A Verified Solver for Linear Recurrences

Manuel Eberl

March 17, 2025

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

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* ⇒ *'a*::*zero*
  where *constant-term p* ≡ *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* ≠ *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* ≠ *0* *coeff* (*snd* (*quot-of-fract y*)) *0* ≠ *0*
  shows *coeff* (*snd* (*quot-of-fract* (*x* + *y*))) *0* ≠ *0*
⟨*proof*⟩

lemma *coeff-0-normalize-quot-nonzero* [*simp*]:
  assumes *coeff* (*snd x*) *0* ≠ *0*
  shows *coeff* (*snd* (*normalize-quot x*)) *0* ≠ *0*
⟨*proof*⟩

abbreviation *numerator* :: *'a fract* ⇒ *'a*::{*ring-gcd*,*idom-divide*,*semiring-gcd-mult-normalize*}
  where *numerator x* ≡ *fst* (*quot-of-fract x*)

abbreviation *denominator* :: *'a fract* ⇒ *'a*::{*ring-gcd*,*idom-divide*,*semiring-gcd-mult-normalize*}
  where *denominator x* ≡ *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*)) ≠ *0*
  if *constant-term* (*denominator x*) ≠ *0*
  ⟨*proof*⟩

1.2 The type of rational formal power series

typedef (**overloaded**) *'a* :: *field-gcd ratfps* =

$\{x :: 'a\ poly\ fract.\ constant\text{-}term\ (denominator\ x) \neq 0\}$
⟨*proof*⟩

setup-lifting *type-definition-ratfps*

instantiation *ratfps* :: (*field-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-gcd ratfps* ⇒ *nat* ⇒ *'a* **is**
λx n. let (a,b) = quot-of-fract x
in $(\sum i = 0..n.\ coeff\ a\ i * ratfps\text{-}nth\text{-}aux\ b\ (n - i))$ ⟨*proof*⟩

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

context
includes *lifting-syntax*
begin

lemma *RatFPS-parametric*: (*rel-prod* (=) (=) ===> (=))
  (λ(*p*,*q*). *if coeff q 0* = *0 then 0 else quot-to-fract* (*p*, *q*))
  (λ(*p*,*q*). *if coeff q 0* = *0 then 0 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-gcd*)
begin

lift-definition *quot-of-ratfps* :: ′*a ratfps* ⇒ (′*a poly* × ′*a poly*) **is**
  quot-of-fract :: ′*a poly fract* ⇒ (′*a poly* × ′*a poly*) ⟨*proof*⟩

lift-definition *quot-to-ratfps* :: (′*a poly* × ′*a poly*) ⇒ ′*a ratfps* **is**
  λ(*x*,*y*). *let* (*x*′,*y*′) = *normalize-quot* (*x*,*y*)
          in if coeff y′ *0* = *0 then 0 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* ≠ *0*
  ⟨*proof*⟩

lemma *quot-of-ratfps-quot-to-ratfps*:
  coeff (*snd x*) *0* ≠ *0* ⟹ *x* ∈ *normalized-fracts* ⟹ *quot-of-ratfps* (*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* ⇒ ′*a ratfps* **is**
  to-fract :: ′*a poly* ⇒ -
  ⟨*proof*⟩

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

quot-of-ratfps (*ratfps-of-poly* *p*) = (*p*, *1*)
⟨*proof*⟩

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*) = (*let* (*a*, *b*) = *quot-of-ratfps* *x* *in* (−*a*, *b*))
⟨*proof*⟩

lemma *plus-ratfps-code* [*code abstract*]:
quot-of-ratfps (*x* + *y*) =
(*let* (*a*,*b*) = *quot-of-ratfps* *x*; (*c*,*d*) = *quot-of-ratfps* *y*
in *normalize-quot* (*a* ∗ *d* + *b* ∗ *c*, *b* ∗ *d*))
⟨*proof*⟩

lemma *minus-ratfps-code* [*code abstract*]:
quot-of-ratfps (*x* − *y*) =
(*let* (*a*,*b*) = *quot-of-ratfps* *x*; (*c*,*d*) = *quot-of-ratfps* *y*
in *normalize-quot* (*a* ∗ *d* − *b* ∗ *c*, *b* ∗ *d*))
⟨*proof*⟩

definition *ratfps-cutoff* :: *nat* ⇒ ′*a* :: *field-gcd ratfps* ⇒ ′*a poly* **where**
ratfps-cutoff *n* *x* = *poly-of-list* (*map* (*ratfps-nth* *x*) [*0*..<*n*])

definition *ratfps-shift* :: *nat* ⇒ ′*a* :: *field-gcd ratfps* ⇒ ′*a ratfps* **where**
ratfps-shift *n* *x* = (*let* (*a*, *b*) = *quot-of-ratfps* (*x* − *ratfps-of-poly* (*ratfps-cutoff* *n* *x*))
in *quot-to-ratfps* (*poly-shift* *n* *a*, *b*))

lemma *times-ratfps-code* [*code abstract*]:
quot-of-ratfps (*x* ∗ *y*) =
(*let* (*a*,*b*) = *quot-of-ratfps* *x*; (*c*,*d*) = *quot-of-ratfps* *y*;
(*e*,*f*) = *normalize-quot* (*a*,*d*); (*g*,*h*) = *normalize-quot* (*c*,*b*)
in (*e*∗*g*, *f*∗*h*))
⟨*proof*⟩

lemma *ratfps-nth-code* [*code*]:
ratfps-nth *x* *n* =
(*let* (*a*,*b*) = *quot-of-ratfps* *x*
in $\sum$ *i* = *0*..*n*. *coeff* *a* *i* ∗ *ratfps-nth-aux* *b* (*n* − *i*))
⟨*proof*⟩

lemma *ratfps-subdegree-code* [*code*]:
ratfps-subdegree *x* = *poly-subdegree* (*fst* (*quot-of-ratfps* *x*))
⟨*proof*⟩

end

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

lift-definition *inverse-ratfps* :: *'a ratfps* ⇒ *'a ratfps* **is**
  λx. let (*a*,*b*) = *quot-of-fract x*
    in if coeff a 0 = 0 then 0 else inverse x
  ⟨*proof*⟩

lift-definition *divide-ratfps* :: *'a ratfps* ⇒ *'a ratfps* ⇒ *'a ratfps* **is**
  λf g. (*if g = 0 then 0 else*
      let n = ratfps-subdegree g; *h = ratfps-shift n g*
      in ratfps-shift n (*f* ∗ *inverse h*)) ⟨*proof*⟩

instance ⟨*proof*⟩
end

lemma *ratfps-inverse-code* [*code abstract*]:
  quot-of-ratfps (*inverse x*) =
    (*let* (*a*,*b*) = *quot-of-ratfps x*
     in if coeff a 0 = 0 then (*0*, *1*)
       else let u = unit-factor a in (*b div u*, *a div u*))
  ⟨*proof*⟩

