

# A Verified Solver for Linear Recurrences

Manuel Eberl

December 14, 2021

## Abstract

Linear recurrences with constant coefficients are an interesting class of recurrence equations that can be solved explicitly. The most famous example are certainly the Fibonacci numbers with the equation  $f(n) = f(n - 1) + f(n - 2)$  and the quite non-obvious closed form

$$\frac{1}{\sqrt{5}}(\varphi^n - (-\varphi)^{-n})$$

where  $\varphi$  is the golden ratio.

In this work, I build on existing tools in Isabelle – such as formal power series and polynomial factorisation algorithms – to develop a theory of these recurrences and derive a fully executable solver for them that can be exported to programming languages like Haskell.

## Contents

<b>1</b>	<b>Rational formal power series</b>	<b>2</b>
1.1	Some auxiliary . . . . .	2
1.2	The type of rational formal power series . . . . .	3
<b>2</b>	<b>Falling factorial as a polynomial</b>	<b>22</b>
<b>3</b>	<b>Miscellaneous material required for linear recurrences</b>	<b>23</b>
<b>4</b>	<b>Partial Fraction Decomposition</b>	<b>28</b>
4.1	Decomposition on general Euclidean rings . . . . .	28
4.2	Specific results for polynomials . . . . .	34
<b>5</b>	<b>Factorizations of polynomials</b>	<b>36</b>
<b>6</b>	<b>Solver for rational formal power series</b>	<b>41</b>
<b>7</b>	<b>Material common to homogenous and inhomogenous linear recurrences</b>	<b>49</b>

8	Homogenous linear recurrences	50
9	Eulerian polynomials	56
10	Inhomogenous linear recurrences	59

# 1 Rational formal power series

```
theory RatFPS
imports
  Complex-Main
  HOL-Computational-Algebra.Computational-Algebra
  HOL-Computational-Algebra.Polynomial-Factorial
begin
```

## 1.1 Some auxiliary

```
abbreviation constant-term :: 'a poly  $\Rightarrow$  'a::zero
  where constant-term p  $\equiv$  coeff p 0
```

```
lemma coeff-0-mult: coeff (p * q) 0 = coeff p 0 * coeff q 0
  by (simp add: coeff-mult)
```

```
lemma coeff-0-div:
  assumes coeff p 0  $\neq$  0
  assumes (q :: 'a :: field poly) dvd p
  shows coeff (p div q) 0 = coeff p 0 div coeff q 0
proof (cases q = 0)
  case False
  from assms have p = p div q * q by simp
  also have coeff ... 0 = coeff (p div q) 0 * coeff q 0 by (simp add: coeff-0-mult)
  finally show ?thesis using assms by auto
qed simp-all
```

```
lemma coeff-0-add-fract-nonzero:
  assumes coeff (snd (quot-of-fract x)) 0  $\neq$  0 coeff (snd (quot-of-fract y)) 0  $\neq$  0
  shows coeff (snd (quot-of-fract (x + y))) 0  $\neq$  0
proof -
  define num where num = fst (quot-of-fract x) * snd (quot-of-fract y) +
    snd (quot-of-fract x) * fst (quot-of-fract y)
  define denom where denom = snd (quot-of-fract x) * snd (quot-of-fract y)
  define z where z = (num, denom)
  from assms have snd z  $\neq$  0 by (auto simp: denom-def z-def)
  then obtain d where d:
    fst z = fst (normalize-quot z) * d
    snd z = snd (normalize-quot z) * d
    d dvd fst z
    d dvd snd z
    d  $\neq$  0
  by (rule normalize-quotE')
  from assms have z: coeff (snd z) 0  $\neq$  0 by (simp add: z-def denom-def coeff-0-mult)

  have coeff (snd (quot-of-fract (x + y))) 0 = coeff (snd (normalize-quot z)) 0
    by (simp add: quot-of-fract-add Let-def case-prod-unfold z-def num-def denom-def)
```

also from  $z$  have  $\dots \neq 0$  using  $d$  by (*simp add: d coeff-0-mult*)  
 finally show *?thesis* .  
 qed

**lemma** *coeff-0-normalize-quot-nonzero* [*simp*]:  
 assumes *coeff (snd x) 0  $\neq$  0*  
 shows *coeff (snd (normalize-quot x)) 0  $\neq$  0*  
**proof** –  
 from *assms* have *snd x  $\neq$  0* by *auto*  
 then obtain *d* where  
   *fst x = fst (normalize-quot x) \* d*  
   *snd x = snd (normalize-quot x) \* d*  
   *d dvd fst x*  
   *d dvd snd x*  
   *d  $\neq$  0*  
 by (*rule normalize-quotE'*)  
 with *assms* show *?thesis* by (*auto simp: coeff-0-mult*)  
 qed

**abbreviation** *numerator* :: '*a* fract  $\Rightarrow$  '*a*::{*ring-gcd, idom-divide, semiring-gcd-mult-normalize*}

where *numerator x  $\equiv$  fst (quot-of-fract x)*

**abbreviation** *denominator* :: '*a* fract  $\Rightarrow$  '*a*::{*ring-gcd, idom-divide, semiring-gcd-mult-normalize*}

where *denominator x  $\equiv$  snd (quot-of-fract x)*

**declare** *unit-factor-snd-quot-of-fract* [*simp*]  
*normalize-snd-quot-of-fract* [*simp*]

**lemma** *constant-term-denominator-nonzero-imp-constant-term-denominator-div-gcd-nonzero*:  
*constant-term (denominator x div gcd a (denominator x))  $\neq$  0*  
**if** *constant-term (denominator x)  $\neq$  0*  
**using** *that coeff-0-normalize-quot-nonzero* [*of (a, denominator x)*]  
*normalize-quot-proj(2)* [*of denominator x a*]  
**by** *simp*

## 1.2 The type of rational formal power series

**typedef** (**overloaded**) '*a* :: *field-gcd ratfps* =  
 {*x* :: '*a* poly fract. *constant-term (denominator x)  $\neq$  0*}  
 by (*rule exI* [*of - 0*]) *simp*

**setup-lifting** *type-definition-ratfps*

**instantiation** *ratfps* :: (*field-gcd*) *idom*  
**begin**

**lift-definition** *zero-ratfps* :: '*a* *ratfps* is 0 by *simp*

**lift-definition** *one-ratfps* :: '*a* *ratfps* is 1 by *simp*

**lift-definition** *uminus-ratfps* :: 'a ratfps  $\Rightarrow$  'a ratfps **is** *uminus*  
**by** (*simp add: quot-of-fract-uminus case-prod-unfold Let-def*)

**lift-definition** *plus-ratfps* :: 'a ratfps  $\Rightarrow$  'a ratfps  $\Rightarrow$  'a ratfps **is** (+)  
**by** (*rule coeff-0-add-fract-nonzero*)

**lift-definition** *minus-ratfps* :: 'a ratfps  $\Rightarrow$  'a ratfps  $\Rightarrow$  'a ratfps **is** (-)  
**by** (*simp only: diff-conv-add-uminus, rule coeff-0-add-fract-nonzero*)  
(*simp-all add: quot-of-fract-uminus Let-def case-prod-unfold*)

**lift-definition** *times-ratfps* :: 'a ratfps  $\Rightarrow$  'a ratfps  $\Rightarrow$  'a ratfps **is** (\*)  
**by** (*simp add: quot-of-fract-mult Let-def case-prod-unfold coeff-0-mult*  
*constant-term-denominator-nonzero-imp-constant-term-denominator-div-gcd-nonzero*)

**instance**  
**by** (*standard; transfer*) (*simp-all add: ring-distrib*)

**end**

**fun** *ratfps-nth-aux* :: ('a::field) poly  $\Rightarrow$  nat  $\Rightarrow$  'a  
**where**  
*ratfps-nth-aux* p 0 = *inverse* (*coeff* p 0)  
| *ratfps-nth-aux* p n =  
- *inverse* (*coeff* p 0) \* *sum* ( $\lambda i. \text{coeff } p \ i * \text{ratfps-nth-aux } p \ (n - i)$ ) {1..n}

**lemma** *ratfps-nth-aux-correct*: *ratfps-nth-aux* p n = *natfun-inverse* (*fps-of-poly* p)  
n  
**by** (*induction* p n *rule: ratfps-nth-aux.induct*) *simp-all*

**lift-definition** *ratfps-nth* :: 'a :: field-gcd ratfps  $\Rightarrow$  nat  $\Rightarrow$  'a **is**  
 $\lambda x \ n. \text{let } (a,b) = \text{quot-of-fract } x$   
in  $(\sum_{i=0..n} \text{coeff } a \ i * \text{ratfps-nth-aux } b \ (n - i))$  .

**lift-definition** *ratfps-subdegree* :: 'a :: field-gcd ratfps  $\Rightarrow$  nat **is**  
 $\lambda x. \text{poly-subdegree } (\text{fst } (\text{quot-of-fract } x))$  .

**context**  
**includes** *lifting-syntax*  
**begin**

**lemma** *RatFPS-parametric*: (*rel-prod* (=) (=)  $\implies$  (=))  
( $\lambda(p,q). \text{if } \text{coeff } q \ 0 = 0 \text{ then } 0 \text{ else } \text{quot-to-fract } (p, q)$ )  
( $\lambda(p,q). \text{if } \text{coeff } q \ 0 = 0 \text{ then } 0 \text{ else } \text{quot-to-fract } (p, q)$ )  
**by** *transfer-prover*

**end**

**lemma** *normalize-quot-quot-of-fract* [*simp*]:  
*normalize-quot* (*quot-of-fract*  $x$ ) = *quot-of-fract*  $x$   
**by** (*rule normalize-quot-id*, *rule quot-of-fract-in-normalized-fracts*)

**context**  
**assumes** *SORT-CONSTRAINT*('a::field-gcd)  
**begin**

**lift-definition** *quot-of-ratfps* :: 'a ratfps  $\Rightarrow$  ('a poly  $\times$  'a poly) **is**  
*quot-of-fract* :: 'a poly fract  $\Rightarrow$  ('a poly  $\times$  'a poly) .

**lift-definition** *quot-to-ratfps* :: ('a poly  $\times$  'a poly)  $\Rightarrow$  'a ratfps **is**  
 $\lambda(x,y).$  *let* ( $x',y'$ ) = *normalize-quot* ( $x,y$ )  
*in if* *coeff*  $y' 0 = 0$  *then*  $0$  *else* *quot-to-fract* ( $x',y'$ )  
**by** (*simp add: case-prod-unfold Let-def quot-of-fract-quot-to-fract*)

**lemma** *quot-to-ratfps-quot-of-ratfps* [*code abstype*]:  
*quot-to-ratfps* (*quot-of-ratfps*  $x$ ) =  $x$   
**by** *transfer* (*simp add: case-prod-unfold Let-def*)

**lemma** *coeff-0-snd-quot-of-ratfps-nonzero* [*simp*]:  
*coeff* (*snd* (*quot-of-ratfps*  $x$ ))  $0 \neq 0$   
**by** *transfer simp*

**lemma** *quot-of-ratfps-quot-to-ratfps*:  
*coeff* (*snd*  $x$ )  $0 \neq 0 \implies x \in$  *normalized-fracts*  $\implies$  *quot-of-ratfps* (*quot-to-ratfps*  
 $x$ ) =  $x$   
**by** *transfer* (*simp add: Let-def case-prod-unfold coeff-0-normalize-quot-nonzero*  
*quot-of-fract-quot-to-fract normalize-quot-id*)

**lemma** *quot-of-ratfps-0* [*simp, code abstract*]: *quot-of-ratfps*  $0 = (0, 1)$   
**by** *transfer simp-all*

**lemma** *quot-of-ratfps-1* [*simp, code abstract*]: *quot-of-ratfps*  $1 = (1, 1)$   
**by** *transfer simp-all*

**lift-definition** *ratfps-of-poly* :: 'a poly  $\Rightarrow$  'a ratfps **is**  
*to-fract* :: 'a poly  $\Rightarrow$  -  
**by** *transfer simp*

**lemma** *ratfps-of-poly-code* [*code abstract*]:  
*quot-of-ratfps* (*ratfps-of-poly*  $p$ ) = ( $p, 1$ )  
**by** *transfer' simp*

**lemmas** *zero-ratfps-code* = *quot-of-ratfps-0*

**lemmas** *one-ratfps-code* = *quot-of-ratfps-1*

**lemma** *uminus-ratfps-code* [*code abstract*]:

*quot-of-ratfps*  $(- x) = (\text{let } (a, b) = \text{quot-of-ratfps } x \text{ in } (-a, b))$   
**by** *transfer* (rule *quot-of-fract-uminus*)

**lemma** *plus-ratfps-code* [*code abstract*]:  
*quot-of-ratfps*  $(x + y) =$   
 $(\text{let } (a,b) = \text{quot-of-ratfps } x; (c,d) = \text{quot-of-ratfps } y$   
 $\text{in } \text{normalize-quot } (a * d + b * c, b * d))$   
**by** *transfer'* (rule *quot-of-fract-add*)

**lemma** *minus-ratfps-code* [*code abstract*]:  
*quot-of-ratfps*  $(x - y) =$   
 $(\text{let } (a,b) = \text{quot-of-ratfps } x; (c,d) = \text{quot-of-ratfps } y$   
 $\text{in } \text{normalize-quot } (a * d - b * c, b * d))$   
**by** *transfer'* (rule *quot-of-fract-diff*)

**definition** *ratfps-cutoff*  $:: \text{nat} \Rightarrow 'a :: \text{field-gcd ratfps} \Rightarrow 'a \text{ poly}$  **where**  
*ratfps-cutoff*  $n x = \text{poly-of-list } (\text{map } (\text{ratfps-nth } x) [0..<n])$

**definition** *ratfps-shift*  $:: \text{nat} \Rightarrow 'a :: \text{field-gcd ratfps} \Rightarrow 'a \text{ ratfps}$  **where**  
*ratfps-shift*  $n x = (\text{let } (a, b) = \text{quot-of-ratfps } (x - \text{ratfps-of-poly } (\text{ratfps-cutoff } n$   
 $x))$   
 $\text{in } \text{quot-to-ratfps } (\text{poly-shift } n a, b))$

**lemma** *times-ratfps-code* [*code abstract*]:  
*quot-of-ratfps*  $(x * y) =$   
 $(\text{let } (a,b) = \text{quot-of-ratfps } x; (c,d) = \text{quot-of-ratfps } y;$   
 $(e,f) = \text{normalize-quot } (a,d); (g,h) = \text{normalize-quot } (c,b)$   
 $\text{in } (e*g, f*h))$   
**by** *transfer'* (rule *quot-of-fract-mult*)

**lemma** *ratfps-nth-code* [*code*]:  
*ratfps-nth*  $x n =$   
 $(\text{let } (a,b) = \text{quot-of-ratfps } x$   
 $\text{in } \sum i = 0..n. \text{coeff } a i * \text{ratfps-nth-aux } b (n - i))$   
**by** *transfer' simp*

**lemma** *ratfps-subdegree-code* [*code*]:  
*ratfps-subdegree*  $x = \text{poly-subdegree } (\text{fst } (\text{quot-of-ratfps } x))$   
**by** *transfer simp*

**end**

**instantiation** *ratfps*  $:: (\text{field-gcd}) \text{ inverse}$   
**begin**

**lift-definition** *inverse-ratfps*  $:: 'a \text{ ratfps} \Rightarrow 'a \text{ ratfps}$  **is**  
 $\lambda x. \text{let } (a,b) = \text{quot-of-fract } x$   
 $\text{in } \text{if } \text{coeff } a 0 = 0 \text{ then } 0 \text{ else } \text{inverse } x$   
**by** (*auto simp: case-prod-unfold Let-def quot-of-fract-inverse*)

**lift-definition** *divide-ratfps* :: 'a ratfps  $\Rightarrow$  'a ratfps  $\Rightarrow$  'a ratfps **is**  
 $\lambda f g.$  (if  $g = 0$  then  $0$  else  
 let  $n = \text{ratfps-subdegree } g$ ;  $h = \text{ratfps-shift } n \ g$   
 in  $\text{ratfps-shift } n \ (f * \text{inverse } h)$ ).

**instance ..**  
**end**

**lemma** *ratfps-inverse-code* [*code abstract*]:  
 $\text{quot-of-ratfps } (\text{inverse } x) =$   
 (let  $(a,b) = \text{quot-of-ratfps } x$   
 in if  $\text{coeff } a \ 0 = 0$  then  $(0, 1)$   
 else let  $u = \text{unit-factor } a$  in  $(b \ \text{div } u, a \ \text{div } u)$ )  
**by** *transfer'* (*simp-all add: Let-def case-prod-unfold quot-of-fract-inverse*)

**instantiation** *ratfps* :: (*equal*) *equal*  
**begin**

**definition** *equal-ratfps* :: 'a ratfps  $\Rightarrow$  'a ratfps  $\Rightarrow$  bool **where**  
 [*simp*]:  $\text{equal-ratfps } x \ y \longleftrightarrow x = y$

**instance by** *standard simp*

**end**

**lemma** *quot-of-fract-eq-iff* [*simp*]:  $\text{quot-of-fract } x = \text{quot-of-fract } y \longleftrightarrow x = y$   
**by** *transfer* (*auto simp: normalize-quot-eq-iff*)

**lemma** *equal-ratfps-code* [*code*]:  $\text{HOL.equal } x \ y \longleftrightarrow \text{quot-of-ratfps } x = \text{quot-of-ratfps } y$   
**unfolding** *equal-ratfps-def* **by** *transfer simp*

**lemma** *fps-of-poly-quot-normalize-quot* [*simp*]:  
 $\text{fps-of-poly } (\text{fst } (\text{normalize-quot } x)) / \text{fps-of-poly } (\text{snd } (\text{normalize-quot } x)) =$   
 $\text{fps-of-poly } (\text{fst } x) / \text{fps-of-poly } (\text{snd } x)$   
**if**  $(\text{snd } x :: 'a :: \text{field-gcd poly}) \neq 0$

**proof** –

**from** *that* **obtain**  $d$  **where**  $\text{fst } x = \text{fst } (\text{normalize-quot } x) * d$   
**and**  $\text{snd } x = \text{snd } (\text{normalize-quot } x) * d$  **and**  $d \neq 0$   
**by** (*rule normalize-quotE'*)  
**then show** *?thesis*  
**by** (*simp add: fps-of-poly-mult*)

**qed**

**lemma** *fps-of-poly-quot-normalize-quot'* [*simp*]:  
 $\text{fps-of-poly } (\text{fst } (\text{normalize-quot } x)) / \text{fps-of-poly } (\text{snd } (\text{normalize-quot } x)) =$   
 $\text{fps-of-poly } (\text{fst } x) / \text{fps-of-poly } (\text{snd } x)$   
**if**  $\text{coeff } (\text{snd } x) \ 0 \neq (0 :: 'a :: \text{field-gcd})$



using *that* by (auto intro: fps-of-poly-quot-normalize-quot)

**lift-definition** *fps-of-ratfps* :: 'a :: field-gcd ratfps  $\Rightarrow$  'a fps is  
 $\lambda x. \text{fps-of-poly } (\text{numerator } x) / \text{fps-of-poly } (\text{denominator } x) .$

**lemma** *fps-of-ratfps-altdef*:  
*fps-of-ratfps*  $x = (\text{case quot-of-ratfps } x \text{ of } (a, b) \Rightarrow \text{fps-of-poly } a / \text{fps-of-poly } b)$   
by *transfer* (*simp add: case-prod-unfold*)

**lemma** *fps-of-ratfps-ratfps-of-poly* [*simp*]: *fps-of-ratfps* (*ratfps-of-poly*  $p$ ) = *fps-of-poly*  $p$   
by *transfer simp*

**lemma** *fps-of-ratfps-0* [*simp*]: *fps-of-ratfps*  $0 = 0$   
by *transfer simp*

**lemma** *fps-of-ratfps-1* [*simp*]: *fps-of-ratfps*  $1 = 1$   
by *transfer simp*

**lemma** *fps-of-ratfps-uminus* [*simp*]: *fps-of-ratfps*  $(-x) = - \text{fps-of-ratfps } x$   
by *transfer* (*simp add: quot-of-fract-uminus case-prod-unfold Let-def fps-of-poly-simps dvd-neg-div*)

**lemma** *fps-of-ratfps-add* [*simp*]: *fps-of-ratfps*  $(x + y) = \text{fps-of-ratfps } x + \text{fps-of-ratfps } y$   
by *transfer* (*simp add: quot-of-fract-add Let-def case-prod-unfold fps-of-poly-simps*)

**lemma** *fps-of-ratfps-diff* [*simp*]: *fps-of-ratfps*  $(x - y) = \text{fps-of-ratfps } x - \text{fps-of-ratfps } y$   
by *transfer* (*simp add: quot-of-fract-diff Let-def case-prod-unfold fps-of-poly-simps*)

**lemma** *is-unit-div-div-commute*: *is-unit*  $b \Longrightarrow \text{is-unit } c \Longrightarrow a \text{ div } b \text{ div } c = a \text{ div } c \text{ div } b$   
by (*metis is-unit-div-mult2-eq mult.commute*)

**lemma** *fps-of-ratfps-mult* [*simp*]: *fps-of-ratfps*  $(x * y) = \text{fps-of-ratfps } x * \text{fps-of-ratfps } y$

**proof** (*transfer, goal-cases*)

case  $(1 \ x \ y)$

moreover **define**  $x' \ y'$  where  $x' = \text{quot-of-fract } x$  and  $y' = \text{quot-of-fract } y$

ultimately **have** *assms*: *coeff* (*snd*  $x'$ )  $0 \neq 0$  *coeff* (*snd*  $y'$ )  $0 \neq 0$

by *simp-all*

moreover **define**  $w \ z$  where  $w = \text{normalize-quot } (\text{fst } x', \text{snd } y')$  and  $z = \text{normalize-quot } (\text{fst } y', \text{snd } x')$

ultimately **have** *unit*: *coeff* (*snd*  $x'$ )  $0 \neq 0$  *coeff* (*snd*  $y'$ )  $0 \neq 0$

*coeff* (*snd*  $w$ )  $0 \neq 0$  *coeff* (*snd*  $z$ )  $0 \neq 0$

by (*simp-all add: coeff-0-normalize-quot-nonzero*)

**have** *fps-of-poly* (*fst*  $w * \text{fst } z$ ) / *fps-of-poly* (*snd*  $w * \text{snd } z$ ) =  
(*fps-of-poly* (*fst*  $w$ ) / *fps-of-poly* (*snd*  $w$ )) \*

$(fps\text{-of-poly } (fst\ z) / fps\text{-of-poly } (snd\ z))$  (**is** - = ?A \* ?B)  
**by** (*simp add: is-unit-div-mult2-eq fps-of-poly-mult unit-div-mult-swap unit-div-commute unit*)  
**also have** ... =  $(fps\text{-of-poly } (fst\ x') / fps\text{-of-poly } (snd\ x')) * (fps\text{-of-poly } (fst\ y') / fps\text{-of-poly } (snd\ y'))$  **using** *unit*  
**by** (*simp add: w-def z-def unit-div-commute unit-div-mult-swap is-unit-div-div-commute*)  
**finally show** ?case  
**by** (*simp add: w-def z-def x'-def y'-def Let-def case-prod-unfold quot-of-fract-mult mult-ac*)  
**qed**

**lemma** *div-const-unit-poly*:  $is\text{-unit } c \implies p\ div\ [:c:] = smult\ (1\ div\ c)\ p$   
**by** (*simp add: is-unit-const-poly-iff unit-eq-div1*)

**lemma** *normalize-field*:  
 $normalize\ (x :: 'a :: \{normalization\text{-semidom, field}\}) = (if\ x = 0\ then\ 0\ else\ 1)$   
**by** (*auto simp: normalize-1-iff dvd-field-iff*)

**lemma** *unit-factor-field* [*simp*]:  
 $unit\text{-factor } (x :: 'a :: \{normalization\text{-semidom, field}\}) = x$   
**using** *unit-factor-mult-normalize[of x] normalize-field[of x]*  
**by** (*simp split: if-splits*)

**lemma** *fps-of-poly-normalize-field*:  
 $fps\text{-of-poly } (normalize\ (p :: 'a :: \{field, normalization\text{-semidom}\}\ poly)) = fps\text{-of-poly } p * fps\text{-const } (inverse\ (lead\text{-coeff } p))$   
**by** (*cases p = 0*)  
*(simp-all add: normalize-poly-def div-const-unit-poly divide-simps dvd-field-iff)*

**lemma** *unit-factor-poly-altdef*:  $unit\text{-factor } p = monom\ (unit\text{-factor } (lead\text{-coeff } p))\ 0$   
**by** (*simp add: unit-factor-poly-def monom-altdef*)

**lemma** *div-const-poly*:  $p\ div\ [:c::'a::field:] = smult\ (inverse\ c)\ p$   
**by** (*cases c = 0*) (*simp-all add: unit-eq-div1 is-unit-triv*)

**lemma** *fps-of-ratfps-inverse* [*simp*]:  $fps\text{-of-ratfps } (inverse\ x) = inverse\ (fps\text{-of-ratfps } x)$

**proof** (*transfer, goal-cases*)

**case** (1 x)

**hence**  $smult\ (lead\text{-coeff } (fst\ (quot\text{-of-fract } x)))\ (snd\ (quot\text{-of-fract } x))\ div\ unit\text{-factor } (fst\ (quot\text{-of-fract } x)) = snd\ (quot\text{-of-fract } x)$

**if**  $fst\ (quot\text{-of-fract } x) \neq 0$  **using** *that*

**by** (*simp add: unit-factor-poly-altdef monom-0 div-const-poly*)

**with** 1 **show** ?case

**by** (*auto simp: Let-def case-prod-unfold fps-divide-unit fps-inverse-mult quot-of-fract-inverse mult-ac*)

*fps-of-poly-simps fps-const-inverse*

*fps-of-poly-normalize-field div-smult-left [symmetric]*)

**qed**

**context**

**includes** *fps-notation*

**begin**

**lemma** *ratfps-nth-altdef*:  $\text{ratfps-nth } x \ n = \text{fps-of-ratfps } x \ \$ \ n$

**by** *transfer*

(*simp-all add: case-prod-unfold fps-divide-unit fps-times-def fps-inverse-def ratfps-nth-aux-correct Let-def*)

**lemma** *fps-of-ratfps-is-unit*:  $\text{fps-of-ratfps } a \ \$ \ 0 \neq 0 \iff \text{ratfps-nth } a \ 0 \neq 0$

**by** (*simp add: ratfps-nth-altdef*)

**lemma** *ratfps-nth-0* [*simp*]:  $\text{ratfps-nth } 0 \ n = 0$

**by** (*simp add: ratfps-nth-altdef*)

**lemma** *fps-of-ratfps-cases*:

**obtains**  $p \ q$  **where**  $\text{coeff } q \ 0 \neq 0 \ \text{fps-of-ratfps } f = \text{fps-of-poly } p \ / \ \text{fps-of-poly } q$

**by** (*rule that[of snd (quot-of-ratfps f) fst (quot-of-ratfps f)]*)

(*simp-all add: fps-of-ratfps-altdef case-prod-unfold*)

**lemma** *fps-of-ratfps-cutoff* [*simp*]:

$\text{fps-of-poly } (\text{ratfps-cutoff } n \ x) = \text{fps-cutoff } n \ (\text{fps-of-ratfps } x)$

**by** (*simp add: fps-eq-iff ratfps-cutoff-def nth-default-def ratfps-nth-altdef*)

**lemma** *subdegree-fps-of-ratfps*:

$\text{subdegree } (\text{fps-of-ratfps } x) = \text{ratfps-subdegree } x$

**by** *transfer* (*simp-all add: case-prod-unfold subdegree-div-unit poly-subdegree-def*)

**lemma** *ratfps-subdegree-altdef*:

$\text{ratfps-subdegree } x = \text{subdegree } (\text{fps-of-ratfps } x)$

**using** *subdegree-fps-of-ratfps ..*

**end**

**code-datatype** *fps-of-ratfps*

**lemma** *fps-zero-code* [*code*]:  $0 = \text{fps-of-ratfps } 0$  **by** *simp*

**lemma** *fps-one-code* [*code*]:  $1 = \text{fps-of-ratfps } 1$  **by** *simp*

**lemma** *fps-const-code* [*code*]:  $\text{fps-const } c = \text{fps-of-poly } [:c:]$  **by** *simp*

**lemma** *fps-of-poly-code* [*code*]:  $\text{fps-of-poly } p = \text{fps-of-ratfps } (\text{ratfps-of-poly } p)$  **by** *simp*

**lemma** *fps-X-code* [*code*]:  $\text{fps-X} = \text{fps-of-ratfps } (\text{ratfps-of-poly } [:0,1:])$  **by** *simp*

**lemma** *fps-nth-code* [*code*]:  $\text{fps-nth } (\text{fps-of-ratfps } x) \ n = \text{ratfps-nth } x \ n$   
**by** (*simp add: ratfps-nth-altdef*)

