Subresultants*

Sebastiaan Joosten, René Thiemann and Akihisa Yamada

March 19, 2025

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*
 ⟨*proof*⟩

lemma *sylvester-mat-sub-carrier*:
 shows *sylvester-mat-sub m n p q* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) ⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma *sylvester-carrier-mat*:
 fixes *p q*
 defines *d* ≡ *degree p* + *degree q*
 shows *sylvester-mat p q* ∈ *carrier-mat d d* ⟨*proof*⟩

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*)
 ⟨*proof*⟩

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*)
 ⟨*proof*⟩

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 $sylvester\text{-}mat\ p\ q\ \$\$\ (i,j) =$
$(if\ i < n\ then\ coeff\ (monom\ 1\ (n - i) * p)\ (m+n-j)$
$else\ coeff\ (monom\ 1\ (m + n - i) * q)\ (m+n-j))$
⟨*proof*⟩

lemma *sylvester-mat-sub-0*[*simp*]: $sylvester\text{-}mat\text{-}sub\ 0\ n\ 0\ q = 0_m\ n\ n$
⟨*proof*⟩

lemma *sylvester-mat-0*[*simp*]: $sylvester\text{-}mat\ 0\ q = 0_m\ (degree\ q)\ (degree\ q)$
⟨*proof*⟩

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 $sylvester\text{-}mat\ p\ [:a:] = a \cdot_m 1_m\ (degree\ p)$
⟨*proof*⟩

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*⟩

definition $resultant :: {}'a\ poly \Rightarrow {}'a\ poly \Rightarrow {}'a :: comm\text{-}ring\text{-}1$ **where**
$resultant\ p\ q = det\ (sylvester\text{-}mat\ p\ q)$

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

definition $resultant\text{-}sub\ m\ n\ p\ q = det\ (sylvester\text{-}mat\text{-}sub\ m\ n\ p\ q)$

lemma *resultant-sub*: $resultant\ p\ q = resultant\text{-}sub\ (degree\ p)\ (degree\ q)\ p\ q$
⟨*proof*⟩

lemma *resultant-const*[*simp*]:
fixes $a :: {}'a :: comm\text{-}ring\text{-}1$
shows $resultant\ [:a:]\ q = a\ \widehat{}\ (degree\ q)$
and $resultant\ p\ [:a:] = a\ \widehat{}\ (degree\ p)$
⟨*proof*⟩

lemma *resultant-1*[*simp*]:
fixes $p :: {}'a :: comm\text{-}ring\text{-}1\ poly$
shows $resultant\ 1\ p = 1\ resultant\ p\ 1 = 1$
⟨*proof*⟩

lemma *resultant-0*[*simp*]:

fixes *p* :: *'a* :: *comm-ring-1 poly*
assumes *degree p* > *0*
shows *resultant 0 p* = *0 resultant p 0* = *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma (**in** *inj-comm-ring-hom*) *resultant-hom*: *resultant* (*map-poly hom p*) (*map-poly hom q*) = *hom* (*resultant p q*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *range-to-fract-dvd-iff*: **assumes** *b*: *b* ≠ *0*
shows *Fract a b* ∈ *range to-fract* ⟷ *b dvd a*
⟨*proof*⟩

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*⟩

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*⟩

lemma *div-divide-to-fract*: **assumes** *x* ∈ *range to-fract*
and *x* = (*y* :: *'a* :: *idom-divide fract*) / *z*
and *x'* = *y' div z'*

and $y = \textit{to-fract } y'$ $z = \textit{to-fract } z'$
shows $x = \textit{to-fract } x'$
⟨*proof*⟩

declare *Euclidean-Rings.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-Rings.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*)

⟨*proof*⟩

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*⟩

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-Rings.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-Rings.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*⟩

lemma *binary-power-code-unfold*[*code-unfold*]: (^) = *binary-power*
⟨*proof*⟩

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* == λ*a*. [: *a* :]

interpretation *coeff-lift-hom*: *inj-comm-monoid-add-hom coeff-lift* ⟨*proof*⟩
interpretation *coeff-lift-hom*: *inj-ab-group-add-hom coeff-lift*⟨*proof*⟩
interpretation *coeff-lift-hom*: *inj-comm-semiring-hom coeff-lift*
⟨*proof*⟩
interpretation *coeff-lift-hom*: *inj-comm-ring-hom coeff-lift*⟨*proof*⟩
interpretation *coeff-lift-hom*: *inj-idom-hom coeff-lift*⟨*proof*⟩