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

definition *equal-ratfps* :: *'a ratfps* ⇒ *'a ratfps* ⇒ *bool* **where**
  [*simp*]: *equal-ratfps x y* ⟷ *x* = *y*

instance ⟨*proof*⟩

end

lemma *quot-of-fract-eq-iff* [*simp*]: *quot-of-fract x* = *quot-of-fract y* ⟷ *x* = *y*
  ⟨*proof*⟩

lemma *equal-ratfps-code* [*code*]: *HOL.equal x y* ⟷ *quot-of-ratfps x* = *quot-of-ratfps y*
  ⟨*proof*⟩

lemma *fps-of-poly-quot-normalize-quot* [*simp*]:
  fps-of-poly (*fst* (*normalize-quot x*)) / *fps-of-poly* (*snd* (*normalize-quot x*)) =
    fps-of-poly (*fst x*) / *fps-of-poly* (*snd x*)
  if (*snd x* :: *'a* :: *field-gcd poly*) ≠ *0*
⟨*proof*⟩

lemma *fps-of-poly-quot-normalize-quot'* [*simp*]:
  fps-of-poly (*fst* (*normalize-quot x*)) / *fps-of-poly* (*snd* (*normalize-quot x*)) =

fps-of-poly (*fst x*) / *fps-of-poly* (*snd x*)
if *coeff* (*snd x*) *0* ≠ (*0* :: *'a* :: *field-gcd*)
⟨*proof*⟩

lift-definition *fps-of-ratfps* :: *'a* :: *field-gcd ratfps* ⇒ *'a fps* **is**
λ*x*. *fps-of-poly* (*numerator x*) / *fps-of-poly* (*denominator x*) ⟨*proof*⟩

lemma *fps-of-ratfps-altdef*:
fps-of-ratfps x = (*case quot-of-ratfps x of* (*a*, *b*) ⇒ *fps-of-poly a* / *fps-of-poly b*)
⟨*proof*⟩

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

lemma *fps-of-ratfps-0* [*simp*]: *fps-of-ratfps 0* = *0*
⟨*proof*⟩

lemma *fps-of-ratfps-1* [*simp*]: *fps-of-ratfps 1* = *1*
⟨*proof*⟩

lemma *fps-of-ratfps-uminus* [*simp*]: *fps-of-ratfps* (−*x*) = − *fps-of-ratfps x*
⟨*proof*⟩

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

lemma *fps-of-ratfps-diff* [*simp*]: *fps-of-ratfps* (*x* − *y*) = *fps-of-ratfps x* − *fps-of-ratfps y*
⟨*proof*⟩

lemma *is-unit-div-div-commute*: *is-unit b* ⟹ *is-unit c* ⟹ *a div b div c* = *a div c div b*
⟨*proof*⟩

lemma *fps-of-ratfps-mult* [*simp*]: *fps-of-ratfps* (*x* ∗ *y*) = *fps-of-ratfps x* ∗ *fps-of-ratfps y*
⟨*proof*⟩

lemma *div-const-unit-poly*: *is-unit c* ⟹ *p div* [:*c*:] = *smult* (*1 div c*) *p*
⟨*proof*⟩

lemma *normalize-field*:
normalize (*x* :: *'a* :: {*normalization-semidom*,*field*}) = (*if x* = *0 then 0 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-syntax*
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* ≠ *0* ⟷ *ratfps-nth* *a* *0* ≠ *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* ≠ *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* = *fps-of-ratfps 0* ⟨*proof*⟩

lemma *fps-one-code* [*code*]: *1* = *fps-of-ratfps 1* ⟨*proof*⟩

lemma *fps-const-code* [*code*]: *fps-const c* = *fps-of-poly* [:*c*:] ⟨*proof*⟩

lemma *fps-of-poly-code* [*code*]: *fps-of-poly p* = *fps-of-ratfps* (*ratfps-of-poly p*) ⟨*proof*⟩

lemma *fps-X-code* [*code*]: *fps-X* = *fps-of-ratfps* (*ratfps-of-poly* [:*0*,*1*:]) ⟨*proof*⟩

lemma *fps-nth-code* [*code*]: *fps-nth* (*fps-of-ratfps x*) *n* = *ratfps-nth x n*
⟨*proof*⟩

lemma *fps-uminus-code* [*code*]: − *fps-of-ratfps x* = *fps-of-ratfps* (−*x*) ⟨*proof*⟩

lemma *fps-add-code* [*code*]: *fps-of-ratfps x* + *fps-of-ratfps y* = *fps-of-ratfps* (*x* + *y*) ⟨*proof*⟩

lemma *fps-diff-code* [*code*]: *fps-of-ratfps x* − *fps-of-ratfps y* = *fps-of-ratfps* (*x* − *y*) ⟨*proof*⟩

lemma *fps-mult-code* [*code*]: *fps-of-ratfps x* ∗ *fps-of-ratfps y* = *fps-of-ratfps* (*x* ∗ *y*) ⟨*proof*⟩

lemma *fps-inverse-code* [*code*]: *inverse* (*fps-of-ratfps x*) = *fps-of-ratfps* (*inverse x*)

⟨*proof*⟩

lemma *fps-cutoff-code* [*code*]: *fps-cutoff n* (*fps-of-ratfps x*) = *fps-of-poly* (*ratfps-cutoff n x*)
⟨*proof*⟩

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

lemma *fractrel-normalize-quot*:
fractrel p p ⟹ *fractrel q q* ⟹
fractrel (*normalize-quot p*) (*normalize-quot q*) ⟷ *fractrel p q*
⟨*proof*⟩

lemma *fps-of-ratfps-eq-iff* [*simp*]:
fps-of-ratfps p = *fps-of-ratfps q* ⟷ *p* = *q*
⟨*proof*⟩

lemma *fps-of-ratfps-eq-zero-iff* [*simp*]:
fps-of-ratfps p = *0* ⟷ *p* = *0*

⟨*proof*⟩

lemma *unit-factor-snd-quot-of-ratfps* [*simp*]:
unit-factor (*snd* (*quot-of-ratfps x*)) = *1*
⟨*proof*⟩

lemma *poly-shift-times-monom-le*:
$n \leq m \Longrightarrow$ *poly-shift n* (*monom c m* $*$ *p*) = *monom c* ($m - n$) $*$ *p*
⟨*proof*⟩

lemma *poly-shift-times-monom-ge*:
$n \geq m \Longrightarrow$ *poly-shift n* (*monom c m* $*$ *p*) = *smult c* (*poly-shift* ($n - m$) *p*)
⟨*proof*⟩

lemma *poly-shift-times-monom*:
poly-shift n (*monom c n* $*$ *p*) = *smult c p*
⟨*proof*⟩

