ?dfj = ?dfj + ?dgj by simp
  let ?bij = λ i j. b (db − int i + int (j − dfj))
  let ?M = mat n n (λ (i,j). if i = j then 1 else if j < dfj then 0 else if i < j
    then [: ?bij i j :] else 0)
  let ?GF = λ i j.
            if j < dfj
            then if i = n − 1 then monom 1 (dfj − 1 − j) * G
                 else [:g (int dg − int i + int j):]
            else if i = n − 1 then monom 1 (dgj − 1 − (j − dfj)) * F
                 else [:f (int df − int i + int (j − dfj)):]
  let ?G-F = mat n n (λ (i,j). ?GF i j)
  let ?GH = λ i j.
            if j < dfj
            then if i = n − 1 then monom 1 (dfj − 1 − j) * G
                 else [:g (int dg − int i + int j):]
            else if i = n − 1 then monom 1 (dgj − 1 − (j − dfj)) * H
                 else [:h (int df − int i + int (j − dfj)):]
  let ?G-H = mat n n (λ (i,j). ?GH i j)
  have hfg: h i = f i + coeff-int (B * G) i for i
```

unfolding *FGH*[*symmetric*] *f g h* **unfolding** *coeff-int-def* **by** *simp*
have *dM1*: *det ?M = 1*
by (*subst det-upper-triangular*, (*auto*)[*2*], *subst prod-list-diag-prod*, *auto*)
have *subresultant J F G = smult ?m1* (*subresultant J G F*)
unfolding *subresultant-swap*[*of - F*] *d* **by** *simp*
also have *subresultant J G F = det ?G-F*
unfolding *subresultant-def n norm subresultant-mat-def g f Let-def d*[*symmetric*] *dfj-def dgj-def* **by** *simp*
also have ... = *det* (*?G-F* ∗ *?M*)
by (*subst det-mult*[*of - n*], *unfold dM1*, *auto*)
also have *?G-F* ∗ *?M = ?G-H*
proof (*rule eq-matI*, *unfold dim-col-mat dim-row-mat*)
fix *i j*
assume *i*: *i* < *n* **and** *j*: *j* < *n*
have (*?G-F* ∗ *?M*) $$ (*i*,*j*) = *row* (*?G-F*) *i* · *col ?M j*
using *i j* **by** *simp*
also have ... = *?GH i j*
proof (*cases j* < *dfj*)
case *True*
have *id*: *col ?M j = unit-vec n j*
by (*rule eq-vecI*, *insert True i j*, *auto*)
show *?thesis* **unfolding** *id* **using** *True i j* **by** *simp*
next
case *False*
define *d* **where** *d = j − dfj*
from *False* **have** *jd*: *j = d + dfj* **unfolding** *d-def* **by** *auto*
hence *idj*: {*0* ..< *j*} = {*0* ..< *dfj*} ∪ {*dfj* ..< *dfj* + *d*} **by** *auto*
let *?H* = (*if i = n − 1 then monom 1* (*dgj − Suc d*) ∗ *H else* [:*h* (*int df − int i + int d*):])
have *idr*: *?GH i j = ?H* **unfolding** *d-def* **using** *jd* **by** *auto*
let *?bi* = λ *i*. *b* (*db − int i + int d*)
let *?m* = λ *i*. *if i = j then 1 else if i < j then* [:*?bij i j*:] *else 0*
let *?P* = λ *k*. (*?GF i k* ∗ *?m k*)
let *?Q* = λ *k*. *?GF i k* ∗ [: *?bi k* :]
let *?G* = λ *k*. *if i = n − 1 then monom 1* (*dfj − 1 − k*) ∗ *G else* [:*g* (*int dg − int i + int k*):]
let *?Gb* = λ *k*. *?G k* ∗ [:*?bi k*:]
let *?off* = − (*int db − int dfj + 1 + int d*)
have *off0*: *?off* ≥ *0* **using** *False dfg j* **unfolding** *dfj-def d-def dbfg n* **by** *simp*
from *nat-0-le*[*OF this*]
obtain *off* **where** *off*: *int off = ?off* **by** *blast*
have *int off* ≤ *int dfj* **unfolding** *off* **by** *auto*
hence *off* ≤ *dfj* **by** *simp*
hence *split1*: {*0* ..< *dfj*} = {*0* ..< *off*} ∪ {*off* ..< *dfj*} **by** *auto*
have *int off + Suc db* ≤ *dfj* **unfolding** *off* **by** *auto*
hence *split2*: {*off* ..< *dfj*} = {*off* .. *off + db*} ∪ {*off + Suc db* ..< *dfj*} **by** *auto*
let *?g-b* = λ*k*. (*if i = n − 1 then monom 1 k* ∗ *G else* [:*g* (*int dg − int i + int* (*dfj − Suc k*)):]) ∗

```
        [:b (k − int off):]
    let ?gb = λk. (if i = n − 1 then monom 1 (k + off) ∗ G else [:g (int dg −
int i + int (dfj − Suc k − off)):]) ∗
        [:coeff B k:]
    let ?F = λ k. if i = n − 1 then monom 1 (dgj − 1 − (k − dfj)) ∗ F
             else [:f (int df − int i + int (k − dfj)):]
    let ?Fb = λ k. ?F k ∗ [:?bi k:]
    let ?Pj = if i = n − 1 then  monom 1 (dgj − Suc d) ∗ F else [:f (int df −
int i + int d):]
    from False have id: col ?M j = vec n ?m
      using j i by (intro eq-vecI, auto)
    have row ?G-F i · col ?M j = sum ?P {0 ..< n}
      using i j unfolding id by (simp add: scalar-prod-def)
    also have {0 ..< n} = {0 ..< j} ∪ {j} ∪ {Suc j ..< n} using j by auto
    also have sum ?P ... = sum ?P {0 ..< j} + ?P j + sum ?P {Suc j ..< n}
      by (simp add: sum.union-disjoint)
    also have sum ?P {Suc j ..< n} = 0 by (rule sum.neutral, auto)
    also have ?P j = ?Pj
      unfolding d-def using jd by simp
    also have sum ?P {0 ..< j} = sum ?Q {0 ..< j}
      by (rule sum.cong[OF refl], unfold d-def, insert jd, auto)
   also have sum ?Q {0 ..< j} = sum ?Q {0 ..< dfj} + sum ?Q {dfj ..< dfj+d}
unfolding idj
      by (simp add: sum.union-disjoint)
    also have sum ?Q {0 ..< dfj} = sum ?Gb {0 ..< dfj}
      by (rule sum.cong, auto)
    also have sum ?Q {dfj ..< dfj+d} = sum ?Fb {dfj ..< dfj+d}
      by (rule sum.cong, auto)
    also have ... = 0
    proof (rule sum.neutral, intro ballI)
      fix k
      assume k: k ∈ {dfj ..< dfj+d}
     hence k: db + d < k using k j False unfolding n db[symmetric] dfbg dfj-def
d-def by auto
      let ?k = (int db − int k + int d)
      have ?k < 0 using k by auto
      hence b ?k = 0 unfolding b by (intro coeff-int-eq-0, auto)
      thus ?Fb k = 0 by simp
    qed
    also have sum ?Gb {0 ..< dfj} = sum ?g-b {0 ..< dfj}
     proof (rule sum.reindex-cong[of λ k. dfj − Suc k], (auto simp: inj-on-def
off)[2], goal-cases)
      case (1 k)
       hence k = dfj − (Suc (dfj − Suc k)) and (dfj − Suc k) ∈ {0..<dfj}  by
auto
      thus ?case by blast
    next
      case (2 k)
      hence [simp]: dfj − Suc (dfj − Suc k) = k
```

```
      int db − int (dfj − Suc k) + int d = int k − off by (auto simp: off)
    show ?case by auto
  qed
  also have ... = sum ?g-b {0 ..< off} + sum ?g-b {off ..< dfj} unfolding split1
    by (simp add: sum.union-disjoint)
  also have sum ?g-b {0 ..< off} = 0
    by (rule sum.neutral, intro ballI, auto simp: b coeff-int-def)
  also have sum ?g-b {off ..< dfj} = sum ?g-b {off .. off + db} + sum ?g-b
{off + Suc db ..< dfj}
    unfolding split2 by (rule sum.union-disjoint, auto)
  also have sum ?g-b {off + Suc db ..< dfj} = 0
  proof (rule sum.neutral, intro ballI, goal-cases)
    case (1 k)
    hence b (int k − int off) = 0 unfolding b db
      by (intro coeff-int-eq-0, auto)
    thus ?case by simp
  qed
  also have sum ?g-b {off .. off + db} = sum ?gb {0 .. db}
    using sum.atLeastAtMost-shift-bounds [of ?g-b 0 off db]
    by (auto intro: sum.cong simp add: b ac-simps)
  finally have id: row ?G-F i · col ?M j − ?H = ?Pj + sum ?gb {0 .. db} −
?H
    (is - = ?E)
    by (simp add: ac-simps)
  define E where E = ?E
  let ?b = coeff B
  have Bsum: (∑ k = 0..db. monom (?b k) k) = B unfolding db
    using atMost-atLeast0 poly-as-sum-of-monoms by auto
  have E = 0
  proof (cases i = n − 1)
    case i-n: False
    hence id: (i = n − 1) = False by simp
    with i have i: i < n − 1 by auto
    let ?ii = int df − int i + int d
    have ?thesis = ([:f ?ii:] +
     (∑ k = 0..db.
      [:g (int dg − int i + int (dfj − Suc k − off)):] ∗ [:?b k:]) −
      [:h ?ii:] = 0) (is - = (?e = 0)) unfolding E-def id if-False by simp
    also have ?e = [: f ?ii +
     (∑ k = 0..db.
      g (int dg − int i + int (dfj − Suc k − off)) ∗ ?b k) −
      h ?ii:] (is - = [: ?e :])
    proof (rule poly-eqI, goal-cases)
      case (1 n)
      show ?case unfolding coeff-diff coeff-add coeff-sum coeff-const
        by (cases n, auto simp: ac-simps)
    qed
    also have [: ?e :] = 0 ⟷ ?e = 0 by simp
```

```
      also have ?e = (∑ k = 0..db. g (int dg − int i + int (dfj − Suc k − off))
∗ ?b k)
        − coeff-int (B ∗ G) ?ii
        unfolding hfg by simp
      also have (B ∗ G) = (∑ k = 0..db. monom (?b k) k) ∗ G unfolding Bsum
by simp
     also have ... = (∑ k = 0..db. monom (?b k) k ∗ G) by (rule sum-distrib-right)
      also have coeff-int ... ?ii = (∑ k = 0..db. g (?ii − k) ∗ ?b k)
        unfolding coeff-int-sum coeff-int-monom-mult g by (simp add: ac-simps)
      also have ... = (∑ k = 0..db. g (int dg − int i + int (dfj − Suc k − off))
∗ ?b k)
       proof (rule sum.cong[OF refl], goal-cases)
        case (1 k)
        hence k ≤ db by simp
        hence id: int dg − int i + int (dfj − Suc k − off) = ?ii − k
          using False i j off dfg
          unfolding dbfg d-def dfj-def n by linarith
        show ?case unfolding id ..
       qed
       finally show ?thesis by simp
     next
       case True
       let ?jj = dgj − Suc d
        have zero: int off − (dgj − Suc d) = 0 using dfg False j unfolding off
dbfg dfj-def d-def dgj-def n
         by linarith
       from True have E = monom 1 ?jj ∗ F + (∑ k = 0.. db.
         monom 1 (k + off) ∗ G ∗ [: ?b k :]) − monom 1 ?jj ∗ H
         (is - = ?A + ?sum − ?mon) unfolding id E-def by simp
       also have ?mon = monom 1 ?jj ∗ F + monom 1 ?jj ∗ (B ∗ G)
         unfolding FGH[symmetric] by (simp add: ring-distribs)
       also have ?A + ?sum − ... = ?sum − (monom 1 ?jj ∗ G) ∗ B (is - = - −
?GB ∗ B) by simp
       also have ?sum = (∑ k = 0..db.
         (monom 1 ?jj ∗ G) ∗ (monom 1 (k + off − ?jj) ∗ [: ?b k :]))
       proof (rule sum.cong[OF refl], goal-cases)
         case (1 k)
         let ?one = 1 :: 'a
         have int off ≥ int ?jj using j False i True
           unfolding off d-def dfj-def dgj-def dfbg n by linarith
         hence k + off = ?jj + (k + off − ?jj) by linarith
          hence id: monom ?one (k + off) = monom (1 ∗ 1) (?jj + (k + off −
?jj)) by simp
         show ?case unfolding id[folded mult-monom] by (simp add: ac-simps)
       qed
       also have ... = (monom 1 ?jj ∗ G) ∗ (∑ k = 0..db.  monom 1 (k + off −
?jj) ∗ [:?b k:])
         (is - = - ∗ ?sum)
         unfolding sum-distrib-left ..
```

```
      also have ... − (monom 1 ?jj ∗ G) ∗ B = (monom 1 ?jj ∗ G) ∗ (?sum −
B) by (simp add: ring-distribs)
      also have ?sum = (∑ k = 0..db.  monom 1 k ∗ [:?b k:])
        by (rule sum.cong[OF refl], insert zero, auto)
      also have ... = (∑ k = 0..db.  monom (?b k) k)
        by (rule sum.cong[OF refl], rule poly-eqI, auto)
      also have ... = B unfolding Bsum ..
      finally show ?thesis by simp
    qed
    from id[folded E-def, unfolded this]
    show ?thesis using False unfolding d-def by simp
   qed
   also have ... = ?G-H $$ (i,j) using i j by simp
   finally show (?G-F ∗ ?M) $$ (i,j) = ?G-H $$ (i,j) .
  qed auto
  finally show eq-18: subresultant J F G = smult ?m1 (det ?G-H) unfolding
dfj-def dgj-def .
  {
   fix i j
   assume ij: i < j and j: j < n
   with dgh have int dg − int i + int j > int dg by auto
   hence g (int dg − int i + int j) = 0 unfolding g dg by (intro coeff-int-eq-0,
auto)
  } note g0 = this
  {
   assume ∗: dh ≤ J J < dg
   have n-dfj: n > dfj using ∗ unfolding n dfj-def by auto
   note eq-18
   also have det ?G-H = prod-list (diag-mat ?G-H)
   proof (rule det-lower-triangular[of n])
    fix i j
    assume ij: i < j and j: j < n
    from ij j have if-e: i = n − 1 ⟷ False by auto
    have ?G-H $$ (i,j) = ?GH i j using ij j by auto
    also have ... = 0
    proof (cases j < dfj)
     case True
     with True g0[OF ij j] show ?thesis unfolding if-e by simp
    next
     case False
     have h (int df − int i + int (j − dfj)) = 0 unfolding h
      by (rule coeff-int-eq-0, insert False ∗ ij j dfg, unfold dfj-def dh[symmetric],
auto)
     with False show ?thesis unfolding if-e by auto
    qed
    finally show ?G-H $$ (i,j) = 0 .
   qed auto
   also have ... = (∏ i = 0..<n. ?GH i i)
    by (subst prod-list-diag-prod, simp)
```

```
  also have {0 ..< n} = {0 ..< dfj} ∪ {dfj ..< n} unfolding n dfj-def by auto
  also have prod (λ i. ?GH i i) ... = prod (λ i. ?GH i i) {0 ..< dfj} * prod (λ
i. ?GH i i) {dfj ..< n}
    by (simp add: prod.union-disjoint)
  also have prod (λ i. ?GH i i) {0 ..< dfj} = prod (λ i. [: lead-coeff G :]) {0 ..<
dfj}
  proof −
    show ?thesis
      by (rule prod.cong[OF refl], insert n-dfj, auto simp: g coeff-int-def dg)
  qed
  also have ... = [: (lead-coeff G)^dfj :] by (simp add: poly-const-pow)
  also have {dfj ..< n} = {dfj ..< n−1} ∪ {n − 1} using n-dfj by auto
  also have prod (λ i. ?GH i i) ... = prod (λ i. ?GH i i) {dfj ..< n−1} * ?GH
(n − 1) (n − 1)
    by (simp add: prod.union-disjoint)
  also have ?GH (n − 1) (n − 1) = H
  proof −
    have dgj − 1 − (n − 1 − dfj) = 0 using n-dfj unfolding dgj-def dfj-def n
by auto
    with n-dfj show ?thesis by auto
  qed
  also have prod (λ i. ?GH i i) {dfj ..< n−1} = prod (λ i. [:h (int df − dfj):])
{dfj ..< n−1}
    by (rule prod.cong[OF refl], auto intro!: arg-cong[of - - h])
  also have ... = [: h (int df − dfj) ^(n − 1 − dfj) :]
    unfolding prod-constant by (simp add: poly-const-pow)
  also have n − 1 − dfj = dg − J − 1 unfolding n dfj-def by simp
  also have int df − dfj = J using * dfg unfolding dfj-def by auto
  also have h J = coeff H J unfolding h coeff-int-def by simp
  finally show subresultant J F G = smult (?m1 * ?G * ?H) H by (simp add:
dfj-def ac-simps)
 } note eq-19 = this
 {
  assume J: J < dh
  define dhj where dhj = dh − J
  have n-add: n = (df − dh) + (dhj + dgj) unfolding dhj-def dgj-def n using
J dfg dgh by auto
  let ?split = split-block ?G-H (df − dh) (df − dh)
  have dim: dim-row ?G-H = (df − dh) + (dhj + dgj)
    dim-col ?G-H = (df − dh) + (dhj + dgj)
    unfolding n-add by auto
  obtain UL UR LL LR where spl: ?split = (UL,UR,LL,LR) by (cases ?split,
auto)
  note spl' = spl[unfolded split-block-def Let-def, simplified]
  let ?LR = subresultant-mat J G H
  have LR = mat (dgj + dhj) (dgj + dhj)
    (λ (i,j). ?GH (i + (df − dh)) (j + (df − dh)))
    using spl' by (auto simp: n-add)
  also have ... = ?LR
```

unfolding *subresultant-mat-def Let-def dhj-def dgj-def d*[*symmetric*]
proof (*rule eq-matI*, *unfold dim-row-mat dim-col-mat index-mat split dfj-def*, *goal-cases*)
case (*1 i j*)
hence *id1*: $(j + (df - dh) < df - J) = (j < dh - J)$ **using** *dgh dfg J* **by** *auto*
have *id2*: $(i + (df - dh) = n - 1) = (i = dg - J + (dh - J) - 1)$
unfolding *n-add dhj-def dgj-def* **using** *dgh dfg J* **by** *auto*
have *id3*: $(df - J - 1 - (j + (df - dh))) = (dh - J - 1 - j)$
and *id4*: $(int\ dg - int\ (i + (df - dh)) + int\ (j + (df - dh))) = (int\ dg - int\ i + int\ j)$
and *id5*: $(dg - J - 1 - (j + (df - dh) - (df - J))) = (dg - J - 1 - (j - (dh - J)))$
and *id6*: $(int\ df - int\ (i + (df - dh)) + int\ (j + (df - dh) - (df - J))) = (int\ dh - int\ i + int\ (j - (dh - J)))$
using *dgh dfg J* **by** *auto*
show *?case* **unfolding** *g*[*symmetric*] *h*[*symmetric*] *id3 id4 id5 id6*
by (*rule if-cong*[*OF id1 if-cong*[*OF id2 refl refl*] *if-cong*[*OF id2 refl refl*]])
qed *auto*
finally have $LR = ?LR$ **.**
note *spl* = *spl*[*unfolded this*]
let $?UR = 0_m\ (df - dh)\ (dgj + dhj)$
have $UR = mat\ (df - dh)\ (dgj + dhj)$
$(\lambda\ (i,j).\ ?GH\ i\ (j + (df - dh)))$
using *spl′* **by** (*auto simp*: *n-add*)
also have $\ldots = ?UR$
proof (*rule eq-matI*, *unfold dim-row-mat dim-col-mat index-mat split dfj-def index-zero-mat*, *goal-cases*)
case (*1 i j*)
hence *in1*: $i \neq n - 1$ **using** *J* **unfolding** *dgj-def dhj-def n-add* **by** *auto*
{
assume $j + (df - dh) < df - J$
hence $dg < int\ dg - int\ i + int\ (j + (df - dh))$ **using** *1 J* **unfolding** *dgj-def dhj-def* **by** *auto*
hence $g\ \ldots = 0$ **unfolding** *dg g* **by** (*intro coeff-int-eq-0*, *auto*)
} note *g* = *this*
{
assume $\neg\ (j + (df - dh) < df - J)$
hence $dh < int\ df - int\ i + int\ (j + (df - dh) - (df - J))$ **using** *1 J* **unfolding** *dgj-def dhj-def* **by** *auto*
hence $h\ \ldots = 0$ **unfolding** *dh h* **by** (*intro coeff-int-eq-0*, *auto*)
} note *h* = *this*
show *?case* **using** *in1 g h* **by** *auto*
qed *auto*
finally have $UR = ?UR$ **.**
note *spl* = *spl*[*unfolded this*]
let $?G = \lambda\ (i,j).\ if\ i = j\ then\ [:lead\text{-}coeff\ G:]\ else\ if\ i < j\ then\ 0\ else\ ?GH\ i\ j$
let $?UL = mat\ (df - dh)\ (df - dh)\ ?G$
have $UL = mat\ (df - dh)\ (df - dh)\ (\lambda\ (i,j).\ ?GH\ i\ j)$

