Number Theory

Jeremy Avigad, Kevin Donnelly, David Gray, Adam Kramer,
Lawrence C. Paulson, Paul Raff, Thomas M. Rasmussen, Christophe Tabaczync

September 10, 2004

Contents

1 Permutations

theory *Permutation* = *Main*:

consts
perm :: (*'a list* ∗ *'a list*) *set*

syntax
-perm :: *'a list* => *'a list* => *bool* (- <~~> - [*50*, *50*] *50*)
translations
x <~~> *y* == (*x*, *y*) ∈ *perm*

inductive *perm*
intros
Nil [*intro*!]: [] <~~> []
swap [*intro*!]: *y* # *x* # *l* <~~> *x* # *y* # *l*
Cons [*intro*!]: *xs* <~~> *ys* ==> *z* # *xs* <~~> *z* # *ys*
trans [*intro*]: *xs* <~~> *ys* ==> *ys* <~~> *zs* ==> *xs* <~~> *zs*

lemma *perm-refl* [*iff*]: *l* <~~> *l*
apply (*induct l*)
apply *auto*
done

1.1 Some examples of rule induction on permutations

lemma *xperm-empty-imp-aux*: *xs* <~~> *ys* ==> *xs* = [] --> *ys* = []
— the form of the premise lets the induction bind *xs* and *ys*
apply (*erule perm.induct*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *xperm-empty-imp*: [] <~~> *ys* ==> *ys* = []
apply (*insert xperm-empty-imp-aux*)
apply *blast*
done

This more general theorem is easier to understand!

lemma *perm-length*: *xs* <~~> *ys* ==> *length xs* = *length ys*
apply (*erule perm.induct*)
apply *simp-all*
done

lemma *perm-empty-imp*: [] <~~> *xs* ==> *xs* = []

apply (*drule perm-length*)
apply *auto*
done

lemma *perm-sym*: *xs* <~~> *ys* ==> *ys* <~~> *xs*
apply (*erule perm.induct*)
apply *auto*
done

lemma *perm-mem* [*rule-format*]: *xs* <~~> *ys* ==> *x mem xs* −−> *x mem ys*
apply (*erule perm.induct*)
apply *auto*
done

1.2 Ways of making new permutations

We can insert the head anywhere in the list.

lemma *perm-append-Cons*: *a* # *xs* @ *ys* <~~> *xs* @ *a* # *ys*
apply (*induct xs*)
apply *auto*
done

lemma *perm-append-swap*: *xs* @ *ys* <~~> *ys* @ *xs*
apply (*induct xs*)
apply *simp-all*
apply (*blast intro*: *perm-append-Cons*)
done

lemma *perm-append-single*: *a* # *xs* <~~> *xs* @ [*a*]
apply (*rule perm.trans*)
prefer *2*
apply (*rule perm-append-swap*)
apply *simp*
done

lemma *perm-rev*: *rev xs* <~~> *xs*
apply (*induct xs*)
apply *simp-all*
apply (*blast intro*!: *perm-append-single intro*: *perm-sym*)
done

lemma *perm-append1*: *xs* <~~> *ys* ==> *l* @ *xs* <~~> *l* @ *ys*
apply (*induct l*)
apply *auto*

done

lemma *perm-append2*: *xs* <~~> *ys* ==> *xs* @ *l* <~~> *ys* @ *l*
apply (*blast intro*!: *perm-append-swap perm-append1*)
done

1.3 Further results

lemma *perm-empty* [*iff*]: ([] <~~> *xs*) = (*xs* = [])
apply (*blast intro*: *perm-empty-imp*)
done

lemma *perm-empty2* [*iff*]: (*xs* <~~> []) = (*xs* = [])
apply *auto*
apply (*erule perm-sym* [*THEN perm-empty-imp*])
done

lemma *perm-sing-imp* [*rule-format*]: *ys* <~~> *xs* ==> *xs* = [*y*] --> *ys* = [*y*]
apply (*erule perm.induct*)
apply *auto*
done

lemma *perm-sing-eq* [*iff*]: (*ys* <~~> [*y*]) = (*ys* = [*y*])
apply (*blast intro*: *perm-sing-imp*)
done

lemma *perm-sing-eq2* [*iff*]: ([*y*] <~~> *ys*) = (*ys* = [*y*])
apply (*blast dest*: *perm-sym*)
done

1.4 Removing elements

consts
remove :: *'a* => *'a list* => *'a list*
primrec
remove x [] = []
remove x (*y* # *ys*) = (*if x* = *y then ys else y* # *remove x ys*)

lemma *perm-remove*: *x* ∈ *set ys* ==> *ys* <~~> *x* # *remove x ys*
apply (*induct ys*)
apply *auto*
done

lemma *remove-commute*: *remove x* (*remove y l*) = *remove y* (*remove x l*)
apply (*induct l*)

apply *auto*
done

Congruence rule

lemma *perm-remove-perm*: *xs* <~~> *ys* ==> *remove z xs* <~~> *remove z ys*
apply (*erule perm.induct*)
apply *auto*
done

lemma *remove-hd* [*simp*]: *remove z* (*z* # *xs*) = *xs*
apply *auto*
done

lemma *cons-perm-imp-perm*: *z* # *xs* <~~> *z* # *ys* ==> *xs* <~~> *ys*
apply (*drule-tac z* = *z* **in** *perm-remove-perm*)
apply *auto*
done

lemma *cons-perm-eq* [*iff*]: (*z*#*xs* <~~> *z*#*ys*) = (*xs* <~~> *ys*)
apply (*blast intro*: *cons-perm-imp-perm*)
done

lemma *append-perm-imp-perm*: !!*xs ys*. *zs* @ *xs* <~~> *zs* @ *ys* ==> *xs* <~~> *ys*
apply (*induct zs rule*: *rev-induct*)
apply (*simp-all* (*no-asm-use*))
apply *blast*
done

lemma *perm-append1-eq* [*iff*]: (*zs* @ *xs* <~~> *zs* @ *ys*) = (*xs* <~~> *ys*)
apply (*blast intro*: *append-perm-imp-perm perm-append1*)
done

lemma *perm-append2-eq* [*iff*]: (*xs* @ *zs* <~~> *ys* @ *zs*) = (*xs* <~~> *ys*)
apply (*safe intro*!: *perm-append2*)
apply (*rule append-perm-imp-perm*)
apply (*rule perm-append-swap* [*THEN perm.trans*])
— the previous step helps this *blast* call succeed quickly
apply (*blast intro*: *perm-append-swap*)
done

end

2 The Greatest Common Divisor and Euclid's algorithm

theory *Primes* = *Main*:

See [**?**].

consts
  gcd :: *nat* × *nat* => *nat* — Euclid's algorithm

recdef *gcd* *measure* ((λ(*m*, *n*). *n*) :: *nat* × *nat* => *nat*)
  gcd (*m*, *n*) = (*if n = 0 then m else gcd* (*n*, *m mod n*))

constdefs
  is-gcd :: *nat* => *nat* => *nat* => *bool* — *gcd* as a relation
  is-gcd p m n == *p dvd m* ∧ *p dvd n* ∧
    (∀ *d*. *d dvd m* ∧ *d dvd n* −−> *d dvd p*)

  coprime :: *nat* => *nat* => *bool*
  coprime m n == *gcd* (*m*, *n*) = *1*

  prime :: *nat set*
  prime == {*p*. *1* < *p* ∧ (∀ *m*. *m dvd p* −−> *m* = *1* ∨ *m* = *p*)}

lemma *gcd-induct*:
  (!!*m*. *P m 0*) ==>
    (!!*m n*. *0* < *n* ==> *P n* (*m mod n*) ==> *P m n*)
  ==> *P* (*m*::*nat*) (*n*::*nat*)
  apply (*induct m n rule*: *gcd.induct*)
  apply (*case-tac n* = *0*)
   apply *simp-all*
  done

lemma *gcd-0* [*simp*]: *gcd* (*m*, *0*) = *m*
  apply *simp*
  done

lemma *gcd-non-0*: *0* < *n* ==> *gcd* (*m*, *n*) = *gcd* (*n*, *m mod n*)
  apply *simp*
  done

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

lemma *gcd-1* [*simp*]: *gcd* (*m*, *Suc 0*) = *1*
apply (*simp add*: *gcd-non-0*)
done

gcd (*m*, *n*) divides *m* and *n*. The conjunctions don't seem provable separately.

lemma *gcd-dvd1* [*iff*]: *gcd* (*m*, *n*) *dvd m*
and *gcd-dvd2* [*iff*]: *gcd* (*m*, *n*) *dvd n*
apply (*induct m n rule*: *gcd-induct*)
apply (*simp-all add*: *gcd-non-0*)
apply (*blast dest*: *dvd-mod-imp-dvd*)
done

Maximality: for all *m*, *n*, *k* naturals, if *k* divides *m* and *k* divides *n* then *k* divides *gcd* (*m*, *n*).

lemma *gcd-greatest*: *k dvd m* ==> *k dvd n* ==> *k dvd gcd* (*m*, *n*)
apply (*induct m n rule*: *gcd-induct*)
apply (*simp-all add*: *gcd-non-0 dvd-mod*)
done

lemma *gcd-greatest-iff* [*iff*]: (*k dvd gcd* (*m*, *n*)) = (*k dvd m* ∧ *k dvd n*)
apply (*blast intro*!: *gcd-greatest intro*: *dvd-trans*)
done

lemma *gcd-zero*: (*gcd* (*m*, *n*) = *0*) = (*m* = *0* ∧ *n* = *0*)
by (*simp only*: *dvd-0-left-iff* [*THEN sym*] *gcd-greatest-iff*)

Function gcd yields the Greatest Common Divisor.

lemma *is-gcd*: *is-gcd* (*gcd* (*m*, *n*)) *m n*
apply (*simp add*: *is-gcd-def gcd-greatest*)
done

Uniqueness of GCDs.

lemma *is-gcd-unique*: *is-gcd m a b* ==> *is-gcd n a b* ==> *m* = *n*
apply (*simp add*: *is-gcd-def*)
apply (*blast intro*: *dvd-anti-sym*)
done

lemma *is-gcd-dvd*: *is-gcd m a b* ==> *k dvd a* ==> *k dvd b* ==> *k dvd m*
apply (*auto simp add*: *is-gcd-def*)

done

Commutativity

lemma *is-gcd-commute*: *is-gcd k m n* = *is-gcd k n m*
apply (*auto simp add*: *is-gcd-def*)
done

lemma *gcd-commute*: *gcd* (*m*, *n*) = *gcd* (*n*, *m*)
apply (*rule is-gcd-unique*)
apply (*rule is-gcd*)
apply (*subst is-gcd-commute*)
apply (*simp add*: *is-gcd*)
done

lemma *gcd-assoc*: *gcd* (*gcd* (*k*, *m*), *n*) = *gcd* (*k*, *gcd* (*m*, *n*))
apply (*rule is-gcd-unique*)
apply (*rule is-gcd*)
apply (*simp add*: *is-gcd-def*)
apply (*blast intro*: *dvd-trans*)
done

lemma *gcd-0-left* [*simp*]: *gcd* (*0*, *m*) = *m*
apply (*simp add*: *gcd-commute* [*of 0*])
done

lemma *gcd-1-left* [*simp*]: *gcd* (*Suc 0*, *m*) = *1*
apply (*simp add*: *gcd-commute* [*of Suc 0*])
done

Multiplication laws

lemma *gcd-mult-distrib2*: *k* ∗ *gcd* (*m*, *n*) = *gcd* (*k* ∗ *m*, *k* ∗ *n*)
— [**?**, page 27]
apply (*induct m n rule*: *gcd-induct*)
apply *simp*
apply (*case-tac k* = *0*)
apply (*simp-all add*: *mod-geq gcd-non-0 mod-mult-distrib2*)
done

lemma *gcd-mult* [*simp*]: *gcd* (*k*, *k* ∗ *n*) = *k*
apply (*rule gcd-mult-distrib2* [*of k 1 n*, *simplified*, *symmetric*])
done

lemma *gcd-self* [*simp*]: *gcd* (*k*, *k*) = *k*

apply (*rule gcd-mult* [*of k 1*, *simplified*])
done

lemma *relprime-dvd-mult*: *gcd* (*k*, *n*) = *1* ==> *k dvd m* * *n* ==> *k dvd m*
apply (*insert gcd-mult-distrib2* [*of m k n*])
apply *simp*
apply (*erule-tac t* = *m* **in** *ssubst*)
apply *simp*
done

lemma *relprime-dvd-mult-iff*: *gcd* (*k*, *n*) = *1* ==> (*k dvd m* * *n*) = (*k dvd m*)
apply (*blast intro*: *relprime-dvd-mult dvd-trans*)
done

lemma *prime-imp-relprime*: *p* ∈ *prime* ==> ¬ *p dvd n* ==> *gcd* (*p*, *n*) = *1*
apply (*auto simp add*: *prime-def*)
apply (*drule-tac x* = *gcd* (*p*, *n*) **in** *spec*)
apply *auto*
apply (*insert gcd-dvd2* [*of p n*])
apply *simp*
done

lemma *two-is-prime*: *2* ∈ *prime*
apply (*auto simp add*: *prime-def*)
apply (*case-tac m*)
apply (*auto dest*!: *dvd-imp-le*)
done

This theorem leads immediately to a proof of the uniqueness of factorization. If *p* divides a product of primes then it is one of those primes.

lemma *prime-dvd-mult*: *p* ∈ *prime* ==> *p dvd m* * *n* ==> *p dvd m* ∨ *p dvd n*
by (*blast intro*: *relprime-dvd-mult prime-imp-relprime*)

lemma *prime-dvd-square*: *p* ∈ *prime* ==> *p dvd m*^*Suc* (*Suc 0*) ==> *p dvd m*
by (*auto dest*: *prime-dvd-mult*)

lemma *prime-dvd-power-two*: *p* ∈ *prime* ==> *p dvd* m^2 ==> *p dvd m*
by (*rule prime-dvd-square*) (*simp-all add*: *power2-eq-square*)

Addition laws

lemma *gcd-add1* [*simp*]: *gcd* (*m* + *n*, *n*) = *gcd* (*m*, *n*)
apply (*case-tac n* = *0*)
apply (*simp-all add*: *gcd-non-0*)
done

lemma *gcd-add2* [*simp*]: *gcd* (*m*, *m* + *n*) = *gcd* (*m*, *n*)
apply (*rule gcd-commute* [*THEN trans*])
apply (*subst add-commute*)
apply (*simp add*: *gcd-add1*)
apply (*rule gcd-commute*)
done

lemma *gcd-add2'* [*simp*]: *gcd* (*m*, *n* + *m*) = *gcd* (*m*, *n*)
apply (*subst add-commute*)
apply (*rule gcd-add2*)
done

lemma *gcd-add-mult*: *gcd* (*m*, *k* ∗ *m* + *n*) = *gcd* (*m*, *n*)
apply (*induct k*)
apply (*simp-all add*: *gcd-add2 add-assoc*)
done

More multiplication laws

lemma *gcd-mult-cancel*: *gcd* (*k*, *n*) = *1* ==> *gcd* (*k* ∗ *m*, *n*) = *gcd* (*m*, *n*)
apply (*rule dvd-anti-sym*)
apply (*rule gcd-greatest*)
apply (*rule-tac n* = *k* **in** *relprime-dvd-mult*)
apply (*simp add*: *gcd-assoc*)
apply (*simp add*: *gcd-commute*)
apply (*simp-all add*: *mult-commute gcd-dvd1 gcd-dvd2*)
apply (*blast intro*: *gcd-dvd1 dvd-trans*)
done

end

3 The Fibonacci function

theory *Fib* = *Primes*:

Fibonacci numbers: proofs of laws taken from: R. L. Graham, D. E. Knuth, O. Patashnik. Concrete Mathematics. (Addison-Wesley, 1989)

consts *fib* :: *nat* => *nat*
recdef *fib* *less-than*
zero: *fib 0* = *0*
one: *fib* (*Suc 0*) = *Suc 0*

Suc-Suc: *fib* (*Suc* (*Suc x*)) = *fib x* + *fib* (*Suc x*)

The difficulty in these proofs is to ensure that the induction hypotheses are applied before the definition of *fib*. Towards this end, the *fib* equations are not declared to the Simplifier and are applied very selectively at first.

declare *fib.Suc-Suc* [*simp del*]

lemma *fib-Suc3*: *fib* (*Suc* (*Suc* (*Suc n*))) = *fib* (*Suc n*) + *fib* (*Suc* (*Suc n*))
apply (*rule fib.Suc-Suc*)
done

Concrete Mathematics, page 280

lemma *fib-add*: *fib* (*Suc* (*n* + *k*)) = *fib* (*Suc k*) ∗ *fib* (*Suc n*) + *fib k* ∗ *fib n*
apply (*induct n rule*: *fib.induct*)
prefer *3*

simplify the LHS just enough to apply the induction hypotheses

apply (*simp add*: *fib.Suc-Suc* [*of Suc* (*m* + *n*), *standard*])
apply (*simp-all* (*no-asm-simp*) *add*: *fib.Suc-Suc add-mult-distrib add-mult-distrib2*)
done

lemma *fib-Suc-neq-0* [*simp*]: *fib* (*Suc n*) ≠ *0*
apply (*induct n rule*: *fib.induct*)
apply (*simp-all add*: *fib.Suc-Suc*)
done

lemma [*simp*]: *0* < *fib* (*Suc n*)
apply (*simp add*: *neq0-conv* [*symmetric*])
done

lemma *fib-gr-0*: *0* < *n* ==> *0* < *fib n*
apply (*rule not0-implies-Suc* [*THEN exE*])
apply *auto*
done

Concrete Mathematics, page 278: Cassini's identity. It is much easier to prove using integers!

lemma *fib-Cassini*: *int* (*fib* (*Suc* (*Suc n*)) ∗ *fib n*) =
(*if n mod 2* = *0 then int* (*fib* (*Suc n*) ∗ *fib* (*Suc n*)) − *1*
else int (*fib* (*Suc n*) ∗ *fib* (*Suc n*)) + *1*)
apply (*induct n rule*: *fib.induct*)
apply (*simp add*: *fib.Suc-Suc*)

apply (*simp add*: *fib.Suc-Suc mod-Suc*)
apply (*simp add*: *fib.Suc-Suc*
add-mult-distrib add-mult-distrib2 mod-Suc zmult-int [*symmetric*] *zmult-ac*)
apply (*subgoal-tac x mod 2 < 2*, *arith*)
apply *simp*
done

Towards Law 6.111 of Concrete Mathematics

lemma *gcd-fib-Suc-eq-1*: *gcd* (*fib n*, *fib* (*Suc n*)) = *Suc 0*
apply (*induct n rule*: *fib.induct*)
prefer *3*
apply (*simp add*: *gcd-commute fib-Suc3*)
apply (*simp-all add*: *fib.Suc-Suc*)
done

lemma *gcd-fib-add*: *gcd* (*fib m*, *fib* (*n* + *m*)) = *gcd* (*fib m*, *fib n*)
apply (*simp* (*no-asm*) *add*: *gcd-commute* [*of fib m*])
apply (*case-tac m = 0*)
apply *simp*
apply (*clarify dest*!: *not0-implies-Suc*)
apply (*simp add*: *fib-add*)
apply (*simp add*: *add-commute gcd-non-0*)
apply (*simp add*: *gcd-non-0* [*symmetric*])
apply (*simp add*: *gcd-fib-Suc-eq-1 gcd-mult-cancel*)
done

lemma *gcd-fib-diff*: *m* ≤ *n* ==> *gcd* (*fib m*, *fib* (*n* − *m*)) = *gcd* (*fib m*, *fib n*)
apply (*rule gcd-fib-add* [*symmetric*, *THEN trans*])
apply *simp*
done

lemma *gcd-fib-mod*: *0 < m* ==> *gcd* (*fib m*, *fib* (*n mod m*)) = *gcd* (*fib m*, *fib n*)
apply (*induct n rule*: *nat-less-induct*)
apply (*subst mod-if*)
apply (*simp add*: *gcd-fib-diff mod-geq not-less-iff-le diff-less*)
done

lemma *fib-gcd*: *fib* (*gcd* (*m*, *n*)) = *gcd* (*fib m*, *fib n*) — Law 6.111
apply (*induct m n rule*: *gcd-induct*)
apply *simp*
apply (*simp add*: *gcd-non-0*)
apply (*simp add*: *gcd-commute gcd-fib-mod*)
done

lemma *fib-mult-eq-setsum*:
  fib (*Suc n*) ∗ *fib n* = (∑ *k* ∈ {..*n*}. *fib k* ∗ *fib k*)
apply (*induct n rule*: *fib.induct*)
  apply (*auto simp add*: *atMost-Suc fib.Suc-Suc*)
apply (*simp add*: *add-mult-distrib add-mult-distrib2*)
done

end

4 Fundamental Theorem of Arithmetic (unique factorization into primes)

theory *Factorization* = *Primes* + *Permutation*:

4.1 Definitions

consts
  primel :: *nat list* => *bool*
  nondec :: *nat list* => *bool*
  prod :: *nat list* => *nat*
  oinsert :: *nat* => *nat list* => *nat list*
  sort :: *nat list* => *nat list*

defs
  primel-def: *primel xs* == *set xs* ⊆ *prime*

primrec
  nondec [] = *True*
  nondec (*x* # *xs*) = (*case xs of* [] => *True* | *y* # *ys* => *x* ≤ *y* ∧ *nondec xs*)

primrec
  prod [] = *Suc 0*
  prod (*x* # *xs*) = *x* ∗ *prod xs*

primrec
  oinsert x [] = [*x*]
  oinsert x (*y* # *ys*) = (*if x* ≤ *y then x* # *y* # *ys else y* # *oinsert x ys*)

primrec
  sort [] = []
  sort (*x* # *xs*) = *oinsert x* (*sort xs*)

4.2 Arithmetic

lemma *one-less-m*: $(m::nat) \neq m * k \Longrightarrow m \neq Suc\ 0 \Longrightarrow Suc\ 0 < m$
apply (*case-tac m*)
apply *auto*
done

lemma *one-less-k*: $(m::nat) \neq m * k \Longrightarrow Suc\ 0 < m * k \Longrightarrow Suc\ 0 < k$
apply (*case-tac k*)
apply *auto*
done

lemma *mult-left-cancel*: $(0::nat) < k \Longrightarrow k * n = k * m \Longrightarrow n = m$
apply *auto*
done

lemma *mn-eq-m-one*: $(0::nat) < m \Longrightarrow m * n = m \Longrightarrow n = Suc\ 0$
apply (*case-tac n*)
apply *auto*
done

lemma *prod-mn-less-k*:
$(0::nat) < n \Longrightarrow 0 < k \Longrightarrow Suc\ 0 < m \Longrightarrow m * n = k \Longrightarrow n < k$
apply (*induct m*)
apply *auto*
done

4.3 Prime list and product

lemma *prod-append*: $prod\ (xs\ @\ ys) = prod\ xs * prod\ ys$
apply (*induct xs*)
apply (*simp-all add*: *mult-assoc*)
done

lemma *prod-xy-prod*:
$prod\ (x\ \#\ xs) = prod\ (y\ \#\ ys) \Longrightarrow x * prod\ xs = y * prod\ ys$
apply *auto*
done

lemma *primel-append*: $primel\ (xs\ @\ ys) = (primel\ xs \wedge primel\ ys)$
apply (*unfold primel-def*)
apply *auto*
done

lemma *prime-primel*: $n \in prime \Longrightarrow primel\ [n] \wedge prod\ [n] = n$

apply (*unfold primel-def*)
apply *auto*
done

lemma *prime-nd-one*: $p \in prime \Longrightarrow \neg\ p\ dvd\ Suc\ 0$
apply (*unfold prime-def dvd-def*)
apply *auto*
done

lemma *hd-dvd-prod*: $prod\ (x\ \#\ xs) = prod\ ys \Longrightarrow x\ dvd\ (prod\ ys)$
apply (*unfold dvd-def*)
apply (*rule exI*)
apply (*rule sym*)
apply *simp*
done

lemma *primel-tl*: $primel\ (x\ \#\ xs) \Longrightarrow primel\ xs$
apply (*unfold primel-def*)
apply *auto*
done

lemma *primel-hd-tl*: $(primel\ (x\ \#\ xs)) = (x \in prime \wedge primel\ xs)$
apply (*unfold primel-def*)
apply *auto*
done

lemma *primes-eq*: $p \in prime \Longrightarrow q \in prime \Longrightarrow p\ dvd\ q \Longrightarrow p = q$
apply (*unfold prime-def*)
apply *auto*
done

lemma *primel-one-empty*: $primel\ xs \Longrightarrow prod\ xs = Suc\ 0 \Longrightarrow xs = []$
apply (*unfold primel-def prime-def*)
apply (*case-tac xs*)
apply *simp-all*
done

lemma *prime-g-one*: $p \in prime \Longrightarrow Suc\ 0 < p$
apply (*unfold prime-def*)
apply *auto*
done

lemma *prime-g-zero*: $p \in prime \Longrightarrow 0 < p$
apply (*unfold prime-def*)
apply *auto*

done

lemma *primel-nempty-g-one* [*rule-format*]:
primel xs −−> *xs* ≠ [] −−> *Suc 0* < *prod xs*
apply (*unfold primel-def prime-def*)
apply (*induct xs*)
apply (*auto elim*: *one-less-mult*)
done

lemma *primel-prod-gz*: *primel xs* ==> *0* < *prod xs*
apply (*unfold primel-def prime-def*)
apply (*induct xs*)
apply *auto*
done

4.4 Sorting

lemma *nondec-oinsert* [*rule-format*]: *nondec xs* −−> *nondec* (*oinsert x xs*)
apply (*induct xs*)
apply (*case-tac* [*2*] *list*)
apply (*simp-all cong del*: *list.weak-case-cong*)
done

lemma *nondec-sort*: *nondec* (*sort xs*)
apply (*induct xs*)
apply *simp-all*
apply (*erule nondec-oinsert*)
done

lemma *x-less-y-oinsert*: *x* ≤ *y* ==> *l* = *y* # *ys* ==> *x* # *l* = *oinsert x l*
apply *simp-all*
done

lemma *nondec-sort-eq* [*rule-format*]: *nondec xs* −−> *xs* = *sort xs*
apply (*induct xs*)
apply *safe*
apply *simp-all*
apply (*case-tac list*)
apply *simp-all*
apply (*case-tac list*)
apply *simp*
apply (*rule-tac y* = *aa* **and** *ys* = *lista* **in** *x-less-y-oinsert*)
apply *simp-all*
done

lemma *oinsert-x-y*: *oinsert x* (*oinsert y l*) = *oinsert y* (*oinsert x l*)
apply (*induct l*)
apply *auto*
done

4.5 Permutation

lemma *perm-primel* [*rule-format*]: *xs* <~~> *ys* ==> *primel xs* −−> *primel ys*
apply (*unfold primel-def*)
apply (*erule perm.induct*)
apply *simp-all*
done

lemma *perm-prod* [*rule-format*]: *xs* <~~> *ys* ==> *prod xs* = *prod ys*
apply (*erule perm.induct*)
apply (*simp-all add*: *mult-ac*)
done

lemma *perm-subst-oinsert*: *xs* <~~> *ys* ==> *oinsert a xs* <~~> *oinsert a ys*
apply (*erule perm.induct*)
apply *auto*
done

lemma *perm-oinsert*: *x* # *xs* <~~> *oinsert x xs*
apply (*induct xs*)
apply *auto*
done

lemma *perm-sort*: *xs* <~~> *sort xs*
apply (*induct xs*)
apply (*auto intro*: *perm-oinsert elim*: *perm-subst-oinsert*)
done

lemma *perm-sort-eq*: *xs* <~~> *ys* ==> *sort xs* = *sort ys*
apply (*erule perm.induct*)
apply (*simp-all add*: *oinsert-x-y*)
done

4.6 Existence

lemma *ex-nondec-lemma*:
primel xs ==> ∃ *ys*. *primel ys* ∧ *nondec ys* ∧ *prod ys* = *prod xs*
apply (*blast intro*: *nondec-sort perm-prod perm-primel perm-sort perm-sym*)
done

lemma *not-prime-ex-mk*:
Suc 0 < n ∧ n ∉ prime ==>
*∃ m k. Suc 0 < m ∧ Suc 0 < k ∧ m < n ∧ k < n ∧ n = m * k*
apply (*unfold prime-def dvd-def*)
apply (*auto intro*: *n-less-m-mult-n n-less-n-mult-m one-less-m one-less-k*)
done

lemma *split-primel*:
*primel xs ==> primel ys ==> ∃ l. primel l ∧ prod l = prod xs * prod ys*
apply (*rule exI*)
apply *safe*
apply (*rule-tac* [*2*] *prod-append*)
apply (*simp add*: *primel-append*)
done

lemma *factor-exists* [*rule-format*]: *Suc 0 < n --> (∃ l. primel l ∧ prod l = n)*
apply (*induct n rule*: *nat-less-induct*)
apply (*rule impI*)
apply (*case-tac n ∈ prime*)
apply (*rule exI*)
apply (*erule prime-primel*)
apply (*cut-tac n = n* **in** *not-prime-ex-mk*)
apply (*auto intro*!: *split-primel*)
done

lemma *nondec-factor-exists*: *Suc 0 < n ==> ∃ l. primel l ∧ nondec l ∧ prod l = n*
apply (*erule factor-exists* [*THEN exE*])
apply (*blast intro*!: *ex-nondec-lemma*)
done

4.7 Uniqueness

lemma *prime-dvd-mult-list* [*rule-format*]:
p ∈ prime ==> p dvd (prod xs) --> (∃ m. m:set xs ∧ p dvd m)
apply (*induct xs*)
apply (*force simp add*: *prime-def*)
apply (*force dest*: *prime-dvd-mult*)
done

lemma *hd-xs-dvd-prod*:
primel (x # xs) ==> primel ys ==> prod (x # xs) = prod ys
==> ∃ m. m ∈ set ys ∧ x dvd m
apply (*rule prime-dvd-mult-list*)
apply (*simp add*: *primel-hd-tl*)

apply (*erule hd-dvd-prod*)
done

lemma *prime-dvd-eq*: *primel* (*x* # *xs*) ==> *primel ys* ==> *m* ∈ *set ys* ==> *x dvd m* ==> *x* = *m*
apply (*rule primes-eq*)
apply (*auto simp add*: *primel-def primel-hd-tl*)
done

lemma *hd-xs-eq-prod*:
primel (*x* # *xs*) ==>
primel ys ==> *prod* (*x* # *xs*) = *prod ys* ==> *x* ∈ *set ys*
apply (*frule hd-xs-dvd-prod*)
apply *auto*
apply (*drule prime-dvd-eq*)
apply *auto*
done

lemma *perm-primel-ex*:
primel (*x* # *xs*) ==>
primel ys ==> *prod* (*x* # *xs*) = *prod ys* ==> ∃ *l*. *ys* <~~> (*x* # *l*)
apply (*rule exI*)
apply (*rule perm-remove*)
apply (*erule hd-xs-eq-prod*)
apply *simp-all*
done

lemma *primel-prod-less*:
primel (*x* # *xs*) ==>
primel ys ==> *prod* (*x* # *xs*) = *prod ys* ==> *prod xs* < *prod ys*
apply (*auto intro*: *prod-mn-less-k prime-g-one primel-prod-gz simp add*: *primel-hd-tl*)
done

lemma *prod-one-empty*:
primel xs ==> *p* ∗ *prod xs* = *p* ==> *p* ∈ *prime* ==> *xs* = []
apply (*auto intro*: *primel-one-empty simp add*: *prime-def*)
done

lemma *uniq-ex-aux*:
∀ *m*. *m* < *prod ys* −−> (∀ *xs ys*. *primel xs* ∧ *primel ys* ∧
prod xs = *prod ys* ∧ *prod xs* = *m* −−> *xs* <~~> *ys*) ==>
primel list ==> *primel x* ==> *prod list* = *prod x* ==> *prod x* < *prod ys*
==> *x* <~~> *list*
apply *simp*
done

lemma *factor-unique* [*rule-format*]:
 ∀ *xs ys*. *primel xs* ∧ *primel ys* ∧ *prod xs* = *prod ys* ∧ *prod xs* = *n*
  −−> *xs* <~~> *ys*
 apply (*induct n rule*: *nat-less-induct*)
 apply *safe*
 apply (*case-tac xs*)
  apply (*force intro*: *primel-one-empty*)
 apply (*rule perm-primel-ex* [*THEN exE*])
   apply *simp-all*
 apply (*rule perm.trans* [*THEN perm-sym*])
 apply *assumption*
 apply (*rule perm.Cons*)
 apply (*case-tac x* = [])
  apply (*simp add*: *perm-sing-eq primel-hd-tl*)
  apply (*rule-tac p* = *a* **in** *prod-one-empty*)
   apply *simp-all*
 apply (*erule uniq-ex-aux*)
   apply (*auto intro*: *primel-tl perm-primel simp add*: *primel-hd-tl*)
  apply (*rule-tac k* = *a* **and** *n* = *prod list* **and** *m* = *prod x* **in** *mult-left-cancel*)
   apply (*rule-tac* [*3*] *x* = *a* **in** *primel-prod-less*)
    apply (*rule-tac* [*2*] *prod-xy-prod*)
    apply (*rule-tac* [*2*] *s* = *prod ys* **in** *HOL.trans*)
     apply (*erule-tac* [*3*] *perm-prod*)
    apply (*erule-tac* [*5*] *perm-prod* [*symmetric*])
   apply (*auto intro*: *perm-primel prime-g-zero*)
 done

lemma *perm-nondec-unique*:
  xs <~~> *ys* ==> *nondec xs* ==> *nondec ys* ==> *xs* = *ys*
 apply (*rule HOL.trans*)
  apply (*rule HOL.trans*)
   apply (*erule nondec-sort-eq*)
  apply (*erule perm-sort-eq*)
 apply (*erule nondec-sort-eq* [*symmetric*])
 done

lemma *unique-prime-factorization* [*rule-format*]:
  ∀ *n*. *Suc 0* < *n* −−> (∃!*l*. *primel l* ∧ *nondec l* ∧ *prod l* = *n*)
 apply *safe*
  apply (*erule nondec-factor-exists*)
 apply (*rule perm-nondec-unique*)
   apply (*rule factor-unique*)
   apply *simp-all*
 done

end

5 Divisibility and prime numbers (on integers)

theory *IntPrimes* = *Primes*:

The *dvd* relation, GCD, Euclid's extended algorithm, primes, congruences (all on the Integers). Comparable to theory *Primes*, but *dvd* is included here as it is not present in main HOL. Also includes extended GCD and congruences not present in *Primes*.

5.1 Definitions

consts
xzgcda :: *int* ∗ *int* ∗ *int* ∗ *int* ∗ *int* ∗ *int* ∗ *int* ∗ *int* => *int* ∗ *int* ∗ *int*

recdef *xzgcda*
measure ((λ(*m*, *n*, *r*′, *r*, *s*′, *s*, *t*′, *t*). *nat r*)
:: *int* ∗ *int* ∗ *int* ∗ *int* ∗*int* ∗ *int* ∗ *int* ∗ *int* => *nat*)
xzgcda (*m*, *n*, *r*′, *r*, *s*′, *s*, *t*′, *t*) =
(*if r* ≤ *0 then* (*r*′, *s*′, *t*′)
else xzgcda (*m*, *n*, *r*, *r*′ *mod r*,
s, *s*′ − (*r*′ *div r*) ∗ *s*,
t, *t*′ − (*r*′ *div r*) ∗ *t*))

constdefs
zgcd :: *int* ∗ *int* => *int*
zgcd == λ(*x*,*y*). *int* (*gcd* (*nat* (*abs x*), *nat* (*abs y*)))

zprime :: *int set*
zprime == {*p*. *1* < *p* ∧ (∀ *m*. *0* <= *m* & *m dvd p* −−> *m* = *1* ∨ *m* = *p*)}

xzgcd :: *int* => *int* => *int* ∗ *int* ∗ *int*
xzgcd m n == *xzgcda* (*m*, *n*, *m*, *n*, *1*, *0*, *0*, *1*)

zcong :: *int* => *int* => *int* => *bool* ((*1*[- = -] ′(*mod* -′)))
[*a* = *b*] (*mod m*) == *m dvd* (*a* − *b*)

gcd lemmas

lemma *gcd-add1-eq*: *gcd* (*m* + *k*, *k*) = *gcd* (*m* + *k*, *m*)
by (*simp add*: *gcd-commute*)

lemma *gcd-diff2*: $m \leq n \Longrightarrow gcd\ (n,\ n - m) = gcd\ (n,\ m)$
apply (*subgoal-tac* $n = m + (n - m)$)
apply (*erule ssubst*, *rule gcd-add1-eq*, *simp*)
done

5.2 Euclid's Algorithm and GCD

lemma *zgcd-0* [*simp*]: $zgcd\ (m,\ 0) = abs\ m$
by (*simp add*: *zgcd-def zabs-def*)

lemma *zgcd-0-left* [*simp*]: $zgcd\ (0,\ m) = abs\ m$
by (*simp add*: *zgcd-def zabs-def*)

lemma *zgcd-zminus* [*simp*]: $zgcd\ (-m,\ n) = zgcd\ (m,\ n)$
by (*simp add*: *zgcd-def*)

lemma *zgcd-zminus2* [*simp*]: $zgcd\ (m,\ -n) = zgcd\ (m,\ n)$
by (*simp add*: *zgcd-def*)

lemma *zgcd-non-0*: $0 < n \Longrightarrow zgcd\ (m,\ n) = zgcd\ (n,\ m\ mod\ n)$
apply (*frule-tac* $b = n$ **and** $a = m$ **in** *pos-mod-sign*)
apply (*simp del*: *pos-mod-sign add*: *zgcd-def zabs-def nat-mod-distrib*)
apply (*auto simp add*: *gcd-non-0 nat-mod-distrib* [*symmetric*] *zmod-zminus1-eq-if*)
apply (*frule-tac* $a = m$ **in** *pos-mod-bound*)
apply (*simp del*: *pos-mod-bound add*: *nat-diff-distrib gcd-diff2 nat-le-eq-zle*)
done

lemma *zgcd-eq*: $zgcd\ (m,\ n) = zgcd\ (n,\ m\ mod\ n)$
apply (*case-tac* $n = 0$, *simp add*: *DIVISION-BY-ZERO*)
apply (*auto simp add*: *linorder-neq-iff zgcd-non-0*)
apply (*cut-tac* $m = -m$ **and** $n = -n$ **in** *zgcd-non-0*, *auto*)
done

lemma *zgcd-1* [*simp*]: $zgcd\ (m,\ 1) = 1$
by (*simp add*: *zgcd-def zabs-def*)

lemma *zgcd-0-1-iff* [*simp*]: $(zgcd\ (0,\ m) = 1) = (abs\ m = 1)$
by (*simp add*: *zgcd-def zabs-def*)

lemma *zgcd-zdvd1* [*iff*]: $zgcd\ (m,\ n)\ dvd\ m$
by (*simp add*: *zgcd-def zabs-def int-dvd-iff*)

lemma *zgcd-zdvd2* [*iff*]: $zgcd\ (m,\ n)\ dvd\ n$
by (*simp add*: *zgcd-def zabs-def int-dvd-iff*)

lemma *zgcd-greatest-iff*: *k dvd zgcd* (*m*, *n*) = (*k dvd m* ∧ *k dvd n*)
by (*simp add*: *zgcd-def zabs-def int-dvd-iff dvd-int-iff nat-dvd-iff*)

lemma *zgcd-commute*: *zgcd* (*m*, *n*) = *zgcd* (*n*, *m*)
by (*simp add*: *zgcd-def gcd-commute*)

lemma *zgcd-1-left* [*simp*]: *zgcd* (*1*, *m*) = *1*
by (*simp add*: *zgcd-def gcd-1-left*)

lemma *zgcd-assoc*: *zgcd* (*zgcd* (*k*, *m*), *n*) = *zgcd* (*k*, *zgcd* (*m*, *n*))
by (*simp add*: *zgcd-def gcd-assoc*)

lemma *zgcd-left-commute*: *zgcd* (*k*, *zgcd* (*m*, *n*)) = *zgcd* (*m*, *zgcd* (*k*, *n*))
apply (*rule zgcd-commute* [*THEN trans*])
apply (*rule zgcd-assoc* [*THEN trans*])
apply (*rule zgcd-commute* [*THEN arg-cong*])
done

lemmas *zgcd-ac* = *zgcd-assoc zgcd-commute zgcd-left-commute*
— addition is an AC-operator

lemma *zgcd-zmult-distrib2*: *0* ≤ *k* ==> *k* ∗ *zgcd* (*m*, *n*) = *zgcd* (*k* ∗ *m*, *k* ∗ *n*)
by (*simp del*: *minus-mult-right* [*symmetric*]
add: *minus-mult-right nat-mult-distrib zgcd-def zabs-def*
mult-less-0-iff gcd-mult-distrib2 [*symmetric*] *zmult-int* [*symmetric*])

lemma *zgcd-zmult-distrib2-abs*: *zgcd* (*k* ∗ *m*, *k* ∗ *n*) = *abs k* ∗ *zgcd* (*m*, *n*)
by (*simp add*: *zabs-def zgcd-zmult-distrib2*)

lemma *zgcd-self* [*simp*]: *0* ≤ *m* ==> *zgcd* (*m*, *m*) = *m*
by (*cut-tac k* = *m* **and** *m* = *1* **and** *n* = *1* **in** *zgcd-zmult-distrib2*, *simp-all*)

lemma *zgcd-zmult-eq-self* [*simp*]: *0* ≤ *k* ==> *zgcd* (*k*, *k* ∗ *n*) = *k*
by (*cut-tac k* = *k* **and** *m* = *1* **and** *n* = *n* **in** *zgcd-zmult-distrib2*, *simp-all*)

lemma *zgcd-zmult-eq-self2* [*simp*]: *0* ≤ *k* ==> *zgcd* (*k* ∗ *n*, *k*) = *k*
by (*cut-tac k* = *k* **and** *m* = *n* **and** *n* = *1* **in** *zgcd-zmult-distrib2*, *simp-all*)

lemma *zrelprime-zdvd-zmult-aux*:
zgcd (*n*, *k*) = *1* ==> *k dvd m* ∗ *n* ==> *0* ≤ *m* ==> *k dvd m*
apply (*subgoal-tac m* = *zgcd* (*m* ∗ *n*, *m* ∗ *k*))
apply (*erule ssubst*, *rule zgcd-greatest-iff* [*THEN iffD2*])
apply (*simp-all add*: *zgcd-zmult-distrib2* [*symmetric*] *zero-le-mult-iff*)
done

lemma *zrelprime-zdvd-zmult*: *zgcd* (*n*, *k*) = *1* ==> *k dvd m* * *n* ==> *k dvd m*
apply (*case-tac* *0* ≤ *m*)
apply (*blast intro*: *zrelprime-zdvd-zmult-aux*)
apply (*subgoal-tac k dvd* −*m*)
apply (*rule-tac* [*2*] *zrelprime-zdvd-zmult-aux*, *auto*)
done

lemma *zgcd-geq-zero*: *0* <= *zgcd*(*x*,*y*)
by (*auto simp add*: *zgcd-def*)

