

A Verified Solver for Linear Recurrences

Manuel Eberl

June 24, 2019

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 φ 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

| | | |
|----------|--------------------------------------------------------------------------|-----------|
| 1 | Rational formal power series | 3 |
| 1.1 | Some auxiliary | 3 |
| 1.2 | The type of rational formal power series | 3 |
| 2 | Falling factorial as a polynomial | 18 |
| 3 | Miscellaneous material required for linear recurrences | 18 |
| 4 | Partial Fraction Decomposition | 21 |
| 4.1 | Decomposition on general Euclidean rings | 21 |
| 4.2 | Specific results for polynomials | 24 |
| 5 | Factorizations of polynomials | 26 |
| 6 | Solver for rational formal power series | 28 |
| 7 | Material common to homogenous and inhomogenous linear recurrences | 31 |

| | | |
|----|---------------------------------|----|
| 8 | Homogenous linear recurrences | 31 |
| 9 | Eulerian polynomials | 33 |
| 10 | Inhomogenous linear recurrences | 35 |

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
  <proof>
```

```
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>
```

```
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>
```

```
lemma coeff-0-normalize-quot-nonzero [simp]:
  assumes coeff (snd x) 0  $\neq$  0
  shows coeff (snd (normalize-quot x)) 0  $\neq$  0
  <proof>
```

```
abbreviation numerator :: 'a fract  $\Rightarrow$  'a::{ring-gcd, idom-divide}
where numerator x  $\equiv$  fst (quot-of-fract x)
```

```
abbreviation denominator :: 'a fract  $\Rightarrow$  'a::{ring-gcd, idom-divide}
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
  <proof>
```

1.2 The type of rational formal power series

```
typedef (overloaded) 'a :: {field, factorial-ring-gcd} ratfps =
```

```

    {x :: 'a poly fract. constant-term (denominator x) ≠ 0}
    ⟨proof⟩

setup-lifting type-definition-ratfps

instantiation ratfps :: ({field,factorial-ring-gcd}) idom
begin

lift-definition zero-ratfps :: 'a ratfps is 0 ⟨proof⟩

lift-definition one-ratfps :: 'a ratfps is 1 ⟨proof⟩

lift-definition uminus-ratfps :: 'a ratfps ⇒ 'a ratfps is uminus
    ⟨proof⟩

lift-definition plus-ratfps :: 'a ratfps ⇒ 'a ratfps ⇒ 'a ratfps is (+)
    ⟨proof⟩

lift-definition minus-ratfps :: 'a ratfps ⇒ 'a ratfps ⇒ 'a ratfps is (-)
    ⟨proof⟩

lift-definition times-ratfps :: 'a ratfps ⇒ 'a ratfps ⇒ 'a ratfps is (*)
    ⟨proof⟩

instance
    ⟨proof⟩

end

fun ratfps-nth-aux :: ('a::field) poly ⇒ nat ⇒ 'a
where
    ratfps-nth-aux p 0 = inverse (coeff p 0)
  | ratfps-nth-aux p n =
    - inverse (coeff p 0) * sum (λi. coeff p i * 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
    ⟨proof⟩

lift-definition ratfps-nth :: 'a :: {field,factorial-ring-gcd} ratfps ⇒ nat ⇒ 'a is
    λx n. let (a,b) = quot-of-fract x
    in (∑ i = 0..n. coeff a i * ratfps-nth-aux b (n - i)) ⟨proof⟩

lift-definition ratfps-subdegree :: 'a :: {field,factorial-ring-gcd} ratfps ⇒ nat is
    λx. poly-subdegree (fst (quot-of-fract x)) ⟨proof⟩

