Dirichlet Series

Manuel Eberl

March 17, 2025

Abstract

This entry is a formalisation of much of Chapters 2, 3, and 11 of Apostol's "Introduction to Analytic Number Theory" [1]. This includes:

- Definitions and basic properties for several number-theoretic functions (Euler's φ, Möbius μ, Liouville's λ, the divisor function σ, von Mangoldt's Λ)
- Executable code for most of these functions, the most efficient implementations using the factoring algorithm by Thiemann *et al.*
- Dirichlet products and formal Dirichlet series
- Analytic results connecting convergent formal Dirichlet series to complex functions
- Euler product expansions
- Asymptotic estimates of number-theoretic functions including the density of squarefree integers and the average number of divisors of a natural number

These results are useful as a basis for developing more number-theoretic results, such as the Prime Number Theorem.

Contents

1 Miscellaneous auxiliary facts

theory *Dirichlet-Misc*
  imports
    HOL−Number-Theory.Number-Theory
begin

lemma
  fixes *a k* :: *nat*
  assumes $a > 1$ $k > 0$
  shows *geometric-sum-nat-aux*: $(a - 1) * (\sum i<k.\ a \mathbin{\hat{}} i) = a \mathbin{\hat{}} k - 1$
    and *geometric-sum-nat-dvd*: $a - 1$ *dvd* $a \mathbin{\hat{}} k - 1$
    and *geometric-sum-nat*: $(\sum i<k.\ a \mathbin{\hat{}} i) = (a \mathbin{\hat{}} k - 1)$ *div* $(a - 1)$
proof −
  have (*real* $a - 1) * (\sum i<k.$ *real* $a \mathbin{\hat{}} i) =$ *real* $a \mathbin{\hat{}} k - 1$
    using *assms* **by** (*subst geometric-sum*) *auto*
  also have (*real* $a - 1) * (\sum i<k.$ *real* $a \mathbin{\hat{}} i) =$ *real* $((a - 1) * (\sum i<k.\ a \mathbin{\hat{}} i))$
    using *assms* **by** (*simp add*: *of-nat-diff*)
  also have *real* $a \mathbin{\hat{}} k - 1 =$ *real* $(a \mathbin{\hat{}} k - 1)$ **using** *assms* **by** (*subst of-nat-diff*) *auto*
  finally show ∗: $(a - 1) * (\sum i<k.\ a \mathbin{\hat{}} i) = a \mathbin{\hat{}} k - 1$ **by** (*subst* (*asm*) *of-nat-eq-iff*)
  show $a - 1$ *dvd* $a \mathbin{\hat{}} k - 1$ **by** (*subst* ∗ [*symmetric*]) *simp*
  from *assms* **show** $(\sum i<k.\ a \mathbin{\hat{}} i) = (a \mathbin{\hat{}} k - 1)$ *div* $(a - 1)$
    by (*subst* ∗ [*symmetric*]) *simp*
qed

lemma *dvd-div-gt0*: *d dvd n* $\Longrightarrow n > 0 \Longrightarrow n$ *div* $d > (0{::}nat)$
  by *auto*

lemma *Set-filter-insert*:
  Set.filter P (*insert x A*) = (*if P x then insert x* (*Set.filter P A*) *else Set.filter P A*)
  by *auto*

lemma *Set-filter-union*: *Set.filter P* $(A \cup B) =$ *Set.filter P A* ∪ *Set.filter P B*
  by *auto*

lemma *Set-filter-empty* [*simp*]: *Set.filter P* {} = {}
  by *auto*

lemma *Set-filter-image*: *Set.filter P* (*f* ‘ *A*) = *f* ‘ *Set.filter* $(P \circ f)$ *A*
  by *auto*

lemma *Set-filter-cong* [*cong*]:
    $(\bigwedge x.\ x \in A \Longrightarrow P\ x \longleftrightarrow Q\ x) \Longrightarrow A = B \Longrightarrow$ *Set.filter P A* = *Set.filter Q B*
  by *auto*

lemma *inj-on-insert′*: $(\bigwedge B.\ B \in A \Longrightarrow x \notin B) \Longrightarrow$ *inj-on* (*insert x*) *A*

by (*auto simp*: *inj-on-def insert-eq-iff*)

lemma
assumes *finite A A* ≠ {}
shows *card-even-subset-aux*: *card* {*B*. *B* ⊆ *A* ∧ *even* (*card B*)} = *2* ^ (*card A* − *1*)
and *card-odd-subset-aux*: *card* {*B*. *B* ⊆ *A* ∧ *odd* (*card B*)} = *2* ^ (*card A* − *1*)
and *card-even-odd-subset*: *card* {*B*. *B* ⊆ *A* ∧ *even* (*card B*)} = *card* {*B*. *B* ⊆ *A* ∧ *odd* (*card B*)}
proof −
from *assms* **have** ∗: *2* ∗ *card* (*Set.filter* (*even* ∘ *card*) (*Pow A*)) = *2* ^ *card A*
proof (*induction A rule*: *finite-ne-induct*)
case (*singleton x*)
hence *Pow* {*x*} = {{}, {*x*}} **by** *auto*
thus *?case* **by** (*simp add*: *Set-filter-insert*)
next
case (*insert x A*)
note *fin* = *finite-subset*[*OF* - ‹*finite A*›]
have *Pow* (*insert x A*) = *Pow A* ∪ *insert x* ‘ *Pow A* **by** (*rule Pow-insert*)
have *Set.filter* (*even* ∘ *card*) (*Pow* (*insert x A*)) =
Set.filter (*even* ∘ *card*) (*Pow A*) ∪
insert x ‘ *Set.filter* (*even* ∘ *card* ∘ *insert x*) (*Pow A*)
unfolding *Pow-insert Set-filter-union Set-filter-image* **by** *blast*
also have *Set.filter* (*even* ∘ *card* ∘ *insert x*) (*Pow A*) = *Set.filter* (*odd* ∘ *card*) (*Pow A*)
unfolding *o-def*
by (*intro Set-filter-cong refl*, *subst card-insert-disjoint*)
(*insert insert.hyps*, *auto dest*: *finite-subset*)
also have *card* (*Set.filter* (*even* ∘ *card*) (*Pow A*) ∪ *insert x* ‘ . . .) =
card (*Set.filter* (*even* ∘ *card*) (*Pow A*)) + *card* (*insert x* ‘ . . .)
(**is** *card* (*?A* ∪ *?B*) = -)
by (*intro card-Un-disjoint finite-filter finite-imageI*) (*auto simp*: *insert.hyps*)
also have *card ?B* = *card* (*Set.filter* (*odd* ∘ *card*) (*Pow A*))
using *insert.hyps* **by** (*intro card-image inj-on-insert′*) *auto*
also have *Set.filter* (*odd* ∘ *card*) (*Pow A*) = *Pow A* − *Set.filter* (*even* ∘ *card*) (*Pow A*)
by *auto*
also have *card* . . . = *card* (*Pow A*) − *card* (*Set.filter* (*even* ∘ *card*) (*Pow A*))
using *insert.hyps* **by** (*subst card-Diff-subset*) (*auto simp*: *finite-filter*)
also have *card* (*Set.filter* (*even* ∘ *card*) (*Pow A*)) + . . . = *card* (*Pow A*)
by (*intro add-diff-inverse-nat*, *subst not-less*, *rule card-mono*) (*insert insert.hyps*, *auto*)
also have *2* ∗ . . . = *2* ^ *card* (*insert x A*)
using *insert.hyps* **by** (*simp add*: *card-Pow*)
finally show *?case* **.**
qed
from ∗ **show** *A*: *card* {*B*. *B* ⊆ *A* ∧ *even* (*card B*)} = *2* ^ (*card A* − *1*)
by (*cases card A*) (*simp-all add*: *Set.filter-def*)

have *Set.filter* (*odd* ∘ *card*) (*Pow A*) = *Pow A* − *Set.filter* (*even* ∘ *card*) (*Pow A*) **by** *auto*
also have *2* ∗ *card* ... = *2* ∗ *2* ^ *card A* − *2* ∗ *card* (*Set.filter* (*even* ∘ *card*) (*Pow A*))
using *assms* **by** (*subst card-Diff-subset*) (*auto intro*!: *finite-filter simp*: *card-Pow*)
also note ∗
also have *2* ∗ *2* ^ *card A* − *2* ^ *card A* = (*2* ^ *card A* :: *nat*) **by** *simp*
finally show *B*: *card* {*B*. *B* ⊆ *A* ∧ *odd* (*card B*)} = *2* ^ (*card A* − *1*)
by (*cases card A*) (*simp-all add*: *Set.filter-def*)

from *A* **and** *B* **show** *card* {*B*. *B* ⊆ *A* ∧ *even* (*card B*)} = *card* {*B*. *B* ⊆ *A* ∧ *odd* (*card B*)} **by** *simp*
qed

lemma *bij-betw-prod-divisors-coprime*:
assumes *coprime a* (*b* :: *nat*)
shows *bij-betw* (λ*x*. *fst x* ∗ *snd x*) ({*d*. *d dvd a*} × {*d*. *d dvd b*}) {*k*. *k dvd a* ∗ *b*}
unfolding *bij-betw-def*
proof
from *assms* **show** *inj-on* (λ*x*. *fst x* ∗ *snd x*) ({*d*. *d dvd a*} × {*d*. *d dvd b*})
by (*auto simp*: *inj-on-def coprime-crossproduct-nat coprime-divisors*)
show (λ*x*. *fst x* ∗ *snd x*) ‘ ({*d*. *d dvd a*} × {*d*. *d dvd b*}) = {*k*. *k dvd a* ∗ *b*}
proof *safe*
fix *x* **assume** *x dvd a* ∗ *b*
then obtain *b′ c′* **where** *x* = *b′* ∗ *c′ b′ dvd a c′ dvd b*
using *division-decomp* **by** *blast*
thus *x* ∈ (λ*x*. *fst x* ∗ *snd x*) ‘ ({*d*. *d dvd a*} × {*d*. *d dvd b*}) **by** *force*
qed (*insert assms*, *auto intro*: *mult-dvd-mono*)
qed

lemma *bij-betw-prime-power-divisors*:
assumes *prime* (*p* :: *nat*)
shows *bij-betw* ((^) *p*) {..*k*} {*d*. *d dvd p* ^ *k*}
unfolding *bij-betw-def*
proof
from *assms* **have** ∗: *p* > *1* **by** (*simp add*: *prime-gt-Suc-0-nat*)
show *inj-on* ((^) *p*) {..*k*} **using** *assms*
by (*auto simp*: *inj-on-def prime-gt-Suc-0-nat power-inject-exp*[*OF* ∗])
show (^) *p* ‘ {..*k*} = {*d*. *d dvd p* ^ *k*}
using *assms* **by** (*auto simp*: *le-imp-power-dvd divides-primepow-nat*)
qed

lemma *sum-divisors-coprime-mult*:
assumes *coprime a* (*b* :: *nat*)
shows (∑ *d* | *d dvd a* ∗ *b*. *f d*) = (∑ *r* | *r dvd a*. ∑ *s* | *s dvd b*. *f* (*r* ∗ *s*))
proof −
have (∑ *r* | *r dvd a*. ∑ *s* | *s dvd b*. *f* (*r* ∗ *s*)) =

$(\sum z{\in}\{r.\ r\ dvd\ a\} \times \{s.\ s\ dvd\ b\}.\ f\ (fst\ z * snd\ z))$
by (*subst sum.cartesian-product*) (*simp add*: *case-prod-unfold*)
also have ... $= (\sum d \mid d\ dvd\ a * b.\ f\ d)$
by (*intro sum.reindex-bij-betw bij-betw-prod-divisors-coprime assms*)
finally show *?thesis* **..**
qed

end

2 Multiplicative arithmetic functions

theory *Multiplicative-Function*
imports
HOL−Number-Theory.Number-Theory
Dirichlet-Misc
begin

2.1 Definition

locale *multiplicative-function* =
fixes $f :: nat \Rightarrow 'a :: comm\text{-}semiring\text{-}1$
assumes *zero* [*simp*]: $f\ 0 = 0$
assumes *one* [*simp*]: $f\ 1 = 1$
assumes *mult-coprime-aux*: $a > 1 \Longrightarrow b > 1 \Longrightarrow coprime\ a\ b \Longrightarrow f\ (a * b) = f\ a * f\ b$
begin

lemma *Suc-0* [*simp*]: $f\ (Suc\ 0) = 1$
using *one* **by** (*simp del*: *one*)

lemma *mult-coprime*:
assumes *coprime a b*
shows $f\ (a * b) = f\ a * f\ b$
proof −
{fix $n :: nat$ **consider** $n = 0 \mid n = 1 \mid n > 1$ **by** *force***}** **note** $P = this$
show *?thesis* **by** (*cases a rule*: *P*; *cases b rule*: *P*) (*simp-all add*: *mult-coprime-aux assms*)
qed

lemma *prod-coprime*:
assumes $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x \neq y \Longrightarrow coprime\ (g\ x)\ (g\ y)$
shows $f\ (prod\ g\ A) = (\prod x{\in}A.\ f\ (g\ x))$
using *assms*
proof (*induction rule*: *infinite-finite-induct*)
case (*insert x A*)
from *insert* **have** $f\ (prod\ g\ (insert\ x\ A)) = f\ (g\ x * prod\ g\ A)$ **by** *simp*
also have ... $= f\ (g\ x) * f\ (prod\ g\ A)$ **using** *insert.prems insert.hyps*
by (*auto intro*: *mult-coprime prod-coprime-right*)
also have ... $= (\prod x{\in}insert\ x\ A.\ f\ (g\ x))$ **using** *insert* **by** *simp*

finally show *?case* **.**
qed *auto*

lemma *prod-prime-factors*:
assumes $n > 0$
shows $f\ n = (\prod p{\in}prime\text{-}factors\ n.\ f\ (p\ \widehat{}\ multiplicity\ p\ n))$
proof −
have $n = (\prod p{\in}prime\text{-}factors\ n.\ p\ \widehat{}\ multiplicity\ p\ n)$
using *Primes.prime-factorization-nat assms* **by** *blast*
also have $f\ \ldots = (\prod p{\in}prime\text{-}factors\ n.\ f\ (p\ \widehat{}\ multiplicity\ p\ n))$
by (*rule prod-coprime*) (*auto simp add*: *in-prime-factors-imp-prime primes-coprime*)

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

lemma *multiplicative-sum-divisors*: *multiplicative-function* $(\lambda n.\ \sum d \mid d\ dvd\ n.\ f\ d)$
proof
fix *a b* :: *nat* **assume** *ab*: $a > 1\ b > 1$ *coprime a b*
hence $(\sum d \mid d\ dvd\ a * b.\ f\ d) = (\sum r \mid r\ dvd\ a.\ \sum s \mid s\ dvd\ b.\ f\ (r * s))$
by (*intro sum-divisors-coprime-mult*)
also have $\ldots = (\sum r \mid r\ dvd\ a.\ \sum s \mid s\ dvd\ b.\ f\ r * f\ s)$
using *ab*(*3*)
by (*auto intro*!: *sum.cong intro*: *mult-coprime coprime-imp-coprime dvd-trans*)
also have $\ldots = (\sum r \mid r\ dvd\ a.\ f\ r) * (\sum s \mid s\ dvd\ b.\ f\ s)$
by (*subst sum-distrib-right*, *subst sum-distrib-left*) *simp-all*
finally show $(\sum d \mid d\ dvd\ a * b.\ f\ d) = (\sum r \mid r\ dvd\ a.\ f\ r) * (\sum s \mid s\ dvd\ b.\ f\ s)$ **.**
qed *auto*

end

locale *multiplicative-function′* = *multiplicative-function f* **for** *f* :: *nat* ⇒ *′a* :: *comm-semiring-1* +
fixes *f-prime-power* :: *nat* ⇒ *nat* ⇒ *′a* **and** *f-prime* :: *nat* ⇒ *′a*
assumes *prime-power*: *prime p* ⟹ $k > 0$ ⟹ $f\ (p\ \widehat{}\ k)$ = *f-prime-power p k*
assumes *prime-aux*: *prime p* ⟹ *f-prime-power p 1* = *f-prime p*
begin

lemma *prime*: *prime p* ⟹ *f p* = *f-prime p*
using *prime-power*[*of p 1*] *prime-aux*[*of p*] **by** *simp*

lemma *prod-prime-factors′*:
assumes $n > 0$
shows $f\ n = (\prod p{\in}prime\text{-}factors\ n.\ f\text{-}prime\text{-}power\ p\ (multiplicity\ p\ n))$
by (*subst prod-prime-factors*[*OF assms*(*1*)])
(*intro prod.cong refl prime-power*, *auto simp*: *prime-factors-multiplicity*)

lemma *efficient-code-aux*:

assumes *n* > *0* *set ps* = (λ*p*. (*p*, *multiplicity p n* − *1*)) ‘ *prime-factors n distinct ps*
shows *f n* = (∏ (*p*,*d*) ← *ps*. *f-prime-power p* (*Suc d*))
proof −
from *assms* **have**
(∏ (*p*,*d*) ← *ps*. *f-prime-power p* (*Suc d*)) =
(∏ (*p*,*d*)∈(λ*p*. (*p*, *multiplicity p n* − *1*)) ‘ *prime-factors n*. *f-prime-power p* (*Suc d*))
by (*subst prod.distinct-set-conv-list* [*symmetric*]) *simp-all*
also have ... = (∏ *x*∈*prime-factors n*. *f-prime-power x* (*multiplicity x n*))
by (*subst prod.reindex*) (*auto simp*: *inj-on-def prime-factors-multiplicity intro*!: *prod.cong*)
also have ... = *f n* **by** (*rule prod-prime-factors′* [*symmetric*]) *fact+*
finally show *?thesis* **..**
qed

lemma *efficient-code*:
assumes *set* (*ps* ()) = (λ*p*. (*p*, *multiplicity p n* − *1*)) ‘ *prime-factors n distinct* (*ps* ())
shows *f n* = (*if n* = *0* *then* *0* *else* (∏ (*p*,*d*) ← *ps* (). *f-prime-power p* (*Suc d*)))
using *efficient-code-aux*[*of n ps* ()] *assms* **by** *simp*

end

locale *completely-multiplicative-function* =
fixes *f* :: *nat* ⇒ ′*a* :: *comm-semiring-1*
assumes *zero-aux*: *f* *0* = *0*
assumes *one-aux*: *f* (*Suc* *0*) = *1*
assumes *mult-aux*: *a* > *1* ⟹ *b* > *1* ⟹ *f* (*a* ∗ *b*) = *f a* ∗ *f b*
begin

lemma *mult*: *f* (*a* ∗ *b*) = *f a* ∗ *f b*
proof −
{fix *n* :: *nat* **consider** *n* = *0* | *n* = *1* | *n* > *1* **by** *force***}** **note** *P* = *this*
show *?thesis* **by** (*cases a rule*: *P*; *cases b rule*: *P*) (*simp-all add*: *zero-aux one-aux mult-aux*)
qed

sublocale *multiplicative-function f*
by *standard* (*simp-all add*: *zero-aux one-aux mult*)

lemma *prod*: *f* (*prod g A*) = (∏ *x*∈*A*. *f* (*g x*))
by (*induction A rule*: *infinite-finite-induct*) (*simp-all add*: *mult*)

lemma *power*: *f* (*n* ^ *m*) = *f n* ^ *m*
by (*induction m*) (*simp-all add*: *mult*)

lemma *prod-prime-factors′*: *n* > *0* ⟹ *f n* = (∏ *p*∈*prime-factors n*. *f p* ^ *multi-*

plicity p n)
by (*subst prime-factorization-nat*) (*simp-all add*: *prod power*)

end

locale *completely-multiplicative-function′* =
completely-multiplicative-function f **for** *f* :: *nat* ⇒ ′*a* :: *comm-semiring-1* +
fixes *f-prime* :: *nat* ⇒ ′*a*
assumes *f-prime*: *prime p* ⟹ *f p* = *f-prime p*
begin

lemma *prod-prime-factors″*: *n* > *0* ⟹ *f n* = (∏ *p*∈*prime-factors n*. *f-prime p* ^ *multiplicity p n*)
by (*subst prod-prime-factors′*) (*auto simp*: *f-prime prime-factors-multiplicity intro*!: *prod.cong*)

lemma *efficient-code-aux*:
assumes *n* > *0 set ps* = (λ*p*. (*p*, *multiplicity p n* − *1*)) ‘ *prime-factors n distinct ps*
shows *f n* = (∏ (*p*,*d*) ← *ps*. *f-prime p* ^ *Suc d*)
proof −
from *assms* **have**
(∏ (*p*,*d*) ← *ps*. *f-prime p* ^ *Suc d*) =
(∏ (*p*,*d*)∈(λ*p*. (*p*, *multiplicity p n* − *1*)) ‘ *prime-factors n*. *f-prime p* ^ *Suc d*)
by (*subst prod.distinct-set-conv-list* [*symmetric*]) *simp-all*
also have . . . = (∏ *x*∈*prime-factors n*. *f-prime x* ^ *multiplicity x n*)
by (*subst prod.reindex*) (*auto simp*: *inj-on-def prime-factors-multiplicity*
simp del: *power-Suc intro*!: *prod.cong*)
also have . . . = *f n* **by** (*rule prod-prime-factors″* [*symmetric*]) *fact*+
finally show *?thesis* **..**
qed

lemma *efficient-code*:
assumes *set* (*ps* ()) = (λ*p*. (*p*, *multiplicity p n* − *1*)) ‘ *prime-factors n distinct* (*ps* ())
shows *f n* = (*if n* = *0 then 0 else* (∏ (*p*,*d*) ← *ps* (). *f-prime p* ^ *Suc d*))
using *efficient-code-aux*[*of n ps* ()] *assms* **by** *simp*

end

lemma *multiplicative-function-eqI*:
assumes *multiplicative-function f multiplicative-function g*
assumes ⋀*p k*. *prime p* ⟹ *k* > *0* ⟹ *f* (*p* ^ *k*) = *g* (*p* ^ *k*)
shows *f n* = *g n*
proof −
interpret *f*: *multiplicative-function f* **by** *fact*
interpret *g*: *multiplicative-function g* **by** *fact*
show *?thesis*

```
  proof (cases n > 0)
    case True
    thus ?thesis
      using f.prod-prime-factors[OF True] g.prod-prime-factors[OF True]
      by (auto intro!: prod.cong assms simp: prime-factors-multiplicity)
  qed simp-all
qed

lemma multiplicative-function-of-natI:
  multiplicative-function f ⟹ multiplicative-function (λn. of-nat (f n))
  unfolding multiplicative-function-def by auto

lemma multiplicative-function-of-natD:
  multiplicative-function (λn. of-nat (f n) :: 'a :: {ring-char-0, comm-semiring-1})
⟹
     multiplicative-function f
  unfolding multiplicative-function-def
  by (auto simp: of-nat-mult [symmetric] of-nat-eq-1-iff simp del: of-nat-mult)

lemma multiplicative-function-mult:
  assumes multiplicative-function f  multiplicative-function g
  shows   multiplicative-function (λn. f n ∗ g n)
proof
  interpret f: multiplicative-function f by fact
  interpret g: multiplicative-function g by fact
  show f 0 ∗ g 0 = 0 f 1 ∗ g 1 = 1 by simp-all
  fix a b :: nat assume a > 1 b > 1 coprime a b
  thus f (a ∗ b) ∗ g (a ∗ b) = (f a ∗ g a) ∗ (f b ∗ g b)
    by (simp-all add: f.mult-coprime g.mult-coprime mult-ac)
qed

lemma multiplicative-function-inverse:
  fixes f :: nat ⇒ 'a :: field
  assumes multiplicative-function f
  shows   multiplicative-function (λn. inverse (f n))
proof
  interpret f: multiplicative-function f by fact
  show inverse (f 0) = 0 inverse (f 1) = 1 by simp-all
  fix a b :: nat assume a > 1 b > 1 coprime a b
  thus inverse (f (a ∗ b)) = inverse (f a) ∗ inverse (f b)
    by (simp-all add: f.mult-coprime field-simps)
qed

lemma multiplicative-function-divide:
  fixes f :: nat ⇒ 'a :: field
  assumes multiplicative-function f  multiplicative-function g
  shows   multiplicative-function (λn. f n / g n)
proof −
  have multiplicative-function (λn. f n ∗ inverse (g n))
```

by (*intro multiplicative-function-mult multiplicative-function-inverse assms*)
also have ($\lambda n.$ *f n* ∗ *inverse* (*g n*)) = ($\lambda n.$ *f n* / *g n*)
by (*simp add*: *field-simps*)
finally show *?thesis* **.**
qed

lemma *completely-multiplicative-function-mult*:
assumes *completely-multiplicative-function f completely-multiplicative-function g*
shows *completely-multiplicative-function* ($\lambda n.$ *f n* ∗ *g n*)
proof
interpret *f*: *completely-multiplicative-function f* **by** *fact*
interpret *g*: *completely-multiplicative-function g* **by** *fact*
show *f 0* ∗ *g 0* = *0 f* (*Suc 0*) ∗ *g* (*Suc 0*) = *1* **by** *simp-all*
fix *a b* :: *nat* **assume** *a* > *1 b* > *1*
thus *f* (*a* ∗ *b*) ∗ *g* (*a* ∗ *b*) = (*f a* ∗ *g a*) ∗ (*f b* ∗ *g b*)
by (*simp-all add*: *f.mult g.mult mult-ac*)
qed

lemma *completely-multiplicative-function-inverse*:
fixes *f* :: *nat* ⇒ *'a* :: *field*
assumes *completely-multiplicative-function f*
shows *completely-multiplicative-function* ($\lambda n.$ *inverse* (*f n*))
proof
interpret *f*: *completely-multiplicative-function f* **by** *fact*
show *inverse* (*f 0*) = *0 inverse* (*f* (*Suc 0*)) = *1* **by** *simp-all*
fix *a b* :: *nat* **assume** *a* > *1 b* > *1*
thus *inverse* (*f* (*a* ∗ *b*)) = *inverse* (*f a*) ∗ *inverse* (*f b*)
by (*simp-all add*: *f.mult field-simps*)
qed

lemma *completely-multiplicative-function-divide*:
fixes *f* :: *nat* ⇒ *'a* :: *field*
assumes *completely-multiplicative-function f completely-multiplicative-function g*
shows *completely-multiplicative-function* ($\lambda n.$ *f n* / *g n*)
proof −
have *completely-multiplicative-function* ($\lambda n.$ *f n* ∗ *inverse* (*g n*))
by (*intro completely-multiplicative-function-mult completely-multiplicative-function-inverse assms*)
also have ($\lambda n.$ *f n* ∗ *inverse* (*g n*)) = ($\lambda n.$ *f n* / *g n*)
by (*simp add*: *field-simps*)
finally show *?thesis* **.**
qed

lemma (**in** *multiplicative-function*) *completely-multiplicativeI*:
assumes $\bigwedge p\ k.$ *prime p* ⟹ *k* > *0* ⟹ *f* (*p* ^ *k*) = *f p* ^ *k*
shows *completely-multiplicative-function f*
proof
fix *m n* :: *nat* **assume** *mn*: *m* > *1 n* > *1*

define *P* **where** *P* = *prime-factors* (*m* ∗ *n*)
have *f* (*m* ∗ *n*) = (∏ *p*∈*P*. *f* (*p* ^ *multiplicity p* (*m* ∗ *n*)))
using *mn* **by** (*subst prod-prime-factors*) (*auto simp*: *P-def*)
also have ... = (∏ *p*∈*P*. *f p* ^ *multiplicity p* (*m* ∗ *n*))
by (*intro prod.cong*) (*auto simp*: *assms prime-factors-multiplicity P-def*)
also have ... = (∏ *p*∈*P*. *f p* ^ *multiplicity p m* ∗ *f p* ^ *multiplicity p n*)
by (*intro prod.cong refl*, *subst prime-elem-multiplicity-mult-distrib*)
(*use mn* **in** ‹*auto simp*: *P-def prime-factors-multiplicity power-add*›)
also have ... = (∏ *p*∈*P*. *f p* ^ *multiplicity p m*) ∗ (∏ *p*∈*P*. *f p* ^ *multiplicity p n*)
by (*rule prod.distrib*)
also have (∏ *p*∈*P*. *f p* ^ *multiplicity p m*) = (∏ *p*∈*prime-factors m*. *f p* ^ *multiplicity p m*)
unfolding *P-def* **by** (*intro prod.mono-neutral-right dvd-prime-factors finite-set-mset*)
(*use mn* **in** ‹*auto simp*: *prime-factors-multiplicity*›)
also have ... = (∏ *p*∈*prime-factors m*. *f* (*p* ^ *multiplicity p m*))
by (*intro prod.cong*) (*auto simp*: *assms prime-factors-multiplicity*)
also have ... = *f m*
using *mn* **by** (*intro prod-prime-factors* [*symmetric*]) *auto*
also have (∏ *p*∈*P*. *f p* ^ *multiplicity p n*) = (∏ *p*∈*prime-factors n*. *f p* ^ *multiplicity p n*)
unfolding *P-def* **by** (*intro prod.mono-neutral-right dvd-prime-factors finite-set-mset*)
(*use mn* **in** ‹*auto simp*: *prime-factors-multiplicity*›)
also have ... = (∏ *p*∈*prime-factors n*. *f* (*p* ^ *multiplicity p n*))
by (*intro prod.cong*) (*auto simp*: *assms prime-factors-multiplicity*)
also have ... = *f n*
using *mn* **by** (*intro prod-prime-factors* [*symmetric*]) *auto*
finally show *f* (*m* ∗ *n*) = *f m* ∗ *f n* **.**
qed *auto*

2.2 Indicator function

definition *ind* :: (*nat* ⇒ *bool*) ⇒ *nat* ⇒ *'a* :: *semiring-1* **where**
ind P n = (*if n* > *0* ∧ *P n then 1 else 0*)

lemma *ind-0* [*simp*]: *ind P 0* = *0* **by** (*simp add*: *ind-def*)

lemma *ind-nonzero*: *n* > *0* ⟹ *ind P n* = (*if P n then 1 else 0*)
by (*simp add*: *ind-def*)

lemma *ind-True* [*simp*]: *P n* ⟹ *n* > *0* ⟹ *ind P n* = *1*
by (*simp add*: *ind-nonzero*)

lemma *ind-False* [*simp*]: ¬*P n* ⟹ *n* > *0* ⟹ *ind P n* = *0*
by (*simp add*: *ind-nonzero*)

lemma *ind-eq-1-iff*: *ind P n* = *1* ⟷ *n* > *0* ∧ *P n*
by (*simp add*: *ind-def*)

```
lemma ind-eq-0-iff: ind P n = 0 ⟷ n = 0 ∨ ¬P n
  by (simp add: ind-def)

lemma multiplicative-function-ind [intro?]:
  assumes P 1 ⋀a b. a > 1 ⟹ b > 1 ⟹ coprime a b ⟹ P (a ∗ b) ⟷ P a
∧ P b
  shows   multiplicative-function (ind P)
  by standard (insert assms, auto simp: ind-nonzero)

end
```

3 Dirichlet convolution

```
theory Dirichlet-Product
  imports
    Complex-Main
    Multiplicative-Function
begin

lemma sum-coprime-dvd-cong:
  (∑r | r dvd a. ∑s | s dvd b. f r s) = (∑r | r dvd a. ∑s | s dvd b. g r s)
  if coprime a b ⋀r s. coprime r s ⟹ r dvd a ⟹ s dvd b ⟹ f r s = g r s
proof (intro sum.cong)
  fix r s
  assume r ∈ {r. r dvd a} and s ∈ {s. s dvd b}
  then have r dvd a and s dvd b
    by simp-all
  moreover from ‹coprime a b› have coprime r s
    using ‹r dvd a› ‹s dvd b›
    by (auto intro: coprime-imp-coprime dvd-trans)
  ultimately show f r s = g r s
    using that by simp
qed auto

definition dirichlet-prod :: (nat ⇒ 'a :: semiring-0) ⇒ (nat ⇒ 'a) ⇒ nat ⇒ 'a
where
  dirichlet-prod f g = (λn. ∑d | d dvd n. f d ∗ g (n div d))

lemma sum-divisors-code:
  assumes n > (0::nat)
  shows   (∑d | d dvd n. f d) =
            fold-atLeastAtMost-nat (λd acc. if d dvd n then f d + acc else acc) 1 n 0
proof −
  have (λd acc. if d dvd n then f d + acc else acc) = (λd acc. (if d dvd n then f d
else 0) + acc)
    by (simp add: fun-eq-iff)
  hence fold-atLeastAtMost-nat (λd acc. if d dvd n then f d + acc else acc) 1 n 0
=
           fold-atLeastAtMost-nat (λd acc. (if d dvd n then f d else 0) + acc) 1 n 0
```

by (*simp only*:)
also have … = (∑ d = 1..n. if d dvd n then f d else 0)
by (*rule sum-atLeastAtMost-code* [*symmetric*])
also from *assms* **have** … = (∑ d | d dvd n. f d)
by (*intro sum.mono-neutral-cong-right*) (*auto elim*: *dvdE dest*: *dvd-imp-le*)
finally show *?thesis* **..**
qed

lemma *dirichlet-prod-code* [*code*]:
dirichlet-prod f g n = (*if n* = *0 then 0 else*
fold-atLeastAtMost-nat (λ*d acc. if d dvd n then f d* ∗ *g* (*n div d*) + *acc else acc*) *1 n 0*)
unfolding *dirichlet-prod-def* **by** (*simp add*: *sum-divisors-code*)

lemma *dirichlet-prod-0* [*simp*]: *dirichlet-prod f g 0* = *0*
by (*simp add*: *dirichlet-prod-def*)

lemma *dirichlet-prod-Suc-0* [*simp*]: *dirichlet-prod f g* (*Suc 0*) = *f* (*Suc 0*) ∗ *g* (*Suc 0*)
by (*simp add*: *dirichlet-prod-def*)

lemma *dirichlet-prod-cong* [*cong*]:
assumes (⋀*n. n* > *0* ⟹ *f n* = *f′ n*) (⋀*n. n* > *0* ⟹ *g n* = *g′ n*)
shows *dirichlet-prod f g* = *dirichlet-prod f′ g′*
proof
fix *n* :: *nat*
show *dirichlet-prod f g n* = *dirichlet-prod f′ g′ n*
proof (*cases n* = *0*)
case *False*
with *assms* **show** *?thesis* **unfolding** *dirichlet-prod-def*
by (*intro ext sum.cong refl*) (*auto elim*!: *dvdE*)
qed *simp-all*
qed

lemma *dirichlet-prod-altdef1*:
dirichlet-prod f g = (λ*n.* ∑ *d* | *d dvd n. f* (*n div d*) ∗ *g d*)
proof
fix *n* :: *nat*
show *dirichlet-prod f g n* = (∑ *d* | *d dvd n. f* (*n div d*) ∗ *g d*)
proof (*cases n* = *0*)
case *False*
hence *dirichlet-prod f g n* = (∑ *d* | *d dvd n. f* (*n div* (*n div d*)) ∗ *g* (*n div d*))
unfolding *dirichlet-prod-def* **by** (*intro sum.cong refl*) (*auto elim*!: *dvdE*)
also from *False* **have** … = (∑ *d* | *d dvd n. f* (*n div d*) ∗ *g d*)
by (*intro sum.reindex-bij-witness*[*of* - (*div*) *n* (*div*) *n*]) (*auto elim*!: *dvdE*)
finally show *?thesis* **.**
qed *simp*
qed

lemma *dirichlet-prod-altdef2*:
dirichlet-prod f g = (λ*n*. ∑(*r*,*d*) | *r* ∗ *d* = *n*. *f r* ∗ *g d*)
proof
fix *n*
show *dirichlet-prod f g n* = (∑(*r*,*d*) | *r* ∗ *d* = *n*. *f r* ∗ *g d*)
proof (*cases n* = *0*)
case *True*
have (λ*n*::*nat*. (*0*, *n*)) ‘ *UNIV* ⊆ {(*r*,*d*). *r* ∗ *d* = *0*} **by** *auto*
moreover have ¬*finite* ((λ*n*::*nat*. (*0*, *n*)) ‘ *UNIV*)
by (*subst finite-image-iff*) (*auto simp*: *inj-on-def*)
ultimately have *infinite* {(*r*,*d*). *r* ∗ *d* = (*0*::*nat*)}
by (*blast dest*: *finite-subset*)
with *True* **show** *?thesis* **by** *simp*
next
case *False*
have (∑*d* | *d dvd n*. *f d* ∗ *g* (*n div d*)) = (∑*r* | *r dvd n*. (∑*d* | *d* = *n div r*. *f r* ∗ *g d*))
by (*intro sum.cong refl*) *auto*
also from *False* **have** . . . = (∑(*r*,*d*)∈(*SIGMA x*:{*r*. *r dvd n*}. {*d*. *d* = *n div x*}). *f r* ∗ *g d*)
by (*intro sum.Sigma*) *auto*
also from *False* **have** (*SIGMA x*:{*r*. *r dvd n*}. {*d*. *d* = *n div x*}) = {(*r*,*d*). *r* ∗ *d* = *n*}
by *auto*
finally show *?thesis* **by** (*simp add*: *dirichlet-prod-def*)
qed
qed

lemma *dirichlet-prod-commutes*:
dirichlet-prod (*f* :: *nat* ⇒ *′a* :: *comm-semiring-0*) *g* = *dirichlet-prod g f*
proof
fix *n* :: *nat*
show *dirichlet-prod f g n* = *dirichlet-prod g f n*
proof (*cases n* = *0*)
case *False*
have (∑(*r*,*d*) | *r* ∗ *d* = *n*. *f r* ∗ *g d*) = (∑(*d*,*r*) | *r* ∗ *d* = *n*. *f r* ∗ *g d*)
by (*rule sum.reindex-bij-witness* [*of* - λ(*x*,*y*). (*y*,*x*) λ(*x*,*y*). (*y*,*x*)]) *auto*
thus *?thesis* **by** (*simp add*: *dirichlet-prod-altdef2 mult.commute*)
qed (*simp add*: *dirichlet-prod-def*)
qed

lemma *finite-divisors-nat′*: *n* > (*0* :: *nat*) ⟹ *finite* {(*a*,*b*). *a* ∗ *b* = *n*}
by (*rule finite-subset*[*of* - {*0*<..*n*} × {*0*<..*n*}]) *auto*

lemma *dirichlet-prod-assoc-aux1*:
assumes *n* > *0*
shows *dirichlet-prod f* (*dirichlet-prod g h*) *n* =
(∑(*a*, *b*, *c*)∈{(*a*, *b*, *c*). *a* ∗ *b* ∗ *c* = *n*}. *f a* ∗ *g b* ∗ *h c*)
proof −

have *dirichlet-prod f* (*dirichlet-prod g h*) *n* =
($\sum$ *x*∈{(*a*,*b*). *a* ∗ *b* = *n*}. ($\sum$ (*c*,*d*) | *c* ∗ *d* = *snd x*. *f* (*fst x*) ∗ *g c* ∗ *h d*))
by (*auto intro*!: *sum.cong simp*: *dirichlet-prod-altdef2 sum-distrib-left mult.assoc*)
also from *assms* **have** ... = ($\sum$ *x*∈(*SIGMA x*:{(*a*, *b*). *a* ∗ *b* = *n*}. {(*c*, *d*). *c* ∗ *d* = *snd x*}).
case x of (*x*, *c*, *d*) ⇒ *f* (*fst x*) ∗ *g c* ∗ *h d*)
by (*intro sum.Sigma finite-divisors-nat′ ballI*) *auto*
also have ... = ($\sum$ (*a*,*b*,*c*) | *a* ∗ *b* ∗ *c* = *n*. *f a* ∗ *g b* ∗ *h c*)
by (*rule sum.reindex-bij-witness*
[*of* - λ(*a*,*b*,*c*). ((*a*, *b*∗*c*), (*b*,*c*)) λ((*a*,*b*),(*c*,*d*)). (*a*, *c*, *d*)])
(*auto simp*: *mult-ac*)
finally show *?thesis* **.**
qed

lemma *dirichlet-prod-assoc-aux2*:
assumes *n* > *0*
shows *dirichlet-prod* (*dirichlet-prod f g*) *h n* =
($\sum$ (*a*, *b*, *c*)∈{(*a*, *b*, *c*). *a* ∗ *b* ∗ *c* = *n*}. *f a* ∗ *g b* ∗ *h c*)
proof −
have *dirichlet-prod* (*dirichlet-prod f g*) *h n* =
($\sum$ *x*∈{(*a*,*b*). *a* ∗ *b* = *n*}. ($\sum$ (*c*,*d*) | *c* ∗ *d* = *fst x*. *f c* ∗ *g d* ∗ *h* (*snd x*)))
by (*auto intro*!: *sum.cong simp*: *dirichlet-prod-altdef2 sum-distrib-right mult.assoc*)
also from *assms* **have** ... = ($\sum$ *x*∈(*SIGMA x*:{(*a*, *b*). *a* ∗ *b* = *n*}. {(*c*, *d*). *c* ∗ *d* = *fst x*}).
case x of (*x*, *c*, *d*) ⇒ *f c* ∗ *g d* ∗ *h* (*snd x*))
by (*intro sum.Sigma finite-divisors-nat′ ballI*) *auto*
also have ... = ($\sum$ (*a*,*b*,*c*) | *a* ∗ *b* ∗ *c* = *n*. *f a* ∗ *g b* ∗ *h c*)
by (*rule sum.reindex-bij-witness*
[*of* - λ(*a*,*b*,*c*). ((*a*∗*b*, *c*), (*a*,*b*)) λ((*a*,*b*),(*c*,*d*)). (*c*, *d*, *b*)])
(*auto simp*: *mult-ac*)
finally show *?thesis* **.**
qed

lemma *dirichlet-prod-assoc*:
dirichlet-prod (*dirichlet-prod f g*) *h* = *dirichlet-prod f* (*dirichlet-prod g h*)
proof
fix *n* :: *nat*
show *dirichlet-prod* (*dirichlet-prod f g*) *h n* = *dirichlet-prod f* (*dirichlet-prod g h*) *n*
by (*cases n* = *0*) (*simp-all add*: *dirichlet-prod-assoc-aux1 dirichlet-prod-assoc-aux2*)
qed

lemma *dirichlet-prod-const-right* [*simp*]:
assumes *n* > *0*
shows *dirichlet-prod f* (λ*n*. *if n* = *Suc 0 then c else 0*) *n* = *f n* ∗ *c*
proof −
have *dirichlet-prod f* (λ*n*. *if n* = *Suc 0 then c else 0*) *n* =
($\sum$ *d* | *d dvd n*. (*if d* = *n then f n* ∗ *c else 0*))
unfolding *dirichlet-prod-def* **using** *assms*

by (*intro sum.cong refl*) (*auto elim*!: *dvdE split*: *if-splits*)
also have ... = *f n* ∗ *c* **using** *assms* **by** (*subst sum.delta*) *auto*
finally show *?thesis* **.**
qed

lemma *dirichlet-prod-const-left* [*simp*]:
assumes *n* > *0*
shows *dirichlet-prod* (λ*n. if n* = *Suc 0 then c else 0*) *g n* = *c* ∗ *g n*
proof −
have *dirichlet-prod* (λ*n. if n* = *Suc 0 then c else 0*) *g n* =
(∑ *d* | *d dvd n*. (*if d* = *1 then c* ∗ *g n else 0*))
unfolding *dirichlet-prod-def* **using** *assms*
by (*intro sum.cong refl*) (*auto elim*!: *dvdE split*: *if-splits*)
also have ... = *c* ∗ *g n* **using** *assms* **by** (*subst sum.delta*) *auto*
finally show *?thesis* **.**
qed

fun *dirichlet-inverse* :: (*nat* ⇒ '*a* :: *comm-ring-1*) ⇒ '*a* ⇒ *nat* ⇒ '*a* **where**
dirichlet-inverse f i n =
(*if n* = *0 then 0 else if n* = *1 then i*
else −*i* ∗ (∑ *d* | *d dvd n* ∧ *d* < *n*. *f* (*n div d*) ∗ *dirichlet-inverse f i d*))

lemma *dirichlet-inverse-induct* [*case-names 0 1 gt1*]:
P 0 ⟹ *P* (*Suc 0*) ⟹ (⋀*n*. *n* > *1* ⟹ (⋀*k*. *k* < *n* ⟹ *P k*) ⟹ *P n*) ⟹ *P n*
by *induction-schema* (*force*, *rule wf-measure* [*of id*], *simp*)

lemma *dirichlet-inverse-0* [*simp*]: *dirichlet-inverse f i 0* = *0*
by *simp*

lemma *dirichlet-inverse-Suc-0* [*simp*]: *dirichlet-inverse f i* (*Suc 0*) = *i*
by *simp*

declare *dirichlet-inverse.simps* [*simp del*]

lemma *dirichlet-inverse-gt-1*:
n > *1* ⟹ *dirichlet-inverse f i n* =
−*i* ∗ (∑ *d* | *d dvd n* ∧ *d* < *n*. *f* (*n div d*) ∗ *dirichlet-inverse f i d*)
by (*simp add*: *dirichlet-inverse.simps*)

lemma *dirichlet-inverse-cong* [*cong*]:
assumes ⋀*n*. *n* > *0* ⟹ *f n* = *f' n i* = *i' n* = *n'*
shows *dirichlet-inverse f i n* = *dirichlet-inverse f' i' n'*
proof −
have *dirichlet-inverse f i n* = *dirichlet-inverse f' i n*
using *assms*(*1*)
proof (*induction n rule*: *dirichlet-inverse-induct*)
case (*gt1 n*)
have ∗: *dirichlet-inverse f i k* = *dirichlet-inverse f' i k* **if** *k dvd n* ∧ *k* < *n* **for** *k*

using *that* **by** (*intro gt1*) *auto*
have ∗: (∑ *d* | *d dvd n* ∧ *d* < *n*. *f* (*n div d*) ∗ *dirichlet-inverse f i d*) =
(∑ *d* | *d dvd n* ∧ *d* < *n*. *f′* (*n div d*) ∗ *dirichlet-inverse f′ i d*)
by (*intro sum.cong refl*) (*subst gt1.prems*, *auto elim*: *dvdE simp*: ∗)
consider *n* = *0* | *n* = *1* | *n* > *1* **by** *force*
thus *?case*
by *cases* (*insert* ∗, *simp-all add*: *dirichlet-inverse-gt-1* ∗ *cong*: *sum.cong*)
qed *auto*
with *assms*(*2*,*3*) **show** *?thesis* **by** *simp*
qed

lemma *dirichlet-inverse-gt-1′*:
assumes *n* > *1*
shows *dirichlet-inverse f i n* =
−*i* ∗ *dirichlet-prod* (λ*n*. *if n* = *1 then 0 else f n*) (*dirichlet-inverse f i*) *n*
proof −
have *dirichlet-prod* (λ*n*. *if n* = *1 then 0 else f n*) (*dirichlet-inverse f i*) *n* =
(∑ *d* | *d dvd n*. (*if n div d* = *Suc 0 then 0 else f* (*n div d*)) ∗ *dirichlet-inverse f i d*)
by (*simp add*: *dirichlet-prod-altdef1*)
also from *assms* **have** ... = (∑ *d* | *d dvd n* ∧ *d* ≠ *n*. *f* (*n div d*) ∗ *dirichlet-inverse f i d*)
by (*intro sum.mono-neutral-cong-right*) (*auto elim*: *dvdE*)
also from *assms* **have** {*d*. *d dvd n* ∧ *d* ≠ *n*} = {*d*. *d dvd n* ∧ *d* < *n*} **by** (*auto dest*: *dvd-imp-le*)
also from *assms* **have** −*i* ∗ (∑ *d*∈... *f* (*n div d*) ∗ *dirichlet-inverse f i d*) =
dirichlet-inverse f i n
by (*simp add*: *dirichlet-inverse-gt-1*)
finally show *?thesis* **..**
qed

lemma *of-int-dirichlet-prod*:
of-int (*dirichlet-prod f g n*) = *dirichlet-prod* (λ*n*. *of-int* (*f n*)) (λ*n*. *of-int* (*g n*)) *n*
by (*simp add*: *dirichlet-prod-def*)

lemma *of-int-dirichlet-inverse*:
of-int (*dirichlet-inverse f i n*) = *dirichlet-inverse* (λ*n*. *of-int* (*f n*)) (*of-int i*) *n*
proof (*induction n rule*: *dirichlet-inverse-induct*)
case (*gt1 n*)
from *gt1* **have** (*of-int* (*dirichlet-inverse f i n*) :: ′*a*) =
− (*of-int i* ∗ (∑ *d* | *d dvd n* ∧ *d* < *n*. *of-int* (*f* (*n div d*) ∗ *dirichlet-inverse f i d*)))
(**is** - = − (- ∗ *?A*))
by (*simp add*: *dirichlet-inverse-gt-1 of-int-dirichlet-prod*)
also have *?A* = (∑ *d* | *d dvd n* ∧ *d* < *n*. *of-int* (*f* (*n div d*)) ∗
dirichlet-inverse (λ*n*. *of-int* (*f n*)) (*of-int i*) *d*)
by (*intro sum.cong refl*) (*auto simp*: *gt1*)
also have −(*of-int i* ∗ ...) = *dirichlet-inverse* (λ*n*. *of-int* (*f n*)) (*of-int i*) *n*
using *gt1.hyps* **by** (*simp add*: *dirichlet-inverse-gt-1*)

```
  finally show ?case .
qed simp-all

lemma dirichlet-inverse-code [code]:
  dirichlet-inverse f i n = (if n = 0 then 0 else if n = 1 then i else
     −i ∗ fold-atLeastAtMost-nat (λd acc. if d dvd n then f (n div d) ∗
     dirichlet-inverse f i d + acc else acc) 1 (n − 1) 0)
proof −
  consider n = 0 | n = 1 | n > 1 by force
  thus ?thesis
  proof cases
    assume n: n > 1
    have ∗: (λd acc. if d dvd n then f (n div d) ∗ dirichlet-inverse f i d + acc else
acc) =
              (λd acc. (if d dvd n then f (n div d) ∗ dirichlet-inverse f i d else 0) +
acc)
      by (simp add: fun-eq-iff)
    have fold-atLeastAtMost-nat (λd acc. if d dvd n then f (n div d) ∗
           dirichlet-inverse f i d + acc else acc) 1 (n − 1) 0 =
          (∑ d = 1..n − 1. if d dvd n then f (n div d) ∗ dirichlet-inverse f i d else
0)
      by (subst ∗, subst sum-atLeastAtMost-code [symmetric]) simp
   also from n have ... = (∑ d | d dvd n ∧ d < n. f (n div d) ∗ dirichlet-inverse
f i d)
      by (intro sum.mono-neutral-cong-right; cases n)
         (auto dest: dvd-imp-le elim: dvdE simp: Suc-le-eq intro!: Nat.gr0I)
    also from n have −i ∗ ... = dirichlet-inverse f i n
      by (simp add: dirichlet-inverse-gt-1)
    finally show ?thesis using n by simp
  qed auto
qed

lemma dirichlet-prod-inverse:
  assumes f 1 ∗ i = 1
  shows   dirichlet-prod f (dirichlet-inverse f i) = (λn. if n = 1 then 1 else 0)
proof
  fix n :: nat
  consider n = 0 | n = 1 | n > 1 by force
  thus dirichlet-prod f (dirichlet-inverse f i) n = (if n = 1 then 1 else 0)
  proof cases
    assume n: n > 1
    have fin: finite {d. d dvd n ∧ d ≠ n}
      by (rule finite-subset[of - {d. d dvd n}]) (insert n, auto)
    have dirichlet-prod f (dirichlet-inverse f i) n =
           (∑ d | d dvd n. f (n div d) ∗ dirichlet-inverse f i d)
      by (simp add: dirichlet-prod-altdef1)
    also have {d. d dvd n} = insert n {d. d dvd n ∧ d ≠ n} by auto
    also have (∑ d∈... . f (n div d) ∗ dirichlet-inverse f i d) =
                f 1 ∗ dirichlet-inverse f i n +
```

```
            (∑ d | d dvd n ∧ d ≠ n. f (n div d) * dirichlet-inverse f i d)
      using fin n by (subst sum.insert) auto
    also from n have dirichlet-inverse f i n =
      − i * (∑ d | d dvd n ∧ d < n. f (n div d) * dirichlet-inverse f i d)
      by (subst dirichlet-inverse-gt-1) auto
    also from n have {d. d dvd n ∧ d < n} = {d. d dvd n ∧ d ≠ n} by (auto
dest: dvd-imp-le)
    also have f 1 * (− i *
              (∑ d | d dvd n ∧ d ≠ n. f (n div d) * dirichlet-inverse f i d)) =
            −(f 1 * i) *
              (∑ d | d dvd n ∧ d ≠ n. f (n div d) * dirichlet-inverse f i d)
      by (simp add: mult.assoc)
    also have f 1 * i = 1 by fact
    finally show ?thesis using n by simp
  qed (insert assms, simp-all add: dirichlet-prod-def)
qed

lemma dirichlet-prod-inverse':
  assumes f 1 * i = 1
  shows   dirichlet-prod (dirichlet-inverse f i) f = (λn. if n = 1 then 1 else 0)
  using dirichlet-prod-inverse[of f] assms by (simp add: dirichlet-prod-commutes)

lemma dirichlet-inverse-noninvertible:
  assumes f (Suc 0) = (0 :: 'a :: {comm-ring-1}) i = 0
  shows   dirichlet-inverse f i n = 0
  using assms
  by (induction f i n rule: dirichlet-inverse.induct) (auto simp: dirichlet-inverse.simps)

lemma multiplicative-dirichlet-prod:
  assumes multiplicative-function f
  assumes multiplicative-function g
  shows   multiplicative-function (dirichlet-prod f g)
proof −
  interpret f: multiplicative-function f by fact
  interpret g: multiplicative-function g by fact
  show ?thesis
  proof
    fix a b :: nat assume a > 1 b > 1 and coprime: coprime a b
    hence dirichlet-prod f g (a * b) =
            (∑ r | r dvd a. ∑ s | s dvd b. f (r * s) * g (a * b div (r * s)))
      by (simp add: dirichlet-prod-def sum-divisors-coprime-mult)
    also have ... = (∑ r | r dvd a. ∑ s | s dvd b. f r * f s * g (a div r) * g (b div
s))
    using ‹coprime a b› proof (rule sum-coprime-dvd-cong)
      fix r s
      assume coprime r s and r dvd a and s dvd b
      with ‹a > 1› ‹b > 1› have r > 0 s > 0
        by (auto intro: ccontr)
      from ‹coprime r s› have f (r * s) = f r * f s
```

by (*rule f.mult-coprime*)
moreover from ‹*coprime a b*› **have** ‹*coprime* (*a div r*) (*b div s*)›
using ‹*r* > *0*› ‹*s* > *0*› ‹*r dvd a*› ‹*s dvd b*› *dvd-div-iff-mult* [*of r a*] *dvd-div-iff-mult* [*of s b*]
by (*auto dest*: *coprime-imp-coprime dvd-mult-left*)
then have *g* (*a div r* ∗ (*b div s*)) = *g* (*a div r*) ∗ *g* (*b div s*)
by (*rule g.mult-coprime*)
ultimately show *f* (*r* ∗ *s*) ∗ *g* (*a* ∗ *b div* (*r* ∗ *s*)) = *f r* ∗ *f s* ∗ *g* (*a div r*) ∗ *g* (*b div s*)
using ‹*r dvd a*› ‹*s dvd b*› **by** (*simp add*: *div-mult-div-if-dvd ac-simps*)
qed
also have . . . = *dirichlet-prod f g a* ∗ *dirichlet-prod f g b*
unfolding *dirichlet-prod-def* **by** (*simp add*: *sum-product mult-ac*)
finally show *dirichlet-prod f g* (*a* ∗ *b*) = . . . **.**
qed *simp-all*
qed

lemma *multiplicative-dirichlet-prodD1*:
fixes *f g* :: *nat* ⇒ *'a* :: *comm-semiring-1-cancel*
assumes *multiplicative-function* (*dirichlet-prod f g*)
assumes *multiplicative-function f*
assumes [*simp*]: *g 0* = *0*
shows *multiplicative-function g*
proof −
interpret *f*: *multiplicative-function f* **by** *fact*
interpret *fg*: *multiplicative-function dirichlet-prod f g* **by** *fact*
show *?thesis*
proof
have *dirichlet-prod f g* (*Suc 0*) = *1* **by** (*rule fg.Suc-0*)
also have *dirichlet-prod f g* (*Suc 0*) = *g 1* **by** (*subst dirichlet-prod-Suc-0*) *simp*
finally show *g 1* = *1* **by** *simp*
next
fix *a b* :: *nat* **assume** *ab*: *a* > *1 b* > *1 coprime a b*
hence *a* > *0 b* > *0 coprime a b* **by** *simp-all*
thus *g* (*a* ∗ *b*) = *g a* ∗ *g b*
proof (*induction a* ∗ *b arbitrary*: *a b rule*: *less-induct*)
case (*less a b*)
have *dirichlet-prod f g* (*a* ∗ *b*) + *g a* ∗ *g b* =
(∑ *r* | *r dvd a* ∗ *b*. *f r* ∗ *g* (*a* ∗ *b div r*)) + *g a* ∗ *g b*
by (*simp add*: *dirichlet-prod-def*)
also have {*r*. *r dvd a* ∗ *b*} = *insert 1* {*r*. *r dvd a* ∗ *b* ∧ *r* ≠ *1*} **by** *auto*
also have (∑ *r*∈. . . . *f r* ∗ *g* (*a* ∗ *b div r*)) + *g a* ∗ *g b* =
g (*a* ∗ *b*) + ((∑ *r* | *r dvd a* ∗ *b* ∧ *r* ≠ *1*. *f r* ∗ *g* (*a* ∗ *b div r*)) + *g a* ∗ *g b*)
using *less.prems*
by (*subst sum.insert*) (*auto intro*!: *finite-subset*[*OF - finite-divisors-nat'*] *simp*: *add-ac*)
also have (∑ *r* | *r dvd a* ∗ *b* ∧ *r* ≠ *1*. *f r* ∗ *g* (*a* ∗ *b div r*)) =
(∑ *r* | *r dvd a* ∗ *b*. *if r* = *1 then 0 else f r* ∗ *g* (*a* ∗ *b div r*))

```
    using less.prems by (intro sum.mono-neutral-cong-left) (auto intro: finite-divisors-nat')
  also have ... = (∑ r | r dvd a. ∑ d | d dvd b.
                  if r ∗ d = 1 then 0 else f (r ∗ d) ∗ g (a ∗ b div (r ∗ d)))
   using ‹coprime a b› by (rule sum-divisors-coprime-mult)
  also have ... = (∑ r | r dvd a. ∑ d | d dvd b.
                  if r ∗ d = 1 then 0 else f (r ∗ d) ∗ g ((a div r) ∗ (b div d)))
   by (intro sum.cong refl) (auto elim!: dvdE)
  also have ... = (∑ r | r dvd a. ∑ d | d dvd b.
                  if r ∗ d = 1 then 0 else f r ∗ f d ∗ g (a div r) ∗ g (b div d))
  using ‹coprime a b› proof (rule sum-coprime-dvd-cong)
   fix r s
   assume coprime r s and r dvd a and s dvd b
   with ‹a > 0› ‹b > 0› have r > 0 s > 0
    by (auto intro: ccontr)
   from ‹coprime r s› have f: f (r ∗ s) = f r ∗ f s
    by (rule f.mult-coprime)
   show (if r ∗ s = 1 then 0 else f (r ∗ s) ∗ g (a div r ∗ (b div s))) =
    (if r ∗ s = 1 then 0 else f r ∗ f s ∗ g (a div r) ∗ g (b div s))
   proof (cases r ∗ s = 1)
    case True
    then show ?thesis
     by simp
   next
    case False
    with ‹r dvd a› ‹s dvd b› less.prems
    have (a div r) ∗ (b div s) ≠ a ∗ b
     by (intro notI) (auto elim!: dvdE)
    moreover from ‹r dvd a› ‹s dvd b› less.prems
    have (a div r) ∗ (b div s) ≤ a ∗ b
     by (intro dvd-imp-le mult-dvd-mono Nat.gr0I) (auto elim!: dvdE)
    ultimately have (a div r) ∗ (b div s) < a ∗ b
     by arith
    with ‹r dvd a› ‹s dvd b› less.prems
    have g: g ((a div r) ∗ (b div s)) = g (a div r) ∗ g (b div s)
     by (auto intro: less coprime-divisors [OF - - ‹coprime a b›] elim!: dvdE)
    from False show ?thesis
     by (auto simp: less f g ac-simps)
   qed
  qed
  also have ... = (∑ (r,d)∈{r. r dvd a}×{d. d dvd b}.
                if r ∗ d = 1 then 0 else f r ∗ f d ∗ g (a div r) ∗ g (b div d))
   by (simp add: sum.cartesian-product)
  also have ... = (∑ (r1,r2) ∈ {r1. r1 dvd a} × {r2. r2 dvd b} − {(1,1)}.
                  (f r1 ∗ f r2) ∗ g (a div r1) ∗ g (b div r2)) (is - = sum ?f ?A)
  using less.prems by (intro sum.mono-neutral-cong-right) (auto split: if-splits)
  also have ... + g a ∗ g b = ?f (1, 1) + sum ?f ?A by (simp add: add-ac)
  also have ... = sum ?f ({r1. r1 dvd a} × {r2. r2 dvd b}) using less.prems
   by (intro sum.remove [symmetric]) auto
```

```
      also have ... = dirichlet-prod f g a * dirichlet-prod f g b
        by (simp add: sum.cartesian-product sum-product dirichlet-prod-def mult-ac)
      also have g (a * b) + dirichlet-prod f g a * dirichlet-prod f g b =
                  dirichlet-prod f g (a * b) + g (a * b)
        using less.prems by (simp add: fg.mult-coprime add-ac)
      finally show ?case by simp
    qed
  qed simp-all
qed

lemma multiplicative-dirichlet-prodD2:
  fixes f g :: nat ⇒ 'a :: comm-semiring-1-cancel
  assumes multiplicative-function (dirichlet-prod f g)
  assumes multiplicative-function g
  assumes [simp]: f 0 = 0
  shows   multiplicative-function f
proof -
  from assms(1) have multiplicative-function (dirichlet-prod g f)
    by (simp add: dirichlet-prod-commutes)
  from multiplicative-dirichlet-prodD1[OF this assms(2)] show ?thesis by simp
qed

lemma multiplicative-dirichlet-inverse:
  assumes multiplicative-function f
  shows   multiplicative-function (dirichlet-inverse f 1)
proof (rule multiplicative-dirichlet-prodD1[OF - assms])
  interpret multiplicative-function f by fact
  have multiplicative-function (λn. if n = 1 then 1 else 0)
    by standard simp-all
  thus multiplicative-function (dirichlet-prod f (dirichlet-inverse f 1))
    by (subst dirichlet-prod-inverse) simp-all
qed simp-all

lemma dirichlet-prod-prime-power:
  assumes prime p
  shows   dirichlet-prod f g (p ^ k) = (∑i≤k. f (p ^ i) * g (p ^ (k - i)))
proof -
  have dirichlet-prod f g (p ^ k) = (∑i≤k. f (p ^ i) * g (p ^ k div p ^ i))
    unfolding dirichlet-prod-def using assms
    by (intro sum.reindex-bij-betw [symmetric] bij-betw-prime-power-divisors)
  also from assms have ... = (∑i≤k. f (p ^ i) * g (p ^ (k - i)))
    by (intro sum.cong refl) (auto simp: power-diff)
  finally show ?thesis .
qed

lemma dirichlet-prod-prime:
  assumes prime p
  shows   dirichlet-prod f g p  = f 1 * g p + f p * g 1
  using dirichlet-prod-prime-power[of p f g 1] assms by simp
```

```
locale multiplicative-dirichlet-prod =
  f: multiplicative-function f + g: multiplicative-function g
  for f g :: nat ⇒ 'a :: comm-semiring-1
begin

sublocale multiplicative-function dirichlet-prod f g
  by (intro multiplicative-dirichlet-prod
       f.multiplicative-function-axioms g.multiplicative-function-axioms)

end

locale multiplicative-dirichlet-prod' =
  f: multiplicative-function' f f-prime-power f-prime +
  g: multiplicative-function' g g-prime-power g-prime
  for f g :: nat ⇒ 'a :: comm-semiring-1 and f-prime-power g-prime-power f-prime
g-prime
begin

sublocale multiplicative-dirichlet-prod f g ..

sublocale multiplicative-function' dirichlet-prod f g
  λp k. f-prime-power p k + g-prime-power p k +
     (∑ i∈{0<..<k}. f-prime-power p i ∗ g-prime-power p (k − i))
  λp. f-prime p + g-prime p
proof (standard, goal-cases)
  case (1 p k)
  hence dirichlet-prod f g (p ^ k) = (∑ i≤k. f (p ^ i) ∗ g (p ^ (k − i)))
    by (intro dirichlet-prod-prime-power)
  also from 1 have {..k} = insert 0 (insert k {0<..<k}) by auto
  also have (∑ i∈. . . . f (p ^ i) ∗ g (p ^ (k − i))) =
              f-prime-power p k + g-prime-power p k +
              (∑ i∈{0<..<k}. f (p ^ i) ∗ g (p ^ (k − i))) using 1
    by (auto simp: f.prime-power g.prime-power add-ac)
  also have (∑ i∈{0<..<k}. f (p ^ i) ∗ g (p ^ (k − i))) =
              (∑ i∈{0<..<k}. f-prime-power p i ∗ g-prime-power p (k − i))
    using 1 by (intro sum.cong) (auto simp: f.prime-power g.prime-power)
  finally show ?case .
next
  case (2 p)
  have {0<..<Suc 0} = {} by auto
  with 2 show ?case
    by (auto simp: f.prime-power [symmetric] g.prime-power [symmetric] f.prime
g.prime add-ac)
qed

end

end
```

4 Formal Dirichlet series

theory *Dirichlet-Series*
imports
  Complex-Main
  Dirichlet-Product
  Multiplicative-Function
  HOL−Computational-Algebra.Computational-Algebra
  HOL−Number-Theory.Number-Theory
  HOL−Library.FuncSet
begin

A formal Dirichlet series

$$A(s) = \sum_{n=1}^{\infty} \frac{a_n}{n^s}$$

is represented its coefficient sequence starting from 1. For simplicity, we represent this in Isabelle with a function of type $nat \Rightarrow 'a$ whose value for n is the $n+1$-th coefficient.

typedef *'a fds* = *UNIV* :: (*nat* ⇒ *'a*) *set*
  by *simp*

setup-lifting *type-definition-fds*

lift-definition *fds-nth* :: *'a fds* ⇒ *nat* ⇒ *'a* :: *zero* **is**
  λ*f*::*nat* ⇒ *'a*. *case-nat 0 f* **.**

lift-definition *fds* :: (*nat* ⇒ *'a*) ⇒ *'a fds* **is**
  λ*f*. *f* ∘ *Suc* **.**

lemma *fds-nth-fds*: *fds-nth* (*fds f*) *n* = (*if n* = *0 then 0 else f n*)
  by *transfer* (*simp split*: *nat.splits*)

lemma *fds-nth-fds'*: *f 0* = *0* ⟹ *fds-nth* (*fds f*) = *f*
  by (*simp add*: *fun-eq-iff fds-nth-fds*)

lemma *fds-nth-0* [*simp*]: *fds-nth f 0* = *0*
  by *transfer simp*

lemma *fds-nth-fds-pos* [*simp*]: *n* > *0* ⟹ *fds-nth* (*fds f*) *n* = *f n*
  by *transfer* (*simp split*: *nat.splits*)

lemma *fds-fds-nth* [*simp*]: *fds* (*fds-nth f*) = *f*
  by *transfer* (*simp add*: *fun-eq-iff split*: *nat.splits*)

lemma *fds-eq-fds-iff*:
  fds f = *fds g* ⟷ (∀ *n*>*0*. *f n* = *g n*)
proof *transfer*
  fix *f g* :: *nat* ⇒ *'a*

have (*f* ∘ *Suc* = *g* ∘ *Suc*) ⟷ (∀ *n*. *f* (*Suc n*) = *g* (*Suc n*)) **by** (*auto simp*: *fun-eq-iff*)
also have … ⟷ (∀ *n*>*0*. *f n* = *g n*)
proof *safe*
fix *n* :: *nat* **assume** ∀ *n*. *f* (*Suc n*) = *g* (*Suc n*) *n* > *0*
thus *f n* = *g n* **by** (*cases n*) *auto*
qed *auto*
finally show (*f* ∘ *Suc* = *g* ∘ *Suc*) = (∀ *n*>*0*. *f n* = *g n*) **.**
qed

lemma *fds-eq-fds-iff'*: *f 0* = *g 0* ⟹ *fds f* = *fds g* ⟷ *f* = *g*
proof *safe*
assume *f 0* = *g 0* *fds f* = *fds g*
hence *f n* = *g n* **for** *n* **by** (*cases n*) (*auto simp*: *fds-eq-fds-iff*)
thus *f* = *g* **by** (*simp add*: *fun-eq-iff*)
qed

lemma *fds-eqI* [*intro?*]:
assumes (⋀*n*. *n* > *0* ⟹ *fds-nth f n* = *fds-nth g n*)
shows *f* = *g*
proof −
from *assms* **have** *fds-nth f n* = *fds-nth g n* **if** *n* > *0* **for** *n*
by (*cases n*) (*simp-all add*: *fun-eq-iff*)
hence *fds* (*fds-nth f*) = *fds* (*fds-nth g*) **by** (*subst fds-eq-fds-iff*) *auto*
thus *?thesis* **by** *simp*
qed

lemma *fds-cong* [*cong*]: (⋀*n*. *n* > *0* ⟹ *f n* = (*g n* :: *'a* :: *zero*)) ⟹ *fds f* = *fds g*
by (*rule fds-eqI*) *simp*

lemma *fds-eq-iff*: *f* = *g* ⟷ (∀ *n*>*0*. *fds-nth f n* = *fds-nth g n*)
by (*auto intro*: *fds-eqI*)

lemma *dirichlet-prod-fds-nth-fds-left* [*simp*]:
dirichlet-prod (*fds-nth* (*fds f*)) *g* = *dirichlet-prod f g*
by (*simp add*: *fds-nth-fds*)

lemma *dirichlet-prod-fds-nth-fds-right* [*simp*]:
dirichlet-prod f (*fds-nth* (*fds g*)) = *dirichlet-prod f g*
by (*simp add*: *fds-nth-fds*)

definition *fds-const* :: *'a* :: *zero* ⇒ *'a fds* **where**
fds-const c = *fds* (λ*n*. *if n* = *1 then c else 0*)

abbreviation *fds-ind* **where** *fds-ind P* ≡ *fds* (*ind P*)

```
bundle fds-syntax
begin

notation fds-nth (infixl ‹$› 75)
notation fds (binder ‹χ› 10)
notation dirichlet-prod (infixl ‹⋆› 70)

end

instantiation fds :: (zero) zero
begin
definition zero-fds :: 'a fds where zero-fds = fds (λ-. 0)
instance ..
end

instantiation fds :: ({zero,one}) one
begin
definition one-fds :: 'a fds where one-fds = fds (λn. if n = 1 then 1 else 0)
instance ..
end

instantiation fds :: ({plus,zero}) plus
begin
definition plus-fds :: 'a fds ⇒ 'a fds ⇒ 'a fds
  where plus-fds f g = fds (λn. fds-nth f n + fds-nth g n)
instance ..
end

instantiation fds :: (semiring-0) times
begin
definition times-fds :: 'a fds ⇒ 'a fds ⇒ 'a fds
  where times-fds f g = fds (dirichlet-prod (fds-nth f) (fds-nth g))
instance ..
end

instantiation fds :: ({uminus,zero}) uminus
begin
definition uminus-fds :: 'a fds ⇒ 'a fds
  where uminus-fds f = fds (λn. −fds-nth f n)
instance ..
end

instantiation fds :: ({minus,zero}) minus
begin
definition minus-fds :: 'a fds ⇒ 'a fds ⇒ 'a fds
  where minus-fds f g = fds (λn. fds-nth f n − fds-nth g n)
instance ..
end
```