This is merely a sanity check on zprime, since the previous version denoted the empty set.

lemma *2* ∈ *zprime*
apply (*auto simp add*: *zprime-def*)
apply (*frule zdvd-imp-le*, *simp*)
apply (*auto simp add*: *order-le-less dvd-def*)
done

lemma *zprime-imp-zrelprime*:
p ∈ *zprime* ==> ¬ *p dvd n* ==> *zgcd* (*n*, *p*) = *1*
apply (*auto simp add*: *zprime-def*)
apply (*drule-tac x* = *zgcd*(*n*, *p*) **in** *allE*)
apply (*auto simp add*: *zgcd-zdvd2* [*of n p*] *zgcd-geq-zero*)
apply (*insert zgcd-zdvd1* [*of n p*], *auto*)
done

lemma *zless-zprime-imp-zrelprime*:
p ∈ *zprime* ==> *0* < *n* ==> *n* < *p* ==> *zgcd* (*n*, *p*) = *1*
apply (*erule zprime-imp-zrelprime*)
apply (*erule zdvd-not-zless*, *assumption*)
done

lemma *zprime-zdvd-zmult*:
0 ≤ (*m*::*int*) ==> *p* ∈ *zprime* ==> *p dvd m* * *n* ==> *p dvd m* ∨ *p dvd n*
apply *safe*
apply (*rule zrelprime-zdvd-zmult*)
apply (*rule zprime-imp-zrelprime*, *auto*)
done

lemma *zgcd-zadd-zmult* [*simp*]: *zgcd* (*m* + *n* * *k*, *n*) = *zgcd* (*m*, *n*)
apply (*rule zgcd-eq* [*THEN trans*])
apply (*simp add*: *zmod-zadd1-eq*)
apply (*rule zgcd-eq* [*symmetric*])

done

lemma *zgcd-zdvd-zgcd-zmult*: *zgcd* (*m*, *n*) *dvd zgcd* (*k* * *m*, *n*)
apply (*simp add*: *zgcd-greatest-iff*)
apply (*blast intro*: *zdvd-trans*)
done

lemma *zgcd-zmult-zdvd-zgcd*:
zgcd (*k*, *n*) = *1* ==> *zgcd* (*k* * *m*, *n*) *dvd zgcd* (*m*, *n*)
apply (*simp add*: *zgcd-greatest-iff*)
apply (*rule-tac n* = *k* **in** *zrelprime-zdvd-zmult*)
prefer *2*
apply (*simp add*: *zmult-commute*)
apply (*subgoal-tac zgcd* (*k*, *zgcd* (*k* * *m*, *n*)) = *zgcd* (*k* * *m*, *zgcd* (*k*, *n*)))
apply *simp*
apply (*simp* (*no-asm*) *add*: *zgcd-ac*)
done

lemma *zgcd-zmult-cancel*: *zgcd* (*k*, *n*) = *1* ==> *zgcd* (*k* * *m*, *n*) = *zgcd* (*m*, *n*)
by (*simp add*: *zgcd-def nat-abs-mult-distrib gcd-mult-cancel*)

lemma *zgcd-zgcd-zmult*:
zgcd (*k*, *m*) = *1* ==> *zgcd* (*n*, *m*) = *1* ==> *zgcd* (*k* * *n*, *m*) = *1*
by (*simp add*: *zgcd-zmult-cancel*)

lemma *zdvd-iff-zgcd*: *0* < *m* ==> (*m dvd n*) = (*zgcd* (*n*, *m*) = *m*)
apply *safe*
apply (*rule-tac* [*2*] *n* = *zgcd* (*n*, *m*) **in** *zdvd-trans*)
apply (*rule-tac* [*3*] *zgcd-zdvd1*, *simp-all*)
apply (*unfold dvd-def*, *auto*)
done

5.3 Congruences

lemma *zcong-1* [*simp*]: [*a* = *b*] (*mod 1*)
by (*unfold zcong-def*, *auto*)

lemma *zcong-refl* [*simp*]: [*k* = *k*] (*mod m*)
by (*unfold zcong-def*, *auto*)

lemma *zcong-sym*: [*a* = *b*] (*mod m*) = [*b* = *a*] (*mod m*)
apply (*unfold zcong-def dvd-def*, *auto*)
apply (*rule-tac* [!] *x* = −*k* **in** *exI*, *auto*)
done

lemma *zcong-zadd*:
  $[a = b]\ (mod\ m) ==> [c = d]\ (mod\ m) ==> [a + c = b + d]\ (mod\ m)$
 apply (*unfold zcong-def*)
 apply (*rule-tac* $s = (a - b) + (c - d)$ **in** *subst*)
  apply (*rule-tac* [*2*] *zdvd-zadd*, *auto*)
 done

lemma *zcong-zdiff*:
  $[a = b]\ (mod\ m) ==> [c = d]\ (mod\ m) ==> [a - c = b - d]\ (mod\ m)$
 apply (*unfold zcong-def*)
 apply (*rule-tac* $s = (a - b) - (c - d)$ **in** *subst*)
  apply (*rule-tac* [*2*] *zdvd-zdiff*, *auto*)
 done

lemma *zcong-trans*:
  $[a = b]\ (mod\ m) ==> [b = c]\ (mod\ m) ==> [a = c]\ (mod\ m)$
 apply (*unfold zcong-def dvd-def*, *auto*)
 apply (*rule-tac* $x = k + ka$ **in** *exI*)
 apply (*simp add*: *zadd-ac zadd-zmult-distrib2*)
 done

lemma *zcong-zmult*:
  $[a = b]\ (mod\ m) ==> [c = d]\ (mod\ m) ==> [a * c = b * d]\ (mod\ m)$
 apply (*rule-tac* $b = b * c$ **in** *zcong-trans*)
  apply (*unfold zcong-def*)
  apply (*rule-tac* $s = c * (a - b)$ **in** *subst*)
   apply (*rule-tac* [*3*] $s = b * (c - d)$ **in** *subst*)
    prefer *4*
    apply (*blast intro*: *zdvd-zmult*)
   prefer *2*
   apply (*blast intro*: *zdvd-zmult*)
  apply (*simp-all add*: *zdiff-zmult-distrib2 zmult-commute*)
 done

lemma *zcong-scalar*: $[a = b]\ (mod\ m) ==> [a * k = b * k]\ (mod\ m)$
 by (*rule zcong-zmult*, *simp-all*)

lemma *zcong-scalar2*: $[a = b]\ (mod\ m) ==> [k * a = k * b]\ (mod\ m)$
 by (*rule zcong-zmult*, *simp-all*)

lemma *zcong-zmult-self*: $[a * m = b * m]\ (mod\ m)$
 apply (*unfold zcong-def*)
 apply (*rule zdvd-zdiff*, *simp-all*)
 done

lemma *zcong-square*:
[|$p \in$ *zprime*; $0 < a$; $[a * a = 1]$ (*mod* p)|]
==> $[a = 1]$ (*mod* p) $\vee$ $[a = p - 1]$ (*mod* p)
apply (*unfold zcong-def*)
apply (*rule zprime-zdvd-zmult*)
apply (*rule-tac* [3] $s = a * a - 1 + p * (1 - a)$ **in** *subst*)
prefer 4
apply (*simp add*: *zdvd-reduce*)
apply (*simp-all add*: *zdiff-zmult-distrib zmult-commute zdiff-zmult-distrib2*)
done

lemma *zcong-cancel*:
$0 \leq m$ ==>
zgcd (k, m) = 1 ==> $[a * k = b * k]$ (*mod* m) = $[a = b]$ (*mod* m)
apply *safe*
prefer 2
apply (*blast intro*: *zcong-scalar*)
apply (*case-tac* $b < a$)
prefer 2
apply (*subst zcong-sym*)
apply (*unfold zcong-def*)
apply (*rule-tac* [!] *zrelprime-zdvd-zmult*)
apply (*simp-all add*: *zdiff-zmult-distrib*)
apply (*subgoal-tac m dvd* $(-(a * k - b * k))$)
apply *simp*
apply (*subst zdvd-zminus-iff*, *assumption*)
done

lemma *zcong-cancel2*:
$0 \leq m$ ==>
zgcd (k, m) = 1 ==> $[k * a = k * b]$ (*mod* m) = $[a = b]$ (*mod* m)
by (*simp add*: *zmult-commute zcong-cancel*)

lemma *zcong-zgcd-zmult-zmod*:
$[a = b]$ (*mod* m) ==> $[a = b]$ (*mod* n) ==> *zgcd* (m, n) = 1
==> $[a = b]$ (*mod* $m * n$)
apply (*unfold zcong-def dvd-def*, *auto*)
apply (*subgoal-tac m dvd n* $*$ *ka*)
apply (*subgoal-tac m dvd ka*)
apply (*case-tac* [2] $0 \leq ka$)
prefer 3
apply (*subst zdvd-zminus-iff* [*symmetric*])
apply (*rule-tac* $n = n$ **in** *zrelprime-zdvd-zmult*)
apply (*simp add*: *zgcd-commute*)
apply (*simp add*: *zmult-commute zdvd-zminus-iff*)

```
  prefer 2
  apply (rule-tac n = n in zrelprime-zdvd-zmult)
   apply (simp add: zgcd-commute)
  apply (simp add: zmult-commute)
 apply (auto simp add: dvd-def)
done

lemma zcong-zless-imp-eq:
 0 ≤ a ==>
  a < m ==> 0 ≤ b ==> b < m ==> [a = b] (mod m) ==> a = b
 apply (unfold zcong-def dvd-def, auto)
 apply (drule-tac f = λz. z mod m in arg-cong)
 apply (cut-tac x = a and y = b in linorder-less-linear, auto)
  apply (subgoal-tac [2] (a - b) mod m = a - b)
   apply (rule-tac [3] mod-pos-pos-trivial, auto)
 apply (subgoal-tac (m + (a - b)) mod m = m + (a - b))
  apply (rule-tac [2] mod-pos-pos-trivial, auto)
 done

lemma zcong-square-zless:
 p ∈ zprime ==> 0 < a ==> a < p ==>
  [a * a = 1] (mod p) ==> a = 1 ∨ a = p - 1
 apply (cut-tac p = p and a = a in zcong-square)
    apply (simp add: zprime-def)
   apply (auto intro: zcong-zless-imp-eq)
 done

lemma zcong-not:
   0 < a ==> a < m ==> 0 < b ==> b < a ==> ¬ [a = b] (mod m)
 apply (unfold zcong-def)
 apply (rule zdvd-not-zless, auto)
 done

lemma zcong-zless-0:
   0 ≤ a ==> a < m ==> [a = 0] (mod m) ==> a = 0
 apply (unfold zcong-def dvd-def, auto)
 apply (subgoal-tac 0 < m)
  apply (simp add: zero-le-mult-iff)
  apply (subgoal-tac m * k < m * 1)
   apply (drule mult-less-cancel-left [THEN iffD1])
   apply (auto simp add: linorder-neq-iff)
 done

lemma zcong-zless-unique:
   0 < m ==> (∃!b. 0 ≤ b ∧ b < m ∧ [a = b] (mod m))
```

```
 apply auto
  apply (subgoal-tac [2] [b = y] (mod m))
   apply (case-tac [2] b = 0)
    apply (case-tac [3] y = 0)
     apply (auto intro: zcong-trans zcong-zless-0 zcong-zless-imp-eq order-less-le
       simp add: zcong-sym)
 apply (unfold zcong-def dvd-def)
 apply (rule-tac x = a mod m in exI, auto)
 apply (rule-tac x = −(a div m) in exI)
 apply (simp add: diff-eq-eq eq-diff-eq add-commute)
 done

lemma zcong-iff-lin: ([a = b] (mod m)) = (∃ k. b = a + m ∗ k)
 apply (unfold zcong-def dvd-def, auto)
  apply (rule-tac [!] x = −k in exI, auto)
 done

lemma zgcd-zcong-zgcd:
 0 < m ==>
   zgcd (a, m) = 1 ==> [a = b] (mod m) ==> zgcd (b, m) = 1
 by (auto simp add: zcong-iff-lin)

lemma zcong-zmod-aux:
    a − b = (m::int) ∗ (a div m − b div m) + (a mod m − b mod m)
 by(simp add: zdiff-zmult-distrib2 add-diff-eq eq-diff-eq add-ac)

lemma zcong-zmod: [a = b] (mod m) = [a mod m = b mod m] (mod m)
 apply (unfold zcong-def)
 apply (rule-tac t = a − b in ssubst)
 apply (rule-tac m = m in zcong-zmod-aux)
 apply (rule trans)
  apply (rule-tac [2] k = m and m = a div m − b div m in zdvd-reduce)
 apply (simp add: zadd-commute)
 done

lemma zcong-zmod-eq: 0 < m ==> [a = b] (mod m) = (a mod m = b mod m)
 apply auto
  apply (rule-tac m = m in zcong-zless-imp-eq)
     prefer 5
     apply (subst zcong-zmod [symmetric], simp-all)
 apply (unfold zcong-def dvd-def)
 apply (rule-tac x = a div m − b div m in exI)
 apply (rule-tac m1 = m in zcong-zmod-aux [THEN trans], auto)
 done
```

lemma *zcong-zminus* [*iff*]: $[a = b]\ (mod\ -m) = [a = b]\ (mod\ m)$
by (*auto simp add*: *zcong-def*)

lemma *zcong-zero* [*iff*]: $[a = b]\ (mod\ 0) = (a = b)$
by (*auto simp add*: *zcong-def*)

lemma $[a = b]\ (mod\ m) = (a\ mod\ m = b\ mod\ m)$
apply (*case-tac* $m = 0$, *simp add*: *DIVISION-BY-ZERO*)
apply (*simp add*: *linorder-neq-iff*)
apply (*erule disjE*)
prefer *2* **apply** (*simp add*: *zcong-zmod-eq*)

Remainding case: $m < 0$

apply (*rule-tac* $t = m$ **in** *zminus-zminus* [*THEN subst*])
apply (*subst zcong-zminus*)
apply (*subst zcong-zmod-eq*, *arith*)
apply (*frule neg-mod-bound* [*of* - *a*], *frule neg-mod-bound* [*of* - *b*])
apply (*simp add*: *zmod-zminus2-eq-if del*: *neg-mod-bound*)
done

5.4 Modulo

lemma *zmod-zdvd-zmod*:
$0 < (m::int) \Longrightarrow m\ dvd\ b \Longrightarrow (a\ mod\ b\ mod\ m) = (a\ mod\ m)$
apply (*unfold dvd-def*, *auto*)
apply (*subst zcong-zmod-eq* [*symmetric*])
prefer *2*
apply (*subst zcong-iff-lin*)
apply (*rule-tac* $x = k * (a\ div\ (m * k))$ **in** *exI*)
apply (*simp add*:*zmult-assoc* [*symmetric*], *assumption*)
done

5.5 Extended GCD

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

lemma *xzgcd-correct-aux1*:
zgcd $(r', r) = k \longrightarrow 0 < r \longrightarrow$
$(\exists\, sn\ tn.\ xzgcda\ (m, n, r', r, s', s, t', t) = (k, sn, tn))$
apply (*rule-tac* $u = m$ **and** $v = n$ **and** $w = r'$ **and** $x = r$ **and** $y = s'$ **and**
$z = s$ **and** $aa = t'$ **and** $ab = t$ **in** *xzgcda.induct*)
apply (*subst zgcd-eq*)
apply (*subst xzgcda.simps*, *auto*)
apply (*case-tac* $r'\ mod\ r = 0$)
prefer *2*

apply (*frule-tac* $a = r'$ **in** *pos-mod-sign*, *auto*)
apply (*rule exI*)
apply (*rule exI*)
apply (*subst xzgcda.simps*, *auto*)
apply (*simp add*: *zabs-def*)
done

lemma *xzgcd-correct-aux2*:
$(\exists\, sn\ tn.\ xzgcda\ (m,\ n,\ r',\ r,\ s',\ s,\ t',\ t) = (k,\ sn,\ tn)) \longrightarrow 0 < r \longrightarrow$
zgcd $(r',\ r) = k$
apply (*rule-tac* $u = m$ **and** $v = n$ **and** $w = r'$ **and** $x = r$ **and** $y = s'$ **and**
$z = s$ **and** $aa = t'$ **and** $ab = t$ **in** *xzgcda.induct*)
apply (*subst zgcd-eq*)
apply (*subst xzgcda.simps*)
apply (*auto simp add*: *linorder-not-le*)
apply (*case-tac* r' *mod* $r = 0$)
prefer *2*
apply (*frule-tac* $a = r'$ **in** *pos-mod-sign*, *auto*)
apply (*erule-tac* $P = xzgcda\ ?u = ?v$ **in** *rev-mp*)
apply (*subst xzgcda.simps*, *auto*)
apply (*simp add*: *zabs-def*)
done

lemma *xzgcd-correct*:
$0 < n \Longrightarrow (zgcd\ (m,\ n) = k) = (\exists\, s\ t.\ xzgcd\ m\ n = (k,\ s,\ t))$
apply (*unfold xzgcd-def*)
apply (*rule iffI*)
apply (*rule-tac* [*2*] *xzgcd-correct-aux2* [*THEN mp*, *THEN mp*])
apply (*rule xzgcd-correct-aux1* [*THEN mp*, *THEN mp*], *auto*)
done

xzgcd linear

lemma *xzgcda-linear-aux1*:
$(a - r * b) * m + (c - r * d) * (n::int) =$
$(a * m + c * n) - r * (b * m + d * n)$
by (*simp add*: *zdiff-zmult-distrib zadd-zmult-distrib2 zmult-assoc*)

lemma *xzgcda-linear-aux2*:
$r' = s' * m + t' * n \Longrightarrow r = s * m + t * n$
$\Longrightarrow (r'\ mod\ r) = (s' - (r'\ div\ r) * s) * m + (t' - (r'\ div\ r) * t) * (n::int)$
apply (*rule trans*)
apply (*rule-tac* [*2*] *xzgcda-linear-aux1* [*symmetric*])
apply (*simp add*: *eq-diff-eq mult-commute*)
done

lemma *order-le-neq-implies-less*: $(x::'a::order) \leq y \Longrightarrow x \neq y \Longrightarrow x < y$
by (*rule iffD2* [*OF order-less-le conjI*])

lemma *xzgcda-linear* [*rule-format*]:
$0 < r \longrightarrow xzgcda\ (m,\ n,\ r',\ r,\ s',\ s,\ t',\ t) = (rn,\ sn,\ tn) \longrightarrow$
$r' = s' * m + t' * n \longrightarrow\ r = s * m + t * n \longrightarrow rn = sn * m + tn * n$
apply (*rule-tac* $u = m$ **and** $v = n$ **and** $w = r'$ **and** $x = r$ **and** $y = s'$ **and**
$z = s$ **and** $aa = t'$ **and** $ab = t$ **in** *xzgcda.induct*)
apply (*subst xzgcda.simps*)
apply (*simp* (*no-asm*))
apply (*rule impI*)+
apply (*case-tac* r' *mod* $r = 0$)
apply (*simp add*: *xzgcda.simps*, *clarify*)
apply (*subgoal-tac* $0 < r'$ *mod* r)
apply (*rule-tac* [*2*] *order-le-neq-implies-less*)
apply (*rule-tac* [*2*] *pos-mod-sign*)
apply (*cut-tac* $m = m$ **and** $n = n$ **and** $r' = r'$ **and** $r = r$ **and** $s' = s'$ **and**
$s = s$ **and** $t' = t'$ **and** $t = t$ **in** *xzgcda-linear-aux2*, *auto*)
done

lemma *xzgcd-linear*:
$0 < n \Longrightarrow xzgcd\ m\ n = (r,\ s,\ t) \Longrightarrow r = s * m + t * n$
apply (*unfold xzgcd-def*)
apply (*erule xzgcda-linear*, *assumption*, *auto*)
done

lemma *zgcd-ex-linear*:
$0 < n \Longrightarrow zgcd\ (m,\ n) = k \Longrightarrow (\exists\, s\ t.\ k = s * m + t * n)$
apply (*simp add*: *xzgcd-correct*, *safe*)
apply (*rule exI*)+
apply (*erule xzgcd-linear*, *auto*)
done

lemma *zcong-lineq-ex*:
$0 < n \Longrightarrow zgcd\ (a,\ n) = 1 \Longrightarrow \exists x.\ [a * x = 1]\ (mod\ n)$
apply (*cut-tac* $m = a$ **and** $n = n$ **and** $k = 1$ **in** *zgcd-ex-linear*, *safe*)
apply (*rule-tac* $x = s$ **in** *exI*)
apply (*rule-tac* $b = s * a + t * n$ **in** *zcong-trans*)
prefer *2*
apply *simp*
apply (*unfold zcong-def*)
apply (*simp* (*no-asm*) *add*: *zmult-commute zdvd-zminus-iff*)
done

lemma *zcong-lineq-unique*:
0 < n ==>
*zgcd (a, n) = 1 ==> ∃!x. 0 ≤ x ∧ x < n ∧ [a * x = b] (mod n)*
apply *auto*
apply (*rule-tac [2] zcong-zless-imp-eq*)
apply (*tactic {* stac (thm zcong-cancel2 RS sym) 6 *}*)
apply (*rule-tac [8] zcong-trans*)
apply (*simp-all (no-asm-simp)*)
prefer *2*
apply (*simp add*: *zcong-sym*)
apply (*cut-tac a = a* **and** *n = n* **in** *zcong-lineq-ex*, *auto*)
apply (*rule-tac x = x * b mod n* **in** *exI*, *safe*)
apply (*simp-all (no-asm-simp)*)
apply (*subst zcong-zmod*)
apply (*subst zmod-zmult1-eq* [*symmetric*])
apply (*subst zcong-zmod* [*symmetric*])
apply (*subgoal-tac* [*a * x * b = 1 * b*] (*mod n*))
apply (*rule-tac* [*2*] *zcong-zmult*)
apply (*simp-all add*: *zmult-assoc*)
done

end

6 The Chinese Remainder Theorem

theory *Chinese = IntPrimes*:

The Chinese Remainder Theorem for an arbitrary finite number of equations. (The one-equation case is included in theory *IntPrimes*. Uses functions for indexing.[1]

6.1 Definitions

consts
funprod :: (*nat => int*) => *nat => nat => int*
funsum :: (*nat => int*) => *nat => nat => int*

primrec
funprod f i 0 = f i
*funprod f i (Suc n) = f (Suc (i + n)) * funprod f i n*

[1] Maybe *funprod* and *funsum* should be based on general *fold* on indices?

primrec
funsum f i 0 = f i
funsum f i (Suc n) = f (Suc (i + n)) + funsum f i n

consts
m-cond :: *nat => (nat => int) => bool*
km-cond :: *nat => (nat => int) => (nat => int) => bool*
lincong-sol ::
nat => (nat => int) => (nat => int) => (nat => int) => int => bool

mhf :: *(nat => int) => nat => nat => int*
xilin-sol ::
nat => nat => (nat => int) => (nat => int) => (nat => int) => int
x-sol :: *nat => (nat => int) => (nat => int) => (nat => int) => int*

defs
m-cond-def:
m-cond n mf ==
(∀ i. i ≤ n −−> 0 < mf i) ∧
(∀ i j. i ≤ n ∧ j ≤ n ∧ i ≠ j −−> zgcd (mf i, mf j) = 1)

km-cond-def:
km-cond n kf mf == ∀ i. i ≤ n −−> zgcd (kf i, mf i) = 1

lincong-sol-def:
*lincong-sol n kf bf mf x == ∀ i. i ≤ n −−> zcong (kf i * x) (bf i) (mf i)*

mhf-def:
mhf mf n i ==
if i = 0 then funprod mf (Suc 0) (n − Suc 0)
else if i = n then funprod mf 0 (n − Suc 0)
*else funprod mf 0 (i − Suc 0) * funprod mf (Suc i) (n − Suc 0 − i)*

xilin-sol-def:
xilin-sol i n kf bf mf ==
if 0 < n ∧ i ≤ n ∧ m-cond n mf ∧ km-cond n kf mf then
*(SOME x. 0 ≤ x ∧ x < mf i ∧ zcong (kf i * mhf mf n i * x) (bf i) (mf i))*
else 0

x-sol-def:
*x-sol n kf bf mf == funsum (λi. xilin-sol i n kf bf mf * mhf mf n i) 0 n*

funprod and *funsum*

lemma *funprod-pos*: *(∀ i. i ≤ n −−> 0 < mf i) ==> 0 < funprod mf 0 n*

apply (*induct n*)
apply *auto*
apply (*simp add*: *zero-less-mult-iff*)
done

lemma *funprod-zgcd* [*rule-format* (*no-asm*)]:
(∀ *i*. *k* ≤ *i* ∧ *i* ≤ *k* + *l* −−> *zgcd* (*mf i*, *mf m*) = *1*) −−>
zgcd (*funprod mf k l*, *mf m*) = *1*
apply (*induct l*)
apply *simp-all*
apply (*rule impI*)+
apply (*subst zgcd-zmult-cancel*)
apply *auto*
done

lemma *funprod-zdvd* [*rule-format*]:
k ≤ *i* −−> *i* ≤ *k* + *l* −−> *mf i dvd funprod mf k l*
apply (*induct l*)
apply *auto*
apply (*rule-tac* [*2*] *zdvd-zmult2*)
apply (*rule-tac* [*3*] *zdvd-zmult*)
apply (*subgoal-tac i* = *k*)
apply (*subgoal-tac* [*3*] *i* = *Suc* (*k* + *n*))
apply (*simp-all* (*no-asm-simp*))
done

lemma *funsum-mod*:
funsum f k l mod m = *funsum* (λ*i*. (*f i*) *mod m*) *k l mod m*
apply (*induct l*)
apply *auto*
apply (*rule trans*)
apply (*rule zmod-zadd1-eq*)
apply *simp*
apply (*rule zmod-zadd-right-eq* [*symmetric*])
done

lemma *funsum-zero* [*rule-format* (*no-asm*)]:
(∀ *i*. *k* ≤ *i* ∧ *i* ≤ *k* + *l* −−> *f i* = *0*) −−> (*funsum f k l*) = *0*
apply (*induct l*)
apply *auto*
done

lemma *funsum-oneelem* [*rule-format* (*no-asm*)]:
k ≤ *j* −−> *j* ≤ *k* + *l* −−>
(∀ *i*. *k* ≤ *i* ∧ *i* ≤ *k* + *l* ∧ *i* ≠ *j* −−> *f i* = *0*) −−>

```
   funsum f k l = f j
  apply (induct l)
   prefer 2
   apply clarify
   defer
   apply clarify
   apply (subgoal-tac k = j)
    apply (simp-all (no-asm-simp))
  apply (case-tac Suc (k + n) = j)
   apply (subgoal-tac funsum f k n = 0)
    apply (rule-tac [2] funsum-zero)
    apply (subgoal-tac [3] f (Suc (k + n)) = 0)
     apply (subgoal-tac [3] j ≤ k + n)
      prefer 4
      apply arith
     apply auto
  done
```

6.2 Chinese: uniqueness

```
lemma zcong-funprod-aux:
  m-cond n mf ==> km-cond n kf mf
    ==> lincong-sol n kf bf mf x ==> lincong-sol n kf bf mf y
    ==> [x = y] (mod mf n)
  apply (unfold m-cond-def km-cond-def lincong-sol-def)
  apply (rule iffD1)
   apply (rule-tac k = kf n in zcong-cancel2)
    apply (rule-tac [3] b = bf n in zcong-trans)
     prefer 4
     apply (subst zcong-sym)
     defer
     apply (rule order-less-imp-le)
     apply simp-all
  done

lemma zcong-funprod [rule-format]:
  m-cond n mf --> km-cond n kf mf -->
    lincong-sol n kf bf mf x --> lincong-sol n kf bf mf y -->
    [x = y] (mod funprod mf 0 n)
  apply (induct n)
   apply (simp-all (no-asm))
   apply (blast intro: zcong-funprod-aux)
  apply (rule impI)+
  apply (rule zcong-zgcd-zmult-zmod)
    apply (blast intro: zcong-funprod-aux)
```

```
  prefer 2
  apply (subst zgcd-commute)
  apply (rule funprod-zgcd)
 apply (auto simp add: m-cond-def km-cond-def lincong-sol-def)
done
```

6.3 Chinese: existence

```
lemma unique-xi-sol:
 0 < n ==> i ≤ n ==> m-cond n mf ==> km-cond n kf mf
  ==> ∃!x. 0 ≤ x ∧ x < mf i ∧ [kf i * mhf mf n i * x = bf i] (mod mf i)
 apply (rule zcong-lineq-unique)
  apply (tactic {* stac (thm zgcd-zmult-cancel) 2 *})
   apply (unfold m-cond-def km-cond-def mhf-def)
   apply (simp-all (no-asm-simp))
 apply safe
  apply (tactic {* stac (thm zgcd-zmult-cancel) 3 *})
   apply (rule-tac [!] funprod-zgcd)
   apply safe
   apply simp-all
  apply (subgoal-tac i<n)
   prefer 2
   apply arith
  apply (case-tac [2] i)
   apply simp-all
 done

lemma x-sol-lin-aux:
   0 < n ==> i ≤ n ==> j ≤ n ==> j ≠ i ==> mf j dvd mhf mf n i
 apply (unfold mhf-def)
 apply (case-tac i = 0)
  apply (case-tac [2] i = n)
   apply (simp-all (no-asm-simp))
   apply (case-tac [3] j < i)
    apply (rule-tac [3] zdvd-zmult2)
    apply (rule-tac [4] zdvd-zmult)
    apply (rule-tac [!] funprod-zdvd)
       apply arith+
 done

lemma x-sol-lin:
 0 < n ==> i ≤ n
  ==> x-sol n kf bf mf mod mf i =
    xilin-sol i n kf bf mf * mhf mf n i mod mf i
 apply (unfold x-sol-def)
```

```
  apply (subst funsum-mod)
  apply (subst funsum-oneelem)
    apply auto
  apply (subst zdvd-iff-zmod-eq-0 [symmetric])
  apply (rule zdvd-zmult)
  apply (rule x-sol-lin-aux)
  apply auto
  done
```

6.4 Chinese

```
lemma chinese-remainder:
  0 < n ==> m-cond n mf ==> km-cond n kf mf
    ==> ∃!x. 0 ≤ x ∧ x < funprod mf 0 n ∧ lincong-sol n kf bf mf x
  apply safe
   apply (rule-tac [2] m = funprod mf 0 n in zcong-zless-imp-eq)
      apply (rule-tac [6] zcong-funprod)
        apply auto
  apply (rule-tac x = x-sol n kf bf mf mod funprod mf 0 n in exI)
  apply (unfold lincong-sol-def)
  apply safe
    apply (tactic {* stac (thm zcong-zmod) 3 *})
    apply (tactic {* stac (thm zmod-zmult-distrib) 3 *})
    apply (tactic {* stac (thm zmod-zdvd-zmod) 3 *})
      apply (tactic {* stac (thm x-sol-lin) 5 *})
        apply (tactic {* stac (thm zmod-zmult-distrib RS sym) 7 *})
        apply (tactic {* stac (thm zcong-zmod RS sym) 7 *})
        apply (subgoal-tac [7]
          0 ≤ xilin-sol i n kf bf mf ∧ xilin-sol i n kf bf mf < mf i
          ∧ [kf i * mhf mf n i * xilin-sol i n kf bf mf = bf i] (mod mf i))
         prefer 7
         apply (simp add: zmult-ac)
        apply (unfold xilin-sol-def)
        apply (tactic {* Asm-simp-tac 7 *})
        apply (rule-tac [7] ex1-implies-ex [THEN someI-ex])
        apply (rule-tac [7] unique-xi-sol)
           apply (rule-tac [4] funprod-zdvd)
            apply (unfold m-cond-def)
            apply (rule funprod-pos [THEN pos-mod-sign])
            apply (rule-tac [2] funprod-pos [THEN pos-mod-bound])
            apply auto
  done

end
```

7 Bijections between sets

theory *BijectionRel* = *Main*:

Inductive definitions of bijections between two different sets and between the same set. Theorem for relating the two definitions.

consts
bijR :: ('*a* => '*b* => *bool*) => ('*a set* * '*b set*) *set*

inductive *bijR P*
intros
empty [*simp*]: ({}, {}) ∈ *bijR P*
insert: *P a b* ==> *a* ∉ *A* ==> *b* ∉ *B* ==> (*A*, *B*) ∈ *bijR P*
==> (*insert a A*, *insert b B*) ∈ *bijR P*

Add extra condition to *insert*: ∀ *b*∈*B*. ¬ *P a b* (and similar for *A*).

constdefs
bijP :: ('*a* => '*a* => *bool*) => '*a set* => *bool*
bijP P F == ∀ *a b*. *a* ∈ *F* ∧ *P a b* −−> *b* ∈ *F*

uniqP :: ('*a* => '*a* => *bool*) => *bool*
uniqP P == ∀ *a b c d*. *P a b* ∧ *P c d* −−> (*a* = *c*) = (*b* = *d*)

symP :: ('*a* => '*a* => *bool*) => *bool*
symP P == ∀ *a b*. *P a b* = *P b a*

consts
bijER :: ('*a* => '*a* => *bool*) => '*a set set*

inductive *bijER P*
intros
empty [*simp*]: {} ∈ *bijER P*
insert1: *P a a* ==> *a* ∉ *A* ==> *A* ∈ *bijER P* ==> *insert a A* ∈ *bijER P*
insert2: *P a b* ==> *a* ≠ *b* ==> *a* ∉ *A* ==> *b* ∉ *A* ==> *A* ∈ *bijER P*
==> *insert a* (*insert b A*) ∈ *bijER P*

bijR

lemma *fin-bijRl*: (*A*, *B*) ∈ *bijR P* ==> *finite A*
apply (*erule bijR.induct*)
apply *auto*
done

lemma *fin-bijRr*: $(A,\ B) \in bijR\ P \Longrightarrow finite\ B$
apply (*erule bijR.induct*)
apply *auto*
done

lemma *aux-induct*:
$finite\ F \Longrightarrow F \subseteq A \Longrightarrow P\ \{\} \Longrightarrow$
$(!!F\ a.\ F \subseteq A \Longrightarrow a \in A \Longrightarrow a \notin F \Longrightarrow P\ F \Longrightarrow P\ (insert\ a\ F))$
$\Longrightarrow P\ F$
proof −
case *rule-context*
assume *major*: *finite F*
and *subs*: $F \subseteq A$
show *?thesis*
apply (*rule subs* [*THEN rev-mp*])
apply (*rule major* [*THEN finite-induct*])
apply (*blast intro*: *rule-context*)+
done
qed

lemma *inj-func-bijR-aux1*:
$A \subseteq B \Longrightarrow a \notin A \Longrightarrow a \in B \Longrightarrow inj\text{-}on\ f\ B \Longrightarrow f\ a \notin f\ `\ A$
apply (*unfold inj-on-def*)
apply *auto*
done

lemma *inj-func-bijR-aux2*:
$\forall a.\ a \in A \longrightarrow P\ a\ (f\ a) \Longrightarrow inj\text{-}on\ f\ A \Longrightarrow finite\ A \Longrightarrow F <= A$
$\Longrightarrow (F,\ f\ `\ F) \in bijR\ P$
apply (*rule-tac* $F = F$ **and** $A = A$ **in** *aux-induct*)
apply (*rule finite-subset*)
apply *auto*
apply (*rule bijR.insert*)
apply (*rule-tac* [*3*] *inj-func-bijR-aux1*)
apply *auto*
done

lemma *inj-func-bijR*:
$\forall a.\ a \in A \longrightarrow P\ a\ (f\ a) \Longrightarrow inj\text{-}on\ f\ A \Longrightarrow finite\ A$
$\Longrightarrow (A,\ f\ `\ A) \in bijR\ P$
apply (*rule inj-func-bijR-aux2*)
apply *auto*
done

bijER

lemma *fin-bijER*: $A \in bijER\ P \Longrightarrow finite\ A$
apply (*erule bijER.induct*)
apply *auto*
done

lemma *aux1*:
$a \notin A \Longrightarrow a \notin B \Longrightarrow F \subseteq insert\ a\ A \Longrightarrow F \subseteq insert\ a\ B \Longrightarrow a \in F$
$\Longrightarrow \exists C.\ F = insert\ a\ C \wedge a \notin C \wedge C <= A \wedge C <= B$
apply (*rule-tac* $x = F - \{a\}$ **in** *exI*)
apply *auto*
done

lemma *aux2*: $a \neq b \Longrightarrow a \notin A \Longrightarrow b \notin B \Longrightarrow a \in F \Longrightarrow b \in F$
$\Longrightarrow F \subseteq insert\ a\ A \Longrightarrow F \subseteq insert\ b\ B$
$\Longrightarrow \exists C.\ F = insert\ a\ (insert\ b\ C) \wedge a \notin C \wedge b \notin C \wedge C \subseteq A \wedge C \subseteq B$
apply (*rule-tac* $x = F - \{a, b\}$ **in** *exI*)
apply *auto*
done

lemma *aux-uniq*: $uniqP\ P \Longrightarrow P\ a\ b \Longrightarrow P\ c\ d \Longrightarrow (a = c) = (b = d)$
apply (*unfold uniqP-def*)
apply *auto*
done

lemma *aux-sym*: $symP\ P \Longrightarrow P\ a\ b = P\ b\ a$
apply (*unfold symP-def*)
apply *auto*
done

lemma *aux-in1*:
$uniqP\ P \Longrightarrow b \notin C \Longrightarrow P\ b\ b \Longrightarrow bijP\ P\ (insert\ b\ C) \Longrightarrow bijP\ P\ C$
apply (*unfold bijP-def*)
apply *auto*
apply (*subgoal-tac* $b \neq a$)
prefer *2*
apply *clarify*
apply (*simp add*: *aux-uniq*)
apply *auto*
done

lemma *aux-in2*:
$symP\ P \Longrightarrow uniqP\ P \Longrightarrow a \notin C \Longrightarrow b \notin C \Longrightarrow a \neq b \Longrightarrow P\ a\ b$
$\Longrightarrow bijP\ P\ (insert\ a\ (insert\ b\ C)) \Longrightarrow bijP\ P\ C$
apply (*unfold bijP-def*)
apply *auto*

apply (*subgoal-tac aa* $\neq$ *a*)
prefer *2*
apply *clarify*
apply (*subgoal-tac aa* $\neq$ *b*)
prefer *2*
apply *clarify*
apply (*simp add*: *aux-uniq*)
apply (*subgoal-tac ba* $\neq$ *a*)
apply *auto*
apply (*subgoal-tac P a aa*)
prefer *2*
apply (*simp add*: *aux-sym*)
apply (*subgoal-tac b* = *aa*)
apply (*rule-tac* [*2*] *iffD1*)
apply (*rule-tac* [*2*] *a* = *a* **and** *c* = *a* **and** *P* = *P* **in** *aux-uniq*)
apply *auto*
done

lemma *aux-foo*: $\forall$ *a b*. *Q a* $\wedge$ *P a b* −−> *R b* ==> *P a b* ==> *Q a* ==> *R b*
apply *auto*
done

lemma *aux-bij*: *bijP P F* ==> *symP P* ==> *P a b* ==> (*a* $\in$ *F*) = (*b* $\in$ *F*)
apply (*unfold bijP-def*)
apply (*rule iffI*)
apply (*erule-tac* [!] *aux-foo*)
apply *simp-all*
apply (*rule iffD2*)
apply (*rule-tac P* = *P* **in** *aux-sym*)
apply *simp-all*
done

lemma *aux-bijRER*:
(*A*, *B*) $\in$ *bijR P* ==> *uniqP P* ==> *symP P*
==> $\forall$ *F*. *bijP P F* $\wedge$ *F* $\subseteq$ *A* $\wedge$ *F* $\subseteq$ *B* −−> *F* $\in$ *bijER P*
apply (*erule bijR.induct*)
apply *simp*
apply (*case-tac a* = *b*)
apply *clarify*
apply (*case-tac b* $\in$ *F*)
prefer *2*
apply (*simp add*: *subset-insert*)
apply (*cut-tac F* = *F* **and** *a* = *b* **and** *A* = *A* **and** *B* = *B* **in** *aux1*)
prefer *6*

```
      apply clarify
      apply (rule bijER.insert1)
       apply simp-all
  apply (subgoal-tac bijP P C)
   apply simp
  apply (rule aux-in1)
    apply simp-all
 apply clarify
 apply (case-tac a ∈ F)
  apply (case-tac [!] b ∈ F)
   apply (cut-tac F = F and a = a and b = b and A = A and B = B
     in aux2)
        apply (simp-all add: subset-insert)
   apply clarify
   apply (rule bijER.insert2)
     apply simp-all
   apply (subgoal-tac bijP P C)
    apply simp
   apply (rule aux-in2)
       apply simp-all
  apply (subgoal-tac b ∈ F)
   apply (rule-tac [2] iffD1)
    apply (rule-tac [2] a = a and F = F and P = P in aux-bij)
     apply (simp-all (no-asm-simp))
  apply (subgoal-tac [2] a ∈ F)
   apply (rule-tac [3] iffD2)
    apply (rule-tac [3] b = b and F = F and P = P in aux-bij)
     apply auto
 done

lemma bijR-bijER:
 (A, A) ∈ bijR P ==>
   bijP P A ==> uniqP P ==> symP P ==> A ∈ bijER P
 apply (cut-tac A = A and B = A and P = P in aux-bijRER)
   apply auto
 done

end
```

8 Factorial on integers

theory *IntFact* = *IntPrimes*:

Factorial on integers and recursively defined set including all Integers from *2* up to *a*. Plus definition of product of finite set.

```
consts
  zfact :: int => int
  ssetprod :: int set => int
  d22set :: int => int set

recdef zfact  measure ((λn. nat n) :: int => nat)
  zfact n = (if n ≤ 0 then 1 else n * zfact (n - 1))

defs
  ssetprod-def: ssetprod A == (if finite A then fold (op *) 1 A else 1)

recdef d22set  measure ((λa. nat a) :: int => nat)
  d22set a = (if 1 < a then insert a (d22set (a - 1)) else {})
```

ssetprod — product of finite set

```
lemma ssetprod-empty [simp]: ssetprod {} = 1
  apply (simp add: ssetprod-def)
  done

lemma ssetprod-insert [simp]:
    finite A ==> a ∉ A ==> ssetprod (insert a A) = a * ssetprod A
  by (simp add: ssetprod-def mult-left-commute LC.fold-insert [OF LC.intro])
```

d22set — recursively defined set including all integers from *2* up to *a*

```
declare d22set.simps [simp del]


lemma d22set-induct:
  (!!a. P {} a) ==>
    (!!a. 1 < (a::int) ==> P (d22set (a - 1)) (a - 1)
      ==> P (d22set a) a)
    ==> P (d22set u) u
proof -
  case rule-context
  show ?thesis
    apply (rule d22set.induct)
    apply safe
     apply (case-tac [2] 1 < a)
      apply (rule-tac [2] rule-context)
       apply (simp-all (no-asm-simp))
```

```
    apply (simp-all (no-asm-simp) add: d22set.simps rule-context)
   done
qed

lemma d22set-g-1 [rule-format]: b ∈ d22set a --> 1 < b
 apply (induct a rule: d22set-induct)
  prefer 2
  apply (subst d22set.simps)
  apply auto
 done

lemma d22set-le [rule-format]: b ∈ d22set a --> b ≤ a
 apply (induct a rule: d22set-induct)
  prefer 2
  apply (subst d22set.simps)
  apply auto
 done

lemma d22set-le-swap: a < b ==> b ∉ d22set a
 apply (auto dest: d22set-le)
 done

lemma d22set-mem [rule-format]: 1 < b --> b ≤ a --> b ∈ d22set a
 apply (induct a rule: d22set.induct)
 apply auto
  apply (simp-all add: d22set.simps)
 done

lemma d22set-fin: finite (d22set a)
 apply (induct a rule: d22set-induct)
  prefer 2
  apply (subst d22set.simps)
  apply auto
 done


declare zfact.simps [simp del]

lemma d22set-prod-zfact: ssetprod (d22set a) = zfact a
 apply (induct a rule: d22set.induct)
 apply safe
  apply (simp add: d22set.simps zfact.simps)
 apply (subst d22set.simps)
 apply (subst zfact.simps)
 apply (case-tac 1 < a)
```

prefer *2*
apply (*simp add*: *d22set.simps zfact.simps*)
apply (*simp add*: *d22set-fin d22set-le-swap*)
done

end

9 Fermat's Little Theorem extended to Euler's Totient function

theory *EulerFermat* = *BijectionRel* + *IntFact*:

Fermat's Little Theorem extended to Euler's Totient function. More abstract approach than Boyer-Moore (which seems necessary to achieve the extended version).