**lemma** *fps-uminus-code* [*code*]:  $-\ \text{fps-of-ratfps } x = \text{fps-of-ratfps } (-x)$  **by** *simp*

**lemma** *fps-add-code* [*code*]:  $\text{fps-of-ratfps } x + \text{fps-of-ratfps } y = \text{fps-of-ratfps } (x + y)$  **by** *simp*

**lemma** *fps-diff-code* [*code*]:  $\text{fps-of-ratfps } x - \text{fps-of-ratfps } y = \text{fps-of-ratfps } (x - y)$   
**by** *simp*

**lemma** *fps-mult-code* [*code*]:  $\text{fps-of-ratfps } x * \text{fps-of-ratfps } y = \text{fps-of-ratfps } (x * y)$   
**by** *simp*

**lemma** *fps-inverse-code* [*code*]:  $\text{inverse } (\text{fps-of-ratfps } x) = \text{fps-of-ratfps } (\text{inverse } x)$   
**by** *simp*

**lemma** *fps-cutoff-code* [*code*]:  $\text{fps-cutoff } n \ (\text{fps-of-ratfps } x) = \text{fps-of-poly } (\text{ratfps-cutoff } n \ x)$   
**by** *simp*

**lemmas** *subdegree-code* [*code*] = *subdegree-fps-of-ratfps*

**lemma** *fractrel-normalize-quot*:  
 $\text{fractrel } p \ p \implies \text{fractrel } q \ q \implies$   
 $\text{fractrel } (\text{normalize-quot } p) \ (\text{normalize-quot } q) \longleftrightarrow \text{fractrel } p \ q$   
**by** (*subst fractrel-normalize-quot-left fractrel-normalize-quot-right, simp*)<sup>+</sup> (*rule refl*)

**lemma** *fps-of-ratfps-eq-iff* [*simp*]:  
 $\text{fps-of-ratfps } p = \text{fps-of-ratfps } q \longleftrightarrow p = q$   
**proof** –  
{  
  **fix**  $p \ q :: 'a \ \text{poly } \text{fract}$   
  **assume**  $\text{fractrel } (\text{quot-of-fract } p) \ (\text{quot-of-fract } q)$   
  **hence**  $p = q$  **by** *transfer (simp only: fractrel-normalize-quot)*  
} **note**  $A = \text{this}$   
**show** *?thesis*  
**by** *transfer (auto simp: case-prod-unfold unit-eq-div1 unit-eq-div2 unit-div-commute intro: A)*  
**qed**

**lemma** *fps-of-ratfps-eq-zero-iff* [*simp*]:  
 $\text{fps-of-ratfps } p = 0 \longleftrightarrow p = 0$   
**by** (*simp del: fps-of-ratfps-0 add: fps-of-ratfps-0 [symmetric]*)

**lemma** *unit-factor-snd-quot-of-ratfps* [*simp*]:

*unit-factor* (*snd* (*quot-of-ratfps* *x*)) = 1

**by** *transfer simp*

**lemma** *poly-shift-times-monom-le*:

$n \leq m \implies \text{poly-shift } n (\text{monom } c \ m * p) = \text{monom } c \ (m - n) * p$

**by** (*intro poly-eqI*) (*auto simp: coeff-monom-mult coeff-poly-shift*)

**lemma** *poly-shift-times-monom-ge*:

$n \geq m \implies \text{poly-shift } n (\text{monom } c \ m * p) = \text{smult } c \ (\text{poly-shift } (n - m) \ p)$

**by** (*intro poly-eqI*) (*auto simp: coeff-monom-mult coeff-poly-shift*)

**lemma** *poly-shift-times-monom*:

*poly-shift* *n* (*monom* *c* *n* \* *p*) = *smult* *c* *p*

**by** (*intro poly-eqI*) (*auto simp: coeff-monom-mult coeff-poly-shift*)

**lemma** *monom-times-poly-shift*:

**assumes** *poly-subdegree* *p*  $\geq$  *n*

**shows** *monom* *c* *n* \* *poly-shift* *n* *p* = *smult* *c* *p* (**is** *?lhs* = *?rhs*)

**proof** (*intro poly-eqI*)

**fix** *k*

**show** *coeff* *?lhs* *k* = *coeff* *?rhs* *k*

**proof** (*cases*  $k < n$ )

**case** *True*

**with** *assms* **have**  $k < \text{poly-subdegree } p$  **by** *simp*

**hence** *coeff* *p* *k* = 0 **by** (*simp add: coeff-less-poly-subdegree*)

**thus** *?thesis* **by** (*auto simp: coeff-monom-mult coeff-poly-shift*)

**qed** (*auto simp: coeff-monom-mult coeff-poly-shift*)

**qed**

**lemma** *monom-times-poly-shift'*:

**assumes** *poly-subdegree* *p*  $\geq$  *n*

**shows** *monom* (1 :: 'a :: *comm-semiring-1*) *n* \* *poly-shift* *n* *p* = *p*

**by** (*simp add: monom-times-poly-shift[OF assms]*)

**lemma** *subdegree-minus-cutoff-ge*:

**assumes** *f* - *fps-cutoff* *n* (*f* :: 'a :: *ab-group-add fps*)  $\neq$  0

**shows** *subdegree* (*f* - *fps-cutoff* *n* *f*)  $\geq$  *n*

**using** *assms* **by** (*rule subdegree-geI*) *simp-all*

**lemma** *fps-shift-times-X-power'*: *fps-shift* *n* (*fps-X*  $\wedge$  *n* \* *f* :: 'a :: *comm-ring-1* *fps*) = *f*

**using** *fps-shift-times-fps-X-power'[of n f]* **by** (*simp add: mult.commute*)

**lemma**

*ratfps-shift-code* [*code abstract*]:

*quot-of-ratfps* (*ratfps-shift* *n* *x*) =

(*let* (*a*, *b*) = *quot-of-ratfps* (*x* - *ratfps-of-poly* (*ratfps-cutoff* *n* *x*))

*in* (*poly-shift* *n* *a*, *b*)) (**is** *?lhs1* = *?rhs1*) **and**

```

fps-of-ratfps-shift [simp]:
  fps-of-ratfps (ratfps-shift n x) = fps-shift n (fps-of-ratfps x)
proof –
  include fps-notation
  define x' where x' = ratfps-of-poly (ratfps-cutoff n x)
  define y where y = quot-of-ratfps (x – x')

  have coprime (fst y) (snd y) unfolding y-def
    by transfer (rule coprime-quot-of-fract)
  also have fst-y: fst y = monom 1 n * poly-shift n (fst y)
  proof (cases x = x')
    case False
      have poly-subdegree (fst y) = subdegree (fps-of-poly (fst y))
        by (simp add: poly-subdegree-def)
      also have ... = subdegree (fps-of-poly (fst y) / fps-of-poly (snd y))
        by (subst subdegree-div-unit) (simp-all add: y-def)
      also have fps-of-poly (fst y) / fps-of-poly (snd y) = fps-of-ratfps (x – x')
        unfolding y-def by transfer (simp add: case-prod-unfold)
      also from False have subdegree ... ≥ n
      proof (intro subdegree-geI)
        fix k assume k < n
        thus fps-of-ratfps (x – x') $ k = 0 by (simp add: x'-def)
      qed simp-all
      finally show ?thesis by (rule monom-times-poly-shift' [symmetric])
    qed (simp-all add: y-def)
  finally have coprime: coprime (poly-shift n (fst y)) (snd y)
    by simp

  have quot-of-ratfps (ratfps-shift n x) =
    quot-of-ratfps (quot-to-ratfps (poly-shift n (fst y), snd y))
    by (simp add: ratfps-shift-def Let-def case-prod-unfold x'-def y-def)
  also from coprime have ... = (poly-shift n (fst y), snd y)
    by (intro quot-of-ratfps-quot-to-ratfps) (simp-all add: y-def normalized-fracts-def)
  also have ... = ?rhs1 by (simp add: case-prod-unfold Let-def y-def x'-def)
  finally show eq: ?lhs1 = ?rhs1 .

  have fps-shift n (fps-of-ratfps x) = fps-shift n (fps-of-ratfps (x – x'))
    by (intro fps-ext) (simp-all add: x'-def)
  also have fps-of-ratfps (x – x') = fps-of-poly (fst y) / fps-of-poly (snd y)
    by (simp add: fps-of-ratfps-altdef y-def case-prod-unfold)
  also have fps-shift n ... = fps-of-ratfps (ratfps-shift n x)
    by (subst fst-y, subst fps-of-poly-mult, subst unit-div-mult-swap [symmetric])
    (simp-all add: y-def fps-of-poly-monom fps-shift-times-X-power'' eq
      fps-of-ratfps-altdef case-prod-unfold Let-def x'-def)
  finally show fps-of-ratfps (ratfps-shift n x) = fps-shift n (fps-of-ratfps x) ..
qed

lemma fps-shift-code [code]: fps-shift n (fps-of-ratfps x) = fps-of-ratfps (ratfps-shift
n x)

```

by *simp*

**instantiation** *fps* :: (*equal*) *equal*  
**begin**

**definition** *equal-fps* :: '*a fps*  $\Rightarrow$  '*a fps*  $\Rightarrow$  *bool* **where**  
[*simp*]: *equal-fps* *f g*  $\longleftrightarrow$  *f = g*

**instance** by *standard simp-all*

**end**

**lemma** *equal-fps-code* [*code*]: *HOL.equal* (*fps-of-ratfps* *f*) (*fps-of-ratfps* *g*)  $\longleftrightarrow$  *f = g*  
by *simp*

**lemma** *fps-of-ratfps-divide* [*simp*]:  
*fps-of-ratfps* (*f div g*) = *fps-of-ratfps* *f* *div* *fps-of-ratfps* *g*  
**unfolding** *fps-divide-def* *Let-def* **by** *transfer'* (*simp* *add*: *Let-def ratfps-subdegree-altdef*)

**lemma** *ratfps-eqI*: *fps-of-ratfps* *x = fps-of-ratfps* *y*  $\Longrightarrow$  *x = y* **by** *simp*

**instance** *ratfps* :: (*field-gcd*) *algebraic-semidom*  
**by** *standard* (*auto* *intro*: *ratfps-eqI*)

**lemma** *fps-of-ratfps-dvd* [*simp*]:  
*fps-of-ratfps* *x dvd fps-of-ratfps* *y*  $\longleftrightarrow$  *x dvd y*  
**proof**  
**assume** *fps-of-ratfps* *x dvd fps-of-ratfps* *y*  
**hence** *fps-of-ratfps* *y = fps-of-ratfps* *y div fps-of-ratfps* *x* \* *fps-of-ratfps* *x* **by** *simp*  
**also have**  $\dots =$  *fps-of-ratfps* (*y div x* \* *x*) **by** *simp*  
**finally have** *y = y div x* \* *x* **by** (*subst* (*asm*) *fps-of-ratfps-eq-iff*)  
**thus** *x dvd y* **by** (*intro* *dvdI*[*of - - y div x*]) (*simp* *add*: *mult-ac*)  
**next**  
**assume** *x dvd y*  
**hence** *y = y div x* \* *x* **by** *simp*  
**also have** *fps-of-ratfps*  $\dots =$  *fps-of-ratfps* (*y div x*) \* *fps-of-ratfps* *x* **by** *simp*  
**finally show** *fps-of-ratfps* *x dvd fps-of-ratfps* *y* **by** (*simp* *del*: *fps-of-ratfps-divide*)  
**qed**

**lemma** *is-unit-ratfps-iff* [*simp*]:  
*is-unit* *x*  $\longleftrightarrow$  *ratfps-nth* *x* *0*  $\neq$  *0*  
**proof**  
**assume** *is-unit* *x*  
**then obtain** *y* **where** *1 = x* \* *y* **by** (*auto* *elim*!: *dvdE*)  
**hence** *1 = fps-of-ratfps* (*x* \* *y*) **by** (*simp* *del*: *fps-of-ratfps-mult*)  
**also have**  $\dots =$  *fps-of-ratfps* *x* \* *fps-of-ratfps* *y* **by** *simp*  
**finally have** *is-unit* (*fps-of-ratfps* *x*) **by** (*rule* *dvdI*[*of - - fps-of-ratfps* *y*])  
**thus** *ratfps-nth* *x* *0*  $\neq$  *0* **by** (*simp* *add*: *ratfps-nth-altdef*)

**next**  
**assume**  $\text{ratfps-nth } x \ 0 \neq 0$   
**hence**  $\text{fps-of-ratfps } (x * \text{inverse } x) = 1$   
**by** (*simp add: ratfps-nth-altdef inverse-mult-eq-1*)  
**also have**  $\dots = \text{fps-of-ratfps } 1$  **by** *simp*  
**finally have**  $x * \text{inverse } x = 1$  **by** (*subst (asm) fps-of-ratfps-eq-iff*)  
**thus is-unit**  $x$  **by** (*intro dvdI[of - - inverse x] simp-all*)  
**qed**

**instantiation**  $\text{ratfps} :: (\text{field-gcd}) \text{ normalization-semidom}$   
**begin**

**definition**  $\text{unit-factor-ratfps} :: 'a \text{ ratfps} \Rightarrow 'a \text{ ratfps}$  **where**  
 $\text{unit-factor } x = \text{ratfps-shift } (\text{ratfps-subdegree } x) \ x$

**definition**  $\text{normalize-ratfps} :: 'a \text{ ratfps} \Rightarrow 'a \text{ ratfps}$  **where**  
 $\text{normalize } x = (\text{if } x = 0 \text{ then } 0 \text{ else } \text{ratfps-of-poly } (\text{monom } 1 \ (\text{ratfps-subdegree } x)))$

**lemma**  $\text{fps-of-ratfps-unit-factor}$  [*simp*]:  
 $\text{fps-of-ratfps } (\text{unit-factor } x) = \text{unit-factor } (\text{fps-of-ratfps } x)$   
**unfolding**  $\text{unit-factor-ratfps-def}$  **by** (*simp add: ratfps-subdegree-altdef*)

**lemma**  $\text{fps-of-ratfps-normalize}$  [*simp*]:  
 $\text{fps-of-ratfps } (\text{normalize } x) = \text{normalize } (\text{fps-of-ratfps } x)$   
**unfolding**  $\text{normalize-ratfps-def}$  **by** (*simp add: fps-of-poly-monom ratfps-subdegree-altdef*)

**instance proof**

**show**  $\text{unit-factor } x * \text{normalize } x = x \ \text{normalize } (0 :: 'a \text{ ratfps}) = 0$   
 $\text{unit-factor } (0 :: 'a \text{ ratfps}) = 0$  **for**  $x :: 'a \text{ ratfps}$   
**by** (*rule ratfps-eqI, simp add: ratfps-subdegree-code*  
 $\text{del: fps-of-ratfps-eq-iff fps-unit-factor-def fps-normalize-def}$ )  
**show**  $\text{is-unit } (\text{unit-factor } a)$  **if**  $a \neq 0$  **for**  $a :: 'a \text{ ratfps}$   
**using that by** (*auto simp: ratfps-nth-altdef*)  
**fix**  $a \ b :: 'a \text{ ratfps}$   
**assume**  $\text{is-unit } a$   
**thus**  $\text{unit-factor } (a * b) = a * \text{unit-factor } b$   
**by** (*intro ratfps-eqI, unfold fps-of-ratfps-unit-factor fps-of-ratfps-mult,*  
 $\text{subst unit-factor-mult-unit-left}$ ) (*auto simp: ratfps-nth-altdef*)  
**show**  $\text{unit-factor } a = a$  **if**  $\text{is-unit } a$  **for**  $a :: 'a \text{ ratfps}$   
**by** (*rule ratfps-eqI*) (*insert that, auto simp: fps-of-ratfps-is-unit*)  
**qed**

**end**

**instance**  $\text{ratfps} :: (\text{field-gcd}) \text{ normalization-semidom-multiplicative}$   
**proof**

**show**  $\text{unit-factor } (a * b) = \text{unit-factor } a * \text{unit-factor } b$  **for**  $a \ b :: 'a \text{ ratfps}$   
**by** (*rule ratfps-eqI, insert unit-factor-mult[of fps-of-ratfps a fps-of-ratfps b]*)



(*simp del: fps-of-ratfps-eq-iff*)  
**qed**

**instantiation** *ratfps* :: (*field-gcd*) *semidom-modulo*  
**begin**

**lift-definition** *modulo-ratfps* :: 'a *ratfps*  $\Rightarrow$  'a *ratfps*  $\Rightarrow$  'a *ratfps* **is**  
 $\lambda f g$ . if  $g = 0$  then  $f$  else  
let  $n = \text{ratfps-subdegree } g$ ;  $h = \text{ratfps-shift } n \ g$   
in *ratfps-of-poly* (*ratfps-cutoff*  $n$  ( $f * \text{inverse } h$ )) \*  $h$  .

**lemma** *fps-of-ratfps-mod* [*simp*]:  
*fps-of-ratfps* ( $f \text{ mod } g$  :: 'a *ratfps*) = *fps-of-ratfps*  $f \text{ mod } \text{fps-of-ratfps } g$   
**unfolding** *fps-mod-def* **by** *transfer'* (*simp add: Let-def ratfps-subdegree-altdef*)

**instance**  
**by** *standard* (*auto intro: ratfps-eqI*)

**end**

**instantiation** *ratfps* :: (*field-gcd*) *euclidean-ring*  
**begin**

**definition** *euclidean-size-ratfps* :: 'a *ratfps*  $\Rightarrow$  *nat* **where**  
*euclidean-size-ratfps*  $x = (\text{if } x = 0 \text{ then } 0 \text{ else } 2 \wedge \text{ratfps-subdegree } x)$

**lemma** *fps-of-ratfps-euclidean-size* [*simp*]:  
*euclidean-size*  $x = \text{euclidean-size } (\text{fps-of-ratfps } x)$   
**unfolding** *euclidean-size-ratfps-def* *fps-euclidean-size-def*  
**by** (*simp add: ratfps-subdegree-altdef*)

**instance proof**  
**show** *euclidean-size* ( $0$  :: 'a *ratfps*) =  $0$  **by** *simp*  
**show** *euclidean-size* ( $a \text{ mod } b$ ) < *euclidean-size*  $b$   
*euclidean-size*  $a \leq \text{euclidean-size } (a * b)$  **if**  $b \neq 0$  **for**  $a \ b$  :: 'a *ratfps*  
**using that** **by** (*simp-all add: mod-size-less size-mult-mono*)

**qed**

**end**

**instantiation** *ratfps* :: (*field-gcd*) *euclidean-ring-cancel*  
**begin**

**instance**  
**by** *standard* (*auto intro: ratfps-eqI*)

**end**

**lemma** *quot-of-ratfps-eq-iff* [*simp*]: *quot-of-ratfps*  $x = \text{quot-of-ratfps } y \longleftrightarrow x = y$

by *transfer simp*

**lemma** *ratfps-eq-0-code*:  $x = 0 \iff \text{fst} (\text{quot-of-ratfps } x) = 0$

**proof**

assume  $\text{fst} (\text{quot-of-ratfps } x) = 0$

moreover have *coprime* ( $\text{fst} (\text{quot-of-ratfps } x)$ ) ( $\text{snd} (\text{quot-of-ratfps } x)$ )

by *transfer (simp add: coprime-quot-of-fract)*

moreover have *normalize* ( $\text{snd} (\text{quot-of-ratfps } x)$ ) =  $\text{snd} (\text{quot-of-ratfps } x)$

by (*simp add: div-unit-factor [symmetric] del: div-unit-factor*)

ultimately have  $\text{quot-of-ratfps } x = (0, 1)$

by (*simp add: prod-eq-iff normalize-idem-imp-is-unit-iff*)

also have  $\dots = \text{quot-of-ratfps } 0$  by *simp*

finally show  $x = 0$  by (*subst (asm) quot-of-ratfps-eq-iff*)

**qed** *simp-all*

**lemma** *fps-dvd-code* [*code-unfold*]:

$x \text{ dvd } y \iff y = 0 \vee ((x::'a)::\text{field-gcd } \text{fps}) \neq 0 \wedge \text{subdegree } x \leq \text{subdegree } y$

using *fps-dvd-iff*[*of x y*] by (*cases x = 0*) *auto*

**lemma** *ratfps-dvd-code* [*code-unfold*]:

$x \text{ dvd } y \iff y = 0 \vee (x \neq 0 \wedge \text{ratfps-subdegree } x \leq \text{ratfps-subdegree } y)$

using *fps-dvd-code* [*of fps-of-ratfps x fps-of-ratfps y*]

by (*simp add: ratfps-subdegree-altdef*)

**instance** *ratfps* :: (*field-gcd*) *normalization-euclidean-semiring* ..

**instantiation** *ratfps* :: (*field-gcd*) *euclidean-ring-gcd*

**begin**

**definition** *gcd-ratfps* = (*Euclidean-Algorithm.gcd* :: 'a *ratfps*  $\Rightarrow$  -)

**definition** *lcm-ratfps* = (*Euclidean-Algorithm.lcm* :: 'a *ratfps*  $\Rightarrow$  -)

**definition** *Gcd-ratfps* = (*Euclidean-Algorithm.Gcd* :: 'a *ratfps set*  $\Rightarrow$  -)

**definition** *Lcm-ratfps* = (*Euclidean-Algorithm.Lcm*:: 'a *ratfps set*  $\Rightarrow$  -)

**instance** by *standard* (*simp-all add: gcd-ratfps-def lcm-ratfps-def Gcd-ratfps-def Lcm-ratfps-def*)

**end**

**lemma** *ratfps-eq-0-iff*:  $x = 0 \iff \text{fps-of-ratfps } x = 0$

using *fps-of-ratfps-eq-iff*[*of x 0*] **unfolding** *fps-of-ratfps-0* by *simp*

**lemma** *ratfps-of-poly-eq-0-iff*:  $\text{ratfps-of-poly } x = 0 \iff x = 0$

by (*auto simp: ratfps-eq-0-iff*)

**lemma** *ratfps-gcd*:

assumes [*simp*]:  $f \neq 0$   $g \neq 0$

shows  $\text{gcd } f g = \text{ratfps-of-poly} (\text{monom } 1 (\text{min} (\text{ratfps-subdegree } f) (\text{ratfps-subdegree } g)))$

$g)))$   
**by** (*rule sym*, *rule gcdI*)  
*(auto simp: ratfps-subdegree-altdef ratfps-dvd-code subdegree-fps-of-poly ratfps-of-poly-eq-0-iff normalize-ratfps-def)*

**lemma** *ratfps-gcd-altdef*:  $\text{gcd } (f :: 'a :: \text{field-gcd ratfps}) \ g =$   
*(if  $f = 0 \wedge g = 0$  then  $0$  else*  
*if  $f = 0$  then  $\text{ratfps-of-poly } (\text{monom } 1 \ (\text{ratfps-subdegree } g))$  else*  
*if  $g = 0$  then  $\text{ratfps-of-poly } (\text{monom } 1 \ (\text{ratfps-subdegree } f))$  else*  
 *$\text{ratfps-of-poly } (\text{monom } 1 \ (\text{min } (\text{ratfps-subdegree } f) \ (\text{ratfps-subdegree } g)))$ )*  
**by** (*simp add: ratfps-gcd normalize-ratfps-def*)

**lemma** *ratfps-lcm*:  
**assumes** [*simp*]:  $f \neq 0 \ g \neq 0$   
**shows**  $\text{lcm } f \ g = \text{ratfps-of-poly } (\text{monom } 1 \ (\text{max } (\text{ratfps-subdegree } f) \ (\text{ratfps-subdegree } g)))$   
 $g)))$   
**by** (*rule sym*, *rule lcmI*)  
*(auto simp: ratfps-subdegree-altdef ratfps-dvd-code subdegree-fps-of-poly ratfps-of-poly-eq-0-iff normalize-ratfps-def)*

**lemma** *ratfps-lcm-altdef*:  $\text{lcm } (f :: 'a :: \text{field-gcd ratfps}) \ g =$   
*(if  $f = 0 \vee g = 0$  then  $0$  else*  
 *$\text{ratfps-of-poly } (\text{monom } 1 \ (\text{max } (\text{ratfps-subdegree } f) \ (\text{ratfps-subdegree } g)))$ )*  
**by** (*simp add: ratfps-lcm*)

**lemma** *ratfps-Gcd*:  
**assumes**  $A - \{0\} \neq \{\}$   
**shows**  $\text{Gcd } A = \text{ratfps-of-poly } (\text{monom } 1 \ (\text{INF } f \in A - \{0\}. \text{ratfps-subdegree } f))$   
**proof** (*rule sym*, *rule GcdI*)  
**fix**  $f$  **assume**  $f \in A$   
**thus**  $\text{ratfps-of-poly } (\text{monom } 1 \ (\text{INF } f \in A - \{0\}. \text{ratfps-subdegree } f)) \ \text{dvd } f$   
**by** (*cases  $f = 0$* ) (*auto simp: ratfps-dvd-code ratfps-of-poly-eq-0-iff ratfps-subdegree-altdef subdegree-fps-of-poly intro!: cINF-lower*)

**next**  
**fix**  $d$  **assume**  $d: \bigwedge f. f \in A \implies d \ \text{dvd } f$   
**from** *assms* **obtain**  $f$  **where**  $f \in A - \{0\}$  **by** *auto*  
**with**  $d[\text{of } f]$  **have** [*simp*]:  $d \neq 0$  **by** *auto*  
**from**  $d$  *assms* **have**  $\text{ratfps-subdegree } d \leq (\text{INF } f \in A - \{0\}. \text{ratfps-subdegree } f)$   
**by** (*intro cINF-greatest*) (*auto simp: ratfps-dvd-code*)  
**with**  $d$  *assms* **show**  $d \ \text{dvd } \text{ratfps-of-poly } (\text{monom } 1 \ (\text{INF } f \in A - \{0\}. \text{ratfps-subdegree } f))$   
 $f))$   
**by** (*simp add: ratfps-dvd-code ratfps-subdegree-altdef subdegree-fps-of-poly*)  
**qed** (*simp-all add: ratfps-subdegree-altdef subdegree-fps-of-poly normalize-ratfps-def*)

**lemma** *ratfps-Gcd-altdef*:  $\text{Gcd } (A :: 'a :: \text{field-gcd ratfps set}) =$   
*(if  $A \subseteq \{0\}$  then  $0$  else  $\text{ratfps-of-poly } (\text{monom } 1 \ (\text{INF } f \in A - \{0\}. \text{ratfps-subdegree } f))$ )*  
**using** *ratfps-Gcd* **by** *auto*

**lemma** *ratfps-Lcm*:  
**assumes**  $A \neq \{\}$   $0 \notin A$  *bdd-above* (*ratfps-subdegree* 'A)  
**shows**  $Lcm\ A = ratfps\text{-of-poly}\ (monom\ 1\ (SUP\ f \in A.\ ratfps\text{-subdegree}\ f))$   
**proof** (*rule sym*, *rule LcmI*)  
**fix**  $f$  **assume**  $f \in A$   
**moreover from** *assms*(3) **have** *bdd-above* (*ratfps-subdegree* 'A) **by** *auto*  
**ultimately show**  $f\ dvd\ ratfps\text{-of-poly}\ (monom\ 1\ (SUP\ f \in A.\ ratfps\text{-subdegree}\ f))$   
**using** *assms*(2)  
**by** (*cases*  $f = 0$ ) (*auto simp: ratfps-dvd-code ratfps-of-poly-eq-0-iff subde-*  
*gree-fps-of-poly*  
*ratfps-subdegree-altdef [abs-def] intro!: cSUP-upper*)  
**next**  
**fix**  $d$  **assume**  $d: \bigwedge f. f \in A \implies f\ dvd\ d$   
**from** *assms* **obtain**  $f$  **where**  $f: f \in A\ f \neq 0$  **by** *auto*  
**show**  $ratfps\text{-of-poly}\ (monom\ 1\ (SUP\ f \in A.\ ratfps\text{-subdegree}\ f))\ dvd\ d$   
**proof** (*cases*  $d = 0$ )  
**assume**  $d \neq 0$   
**moreover from**  $d$  **have**  $\bigwedge f. f \in A \implies f \neq 0 \implies f\ dvd\ d$  **by** *blast*  
**ultimately have**  $ratfps\text{-subdegree}\ d \geq (SUP\ f \in A.\ ratfps\text{-subdegree}\ f)$  **using**  
*assms*  
**by** (*intro cSUP-least*) (*auto simp: ratfps-dvd-code*)  
**with**  $\langle d \neq 0 \rangle$  **show** *?thesis* **by** (*simp add: ratfps-dvd-code ratfps-of-poly-eq-0-iff*  
*ratfps-subdegree-altdef subdegree-fps-of-poly*)  
**qed** *simp-all*  
**qed** (*simp-all add: ratfps-subdegree-altdef subdegree-fps-of-poly normalize-ratfps-def*)