4.1 General properties

lemma *fds-nth-zero* [*simp*]: *fds-nth 0* = (λ-. *0*)
 by (*simp add*: *zero-fds-def fds-nth-fds fun-eq-iff*)

lemma *fds-nth-one*: *fds-nth 1* = (λ*n*. *if n* = *1 then 1 else 0*)
 by (*simp add*: *one-fds-def fds-nth-fds fun-eq-iff*)

lemma *fds-nth-one-Suc-0* [*simp*]: *fds-nth 1* (*Suc 0*) = *1*
 by (*simp add*: *fds-nth-one*)

lemma *fds-nth-one-not-Suc-0* [*simp*]: *n* ≠ *Suc 0* ⟹ *fds-nth 1 n* = *0*
 by (*simp add*: *fds-nth-one*)

lemma *fds-nth-plus* [*simp*]:
 fds-nth (*f* + *g*) = (λ*n*. *fds-nth f n* + *fds-nth g n* :: *'a* :: *monoid-add*)
 by (*simp add*: *plus-fds-def fds-nth-fds fun-eq-iff*)

lemma *fds-nth-minus* [*simp*]:
 fds-nth (*f* − *g*) = (λ*n*. *fds-nth f n* − *fds-nth g n* :: *'a* :: {*cancel-comm-monoid-add*})
 by (*simp add*: *minus-fds-def fds-nth-fds fun-eq-iff*)

lemma *fds-nth-uminus* [*simp*]: *fds-nth* (−*g*) = (λ*n*. − *fds-nth g n* :: *'a* :: *group-add*)
 by (*simp add*: *uminus-fds-def fds-nth-fds fun-eq-iff*)

lemma *fds-nth-mult*: *fds-nth* (*f* ∗ *g*) = *dirichlet-prod* (*fds-nth f*) (*fds-nth g*)
 by (*simp add*: *times-fds-def fds-nth-fds dirichlet-prod-def fun-eq-iff*)

lemma *fds-nth-mult-const-left* [*simp*]: *fds-nth* (*fds-const c* ∗ *f*) *n* = *c* ∗ *fds-nth f n*
 by (*cases n* = *0*) (*simp-all add*: *fds-nth-mult fds-const-def*)

lemma *fds-nth-mult-const-right* [*simp*]: *fds-nth* (*f* ∗ *fds-const c*) *n* = *fds-nth f n* ∗ *c*
 by (*cases n* = *0*) (*simp-all add*: *fds-nth-mult fds-const-def*)

instance *fds* :: ({*semigroup-add*, *zero*}) *semigroup-add*
 by *standard* (*simp-all add*: *fds-eq-iff algebra-simps plus-fds-def*)

instance *fds* :: ({*ab-semigroup-add*, *zero*}) *ab-semigroup-add*
 by *standard* (*simp-all add*: *fds-eq-iff algebra-simps plus-fds-def*)

instance *fds* :: ({*cancel-semigroup-add*, *zero*}) *cancel-semigroup-add*
 by *standard* (*simp-all add*: *fds-eq-iff algebra-simps plus-fds-def*)

instance *fds* :: ({*cancel-ab-semigroup-add*, *zero*}) *cancel-ab-semigroup-add*
 by *standard* (*simp-all add*: *fds-eq-iff algebra-simps plus-fds-def minus-fds-def*)

instance *fds* :: (*monoid-add*) *monoid-add*
 by *standard* (*simp-all add*: *fds-eq-iff algebra-simps*)

instance *fds* :: (*comm-monoid-add*) *comm-monoid-add*
 by *standard* (*simp-all add*: *fds-eq-iff algebra-simps*)

instance *fds* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add*
 by *standard* (*simp-all add*: *fds-eq-iff algebra-simps*)

instance *fds* :: (*group-add*) *group-add*
 by *standard* (*simp-all add*: *fds-eq-iff algebra-simps minus-fds-def*)

instance *fds* :: (*ab-group-add*) *ab-group-add*
 by *standard* (*simp-all add*: *fds-eq-iff algebra-simps*)

instance *fds* :: (*semiring-0*) *semiring-0*
proof
 fix *f g h* :: *'a fds*
 show $(f + g) * h = f * h + g * h$
  by (*simp add*: *fds-eq-iff fds-nth-mult dirichlet-prod-def algebra-simps sum.distrib*)
next
 fix *f g h* :: *'a fds*
 show $f * g * h = f * (g * h)$
  by (*intro fds-eqI*) (*simp add*: *fds-nth-mult dirichlet-prod-assoc*)
qed (*simp-all add*: *fds-eq-iff fds-nth-mult dirichlet-prod-def algebra-simps sum.distrib*)

instance *fds* :: (*comm-semiring-0*) *comm-semiring-0*
proof
 fix *f g* :: *'a fds*
 show $f * g = g * f$
  by (*simp add*: *fds-eq-iff fds-nth-mult dirichlet-prod-commutes*)
qed (*simp-all add*: *fds-eq-iff fds-nth-mult dirichlet-prod-def algebra-simps sum.distrib*)

instance *fds* :: (*semiring-0-cancel*) *semiring-0-cancel*
 by *standard* (*simp-all add*: *fds-eq-iff fds-nth-one fds-nth-mult*)

instance *fds* :: (*comm-semiring-0-cancel*) *comm-semiring-0-cancel* **..**

instance *fds* :: (*semiring-1*) *semiring-1*
 by *standard* (*simp-all add*: *fds-eq-iff fds-nth-one fds-nth-mult*)

instance *fds* :: (*comm-semiring-1*) *comm-semiring-1*
 by *standard* (*simp-all add*: *fds-eq-iff fds-nth-one fds-nth-mult*)

instance *fds* :: (*semiring-1-cancel*) *semiring-1-cancel* **..**
instance *fds* :: (*ring*) *ring* **..**
instance *fds* :: (*ring-1*) *ring-1* **..**
instance *fds* :: (*comm-ring*) *comm-ring* **..**

instance *fds* :: (*semiring-no-zero-divisors*) *semiring-no-zero-divisors*
proof

fix *f g* :: *'a fds*
assume $f \neq 0$ $g \neq 0$
hence *ex*: $\exists m>0.$ *fds-nth f m* $\neq 0$ $\exists n>0.$ *fds-nth g n* $\neq 0$
by (*auto simp*: *fds-eq-iff*)
define *m* **where** $m = (LEAST\ m.\ m > 0 \wedge$ *fds-nth f m* $\neq 0)$
define *n* **where** $n = (LEAST\ n.\ n > 0 \wedge$ *fds-nth g n* $\neq 0)$
from *ex*[*THEN LeastI-ex*, *folded m-def n-def*]
have *mn*: $m > 0$ *fds-nth f m* $\neq 0$ $n > 0$ *fds-nth g n* $\neq 0$ **by** *auto*

have $*$: $m \leq m'$ **if** $m' > 0$ *fds-nth f m'* $\neq 0$ **for** m'
using *conjI*[*OF that*] **unfolding** *m-def* **by** (*rule Least-le*)
have m': *fds-nth f m'* $= 0$ **if** $m' \in \{0<..<m\}$ **for** m' **using** *that* $*$[*of m'*] **by** *auto*

have $*$: $n \leq n'$ **if** $n' > 0$ *fds-nth g n'* $\neq 0$ **for** n'
using *conjI*[*OF that*] **unfolding** *n-def* **by** (*rule Least-le*)
have n': *fds-nth g n'* $= 0$ **if** $n' \in \{0<..<n\}$ **for** n' **using** *that* $*$[*of n'*] **by** *auto*

have *fds-nth* $(f * g)$ $(m * n)$ =
($\sum d \mid d$ *dvd* $m * n$. *fds-nth f d* $*$ *fds-nth g* $(m * n$ *div* $d)$)
by (*simp add*: *fds-nth-mult dirichlet-prod-def*)
also have ... = ($\sum d \mid d$ *dvd* $m * n$. *if* $d = m$ *then fds-nth f m* $*$ *fds-nth g n* *else 0*)
proof (*intro sum.cong refl*, *goal-cases*)
case (*1 d*)
thus *?case*
proof (*cases* $d \leq m$)
case *True*
with *mn*(*1*,*3*) *1* **show** *?thesis* **by** (*auto elim*!: *dvdE simp*: *m' n' split*: *if-splits*)
next
case *False*
from *1* **obtain** *k* **where** *k*: $m * n = d * k$ **by** (*auto elim*!: *dvdE*)
with *mn*(*1*,*3*) **have** [*simp*]: $k > 0$ **by** (*auto intro*!: *Nat.gr0I*)
from *False mn*(*3*) **have** $m * n < d * n$ **by** (*intro mult-strict-right-mono*) *auto*
also note *k*
finally have $k < n$ **by** (*subst* (*asm*) *mult-less-cancel1*) *auto*
with *mn*(*1*,*3*) **and** *1* **and** *False* **show** *?thesis*
by (*auto simp*: *k m' n' split*: *if-splits*)
qed
qed
also have ... = *fds-nth f m* $*$ *fds-nth g n* **using** *mn*(*1*,*3*) **by** (*subst sum.delta*) *auto*
also have ... $\neq 0$ **using** *mn* **by** *auto*
finally show $f * g \neq 0$ **by** *auto*
qed

instance *fds* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors* **..**
instance *fds* :: (*idom*) *idom* **..**

instantiation *fds* :: (*real-vector*) *real-vector*
begin

definition *scaleR-fds* :: *real* ⇒ *'a fds* ⇒ *'a fds* **where**
scaleR-fds c f = *fds* (λ*n*. *c* $*_R$ *fds-nth f n*)

lemma *fds-nth-scaleR* [*simp*]: *fds-nth* (*c* $*_R$ *f*) = (λ*n*. *c* $*_R$ *fds-nth f n*)
by (*simp add*: *scaleR-fds-def fun-eq-iff fds-nth-fds*)

instance by *standard* (*simp-all add*: *fds-eq-iff algebra-simps*)

end

instance *fds* :: (*real-algebra*) *real-algebra*
by *standard* (*simp-all add*: *fds-eq-iff algebra-simps fds-nth-mult*
dirichlet-prod-def scaleR-sum-right)

instance *fds* :: (*real-algebra-1*) *real-algebra-1* **..**

lemma *fds-nth-sum* [*simp*]: *fds-nth* (*sum f A*) *n* = *sum* (λ*x*. *fds-nth* (*f x*) *n*) *A*
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *sum-fds* [*simp*]: (∑*x*∈*A*. *fds* (*f x*)) = *fds* (λ*n*. ∑*x*∈*A*. *f x n*)
by (*rule fds-eqI*) *simp-all*

lemma *fds-nth-const*: *fds-nth* (*fds-const c*) = (λ*n*. *if n* = *1 then c else 0*)
by (*simp add*: *fds-const-def fds-nth-fds fun-eq-iff*)

lemma *fds-nth-const-Suc-0* [*simp*]: *fds-nth* (*fds-const c*) (*Suc 0*) = *c*
by (*simp add*: *fds-nth-const*)

lemma *fds-nth-const-not-Suc-0* [*simp*]: *n* ≠ *1* ⟹ *fds-nth* (*fds-const c*) *n* = *0*
by (*simp add*: *fds-nth-const*)

lemma *fds-const-zero* [*simp*]: *fds-const 0* = *0*
by (*simp add*: *fds-eq-iff fds-nth-const*)

lemma *fds-const-one* [*simp*]: *fds-const 1* = *1*
by (*simp add*: *fds-eq-iff fds-nth-const fds-nth-one*)

lemma *fds-const-add* [*simp*]: *fds-const* (*a* + *b* :: *'a* :: *monoid-add*) = *fds-const a* + *fds-const b*
by (*simp add*: *fds-eq-iff fds-nth-const*)

lemma *fds-const-minus* [*simp*]:
fds-const (*a* − *b* :: *'a* :: *cancel-comm-monoid-add*) = *fds-const a* − *fds-const b*

by (*simp add*: *fds-eq-iff fds-nth-const*)

lemma *fds-const-uminus* [*simp*]:
fds-const (− *b* :: ′*a* :: *ab-group-add*) = − *fds-const b*
by (*simp add*: *fds-eq-iff fds-nth-const*)

lemma *fds-const-mult* [*simp*]:
fds-const (*a* ∗ *b* :: ′*a* :: *semiring-0*) = *fds-const a* ∗ *fds-const b*
by (*simp add*: *fds-eq-iff fds-nth-const fds-nth-mult*)

lemma *fds-const-of-nat* [*simp*]: *fds-const* (*of-nat c*) = *of-nat c*
by (*induction c*) (*simp-all*)

lemma *fds-const-of-int* [*simp*]: *fds-const* (*of-int c*) = *of-int c*
by (*cases c*) *simp-all*

lemma *fds-const-of-real* [*simp*]: *fds-const* (*of-real c*) = *of-real c*
by (*simp add*: *of-real-def fds-eq-iff fds-const-def fds-nth-one*)

instantiation *fds* :: ({*inverse*, *comm-ring-1*}) *inverse*
begin

definition *inverse-fds* :: ′*a fds* ⇒ ′*a fds* **where**
inverse-fds f = *fds* (λ*n*. *dirichlet-inverse* (*fds-nth f*) (*inverse* (*fds-nth f 1*)) *n*)

definition *divide-fds* :: ′*a fds* ⇒ ′*a fds* ⇒ ′*a fds* **where**
divide-fds f g = *f* ∗ *inverse g*

instance ..

end

lemma *numeral-fds*: *numeral n* = *fds-const* (*numeral n*)
proof −
have *numeral n* = (*of-nat* (*numeral n*) :: ′*a fds*) **by** *simp*
also have . . . = *fds-const* (*of-nat* (*numeral n*)) **by** (*rule fds-const-of-nat* [*symmetric*])
also have *of-nat* (*numeral n*) = (*numeral n* :: ′*a*) **by** *simp*
finally show *?thesis* **.**
qed

lemma *fds-ind-False* [*simp*]: *fds-ind* (λ-. *False*) = *0*
by (*rule fds-eqI*) *simp*

lemma *fds-commutes*:
assumes ⋀*m n*. *m* > *0* =⇒ *n* > *0* =⇒ *fds-nth f m* ∗ *fds-nth g n* = *fds-nth g n* ∗ *fds-nth f m*
shows *f* ∗ *g* = *g* ∗ *f*
by (*intro fds-eqI*, *unfold fds-nth-mult*, *subst dirichlet-prod-def*,

subst dirichlet-prod-altdef1, intro sum.cong refl assms) (*auto elim*: *dvdE*)

lemma *fds-nth-mult-Suc-0* [*simp*]:
fds-nth (*f* ∗ *g*) (*Suc 0*) = *fds-nth f* (*Suc 0*) ∗ *fds-nth g* (*Suc 0*)
by (*simp add*: *fds-nth-mult*)

lemma *fds-nth-inverse*:
fds-nth (*inverse f*) = *dirichlet-inverse* (*fds-nth f*) (*inverse* (*fds-nth f 1*))
by (*simp add*: *inverse-fds-def fds-nth-fds fun-eq-iff*)

lemma *inverse-fds-nonunit*:
fds-nth f 1 = (*0* :: *'a* :: *field*) ⟹ *inverse f* = *0*
by (*auto simp*: *fds-eq-iff fds-nth-inverse dirichlet-inverse-noninvertible*)

lemma *inverse-0-fds* [*simp*]: *inverse* (*0* :: *'a* :: *field fds*) = *0*
by (*simp add*: *inverse-fds-def fds-eq-iff dirichlet-inverse.simps*)

lemma *fds-left-inverse*:
fds-nth f 1 ≠ (*0* :: *'a* :: *field*) ⟹ *inverse f* ∗ *f* = *1*
by (*auto simp*: *fds-eq-iff fds-nth-mult fds-nth-inverse dirichlet-prod-inverse' fds-nth-one*)

lemma *fds-right-inverse*:
fds-nth f 1 ≠ (*0* :: *'a* :: *field*) ⟹ *f* ∗ *inverse f* = *1*
by (*auto simp*: *fds-eq-iff fds-nth-mult fds-nth-inverse dirichlet-prod-inverse fds-nth-one*)

lemma *fds-left-inverse-unique*:
assumes *f* ∗ *g* = (*1* :: *'a* :: *field fds*)
shows *f* = *inverse g*
proof −
have *fds-nth* (*f* ∗ *g*) *1* = *1* **by** (*subst assms*) *simp*
hence *fds-nth g 1* ≠ *0* **by** *auto*
hence (*f* − *inverse g*) ∗ *g* = *0*
unfolding *ring-distribs* **by** (*subst fds-left-inverse*) (*simp-all add*: *assms*)
moreover from *assms* **have** *g* ≠ *0* **by** *auto*
ultimately show *f* = *inverse g* **by** *simp*
qed

lemma *fds-right-inverse-unique*:
assumes *f* ∗ *g* = (*1* :: *'a* :: *field fds*)
shows *g* = *inverse f*
using *fds-left-inverse-unique*[*of g f*] *assms* **by** (*simp add*: *mult.commute*)

lemma *inverse-1-fds* [*simp*]: *inverse* (*1* :: *'a* :: *field fds*) = *1*
by (*rule fds-left-inverse-unique* [*symmetric*]) *simp*

lemma *inverse-const-fds* [*simp*]:
inverse (*fds-const c* :: *'a* :: *field fds*) = *fds-const* (*inverse c*)
proof (*cases c* = *0*)
case *False*

thus *?thesis*
  by (*intro fds-right-inverse-unique*[*symmetric*])
    (*auto simp del*: *fds-const-mult simp*: *fds-const-mult* [*symmetric*])
qed *auto*

lemma *inverse-mult-fds*: *inverse* ($f * g$ :: $'a$ :: *field fds*) = *inverse f* $*$ *inverse g*
proof (*cases fds-nth* ($f * g$) (*Suc 0*) = *0*)
  case *False*
  hence (f $*$ *inverse f*) $*$ (g $*$ *inverse g*) = *1* **by** (*subst* (*1 2*) *fds-right-inverse*) *auto*
  thus *?thesis* **by** (*intro fds-right-inverse-unique* [*symmetric*]) (*simp-all add*: *mult-ac*)
qed (*auto simp*: *inverse-fds-nonunit*)

definition *fds-zeta* :: $'a$ :: *one fds*
  where *fds-zeta* = *fds* (λ-. *1*)

lemma *fds-zeta-altdef*: *fds-zeta* = *fds* (λn. *if n* = *0 then 0 else 1*)
  by (*rule fds-eqI*) (*simp add*: *fds-zeta-def*)

lemma *fds-nth-zeta*: *fds-nth fds-zeta* = (λn. *if n* = *0 then 0 else 1*)
  by (*simp add*: *fds-zeta-def fun-eq-iff*)

lemma *fds-nth-zeta-pos* [*simp*]: $n > 0 \Longrightarrow$ *fds-nth fds-zeta n* = *1*
  by (*simp add*: *fds-nth-zeta*)

lemma *fds-zeta-commutes*: *fds-zeta* $*$ (f :: $'a$ :: *semiring-1 fds*) = f $*$ *fds-zeta*
  by (*intro fds-commutes*) *simp-all*

lemma *fds-ind-True* [*simp*]: *fds-ind* (λ-. *True*) = *fds-zeta*
  by (*rule fds-eqI*) *simp*

lemma *finite-extensional-prod-nat*:
  assumes *finite A* $b > 0$
  shows *finite* {$d \in$ *extensional A*. *prod d A* = (b :: *nat*)}
proof (*rule finite-subset*)
  from *assms*(*1*) **show** *finite* (*PiE A* (λ-. {..b})) **by** (*rule finite-PiE*) *auto*
  {
    fix d :: $'a \Rightarrow$ *nat* **and** x :: $'a$ **assume** $*$: $x \in A$ *prod d A* = b
    with *prod-dvd-prod-subset*[*of A* {x} *d*] *assms* **have** *d x dvd b* **by** *auto*
    with *assms* **have** $d\ x \le b$ **by** (*auto dest*: *dvd-imp-le*)
  }
  thus {$d \in$ *extensional A*. *prod d A* = (b :: *nat*)} $\subseteq \ldots$
    by (*auto simp*: *extensional-def*)
qed

The n-th coefficient of a product of Dirichlet series can be determined by summing over all products of k_i-th coefficients of the series such that the product of the k_i is n.

lemma *fds-nth-prod*:
  assumes *finite A* $A \neq \{\}$ $n > 0$
  shows *fds-nth* $(\prod x \in A.\ f\ x)\ n =$
    $(\sum d \mid d \in extensional\ A \wedge prod\ d\ A = n.\ \prod x \in A.\ fds\text{-}nth\ (f\ x)\ (d\ x))$
using *assms*
proof (*induction arbitrary*: *n rule*: *finite-ne-induct*)
  case (*singleton x n*)
  have $\{d \in extensional\ \{x\}.\ d\ x = n\} = \{\lambda y.\ if\ y = x\ then\ n\ else\ undefined\}$
    by (*auto simp*: *extensional-def*)
  thus *?case* **by** *simp*
next
  case (*insert x A n*)
  let $?f = \lambda d.\ ((d\ x,\ n\ div\ d\ x),\ d(x := undefined))$
  let $?g = \lambda(z,d).\ d(x := fst\ z)$
  from *insert* **have** *fds-nth* $(\prod x \in insert\ x\ A.\ f\ x)\ n =$
    $(\sum z \mid fst\ z * snd\ z = n.\ \sum d \mid d \in extensional\ A \wedge prod\ d\ A = snd\ z.$
      $fds\text{-}nth\ (f\ x)\ (fst\ z) * (\prod x \in A.\ fds\text{-}nth\ (f\ x)\ (d\ x)))$
   by (*simp add*: *fds-nth-mult dirichlet-prod-altdef2 sum-distrib-left case-prod-unfold*)
  also have ... $= (\sum (z,d) \in (SIGMA\ x{:}\{z.\ fst\ z * snd\ z = n\}.\ \{d \in extensional$
$A.\ prod\ d\ A = snd\ x\}).$
      $fds\text{-}nth\ (f\ x)\ (fst\ z) * (\prod x \in A.\ fds\text{-}nth\ (f\ x)\ (d\ x)))$
    using *finite-divisors-nat'*[*of n*] **and** *insert.hyps* **and** ‹$n > 0$›
   by (*intro sum.Sigma finite-extensional-prod-nat ballI*) (*auto simp*: *case-prod-unfold*)
  also have ... $= (\sum d \mid d \in extensional\ (insert\ x\ A) \wedge prod\ d\ (insert\ x\ A) = n.$
      $(\prod x \in insert\ x\ A.\ fds\text{-}nth\ (f\ x)\ (d\ x)))$
  proof (*rule sum.reindex-bij-witness* [*of - ?f ?g*], *goal-cases*)
    case (*1 z*)
    thus *?case* **using** *insert.hyps insert.prems* **by** (*auto simp*: *extensional-def*)
  next
    case (*2 z*)
    thus *?case* **using** *insert.hyps insert.prems*
      by (*auto simp*: *extensional-def sum.delta intro*!: *prod.cong*)
  next
    case (*4 z*)
    thus *?case* **using** *insert.hyps insert.prems* **by** (*auto intro*!: *prod.cong*)
  next
    case (*5 z*)
    with *insert.hyps insert.prems*
      have $(\prod xa \in A.\ fds\text{-}nth\ (f\ xa)\ (if\ xa = x\ then\ fst\ (fst\ z)\ else\ snd\ z\ xa)) =$
        $(\prod x \in A.\ fds\text{-}nth\ (f\ x)\ (snd\ z\ x))$ **by** (*intro prod.cong*) *auto*
    with *5 insert.hyps insert.prems* **show** *?case* **by** (*simp add*: *case-prod-unfold*)
  qed *auto*
  finally show *?case* **.**
qed

lemma *fds-nth-power-Suc-0* [*simp*]: $fds\text{-}nth\ (f\ \hat{}\ n)\ (Suc\ 0) = fds\text{-}nth\ f\ (Suc\ 0)\ \hat{}$
n
  by (*induction n*) *simp-all*

```
lemma fds-nth-prod-Suc-0 [simp]: fds-nth (prod f A) (Suc 0) = (∏ x∈A. fds-nth
(f x) (Suc 0))
  by (induction A rule: infinite-finite-induct) simp-all

lemma fds-nth-power-eq-0:
  assumes n < 2 ^ k fds-nth f 1 = 0
  shows   fds-nth (f ^ k) n = 0
  using assms(1)
proof (induction k arbitrary: n)
  case 0
  thus ?case by (simp add: one-fds-def)
next
  case (Suc k n)
  have fds-nth (f ^ Suc k) n = dirichlet-prod (fds-nth (f ^ k)) (fds-nth f) n
    by (subst power-Suc2) (simp add: fds-nth-mult dirichlet-prod-commutes)
  also have ... = 0 unfolding dirichlet-prod-def
  proof (intro sum.neutral ballI)
    fix d assume d: d ∈ {d. d dvd n}
    show fds-nth (f ^ k) d * fds-nth f (n div d) = 0
    proof (cases d < 2 ^ k)
      case True
      thus ?thesis using Suc.IH[of d] by simp
    next
      case False
      hence (n div d) * 2 ^ k ≤ (n div d) * d by (intro mult-left-mono) auto
      also from d have (n div d) * d = n by simp
      also from Suc have n < 2 * 2 ^ k by simp
      finally have n div d ≤ 1 by simp
      with assms(2) show ?thesis by (cases n div d) simp-all
    qed
  qed
  finally show ?case .
qed
```

4.2 Shifting the argument

```
class nat-power = semiring-1 +
  fixes nat-power :: nat ⇒ 'a ⇒ 'a
  assumes nat-power-0-left [simp]:  x ≠ 0 ⟹ nat-power 0 x = 0
  assumes nat-power-0-right [simp]: n > 0 ⟹ nat-power n 0 = 1
  assumes nat-power-1-left [simp]:  nat-power (Suc 0) x = 1
  assumes nat-power-1-right [simp]: nat-power n 1 = of-nat n
  assumes nat-power-add:            n > 0 ⟹ nat-power n (a + b) = nat-power
n a * nat-power n b
  assumes nat-power-mult-distrib:
    m > 0 ⟹ n > 0 ⟹ nat-power (m * n) a = nat-power m a * nat-power n a
  assumes nat-power-power:
    n > 0 ⟹ nat-power n (a * of-nat m) = nat-power n a ^ m
begin
```

lemma *nat-power-of-nat* [*simp*]: *m* > *0* ⟹ *nat-power m* (*of-nat n*) = *of-nat* (*m* ^ *n*)
by (*induction n*) (*simp-all add*: *nat-power-add*)

lemma *nat-power-power-left*: *m* > *0* ⟹ *nat-power* (*m* ^ *k*) *n* = *nat-power m n* ^ *k*
by (*induction k*) (*simp-all add*: *nat-power-mult-distrib*)

end

class *nat-power-field* = *nat-power* + *field* +
assumes *nat-power-nonzero* [*simp*]: *n* > *0* ⟹ *nat-power n z* ≠ *0*
begin

lemma *nat-power-diff*: *n* > *0* ⟹ *nat-power n* (*a* − *b*) = *nat-power n a* / *nat-power n b*
using *nat-power-add*[*of n a* − *b b*] **by** (*simp add*: *divide-simps*)

end

instantiation *nat* :: *nat-power*
begin
definition [*simp*]: *nat-power-nat a b* = (*a* ^ *b* :: *nat*)
instance by *standard* (*simp-all add*: *power-add power-mult-distrib power-mult*)
end

instantiation *real* :: *nat-power-field*
begin
definition [*simp*]: *nat-power-real a b* = (*real a powr b*)
instance proof
fix *n m* :: *nat* **and** *a* :: *real* **assume** *n* > *0*
thus *nat-power n* (*a* ∗ *real m*) = *nat-power n a* ^ *m*
by (*simp add*: *powr-def exp-of-nat-mult* [*symmetric*])
qed (*simp-all add*: *powr-add powr-mult*)
end

The following operation corresponds to shifting the argument of a Dirichlet series, i. e. subtracting a constant from it. In effect, this turns the series

$$A(s) = \sum_{n=1}^{\infty} \frac{a_n}{n^s}$$

into the series

$$A(s-c) = \sum_{n=1}^{\infty} \frac{n^c \cdot a_n}{n^s} \ .$$

definition *fds-shift* :: *'a* :: *nat-power* ⇒ *'a fds* ⇒ *'a fds* **where**
fds-shift c f = *fds* (*λn*. *fds-nth f n* ∗ *nat-power n c*)

lemma *fds-nth-shift* [*simp*]: *fds-nth* (*fds-shift c f*) *n* = *fds-nth f n* ∗ *nat-power n c*
  by (*simp add*: *fds-shift-def fds-nth-fds*)

lemma *fds-shift-shift* [*simp*]: *fds-shift c* (*fds-shift c′ f*) = *fds-shift* (*c′* + *c*) *f*
  by (*rule fds-eqI*) (*simp add*: *nat-power-add mult-ac*)

lemma *fds-shift-zero* [*simp*]: *fds-shift c 0* = *0*
  by (*rule fds-eqI*) *simp*

lemma *fds-shift-1* [*simp*]: *fds-shift a 1* = *1*
  by (*rule fds-eqI*) (*simp add*: *fds-shift-def one-fds-def*)

lemma *fds-shift-const* [*simp*]: *fds-shift a* (*fds-const c*) = *fds-const c*
  by (*rule fds-eqI*) (*simp add*: *fds-shift-def fds-const-def*)

lemma *fds-shift-add* [*simp*]:
  fixes *f g* :: *′a* :: {*monoid-add*, *nat-power*} *fds*
  shows *fds-shift c* (*f* + *g*) = *fds-shift c f* + *fds-shift c g*
  by (*rule fds-eqI*) (*simp add*: *algebra-simps*)

lemma *fds-shift-minus* [*simp*]:
  fixes *f g* :: *′a* :: {*comm-semiring-1-cancel*, *nat-power*} *fds*
  shows *fds-shift c* (*f* − *g*) = *fds-shift c f* − *fds-shift c g*
  by (*rule fds-eqI*) (*simp add*: *algebra-simps*)

lemma *fds-shift-uminus* [*simp*]:
  fixes *f* :: *′a* :: {*ring*, *nat-power*} *fds*
  shows *fds-shift c* (−*f*) = −*fds-shift c f*
  by (*rule fds-eqI*) (*simp add*: *algebra-simps*)

lemma *fds-shift-mult* [*simp*]:
  fixes *f g* :: *′a* :: {*comm-semiring*, *nat-power*} *fds*
  shows *fds-shift c* (*f* ∗ *g*) = *fds-shift c f* ∗ *fds-shift c g*
  by (*rule fds-eqI*)
    (*auto simp*: *algebra-simps fds-nth-mult dirichlet-prod-altdef2*
      sum-distrib-left sum-distrib-right nat-power-mult-distrib intro!: *sum.cong*)

lemma *fds-shift-power* [*simp*]:
  fixes *f* :: *′a* :: {*comm-semiring*, *nat-power*} *fds*
  shows *fds-shift c* (*f* ^ *n*) = *fds-shift c f* ^ *n*
  by (*induction n*) *simp-all*

lemma *fds-shift-by-0* [*simp*]: *fds-shift 0 f* = *f*
  by (*simp add*: *fds-shift-def*)

lemma *fds-shift-inverse* [*simp*]:
  fds-shift (*a* :: *′a* :: {*field*, *nat-power*}) (*inverse f*) = *inverse* (*fds-shift a f*)
proof (*cases fds-nth f 1* = *0*)

case *False*
have *fds-shift a f* ∗ *fds-shift a* (*inverse f*) = *fds-shift a* (*f* ∗ *inverse f*)
by *simp*
also from *False* **have** *f* ∗ *inverse f* = *1* **by** (*intro fds-right-inverse*)
finally have *fds-shift a f* ∗ *fds-shift a* (*inverse f*) = *1* **by** *simp*
thus *?thesis* **by** (*rule fds-right-inverse-unique*)
qed (*auto simp*: *inverse-fds-nonunit*)

lemma *fds-shift-divide* [*simp*]:
fds-shift (*a* :: *'a* :: {*field*, *nat-power*}) (*f* / *g*) = *fds-shift a f* / *fds-shift a g*
by (*simp add*: *divide-fds-def*)

lemma *fds-shift-sum* [*simp*]: *fds-shift a* (∑ *x*∈*A*. *f x*) = (∑ *x*∈*A*. *fds-shift a* (*f x*))
by (*induction A rule*: *infinite-finite-induct*) *simp-all*

lemma *fds-shift-prod* [*simp*]: *fds-shift a* (∏ *x*∈*A*. *f x*) = (∏ *x*∈*A*. *fds-shift a* (*f x*))
by (*induction A rule*: *infinite-finite-induct*) *simp-all*

4.3 Scaling the argument

The following operation corresponds to scaling the argument of a Dirichlet series with a natural number, i. e. turning the series

$$A(s) = \sum_{n=1}^{\infty} \frac{a_n}{n^s}$$

into the series

$$A(ks) = \sum_{n=1}^{\infty} \frac{a_n}{(n^k)^2} \ .$$

definition *fds-scale* :: *nat* ⇒ (*'a* :: *zero*) *fds* ⇒ *'a fds* **where**
fds-scale c f =
fds (λ*n*. *if n* > *0* ∧ *is-nth-power c n then fds-nth f* (*nth-root-nat c n*) *else 0*)

lemma *fds-scale-0* [*simp*]: *fds-scale 0 f* = *0*
by (*auto simp*: *fds-scale-def fds-eq-iff*)

lemma *fds-scale-1* [*simp*]: *fds-scale 1 f* = *f*
by (*auto simp*: *fds-scale-def fds-eq-iff*)

lemma *fds-nth-scale-power* [*simp*]:
c > *0* ⟹ *fds-nth* (*fds-scale c f*) (*n* ^ *c*) = *fds-nth f n*
by (*simp add*: *fds-scale-def fds-nth-fds*)

lemma *fds-nth-scale-nonpower* [*simp*]:
¬*is-nth-power c n* ⟹ *fds-nth* (*fds-scale c f*) *n* = *0*
by (*simp add*: *fds-scale-def fds-nth-fds*)

```
lemma fds-nth-scale:
  fds-nth (fds-scale c f) n =
     (if n > 0 ∧ is-nth-power c n then fds-nth f (nth-root-nat c n) else 0)
  by (cases c = 0) (auto simp: is-nth-power-def)

lemma fds-scale-const [simp]: c > 0 ⟹ fds-scale c (fds-const c′) = fds-const c′
  by (rule fds-eqI) (auto simp: fds-nth-scale fds-nth-const elim!: is-nth-powerE)

lemma fds-scale-zero [simp]: fds-scale c 0 = 0
  by (rule fds-eqI) (simp add: fds-nth-scale)

lemma fds-scale-one [simp]: c > 0 ⟹ fds-scale c 1 = 1
  by (simp only: fds-const-one [symmetric] fds-scale-const)

lemma fds-scale-of-nat [simp]: c > 0 ⟹ fds-scale c (of-nat n) = of-nat n
  by (simp only: fds-const-of-nat [symmetric] fds-scale-const)

lemma fds-scale-of-int [simp]: c > 0 ⟹ fds-scale c (of-int n) = of-int n
  by (simp only: fds-const-of-int [symmetric] fds-scale-const)

lemma fds-scale-numeral [simp]: c > 0 ⟹ fds-scale c (numeral n) = numeral n
  using fds-scale-of-nat[of c numeral n] by (simp del: fds-scale-of-nat)

lemma fds-scale-scale: fds-scale c (fds-scale c′ f) = fds-scale (c ∗ c′) f
proof (cases c = 0 ∨ c′ = 0)
  case False
  hence cc′: c > 0 c′ > 0 by auto
  show ?thesis
  proof (rule fds-eqI, goal-cases)
    case (1 n)
    show ?case
    proof (cases is-nth-power (c ∗ c′) n)
      case False
      with cc′ 1 have fds-nth (fds-scale c (fds-scale c′ f)) n = 0
      by (auto simp: fds-nth-scale is-nth-power-def power-mult [symmetric] mult.commute)
      with False cc′ show ?thesis by simp
    next
      case True
      from True obtain n′ where [simp]: n = n′ ^ (c′ ∗ c)
        by (auto elim: is-nth-powerE simp: mult.commute)
      with cc′ have fds-nth (fds-scale (c ∗ c′) f) n = fds-nth f n′
        by (simp add: mult.commute)
      also have ... = fds-nth (fds-scale c (fds-scale c′ f)) n
        using cc′ by (simp add: power-mult)
      finally show ?thesis ..
    qed
  qed
qed auto
```

```
lemma fds-scale-add [simp]:
  fixes f g :: 'a :: monoid-add fds
  shows fds-scale c (f + g) = fds-scale c f + fds-scale c g
  by (rule fds-eqI) (auto simp: fds-nth-scale)

lemma fds-scale-minus [simp]:
  fixes f g :: 'a :: {cancel-comm-monoid-add} fds
  shows fds-scale c (f − g) = fds-scale c f − fds-scale c g
  by (rule fds-eqI) (auto simp: fds-nth-scale)

lemma fds-scale-uminus [simp]:
  fixes f :: 'a :: group-add fds
  shows fds-scale c (−f) = −fds-scale c f
  by (rule fds-eqI) (auto simp: fds-nth-scale)

lemma fds-scale-mult [simp]:
  fixes f g :: 'a :: semiring-0 fds
  shows fds-scale c (f ∗ g) = fds-scale c f ∗ fds-scale c g
proof (cases c > 0)
  case True
  show ?thesis
  proof (rule fds-eqI, goal-cases)
    case (1 n)
    show ?case
    proof (cases is-nth-power c n)
      case False
      have fds-nth (fds-scale c f ∗ fds-scale c g) n =
             (∑(r, d) | r ∗ d = n. fds-nth (fds-scale c f) r ∗ fds-nth (fds-scale c g)
d)
        by (simp add: fds-nth-mult dirichlet-prod-altdef2)
      also from False have … = (∑(r, d) | r ∗ d = n. 0)
        by (intro sum.cong refl) (auto simp: fds-nth-scale dest: is-nth-power-mult)
      also from False have … = fds-nth (fds-scale c (f ∗ g)) n by simp
      finally show ?thesis ..
    next
      case True
      then obtain n' where [simp]: n = n' ^ c by (elim is-nth-powerE)
      define h where h = map-prod (nth-root-nat c) (nth-root-nat c)
      define i where i = map-prod (λn::nat. n ^ c) (λn::nat. n ^ c)
      define A where A = {(r, d). r ∗ d = n}
      define S where S = {rs∈A. ¬is-nth-power c (fst rs) ∨ ¬is-nth-power c (snd
rs)}

      have fds-nth (fds-scale c f ∗ fds-scale c g) n =
             (∑(r, d) | r ∗ d = n. fds-nth (fds-scale c f) r ∗ fds-nth (fds-scale c g)
d)
        by (simp add: fds-nth-mult dirichlet-prod-altdef2)
      also have … = (∑(r, d) | r ∗ d = n'. fds-nth f r ∗ fds-nth g d)
      proof (rule sym, intro sum.reindex-bij-witness-not-neutral[of {} S - h i])
```

show *finite S* **unfolding** *S-def A-def*
by (*rule finite-subset*[*OF - finite-divisors-nat'*[*of n*]]) (*insert* ‹*n* > *0*›, *auto*)
show *i* (*h rd*) = *rd* **if** *rd* ∈ {(*r*, *d*). *r* ∗ *d* = *n*} − *S* **for** *rd*
using ‹*c* > *0*› *that* **by** (*auto elim*!: *is-nth-powerE simp*: *S-def i-def h-def A-def*)
show *h rd* ∈ {(*r*,*d*). *r* ∗ *d* = *n'*} − {} **if** *rd* ∈ {(*r*, *d*). *r* ∗ *d* = *n*} − *S* **for** *rd*
using ‹*c* > *0*› *that* **by** (*auto elim*!: *is-nth-powerE*
simp: *S-def i-def h-def A-def power-mult-distrib* [*symmetric*] *power-eq-iff-eq-base*)
show *h* (*i rd*) = *rd* **if** *rd* ∈ {(*r*, *d*). *r* ∗ *d* = *n'*} − {} **for** *rd*
using *that* ‹*c* > *0*› **by** (*auto simp*: *h-def i-def*)
show *i rd* ∈ {(*r*, *d*). *r* ∗ *d* = *n*} − *S* **if** *rd* ∈ {(*r*,*d*). *r* ∗ *d* = *n'*} − {} **for** *rd*
using *that* ‹*c* > *0*› **by** (*auto simp*: *i-def S-def power-mult-distrib* [*symmetric*])
show (*case rd of* (*r*, *d*) ⇒ *fds-nth* (*fds-scale c f*) *r* ∗ *fds-nth* (*fds-scale c g*) *d*) = *0*
if *rd* ∈ *S* **for** *rd* **using** *that* **by** (*auto simp*: *S-def case-prod-unfold*)
qed (*insert* ‹*c* > *0*›, *auto simp*: *case-prod-unfold i-def*)
also have ... = *fds-nth* (*f* ∗ *g*) *n'* **by** (*simp add*: *fds-nth-mult dirichlet-prod-altdef2*)
also from ‹*c* > *0*› **have** ... = *fds-nth* (*fds-scale c* (*f* ∗ *g*)) *n* **by** *simp*
finally show *?thesis* **..**
qed
qed
qed *auto*

lemma *fds-scale-shift*:
fds-shift d (*fds-scale c f*) = *fds-scale c* (*fds-shift* (*c* ∗ *d*) *f*)
proof (*cases c* > *0*)
case *True*
thus *?thesis*
by (*intro fds-eqI*) (*auto simp*: *fds-nth-scale power-mult elim*!: *is-nth-powerE*)
qed *auto*

lemma *fds-ind-nth-power*: *k* > *0* ⟹ *fds-ind* (*is-nth-power k*) = *fds-scale k fds-zeta*
by (*rule fds-eqI*) (*auto simp*: *ind-def fds-nth-scale elim*!: *is-nth-powerE*)

4.4 Formal derivative

The formal derivative of a series

$$A(s) = \sum_{n=1}^{\infty} \frac{a_n}{n^s}$$

can easily be seen to be

$$A'(s) = -\sum_{n=1}^{\infty} \frac{\ln n \cdot a_n}{n^s} \ .$$

definition *fds-deriv* :: *'a* :: *real-algebra fds* ⇒ *'a fds* **where**

fds-deriv f = *fds* (λ*n*. − *ln* (*real n*) $*_R$ *fds-nth f n*)

lemma *fds-nth-deriv*: *fds-nth* (*fds-deriv f*) *n* = −*ln* (*real n*) $*_R$ *fds-nth f n*
by (*cases n* = *0*) (*simp-all add*: *fds-deriv-def*)

lemma *fds-deriv-const* [*simp*]: *fds-deriv* (*fds-const c*) = *0*
by (*rule fds-eqI*) (*simp add*: *fds-nth-deriv fds-nth-const*)

lemma *fds-deriv-0* [*simp*]: *fds-deriv 0* = *0*
by (*rule fds-eqI*) (*simp add*: *fds-nth-deriv*)

lemma *fds-deriv-1* [*simp*]: *fds-deriv 1* = *0*
by (*rule fds-eqI*) (*simp add*: *fds-nth-deriv fds-nth-one*)

lemma *fds-deriv-of-nat* [*simp*]: *fds-deriv* (*of-nat n*) = *0*
by (*simp only*: *fds-const-of-nat* [*symmetric*] *fds-deriv-const*)

lemma *fds-deriv-of-int* [*simp*]: *fds-deriv* (*of-int n*) = *0*
by (*simp only*: *fds-const-of-int* [*symmetric*] *fds-deriv-const*)

lemma *fds-deriv-of-real* [*simp*]: *fds-deriv* (*of-real n*) = *0*
by (*simp only*: *fds-const-of-real* [*symmetric*] *fds-deriv-const*)

lemma *fds-deriv-uminus* [*simp*]: *fds-deriv* (−*f*) = −*fds-deriv f*
by (*rule fds-eqI*) (*simp add*: *fds-nth-deriv*)

lemma *fds-deriv-add* [*simp*]: *fds-deriv* (*f* + *g*) = *fds-deriv f* + *fds-deriv g*
by (*rule fds-eqI*) (*simp add*: *fds-nth-deriv algebra-simps*)

lemma *fds-deriv-minus* [*simp*]: *fds-deriv* (*f* − *g*) = *fds-deriv f* − *fds-deriv g*
by (*rule fds-eqI*) (*simp add*: *fds-nth-deriv algebra-simps*)

lemma *fds-deriv-times* [*simp*]:
fds-deriv (*f* ∗ *g*) = *fds-deriv f* ∗ *g* + *f* ∗ *fds-deriv g*
by (*rule fds-eqI*)
(*auto simp add*: *fds-nth-deriv fds-nth-mult dirichlet-prod-altdef2 scaleR-right.sum*

algebra-simps sum.distrib [*symmetric*] *ln-mult intro*!: *sum.cong*)

lemma *fds-deriv-inverse* [*simp*]:
fixes *f* :: ′*a* :: {*real-algebra*, *field*} *fds*
assumes *fds-nth f* (*Suc 0*) ≠ *0*
shows *fds-deriv* (*inverse f*) = −*fds-deriv f* / *f* ^ *2*
proof −
have (*0* :: ′*a fds*) = *fds-deriv 1* **by** *simp*
also from *assms* **have** (*1* :: ′*a fds*) = *inverse f* ∗ *f* **by** (*simp add*: *fds-left-inverse*)
also have *fds-deriv* . . . = *fds-deriv* (*inverse f*) ∗ *f* + *inverse f* ∗ *fds-deriv f* **by** *simp*
also have . . . ∗ *inverse f* = *fds-deriv* (*inverse f*) ∗ (*f* ∗ *inverse f*) +

inverse f ^ 2 ∗ fds-deriv f
by (*simp add*: *algebra-simps power2-eq-square*)
also from *assms* **have** *f ∗ inverse f = 1* **by** (*simp add*: *fds-right-inverse*)
finally show *?thesis*
by (*simp add*: *algebra-simps power2-eq-square divide-fds-def inverse-mult-fds add-eq-0-iff*)
qed

lemma *fds-deriv-shift* [*simp*]: *fds-deriv* (*fds-shift c f*) = *fds-shift c* (*fds-deriv f*)
by (*rule fds-eqI*) (*simp add*: *fds-nth-deriv algebra-simps*)

lemma *fds-deriv-scale*: *fds-deriv* (*fds-scale c f*) = *of-nat c ∗ fds-scale c* (*fds-deriv f*)
proof (*cases c > 0*)
case *True*
have ∗: *of-nat a ∗* (*b* :: *'a*) = *real a* $*_R$ *b* **for** *a b*
by (*induction a*) (*simp-all add*: *algebra-simps*)
from *True* **show** *?thesis*
by (*intro fds-eqI*)
(*auto simp*: *fds-nth-deriv fds-nth-scale is-nth-powerE fds-const-of-nat* [*symmetric*]
ln-realpow ∗ *simp del*: *fds-const-of-nat elim*!: *is-nth-powerE*)
qed *auto*

lemma *fds-deriv-eq-imp-eq*:
assumes *fds-deriv f = fds-deriv g fds-nth f* (*Suc 0*) = *fds-nth g* (*Suc 0*)
shows *f = g*
proof (*rule fds-eqI*)
fix *n* :: *nat* **assume** *n*: *n > 0*
show *fds-nth f n = fds-nth g n*
proof (*cases n = 1*)
case *False*
with *n* **have** *n > 1* **by** *auto*
hence *fds-nth f n* = −*fds-nth* (*fds-deriv f*) *n* $/_R$ *ln n*
by (*simp add*: *fds-deriv-def*)
also note *assms*(*1*)
also from ‹*n > 1*› **have** −*fds-nth* (*fds-deriv g*) *n* $/_R$ *ln n = fds-nth g n*
by (*simp add*: *fds-deriv-def*)
finally show *?thesis* **.**
qed (*auto simp*: *assms*)
qed

lemma *completely-multiplicative-fds-deriv*:
assumes *completely-multiplicative-function f*
shows *fds-deriv* (*fds f*) = −*fds* (λ*n. f n ∗ mangoldt n*) ∗ *fds f*
proof (*rule fds-eqI*, *goal-cases*)
case (*1 n*)
interpret *completely-multiplicative-function f* **by** *fact*
have *fds-nth* (−*fds* (λ*n. f n ∗ mangoldt n*) ∗ *fds f*) *n* =
−(∑ (*r*, *d*) | *r ∗ d = n. f r ∗ mangoldt r ∗ f d*)

by (*simp add*: *fds-nth-mult fds-nth-deriv dirichlet-prod-altdef2*)
also have ($\sum$(*r*, *d*) | *r* ∗ *d* = *n*. *f r* ∗ *mangoldt r* ∗ *f d*) =
($\sum$(*r*, *d*) | *r* ∗ *d* = *n*. *mangoldt r* ∗ *f n*)
using *1* **by** (*intro sum.mono-neutral-cong-right refl*)
(*auto simp*: *mangoldt-def mult mult-ac intro*!: *finite-divisors-nat′ split*: *if-splits*)
also have ... = ($\sum$ *r* | *r dvd n*. *mangoldt r* ∗ *f n*) **using** *1*
by (*intro sum.reindex-bij-witness*[*of* - *λr*. (*r*, *n div r*) *fst*]) *auto*
also have ... = ($\sum$ *r* | *r dvd n*. *mangoldt r*) ∗ *f n* (**is** - = *?S* ∗ -)
by (*subst sum-distrib-right* [*symmetric*]) *simp*
also have ($\sum$ *r* | *r dvd n*. *mangoldt r*) = *of-real* (*ln* (*real n*))
using *1* **by** (*intro mangoldt-sum*) *simp*
also have − (*of-real* (*ln* (*real n*)) ∗ *f n*) = *fds-nth* (*fds-deriv* (*fds f*)) *n*
using *1* **by** (*simp add*: *fds-nth-deriv scaleR-conv-of-real*)
finally show *?case* **..**
qed

lemma *completely-multiplicative-fds-deriv′*:
completely-multiplicative-function (*fds-nth f*) ⟹
fds-deriv f = − *fds* (*λn*. *fds-nth f n* ∗ *mangoldt n*) ∗ *f*
using *completely-multiplicative-fds-deriv*[*of fds-nth f*] **by** *simp*

lemma *fds-deriv-zeta*:
fds-deriv fds-zeta =
−*fds mangoldt* ∗ (*fds-zeta* :: ′*a* :: {*comm-semiring-1*,*real-algebra-1*} *fds*)
proof −
have *completely-multiplicative-function* (*λn*. *if n* = *0 then 0 else 1*)
by *standard simp-all*
from *completely-multiplicative-fds-deriv* [*OF this*, *folded fds-zeta-altdef*]
show *?thesis* **by** *simp*
qed

lemma *fds-mangoldt-times-zeta*: *fds mangoldt* ∗ *fds-zeta* = *fds* (*λx*. *of-real* (*ln* (*real x*)))
by (*rule fds-eqI*) (*simp add*: *fds-nth-mult dirichlet-prod-def mangoldt-sum*)

lemma *fds-deriv-zeta′*: *fds-deriv fds-zeta* =
−*fds* (*λx*. *of-real* (*ln* (*real x*)):: ′*a* :: {*comm-semiring-1*,*real-algebra-1*})
by (*simp add*: *fds-deriv-zeta fds-mangoldt-times-zeta*)

4.5 Formal integral

definition *fds-integral* :: ′*a* ⇒ ′*a* :: *real-algebra fds* ⇒ ′*a fds* **where**
fds-integral c f = *fds* (*λn*. *if n* = *1 then c else* − *fds-nth f n* $/_R$ *ln* (*real n*))

lemma *fds-integral-0* [*simp*]: *fds-integral a 0* = *fds-const a*
by (*simp add*: *fds-integral-def fds-eq-iff*)

lemma *fds-integral-add*: *fds-integral* (*a* + *b*) (*f* + *g*) = *fds-integral a f* + *fds-integral*

b g
by (*rule fds-eqI*) (*auto simp*: *fds-integral-def scaleR-diff-right*)

lemma *fds-integral-diff*: *fds-integral* $(a - b)$ $(f - g)$ = *fds-integral a f* − *fds-integral b g*
by (*rule fds-eqI*) (*auto simp*: *fds-integral-def scaleR-diff-right*)

lemma *fds-integral-minus*: *fds-integral* $(-a)$ $(-f)$ = −*fds-integral a f*
by (*rule fds-eqI*) (*auto simp*: *fds-integral-def scaleR-diff-right*)

lemma *fds-shift-integral*: *fds-shift b* (*fds-integral a f*) = *fds-integral a* (*fds-shift b f*)
by (*rule fds-eqI*) (*simp add*: *fds-integral-def fds-shift-def*)

lemma *fds-deriv-fds-integral* [*simp*]:
fds-nth f (*Suc 0*) = *0* ⟹ *fds-deriv* (*fds-integral c f*) = *f*
by (*simp add*: *fds-deriv-def fds-integral-def fds-eq-iff*)

lemma *fds-integral-fds-deriv* [*simp*]: *fds-integral* (*fds-nth f 1*) (*fds-deriv f*) = *f*
by (*simp add*: *fds-deriv-def fds-integral-def fds-eq-iff*)

4.6 Formal logarithm

definition *fds-ln* :: *'a* ⇒ *'a* :: {*real-normed-field*} *fds* ⇒ *'a fds* **where**
fds-ln l f = *fds-integral l* (*fds-deriv f* / *f*)

lemma *fds-nth-Suc-0-fds-deriv* [*simp*]: *fds-nth* (*fds-deriv f*) (*Suc 0*) = *0*
by (*simp add*: *fds-deriv-def*)

lemma *fds-deriv-fds-ln* [*simp*]: *fds-deriv* (*fds-ln l f*) = *fds-deriv f* / *f*
unfolding *fds-ln-def* **by** (*subst fds-deriv-fds-integral*) (*simp-all add*: *divide-fds-def*)

lemma *fds-nth-Suc-0-fds-ln* [*simp*]: *fds-nth* (*fds-ln l f*) (*Suc 0*) = *l*
by (*simp add*: *fds-ln-def fds-integral-def*)

lemma *fds-ln-const* [*simp*]: *fds-ln l* (*fds-const c*) = *fds-const l*
by (*rule fds-eqI*) (*simp add*: *fds-ln-def fds-integral-def divide-fds-def*)

lemma *fds-ln-0* [*simp*]: *fds-ln l 0* = *fds-const l*
by (*rule fds-eqI*) (*simp add*: *fds-ln-def fds-integral-def divide-fds-def*)

lemma *fds-ln-1* [*simp*]: *fds-ln l 1* = *fds-const l*
by (*rule fds-eqI*) (*simp add*: *fds-ln-def fds-integral-def divide-fds-def*)

lemma *fds-shift-ln* [*simp*]: *fds-shift a* (*fds-ln l f*) = *fds-ln l* (*fds-shift a f*)
by (*simp add*: *fds-ln-def fds-shift-integral*)

lemma *fds-ln-mult*:
assumes *fds-nth f 1* ≠ *0* *fds-nth g 1* ≠ *0* $l' + l'' = l$

shows *fds-ln l (f ∗ g) = fds-ln l′ f + fds-ln l″ g*
proof −
have *fds-ln l (f ∗ g) = fds-integral (l′ + l″) ((fds-deriv f ∗ g + f ∗ fds-deriv g) / (f ∗ g))*
by (*simp add*: *fds-ln-def assms*)
also have (*fds-deriv f ∗ g + f ∗ fds-deriv g) / (f ∗ g) =*
fds-deriv f / f ∗ (g ∗ inverse g) + fds-deriv g / g ∗ (f ∗ inverse f)
by (*simp add*: *divide-fds-def algebra-simps inverse-mult-fds*)
also from *assms* **have** *f ∗ inverse f = 1* **by** (*intro fds-right-inverse*) *auto*
also from *assms* **have** *g ∗ inverse g = 1* **by** (*intro fds-right-inverse*) *auto*
finally show *?thesis* **by** (*simp add*: *fds-integral-add fds-ln-def*)
qed

lemma *fds-ln-power*:
assumes *fds-nth f 1 ≠ 0 l = of-nat n ∗ l′*
shows *fds-ln l (f ^ n) = of-nat n ∗ fds-ln l′ f*
proof −
have *fds-ln (of-nat n ∗ l′) (f ^ n) = of-nat n ∗ fds-ln l′ f*
using *assms*(*1*) **by** (*induction n*) (*simp-all add*: *fds-ln-mult algebra-simps*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *fds-ln-prod*:
assumes *⋀x. x ∈ A ⟹ fds-nth (f x) 1 ≠ 0 (∑x∈A. l′ x) = l*
shows *fds-ln l (∏x∈A. f x) = (∑x∈A. fds-ln (l′ x) (f x))*
proof −
have *fds-ln (∑x∈A. l′ x) (∏x∈A. f x) = (∑x∈A. fds-ln (l′ x) (f x))*
using *assms*(*1*) **by** (*induction A rule*: *infinite-finite-induct*) (*simp-all add*: *fds-ln-mult*)
with *assms* **show** *?thesis* **by** *simp*
qed

4.7 Formal exponential

definition *fds-exp* :: *′a* :: {*real-normed-algebra-1*,*banach*} *fds ⇒ ′a fds* **where**
fds-exp f = (let f′ = fds (λn. if n = 1 then 0 else fds-nth f n)
in fds (λn. exp (fds-nth f 1) ∗ (∑k. fds-nth (f′ ^ k) n $/_R$ *fact k)))*

lemma *fds-nth-exp-Suc-0* [*simp*]: *fds-nth (fds-exp f) (Suc 0) = exp (fds-nth f 1)*
proof −
have *fds-nth (fds-exp f) (Suc 0) = exp (fds-nth f 1) ∗ (∑k. 0 ^ k* $/_R$ *fact k)*
by (*simp add*: *fds-exp-def*)
also have *(∑k. (0::′a) ^ k* $/_R$ *fact k) = (∑k. if k = 0 then 1 else 0)*
by (*intro suminf-cong*) (*auto simp*: *power-0-left*)
also have ... *= 1* **using** *sums-If-finite*[*of λk. k = 0 λ-. 1 :: ′a*]
by (*simp add*: *sums-iff*)
finally show *?thesis* **by** *simp*
qed

lemma *fds-exp-times-fds-nth-0*:
fds-const (*exp* (*fds-nth f* (*Suc 0*))) ∗ *fds-exp* (*f* − *fds-const* (*fds-nth f* (*Suc 0*))) = *fds-exp f*
by (*rule fds-eqI*) (*simp add*: *fds-exp-def fds-nth-fds′ cong*: *if-cong*)

lemma *fds-exp-const* [*simp*]: *fds-exp* (*fds-const c*) = *fds-const* (*exp c*)
proof −
have *fds-exp* (*fds-const c*) = *fds* (λ*n*. *exp c* ∗ (∑ *k*. *fds-nth* (*fds* (λ*n*. *0*) ^ *k*) *n* /$_R$ *fact k*))
by (*simp add*: *fds-exp-def fds-nth-fds′ one-fds-def cong*: *if-cong*)
also have *fds* (λ-. *0* :: ′*a*) = *0* **by** (*simp add*: *fds-eq-iff*)
also have (λ(*k*::*nat*) (*n*::*nat*). *fds-nth* (*0* ^ *k*) *n*) = (λ*k n*. *if k* = *0* ∧ *n* = *1 then 1 else 0*)
by (*intro ext*) (*auto simp*: *one-fds-def fds-nth-fds′ power-0-left*)
also have (λ*n*::*nat*. ∑ *k*. (*if k* = *0* ∧ *n* = *1 then 1 else* (*0*::′*a*)) /$_R$ *fact k*) =
(λ*n*. *if n* = *1 then* (∑ *k*. (*if k* = *0 then 1 else 0*) /$_R$ *fact k*) *else 0*)
by (*intro ext*) *auto*
also have ... = (λ*n*::*nat*. *if n* = *1 then* (∑ *k*∈{*0*}. (*if k* = (*0*::*nat*) *then 1 else 0*)) *else 0* :: ′*a*)
by (*subst suminf-finite*[*of* {*0*}]) *auto*
also have *fds* (λ*n*. *exp c* ∗ ... *n*) = *fds-const* (*exp c*)
by (*simp add*: *fds-const-def fds-eq-iff fds-nth-fds′ cong*: *if-cong*)
finally show *?thesis* **.**
qed

lemma *fds-exp-numeral* [*simp*]: *fds-exp* (*numeral n*) = *fds-const* (*exp* (*numeral n*))
using *fds-exp-const*[*of numeral n* :: ′*a*] **by** (*simp del*: *fds-exp-const add*: *numeral-fds*)

lemma *fds-exp-0* [*simp*]: *fds-exp 0* = *1*
using *fds-exp-const*[*of 0*] **by** (*simp del*: *fds-exp-const*)

lemma *fds-exp-1* [*simp*]: *fds-exp 1* = *fds-const* (*exp 1*)
using *fds-exp-const*[*of 1*] **by** (*simp del*: *fds-exp-const*)

lemma *fds-nth-Suc-0-exp* [*simp*]: *fds-nth* (*fds-exp f*) (*Suc 0*) = *exp* (*fds-nth f* (*Suc 0*))
proof −
have (∑ *k*. *0* ^ *k* /$_R$ *fact k*) = (∑ *k*∈{*0*}. *0* ^ *k* /$_R$ *fact k* :: ′*a*)
by (*intro suminf-finite*) (*auto simp*: *power-0-left*)
also have ... = *1* **by** *simp*
finally show *?thesis* **by** (*simp add*: *fds-exp-def*)
qed

4.8 Subseries

definition *fds-subseries* :: (*nat* ⇒ *bool*) ⇒ (′*a* :: *semiring-1*) *fds* ⇒ ′*a fds* **where**
fds-subseries P f = *fds* (λ*n*. *if P n then fds-nth f n else 0*)

lemma *fds-nth-subseries*:
fds-nth (*fds-subseries P f*) *n* = (*if P n then fds-nth f n else 0*)
by (*simp add*: *fds-subseries-def fds-nth-fds'*)

lemma *fds-subseries-0* [*simp*]: *fds-subseries P 0* = *0*
by (*simp add*: *fds-subseries-def fds-eq-iff*)

lemma *fds-subseries-1* [*simp*]: *P 1* ⟹ *fds-subseries P 1* = *1*
by (*simp add*: *fds-subseries-def fds-eq-iff one-fds-def*)

lemma *fds-subseries-const* [*simp*]: *P 1* ⟹ *fds-subseries P* (*fds-const c*) = *fds-const c*
by (*simp add*: *fds-subseries-def fds-eq-iff fds-const-def*)

lemma *fds-subseries-add* [*simp*]: *fds-subseries P* (*f* + *g*) = *fds-subseries P f* + *fds-subseries P g*
by (*simp add*: *fds-subseries-def fds-eq-iff plus-fds-def*)

lemma *fds-subseries-diff* [*simp*]:
fds-subseries P (*f* − *g* :: *'a* :: *ring-1 fds*) = *fds-subseries P f* − *fds-subseries P g*
by (*simp add*: *fds-subseries-def fds-eq-iff minus-fds-def*)

lemma *fds-subseries-minus* [*simp*]:
fds-subseries P (−*f* :: *'a* :: *ring-1 fds*) = − *fds-subseries P f*
by (*simp add*: *fds-subseries-def fds-eq-iff minus-fds-def*)

lemma *fds-subseries-sum* [*simp*]: *fds-subseries P* ($\sum$ *x*∈*A*. *f x*) = ($\sum$ *x*∈*A*. *fds-subseries P* (*f x*))
by (*induction A rule*: *infinite-finite-induct*) *simp-all*

lemma *fds-subseries-shift* [*simp*]:
fds-subseries P (*fds-shift c f*) = *fds-shift c* (*fds-subseries P f*)
by (*simp add*: *fds-subseries-def fds-eq-iff*)

lemma *fds-subseries-deriv* [*simp*]:
fds-subseries P (*fds-deriv f*) = *fds-deriv* (*fds-subseries P f*)
by (*simp add*: *fds-subseries-def fds-deriv-def fds-eq-iff*)

lemma *fds-subseries-integral* [*simp*]:
P 1 ∨ *c* = *0* ⟹ *fds-subseries P* (*fds-integral c f*) = *fds-integral c* (*fds-subseries P f*)
by (*auto simp*: *fds-subseries-def fds-integral-def fds-eq-iff*)

abbreviation *fds-primepow-subseries* :: *nat* ⇒ (*'a* :: *semiring-1*) *fds* ⇒ *'a fds* **where**
fds-primepow-subseries p f ≡ *fds-subseries* (λ*n*. *prime-factors n* ⊆ {*p*}) *f*

lemma *fds-primepow-subseries-mult* [*simp*]:
fixes *p* :: *nat*

defines *P* ≡ (*λn. prime-factors n* ⊆ {*p*})
shows *fds-subseries P* (*f* ∗ *g*) = *fds-subseries P f* ∗ *fds-subseries P g*
proof (*rule fds-eqI*)
fix *n* :: *nat*
consider *n* = *0* | *P n n* > *0* | ¬*P n n* > *0* **by** *blast*
thus *fds-nth* (*fds-subseries P* (*f* ∗ *g*)) *n* = *fds-nth* (*fds-subseries P f* ∗ *fds-subseries P g*) *n*
proof *cases*
case *2*
have *P*: *P d* **if** *d dvd n* **for** *d*
proof −
have *prime-factors d* ⊆ *prime-factors n* **using** *that 2*
by (*intro dvd-prime-factors*) *auto*
also have ... ⊆ {*p*} **using** *2* **by** (*simp add*: *P-def*)
finally show *?thesis* **by** (*simp add*: *P-def*)
qed
have *P'*: *P a P b* **if** *n* = *a* ∗ *b* **for** *a b*
using *P*[*of a*] *P*[*of b*] *that* **by** *auto*

have *fds-nth* (*fds-subseries P* (*f* ∗ *g*)) *n* = *dirichlet-prod* (*fds-nth f*) (*fds-nth g*) *n*
using *2* **by** (*simp add*: *fds-subseries-def fds-nth-fds' fds-nth-mult*)
also have ... = *dirichlet-prod* (*fds-nth* (*fds-subseries P f*)) (*fds-nth* (*fds-subseries P g*)) *n*
unfolding *dirichlet-prod-altdef2* **using** *2*
by (*intro sum.cong refl*) (*auto simp*: *fds-subseries-def fds-nth-fds' dest*: *P'*)
finally show *?thesis* **by** (*simp add*: *fds-nth-mult*)
next
case *3*
have ¬(*P a* ∧ *P b*) **if** *n* = *a* ∗ *b* **for** *a b*
proof −
have *prime-factors n* = *prime-factors* (*a* ∗ *b*) **by** (*simp add*: *that*)
also have ... = *prime-factors a* ∪ *prime-factors b*
using *3 that* **by** (*intro prime-factors-product*) *auto*
finally show *?thesis* **using** *3* **by** (*auto simp*: *P-def*)
qed
hence *dirichlet-prod* (*fds-nth* (*fds-subseries P f*)) (*fds-nth* (*fds-subseries P g*)) *n* = *0*
unfolding *dirichlet-prod-altdef2*
by (*intro sum.neutral*) (*auto simp*: *fds-subseries-def fds-nth-fds'*)
also have ... = *fds-nth* (*fds-subseries P* (*f* ∗ *g*)) *n*
using *3* **by** (*simp add*: *fds-subseries-def*)
finally show *?thesis* **by** (*simp add*: *fds-nth-mult*)
qed *auto*
qed

lemma *fds-primepow-subseries-power* [*simp*]:
fds-primepow-subseries p (*f* ^ *n*) = *fds-primepow-subseries p f* ^ *n*
by (*induction n*) *simp-all*

lemma *fds-primepow-subseries-prod* [*simp*]:
fds-primepow-subseries p ($\prod x \in A$. *f x*) = ($\prod x \in A$. *fds-primepow-subseries p* (*f x*))
by (*induction A rule*: *infinite-finite-induct*) *simp-all*

lemma *completely-multiplicative-function-only-pows*:
assumes *completely-multiplicative-function* (*fds-nth f*)
shows *completely-multiplicative-function* (*fds-nth* (*fds-primepow-subseries p f*))
proof −
interpret *completely-multiplicative-function fds-nth f* **by** *fact*
show *?thesis*
by *standard* (*auto simp*: *fds-nth-subseries prime-factors-product mult*)
qed