9.1 Definitions and lemmas

consts
RsetR :: *int* => *int set set*
BnorRset :: *int* * *int* => *int set*
norRRset :: *int* => *int set*
noXRRset :: *int* => *int* => *int set*
phi :: *int* => *nat*
is-RRset :: *int set* => *int* => *bool*
RRset2norRR :: *int set* => *int* => *int* => *int*

inductive *RsetR m*
intros
empty [*simp*]: {} ∈ *RsetR m*
insert: *A* ∈ *RsetR m* ==> *zgcd* (*a*, *m*) = *1* ==>
∀ *a'*. *a'* ∈ *A* --> ¬ *zcong a a' m* ==> *insert a A* ∈ *RsetR m*

recdef *BnorRset*
measure ((λ(*a*, *m*). *nat a*) :: *int* * *int* => *nat*)
BnorRset (*a*, *m*) =
(*if 0* < *a then*
let na = *BnorRset* (*a* − *1*, *m*)
in (*if zgcd* (*a*, *m*) = *1 then insert a na else na*)
else {})

defs

norRRset-def: *norRRset m == BnorRset (m − 1, m)*
noXRRset-def: *noXRRset m x == (λa. a * x) ‘ norRRset m*
phi-def: *phi m == card (norRRset m)*
is-RRset-def: *is-RRset A m == A ∈ RsetR m ∧ card A = phi m*
RRset2norRR-def:
RRset2norRR A m a ==
(if 1 < m ∧ is-RRset A m ∧ a ∈ A then
SOME b. zcong a b m ∧ b ∈ norRRset m
else 0)

constdefs
zcongm :: int => int => int => bool
zcongm m == λa b. zcong a b m

lemma *abs-eq-1-iff* [*iff*]: *(abs z = (1::int)) = (z = 1 ∨ z = −1)*
— LCP: not sure why this lemma is needed now
by (*auto simp add*: *zabs-def*)

norRRset

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

lemma *BnorRset-induct*:
(!!a m. P {} a m) ==>
(!!a m. 0 < (a::int) ==> P (BnorRset (a − 1, m::int)) (a − 1) m
==> P (BnorRset(a,m)) a m)
==> P (BnorRset(u,v)) u v
proof −
case *rule-context*
show *?thesis*
apply (*rule BnorRset.induct*, *safe*)
apply (*case-tac* [*2*] *0 < a*)
apply (*rule-tac* [*2*] *rule-context*, *simp-all*)
apply (*simp-all add*: *BnorRset.simps rule-context*)
done
qed

lemma *Bnor-mem-zle* [*rule-format*]: *b ∈ BnorRset (a, m) −−> b ≤ a*
apply (*induct a m rule*: *BnorRset-induct*)
prefer *2*
apply (*subst BnorRset.simps*)
apply (*unfold Let-def*, *auto*)
done

lemma *Bnor-mem-zle-swap*: *a < b ==> b ∉ BnorRset (a, m)*

by (*auto dest*: *Bnor-mem-zle*)

lemma *Bnor-mem-zg* [*rule-format*]: $b \in BnorRset\ (a,\ m) \longrightarrow 0 < b$
apply (*induct a m rule*: *BnorRset-induct*)
prefer *2*
apply (*subst BnorRset.simps*)
apply (*unfold Let-def*, *auto*)
done

lemma *Bnor-mem-if* [*rule-format*]:
$zgcd\ (b,\ m) = 1 \longrightarrow 0 < b \longrightarrow b \leq a \longrightarrow b \in BnorRset\ (a,\ m)$
apply (*induct a m rule*: *BnorRset.induct*, *auto*)
apply (*case-tac* $a = b$)
prefer *2*
apply (*simp add*: *order-less-le*)
apply (*simp* (*no-asm-simp*))
prefer *2*
apply (*subst BnorRset.simps*)
defer
apply (*subst BnorRset.simps*)
apply (*unfold Let-def*, *auto*)
done

lemma *Bnor-in-RsetR* [*rule-format*]: $a < m \longrightarrow BnorRset\ (a,\ m) \in RsetR\ m$
apply (*induct a m rule*: *BnorRset-induct*, *simp*)
apply (*subst BnorRset.simps*)
apply (*unfold Let-def*, *auto*)
apply (*rule RsetR.insert*)
apply (*rule-tac* [*3*] *allI*)
apply (*rule-tac* [*3*] *impI*)
apply (*rule-tac* [*3*] *zcong-not*)
apply (*subgoal-tac* [*6*] $a' \leq a - 1$)
apply (*rule-tac* [*7*] *Bnor-mem-zle*)
apply (*rule-tac* [*5*] *Bnor-mem-zg*, *auto*)
done

lemma *Bnor-fin*: *finite* (*BnorRset* (*a*, *m*))
apply (*induct a m rule*: *BnorRset-induct*)
prefer *2*
apply (*subst BnorRset.simps*)
apply (*unfold Let-def*, *auto*)
done

lemma *norR-mem-unique-aux*: $a \leq b - 1 \Longrightarrow a < (b::int)$
apply *auto*

done

lemma *norR-mem-unique*:
1 < m ==>
zgcd (a, m) = 1 ==> ∃!b. [a = b] (mod m) ∧ b ∈ norRRset m
apply (*unfold norRRset-def*)
apply (*cut-tac a = a* **and** *m = m* **in** *zcong-zless-unique*, *auto*)
apply (*rule-tac [2] m = m* **in** *zcong-zless-imp-eq*)
apply (*auto intro*: *Bnor-mem-zle Bnor-mem-zg zcong-trans order-less-imp-le norR-mem-unique-aux simp add*: *zcong-sym*)
apply (*rule-tac x = b* **in** *exI*, *safe*)
apply (*rule Bnor-mem-if*)
apply (*case-tac [2] b = 0*)
apply (*auto intro*: *order-less-le* [*THEN iffD2*])
prefer *2*
apply (*simp only*: *zcong-def*)
apply (*subgoal-tac zgcd (a, m) = m*)
prefer *2*
apply (*subst zdvd-iff-zgcd* [*symmetric*])
apply (*rule-tac [4] zgcd-zcong-zgcd*)
apply (*simp-all add*: *zdvd-zminus-iff zcong-sym*)
done

noXRRset

lemma *RRset-gcd* [*rule-format*]:
is-RRset A m ==> a ∈ A --> zgcd (a, m) = 1
apply (*unfold is-RRset-def*)
apply (*rule RsetR.induct*, *auto*)
done

lemma *RsetR-zmult-mono*:
A ∈ RsetR m ==>
*0 < m ==> zgcd (x, m) = 1 ==> (λa. a * x) ‘ A ∈ RsetR m*
apply (*erule RsetR.induct*, *simp-all*)
apply (*rule RsetR.insert*, *auto*)
apply (*blast intro*: *zgcd-zgcd-zmult*)
apply (*simp add*: *zcong-cancel*)
done

lemma *card-nor-eq-noX*:
0 < m ==>
zgcd (x, m) = 1 ==> card (noXRRset m x) = card (norRRset m)
apply (*unfold norRRset-def noXRRset-def*)
apply (*rule card-image*)

apply (*auto simp add*: *inj-on-def Bnor-fin*)
apply (*simp add*: *BnorRset.simps*)
done

lemma *noX-is-RRset*:
0 < m ==> zgcd (x, m) = 1 ==> is-RRset (noXRRset m x) m
apply (*unfold is-RRset-def phi-def*)
apply (*auto simp add*: *card-nor-eq-noX*)
apply (*unfold noXRRset-def norRRset-def*)
apply (*rule RsetR-zmult-mono*)
apply (*rule Bnor-in-RsetR*, *simp-all*)
done

lemma *aux-some*:
1 < m ==> is-RRset A m ==> a ∈ A
==> zcong a (SOME b. [a = b] (mod m) ∧ b ∈ norRRset m) m ∧
(SOME b. [a = b] (mod m) ∧ b ∈ norRRset m) ∈ norRRset m
apply (*rule norR-mem-unique* [*THEN ex1-implies-ex*, *THEN someI-ex*])
apply (*rule-tac* [*2*] *RRset-gcd*, *simp-all*)
done

lemma *RRset2norRR-correct*:
1 < m ==> is-RRset A m ==> a ∈ A ==>
[a = RRset2norRR A m a] (mod m) ∧ RRset2norRR A m a ∈ norRRset m
apply (*unfold RRset2norRR-def*, *simp*)
apply (*rule aux-some*, *simp-all*)
done

lemmas *RRset2norRR-correct1 =*
RRset2norRR-correct [*THEN conjunct1*, *standard*]
lemmas *RRset2norRR-correct2 =*
RRset2norRR-correct [*THEN conjunct2*, *standard*]

lemma *RsetR-fin*: *A ∈ RsetR m ==> finite A*
by (*erule RsetR.induct*, *auto*)

lemma *RRset-zcong-eq* [*rule-format*]:
1 < m ==>
is-RRset A m ==> [a = b] (mod m) ==> a ∈ A --> b ∈ A --> a = b
apply (*unfold is-RRset-def*)
apply (*rule RsetR.induct*)
apply (*auto simp add*: *zcong-sym*)
done

lemma *aux*:

```
  P (SOME a. P a) ==> Q (SOME a. Q a) ==>
   (SOME a. P a) = (SOME a. Q a) ==> ∃ a. P a ∧ Q a
  apply auto
  done

lemma RRset2norRR-inj:
    1 < m ==> is-RRset A m ==> inj-on (RRset2norRR A m) A
  apply (unfold RRset2norRR-def inj-on-def, auto)
  apply (subgoal-tac ∃ b. ([x = b] (mod m) ∧ b ∈ norRRset m) ∧
     ([y = b] (mod m) ∧ b ∈ norRRset m))
   apply (rule-tac [2] aux)
    apply (rule-tac [3] aux-some)
     apply (rule-tac [2] aux-some)
      apply (rule RRset-zcong-eq, auto)
  apply (rule-tac b = b in zcong-trans)
   apply (simp-all add: zcong-sym)
  done

lemma RRset2norRR-eq-norR:
    1 < m ==> is-RRset A m ==> RRset2norRR A m ` A = norRRset m
  apply (rule card-seteq)
    prefer 3
    apply (subst card-image)
     apply (rule-tac [2] RRset2norRR-inj, auto)
    apply (rule-tac [4] RRset2norRR-correct2, auto)
    apply (unfold is-RRset-def phi-def norRRset-def)
    apply (auto simp add: RsetR-fin Bnor-fin)
  done


lemma Bnor-prod-power-aux: a ∉ A ==> inj f ==> f a ∉ f ` A
by (unfold inj-on-def, auto)

lemma Bnor-prod-power [rule-format]:
  x ≠ 0 ==> a < m --> ssetprod ((λa. a * x) ` BnorRset (a, m)) =
      ssetprod (BnorRset(a, m)) * x^card (BnorRset (a, m))
  apply (induct a m rule: BnorRset-induct)
   prefer 2
   apply (subst BnorRset.simps)
   apply (unfold Let-def, auto)
  apply (simp add: Bnor-fin Bnor-mem-zle-swap)
  apply (subst ssetprod-insert)
    apply (rule-tac [2] Bnor-prod-power-aux)
     apply (unfold inj-on-def)
     apply (simp-all add: zmult-ac Bnor-fin finite-imageI
```

Bnor-mem-zle-swap)
done

9.2 Fermat

lemma *bijzcong-zcong-prod*:
(A, B) ∈ bijR (zcongm m) ==> [ssetprod A = ssetprod B] (mod m)
apply (*unfold zcongm-def*)
apply (*erule bijR.induct*)
apply (*subgoal-tac [2] a ∉ A ∧ b ∉ B ∧ finite A ∧ finite B*)
apply (*auto intro*: *fin-bijRl fin-bijRr zcong-zmult*)
done

lemma *Bnor-prod-zgcd* [*rule-format*]:
a < m --> zgcd (ssetprod (BnorRset (a, m)), m) = 1
apply (*induct a m rule*: *BnorRset-induct*)
prefer *2*
apply (*subst BnorRset.simps*)
apply (*unfold Let-def*, *auto*)
apply (*simp add*: *Bnor-fin Bnor-mem-zle-swap*)
apply (*blast intro*: *zgcd-zgcd-zmult*)
done

theorem *Euler-Fermat*:
0 < m ==> zgcd (x, m) = 1 ==> [x^(phi m) = 1] (mod m)
apply (*unfold norRRset-def phi-def*)
apply (*case-tac x = 0*)
apply (*case-tac [2] m = 1*)
apply (*rule-tac [3] iffD1*)
apply (*rule-tac [3] k = ssetprod (BnorRset (m − 1, m))*
in *zcong-cancel2*)
prefer *5*
apply (*subst Bnor-prod-power* [*symmetric*])
apply (*rule-tac [7] Bnor-prod-zgcd*, *simp-all*)
apply (*rule bijzcong-zcong-prod*)
apply (*fold norRRset-def noXRRset-def*)
apply (*subst RRset2norRR-eq-norR* [*symmetric*])
apply (*rule-tac [3] inj-func-bijR*, *auto*)
apply (*unfold zcongm-def*)
apply (*rule-tac [2] RRset2norRR-correct1*)
apply (*rule-tac [5] RRset2norRR-inj*)
apply (*auto intro*: *order-less-le* [*THEN iffD2*]
simp add: *noX-is-RRset*)
apply (*unfold noXRRset-def norRRset-def*)
apply (*rule finite-imageI*)

apply (*rule Bnor-fin*)
done

lemma *Bnor-prime* [*rule-format* (*no-asm*)]:
p ∈ *zprime* ==>
a < *p* --> (∀ *b*. *0* < *b* ∧ *b* ≤ *a* --> *zgcd* (*b*, *p*) = *1*)
--> *card* (*BnorRset* (*a*, *p*)) = *nat a*
apply (*auto simp add*: *zless-zprime-imp-zrelprime*)
apply (*induct a p rule*: *BnorRset.induct*)
apply (*subst BnorRset.simps*)
apply (*unfold Let-def*, *auto*)
apply (*subgoal-tac finite* (*BnorRset* (*a* − *1*,*m*)))
apply (*subgoal-tac a* ~: *BnorRset* (*a* − *1*,*m*))
apply (*auto simp add*: *card-insert-disjoint Suc-nat-eq-nat-zadd1*)
apply (*frule Bnor-mem-zle*, *arith*)
apply (*frule Bnor-fin*)
done

lemma *phi-prime*: *p* ∈ *zprime* ==> *phi p* = *nat* (*p* − *1*)
apply (*unfold phi-def norRRset-def*)
apply (*rule Bnor-prime*, *auto*)
apply (*erule zless-zprime-imp-zrelprime*, *simp-all*)
done

theorem *Little-Fermat*:
p ∈ *zprime* ==> ¬ *p dvd x* ==> [*x*^(*nat* (*p* − *1*)) = *1*] (*mod p*)
apply (*subst phi-prime* [*symmetric*])
apply (*rule-tac* [*2*] *Euler-Fermat*)
apply (*erule-tac* [*3*] *zprime-imp-zrelprime*)
apply (*unfold zprime-def*, *auto*)
done

end

10 Wilson's Theorem according to Russinoff

theory *WilsonRuss* = *EulerFermat*:

Wilson's Theorem following quite closely Russinoff's approach using Boyer-Moore (using finite sets instead of lists, though).

10.1 Definitions and lemmas

consts
inv :: *int* => *int* => *int*
wset :: *int* * *int* => *int set*

defs
inv-def: *inv p a* == (*a*^(*nat* (*p* − *2*))) *mod p*

recdef *wset*
measure ((λ(*a*, *p*). *nat a*) :: *int* * *int* => *nat*)
wset (*a*, *p*) =
(*if 1* < *a then*
let ws = *wset* (*a* − *1*, *p*)
in (*if a* ∈ *ws then ws else insert a* (*insert* (*inv p a*) *ws*)) *else* {})

inv

lemma *inv-is-inv-aux*: *1* < *m* ==> *Suc* (*nat* (*m* − *2*)) = *nat* (*m* − *1*)
by (*subst int-int-eq* [*symmetric*], *auto*)

lemma *inv-is-inv*:
p ∈ *zprime* ⟹ *0* < *a* ⟹ *a* < *p* ==> [*a* * *inv p a* = *1*] (*mod p*)
apply (*unfold inv-def*)
apply (*subst zcong-zmod*)
apply (*subst zmod-zmult1-eq* [*symmetric*])
apply (*subst zcong-zmod* [*symmetric*])
apply (*subst power-Suc* [*symmetric*])
apply (*subst inv-is-inv-aux*)
apply (*erule-tac* [*2*] *Little-Fermat*)
apply (*erule-tac* [*2*] *zdvd-not-zless*)
apply (*unfold zprime-def*, *auto*)
done

lemma *inv-distinct*:
p ∈ *zprime* ⟹ *1* < *a* ⟹ *a* < *p* − *1* ==> *a* ≠ *inv p a*
apply *safe*
apply (*cut-tac a* = *a* **and** *p* = *p* **in** *zcong-square*)
apply (*cut-tac* [*3*] *a* = *a* **and** *p* = *p* **in** *inv-is-inv*, *auto*)
apply (*subgoal-tac a* = *1*)
apply (*rule-tac* [*2*] *m* = *p* **in** *zcong-zless-imp-eq*)
apply (*subgoal-tac* [*7*] *a* = *p* − *1*)
apply (*rule-tac* [*8*] *m* = *p* **in** *zcong-zless-imp-eq*, *auto*)
done

lemma *inv-not-0*:

```
  p ∈ zprime ⟹ 1 < a ⟹ a < p − 1 ==> inv p a ≠ 0
 apply safe
 apply (cut-tac a = a and p = p in inv-is-inv)
   apply (unfold zcong-def, auto)
 apply (subgoal-tac ¬ p dvd 1)
  apply (rule-tac [2] zdvd-not-zless)
   apply (subgoal-tac p dvd 1)
    prefer 2
    apply (subst zdvd-zminus-iff [symmetric], auto)
 done

lemma inv-not-1:
  p ∈ zprime ⟹ 1 < a ⟹ a < p − 1 ==> inv p a ≠ 1
 apply safe
 apply (cut-tac a = a and p = p in inv-is-inv)
   prefer 4
   apply simp
   apply (subgoal-tac a = 1)
    apply (rule-tac [2] zcong-zless-imp-eq, auto)
 done

lemma inv-not-p-minus-1-aux: [a * (p − 1) = 1] (mod p) = [a = p − 1] (mod
p)
 apply (unfold zcong-def)
 apply (simp add: Ring-and-Field.diff-diff-eq diff-diff-eq2 zdiff-zmult-distrib2)
 apply (rule-tac s = p dvd −((a + 1) + (p * −a)) in trans)
  apply (simp add: mult-commute)
 apply (subst zdvd-zminus-iff)
 apply (subst zdvd-reduce)
 apply (rule-tac s = p dvd (a + 1) + (p * −1) in trans)
  apply (subst zdvd-reduce, auto)
 done

lemma inv-not-p-minus-1:
  p ∈ zprime ⟹ 1 < a ⟹ a < p − 1 ==> inv p a ≠ p − 1
 apply safe
 apply (cut-tac a = a and p = p in inv-is-inv, auto)
 apply (simp add: inv-not-p-minus-1-aux)
 apply (subgoal-tac a = p − 1)
  apply (rule-tac [2] zcong-zless-imp-eq, auto)
 done

lemma inv-g-1:
  p ∈ zprime ⟹ 1 < a ⟹ a < p − 1 ==> 1 < inv p a
 apply (case-tac 0≤ inv p a)
```

```
  apply (subgoal-tac inv p a ≠ 1)
   apply (subgoal-tac inv p a ≠ 0)
    apply (subst order-less-le)
    apply (subst zle-add1-eq-le [symmetric])
    apply (subst order-less-le)
    apply (rule-tac [2] inv-not-0)
     apply (rule-tac [5] inv-not-1, auto)
  apply (unfold inv-def zprime-def, simp)
  done

lemma inv-less-p-minus-1:
    p ∈ zprime ⟹ 1 < a ⟹ a < p - 1 ==> inv p a < p - 1
  apply (case-tac inv p a < p)
   apply (subst order-less-le)
   apply (simp add: inv-not-p-minus-1, auto)
  apply (unfold inv-def zprime-def, simp)
  done

lemma inv-inv-aux: 5 ≤ p ==>
    nat (p - 2) * nat (p - 2) = Suc (nat (p - 1) * nat (p - 3))
  apply (subst int-int-eq [symmetric])
  apply (simp add: zmult-int [symmetric])
  apply (simp add: zdiff-zmult-distrib zdiff-zmult-distrib2)
  done

lemma zcong-zpower-zmult:
    [x^y = 1] (mod p) ⟹ [x^(y * z) = 1] (mod p)
  apply (induct z)
   apply (auto simp add: zpower-zadd-distrib)
  apply (subgoal-tac zcong (x^y * x^(y * n)) (1 * 1) p)
   apply (rule-tac [2] zcong-zmult, simp-all)
  done

lemma inv-inv: p ∈ zprime ⟹
    5 ≤ p ⟹ 0 < a ⟹ a < p ==> inv p (inv p a) = a
  apply (unfold inv-def)
  apply (subst zpower-zmod)
  apply (subst zpower-zpower)
  apply (rule zcong-zless-imp-eq)
      prefer 5
      apply (subst zcong-zmod)
      apply (subst mod-mod-trivial)
      apply (subst zcong-zmod [symmetric])
      apply (subst inv-inv-aux)
       apply (subgoal-tac [2]
```

```
    zcong (a * a^(nat (p - 1) * nat (p - 3))) (a * 1) p)
   apply (rule-tac [3] zcong-zmult)
    apply (rule-tac [4] zcong-zpower-zmult)
    apply (erule-tac [4] Little-Fermat)
    apply (rule-tac [4] zdvd-not-zless, simp-all)
  done


wset

declare wset.simps [simp del]

lemma wset-induct:
  (!!a p. P {} a p) ==>
    (!!a p. 1 < (a::int) ==> P (wset (a - 1, p)) (a - 1) p
      ==> P (wset (a, p)) a p)
    ==> P (wset (u, v)) u v
proof -
  case rule-context
  show ?thesis
    apply (rule wset.induct, safe)
     apply (case-tac [2] 1 < a)
      apply (rule-tac [2] rule-context, simp-all)
      apply (simp-all add: wset.simps rule-context)
    done
qed

lemma wset-mem-imp-or [rule-format]:
  1 < a ==> b ~: wset (a - 1, p)
    ==> b : wset (a, p) --> b = a | b = inv p a
  apply (subst wset.simps)
  apply (unfold Let-def, simp)
  done

lemma wset-mem-mem [simp]: 1 < a ==> a : wset (a, p)
  apply (subst wset.simps)
  apply (unfold Let-def, simp)
  done

lemma wset-subset: 1 < a ==> b : wset (a - 1, p) ==> b : wset (a, p)
  apply (subst wset.simps)
  apply (unfold Let-def, auto)
  done

lemma wset-g-1 [rule-format]:
    p : zprime --> a < p - 1 --> b : wset (a, p) --> 1 < b
```

```
  apply (induct a p rule: wset-induct, auto)
  apply (case-tac b = a)
   apply (case-tac [2] b = inv p a)
    apply (subgoal-tac [3] b = a ∨ b = inv p a)
     apply (rule-tac [4] wset-mem-imp-or)
       prefer 2
       apply simp
       apply (rule inv-g-1, auto)
  done

lemma wset-less [rule-format]:
    p ∈ zprime --> a < p - 1 --> b ∈ wset (a, p) --> b < p - 1
  apply (induct a p rule: wset-induct, auto)
  apply (case-tac b = a)
   apply (case-tac [2] b = inv p a)
    apply (subgoal-tac [3] b = a ∨ b = inv p a)
     apply (rule-tac [4] wset-mem-imp-or)
       prefer 2
       apply simp
       apply (rule inv-less-p-minus-1, auto)
  done

lemma wset-mem [rule-format]:
  p ∈ zprime -->
    a < p - 1 --> 1 < b --> b ≤ a --> b ∈ wset (a, p)
  apply (induct a p rule: wset.induct, auto)
   apply (subgoal-tac b = a)
    apply (rule-tac [2] zle-anti-sym)
     apply (rule-tac [4] wset-subset)
      apply (simp (no-asm-simp))
     apply auto
  done

lemma wset-mem-inv-mem [rule-format]:
  p ∈ zprime --> 5 ≤ p --> a < p - 1 --> b ∈ wset (a, p)
    --> inv p b ∈ wset (a, p)
  apply (induct a p rule: wset-induct, auto)
   apply (case-tac b = a)
    apply (subst wset.simps)
    apply (unfold Let-def)
    apply (rule-tac [3] wset-subset, auto)
  apply (case-tac b = inv p a)
   apply (simp (no-asm-simp))
   apply (subst inv-inv)
      apply (subgoal-tac [6] b = a ∨ b = inv p a)
```

```
      apply (rule-tac [7] wset-mem-imp-or, auto)
  done

lemma wset-inv-mem-mem:
  p ∈ zprime ⟹ 5 ≤ p ⟹ a < p − 1 ⟹ 1 < b ⟹ b < p − 1
    ⟹ inv p b ∈ wset (a, p) ⟹ b ∈ wset (a, p)
  apply (rule-tac s = inv p (inv p b) and t = b in subst)
   apply (rule-tac [2] wset-mem-inv-mem)
     apply (rule inv-inv, simp-all)
  done

lemma wset-fin: finite (wset (a, p))
  apply (induct a p rule: wset-induct)
   prefer 2
   apply (subst wset.simps)
   apply (unfold Let-def, auto)
  done

lemma wset-zcong-prod-1 [rule-format]:
  p ∈ zprime −−>
    5 ≤ p −−> a < p − 1 −−> [ssetprod (wset (a, p)) = 1] (mod p)
  apply (induct a p rule: wset-induct)
   prefer 2
   apply (subst wset.simps)
   apply (unfold Let-def, auto)
  apply (subst ssetprod-insert)
    apply (tactic {* stac (thm ssetprod-insert) 3 *})
     apply (subgoal-tac [5]
       zcong (a * inv p a * ssetprod (wset (a − 1, p))) (1 * 1) p)
      prefer 5
      apply (simp add: zmult-assoc)
     apply (rule-tac [5] zcong-zmult)
      apply (rule-tac [5] inv-is-inv)
        apply (tactic Clarify-tac 4)
        apply (subgoal-tac [4] a ∈ wset (a − 1, p))
         apply (rule-tac [5] wset-inv-mem-mem)
             apply (simp-all add: wset-fin)
  apply (rule inv-distinct, auto)
  done

lemma d22set-eq-wset: p ∈ zprime ==> d22set (p − 2) = wset (p − 2, p)
  apply safe
   apply (erule wset-mem)
    apply (rule-tac [2] d22set-g-1)
    apply (rule-tac [3] d22set-le)
```

```
    apply (rule-tac [4] d22set-mem)
     apply (erule-tac [4] wset-g-1)
      prefer 6
      apply (subst zle-add1-eq-le [symmetric])
      apply (subgoal-tac p - 2 + 1 = p - 1)
       apply (simp (no-asm-simp))
       apply (erule wset-less, auto)
  done
```

10.2 Wilson

```
lemma prime-g-5: p ∈ zprime ⟹ p ≠ 2 ⟹ p ≠ 3 ==> 5 ≤ p
  apply (unfold zprime-def dvd-def)
  apply (case-tac p = 4, auto)
   apply (rule notE)
    prefer 2
    apply assumption
   apply (simp (no-asm))
   apply (rule-tac x = 2 in exI)
   apply (safe, arith)
    apply (rule-tac x = 2 in exI, auto)
  done

theorem Wilson-Russ:
    p ∈ zprime ==> [zfact (p - 1) = -1] (mod p)
  apply (subgoal-tac [(p - 1) * zfact (p - 2) = -1 * 1] (mod p))
   apply (rule-tac [2] zcong-zmult)
    apply (simp only: zprime-def)
    apply (subst zfact.simps)
    apply (rule-tac t = p - 1 - 1 and s = p - 2 in subst, auto)
   apply (simp only: zcong-def)
   apply (simp (no-asm-simp))
  apply (case-tac p = 2)
   apply (simp add: zfact.simps)
  apply (case-tac p = 3)
   apply (simp add: zfact.simps)
  apply (subgoal-tac 5 ≤ p)
   apply (erule-tac [2] prime-g-5)
    apply (subst d22set-prod-zfact [symmetric])
    apply (subst d22set-eq-wset)
     apply (rule-tac [2] wset-zcong-prod-1, auto)
  done

end
```

11 Wilson's Theorem using a more abstract approach

theory *WilsonBij* = *BijectionRel* + *IntFact*:

Wilson's Theorem using a more "abstract" approach based on bijections between sets. Does not use Fermat's Little Theorem (unlike Russinoff).

11.1 Definitions and lemmas

constdefs
  reciR :: *int* => *int* => *int* => *bool*
  reciR p ==
    λ*a b*. *zcong* (*a* ∗ *b*) *1 p* ∧ *1* < *a* ∧ *a* < *p* − *1* ∧ *1* < *b* ∧ *b* < *p* − *1*
  inv :: *int* => *int* => *int*
  inv p a ==
    if p ∈ *zprime* ∧ *0* < *a* ∧ *a* < *p then*
      (*SOME x*. *0* ≤ *x* ∧ *x* < *p* ∧ *zcong* (*a* ∗ *x*) *1 p*)
    else 0

Inverse

lemma *inv-correct*:
  p ∈ *zprime* ==> *0* < *a* ==> *a* < *p*
    ==> *0* ≤ *inv p a* ∧ *inv p a* < *p* ∧ [*a* ∗ *inv p a* = *1*] (*mod p*)
  apply (*unfold inv-def*)
  apply (*simp* (*no-asm-simp*))
  apply (*rule zcong-lineq-unique* [*THEN ex1-implies-ex*, *THEN someI-ex*])
   apply (*erule-tac* [*2*] *zless-zprime-imp-zrelprime*)
    apply (*unfold zprime-def*)
    apply *auto*
  done

lemmas *inv-ge* = *inv-correct* [*THEN conjunct1*, *standard*]
lemmas *inv-less* = *inv-correct* [*THEN conjunct2*, *THEN conjunct1*, *standard*]
lemmas *inv-is-inv* = *inv-correct* [*THEN conjunct2*, *THEN conjunct2*, *standard*]

lemma *inv-not-0*:
  p ∈ *zprime* ==> *1* < *a* ==> *a* < *p* − *1* ==> *inv p a* ≠ *0*
  — same as *WilsonRuss*
  apply *safe*
  apply (*cut-tac a* = *a* **and** *p* = *p* **in** *inv-is-inv*)
    apply (*unfold zcong-def*)
    apply *auto*

```
  apply (subgoal-tac ¬ p dvd 1)
   apply (rule-tac [2] zdvd-not-zless)
    apply (subgoal-tac p dvd 1)
     prefer 2
     apply (subst zdvd-zminus-iff [symmetric])
     apply auto
  done

lemma inv-not-1:
  p ∈ zprime ==> 1 < a ==> a < p - 1 ==> inv p a ≠ 1
  — same as WilsonRuss
  apply safe
  apply (cut-tac a = a and p = p in inv-is-inv)
     prefer 4
     apply simp
     apply (subgoal-tac a = 1)
      apply (rule-tac [2] zcong-zless-imp-eq)
         apply auto
  done

lemma aux: [a * (p - 1) = 1] (mod p) = [a = p - 1] (mod p)
  — same as WilsonRuss
  apply (unfold zcong-def)
  apply (simp add: Ring-and-Field.diff-diff-eq diff-diff-eq2 zdiff-zmult-distrib2)
  apply (rule-tac s = p dvd -((a + 1) + (p * -a)) in trans)
   apply (simp add: mult-commute)
  apply (subst zdvd-zminus-iff)
  apply (subst zdvd-reduce)
  apply (rule-tac s = p dvd (a + 1) + (p * -1) in trans)
   apply (subst zdvd-reduce)
   apply auto
  done

lemma inv-not-p-minus-1:
  p ∈ zprime ==> 1 < a ==> a < p - 1 ==> inv p a ≠ p - 1
  — same as WilsonRuss
  apply safe
  apply (cut-tac a = a and p = p in inv-is-inv)
     apply auto
  apply (simp add: aux)
  apply (subgoal-tac a = p - 1)
   apply (rule-tac [2] zcong-zless-imp-eq)
      apply auto
  done
```

Below is slightly different as we don't expand *inv* but use "*correct*" theorems.

lemma *inv-g-1*: $p \in zprime \Longrightarrow 1 < a \Longrightarrow a < p - 1 \Longrightarrow 1 < inv\ p\ a$
apply (*subgoal-tac inv p a* $\neq$ *1*)
apply (*subgoal-tac inv p a* $\neq$ *0*)
apply (*subst order-less-le*)
apply (*subst zle-add1-eq-le* [*symmetric*])
apply (*subst order-less-le*)
apply (*rule-tac* [*2*] *inv-not-0*)
apply (*rule-tac* [*5*] *inv-not-1*)
apply *auto*
apply (*rule inv-ge*)
apply *auto*
done

lemma *inv-less-p-minus-1*:
$p \in zprime \Longrightarrow 1 < a \Longrightarrow a < p - 1 \Longrightarrow inv\ p\ a < p - 1$
— ditto
apply (*subst order-less-le*)
apply (*simp add*: *inv-not-p-minus-1 inv-less*)
done

Bijection

lemma *aux1*: $1 < x \Longrightarrow 0 \leq (x::int)$
apply *auto*
done

lemma *aux2*: $1 < x \Longrightarrow 0 < (x::int)$
apply *auto*
done

lemma *aux3*: $x \leq p - 2 \Longrightarrow x < (p::int)$
apply *auto*
done

lemma *aux4*: $x \leq p - 2 \Longrightarrow x < (p::int) - 1$
apply *auto*
done

lemma *inv-inj*: $p \in zprime \Longrightarrow inj\text{-}on\ (inv\ p)\ (d22set\ (p - 2))$
apply (*unfold inj-on-def*)
apply *auto*
apply (*rule zcong-zless-imp-eq*)
apply (*tactic* {$*$ *stac* (*thm zcong-cancel RS sym*) *5* $*$})
apply (*rule-tac* [*7*] *zcong-trans*)

```
        apply (tactic {* stac (thm zcong-sym) 8 *})
        apply (erule-tac [7] inv-is-inv)
         apply (tactic Asm-simp-tac 9)
         apply (erule-tac [9] inv-is-inv)
          apply (rule-tac [6] zless-zprime-imp-zrelprime)
            apply (rule-tac [8] inv-less)
              apply (rule-tac [7] inv-g-1 [THEN aux2])
                apply (unfold zprime-def)
                apply (auto intro: d22set-g-1 d22set-le
                  aux1 aux2 aux3 aux4)
  done

lemma inv-d22set-d22set:
    p ∈ zprime ==> inv p ‘ d22set (p − 2) = d22set (p − 2)
  apply (rule endo-inj-surj)
    apply (rule d22set-fin)
   apply (erule-tac [2] inv-inj)
  apply auto
  apply (rule d22set-mem)
   apply (erule inv-g-1)
    apply (subgoal-tac [3] inv p xa < p − 1)
     apply (erule-tac [4] inv-less-p-minus-1)
      apply (auto intro: d22set-g-1 d22set-le aux4)
  done

lemma d22set-d22set-bij:
    p ∈ zprime ==> (d22set (p − 2), d22set (p − 2)) ∈ bijR (reciR p)
  apply (unfold reciR-def)
  apply (rule-tac s = (d22set (p − 2), inv p ‘ d22set (p − 2)) in subst)
   apply (simp add: inv-d22set-d22set)
  apply (rule inj-func-bijR)
    apply (rule-tac [3] d22set-fin)
   apply (erule-tac [2] inv-inj)
  apply auto
      apply (erule inv-is-inv)
       apply (erule-tac [5] inv-g-1)
        apply (erule-tac [7] inv-less-p-minus-1)
         apply (auto intro: d22set-g-1 d22set-le aux2 aux3 aux4)
  done

lemma reciP-bijP: p ∈ zprime ==> bijP (reciR p) (d22set (p − 2))
  apply (unfold reciR-def bijP-def)
  apply auto
  apply (rule d22set-mem)
   apply auto
```

```
  done

lemma reciP-uniq: p ∈ zprime ==> uniqP (reciR p)
  apply (unfold reciR-def uniqP-def)
  apply auto
   apply (rule zcong-zless-imp-eq)
       apply (tactic {* stac (thm zcong-cancel2 RS sym) 5 *})
         apply (rule-tac [7] zcong-trans)
          apply (tactic {* stac (thm zcong-sym) 8 *})
          apply (rule-tac [6] zless-zprime-imp-zrelprime)
            apply auto
  apply (rule zcong-zless-imp-eq)
      apply (tactic {* stac (thm zcong-cancel RS sym) 5 *})
        apply (rule-tac [7] zcong-trans)
         apply (tactic {* stac (thm zcong-sym) 8 *})
         apply (rule-tac [6] zless-zprime-imp-zrelprime)
           apply auto
  done

lemma reciP-sym: p ∈ zprime ==> symP (reciR p)
  apply (unfold reciR-def symP-def)
  apply (simp add: zmult-commute)
  apply auto
  done

lemma bijER-d22set: p ∈ zprime ==> d22set (p - 2) ∈ bijER (reciR p)
  apply (rule bijR-bijER)
     apply (erule d22set-d22set-bij)
    apply (erule reciP-bijP)
   apply (erule reciP-uniq)
  apply (erule reciP-sym)
  done
```

11.2 Wilson

```
lemma bijER-zcong-prod-1:
    p ∈ zprime ==> A ∈ bijER (reciR p) ==> [ssetprod A = 1] (mod p)
  apply (unfold reciR-def)
  apply (erule bijER.induct)
    apply (subgoal-tac [2] a = 1 ∨ a = p - 1)
     apply (rule-tac [3] zcong-square-zless)
        apply auto
  apply (subst ssetprod-insert)
    prefer 3
    apply (subst ssetprod-insert)
```

```
    apply (auto simp add: fin-bijER)
  apply (subgoal-tac zcong ((a * b) * ssetprod A) (1 * 1) p)
   apply (simp add: zmult-assoc)
  apply (rule zcong-zmult)
   apply auto
  done

theorem Wilson-Bij: p ∈ zprime ==> [zfact (p - 1) = -1] (mod p)
  apply (subgoal-tac zcong ((p - 1) * zfact (p - 2)) (-1 * 1) p)
   apply (rule-tac [2] zcong-zmult)
    apply (simp add: zprime-def)
    apply (subst zfact.simps)
    apply (rule-tac t = p - 1 - 1 and s = p - 2 in subst)
     apply auto
   apply (simp add: zcong-def)
  apply (subst d22set-prod-zfact [symmetric])
  apply (rule bijER-zcong-prod-1)
   apply (rule-tac [2] bijER-d22set)
   apply auto
  done

end
```