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*⟩

lemma *monom-times-poly-shift'*:
assumes *poly-subdegree p* $\geq$ *n*
shows *monom* (*1* :: *'a* :: *comm-semiring-1*) *n* $*$ *poly-shift n p* = *p*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fps-shift-times-X-power''*: *fps-shift n* (*fps-X* ^ *n* $*$ *f* :: *'a* :: *comm-ring-1 fps*) = *f*
⟨*proof*⟩

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-syntax*
⟨*proof*⟩

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

n x)
⟨*proof*⟩

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

definition *equal-fps* :: *'a fps* ⇒ *'a fps* ⇒ *bool* **where**
[*simp*]: *equal-fps f g* ⟷ *f* = *g*

instance ⟨*proof*⟩

end

lemma *equal-fps-code* [*code*]: *HOL.equal* (*fps-of-ratfps f*) (*fps-of-ratfps g*) ⟷ *f* = *g*
⟨*proof*⟩

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

lemma *ratfps-eqI*: *fps-of-ratfps x* = *fps-of-ratfps y* ⟹ *x* = *y* ⟨*proof*⟩

instance *ratfps* :: (*field-gcd*) *algebraic-semidom*
⟨*proof*⟩

lemma *fps-of-ratfps-dvd* [*simp*]:
fps-of-ratfps x dvd fps-of-ratfps y ⟷ *x dvd y*
⟨*proof*⟩

lemma *is-unit-ratfps-iff* [*simp*]:
is-unit x ⟷ *ratfps-nth x 0* ≠ *0*
⟨*proof*⟩

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

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

definition *normalize-ratfps* :: *'a ratfps* ⇒ *'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*]:
fps-of-ratfps (*unit-factor x*) = *unit-factor* (*fps-of-ratfps x*)
⟨*proof*⟩

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

fps-of-ratfps (*normalize* x) = *normalize* (*fps-of-ratfps* x)
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *ratfps* :: (*field-gcd*) *normalization-semidom-multiplicative*
⟨*proof*⟩

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

lift-definition *modulo-ratfps* :: $'a$ *ratfps* ⇒ $'a$ *ratfps* ⇒ $'a$ *ratfps* **is**
$\lambda f\ g.$ *if* $g = 0$ *then* f *else*
let n = *ratfps-subdegree* g; h = *ratfps-shift* n g
in *ratfps-of-poly* (*ratfps-cutoff* n (f ∗ *inverse* h)) ∗ h ⟨*proof*⟩

lemma *fps-of-ratfps-mod* [*simp*]:
fps-of-ratfps (f *mod* g :: $'a$ *ratfps*) = *fps-of-ratfps* f *mod* *fps-of-ratfps* g
⟨*proof*⟩

instance
⟨*proof*⟩

end

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

definition *euclidean-size-ratfps* :: $'a$ *ratfps* ⇒ *nat* **where**
euclidean-size-ratfps x = (*if* $x = 0$ *then* 0 *else* 2 ^ *ratfps-subdegree* x)

lemma *fps-of-ratfps-euclidean-size* [*simp*]:
euclidean-size x = *euclidean-size* (*fps-of-ratfps* x)
⟨*proof*⟩

instance ⟨*proof*⟩

end

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

instance
⟨*proof*⟩

end

lemma *quot-of-ratfps-eq-iff* [*simp*]: *quot-of-ratfps* x = *quot-of-ratfps* $y \longleftrightarrow x = y$
⟨*proof*⟩

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

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

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

instance *ratfps* :: (*field-gcd*) *normalization-euclidean-semiring* ⟨*proof*⟩

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

definition *gcd-ratfps* = (*Euclidean-Algorithm.gcd* :: *'a ratfps* ⇒ -)
definition *lcm-ratfps* = (*Euclidean-Algorithm.lcm* :: *'a ratfps* ⇒ -)
definition *Gcd-ratfps* = (*Euclidean-Algorithm.Gcd* :: *'a ratfps set* ⇒ -)
definition *Lcm-ratfps* = (*Euclidean-Algorithm.Lcm*:: *'a ratfps set* ⇒ -)

instance ⟨*proof*⟩
end

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

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

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

lemma *ratfps-gcd-altdef*: *gcd* (*f* :: *'a* :: *field-gcd ratfps*) *g* =
(*if* $f = 0 \wedge g = 0$ *then 0 else*
if $f = 0$ *then ratfps-of-poly* (*monom 1* (*ratfps-subdegree g*)) *else*
if $g = 0$ *then ratfps-of-poly* (*monom 1* (*ratfps-subdegree f*)) *else*
ratfps-of-poly (*monom 1* (*min* (*ratfps-subdegree f*) (*ratfps-subdegree g*))))
⟨*proof*⟩

lemma *ratfps-lcm*:

assumes [*simp*]: $f \neq 0$ $g \neq 0$
shows *lcm f g = ratfps-of-poly (monom 1 (max (ratfps-subdegree f) (ratfps-subdegree g)))*
⟨*proof*⟩

lemma *ratfps-lcm-altdef*: *lcm (f :: 'a :: field-gcd ratfps) g =*
(if f = 0 ∨ g = 0 then 0 else
ratfps-of-poly (monom 1 (max (ratfps-subdegree f) (ratfps-subdegree g))))
⟨*proof*⟩

lemma *ratfps-Gcd*:
assumes *A − {0} ≠ {}*
shows *Gcd A = ratfps-of-poly (monom 1 (INF f∈A−{0}. ratfps-subdegree f))*
⟨*proof*⟩

lemma *ratfps-Gcd-altdef*: *Gcd (A :: 'a :: field-gcd ratfps set) =*
(if A ⊆ {0} then 0 else ratfps-of-poly (monom 1 (INF f∈A−{0}. ratfps-subdegree f)))
⟨*proof*⟩

lemma *ratfps-Lcm*:
assumes *A ≠ {} 0 ∉ A bdd-above (ratfps-subdegree'A)*
shows *Lcm A = ratfps-of-poly (monom 1 (SUP f∈A. ratfps-subdegree f))*
⟨*proof*⟩

