

Subresultants*

Sebastian 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

1	Introduction	2
2	Resultants	2
3	Dichotomous Lazard	5
4	Binary Exponentiation	6
5	Homomorphisms	7
6	Polynomial coefficients with integer index	7
7	Subresultants and the subresultant PRS	9
7.1	Algorithm	9
7.2	Soundness Proof for <i>div-exp-param.resultant-impl div-exp = resultant</i>	10
7.3	Code Equations	22
8	Computing the Gcd via the subresultant PRS	26
8.1	Algorithm	26
8.2	Soundness Proof for <i>gcd-impl = gcd</i>	26
8.3	Code Equations	28

*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* \Rightarrow *nat* \Rightarrow *'a poly* \Rightarrow *'a poly* \Rightarrow *'a* :: *zero mat*
where

sylvester-mat-sub $m\ n\ p\ q \equiv$
 $\text{mat } (m+n)\ (m+n)\ (\lambda\ (i,j)).$
 if $i < n$ then
 if $i \leq j \wedge j - i \leq m$ then $\text{coeff } p\ (m + i - j)$ else 0
 else if $i - n \leq j \wedge j \leq i$ then $\text{coeff } q\ (i-j)$ else 0

definition *sylvester-mat* :: 'a poly \Rightarrow 'a poly \Rightarrow 'a :: zero mat **where**
sylvester-mat $p\ q \equiv \text{sylvester-mat-sub } (\text{degree } p)\ (\text{degree } q)\ p\ q$

lemma *sylvester-mat-sub-dim[simp]*:
 fixes $m\ n\ p\ q$
 defines $S \equiv \text{sylvester-mat-sub } m\ n\ p\ q$
 shows $\text{dim-row } S = m+n$ and $\text{dim-col } S = m+n$
 <proof>

lemma *sylvester-mat-sub-carrier*:
 shows $\text{sylvester-mat-sub } m\ n\ p\ q \in \text{carrier-mat } (m+n)\ (m+n)$ <proof>

lemma *sylvester-mat-dim[simp]*:
 fixes $p\ q$
 defines $d \equiv \text{degree } p + \text{degree } q$
 shows $\text{dim-row } (\text{sylvester-mat } p\ q) = d$ and $\text{dim-col } (\text{sylvester-mat } p\ q) = d$
 <proof>

lemma *sylvester-carrier-mat*:
 fixes $p\ q$
 defines $d \equiv \text{degree } p + \text{degree } q$
 shows $\text{sylvester-mat } p\ q \in \text{carrier-mat } d\ d$ <proof>

lemma *sylvester-mat-sub-index*:
 fixes $p\ q$
 assumes $i: i < m+n$ and $j: j < m+n$
 shows $\text{sylvester-mat-sub } m\ n\ p\ q\ \$\$ (i,j) =$
 (if $i < n$ then
 if $i \leq j \wedge j - i \leq m$ then $\text{coeff } p\ (m + i - j)$ else 0
 else if $i - n \leq j \wedge j \leq i$ then $\text{coeff } q\ (i-j)$ else 0)
 <proof>

lemma *sylvester-index-mat*:
 fixes $p\ q$
 defines $m \equiv \text{degree } p$ and $n \equiv \text{degree } q$
 assumes $i: i < m+n$ and $j: j < m+n$
 shows $\text{sylvester-mat } p\ q\ \$\$ (i,j) =$
 (if $i < n$ then
 if $i \leq j \wedge j - i \leq m$ then $\text{coeff } p\ (m + i - j)$ else 0
 else if $i - n \leq j \wedge j \leq i$ then $\text{coeff } q\ (i - j)$ else 0)
 <proof>

lemma *sylvester-index-mat2*:

fixes $p\ q :: 'a :: \text{comm-semiring-1 poly}$
defines $m \equiv \text{degree } p$ **and** $n \equiv \text{degree } q$
assumes $i: i < m+n$ **and** $j: j < m+n$
shows $\text{sylvester-mat } p\ q\ \$\$ (i,j) =$
 (if $i < n$ *then* $\text{coeff } (\text{monom } 1\ (n - i) * p)\ (m+n-j)$
 else $\text{coeff } (\text{monom } 1\ (m + n - i) * q)\ (m+n-j)$)
 <proof>

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

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

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

lemma $\text{sylvester-mat-sub-map}$:
assumes $f0: f\ 0 = 0$
shows $\text{map-mat } f\ (\text{sylvester-mat-sub } m\ n\ p\ q) = \text{sylvester-mat-sub } m\ n\ (\text{map-poly } f\ p)$
 (is ?l = ?r)
 <proof>

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

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

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

lemma resultant-sub : $\text{resultant } p\ q = \text{resultant-sub } (\text{degree } p)\ (\text{degree } q)\ p\ q$
 <proof>