12 Facts about rings and fields

```
theory RingLib = Main:
```

12.1 Misc theorems for rings and ordered rings

```
lemma abs-nonneg [simp]: 0 ≤ (x::'a::ordered-ring) ⟹ abs x = x
  by (auto simp add: abs-if)

lemma abs-nonpos [simp]: (x::'a::ordered-field) <= 0 ==> abs x = -x
  apply (simp only: abs-if)
  apply (subgoal-tac x < 0 | x = 0)
  apply auto
  done

lemma nonneg-times-nonneg: 0 <= (x::'a::ordered-ring) ==> 0 <= y ==>
    0 <= x * y
proof -
  assume 0 <= (x::'a) and 0 <= (y::'a)
  then have 0 * 0 <= x * y
```

```
    by (rule mult-mono, auto)
  thus ?thesis
    by simp
qed

lemma abs-pos [simp]: 0 < (x::'a::ordered-ring) ⟹ abs x = x
  by (auto simp add: abs-if)

lemma abs-neg [simp]: (x::'a::ordered-field) < 0 ==> abs x = −x
  apply (simp only: abs-if)
  apply (subgoal-tac x < 0 | x = 0)
  apply auto
  done

lemma pos-plus-pos: 0 < (x::'a::ordered-semiring) ⟹ 0 < y ⟹ 0 < x + y
apply (subgoal-tac 0 + 0 < x + y)
apply simp
apply (rule add-strict-mono)
apply (assumption)+
done

lemma nonneg-times-nonneg: 0 ≤ (x::'a::ordered-semiring) ⟹ 0 ≤ y ⟹
    0 ≤ x * y
apply (subgoal-tac 0 * 0 ≤ x * y)
apply simp
apply (subgoal-tac 0 * 0 ≤ 0 * y)
apply (subgoal-tac 0 * y ≤ x * y)
apply force
apply (erule mult-right-mono, assumption)
apply (erule mult-left-mono)
apply simp
done

lemma nonneg-plus-nonneg: 0 ≤ (x::'a::ordered-semiring) ⟹ 0 ≤ y ⟹
    0 ≤ x + y
  apply (subgoal-tac 0 + 0 ≤ x + y)
  apply simp
  apply (rule add-mono, assumption+)
done

theorem ring-less-mult-cancel-left: 0 < a ==>
  (a * b < a * c) = (b < (c::'a::ordered-ring))
apply(auto)
apply(case-tac b < c)
apply(auto simp add: linorder-not-less)
```

apply(*subgoal-tac* $a * c <= a * b$)
apply(*subgoal-tac* $\sim a * c <= a * b$)
apply(*simp*)
apply(*subst linorder-not-le*)
apply(*simp*)
apply(*simp add*: *mult-left-mono order-less-le*)
by(*auto simp add*: *mult-strict-left-mono*)

lemma *abs-times-pos*: $(0::'a::ordered\text{-}ring) <= x ==>$
  $(abs\ y) * x = abs\ (y * x)$
 apply (*subst abs-mult*)
 apply *simp*
done

lemma *abs-div*: $y \sim= 0 ==> abs\ (x::'a::ordered\text{-}field)\ /\ abs\ y = abs(x\ /\ y)$
 apply (*subst nonzero-divide-eq-eq*)
 apply *simp*
 apply (*subst abs-mult* [*THEN sym*])
 apply (*subst times-divide-eq-left*)
 apply (*subst times-divide-eq-right* [*THEN sym*])
 apply *simp*
done

lemma *abs-div-pos*: $(0::'a::ordered\text{-}field) < y ==> abs\ x\ /\ y = abs\ (x\ /\ y)$
 apply (*subst abs-div* [*THEN sym*])
 apply *simp*
 apply (*subst abs-nonneg*)
 apply *simp-all*
done

lemma *abs-diff*: $abs\ ((a::'a::ordered\text{-}ring) - b) = abs(b - a)$
 apply (*subst abs-minus-cancel* [*THEN sym*])
 apply *simp*
done

lemma *add-frac-eq*: $(y::'a::field) \sim= 0 ==> z \sim= 0 ==>$
  $x\ /\ y + w\ /\ z = (x * z + w * y)\ /\ (y * z)$
 apply (*subgoal-tac* $x\ /\ y = (x * z)\ /\ (y * z)$)
 apply (*erule ssubst*)
 apply (*subgoal-tac* $w\ /\ z = (w * y)\ /\ (y * z)$)
 apply (*erule ssubst*)
 apply (*rule add-divide-distrib* [*THEN sym*])
 apply (*subst mult-commute*)
 apply (*erule nonzero-mult-divide-cancel-right* [*THEN sym*])
 apply *assumption*

apply (*erule nonzero-mult-divide-cancel-right* [*THEN sym*])
apply *assumption*
done

lemma *diff-frac-eq*: (*y*::'*a*::*field*) ~= *0* ==> *z* ~= *0* ==>
x / *y* − *w* / *z* = (*x* ∗ *z* − *w* ∗ *y*) / (*y* ∗ *z*)
apply (*subgoal-tac x* / *y* = (*x* ∗ *z*) / (*y* ∗ *z*))
apply (*erule ssubst*)
apply (*subgoal-tac w* / *z* = (*w* ∗ *y*) / (*y* ∗ *z*))
apply (*erule ssubst*)
apply (*rule diff-divide-distrib* [*THEN sym*])
apply (*subst mult-commute*)
apply (*erule nonzero-mult-divide-cancel-right* [*THEN sym*])
apply *assumption*
apply (*erule nonzero-mult-divide-cancel-right* [*THEN sym*])
apply *assumption*
done

lemma *frac-eq-eq*: (*y*::'*a*::*field*) ~= *0* ==> *z* ~= *0* ==>
(*x* / *y* = *w* / *z*) = (*x* ∗ *z* = *w* ∗ *y*)
apply (*subst nonzero-eq-divide-eq*)
apply *assumption*
apply (*subst times-divide-eq-left*)
apply (*erule nonzero-divide-eq-eq*)
done

lemma *abs-triangle-ineq2*: *abs* (*a*::'*a*::*ordered-ring*) − *abs b* <= *abs* (*a* − *b*)
apply (*simp add*: *compare-rls*)
apply (*subgoal-tac abs a* = *abs* (*a* − *b* + *b*))
apply (*erule ssubst*)
apply (*rule abs-triangle-ineq*)
apply *simp*
done

lemma *abs-triangle-ineq3*:
abs(*abs* (*a*::'*a*::*ordered-ring*) − *abs b*) <= *abs* (*a* − *b*)
apply (*subst abs-le-iff*)
apply *auto*
apply (*rule abs-triangle-ineq2*)
apply (*subst abs-minus-cancel* [*THEN sym*])
apply *simp*
apply (*rule abs-triangle-ineq2*)
done

lemma *abs-triangle-ineq4*: *abs* ((*a*::'*a*::*ordered-ring*) − *b*) <= *abs a* + *abs b*

proof −
have *abs*(*a* − *b*) = *abs*(*a* + − *b*)
by (*subst diff-minus*, *rule refl*)
also have ... <= *abs a* + *abs* (− *b*)
by (*rule abs-triangle-ineq*)
finally show *?thesis*
by *simp*
qed

lemma *pos-div-pos*: *0* < (*x*::′*a*::*ordered-field*) ==> *0* < *y* ==> *0* < *x* / *y*
apply (*subst pos-less-divide-eq*)
apply *assumption*
apply *simp*
done

lemma *setsum-le-cong2* [*rule-format*]: *finite B* ==> *A* <= *B* ==>
ALL x: *B* − *A*. *0* <= ((*f x*)::′*a*::*ordered-semiring*) ==>
setsum f A <= *setsum f B*
apply (*subgoal-tac setsum f B* = *setsum f A* + *setsum f* (*B* − *A*))
apply (*erule ssubst*)
apply (*subgoal-tac setsum f A* + *0* <= *setsum f A* + *setsum f* (*B* − *A*))
apply *simp*
apply (*rule add-left-mono*)
apply (*rule setsum-nonneg*)
apply (*rule finite-subset*)
prefer *2*
apply *assumption*
apply *force*
apply *assumption*
apply (*subst setsum-Un-disjoint* [*THEN sym*])
apply (*erule finite-subset*)
apply *assumption*
apply (*rule finite-subset*)
prefer *2*
apply *assumption*
apply *force*
apply *force*
apply (*rule setsum-cong*)
apply *force*
apply (*rule refl*)
done

lemma *order-trans2*: (*b*::′*a*::*order*) <= *c* ==> *a* <= *b* ==> *a* <= *c*

apply (*erule order-trans*)
apply *assumption*
done

lemma *power3-eq-cube*: (*x*::'*a*::*ringpower*) ^ *3* = *x* * *x* * *x*
apply (*subgoal-tac 3* = *Suc* (*Suc* (*Suc 0*)))
apply (*erule ssubst*)
apply (*simp add*: *power-Suc mult-ac*)
apply *simp*
done

end

13 Facts about finite sets, sums, and products

theory *FiniteLib* = *Main*:

syntax
-qsetsum :: *idt* ⇒ *bool* ⇒ '*a* ⇒ '*a* ((*3*∑ - | -./ -) [*0*,*0*,*10*] *10*)

syntax (*HTML* **output**)
-qsetsum :: *idt* ⇒ *bool* ⇒ '*a* ⇒ '*a* ((*3*∑ - | (-)./ -) [*0*,*0*,*10*] *10*)

translations ∑*x*|*P*. *t* => *setsum* (%*x*. *t*) {*x*. *P*}

print-translation {∗
let
fun setsum-tr' [*Abs*(*x*,*Tx*,*t*), *Const* (*Collect*,-) $ *Abs*(*y*,*Ty*,*P*)] =
(*if x*<>*y then raise Match*
else let val x' = *Syntax.mark-bound x*
val t' = *subst-bound*(*x'*,*t*)
val P' = *subst-bound*(*x'*,*P*)
in Syntax.const -qsetsum $ *Syntax.mark-bound x* $ *P'* $ *t' end*) *in* [(*setsum*, *setsum-tr'*)] *end* ∗}

syntax
-from-to-setsum :: *idt* ⇒ '*a* ⇒ '*a* ⇒ '*b* ⇒ '*b* ((∑ - = -..-./ -) [*0*,*0*,*0*,*10*] *10*)

syntax (*HTML* **output**)
-from-to-setsum :: *idt* ⇒ '*a* ⇒ '*a* ⇒ '*b* ⇒ '*b* ((*3*∑ - = -..-./ -) [*0*,*0*,*0*,*10*] *10*)

translations $\sum x{=}a..b.\ t == setsum\ (\%x.\ t)\ \{a..b\}$

lemma *finite-union-finite-subsets*: *finite* S ==> $ALL\ x{:}S.\ (finite\ x)$
==> *finite* $(Union\ S)$
by $(unfold\ Union\text{-}def,\ auto)$

lemma *setsum-cong2*: $[\![\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x]\!] \Longrightarrow setsum\ f\ A = setsum\ g\ A$
by $(rule\ setsum\text{-}cong,\ auto)$

lemma *setsum-const-times*: $setsum\ (\%x.\ (c{::}'a{::}semiring) * f\ x)\ A =$
$c * setsum\ f\ A$
apply $(case\text{-}tac\ finite\ A)$
apply $(induct\ set{:}\ Finites)$
apply $(simp\text{-}all\ add{:}\ ring\text{-}distrib)$
apply $(simp\ add{:}\ setsum\text{-}def)$
done

lemma *setsum-reindex'*: $[\![finite\ B;\ inj\text{-}on\ f\ B]\!] \Longrightarrow$
$setsum\ h\ (f\ `\ B) = setsum\ (\%x.\ h(f\ x))\ B$
apply $(frule\ setsum\text{-}reindex)$
apply *assumption*
apply $(erule\ ssubst)$
apply $(unfold\ o\text{-}def)$
apply $(rule\ refl)$
done

lemma *setsum-reindex-cong'*: $[\![finite\ A;\ inj\text{-}on\ f\ A;\ B = f\ `\ A;$
$g = (\%x.\ h\ (f\ x))]\!] \Longrightarrow setsum\ h\ B = setsum\ g\ A$
apply $(frule\ setsum\text{-}reindex\text{-}cong)$
apply *assumption*+
apply $(unfold\ o\text{-}def)$
apply *assumption*+
done

lemma *setsum-reindex-cong''*: $[\![finite\ A;\ inj\text{-}on\ f\ A;\ B = f\ `\ A;$
$ALL\ x{:}A.\ (g\ x = h\ (f\ x))]\!] \Longrightarrow setsum\ h\ B = setsum\ g\ A$
apply $(subgoal\text{-}tac\ setsum\ g\ A = setsum\ (h\ o\ f)\ A)$
apply $(subgoal\text{-}tac\ setsum\ h\ B = setsum\ (h\ o\ f)\ A)$
apply *simp*
apply $(rule\ setsum\text{-}reindex\text{-}cong)$
apply *assumption*+
apply $(rule\ refl)$
apply $(rule\ setsum\text{-}cong)$
apply $(simp\text{-}all\ add{:}\ o\text{-}def)$

done

lemma *setsum-of-nat'*: *of-nat* (*setsum f A*) = *setsum* (%*x*. *of-nat*(*f x*)) *A*
apply (*subst setsum-of-nat*)
apply (*simp add*: *o-def*)
done

lemma *setsum-of-int'*: *of-int* (*setsum f A*) = *setsum* (%*x*. *of-int*(*f x*)) *A*
apply (*subst setsum-of-int*)
apply (*simp add*: *o-def*)
done

lemma *interval-singleton* [*simp*]: {*a*::'*a*::*order*..*a*} = {*a*}
by (*auto simp add*: *atLeastAtMost-def atMost-def atLeast-def*)

lemma *interval-empty* [*simp*]: *b* < *a* ==> {*a*::'*a*::*order*..*b*} = {}
by (*auto simp add*: *atLeastAtMost-def atMost-def atLeast-def*)

lemma *interval-plus-one-nat*: (*n*::*nat*) <= *m* ==> {*n*..*m*+*1*} = {*n*..*m*} *Un* {*m*+*1*}
by *auto*

lemma *setsum-range-plus-one-nat*: (*n*::*nat*) <= *m* ==>
(∑ *i*=*n*..*m*+*1*. *f i*) = (∑ *i*=*n*..*m*. *f i*) + *f*(*m* + *1*)
apply (*subst interval-plus-one-nat*)
apply *assumption*
apply (*subst setsum-Un-disjoint*)
apply *auto*
done

lemma *abs-setsum*: *abs*(*setsum* (*f*::'*a*=>'*b*::*ordered-ring*) *A*) <=
setsum (%*x*. *abs* (*f x*)) *A*
apply (*case-tac finite A*)
apply (*induct set*: *Finites*)
apply *simp*
apply *simp*
apply (*rule order-trans*)
apply (*rule abs-triangle-ineq*)
apply *simp*
apply (*simp add*: *setsum-def*)
done

lemma *setsum-nonneg'*: *ALL x*:*A*. (*0*::'*a*::*ordered-ring*) <=
f x ==> *0* <= *setsum f A*
apply (*case-tac finite A*)
apply (*erule setsum-nonneg*)

apply *assumption*
apply (*simp add*: *setsum-def*)
done

lemma *setsum-nonpos*: *finite A* ==>
ALL x : *A*. *f x* <= (*0*::'*a*::*ordered-semiring*) ==>
setsum f A <= *0*
apply (*induct set*: *Finites*, *auto*)
apply (*subgoal-tac f x* + *setsum f F* <= *0* + *0*, *simp*)
apply (*blast intro*: *add-mono*)
done

lemma *setsum-nonpos'*: *ALL x* : *A*. *f x* <= (*0*::'*a*::*ordered-semiring*) ==>
setsum f A <= *0*
apply (*case-tac finite A*)
apply (*erule setsum-nonpos*)
apply *assumption*
apply (*simp add*: *setsum-def*)
done

lemma *setsum-le-cong* [*rule-format*]:
(*ALL x*:*A*. *f x* <= ((*g x*)::'*a*::*ordered-ring*)) -->
setsum f A <= *setsum g A*
apply (*case-tac finite A*)
apply (*induct set*: *Finites*)
apply *simp*
apply *auto*
apply (*erule add-mono*)
apply *assumption*
apply (*simp add*: *setsum-def*)
done

lemma *setsum-lt-cong*: *finite A* ==> *A* ~= {} ==> *ALL x*:*A*. (*f x* < ((*g x*)::'*a*::*ordered-semiring*))
==> *setsum f A* < *setsum g A*
apply (*induct set*: *Finites*)
apply (*force*)
apply (*simp add*: *setsum-insert*)
apply (*case-tac F* = {})
apply *force*
apply (*auto simp add*: *add-strict-mono*)
done

lemma *setsum-negf'*: (∑ *x*:*A*. − (*f*::'*a*=>'*b*::*ring*) *x*) = − *setsum f A*
apply (*case-tac finite A*)
apply (*erule setsum-negf*)

apply (*simp add*: *setsum-def*)
done

lemma *nat-interval-Suc*: {*1*..*Suc n*} = {*1*..*n*} *Un* {*Suc n*}
by *auto*

lemma *nat-setsum-Suc*: (∑ *i*=*1*..*Suc n*. *f i*) = (∑ *i*=*1*..*n*. *f i*) +
f (*Suc n*)
apply (*subst nat-interval-Suc*)
apply (*subst setsum-Un-disjoint*)
apply *auto*
done

lemma *interval-plus-one-nat'*: (*n*::*nat*) <= *m* + *1* ==>
{*n*..*m*+*1*} = {*n*..*m*} *Un* {*m*+*1*}
by *auto*

lemma *setsum-range-plus-one-nat'*: (*n*::*nat*) <= *m* + *1* ==>
(∑ *i*=*n*..*m*+*1*. *f i*) = (∑ *i*=*n*..*m*. *f i*) + *f*(*m* + *1*)
apply (*subst interval-plus-one-nat'*)
apply *assumption*
apply (*subst setsum-Un-disjoint*)
apply *auto*
done

end

14 Facts about integers and natural numbers

theory *NatIntLib* = *WilsonRuss*:

lemma [*simp*]: *of-nat*(*n*::*nat*) = *n*
apply (*induct-tac n*)
apply *auto*
done

14.1 Integer divisibility and powers

lemma *zpower-zdvd-prop1* [*rule-format*]: ((*0* < *n*) & (*p dvd y*)) −−>
p dvd ((*y*::*int*) ^ *n*)
by (*induct-tac n*, *auto simp add*: *zdvd-zmult zdvd-zmult2* [*of p y*])

lemma *zdvd-bounds*: *n dvd m ==> (m ≤ (0::int) | n ≤ m)*
proof −
  assume *n dvd m*
  then have ~*(0 < m & m < n)*
    apply (*insert zdvd-not-zless* [*of m n*])
    by (*rule contrapos-pn*, *auto*)
  then have (~*0 < m* | ~*m < n*) **by** *auto*
  then show *?thesis* **by** *auto*
qed

lemma *zprime-zdvd-zmult-better*: [| *p ∈ zprime*; *p dvd (m * n)* |] ==>
    (*p dvd m*) | (*p dvd n*)
  apply (*case-tac 0 ≤ m*)
  apply (*simp add*: *zprime-zdvd-zmult*)
  by (*insert zprime-zdvd-zmult* [*of* −*m p n*], *auto*)

lemma *zpower-zdvd-prop2* [*rule-format*]: *p ∈ zprime* −−> *p dvd* ((*y*::*int*) ^ *n*)
    −−> *0 < n* −−> *p dvd y*
  apply (*induct-tac n*, *auto*)
  apply (*frule zprime-zdvd-zmult-better*, *auto*)
done

lemma *stupid*: *(0 :: int) ≤ y ==> x ≤ x + y*
  by *arith*

lemma *div-prop1*: [| *0 < z*; (*x*::*int*) *< y * z* |] ==> *x div z < y*
proof −
  assume *0 < z*
  then have *(x div z) * z ≤ (x div z) * z + x mod z*
  apply (*rule-tac x = x div z * z* **in** *stupid*)
  by (*simp add*: *pos-mod-sign*)
  also have ... = *x*
    by (*auto simp add*: *zmod-zdiv-equality* [*THEN sym*] *zmult-ac*)
  also assume *x < y * z*
  finally show *?thesis*
    by (*auto simp add*: *prems mult-less-cancel-right*, *insert prems*, *arith*)
qed

lemma *div-prop2*: [| *0 < z*; (*x*::*int*) *< (y * z) + z* |] ==> *x div z ≤ y*
proof −
  assume *0 < z* **and** *x < (y * z) + z*
  then have *x < (y + 1) * z* **by** (*auto simp add*: *int-distrib*)
  then have *x div z < y + 1*
    by (*rule-tac y = y + 1* **in** *div-prop1*, *auto simp add*: *prems*)

```
  then show ?thesis by auto
qed

lemma zdiv-leq-prop: [| 0 < y |] ==> y * (x div y) ≤ (x::int)
proof−
  assume 0 < y
  from zmod-zdiv-equality have x = y * (x div y) + x mod y by auto
  moreover have 0 ≤ x mod y
    by (auto simp add: prems pos-mod-sign)
  ultimately show ?thesis
    by arith
qed

lemma zdiv-gr-zero: [| 0 < a; 0 < b; a dvd b |] ==> (0::int) < (b div a)
  by (auto simp add: dvd-def zero-less-mult-iff)

lemma zdiv-zdiv-prop: [| 0 < a; 0 < b; b dvd a |] ==> a div (a div b)
    = (b::int)
proof −
  assume 0 < a and 0 < b and b dvd a
  then have p: 0 < a div b by (auto simp add: zdiv-gr-zero)
  have a = a by auto
  with prems have a = b * (a div b) by (auto simp add: dvd-def)
  with prems have a div (a div b) = (b * (a div b)) div (a div b) by auto
  also from prems p have ... = b by auto
  finally show ?thesis .
qed

lemma x-div-pa-prop: [| p:zprime; x dvd (p * a) |] ==>
    (p dvd x) | (x dvd a)
  apply (simp only: dvd-def, drule-tac Q = (EX k. x = p * k) | (EX
      k. a = x * k) in exE)
  defer 1 apply (force)
proof −
  fix y
  assume p * a = x * y and p:zprime
  then have p dvd (x * y)
    apply (auto simp add: dvd-def)
    apply (rule-tac x = a in exI)
    apply (auto)
  done
  then have (p dvd x) | (p dvd y)
    by (insert prems, auto simp add: zprime-zdvd-zmult-better)
  moreover have p dvd y ==> x dvd a
  proof−
```

assume *p dvd y*
then have *x * y = x * (y div p) * p*
by (*auto simp add*: *dvd-def*)
also have *x * y = a * p* **by** (*insert prems*, *auto simp add*: *zmult-ac*)
finally have *a * p = x * (y div p) * p*.
then have *a = x * (y div p)*
by (*insert prems*, *auto simp add*: *zprime-def*)
thus *x dvd a* **by** (*auto simp add*: *dvd-def*)
qed
ultimately have (*p dvd x*) | (*x dvd a*) **by** (*auto*)
thus (*EX k*. *x = p * k*) | (*EX k*. *a = x * k*) **by** (*auto simp add*: *dvd-def*)
qed

14.2 Divisibility and powers

lemma *zpower-gr-0*: *0 < (x::int) ==> 0 < x ^ k*
by (*induct-tac k*, *auto simp add*: *mult-pos*)

lemma *zpower-minus-one*: [| *0 < k*; *0 < p* |] ==> ((*p::int*) ^ *k*) *div p* =
p ^ (k − 1)
proof−
assume *0 < k* **and** *0 < p*
then have (*p^k*) = (*p ^ Suc*(*k − 1*))
by *auto*
also have ...= *p^(k − 1) * p*
by (*auto simp add*: *power-Suc*)
finally have *p^k = p^(k − 1) * p*.
then have *p^k div p = (p^(k − 1) * p) div p*
by (*auto*)
thus *?thesis* **by** (*insert prems*, *auto*)
qed

lemma *zpower-zmult*: [| *0 < k*; *0 < p* |] ==> (*p::int*) ^ *k* =
*p ^ (k − 1) * p*
proof −
assume *0 < k* **and** *0 < p*
then have (*p^k*) = (*p ^ Suc*(*k − 1*))
by *auto*
thus *p ^ k = p^(k − 1) * p*
by (*auto simp add*: *power-Suc*)
qed

lemma *x-dvd-pk-prop* [*rule-format*]: *p:zprime −−> 0 <= x −−> x dvd* (*p^k*)
−−> (*x = 1*) | (*p dvd x*)
apply (*induct-tac k*)

apply (*clarsimp*)
apply (*frule zdvd-bounds*)
apply (*subgoal-tac x = 0 | x = 1*)
apply (*force*, *arith*)
apply (*clarsimp*)
apply (*frule x-div-pa-prop*, *auto*)
done

14.3 Properties of gcd

lemma *gcd-ne-zero*: (*m*::*nat*) ~= *0* ==> *gcd*(*m*,*n*) ~= *0*
by (*auto simp add*: *gcd-zero*)

lemma *zgcd-gr-zero1*: *0* < *a* ==> *0* < *zgcd*(*a*, *b*)
apply (*auto simp add*: *zgcd-def*)
apply (*subgoal-tac abs a = a*)
apply (*rule ssubst*)
apply (*auto*)
apply (*subgoal-tac nat a ~= 0*)
apply (*drule gcd-ne-zero*)
apply (*auto simp add*: *zabs-def*)
done

lemma *zgcd-gr-zero2*: *0* < *a* ==> *0* < *zgcd*(*b*, *a*)
by (*auto simp add*: *zgcd-gr-zero1 zgcd-ac*)

lemma *zdvd-zgcd-prop*: [| *0* < *d*; *d dvd m*; *d dvd n* |] ==> *zgcd*(*m*,*n*) =
d * *zgcd*(*m div d*, *n div d*)
proof −
assume *0* < *d* **and** *d dvd m* **and** *d dvd n*
then have *d* * *zgcd* (*m div d*, *n div d*) = *zgcd*(*d* * (*m div d*), *d* * (*n div d*))
apply (*rule-tac m = m div d* **and** *n = n div d* **in** *zgcd-zmult-distrib2*)
apply (*insert prems*, *auto*)
done
also have ... = *zgcd*(*m*,*n*)
by (*insert prems*, *auto simp add*: *dvd-def zmult-ac*)
finally show *?thesis* **by** *auto*
qed

lemma *zgcd-equiv*: *0* < *d* ==> (*zgcd*(*k*, *n*) = *d*) =
(*d dvd k* & *d dvd n* & *zgcd*(*k div d*, *n div d*) = *1*)
proof
assume *0* < *d* **and** *zgcd*(*k*, *n*) = *d*
then show *d dvd k* & *d dvd n* & *zgcd*(*k div d*, *n div d*) = *1*
apply (*subgoal-tac 0* < *d*)

```
    apply (drule-tac d = d and m = k and n = n in zdvd-zgcd-prop)
    apply (auto)
  done
  next
  assume 0 < d and d dvd k & d dvd n & zgcd (k div d, n div d) = 1
  then show (d dvd k & d dvd n & zgcd(k div d, n div d) = 1) ==>
      zgcd(k, n) = d

    apply (frule-tac d = d and m = k and n = n in zdvd-zgcd-prop)
    apply (auto)
  done
qed

lemma gcd-prime-power-iff-zdvd-prop: [| 0 < k; p:zprime |] ==>
  (zgcd(x, p^k) ~= 1) = (p dvd x)
proof
  assume p:zprime and zgcd(x, p ^ k) ~= 1
  then have zgcd(x, p ^ k) dvd p ^ k by auto
  then have (zgcd(x, p ^ k) = 1) | (p dvd zgcd(x, p ^ k))
    apply (insert prems)
    apply (rule-tac x = zgcd(x, p ^ k) in x-dvd-pk-prop)
    apply (auto)
    proof -
      have 0 < p ^ k by (insert prems, auto simp add: zpower-gr-0 zprime-def)
      then have 0 < zgcd (x, p ^ k) by (rule zgcd-gr-zero2)
      thus 0 <= zgcd (x, p ^ k) by auto
    qed
  then have p dvd zgcd(x, p ^ k) by (insert prems, auto)
  moreover have zgcd(x, p ^ k) dvd x by auto
  ultimately show p dvd x by (rule zdvd-trans)
next
  assume 0 < k and p:zprime and p dvd x
  moreover have p dvd (p ^ k)
    by (rule zpower-zdvd-prop1, auto simp add: prems)
  ultimately have p dvd zgcd(x, p ^ k)
    by (auto simp add: zgcd-greatest-iff)
  thus zgcd(x, p ^ k) ~= 1
    apply (insert prems, auto simp add: zprime-def)
    apply (drule-tac n = p and m = 1 in zdvd-bounds, auto)
  done
qed
```

14.4 Properties of integers and congruence

```
lemma zcong-eq-zdvd-prop: [x = 0](mod p) = (p dvd x)
```

by (*auto simp add*: *zcong-def*)

lemma *zcong-id*: [*m* = *0*] (*mod m*)
by (*auto simp add*: *zcong-def zdvd-0-right*)

lemma *zcong-shift*: [*a* = *b*] (*mod m*) ==> [*a* + *c* = *b* + *c*] (*mod m*)
by (*auto simp add*: *zcong-refl zcong-zadd*)

lemma *zcong-zpower*: [*x* = *y*](*mod m*) ==> [*x*^*z* = *y*^*z*](*mod m*)
by (*induct-tac z*, *auto simp add*: *zcong-zmult*)

lemma *zcong-eq-trans*: [| [*a* = *b*](*mod m*); *b* = *c*; [*c* = *d*](*mod m*) |] ==>
[*a* = *d*](*mod m*)
by (*auto*, *rule-tac b* = *c* **in** *zcong-trans*)

lemma *aux1*: *a* − *b* = (*c*::*int*) ==> *a* = *c* + *b*
by *auto*

lemma *zcong-zmult-prop1*: [*a* = *b*](*mod m*) ==> ([*c* = *a* ∗ *d*](*mod m*) =
[*c* = *b* ∗ *d*] (*mod m*))
apply (*auto simp add*: *zcong-def dvd-def*)
apply (*rule-tac x* = *ka* + *k* ∗ *d* **in** *exI*)
apply (*drule aux1*)+
apply (*auto simp add*: *int-distrib*)
apply (*rule-tac x* = *ka* − *k* ∗ *d* **in** *exI*)
apply (*drule aux1*)+
apply (*auto simp add*: *int-distrib*)
done

lemma *zcong-zmult-prop2*: [*a* = *b*](*mod m*) ==>
([*c* = *d* ∗ *a*](*mod m*) = [*c* = *d* ∗ *b*] (*mod m*))
by (*auto simp add*: *zmult-ac zcong-zmult-prop1*)

lemma *zcong-zmult-prop3*: [|*p* ∈ *zprime*; ~[*x* = *0*] (*mod p*);
~[*y* = *0*] (*mod p*) |] ==> ~[*x* ∗ *y* = *0*] (*mod p*)
apply (*auto simp add*: *zcong-def*)
apply (*drule zprime-zdvd-zmult-better*, *auto*)
done

lemma *zcong-less-eq*: [| *0* < *x*; *0* < *y*; *0* < *m*; [*x* = *y*] (*mod m*);
x < *m*; *y* < *m* |] ==> *x* = *y*
apply (*simp add*: *zcong-zmod-eq*)
apply (*subgoal-tac* (*x mod m*) = *x*)
apply (*subgoal-tac* (*y mod m*) = *y*)
apply *simp*

apply (*rule-tac* [*1−2*] *mod-pos-pos-trivial*)
by *auto*

lemma *zcong-neg-1-impl-ne-1*: [| *2* < *p*; [*x* = *−1*] (*mod p*) |] ==>
  ~([*x* = *1*] (*mod p*))
proof
  assume *2* < *p* **and** [*x* = *1*] (*mod p*) **and** [*x* = *−1*] (*mod p*)
  then have [*1* = *−1*] (*mod p*)
    apply (*auto simp add*: *zcong-sym*)
    apply (*drule zcong-trans*, *auto*)
  done
  then have [*1* + *1* = *−1* + *1*] (*mod p*)
    by (*simp only*: *zcong-shift*)
  then have [*2* = *0*] (*mod p*)
    by *auto*
  then have *p dvd 2*
    by (*auto simp add*: *dvd-def zcong-def*)
  with *prems* **show** *False*
    by (*auto simp add*: *zdvd-not-zless*)
qed

lemma *zcong-zero-equiv-div*: [*a* = *0*] (*mod m*) = (*m dvd a*)
  by (*auto simp add*: *zcong-def*)

lemma *zcong-zprime-prod-zero*: [| *p* ∈ *zprime*; *0* < *a* |] ==>
  [*a* * *b* = *0*] (*mod p*) ==> [*a* = *0*] (*mod p*) | [*b* = *0*] (*mod p*)
  by (*auto simp add*: *zcong-zero-equiv-div zprime-zdvd-zmult*)

lemma *zcong-zprime-prod-zero-contra*: [| *p* ∈ *zprime*; *0* < *a* |] ==>
  ~[*a* = *0*](*mod p*) & ~[*b* = *0*](*mod p*) ==> ~[*a* * *b* = *0*] (*mod p*)
  apply *auto*
  apply (*frule-tac a* = *a* **and** *b* = *b* **and** *p* = *p* **in** *zcong-zprime-prod-zero*)
by *auto*

lemma *zcong-not-zero*: [| *0* < *x*; *x* < *m* |] ==> ~[*x* = *0*] (*mod m*)
  by (*auto simp add*: *zcong-zero-equiv-div zdvd-not-zless*)

lemma *zcong-zero*: [| *0* ≤ *x*; *x* < *m*; [*x* = *0*](*mod m*) |] ==> *x* = *0*
  apply (*drule order-le-imp-less-or-eq*, *auto*)
by (*frule-tac m* = *m* **in** *zcong-not-zero*, *auto*)

lemma *all-relprime-prod-relprime*: [| *finite A*; ∀ *x* ∈ *A*.
    (*zgcd*(*x*,*y*) = *1*) |] ==> *zgcd* (*setprod id A*,*y*) = *1*
  by (*induct set*: *Finites*, *auto simp add*: *zgcd-zgcd-zmult*)

lemma *zmod-zmult-zmod*: $0 < m \Longrightarrow (x::int) \bmod m = (x \bmod (m*n)) \bmod m$
proof−
assume $0 < m$
moreover have m *dvd* $(m*n)$ **by** *auto*
ultimately show *?thesis* **by** (*auto simp add*: *zmod-zdvd-zmod*)
qed

lemma *zmod-zmult-zmod2*: $0 < n \Longrightarrow (x::int) \bmod n = (x \bmod (m*n)) \bmod n$
proof−
assume $0 < n$
moreover have n *dvd* $(m*n)$ **by** *auto*
ultimately show *?thesis* **by** (*auto simp add*: *zmod-zdvd-zmod*)
qed

lemma *zgcd-eq2*: $zgcd(m, n) = zgcd(m, n \bmod m)$
proof−
have $zgcd(m, n) = zgcd(n, m)$ **by** (*auto simp add*: *zgcd-commute*)
also have ... $= zgcd(m, n \bmod m)$ **by** (*insert zgcd-eq* [*of n m*], *auto*)
finally show *?thesis* **.**
qed

lemma *zcong-m-scalar-prop*: $[a = b] \ (mod\ m) \Longrightarrow [a + (m*c) = b] \ (mod\ m)$
apply (*auto simp add*: *zcong-def*)
apply (*auto simp add*: *dvd-def zmult-ac*)
apply (*rule-tac* $x = k + c$ **in** *exI*)
apply (*auto simp add*: *int-distrib*)
done

lemma *setsum-same-function-zcong*:
$[|$ *finite* $S; \forall x \in S. [f\ x = g\ x](mod\ m)\ |]$
$\Longrightarrow [setsum\ f\ S = setsum\ g\ S] \ (mod\ m)$
by (*induct set*: *Finites*, *auto simp add*: *zcong-zadd*)

lemma *setprod-same-function-zcong*:
$[|$ *finite* $S; \forall x \in S. [f\ x = g\ x](mod\ m)\ |]$
$\Longrightarrow [setprod\ f\ S = setprod\ g\ S] \ (mod\ m)$
by (*induct set*: *Finites*, *auto simp add*: *zcong-zmult*)

lemma *le-imp-zpower-zdvd* [*rule-format*]: *a <= b --> (p::int) ^a dvd p ^b*
apply (*induct b*, *auto simp add*: *dvd-def*)
apply (*rule-tac x = 1* **in** *exI*, *auto*)
apply (*subgoal-tac a = Suc n*, *auto*)
done

lemma *ge-1-imp-zpower-ge-1*: *1 <= (p::int) ==> 1 <= p ^ k*
apply (*induct k*, *auto*)
apply (*subgoal-tac 0 < p ^ n & 0 < p & 0 < p * p ^n*, *auto*)
apply (*auto simp add*: *zero-less-mult-iff*)
done

lemma *ge-0-zdvd-1*: [| (*a::int*) *dvd 1*; *0 <= a* |] ==> *a = 1*
apply (*insert zdvd-imp-le* [*of a 1*], *auto*)
apply (*case-tac a = 0*, *auto*)
done

lemma *ne-0-zdvd-prop*: [| *a ~= 0*; (*a::int*) * *b dvd a * c* |] ==> *b dvd c*
by (*auto simp add*: *dvd-def*)

lemma *aux* [*rule-format*]:
*p:zprime --> p ~= 0 --> p ^ k dvd a * b --> ~ p dvd b --> p ^ k dvd a*
apply (*induct k*)
apply (*auto simp add*: *dvd-def*)
proof−
fix *k* **fix** *ka*
assume *ka * b = p * k* **and** *ALL k. b ~= p * k*
then have *p dvd* (*ka * b*) **and** *~ p dvd b* **by** (*auto simp add*: *dvd-def*)
moreover assume *p:zprime*
moreover note *zprime-zdvd-zmult-better*
ultimately have *p dvd ka* **by** *auto*
then show *EX k. ka = p * k* **by** (*auto simp add*: *dvd-def*)
qed

lemma *zprime-zdvd-zpower*: [| *p : zprime*; *p ^ k dvd a * b*; *~ p dvd b* |] ==> *p ^ k dvd a*

apply (*subgoal-tac p ~= 0*)
apply (*insert aux* [*of p k a b*])
apply (*auto simp add*: *zprime-def*)
done

14.5 Imported from later files

lemma *prime-ge-2*: *p*:*prime* ==> *2*<=*p*
apply (*auto simp add*: *prime-def*)
done

lemma *prime-pos*: *p* :*prime* ==> *0* < *p*
apply (*subgoal-tac 2* <= *p*)
apply *simp*
apply (*rule prime-ge-2*)
by *auto*

lemma *zero-not-prime*: *0*~:*prime*
apply (*auto simp add*: *prime-def*)
done

lemma *one-not-prime*: (*Suc 0*)~:*prime*
apply (*auto simp add*: *prime-def*)
done

lemma *prime-dvd-prime-eq*: *a*:*prime* ==> *b*:*prime* ==> *a dvd b* ==> *a*=*b*
apply (*auto simp add*: *prime-def*)
done

lemma *prime-dvd-power* [*rule-format*]: *p*:*prime* ==> *p dvd m*^*n* --> *p dvd m*
apply (*induct-tac n*)
apply *auto*
apply (*frule prime-dvd-mult*)
by *auto*

14.6 Finite sets of nats and integers

lemma *finite-subset-AtMost-nat*: *A* <= {..(*x*::*nat*)} ==> *finite A*
apply (*rule finite-subset*)
apply *assumption*
apply *auto*
done

lemma *finite-subset-GreaterThan0AtMost-int*: *A* <= {)*0*..(*x*::*int*)} ==> *finite A*
apply (*erule finite-subset*)

apply *simp*
done

lemma *finite-subset-GreaterThan0AtMost-nat*: $A <= \{)0..(x::nat)\} ==> finite\ A$
apply (*erule finite-subset*)
apply *simp*
done

lemma *int-card-eq-setsum*: $finite\ A ==> int\ (card\ A) = setsum\ (\%x.1)\ A$
apply (*subst card-eq-setsum*)
apply *assumption*
apply (*subst int-setsum*)
apply (*unfold o-def*)
apply *simp*
done