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* ⟨*proof*⟩
interpretation *to-fract-hom*: *idom-hom to-fract*⟨*proof*⟩
interpretation *to-fract-hom*: *inj-idom-hom to-fract*⟨*proof*⟩

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*
⟨*proof*⟩

lemma *coeff-int-smult*[*simp*]: *coeff-int* (*smult c p*) *i* = *c* ∗ *coeff-int p i*
⟨*proof*⟩

lemma *coeff-int-signof-mult*: *coeff-int* (*of-int* (*sign x*) ∗ *f*) *i* = *of-int* (*sign x*) ∗ *coeff-int f i*
⟨*proof*⟩

lemma *coeff-int-sum*: *coeff-int* (*sum p A*) *i* = (∑*x*∈*A*. *coeff-int* (*p x*) *i*)
⟨*proof*⟩

lemma *coeff-int-0*[*simp*]: *coeff-int f 0* = *coeff f 0* ⟨*proof*⟩

lemma *coeff-int-monom-mult*: *coeff-int* (*monom a d* ∗ *f*) *i* = (*a* ∗ *coeff-int f* (*i* − *d*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *coeff-int*[*simp*]: *coeff-int p n* = *coeff p n* ⟨*proof*⟩

lemma *coeff-int-minus*[*simp*]:
coeff-int (*a* − *b*) *i* = *coeff-int a i* − *coeff-int b i*
⟨*proof*⟩

lemma *coeff-int-pCons-0*[*simp*]: *coeff-int* (*pCons 0 b*) *i* = *coeff-int b* (*i* − *1*)
⟨*proof*⟩

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 optimization 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*
⟨*proof*⟩

Lazard's optimization is only proven for factorial rings.

lemma *dichotomous-Lazard*: *div-exp-sound (dichotomous-Lazard :: 'a :: factorial-ring-gcd ⇒ -)*
⟨*proof*⟩

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))*
⟨*proof*⟩

lemma *for-all-Suc*: *P i ⟹ (∀ j ≥ Suc i. P j) = (∀ j ≥ i. P j)* **for** *P*
⟨*proof*⟩

lemma *pseudo-mod-left-0*[*simp*]: *pseudo-mod 0 x = 0*
⟨*proof*⟩

lemma *pseudo-mod-right-0*[*simp*]: *pseudo-mod x 0 = x*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *snd-pseudo-mod-smult-left*[*simp*]:
shows *snd* (*pseudo-divmod* (*smult* (*x*::′*a*::*idom*) *p*) *q*) = (*smult x* (*snd* (*pseudo-divmod p q*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *pseudo-mod-zero*[*simp*]:
pseudo-mod 0 f = (*0*::′*a* :: {*idom*} *poly*)
pseudo-mod f 0 = *f*
⟨*proof*⟩

lemma *prod-combine*:
assumes *j* ≤ *i*
shows *f i* ∗ (∏ *l*←[*j*..<*i*]. (*f l* :: ′*a*::*comm-monoid-mult*)) = *prod-list* (*map f* [*j*..<*Suc i*])
⟨*proof*⟩

lemma *prod-list-minus-1-exp*: *prod-list* (*map* (λ *i*. (−*1*)^(*f i*)) *xs*)
= (−*1*)^(*sum-list* (*map f xs*))
⟨*proof*⟩

lemma *minus-1-power-even*: (− (*1* :: ′*b* :: *comm-ring-1*))^ *k* = (*if even k then 1 else* (−*1*))
⟨*proof*⟩

lemma *minus-1-even-eqI*: **assumes** *even k* = *even l* **shows**

$(- (1 :: 'b :: comm\text{-}ring\text{-}1))\hat{\ }k = (- 1)\hat{\ }l$
⟨*proof*⟩

lemma (**in** *comm-monoid-mult*) *prod-list-multf*:
$(\prod x{\leftarrow}xs.\ f\ x * g\ x) = prod\text{-}list\ (map\ f\ xs) * prod\text{-}list\ (map\ g\ xs)$
⟨*proof*⟩