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

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

lemma resultant-0[simp] :

fixes $p :: 'a :: \text{comm-ring-1 poly}$
assumes $\text{degree } p > 0$
shows $\text{resultant } 0 \ p = 0 \ \text{resultant } p \ 0 = 0$
 $\langle \text{proof} \rangle$

lemma (**in** comm-ring-hom) $\text{resultant-map-poly}$: $\text{degree } (\text{map-poly hom } p) = \text{degree } p \implies$
 $\text{degree } (\text{map-poly hom } q) = \text{degree } q \implies \text{resultant } (\text{map-poly hom } p) (\text{map-poly hom } q) = \text{hom } (\text{resultant } p \ q)$
 $\langle \text{proof} \rangle$

lemma (**in** inj-comm-ring-hom) resultant-hom : $\text{resultant } (\text{map-poly hom } p) (\text{map-poly hom } q) = \text{hom } (\text{resultant } p \ q)$
 $\langle \text{proof} \rangle$

end

3 Dichotomous Lazard

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

theory $\text{Dichotomous-Lazard}$

imports

$\text{HOL-Computational-Algebra.Polynomial-Factorial}$

begin

lemma $\text{power-fract}[simp]$: $(\text{Fract } a \ b)^{\wedge n} = \text{Fract } (a^{\wedge n}) \ (b^{\wedge n})$
 $\langle \text{proof} \rangle$

lemma $\text{range-to-fract-dvd-iff}$: **assumes** $b: b \neq 0$
shows $\text{Fract } a \ b \in \text{range to-fract} \iff b \ \text{dvd } a$
 $\langle \text{proof} \rangle$

lemma $\text{Fract-cases-coprime}$ [cases type: fract]:
fixes $q :: 'a :: \text{factorial-ring-gcd fract}$
obtains $(\text{Fract}) \ a \ b$ **where** $q = \text{Fract } a \ b \ b \neq 0 \ \text{coprime } a \ b$
 $\langle \text{proof} \rangle$

lemma to-fract-power-le : **fixes** $a :: 'a :: \text{factorial-ring-gcd fract}$
assumes $\text{no-fract}: a * b^{\wedge e} \in \text{range to-fract}$
and $a: a \in \text{range to-fract}$
and $le: f \leq e$
shows $a * b^{\wedge f} \in \text{range to-fract}$
 $\langle \text{proof} \rangle$

lemma $\text{div-divide-to-fract}$: **assumes** $x \in \text{range to-fract}$
and $x = (y :: 'a :: \text{idom-divide fract}) / z$
and $x' = y' \ \text{div } z'$

```

and  $y = \text{to-fract } y' \ z = \text{to-fract } z'$ 
shows  $x = \text{to-fract } x'$ 
<proof>

```

```

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

```

```

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

```

```

lemma dichotomous-Lazard-main: fixes  $x :: 'a :: \text{idom-divide}$ 
assumes  $\bigwedge i. i \leq n \implies (\text{to-fract } x)^{\wedge i} / (\text{to-fract } y)^{\wedge (i-1)} \in \text{range to-fract}$ 
shows  $\text{to-fract } (\text{dichotomous-Lazard } x \ y \ n) = (\text{to-fract } x)^{\wedge n} / (\text{to-fract } y)^{\wedge (n-1)}$ 

```

```

<proof>

```

```

lemma dichotomous-Lazard: fixes  $x :: 'a :: \text{factorial-ring-gcd}$ 
assumes  $(\text{to-fract } x)^{\wedge n} / (\text{to-fract } y)^{\wedge (n-1)} \in \text{range to-fract}$ 
shows  $\text{to-fract } (\text{dichotomous-Lazard } x \ y \ n) = (\text{to-fract } x)^{\wedge n} / (\text{to-fract } y)^{\wedge (n-1)}$ 

```

```

<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-Division.divmod-nat-def[termination-simp]

```

```

context monoid-mult

```

```

begin

```

```

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

```

lemma *binary-power*[simp]: *binary-power* = (\frown)
<proof>

lemma *binary-power-code-unfold*[code-unfold]: (\frown) = *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* == $\lambda a. [: a :]$

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 coefficient 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  $\Rightarrow$  int  $\Rightarrow$  'a where
  coeff-int p i = (if i < 0 then 0 else coeff p (nat i))

lemma coeff-int-eq-0: i < 0  $\vee$  i > int (degree p)  $\implies$  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 = ( $\sum$  x  $\in$  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  $\notin$  xs
  and  $\bigwedge$  x. x  $\in$  xs  $\implies$  degree (f x) = 0
shows coeff (prod f (insert y xs)) i = prod ( $\lambda$  x. coeff (f x) 0) xs * coeff (f y) i
  <proof>

lemma coeff-int-prod-const: assumes finite xs and y  $\notin$  xs
  and  $\bigwedge$  x. x  $\in$  xs  $\implies$  degree (f x) = 0
shows coeff-int (prod f (insert y xs)) i = prod ( $\lambda$  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  $\Rightarrow$  'a  $\Rightarrow$  nat  $\Rightarrow$  '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;
     $\delta$  = m - n;
    d = div-exp lg c  $\delta$ ;
    h = pseudo-mod f g
  in if h = 0 then (g,d)
    else subresultant-prs-main g (sdiv-poly h ((-1)  $\wedge$  ( $\delta$  + 1) * lf * (c  $\wedge$   $\delta$ ))) d)

definition subresultant-prs where
  subresultant-prs f g = (let
    h = pseudo-mod f g;
     $\delta$  = (degree f - degree g);
    d = lead-coeff g  $\wedge$   $\delta$ 
  in if h = 0 then (g,d)
    else subresultant-prs-main g ((- 1)  $\wedge$  ( $\delta$  + 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: $\bigwedge x y n.$