4.9 Truncation

definition *fds-truncate* :: *nat* ⇒ *'a* ::{*zero*} *fds* ⇒ *'a fds* **where**
fds-truncate m f = *fds* (λn. *if n* ≤ *m then fds-nth f n else 0*)

lemma *fds-nth-truncate*: *fds-nth* (*fds-truncate m f*) *n* = (*if n* ≤ *m then fds-nth f n else 0*)
by (*simp add*: *fds-truncate-def fds-nth-fds'*)

lemma *fds-truncate-0* [*simp*]: *fds-truncate 0 f* = *0*
by (*simp add*: *fds-eq-iff fds-nth-truncate*)

lemma *fds-truncate-zero* [*simp*]: *fds-truncate m 0* = *0*
by (*simp add*: *fds-truncate-def fds-eq-iff*)

lemma *fds-truncate-one* [*simp*]: *m* > *0* ⟹ *fds-truncate m 1* = *1*
by (*simp add*: *fds-truncate-def fds-eq-iff*)

lemma *fds-truncate-const* [*simp*]: *m* > *0* ⟹ *fds-truncate m* (*fds-const c*) = *fds-const c*
by (*simp add*: *fds-truncate-def fds-eq-iff*)

lemma *fds-truncate-truncate* [*simp*]: *fds-truncate m* (*fds-truncate n f*) = *fds-truncate* (*min m n*) *f*
by (*rule fds-eqI*) (*simp add*: *fds-nth-truncate*)

lemma *fds-truncate-truncate'* [*simp*]: *fds-truncate m* (*fds-truncate m f*) = *fds-truncate m f*
by (*rule fds-eqI*) (*simp add*: *fds-nth-truncate*)

lemma *fds-truncate-shift* [*simp*]: *fds-truncate m* (*fds-shift a f*) = *fds-shift a* (*fds-truncate m f*)
by (*simp add*: *fds-eq-iff fds-nth-truncate*)

lemma *fds-truncate-add-strong*:
fds-truncate m (*f* + *g* :: *'a* :: *monoid-add fds*) = *fds-truncate m f* + *fds-truncate m g*
by (*auto simp*: *fds-eq-iff fds-nth-truncate*)

lemma *fds-truncate-add*:
fds-truncate m (*fds-truncate m f* + *fds-truncate m g* :: *'a* :: *monoid-add fds*) =
fds-truncate m (*f* + *g*)
by (*auto simp*: *fds-eq-iff fds-nth-truncate*)

lemma *fds-truncate-mult*:
fds-truncate m (*fds-truncate m f* ∗ *fds-truncate m g*) = *fds-truncate m* (*f* ∗ *g*) (**is** *?A* = *?B*)
proof (*intro fds-eqI*, *goal-cases*)
case (*1 n*)
show *?case*
proof (*cases n* ≤ *m*)
case *True*
hence *fds-nth ?B n* = *dirichlet-prod* (*fds-nth f*) (*fds-nth g*) *n*
by (*simp add*: *fds-nth-truncate fds-nth-mult*)
also have ... = *dirichlet-prod* (*fds-nth* (*fds-truncate m f*)) (*fds-nth* (*fds-truncate m g*)) *n*
unfolding *dirichlet-prod-def*
proof (*intro sum.cong refl*, *goal-cases*)
case (*1 d*)
with ‹*n* > *0*› **have** *d* ≤ *m n div d* ≤ *m*
by (*auto dest*: *dvd-imp-le intro*: *order.trans*[*OF* - *True*])
thus *?case* **by** (*auto simp add*: *fds-nth-truncate*)
qed
also have ... = *fds-nth ?A n* **using** *True* **by** (*simp add*: *fds-nth-truncate fds-nth-mult*)
finally show *?thesis* **..**
qed (*auto simp*: *fds-nth-truncate*)
qed

lemma *fds-truncate-deriv*: *fds-truncate m* (*fds-deriv f*) = *fds-deriv* (*fds-truncate m f*)
by (*simp add*: *fds-eq-iff fds-nth-truncate fds-deriv-def*)

lemma *fds-truncate-integral*:
m > *0* ∨ *c* = *0* ⟹ *fds-truncate m* (*fds-integral c f*) = *fds-integral c* (*fds-truncate m f*)
by (*auto simp*: *fds-eq-iff fds-nth-truncate fds-integral-def*)

lemma *fds-truncate-power*: *fds-truncate m* (*fds-truncate m f* ^ *n*) = *fds-truncate m* (*f* ^ *n*)
proof (*cases m* = *0*)
case *False*
show *?thesis*

proof (*induction n*)
  case (*Suc n*)
  have *fds-truncate m* (*fds-truncate m f* ^ *Suc n*) =
      fds-truncate m (*fds-truncate m f* ∗ *fds-truncate m f* ^ *n*) **by** *simp*
  also have ... = *fds-truncate m* (*fds-truncate m f* ∗ *fds-truncate m* (*f* ^ *n*))
    by (*subst fds-truncate-mult* [*symmetric*]) (*simp add*: *Suc*)
  also have ... = *fds-truncate m* (*f* ^ *Suc n*)
    by (*simp add*: *fds-truncate-mult*)
  finally show *?case* **.**
qed (*simp-all add*: *fds-truncate-mult*)
qed *simp-all*

lemma *dirichlet-inverse-cong-simp*:
  assumes $\bigwedge m.\ m > 0 \Longrightarrow m \le n \Longrightarrow f\ m = f'\ m$ $i = i'$ $n = n'$
  shows *dirichlet-inverse f i n* = *dirichlet-inverse f′ i′ n′*
proof −
  have *dirichlet-inverse f i n* = *dirichlet-inverse f′ i n*
  using *assms*(*1*)
  proof (*induction n rule*: *dirichlet-inverse-induct*)
    case (*gt1 n*)
    have ∗: *dirichlet-inverse f i k* = *dirichlet-inverse f′ i k* **if** k *dvd* $n \wedge k < n$ **for** *k*
      using *that* **by** (*intro gt1*) *auto*
    have ∗: ($\sum d$ | *d dvd n* ∧ *d* < *n*. *f* (*n div d*) ∗ *dirichlet-inverse f i d*) =
        ($\sum d$ | *d dvd n* ∧ *d* < *n*. *f′* (*n div d*) ∗ *dirichlet-inverse f′ i d*)
      by (*intro sum.cong refl*) (*subst gt1.prems*, *auto elim*: *dvdE simp*: ∗)
    consider *n* = *0* | *n* = *1* | *n* > *1* **by** *force*
    thus *?case*
      by *cases* (*insert* ∗, *simp-all add*: *dirichlet-inverse-gt-1* ∗ *cong*: *sum.cong*)
  qed *auto*
  with *assms*(*2*,*3*) **show** *?thesis* **by** *simp*
qed

lemma *fds-truncate-cong*:
  ($\bigwedge n.\ m > 0 \Longrightarrow n > 0 \Longrightarrow n \le m \Longrightarrow$ *fds-nth f n* = *fds-nth f′ n*) ⟹
   fds-truncate m f = *fds-truncate m f′*
  by (*rule fds-eqI*) (*simp add*: *fds-nth-truncate*)

lemma *fds-truncate-inverse*:
  fds-truncate m (*inverse* (*fds-truncate m* (*f* :: ′*a* :: *field fds*))) = *fds-truncate m*
(*inverse f*)
proof (*rule fds-truncate-cong*, *goal-cases*)
  case (*1 n*)
  have ∗: *dirichlet-inverse* (λ*n*. *if n* ≤ *m then fds-nth f n else 0*) (*inverse* (*fds-nth*
f 1)) *n* =
      dirichlet-inverse (*fds-nth f*) (*inverse* (*fds-nth f 1*)) *n* **using** *1*
    by (*intro dirichlet-inverse-cong-simp*) *auto*
  show *?case*
  proof (*cases fds-nth f 1* = *0*)
    case *True*

thus *?thesis* **by** (*auto simp*: *inverse-fds-nonunit fds-nth-truncate*)
qed (*insert* ∗ *1*, *auto simp*: *inverse-fds-def fds-nth-fds′ fds-nth-truncate Suc-le-eq*)
qed

lemma *fds-truncate-divide*:
fixes *f g* :: *′a* :: *field fds*
shows *fds-truncate m* (*fds-truncate m f* / *fds-truncate m g*) = *fds-truncate m* (*f* / *g*)
proof −
have *fds-truncate m* (*f* / *g*) = *fds-truncate m* (*fds-truncate m* (*fds-truncate m f*) ∗
fds-truncate m (*inverse* (*fds-truncate m g*)))
by (*simp add*: *fds-truncate-inverse fds-truncate-mult divide-fds-def*)
also have . . . = *fds-truncate m* (*fds-truncate m f* ∗ *inverse* (*fds-truncate m g*))
by (*rule fds-truncate-mult*)
also have . . . = *fds-truncate m* (*fds-truncate m f* / *fds-truncate m g*)
by (*simp add*: *divide-fds-def*)
finally show *?thesis* **..**
qed

lemma *fds-truncate-ln*:
fixes *f* :: *′a* :: *real-normed-field fds*
shows *fds-truncate m* (*fds-ln l* (*fds-truncate m f*)) = *fds-truncate m* (*fds-ln l f*)
by (*cases m* = *0*)
(*simp-all add*: *fds-ln-def fds-truncate-integral fds-truncate-deriv* [*symmetric*]
fds-truncate-divide)

lemma *fds-truncate-exp*:
shows *fds-truncate m* (*fds-exp* (*fds-truncate m f*)) = *fds-truncate m* (*fds-exp f*)
proof (*rule fds-truncate-cong*, *goal-cases*)
case (*1 n*)
define *a* **where** *a* = *exp* (*fds-nth f* (*Suc 0*))
define *f′* **where** *f′* = *fds* (λ*n*. *if n* = *Suc 0 then 0 else fds-nth f n*)
have *truncate-f′*: *fds-truncate m f′* = *fds* (λ*n*. *if n* = *Suc 0 then 0 else fds-nth* (*fds-truncate m f*) *n*)
by (*simp add*: *f′-def fds-eq-iff fds-nth-truncate*)

have *fds-nth* (*fds-exp* (*fds-truncate m f*)) *n* =
a ∗ (∑*k*. *fds-nth* (*fds-truncate m f′* ^ *k*) *n* $/_R$ *fact k*) **using** *1*
by (*simp add*: *fds-exp-def fds-nth-fds′ a-def* [*symmetric*] *f′-def* [*symmetric*]
fds-nth-truncate truncate-f′ [*symmetric*])
also have (λ*k*. *fds-nth* (*fds-truncate m f′* ^ *k*) *n*) = (λ*k*. *fds-nth* (*f′* ^ *k*) *n*)
proof (*rule ext*, *goal-cases*)
case (*1 k*)
have *fds-nth* (*fds-truncate m f′* ^ *k*) *n* = *fds-nth* (*fds-truncate m* (*fds-truncate m f′* ^ *k*)) *n*
using ‹*n* ≤ *m*› **by** (*simp add*: *fds-nth-truncate*)
also have *fds-truncate m* (*fds-truncate m f′* ^ *k*) = *fds-truncate m* (*f′* ^ *k*)
by (*simp add*: *fds-truncate-power*)

also have *fds-nth* ... *n* = *fds-nth* ($f' \,\widehat{}\, k$) *n* **using** ‹*n* ≤ *m*› **by** (*simp add*: *fds-nth-truncate*)
finally show *?case* **.**
qed
also have $a * (\sum k. \ldots\ k\ /_R\ fact\ k)$ = *fds-nth* (*fds-exp f*) *n*
by (*simp add*: *fds-exp-def fds-nth-fds′ a-def f′-def*)
finally show *?case* **.**
qed

lemma *fds-eqI-truncate*:
assumes $\bigwedge m.\ m > 0 \Longrightarrow$ *fds-truncate m f* = *fds-truncate m g*
shows *f* = *g*
proof (*rule fds-eqI*)
fix *n* :: *nat* **assume** *n* > *0*
have *fds-nth f n* = *fds-nth* (*fds-truncate n f*) *n*
by (*simp add*: *fds-nth-truncate*)
also note *assms*[*OF* ‹*n* > *0*›]
also have *fds-nth* (*fds-truncate n g*) *n* = *fds-nth g n*
by (*simp add*: *fds-nth-truncate*)
finally show *fds-nth f n* = *fds-nth g n* **.**
qed

4.10 Normed series

definition *fds-norm* :: *′a* :: {*real-normed-div-algebra*} *fds* ⇒ *real fds*
where *fds-norm f* = *fds* (λ*n*. *of-real* (*norm* (*fds-nth f n*)))

lemma *fds-nth-norm* [*simp*]: *fds-nth* (*fds-norm f*) *n* = *norm* (*fds-nth f n*)
by (*simp add*: *fds-norm-def fds-nth-fds′*)

lemma *fds-norm-1* [*simp*]: *fds-norm 1* = *1*
by (*simp add*: *fds-eq-iff one-fds-def*)

lemma *fds-nth-norm-mult-le*:
shows *norm* (*fds-nth* (*f* ∗ *g*) *n*) ≤ *fds-nth* (*fds-norm f* ∗ *fds-norm g*) *n*
by (*auto simp add*: *fds-nth-mult dirichlet-prod-def norm-mult intro*!: *sum-norm-le*)

lemma *fds-nth-norm-mult-nonneg* [*simp*]: *fds-nth* (*fds-norm f* ∗ *fds-norm g*) *n* ≥ *0*
by (*auto simp*: *fds-nth-mult dirichlet-prod-def intro*!: *sum-nonneg*)

4.11 Lifting a real series to a real algebra

definition *fds-of-real* :: *real fds* ⇒ *′a* :: {*real-normed-algebra-1*} *fds* **where**
fds-of-real f = *fds* (λ*n*. *of-real* (*fds-nth f n*))

lemma *fds-nth-of-real* [*simp*]: *fds-nth* (*fds-of-real f*) *n* = *of-real* (*fds-nth f n*)
by (*simp add*: *fds-of-real-def fds-nth-fds′*)

lemma *fds-of-real-0* [*simp*]: *fds-of-real 0* = *0*

and *fds-of-real-1* [*simp*]: *fds-of-real 1* = *1*
and *fds-of-real-const* [*simp*]: *fds-of-real* (*fds-const c*) = *fds-const* (*of-real c*)
and *fds-of-real-minus* [*simp*]: *fds-of-real* (−*f*) = −*fds-of-real f*
and *fds-of-real-add* [*simp*]: *fds-of-real* (*f* + *g*) = *fds-of-real f* + *fds-of-real g*
and *fds-of-real-mult* [*simp*]: *fds-of-real* (*f* ∗ *g*) = *fds-of-real f* ∗ *fds-of-real g*
and *fds-of-real-deriv* [*simp*]: *fds-of-real* (*fds-deriv f*) = *fds-deriv* (*fds-of-real f*)
by (*simp-all add*: *fds-eq-iff one-fds-def fds-const-def fds-nth-mult dirichlet-prod-def fds-deriv-def scaleR-conv-of-real*)

lemma *fds-of-real-higher-deriv* [*simp*]:
(*fds-deriv* ^^ *n*) (*fds-of-real f*) = *fds-of-real* ((*fds-deriv* ^^ *n*) *f*)
by (*induction n*) *simp-all*

4.12 Convergence and connection to concrete functions

The following definitions establish a connection of a formal Dirichlet series to the concrete analytic function that it corresponds to. This correspondence is usually partial in the sense that a series may not converge everywhere.

definition *eval-fds* :: (*'a* :: {*nat-power*, *real-normed-field*, *banach*}) *fds* ⇒ *'a* ⇒ *'a*
where
eval-fds f s = (∑*n*. *fds-nth f n* / *nat-power n s*)

lemma *eval-fds-eqI*:
assumes (λ*n*. *fds-nth f* (*Suc n*) / *nat-power* (*Suc n*) *s*) *sums L*
shows *eval-fds f s* = *L*
proof −
from *assms* **have** (λ*n*. *fds-nth f n* / *nat-power n s*) *sums L*
by (*subst* (*asm*) *sums-Suc-iff*) *auto*
thus *?thesis* **by** (*simp add*: *eval-fds-def sums-iff*)
qed

definition *fds-converges* ::
(*'a* :: {*nat-power*, *real-normed-field*, *banach*}) *fds* ⇒ *'a* ⇒ *bool* **where**
fds-converges f s ⟷ *summable* (λ*n*. *fds-nth f n* / *nat-power n s*)

lemma *fds-converges-iff*:
fds-converges f s ⟷ (λ*n*. *fds-nth f n* / *nat-power n s*) *sums eval-fds f s*
by (*simp add*: *fds-converges-def sums-iff eval-fds-def*)

definition *fds-abs-converges* ::
(*'a* :: {*nat-power*, *real-normed-field*, *banach*}) *fds* ⇒ *'a* ⇒ *bool* **where**
fds-abs-converges f s ⟷ *summable* (λ*n*. *norm* (*fds-nth f n* / *nat-power n s*))

lemma *fds-abs-converges-imp-converges* [*dest*, *intro*]:
fds-abs-converges f s ⟹ *fds-converges f s*
unfolding *fds-abs-converges-def fds-converges-def* **by** (*rule summable-norm-cancel*)

lemma *fds-converges-altdef*:

fds-converges f s ⟷ (*λn. fds-nth f* (*Suc n*) / *nat-power* (*Suc n*) *s*) *sums eval-fds f s*
unfolding *fds-converges-def summable-sums-iff*
by (*subst sums-Suc-iff*) (*simp-all add*: *eval-fds-def*)

lemma *fds-const-abs-converges* [*simp*]: *fds-abs-converges* (*fds-const c*) *s*
proof −
have *summable* (*λn. norm* (*fds-nth* (*fds-const c*) *n* / *nat-power n s*)) ⟷
summable (*λn. if n* = *1 then norm c else* (*0* :: *real*))
by (*intro summable-cong*) *simp*
also have . . . **by** *simp*
finally show *?thesis* **by** (*simp add*: *fds-abs-converges-def*)
qed

lemma *fds-const-converges* [*simp*]: *fds-converges* (*fds-const c*) *s*
by (*rule fds-abs-converges-imp-converges*) *simp*

lemma *eval-fds-const* [*simp*]: *eval-fds* (*fds-const c*) = (*λ-. c*)
proof
fix *s*
have *eval-fds* (*fds-const c*) *s* = ($\sum$ *n. if n* = *1 then c else 0*) **unfolding** *eval-fds-def*
by (*intro suminf-cong*) *simp*
also have . . . = *c* **using** *sums-single*[*of 1 λ-. c*] **by** (*simp add*: *sums-iff*)
finally show *eval-fds* (*fds-const c*) *s* = *c* **.**
qed

lemma *fds-zero-abs-converges* [*simp*]: *fds-abs-converges 0 s*
by (*simp add*: *fds-abs-converges-def*)

lemma *fds-zero-converges* [*simp*]: *fds-converges 0 s*
by (*simp add*: *fds-converges-def*)

lemma *eval-fds-zero* [*simp*]: *eval-fds 0* = (*λ-. 0*)
by (*simp only*: *fds-const-zero* [*symmetric*] *eval-fds-const*)

lemma *fds-one-abs-converges* [*simp*]: *fds-abs-converges 1 s*
by (*simp only*: *fds-const-one* [*symmetric*] *fds-const-abs-converges*)

lemma *fds-one-converges* [*simp*]: *fds-converges 1 s*
by (*simp only*: *fds-const-one* [*symmetric*] *fds-const-converges*)

lemma *fds-converges-truncate* [*simp*]: *fds-converges* (*fds-truncate n f*) *s*
proof −
have *summable* (*λk. fds-nth* (*fds-truncate n f*) *k* / *nat-power k s*) ⟷ *summable* (*λ-. 0* :: *'a*)
by (*intro summable-cong*[*OF eventually-mono*[*OF eventually-gt-at-top*[*of n*]]])
(*auto simp*: *fds-nth-truncate*)
thus *?thesis* **by** (*simp add*: *fds-converges-def*)
qed

lemma *fds-abs-converges-truncate* [*simp*]: *fds-abs-converges* (*fds-truncate n f*) *s*
proof −
  have *summable* (λ*k*. *norm* (*fds-nth* (*fds-truncate n f*) *k* / *nat-power k s*)) ⟷ *summable* (λ-. *0* :: *real*)
    by (*intro summable-cong*[*OF eventually-mono*[*OF eventually-gt-at-top*[*of n*]]])
      (*auto simp*: *fds-nth-truncate*)
  thus *?thesis* **by** (*simp add*: *fds-abs-converges-def*)
qed

lemma *fds-abs-converges-subseries* [*simp*, *intro*]:
  assumes *fds-abs-converges f s*
  shows *fds-abs-converges* (*fds-subseries P f*) *s*
  unfolding *fds-abs-converges-def*
proof (*rule summable-comparison-test-ev*)
  show *summable* (λ*n*. *norm* (*fds-nth f n* / *nat-power n s*))
    using *assms* **unfolding** *fds-abs-converges-def* **.**
qed (*auto simp*: *fds-nth-subseries*)

lemma *eval-fds-one* [*simp*]: *eval-fds 1* = (λ-. *1*)
  by (*simp only*: *fds-const-one* [*symmetric*] *eval-fds-const*)

lemma *eval-fds-truncate*: *eval-fds* (*fds-truncate n f*) *s* = ($\sum$ *k*=*1*..*n*. *fds-nth f k* / *nat-power k s*)
proof −
  have *eval-fds* (*fds-truncate n f*) *s* = ($\sum$ *k*=*1*..*n*. *fds-nth* (*fds-truncate n f*) *k* / *nat-power k s*)
    unfolding *eval-fds-def* **by** (*intro suminf-finite*) (*auto simp*: *fds-nth-truncate Suc-le-eq*)
  also have ... = ($\sum$ *k*=*1*..*n*. *fds-nth f k* / *nat-power k s*)
    by (*intro sum.cong*) (*auto simp*: *fds-nth-truncate*)
  finally show *?thesis* **.**
qed

lemma *fds-converges-add*:
  assumes *fds-converges f s fds-converges g s*
  shows *fds-converges* (*f* + *g*) *s*
  using *summable-add*[*OF assms*[*unfolded fds-converges-def*]]
  by (*simp add*: *fds-converges-def add-divide-distrib*)

lemma *fds-abs-converges-add*:
  assumes *fds-abs-converges f s fds-abs-converges g s*
  shows *fds-abs-converges* (*f* + *g*) *s*
  unfolding *fds-abs-converges-def*
proof (*rule summable-comparison-test*, *intro exI allI impI*)
  let *?A* = (λ*n*. *norm* (*fds-nth f n* / *nat-power n s*) + *norm* (*fds-nth g n* / *nat-power n s*))
  from *summable-add*[*OF assms*[*unfolded fds-abs-converges-def*]] **show** *summable*

?A .
  fix *n* :: *nat*
  show *norm* (*norm* (*fds-nth* (*f* + *g*) *n* / *nat-power n s*)) ≤ *?A n*
    by (*simp add*: *norm-triangle-ineq add-divide-distrib*)
qed

lemma *eval-fds-add*:
  assumes *fds-converges f s fds-converges g s*
  shows *eval-fds* (*f* + *g*) *s* = *eval-fds f s* + *eval-fds g s*
proof −
  from *assms* **have** (λ*n*. *fds-nth f n* / *nat-power n s*) *sums eval-fds f s*
    (λ*n*. *fds-nth g n* / *nat-power n s*) *sums eval-fds g s*
    by (*simp-all add*: *fds-converges-def sums-iff eval-fds-def*)
  from *sums-add*[*OF this*] **show** *?thesis* **by** (*simp add*: *eval-fds-def sums-iff add-divide-distrib*)
qed

lemma *fds-converges-uminus*:
  assumes *fds-converges f s*
  shows *fds-converges* (−*f*) *s*
  using *summable-minus*[*OF assms*[*unfolded fds-converges-def*]]
  by (*simp add*: *fds-converges-def add-divide-distrib*)

lemma *The-cong*: *The P* = *The Q* **if** ⋀*x*. *P x* ⟷ *Q x*
proof −
  from *that* **have** *P* = *Q* **by** *auto*
  thus *?thesis* **by** *simp*
qed

lemma *fds-abs-converges-uminus*:
  assumes *fds-abs-converges f s*
  shows *fds-abs-converges* (−*f*) *s*
  using *assms* **by** (*simp add*: *fds-abs-converges-def*)

lemma *eval-fds-uminus*: *fds-converges f s* ⟹ *eval-fds* (−*f*) *s* = −*eval-fds f s*
  by (*simp add*: *fds-converges-def eval-fds-def suminf-minus*)

lemma *fds-converges-diff*:
  assumes *fds-converges f s fds-converges g s*
  shows *fds-converges* (*f* − *g*) *s*
  using *summable-diff*[*OF assms*[*unfolded fds-converges-def*]]
  by (*simp add*: *fds-converges-def diff-divide-distrib*)

lemma *fds-abs-converges-diff*:
  assumes *fds-abs-converges f s fds-abs-converges g s*
  shows *fds-abs-converges* (*f* − *g*) *s*
  unfolding *fds-abs-converges-def*
proof (*rule summable-comparison-test*, *intro exI allI impI*)

let *?A* = (*λn. norm* (*fds-nth f n* / *nat-power n s*) + *norm* (*fds-nth g n* / *nat-power n s*))
from *summable-add*[*OF assms*[*unfolded fds-abs-converges-def*]] **show** *summable ?A* **.**
fix *n* :: *nat*
show *norm* (*norm* (*fds-nth* (*f* − *g*) *n* / *nat-power n s*)) ≤ *?A n*
by (*simp add*: *norm-triangle-ineq4 diff-divide-distrib*)
qed

lemma *eval-fds-diff*:
assumes *fds-converges f s fds-converges g s*
shows *eval-fds* (*f* − *g*) *s* = *eval-fds f s* − *eval-fds g s*
proof −
from *assms* **have** (*λn. fds-nth f n* / *nat-power n s*) *sums eval-fds f s*
(*λn. fds-nth g n* / *nat-power n s*) *sums eval-fds g s*
by (*simp-all add*: *fds-converges-def sums-iff eval-fds-def*)
from *sums-diff*[*OF this*] **show** *?thesis* **by** (*simp add*: *eval-fds-def sums-iff diff-divide-distrib*)
qed

lemma *eval-fds-at-nat*: *eval-fds f* (*of-nat k*) = (∑*n. fds-nth f n* / *of-nat n* ^ *k*)
unfolding *eval-fds-def*
proof (*intro suminf-cong*, *goal-cases*)
case (*1 n*)
thus *?case* **by** (*cases n* = *0*) *simp-all*
qed

lemma *eval-fds-at-numeral*: *eval-fds f* (*numeral k*) = (∑*n. fds-nth f n* / *of-nat n* ^ *numeral k*)
using *eval-fds-at-nat*[*of f numeral k*] **by** *simp*

lemma *eval-fds-at-1*: *eval-fds f 1* = (∑*n. fds-nth f n* / *of-nat n*)
using *eval-fds-at-nat*[*of f 1*] **by** *simp*

lemma *eval-fds-at-0*: *eval-fds f 0* = (∑*n. fds-nth f n*)
using *eval-fds-at-nat*[*of f 0*] **by** *simp*

lemma *suminf-fds-zeta-aux*:
f 0 = *0* ⟹ (∑*n. fds-nth fds-zeta n* / *f n*) = (∑*n. 1* / *f n* :: ′*a* :: *real-normed-field*)
by (*intro suminf-cong*) (*auto simp*: *fds-nth-zeta*)

lemma *fds-converges-shift* [*simp*]:
fixes *z* :: ′*a* :: {*banach*, *nat-power-field*, *real-normed-field*}
shows *fds-converges* (*fds-shift c f*) *z* ⟷ *fds-converges f* (*z* − *c*)
unfolding *fds-converges-def*
by (*intro summable-cong*)
(*auto intro*: *eventually-mono* [*OF eventually-gt-at-top*[*of 0*::*nat*]] *simp*: *nat-power-diff*)

lemma *fds-abs-converges-shift* [*simp*]:
fixes *z* :: *'a* :: {*banach*, *nat-power-field*, *real-normed-field*}
shows *fds-abs-converges* (*fds-shift c f*) *z* ⟷ *fds-abs-converges f* (*z* − *c*)
unfolding *fds-abs-converges-def*
by (*intro summable-cong*)
(*auto intro*: *eventually-mono* [*OF eventually-gt-at-top*[*of 0*::*nat*]] *simp*: *nat-power-diff*)

lemma *fds-eval-shift* [*simp*]:
fixes *z* :: *'a* :: {*banach*, *nat-power-field*, *real-normed-field*}
shows *eval-fds* (*fds-shift c f*) *z* = *eval-fds f* (*z* − *c*)
unfolding *eval-fds-def*
proof (*rule suminf-cong*, *goal-cases*)
case (*1 n*)
show *?case* **by** (*cases n* = *0*) (*simp-all add*: *nat-power-diff*)
qed

lemma *fds-converges-scale* [*simp*]:
fixes *z* :: *'a* :: {*banach*, *nat-power-field*, *real-normed-field*}
assumes *c*: *c* > *0*
shows *fds-converges* (*fds-scale c f*) *z* ⟷ *fds-converges f* (*of-nat c* ∗ *z*)
proof −
have *fds-converges* (*fds-scale c f*) *z* ⟷
summable (λ*n*. *fds-nth* (*fds-scale c f*) (*n* ^ *c*) / *nat-power* (*n* ^ *c*) *z*)
(**is** - = *summable ?g*) **unfolding** *fds-converges-def*
by (*rule summable-mono-reindex* [*symmetric*])
(*insert c*, *auto simp*: *fds-nth-scale is-nth-power-def strict-mono-def power-strict-mono*)
also have *?g* = (λ*n*. *fds-nth f n* / *nat-power n* (*of-nat c* ∗ *z*))
proof (*intro ext*, *goal-cases*)
case (*1 n*)
thus *?case* **using** *c*
by (*cases n* = *0*) (*simp-all add*: *nat-power-power-left nat-power-power* [*symmetric*]
mult-ac)
qed
finally show *?thesis* **by** (*simp add*: *fds-converges-def*)
qed

lemma *fds-abs-converges-scale* [*simp*]:
fixes *z* :: *'a* :: {*banach*, *nat-power-field*, *real-normed-field*}
assumes *c*: *c* > *0*
shows *fds-abs-converges* (*fds-scale c f*) *z* ⟷ *fds-abs-converges f* (*of-nat c* ∗ *z*)
proof −
have *fds-abs-converges* (*fds-scale c f*) *z* ⟷
summable (λ*n*. *norm* (*fds-nth* (*fds-scale c f*) (*n* ^ *c*) / *nat-power* (*n* ^ *c*)
z))
(**is** - = *summable ?g*) **unfolding** *fds-abs-converges-def*
by (*rule summable-mono-reindex* [*symmetric*])
(*insert c*, *auto simp*: *fds-nth-scale is-nth-power-def strict-mono-def power-strict-mono*)
also have *?g* = (λ*n*. *norm* (*fds-nth f n* / *nat-power n* (*of-nat c* ∗ *z*)))

proof (*intro ext*, *goal-cases*)
case (*1 n*)
thus *?case* **using** *c*
by (*cases n* = *0*) (*simp-all add*: *nat-power-power-left nat-power-power* [*symmetric*] *mult-ac*)
qed
finally show *?thesis* **by** (*simp add*: *fds-abs-converges-def*)
qed

lemma *eval-fds-scale* [*simp*]:
fixes *z* :: *'a* :: {*banach*, *nat-power-field*, *real-normed-field*}
assumes *c*: *c* > *0*
shows *eval-fds* (*fds-scale c f*) *z* = *eval-fds f* (*of-nat c* ∗ *z*)
proof −
have *eval-fds* (*fds-scale c f*) *z* =
($\sum$ *n*. *fds-nth* (*fds-scale c f*) (*n* ^ *c*) / *nat-power* (*n* ^ *c*) *z*)
unfolding *eval-fds-def*
by (*rule suminf-mono-reindex* [*symmetric*])
(*insert c*, *auto simp*: *fds-nth-scale is-nth-power-def strict-mono-def power-strict-mono*)
also have . . . = ($\sum$ *n*. *fds-nth f n* / *nat-power n* (*of-nat c* ∗ *z*))
proof (*intro suminf-cong*, *goal-cases*)
case (*1 n*)
thus *?case* **using** *c*
by (*cases n* = *0*) (*simp-all add*: *nat-power-power-left nat-power-power* [*symmetric*] *mult-ac*)
qed
finally show *?thesis* **by** (*simp add*: *eval-fds-def*)
qed

lemma *fds-abs-converges-integral*:
assumes *fds-abs-converges f s*
shows *fds-abs-converges* (*fds-integral c f*) *s*
unfolding *fds-abs-converges-def*
proof (*rule summable-comparison-test-ev*)
show *summable* (λ*n*. *norm* (*fds-nth f n* / *nat-power n s*))
using *assms* **by** (*simp add*: *fds-abs-converges-def*)
show *eventually* (λ*n*. *norm* (*norm* (*fds-nth* (*fds-integral c f*) *n* / *nat-power n s*))
≤ *norm* (*fds-nth f n* / *nat-power n s*)) *at-top*
using *eventually-gt-at-top*[*of 3*]
proof *eventually-elim*
case (*elim n*)
hence *ln n* ≥ *ln* (*exp 1*)
using *exp-le* **by** (*subst ln-le-cancel-iff*) *auto*
hence *norm* (*fds-nth f n*) ∗ *1* ≤ *norm* (*fds-nth f n*) ∗ *ln* (*real n*)
by (*intro mult-left-mono*) *auto*
with *elim* **show** *?case*
by (*simp-all add*: *fds-integral-def norm-divide divide-simps*)
qed
qed

lemma *fds-abs-converges-ln*:
  assumes *fds-abs-converges* (*fds-deriv f* / *f*) *s*
  shows *fds-abs-converges* (*fds-ln l f*) *s*
  using *assms* **unfolding** *fds-ln-def* **by** (*intro fds-abs-converges-integral*)

end

5 The Möbius μ function

theory *Moebius-Mu*
imports
  Main
  HOL−Number-Theory.Number-Theory
  HOL−Computational-Algebra.Squarefree
  Dirichlet-Series
  Dirichlet-Misc
begin

definition *moebius-mu* :: *nat* ⇒ *'a* :: *comm-ring-1* **where**
  moebius-mu n =
    (*if squarefree n then* (−*1*) ^ *card* (*prime-factors n*) *else 0*)

lemma *abs-moebius-mu-le*: *abs* (*moebius-mu n* :: *'a* :: {*linordered-idom*}) ≤ *1*
  by (*auto simp add*: *moebius-mu-def*)

lemma *of-int-moebius-mu* [*simp*]: *of-int* (*moebius-mu n*) = *moebius-mu n*
  by (*simp add*: *moebius-mu-def*)

lemma *minus-1-power-ring-neq-zero* [*simp*]: (− *1* :: *'a* :: *ring-1*) ^ *n* ≠ *0*
  by (*cases even n*) *simp-all*

lemma *moebius-mu-0* [*simp*]: *moebius-mu 0* = *0*
  by (*simp add*: *moebius-mu-def*)

lemma *fds-nth-fds-moebius-mu* [*simp*]: *fds-nth* (*fds moebius-mu*) = *moebius-mu*
  by (*simp add*: *fun-eq-iff fds-nth-fds*)

lemma *prime-factors-Suc-0* [*simp*]: *prime-factors* (*Suc 0*) = {}
  by *simp*

lemma *moebius-mu-Suc-0* [*simp*]: *moebius-mu* (*Suc 0*) = *1*
  by (*simp add*: *moebius-mu-def*)

lemma *moebius-mu-1* [*simp*]: *moebius-mu 1* = *1*
  by (*simp add*: *moebius-mu-def*)

lemma *moebius-mu-eq-zero-iff*: *moebius-mu n* = *0* ⟷ ¬*squarefree n*
  by (*simp add*: *moebius-mu-def*)

```
lemma moebius-mu-not-squarefree [simp]: ¬squarefree n ⟹ moebius-mu n = 0
  by (simp add: moebius-mu-def)

lemma moebius-mu-power:
  assumes a > 1 n > 1
  shows   moebius-mu (a ^ n) = 0
proof -
  from assms have a ^ 2 dvd a ^ n by (simp add: le-imp-power-dvd)
  with moebius-mu-eq-zero-iff[of a ^ n] and ‹a > 1› show ?thesis by (auto simp:
squarefree-def)
qed

lemma moebius-mu-power':
  moebius-mu (a ^ n) = (if a = 1 ∨ n = 0 then 1 else if n = 1 then moebius-mu
a else 0)
  by (simp add: squarefree-power-iff)

lemma moebius-mu-squarefree-eq:
  squarefree n ⟹ moebius-mu n = (−1) ^ card (prime-factors n)
  by (simp add: moebius-mu-def split: if-splits)

lemma moebius-mu-squarefree-eq':
  assumes squarefree n
  shows   moebius-mu n = (−1) ^ size (prime-factorization n)
proof -
  let ?P = prime-factorization n
  from assms have [simp]: n > 0 by (auto intro!: Nat.gr0I)
 have size ?P = sum (count ?P) (set-mset ?P) by (rule size-multiset-overloaded-eq)
  also from assms have ... = sum (λ-. 1) (set-mset ?P)
    by (intro sum.cong refl, subst count-prime-factorization-prime)
       (auto simp: moebius-mu-eq-zero-iff squarefree-factorial-semiring')
  also have ... = card (set-mset ?P) by simp
  finally show ?thesis by (simp add: moebius-mu-squarefree-eq[OF assms])
qed

lemma sum-moebius-mu-divisors:
  assumes n > 1
  shows   (∑d | d dvd n. moebius-mu d) = (0 :: 'a :: comm-ring-1)
proof -
  have (∑d | d dvd n. moebius-mu d :: int) =
         (∑d ∈ Prod ‘ {P. P ⊆ prime-factors n}. moebius-mu d)
  proof (rule sum.mono-neutral-right; safe?)
    fix A assume A: A ⊆ prime-factors n
    from A have [simp]: finite A by (rule finite-subset) auto
    from A have A': x > 0 prime x if x ∈ A for x using that
      by (auto simp: prime-factors-multiplicity prime-gt-0-nat)
    from A' have A-nz: ∏A ≠ 0 by (intro notI) auto
    from A' have prime-factorization (∏A) = sum prime-factorization A
```

by (*subst prime-factorization-prod*) (*auto dest*: *finite-subset*)
also from *A′* **have** . . . = *sum* (λ*x*. {#*x*#}) *A*
by (*intro sum.cong refl*) (*auto simp*: *prime-factorization-prime*)
also have . . . = *mset-set A* **by** *simp*
also from *A* **have** . . . ⊆# *mset-set* (*prime-factors n*)
by (*rule subset-imp-msubset-mset-set*) *simp-all*
also have . . . ⊆# *prime-factorization n* **by** (*rule mset-set-set-mset-msubset*)
finally show ∏*A dvd n* **using** *A-nz*
by (*intro prime-factorization-subset-imp-dvd*) *auto*
next
fix *x* **assume** *x*: *x* ∉ *Prod* ‘ {*P*. *P* ⊆ *prime-factors n*} *x dvd n*
from *x assms* **have** [*simp*]: *x* > *0* **by** (*auto intro*!: *Nat.gr0I*)
{
assume *nz*: *moebius-mu x* ≠ *0*
have (∏(*set-mset* (*prime-factorization x*))) = (∏*p*∈*prime-factors x*. *p* ^ *multiplicity p x*)
using *nz* **by** (*intro prod.cong refl*)
(*auto simp*: *moebius-mu-eq-zero-iff squarefree-factorial-semiring′*)
also have . . . = *x* **by** (*intro Primes.prime-factorization-nat* [*symmetric*]) *auto*
finally have *x* = ∏(*prime-factors x*) *prime-factors x* ⊆ *prime-factors n*
using *dvd-prime-factors*[*of n x*] *assms* ‹*x dvd n*› **by** *auto*
hence *x* ∈ *Prod* ‘ {*P*. *P* ⊆ *prime-factors n*} **by** *blast*
with *x*(*1*) **have** *False* **by** *contradiction*
}
thus *moebius-mu x* = *0* **by** *blast*
qed (*insert assms*, *auto*)
also have . . . = (∑*P* | *P* ⊆ *prime-factors n*. *moebius-mu* (∏*P*))
by (*subst sum.reindex*) (*auto intro*!: *inj-on-Prod-primes dest*: *finite-subset*)
also have . . . = (∑*P* | *P* ⊆ *prime-factors n*. (−*1*) ^ *card P*)
proof (*intro sum.cong refl*)
fix *P* **assume** *P*: *P* ∈ {*P*. *P* ⊆ *prime-factors n*}
hence [*simp*]: *finite P* **by** (*auto dest*: *finite-subset*)
from *P* **have** *prime*: *prime p* **if** *p* ∈ *P* **for** *p* **using** *that* **by** (*auto simp*: *prime-factors-dvd*)
hence *squarefree* (∏*P*)
by (*intro squarefree-prod-coprime prime-imp-coprime squarefree-prime*)
(*auto simp*: *primes-dvd-imp-eq*)
hence *moebius-mu* (∏*P*) = (−*1*) ^ *card* (*prime-factors* (∏*P*))
by (*rule moebius-mu-squarefree-eq*)
also from *P* **have** *prime-factors* (∏*P*) = *P*
by (*subst prime-factors-prod*) (*auto simp*: *prime-factorization-prime prime*)
finally show *moebius-mu* (∏*P*) = (−*1*) ^ *card P* **.**
qed
also have {*P*. *P* ⊆ *prime-factors n*} =
{*P*. *P* ⊆ *prime-factors n* ∧ *even* (*card P*)} ∪ {*P*. *P* ⊆ *prime-factors n* ∧ *odd* (*card P*)}
(**is** - = *?A* ∪ *?B*) **by** *blast*
also have (∑*P* ∈ . . . . (−*1*) ^ *card P*) = (∑*P* ∈ *?A*. (−*1*) ^ *card P*) + (∑*P* ∈ *?B*. (−*1*) ^ *card P*)

by (*intro sum.union-disjoint*) *auto*
also have ($\sum P \in ?A.\ (-1)$ ^ *card P* :: *int*) = ($\sum P \in ?A.\ 1$) **by** (*intro sum.cong refl*) *auto*
also have ... = *int* (*card ?A*) **by** *simp*
also have ($\sum P \in ?B.\ (-1)$ ^ *card P* :: *int*) = ($\sum P \in ?B.\ -1$) **by** (*intro sum.cong refl*) *auto*
also have ... = −*int* (*card ?B*) **by** *simp*
also have *card ?B* = *card ?A*
by (*rule card-even-odd-subset* [*symmetric*])
(*insert assms*, *auto simp*: *prime-factorization-empty-iff*)
also have *int* (*card ?A*) + (− *int* (*card ?A*)) = *0* **by** *simp*
finally have ($\sum d$ | *d dvd n*. *of-int* (*moebius-mu d*) :: *'a*) = *0*
unfolding *of-int-sum* [*symmetric*] **by** (*simp only*: *of-int-0*)
thus *?thesis* **by** *simp*
qed

lemma *sum-moebius-mu-divisors'*:
($\sum d$ | *d dvd n*. *moebius-mu d*) = (*if n* = *1 then 1 else 0*)
proof −
have *n* = *0* ∨ *n* = *1* ∨ *n* > *1* **by** *force*
thus *?thesis* **using** *sum-moebius-mu-divisors*[*of n*] **by** *auto*
qed

lemma *fds-zeta-times-moebius-mu*: *fds-zeta* ∗ *fds moebius-mu* = *1*
proof
fix *n* :: *nat* **assume** *n*: *n* > *0*
from *n* **have** *fds-nth* (*fds-zeta* ∗ *fds moebius-mu* :: *'a fds*) *n* = ($\sum d$ | *d dvd n*. *moebius-mu d*)
unfolding *fds-nth-mult dirichlet-prod-altdef1*
by (*intro sum.cong refl*) (*auto simp*: *fds-nth-fds elim*: *dvdE*)
also have ... = *fds-nth 1 n* **by** (*simp add*: *sum-moebius-mu-divisors'*)
finally show *fds-nth* (*fds-zeta* ∗ *fds moebius-mu* :: *'a fds*) *n* = *fds-nth 1 n* **.**
qed

lemma *fds-moebius-inverse-zeta*:
fds moebius-mu = *inverse* (*fds-zeta* :: *'a* :: *field fds*)
using *fds-right-inverse-unique fds-zeta-times-moebius-mu* **by** *blast*

lemma *moebius-mu-formula-real*: (*moebius-mu n* :: *real*) = *dirichlet-inverse* (λ-. *1*) *1 n*
proof −
have *moebius-mu n* = (*fds-nth* (*fds moebius-mu*) *n* :: *real*) **by** *simp*
also have *fds moebius-mu* = (*inverse fds-zeta* :: *real fds*) **by** (*fact fds-moebius-inverse-zeta*)
also have *fds-nth* ... *n* = *dirichlet-inverse* (*fds-nth fds-zeta*) *1 n*
unfolding *fds-nth-inverse* **by** *simp*
also have ... = *dirichlet-inverse* (λ-. *1*) *1 n* **by** (*rule dirichlet-inverse-cong*) *simp-all*
finally show *?thesis* **.**
qed

lemma *moebius-mu-formula-int*: *moebius-mu n* = *dirichlet-inverse* (λ-. *1* :: *int*) *1* *n*
proof −
  have *real-of-int* (*moebius-mu n*) = *moebius-mu n* **by** *simp*
  also have ... = *dirichlet-inverse* (λ-. *1*) *1 n* **by** (*fact moebius-mu-formula-real*)
  also have ... = *real-of-int* (*dirichlet-inverse* (λ-. *1*) *1 n*)
   by (*induction n rule*: *dirichlet-inverse-induct*) (*simp-all add*: *dirichlet-inverse-gt-1*)
  finally show *?thesis* **by** (*subst* (*asm*) *of-int-eq-iff*)
qed

lemma *moebius-mu-formula*: *moebius-mu n* = *dirichlet-inverse* (λ-. *1*) *1 n*
  by (*subst of-int-moebius-mu* [*symmetric*], *subst moebius-mu-formula-int*)
    (*simp add*: *of-int-dirichlet-inverse*)

interpretation *moebius-mu*: *multiplicative-function moebius-mu*
proof −
  have *multiplicative-function* (*dirichlet-inverse* (λ*n*. *if n* = *0 then 0 else 1* :: ′*a*) *1*)
    by (*rule multiplicative-dirichlet-inverse*, *standard*) *simp-all*
  also have *dirichlet-inverse* (λ*n*. *if n* = *0 then 0 else 1* :: ′*a*) *1* = *moebius-mu*
    by (*auto simp*: *fun-eq-iff moebius-mu-formula*)
  finally show *multiplicative-function* (*moebius-mu* :: *nat* ⇒ ′*a*) **.**
qed

interpretation *moebius-mu*:
  multiplicative-function′ moebius-mu λ*p k*. *if k* = *1 then* −*1 else 0* λ-. −*1*
proof
  fix *p k* :: *nat* **assume** *prime p k* > *0*
  moreover from *this* **have** *moebius-mu p* = −*1*
    by (*simp add*: *moebius-mu-def prime-factorization-prime squarefree-prime*)
  ultimately show *moebius-mu* (*p* ^ *k*) = (*if k* = *1 then* − *1 else 0*)
    by (*auto simp*: *moebius-mu-power′*)
qed *auto*

lemma *moebius-mu-2* [*simp*]: *moebius-mu 2* = −*1*
  and *moebius-mu-3* [*simp*]: *moebius-mu 3* = −*1*
  by (*rule moebius-mu.prime*; *simp*)+

lemma *moebius-mu-code* [*code*]:
  moebius-mu n = *of-int* (*dirichlet-inverse* (λ-. *1* :: *int*) *1 n*)
  by (*subst moebius-mu-formula-int* [*symmetric*]) *simp*

lemma *fds-moebius-inversion*: *f* = *fds moebius-mu* ∗ *g* ⟷ *g* = *f* ∗ *fds-zeta*
 by (*metis fds-zeta-times-moebius-mu mult.commute mult.left-commute mult.right-neutral*)

lemma *moebius-inversion*:

assumes $\bigwedge n.\ n > 0 \Longrightarrow g\ n = (\sum d \mid d\ dvd\ n.\ f\ d)\ n > 0$
shows $f\ n = dirichlet\text{-}prod\ moebius\text{-}mu\ g\ n$
proof −
from *assms* **have** *fds g = fds f ∗ fds-zeta*
by (*intro fds-eqI*) (*simp add*: *fds-nth-mult dirichlet-prod-def*)
thus *?thesis* **using** *assms*
by (*subst* (*asm*) *fds-moebius-inversion* [*symmetric*]) (*simp add*: *fds-eq-iff fds-nth-mult*)
qed

lemma *fds-mangoldt*: *fds mangoldt = fds moebius-mu ∗ fds* (*λn. of-real* (*ln* (*real n*)))
by (*subst fds-moebius-inversion*) (*rule fds-mangoldt-times-zeta* [*symmetric*])

lemma *sum-divisors-moebius-mu-times-multiplicative*:
fixes *f* :: *nat* ⇒ *'a* :: {*comm-ring-1*}
assumes *multiplicative-function f n > 0*
shows $(\sum d \mid d\ dvd\ n.\ moebius\text{-}mu\ d * f\ d) = (\prod p{\in}prime\text{-}factors\ n.\ 1 - f\ p)$
proof −
define *g* **where** $g = (\lambda n. \sum d \mid d\ dvd\ n.\ moebius\text{-}mu\ d * f\ d)$
define *g'* **where** *g' = dirichlet-prod* (*λn. moebius-mu n ∗ f n*) (*λn. if n = 0 then 0 else 1*)
interpret *f*: *multiplicative-function f* **by** *fact*
have *multiplicative-function* (*λn. if n = 0 then 0 else 1* :: *'a*)
by *standard auto*
interpret *multiplicative-function g'* **unfolding** *g'-def*
by (*intro multiplicative-dirichlet-prod multiplicative-function-mult*
moebius-mu.multiplicative-function-axioms assms) *fact+*

have *g'-primepow*: *g'* (*p ^ k*) *= 1 − f p* **if** *prime p k > 0* **for** *p k*
proof −
have $g'\ (p\ \hat{}\ k) = (\sum i{\le}k.\ moebius\text{-}mu\ (p\ \hat{}\ i) * f\ (p\ \hat{}\ i))$
using *that* **by** (*simp add*: *g'-def dirichlet-prod-prime-power*)
also have $\ldots = (\sum i{\in}\{0,\ 1\}.\ moebius\text{-}mu\ (p\ \hat{}\ i) * f\ (p\ \hat{}\ i))$
using *that* **by** (*intro sum.mono-neutral-right*) (*auto simp*: *moebius-mu-power'*)
also have … *= 1 − f p*
using *that* **by** (*simp add*: *moebius-mu.prime*)
finally show *?thesis* **.**
qed

have *g' n = g n*
by (*simp add*: *g-def g'-def dirichlet-prod-def*)
also from *assms* **have** $g'\ n = (\prod p{\in}prime\text{-}factors\ n.\ g'\ (p\ \hat{}\ multiplicity\ p\ n))$
by (*intro prod-prime-factors*) *auto*
also have $\ldots = (\prod p{\in}prime\text{-}factors\ n.\ 1 - f\ p)$
by (*intro prod.cong*) (*auto simp*: *g'-primepow prime-factors-multiplicity*)
finally show *?thesis* **by** (*simp add*: *g-def*)
qed

lemma *completely-multiplicative-iff-inverse-moebius-mu*:
  fixes *f* :: *nat* ⇒ *'a* :: {*comm-ring-1*, *ring-no-zero-divisors*}
  assumes *multiplicative-function f*
  defines *g* ≡ *dirichlet-inverse f 1*
  shows *completely-multiplicative-function f* ⟷
    (∀ *n*. *g n* = *moebius-mu n* ∗ *f n*)
proof −
  interpret *multiplicative-function f* **by** *fact*
  show *?thesis*
  proof *safe*
    assume *completely-multiplicative-function f*
    then interpret *completely-multiplicative-function f* **.**
    have [*simp*]: *fds f* ≠ *0* **by** (*auto simp*: *fds-eq-iff*)

    have *fds* (λ*n*. *moebius-mu n* ∗ *f n*) ∗ *fds f* = *1*
    proof
      fix *n* :: *nat*
      have *fds-nth* (*fds* (λ*n*. *moebius-mu n* ∗ *f n*) ∗ *fds f*) *n* =
          (∑ (*r*, *d*) | *r* ∗ *d* = *n*. *moebius-mu r* ∗ *f* (*r* ∗ *d*))
        by (*simp add*: *fds-eq-iff fds-nth-mult fds-nth-fds dirichlet-prod-altdef2 mult mult.assoc*)
      also have . . . = (∑ (*r*, *d*) | *r* ∗ *d* = *n*. *moebius-mu r* ∗ *f n*)
        by (*intro sum.cong*) *auto*
      also have . . . = *dirichlet-prod moebius-mu* (λ-. *1*) *n* ∗ *f n*
        by (*simp add*: *dirichlet-prod-altdef2 sum-distrib-right case-prod-unfold mult*)
      also have *dirichlet-prod moebius-mu* (λ-. *1*) *n* = *fds-nth* (*fds moebius-mu* ∗ *fds-zeta*) *n*
        by (*simp add*: *fds-nth-mult*)
      also have *fds moebius-mu* ∗ *fds-zeta* = *1*
        by (*simp add*: *mult-ac fds-zeta-times-moebius-mu*)
      also have *fds-nth 1 n* ∗ *f n* = *fds-nth 1 n*
        by (*auto simp*: *fds-eq-iff fds-nth-one*)
      finally show *fds-nth* (*fds* (λ*n*. *moebius-mu n* ∗ *f n*) ∗ *fds f*) *n* = *fds-nth 1 n* **.**
    qed
    also have *1* = *fds g* ∗ *fds f*
      by (*auto simp*: *fds-eq-iff g-def fds-nth-mult dirichlet-prod-inverse'*)
    finally have *fds g* = *fds* (λ*n*. *moebius-mu n* ∗ *f n*)
      by (*subst* (*asm*) *mult-cancel-right*) *auto*
    thus *g n* = *moebius-mu n* ∗ *f n* **for** *n*
      by (*cases n* = *0*) (*auto simp*: *fds-eq-iff g-def*)
  next
    assume *g*: ∀ *n*. *g n* = *moebius-mu n* ∗ *f n*
    show *completely-multiplicative-function f*
    proof (*rule completely-multiplicativeI*)
      fix *p k* :: *nat* **assume** *pk*: *prime p k* > *0*
      show *f* (*p* ^ *k*) = *f p* ^ *k*
      proof (*induction k*)

```
        case (Suc k)
        have eq: dirichlet-prod g f n = 0 if n ≠ 1 for n
          unfolding g-def using dirichlet-prod-inverse'[of f 1] that by auto
        have dirichlet-prod g f (p ^ Suc k) = 0
          using pk by (intro eq) auto
        also have dirichlet-prod g f (p ^ Suc k) = (∑ i≤Suc k. g (p ^ i) * f (p ^
(Suc k − i)))
          by (intro dirichlet-prod-prime-power) fact+
        also have ... = (∑ i≤Suc k. moebius-mu (p ^ i) * f (p ^ i) * f (p ^(Suc
k − i)))
          by (intro sum.cong refl, subst g) auto
        also have ... = (∑ i∈{0, 1}. moebius-mu (p ^ i) * f (p ^ i) * f (p ^(Suc
k − i)))
        using pk by (intro sum.mono-neutral-right) (auto simp: moebius-mu-power')
        also have ... = f (p ^ Suc k) − f p ^ Suc k
          using pk Suc.IH by (auto simp: moebius-mu.prime)
        finally show f (p ^ Suc k) = f p ^ Suc k by simp
      qed auto
    qed
  qed
qed

lemma completely-multiplicative-fds-inverse:
  fixes f :: nat ⇒ 'a :: field
  assumes completely-multiplicative-function f
  shows   inverse (fds f) = fds (λn. moebius-mu n * f n)
proof −
  interpret completely-multiplicative-function f by fact
  from assms show ?thesis
    by (subst (asm) completely-multiplicative-iff-inverse-moebius-mu)
       (auto simp: inverse-fds-def multiplicative-function-axioms)
qed

lemma completely-multiplicative-fds-inverse':
  fixes f :: 'a :: field fds
  assumes completely-multiplicative-function (fds-nth f)
  shows   inverse f = fds (λn. moebius-mu n * fds-nth f n)
  by (metis assms completely-multiplicative-fds-inverse fds-fds-nth)


context
  includes fds-syntax
begin

lemma selberg-aux:
  (χ n. of-real ((ln n)^2)) * fds moebius-mu =
    (fds mangoldt)^2 − fds-deriv (fds mangoldt :: 'a :: {comm-ring-1,real-algebra-1}
fds)
proof −
```

have (*χ n. of-real* (*ln* (*real n*) ^ *2*)) = *fds-deriv* (*fds-deriv fds-zeta* :: *'a fds*)
by (*rule fds-eqI*) (*simp add*: *fds-nth-fds fds-nth-deriv power2-eq-square scaleR-conv-of-real*)
also have ... = (*fds mangoldt* ^ *2* − *fds-deriv* (*fds mangoldt*)) ∗ *fds-zeta*
by (*simp add*: *fds-deriv-zeta algebra-simps power2-eq-square*)
also have ... ∗ *fds moebius-mu* = ((*fds mangoldt*)^2 − *fds-deriv* (*fds mangoldt*))
∗
(*fds-zeta* ∗ *fds moebius-mu*) **by** (*simp add*: *mult-ac*)
also have *fds-zeta* ∗ *fds moebius-mu* = (*1* :: *'a fds*) **by** (*fact fds-zeta-times-moebius-mu*)
finally show *?thesis* **by** *simp*
qed

lemma *selberg-aux'*:
mangoldt n ∗ *of-real* (*ln n*) + (*mangoldt* ⋆ *mangoldt*) *n* =
((*moebius-mu* ⋆ (*λb. of-real* (*ln b*) ^ *2*)) *n*
:: *'a* :: {*comm-ring-1*,*real-algebra-1*}) **if** *n* > *0*
using *selberg-aux* [*symmetric*] *that*
by (*auto simp add*: *fds-eq-iff fds-nth-mult power2-eq-square fds-nth-deriv*
dirichlet-prod-commutes algebra-simps scaleR-conv-of-real)

end

end

6 Euler's ϕ function

theory *More-Totient*
imports
Moebius-Mu
HOL−Number-Theory.Number-Theory
begin

lemma *fds-totient-times-zeta*:
fds (*λn. of-nat* (*totient n*) :: *'a* :: *comm-semiring-1*) ∗ *fds-zeta* = *fds of-nat*
proof
fix *n* :: *nat* **assume** *n*: *n* > *0*
have *fds-nth* (*fds* (*λn. of-nat* (*totient n*)) ∗ *fds-zeta*) *n* =
dirichlet-prod (*λn. of-nat* (*totient n*)) (*λ-. 1*) *n*
by (*simp add*: *fds-nth-mult*)
also from *n* **have** ... = *fds-nth* (*fds of-nat*) *n*
by (*simp add*: *fds-nth-fds dirichlet-prod-def totient-divisor-sum of-nat-sum* [*symmetric*]
del: *of-nat-sum*)
finally show *fds-nth* (*fds* (*λn. of-nat* (*totient n*)) ∗ *fds-zeta*) *n* = *fds-nth* (*fds*
of-nat) *n* **.**
qed

lemma *fds-totient-times-zeta'*: *fds totient* ∗ *fds-zeta* = *fds id*
using *fds-totient-times-zeta*[**where** *'a* = *nat*] **by** *simp*

lemma *fds-totient*: *fds* (*λn. of-nat* (*totient n*)) = *fds of-nat* ∗ *fds moebius-mu*

proof −
  have fds (λn. of-nat (totient n)) ∗ fds-zeta ∗ fds moebius-mu = fds of-nat ∗ fds moebius-mu
    by (simp add: fds-totient-times-zeta)
  also have fds (λn. of-nat (totient n)) ∗ fds-zeta ∗ fds moebius-mu =
              fds (λn. of-nat (totient n))
    by (simp only: mult.assoc fds-zeta-times-moebius-mu mult-1-right)
  finally show ?thesis .
qed

lemma totient-conv-moebius-mu:
  int (totient n) = dirichlet-prod moebius-mu int n
proof (cases n = 0)
  case False
  show ?thesis
    by (rule moebius-inversion)
       (insert False, simp-all add: of-nat-sum [symmetric] totient-divisor-sum del: of-nat-sum)
qed simp-all

interpretation totient: multiplicative-function totient
proof −
  have multiplicative-function int by standard simp-all
  hence multiplicative-function (dirichlet-prod moebius-mu int)
   by (intro multiplicative-dirichlet-prod moebius-mu.multiplicative-function-axioms)
  also have dirichlet-prod moebius-mu int = (λn. int (totient n))
    by (simp add: fun-eq-iff totient-conv-moebius-mu)
  finally show multiplicative-function totient by (rule multiplicative-function-of-natD)
qed

lemma even-prime-nat: prime p ⟹ even p ⟹ p = (2::nat)
  using prime-odd-nat[of p] prime-gt-1-nat[of p] by (cases p = 2) auto

lemma twopow-dvd-totient:
  fixes n :: nat
  assumes n > 0
  defines k ≡ card {p∈prime-factors n. odd p}
  shows   2 ^ k dvd totient n
proof −
  define P where P = {p∈prime-factors n. odd p}
  define P′ where P′ = {p∈prime-factors n. even p}
  define r where r = (λp. multiplicity p n)
  from ‹n > 0› have totient n = (∏ p∈prime-factors n. totient (p ^ r p))
    unfolding r-def by (rule totient.prod-prime-factors)
  also have prime-factors n = P ∪ P′
    by (auto simp: P-def P′-def)
  also have (∏ p∈. . . . totient (p ^ r p)) =
              (∏ p∈P. totient (p ^ r p)) ∗ (∏ p∈P′. totient (p ^ r p))
    by (subst prod.union-disjoint) (auto simp: P-def P′-def)

```
  finally have eq: totient n = ... .

  have p ^ r p > 2 if p ∈ P for p
  proof -
    have p ≠ 2 using that by (auto simp: P-def)
    moreover have p > 1 using prime-gt-1-nat[of p] that by (auto simp: P-def)
    ultimately have 2 < p by linarith
    also have p = p ^ 1 by simp
    also have p ^ 1 ≤ p ^ r p
      using that prime-gt-1-nat[of p]
     by (intro power-increasing) (auto simp: P-def prime-factors-multiplicity r-def)
    finally show ?thesis .
  qed
  hence (∏ p∈P. 2) dvd (∏ p∈P. totient (p ^ r p))
    by (intro prod-dvd-prod totient-even)
  hence 2 ^ card P dvd (∏ p∈P. totient (p ^ r p))
    by simp
  also have ... dvd (∏ p∈P. totient (p ^ r p)) * (∏ p∈P'. totient (p ^ r p))
    by simp
  also have ... = totient n
    by (rule eq [symmetric])
  finally show ?thesis unfolding k-def P-def .
qed

lemma totient-conv-moebius-mu':
  assumes n > (0::nat)
  shows   real (totient n) = real n * (∑ d | d dvd n. moebius-mu d / real d)
proof -
  have real (totient n) = of-int (int (totient n)) by simp
  also have int (totient n) = (∑ d | d dvd n. moebius-mu d * int (n div d))
    using totient-conv-moebius-mu by (simp add: dirichlet-prod-def assms)
  also have real-of-int (∑ d | d dvd n. moebius-mu d * int (n div d)) =
              (∑ d | d dvd n. moebius-mu d * real (n div d)) by simp
  also have ... = (∑ d | d dvd n. real n * moebius-mu d / real d)
    by (rule sum.cong) (simp-all add: field-char-0-class.of-nat-div)
  also have ... = real n * (∑ d | d dvd n. moebius-mu d / real d)
    by (simp add: sum-distrib-left)
  finally show ?thesis .
qed

lemma totient-prime-power-Suc:
  assumes prime p
  shows   totient (p ^ Suc n) = p ^ Suc n - p ^ n
proof -
  have totient (p ^ Suc n) = p ^ Suc n - card ((*) p ` {0<..p ^ n})
    unfolding totient-def totatives-prime-power-Suc[OF assms]
    by (subst card-Diff-subset) (insert assms, auto simp: prime-gt-0-nat)
  also from assms have card ((*) p ` {0<..p^n}) = p ^ n
    by (subst card-image) (auto simp: inj-on-def)
```

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

interpretation *totient*: *multiplicative-function′ totient λp k. p ^ k − p ^ (k − 1) λp. p − 1*
proof
fix *p k* :: *nat* **assume** *prime p k > 0*
thus *totient* (*p ^ k*) = *p ^ k − p ^ (k − 1)*
by (*cases k*) (*simp-all add*: *totient-prime-power-Suc del*: *power-Suc*)
qed *simp-all*

end

7 The Liouville λ function

theory *Liouville-Lambda*
imports
HOL−Computational-Algebra.Computational-Algebra
HOL−Number-Theory.Number-Theory
Dirichlet-Series
Multiplicative-Function
Moebius-Mu
begin

definition *liouville-lambda* :: *nat* ⇒ *′a* :: *comm-ring-1* **where**
liouville-lambda n = (*if n* = *0 then 0 else* (−*1*) *^ size* (*prime-factorization n*))

interpretation *liouville-lambda*: *completely-multiplicative-function′ liouville-lambda λ-. −1*
proof
fix *a b* :: *nat* **assume** *a > 1 b > 1*
thus *liouville-lambda* (*a ∗ b*) = *liouville-lambda a ∗ liouville-lambda b*
by (*simp add*: *liouville-lambda-def prime-factorization-mult power-add*)
qed (*simp-all add*: *liouville-lambda-def prime-factorization-prime One-nat-def* [*symmetric*]

del: *One-nat-def*)

lemma *liouville-lambda-prime* [*simp*]: *prime p* ⟹ *liouville-lambda p* = −*1*
by (*simp add*: *liouville-lambda-def prime-factorization-prime*)

lemma *liouville-lambda-prime-power* [*simp*]: *prime p* ⟹ *liouville-lambda* (*p ^ k*) = (−*1*) *^ k*
by (*simp add*: *liouville-lambda-def prime-factorization-prime-power*)

lemma *liouville-lambda-squarefree*: *squarefree n* ⟹ *liouville-lambda n* = *moebius-mu n*
by (*auto simp*: *liouville-lambda-def moebius-mu-squarefree-eq′ intro*!: *Nat.gr0I*)

lemma *power-neg-one-If*: (−*1*) *^ n* = (*if even n then 1 else* −*1* :: *′a* :: *ring-1*)

by (*induction n*) (*simp-all split*: *if-splits*)

lemma *liouville-lambda-power-even*:
n > *0* ⟹ *even m* ⟹ *liouville-lambda* (*n* ^ *m*) = *1*
by (*subst liouville-lambda.power*) (*auto elim*!: *evenE simp*: *liouville-lambda-def power-neg-one-If*)

lemma *liouville-lambda-power-odd*:
odd m ⟹ *liouville-lambda* (*n* ^ *m*) = *liouville-lambda n*
by (*subst liouville-lambda.power*) (*auto elim*!: *oddE simp*: *liouville-lambda-def power-neg-one-If*)

lemma *liouville-lambda-power*:
liouville-lambda (*n* ^ *m*) =
(*if n* = *0* ∧ *m* > *0 then 0 else if even m then 1 else liouville-lambda n*)
by (*auto simp*: *liouville-lambda-power-even liouville-lambda-power-odd power-0-left*)

interpretation *squarefree*: *multiplicative-function′*
ind squarefree λ*p k*. *if k* > *1 then 0 else 1* λ-. *1*
proof
fix *p k* :: *nat* **assume** *prime p k* > *0*
thus *ind squarefree* (*p* ^ *k*) = (*if 1* < *k then 0 else 1* :: ′*a*)
by (*cases k* = *1*) (*auto simp*: *squarefree-power-iff squarefree-prime ind-def*)
qed (*auto simp*: *squarefree-mult-coprime squarefree-power-iff ind-def dest*: *squarefree-multD*
simp del: *One-nat-def*)

interpretation *is-nth-power*: *multiplicative-function ind* (*is-nth-power n*)
by *standard* (*auto simp*: *is-nth-power-mult-coprime-nat-iff*)

interpretation *is-nth-power*: *multiplicative-function′*
ind (*is-nth-power n*) λ*p k*. *if n dvd k then 1 else 0* λ-. *if n* = *1 then 1 else 0*
by *standard* (*simp-all add*: *is-nth-power-prime-power-nat-iff ind-def*)

interpretation *is-square*: *multiplicative-function ind is-square*
by *standard* (*auto simp*: *is-nth-power-mult-coprime-nat-iff*)

interpretation *is-square*: *multiplicative-function′*
ind is-square λ*p k*. *if even k then 1 else 0* λ-. *0*
by *standard* (*simp-all add*: *is-nth-power-prime-power-nat-iff ind-def*)

lemma *liouville-lambda-divisors-sum*:
(∑ *d* | *d dvd n*. *liouville-lambda d*) = *ind is-square n*
proof (*rule multiplicative-function-eqI*)
show *multiplicative-function* (λ*n*. (∑ *d* | *d dvd n*. *liouville-lambda d*))
by (*rule liouville-lambda.multiplicative-sum-divisors*)
show *multiplicative-function* (*ind is-square*)

by (*rule is-nth-power.multiplicative-function-axioms*)
next
fix *p k* :: *nat* **assume** *pk*: *prime p k* > *0*
hence *p-gt-1*: *p* > *1* **by** (*simp add*: *prime-gt-Suc-0-nat*)
have ($\sum$ *d* | *d dvd p* ^ *k*. *liouville-lambda d*) = ($\sum$ *d*∈(λ*i*. *p* ^ *i*) ' {..*k*}. *liouville-lambda d*)
using *pk* **by** (*intro sum.cong refl*) (*auto intro*: *le-imp-power-dvd simp*: *divides-primepow-nat*)
also from *pk* **and** *p-gt-1* **have** ... = ($\sum$ *i*≤*k*. *liouville-lambda* (*p* ^ *i*))
by (*subst sum.reindex*) (*auto simp*: *inj-on-def prime-gt-1-nat*)
also from *pk* **have** ... = ($\sum$ *i*≤*k*. (−*1*) ^ *i*) **by** (*intro sum.cong refl*) *simp*
also have ... = (*if even k then 1 else 0*) **by** (*induction k*) *auto*
also from *pk* **have** ... = *ind is-square* (*p* ^ *k*) **by** (*simp add*: *is-square.prime-power*)
finally show ($\sum$ *d* | *d dvd p* ^ *k*. *liouville-lambda d*) = *ind is-square* (*p* ^ *k*) **.**
qed

lemma *fds-liouville-lambda-times-zeta*: *fds liouville-lambda* ∗ *fds-zeta* = *fds-ind is-square*
by (*rule fds-eqI*) (*simp add*: *liouville-lambda-divisors-sum fds-nth-mult dirichlet-prod-def*)

lemma *fds-liouville-lambda*: *fds liouville-lambda* = *fds-ind is-square* ∗ *fds moebius-mu*
proof −
have *fds liouville-lambda* ∗ *fds-zeta* ∗ *fds moebius-mu* = *fds-ind is-square* ∗ *fds moebius-mu*
by (*simp add*: *fds-liouville-lambda-times-zeta*)
also have *fds liouville-lambda* ∗ *fds-zeta* ∗ *fds moebius-mu* = *fds liouville-lambda*
by (*simp only*: *mult.assoc fds-zeta-times-moebius-mu mult-1-right*)
finally show *?thesis* **.**
qed

lemma *liouville-lambda-altdef*:
liouville-lambda n = ($\sum$ *d* | *d* ^ *2 dvd n*. *moebius-mu* (*n div d* ^ *2*))
proof (*cases n* = *0*)
case *False*
have *liouville-lambda n* = *fds-nth* (*fds liouville-lambda*) *n* **by** (*simp add*: *fds-nth-fds*)
also have *fds liouville-lambda* = *fds-ind is-square* ∗ (*fds moebius-mu* :: ′*a fds*)
by (*rule fds-liouville-lambda*)
also have *fds-nth* ... *n* = ($\sum$ *d* | *d dvd n*. *ind is-square d* ∗ *moebius-mu* (*n div d*))
by (*simp add*: *fds-nth-mult dirichlet-prod-def*)
also have ... = ($\sum$ *d* ∈ (λ*d*. *d*^*2*) ' {*d*. *d* ^ *2 dvd n*}. *moebius-mu* (*n div d*))
using *False*
by (*intro sum.mono-neutral-cong-right*) (*auto simp*: *ind-def is-nth-power-def*)
also have ... = ($\sum$ *d* | *d* ^ *2 dvd n*. *moebius-mu* (*n div d* ^ *2*))
by (*subst sum.reindex*) (*auto simp*: *inj-on-def dest*: *power2-eq-imp-eq*)
finally show *?thesis* **.**
qed *auto*

lemma *abs-moebius-mu*: *abs* (*moebius-mu n* :: *'a* :: *linordered-idom*) = *ind square-free n*
  by (*auto simp*: *ind-def moebius-mu-def*)

end

8 The divisor functions

theory *Divisor-Count*
imports
  Complex-Main
  HOL−Number-Theory.Number-Theory
  Dirichlet-Series
  More-Totient
  Moebius-Mu
begin