context
includes lifting-syntax
begin

```

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

end

lemma *normalize-quot-quot-of-fract* [*simp*]:
normalize-quot (*quot-of-fract* x) = *quot-of-fract* x
<proof>

context
assumes *SORT-CONSTRAINT*('a::{*field*,*factorial-ring-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*) *<proof>*

lift-definition *quot-to-ratfps* :: ('a *poly* \times 'a *poly*) \Rightarrow 'a *ratfps* **is**
 $\lambda(x,y). \text{let } (x',y') = \text{normalize-quot } (x,y)$
 *in if coeff } y' \ 0 = 0 \text{ then } 0 \text{ else quot-to-fract } (x',y')
*<proof>**

lemma *quot-to-ratfps-quot-of-ratfps* [*code abstype*]:
quot-to-ratfps (*quot-of-ratfps* x) = x
<proof>

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

lemma *quot-of-ratfps-quot-to-ratfps*:
coeff (*snd* x) $0 \neq 0 \implies x \in \text{normalized-fracts} \implies \text{quot-of-ratfps } (\text{quot-to-ratfps } x) = x$
<proof>

lemma *quot-of-ratfps-0* [*simp*, *code abstract*]: *quot-of-ratfps* $0 = (0, 1)$
<proof>

lemma *quot-of-ratfps-1* [*simp*, *code abstract*]: *quot-of-ratfps* $1 = (1, 1)$
<proof>

lift-definition *ratfps-of-poly* :: 'a *poly* \Rightarrow 'a *ratfps* **is**
to-fract :: 'a *poly* \Rightarrow -
<proof>

lemma *ratfps-of-poly-code* [*code abstract*]:

$\text{quot-of-ratfps} (\text{ratfps-of-poly } p) = (p, 1)$
 $\langle \text{proof} \rangle$

lemmas $\text{zero-ratfps-code} = \text{quot-of-ratfps-0}$

lemmas $\text{one-ratfps-code} = \text{quot-of-ratfps-1}$

lemma $\text{uminus-ratfps-code}$ [code abstract]:
 $\text{quot-of-ratfps} (-x) = (\text{let } (a, b) = \text{quot-of-ratfps } x \text{ in } (-a, b))$
 $\langle \text{proof} \rangle$

lemma plus-ratfps-code [code abstract]:
 $\text{quot-of-ratfps} (x + y) =$
 $(\text{let } (a, b) = \text{quot-of-ratfps } x; (c, d) = \text{quot-of-ratfps } y$
 $\text{in } \text{normalize-quot} (a * d + b * c, b * d))$
 $\langle \text{proof} \rangle$

lemma minus-ratfps-code [code abstract]:
 $\text{quot-of-ratfps} (x - y) =$
 $(\text{let } (a, b) = \text{quot-of-ratfps } x; (c, d) = \text{quot-of-ratfps } y$
 $\text{in } \text{normalize-quot} (a * d - b * c, b * d))$
 $\langle \text{proof} \rangle$

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

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

lemma times-ratfps-code [code abstract]:
 $\text{quot-of-ratfps} (x * y) =$
 $(\text{let } (a, b) = \text{quot-of-ratfps } x; (c, d) = \text{quot-of-ratfps } y;$
 $(e, f) = \text{normalize-quot} (a, d); (g, h) = \text{normalize-quot} (c, b)$
 $\text{in } (e * g, f * h))$
 $\langle \text{proof} \rangle$

lemma ratfps-nth-code [code]:
 $\text{ratfps-nth } x \ n =$
 $(\text{let } (a, b) = \text{quot-of-ratfps } x$
 $\text{in } \sum_{i = 0..n} \text{coeff } a \ i * \text{ratfps-nth-aux } b \ (n - i))$
 $\langle \text{proof} \rangle$

lemma $\text{ratfps-subdegree-code}$ [code]:
 $\text{ratfps-subdegree } x = \text{poly-subdegree} (\text{fst} (\text{quot-of-ratfps } x))$
 $\langle \text{proof} \rangle$

end

instantiation $\text{ratfps} :: (\{\text{field}, \text{factorial-ring-gcd}\}) \text{ inverse}$
begin

lift-definition $\text{inverse-ratfps} :: 'a \text{ ratfps} \Rightarrow 'a \text{ ratfps}$ **is**
 $\lambda x. \text{let } (a,b) = \text{quot-of-fract } x$
 $\text{in if coeff } a \ 0 = 0 \text{ then } 0 \text{ else inverse } x$
 $\langle \text{proof} \rangle$

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

instance $\langle \text{proof} \rangle$
end

lemma $\text{ratfps-inverse-code}$ [code abstract]:
 $\text{quot-of-ratfps } (\text{inverse } x) =$
 $(\text{let } (a,b) = \text{quot-of-ratfps } x$
 $\text{in if coeff } a \ 0 = 0 \text{ then } (0, 1)$
 $\text{else let } u = \text{unit-factor } a \text{ in } (b \ \text{div } u, a \ \text{div } u))$
 $\langle \text{proof} \rangle$

instantiation $\text{ratfps} :: (\text{equal}) \text{ equal}$
begin

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

instance $\langle \text{proof} \rangle$

end

lemma $\text{quot-of-fract-eq-iff}$ [simp]: $\text{quot-of-fract } x = \text{quot-of-fract } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma equal-ratfps-code [code]: $\text{HOL.equal } x \ y \longleftrightarrow \text{quot-of-ratfps } x = \text{quot-of-ratfps } y$
 $\langle \text{proof} \rangle$

lemma $\text{fps-of-poly-quot-normalize-quot}$ [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 $(\text{snd } x :: 'a :: \{\text{field}, \text{factorial-ring-gcd}\} \text{ poly}) \neq 0$
 $\langle \text{proof} \rangle$

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 $\text{coeff } (\text{snd } x) \neq 0$ ($0 :: 'a :: \{\text{field}, \text{factorial-ring-gcd}\}$)
 ⟨proof⟩

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

lemma *fps-of-ratfps-altdef*:

$$\text{fps-of-ratfps } x = (\text{case } \text{quot-of-ratfps } x \text{ of } (a, b) \Rightarrow \text{fps-of-poly } a / \text{fps-of-poly } b)$$
 ⟨proof⟩

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

lemma *fps-of-ratfps-0* [simp]: $\text{fps-of-ratfps } 0 = 0$
 ⟨proof⟩

lemma *fps-of-ratfps-1* [simp]: $\text{fps-of-ratfps } 1 = 1$
 ⟨proof⟩

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

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

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

lemma *is-unit-div-div-commute*: $\text{is-unit } b \Longrightarrow \text{is-unit } c \Longrightarrow a \text{ div } b \text{ div } c = a \text{ div } c \text{ div } b$
 ⟨proof⟩

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

lemma *div-const-unit-poly*: $\text{is-unit } c \Longrightarrow p \text{ div } [:c:] = \text{smult } (1 \text{ div } c) p$
 ⟨proof⟩

lemma *normalize-field*:

$$\text{normalize } (x :: 'a :: \{\text{normalization-semidom}, \text{field}\}) = (\text{if } x = 0 \text{ then } 0 \text{ else } 1)$$
 ⟨proof⟩

lemma *unit-factor-field* [simp]:
 $unit_factor\ (x :: 'a :: \{normalization-semidom, field\}) = x$
 ⟨proof⟩

lemma *fps-of-poly-normalize-field*:
 $fps_of_poly\ (normalize\ (p :: 'a :: \{field, normalization-semidom\}\ poly)) =$
 $fps_of_poly\ p * fps_const\ (inverse\ (lead_coeff\ p))$
 ⟨proof⟩

lemma *unit-factor-poly-altdef*: $unit_factor\ p = monom\ (unit_factor\ (lead_coeff\ p))$
 0
 ⟨proof⟩

lemma *div-const-poly*: $p\ div\ [:c::'a::field:] = smult\ (inverse\ c)\ p$
 ⟨proof⟩

lemma *fps-of-ratfps-inverse* [simp]: $fps_of_ratfps\ (inverse\ x) = inverse\ (fps_of_ratfps\ x)$
 ⟨proof⟩

context
includes *fps-notation*
begin

lemma *ratfps-nth-altdef*: $ratfps_nth\ x\ n = fps_of_ratfps\ x\ \$\ n$
 ⟨proof⟩

lemma *fps-of-ratfps-is-unit*: $fps_of_ratfps\ a\ \$\ 0 \neq 0 \longleftrightarrow ratfps_nth\ a\ 0 \neq 0$
 ⟨proof⟩

lemma *ratfps-nth-0* [simp]: $ratfps_nth\ 0\ n = 0$
 ⟨proof⟩

lemma *fps-of-ratfps-cases*:
obtains $p\ q$ **where** $coeff\ q\ 0 \neq 0$ $fps_of_ratfps\ f = fps_of_poly\ p / fps_of_poly\ q$
 ⟨proof⟩

lemma *fps-of-ratfps-cutoff* [simp]:
 $fps_of_poly\ (ratfps_cutoff\ n\ x) = fps_cutoff\ n\ (fps_of_ratfps\ x)$
 ⟨proof⟩

lemma *subdegree-fps-of-ratfps*:
 $subdegree\ (fps_of_ratfps\ x) = ratfps_subdegree\ x$
 ⟨proof⟩

lemma *ratfps-subdegree-altdef*:
 $ratfps_subdegree\ x = subdegree\ (fps_of_ratfps\ x)$
 ⟨proof⟩

end

code-datatype *fps-of-ratfps*

lemma *fps-zero-code* [*code*]: $0 = \text{fps-of-ratfps } 0$ *<proof>*

lemma *fps-one-code* [*code*]: $1 = \text{fps-of-ratfps } 1$ *<proof>*

lemma *fps-const-code* [*code*]: $\text{fps-const } c = \text{fps-of-poly } [:c:]$ *<proof>*

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

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

lemma *fps-nth-code* [*code*]: $\text{fps-nth } (\text{fps-of-ratfps } x) n = \text{ratfps-nth } x n$
<proof>

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

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

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

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

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

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

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

lemma *fractrel-normalize-quot*:

$\text{fractrel } p p \implies \text{fractrel } q q \implies$

$\text{fractrel } (\text{normalize-quot } p) (\text{normalize-quot } q) \iff \text{fractrel } p q$

<proof>

lemma *fps-of-ratfps-eq-iff* [*simp*]:

$\text{fps-of-ratfps } p = \text{fps-of-ratfps } q \iff p = q$

<proof>

lemma *fps-of-ratfps-eq-zero-iff* [simp]:

$$\text{fps-of-ratfps } p = 0 \longleftrightarrow p = 0$$

<proof>

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

$$\text{unit-factor (snd (quot-of-ratfps } x)) = 1$$

<proof>

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

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

<proof>

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

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

<proof>

lemma *poly-shift-times-monom*:

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

<proof>

lemma *monom-times-poly-shift*:

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

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

<proof>