(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 = xⁿ 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** $\bigwedge x. x \in \text{set } xs \implies \text{even } (f x) = \text{even } (g x)$

shows even (sum-list (map f xs)) = even (sum-list (map g xs))

⟨proof⟩

lemma for-all-Suc: $P i \implies (\forall j \geq \text{Suc } i. P j) = (\forall j \geq i. P j)$ 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 ($x::'a::idom$) $\neq 0$ $q\neq 0$

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 $j \leq i$

shows f $i * (\prod_{l \leftarrow [j..<i]}$. (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**

$(- (1 :: 'b :: comm-ring-1))^k = (- 1)^l$
 ⟨proof⟩

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

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

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

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

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

lemma *exp-pow-int*: $x ^ n = pow-int x n$
 ⟨proof⟩

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

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

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

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

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

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

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 ($\lambda (i,j). if j < dg - J then$
 $if i = n - 1 then monom 1 (dg - J - 1 - j) * F else [f (df - int i + int j) :]$)

else let $jj = j - (dg - J)$ in
 if $i = n - 1$ then monom 1 $(df - J - 1 - jj) * G$ else $[: g (dg - \text{int } i + \text{int } jj) :])$)

lemma *subresultant-mat-dim*[simp]:

fixes $j p q$
defines $S \equiv \text{subresultant-mat } j p q$
shows $\text{dim-row } S = (\text{degree } p - j) + (\text{degree } q - j)$ **and** $\text{dim-col } S = (\text{degree } p - j) + (\text{degree } q - j)$
 ⟨proof⟩

definition *subresultant'-mat* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow 'a :: \text{comm-ring-1 poly} \Rightarrow 'a \text{ poly} \Rightarrow 'a \text{ mat}$ **where**

subresultant'-mat $J l F G = (\text{let}$
 $\gamma = \text{degree } G; \varphi = \text{degree } F; f = \text{coeff-int } F; g = \text{coeff-int } G; n = (\varphi - J) + (\gamma - J)$
 in mat $n n (\lambda (i,j). \text{if } j < \gamma - J \text{ then}$
 if $i = n - 1$ then $(f (l - \text{int } (\gamma - J - 1) + \text{int } j))$ else $(f (\varphi - \text{int } i + \text{int } j))$)
 else let $jj = j - (\gamma - J)$ in
 if $i = n - 1$ then $(g (l - \text{int } (\varphi - J - 1) + \text{int } jj))$ else $(g (\gamma - \text{int } i + \text{int } jj))$))

lemma *subresultant-index-mat*:

fixes $F G$
assumes $i < (\text{degree } F - J) + (\text{degree } G - J)$ **and** $j < (\text{degree } F - J) + (\text{degree } G - J)$
shows *subresultant-mat* $J F G \text{ \$\$ } (i,j) =$
 (if $j < \text{degree } G - J$ then
 if $i = (\text{degree } F - J) + (\text{degree } G - J) - 1$ then monom 1 $(\text{degree } G - J - 1 - j) * F$ else $[: \text{coeff-int } F (\text{degree } F - \text{int } i + \text{int } j) :])$
 else let $jj = j - (\text{degree } G - J)$ in
 if $i = (\text{degree } F - J) + (\text{degree } G - J) - 1$ then monom 1 $(\text{degree } F - J - 1 - jj) * G$ else $[: \text{coeff-int } G (\text{degree } G - \text{int } i + \text{int } jj) :])$)
 ⟨proof⟩

definition *subresultant* :: $\text{nat} \Rightarrow 'a :: \text{comm-ring-1 poly} \Rightarrow 'a \text{ poly} \Rightarrow 'a \text{ poly}$ **where**
subresultant $J F G = \text{det } (\text{subresultant-mat } J F G)$

lemma *subresultant-smult-left*: **assumes** $(c :: 'a :: \{\text{comm-ring-1, semiring-no-zero-divisors}\}) \neq 0$

shows *subresultant* $J (\text{smult } c f) g = \text{smult } (c \wedge (\text{degree } g - J)) (\text{subresultant } J f g)$
 ⟨proof⟩

lemma *subresultant-swap*:

shows *subresultant* $J f g = \text{smult } ((-1) \wedge ((\text{degree } f - J) * (\text{degree } g - J))) (\text{subresultant } J g f)$