**lemma** *ratfps-Lcm-altdef*:  
 $Lcm\ (A :: 'a :: field\text{-gcd}\ ratfps\ set) =$   
*(if*  $0 \in A \vee \neg bdd\text{-above}\ (ratfps\text{-subdegree}\ 'A)$  *then*  $0$  *else*  
*if*  $A = \{\}$  *then*  $1$  *else*  $ratfps\text{-of-poly}\ (monom\ 1\ (SUP\ f \in A.\ ratfps\text{-subdegree}\ f))$   
**proof** (*cases bdd-above* (*ratfps-subdegree* 'A))  
**assume** *unbounded:  $\neg bdd\text{-above}\ (ratfps\text{-subdegree}\ 'A)$*   
**have**  $Lcm\ A = 0$   
**proof** (*rule ccontr*)  
**assume**  $Lcm\ A \neq 0$   
**from** *unbounded* **obtain**  $f$  **where**  $f: f \in A\ ratfps\text{-subdegree}\ (Lcm\ A) <$   
 $ratfps\text{-subdegree}\ f$   
**unfolding** *bdd-above-def* **by** (*auto simp: not-le*)  
**moreover from** *this* **and**  $\langle Lcm\ A \neq 0 \rangle$  **have**  $ratfps\text{-subdegree}\ f \leq ratfps\text{-subdegree}$   
 $(Lcm\ A)$   
**using** *dvd-Lcm[of f A]* **by** (*auto simp: ratfps-dvd-code*)  
**ultimately show** *False* **by** *simp*  
**qed**  
**with** *unbounded* **show** *?thesis* **by** *simp*  
**qed** (*simp-all add: ratfps-Lcm Lcm-eq-0-I*)

**lemma** *fps-of-ratfps-quot-to-ratfps*:  
 $coeff\ y\ 0 \neq 0 \implies fps\text{-of-ratfps}\ (quot\text{-to-ratfps}\ (x,y)) = fps\text{-of-poly}\ x / fps\text{-of-poly}$

$y$   
**proof** (*transfer, goal-cases*)  
**case** ( $1\ y\ x$ )  
**define**  $x'\ y'$  **where**  $x' = \text{fst}(\text{normalize-quot}(x,y))$  **and**  $y' = \text{snd}(\text{normalize-quot}(x,y))$   
**from**  $1$  **have**  $\text{nz}: y \neq 0$  **by** *auto*  
**have**  $\text{eq}: \text{normalize-quot}(x', y') = (x', y')$  **by** (*simp add: x'-def y'-def*)  
**from**  $\text{normalize-quotE}[OF\ \text{nz},\ \text{of}\ x]$  **obtain**  $d$  **where**  
 $x = \text{fst}(\text{normalize-quot}(x, y)) * d$   
 $y = \text{snd}(\text{normalize-quot}(x, y)) * d$   
 $d\ \text{dvd}\ x$   
 $d\ \text{dvd}\ y$   
 $d \neq 0$  .  
**note**  $d[\text{folded}\ x'\text{-def}\ y'\text{-def}] = \text{this}$   
**have** (*case quot-of-fract (if coeff y' 0 = 0 then 0 else quot-to-fract (x', y')) of*  
 $(a, b) \Rightarrow \text{fps-of-poly}\ a / \text{fps-of-poly}\ b = \text{fps-of-poly}\ x / \text{fps-of-poly}\ y$   
**using**  $d\ \text{eq}\ 1$  **by** (*auto simp: case-prod-unfold fps-of-poly-simps quot-of-fract-quot-to-fract*)  

$$\text{Let-def coeff-0-mult}$$
  
**thus**  $?case$  **by** (*auto simp add: Let-def case-prod-unfold x'-def y'-def*)  
**qed**

**lemma** *fps-of-ratfps-quot-to-ratfps-code-post1*:  
 $\text{fps-of-ratfps}(\text{quot-to-ratfps}(x, \text{pCons}\ 1\ y)) = \text{fps-of-poly}\ x / \text{fps-of-poly}(\text{pCons}\ 1\ y)$   
 $\text{fps-of-ratfps}(\text{quot-to-ratfps}(x, \text{pCons}\ (-1)\ y)) = \text{fps-of-poly}\ x / \text{fps-of-poly}(\text{pCons}\ (-1)\ y)$   
**by** (*simp-all add: fps-of-ratfps-quot-to-ratfps*)

**lemma** *fps-of-ratfps-quot-to-ratfps-code-post2*:  
 $\text{fps-of-ratfps}(\text{quot-to-ratfps}(x'::'a::\{\text{field-char-0}, \text{field-gcd}\}\ \text{poly}, \text{pCons}(\text{numeral}\ n)\ y')) =$   
 $\text{fps-of-poly}\ x' / \text{fps-of-poly}(\text{pCons}(\text{numeral}\ n)\ y')$   
 $\text{fps-of-ratfps}(\text{quot-to-ratfps}(x'::'a::\{\text{field-char-0}, \text{field-gcd}\}\ \text{poly}, \text{pCons}(-\text{numeral}\ n)\ y')) =$   
 $\text{fps-of-poly}\ x' / \text{fps-of-poly}(\text{pCons}(-\text{numeral}\ n)\ y')$   
**by** (*simp-all add: fps-of-ratfps-quot-to-ratfps*)

**lemmas** *fps-of-ratfps-quot-to-ratfps-code-post [code-post] =*  
 $\text{fps-of-ratfps-quot-to-ratfps-code-post1}$   
 $\text{fps-of-ratfps-quot-to-ratfps-code-post2}$

**lemma** *fps-dehorner*:  
**fixes**  $a\ b\ c :: 'a :: \text{semiring-1}\ \text{fps}$  **and**  $d\ e\ f :: 'b :: \text{ring-1}\ \text{fps}$   
**shows**  
 $(b + c) * \text{fps-X} = b * \text{fps-X} + c * \text{fps-X}$   $(a * \text{fps-X}) * \text{fps-X} = a * \text{fps-X}^{\wedge} 2$   
 $a * \text{fps-X}^{\wedge} m * \text{fps-X} = a * \text{fps-X}^{\wedge} (\text{Suc}\ m)$   $a * \text{fps-X} * \text{fps-X}^{\wedge} m = a * \text{fps-X}^{\wedge} (\text{Suc}\ m)$   
 $a * \text{fps-X}^{\wedge} m * \text{fps-X}^{\wedge} n = a * \text{fps-X}^{\wedge} (m+n)$   $a + (b + c) = a + b + c$   $a * 1 =$

$a \cdot 1 * a = a$   
 $d + - e = d - e \quad (-d) * e = - (d * e) \quad d + (e - f) = d + e - f$   
 $(d - e) * fps-X = d * fps-X - e * fps-X \quad fps-X * fps-X = fps-X^2 \quad fps-X * fps-X^m = fps-X^{Suc m}$   
 $fps-X^m * fps-X^n = fps-X^{m + n}$   
**by** (*simp-all add: algebra-simps power2-eq-square power-add power-commutes*)

**lemma** *fps-divide-1*:  $(a :: 'a :: field\ fps) / 1 = a$  **by** *simp*

**lemmas** *fps-of-poly-code-post* [*code-post*] =  
*fps-of-poly-simps fps-const-0-eq-0 fps-const-1-eq-1 numeral-fps-const* [*symmetric*]  
*fps-const-neg* [*symmetric*] *fps-const-divide* [*symmetric*]  
*fps-dehorner Suc-numeral arith-simps fps-divide-1*

**context**

**includes** *term-syntax*

**begin**

**definition**

*valterm-ratfps* ::  
 $'a :: \{field-gcd, typerep\} poly \times (unit \Rightarrow Code-Evaluation.term) \Rightarrow$   
 $'a poly \times (unit \Rightarrow Code-Evaluation.term) \Rightarrow 'a ratfps \times (unit \Rightarrow Code-Evaluation.term)$

**where**

[*code-unfold*]: *valterm-ratfps*  $k\ l =$   
 $Code-Evaluation.valtermify (/) \{.\}$   
 $(Code-Evaluation.valtermify ratfps-of-poly \{.\} k) \{.\}$   
 $(Code-Evaluation.valtermify ratfps-of-poly \{.\} l)$

**end**

**instantiation** *ratfps* ::  $(\{field-gcd, random\}) random$

**begin**

**context**

**includes** *state-combinator-syntax term-syntax*

**begin**

**definition**

$Quickcheck-Random.random\ i =$   
 $Quickcheck-Random.random\ i \circ \rightarrow (\lambda num :: 'a\ poly \times (unit \Rightarrow term).$   
 $Quickcheck-Random.random\ i \circ \rightarrow (\lambda denom :: 'a\ poly \times (unit \Rightarrow term).$   
 $Pair\ (let\ denom = (if\ fst\ denom = 0\ then\ Code-Evaluation.valtermify\ 1\ else$   
 $denom)$   
 $in\ valterm-ratfps\ num\ denom)))$

**instance** ..

**end**

**end**

**instantiation** *ratfps* :: (*field*, *factorial-ring-gcd*, *exhaustive*) *exhaustive*  
**begin**

**definition**

*exhaustive-ratfps* *f* *d* =  
  *Quickcheck-Exhaustive.exhaustive* ( $\lambda$ *num*.  
    *Quickcheck-Exhaustive.exhaustive* ( $\lambda$ *denom*. *f* (  
      *let* *denom* = *if* *denom* = 0 *then* 1 *else* *denom*  
      in ratfps-of-poly num / ratfps-of-poly denom))) *d*) *d*

**instance** ..

**end**

**instantiation** *ratfps* :: (*field-gcd*, *full-exhaustive*) *full-exhaustive*  
**begin**

**definition**

*full-exhaustive-ratfps* *f* *d* =  
  *Quickcheck-Exhaustive.full-exhaustive* ( $\lambda$ *num*::'a poly  $\times$  (*unit*  $\Rightarrow$  *term*).  
    *Quickcheck-Exhaustive.full-exhaustive* ( $\lambda$ *denom*::'a poly  $\times$  (*unit*  $\Rightarrow$  *term*).  
      *f* (*let* *denom* = *if* *fst* *denom* = 0 *then* Code-Evaluation.valtermify 1 *else*  
      denom  
      in valterm-ratfps num denom))) *d*) *d*

**instance** ..

**end**

**quickcheck-generator** *fps* *constructors*: *fps-of-ratfps*

**end**

## 2 Falling factorial as a polynomial

**theory** *Pochhammer-Polynomials*

**imports**

*Complex-Main*  
  *HOL-Combinatorics.Stirling*  
  *HOL-Computational-Algebra.Polynomial*

**begin**

**definition** *pochhammer-poly* :: *nat*  $\Rightarrow$  'a :: {*comm-semiring-1*} *poly* **where**  
  *pochhammer-poly* *n* = *Poly* [*of-nat* (*stirling* *n* *k*). *k*  $\leftarrow$  [0..*Suc* *n*]]

**lemma** *pochhammer-poly-code* [*code abstract*]:

*coeffs* (*pochhammer-poly* *n*) = *map of-nat* (*stirling-row* *n*)

**by** (*simp add: pochhammer-poly-def stirling-row-def Let-def*)

**lemma** *coeff-pochhammer-poly*: *coeff (pochhammer-poly n) k = of-nat (stirling n k)*  
**by** (*simp add: pochhammer-poly-def nth-default-def del: upt-Suc*)

**lemma** *degree-pochhammer-poly* [*simp*]: *degree (pochhammer-poly n) = n*  
**by** (*simp add: degree-eq-length-coeffs pochhammer-poly-def*)

**lemma** *pochhammer-poly-0* [*simp*]: *pochhammer-poly 0 = 1*  
**by** (*simp add: pochhammer-poly-def*)

**lemma** *pochhammer-poly-Suc*: *pochhammer-poly (Suc n) = [:of-nat n,1:] \* pochhammer-poly n*  
**by** (*cases n = 0*) (*simp-all add: poly-eq-iff coeff-pochhammer-poly coeff-pCons split: nat.split*)

**lemma** *pochhammer-poly-altdef*: *pochhammer-poly n = (∏ i < n. [:of-nat i,1:])*  
**by** (*induction n*) (*simp-all add: pochhammer-poly-Suc*)

**lemma** *eval-pochhammer-poly*: *poly (pochhammer-poly n) k = pochhammer k n*  
**by** (*cases n*) (*auto simp add: pochhammer-poly-altdef poly-prod add-ac lessThan-Suc-atMost pochhammer-Suc-prod atLeast0AtMost*)

**lemma** *pochhammer-poly-Suc'*:  
*pochhammer-poly (Suc n) = pCons 0 (pcompose (pochhammer-poly n) [:1,1:])*  
**by** (*simp add: pochhammer-poly-altdef prod.lessThan-Suc-shift pcompose-prod pcompose-pCons add-ac del: prod.lessThan-Suc*)

**end**

### 3 Miscellaneous material required for linear recurrences

**theory** *Linear-Recurrences-Misc*  
**imports**  
*Complex-Main*  
*HOL-Computational-Algebra.Computational-Algebra*  
*HOL-Computational-Algebra.Polynomial-Factorial*  
**begin**

**fun** *zip-with* **where**  
*zip-with f (x#xs) (y#ys) = f x y # zip-with f xs ys*  
*| zip-with f - - = []*

**lemma** *length-zip-with* [*simp*]: *length (zip-with f xs ys) = min (length xs) (length ys)*



by (induction f xs ys rule: zip-with.induct) simp-all

**lemma** zip-with-altdef: zip-with f xs ys = map (λ(x,y). f x y) (zip xs ys)  
by (induction f xs ys rule: zip-with.induct) simp-all

**lemma** zip-with-nth [simp]:  
 $n < \text{length } xs \implies n < \text{length } ys \implies \text{zip-with } f \text{ } xs \text{ } ys ! n = f (xs!n) (ys!n)$   
by (simp add: zip-with-altdef)

**lemma** take-zip-with: take n (zip-with f xs ys) = zip-with f (take n xs) (take n ys)  
**proof** (induction f xs ys arbitrary: n rule: zip-with.induct)  
case (1 f x xs y ys n)  
thus ?case by (cases n) simp-all  
**qed** simp-all

**lemma** drop-zip-with: drop n (zip-with f xs ys) = zip-with f (drop n xs) (drop n ys)  
**proof** (induction f xs ys arbitrary: n rule: zip-with.induct)  
case (1 f x xs y ys n)  
thus ?case by (cases n) simp-all  
**qed** simp-all

**lemma** map-zip-with: map f (zip-with g xs ys) = zip-with (λx y. f (g x y)) xs ys  
by (induction g xs ys rule: zip-with.induct) simp-all

**lemma** zip-with-map: zip-with f (map g xs) (map h ys) = zip-with (λx y. f (g x) (h y)) xs ys  
by (induction λx y. f (g x) (h y) xs ys rule: zip-with.induct) simp-all

**lemma** zip-with-map-left: zip-with f (map g xs) ys = zip-with (λx y. f (g x) y) xs ys  
using zip-with-map[of f g xs λx. x ys] by simp

**lemma** zip-with-map-right: zip-with f xs (map g ys) = zip-with (λx y. f x (g y)) xs ys  
using zip-with-map[of f λx. x xs g ys] by simp

**lemma** zip-with-swap: zip-with (λx y. f y x) xs ys = zip-with f ys xs  
by (induction f ys xs rule: zip-with.induct) simp-all

**lemma** set-zip-with: set (zip-with f xs ys) = (λ(x,y). f x y) ‘ set (zip xs ys)  
by (induction f xs ys rule: zip-with.induct) simp-all

**lemma** zip-with-Pair: zip-with Pair (xs :: 'a list) (ys :: 'b list) = zip xs ys  
by (induction Pair :: 'a ⇒ 'b ⇒ - xs ys rule: zip-with.induct) simp-all

**lemma** zip-with-altdef':  
 $\text{zip-with } f \text{ } xs \text{ } ys = [f (xs!i) (ys!i). i \leftarrow [0..<\min(\text{length } xs) (\text{length } ys)]]$   
by (induction f xs ys rule: zip-with.induct) (simp-all add: map-upt-Suc del:

*upt-Suc*)

**lemma** *zip-altdef*:  $\text{zip } xs \ ys = [(xs!i, ys!i). i \leftarrow [0..<\min (\text{length } xs) (\text{length } ys)]]$   
**by** (*simp add: zip-with-Pair [symmetric] zip-with-altdef'*)

**lemma** *card-poly-roots-bound*:

**fixes**  $p :: 'a :: \{\text{comm-ring-1, ring-no-zero-divisors}\}$  *poly*  
**assumes**  $p \neq 0$   
**shows**  $\text{card } \{x. \text{poly } p \ x = 0\} \leq \text{degree } p$   
**using** *assms*  
**proof** (*induction degree p arbitrary: p rule: less-induct*)  
  **case** (*less p*)  
  **show** *?case*  
  **proof** (*cases*  $\exists x. \text{poly } p \ x = 0$ )  
    **case** *False*  
    **hence**  $\{x. \text{poly } p \ x = 0\} = \{\}$  **by** *blast*  
    **thus** *?thesis* **by** *simp*  
  **next**  
  **case** *True*  
  **then obtain**  $x$  **where**  $x: \text{poly } p \ x = 0$  **by** *blast*  
  **hence**  $[-x, 1:] \ \text{dvd } p$  **by** (*subst (asm) poly-eq-0-iff-dvd*)  
  **then obtain**  $q$  **where**  $q: p = [-x, 1:] * q$  **by** (*auto simp: dvd-def*)  
  **with**  $\langle p \neq 0 \rangle$  **have** [*simp*]:  $q \neq 0$  **by** *auto*  
  **have**  $\text{deg: degree } p = \text{Suc } (\text{degree } q)$   
    **by** (*subst q, subst degree-mult-eq*) *auto*  
  **have**  $\text{card } \{x. \text{poly } p \ x = 0\} \leq \text{card } (\text{insert } x \ \{x. \text{poly } q \ x = 0\})$   
    **by** (*intro card-mono*) (*auto intro: poly-roots-finite simp: q*)  
  **also have**  $\dots \leq \text{Suc } (\text{card } \{x. \text{poly } q \ x = 0\})$   
    **by** (*rule card-insert-le-m1*) *auto*  
  **also from**  $\text{deg}$  **have**  $\text{card } \{x. \text{poly } q \ x = 0\} \leq \text{degree } q$   
    **using**  $\langle p \neq 0 \rangle$  **and**  $q$  **by** (*intro less*) *auto*  
  **also have**  $\text{Suc } \dots = \text{degree } p$  **by** (*simp add: deg*)  
  **finally show** *?thesis* **by** *- simp-all*  
**qed**  
**qed**

**lemma** *poly-eqI-degree*:

**fixes**  $p \ q :: 'a :: \{\text{comm-ring-1, ring-no-zero-divisors}\}$  *poly*  
**assumes**  $\bigwedge x. x \in A \implies \text{poly } p \ x = \text{poly } q \ x$   
**assumes**  $\text{card } A > \text{degree } p \ \text{card } A > \text{degree } q$   
**shows**  $p = q$   
**proof** (*rule ccontr*)  
  **assume** *neq*:  $p \neq q$   
  **have**  $\text{degree } (p - q) \leq \max (\text{degree } p) (\text{degree } q)$   
    **by** (*rule degree-diff-le-max*)  
  **also from** *assms* **have**  $\dots < \text{card } A$  **by** *linarith*  
  **also have**  $\dots \leq \text{card } \{x. \text{poly } (p - q) \ x = 0\}$

**using** *neq* **and** *assms* **by** (*intro card-mono poly-roots-finite*) *auto*  
**finally have**  $\text{degree } (p - q) < \text{card } \{x. \text{poly } (p - q) x = 0\}$  .  
**moreover have**  $\text{degree } (p - q) \geq \text{card } \{x. \text{poly } (p - q) x = 0\}$   
**using** *neq* **by** (*intro card-poly-roots-bound*) *auto*  
**ultimately show** *False* **by** *linarith*  
**qed**

**lemma** *poly-root-order-induct* [*case-names 0 no-roots root*]:

**fixes**  $p :: 'a :: \text{idom poly}$   
**assumes**  $P\ 0 \wedge p. (\bigwedge x. \text{poly } p\ x \neq 0) \implies P\ p$   
 $\bigwedge p\ x\ n. n > 0 \implies \text{poly } p\ x \neq 0 \implies P\ p \implies P\ ([:-x, 1:] \wedge^n * p)$   
**shows**  $P\ p$   
**proof** (*induction degree p arbitrary: p rule: less-induct*)  
**case** (*less p*)  
**consider**  $p = 0 \mid p \neq 0 \exists x. \text{poly } p\ x = 0 \mid \bigwedge x. \text{poly } p\ x \neq 0$  **by** *blast*  
**thus** *?case*  
**proof** *cases*  
**case** 3  
**with** *assms(2)[of p]* **show** *?thesis* **by** *simp*  
**next**  
**case** 2  
**then obtain**  $x$  **where**  $x: \text{poly } p\ x = 0$  **by** *auto*  
**have**  $[:-x, 1:] \wedge^{\text{order } x} p\ \text{dvd } p$  **by** (*intro order-1*)  
**then obtain**  $q$  **where**  $q: p = [:-x, 1:] \wedge^{\text{order } x} p * q$  **by** (*auto simp: dvd-def*)  
**with** 2 **have** *[simp]: q ≠ 0* **by** *auto*  
**have** *order-pos: order x p > 0*  
**using**  $\langle p \neq 0 \rangle$  **and**  $x$  **by** (*auto simp: order-root*)  
**have**  $\text{order } x\ p = \text{order } x\ p + \text{order } x\ q$   
**by** (*subst q, subst order-mult*) (*auto simp: order-power-n-n*)  
**hence** *[simp]: order x q = 0* **by** *simp*  
**have** *deg: degree p = order x p + degree q*  
**by** (*subst q, subst degree-mult-eq*) (*auto simp: degree-power-eq*)  
**with** *order-pos* **have**  $\text{degree } q < \text{degree } p$  **by** *simp*  
**hence**  $P\ q$  **by** (*rule less*)  
**with** *order-pos* **have**  $P\ ([:-x, 1:] \wedge^{\text{order } x} p * q)$   
**by** (*intro assms(3)*) (*auto simp: order-root*)  
**with**  $q$  **show** *?thesis* **by** *simp*  
**qed** (*simp-all add: assms(1)*)  
**qed**

**lemma** *complex-poly-decompose*:

$\text{smult } (\text{lead-coeff } p) (\prod z. \text{poly } p\ z = 0. [:-z, 1:] \wedge^{\text{order } z} p) = (p :: \text{complex poly})$   
**proof** (*induction p rule: poly-root-order-induct*)  
**case** (*no-roots p*)  
**show** *?case*  
**proof** (*cases degree p = 0*)  
**case** *False*  
**hence**  $\neg \text{constant } (poly\ p)$  **by** (*subst constant-degree*)  
**with** *fundamental-theorem-of-algebra* **and** *no-roots* **show** *?thesis* **by** *blast*

```

qed (auto elim!: degree-eq-zeroE)
next
  case (root p x n)
  from root have *: {z. poly ([: - x, 1:] ^ n * p) z = 0} = insert x {z. poly p z = 0}
  by auto
  have smult (lead-coeff ([: - x, 1:] ^ n * p))
    (∏ z | poly ([: - x, 1:] ^ n * p) z = 0. [: - z, 1:] ^ order z ([: - x, 1:] ^ n *
  p)) =
    [: - x, 1:] ^ order x ([: - x, 1:] ^ n * p) *
    smult (lead-coeff p) (∏ z ∈ {z. poly p z = 0}. [: - z, 1:] ^ order z ([: - x, 1:]
  ^ n * p))
  by (subst *, subst prod.insert)
    (insert root, auto intro: poly-roots-finite simp: mult-ac lead-coeff-mult lead-coeff-power)
  also have order x ([: - x, 1:] ^ n * p) = n
  using root by (subst order-mult) (auto simp: order-power-n-n order-0I)
  also have (∏ z ∈ {z. poly p z = 0}. [: - z, 1:] ^ order z ([: - x, 1:] ^ n * p)) =
    (∏ z ∈ {z. poly p z = 0}. [: - z, 1:] ^ order z p)
  proof (intro prod.cong refl, goal-cases)
    case (1 y)
    with root have order y ([: - x, 1:] ^ n) = 0 by (intro order-0I) auto
    thus ?case using root by (subst order-mult) auto
  qed
  also note root.IH
  finally show ?case .
qed simp-all

```

**lemma** normalize-field:  
 normalize (x :: 'a :: {normalization-semidom,field}) = (if x = 0 then 0 else 1)  
**by** (auto simp: normalize-1-iff dvd-field-iff)

**lemma** unit-factor-field [simp]:  
 unit-factor (x :: 'a :: {normalization-semidom,field}) = x  
**using** unit-factor-mult-normalize[of x] normalize-field[of x]  
**by** (simp split: if-splits)

**lemma** coprime-linear-poly:  
**fixes** c :: 'a :: field-gcd  
**assumes** c ≠ c'  
**shows** coprime [:c,1:] [:c',1:]  
**proof** -  
**have** gcd [:c,1:] [:c',1:] = gcd ([:c,1:] - [:c',1:]) [:c',1:]  
**by** (rule gcd-diff1 [symmetric])  
**also** **have** [:c,1:] - [:c',1:] = [:c-c',1:] **by** simp  
**also** **from** assms **have** gcd ... [:c',1:] = normalize [:c-c',1:]  
**by** (intro gcd-proj1-if-dvd) (auto simp: const-poly-dvd-iff dvd-field-iff)  
**also** **from** assms **have** ... = 1 **by** (simp add: normalize-poly-def)  
**finally** **show** coprime [:c,1:] [:c',1:]

```

    by (simp add: gcd-eq-1-imp-coprime)
qed

lemma coprime-linear-poly':
  fixes c :: 'a :: field-gcd
  assumes c ≠ c' c ≠ 0 c' ≠ 0
  shows coprime [1,c] [1,c']
proof -
  have gcd [1,c] [1,c'] = gcd ([1,c] mod [1,c']) [1,c']
    by simp
  also have eucl-rel-poly [1, c] [1, c'] ([c/c'], [1-c/c'])
    using assms by (auto simp add: eucl-rel-poly-iff one-pCons)
  hence [1,c] mod [1,c'] = [1 - c / c']
    by (rule mod-poly-eq)
  also from assms have gcd ... [1,c'] = normalize ([1 - c / c'])
    by (intro gcd-proj1-if-dvd) (auto simp: const-poly-dvd-iff dvd-field-iff)
  also from assms have ... = 1 by (auto simp: normalize-poly-def)
  finally show ?thesis
    by (rule gcd-eq-1-imp-coprime)
qed

end

```

## 4 Partial Fraction Decomposition

```

theory Partial-Fraction-Decomposition
imports
  Main
  HOL-Computational-Algebra.Computational-Algebra
  HOL-Computational-Algebra.Polynomial-Factorial
  HOL-Library.Sublist
  Linear-Recurrences-Misc
begin

```

### 4.1 Decomposition on general Euclidean rings

Consider elements  $x, y_1, \dots, y_n$  of a ring  $R$ , where the  $y_i$  are pairwise coprime. A *Partial Fraction Decomposition* of these elements (or rather the formal quotient  $x/(y_1 \dots y_n)$  that they represent) is a finite sum of summands of the form  $a/y_i^k$ . Obviously, the sum can be arranged such that there is at most one summand with denominator  $y_i^n$  for any combination of  $i$  and  $n$ ; in particular, there is at most one summand with denominator 1.

We can decompose the summands further by performing division with remainder until in all quotients, the numerator's Euclidean size is less than that of the denominator.

The following function performs the first step of the above process: it takes

the values  $x$  and  $y_1, \dots, y_n$  and returns the numerators of the summands in the decomposition. (the denominators are simply the  $y_i$  from the input)

```
fun decompose :: ('a :: euclidean-ring-gcd)  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  decompose x [] = []
| decompose x [y] = [x]
| decompose x (y#ys) =
  (case bezout-coefficients y (prod-list ys) of
    (a, b)  $\Rightarrow$  (b*x) # decompose (a*x) ys)
```

```
lemma decompose-rec:
  ys  $\neq$  []  $\implies$  decompose x (y#ys) =
    (case bezout-coefficients y (prod-list ys) of
      (a, b)  $\Rightarrow$  (b*x) # decompose (a*x) ys)
by (cases ys) simp-all
```

```
lemma length-decompose [simp]: length (decompose x ys) = length ys
proof (induction x ys rule: decompose.induct)
  case ( $\exists$  x y z ys)
  obtain a b where ab: (a,b) = bezout-coefficients y (prod-list (z#ys))
  by (cases bezout-coefficients y (z * prod-list ys)) simp-all
from  $\exists$ [OF ab] ab[symmetric] show ?case by simp
qed simp-all
```

```
fun decompose' :: ('a :: euclidean-ring-gcd)  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  decompose' x [] - = []
| decompose' x [y] - = [x]
| decompose' - - [] = []
| decompose' x (y#ys) (p#ps) =
  (case bezout-coefficients y p of
    (a, b)  $\Rightarrow$  (b*x) # decompose' (a*x) ys ps)
```

```
primrec decompose-aux :: 'a :: {ab-semigroup-mult, monoid-mult}  $\Rightarrow$  - where
  decompose-aux acc [] = [acc]
| decompose-aux acc (x#xs) = acc # decompose-aux (x * acc) xs
```

```
lemma decompose-code [code]:
  decompose x ys = decompose' x ys (tl (rev (decompose-aux 1 (rev ys))))
proof (induction x ys rule: decompose.induct)
  case ( $\exists$  x y1 y2 ys)
  have [simp]:
    decompose-aux acc xs = map ( $\lambda$ x. prod-list x * acc) (prefixes xs) for acc :: 'a
and xs
  by (induction xs arbitrary: acc) (simp-all add: mult-ac)
  show ?case
  using  $\exists$ [of fst (bezout-coefficients y1 (y2 * prod-list ys))
    snd (bezout-coefficients y1 (y2 * prod-list ys))]
  by (simp add: case-prod-unfold rev-map prefixes-conv-suffixes o-def mult-ac)
qed simp-all
```

The next function performs the second step: Given a quotient of the form  $x/y^n$ , it returns a list of  $x_0, \dots, x_n$  such that  $x/y^n = x_0/y^n + \dots + x_{n-1}/y + x_n$  and all  $x_i$  have a Euclidean size less than that of  $y$ .