8.1 The general divisor function

definition *divisor-sigma* :: *'a* :: *nat-power* ⇒ *nat* ⇒ *'a* **where**
  divisor-sigma x n = ($\sum$ *d* | *d dvd n*. *nat-power d x*)

lemma *divisor-sigma-0* [*simp*]: *divisor-sigma x 0* = *0*
  by (*simp add*: *divisor-sigma-def*)

lemma *divisor-sigma-Suc-0* [*simp*]: *divisor-sigma x* (*Suc 0*) = *1*
  by (*simp add*: *divisor-sigma-def*)

lemma *divisor-sigma-1* [*simp*]: *divisor-sigma x 1* = *1*
  by *simp*

lemma *fds-divisor-sigma*: *fds* (*divisor-sigma x*) = *fds-zeta* ∗ *fds-shift x fds-zeta*
  by (*rule fds-eqI*) (*simp add*: *fds-nth-mult dirichlet-prod-altdef1 divisor-sigma-def*)

interpretation *divisor-sigma*: *multiplicative-function divisor-sigma x*
proof −
  have *multiplicative-function* (*dirichlet-prod* (λ*n*. *if n* = *0 then 0 else 1*)
      (λ*n*. *if n* = *0 then 0 else nat-power n x*)) (**is** *multiplicative-function ?f*)
    by (*rule multiplicative-dirichlet-prod*; *standard*)
      (*simp-all add*: *nat-power-mult-distrib*)
  also have *?f n* = *divisor-sigma x n* **for** *n*
    using *fds-divisor-sigma*[*of x*]
    by (*cases n* = *0*) (*simp-all add*: *fds-eq-iff fds-nth-mult*)
  hence *?f* = *divisor-sigma x* **..**
  finally show *multiplicative-function* (*divisor-sigma x*) **.**
qed

lemma *divisor-sigma-naive* [*code*]:

divisor-sigma x n = (*if n* = *0 then 0 else fold-atLeastAtMost-nat*
(λ*d acc*. *if d dvd n then nat-power d x* + *acc else acc*) *1 n 0*)
proof (*cases n* = *0*)
case *False*
have *divisor-sigma x n* = ($\sum$ *d*∈{*1*..*n*}. *if d dvd n then nat-power d x else 0*)
unfolding *divisor-sigma-def* **using** *False* **by** (*intro sum.mono-neutral-cong-left*)
(*auto elim*: *dvdE*)
also have ... = *fold-atLeastAtMost-nat*
(λ*d acc*. (*if d dvd n then nat-power d x else 0*) + *acc*) *1 n 0*
by (*rule sum-atLeastAtMost-code*)
also have (λ*d acc*. (*if d dvd n then nat-power d x else 0*) + *acc*) =
(λ*d acc*. (*if d dvd n then nat-power d x* + *acc else acc*))
by (*auto simp*: *fun-eq-iff*)
finally show *?thesis* **using** *False* **by** *simp*
qed *auto*

lemma *divisor-sigma-of-nat*: *divisor-sigma* (*of-nat x*) *n* = *of-nat* (*divisor-sigma x n*)
proof (*cases n* = *0*)
case *False*
show *?thesis* **unfolding** *divisor-sigma-def of-nat-sum*
by (*intro sum.cong refl*, *subst nat-power-of-nat*) (*insert False*, *auto elim*: *dvdE*)
qed *auto*

lemma *divisor-sigma-prime-power-field*:
fixes *x* :: *'a* :: {*field*, *nat-power*}
assumes *prime p*
shows *divisor-sigma x* (*p* ^ *k*) =
(*if nat-power p x* = *1 then of-nat* (*k* + *1*) *else*
(*nat-power p x* ^ *Suc k* − *1*) / (*nat-power p x* − *1*))
proof −
have *divisor-sigma x* (*p* ^ *k*) = ($\sum$ *i*≤*k*. *nat-power* (*p*^*i*) *x*)
unfolding *divisor-sigma-def*
by (*rule sum.reindex-bij-betw* [*symmetric*])
(*insert assms*, *auto simp*: *bij-betw-def inj-on-def prime-gt-Suc-0-nat*
divides-primepow-nat intro: *le-imp-power-dvd*)
also have ... = ($\sum$ *i*≤*k*. *nat-power p x* ^ *i*)
using *assms* **by** (*intro sum.cong refl*) (*simp-all add*: *prime-gt-0-nat nat-power-power-left*)
also have ... = (*if nat-power p x* = *1 then of-nat* (*k* + *1*) *else*
(*nat-power p x* ^ *Suc k* − *1*) / (*nat-power p x* − *1*))
using *geometric-sum*[*of nat-power p x Suc k*] **unfolding** *lessThan-Suc-atMost*
by (*auto split*: *if-splits*)
finally show *?thesis* **.**
qed

lemma *divisor-sigma-prime-power-nat*:
assumes *prime p*
shows *divisor-sigma x* (*p* ^ *k*) = (*if x* = *0 then Suc k else*
(*p* ^ (*x* ∗ *Suc k*) − *1*) *div* (*p* ^ *x* − *1*))

proof (*cases x = 0*)
  case *True*
  with *assms* **have** *nat-power p* (*real x*) = *1* **by** *simp*
  hence *divisor-sigma* (*real x*) (*p* ^ *k*) = *real* (*Suc k*)
    by (*subst divisor-sigma-prime-power-field*) (*simp-all del*: *nat-power-real-def add*: *assms*)
  thus *?thesis* **unfolding** *divisor-sigma-of-nat* **by** (*subst* (*asm*) *of-nat-eq-iff*) (*insert True*, *simp*)
next
  case *False*
  with *assms* **have** *gt-1*: *p* ^ *x* > *1*
    using *power-gt1*[*of p x* − *1*] **by** (*simp add*: *prime-gt-Suc-0-nat*)
  hence *not-one*: *real p* ^ *x* ≠ *1*
    unfolding *of-nat-power* [*symmetric*] *of-nat-eq-1-iff* **by** (*intro notI*) *simp*
  from *gt-1* **have** *dvd*: *p* ^ *x* − *1 dvd p* ^ (*x* ∗ *Suc k*) − *1*
    using *geometric-sum-nat-dvd*[*of p* ^ *x Suc k*] *assms*
    by (*simp add*: *power-mult prime-gt-Suc-0-nat power-add*)
  have *divisor-sigma* (*real x*) (*p* ^ *k*) =
        real (*if x* = *0 then Suc k else* (*p* ^ (*x* ∗ *Suc k*) − *1*) *div* (*p* ^ *x* − *1*))
    by (*subst divisor-sigma-prime-power-field* [*OF assms*, **where** *'a* = *real*])
      (*insert assms False dvd not-one*, *auto simp del*: *power-Suc nat-power-real-def*
      simp: *prime-gt-0-nat real-of-nat-div of-nat-diff prime-ge-Suc-0-nat power-mult* [*symmetric*])
  thus *?thesis* **unfolding** *divisor-sigma-of-nat* **by** (*subst* (*asm*) *of-nat-eq-iff*)
qed

interpretation *divisor-sigma-field*:
  multiplicative-function′ divisor-sigma (*x* :: *′a* :: {*field*, *nat-power*})
    λp k. *if nat-power p x* = *1 then of-nat* (*Suc k*) *else*
        (*nat-power p x* ^ *Suc k* − *1*) / (*nat-power p x* − *1*)
    λp. *nat-power p x* + *1*
  by *standard* (*auto simp*: *divisor-sigma-prime-power-field prime-gt-0-nat field-simps*)

interpretation *divisor-sigma-real*:
  multiplicative-function′ divisor-sigma (*x* :: *real*)
    λp k. *if x* = *0 then of-nat* (*Suc k*) *else* ((*real p powr x*) ^ *Suc k* − *1*) / (*real p powr x* − *1*)
    λp. *real p powr x* + *1*
proof (*standard*, *goal-cases*)
  case (*1 p k*)
  thus *?case*
    by (*auto simp*: *divisor-sigma-prime-power-field prime-gt-0-nat powr-def of-nat-eq-1-iff*
          exp-of-nat-mult [*symmetric*] *mult-ac simp del*: *of-nat-Suc power-Suc*)
next
  case (*2 p*)
  hence *real p powr x* ≠ *1* **if** *x* ≠ *0* **by** (*auto simp*: *powr-def that prime-gt-0-nat of-nat-eq-1-iff*)
  with *2* **show** *?case* **by** (*auto simp*: *field-simps*)
qed

interpretation *divisor-sigma-nat*:
multiplicative-function′ divisor-sigma (*x* :: *nat*)
λ*p k*. *if x* = *0 then Suc k else* (*p* ^ (*Suc k* ∗ *x*) − *1*) *div* (*p* ^ *x* − *1*)
λ*p*. *p* ^ *x* + *1*
proof (*standard*, *goal-cases*)
case (*2 p*)
have (*p* ^ (*x* + *x*) − *1*) = (*p* ^ *x* + *1*) ∗ (*p* ^ *x* − *1*)
by (*simp add*: *algebra-simps power-add*)
moreover have *p* ^ *x* > *1* **if** *x* > *0* **using** *that 2 one-less-power prime-gt-1-nat* **by** *blast*
ultimately show *?case* **using** *prime-ge-Suc-0-nat*[*of p*] **by** *auto*
qed (*auto simp*: *divisor-sigma-prime-power-nat mult-ac*)

lemma *divisor-sigma-prime*:
assumes *prime p*
shows *divisor-sigma x p* = *nat-power p x* + *1*
proof −
have *divisor-sigma x p* = (∑ *d* | *d dvd p*. *nat-power d x*)
by (*simp add*: *divisor-sigma-def*)
also from *assms* **have** {*d*. *d dvd p*} = {*1*, *p*} **by** (*auto simp*: *prime-nat-iff*)
also have (∑ *d*∈. . . . *nat-power d x*) = *nat-power p x* + *1*
using *assms* **by** (*subst sum.insert*) (*auto simp*: *add-ac*)
finally show *?thesis* **.**
qed

8.2 The divisor-counting function

definition *divisor-count* :: *nat* ⇒ *nat* **where**
divisor-count n = *card* {*d*. *d dvd n*}

lemma *divisor-count-0* [*simp*]: *divisor-count 0* = *0*
by (*simp add*: *divisor-count-def*)

lemma *divisor-count-Suc-0* [*simp*]: *divisor-count* (*Suc 0*) = *1*
by (*simp add*: *divisor-count-def*)

lemma *divisor-sigma-0-left-nat*: *divisor-sigma 0 n* = *divisor-count n*
by (*simp add*: *divisor-sigma-def divisor-count-def*)

lemma *divisor-sigma-0-left*: *divisor-sigma 0 n* = *of-nat* (*divisor-count n*)
unfolding *divisor-sigma-0-left-nat* [*symmetric*] *divisor-sigma-of-nat* [*symmetric*] **by** *simp*

lemma *divisor-count-altdef*: *divisor-count n* = *divisor-sigma 0 n*
by (*simp add*: *divisor-sigma-0-left*)

lemma *divisor-count-naive* [*code*]:
divisor-count n = (*if n* = *0 then 0 else*

fold-atLeastAtMost-nat (*λd acc. if d dvd n then Suc acc else acc*) *1 n 0*)
using *divisor-sigma-naive*[*of 0* :: *nat n*]
by (*simp split*: *if-splits add*: *divisor-count-altdef cong*: *if-cong*)

interpretation *divisor-count*: *multiplicative-function′ divisor-count λp k. Suc k λ-. 2*
by *standard* (*simp-all add*: *divisor-count-altdef divisor-sigma.mult-coprime divisor-sigma-nat.prime-power*)

lemma *divisor-count-dvd-mono*:
assumes *a dvd b b ≠ 0*
shows *divisor-count a ≤ divisor-count b*
using *assms* **by** (*auto simp*: *divisor-count-def intro*!: *card-mono intro*: *dvd-trans*)

8.3 The divisor sum function

definition *divisor-sum* :: *nat ⇒ nat* **where**
divisor-sum n = $\sum$ {*d. d dvd n*}

lemma *divisor-sum-0* [*simp*]: *divisor-sum 0 = 0*
by (*simp add*: *divisor-sum-def*)

lemma *divisor-sum-Suc-0* [*simp*]: *divisor-sum* (*Suc 0*) = *Suc 0*
by (*simp add*: *divisor-sum-def*)

lemma *divisor-sigma-1-left-nat*: *divisor-sigma* (*Suc 0*) *n* = *divisor-sum n*
by (*simp add*: *divisor-sum-def divisor-sigma-def*)

lemma *divisor-sigma-1-left*: *divisor-sigma 1 n* = *of-nat* (*divisor-sum n*)
by (*simp add*: *divisor-sum-def divisor-sigma-def*)

lemma *divisor-sum-altdef*: *divisor-sum n* = *divisor-sigma 1 n*
by (*simp add*: *divisor-sigma-1-left-nat*)

interpretation *divisor-sum*:
multiplicative-function′ divisor-sum λp k. (*p ^ Suc k − 1*) *div* (*p − 1*) *λp. Suc p*
proof (*standard*, *goal-cases*)
case (*5 p*)
thus *?case* **using** *divisor-sigma-nat.prime-aux*[*of p 1*]
by (*simp-all add*: *divisor-sum-altdef*)
qed (*simp-all add*: *divisor-sum-altdef divisor-sigma-nat.prime-power divisor-sigma.mult-coprime*)

lemma *divisor-sum-dvd-mono*:
assumes *a dvd b b ≠ 0*
shows *divisor-sum a ≤ divisor-sum b*
using *assms*
by (*cases a = 0*) (*auto simp*: *divisor-sum-def intro*!: *sum-le-included intro*: *dvd-trans*)

lemma *divisor-sum-naive* [*code*]:

divisor-sum n = (*if n* = *0 then 0 else*
fold-atLeastAtMost-nat (*λd acc. if d dvd n then d* + *acc else acc*) *1 n 0*)
using *divisor-sigma-naive*[*of Suc 0 n*]
by (*simp split*: *if-splits add*: *divisor-sum-altdef cong*: *if-cong*)

lemma *fds-divisor-count*: *fds divisor-count* = *fds-zeta* ^ *2*
by (*rule fds-eqI*)
(*simp add*: *fds-nth-mult dirichlet-prod-altdef1 divisor-count-def power2-eq-square*)

lemma *fds-shift-zeta-1*: *fds-shift 1 fds-zeta* = *fds of-nat*
by (*rule fds-eqI*) (*simp add*: *fds-nth-mult*)

lemma *fds-shift-zeta-Suc-0*: *fds-shift* (*Suc 0*) *fds-zeta* = *fds id*
by (*rule fds-eqI*) (*simp add*: *fds-nth-mult*)

lemma *fds-divisor-sum*: *fds divisor-sum* = *fds-zeta* ∗ *fds id*
by (*rule fds-eqI*) (*simp add*: *fds-nth-mult dirichlet-prod-altdef1 divisor-sum-def*)

lemma *fds-divisor-sum-eq-totient-times-d*: *fds divisor-sum* = *fds totient* ∗ *fds divisor-count*
proof −
have *fds divisor-sum* = *fds-zeta* ∗ *fds id* **by** (*fact fds-divisor-sum*)
also have *fds id* = *fds totient* ∗ *fds-zeta* **by** (*rule fds-totient-times-zeta′* [*symmetric*])
also have *fds-zeta* ∗ ... = *fds totient* ∗ *fds divisor-count*
using *fds-divisor-count* **by** (*simp add*: *power2-eq-square mult-ac*)
finally show *?thesis* **.**
qed

lemma *fds-divisor-sum-times-moebius-mu*:
fds (*divisor-sigma* (*1* :: *′a* :: {*nat-power*,*comm-ring-1*})) ∗ *fds moebius-mu* = *fds of-nat*
proof −
have *fds* (*divisor-sigma 1*) ∗ *fds moebius-mu* =
fds of-nat ∗ (*fds-zeta* ∗ *fds moebius-mu* :: *′a fds*)
by (*subst mult.assoc* [*symmetric*], *subst fds-zeta-commutes* [*symmetric*])
(*simp add*: *fds-divisor-sigma fds-shift-zeta-1*)
also have *fds-zeta* ∗ *fds moebius-mu* = (*1* :: *′a fds*) **by** (*fact fds-zeta-times-moebius-mu*)
finally show *?thesis* **by** *simp*
qed

lemma *inverse-divisor-sigma*:
fixes *a* :: *′a* :: {*field*, *nat-power*}
shows *inverse* (*fds* (*divisor-sigma a*)) = *fds-shift a* (*fds moebius-mu*) ∗ *fds moebius-mu*
proof −
have *fds* (*divisor-sigma a*) = *fds-zeta* ∗ *fds-shift a fds-zeta*

by (*simp add*: *fds-divisor-sigma*)
also have *inverse* ... = *fds moebius-mu* ∗ *inverse* (*fds-shift a fds-zeta*)
by (*simp add*: *fds-moebius-inverse-zeta inverse-mult-fds*)
also have *inverse* (*fds-shift a fds-zeta*) =
fds (λ*n*. *moebius-mu n* ∗ *fds-nth* (*fds-shift a fds-zeta*) *n*)
by (*intro completely-multiplicative-fds-inverse'*, *unfold-locales*)
(*auto simp*: *nat-power-mult-distrib*)
also have ... = *fds-shift a* (*fds moebius-mu*)
by (*auto simp*: *fds-eq-iff*)
finally show *?thesis* **by** (*simp add*: *mult.commute*)
qed

end

9 Summatory arithmetic functions

theory *Arithmetic-Summatory*
imports
More-Totient
Moebius-Mu
Liouville-Lambda
Divisor-Count
Dirichlet-Series
begin

9.1 Definition

definition *sum-upto* :: (*nat* ⇒ *'a* :: *comm-monoid-add*) ⇒ *real* ⇒ *'a* **where**
sum-upto f x = ($\sum$ *i* | *0* < *i* ∧ *real i* ≤ *x*. *f i*)

lemma *sum-upto-altdef*: *sum-upto f x* = ($\sum$ *i*∈{*0*<..*nat* ⌊*x*⌋}. *f i*)
unfolding *sum-upto-def*
by (*cases x* ≥ *0*; *intro sum.cong refl*) (*auto simp*: *le-nat-iff le-floor-iff*)

lemma *sum-upto-0* [*simp*]: *sum-upto f 0* = *0*
by (*simp add*: *sum-upto-altdef*)

lemma *sum-upto-cong* [*cong*]:
(⋀*n*. *n* > *0* ⟹ *f n* = *f' n*) ⟹ *n* = *n'* ⟹ *sum-upto f n* = *sum-upto f' n'*
by (*simp add*: *sum-upto-def*)

lemma *finite-Nats-le-real* [*simp,intro*]: *finite* {*n*. *0* < *n* ∧ *real n* ≤ *x*}
proof (*rule finite-subset*)
show *finite* {*n*. *n* ≤ *nat* ⌊*x*⌋} **by** *auto*
show {*n*. *0* < *n* ∧ *real n* ≤ *x*} ⊆ {*n*. *n* ≤ *nat* ⌊*x*⌋} **by** *safe linarith*
qed

lemma *sum-upto-ind*: *sum-upto* (*ind P*) *x* = *of-nat* (*card* {*n*. *n* > *0* ∧ *real n* ≤ *x*
∧ *P n*})

proof −
have *sum-upto* (*ind P* :: *nat* ⇒ *'a*) *x* = ($\sum$*n* | *0* < *n* ∧ *real n* ≤ *x* ∧ *P n*. *1*)
unfolding *sum-upto-def* **by** (*intro sum.mono-neutral-cong-right*) (*auto simp*: *ind-def*)
also have ... = *of-nat* (*card* {*n*. *n* > *0* ∧ *real n* ≤ *x* ∧ *P n*}) **by** *simp*
finally show *?thesis* **.**
qed

lemma *sum-upto-sum-divisors*:
sum-upto (λ*n*. $\sum$*d* | *d dvd n*. *f n d*) *x* = *sum-upto* (λ*k*. *sum-upto* (λ*d*. *f* (*d* ∗ *k*) *k*) (*x* / *k*)) *x*
proof −
let *?B* = (*SIGMA k*:{*k*. *0* < *k* ∧ *real k* ≤ *x*}. {*d*. *0* < *d* ∧ *real d* ≤ *x* / *real k*})
let *?A* = (*SIGMA k*:{*k*. *0* < *k* ∧ *real k* ≤ *x*}. {*d*. *d dvd k*})
have ∗: *real a* ≤ *x* **if** *real* (*a* ∗ *b*) ≤ *x b* > *0* **for** *a b*
proof −
have *real a* ∗ *1* ≤ *real* (*a* ∗ *b*) **unfolding** *of-nat-mult* **using** *that*
by (*intro mult-left-mono*) *auto*
also have ... ≤ *x* **by** *fact*
finally show *?thesis* **by** *simp*
qed
have *bij*: *bij-betw* (λ(*k*,*d*). (*d* ∗ *k*, *k*)) *?B ?A*
by (*rule bij-betwI*[**where** *g* = λ(*k*,*d*). (*d*, *k div d*)])
(*auto simp*: ∗ *divide-simps mult.commute elim*!: *dvdE*)

have *sum-upto* (λ*n*. $\sum$*d* | *d dvd n*. *f n d*) *x* = ($\sum$(*k*,*d*)∈*?A*. *f k d*)
unfolding *sum-upto-def* **by** (*rule sum.Sigma*) *auto*
also have ... = ($\sum$(*k*,*d*)∈*?B*. *f* (*d* ∗ *k*) *k*)
by (*subst sum.reindex-bij-betw*[*OF bij*, *symmetric*]) (*auto simp*: *case-prod-unfold*)
also have ... = *sum-upto* (λ*k*. *sum-upto* (λ*d*. *f* (*d* ∗ *k*) *k*) (*x* / *k*)) *x*
unfolding *sum-upto-def* **by** (*rule sum.Sigma* [*symmetric*]) *auto*
finally show *?thesis* **.**
qed

lemma *sum-upto-dirichlet-prod*:
sum-upto (*dirichlet-prod f g*) *x* = *sum-upto* (λ*d*. *f d* ∗ *sum-upto g* (*x* / *real d*)) *x*
unfolding *dirichlet-prod-def*
by (*subst sum-upto-sum-divisors*) (*simp add*: *sum-upto-def sum-distrib-left*)

lemma *sum-upto-real*:
assumes *x* ≥ *0*
shows *sum-upto real x* = *of-int* (*floor x*) ∗ (*of-int* (*floor x*) + *1*) / *2*
proof −
have *A*: *2* ∗ $\sum${*1*..*n*} = *n* ∗ *Suc n* **for** *n* **by** (*induction n*) *simp-all*
have *2* ∗ *sum-upto real x* = *real* (*2* ∗ $\sum${*0*<..*nat* ⌊*x*⌋}) **by** (*simp add*: *sum-upto-altdef*)
also have {*0*<..*nat* ⌊*x*⌋} = {*1*..*nat* ⌊*x*⌋} **by** *auto*
also note *A*
also have *real* (*nat* ⌊*x*⌋ ∗ *Suc* (*nat* ⌊*x*⌋)) = *of-int* (*floor x*) ∗ (*of-int* (*floor x*) + *1*) **using** *assms*

```
    by (simp add: algebra-simps)
  finally show ?thesis by simp
qed

lemma summable-imp-convergent-sum-upto:
  assumes summable (f :: nat ⇒ 'a :: real-normed-vector)
  obtains c where (sum-upto f ⟶ c) at-top
proof −
  from assms have summable (λn. f (Suc n))
    by (subst summable-Suc-iff)
  then obtain c where (λn. f (Suc n)) sums c by (auto simp: summable-def)
  hence (λn. ∑k<n. f (Suc k)) ⟶ c by (auto simp: sums-def)
  also have (λn. ∑k<n. f (Suc k)) = (λn. ∑k∈{0<..n}. f k)
   by (subst sum.atLeast1-atMost-eq [symmetric]) (auto simp: atLeastSucAtMost-greaterThanAtMost)
  finally have ((λx. sum f {0<..nat ⌊x⌋}) ⟶ c) at-top
    by (rule filterlim-compose)
     (auto intro!: filterlim-compose[OF filterlim-nat-sequentially] filterlim-floor-sequentially)
  also have (λx. sum f {0<..nat ⌊x⌋}) = sum-upto f
    by (intro ext) (simp-all add: sum-upto-altdef)
  finally show ?thesis using that[of c] by blast
qed
```

9.2 The Hyperbola method

```
lemma hyperbola-method-semiring:
  fixes f g :: nat ⇒ 'a :: comm-semiring-0
  assumes A ≥ 0 and B ≥ 0 and A ∗ B = x
  shows   sum-upto (dirichlet-prod f g) x + sum-upto f A ∗ sum-upto g B =
            sum-upto (λn. f n ∗ sum-upto g (x / real n)) A +
            sum-upto (λn. sum-upto f (x / real n) ∗ g n) B
proof −
  from assms have [simp]: x ≥ 0 by auto
  {
    fix a b :: real assume ab: a > 0 b > 0 x ≥ 0 a ∗ b ≤ x a > A b > B
    hence a ∗ b > A ∗ B using assms by (intro mult-strict-mono) auto
    also from assms have A ∗ B = x by simp
    finally have False using ‹a ∗ b ≤ x› by simp
  } note ∗ = this
  have ∗: a ≤ A ∨ b ≤ B if a ∗ b ≤ x a > 0 b > 0 x ≥ 0 for a b
    by (rule ccontr) (insert ∗[of a b] that, auto)

  have nat-mult-leD1: real a ≤ x if real a ∗ real b ≤ x b > 0 for a b
  proof −
    from that have real a ∗ 1 ≤ real a ∗ real b by (intro mult-left-mono) simp-all
    also have ... ≤ x by fact
    finally show ?thesis by simp
  qed
  have nat-mult-leD2: real b ≤ x if real a ∗ real b ≤ x a > 0 for a b
    using nat-mult-leD1[of b a] that by (simp add: mult-ac)
```

have *le-sqrt-mult-imp-le*: $a * b \le x$
if $a \ge 0$ $b \ge 0$ $a \le A$ $b \le B$ **for** a b :: *real*
proof −
from *that* **and** *assms* **have** $a * b \le A * B$ **by** (*intro mult-mono*) *auto*
with *assms* **show** $a * b \le x$ **by** *simp*
qed

define F G **where** F = *sum-upto f* **and** G = *sum-upto g*
let $?Bound = \{0<..nat \lfloor x \rfloor\} \times \{0<..nat \lfloor x \rfloor\}$
let $?B = \{(r,d).\ 0 < r \wedge real\ r \le A \wedge 0 < d \wedge real\ d \le x\ /\ real\ r\}$
let $?C = \{(r,d).\ 0 < d \wedge real\ d \le B \wedge 0 < r \wedge real\ r \le x\ /\ real\ d\}$
let $?B' = SIGMA\ r{:}\{r.\ 0 < r \wedge real\ r \le A\}.\ \{d.\ 0 < d \wedge real\ d \le x\ /\ real\ r\}$
let $?C' = SIGMA\ d{:}\{d.\ 0 < d \wedge real\ d \le B\}.\ \{r.\ 0 < r \wedge real\ r \le x\ /\ real\ d\}$
have *sum-upto* (*dirichlet-prod f g*) $x + F\ A * G\ B =$
$(\sum (i,(r,d)) \in (SIGMA\ i{:}\{i.\ 0 < i \wedge real\ i \le x\}.\ \{(r,d).\ r * d = i\}).\ f\ r * g\ d) +$
sum-upto f A $*$ *sum-upto g B* (**is** - = $?S$ + -)
unfolding *sum-upto-def dirichlet-prod-altdef2 F-def G-def*
by (*subst sum.Sigma*) (*auto intro*: *finite-divisors-nat′*)
also have $?S = (\sum (r,d) \mid 0 < r \wedge 0 < d \wedge real\ (r * d) \le x.\ f\ r * g\ d)$
(**is** - = *sum* - $?A$) **by** (*intro sum.reindex-bij-witness*[*of* - $\lambda(r,d).\ (r{*}d,(r,d))$ *snd*]) *auto*
also have $?A = ?B \cup ?C$ **by** (*auto simp*: *field-simps dest*: $*$)
also have *sum-upto f A* $*$ *sum-upto g B* =
$(\sum r \mid 0 < r \wedge real\ r \le A.\ \sum d \mid 0 < d \wedge real\ d \le B.\ f\ r * g\ d)$
by (*simp add*: *sum-upto-def sum-product*)
also have $\ldots = (\sum (r,d){\in}\{r.\ 0 < r \wedge real\ r \le A\} \times \{d.\ 0 < d \wedge real\ d \le B\}.\ f\ r * g\ d)$
(**is** - = *sum* - $?X$) **by** (*rule sum.cartesian-product*)
also have $?X = ?B \cap ?C$ **by** (*auto simp*: *field-simps le-sqrt-mult-imp-le*)
also have $(\sum (r,d){\in}?B \cup ?C.\ f\ r * g\ d) + (\sum (r,d){\in}?B \cap ?C.\ f\ r * g\ d) =$
$(\sum (r,d){\in}?B.\ f\ r * g\ d) + (\sum (r,d){\in}?C.\ f\ r * g\ d)$
by (*intro sum.union-inter finite-subset*[*of ?B ?Bound*] *finite-subset*[*of ?C ?Bound*])
(*auto simp*: *field-simps le-nat-iff le-floor-iff dest*: *nat-mult-leD1 nat-mult-leD2*)
also have $?B = ?B'$ **by** *auto*
hence $(\lambda f.\ sum\ f\ ?B) = (\lambda f.\ sum\ f\ ?B')$ **by** *simp*
also have $(\sum (r,d){\in}?B'.\ f\ r * g\ d) = sum\text{-}upto\ (\lambda n.\ f\ n * G\ (x\ /\ real\ n))\ A$
by (*subst sum.Sigma* [*symmetric*]) (*simp-all add*: *sum-upto-def sum-distrib-left G-def*)
also have $(\sum (r,d){\in}?C.\ f\ r * g\ d) = (\sum (d,r){\in}?C'.\ f\ r * g\ d)$
by (*intro sum.reindex-bij-witness*[*of* - $\lambda(x,y).\ (y,x)$ $\lambda(x,y).\ (y,x)$]) *auto*
also have $\ldots = sum\text{-}upto\ (\lambda n.\ F\ (x\ /\ real\ n) * g\ n)\ B$
by (*subst sum.Sigma* [*symmetric*]) (*simp-all add*: *sum-upto-def sum-distrib-right F-def*)
finally show *?thesis* **by** (*simp only*: *F-def G-def*)
qed

lemma *hyperbola-method-semiring-sqrt*:

fixes *f g* :: *nat* ⇒ *'a* :: *comm-semiring-0*
assumes $x \geq 0$
shows *sum-upto* (*dirichlet-prod f g*) *x* + *sum-upto f* (*sqrt x*) ∗ *sum-upto g* (*sqrt x*) =
sum-upto (λ*n*. *f n* ∗ *sum-upto g* (*x* / *real n*)) (*sqrt x*) +
sum-upto (λ*n*. *sum-upto f* (*x* / *real n*) ∗ *g n*) (*sqrt x*)
using *assms hyperbola-method-semiring*[*of sqrt x sqrt x x*] **by** *simp*

lemma *hyperbola-method*:
fixes *f g* :: *nat* ⇒ *'a* :: *comm-ring*
assumes $A \geq 0$ $B \geq 0$ $A * B = x$
shows *sum-upto* (*dirichlet-prod f g*) *x* =
sum-upto (λ*n*. *f n* ∗ *sum-upto g* (*x* / *real n*)) *A* +
sum-upto (λ*n*. *sum-upto f* (*x* / *real n*) ∗ *g n*) *B* −
sum-upto f A ∗ *sum-upto g B*
using *hyperbola-method-semiring*[*OF assms*, *of f g*] **by** (*simp add*: *algebra-simps*)

lemma *hyperbola-method-sqrt*:
fixes *f g* :: *nat* ⇒ *'a* :: *comm-ring*
assumes $x \geq 0$
shows *sum-upto* (*dirichlet-prod f g*) *x* =
sum-upto (λ*n*. *f n* ∗ *sum-upto g* (*x* / *real n*)) (*sqrt x*) +
sum-upto (λ*n*. *sum-upto f* (*x* / *real n*) ∗ *g n*) (*sqrt x*) −
sum-upto f (*sqrt x*) ∗ *sum-upto g* (*sqrt x*)
using *assms hyperbola-method*[*of sqrt x sqrt x x*] **by** *simp*

end

10 Partial summation

theory *Partial-Summation*
imports
HOL−Analysis.Analysis
Arithmetic-Summatory
begin

lemma *finite-vimage-real-of-nat-greaterThanAtMost*: *finite* (*real* −' {*y*<..*x*})
proof (*rule finite-subset*)
show *real* −' {*y*<..*x*} ⊆ {*nat* ⌊*y*⌋..*nat* ⌈*x*⌉}
by (*cases* $x \geq 0$; *cases* $y \geq 0$)
(*auto simp*: *nat-le-iff le-nat-iff le-ceiling-iff floor-le-iff*)
qed *auto*

context
fixes *a* :: *nat* ⇒ *'a* :: {*banach*, *real-normed-algebra*}
fixes *f f'* :: *real* ⇒ *'a*
fixes *A*
fixes *X* :: *real set*
fixes *x y* :: *real*

```
  defines A ≡ sum-upto a
  assumes fin: finite X
  assumes xy: 0 ≤ y y < x
  assumes deriv: ⋀z. z ∈ {y..x} − X ⟹ (f has-vector-derivative f′ z) (at z)
  assumes cont-f: continuous-on {y..x} f
begin

lemma partial-summation-strong:
  ((λt. A t ∗ f′ t) has-integral
     (A x ∗ f x − A y ∗ f y − (∑n ∈ real −‘ {y<..x}. a n ∗ f n))) {y..x}
proof −
  define chi :: nat ⇒ real ⇒ real where chi = (λn t. if n ≤ t then 1 else 0)
  have ((λt. sum-upto (λn. a n ∗ (chi n t ∗R f′ t)) x) has-integral
        (sum-upto (λn. a n ∗ (f x − f (max n y))) x)) {y..x} (is (- has-integral
?I) -)
    unfolding sum-upto-def
  proof (intro has-integral-sum ballI finite-Nats-le-real, goal-cases)
    case (1 n)
    have (f′ has-integral (f x − f (max n y))) {max n y..x}
      using xy 1
      by (intro fundamental-theorem-of-calculus-strong[OF fin])
        (auto intro!: continuous-on-subset[OF cont-f] deriv)
    also have ?this ⟷ ((λt. (if t ∈ {max n y..x} then 1 else 0) ∗R f′ t)
                  has-integral (f x − f (max n y))) {max n y..x}
      by (intro has-integral-cong) (simp-all add: chi-def)
    finally have ((λt. (if t ∈ {max n y..x} then 1 else 0) ∗R f′ t)
                    has-integral (f x − f (max n y))) {y..x}
      by (rule has-integral-on-superset) auto
    also have ?this ⟷ ((λt. chi n t ∗R f′ t) has-integral (f x − f (max n y)))
{y..x}
      by (intro has-integral-cong) (auto simp: chi-def)
    finally show ?case by (intro has-integral-mult-right)
  qed
  also have ?this ⟷ ((λt. A t ∗ f′ t) has-integral ?I) {y..x}
    unfolding sum-upto-def A-def chi-def sum-distrib-right using xy
   by (intro has-integral-cong sum.mono-neutral-cong-right finite-Nats-le-real) auto
  also have sum-upto (λn. a n ∗ (f x − f (max (real n) y))) x =
             A x ∗ f x − (∑n | n > 0 ∧ real n ≤ x.  a n ∗ f (max (real n) y))
   by (simp add: sum-upto-def ring-distribs sum-subtractf sum-distrib-right A-def)
  also have {n. n > 0 ∧ real n ≤ x} = {n. n > 0 ∧ real n ≤ y} ∪ real −‘ {y<..x}
    using xy by auto
  also have sum (λn. a n ∗ f (max (real n) y)) … =
             (∑n | 0 < n ∧ real n ≤ y. a n ∗ f (max (real n) y)) +
             (∑n ∈ real −‘ {y<..x}. a n ∗ f (max (real n) y)) (is - = ?S1 + ?S2)
   by (intro sum.union-disjoint finite-Nats-le-real finite-vimage-real-of-nat-greaterThanAtMost)

      auto
  also have ?S1 = sum-upto (λn. a n ∗ f y) y unfolding sum-upto-def
    by (intro sum.cong refl) (auto simp: max-def)
```

also have ... = $A\ y * f\ y$ **by** (*simp add*: *A-def sum-upto-def sum-distrib-right*)
also have *?S2* = ($\sum n \in$ *real* −‘ $\{y<..x\}$. $a\ n * f\ n$)
by (*intro sum.cong refl*) (*auto simp*: *max-def*)
finally show *?thesis* **by** (*simp add*: *algebra-simps*)
qed

lemma *partial-summation-integrable-strong*:
($\lambda t.\ A\ t * f'\ t$) *integrable-on* $\{y..x\}$
and *partial-summation-strong'*:
($\sum n{\in}$*real* −‘ $\{y<..x\}$. $a\ n * f\ n$) =
$A\ x * f\ x - A\ y * f\ y$ − *integral* $\{y..x\}$ ($\lambda t.\ A\ t * f'\ t$)
using *partial-summation-strong* **by** (*simp-all add*: *has-integral-iff algebra-simps*)

end

context
fixes a :: *nat* ⇒ *'a* :: {*banach*, *real-normed-algebra*}
fixes $f\ f'$ :: *real* ⇒ *'a*
fixes A
fixes X :: *real set*
fixes x :: *real*
defines $A \equiv$ *sum-upto a*
assumes *fin*: *finite X*
assumes *x*: $x > 0$
assumes *deriv*: $\bigwedge z.\ z \in \{0..x\} - X \Longrightarrow$ (*f has-vector-derivative* $f'\ z$) (*at z*)
assumes *cont-f*: *continuous-on* $\{0..x\}$ *f*
begin

lemma *partial-summation-sum-upto-strong*:
(($\lambda t.\ A\ t * f'\ t$) *has-integral* ($A\ x * f\ x$ − *sum-upto* ($\lambda n.\ a\ n * f\ n$) *x*)) $\{0..x\}$
proof −
have ($\sum n{\in}$*real* −‘ $\{0<..x\}$. $a\ n * f\ n$) = *sum-upto* ($\lambda n.\ a\ n * f\ n$) *x*
unfolding *sum-upto-def* **by** (*intro sum.cong refl*) *auto*
thus *?thesis*
using *partial-summation-strong*[*OF fin order.refl x deriv cont-f*, *of a*]
by (*simp-all add*: *A-def*)
qed

lemma *partial-summation-integrable-sum-upto-strong*:
($\lambda t.\ A\ t * f'\ t$) *integrable-on* $\{0..x\}$
and *partial-summation-sum-upto-strong'*:
sum-upto ($\lambda n.\ a\ n * f\ n$) x =
$A\ x * f\ x$ − *integral* $\{0..x\}$ ($\lambda t.\ A\ t * f'\ t$)
using *partial-summation-sum-upto-strong* **by** (*simp-all add*: *has-integral-iff algebra-simps*)

end

end

11 Euler product expansions

```
theory Euler-Products
imports
  HOL−Analysis.Analysis
  Multiplicative-Function
begin
```

Conflicting notation from *HOL−Analysis.Infinite-Sum*

```
no-notation Infinite-Sum.abs-summable-on (infixr ‹abs′-summable′-on› 46)

lemma prime-factors-power-subset:
  prime-factors (x ^ n) ⊆ prime-factors x
  by (cases n = 0) (auto simp: prime-factors-power)

lemma prime-power-product-in-Pi:
  (λg. ∏p∈{p. p ≤ (n::nat) ∧ prime p}. p ^ g p)
    ∈ ({p. p ≤ n ∧ prime p} →E UNIV) →
       {m. 0 < m ∧ prime-factors m ⊆ {..n}}
proof (safe, goal-cases)
  case (2 f p)
  have prime-factors (∏p∈{p. p ≤ n ∧ prime p}. p ^ f p) =
          (⋃p∈{p. p ≤ n ∧ prime p}. prime-factors (p ^ f p))
    by (subst prime-factors-prod) auto
  also have … ⊆ (⋃p∈{p. p ≤ n ∧ prime p}. prime-factors p)
    using prime-factors-power-subset by blast
  also have … ⊆ (⋃p∈{p. p ≤ n ∧ prime p}. {p})
    by (auto simp: prime-factors-dvd prime-gt-0-nat dest!: dvd-imp-le)
  also have … ⊆ {..n} by auto
  finally show ?case using 2 by auto
qed (auto simp: prime-gt-0-nat)

lemma inj-prime-power: inj-on (λx. fst x ^ snd x :: nat) ({a. prime a} × {0<..})
proof (intro inj-onI, clarify, goal-cases)
  case (1 p m q n)
  with prime-power-eq-imp-eq[of p q m n] and 1
    have p = q by auto
  moreover from this have m = n
    using prime-gt-1-nat 1 by auto
  ultimately show ?case by simp
qed

lemma bij-betw-prime-powers:
  bij-betw (λg. ∏p∈{p. p ≤ n ∧ prime p}. p ^ g p) ({p. p ≤ n ∧ prime p} →E
UNIV)
     {m. 0 < m ∧ prime-factors m ⊆ {..(n::nat)}}
```

```
proof (rule bij-betwI[of - - - (λm p. if p ≤ n ∧ prime p then multiplicity p m else
undefined)],
        goal-cases)
  case 1
  show ?case by (rule prime-power-product-in-Pi)
next
  case 2
  show ?case
    by (auto split: if-splits)
next
  case (3 f)
  show ?case
  proof (rule ext, goal-cases)
    case (1 q)
    show ?case
    proof (cases q ≤ n ∧ prime q)
      case True
      hence multiplicity q (∏ p∈{p. p ≤ n ∧ prime p}. p ^ f p) =
             (∑ x∈{p. p ≤ n ∧ prime p}. multiplicity q (x ^ f x))
        by (subst prime-elem-multiplicity-prod-distrib) auto
      also have ... = (∑ x∈{p. p ≤ n ∧ prime p}. if x = q then f q else 0)
      using True by (intro sum.cong refl) (auto simp: multiplicity-distinct-prime-power)
      also have ... = f q using True by auto
      finally show ?thesis using True by simp
    qed (insert 3, force+)
  qed
next
  case (4 m)
  have (∏ p | p ≤ n ∧ prime p. p ^(if p ≤ n ∧ prime p then multiplicity p m else
undefined)) =
         (∏ p∈prime-factors m. p ^ multiplicity p m)
  proof (rule prod.mono-neutral-cong)
    show finite (prime-factors m) by simp
  qed (insert 4, auto simp: prime-factors-multiplicity)
  also from 4 have ... = m
    by (intro prime-factorization-nat [symmetric]) auto
  finally show ?case .
qed

lemma
  fixes f :: nat ⇒ 'a :: {real-normed-field,banach,second-countable-topology}
  assumes summable: summable (λn. norm (f n))
  assumes multiplicative-function f
  shows   abs-convergent-euler-product:
            abs-convergent-prod (λp. if prime p then ∑ n. f (p ^ n) else 1)
    and   euler-product-LIMSEQ:
            (λn. (∏ p≤n. if prime p then ∑ n. f (p ^ n) else 1)) ⟶ (∑ n. f n)
proof -
  interpret f: multiplicative-function f by fact
```

define N **where** $N = (\sum n.\ norm\ (f\ n))$

have *summable′*: *f abs-summable-on A* **for** *A*
  by (*rule abs-summable-on-subset*[*of - UNIV*])
    (*insert summable*, *auto simp*: *abs-summable-on-nat-iff′*)

have *summable″*: $(\lambda x.\ f\ (p\ \widehat{}\ x))$ *abs-summable-on A* **if** *prime p* **for** *A p*
proof (*subst abs-summable-on-reindex-iff*[*of - - f*])
  from ‹*prime p*› **have** $p > 1$
    by (*rule prime-gt-1-nat*)
  thus *inj-on* $(\lambda i.\ p\ \widehat{}\ i)$ *A*
    by (*auto simp*: *inj-on-def*)
qed (*intro summable′*)

have $(\lambda n.\ norm\ ((\sum m.\ f\ m) - (\prod p \in \{p.\ p \le n \wedge prime\ p\}.\ \sum i.\ f\ (p\ \widehat{}\ i))))$
$\longrightarrow 0$
    (**is** *filterlim ?h - -*)
proof (*rule tendsto-sandwich*)
  show *eventually* $(\lambda n.\ ?h\ n \le N - (\sum m \le n.\ norm\ (f\ m)))$ *at-top*
  proof (*intro always-eventually allI*)
    fix *n* :: *nat*
    interpret *product-sigma-finite* λ-::*nat*. *count-space* (*UNIV* :: *nat set*)
      by (*intro product-sigma-finite.intro sigma-finite-measure-count-space*)

    have $(\prod p \mid p \le n \wedge prime\ p.\ \sum i.\ f\ (p\ \widehat{}\ i)) =$
        $(\prod p \mid p \le n \wedge prime\ p.\ \sum_a i \in UNIV.\ f\ (p\ \widehat{}\ i))$
      by (*intro prod.cong refl infsetsum-nat′* [*symmetric*] *summable″*) *auto*
    also have $\ldots = (\sum_a g \in \{p.\ p \le n \wedge prime\ p\} \rightarrow_E UNIV.$
        $\prod x \in \{p.\ p \le n \wedge prime\ p\}.\ f\ (x\ \widehat{}\ g\ x))$
      by (*subst infsetsum-prod-PiE* [*symmetric*])
        (*auto simp*: *prime-gt-Suc-0-nat summable″*)
    also have $\ldots = (\sum_a g \in \{p.\ p \le n \wedge prime\ p\} \rightarrow_E UNIV.$
        $f\ (\prod x \in \{p.\ p \le n \wedge prime\ p\}.\ x\ \widehat{}\ g\ x))$
      by (*subst f.prod-coprime*) (*auto simp add*: *primes-coprime*)
    also have $\ldots = (\sum_a m \mid m > 0 \wedge prime\text{-}factors\ m \subseteq \{..n\}.\ f\ m)$
      by (*intro infsetsum-reindex-bij-betw bij-betw-prime-powers*)
     also have $(\sum_a m \in UNIV.\ f\ m) - \ldots = (\sum_a m \in UNIV - \{m.\ m > 0 \wedge$
$prime\text{-}factors\ m \subseteq \{..n\}\}.\ f\ m)$
      by (*intro infsetsum-Diff* [*symmetric*] *summable′*) *auto*
    also have $(\sum_a m \in UNIV.\ f\ m) = (\sum m.\ f\ m)$
      by (*intro infsetsum-nat′ summable′*)
    also have $UNIV - \{m.\ m > 0 \wedge prime\text{-}factors\ m \subseteq \{..n\}\} =$
        $insert\ 0\ \{m.\ \neg prime\text{-}factors\ m \subseteq \{..n\}\}$
      by *auto*
    also have $(\sum_a m \in \ldots.\ f\ m) = (\sum_a m \mid \neg prime\text{-}factors\ m \subseteq \{..n\}.\ f\ m)$
      by (*intro infsetsum-cong-neutral*) *auto*
    also have $norm\ \ldots \le (\sum_a m \mid \neg prime\text{-}factors\ m \subseteq \{..n\}.\ norm\ (f\ m))$
      by (*rule norm-infsetsum-bound*)
    also have $\ldots \le (\sum_a m \in \{n<..\}.\ norm\ (f\ m))$

proof (*intro infsetsum-mono-neutral-left summable′ abs-summable-on-normI*)
show {*m*. ¬ *prime-factors m* ⊆ {..*n*}} ⊆ {*n*<..}
proof *safe*
fix *m k* **assume** ¬*m* > *n* **and** *k* ∈ *prime-factors m*
thus *k* ≤ *n* **by** (*cases m* = *0*) (*auto simp*: *prime-factors-dvd dest*: *dvd-imp-le*)
qed
qed *auto*
also have {*n*<..} = *UNIV* − {..*n*}
by *auto*
also have ($\sum_a$*m*∈. . . . *norm* (*f m*)) = ($\sum_a$*m*∈*UNIV*. *norm* (*f m*)) − ($\sum_a$*m*∈{..*n*}. *norm* (*f m*))
using *summable* **by** (*intro infsetsum-Diff*) (*auto simp*: *abs-summable-on-nat-iff′*)
also have ($\sum_a$*m*∈*UNIV*. *norm* (*f m*)) = *N*
unfolding *N-def* **using** *summable*
by (*intro infsetsum-nat′*) (*auto simp*: *abs-summable-on-nat-iff′*)
also have ($\sum_a$*m*∈{..*n*}. *norm* (*f m*)) = ($\sum$*m*≤*n*. *norm* (*f m*))
by (*simp add*: *suminf-finite*)
finally show *?h n* ≤ *N* − ($\sum$*m*≤*n*. *norm* (*f m*)) **.**
qed
next
show *eventually* (λ*n*. *?h n* ≥ *0*) *at-top* **by** *simp*
next
show (λ*n*. *N* − ($\sum$*m*≤*n*. *norm* (*f m*))) ⟶ *0* **unfolding** *N-def*
by (*rule tendsto-eq-intros refl summable-LIMSEQ′ summable*)+ *simp-all*
qed *simp-all*
hence (λ*n*. ($\sum$*m*. *f m*) − ($\prod$*p*∈{*p*. *p* ≤ *n* ∧ *prime p*}. $\sum$*i*. *f* (*p* ^ *i*))) ⟶ *0*
by (*simp add*: *tendsto-norm-zero-iff*)
from *tendsto-diff*[*OF tendsto-const*[*of* $\sum$*m*. *f m*] *this*]
have (λ*n*. $\prod$*p* | *p* ≤ *n* ∧ *prime p*. $\sum$*i*. *f* (*p* ^ *i*)) ⟶ ($\sum$*m*. *f m*) **by** *simp*
also have (λ*n*. $\prod$*p* | *p* ≤ *n* ∧ *prime p*. $\sum$*i*. *f* (*p* ^ *i*)) =
(λ*n*. $\prod$*p*≤*n*. *if prime p then* ($\sum$*i*. *f* (*p* ^ *i*)) *else 1*)
by (*intro ext prod.mono-neutral-cong-left*) *auto*
finally show . . . ⟶ ($\sum$*m*. *f m*) **.**

show *abs-convergent-prod* (λ*p*. *if prime p then* ($\sum$*i*. *f* (*p* ^ *i*)) *else 1*)
proof (*rule summable-imp-abs-convergent-prod*)
have (λ(*p*,*i*). *f* (*p* ^ *i*)) *abs-summable-on* {*p*. *prime p*} × {*0*<..}
unfolding *case-prod-unfold*
by (*subst abs-summable-on-reindex-iff*[*OF inj-prime-power*]) *fact*
hence (λ*p*. $\sum_a$*i*∈{*0*<..}. *f* (*p* ^ *i*)) *abs-summable-on* {*p*. *prime p*}
by (*rule abs-summable-on-Sigma-project1′*) *simp-all*
also have *?this* ⟷ (λ*p*. ($\sum$*i*. *f* (*p* ^ *i*)) − *1*) *abs-summable-on* {*p*. *prime p*}
proof (*intro abs-summable-on-cong refl*)
fix *p* :: *nat* **assume** *p*: *p* ∈ {*p*. *prime p*}
have {*0*<..} = *UNIV* − {*0*::*nat*} **by** *auto*
also have ($\sum_a$*i*∈. . . . *f* (*p* ^ *i*)) = ($\sum$*i*. *f* (*p* ^ *i*)) − *1*
using *p* **by** (*subst infsetsum-Diff*) (*simp-all add*: *infsetsum-nat′ summable″*)
finally show ($\sum_a$*i*∈{*0*<..}. *f* (*p* ^ *i*)) = ($\sum$*i*. *f* (*p* ^ *i*)) − *1* **.**
qed

finally have *summable* (*λp. if prime p then norm* ((∑ *i. f* (*p* ^ *i*)) − *1*) *else 0*)
(**is** *summable ?T*) **by** (*simp add*: *abs-summable-on-nat-iff*)
also have *?T* = (*λp. norm* ((*if prime p then* ∑ *i. f* (*p* ^ *i*) *else 1*) − *1*))
by (*rule ext*) (*simp add*: *if-splits*)
finally show *summable* ... **.**
qed
qed

lemma
fixes *f* :: *nat* ⇒ *'a* :: {*real-normed-field,banach,second-countable-topology*}
assumes *summable*: *summable* (*λn. norm* (*f n*))
assumes *completely-multiplicative-function f*
shows *abs-convergent-euler-product'*:
abs-convergent-prod (*λp. if prime p then inverse* (*1* − *f p*) *else 1*)
and *completely-multiplicative-summable-norm*:
⋀*p. prime p* ⟹ *norm* (*f p*) < *1*
and *euler-product-LIMSEQ'*:
(*λn.* (∏ *p*≤*n. if prime p then inverse* (*1* − *f p*) *else 1*)) ⟶ (∑ *n. f n*)
proof −
interpret *f*: *completely-multiplicative-function f* **by** *fact*
{
fix *p* :: *nat* **assume** *prime p*
hence *inj* (*λi. p* ^ *i*)
by (*auto simp*: *inj-on-def dest*: *prime-gt-1-nat*)
from *summable-reindex*[*OF summable this*]
have ∗: *summable* (*λi. norm* (*f* (*p* ^ *i*))) **by** (*auto simp*: *o-def*)
also have (*λi. norm* (*f* (*p* ^ *i*))) = (*λi. norm* (*f p*) ^ *i*)
by (*simp add*: *f.power norm-power*)
finally show *norm* (*f p*) < *1*
by (*subst* (*asm*) *summable-geometric-iff*) *simp-all*
note ∗ **and** *this*
} **note** *summable'* = *this*

have *eq*: (*λp. if prime p then* (∑ *i. f* (*p* ^ *i*)) *else 1*) =
(*λp. if prime p then inverse* (*1* − *f p*) *else 1*)
proof (*rule ext*, *goal-cases*)
case (*1 p*)
show *?case*
proof (*cases prime p*)
case *True*
hence *norm* (*f p*) < *1* **by** (*rule summable'*)
from *suminf-geometric*[*OF this*] **and** *True* **show** *?thesis*
by (*simp add*: *field-simps f.power*)
qed *simp-all*
qed
hence *eq'*: (*λn.* ∏ *p*≤*n. if prime p then* ∑ *n. f* (*p* ^ *n*) *else 1*) =
(*λn.* ∏ *p*≤*n. if prime p then inverse* (*1* − *f p*) *else 1*)

by (*auto simp*: *fun-eq-iff*)

have *f*: *multiplicative-function f* **..**
from *abs-convergent-euler-product*[*OF assms*(*1*) *f*] **and** *euler-product-LIMSEQ*[*OF assms*(*1*) *f*]
show *abs-convergent-prod* (*λp. if prime p then inverse* (*1* − *f p*) *else 1*)
and (*λn.* ∏*p*≤*n. if prime p then inverse* (*1* − *f p*) *else 1*) ⟶ (∑ *n. f n*)
by (*simp-all only*: *eq eq′*)
qed

end

12 Analytic properties of Dirichlet series

theory *Dirichlet-Series-Analysis*
imports
HOL−Complex-Analysis.Complex-Analysis
HOL−Library.Going-To-Filter
HOL−Real-Asymp.Real-Asymp
Dirichlet-Series
Moebius-Mu
Partial-Summation
Euler-Products
begin

Conflicting notation from *HOL−Analysis.Infinite-Sum*

no-notation *Infinite-Sum.abs-summable-on* (**infixr** ‹*abs′-summable′-on*› *46*)

The following illustrates a concept we will need later on: A property holds for f going to F if we can find e. g. a sequence that tends to F and whose elements eventually satisfy P.

lemma *frequently-going-toI*:
assumes *filterlim* (*λn. f* (*g n*)) *F G*
assumes *eventually* (*λn. P* (*g n*)) *G*
assumes *eventually* (*λn. g n* ∈ *A*) *G*
assumes *G* ≠ *bot*
shows *frequently P* (*f going-to F within A*)
unfolding *frequently-def*
proof
assume *eventually* (*λx.* ¬*P x*) (*f going-to F within A*)
hence *eventually* (*λx.* ¬*P x*) (*inf* (*filtercomap f F*) (*principal A*))
by (*simp add*: *going-to-within-def*)
moreover have *filterlim* (*λn. g n*) (*inf* (*filtercomap f F*) (*principal A*)) *G*
using *assms* **unfolding** *filterlim-inf filterlim-principal*
by (*auto simp add*: *filterlim-iff-le-filtercomap filtercomap-filtercomap*)
ultimately have *eventually* (*λn.* ¬*P* (*g n*)) *G*
by (*rule eventually-compose-filterlim*)
with *assms*(*2*) **have** *eventually* (*λ-. False*) *G* **by** *eventually-elim auto*

with *assms*(*4*) **show** *False* **by** *simp*
qed

lemma *frequently-filtercomapI*:
assumes *filterlim* (λn. *f* (*g n*)) *F G*
assumes *eventually* (λn. *P* (*g n*)) *G*
assumes $G \neq$ *bot*
shows *frequently P* (*filtercomap f F*)
using *frequently-going-toI*[*of f g F G P UNIV*] *assms* **by** (*simp add*: *going-to-def*)

lemma *frequently-going-to-at-topE*:
fixes *f* :: *'a* $\Rightarrow$ *real*
assumes *frequently P* (*f going-to at-top*)
obtains *g* **where** $\bigwedge n$. *P* (*g n*) **and** *filterlim* (λn. *f* (*g n*)) *at-top sequentially*
proof −
from *assms* **have** $\forall k$. $\exists x$. *f x* $\geq$ *real k* $\wedge$ *P x*
by (*auto simp*: *frequently-def eventually-going-to-at-top-linorder*)
hence $\exists g$. $\forall k$. *f* (*g k*) $\geq$ *real k* $\wedge$ *P* (*g k*)
by *metis*
then obtain *g* **where** *g*: $\bigwedge k$. *f* (*g k*) $\geq$ *real k* $\bigwedge k$. *P* (*g k*)
by *blast*
have *filterlim* (λn. *f* (*g n*)) *at-top sequentially*
by (*rule filterlim-at-top-mono*[*OF filterlim-real-sequentially*]) (*use g* **in** *auto*)
from *g*(*2*) **and** *this* **show** *?thesis* **using** *that*[*of g*] **by** *blast*
qed

Apostol often uses statements like '$P(s_k)$ for all k in an infinite sequence s_k such that $\Re(s_k) \longrightarrow \infty$ as $k \to \infty$'.