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

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

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

<proof>

lemma *subdegree-minus-cutoff-ge*:

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

shows $\text{subdegree } (f - \text{fps-cutoff } n \ f) \geq n$

<proof>

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

$\text{fps}) = f$

<proof>

lemma

ratfps-shift-code [code abstract]:

$$\text{quot-of-ratfps } (\text{ratfps-shift } n \ x) =$$

$$(\text{let } (a, b) = \text{quot-of-ratfps } (x - \text{ratfps-of-poly } (\text{ratfps-cutoff } n \ x))$$

$$\text{in } (\text{poly-shift } n \ a, b)) \text{ (is ?lhs1 = ?rhs1) and}$$

fps-of-ratfps-shift [simp]:

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

<proof>

include *fps-notation*

<proof>

lemma *fps-shift-code* [code]: $\text{fps-shift } n (\text{fps-of-ratfps } x) = \text{fps-of-ratfps } (\text{ratfps-shift } n \ x)$
 ⟨proof⟩

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 ⟨proof⟩

end

lemma *equal-fps-code* [code]: $\text{HOL.equal } (\text{fps-of-ratfps } f) (\text{fps-of-ratfps } g) \longleftrightarrow f = g$
 ⟨proof⟩

lemma *fps-of-ratfps-divide* [simp]:
 $\text{fps-of-ratfps } (f \ \text{div} \ g) = \text{fps-of-ratfps } f \ \text{div} \ \text{fps-of-ratfps } g$
 ⟨proof⟩

lemma *ratfps-eqI*: $\text{fps-of-ratfps } x = \text{fps-of-ratfps } y \Longrightarrow x = y$ ⟨proof⟩

instance *ratfps* :: ({field,factorial-ring-gcd}) algebraic-semidom
 ⟨proof⟩

lemma *fps-of-ratfps-dvd* [simp]:
 $\text{fps-of-ratfps } x \ \text{dvd} \ \text{fps-of-ratfps } y \longleftrightarrow x \ \text{dvd} \ y$
 ⟨proof⟩

lemma *is-unit-ratfps-iff* [simp]:
 $\text{is-unit } x \longleftrightarrow \text{ratfps-nth } x \ 0 \neq 0$
 ⟨proof⟩

instantiation *ratfps* :: ({field,factorial-ring-gcd}) normalization-semidom
begin

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

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

lemma *fps-of-ratfps-unit-factor* [simp]:
 $\text{fps-of-ratfps } (\text{unit-factor } x) = \text{unit-factor } (\text{fps-of-ratfps } x)$
 ⟨proof⟩

lemma *fps-of-ratfps-normalize* [*simp*]:
 $fps\text{-of-ratfps } (normalize\ x) = normalize\ (fps\text{-of-ratfps } x)$
<proof>

instance *<proof>*

end

instantiation *ratfps* :: (*{field,factorial-ring-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*
 $let\ n = ratfps\text{-subdegree } g; h = ratfps\text{-shift } n\ g$
 $in\ ratfps\text{-of-poly } (ratfps\text{-cutoff } n\ (f * inverse\ h)) * h$ *<proof>*

lemma *fps-of-ratfps-mod* [*simp*]:
 $fps\text{-of-ratfps } (f\ mod\ g :: 'a\ ratfps) = fps\text{-of-ratfps } f\ mod\ fps\text{-of-ratfps } g$
<proof>

instance
<proof>

end

instantiation *ratfps* :: (*{field,factorial-ring-gcd}*) *euclidean-ring*
begin

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

lemma *fps-of-ratfps-euclidean-size* [*simp*]:
 $euclidean\text{-size } x = euclidean\text{-size } (fps\text{-of-ratfps } x)$
<proof>

instance *<proof>*

end

instantiation *ratfps* :: (*{field,factorial-ring-gcd}*) *euclidean-ring-cancel*
begin

instance
<proof>

end

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

⟨proof⟩

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

lemma *fps-dvd-code* [code-unfold]:
 $x \text{ dvd } y \longleftrightarrow y = 0 \vee ((x :: 'a :: \{\text{field, factorial-ring-gcd}\} \text{ fps}) \neq 0 \wedge \text{subdegree } x \leq \text{subdegree } y)$
⟨proof⟩

lemma *ratfps-dvd-code* [code-unfold]:
 $x \text{ dvd } y \longleftrightarrow y = 0 \vee (x \neq 0 \wedge \text{ratfps-subdegree } x \leq \text{ratfps-subdegree } y)$
⟨proof⟩

instance *ratfps* :: ($\{\text{field, factorial-ring-gcd}\}$) *normalization-euclidean-semiring* ⟨proof⟩

instantiation *ratfps* :: ($\{\text{field, factorial-ring-gcd}\}$) *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 ⟨proof⟩
end

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

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

lemma *ratfps-gcd*:
assumes [simp]: $f \neq 0 \ g \neq 0$
shows $\text{gcd } f \ g = \text{ratfps-of-poly} (\text{monom } 1 (\text{min} (\text{ratfps-subdegree } f) (\text{ratfps-subdegree } g)))$
⟨proof⟩

lemma *ratfps-gcd-altdef*: $\text{gcd} (f :: 'a :: \{\text{field, factorial-ring-gcd}\} \text{ ratfps}) \ g =$
(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
 $\text{ratfps-of-poly} (\text{monom } 1 (\text{min} (\text{ratfps-subdegree } f) (\text{ratfps-subdegree } g))))$
⟨proof⟩

lemma *ratfps-lcm*:

assumes $[simp]: f \neq 0 \ g \neq 0$
shows $lcm\ f\ g = ratfps\text{-of-poly}\ (monom\ 1\ (max\ (ratfps\text{-subdegree}\ f)\ (ratfps\text{-subdegree}\ g)))$
 $\langle proof \rangle$

lemma $ratfps\text{-lcm-altdef}: lcm\ (f :: 'a :: \{field, factorial\text{-ring-gcd}\}\ ratfps)\ g =$
 $(if\ f = 0 \vee g = 0\ then\ 0\ else$
 $\quad ratfps\text{-of-poly}\ (monom\ 1\ (max\ (ratfps\text{-subdegree}\ f)\ (ratfps\text{-subdegree}\ g))))$
 $\langle proof \rangle$

lemma $ratfps\text{-Gcd}$:
assumes $A - \{0\} \neq \{\}$
shows $Gcd\ A = ratfps\text{-of-poly}\ (monom\ 1\ (INF\ f \in A - \{0\}.\ ratfps\text{-subdegree}\ f))$
 $\langle proof \rangle$

lemma $ratfps\text{-Gcd-altdef}: Gcd\ (A :: 'a :: \{field, factorial\text{-ring-gcd}\}\ ratfps\ set) =$
 $(if\ A \subseteq \{0\}\ then\ 0\ else\ ratfps\text{-of-poly}\ (monom\ 1\ (INF\ f \in A - \{0\}.\ ratfps\text{-subdegree}\ f)))$
 $\langle proof \rangle$

lemma $ratfps\text{-Lcm}$:
assumes $A \neq \{\}\ 0 \notin A\ bdd\text{-above}\ (ratfps\text{-subdegree}\ 'A)$
shows $Lcm\ A = ratfps\text{-of-poly}\ (monom\ 1\ (SUP\ f \in A.\ ratfps\text{-subdegree}\ f))$
 $\langle proof \rangle$

lemma $ratfps\text{-Lcm-altdef}$:
 $Lcm\ (A :: 'a :: \{field, factorial\text{-ring-gcd}\}\ ratfps\ set) =$
 $(if\ 0 \in A \vee \neg bdd\text{-above}\ (ratfps\text{-subdegree}\ 'A)\ then\ 0\ else$
 $\quad if\ A = \{\}\ then\ 1\ else\ ratfps\text{-of-poly}\ (monom\ 1\ (SUP\ f \in A.\ ratfps\text{-subdegree}\ f)))$
 $\langle proof \rangle$

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

lemma $fps\text{-of-ratfps-quot-to-ratfps-code-post1}$:
 $fps\text{-of-ratfps}\ (quot\text{-to-ratfps}\ (x, pCons\ 1\ y)) = fps\text{-of-poly}\ x / fps\text{-of-poly}\ (pCons\ 1\ y)$
 $fps\text{-of-ratfps}\ (quot\text{-to-ratfps}\ (x, pCons\ (-1)\ y)) = fps\text{-of-poly}\ x / fps\text{-of-poly}\ (pCons\ (-1)\ y)$
 $\langle proof \rangle$

lemma $fps\text{-of-ratfps-quot-to-ratfps-code-post2}$:
 $fps\text{-of-ratfps}\ (quot\text{-to-ratfps}\ (x' :: 'a :: \{field\text{-char-0}, factorial\text{-ring-gcd}\}\ poly, pCons\ (numeral\ n)\ y')) =$
 $fps\text{-of-poly}\ x' / fps\text{-of-poly}\ (pCons\ (numeral\ n)\ y')$
 $fps\text{-of-ratfps}\ (quot\text{-to-ratfps}\ (x' :: 'a :: \{field\text{-char-0}, factorial\text{-ring-gcd}\}\ poly, pCons\ (numeral\ n)\ y')) =$
 $fps\text{-of-poly}\ x' / fps\text{-of-poly}\ (pCons\ (numeral\ n)\ y')$
 $\langle proof \rangle$

$(- \text{numeral } n) \ y') =$
 $\text{fps-of-poly } x' / \text{fps-of-poly } (\text{pCons } (- \text{numeral } n) \ y')$
 $\langle \text{proof} \rangle$

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

lemma fps-dehorner :

fixes $a \ b \ c :: 'a :: \text{semiring-1 fps}$ **and** $d \ e \ f :: 'b :: \text{ring-1 fps}$

shows

$(b + c) * \text{fps-X} = b * \text{fps-X} + c * \text{fps-X}$ $(a * \text{fps-X}) * \text{fps-X} = a * \text{fps-X} \wedge 2$
 $a * \text{fps-X} \wedge m * \text{fps-X} = a * \text{fps-X} \wedge (\text{Suc } m)$ $a * \text{fps-X} * \text{fps-X} \wedge m = a * \text{fps-X} \wedge (\text{Suc } m)$
 $a * \text{fps-X} \wedge m * \text{fps-X} \wedge n = a * \text{fps-X} \wedge (m+n)$ $a + (b + c) = a + b + c$ $a * 1 = a$
 $1 * a = a$
 $d + - e = d - e$ $(-d) * e = - (d * e)$ $d + (e - f) = d + e - f$
 $(d - e) * \text{fps-X} = d * \text{fps-X} - e * \text{fps-X}$ $\text{fps-X} * \text{fps-X} = \text{fps-X} \wedge 2$ $\text{fps-X} * \text{fps-X} \wedge m = \text{fps-X} \wedge (\text{Suc } m)$
 $\text{fps-X} \wedge m * \text{fps-X} \wedge n = \text{fps-X} \wedge (m + n)$
 $\langle \text{proof} \rangle$

lemma fps-divide-1 : $(a :: 'a :: \text{field fps}) / 1 = a$ $\langle \text{proof} \rangle$

lemmas $\text{fps-of-poly-code-post}$ [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

definition (in term-syntax)

$\text{valterm-ratfps} ::$

$'a :: \{\text{field}, \text{factorial-ring-gcd}, \text{typerep}\} \text{poly} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term}) \Rightarrow$

$'a \text{poly} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term}) \Rightarrow 'a \text{ratfps} \times (\text{unit} \Rightarrow \text{Code-Evaluation.term})$

where

[code-unfold]: $\text{valterm-ratfps } k \ l =$

$\text{Code-Evaluation.valtermify } (/) \ \{\cdot\}$

$(\text{Code-Evaluation.valtermify } \text{ratfps-of-poly } \{\cdot\} \ k) \ \{\cdot\}$

$(\text{Code-Evaluation.valtermify } \text{ratfps-of-poly } \{\cdot\} \ l)$

notation fcomp (infixl $\circ >$ 60)

notation scomp (infixl $\circ \rightarrow$ 60)

instantiation $\text{ratfps} :: (\{\text{field}, \text{factorial-ring-gcd}, \text{random}\}) \ \text{random}$

begin

definition