lemma *inverse-prod-list*: $inverse\ (prod\text{-}list\ xs) = prod\text{-}list\ (map\ inverse\ (xs :: 'a :: field\ list))$
⟨*proof*⟩

definition *pow-int* :: *'a* :: *field* ⇒ *int* ⇒ *'a* **where**
$pow\text{-}int\ x\ e = (if\ e < 0\ then\ 1\ /\ (x\ \hat{\ }(nat\ (-e)))\ else\ x\ \hat{\ }(nat\ e))$

lemma *pow-int-0*[*simp*]: $pow\text{-}int\ x\ 0 = 1$ ⟨*proof*⟩

lemma *pow-int-1*[*simp*]: $pow\text{-}int\ x\ 1 = x$ ⟨*proof*⟩

lemma *exp-pow-int*: $x\ \hat{\ }n = pow\text{-}int\ x\ n$
⟨*proof*⟩

lemma *pow-int-add*: **assumes** *x*: $x \neq 0$ **shows** $pow\text{-}int\ x\ (a + b) = pow\text{-}int\ x\ a * pow\text{-}int\ x\ b$
⟨*proof*⟩

lemma *pow-int-mult*: $pow\text{-}int\ (x * y)\ a = pow\text{-}int\ x\ a * pow\text{-}int\ y\ a$
⟨*proof*⟩

lemma *pow-int-base-1*[*simp*]: $pow\text{-}int\ 1\ a = 1$
⟨*proof*⟩

lemma *pow-int-divide*: $a\ /\ pow\text{-}int\ x\ b = a * pow\text{-}int\ x\ (-b)$
⟨*proof*⟩

lemma *divide-prod-assoc*: $x\ /\ (y * z :: 'a :: field) = x\ /\ y\ /\ z$ ⟨*proof*⟩

lemma *minus-1-inverse-pow*[*simp*]: $x\ /\ (-1)\hat{\ }n = (x :: 'a :: field) * (-1)\hat{\ }n$
⟨*proof*⟩

definition *subresultant-mat* :: *nat* ⇒ *'a* :: *comm-ring-1 poly* ⇒ *'a poly* ⇒ *'a poly mat* **where**
subresultant-mat J F G = (*let*
$dg = degree\ G;\ df = degree\ F;\ f = coeff\text{-}int\ F;\ g = coeff\text{-}int\ G;\ n = (df - J) + (dg - J)$
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 ≡ 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*)
⟨*proof*⟩

definition *subresultant′-mat* :: *nat* ⇒ *nat* ⇒ ′*a* :: *comm-ring-1 poly* ⇒ ′*a poly* ⇒ ′*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*) :]))
⟨*proof*⟩

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*⟩

lemma *subresultant-swap*:
shows *subresultant J f g* = *smult* ((− *1*) ^ ((*degree f* − *J*) ∗ (*degree g* − *J*))) (*subresultant J g f*)

⟨*proof*⟩

lemma *subresultant-smult-right*:**assumes** $(c :: 'a :: \{comm\text{-}ring\text{-}1, semiring\text{-}no\text{-}zero\text{-}divisors\})$ $\neq 0$
  shows *subresultant J f (smult c g) = smult (c ^ (degree f − J)) (subresultant J f g)*
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *subresultant′-mat-sylvester-mat*: *transpose-mat (subresultant′-mat 0 0 f g) = sylvester-mat f g*
⟨*proof*⟩

lemma *coeff-subresultant-0-0-resultant*: *coeff (subresultant 0 f g) 0 = resultant f g*
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *subresultant-resultant*: *subresultant 0 f g = [: resultant f g :]*
⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

lemma *resultant-smult-right*: **assumes** (*c* :: *'a* :: *idom*) ≠ *0*
  shows *resultant f* (*smult c g*) = *c* ^ *degree f* ∗ *resultant f g*
  ⟨*proof*⟩

lemma *resultant-swap*: *resultant f g* = (−*1*)^(*degree f* ∗ *degree g*) ∗ (*resultant g f*)
  ⟨*proof*⟩