```
    using spl' by (auto simp: n-add)
  also have ... = ?UL
 proof (rule eq-matI, unfold dim-row-mat dim-col-mat index-mat split, goal-cases)
    case (1 i j)
    {
     assume i = j
     hence int dg − int i + int j = dg using 1 by auto
     hence g (int dg − int i + int j) = lead-coeff G
       unfolding g dg coeff-int-def by simp
    } note eq = this
    {
     assume i < j
     hence dg < int dg − int i + int j using 1 by auto
     hence g (int dg − int i + int j) = 0
       unfolding g dg by (intro coeff-int-eq-0, auto)
    } note lt = this
    from 1 have ∗: j < dfj i ≠ n − 1 using J unfolding n-add dhj-def dgj-def
dfj-def by auto
    hence ?GH i j = [:g (int dg − int i + int j):] by simp
    also have ... = (if i = j then [: lead-coeff G :] else if i < j then 0 else ?GH i
j)
      using eq lt ∗ by auto
    finally show ?case by simp
  qed auto
  finally have UL = ?UL .
  note spl = spl[unfolded this]
  from split-block[OF spl dim]
  have GH: ?G-H = four-block-mat ?UL ?UR LL ?LR
    and C: ?UL ∈ carrier-mat (df − dh) (df − dh)
    ?UR ∈ carrier-mat (df − dh) (dhj + dgj)
    LL ∈ carrier-mat (dhj + dgj) (df − dh)
    ?LR ∈ carrier-mat (dhj + dgj) (dhj + dgj) by auto
  from arg-cong[OF GH, of det]
 have det ?G-H = det (four-block-mat ?UL ?UR LL ?LR) unfolding GH[symmetric]
..
  also have ... = det ?UL ∗ det ?LR
    by (rule det-four-block-mat-upper-right-zero[OF - refl], insert C, auto simp:
ac-simps)
  also have det ?LR = subresultant J G H unfolding subresultant-def by simp
  also have det ?UL = prod-list (diag-mat ?UL)
    by (rule det-lower-triangular[of df − dh], auto)
 also have ... = (∏ i = 0..< (df − dh). [: lead-coeff G :]) unfolding prod-list-diag-prod
by simp
  also have ... = [: lead-coeff G ^ (df − dh) :] by (simp add: poly-const-pow)
  finally have det: det ?G-H = [:lead-coeff G ^ (df − dh):] ∗ subresultant J G
H by auto
 show subresultant J F G = smult (?m1 ∗ lead-coeff G ^ (df − dh)) (subresultant
J G H)
    unfolding eq-18 det by simp
```

```
  }
  {
    assume J: dh < J J < dg − 1
    hence dh ≤ J J < dg by auto
    from eq-19[OF this]
    have subresultant J F G = smult ((− 1) ^ ((df − J) ∗ (dg − J)) ∗ lead-coeff
G ^ (df − J) ∗ coeff H J ^ (dg − J − 1)) H
      by simp
    also have coeff H J = 0 by (rule coeff-eq-0, insert J, auto simp: dh)
    also have ... ^ (dg − J − 1) = 0 using J by auto
    finally show subresultant J F G = 0 by simp
  }
  {
    assume J: J = dh and dg > dh ∨ H ≠ 0
    with choice have dgh: dg > dh by auto
    show subresultant dh F G = smult (
      (−1) ^ ((df − dh) ∗ (dg − dh)) ∗ lead-coeff G ^ (df − dh) ∗ lead-coeff H ^ (dg
− dh − 1)) H
      unfolding eq-19[unfolded J, OF le-refl dgh] unfolding dh by simp
  }
  {
    assume J: J = dg − 1 and dg > dh ∨ H ≠ 0
    with choice have dgh: dg > dh by auto
    have ∗: dh ≤ dg − 1 dg − 1 < dg using dgh by auto
    have ∗∗: df − (dg − 1) = df − dg + 1 dg − (dg − 1) − 1 = 0 dg − (dg −
1) = 1
      using dfg dgh by linarith+
    show subresultant (dg − 1) F G = smult (
      (−1) ^ (df − dg + 1) ∗ lead-coeff G ^ (df − dg + 1)) H
      unfolding eq-19[unfolded J, OF ∗] unfolding ∗∗ by simp
  }
qed

lemmas BT-lemma-1-13 = BT-lemma-1-13'[OF - - - refl]
lemmas BT-lemma-1-15 = BT-lemma-1-15'[OF - - - refl]

lemma subresultant-product: fixes F :: 'a :: idom poly
  assumes F = B ∗ G
  and FG: degree F ≥ degree G
shows subresultant J F G = (if J < degree G then 0 else
   if J < degree F then smult (lead-coeff G ^ (degree F − J − 1)) G else 1)
proof (cases J < degree G)
  case J: True
  from assms have eq: F + (−B) ∗ G = 0 by auto
  from J have lt: degree 0 < degree G ∨ b for b by auto
  from BT-lemma-1-13[OF eq FG lt lt]
  have subresultant 0 F G = 0 using J by auto
  with BT-lemma-1-14[OF eq FG lt, of J] have 00: J = 0 ∨ J < degree G − 1
⟹ subresultant J F G = 0 by auto
```

```
  from BT-lemma-1-15[OF eq FG lt lt] J have 01: subresultant (degree G − 1)
F G = 0 by simp
  from J have (J = 0 ∨ J < degree G − 1) ∨ J = degree G − 1 by linarith
  with 00 01 have subresultant J F G = 0 by auto
  thus ?thesis using J by simp
next
  case J: False
  hence dg: degree G − J = 0 by simp
  let ?n = degree F − J
  have ∗: (j :: nat) < 0 ⟷ False j − 0 = j for j by auto
  let ?M = mat ?n ?n
        (λ(i, j).
           if i = ?n − 1 then monom 1 (?n − 1 − j) ∗ G
           else [:coeff-int G (int (degree G) − int i + int j):])
  have subresultant J F G = det ?M
    unfolding subresultant-def subresultant-mat-def Let-def dg ∗ by auto
  also have det ?M = prod-list (diag-mat ?M)
    by (rule det-lower-triangular[of ?n], auto intro: coeff-int-eq-0)
  also have ... = (∏ i = 0..< ?n. ?M $$ (i,i)) unfolding prod-list-diag-prod by
simp
  also have ... = (∏ i = 0..< ?n. if i = ?n − 1 then G else [: lead-coeff G :])
    by (rule prod.cong[OF refl], auto simp: coeff-int-def)
  also have ... = (if J < degree F then smult (lead-coeff G ^ (?n − 1)) G else 1)
  proof (cases J < degree F)
    case True
    hence id: { 0 ..< ?n} = { 0 ..< ?n − 1} ∪ {?n − 1} by auto
    have (∏ i = 0..< ?n. if i = ?n − 1 then G else [: lead-coeff G :])
      = (∏ i = 0 ..< ?n − 1. if i = ?n − 1 then G else [: lead-coeff G :]) ∗ G (is
- = ?P ∗ G)
      unfolding id
      by (subst prod.union-disjoint, auto)
    also have ?P = (∏ i = 0 ..< ?n − 1. [: lead-coeff G :])
      by (rule prod.cong, auto)
    also have ... = [: lead-coeff G ^ (?n − 1) :]
      by (simp add: poly-const-pow)
    finally show ?thesis by auto
  qed auto
  finally have subresultant J F G =
     (if J < degree F then smult (lead-coeff G ^ (degree F − J − 1)) G else 1) .
  thus ?thesis using J by simp
qed

lemma resultant-pseudo-mod-0: assumes pseudo-mod f g = (0 :: 'a :: idom-divide
poly)
  and dfg: degree f ≥ degree g
  and f: f ≠ 0 and g: g ≠ 0
  shows resultant f g = (if degree g = 0 then lead-coeff g^degree f else 0)
proof −
  let ?df = degree f let ?dg = degree g
```

```
  obtain d r where pd: pseudo-divmod f g = (d,r) by force
  from pd have r: r = pseudo-mod f g unfolding pseudo-mod-def by simp
  with assms pd have pd: pseudo-divmod f g = (d,0) by auto
  from pseudo-divmod[OF g pd] g
  obtain a q where prod: smult a f = g * q and a: a ≠ 0 a = lead-coeff g ^ (Suc
?df - ?dg)
    by auto
  from a dfg have dfg: degree g ≤ degree (smult a f) by auto
  have g0: degree g = 0 ⟹ coeff g 0 = 0 ⟹ g = 0
    using leading-coeff-0-iff by fastforce
  from prod have smult a f = q * g by simp
   from arg-cong[OF subresultant-product[OF this dfg, of 0, unfolded subresul-
tant-resultant
    resultant-smult-left[OF a(1)]], of λ x. coeff x 0]
  show ?thesis using a g0 by (cases degree f, auto)
qed

locale primitive-remainder-sequence =
  fixes F :: nat ⇒ 'a :: idom-divide poly
    and n :: nat ⇒ nat
    and δ :: nat ⇒ nat
    and f :: nat ⇒ 'a
    and k :: nat
    and β :: nat ⇒ 'a
  assumes f: ⋀ i. f i = lead-coeff (F i)
      and n: ⋀ i. n i = degree (F i)
      and δ: ⋀ i. δ i = n i - n (Suc i)
      and n12: n 1 ≥ n 2
      and F12: F 1 ≠ 0 F 2 ≠ 0
      and F0: ⋀ i. i ≠ 0 ⟹ F i = 0 ⟷ i > k
      and β0: ⋀ i. β i ≠ 0
     and pmod: ⋀ i. i ≥ 3 ⟹ i ≤ Suc k ⟹ smult (β i) (F i) = pseudo-mod (F
(i - 2)) (F (i - 1))
begin

lemma f10: f 1 ≠ 0 and f20: f 2 ≠ 0 unfolding f using F12 by auto

lemma f0: i ≠ 0 ⟹ f i = 0 ⟷ i > k
  using F0[of i] unfolding f by auto

lemma n-gt: assumes 2 ≤ i i < k
  shows n i > n (Suc i)
proof -
  from assms have 3 ≤ Suc i Suc i ≤ Suc k by auto
  note pmod = pmod[OF this]
  from assms F0 have F (Suc i - 1) ≠ 0 F (Suc i) ≠ 0 by auto
  from pseudo-mod(2)[OF this(1), of F (Suc i - 2), folded pmod] this(2)
  show ?thesis unfolding n using β0 by auto
qed
```

lemma *n-ge*: **assumes** *1* ≤ *i i* < *k*
shows *n i* ≥ *n* (*Suc i*)
using *n12 n-gt*[*OF - assms*(*2*)] *assms*(*1*) **by** (*cases i* = *1*, *auto simp*: *numeral-2-eq-2*)

lemma *n-ge-trans*: **assumes** *1* ≤ *i i* ≤ *j j* ≤ *k*
shows *n i* ≥ *n j*
proof −
from *assms*(*2*) **have** *j* = *i* + (*j* − *i*) **by** *simp*
then obtain *jj* **where** *j*: *j* = *i* + *jj* **by** *blast*
from *assms*(*3*)[*unfolded j*] **show** *?thesis* **unfolding** *j*
proof (*induct jj*)
case (*Suc j*)
from *Suc*(*2*) **have** *i* + *j* ≤ *k* **by** *simp*
from *Suc*(*1*)[*OF this*] **have** *IH*: *n* (*i* + *j*) ≤ *n i* **.**
have *n* (*Suc* (*i* + *j*)) ≤ *n* (*i* + *j*)
by (*rule n-ge*, *insert assms*(*1*) *Suc*(*2*), *auto*)
with *IH* **show** *?case* **by** *auto*
qed *auto*
qed

lemma *delta-gt*: **assumes** *2* ≤ *i i* < *k*
shows *δ i* > *0* **using** *n-gt*[*OF assms*] **unfolding** *δ* **by** *auto*

lemma *k2*:*2* ≤ *k*
by (*metis le-cases linorder-not-le F0 F12*(*2*) *zero-order*(*2*))

lemma *k0*: *k* ≠ *0* **using** *k2* **by** *auto*

lemma *ni2*:*3* ≤ *i* ⟹ *i* ≤ *k* ⟹ *n i* ≠ *n 2*
by (*metis Suc-numeral δ delta-gt k2 le-imp-less-Suc le-less n-ge-trans not-le one-le-numeral*
semiring-norm(*5*) *zero-less-diff*)
end

locale *subresultant-prs-locale* = *primitive-remainder-sequence F n δ f k β* **for**
F :: *nat* ⇒ *'a* :: *idom-divide fract poly*
and *n* :: *nat* ⇒ *nat*
and *δ* :: *nat* ⇒ *nat*
and *f* :: *nat* ⇒ *'a fract*
and *k* :: *nat*
and *β* :: *nat* ⇒ *'a fract* +
fixes *G1 G2* :: *'a poly*
assumes *F1*: *F 1* = *map-poly to-fract G1*
and *F2*: *F 2* = *map-poly to-fract G2*

begin

definition $\alpha\ i = (f\ (i - 1)) \hat{}(Suc\ (\delta\ (i - 2)))$

lemma $\alpha 0$: $i > 1 \Longrightarrow \alpha\ i = 0 \longleftrightarrow (i - 1) > k$
unfolding α-*def* **using** *f0*[*of* $i - 1$] **by** *auto*

lemma α-*char*:
assumes $3 \leq i\ i < k + 2$
shows $\alpha\ i = (f\ (i - 1))\ \hat{}(Suc\ (length\ (coeffs\ (F\ (i - 2)))) - length\ (coeffs\ (F\ (i - 1))))$
proof (*cases* $i = 3$)
case *True*
have *triv*:*Suc* (*Suc* 0) = 2 **by** *auto*
have *l*:*length* (*coeffs* (*F* 2)) ≠ 0 *length* (*coeffs* (*F* 1)) ≠ 0 **using** *F12* **by** *auto*
hence *length* (*coeffs* (*F* 2)) ≤ *length* (*coeffs* (*F* (*Suc* 0))) **using** *n12*
unfolding *n degree-eq-length-coeffs One-nat-def* **by** *linarith*
hence *Suc* (*length* (*coeffs* (*F* 1)) − 1 − (*length* (*coeffs* (*F* 2)) − 1)) =
(*Suc* (*length* (*coeffs* (*F* 1))) − *length* (*coeffs* (*F* (3 − 1)))) **using** *l* **by** *simp*
thus *?thesis* **unfolding** *True* α-*def n* δ *degree-eq-length-coeffs* **by** (*simp add*:*triv*)
next
case *False*
hence *assms*:2 ≤ *i* − 2 *i* − 2 < *k* **using** *assms* **by** *auto*
have *i*:*i* − 2 ≠ 0 *i* − 1 ≠ 0 **using** *assms* **by** *auto*
hence [*simp*]: *Suc* (*i* − 2) = *i* − 1 **by** *auto*
from *assms*(2) *F0*[*OF i*(2)] **have** *F* (*i* − 1) ≠ 0 **by** *auto*
then have *length* (*coeffs* (*F* (*i* − 1))) > 0 **by** (*cases F* (*i* − 1)) *auto*
with *delta-gt*[*unfolded* δ *n degree-eq-length-coeffs*,*OF assms*]
have ∗ : *Suc* (δ (*i* − 2)) = *Suc* (*length* (*coeffs* (*F* (*i* − 2)))) − (*length* (*coeffs* (*F* (*Suc* (*i* − 2))))))
by (*auto simp*:δ *n degree-eq-length-coeffs*)
show *?thesis* **unfolding** α-*def* ∗ **by** *simp*
qed

definition *Q* :: *nat* ⇒ *'a fract poly* **where**
Q i ≡ *smult* (α *i*) (*fst* (*pdivmod* (*F* (*i* − 2)) (*F* (*i* − 1))))

lemma *beta-F-as-sum*:
assumes $3 \leq i\ i \leq Suc\ k$
shows *smult* (β *i*) (*F i*) = *smult* (α *i*) (*F* (*i* − 2)) + − *Q i* ∗ *F* (*i* − 1) (**is** *?t1*)
proof −
have *ik*:*i* < *k* + 2 **using** *assms* **by** *auto*
have *f0*:*F* (*i* − 1) = 0 ⟷ *False F* (*i* − *Suc* 0) = 0 ⟷ *False*
using *F0*[*of i* − 1] *assms* **by** *auto*
hence *f0-b*:(*inverse* (*coeff* (*F* (*i* − 1)) (*degree* (*F* (*i* − 1))))) ≠ 0 *F* (*i* − 1) ≠ 0 **by** *auto*
have *i*:*i* − 2 ≠ 0 *Suc* (*i* − 2) = *i* − 1 (*k* < *i* − 2) ⟷ *False* **using** *assms* **by** *auto*
have *F* (*i* − 2) ≠ 0 **using** *F0*[*of i* − 2] *assms* **by** *auto*

let *?c* = (*inverse* (*f* (*i* − *1*)) ^(*Suc* (*length* (*coeffs* (*F* (*i* − *2*)))) − *length* (*coeffs* (*F* (*i* − *1*)))))
have *inv*:*inverse* (α *i*) = *?c* **unfolding** α-*char*[*OF assms*(*1*) *ik*] *power-inverse* **by** *auto*
have *alpha0*:α *i* ≠ *0* **unfolding** α-*def f* **using** *f0* **by** *auto*
have α-*inv*[*simp*]:α *i* ∗ *inverse* (α *i*) = *1*
using *field-class.field-inverse*[*OF alpha0*] *mult.commute* **by** *metis*
with *field-class.field-inverse*[*OF alpha0*,*unfolded inv*]
have *c-times-Q*:*smult ?c* (*Q i*) = *fst* (*pdivmod* (*F* (*i* − *2*)) (*F* (*i* − *1*)))
unfolding *Q-def* **by** *auto*
have *pdivmod* (*F* (*i* − *2*)) (*F* (*i* − *1*)) = (*smult ?c* (*Q i*), *smult ?c* (*smult* (β *i*) (*F i*)))
unfolding *c-times-Q*
unfolding *pdivmod-via-pseudo-divmod pmod*[*OF assms*] *f n c-times-Q*
pseudo-mod-smult-right[*OF f0-b*, *of F* (*i* − *2*),*symmetric*] *f0 if-False Let-def*
unfolding *pseudo-mod-def* **by** (*auto split*:*prod.split*)
from *this* [*symmetric*]
have *pr*: ‹*F* (*i* − *2*) = *smult ?c* (*Q i*) ∗ *F* (*i* − *1*) + *smult ?c* (*smult* (β *i*) (*F i*))›
by (*simp only*: *prod-eq-iff fst-conv snd-conv div-mult-mod-eq*)
then have *F* (*i* − *2*) = *smult* (*inverse* (α *i*)) (*Q i*) ∗ *F* (*i* − *1*)
+ *smult* (*inverse* (α *i*)) (*smult* (β *i*) (*F i*)) (**is** *?l* = *?r* **is** - = *?t* + -)
unfolding *inv*.
hence *eq*:*smult* (α *i*) (*?l* − *?t*) = *smult* (α *i*) (*?r* − *?t*) **by** *auto*
have *smult* (α *i*) (*F* (*i* − *2*)) − *Q i* ∗ (*F* (*i* − *1*)) = *smult* (α *i*) (*?l* − *?t*)
unfolding *smult-diff-right* **by** *auto*
also have ... = *smult* (α *i*) (*?r* − *?t*) **unfolding** *eq*..
also have ... = *smult* (β *i*) (*F i*) **by** (*auto simp*:*mult.assoc*[*symmetric*])
finally show *?t1* **by** *auto*
qed

lemma assumes *3* ≤ *i i* ≤ *k* **shows**
BT-lemma-2-21: *j* < *n i* ⟹ *smult* (α *i* ^(*n* (*i* − *1*) − *j*)) (*subresultant j* (*F* (*i* − *2*)) (*F* (*i* − *1*)))
= *smult* ((− *1*) ^((*n* (*i* − *2*) − *j*) ∗ (*n* (*i* − *1*) − *j*)) ∗ (*f* (*i* − *1*)) ^(δ (*i* − *2*) + δ (*i* − *1*)) ∗ (β *i*) ^(*n* (*i* − *1*) − *j*)) (*subresultant j* (*F* (*i* − *1*)) (*F i*))
(**is** - ⟹ *?eq-21*) **and**
BT-lemma-2-22: *smult* (α *i* ^(δ (*i* − *1*))) (*subresultant* (*n i*) (*F* (*i* − *2*)) (*F* (*i* − *1*)))
= *smult* ((− *1*) ^((δ (*i* − *2*) + δ (*i* − *1*)) ∗ δ (*i* − *1*)) ∗ *f* (*i* − *1*) ^(δ (*i* − *2*) + δ (*i* − *1*)) ∗ *f i* ^(δ (*i* − *1*) − *1*) ∗ (β *i*) ^δ (*i* − *1*)) (*F i*)
(**is** *?eq-22*) **and**
BT-lemma-2-23: *n i* < *j* ⟹ *j* < *n* (*i* − *1*) − *1* ⟹ *subresultant j* (*F* (*i* − *2*)) (*F* (*i* − *1*)) = *0*
(**is** - ⟹ - ⟹ *?eq-23*) **and**
BT-lemma-2-24: *smult* (α *i*) (*subresultant* (*n* (*i* − *1*) − *1*) (*F* (*i* − *2*)) (*F* (*i* − *1*)))