```

Quickcheck-Random.random i =
  Quickcheck-Random.random i ◦→ (λnum::'a poly × (unit ⇒ term)).
  Quickcheck-Random.random i ◦→ (λdenom::'a poly × (unit ⇒ term)).
  Pair (let denom = (if fst denom = 0 then Code-Evaluation.valtermify 1 else
denom)
      in valterm-ratfps num denom)))

```

instance ⟨proof⟩**end****no-notation** *fcomp* (**infixl** ◦> 60)**no-notation** *scomp* (**infixl** ◦→ 60)**instantiation** *ratfps* :: ({field,factorial-ring-gcd,exhaustive}) *exhaustive*
begin**definition**

```

exhaustive-ratfps f d =
  Quickcheck-Exhaustive.exhaustive (λnum.
  Quickcheck-Exhaustive.exhaustive (λdenom. f (
  let denom = if denom = 0 then 1 else denom
  in ratfps-of-poly num / ratfps-of-poly denom)) d) d

```

instance ⟨proof⟩**end****instantiation** *ratfps* :: ({field,factorial-ring-gcd,full-exhaustive}) *full-exhaustive*
begin**definition**

```

full-exhaustive-ratfps f d =
  Quickcheck-Exhaustive.full-exhaustive (λnum::'a poly × (unit ⇒ term).
  Quickcheck-Exhaustive.full-exhaustive (λdenom::'a poly × (unit ⇒ term).
  f (let denom = if fst denom = 0 then Code-Evaluation.valtermify 1 else
denom
  in valterm-ratfps num denom)) d) d

```

instance ⟨proof⟩**end****quickcheck-generator** *fps constructors: fps-of-ratfps***end**

2 Falling factorial as a polynomial

theory *Pochhammer-Polynomials*

imports

Complex-Main

HOL-Library.Stirling

HOL-Computational-Algebra.Polynomial

begin

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

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

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

<proof>

lemma *coeff-pochhammer-poly*: *coeff* (*pochhammer-poly* n) $k = \text{of-nat } (\text{stirling } n \ k)$

<proof>

lemma *degree-pochhammer-poly* [*simp*]: *degree* (*pochhammer-poly* n) = n

<proof>

lemma *pochhammer-poly-0* [*simp*]: *pochhammer-poly* $0 = 1$

<proof>

lemma *pochhammer-poly-Suc*: *pochhammer-poly* (*Suc* n) = $[\text{of-nat } n, 1:] * \text{pochhammer-poly } n$

<proof>

lemma *pochhammer-poly-altdef*: *pochhammer-poly* $n = (\prod_{i < n}. [\text{of-nat } i, 1:])$

<proof>

lemma *eval-pochhammer-poly*: *poly* (*pochhammer-poly* n) $k = \text{pochhammer } k \ n$

<proof>

lemma *pochhammer-poly-Suc'*:

pochhammer-poly (*Suc* n) = *pCons* 0 (*pcompose* (*pochhammer-poly* n) $[\text{of-nat } n, 1:]$)

<proof>

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*)
<*proof*>

lemma *zip-with-altdef*: *zip-with* *f* *xs* *ys* = *map* ($\lambda(x,y). f\ x\ y$) (*zip* *xs* *ys*)
<*proof*>

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

lemma *take-zip-with*: *take* *n* (*zip-with* *f* *xs* *ys*) = *zip-with* *f* (*take* *n* *xs*) (*take* *n* *ys*)
<*proof*>

lemma *drop-zip-with*: *drop* *n* (*zip-with* *f* *xs* *ys*) = *zip-with* *f* (*drop* *n* *xs*) (*drop* *n* *ys*)
<*proof*>

lemma *map-zip-with*: *map* *f* (*zip-with* *g* *xs* *ys*) = *zip-with* ($\lambda x\ y. f\ (g\ x\ y)$) *xs* *ys*
<*proof*>

lemma *zip-with-map*: *zip-with* *f* (*map* *g* *xs*) (*map* *h* *ys*) = *zip-with* ($\lambda x\ y. f\ (g\ x)\ (h\ y)$) *xs* *ys*
<*proof*>

lemma *zip-with-map-left*: *zip-with* *f* (*map* *g* *xs*) *ys* = *zip-with* ($\lambda x\ y. f\ (g\ x)\ y$) *xs* *ys*
<*proof*>

lemma *zip-with-map-right*: *zip-with* *f* *xs* (*map* *g* *ys*) = *zip-with* ($\lambda x\ y. f\ x\ (g\ y)$) *xs* *ys*
<*proof*>

lemma *zip-with-swap*: *zip-with* ($\lambda x\ y. f\ y\ x$) *xs* *ys* = *zip-with* *f* *ys* *xs*
<*proof*>

lemma *set-zip-with*: *set* (*zip-with* *f* *xs* *ys*) = ($\lambda(x,y). f\ x\ y$) ‘ *set* (*zip* *xs* *ys*)
<*proof*>

lemma *zip-with-Pair*: *zip-with* *Pair* (*xs* :: 'a list) (*ys* :: 'b list) = *zip* *xs* *ys*
<*proof*>

lemma *zip-with-altdef*':

$zip\text{-}with\ f\ xs\ ys = [f\ (xs!i)\ (ys!i).\ i \leftarrow [0..\lt\ min\ (length\ xs)\ (length\ ys)]]$
<proof>

lemma *zip-altdef*: $zip\ xs\ ys = [(xs!i,\ ys!i).\ i \leftarrow [0..\lt\ min\ (length\ xs)\ (length\ ys)]]$
<proof>

lemma *card-poly-roots-bound*:

fixes $p :: 'a :: \{comm\text{-}ring\text{-}1,\ ring\text{-}no\text{-}zero\text{-}divisors\}$ *poly*
assumes $p \neq 0$
shows $card\ \{x.\ poly\ p\ x = 0\} \leq degree\ p$
<proof>

lemma *poly-eqI-degree*:

fixes $p\ q :: 'a :: \{comm\text{-}ring\text{-}1,\ ring\text{-}no\text{-}zero\text{-}divisors\}$ *poly*
assumes $\bigwedge x.\ x \in A \implies poly\ p\ x = poly\ q\ x$
assumes $card\ A > degree\ p\ card\ A > degree\ q$
shows $p = q$
<proof>

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

fixes $p :: 'a :: idom\ poly$
assumes $P\ 0 \bigwedge p.\ (\bigwedge x.\ poly\ p\ x \neq 0) \implies P\ p$
 $\bigwedge p\ x\ n.\ n > 0 \implies poly\ p\ x \neq 0 \implies P\ p \implies P\ ([: -x,\ 1:] ^ n * p)$
shows $P\ p$
<proof>

lemma *complex-poly-decompose*:

$smult\ (lead\text{-}coeff\ p)\ (\prod z.\ poly\ p\ z = 0.\ [: -z,\ 1:] ^ order\ z\ p) = (p :: complex\ poly)$
<proof>

lemma *normalize-field*:

$normalize\ (x :: 'a :: \{normalization\text{-}semidom,\ field\}) = (if\ x = 0\ then\ 0\ else\ 1)$
<proof>

lemma *unit-factor-field* [*simp*]:

$unit\text{-}factor\ (x :: 'a :: \{normalization\text{-}semidom,\ field\}) = x$
<proof>

lemma *coprime-linear-poly*:

fixes $c :: 'a :: \{field,\ factorial\text{-}ring\text{-}gcd\}$
assumes $c \neq c'$
shows $coprime\ [:c,\ 1:]\ [:c',\ 1:]$
<proof>