lemma *ratfps-Lcm-altdef*:
Lcm (A :: 'a :: field-gcd ratfps set) =
(if 0 ∈ A ∨ ¬bdd-above (ratfps-subdegree'A) then 0 else
if A = {} then 1 else ratfps-of-poly (monom 1 (SUP f∈A. ratfps-subdegree f)))
⟨*proof*⟩

lemma *fps-of-ratfps-quot-to-ratfps*:
coeff y 0 ≠ 0 ⟹ fps-of-ratfps (quot-to-ratfps (x,y)) = fps-of-poly x / fps-of-poly y
⟨*proof*⟩

lemma *fps-of-ratfps-quot-to-ratfps-code-post1*:
fps-of-ratfps (quot-to-ratfps (x,pCons 1 y)) = fps-of-poly x / fps-of-poly (pCons 1 y)
fps-of-ratfps (quot-to-ratfps (x,pCons (−1) y)) = fps-of-poly x / fps-of-poly (pCons (−1) y)
⟨*proof*⟩

lemma *fps-of-ratfps-quot-to-ratfps-code-post2*:
fps-of-ratfps (quot-to-ratfps (x'::'a::{field-char-0,field-gcd} poly,pCons (numeral n) y')) =
fps-of-poly x' / fps-of-poly (pCons (numeral n) y')
fps-of-ratfps (quot-to-ratfps (x'::'a::{field-char-0,field-gcd} poly,pCons (−numeral n) y')) =

fps-of-poly x′ / fps-of-poly (pCons (−numeral n) y′)
⟨*proof*⟩

lemmas *fps-of-ratfps-quot-to-ratfps-code-post* [*code-post*] =
fps-of-ratfps-quot-to-ratfps-code-post1
fps-of-ratfps-quot-to-ratfps-code-post2

lemma *fps-dehorner*:
fixes *a b c* :: ′*a* :: *semiring-1 fps* **and** *d e f* :: ′*b* :: *ring-1 fps*
shows
$(b + c) * fps\text{-}X = b * fps\text{-}X + c * fps\text{-}X$ $(a * fps\text{-}X) * fps\text{-}X = a * fps\text{-}X \mathbin{\hat{}} 2$
$a * fps\text{-}X \mathbin{\hat{}} m * fps\text{-}X = a * fps\text{-}X \mathbin{\hat{}} (Suc\ m)$ $a * fps\text{-}X * fps\text{-}X \mathbin{\hat{}} m = a * fps\text{-}X \mathbin{\hat{}} (Suc\ m)$
$a * fps\text{-}X\hat{}m * fps\text{-}X\hat{}n = a * fps\text{-}X\hat{}(m+n)$ $a + (b + c) = a + b + c$ $a * 1 = a$ $1 * a = a$
$d + - e = d - e$ $(-d) * e = - (d * e)$ $d + (e - f) = d + e - f$
$(d - e) * fps\text{-}X = d * fps\text{-}X - e * fps\text{-}X$ $fps\text{-}X * fps\text{-}X = fps\text{-}X\hat{}2$ $fps\text{-}X * fps\text{-}X\hat{}m = fps\text{-}X\hat{}(Suc\ m)$ $fps\text{-}X\hat{}m * fps\text{-}X = fps\text{-}X\hat{}Suc\ m$
$fps\text{-}X\hat{}m * fps\text{-}X\hat{}n = fps\text{-}X\hat{}(m + n)$
⟨*proof*⟩

lemma *fps-divide-1*: (*a* :: ′*a* :: *field fps*) / *1* = *a* ⟨*proof*⟩

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* × (*unit* ⇒ *Code-Evaluation.term*) ⇒
′*a poly* × (*unit* ⇒ *Code-Evaluation.term*) ⇒ ′*a ratfps* × (*unit* ⇒ *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* **and** *term-syntax*

begin

definition
Quickcheck-Random.random i =
Quickcheck-Random.random i ∘→ (λ*num*::′*a poly* × (*unit* ⇒ *term*).
Quickcheck-Random.random i ∘→ (λ*denom*::′*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

end

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

definition
exhaustive-ratfps f d =
Quickcheck-Exhaustive.exhaustive (λ*num*.
Quickcheck-Exhaustive.exhaustive (λ*denom*. *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-gcd*,*full-exhaustive*}) *full-exhaustive*
begin

definition
full-exhaustive-ratfps f d =
Quickcheck-Exhaustive.full-exhaustive (λ*num*::′*a poly* × (*unit* ⇒ *term*).
Quickcheck-Exhaustive.full-exhaustive (λ*denom*::′*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−Combinatorics.Stirling
  HOL−Computational-Algebra.Polynomial
begin

definition *pochhammer-poly* :: *nat* ⇒ *'a* :: {*comm-semiring-1*} *poly* **where**
  pochhammer-poly n = *Poly* [*of-nat* (*stirling n k*). *k* ← [*0*..<*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* = *of-nat* (*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*) = [:*of-nat n*,*1*:] ∗ *pochhammer-poly n*
  ⟨*proof*⟩

lemma *pochhammer-poly-altdef*: *pochhammer-poly n* = (∏ *i*<*n*. [:*of-nat i*,*1*:])
  ⟨*proof*⟩

lemma *eval-pochhammer-poly*: *poly* (*pochhammer-poly n*) *k* = *pochhammer k n*
  ⟨*proof*⟩

lemma *pochhammer-poly-Suc'*:
  pochhammer-poly (*Suc n*) = *pCons 0* (*pcompose* (*pochhammer-poly n*) [:*1*,*1*:])
  ⟨*proof*⟩

end

3 Miscellaneous material required for linear recurrences

theory *Linear-Recurrences-Misc*
imports
  Complex-Main

$HOL-Computational\text{-}Algebra.Computational\text{-}Algebra$
$HOL-Computational\text{-}Algebra.Polynomial\text{-}Factorial$
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* (λ(*x*,*y*). *f x y*) (*zip xs ys*)
⟨*proof*⟩

lemma *zip-with-nth* [*simp*]:
n < *length xs* ⟹ *n* < *length ys* ⟹ *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* (λ*x y*. *f* (*g x y*)) *xs ys*
⟨*proof*⟩

lemma *zip-with-map*: *zip-with f* (*map g xs*) (*map h ys*) = *zip-with* (λ*x y*. *f* (*g x*) (*h y*)) *xs ys*
⟨*proof*⟩

lemma *zip-with-map-left*: *zip-with f* (*map g xs*) *ys* = *zip-with* (λ*x y*. *f* (*g x*) *y*) *xs ys*
⟨*proof*⟩