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⟩

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⟩

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⟩

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 ⟨proof⟩

lemma f0: i ≠ 0 ⟹ f i = 0 ⟷ i > k
 ⟨proof⟩

lemma n-gt: **assumes** 2 ≤ i i < k
 shows n i > n (Suc i)
⟨proof⟩

lemma *n-ge*: **assumes** $1 \leq i$ $i < k$
shows $n\ i \geq n\ (Suc\ i)$
⟨*proof*⟩

lemma *n-ge-trans*: **assumes** $1 \leq i$ $i \leq j$ $j \leq k$
shows $n\ i \geq n\ j$
⟨*proof*⟩

lemma *delta-gt*: **assumes** $2 \leq i$ $i < k$
shows $\delta\ i > 0$ ⟨*proof*⟩

lemma *k2*:$2 \leq k$
⟨*proof*⟩

lemma *k0*: $k \neq 0$ ⟨*proof*⟩

lemma *ni2*:$3 \leq i \Longrightarrow i \leq k \Longrightarrow n\ i \neq n\ 2$
⟨*proof*⟩
end

locale *subresultant-prs-locale* = *primitive-remainder-sequence* $F\ n\ \delta\ f\ k\ \beta$ **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 $\alpha 0$: $i > 1 \Longrightarrow \alpha\ i = 0 \longleftrightarrow (i - 1) > k$
⟨*proof*⟩

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*⟩

definition Q :: *nat* ⇒ *'a fract poly* **where**
$Q\ i \equiv smult\ (\alpha\ i)\ (fst\ (pdivmod\ (F\ (i - 2))\ (F\ (i - 1))))$

lemma *beta-F-as-sum*:

assumes *3* ≤ *i i* ≤ *Suc k*
shows *smult* (*β i*) (*F i*) = *smult* (*α i*) (*F* (*i* − *2*)) + − *Q i* ∗ *F* (*i* − *1*) (**is** *?t1*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *nonzero-alphaprod*: **assumes** *i* ≤ *k* + *1* **shows** (∏ *l*←[*3*..<*i*]. *α l* ^(*p l*)) ≠ *0*
⟨*proof*⟩

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*⟩

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*⟨*proof*⟩

definition $\sigma\ i = (\sum l{\leftarrow}[3..{<}Suc\ i].\ (n\ (l - 2) + n\ (i - 1) + 1) * (n\ (l - 1) + n\ (i - 1) + 1))$
definition $\tau\ i = (\sum l{\leftarrow}[3..{<}Suc\ i].\ (n\ (l - 2) + n\ i) * (n\ (l - 1) + n\ i))$

definition $\gamma\ i = (-1)\hat{\ }(\sigma\ i) * pow\text{-}int\ (\ f\ (i - 1))\ (1 - int\ (\delta\ (i - 1))) * (\prod l{\leftarrow}[3..{<}Suc\ i].$
$(\beta\ l\ /\ \alpha\ l)\hat{\ }(n\ (l - 1) - n\ (i - 1) + 1) * (f\ (l - 1))\hat{\ }(\delta\ (l - 2) + \delta\ (l - 1)))$
definition $\Theta\ i = (-1)\hat{\ }(\tau\ i) * pow\text{-}int\ (f\ i)\ (int\ (\delta\ (i - 1)) - 1) * (\prod l{\leftarrow}[3..{<}Suc\ i].$
$(\beta\ l\ /\ \alpha\ l)\hat{\ }(n\ (l - 1) - n\ i) * (f\ (l - 1))\hat{\ }(\delta\ (l - 2) + \delta\ (l - 1)))$

lemma *fundamental-theorem-eq-4*: **assumes** *i*: $3 \leq i\ i \leq k$
shows *ffp* $(R\ (n\ (i - 1) - 1)) = smult\ (\gamma\ i)\ (F\ i)$
⟨*proof*⟩

lemma *fundamental-theorem-eq-5*: **assumes** *i*: $3 \leq i\ i \leq k\ n\ i < j\ j < n\ (i - 1) - 1$
shows $R\ j = 0$
⟨*proof*⟩

lemma *fundamental-theorem-eq-6*: **assumes** $3 \leq i\ i \leq k$ **shows** *ffp* $(R\ (n\ i)) = smult\ (\Theta\ i)\ (F\ i)$
(**is** *?lhs=?rhs*)
⟨*proof*⟩

lemma *fundamental-theorem-eq-7*: **assumes** *j*: $j < n\ k$ **shows** $R\ j = 0$
⟨*proof*⟩