```

lemma coprime-linear-poly':
  fixes c :: 'a :: {field,factorial-ring-gcd,normalization-euclidean-semiring}
  assumes c ≠ c' c ≠ 0 c' ≠ 0
  shows coprime [1,c] [1,c']
  ⟨proof⟩

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) ⇒ 'a list ⇒ 'a list where
  decompose x [] = []
| decompose x [y] = [x]
| decompose x (y#ys) =
  (case bezout-coefficients y (prod-list ys) of
    (a, b) ⇒ (b*x) # decompose (a*x) ys)

```

```

lemma decompose-rec:
  ys ≠ [] ⇒ decompose x (y#ys) =
    (case bezout-coefficients y (prod-list ys) of
      (a, b) ⇒ (b*x) # decompose (a*x) ys)
  ⟨proof⟩

```

lemma *length-decompose* [simp]: $\text{length } (\text{decompose } x \text{ } ys) = \text{length } ys$
 ⟨proof⟩

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

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

lemma *decompose-code* [code]:
 $\text{decompose } x \text{ } ys = \text{decompose}' \ x \text{ } ys \ (\text{tl } (\text{rev } (\text{decompose-aux } 1 \text{ } (\text{rev } ys))))$
 ⟨proof⟩

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]: $\text{length } (\text{snd } (\text{normalise-decomp } x \text{ } y \text{ } n)) = n$
 ⟨proof⟩

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]:

$length (snd (partial-fraction-decomposition x ys)) = length ys$
 ⟨proof⟩

lemma *length-pfd2* [simp]:

$i < length ys \implies length (snd (partial-fraction-decomposition x ys) ! i) = snd (ys ! i) + 1$
 ⟨proof⟩

lemma *size-normalise-decomp*:

$a \in set (snd (normalise-decomp x y n)) \implies y \neq 0 \implies euclidean-size a < euclidean-size y$
 ⟨proof⟩

lemma *size-partial-fraction-decomposition*:

$i < length xs \implies fst (xs ! i) \neq 0 \implies x \in set (snd (partial-fraction-decomposition y xs) ! i)$
 $\implies euclidean-size x < euclidean-size (fst (xs ! i))$
 ⟨proof⟩

A homomorphism φ 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 φ the evaluation homomorphism. Or one could let $S = K[[X]]$ (the ring of formal power series) and φ 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*:

$lift (a ^ n) = lift a ^ n$

⟨proof⟩

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

$\bigwedge y. y \in set ys \implies is-unit (lift y)$

shows $(\sum i < length ys. lift (decompose x ys ! i) div lift (ys ! i)) = lift x div lift (prod-list ys)$

<proof>

lemma *normalise-decomp*:

fixes $x\ y :: 'a$ **and** $n :: nat$

assumes *is-unit* (*lift* y)

defines $xs \equiv snd$ (*normalise-decomp* $x\ y\ n$)

shows $lift$ (fst (*normalise-decomp* $x\ y\ n$)) + $(\sum_{i < n. from-decomp\ (xs!i)\ y\ (n-i)) =$
 $lift\ x\ div\ lift\ y\ ^\ n$

<proof>

lemma *lift-prod-list*: $lift$ (*prod-list* xs) = *prod-list* (*map* *lift* xs)

<proof>

lemma *lift-sum*: $lift$ (*sum* $f\ A$) = *sum* ($\lambda x. lift$ ($f\ x$)) A

<proof>

lemma *partial-fraction-decomposition*:

fixes $ys :: ('a \times nat)$ *list*

defines $ys' \equiv map$ ($\lambda(x,n). x\ ^\ Suc\ n$) $ys :: 'a$ *list*

assumes *unit*: $\bigwedge y. y \in fst$ ' *set* $ys \implies is-unit$ (*lift* y)

assumes *coprime*: *pairwise coprime* (*set* ys')

assumes *distinct*: *distinct* ys'

assumes *partial-fraction-decomposition* $x\ ys = (a, zs)$

shows $lift\ a + (\sum_{i < length\ ys. \sum_{j \leq snd\ (ys!i). from-decomp\ (zs!i!j)\ (fst\ (ys!i))\ (snd\ (ys!i)+1 - j)) =$
 $lift\ x\ div\ lift\ (prod-list\ ys')$

<proof>

end

4.2 Specific results for polynomials

definition *divmod-field-poly* $:: 'a :: field\ poly \Rightarrow 'a\ poly \Rightarrow 'a\ poly \times 'a\ poly$ **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)))

<proof>

definition *normalise-decomp-poly* $:: 'a :: field\ poly \Rightarrow 'a\ poly \Rightarrow nat \Rightarrow 'a\ poly \times$

'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) = divmod-field-poly x y;
(z, rs) = normalise-decomp-poly x' y n
in (z, r # rs))
⟨proof⟩

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 (λ(c,n). [:1,-c:] ^ Suc n) cs) ! i) [:1,-c:] (n+1).
i ← [0..<length cs]]
in (sum-list (map fst zs), map (map (λp. coeff p 0) ∘ snd) zs)))

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

poly-pfd-simple x cs =
(if cs = [] then (x, []) else
let zs = zip-with (λ(c,n) decomp. normalise-decomp-poly decomp [:1,-c:]
(n+1))
cs (decompose x (map (λ(c,n). [:1,-c:] ^ Suc n) cs))
in (sum-list (map fst zs), map (map (λp. coeff p 0) ∘ snd) zs))
⟨proof⟩

lemma fst-poly-pfd-simple:

fst (poly-pfd-simple x cs) =
fst (partial-fraction-decomposition x (map (λ(c,n). ([:1,-c:],n)) cs))
⟨proof⟩

lemma const-polyI: degree p = 0 ⇒ [:coeff p 0:] = p

⟨proof⟩

lemma snd-poly-pfd-simple:

map (map (λc. [:c :: 'a :: {field,factorial-ring-gcd,normalization-euclidean-semiring}:]))
(snd (poly-pfd-simple x cs)) =
(snd (partial-fraction-decomposition x (map (λ(c,n). ([:1,-c:],n)) cs)))
⟨proof⟩

lemma poly-pfd-simple:

partial-fraction-decomposition x (map (λ(c,n). ([:1,-c:],n)) cs) =
(fst (poly-pfd-simple x cs), map (map (λc. [:c:])) (snd (poly-pfd-simple x
cs)))
⟨proof⟩

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 = $(\lambda(a, cs). \text{Polynomial.smult } a (\prod (c, n) \leftarrow cs. [:-c, 1:] \wedge \text{Suc } n))$

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 = $(\lambda(a, cs). \text{Polynomial.smult } a (\prod (c, n) \leftarrow cs. [:1, -c:] \wedge \text{Suc } n))$

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

is-factorization-of *fctrs* *p* =

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

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

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

$(\text{interp-alt-factorization } fctrs = p \wedge 0 \notin \text{set } (\text{map } fst (\text{snd } fctrs)) \wedge \text{distinct } (\text{map } fst (\text{snd } fctrs)))$

Regular and alternative factorisations are related by reflecting the polynomial.

lemma *interp-factorization-reflect*:

assumes $(0 :: 'a :: idom) \notin \text{fst } ' \text{set } (\text{snd } fctrs)$

shows $\text{reflect-poly } (\text{interp-factorization } fctrs) = \text{interp-alt-factorization } fctrs$
 $\langle \text{proof} \rangle$

lemma *interp-alt-factorization-reflect*:

assumes $(0 :: 'a :: idom) \notin \text{fst } ' \text{set } (\text{snd } fctrs)$

shows $\text{reflect-poly } (\text{interp-alt-factorization } fctrs) = \text{interp-factorization } fctrs$
 $\langle \text{proof} \rangle$

lemma *coeff-0-interp-factorization*:

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

<proof>

lemma *reflect-factorization*:

assumes $\text{coeff } p \ 0 \neq (0::'a::\text{idom})$

assumes *is-factorization-of fctrs* p

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