```
  = smult ((- 1) ^(δ (i - 2) + 1) * f (i - 1) ^(δ (i - 2) + 1) * β i) (F i) (is
?eq-24)
proof -
  from assms have ik:i ≤ Suc k by auto
  note beta-F-as-sum = beta-F-as-sum[OF assms(1) ik, symmetric]
  have s[simp]:Suc (i - 2) = i - 1 Suc (i - 1) = i using assms by auto
  have α0:α i ≠ 0 using assms f0[of i - 1] unfolding α-def f by auto
  hence α0pow:⋀ x. α i ^ x ≠ 0 by auto
  have df:degree (F (i - 1)) ≤ degree (smult (α i) (F (i - 2)))
        degree (smult (β i) (F i)) < degree (F (i - 1)) ∨ b for b
    using n-ge[of i-2] n-gt[of i-1] assms α0 β0 unfolding n by auto
  have degree-smult-eq:⋀ c f. (c::-::{idom-divide}) ≠ 0 ⟹ degree (smult c f) =
degree f by auto
  have n-lt:n i < n (i - 1) using n-gt[of i-1] assms unfolding n by auto
  from semiring-normalization-rules(30) mult.commute
    have *:⋀ x y q. (x * (y::'a fract)) ^ q = y ^ q * x ^ q by metis
  have n (i - 1) - n i > 0 using n-lt by auto
  hence **:β i ^(n (i - 1) - n i - 1) * β i = β i ^(n (i - 1) - n i)
    by (subst power-minus-mult) auto
  have max (n (i - 2)) (n (i - 1)) = n (i - 2) using n-ge[of i-2] assms
    unfolding max-def by auto
  with diff-add-assoc[OF n-ge[of i-1],symmetric] assms
  have ns : n (i - 2) - n (i - 1) + (n (i - 1) - n i) = n (i - 2) - n i
     by (auto simp:nat-minus-add-max)
  { assume j < n i
    hence j:j < degree (smult (β i) (F i)) using β0 unfolding n by auto
    from BT-lemma-1-12[OF beta-F-as-sum df j]
    show ?eq-21
      unfolding subresultant-smult-right[OF β0] subresultant-smult-left[OF α0]
                  degree-smult-eq[OF α0] degree-smult-eq[OF β0] n[symmetric]
f[symmetric] δ s ns
      using f n
      by auto}
  { from BT-lemma-1-13[OF beta-F-as-sum df df(2)]
    show ?eq-22
      unfolding subresultant-smult-left[OF α0] lead-coeff-smult smult-smult
                  degree-smult-eq[OF α0] degree-smult-eq[OF β0] n[symmetric]
f[symmetric] δ s ns
      by (metis (no-types, lifting) * ** coeff-smult f mult.assoc n)}
  { assume n i < j j < n (i - 1) - 1
    hence j:degree (smult (β i) (F i)) < j j < degree (F (i - 1)) - 1
      using β0 unfolding n by auto
    from BT-lemma-1-14[OF beta-F-as-sum df j]
     show ?eq-23 unfolding subresultant-smult-left[OF α0] smult-eq-0-iff using
α0pow by auto}
  { have ***: n (i - 1) - (n (i - 1) - 1) = 1 using n-lt by auto
    from BT-lemma-1-15[OF beta-F-as-sum df df(2)]
    show ?eq-24
    unfolding subresultant-smult-left[OF α0] *** degree-smult-eq[OF α0] n[symmetric]
```

```
f δ
    by (auto simp:mult.commute)}
qed

lemma BT-eq-30: 3 ≤ i ⟹ i ≤ k + 1 ⟹ j < n (i − 1) ⟹
   smult (∏ l←[3..<i]. α l ^ (n (l − 1) − j)) (subresultant j (F 1) (F 2))
  = smult (∏ l←[3..<i]. β l ^ (n (l − 1) − j) * f (l − 1) ^ (δ (l − 2) + δ (l −
1))
       * (− 1) ^ ((n (l − 2) − j) * (n (l − 1) − j))) (subresultant j (F (i − 2))
(F (i − 1)))
proof (induct i − 3 arbitrary:i)
  case (Suc x)
  from Suc.hyps(2) Suc.prems(1−2)
    have prems:x = (i − 1) − 3 3 ≤ i − 1 i − 1 ≤ k + 1 2 ≤ i − 1 − 1 i − 1
− 1 < k
            i − 1 ≤ k by auto
  from prems(2) have inset:i − 1 ∈ set [3..<i] by auto
 have r1:remove1 (i − 1) [3..<i] = [3..<i−1] by (induct i,auto simp:remove1-append)
  from Suc.prems(1) have Suc (i − 1 − 1) = i − 1 by auto
  from n-gt[OF prems(4,5),unfolded this] Suc.prems(3) have j:j < n (i − 1 − 1)
by auto
  have *:⋀ c d e x. smult c d = e ⟹ smult (x * c) d = smult x e by auto
  have **:⋀ c d e x. smult c d = e ⟹ smult c (smult x d) = smult x e by (auto
simp:mult.commute)
  show ?case unfolding prod-list-map-remove1[OF inset(1),unfolded r1]
     *[OF Suc.hyps(1)[OF prems(1−3) j]]
     **[OF BT-lemma-2-21[OF prems(2,6) Suc.prems(3)]]
     by (auto simp: numeral-2-eq-2 ac-simps)
qed auto

lemma nonzero-alphaprod: assumes i ≤ k + 1 shows (∏ l←[3..<i]. α l ^ (p l))
≠ 0
  unfolding prod-list-zero-iff using assms by (auto simp: α0)

lemma BT-eq-30': assumes i: 3 ≤ i i ≤ k + 1 j < n (i − 1)
shows subresultant j (F 1) (F 2)
  = smult ((− 1) ^ (∑ l←[3..<i]. (n (l − 2) − j) * (n (l − 1) − j))
    * (∏ l←[3..<i]. (β l / α l) ^ (n (l − 1) − j)) * (∏ l←[3..<i]. f (l − 1) ^ (δ
(l − 2) + δ (l − 1)))) (subresultant j (F (i − 2)) (F (i − 1)))
  (is - = smult (?mm * ?b * ?f) -)
proof −
  let ?a = ∏ l←[3..<i]. α l ^ (n (l − 1) − j)
  let ?d = ∏ l←[3..<i]. β l ^ (n (l − 1) − j) * f (l − 1) ^ (δ (l − 2) + δ (l −
1)) *
                  (− 1) ^ ((n (l − 2) − j) * (n (l − 1) − j))
  let ?m = ∏ l←[3..<i]. (− 1) ^ ((n (l − 2) − j) * (n (l − 1) − j))
  have a0: ?a ≠ 0 by (rule nonzero-alphaprod, rule i)
  with arg-cong[OF BT-eq-30[OF i], of smult (inverse ?a), unfolded smult-smult]
  have subresultant j (F 1) (F 2) = smult (inverse ?a * ?d)
```

(*subresultant j* (*F* (*i* − *2*)) (*F* (*i* − *1*)))
by *simp*
also have *inverse ?a* ∗ *?d* = *?b* ∗ *?f* ∗ *?m* **unfolding** *prod-list-multf inverse-prod-list map-map o-def*
power-inverse[*symmetric*] *power-mult-distrib divide-inverse-commute*
by *simp*
also have *?m* = *?mm*
unfolding *prod-list-minus-1-exp* **by** *simp*
finally show *?thesis* **by** (*simp add*: *ac-simps*)
qed

For defining the subresultant PRS, we mainly follow Brown's "The Subresultant PRS Algorithm" (B).

definition *R j* = (*if j* = *n 2 then sdiv-poly* (*smult* ((*lead-coeff G2*)^(δ *1*)) *G2*) (*lead-coeff G2*) *else subresultant j G1 G2*)

abbreviation *ff i* ≡ *to-fract* (*i* :: *'a*)
abbreviation *ffp* ≡ *map-poly ff*

sublocale *map-poly-hom*: *map-poly-inj-idom-hom to-fract*..

definition σ *i* = (∑ *l*←[*3*..<*Suc i*]. (*n* (*l* − *2*) + *n* (*i* − *1*) + *1*) ∗ (*n* (*l* − *1*) + *n* (*i* − *1*) + *1*))
definition τ *i* = (∑ *l*←[*3*..<*Suc i*]. (*n* (*l* − *2*) + *n i*) ∗ (*n* (*l* − *1*) + *n i*))

definition γ *i* = (−*1*)^(σ *i*) ∗ *pow-int* (*f* (*i* − *1*)) (*1* − *int* (δ (*i* − *1*))) ∗ (∏ *l*←[*3*..<*Suc i*].
(β *l* / α *l*)^(*n* (*l* − *1*) − *n* (*i* − *1*) + *1*) ∗ (*f* (*l* − *1*))^(δ (*l* − *2*) + δ (*l* − *1*)))
definition Θ *i* = (−*1*)^(τ *i*) ∗ *pow-int* (*f i*) (*int* (δ (*i* − *1*)) − *1*) ∗ (∏ *l*←[*3*..<*Suc i*].
(β *l* / α *l*)^(*n* (*l* − *1*) − *n i*) ∗ (*f* (*l* − *1*))^(δ (*l* − *2*) + δ (*l* − *1*)))

lemma *fundamental-theorem-eq-4*: **assumes** *i*: *3* ≤ *i i* ≤ *k*
shows *ffp* (*R* (*n* (*i* − *1*) − *1*)) = *smult* (γ *i*) (*F i*)
proof −
have *n* (*i* − *1*) ≤ *n 2* **by** (*rule n-ge-trans*, *insert i*, *auto*)
with *n-gt*[*of i* − *1*] *i* **have** *n* (*i* − *1*) − *1* < *n 2*
and *lt*: *n* (*i* − *1*) − *1* < *n* (*i* − *1*) **by** *linarith*+
hence *R* (*n* (*i* − *1*) − *1*) = *subresultant* (*n* (*i* − *1*) − *1*) *G1 G2*
unfolding *R-def* **by** *auto*
from *arg-cong*[*OF this*, *of ffp*, *unfolded to-fract-hom.subresultant-hom*, *folded F1 F2*]
have *id1*: *ffp* (*R* (*n* (*i* − *1*) − *1*)) = *subresultant* (*n* (*i* − *1*) − *1*) (*F 1*) (*F 2*) .
note *eq-24* = *BT-lemma-2-24*[*OF i*]
let *?o* = (− *1*) :: *'a fract*
let *?m1* = (δ (*i* − *2*) + *1*)
let *?d1* = *f* (*i* − *1*) ^ (δ (*i* − *2*) + *1*) ∗ β *i*

let *?c1* = *?o* ^ *?m1* * *?d1*
let *?c0* = α *i*
have *?c0* ≠ *0* **using** α*0*[*of i*] *i* **by** *auto*
with *arg-cong*[*OF eq-24*, *of smult* (*inverse ?c0*)]
have *id2*: *subresultant* (*n* (*i* − *1*) − *1*) (*F* (*i* − *2*)) (*F* (*i* − *1*)) =
smult (*inverse ?c0* * *?c1*) (*F i*)
by (*auto intro*: *poly-eqI*)
from *i* **have** *3* ≤ *i i* ≤ *k* + *1* **by** *auto*
note *id3* = *BT-eq-30'*[*OF this lt*]
let *?f* = λ *l*. *f* (*l* − *1*) ^(δ (*l* − *2*) + δ (*l* − *1*))
let *?b* = λ *l*. (β *l* / α *l*) ^(*n* (*l* − *1*) − (*n* (*i* − *1*) − *1*))
let *?b'* = λ *l*. (β *l* / α *l*) ^(*n* (*l* − *1*) − *n* (*i* − *1*) + *1*)
let *?m* = λ *l*. (*n* (*l* − *2*) − (*n* (*i* − *1*) − *1*)) * (*n* (*l* − *1*) − (*n* (*i* − *1*) − *1*))
let *?m'* = λ *l*. (*n* (*l* − *2*) + *n* (*i* − *1*) + *1*) * (*n* (*l* − *1*) + *n* (*i* − *1*) + *1*)
let *?m2* = (∑ *l*←[*3*..<*i*]. *?m l*)
let *?b2* = (∏ *l*←[*3*..<*i*]. *?b l*)
let *?f2* = (∏ *l*←[*3*..<*i*]. *?f l*)
let *?f1* = *pow-int* (*f* (*i* − *1*)) (*1* − *int* (δ (*i* − *1*)))
have *id4*: γ *i* = *?o* ^(*?m1* + *?m2*) * (*inverse ?c0* * *?d1* * *?b2* * *?f2*)
proof −
have *id*: γ *i* = (−*1*)^(σ *i*) * (*?f1* * (∏ *l*←[*3*..<*Suc i*]. *?b' l*) * (∏ *l*←[*3*..<*Suc i*]. *?f l*))
unfolding γ*-def prod-list-multf* **by** *simp*
have *cong*: *even m1* = *even m2* ⟹ *c1* = *c2* ⟹ *?o*^*m1* * *c1* = *?o*^*m2* * *c2* **for** *m1 m2 c1 c2*
unfolding *minus-1-power-even* **by** *auto*
show *?thesis* **unfolding** *id*
proof (*rule cong*)
from *n-gt*[*of i* − *1*] *i* **have** *n1*: *n* (*i* − *1*) ≠ *0* **by** *linarith*
{
fix *l*
assume *2* ≤ *l l* ≤ *i*
hence *l*: *l* ≥ *2 l* − *1* ≤ *i* − *1 l* ≤ *k* **using** *i* **by** *auto*
from *n-ge-trans*[*OF - l*(*2*)] *l i* **have** *n2*: *n* (*i* − *1*) ≤ *n* (*l* − *1*) **by** *auto*
from *n1 n2* **have** *id*: *n* (*l* − *1*) − (*n* (*i* − *1*) − *1*) = *n* (*l* − *1*) − *n* (*i* − *1*) + *1* **by** *auto*
have *even* (*n* (*l* − *1*) − (*n* (*i* − *1*) − *1*)) = *even* (*n* (*l* − *1*) + *n* (*i* − *1*) + *1*)
unfolding *id* **using** *n2* **by** *auto*
note *id n2 this*
} note *diff* = *this*
have *f0*: *f* (*i* − *1*) ≠ *0* **using** *f0*[*of i* − *1*] *i* **by** *auto*
have (∏ *l*←[*3*..<*Suc i*]. *?b' l*) = (∏ *l*←[*3*..<*Suc i*]. *?b l*)
by (*rule arg-cong*, *rule map-cong*, *use diff*(*1*) **in** *auto*)
also have ... = *?b2* * *?b i* **using** *i* **by** *auto*
finally have *?f1* * (∏ *l*←[*3*..<*Suc i*]. *?b' l*) * (∏ *l*←[*3*..<*Suc i*]. *?f l*) =
(*?b2* * *?f2*) * (*?f1* * *?b i* * *?f i*) **using** *i* **by** *simp*
also have *?f1* * *?b i* * *?f i* = (*?f1* * *?f i*) * β *i* * *inverse ?c0* **using** *n1* **by** (*simp add*: *divide-inverse*)

```
    also have ?f1 * ?f i = f (i - 1) ^ (δ (i - 2) + 1)
      unfolding exp-pow-int pow-int-add[OF f0, symmetric] by simp
    finally
    show ?f1 * (∏ l←[3..<Suc i]. ?b' l) * (∏ l←[3..<Suc i]. ?f l)
      = inverse ?c0 * ?d1 * ?b2 * ?f2 by simp
   have even (σ i) = even ((∑ l←[3..<i]. ?m' l) + ?m' i) unfolding σ-def
using i by simp
    also have ... = (even (∑ l←[3..<i]. ?m' l) = even (?m' i)) by simp
    also have even (∑ l←[3..<i]. ?m' l) = even ?m2
    proof (rule even-sum-list, goal-cases)
      case (1 l)
      hence l: l ≥ 2 l ≤ i and l1: l - 1 ≥ 2 l - 1 ≤ i by auto
      have l2: l - 2 = l - 1 - 1 by simp
      show ?case using diff(3) [OF l] diff(3) [OF l1] l2
        by auto
    qed
    also have even (?m' i) = even ?m1
    proof -
      from i have id: Suc (i - 1 - 1) = i - 1 i - 2 = i - 1 - 1 by auto
      have even ?m1 = even (n (i - 2) + n (i - 1) + 1) unfolding δ id
        using diff[of i - 1] i by auto
      also have ... = even (?m' i) by auto
      finally show ?thesis by simp
    qed
   also have (even ?m2 = even ?m1) = even (?m2 + ?m1) unfolding even-add
by simp
    also have ?m2 + ?m1 = ?m1 + ?m2 by simp
    finally show even (σ i) = even (?m1 + ?m2) .
  qed
 qed
 show ?thesis unfolding id1 id3 id2 smult-smult id4 by (simp add: ac-simps
power-add)
qed


lemma fundamental-theorem-eq-5: assumes i: 3 ≤ i i ≤ k n i < j j < n (i - 1)
- 1
 shows R j = 0
proof -
 from BT-lemma-2-23[OF i] have id1: subresultant j (F (i - 2)) (F (i - 1)) =
0 .
 have n (i - 1) ≤ n 2 by (rule n-ge-trans, insert i, auto)
 with n-gt[of i - 1] i have n (i - 1) - 1 < n 2
   and lt: j < n (i - 1) by linarith+
 with i have R j = subresultant j G1 G2 unfolding R-def by auto
 from arg-cong[OF this, of ffp, unfolded to-fract-hom.subresultant-hom, folded F1
F2]
 have id2: ffp (R j) = subresultant j (F 1) (F 2) .
 from i have 3 ≤ i i ≤ k + 1 by auto
```

note *eq-30* = *BT-eq-30*[*OF this lt*]
let *?c3* = $\prod l{\leftarrow}[3..{<}i].\ \alpha\ l\ \hat{}\ (n\ (l-1)-j)$
let *?c2* = $\prod l{\leftarrow}[3..{<}i].\ \beta\ l\ \hat{}\ (n\ (l-1)-j) * f\ (l-1)\ \hat{}\ (\delta\ (l-2)+\delta\ (l-1)) *$
$(-1)\ \hat{}\ ((n\ (l-2)-j) * (n\ (l-1)-j))$
have *?c3* $\neq$ *0* **by** (*rule nonzero-alphaprod*, *insert i*, *auto*)
with *arg-cong*[*OF eq-30*, *of smult* (*inverse ?c3*)]
have *id3*: *subresultant j* (*F 1*) (*F 2*) = *smult* (*inverse ?c3* * *?c2*)
(*subresultant j* (*F* (*i* − *2*)) (*F* (*i* − *1*)))
by (*auto intro*: *poly-eqI*)
have *ffp* (*R j*) = *0* **unfolding** *id1 id2 id3* **by** *simp*
thus *?thesis* **by** *simp*
qed

lemma *fundamental-theorem-eq-6*: **assumes** *3* $\leq$ *i i* $\leq$ *k* **shows** *ffp* (*R* (*n i*)) = *smult* (Θ *i*) (*F i*)
(**is** *?lhs*=*?rhs*)
proof −
from *assms* **have** *i1*:*1* $\leq$ *i* **by** *auto*
from *assms* **have** *nlt*:*i* $\leq$ *k* + *1 n i* < *n* (*i* − *1*) **using** *n-gt*[*of i* − *1*] **by** *auto*
from *assms* **have** *αnz*:*α i* $\hat{}$ *δ* (*i* − *1*) $\neq$ *0* **using** *α0* **by** *auto*
have *:$\bigwedge$ *a f b*. *a* $\neq$ *0* $\Longrightarrow$ *smult a f* = *b* $\Longrightarrow$ *f* = *smult* (*inverse* (*a*::*'a fract*)) *b*
by *auto*
have **:$\bigwedge$ *f g xs c*. *c* * *prod-list* (*map f xs*) * *prod-list* (*map g xs*)
= *c* * ($\prod x{\leftarrow}xs$. *f x* * (*g*:: - ⇒ (- :: *comm-monoid-mult*)) *x*)
by (*auto simp*:*ac-simps prod-list-multf*)
have ***:$\bigwedge$ *c*. *β i* $\hat{}$ *δ* (*i* − *Suc 0*) * (*inverse* (*α i* $\hat{}$ *δ* (*i* − *Suc 0*)) * *c*) = (*β i* / *α i*) $\hat{}$ *δ* (*i* − *1*) * *c*
by (*auto simp*:*inverse-eq-divide power-divide*)
have ****:*int* (*n* (*i* − *Suc 0*) − *n i*) − *1* = *int* (*n* (*i* − *1*) − *Suc* (*n i*))
using *assms nlt* **by** *auto*
from *assms n-ge*[*of i*−*2*] *nlt n-ge*[*of i*]
have *nge*:*n* (*i* − *Suc 0*) $\leq$ *n* (*i* − *2*) *n i* < *n* (*i* − *Suc 0*) *n i* < *n* (*i* − *1*) *Suc* (*i* − *2*) = *i* − *1*
by (*cases i*,*auto simp*:*numeral-2-eq-2 numeral-3-eq-3*)
have *****:(− *1* :: *'a fract*) $\hat{}$ ((*n* (*i* − *Suc 0*) − *n i*) * (*n* (*i* − *Suc 0*) − *n i* + (*n* (*i* − *2*) − *n* (*Suc* (*i* − *2*)))))
= (− *1*) $\hat{}$ ((*n i* + *n* (*i* − *Suc 0*)) * (*n i* + *n* (*i* − *2*)))
(− *1* :: *'a fract*) $\hat{}$ ($\sum l{\leftarrow}[3..{<}i]$. (*n* (*l* − *Suc 0*) − *n i*) * (*n* (*l* − *2*) − *n i*))
= (− *1*) $\hat{}$ ($\sum l{\leftarrow}[3..{<}i]$. (*n i* + *n* (*l* − *Suc 0*)) * (*n i* + *n* (*l* − *2*)))
using *nge* **apply** (*intro minus-1-even-eqI*,*auto*)
apply (*intro minus-1-even-eqI*)
apply (*intro even-sum-list*)
proof(*goal-cases*) **case** (*1 x*)
with *n-ge-trans assms*
have *n i* $\leq$ *n* (*x* − *Suc 0*) *n* (*x* − *2*) $\geq$ *n i* **by** *auto*
with *1* **show** *?case* **by** *auto*
qed