lemma *zip-with-map-right*: *zip-with f xs* (*map g ys*) = *zip-with* (λ*x y*. *f x* (*g y*)) *xs ys*
⟨*proof*⟩

lemma *zip-with-swap*: *zip-with* (λ*x y*. *f y x*) *xs ys* = *zip-with f ys xs*
⟨*proof*⟩

lemma *set-zip-with*: *set* (*zip-with f xs ys*) = (λ(*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′*:
  $zip\text{-}with\ f\ xs\ ys = [f\ (xs!i)\ (ys!i).\ i \leftarrow [0..<min\ (length\ xs)\ (length\ ys)]]$
  ⟨*proof*⟩

lemma *zip-altdef*: $zip\ xs\ ys = [(xs!i,\ ys!i).\ i \leftarrow [0..<min\ (length\ xs)\ (length\ ys)]]$
  ⟨*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 $\bigwedge x.\ x \in A \Longrightarrow poly\ p\ x = poly\ q\ x$
  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 $P\ 0\ \bigwedge p.\ (\bigwedge x.\ poly\ p\ x \neq 0) \Longrightarrow P\ p$
    $\bigwedge p\ x\ n.\ n > 0 \Longrightarrow poly\ p\ x \neq 0 \Longrightarrow P\ p \Longrightarrow P\ ([:-x,\ 1:]\ \hat{}\ n * p)$
  shows $P\ p$
⟨*proof*⟩

lemma *complex-poly-decompose*:
  $smult\ (lead\text{-}coeff\ p)\ (\prod z|poly\ p\ z = 0.\ [:-z,\ 1:]\ \hat{}\ order\ z\ p) = (p :: complex\ poly)$
⟨*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 $c :: 'a :: field\text{-}gcd$
  assumes $c \neq c'$
  shows $coprime\ [:c,1:]\ [:c',1:]$
⟨*proof*⟩

lemma *coprime-linear-poly'*:
  fixes c :: *'a* :: *field-gcd*
  assumes $c \neq c'$ $c \neq 0$ $c' \neq 0$
  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 $x, y_1, \ldots, y_n$ of a ring R, where the y_i are pairwise coprime. A *Partial Fraction Decomposition* of these elements (or rather the formal quotient $x/(y_1 \ldots y_n)$ 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 $y_1, \ldots, y_n$ 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*]: *length* (*decompose x ys*) = *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*]:
  decompose x ys = *decompose' x ys* (*tl* (*rev* (*decompose-aux 1* (*rev ys*))))
⟨*proof*⟩

The next function performs the second step: Given a quotient of the form x/y^n, it returns a list of $x_0, \ldots, x_n$ such that $x/y^n = x_0/y^n + \ldots + x_{n-1}/y + x_n$ 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*]: *length* (*snd* (*normalise-decomp x y n*)) = *n*
  ⟨*proof*⟩

The following constant implements the full process of partial fraction decomposition: The input is a quotient $x/(y_1^{k_1} \ldots y_n^{k_n})$ 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*

$\langle proof \rangle$

lemma *length-pfd2* [*simp*]:
$i < length\ ys \Longrightarrow length\ (snd\ (partial\text{-}fraction\text{-}decomposition\ x\ ys)\ !\ i) = snd\ (ys\ !\ i) + 1$
$\langle proof \rangle$

lemma *size-normalise-decomp*:
$a \in set\ (snd\ (normalise\text{-}decomp\ x\ y\ n)) \Longrightarrow y \neq 0 \Longrightarrow euclidean\text{-}size\ a < euclidean\text{-}size\ y$
$\langle proof \rangle$

lemma *size-partial-fraction-decomposition*:
$i < length\ xs \Longrightarrow fst\ (xs\ !\ i) \neq 0 \Longrightarrow x \in set\ (snd\ (partial\text{-}fraction\text{-}decomposition\ y\ xs)\ !\ i)$
$\Longrightarrow euclidean\text{-}size\ x < euclidean\text{-}size\ (fst\ (xs\ !\ i))$
$\langle proof \rangle$

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* :: $('a :: euclidean\text{-}ring\text{-}gcd) \Rightarrow ('b :: euclidean\text{-}semiring\text{-}cancel)$
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*:
$lift\ (a\ \widehat{}\ n) = lift\ a\ \widehat{}\ n$
$\langle proof \rangle$

definition *from-decomp* :: $'a \Rightarrow 'a \Rightarrow nat \Rightarrow 'b$ **where**
$from\text{-}decomp\ x\ y\ n = lift\ x\ div\ lift\ y\ \widehat{}\ n$

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

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* $\Longrightarrow$ *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 + ($\sum i<length\ ys.\ \sum j\le snd\ (ys!i).$
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* ⇒ $'a$ *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*)))
⟨*proof*⟩

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*) = *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* (λ*p*. *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* (λ*p*. *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* (λ*c*. [:*c* :: '*a* :: *field-gcd*:])) (*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* (λ*c*. [:*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 = (λ(*a*,*cs*). *Polynomial.smult a* (∏ (*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 = (λ(*a*,*cs*). *Polynomial.smult a* (∏ (*c*,*n*)←*cs*. [:*1*,−*c*:] ^ *Suc n*))

definition *is-factorization-of* **where**
is-factorization-of fctrs p =
(*interp-factorization fctrs* = *p* ∧ *distinct* (*map fst* (*snd fctrs*)))

definition *is-alt-factorization-of* **where**
is-alt-factorization-of fctrs p =
(*interp-alt-factorization fctrs* = *p* ∧ *0* ∉ *set* (*map fst* (*snd fctrs*)) ∧
distinct (*map fst* (*snd fctrs*)))

Regular and alternative factorisations are related by reflecting the polynomial.

lemma *interp-factorization-reflect*:
assumes (*0*::′*a*::*idom*) ∉ *fst* ‘ *set* (*snd fctrs*)
shows *reflect-poly* (*interp-factorization fctrs*) = *interp-alt-factorization fctrs*
⟨*proof*⟩

lemma *interp-alt-factorization-reflect*:
assumes (*0*::′*a*::*idom*) ∉ *fst* ‘ *set* (*snd fctrs*)
shows *reflect-poly* (*interp-alt-factorization fctrs*) = *interp-factorization fctrs*
⟨*proof*⟩

lemma *coeff-0-interp-factorization*:
coeff (*interp-factorization fctrs*) *0* = (*0* :: ′*a* :: *idom*) ⟷
fst fctrs = *0* ∨ *0* ∈ *fst* ‘ *set* (*snd fctrs*)