14.7 Stuff from later files

lemma *int-dvd-times-div-cancel*:
$(y::int) \sim= 0 \Longrightarrow y\ dvd\ x \Longrightarrow y * (x\ div\ y) = x$
by (*auto simp add*: *dvd-def*)

lemma *int-dvd-times-div-cancel2*: $[|\ 0 < (y::int);\ y\ dvd\ x\ |] ==>$
$y * (x\ div\ y) = x$
by (*auto simp add*: *dvd-def*)

lemma *int-inj-div*: $0 < (n::int) \Longrightarrow x\ dvd\ n \Longrightarrow y\ dvd\ n \Longrightarrow$
$n\ div\ x = n\ div\ y \Longrightarrow x = y$
apply (*subgoal-tac* $x * (n\ div\ x) = y * (n\ div\ y)$)
apply *simp*
apply *clarsimp*
apply (*subgoal-tac* $x * (n\ div\ x) = 0$)
apply (*subgoal-tac* $x * (n\ div\ x) = n$)
apply *force*
apply (*rule int-dvd-times-div-cancel*)
apply *force*
apply *force*
apply *simp*
apply (*subst int-dvd-times-div-cancel*)
apply (*force*, *assumption*)
apply (*subst int-dvd-times-div-cancel*)
apply *auto*
done

lemma *int-dvd-div-eq*: $x \sim= 0 \Longrightarrow (x::int)\ dvd\ y \Longrightarrow$

```
  (y div x = z) = (y = x * z)
 apply auto
 apply (erule int-dvd-times-div-cancel [THEN sym], assumption)
 done

lemma int-div-div: n ~= 0 ==> x dvd (n::int) ==> n div (n div x) = x
 apply (subst int-dvd-div-eq)
 apply (rule notI)
 apply (subgoal-tac x * (n div x) = n)
 apply simp
 apply (rule int-dvd-times-div-cancel)
 apply force
 apply assumption
 apply (auto simp add: dvd-def)
done

lemma int-pos-mult-le: 0 <= y ==> (0::int) < x ==> y <= x * y
 apply (subgoal-tac 1 * y <= x * y)
 apply simp
 apply (rule mult-right-mono)
 apply auto
done

lemma int-nonneg-div-nonneg: (0::int) <= x ==> 0 <= y ==> 0 <= y div x
 apply (case-tac x = 0)
 apply simp
 apply (subgoal-tac 0 = 0 div x)
 apply (erule ssubst)
 apply (rule zdiv-mono1)
 apply auto
done

lemma zdvd-leq: 0 <= n ==> n div x <= (n::int)
 apply (case-tac 0 < x)
 apply (subgoal-tac n div x <= n div 1)
 apply simp
 apply (rule zdiv-mono2)
 apply auto
 apply (case-tac x = 0)
 apply simp
 apply (subgoal-tac x < 0)
 apply (frule div-nonneg-neg-le0)
 apply assumption
 apply (erule order-trans)
 apply assumption
```

apply *force*
done

lemma *finite-nat-dvd-set*: $0 < (n::nat) \Longrightarrow$ *finite* $\{x.\ x\ dvd\ n\}$
apply (*rule finite-subset-AtMost-nat*)
apply *auto*
apply (*erule dvd-imp-le*)
apply *assumption*
done

lemma *finite-int-dvd-set*: $0 < n$ ==> *finite* $\{(d::int).\ 0 < d\ \&\ d\ dvd\ n\}$
apply (*rule finite-subset-GreaterThan0AtMost-int*)
apply *auto*
apply (*erule zdvd-imp-le*)
apply *assumption*
done

lemma *image-int-dvd-set*: $0 < n \Longrightarrow$
$int\ `\ \{x.\ x\ dvd\ n\} = \{x.\ 0 < x\ \&\ x\ dvd\ (int\ n)\}$
apply (*unfold image-def*)
apply *auto*
apply (*subgoal-tac xa* $\sim=$ *0*)
apply *force*
apply (*force simp add*: *dvd-def*)
apply (*subst zdvd-int* [*THEN sym*])
apply *assumption*
apply (*rule-tac x* = *nat x* **in** *exI*)
apply *simp*
apply (*simp add*: *dvd-int-iff abs-if*)
done

lemma *le-int-eq-nat-le*: $(x \leq int\ y) = (nat\ x \leq y)$
apply *auto*
apply (*subgoal-tac nat x* <= *nat* (*int y*))
apply *simp*
apply (*subst nat-le-eq-zle*)
apply *simp*
apply *assumption*
apply (*case-tac x* < *0*)
apply *simp*
apply (*subgoal-tac int* (*nat x*) <= *int y*)
apply *simp*
apply (*subst zle-int*)
apply *assumption*
done

lemma *image-int-GreaterThanAtMost*: *int ‘ {}x..y} =*
{}int x..int y}
apply (*unfold image-def*)
apply *auto*
apply (*rule-tac x = nat xa* **in** *bexI*)
apply *auto*
apply (*subst zless-nat-eq-int-zless*)
apply *assumption*
apply (*subst le-int-eq-nat-le* [*THEN sym*])
apply *assumption*
done

lemma *int-div*: *int*(*x div y*) = ((*int x*) *div* (*int y*))
proof −
have *x* = (*x div y*) ∗ *y* + *x mod y*
by (*rule mod-div-equality* [*THEN sym*])
then have *int x* = *int* (. . .) **by** *simp*
also have . . . = *int* (*x div y*) ∗ *int y* + *int* (*x mod y*)
by (*simp add*: *zmult-int zadd-int*)
finally have *int x* = *int* (*x div y*) ∗ *int y* + *int* (*x mod y*)**.**
then have *int x div int y* = (. . .) *div int y***by** *simp*
also have . . . = *int* (*x div y*)
apply (*subst zdiv-zadd1-eq*)
apply *auto*
apply (*rule div-pos-pos-trivial*)
apply *force*
apply *force*
done
finally show *?thesis* **by** (*rule sym*)
qed

lemma *nat-dvd-mult-div*: (*y*::*nat*) ~= *0* ⟹ *y dvd x* ⟹ *y* ∗ (*x div y*) = *x*
by (*auto simp add*: *dvd-def*)

lemma *nat-dvd-div-eq*: *x* ~= *0* ⟹ (*x*::*nat*) *dvd y* ⟹
(*y div x* = *z*) = (*y* = *x* ∗ *z*)
apply *auto*
apply (*rule nat-dvd-mult-div* [*THEN sym*])
apply *auto*
done

lemma *nat-div-div*: *n* ~= *0* ⟹ *x dvd* (*n*::*nat*) ⟹ *n div* (*n div x*) = *x*
apply (*subst nat-dvd-div-eq*)
apply (*rule notI*)

apply (*subgoal-tac* $x * (n\ div\ x) = n$)
apply *simp*
apply (*rule nat-dvd-mult-div*)
apply (*rule notI*)
apply *force*
apply *assumption*
apply (*auto simp add*: *dvd-def*)
done

lemma *nat-pos-div-dvd-gr-0*: $0 < (n::nat) \Longrightarrow x\ dvd\ n \Longrightarrow 0 < n\ div\ x$
apply (*subgoal-tac* $0 \sim= n\ div\ x$)
apply *force*
apply (*rule notI*)
apply (*subgoal-tac* $n = x * (n\ div\ x)$)
apply *force*
apply (*rule nat-dvd-mult-div* [*THEN sym*])
apply (*rule notI*)
apply *simp*
apply *assumption*
done

lemma *nat-pos-dvd-pos*: $[\![(x::nat)\ dvd\ n;\ 0 < n]\!] \Longrightarrow 0 < x$
apply (*subgoal-tac* $x \sim= 0$)
apply *force*
apply (*rule notI*)
apply *force*
done

lemma *nat-div-div-eq-div*: $y\ dvd\ z \Longrightarrow z \sim= 0 \Longrightarrow$
$((x::nat)\ div\ y\ div\ (z\ div\ y)) = x\ div\ z$
apply (*subst div-mult2-eq* [*THEN sym*])
apply (*subst nat-dvd-mult-div*)
apply (*subgoal-tac* $0 < y$)
apply *force*
apply (*rule nat-pos-dvd-pos*)
apply *assumption*
apply *force*
apply *assumption*
apply (*rule refl*)
done

lemma *dvd-pos-pos*: $0 < (n::nat) \Longrightarrow m\ dvd\ n \Longrightarrow 0 < m$
apply (*subgoal-tac* $m \sim= 0$)
apply *force*
apply (*rule notI*)

apply *simp*
done

lemma *nat-le-imp-1-le-div*: *0 < y ==> y <= (x::nat) ==> 1 <= x div y*
apply (*subgoal-tac y div y <= x div y*)
apply *simp*
apply (*rule div-le-mono*)
apply *assumption*
done

lemma *nat-div-times-le*: *((x::nat) div y) * y <= x*
apply (*subgoal-tac (x div y) * y + 0 <= (x div y) * y + x mod y*)
apply *force*
apply (*rule add-le-mono*)
apply *auto*
done

lemma *nat-pos-prop*: *[| 0 <= x; 0 <= y; nat x = nat y |] ==> x = y*
proof−
assume *0 <= x* **and** *0 <= y* **and** *nat x = nat y*
then have *int (nat x) = int (nat y)* **by** (*auto*)
also have *int (nat x) = x* **by** *auto*
also have *int (nat y) = y* **by** *auto*
finally show *x = y* **by** *auto*
qed

lemma *relprime-dvd-prod-dvd*: *gcd(a,b) = 1 ==> a dvd m ==>*
*b dvd m ==> (a * b) dvd (m::nat)*
apply (*unfold dvd-def*)
apply (*clarify*)
apply (*subgoal-tac a dvd ka*)
apply (*force simp add*: *dvd-def*)
apply (*subst relprime-dvd-mult-iff* [*THEN sym*])
apply *assumption*
apply (*auto simp add*: *mult-commute dvd-def*)
apply (*rule exI*)
apply (*erule sym*)
done

lemma *distinct-primes-gcd-1*: *p:prime ==> q:prime ==> p ~= q ==> gcd(p,q)*
= 1
apply (*rule prime-imp-relprime*)
apply *assumption*
apply (*auto simp add*: *prime-def*)
done

lemma *all-relprime-prod-relprime-nat*: *finite A ==> ALL x:A. gcd(x,y) = 1 ==>*
gcd(setprod id A, y) = 1
apply (*induct set*: *Finites*)
apply (*auto simp add*: *gcd-mult-cancel*)
done

lemma *distinct-primes-power-gcd-1-aux*: *p:prime ==> q:prime ==> p ~= q ==>*
gcd(p^a, q) = 1
apply (*induct-tac a*)
apply *simp*
apply (*subst power-Suc*)
apply (*subst gcd-mult-cancel*)
apply (*rule distinct-primes-gcd-1*)
apply *assumption+*
done

lemma *distinct-primes-power-gcd-1*: *p:prime ==> q:prime ==> p ~= q ==>*
gcd(p^a, q^b) = 1
apply (*induct-tac a*)
apply *simp*
apply (*subst power-Suc*)
apply (*subst gcd-mult-cancel*)
apply (*subst gcd-commute*)
apply (*rule distinct-primes-power-gcd-1-aux*)
apply *auto*
done

lemma *setprod-primes-dvd*: *finite A ==>*
ALL x:A. (x : prime & x dvd M) ==> setprod id A dvd M
apply (*induct set*: *Finites*)
apply *auto*
apply (*rule relprime-dvd-prod-dvd*)
apply (*subst gcd-commute*)
apply (*rule all-relprime-prod-relprime-nat*)
apply *assumption*
apply (*rule ballI*)
apply (*rule distinct-primes-gcd-1*)
apply *auto*
done

lemma *nat-div-gr-0*: *0 < x ==> (x::nat) <= y ==> 0 < y div x*

apply (*subgoal-tac y div x ~= 0*)
apply *force*
apply (*rule notI*)
apply (*insert mod-div-equality* [*of y x*])
apply *simp*
apply (*subgoal-tac y mod x < x*)
apply *arith*
apply (*erule mod-less-divisor*)
done

end

15 Unique factorization for integers

theory *IntFactorization* = *Factorization* + *NatIntLib*:

lemma *gr-1-prop*: (*1 < p*) = (*1 < nat p*)
by (*auto simp add*: *zless-nat-eq-int-zless*)

lemma *int-nat-dvd-prop*: (*m dvd p*) = (*int m dvd int p*)
apply (*auto simp add*: *dvd-def*)
apply (*rule-tac x = int k* **in** *exI*)
apply (*auto simp add*: *zmult-int*)
apply (*rule-tac x = nat k* **in** *exI*)
proof−
fix *k*
assume *int p = int m * k*
then have *nat* (*int p*) = *nat*(*int m * k*)
by (*auto*)
then have *p = nat*(*int m * k*)
by (*auto*)
also have ... = *nat*(*int m*) * *nat k*
apply (*insert nat-mult-distrib* [*of int m k*])
apply (*auto*)
done

```
  finally have p = nat (int m) * nat k.
  thus p = m * nat k by auto
qed

lemma nat-int-dvd-prop: [| 0 < m; m dvd p |] ==> (nat m dvd nat p)
  by (auto simp add: int-nat-dvd-prop)

lemma nat-prime-prop: (p: zprime) = ((nat p): prime)
proof
  assume p : zprime
  then have p:0 <= p by (auto simp add: zprime-def)
  from prems show (nat p):prime
    apply (auto simp add: prems zprime-def prime-def gr-1-prop)
    apply (drule-tac x = int m in allE, auto)
  proof-
    fix m
    assume m dvd nat p and ~ int m dvd p
    then have (int m) dvd int (nat p)
      by (insert int-nat-dvd-prop [of m nat p] prems, auto)
    also have int (nat p) = p by (auto simp add: p)
    finally have int m dvd p.
    with prems have False by auto
    thus m = Suc 0 by auto
  qed
next
  assume nat p : prime
  thus p : zprime
    apply (auto simp add: prime-def zprime-def gr-1-prop)
    apply (drule-tac x = nat m in allE)
    apply (auto)
  proof-
    fix m
    assume Suc 0 < nat p and 0 <= m and
          m dvd p and ~ nat m dvd nat p
    then have p: 1 < p by (auto simp add: gr-1-prop)
    from prems have ~(int (nat m) dvd int (nat p))
      by (auto simp add: int-nat-dvd-prop)
    also have int (nat m) = m by (auto simp add: prems)
    also have int (nat p) = p by (insert p, auto)
    finally have ~(m dvd p) .
    with prems have False by auto
    thus m = 1 by auto
  next
    fix m
    assume nat m = Suc 0 and p:0 <= m
```

```
    then have int (nat m) = int (Suc 0) by auto
    also have int (nat m) = m by (auto simp add: p)
    also have int (Suc 0) = 1 by auto
    finally show m = 1 .
  next
    fix m
    assume Suc 0 < nat p and p1:0 <= m and
           m ~= p and nat m = nat p
    then have p2: 1 < p by (auto simp add: gr-1-prop)
    from prems have int (nat m) = int (nat p)
      by auto
    also have int (nat m) = m by (auto simp add: p1)
    also have int (nat p) = p by (insert p2, auto)
    finally have m = p .
    with prems have False by auto
    thus m = 1 by auto
  qed
qed

lemma int-prime-prop: (int p : zprime) = (p : prime)
  by (auto simp add: nat-prime-prop)




consts
  intl     :: nat list => int list
  natl     :: int list => nat list
  zprimel  :: int list => bool
  znondec  :: int list => bool
  zprod    :: int list => int

primrec
  natl []       = []
  natl (x # xs) = (nat x) # (natl xs)

primrec
  intl []       = []
  intl (x # xs) = (int x) # (intl xs)

defs
  zprimel-def: zprimel xs == set xs ⊆ zprime

primrec
```

znondec [] = *True*
znondec (*x* # *xs*) = (*case xs of* [] => *True* | *y* # *ys* => *x* <= *y* ∧ *znondec xs*)

primrec
zprod [] = *1*
zprod (*x* # *xs*) = *x* ∗ *zprod xs*

15.1 Properties about intl and natl

lemma *intl-natl-prime* [*rule-format*]: *zprimel x* −−> *intl* (*natl x*) = *x*
by (*induct-tac x*, *auto simp add*: *zprimel-def zprime-def*)

lemma *intl-natl-prop* [*rule-format*]: *x* = *y* −−> *intl x* = *intl y*
by (*induct x*, *auto*)

15.2 Properties about zprimel

lemma *zprimel-distrib1*: *zprimel* (*h* # *t*) ==> (*h* : *zprime*)
by (*auto simp add*: *zprime-def zprimel-def*)

lemma *zprimel-distrib2*: *zprimel* (*h* # *t*) ==> *zprimel t*
by (*auto simp add*: *zprime-def zprimel-def*)

lemma *aux* [*rule-format*]: *zprimel* (*a* # *list*) −−> *zprimel y* −−> *nat a* # *natl list* = *natl y* −−>
a # *list* ~= *y* −−> *natl list* = *natl y*
apply (*induct y*, *force*)
apply (*clarify*, *auto*)
proof−
fix *aa* **and** *lista*
assume *zprimel* (*a* # *list*) **and** *zprimel* (*aa* # *lista*) **and** *nat a* = *nat aa*
then have *0* <= *a* **and** *0* <= *aa*
by (*auto simp add*: *zprimel-def zprime-def*)
thus *a* = *aa* **by** (*auto simp add*: *nat-pos-prop prems*)
next
fix *aa* **and** *lista*
assume *zprimel* (*a* # *list*) **and** *zprimel* (*aa* # *lista*) **and** *natl list* = *natl lista*
then have *intl* (*natl list*) = *intl* (*natl lista*)
by (*rule-tac x* = *natl list* **in** *intl-natl-prop*, *auto*)
also have *intl* (*natl list*) = *list*
proof−
from *prems* **have** *zprimel* (*a* # *list*) **by** *auto*
then have *zprimel list* **by** (*rule zprimel-distrib2*)
thus *intl* (*natl list*) = *list* **by** (*rule intl-natl-prime*)
qed

```
  also have intl (natl lista) = lista
  proof−
    from prems have zprimel (aa # lista) by auto
    then have zprimel lista by (rule zprimel-distrib2)
    thus intl (natl lista) = lista by (rule intl-natl-prime)
  qed
  finally show list = lista .
qed

lemma zprimel-natl-prop: [| zprimel x; zprimel y; natl x = natl y |] ==> x = y
proof−
  assume zprimel x and zprimel y and natl x = natl y
  have zprimel x & zprimel y & natl x = natl y −−> x = y
    apply (induct x, induct y, auto)
    apply (simp add: zprimel-def)
    apply (rule aux, auto)
  done
  with prems show x = y by auto
qed

lemma zprimel-conversion: primel l = zprimel (intl l)
  apply (induct l)
  apply (auto simp add: primel-def zprimel-def int-prime-prop)
done
```

15.3 Properties about zprod

```
lemma zprod-pos: 0 <= zprod(intl l)
  apply (induct l, auto)
proof−
  fix a and list
  assume 0 <= zprod (intl list)
  moreover have 0 <= int a by auto
  ultimately show 0 <= int a * zprod (intl list)
    by (auto simp add: zero-le-mult-iff)
qed

lemma zprod-conversion: prod l = nat (zprod(intl l))
  apply (induct l)
  apply (auto)
proof−
  fix a and list
  have a * nat (zprod (intl list)) =  nat (int a) * nat (zprod (intl list))
    by (auto)
  also have ... = nat (int a * zprod (intl list))
```

by (*auto simp add*: *nat-mult-distrib*)
finally show *a* ∗ *nat* (*zprod* (*intl list*)) = *nat* (*int a* ∗ *zprod* (*intl list*)).
qed

lemma *zprodl-zprimel-pos* [*rule-format*]: *zprimel pl* −−> *0* <= *zprod* (*pl*)
apply (*induct pl*)
apply (*auto simp add*: *zprimel-def zprime-def*)
apply (*auto simp add*: *zero-le-mult-iff*)
done

lemma *zprodl-zprimel-gr-0* [*rule-format*]: *zprimel pl* −−> *0* < *zprod* (*pl*)
apply (*induct pl*)
apply (*auto simp add*: *zprimel-def zprime-def*)
apply (*auto simp add*: *zero-less-mult-iff*)
done

15.4 Properties about znondec

lemma *znondec-conversion*: *nondec l* = *znondec* (*intl l*)
apply (*induct l*, *auto*)
apply (*case-tac list*, *auto simp add*: *neq-Nil-conv*)
apply (*case-tac list*, *auto simp add*: *neq-Nil-conv*)
apply (*case-tac list*, *auto simp add*: *neq-Nil-conv*)
apply (*case-tac list*, *auto simp add*: *neq-Nil-conv*)
done

lemma *znondec-distrib* [*rule-format*]: *znondec*(*a* # *list*) −−> *znondec*(*list*)
by (*induct list*, *auto*)

15.5 Uniqueness

lemma *temp*: (*1*::*int*) < *p* ==> ~(*p dvd 1*)
apply (*auto*)
proof−
assume *1* < *p* **and** *p dvd 1*
moreover from *prems* **have** *0* < *p* **by** *auto*
ultimately have *nat p dvd nat 1*
by (*rule-tac m* = *p* **in** *nat-int-dvd-prop*, *auto*)
with *dvd-1-iff-1* **have** *nat p* = *nat 1* **by** *auto*
then have *int* (*nat p*) = *int* (*nat 1*) **by** *auto*
with *prems* **show** *False* **by** (*auto*)
qed

lemma *zprime-zdvd-zmult-list* [*rule-format*]:
p : *zprime* ==> (*p dvd zprod xs*) −−> (*EX m*. *m* : *set xs* & *p dvd m*)

```
  apply (induct xs)
  apply (simp add: zprime-def)defer
  apply (clarsimp)
  apply (drule-tac p = p and m = a in zprime-zdvd-zmult-better)
  apply (force, force)
proof (clarify)
  assume 1 < p and p dvd 1
  moreover from prems have 0 < p by auto
  ultimately have nat p dvd nat 1
    by (rule-tac m = p in nat-int-dvd-prop, auto)
  with dvd-1-iff-1 have nat p = nat 1 by auto
  then have int (nat p) = int (nat 1) by auto
  with prems show False by (auto)
qed

lemma zprimes-eq: [| p : zprime; q : zprime; p dvd q |] ==> p = q
  apply (auto simp add: zprime-def)
  apply (drule-tac x = q in allE, auto)
  apply (drule-tac x = p in allE, auto)
done

lemma zprime-zdvd-eq: [| zprimel (x # xs); zprimel ys; m : set ys; x dvd m |]
==> x = m
  apply (rule zprimes-eq)
  apply (auto simp add: zprimel-distrib1)
  apply (auto simp add: zprimel-def)
done

lemma zfactor-unique: [| zprimel (intl x); zprimel (intl y);
                         znondec (intl x); znondec (intl y);
                         zprod (intl x) = zprod (intl y) |] ==> x = y
proof-
  assume zprimel (intl x) and zprimel (intl y) and
         znondec (intl x) and znondec (intl y) and
         zprod (intl x) = zprod (intl y)
  then have nat (zprod (intl y)) = nat (zprod (intl x)) by auto
  then have primel x & primel y & prod x = prod y
    by (auto simp add: zprimel-conversion zprod-conversion)
  with factor-unique have x <~~> y by auto
  moreover from prems have nondec x and nondec y
    by (auto simp add: znondec-conversion)
  moreover note perm-nondec-unique
  ultimately show x = y by auto
qed
```

15.6 Unique Factorization into Prime Integers

lemma *aux1*: *1 < n ==> EX! l. zprimel (intl l) & znondec (intl l) & zprod (intl l) = n*
proof (*auto*)
assume *1 < n*
then have *Suc 0 < nat n* **by** (*auto simp add*: *gr-1-prop*)
with *unique-prime-factorization* **have** *EX! l. primel l & nondec l & prod l = nat n*
by (*auto*)
then have *EX l. primel l & nondec l & prod l = nat n* **by** *auto*
then show *EX l. zprimel (intl l) & znondec (intl l) & zprod (intl l) = n*
apply (*auto simp add*: *zprimel-conversion zprod-conversion znondec-conversion*)
apply (*rule-tac x = l* **in** *exI*, *auto*)
apply (*insert prems*, *rule nat-pos-prop*, *auto simp add*: *zprod-pos*)
done
next
fix *l* **and** *y*
assume *zprimel (intl l)* **and** *zprimel (intl y)* **and**
znondec (intl l) **and** *znondec (intl y)* **and**
zprod (intl y) = zprod (intl l)
with *zfactor-unique* **show** *l = y* **by** *auto*
qed

theorem *unique-zprime-factorization*: *1 < n ==> EX! l. zprimel l & znondec l & zprod l = n*
proof−
assume *1 < n*
with *aux1* **have** *EX! l. zprimel (intl l) & znondec (intl l) & zprod (intl l) = n*
by *auto*
then show *EX! l. zprimel l & znondec l & zprod l = n*
apply (*auto*)
apply (*rule-tac x = la* **and** *y = y* **in** *zprimel-natl-prop*)
apply (*auto*)
apply (*rule-tac x = natl la* **in** *zfactor-unique*)
proof−
fix *la*
assume *zprimel la*
with *intl-natl-prime* **have** *intl (natl la) = la* **by** *auto*
then show *zprimel (intl (natl la))* **by** *auto*
next
fix *y*
assume *zprimel y*
with *intl-natl-prime* **have** *intl (natl y) = y* **by** *auto*
then show *zprimel (intl (natl y))* **by** *auto*

```
  next
    fix la
    assume zprimel la and znondec la
    with intl-natl-prime have intl (natl la) = la by auto
    then show znondec (intl (natl la)) by auto
  next
    fix y
    assume zprimel y and znondec y
    with intl-natl-prime have intl (natl y) = y by auto
    then show znondec (intl (natl y)) by auto
  next
    fix l and la and y
    assume zprimel la and zprimel y
    assume zprod la = zprod (intl l) and zprod y = zprod (intl l)
    then have zprod la = zprod y by auto
    also have la = intl (natl la)
      by (auto simp add: prems intl-natl-prime)
    also have y = intl (natl y)
      by (auto simp add: prems intl-natl-prime)
    finally show zprod (intl (natl la)) = zprod (intl (natl y)).
  qed
qed

end
```

16 Primes and multiplicity

```
theory PrimeFactorsList = IntFactorization:
```

```
consts
  zinsert      :: int => int list => int list
  zsort        :: int list => int list

primrec
  zinsert x []      = [x]
  zinsert x (h # t) = (if (x <= h) then (x # (h#t))
                       else (h # (zinsert x t)))
primrec
```

```
  zsort []      = []
  zsort (h # t) = (zinsert h (zsort t))
```

```
consts
  pfactors     :: int => int list
  numoccurs    :: 'a => 'a list => nat
  multiplicity :: int => int => nat

primrec
  numoccurs x []      = 0
  numoccurs x (h # t) = (if (x = h) then 1 + (numoccurs x t)
                         else (numoccurs x t))
defs
  pfactors-def:      pfactors n ==
                     (if (1 < n) then
                        (THE l. (zprimel l & znondec l & zprod l = n))
                      else [])
  multiplicity-def: multiplicity p n == (numoccurs p (pfactors n))
```

16.1 Show that zinsert and zsort play well with others

```
lemma znondec-zinsert [rule-format]: znondec (l) --> znondec (zinsert a (l))
  apply (induct l, force)
  apply (case-tac list, auto)
done

lemma znondec-zsort: znondec (zsort l)
  by (induct l, auto simp add: znondec-zinsert)

lemma zprimel-zinsert [rule-format]: zprimel l --> a : zprime --> zprimel
(zinsert a (l))
  by (induct l, auto simp add: zprimel-def)

lemma zprimel-zsort [rule-format]: zprimel l --> zprimel (zsort l)
  apply (induct l, force, clarify)
  apply (frule zprimel-distrib2)
  apply (drule zprimel-distrib1, clarify)
  apply (auto simp add: zprimel-zinsert)
done

lemma zprod-zinsert: (a * zprod l) = (zprod (zinsert a l))
```

by (*induct l*, *auto*)

lemma *zprod-zsort*: *zprod l* = *zprod* (*zsort l*)
by (*induct l*, *auto simp add*: *zprod-zinsert*)

lemma *numoccurs-zinsert1*: *numoccurs a l* + *1* = *numoccurs a* (*zinsert a l*)
by (*induct l*, *auto*)

lemma *numoccurs-zinsert2*: *a* ~= *p* ==> *numoccurs p l* = *numoccurs p* (*zinsert a l*)
by (*induct l*, *auto*)

lemma *numoccurs-zsort*: *numoccurs p l* = *numoccurs p* (*zsort l*)
apply (*induct l*, *auto simp add*: *numoccurs-zinsert2*)
apply (*insert numoccurs-zinsert1*, *auto*)
done

lemma *length-zinsert*: *length l* + *1* = *length* (*zinsert a l*)
by (*induct l*, *auto*)

lemma *length-zsort*: *length l* = *length* (*zsort l*)
by (*insert length-zinsert*, *induct l*, *auto*)

16.2 Some more intial properties for zprime and zprod

lemma *zprimel-cons*: (*zprimel lista* & *zprimel listb*) = *zprimel* (*lista* @ *listb*)
by (*auto simp add*: *zprimel-def*)

lemma *aux1* [*rule-format*]: *zprod lista* ∗ *zprod list* = *zprod* (*lista* @ *list*) −−>
zprod lista ∗ (*a* ∗ *zprod list*) = *zprod* (*lista* @ *a* # *list*)
by (*induct lista*, *auto*)

lemma *zprod-cons*: *zprod lista* ∗ *zprod listb* = *zprod* (*lista* @ *listb*)
by (*induct listb*, *auto simp add*: *aux1*)

lemma *zprod-zprime-prop* [*rule-format*]: *p*:*zprime* ==> *zprimel l* −−> *p dvd* (*zprod l*) −−> *p mem l*
proof (*induct l*, *auto*)
assume *p* :*zprime* **and** *p dvd 1*
thus *False*
apply (*auto simp add*: *zprime-def*)
apply (*drule zdvd-bounds*, *auto*)
done
next

fix *a* **and** *list*
assume *zprimel* (*a # list*) **and** ~ *zprimel list*
thus *p mem list* **by** (*simp add*: *zprimel-def*)
next
fix *a* **and** *list*
assume *p1*:*p*:*zprime* **and** *zprimel* (*a # list*) **and** *p dvd a * zprod list* **and** ~ *p dvd zprod list*
with *zprime-zdvd-zmult-better* **have** *p2*:*p dvd a* **by** *auto*
have *p3*: *a*: *zprime* **by** (*insert prems*, *auto simp add*: *zprimel-def*)
have *p = a*
apply (*insert p1 p2 p3*)
apply (*rule-tac zprimes-eq*, *auto*)
done
moreover assume *a ~= p*
ultimately show *p mem list* **by** *auto*
qed

16.3 Basic properties of numoccurs

lemma *not-mem-numoccurs-eq-0* [*rule-format*]: ~(*a mem list*) --> *numoccurs a list = 0*
by (*induct list*, *auto*)

lemma *mem-numoccurs-gr-0* [*rule-format*]: *a mem list --> 0 < numoccurs a list*
by (*induct list*, *auto*)

lemma *zprimel-not-zprime-numoccrs-eq-0* [*rule-format*]: *zprimel l --> p ~: zprime --> numoccurs p l = 0*
by (*induct l*, *auto simp add*: *zprimel-def*)

lemma *aux2* [*rule-format*]: *numoccurs a* (*l1 @ list*) = *numoccurs a l1 + numoccurs a list*
--> numoccurs a (*l1 @ aa # list*) = *numoccurs a l1 + numoccurs a* (*aa # list*)
by (*induct l1*, *auto*)

lemma *numoccurs-concat-zplus*: *numoccurs a* (*l1 @ l2*) = *numoccurs a l1 + numoccurs a l2*
by (*induct l2*, *auto simp add*: *aux2*)

16.4 More Properties for numoccurs

lemma *aux3*: (*ALL p. numoccurs p* (*a # l1*) <= *numoccurs p* (*a # l2*)) =
(*ALL p. numoccurs p l1 <= numoccurs p l2*)
by (*auto*)

lemma *aux4* [*rule-format*]: *znondec* (*b* # *aa* # *list*) −−> *znondec* (*b* # *list*)
by (*case-tac list*, *auto*)

lemma *znondec-l-numoccurs* [*rule-format*]: *a* < *b* −−> *znondec* (*b* # *list*) −−> *numoccurs a* (*b* # *list*) = *0*
apply (*induct list*, *force*, *clarify*)
apply (*frule aux4*)
apply (*drule mp*, *auto*)
done

lemma *aux5*: [| *znondec* (*b* # *lista*); *a* < *b*;
ALL p. *numoccurs p* (*b* # *lista*) <= *numoccurs p* (*a* # *listb*);
ALL l1. *znondec l1* & (*ALL p*. *numoccurs p l1* <= *numoccurs p listb*) −−> *zprod l1 dvd zprod listb* |]
==> *zprod* (*b* # *lista*) *dvd zprod listb*
apply (*drule-tac x* = *b* # *lista* **in** *spec*)
apply (*erule mp*)
apply (*rule conjI*, *force*)
apply (*rule allI*)
apply (*drule-tac x* = *p* **in** *spec*)
apply (*case-tac p* = *a*)
apply (*drule znondec-l-numoccurs*, *auto*)
done

lemma *aux6* [*rule-format*]: *znondec l2* −−>
(*ALL l1*. (*znondec l1* & (*ALL p*. (*numoccurs p l1* <= *numoccurs p l2*))) −−>
zprod l1 dvd zprod l2)
apply (*induct l2*, *clarify*)
apply (*case-tac l1*, *force*, *clarify*)
apply (*drule-tac x* = *a* **in** *spec*, *force*, *clarify*)
apply (*frule znondec-distrib*)
apply (*drule mp*, *force*)
apply (*case-tac l1*, *force*, *clarify*)
apply (*case-tac aa* = *a*, *clarify*)
apply (*drule-tac x* = *lista* **in** *spec*)
apply (*drule znondec-distrib*)+
apply (*clarsimp simp only*: *aux3*, *clarsimp*)
apply (*subgoal-tac a dvd a*)
apply (*simp add*: *zdvd-zmult-mono*, *force*)
apply (*case-tac a* < *aa*)
apply (*frule-tac a* = *a* **and** *b* = *aa* **and** *lista* = *lista* **and** *listb* = *list* **in** *aux5*)
apply (*force*)+
apply (*simp add*: *zdvd-zmult*)

```
  apply (subgoal-tac aa < a)
  apply (frule znondec-l-numoccurs, force)
  apply (drule-tac x = aa in spec)
  apply (force)+
done

lemma znondec-imp-zprod-zdvd: [| znondec lista; znondec listb;
                              ALL p. numoccurs p lista <= numoccurs p listb |] ==>
                              zprod lista dvd zprod listb
  by (auto simp add: aux6)

lemma zpower-numoccurs-zdvd-zprod: p^(numoccurs p l) dvd zprod l
  by (induct l, auto simp add: dvd-def)

lemma aux7: ALL k. p ^ k dvd zprod l --> zprimel l --> p:zprime --> k <=
(numoccurs p l)
  apply (induct l, auto) defer
  apply (case-tac k, auto)deferdefer
proof-
  fix k
  assume p : zprime
  then have p1: 1 < p by (auto simp add: zprime-def)
  then have 0 <= p ^ k
    by (induct k, auto simp add: zero-le-mult-iff)
  moreover assume p^k dvd 1
  ultimately have p ^ k = 1
    by (auto simp: ge-0-zdvd-1)
  then show k = 0
    apply (induct k)
    apply (insert p1)
    apply (auto simp add:zmult-eq-1-iff)
  done
next
  fix list fix nat
  assume IH: ALL k. p ^ k dvd zprod list --> zprimel list --> k <= numoccurs
p list and
        p1: p * p ^ nat dvd p * zprod list and
        p2: zprimel (p # list) and p3: p : zprime
  have p ^ nat dvd zprod list
  proof-
    from p3 have p ~= 0 by (auto simp add: zprime-def)
    with p1 show p ^ nat dvd zprod list
      by (rule-tac a = p in ne-0-zdvd-prop, auto)
  qed
  moreover from p2 have zprimel list
```

```
    by (auto simp add: zprimel-def)
  ultimately show nat <= numoccurs p list
    by (auto simp add: IH)
next
  fix a fix list fix k
  assume IH: ALL k. p ^ k dvd zprod list --> zprimel list --> k <= numoccurs
p list and
        p1: p ~= a and p2: p ^ k dvd a * zprod list and
        p3: zprimel (a # list) and p4: p : zprime
  from p3 have a:zprime by (auto simp add: zprimel-def)
  with p1 p4 have ~ p dvd a
    apply (auto simp add: zprime-def)
    apply (drule-tac x = a in allE, force)
    apply (drule-tac x = p in allE, auto)
  done
  moreover note p4 p2
  ultimately have p ^ k dvd zprod list
    apply (rule-tac p = p in zprime-zdvd-zpower)
    apply (auto simp add: zmult-ac)
  done
  moreover from p3 have zprimel list
    by (auto simp add: zprimel-def)
  ultimately show k <= numoccurs p list
    by (auto simp add: IH)
qed

lemma zpower-zdvd-zprod-impl-numoccurs [rule-format]: p ^ k dvd zprod l -->
zprimel l --> p:zprime --> k <= (numoccurs p l)
  by (auto simp add: aux7)
```

16.5 A Few Useful Lemmas

```
lemma zprimel-zprod-eq-1-impl-empty [rule-format]: zprimel l --> zprod l = 1
--> l = []
  apply (induct l)
  apply (auto simp add: zprimel-def zprime-def pos-zmult-eq-1-iff)
done

lemma zprimel-zprod-gr-1-impl-not-empty [rule-format]: zprimel l --> 1 < zprod
l --> l ~= []
  apply (induct l)
  apply (auto simp add: zprimel-def zprime-def pos-zmult-eq-1-iff)
done
```

16.6 Basic Properties about pfactors (from Unique Factorization)

lemma *pfactors-simp-prop*: [|*zprimel l*; *znondec l*; *zprod l = n* |] ==> *pfactors n = l*
apply (*simp add*: *pfactors-def*, *auto*)
apply (*rule the1-equality*)
apply (*auto simp add*: *unique-zprime-factorization*)
apply (*frule zprodl-zprimel-gr-0*)
apply (*case-tac zprod l = 1*)
apply (*auto simp add*: *zprimel-zprod-eq-1-impl-empty* [*THEN sym*])
done

lemma *pfactors-fundamental-prop*: *1 < n* ==> *zprimel* (*pfactors n*) & *znondec* (*pfactors n*) &
zprod (*pfactors n*) = *n*
apply (*simp add*: *pfactors-def*)
apply (*rule theI'*, *drule unique-zprime-factorization*)
apply (*auto*)
done

lemma *pfactors-zprimel*: *zprimel* (*pfactors n*)
apply (*case-tac 1 < n*)
apply (*auto simp add*: *pfactors-fundamental-prop*)
apply (*auto simp add*: *zprimel-def pfactors-def*)
done

lemma *pfactors-znondec*: *znondec*(*pfactors n*)
apply (*case-tac 1 < n*)
apply (*auto simp add*: *pfactors-fundamental-prop*)
apply (*auto simp add*: *pfactors-def*)
done

lemma *pfactors-zprod*: *1 <= n* ==> *zprod* (*pfactors n*) = *n*
apply (*case-tac 1 < n*)
apply (*auto simp add*: *pfactors-fundamental-prop*)
apply (*auto simp add*: *pfactors-def*)
done

lemma *pfactors-le-1*: *n <= 1* ==> *pfactors n* = []
by (*auto simp add*: *pfactors-def*)

lemma *pfactors-gr-1*: *1 < n* ==> *pfactors n* ~= []
proof−
assume *1 < n*

also have *n = zprod(pfactors n)*
by (*insert prems*, *auto simp add*: *pfactors-zprod*)
finally have *1 < zprod(pfactors n)* **.**
moreover have *zprimel* (*pfactors n*)
by(*auto simp add*: *pfactors-zprimel*)
ultimately show *?thesis*
by (*auto simp add*:*zprimel-zprod-gr-1-impl-not-empty*)
qed

lemmas *pfactors-ac = pfactors-le-1 pfactors-gr-1 pfactors-zprimel*
pfactors-znondec pfactors-zprod

16.7 More Properties About pfactors

lemma *pfactors-zprime*: *p*:*zprime* ==> *pfactors p* = [*p*]
apply (*auto simp add*: *pfactors-def*)
apply (*rule the1-equality*)
apply (*drule unique-zprime-factorization*)
apply (*auto simp add*: *zprime-def zprimel-def*)
done

lemma *length-pfactors-zprime*: *p*:*zprime* ==> *length* (*pfactors p*) = *1*
by (*auto simp add*: *pfactors-zprime*)

lemma *zprod-zdvd* [*rule-format*]: *x mem list* −−> *x dvd* (*zprod list*)
by (*induct list*, *auto simp add*: *dvd-def*)

lemma *mem-pfactors-imp-zdvd*: [| *1* <= *n*; *p mem* (*pfactors n*) |] ==> *p dvd n*
proof−
assume *1* <= *n* **and** *p mem* (*pfactors n*)
then have *p dvd* (*zprod* (*pfactors n*))
by (*auto simp add*: *zprod-zdvd*)
also have *zprod* (*pfactors n*) = *n*
by (*insert prems*, *auto simp add*: *pfactors-ac*)
finally show *?thesis***.**
qed

lemma *zdvd-imp-mem-pfactors*: [| *1* <= *n*; *p*:*zprime*; *p dvd n* |] ==> *p mem* (*pfactors n*)
proof−
assume *1* <= *n* **and** *p dvd n*
then have *p dvd zprod* (*pfactors n*)
by (*auto simp add*: *pfactors-ac*)
moreover have *zprimel* (*pfactors n*)
by (*insert prems*, *auto simp add*: *pfactors-zprimel*)

moreover assume *p*:*zprime*
ultimately show *p mem* (*pfactors n*)
by (*auto simp add*: *zprod-zprime-prop*)
qed

lemma *pfactors-zsort-cons*: [| *1* <= *a*; *1* <= *b* |] ==> *pfactors* (*a* * *b*) = *zsort*((*pfactors a*) @ (*pfactors b*))
apply (*rule pfactors-simp-prop*)
proof−
have *zprimel* (*pfactors a*) **and** *zprimel* (*pfactors b*)
by (*auto simp add*: *pfactors-ac*)
then have *zprimel* (*pfactors a* @ *pfactors b*)
by (*insert zprimel-cons*, *auto*)
thus *zprimel* (*zsort* (*pfactors a* @ *pfactors b*))
by (*auto simp add*: *zprimel-zsort*)
next
show *znondec* (*zsort* (*pfactors a* @ *pfactors b*))
by (*auto simp add*: *znondec-zsort*)
next
assume *1* <= *a* **and** *1* <= *b*
have *zprod* (*zsort* (*pfactors a* @ *pfactors b*)) = *zprod* (*pfactors a* @ *pfactors b*)
by (*auto simp add*: *zprod-zsort* [*THEN sym*])
also have ... = *zprod* (*pfactors a*) * *zprod* (*pfactors b*)
by (*auto simp add*: *zprod-cons* [*THEN sym*])
also have *zprod* (*pfactors a*) = *a*
by (*insert prems*, *auto simp add*: *pfactors-ac*)
also have *zprod* (*pfactors b*) = *b*
by (*insert prems*, *auto simp add*: *pfactors-ac*)
finally show *zprod* (*zsort* (*pfactors a* @ *pfactors b*)) = *a* * *b* **by** *auto*
qed

lemma *pfactors-zmult-length*: [| *1* <= *a*; *1* <= *b* |] ==>
length (*pfactors* (*a* * *b*)) = *length*(*pfactors a*) + *length*(*pfactors b*)
proof−
assume *1* <= *a* **and** *1* <= *b*
then have *length*(*pfactors* (*a* * *b*)) = *length*(*zsort*((*pfactors a*) @ (*pfactors b*)))
by (*auto simp add*: *pfactors-zsort-cons*)
also have ... = *length*(*pfactors a* @ *pfactors b*)
by (*insert length-zsort* [*of pfactors a* @ *pfactors b*], *auto*)
also have ... = *length* (*pfactors a*) + *length* (*pfactors b*)
by (*auto*)
finally show *?thesis* **by** *auto*
qed

lemma *pfactors-zprime-zmult-length*: [| *1 <= d*; *p*:*zprime*|] ==>
length (*pfactors* (*d* * *p*)) = *length*(*pfactors d*) + *1*
proof−
assume *1 <= d*
assume *p* : *zprime* **then have** *1 <= p* **by** (*auto simp add*: *zprime-def*)
then have *length* (*pfactors* (*d* * *p*)) = *length*(*pfactors d*) + *length*(*pfactors p*)
by (*auto simp add*: *prems pfactors-zmult-length*)
also have *length* (*pfactors p*) = *1*
by (*insert prems*, *auto simp add*: *pfactors-zprime*)
finally show *?thesis* **by** *auto*
qed