```
  have ffp (R (n i)) = subresultant (n i) (F 1) (F 2) unfolding R-def F1 F2
    by (auto simp: to-fract-hom.subresultant-hom ni2[OF assms])
  also have ... = smult
     ((− 1) ^(∑ l←[3..<i]. (n (l − 2) − n i) ∗ (n (l − 1) − n i)) ∗
      (∏ x←[3..<i]. (β x / α x) ^(n (x − 1) − n i) ∗ f (x − 1) ^(δ (x − 1) +
δ (x − 2))) ∗
      (((β i / α i) ^δ (i − 1)) ∗ f (i − 1) ^(δ (i − 1) + δ (i − 2))) ∗
       ((− 1) ^((δ (i − 2) + δ (i − 1)) ∗ δ (i − 1)) ∗ f i ^(δ (i − 1) − 1)
        ))
     (F i)
    unfolding BT-eq-30'[OF assms(1) nlt] **
         *[OF αnz BT-lemma-2-22[OF assms]] smult-smult by (auto simp:ac-simps
*** )
  also have ... = ?rhs unfolding Θ-def τ-def
    using prod-combine[OF assms(1)] δ assms
    by (auto simp:ac-simps exp-pow-int[symmetric] power-add ***** ****)
  finally show ?thesis.
qed


lemma fundamental-theorem-eq-7: assumes j: j < n k shows R j = 0
proof −
  let ?P = pseudo-divmod (F (k − 1)) (F k)
  from F0[of k] k2 have Fk: F k ≠ 0 by auto
  from pmod[of Suc k] k2 F0[of Suc k]
  have pseudo-mod (F (k − 1)) (F k) = 0 by auto
  then obtain Q where ?P = (Q,0)
    unfolding pseudo-mod-def by (cases ?P, auto)
  from pseudo-divmod(1)[OF Fk this] Fk obtain c where id: smult c (F (k −
1)) = F k ∗ Q
    and c: c ≠ 0 by auto
  from id have id: smult c (F (k − 1)) = Q ∗ F k by auto
  from n-ge[unfolded n, of k − 1] k2 c have degree (F k) ≤ degree (smult c (F (k
− 1))) by auto
  from subresultant-product[OF id this, unfolded subresultant-smult-left[OF c], of
j] j
  have ∗:subresultant j (F (k + 1 − 2)) (F (k + 1 − 1)) = 0 using c unfolding
n by simp
  from assms have **:j ≠ n 2
    by (meson k2 n-ge-trans not-le one-le-numeral order-refl)
  from k2 assms have 3 ≤ k + 1 k + 1 ≤ k + 1 j < n (k + 1 − 1) by auto
  from BT-eq-30[OF this,unfolded ∗] nonzero-alphaprod[OF le-refl] ** F1 F2
  show ?thesis by (auto simp:R-def F0 to-fract-hom.subresultant-hom[symmetric])
qed


definition G i = R (n (i − 1) − 1)
definition H i = R (n i)
```

lemma *gamma-delta-beta-3*: $\gamma\ 3 = (-\ 1)\ \hat{}\ (\delta\ 1 + 1) * \beta\ 3$
proof −
have $\gamma\ 3 = (-\ 1)\ \hat{}\ \sigma\ 3 * pow\text{-}int\ (f\ 2)\ (1 - int\ (\delta\ 2)) *$
$(\beta\ 3\ /\ (f\ 2\ \hat{}\ Suc\ (\delta\ 1)) * f\ 2\ \hat{}\ (\delta\ 1 + \delta\ 2))$
unfolding γ*-def* δ α*-def* **by** (*simp add*: δ)
also have $f\ 2\ \hat{}\ (\delta\ 1 + \delta\ 2) = pow\text{-}int\ (f\ 2)\ (int\ (\delta\ 1 + \delta\ 2))$
unfolding *pow-int-def nat-int* **by** *auto*
also have $int\ (\delta\ 1 + \delta\ 2) = int\ (Suc\ (\delta\ 1)) + (int\ (\delta\ 2) - 1)$ **by** *simp*
also have $pow\text{-}int\ (f\ 2)\ \ldots = pow\text{-}int\ (f\ 2)\ (Suc\ (\delta\ 1)) * pow\text{-}int\ (f\ 2)\ (int\ (\delta\ 2) - 1)$
by (*rule pow-int-add*, *insert f20*, *auto*)
also have $pow\text{-}int\ (f\ 2)\ (Suc\ (\delta\ 1)) = f\ 2\ \hat{}\ (Suc\ (\delta\ 1))$ **unfolding** *pow-int-def nat-int* **by** *simp*
also have $\beta\ 3\ /\ (f\ 2\ \hat{}\ Suc\ (\delta\ 1)) *$
$(f\ 2\ \hat{}\ Suc\ (\delta\ 1) * pow\text{-}int\ (f\ 2)\ (int\ (\delta\ 2) - 1))$
$= (\beta\ 3\ /\ (f\ 2\ \hat{}\ Suc\ (\delta\ 1)) *\ f\ 2\ \hat{}\ Suc\ (\delta\ 1) * pow\text{-}int\ (f\ 2)\ (int\ (\delta\ 2) - 1))$
by *simp*
also have $\beta\ 3\ /\ (f\ 2\ \hat{}\ Suc\ (\delta\ 1)) * f\ 2\ \hat{}\ Suc\ (\delta\ 1) = \beta\ 3$ **using** *f20* **by** *auto*
finally have $\gamma\ 3 = ((-\ 1)\ \hat{}\ \sigma\ 3 * \beta\ 3) * (pow\text{-}int\ (f\ 2)\ (1 - int\ (\delta\ 2)) * pow\text{-}int\ (f\ 2)\ (int\ (\delta\ 2) - 1))$
by *simp*
also have $pow\text{-}int\ (f\ 2)\ (1 - int\ (\delta\ 2)) * pow\text{-}int\ (f\ 2)\ (int\ (\delta\ 2) - 1)$
$= 1$
by (*subst pow-int-add*[*symmetric*], *insert f20*, *auto*)
finally have $\gamma\ 3 = (-\ 1)\ \hat{}\ \sigma\ 3 * \beta\ 3$ **by** *simp*
also have $\sigma\ 3 = (n\ 1 + n\ 2 + 1) * (n\ 2 + n\ 2 + 1)$ **unfolding** σ*-def*
by *simp*
also have $(-\ (1 :: 'a\ fract))\ \hat{}\ \ldots = (-\ 1)\ \hat{}\ (n\ 1 - n\ 2 + 1)$
by (*rule minus-1-even-eqI*, *insert n12*, *auto*)
also have $\ldots = (-\ 1)\hat{}(\delta\ 1 + 1)$ **unfolding** δ **by** (*simp add*: *numeral-2-eq-2*)
finally show $\gamma\ 3 = (-\ 1)\ \hat{}\ (\delta\ 1 + 1) * \beta\ 3$ **.**
qed

fun $h :: nat \Rightarrow 'a\ fract$ **where**
$h\ i = (if\ (i \leq 1)\ then\ 1\ else\ if\ i = 2\ then\ (f\ 2\ \hat{}\ \delta\ 1)\ else\ (f\ i\ \hat{}\ \delta\ (i - 1)\ /\ (h\ (i - 1)\ \hat{}\ (\delta\ (i - 1) - 1))))$

lemma *smult-inverse-sdiv-poly*: **assumes** *ffp*: $p \in range\ ffp$
and p: $p = smult\ (inverse\ x)\ q$
and p': $p' = sdiv\text{-}poly\ q'\ x'$
and xx: $x = ff\ x'$
and qq: $q = ffp\ q'$
shows $p = ffp\ p'$
proof (*rule poly-eqI*)
fix i
have $coeff\ p\ i = coeff\ q\ i\ /\ x$ **unfolding** p **by** (*simp add*: *field-simps*)
also have $\ldots = ff\ (coeff\ q'\ i)\ /\ ff\ x'$ **unfolding** $qq\ xx$ **by** *simp*
finally have *cpi*: $coeff\ p\ i = ff\ (coeff\ q'\ i)\ /\ ff\ x'$ **.**
from *ffp* **obtain** r **where** pr: $p = ffp\ r$ **by** *auto*

from *arg-cong*[*OF this*, *of* λ *p*. *coeff p i*, *unfolded cpi*]
have *ff* (*coeff q′ i*) / *ff x′* ∈ *range ff* **by** *auto*
hence *id*: *ff* (*coeff q′ i*) / *ff x′* = *ff* (*coeff q′ i div x′*)
by (*rule div-divide-to-fract*, *auto*)
show *coeff p i* = *coeff* (*ffp p′*) *i* **unfolding** *cpi id p′*
by (*simp add*: *sdiv-poly-def coeff-map-poly*)
qed

end

locale *subresultant-prs-locale2* = *subresultant-prs-locale F n δ f k β G1 G2* **for**
F :: *nat* ⇒ *′a* :: *idom-divide fract poly*
and *n* :: *nat* ⇒ *nat*
and *δ* :: *nat* ⇒ *nat*
and *f* :: *nat* ⇒ *′a fract*
and *k* :: *nat*
and *β* :: *nat* ⇒ *′a fract*
and *G1 G2* :: *′a poly* +
assumes *β3*: *β 3* = (*−1*)^(*δ 1* + *1*)
and *βi*: ⋀ *i*. *4* ≤ *i* ⟹ *i* ≤ *Suc k* ⟹ *β i* = (*−1*)^(*δ* (*i* − *2*) + *1*) ∗ *f* (*i* − *2*) ∗ *h* (*i* − *2*) ^ (*δ* (*i* − *2*))
begin

lemma *B-eq-17-main*: *2* ≤ *i* ⟹ *i* ≤ *k* ⟹
h i = (*−1*) ^ (*n 1* + *n i* + *i* + *1*) / *f i*
∗ (∏ *l*←[*3*..< *Suc* (*Suc i*)]. (*α l* / *β l*)) ∧ *h i* ≠ *0*
proof (*induct i rule*: *less-induct*)
case (*less i*)
from *less*(*2−*) **have** *fi0*: *f i* ≠ *0* **using** *f0*[*of i*] **by** *simp*
have *1*: (*− 1*) ≠ (*0* :: *′a fract*) **by** *simp*
show *?case* (**is** *h i* = *?r i* ∧ -)
proof (*cases i* = *2*)
case *True*
have *f20*: *f 2* ≠ *0* **using** *f20* **by** *auto*
have *hi*: *h i* = *f 2* ^ *δ 1* **unfolding** *True h.simps*[*of 2*] **by** *simp*
have *id*: *int* (*δ 1*) = *int* (*n 1*) − *int* (*n 2*) **using** *n12* **unfolding** *δ numeral-2-eq-2* **by** *simp*
have *?r i* = (*− 1*) ^ (*1* + *n 1* + *n 2*)
∗ ((*f 2* ^ *Suc* (*δ 1*)) / (*β 3*)) / *pow-int* (*f 2*) *1* **unfolding** *True α-def* **by** *simp*
also have *β 3* = (*− 1*) ^ (*δ 1* + *1*) **by** (*rule β3*)
also have *f 2* ^ *Suc* (*δ 1*) / ... = ... ∗ *f 2* ^ *Suc* (*δ 1*) **by** *simp*
finally have *?r i* = ((*− 1*) ^ (*1* + *n 1* + *n 2*) ∗ ((*− 1*) ^ (*δ 1* + *1*))) ∗
pow-int (*f 2*) (*int* (*Suc* (*δ 1*)) + (*−1*)) (**is** - = *?a* ∗ -)
unfolding *pow-int-divide exp-pow-int power-add pow-int-add*[*OF f20*] **by** (*simp add*: *ac-simps pow-int-add*)
also have *?a* = (*−1*)^(*1* + *n 1* + *n 2* + *δ 1* + *1*) **unfolding** *power-add* **by** *simp*
also have ... = (*−1*)^*0*
by (*rule minus-1-even-eqI*, *insert n12*, *auto simp*: *δ numeral-2-eq-2*, *presburger*)

finally have *ri*: *?r i* = *pow-int* (*f 2*) (*int* (*δ 1*)) **by** *simp*
show *?thesis* **unfolding** *ri hi exp-pow-int*[*symmetric*] **using** *f20* **by** *simp*
next
case *False*
hence *i*: $i \geq 3$ **and** *ii*: $i - 1 < i\ 2 \leq i - 1\ i - 1 \leq k$ **using** *less*(*2*−) **by** *auto*
from *i less*(*2*−) **have** *cc*: $4 \leq Suc\ i\ Suc\ i \leq Suc\ k$ **by** *auto*
define *P* **where** *P* = ($\prod$ *l*←[*3*..< *Suc i*]. *α l* / *β l*)
define *Q* **where** *Q* = *P* ∗ *pow-int* (*h* (*i* − *1*)) (− *int* (*δ* (*i* − *1*)))
define *R* **where** *R* = *f i* ^ *δ* (*i* − *1*)
define *S* **where** *S* = *pow-int* (*f* (*i* − *1*)) (− *1*)
note *IH* = *less*(*1*)[*OF ii*]
hence *hi0*: *h* (*i* − *1*) ≠ *0* **by** *auto*
have *hii*: *h i* = *f i* ^ *δ* (*i* − *1*) / *h* (*i* − *1*) ^ (*δ* (*i* − *1*) − *1*)
unfolding *h.simps*[*of i*] **using** *i* **by** *simp*
also have ... = *f i* ^ *δ* (*i* − *1*) ∗ *pow-int* (*h* (*i* − *1*)) (− *int* (*δ* (*i* − *1*) − *1*))
unfolding *exp-pow-int pow-int-divide* **by** *simp*
also have *int* (*δ* (*i* − *1*) − *1*) = *int* (*δ* (*i* − *1*)) − *1*
proof −
have *δ* (*i* − *1*) > *0* **unfolding** *δ*[*of i* − *1*] **using** *n-gt*[*OF ii*(*2*)] *less*(*2*−) **by** *auto*
thus *?thesis* **by** *simp*
qed
also have − (*int* (*δ* (*i* − *1*)) − *1*) = *1* + (− *int* (*δ* (*i* − *1*))) **by** *simp*
finally have *hi*: *h i* = (− *1*) ^ (*n 1* + *n* (*i* − *1*) + *i*) ∗ (*R* ∗ *Q* ∗ *S*)
unfolding *pow-int-add*[*OF hi0*] *P-def Q-def pow-int-divide*[*symmetric*] *R-def S-def* **using** *IH i* **by** (*simp add*: *ac-simps*)
from *i* **have** *id*: [*3*..<*Suc* (*Suc i*)] = [*3* ..< *Suc i*] @ [*Suc i*] **by** *simp*
have *?r i* = (− *1*) ^ (*n 1* + *n i* + *i* + *1*)
∗ *pow-int* (*f i*) (− *1*) ∗ *P* ∗ *α* (*Suc i*) / *β* (*Suc i*)
unfolding *pow-int-divide*[*symmetric*] *P-def id Fract-conv-to-fract* **by** *simp*
also have *β* (*Suc i*) = (− *1*) ^ (*δ* (*i* − *1*) + *1*) ∗ *f* (*i* − *1*) ∗ *h* (*i* − *1*) ^ *δ* (*i* − *1*)
using *βi*[*OF cc*] **by** *simp*
also have *α* (*Suc i*) = *f i* ^ *Suc* (*δ* (*i* − *1*)) **unfolding** *α-def* **by** *simp*
finally have *?r i* = (− *1*) ^ (*n 1* + *n i* + *i* + *1*) ∗ *pow-int* (*f i*) (− *1*) ∗ *P* ∗ (*f i* ^ *Suc* (*δ* (*i* − *1*))) /
(− *1*) ^ (*δ* (*i* − *1*) + *1*) ∗ *pow-int* (*f* (*i* − *1*)) (− *1*) / *h* (*i* − *1*) ^ *δ* (*i* − *1*)
(**is** - = *?a1* ∗ *?fi1* ∗ *P* ∗ *?fi2* / *?a2* ∗ *?b* / *?c*)
unfolding *exp-pow-int pow-int-divide*[*symmetric*] **by** *simp*
also have ... = (*?a1* / *?a2*) ∗ (*?fi1* ∗ *?fi2*) ∗ (*P* / *?c*) ∗ *?b* **by** (*simp add*: *ac-simps*)
also have *?a1* / *?a2* = (− *1*) ^ (*n 1* + *n i* + *i* + *1* + *δ* (*i* − *1*) + *1*)
by (*simp add*: *power-add*)
also have ... = (*−1*) ^ (*n 1* + *n i* + *i* + *δ* (*i* − *1*))
by (*rule minus-1-even-eqI*, *auto*)
also have *n 1* + *n i* + *i* + *δ* (*i* − *1*) = *n 1* + *n* (*i* − *1*) + *i*
unfolding *δ* **using** *i less*(*2*−) *n-ge*[*of i* − *1*] **by** *simp*
also have *?fi1* ∗ *?fi2* = *pow-int* (*f i*) (*−1* + *int* (*Suc* (*δ* (*i* − *1*))))
unfolding *exp-pow-int pow-int-add*[*OF fi0*] **by** *simp*

```
    also have ... = pow-int (f i) (int (δ (i − 1))) by simp
    also have P / ?c = Q  unfolding Q-def exp-pow-int pow-int-divide by simp
    also have ?b = S unfolding S-def by simp
    finally have ri: ?r i = (−1) ^ (n 1 + n (i − 1) + i)
      ∗ (R ∗ Q ∗ S) by (simp add: exp-pow-int R-def)
    have id: h i = ?r i unfolding hi ri ..
    show ?thesis
      by (rule conjI[OF id], unfold hii, insert IH fi0, auto)
  qed
qed

lemma B-eq-17: 2 ≤ i ⟹ i ≤ k ⟹
    h i = (−1) ^ (n 1 + n i + i + 1) / f i ∗ (∏ l←[3..< Suc (Suc i)]. (α l / β l))
  using B-eq-17-main by blast