<proof>

lemma *reflect-factorization'*:

assumes $\text{coeff } p \ 0 \neq (0::'a::\text{idom})$

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

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

<proof>

lemma *zero-in-factorization-iff*:

assumes *is-factorization-of fctrs* p

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

<proof>

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

<proof>

lemma *is-factorization-of-roots*:

fixes $a :: 'a :: \text{idom}$

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

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

<proof>

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

<proof>

lemma *order-prod*:

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

assumes $\bigwedge x \ y. x \in A \implies y \in A \implies x \neq y \implies \text{coprime } (f \ x) \ (f \ y)$

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

<proof>

lemma *is-factorization-of-order*:

fixes $p :: 'a :: \{\text{field}, \text{factorial-ring-gcd}\} \ \text{poly}$

assumes $p \neq 0$

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

assumes $(c, n) \in \text{set } \text{fctrs}$

shows $\text{order } c \ p = \text{Suc } n$

<proof>

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

lemma *complex-factorization-exists*:

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

<proof>

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

<proof>

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-nat (stirling n (k+1))) / (fact (n - 1)). k ← [0..<n]]*

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

fixes *c :: 'a :: field-char-0*

assumes *n: n > 0*

shows *fps-const d / ((1 - fps-const (c :: 'a :: field-char-0) * fps-X) ^ n) =*

*Abs-fps (λk. poly (inverse-irred-power-poly d n) (of-nat k) * c^k) (is ?lhs*

= ?rhs)

<proof>

include *fps-notation*

<proof>

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

coeffs (inverse-irred-power-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>

lemma *solve-rat-fps-aux:*

fixes $p :: 'a :: \{\text{field-char-0, factorial-ring-gcd, normalization-euclidean-semiring}\}$
poly and $cs :: ('a \times \text{nat}) \text{ list}$
assumes $\text{distinct: distinct (map fst cs)}$
assumes $\text{azs: (a, zs) = poly-pfd-simple p cs}$
assumes $\text{nz: } 0 \notin \text{fst ' set cs}$
shows $\text{fps-of-poly p / fps-of-poly } (\prod (c,n) \leftarrow cs. [:1, -c:] \wedge \text{Suc n}) =$
 $\text{Abs-fps } (\lambda k. \text{coeff a k} + (\sum i < \text{length cs. poly } (\sum j \leq \text{snd (cs ! i)}. \text{inverse-irred-power-poly (zs ! i ! j) (snd (cs ! i) + 1 - j))))$
 $(\text{of-nat k}) * (\text{fst (cs ! i)}) \wedge k))$ (**is** $- = ?\text{rhs}$)
 $\langle \text{proof} \rangle$

definition solve-factored-ratfps ::
 $('a :: \{\text{field-char-0, factorial-ring-gcd, normalization-euclidean-semiring}\}) \text{ poly} \Rightarrow$
 $('a \times \text{nat}) \text{ list} \Rightarrow 'a \text{ poly} \times ('a \text{ poly} \times 'a) \text{ list}$ **where**
 $\text{solve-factored-ratfps p cs} = (\text{let } n = \text{length cs} \text{ in case poly-pfd-simple p cs of (a, zs) } \Rightarrow$
 $(a, \text{zip-with } (\lambda zs (c,n). ((\sum (z,j) \leftarrow \text{zip zs } [0..<\text{Suc } n]. \text{inverse-irred-power-poly z (n + 1 - j)), c)) \text{ zs cs}))$

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

lemma length-nth-snd-poly-pfd-simple [simp]:
 $i < \text{length cs} \implies \text{length (snd (poly-pfd-simple p cs) ! i)} = \text{snd (cs ! i)} + 1$
 $\langle \text{proof} \rangle$

lemma solve-factored-ratfps-roots:
 $\text{map snd (snd (solve-factored-ratfps p cs))} = \text{map fst cs}$
 $\langle \text{proof} \rangle$

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

lemma solve-factored-ratfps:
fixes $p :: 'a :: \{\text{field-char-0, factorial-ring-gcd, normalization-euclidean-semiring}\}$
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 $\text{fps-of-poly p / fps-of-poly } (\prod (c,n) \leftarrow cs. [:1, -c:] \wedge \text{Suc n}) =$
 $\text{Abs-fps (interp-ratfps-solution (solve-factored-ratfps p cs))}$ (**is** $?\text{lhs} = ?\text{rhs}$)
 $\langle \text{proof} \rangle$

definition solve-factored-ratfps' where

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

lemma *solve-factored-ratfps'*:

assumes *is-alt-factorization-of fctrs* $q \neq 0$

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

<proof>

lemma *degree-Poly-eq*:

assumes $xs = [] \vee \text{last } xs \neq 0$

shows $\text{degree} (\text{Poly } xs) = \text{length } xs - 1$

<proof>

lemma *degree-Poly'*: $\text{degree} (\text{Poly } xs) \leq \text{length } xs - 1$

<proof>

lemma *degree-inverse-irred-power-poly-le*:

$\text{degree} (\text{inverse-irred-power-poly } c \ n) \leq n - 1$

<proof>

lemma *degree-inverse-irred-power-poly*:

assumes $c \neq 0$

shows $\text{degree} (\text{inverse-irred-power-poly } c \ n) = n - 1$

<proof>

lemma *reflect-poly-0-iff [simp]*: $\text{reflect-poly } p = 0 \iff p = 0$

<proof>

lemma *degree-sum-list-le*: $(\bigwedge p. p \in \text{set } ps \implies \text{degree } p \leq T) \implies \text{degree} (\text{sum-list } ps) \leq T$

<proof>

theorem *ratfps-closed-form-exists*:

fixes $q :: \text{complex poly}$

assumes $\text{nz}: \text{coeff } q \ 0 \neq 0$

defines $q' \equiv \text{reflect-poly } q$

obtains $r \ rs$

where $\bigwedge n. \text{fps-nth} (\text{fps-of-poly } p / \text{fps-of-poly } q) \ n = \text{coeff } r \ n + (\sum c \mid \text{poly } q' \ c = 0. \text{poly} (rs \ c) (\text{of-nat } n) * c \ ^n)$

and $\bigwedge z. \text{poly } q' \ z = 0 \implies \text{degree} (rs \ z) \leq \text{order } z \ q' - 1$

<proof>

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)

<proof>

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

<proof>

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

<proof>

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

<proof>

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*]])
⟨*proof*⟩

lemma *lhr-fps-aux*:
fixes *f* :: *nat* ⇒ '*a* :: *field*
assumes $\bigwedge n. n \geq m \implies (\sum_{k \leq N. c k * f (n + k)} = 0$
assumes *cN*: *c N* ≠ 0
defines *p* ≡ *Poly* [*c* (*N* - *k*). *k* ← [0..*Suc N*]]
defines *q* ≡ *Poly* [($\sum_{i \leq \min N k. c (N - i) * f (k - i)}$). *k* ← [0..*N+m*]]
shows *Abs-fps f* = *fps-of-poly q* / *fps-of-poly p*
⟨*proof*⟩
include *fps-notation*
⟨*proof*⟩

lemma *lhr-fps*:
fixes *f* :: *nat* ⇒ '*a* :: *field* **and** *cs* :: '*a* *list*
defines *N* ≡ *length cs* - 1
assumes *cs*: *cs* ≠ []