**fun** *normalise-decomp* :: ('a :: semiring-modulo) ⇒ 'a ⇒ nat ⇒ 'a × ('a list)  
**where**

```

  normalise-decomp x y 0 = (x, [])
| normalise-decomp x y (Suc n) = (
  case normalise-decomp (x div y) y n of
    (z, rs) ⇒ (z, x mod y # rs)

```

**lemma** *length-normalise-decomp [simp]*:  $\text{length} (\text{snd} (\text{normalise-decomp } x \ y \ n)) = n$

**by** (*induction x y n rule: normalise-decomp.induct*) (*auto split: prod.split*)

The following constant implements the full process of partial fraction decomposition: The input is a quotient  $x/(y_1^{k_1} \dots y_n^{k_n})$  and the output is a sum of an entire element and terms of the form  $a/y_i^k$  where  $a$  has a Euclidean size less than  $y_i$ .

**definition** *partial-fraction-decomposition* ::

```

'a :: euclidean-ring-gcd ⇒ ('a × nat) list ⇒ 'a × 'a list list where
  partial-fraction-decomposition x ys = (if ys = [] then (x, []) else
    (let zs = [let (y, n) = ys ! i
      in normalise-decomp (decompose x (map (λ(y,n). y ^ Suc n) ys) ! i)
      y (Suc n).
      i ← [0..<length ys]]
    in (sum-list (map fst zs), map snd zs)))

```

**lemma** *length-pfd1 [simp]*:

$\text{length} (\text{snd} (\text{partial-fraction-decomposition } x \ ys)) = \text{length } ys$

**by** (*simp add: partial-fraction-decomposition-def*)

**lemma** *length-pfd2 [simp]*:

$i < \text{length } ys \implies \text{length} (\text{snd} (\text{partial-fraction-decomposition } x \ ys) ! i) = \text{snd} (ys ! i) + 1$

**by** (*auto simp: partial-fraction-decomposition-def case-prod-unfold Let-def*)

**lemma** *size-normalise-decomp*:

$a \in \text{set} (\text{snd} (\text{normalise-decomp } x \ y \ n)) \implies y \neq 0 \implies \text{euclidean-size } a < \text{euclidean-size } y$

**by** (*induction x y n rule: normalise-decomp.induct*)

(*auto simp: case-prod-unfold Let-def mod-size-less*)

**lemma** *size-partial-fraction-decomposition*:

$i < \text{length } xs \implies \text{fst } (xs ! i) \neq 0 \implies x \in \text{set} (\text{snd} (\text{partial-fraction-decomposition } y \ xs) ! i)$

$\implies \text{euclidean-size } x < \text{euclidean-size } (\text{fst } (xs ! i))$

**by** (*auto simp: partial-fraction-decomposition-def Let-def case-prod-unfold*)

*simp del: normalise-decomp.simps split: if-split-asm intro!: size-normalise-decomp*)

A homomorphism  $\varphi$  from a Euclidean ring  $R$  into another ring  $S$  with a notion of division. We will show that for any  $x, y \in R$  such that  $\phi(y)$  is a unit, we can perform partial fraction decomposition on the quotient  $\varphi(x)/\varphi(y)$ .

The obvious choice for  $S$  is the fraction field of  $R$ , but other choices may also make sense: If, for example,  $R$  is a ring of polynomials  $K[X]$ , then one could let  $S = K$  and  $\varphi$  the evaluation homomorphism. Or one could let  $S = K[[X]]$  (the ring of formal power series) and  $\varphi$  the canonical homomorphism from polynomials to formal power series.

```

locale pfd-homomorphism =
fixes lift :: ('a :: euclidean-ring-gcd)  $\Rightarrow$  ('b :: euclidean-semiring-cancel)
assumes lift-add: lift (a + b) = lift a + lift b
assumes lift-mult: lift (a * b) = lift a * lift b
assumes lift-0 [simp]: lift 0 = 0
assumes lift-1 [simp]: lift 1 = 1
begin

```

```

lemma lift-power:
  lift (a ^ n) = lift a ^ n
  by (induction n) (simp-all add: lift-mult)

```

```

definition from-decomp :: 'a  $\Rightarrow$  'a  $\Rightarrow$  nat  $\Rightarrow$  'b where
  from-decomp x y n = lift x div lift y ^ n

```

```

lemma decompose:
  assumes ys  $\neq$  [] pairwise coprime (set ys) distinct ys
   $\bigwedge y. y \in \text{set } ys \implies \text{is-unit } (\text{lift } y)$ 
  shows  $(\sum i < \text{length } ys. \text{lift } (\text{decompose } x \text{ ys } ! i) \text{ div lift } (ys ! i)) =$ 
  lift x div lift (prod-list ys)
  using assms

```

```

proof (induction ys arbitrary: x rule: list-nonempty-induct)
  case (cons y ys x)
  from cons.prem1 have coprime (prod-list ys) y
  by (auto simp add: pairwise-insert intro: prod-list-coprime-left)
  from cons.prem2 have unit: is-unit (lift y) by simp
  moreover from cons.prem3 have  $\forall y \in \text{set } ys. \text{is-unit } (\text{lift } y)$  by simp
  hence unit': is-unit (lift (prod-list ys)) by (induction ys) (auto simp: lift-mult)
  ultimately have unit: lift y dvd b lift (prod-list ys) dvd b for b by auto

```

```

obtain s t where st: bezout-coefficients y (prod-list ys) = (s, t)
  by (cases bezout-coefficients y (prod-list ys)) simp-all

```

```

from  $\langle \text{pairwise coprime } (\text{set } (y \# \text{ys})) \rangle$ 
have coprime: pairwise coprime (set ys)
  by (rule pairwise-subset) auto

```

```

have  $(\sum i < \text{length } (y \# \text{ys}). \text{lift } (\text{decompose } x (y \# \text{ys}) ! i) \text{ div lift } ((y \# \text{ys}) ! i)) =$ 

```



$\text{lift } (t * x) \text{ div lift } y + \text{lift } (s * x) \text{ div lift } (\text{prod-list } ys)$   
**using** *cons.hyps cons.premis coprime unfolding length-Cons atLeast0LessThan*  
*[symmetric]*  
**by** (*subst sum.atLeast-Suc-lessThan, simp, subst sum.shift-bounds-Suc-ivl*)  
*(simp add: atLeast0LessThan decompose-rec st cons.IH lift-mult)*  
**also have**  $(\text{lift } (t * x) \text{ div lift } y + \text{lift } (s * x) \text{ div lift } (\text{prod-list } ys)) * \text{lift } (\text{prod-list } (y \# ys)) =$   
 $\text{lift } (\text{prod-list } ys) * (\text{lift } y * (\text{lift } (t * x) \text{ div lift } y)) +$   
 $\text{lift } y * (\text{lift } (\text{prod-list } ys) * (\text{lift } (s * x) \text{ div lift } (\text{prod-list } ys)))$   
**by** (*simp-all add: lift-mult algebra-simps*)  
**also have**  $\dots = \text{lift } (\text{prod-list } ys * t * x + y * s * x)$  **using** *assms unit*  
**by** (*simp add: lift-mult lift-add algebra-simps*)  
**finally have**  $(\sum i < \text{length } (y \# ys). \text{lift } (\text{decompose } x (y \# ys) ! i) \text{ div lift } ((y \# ys) ! i)) =$   
 $\text{lift } ((s * y + t * \text{prod-list } ys) * x) \text{ div lift } (\text{prod-list } (y \# ys))$   
**using** *unit by (subst unit-eq-div2) (auto simp: lift-mult lift-add algebra-simps)*  
**also have**  $s * y + t * \text{prod-list } ys = \text{gcd } (\text{prod-list } ys) y$   
**using** *bezout-coefficients-fst-snd[of y prod-list ys] by (simp add: st gcd.commute)*  
**also have**  $\dots = 1$   
**using** *<coprime (prod-list ys) y> by simp*  
**finally show** *?case by simp*  
**qed** *simp-all*

**lemma** *normalise-decomp:*

**fixes**  $x y :: 'a$  **and**  $n :: \text{nat}$   
**assumes** *is-unit (lift y)*  
**defines**  $xs \equiv \text{snd } (\text{normalise-decomp } x y n)$   
**shows**  $\text{lift } (\text{fst } (\text{normalise-decomp } x y n)) + (\sum i < n. \text{from-decomp } (xs!i) y (n-i)) =$   
 $\text{lift } x \text{ div lift } y \wedge n$   
**using** *assms unfolding xs-def*  
**proof** (*induction x y n rule: normalise-decomp.induct, goal-cases*)  
**case**  $(2 x y n)$   
**from**  $2(2)$  **have** *unit: is-unit (lift y ^ n)*  
**by** (*simp add: is-unit-power-iff*)  
**obtain**  $a b$  **where** *ab: normalise-decomp (x div y) y n = (a, b)*  
**by** (*cases normalise-decomp (x div y) y n simp-all*)  
**have**  $\text{lift } (\text{fst } (\text{normalise-decomp } x y (\text{Suc } n))) +$   
 $(\sum i < \text{Suc } n. \text{from-decomp } (\text{snd } (\text{normalise-decomp } x y (\text{Suc } n)) ! i) y (\text{Suc } n - i)) =$   
 $\text{lift } a + (\sum i < n. \text{from-decomp } (b ! i) y (n - i)) + \text{from-decomp } (x \text{ mod } y) y (\text{Suc } n)$   
**unfolding** *atLeast0LessThan[symmetric]*  
**apply** (*subst sum.atLeast-Suc-lessThan*)  
**apply** *simp*  
**apply** (*subst sum.shift-bounds-Suc-ivl*)  
**apply** (*simp add: ab atLeast0LessThan ac-simps*)  
**done**  
**also have**  $\text{lift } a + (\sum i < n. \text{from-decomp } (b ! i) y (n - i)) =$

```

      lift (x div y) div lift y ^ n
    using 2 by (simp add: ab)
  also from 2(2) unit have (... + from-decomp (x mod y) y (Suc n)) * lift y =
    (lift ((x div y) * y + x mod y) div lift y ^ n) (is ?A * - = ?B div -)
  unfolding lift-add lift-mult
  apply (subst div-add)
  apply (auto simp add: from-decomp-def algebra-simps dvd-div-mult2-eq
    unit-div-mult-swap dvd-div-mult2-eq[OF unit-imp-dvd] is-unit-mult-iff)
  done
  with 2(2) have ?A = ... div lift y by (subst eq-commute, subst dvd-div-eq-mult)
  auto
  also from 2(2) unit have ... = ?B div (lift y ^ Suc n)
    by (subst is-unit-div-mult2-eq [symmetric]) (auto simp: mult-ac)
  also have x div y * y + x mod y = x by (rule div-mult-mod-eq)
  finally show ?case .
qed simp-all

```

**lemma lift-prod-list:**  $\text{lift} (\text{prod-list } xs) = \text{prod-list} (\text{map lift } xs)$   
 by (induction xs) (simp-all add: lift-mult)

**lemma lift-sum:**  $\text{lift} (\text{sum } f A) = \text{sum} (\lambda x. \text{lift} (f x)) A$   
 by (cases finite A, induction A rule: finite-induct) (simp-all add: lift-add)

**lemma partial-fraction-decomposition:**

```

  fixes ys :: ('a × nat) list
  defines ys' ≡ map (λ(x,n). x ^ Suc n) ys :: 'a list
  assumes unit: ∧y. y ∈ fst ' set ys ⇒ is-unit (lift y)
  assumes coprime: pairwise coprime (set ys')
  assumes distinct: distinct ys'
  assumes partial-fraction-decomposition x ys = (a, zs)
  shows lift a + (∑ i < length ys. ∑ j ≤ snd (ys!i).
    from-decomp (zs!i!j) (fst (ys!i)) (snd (ys!i)+1 - j)) =
    lift x div lift (prod-list ys')

```

**proof** (cases ys = [])  
 assume [simp]: ys ≠ []  
 define n where n = length ys

**have**  $\text{lift } x \text{ div lift} (\text{prod-list } ys') = (\sum i < n. \text{lift} (\text{decompose } x \text{ } ys' ! i) \text{ div lift} (ys' ! i))$

**using** *assms* **by** (subst decompose [symmetric])  
 (force simp: lift-prod-list prod-list-zero-iff lift-power lift-mult o-def n-def  
 is-unit-mult-iff is-unit-power-iff)+

**also have** ... =  
 $(\sum i < n. \text{lift} (\text{fst} (\text{normalise-decomp} (\text{decompose } x \text{ } ys' ! i) (\text{fst} (ys!i)) (\text{snd} (ys!i)+1)))) +$   
 $(\sum i < n. (\sum j \leq \text{snd} (ys!i). \text{from-decomp} (zs!i!j) (\text{fst} (ys!i)) (\text{snd} (ys!i)+1 - j)))$   
 (is - = ?A + ?B)

**proof** (subst sum.distrib [symmetric], intro sum.cong refl, goal-cases)  
 case (1 i)

```

from 1 have lift (ys' ! i) = lift (fst (ys ! i)) ^ Suc (snd (ys ! i))
  by (simp add: ys'-def n-def lift-power lift-mult split: prod.split)
also from 1 have lift (decompose x ys' ! i) div ... =
  lift (fst (normalise-decomp (decompose x ys' ! i) (fst (ys!i)) (snd (ys!i)+1)))
+
  (∑ j < Suc (snd (ys ! i)). from-decomp (snd (normalise-decomp (decompose x
ys' ! i)
  (fst (ys!i)) (snd (ys!i)+1) ! j) (fst (ys ! i)) (snd (ys!i)+1 - j)) (is - =
- + ?C)
  by (subst normalise-decomp [symmetric]) (simp-all add: n-def unit)
also have ?C = (∑ j ≤ snd (ys!i). from-decomp (zs!i!j) (fst (ys!i)) (snd (ys!i)+1
- j))
  using assms 1
  by (intro sum.cong refl)
  (auto simp: partial-fraction-decomposition-def case-prod-unfold Let-def o-def
n-def
  simp del: normalise-decomp.simps)
  finally show ?case .
qed
also from assms have ?A = lift a
  by (auto simp: partial-fraction-decomposition-def o-def sum-list-sum-nth atLeast0LessThan
case-prod-unfold Let-def lift-sum n-def intro!: sum.cong)
  finally show ?thesis by (simp add: n-def)
qed (insert assms, simp add: partial-fraction-decomposition-def)

end

```

## 4.2 Specific results for polynomials

**definition** *divmod-field-poly* :: 'a :: field poly ⇒ 'a poly ⇒ 'a poly × 'a poly **where**  
*divmod-field-poly* p q = (p div q, p mod q)

**lemma** *divmod-field-poly-code* [code]:

```

divmod-field-poly p q =
  (let cg = coeffs q
   in if cg = [] then (0, p)
   else let cf = coeffs p; ilc = inverse (last cg);
         ch = map ((* ) ilc) cg;
         (q, r) =
           divmod-poly-one-main-list [] (rev cf) (rev ch)
           (1 + length cf - length cg)
         in (poly-of-list (map ((* ) ilc) q), poly-of-list (rev r)))
unfolding divmod-field-poly-def by (rule pdivmod-via-divmod-list)

```

**definition** *normalise-decomp-poly* :: 'a::field-gcd poly ⇒ 'a poly ⇒ nat ⇒ 'a poly  
× 'a poly list

**where** [simp]: *normalise-decomp-poly* (p :: - poly) q n = *normalise-decomp* p q n

**lemma** *normalise-decomp-poly-code* [code]:

*normalise-decomp-poly*  $x\ y\ 0 = (x, [])$   
*normalise-decomp-poly*  $x\ y\ (Suc\ n) =$   
   *let*  $(x', r) = \text{divmod-field-poly } x\ y;$   
    $(z, rs) = \text{normalise-decomp-poly } x'\ y\ n$   
   *in*  $(z, r \# rs)$   
**by** (*simp-all add: divmod-field-poly-def*)

**definition** *poly-pfd-simple* **where**

*poly-pfd-simple*  $x\ cs =$  (*if*  $cs = []$  *then*  $(x, [])$  *else*  
   (*let*  $zs = [let\ (c, n) = cs\ !\ i$   
     *in* *normalise-decomp-poly* (*decompose*  $x$   
       (*map*  $(\lambda(c,n). [:1,-c:] \wedge Suc\ n)\ cs) !\ i$   $[:1,-c:] (n+1)$ .  
        $i \leftarrow [0..<length\ cs]$ )  
     *in* (*sum-list* (*map* *fst*  $zs$ ), *map* (*map*  $(\lambda p. coeff\ p\ 0) \circ snd$ )  $zs$ )))

**lemma** *poly-pfd-simple-code* [*code*]:

*poly-pfd-simple*  $x\ cs =$   
   (*if*  $cs = []$  *then*  $(x, [])$  *else*  
     *let*  $zs = \text{zip-with } (\lambda(c,n)\ \text{decomp. normalise-decomp-poly decomp } [:1,-c:]$   
        $(n+1))$   
        $cs\ (\text{decompose } x\ (\text{map } (\lambda(c,n). [:1,-c:] \wedge Suc\ n)\ cs))$   
     *in* (*sum-list* (*map* *fst*  $zs$ ), *map* (*map*  $(\lambda p. coeff\ p\ 0) \circ snd$ )  $zs$ ))  
**unfolding** *poly-pfd-simple-def zip-with-altdef'*  
**by** (*simp add: Let-def case-prod-unfold*)

**lemma** *fst-poly-pfd-simple*:

*fst* (*poly-pfd-simple*  $x\ cs$ ) =  
   *fst* (*partial-fraction-decomposition*  $x\ (\text{map } (\lambda(c,n). ([:1,-c:],n))\ cs)$ )  
**by** (*auto simp: poly-pfd-simple-def partial-fraction-decomposition-def o-def*  
   *case-prod-unfold Let-def sum-list-sum-nth intro!: sum.cong*)

**lemma** *const-polyI*: *degree*  $p = 0 \implies [:coeff\ p\ 0:] = p$

**by** (*elim degree-eq-zeroE*) *simp-all*

**lemma** *snd-poly-pfd-simple*:

*map* (*map*  $(\lambda c. [:c :: 'a :: field-gcd:])$ ) (*snd* (*poly-pfd-simple*  $x\ cs$ )) =  
   (*snd* (*partial-fraction-decomposition*  $x\ (\text{map } (\lambda(c,n). ([:1,-c:],n))\ cs)$ ))

**proof** –

**have** *snd* (*poly-pfd-simple*  $x\ cs$ ) = *map* (*map*  $(\lambda p. coeff\ p\ 0)$ )  
   (*snd* (*partial-fraction-decomposition*  $x\ (\text{map } (\lambda(c,n). ([:1,-c:],n))\ cs)$ ))  
**(is - = map ?f ?B)**

**by** (*auto simp: poly-pfd-simple-def partial-fraction-decomposition-def o-def*  
   *case-prod-unfold Let-def sum-list-sum-nth intro!: sum.cong*)

**also have** *map* (*map*  $(\lambda c. [:c:])$ ) (*map* ?f ?B) = *map* (*map*  $(\lambda x. x)$ ) ?B

**unfolding** *map-map o-def*

**proof** (*intro map-cong refl const-polyI, goal-cases*)

**case**  $(1\ ys\ y)$

**from 1 obtain**  $i$  **where**  $i < length\ cs$

$ys = snd\ (\text{partial-fraction-decomposition } x\ (\text{map } (\lambda(c,n). ([:1,-c:],n))\ cs)) !\ i$

```

    by (auto simp: in-set-conv-nth)
  with 1 have euclidean-size  $y < \text{euclidean-size (fst (map (\lambda(c,n). ([:1,-c:],n)) cs ! i))}$ 
    by (intro size-partial-fraction-decomposition[of i - x])
      (auto simp: case-prod-unfold Let-def)
  with  $i(1)$  have euclidean-size  $y < 2$ 
  by (auto simp: case-prod-unfold Let-def euclidean-size-poly-def split: if-split-asm)
  thus ?case
    by (cases  $y$  rule: pCons-cases) (auto simp: euclidean-size-poly-def split: if-split-asm)
  qed
  finally show ?thesis by simp
qed

```

**lemma** *poly-pfd-simple*:

```

  partial-fraction-decomposition  $x$  (map (\lambda(c,n). ([:1,-c:],n)) cs) =
    (fst (poly-pfd-simple  $x$  cs), map (map (\lambda c. [:c:])) (snd (poly-pfd-simple  $x$  cs)))
  by (simp add: fst-poly-pfd-simple snd-poly-pfd-simple)

```

end

## 5 Factorizations of polynomials

**theory** *Factorizations*

**imports**

*Complex-Main*

*Linear-Recurrences-Misc*

*HOL-Computational-Algebra.Computational-Algebra*

*HOL-Computational-Algebra.Polynomial-Factorial*

**begin**

We view a factorisation of a polynomial as a pair consisting of the leading coefficient and a list of roots with multiplicities. This gives us a factorization into factors of the form  $(X - c)^{n+1}$ .

**definition** *interp-factorization where*

```

  interp-factorization = (\lambda(a,cs). Polynomial.smult a (\prod (c,n)←cs. [:-c,1:] ^ Suc n))

```

An alternative way to factorise is as a pair of the leading coefficient and factors of the form  $(1 - cX)^{n+1}$ .

**definition** *interp-alt-factorization where*

```

  interp-alt-factorization = (\lambda(a,cs). Polynomial.smult a (\prod (c,n)←cs. [:1,-c:] ^ Suc n))

```

**definition** *is-factorization-of where*

```

  is-factorization-of fctrs  $p$  =
    (interp-factorization fctrs =  $p \wedge \text{distinct (map fst (snd fctrs))}$ )

```

**definition** *is-alt-factorization-of* **where**

*is-alt-factorization-of* *fctrs* *p* =  
 (*interp-alt-factorization* *fctrs* = *p*  $\wedge$  *0*  $\notin$  *set* (*map* *fst* (*snd* *fctrs*)))  $\wedge$   
*distinct* (*map* *fst* (*snd* *fctrs*)))

Regular and alternative factorisations are related by reflecting the polynomial.

**lemma** *interp-factorization-reflect*:

**assumes** (*0*::'*a*::*idom*)  $\notin$  *fst* ' *set* (*snd* *fctrs*)

**shows** *reflect-poly* (*interp-factorization* *fctrs*) = *interp-alt-factorization* *fctrs*

**proof** –

**have** *reflect-poly* (*interp-factorization* *fctrs*) =

*Polynomial.smult* (*fst* *fctrs*) ( $\prod x \leftarrow \text{snd } fctrs. \text{reflect-poly } [:- \text{fst } x, 1:] \wedge$   
*Suc* (*snd* *x*))

**by** (*simp* *add*: *interp-factorization-def* *interp-alt-factorization-def* *case-prod-unfold*  
*reflect-poly-smult* *reflect-poly-prod-list* *reflect-poly-power* *o-def* *del*:  
*power-Suc*)

**also have** *map* ( $\lambda x. \text{reflect-poly } [:- \text{fst } x, 1:] \wedge \text{Suc } (\text{snd } x)$ ) (*snd* *fctrs*) =  
*map* ( $\lambda x. [:- \text{fst } x, 1:] \wedge \text{Suc } (\text{snd } x)$ ) (*snd* *fctrs*)

**using** *assms* **by** (*intro* *list.map-cong0*, *subst* *reflect-poly-pCons*) *auto*

**also have** *Polynomial.smult* (*fst* *fctrs*) (*prod-list* ...) = *interp-alt-factorization*  
*fctrs*

**by** (*simp* *add*: *interp-alt-factorization-def* *case-prod-unfold*)

**finally show** *?thesis* .

**qed**

**lemma** *interp-alt-factorization-reflect*:

**assumes** (*0*::'*a*::*idom*)  $\notin$  *fst* ' *set* (*snd* *fctrs*)

**shows** *reflect-poly* (*interp-alt-factorization* *fctrs*) = *interp-factorization* *fctrs*

**proof** –

**have** *reflect-poly* (*interp-alt-factorization* *fctrs*) =

*Polynomial.smult* (*fst* *fctrs*) ( $\prod x \leftarrow \text{snd } fctrs. \text{reflect-poly } [:- \text{fst } x, 1:] \wedge$   
*Suc* (*snd* *x*))

**by** (*simp* *add*: *interp-factorization-def* *interp-alt-factorization-def* *case-prod-unfold*  
*reflect-poly-smult* *reflect-poly-prod-list* *reflect-poly-power* *o-def* *del*:  
*power-Suc*)

**also have** *map* ( $\lambda x. \text{reflect-poly } [:- \text{fst } x, 1:] \wedge \text{Suc } (\text{snd } x)$ ) (*snd* *fctrs*) =  
*map* ( $\lambda x. [:- \text{fst } x, 1:] \wedge \text{Suc } (\text{snd } x)$ ) (*snd* *fctrs*)

**proof** (*intro* *list.map-cong0*, *clarsimp* *simp* *del*: *power-Suc*, *goal-cases*)

**fix** *c* *n* **assume** (*c*, *n*)  $\in$  *set* (*snd* *fctrs*)

**with** *assms* **have** *c*  $\neq$  *0* **by** *force*

**thus** *reflect-poly* [:- *c*, 1:]  $\wedge$  *Suc* *n* = [:- *c*, 1:]  $\wedge$  *Suc* *n*

**by** (*simp* *add*: *reflect-poly-pCons* *del*: *power-Suc*)

**qed**

**also have** *Polynomial.smult* (*fst* *fctrs*) (*prod-list* ...) = *interp-factorization* *fctrs*

**by** (*simp* *add*: *interp-factorization-def* *case-prod-unfold*)

**finally show** *?thesis* .

**qed**

**lemma** *coeff-0-interp-factorization*:

*coeff* (*interp-factorization* *fctrs*)  $0 = (0 :: 'a :: idom) \longleftrightarrow$   
 $\text{fst } fctrs = 0 \vee 0 \in \text{fst } ' \text{ set } (\text{snd } fctrs)$

**by** (*force simp: interp-factorization-def case-prod-unfold coeff-0-prod-list o-def*  
*coeff-0-power prod-list-zero-iff simp del: power-Suc*)

**lemma** *reflect-factorization*:

**assumes** *coeff* *p*  $0 \neq (0 :: 'a :: idom)$

**assumes** *is-factorization-of* *fctrs* *p*

**shows** *is-alt-factorization-of* *fctrs* (*reflect-poly* *p*)

**using** *assms* **by** (*force simp: interp-factorization-reflect is-factorization-of-def*  
*is-alt-factorization-of-def coeff-0-interp-factorization*)

**lemma** *reflect-factorization'*:

**assumes** *coeff* *p*  $0 \neq (0 :: 'a :: idom)$

**assumes** *is-alt-factorization-of* *fctrs* *p*

**shows** *is-factorization-of* *fctrs* (*reflect-poly* *p*)

**using** *assms* **by** (*force simp: interp-alt-factorization-reflect is-factorization-of-def*  
*is-alt-factorization-of-def coeff-0-interp-factorization*)

**lemma** *zero-in-factorization-iff*:

**assumes** *is-factorization-of* *fctrs* *p*

**shows** *coeff* *p*  $0 = 0 \longleftrightarrow p = 0 \vee (0 :: 'a :: idom) \in \text{fst } ' \text{ set } (\text{snd } fctrs)$

**proof** (*cases*  $p = 0$ )

**assume**  $p \neq 0$

**with** *assms* **have** [*simp*]: *fst* *fctrs*  $\neq 0$

**by** (*auto simp: is-factorization-of-def interp-factorization-def case-prod-unfold*)

**from** *assms* **have**  $p = \text{interp-factorization } fctrs$  **by** (*simp add: is-factorization-of-def*)

**also** **have** *coeff* ...  $0 = 0 \longleftrightarrow 0 \in \text{fst } ' \text{ set } (\text{snd } fctrs)$

**by** (*force simp add: interp-factorization-def case-prod-unfold coeff-0-prod-list*  
*prod-list-zero-iff o-def coeff-0-power*)