lemma B-theorem-2: 3 ≤ i ⟹ i ≤ Suc k ⟹ γ i = 1
proof (induct i rule: less-induct)
  case (less i)
  show ?case
  proof (cases i = 3)
    case True
    show ?thesis unfolding True unfolding gamma-delta-beta-3 β3 by simp
  next
    case False
    with less(2−)
    have i: i ≥ 4 and ii: i − 1 < i 3 ≤ i − 1 i − 1 ≤ Suc k
      and iii: 4 ≤ i i ≤ Suc k
      and iv: 2 ≤ i − 2 i − 2 ≤ k by auto
    from less(1)[OF ii] have IH: γ (i − 1) = 1 .
    define L where L = [3..< i]
    have id: [3..<Suc (i − 1)] = L [3..<Suc i] = L @ [i] Suc (Suc (i − 2)) = i
      unfolding L-def using i by auto
    define B where B = (λ l. β l / α l)
    define A where A = (λ l. α l / β l)
    define Q where Q = (λ l. f (l − 1) ^ (δ (l − 2) + δ (l − 1)))
    define R where R = (λ i l. B l ^ (n (l − 1) − n (i − 1) + 1))
    define P where P = (λ i l. R i l ∗ Q l)
    have fi0: f (i − 1) ≠ 0 using f0[of i − 1] less(2−) by auto
    have fi0': f (i − 2) ≠ 0 using f0[of i − 2] less(2−) by auto
    {
      fix j
      assume j ∈ set L
      hence j ≥ 3 j < i unfolding L-def by auto
      with less(3) have j: j − 1 ≠ 0 j − 1 < k by auto
      hence Q: Q j ≠ 0 unfolding Q-def using f0[of j − 1] by auto
      from j α0 β0[of j] have 0: α j ≠ 0 β j ≠ 0 by auto
      hence B j ≠ 0 A j ≠ 0 unfolding B-def A-def by auto
      note Q this
    } note L0 = this
```

let *?exp* = $\delta\ (i - 2)$
have $\gamma\ i = \gamma\ i\ /\ \gamma\ (i - 1)$ **unfolding** *IH* **by** *simp*
also have $\ldots = (-1)\ \hat{}\sigma\ i * pow\text{-}int\ (f\ (i - 1))\ (1 - int\ (\delta\ (i - 1))) *$
$(\prod l{\leftarrow}L.\ P\ i\ l) * P\ i\ i\ /$
$((-1)\ \hat{}\sigma\ (i - 1) * pow\text{-}int\ (f\ (i - 2))\ (1 - int\ (\delta\ (i - 2))) *$
$(\prod l{\leftarrow}L.\ P\ (i - 1)\ l))$ (**is** - = *?a1* * *?f1* * *?L1* * *P i i* / (*?a2* * *?f2* * *?L2*))
unfolding γ*-def id P-def Q-def R-def B-def* **by** (*simp add*: *numeral-2-eq-2*)
also have $\ldots = (?a1 * ?a2) * (?f1 * P\ i\ i)\ /\ ?f2 * (?L1\ /\ ?L2)$ **unfolding** *divide-prod-assoc* **by** *simp*
also have *?a1* * *?a2* = $(-1)\hat{}(\sigma\ i + \sigma\ (i - 1))$ (**is** - = *?a*) **unfolding** *power-add* **by** *simp*
also have *?L1* / *?L2* = $(\prod l{\leftarrow}L.\ R\ i\ l)\ /\ (\prod l{\leftarrow}L.\ R\ (i - 1)\ l) * ((\prod l{\leftarrow}L.\ Q\ l)\ /\ (\prod l{\leftarrow}L.\ Q\ l))$
unfolding *P-def prod-list-multf divide-prod-assoc* **by** *simp*
also have $\ldots = (\prod l{\leftarrow}L.\ R\ i\ l)\ /\ (\prod l{\leftarrow}L.\ R\ (i - 1)\ l)$ (**is** - = *?L1* / *?L2*)
proof −
have $(\prod l{\leftarrow}L.\ Q\ l) \neq 0$ **unfolding** *prod-list-zero-iff* **using** *L0* **by** *auto*
thus *?thesis* **by** *simp*
qed
also have *?f1* * *P i i* = (*?f1* * *pow-int* (*f* (*i* − *1*)) (*int ?exp* + *int* (δ (*i* − *1*)))) * *R i i* **unfolding** *P-def Q-def*
exp-pow-int **by** *simp*
also have *?f1* * *pow-int* (*f* (*i* − *1*)) (*int ?exp* + δ (*i* − *1*)) = *pow-int* (*f* (*i* − *1*))
(*1* + *int ?exp*) (**is** - = *?f1*)
unfolding *pow-int-add*[*OF fi0*, *symmetric*] **by** *simp*
also have *R i i* = $\beta\ i\ /\ \alpha\ i$ **unfolding** *B-def R-def Fract-conv-to-fract* **by** *simp*
also have $\alpha\ i = f\ (i - 1)\ \hat{}$ *Suc ?exp* **unfolding** α*-def* **by** *simp*
also have $\beta\ i\ /\ \ldots = \beta\ i * pow\text{-}int\ (f\ (i - 1))\ (-1 - ?exp)$
(**is** - = *?β* * *?f12*)
unfolding *exp-pow-int pow-int-divide* **by** *simp*
finally have $\gamma\ i = (?a * (?f1 * ?f12)) *\ ?\beta\ /\ ?f2 * (?L1\ /\ ?L2)$
by *simp*
also have *?a* * (*?f1* * *?f12*) = *?a* **unfolding** *pow-int-add*[*OF fi0*, *symmetric*] **by** *simp*
also have *?L1* / *?L2* = *pow-int* $(\prod l{\leftarrow}L.\ A\ l)$ (− *?exp*)
proof −
have *id*: *i* − *1* − *1* = *i* − *2* **by** *simp*
have *set L* ⊆ {*l*. *3* ≤ *l* ∧ *l* ≤ *k* ∧ *l* < *i*} **unfolding** *L-def* **using** *less*(*3*) **by** *auto*
thus *?thesis* **unfolding** *R-def id*
proof (*induct L*)
case (*Cons l L*)
from *Cons*(*2*) **have** *l*: *3* ≤ *l l* ≤ *k l* < *i* **and** *L*: *set L* ⊆ {*l*. *3* ≤ *l* ∧ *l* ≤ *k* ∧ *l* < *i*} **by** *auto*
note *IH* = *Cons*(*1*)[*OF L*]
from *l* α*0* β*0*[*of l*] **have** *0*: $\alpha\ l \neq 0\ \beta\ l \neq 0$ **by** *auto*
hence *B0*: *B l* ≠ *0* **unfolding** *B-def* **by** *auto*
have $(\prod l{\leftarrow}l\ \#\ L.\ B\ l\ \hat{}(n\ (l - 1) - n\ (i - 1) + 1))\ /\ (\prod l{\leftarrow}l\ \#\ L.\ B\ l$

$\hat{}\,(n\ (l - 1) - n\ (i - 2) + 1))$
$= (B\ l\ \hat{}\,(n\ (l - 1) - n\ (i - 1) + 1) * (\prod l{\leftarrow}L.\ B\ l\ \hat{}\,(n\ (l - 1) - n\ (i - 1) + 1))) /$
$(B\ l\ \hat{}\,(n\ (l - 1) - n\ (i - 2) + 1) * (\prod l{\leftarrow}L.\ B\ l\ \hat{}\,(n\ (l - 1) - n\ (i - 2) + 1)))$
(**is** - = (*?l1* * *?L1*) / (*?l2* * *?L2*)) **by** *simp*
also have ... = (*?l1* / *?l2*) * (*?L1* / *?L2*) **by** *simp*
also have *?L1* / *?L2* = *pow-int* (*prod-list* (*map A L*)) (− *int* (*δ* (*i* − *2*))) **by** (*rule IH*)
also have *?l1* / *?l2* = *pow-int* (*B l*) (*int* (*n* (*l* − *1*) − *n* (*i* − *1*)) − *int* (*n* (*l* − *1*) − *n* (*i* − *2*))) **unfolding** *exp-pow-int pow-int-divide pow-int-add*[*OF B0*, *symmetric*]
by *simp*
also have *int* (*n* (*l* − *1*) − *n* (*i* − *1*)) − *int* (*n* (*l* − *1*) − *n* (*i* − *2*)) = *int* *?exp*
proof −
have *n* (*l* − *1*) ≥ *n* (*i* − *2*) *n* (*l* − *1*) ≥ *n* (*i* − *1*) *n* (*i* − *2*) ≥ *n* (*i* − *1*)
using *i l less*(*3*)
by (*intro n-ge-trans*, *auto*)+
hence *id*: *int* (*n* (*l* − *1*) − *n* (*i* − *1*)) = *int* (*n* (*l* − *1*)) − *int* (*n* (*i* − *1*))
int (*n* (*l* − *1*) − *n* (*i* − *2*)) = *int* (*n* (*l* − *1*)) − *int* (*n* (*i* − *2*))
int (*n* (*i* − *2*) − *n* (*i* − *1*)) = *int* (*n* (*i* − *2*)) − *int* (*n* (*i* − *1*))
by *simp-all*
have *id2*: *int* *?exp* = *int* (*n* (*i* − *2*) − *n* (*i* − *1*))
unfolding *δ* **using** *i* **by** (*cases i*; *cases i* − *1*, *auto*)
show *?thesis* **unfolding** *id2* **unfolding** *id* **by** *simp*
qed
also have *pow-int* (*B l*) ... = *pow-int* (*inverse* (*B l*)) (− ...) **unfolding** *pow-int-def*
by (*cases int* (*δ* (*i* − *2*)) *rule*: *linorder-cases*, *auto simp*: *field-simps*)
also have *inverse* (*B l*) = *A l* **unfolding** *B-def A-def* **by** *simp*
also have *pow-int* (*A l*) (− *int* *?exp*) * *pow-int* (*prod-list* (*map A L*)) (− *int* *?exp*)
= *pow-int* (*prod-list* (*map A* (*l* # *L*))) (− *int* *?exp*)
by (*simp add*: *pow-int-mult*)
finally show *?case* **.**
qed *simp*
qed
also have *β i* = (− *1*) $\hat{}$ (*?exp* + *1*) * *f* (*i* − *2*) * *h* (*i* − *2*) $\hat{}$ *?exp*
unfolding *βi*[*OF iii*] **..**
finally have *γ i* = (((− *1*) $\hat{}$ (*σ i* + *σ* (*i* − *1*)) * (− *1*) $\hat{}$ (*?exp* + *1*))) *
(*pow-int* (*f* (*i* − *2*)) *1* *
pow-int (*f* (*i* − *2*)) (*int* *?exp* − *1*)) *
h (*i* − *2*) $\hat{}$ *?exp* /
(∏ *l*←*L*. *A l*) $\hat{}$ *?exp* (**is** - = *?a* * *?f1* * *?H* / *?L*) **unfolding** *pow-int-divide exp-pow-int* **by** *simp*
also have *?f1* = *pow-int* (*f* (*i* − *2*)) (*int* *?exp*) (**is** - = *?f1*) **unfolding** *pow-int-add*[*OF fi0'*, *symmetric*]
by *simp*

```
  also have h (i - 2) = (- 1) ^ (n 1 + n (i - 2) + (i - 2) + 1) / f (i - 2) *
    (∏ l←L. A l) (is - = ?a2 / ?f2 * ?L) unfolding B-eq-17[OF iv] A-def id
L-def by simp
  also have ((- (1 :: 'a fract)) ^ (σ i + σ (i - 1)) * (- 1) ^ (?exp + 1)) =
   ((- 1) ^ (σ i + σ (i - 1) + ?exp + 1)) (is - = ?a1) by (simp add: power-add)
  finally have γ i = ?a1 * ?f1 * (?a2 / ?f2 * ?L) ^ ?exp / ?L ^ ?exp by simp
   also have ... = (?a1 * ?a2^?exp) * (?f1 / ?f2 ^ ?exp) * (?L^?exp / ?L ^
?exp)
    unfolding power-mult-distrib power-divide by auto
  also have  ?L ^ ?exp / ?L ^ ?exp = 1
  proof -
   have ?L ≠ 0 unfolding prod-list-zero-iff using L0 by auto
   thus ?thesis by simp
  qed
  also have ?f1 / ?f2 ^ ?exp = 1 unfolding exp-pow-int pow-int-divide
   pow-int-add[OF fi0', symmetric] by simp
  also have ?a2^?exp = (- 1) ^ ((n 1 + n (i - 2) + (i - 2) + 1) * ?exp)
   by (rule semiring-normalization-rules)
  also have ?a1 * ... = (- 1) ^ (σ i + σ (i - 1) + ?exp + 1 + (n 1 + n (i
- 2) + (i - 2) + 1) * ?exp)
    (is - = - ^ ?e)
   by (simp add: power-add)
  also have ... = (-1)^0
  proof -
   define e where e = ?e
   have *: ?e = (2 * ?exp + σ i + σ (i - 1) + 1 + (n 1 + n (i - 2) + (i -
2)) * ?exp) by simp
   define A where A = (λ i l. (n (l - 2) + n (i - 1) + 1) * (n (l - 1) + n
(i - 1) + 1))
   define B where B = (λ i. (n (i - 1) + 1) * (n (i - 1) + 1))
   define C where C = (λ l. (n (l - 1) + n (l - 2) + n (l - 1) * n (l - 2)))
   define D where D = (λ l. n (l - 1) + n (l - 2))
   define m2 where m2 = n (i - 2)
   define m1 where m1 = n (i - 1)
   define m0 where m0 = n 1
   define i3 where i3 = i - 3
   have m12: m2 ≥ m1 unfolding m2-def m1-def using n-ge[of i - 2] i less(3)
    by (cases i, auto)
   have idd: Suc (i - 2) = i - 1 i - 1 - 1 = i - 2 using i by auto
   have id4: i - 2 = Suc i3 unfolding i3-def using i by auto
   from i have 3 < i by auto
   hence ∃ k. sum-list (map D L) = n 1 + n (i - 2) + 2 * k unfolding L-def
   proof (induct i rule: less-induct)
    case (less i)
    show ?case
    proof (cases i = 4)
     case True
     thus ?thesis by (simp add: D-def)
    next
```

case *False*
obtain *ii* **where** *i*: $i = Suc\ ii$ **and** *ii*: $ii < i\ 3 < ii$ **using** *False less*(*2*) **by** (*cases i*, *auto*)
from *less*(*1*)[*OF ii*] **obtain** *k* **where** *IH*: $sum\text{-}list\ (map\ D\ [3\ ..<\ ii]) = n\ 1 + n\ (ii - 2) + 2 * k$ **by** *auto*
have $map\ D\ [3\ ..<\ i] = map\ D\ [3\ ..<\ ii] @ [D\ ii]$ **unfolding** *i* **using** *ii* **by** *auto*
hence $sum\text{-}list\ (map\ D\ [3..<i]) = n\ 1 + n\ (ii - 2) + 2 * k + D\ ii$ **using** *IH* **by** *simp*
also have $\ldots = n\ 1 + n\ (ii - 1) + 2 * (n\ (ii - 2) + k)$ **unfolding** *D-def* **by** *simp*
also have $n\ (ii - 1) = n\ (i - 2)$ **unfolding** *i* **by** *simp*
finally show *?thesis* **by** *blast*
qed
qed
then obtain *kk* **where** *DL*: $sum\text{-}list\ (map\ D\ L) = n\ 1 + n\ (i - 2) + 2 * kk$ **..**
let $?l = i - 3$
have *len*: $length\ L = i - 3$ **unfolding** *L-def* **using** *i* **by** *auto*
have *A*: $A\ i\ l = B\ i + D\ l * n\ (i - 1) + C\ l$ **for** *i l*
unfolding *A-def B-def C-def D-def ring-distribs* **by** *simp*
have *id2*: $[3..<Suc\ i] = 3\ \#\ [Suc\ 3\ ..<\ Suc\ i]$
unfolding *L-def* **using** *i* **by** (*auto simp*: *upt-rec*[*of 3*])
have $even\ e = even\ ?e$ **unfolding** *e-def* **by** *simp*
also have $\ldots = even\ ((1 + (n\ 1 + n\ (i - 2) + (i - 2)) * ?exp) + (\sigma\ i + \sigma\ (i - 1)))$
(**is** - = $even\ (?g + ?j)$)
unfolding $*$ **by** (*simp add*: *ac-simps*)
also have $?j = (\sum l{\leftarrow}L\ @\ [i].\ A\ i\ l) + (\sum l{\leftarrow}L.\ A\ (i - 1)\ l)$
unfolding σ*-def id A-def* **by** *simp*
also have $\ldots = 2 * (\sum l{\leftarrow}L.\ C\ l) + (Suc\ ?l) * B\ i + (\sum l{\leftarrow}L\ @\ [i].\ D\ l * n\ (i - 1)) + C\ i +$
$?l * B\ (i - 1) + (\sum l{\leftarrow}L.\ D\ l * n\ (i - 1 - 1))$
unfolding *A sum-list-addf* **by** (*simp add*: *sum-list-triv len*)
also have $\ldots = ((Suc\ ?l * B\ i + C\ i +$
$?l * B\ (i - 1) + D\ i * n\ (i - 1)) + ((\sum l{\leftarrow}L.\ D\ l) * (n\ (i - 1) + n\ (i - 2)) + 2 * (\sum l{\leftarrow}L.\ C\ l)))$
(**is** - = $?i + ?j$)
unfolding *sum-list-mult-const* **by** (*simp add*: *ring-distribs numeral-2-eq-2*)
also have $?j =$
$(n\ 1 + n\ (i - 2)) * (n\ (i - 1) + n\ (i - 2)) + 2 * (kk * (n\ (i - 1) + n\ (i - 2)) + (\sum l{\leftarrow}L.\ C\ l))$
(**is** - = $?h + 2 * ?f$)
unfolding *DL* **by** (*simp add*: *ring-distribs*)
finally have $even\ e = even\ (?g + ?i + ?h + 2 * ?f)$ **by** *presburger*
also have $\ldots = even\ (?g + ?i + ?h)$ **by** *presburger*
also have $?g + ?i + ?h =$
$i3 * (m2 - m1 + m1 * m1 + m2 * m2)$
$+ (m2 - m1 + m1 + m2) * (m0 + m2)$

```
        + (m1 + m2 + (m2 − m1))
          + 2 ∗ (m1 ∗ m2 + m1 ∗ m1 + 1 + i3 + m1 ∗ Suc i3 + m2 ∗ i3)
unfolding idd B-def D-def C-def δ
        m1-def[symmetric] m2-def[symmetric] m0-def[symmetric]
        unfolding i3-def[symmetric] id4
        by (simp add: ring-distribs)
      also have (m1 + m2 + (m2 − m1)) = 2 ∗ m2 using m12 by simp
     also have (m2 − m1 + m1 + m2) ∗ (m0 + m2) = 2 ∗ (m2 ∗ (m0 + m2))
using m12 by simp
      finally obtain l1 l2 l3 where
        even e = even (i3 ∗ (m2 − m1 + m1 ∗ m1 + m2 ∗ m2)  + 2 ∗ l1 + 2 ∗
l2 + 2 ∗ l3)
        by blast
      also have ... = even (i3 ∗ (m2 − m1 + m1 ∗ m1 + m2 ∗ m2)) by simp
     also have ... = even (i3 ∗ (2 ∗ m1 + (m2 − m1 + m1 ∗ m1 + m2 ∗ m2)))
by simp
      also have 2 ∗ m1 + (m2 − m1 + m1 ∗ m1 + m2 ∗ m2) = m1 + m2 +
m1 ∗ m1 + m2 ∗ m2
        using m12 by simp
      also have even (i3 ∗ ...) by auto
      finally have even e .
      thus ?thesis unfolding e-def
        by (intro minus-1-even-eqI, auto)
    qed
    finally show γ i = 1 by simp
  qed
qed

context
  fixes i :: nat
  assumes i: 3 ≤ i i ≤ k
begin
lemma B-theorem-3-b: Θ i ∗ f i = ff (lead-coeff (H i))
  using arg-cong[OF fundamental-theorem-eq-6[folded H-def, OF i], of lead-coeff]
unfolding f[of i]
  lead-coeff-smult by simp

lemma B-theorem-3-main: Θ i ∗ f i / γ (i + 1) = (−1)^(n 1 + n i + i + 1) /
f i ∗ (∏ l←[3..< Suc (Suc i)]. (α l / β l))
proof (cases f i = 0)
  case True
  thus ?thesis by simp
next
  case False note ff0 = this
  from i(1) have Suc (Suc i) > 3 by auto
  hence id: [3 ..< Suc (i + 1)] = [3 ..< Suc i] @ [Suc i] [3 ..< Suc (Suc i)] = [3
..< Suc i] @ [Suc i] by auto
  have cong: ⋀ a b c d. a = c ⟹ b = d ⟹ a ∗ b = c ∗ (d :: 'a fract) by auto
  define AB where AB = (λ l. β l / α l)
```

define *ABP* **where** *ABP* = (λ *l*. *AB l* ^ (*n* (*l* − *1*) − *n i*) ∗ *f* (*l* − *1*) ^ (δ (*l* − *2*) + δ (*l* − *1*)))
define *PR* **where** *PR* = (∏ *l*←[*3*..<*Suc i*]. *ABP l*)
define *PR2* **where** *PR2* = (∏ *l*←[*3*..<*Suc i*]. *AB l*)
from *F0*[*of i*]
have Θ *i* ∗ *f i* / γ (*i* + *1*) = (
((− *1*) ^ τ *i* ∗ (− *1*) ^ σ (*i* + *1*)) ∗ (*pow-int* (*f i*) (*int* (δ (*i* − *1*)) − *1*) ∗
PR ∗ *f i* /
pow-int (*f i*) (*1* − *int* (δ *i*)) / ((∏ *l*←[*3*..<*Suc i*]. *ABP l* ∗ *AB l*) ∗
AB (*Suc i*) ∗ *f i* ^ (δ (*i* − *1*) + δ *i*))))
unfolding *id prod-list.append map-append* Θ*-def* γ*-def divide-prod-assoc*
by (*simp add*: *field-simps power-add AB-def ABP-def PR-def*)
also have (− *1* :: *'a fract*) ^ τ *i* ∗ (− *1*) ^ σ (*i* + *1*) = (− *1*) ^ (τ *i* + σ (*i* + *1*))
unfolding *power-add* **by** (*auto simp*: *field-simps*)
also have ... = (− *1*) ^ (*n 1* + *n i* + *i* + *1*)
proof (*cases i* = *2*)
case *True*
show *?thesis* **unfolding** τ*-def* σ*-def True* **by** (*auto*, *rule minus-1-even-eqI*, *auto*)
next
case *False*
define *a* **where** *a* = (λ *l*. *n* (*l* − *2*) + *n i*)
define *b* **where** *b* = (λ *l*. *n* (*l* − *1*) + *n i*)
define *c* **where** *c* = (∑ *l*←[*3*..<*Suc i*]. (*a l* ∗ *b l* + *n i*))
define *d* **where** *d* = *c* + (∑ *l*←[*3*..<*i*]. *n* (*l* − *1*))
define *e* **where** *e* = (*n* (*i* − *1*) + *n i* + *1*) ∗ *n i*
have (τ *i* + σ (*i* + *1*)) =
((∑ *l*←[*3*..<*Suc i*]. (*a l* ∗ *b l*) + (*a l* + *1*) ∗ (*b l* + *1*))) + (*a* (*Suc i*) + *1*) ∗ (*b* (*Suc i*) + *1*)
unfolding σ*-def* τ*-def id a-def b-def sum-list-addf* **by** *simp*
also have (∑ *l*←[*3*..<*Suc i*]. (*a l* ∗ *b l*) + (*a l* + *1*) ∗ (*b l* + *1*)) =
(∑ *l*←[*3*..<*Suc i*]. *2* ∗ *a l* ∗ *b l* + (*a l* + *b l*) + *1*)
by (*rule arg-cong*, *rule map-cong*, *auto*)
also have ... = (∑ *l*←[*3*..<*Suc i*]. *2* ∗ (*a l* ∗ *b l* + *n i*) + (*n* (*l* − *1*) + *n* (*l* − *2*)) + *1*)
by (*simp add*: *field-simps a-def b-def*)
also have ... = *2* ∗ *c* + (∑ *l*←[*3*..<*Suc i*]. (*n* (*l* − *1*) + *n* (*l* − *2*))) + *length* [*3* ..< *Suc i*]
unfolding *sum-list-addf c-def sum-list-const-mult sum-list-triv* **by** *simp*
also have (∑ *l*←[*3*..<*Suc i*]. (*n* (*l* − *1*) + *n* (*l* − *2*)))
= (∑ *l*←[*3*..<*Suc i*]. *n* (*l* − *1*)) + (∑ *l*←[*3*..<*Suc i*]. *n* (*l* − *2*))
by (*simp add*: *sum-list-addf*)
also have (∑ *l*←[*3*..<*Suc i*]. *n* (*l* − *2*)) = (∑ *l*←*3* # [*4*..<*Suc i*]. *n* (*l* − *2*))
by (*rule arg-cong*, *rule map-cong*, *insert i False*, *auto simp*: *upt-rec*[*of 3*])
also have ... = *n 1* + (∑ *l*←[(*Suc 3*)..<*Suc i*]. *n* (*l* − *2*)) **by** *auto*
also have (∑ *l*←[(*Suc 3*)..<*Suc i*]. *n* (*l* − *2*)) = (∑ *l*←[*3*..<*i*]. *n* (*l* − *1*))
proof (*rule arg-cong*[*of - - sum-list*], *rule nth-equalityI*, *force*, *auto simp*: *nth-append*, *goal-cases*)

case (*1 j*)
hence *i − 2 = Suc* (*Suc j*) **by** *simp*
thus *?case* **by** *simp*
qed
also have (∑ *l*←[*3*..<*Suc i*]. *n* (*l − 1*)) = (∑ *l*←[*3*..<*i*] @ [*i*]. *n* (*l − 1*))
by (*rule arg-cong*, *rule map-cong*, *insert i False*, *auto*)
finally have *τ i* + *σ* (*i* + *1*) =
2 ∗ d + *n* (*i − 1*) + *n 1* + *length* [*3*..<*Suc i*] + (*a* (*Suc i*) + *1*) ∗ (*b* (*Suc i*) + *1*)
by (*simp add*: *d-def*)
also have *length* [*3* ..< *Suc i*] = *i − 2* **using** *i* **by** *auto*
also have (*a* (*Suc i*) + *1*) ∗ (*b* (*Suc i*) + *1*) = *2 ∗ e* + *n* (*i − 1*) + *n i* + *1* **unfolding** *a-def b-def e-def*
by *simp*
finally have *id*: *τ i* + *σ* (*i* + *1*) = *2* ∗ (*d* + *n* (*i − 1*) + *e*) + *n 1* + (*i − 2*) + *n i* + *1*
by *simp*
show *?thesis*
by (*rule minus-1-even-eqI*, *unfold id*, *insert i*, *auto*)
qed
also have (∏ *l*←[*3*..<*Suc i*]. *ABP l ∗ AB l*) = *PR ∗ PR2*
unfolding *PR-def prod-list-multf PR2-def* **by** *simp*
also have (*pow-int* (*f i*) (*int* (*δ* (*i − 1*)) − *1*) ∗ *PR ∗ f i* / *pow-int* (*f i*) (*1* − *int* (*δ i*))
/ (*PR ∗ PR2 ∗ AB* (*Suc i*) ∗ *f i* ^ (*δ* (*i − 1*) + *δ i*))) =
((*pow-int* (*f i*) (*int* (*δ* (*i − 1*)) − *1*) ∗ *pow-int* (*f i*) *1* ∗ *pow-int* (*f i*) (*int* (*δ i*) − *1*)
/ *pow-int* (*f i*) (*int* (*δ* (*i − 1*) + *δ i*)))) ∗ (*PR* / *PR* / (*PR2 ∗ AB* (*Suc i*)))
(**is** ... = *?x ∗ ?y*)
unfolding *exp-pow-int*[*symmetric*] **by** (*simp add*: *pow-int-divide ac-simps*)
also have *?x* = *pow-int* (*f i*) (*−1*)
unfolding *pow-int-divide pow-int-add*[*OF ff0*, *symmetric*] **by** *simp*
also have ... = *1* / (*f i*)
unfolding *pow-int-def* **by** *simp*
also have *PR* / *PR* = *1*
proof −
have *PR* ≠ *0* **unfolding** *PR-def prod-list-zero-iff set-map*
proof
assume *0* ∈ *ABP* ‘ *set* [*3* ..< *Suc i*]
then obtain *j* **where** *j*: *3* ≤ *j j* < *Suc i* **and** *0*: *ABP j* = *0* **by** *auto*
with *i* **have** *jk*: *j* ≤ *k* **and** *j1*: *j − 1* ≠ *0 j − 1* < *k* **by** *auto*
hence *1*: *α j* ≠ *0 f* (*j − 1*) ≠ *0* **using** *α0 f0* **by** *auto*
with *0* **have** *AB j* = *0* **unfolding** *ABP-def* **by** *simp*
from *this*[*unfolded AB-def*] *1*(*1*) *β0*[*of j*] **show** *False* **by** *auto*
qed
thus *?thesis* **by** *simp*
qed
also have *PR2 ∗ AB* (*Suc i*) = (∏ *l*←[*3*..<*Suc* (*Suc i*)]. *AB l*) **unfolding** *id PR2-def* **by** *auto*