<proof>

lemma *subresultant-smult-right:assumes* ($c :: 'a :: \{\text{comm-ring-1}, \text{semiring-no-zero-divisors}\}$)
 $\neq 0$

shows $\text{subresultant } J f (\text{smult } c g) = \text{smult } (c \wedge (\text{degree } f - J)) (\text{subresultant } J f g)$
<proof>

lemma *coeff-subresultant: coeff* ($\text{subresultant } J F G$) $l =$

(*if* $\text{degree } F - J + (\text{degree } G - J) = 0 \wedge l \neq 0$ *then* 0 *else* $\det (\text{subresultant}'\text{-mat } J l F G)$)
<proof>

lemma *subresultant'-zero-ge: assumes* ($\text{degree } f - J + (\text{degree } g - J) \neq 0$ **and**
 $k \geq \text{degree } f + (\text{degree } g - J)$)

shows $\det (\text{subresultant}'\text{-mat } J k f g) = 0$
<proof>

lemma *subresultant'-zero-lt: assumes*

$J: J \leq \text{degree } f \ J \leq \text{degree } g \ J < k$

and $k: k < \text{degree } f + (\text{degree } g - J)$

shows $\det (\text{subresultant}'\text{-mat } J k f g) = 0$
<proof>

lemma *subresultant'-mat-sylvester-mat: transpose-mat* ($\text{subresultant}'\text{-mat } 0 0 f g$)
 $= \text{sylvester-mat } f g$

<proof>

lemma *coeff-subresultant-0-0-resultant: coeff* ($\text{subresultant } 0 f g$) $0 = \text{resultant } f g$

<proof>

lemma *subresultant-zero-ge: assumes* $k \geq \text{degree } f + (\text{degree } g - J)$

and ($\text{degree } f - J + (\text{degree } g - J) \neq 0$)

shows $\text{coeff } (\text{subresultant } J f g) k = 0$

<proof>

lemma *subresultant-zero-lt: assumes* $k < \text{degree } f + (\text{degree } g - J)$

and $J \leq \text{degree } f \ J \leq \text{degree } g \ J < k$

shows $\text{coeff } (\text{subresultant } J f g) k = 0$

<proof>

lemma *subresultant-resultant: subresultant* $0 f g = [; \text{resultant } f g ;]$

<proof>

lemma (*in inj-comm-ring-hom*) *subresultant-hom:*

$\text{map-poly hom } (\text{subresultant } J f g) = \text{subresultant } J (\text{map-poly hom } f) (\text{map-poly hom } g)$

<proof>

We now derive properties of the resultant via the connection to subre-

sultants.

lemma resultant-smult-left: **assumes** $(c :: 'a :: idom) \neq 0$
shows $\text{resultant} (\text{smult } c \ f) \ g = c \wedge \text{degree } g * \text{resultant } f \ g$
 $\langle \text{proof} \rangle$

lemma resultant-smult-right: **assumes** $(c :: 'a :: idom) \neq 0$
shows $\text{resultant } f (\text{smult } c \ g) = c \wedge \text{degree } f * \text{resultant } f \ g$
 $\langle \text{proof} \rangle$

lemma resultant-swap: $\text{resultant } f \ g = (-1) \wedge (\text{degree } f * \text{degree } g) * (\text{resultant } g \ f)$
 $\langle \text{proof} \rangle$

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

lemma fixes $F \ B \ G \ H :: 'a :: idom \ \text{poly}$ **and** $J :: nat$

defines $df: df \equiv \text{degree } F$

and $dg: dg \equiv \text{degree } G$

and $dh: dh \equiv \text{degree } H$

and $db: db \equiv \text{degree } B$

defines

$n: n \equiv (df - J) + (dg - J)$

and $f: f \equiv \text{coeff-int } F$

and $b: b \equiv \text{coeff-int } B$

and $g: g \equiv \text{coeff-int } G$

and $h: h \equiv \text{coeff-int } H$

assumes $FGH: F + B * G = H$

and $dfg: df \geq dg$

and $\text{choice: } dg > dh \vee H = 0 \wedge F \neq 0 \wedge G \neq 0$

shows $BT\text{-eq-18: subresultant } J \ F \ G = \text{smult } ((-1) \wedge ((df - J) * (dg - J))) (\text{det} (\text{mat } n \ n$
 $(\lambda \ (i,j).$

$\text{if } j < df - J$

$\text{then if } i = n - 1 \text{ then monom } 1 \ ((df - J) - 1 - j) * G$

$\text{else } [:g \ (\text{int } dg - \text{int } i + \text{int } j):]$

$\text{else if } i = n - 1 \text{ then monom } 1 \ ((dg - J) - 1 - (j - (df - J))) * H$

$\text{else } [:h \ (\text{int } df - \text{int } i + \text{int } (j - (df - J))):]$

$(\text{is } - = \text{smult } ?m1 \ ?right)$

and $BT\text{-eq-19: } dh \leq J \implies J < dg \implies \text{subresultant } J \ F \ G = \text{smult } ($