$\langle proof \rangle$

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*)
$\langle proof \rangle$

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*)
$\langle proof \rangle$

lemma *zero-in-factorization-iff*:
assumes *is-factorization-of fctrs p*
shows *coeff p 0 = 0* $\longleftrightarrow$ *p = 0* $\vee$ (*0*::$'$*a*::*idom*) $\in$ *fst* ‘ *set* (*snd fctrs*)
$\langle proof \rangle$

lemma *poly-prod-list* [*simp*]: *poly* (*prod-list ps*) *x* = *prod-list* (*map* (λ*p. poly p x*) *ps*)
$\langle proof \rangle$

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*}
$\langle proof \rangle$

lemma (**in** *monoid-mult*) *prod-list-prod-nth*: *prod-list xs* = ($\prod$ *i*<*length xs. xs* ! *i*)
$\langle proof \rangle$

lemma *order-prod*:
assumes $\bigwedge$*x. x* $\in$ *A* $\Longrightarrow$ *f x* $\neq$ *0*
assumes $\bigwedge$*x y. x* $\in$ *A* $\Longrightarrow$ *y* $\in$ *A* $\Longrightarrow$ *x* $\neq$ *y* $\Longrightarrow$ *coprime* (*f x*) (*f y*)
shows *order c* (*prod f A*) = ($\sum$ *x*$\in$*A. order c* (*f x*))
$\langle proof \rangle$

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*
$\langle proof \rangle$

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*⟩

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* $\neq$ *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 (λ*k*. *poly* (*inverse-irred-power-poly d n*) (*of-nat k*) ∗ *c*^*k*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩
  include *fps-syntax*
  ⟨*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* :: {*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 (λ*k*. *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*⟩

definition *solve-factored-ratfps* ::
(*'a* :: {*field-char-0*,*field-gcd*}) *poly* ⇒ (*'a* × *nat*) *list* ⇒ *'a poly* × (*'a poly* × *'a*) *list* **where**
solve-factored-ratfps p cs = (*let n* = *length cs in case poly-pfd-simple p cs of* (*a*, *zs*) ⇒
(*a*, *zip-with* (λ*zs* (*c*,*n*). ((∑ (*z*,*j*) ← *zip zs* [*0*..<*Suc n*].
inverse-irred-power-poly z (*n* + *1* − *j*)), *c*)) *zs cs*))

lemma *length-snd-poly-pfd-simple* [*simp*]: *length* (*snd* (*poly-pfd-simple p cs*)) = *length cs*
⟨*proof*⟩

lemma *length-nth-snd-poly-pfd-simple* [*simp*]:
i < *length cs* ⟹ *length* (*snd* (*poly-pfd-simple p cs*) ! *i*) = *snd* (*cs*!*i*) + *1*
⟨*proof*⟩

lemma *solve-factored-ratfps-roots*:
map snd (*snd* (*solve-factored-ratfps p cs*)) = *map fst cs*
⟨*proof*⟩

definition *interp-ratfps-solution* **where**
interp-ratfps-solution = (λ(*p*,*cs*) *n*. *coeff p n* + (∑ (*q*,*c*)←*cs*. *poly q* (*of-nat n*) ∗ *c* ^ *n*))

lemma *solve-factored-ratfps*:
fixes *p* :: *'a* :: {*field-char-0*,*field-gcd*} *poly* **and** *cs* :: (*'a* × *nat*) *list*
assumes *distinct*: *distinct* (*map fst cs*)
assumes *nz*: *0* ∉ *fst ‘ set cs*
shows *fps-of-poly p* / *fps-of-poly* (∏ (*c*,*n*)←*cs*. [:*1*,−*c*:]^*Suc n*) =
Abs-fps (*interp-ratfps-solution* (*solve-factored-ratfps p cs*)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

definition *solve-factored-ratfps'* **where**
solve-factored-ratfps' = (λ*p* (*a*,*cs*). *solve-factored-ratfps* (*smult* (*inverse a*) *p*) *cs*)

lemma *solve-factored-ratfps'*:
  assumes *is-alt-factorization-of fctrs q* $q \neq 0$
  shows *Abs-fps* (*interp-ratfps-solution* (*solve-factored-ratfps' p fctrs*)) =
  fps-of-poly p / *fps-of-poly q*
⟨*proof*⟩

lemma *degree-Poly-eq*:
  assumes $xs = [] \vee$ *last xs* $\neq 0$
  shows *degree* (*Poly xs*) = *length xs* − *1*
⟨*proof*⟩

lemma *degree-Poly'*: *degree* (*Poly xs*) ≤ *length xs* − *1*
  ⟨*proof*⟩

lemma *degree-inverse-irred-power-poly-le*:
  degree (*inverse-irred-power-poly c n*) ≤ *n* − *1*
  ⟨*proof*⟩

lemma *degree-inverse-irred-power-poly*:
  assumes $c \neq 0$
  shows *degree* (*inverse-irred-power-poly c n*) = *n* − *1*
  ⟨*proof*⟩

lemma *reflect-poly-0-iff* [*simp*]: *reflect-poly p* = *0* ⟷ *p* = *0*
  ⟨*proof*⟩

lemma *degree-sum-list-le*: ($\bigwedge$*p*. *p* ∈ *set ps* ⟹ *degree p* ≤ *T*) ⟹ *degree* (*sum-list ps*) ≤ *T*
  ⟨*proof*⟩

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*. *fps-nth* (*fps-of-poly p* / *fps-of-poly q*) *n* =
  coeff r n + ($\sum$ *c* | *poly q' c* = *0*. *poly* (*rs c*) (*of-nat n*) ∗ *c* ^ *n*)
  and $\bigwedge$*z*. *poly q' z* = *0* ⟹ *degree* (*rs z*) ≤ *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*≤*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*≤*min N k*. *cs* ! (*N* − *i*) ∗ *f* (*k* − *i*)). *k* ← [*0*..<*N*+*m*]])
⟨*proof*⟩

lemma *lhr-fps-aux*:
fixes f :: $nat \Rightarrow 'a$ :: *field*
assumes $\bigwedge n.\ n \geq m \Longrightarrow (\sum k \leq N.\ c\ k * f\ (n + k)) = 0$
assumes cN: $c\ N \neq 0$
defines $p \equiv Poly\ [c\ (N - k).\ k \leftarrow [0..<Suc\ N]]$
defines $q \equiv Poly\ [(\sum i \leq min\ N\ k.\ c\ (N - i) * f\ (k - i)).\ k \leftarrow [0..<N+m]]$
shows *Abs-fps f = fps-of-poly q / fps-of-poly p*
⟨*proof*⟩
include *fps-syntax*
⟨*proof*⟩

lemma *lhr-fps*:
fixes f :: $nat \Rightarrow 'a$ :: *field* **and** *cs* :: *'a list*
defines $N \equiv length\ cs - 1$
assumes *cs*: $cs \neq []$
assumes $\bigwedge n.\ n \geq m \Longrightarrow (\sum k \leq N.\ cs\ !\ k * f\ (n + k)) = 0$
assumes cN: $last\ cs \neq 0$
shows *Abs-fps f = fps-of-poly (lhr-fps-numerator m cs f) /*
fps-of-poly (lr-fps-denominator cs)
⟨*proof*⟩