**finally** **show** *?thesis* **using**  $\langle p \neq 0 \rangle$  **by** *blast*

**next**

**assume**  $p = 0$

**with** *assms* **have** *interp-factorization* *fctrs*  $= 0$  **by** (*simp add: is-factorization-of-def*)

**also** **have** *interp-factorization* *fctrs*  $= 0 \longleftrightarrow$

$\text{fst } fctrs = 0 \vee (\prod (c,n) \leftarrow \text{snd } fctrs. [:-c,1:] \hat{\sim} \text{Suc } n) = 0$

**by** (*simp add: interp-factorization-def case-prod-unfold*)

**also** **have**  $(\prod (c,n) \leftarrow \text{snd } fctrs. [:-c,1:] \hat{\sim} \text{Suc } n) = 0 \longleftrightarrow \text{False}$

**by** (*auto simp: prod-list-zero-iff simp del: power-Suc*)

**finally** **show** *?thesis* **by** (*simp add:  $\langle p = 0 \rangle$* )

**qed**

**lemma** *poly-prod-list* [*simp*]: *poly* (*prod-list* *ps*) *x*  $= \text{prod-list } (\text{map } (\lambda p. \text{poly } p \ x)$   
 $\text{ps})$

**by** (*induction* *ps*) *auto*

**lemma** *is-factorization-of-roots*:

**fixes**  $a :: 'a :: idom$

**assumes** *is-factorization-of*  $(a, fctrs) p p \neq 0$

**shows**  $set (map fst fctrs) = \{x. poly p x = 0\}$

**using** *assms*

**by** (*force simp: is-factorization-of-def interp-factorization-def o-def case-prod-unfold prod-list-zero-iff simp del: power-Suc*)

**lemma** (*in monoid-mult*) *prod-list-prod-nth*:  $prod-list\ xs = (\prod_{i < length\ xs} xs\ !\ i)$

**by** (*induction xs*) (*auto simp: prod.lessThan-Suc-shift simp del: prod.lessThan-Suc*)

**lemma** *order-prod*:

**assumes**  $\bigwedge x. x \in A \implies f\ x \neq 0$

**assumes**  $\bigwedge x\ y. x \in A \implies y \in A \implies x \neq y \implies coprime\ (f\ x)\ (f\ y)$

**shows**  $order\ c\ (prod\ f\ A) = (\sum_{x \in A}. order\ c\ (f\ x))$

**using** *assms*

**proof** (*induction A rule: infinite-finite-induct*)

**case** (*insert x A*)

**from** *insert.hyps* **have**  $order\ c\ (prod\ f\ (insert\ x\ A)) = order\ c\ (f\ x * prod\ f\ A)$

**by** *simp*

**also have**  $\dots = order\ c\ (f\ x) + order\ c\ (prod\ f\ A)$

**using** *insert.prem*s **and** *insert.hyps* **by** (*intro order-mult*) *auto*

**also have**  $order\ c\ (prod\ f\ A) = (\sum_{x \in A}. order\ c\ (f\ x))$

**using** *insert.prem*s **and** *insert.hyps* **by** (*intro insert.IH*) *auto*

**finally show** *?case* **using** *insert.hyps* **by** *simp*

**qed** *auto*

**lemma** *is-factorization-of-order*:

**fixes**  $p :: 'a :: field-gcd\ poly$

**assumes**  $p \neq 0$

**assumes** *is-factorization-of*  $(a, fctrs) p$

**assumes**  $(c, n) \in set\ fctrs$

**shows**  $order\ c\ p = Suc\ n$

**proof** –

**from** *assms* **have** *distinct: distinct*  $(map\ fst\ (fctrs))$

**by** (*simp add: is-factorization-of-def*)

**from** *assms* **have** [*simp*]:  $a \neq 0$

**by** (*auto simp: is-factorization-of-def interp-factorization-def*)

**from** *assms*(2) **have**  $p = interp-factorization\ (a, fctrs)$

**unfolding** *is-factorization-of-def* **by** *simp*

**also have**  $order\ c\ \dots = order\ c\ (\prod_{(c,n) \leftarrow fctrs}. [:-c, 1:] \wedge Suc\ n)$

**unfolding** *interp-factorization-def* **by** (*simp add: order-smult*)

**also have**  $(\prod_{(c,n) \leftarrow fctrs}. [:-c, 1:] \wedge Suc\ n) =$

$(\prod_{i \in \{..<length\ fctrs\}}. [:-fst\ (fctrs\ !\ i), 1:] \wedge Suc\ (snd\ (fctrs\ !\ i)))$

**by** (*simp add: prod-list-prod-nth case-prod-unfold*)

**also have**  $order\ c\ \dots =$

$(\sum_{x < length\ fctrs}. order\ c\ ([:-fst\ (fctrs\ !\ x), 1:] \wedge Suc\ (snd\ (fctrs\ !$

$x))))$

**proof** (*rule order-prod*)



```

fix  $i$ 
assume  $i \in \{..<length\ fctrs\}$ 
then show  $[: - fst (fctrs ! i), 1:] \wedge Suc (snd (fctrs ! i)) \neq 0$ 
  by (simp only: power-eq-0-iff) simp
next
fix  $i\ j :: nat$ 
assume  $i \neq j\ i \in \{..<length\ fctrs\}\ j \in \{..<length\ fctrs\}$ 
then have  $fst (fctrs ! i) \neq fst (fctrs ! j)$ 
  using nth-eq-iff-index-eq [OF distinct, of i j] by simp
then show coprime ( $[: - fst (fctrs ! i), 1:] \wedge Suc (snd (fctrs ! i))$ )
  ( $[: - fst (fctrs ! j), 1:] \wedge Suc (snd (fctrs ! j))$ )
  by (simp only: coprime-power-left-iff coprime-power-right-iff)
  (auto simp add: coprime-linear-poly)
qed
also have  $\dots = (\sum (c', n') \leftarrow fctrs. order\ c\ ([: - c', 1:] \wedge Suc\ n')$ 
  by (simp add: sum-list-sum-nth case-prod-unfold atLeast0LessThan)
also have  $\dots = (\sum (c', n') \leftarrow fctrs. if\ c = c'\ then\ Suc\ n'\ else\ 0)$ 
  by (intro arg-cong [OF map-cong]) (auto simp add: order-power-n-n order-0I)
simp del: power-Suc
also have  $\dots = (\sum x \leftarrow fctrs. if\ x = (c, n)\ then\ Suc\ (snd\ x)\ else\ 0)$ 
  using distinct assms by (intro arg-cong [OF map-cong]) (force simp: distinct-map)
inj-on-def) +
also from distinct have  $\dots = (\sum x \in set\ fctrs. if\ x = (c, n)\ then\ Suc\ (snd\ x)$ 
  else 0)
  by (intro sum-list-distinct-conv-sum-set) (simp-all add: distinct-map)
also from assms have  $\dots = Suc\ n$  by simp
finally show ?thesis .
qed

```

For complex polynomials, a factorisation in the above sense always exists.

**lemma** *complex-factorization-exists:*

$\exists fctrs. is-factorization-of\ fctrs\ (p :: complex\ poly)$

**proof** (*cases*  $p = 0$ )

**case** *True*

**thus** *?thesis*

**by** (*intro exI [of - (0, [])]*) (*auto simp: is-factorization-of-def interp-factorization-def*)

**next**

**case** *False*

**hence**  $\exists xs. set\ xs = \{x. poly\ p\ x = 0\} \wedge distinct\ xs$

**by** (*intro finite-distinct-list poly-roots-finite*)

**then obtain**  $xs$  **where** [*simp*]:  $set\ xs = \{x. poly\ p\ x = 0\}$  *distinct*  $xs$  **by** *blast*

**have** *interp-factorization* (*lead-coeff*  $p$ , *map*  $(\lambda x. (x, order\ x\ p - 1))\ xs$ ) =

*smult* (*lead-coeff*  $p$ )  $(\prod x \leftarrow xs. [: - x, 1:] \wedge Suc (order\ x\ p - 1))$

**by** (*simp add: interp-factorization-def o-def*)

**also have**  $(\prod x \leftarrow xs. [: - x, 1:] \wedge Suc (order\ x\ p - 1)) =$

$(\prod x | poly\ p\ x = 0. [: - x, 1:] \wedge Suc (order\ x\ p - 1))$

**by** (*subst prod.distinct-set-conv-list [symmetric]*) *simp-all*

**also have**  $\dots = (\prod x | poly\ p\ x = 0. [: - x, 1:] \wedge order\ x\ p)$

**proof** (*intro prod.cong refl, goal-cases*)

```

case (1 x)
with False have order x p ≠ 0 by (subst (asm) order-root) auto
hence *: Suc (order x p - 1) = order x p by simp
show ?case by (simp only: *)
qed
also have smult (lead-coeff p) ... = p
  by (rule complex-poly-decompose)
finally have is-factorization-of (lead-coeff p, map (λx. (x, order x p - 1)) xs) p
  by (auto simp: is-factorization-of-def o-def)
thus ?thesis ..
qed

```

By reflecting the polynomial, this means that for complex polynomials with non-zero constant coefficient, the alternative factorisation also exists.

```

corollary complex-alt-factorization-exists:
  assumes coeff p 0 ≠ 0
  shows ∃ fctrs. is-alt-factorization-of fctrs (p :: complex poly)
proof -
  from assms have coeff (reflect-poly p) 0 ≠ 0
    by auto
  moreover from complex-factorization-exists [of reflect-poly p]
  obtain fctrs where is-factorization-of fctrs (reflect-poly p) ..
  ultimately have is-alt-factorization-of fctrs (reflect-poly (reflect-poly p))
    by (rule reflect-factorization)
  also from assms have reflect-poly (reflect-poly p) = p
    by simp
  finally show ?thesis ..
qed
end

```

## 6 Solver for rational formal power series

```

theory Rational-FPS-Solver
imports
  Complex-Main
  Pochhammer-Polynomials
  Partial-Fraction-Decomposition
  Factorizations
  HOL-Computational-Algebra.Field-as-Ring
begin

```

We can determine the  $k$ -th coefficient of an FPS of the form  $d/(1 - cX)^n$ , which is an important step in solving linear recurrences. The  $k$ -th coefficient of such an FPS is always of the form  $p(k)c^k$  where  $p$  is the following polynomial:

```

definition inverse-irred-power-poly :: 'a :: field-char-0 ⇒ nat ⇒ 'a poly where
  inverse-irred-power-poly d n =

```

$Poly [(d * of\text{-}nat (stirling\ n\ (k+1))) / (fact\ (n - 1)).\ k \leftarrow [0..<n]]$

**lemma** *one-minus-const-fps-X-neg-power''*:

**fixes**  $c :: 'a :: field\text{-}char\text{-}0$

**assumes**  $n: n > 0$

**shows**  $fps\text{-}const\ d / ((1 - fps\text{-}const\ (c :: 'a :: field\text{-}char\text{-}0) * fps\text{-}X) ^ n) =$   
 $Abs\text{-}fps\ (\lambda k. poly\ (inverse\text{-}irred\text{-}power\text{-}poly\ d\ n)\ (of\text{-}nat\ k) * c ^ k)$  **(is ?lhs**  
 $= ?rhs)$

**proof** (*rule fps-ext*)

**include** *fps-notation*

**fix**  $k :: nat$

**let**  $?p = smult\ (d / (fact\ (n - 1)))\ (pcompose\ (pochhammer\text{-}poly\ (n - 1))$   
 $[:1,1:])$

**from**  $n$  **have**  $?lhs = fps\text{-}const\ d * inverse\ ((1 - fps\text{-}const\ c * fps\text{-}X) ^ n)$

**by** (*subst fps-divide-unit*) *auto*

**also have**  $inverse\ ((1 - fps\text{-}const\ c * fps\text{-}X) ^ n) =$

$Abs\text{-}fps\ (\lambda k. of\text{-}nat\ ((n + k - 1)\ choose\ k) * c ^ k)$

**by** (*intro one-minus-const-fps-X-neg-power' n*)

**also have**  $(fps\text{-}const\ d * \dots) \$ k = d * of\text{-}nat\ ((n + k - 1)\ choose\ k) * c ^ k$

**by** *simp*

**also from**  $n$  **have**  $(n + k - 1\ choose\ k) = (n + k - 1\ choose\ (n - 1))$

**by** (*subst binomial-symmetric*) *simp-all*

**also from**  $n$  **have**  $of\text{-}nat\ \dots = (pochhammer\ (of\text{-}nat\ k + 1)\ (n - 1) / fact\ (n$   
 $- 1)) :: 'a$

**by** (*simp-all add: binomial-gbinomial gbinomial-pochhammer' of-nat-diff*)

**also have**  $d * \dots = poly\ ?p\ (of\text{-}nat\ k)$

**by** (*simp add: divide-inverse eval-pochhammer-poly poly-pcompose add-ac*)

**also** {

**from** *assms* **have**  $pCons\ 0\ (pcompose\ (pochhammer\text{-}poly\ (n-1))\ [:1,1>::'a:]) =$   
 $pochhammer\text{-}poly\ n$

**by** (*subst pochhammer-poly-Suc' [symmetric]*) *simp*

**also from** *assms* **have**  $\dots = pCons\ 0\ (Poly\ [of\text{-}nat\ (stirling\ n\ (k+1)).\ k \leftarrow$   
 $[0..<Suc\ n]])$

**unfolding** *pochhammer-poly-def*

**by** (*auto simp add: poly-eq-iff nth-default-def coeff-pCons*

*split: nat.split simp del: upt-Suc*)

**finally have**  $pcompose\ (pochhammer\text{-}poly\ (n-1))\ [:1,1>::'a:] =$

$Poly\ [of\text{-}nat\ (stirling\ n\ (k+1)).\ k \leftarrow [0..<Suc\ n]]$  **by** *simp*

}

**also have**  $smult\ (d / fact\ (n - 1))\ (Poly\ [of\text{-}nat\ (stirling\ n\ (k+1)).\ k \leftarrow [0..<Suc$   
 $n]]) =$

$inverse\text{-}irred\text{-}power\text{-}poly\ d\ n$

**by** (*auto simp: poly-eq-iff inverse-irred-power-poly-def nth-default-def*)

**also have**  $poly\ \dots\ (of\text{-}nat\ k) * c ^ k = ?rhs\ \$ k$  **by** *simp*

**finally show**  $?lhs\ \$ k = ?rhs\ \$ k$  .

**qed**

**lemma** *inverse-irred-power-poly-code [code abstract]*:

$coeffs\ (inverse\text{-}irred\text{-}power\text{-}poly\ d\ n) =$

```

    (if n = 0 ∨ d = 0 then [] else
     let e = d / (fact (n - 1))
     in [e * of-nat x. x ← tl (stirling-row n)])
proof (cases n = 0 ∨ d = 0)
case False
define e where e = d / (fact (n - 1))
from False have coeffs (inverse-irred-power-poly d n) =
    [e * of-nat (stirling n (k+1)). k ← [0..<n]]
by (auto simp: inverse-irred-power-poly-def Let-def divide-inverse mult-ac last-map
    stirling-row-def map-tl [symmetric] tl-upt e-def no-trailing-unfold)
also have ... = [e * of-nat x. x ← tl (stirling-row n)]
by (simp add: stirling-row-def map-tl [symmetric] o-def tl-upt
    map-Suc-upt [symmetric] del: upt-Suc)
finally show ?thesis using False by (simp add: Let-def e-def)
qed (auto simp: inverse-irred-power-poly-def)

```

**lemma** solve-rat-fps-aux:

```

fixes p :: 'a :: {field-char-0,field-gcd} poly and cs :: ('a × nat) list
assumes distinct: distinct (map fst cs)
assumes azs: (a, zs) = poly-pfd-simple p cs
assumes nz: 0 ∉ fst ` set cs
shows fps-of-poly p / fps-of-poly (∏ (c,n)←cs. [:1,-c:] ^ Suc n) =
    Abs-fps (λk. coeff a k + (∑ i<length cs. poly (∑ j≤snd (cs ! i).
    (inverse-irred-power-poly (zs ! i ! j) (snd (cs ! i)+1 - j)))
    (of-nat k) * (fst (cs ! i) ^ k)) (is - = ?rhs)

```

**proof** –

```

interpret pfd-homomorphism fps-of-poly :: 'a poly ⇒ 'a fps
by standard (auto simp: fps-of-poly-add fps-of-poly-mult)
from distinct have distinct': (a, b1) ∈ set cs ⇒
    (a, b2) ∈ set cs ⇒ b1 = b2 for a b1 b2
by (metis (no-types, opaque-lifting) Some-eq-map-of-iff image-set in-set-zipE
    insert-iff list.simps(15) map-of-Cons-code(2) map-of-SomeD nz snd-conv)
from nz have nz': (0, b) ∉ set cs for b
by (auto simp add: image-iff)
define n where n = length cs
let ?g = λ(c, n). [:1, - c:] ^ Suc n
have inj-on ?g (set cs)
proof
fix x y
assume x ∈ set cs y ∈ set cs ?g x = ?g y
moreover obtain c1 n1 c2 n2 where [simp]: x = (c1, n1) y = (c2, n2)
by (cases x, cases y)
ultimately have in-cs: (c1, n1) ∈ set cs
    (c2, n2) ∈ set cs
and eq: [:1, - c1:] ^ Suc n1 = [:1, - c2:] ^ Suc n2
by simp-all
with nz have [simp]: c1 ≠ 0 c2 ≠ 0
by (auto simp add: image-iff)
have Suc n1 = degree ([:1, - c1:] ^ Suc n1)

```

by (simp add: degree-power-eq del: power-Suc)  
 also have ... = degree ([:1, - c2:] ^ Suc n2)  
 using eq by simp  
 also have ... = Suc n2  
 by (simp add: degree-power-eq del: power-Suc)  
 finally have n1 = n2 by simp  
 then have 0 = poly ([:1, - c1:] ^ Suc n1) (1 / c1)  
 by simp  
 also have ... = poly ([:1, - c2:] ^ Suc n2) (1 / c1)  
 using eq by simp  
 finally show x = y using ⟨n1 = n2⟩  
 by (auto simp: field-simps)

**qed**  
 with distinct have distinct': distinct (map ?g cs)  
 by (simp add: distinct-map del: power-Suc)  
 from nz' distinct have coprime: pairwise coprime (?g ' set cs)  
 by (auto intro!: pairwise-imageI coprime-linear-poly' simp add: eq-key-imp-eq-value  
 simp del: power-Suc)  
 have [simp]: length zs = n  
 using assms by (simp add: poly-pfd-simple-def n-def split: if-split-asm)  
 have [simp]: i < length cs  $\implies$  length (zs!i) = snd (cs!i)+1 for i  
 using assms by (simp add: poly-pfd-simple-def Let-def case-prod-unfold split:  
 if-split-asm)

let ?f =  $\lambda(c, n). ([:1, -c:], n)$   
 let ?cs' = map ?f cs  
 have fps-of-poly (fst (poly-pfd-simple p cs)) +  
 ( $\sum i < \text{length } ?cs'. \sum j \leq \text{snd } (?cs' ! i).$   
 from-decomp (map (map ( $\lambda c. [:c:]$ )) (snd (poly-pfd-simple p cs)) ! i ! j)  
 (fst (?cs' ! i) (snd (?cs' ! i)+1 - j)) =  
 fps-of-poly p / fps-of-poly ( $\prod (x, n) \leftarrow ?cs'. x \wedge \text{Suc } n$ )  
 (is ?A = ?B) using nz distinct' coprime  
 by (intro partial-fraction-decomposition poly-pfd-simple)  
 (force simp: o-def case-prod-unfold simp del: power-Suc)+  
 note this [symmetric]  
 also from azs [symmetric]  
 have ?A = fps-of-poly a + ( $\sum i < n. \sum j \leq \text{snd } (cs ! i).$  from-decomp  
 (map (map ( $\lambda c. [:c:]$ )) zs ! i ! j) [:1, -fst (cs ! i):] (snd (cs ! i)+1 -  
 j))  
 (is - = - + ?S) by (simp add: case-prod-unfold Let-def n-def)  
 also have ?S = ( $\sum i < \text{length } cs. \sum j \leq \text{snd } (cs ! i).$  fps-const (zs ! i ! j) /  
 ((1 - fps-const (fst (cs!i))\*fps-X) ^ (snd (cs!i)+1 - j)))  
 by (intro sum.cong refl)  
 (auto simp: from-decomp-def map-nth n-def fps-of-poly-linear' fps-of-poly-simps  
 fps-const-neg [symmetric] mult-ac simp del: fps-const-neg)  
 also have ... = ( $\sum i < \text{length } cs. \sum j \leq \text{snd } (cs ! i).$   
 Abs-fps ( $\lambda k. \text{poly } (\text{inverse-irred-power-poly } (zs ! i ! j)$   
 (snd (cs ! i)+1 - j)) (of-nat k) \* (fst (cs ! i)) ^ k))  
 using nz by (intro sum.cong refl one-minus-const-fps-X-neg-power'') auto

also have  $\text{fps-of-poly } a + \dots = ?rhs$   
 by (intro  $\text{fps-ext}$ ) (simp-all add:  $\text{sum-distrib-right fps-sum-nth poly-sum}$ )  
 finally show  $?thesis$  by (simp add:  $\text{o-def case-prod-unfold}$ )  
 qed

**definition**  $\text{solve-factored-ratfps} ::$

$('a :: \{\text{field-char-0, field-gcd}\}) \text{poly} \Rightarrow ('a \times \text{nat}) \text{list} \Rightarrow 'a \text{poly} \times ('a \text{poly} \times 'a)$   
 list where

$\text{solve-factored-ratfps } p \text{ cs} = (\text{let } n = \text{length } cs \text{ in case poly-pfd-simple } p \text{ cs of } (a, zs) \Rightarrow$   
 $(a, \text{zip-with } (\lambda zs (c,n). ((\sum (z,j) \leftarrow \text{zip } zs [0..<Suc } n].$   
 $\text{inverse-irred-power-poly } z (n + 1 - j)), c)) \text{ } zs \text{ } cs))$

**lemma**  $\text{length-snd-poly-pfd-simple [simp]: length (snd (poly-pfd-simple } p \text{ cs)) = length } cs$

by (simp add:  $\text{poly-pfd-simple-def}$ )

**lemma**  $\text{length-nth-snd-poly-pfd-simple [simp]:}$

$i < \text{length } cs \implies \text{length (snd (poly-pfd-simple } p \text{ cs) ! } i) = \text{snd (cs!} i) + 1$

by (auto simp:  $\text{poly-pfd-simple-def case-prod-unfold Let-def}$ )

**lemma**  $\text{solve-factored-ratfps-roots:}$

$\text{map snd (snd (solve-factored-ratfps } p \text{ cs))} = \text{map fst } cs$

by (rule  $\text{nth-equalityI}$ )

(simp-all add:  $\text{solve-factored-ratfps-def poly-pfd-simple case-prod-unfold Let-def zip-with-altdef o-def}$ )

**definition**  $\text{interp-ratfps-solution}$  where

$\text{interp-ratfps-solution} = (\lambda(p,cs) n. \text{coeff } p \text{ } n + (\sum (q,c) \leftarrow cs. \text{poly } q \text{ (of-nat } n) * c ^ n))$

**lemma**  $\text{solve-factored-ratfps:}$

fixes  $p :: 'a :: \{\text{field-char-0, field-gcd}\} \text{poly}$  and  $cs :: ('a \times \text{nat}) \text{list}$

assumes  $\text{distinct: distinct (map fst } cs)$

assumes  $\text{nz: } 0 \notin \text{fst ' set } cs$

shows  $\text{fps-of-poly } p / \text{fps-of-poly } (\prod (c,n) \leftarrow cs. [:1, -c:] ^ Suc } n) =$

$\text{Abs-fps (interp-ratfps-solution (solve-factored-ratfps } p \text{ cs)) (is ?lhs = ?rhs)}$

**proof** –

obtain  $a \text{ } zs$  where  $\text{azs: } (a, zs) = \text{solve-factored-ratfps } p \text{ } cs$

using  $\text{prod.exhaust}$  by  $\text{metis}$

from  $\text{azs}$  have  $a: a = \text{fst (poly-pfd-simple } p \text{ } cs)$

by (simp add:  $\text{solve-factored-ratfps-def Let-def case-prod-unfold}$ )

define  $zs'$  where  $zs' = \text{snd (poly-pfd-simple } p \text{ } cs)$

with  $a$  have  $\text{azs': } (a, zs') = \text{poly-pfd-simple } p \text{ } cs$  by  $\text{simp}$

from  $\text{azs}$  have  $zs: zs = \text{snd (solve-factored-ratfps } p \text{ } cs)$

by (auto simp add:  $\text{snd-def split: prod.split}$ )

**have**  $?lhs = Abs\text{-}fps (\lambda k. coeff\ a\ k + (\sum i < length\ cs. poly (\sum j \leq snd\ (cs\ !\ i). inverse\text{-}irred\text{-}power\text{-}poly\ (zs'\ !\ i\ !\ j)\ (snd\ (cs\ !\ i) + 1 - j)) (of\text{-}nat\ k) * (fst\ (cs\ !\ i)) ^ k))$   
**by** (*rule solve-rat-fps-aux*[*OF distinct azs' nz*])  
**also from** *azs* **have**  $\dots = ?rhs$  **unfolding** *interp-ratfps-solution-def*  
**by** (*auto simp: a zs solve-factored-ratfps-def Let-def case-prod-unfold zip-altdef zip-with-altdef' sum-list-sum-nth atLeast0LessThan zs'-def lessThan-Suc-atMost intro!: fps-ext sum.cong simp del: upt-Suc*)  
**finally show** *?thesis* .  
**qed**

**definition** *solve-factored-ratfps'* **where**

$solve\text{-}factored\text{-}ratfps' = (\lambda p\ (a, cs). solve\text{-}factored\text{-}ratfps\ (smult\ (inverse\ a)\ p)\ cs)$

**lemma** *solve-factored-ratfps'*:

**assumes** *is-alt-factorization-of fctrs q q ≠ 0*

**shows**  $Abs\text{-}fps\ (interp\text{-}ratfps\text{-}solution\ (solve\text{-}factored\text{-}ratfps'\ p\ fctrs)) = fps\text{-}of\text{-}poly\ p / fps\text{-}of\text{-}poly\ q$

**proof** –

**from** *assms* **have**  $q: q = interp\text{-}alt\text{-}factorization\ fctrs$

**by** (*simp add: is-alt-factorization-of-def*)

**from** *assms*(2) **have**  $nz: fst\ fctrs \neq 0$

**by** (*subst (asm) q (auto simp: interp-alt-factorization-def case-prod-unfold)*)

**note**  $q$

**also from** *nz* **have**  $coeff\ (interp\text{-}alt\text{-}factorization\ fctrs)\ 0 \neq 0$

**by** (*auto simp: interp-alt-factorization-def case-prod-unfold coeff-0-prod-list o-def coeff-0-power prod-list-zero-iff*)

**finally have**  $coeff\ q\ 0 \neq 0$  .

**obtain**  $a\ cs$  **where**  $fctrs: fctrs = (a, cs)$  **by** (*cases fctrs simp-all*)

**obtain**  $b\ zs$  **where**  $sol: solve\text{-}factored\text{-}ratfps'\ p\ fctrs = (b, zs)$  **using** *prod.exhaust*  
**by** *metis*

**from** *assms* **have** [*simp*]:  $a \neq 0$

**by** (*auto simp: is-alt-factorization-of-def interp-alt-factorization-def fctrs*)

**have**  $fps\text{-}of\text{-}poly\ p / fps\text{-}of\text{-}poly\ (smult\ a\ (\prod (c, n) \leftarrow cs. [:1, - c:] ^ Suc\ n)) = fps\text{-}of\text{-}poly\ p / (fps\text{-}const\ a * fps\text{-}of\text{-}poly\ (\prod (c, n) \leftarrow cs. [:1, - c:] ^ Suc\ n))$

**by** (*simp-all add: fps-of-poly-smult case-prod-unfold del: power-Suc*)

**also have**  $\dots = fps\text{-}of\text{-}poly\ p / fps\text{-}const\ a / fps\text{-}of\text{-}poly\ (\prod (c, n) \leftarrow cs. [:1, - c:] ^ Suc\ n)$

**by** (*subst is-unit-div-mult2-eq*)

(*auto simp: coeff-0-power coeff-0-prod-list prod-list-zero-iff*)

**also have**  $fps\text{-}of\text{-}poly\ p / fps\text{-}const\ a = fps\text{-}of\text{-}poly\ (smult\ (inverse\ a)\ p)$