$(-1) \wedge ((df - J) * (dg - J)) * \text{lead-coeff } G \wedge (df - J) * \text{coeff } H \ J \wedge (dg - J - 1)) \ H$

$(\text{is } - \implies - \implies - = \text{smult } (- * ?G * ?H) \ H)$

and $BT\text{-lemma-1-12: } J < dh \implies \text{subresultant } J \ F \ G = \text{smult } ($

$(-1) \wedge ((df - J) * (dg - J)) * \text{lead-coeff } G \wedge (df - dh)) (\text{subresultant } J \ G \ H)$

and $BT\text{-lemma-1-13': } J = dh \implies dg > dh \vee H \neq 0 \implies \text{subresultant } dh \ F \ G = \text{smult } ($

$(-1) \wedge ((df - dh) * (dg - dh)) * \text{lead-coeff } G \wedge (df - dh) * \text{lead-coeff } H \wedge (dg - dh - 1)) \ H$

and $BT\text{-lemma-1-14: } dh < J \implies J < dg - 1 \implies \text{subresultant } J \ F \ G = 0$

and $BT\text{-lemma-1-15': } J = dg - 1 \implies dg > dh \vee H \neq 0 \implies \text{subresultant } (dg$

- 1) $F G = \text{smult } (-1)^{\wedge(df - dg + 1)} * \text{lead-coeff } G \wedge (df - dg + 1)) H$
 <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** $F :: 'a :: \text{idom poly}$
assumes $F = B * G$
and FG : $\text{degree } F \geq \text{degree } G$
shows *subresultant* $J F G = (\text{if } J < \text{degree } G \text{ then } 0 \text{ else}$
 $\text{if } J < \text{degree } F \text{ then } \text{smult } (\text{lead-coeff } G \wedge (\text{degree } F - J - 1)) G \text{ else } 1)$
 <proof>

lemma *resultant-pseudo-mod-0*: **assumes** $\text{pseudo-mod } f g = (0 :: 'a :: \text{idom-divide poly})$
and dfg : $\text{degree } f \geq \text{degree } g$
and f : $f \neq 0$ **and** g : $g \neq 0$
shows $\text{resultant } f g = (\text{if } \text{degree } g = 0 \text{ then } \text{lead-coeff } g \wedge \text{degree } f \text{ else } 0)$
 <proof>

locale *primitive-remainder-sequence* =
fixes $F :: \text{nat} \Rightarrow 'a :: \text{idom-divide poly}$
and $n :: \text{nat} \Rightarrow \text{nat}$
and $\delta :: \text{nat} \Rightarrow \text{nat}$
and $f :: \text{nat} \Rightarrow 'a$
and $k :: \text{nat}$
and $\beta :: \text{nat} \Rightarrow 'a$
assumes f : $\bigwedge i. f i = \text{lead-coeff } (F i)$
and n : $\bigwedge i. n i = \text{degree } (F i)$
and δ : $\bigwedge i. \delta i = n i - n (\text{Suc } i)$
and $n12$: $n 1 \geq n 2$
and $F12$: $F 1 \neq 0 \wedge F 2 \neq 0$
and $F0$: $\bigwedge i. i \neq 0 \implies F i = 0 \iff i > k$
and $\beta 0$: $\bigwedge i. \beta i \neq 0$
and pmod : $\bigwedge i. i \geq 3 \implies i \leq \text{Suc } k \implies \text{smult } (\beta i) (F i) = \text{pseudo-mod } (F$
 $(i - 2)) (F (i - 1))$
begin

lemma *f10*: $f 1 \neq 0$ **and** *f20*: $f 2 \neq 0$ <proof>

lemma *f0*: $i \neq 0 \implies f i = 0 \iff i > k$
 <proof>

lemma *n-gt*: **assumes** $2 \leq i < k$
shows $n i > n (\text{Suc } i)$
 <proof>

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

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

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

lemma *k2:2*: $2 \leq k$
 $\langle proof \rangle$

lemma *k0*: $k \neq 0$ $\langle proof \rangle$

lemma *ni2:3*: $3 \leq i \implies i \leq k \implies n\ i \neq n\ 2$
 $\langle proof \rangle$
end

locale *subresultant-prs-locale* = *primitive-remainder-sequence* $F\ n\ \delta\ f\ k\ \beta$ **for**

$F :: nat \Rightarrow 'a :: idom-divide\ fract\ poly$

and $n :: nat \Rightarrow nat$

and $\delta :: nat \Rightarrow nat$

and $f :: nat \Rightarrow 'a\ fract$

and $k :: nat$

and $\beta :: nat \Rightarrow 'a\ fract\ +$

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 $\alpha\ i = (f\ (i - 1)) \wedge (Suc\ (\delta\ (i - 2)))$

lemma $\alpha 0: i > 1 \implies \alpha\ i = 0 \iff (i - 1) > k$
 $\langle proof \rangle$

lemma *α -char*:

assumes $3 \leq i < k + 2$

shows $\alpha\ i = (f\ (i - 1)) \wedge (Suc\ (length\ (coeffs\ (F\ (i - 2)))) - length\ (coeffs\ (F\ (i - 1))))$