```
  also have 1 / ... = inverse ... by (simp add: inverse-eq-divide)
  also have ... = (∏ l←[3..<Suc (Suc i)]. α l / β l) unfolding AB-def
    inverse-prod-list map-map o-def
    by (auto cong: map-cong)
  finally show ?thesis by simp
qed

lemma B-theorem-3: h i = Θ i * f i h i = ff (lead-coeff (H i))
proof −
  have Θ i * f i = Θ i * f i / γ (i + 1)
    using B-theorem-2[of i + 1] i by auto
  also have ... = (− 1) ^(n 1 + n i + i + 1) / f i *
    (∏ l←[3..<Suc (Suc i)]. α l / β l) by (rule B-theorem-3-main)
  also have ... = h i using B-eq-17[of i] i by simp
  finally show h i = Θ i * f i ..
  thus h i = ff (lead-coeff (H i)) using B-theorem-3-b by auto
qed
end

lemma h0: i ≤ k ⟹ h i ≠ 0
proof (induct i)
  case (Suc i)
  thus ?case unfolding h.simps[of Suc i] using f0 by (auto simp del: h.simps)
qed auto

lemma deg-G12: degree G1 ≥ degree G2 using n12
  unfolding n F1 F2 by auto

lemma R0: shows R 0 = [: resultant G1 G2 :]
proof(cases n 2 = 0)
  case True
  hence d:degree G2 = 0 unfolding n F2 by auto
  from degree0-coeffs[OF d] F2 F12 obtain a where
    G2: G2 = [:a:] and a: a ≠ 0 by auto
  have sdiv-poly [:a * a ^ degree G1:] a = [:a ^ degree G1:] using a
    unfolding sdiv-poly-def by auto
  note dp = this
  show ?thesis using G2 F12
    unfolding R-def δ n F1 F2 Suc-1 by (auto split:if-splits simp:mult.commute
dp)
next
  case False
  from False n12 have d:degree G2 ≠ 0 degree G2 ≤ degree G1 unfolding n F2
F1 by auto
  from False have R 0 = subresultant 0 G1 G2 unfolding R-def by simp
  also have ... = [: resultant G1 G2 :] unfolding subresultant-resultant by simp
  finally show ?thesis .
qed
```

context
  fixes *div-exp* :: *'a ⇒ 'a ⇒ nat ⇒ 'a*
  assumes *div-exp-sound*: *div-exp-sound div-exp*
begin

interpretation *div-exp-sound div-exp* **by** (*rule div-exp-sound*)

lemma *subresultant-prs-main*: **assumes** *subresultant-prs-main Gi-1 Gi hi-1* = (*Gk*, *hk*)
  and *F i* = *ffp Gi*
  and *F* (*i* − *1*) = *ffp Gi-1*
  and *h* (*i* − *1*) = *ff hi-1*
  and *i* ≥ *3 i* ≤ *k*
shows *F k* = *ffp Gk* ∧ *h k* = *ff hk* ∧ (∀ *j*. *i* ≤ *j* ⟶ *j* ≤ *k* ⟶ *F j* ∈ *range ffp* ∧ *β* (*Suc j*) ∈ *range ff*)
proof −
  obtain *m* **where** *m*: *m* = *k* − *i* **by** *auto*
  show *?thesis* **using** *m assms*
  proof (*induct m arbitrary*: *Gi-1 Gi hi-1 i rule*: *less-induct*)
    case (*less m Gi-1 Gi hi-1 i*)
    note *IH* = *less*(*1*)
    note *m* = *less*(*2*)
    note *res* = *less*(*3*)
    note *id* = *less*(*4*−*6*)
    note *i* = *less*(*7*−*8*)
    let *?pmod* = *pseudo-mod Gi-1 Gi*
    let *?ni* = *degree Gi*
    let *?ni-1* = *degree Gi-1*
    let *?gi* = *lead-coeff Gi*
    let *?gi-1* = *lead-coeff Gi-1*
    let *?d1* = *?ni-1* − *?ni*
    obtain *hi* **where** *hi*: *hi* = *div-exp ?gi hi-1 ?d1* **by** *auto*
    obtain *divisor* **where** *div*: *divisor* = (−*1*) ^ (*?d1* + *1*) ∗ *?gi-1* ∗ (*hi-1* ^ *?d1*) **by** *auto*
    obtain *G1-p1* **where** *G1-p1*: *G1-p1* = *sdiv-poly ?pmod divisor* **by** *auto*
    note *res* = *res*[*unfolded subresultant-prs-main.simps*[*of Gi-1*] *Let-def*, *folded hi*, *folded div*, *folded G1-p1*]
    have *h-i*: *h i* = *f i* ^ *δ* (*i* − *1*) / *h* (*i* − *1*) ^ (*δ* (*i* − *1*) − *1*) **unfolding** *h.simps*[*of i*] **using** *i* **by** *simp*
    have *hi-ff*: *h i* ∈ *range ff* **using** *B-theorem-3*[*OF - i*(*2*)] *i* **by** *auto*
    have *d1*: *δ* (*i* − *1*) = *?d1* **unfolding** *δ n* **using** *id*(*1*,*2*) **using** *i* **by** *simp*
    have *fi*: *f i* = *ff ?gi* **unfolding** *f id* **by** *simp*
    have *fi1*: *f* (*i* − *1*) = *ff ?gi-1* **unfolding** *f id* **by** *simp*
    have *eq′*: *h i* = *ff* (*lead-coeff Gi*) ^ *δ* (*i* − *1*) / *ff hi-1* ^ (*δ* (*i* − *1*) − *1*)
      unfolding *h-i fi id* **..**
    have *idh*: *h i* = *ff hi* **using** *hi-ff h-i fi id*
      unfolding *hi d1*[*symmetric*]
      by (*subst div-exp*[*of ?gi δ* (*i* − *1*) *hi-1*], *unfold eq′*[*symmetric*], *insert assms*, *blast*+)

have $\beta\ (Suc\ i) = (-\ 1)\ \hat{}(\delta\ (i - 1) + 1) * f\ (i - 1) * h\ (i - 1)\ \hat{}\delta\ (i - 1)$
using $\beta i[of\ Suc\ i]\ i$ **by** *auto*
also have $\ldots = ff\ ((-\ 1)\ \hat{}(\delta\ (i - 1) + 1) * lead\text{-}coeff\ Gi\text{-}1 * hi\text{-}1\ \hat{}\delta\ (i - 1))$
unfolding *id f* **by** (*simp add*: *hom-distribs*)
also have $\ldots \in range\ ff$ **by** *blast*
finally have *beta*: $\beta\ (Suc\ i) \in range\ ff$ **.**
have *pm*: *pseudo-mod* $(F\ (i - 1))\ (F\ i) = ffp\ ?pmod$
unfolding *to-fract-hom.pseudo-mod-hom*[*symmetric*] *id* **by** *simp*
have *eq*: $(?pmod = 0) = (i = k)$
using *pm i pmod*[*of Suc i*] *F0*[*of Suc i*] *i* $\beta 0$[*of Suc i*] **by** *auto*
show *?case*
proof (*cases* $i = k$)
case *True*
with *res eq* **have** *res*: $Gk = Gi\ hk = hi$ **by** *auto*
with *pmod*
have $F\ k = ffp\ Gk \wedge h\ k = ff\ hk$ **unfolding** *res idh*[*symmetric*] *id*[*symmetric*] *True* **by** *auto*
thus *?thesis* **using** *beta* **unfolding** *True* **by** *auto*
next
case *False*
with *res eq* **have** *res*:
subresultant-prs-main Gi G1-p1 hi = (*Gk*, *hk*) **by** *auto*
from *m False i* **have** *m*: $m - 1 < m\ m - 1 = k - Suc\ i$ **by** *auto*
have *si*: $Suc\ i - 1 = i$ **and** *ii*: $3 \leq Suc\ i\ Suc\ i \leq k$ **and** *iii*: $3 \leq Suc\ i\ Suc\ i \leq Suc\ k$
using *False i* **by** *auto*
have $*$: $(\forall j{\geq}Suc\ i.\ j \leq k \longrightarrow F\ j \in range\ ffp \wedge \beta\ (Suc\ j) \in range\ ff) = (\forall j{\geq}i.\ j \leq k \longrightarrow F\ j \in range\ ffp\ \wedge \beta\ (Suc\ j) \in range\ ff)$
by (*rule for-all-Suc*, *insert id*(*1*) *beta*, *auto*)
show *?thesis*
proof (*rule IH*[*OF m res*, *unfolded si*, *OF - id*(*1*) *idh ii*, *unfolded* $*$])
have *F-ffp*: $F\ (Suc\ i) \in range\ ffp$ **using** *fundamental-theorem-eq-4*[*OF ii*, *symmetric*] *B-theorem-2*[*OF iii*] **by** *auto*
from *pmod*[*OF iii*] **have** *smult* $(\beta\ (Suc\ i))\ (F\ (Suc\ i)) = pseudo\text{-}mod\ (F\ (i - 1))\ (F\ i)$
by *simp*
from *arg-cong*[*OF this*, *of* $\lambda\ x.\ smult\ (inverse\ (\beta\ (Suc\ i)))\ x$]
have *Fsi*: $F\ (Suc\ i) = smult\ (inverse\ (\beta\ (Suc\ i)))\ (pseudo\text{-}mod\ (F\ (i - 1))\ (F\ i))$
using $\beta 0$[*of Suc i*] **by** *auto*
show $F\ (Suc\ i) = ffp\ G1\text{-}p1$
proof (*rule smult-inverse-sdiv-poly*[*OF F-ffp Fsi G1-p1 - pm*])
from *i ii* **have** *iv*: $4 \leq Suc\ i\ Suc\ i \leq Suc\ k$ **by** *auto*
have $*$: $Suc\ i - 2 = i - 1$ **by** *auto*
show $\beta\ (Suc\ i) = ff\ divisor$ **unfolding** $\beta i[OF\ iv]\ div\ d1 * fi1$
using *id* **by** (*simp add*: *hom-distribs*)
qed
qed

```
    qed
  qed
qed

lemma subresultant-prs: assumes res: subresultant-prs G1 G2 = (Gk, hk)
  shows F k = ffp Gk ∧ h k = ff hk ∧ (i ≠ 0 ⟶ F i ∈ range ffp) ∧ (3 ≤ i ⟶
i ≤ Suc k ⟶ β i ∈ range ff)
proof −
  let ?pmod = pseudo-mod G1 G2
  have pm: pseudo-mod (F 1) (F 2) = ffp ?pmod
    unfolding to-fract-hom.pseudo-mod-hom[symmetric] F1 F2 by simp
  let ?g2 = lead-coeff G2
  let ?n2 = degree G2
  obtain d1 where d1: d1 = degree G1 − ?n2 by auto
  obtain h2 where h2: h2 = ?g2 ^ d1 by auto
  have (?pmod = 0) = (pseudo-mod (F 1) (F 2) = 0) using pm by auto
  also have ... = (k < 3) using k2 pmod[of 3] F0[of 3] β0[of 3] by auto
  finally have eq: ?pmod = 0 ⟷ k = 2 using k2 by linarith
  note res = res[unfolded subresultant-prs-def Let-def eq, folded d1, folded h2]
  have idh2: h 2 = ff h2 unfolding h2 d1 h.simps[of 2] δ n F1
    using F2 by (simp add: numeral-2-eq-2 f hom-distribs)
  have main: F k = ffp Gk ∧ h k = ff hk ∧ (i ≥ 3 ⟶ i ≤ k ⟶ F i ∈ range ffp
∧ β (Suc i) ∈ range ff) for i
  proof (cases k = 2)
    case True
    with res have Gk = G2 hk = h2 by auto
    thus ?thesis using True idh2 F2 by auto
  next
    case False
    hence (k = 2) = False by simp
    note res = res[unfolded this if-False]
    have F-2: F (3 − 1) = ffp G2 using F2 by simp
    have h2: h (3 − 1) = ff h2 using idh2 by simp
    have n2: degree G2 = n (3 − 1) unfolding n using F2 by simp
    from False k2 have k3: 3 ≤ k by auto
    have F k = ffp Gk ∧ h k = ff hk ∧ (∀ j≥3. j ≤ k ⟶ F j ∈ range ffp ∧ β (Suc
j) ∈ range ff)
    proof (rule subresultant-prs-main[OF res - F-2 h2 le-refl k3])
      let ?pow = (− 1) ^(δ 1 + 1) :: 'a fract
      from pmod[of 3] k3
      have smult (β 3) (F 3) = pseudo-mod (F 1) (F 2) by simp
      also have ... = pseudo-mod (ffp G1) (ffp G2) using F1 F2 by auto
    also have ... = ffp (pseudo-mod G1 G2) unfolding to-fract-hom.pseudo-mod-hom
by simp
      also have β 3 = (− 1) ^(δ 1 + 1) unfolding β3 by simp
     finally have smult ((− 1) ^(δ 1 + 1)) (F 3) = ffp (pseudo-mod G1 G2) by
simp
      also have smult ((− 1) ^(δ 1 + 1)) (F 3) = [: ?pow :] ∗ F 3
        by simp
```

```
    also have [: ?pow :] = (− 1) ^ (δ 1 + 1) by (unfold hom-distribs, simp)
    finally have (− 1) ^ (δ 1 + 1) ∗ F 3 = ffp (pseudo-mod G1 G2) by simp
    from arg-cong[OF this, of λ i. (− 1) ^ (δ 1 + 1) ∗ i]
    have F 3 = (− 1) ^ (δ 1 + 1) ∗ ffp (pseudo-mod G1 G2) by simp
     also have δ 1 = d1 unfolding δ n d1 using F1 F2 by (simp add: numeral-2-eq-2)
    finally show F3: F 3 = ffp ((− 1) ^ (d1 + 1) ∗ pseudo-mod G1 G2) by (simp add: hom-distribs)
   qed
   thus ?thesis by auto
  qed
  show ?thesis
  proof (intro conjI impI)
   assume i ≠ 0
   then consider (12) i = 1 ∨ i = 2 | (i3) i ≥ 3 ∧ i ≤ k | (ik) i > k by linarith
   thus F i ∈ range ffp
   proof cases
    case 12
    thus ?thesis using F1 F2 by auto
   next
    case i3
    thus ?thesis using main by auto
   next
    case ik
    hence F i = 0 using F0 by auto
    thus ?thesis by simp
   qed
  next
   assume 3 ≤ i and i ≤ Suc k
   then consider (3) i = 3 | (4) 3 ≤ i − 1 i − 1 ≤ k by linarith
   thus β i ∈ range ff
   proof (cases)
    case 3
     have β i = ff ((− 1) ^ (δ 1 + 1)) unfolding 3 β3 by (auto simp: hom-distribs)
    thus ?thesis by blast
   next
    case 4
    with main[of i − 1] show ?thesis by auto
   qed
  qed (insert main, auto)
qed

lemma resultant-impl-main: resultant-impl-main G1 G2 = resultant G1 G2
proof −
  from F0[of 2] F12(2) have k2: k ≥ 2 by auto
  obtain Gk hk where sub: subresultant-prs G1 G2 = (Gk, hk) by force
  from subresultant-prs[OF this] have ∗: F k = ffp Gk h k = ff hk by auto
  have resultant-impl-main G1 G2 = (if degree (F k) = 0 then hk else 0)
```

```
    unfolding resultant-impl-main-def sub split ∗ using F2 F12 by auto
  also have ... = resultant G1 G2
  proof (cases n k = 0)
    case False
   with fundamental-theorem-eq-7[of 0] show ?thesis unfolding n[of k] ∗ R0 by
auto
  next
    case True
    from H-def[of k, unfolded True] have R: R 0 = H k by simp
    show ?thesis
    proof (cases k = 2)
      case False
      with k2 have k3: k ≥ 3 by auto
      from B-theorem-3[OF k3] R0 R have h k = ff (resultant G1 G2) by simp
      from this[folded ∗] ∗ have hk = resultant G1 G2 by simp
      with True show ?thesis unfolding n by auto
    next
      case 2: True
      have id: (if degree (F k) = 0 then hk else 0) = hk using True unfolding n
by simp
      from F0[of 3, unfolded 2] have F 3 = 0 by simp
      with pmod[of 3, unfolded 2] β0[of 3] have pseudo-mod (F 1) (F 2) = 0 by
auto
    hence pm: pseudo-mod G1 G2 = 0 unfolding F1 F2 to-fract-hom.pseudo-mod-hom
by simp
      from subresultant-prs-def[of G1 G2, unfolded sub Let-def this]
      have id: Gk = G2 hk = lead-coeff G2 ^ (degree G1 − degree G2) by auto
      from F12 F1 F2 have G1 ≠ 0 G2 ≠ 0 by auto
      from resultant-pseudo-mod-0[OF pm deg-G12 this]
       have res: resultant G1 G2 = (if degree G2 = 0 then lead-coeff G2 ^ degree
G1 else 0)
        by simp
      from True[unfolded 2 n F2] have degree G2 = 0 by simp
      thus ?thesis unfolding res 2 F2 id by simp
    qed
  qed
  finally show ?thesis .
qed
end
end
```

At this point, we have soundness of the resultant-implementation, provided that we can instantiate the locale by constructing suitable values of F, b, h, etc. Now we show the existence of suitable locale parameters by constructively computing them.