**by** (*simp add: fps-const-inverse fps-divide-unit*)

**also from** *assms* **have**  $smult\ a\ (\prod (c, n) \leftarrow cs. [:1, - c:] ^ Suc\ n) = q$

**by** (*simp add: is-alt-factorization-of-def interp-alt-factorization-def fctrs del:*)

*power-Suc*)  
**also have** *fps-of-poly* (*smult* (*inverse a*) *p*) /  
*fps-of-poly* ( $\prod (c, n) \leftarrow cs. [:1, - c:] \wedge Suc\ n =$   
*Abs-fps* (*interp-ratfps-solution* (*solve-factored-ratfps* (*smult* (*inverse a*)  
*p*) *cs*))  
**(is** *?lhs = -*) **using** *assms*  
**by** (*intro solve-factored-ratfps*)  
*(simp-all add: is-alt-factorization-of-def fctrs solve-factored-ratfps'-def)*  
**also have**  $\dots = Abs-fps$  (*interp-ratfps-solution* (*solve-factored-ratfps'* *p fctrs*))  
**by** (*simp add: solve-factored-ratfps'-def fctrs*)  
**finally show** *?thesis ..*  
**qed**

**lemma** *degree-Poly-eq*:  
**assumes**  $xs = [] \vee last\ xs \neq 0$   
**shows**  $degree\ (Poly\ xs) = length\ xs - 1$   
**proof** –  
**from** *assms* **consider**  $xs = [] \mid xs \neq []\ last\ xs \neq 0$  **by** *blast*  
**thus** *?thesis*  
**proof** *cases*  
**assume**  $last\ xs \neq 0\ xs \neq []$   
**hence** *no-trailing* ( $(=)\ 0$ ) *xs* **by** (*auto simp: no-trailing-unfold*)  
**thus** *?thesis* **by** (*simp add: degree-eq-length-coeffs*)  
**qed** *auto*  
**qed**

**lemma** *degree-Poly'*:  $degree\ (Poly\ xs) \leq length\ xs - 1$   
**using** *length-strip-while-le*[*of* ( $(=)\ 0\ xs$ )] **by** (*simp add: degree-eq-length-coeffs*)

**lemma** *degree-inverse-irred-power-poly-le*:  
 $degree\ (inverse-irred-power-poly\ c\ n) \leq n - 1$   
**by** (*auto simp: inverse-irred-power-poly-def intro: order.trans[OF degree-Poly']*)

**lemma** *degree-inverse-irred-power-poly*:  
**assumes**  $c \neq 0$   
**shows**  $degree\ (inverse-irred-power-poly\ c\ n) = n - 1$   
**unfolding** *inverse-irred-power-poly-def* **using** *assms*  
**by** (*subst degree-Poly-eq*) (*auto simp: last-conv-nth*)

**lemma** *reflect-poly-0-iff* [*simp*]:  $reflect-poly\ p = 0 \iff p = 0$   
**using** *coeff-0-reflect-poly-0-iff*[*of p*] **by** *fastforce*

**lemma** *degree-sum-list-le*:  $(\bigwedge p. p \in set\ ps \implies degree\ p \leq T) \implies degree\ (sum-list\ ps) \leq T$   
**by** (*induction ps*) (*auto intro: degree-add-le*)

**theorem** *ratfps-closed-form-exists*:  
**fixes**  $q :: complex\ poly$



```

assumes nz: coeff q 0 ≠ 0
defines q' ≡ reflect-poly q
obtains r rs
where  $\bigwedge n. \text{fps-nth } (\text{fps-of-poly } p / \text{fps-of-poly } q) n =$ 
       $\text{coeff } r n + (\sum c \mid \text{poly } q' c = 0. \text{poly } (rs c) (\text{of-nat } n) * c \wedge n)$ 
and  $\bigwedge z. \text{poly } q' z = 0 \implies \text{degree } (rs z) \leq \text{order } z q' - 1$ 
proof –
from assms have nz': q ≠ 0 by auto
from complex-alt-factorization-exists [OF nz]
obtain fctrs where fctrs: is-alt-factorization-of fctrs q ..
with nz have fctrs': is-factorization-of fctrs q' unfolding q'-def
  by (rule reflect-factorization')
define r where r = fst (solve-factored-ratfps' p fctrs)
define ts where ts = snd (solve-factored-ratfps' p fctrs)
define rs where rs = the ∘ map-of (map (λ(x,y). (y,x)) ts)

from nz' have q' ≠ 0 by (simp add: q'-def)
hence roots: {z. poly q' z = 0} = set (map fst (snd fctrs))
  using is-factorization-of-roots [of fst fctrs snd fctrs q'] fctrs' by simp

have rs c = r if (r, c) ∈ set ts for c r
proof –
  have map-of (map (λ(x,y). (y, x)) (snd (solve-factored-ratfps' p fctrs))) c =
  Some r
    using that fctrs
    by (intro map-of-is-SomeI)
      (force simp: o-def case-prod-unfold solve-factored-ratfps'-def ts-def
        solve-factored-ratfps-roots is-alt-factorization-of-def) +
    thus ?thesis by (simp add: rs-def ts-def)
qed

have [simp]: length ts = length (snd fctrs)
by (auto simp: ts-def solve-factored-ratfps'-def case-prod-unfold solve-factored-ratfps-def)

{
  fix n :: nat
  have fps-of-poly p / fps-of-poly q =
    Abs-fps (interp-ratfps-solution (solve-factored-ratfps' p fctrs))
    using solve-factored-ratfps' [OF fctrs nz'] ..
  also have fps-nth ... n = interp-ratfps-solution (solve-factored-ratfps' p fctrs)
  by simp
  also have ... = coeff r n + ( $\sum p \leftarrow \text{snd } (\text{solve-factored-ratfps}' p \text{ fctrs}).$ 
     $\text{poly } (\text{fst } p) (\text{of-nat } n) * \text{snd } p \wedge n$ ) (is - = - + ?A)
    unfolding interp-ratfps-solution-def case-prod-unfold r-def by simp
  also have ?A = ( $\sum p \leftarrow \text{ts}. \text{poly } (rs (\text{snd } p)) (\text{of-nat } n) * \text{snd } p \wedge n$ )
    by (intro arg-cong[OF map-cong] refl) (auto simp: rs ts-def)
  also have ... = ( $\sum c \leftarrow \text{map } \text{snd } \text{ts}.$ 
     $\text{poly } (rs c) (\text{of-nat } n) * c \wedge n$ ) by (simp add: o-def)
}

```

```

also have map snd ts = map fst (snd fctrs)
  unfolding solve-factored-ratfps'-def case-prod-unfold ts-def
  by (rule solve-factored-ratfps-roots)
also have  $(\sum c \leftarrow \dots \text{poly } (rs \ c) \ (of\text{-nat } n) * c \wedge n) =$ 
   $(\sum c \mid \text{poly } q' \ c = 0. \text{poly } (rs \ c) \ (of\text{-nat } n) * c \wedge n)$  unfolding roots
using fctrs by (intro sum-list-distinct-conv-sum-set) (auto simp: is-alt-factorization-of-def)
finally have fps-nth (fps-of-poly p / fps-of-poly q) n =
  coeff r n +  $(\sum c \in \{z. \text{poly } q' \ z = 0\}. \text{poly } (rs \ c) \ (of\text{-nat } n) * c \wedge$ 
n) .
} moreover {
  fix z assume poly q' z = 0
  hence z  $\in$  set (map fst (snd fctrs)) using roots by blast
  then obtain i where i: i < length (snd fctrs) and [simp]: z = fst (snd fctrs !
i)
  by (auto simp: set-conv-nth)
from i have (fst (ts ! i), snd (ts ! i))  $\in$  set ts
  by (auto simp: set-conv-nth)
also from i have snd (ts ! i) = z
  by (simp add: ts-def solve-factored-ratfps'-def case-prod-unfold solve-factored-ratfps-def)
finally have rs z = fst (ts ! i) by (intro rs) auto
also have ... =  $(\sum p \leftarrow \text{zip } (snd \ (poly\text{-pfd-simple } (smult \ (inverse \ (fst \ fctrs)) \ p)$ 
(snd fctrs)) ! i)
  [0..<Suc (snd (snd fctrs ! i))].
  inverse-irred-power-poly (fst p) (Suc (snd (snd fctrs ! i)) - snd
p))
using i by (auto simp: ts-def solve-factored-ratfps'-def solve-factored-ratfps-def
o-def
  case-prod-unfold Let-def simp del: upt-Suc power-Suc)
also have degree ...  $\leq$  snd (snd fctrs ! i)
  by (intro degree-sum-list-le)
  (auto intro!: order.trans [OF degree-inverse-irred-power-poly-le])
also have order z q' = Suc ...
  using nz' fctrs' i
  by (intro is-factorization-of-order[of q' fst fctrs snd fctrs]) (auto simp: q'-def)
hence snd (snd fctrs ! i) = order z q' - 1 by simp
finally have degree (rs z)  $\leq$  ... .
}
ultimately show ?thesis
using that[of r rs] by blast
qed
end

```

## 7 Material common to homogenous and inhomogenous linear recurrences

```

theory Linear-Recurrences-Common
imports

```

*Complex-Main*  
*HOL-Computational-Algebra.Computational-Algebra*  
**begin**

**definition** *lr-fps-denominator* **where**  
*lr-fps-denominator cs = Poly (rev cs)*

**lemma** *lr-fps-denominator-code* [*code abstract*]:  
*coeffs (lr-fps-denominator cs) = rev (dropWhile ((=) 0) cs)*  
**by** (*simp add: lr-fps-denominator-def*)

**definition** *lr-fps-denominator'* **where**  
*lr-fps-denominator' cs = Poly cs*

**lemma** *lr-fps-denominator'-code* [*code abstract*]:  
*coeffs (lr-fps-denominator' cs) = strip-while ((=) 0) cs*  
**by** (*simp add: lr-fps-denominator'-def*)

**lemma** *lr-fps-denominator-nz*: *last cs ≠ 0 ⇒ cs ≠ [] ⇒ lr-fps-denominator cs ≠ 0*

**unfolding** *lr-fps-denominator-def*  
**by** (*subst coeffs-eq-iff*) (*auto simp: poly-eq-iff intro!: bexI[of - last cs]*)

**lemma** *lr-fps-denominator'-nz*: *last cs ≠ 0 ⇒ cs ≠ [] ⇒ lr-fps-denominator' cs ≠ 0*

**unfolding** *lr-fps-denominator'-def*  
**by** (*subst coeffs-eq-iff*) (*auto simp: poly-eq-iff intro!: bexI[of - last cs]*)

**end**

## 8 Homogenous linear recurrences

**theory** *Linear-Homogenous-Recurrences*  
**imports**

*Complex-Main*  
*RatFPS*  
*Rational-FPS-Solver*  
*Linear-Recurrences-Common*

**begin**

The following is the numerator of the rational generating function of a linear homogenous recurrence.

**definition** *lhr-fps-numerator* **where**

*lhr-fps-numerator m cs f = (let N = length cs - 1 in*  
*Poly [( $\sum_{i \leq \min N k. cs ! (N - i) * f (k - i)$ ). k ← [0.. $N+m$ ]])*

**lemma** *lhr-fps-numerator-code* [*code abstract*]:

*coeffs (lhr-fps-numerator m cs f) = (let N = length cs - 1 in*  
*strip-while ((=) 0) [( $\sum_{i \leq \min N k. cs ! (N - i) * f (k - i)$ ). k ← [0.. $N+m$ ]])*

by (simp add: lhr-fps-numerator-def Let-def)

**lemma** *lhr-fps-aux*:

fixes  $f :: \text{nat} \Rightarrow 'a :: \text{field}$

assumes  $\bigwedge n. n \geq m \implies (\sum k \leq N. c k * f (n + k)) = 0$

assumes  $cN: c N \neq 0$

defines  $p \equiv \text{Poly } [c (N - k). k \leftarrow [0..<\text{Suc } N]]$

defines  $q \equiv \text{Poly } [(\sum i \leq \min N k. c (N - i) * f (k - i)). k \leftarrow [0..<N+m]]$

shows  $\text{Abs-fps } f = \text{fps-of-poly } q / \text{fps-of-poly } p$

**proof** –

include *fps-notation*

define  $F$  where  $F = \text{Abs-fps } f$

have [simp]:  $F \$ n = f n$  for  $n$  by (simp add: *F-def*)

have [simp]:  $\text{coeff } p 0 = c N$

by (simp add: *p-def nth-default-def del: upt-Suc*)

have  $(\text{fps-of-poly } p * F) \$ n = \text{coeff } q n$  for  $n$

**proof** (*cases*  $n \geq N + m$ )

case *True*

let  $?f = \lambda i. N - i$

have  $(\text{fps-of-poly } p * F) \$ n = (\sum i \leq n. \text{coeff } p i * f (n - i))$

by (simp add: *fps-mult-nth atLeast0AtMost*)

also from *True* have  $\dots = (\sum i \leq N. \text{coeff } p i * f (n - i))$

by (*intro sum.mono-neutral-right*) (*auto simp: nth-default-def p-def*)

also have  $\dots = (\sum i \leq N. c (N - i) * f (n - i))$

by (*intro sum.cong*) (*auto simp: nth-default-def p-def simp del: upt-Suc*)

also from *True* have  $\dots = (\sum i \leq N. c i * f (n - N + i))$

by (*intro sum.reindex-bij-witness[of - ?f ?f]*) *auto*

also from *True* have  $\dots = 0$  by (*intro assms*) *simp-all*

also from *True* have  $\dots = \text{coeff } q n$

by (simp add: *q-def nth-default-def del: upt-Suc*)

finally show *?thesis* .

next

case *False*

hence  $(\text{fps-of-poly } p * F) \$ n = (\sum i \leq n. \text{coeff } p i * f (n - i))$

by (simp add: *fps-mult-nth atLeast0AtMost*)

also have  $\dots = (\sum i \leq \min N n. \text{coeff } p i * f (n - i))$

by (*intro sum.mono-neutral-right*)

(*auto simp: p-def nth-default-def simp del: upt-Suc*)

also have  $\dots = (\sum i \leq \min N n. c (N - i) * f (n - i))$

by (*intro sum.cong*) (*simp-all add: p-def nth-default-def del: upt-Suc*)

also from *False* have  $\dots = \text{coeff } q n$  by (simp add: *q-def nth-default-def*)

finally show *?thesis* .

qed

hence  $\text{fps-of-poly } p * F = \text{fps-of-poly } q$

by (*intro fps-ext*) *simp*

with  $cN$  show  $F = \text{fps-of-poly } q / \text{fps-of-poly } p$

by (*subst unit-eq-div2*) (*simp-all add: mult-ac*)

qed

**lemma** *lhr-fps*:

**fixes**  $f :: \text{nat} \Rightarrow 'a :: \text{field}$  **and**  $cs :: 'a \text{ list}$

**defines**  $N \equiv \text{length } cs - 1$

**assumes**  $cs: cs \neq []$

**assumes**  $\bigwedge n. n \geq m \implies (\sum k \leq N. cs ! k * f (n + k)) = 0$

**assumes**  $cN: \text{last } cs \neq 0$

**shows**  $\text{Abs-fps } f = \text{fps-of-poly } (\text{lhr-fps-numerator } m \text{ } cs \text{ } f) / \text{fps-of-poly } (\text{lr-fps-denominator } cs)$

**proof** –

**define**  $p$  **and**  $q$

**where**  $p = \text{Poly } (\text{map } (\lambda k. \sum i \leq \min N \ k. cs ! (N - i) * f (k - i)) [0..<N + m])$

**and**  $q = \text{Poly } (\text{map } (\lambda k. cs ! (N - k)) [0..<Suc N])$

**from** *assms* **have**  $\text{Abs-fps } f = \text{fps-of-poly } p / \text{fps-of-poly } q$  **unfolding**  $p\text{-def } q\text{-def}$

**by** (*intro lhr-fps-aux*) (*simp-all add: last-conv-nth*)

**also** **have**  $p = \text{lhr-fps-numerator } m \text{ } cs \text{ } f$

**unfolding**  $p\text{-def } \text{lhr-fps-numerator-def}$  **by** (*auto simp: Let-def N-def*)

**also** **from**  $cN$  **have**  $q = \text{lr-fps-denominator } cs$

**unfolding**  $q\text{-def } \text{lr-fps-denominator-def}$

**by** (*intro poly-eqI*)

(*auto simp add: nth-default-def rev-nth N-def not-less cs simp del: upt-Suc*)

**finally** **show** *?thesis* .

**qed**

**fun** *lhr* **where**

$\text{lhr } cs \text{ } fs \text{ } n =$

(*if* ( $cs :: 'a :: \text{field list}$ ) =  $[] \vee \text{last } cs = 0 \vee \text{length } fs < \text{length } cs - 1$  *then* *undefined* *else*

(*if*  $n < \text{length } fs$  *then*  $fs ! n$  *else*

( $\sum k < \text{length } cs - 1. cs ! k * \text{lhr } cs \text{ } fs (n + 1 - \text{length } cs + k)$ ) /  $-\text{last } cs$ ))

**declare**  $\text{lhr.simps} [simp \text{ del}]$

**lemma** *lhr-rec*:

**assumes**  $cs \neq []$   $\text{last } cs \neq 0$   $\text{length } fs \geq \text{length } cs - 1$   $n \geq \text{length } fs$

**shows**  $(\sum k < \text{length } cs. cs ! k * \text{lhr } cs \text{ } fs (n + 1 - \text{length } cs + k)) = 0$

**proof** –

**from** *assms* **have**  $\{..<\text{length } cs\} = \text{insert } (\text{length } cs - 1) \{..<\text{length } cs - 1\}$  **by** *auto*

**also** **have**  $(\sum k \in \dots . cs ! k * \text{lhr } cs \text{ } fs (n + 1 - \text{length } cs + k)) =$

$(\sum k < \text{length } cs - 1. cs ! k * \text{lhr } cs \text{ } fs (n + 1 - \text{length } cs + k)) +$

$\text{last } cs * \text{lhr } cs \text{ } fs \text{ } n$  **using** *assms*

**by** (*cases cs*) (*simp-all add: algebra-simps last-conv-nth*)

**also** **from** *assms* **have**  $\dots = 0$  **by** (*subst (2) lhr.simps*) (*simp-all add: field-simps*)

**finally** **show** *?thesis* .

qed

lemma *lhrI*:

assumes  $cs \neq []$   $last\ cs \neq 0$   $length\ fs \geq length\ cs - 1$   
assumes  $\bigwedge n. n < length\ fs \implies f\ n = fs\ !\ n$   
assumes  $\bigwedge n. n \geq length\ fs \implies (\sum_{k < length\ cs} cs\ !\ k * f\ (n + 1 - length\ cs + k)) = 0$   
shows  $f\ n = lhr\ cs\ fs\ n$   
using *assms*  
proof (*induction cs fs n rule: lhr.induct*)  
case ( $1\ cs\ fs\ n$ )  
show ?*case*  
proof (*cases n < length fs*)  
case *False*  
with  $1$  have  $0 = (\sum_{k < length\ cs} cs\ !\ k * f\ (n + 1 - length\ cs + k))$  by *simp*  
also from  $1$  have  $\{.. < length\ cs\} = insert\ (length\ cs - 1)\ \{.. < length\ cs - 1\}$   
by *auto*  
also have  $(\sum_{k \in \dots} cs\ !\ k * f\ (n + 1 - length\ cs + k)) =$   
 $(\sum_{k < length\ cs - 1} cs\ !\ k * f\ (n + 1 - length\ cs + k)) +$   
 $last\ cs * f\ n$  using  $1\ False$   
by (*cases cs*) (*simp-all add: algebra-simps last-conv-nth*)  
also have  $(\sum_{k < length\ cs - 1} cs\ !\ k * f\ (n + 1 - length\ cs + k)) =$   
 $(\sum_{k < length\ cs - 1} cs\ !\ k * lhr\ cs\ fs\ (n + 1 - length\ cs + k))$   
using *False 1* by (*intro sum.cong refl*) *simp*  
finally have  $f\ n = (\sum_{k < length\ cs - 1} cs\ !\ k * lhr\ cs\ fs\ (n + 1 - length\ cs + k)) / -last\ cs$   
using  $\langle last\ cs \neq 0 \rangle$  by (*simp add: field-simps eq-neg-iff-add-eq-0*)  
also from  $1(2-4)\ False$  have  $\dots = lhr\ cs\ fs\ n$  by (*subst lhr.simps*) *simp*  
finally show ?*thesis* .  
qed (*insert 1(2-5), simp add: lhr.simps*)  
qed

locale *linear-homogenous-recurrence* =

fixes  $f :: nat \Rightarrow 'a :: comm-semiring-0$  and  $cs\ fs :: 'a\ list$   
assumes *base*:  $n < length\ fs \implies f\ n = fs\ !\ n$   
assumes *cs-not-null* [*simp*]:  $cs \neq []$  and *last-cs* [*simp*]:  $last\ cs \neq 0$   
and *hd-cs* [*simp*]:  $hd\ cs \neq 0$  and *enough-base*:  $length\ fs + 1 \geq length\ cs$   
assumes *rec*:  $n \geq length\ fs - length\ cs \implies (\sum_{k < length\ cs} cs\ !\ k * f\ (n + k)) = 0$   
begin

lemma *lhr-fps-numerator-altdef*:

$lhr-fps-numerator\ (length\ fs + 1 - length\ cs)\ cs\ f =$   
 $lhr-fps-numerator\ (length\ fs + 1 - length\ cs)\ cs\ (!)\ fs$   
proof -  
define  $N$  where  $N = length\ cs - 1$   
define  $m$  where  $m = length\ fs + 1 - length\ cs$   
have  $lhr-fps-numerator\ m\ cs\ f =$

**Poly** (*map* ( $\lambda k. (\sum_{i \leq \min N k. cs ! (N - i) * f (k - i)}) [0..<N + m]$ )  
**by** (*simp add: lhr-fps-numerator-def Let-def N-def*)  
**also from** *enough-base* **have**  $N + m = \text{length } fs$   
**by** (*cases cs*) (*simp-all add: N-def m-def algebra-simps*)  
**also** {  
**fix**  $k$  **assume**  $k: k \in \{0..<\text{length } fs\}$   
**hence**  $f (k - i) = fs ! (k - i)$  **if**  $i \leq \min N k$  **for**  $i$   
**using** *enough-base that* **by** (*intro base*) (*auto simp: Suc-le-eq N-def m-def algebra-simps*)  
**hence**  $(\sum_{i \leq \min N k. cs ! (N - i) * f (k - i)}) = (\sum_{i \leq \min N k. cs ! (N - i) * fs ! (k - i)})$   
**by** *simp*  
**}**  
**hence**  $\text{map } (\lambda k. (\sum_{i \leq \min N k. cs ! (N - i) * f (k - i)}) [0..<\text{length } fs]) = \text{map } (\lambda k. (\sum_{i \leq \min N k. cs ! (N - i) * fs ! (k - i)}) [0..<\text{length } fs])$   
**by** (*intro map-cong*) *simp-all*  
**also have** *Poly ... = lhr-fps-numerator m cs (!) fs* **using** *enough-base*  
**by** (*cases cs*) (*simp-all add: lhr-fps-numerator-def Let-def m-def N-def*)  
**finally show** *?thesis unfolding m-def* .  
**qed**  
**end**

**lemma** *solve-lhr-aux:*

**assumes** *linear-homogenous-recurrence f cs fs*  
**assumes** *is-factorization-of fctrs (lr-fps-denominator' cs)*  
**shows**  $f = \text{interp-ratfps-solution } (\text{solve-factored-ratfps}' (lhr-fps-numerator (\text{length } fs + 1 - \text{length } cs) cs (!) fs)) fctrs$

**proof** –

**interpret** *linear-homogenous-recurrence f cs fs* **by** *fact*

**note** *assms(2)*

**hence** *is-alt-factorization-of fctrs (reflect-poly (lr-fps-denominator' cs))*

**by** (*intro reflect-factorization*)

*(simp-all add: lr-fps-denominator'-def*  
*nth-default-def hd-conv-nth [symmetric])*

**also have** *reflect-poly (lr-fps-denominator' cs) = lr-fps-denominator cs*

**unfolding** *lr-fps-denominator-def lr-fps-denominator'-def*

**by** (*subst coeffs-eq-iff*) (*simp add: coeffs-reflect-poly strip-while-rev [symmetric]*  
*no-trailing-unfold last-rev del: strip-while-rev*)

**finally have** *factorization: is-alt-factorization-of fctrs (lr-fps-denominator cs)* .

**define**  $m$  **where**  $m = \text{length } fs + 1 - \text{length } cs$

**obtain**  $a$   $ds$  **where** *fctrs: fctrs = (a, ds)* **by** (*cases fctrs*) *simp-all*

**define**  $p$  **and**  $p'$  **where**  $p = \text{lhr-fps-numerator } m \text{ cs } (!) \text{ fs}$  **and**  $p' = \text{smult } (\text{inverse } a) \text{ } p$

**obtain**  $b$   $es$  **where** *sol: solve-factored-ratfps' p fctrs = (b, es)*

**by** (*cases solve-factored-ratfps' p fctrs*) *simp-all*

```

have sol': (b, es) = solve-factored-ratfps p' ds
  by (subst sol [symmetric]) (simp add: fctrs p'-def solve-factored-ratfps-def
    solve-factored-ratfps'-def case-prod-unfold)
have factorization': lr-fps-denominator cs = interp-alt-factorization fctrs
  using factorization by (simp add: is-alt-factorization-of-def)
from assms(2) have distinct: distinct (map fst ds)
  by (simp add: fctrs is-factorization-of-def)
have coeff-0-denom: coeff (lr-fps-denominator cs) 0 ≠ 0
  by (simp add: lr-fps-denominator-def nth-default-def
    hd-conv-nth [symmetric] hd-rev)
have coeff (lr-fps-denominator' cs) 0 ≠ 0
  by (simp add: lr-fps-denominator'-def nth-default-def hd-conv-nth [symmetric])
with assms(2) have no-zero: 0 ∉ fst ' set ds by (simp add: zero-in-factorization-iff
  fctrs)

from assms(2) have a-nz [simp]: a ≠ 0
  by (auto simp: fctrs interp-factorization-def is-factorization-of-def lr-fps-denominator'-nz)
hence unit1: is-unit (fps-const a) by simp
moreover have is-unit (fps-of-poly (interp-alt-factorization fctrs))
  by (simp add: coeff-0-denom factorization' [symmetric])
ultimately have unit2: is-unit (fps-of-poly (∏ p←ds. [:1, - fst p:] ^ Suc (snd
  p)))
  by (simp add: fctrs case-prod-unfold interp-alt-factorization-def del: power-Suc)

have Abs-fps f = fps-of-poly (lhr-fps-numerator m cs f) /
  fps-of-poly (lr-fps-denominator cs)
proof (intro lhr-fps)
  fix n assume n: n ≥ m
  have {..length cs - 1} = {..<length cs} by (cases cs) auto
  also from n have (∑ k∈.. . cs ! k * f (n + k)) = 0
    by (intro rec) (simp-all add: m-def algebra-simps)
  finally show (∑ k≤length cs - 1. cs ! k * f (n + k)) = 0 .
qed (simp-all add: m-def)
also have lhr-fps-numerator m cs f = lhr-fps-numerator m cs (!) fs
  unfolding lhr-fps-numerator-def using enough-base
  by (auto simp: Let-def poly-eq-iff nth-default-def base
    m-def Suc-le-eq intro!: sum.cong)
also have fps-of-poly ... / fps-of-poly (lr-fps-denominator cs) =
  fps-of-poly (lhr-fps-numerator m cs (!) fs) /
  (fps-const (fst fctrs) *
    fps-of-poly (∏ p←snd fctrs. [:1, - fst p:] ^ Suc (snd p)))
  unfolding assms factorization' interp-alt-factorization-def
  by (simp add: case-prod-unfold Let-def fps-of-poly-smult)
also from unit1 unit2 have ... = fps-of-poly p / fps-const a /
  fps-of-poly (∏ (c,n)←ds. [:1, -c:] ^ Suc n)
  by (subst is-unit-div-mult2-eq) (simp-all add: fctrs case-prod-unfold p-def)
also from unit1 have fps-of-poly p / fps-const a = fps-of-poly p'
  by (simp add: fps-divide-unit fps-of-poly-smult fps-const-inverse p'-def)
also from distinct no-zero have ... / fps-of-poly (∏ (c,n)←ds. [:1, -c:] ^ Suc n)

```



```

=
  Abs-fps (interp-ratfps-solution (solve-factored-ratfps' p fctrs))
  by (subst solve-factored-ratfps) (simp-all add: case-prod-unfold sol' sol)
  finally show ?thesis unfolding p-def m-def
  by (intro ext) (simp add: fps-eq-iff)
qed

```

**definition**

```

lhr-fps as fs = (
  let m = length fs + 1 - length as;
      p = lhr-fps-numerator m as (λn. fs ! n);
      q = lr-fps-denominator as
  in ratfps-of-poly p / ratfps-of-poly q)

```

**lemma** *lhr-fps-correct*:

```

fixes f :: nat ⇒ 'a :: {field-char-0,field-gcd}
assumes linear-homogenous-recurrence f cs fs
shows fps-of-ratfps (lhr-fps cs fs) = Abs-fps f
proof -
  interpret linear-homogenous-recurrence f cs fs by fact
  define m where m = length fs + 1 - length cs
  let ?num = lhr-fps-numerator m cs f
  let ?num' = lhr-fps-numerator m cs (!) fs
  let ?denom = lr-fps-denominator cs

  have {..length cs - 1} = {..<length cs} by (cases cs) auto
  moreover have length cs ≥ 1 by (cases cs) auto
  ultimately have Abs-fps f = fps-of-poly ?num / fps-of-poly ?denom
    by (intro lhr-fps) (insert rec, simp-all add: m-def)
  also have ?num = ?num'
    by (rule lhr-fps-numerator-altdef [folded m-def])
  also have fps-of-poly ?num' / fps-of-poly ?denom =
    fps-of-ratfps (ratfps-of-poly ?num' / ratfps-of-poly ?denom)
    by simp
  also from enough-base have ... = fps-of-ratfps (lhr-fps cs fs)
    by (cases cs) (simp-all add: base fps-of-ratfps-def case-prod-unfold lhr-fps-def
m-def)
  finally show ?thesis ..
qed

```

**end**

## 9 Eulerian polynomials

**theory** *Eulerian-Polynomials*

**imports**

```

  Complex-Main
  HOL-Combinatorics.Stirling
  HOL-Computational-Algebra.Computational-Algebra

```

**begin**

The Eulerian polynomials are a sequence of polynomials that is related to the closed forms of the power series

$$\sum_{n=0}^{\infty} n^k X^n$$

for a fixed  $k$ .