$\langle proof \rangle$

definition $Q :: nat \Rightarrow 'a\ fract\ poly$ **where**

$Q\ i \equiv smult\ (\alpha\ i)\ (fst\ (pdivmod\ (F\ (i - 2))\ (F\ (i - 1))))$

lemma *beta-F-as-sum*:

assumes $3 \leq i \leq \text{Suc } k$
shows $\text{smult } (\beta \ i) \ (F \ i) = \text{smult } (\alpha \ i) \ (F \ (i - 2)) + - \ Q \ i * F \ (i - 1)$ (**is** ?t1)
⟨proof⟩

lemma **assumes** $3 \leq i \leq k$ **shows**

BT-lemma-2-21: $j < n \ i \implies \text{smult } (\alpha \ i \wedge (n \ (i - 1) - j)) \ (\text{subresultant } j \ (F \ (i - 2)) \ (F \ (i - 1)))$

$= \text{smult } ((- 1) \wedge ((n \ (i - 2) - j) * (n \ (i - 1) - j)) * (f \ (i - 1)) \wedge (\delta \ (i - 2) + \delta \ (i - 1)) * (\beta \ i) \wedge (n \ (i - 1) - j)) \ (\text{subresultant } j \ (F \ (i - 1)) \ (F \ i))$

(**is** $- \implies$?eq-21) **and**

BT-lemma-2-22: $\text{smult } (\alpha \ i \wedge (\delta \ (i - 1))) \ (\text{subresultant } (n \ i) \ (F \ (i - 2)) \ (F \ (i - 1)))$

$= \text{smult } ((- 1) \wedge ((\delta \ (i - 2) + \delta \ (i - 1)) * \delta \ (i - 1)) * f \ (i - 1) \wedge (\delta \ (i - 2) + \delta \ (i - 1)) * f \ i \wedge (\delta \ (i - 1) - 1) * (\beta \ i) \wedge \delta \ (i - 1)) \ (F \ i)$

(**is** ?eq-22) **and**

BT-lemma-2-23: $n \ i < j \implies j < n \ (i - 1) - 1 \implies \text{subresultant } j \ (F \ (i - 2)) \ (F \ (i - 1)) = 0$

(**is** $- \implies - \implies$?eq-23) **and**

BT-lemma-2-24: $\text{smult } (\alpha \ i) \ (\text{subresultant } (n \ (i - 1) - 1) \ (F \ (i - 2)) \ (F \ (i - 1)))$

$= \text{smult } ((- 1) \wedge (\delta \ (i - 2) + 1) * f \ (i - 1) \wedge (\delta \ (i - 2) + 1) * \beta \ i) \ (F \ i)$ (**is** ?eq-24)

⟨proof⟩

lemma *BT-eq-30*: $3 \leq i \implies i \leq k + 1 \implies j < n \ (i - 1) \implies$

$\text{smult } (\prod l \leftarrow [3..<i]. \alpha \ l \wedge (n \ (l - 1) - j)) \ (\text{subresultant } j \ (F \ 1) \ (F \ 2))$

$= \text{smult } (\prod l \leftarrow [3..<i]. \beta \ l \wedge (n \ (l - 1) - j) * f \ (l - 1) \wedge (\delta \ (l - 2) + \delta \ (l - 1))$

$* (- 1) \wedge ((n \ (l - 2) - j) * (n \ (l - 1) - j))) \ (\text{subresultant } j \ (F \ (i - 2)) \ (F \ (i - 1)))$

⟨proof⟩

lemma *nonzero-alphaprod*: **assumes** $i \leq k + 1$ **shows** $(\prod l \leftarrow [3..<i]. \alpha \ l \wedge (p \ l)) \neq 0$

⟨proof⟩

lemma *BT-eq-30'*: **assumes** $i: 3 \leq i \leq k + 1 \ j < n \ (i - 1)$

shows $\text{subresultant } j \ (F \ 1) \ (F \ 2)$

$= \text{smult } ((- 1) \wedge (\sum l \leftarrow [3..<i]. (n \ (l - 2) - j) * (n \ (l - 1) - j))$

$* (\prod l \leftarrow [3..<i]. (\beta \ l / \alpha \ l) \wedge (n \ (l - 1) - j)) * (\prod l \leftarrow [3..<i]. f \ (l - 1) \wedge (\delta \ (l - 2) + \delta \ (l - 1))) \ (\text{subresultant } j \ (F \ (i - 2)) \ (F \ (i - 1)))$

(**is** $- = \text{smult } (?mm * ?b * ?f) \ -$)