16.8 Properties for Multiplicity

lemma *multiplicity-base*: *p* :*zprime* ==> *multiplicity p p* = *1*
by (*auto simp add*: *multiplicity-def pfactors-zprime*)

lemma *multiplicity-zmult-distrib*: [| *1 <= a*; *1 <= b* |] ==>
multiplicity p (*a* * *b*) = *multiplicity p a* + *multiplicity p b*
apply (*auto simp add*: *multiplicity-def*)
proof−
assume *1 <= a* **and** *1 <= b*
then have *pfactors* (*a* * *b*) = *zsort*((*pfactors a*) @ (*pfactors b*))
by (*auto simp add*: *pfactors-zsort-cons*)
then have *numoccurs p* (*pfactors* (*a* * *b*)) = *numoccurs p* (*zsort*((*pfactors a*) @ (*pfactors b*)))
by (*auto*)
also have ... = *numoccurs p* ((*pfactors a*) @ (*pfactors b*))
by (*auto simp add*: *numoccurs-zsort* [*THEN sym*])
also have ... = *numoccurs p* (*pfactors a*) + *numoccurs p* (*pfactors b*)
by (*auto simp add*: *numoccurs-concat-zplus*)
finally show *numoccurs p* (*pfactors* (*a* * *b*)) = *numoccurs p* (*pfactors a*) + *numoccurs p* (*pfactors b*)
by *auto*
qed

lemma *multiplicity-zpower-prop*: *p*:*zprime* ==> *multiplicity p* (*p*^*k*) = *k*
apply (*induct k*, *simp add*: *multiplicity-def pfactors-def*)
proof (*auto*)
fix *n*
assume *multiplicity p* (*p* ^ *n*) = *n*
assume *p*:*zprime* **then have** *p1*: *1 <= p* **by** (*auto simp add*: *zprime-def*)
then have *1 <= p* ^ *n* **by** (*auto simp add*: *ge-1-imp-zpower-ge-1*)
then have *multiplicity p* (*p* * *p* ^ *n*) = *multiplicity p p* + *multiplicity p* (*p* ^ *n*)

```
    by (insert p1 multiplicity-zmult-distrib, auto)
  also have multiplicity p p = 1
    by (insert prems multiplicity-base, auto)
  also have multiplicity p (p ^ n) = n by (auto simp add: prems)
  finally show multiplicity p (p * p ^ n) = Suc n by auto
qed

lemma multiplicity-zdvd: p ^ (multiplicity p n) dvd n
proof (case-tac 1 < n)
  assume 1 < n
  have p^(numoccurs p (pfactors n)) dvd zprod (pfactors n)
    by ( auto simp add:  zpower-numoccurs-zdvd-zprod)
  also have zprod (pfactors n) = n
    by (insert prems, auto simp add: pfactors-ac)
  also have numoccurs p (pfactors n) = multiplicity p n
    by (auto simp add: multiplicity-def)
  finally show ?thesis .
next
  assume ~(1 < n)
  then have p1: n <= 1 by auto
  have p ^ 0 dvd n by auto
  also have 0 = multiplicity p n
    by (insert p1, auto simp add: multiplicity-def pfactors-def)
  finally show ?thesis .
qed

lemma not-zprime-multiplicity-eq-0: [| p ~: zprime |] ==> multiplicity p n = 0
  apply (insert pfactors-zprimel [of n])
  apply (auto simp add: multiplicity-def zprimel-not-zprime-numoccrs-eq-0)
done

lemma aux8: k <= multiplicity p n ==> p ^ k dvd n
proof-
  assume k <= multiplicity p n
  then have p ^ k dvd p ^ multiplicity p n
    by (auto simp add: le-imp-zpower-zdvd)
  moreover from  multiplicity-zdvd have p ^ multiplicity p n dvd n
    by auto
  ultimately show ?thesis by (auto simp add: dvd-def)
qed

lemma aux9: [| 0 < n; p:zprime; p ^ k dvd n |]==> k <= multiplicity p n
  apply (auto simp add: multiplicity-def)
  apply (case-tac 1 < n)
  apply (rule zpower-zdvd-zprod-impl-numoccurs)
```

```
  apply (auto simp add: pfactors-ac)
  apply (subgoal-tac n <= 1, auto)
  apply (case-tac n = 1, auto)
proof−
  assume p:zprime
  then have p1: 1 < p by (auto simp add: zprime-def)
  then have 0 <= p ^ k
    by (induct k, auto simp add: zero-le-mult-iff)
  moreover assume p^k dvd 1
  ultimately have p ^ k = 1
    by (auto simp: ge-0-zdvd-1)
  then show k = 0
    apply (induct k, insert p1)
    apply (auto simp add:zmult-eq-1-iff)
  done
qed

lemma multiplicity-zpower-zdvd: [| 0 < n; p:zprime |] ==> (k <= multiplicity p
n) = (p ^ k dvd n)
  by (auto simp add: aux8, simp add: aux9)

lemma zdvd-zprime-imp-multiplicity-ge-1: [| 0 < n; p dvd n; p:zprime |] ==> 1
<= multiplicity p n
  by (insert multiplicity-zpower-zdvd [of n p 1], auto)

lemma multiplicity-eq-1-imp-zdvd: [| multiplicity p n = 1 |] ==> p dvd n
  apply (case-tac 0 < n)
proof−
  assume multiplicity p n = 1
  then have multiplicity p n ~= 0 by auto
  then have p1: p : zprime
    by (insert not-zprime-multiplicity-eq-0 [of p n], auto)
  moreover assume 0 < n
  moreover have 1 <= multiplicity p n by (insert prems, auto)
  ultimately have (p ^ 1 dvd n)
    by (insert multiplicity-zpower-zdvd [of n p 1], auto)
  thus ?thesis by auto
next
  assume multiplicity p n = 1 and ~ 0 < n
  thus ?thesis by (simp add: multiplicity-def pfactors-def)
qed

lemma zdvd-imp-multiplicity-le: [| 0 < n; m dvd n |] ==> ALL p. (multiplicity p
m <= multiplicity p n)
proof (clarify, case-tac p : zprime)
```

```
  fix p
  assume m dvd n
  then have p ^ (multiplicity p m) dvd m
    by (auto simp add: multiplicity-zdvd)
  then have p ^ (multiplicity p m) dvd n
    by (insert prems zdvd-trans [of p ^ (multiplicity p m) m n], auto)
  moreover assume p : zprime and 0 < n
  ultimately show (multiplicity p m <= multiplicity p n)
    by (insert multiplicity-zpower-zdvd [of n p multiplicity p m], auto)
next
  fix p
  assume p ~: zprime
  thus multiplicity p m <= multiplicity p n
     by (auto simp add: not-zprime-multiplicity-eq-0)
qed

lemma multiplicity-le-imp-zdvd: [| 0 < m; 0 < n;
                                ALL p. (multiplicity p m <= multiplicity p n) |] ==>
                                m dvd n
proof-
  assume 0 < m and 0 < n
  assume ALL p. (multiplicity p m <= multiplicity p n)
  then have ALL p. (numoccurs p (pfactors m) <= numoccurs p (pfactors n))
    by (auto simp add: multiplicity-def)
  moreover have znondec (pfactors m) by (auto simp add: pfactors-ac)
  moreover have znondec (pfactors n) by (auto simp add: pfactors-ac)
  moreover note znondec-imp-zprod-zdvd
  ultimately have zprod (pfactors m) dvd zprod (pfactors n) by auto
 also have zprod (pfactors m) = m by (insert prems, auto simp add: pfactors-ac)
  also have zprod (pfactors n) = n by (insert prems, auto simp add: pfactors-ac)
  finally show ?thesis .
qed



lemma p-neq-q-impl-nzdvd: [| p ~= q; p:zprime; q:zprime |] ==> ~(p dvd q)
proof (auto simp add: zprime-def)
  assume 1 < p and p dvd q and ALL m. 0 <= m & m dvd q --> m = 1 | m
= q
  then have p = q by (drule-tac x = p in allE, auto)
  moreover assume p ~= q
  ultimately show False by auto
qed
```

lemma *multiplicity-p-1-eq-0*: *multiplicity p 1 = 0*
by (*auto simp add*: *multiplicity-def pfactors-def*)

lemma *zdvd-zprime-multiplicity-ge-1*: [| *0 < n*; *p*:*zprime* |] ==> (*p dvd n*) = (*1 <= multiplicity p n*)
by (*insert multiplicity-zpower-zdvd* [*of n p 1*], *auto*)

lemma *finite-prime-divisors*: *0 < n* ==> *finite* {*p*. *p*:*zprime* & *p dvd n*}
proof−
assume *0 < n*
then have {*p*. *p*:*zprime* & *p dvd n*} <= {*0*..*n*}
apply (*auto simp add*: *zprime-def*)
apply (*frule zdvd-bounds*, *auto*)
done
moreover have *finite* {*0*..*n*} **by** *auto*
ultimately show *?thesis* **by** (*auto simp add*: *finite-subset*)
qed

lemma *zdvd-imp-m-eq-n*: [| (*0*::*int*) < *m*; *0 < n*; *m dvd n*; *n dvd m* |] ==> *m = n*
apply (*auto simp add*: *dvd-def*)
proof−
fix *k* **and** *ka*
assume *0 < m* **and** *0 < m * k*
then have *0 < k* **by** (*elim zero-less-mult-pos*)
moreover assume *k * ka = 1*
ultimately show *k = 1* **by** (*auto simp add*: *pos-zmult-eq-1-iff*)
qed

lemma *multiplicity-imp-m-eq-n*: [| (*0*::*int*) < *m*; *0 < n*;
ALL p. (*multiplicity p m*) = (*multiplicity p n*) |] ==> *m = n*
proof−
assume *ALL p*. *multiplicity p m = multiplicity p n*
then have *ALL p*. *multiplicity p m <= multiplicity p n* **and**
ALL p. *multiplicity p n <= multiplicity p m* **by** *auto*
moreover assume *0 < m* **and** *0 < n*

ultimately have *m dvd n* **and** *n dvd m* **by** (*auto simp add*: *multiplicity-le-imp-zdvd*)
with *prems* **show** *m = n* **by** (*auto simp add*: *zdvd-imp-m-eq-n*)
qed

lemma *aux2*: *finite F ==> ALL a:F. 0 < a ==> (0::int) < setprod (%q. q ^ g q x) (F::int set)*
by (*induct F rule*: *finite-induct*, *auto simp add*: *mult-pos zero-less-power*)

lemma *aux3*: [| *finite A*; *0 < p*; *ALL a:A. 0 < a* |] ==>
multiplicity p (setprod (%q. q ^ (g q x)) A) =
setsum (%q. multiplicity p (q ^ (g q x))) A
apply (*induct A rule*: *finite-induct*, *auto simp add*: *multiplicity-p-1-eq-0*)
proof−
fix *F* **and** *xa*
assume *finite (F::int set)* **and** *ALL x:F. 0 < x* **and** *(0::int) < xa*
then have *0 < xa ^ (g xa x)* **by** (*auto simp add*: *zero-less-power*)
moreover have *0 < setprod (%q. q ^ g q x) F*
by (*insert prems*, *auto simp add*: *aux2*)
ultimately have *1 <= xa ^ (g xa x)* **and** *1 <= setprod (%q. q ^ g q x) F* **by** *auto*
then have *multiplicity p (xa ^ g xa x * setprod (%q. q ^ g q x) F) =*
multiplicity p (xa ^ g xa x) + multiplicity p (setprod (%q. q ^ g q x) F)
by (*rule multiplicity-zmult-distrib*)
also assume *multiplicity p (setprod (%q. q ^ g q x) F) = (∑ q:F. multiplicity p (q ^ g q x))*
finally show *multiplicity p (xa ^ g xa x * setprod (%q. q ^ g q x) F) =*
multiplicity p (xa ^ g xa x) + (∑ q:F. multiplicity p (q ^ g q x)) **.**
qed

lemma *multiplicity-zpower-zmult*: *0 < x ==> (multiplicity p (x ^ n)) = n * (multiplicity p x)*
apply (*induct n*, *auto simp add*: *multiplicity-p-1-eq-0*)
proof−
fix *n*
assume *0 < x*
then have *0 < x ^ n* **by** (*induct n*, *auto simp add*: *mult-pos*)
then have *multiplicity p (x * x ^ n) = (multiplicity p x) + (multiplicity p (x^n))*
by (*insert prems*, *auto simp add*: *multiplicity-zmult-distrib*)
also assume *multiplicity p (x ^ n) = n * multiplicity p x*
finally show *multiplicity p (x * x ^ n) = multiplicity p x + n * multiplicity p x***.**
qed

lemma *p-neq-q-multiplicity-eq-0*: [| *q ~= p*; *p:zprime*; *q:zprime* |] ==> *multiplicity*

p q = 0
proof−
assume *q ~= p* **and** *p : zprime* **and** *q : zprime*
then have *~(p dvd q)* **by** (*auto simp add*: *p-neq-q-impl-nzdvd*)
moreover have *0 < q* **by** (*insert prems*, *auto simp add*: *zprime-def*)
ultimately have *~(1 <= multiplicity p q)*
by (*auto simp add*: *prems zdvd-zprime-multiplicity-ge-1*)
thus *?thesis* **by** *auto*
qed

lemma *p-neq-q-multiplicity-zpower-eq-0*: [| *q ~= p*; *p*:*zprime*; *q*:*zprime* |] ==> *multiplicity p (q ^ n) = 0*
proof−
assume *q ~= p* **and** *p*:*zprime* **and** *q*:*zprime*
then have *0 < q* **by** (*auto simp add*: *zprime-def*)
then have (*multiplicity p (q ^ n)*) = *n* * (*multiplicity p q*)
by (*auto simp add*: *multiplicity-zpower-zmult*)
also have *multiplicity p q = 0* **by** (*insert prems*, *auto simp add*: *p-neq-q-multiplicity-eq-0*)
finally show *?thesis* **by** *auto*
qed

lemma *finite-A-setsum-eq-0*: *finite A* ==> *ALL x*:*A*. *f x = 0* ==> *setsum f A* = (*0*::*int*)
by (*induct A rule*: *finite-induct*, *auto*)

lemma *multiplicity-setprod-eq-0*: [| *finite A*; *ALL p*:*A*. (*p*:*zprime*); *p ~*: *A*; *p*:*zprime* |] ==>
multiplicity p (setprod (%q. q ^ (g q x)) A) = 0
proof−
assume *finite A* **and** *ALL p*:*A*. *p : zprime* **and** *p ~*:*A* **and** *p*:*zprime*
moreover have *0 < p* **and** *ALL p*:*A*. *0 < p* **by** (*insert prems*, *auto simp add*: *zprime-def*)
ultimately have *multiplicity p (setprod (%q. q ^ (g q x)) A) =*
setsum (%q. multiplicity p (q ^ (g q x))) A
by (*auto simp add*: *aux3*)
also have *... = 0*
apply (*insert prems*, *auto simp add*: *finite-A-setsum-eq-0*)
apply (*rule p-neq-q-multiplicity-zpower-eq-0*, *auto*)
done
finally show *?thesis* **.**
qed

lemma *multiplicity-zdvd-setprod-eq-n*: [| (*0*::*int*) < *n*; *q*:*zprime*; *q dvd n* |] ==>
multiplicity q (setprod (%p. p^(multiplicity p n)) {p. p: zprime & p dvd n}) =

```
proof−
  assume 0 < n and q:zprime and q dvd n
  have p1: finite {p. p: zprime & p dvd n & q ~= p}
     apply (insert finite-prime-divisors [of n] prems)
     apply (subgoal-tac {p. p: zprime & p dvd n & q ~= p} <= {p. p:zprime &
p dvd n})
     apply (auto simp add: finite-subset)
  done
  have {p. p:zprime & p dvd n} = insert q {p . p:zprime & p dvd n & q ~= p}
    by (insert prems, auto)
  then have multiplicity q (setprod (%p. p^(multiplicity p n)) {p. p: zprime & p
dvd n}) =
          multiplicity q (setprod (%p. p^(multiplicity p n)) (insert q {p. p: zprime
& p dvd n & q ~= p}))
    by auto
  also have (setprod (%p. p^(multiplicity p n)) (insert q {p. p: zprime & p dvd n
& q ~= p})) =
           (%p. p^(multiplicity p n)) q * setprod (%p. p^(multiplicity p n)) {p. p:
zprime & p dvd n & q ~= p}
  proof−
    note p1
    moreover have q ~: {p. p: zprime & p dvd n & q ~= p} by auto
    ultimately show ?thesis by (auto simp add: setprod-insert)
  qed
  finally have multiplicity q (setprod (%p. p ^ multiplicity p n) {p. p : zprime &
p dvd n}) =
              multiplicity q (q ^ multiplicity q n *
                          setprod (%p. p ^ multiplicity p n) {p. p : zprime & p dvd
n & q ~= p}) .
  also have ... = (multiplicity q (q^(multiplicity q n))) +
                (multiplicity q (setprod (%p. p^(multiplicity p n)) {p. p: zprime &
p dvd n & q ~= p}))
  proof−
     have p2: ALL p:{p. p : zprime & p dvd n & q ~= p}. 0 < p by (auto simp
add: zprime-def)
    have 0 < q by (insert prems, auto simp add: zprime-def)
    then have 0 < q ^ multiplicity q n by (auto simp add: zero-less-power)
    moreover have 0 < setprod (%p. p ^ multiplicity p n) {p. p : zprime & p dvd
n & q ~= p}
      apply (insert p1 p2 prems)
      apply (induct {p. p : zprime & p dvd n & q ~= p} rule: finite-induct)
      apply (auto simp add: mult-pos zero-less-power)
    done
    ultimately show ?thesis by (auto simp add: multiplicity-zmult-distrib)
```

```
  qed
  also have (multiplicity q (setprod (%p. p^(multiplicity p n)) {p. p: zprime & p
dvd n & q ~= p})) = 0
  proof−
    note p1
    moreover have ALL p:{p. p : zprime & p dvd n & q ~= p}. (p:zprime) by
auto
    moreover have q ~: {p. p : zprime & p dvd n & q ~= p} by auto
    moreover have q:zprime by (insert prems, auto)
    ultimately show ?thesis by (rule   multiplicity-setprod-eq-0)
  qed
  also have (multiplicity q (q^(multiplicity q n))) = multiplicity q n
    by (insert prems, auto simp add: multiplicity-zpower-prop)
  finally show ?thesis by auto
qed

lemma multiplicity-nzdvd-setprod-eq-n: [| 0 < n; q: zprime; ~(q dvd n) |] ==>
                                    multiplicity q (setprod (%p. p ^ (multiplicity p
n)) {p. p:zprime & p dvd n}) =
                                  multiplicity q n
proof−
  assume 0 < n and q: zprime and ~(q dvd n)
  then have multiplicity q n = 0 by (insert zdvd-zprime-multiplicity-ge-1, auto)
  moreover have multiplicity q (setprod (%p. p ^ (multiplicity p n)) {p. p:zprime
& p dvd n}) = 0
  proof−
   have finite {p. p:zprime & p dvd n} by (insert prems, auto simp add: finite-prime-divisors)
    moreover have ALL p:{p. p:zprime & p dvd n}. p:zprime by (auto)
    moreover have q ~: {p. p:zprime & p dvd n} by (auto simp add: prems)
    moreover note prems
    ultimately show ?thesis
      apply (rule-tac  multiplicity-setprod-eq-0)
      apply (force)+
    done
  qed
  ultimately show ?thesis by auto
qed

lemma multiplicity-setprod-eq-n: [| 0 < n; q: zprime |] ==>
          multiplicity q (setprod (%p. p ^ (multiplicity p n)) {p. p:zprime & p dvd
n}) =
           multiplicity q n
  apply (case-tac q dvd n)
 apply (auto simp add: multiplicity-nzdvd-setprod-eq-n multiplicity-zdvd-setprod-eq-n)
done
```

theorem *n-eq-setprod-multiplicity*: *0 < n ==> n = setprod (%p. p ^(multiplicity p n)) {p. p:zprime & p dvd n}*
apply (*rule multiplicity-imp-m-eq-n, auto*)
apply (*rule aux2, auto simp add*: *finite-prime-divisors*)
apply (*force simp add*: *zprime-def*)
apply (*case-tac p : zprime*)
apply (*auto simp add*: *multiplicity-setprod-eq-n* [*THEN sym*] *not-zprime-multiplicity-eq-0*)
done

end

17 Parity: Even and Odd Integers

theory *EvenOdd2 = NatIntLib*:

constdefs
zOdd :: *int set*
*zOdd == {x. ∃k. x = 2*k + 1}*
zEven :: *int set*
*zEven == {x. ∃k. x = 2 * k}*

lemma *one-not-even*: *~(1 ∈ zEven)*
apply (*simp add*: *zEven-def*)
apply (*rule allI, case-tac k ≤ 0, auto*)
done

lemma *even-odd-conj*: *~(x ∈ zOdd & x ∈ zEven)*
apply (*auto simp add*: *zOdd-def zEven-def*)
proof −
fix *a b*
assume *2 * (a::int) = 2 * (b::int) + 1*
then have *2 * (a::int) − 2 * (b :: int) = 1*
by *arith*
then have *2 * (a − b) = 1*
by (*auto simp add*: *zdiff-zmult-distrib*)
moreover have *(2 * (a − b)):zEven*

```
    by (auto simp only: zEven-def)
  ultimately show False
    by (auto simp add: one-not-even)
qed

lemma even-odd-disj: (x ∈ zOdd | x ∈ zEven)
 apply (auto simp add: zOdd-def zEven-def)
 proof −
  assume ∀k. x ≠ 2 * k
  have 0 ≤ (x mod 2) & (x mod 2) < 2
    by (auto intro: pos-mod-sign pos-mod-bound)
  then have x mod 2 = 0 | x mod 2 = 1 by arith
  moreover from prems have x mod 2 ≠ 0 by arith
  ultimately have x mod 2 = 1 by auto
  thus ∃k. x = 2 * k + 1
    by (insert zmod-zdiv-equality [of x 2], auto)
qed

lemma not-odd-impl-even: ~(x ∈ zOdd) ==> x ∈ zEven
 by (insert even-odd-disj, auto)

lemma odd-mult-odd-prop: (x*y):zOdd ==> x ∈ zOdd
 apply (case-tac x ∈ zOdd, auto)
 apply (drule not-odd-impl-even)
 apply (auto simp add: zEven-def zOdd-def)
 proof −
  fix a b
  assume 2 * a * y = 2 * b + 1
  then have 2 * a * y − 2 * b = 1
    by arith
  then have 2 * (a * y − b) = 1
    by (auto simp add: zdiff-zmult-distrib)
  moreover have (2 * (a * y − b)):zEven
    by (auto simp only: zEven-def)
  ultimately show False
    by (auto simp add: one-not-even)
qed

lemma odd-minus-one-even: x ∈ zOdd ==> (x − 1):zEven
 by (auto simp add: zOdd-def zEven-def)

lemma even-div-2-prop1: x ∈ zEven ==> (x mod 2) = 0
 by (auto simp add: zEven-def)

lemma even-div-2-prop2: x ∈ zEven ==> (2 * (x div 2)) = x
```

by (*auto simp add*: *zEven-def*)

lemma *even-plus-even*: [| $x \in zEven$; $y \in zEven$ |] ==> $x + y \in zEven$
apply (*auto simp add*: *zEven-def*)
by (*auto simp only*: *zadd-zmult-distrib2* [*THEN sym*])

lemma *even-times-either*: $x \in zEven$ ==> $x * y \in zEven$
by (*auto simp add*: *zEven-def*)

lemma *even-minus-even*: [| $x \in zEven$; $y \in zEven$ |] ==> $x - y \in zEven$
apply (*auto simp add*: *zEven-def*)
by (*auto simp only*: *zdiff-zmult-distrib2* [*THEN sym*])

lemma *odd-minus-odd*: [| $x \in zOdd$; $y \in zOdd$ |] ==> $x - y \in zEven$
apply (*auto simp add*: *zOdd-def zEven-def*)
by (*auto simp only*: *zdiff-zmult-distrib2* [*THEN sym*])

lemma *even-minus-odd*: [| $x \in zEven$; $y \in zOdd$ |] ==> $x - y \in zOdd$
apply (*auto simp add*: *zOdd-def zEven-def*)
apply (*rule-tac* $x = k - ka - 1$ **in** *exI*)
by *auto*

lemma *odd-minus-even*: [| $x \in zOdd$; $y \in zEven$ |] ==> $x - y \in zOdd$
apply (*auto simp add*: *zOdd-def zEven-def*)
by (*auto simp only*: *zdiff-zmult-distrib2* [*THEN sym*])

lemma *odd-times-odd*: [| $x \in zOdd$; $y \in zOdd$ |] ==> $x * y \in zOdd$
apply (*auto simp add*: *zOdd-def zadd-zmult-distrib zadd-zmult-distrib2*)
apply (*rule-tac* $x = 2 * ka * k + ka + k$ **in** *exI*)
by (*auto simp add*: *zadd-zmult-distrib*)

lemma *odd-iff-not-even*: $(x \in zOdd) = (\sim (x \in zEven))$
by (*insert even-odd-conj even-odd-disj*, *auto*)

lemma *even-product*: $x * y \in zEven$ ==> $x \in zEven \mid y \in zEven$
by (*insert odd-iff-not-even odd-times-odd*, *auto*)

lemma *even-diff*: $x - y \in zEven = ((x \in zEven) = (y \in zEven))$
apply (*auto simp add*: *odd-iff-not-even even-minus-even odd-minus-odd*
even-minus-odd odd-minus-even)
proof −
assume $x - y \in zEven$ **and** $x \in zEven$
show $y \in zEven$
proof (*rule classical*)
assume $\sim(y \in zEven)$

```
    then have y ∈ zOdd
      by (auto simp add: odd-iff-not-even)
    with prems have x − y ∈ zOdd
      by (simp add: even-minus-odd)
    with prems have False
      by (auto simp add: odd-iff-not-even)
    thus ?thesis
      by auto
  qed
  next assume x − y ∈ zEven and y ∈ zEven
  show x ∈ zEven
  proof (rule classical)
    assume ~(x ∈ zEven)
    then have x ∈ zOdd
      by (auto simp add: odd-iff-not-even)
    with prems have x − y ∈ zOdd
      by (simp add: odd-minus-even)
    with prems have False
      by (auto simp add: odd-iff-not-even)
    thus ?thesis
      by auto
  qed
 qed

lemma neg-one-even-power: [| x ∈ zEven; 0 ≤ x |] ==> (−1::int) ^(nat x) = 1
proof −
  assume x ∈ zEven and 0 ≤ x
  then have ∃ k. x = 2 ∗ k
    by (auto simp only: zEven-def)
  then show ?thesis
    proof
      fix a
      assume x = 2 ∗ a
      from prems have a: 0 ≤ a
        by arith
      from prems have nat x = nat(2 ∗ a)
        by auto
      also from a have nat (2 ∗ a) = 2 ∗ nat a
        by (auto simp add: nat-mult-distrib)
      finally have (−1::int) ^nat x = (−1) ^(2 ∗ nat a)
        by auto
      also have ... = ((−1::int) ^2) ^ (nat a)
        by (auto simp add: zpower-zpower [THEN sym])
      also have (−1::int) ^2 = 1
        by auto
```

```
    finally show ?thesis
      by auto
  qed
qed

lemma neg-one-odd-power: [| x ∈ zOdd; 0 ≤ x |] ==> (−1::int) ^(nat x) = −1
proof −
  assume x ∈ zOdd and 0 ≤ x
  then have ∃ k. x = 2 * k + 1
    by (auto simp only: zOdd-def)
  then show ?thesis
    proof
      fix a
      assume x = 2 * a + 1
      from prems have a: 0 ≤ a
        by arith
      from prems have nat x = nat(2 * a + 1)
        by auto
      also from a have nat (2 * a + 1) = 2 * nat a + 1
        by (auto simp add: nat-mult-distrib nat-add-distrib)
      finally have (−1::int) ^nat x = (−1) ^(2 * nat a + 1)
        by auto
      also have ... = ((−1::int) ^2) ^ (nat a) * (−1) ^1
        by (auto simp add: zpower-zpower [THEN sym] zpower-zadd-distrib)
      also have (−1::int) ^2 = 1
        by auto
      finally show ?thesis
        by auto
    qed
qed

lemma neg-one-power-parity: [| 0 ≤ x; 0 ≤ y; (x ∈ zEven) = (y ∈ zEven) |] ==>

  (−1::int) ^(nat x) = (−1::int) ^(nat y)
  apply (insert even-odd-disj [of x])
  apply (insert even-odd-disj [of y])
  by (auto simp add: neg-one-even-power neg-one-odd-power)

lemma one-not-neg-one-mod-m: 2 < m ==> ~([1 = −1] (mod m))
  by (auto simp add: zcong-def zdvd-not-zless)

lemma even-div-2-l: [| y ∈ zEven; x < y |] ==> x div 2 < y div 2
  apply (auto simp only: zEven-def)
  proof −
    fix k assume x < 2 * k
```

```
    then have x div 2 < k by (auto simp add: div-prop1)
    also have k = (2 * k) div 2 by auto
    finally show x div 2 < 2 * k div 2 by auto
  qed

lemma even-sum-div-2: [| x ∈ zEven; y ∈ zEven |] ==> (x + y) div 2 = x div 2
+ y div 2
  by (auto simp add: zEven-def, auto simp add: zdiv-zadd1-eq)

lemma even-prod-div-2: [| x ∈ zEven |] ==> (x * y) div 2 = (x div 2) * y
  by (auto simp add: zEven-def)



lemma zprime-zOdd-eq-grt-2: p ∈ zprime ==> (p ∈ zOdd) = (2 < p)
  apply (auto simp add: zOdd-def zprime-def)
  apply (drule-tac x = 2 in allE)
  apply (insert odd-iff-not-even [of p])
by (auto simp add: zOdd-def zEven-def)



lemma neg-one-special: finite A ==>
    ((-1 :: int) ^ card A) * (-1 ^ card A) = 1
  by (induct set: Finites, auto)

lemma neg-one-power: (-1::int) ^n = 1 | (-1::int) ^n = -1
  apply (induct-tac n)
  by auto

lemma neg-one-power-eq-mod-m: [| 2 < m; [(-1::int) ^j = (-1::int) ^k] (mod m)
|]
  ==> ((-1::int) ^j = (-1::int) ^k)
  apply (insert neg-one-power [of j])
  apply (insert neg-one-power [of k])
  by (auto simp add: one-not-neg-one-mod-m zcong-sym)

end
```

18 Library for proof of QR

```
theory QRLib = NatIntLib:
```

Note. This is an old library. Parts of it have been supplanted by other library files.

18.1 Cardinality of explicit finite sets

lemma *finite-surjI*: [| $B \subseteq f$ ‘ A; *finite* A |] ==> *finite* B
by (*simp add*: *finite-subset finite-imageI*)

lemma *bdd-nat-set-l-finite*: *finite* { y::*nat* . $y < x$}
apply (*rule-tac* $N = \{y.\ y < x\}$ **and** $n = x$ **in** *bounded-nat-set-is-finite*)
by *auto*

lemma *bdd-nat-set-le-finite*: *finite* { y::*nat* . $y \leq x$ }
apply (*subgoal-tac* { y::*nat* . $y \leq x$ } = { y::*nat* . $y <$ *Suc* x})
by (*auto simp add*: *bdd-nat-set-l-finite*)

lemma *bdd-int-set-l-finite*: *finite* { x::*int* . $0 \leq x$ & $x < n$}
apply (*subgoal-tac* {(x :: *int*). $0 \leq x$ & $x < n$} ⊆
int ‘ {(x :: *nat*). $x <$ *nat* n})
apply (*erule finite-surjI*)
apply (*auto simp add*: *bdd-nat-set-l-finite image-def*)
apply (*rule-tac* x = *nat* x **in** *exI*, *simp*)
done

lemma *bdd-int-set-le-finite*: *finite* {x::*int*. $0 \leq x$ & $x \leq n$}
apply (*subgoal-tac* {x. $0 \leq x$ & $x \leq n$} = {x. $0 \leq x$ & $x < n + 1$})
apply (*erule ssubst*)
apply (*rule bdd-int-set-l-finite*)
by *auto*

lemma *bdd-int-set-l-l-finite*: *finite* {x::*int*. $0 < x$ & $x < n$}
apply (*subgoal-tac* {x::*int*. $0 < x$ & $x < n$} ⊆ {x::*int*. $0 \leq x$ & $x < n$})
by (*auto simp add*: *bdd-int-set-l-finite finite-subset*)

lemma *bdd-int-set-l-le-finite*: *finite* {x::*int*. $0 < x$ & $x \leq n$}
apply (*subgoal-tac* {x::*int*. $0 < x$ & $x \leq n$} ⊆ {x::*int*. $0 \leq x$ & $x \leq n$})
by (*auto simp add*: *bdd-int-set-le-finite finite-subset*)

lemma *card-bdd-nat-set-l*: *card* {y::*nat* . $y < x$} = x
apply (*induct-tac* x, *force*)
proof −
fix n::*nat*
assume *card* {y. $y < n$} = n
have {y. $y <$ *Suc* n} = *insert* n {y. $y < n$}

```
    by auto
  then have card {y. y < Suc n} = card (insert n {y. y < n})
    by auto
  also have ... = Suc (card {y. y < n})
    apply (rule card-insert-disjoint)
    by (auto simp add: bdd-nat-set-l-finite)
  finally show card {y. y < Suc n} = Suc n
    by (simp add: prems)
qed

lemma card-bdd-nat-set-le: card { y::nat. y ≤ x} = Suc x
apply (subgoal-tac { y::nat. y ≤ x} = { y::nat. y < Suc x})
by (auto simp add: card-bdd-nat-set-l)

lemma card-bdd-int-set-l: 0 ≤ (n::int) ==> card {y. 0 ≤ y & y < n} = nat n
proof −
  fix n::int
  assume 0 ≤ n
  have finite {y. y < nat n}
    by (rule bdd-nat-set-l-finite)
  moreover have inj-on (%y. int y) {y. y < nat n}
    by (auto simp add: inj-on-def)
  ultimately have card (int ' {y. y < nat n}) = card {y. y < nat n}
    by (rule card-image)
  also from prems have int ' {y. y < nat n} = {y. 0 ≤ y & y < n}
    apply (auto simp add: zless-nat-eq-int-zless image-def)
    apply (rule-tac x = nat x in exI)
    by (auto simp add: nat-0-le)
  also have card {y. y < nat n} = nat n
    by (rule card-bdd-nat-set-l)
  finally show card {y. 0 ≤ y & y < n} = nat n .
qed

lemma card-bdd-int-set-le: 0 ≤ (n::int) ==> card {y. 0 ≤ y & y ≤ n} =
  nat n + 1
apply (subgoal-tac {y. 0 ≤ y & y ≤ n} = {y. 0 ≤ y & y < n+1})
apply (insert card-bdd-int-set-l [of n+1])
by (auto simp add: nat-add-distrib)

lemma card-bdd-int-set-l-le: 0 ≤ (n::int) ==>
    card {x. 0 < x & x ≤ n} = nat n
proof −
  fix n::int
  assume 0 ≤ n
  have finite {x. 0 ≤ x & x < n}
```

by (*rule bdd-int-set-l-finite*)
moreover have *inj-on* (%*x*. *x*+*1*) {*x*. *0* ≤ *x* & *x* < *n*}
by (*auto simp add*: *inj-on-def*)
ultimately have *card* ((%*x*. *x*+*1*) ‘ {*x*. *0* ≤ *x* & *x* < *n*}) =
card {*x*. *0* ≤ *x* & *x* < *n*}
by (*rule card-image*)
also from *prems* **have** ... = *nat n*
by (*rule card-bdd-int-set-l*)
also have (%*x*. *x* + *1*) ‘ {*x*. *0* ≤ *x* & *x* < *n*} = {*x*. *0* < *x* & *x*<= *n*}
apply (*auto simp add*: *image-def*)
apply (*rule-tac x* = *x* − *1* **in** *exI*)
by *arith*
finally show *card* {*x*. *0* < *x* & *x* ≤ *n*} = *nat n*.
qed

lemma *card-bdd-int-set-l-l*: *0* < (*n*::*int*) ==>
card {*x*. *0* < *x* & *x* < *n*} = *nat n* − *1*
apply (*subgoal-tac* {*x*. *0* < *x* & *x* < *n*} = {*x*. *0* < *x* & *x* ≤ *n* − *1*})
apply (*insert card-bdd-int-set-l-le* [*of n* − *1*])
by (*auto simp add*: *nat-diff-distrib*)

lemma *int-card-bdd-int-set-l-l*: *0* < *n* ==>
int(*card* {*x*. *0* < *x* & *x* < *n*}) = *n* − *1*
apply (*auto simp add*: *card-bdd-int-set-l-l*)
apply (*subgoal-tac Suc 0* ≤ *nat n*)
apply (*auto simp add*: *zdiff-int* [*THEN sym*])
apply (*subgoal-tac 0* < *nat n*, *arith*)
by (*simp add*: *zero-less-nat-eq*)

lemma *int-card-bdd-int-set-l-le*: *0* ≤ *n* ==>
int(*card* {*x*. *0* < *x* & *x* ≤ *n*}) = *n*
by (*auto simp add*: *card-bdd-int-set-l-le*)

constdefs
ResSet :: *int* => *int set* => *bool*
ResSet m X == ∀ *y1 y2*. (((*y1* ∈ *X*) & (*y2* ∈ *X*) &
[*y1* = *y2*] (*mod m*)) −−> *y1* = *y2*)

StandardRes :: *int* => *int* => *int*
StandardRes m x == *x mod m*

QuadRes :: *int* => *int* => *bool*
QuadRes m x == ∃ *y*. ([(*y* ^ 2) = *x*] (*mod m*))

Legendre :: *int* => *int* => *int*
Legendre a p == (*if* ([*a* = 0] (*mod p*)) *then* 0
else if (*QuadRes p a*) *then* 1
else −1)

SR :: *int* => *int set*
SR p == {*x*. (0 ≤ *x*) & (*x* < *p*)}

SRStar :: *int* => *int set*
SRStar p == {*x*. (0 < *x*) & (*x* < *p*)}

MultInv :: *int* => *int* => *int*
MultInv p x == *x* ^ *nat* (*p* − 2)

18.2 A multiplicative inverse mod p

lemma *MultInv-prop1*: [| 2 < *p*; [*x* = *y*] (*mod p*) |] ==>
[(*MultInv p x*) = (*MultInv p y*)] (*mod p*)
by (*auto simp add*: *MultInv-def zcong-zpower*)

lemma *MultInv-prop2*: [| 2 < *p*; *p* ∈ *zprime*; ~([*x* = 0](*mod p*)) |] ==>
[(*x* ∗ (*MultInv p x*)) = 1] (*mod p*)
proof (*simp add*: *MultInv-def zcong-eq-zdvd-prop*)
assume 2 < *p* **and** *p* ∈ *zprime* **and** ~ *p dvd x*
have *x* ∗ *x* ^ *nat* (*p* − 2) = *x* ^ (*nat* (*p* − 2) + 1)
by *auto*
also from *prems* **have** *nat* (*p* − 2) + 1 = *nat* (*p* − 2 + 1)
by (*simp only*: *nat-add-distrib*, *auto*)
also have *p* − 2 + 1 = *p* − 1 **by** *arith*
finally have [*x* ∗ *x* ^ *nat* (*p* − 2) = *x* ^ *nat* (*p* − 1)] (*mod p*)
by (*rule ssubst*, *auto*)
also from *prems* **have** [*x* ^ *nat* (*p* − 1) = 1] (*mod p*)
by (*auto simp add*: *Little-Fermat*)
finally (*zcong-trans*) **show** [*x* ∗ *x* ^ *nat* (*p* − 2) = 1] (*mod p*).
qed

lemma *MultInv-prop2a*: [| 2 < *p*; *p* ∈ *zprime*; ~([*x* = 0](*mod p*)) |] ==>
[(*MultInv p x*) ∗ *x* = 1] (*mod p*)
by (*auto simp add*: *MultInv-prop2 zmult-ac*)

lemma *aux-1*: *2 < p ==> ((nat p) - 2) = (nat (p - 2))*
by (*simp add*: *nat-diff-distrib*)

lemma *aux-2*: *2 < p ==> 0 < nat (p - 2)*
by *auto*

lemma *MultInv-prop3*: [| *2 < p*; *p ∈ zprime*; ~([*x = 0*](*mod p*)) |] ==>
~([*MultInv p x = 0*](*mod p*))
apply (*auto simp add*: *MultInv-def zcong-eq-zdvd-prop aux-1*)
apply (*drule aux-2*)
apply (*drule zpower-zdvd-prop2*, *auto*)
done

lemma *aux--1*: [| *2 < p*; *p ∈ zprime*; ~([*x = 0*](*mod p*))|] ==>
[(*MultInv p* (*MultInv p x*)) = (*x* * (*MultInv p x*) *
(*MultInv p* (*MultInv p x*)))] (*mod p*)
apply (*drule MultInv-prop2*, *auto*)
apply (*drule-tac k = MultInv p* (*MultInv p x*) **in** *zcong-scalar*, *auto*)
apply (*auto simp add*: *zcong-sym*)
done

lemma *aux--2*: [| *2 < p*; *p ∈ zprime*; ~([*x = 0*](*mod p*))|] ==>
[(*x* * (*MultInv p x*) * (*MultInv p* (*MultInv p x*))) = *x*] (*mod p*)
apply (*frule MultInv-prop3*, *auto*)
apply (*insert MultInv-prop2* [*of p MultInv p x*], *auto*)
apply (*drule MultInv-prop2*, *auto*)
apply (*drule-tac k = x* **in** *zcong-scalar2*, *auto*)
apply (*auto simp add*: *zmult-ac*)
done

lemma *MultInv-prop4*: [| *2 < p*; *p ∈ zprime*; ~([*x = 0*](*mod p*)) |] ==>
[(*MultInv p* (*MultInv p x*)) = *x*] (*mod p*)
apply (*frule aux--1*, *auto*)
apply (*drule aux--2*, *auto*)
apply (*drule zcong-trans*, *auto*)
done

lemma *MultInv-prop5*: [| *2 < p*; *p ∈ zprime*; ~([*x = 0*](*mod p*));
~([*y = 0*](*mod p*)); [(*MultInv p x*) = (*MultInv p y*)] (*mod p*) |] ==>
[*x = y*] (*mod p*)
apply (*drule-tac a = MultInv p x* **and** *b = MultInv p y* **and**
m = p **and** *k = x* **in** *zcong-scalar*)
apply (*insert MultInv-prop2* [*of p x*], *simp*)
apply (*auto simp only*: *zcong-sym* [*of MultInv p x * x*])

```
  apply (auto simp add:  zmult-ac)
  apply (drule zcong-trans, auto)
  apply (drule-tac a = x * MultInv p y and k = y in zcong-scalar, auto)
  apply (insert MultInv-prop2a [of p y], auto simp add: zmult-ac)
  apply (insert zcong-zmult-prop2 [of y * MultInv p y 1 p y x])
  apply (auto simp add: zcong-sym)
done

lemma MultInv-zcong-prop1: [| 2 < p; [j = k] (mod p) |] ==>
    [a * MultInv p j = a * MultInv p k] (mod p)
  by (drule MultInv-prop1, auto simp add: zcong-scalar2)

lemma aux---1: [j = a * MultInv p k] (mod p) ==>
    [j * k = a * MultInv p k * k] (mod p)
  by (auto simp add: zcong-scalar)

lemma aux---2: [|2 < p; p ∈ zprime; ~([k = 0](mod p));
    [j * k = a * MultInv p k * k] (mod p) |] ==> [j * k = a] (mod p)
  apply (insert MultInv-prop2a [of p k] zcong-zmult-prop2
    [of MultInv p k * k 1 p j * k a])
  apply (auto simp add: zmult-ac)
done

lemma aux---3: [j * k = a] (mod p) ==> [(MultInv p j) * j * k =
     (MultInv p j) * a] (mod p)
  by (auto simp add: zmult-assoc zcong-scalar2)

lemma aux---4: [|2 < p; p ∈ zprime; ~([j = 0](mod p));
    [(MultInv p j) * j * k = (MultInv p j) * a] (mod p) |]
      ==> [k = a * (MultInv p j)] (mod p)
  apply (insert MultInv-prop2a [of p j] zcong-zmult-prop1
    [of MultInv p j * j 1 p MultInv p j * a k])
  apply (auto simp add: zmult-ac zcong-sym)
done

lemma MultInv-zcong-prop2: [| 2 < p; p ∈ zprime; ~([k = 0](mod p));
    ~([j = 0](mod p)); [j = a * MultInv p k] (mod p) |] ==>
    [k = a * MultInv p j] (mod p)
  apply (drule aux---1)
  apply (frule aux---2, auto)
  by (drule aux---3, drule aux---4, auto)

lemma MultInv-zcong-prop3: [| 2 < p; p ∈ zprime; ~([a = 0](mod p));
    ~([k = 0](mod p)); ~([j = 0](mod p));
    [a * MultInv p j = a * MultInv p k] (mod p) |] ==>
```