Instead, we write *frequently P* (*Re going-to at-top*). This lemma shows that our statement is equivalent to his.

lemma *frequently-going-to-at-top-iff*:
frequently P (*f going-to* (*at-top* :: *real filter*)) $\longleftrightarrow$
($\exists g$. $\forall n$. *P* (*g n*) $\wedge$ *filterlim* (λn. *f* (*g n*)) *at-top sequentially*)
by (*auto intro*: *frequently-going-toI elim*!: *frequently-going-to-at-topE*)

lemma *surj-bullet-1*: *surj* (λs::*'a*::{*real-normed-algebra-1*, *real-inner*}. *s* · *1*)
proof (*rule surjI*)
fix *x* :: *real* **show** ($x *_R 1$) · (*1* :: *'a*) = *x*
by (*simp add*: *dot-square-norm*)
qed

lemma *bullet-1-going-to-at-top-neq-bot* [*simp*]:
((λs::*'a*::{*real-normed-algebra-1*, *real-inner*}. *s* · *1*) *going-to at-top*) $\neq$ *bot*
unfolding *going-to-def* **by** (*rule filtercomap-neq-bot-surj*[*OF - surj-bullet-1*]) *auto*

lemma *fds-abs-converges-altdef*:

fds-abs-converges f s ⟷ (*λn. fds-nth f n* / *nat-power n s*) *abs-summable-on* {*1*..}
by (*auto simp add*: *fds-abs-converges-def abs-summable-on-nat-iff*
intro!: *summable-cong eventually-mono*[*OF eventually-gt-at-top*[*of 0*]])

lemma *fds-abs-converges-altdef'*:
fds-abs-converges f s ⟷ (*λn. fds-nth f n* / *nat-power n s*) *abs-summable-on UNIV*
by (*subst fds-abs-converges-altdef*, *rule abs-summable-on-cong-neutral*) (*auto simp*: *Suc-le-eq*)

lemma *eval-fds-altdef*:
assumes *fds-abs-converges f s*
shows *eval-fds f s* = ($\sum_a$*n. fds-nth f n* / *nat-power n s*)
proof −
have *fds-abs-converges f s* ⟷ (*λn. fds-nth f n* / *nat-power n s*) *abs-summable-on UNIV*
unfolding *fds-abs-converges-altdef*
by (*intro abs-summable-on-cong-neutral*) (*auto simp*: *Suc-le-eq*)
with *assms* **show** *?thesis* **unfolding** *eval-fds-def fds-abs-converges-altdef*
by (*intro infsetsum-nat'* [*symmetric*]) *simp-all*
qed

lemma *multiplicative-function-divide-nat-power*:
fixes *f* :: *nat* ⇒ *'a* :: {*nat-power*, *field*}
assumes *multiplicative-function f*
shows *multiplicative-function* (*λn. f n* / *nat-power n s*)
proof
interpret *f*: *multiplicative-function f* **by** *fact*
show *f 0* / *nat-power 0 s* = *0 f 1* / *nat-power 1 s* = *1*
by *simp-all*
fix *a b* :: *nat* **assume** *a* > *1 b* > *1 coprime a b*
thus *f* (*a* ∗ *b*) / *nat-power* (*a* ∗ *b*) *s* = *f a* / *nat-power a s* ∗ (*f b* / *nat-power b s*)
by (*simp-all add*: *f.mult-coprime nat-power-mult-distrib*)
qed

lemma *completely-multiplicative-function-divide-nat-power*:
fixes *f* :: *nat* ⇒ *'a* :: {*nat-power*, *field*}
assumes *completely-multiplicative-function f*
shows *completely-multiplicative-function* (*λn. f n* / *nat-power n s*)
proof
interpret *f*: *completely-multiplicative-function f* **by** *fact*
show *f 0* / *nat-power 0 s* = *0 f* (*Suc 0*) / *nat-power* (*Suc 0*) *s* = *1*
by *simp-all*
fix *a b* :: *nat* **assume** *a* > *1 b* > *1*
thus *f* (*a* ∗ *b*) / *nat-power* (*a* ∗ *b*) *s* = *f a* / *nat-power a s* ∗ (*f b* / *nat-power b s*)
by (*simp-all add*: *f.mult nat-power-mult-distrib*)
qed

12.1 Convergence and absolute convergence

class *nat-power-normed-field* = *nat-power-field* + *real-normed-field* + *real-inner* + *real-algebra-1* +
fixes *real-power* :: *real* ⇒ ′*a* ⇒ ′*a*
assumes *real-power-nat-power*: *n* > *0* ⟹ *real-power* (*real n*) *c* = *nat-power n c*
assumes *real-power-1-right-aux*: *d* > *0* ⟹ *real-power d 1* = *d* $*_R$ *1*
assumes *real-power-add*: *d* > *0* ⟹ *real-power d* (*a* + *b*) = *real-power d a* ∗ *real-power d b*
assumes *real-power-nonzero* [*simp*]: *d* > *0* ⟹ *real-power d a* ≠ *0*
assumes *norm-real-power*: *x* > *0* ⟹ *norm* (*real-power x c*) = *x powr* (*c* • *1*)
assumes *nat-power-of-real-aux*: *nat-power n* (*x* $*_R$ *1*) = ((*real n powr x*) $*_R$ *1*)
assumes *has-field-derivative-nat-power-aux*:
⋀*x*::′*a*. *n* > *0* ⟹ *LIM y inf-class.inf*
(*Inf* (*principal* ‘ {*S*. *open S* ∧ *x* ∈ *S*})) (*principal* (*UNIV* − {*x*})).
(*nat-power n y* − *nat-power n x* − *ln* (*real n*) $*_R$ *nat-power n x* ∗ (*y* − *x*)) $/_R$
norm (*y* − *x*) :> *Inf* (*principal* ‘ {*S*. *open S* ∧ *0* ∈ *S*})
assumes *has-vector-derivative-real-power-aux*:
x > *0* ⟹ *filterlim* (λ*y*. (*real-power y c* − *real-power x* (*c* :: ′*a*) −
(*y* − *x*) $*_R$ (*c* ∗ *real-power x* (*c* − *1*))) $/_R$
norm (*y* − *x*)) (*INF S*∈{*S*. *open S* ∧ *0* ∈ *S*}. *principal S*) (*at x*)
assumes *norm-nat-power*: *n* > *0* ⟹ *norm* (*nat-power n y*) = *real n powr* (*y* • *1*)
begin

lemma *real-power-diff*: *d* > *0* ⟹ *real-power d* (*a* − *b*) = *real-power d a* / *real-power d b*
using *real-power-add*[*of d b a* − *b*] **by** (*simp add*: *field-simps*)

end

lemma *real-power-1-right* [*simp*]: *d* > *0* ⟹ *real-power d 1* = *of-real d*
using *real-power-1-right-aux*[*of d*] **by** (*simp add*: *scaleR-conv-of-real*)

lemma *has-vector-derivative-real-power* [*derivative-intros*]:
x > *0* ⟹ ((λ*y*. *real-power y c*) *has-vector-derivative c* ∗ *real-power x* (*c* − *1*)) (*at x within A*)
by (*rule has-vector-derivative-at-within*)
(*insert has-vector-derivative-real-power-aux*[*of x c*],
simp add: *has-vector-derivative-def has-derivative-def*
nhds-def bounded-linear-scaleR-left)

lemma *has-field-derivative-nat-power* [*derivative-intros*]:
n > *0* ⟹ ((λ*y*. *nat-power n y*) *has-field-derivative ln* (*real n*) $*_R$ *nat-power n x*)
(*at* (*x* :: ′*a* :: *nat-power-normed-field*) *within A*)
by (*rule has-field-derivative-at-within*)
(*insert has-field-derivative-nat-power-aux*[*of n x*],
simp only: *has-field-derivative-def has-derivative-def netlimit-at*,

simp add: *nhds-def at-within-def bounded-linear-mult-right*)

lemma *continuous-on-real-power* [*continuous-intros*]:
  $A \subseteq \{0<..\} \Longrightarrow$ *continuous-on* A (λx. *real-power* x s)
  by (*rule continuous-on-vector-derivative has-vector-derivative-real-power*)+ *auto*

instantiation *real* :: *nat-power-normed-field*
begin

definition *real-power-real* :: *real* ⇒ *real* ⇒ *real* **where**
  [*simp*]: *real-power-real* = (*powr*)

instance proof (*standard*, *goal-cases*)
  case (*7 n x*)
  hence ((λx. *nat-power n x*) *has-field-derivative ln* (*real n*) $*_R$ *nat-power n x*) (*at x*)
    by (*auto intro*!: *derivative-eq-intros simp*: *powr-def*)
  thus *?case* **unfolding** *has-field-derivative-def netlimit-at has-derivative-def*
    by (*simp add*: *nhds-def at-within-def*)
next
  case (*8 x c*)
  hence ((λy. *real-power y c*) *has-vector-derivative c* ∗ *real-power x* (*c* − *1*)) (*at x*)
    by (*auto intro*!: *derivative-eq-intros*
          simp: *has-real-derivative-iff-has-vector-derivative* [*symmetric*])
  thus *?case* **by** (*simp add*: *has-vector-derivative-def has-derivative-def nhds-def*)
qed (*simp-all add*: *powr-add*)

end

instantiation *complex* :: *nat-power-normed-field*
begin

definition *nat-power-complex* :: *nat* ⇒ *complex* ⇒ *complex* **where**
  [*simp*]: *nat-power-complex n z* = *of-nat n powr z*

definition *real-power-complex* :: *real* ⇒ *complex* ⇒ *complex* **where**
  [*simp*]: *real-power-complex* = (λx *y*. *of-real x powr y*)

instance proof
  fix *m n* :: *nat* **and** *z* :: *complex*
  assume $m > 0$ $n > 0$
  thus *nat-power* ($m * n$) *z* = *nat-power m z* ∗ *nat-power n z*
    unfolding *nat-power-complex-def of-nat-mult* **by** (*subst powr-times-real*) *simp-all*
next
  fix *n* :: *nat* **and** *z* :: *complex*
  assume $n > 0$
  show *norm* (*nat-power n z*) = *real n powr* ($z \cdot 1$) **unfolding** *nat-power-complex-def*

using *norm-powr-real-powr*[*of of-nat n z*] **by** *simp*
next
fix *n* :: *nat* **and** *x* :: *complex* **assume** *n*: *n* > *0*
hence ((*λx. nat-power n x*) *has-field-derivative ln* (*real n*) $*_R$ *nat-power n x*) (*at x*)
by (*auto intro*!: *derivative-eq-intros simp*: *powr-def scaleR-conv-of-real mult-ac*)
thus *LIM y inf-class.inf* (*Inf* (*principal* ' {*S*. *open S* ∧ *x* ∈ *S*})) (*principal* (*UNIV* − {*x*})).
(*nat-power n y* − *nat-power n x* − *ln* (*real n*) $*_R$ *nat-power n x* ∗ (*y* − *x*)) $/_R$
cmod (*y* − *x*) :> (*Inf* (*principal* ' {*S*. *open S* ∧ *0* ∈ *S*}))
unfolding *has-field-derivative-def netlimit-at has-derivative-def*
by (*simp add*: *nhds-def at-within-def*)
next
fix *x* :: *real* **and** *c* :: *complex* **assume** *x* > *0*
hence ((*λy. real-power y c*) *has-vector-derivative c* ∗ *real-power x* (*c* − *1*)) (*at x*)
by (*auto intro*!: *derivative-eq-intros has-vector-derivative-real-field*)
thus *LIM y at x*. (*real-power y c* − *real-power x c* − (*y* − *x*) $*_R$ (*c* ∗ *real-power x* (*c* − *1*))) $/_R$
norm (*y* − *x*) :> *INF S*∈{*S*. *open S* ∧ *0* ∈ *S*}. *principal S*
by (*simp add*: *has-vector-derivative-def has-derivative-def nhds-def*)
next
fix *n* :: *nat* **and** *x* :: *real*
show *nat-power n* (*x* $*_R$ *1* :: *complex*) = (*real n powr x*) $*_R$ *1*
by (*simp add*: *powr-Reals-eq scaleR-conv-of-real*)
qed (*auto simp*: *powr-def exp-add exp-of-nat-mult* [*symmetric*] *algebra-simps scaleR-conv-of-real*
simp del: *Ln-of-nat*)

end

lemma *nat-power-of-real* [*simp*]:
nat-power n (*of-real x* :: *'a* :: *nat-power-normed-field*) = *of-real* (*real n powr x*)
using *nat-power-of-real-aux*[*of n x*] **by** (*simp add*: *scaleR-conv-of-real*)

lemma *fds-abs-converges-of-real* [*simp*]:
fds-abs-converges (*fds-of-real f*)
(*of-real s* :: *'a* :: {*nat-power-normed-field*,*banach*}) ⟷ *fds-abs-converges f s*
unfolding *fds-abs-converges-def*
by (*subst* (*1 2*) *summable-Suc-iff* [*symmetric*]) (*simp add*: *norm-divide norm-nat-power*)

lemma *eval-fds-of-real* [*simp*]:
assumes *fds-converges f s*
shows *eval-fds* (*fds-of-real f*) (*of-real s* :: *'a* :: {*nat-power-normed-field*,*banach*}) =
of-real (*eval-fds f s*)
using *assms* **unfolding** *eval-fds-def* **by** (*auto simp*: *fds-converges-def suminf-of-real*)

lemma *fds-abs-summable-zeta-iff* [*simp*]:

fixes *s* :: *'a* :: {*banach*, *nat-power-normed-field*}
shows *fds-abs-converges fds-zeta s* ⟷ *s* · *1* > (*1* :: *real*)
proof −
have *fds-abs-converges fds-zeta s* ⟷ *summable* (*λn. real n powr* −(*s* · *1*))
unfolding *fds-abs-converges-def*
by (*intro summable-cong always-eventually*)
(*auto simp*: *norm-divide fds-nth-zeta powr-minus norm-nat-power divide-simps*)
also have ... ⟷ *s* · *1* > *1* **by** (*simp add*: *summable-real-powr-iff*)
finally show *?thesis* **.**
qed

lemma *fds-abs-summable-zeta*:
(*s* :: *'a* :: {*banach*, *nat-power-normed-field*}) · *1* > *1* ⟹ *fds-abs-converges fds-zeta s*
by *simp*

lemma *fds-abs-converges-moebius-mu*:
fixes *s* :: *'a* :: {*banach*,*nat-power-normed-field*}
assumes *s* · *1* > *1*
shows *fds-abs-converges* (*fds moebius-mu*) *s*
unfolding *fds-abs-converges-def*
proof (*rule summable-comparison-test*, *intro exI allI impI*)
fix *n* :: *nat*
show *norm* (*norm* (*fds-nth* (*fds moebius-mu*) *n* / *nat-power n s*)) ≤ *real n powr* (−*s* · *1*)
by (*auto simp*: *powr-minus divide-simps abs-moebius-mu-le norm-nat-power norm-divide*
moebius-mu-def norm-power)
next
from *assms* **show** *summable* (*λn. real n powr* (−*s* · *1*)) **by** (*simp add*: *summable-real-powr-iff*)
qed

definition *conv-abscissa*
:: *'a* :: {*nat-power*,*banach*,*real-normed-field*, *real-inner*} *fds* ⇒ *ereal* **where**
conv-abscissa f = (*INF s*∈{*s. fds-converges f s*}. *ereal* (*s* · *1*))

definition *abs-conv-abscissa*
:: *'a* :: {*nat-power*,*banach*,*real-normed-field*, *real-inner*} *fds* ⇒ *ereal* **where**
abs-conv-abscissa f = (*INF s*∈{*s. fds-abs-converges f s*}. *ereal* (*s* · *1*))

lemma *conv-abscissa-mono*:
assumes ⋀*s. fds-converges g s* ⟹ *fds-converges f s*
shows *conv-abscissa f* ≤ *conv-abscissa g*
unfolding *conv-abscissa-def* **by** (*rule INF-mono*) (*use assms* **in** *auto*)

lemma *abs-conv-abscissa-mono*:
assumes ⋀*s. fds-abs-converges g s* ⟹ *fds-abs-converges f s*

shows *abs-conv-abscissa f* ≤ *abs-conv-abscissa g*
unfolding *abs-conv-abscissa-def* **by** (*rule INF-mono*) (*use assms* **in** *auto*)

class *dirichlet-series* = *euclidean-space* + *real-normed-field* + *nat-power-normed-field* +
assumes *one-in-Basis*: *1* ∈ *Basis*

instance *real* :: *dirichlet-series* **by** *standard simp-all*
instance *complex* :: *dirichlet-series* **by** *standard* (*simp-all add*: *Basis-complex-def*)

context
assumes *SORT-CONSTRAINT*(*'a* :: *dirichlet-series*)
begin

lemma *fds-abs-converges-Re-le*:
fixes *f* :: *'a fds*
assumes *fds-abs-converges f z z* · *1* ≤ *z'* · *1*
shows *fds-abs-converges f z'*
unfolding *fds-abs-converges-def*
proof (*rule summable-comparison-test*, *intro exI allI impI*)
fix *n* :: *nat* **assume** *n*: *n* ≥ *1*
thus *norm* (*norm* (*fds-nth f n* / *nat-power n z'*)) ≤ *norm* (*fds-nth f n* / *nat-power n z*)
using *assms*(*2*) **by** (*simp add*: *norm-divide norm-nat-power divide-simps powr-mono mult-left-mono*)
qed (*insert assms*(*1*), *simp add*: *fds-abs-converges-def*)

lemma *fds-abs-converges*:
assumes *s* · *1* > *abs-conv-abscissa* (*f* :: *'a fds*)
shows *fds-abs-converges f s*
proof −
from *assms* **obtain** *s0* **where** *fds-abs-converges f s0 s0* · *1* < *s* · *1*
by (*auto simp*: *INF-less-iff abs-conv-abscissa-def*)
with *fds-abs-converges-Re-le*[*OF this*(*1*), *of s*] *this*(*2*) **show** *?thesis* **by** *simp*
qed

lemma *fds-abs-diverges*:
assumes *s* · *1* < *abs-conv-abscissa* (*f* :: *'a fds*)
shows ¬*fds-abs-converges f s*
proof
assume *fds-abs-converges f s*
hence *abs-conv-abscissa f* ≤ *s* · *1* **unfolding** *abs-conv-abscissa-def*
by (*intro INF-lower*) *auto*
with *assms* **show** *False* **by** *simp*
qed

lemma *uniformly-Cauchy-eval-fds-aux*:

fixes *s0* :: *'a* :: *dirichlet-series*
assumes *bounded*: *Bseq* $(\lambda n.\ \sum k{\le}n.\ fds\text{-}nth\ f\ k\ /\ nat\text{-}power\ k\ s0)$
assumes *B*: *compact B* $\bigwedge z.\ z \in B \Longrightarrow z \cdot 1 > s0 \cdot 1$
shows *uniformly-Cauchy-on B* $(\lambda N\ z.\ \sum n{\le}N.\ fds\text{-}nth\ f\ n\ /\ nat\text{-}power\ n\ z)$
proof (*cases* $B = \{\}$)
case *False*
show *?thesis*
proof (*rule uniformly-Cauchy-onI'*, *goal-cases*)
case (*1* ε)
define σ **where** $\sigma = Inf\ ((\lambda s.\ s \cdot 1)\ `\ B)$
have σ*-le*: $s \cdot 1 \ge \sigma$ **if** $s \in B$ **for** *s*
unfolding σ*-def* **using** *that*
by (*intro cInf-lower bounded-inner-imp-bdd-below compact-imp-bounded B*) *auto*
have $\sigma \in ((\lambda s.\ s \cdot 1)\ `\ B)$
unfolding σ*-def* **using** *B* ‹$B \ne \{\}$›
by (*intro closed-contains-Inf bounded-inner-imp-bdd-below compact-imp-bounded B*
compact-imp-closed compact-continuous-image continuous-intros) *auto*
with *B*(*2*) **have** σ*-gt*: $\sigma > s0 \cdot 1$ **by** *auto*
define δ **where** $\delta = \sigma - s0 \cdot 1$

have *bounded B* **by** (*rule compact-imp-bounded*) *fact*
then obtain *norm-B-aux* **where** *norm-B-aux*: $\bigwedge s.\ s \in B \Longrightarrow norm\ s \le norm\text{-}B\text{-}aux$
by (*auto simp*: *bounded-iff*)
define *norm-B* **where** *norm-B = norm-B-aux + norm s0*
from *norm-B-aux* **have** *norm-B*: *norm* $(s - s0) \le$ *norm-B* **if** $s \in B$ **for** *s*
using *norm-triangle-ineq4*[*of s s0*] *norm-B-aux*[*OF that*] **by** (*simp add*: *norm-B-def*)
then have $0 \le$ *norm-B*
by (*meson* ‹$\sigma \in (\lambda s.\ s \cdot 1)\ `\ B$› *imageE norm-ge-zero order.trans*)
define *A* **where** *A = sum-upto* $(\lambda k.\ fds\text{-}nth\ f\ k\ /\ nat\text{-}power\ k\ s0)$
from *bounded* **obtain** *C-aux* **where** *C-aux*: $\bigwedge n.\ norm\ (\sum k{\le}n.\ fds\text{-}nth\ f\ k\ /\ nat\text{-}power\ k\ s0) \le C\text{-}aux$
by (*auto simp*: *Bseq-def*)
define *C* **where** *C = max C-aux 1*
have *C-pos*: $C > 0$ **by** (*simp add*: *C-def*)
have *C*: *norm* $(A\ x) \le C$ **for** *x*
proof −
have $A\ x = (\sum k{\le}nat\ \lfloor x\rfloor.\ fds\text{-}nth\ f\ k\ /\ nat\text{-}power\ k\ s0)$
unfolding *A-def sum-upto-altdef* **by** (*intro sum.mono-neutral-left*) *auto*
also have *norm* $\ldots \le C\text{-}aux$ **by** (*rule C-aux*)
also have $\ldots \le C$ **by** (*simp add*: *C-def*)
finally show *?thesis* **.**
qed

have $(\lambda m.\ 2 * C * (1 + norm\text{-}B\ /\ \delta) * real\ m\ powr\ (-\delta)) \longlongrightarrow 0$ **unfolding** δ*-def* **using** σ*-gt*

by (*intro tendsto-mult-right-zero tendsto-neg-powr filterlim-real-sequentially*) *simp-all*
from *order-tendstoD*(*2*)[*OF this* ‹$\varepsilon > 0$›] **obtain** *M* **where**
M: $\bigwedge m.\ m \geq M \Longrightarrow 2 * C * (1 + \textit{norm-B} / \delta) * \textit{real } m \textit{ powr } - \delta < \varepsilon$
by (*auto simp*: *eventually-at-top-linorder*)

show *?case*
proof (*intro exI*[*of - max M 1*] *ballI allI impI*, *goal-cases*)
case (*1 s m n*)
from *1* **have** *s*: $s \cdot 1 > s0 \cdot 1$ **using** *B*(*2*)[*of s*] **by** *simp*
have *mn*: $m \geq M$ $m < n$ $m > 0$ $n > 0$ **using** *1* **by** (*simp-all add*:)
have *dist* ($\sum n \leq m$. *fds-nth f n* / *nat-power n s*) ($\sum n \leq n$. *fds-nth f n* / *nat-power n s*) =
dist ($\sum n \leq n$. *fds-nth f n* / *nat-power n s*) ($\sum n \leq m$. *fds-nth f n* / *nat-power n s*)
by (*simp add*: *dist-commute*)
also from *1* **have** ... = *norm* ($\sum k \in \{..n\} - \{..m\}$. *fds-nth f k* / *nat-power k s*)
by (*subst Groups-Big.sum-diff*) (*simp-all add*: *dist-norm*)
also from *1* **have** $\{..n\} - \{..m\}$ = *real* −‘ {*real m*<..*real n*} **by** *auto*
also have ($\sum k \in \ldots$. *fds-nth f k* / *nat-power k s*) =
($\sum k \in \ldots$. *fds-nth f k* / *nat-power k s0* ∗ *real-power* (*real k*) (*s0* − *s*))
(**is** - = *?S*) **by** (*intro sum.cong refl*) (*simp-all add*: *nat-power-diff real-power-nat-power*)
also have ∗: ((λt. *A t* ∗ ((*s0* − *s*) ∗ *real-power t* (*s0* − *s* − *1*))) *has-integral*
(*A* (*real n*) ∗ *real-power n* (*s0* − *s*) − *A* (*real m*) ∗ *real-power m* (*s0* − *s*) − *?S*))
{*real m*..*real n*} (**is** (*?h has-integral* -) -) **unfolding** *A-def* **using** *mn*
by (*intro partial-summation-strong*[*of* {}])
(*auto intro*!: *derivative-eq-intros continuous-intros*)
hence *?S* = *A* (*real n*) ∗ *nat-power n* (*s0* − *s*) − *A* (*real m*) ∗ *nat-power m* (*s0* − *s*) −
integral {*real m*..*real n*} *?h*
using *mn* **by** (*simp add*: *has-integral-iff real-power-nat-power*)
also have *norm* ... ≤ *norm* (*A* (*real n*) ∗ *nat-power n* (*s0* − *s*)) +
norm (*A* (*real m*) ∗ *nat-power m* (*s0* − *s*)) + *norm* (*integral* {*real m*..*real n*} *?h*)
by (*intro order.trans*[*OF norm-triangle-ineq4*] *add-right-mono order.refl*)
also have *norm* (*A* (*real n*) ∗ *nat-power n* (*s0* − *s*)) ≤ *C* ∗ *nat-power m* ((*s0* − *s*) · *1*)
using *mn* ‹$s \in B$› *C-pos s*
by (*auto simp*: *norm-mult norm-nat-power algebra-simps intro*!: *mult-mono C powr-mono2'*)
also have *norm* (*A* (*real m*) ∗ *nat-power m* (*s0* − *s*)) ≤ *C* ∗ *nat-power m* ((*s0* − *s*) · *1*)
using *mn* **by** (*auto simp*: *norm-mult norm-nat-power intro*!: *mult-mono C*)
also have *norm* (*integral* {*real m*..*real n*} *?h*) ≤
integral {*real m*..*real n*} (λt. *C* ∗ (*norm* (*s0* − *s*) ∗ *t powr* ((*s0* −

s) · *1* − *1*)))
proof (*intro integral-norm-bound-integral ballI*, *goal-cases*)
case *1*
with ∗ **show** *?case* **by** (*simp add*: *has-integral-iff*)
next
case *2*
from *mn* **show** *?case* **by** (*auto intro*!: *integrable-continuous-real continuous-intros*)
next
case (*3 t*)
thus *?case* **unfolding** *norm-mult* **using** *C-pos mn*
by (*intro mult-mono C*) (*auto simp*: *norm-real-power dot-square-norm algebra-simps*)
qed
also have ... = *C* ∗ *norm* (*s0* − *s*) ∗ *integral* {*real m..real n*} (*λt. t powr* ((*s0* − *s*) · *1* − *1*))
by (*simp add*: *algebra-simps dot-square-norm*)
also {
have ((*λt. t powr* ((*s0* − *s*) · *1* − *1*)) *has-integral*
(*real n powr* ((*s0* − *s*) · *1*) / ((*s0* − *s*) · *1*) −
real m powr ((*s0* − *s*) · *1*) / ((*s0* − *s*) · *1*))) {*m..n*}
(**is** (*?l has-integral ?I*) -) **using** *mn s*
by (*intro fundamental-theorem-of-calculus*)
(*auto intro*!: *derivative-eq-intros*
simp: *has-real-derivative-iff-has-vector-derivative* [*symmetric*] *inner-diff-left*)
hence *integral* {*real m..real n*} *?l* = *?I* **by** (*simp add*: *has-integral-iff*)
also have ... ≤ −(*real m powr* ((*s0* − *s*) · *1*) / ((*s0* − *s*) · *1*)) **using** *s mn*
by (*simp add*: *divide-simps inner-diff-left*)
also have ... = *1* ∗ (*real m powr* ((*s0* − *s*) · *1*) / ((*s* − *s0*) · *1*))
using *s* **by** (*simp add*: *field-simps inner-diff-left*)
also have ... ≤ *2* ∗ (*real m powr* ((*s0* − *s*) · *1*) / ((*s* − *s0*) · *1*)) **using** *mn s*
by (*intro mult-right-mono divide-nonneg-pos*) (*simp-all add*: *inner-diff-left*)
finally have *integral* {*m..n*} *?l* ≤ ... **.**
}
hence *C* ∗ *norm* (*s0* − *s*) ∗ *integral* {*real m..real n*} (*λt. t powr* ((*s0* − *s*) · *1* − *1*)) ≤
C ∗ *norm* (*s0* − *s*) ∗ (*2* ∗ (*real m powr* ((*s0* − *s*) · *1*) / ((*s* − *s0*) · *1*)))
using *C-pos mn*
by (*intro mult-mono mult-nonneg-nonneg integral-nonneg*
integrable-continuous-real continuous-intros) *auto*
also have *C* ∗ *nat-power m* ((*s0* − *s*) · *1*) + *C* ∗ *nat-power m* ((*s0* − *s*) · *1*) + ... =
2 ∗ *C* ∗ *nat-power m* ((*s0* − *s*) · *1*) ∗ (*1* + *norm* (*s* − *s0*) / ((*s* − *s0*) · *1*))
by (*simp add*: *algebra-simps norm-minus-commute*)
also have ... ≤ *2* ∗ *C* ∗ *nat-power m* (−*δ*) ∗ (*1* + *norm-B* / *δ*)

```
      using C-pos s mn σ-le[of s] ‹s ∈ B› σ-gt ‹0 ≤ norm-B›
      unfolding nat-power-real-def δ-def
        by (intro mult-mono powr-mono frac-le add-mono norm-B; simp add:
inner-diff-left)
    also have ... = 2 * C * (1 + norm-B / δ) * real m powr (−δ) by simp
    also from ‹m ≥ M› have ... < ε by (rule M)
    finally show ?case by − simp-all
   qed
  qed
qed (auto simp: uniformly-Cauchy-on-def)

lemma uniformly-convergent-eval-fds-aux:
  assumes Bseq (λn. ∑k≤n. fds-nth f k / nat-power k (s0 :: 'a))
  assumes B: compact B ⋀z. z ∈ B ⟹ z · 1 > s0 · 1
  shows   uniformly-convergent-on B (λN z. ∑n≤N. fds-nth f n / nat-power n z)
  by (rule Cauchy-uniformly-convergent uniformly-Cauchy-eval-fds-aux assms)+

lemma uniformly-convergent-eval-fds-aux':
  assumes conv: fds-converges f (s0 :: 'a)
  assumes B: compact B ⋀z. z ∈ B ⟹ z · 1 > s0 · 1
  shows   uniformly-convergent-on B (λN z. ∑n≤N. fds-nth f n / nat-power n z)
proof (rule uniformly-convergent-eval-fds-aux)
  from conv have convergent (λn. ∑k≤n. fds-nth f k / nat-power k s0)
    by (simp add: fds-converges-def summable-iff-convergent')
  thus Bseq (λn. ∑k≤n. fds-nth f k / nat-power k s0) by (rule convergent-imp-Bseq)
qed (insert assms, auto)

lemma bounded-partial-sums-imp-fps-converges:
  fixes s0 :: 'a :: dirichlet-series
  assumes Bseq (λn. ∑k≤n. fds-nth f k / nat-power k s0) and s · 1 > s0 · 1
  shows   fds-converges f s
proof −
  have uniformly-convergent-on {s} (λN z. ∑n≤N. fds-nth f n / nat-power n z)
using assms(2)
    by (intro uniformly-convergent-eval-fds-aux[OF assms(1)]) auto
  thus ?thesis
    by (auto simp: fds-converges-def summable-iff-convergent'
             dest: uniformly-convergent-imp-convergent)
qed

theorem fds-converges-Re-le:
  assumes fds-converges f (s0 :: 'a) s · 1 > s0 · 1
  shows   fds-converges f s
proof −
  have uniformly-convergent-on {s} (λN z. ∑n≤N. fds-nth f n / nat-power n z)
    by (rule uniformly-convergent-eval-fds-aux' assms)+ (insert assms(2), auto)
  then obtain l where uniform-limit {s} (λN z. ∑n≤N. fds-nth f n / nat-power
n z) l at-top
    by (auto simp: uniformly-convergent-on-def)
```

from *tendsto-uniform-limitI*[*OF this*, *of s*]
have (λ*n*. *fds-nth f n* / *nat-power n s*) *sums l s* **unfolding** *sums-def'*
by (*simp add*: *atLeast0AtMost*)
thus *?thesis* **by** (*simp add*: *fds-converges-def sums-iff*)
qed

lemma *fds-converges*:
assumes $s \cdot 1 > conv\text{-}abscissa\ (f :: 'a\ fds)$
shows *fds-converges f s*
proof −
from *assms* **obtain** *s0* **where** *fds-converges f s0* $s0 \cdot 1 < s \cdot 1$
by (*auto simp*: *INF-less-iff conv-abscissa-def*)
with *fds-converges-Re-le*[*OF this*(*1*), *of s*] *this*(*2*) **show** *?thesis* **by** *simp*
qed

lemma *fds-diverges*:
assumes $s \cdot 1 < conv\text{-}abscissa\ (f :: 'a\ fds)$
shows ¬*fds-converges f s*
proof
assume *fds-converges f s*
hence $conv\text{-}abscissa\ f \le s \cdot 1$ **unfolding** *conv-abscissa-def*
by (*intro INF-lower*) *auto*
with *assms* **show** *False* **by** *simp*
qed

theorem *fds-converges-imp-abs-converges*:
assumes *fds-converges* (*f* :: *'a fds*) *s* $s' \cdot 1 > s \cdot 1 + 1$
shows *fds-abs-converges f s'*
unfolding *fds-abs-converges-def*
proof (*rule summable-comparison-test-ev*)
from *assms*(*2*) **show** *summable* (λ*n*. *real n powr* $((s - s') \cdot 1)$)
by (*subst summable-real-powr-iff*) (*simp-all add*: *inner-diff-left*)
next
from *assms*(*1*) **have** (λ*n*. *fds-nth f n* / *nat-power n s*) ⟶ *0*
unfolding *fds-converges-def* **by** (*rule summable-LIMSEQ-zero*)
from *tendsto-norm*[*OF this*] **have** (λ*n*. *norm* (*fds-nth f n* / *nat-power n s*)) ⟶ *0* **by** *simp*
hence *eventually* (λ*n*. *norm* (*fds-nth f n* / *nat-power n s*) < *1*) *at-top*
by (*rule order-tendstoD*) *simp-all*
thus *eventually* (λ*n*. *norm* (*norm* (*fds-nth f n* / *nat-power n s'*)) ≤
real n powr $((s - s') \cdot 1)$) *at-top*
proof *eventually-elim*
case (*elim n*)
thus *?case*
proof (*cases n* = *0*)
case *False*
have *norm* (*fds-nth f n* / *nat-power n s'*) =
norm (*fds-nth f n*) / *real n powr* $(s' \cdot 1)$ **using** *False*
by (*simp add*: *norm-divide norm-nat-power*)

also have ... = *norm* (*fds-nth f n* / *nat-power n s*) / *real n powr* ((s' − s) · *1*) **using** *False*
by (*simp add*: *norm-divide norm-nat-power inner-diff-left powr-diff*)
also have ... ≤ *1* / *real n powr* ((s' − s) · *1*) **using** *elim*
by (*intro divide-right-mono elim*) *simp-all*
also have ... = *real n powr* ((s − s') · *1*) **using** *False*
by (*simp add*: *field-simps inner-diff-left powr-diff*)
finally show *?thesis* **by** *simp*
qed *simp-all*
qed
qed

lemma *conv-le-abs-conv-abscissa*: *conv-abscissa f* ≤ *abs-conv-abscissa f*
unfolding *conv-abscissa-def abs-conv-abscissa-def*
by (*intro INF-superset-mono*) *auto*

lemma *conv-abscissa-PInf-iff*: *conv-abscissa f* = ∞ ⟷ (∀ *s*. ¬*fds-converges f s*)
unfolding *conv-abscissa-def* **by** (*subst Inf-eq-PInfty*) *auto*

lemma *conv-abscissa-PInfI* [*intro*]: (⋀*s*. ¬*fds-converges f s*) ⟹ *conv-abscissa f* = ∞
by (*subst conv-abscissa-PInf-iff*) *auto*

lemma *conv-abscissa-MInf-iff*: *conv-abscissa* (*f* :: *'a fds*) = −∞ ⟷ (∀ *s*. *fds-converges f s*)
proof *safe*
assume ∗: ∀ *s*. *fds-converges f s*
have *conv-abscissa f* ≤ *B* **for** *B* :: *real*
using *spec*[*OF* ∗, *of of-real B*] *fds-diverges*[*of of-real B f*]
by (*cases conv-abscissa f* ≤ *B*) *simp-all*
thus *conv-abscissa f* = −∞ **by** (*rule ereal-bot*)
qed (*auto intro*: *fds-converges*)

lemma *conv-abscissa-MInfI* [*intro*]: (⋀*s*. *fds-converges* (*f*::*'a fds*) *s*) ⟹ *conv-abscissa f* = −∞
by (*subst conv-abscissa-MInf-iff*) *auto*

lemma *abs-conv-abscissa-PInf-iff*: *abs-conv-abscissa f* = ∞ ⟷ (∀ *s*. ¬*fds-abs-converges f s*)
unfolding *abs-conv-abscissa-def* **by** (*subst Inf-eq-PInfty*) *auto*

lemma *abs-conv-abscissa-PInfI* [*intro*]: (⋀*s*. ¬*fds-converges f s*) ⟹ *abs-conv-abscissa f* = ∞
by (*subst abs-conv-abscissa-PInf-iff*) *auto*

lemma *abs-conv-abscissa-MInf-iff*:
abs-conv-abscissa (*f* :: *'a fds*) = −∞ ⟷ (∀ *s*. *fds-abs-converges f s*)
proof *safe*
assume ∗: ∀ *s*. *fds-abs-converges f s*

have *abs-conv-abscissa f* $\leq$ *B* **for** *B* :: *real*
using *spec*[*OF* ∗, *of of-real B*] *fds-abs-diverges*[*of of-real B f*]
by (*cases abs-conv-abscissa f* $\leq$ *B*) *simp-all*
thus *abs-conv-abscissa f* $= -\infty$ **by** (*rule ereal-bot*)
qed (*auto intro*: *fds-abs-converges*)

lemma *abs-conv-abscissa-MInfI* [*intro*]:
($\bigwedge$*s. fds-abs-converges* (*f*::'*a fds*) *s*) $\Longrightarrow$ *abs-conv-abscissa f* $= -\infty$
by (*subst abs-conv-abscissa-MInf-iff*) *auto*

lemma *conv-abscissa-geI*:
assumes $\bigwedge$*c'*. *ereal c'* < *c* $\Longrightarrow$ ∃ *s*. *s* • *1* = *c'* ∧ ¬*fds-converges f s*
shows *conv-abscissa* (*f* :: '*a fds*) ≥ *c*
proof (*rule ccontr*)
assume ¬*conv-abscissa f* ≥ *c*
hence *c* > *conv-abscissa f* **by** *simp*
from *ereal-dense2*[*OF this*] **obtain** *c'* **where** *c* > *ereal c' c'* > *conv-abscissa f* **by** *auto*
moreover from *assms*[*OF this*(*1*)] **obtain** *s* **where** *s* • *1* = *c'* ¬*fds-converges f s* **by** *blast*
ultimately show *False* **using** *fds-converges*[*of f s*] **by** *auto*
qed

lemma *conv-abscissa-leI*:
assumes $\bigwedge$*c'*. *ereal c'* > *c* $\Longrightarrow$ ∃ *s*. *s* • *1* = *c'* ∧ *fds-converges f s*
shows *conv-abscissa* (*f* :: '*a fds*) ≤ *c*
proof (*rule ccontr*)
assume ¬*conv-abscissa f* ≤ *c*
hence *c* < *conv-abscissa f* **by** *simp*
from *ereal-dense2*[*OF this*] **obtain** *c'* **where** *c* < *ereal c' c'* < *conv-abscissa f* **by** *auto*
moreover from *assms*[*OF this*(*1*)] **obtain** *s* **where** *s* • *1* = *c' fds-converges f s* **by** *blast*
ultimately show *False* **using** *fds-diverges*[*of s f*] **by** *auto*
qed

lemma *abs-conv-abscissa-geI*:
assumes $\bigwedge$*c'*. *ereal c'* < *c* $\Longrightarrow$ ∃ *s*. *s* • *1* = *c'* ∧ ¬*fds-abs-converges f s*
shows *abs-conv-abscissa* (*f* :: '*a fds*) ≥ *c*
proof (*rule ccontr*)
assume ¬*abs-conv-abscissa f* ≥ *c*
hence *c* > *abs-conv-abscissa f* **by** *simp*
from *ereal-dense2*[*OF this*] **obtain** *c'* **where** *c* > *ereal c' c'* > *abs-conv-abscissa f* **by** *auto*
moreover from *assms*[*OF this*(*1*)] **obtain** *s* **where** *s* • *1* = *c'* ¬*fds-abs-converges f s* **by** *blast*
ultimately show *False* **using** *fds-abs-converges*[*of f s*] **by** *auto*
qed

lemma *abs-conv-abscissa-leI*:
  assumes $\bigwedge c'.$ *ereal* $c' > c \Longrightarrow \exists s.\ s \cdot 1 = c' \wedge$ *fds-abs-converges f s*
  shows *abs-conv-abscissa* (*f* :: *'a fds*) $\leq c$
proof (*rule ccontr*)
  assume ¬*abs-conv-abscissa f* $\leq c$
  hence $c <$ *abs-conv-abscissa f* **by** *simp*
  from *ereal-dense2*[*OF this*] **obtain** c' **where** $c <$ *ereal* c' $c' <$ *abs-conv-abscissa f* **by** *auto*
  moreover from *assms*[*OF this*(*1*)] **obtain** *s* **where** $s \cdot 1 = c'$ *fds-abs-converges f s* **by** *blast*
  ultimately show *False* **using** *fds-abs-diverges*[*of s f*] **by** *auto*
qed

lemma *conv-abscissa-leI-weak*:
  assumes $\bigwedge x.$ *ereal* $x > d \Longrightarrow$ *fds-converges f* (*of-real x*)
  shows *conv-abscissa* (*f* :: *'a fds*) $\leq d$
proof (*rule conv-abscissa-leI*)
  fix *x* **assume** $d <$ *ereal x*
  from *assms*[*OF this*] **show** $\exists s.\ s \cdot 1 = x \wedge$ *fds-converges f s*
    by (*intro exI*[*of - of-real x*]) *auto*
qed

lemma *abs-conv-abscissa-leI-weak*:
  assumes $\bigwedge x.$ *ereal* $x > d \Longrightarrow$ *fds-abs-converges f* (*of-real x*)
  shows *abs-conv-abscissa* (*f* :: *'a fds*) $\leq d$
proof (*rule abs-conv-abscissa-leI*)
  fix *x* **assume** $d <$ *ereal x*
  from *assms*[*OF this*] **show** $\exists s.\ s \cdot 1 = x \wedge$ *fds-abs-converges f s*
    by (*intro exI*[*of - of-real x*]) *auto*
qed

lemma *conv-abscissa-truncate* [*simp*]:
  conv-abscissa (*fds-truncate m* (*f* :: *'a fds*)) $= -\infty$
  by (*auto simp*: *conv-abscissa-MInf-iff*)

lemma *abs-conv-abscissa-truncate* [*simp*]:
  abs-conv-abscissa (*fds-truncate m* (*f* :: *'a fds*)) $= -\infty$
  by (*auto simp*: *abs-conv-abscissa-MInf-iff*)

theorem *abs-conv-le-conv-abscissa-plus-1*: *abs-conv-abscissa* (*f* :: *'a fds*) $\leq$ *conv-abscissa f* + *1*
proof (*rule abs-conv-abscissa-leI*)
  fix *c* **assume** *less*: *conv-abscissa f* + *1* < *ereal c*
  define c' **where** $c' =$ (*if conv-abscissa f* $= -\infty$ *then* $c - 2$
      else ($c - 1$ + *real-of-ereal* (*conv-abscissa f*)) / *2*)
  from *less* **have** c': *conv-abscissa f* < *ereal* $c' \wedge c' < c - 1$
    by (*cases conv-abscissa f*) (*simp-all add*: c'*-def field-simps*)

from *c′* **have** *fds-converges f* (*of-real c′*)
  by (*intro fds-converges*) (*simp-all add*: *inner-diff-left dot-square-norm*)
hence *fds-abs-converges f* (*of-real c*)
  by (*rule fds-converges-imp-abs-converges*) (*insert c′*, *simp-all*)
thus ∃ *s*. *s* · *1* = *c* ∧ *fds-abs-converges f s*
  by (*intro exI*[*of - of-real c*]) *auto*
qed

lemma *uniformly-convergent-eval-fds*:
  assumes *B*: *compact B* ⋀*z*. *z* ∈ *B* ⟹ *z* · *1* > *conv-abscissa* (*f* :: ′*a fds*)
  shows *uniformly-convergent-on B* (λ*N z*. ∑ *n*≤*N*. *fds-nth f n* / *nat-power n z*)
proof (*cases B* = {})
  case *False*
  define σ **where** σ = *Inf* ((λ*s*. *s* · *1*) ‘ *B*)
  have σ*-le*: *s* · *1* ≥ σ **if** *s* ∈ *B* **for** *s*
    unfolding σ*-def* **using** *that*
   by (*intro cInf-lower bounded-inner-imp-bdd-below compact-imp-bounded B*) *auto*
  have σ ∈ ((λ*s*. *s* · *1*) ‘ *B*)
    unfolding σ*-def* **using** *B* ‹*B* ≠ {}›
   by (*intro closed-contains-Inf bounded-inner-imp-bdd-below compact-imp-bounded*
B
        compact-imp-closed compact-continuous-image continuous-intros) *auto*
  with *B*(*2*) **have** σ*-gt*: σ > *conv-abscissa f* **by** *auto*
  define *s* **where** *s* = (*if conv-abscissa f* = −∞ *then* σ − *1 else*
                (σ + *real-of-ereal* (*conv-abscissa f*)) / *2*)
  from σ*-gt* **have** *s*: *conv-abscissa f* < *s* ∧ *s* < σ
    by (*cases conv-abscissa f*) (*auto simp*: *s-def*)
  show *?thesis* **using** *s* ‹*compact B*›
    by (*intro uniformly-convergent-eval-fds-aux′*[*of f of-real s*] *fds-converges*)
      (*auto dest*: σ*-le*)
qed *auto*

corollary *uniformly-convergent-eval-fds′*:
  assumes *B*: *compact B* ⋀*z*. *z* ∈ *B* ⟹ *z* · *1* > *conv-abscissa* (*f* :: ′*a fds*)
  shows *uniformly-convergent-on B* (λ*N z*. ∑ *n*<*N*. *fds-nth f n* / *nat-power n z*)
proof −
  from *uniformly-convergent-eval-fds*[*OF assms*] **obtain** *l* **where**
    uniform-limit B (λ*N z*. ∑ *n*≤*N*. *fds-nth f n* / *nat-power n z*) *l at-top*
    by (*auto simp*: *uniformly-convergent-on-def*)
  also have (λ*N z*. ∑ *n*≤*N*. *fds-nth f n* / *nat-power n z*) =
            (λ*N z*. ∑ *n*<*Suc N*. *fds-nth f n* / *nat-power n z*)
    by (*simp only*: *lessThan-Suc-atMost*)
  finally have *uniform-limit B* (λ*N z*. ∑ *n*<*N*. *fds-nth f n* / *nat-power n z*) *l*
at-top
    unfolding *uniform-limit-iff* **by** (*subst* (*asm*) *eventually-sequentially-Suc*)
  thus *?thesis* **by** (*auto simp*: *uniformly-convergent-on-def*)
qed

12.2 Derivative of a Dirichlet series

lemma *fds-converges-deriv-aux*:
assumes *conv*: *fds-converges f* ($s0$ :: $'a$) **and** *gt*: $s \cdot 1 > s0 \cdot 1$
shows *fds-converges* (*fds-deriv f*) *s*
proof −
have *Cauchy* ($\lambda n.\ \sum k \le n.\ (-ln\ (real\ k) *_R fds\text{-}nth\ f\ k)\ /\ nat\text{-}power\ k\ s$)
proof (*rule CauchyI'*, *goal-cases*)
case (*1* ε)
define δ **where** $\delta = s \cdot 1 - s0 \cdot 1$
define δ' **where** $\delta' = \delta\ /\ 2$
from *gt* **have** *δ-pos*: $\delta > 0$ **by** (*simp add*: *δ-def*)
define *A* **where** $A = sum\text{-}upto\ (\lambda k.\ fds\text{-}nth\ f\ k\ /\ nat\text{-}power\ k\ s0)$
from *conv* **have** *convergent* ($\lambda n.\ \sum k \le n.\ fds\text{-}nth\ f\ k\ /\ nat\text{-}power\ k\ s0$)
by (*simp add*: *fds-converges-def summable-iff-convergent'*)
hence *Bseq* ($\lambda n.\ \sum k \le n.\ fds\text{-}nth\ f\ k\ /\ nat\text{-}power\ k\ s0$) **by** (*rule convergent-imp-Bseq*)
then obtain *C-aux* **where** *C-aux*: $\bigwedge n.\ norm\ (\sum k \le n.\ fds\text{-}nth\ f\ k\ /\ nat\text{-}power\ k\ s0) \le C\text{-}aux$
by (*auto simp*: *Bseq-def*)
define *C* **where** $C = max\ C\text{-}aux\ 1$
have *C-pos*: $C > 0$ **by** (*simp add*: *C-def*)
have *C*: $norm\ (A\ x) \le C$ **for** *x*
proof −
have $A\ x = (\sum k \le nat\ \lfloor x \rfloor.\ fds\text{-}nth\ f\ k\ /\ nat\text{-}power\ k\ s0)$
unfolding *A-def sum-upto-altdef* **by** (*intro sum.mono-neutral-left*) *auto*
also have $norm\ \ldots \le C\text{-}aux$ **by** (*rule C-aux*)
also have $\ldots \le C$ **by** (*simp add*: *C-def*)
finally show *?thesis* **.**
qed
define C' **where** $C' = 2 * C + C * (norm\ (s0 - s) * (1 + 1\ /\ \delta) + 1)\ /\ \delta$

have ($\lambda m.\ C' * real\ m\ powr\ (-\delta')) \longrightarrow 0$ **unfolding** *δ'-def* **using** *gt δ-pos*
by (*intro tendsto-mult-right-zero tendsto-neg-powr filterlim-real-sequentially*) *simp-all*
from *order-tendstoD*(*2*)[*OF this* ‹$\varepsilon > 0$›] **obtain** *M1* **where**
M1: $\bigwedge m.\ m \ge M1 \Longrightarrow C' * real\ m\ powr - \delta' < \varepsilon$
by (*auto simp*: *eventually-at-top-linorder*)
have (($\lambda x.\ ln\ (real\ x)\ /\ real\ x\ powr\ \delta') \longrightarrow 0$) *at-top* **using** *δ-pos*
by (*intro lim-ln-over-power*) (*simp-all add*: *δ'-def*)
from *order-tendstoD*(*2*)[*OF this zero-less-one*] *eventually-gt-at-top*[*of 1*::*nat*]
have *eventually* ($\lambda n.\ ln\ (real\ n) \le n\ powr\ \delta'$) *at-top* **by** *eventually-elim simp-all*
then obtain *M2* **where** *M2*: $\bigwedge n.\ n \ge M2 \Longrightarrow ln\ (real\ n) \le n\ powr\ \delta'$
by (*auto simp*: *eventually-at-top-linorder*)
let $?f' = \lambda k.\ -ln\ (real\ k) *_R fds\text{-}nth\ f\ k$

show *?case*
proof (*intro exI*[*of - max* (*max M1 M2*) *1*] *allI impI*, *goal-cases*)
case (*1 m n*)

hence *mn*: $m \geq M1\ m \geq M2\ m > 0\ m < n$ **by** *simp-all*
define *g* :: *real* ⇒ *'a* **where** $g = (\lambda t.\ \textit{real-power}\ t\ (s0 - s) * \textit{of-real}\ (\ln t))$
define *g'* :: *real* ⇒ *'a*
where $g' = (\lambda t.\ \textit{real-power}\ t\ (s0 - s - 1) * ((s0 - s) * \textit{of-real}\ (\ln t) + 1))$
define *norm-g'* :: *real* ⇒ *real*
where $\textit{norm-g}' = (\lambda t.\ t\ \textit{powr}\ (-\delta - 1) * (\textit{norm}\ (s0 - s) * \ln t + 1))$
define *norm-g* :: *real* ⇒ *real*
where $\textit{norm-g} = (\lambda t.\ -(t\ \textit{powr}\ -\delta) * (\textit{norm}\ (s0 - s) * (\delta * \ln t + 1) + \delta) / \delta\hat{}2)$
have *g-g'*: (*g has-vector-derivative g' t*) (*at t*) **if** $t \in \{\textit{real}\ m..\textit{real}\ n\}$ **for** *t*
using *mn that* **by** (*auto simp*: *g-def g'-def real-power-diff field-simps real-power-add*
intro!: *derivative-eq-intros*)
have [*continuous-intros*]: *continuous-on* {*real m..real n*} *g* **using** *mn*
by (*auto simp*: *g-def intro*!: *continuous-intros*)

let $?S = \sum k \in \textit{real} -` \{\textit{real}\ m<..\textit{real}\ n\}.\ \textit{fds-nth}\ f\ k / \textit{nat-power}\ k\ s0 * g\ k$
have *dist* ($\sum k \leq m.\ ?f'\ k / \textit{nat-power}\ k\ s$) ($\sum k \leq n.\ ?f'\ k / \textit{nat-power}\ k\ s$) =
norm ($\sum k \in \{..n\} - \{..m\}.\ \textit{fds-nth}\ f\ k / \textit{nat-power}\ k\ s * \textit{of-real}\ (\ln (\textit{real}\ k))$)
using *mn* **by** (*subst sum-diff*)
(*simp-all add*: *dist-norm norm-minus-commute sum-negf scaleR-conv-of-real mult-ac*)
also have $\{..n\} - \{..m\} = \textit{real} -` \{\textit{real}\ m<..\textit{real}\ n\}$ **by** *auto*
also have ($\sum k \in \ldots.\ \textit{fds-nth}\ f\ k / \textit{nat-power}\ k\ s * \textit{of-real}\ (\ln (\textit{real}\ k))$) =
($\sum k \in \ldots.\ \textit{fds-nth}\ f\ k / \textit{nat-power}\ k\ s0 * g\ k$) **using** *mn* **unfolding** *g-def*
by (*intro sum.cong refl*) (*auto simp*: *real-power-nat-power field-simps nat-power-diff*)
also have ∗: (($\lambda t.\ A\ t * g'\ t$) *has-integral*
($A\ (\textit{real}\ n) * g\ n - A\ (\textit{real}\ m) * g\ m - ?S$))
{*real m..real n*} (**is** (*?h has-integral* -) -) **unfolding** *A-def* **using** *mn*
by (*intro partial-summation-strong*[*of* {}])
(*auto intro*!: *g-g' simp*: *field-simps continuous-intros*)
hence $?S = A\ (\textit{real}\ n) * g\ n - A\ (\textit{real}\ m) * g\ m - \textit{integral}\ \{\textit{real}\ m..\textit{real}\ n\}\ ?h$
using *mn* **by** (*simp add*: *has-integral-iff field-simps*)
also have $\textit{norm} \ldots \leq \textit{norm}\ (A\ (\textit{real}\ n) * g\ n) + \textit{norm}\ (A\ (\textit{real}\ m) * g\ m) +$
norm (*integral* {*real m..real n*} *?h*)
by (*intro order.trans*[*OF norm-triangle-ineq4*] *add-right-mono order.refl*)
also have $\textit{norm}\ (A\ (\textit{real}\ n) * g\ n) \leq C * \textit{norm}\ (g\ n)$
unfolding *norm-mult* **using** *mn C-pos* **by** (*intro mult-mono C*) *auto*
also have $\textit{norm}\ (g\ n) \leq n\ \textit{powr}\ -\delta * n\ \textit{powr}\ \delta'$ **using** *mn M2*[*of n*]
by (*simp add*: *g-def norm-real-power norm-mult δ-def inner-diff-left*)
also have $\ldots = n\ \textit{powr}\ -\delta'$ **using** *mn*
by (*simp add*: *δ'-def powr-minus field-simps powr-add* [*symmetric*])
also have $\textit{norm}\ (A\ (\textit{real}\ m) * g\ m) \leq C * \textit{norm}\ (g\ m)$
unfolding *norm-mult* **using** *mn C-pos* **by** (*intro mult-mono C*) *auto*

```
    also have norm (g m) ≤ m powr −δ ∗ m powr δ′ using mn M2[of m]
      by (simp add: g-def norm-real-power norm-mult δ-def inner-diff-left)
    also have . . . = m powr −δ′ using mn
      by (simp add: δ′-def powr-minus field-simps powr-add [symmetric])
     also have C ∗ real n powr − δ′ ≤ C ∗ real m powr − δ′ using δ-pos mn
C-pos
      by (intro mult-left-mono powr-mono2′) (simp-all add: δ′-def)
    also have . . . + . . . = 2 ∗ . . . by simp
    also have norm (integral {m..n} ?h) ≤ integral {m..n} (λt. C ∗ norm-g′ t)
    proof (intro integral-norm-bound-integral ballI, goal-cases)
      case 1
      with ∗ show ?case by (simp add: has-integral-iff)
    next
      case 2
      from mn show ?case
     by (auto intro!: integrable-continuous-real continuous-intros simp: norm-g′-def)
    next
      case (3 t)
      have norm (g′ t) ≤ norm-g′ t unfolding g′-def norm-g′-def using 3 mn
        unfolding norm-mult
        by (intro mult-mono order.trans[OF norm-triangle-ineq])
           (auto simp: norm-real-power inner-diff-left dot-square-norm norm-mult
δ-def
                intro!: mult-left-mono)
      thus ?case unfolding norm-mult using C-pos mn
        by (intro mult-mono C) simp-all
    qed
    also have . . . = C ∗ integral {m..n} norm-g′
      unfolding norm-g′-def by (simp add: norm-g′-def δ-def inner-diff-left)
    also {
      have (norm-g′ has-integral (norm-g n − norm-g m)) {m..n}
        unfolding norm-g′-def norm-g-def power2-eq-square using mn δ-pos
        by (intro fundamental-theorem-of-calculus)
           (auto simp: has-real-derivative-iff-has-vector-derivative [symmetric]
              field-simps powr-diff intro!: derivative-eq-intros)
       hence integral {m..n} norm-g′ = norm-g n − norm-g m by (simp add:
has-integral-iff)
      also have norm-g n ≤ 0 unfolding norm-g-def using δ-pos mn
           by (intro divide-nonpos-pos mult-nonpos-nonneg add-nonneg-nonneg
mult-nonneg-nonneg)
          simp-all
      hence norm-g n − norm-g m ≤ −norm-g m by simp
      also have . . . = real m powr −δ ∗ ln (real m) ∗ (norm (s0 − s)) / δ +
                   real m powr −δ ∗ ((norm (s0 − s) / δ + 1) / δ) using δ-pos
        by (simp add: field-simps norm-g-def power2-eq-square)
      also {
        have ln (real m) ≤ real m powr δ′ using M2[of m] mn by simp
        also have real m powr −δ ∗ . . . = real m powr −δ′
          by (simp add: powr-add [symmetric] δ′-def)
```

finally have *real m powr* $-\delta$ $*$ *ln* (*real m*) $*$ (*norm* (*s0* $-$ *s*)) / δ $\leq$
. . . $*$ (*norm* (*s0* $-$ *s*)) / δ **using** δ*-pos*
by (*intro divide-right-mono mult-right-mono*) (*simp-all add*: *mult-left-mono*)
}
also have *real m powr* $-\delta$ $*$ ((*norm* (*s0* $-$ *s*) / δ + *1*) / δ) $\leq$
real m powr $-\delta'$ $*$ ((*norm* (*s0* $-$ *s*) / δ + *1*) / δ) **using** *mn* δ*-pos*
by (*intro mult-right-mono powr-mono*) (*simp-all add*: δ'*-def*)
also have *real m powr* $-$ δ' $*$ *norm* (*s0* $-$ *s*) / δ + . . . =
real m powr $-\delta'$ $*$ (*norm* (*s0* $-$ *s*) $*$ (*1* + *1* / δ) + *1*) / δ **using**
δ*-pos*
by (*simp add*: *field-simps power2-eq-square*)
finally have *integral* {*real m..real n*} *norm-g'* $\leq$
real m powr $-$ δ' $*$ (*norm* (*s0* $-$ *s*) $*$ (*1* + *1* / δ) + *1*) / δ **by**
$-$ *simp-all*
}
also have *2* $*$ (*C* $*$ *m powr* $-$ δ') + *C* $*$ (*m powr* $-$ δ' $*$ (*norm* (*s0* $-$ *s*) $*$
(*1* + *1* / δ) + *1*) / δ) =
C' $*$ *m powr* $-\delta'$ **by** (*simp add*: *algebra-simps C'-def*)
also have . . . < ε **using** *M1*[*of m*] *mn* **by** *simp*
finally show *?case* **using** *C-pos* **by** $-$ *simp-all*
qed
qed
from *Cauchy-convergent*[*OF this*]
show *?thesis* **by** (*simp add*: *summable-iff-convergent' fds-converges-def fds-nth-deriv*)
qed

theorem
assumes *s* $\cdot$ *1* > *conv-abscissa* (*f* :: *'a fds*)
shows *fds-converges-deriv*: *fds-converges* (*fds-deriv f*) *s*
and *has-field-derivative-eval-fds* [*derivative-intros*]:
(*eval-fds f has-field-derivative eval-fds* (*fds-deriv f*) *s*) (*at s within A*)
proof $-$
define *s1* :: *real* **where**
s1 = (*if conv-abscissa f* = $-\infty$ *then s* $\cdot$ *1* $-$ *2 else*
(*s* $\cdot$ *1* $*$ *1* / *3* + *real-of-ereal* (*conv-abscissa f*) $*$ *2* / *3*))
define *s2* :: *real* **where**
s2 = (*if conv-abscissa f* = $-\infty$ *then s* $\cdot$ *1* $-$ *1 else*
(*s* $\cdot$ *1* $*$ *2* / *3* + *real-of-ereal* (*conv-abscissa f*) $*$ *1* / *3*))
from *assms* **have** *s*: *conv-abscissa f* < *s1* $\wedge$ *s1* < *s2* $\wedge$ *s2* < *s* $\cdot$ *1*
by (*cases conv-abscissa f*) (*auto simp*: *s1-def s2-def field-simps*)
from *s* **have** $*$: *fds-converges f* (*of-real s1*) **by** (*intro fds-converges*) *simp-all*
thus *conv'*: *fds-converges* (*fds-deriv f*) *s*
by (*rule fds-converges-deriv-aux*) (*insert s*, *simp-all*)
from $*$ **have** *conv*: *fds-converges* (*fds-deriv f*) (*of-real s2*)
by (*rule fds-converges-deriv-aux*) (*insert s*, *simp-all*)

define δ :: *real* **where** δ = (*s* $\cdot$ *1* $-$ *s2*) / *2*
from *s* **have** δ*-pos*: δ > *0* **by** (*simp add*: δ*-def*)

```
  have uniformly-convergent-on (cball s δ)
        (λn s. ∑k≤n. fds-nth (fds-deriv f) k / nat-power k s)
  proof (intro uniformly-convergent-eval-fds-aux'[OF conv])
    fix s'' :: 'a assume s'': s'' ∈ cball s δ
    have dist (s · 1) (s'' · 1) ≤ dist s s''
      by (intro Euclidean-dist-upper) (simp-all add: one-in-Basis)
    also from s'' have ... ≤ δ by simp
    finally show s'' · 1 > (of-real s2 :: 'a) · 1 using s
      by (auto simp: δ-def dist-real-def abs-if split: if-splits)
  qed (insert δ-pos, auto)
  then obtain l where
      uniform-limit (cball s δ) (λn s. ∑k≤n. fds-nth (fds-deriv f) k / nat-power k
s) l at-top
    by (auto simp: uniformly-convergent-on-def)
  also have (λn s. ∑k≤n. fds-nth (fds-deriv f) k / nat-power k s) =
              (λn s. ∑k<Suc n. fds-nth (fds-deriv f) k / nat-power k s)
    by (simp only: lessThan-Suc-atMost)
  finally have uniform-limit (cball s δ) (λn s. ∑k<n. fds-nth (fds-deriv f) k /
nat-power k s)
                  l at-top
    unfolding uniform-limit-iff by (subst (asm) eventually-sequentially-Suc)
  hence *: uniformly-convergent-on (cball s δ)
             (λn s. ∑k<n. fds-nth (fds-deriv f) k / nat-power k s)
    unfolding uniformly-convergent-on-def by blast

  have (eval-fds f has-field-derivative eval-fds (fds-deriv f) s) (at s)
    unfolding eval-fds-def
  proof (rule has-field-derivative-series'(2)[OF - - *])
    show s ∈ cball s δ s ∈ interior (cball s δ) using s by (simp-all add: δ-def)
    show summable (λn. fds-nth f n / nat-power n s)
      using assms fds-converges[of f s] by (simp add: fds-converges-def)
  next
    fix s' :: 'a and n :: nat
    show ((λs. fds-nth f n / nat-power n s) has-field-derivative
           fds-nth (fds-deriv f) n / nat-power n s') (at s' within cball s δ)
      by (cases n = 0)
         (simp, auto intro!: derivative-eq-intros simp: fds-nth-deriv field-simps)
  qed (auto simp: fds-nth-deriv intro!: derivative-eq-intros)
  thus (eval-fds f has-field-derivative eval-fds (fds-deriv f) s) (at s within A)
    by (rule has-field-derivative-at-within)
qed

lemmas has-field-derivative-eval-fds' [derivative-intros] =
  DERIV-chain2[OF has-field-derivative-eval-fds]

lemma continuous-eval-fds [continuous-intros]:
  assumes s · 1 > conv-abscissa f
  shows   continuous (at s within A) (eval-fds (f :: 'a :: dirichlet-series fds))
proof -
```

```
  have isCont (eval-fds f) s
    by (rule has-field-derivative-eval-fds DERIV-isCont assms)+
  thus ?thesis by (rule continuous-within-subset) auto
qed

lemma continuous-eval-fds' [continuous-intros]:
  fixes f :: 'a :: dirichlet-series fds
  assumes continuous (at s within A) g g s · 1 > conv-abscissa f
  shows   continuous (at s within A) (λx. eval-fds f (g x))
  by (rule continuous-within-compose3[OF - assms(1)] continuous-intros assms)+

lemma continuous-on-eval-fds [continuous-intros]:
  fixes f :: 'a :: dirichlet-series fds
  assumes A ⊆ {s. s · 1 > conv-abscissa f}
  shows   continuous-on A (eval-fds f)
  by (rule DERIV-continuous-on derivative-intros)+ (insert assms, auto)

lemma continuous-on-eval-fds' [continuous-intros]:
  fixes f :: 'a :: dirichlet-series fds
  assumes continuous-on A g g ‘ A ⊆ {s. s · 1 > conv-abscissa f}
  shows   continuous-on A (λx. eval-fds f (g x))
  by (rule continuous-on-compose2[OF continuous-on-eval-fds assms(1)])
     (insert assms, auto simp: image-iff)

lemma conv-abscissa-deriv-le:
  fixes f :: 'a fds
  shows conv-abscissa (fds-deriv f) ≤ conv-abscissa f
proof (rule conv-abscissa-leI)
  fix c' :: real
  assume ereal c' > conv-abscissa f
  thus ∃s. s · 1 = c' ∧ fds-converges (fds-deriv f) s
    by (intro exI[of - of-real c']) (auto simp: fds-converges-deriv)
qed

lemma abs-conv-abscissa-integral:
  fixes f :: 'a fds
  shows abs-conv-abscissa (fds-integral a f) = abs-conv-abscissa f
proof (rule antisym)
  show abs-conv-abscissa (fds-integral a f) ≤ abs-conv-abscissa f
  proof (rule abs-conv-abscissa-leI, goal-cases)
    case (1 c)
    have fds-abs-converges (fds-integral a f) (of-real c)
      unfolding fds-abs-converges-def
    proof (rule summable-comparison-test-ev)
      from 1 have fds-abs-converges f (of-real c)
        by (intro fds-abs-converges) auto
      thus summable (λn. norm (fds-nth f n / nat-power n (of-real c)))
        by (simp add: fds-abs-converges-def)
    next
```

show $\forall_F$ *n in sequentially. norm* (*norm* (*fds-nth* (*fds-integral a f*) *n* / *nat-power n* (*of-real c*))) ≤
norm (*fds-nth f n* / *nat-power n* (*of-real c*))
using *eventually-gt-at-top*[*of 3*]
proof *eventually-elim*
case (*elim n*)
from *elim* **and** *exp-le* **have** *ln* (*exp 1*) ≤ *ln* (*real n*)
by (*subst ln-le-cancel-iff*) *auto*
hence *1* ∗ *norm* (*fds-nth f n*) ≤ *ln* (*real n*) ∗ *norm* (*fds-nth f n*)
by (*intro mult-right-mono*) *auto*
with *elim* **show** *?case*
by (*simp add*: *norm-divide norm-nat-power fds-integral-def field-simps*)
qed
qed
thus *?case* **by** (*intro exI*[*of - of-real c*]) *auto*
qed
next
show *abs-conv-abscissa f* ≤ *abs-conv-abscissa* (*fds-integral a f*) (**is** - ≤ *?s0*)
proof (*cases abs-conv-abscissa* (*fds-integral a f*) = ∞)
case *False*
show *?thesis*
proof (*rule abs-conv-abscissa-leI*)
fix *c* :: *real*
define ε **where** ε = (*if ?s0* = −∞ *then 1 else* (*c* − *real-of-ereal ?s0*) / *2*)
assume *ereal c* > *?s0*
with *False* **have** ε: ε > *0 c* − ε > *?s0*
by (*cases ?s0*; *force simp*: ε*-def field-simps*)+

have *fds-abs-converges f* (*of-real c*)
unfolding *fds-abs-converges-def*
proof (*rule summable-comparison-test-ev*)
from ε **have** *fds-abs-converges* (*fds-integral a f*) (*of-real* (*c* − ε))
by (*intro fds-abs-converges*) (*auto simp*: *algebra-simps*)
thus *summable* (λ*n. norm* (*fds-nth* (*fds-integral a f*) *n* / *nat-power n* (*of-real* (*c* − ε))))
by (*simp add*: *fds-abs-converges-def*)
next
have $\forall_F$ *n in at-top. ln* (*real n*) / *real n powr* ε < *1*
by (*rule order-tendstoD lim-ln-over-power* ‹ε > *0*› *zero-less-one*)+
thus $\forall_F$ *n in sequentially. norm* (*norm* (*fds-nth f n* / *nat-power n* (*of-real c*)))
≤ *norm* (*fds-nth* (*fds-integral a f*) *n* / *nat-power n* (*of-real* (*c* − ε)))
using *eventually-gt-at-top*[*of 1*]
proof *eventually-elim*
case (*elim n*)
hence *ln* (*real n*) ∗ *norm* (*fds-nth f n*) ≤ *real n powr* ε ∗ *norm* (*fds-nth f n*)
by (*intro mult-right-mono*) *auto*
with *elim* **show** *?case*

```
          by (simp add: norm-divide norm-nat-power field-simps
                    powr-diff inner-diff-left fds-integral-def)
      qed
    qed
    thus ∃ s. s · 1 = c ∧ fds-abs-converges f s
      by (intro exI[of - of-real c]) auto
  qed
 qed auto
qed

lemma abs-conv-abscissa-ln:
 abs-conv-abscissa (fds-ln l (f :: 'a :: dirichlet-series fds)) =
   abs-conv-abscissa (fds-deriv f / f)
 by (simp add: fds-ln-def abs-conv-abscissa-integral)

lemma abs-conv-abscissa-deriv:
 fixes f :: 'a fds
 shows abs-conv-abscissa (fds-deriv f) = abs-conv-abscissa f
proof −
 have abs-conv-abscissa (fds-deriv f) =
      abs-conv-abscissa (fds-integral (fds-nth f 1) (fds-deriv f))
  by (rule abs-conv-abscissa-integral [symmetric])
 also have fds-integral (fds-nth f 1) (fds-deriv f) = f
  by (rule fds-integral-fds-deriv)
 finally show ?thesis .
qed

lemma abs-conv-abscissa-higher-deriv:
 abs-conv-abscissa ((fds-deriv ^^ n) f) = abs-conv-abscissa (f :: 'a :: dirichlet-series
fds)
 by (induction n) (simp-all add: abs-conv-abscissa-deriv)

lemma conv-abscissa-higher-deriv-le:
 conv-abscissa ((fds-deriv ^^ n) f) ≤ conv-abscissa (f :: 'a :: dirichlet-series fds)
 by (induction n) (auto intro: order.trans[OF conv-abscissa-deriv-le])

lemma abs-conv-abscissa-restrict:
 abs-conv-abscissa (fds-subseries P f) ≤ abs-conv-abscissa f
 by (rule abs-conv-abscissa-mono) auto

lemma eval-fds-deriv:
 fixes f :: 'a fds
 assumes s · 1 > conv-abscissa f
 shows  eval-fds (fds-deriv f) s = deriv (eval-fds f) s
 by (intro DERIV-imp-deriv [symmetric] derivative-intros assms)

lemma eval-fds-higher-deriv:
 assumes (s :: 'a :: dirichlet-series) · 1 > conv-abscissa f
 shows  eval-fds ((fds-deriv ^^ n) f) s = (deriv ^^ n) (eval-fds f) s
```

using *assms*
proof (*induction n arbitrary*: *f s*)
case (*Suc n f s*)
have *ev*: *eventually* (λs. $s \in \{s.\ s \cdot 1 > conv\text{-}abscissa\ f\}$) (*nhds s*)
using *Suc.prems open-halfspace-gt*[*of - 1*::′*a*]
by (*intro eventually-nhds-in-open*, *cases conv-abscissa f*)
(*auto simp*: *open-halfspace-gt inner-commute*)
have *eval-fds* ((*fds-deriv* ^^ *Suc n*) *f*) *s* = *eval-fds* ((*fds-deriv* ^^ *n*) (*fds-deriv f*)) *s*
by (*subst funpow-Suc-right*) *simp*
also have ... = (*deriv* ^^ *n*) (*eval-fds* (*fds-deriv f*)) *s*
by (*intro Suc.IH le-less-trans*[*OF conv-abscissa-deriv-le*] *Suc.prems*)
also have ... = (*deriv* ^^ *n*) (*deriv* (*eval-fds f*)) *s*
by (*intro higher-deriv-cong-ev refl eventually-mono*[*OF ev*] *eval-fds-deriv*) *auto*
also have ... = (*deriv* ^^ *Suc n*) (*eval-fds f*) *s*
by (*subst funpow-Suc-right*) *simp*
finally show *?case* **.**
qed *auto*

end

12.3 Multiplication of two series

lemma
fixes *f g* :: *nat* ⇒ ′*a* :: {*banach*, *real-normed-field*, *second-countable-topology*, *nat-power*}
fixes *s* :: ′*a*
assumes [*simp*]: *f 0* = *0 g 0* = *0*
assumes *summable*: *summable* (λn. *norm* (*f n* / *nat-power n s*))
summable (λn. *norm* (*g n* / *nat-power n s*))
shows *summable-dirichlet-prod*: *summable* (λn. *norm* (*dirichlet-prod f g n* / *nat-power n s*))
and *suminf-dirichlet-prod*:
($\sum n$. *dirichlet-prod f g n* / *nat-power n s*) =
($\sum n$. *f n* / *nat-power n s*) ∗ ($\sum n$. *g n* / *nat-power n s*)
proof −
have *summable*′: (λn. *f n* / *nat-power n s*) *abs-summable-on A*
(λn. *g n* / *nat-power n s*) *abs-summable-on A* **for** *A*
by ((*rule abs-summable-on-subset*[*OF - subset-UNIV*], *insert summable*,
simp add: *abs-summable-on-nat-iff*′); *fail*)+
have *f-g*: *f a* / *nat-power a s* ∗ (*g b* / *nat-power b s*) =
f a ∗ *g b* / *nat-power* (*a* ∗ *b*) *s* **for** *a b*
by (*cases a* ∗ *b* = *0*) (*auto simp*: *nat-power-mult-distrib*)

have *eq*: ($\sum_a$(*m*, *n*)∈{(*m*, *n*). *m* ∗ *n* = *x*}. *f m* ∗ *g n* / *nat-power x s*) =
dirichlet-prod f g x / *nat-power x s* **for** *x* :: *nat*
proof (*cases x* > *0*)
case *False*
hence ($\sum_a$(*m*,*n*) | *m* ∗ *n* = *x*. *f m* ∗ *g n* / *nat-power x s*) = ($\sum_a$(*m*,*n*) | *m* ∗

n = *x*. *0*)
by (*intro infsetsum-cong*) *auto*
with *False* **show** *?thesis* **by** *simp*
next
case *True*
from *finite-divisors-nat'*[*OF this*] **show** *?thesis*
by (*simp add*: *dirichlet-prod-altdef2 case-prod-unfold sum-divide-distrib*)
qed

have (λ(*m*,*n*). (*f m* / *nat-power m s*) ∗ (*g n* / *nat-power n s*)) *abs-summable-on UNIV* × *UNIV*
using *summable'* **by** (*intro abs-summable-on-product*) *auto*
also have *?this* ⟷ (λ(*m*,*n*). *f m* ∗ *g n* / *nat-power* (*m*∗*n*) *s*) *abs-summable-on UNIV*
using *f-g* **by** (*intro abs-summable-on-cong*) *auto*
also have ... ⟷ (λ(*x*,(*m*,*n*)). *f m* ∗ *g n* / *nat-power* (*m*∗*n*) *s*) *abs-summable-on*