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

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

lemma *lhr-rec*:
assumes $cs \neq []\ last\ cs \neq 0\ length\ fs \geq length\ cs - 1\ n \geq length\ fs$
shows $(\sum k<length\ cs.\ cs\ !\ k * lhr\ cs\ fs\ (n + 1 - length\ cs + k)) = 0$
⟨*proof*⟩

lemma *lhrI*:
assumes $cs \neq []\ last\ cs \neq 0\ length\ fs \geq length\ cs - 1$
assumes $\bigwedge n.\ n < length\ fs \Longrightarrow f\ n = fs\ !\ n$
assumes $\bigwedge n.\ n \geq length\ fs \Longrightarrow (\sum k<length\ cs.\ cs\ !\ k * f\ (n + 1 - length\ cs$ $+ k)) = 0$
shows $f\ n = lhr\ cs\ fs\ n$
⟨*proof*⟩

locale *linear-homogenous-recurrence* =
fixes f :: $nat \Rightarrow 'a$ :: *comm-semiring-0* **and** *cs fs* :: *'a list*
assumes *base*: $n < length\ fs \Longrightarrow f\ n = fs\ !\ n$

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* $\Longrightarrow$ ($\sum k<$*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*⟩

end

lemma *solve-lhr-aux*:
assumes *linear-homogenous-recurrence f cs fs*
assumes *is-factorization-of fctrs* (*lr-fps-denominator′ cs*)
shows *f* = *interp-ratfps-solution* (*solve-factored-ratfps′* (*lhr-fps-numerator*
(*length fs* + *1* $-$ *length cs*) *cs* ((!) *fs*)) *fctrs*)
⟨*proof*⟩

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* $\Rightarrow$ *′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*⟩

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* ⇒ *'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 (λ*n*. *of-nat* (*n*+*1*) ^ *k*) ∗ (*1* − *fps-X*) ^ (*k* + *1*)
⟨*proof*⟩

lemma *eulerian-poly'*:
Abs-fps (λ*n*. *of-nat* (*n*+*1*) ^ *k*) =
fps-of-poly (*eulerian-poly k* :: *'a* :: *field poly*) / (*1* − *fps-X*) ^ (*k* + *1*)
⟨*proof*⟩

lemma *eulerian-poly''*:
assumes *k*: *k* > *0*
shows *Abs-fps* (λ*n*. *of-nat n* ^ *k*) =
fps-of-poly (*pCons 0* (*eulerian-poly k* :: *'a* :: *field poly*)) / (*1* − *fps-X*) ^
(*k* + *1*)
⟨*proof*⟩

definition *fps-monom-poly* :: *'a* :: *field* ⇒ *nat* ⇒ *'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* ⇒ *nat* ⇒ *'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-fps (*λn. of-nat n* ^ *k*) = *fps-of-poly* (*fps-monom-poly 1 k*) / (*1* − *fps-X*) ^ (*k*+*1*)
⟨*proof*⟩

lemma *fps-monom*:
Abs-fps (*λn. of-nat n* ^ *k* ∗ *c* ^ *n*) =
fps-of-poly (*fps-monom-poly c k*) / (*1* − *fps-const c* ∗ *fps-X*) ^ (*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**
lir-fps-numerator m cs f g = (*let N* = *length cs* − *1 in*
Poly [(∑ *i*≤*min N k*. *cs* ! (*N* − *i*) ∗ *f* (*k* − *i*)) − *g k*. *k* ← [*0*..<*N*+*m*]])

lemma *lir-fps-numerator-code* [*code abstract*]:
coeffs (*lir-fps-numerator m cs f g*) = (*let N* = *length cs* − *1 in*
strip-while ((=) *0*) [(∑ *i*≤*min N k*. *cs* ! (*N* − *i*) ∗ *f* (*k* − *i*)) − *g k*. *k* ← [*0*..<*N*+*m*]])
⟨*proof*⟩

locale *linear-inhomogenous-recurrence* =
fixes *f g* :: *nat* ⇒ *'a* :: *comm-ring* **and** *cs fs* :: *'a list*
assumes *base*: *n* < *length fs* ⟹ *f n* = *fs* ! *n*
assumes *cs-not-null* [*simp*]: *cs* ≠ [] **and** *last-cs* [*simp*]: *last cs* ≠ *0*
and *hd-cs* [*simp*]: *hd cs* ≠ *0* **and** *enough-base*: *length fs* + *1* ≥ *length cs*
assumes *rec*: *n* ≥ *length fs* + *1* − *length cs* ⟹
(∑ *k*<*length cs*. *cs* ! *k* ∗ *f* (*n* + *k*)) = *g* (*n* + *length cs* − *1*)
begin

lemma *coeff-0-lr-fps-denominator* [*simp*]: *coeff* (*lr-fps-denominator cs*) *0* = *last cs*
⟨*proof*⟩

lemma *lir-fps-numerator-altdef*:
lir-fps-numerator (*length fs* + *1* − *length cs*) *cs f g* =
lir-fps-numerator (*length fs* + *1* − *length cs*) *cs* ((!) *fs*) *g*
⟨*proof*⟩

end

context
begin

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

lemma *lir-fps*:
  fixes $f\ g :: nat \Rightarrow 'a :: field$ **and** $cs :: 'a\ list$
  defines $N \equiv length\ cs - 1$
  assumes *cs*: $cs \neq []$
  assumes $\bigwedge n.\ n \geq m \Longrightarrow (\sum k \leq N.\ cs\ !\ k * f\ (n + k)) = g\ (n + N)$
  assumes *cN*: $last\ cs \neq 0$
  shows $Abs\text{-}fps\ f = (fps\text{-}of\text{-}poly\ (lir\text{-}fps\text{-}numerator\ m\ cs\ f\ g) + Abs\text{-}fps\ g)\ /$
  $fps\text{-}of\text{-}poly\ (lr\text{-}fps\text{-}denominator\ cs)$
⟨*proof*⟩

end

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

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

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

lemma *eval-polyexp-Cons*:
  $eval\text{-}polyexp\ (x\#xs) = (\lambda n.\ (case\ x\ of\ (a,k,b) \Rightarrow a * of\text{-}nat\ n\ \hat{}\ k * b\ \hat{}\ n) + eval\text{-}polyexp\ xs\ n)$
  ⟨*proof*⟩

definition $polyexp\text{-}fps :: ('a :: field)\ polyexp \Rightarrow 'a\ fps$ **where**
  $polyexp\text{-}fps\ xs =$
  $(\sum (a,k,b) \leftarrow xs.\ fps\text{-}of\text{-}poly\ (Polynomial.smult\ a\ (fps\text{-}monom\text{-}poly\ b\ k))\ /$
  $(1 - fps\text{-}const\ b * fps\text{-}X)\ \hat{}\ (k + 1))$