⟨proof⟩

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

definition $R \ j =$ (if $j = n \ 2$ then $\text{sdiv-poly } (\text{smult } ((\text{lead-coeff } G2) \wedge (\delta \ 1)) \ G2)$ ($\text{lead-coeff } G2$) else $\text{subresultant } j \ G1 \ G2$)

abbreviation $ff\ i \equiv to\text{-}fract\ (i :: 'a)$

abbreviation $ffp \equiv map\text{-}poly\ ff$

sublocale $map\text{-}poly\text{-}hom: map\text{-}poly\text{-}inj\text{-}idom\text{-}hom\ to\text{-}fract\langle proof \rangle$

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 $\gamma\ i = (-1) \wedge (\sigma\ i) * pow\text{-}int\ (f\ (i - 1))\ (1 - int\ (\delta\ (i - 1))) * (\prod l \leftarrow [3..<Suc\ i].$

$(\beta\ l / \alpha\ l) \wedge (n\ (l - 1) - n\ (i - 1) + 1) * (f\ (l - 1)) \wedge (\delta\ (l - 2) + \delta\ (l - 1)))$

definition $\Theta\ i = (-1) \wedge (\tau\ i) * pow\text{-}int\ (f\ i)\ (int\ (\delta\ (i - 1)) - 1) * (\prod l \leftarrow [3..<Suc\ i].$

$(\beta\ l / \alpha\ l) \wedge (n\ (l - 1) - n\ i) * (f\ (l - 1)) \wedge (\delta\ (l - 2) + \delta\ (l - 1)))$

lemma *fundamental-theorem-eq-4*: **assumes** $i: 3 \leq i\ i \leq k$

shows $ffp\ (R\ (n\ (i - 1) - 1)) = smult\ (\gamma\ i)\ (F\ i)$

$\langle proof \rangle$

lemma *fundamental-theorem-eq-5*: **assumes** $i: 3 \leq i\ i \leq k\ n\ i < j\ j < n\ (i - 1) - 1$

shows $R\ j = 0$

$\langle proof \rangle$

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)

$\langle proof \rangle$

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

$\langle proof \rangle$

definition $G\ i = R\ (n\ (i - 1) - 1)$

definition $H\ i = R\ (n\ i)$

lemma *gamma-delta-beta-3*: $\gamma\ 3 = (-1) \wedge (\delta\ 1 + 1) * \beta\ 3$

$\langle proof \rangle$

fun $h :: nat \Rightarrow 'a\ fract$ **where**

$h\ i = (if\ (i \leq 1)\ then\ 1\ else\ if\ i = 2\ then\ (f\ 2 \wedge \delta\ 1)\ else\ (f\ i \wedge \delta\ (i - 1) / (h\ (i - 1) \wedge (\delta\ (i - 1) - 1))))$

lemma *smult-inverse-sdiv-poly*: **assumes** *ffp*: $p \in \text{range ffp}$
and *p*: $p = \text{smult (inverse } x) q$
and *p'*: $p' = \text{sdiv-poly } q' x'$
and *xx*: $x = \text{ff } x'$
and *qq*: $q = \text{ffp } q'$
shows $p = \text{ffp } p'$
 $\langle \text{proof} \rangle$

end

locale *subresultant-prs-locale2* = *subresultant-prs-locale* $F n \delta f k \beta G1 G2$ **for**
 $F :: \text{nat} \Rightarrow 'a :: \text{idom-divide fract poly}$
and $n :: \text{nat} \Rightarrow \text{nat}$
and $\delta :: \text{nat} \Rightarrow \text{nat}$
and $f :: \text{nat} \Rightarrow 'a \text{ fract}$
and $k :: \text{nat}$
and $\beta :: \text{nat} \Rightarrow 'a \text{ fract}$
and $G1 G2 :: 'a \text{ poly} +$
assumes $\beta 3: \beta 3 = (-1) \wedge (\delta 1 + 1)$
and $\beta i: \bigwedge i. 4 \leq i \implies i \leq \text{Suc } k \implies \beta i = (-1) \wedge (\delta (i - 2) + 1) * f (i - 2)$
 $* h (i - 2) \wedge (\delta (i - 2))$
begin

lemma *B-eq-17-main*: $2 \leq i \implies i \leq k \implies$
 $h i = (-1) \wedge (n 1 + n i + i + 1) / f i$
 $* (\prod l \leftarrow [3.. < \text{Suc } (\text{Suc } i)]. (\alpha l / \beta l)) \wedge h i \neq 0$
 $\langle \text{proof} \rangle$

lemma *B-eq-17*: $2 \leq i \implies i \leq k \implies$
 $h i = (-1) \wedge (n 1 + n i + i + 1) / f i * (\prod l \leftarrow [3.. < \text{Suc } (\text{Suc } i)]. (\alpha l / \beta l))$
 $\langle \text{proof} \rangle$

lemma *B-theorem-2*: $3 \leq i \implies i \leq \text{Suc } k \implies \gamma i = 1$
 $\langle \text{proof} \rangle$

context

fixes $i :: \text{nat}$
assumes $i: 3 \leq i \leq k$

begin

lemma *B-theorem-3-b*: $\Theta i * f i = \text{ff (lead-coeff (H } i))$
 $\langle \text{proof} \rangle$