(*SIGMA x*:*UNIV*. {(*m*,*n*). *m* ∗ *n* = *x*})
unfolding *case-prod-unfold*
by (*rule abs-summable-on-reindex-bij-betw* [*symmetric*])
(*auto simp*: *bij-betw-def inj-on-def image-iff*)
also have ... ⟷ (λ(*x*,(*m*,*n*)). *f m* ∗ *g n* / *nat-power x s*) *abs-summable-on*
(*SIGMA x*:*UNIV*. {(*m*,*n*). *m* ∗ *n* = *x*})
by (*intro abs-summable-on-cong*) *auto*
finally have *summable''*: ... .
from *abs-summable-on-Sigma-project1'*[*OF this*]
show *summable'''*: *summable* (λ*n*. *norm* (*dirichlet-prod f g n* / *nat-power n s*))
by (*simp add*: *eq abs-summable-on-nat-iff'*)

have (∑*n*. *f n* / *nat-power n s*) ∗ (∑*n*. *g n* / *nat-power n s*) =
(∑$_a$*n*. *f n* / *nat-power n s*) ∗ (∑$_a$*n*. *g n* / *nat-power n s*)
using *summable'* **by** (*simp add*: *infsetsum-nat'*)
also have ... = (∑$_a$(*m*,*n*). (*f m* / *nat-power m s*) ∗ (*g n* / *nat-power n s*))
using *summable'* **by** (*subst infsetsum-product* [*symmetric*]) *simp-all*
also have ... = (∑$_a$(*m*,*n*). *f m* ∗ *g n* / *nat-power* (*m* ∗ *n*) *s*)
using *f-g* **by** (*intro infsetsum-cong refl*) *auto*
also have ... = (∑$_a$(*x*,(*m*,*n*))∈(*SIGMA x*:*UNIV*. {(*m*,*n*). *m* ∗ *n* = *x*}).
f m ∗ *g n* / *nat-power* (*m* ∗ *n*) *s*)
unfolding *case-prod-unfold*
by (*rule infsetsum-reindex-bij-betw* [*symmetric*]) (*auto simp*: *bij-betw-def inj-on-def image-iff*)
also have ... = (∑$_a$(*x*,(*m*,*n*))∈(*SIGMA x*:*UNIV*. {(*m*,*n*). *m* ∗ *n* = *x*}).
f m ∗ *g n* / *nat-power x s*)
by (*intro infsetsum-cong refl*) (*auto simp*: *case-prod-unfold*)
also have ... = (∑$_a$*x*. *dirichlet-prod f g x* / *nat-power x s*)
(**is** - = *infsetsum ?T* -) **using** *summable''* **by** (*subst infsetsum-Sigma*) (*auto simp*: *eq*)
also have ... = (∑*x*. *dirichlet-prod f g x* / *nat-power x s*)
using *summable'''* **by** (*intro infsetsum-nat'*) (*simp-all add*: *abs-summable-on-nat-iff'*)

finally show ... = ($\sum$*n. f n / nat-power n s*) ∗ ($\sum$*n. g n / nat-power n s*) **..**
qed

lemma
fixes *f g* :: *nat* ⇒ *real*
fixes *s* :: *real*
assumes *f 0* = *0 g 0* = *0*
assumes *summable*: *summable* (*λn. norm* (*f n* / *real n powr s*))
summable (*λn. norm* (*g n* / *real n powr s*))
shows *summable-dirichlet-prod-real*: *summable* (*λn. norm* (*dirichlet-prod f g n* / *real n powr s*))
and *suminf-dirichlet-prod-real*:
($\sum$*n. dirichlet-prod f g n* / *real n powr s*) =
($\sum$*n. f n* / *nat-power n s*) ∗ ($\sum$*n. g n* / *real n powr s*)
using *summable-dirichlet-prod*[*of f g s*] *suminf-dirichlet-prod*[*of f g s*] *assms* **by** *simp-all*

lemma *fds-abs-converges-mult*:
fixes *s* :: *'a* :: {*nat-power*, *real-normed-field*, *banach*, *second-countable-topology*}
assumes *fds-abs-converges f s fds-abs-converges g s*
shows *fds-abs-converges* (*f* ∗ *g*) *s*
using *summable-dirichlet-prod*[*OF - - assms*[*unfolded fds-abs-converges-def*]]
by (*simp add*: *fds-abs-converges-def fds-nth-mult*)

lemma *fds-abs-converges-power*:
fixes *s* :: *'a* :: {*nat-power*, *real-normed-field*, *banach*, *second-countable-topology*}
shows *fds-abs-converges f s* ⟹ *fds-abs-converges* (*f* ^ *n*) *s*
by (*induction n*) (*auto intro*!: *fds-abs-converges-mult*)

lemma *fds-abs-converges-prod*:
fixes *s* :: *'a* :: {*nat-power*, *real-normed-field*, *banach*, *second-countable-topology*}
shows (⋀*x. x* ∈ *A* ⟹ *fds-abs-converges* (*f x*) *s*) ⟹ *fds-abs-converges* (*prod f A*) *s*
by (*induction A rule*: *infinite-finite-induct*) (*auto intro*!: *fds-abs-converges-mult*)

lemma *abs-conv-abscissa-mult-le*:
abs-conv-abscissa (*f* ∗ *g* :: *'a* :: *dirichlet-series fds*) ≤
max (*abs-conv-abscissa f*) (*abs-conv-abscissa g*)
proof (*rule abs-conv-abscissa-leI*, *goal-cases*)
case (*1 c'*)
thus *?case*
by (*auto intro*!: *exI*[*of - of-real c'*] *fds-abs-converges-mult intro*: *fds-abs-converges*)
qed

lemma *abs-conv-abscissa-mult-leI*:
abs-conv-abscissa (*f* :: *'a* :: *dirichlet-series fds*) ≤ *d* ⟹
abs-conv-abscissa g ≤ *d* ⟹ *abs-conv-abscissa* (*f* ∗ *g*) ≤ *d*
using *abs-conv-abscissa-mult-le*[*of f g*] **by** (*auto simp add*: *le-max-iff-disj*)

lemma *abs-conv-abscissa-shift* [*simp*]:
abs-conv-abscissa (*fds-shift c f*) = *abs-conv-abscissa* (*f* :: ′*a* :: *dirichlet-series fds*) + *c* · *1*
proof −
have *abs-conv-abscissa* (*fds-shift c f*) ≤ *abs-conv-abscissa f* + *c* · *1* **for** *c* :: ′*a* **and** *f*
proof (*rule abs-conv-abscissa-leI*)
fix *d* **assume** *abs-conv-abscissa f* + *c* · *1* < *ereal d*
hence *abs-conv-abscissa f* < *ereal* (*d* − *c* · *1*) **by** (*cases abs-conv-abscissa f*) *auto*
hence *fds-abs-converges* (*fds-shift c f*) (*of-real d*)
by (*auto intro*!: *fds-abs-converges-shift fds-abs-converges simp*: *algebra-simps*)
thus ∃ *s*. *s* · *1* = *d* ∧ *fds-abs-converges* (*fds-shift c f*) *s*
by (*auto intro*!: *exI*[*of - of-real d*])
qed
note ∗ = *this*[*of c f*] *this*[*of* −*c fds-shift c f*]
show *?thesis* **by** (*cases abs-conv-abscissa* (*fds-shift c f*); *cases abs-conv-abscissa f*)
(*insert* ∗, *auto intro*!: *antisym*)
qed

lemma *eval-fds-mult*:
fixes *s* :: ′*a* :: {*nat-power*, *real-normed-field*, *banach*, *second-countable-topology*}
assumes *fds-abs-converges f s fds-abs-converges g s*
shows *eval-fds* (*f* ∗ *g*) *s* = *eval-fds f s* ∗ *eval-fds g s*
using *suminf-dirichlet-prod*[*OF - - assms*[*unfolded fds-abs-converges-def*]]
by (*simp-all add*: *eval-fds-def fds-nth-mult*)

lemma *eval-fds-power*:
fixes *s* :: ′*a* :: {*nat-power*, *real-normed-field*, *banach*, *second-countable-topology*}
assumes *fds-abs-converges f s*
shows *eval-fds* (*f* ^ *n*) *s* = *eval-fds f s* ^ *n*
using *assms* **by** (*induction n*) (*simp-all add*: *eval-fds-mult fds-abs-converges-power*)

lemma *eval-fds-prod*:
fixes *s* :: ′*a* :: {*nat-power*, *real-normed-field*, *banach*, *second-countable-topology*}
assumes (⋀*x*. *x* ∈ *A* ⟹ *fds-abs-converges* (*f x*) *s*)
shows *eval-fds* (*prod f A*) *s* = (∏ *x*∈*A*. *eval-fds* (*f x*) *s*) **using** *assms*
by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *eval-fds-mult fds-abs-converges-prod*)

lemma *eval-fds-inverse*:
fixes *s* :: ′*a* :: {*nat-power*, *real-normed-field*, *banach*, *second-countable-topology*}
assumes *fds-abs-converges f s fds-abs-converges* (*inverse f*) *s fds-nth f 1* ≠ *0*
shows *eval-fds* (*inverse f*) *s* = *inverse* (*eval-fds f s*)
proof −
have *eval-fds* (*inverse f* ∗ *f*) *s* = *eval-fds* (*inverse f*) *s* ∗ *eval-fds f s*
by (*intro eval-fds-mult assms*)
also have *inverse f* ∗ *f* = *1* **by** (*intro fds-left-inverse assms*)
also have *eval-fds 1 s* = *1* **by** *simp*

finally show *?thesis* **by** (*auto simp*: *divide-simps*)
qed

lemma *eval-fds-integral-has-field-derivative*:
fixes *s* :: *'a* :: *dirichlet-series*
assumes *ereal* (*s* • *1*) $>$ *abs-conv-abscissa f*
assumes *fds-nth f 1* = *0*
shows (*eval-fds* (*fds-integral c f*) *has-field-derivative eval-fds f s*) (*at s*)
proof −
have *conv-abscissa* (*fds-integral c f*) ≤ *abs-conv-abscissa* (*fds-integral c f*)
by (*rule conv-le-abs-conv-abscissa*)
also from *assms* **have** . . . $<$ *ereal* (*s* • *1*) **by** (*simp add*: *abs-conv-abscissa-integral*)
finally have (*eval-fds* (*fds-integral c f*) *has-field-derivative*
eval-fds (*fds-deriv* (*fds-integral c f*)) *s*) (*at s*)
by (*intro derivative-eq-intros*) *auto*
also from *assms* **have** *fds-deriv* (*fds-integral c f*) = *f*
by *simp*
finally show *?thesis* **.**
qed

lemma *holomorphic-fds-eval* [*holomorphic-intros*]:
A ⊆ {*z*. *Re z* $>$ *conv-abscissa f*} ⟹ *eval-fds f holomorphic-on A*
unfolding *holomorphic-on-def field-differentiable-def*
by (*rule ballI exI derivative-intros*)+ *auto*

lemma *analytic-fds-eval* [*holomorphic-intros*]:
assumes *A* ⊆ {*z*. *Re z* $>$ *conv-abscissa f*}
shows *eval-fds f analytic-on A*
proof −
have *eval-fds f analytic-on* {*z*. *Re z* $>$ *conv-abscissa f*}
proof (*subst analytic-on-open*)
show *open* {*z*. *Re z* $>$ *conv-abscissa f*}
by (*cases conv-abscissa f*) (*simp-all add*: *open-halfspace-Re-gt*)
qed (*intro holomorphic-intros*, *simp-all*)
from *analytic-on-subset*[*OF this assms*] **show** *?thesis* **.**
qed

lemma *conv-abscissa-0* [*simp*]:
conv-abscissa (*0* :: *'a* :: *dirichlet-series fds*) = −∞
by (*auto simp*: *conv-abscissa-MInf-iff*)

lemma *abs-conv-abscissa-0* [*simp*]:
abs-conv-abscissa (*0* :: *'a* :: *dirichlet-series fds*) = −∞
by (*auto simp*: *abs-conv-abscissa-MInf-iff*)

lemma *conv-abscissa-1* [*simp*]:
conv-abscissa (*1* :: *'a* :: *dirichlet-series fds*) = −∞
by (*auto simp*: *conv-abscissa-MInf-iff*)

lemma *abs-conv-abscissa-1* [*simp*]:
abs-conv-abscissa (*1* :: *'a* :: *dirichlet-series fds*) = −∞
by (*auto simp*: *abs-conv-abscissa-MInf-iff*)

lemma *conv-abscissa-const* [*simp*]:
conv-abscissa (*fds-const* (*c* :: *'a* :: *dirichlet-series*)) = −∞
by (*auto simp*: *conv-abscissa-MInf-iff*)

lemma *abs-conv-abscissa-const* [*simp*]:
abs-conv-abscissa (*fds-const* (*c* :: *'a* :: *dirichlet-series*)) = −∞
by (*auto simp*: *abs-conv-abscissa-MInf-iff*)

lemma *conv-abscissa-numeral* [*simp*]:
conv-abscissa (*numeral n* :: *'a* :: *dirichlet-series fds*) = −∞
by (*auto simp*: *numeral-fds*)

lemma *abs-conv-abscissa-numeral* [*simp*]:
abs-conv-abscissa (*numeral n* :: *'a* :: *dirichlet-series fds*) = −∞
by (*auto simp*: *numeral-fds*)

lemma *abs-conv-abscissa-power-le*:
abs-conv-abscissa (*f* ^ *n* :: *'a* :: *dirichlet-series fds*) ≤ *abs-conv-abscissa f*
by (*induction n*) (*auto intro*!: *order.trans*[*OF abs-conv-abscissa-mult-le*])

lemma *abs-conv-abscissa-power-leI*:
abs-conv-abscissa (*f* :: *'a* :: *dirichlet-series fds*) ≤ *d* ⟹ *abs-conv-abscissa* (*f* ^ *n*) ≤ *d*
by (*rule order.trans*[*OF abs-conv-abscissa-power-le*])

lemma *abs-conv-abscissa-prod-le*:
assumes ⋀*x*. *x* ∈ *A* ⟹ *abs-conv-abscissa* (*f x* :: *'a* :: *dirichlet-series fds*) ≤ *d*
shows *abs-conv-abscissa* (*prod f A*) ≤ *d* **using** *assms*
by (*induction A rule*: *infinite-finite-induct*) (*auto intro*!: *abs-conv-abscissa-mult-leI*)

lemma *conv-abscissa-add-le*:
conv-abscissa (*f* + *g* :: *'a* :: *dirichlet-series fds*) ≤ *max* (*conv-abscissa f*) (*conv-abscissa g*)
by (*rule conv-abscissa-leI-weak*) (*auto intro*!: *fds-converges-add intro*: *fds-converges*)

lemma *conv-abscissa-add-leI*:
conv-abscissa (*f* :: *'a* :: *dirichlet-series fds*) ≤ *d* ⟹ *conv-abscissa g* ≤ *d* ⟹ *conv-abscissa* (*f* + *g*) ≤ *d*
using *conv-abscissa-add-le*[*of f g*] **by** (*auto simp*: *le-max-iff-disj*)

lemma *conv-abscissa-sum-leI*:
assumes ⋀*x*. *x* ∈ *A* ⟹ *conv-abscissa* (*f x* :: *'a* :: *dirichlet-series fds*) ≤ *d*
shows *conv-abscissa* (*sum f A*) ≤ *d* **using** *assms*
by (*induction A rule*: *infinite-finite-induct*) (*auto intro*!: *conv-abscissa-add-leI*)

lemma *abs-conv-abscissa-add-le*:
abs-conv-abscissa (*f* + *g* :: *'a* :: *dirichlet-series fds*) ≤ *max* (*abs-conv-abscissa f*) (*abs-conv-abscissa g*)
by (*rule abs-conv-abscissa-leI-weak*) (*auto intro*!: *fds-abs-converges-add intro*: *fds-abs-converges*)

lemma *abs-conv-abscissa-add-leI*:
abs-conv-abscissa (*f* :: *'a* :: *dirichlet-series fds*) ≤ *d* ⟹ *abs-conv-abscissa g* ≤ *d* ⟹
abs-conv-abscissa (*f* + *g*) ≤ *d*
using *abs-conv-abscissa-add-le*[*of f g*] **by** (*auto simp*: *le-max-iff-disj*)

lemma *abs-conv-abscissa-sum-leI*:
assumes ⋀*x*. *x* ∈ *A* ⟹ *abs-conv-abscissa* (*f x* :: *'a* :: *dirichlet-series fds*) ≤ *d*
shows *abs-conv-abscissa* (*sum f A*) ≤ *d* **using** *assms*
by (*induction A rule*: *infinite-finite-induct*) (*auto intro*!: *abs-conv-abscissa-add-leI*)

lemma *fds-converges-cmult-left* [*intro*]:
assumes *fds-converges f s*
shows *fds-converges* (*fds-const c* ∗ *f*) *s*
proof −
from *assms* **have** *summable* (λ*n*. *c* ∗ (*fds-nth f n* / *nat-power n s*))
by (*intro summable-mult*) (*auto simp*: *fds-converges-def*)
thus *?thesis* **by** (*simp add*: *fds-converges-def mult-ac*)
qed

lemma *fds-converges-cmult-right* [*intro*]:
assumes *fds-converges f s*
shows *fds-converges* (*f* ∗ *fds-const c*) *s*
using *fds-converges-cmult-left*[*OF assms*] **by** (*simp add*: *mult-ac*)

lemma *conv-abscissa-cmult-left* [*simp*]:
fixes *c* :: *'a* :: *dirichlet-series* **assumes** *c* ≠ *0*
shows *conv-abscissa* (*fds-const c* ∗ *f*) = *conv-abscissa f*
proof −
have *fds-converges* (*fds-const c* ∗ *f*) *s* ⟷ *fds-converges f s* **for** *s*
proof
assume *fds-converges* (*fds-const c* ∗ *f*) *s*
hence *fds-converges* (*fds-const* (*inverse c*) ∗ (*fds-const c* ∗ *f*)) *s*
by (*rule fds-converges-cmult-left*)
also have *fds-const* (*inverse c*) ∗ (*fds-const c* ∗ *f*) = *fds-const* (*inverse c* ∗ *c*) ∗ *f*
by *simp*
also have *inverse c* ∗ *c* = *1*
using *assms* **by** *simp*
finally show *fds-converges f s* **by** *simp*
qed *auto*
thus *?thesis* **by** (*simp add*: *conv-abscissa-def*)
qed

lemma *conv-abscissa-cmult-right* [*simp*]:
fixes *c* :: *'a* :: *dirichlet-series* **assumes** $c \neq 0$
shows *conv-abscissa* (*f* ∗ *fds-const c*) = *conv-abscissa f*
using *assms* **by** (*subst mult.commute*) *auto*

lemma *abs-conv-abscissa-cmult*:
fixes *c* :: *'a* :: *dirichlet-series* **assumes** $c \neq 0$
shows *abs-conv-abscissa* (*fds-const c* ∗ *f*) = *abs-conv-abscissa f*
proof (*intro antisym*)
have *abs-conv-abscissa* (*fds-const* (*inverse c*) ∗ (*fds-const c* ∗ *f*)) ≤
abs-conv-abscissa (*fds-const c* ∗ *f*)
using *abs-conv-abscissa-mult-le*[*of fds-const* (*inverse c*) *fds-const c* ∗ *f*]
by (*auto simp*: *max-def*)
also have *fds-const* (*inverse c*) ∗ (*fds-const c* ∗ *f*) = *fds-const* (*inverse c* ∗ *c*) ∗ *f*
by (*simp add*: *mult-ac*)
also have *inverse c* ∗ *c* = *1* **using** *assms* **by** *simp*
finally show *abs-conv-abscissa f* ≤ *abs-conv-abscissa* (*fds-const c* ∗ *f*) **by** *simp*
qed (*insert abs-conv-abscissa-mult-le*[*of fds-const c f*], *auto simp*: *max-def*)

lemma *conv-abscissa-minus* [*simp*]:
fixes *f* :: *'a* :: *dirichlet-series fds*
shows *conv-abscissa* (−*f*) = *conv-abscissa f*
using *conv-abscissa-cmult-left*[*of* −*1 f*] **by** *simp*

lemma *abs-conv-abscissa-minus* [*simp*]:
fixes *f* :: *'a* :: *dirichlet-series fds*
shows *abs-conv-abscissa* (−*f*) = *abs-conv-abscissa f*
using *abs-conv-abscissa-cmult*[*of* −*1 f*] **by** *simp*

lemma *conv-abscissa-diff-le*:
conv-abscissa (*f* − *g* :: *'a* :: *dirichlet-series fds*) ≤ *max* (*conv-abscissa f*) (*conv-abscissa*
g)
using *conv-abscissa-add-le*[*of f* −*g*] **by** *simp*

lemma *conv-abscissa-diff-leI*:
conv-abscissa (*f* :: *'a* :: *dirichlet-series fds*) ≤ *d* ⟹ *conv-abscissa g* ≤ *d* ⟹
conv-abscissa (*f* − *g*) ≤ *d*
using *conv-abscissa-add-le*[*of f* −*g*] **by** (*auto simp*: *le-max-iff-disj*)

lemma *abs-conv-abscissa-diff-le*:
abs-conv-abscissa (*f* − *g* :: *'a* :: *dirichlet-series fds*) ≤
max (*abs-conv-abscissa f*) (*abs-conv-abscissa g*)
using *abs-conv-abscissa-add-le*[*of f* −*g*] **by** *simp*

lemma *abs-conv-abscissa-diff-leI*:
abs-conv-abscissa (*f* :: *'a* :: *dirichlet-series fds*) ≤ *d* ⟹ *abs-conv-abscissa g* ≤ *d*
⟹
abs-conv-abscissa (*f* − *g*) ≤ *d*

using *abs-conv-abscissa-add-le*[*of f −g*] **by** (*auto simp*: *le-max-iff-disj*)

lemmas *eval-fds-integral-has-field-derivative′* [*derivative-intros*] =
DERIV-chain′[*OF - eval-fds-integral-has-field-derivative*]

lemma *abs-conv-abscissa-completely-multiplicative-log-deriv*:
fixes *f* :: *′a* :: *dirichlet-series fds*
assumes *completely-multiplicative-function* (*fds-nth f*) *fds-nth f 1* ≠ *0*
shows *abs-conv-abscissa* (*fds-deriv f* / *f*) ≤ *abs-conv-abscissa f*
proof −
have *fds-deriv f* = − *fds* (λ*n*. *fds-nth f n* ∗ *mangoldt n*) ∗ *f*
using *assms* **by** (*subst completely-multiplicative-fds-deriv′*) *simp-all*
also have ... / *f* = − *fds* (λ*n*. *fds-nth f n* ∗ *mangoldt n*) ∗ (*f* ∗ *inverse f*)
by (*simp add*: *divide-fds-def*)
also have *f* ∗ *inverse f* = *1* **using** *assms* **by** (*intro fds-right-inverse*)
finally have *fds-deriv f* / *f* = − *fds* (λ*n*. *fds-nth f n* ∗ *mangoldt n*) **by** *simp*
also have *abs-conv-abscissa* ... =
abs-conv-abscissa (*fds* (λ*n*. *fds-nth f n* ∗ *mangoldt n*))
(**is** - = *abs-conv-abscissa ?f*) **by** (*rule abs-conv-abscissa-minus*)
also have ... ≤ *abs-conv-abscissa f*
proof (*rule abs-conv-abscissa-leI*, *goal-cases*)
case (*1 c*)
have *fds-abs-converges ?f* (*of-real c*) **unfolding** *fds-abs-converges-def*
proof (*rule summable-comparison-test-ev*)
from *1* **have** *fds-abs-converges* (*fds-deriv f*) (*of-real c*)
by (*intro fds-abs-converges*) (*auto simp*: *abs-conv-abscissa-deriv*)
thus *summable* (λ*n*. |*ln* (*real n*)| ∗ *norm* (*fds-nth f n*) / *norm* (*nat-power n* (*of-real c* :: *′a*)))
by (*simp add*: *fds-abs-converges-def fds-deriv-def fds-nth-fds′*
scaleR-conv-of-real powr-minus norm-mult norm-divide norm-nat-power)
next
show ∀$_F$ *n in sequentially*.
norm (*norm* (*fds-nth* (*fds* (λ*n*. *fds-nth f n* ∗ *mangoldt n*)) *n* /
nat-power n (*of-real c*)))
≤ |*ln* (*real n*)| ∗ *norm* (*fds-nth f n*) / *norm* (*nat-power n* (*of-real c*) :: *′a*)
using *eventually-gt-at-top*[*of 0*]
proof *eventually-elim*
case (*elim n*)
have *norm* (*norm* (*fds-nth* (*fds* (λ*n*. *fds-nth f n* ∗ *mangoldt n*)) *n* /
nat-power n (*of-real c*))) =
norm (*fds-nth f n*) ∗ *mangoldt n* / *real n powr c*
using *elim* **by** (*simp add*: *fds-nth-fds′ norm-mult norm-divide*
norm-nat-power abs-mult mangoldt-nonneg)
also have ... ≤ *norm* (*fds-nth f n*) ∗ *ln n* / *real n powr c* **using** *elim*
by (*intro mult-left-mono divide-right-mono mangoldt-le*)
(*simp-all add*: *mangoldt-def*)
finally show *?case* **using** *elim* **by** (*simp add*: *norm-nat-power algebra-simps*)

qed
qed
thus *?case* **by** (*intro exI*[*of - of-real c*]) *auto*
qed
finally show *?thesis* **.**
qed

12.4 Uniqueness

context
assumes *SORT-CONSTRAINT* (*'a* :: *dirichlet-series*)
begin

lemma *norm-dirichlet-series-cutoff-le*:
assumes *fds-abs-converges f* (*s0* :: *'a*) *N* > *0 s* • *1* ≥ *c c* ≥ *s0* • *1*
shows *summable* (λ*n. fds-nth f* (*n* + *N*) / *nat-power* (*n* + *N*) *s*)
summable (λ*n. norm* (*fds-nth f* (*n* + *N*)) / *nat-power* (*n* + *N*) *c*)
and *norm* (∑*n. fds-nth f* (*n* + *N*) / *nat-power* (*n* + *N*) *s*) ≤
(∑*n. norm* (*fds-nth f* (*n* + *N*)) / *nat-power* (*n* + *N*) *c*) / *nat-power* *N* (*s* • *1* − *c*)
proof −
from *assms* **have** *fds-abs-converges f* (*of-real c*)
using *fds-abs-converges-Re-le*[*of f s0 of-real c*] **by** *auto*
hence *summable* (λ*n. norm* (*fds-nth f* (*n* + *N*) / *nat-power* (*n* + *N*) (*of-real* *c*)))
unfolding *fds-abs-converges-def* **by** (*rule summable-ignore-initial-segment*)
also have *?this* ⟷ *summable* (λ*n. norm* (*fds-nth f* (*n* + *N*)) / *nat-power* (*n* + *N*) *c*)
by (*intro summable-cong eventually-mono*[*OF eventually-gt-at-top*[*of 0*::*nat*]])
(*auto simp*: *norm-divide norm-nat-power*)
finally show ∗: *summable* (λ*n. norm* (*fds-nth f* (*n* + *N*)) / *nat-power* (*n* + *N*) *c*) **.**

from *assms* **have** *fds-abs-converges f s* **using** *fds-abs-converges-Re-le*[*of f s0 s*] **by** *auto*
hence ∗∗: *summable* (λ*n. norm* (*fds-nth f* (*n* + *N*) / *nat-power* (*n* + *N*) *s*))
unfolding *fds-abs-converges-def* **by** (*rule summable-ignore-initial-segment*)
thus *summable* (λ*n. fds-nth f* (*n* + *N*) / *nat-power* (*n* + *N*) *s*)
by (*rule summable-norm-cancel*)

have *norm* (∑*n. fds-nth f* (*n* + *N*) / *nat-power* (*n* + *N*) *s*)
≤ (∑*n. norm* (*fds-nth f* (*n* + *N*) / *nat-power* (*n* + *N*) *s*))
by (*intro summable-norm* ∗∗)
also have ... ≤ (∑*n. norm* (*fds-nth f* (*n* + *N*)) / *nat-power* (*n* + *N*) *c* / *nat-power N* (*s* • *1* − *c*))
proof (*intro suminf-le* ∗ ∗∗ *summable-divide allI*)
fix *n* :: *nat*
have *real N powr* (*s* • *1* − *c*) ≤ *real* (*n* + *N*) *powr* (*s* • *1* − *c*)
using *assms* **by** (*intro powr-mono2*) *simp-all*

also have *real* $(n + N)$ *powr* $c * \ldots = $ *real* $(n + N)$ *powr* $(s \cdot 1)$
by (*simp add*: *powr-diff*)
finally have *norm* (*fds-nth f* $(n + N)$) / *real* $(n + N)$ *powr* $(s \cdot 1) \leq$
norm (*fds-nth f* $(n + N)$) / (*real* $(n + N)$ *powr* c $*$ *real* N *powr* $(s \cdot 1 - c)$)
using ‹$N > 0$› **by** (*intro divide-left-mono*) (*simp-all add*: *mult-left-mono*)
thus *norm* (*fds-nth f* $(n + N)$ / *nat-power* $(n + N)$ s) $\leq$
norm (*fds-nth f* $(n + N)$) / *nat-power* $(n + N)$ c / *nat-power* N $(s \cdot 1 - c)$
using ‹$N > 0$› **by** (*simp add*: *norm-divide norm-nat-power*)
qed
also have $\ldots = (\sum n.$ *norm* (*fds-nth f* $(n + N)$) / *nat-power* $(n + N)$ c) / *nat-power* N $(s \cdot 1 - c)$
using $*$ **by** (*rule suminf-divide*)
finally show *norm* ($\sum n.$ *fds-nth f* $(n + N)$ / *nat-power* $(n + N)$ s) $\leq \ldots$ **.**
qed

lemma *eval-fds-zeroD-aux*:
fixes h :: *'a fds*
assumes *conv*: *fds-abs-converges h* (*s0* :: *'a*)
assumes *freq*: *frequently* ($\lambda s.$ *eval-fds h s* $= 0$) (($\lambda s.$ $s \cdot 1$) *going-to at-top*)
shows $h = 0$
proof (*rule ccontr*)
assume $h \neq 0$
hence *ex*: $\exists n{>}0.$ *fds-nth h n* $\neq 0$ **by** (*auto simp*: *fds-eq-iff*)
define N :: *nat* **where** $N = $ (*LEAST* $n.$ $n > 0 \wedge$ *fds-nth h n* $\neq 0$)
have N: $N > 0$ *fds-nth h N* $\neq 0$
using *LeastI-ex*[*OF ex*, *folded N-def*] **by** *auto*
have *less-N*: *fds-nth h n* $= 0$ **if** $n < N$ **for** n
using *Least-le*[*of* $\lambda n.$ $n > 0 \wedge$ *fds-nth h n* $\neq 0$ *n*, *folded N-def*] *that*
by (*cases* $n = 0$) (*auto simp*: *not-less*)

define c **where** $c = s0 \cdot 1$
define *remainder* **where** *remainder* $= (\lambda s.$ ($\sum n.$ *fds-nth h* $(n +$ *Suc N*$)$ / *nat-power* $(n +$ *Suc N*$)$ s))
define A **where** $A = (\sum n.$ *norm* (*fds-nth h* $(n +$ *Suc N*$)$) / *nat-power* $(n +$ *Suc N*$)$ c) $*$
nat-power (*Suc N*) c

have *eq*: *fds-nth h N* $=$ *nat-power N s* $*$ *eval-fds h s* $-$ *nat-power N s* $*$ *remainder s*
if $s \cdot 1 \geq c$ **for** s :: *'a*
proof $-$
from *conv* **and** *that* **have** *conv'*: *fds-abs-converges h s*
unfolding *c-def* **by** (*rule fds-abs-converges-Re-le*)
hence *conv''*: *fds-converges h s* **by** *blast*
from *conv''* **have** ($\lambda n.$ *fds-nth h n* / *nat-power n s*) *sums eval-fds h s*
by (*simp add*: *fds-converges-iff*)
hence ($\lambda n.$ *fds-nth h* $(n +$ *Suc N*$)$ / *nat-power* $(n +$ *Suc N*$)$ s) *sums*

(*eval-fds h s* − (∑ *n*<*Suc N*. *fds-nth h n* / *nat-power n s*))
by (*rule sums-split-initial-segment*)
also have (∑ *n*<*Suc N*. *fds-nth h n* / *nat-power n s*) =
(∑ *n*<*Suc N*. *if n* = *N then fds-nth h N* / *nat-power N s else 0*)
by (*intro sum.cong refl*) (*auto simp*: *less-N*)
also have ... = *fds-nth h N* / *nat-power N s* **by** (*subst sum.delta*) *auto*
finally show *?thesis* **unfolding** *remainder-def* **using** ‹*N* > *0*› **by** (*auto simp*: *sums-iff field-simps*)
qed

have *remainder-bound*: *norm* (*remainder s*) ≤ *A* / *real* (*Suc N*) *powr* (*s* · *1*)
if *s* · *1* ≥ *c* **for** *s* :: *'a*
proof −
note ∗ = *norm-dirichlet-series-cutoff-le*[*of h s0 Suc N c s*, *folded remainder-def*]
have *norm* (*remainder s*) ≤ (∑ *n*. *norm* (*fds-nth h* (*n* + *Suc N*)) /
nat-power (*n* + *Suc N*) *c*) / *nat-power* (*Suc N*) (*s* · *1* − *c*)
using *that assms* **unfolding** *remainder-def* **by** (*intro* ∗) (*simp-all add*: *c-def*)
also have ... = *A* / *real* (*Suc N*) *powr* (*s* · *1*) **by** (*simp add*: *A-def powr-diff*)
finally show *?thesis* **.**
qed

from *freq* **have** ∀ *c*. ∃ *s*. *s* · *1* ≥ *c* ∧ *eval-fds h s* = *0*
unfolding *frequently-def* **by** (*auto simp*: *eventually-going-to-at-top-linorder*)
hence ∀ *k*. ∃ *s*. *s* · *1* ≥ *real k* ∧ *eval-fds h s* = *0* **by** *blast*
then obtain *S* **where** *S*: ⋀*k*. *S k* · *1* ≥ *real k* ∧ *eval-fds h* (*S k*) = *0*
by *metis*
have *S-limit*: *filterlim* (λ*k*. *S k* · *1*) *at-top sequentially*
by (*rule filterlim-at-top-mono*[*OF filterlim-real-sequentially*]) (*use S* **in** *auto*)

have *eventually* (λ*k*. *real k* ≥ *c*) *sequentially* **by** *real-asymp*
hence *eventually* (λ*k*. *norm* (*fds-nth h N*) ≤
(*real N* / *real* (*Suc N*)) *powr* (*S k* · *1*) ∗ *A*) *sequentially*
proof *eventually-elim*
case (*elim k*)
hence *norm* (*fds-nth h N*) = *real N powr* (*S k* · *1*) ∗ *norm* (*remainder* (*S k*))
(**is** - = - ∗ *?X*) **using** ‹*N* > *0*› *S*[*of k*] *eq*[*of S k*]
by (*auto simp*: *norm-mult norm-nat-power c-def*)
also have *norm* (*remainder* (*S k*)) ≤ *A* / *real* (*Suc N*) *powr* (*S k* · *1*)
using *elim S*[*of k*] **by** (*intro remainder-bound*) (*simp-all add*: *c-def*)
finally show *?case*
using *N* **by** (*simp add*: *mult-left-mono powr-divide field-simps del*: *of-nat-Suc*)
qed
moreover have ((λ*k*. (*real N* / *real* (*Suc N*)) *powr* (*S k* · *1*) ∗ *A*) ⟶ *0*) *sequentially*
by (*rule filterlim-compose*[*OF - S-limit*]) (*use* ‹*N* > *0*› **in** *real-asymp*)
ultimately have ((λ-. *fds-nth h N*) ⟶ *0*) *sequentially*
by (*rule Lim-null-comparison*)
hence *fds-nth h N* = *0* **by** (*simp add*: *tendsto-const-iff*)
with ‹*fds-nth h N* ≠ *0*› **show** *False* **by** *contradiction*

```
qed

lemma eval-fds-zeroD:
  fixes h :: 'a fds
  assumes conv: conv-abscissa h < ∞
  assumes freq: frequently (λs. eval-fds h s = 0) ((λs. s · 1) going-to at-top)
  shows   h = 0
proof −
  have [simp]: 2 · (1 :: 'a) = 2
    using of-real-inner-1[of 2] unfolding of-real-numeral by simp
  from conv obtain s where fds-converges h s
    by auto
  hence fds-abs-converges h (s + 2)
    by (rule fds-converges-imp-abs-converges) (auto simp: algebra-simps)
  from this assms(2−) show ?thesis by (rule eval-fds-zeroD-aux)
qed

lemma eval-fds-eqD:
  fixes f g :: 'a fds
  assumes conv: conv-abscissa f < ∞ conv-abscissa g < ∞
  assumes eq: frequently (λs. eval-fds f s = eval-fds g s) ((λs. s · 1) going-to
at-top)
  shows   f = g
proof −
  have conv': conv-abscissa (f − g) < ∞
   using assms by (intro le-less-trans[OF conv-abscissa-diff-le]) (auto simp: max-def)

  have max (conv-abscissa f) (conv-abscissa g) < ∞
    using conv by (auto simp: max-def)
 from ereal-dense2[OF this] obtain c where c: max (conv-abscissa f) (conv-abscissa
g) < ereal c
    by auto


  have frequently (λs. eval-fds f s = eval-fds g s ∧ s · 1 ≥ c) ((λs. s · 1) going-to
at-top)
    using eq by (rule frequently-eventually-frequently) auto
  hence *: frequently (λs. eval-fds (f − g) s = 0) ((λs. s · 1) going-to at-top)
  proof (rule frequently-mono [rotated], safe, goal-cases)
    case (1 s)
    thus ?case using c
      by (subst eval-fds-diff) (auto intro!: fds-converges intro: less-le-trans)
  qed
  have f − g = 0 by (rule eval-fds-zeroD fds-abs-converges-diff assms * conv')+
  thus ?thesis by simp
qed

end
```

12.5 Limit at infinity

lemma *eval-fds-at-top-tail-bound*:
  fixes *f* :: *'a* :: *dirichlet-series fds*
  assumes *c*: *ereal c* > *abs-conv-abscissa f*
  defines $B \equiv (\sum n.$ *norm* (*fds-nth f* (*n*+*2*)) / *real* (*n*+*2*) *powr c*) ∗ *2 powr c*
  assumes *s*: *s* · *1* ≥ *c*
  shows *norm* (*eval-fds f s* − *fds-nth f 1*) ≤ *B* / *2 powr* (*s* · *1*)
proof −
  from *c* **have** *fds-abs-converges f* (*of-real c*) **by** (*intro fds-abs-converges*) *simp-all*
  also have *?this* ⟷ *summable* (*λn. norm* (*fds-nth f n*) / *real n powr c*)
    unfolding *fds-abs-converges-def*
    by (*intro summable-cong eventually-mono*[*OF eventually-gt-at-top*[*of 0*::*nat*]])
      (*auto simp*: *norm-divide norm-nat-power norm-powr-real-powr*)
  finally have *summable-c*: . . . **.**

  note *c*
  also from *s* **have** *ereal c* ≤ *ereal* (*s* · *1*) **by** *simp*
  finally have *fds-abs-converges f s* **by** (*intro fds-abs-converges*) *auto*
  hence *summable*: *summable* (*λn. norm* (*fds-nth f n* / *nat-power n s*))
    by (*simp add*: *fds-abs-converges-def*)
  from *summable-norm-cancel*[*OF this*]
    have (*λn. fds-nth f n* / *nat-power n s*) *sums eval-fds f s*
    by (*simp add*: *eval-fds-def sums-iff*)
  from *sums-split-initial-segment*[*OF this*, *of Suc* (*Suc 0*)]
    have *norm* (*eval-fds f s* − *fds-nth f 1*) = *norm* ($\sum$ *n. fds-nth f* (*n*+*2*) / *nat-power* (*n*+*2*) *s*)
    by (*auto simp*: *sums-iff*)
  also have . . . ≤ ($\sum$ *n. norm* (*fds-nth f* (*n*+*2*) / *nat-power* (*n*+*2*) *s*))
    by (*intro summable-norm summable-ignore-initial-segment summable*)
  also have . . . ≤ ($\sum$ *n. norm* (*fds-nth f* (*n*+*2*)) / *real* (*n*+*2*) *powr c* / *2 powr* (*s* · *1* − *c*))
  proof (*intro suminf-le allI*)
    fix *n* :: *nat*
    have *norm* (*fds-nth f* (*n* + *2*) / *nat-power* (*n* + *2*) *s*) =
        norm (*fds-nth f* (*n* + *2*)) / *real* (*n*+*2*) *powr c* / *real* (*n*+*2*) *powr* (*s* · *1* − *c*)
      by (*simp add*: *field-simps powr-diff norm-divide norm-nat-power*)
    also have . . . ≤ *norm* (*fds-nth f* (*n* + *2*)) / *real* (*n*+*2*) *powr c* / *2 powr* (*s* · *1* − *c*) **using** *s*
    by (*intro divide-left-mono divide-nonneg-pos powr-mono2 mult-pos-pos*) *simp-all*
    finally show *norm* (*fds-nth f* (*n* + *2*) / *nat-power* (*n* + *2*) *s*) ≤ . . . **.**
  qed (*intro summable-ignore-initial-segment summable summable-divide summable-c*)+
  also have . . . = ($\sum$ *n. norm* (*fds-nth f* (*n*+*2*)) / *real* (*n*+*2*) *powr c*) / *2 powr* (*s* · *1* − *c*)
    by (*intro suminf-divide summable-ignore-initial-segment summable-c*)
  also have . . . = *B* / *2 powr* (*s* · *1*) **by** (*simp add*: *B-def powr-diff*)
  finally show *?thesis* **.**
qed

lemma *tendsto-eval-fds-Re-at-top*:
assumes *conv-abscissa* (*f* :: *'a* :: *dirichlet-series fds*) ≠ ∞
assumes *lim*: *filterlim* (λ*x*. *S x* · *1*) *at-top F*
shows ((λ*x*. *eval-fds f* (*S x*)) ⟶ *fds-nth f 1*) *F*
proof −
from *assms*(*1*) **have** *abs-conv-abscissa f* < ∞
using *abs-conv-le-conv-abscissa-plus-1*[*of f*] **by** *auto*
from *ereal-dense2*[*OF this*] **obtain** *c* **where** *c*: *abs-conv-abscissa f* < *ereal c* **by** *auto*
define *B* **where** *B* = (∑*n*. *norm* (*fds-nth f* (*n*+*2*)) / *real* (*n*+*2*) *powr c*) ∗ *2 powr c*

have ∗: *norm* (*eval-fds f s* − *fds-nth f 1*) ≤ *B* / *2 powr* (*s* · *1*) **if** *s*: *s* · *1* ≥ *c* **for** *s*
using *eval-fds-at-top-tail-bound*[*of f c s*] *that c* **by** (*simp add*: *B-def*)
moreover from *lim* **have** *eventually* (λ*x*. *S x* · *1* ≥ *c*) *F* **by** (*auto simp*: *filterlim-at-top*)
ultimately have *eventually* (λ*x*. *norm* (*eval-fds f* (*S x*) − *fds-nth f 1*) ≤
B / *2 powr* (*S x* · *1*)) *F* **by** (*auto elim*!: *eventually-mono*)
moreover have ((λ*x*. *B* / *2 powr* (*S x* · *1*)) ⟶ *0*) *F*
using *filterlim-tendsto-pos-mult-at-top*[*OF tendsto-const*[*of ln 2*] - *lim*]
by (*intro real-tendsto-divide-at-top*[*OF tendsto-const*])
(*auto simp*: *powr-def mult-ac intro*!: *filterlim-compose*[*OF exp-at-top*])
ultimately have ((λ*x*. *eval-fds f* (*S x*) − *fds-nth f 1*) ⟶ *0*) *F*
by (*rule Lim-null-comparison*)
thus *?thesis* **by** (*subst* (*asm*) *Lim-null* [*symmetric*])
qed

lemma *tendsto-eval-fds-Re-at-top'*:
assumes *conv-abscissa* (*f* :: *complex fds*) ≠ ∞
shows *uniform-limit UNIV* (λσ *t*. *eval-fds f* (*of-real* σ + *of-real t* ∗ i)
) (λ- .*fds-nth f 1*) *at-top*
proof −
from *assms*(*1*) **have** *abs-conv-abscissa f* < ∞
using *abs-conv-le-conv-abscissa-plus-1*[*of f*] **by** *auto*
from *ereal-dense2*[*OF this*] **obtain** *c* **where** *c*: *abs-conv-abscissa f* < *ereal c* **by** *auto*
define *B* **where** *B* ≡ (∑*n*. *norm* (*fds-nth f* (*n*+*2*)) / *real* (*n*+*2*) *powr c*) ∗ *2 powr c*

show *?thesis*
unfolding *uniform-limit-iff*
proof *safe*
fix ε :: *real* **assume** ε > *0*
hence *eventually* (λσ. *B* / *2 powr* σ < ε) *at-top*
by *real-asymp*
thus *eventually* (λσ. ∀ *t*∈*UNIV*.
dist (*eval-fds f* (*of-real* σ + *of-real t* ∗ i)) (*fds-nth f 1*) < ε) *at-top*
using *eventually-ge-at-top*[*of c*]

proof *eventually-elim*
  case (*elim* σ)
  show *?case*
  proof
   fix *t* :: *real*
   have *dist* (*eval-fds f* (*of-real* σ + *of-real t* ∗ i)) (*fds-nth f 1*) ≤ *B* / *2 powr* σ
    using *eval-fds-at-top-tail-bound*[*of f c of-real* σ + *of-real t* ∗ i] *elim c*
    by (*simp add*: *dist-norm B-def*)
   also have ... < ε **by** *fact*
   finally show *dist* (*eval-fds f* (*of-real* σ + *of-real t* ∗ i)) (*fds-nth f 1*) < ε **.**
  qed
 qed
qed
qed

lemma *tendsto-eval-fds-Re-going-to-at-top*:
 assumes *conv-abscissa* (*f* :: *'a* :: *dirichlet-series fds*) ≠ ∞
 shows ((λ*s. eval-fds f s*) ⟶ *fds-nth f 1*) ((λ*s. s* · *1*) *going-to at-top*)
 using *assms* **by** (*rule tendsto-eval-fds-Re-at-top*) *auto*

lemma *tendsto-eval-fds-Re-going-to-at-top'*:
 assumes *conv-abscissa* (*f* :: *complex fds*) ≠ ∞
 shows ((λ*s. eval-fds f s*) ⟶ *fds-nth f 1*) (*Re going-to at-top*)
 using *assms* **by** (*rule tendsto-eval-fds-Re-at-top*) *auto*

Any Dirichlet series that is not identically zero and does not diverge everywhere has a half-plane in which it converges and is non-zero.

theorem *fds-nonzero-halfplane-exists*:
 fixes *f* :: *'a* :: *dirichlet-series fds*
 assumes *conv-abscissa f* < ∞ *f* ≠ *0*
 shows *eventually* (λ*s. fds-converges f s* ∧ *eval-fds f s* ≠ *0*) ((λ*s. s* · *1*) *going-to at-top*)
proof −
 from *ereal-dense2*[*OF assms*(*1*)] **obtain** *c* **where** *c*: *conv-abscissa f* < *ereal c* **by** *auto*
 have *eventually* (λ*s*::*'a. s* · *1* > *c*) ((λ*s. s* · *1*) *going-to at-top*)
  using *eventually-gt-at-top*[*of c*] **by** *auto*
 hence *eventually* (λ*s. fds-converges f s*) ((λ*s. s* · *1*) *going-to at-top*)
  by *eventually-elim* (*use c* **in** ‹*auto intro*!: *fds-converges simp*: *less-le-trans*›)
 moreover have *eventually* (λ*s. eval-fds f s* ≠ *0*) ((λ*s. s* · *1*) *going-to at-top*)
  using *eval-fds-zeroD*[*OF assms*(*1*)] *assms*(*2*) **by** (*auto simp*: *frequently-def*)
 ultimately show *?thesis* **by** (*rule eventually-conj*)
qed

12.6 Normed series

lemma *fds-converges-norm-iff* [*simp*]:
 fixes *s* :: *'a* :: {*nat-power-normed-field*,*banach*}
 shows *fds-converges* (*fds-norm f*) (*s* · *1*) ⟷ *fds-abs-converges f s*

unfolding *fds-converges-def fds-abs-converges-def*
by (*rule summable-cong* [*OF eventually-mono*[*OF eventually-gt-at-top*[*of 0*]]])
(*simp add*: *fds-abs-converges-def fds-norm-def fds-nth-fds′ norm-divide norm-nat-power*)

lemma *fds-abs-converges-norm-iff* [*simp*]:
fixes *s* :: ′*a* :: {*nat-power-normed-field*,*banach*}
shows *fds-abs-converges* (*fds-norm f*) (*s* · *1*) ⟷ *fds-abs-converges f s*
unfolding *fds-abs-converges-def*
by (*rule summable-cong* [*OF eventually-mono*[*OF eventually-gt-at-top*[*of 0*]]])
(*simp add*: *fds-abs-converges-def fds-norm-def fds-nth-fds′ norm-divide norm-nat-power*)

lemma *fds-converges-norm-iff ′*:
fixes *f* :: ′*a* :: {*nat-power-normed-field*,*banach*} *fds*
shows *fds-converges* (*fds-norm f*) *s* ⟷ *fds-abs-converges f* (*of-real s*)
unfolding *fds-converges-def fds-abs-converges-def*
by (*rule summable-cong* [*OF eventually-mono*[*OF eventually-gt-at-top*[*of 0*]]])
(*simp add*: *fds-abs-converges-def fds-norm-def fds-nth-fds′ norm-divide norm-nat-power*)

lemma *fds-abs-converges-norm-iff ′*:
fixes *f* :: ′*a* :: {*nat-power-normed-field*,*banach*} *fds*
shows *fds-abs-converges* (*fds-norm f*) *s* ⟷ *fds-abs-converges f* (*of-real s*)
unfolding *fds-abs-converges-def*
by (*rule summable-cong* [*OF eventually-mono*[*OF eventually-gt-at-top*[*of 0*]]])
(*simp add*: *fds-abs-converges-def fds-norm-def fds-nth-fds′ norm-divide norm-nat-power*)

lemma *abs-conv-abscissa-norm* [*simp*]:
fixes *f* :: ′*a* :: *dirichlet-series fds*
shows *abs-conv-abscissa* (*fds-norm f*) = *abs-conv-abscissa f*
proof (*rule antisym*)
show *abs-conv-abscissa f* ≤ *abs-conv-abscissa* (*fds-norm f*)
proof (*rule abs-conv-abscissa-leI-weak*)
fix *x* **assume** *abs-conv-abscissa* (*fds-norm f*) < *ereal x*
hence *fds-abs-converges* (*fds-norm f*) (*of-real x*) **by** (*intro fds-abs-converges*)
auto
thus *fds-abs-converges f* (*of-real x*) **by** (*simp add*: *fds-abs-converges-norm-iff ′*)
qed
qed (*auto intro*!: *abs-conv-abscissa-leI-weak simp*: *fds-abs-converges-norm-iff ′ fds-abs-converges*)

lemma *conv-abscissa-norm* [*simp*]:
fixes *f* :: ′*a* :: *dirichlet-series fds*
shows *conv-abscissa* (*fds-norm f*) = *abs-conv-abscissa f*
proof (*rule antisym*)
show *abs-conv-abscissa f* ≤ *conv-abscissa* (*fds-norm f*)
proof (*rule abs-conv-abscissa-leI-weak*)
fix *x* **assume** *conv-abscissa* (*fds-norm f*) < *ereal x*
hence *fds-converges* (*fds-norm f*) (*of-real x*) **by** (*intro fds-converges*) *auto*
thus *fds-abs-converges f* (*of-real x*) **by** (*simp add*: *fds-converges-norm-iff ′*)
qed
qed (*auto intro*!: *conv-abscissa-leI-weak simp*: *fds-abs-converges*)

```
lemma
  fixes f g :: 'a :: dirichlet-series fds
  assumes fds-abs-converges (fds-norm f) s fds-abs-converges (fds-norm g) s
  shows   fds-abs-converges-norm-mult: fds-abs-converges (fds-norm (f * g)) s
  and     eval-fds-norm-mult-le:
          eval-fds (fds-norm (f * g)) s ≤ eval-fds (fds-norm f) s * eval-fds (fds-norm
g) s
proof -
 show conv: fds-abs-converges (fds-norm (f * g)) s unfolding fds-abs-converges-def
  proof (rule summable-comparison-test-ev)
   have fds-abs-converges (fds-norm f * fds-norm g) s by (rule fds-abs-converges-mult
assms)+
    thus summable (λn. norm (fds-nth (fds-norm f * fds-norm g) n) / nat-power
n s)
      by (simp add: fds-abs-converges-def)
 qed (auto intro!: always-eventually divide-right-mono order.trans[OF fds-nth-norm-mult-le]

            simp: norm-divide)
  have conv': fds-abs-converges (fds-norm f * fds-norm g) s
    by (intro fds-abs-converges-mult assms)
  hence eval-fds (fds-norm (f * g)) s ≤ eval-fds (fds-norm f * fds-norm g) s
    using conv unfolding eval-fds-def fds-abs-converges-def norm-divide
   by (intro suminf-le allI divide-right-mono) (simp-all add: norm-mult fds-nth-norm-mult-le)
  also have ... = eval-fds (fds-norm f) s * eval-fds (fds-norm g) s
    by (intro eval-fds-mult assms)
  finally show eval-fds (fds-norm (f * g)) s ≤ eval-fds (fds-norm f) s * eval-fds
(fds-norm g) s .
qed

lemma eval-fds-norm-nonneg:
  assumes fds-abs-converges (fds-norm f) s
  shows   eval-fds (fds-norm f) s ≥ 0
  using assms unfolding eval-fds-def fds-abs-converges-def
  by (intro suminf-nonneg) auto

lemma
  fixes f :: 'a :: dirichlet-series fds
  assumes fds-abs-converges (fds-norm f) s
  shows   fds-abs-converges-norm-power: fds-abs-converges (fds-norm (f ^ n)) s
  and     eval-fds-norm-power-le:
          eval-fds (fds-norm (f ^ n)) s ≤ eval-fds (fds-norm f) s ^ n
proof -
  show *: fds-abs-converges (fds-norm (f ^ n)) s for n
    by (induction n) (auto intro!: fds-abs-converges-norm-mult assms)
  show eval-fds (fds-norm (f ^ n)) s ≤ eval-fds (fds-norm f) s ^ n
    by (induction n) (auto intro!: order.trans[OF eval-fds-norm-mult-le] assms *
                              mult-left-mono eval-fds-norm-nonneg)
qed
```

12.7 Logarithms of Dirichlet series

lemma *eventually-gt-ereal-at-top*: $c \neq \infty \Longrightarrow$ *eventually* $(\lambda x.\ ereal\ x > c)$ *at-top*
by (*cases c*) *auto*

lemma *eval-fds-log-deriv*:
fixes $s :: 'a ::$ *dirichlet-series*
assumes *fds-nth f 1* $\neq 0$ $s \cdot 1 >$ *abs-conv-abscissa f*
$s \cdot 1 >$ *abs-conv-abscissa* (*fds-deriv f* / *f*)
assumes *eval-fds f s* $\neq 0$
shows *eval-fds* (*fds-deriv f* / *f*) *s* = *eval-fds* (*fds-deriv f*) *s* / *eval-fds f s*
proof −
have *eval-fds* (*fds-deriv f* / *f* ∗ *f*) *s* = *eval-fds* (*fds-deriv f* / *f*) *s* ∗ *eval-fds f s*
using *assms* **by** (*intro eval-fds-mult fds-abs-converges*) *auto*
also have *fds-deriv f* / *f* ∗ *f* = *fds-deriv f* ∗ (*f* ∗ *inverse f*)
by (*simp add*: *divide-fds-def algebra-simps*)
also have *f* ∗ *inverse f* = *1* **using** *assms* **by** (*intro fds-right-inverse*)
finally show *?thesis* **using** *assms* **by** *simp*
qed

Given a sufficiently nice absolutely convergent Dirichlet series that converges to some function $f(s)$ and a holomorphic branch of $\ln f(s)$, we can construct a Dirichlet series that absolutely converges to that logarithm.

lemma *eval-fds-ln*:
fixes *s0* :: *ereal*
assumes *nz*: $\bigwedge s.$ *Re s* > *s0* $\Longrightarrow$ *eval-fds f s* $\neq 0$ *fds-nth f 1* $\neq 0$
assumes *l*: *exp l* = *fds-nth f 1* ((*g* ∘ *of-real*) ⟶ *l*) *at-top*
assumes *g*: $\bigwedge s.$ *Re s* > *s0* $\Longrightarrow$ *exp* (*g s*) = *eval-fds f s*
assumes *holo-g*: *g holomorphic-on* {*s. Re s* > *s0*}
assumes *ereal* (*Re s*) > *s0*
assumes *s0* ≥ *abs-conv-abscissa f* **and** *s0* ≥ *abs-conv-abscissa* (*fds-deriv f* / *f*)
shows *eval-fds* (*fds-ln l f*) *s* = *g s*
proof −
let *?s0* = *abs-conv-abscissa f* **and** *?s1* = *abs-conv-abscissa* (*inverse f*)
let *?h* = λ*s. eval-fds* (*fds-ln l f*) *s* − *g s*
let *?A* = {*s. Re s* > *s0*}
have *open-A*: *open ?A* **by** (*cases s0*) (*auto simp*: *open-halfspace-Re-gt*)

have *conv-abscissa f* ≤ *abs-conv-abscissa f* **by** (*rule conv-le-abs-conv-abscissa*)
moreover from *assms* **have** ... ≠ ∞ **by** *auto*
ultimately have *conv-abscissa f* ≠ ∞ **by** *auto*

have *conv-abscissa* (*fds-ln l f*) ≤ *abs-conv-abscissa* (*fds-ln l f*)
by (*rule conv-le-abs-conv-abscissa*)
also have ... ≤ *abs-conv-abscissa* (*fds-deriv f* / *f*)
unfolding *fds-ln-def* **by** (*simp add*: *abs-conv-abscissa-integral*)
finally have *conv-abscissa* (*fds-ln l f*) ≠ ∞
using *assms* **by** (*auto simp*: *max-def abs-conv-abscissa-deriv split*: *if-splits*)

```
have deriv-g [derivative-intros]:
  (g has-field-derivative eval-fds (fds-deriv f) s / eval-fds f s) (at s within B)
  if s: Re s > s0 for s B
proof −
  have conv-abscissa f ≤ abs-conv-abscissa f by (rule conv-le-abs-conv-abscissa)
  also have ... ≤ s0 using assms by simp
  also have ... < Re s by fact
  finally have s': Re s > conv-abscissa f .

  have deriv-g: (g has-field-derivative deriv g s) (at s)
    using holomorphic-derivI[OF holo-g open-A, of s] s
    by (auto simp: at-within-open[OF - open-A])
  have ((λs. exp (g s)) has-field-derivative eval-fds f s ∗ deriv g s) (at s) (is ?P)
    by (rule derivative-eq-intros deriv-g s)+ (insert s, simp-all add: g)
  also from s have ev: eventually (λt. t ∈ ?A) (nhds s)
    by (intro eventually-nhds-in-open open-A) auto
  have ?P ←→ (eval-fds f has-field-derivative eval-fds f s ∗ deriv g s) (at s)
    by (intro DERIV-cong-ev refl eventually-mono[OF ev]) (auto simp: g)
  finally have (eval-fds f has-field-derivative eval-fds f s ∗ deriv g s) (at s) .
  moreover have (eval-fds f has-field-derivative eval-fds (fds-deriv f) s) (at s)
    using s' assms by (intro derivative-intros) auto
  ultimately have eval-fds f s ∗ deriv g s = eval-fds (fds-deriv f) s
    by (rule DERIV-unique)
  hence deriv g s = eval-fds (fds-deriv f) s / eval-fds f s
    using s nz by (simp add: field-simps)
  with deriv-g show ?thesis by (auto intro: has-field-derivative-at-within)
qed

have ∃ c. ∀ z∈{z. Re z > s0}. ?h z = c
proof (rule has-field-derivative-zero-constant, goal-cases)
  case 1
  show ?case using convex-halfspace-gt[of - 1::complex]
    by (cases s0) auto
next
  case (2 z)
  have conv-abscissa (fds-ln l f) ≤ abs-conv-abscissa (fds-ln l f)
    by (rule conv-le-abs-conv-abscissa)
  also have ... ≤ abs-conv-abscissa (fds-deriv f / f)
    by (simp add: abs-conv-abscissa-ln)
  also have ... < Re z using 2 assms by (auto simp: abs-conv-abscissa-deriv)
  finally have s1: conv-abscissa (fds-ln l f) < ereal (Re z) .

  have conv-abscissa f ≤ abs-conv-abscissa f
    by (rule conv-le-abs-conv-abscissa)
  also have ... < Re z using 2 assms by auto
  finally have s2: conv-abscissa f < ereal (Re z) .

  from l have fds-nth f 1 ≠ 0 by auto
  with 2 assms have ∗: eval-fds (fds-deriv f / f) z = eval-fds (fds-deriv f) z /
```

(*eval-fds f z*)
    by (*auto simp*: *eval-fds-log-deriv*)
  have *eval-fds f z* ≠ *0* **using** *2 assms* **by** *auto*
  show *?case* **using** *s1 s2 2 nz*
    by (*auto intro*!: *derivative-eq-intros simp*: ∗ *field-simps*)
qed
then obtain *c* **where** *c*: ⋀*z*. *Re z* > *s0* ⟹ *?h z* = *c* **by** *blast*

have (*at-top* :: *real filter*) ≠ *bot* **by** *simp*
moreover from *assms* **have** *s0* ≠ ∞ **by** *auto*
have *eventually* (λ*x*. *c* = (*?h* ∘ *of-real*) *x*) *at-top*
  using *eventually-gt-ereal-at-top*[*OF* ‹*s0* ≠ ∞›] **by** *eventually-elim* (*simp add*: *c*)
hence ((*?h* ∘ *of-real*) ⟶ *c*) *at-top*
  by (*force intro*: *Lim-transform-eventually*)
moreover have ((*?h* ∘ *of-real*) ⟶ *fds-nth* (*fds-ln l f*) *1* − *l*) *at-top*
  using ‹*conv-abscissa* (*fds-ln l f*) ≠ ∞› **and** *l* **unfolding** *o-def*
  by (*intro tendsto-intros tendsto-eval-fds-Re-at-top*) (*auto simp*: *filterlim-ident*)
ultimately have *c* = *fds-nth* (*fds-ln l f*) *1* − *l*
  by (*rule tendsto-unique*)
with *c*[*OF* ‹*Re s* > *s0*›] **and** *l* **and** *nz* **show** *?thesis*
  by (*simp add*: *exp-minus field-simps*)
qed

Less explicitly: For a sufficiently nice absolutely convergent Dirichlet series converging to a function $f(s)$, the formal logarithm absolutely converges to some logarithm of $f(s)$.

lemma *eval-fds-ln'*:
  fixes *s0* :: *ereal*
  assumes *ereal* (*Re s*) > *s0*
  assumes *s0* ≥ *abs-conv-abscissa f* **and** *s0* ≥ *abs-conv-abscissa* (*fds-deriv f* / *f*)
    and *nz*: ⋀*s*. *Re s* > *s0* ⟹ *eval-fds f s* ≠ *0 fds-nth f 1* ≠ *0*
  assumes *l*: *exp l* = *fds-nth f 1*
  shows *exp* (*eval-fds* (*fds-ln l f*) *s*) = *eval-fds f s*
proof −
  let *?s0* = *abs-conv-abscissa f* **and** *?s1* = *abs-conv-abscissa* (*inverse f*)
  let *?h* = λ*s*. *eval-fds f s* ∗ *exp* (−*eval-fds* (*fds-ln l f*) *s*)

  have *conv-abscissa f* ≤ *abs-conv-abscissa f* **by** (*rule conv-le-abs-conv-abscissa*)
  moreover from *assms* **have** ... ≠ ∞ **by** *auto*
  ultimately have *conv-abscissa f* ≠ ∞ **by** *auto*

  have *conv-abscissa* (*fds-ln l f*) ≤ *abs-conv-abscissa* (*fds-ln l f*)
    by (*rule conv-le-abs-conv-abscissa*)
  also have ... ≤ *abs-conv-abscissa* (*fds-deriv f* / *f*)
    unfolding *fds-ln-def* **by** (*simp add*: *abs-conv-abscissa-integral*)
  finally have *conv-abscissa* (*fds-ln l f*) ≠ ∞
    using *assms* **by** (*auto simp*: *max-def abs-conv-abscissa-deriv split*: *if-splits*)

have ∃ *c*. ∀ *z*∈{*z*. *Re z* > *s0*}. *?h z* = *c*
proof (*rule has-field-derivative-zero-constant*, *goal-cases*)
case *1*
show *?case* **using** *convex-halfspace-gt*[*of* - *1*::*complex*]
by (*cases s0*) *auto*
next
case (*2 z*)
have *conv-abscissa* (*fds-ln l f*) ≤ *abs-conv-abscissa* (*fds-ln l f*)
by (*rule conv-le-abs-conv-abscissa*)
also have . . . ≤ *abs-conv-abscissa* (*fds-deriv f* / *f*)
unfolding *fds-ln-def* **by** (*simp add*: *abs-conv-abscissa-integral*)
also have . . . < *Re z* **using** *2 assms* **by** (*auto simp*: *abs-conv-abscissa-deriv*)
finally have *s1*: *conv-abscissa* (*fds-ln l f*) < *ereal* (*Re z*) **.**

have *conv-abscissa f* ≤ *abs-conv-abscissa f*
by (*rule conv-le-abs-conv-abscissa*)
also have . . . < *Re z* **using** *2 assms* **by** *auto*
finally have *s2*: *conv-abscissa f* < *ereal* (*Re z*) **.**

from *l* **have** *fds-nth f 1* ≠ *0* **by** *auto*
with *2 assms* **have** ∗: *eval-fds* (*fds-deriv f* / *f*) *z* = *eval-fds* (*fds-deriv f*) *z* / (*eval-fds f z*)
by (*subst eval-fds-log-deriv*) *auto*
have *eval-fds f z* ≠ *0* **using** *2 assms* **by** *auto*
thus *?case* **using** *s1 s2*
by (*auto intro*!: *derivative-eq-intros simp*: ∗)
qed
then obtain *c* **where** *c*: ⋀*z*. *Re z* > *s0* ⟹ *?h z* = *c* **by** *blast*

have (*at-top* :: *real filter*) ≠ *bot* **by** *simp*
moreover from *assms* **have** *s0* ≠ ∞ **by** *auto*
have *eventually* (λ*x*. *c* = (*?h* ∘ *of-real*) *x*) *at-top*
using *eventually-gt-ereal-at-top*[*OF* ‹*s0* ≠ ∞›] **by** *eventually-elim* (*simp add*: *c*)
hence ((*?h* ∘ *of-real*) ⟶ *c*) *at-top*
by (*force intro*: *Lim-transform-eventually*)
moreover have ((*?h* ∘ *of-real*) ⟶ *fds-nth f 1* ∗ *exp* (−*fds-nth* (*fds-ln l f*) *1*)) *at-top*
unfolding *o-def* **using** ‹*conv-abscissa* (*fds-ln l f*) ≠ ∞› **and** ‹*conv-abscissa f* ≠ ∞›
by (*intro tendsto-intros tendsto-eval-fds-Re-at-top*) (*auto simp*: *filterlim-ident*)
ultimately have *c* = *fds-nth f 1* ∗ *exp* (−*fds-nth* (*fds-ln l f*) *1*)
by (*rule tendsto-unique*)
with *c*[*OF* ‹*Re s* > *s0*›] **and** *l* **and** *nz* **show** *?thesis*
by (*simp add*: *exp-minus field-simps*)
qed

lemma *fds-ln-completely-multiplicative*:
fixes *f* :: ′*a* :: *dirichlet-series fds*

assumes *completely-multiplicative-function* (*fds-nth f*)
assumes *fds-nth f 1* ≠ *0*
shows *fds-ln l f* = *fds* (λ*n*. *if n* = *1 then l else fds-nth f n* ∗ *mangoldt n* $/_R$ *ln n*)
proof −
have *fds-ln l f* = *fds-integral l* (*fds-deriv f* / *f*)
by (*simp add*: *fds-ln-def*)
also have *fds-deriv f* = −*fds* (λ*n*. *fds-nth f n* ∗ *mangoldt n*) ∗ *f*
by (*intro completely-multiplicative-fds-deriv′ assms*)
also have ... / *f* = −*fds* (λ*n*. *fds-nth f n* ∗ *mangoldt n*) ∗ (*f* ∗ *inverse f*)
by (*simp add*: *divide-fds-def*)
also from *assms* **have** *f* ∗ *inverse f* = *1*
by (*simp add*: *fds-right-inverse*)
also have *fds-integral l* (− *fds* (λ*n*. *fds-nth f n* ∗ *mangoldt n*) ∗ *1*) =
fds (λ*n*. *if n* = *1 then l else fds-nth f n* ∗ *mangoldt n* $/_R$ *ln n*)
by (*simp add*: *fds-integral-def cong*: *if-cong*)
finally show *?thesis* **.**
qed

lemma *eval-fds-ln-completely-multiplicative-strong*:
fixes *s* :: ′*a* :: *dirichlet-series* **and** *l* :: ′*a* **and** *f* :: ′*a fds* **and** *g* :: *nat* ⇒ ′*a*
defines *h* ≡ *fds* (λ*n*. *fds-nth* (*fds-ln l f*) *n* ∗ *g n*)
assumes *fds-abs-converges h s*
assumes *completely-multiplicative-function* (*fds-nth f*) **and** *fds-nth f 1* ≠ *0*
shows (λ(*p*,*k*). (*fds-nth f p* / *nat-power p s*) ^ *Suc k* ∗ *g* (*p* ^ *Suc k*) / *of-nat* (*Suc k*))
abs-summable-on ({*p*. *prime p*} × *UNIV*) (**is** *?th1*)
and *eval-fds h s* = *l* ∗ *g 1* + ($\sum_a$(*p*, *k*)∈{*p*. *prime p*}×*UNIV*.
(*fds-nth f p* / *nat-power p s*) ^ *Suc k* ∗ *g* (*p* ^ *Suc k*) / *of-nat* (*Suc k*))
(**is** *?th2*)
proof −
let *?P* = {*p*::*nat*. *prime p*}
interpret *f*: *completely-multiplicative-function fds-nth f* **by** *fact*
from *assms* **have** ∗: (λ*n*. *fds-nth h n* / *nat-power n s*) *abs-summable-on UNIV*
by (*auto simp*: *abs-summable-on-nat-iff′ fds-abs-converges-def*)
have *eq*: *h* = *fds* (λ*n*. *if n* = *1 then l* ∗ *g 1 else fds-nth f n* ∗ *g n* ∗ *mangoldt n* $/_R$ *ln* (*real n*))
using *fds-ln-completely-multiplicative* [*OF assms*(*3*), *of l*]
by (*simp add*: *h-def fds-eq-iff*)

note ∗
also have (λ*n*. *fds-nth h n* / *nat-power n s*) *abs-summable-on UNIV* ⟷
(λ*x*. *if x* = *Suc 0 then l* ∗ *g 1 else fds-nth f x* ∗ *g x* ∗ *mangoldt x* $/_R$ *ln* (*real x*) /
nat-power x s) *abs-summable-on* {*1*} ∪ *Collect primepow*
using *eq* **by** (*intro abs-summable-on-cong-neutral*) (*auto simp*: *fds-nth-fds mangoldt-def*)
finally have *sum1*: (λ*x*. *if x* = *Suc 0 then l* ∗ *g 1 else*
fds-nth f x ∗ *g x* ∗ *mangoldt x* $/_R$ *ln* (*real x*) / *nat-power x s*)

abs-summable-on Collect primepow
by (*rule abs-summable-on-subset*) *auto*
also have *?this* ⟷ (λ*x*. *fds-nth f x* ∗ *g x* ∗ *mangoldt x* $/_R$ *ln* (*real x*) / *nat-power x s*)
abs-summable-on Collect primepow
by (*intro abs-summable-on-cong*) (*insert primepow-gt-Suc-0*, *auto*)
also have ... ⟷ (λ(*p*,*k*). *fds-nth f* (*p* ^ *Suc k*) ∗ *g* (*p* ^ *Suc k*) ∗ *mangoldt* (*p* ^ *Suc k*)
$/_R$ *ln* (*real* (*p* ^ *Suc k*)) / *nat-power* (*p* ^ *Suc k*) *s*) *abs-summable-on* (*?P* × *UNIV*)
using *bij-betw-primepows* **unfolding** *case-prod-unfold*
by (*intro abs-summable-on-reindex-bij-betw* [*symmetric*])
also have ... ⟷ *?th1*
by (*intro abs-summable-on-cong*)
(*auto simp*: *f.mult f.power mangoldt-def aprimedivisor-prime-power ln-realpow prime-gt-0-nat*
nat-power-power-left divide-simps scaleR-conv-of-real simp del: *power-Suc*)
finally show *?th1* **.**

have *eval-fds h s* = ($\sum_a$*n*. *fds-nth h n* / *nat-power n s*)
using ∗ **unfolding** *eval-fds-def* **by** (*subst infsetsum-nat'*) *auto*
also have ... = ($\sum_a$*n* ∈ {*1*} ∪ {*n*. *primepow n*}.
if n = *1 then l* ∗ *g 1 else fds-nth f n* ∗ *g n* ∗ *mangoldt n* $/_R$ *ln* (*real n*) / *nat-power n s*)
by (*intro infsetsum-cong-neutral*) (*auto simp*: *eq fds-nth-fds mangoldt-def*)
also have ... = *l* ∗ *g 1* + ($\sum_a$*n* | *primepow n*.
if n = *1 then l* ∗ *g 1 else fds-nth f n* ∗ *g n* ∗ *mangoldt n* $/_R$ *ln* (*real n*) / *nat-power n s*)
(**is** - = - + *?x*) **using** *sum1 primepow-gt-Suc-0* **by** (*subst infsetsum-Un-disjoint*) *auto*
also have *?x* =
($\sum_a$*n*∈*Collect primepow*. *fds-nth f n* ∗ *g n* ∗ *mangoldt n* $/_R$ *ln* (*real n*) / *nat-power n s*)
(**is** - = *infsetsum ?f* -) **by** (*intro infsetsum-cong refl*) (*insert primepow-gt-Suc-0*, *auto*)
also have ... = ($\sum_a$(*p*,*k*)∈(*?P* × *UNIV*). *fds-nth f* (*p* ^ *Suc k*) ∗ *g* (*p* ^ *Suc k*) ∗
mangoldt (*p* ^ *Suc k*) $/_R$ *ln* (*p* ^ *Suc k*) / *nat-power* (*p* ^ *Suc k*) *s*)
using *bij-betw-primepows* **unfolding** *case-prod-unfold*
by (*intro infsetsum-reindex-bij-betw* [*symmetric*])
also have ... = ($\sum_a$(*p*,*k*)∈(*?P* × *UNIV*).
(*fds-nth f p* / *nat-power p s*) ^ *Suc k* ∗ *g* (*p* ^ *Suc k*) / *of-nat* (*Suc k*))
by (*intro infsetsum-cong*)
(*auto simp*: *f.mult f.power mangoldt-def aprimedivisor-prime-power ln-realpow prime-gt-0-nat*
nat-power-power-left divide-simps scaleR-conv-of-real simp del: *power-Suc*)
finally show *?th2* **.**
qed

lemma *eval-fds-ln-completely-multiplicative*:
fixes *s* :: *'a* :: *dirichlet-series* **and** *l* :: *'a* **and** *f* :: *'a fds*
assumes *completely-multiplicative-function* (*fds-nth f*) **and** *fds-nth f 1* $\neq$ *0*
assumes *s* · *1* > *abs-conv-abscissa* (*fds-deriv f* / *f*)
shows (λ(*p*,*k*). (*fds-nth f p* / *nat-power p s*) ^ *Suc k* / *of-nat* (*Suc k*))
abs-summable-on ({*p*. *prime p*} × *UNIV*) (**is** *?th1*)
and *eval-fds* (*fds-ln l f*) *s* =
l + ($\sum_a$(*p*, *k*)∈{*p*. *prime p*}×*UNIV*.
(*fds-nth f p* / *nat-power p s*) ^ *Suc k* / *of-nat* (*Suc k*)) (**is** *?th2*)
proof −
from *assms* **have** *fds-abs-converges* (*fds-ln l f*) *s*
by (*intro fds-abs-converges-ln*) (*auto intro*!: *fds-abs-converges-mult intro*: *fds-abs-converges*)
hence *fds-abs-converges* (*fds* (λ*n*. *fds-nth* (*fds-ln l f*) *n* ∗ *1*)) *s*
by *simp*
from *eval-fds-ln-completely-multiplicative-strong* [*OF this assms*(*1*,*2*)] **show** *?th1*
?th2
by *simp-all*
qed