[$j = k$] ($mod\ p$)
apply (*auto simp add*: *zcong-eq-zdvd-prop* [*of a p*])
apply (*frule zprime-imp-zrelprime*, *auto*)
apply (*insert zcong-cancel2* [*of p a MultInv p j MultInv p k*], *auto*)
apply (*drule MultInv-prop5*, *auto*)
done

18.3 Properties of StandardRes

lemma *StandardRes-prop1*: [x = *StandardRes m x*] ($mod\ m$)
by (*auto simp add*: *StandardRes-def zcong-zmod*)

lemma *StandardRes-prop2*: $0 < m$ ==> (*StandardRes m x1* = *StandardRes m x2*)
= ([*x1* = *x2*] ($mod\ m$))
by (*auto simp add*: *StandardRes-def zcong-zmod-eq*)

lemma *StandardRes-prop3*: (~[$x = 0$] ($mod\ p$)) = (~(*StandardRes p x* = 0))
by (*auto simp add*: *StandardRes-def zcong-def zdvd-iff-zmod-eq-0*)

lemma *StandardRes-prop4*: $2 < m$
==> [*StandardRes m x* * *StandardRes m y* = ($x * y$)] ($mod\ m$)
by (*auto simp add*: *StandardRes-def zcong-zmod-eq*
zmod-zmult-distrib [*of x y m*])

lemma *StandardRes-lbound*: $0 < p$ ==> $0 \leq$ *StandardRes p x*
by (*auto simp add*: *StandardRes-def pos-mod-sign*)

lemma *StandardRes-ubound*: $0 < p$ ==> *StandardRes p x* < p
by (*auto simp add*: *StandardRes-def pos-mod-bound*)

lemma *StandardRes-eq-zcong*:
(*StandardRes m x* = 0) = ([$x = 0$]($mod\ m$))
by (*auto simp add*: *StandardRes-def zcong-eq-zdvd-prop dvd-def*)

18.4 Relations between StandardRes, SRStar, and SR

lemma *SRStar-SR-prop*: $x \in$ *SRStar p* ==> $x \in$ *SR p*
by (*auto simp add*: *SRStar-def SR-def*)

lemma *StandardRes-SR-prop*: $x \in$ *SR p* ==> *StandardRes p x* = x
by (*auto simp add*: *SR-def StandardRes-def mod-pos-pos-trivial*)

lemma *StandardRes-SRStar-prop1*: $2 < p$ ==> (*StandardRes p x* $\in$ *SRStar p*)
= (~[$x = 0$] ($mod\ p$))

apply (*auto simp add*: *StandardRes-prop3 StandardRes-def*
SRStar-def pos-mod-bound)
apply (*subgoal-tac* $0 < p$)
by (*drule-tac* $a = x$ **in** *pos-mod-sign*, *arith*, *simp*)

lemma *StandardRes-SRStar-prop1a*: $x \in SRStar\ p \Longrightarrow \sim([x = 0]\ (mod\ p))$
by (*auto simp add*: *SRStar-def zcong-def zdvd-not-zless*)

lemma *StandardRes-SRStar-prop2*: [| $2 < p$; $p \in zprime$; $x \in SRStar\ p$ |]
$\Longrightarrow StandardRes\ p\ (MultInv\ p\ x) \in SRStar\ p$
apply (*frule-tac* $x = (MultInv\ p\ x)$ **in** *StandardRes-SRStar-prop1*, *simp*)
apply (*rule MultInv-prop3*)
apply (*auto simp add*: *SRStar-def zcong-def zdvd-not-zless*)
done

lemma *StandardRes-SRStar-prop3*: $x \in SRStar\ p \Longrightarrow StandardRes\ p\ x = x$
by (*auto simp add*: *SRStar-SR-prop StandardRes-SR-prop*)

lemma *StandardRes-SRStar-prop4*: [| $p \in zprime$; $2 < p$; $x \in SRStar\ p$ |]
$\Longrightarrow StandardRes\ p\ x \in SRStar\ p$
by (*frule StandardRes-SRStar-prop3*, *auto*)

lemma *SRStar-mult-prop1*: [| $p \in zprime$; $2 < p$; $x \in SRStar\ p$; $y \in SRStar\ p$|]
$\Longrightarrow (StandardRes\ p\ (x * y)):SRStar\ p$
apply (*frule-tac* $x = x$ **in** *StandardRes-SRStar-prop4*, *auto*)
apply (*frule-tac* $x = y$ **in** *StandardRes-SRStar-prop4*, *auto*)
apply (*auto simp add*: *StandardRes-SRStar-prop1 zcong-zmult-prop3*)
done

lemma *SRStar-mult-prop2*: [| $p \in zprime$; $2 < p$; $\sim([a = 0](mod\ p))$;
$x \in SRStar\ p$ |]
$\Longrightarrow StandardRes\ p\ (a * MultInv\ p\ x) \in SRStar\ p$
apply (*frule-tac* $x = x$ **in** *StandardRes-SRStar-prop2*, *auto*)
apply (*frule-tac* $x = MultInv\ p\ x$ **in** *StandardRes-SRStar-prop1*)
apply (*auto simp add*: *StandardRes-SRStar-prop1 zcong-zmult-prop3*)
done

lemma *SRStar-card*: $2 < p \Longrightarrow int(card(SRStar\ p)) = p - 1$
by (*auto simp add*: *SRStar-def int-card-bdd-int-set-l-l*)

lemma *SRStar-finite*: $2 < p \Longrightarrow finite(\ SRStar\ p)$
by (*auto simp add*: *SRStar-def bdd-int-set-l-l-finite*)

18.5 Properties relating ResSets with StandardRes

lemma *aux*: $x\ mod\ m = y\ mod\ m \Longrightarrow [x = y]\ (mod\ m)$
apply (*subgoal-tac* $x = y \Longrightarrow [x = y](mod\ m)$)
apply (*subgoal-tac* $[x\ mod\ m = y\ mod\ m]\ (mod\ m) \Longrightarrow [x = y]\ (mod\ m)$)
apply (*auto simp add*: *zcong-zmod* [*of x y m*])
done

lemma *StandardRes-inj-on-ResSet*: *ResSet m X* ==> (*inj-on* (*StandardRes m*) *X*)
apply (*auto simp add*: *ResSet-def StandardRes-def inj-on-def*)
apply (*drule-tac* $m = m$ **in** *aux*, *auto*)
done

lemma *StandardRes-Sum*: [| *finite X*; $0 < m$ |]
$\Longrightarrow [setsum\ f\ X = setsum\ (StandardRes\ m\ o\ f)\ X](mod\ m)$
apply (*rule-tac F* = *X* **in** *finite-induct*)
apply (*auto intro*!: *zcong-zadd simp add*: *StandardRes-prop1*)
done

lemma *SR-pos*: $0 < m \Longrightarrow (StandardRes\ m\ `\ X) \subseteq \{x.\ 0 \leq x\ \&\ x < m\}$
by (*auto simp add*: *StandardRes-ubound StandardRes-lbound*)

lemma *ResSet-finite*: $0 < m \Longrightarrow ResSet\ m\ X \Longrightarrow finite\ X$
apply (*rule-tac f* = *StandardRes m* **in** *finite-imageD*)
apply (*rule-tac B* = $\{x.\ (0 :: int) \leq x\ \&\ x < m\}$ **in** *finite-subset*)
by (*auto simp add*: *StandardRes-inj-on-ResSet bdd-int-set-l-finite SR-pos*)

lemma *mod-mod-is-mod*: $[x = x\ mod\ m](mod\ m)$
by (*auto simp add*: *zcong-zmod*)

lemma *StandardRes-prod*: [| *finite X*; $0 < m$ |]
$\Longrightarrow [setprod\ f\ X = setprod\ (StandardRes\ m\ o\ f)\ X]\ (mod\ m)$
apply (*rule-tac F* = *X* **in** *finite-induct*)
by (*auto intro*!: *zcong-zmult simp add*: *StandardRes-prop1*)

lemma *ResSet-image*: [| $0 < m$; *ResSet m A*; $\forall x \in A.$
$\forall y \in A.\ ([f\ x = f\ y](mod\ m) \longrightarrow x = y)$ |] ==>
ResSet m (*f* ` *A*)
by (*auto simp add*: *ResSet-def*)

lemma *ResSet-SRStar-prop*: *ResSet p* (*SRStar p*)
by (*auto simp add*: *SRStar-def ResSet-def zcong-zless-imp-eq*)

end

19 Euler's criterion

theory *Euler* = *FiniteLib* + *QRLib* + *EvenOdd2*:

constdefs
MultInvPair :: *int* => *int* => *int* => *int set*
MultInvPair a p j == {*StandardRes p j*, *StandardRes p* (*a* ∗ (*MultInv p j*))}
SetS :: *int* => *int* => *int set set*
SetS a p == ((*MultInvPair a p*) ' (*SRStar p*))

lemma *MultInvPair-prop1a*: [| *p* ∈ *zprime*; 2 < *p*; ~([*a* = 0](*mod p*));
X ∈ (*SetS a p*); *Y* ∈ (*SetS a p*);
~((*X* ∩ *Y*) = {}) |] ==>
X = *Y*
apply (*auto simp add*: *SetS-def*)
apply (*drule StandardRes-SRStar-prop1a*)+ **defer** *1*
apply (*drule StandardRes-SRStar-prop1a*)+
apply (*auto simp add*: *MultInvPair-def StandardRes-prop2 zcong-sym*)
apply (*drule notE*, *rule MultInv-zcong-prop1*, *auto*)
apply (*drule notE*, *rule MultInv-zcong-prop2*, *auto*)
apply (*drule MultInv-zcong-prop2*, *auto*)
apply (*drule MultInv-zcong-prop3*, *auto simp add*: *zcong-sym*)
apply (*drule MultInv-zcong-prop1*, *auto*)
apply (*drule MultInv-zcong-prop2*, *auto*)
apply (*drule MultInv-zcong-prop2*, *auto*)
apply (*drule MultInv-zcong-prop3*, *auto simp add*: *zcong-sym*)
done

lemma *MultInvPair-prop1b*: [| *p* ∈ *zprime*; 2 < *p*; ~([*a* = 0](*mod p*));
X ∈ (*SetS a p*); *Y* ∈ (*SetS a p*);

$X \neq Y$ |] ==>
$X \cap Y = \{\}$
apply (*rule notnotD*)
apply (*rule notI*)
apply (*drule MultInvPair-prop1a*, *auto*)
done

lemma *MultInvPair-prop1c*: [| $p \in$ *zprime*; *2* $<$ p; ~([a = *0*](*mod* p)) |] ==>
$\forall X \in$ *SetS a p*. $\forall Y \in$ *SetS a p*. $X \neq Y$ --> $X \cap Y = \{\}$
by (*auto simp add*: *MultInvPair-prop1b*)

lemma *MultInvPair-prop2*: [| $p \in$ *zprime*; *2* $<$ p; ~([a = *0*](*mod* p)) |] ==>
Union (*SetS a p*) = *SRStar p*
apply (*auto simp add*: *SetS-def MultInvPair-def StandardRes-SRStar-prop4 SRStar-mult-prop2*)
apply (*frule StandardRes-SRStar-prop3*)
apply (*rule bexI*, *auto*)
done

lemma *MultInvPair-distinct*: [| $p \in$ *zprime*; *2* $<$ p; ~([a = *0*] (*mod* p));
~([j = *0*] (*mod* p));
~(*QuadRes p a*) |] ==>
~([j = a $*$ *MultInv p j*] (*mod* p))
apply *auto*
proof −
assume $p \in$ *zprime* **and** *2* $<$ p **and** ~([a = *0*] (*mod* p)) **and**
~([j = *0*] (*mod* p)) **and** ~(*QuadRes p a*)
assume [j = a $*$ *MultInv p j*] (*mod* p)
then have [j $*$ j = (a $*$ *MultInv p j*) $*$ j] (*mod* p)
by (*auto simp add*: *zcong-scalar*)
then have *a*:[j $*$ j = a $*$ (*MultInv p j* $*$ j)] (*mod* p)
by (*auto simp add*: *zmult-ac*)
have [j $*$ j = a] (*mod* p)
proof −
from *prems* **have** *b*: [*MultInv p j* $*$ j = *1*] (*mod* p)
by (*simp add*: *MultInv-prop2a*)
from *b a* **show** *?thesis*
by (*auto simp add*: *zcong-zmult-prop2*)
qed
then have [j^*2* = a] (*mod* p)
apply(*subgoal-tac* *2* = *Suc*(*Suc*(*0*)))
apply (*erule ssubst*)
apply (*auto simp only*: *power-Suc power-0*)
by *auto*
with *prems* **show** *False*

```
  by (simp add: QuadRes-def)
qed

lemma MultInvPair-card-two: [| p ∈ zprime; 2 < p; ~([a = 0] (mod p));
                              ~(QuadRes p a); ~([j = 0] (mod p)) |] ==>
                            card (MultInvPair a p j) = 2
 apply (auto simp add: MultInvPair-def)
 apply (subgoal-tac ~ (StandardRes p j = StandardRes p (a * MultInv p j)))
 apply auto
 apply (simp only: StandardRes-prop2)
 apply (drule MultInvPair-distinct)
by auto




lemma SetS-finite: 2 < p ==> finite (SetS a p)
 by (auto simp add: SetS-def SRStar-finite [of p] finite-imageI)

lemma SetS-elems-finite: ∀ X ∈ SetS a p. finite X
 by (auto simp add: SetS-def MultInvPair-def)

lemma SetS-elems-card: [| p ∈ zprime; 2 < p; ~([a = 0] (mod p));
                       ~(QuadRes p a) |] ==>
                       ∀ X ∈ SetS a p. card X = 2
 apply (auto simp add: SetS-def)
 apply (frule StandardRes-SRStar-prop1a)
 apply (rule MultInvPair-card-two, auto)
done

lemma Union-SetS-finite: 2 < p ==> finite (Union (SetS a p))
 by (auto simp add: SetS-finite SetS-elems-finite
   finite-union-finite-subsets)

lemma card-setsum-aux: [| finite S; ∀ X ∈ S. finite (X::int set);
   ∀ X ∈ S. card X = n |] ==> setsum card S = setsum (%x. n) S
by (induct set: Finites, auto)

lemma SetS-card: [| p ∈ zprime; 2 < p; ~([a = 0] (mod p)); ~(QuadRes p a) |]
==>
                int(card(SetS a p)) = (p - 1) div 2
proof -
```

```
  assume p ∈ zprime and 2 < p and ~([a = 0] (mod p)) and ~(QuadRes p a)
  then have (p - 1) = 2 * int(card(SetS a p))
  proof -
    have p - 1 = int(card(Union (SetS a p)))
      by (auto simp add: prems MultInvPair-prop2 SRStar-card)
    also have ... = int (setsum card (SetS a p))
      by (auto simp add: prems SetS-finite SetS-elems-finite
                         MultInvPair-prop1c [of p a] card-Union-disjoint)
    also have ... = int(setsum (%x.2) (SetS a p))
      apply simp
      apply (rule card-setsum-aux)
      apply (rule SetS-finite)
      apply (rule prems)
      apply (rule SetS-elems-finite)
      apply (rule SetS-elems-card)
      apply (rule prems)+
      done
    also have ... = 2 * int(card( SetS a p))
      by (auto simp add: prems SetS-finite)
    finally show ?thesis .
  qed
  from this show ?thesis
    by auto
qed

lemma SetS-ssetprod-prop: [| p ∈ zprime; 2 < p; ~([a = 0] (mod p));
                             ~(QuadRes p a); x ∈ (SetS a p) |] ==>
                          [ssetprod x = a] (mod p)
  apply (auto simp add: SetS-def MultInvPair-def)
  apply (frule StandardRes-SRStar-prop1a)
  apply (subgoal-tac StandardRes p x ≠ StandardRes p (a * MultInv p x))
  apply (auto simp add: StandardRes-prop2 MultInvPair-distinct)
  apply (frule-tac m = p and x = x and y = (a * MultInv p x) in
    StandardRes-prop4)
  apply (subgoal-tac [x * (a * MultInv p x) = a * (x * MultInv p x)] (mod p))
  apply (drule-tac a = StandardRes p x * StandardRes p (a * MultInv p x) and
                   b = x * (a * MultInv p x) and
                   c = a * (x * MultInv p x) in  zcong-trans, force)
  apply (frule-tac p = p and x = x in MultInv-prop2, auto)
  apply (drule-tac a = x * MultInv p x and b = 1 in zcong-zmult-prop2)
  apply (auto simp add: zmult-ac)
done

lemma aux1: [| 0 < x; (x::int) < a; x ≠ (a - 1) |] ==> x < a - 1
  by arith
```

lemma *aux2*: [| (*a*::*int*) < *c*; *b* < *c* |] ==> (*a* ≤ *b* | *b* ≤ *a*)
by *auto*

lemma *SRStar-d22set-prop* [*rule-format*]: *2* < *p* −−> (*SRStar p*) = {*1*} ∪
(*d22set* (*p* − *1*))
apply (*induct p rule*: *d22set.induct*, *auto*)
apply (*simp add*: *SRStar-def d22set.simps*, *arith*)
apply (*simp add*: *SRStar-def d22set.simps*, *clarify*)
apply (*frule aux1*)
apply (*frule aux2*, *auto*)
apply (*simp-all add*: *SRStar-def*)
apply (*simp add*: *d22set.simps*)
apply (*frule d22set-le*)
apply (*frule d22set-g-1*, *auto*)
done

lemma *ssetprod-setprod-id*: *ssetprod A* = *setprod id A*
by (*auto simp add*: *ssetprod-def setprod-def*)

lemma *ssetprod-disj-sets*: [| *finite* (*A*::*int set set*);
∀ *X* ∈ *A*. *finite X*;
∀ *X* ∈ *A*. ∀ *Y* ∈ *A*. (*X* ≠ *Y* −−> *X* ∩ *Y* = {}) |] ==>
ssetprod (*Union A*) = *setprod* (%*x*. *ssetprod x*) *A*
by (*auto simp add*: *ssetprod-setprod-id setprod-Union-disjoint*)

lemma *Union-SetS-ssetprod-prop1*: [| *p* ∈ *zprime*; *2* < *p*; ~([*a* = *0*] (*mod p*));
~(*QuadRes p a*) |] ==>
[*ssetprod* (*Union* (*SetS a p*)) = *a* ^ *nat* ((*p* − *1*) *div 2*)]
(*mod p*)
proof −
assume *p* ∈ *zprime* **and** *2* < *p* **and** ~([*a* = *0*] (*mod p*)) **and** ~(*QuadRes p a*)
then have [*ssetprod* (*Union* (*SetS a p*)) =
setprod ssetprod (*SetS a p*)] (*mod p*)
by (*auto simp add*: *SetS-finite SetS-elems-finite*
MultInvPair-prop1c ssetprod-disj-sets)
also have [*setprod ssetprod* (*SetS a p*) =
setprod (%*x*. *a*) (*SetS a p*)] (*mod p*)
apply (*rule setprod-same-function-zcong*)
by (*auto simp add*: *prems SetS-ssetprod-prop SetS-finite*)
also (*zcong-trans*) **have** [*setprod* (%*x*. *a*) (*SetS a p*) =
a^(*card* (*SetS a p*))] (*mod p*)
by (*auto simp add*: *prems SetS-finite setprod-constant*)
finally (*zcong-trans*) **show** *?thesis*
apply (*rule zcong-trans*)

```
    apply (subgoal-tac card(SetS a p) = nat((p - 1) div 2), auto)
    apply (subgoal-tac nat(int(card(SetS a p))) = nat((p - 1) div 2), force)
    apply (auto simp add: prems SetS-card)
  done
qed

lemma Union-SetS-ssetprod-prop2: [| p ∈ zprime; 2 < p; ~([a = 0](mod p)) |]
==>
                                  ssetprod (Union (SetS a p)) = zfact (p - 1)
proof -
  assume p ∈ zprime and 2 < p and ~([a = 0](mod p))
  then have ssetprod (Union (SetS a p)) = ssetprod (SRStar p)
    by (auto simp add: MultInvPair-prop2)
  also have ... = ssetprod ({1} ∪ (d22set (p - 1)))
    by (auto simp add: prems SRStar-d22set-prop)
  also have ... = zfact(p - 1)
  proof -
    have ~(1 ∈ d22set (p - 1)) & finite( d22set (p - 1))
     apply (insert prems, auto)
     apply (drule d22set-g-1)
     apply (auto simp add: d22set-fin)
    done
    then have ssetprod({1} ∪ (d22set (p - 1))) = ssetprod (d22set (p - 1))
      by auto
    then show ?thesis
      by (auto simp add: d22set-prod-zfact)
  qed
  finally show ?thesis .
qed

lemma zfact-prop: [| p ∈ zprime; 2 < p; ~([a = 0] (mod p)); ~(QuadRes p a) |]
==>
                  [zfact (p - 1) = a ^ nat ((p - 1) div 2)] (mod p)
  apply (frule Union-SetS-ssetprod-prop1)
  apply (auto simp add: Union-SetS-ssetprod-prop2)
done
```

lemma *Euler-part1*: [| *2* < *p*; *p* ∈ *zprime*; ~([*x* = *0*](*mod p*)); ~(*QuadRes p x*) |] ==> [*x*^(*nat* (((*p*) − *1*) *div 2*)) = −*1*](*mod p*)
apply (*frule zfact-prop*, *auto*)
apply (*frule Wilson-Russ*)
apply (*auto simp add*: *zcong-sym*)
apply (*rule zcong-trans*, *auto*)
done

lemma *aux-1*: *0* < *p* ==> (*a*::*int*) ^ *nat* (*p*) = *a* ∗ *a* ^ (*nat* (*p*) − *1*)
proof −
assume *0* < *p*
then have *a* ^ (*nat p*) = *a* ^ (*1* + (*nat p* − *1*))
by (*auto simp add*: *diff-add-assoc*)
also have ... = (*a* ^ *1*) ∗ *a* ^ (*nat*(*p*) − *1*)
by (*simp only*: *zpower-zadd-distrib*)
also have ... = *a* ∗ *a* ^ (*nat*(*p*) − *1*)
by *auto*
finally show *?thesis* **.**
qed

lemma *aux-2*: [| (*2*::*int*) < *p*; *p* ∈ *zOdd* |] ==> *0* < ((*p* − *1*) *div 2*)
proof −
assume *2* < *p* **and** *p* ∈ *zOdd*
then have (*p* − *1*):*zEven*
by (*auto simp add*: *zEven-def zOdd-def*)
then have *aux-1*: *2* ∗ ((*p* − *1*) *div 2*) = (*p* − *1*)
by (*auto simp add*: *even-div-2-prop2*)
then have *1* < (*p* − *1*)
by *auto*
then have *1* < (*2* ∗ ((*p* − *1*) *div 2*))
by (*auto simp add*: *aux-1*)
then have *0* < (*2* ∗ ((*p* − *1*) *div 2*)) *div 2*
by *auto*
then show *?thesis* **by** *auto*
qed

lemma *Euler-part2*: [| *2* < *p*; *p* ∈ *zprime*; [*a* = *0*] (*mod p*) |] ==> [*0* = *a* ^ *nat*

```
((p − 1) div 2)] (mod p)
  apply (frule zprime-zOdd-eq-grt-2)
  apply (frule aux-2, auto)
  apply (frule-tac a = a in aux-1, auto)
  apply (frule zcong-zmult-prop1, auto)
done
```

```
lemma aux--1: [| ~([x = 0] (mod p)); [y ^ 2 = x] (mod p)|] ==> ~(p dvd y)
  apply (subgoal-tac [| ~([x = 0] (mod p)); [y ^ 2 = x] (mod p)|] ==>
      ~([y ^ 2 = 0] (mod p)))
  apply (auto simp add: zcong-sym [of y^2 x p] intro: zcong-trans)
  apply (auto simp add: zcong-eq-zdvd-prop intro: zpower-zdvd-prop1)
done

lemma aux--2: 2 * nat((p − 1) div 2) =  nat (2 * ((p − 1) div 2))
  by (auto simp add: nat-mult-distrib)

lemma Euler-part3: [| 2 < p; p ∈ zprime; ~([x = 0](mod p));
    QuadRes p x |] ==> [x^(nat (((p) − 1) div 2)) = 1](mod p)
  apply (subgoal-tac p ∈ zOdd)
  apply (auto simp add: QuadRes-def)
  apply (frule aux--1, auto)
  apply (drule-tac z = nat ((p − 1) div 2) in zcong-zpower)
  apply (auto simp add: zpower-zpower)
  apply (rule zcong-trans)
  apply (auto simp add: zcong-sym [of x ^ nat ((p − 1) div 2)])
  apply (simp add: aux--2)
  apply (frule odd-minus-one-even)
  apply (frule even-div-2-prop2)
  apply (auto intro: Little-Fermat simp add: zprime-zOdd-eq-grt-2)
done
```

theorem *Euler-Criterion*: [| *2 < p*; *p ∈ zprime* |] ==> [(*Legendre a p*) = *a^(nat (((p) − 1) div 2))*] (*mod p*)
apply (*auto simp add*: *Legendre-def Euler-part2*)
apply (*frule Euler-part3*, *auto simp add*: *zcong-sym*)
apply (*frule Euler-part1*, *auto simp add*: *zcong-sym*)
done

end

20 Gauss' Lemma

theory *Gauss = Euler*:

locale *GAUSS* =
fixes *p* :: *int*
fixes *a* :: *int*
fixes *A* :: *int set*
fixes *B* :: *int set*
fixes *C* :: *int set*
fixes *D* :: *int set*
fixes *E* :: *int set*
fixes *F* :: *int set*

assumes *p-prime*: *p ∈ zprime*
assumes *p-g-2*: *2 < p*
assumes *p-a-relprime*: ~[*a = 0*](*mod p*)
assumes *a-nonzero*: *0 < a*

defines *A-def*: *A == {(x::int). 0 < x & x ≤ ((p − 1) div 2)}*
defines *B-def*: *B == (%x. x * a) ' A*
defines *C-def*: *C == (StandardRes p) ' B*
defines *D-def*: *D == C ∩ {x. x ≤ ((p − 1) div 2)}*
defines *E-def*: *E == C ∩ {x. ((p − 1) div 2) < x}*
defines *F-def*: *F == (%x. (p − x)) ' E*

20.1 Basic properties of p

lemma (**in** *GAUSS*) *p-odd*: *p ∈ zOdd*
by (*auto simp add*: *p-prime p-g-2 zprime-zOdd-eq-grt-2*)

lemma (**in** *GAUSS*) *p-g-0*: *0 < p*
by (*insert p-g-2*, *auto*)

lemma (**in** *GAUSS*) *int-nat*: *int* (*nat* ((*p* − *1*) *div 2*)) = (*p* − *1*) *div 2*
by (*insert p-g-2*, *auto simp add*: *pos-imp-zdiv-nonneg-iff*)

lemma (**in** *GAUSS*) *p-minus-one-l*: (*p* − *1*) *div 2* < *p*
proof −
have *p* − *1* = (*p* − *1*) *div 1* **by** *auto*
then have (*p* − *1*) *div 2* ≤ *p* − *1*
apply (*rule ssubst*) **back**
apply (*rule zdiv-mono2*)
by (*auto simp add*: *p-g-0*)
then have (*p* − *1*) *div 2* ≤ *p* − *1*
by *auto*
then show *?thesis* **by** *simp*
qed

lemma (**in** *GAUSS*) *p-eq*: *p* = (*2* ∗ (*p* − *1*) *div 2*) + *1*
apply (*insert zdiv-zmult-self2* [*of 2 p* − *1*])
by *auto*

lemma *zodd-imp-zdiv-eq*: *x* ∈ *zOdd* ==> *2* ∗ (*x* − *1*) *div 2* = *2* ∗ ((*x* − *1*) *div 2*)
apply (*frule odd-minus-one-even*)
apply (*simp add*: *zEven-def*)
apply (*subgoal-tac 2* ≠ *0*)
apply (*frule-tac b* = *2* :: *int* **and** *a* = *x* − *1* **in** *zdiv-zmult-self2*)
by (*auto simp add*: *even-div-2-prop2*)

lemma (**in** *GAUSS*) *p-eq2*: *p* = (*2* ∗ ((*p* − *1*) *div 2*)) + *1*
apply (*insert p-eq p-prime p-g-2 zprime-zOdd-eq-grt-2* [*of p*], *auto*)
by (*frule zodd-imp-zdiv-eq*, *auto*)

20.2 Basic Properties of the Gauss Sets

lemma (**in** *GAUSS*) *finite-A*: *finite* (*A*)
apply (*auto simp add*: *A-def*)
thm *bdd-int-set-l-finite*
apply (*subgoal-tac* {*x*. *0* < *x* & *x* ≤ (*p* − *1*) *div 2*} ⊆ {*x*. *0* ≤ *x* & *x* < *1* + (*p* − *1*) *div 2*})
by (*auto simp add*: *bdd-int-set-l-finite finite-subset*)

lemma (**in** *GAUSS*) *finite-B*: *finite* (*B*)
by (*auto simp add*: *B-def finite-A finite-imageI*)

lemma (**in** *GAUSS*) *finite-C*: *finite* (*C*)

by (*auto simp add*: *C-def finite-B finite-imageI*)

lemma (**in** *GAUSS*) *finite-D*: *finite* (*D*)
by (*auto simp add*: *D-def finite-Int finite-C*)

lemma (**in** *GAUSS*) *finite-E*: *finite* (*E*)
by (*auto simp add*: *E-def finite-Int finite-C*)

lemma (**in** *GAUSS*) *finite-F*: *finite* (*F*)
by (*auto simp add*: *F-def finite-E finite-imageI*)

lemma (**in** *GAUSS*) *C-eq*: $C = D \cup E$
by (*auto simp add*: *C-def D-def E-def*)

lemma (**in** *GAUSS*) *A-card-eq*: *card A* = *nat* ((*p* − *1*) *div 2*)
apply (*auto simp add*: *A-def*)
apply (*insert int-nat*)
apply (*erule subst*)
by (*auto simp add*: *card-bdd-int-set-l-le*)

lemma (**in** *GAUSS*) *inj-on-xa-A*: *inj-on* (%*x*. *x* ∗ *a*) *A*
apply (*insert a-nonzero*)
by (*simp add*: *A-def inj-on-def*)

lemma (**in** *GAUSS*) *A-res*: *ResSet p A*
apply (*auto simp add*: *A-def ResSet-def*)
apply (*rule-tac m* = *p* **in** *zcong-less-eq*)
apply (*insert p-g-2*, *auto*)
apply (*subgoal-tac* [*1*−*2*] (*p* − *1*) *div 2* < *p*)
by (*auto*, *auto simp add*: *p-minus-one-l*)

lemma (**in** *GAUSS*) *B-res*: *ResSet p B*
apply (*insert p-g-2 p-a-relprime p-minus-one-l*)
apply (*auto simp add*: *B-def*)
apply (*rule ResSet-image*)
apply (*auto simp add*: *A-res*)
apply (*auto simp add*: *A-def*)
proof −
fix *x* **fix** *y*
assume *a*: [*x* ∗ *a* = *y* ∗ *a*] (*mod p*)
assume *b*: *0* < *x*
assume *c*: *x* ≤ (*p* − *1*) *div 2*
assume *d*: *0* < *y*
assume *e*: *y* ≤ (*p* − *1*) *div 2*
from *a p-a-relprime p-prime a-nonzero zcong-cancel* [*of p a x y*]

```
      have [x = y](mod p)
    by (simp add: zprime-imp-zrelprime zcong-def p-g-0 order-le-less)
  with zcong-less-eq [of x y p] p-minus-one-l
      order-le-less-trans [of x (p - 1) div 2 p]
      order-le-less-trans [of y (p - 1) div 2 p] show x = y
    by (simp add: prems p-minus-one-l p-g-0)
qed

lemma (in GAUSS) SR-B-inj: inj-on (StandardRes p) B
 apply (auto simp add: B-def StandardRes-def inj-on-def A-def prems)
 proof -
  fix x fix y
  assume a: x * a mod p = y * a mod p
  assume b: 0 < x
  assume c: x ≤ (p - 1) div 2
  assume d: 0 < y
  assume e: y ≤ (p - 1) div 2
  assume f: x ≠ y
  from a have [x * a = y * a](mod p)
    by (simp add: zcong-zmod-eq p-g-0)
  with p-a-relprime p-prime a-nonzero zcong-cancel [of p a x y]
     have [x = y](mod p)
    by (simp add: zprime-imp-zrelprime zcong-def p-g-0 order-le-less)
  with zcong-less-eq [of x y p] p-minus-one-l
      order-le-less-trans [of x (p - 1) div 2 p]
      order-le-less-trans [of y (p - 1) div 2 p] have x = y
    by (simp add: prems p-minus-one-l p-g-0)
  then have False
    by (simp add: f)
  then show a = 0
    by simp
qed

lemma (in GAUSS) inj-on-pminusx-E: inj-on (%x. p - x) E
 apply (auto simp add: E-def C-def B-def A-def)
 apply (rule-tac g = %x. -1 * (x - p) in inj-on-inverseI)
by auto

lemma (in GAUSS) A-ncong-p: x ∈ A ==> ~[x = 0](mod p)
 apply (auto simp add: A-def)
 apply (frule-tac m = p in zcong-not-zero)
 apply (insert p-minus-one-l)
by auto

lemma (in GAUSS) A-greater-zero: x ∈ A ==> 0 < x
```

```
  by (auto simp add: A-def)

lemma (in GAUSS) B-ncong-p: x ∈ B ==> ~[x = 0](mod p)
  apply (auto simp add: B-def)
  apply (frule A-ncong-p)
  apply (insert p-a-relprime p-prime a-nonzero)
  apply (frule-tac a = x and b = a in zcong-zprime-prod-zero-contra)
by (auto simp add: A-greater-zero)

lemma (in GAUSS) B-greater-zero: x ∈ B ==> 0 < x
  apply (insert a-nonzero)
by (auto simp add: B-def A-greater-zero mult-pos)

lemma (in GAUSS) C-ncong-p: x ∈ C ==>  ~[x = 0](mod p)
  apply (auto simp add: C-def)
  apply (frule B-ncong-p)
  apply (subgoal-tac [x = StandardRes p x](mod p))
  defer apply (simp add: StandardRes-prop1)
  apply (frule-tac a = x and b = StandardRes p x and c = 0 in zcong-trans)
by auto

lemma (in GAUSS) C-greater-zero: y ∈ C ==> 0 < y
  apply (auto simp add: C-def)
  proof -
    fix x
    assume a: x ∈ B
    from p-g-0 have 0 ≤ StandardRes p x
      by (simp add: StandardRes-lbound)
    moreover have ~[x = 0] (mod p)
      by (simp add: a B-ncong-p)
    then have StandardRes p x ≠ 0
      by (simp add: StandardRes-prop3)
    ultimately show 0 < StandardRes p x
      by (simp add: order-le-less)
qed

lemma (in GAUSS) D-ncong-p: x ∈ D ==> ~[x = 0](mod p)
  by (auto simp add: D-def C-ncong-p)

lemma (in GAUSS) E-ncong-p: x ∈ E ==> ~[x = 0](mod p)
  by (auto simp add: E-def C-ncong-p)

lemma (in GAUSS) F-ncong-p: x ∈ F ==> ~[x = 0](mod p)
  apply (auto simp add: F-def)
  proof -
```

```
    fix x assume a: x ∈ E assume b: [p − x = 0] (mod p)
    from E-ncong-p have ~[x = 0] (mod p)
      by (simp add: a)
    moreover from a have 0 < x
      by (simp add: a E-def C-greater-zero)
    moreover from a have x < p
      by (auto simp add: E-def C-def p-g-0 StandardRes-ubound)
    ultimately have ~[p − x = 0] (mod p)
      by (simp add: zcong-not-zero)
    from this show False by (simp add: b)
qed

lemma (in GAUSS) F-subset: F ⊆ {x. 0 < x & x ≤ ((p − 1) div 2)}
  apply (auto simp add: F-def E-def)
  apply (insert p-g-0)
  apply (frule-tac x = xa in StandardRes-ubound)
  apply (frule-tac x = x in StandardRes-ubound)
  apply (subgoal-tac xa = StandardRes p xa)
  apply (auto simp add: C-def StandardRes-prop2 StandardRes-prop1)
  proof −
    from zodd-imp-zdiv-eq p-prime p-g-2 zprime-zOdd-eq-grt-2 have
       2 ∗ (p − 1) div 2 = 2 ∗ ((p − 1) div 2)
      by simp
    with p-eq2 show  !!x. [| (p − 1) div 2 < StandardRes p x; x ∈ B |]
       ==> p − StandardRes p x ≤ (p − 1) div 2
      by simp
qed

lemma (in GAUSS) D-subset: D ⊆ {x. 0 < x & x ≤ ((p − 1) div 2)}
  by (auto simp add: D-def C-greater-zero)

lemma (in GAUSS) F-eq: F = {x. ∃ y ∈ A. ( x = p − (StandardRes p (y∗a)) &
(p − 1) div 2 < StandardRes p (y∗a))}
  by (auto simp add: F-def E-def D-def C-def B-def A-def)

lemma (in GAUSS) D-eq: D = {x. ∃ y ∈ A. ( x = StandardRes p (y∗a) &
StandardRes p (y∗a) ≤ (p − 1) div 2)}
  by (auto simp add: D-def C-def B-def A-def)

lemma (in GAUSS) D-leq: x ∈ D ==> x ≤ (p − 1) div 2
  by (auto simp add: D-eq)

lemma (in GAUSS) F-ge: x ∈ F ==> x ≤ (p − 1) div 2
  apply (auto simp add: F-eq A-def)
  proof −
```

fix *y*
assume *(p − 1) div 2 < StandardRes p (y * a)*
then have *p − StandardRes p (y * a) < p − ((p − 1) div 2)*
by *arith*
also from *p-eq2* **have** *... = 2 * ((p − 1) div 2) + 1 − ((p − 1) div 2)*
by (*rule subst*, *auto*)
also have *2 * ((p − 1) div 2) + 1 − (p − 1) div 2 = (p − 1) div 2 + 1*
by *arith*
finally show *p − StandardRes p (y * a) ≤ (p − 1) div 2*
by (*insert zless-add1-eq* [*of p − StandardRes p (y * a)*
(p − 1) div 2],*auto*)
qed

lemma (**in** *GAUSS*) *all-A-relprime*: ∀ *x* ∈ *A*. *zgcd*(*x*,*p*) = *1*
apply (*insert p-prime p-minus-one-l*)
by (*auto simp add*: *A-def zless-zprime-imp-zrelprime*)

lemma (**in** *GAUSS*) *A-prod-relprime*: *zgcd*((*setprod id A*),*p*) = *1*
by (*insert all-A-relprime finite-A*, *simp add*: *all-relprime-prod-relprime*)