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*⟩

fun *h* :: *nat* ⇒ *'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*⟩

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*: $\beta\ 3 = (-1)\hat{}(\delta\ 1 + 1)$
and *βi*: $\bigwedge i.\ 4 \le i \Longrightarrow i \le Suc\ k \Longrightarrow \beta\ i = (-1)\hat{}(\delta\ (i - 2) + 1) * f\ (i - 2) * h\ (i - 2)\ \hat{}\ (\delta\ (i - 2))$
begin

lemma *B-eq-17-main*: $2 \le i \Longrightarrow i \le k \Longrightarrow$
$h\ i = (-1)\ \hat{}\ (n\ 1 + n\ i + i + 1)\ /\ f\ i$
$* (\prod l \leftarrow [3..<\ Suc\ (Suc\ i)].\ (\alpha\ l\ /\ \beta\ l)) \wedge h\ i \neq 0$
⟨*proof*⟩

lemma *B-eq-17*: $2 \le i \Longrightarrow i \le k \Longrightarrow$
$h\ i = (-1)\ \hat{}\ (n\ 1 + n\ i + i + 1)\ /\ f\ i * (\prod l \leftarrow [3..<\ Suc\ (Suc\ i)].\ (\alpha\ l\ /\ \beta\ l))$
⟨*proof*⟩

lemma *B-theorem-2*: $3 \le i \Longrightarrow i \le Suc\ k \Longrightarrow \gamma\ i = 1$
⟨*proof*⟩

context
fixes *i* :: *nat*
assumes *i*: $3 \le i\ i \le k$
begin
lemma *B-theorem-3-b*: $\Theta\ i * f\ i = ff\ (lead\text{-}coeff\ (H\ i))$
⟨*proof*⟩

lemma *B-theorem-3-main*: $\Theta\ i * f\ i\ /\ \gamma\ (i + 1) = (-1)\hat{}(n\ 1 + n\ i + i + 1)\ / f\ i * (\prod l \leftarrow [3..<\ Suc\ (Suc\ i)].\ (\alpha\ l\ /\ \beta\ l))$
⟨*proof*⟩

lemma *B-theorem-3*: $h\ i = \Theta\ i * f\ i\ h\ i = ff\ (lead\text{-}coeff\ (H\ i))$
⟨*proof*⟩
end

lemma *h0*: $i \leq k \Longrightarrow h\ i \neq 0$
⟨*proof*⟩

lemma *deg-G12*: *degree G1* ≥ *degree G2* ⟨*proof*⟩

lemma *R0*: **shows** *R 0* = [: *resultant G1 G2* :]
⟨*proof*⟩

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

interpretation *div-exp-sound div-exp* ⟨*proof*⟩

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*⟩

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*⟩

lemma *resultant-impl-main*: *resultant-impl-main G1 G2* = *resultant G1 G2*
⟨*proof*⟩
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*))))
⟨*proof*⟩
termination
⟨*proof*⟩

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*⟩ **definition** *k* = (*LEAST i. F* (*Suc i*) = *0*)

private lemma *k-exists*: ∃ *i. F* (*Suc i*) = *0*
⟨*proof*⟩ **lemma** *k*: *F* (*Suc k*) = *0 i* < *k* ⟹ *F* (*Suc i*) ≠ *0*
⟨*proof*⟩

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*⟩
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*⟩

theorem *resultant-impl*: *resultant-impl* = *resultant*
⟨*proof*⟩
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*
⟨*proof*⟩

lemma *subresultant-prs-impl*: *subresultant-prs-impl f G1 G2* = *f* (*subresultant-prs G1 G2*)
⟨*proof*⟩

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*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *resultant-code*: *resultant* = *resultant-impl*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *gcd-impl-primitive*: **assumes** *primitive-part G1* = *G1* **and** *primitive-part G2* = *G2*
shows *gcd-impl-primitive G1 G2* = *gcd G1 G2*
⟨*proof*⟩
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*⟩

theorem *gcd-impl*[*simp*]: *gcd-impl* = *gcd*
⟨*proof*⟩

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*⟩

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)
  ⟨proof⟩
```

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⟩
```

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*))))
⟨*proof*⟩

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* ⟨*proof*⟩
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*
⟨*proof*⟩

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.