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

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

definition *polyexp-ratfps* :: (′*a* :: *field-gcd*) *polyexp* ⇒ ′*a ratfps* **where**
polyexp-ratfps xs =
(∑(*a*,*k*,*b*)←*xs*. *ratfps-of-poly* (*Polynomial.smult a* (*fps-monom-poly b k*)) /
ratfps-of-poly ([:*1*, −*b*:] ^ (*k* + *1*)))

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* :: *field-gcd list* ⇒ ′*a list* ⇒ ′*a polyexp* ⇒ (′*a ratfps*) *option* **where**
lir-fps cs fs g = (*if cs* = [] ∨ *length fs* < *length cs* − *1 then None else*
let m = *length fs* + *1* − *length cs*;
p = *lir-fps-numerator m cs* (λ*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* ⇒ ′*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*⟩

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** $F \leq$ *at-infinity*
shows *poly p* $\sim[F]$ ($\lambda x.$ *lead-coeff p* $*$ x ^ *degree p*)
⟨*proof*⟩

lemma *poly-bigtheta*:
assumes $p \neq 0$ **and** $F \leq$ *at-infinity*
shows *poly p* $\in \Theta[F](\lambda x.$ x ^ *degree p*)
⟨*proof*⟩

lemma *poly-bigo*:
assumes $F \leq$ *at-infinity* **and** *degree p* $\leq k$
shows *poly p* $\in O[F](\lambda x.$ x ^ k)
⟨*proof*⟩

lemma *reflect-poly-dvdI*:
fixes *p q* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
assumes *p dvd q*
shows *reflect-poly p dvd reflect-poly q*
⟨*proof*⟩

lemma *smult-altdef*: *smult c p* = [:*c*:] $*$ *p*
⟨*proof*⟩

lemma *smult-power*: *smult* (*c* ^ *n*) (*p* ^ *n*) = (*smult c p*) ^ *n*
⟨*proof*⟩

lemma *order-reflect-poly-ge*:
fixes *c* :: *'a* :: *field*
assumes $c \neq 0$ **and** $p \neq 0$
shows *order c* (*reflect-poly p*) $\geq$ *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-poly p*) = *order* (*1* / *c*) *p*

$\langle proof \rangle$

lemma *poly-reflect-eq-0-iff*:
poly (reflect-poly p) (x :: 'a :: field) = 0 $\longleftrightarrow$ *p = 0* $\vee$ *x* $\neq$ *0* $\wedge$ *poly p (1 / x) = 0*
$\langle proof \rangle$

theorem *ratfps-nth-bigo*:
fixes *q :: complex poly*
assumes *R > 0*
assumes *roots1*: $\bigwedge$*z. z* $\in$ *ball 0 (1 / R)* $\Longrightarrow$ *poly q z* $\neq$ *0*
assumes *roots2*: $\bigwedge$*z. z* $\in$ *sphere 0 (1 / R)* $\Longrightarrow$ *poly q z = 0* $\Longrightarrow$ *order z q* $\leq$ *Suc k*
shows *fps-nth (fps-of-poly p / fps-of-poly q)* $\in$ *O*(λ*n. of-nat n ^ k * of-real R ^ n*)
$\langle proof \rangle$

lemma *order-power*: *p* $\neq$ *0* $\Longrightarrow$ *order c (p ^ n) = n * order c p*
$\langle proof \rangle$

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 $\neg$*coprime p q*
$\langle proof \rangle$

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* $\Longrightarrow$ $\forall$*x*$\in$*ball 0 (1 / R). poly c x* $\neq$ *0*
assumes *roots2*: $\bigwedge$*c l. (c, l)* $\in$ *set cs* $\Longrightarrow$ *l > Suc k* $\Longrightarrow$ $\forall$*x*$\in$*sphere 0 (1 / R). poly c x* $\neq$ *0*
shows *fps-nth (fps-of-poly p / fps-of-poly q)* $\in$ *O*(λ*n. of-nat n ^ k * of-real R ^ n*)
$\langle proof \rangle$

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

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

definition *ratfps-has-asymptotics* **where**
ratfps-has-asymptotics q k R $\longleftrightarrow$ $q \neq 0 \wedge R > 0 \wedge$
(*let cs* = *snd* (*yun-factorization gcd q*)
in *list-all* (λ*cl. proots-ball-card* (*fst cl*) *0* (*1* / *R*) = *0*) *cs* $\wedge$
list-all (λ*cl. proots-sphere-card* (*fst cl*) *0* (*1* / *R*) = *0*) (*filter* (λ*cl. snd cl* > *Suc k*) *cs*))

lemma *ratfps-has-asymptotics-correct*:
assumes *ratfps-has-asymptotics q k R*
shows *fps-nth* (*fps-of-poly p* / *fps-of-poly q*) $\in$ *O*(λ*n. of-nat n* ^ *k* $*$ *of-real R* ^ *n*)
⟨*proof*⟩

value *map* (*fps-nth* (*fps-of-poly* [:*0*, *1*:] / *fps-of-poly* [:*1*, −*1*, −*1* :: *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*:]) $\in$
O(λ*n. of-nat n* ^ *0* $*$ *complex-of-real 1.618034* ^ *n*)
⟨*proof*⟩

lemma *fps-nth* (*fps-of-poly 1* / *fps-of-poly* [:*1*, −*3*, *3*, −*1* :: *complex*:]) $\in$
O(λ*n. of-nat n* ^ *2* $*$ *complex-of-real 1* ^ *n*)
⟨*proof*⟩

lemma *fps-nth* (*fps-of-poly f* / *fps-of-poly* [:*5*, *4*, *3*, *2*, *1* :: *complex*:]) $\in$
O(λ*n. of-nat n* ^ *0* $*$ *complex-of-real 0.69202* ^ *n*)
⟨*proof*⟩

end