```
context
  fixes G1 G2 :: 'a :: idom-divide poly
begin
```

private function *F* **and** *b* **and** *h* **where** *F i* = (*if i* = (*0* :: *nat*) *then 1*
else if i = *1 then map-poly to-fract G1 else if i* = *2 then map-poly to-fract G2*
else (*let G* = *pseudo-mod* (*F* (*i* − *2*)) (*F* (*i* − *1*))
in if F (*i* − *1*) = *0* ∨ *G* = *0 then 0 else smult* (*inverse* (*b i*)) *G*))
| *b i* = (*if i* ≤ *2 then 1 else*
if i = *3 then* (− *1*) ^(*degree* (*F 1*) − *degree* (*F 2*) + *1*)
else if F (*i* − *2*) = *0 then 1 else* (− *1*) ^(*degree* (*F* (*i* − *2*)) − *degree* (*F* (*i* − *1*)) + *1*) ∗ *lead-coeff* (*F* (*i* − *2*)) ∗
h (*i* − *2*) ^(*degree* (*F* (*i* − *2*)) − *degree* (*F* (*i* − *1*))))
| *h i* = (*if* (*i* ≤ *1*) *then 1 else if i* = *2 then* (*lead-coeff* (*F 2*) ^(*degree* (*F 1*) − *degree* (*F 2*))) *else*
if F i = *0 then 1 else* (*lead-coeff* (*F i*) ^(*degree* (*F* (*i* − *1*)) − *degree* (*F i*)) / (*h* (*i* − *1*) ^((*degree* (*F* (*i* − *1*)) − *degree* (*F i*)) − *1*))))
by *pat-completeness auto*
termination
proof
show *wf* (*measure* (*case-sum* (λ *fi*. *3* ∗ *fi* +*1*) (*case-sum* (λ *bi*. *3* ∗ *bi*) (λ *hi*. *3* ∗ *hi* + *2*)))) **by** *simp*
qed (*auto simp*: *termination-simp*)

declare *h.simps*[*simp del*] *b.simps*[*simp del*] *F.simps*[*simp del*]

private lemma *Fb0*: **assumes** *base*: *G1* ≠ *0 G2* ≠ *0*
shows (*F i* = *0* ⟶ *F* (*Suc i*) = *0*) ∧ *b i* ≠ *0* ∧ *h i* ≠ *0*
proof (*induct i rule*: *less-induct*)
case (*less i*)
note ∗ [*simp*] = *F.simps*[*of i*] *b.simps*[*of i*] *h.simps*[*of i*]
consider (*0*) *i* = *0* | (*1*) *i* = *1* | (*2*) *i* ≥ *2* **by** *linarith*
thus *?case*
proof *cases*
case *0*
show *?thesis* **unfolding** ∗ **unfolding** *0* **by** *simp*
next
case *1*
show *?thesis* **unfolding** ∗ **unfolding** *1* **using** *assms* **by** *simp*
next
case *2*
have *F*: *F i* = *0* ⟹ *F* (*Suc i*) = *0* **unfolding** *F.simps*[*of Suc i*] **using** *2* **by** *simp*
from *assms* **have** *F2*: *F 2* ≠ *0* **unfolding** *F.simps*[*of 2*] **by** *simp*
from *2* **have** *i* − *1* < *i i* − *2* < *i* **by** *auto*
note *IH* = *less*[*OF this*(*1*)] *less*[*OF this*(*2*)]
hence *b*: *b* (*i* − *1*) ≠ *0* **and** *h*: *h* (*i* − *1*) ≠ *0 h* (*i* − *2*) ≠ *0* **by** *auto*
from *h* **have** *hi*: *h i* ≠ *0* **unfolding** *h.simps*[*of i*] **using** *2 F2* **by** *auto*
have *bi*: *b i* ≠ *0* **unfolding** *b.simps*[*of i*] **using** *h*(*2*) **by** *auto*
show *?thesis* **using** *hi bi F* **by** *blast*
qed
qed

```
private definition k = (LEAST i. F (Suc i) = 0)

private lemma k-exists: ∃ i. F (Suc i) = 0
proof −
  obtain n i where i ≥ 3 length (coeffs (F (Suc i))) = n by blast
  thus ?thesis
  proof (induct n arbitrary: i rule: less-induct)
    case (less n i)
    let ?ii = Suc (Suc i)
    let ?i = Suc i
    from less(2) have i: ?i ≥ 3 by auto
    let ?mod = pseudo-mod (F (?ii − 2)) (F ?i)
    have Fi: F ?ii = (if F ?i = 0 ∨ ?mod = 0 then 0 else smult (inverse (b ?ii))
?mod)
      unfolding F.simps[of ?ii] using i by auto
    show ?case
    proof (cases F ?ii = 0)
      case False
       hence Fi: F ?ii = smult (inverse (b ?ii)) ?mod and mod: ?mod ≠ 0 and
Fi1: F ?i ≠ 0
        unfolding Fi by auto
      from pseudo-mod[OF Fi1, of F (?ii − 2)] mod have degree ?mod < degree
(F ?i) by simp
      hence deg: degree (F ?ii) < degree (F ?i) unfolding Fi by auto
    hence length (coeffs (F ?ii)) < length (coeffs (F ?i)) unfolding degree-eq-length-coeffs
by auto
      from less(1)[OF - i refl, folded less(3), OF this] show ?thesis by auto
    qed blast
  qed
qed

private lemma k: F (Suc k) = 0 i < k ⟹ F (Suc i) ≠ 0
proof −
  show F (Suc k) = 0 unfolding k-def using k-exists by (rule LeastI2-ex)
  assume i < k from not-less-Least[OF this[unfolded k-def]] show F (Suc i) ≠ 0
.
qed

lemma enter-subresultant-prs: assumes len: length (coeffs G1) ≥ length (coeffs
G2)
  and G2: G2 ≠ 0
shows ∃ F n d f k b. subresultant-prs-locale2 F n d f k b G1 G2
proof (intro exI)
  from G2 len have G1: G1 ≠ 0 by auto
  from len have deg-le: degree (F 2) ≤ degree (F 1)
    by (simp add: F.simps degree-eq-length-coeffs)
  from G2 G1 have F1: F 1 ≠ 0 and F2: F 2 ≠ 0 by (auto simp: F.simps)
  note Fb0 = Fb0[OF G1 G2]
  interpret s: subresultant-prs-locale F λ i. degree (F i) λ i. degree (F i) − degree
```

```
(F (Suc i))
    λ i. lead-coeff (F i) k b G1 G2
  proof (unfold-locales, rule refl, rule refl, rule refl, rule deg-le, rule F1, rule F2)
    from k(1) F1 have k0: k ≠ 0 by (cases k, auto)
    show Fk: (F i = 0) = (k < i) for i
    proof
      assume F i = 0 with k(2)[of i − 1]
      have ¬ (i − 1 < k) by (cases i, auto simp: F.simps)
      thus i > k using k0 by auto
    next
      assume i > k
      then obtain j l where i: i = j + l and j = Suc k and l = i − Suc k and
Fj: F j = 0 using k(1)
        by auto
      with F1 F2 k0 have j2: j ≥ 2 by auto
      show F i = 0 unfolding i
      proof (induct l)
        case (Suc l)
        thus ?case unfolding F.simps[of j + Suc l] using j2 by auto
      qed (auto simp: Fj)
    qed
    show b: b i ≠ 0 for i using Fb0 by blast
    show F 1 = map-poly to-fract G1 unfolding F.simps[of 1] by simp
    show F 2 = map-poly to-fract G2 unfolding F.simps[of 2] by simp
    fix i
    let ?mod = pseudo-mod (F (i − 2)) (F (i − 1))
    assume i: 3 ≤ i i ≤ Suc k
    from Fk[of i − 1] i have F (i − 1) ≠ 0 by auto
    with i have Fi: F i = (if ?mod = 0 then 0 else smult (inverse (b i)) ?mod)
unfolding F.simps[of i]
      Let-def by simp
    show smult (b i) (F i) = ?mod
    proof (cases ?mod = 0)
      case True
      thus ?thesis unfolding Fi by simp
    next
      case False
      with Fi have Fi: F i = smult (inverse (b i)) ?mod by simp
      from arg-cong[OF this, of smult (b i)] b[of i] show ?thesis by simp
    qed
  qed
  note s.h.simps[simp del]
  show subresultant-prs-locale2 F (λ i. degree (F i)) (λ i. degree (F i) − degree (F
(Suc i)))
    (λ i. lead-coeff (F i)) k b G1 G2
  proof
    show b 3 = (− 1) ^ (degree (F 1) − degree (F (Suc 1)) + 1) unfolding
b.simps numeral-2-eq-2 by simp
    fix i
```

```
    assume i: 4 ≤ i i ≤ Suc k
    with s.F0[of i − 2] have F (i − 2) ≠ 0 by auto
     hence bi: b i = (− 1) ^(degree (F (i − 2)) − degree (F (i − 1)) + 1) *
lead-coeff (F (i − 2)) *
                  h (i − 2) ^(degree (F (i − 2)) − degree (F (i − 1))) unfolding
b.simps
      using i by auto
    have i < k ⟹ s.h i = h i for i
    proof (induct i)
      case 0
      thus ?case by (simp add: h.simps s.h.simps)
    next
      case (Suc i)
      from Suc(2) s.F0[of Suc i] have F (Suc i) ≠ 0 by auto
       with Suc show ?case unfolding h.simps[of Suc i] s.h.simps[of Suc i] nu-
meral-2-eq-2 by simp
    qed
    hence sh: s.h (i − 2) = h (i − 2) using i by simp
    from i have *: Suc (i − 2) = i − 1 by auto
    show b i = (− 1) ^(degree (F (i − 2)) − degree (F (Suc (i − 2))) + 1) *
lead-coeff (F (i − 2)) *
        s.h (i − 2) ^(degree (F (i − 2)) − degree (F (Suc (i − 2))))
      unfolding sh bi * ..
  qed
qed
end
```

Now we obtain the soundness lemma outside the locale.

```
context div-exp-sound
begin

lemma resultant-impl-main: assumes len: length (coeffs G1) ≥ length (coeffs G2)
  shows resultant-impl-main G1 G2 = resultant G1 G2
proof (cases G2 = 0)
  case G2: False
  from enter-subresultant-prs[OF len G2] obtain F n d f k b
    where subresultant-prs-locale2 F n d f k b G1 G2 by auto
  interpret subresultant-prs-locale2 F n d f k b G1 G2 by fact
  show ?thesis by (rule resultant-impl-main, standard)
next
  case G2: True
  show ?thesis unfolding G2
    resultant-impl-main-def using resultant-const(2)[of G1 0] by simp
qed

theorem resultant-impl: resultant-impl = resultant
proof (intro ext)
  fix f g :: 'a poly
  show resultant-impl f g = resultant f g
```

proof (*cases length* (*coeffs f*) ≥ *length* (*coeffs g*))
case *True*
thus *?thesis* **unfolding** *resultant-impl-def resultant-impl-main*[*OF True*] **by** *auto*
next
case *False*
hence *length* (*coeffs g*) ≥ *length* (*coeffs f*) **by** *auto*
from *resultant-impl-main*[*OF this*]
show *?thesis* **unfolding** *resultant-impl-def resultant-swap*[*of f g*] **using** *False*
by (*auto simp*: *Let-def*)
qed
qed
end

7.3 Code Equations

In the following code-equations, we only compute the required values, e.g., h_k is not required if $n_k > 0$, we compute $(-1)^{\cdots} * \ldots$ via a case-analysis, and we perform special cases for $\delta_i = 1$, which is the most frequent case.

context *div-exp-param*
begin

partial-function(*tailrec*) *subresultant-prs-main-impl* **where**
subresultant-prs-main-impl f Gi-1 Gi ni-1 d1-1 hi-2 = (*let*
gi-1 = *lead-coeff Gi-1*;
ni = *degree Gi*;
hi-1 = (*if d1-1* = *1 then gi-1 else div-exp gi-1 hi-2 d1-1*);
d1 = *ni-1* − *ni*;
pmod = *pseudo-mod Gi-1 Gi*
in (*if pmod* = *0 then f* (*Gi*, (*if d1* = *1 then lead-coeff Gi*
else div-exp (*lead-coeff Gi*) *hi-1 d1*)) *else*
let
gi = *lead-coeff Gi*;
divisor = (−*1*) ^ (*d1* + *1*) ∗ *gi-1* ∗ (*hi-1* ^ *d1*) ;
Gi-p1 = *sdiv-poly pmod divisor*
in subresultant-prs-main-impl f Gi Gi-p1 ni d1 hi-1))

definition *subresultant-prs-impl* **where**
subresultant-prs-impl f G1 G2 = (*let*
pmod = *pseudo-mod G1 G2*;
n2 = *degree G2*;
delta-1 = (*degree G1* − *n2*);
g2 = *lead-coeff G2*;
h2 = *g2* ^ *delta-1*
in if pmod = *0 then f* (*G2*,*h2*) *else let*
G3 = (− *1*) ^ (*delta-1* + *1*) ∗ *pmod*;
g3 = *lead-coeff G3*;
n3 = *degree G3*;
d2 = *n2* − *n3*;

```
    pmod = pseudo-mod G2 G3
  in if pmod = 0 then f (G3, if d2 = 1 then g3 else div-exp g3 h2 d2)
   else let divisor = (− 1) ^ (d2 + 1) ∗ g2 ∗ h2 ^ d2; G4 = sdiv-poly pmod
divisor
      in subresultant-prs-main-impl f G3 G4 n3 d2 h2
  )
end

context div-exp-sound
begin

lemma div-exp-1: div-exp g h (Suc 0) = g
  using div-exp[of g Suc 0 h] by simp

lemma subresultant-prs-impl: subresultant-prs-impl f G1 G2 = f (subresultant-prs
G1 G2)
proof −
  define h2 where h2 = lead-coeff G2 ^ (degree G1 − degree G2)
  define G3 where G3 = ((− 1) ^ (degree G1 − degree G2 + 1) ∗ pseudo-mod
G1 G2)
  define G4 where G4 = sdiv-poly (pseudo-mod G2 G3)
      ((− 1) ^ (degree G2 − degree G3 + 1) ∗ lead-coeff G2 ∗
       h2 ^ (degree G2 − degree G3))
  define d2 where d2 = degree G2 − degree G3
  have dl1: (if d = 1 then (g :: 'a) else div-exp g h d) = div-exp g h d for d g h
    by (cases d = 1, auto simp: div-exp-1)
  show ?thesis
    unfolding subresultant-prs-impl-def subresultant-prs-def Let-def
      subresultant-prs-main.simps[of G2]
      if-distrib[of f] dl1
  proof (rule if-cong[OF refl refl if-cong[OF refl refl]], unfold h2-def[symmetric],
   unfold G3-def[symmetric], unfold G4-def[symmetric], unfold d2-def[symmetric])
   note simp = subresultant-prs-main-impl.simps[of f] subresultant-prs-main.simps
    show subresultant-prs-main-impl f G3 G4 (degree G3) d2 h2 =
      f (subresultant-prs-main G3 G4 (div-exp (lead-coeff G3) h2 d2))
   proof (induct G4 arbitrary: G3 d2 h2 rule: wf-induct[OF wf-measure[of degree]])
      case (1 G4 G3 d2 h2)
      let ?M = pseudo-mod G3 G4
      show ?case
      proof (cases ?M = 0)
        case True
        thus ?thesis unfolding simp[of G3] Let-def dl1 by simp
      next
        case False
        hence id: (?M = 0) = False by auto
        let ?c = ((− 1) ^ (degree G3 − degree G4 + 1) ∗ lead-coeff G3 ∗
            (div-exp (lead-coeff G3) h2 d2) ^ (degree G3 − degree G4))
        let ?N = sdiv-poly ?M ?c
        show ?thesis
```

```
      proof (cases G4 = 0)
        case G4: False
          have degree ?N ≤ degree ?M unfolding sdiv-poly-def by (rule de-
gree-map-poly-le)
         also have ... < degree G4 using pseudo-mod[OF G4, of G3] False by
auto
        finally show ?thesis unfolding simp[of G3] Let-def id if-False dl1
          by (intro 1(1)[rule-format], auto)
      next
        case 0: True
        with False have G3 ≠ 0 by auto
       show ?thesis unfolding 0 unfolding simp[of G3] Let-def unfolding dl1
simp[of 0] by simp
      qed
    qed
  qed
 qed
qed

definition
 resultant-impl-rec = subresultant-prs-main-impl (λ (Gk,hk). if degree Gk = 0 then
hk else 0)
definition
 resultant-impl-start = subresultant-prs-impl (λ (Gk,hk). if degree Gk = 0 then hk
else 0)

lemma resultant-impl-start-code:
 resultant-impl-start G1 G2 =
   (let pmod = pseudo-mod G1 G2;
        n2 = degree G2;
        n1 = degree G1;
        g2 = lead-coeff G2;
        d1 = n1 − n2
       in if pmod = 0 then if n2 = 0 then if d1 = 0 then 1 else if d1 = 1 then g2
else g2 ^ d1 else 0
          else let
               G3 = if even d1 then − pmod else pmod;
               n3 = degree G3;
               pmod = pseudo-mod G2 G3
               in if pmod = 0
                  then if n3 = 0 then
                   let d2 = n2 − n3;
                       g3 = lead-coeff G3
                      in (if d2 = 1 then g3 else
                          div-exp g3 (if d1 = 1 then g2 else g2 ^ d1) d2) else 0
                  else let
                         h2 = (if d1 = 1 then g2 else g2 ^ d1);
                         d2 = n2 − n3;
                         divisor = (if d2 = 1 then g2 ∗ h2 else if even d2 then − g2
```

```
* h2 ^ d2 else g2 * h2 ^ d2);
                        G4 = sdiv-poly pmod divisor
                       in resultant-impl-rec G3 G4 n3 d2 h2)
proof −
  obtain d1 where d1: degree G1 − degree G2 = d1 by auto
  have id1: (if even d1 then − pmod else pmod) = (−1)^(d1 + 1) * (pmod :: 'a
poly) for pmod by simp
  have id3: (if d2 = 1 then g2 * h2 else if even d2 then − g2 * h2 ^ d2 else g2 *
h2 ^ d2) =
    ((− 1) ^ (d2 + 1) * g2 * h2 ^ d2)
    for d2 and g2 h2 :: 'a by auto
  show ?thesis
   unfolding resultant-impl-start-def subresultant-prs-impl-def resultant-impl-rec-def[symmetric]
Let-def split
    unfolding d1
    unfolding id1
    unfolding id3
    by (rule if-cong[OF refl if-cong if-cong], auto simp: power2-eq-square)
qed