**primrec** *eulerian-poly* :: *nat*  $\Rightarrow$  'a :: *idom poly* **where**  
*eulerian-poly* 0 = 1  
| *eulerian-poly* (Suc n) = (let p = *eulerian-poly* n in  
[:0,1,-1:] \* *pderiv* p + p \* [:1, of-nat n:])

**lemmas** *eulerian-poly-Suc* [*simp del*] = *eulerian-poly.simps*(2)

**lemma** *eulerian-poly*:

*fps-of-poly* (*eulerian-poly* k :: 'a :: *field poly*) =  
*Abs-fps* ( $\lambda n. \text{of-nat } (n+1) \wedge k$ ) \* (1 - *fps-X*)  $\wedge$  (k + 1)

**proof** (*induction* k)

**case** 0

**have** *Abs-fps* ( $\lambda-. 1 :: 'a$ ) = *inverse* (1 - *fps-X*)

**by** (*rule fps-inverse-unique* [*symmetric*])

(*simp add: inverse-mult-eq-1 fps-inverse-gp'* [*symmetric*])

**thus** ?*case* **by** (*simp add: inverse-mult-eq-1*)

**next**

**case** (Suc k)

**define** p :: 'a *fps* **where** p = *fps-of-poly* (*eulerian-poly* k)

**define** F :: 'a *fps* **where** F = *Abs-fps* ( $\lambda n. \text{of-nat } (n+1) \wedge k$ )

**have** p: p = F \* (1 - *fps-X*)  $\wedge$  (k+1) **by** (*simp add: p-def Suc F-def*)

**have** p': *fps-deriv* p = *fps-deriv* F \* (1 - *fps-X*)  $\wedge$  (k + 1) - F \* (1 - *fps-X*)  
 $\wedge$  k \* *of-nat* (k + 1)

**by** (*simp add: p fps-deriv-power algebra-simps fps-const-neg* [*symmetric*] *fps-of-nat*

*del: power-Suc of-nat-Suc fps-const-neg*)

**have** *fps-of-poly* (*eulerian-poly* (Suc k)) = (*fps-X* \* *fps-deriv* F + F) \* (1 -  
*fps-X*)  $\wedge$  (Suc k + 1)

**apply** (*simp add: Let-def p-def* [*symmetric*] *fps-of-poly-simps eulerian-poly-Suc*  
*del: power-Suc*)

**apply** (*simp add: p p' fps-deriv-power fps-const-neg* [*symmetric*] *fps-of-nat*  
*del: power-Suc of-nat-Suc fps-const-neg*)

**apply** (*simp add: algebra-simps*)

**done**

**also have** *fps-X* \* *fps-deriv* F + F = *Abs-fps* ( $\lambda n. \text{of-nat } (n + 1) \wedge \text{Suc } k$ )

**unfolding** F-def **by** (*intro fps-ext*) (*auto simp: algebra-simps*)

**finally show** ?*case* .

**qed**

**lemma** *eulerian-poly'*:

*Abs-fps* ( $\lambda n. \text{of-nat } (n+1) \wedge k$ ) =  
*fps-of-poly* (*eulerian-poly*  $k :: 'a :: \text{field poly}$ ) /  $(1 - \text{fps-X}) \wedge (k + 1)$   
**by** (*subst eulerian-poly simp*)

**lemma** *eulerian-poly''*:

**assumes**  $k: k > 0$

**shows** *Abs-fps* ( $\lambda n. \text{of-nat } n \wedge k$ ) =

*fps-of-poly* (*pCons* 0 (*eulerian-poly*  $k :: 'a :: \text{field poly}$ )) /  $(1 - \text{fps-X}) \wedge (k + 1)$

**proof** –

**from** *assms* **have** *Abs-fps* ( $\lambda n. \text{of-nat } n \wedge k :: 'a$ ) = *fps-X* \* *Abs-fps* ( $\lambda n. \text{of-nat } (n + 1) \wedge k$ )

**by** (*intro fps-ext*) (*auto simp: of-nat-diff*)

**also have** *Abs-fps* ( $\lambda n. \text{of-nat } (n + 1) \wedge k :: 'a$ ) =

*fps-of-poly* (*eulerian-poly*  $k$ ) /  $(1 - \text{fps-X}) \wedge (k + 1)$  **by** (*rule eulerian-poly'*)

**also have** *fps-X* \* ... = *fps-of-poly* (*pCons* 0 (*eulerian-poly*  $k$ )) /  $(1 - \text{fps-X}) \wedge (k + 1)$

**by** (*simp add: fps-of-poly-pCons fps-divide-unit*)

**finally show** *?thesis* .

**qed**

**definition** *fps-monom-poly*  $:: 'a :: \text{field} \Rightarrow \text{nat} \Rightarrow 'a \text{ poly}$

**where** *fps-monom-poly*  $c k = (\text{if } k = 0 \text{ then } 1 \text{ else } \text{pcompose } (\text{pCons } 0 (\text{eulerian-poly } k)) \text{ [:0,c:]})$

**primrec** *fps-monom-poly-aux*  $:: 'a :: \text{field} \Rightarrow \text{nat} \Rightarrow 'a \text{ poly}$  **where**

*fps-monom-poly-aux*  $c 0 = \text{[:c:]}$

| *fps-monom-poly-aux*  $c (\text{Suc } k) =$

(*let*  $p = \text{fps-monom-poly-aux } c k$

*in*  $\text{[:0,1,-c:]} * \text{pderiv } p + \text{[:1, of-nat } k * \text{c:]} * p$ )

**lemma** *fps-monom-poly-aux*:

*fps-monom-poly-aux*  $c k = \text{smult } c (\text{pcompose } (\text{eulerian-poly } k) \text{ [:0,c:]})$

**by** (*induction k*)

(*simp-all add: eulerian-poly-Suc Let-def pderiv-pcompose pcompose-pCons*

*pcompose-add pcompose-smult pcompose-uminus smult-add-right*

*pderiv-pCons*

*pderiv-smult algebra-simps one-pCons*)

**lemma** *fps-monom-poly-code* [*code*]:

*fps-monom-poly*  $c k = (\text{if } k = 0 \text{ then } 1 \text{ else } \text{pCons } 0 (\text{fps-monom-poly-aux } c k))$

**by** (*simp add: fps-monom-poly-def fps-monom-poly-aux pcompose-pCons*)

**lemma** *fps-monom-aux*:

*Abs-fps* ( $\lambda n. \text{of-nat } n \wedge k$ ) = *fps-of-poly* (*fps-monom-poly* 1  $k$ ) /  $(1 - \text{fps-X}) \wedge (k+1)$

**proof** (*cases*  $k = 0$ )  
**assume** [*simp*]:  $k = 0$   
**hence**  $Abs-fps (\lambda n. of-nat n \wedge k :: 'a) = Abs-fps (\lambda-. 1)$  **by** *simp*  
**also have**  $\dots = 1 / (1 - fps-X)$  **by** (*subst gp [symmetric]*) *simp-all*  
**finally show** *?thesis* **by** (*simp add: fps-monom-poly-def*)  
**qed** (*insert eulerian-poly'[of k, where ?'a = 'a], simp add: fps-monom-poly-def*)

**lemma** *fps-monom*:

$Abs-fps (\lambda n. of-nat n \wedge k * c \wedge n) =$   
 $fps-of-poly (fps-monom-poly c k) / (1 - fps-const c * fps-X) \wedge (k+1)$

**proof** –

**have**  $Abs-fps (\lambda n. of-nat n \wedge k * c \wedge n) =$   
 $fps-compose (Abs-fps (\lambda n. of-nat n \wedge k)) (fps-const c * fps-X)$   
**by** (*subst fps-compose-linear*) (*simp add: mult-ac*)  
**also have**  $Abs-fps (\lambda n. of-nat n \wedge k) = fps-of-poly (fps-monom-poly 1 k) / (1 -$   
 $fps-X) \wedge (k+1)$   
**by** (*rule fps-monom-aux*)  
**also have**  $fps-compose \dots (fps-const c * fps-X) =$   
 $(fps-of-poly (fps-monom-poly 1 k) oo fps-const c * fps-X) /$   
 $((1 - fps-X) \wedge (k + 1) oo fps-const c * fps-X)$   
**by** (*intro fps-compose-divide-distrib*)  
*(simp-all add: fps-compose-power [symmetric] fps-compose-sub-distrib del:*  
*power-Suc)*  
**also have**  $fps-of-poly (fps-monom-poly 1 k) oo (fps-const c * fps-X) =$   
 $fps-of-poly (fps-monom-poly c k)$   
**by** (*simp add: fps-monom-poly-def fps-of-poly-pcompose fps-of-poly-simps*  
 $fps-of-poly-pCons mult-ac$ )  
**also have**  $((1 - fps-X) \wedge (k + 1) oo fps-const c * fps-X) = (1 - fps-const c *$   
 $fps-X) \wedge (k + 1)$   
**by** (*simp add: fps-compose-power [symmetric] fps-compose-sub-distrib del: power-Suc*)  
**finally show** *?thesis* .

**qed**

**end**

## 10 Inhomogenous linear recurrences

**theory** *Linear-Inhomogenous-Recurrences*

**imports**

*Complex-Main*

*Linear-Homogenous-Recurrences*

*Eulerian-Polynomials*

*RatFPS*

**begin**

**definition** *lir-fps-numerator* **where**

*lir-fps-numerator m cs f g = (let N = length cs - 1 in*

*Poly [( $\sum_{i \leq \min N k. cs ! (N - i) * f (k - i) - g k. k \leftarrow [0..<N+m]$ )]*)

**lemma** *lir-fps-numerator-code* [code abstract]:  
 $\text{coeffs } (\text{lir-fps-numerator } m \text{ cs } f \text{ g}) = (\text{let } N = \text{length } cs - 1 \text{ in}$   
 $\text{strip-while } ((=) 0) [(\sum i \leq \min N \text{ k. } cs ! (N - i) * f (k - i)) - g \text{ k. } k \leftarrow$   
 $[0..<N+m]])$   
**by** (*simp add: lir-fps-numerator-def Let-def*)

**locale** *linear-inhomogenous-recurrence* =  
**fixes**  $f \text{ g} :: \text{nat} \Rightarrow 'a :: \text{comm-ring}$  **and**  $cs \text{ fs} :: 'a \text{ list}$   
**assumes** *base*:  $n < \text{length } fs \implies f \text{ n} = fs ! n$   
**assumes** *cs-not-null* [simp]:  $cs \neq []$  **and** *last-cs* [simp]:  $\text{last } cs \neq 0$   
**and** *hd-cs* [simp]:  $\text{hd } cs \neq 0$  **and** *enough-base*:  $\text{length } fs + 1 \geq \text{length } cs$   
**assumes** *rec*:  $n \geq \text{length } fs + 1 - \text{length } cs \implies$   
 $(\sum k < \text{length } cs. cs ! k * f (n + k)) = g (n + \text{length } cs - 1)$   
**begin**

**lemma** *coeff-0-lr-fps-denominator* [simp]:  $\text{coeff } (\text{lr-fps-denominator } cs) 0 = \text{last}$   
 $cs$   
**by** (*auto simp: lr-fps-denominator-def nth-default-def nth-Cons hd-conv-nth [symmetric]*  
*hd-rev*)

**lemma** *lir-fps-numerator-altdef*:  
 $\text{lir-fps-numerator } (\text{length } fs + 1 - \text{length } cs) \text{ cs } f \text{ g} =$   
 $\text{lir-fps-numerator } (\text{length } fs + 1 - \text{length } cs) \text{ cs } (!) \text{ fs } g$   
**proof** –  
**define**  $N$  **where**  $N = \text{length } cs - 1$   
**define**  $m$  **where**  $m = \text{length } fs + 1 - \text{length } cs$   
**have** *lir-fps-numerator*  $m \text{ cs } f \text{ g} =$   
 $\text{Poly } (\text{map } (\lambda k. (\sum i \leq \min N \text{ k. } cs ! (N - i) * f (k - i)) - g \text{ k}) [0..<N$   
 $+ m])$   
**by** (*simp add: lir-fps-numerator-def Let-def N-def*)  
**also from** *enough-base* **have**  $N + m = \text{length } fs$   
**by** (*cases cs*) (*simp-all add: N-def m-def algebra-simps*)  
**also** {  
**fix**  $k$  **assume**  $k: k \in \{0..<\text{length } fs\}$   
**hence**  $f (k - i) = fs ! (k - i)$  **if**  $i \leq \min N \text{ k}$  **for**  $i$   
**using** *enough-base that* **by** (*intro base*) (*auto simp: Suc-le-eq N-def m-def*  
*algebra-simps*)  
**hence**  $(\sum i \leq \min N \text{ k. } cs ! (N - i) * f (k - i)) = (\sum i \leq \min N \text{ k. } cs ! (N -$   
 $i) * fs ! (k - i))$   
**by** *simp*  
**}**  
**hence**  $\text{map } (\lambda k. (\sum i \leq \min N \text{ k. } cs ! (N - i) * f (k - i)) - g \text{ k}) [0..<\text{length } fs]$   
 $=$   
 $\text{map } (\lambda k. (\sum i \leq \min N \text{ k. } cs ! (N - i) * fs ! (k - i)) - g \text{ k}) [0..<\text{length}$   
 $fs]$   
**by** (*intro map-cong*) *simp-all*  
**also have**  $\text{Poly } \dots = \text{lir-fps-numerator } m \text{ cs } (!) \text{ fs } g$  **using** *enough-base*  
**by** (*cases cs*) (*simp-all add: lir-fps-numerator-def Let-def m-def N-def*)

```

    finally show ?thesis unfolding m-def .
qed

end

context
begin

private lemma lir-fps-aux:
  fixes f :: nat => 'a :: field
  assumes rec:  $\bigwedge n. n \geq m \implies (\sum k \leq N. c k * f (n + k)) = g (n + N)$ 
  assumes cN:  $c N \neq 0$ 
  defines p  $\equiv$  Poly [c (N - k). k  $\leftarrow$  [0.. $\text{Suc } N$ ]]
  defines q  $\equiv$  Poly [( $\sum i \leq \min N k. c (N - i) * f (k - i)$ ) - g k. k  $\leftarrow$  [0.. $N+m$ ]]
  shows Abs-fps f = (fps-of-poly q + Abs-fps g) / fps-of-poly p
proof -
  include fps-notation
  define F where F = Abs-fps f
  have [simp]: F $ n = f n for n by (simp add: F-def)
  have [simp]: coeff p 0 = c N
    by (simp add: p-def nth-default-def del: upt-Suc)

  have (fps-of-poly p * F) $ n = coeff q n + g n for n
  proof (cases n  $\geq$  N + m)
  case True
    let ?f =  $\lambda i. N - i$ 
    have (fps-of-poly p * F) $ n = ( $\sum i \leq n. \text{coeff } p i * f (n - i)$ )
      by (simp add: fps-mult-nth atLeast0AtMost)
    also from True have ... = ( $\sum i \leq N. \text{coeff } p i * f (n - i)$ )
      by (intro sum.mono-neutral-right) (auto simp: nth-default-def p-def)
    also have ... = ( $\sum i \leq N. c (N - i) * f (n - i)$ )
      by (intro sum.cong) (auto simp: nth-default-def p-def simp del: upt-Suc)
    also from True have ... = ( $\sum i \leq N. c i * f (n - N + i)$ )
      by (intro sum.reindex-bij-witness[of - ?f ?f]) auto
    also from True have ... = g (n - N + N) by (intro rec) simp-all
    also from True have ... = coeff q n + g n
      by (simp add: q-def nth-default-def del: upt-Suc)
    finally show ?thesis .
  case False
  hence (fps-of-poly p * F) $ n = ( $\sum i \leq n. \text{coeff } p i * f (n - i)$ )
    by (simp add: fps-mult-nth atLeast0AtMost)
  also have ... = ( $\sum i \leq \min N n. \text{coeff } p i * f (n - i)$ )
    by (intro sum.mono-neutral-right)
    (auto simp: p-def nth-default-def simp del: upt-Suc)
  also have ... = ( $\sum i \leq \min N n. c (N - i) * f (n - i)$ )
    by (intro sum.cong) (simp-all add: p-def nth-default-def del: upt-Suc)
  also from False have ... = coeff q n + g n by (simp add: q-def nth-default-def)
  end
end

```

**finally show** *?thesis* .  
**qed**  
**hence**  $\text{fps-of-poly } p * F = \text{fps-of-poly } q + \text{Abs-fps } g$   
**by** (*intro fps-ext*) (*simp add:*)  
**with**  $cN$  **show**  $F = (\text{fps-of-poly } q + \text{Abs-fps } g) / \text{fps-of-poly } p$   
**by** (*subst unit-eq-div2*) (*simp-all add: mult-ac*)  
**qed**

**lemma** *lir-fps*:  
**fixes**  $f g :: \text{nat} \Rightarrow 'a :: \text{field}$  **and**  $cs :: 'a \text{ list}$   
**defines**  $N \equiv \text{length } cs - 1$   
**assumes**  $cs: cs \neq []$   
**assumes**  $\bigwedge n. n \geq m \implies (\sum k \leq N. cs ! k * f (n + k)) = g (n + N)$   
**assumes**  $cN: \text{last } cs \neq 0$   
**shows**  $\text{Abs-fps } f = (\text{fps-of-poly } (\text{lir-fps-numerator } m \text{ } cs \text{ } f \text{ } g) + \text{Abs-fps } g) /$   
 $\text{fps-of-poly } (\text{lr-fps-denominator } cs)$

**proof** –  
**define**  $p$  **and**  $q$   
**where**  $p = \text{Poly } [(\sum i \leq \min N \text{ } k. cs ! (N - i) * f (k - i)) - g \text{ } k. k \leftarrow$   
 $[0..<N+m]]$   
**and**  $q = \text{Poly } (\text{map } (\lambda k. cs ! (N - k)) [0..< \text{Suc } N])$   
**from** *assms* **have**  $\text{Abs-fps } f = (\text{fps-of-poly } p + \text{Abs-fps } g) / \text{fps-of-poly } q$   
**unfolding**  $p\text{-def } q\text{-def}$  **by** (*intro lir-fps-aux*) (*simp-all add: last-conv-nth*)  
**also have**  $p = \text{lir-fps-numerator } m \text{ } cs \text{ } f \text{ } g$   
**unfolding**  $p\text{-def } \text{lir-fps-numerator-def}$  **by** (*auto simp: Let-def N-def*)  
**also from**  $cN$  **have**  $q = \text{lr-fps-denominator } cs$   
**unfolding**  $q\text{-def } \text{lr-fps-denominator-def}$   
**by** (*intro poly-eqI*)  
*(auto simp add: nth-default-def rev-nth N-def not-less cs simp del: upt-Suc)*  
**finally show** *?thesis* .

**qed**

**end**

**type-synonym**  $'a \text{ polyexp} = ('a \times \text{nat} \times 'a) \text{ list}$

**definition**  $\text{eval-polyexp} :: ('a :: \text{semiring-1}) \text{ polyexp} \Rightarrow \text{nat} \Rightarrow 'a$  **where**  
 $\text{eval-polyexp } xs = (\lambda n. \sum (a,k,b) \leftarrow xs. a * \text{of-nat } n \wedge k * b \wedge n)$

**lemma**  $\text{eval-polyexp-Nil}$  [*simp*]:  $\text{eval-polyexp } [] = (\lambda -. 0)$   
**by** (*simp add: eval-polyexp-def*)

**lemma**  $\text{eval-polyexp-Cons}$ :

$\text{eval-polyexp } (x \# xs) = (\lambda n. (\text{case } x \text{ of } (a,k,b) \Rightarrow a * \text{of-nat } n \wedge k * b \wedge n) +$   
 $\text{eval-polyexp } xs \text{ } n)$   
**by** (*simp add: eval-polyexp-def*)

**definition** *polyexp-fps* :: ('a :: field) *polyexp*  $\Rightarrow$  'a *fps* **where**

$$\begin{aligned} \text{polyexp-fps } xs = \\ (\sum (a,k,b) \leftarrow xs. \text{fps-of-poly } (\text{Polynomial.smult } a \text{ (fps-monom-poly } b \text{ } k)) / \\ (1 - \text{fps-const } b * \text{fps-X}) \wedge (k + 1)) \end{aligned}$$

**lemma** *polyexp-fps-Nil* [*simp*]: *polyexp-fps* [] = 0  
**by** (*simp add: polyexp-fps-def*)

**lemma** *polyexp-fps-Cons*:

$$\begin{aligned} \text{polyexp-fps } (x\#xs) = (\text{case } x \text{ of } (a,k,b) \Rightarrow \\ \text{fps-of-poly } (\text{Polynomial.smult } a \text{ (fps-monom-poly } b \text{ } k)) / (1 - \text{fps-const } b * \\ \text{fps-X}) \wedge (k + 1)) + \\ \text{polyexp-fps } xs \\ \text{by } (\text{simp add: polyexp-fps-def}) \end{aligned}$$

**definition** *polyexp-ratfps* :: ('a :: field-gcd) *polyexp*  $\Rightarrow$  'a *ratfps* **where**

$$\begin{aligned} \text{polyexp-ratfps } xs = \\ (\sum (a,k,b) \leftarrow xs. \text{ratfps-of-poly } (\text{Polynomial.smult } a \text{ (fps-monom-poly } b \text{ } k)) / \\ \text{ratfps-of-poly } ([:1, -b:] \wedge (k + 1))) \end{aligned}$$

**lemma** *polyexp-ratfps-Nil* [*simp*]: *polyexp-ratfps* [] = 0  
**by** (*simp add: polyexp-ratfps-def*)

**lemma** *polyexp-ratfps-Cons*: *polyexp-ratfps* (x#xs) = (case x of (a,k,b)  $\Rightarrow$   
*ratfps-of-poly* (Polynomial.smult a (fps-monom-poly b k)) /  
*ratfps-of-poly* ([:1, -b:]  $\wedge$  (k + 1))) + *polyexp-ratfps* xs  
**by** (*simp add: polyexp-ratfps-def*)

**lemma** *polyexp-fps: Abs-fps* (*eval-polyexp* xs) = *polyexp-fps* xs

**proof** (*induction xs*)

**case** (*Cons x xs*)

**obtain** a k b **where** [*simp*]: x = (a, k, b) **by** (*metis prod.exhaust*)

**have** *Abs-fps* (*eval-polyexp* (x#xs)) =

$$\text{fps-const } a * \text{Abs-fps } (\lambda n. \text{of-nat } n \wedge k * b \wedge n) + \text{Abs-fps } (\text{eval-polyexp } xs)$$

**by** (*simp add: eval-polyexp-Cons fps-plus-def mult-ac*)

**also have** *Abs-fps* ( $\lambda n. \text{of-nat } n \wedge k * b \wedge n$ ) =

$$\text{fps-of-poly } (\text{fps-monom-poly } b \text{ } k) / (1 - \text{fps-const } b * \text{fps-X}) \wedge (k +$$

1)

$$(\text{is } - = ?A / ?B)$$

**by** (*rule fps-monom*)

**also have** *fps-const* a \* (?A / ?B) = (*fps-const* a \* ?A) / ?B

**by** (*intro unit-div-mult-swap simp-all*)

**also have** *fps-const* a \* ?A = *fps-of-poly* (Polynomial.smult a (fps-monom-poly b k))

**by** *simp*

**also note** *Cons.IH*

**finally show** ?case **by** (*simp add: polyexp-fps-Cons*)

**qed** (*simp-all add: fps-zero-def*)



**lemma** *polyexp-ratfps* [*simp*]:  $\text{fps-of-ratfps } (\text{polyexp-ratfps } xs) = \text{polyexp-fps } xs$   
**by** (*induction xs*)  
*(auto simp del: power-Suc fps-const-neg*  
*simp: coeff-0-power fps-of-poly-power fps-of-poly-smult fps-of-poly-pCons*  
*fps-const-neg [symmetric] mult-ac polyexp-ratfps-Cons polyexp-fps-Cons)*

**definition** *lir-fps* ::

*'a* :: *field-gcd list*  $\Rightarrow$  *'a list*  $\Rightarrow$  *'a polyexp*  $\Rightarrow$  (*'a ratfps*) **option** **where**  
*lir-fps cs fs g* = (*if cs* = []  $\vee$  *length fs* < *length cs* - 1 **then** *None* **else**  
*let m* = *length fs* + 1 - *length cs*;  
*p* = *lir-fps-numerator m cs* ( $\lambda n. fs ! n$ ) (*eval-polyexp g*);  
*q* = *lr-fps-denominator cs*  
**in** *Some* ((*ratfps-of-poly p* + *polyexp-ratfps g*) \* *inverse* (*ratfps-of-poly q*)))

**lemma** *lir-fps-correct*:

**fixes** *f* :: *nat*  $\Rightarrow$  *'a* :: *field-gcd*  
**assumes** *linear-inhomogenous-recurrence f* (*eval-polyexp g*) *cs fs*  
**shows** *map-option fps-of-ratfps* (*lir-fps cs fs g*) = *Some* (*Abs-fps f*)

**proof** -

**interpret** *linear-inhomogenous-recurrence f eval-polyexp g cs fs* **by fact**  
**define** *m* **where** *m* = *length fs* + 1 - *length cs*  
**let** *?num* = *lir-fps-numerator m cs f* (*eval-polyexp g*)  
**let** *?num'* = *lir-fps-numerator m cs* (!) *fs* (*eval-polyexp g*)  
**let** *?denom* = *lr-fps-denominator cs*

**have** {..*length cs* - 1} = {..*length cs*} **by** (*cases cs*) *auto*  
**moreover have** *length cs*  $\geq$  1 **by** (*cases cs*) *auto*  
**ultimately have** *Abs-fps f* = (*fps-of-poly ?num* + *Abs-fps* (*eval-polyexp g*)) /  
*fps-of-poly ?denom*

**by** (*intro lir-fps*) (*insert rec, simp-all add: m-def*)  
**also have** *?num* = *?num'* **by** (*rule lir-fps-numerator-altdef [folded m-def]*)  
**also have** (*fps-of-poly ?num'* + *Abs-fps* (*eval-polyexp g*)) / *fps-of-poly ?denom* =

$$\text{fps-of-ratfps } ((\text{ratfps-of-poly } ?num' + \text{polyexp-ratfps } g) * \text{inverse } (\text{ratfps-of-poly } ?denom))$$

**by** (*simp add: polyexp-fps fps-divide-unit*)  
**also from** *enough-base* **have** *Some ...* = *map-option fps-of-ratfps* (*lir-fps cs fs*  
*g*)  
**by** (*cases cs*) (*simp-all add: base fps-of-ratfps-def case-prod-unfold lir-fps-def*  
*m-def*)

**finally show** *?thesis ..*  
**qed**

**end**

**theory** *Rational-FPS-Asymptotics*

**imports**

*HOL-Library.Landau-Symbols*

*Polynomial-Factorization.Square-Free-Factorization*  
*HOL-Real-Asymp.Real-Asymp*  
*Count-Complex-Roots.Count-Complex-Roots*  
*Linear-Homogenous-Recurrences*  
*Linear-Inhomogenous-Recurrences*  
*RatFPS*  
*Rational-FPS-Solver*  
*HOL-Library.Code-Target-Numeral*

**begin**

**lemma** *poly-asymp-equiv*:

**assumes**  $p \neq 0$  **and**  $F \leq \text{at-infinity}$

**shows**  $\text{poly } p \sim[F] (\lambda x. \text{lead-coeff } p * x \wedge \text{degree } p)$

**proof** –

**have** *poly-pCons'*:  $\text{poly } (p\text{Cons } a \ q) = (\lambda x. a + x * \text{poly } q \ x)$  **for**  $a :: 'a$  **and**  $q$