12.8 Exponential and logarithm

lemma *summable-fds-exp-aux*:
assumes *fds-nth f' 1* = (*0* :: *'a* :: *real-normed-algebra-1*)
shows *summable* (λ*k*. *fds-nth* (*f'* ^ *k*) *n* $/_R$ *fact k*)
proof (*rule summable-finite*)
fix *k* **assume** *k* ∉ {..*n*}
hence *n* < *k* **by** *simp*
also have ... < *2* ^ *k*
by (*rule less-exp*)
finally have *fds-nth* (*f'* ^ *k*) *n* = *0*
using *assms* **by** (*intro fds-nth-power-eq-0*) *auto*
thus *fds-nth* (*f'* ^ *k*) *n* $/_R$ *fact k* = *0* **by** *simp*
qed *auto*

lemma
fixes *f* :: *'a* :: *dirichlet-series fds*
assumes *fds-abs-converges f s*
shows *fds-abs-converges-exp*: *fds-abs-converges* (*fds-exp f*) *s*
and *eval-fds-exp*: *eval-fds* (*fds-exp f*) *s* = *exp* (*eval-fds f s*)
proof −
have *conv*: *fds-abs-converges* (*fds-exp f*) *s* **and** *ev*: *eval-fds* (*fds-exp f*) *s* = *exp*
(*eval-fds f s*)
if *fds-abs-converges f s* **and** [*simp*]: *fds-nth f* (*Suc 0*) = *0* **for** *f*
proof −
have [*simp*]: *fds* (λ*n*. *if n* = *Suc 0 then 0 else fds-nth f n*) = *f*
by (*intro fds-eqI*) *simp-all*
have (λ(*k*,*n*). *fds-nth* (*f* ^ *k*) *n* / *fact k* / *nat-power n s*) *abs-summable-on*
(*UNIV* × {*1*..})

proof (*subst abs-summable-on-Sigma-iff*, *safe*, *goal-cases*)
case (*3 k*)
from *that* **have** *fds-abs-converges* (*f* ^ *k*) *s* **by** (*intro fds-abs-converges-power*)
hence (λ*n*. *fds-nth* (*f* ^ *k*) *n* / *nat-power n s* ∗ *inverse* (*fact k*)) *abs-summable-on* {*1*..}
unfolding *fds-abs-converges-altdef* **by** (*intro abs-summable-on-cmult-left*)
thus *?case* **by** (*simp add*: *field-simps*)
next
case *4*
show *?case* **unfolding** *abs-summable-on-nat-iff′*
proof (*rule summable-comparison-test-ev*[*OF always-eventually*[*OF allI*]])
fix *k* :: *nat*
from *that* **have** ∗: *fds-abs-converges* (*fds-norm* (*f* ^ *k*)) (*s* • *1*)
by (*auto simp*: *fds-abs-converges-power*)
have ($\sum_a$*n*∈{*1*..}. *norm* (*fds-nth* (*f* ^ *k*) *n* / *fact k* / *nat-power n s*)) =
($\sum_a$*n*∈{*1*..}. *fds-nth* (*fds-norm* (*f* ^ *k*)) *n* / *nat-power n* (*s* • *1*) / *fact k*)
(**is** *?S* = -) **by** (*intro infsetsum-cong*) (*simp-all add*: *norm-divide norm-mult norm-nat-power*)
also have ... = ($\sum_a$*n*∈{*1*..}. *fds-nth* (*fds-norm* (*f* ^ *k*)) *n* / *nat-power n* (*s* • *1*)) $/_R$ *fact k*
(**is** - = *?S′* $/_R$ -) **using** ∗ **unfolding** *fds-abs-converges-altdef*
by (*subst infsetsum-cdiv*) (*auto simp*: *abs-summable-on-nat-iff scaleR-conv-of-real divide-simps*)
also have *?S′* = *eval-fds* (*fds-norm* (*f* ^ *k*)) (*s* • *1*)
using ∗ **unfolding** *fds-abs-converges-altdef eval-fds-def*
by (*subst infsetsum-nat*) (*auto intro*!: *suminf-cong*)
finally have *eq*: *?S* = ... $/_R$ *fact k* **.**
note *eq*
also have *?S* ≥ *0* **by** (*intro infsetsum-nonneg*) *auto*
hence *?S* = *norm* (*norm ?S*) **by** *simp*
also have *eval-fds* (*fds-norm* (*f* ^ *k*)) (*s* • *1*) ≤ *eval-fds* (*fds-norm f*) (*s* • *1*) ^ *k*
using *that* **by** (*intro eval-fds-norm-power-le*) *auto*
finally show *norm* (*norm* ($\sum_a$*n*∈{*1*..}. *norm* (*fds-nth* (*f* ^ *k*) *n* / *fact k* / *nat-power n s*))) ≤
eval-fds (*fds-norm f*) (*s* • *1*) ^ *k* $/_R$ *fact k*
by (*simp add*: *divide-right-mono*)
next
from *exp-converges*[*of eval-fds* (*fds-norm f*) (*s* • *1*)]
show *summable* (λ*x*. *eval-fds* (*fds-norm f*) (*s* • *1*) ^ *x* $/_R$ *fact x*)
by (*simp add*: *sums-iff*)
qed
qed *auto*
hence *summable*:
(λ(*n*,*k*). *fds-nth* (*f* ^ *k*) *n* / *fact k* / *nat-power n s*) *abs-summable-on* {*1*..} × *UNIV*
by (*subst abs-summable-on-Times-swap*) (*simp add*: *case-prod-unfold*)

have *summable'*: ($\lambda k.$ *fds-nth* (f ^ k) n / *fact* k) *abs-summable-on UNIV* **for** n
using *abs-summable-on-cmult-left*[*of nat-power n s*,
OF abs-summable-on-Sigma-project2 [*OF summable*, *of n*]] **by** (*cases n* = *0*) *simp-all*

have ($\lambda n. \sum_a k.$ *fds-nth* (f ^ k) n / *fact* k / *nat-power* n s) *abs-summable-on* {*1*..}
using *summable* **by** (*rule abs-summable-on-Sigma-project1'*) *auto*
also have *?this* $\longleftrightarrow$ ($\lambda n.$ ($\sum k.$ *fds-nth* (f ^ k) n / *fact* k) $*$ *inverse* (*nat-power* n s))
abs-summable-on {*1*..}
proof (*intro abs-summable-on-cong refl*, *goal-cases*)
case (*1 n*)
hence ($\sum_a k.$ *fds-nth* (f ^ k) n / *fact* k / *nat-power* n s) =
($\sum_a k.$ *fds-nth* (f ^ k) n / *fact* k) $*$ *inverse* (*nat-power* n s)
using *summable'*[*of n*]
by (*subst infsetsum-cmult-left* [*symmetric*]) (*auto simp*: *field-simps*)
also have ($\sum_a k.$ *fds-nth* (f ^ k) n / *fact* k) = ($\sum k.$ *fds-nth* (f ^ k) n / *fact* k)
using *summable'*[*of n*] *1* **by** (*intro abs-summable-on-cong refl infsetsum-nat'*) *auto*
finally show *?case* **.**
qed
finally show *fds-abs-converges* (*fds-exp f*) *s*
by (*simp add*: *fds-exp-def fds-nth-fds' abs-summable-on-Sigma-iff scaleR-conv-of-real*
fds-abs-converges-altdef field-simps)

have *eval-fds* (*fds-exp f*) s = ($\sum n.$ ($\sum k.$ *fds-nth* (f ^ k) n $/_R$ *fact* k) / *nat-power* n s)
by (*simp add*: *fds-exp-def eval-fds-def fds-nth-fds'*)
also have ... = ($\sum n.$ ($\sum_a k.$ *fds-nth* (f ^ k) n $/_R$ *fact* k) / *nat-power* n s)
proof (*intro suminf-cong*, *goal-cases*)
case (*1 n*)
show *?case*
proof (*cases n* = *0*)
case *False*
have ($\sum k.$ *fds-nth* (f ^ k) n $/_R$ *fact* k) = ($\sum_a k.$ *fds-nth* (f ^ k) n $/_R$ *fact* k)
using *summable'*[*of n*] *False*
by (*intro infsetsum-nat'* [*symmetric*]) (*auto simp*: *scaleR-conv-of-real field-simps*)
thus *?thesis* **by** *simp*
qed *simp-all*
qed
also have ... = ($\sum_a n.$ ($\sum_a k.$ *fds-nth* (f ^ k) n $/_R$ *fact* k) / *nat-power* n s)
proof (*intro infsetsum-nat'* [*symmetric*], *goal-cases*)
case *1*
have $*$: *UNIV* $-$ {*Suc 0*..} = {*0*} **by** *auto*

have ($\lambda x.\ \sum_a y.$ *fds-nth* (f ^ y) x / *fact* y / *nat-power* x s) *abs-summable-on* {*1*..}
by (*intro abs-summable-on-Sigma-project1′*[*OF summable*]) *auto*
also have *?this* $\longleftrightarrow$ ($\lambda x.$ ($\sum_a y.$ *fds-nth* (f ^ y) x / *fact* y) $*$ *inverse* (*nat-power* x s))
abs-summable-on {*1*..}
using *summable′* **by** (*intro abs-summable-on-cong refl*, *subst infsetsum-cmult-left* [*symmetric*])
(*auto simp*: *field-simps*)
also have ... $\longleftrightarrow$ ($\lambda x.$ ($\sum_a y.$ *fds-nth* (f ^ y) x $/_R$ *fact* y) / (*nat-power* x s))

abs-summable-on {*1*..} **by** (*simp add*: *field-simps scaleR-conv-of-real*)
finally show *?case* **by** (*rule abs-summable-on-finite-diff*) (*use* $*$ **in** *auto*)
qed
also have ... = ($\sum_a n.$ ($\sum_a k.$ *fds-nth* (f ^ k) n $/_R$ *fact* k $*$ *inverse* (*nat-power* n s)))
using *summable′* **by** (*subst infsetsum-cmult-left*) (*auto simp*: *field-simps scaleR-conv-of-real*)
also have ... = ($\sum_a n{\in}\{1..\}.$ ($\sum_a k.$ *fds-nth* (f ^ k) n $/_R$ *fact* k $*$ *inverse* (*nat-power* n s)))
by (*intro infsetsum-cong-neutral*) (*auto simp*: *Suc-le-eq*)
also have ... = ($\sum_a k.$ $\sum_a n{\in}\{1..\}.$ *fds-nth* (f ^ k) n / *nat-power* n s $/_R$ *fact* k) **using** *summable*
by (*subst infsetsum-swap*) (*auto simp*: *field-simps scaleR-conv-of-real case-prod-unfold*)
also have ... = ($\sum_a k.$ ($\sum_a n{\in}\{1..\}.$ *fds-nth* (f ^ k) n / *nat-power* n s) $/_R$ *fact* k)
by (*subst infsetsum-scaleR-right*) *simp*
also have ... = ($\sum_a k.$ *eval-fds* f s ^ k $/_R$ *fact* k)
proof (*intro infsetsum-cong refl*, *goal-cases*)
case (*1* k)
have $*$: *fds-abs-converges* (f ^ k) s **by** (*intro fds-abs-converges-power that*)
have ($\sum_a n{\in}\{1..\}.$ *fds-nth* (f ^ k) n / *nat-power* n s) =
($\sum_a n.$ *fds-nth* (f ^ k) n / *nat-power* n s)
by (*intro infsetsum-cong-neutral*) (*auto simp*: *Suc-le-eq*)
also have ... = *eval-fds* (f ^ k) s **using** $*$ **unfolding** *eval-fds-def*
by (*intro infsetsum-nat′*) (*auto simp*: *fds-abs-converges-def abs-summable-on-nat-iff′*)
also from *that* **have** ... = *eval-fds* f s ^ k **by** (*simp add*: *eval-fds-power*)
finally show *?case* **by** *simp*
qed
also have ... = ($\sum k.$ *eval-fds* f s ^ k $/_R$ *fact* k)
using *exp-converges*[*of norm* (*eval-fds* f s)]
by (*intro infsetsum-nat′*) (*auto simp*: *abs-summable-on-nat-iff′ sums-iff field-simps norm-power*)
also have ... = *exp* (*eval-fds* f s) **by** (*simp add*: *exp-def*)
finally show *eval-fds* (*fds-exp* f) s = *exp* (*eval-fds* f s) **.**
qed

define f' **where** $f' = f$ − *fds-const* (*fds-nth* f *1*)
have $*$: *fds-abs-converges* (*fds-exp* f') s

```
    by (auto simp: f'-def intro!: fds-abs-converges-diff conv assms)
  have fds-abs-converges (fds-const (exp (fds-nth f 1)) * fds-exp f') s
    unfolding f'-def
    by (intro fds-abs-converges-mult conv fds-abs-converges-diff assms) auto
  thus fds-abs-converges (fds-exp f) s unfolding f'-def
    by (simp add: fds-exp-times-fds-nth-0)
  have eval-fds (fds-exp f) s = eval-fds (fds-const (exp (fds-nth f 1)) * fds-exp f')
s
    by (simp add: f'-def fds-exp-times-fds-nth-0)
  also have ... = exp (fds-nth f (Suc 0)) * eval-fds (fds-exp f') s using *
    using assms by (subst eval-fds-mult) (simp-all)
  also have ... = exp (eval-fds f s) using ev[of f'] assms unfolding f'-def
   by (auto simp: fds-abs-converges-diff eval-fds-diff fds-abs-converges-imp-converges
exp-diff)
  finally show eval-fds (fds-exp f) s = exp (eval-fds f s) .
qed

lemma fds-exp-add:
  fixes f :: 'a :: dirichlet-series fds
  shows   fds-exp (f + g) = fds-exp f * fds-exp g
proof (rule fds-eqI-truncate)
  fix m :: nat assume m: m > 0
  let ?T = fds-truncate m
  have ?T (fds-exp (f + g)) = ?T (fds-exp (?T f + ?T g))
    by (simp add: fds-truncate-exp fds-truncate-add-strong [symmetric])
  also have fds-exp (?T f + ?T g) = fds-exp (?T f) * fds-exp (?T g)
  proof (rule eval-fds-eqD)
    have fds-abs-converges (fds-exp (?T f + ?T g)) 0
      by (intro fds-abs-converges-exp fds-abs-converges-add) auto
    thus conv-abscissa (fds-exp (?T f + ?T g)) < ∞
      using conv-abscissa-PInf-iff by blast
    hence fds-abs-converges (fds-exp (fds-truncate m f) * fds-exp (fds-truncate m
g)) 0
      by (intro fds-abs-converges-mult fds-abs-converges-exp) auto
    thus conv-abscissa (fds-exp (fds-truncate m f) * fds-exp (fds-truncate m g)) <
∞
      using conv-abscissa-PInf-iff by blast
    show frequently (λs. eval-fds (fds-exp (fds-truncate m f + fds-truncate m g)) s
=
                        eval-fds (fds-exp (fds-truncate m f) * fds-exp (fds-truncate m
g)) s)
           ((λs. s · 1) going-to at-top)
      by (auto simp: eval-fds-add eval-fds-mult eval-fds-exp fds-abs-converges-add
                 fds-abs-converges-exp exp-add)
  qed
  also have ?T ... = ?T (fds-exp f * fds-exp g)
    by (subst fds-truncate-mult [symmetric], subst (1 2) fds-truncate-exp)
       (simp add: fds-truncate-mult)
  finally show ?T (fds-exp (f + g)) = ... .
```

qed

lemma *fds-exp-minus*:
  fixes *f* :: *'a* :: *dirichlet-series fds*
  shows *fds-exp* (−*f*) = *inverse* (*fds-exp f*)
proof (*rule fds-right-inverse-unique*)
  have *fds-exp f* ∗ *fds-exp* (− *f*) = *fds-exp* (*f* + (−*f*))
    by (*subst fds-exp-add*) *simp-all*
  also have *f* + (−*f*) = *0* **by** *simp*
  also have *fds-exp* . . . = *1* **by** *simp*
  finally show *fds-exp f* ∗ *fds-exp* (−*f*) = *1* **.**
qed

lemma *abs-conv-abscissa-exp*:
  fixes *f* :: *'a* :: *dirichlet-series fds*
  shows *abs-conv-abscissa* (*fds-exp f*) ≤ *abs-conv-abscissa f*
  by (*intro abs-conv-abscissa-mono fds-abs-converges-exp*)

lemma *fds-deriv-exp* [*simp*]:
  fixes *f* :: *'a* :: *dirichlet-series fds*
  shows *fds-deriv* (*fds-exp f*) = *fds-exp f* ∗ *fds-deriv f*
proof (*rule fds-eqI-truncate*)
  fix *m* :: *nat* **assume** *m*: *m* > *0*
  let *?T* = *fds-truncate m*
  have *abs-conv-abscissa* (*fds-deriv* (*?T f*)) = −∞
    by (*simp add*: *abs-conv-abscissa-deriv*)

  have *?T* (*fds-deriv* (*fds-exp f*)) = *?T* (*fds-deriv* (*fds-exp* (*?T f*)))
    by (*simp add*: *fds-truncate-deriv fds-truncate-exp*)
  also have *fds-deriv* (*fds-exp* (*?T f*)) = *fds-exp* (*?T f*) ∗ *fds-deriv* (*?T f*)
  proof (*rule eval-fds-eqD*)
    note *abscissa* = *conv-le-abs-conv-abscissa abs-conv-abscissa-exp*
    note *abscissa'* = *abscissa*[*THEN le-less-trans*]
    have *fds-abs-converges* (*fds-deriv* (*fds-exp* (*fds-truncate m f*))) *0*
      by (*intro fds-abs-converges*)
      (*auto simp*: *abs-conv-abscissa-deriv intro*: *le-less-trans*[*OF abs-conv-abscissa-exp*])
    thus *conv-abscissa* (*fds-deriv* (*fds-exp* (*fds-truncate m f*))) < ∞
      using *conv-abscissa-PInf-iff* **by** *blast*
    have *fds-abs-converges* (*fds-exp* (*fds-truncate m f*) ∗ *fds-deriv* (*fds-truncate m*
f)) *0*
      by (*intro fds-abs-converges-mult fds-abs-converges-exp*)
        (*auto intro*: *fds-abs-converges simp add*: *fds-truncate-deriv* [*symmetric*])
    thus *conv-abscissa* (*fds-exp* (*fds-truncate m f*) ∗ *fds-deriv* (*fds-truncate m f*))
< ∞
      using *conv-abscissa-PInf-iff* **by** *blast*
    show $\exists_F$ *s in* (λ*s*. *s* · *1*) *going-to at-top*.
        eval-fds (*fds-deriv* (*fds-exp* (*?T f*))) *s* =
        eval-fds (*fds-exp* (*?T f*) ∗ *fds-deriv* (*?T f*)) *s*
    proof (*intro always-eventually eventually-frequently allI*, *goal-cases*)

case (*2 s*)
have *eval-fds* (*fds-deriv* (*fds-exp* (*?T f*))) *s* =
deriv (*eval-fds* (*fds-exp* (*?T f*))) *s*
by (*auto simp*: *eval-fds-exp eval-fds-mult fds-abs-converges-mult fds-abs-converges-exp fds-abs-converges eval-fds-deriv abscissa'*)
also have *eval-fds* (*fds-exp* (*?T f*)) = (*λs. exp* (*eval-fds* (*?T f*) *s*))
by (*intro ext eval-fds-exp*) *auto*
also have *deriv* ... = (*λs. exp* (*eval-fds* (*?T f*) *s*) ∗ *deriv* (*eval-fds* (*?T f*)) *s*)
by (*auto intro*!: *DERIV-imp-deriv derivative-eq-intros simp*: *eval-fds-deriv*)
also have ... = *eval-fds* (*fds-exp* (*?T f*) ∗ *fds-deriv* (*?T f*))
by (*auto simp*: *eval-fds-exp eval-fds-mult fds-abs-converges-mult fds-abs-converges-exp fds-abs-converges eval-fds-deriv abs-conv-abscissa-deriv*)
finally show *?case* **.**
qed *auto*
qed
also have *?T* ... = *?T* (*fds-exp f* ∗ *fds-deriv f*)
by (*subst fds-truncate-mult* [*symmetric*])
(*simp add*: *fds-truncate-exp fds-truncate-deriv* [*symmetric*], *simp add*: *fds-truncate-mult*)
finally show *?T* (*fds-deriv* (*fds-exp f*)) = ... **.**
qed

lemma *fds-exp-ln-strong*:
fixes *f* :: *'a* :: *dirichlet-series fds*
assumes *fds-nth f* (*Suc 0*) ≠ *0*
shows *fds-exp* (*fds-ln l f*) = *fds-const* (*exp l* / *fds-nth f* (*Suc 0*)) ∗ *f*
proof −
let *?c* = *exp l* / *fds-nth f* (*Suc 0*)
have *f* ∗ *fds-const ?c* = *f* ∗ (*fds-exp* (−*fds-ln l f*) ∗ *fds-exp* (*fds-ln l f*)) ∗ *fds-const ?c*
(**is** - = - ∗ (*?g* ∗ *?h*) ∗ -) **by** (*subst fds-exp-add* [*symmetric*]) *simp*
also have ... = *fds-const ?c* ∗ (*f* ∗ *?g*) ∗ *?h* **by** (*simp add*: *mult-ac*)
also have *f* ∗ *?g* = *fds-const* (*inverse ?c*)
proof (*rule fds-deriv-eq-imp-eq*)
have *fds-deriv* (*f* ∗ *fds-exp* (−*fds-ln l f*)) =
fds-exp (− *fds-ln l f*) ∗ *fds-deriv f* ∗ (*1* − *f* / *f*)
by (*simp add*: *divide-fds-def algebra-simps*)
also from *assms* **have** *f* / *f* = *1* **by** (*simp add*: *divide-fds-def fds-right-inverse*)
finally show *fds-deriv* (*f* ∗ *fds-exp* (−*fds-ln l f*)) = *fds-deriv* (*fds-const* (*inverse ?c*))
by *simp*
qed (*insert assms*, *auto simp*: *exp-minus field-simps*)
also have *fds-const ?c* ∗ *fds-const* (*inverse ?c*) = *1*
using *assms* **by** (*subst fds-const-mult* [*symmetric*]) (*simp add*: *divide-simps*)
finally show *?thesis* **by** (*simp add*: *mult-ac*)
qed

lemma *fds-exp-ln* [*simp*]:
fixes *f* :: *'a* :: *dirichlet-series fds*

assumes *exp l = fds-nth f (Suc 0)*
shows *fds-exp (fds-ln l f) = f*
using *assms* **by** *(subst fds-exp-ln-strong) auto*

lemma *fds-ln-exp [simp]*:
fixes *f :: 'a :: dirichlet-series fds*
assumes *l = fds-nth f (Suc 0)*
shows *fds-ln l (fds-exp f) = f*
proof *(rule fds-deriv-eq-imp-eq)*
have *fds-deriv (fds-ln l (fds-exp f)) = fds-deriv f ∗ (fds-exp f / fds-exp f)*
by *(simp add: algebra-simps divide-fds-def)*
also have *fds-exp f / fds-exp f = 1* **by** *(simp add: divide-fds-def fds-right-inverse)*
finally show *fds-deriv (fds-ln l (fds-exp f)) = fds-deriv f* **by** *simp*
qed *(insert assms, auto simp: field-simps)*

12.9 Euler products

lemma *fds-euler-product-LIMSEQ*:
fixes *f :: 'a :: {nat-power, real-normed-field, banach, second-countable-topology} fds*
assumes *multiplicative-function (fds-nth f)* **and** *fds-abs-converges f s*
shows *(λn. ∏p≤n. if prime p then ∑i. fds-nth f (p ^ i) / nat-power (p ^ i) s else 1) ⟶ eval-fds f s*
unfolding *eval-fds-def*
proof *(rule euler-product-LIMSEQ)*
show *multiplicative-function (λn. fds-nth f n / nat-power n s)*
by *(rule multiplicative-function-divide-nat-power) fact+*
qed *(insert assms, auto simp: fds-abs-converges-def)*

lemma *fds-euler-product-LIMSEQ'*:
fixes *f :: 'a :: {nat-power, real-normed-field, banach, second-countable-topology} fds*
assumes *completely-multiplicative-function (fds-nth f)* **and** *fds-abs-converges f s*
shows *(λn. ∏p≤n. if prime p then inverse (1 − fds-nth f p / nat-power p s) else 1) ⟶ eval-fds f s*
unfolding *eval-fds-def*
proof *(rule euler-product-LIMSEQ')*
show *completely-multiplicative-function (λn. fds-nth f n / nat-power n s)*
by *(rule completely-multiplicative-function-divide-nat-power) fact+*
qed *(insert assms, auto simp: fds-abs-converges-def)*

lemma *fds-abs-convergent-euler-product*:
fixes *f :: 'a :: {nat-power, real-normed-field, banach, second-countable-topology} fds*
assumes *multiplicative-function (fds-nth f)* **and** *fds-abs-converges f s*
shows *abs-convergent-prod*
(λp. if prime p then ∑i. fds-nth f (p ^ i) / nat-power (p ^ i) s else 1)

unfolding *eval-fds-def*
proof (*rule abs-convergent-euler-product*)
show *multiplicative-function* (λn. *fds-nth f n* / *nat-power n s*)
by (*rule multiplicative-function-divide-nat-power*) *fact*+
qed (*insert assms*, *auto simp*: *fds-abs-converges-def*)

lemma *fds-abs-convergent-euler-product'*:
fixes *f* :: *'a* :: {*nat-power*, *real-normed-field*, *banach*, *second-countable-topology*} *fds*
assumes *completely-multiplicative-function* (*fds-nth f*) **and** *fds-abs-converges f s*
shows *abs-convergent-prod*
(λp. *if prime p then inverse* (*1* − *fds-nth f p* / *nat-power p s*) *else 1*)
unfolding *eval-fds-def*
proof (*rule abs-convergent-euler-product'*)
show *completely-multiplicative-function* (λn. *fds-nth f n* / *nat-power n s*)
by (*rule completely-multiplicative-function-divide-nat-power*) *fact*+
qed (*insert assms*, *auto simp*: *fds-abs-converges-def*)

lemma *fds-abs-convergent-zero-iff*:
fixes *f* :: *'a* :: {*nat-power-field*, *real-normed-field*, *banach*, *second-countable-topology*} *fds*
assumes *completely-multiplicative-function* (*fds-nth f*)
assumes *fds-abs-converges f s*
shows *eval-fds f s* = *0* $\longleftrightarrow$ ($\exists$ *p*. *prime p* $\wedge$ *fds-nth f p* = *nat-power p s*)
proof −
let *?g* = λp. *if prime p then inverse* (*1* − *fds-nth f p* / *nat-power p s*) *else 1*
have *lim*: (λn. $\prod p \leq n$. *?g p*) $\longrightarrow$ *eval-fds f s*
by (*intro fds-euler-product-LIMSEQ' assms*)
have *conv*: *convergent-prod ?g*
by (*intro abs-convergent-prod-imp-convergent-prod fds-abs-convergent-euler-product' assms*)

{
assume *eval-fds f s* = *0*
from *convergent-prod-to-zero-iff*[*OF conv*] **and** *this* **and** *lim*
have $\exists$ *p*. *prime p* $\wedge$ *fds-nth f p* = *nat-power p s*
by (*auto split*: *if-splits*)
} **moreover** {
assume $\exists$ *p*. *prime p* $\wedge$ *fds-nth f p* = *nat-power p s*
then obtain *p* **where** *prime p fds-nth f p* = *nat-power p s* **by** *blast*
moreover from *this* **have** *nat-power p s* $\neq$ *0*
by (*intro nat-power-nonzero*) (*auto simp*: *prime-gt-0-nat*)
ultimately have (λn. $\prod p \leq n$. *?g p*) $\longrightarrow$ *0*
using *convergent-prod-to-zero-iff*[*OF conv*]
by (*auto intro*!: *exI*[*of - p*] *split*: *if-splits*)
from *tendsto-unique*[*OF - lim this*] **have** *eval-fds f s* = *0*
by *simp*
}
ultimately show *?thesis* **by** *blast*

qed

lemma
fixes *s* :: *'a* :: {*nat-power-normed-field*,*banach*,*euclidean-space*}
assumes *s* · *1* > *1*
shows *euler-product-fds-zeta*:
$(\lambda n. \prod p{\le}n.\ if\ prime\ p\ then\ inverse\ (1 - 1\ /\ nat\text{-}power\ p\ s)\ else\ 1)$
$\longrightarrow$ *eval-fds fds-zeta s* (**is** *?th1*)
and *eval-fds-zeta-nonzero*: *eval-fds fds-zeta s* ≠ *0*
proof −
have ∗: *completely-multiplicative-function* (*fds-nth fds-zeta*)
by *standard auto*
have *lim*: $(\lambda n. \prod p{\le}n.\ if\ prime\ p\ then\ inverse\ (1 - fds\text{-}nth\ fds\text{-}zeta\ p\ /\ nat\text{-}power\ p\ s)\ else\ 1)$
$\longrightarrow$ *eval-fds fds-zeta s* (**is** *filterlim ?g - -*)
using *assms* **by** (*intro fds-euler-product-LIMSEQ'* ∗ *fds-abs-summable-zeta*)
also have *?g* = $(\lambda n. \prod p{\le}n.\ if\ prime\ p\ then\ inverse\ (1 - 1\ /\ nat\text{-}power\ p\ s)\ else\ 1)$
by (*intro ext prod.cong refl*) (*auto simp*: *fds-zeta-def fds-nth-fds*)
finally show *?th1* **.**

{
fix *p* :: *nat* **assume** *prime p*
from *this* **have** *p* > *1* **by** (*simp add*: *prime-gt-Suc-0-nat*)
hence *norm* (*nat-power p s*) = *real p powr* (*s* · *1*)
by (*simp add*: *norm-nat-power*)
also have . . . > *real p powr 0* **using** *assms* **and** ‹*p* > *1*›
by (*intro powr-less-mono*) *auto*
finally have *nat-power p s* ≠ *1*
using ‹*p* > *1*› **by** *auto*
}
hence ∗∗: ∄ *p*. *prime p* ∧ *fds-nth fds-zeta p* = *nat-power p s*
by (*auto simp*: *fds-zeta-def fds-nth-fds*)
show *eval-fds fds-zeta s* ≠ *0*
using *assms* ∗ ∗∗ **by** (*subst fds-abs-convergent-zero-iff*) *simp-all*
qed

lemma *fds-primepow-subseries-euler-product-cm*:
fixes *f* :: *'a* :: *dirichlet-series fds*
assumes *completely-multiplicative-function* (*fds-nth f*) *prime p*
assumes *s* · *1* > *abs-conv-abscissa f*
shows *eval-fds* (*fds-primepow-subseries p f*) *s* = *1* / (*1* − *fds-nth f p* / *nat-power p s*)
proof −
let *?f* = $(\lambda n. \prod pa{\le}n.\ if\ prime\ pa\ then\ inverse\ (1 - fds\text{-}nth\ (fds\text{-}primepow\text{-}subseries\ p\ f)\ pa\ /$
$nat\text{-}power\ pa\ s)\ else\ 1)$
have *sequentially* ≠ *bot* **by** *simp*
moreover have *?f* $\longrightarrow$ *eval-fds* (*fds-primepow-subseries p f*) *s*

by (*intro fds-euler-product-LIMSEQ′ completely-multiplicative-function-only-pows assms*
fds-abs-converges-subseries) (*insert assms*, *auto intro*!: *fds-abs-converges*)
moreover have *eventually* (λ*n*. *?f n* = *1* / (*1* − *fds-nth f p* / *nat-power p s*)) *at-top*
using *eventually-ge-at-top*[*of p*]
proof *eventually-elim*
case (*elim n*)
have (∏ *pa*≤*n*. *if prime pa then inverse* (*1* − *fds-nth* (*fds-primepow-subseries p f*) *pa* /
nat-power pa s) *else 1*) =
(∏ *q*≤*n*. *if q* = *p then inverse* (*1* − *fds-nth f p* / *nat-power p s*) *else 1*)
using ‹*prime p*›
by (*intro prod.cong*) (*auto simp*: *fds-nth-subseries prime-prime-factors*)
also have . . . = *1* / (*1* − *fds-nth f p* / *nat-power p s*)
using *elim* **by** (*subst prod.delta*) (*auto simp*: *divide-simps*)
finally show *?case* **.**
qed
hence *?f* ⟶ *1* / (*1* − *fds-nth f p* / *nat-power p s*) **by** (*rule tendsto-eventually*)
ultimately show *?thesis* **by** (*rule tendsto-unique*)
qed

12.10 Non-negative Dirichlet series

lemma *nonneg-Reals-sum*: (⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ $\mathbb{R}_{\geq 0}$) ⟹ *sum f A* ∈ $\mathbb{R}_{\geq 0}$
by (*induction A rule*: *infinite-finite-induct*) *auto*

locale *nonneg-dirichlet-series* =
fixes *f* :: ′*a* :: *dirichlet-series fds*
assumes *nonneg-coeffs-aux*: *n* > *0* ⟹ *fds-nth f n* ∈ $\mathbb{R}_{\geq 0}$
begin

lemma *nonneg-coeffs*: *fds-nth f n* ∈ $\mathbb{R}_{\geq 0}$
using *nonneg-coeffs-aux*[*of n*] **by** (*cases n* = *0*) *auto*

end

lemma *nonneg-dirichlet-series-0* [*simp*,*intro*]: *nonneg-dirichlet-series 0*
by *standard* (*auto simp*: *zero-fds-def*)

lemma *nonneg-dirichlet-series-1* [*simp*,*intro*]: *nonneg-dirichlet-series 1*
by *standard* (*auto simp*: *one-fds-def*)

lemma *nonneg-dirichlet-series-const* [*simp*,*intro*]:
c ∈ $\mathbb{R}_{\geq 0}$ ⟹ *nonneg-dirichlet-series* (*fds-const c*)
by *standard* (*auto simp*: *fds-const-def*)

lemma *nonneg-dirichlet-series-add* [*intro*]:
assumes *nonneg-dirichlet-series f nonneg-dirichlet-series g*

```
  shows   nonneg-dirichlet-series (f + g)
proof -
  interpret f: nonneg-dirichlet-series f by fact
  interpret g: nonneg-dirichlet-series g by fact
  show ?thesis
    by standard (auto intro!: nonneg-Reals-add-I f.nonneg-coeffs g.nonneg-coeffs)
qed

lemma nonneg-dirichlet-series-mult [intro]:
  assumes nonneg-dirichlet-series f nonneg-dirichlet-series g
  shows   nonneg-dirichlet-series (f * g)
proof -
  interpret f: nonneg-dirichlet-series f by fact
  interpret g: nonneg-dirichlet-series g by fact
  show ?thesis
    by standard (auto intro!: nonneg-Reals-sum nonneg-Reals-mult-I f.nonneg-coeffs
g.nonneg-coeffs
                  simp: fds-nth-mult dirichlet-prod-def)
qed

lemma nonneg-dirichlet-series-power [intro]:
  assumes nonneg-dirichlet-series f
  shows   nonneg-dirichlet-series (f ^ n)
  using assms by (induction n) auto

context nonneg-dirichlet-series
begin

lemma nonneg-exp [intro]: nonneg-dirichlet-series (fds-exp f)
proof
  fix n :: nat assume n > 0
  define c where c = exp (fds-nth f (Suc 0))
  define f' where f' = fds (λn. if n = Suc 0 then 0 else fds-nth f n)
  from nonneg-coeffs[of 1] obtain c' where fds-nth f (Suc 0) = of-real c'
    by (auto elim!: nonneg-Reals-cases)
  hence c = of-real (exp c') by (simp add: c-def exp-of-real)
  hence c: c ∈ ℝ≥0 by simp
  have less: n < 2 ^ k if n < k for k
  proof -
    have n < k by fact
    also have ... < 2 ^ k
      by (rule less-exp)
    finally show ?thesis .
  qed
  have nonneg-power: fds-nth (f' ^ k) n ∈ ℝ≥0 for k
  proof -
    have nonneg-dirichlet-series f'
      by standard (insert nonneg-coeffs, auto simp: f'-def)
    interpret nonneg-dirichlet-series f' ^ k
```

by (*intro nonneg-dirichlet-series-power*) *fact+*
from *nonneg-coeffs*[*of n*] **show** *?thesis* **.**
qed
hence *fds-nth* (*fds-exp f*) $n = c * (\sum k.$ *fds-nth* $(f' \frown k)$ $n /_R$ *fact k*)
by (*simp add*: *fds-exp-def fds-nth-fds' f'-def c-def*)
also have ($\sum k.$ *fds-nth* $(f' \frown k)$ $n /_R$ *fact k*) = ($\sum k \le n.$ *fds-nth* $(f' \frown k)$ $n /_R$ *fact k*)
by (*intro suminf-finite*) (*auto intro*!: *fds-nth-power-eq-0 less simp*: *f'-def not-le*)
also have $c * \ldots \in \mathbb{R}_{\geq 0}$ **unfolding** *scaleR-conv-of-real*
by (*intro nonneg-Reals-mult-I nonneg-Reals-sum nonneg-power*, *unfold nonneg-Reals-of-real-iff*)
(*auto simp*: *c*)
finally show *fds-nth* (*fds-exp f*) $n \in \mathbb{R}_{\geq 0}$ **.**
qed

end

lemma *nonneg-dirichlet-series-lnD*:
assumes *nonneg-dirichlet-series* (*fds-ln l f*) *exp l* = *fds-nth f* (*Suc 0*)
shows *nonneg-dirichlet-series f*
proof −
from *assms* **have** *nonneg-dirichlet-series* (*fds-exp* (*fds-ln l f*))
by (*intro nonneg-dirichlet-series.nonneg-exp*)
thus *?thesis* **using** *assms* **by** *simp*
qed

context *nonneg-dirichlet-series*
begin

lemma *fds-of-real-norm*: *fds-of-real* (*fds-norm f*) = *f*
proof (*rule fds-eqI*)
fix *n* :: *nat* **assume** *n*: $n > 0$
show *fds-nth* (*fds-of-real* (*fds-norm f*)) *n* = *fds-nth f n*
using *nonneg-coeffs*[*of n*] **by** (*auto elim*!: *nonneg-Reals-cases*)
qed

end

lemma *pringsheim-landau-aux*:
fixes *c* :: *real* **and** *f* :: *complex fds*
assumes *nonneg-dirichlet-series f*
assumes *abscissa*: $c \geq$ *abs-conv-abscissa f*
assumes *g*: $\bigwedge s.\ s \in A \Longrightarrow Re\ s > c \Longrightarrow g\ s =$ *eval-fds f s*
assumes *g holomorphic-on A open A* $c \in A$
shows $\exists x.\ x < c \wedge$ *fds-abs-converges f* (*of-real x*)
proof −
interpret *nonneg-dirichlet-series f* **by** *fact*
define *a* **where** $a = 1 + c$

define g' **where** $g' = (\lambda s.\ if\ s \in \{s.\ Re\ s > c\}\ then\ eval\text{-}fds\ f\ s\ else\ g\ s)$

— We can find some $\varepsilon > 0$ such that the Dirichlet series can be continued analytically in a ball of radius $1 + \varepsilon$ around a.

from ‹*open A*› ‹$c \in A$› **obtain** δ **where** δ: $\delta > 0$ *ball* $c\ \delta \subseteq A$
  by (*auto simp*: *open-contains-ball*)
define ε **where** $\varepsilon = sqrt\ (1 + \delta\hat{}2) - 1$
from δ **have** ε: $\varepsilon > 0$ **by** (*simp add*: ε*-def*)

have *ball-a-subset*: *ball* $a\ (1 + \varepsilon) \subseteq \{s.\ Re\ s > c\} \cup A$
proof (*intro subsetI*)
  fix s :: *complex* **assume** s: $s \in$ *ball* $a\ (1 + \varepsilon)$
  define $x\ y$ **where** $x = Re\ s$ **and** $y = Im\ s$
  have [*simp*]: $s = x + \mathrm{i} * y$ **by** (*simp add*: *complex-eq-iff x-def y-def*)
  show $s \in \{s.\ Re\ s > c\} \cup A$
  proof (*cases* $Re\ s \leq c$)
    case *True*
    hence $(c - x)^2 + y^2 \leq (1 + c - x)^2 + y^2 - 1$
      by (*simp add*: *power2-eq-square algebra-simps*)
    also from s **have** $(1 + c - x)^2 + y^2 - 1 < \delta^2$
      by (*auto simp*: *dist-norm cmod-def a-def* ε*-def*)
    finally have $sqrt\ ((c - x)^2 + y^2) < \delta$ **using** δ
      by (*intro real-less-lsqrt*) *auto*
    hence $s \in$ *ball* $c\ \delta$ **by** (*auto simp*: *dist-norm cmod-def*)
    also have $\ldots \subseteq A$ **by** *fact*
    finally show *?thesis* **..**
  qed *auto*
qed

have *holo*: g' *holomorphic-on ball* $a\ (1 + \varepsilon)$ **unfolding** g'*-def*
proof (*intro holomorphic-on-subset*[*OF - ball-a-subset*] *holomorphic-on-If-Un*)
  have *conv-abscissa f* $\leq$ *abs-conv-abscissa f* **by** (*rule conv-le-abs-conv-abscissa*)
  also have $\ldots \leq$ *ereal c* **by** *fact*
  finally have$*$: *conv-abscissa f* $\leq$ *ereal c* **.**
  show *eval-fds f holomorphic-on* $\{s.\ c < Re\ s\}$
    by (*intro holomorphic-intros*) (*auto intro*: *le-less-trans*[*OF* $*$])
qed (*insert assms*, *auto intro*!: *holomorphic-intros open-halfspace-Re-gt*)

define f' **where** $f' = fds\text{-}norm\ f$
have *f-f'*: $f = fds\text{-}of\text{-}real\ f'$ **by** (*simp add*: f'*-def fds-of-real-norm*)
have f'*-nonneg*: $fds\text{-}nth\ f'\ n \geq 0$ **for** n
  using *nonneg-coeffs*[*of n*] **by** (*auto elim*!: *nonneg-Reals-cases simp*: f'*-def*)

have *deriv*: $(\lambda n.\ (deriv\ \hat{}\hat{}\ n)\ g'\ a) = (\lambda n.\ eval\text{-}fds\ ((fds\text{-}deriv\ \hat{}\hat{}\ n)\ f)\ a)$
proof
  fix n :: *nat*
  have *ev*: *eventually* $(\lambda s.\ s \in \{s.\ Re\ s > c\})$ (*nhds* (*complex-of-real a*))
    by (*intro eventually-nhds-in-open open-halfspace-Re-gt*) (*auto simp*: *a-def*)

have (*deriv* ^^ *n*) *g′ a* = (*deriv* ^^ *n*) (*eval-fds f*) *a*
by (*intro higher-deriv-cong-ev refl eventually-mono*[*OF ev*]) (*auto simp*: *g′-def*)
also have ... = *eval-fds* ((*fds-deriv* ^^ *n*) *f*) *a*
proof (*intro eval-fds-higher-deriv* [*symmetric*])
have *conv-abscissa f* ≤ *abs-conv-abscissa f* **by** (*rule conv-le-abs-conv-abscissa*)
also have ... ≤ *ereal c* **by** (*rule assms*)
also have ... < *a* **by** (*simp add*: *a-def*)
finally show *conv-abscissa f* < *ereal* (*complex-of-real a* · *1*) **by** *simp*
qed
finally show (*deriv* ^^ *n*) *g′ a* = *eval-fds* ((*fds-deriv* ^^ *n*) *f*) *a* **.**
qed

have *nth-deriv-conv*: *fds-abs-converges* ((*fds-deriv* ^^ *n*) *f*) (*of-real a*) **for** *n*
by (*intro fds-abs-converges*)
(*auto simp*: *abs-conv-abscissa-higher-deriv a-def intro*!: *le-less-trans*[*OF abscissa*])

have *nth-deriv-eq*: (*fds-deriv* ^^ *n*) *f* = *fds* (λ*k*. (−*1*) ^ *n* ∗ *fds-nth f k* ∗ *ln* (*real k*) ^ *n*) **for** *n*
proof −
have *fds-nth* ((*fds-deriv* ^^ *n*) *f*) *k* = (−*1*) ^ *n* ∗ *fds-nth f k* ∗ *ln* (*real k*) ^ *n* **for** *k*
by (*induction n*) (*simp-all add*: *fds-deriv-def fds-eq-iff fds-nth-fds′ scaleR-conv-of-real*)
thus *?thesis* **by** (*intro fds-eqI*) *simp-all*
qed

have *deriv′*: (λ*n*. *eval-fds* ((*fds-deriv* ^^ *n*) *f*) (*complex-of-real a*)) =
(λ*n*. (− *1*) ^ *n* ∗ *complex-of-real* (∑$_a$*k*. *fds-nth f′ k* ∗ *ln* (*real k*) ^ *n* / *real k powr a*))
proof
fix *n*
have *eval-fds* ((*fds-deriv* ^^ *n*) *f*) (*of-real a*) =
(∑$_a$*k*. *fds-nth* ((*fds-deriv* ^^ *n*) *f*) *k* / *of-nat k powr complex-of-real a*)
using *nth-deriv-conv* **by** (*subst eval-fds-altdef*) *auto*
hence *eval-fds* ((*fds-deriv* ^^ *n*) *f*) (*of-real a*) =
(∑$_a$*k*. (− *1*) ^ *n* ∗$_R$ (*fds-nth f k* ∗ *ln* (*real k*) ^ *n* / *k powr a*))
by (*simp add*: *nth-deriv-eq fds-nth-fds′ powr-Reals-eq scaleR-conv-of-real algebra-simps*)
also have ... = (− *1*) ^ *n* ∗ (∑$_a$*k*. *of-real* (*fds-nth f′ k* ∗ *ln* (*real k*) ^ *n* / *k powr a*))
by (*subst infsetsum-scaleR-right*) (*simp-all add*: *scaleR-conv-of-real f-f′*)
also have ... = (− *1*) ^ *n* ∗ *of-real* (∑$_a$*k*. *fds-nth f′ k* ∗ *ln* (*real k*) ^ *n* / *k powr a*)
by (*subst infsetsum-of-real*) (*rule refl*)
finally show *eval-fds* ((*fds-deriv* ^^ *n*) *f*) (*complex-of-real a*) =
(− *1*) ^ *n* ∗ *complex-of-real* (∑$_a$*k*. *fds-nth f′ k* ∗ *ln* (*real k*) ^ *n* / *real k powr a*) **.**
qed

define *s* :: *complex* **where** *s* = *c* − *ε* / *2*
have *s*: *Re s* < *c* **using** *assms δ* **by** (*simp-all add*: *s-def ε-def field-simps*)
have *s* ∈ *ball a* (*1* + *ε*) **using** *s* **by** (*simp add*: *a-def dist-norm cmod-def s-def*)
from *holomorphic-power-series*[*OF holo this*]
have *sums*: (*λn.* (*deriv* ^^ *n*) *g′ a* / *fact n* ∗ (*s* − *a*) ^ *n*) *sums g′ s* **by** *simp*
also note *deriv*
also have *s* − *a* = −*of-real* (*1* + *ε* / *2*) **by** (*simp add*: *s-def a-def*)
also have (*λn.* ... ^ *n*) = (*λn. of-real* ((−*1*) ^ *n* ∗ (*1* + *ε* / *2*) ^ *n*))
by (*intro ext*) (*subst power-minus*, *auto*)
also have (*λn. eval-fds* ((*fds-deriv* ^^ *n*) *f*) *a* / *fact n* ∗ ... *n*) =
(*λn. of-real* ((−*1*) ^ *n* ∗ *eval-fds* ((*fds-deriv* ^^ *n*) *f′*) *a* / *fact n* ∗ (*1*+*ε*/*2*) ^ *n*))
using *nth-deriv-conv* **by** (*simp add*: *f-f′ fds-abs-converges-imp-converges mult-ac*)
finally have *summable* ... **by** (*simp add*: *sums-iff*)
hence *summable*: *summable* (*λn.* (−*1*)^*n* ∗ *eval-fds* ((*fds-deriv* ^^ *n*) *f′*) *a* / *fact n* ∗ (*1*+*ε*/*2*)^*n*)
by (*subst* (*asm*) *summable-of-real-iff*)

have (*λ*(*n*,*k*). (−*1*)^*n* ∗ *fds-nth f k* ∗ *ln* (*real k*) ^ *n* / (*real k powr a*) ∗ ((*s*−*a*) ^ *n* / *fact n*))
abs-summable-on (*UNIV* × *UNIV*)
proof (*subst abs-summable-on-Sigma-iff*, *safe*, *goal-cases*)
case (*3 n*)
from *nth-deriv-conv*[*of n*] **show** *?case*
unfolding *fds-abs-converges-altdef′*
by (*intro abs-summable-on-cmult-left*) (*simp add*: *nth-deriv-eq fds-nth-fds′ powr-Reals-eq*)
next
case *4*
have *nth-deriv-f-f′*: (*fds-deriv* ^^ *n*) *f* = *fds-of-real* ((*fds-deriv* ^^ *n*) *f′*) **for** *n*
by (*induction n*) (*auto simp*: *f′-def fds-of-real-norm*)
have *norm-nth-deriv-f*: *norm* (*fds-nth* ((*fds-deriv* ^^ *n*) *f*) *k*) =
(−*1*) ^ *n* ∗ *of-real* (*fds-nth* ((*fds-deriv* ^^ *n*) *f′*) *k*) **for** *n k*
proof (*induction n*)
case (*Suc n*)
thus *?case* **by** (*cases k*) (*auto simp*: *f-f′ fds-nth-deriv scaleR-conv-of-real norm-mult*)
qed (*auto simp*: *f′-nonneg f-f′*)

note *summable*
also have (*λn.* (−*1*)^*n* ∗ *eval-fds* ((*fds-deriv* ^^ *n*) *f′*) *a* / *fact n* ∗ (*1*+*ε*/*2*)^*n*) =
(*λn.* ∑*a k. norm* ((− *1*) ^ *n* ∗ *fds-nth f k* ∗ *ln* (*real k*) ^ *n* /
(*real k powr a*) ∗ ((*s* − *a*) ^ *n* / *fact n*))) (**is** - = *?h*)
proof (*rule ext*, *goal-cases*)
case (*1 n*)
have (∑*a k. norm* ((− *1*) ^ *n* ∗ *fds-nth f k* ∗ *ln* (*real k*) ^ *n* /

(real k powr a) ∗ ((s − a) ^ n / fact n))) =
(norm ((s − a) ^ n / fact n) ∗ (−1) ^ n) $*_R$
($\sum_a$k. (−1) ^ n ∗ norm (fds-nth ((fds-deriv ^^ n) f) k / real k powr a)) (**is** - = - $*_R$ *?S*)
by *(subst infsetsum-scaleR-right [symmetric])*
(auto simp: norm-mult norm-divide norm-power mult-ac nth-deriv-eq fds-nth-fds′)
also have *?S = ($\sum_a$k. fds-nth ((fds-deriv ^^ n) f′) k / real k powr a)*
by *(intro infsetsum-cong) (auto simp: norm-mult norm-divide norm-power norm-nth-deriv-f)*
also have *. . . = eval-fds ((fds-deriv ^^ n) f′) a*
using *nth-deriv-conv[of n]* **by** *(subst eval-fds-altdef) (auto simp: f′-def nth-deriv-f-f′)*
also have *(norm ((s − a) ^ n / fact n) ∗ (− 1) ^ n) $*_R$ eval-fds ((fds-deriv ^^ n) f′) a =*
(−1) ^ n ∗ eval-fds ((fds-deriv ^^ n) f′) a / fact n ∗ norm (s − a) ^ n
by *(simp add: norm-divide norm-power)*
also have *s-a: s − a = −of-real (1 + ε / 2)* **by** *(simp add: s-def a-def)*
have *norm (s − a) = 1 + ε / 2* **unfolding** *s-a norm-minus-cancel norm-of-real* **using** *ε* **by** *simp*
finally show *?case* **..**
qed
also have *?h n ≥ 0* **for** *n* **by** *(intro infsetsum-nonneg) auto*
hence *?h = (λn. norm (?h n))* **by** *simp*
finally show *?case* **unfolding** *abs-summable-on-nat-iff′* **.**
qed *auto*
hence *(λ(k,n). (−1)^n ∗ fds-nth f k ∗ ln (real k) ^ n / (real k powr a) ∗ ((s−a) ^ n / fact n))*
abs-summable-on (UNIV × UNIV)
by *(subst (asm) abs-summable-on-Times-swap) (simp add: case-prod-unfold)*
hence *(λk. $\sum_a$n. (− 1) ^ n ∗ fds-nth f k ∗ ln (real k) ^ n / (k powr a) ∗*
((s − a) ^ n / fact n)) abs-summable-on UNIV (**is** *?h abs-summable-on* -)
by *(rule abs-summable-on-Sigma-project1′) auto*
also have *?this ⟷ (λk. fds-nth f k / nat-power k s) abs-summable-on UNIV*
proof *(intro abs-summable-on-cong refl, goal-cases)*
case *(1 k)*
have *?h k = (fds-nth f′ k / k powr a) $*_R$ ($\sum_a$n. (−ln (real k) ∗ (s − a)) ^ n / fact n)*
by *(subst infsetsum-scaleR-right [symmetric], rule infsetsum-cong)*
(simp-all add: scaleR-conv-of-real f-f′ power-minus′ power-mult-distrib divide-simps)
also have *($\sum_a$n. (−ln (real k) ∗ (s − a)) ^ n / fact n) = exp (−ln (real k) ∗ (s − a))*
using *exp-converges[of −ln k ∗ (s − a)] exp-converges[of norm (−ln k ∗ (s − a))]*
by *(subst infsetsum-nat′) (auto simp: abs-summable-on-nat-iff′ sums-iff scaleR-conv-of-real*
divide-simps norm-divide norm-mult norm-power)

```
    also have (fds-nth f′ k / k powr a) *R ... = fds-nth f k / nat-power k s
      by (auto simp: scaleR-conv-of-real f-f′ powr-def exp-minus
                field-simps exp-of-real [symmetric] exp-diff)
    finally show ?case .
  qed
  finally have fds-abs-converges f s
    by (simp add: fds-abs-converges-def abs-summable-on-nat-iff′)
  thus ?thesis by (intro exI[of - (c − ε / 2)]) (auto simp: s-def a-def ε)
qed

theorem pringsheim-landau:
  fixes c :: real and f :: complex fds
  assumes nonneg-dirichlet-series f
  assumes abscissa: abs-conv-abscissa f = c
  assumes g: ⋀s. s ∈ A ⟹ Re s > c ⟹ g s = eval-fds f s
  assumes g holomorphic-on A open A c ∈ A
  shows   False
proof −
  have ∃x<c. fds-abs-converges f (complex-of-real x)
    by (rule pringsheim-landau-aux[where g = g and A = A]) (insert assms, auto)
  then obtain x where x: x < c fds-abs-converges f (complex-of-real x) by blast
  hence abs-conv-abscissa f ≤ complex-of-real x · 1
    unfolding abs-conv-abscissa-def
    by (intro Inf-lower) (auto simp: image-iff intro!: exI[of - of-real x])
  also have ... < abs-conv-abscissa f using assms x by simp
  finally show False by simp
qed

corollary entire-continuation-imp-abs-conv-abscissa-MInfty:
  assumes nonneg-dirichlet-series f
  assumes c: c ≥ abs-conv-abscissa f
  assumes g: ⋀s. Re s > c ⟹ g s = eval-fds f s
  assumes holo: g holomorphic-on UNIV
  shows   abs-conv-abscissa f = −∞
proof (rule ccontr)
  assume abs-conv-abscissa f ≠ −∞
  with c obtain a where abscissa [simp]: abs-conv-abscissa f = ereal a
    by (cases abs-conv-abscissa f) auto
  show False
  proof (rule pringsheim-landau[OF assms(1) abscissa - holo])
    fix s assume s: Re s > a
    show g s = eval-fds f s
    proof (rule sym, rule analytic-continuation-open[of - - - g])
      show g holomorphic-on {s. Re s > a} by (rule holomorphic-on-subset[OF
holo]) auto
      from assms show {s. Re s > c} ⊆ {s. Re s > a} by auto
    next
      have conv-abscissa f ≤ abs-conv-abscissa f by (rule conv-le-abs-conv-abscissa)
      also have ... = ereal a by simp
```

finally show *eval-fds f holomorphic-on* {*s*. *Re s* > *a*}
    by (*intro holomorphic-intros*) (*auto intro*: *le-less-trans*)
  qed (*insert assms s*, *auto intro*!: *exI*[*of - of-real* (*c* + *1*)]
      open-halfspace-Re-gt convex-connected convex-halfspace-Re-gt)
 qed *auto*
qed

12.11 Convergence of the ζ and Möbius μ series

lemma *fds-abs-summable-zeta-real-iff* [*simp*]:
  fds-abs-converges fds-zeta s ⟷ *s* > (*1* :: *real*)
proof −
  have *fds-abs-converges fds-zeta s* ⟷ *summable* (*λn*. *real n powr* −*s*)
    unfolding *fds-abs-converges-def*
    by (*intro summable-cong always-eventually*)
      (*auto simp*: *fds-nth-zeta powr-minus divide-simps*)
  also have ... ⟷ *s* > *1* **by** (*simp add*: *summable-real-powr-iff*)
  finally show *?thesis* **.**
qed

lemma *fds-abs-summable-zeta-real*: *s* > (*1* :: *real*) ⟹ *fds-abs-converges fds-zeta s*
  by *simp*

lemma *fds-abs-converges-moebius-mu-real*:
  assumes *s* > (*1* :: *real*)
  shows *fds-abs-converges* (*fds moebius-mu*) *s*
  unfolding *fds-abs-converges-def*
proof (*rule summable-comparison-test*, *intro exI allI impI*)
  fix *n* :: *nat*
  show *norm* (*norm* (*fds-nth* (*fds moebius-mu*) *n* / *nat-power n s*)) ≤ *n powr* (−*s*)
    by (*simp add*: *powr-minus divide-simps abs-moebius-mu-le*)
next
  from *assms* **show** *summable* (*λn*. *real n powr* −*s*) **by** (*simp add*: *summable-real-powr-iff*)
qed

12.12 Application to the Möbius μ function

lemma *inverse-squares-sums'*: (*λn*. *1* / *real n* ^ *2*) *sums* (*pi* ^ *2* / *6*)
  using *inverse-squares-sums sums-Suc-iff*[*of λn*. *1* / *real n* ^ *2 pi*^*2* / *6*] **by** *simp*

lemma *norm-summable-moebius-over-square*:
  summable (*λn*. *norm* (*moebius-mu n* / *real n* ^ *2*))
proof (*subst summable-Suc-iff* [*symmetric*], *rule summable-comparison-test*)
  show *summable* (*λn*. *1* / *real* (*Suc n*) ^ *2*)
    using *inverse-squares-sums* **by** (*simp add*: *sums-iff*)
qed (*auto simp del*: *of-nat-Suc simp*: *field-simps abs-moebius-mu-le*)

lemma *summable-moebius-over-square*:
  summable (*λn*. *moebius-mu n* / *real n* ^ *2*)

using *norm-summable-moebius-over-square* **by** (*rule summable-norm-cancel*)

lemma *moebius-over-square-sums*: ($\lambda n.$ *moebius-mu n* / n^2) *sums* (*6* / pi^2)
proof −
have *1* = *eval-fds* (*1* :: *real fds*) *2* **by** *simp*
also have (*1* :: *real fds*) = *fds-zeta* ∗ *fds moebius-mu*
by (*rule fds-zeta-times-moebius-mu* [*symmetric*])
also have *eval-fds* . . . *2* = *eval-fds fds-zeta 2* ∗ *eval-fds* (*fds moebius-mu*) *2*
by (*intro eval-fds-mult fds-abs-converges-moebius-mu-real*) *simp-all*
also have . . . = *pi* ^ *2* / *6* ∗ ($\sum$ *n. moebius-mu n* / (*real n*)^2)
using *inverse-squares-sums′* **by** (*simp add*: *eval-fds-at-numeral suminf-fds-zeta-aux sums-iff*)
finally have ($\sum$ *n. moebius-mu n* / (*real n*)^2) = *6* / *pi* ^ *2* **by** (*simp add*: *field-simps*)
with *summable-moebius-over-square* **show** *?thesis* **by** (*simp add*: *sums-iff*)
qed

end

13 Asymptotics of summatory arithmetic functions

theory *Arithmetic-Summatory-Asymptotics*
imports
Euler-MacLaurin.Euler-MacLaurin-Landau
Arithmetic-Summatory
Dirichlet-Series-Analysis
Landau-Symbols.Landau-More
begin

13.1 Auxiliary bounds

lemma *sum-inverse-squares-tail-bound*:
assumes *d* > *0*
shows *summable* ($\lambda n.$ *1* / (*real* (*Suc n*) + *d*) ^ *2*)
($\sum$ *n. 1* / (*real* (*Suc n*) + *d*) ^ *2*) ≤ *1* / *d*
proof −
show ∗: *summable* ($\lambda n.$ *1* / (*real* (*Suc n*) + *d*) ^ *2*)
proof (*rule summable-comparison-test*, *intro allI exI impI*)
fix *n* :: *nat*
from *assms* **show** *norm* (*1* / (*real* (*Suc n*) + *d*) ^ *2*) ≤ *1* / *real* (*Suc n*) ^ *2*
unfolding *norm-divide norm-one norm-power*
by (*intro divide-left-mono power-mono*) *simp-all*
qed (*insert inverse-squares-sums*, *simp add*: *sums-iff*)
show ($\sum$ *n. 1* / (*real* (*Suc n*) + *d*) ^ *2*) ≤ *1* / *d*
proof (*rule sums-le*)
fix *n* **have** *1* / (*real* (*Suc n*) + *d*) ^ *2* ≤ *1* / ((*real n* + *d*) ∗ (*real* (*Suc n*) + *d*))
unfolding *power2-eq-square* **using** *assms*
by (*intro divide-left-mono mult-mono mult-pos-pos add-nonneg-pos*) *simp-all*

also have ... = *1* / (*real n* + *d*) − *1* / (*real* (*Suc n*) + *d*)
using *assms* **by** (*simp add*: *divide-simps*)
finally show *1* / (*real* (*Suc n*) + *d*)^2 ≤ *1* / (*real n* + *d*) − *1* / (*real* (*Suc n*) + *d*) **.**
next
show (λ*n*. *1* / (*real* (*Suc n*) + *d*)^2) *sums* (∑*n*. *1* / (*real* (*Suc n*) + *d*)^2)
using ∗ **by** (*simp add*: *sums-iff*)
next
have (λ*n*. *1* / (*real n* + *d*) − *1* / (*real* (*Suc n*) + *d*)) *sums* (*1* / (*real 0* + *d*) − *0*)
by (*intro telescope-sums′ real-tendsto-divide-at-top*[*OF tendsto-const*],
subst add.commute, *rule filterlim-tendsto-add-at-top*[*OF tendsto-const*
filterlim-real-sequentially])
thus (λ*n*. *1* / (*real n* + *d*) − *1* / (*real* (*Suc n*) + *d*)) *sums* (*1* / *d*) **by** *simp*
qed
qed

lemma *moebius-sum-tail-bound*:
assumes *d* > *0*
shows *abs* (∑*n*. *moebius-mu* (*Suc n* + *d*) / *real* (*Suc n* + *d*)^*2*) ≤ *1* / *d* (**is** *abs ?S* ≤ -)
proof −
have ∗: *summable* (λ*n*. *1* / (*real* (*Suc n* + *d*))^2)
by (*insert sum-inverse-squares-tail-bound*(*1*)[*of real d*] *assms*, *simp-all add*: *add-ac*)
have ∗∗: *summable* (λ*n*. *abs* (*moebius-mu* (*Suc n* + *d*) / *real* (*Suc n* + *d*)^*2*))
proof (*rule summable-comparison-test*, *intro exI allI impI*)
fix *n* :: *nat*
show *norm* (|*moebius-mu* (*Suc n* + *d*) / (*real* (*Suc n* + *d*))^*2*|) ≤
1 / (*real* (*Suc n* + *d*))^*2*
unfolding *real-norm-def abs-abs abs-divide power-abs abs-of-nat*
by (*intro divide-right-mono abs-moebius-mu-le*) *simp-all*
qed (*insert* ∗)
from ∗∗ **have** *abs ?S* ≤ (∑*n*. *abs* (*moebius-mu* (*Suc n* + *d*) / *real* (*Suc n* + *d*)^*2*))
by (*rule summable-rabs*)
also have ... ≤ (∑*n*. *1* / (*real* (*Suc n*) + *d*) ^ *2*)
proof (*intro suminf-le allI*)
fix *n* :: *nat*
show *abs* (*moebius-mu* (*Suc n* + *d*) / (*real* (*Suc n* + *d*))^*2*) ≤ *1* / (*real* (*Suc n*) + *real d*)^*2*
unfolding *abs-divide abs-of-nat power-abs of-nat-add* [*symmetric*]
by (*intro divide-right-mono abs-moebius-mu-le*) *simp-all*
qed (*insert* ∗ ∗∗, *simp-all add*: *add-ac*)
also from *assms* **have** ... ≤ *1* / *d* **by** (*intro sum-inverse-squares-tail-bound*) *simp-all*
finally show *?thesis* **.**
qed

lemma *sum-upto-inverse-bound*:
  sum-upto (λ*i*. *1* / *real i*) *x* ≥ *0*
  eventually (λ*x*. *sum-upto* (λ*i*. *1* / *real i*) *x* ≤ *ln x* + *13* / *22*) *at-top*
proof −
  show *sum-upto* (λ*i*. *1* / *real i*) *x* ≥ *0*
    by (*simp add*: *sum-upto-def sum-nonneg*)
  from *order-tendstoD*(*2*)[*OF euler-mascheroni-LIMSEQ euler-mascheroni-less-13-over-22*]
  obtain *N* **where** *N*: ⋀*n*. *n* ≥ *N* ⟹ *harm n* − *ln* (*real n*) < *13* / *22*
    unfolding *eventually-at-top-linorder* **by** *blast*
  show *eventually* (λ*x*. *sum-upto* (λ*i*. *1* / *real i*) *x* ≤ *ln x* + *13* / *22*) *at-top*
    using *eventually-ge-at-top*[*of max* (*real N*) *1*]
  proof *eventually-elim*
    case (*elim x*)
    have *sum-upto* (λ*i*. *1* / *real i*) *x* = (∑*i*∈{*0*<..*nat* ⌊*x*⌋}. *1* / *real i*)
      by (*simp add*: *sum-upto-altdef*)
    also have ... = *harm* (*nat* ⌊*x*⌋)
      unfolding *harm-def* **by** (*intro sum.cong refl*) (*auto simp*: *field-simps*)
    also have ... ≤ *ln* (*real* (*nat* ⌊*x*⌋)) + *13* / *22*
      using *N*[*of nat* ⌊*x*⌋] *elim* **by** (*auto simp*: *le-nat-iff le-floor-iff*)
    also have *ln* (*real* (*nat* ⌊*x*⌋)) ≤ *ln x* **using** *elim* **by** (*subst ln-le-cancel-iff*) *auto*
    finally show *?case* **by** − *simp*
  qed
qed

lemma *sum-upto-inverse-bigo*: *sum-upto* (λ*i*. *1* / *real i*) ∈ *O*(λ*x*. *ln x*)
proof −
  have *eventually* (λ*x*. *norm* (*sum-upto* (λ*i*. *1* / *real i*) *x*) ≤ *1* ∗ *norm* (*ln x* + *13*/*22*)) *at-top*
    using *eventually-ge-at-top*[*of 1*::*real*] *sum-upto-inverse-bound*(*2*)
    by *eventually-elim* (*insert sum-upto-inverse-bound*(*1*), *simp-all*)
  hence *sum-upto* (λ*i*. *1* / *real i*) ∈ *O*(λ*x*. *ln x* + *13*/*22*)
    by (*rule bigoI*)
  also have (λ*x*::*real*. *ln x* + *13*/*22*) ∈ *O*(λ*x*. *ln x*) **by** *simp*
  finally show *?thesis* **.**
qed

lemma
  defines *G* ≡ (λ*x*::*real*. (∑*n*. *moebius-mu* (*n* + *Suc* (*nat* ⌊*x*⌋)) / (*n* + *Suc* (*nat* ⌊*x*⌋))^*2*) :: *real*)
  shows *moebius-sum-tail-bound*′: ⋀*t*. *t* ≥ *2* ⟹ |*G t*| ≤ *1* / (*t* − *1*)
    and *moebius-sum-tail-bigo*: *G* ∈ *O*(λ*t*. *1* / *t*)
proof −
  show |*G t*| ≤ *1* / (*t* − *1*) **if** *t*: *t* ≥ *2* **for** *t*
  proof −
    from *t* **have** |*G t*| ≤ *1* / *real* (*nat* ⌊*t*⌋)
      unfolding *G-def* **using** *moebius-sum-tail-bound*[*of nat* ⌊*t*⌋] **by** *simp*
    also have *t* ≤ *1* + *real-of-int* ⌊*t*⌋ **by** *linarith*
    hence *1* / *real* (*nat* ⌊*t*⌋) ≤ *1* / (*t* − *1*) **using** *t* **by** (*simp add*: *field-simps*)
    finally show *?thesis* **.**

qed
hence *G* ∈ *O*(λ*t*. *1* / (*t* − *1*))
by (*intro bigoI*[*of* - *1*] *eventually-mono*[*OF eventually-ge-at-top*[*of 2*::*real*]]) *auto*
also have (λ*t*::*real*. *1* / (*t* − *1*)) ∈ Θ(λ*t*. *1* / *t*) **by** *simp*
finally show *G* ∈ *O*(λ*t*. *1* / *t*) **.**
qed