assumes $\bigwedge n. n \geq m \implies (\sum_{k \leq N. cs ! k * f (n + k)} = 0$
assumes *cN*: *last cs* ≠ 0
shows *Abs-fps f* = *fps-of-poly* (*lhr-fps-numerator* *m cs f*) /
fps-of-poly (*lhr-fps-denominator* *cs*)
⟨*proof*⟩

fun *lhr* **where**
lhr cs fs n =
(if (*cs* :: '*a* :: *field list*) = [] ∨ *last cs* = 0 ∨ *length fs* < *length cs* - 1 then
undefined else
(if *n* < *length fs* then *fs* ! *n* else
($\sum_{k < \text{length } cs - 1. cs ! k * \text{lhr } cs fs (n + 1 - \text{length } cs + k)}$) / -*last*
cs))

declare *lhr.simps* [*simp del*]

lemma *lhr-rec*:
assumes *cs* ≠ [] *last cs* ≠ 0 *length fs* ≥ *length cs* - 1 *n* ≥ *length fs*
shows ($\sum_{k < \text{length } cs. cs ! k * \text{lhr } cs fs (n + 1 - \text{length } cs + k)}$) = 0
⟨*proof*⟩

lemma *lhrI*:
assumes *cs* ≠ [] *last cs* ≠ 0 *length fs* ≥ *length cs* - 1
assumes $\bigwedge n. n < \text{length } fs \implies f n = fs ! n$
assumes $\bigwedge n. n \geq \text{length } fs \implies (\sum_{k < \text{length } cs. cs ! k * f (n + 1 - \text{length } cs + k)} = 0$
shows *f n* = *lhr cs fs n*

<proof>

locale *linear-homogenous-recurrence* =
 fixes $f :: \text{nat} \Rightarrow 'a :: \text{comm-semiring-0}$ **and** $cs\ fs :: 'a\ \text{list}$
 assumes $\text{base}: n < \text{length}\ fs \implies f\ n = fs\ !\ n$
 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** $\text{enough-base}: \text{length}\ fs + 1 \geq \text{length}\ cs$
 assumes $\text{rec}: n \geq \text{length}\ fs - \text{length}\ cs \implies (\sum_{k < \text{length}\ cs} cs\ !\ k * f\ (n + k)) = 0$
begin

lemma *lhr-fps-numerator-altdef*:
 $\text{lhr-fps-numerator}\ (\text{length}\ fs + 1 - \text{length}\ cs)\ cs\ f =$
 $\text{lhr-fps-numerator}\ (\text{length}\ fs + 1 - \text{length}\ cs)\ cs\ (!)\ fs$
<proof>

end

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

definition
 $\text{lhr-fps}\ as\ fs = (\text{let}\ m = \text{length}\ fs + 1 - \text{length}\ as;$
 $p = \text{lhr-fps-numerator}\ m\ as\ (\lambda n. fs\ !\ n);$
 $q = \text{lr-fps-denominator}\ as$
 in $\text{ratfps-of-poly}\ p / \text{ratfps-of-poly}\ q)$

lemma *lhr-fps-correct*:
 fixes $f :: \text{nat} \Rightarrow 'a :: \{\text{field-char-0}, \text{factorial-ring-gcd}\}$
 assumes *linear-homogenous-recurrence* $f\ cs\ fs$
 shows $\text{fps-of-ratfps}\ (\text{lhr-fps}\ cs\ fs) = \text{Abs-fps}\ f$
<proof>

end

9 Eulerian polynomials

theory *Eulerian-Polynomials*
imports
 Complex-Main
 HOL-Computational-Algebra.Computational-Algebra
 HOL-Library.Stirling

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*⟩

lemma *eulerian-poly'*:

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

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* :: *field poly*)) / (1 - *fps-X*) \wedge
(k + 1)
⟨*proof*⟩

definition *fps-monom-poly* :: '*a* :: *field* \Rightarrow *nat* \Rightarrow '*a poly*

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

primrec *fps-monom-poly-aux* :: '*a* :: *field* \Rightarrow *nat* \Rightarrow '*a poly* **where**

fps-monom-poly-aux c 0 = [:c:]
| *fps-monom-poly-aux* c (*Suc* k) =
(let p = *fps-monom-poly-aux* c k
in [:0,1,-c:] * *pderiv* p + [:1, of-nat k * c:] * p)

lemma *fps-monom-poly-aux*:

fps-monom-poly-aux c k = *smult* c (*pcompose* (*eulerian-poly* k) [:0,c:])
⟨*proof*⟩

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

fps-monom-poly c k = (if k = 0 then 1 else *pCons* 0 (*fps-monom-poly-aux* c k))

<proof>

lemma *fps-monom-aux*:

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

lemma *fps-monom*:

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

end

10 Inhomogenous linear recurrences

theory *Linear-Inhomogenous-Recurrences*

imports

Complex-Main

Linear-Homogenous-Recurrences

Eulerian-Polynomials

RatFPS

begin

definition *lir-fps-numerator where*

$\text{lir-fps-numerator } m \ cs \ f \ g = (\text{let } N = \text{length } cs - 1 \text{ in}$
 $\text{Poly } [(\sum_{i \leq \min N \ k. cs \ ! \ (N - i) * f \ (k - i) - g \ k. k \leftarrow [0..<N+m]])$

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

$\text{coeffs } (\text{lir-fps-numerator } m \ cs \ f \ g) = (\text{let } N = \text{length } cs - 1 \text{ in}$
 $\text{strip-while } ((=) \ 0) [(\sum_{i \leq \min N \ k. cs \ ! \ (N - i) * f \ (k - i) - g \ k. k \leftarrow$
 $[0..<N+m]])$
<proof>

locale *linear-inhomogenous-recurrence* =

fixes $f \ g :: \text{nat} \Rightarrow 'a :: \text{comm-ring}$ **and** $cs \ fs :: 'a \ \text{list}$

assumes *base*: $n < \text{length } fs \implies f \ n = fs \ ! \ n$

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*]: $\text{coeff } (\text{lr-fps-denominator } cs) \ 0 = \text{last } cs$

<proof>

lemma *lir-fps-numerator-altdef*:

$\text{lir-fps-numerator } (\text{length } fs + 1 - \text{length } cs) \text{ cs } f \text{ } g =$
 $\text{lir-fps-numerator } (\text{length } fs + 1 - \text{length } cs) \text{ cs } (!) \text{ } fs) \text{ } g$
 <proof>

end

context
begin

private lemma *lir-fps-aux*:

fixes $f :: \text{nat} \Rightarrow 'a :: \text{field}$
assumes $\text{rec}: \bigwedge n. n \geq m \implies (\sum k \leq N. c \ k * f (n + k)) = g (n + N)$
assumes $cN: c \ N \neq 0$
defines $p \equiv \text{Poly } [c \ (N - k). k \leftarrow [0..< \text{Suc } N]]$
defines $q \equiv \text{Poly } [(\sum i \leq \text{min } N \ k. c \ (N - i) * f (k - i)) - g \ k. k \leftarrow [0..< N + m]]$
shows $\text{Abs-fps } f = (\text{fps-of-poly } q + \text{Abs-fps } g) / \text{fps-of-poly } p$
 <proof>
include *fps-notation*
 <proof>

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 $\text{Abs-fps } f = (\text{fps-of-poly } (\text{lir-fps-numerator } m \ cs \ f \ g) + \text{Abs-fps } g) /$
 $\text{fps-of-poly } (\text{lir-fps-denominator } cs)$
 <proof>

end

type-synonym $'a \ \text{polyexp} = ('a \times \text{nat} \times 'a) \ \text{list}$

definition *eval-polyexp* :: $('a :: \text{semiring-1}) \ \text{polyexp} \Rightarrow \text{nat} \Rightarrow 'a$ **where**
 $\text{eval-polyexp } xs = (\lambda n. \sum (a,k,b) \leftarrow xs. a * \text{of-nat } n \ ^k * b \ ^n)$

lemma *eval-polyexp-Nil* [*simp*]: $\text{eval-polyexp } [] = (\lambda -. 0)$
 <proof>

lemma *eval-polyexp-Cons*:

$\text{eval-polyexp } (x \# xs) = (\lambda n. (\text{case } x \ \text{of } (a,k,b) \Rightarrow a * \text{of-nat } n \ ^k * b \ ^n) +$
 $\text{eval-polyexp } xs \ n)$
 <proof>

definition *polyexp-fps* :: $('a :: \text{field}) \ \text{polyexp} \Rightarrow 'a \ \text{fps}$ **where**

$$\text{polyexp-fps } xs = \left(\sum (a,k,b) \leftarrow xs. \text{fps-of-poly } (\text{Polynomial.smult } a \text{ (fps-monom-poly } b \text{ } k)) / (1 - \text{fps-const } b * \text{fps-X}) ^ (k + 1) \right)$$