lemma resultant-impl-rec-code:
  resultant-impl-rec Gi-1 Gi ni-1 d1-1 hi-2 = (
    let ni = degree Gi;
        pmod = pseudo-mod Gi-1 Gi
     in
     if pmod = 0
        then if ni = 0
          then
            let
              d1 = ni-1 − ni;
              gi = lead-coeff Gi
            in if d1 = 1 then gi else
              let gi-1 = lead-coeff Gi-1;
                  hi-1 = (if d1-1 = 1 then gi-1 else div-exp gi-1 hi-2 d1-1) in
                div-exp gi hi-1 d1
          else 0
        else let
            d1 = ni-1 − ni;
            gi-1 = lead-coeff Gi-1;
            hi-1 = (if d1-1 = 1 then gi-1 else div-exp gi-1 hi-2 d1-1);
            divisor = if d1 = 1 then gi-1 * hi-1 else if even d1 then − gi-1 * hi-1 ^
d1 else gi-1 * hi-1 ^ d1;
            Gi-p1 = sdiv-poly pmod divisor
       in resultant-impl-rec Gi Gi-p1 ni d1 hi-1)
  unfolding resultant-impl-rec-def subresultant-prs-main-impl.simps[of - Gi-1] split
Let-def
  unfolding resultant-impl-rec-def[symmetric]
  by (rule if-cong[OF - if-cong -], auto)
```

lemma *resultant-impl-main-code*: *resultant-impl-main G1 G2 =*
(if G2 = 0 then if degree G1 = 0 then 1 else 0
else resultant-impl-start G1 G2)
unfolding *resultant-impl-main-def*
resultant-impl-start-def subresultant-prs-impl **by** *simp*

lemma *resultant-impl-code*: *resultant-impl f g =*
(if length (coeffs f) ≥ length (coeffs g) then resultant-impl-main f g
else let res = resultant-impl-main g f in
if even (degree f) ∨ even (degree g) then res else − res)
unfolding *resultant-impl-def resultant-impl-def* **..**

lemma *resultant-code*: *resultant = resultant-impl*
using *resultant-impl* **by** *fastforce*

lemmas *resultant-code-lemmas =*
resultant-impl-code
resultant-impl-main-code
resultant-impl-start-code
resultant-impl-rec-code
end

global-interpretation *div-exp-Lazard*: *div-exp-sound dichotomous-Lazard* :: *'a* :: *factorial-ring-gcd* ⇒ -
defines
resultant-impl-Lazard = div-exp-Lazard.resultant-impl **and**
resultant-impl-main-Lazard = div-exp-Lazard.resultant-impl-main **and**
resultant-impl-start-Lazard = div-exp-Lazard.resultant-impl-start **and**
resultant-impl-rec-Lazard = div-exp-Lazard.resultant-impl-rec
by (*rule dichotomous-Lazard*)

declare *div-exp-Lazard.resultant-code-lemmas*[*code*]

As default use Lazard-implementation, which implements resultants on factorial rings.

declare *div-exp-Lazard.resultant-code*[*code*]

We also provide a second implementation without Lazard's optimization, which works on integral domains.

global-interpretation *div-exp-basic*: *div-exp-sound basic-div-exp*
defines
resultant-impl-basic = div-exp-basic.resultant-impl **and**
resultant-impl-main-basic = div-exp-basic.resultant-impl-main **and**
resultant-impl-start-basic = div-exp-basic.resultant-impl-start **and**
resultant-impl-rec-basic = div-exp-basic.resultant-impl-rec
by (*rule basic-div-exp*)

declare *div-exp-basic.resultant-code-lemmas*[*code*]

thm *div-exp-basic.resultant-code*

end

8 Computing the Gcd via the subresultant PRS

This theory now formalizes how the subresultant PRS can be used to calculate the gcd of two polynomials. Moreover, it proves the connection between resultants and gcd, namely that the resultant is 0 iff the degree of the gcd is non-zero.

theory *Subresultant-Gcd*
imports
Subresultant
Polynomial-Factorization.Missing-Polynomial-Factorial
begin

8.1 Algorithm

locale *div-exp-sound-gcd* = *div-exp-sound div-exp* **for**
div-exp :: *'a* :: {*semiring-gcd-mult-normalize*,*factorial-ring-gcd*} ⇒ *'a* ⇒ *nat* ⇒ *'a*
begin
definition *gcd-impl-primitive* **where**
[*code del*]: *gcd-impl-primitive G1 G2* = *normalize* (*primitive-part* (*fst* (*subresultant-prs G1 G2*)))

definition *gcd-impl-main* **where**
[*code del*]: *gcd-impl-main G1 G2* = (*if G1* = *0 then 0 else if G2* = *0 then normalize G1 else*
smult (*gcd* (*content G1*) (*content G2*))
(*gcd-impl-primitive* (*primitive-part G1*) (*primitive-part G2*)))

definition *gcd-impl* **where**
gcd-impl f g = (*if length* (*coeffs f*) ≥ *length* (*coeffs g*) *then gcd-impl-main f g else gcd-impl-main g f*)

8.2 Soundness Proof for *gcd-impl* = *gcd*

end

locale *subresultant-prs-gcd* = *subresultant-prs-locale2 F n δ f k β G1 G2* **for**
F :: *nat* ⇒ *'a* :: {*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *fract poly*
and *n* :: *nat* ⇒ *nat*
and *δ* :: *nat* ⇒ *nat*
and *f* :: *nat* ⇒ *'a fract*
and *k* :: *nat*
and *β* :: *nat* ⇒ *'a fract*

and *G1 G2* :: *'a poly*
begin

The subresultant PRS computes the gcd up to a scalar multiple.

context
fixes *div-exp* :: *'a* ⇒ *'a* ⇒ *nat* ⇒ *'a*
assumes *div-exp-sound*: *div-exp-sound div-exp*
begin

interpretation *div-exp-sound-gcd div-exp*
using *div-exp-sound* **by** (*rule div-exp-sound-gcd.intro*)

lemma *subresultant-prs-gcd*: **assumes** *subresultant-prs G1 G2* = (*Gk*, *hk*)
shows ∃ *a b*. *a* ≠ *0* ∧ *b* ≠ *0* ∧ *smult a* (*gcd G1 G2*) = *smult b* (*normalize Gk*)
proof −
from *subresultant-prs*[*OF div-exp-sound assms*]
have *Fk*: *F k* = *ffp Gk* **and** ∀ *i*. ∃ *H*. *i* ≠ *0* ⟶ *F i* = *ffp H*
and ∀ *i*. ∃ *b*. *3* ≤ *i* ⟶ *i* ≤ *Suc k* ⟶ *β i* = *ff b* **by** *auto*
from *choice*[*OF this*(*2*)] *choice*[*OF this*(*3*)] **obtain** *H beta* **where**
FH: ⋀ *i*. *i* ≠ *0* ⟹ *F i* = *ffp* (*H i*) **and**
beta: ⋀ *i*. *3* ≤ *i* ⟹ *i* ≤ *Suc k* ⟹ *β i* = *ff* (*beta i*) **by** *auto*
from *Fk FH*[*OF k0*] *FH*[*of 1*] *FH*[*of 2*] *FH*[*of Suc k*] *F0*[*of Suc k*] *F1 F2*
have *border*: *H k* = *Gk H 1* = *G1 H 2* = *G2 H* (*Suc k*) = *0* **by** *auto*
have *i* ≠ *0* ⟹ *i* ≤ *k* ⟹ ∃ *a b*. *a* ≠ *0* ∧ *b* ≠ *0* ∧ *smult a* (*gcd G1 G2*) = *smult b* (*gcd* (*H i*) (*H* (*Suc i*))) **for** *i*
proof (*induct i rule*: *less-induct*)
case (*less i*)
from *less*(*3*) **have** *ik*: *i* ≤ *k* **.**
from *less*(*2*) **have** *i* = *1* ∨ *i* ≥ *2* **by** *auto*
thus *?case*
proof
assume *i* = *1*
thus *?thesis* **unfolding** *border*[*symmetric*] **by** (*intro exI*[*of - 1*], *auto simp*: *numeral-2-eq-2*)
next
assume *i2*: *i* ≥ *2*
with *ik* **have** *i* − *1* < *i i* − *1* ≠ *0* **and** *imk*: *i* − *1* ≤ *k* **by** *auto*
from *less*(*1*)[*OF this*] *i2*
obtain *a b* **where** *a*: *a* ≠ *0* **and** *b*: *b* ≠ *0* **and** *IH*: *smult a* (*gcd G1 G2*) = *smult b* (*gcd* (*H* (*i* − *1*)) (*H i*)) **by** *auto*
define *M* **where** *M* = *pseudo-mod* (*H* (*i* − *1*)) (*H i*)
define *c* **where** *c* = *β* (*Suc i*)
have *M*: *pseudo-mod* (*F* (*i* − *1*)) (*F i*) = *ffp M* **unfolding** *to-fract-hom.pseudo-mod-hom*[*symmetric*] *M-def*
using *i2 FH* **by** *auto*
have *c*: *c* ≠ *0* **using** *β0* **unfolding** *c-def* **.**
from *i2 ik* **have** *3*: *Suc i* ≥ *3 Suc i* ≤ *Suc k* **by** *auto*
from *pmod*[*OF 3*]

```
      have pm: smult c (F (Suc i)) = pseudo-mod (F (i − 1)) (F i) unfolding
c-def by simp
      from beta[OF 3, folded c-def] obtain d where cd: c = ff d by auto
      with c have d: d ≠ 0 by auto
      from pm[unfolded cd M] FH[of Suc i]
      have ffp (smult d (H (Suc i))) = ffp M by auto
      hence pm: smult d (H (Suc i)) = M by (rule map-poly-hom.injectivity)
      from ik F0[of i] i2 FH[of i] have Hi0: H i ≠ 0 by auto
      from pseudo-mod[OF this, of H (i − 1), folded M-def]
      obtain c Q where c: c ≠ 0 and smult c (H (i − 1)) = H i ∗ Q + M by
auto
     from this[folded pm] have smult c (H (i − 1)) = Q ∗ H i + smult d (H (Suc
i)) by simp
      from gcd-add-mult[of H i Q smult d (H (Suc i)), folded this]
      have gcd (H i) (smult c (H (i − 1))) = gcd (H i) (smult d (H (Suc i))) .
      with gcd-smult-ex[OF c, of H (i − 1) H i] obtain e where
        e: e ≠ 0 and gcd (H i) (smult d (H (Suc i))) = smult e (gcd (H i) (H (i
− 1)))
        unfolding gcd.commute[of H i] by auto
      with gcd-smult-ex[OF d, of H (Suc i) H i] obtain c where
        c: c ≠ 0 and smult c (gcd (H i) (H (Suc i))) = smult e (gcd (H (i − 1))
(H i))
        unfolding gcd.commute[of H i] by auto
      from arg-cong[OF this(2), of smult b] arg-cong[OF IH, of smult e]
       have smult (e ∗ a) (gcd G1 G2) = smult (b ∗ c) (gcd (H i) (H (Suc i)))
unfolding smult-smult
        by (simp add: ac-simps)
      moreover have e ∗ a ≠ 0 b ∗ c ≠ 0 using a b c e by auto
      ultimately show ?thesis by blast
    qed
  qed
  from this[OF k0 le-refl, unfolded border]
  obtain a b where a ≠ 0 b ≠ 0 and smult a (gcd G1 G2) = smult b (normalize
Gk) by auto
  thus ?thesis by auto
qed


lemma gcd-impl-primitive: assumes primitive-part G1 = G1 and primitive-part
G2 = G2
shows gcd-impl-primitive G1 G2 = gcd G1 G2
proof −
  let ?pp = primitive-part
  let ?c = content
  let ?n = normalize
  from F2 F0[of 2] k2 have G2: G2 ≠ 0 by auto
  obtain Gk hk where sub: subresultant-prs G1 G2 = (Gk, hk) by force
  have impl: gcd-impl-primitive G1 G2 = ?n (?pp Gk) unfolding gcd-impl-primitive-def
sub by auto
```

```
  from subresultant-prs-gcd[OF sub]
  obtain a b where a: a ≠ 0 and b: b ≠ 0 and id: smult a (gcd G1 G2) = smult
b (?n Gk)
    by auto
  define c where c = unit-factor (gcd G1 G2)
  define d where d = smult (unit-factor a) c
  from G2 have c: is-unit c unfolding c-def by auto
   from arg-cong[OF id, of ?pp, unfolded primitive-part-smult primitive-part-gcd
assms
     primitive-part-normalize c-def[symmetric]]
  have id: d * gcd G1 G2 = smult (unit-factor b) (?n (?pp Gk)) unfolding d-def
by simp
  have d: is-unit d unfolding d-def using c a
    by (simp add: is-unit-smult-iff)
  from is-unitE[OF d]
  obtain e where e: is-unit e and de: d * e = 1 by metis
  define a where a = smult (unit-factor b) e
  from arg-cong[OF id, of λ x. e * x]
  have (d * e) * gcd G1 G2 = a * (?n (?pp Gk)) by (simp add: ac-simps a-def)
  hence id: gcd G1 G2 = a * (?n (?pp Gk)) using de by simp
  have a: is-unit a unfolding a-def using b e
    by (simp add: is-unit-smult-iff)
  define b where b = unit-factor (?pp Gk)
  have Gk ≠ 0 using subresultant-prs[OF div-exp-sound sub] F0[OF k0] by auto
  hence b: is-unit b unfolding b-def by auto
  from is-unitE[OF b]
  obtain c where c: is-unit c and bc: b * c = 1 by metis
  obtain d where d: is-unit d and dac: d = a * c using c a by auto
  have gcd G1 G2 = d * (b * ?n (?pp Gk))
    unfolding id dac using bc by (simp add: ac-simps)
  also have b * ?n (?pp Gk) = ?pp Gk unfolding b-def by simp
  finally have gcd G1 G2 = d * ?pp Gk by simp
  from arg-cong[OF this, of ?n]
  have gcd G1 G2 = ?n (d * ?pp Gk) by simp
  also have ... = ?n (?pp Gk) using d
    unfolding normalize-mult by (simp add: is-unit-normalize)
  finally show ?thesis unfolding impl ..
qed
end
end

context div-exp-sound-gcd
begin

lemma gcd-impl-main: assumes len: length (coeffs G1) ≥ length (coeffs G2)
  shows gcd-impl-main G1 G2 = gcd G1 G2
proof (cases G1 = 0)
  case G1: False
  show ?thesis
```

```
  proof (cases G2 = 0)
    case G2: False
    let ?pp = primitive-part
    from G2 have G2: ?pp G2 ≠ 0 and id: (G2 = 0) = False by auto
     from len have len: length (coeffs (?pp G1)) ≥ length (coeffs (?pp G2)) by
simp
    from enter-subresultant-prs[OF len G2] obtain F n d f k b
      where subresultant-prs-locale2 F n d f k b (?pp G1) (?pp G2) by auto
    interpret subresultant-prs-locale2 F n d f k b ?pp G1 ?pp G2 by fact
    interpret subresultant-prs-gcd F n d f k b ?pp G1 ?pp G2 ..
   show ?thesis unfolding gcd-impl-main-def gcd-poly-decompose[of G1] id if-False
using G1
      by (subst gcd-impl-primitive, auto intro: div-exp-sound-axioms)
  next
    case True
    thus ?thesis unfolding gcd-impl-main-def by simp
  qed
next
  case True
  with len have G2 = 0 by auto
  thus ?thesis using True unfolding gcd-impl-main-def by simp
qed


theorem gcd-impl[simp]: gcd-impl = gcd
proof (intro ext)
  fix f g :: 'a poly
  show gcd-impl f g = gcd f g
  proof (cases length (coeffs f) ≥ length (coeffs g))
    case True
    thus ?thesis unfolding gcd-impl-def gcd-impl-main[OF True] by auto
  next
    case False
    hence length (coeffs g) ≥ length (coeffs f) by auto
    from gcd-impl-main[OF this]
    show ?thesis unfolding gcd-impl-def gcd.commute[of f g] using False by auto
  qed
qed
```

The implementation also reveals an important connection between resultant and gcd.

```
lemma resultant-0-gcd: resultant (f :: 'a poly) g = 0 ⟷ degree (gcd f g) ≠ 0
proof −
  {
    fix f g :: 'a poly
    assume len: length (coeffs f) ≥ length (coeffs g)
    {
      assume g: g ≠ 0
      with len have f: f ≠ 0 by auto
```

```
      let ?f = primitive-part f
      let ?g = primitive-part g
      let ?c = content
      from len have len: length (coeffs ?f) ≥ length (coeffs ?g) by simp
      obtain Gk hk where sub: subresultant-prs ?f ?g = (Gk,hk) by force
      have cf: ?c f ≠ 0 and cg: ?c g ≠ 0 using f g by auto
      {
        from g have ?g ≠ 0 by auto
        from enter-subresultant-prs[OF len this] obtain F n d f k b
          where subresultant-prs-locale2 F n d f k b ?f ?g by auto
        interpret subresultant-prs-locale2 F n d f k b ?f ?g by fact
         from subresultant-prs[OF div-exp-sound-axioms sub] have h k = ff hk by
auto
        with h0[OF le-refl] have hk ≠ 0 by auto
      } note hk0 = this
      have resultant f g = 0 ⟷ resultant (smult (?c f) ?f) (smult (?c g) ?g) = 0
by simp
      also have ... ⟷ resultant ?f ?g = 0 unfolding resultant-smult-left[OF cf]
resultant-smult-right[OF cg]
        using cf cg by auto
      also have ... ⟷ resultant-impl-main ?f ?g = 0
      unfolding resultant-impl[symmetric] resultant-impl-def resultant-impl-main-def

        using len by auto
      also have ... ⟷ (degree Gk ≠ 0)
        unfolding resultant-impl-main-def sub split using g hk0 by auto
      also have degree Gk = degree (gcd-impl-primitive ?f ?g)
        unfolding gcd-impl-primitive-def sub by simp
      also have ... = degree (gcd-impl-main f g)
        unfolding gcd-impl-main-def using f g by auto
      also have ... = degree (gcd f g) unfolding gcd-impl[symmetric] gcd-impl-def
using len by auto
      finally have (resultant f g = 0) = (degree (gcd f g) ≠ 0) .
    }
    moreover
    {
      assume g: g = 0 and f: degree f ≠ 0
      have (resultant f g = 0) = (degree (gcd f g) ≠ 0)
        unfolding g using f by auto
    }
    moreover
    {
      assume g: g = 0 and f: degree f = 0
      have (resultant f g = 0) = (degree (gcd f g) ≠ 0)
      unfolding g using f by (auto simp: resultant-def sylvester-mat-def sylvester-mat-sub-def)
    }
    ultimately have (resultant f g = 0) = (degree (gcd f g) ≠ 0) by blast
  } note main = this
  show ?thesis
```

```
  proof (cases length (coeffs f) ≥ length (coeffs g))
    case True
    from main[OF True] show ?thesis .
  next
    case False
    hence length (coeffs g) ≥ length (coeffs f) by auto
    from main[OF this] show ?thesis
      unfolding gcd.commute[of g f] resultant-swap[of g f] by (simp split: if-splits)
  qed
qed
```

8.3 Code Equations

```
definition gcd-impl-rec = subresultant-prs-main-impl fst
definition gcd-impl-start = subresultant-prs-impl fst

lemma gcd-impl-rec-code:
  gcd-impl-rec Gi-1 Gi ni-1 d1-1 hi-2 = (
    let pmod = pseudo-mod Gi-1 Gi
     in
     if pmod = 0 then Gi
        else let
           ni = degree Gi;
           d1 = ni-1 − ni;
           gi-1 = lead-coeff Gi-1;
           hi-1 = (if d1-1 = 1 then gi-1 else div-exp gi-1 hi-2 d1-1);
           divisor = if d1 = 1 then gi-1 ∗ hi-1 else if even d1 then − gi-1 ∗ hi-1 ^
d1 else gi-1 ∗ hi-1 ^ d1;
           Gi-p1 = sdiv-poly pmod divisor
       in gcd-impl-rec Gi Gi-p1 ni d1 hi-1)
 unfolding gcd-impl-rec-def subresultant-prs-main-impl.simps[of - Gi-1] split Let-def
 unfolding gcd-impl-rec-def[symmetric]
 by (rule if-cong, auto)

lemma gcd-impl-start-code:
  gcd-impl-start G1 G2 =
     (let pmod = pseudo-mod G1 G2
        in if pmod = 0 then G2
           else let
                n2 = degree G2;
                n1 = degree G1;
                d1 = n1 − n2;
                G3 = if even d1 then − pmod else pmod;
                pmod = pseudo-mod G2 G3
                in if pmod = 0
                   then G3
                   else let
                          g2 = lead-coeff G2;
                          n3 = degree G3;
```

```
                    h2 = (if d1 = 1 then g2 else g2 ^ d1);
                    d2 = n2 − n3;
                    divisor = (if d2 = 1 then g2 ∗ h2 else if even d2 then − g2
∗ h2 ^ d2 else g2 ∗ h2 ^ d2);
                    G4 = sdiv-poly pmod divisor
                  in gcd-impl-rec G3 G4 n3 d2 h2)
proof −
  obtain d1 where d1: degree G1 − degree G2 = d1 by auto
  have id1: (if even d1 then − pmod else pmod) = (−1)^(d1 + 1) ∗ (pmod :: 'a
poly) for pmod by simp
  show ?thesis
   unfolding gcd-impl-start-def subresultant-prs-impl-def gcd-impl-rec-def[symmetric]
Let-def split
    unfolding d1
    unfolding id1
    by (rule if-cong, auto)
qed

lemma gcd-impl-main-code:
  gcd-impl-main G1 G2 = (if G1 = 0 then 0 else if G2 = 0 then normalize G1 else
    let c1 = content G1;
      c2 = content G2;
      p1 = map-poly (λ x. x div c1) G1;
      p2 = map-poly (λ x. x div c2) G2
     in smult (gcd c1 c2) (normalize (primitive-part (gcd-impl-start p1 p2))))
 unfolding gcd-impl-main-def Let-def primitive-part-def gcd-impl-start-def gcd-impl-primitive-def
    subresultant-prs-impl by simp

lemmas gcd-code-lemmas =
  gcd-impl-main-code
  gcd-impl-start-code
  gcd-impl-rec-code
  gcd-impl-def

corollary gcd-via-subresultant: gcd = gcd-impl by simp
end

global-interpretation div-exp-Lazard-gcd: div-exp-sound-gcd dichotomous-Lazard
:: 'a :: {semiring-gcd-mult-normalize,factorial-ring-gcd} ⇒ -
  defines
    gcd-impl-Lazard = div-exp-Lazard-gcd.gcd-impl and
    gcd-impl-main-Lazard = div-exp-Lazard-gcd.gcd-impl-main and
    gcd-impl-start-Lazard = div-exp-Lazard-gcd.gcd-impl-start and
    gcd-impl-rec-Lazard = div-exp-Lazard-gcd.gcd-impl-rec
  by (simp add: Subresultant.dichotomous-Lazard div-exp-sound-gcd-def)

declare div-exp-Lazard-gcd.gcd-code-lemmas[code]

lemmas resultant-0-gcd = div-exp-Lazard-gcd.resultant-0-gcd
```

thm *div-exp-Lazard-gcd.gcd-via-subresultant*

Note that we did not activate *gcd = gcd-impl-Lazard* as code-equation, since according to our experiments, the subresultant-gcd algorithm is not always more efficient than the currently active equation. In particular, on *int poly gcd-impl-Lazard* performs worse, but on multi-variate polynomials, e.g., *int poly poly poly*, *gcd-impl-Lazard* is preferable.

end

References

[1] W. S. Brown. The subresultant PRS algorithm. *ACM Trans. Math. Softw.*, 4(3):237–249, 1978.

[2] W. S. Brown and J. F. Traub. On Euclid's algorithm and the theory of subresultants. *Journal of the ACM*, 18(4):505–514, 1971.

[3] L. Ducos. Optimizations of the subresultant algorithm. *Journal of Pure and Applied Algebra*, 145:149–163, 2000.

[4] A. Mahboubi. Proving formally the implementation of an efficient gcd algorithm for polynomials. In *Proc. IJCAR'06*, volume 4130 of *LNCS*, pages 438–452, 2006.

[5] R. Thiemann and A. Yamada. Algebraic numbers in Isabelle/HOL. In *Proc. ITP'16*, volume 9807 of *LNCS*, pages 391–408, 2016.