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

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

lemma $h0: i \leq k \implies h\ i \neq 0$

$\langle proof \rangle$

lemma $deg-G12: degree\ G1 \geq degree\ G2$ $\langle proof \rangle$

lemma $R0: shows\ R\ 0 = [: resultant\ G1\ G2 :]$

$\langle proof \rangle$

context

fixes $div-exp :: 'a \Rightarrow 'a \Rightarrow nat \Rightarrow 'a$

assumes $div-exp-sound: div-exp-sound\ div-exp$

begin

interpretation $div-exp-sound\ div-exp$ $\langle proof \rangle$

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 $i \geq 3\ i \leq k$

shows $F\ k = ffp\ Gk \wedge h\ k = ff\ hk \wedge (\forall\ j. i \leq j \longrightarrow j \leq k \longrightarrow F\ j \in range\ ffp \wedge \beta\ (Suc\ j) \in range\ ff)$

$\langle proof \rangle$

lemma $subresultant-prs: assumes\ res: subresultant-prs\ G1\ G2 = (Gk, hk)$

shows $F\ k = ffp\ Gk \wedge h\ k = ff\ hk \wedge (i \neq 0 \longrightarrow F\ i \in range\ ffp) \wedge (3 \leq i \longrightarrow i \leq Suc\ k \longrightarrow \beta\ i \in range\ ff)$

$\langle proof \rangle$

lemma $resultant-impl-main: resultant-impl-main\ G1\ G2 = resultant\ G1\ G2$

$\langle proof \rangle$

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** $F\ i = (if\ i = (0 :: nat)\ then\ 1$

$else\ if\ i = 1\ then\ map-poly\ to-fract\ G1\ else\ if\ i = 2\ then\ map-poly\ to-fract\ G2$

$else\ (let\ G = pseudo-mod\ (F\ (i - 2))\ (F\ (i - 1)))$

$in\ if\ F\ (i - 1) = 0 \vee G = 0\ then\ 0\ else\ smult\ (inverse\ (b\ i))\ G)$

$| b\ i = (if\ i \leq 2\ then\ 1\ else$

```

    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 $(-1)^{\dots} * \dots$ 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 G_{i-1} G_i n_{i-1} d_{1-1} h_{i-2} = (let
 g_{i-1} = lead-coeff G_{i-1} ;
 n_i = degree G_i ;
 h_{i-1} = (if $d_{1-1} = 1$ then g_{i-1} else div-exp g_{i-1} h_{i-2} d_{1-1});
 d_1 = $n_{i-1} - n_i$;
 p_{mod} = pseudo-mod G_{i-1} G_i
in (if $p_{mod} = 0$ then f (G_i , (if $d_1 = 1$ then lead-coeff G_i
else div-exp (lead-coeff G_i) h_{i-1} d_1)) else
let
 g_i = lead-coeff G_i ;
divisor = $(-1)^{(d_1 + 1)} * g_{i-1} * (h_{i-1} \wedge d_1)$;
 G_{i-p1} = sdiv-poly p_{mod} divisor
in *subresultant-prs-main-impl* f G_i G_{i-p1} n_i d_1 h_{i-1}))

definition *subresultant-prs-impl* **where**
subresultant-prs-impl f G_1 G_2 = (let
 p_{mod} = pseudo-mod G_1 G_2 ;
 n_2 = degree G_2 ;
 δ_{-1} = (degree $G_1 - n_2$);
 g_2 = lead-coeff G_2 ;
 h_2 = $g_2 \wedge \delta_{-1}$
in if $p_{mod} = 0$ then f (G_2, h_2) else let
 G_3 = $(-1)^{(\delta_{-1} + 1)} * p_{mod}$;
 g_3 = lead-coeff G_3 ;
 n_3 = degree G_3 ;
 d_2 = $n_2 - n_3$;
 p_{mod} = pseudo-mod G_2 G_3
in if $p_{mod} = 0$ then f (G_3 , if $d_2 = 1$ then g_3 else div-exp g_3 h_2 d_2)
else let divisor = $(-1)^{(d_2 + 1)} * g_2 * h_2 \wedge d_2$; G_4 = sdiv-poly p_{mod}
divisor
in *subresultant-prs-main-impl* f G_3 G_4 n_3 d_2 h_2
)
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 G_1 G_2 = f (*subresultant-prs*
 G_1 G_2)
⟨proof⟩

definition
resultant-impl-rec = *subresultant-prs-main-impl* (λ (G_k, h_k). if degree $G_k = 0$ then
 h_k else 0)
definition

resultant-impl-start = *subresultant-prs-impl* ($\lambda (Gk,hk)$. 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 $\text{gcd-impl} = \text{gcd}$

```

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 $\exists a b. a \neq 0 \wedge b \neq 0 \wedge \text{smult } a (\text{gcd } G1 \ G2) = \text{smult } b (\text{normalize } Gk)$
 ⟨*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* \longleftrightarrow *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.