**by** (*simp add: fun-eq-iff*)

**show** *?thesis using assms(1)*

**proof** (*induction p*)

**case** (*pCons a p*)

**define**  $n$  **where**  $n = \text{Suc } (\text{degree } p)$

**show** *?case*

**proof** (*cases p = 0*)

**case** [*simp*]: *False*

**hence**  $*$ :  $\text{poly } p \sim[F] (\lambda x. \text{lead-coeff } p * x \wedge \text{degree } p)$

**by** (*intro pCons.IH*)

**have**  $\text{poly } (p\text{Cons } a \ p) = (\lambda x. a + x * \text{poly } p \ x)$

**by** (*simp add: poly-pCons'*)

**moreover** **have**  $\dots \sim[F] (\lambda x. \text{lead-coeff } p * x \wedge n)$

**proof** (*subst asymp-equiv-add-left*)

**have**  $(\lambda x. x * \text{poly } p \ x) \sim[F] (\lambda x. x * (\text{lead-coeff } p * x \wedge \text{degree } p))$

**by** (*intro asymp-equiv-intros \**)

**also** **have**  $\dots = (\lambda x. \text{lead-coeff } p * x \wedge n)$  **by** (*simp add: n-def mult-ac*)

**finally** **show**  $(\lambda x. x * \text{poly } p \ x) \sim[F] \dots$  .

**next**

**have** *filterlim*  $(\lambda x. x)$  *at-infinity*  $F$

**by** (*simp add: filterlim-def assms*)

**hence**  $(\lambda x. x \wedge n) \in \omega[F](\lambda \cdot. 1 :: 'a)$  **unfolding** *smallomega-1-conv-filterlim*

**by** (*intro Limits.filterlim-power-at-infinity filterlim-ident*) (*auto simp: n-def*)

**hence**  $(\lambda x. a) \in o[F](\lambda x. x \wedge n)$  **unfolding** *smallomega-iff-smallo[symmetric]*

**by** (*cases a = 0*) *auto*

**thus**  $(\lambda x. a) \in o[F](\lambda x. \text{lead-coeff } p * x \wedge n)$

**by** *simp*

**qed**

**ultimately** **show** *?thesis* **by** (*simp add: n-def*)

**qed** *auto*

**qed** *auto*

**qed**

**lemma** *poly-bigtheta*:  
 assumes  $p \neq 0$  and  $F \leq \text{at-infinity}$   
 shows  $\text{poly } p \in \Theta[F](\lambda x. x^{\text{degree } p})$   
**proof** –  
 have  $\text{poly } p \sim[F] (\lambda x. \text{lead-coeff } p * x^{\text{degree } p})$   
 by (*intro poly-asymp-equiv assms*)  
 thus *?thesis* using *assms* by (*auto dest!: asymp-equiv-imp-bigtheta*)  
**qed**

**lemma** *poly-big*:  
 assumes  $F \leq \text{at-infinity}$  and  $\text{degree } p \leq k$   
 shows  $\text{poly } p \in O[F](\lambda x. x^k)$   
**proof** (*cases p = 0*)  
 case *True*  
 hence  $\text{poly } p = (\lambda-. 0)$  by (*auto simp: fun-eq-iff*)  
 thus *?thesis* by *simp*  
**next**  
 case *False*  
 have  $*$ :  $(\lambda x. x^{(k - \text{degree } p)}) \in \Omega[F](\lambda x. 1)$   
**proof** (*cases k = degree p*)  
 case *False*  
 hence  $(\lambda x. x^{(k - \text{degree } p)}) \in \omega[F](\lambda-. 1)$   
 unfolding *smallomega-1-conv-filterlim* using *assms False*  
 by (*intro Limits.filterlim-power-at-infinity filterlim-ident*)  
 (*auto simp: filterlim-def*)  
 thus *?thesis* by (*rule landau-omega.small-imp-big*)  
**qed** *auto*

have  $\text{poly } p \in \Theta[F](\lambda x. x^{\text{degree } p * 1})$   
 using *poly-bigtheta[OF False assms(1)]* by *simp*  
 also have  $(\lambda x. x^{\text{degree } p * 1}) \in O[F](\lambda x. x^{\text{degree } p * x^{(k - \text{degree } p)})$   
**using**  $*$   
 by (*intro landau-o.big.mult landau-o.big-refl*) (*auto simp: bigomega-iff-big*)  
 also have  $(\lambda x. x^{\text{degree } p * x^{(k - \text{degree } p)}) = (\lambda x. x^k)$   
 using *assms* by (*simp add: power-add [symmetric]*)  
 finally show *?thesis* .  
**qed**

**lemma** *reflect-poly-dvdI*:  
 fixes  $p q :: 'a::\{\text{comm-semiring-1, semiring-no-zero-divisors}\}$  *poly*  
 assumes  $p \text{ dvd } q$   
 shows *reflect-poly p dvd reflect-poly q*  
 using *assms* by (*auto simp: reflect-poly-mult*)

**lemma** *smult-altdef*:  $\text{smult } c p = [:c:] * p$   
 by (*induction p*) (*auto simp: mult-ac*)

**lemma** *smult-power*:  $\text{smult } (c^n) (p^n) = (\text{smult } c p)^n$   
**proof** –

**have**  $\text{smult } (c \wedge n) (p \wedge n) = [:c \wedge n:] * p \wedge n$   
**by** *simp*  
**also have**  $[:c:] \wedge n = [:c \wedge n:]$   
**by** (*induction n*) (*auto simp: mult-ac*)  
**hence**  $[:c \wedge n:] = [:c:] \wedge n ..$   
**also have**  $\dots * p \wedge n = ([:c:] * p) \wedge n$   
**by** (*rule power-mult-distrib [symmetric]*)  
**also have**  $\dots = (\text{smult } c \ p) \wedge n$  **by** *simp*  
**finally show** *?thesis* .  
**qed**

**lemma** *order-reflect-poly-ge:*

**fixes**  $c :: 'a :: \text{field}$   
**assumes**  $c \neq 0$  **and**  $p \neq 0$   
**shows**  $\text{order } c \ (\text{reflect-poly } p) \geq \text{order } (1 / c) \ p$   
**proof** –

**have**  $\text{reflect-poly } ([:-(1 / c), 1:] \wedge \text{order } (1 / c) \ p) \ \text{dvd} \ \text{reflect-poly } p$   
**by** (*intro reflect-poly-dvdI, subst order-divides*) *auto*  
**also have**  $\text{reflect-poly } ([:-(1 / c), 1:] \wedge \text{order } (1 / c) \ p) =$   
 $\text{smult } ((-1 / c) \wedge \text{order } (1 / c) \ p) ([: -c, 1:] \wedge \text{order } (1 / c) \ p)$   
**using** *assms* **by** (*simp add: reflect-poly-power reflect-poly-pCons smult-power*)  
**finally have**  $([: -c, 1:] \wedge \text{order } (1 / c) \ p) \ \text{dvd} \ \text{reflect-poly } p$   
**by** (*rule smult-dvd-cancel*)  
**with**  $\langle p \neq 0 \rangle$  **show** *?thesis* **by** (*subst (asm) order-divides*) *auto*  
**qed**

**lemma** *order-reflect-poly:*

**fixes**  $c :: 'a :: \text{field}$   
**assumes**  $c \neq 0$  **and**  $\text{coeff } p \ 0 \neq 0$   
**shows**  $\text{order } c \ (\text{reflect-poly } p) = \text{order } (1 / c) \ p$   
**proof** (*rule antisym*)  
**from** *assms* **show**  $\text{order } c \ (\text{reflect-poly } p) \geq \text{order } (1 / c) \ p$   
**by** (*intro order-reflect-poly-ge*) *auto*

**next**

**from** *assms* **have**  $\text{order } (1 / (1 / c)) \ (\text{reflect-poly } p) \leq$   
 $\text{order } (1 / c) \ (\text{reflect-poly } (\text{reflect-poly } p))$   
**by** (*intro order-reflect-poly-ge*) *auto*  
**with** *assms* **show**  $\text{order } c \ (\text{reflect-poly } p) \leq \text{order } (1 / c) \ p$   
**by** *simp*

**qed**

**lemma** *poly-reflect-eq-0-iff:*

$\text{poly } (\text{reflect-poly } p) \ (x :: 'a :: \text{field}) = 0 \iff p = 0 \vee x \neq 0 \wedge \text{poly } p \ (1 / x) = 0$   
**by** (*cases x = 0*) (*auto simp: poly-reflect-poly-nz inverse-eq-divide*)

**theorem** *ratfps-nth-bigo:*

**fixes**  $q :: \text{complex poly}$

**assumes**  $R > 0$   
**assumes**  $roots1: \bigwedge z. z \in ball\ 0\ (1 / R) \implies poly\ q\ z \neq 0$   
**assumes**  $roots2: \bigwedge z. z \in sphere\ 0\ (1 / R) \implies poly\ q\ z = 0 \implies order\ z\ q \leq$   
*Suc k*  
**shows**  $fps\text{-}nth\ (fps\text{-}of\text{-}poly\ p / fps\text{-}of\text{-}poly\ q) \in O(\lambda n. of\text{-}nat\ n^{\wedge} k * of\text{-}real\ R^{\wedge} n)$   
**proof** –  
**define**  $q'$  **where**  $q' = reflect\text{-}poly\ q$   
**from**  $roots1[of\ 0]$  **and**  $\langle R > 0 \rangle$  **have**  $[simp]: coeff\ q\ 0 \neq 0\ q \neq 0$   
**by**  $(auto\ simp: poly\text{-}0\text{-}coeff\text{-}0)$   
**from**  $ratfps\text{-}closed\text{-}form\text{-}exists[OF\ this(1), of\ p]$   
**obtain**  $r\ rs$  **where**  $closed\text{-}form:$   
 $\bigwedge n. (fps\text{-}of\text{-}poly\ p / fps\text{-}of\text{-}poly\ q)\ \$\ n =$   
 $coeff\ r\ n + (\sum\ c\ | poly\ (reflect\text{-}poly\ q)\ c = 0. poly\ (rs\ c)\ (of\text{-}nat\ n) * c^{\wedge} n)$   
 $\bigwedge z. poly\ (reflect\text{-}poly\ q)\ z = 0 \implies degree\ (rs\ z) \leq order\ z\ (reflect\text{-}poly\ q) - 1$   
**by**  $blast$   
  
**have**  $fps\text{-}nth\ (fps\text{-}of\text{-}poly\ p / fps\text{-}of\text{-}poly\ q) =$   
 $(\lambda n. coeff\ r\ n + (\sum\ c\ | poly\ q'\ c = 0. poly\ (rs\ c)\ (of\text{-}nat\ n) * c^{\wedge} n))$   
**by**  $(intro\ ext, subst\ closed\text{-}form)\ (simp\text{-}all\ add: q'\text{-}def)$   
**also** **have**  $\dots \in O(\lambda n. of\text{-}nat\ n^{\wedge} k * of\text{-}real\ R^{\wedge} n)$   
**proof**  $(intro\ sum\text{-}in\text{-}bigo\ big\text{-}sum\text{-}in\text{-}bigo)$   
**have**  $eventually\ (\lambda n. coeff\ r\ n = 0)$  **at**  $top$   
**using**  $MOST\text{-}nat\ coeff\text{-}eq\text{-}0\ cofinite\text{-}eq\text{-}sequentially$  **by**  $force$   
**hence**  $coeff\ r \in \Theta(\lambda\cdot. 0)$  **by**  $(rule\ bignthetaI\text{-}cong)$   
**also** **have**  $(\lambda\cdot. 0 :: complex) \in O(\lambda n. of\text{-}nat\ n^{\wedge} k * of\text{-}real\ R^{\wedge} n)$   
**by**  $simp$   
**finally** **show**  $coeff\ r \in O(\lambda n. of\text{-}nat\ n^{\wedge} k * of\text{-}real\ R^{\wedge} n)$  .  
**next**  
**fix**  $c$  **assume**  $c: c \in \{c. poly\ q'\ c = 0\}$   
**hence**  $[simp]: c \neq 0$  **by**  $(auto\ simp: q'\text{-}def)$   
  
**show**  $(\lambda n. poly\ (rs\ c)\ n * c^{\wedge} n) \in O(\lambda n. of\text{-}nat\ n^{\wedge} k * of\text{-}real\ R^{\wedge} n)$   
**proof**  $(cases\ norm\ c = R)$   
**case**  $True$  — The case of a root at the border of the disc  
**show**  $?thesis$   
**proof**  $(intro\ landau\text{-}o.\text{big}.\text{mult}\ landau\text{-}o.\text{big}.\text{compose}[OF\ poly\text{-}bigo\ tendsto\text{-}of\text{-}nat])$   
**have**  $degree\ (rs\ c) \leq order\ c\ (reflect\text{-}poly\ q) - 1$   
**using**  $c$  **by**  $(intro\ closed\text{-}form(2))\ (auto\ simp: q'\text{-}def)$   
**also** **have**  $order\ c\ (reflect\text{-}poly\ q) = order\ (1 / c)\ q$   
**using**  $c$  **by**  $(intro\ order\text{-}reflect\text{-}poly)\ (auto\ simp: q'\text{-}def)$   
**also** {  
**have**  $order\ (1 / c)\ q \leq Suc\ k$  **using**  $\langle R > 0 \rangle$  **and**  $True$  **and**  $c$   
**by**  $(intro\ roots2)\ (auto\ simp: q'\text{-}def\ norm\text{-}divide\ poly\text{-}reflect\text{-}eq\text{-}0\text{-}iff)$   
**moreover** **have**  $order\ (1 / c)\ q \neq 0$   
**using**  $order\text{-}root[of\ q\ 1 / c]\ c$  **by**  $(auto\ simp: q'\text{-}def\ poly\text{-}reflect\text{-}eq\text{-}0\text{-}iff)$   
**ultimately** **have**  $order\ (1 / c)\ q - 1 \leq k$  **by**  $simp$   
**}**  
**finally** **show**  $degree\ (rs\ c) \leq k$  .

```

next
  have  $(\lambda n. \text{norm } (c \wedge n)) \in O(\lambda n. \text{norm } (\text{complex-of-real } R \wedge n))$ 
    using True and  $\langle R > 0 \rangle$  by (simp add: norm-power)
  thus  $(\lambda n. c \wedge n) \in O(\lambda n. \text{complex-of-real } R \wedge n)$ 
    by (subst (asm) landau-o.big.norm-iff)
qed auto
next
case False — The case of a root in the interior of the disc
hence  $\text{norm } c < R$  using c and roots1[ $\text{of } 1/c$ ] and  $\langle R > 0 \rangle$ 
  by (cases norm c R rule: linorder-cases)
    (auto simp: q'-def poly-reflect-eq-0-iff norm-divide field-simps)
define l where  $l = \text{degree } (rs \ c)$ 

have  $(\lambda n. \text{poly } (rs \ c) (\text{of-nat } n) * c \wedge n) \in O(\lambda n. \text{of-nat } n \wedge l * c \wedge n)$ 
by (intro landau-o.big.mult landau-o.big.compose[OF poly-bigo tendsto-of-nat])
  (auto simp: l-def)
also have  $(\lambda n. \text{of-nat } n \wedge l * c \wedge n) \in O(\lambda n. \text{of-nat } n \wedge k * \text{of-real } R \wedge n)$ 
proof (subst landau-o.big.norm-iff [symmetric])
  have  $(\lambda n. \text{real } n \wedge l) \in O(\lambda n. \text{real } n \wedge k * (R / \text{norm } c) \wedge n)$ 
    using  $\langle \text{norm } c < R \rangle$  and  $\langle R > 0 \rangle$  by real-asymp
  hence  $(\lambda n. \text{real } n \wedge l * \text{norm } c \wedge n) \in O(\lambda n. \text{real } n \wedge k * R \wedge n)$ 
    by (simp add: power-divide landau-o.big.divide-eq1)
  thus  $(\lambda x. \text{norm } (\text{of-nat } x \wedge l * c \wedge x)) \in$ 
     $O(\lambda x. \text{norm } (\text{of-nat } x \wedge k * \text{complex-of-real } R \wedge x))$ 
    unfolding norm-power norm-mult using  $\langle R > 0 \rangle$  by simp
qed
finally show ?thesis .
qed
qed
finally show ?thesis .
qed

lemma order-power:  $p \neq 0 \implies \text{order } c (p \wedge n) = n * \text{order } c \ p$ 
  by (induction n) (auto simp: order-mult)

lemma same-root-imp-not-coprime:
  assumes poly p x = 0 and poly q (x :: 'a :: {factorial-ring-gcd, semiring-gcd-mult-normalize})
  = 0
  shows  $\neg \text{coprime } p \ q$ 
proof
  assume coprime p q
  from assms have  $[-x, 1:] \ \text{dvd } p$  and  $[-x, 1:] \ \text{dvd } q$ 
    by (simp-all add: poly-eq-0-iff-dvd)
  hence  $[-x, 1:] \ \text{dvd } \text{gcd } p \ q$  by (simp add: poly-eq-0-iff-dvd)
  also from  $\langle \text{coprime } p \ q \rangle$  have  $\text{gcd } p \ q = 1$ 
    by (rule coprime-imp-gcd-eq-1)
  finally show False by (elim is-unit-polyE) auto
qed

```

**lemma** *ratfps-nth-bigo-square-free-factorization*:

**fixes**  $p :: \text{complex poly}$

**assumes** *square-free-factorization*  $q (b, cs)$

**assumes**  $q \neq 0$  **and**  $R > 0$

**assumes** *roots1*:  $\bigwedge c l. (c, l) \in \text{set } cs \implies \forall x \in \text{ball } 0 (1 / R). \text{poly } c x \neq 0$

**assumes** *roots2*:  $\bigwedge c l. (c, l) \in \text{set } cs \implies l > k \implies \forall x \in \text{sphere } 0 (1 / R). \text{poly } c x \neq 0$

**shows**  $\text{fps-nth } (\text{fps-of-poly } p / \text{fps-of-poly } q) \in O(\lambda n. \text{of-nat } n \wedge k * \text{of-real } R \wedge n)$

**proof** –

**from** *assms(1)* **have**  $q: q = \text{smult } b (\prod (c, l) \in \text{set } cs. c \wedge \text{Suc } l)$

**unfolding** *square-free-factorization-def prod.case* **by** *blast*

**with**  $\langle q \neq 0 \rangle$  **have** [*simp*]:  $b \neq 0$  **by** *auto*

**from** *assms(1)* **have** [*simp*]:  $(0, x) \notin \text{set } cs$  **for**  $x$

**by** (*auto simp: square-free-factorization-def*)

**from** *assms(1)* **have** *coprime*:  $c1 = c2 \ m = n$

**if**  $\neg \text{coprime } c1 \ c2$   $(c1, m) \in \text{set } cs$   $(c2, n) \in \text{set } cs$  **for**  $c1 \ c2 \ m \ n$

**using** *that* **by** (*auto simp: square-free-factorization-def case-prod-unfold*)

**show** *?thesis*

**proof** (*rule ratfps-nth-bigo*)

**fix**  $z :: \text{complex}$  **assume**  $z: z \in \text{ball } 0 (1 / R)$

**show**  $\text{poly } q z \neq 0$

**proof**

**assume**  $\text{poly } q z = 0$

**then obtain**  $c \ l$  **where**  $cl: (c, l) \in \text{set } cs$  **and**  $\text{poly } c z = 0$

**by** (*auto simp: q poly-prod image-iff*)

**with** *roots1*[*of c l*] **and**  $z$  **show** *False* **by** *auto*

**qed**

**next**

**fix**  $z :: \text{complex}$  **assume**  $z: z \in \text{sphere } 0 (1 / R)$

**have** *order*:  $\text{order } z q = \text{order } z (\prod (c, l) \in \text{set } cs. c \wedge \text{Suc } l)$

**by** (*simp add: order-smult q*)

**also have**  $\dots = (\sum x \in \text{set } cs. \text{order } z (\text{case } x \text{ of } (c, l) \Rightarrow c \wedge \text{Suc } l))$

**by** (*subst order-prod*) (*auto dest: coprime*)

**also have**  $\dots = (\sum (c, l) \in \text{set } cs. \text{Suc } l * \text{order } z c)$

**unfolding** *case-prod-unfold* **by** (*intro sum.cong refl, subst order-power*) *auto*

**finally have**  $\text{order } z q = \dots$  .

**show**  $\text{order } z q \leq \text{Suc } k$

**proof** (*cases*  $\exists c0 \ l0. (c0, l0) \in \text{set } cs \wedge \text{poly } c0 z = 0$ )

**case** *False*

**have**  $\text{order } z q = (\sum (c, l) \in \text{set } cs. \text{Suc } l * \text{order } z c)$  **by** *fact*

**also have**  $\text{order } z c = 0$  **if**  $(c, l) \in \text{set } cs$  **for**  $c \ l$

**using** *False that* **by** (*auto simp: order-root*)

**hence**  $(\sum (c, l) \in \text{set } cs. \text{Suc } l * \text{order } z c) = 0$   
**by** *(intro sum.neutral) auto*  
**finally show**  $\text{order } z q \leq \text{Suc } k$  **by** *simp*  
**next**  
**case** *True* — The order of a root is determined by the unique polynomial in the square-free factorisation that contains it.  
**then obtain**  $c0\ l0$  **where**  $cl0: (c0, l0) \in \text{set } cs \text{ poly } c0\ z = 0$   
**by** *blast*  
**have**  $\text{order } z q = (\sum (c, l) \in \text{set } cs. \text{Suc } l * \text{order } z c)$  **by** *fact*  
**also have**  $\dots = \text{Suc } l0 * \text{order } z c0 + (\sum (c, l) \in \text{set } cs - \{(c0, l0)\}. \text{Suc } l * \text{order } z c)$   
**using**  $cl0$  **by** *(subst sum.remove[of - (c0, l0)]) auto*  
**also have**  $(\sum (c, l) \in \text{set } cs - \{(c0, l0)\}. \text{Suc } l * \text{order } z c) = 0$   
**proof** *(intro sum.neutral ballI, goal-cases)*  
**case** *(1 cl)*  
**then obtain**  $c\ l$  **where**  $[simp]: cl = (c, l)$  **and**  $cl: (c, l) \in \text{set } cs (c0, l0) \neq (c, l)$   
**by** *(cases cl) auto*  
**from**  $cl$  **and**  $cl0$  **and** *coprime[of c c0 l l0]* **have** *coprime c c0*  
**by** *auto*  
**with** *same-root-imp-not-coprime[of c z c0]* **and**  $cl0$  **have**  $\text{poly } c\ z \neq 0$  **by** *auto*  
**thus** *?case* **by** *(auto simp: order-root)*  
**qed**  
**also have** *square-free c0* **using**  $cl0$  *assms(1)*  
**by** *(auto simp: square-free-factorization-def)*  
**hence** *rsquarefree c0* **by** *(rule square-free-rsquarefree)*  
**with**  $cl0$  **have**  $\text{order } z c0 = 1$   
**by** *(auto simp: rsquarefree-def' order-root intro: antisym)*  
**finally have**  $\text{order } z q = \text{Suc } l0$  **by** *simp*  
  
**also from** *roots2[of c0 l0]*  $cl0\ z$  **have**  $l0 \leq k$   
**by** *(cases l0 k rule: linorder-cases) auto*  
**finally show**  $\text{order } z q \leq \text{Suc } k$  **by** *simp*  
**qed**  
**qed** *fact+*  
**qed**

**find-consts** *name:Count-Complex*

**term** *proots-ball-card*

**term** *proots-sphere-card*

**lemma** *proots-within-card-zero-iff:*

**assumes**  $p \neq (0 :: 'a :: \text{idom poly})$

**shows**  $\text{card } (\text{proots-within } p\ A) = 0 \iff (\forall x \in A. \text{poly } p\ x \neq 0)$

**using** *assms* **by** *(subst card-0-eq) (auto intro: finite-proots)*

**lemma** *ratfps-nth-bigo-square-free-factorization':*



**fixes**  $p :: \text{complex poly}$   
**assumes** *square-free-factorization*  $q (b, cs)$   
**assumes**  $q \neq 0$  **and**  $R > 0$   
**assumes** *roots1*: *list-all*  $(\lambda cl. \text{proots-ball-card } (fst\ cl)\ 0\ (1 / R) = 0)\ cs$   
**assumes** *roots2*: *list-all*  $(\lambda cl. \text{proots-sphere-card } (fst\ cl)\ 0\ (1 / R) = 0)$   
*(filter*  $(\lambda cl. \text{snd } cl > k)\ cs)$   
**shows**  $\text{fps-nth } (\text{fps-of-poly } p / \text{fps-of-poly } q) \in O(\lambda n. \text{of-nat } n \wedge k * \text{of-real } R \wedge n)$   
**proof** (*rule ratfps-nth-bigo-square-free-factorization*[*OF assms*(1)])  
**from** *assms*(1) **have**  $q = \text{smult } b (\prod_{(c, l) \in \text{set } cs} c \wedge \text{Suc } l)$   
**unfolding** *square-free-factorization-def prod.case* **by** *blast*  
**with**  $\langle q \neq 0 \rangle$  **have** [*simp*]:  $b \neq 0$  **by** *auto*  
**from** *assms*(1) **have** [*simp*]:  $(0, x) \notin \text{set } cs$  **for**  $x$   
**by** (*auto simp: square-free-factorization-def*)

**show**  $\forall x \in \text{ball } 0\ (1 / R). \text{poly } c\ x \neq 0$  **if**  $(c, l) \in \text{set } cs$  **for**  $c\ l$   
**proof** –  
**from** *roots1* **that** **have**  $\text{card } (\text{proots-within } c\ (\text{ball } 0\ (1 / R))) = 0$   
**by** (*auto simp: proots-ball-card-def list-all-def*)  
**with** *that* **show** *?thesis* **by** (*subst (asm) proots-within-card-zero-iff*) *auto*  
**qed**

**show**  $\forall x \in \text{sphere } 0\ (1 / R). \text{poly } c\ x \neq 0$  **if**  $(c, l) \in \text{set } cs$   $l > k$  **for**  $c\ l$   
**proof** –  
**from** *roots2* **that** **have**  $\text{card } (\text{proots-within } c\ (\text{sphere } 0\ (1 / R))) = 0$   
**by** (*auto simp: proots-sphere-card-def list-all-def*)  
**with** *that* **show** *?thesis* **by** (*subst (asm) proots-within-card-zero-iff*) *auto*  
**qed**

**qed fact+**

**definition** *ratfps-has-asymptotics* **where**

$\text{ratfps-has-asymptotics } q\ k\ R \longleftrightarrow q \neq 0 \wedge R > 0 \wedge$   
*(let*  $cs = \text{snd } (\text{yun-factorization } \text{gcd } q)$   
*in*  $\text{list-all } (\lambda cl. \text{proots-ball-card } (fst\ cl)\ 0\ (1 / R) = 0)\ cs \wedge$   
 $\text{list-all } (\lambda cl. \text{proots-sphere-card } (fst\ cl)\ 0\ (1 / R) = 0)\ (\text{filter } (\lambda cl. \text{snd } cl > k)\ cs)$ )

**lemma** *ratfps-has-asymptotics-correct*:

**assumes** *ratfps-has-asymptotics*  $q\ k\ R$   
**shows**  $\text{fps-nth } (\text{fps-of-poly } p / \text{fps-of-poly } q) \in O(\lambda n. \text{of-nat } n \wedge k * \text{of-real } R \wedge n)$

**proof** (*rule ratfps-nth-bigo-square-free-factorization'*)

**show** *square-free-factorization*  $q$  (*fst*  $(\text{yun-factorization } \text{gcd } q)$ , *snd*  $(\text{yun-factorization } \text{gcd } q)$ )

**by** (*rule yun-factorization*) *simp*

**qed** (*insert assms, auto simp: ratfps-has-asymptotics-def Let-def list-all-def*)

**value** *map*  $(\text{fps-nth } (\text{fps-of-poly } [:0, 1:] / \text{fps-of-poly } [:1, -1, -1 :: \text{real}])) [0..<5]$

```

method ratfps-bigo = (rule ratfps-has-asymptotics-correct; eval)

lemma fps-nth (fps-of-poly [:0, 1:] / fps-of-poly [:1, -1, -1 :: complex:]) ∈
  O(λn. of-nat n ^ 0 * complex-of-real 1.618034 ^ n)
by ratfps-bigo

lemma fps-nth (fps-of-poly 1 / fps-of-poly [:1, -3, 3, -1 :: complex:]) ∈
  O(λn. of-nat n ^ 2 * complex-of-real 1 ^ n)
by ratfps-bigo

lemma fps-nth (fps-of-poly f / fps-of-poly [:5, 4, 3, 2, 1 :: complex:]) ∈
  O(λn. of-nat n ^ 0 * complex-of-real 0.69202 ^ n)
by ratfps-bigo

end

```