13.2 Summatory totient function

theorem *summatory-totient-asymptotics*:
(λ*x*. *sum-upto* (λ*n*. *real* (*totient n*)) *x* − *3* / *pi*^{2} ∗ *x*^{2}) ∈ *O*(λ*x*. *x* ∗ *ln x*)
proof −
define *H* **where** *H* = (λ*x*. *of-int* (*floor x*) ∗ (*of-int* (*floor x*) + *1*) / *2* − *x* ^ *2* / *2* :: *real*)
define *H′* **where** *H′* = (λ*x*. *sum-upto* (λ*n*. *moebius-mu n* ∗ *H* (*x* / *real n*)) *x*)
have *H*: *sum-upto real x* = *x*^*2*/*2* + *H x* **if** *x* ≥ *0* **for** *x*
using *that* **by** (*simp add*: *sum-upto-real H-def*)
define *G* **where** *G* = (λ*x*::*real*. (∑*n*. *moebius-mu* (*n* + *Suc* (*nat* ⌊*x*⌋)) / (*n* + *Suc* (*nat* ⌊*x*⌋))^*2*))

have *H-bound*: |*H t*| ≤ *t* / *2* **if** *t* ≥ *0* **for** *t*
proof −
have *H t* − *t* / *2* = (−(*t* − *of-int* (*floor t*))) ∗ (*floor t* + *t* + *1*) / *2*
by (*simp add*: *H-def field-simps power2-eq-square*)
also have . . . ≤ *0* **using** *that* **by** (*intro mult-nonpos-nonneg divide-nonpos-nonneg*) *simp-all*
finally have *H t* ≤ *t* / *2* **by** *simp*
have −*H t* − *t* / *2* = (*t* − *of-int* (*floor t*) − *1*) ∗ (*of-int* (*floor t*) + *t*) / *2*
by (*simp add*: *H-def field-simps power2-eq-square*)
also have . . . ≤ *0* **using** *that*
by (*intro divide-nonpos-nonneg mult-nonpos-nonneg*) ((*simp*; *fail*) | *linarith*)+
finally have −*H t* ≤ *t* / *2* **by** *simp*
with ‹*H t* ≤ *t* / *2*› **show** |*H t*| ≤ *t* / *2* **by** *simp*
qed

have *H′-bound*: |*H′ t*| ≤ *t* / *2* ∗ *sum-upto* (λ*i*. *1* / *real i*) *t* **if** *t* ≥ *0* **for** *t*
proof −
have |*H′ t*| ≤ (∑*i* | *0* < *i* ∧ *real i* ≤ *t*. |*moebius-mu i* ∗ *H* (*t* / *real i*)|)
unfolding *H′-def sum-upto-def* **by** (*rule sum-abs*)
also have . . . ≤ (∑*i* | *0* < *i* ∧ *real i* ≤ *t*. *1* ∗ ((*t* / *real i*) / *2*))
unfolding *abs-mult* **using** *that*
by (*intro sum-mono mult-mono abs-moebius-mu-le H-bound*) *simp-all*
also have . . . = *t* / *2* ∗ *sum-upto* (λ*i*. *1* / *real i*) *t*
by (*simp add*: *sum-upto-def sum-distrib-left sum-distrib-right mult-ac*)
finally show *?thesis* **.**
qed
hence *H′* ∈ *O*(λ*t*. *t* ∗ *sum-upto* (λ*i*. *1* / *real i*) *t*)
using *sum-upto-inverse-bound*(*1*)
by (*intro bigoI*[*of* - *1*/*2*] *eventually-mono*[*OF eventually-ge-at-top*[*of 0*::*real*]])

(*auto elim*!: *eventually-mono simp*: *abs-mult*)
also have (λ*t*. *t* ∗ *sum-upto* (λ*i*. *1* / *real i*) *t*) ∈ *O*(λ*t*. *t* ∗ *ln t*)
by (*intro landau-o.big.mult sum-upto-inverse-bigo*) *simp-all*
finally have *H′-bigo*: *H′* ∈ *O*(λ*x*. *x* ∗ *ln x*) **.**

{
fix *x* :: *real* **assume** *x*: *x* ≥ *0*
have *sum-upto* (λ*n*. *real* (*totient n*)) *x* = *sum-upto* (λ*n*. *of-int* (*int* (*totient n*))) *x*
by *simp*
also have . . . = *sum-upto* (λ*n*. *moebius-mu n* ∗ *sum-upto real* (*x* / *real n*)) *x*
by (*subst totient-conv-moebius-mu*) (*simp add*: *sum-upto-dirichlet-prod of-int-dirichlet-prod*)
also have . . . = *sum-upto* (λ*n*. *moebius-mu n* ∗ ((*x* / *real n*) ^ *2* / *2* + *H* (*x* / *real n*))) *x* **using** *x*
by (*intro sum-upto-cong*) (*simp-all add*: *H*)
also have . . . = *x*^*2* / *2* ∗ *sum-upto* (λ*n*. *moebius-mu n* / *real n* ^ *2*) *x* + *H′* *x*
by (*simp add*: *sum-upto-def H′-def sum.distrib ring-distribs*
sum-distrib-left sum-distrib-right power-divide mult-ac)
also have *sum-upto* (λ*n*. *moebius-mu n* / *real n* ^ *2*) *x* =
(∑*n*∈{..<*Suc* (*nat* ⌊*x*⌋)}. *moebius-mu n* / *real n* ^ *2*)
unfolding *sum-upto-altdef* **by** (*intro sum.mono-neutral-cong-left refl*) *auto*
also have . . . = *6* / *pi* ^ *2* − *G x*
using *sums-split-initial-segment*[*OF moebius-over-square-sums*, *of Suc* (*nat* ⌊*x*⌋)]
by (*auto simp*: *sums-iff algebra-simps G-def*)
finally have *sum-upto* (λ*n*. *real* (*totient n*)) *x* = *3* / pi^2 ∗ x^2 − x^2 / *2* ∗ *G x* + *H′* *x*
by (*simp add*: *algebra-simps*)
}
hence (λ*x*. *sum-upto* (λ*n*. *real* (*totient n*)) *x* − *3* / *pi*^*2* ∗ *x*^*2*) ∈
Θ(λ*x*. (−(*x*^*2*) / *2*) ∗ *G x* + *H′* *x*)
by (*intro bigthetaI-cong eventually-mono*[*OF eventually-ge-at-top*[*of 0*::*real*]])
(*auto elim*!: *eventually-mono*)
also have (λ*x*. (−(*x*^*2*) / *2*) ∗ *G x* + *H′* *x*) ∈ *O*(λ*x*. *x* ∗ *ln x*)
proof (*intro sum-in-bigo H′-bigo*)
have (λ*x*. (− (*x*^*2*) / *2*) ∗ *G x*) ∈ *O*(λ*x*. *x*^*2* ∗ (*1* / *x*))
using *moebius-sum-tail-bigo* [*folded G-def*] **by** (*intro landau-o.big.mult*) *simp-all*
also have (λ*x*::*real*. *x*^*2* ∗ (*1* / *x*)) ∈ *O*(λ*x*. *x* ∗ *ln x*) **by** *simp*
finally show (λ*x*. (− (*x*^*2*) / *2*) ∗ *G x*) ∈ *O*(λ*x*. *x* ∗ *ln x*) **.**
qed
finally show *?thesis* **.**
qed

theorem *summatory-totient-asymptotics′*:
(λ*x*. *sum-upto* (λ*n*. *real* (*totient n*)) *x*) =*o* (λ*x*. *3* / pi^2 ∗ x^2) +*o* *O*(λ*x*. *x* ∗ *ln x*)
using *summatory-totient-asymptotics*
by (*subst set-minus-plus* [*symmetric*]) (*simp-all add*: *fun-diff-def*)

theorem *summatory-totient-asymptotics''*:
sum-upto (λ*n*. *real* (*totient n*)) ∼[*at-top*] (λ*x*. *3* / *pi*^{2} ∗ *x*^{2})
proof −
have (λ*x*. *sum-upto* (λ*n*. *real* (*totient n*)) *x* − *3* / *pi*^{2} ∗ *x*^{2}) ∈ *O*(λ*x*. *x* ∗ *ln x*)
by (*rule summatory-totient-asymptotics*)
also have (λ*x*. *x* ∗ *ln x*) ∈ *o*(λ*x*. *3* / *pi* ^ *2* ∗ *x* ^ *2*) **by** *simp*
finally show *?thesis* **by** (*simp add*: *asymp-equiv-altdef*)
qed

13.3 Asymptotic distribution of squarefree numbers

lemma *le-sqrt-iff*: *x* ≥ *0* ⟹ *x* ≤ *sqrt y* ⟷ *x*^*2* ≤ *y*
using *real-sqrt-le-iff*[*of x*^*2 y*] **by** (*simp del*: *real-sqrt-le-iff*)

theorem *squarefree-asymptotics*: (λ*x*. *card* {*n*. *real n* ≤ *x* ∧ *squarefree n*} − *6* / *pi*^{2} ∗ *x*) ∈ *O*(*sqrt*)
proof −
define *f* :: *nat* ⇒ *real* **where** *f* = (λ*n*. *if n* = *0 then 0 else 1*)
define *g* :: *nat* ⇒ *real* **where** *g* = *dirichlet-prod* (*ind squarefree*) *moebius-mu*

interpret *g*: *multiplicative-function g* **unfolding** *g-def*
by (*intro multiplicative-dirichlet-prod squarefree.multiplicative-function-axioms moebius-mu.multiplicative-function-axioms*)
interpret *g*: *multiplicative-function' g* λ*p k*. *if k* = *2 then* −*1 else 0* λ-. *0*
proof
interpret *g'*: *multiplicative-dirichlet-prod' ind squarefree moebius-mu*
λ*p k*. *if 1* < *k then 0 else 1* λ*p k*. *if k* = *1 then* − *1 else 0* λ-. *1* λ-. − *1*
by (*intro multiplicative-dirichlet-prod'.intro squarefree.multiplicative-function'-axioms*

moebius-mu.multiplicative-function'-axioms)
fix *p k* :: *nat* **assume** *prime p k* > *0*
hence *g* (*p* ^ *k*) = (∑*i*∈{*0*<..<*k*}. (*if Suc 0* < *i then 0 else 1*) ∗
(*if k* − *i* = *Suc 0 then* − *1 else 0*))
by (*auto simp*: *g'.prime-power g-def*)
also have ... = (∑*i*∈{*0*<..<*k*}. (*if k* = *2 then* −*1 else 0*))
by (*intro sum.cong refl*) *auto*
also from ‹*k* > *0*› **have** ... = (*if k* = *2 then* −*1 else 0*) **by** *simp*
finally show *g* (*p* ^ *k*) = ... **.**
qed *simp-all*
have *mult-g-square*: *multiplicative-function* (λ*n*. *g* (*n* ^ *2*))
by *standard* (*simp-all add*: *power-mult-distrib g.mult-coprime*)

have *g-square*: *g* (*m* ^ *2*) = *moebius-mu m* **for** *m*
using *mult-g-square moebius-mu.multiplicative-function-axioms*
proof (*rule multiplicative-function-eqI*)
fix *p k* :: *nat* **assume** ∗: *prime p k* > *0*
have *g* ((*p* ^ *k*) ^ *2*) = *g* (*p* ^(*2* ∗ *k*)) **by** (*simp add*: *power-mult* [*symmetric*] *mult-ac*)

also from $*$ **have** . . . = (*if* k = *1 then* −*1 else 0*) **by** (*simp add*: *g.prime-power*)
also from $*$ **have** . . . = *moebius-mu* (*p* ^ *k*) **by** (*simp add*: *moebius-mu.prime-power*)
finally show *g* ((*p* ^ *k*) ^ *2*) = *moebius-mu* (*p* ^ *k*) **.**
qed

have *g-nonsquare*: *g m* = *0* **if** ¬*is-square m* **for** *m*
proof (*cases m* = *0*)
case *False*
from *that False* **obtain** *p* **where** *p*: *prime p odd* (*multiplicity p m*)
using *is-nth-power-conv-multiplicity-nat*[*of 2 m*] **by** *auto*
from *p* **have** *multiplicity p m* ≠ *2* **by** *auto*
moreover from *p* **have** *p* ∈ *prime-factors m*
by (*auto simp*: *prime-factors-multiplicity intro*!: *Nat.gr0I*)
ultimately have (∏ *p*∈*prime-factors m. if multiplicity p m* = *2 then* − *1 else 0* :: *real*) = *0*
(**is** *?P* = -) **by** *auto*
also have *?P* = *g m* **using** *False* **by** (*subst g.prod-prime-factors′*) *auto*
finally show *?thesis* **.**
qed *auto*

have *abs-g-le*: *abs* (*g m*) ≤ *1* **for** *m*
by (*cases is-square m*)
(*auto simp*: *g-square g-nonsquare abs-moebius-mu-le elim*!: *is-nth-powerE*)

have *fds-g*: *fds g* = *fds-ind squarefree* ∗ *fds moebius-mu*
by (*rule fds-eqI*) (*simp add*: *g-def fds-nth-mult*)
have *fds g* ∗ *fds-zeta* = *fds-ind squarefree* ∗ (*fds-zeta* ∗ *fds moebius-mu*)
by (*simp add*: *fds-g mult-ac*)
also have *fds-zeta* ∗ *fds moebius-mu* = (*1* :: *real fds*)
by (*rule fds-zeta-times-moebius-mu*)
finally have ∗: *fds-ind squarefree* = *fds g* ∗ *fds-zeta* **by** *simp*
have *ind-squarefree*: *ind squarefree* = *dirichlet-prod g f*
proof
fix *n* :: *nat*
from ∗ **show** *ind squarefree n* = *dirichlet-prod g f n*
by (*cases n* = *0*) (*simp-all add*: *fds-eq-iff fds-nth-mult f-def*)
qed

define *H* :: *real* ⇒ *real*
where *H* = (λ*x. sum-upto* (λ*m. g* (*m*^*2*) ∗ (*real-of-int* ⌊*x* / *real* (m^2)⌋ − *x* / *real* (*m*^*2*))) (*sqrt x*))
define *J* **where** *J* = (λ*x*::*real.* (∑ *n. moebius-mu* (*n* + *Suc* (*nat* ⌊*x*⌋)) / (*n* + *Suc* (*nat* ⌊*x*⌋))^*2*))

have *eventually* (λ*x. norm* (*H x*) ≤ *1* ∗ *norm* (*sqrt x*)) *at-top*
using *eventually-ge-at-top*[*of 0*::*real*]
proof *eventually-elim*
case (*elim x*)
have *abs* (*H x*) ≤ *sum-upto* (λ*m. abs* (*g* (*m*^*2*) ∗ (*real-of-int* ⌊*x* / *real* (m^2)⌋

–

x / real (m^2)))) (sqrt x) (**is** - ≤ ?S) **unfolding** *H-def sum-upto-def*
by (*rule sum-abs*)
also have x / (*real* $m)^2$ − *real-of-int* ⌊x / (*real* $m)^2$⌋ ≤ *1* **for** *m* **by** *linarith*
hence *?S* ≤ *sum-upto* (λ*m*. *1* ∗ *1*) (*sqrt x*) **unfolding** *abs-mult sum-upto-def*
by (*intro sum-mono mult-mono abs-g-le*) *simp-all*
also have ... = *of-int* ⌊*sqrt x*⌋ **using** *elim* **by** (*simp add*: *sum-upto-altdef*)
also have ... ≤ *sqrt x* **by** *linarith*
finally show *?case* **using** *elim* **by** *simp*
qed
hence *H-bigo*: *H* ∈ *O*(λ*x*. *sqrt x*) **by** (*rule bigoI*)

let *?A* = λ*x*. *card* {*n*. *real n* ≤ *x* ∧ *squarefree n*}
have *eventually* (λ*x*. *?A x* − *6* / *pi*^{2} ∗ *x* = (−*x*) ∗ *J* (*sqrt x*) + *H x*) *at-top*
using *eventually-ge-at-top*[*of 0*::*real*]
proof *eventually-elim*
fix *x* :: *real* **assume** *x*: *x* ≥ *0*
have {*n*. *real n* ≤ *x* ∧ *squarefree n*} = {*n*. *n* > *0* ∧ *real n* ≤ *x* ∧ *squarefree n*}

by (*auto intro*!: *Nat.gr0I*)
also have *card* ... = *sum-upto* (*ind squarefree* :: *nat* ⇒ *real*) *x*
by (*rule sum-upto-ind* [*symmetric*])
also have ... = *sum-upto* (λ*d*. *g d* ∗ *sum-upto f* (*x* / *real d*)) *x* (**is** - = *?S*)
unfolding *ind-squarefree* **by** (*rule sum-upto-dirichlet-prod*)
also have *sum f* {*0*<..*nat* ⌊*x* / *real i*⌋} = *of-int* ⌊*x* / *real i*⌋ **if** *i* > *0* **for** *i*
using *x* **by** (*simp add*: *f-def*)
hence *?S* = *sum-upto* (λ*d*. *g d* ∗ *of-int* ⌊*x* / *real d*⌋) *x*
unfolding *sum-upto-altdef* **by** (*intro sum.cong refl*) *simp-all*
also have ... = *sum-upto* (λ*m*. *g* (*m* ^ *2*) ∗ *of-int* ⌊*x* / *real* (*m* ^ *2*)⌋) (*sqrt x*)
unfolding *sum-upto-def*
proof (*intro sum.reindex-bij-betw-not-neutral* [*symmetric*])
show *bij-betw power2* ({*i*. *0* < *i* ∧ *real i* ≤ *sqrt x*} − {})
({*i*. *0* < *i* ∧ *real i* ≤ *x*} − {*i*∈{*0*<..*nat* ⌊*x*⌋}. ¬*is-square i*})
by (*auto simp*: *bij-betw-def inj-on-def power-eq-iff-eq-base le-sqrt-iff*
is-nth-power-def le-nat-iff le-floor-iff)
qed (*auto simp*: *g-nonsquare*)
also have ... = *x* ∗ *sum-upto* (λ*m*. *g* (*m* ^ *2*) / *real m* ^ *2*) (*sqrt x*) + *H x*
by (*simp add*: *H-def sum-upto-def sum.distrib ring-distribs sum-subtractf*
sum-distrib-left sum-distrib-right mult-ac)
also have *sum-upto* (λ*m*. *g* (*m* ^ *2*) / *real m* ^ *2*) (*sqrt x*) =
sum-upto (λ*m*. *moebius-mu m* / *real m* ^ *2*) (*sqrt x*)
unfolding *sum-upto-altdef* **by** (*intro sum.cong refl*) (*simp-all add*: *g-square*)
also have *sum-upto* (λ*m*. *moebius-mu m* / (*real m*)^2) (*sqrt x*) =
(∑*m*<*Suc* (*nat* ⌊*sqrt x*⌋). *moebius-mu m* / (*real m*) ^ *2*)
unfolding *sum-upto-altdef* **by** (*intro sum.mono-neutral-cong-left*) *auto*
also have ... = (*6* / *pi*^*2* − *J* (*sqrt x*))
using *sums-split-initial-segment*[*OF moebius-over-square-sums*, *of Suc* (*nat* ⌊*sqrt x*⌋)]

by (*auto simp*: *sums-iff algebra-simps J-def sum-upto-altdef*)
finally show *?A x* − *6* / *pi*^{2} ∗ *x* = (−*x*) ∗ *J* (*sqrt x*) + *H x*
by (*simp add*: *algebra-simps*)
qed
hence (λ*x*. *?A x* − *6* / *pi*^{2} ∗ *x*) ∈ Θ(λ*x*. (−*x*) ∗ *J* (*sqrt x*) + *H x*)
by (*rule bigthetaI-cong*)
also have (λ*x*. (−*x*) ∗ *J* (*sqrt x*) + *H x*) ∈ *O*(λ*x*. *sqrt x*)
proof (*intro sum-in-bigo H-bigo*)
have (λ*x*. *J* (*sqrt x*)) ∈ *O*(λ*x*. *1* / *sqrt x*) **unfolding** *J-def*
using *moebius-sum-tail-bigo sqrt-at-top* **by** (*rule landau-o.big.compose*)
hence (λ*x*. (−*x*) ∗ *J* (*sqrt x*)) ∈ *O*(λ*x*. *x* ∗ (*1* / *sqrt x*))
by (*intro landau-o.big.mult*) *simp-all*
also have (λ*x*::*real*. *x* ∗ (*1* / *sqrt x*)) ∈ Θ(λ*x*. *sqrt x*)
by (*intro bigthetaI-cong eventually-mono*[*OF eventually-gt-at-top*[*of 0*::*real*]])
(*auto simp*: *field-simps*)
finally show (λ*x*. (−*x*) ∗ *J* (*sqrt x*)) ∈ *O*(λ*x*. *sqrt x*) **.**
qed
finally show *?thesis* **.**
qed

theorem *squarefree-asymptotics′*:
(λ*x*. *card* {*n*. *real n* ≤ *x* ∧ *squarefree n*}) =*o* (λ*x*. *6* / *pi*^{2} ∗ *x*) +*o* *O*(λ*x*. *sqrt x*)
using *squarefree-asymptotics*
by (*subst set-minus-plus* [*symmetric*]) (*simp-all add*: *fun-diff-def*)

theorem *squarefree-asymptotics″*:
(λ*x*. *card* {*n*. *real n* ≤ *x* ∧ *squarefree n*}) ∼[*at-top*] (λ*x*. *6* / *pi*^{2} ∗ *x*)
proof −
have (λ*x*. *card* {*n*. *real n* ≤ *x* ∧ *squarefree n*} − *6* / *pi*^{2} ∗ *x*) ∈ *O*(λ*x*. *sqrt x*)
by (*rule squarefree-asymptotics*)
also have (*sqrt* :: *real* ⇒ *real*) ∈ Θ(λ*x*. *x powr* (*1/2*))
by (*intro bigthetaI-cong eventually-mono*[*OF eventually-ge-at-top*[*of 0*::*real*]])
(*auto simp*: *powr-half-sqrt*)
also have (λ*x*::*real*. *x powr* (*1/2*)) ∈ *o*(λ*x*. *6* / *pi* ^ *2* ∗ *x*) **by** *simp*
finally show *?thesis* **by** (*simp add*: *asymp-equiv-altdef*)
qed

13.4 The hyperbola method

lemma *hyperbola-method-bigo*:
fixes *f g* :: *nat* ⇒ ′*a* :: *real-normed-field*
assumes (λ*x*. *sum-upto* (λ*n*. *f n* ∗ *sum-upto g* (*x* / *real n*)) (*sqrt x*) − *R x*) ∈ *O*(*b*)
assumes (λ*x*. *sum-upto* (λ*n*. *sum-upto f* (*x* / *real n*) ∗ *g n*) (*sqrt x*) − *S x*) ∈ *O*(*b*)
assumes (λ*x*. *sum-upto f* (*sqrt x*) ∗ *sum-upto g* (*sqrt x*) − *T x*) ∈ *O*(*b*)
shows (λ*x*. *sum-upto* (*dirichlet-prod f g*) *x* − (*R x* + *S x* − *T x*)) ∈ *O*(*b*)
proof −
let *?A* = λ*x*. (*sum-upto* (λ*n*. *f n* ∗ *sum-upto g* (*x* / *real n*)) (*sqrt x*) − *R x*) +

```
          (sum-upto (λn. sum-upto f (x / real n) * g n) (sqrt x) − S x) +
          (−(sum-upto f (sqrt x) * sum-upto g (sqrt x) − T x))
  have (λx. sum-upto (dirichlet-prod f g) x − (R x + S x − T x)) ∈ Θ(?A)
    by (intro bigthetaI-cong eventually-mono[OF eventually-ge-at-top[of 0::real]])
      (auto simp: hyperbola-method-sqrt)
  also from assms have ?A ∈ O(b)
    by (intro sum-in-bigo(1)) (simp-all only: landau-o.big.uminus-in-iff)
  finally show ?thesis .
qed

lemma frac-le-1: frac x ≤ 1
  unfolding frac-def by linarith

lemma ln-minus-ln-floor-bound:
  assumes x ≥ 2
  shows   ln x − ln (floor x) ∈ {0..<1 / (x − 1)}
proof −
  from assms have ln (floor x) ≥ ln (x − 1) by (subst ln-le-cancel-iff) simp-all
  hence ln x − ln (floor x) ≤ ln ((x − 1) + 1) − ln (x − 1) by simp
  also from assms have ... < 1 / (x − 1) by (intro ln-diff-le-inverse) simp-all
  finally have ln x − ln (floor x) < 1 / (x − 1) by simp
  moreover from assms have ln x ≥ ln (of-int ⌊x⌋) by (subst ln-le-cancel-iff)
simp-all
  ultimately show ?thesis by simp
qed

lemma ln-minus-ln-floor-bigo:
  (λx::real. ln x − ln (floor x)) ∈ O(λx. 1 / x)
proof −
  have eventually (λx. norm (ln x − ln (floor x)) ≤ 1 * norm (1 / (x − 1)))
at-top
    using eventually-ge-at-top[of 2::real]
  proof eventually-elim
    case (elim x)
    with ln-minus-ln-floor-bound[OF this] show ?case by auto
  qed
  hence (λx::real. ln x − ln (floor x)) ∈ O(λx. 1 / (x − 1)) by (rule bigoI)
  also have (λx::real. 1 / (x − 1)) ∈ O(λx. 1 / x) by simp
  finally show ?thesis .
qed

lemma divisor-count-asymptotics-aux:
  (λx. sum-upto (λn. sum-upto (λ-. 1) (x / real n)) (sqrt x) −
                (x * ln x / 2 + euler-mascheroni * x)) ∈ O(sqrt)
proof −
  define R where R = (λx. ∑ i∈{0<..nat ⌊sqrt x⌋}. frac (x / real i))
  define S where S = (λx. ln (real (nat ⌊sqrt x⌋)) − ln x / 2)
  have R-bound: R x ∈ {0..sqrt x} if x: x ≥ 0 for x
  proof −
```

have *R x* ≤ (∑ *i*∈{*0*<..*nat* ⌊*sqrt x*⌋}. *1*) **unfolding** *R-def* **by** (*intro sum-mono frac-le-1*)
also from *x* **have** ... = *of-int* ⌊*sqrt x*⌋ **by** *simp*
also have ... ≤ *sqrt x* **by** *simp*
finally have *R x* ≤ *sqrt x* **.**
moreover have *R x* ≥ *0* **unfolding** *R-def* **by** (*intro sum-nonneg*) *simp-all*
ultimately show *?thesis* **by** *simp*
qed
have *R-bound′*: *norm* (*R x*) ≤ *1* ∗ *norm* (*sqrt x*) **if** *x* ≥ *0* **for** *x*
using *R-bound*[*OF that*] *that* **by** *simp*
have *R-bigo*: *R* ∈ *O*(*sqrt*) **using** *eventually-ge-at-top*[*of 0*::*real*]
by (*intro bigoI*[*of - 1*], *elim eventually-mono*) (*rule R-bound′*)

have *eventually* (λ*x*. *sum-upto* (λ*n*. *sum-upto* (λ-. *1* :: *real*) (*x* / *real n*)) (*sqrt x*) =
x ∗ *harm* (*nat* ⌊*sqrt x*⌋) − *R x*) *at-top*
using *eventually-ge-at-top*[*of 0* :: *real*]
proof *eventually-elim*
case (*elim x*)
have *sum-upto* (λ*n*. *sum-upto* (λ-. *1* :: *real*) (*x* / *real n*)) (*sqrt x*) =
(∑ *i*∈{*0*<..*nat* ⌊*sqrt x*⌋}. *of-int* ⌊*x* / *real i*⌋) **using** *elim*
by (*simp add*: *sum-upto-altdef*)
also have ... = *x* ∗ (∑ *i*∈{*0*<..*nat* ⌊*sqrt x*⌋}. *1* / *real i*) − *R x*
by (*simp add*: *sum-subtractf frac-def R-def sum-distrib-left*)
also have {*0*<..*nat* ⌊*sqrt x*⌋} = {*1*..*nat* ⌊*sqrt x*⌋} **by** *auto*
also have (∑ *i*∈.... *1* / *real i*) = *harm* (*nat* ⌊*sqrt x*⌋) **by** (*simp add*: *harm-def divide-simps*)
finally show *?case* **.**
qed
hence (λ*x*. *sum-upto* (λ*n*. *sum-upto* (λ-. *1* :: *real*) (*x* / *real n*)) (*sqrt x*) −
(*x* ∗ *ln x* / *2* + *euler-mascheroni* ∗ *x*)) ∈
Θ(λ*x*. *x* ∗ (*harm* (*nat* ⌊*sqrt x*⌋) − (*ln* (*nat* ⌊*sqrt x*⌋) + *euler-mascheroni*)) − *R x* + *x* ∗ *S x*)
(**is** - ∈ Θ(*?A*))
by (*intro bigthetaI-cong*) (*elim eventually-mono*, *simp-all add*: *algebra-simps S-def*)
also have *?A* ∈ *O*(*sqrt*)
proof (*intro sum-in-bigo*)
have (λ*x*. − *S x*) ∈ Θ(λ*x*. *ln* (*sqrt x*) − *ln* (*of-int* ⌊*sqrt x*⌋))
by (*intro bigthetaI-cong eventually-mono* [*OF eventually-ge-at-top*[*of 1*::*real*]])

(*auto simp*: *S-def ln-sqrt*)
also have (λ*x*. *ln* (*sqrt x*) − *ln* (*of-int* ⌊*sqrt x*⌋)) ∈ *O*(λ*x*. *1* / *sqrt x*)
by (*rule landau-o.big.compose*[*OF ln-minus-ln-floor-bigo sqrt-at-top*])
finally have (λ*x*. *x* ∗ *S x*) ∈ *O*(λ*x*. *x* ∗ (*1* / *sqrt x*)) **by** (*intro landau-o.big.mult*) *simp-all*
also have (λ*x*::*real*. *x* ∗ (*1* / *sqrt x*)) ∈ Θ(λ*x*. *sqrt x*)
by (*intro bigthetaI-cong eventually-mono* [*OF eventually-gt-at-top*[*of 0*::*real*]])

```
        (auto simp: field-simps)
    finally show (λx. x ∗ S x) ∈ O(sqrt) .
  next
    let ?f = λx::real. harm (nat ⌊sqrt x⌋) − (ln (real (nat ⌊sqrt x⌋)) + euler-mascheroni)
    have ?f ∈ O(λx. 1 / real (nat ⌊sqrt x⌋))
    proof (rule landau-o.big.compose[of - - - λx. nat ⌊sqrt x⌋])
      show filterlim (λx::real. nat ⌊sqrt x⌋) at-top at-top
        by (intro filterlim-compose[OF filterlim-nat-sequentially]
                filterlim-compose[OF filterlim-floor-sequentially] sqrt-at-top)
    next
      show (λa. harm a − (ln (real a) + euler-mascheroni)) ∈ O(λa. 1 / real a)
        by (rule harm-expansion-bigo-simple2)
    qed
    also have (λx. 1 / real (nat ⌊sqrt x⌋)) ∈ O(λx. 1 / (sqrt x − 1))
    proof (rule bigoI[of - 1], use eventually-ge-at-top[of 2] in eventually-elim)
      case (elim x)
      have sqrt x ≤ 1 + real-of-int ⌊sqrt x⌋ by linarith
      with elim show ?case by (simp add: field-simps)
    qed
    also have (λx::real. 1 / (sqrt x − 1)) ∈ O(λx. 1 / sqrt x)
      by (rule landau-o.big.compose[OF - sqrt-at-top]) simp-all
    finally have (λx. x ∗ ?f x) ∈ O(λx. x ∗ (1 / sqrt x))
      by (intro landau-o.big.mult landau-o.big-refl)
    also have (λx::real. x ∗ (1 / sqrt x)) ∈ Θ(λx. sqrt x)
      by (intro bigthetaI-cong eventually-mono[OF eventually-gt-at-top[of 0::real]])
         (auto elim!: eventually-mono simp: field-simps)
    finally show (λx. x ∗ ?f x) ∈ O(sqrt) .
  qed fact+
  finally show ?thesis .
qed

lemma sum-upto-sqrt-bound:
  assumes x: x ≥ (0 :: real)
  shows   norm ((sum-upto (λ-. 1) (sqrt x))^2 − x) ≤ 2 ∗ norm (sqrt x)
proof −
  from x have 0 ≤ 2 ∗ sqrt x ∗ (1 − frac (sqrt x)) + frac (sqrt x) ^ 2
    by (intro add-nonneg-nonneg mult-nonneg-nonneg) (simp-all add: frac-le-1)
  also from x have ... = (sqrt x − frac (sqrt x)) ^ 2 − x + 2 ∗ sqrt x
    by (simp add: algebra-simps power2-eq-square)
  also have sqrt x − frac (sqrt x) = of-int ⌊sqrt x⌋ by (simp add: frac-def)
  finally have (of-int ⌊sqrt x⌋) ^ 2 − x ≥ −2 ∗ sqrt x by (simp add: algebra-simps)
  moreover from x have of-int (⌊sqrt x⌋) ^ 2 ≤ sqrt x ^ 2
    by (intro power-mono) simp-all
  with x have of-int (⌊sqrt x⌋) ^ 2 − x ≤ 0 by simp
  ultimately have sum-upto (λ-. 1) (sqrt x) ^ 2 − x ∈ {−2 ∗ sqrt x..0}
    using x by (simp add: sum-upto-altdef)
  with x show ?thesis by simp
qed
```

```
lemma summatory-divisor-count-asymptotics:
  (λx. sum-upto (λn. real (divisor-count n)) x −
        (x ∗ ln x + (2 ∗ euler-mascheroni − 1) ∗ x)) ∈ O(sqrt)
proof −
  let ?f = λx. x ∗ ln x / 2 + euler-mascheroni ∗ x
  have (λx. sum-upto (dirichlet-prod (λ-. 1 :: real) (λ-. 1)) x − (?f x + ?f x − x))
∈ O(sqrt)
    (is ?g ∈ -)
  proof (rule hyperbola-method-bigo)
    have eventually (λx::real. norm (sum-upto (λ-. 1) (sqrt x) ^ 2 − x) ≤
          2 ∗ norm (sqrt x)) at-top
     using eventually-ge-at-top[of 0::real] by eventually-elim (rule sum-upto-sqrt-bound)
     thus (λx::real. sum-upto (λ-. 1) (sqrt x) ∗ sum-upto (λ-. 1) (sqrt x) − x) ∈
O(sqrt)
      by (intro bigoI[of - 2]) (simp-all add: power2-eq-square)
  next
    show (λx. sum-upto (λn. 1 ∗ sum-upto (λ-. 1) (x / real n)) (sqrt x) −
              (x ∗ ln x / 2 + euler-mascheroni ∗ x)) ∈ O(sqrt)
      using divisor-count-asymptotics-aux by simp
  next
    show (λx. sum-upto (λn. sum-upto (λ-. 1) (x / real n) ∗ 1) (sqrt x) −
              (x ∗ ln x / 2 + euler-mascheroni ∗ x)) ∈ O(sqrt)
      using divisor-count-asymptotics-aux by simp
  qed
  also have divisor-count n = dirichlet-prod (λ-. 1) (λ-. 1) n for n
    using fds-divisor-count
    by (cases n = 0) (simp-all add: fds-eq-iff power2-eq-square fds-nth-mult)
  hence ?g = (λx. sum-upto (λn. real (divisor-count n)) x −
            (x ∗ ln x + (2 ∗ euler-mascheroni − 1) ∗ x))
    by (intro ext) (simp-all add: algebra-simps dirichlet-prod-def)
  finally show ?thesis .
qed

theorem summatory-divisor-count-asymptotics':
  (λx. sum-upto (λn. real (divisor-count n)) x) =o
     (λx. x ∗ ln x + (2 ∗ euler-mascheroni − 1) ∗ x) +o O(λx. sqrt x)
  using summatory-divisor-count-asymptotics
  by (subst set-minus-plus [symmetric]) (simp-all add: fun-diff-def)

theorem summatory-divisor-count-asymptotics'':
  sum-upto (λn. real (divisor-count n)) ∼[at-top] (λx. x ∗ ln x)
proof −
  have (λx. sum-upto (λn. real (divisor-count n)) x −
          (x ∗ ln x + (2 ∗ euler-mascheroni − 1) ∗ x)) ∈ O(sqrt)
    by (rule summatory-divisor-count-asymptotics)
  also have sqrt ∈ Θ(λx. x powr (1/2))
    by (intro bigthetaI-cong eventually-mono [OF eventually-ge-at-top[of 0::real]])
       (auto elim!: eventually-mono simp: powr-half-sqrt)
```

also have (λx::real. x powr (1/2)) ∈ o(λx. x ∗ ln x + (2 ∗ euler-mascheroni − 1) ∗ x) **by** simp
finally have sum-upto (λn. real (divisor-count n)) ∼[at-top]
(λx. x ∗ ln x + (2 ∗ euler-mascheroni − 1) ∗ x)
by (simp add: asymp-equiv-altdef)
also have ... ∼[at-top] (λx. x ∗ ln x) **by** (subst asymp-equiv-add-right) simp-all
finally show ?thesis **.**
qed

lemma summatory-divisor-eq:
sum-upto (λn. real (divisor-count n)) (real m) = card {(n,d). n ∈ {0<..m} ∧ d dvd n}
proof −
have sum-upto (λn. real (divisor-count n)) m = card (SIGMA n:{0<..m}. {d. d dvd n})
unfolding sum-upto-altdef divisor-count-def **by** (subst card-SigmaI) simp-all
also have (SIGMA n:{0<..m}. {d. d dvd n}) = {(n,d). n ∈ {0<..m} ∧ d dvd n} **by** auto
finally show ?thesis **.**
qed

context
fixes M :: nat ⇒ real
defines M ≡ λm. card {(n,d). n ∈ {0<..m} ∧ d dvd n} / card {0<..m}
begin

lemma mean-divisor-count-asymptotics:
(λm. M m − (ln m + 2 ∗ euler-mascheroni − 1)) ∈ O(λm. 1 / sqrt m)
proof −
have (λm. M m − (ln m + 2 ∗ euler-mascheroni − 1))
∈ Θ(λm. (sum-upto (λn. real (divisor-count n)) (real m) −
(m ∗ ln m + (2 ∗ euler-mascheroni − 1) ∗ m)) / m) (**is** - ∈ Θ(?f))
unfolding M-def
by (intro bigthetaI-cong eventually-mono [OF eventually-gt-at-top[of 0::nat]])
(auto simp: summatory-divisor-eq field-simps)
also have ?f ∈ O(λm. sqrt m / m)
by (intro landau-o.big.compose[OF - filterlim-real-sequentially] landau-o.big.divide-right
summatory-divisor-count-asymptotics eventually-at-top-not-equal)
also have (λm::nat. sqrt m / m) ∈ Θ(λm. 1 / sqrt m)
by (intro bigthetaI-cong eventually-mono [OF eventually-gt-at-top[of 0::nat]])
(auto simp: field-simps)
finally show ?thesis **.**
qed

theorem mean-divisor-count-asymptotics′:
M =o (λx. ln x + 2 ∗ euler-mascheroni − 1) +o O(λx. 1 / sqrt x)
using mean-divisor-count-asymptotics
by (subst set-minus-plus [symmetric]) (simp-all add: fun-diff-def)

theorem *mean-divisor-count-asymptotics''*: $M \sim$[*at-top*] *ln*
proof −
  have (λx. $M\ x - (\ln x + 2 * $ *euler-mascheroni* $- 1)) \in O(\lambda x.\ 1\ /\ $ *sqrt* $x)$
    by (*rule mean-divisor-count-asymptotics*)
  also have (λx. *1 / sqrt* (*real x*)) $\in \Theta(\lambda x.\ x$ *powr* $(-1/2))$
    using *eventually-gt-at-top*[*of 0*::*nat*]
    by (*intro bigthetaI-cong*)
      (*auto elim*!: *eventually-mono simp*: *powr-half-sqrt field-simps powr-minus*)
  also have (λx::*nat*. x *powr* $(-1/2)) \in o(\lambda x.\ \ln x + 2 *$ *euler-mascheroni* $- 1)$
    by (*intro smallo-real-nat-transfer*) *simp-all*
  finally have $M \sim$[*at-top*] (λx. $\ln x + 2 *$ *euler-mascheroni* $- 1$)
    by (*simp add*: *asymp-equiv-altdef*)
  also have ... = (λx::*nat*. $\ln x + (2 *$ *euler-mascheroni* $- 1)$) **by** (*simp add*: *algebra-simps*)
  also have ... $\sim$[*at-top*] (λx::*nat*. *ln x*) **by** (*subst asymp-equiv-add-right*) *auto*
  finally show *?thesis* **.**
qed

end

13.5 The asymptotic ditribution of coprime pairs

context
  fixes *A* :: *nat* ⇒ (*nat* × *nat*) *set*
  defines $A \equiv (\lambda N.\ \{(m,n) \in \{1..N\} \times \{1..N\}.$ *coprime m n*$\})$
begin

lemma *coprime-pairs-asymptotics*:
  (λN. *real* (*card* (*A N*)) $- 6\ /\ pi^2 * ($*real* $N)^2) \in O(\lambda N.$ *real* $N * \ln ($*real* $N))$
proof −
  define *C* :: *nat* ⇒ (*nat* × *nat*) *set*
    where $C = (\lambda N.\ (\bigcup m \in \{1..N\}.\ (\lambda n.\ (m,n))$ ‘ *totatives m*))
  define *D* :: *nat* ⇒ (*nat* × *nat*) *set*
    where $D = (\lambda N.\ (\bigcup n \in \{1..N\}.\ (\lambda m.\ (m,n))$ ‘ *totatives n*))
  have *fin*: *finite* (*C N*) *finite* (*D N*) **for** *N* **unfolding** *C-def D-def*
    by (*intro finite-UN-I finite-imageI*; *simp*)+

  have ∗: *card* (*A N*) $= 2 * (\sum m \in \{0<..N\}.$ *totient m*$) - 1$ **if** *N*: $N > 0$ **for** *N*
  proof −
    have $A\ N = C\ N \cup D\ N$
      by (*auto simp add*: *A-def C-def D-def totatives-def image-iff ac-simps*)
    also have *card* ... = *card* (*C N*) + *card* (*D N*) − *card* ($C\ N \cap D\ N$)
      using *card-Un-Int*[*OF fin*[*of N*]] **by** *arith*
    also have $C\ N \cap D\ N = \{(1, 1)\}$ **using** *N* **by** (*auto simp*: *image-iff totatives-def C-def D-def*)
    also have $D\ N = (\lambda(x,y).\ (y,x))$ ‘ *C N* **by** (*simp add*: *image-UN image-image C-def D-def*)
    also have *card* ... = *card* (*C N*) **by** (*rule card-image*) (*simp add*: *inj-on-def C-def*)

also have *card* (*C N*) = ($\sum$ *m*∈{*1*..*N*}. *card* ((λ*n*. (*m*,*n*)) ‘ *totatives m*))
unfolding *C-def* **by** (*intro card-UN-disjoint*) *auto*
also have ... = ($\sum$ *m*∈{*1*..*N*}. *totient m*) **unfolding** *totient-def*
by (*subst card-image*) (*auto simp*: *inj-on-def*)
also have ... = ($\sum$ *m*∈{*0*<..*N*}. *totient m*) **by** (*intro sum.cong refl*) *auto*
finally show *card* (*A N*) = *2* ∗ ... − *1* **by** *simp*
qed
have ∗∗: ($\sum$ *m*∈{*0*<..*N*}. *totient m*) ≥ *1* **if** *N* ≥ *1* **for** *N*
proof −
have *1* ≤ *N* **by** *fact*
also have *N* = ($\sum$ *m*∈{*0*<..*N*}. *1*) **by** *simp*
also have ($\sum$ *m*∈{*0*<..*N*}. *1*) ≤ ($\sum$ *m*∈{*0*<..*N*}. *totient m*)
by (*intro sum-mono*) (*simp-all add*: *Suc-le-eq*)
finally show *?thesis* **.**
qed

have (λ*N*. *real* (*card* (*A N*)) − *6* / *pi*^{2} ∗ (*real N*)^2) ∈
Θ(λ*N*. *2* ∗ (*sum-upto* (λ*m*. *real* (*totient m*)) (*real N*) − (*3* / *pi*^{2} ∗ (*real N*)^2)) − *1*)
(**is** - ∈ Θ(*?f*)) **using** ∗ ∗∗
by (*intro bigthetaI-cong eventually-mono* [*OF eventually-gt-at-top*[*of 0*::*nat*]])
(*auto simp*: *of-nat-diff sum-upto-altdef*)
also have *?f* ∈ *O*(λ*N*. *real N* ∗ *ln* (*real N*))
proof (*rule landau-o.big.compose*[*OF - filterlim-real-sequentially*], *rule sum-in-bigo*)
show (λ*x*. *2* ∗ (*sum-upto* (λ*m*. *real* (*totient m*)) *x* − *3* / *pi*^{2} ∗ *x*^{2})) ∈ *O*(λ*x*. *x* ∗ *ln x*)
by (*subst landau-o.big.cmult-in-iff*, *simp*, *rule summatory-totient-asymptotics*)
qed *simp-all*
finally show *?thesis* **.**
qed

theorem *coprime-pairs-asymptotics′*:
(λ*N*. *real* (*card* (*A N*))) =*o* (λ*N*. *6* / *pi*^{2} ∗ (*real N*)^2) +*o* *O*(λ*N*. *real N* ∗ *ln* (*real N*))
using *coprime-pairs-asymptotics*
by (*subst set-minus-plus* [*symmetric*]) (*simp-all add*: *fun-diff-def*)

theorem *coprime-pairs-asymptotics′′*:
(λ*N*. *real* (*card* (*A N*))) ∼[*at-top*] (λ*N*. *6* / *pi*^{2} ∗ (*real N*)^2)
proof −
have (λ*N*. *real* (*card* (*A N*)) − *6* / *pi*^{2} ∗ (*real N*) ^ *2*) ∈ *O*(λ*N*. *real N* ∗ *ln* (*real N*))
by (*rule coprime-pairs-asymptotics*)
also have (λ*N*. *real N* ∗ *ln* (*real N*)) ∈ *o*(λ*N*. *6* / *pi* ^ *2* ∗ *real N* ^ *2*)
by (*rule landau-o.small.compose*[*OF - filterlim-real-sequentially*]) *simp*
finally show *?thesis* **by** (*simp add*: *asymp-equiv-altdef*)
qed

theorem *coprime-probability-tendsto*:

($\lambda N.$ *card* (*A N*) / *card* ({*1..N*} × {*1..N*})) ⟶ *6* / pi^2
proof −
have ($\lambda N.$ *6* / *pi* ^ *2*) ∼[*at-top*] ($\lambda N.$ *6* / *pi* ^ *2* ∗ *real N* ^ *2* / *real N* ^ *2*)
using *eventually-gt-at-top*[*of 0*::*nat*]
by (*intro asymp-equiv-refl-ev*) (*auto elim*!: *eventually-mono*)
also have . . . ∼[*at-top*] ($\lambda N.$ *real* (*card* (*A N*)) / *real N* ^ *2*)
by (*intro asymp-equiv-intros asymp-equiv-symI*[*OF coprime-pairs-asymptotics′*])
also have . . . ∼[*at-top*] ($\lambda N.$ *real* (*card* (*A N*)) / *real* (*card* ({*1..N*} × {*1..N*})))
by (*simp add*: *power2-eq-square*)
finally have . . . ∼[*at-top*] (λ-. *6* / *pi* ^ *2*) **by** (*simp add*: *asymp-equiv-sym*)
thus *?thesis* **by** (*rule asymp-equivD-const*)
qed

end

13.6 The asymptotics of the number of Farey fractions

definition *farey-fractions* :: *nat* ⇒ *rat set* **where**
farey-fractions N = {*q* :: *rat* ∈ {*0*<*..1*}. *snd* (*quotient-of q*) ≤ *int N*}

lemma *Fract-eq-coprime*:
assumes *Rat.Fract a b* = *Rat.Fract c d b* > *0 d* > *0 coprime a b coprime c d*
shows *a* = *c b* = *d*
proof −
from *assms* **have** *a* ∗ *d* = *c* ∗ *b* **by** (*auto simp*: *eq-rat*)
hence *abs* (*a* ∗ *d*) = *abs* (*c* ∗ *b*) **by** (*simp only*:)
hence *abs a* ∗ *abs d* = *abs c* ∗ *abs b* **by** (*simp only*: *abs-mult*)
also have *?this* ⟷ *abs a* = *abs c* ∧ *d* = *b*
using *assms* **by** (*subst coprime-crossproduct-int*) *simp-all*
finally show *b* = *d* **by** *simp*
with ‹*a* ∗ *d* = *c* ∗ *b*› **and** ‹*b* > *0*› **show** *a* = *c* **by** *simp*
qed

lemma *quotient-of-split*:
P (*quotient-of q*) = (∀ *a b*. *b* > *0* ⟶ *coprime a b* ⟶ *q* = *Rat.Fract a b* ⟶ *P* (*a*, *b*))
by (*cases q*) (*auto simp*: *quotient-of-Fract dest*: *Fract-eq-coprime*)

lemma *quotient-of-split-asm*:
P (*Rat.quotient-of q*) = (¬(∃ *a b*. *b* > *0* ∧ *coprime a b* ∧ *q* = *Rat.Fract a b* ∧ ¬*P* (*a*, *b*)))
using *quotient-of-split*[*of P q*] **by** *blast*

lemma *farey-fractions-bij*:
bij-betw (λ(*a*,*b*). *Rat.Fract* (*int a*) (*int b*))
{(*a*,*b*)|*a b*. *0* < *a* ∧ *a* ≤ *b* ∧ *b* ≤ *N* ∧ *coprime a b*} (*farey-fractions N*)
proof (*rule bij-betwI*[*of - - -* $\lambda q.$ *case quotient-of q of* (*a*, *b*) ⇒ (*nat a*, *nat b*)], *goal-cases*)
case *1*

show *?case*
by (*auto simp*: *farey-fractions-def Rat.zero-less-Fract-iff Rat.Fract-le-one-iff Rat.quotient-of-Fract Rat.normalize-def gcd-int-def Let-def*)
next
case *2*
show *?case*
by (*auto simp add*: *farey-fractions-def Rat.Fract-le-one-iff Rat.zero-less-Fract-iff split*: *prod.splits quotient-of-split-asm*)
(*simp add*: *coprime-int-iff* [*symmetric*])
next
case (*3 x*)
thus *?case* **by** (*auto simp*: *Rat.quotient-of-Fract Rat.normalize-def Let-def gcd-int-def*)
next
case (*4 x*)
thus *?case* **unfolding** *farey-fractions-def*
by (*split quotient-of-split*) (*auto simp*: *Rat.zero-less-Fract-iff*)
qed

lemma *card-farey-fractions*: *card* (*farey-fractions N*) = *sum totient* {*0*<..*N*}
proof −
have *card* (*farey-fractions N*) = *card* {(*a*,*b*)|*a b*. *0* < *a* ∧ *a* ≤ *b* ∧ *b* ≤ *N* ∧ *coprime a b*}
using *farey-fractions-bij* **by** (*rule bij-betw-same-card* [*symmetric*])
also have {(*a*,*b*)|*a b*. *0* < *a* ∧ *a* ≤ *b* ∧ *b* ≤ *N* ∧ *coprime a b*} =
(⋃ *b*∈{*0*<..*N*}. (λ*a*. (*a*, *b*)) ‘ *totatives b*)
by (*auto simp*: *totatives-def image-iff*)
also have *card* ... = (∑ *b*∈{*0*<..*N*}. *card* ((λ*a*. (*a*, *b*)) ‘ *totatives b*))
by (*intro card-UN-disjoint*) *auto*
also have ... = (∑ *b*∈{*0*<..*N*}. *totient b*)
unfolding *totient-def* **by** (*intro sum.cong refl card-image*) (*auto simp*: *inj-on-def*)
finally show *?thesis* **.**
qed

lemma *card-farey-fractions-asymptotics*:
(λ*N*. *real* (*card* (*farey-fractions N*)) − *3* / *pi*^{2} ∗ (*real N*)^2) ∈ *O*(λ*N*. *real N* ∗ *ln* (*real N*))
proof −
have (λ*N*. *sum-upto* (λ*n*. *real* (*totient n*)) (*real N*) − *3* / *pi*^{2} ∗ (*real N*)^2)
∈ *O*(λ*N*. *real N* ∗ *ln* (*real N*)) (**is** *?f* ∈ -)
using *summatory-totient-asymptotics filterlim-real-sequentially*
by (*rule landau-o.big.compose*)
also have *?f* = (λ*N*. *real* (*card* (*farey-fractions N*)) − *3* / *pi*^{2} ∗ (*real N*)^2)
by (*intro ext*) (*simp add*: *sum-upto-altdef card-farey-fractions*)
finally show *?thesis* **.**
qed

theorem *card-farey-fractions-asymptotics′*:
(λ*N*. *card* (*farey-fractions N*)) =*o* (λ*N*. *3* / *pi*^{2} ∗ *N*^*2*) +*o* *O*(λ*N*. *N* ∗ *ln N*)
using *card-farey-fractions-asymptotics*

by (*subst set-minus-plus* [*symmetric*]) (*simp-all add*: *fun-diff-def*)

theorem *card-farey-fractions-asymptotics''*:
(λ*N*. *real* (*card* (*farey-fractions N*))) ∼[*at-top*] (λ*N*. *3* / *pi*^{2} ∗ (*real N*)^2)
proof −
have (λ*N*. *real* (*card* (*farey-fractions N*)) − *3* / *pi*^{2} ∗ (*real N*) ^ *2*) ∈ *O*(λ*N*. *real N* ∗ *ln* (*real N*))
by (*rule card-farey-fractions-asymptotics*)
also have (λ*N*. *real N* ∗ *ln* (*real N*)) ∈ *o*(λ*N*. *3* / *pi* ^ *2* ∗ *real N* ^ *2*)
by (*rule landau-o.small.compose*[*OF* - *filterlim-real-sequentially*]) *simp*
finally show *?thesis* **by** (*simp add*: *asymp-equiv-altdef*)
qed

end

14 Efficient code for number-theoretic functions

theory *Dirichlet-Efficient-Code*
imports
Main
Moebius-Mu
More-Totient
Divisor-Count
Liouville-Lambda
HOL−Library.Code-Target-Numeral
Polynomial-Factorization.Prime-Factorization
begin

definition *prime-factorization-nat'* :: *nat* ⇒ (*nat* × *nat*) *list* **where**
prime-factorization-nat' n = (
let ps = *prime-factorization-nat n*
in map (λ*p*. (*p*, *length* (*filter* ((=) *p*) *ps*) − *1*)) (*remdups-adj* (*sort ps*)))

lemma *set-prime-factorization-nat'*:
set (*prime-factorization-nat' n*) = (λ*p*. (*p*, *multiplicity p n* − *1*)) ‘ *prime-factors n*
proof (*intro equalityI subsetI*; *clarify*)
fix *p k* :: *nat*
assume *pk*: (*p*, *k*) ∈ *set* (*prime-factorization-nat' n*)
hence *p*: *p* ∈ *prime-factors n*
by (*auto simp*: *prime-factorization-nat'-def Let-def multiset-prime-factorization-nat-correct*)
hence *p'*: *prime p* **by** (*simp add*: *prime-factors-multiplicity*)
from *pk p'* **have** *k* = *multiplicity p n* − *1*
by (*auto simp*: *prime-factorization-nat'-def Let-def multiset-prime-factorization-nat-correct count-prime-factorization-prime* [*symmetric*] *count-mset*)
with *p* **show** (*p*, *k*) ∈ (λ*p*. (*p*, *multiplicity p n* − *1*)) ‘ *prime-factors n* **by** *auto*
next
fix *p* :: *nat*
assume *p* ∈ *prime-factors n*

moreover from *this* **have** *prime p* **by** (*simp add*: *prime-factors-multiplicity*)
ultimately show (*p*, *multiplicity p n* − *1*) ∈ *set* (*prime-factorization-nat′ n*)
by (*auto simp*: *prime-factorization-nat′-def Let-def multiset-prime-factorization-nat-correct*

count-prime-factorization-prime [*symmetric*] *count-mset*)
qed

lemma *distinct-prime-factorization-nat′* [*simp*]: *distinct* (*prime-factorization-nat′ n*)
by (*simp add*: *distinct-map inj-on-def prime-factorization-nat′-def Let-def*)

lemmas (**in** *multiplicative-function′*) *efficient-code′* =
efficient-code [*of* λ-. *prime-factorization-nat′ n n* **for** *n*,
OF set-prime-factorization-nat′ distinct-prime-factorization-nat′]

14.1 Möbius μ function

definition *moebius-mu-aux* :: *nat* ⇒ (*unit* ⇒ *nat list*) ⇒ *int* **where**
moebius-mu-aux n ps =
(*if n* ≠ *0* ∧ ¬*4 dvd n* ∧ ¬*9 dvd n then*
(*let ps* = *ps* () *in if distinct ps then if even* (*length ps*) *then 1 else* −*1 else 0*) *else 0*)

lemma *moebius-mu-conv-moebius-mu-aux*:
fixes *qs* :: *unit* ⇒ *nat list*
defines *ps* ≡ *qs* ()
assumes *mset ps* = *prime-factorization n*
shows *moebius-mu n* = *of-int* (*moebius-mu-aux n qs*)
proof (*cases n* = *0* ∨ *4 dvd n* ∨ *9 dvd n*)
case *False*
hence [*simp*]: *n* > *0* **by** *auto*
have *set-mset* (*mset ps*) = *prime-factors n* **by** (*subst assms*) *simp*
hence [*simp*]: *set ps* = *prime-factors n* **by** *simp*
show *?thesis*
proof (*cases distinct ps*)
case *True*
have *multiplicity p n* = *1* **if** *p*: *p* ∈ *prime-factors n* **for** *p*
proof −
from *p* **and** *True* **have** *count* (*mset ps*) *p* = *1* **by** (*auto simp*: *distinct-count-atmost-1*)
also from *assms* **and** *p* **have** *count* (*mset ps*) *p* = *multiplicity p n*
by (*simp add*: *prime-factors-multiplicity count-prime-factorization-prime*)
finally show *multiplicity p n* = *1* **.**
qed
moreover from *True* **have** *card* (*prime-factors n*) = *length ps*
by (*simp only*: *assms* [*symmetric*] *set-mset-mset distinct-card*)
ultimately show *?thesis* **using** *False* **and** *True*
by (*auto simp add*: *moebius-mu-def moebius-mu-aux-def ps-def*
Let-def squarefree-factorial-semiring′)
next

case *False*
then obtain *p* **where** *count* (*mset ps*) *p* ≠ (*if p* ∈ *set ps then 1 else 0*)
by (*subst* (*asm*) *distinct-count-atmost-1*) *auto*
moreover from *this* **have** *p*: *p* ∈ *prime-factors n*
by (*cases count* (*mset ps*) *p* = *0*) (*auto split*: *if-splits*)
ultimately have *count* (*mset ps*) *p* > *1* **by** (*cases count* (*mset ps*) *p*) *auto*
with *p* **and** *assms* **have** *multiplicity p n* > *1*
by (*simp add*: *prime-factors-multiplicity count-prime-factorization-prime*)
with *False* **and** *assms* **and** *p* **have** ¬*squarefree n*
by (*auto simp*: *squarefree-factorial-semiring''*)
with *False* **and** *assms* **and** *p* **show** *?thesis*
by (*auto simp*: *moebius-mu-def moebius-mu-aux-def*)
qed
next
case *True*
with *not-squarefreeI*[*of 2 n*] **and** *not-squarefreeI*[*of 3 n*] **show** *?thesis*
by (*auto simp*: *moebius-mu-aux-def*)
qed

lemma *moebius-mu-code* [*code*]:
moebius-mu n = *of-int* (*moebius-mu-aux n* (λ-. *prime-factorization-nat n*))
by (*rule moebius-mu-conv-moebius-mu-aux*) (*simp-all add*: *multiset-prime-factorization-nat-correct*)

value *moebius-mu 12578972695257* :: *int*

14.2 Euler's ϕ function

primrec *totient-aux1* :: *nat* ⇒ *nat list* ⇒ *nat* **where**
totient-aux1 n [] = *n*
| *totient-aux1 n* (*p* # *ps*) = *totient-aux1* (*n* − *n div p*) *ps*

lemma *of-nat-totient-aux1*:
assumes ⋀*p*. *p* ∈ *set ps* ⟹ *prime p* ⋀*p*. *p* ∈ *set ps* ⟹ *p dvd n distinct ps*
shows *real* (*totient-aux1 n ps*) = *real n* ∗ (∏*p*∈*set ps*. *1* − *1* / *real p*)
using *assms*
proof (*induction ps arbitrary*: *n*)
case (*Cons p ps n*)
from *Cons.prems* **have** *p*: *prime p p dvd n* **by** *auto*
have *real* (*totient-aux1 n* (*p* # *ps*)) = *real* (*totient-aux1* (*n* − *n div p*) *ps*) **by** *simp*
also have . . . = *real* (*n* − *n div p*) ∗ (∏*p*∈*set ps*. *1* − *1* / *real p*)
proof (*rule Cons.IH*)
fix *q* **assume** *q*: *q* ∈ *set ps*
define *m* **where** *m* = *n div p*
from *p* **have** *m*: *n* = *p* ∗ *m* **by** (*simp add*: *m-def*)
from *Cons.prems q* **have** *prime q q dvd n p* ≠ *q* **by** *auto*
hence *q dvd m* **using** *primes-dvd-imp-eq*[*of q p*] *p* **by** (*auto simp add*: *m prime-dvd-mult-iff*)
thus *q dvd n* − *n div p* **unfolding** *m-def* **using** *p* ‹*q dvd n*› **by** *simp*

qed (*insert Cons.prems*, *auto*)
also have *real* (*n* − *n div p*) = *real n* ∗ (*1* − *1* / *real p*)
by (*simp add*: *of-nat-diff real-of-nat-div p field-simps*)
also have ... ∗ (∏ *p*∈*set ps*. *1* − *1* / *real p*) = *real n* ∗ (∏ *p*∈*set* (*p*#*ps*). *1* − *1* / *real p*)
using *Cons.prems* **by** *simp*
finally show *?case* **.**
qed *simp-all*

lemma *totient-conv-totient-aux1*:
assumes *set ps* = *prime-factors n distinct ps*
shows *totient n* = *totient-aux1 n ps*
proof −
from *assms* **have** *real* (*totient-aux1 n ps*) = *real n* ∗ (∏ *p*∈*set ps*. *1* − *1* / *real p*)
by (*intro of-nat-totient-aux1*) *auto*
also have *set ps* = *prime-factors n* **by** *fact*
also have *real n* ∗ (∏ *p*∈*prime-factors n*. *1* − *1* / *real p*) = *real* (*totient n*)
by (*rule totient-formula2* [*symmetric*])
finally show *?thesis* **by** (*simp only*: *of-nat-eq-iff*)
qed

definition *prime-factors-nat* :: *nat* ⇒ *nat list* **where**
prime-factors-nat n = *remdups-adj* (*sort* (*prime-factorization-nat n*))

lemma *set-prime-factors-nat* [*simp*]: *set* (*prime-factors-nat n*) = *prime-factors n*
unfolding *prime-factors-nat-def multiset-prime-factorization-nat-correct* **by** *simp*

lemma *distinct-prime-factors-nat* [*simp*]: *distinct* (*prime-factors-nat n*)
by (*simp add*: *prime-factors-nat-def*)

definition *totient-aux2* :: (*nat* × *nat*) *list* ⇒ *nat* **where**
totient-aux2 xs = (∏ (*p*,*k*)←*xs*. *p* ^ *k* ∗ (*p* − *1*))

lemma *totient-conv-totient-aux2*:
assumes *n* ≠ *0*
assumes *set xs* = (λ*p*. (*p*, *multiplicity p n* − *1*)) ‘ *prime-factors n*
assumes *distinct xs*
shows *totient n* = *totient-aux2 xs*
proof −
have *totient-aux2 xs* = (∏ (*p*,*k*)←*xs*. *p* ^ *k* ∗ (*p* − *1*)) **by** (*fact totient-aux2-def*)
also from *assms* **have** ... =
(∏ *x*∈(λ*p*. (*p*, *multiplicity p n* − *1*)) ‘ *prime-factors n*. *case x of* (*p*, *k*) ⇒ *p* ^ *k* ∗ (*p* − *Suc 0*))
by (*subst prod.distinct-set-conv-list* [*symmetric*]) *simp-all*
also have ... = (∏ *p*∈*prime-factors n*. *p* ^ (*multiplicity p n* − *1*) ∗ (*p* − *Suc 0*))
by (*subst prod.reindex*) (*auto simp*: *inj-on-def*)

also have ... = ($\prod$ *p*∈*prime-factors n*. *p* ^ *multiplicity p n* − *p* ^ (*multiplicity p n* − *1*))
by (*intro prod.cong refl*) (*auto simp*: *prime-factors-multiplicity algebra-simps power-Suc* [*symmetric*] *simp del*: *power-Suc*)
also have ... = *totient n* **using** *assms*(*1*) **by** (*subst totient.prod-prime-factors′*) *auto*
finally show *?thesis* **..**
qed

lemma *totient-code1*: *totient n* = *totient-aux1 n* (*prime-factors-nat n*)
by (*intro totient-conv-totient-aux1*) *simp-all*

lemma *totient-code2*: *totient n* = (*if n* = *0 then 0 else totient-aux2* (*prime-factorization-nat′ n*))
by (*simp-all add*: *set-prime-factorization-nat′ totient-conv-totient-aux2 split*: *if-splits*)

declare *totient-code-naive* [*code del*]

lemmas [*code*] = *totient-code2*

value *totient 125789726827482323235784*

14.3 Divisor Functions

lemmas [*code del*] = *divisor-count-naive divisor-sum-naive*
lemmas [*code*] = *divisor-count.efficient-code′ divisor-sum.efficient-code′*

value *int* (*divisor-count 378568418621*)
value *int* (*divisor-sum 378568418621*)

14.4 Liouville's λ function

lemma [*code*]: *liouville-lambda n* =
(*if n* = *0 then 0 else if even* (*length* (*prime-factorization-nat n*)) *then 1 else* −*1*)
by (*auto simp*: *liouville-lambda-def multiset-prime-factorization-nat-correct*)

value *liouville-lambda 1264785343674* :: *int*

end

References

[1] T. M. Apostol. *Introduction to Analytic Number Theory*. Undergraduate Texts in Mathematics. Springer-Verlag, 1976.