lemma *polyexp-fps-Nil* [simp]: *polyexp-fps* [] = 0
 ⟨proof⟩

lemma *polyexp-fps-Cons*:

$$\text{polyexp-fps } (x\#xs) = (\text{case } x \text{ of } (a,k,b) \Rightarrow \text{fps-of-poly } (\text{Polynomial.smult } a \text{ (fps-monom-poly } b \text{ } k)) / (1 - \text{fps-const } b * \text{fps-X}) ^ (k + 1)) + \text{polyexp-fps } xs$$

⟨proof⟩

definition *polyexp-ratfps* :: ('a :: {field,factorial-ring-gcd}) *polyexp* ⇒ 'a *ratfps*
where

$$\text{polyexp-ratfps } xs = \left(\sum (a,k,b) \leftarrow xs. \text{ratfps-of-poly } (\text{Polynomial.smult } a \text{ (fps-monom-poly } b \text{ } k)) / \text{ratfps-of-poly } ([:1, -b:] ^ (k + 1)) \right)$$

lemma *polyexp-ratfps-Nil* [simp]: *polyexp-ratfps* [] = 0
 ⟨proof⟩

lemma *polyexp-ratfps-Cons*: *polyexp-ratfps* (x#xs) = (case x of (a,k,b) ⇒ *ratfps-of-poly* (Polynomial.smult a (fps-monom-poly b k)) / *ratfps-of-poly* ([:1, -b:] ^ (k + 1))) + *polyexp-ratfps* xs
 ⟨proof⟩

lemma *polyexp-fps*: *Abs-fps* (eval-polyexp xs) = *polyexp-fps* xs
 ⟨proof⟩

lemma *polyexp-ratfps* [simp]: *fps-of-ratfps* (*polyexp-ratfps* xs) = *polyexp-fps* xs
 ⟨proof⟩

definition *lir-fps* ::

$$'a :: \{\text{field,factorial-ring-gcd}\} \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ polyexp} \Rightarrow ('a \text{ ratfps}) \text{ option}$$

where

$$\begin{aligned} \text{lir-fps } cs \text{ } fs \text{ } g &= (\text{if } cs = [] \vee \text{length } fs < \text{length } cs - 1 \text{ then None else} \\ &\text{let } m = \text{length } fs + 1 - \text{length } cs; \\ &p = \text{lir-fps-numerator } m \text{ } cs \text{ } (\lambda n. fs ! n) \text{ (eval-polyexp } g); \\ &q = \text{lr-fps-denominator } cs \\ &\text{in Some } ((\text{ratfps-of-poly } p + \text{polyexp-ratfps } g) * \text{inverse } (\text{ratfps-of-poly } q))) \end{aligned}$$

lemma *lir-fps-correct*:

$$\begin{aligned} \text{fixes } f &:: \text{nat} \Rightarrow 'a :: \{\text{field,factorial-ring-gcd}\} \\ \text{assumes } &\text{linear-inhomogenous-recurrence } f \text{ (eval-polyexp } g) \text{ } cs \text{ } fs \\ \text{shows } &\text{map-option } \text{fps-of-ratfps} \text{ (lir-fps } cs \text{ } fs \text{ } g) = \text{Some } (\text{Abs-fps } f) \end{aligned}$$

⟨proof⟩

end

theory *Rational-FPS-Asymptotics*

imports

HOL-Library.Landau-Symbols
HOL-Analysis.Analysis
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 $\text{poly } p \sim [F] (\lambda x. \text{lead-coeff } p * x^{\text{degree } p})$

<proof>

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>

lemma *poly-bigo*:

assumes $F \leq \text{at-infinity}$ **and** $\text{degree } p \leq k$

shows $\text{poly } p \in O[F](\lambda x. x^k)$

<proof>

lemma *reflect-poly-dvdI*:

fixes $p \ q :: 'a :: \{\text{comm-semiring-1, semiring-no-zero-divisors}\}$ *poly*

assumes $p \ \text{dvd} \ q$

shows $\text{reflect-poly } p \ \text{dvd} \ \text{reflect-poly } q$

<proof>

lemma *smult-altdef*: $\text{smult } c \ p = [c:] * p$

<proof>

lemma *smult-power*: $\text{smult } (c^{\wedge} n) \ (p^{\wedge} n) = (\text{smult } c \ p)^{\wedge} n$

<proof>

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>

lemma *order-reflect-poly*:

fixes $c :: 'a :: field$

assumes $c \neq 0$ **and** $coeff\ p\ 0 \neq 0$

shows $order\ c\ (reflect\text{-}poly\ p) = order\ (1 / c)\ p$

<proof>

lemma *poly-reflect-eq-0-iff*:

$poly\ (reflect\text{-}poly\ p)\ (x :: 'a :: field) = 0 \iff p = 0 \vee x \neq 0 \wedge poly\ p\ (1 / x) = 0$

<proof>

theorem *ratfps-nth-bigo*:

fixes $q :: complex\ poly$

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>

lemma *order-power*: $p \neq 0 \implies order\ c\ (p^{\wedge} n) = n * order\ c\ p$

<proof>

lemma *same-root-imp-not-coprime*:

assumes $poly\ p\ x = 0$ **and** $poly\ q\ (x :: 'a :: \{factorial\text{-}ring\text{-}gcd\}) = 0$

shows $\neg coprime\ p\ q$

<proof>

lemma *ratfps-nth-bigo-square-free-factorization*:

fixes $p :: complex\ poly$

assumes *square-free-factorization* $q\ (b, cs)$

assumes $q \neq 0$ **and** $R > 0$

assumes *roots1*: $\bigwedge c\ l. (c, l) \in set\ cs \implies \forall x \in ball\ 0\ (1 / R). poly\ c\ x \neq 0$

assumes *roots2*: $\bigwedge c\ l. (c, l) \in set\ cs \implies l > k \implies \forall x \in sphere\ 0\ (1 / R). poly\ c\ x \neq 0$

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>

find-consts *name:Count-Complex*

term *proots-ball-card*

term *proots-sphere-card*

lemma *proots-within-card-zero-iff*:

assumes $p \neq 0$ ($0 :: 'a :: idom poly$)
shows $\text{card} (\text{proots-within } p A) = 0 \iff (\forall x \in A. \text{poly } p x \neq 0)$
 $\langle \text{proof} \rangle$

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*: $\text{list-all } (\lambda cl. \text{proots-ball-card } (fst cl) 0 (1 / R) = 0) cs$
assumes *roots2*: $\text{list-all } (\lambda cl. \text{proots-sphere-card } (fst cl) 0 (1 / R) = 0)$
 $(\text{filter } (\lambda cl. \text{snd } cl > k) cs)$
shows $\text{fps-nth } (\text{fps-of-poly } p / \text{fps-of-poly } q) \in O(\lambda n. \text{of-nat } n^k * \text{of-real } R$
 $^n)$
 $\langle \text{proof} \rangle$

definition *ratfps-has-asymptotics where*

ratfps-has-asymptotics $q k R \iff q \neq 0 \wedge R > 0 \wedge$
 $(\text{let } cs = \text{snd } (\text{yun-factorization } \text{gcd } q)$
 $\text{in } \text{list-all } (\lambda cl. \text{proots-ball-card } (fst cl) 0 (1 / R) = 0) cs \wedge$
 $\text{list-all } (\lambda cl. \text{proots-sphere-card } (fst cl) 0 (1 / R) = 0) (\text{filter } (\lambda cl. \text{snd } cl$
 $> k) cs))$

lemma *ratfps-has-asymptotics-correct*:

assumes *ratfps-has-asymptotics* $q k R$
shows $\text{fps-nth } (\text{fps-of-poly } p / \text{fps-of-poly } q) \in O(\lambda n. \text{of-nat } n^k * \text{of-real } R$
 $^n)$
 $\langle \text{proof} \rangle$

value $\text{map } (\text{fps-nth } (\text{fps-of-poly } [:0, 1:] / \text{fps-of-poly } [:1, -1, -1 :: \text{real}])) [0..<5]$

method *ratfps-bigo* = (*rule ratfps-has-asymptotics-correct*; *eval*)

lemma $\text{fps-nth } (\text{fps-of-poly } [:0, 1:] / \text{fps-of-poly } [:1, -1, -1 :: \text{complex}]) \in$
 $O(\lambda n. \text{of-nat } n^0 * \text{complex-of-real } 1.618034^n)$
 $\langle \text{proof} \rangle$

lemma $\text{fps-nth } (\text{fps-of-poly } 1 / \text{fps-of-poly } [:1, -3, 3, -1 :: \text{complex}]) \in$
 $O(\lambda n. \text{of-nat } n^2 * \text{complex-of-real } 1^n)$
 $\langle \text{proof} \rangle$

lemma $\text{fps-nth } (\text{fps-of-poly } f / \text{fps-of-poly } [:5, 4, 3, 2, 1 :: \text{complex}]) \in$
 $O(\lambda n. \text{of-nat } n^0 * \text{complex-of-real } 0.69202^n)$
 $\langle \text{proof} \rangle$

end