20.3 Relationships Between Gauss Sets

lemma (**in** *GAUSS*) *B-card-eq-A*: *card B = card A*
apply (*insert finite-A*)
by (*simp add*: *finite-A B-def inj-on-xa-A card-image*)

lemma (**in** *GAUSS*) *B-card-eq*: *card B = nat ((p − 1) div 2)*
by (*auto simp add*: *B-card-eq-A A-card-eq*)

lemma (**in** *GAUSS*) *F-card-eq-E*: *card F = card E*
apply (*insert finite-E*)
by (*simp add*: *F-def inj-on-pminusx-E card-image*)

lemma (**in** *GAUSS*) *C-card-eq-B*: *card C = card B*
apply (*insert finite-B*)
apply (*subgoal-tac inj-on* (*StandardRes p*) *B*)
apply (*simp add*: *B-def C-def card-image*)
apply (*rule StandardRes-inj-on-ResSet*)
by (*simp add*: *B-res*)

lemma (**in** *GAUSS*) *D-E-disj*: *D* ∩ *E* = {}
by (*auto simp add*: *D-def E-def*)

lemma (**in** *GAUSS*) *C-card-eq-D-plus-E*: *card C = card D + card E*
by (*auto simp add*: *C-eq card-Un-disjoint D-E-disj finite-D finite-E*)

lemma (**in** *GAUSS*) *C-prod-eq-D-times-E*: *setprod id E* $*$ *setprod id D* $=$ *setprod id C*
  apply (*insert D-E-disj finite-D finite-E C-eq*)
  apply (*frule setprod-Un-disjoint* [*of D E id*])
by *auto*

lemma (**in** *GAUSS*) *C-B-zcong-prod*: [*setprod id C* $=$ *setprod id B*] (*mod p*)
thm *setprod-same-function-zcong*
  apply (*auto simp add*: *C-def*)
  apply (*insert finite-B SR-B-inj*)
  apply (*frule-tac f* $=$ *StandardRes p* **in** *setprod-reindex-id*)
  apply *force*
  apply (*erule subst*)
  apply (*rule setprod-same-function-zcong*)
by (*auto simp add*: *StandardRes-prop1 zcong-sym p-g-0*)

lemma (**in** *GAUSS*) *F-Un-D-subset*: $(F \cup D) \subseteq A$
  apply (*rule Un-least*)
by (*auto simp add*: *A-def F-subset D-subset*)

lemma *two-eq*: *2* $*$ (*x*::*int*) $=$ *x* $+$ *x*
  by *arith*

lemma (**in** *GAUSS*) *F-D-disj*: $(F \cap D) = \{\}$
  apply (*simp add*: *F-eq D-eq*)
  apply (*auto simp add*: *F-eq D-eq*)
  proof $-$
    fix *y* **fix** *ya*
    assume *p* $-$ *StandardRes p* (*y* $*$ *a*) $=$ *StandardRes p* (*ya* $*$ *a*)
    then have *p* $=$ *StandardRes p* (*y* $*$ *a*) $+$ *StandardRes p* (*ya* $*$ *a*)
      by *arith*
    moreover have *p dvd p*
      by *auto*
    ultimately have *p dvd* (*StandardRes p* (*y* $*$ *a*) $+$ *StandardRes p* (*ya* $*$ *a*))
      by *auto*
    then have *a*: [*StandardRes p* (*y* $*$ *a*) $+$ *StandardRes p* (*ya* $*$ *a*) $=$ *0*] (*mod p*)
      by (*auto simp add*: *zcong-def*)
    have [*y* $*$ *a* $=$ *StandardRes p* (*y* $*$ *a*)] (*mod p*)
      by (*simp only*: *zcong-sym StandardRes-prop1*)
    moreover have [*ya* $*$ *a* $=$ *StandardRes p* (*ya* $*$ *a*)] (*mod p*)
      by (*simp only*: *zcong-sym StandardRes-prop1*)
    ultimately have [*y* $*$ *a* $+$ *ya* $*$ *a* $=$
        StandardRes p (*y* $*$ *a*) $+$ *StandardRes p* (*ya* $*$ *a*)] (*mod p*)
      by (*rule zcong-zadd*)

```
    with a have [y * a + ya * a = 0] (mod p)
      apply (elim zcong-trans)
      by (simp only: zcong-refl)
    also have y * a + ya * a = a * (y + ya)
      by (simp add: zadd-zmult-distrib2 zmult-commute)
    finally have [a * (y + ya) = 0] (mod p).
    with p-prime a-nonzero zcong-zprime-prod-zero [of p a y + ya]
        p-a-relprime
        have a: [y + ya = 0] (mod p)
      by auto
    assume b: y ∈ A and c: ya: A
    with A-def have 0 < y + ya
      by auto
    moreover from b c A-def have y + ya ≤ (p - 1) div 2 + (p - 1) div 2
      by auto
    moreover from b c p-eq2 A-def have y + ya < p
      by auto
    ultimately show False
      apply simp
      apply (frule-tac m = p in zcong-not-zero)
      by (auto simp add: a)
qed

lemma (in GAUSS) F-Un-D-card: card (F ∪ D) = nat ((p - 1) div 2)
  apply (insert F-D-disj finite-F finite-D)
  proof -
    have card (F ∪ D) = card E + card D
      by (auto simp add: finite-F finite-D F-D-disj
                         card-Un-disjoint F-card-eq-E)
    then have card (F ∪ D) = card C
      by (simp add: C-card-eq-D-plus-E)
    from this show card (F ∪ D) = nat ((p - 1) div 2)
      by (simp add: C-card-eq-B B-card-eq)
qed

lemma (in GAUSS) F-Un-D-eq-A: F ∪ D = A
  apply (insert finite-A F-Un-D-subset A-card-eq F-Un-D-card)
by (auto simp add: card-seteq)

lemma (in GAUSS) prod-D-F-eq-prod-A:
    (setprod id D) * (setprod id F) = setprod id A
  apply (insert F-D-disj finite-D finite-F)
  apply (frule setprod-Un-disjoint [of F D id])
by (auto simp add: F-Un-D-eq-A)
```

```
lemma (in GAUSS) prod-F-zcong:
   [setprod id F = ((−1) ^ (card E)) ∗ (setprod id E)] (mod p)
  proof −
    have setprod id F = setprod id (op − p ‘ E)
      by (auto simp add: F-def)
    then have setprod id F = setprod (op − p) E
      apply simp
      apply (insert finite-E inj-on-pminusx-E)
      by (frule-tac f = op − p in setprod-reindex-id, auto)
    then have one:
      [setprod id F = setprod (StandardRes p o (op − p)) E] (mod p)
      apply simp
      apply (insert p-g-0 finite-E)
      by (auto simp add: StandardRes-prod)
    moreover have a: ∀ x ∈ E. [p − x = 0 − x] (mod p)
      apply clarify
      apply (insert zcong-id [of p])
      by (rule-tac a = p and m = p and c = x and d = x in zcong-zdiff, auto)
    moreover have b: ∀ x ∈ E. [StandardRes p (p − x) = p − x](mod p)
      apply clarify
      by (simp add: StandardRes-prop1 zcong-sym)
    moreover have ∀ x ∈ E. [StandardRes p (p − x) = − x](mod p)
      apply clarify
      apply (insert a b)
      by (rule-tac b = p − x in zcong-trans, auto)
    ultimately have c:
      [setprod (StandardRes p o (op − p)) E = setprod (uminus) E](mod p)
      apply simp
      apply (insert finite-E p-g-0)
      by (frule setprod-same-function-zcong [of E StandardRes p o (op − p)
                                                uminus p], auto)
    then have two: [setprod id F = setprod (uminus) E](mod p)
      apply (insert one c)
      by (rule zcong-trans [of setprod id F
                            setprod (StandardRes p o op − p) E p
                            setprod uminus E], auto)
    also have setprod uminus E = (setprod id E) ∗ (−1) ^(card E)
      apply (insert finite-E)
      by (induct set: Finites, auto)
    then have setprod uminus E = (−1) ^ (card E) ∗ (setprod id E)
      by (simp add: zmult-commute)
    with two show ?thesis
      by simp
qed
```

20.4 Gauss' Lemma

lemma (**in** *GAUSS*) *aux*: *setprod id A * −1 ^ card E * a ^ card A * −1 ^ card E = setprod id A * a ^ card A*
by (*auto simp add*: *finite-E neg-one-special*)

theorem (**in** *GAUSS*) *pre-gauss-lemma*:
[*a ^ nat*((*p − 1*) *div 2*) = (*−1*) *^* (*card E*)] (*mod p*)
proof −
have [*setprod id A = setprod id F * setprod id D*](*mod p*)
by (*auto simp add*: *prod-D-F-eq-prod-A zmult-commute*)
then have [*setprod id A* = ((*−1*) ^(*card E*) * *setprod id E*) *
setprod id D] (*mod p*)
apply (*rule zcong-trans*)
by (*auto simp add*: *prod-F-zcong zcong-scalar*)
then have [*setprod id A* = ((*−1*) ^(*card E*) * *setprod id C*)] (*mod p*)
apply (*rule zcong-trans*)
apply (*insert C-prod-eq-D-times-E*, *erule subst*)
by (*subst zmult-assoc*, *auto*)
then have [*setprod id A* = ((*−1*) ^(*card E*) * *setprod id B*)] (*mod p*)
apply (*rule zcong-trans*)
by (*simp add*: *C-B-zcong-prod zcong-scalar2*)
then have [*setprod id A* = ((*−1*) ^(*card E*) *
(*setprod id* ((%*x*. *x * a*) ‘ *A*)))] (*mod p*)
by (*simp add*: *B-def*)
then have [*setprod id A* = ((*−1*) ^(*card E*) * (*setprod* (%*x*. *x * a*) *A*))]
(*mod p*)
apply (*rule zcong-trans*)
by (*simp add*: *finite-A inj-on-xa-A setprod-reindex-id zcong-scalar2*)
moreover have *setprod* (%*x*. *x * a*) *A* =
setprod (%*x*. *a*) *A * setprod id A*
by (*insert finite-A*, *induct set*: *Finites*, *auto*)
ultimately have [*setprod id A* = ((*−1*) ^(*card E*) * (*setprod* (%*x*. *a*) *A * *setprod id A*))] (*mod p*)
by *simp*
then have [*setprod id A* = ((*−1*) ^(*card E*) * *a* ^(*card A*) *
setprod id A)](*mod p*)
apply (*rule zcong-trans*)
by (*simp add*: *zcong-scalar2 zcong-scalar finite-A setprod-constant
zmult-assoc*)
then have *a*: [*setprod id A * (−1)* ^(*card E*) =
((*−1*) ^(*card E*) * *a* ^(*card A*) * *setprod id A * (−1)* ^(*card E*))](*mod p*)
by (*rule zcong-scalar*)
then have [*setprod id A * (−1)* ^(*card E*) = *setprod id A **
(*−1*) ^(*card E*) * *a* ^(*card A*) * (*−1*) ^(*card E*)](*mod p*)

```
    apply (rule zcong-trans)
    by (simp add: a mult-commute mult-left-commute)
  then have [setprod id A * (-1)^(card E) = setprod id A *
      a^(card A)](mod p)
    apply (rule zcong-trans)
    by (simp add: aux)
  with this zcong-cancel2 [of p setprod id A -1 ^ card E a ^ card A]
      p-g-0 A-prod-relprime have [-1 ^ card E = a ^ card A](mod p)
    by (simp add: order-less-imp-le)
  from this show ?thesis
    by (simp add: A-card-eq zcong-sym)
qed

theorem (in GAUSS) gauss-lemma: (Legendre a p) = (-1) ^ (card E)
proof -
  from Euler-Criterion p-prime p-g-2 have
    [(Legendre a p) = a^(nat (((p) - 1) div 2))] (mod p)
    by auto
  moreover note pre-gauss-lemma
  ultimately have [(Legendre a p) = (-1) ^ (card E)] (mod p)
    by (rule zcong-trans)
  moreover from p-a-relprime have (Legendre a p) = 1 | (Legendre a p) = (-1)
    by (auto simp add: Legendre-def)
  moreover have (-1::int) ^ (card E) = 1 | (-1::int) ^ (card E) = -1
    by (rule neg-one-power)
  ultimately show ?thesis
    by (auto simp add: p-g-2 one-not-neg-one-mod-m zcong-sym)
qed

end
```

21 The law of Quadratic reciprocity

```
theory QuadraticReciprocity = Gauss:
```

```
lemma setsum-const-mult: finite A ==> setsum (%x. c * ((f x)::int)) A =
```

```
    c * setsum f A
  apply (induct set: Finites, auto)
  by (auto simp only: zadd-zmult-distrib2)

lemma (in GAUSS) QRLemma1: a * setsum id A =
  p * setsum (%x. ((x * a) div p)) A + setsum id D + setsum id E
proof -
  from finite-A have a * setsum id A = setsum (%x. a * x) A
    by(auto simp add: setsum-const-mult id-def)
  also have setsum (%x. a * x) = setsum (%x. x * a)
    by (auto simp add: zmult-commute)
  also have setsum (%x. x * a) A = setsum id B
    by (auto simp add: B-def setsum-reindex-id finite-A inj-on-xa-A)
  also have ... = setsum (%x. p * (x div p) + StandardRes p x) B
    apply (rule setsum-cong)
    by (auto simp add: finite-B StandardRes-def zmod-zdiv-equality)
  also have ... = setsum (%x. p * (x div p)) B + setsum (StandardRes p) B
    by (rule setsum-addf)
  also have setsum (StandardRes p) B = setsum id C
    by (auto simp add: C-def setsum-reindex-id [THEN sym] finite-B
      SR-B-inj)
  also from C-eq have ... = setsum id (D ∪ E)
    by auto
  also from finite-D finite-E have ... = setsum id D + setsum id E
    apply (rule setsum-Un-disjoint)
    by (auto simp add: D-def E-def)
  also have setsum (%x. p * (x div p)) B =
      setsum ((%x. p * (x div p)) o (%x. (x * a))) A
    by (auto simp add: B-def setsum-reindex finite-A inj-on-xa-A)
  also have ... = setsum (%x. p * ((x * a) div p)) A
    by (auto simp add: o-def)
  also from finite-A have setsum (%x. p * ((x * a) div p)) A =
    p * setsum (%x. ((x * a) div p)) A
    by (auto simp add: setsum-const-mult)
  finally show ?thesis by arith
qed

lemma (in GAUSS) QRLemma2: setsum id A = p * int (card E) - setsum id E
+
  setsum id D
proof -
  from F-Un-D-eq-A have setsum id A = setsum id (D ∪ F)
    by (simp add: Un-commute)
  also from F-D-disj finite-D finite-F have
      ... = setsum id D + setsum id F
```

```
    apply (simp add: Int-commute)
    by (intro setsum-Un-disjoint)
  also from F-def have F = (%x. (p - x)) ` E
    by auto
  also from finite-E inj-on-pminusx-E have setsum id ((%x. (p - x)) ` E) =
      setsum (%x. (p - x)) E
    by (auto simp add: setsum-reindex)
  also from finite-E have setsum (op - p) E = setsum (%x. p) E - setsum id E
    by (auto simp add: setsum-subtractf id-def)
  also from finite-E have setsum (%x. p) E = p * int(card E)
    apply (subst setsum-constant)
    apply (assumption)
    apply (simp add: int-eq-of-nat)
    done
  finally show ?thesis
    by arith
qed

lemma (in GAUSS) QRLemma3: (a - 1) * setsum id A =
    p * (setsum (%x. ((x * a) div p)) A - int(card E)) + 2 * setsum id E
proof -
  have (a - 1) * setsum id A = a * setsum id A - setsum id A
    by (auto simp add: zdiff-zmult-distrib)
  also note QRLemma1
  also from QRLemma2 have p * (∑ x ∈ A. x * a div p) + setsum id D +
     setsum id E - setsum id A =
      p * (∑ x ∈ A. x * a div p) + setsum id D +
      setsum id E - (p * int (card E) - setsum id E + setsum id D)
    by auto
  also have ... = p * (∑ x ∈ A. x * a div p) -
      p * int (card E) + 2 * setsum id E
    by arith
  finally show ?thesis
    by (auto simp only: zdiff-zmult-distrib2)
qed

lemma (in GAUSS) QRLemma4: a ∈ zOdd ==>
    (setsum (%x. ((x * a) div p)) A ∈ zEven) = (int(card E): zEven)
proof -
  assume a-odd: a ∈ zOdd
  from QRLemma3 have a: p * (setsum (%x. ((x * a) div p)) A - int(card E))
=
      (a - 1) * setsum id A - 2 * setsum id E
    by arith
  from a-odd have a - 1 ∈ zEven
```

```
    by (rule odd-minus-one-even)
  hence (a − 1) * setsum id A ∈ zEven
    by (rule even-times-either)
  moreover have 2 * setsum id E ∈ zEven
    by (auto simp add: zEven-def)
  ultimately have (a − 1) * setsum id A − 2 * setsum id E ∈ zEven
    by (rule even-minus-even)
  with a have p * (setsum (%x. ((x * a) div p)) A − int(card E)): zEven
    by simp
  hence p ∈ zEven | (setsum (%x. ((x * a) div p)) A − int(card E)): zEven
    by (rule EvenOdd2.even-product)
  with p-odd have (setsum (%x. ((x * a) div p)) A − int(card E)): zEven
    by (auto simp add: odd-iff-not-even)
  thus ?thesis
    by (auto simp only: even-diff [THEN sym])
qed

lemma (in GAUSS) QRLemma5: a ∈ zOdd ==>
   (−1::int) ^(card E) = (−1::int) ^(nat(setsum (%x. ((x * a) div p)) A))
proof −
  assume a ∈ zOdd
  from QRLemma4 have
    (int(card E): zEven) = (setsum (%x. ((x * a) div p)) A ∈ zEven)..
  moreover have 0 ≤ int(card E)
    by auto
  moreover have 0 ≤ setsum (%x. ((x * a) div p)) A
    proof (intro setsum-nonneg)
      from finite-A show finite A.
      next show ∀ x ∈ A. 0 ≤ x * a div p
      proof
        fix x
        assume x ∈ A
        then have 0 ≤ x
          by (auto simp add: A-def)
        with a-nonzero have 0 ≤ x * a
          by (auto simp add: zero-le-mult-iff)
        with p-g-2 show 0 ≤ x * a div p
          by (auto simp add: pos-imp-zdiv-nonneg-iff)
      qed
    qed
  ultimately have (−1::int) ^nat((int (card E))) =
      (−1) ^nat(((∑ x ∈ A. x * a div p)))
    by (intro neg-one-power-parity, auto)
  also have nat (int(card E)) = card E
    by auto
```

finally show *?thesis*.
qed

lemma *MainQRLemma*: [| $a \in zOdd$; $0 < a$; ~([$a = 0$] (*mod* p));$p \in zprime$; $2 < p$;
$A = \{x.\ 0 < x\ \&\ x \leq (p - 1)\ div\ 2\}$ |] ==>
(*Legendre* $a\ p$) = $(-1::int)\hat{}(nat(setsum\ (\%x.\ ((x * a)\ div\ p))\ A))$
apply (*subst GAUSS.gauss-lemma*)
apply (*auto simp add*: *GAUSS-def*)
apply (*subst GAUSS.QRLemma5*)
by (*auto simp add*: *GAUSS-def*)

locale *QRTEMP* =
fixes p :: *int*
fixes q :: *int*
fixes *P-set* :: *int set*
fixes *Q-set* :: *int set*
fixes *S* :: (*int* * *int*) *set*
fixes *S1* :: (*int* * *int*) *set*
fixes *S2* :: (*int* * *int*) *set*
fixes *f1* :: *int* => (*int* * *int*) *set*
fixes *f2* :: *int* => (*int* * *int*) *set*

assumes *p-prime*: $p \in zprime$
assumes *p-g-2*: $2 < p$
assumes *q-prime*: $q \in zprime$
assumes *q-g-2*: $2 < q$
assumes *p-neq-q*: $p \neq q$

defines *P-set-def*: *P-set* == $\{x.\ 0 < x\ \&\ x \leq ((p - 1)\ div\ 2)\ \}$
defines *Q-set-def*: *Q-set* == $\{x.\ 0 < x\ \&\ x \leq ((q - 1)\ div\ 2)\ \}$
defines *S-def*: *S* == *P-set* <*> *Q-set*
defines *S1-def*: *S1* == $\{\ (x,\ y).\ (x,\ y):S\ \&\ ((p * y) < (q * x))\ \}$
defines *S2-def*: *S2* == $\{\ (x,\ y).\ (x,\ y):S\ \&\ ((q * x) < (p * y))\ \}$
defines *f1-def*: *f1 j* == $\{\ (j1,\ y).\ (j1,\ y):S\ \&\ j1 = j\ \&$
$(y \leq (q * j)\ div\ p)\ \}$
defines *f2-def*: *f2 j* == $\{\ (x,\ j1).\ (x,\ j1):S\ \&\ j1 = j\ \&$
$(x \leq (p * j)\ div\ q)\ \}$

lemma (**in** *QRTEMP*) *p-fact*: $0 < (p - 1)\ div\ 2$
proof −
from *prems* **have** $2 < p$ **by** (*simp add*: *QRTEMP-def*)
then have $2 \leq p - 1$ **by** *arith*
then have $2\ div\ 2 \leq (p - 1)\ div\ 2$ **by** (*rule zdiv-mono1*, *auto*)
then show *?thesis* **by** *auto*
qed

lemma (**in** *QRTEMP*) *q-fact*: $0 < (q - 1)\ div\ 2$
proof −
from *prems* **have** $2 < q$ **by** (*simp add*: *QRTEMP-def*)
then have $2 \leq q - 1$ **by** *arith*
then have $2\ div\ 2 \leq (q - 1)\ div\ 2$ **by** (*rule zdiv-mono1*, *auto*)
then show *?thesis* **by** *auto*
qed

lemma (**in** *QRTEMP*) *pb-neq-qa*: $[|1 \leq b;\ b \leq (q - 1)\ div\ 2\ |] ==>$
$(p * b \neq q * a)$
proof
assume $p * b = q * a$ **and** $1 \leq b$ **and** $b \leq (q - 1)\ div\ 2$
then have $q\ dvd\ (p * b)$ **by** (*auto simp add*: *dvd-def*)
with *q-prime p-g-2* **have** $q\ dvd\ p \mid q\ dvd\ b$
by (*auto simp add*: *zprime-zdvd-zmult*)
moreover have $\sim (q\ dvd\ p)$
proof
assume $q\ dvd\ p$
with *p-prime* **have** $q = 1 \mid q = p$
apply (*auto simp add*: *zprime-def QRTEMP-def*)
apply (*drule-tac* $x = q$ **and** $R = False$ **in** *allE*)
apply (*simp add*: *QRTEMP-def*)
apply (*subgoal-tac* $0 \leq q$, *simp add*: *QRTEMP-def*)
apply (*insert prems*)
by (*auto simp add*: *QRTEMP-def*)
with *q-g-2 p-neq-q* **show** *False* **by** *auto*
qed
ultimately have $q\ dvd\ b$ **by** *auto*
then have $q \leq b$
proof −
assume $q\ dvd\ b$
moreover from *prems* **have** $0 < b$ **by** *auto*
ultimately show *?thesis* **by** (*insert zdvd-bounds* [*of q b*], *auto*)
qed
with *prems* **have** $q \leq (q - 1)\ div\ 2$ **by** *auto*
then have $2 * q \leq 2 * ((q - 1)\ div\ 2)$ **by** *arith*
then have $2 * q \leq q - 1$

```
  proof −
    assume 2 ∗ q ≤ 2 ∗ ((q − 1) div 2)
   with prems have q ∈ zOdd by (auto simp add: QRTEMP-def zprime-zOdd-eq-grt-2)
    with odd-minus-one-even have (q − 1):zEven by auto
    with even-div-2-prop2 have (q − 1) = 2 ∗ ((q − 1) div 2) by auto
    with prems show ?thesis by auto
  qed
  then have p1: q ≤ −1 by arith
  with q-g-2 show False by auto
qed

lemma (in QRTEMP) P-set-finite: finite (P-set)
  by (insert p-fact, auto simp add: P-set-def bdd-int-set-l-le-finite)

lemma (in QRTEMP) Q-set-finite: finite (Q-set)
  by (insert q-fact, auto simp add: Q-set-def bdd-int-set-l-le-finite)

lemma (in QRTEMP) S-finite: finite S
  by (auto simp add: S-def  P-set-finite Q-set-finite)

lemma (in QRTEMP) S1-finite: finite S1
proof −
  have finite S by (auto simp add: S-finite)
  moreover have S1 ⊆ S by (auto simp add: S1-def S-def)
  ultimately show ?thesis by (auto simp add: finite-subset)
qed

lemma (in QRTEMP) S2-finite: finite S2
proof −
  have finite S by (auto simp add: S-finite)
  moreover have S2 ⊆ S by (auto simp add: S2-def S-def)
  ultimately show ?thesis by (auto simp add: finite-subset)
qed

lemma (in QRTEMP) P-set-card: (p − 1) div 2 = int (card (P-set))
  by (insert p-fact, auto simp add: P-set-def card-bdd-int-set-l-le)

lemma (in QRTEMP) Q-set-card: (q − 1) div 2 = int (card (Q-set))
  by (insert q-fact, auto simp add: Q-set-def card-bdd-int-set-l-le)

lemma (in QRTEMP) S-card: ((p − 1) div 2) ∗ ((q − 1) div 2) = int (card(S))
  apply (insert P-set-card Q-set-card P-set-finite Q-set-finite)
  apply (auto simp add: S-def zmult-int setsum-constant)
done
```

lemma (**in** *QRTEMP*) *S1-Int-S2-prop*: $S1 \cap S2 = \{\}$
by (*auto simp add*: *S1-def S2-def*)

lemma (**in** *QRTEMP*) *S1-Union-S2-prop*: $S = S1 \cup S2$
apply (*auto simp add*: *S-def P-set-def Q-set-def S1-def S2-def*)
proof −
fix a **and** b
assume $\sim\ q * a < p * b$ **and** *b1*: $0 < b$ **and** *b2*: $b \leq (q - 1)\ div\ 2$
with *zless-linear* **have** $(p * b < q * a) \mid (p * b = q * a)$ **by** *auto*
moreover from *pb-neq-qa b1 b2* **have** $(p * b \neq q * a)$ **by** *auto*
ultimately show $p * b < q * a$ **by** *auto*
qed

lemma (**in** *QRTEMP*) *card-sum-S1-S2*: $((p - 1)\ div\ 2) * ((q - 1)\ div\ 2) = int(card(S1)) + int(card(S2))$
proof−
have $((p - 1)\ div\ 2) * ((q - 1)\ div\ 2) = int\ (card(S))$
by (*auto simp add*: *S-card*)
also have ... $= int(\ card(S1) + card(S2))$
apply (*insert S1-finite S2-finite S1-Int-S2-prop S1-Union-S2-prop*)
apply (*drule card-Un-disjoint*, *auto*)
done
also have ... $= int(card(S1)) + int(card(S2))$ **by** *auto*
finally show *?thesis* **.**
qed

lemma (**in** *QRTEMP*) *aux1a*: $[\!|\ 0 < a;\ a \leq (p - 1)\ div\ 2;\ 0 < b;\ b \leq (q - 1)\ div\ 2\ |\!] ==> (p * b < q * a) = (b \leq q * a\ div\ p)$
proof −
assume $0 < a$ **and** $a \leq (p - 1)\ div\ 2$ **and** $0 < b$ **and** $b \leq (q - 1)\ div\ 2$
have $p * b < q * a ==> b \leq q * a\ div\ p$
proof −
assume $p * b < q * a$
then have $p * b \leq q * a$ **by** *auto*
then have $(p * b)\ div\ p \leq (q * a)\ div\ p$
by (*rule zdiv-mono1*, *insert p-g-2*, *auto*)
then show $b \leq (q * a)\ div\ p$
apply (*subgoal-tac* $p \neq 0$)
apply (*frule zdiv-zmult-self2*, *force*)
by (*insert p-g-2*, *auto*)
qed
moreover have $b \leq q * a\ div\ p ==> p * b < q * a$
proof −
assume $b \leq q * a\ div\ p$

```
    then have p * b ≤ p * ((q * a) div p)
      by (insert p-g-2, auto simp add: mult-le-cancel-left)
    also have ... ≤ q * a
      by (rule zdiv-leq-prop, insert p-g-2, auto)
    finally have p * b ≤ q * a .
    then have p * b < q * a | p * b = q * a
      by (simp only: order-le-imp-less-or-eq)
    moreover have p * b ≠ q * a
      by (rule  pb-neq-qa, insert prems, auto)
    ultimately show ?thesis by auto
  qed
  ultimately show ?thesis ..
qed

lemma (in QRTEMP) aux1b: [| 0 < a; a ≤ (p - 1) div 2;
                            0 < b; b ≤ (q - 1) div 2 |] ==>
                         (q * a < p * b) = (a ≤ p * b div q)
proof -
  assume 0 < a and a ≤ (p - 1) div 2 and 0 < b and b ≤ (q - 1) div 2
  have q * a < p * b ==> a ≤ p * b div q
  proof -
    assume q * a < p * b
    then have q * a ≤ p * b by auto
    then have (q * a) div q ≤ (p * b) div q
      by (rule zdiv-mono1, insert q-g-2, auto)
    then show a ≤ (p * b) div q
      apply (subgoal-tac q ≠ 0)
      apply (frule zdiv-zmult-self2, force)
      by (insert q-g-2, auto)
  qed
  moreover have a ≤ p * b div q ==> q * a < p * b
  proof -
    assume a ≤ p * b div q
    then have q * a ≤ q * ((p * b) div q)
      by (insert q-g-2, auto simp add: mult-le-cancel-left)
    also have ... ≤ p * b
      by (rule zdiv-leq-prop, insert q-g-2, auto)
    finally have q * a ≤ p * b .
    then have q * a < p * b | q * a = p * b
      by (simp only: order-le-imp-less-or-eq)
    moreover have p * b ≠ q * a
      by (rule  pb-neq-qa, insert prems, auto)
    ultimately show ?thesis by auto
  qed
  ultimately show ?thesis ..
```

qed

lemma *aux2*: [| $p \in zprime$; $q \in zprime$; $2 < p$; $2 < q$ |] ==>
$(q * ((p - 1)\ div\ 2))\ div\ p \leq (q - 1)\ div\ 2$
proof−
assume $p \in zprime$ **and** $q \in zprime$ **and** $2 < p$ **and** $2 < q$

then have $p \in zOdd\ \&\ q \in zOdd$
by (*auto simp add*: *zprime-zOdd-eq-grt-2*)
then have *even1*: $(p - 1)$:*zEven* & $(q - 1)$:*zEven*
by (*auto simp add*: *odd-minus-one-even*)
then have *even2*: $(2 * p)$:*zEven* & $((q - 1) * p)$:*zEven*
by (*auto simp add*: *zEven-def*)
then have *even3*: $(((q - 1) * p) + (2 * p))$:*zEven*
by (*auto simp*: *EvenOdd2.even-plus-even*)

from *prems* **have** $q * (p - 1) < ((q - 1) * p) + (2 * p)$
by (*auto simp add*: *int-distrib*)
then have $((p - 1) * q)\ div\ 2 < (((q - 1) * p) + (2 * p))\ div\ 2$
apply (*rule-tac* $x = ((p - 1) * q)$ **in** *even-div-2-l*)
by (*auto simp add*: *even3*, *auto simp add*: *zmult-ac*)
also have $((p - 1) * q)\ div\ 2 = q * ((p - 1)\ div\ 2)$
by (*auto simp add*: *even1 even-prod-div-2*)
also have $(((q - 1) * p) + (2 * p))\ div\ 2 = (((q - 1)\ div\ 2) * p) + p$
by (*auto simp add*: *even1 even2 even-prod-div-2 even-sum-div-2*)
finally show *?thesis*
apply (*rule-tac* $x = q * ((p - 1)\ div\ 2)$ **and**
$y = (q - 1)\ div\ 2$ **in** *div-prop2*)
by (*insert prems*, *auto*)
qed

lemma (**in** *QRTEMP*) *aux3a*: $\forall j \in P\text{-}set.\ int\ (card\ (f1\ j)) =$
$(q * j)\ div\ p$
proof
fix *j*
assume *j-fact*: $j \in P\text{-}set$
have *int* (*card* (*f1 j*)) = *int* (*card* {*y*. $y \in Q\text{-}set$ &
$y \leq (q * j)\ div\ p$})
proof −
have *finite* (*f1 j*)
proof −
have (*f1 j*) $\subseteq S$ **by** (*auto simp add*: *f1-def*)
with *S-finite* **show** *?thesis* **by** (*auto simp add*: *finite-subset*)
qed
moreover have *inj-on* (%(*x*,*y*). *y*) (*f1 j*)

```
    by (auto simp add: f1-def inj-on-def)
  ultimately have card ((%(x,y). y) ` (f1 j)) = card (f1 j)
    by (auto simp add: f1-def card-image)
  moreover have ((%(x,y). y) ` (f1 j)) = {y. y ∈ Q-set &
    y ≤ (q * j) div p}
    by (insert prems, auto simp add: f1-def S-def Q-set-def P-set-def
      image-def)
  ultimately show ?thesis by (auto simp add: f1-def)
 qed
 also have ... = int (card {y. 0 < y & y ≤ (q * j) div p})
 proof −
  have {y. y ∈ Q-set & y ≤ (q * j) div p} =
      {y. 0 < y & y ≤ (q * j) div p}
    apply (auto simp add: Q-set-def)
    proof −
      fix x
      assume 0 < x and x ≤ q * j div p
      with j-fact P-set-def have j ≤ (p − 1) div 2 by auto
      with q-g-2 have q * j ≤ q * ((p − 1) div 2)
        by (auto simp add: mult-le-cancel-left)
      with p-g-2 have q * j div p ≤ q * ((p − 1) div 2) div p
        by (auto simp add: zdiv-mono1)
      also from prems have ... ≤ (q − 1) div 2
        apply simp apply (insert aux2) by (simp add: QRTEMP-def)
      finally show x ≤ (q − 1) div 2 by (insert prems, auto)
    qed
  then show ?thesis by auto
 qed
 also have ... = (q * j) div p
 proof −
  from j-fact P-set-def have 0 ≤ j by auto
  with q-g-2 have q * 0 ≤ q * j
    by (auto simp only: mult-left-mono)
  then have 0 ≤ q * j by auto
  then have 0 div p ≤ (q * j) div p
    apply (rule-tac a = 0 in zdiv-mono1)
    by (insert p-g-2, auto)
  also have 0 div p = 0 by auto
  finally show ?thesis by (auto simp add: card-bdd-int-set-l-le)
 qed
 finally show int (card (f1 j)) = q * j div p .
qed

lemma (in QRTEMP) aux3b: ∀ j ∈ Q-set. int (card (f2 j)) = (p * j) div q
proof
```

fix *j*
assume *j-fact*: $j \in Q\text{-}set$
have *int* (*card* (*f2 j*)) = *int* (*card* {*y*. $y \in P\text{-}set$ & $y \leq (p * j)$ *div q*})
proof −
  have *finite* (*f2 j*)
  proof −
    have (*f2 j*) ⊆ *S* **by** (*auto simp add*: *f2-def*)
    with *S-finite* **show** *?thesis* **by** (*auto simp add*: *finite-subset*)
  qed
  moreover have *inj-on* (%(*x*,*y*). *x*) (*f2 j*)
    by (*auto simp add*: *f2-def inj-on-def*)
  ultimately have *card* ((%(*x*,*y*). *x*) ‘ (*f2 j*)) = *card* (*f2 j*)
    by (*auto simp add*: *f2-def card-image*)
  moreover have ((%(*x*,*y*). *x*) ‘ (*f2 j*)) = {*y*. $y \in P\text{-}set$ & $y \leq (p * j)$ *div q*}
    by (*insert prems*, *auto simp add*: *f2-def S-def Q-set-def*
      P-set-def image-def)
  ultimately show *?thesis* **by** (*auto simp add*: *f2-def*)
qed
also have ... = *int* (*card* {*y*. 0 < *y* & $y \leq (p * j)$ *div q*})
proof −
  have {*y*. $y \in P\text{-}set$ & $y \leq (p * j)$ *div q*} =
      {*y*. 0 < *y* & $y \leq (p * j)$ *div q*}
    apply (*auto simp add*: *P-set-def*)
    proof −
      fix *x*
      assume 0 < *x* **and** $x \leq p * j$ *div q*
      with *j-fact Q-set-def* **have** $j \leq (q - 1)$ *div* 2 **by** *auto*
      with *p-g-2* **have** $p * j \leq p * ((q - 1)$ *div* 2)
        by (*auto simp add*: *mult-le-cancel-left*)
      with *q-g-2* **have** $p * j$ *div q* $\leq p * ((q - 1)$ *div* 2) *div q*
        by (*auto simp add*: *zdiv-mono1*)
      also from *prems* **have** ... $\leq (p - 1)$ *div* 2
        by (*auto simp add*: *aux2 QRTEMP-def*)
      finally show $x \leq (p - 1)$ *div* 2 **by** (*insert prems*, *auto*)
    qed
  then show *?thesis* **by** *auto*
qed
also have ... = $(p * j)$ *div q*
proof −
  from *j-fact Q-set-def* **have** 0 ≤ *j* **by** *auto*
  with *p-g-2* **have** $p * 0 \leq p * j$ **by** (*auto simp only*: *mult-left-mono*)
  then have 0 ≤ $p * j$ **by** *auto*
  then have 0 *div q* ≤ $(p * j)$ *div q*
    apply (*rule-tac a* = 0 **in** *zdiv-mono1*)
    by (*insert q-g-2*, *auto*)

```
    also have 0 div q = 0 by auto
    finally show ?thesis by (auto simp add: card-bdd-int-set-l-le)
  qed
  finally show int (card (f2 j)) = p * j div q .
qed

lemma (in QRTEMP) S1-card: int (card(S1)) = setsum (%j. (q * j) div p) P-set
proof -
  have ∀ x ∈ P-set. finite (f1 x)
  proof
    fix x
    have f1 x ⊆ S by (auto simp add: f1-def)
    with S-finite show finite (f1 x) by (auto simp add: finite-subset)
  qed
  moreover have (∀ x ∈ P-set. ∀ y ∈ P-set.
      x ≠ y --> (f1 x) ∩ (f1 y) = {})
    by (auto simp add: f1-def)
  moreover note P-set-finite
  ultimately have int(card (UNION P-set f1)) =
      setsum (%x. int(card (f1 x))) P-set
    apply (auto simp add: card-UN-disjoint int-setsum o-def)
    done
  moreover have S1 = UNION P-set f1
    by (auto simp add: f1-def S-def S1-def S2-def P-set-def Q-set-def aux1a)
  ultimately have int(card (S1)) = setsum (%j. int(card (f1 j))) P-set
    by auto
  also have ... = setsum (%j. q * j div p) P-set
  proof -
    note aux3a
    with  P-set-finite show ?thesis
      apply (intro setsum-cong)
      apply auto
      done
  qed
  finally show ?thesis .
qed

lemma (in QRTEMP) S2-card: int (card(S2)) = setsum (%j. (p * j) div q) Q-set
proof -
  have ∀ x ∈ Q-set. finite (f2 x)
  proof
    fix x
    have f2 x ⊆ S by (auto simp add: f2-def)
    with S-finite show finite (f2 x) by (auto simp add: finite-subset)
  qed
```

```
  moreover have (∀ x ∈ Q-set. ∀ y ∈ Q-set.
      x ≠ y −−> (f2 x) ∩ (f2 y) = {})
    by (auto simp add: f2-def)
  moreover note Q-set-finite
  ultimately have int(card (UNION Q-set f2)) =
      setsum (%x. int(card (f2 x))) Q-set
      apply (auto simp add: card-UN-disjoint int-setsum o-def)
      done
  moreover have S2 = UNION Q-set f2
    by (auto simp add: f2-def S-def S1-def S2-def P-set-def Q-set-def aux1b)
  ultimately have int(card (S2)) = setsum (%j. int(card (f2 j))) Q-set
    by auto
  also have ... = setsum (%j. p * j div q) Q-set
  proof −
    note aux3b
    with Q-set-finite show ?thesis
      by (intro setsum-cong, auto)
  qed
  finally show ?thesis .
qed

lemma (in QRTEMP) S1-carda: int (card(S1)) =
    setsum (%j. (j * q) div p) P-set
  by (auto simp add: S1-card zmult-ac)

lemma (in QRTEMP) S2-carda: int (card(S2)) =
    setsum (%j. (j * p) div q) Q-set
  by (auto simp add: S2-card zmult-ac)

lemma (in QRTEMP) pq-sum-prop: (setsum (%j. (j * p) div q) Q-set) +
    (setsum (%j. (j * q) div p) P-set) = ((p − 1) div 2) * ((q − 1) div 2)
proof −
  have (setsum (%j. (j * p) div q) Q-set) +
      (setsum (%j. (j * q) div p) P-set) = int (card S2) + int (card S1)
    by (auto simp add: S1-carda S2-carda)
  also have ... = int (card S1) + int (card S2)
    by auto
  also have ... = ((p − 1) div 2) * ((q − 1) div 2)
    by (auto simp add: card-sum-S1-S2)
  finally show ?thesis .
qed

lemma pq-prime-neq: [| p ∈ zprime; q ∈ zprime; p ≠ q |] ==> (~[p = 0] (mod
q))
  apply (auto simp add: zcong-eq-zdvd-prop zprime-def)
```

```
  apply (drule-tac x = q in allE)
  apply (drule-tac x = p in allE)
by auto

lemma (in QRTEMP) QR-short: (Legendre p q) * (Legendre q p) =
    (-1::int) ^nat(((p - 1) div 2)*((q - 1) div 2))
proof -
  from prems have ~([p = 0] (mod q))
    by (auto simp add: pq-prime-neq QRTEMP-def)
  with prems have a1: (Legendre p q) = (-1::int) ^
      nat(setsum (%x. ((x * p) div q)) Q-set)
    apply (rule-tac p = q in  MainQRLemma)
    by (auto simp add: zprime-zOdd-eq-grt-2 QRTEMP-def)
  from prems have ~([q = 0] (mod p))
    apply (rule-tac p = q and q = p in pq-prime-neq)
    apply (simp add: QRTEMP-def)+
    by arith
  with prems have a2: (Legendre q p) =
      (-1::int) ^ nat(setsum (%x. ((x * q) div p)) P-set)
    apply (rule-tac p = p in  MainQRLemma)
    by (auto simp add: zprime-zOdd-eq-grt-2 QRTEMP-def)
  from a1 a2 have (Legendre p q) * (Legendre q p) =
      (-1::int) ^ nat(setsum (%x. ((x * p) div q)) Q-set) *
        (-1::int) ^ nat(setsum (%x. ((x * q) div p)) P-set)
    by auto
  also have ... = (-1::int) ^ (nat(setsum (%x. ((x * p) div q)) Q-set) +
                  nat(setsum (%x. ((x * q) div p)) P-set))
    by (auto simp add: zpower-zadd-distrib)
  also have nat(setsum (%x. ((x * p) div q)) Q-set) +
      nat(setsum (%x. ((x * q) div p)) P-set) =
        nat((setsum (%x. ((x * p) div q)) Q-set) +
          (setsum (%x. ((x * q) div p)) P-set))
    apply (rule-tac z1 = setsum (%x. ((x * p) div q)) Q-set in
      nat-add-distrib [THEN sym])
    by (auto simp add: S1-carda [THEN sym] S2-carda [THEN sym])
  also have ... = nat(((p - 1) div 2) * ((q - 1) div 2))
    by (auto simp add: pq-sum-prop)
  finally show ?thesis .
qed

theorem Quadratic-Reciprocity:
     [| p ∈ zOdd; p ∈ zprime; q ∈ zOdd; q ∈ zprime;
        p ≠ q |]
     ==> (Legendre p q) * (Legendre q p) =
         (-1::int) ^nat(((p - 1) div 2)*((q - 1) div 2))
```

by (*auto simp add*: *QRTEMP.QR-short zprime-zOdd-eq-grt-2* [*THEN sym*]
QRTEMP-def)

end

22 Euler's phi function

theory *EulerPhi* = *NatIntLib*:

constdefs
phi :: *int* => *int*
phi n == *int* (*card*({(*x*::*int*). (*1* <= *x*) & (*x* <= *n*) & (*zgcd*(*x*, *n*) = *1*)}))

locale *PHI* =
fixes *n* :: *int*
fixes *S* :: *int set*
fixes *I* :: *int set*
fixes *I2* :: *int set*
fixes *A* :: *int* => *int set*
fixes *B* :: *int* => *int set*
fixes *f* :: *int* => *int*

assumes *n-ge-1*: *1* <= *n*

defines *S-def*: *S* == {*x*. *1* <= *x* & *x* <= *n*}
defines *I-def*: *I* == {*d*. *0* <= *d* & *d dvd n*}
defines *I2-def*: *I2* == {*d*. *1* <= *d* & *d dvd n*}
defines *A-def*: *A d* == {*k*. (*k*:*S*) & *zgcd*(*k*, *n*) = *d*}
defines *B-def*: *B d* == {*l*. (*1*::*int*) <= *l* & *l* <= (*n div d*) & *zgcd*(*l*, (*n div d*))
= *1*}
defines *f-def*: *f d* == *n div d*

lemma (**in** *PHI*) *n-dvd-range*: [| *d dvd n*; *0* <= *d*|] ==> *1* <= *d* & *d* <= *n*
proof −
assume *d dvd n* **and** *0* <= *d*
then have *1* <= *d*
apply (*insert prems n-ge-1*, *auto*)
apply (*subgoal-tac 0* ~= *d*, *auto*)
done
moreover have *d* <= *n*
apply (*insert prems n-ge-1*, *auto*)
apply (*frule zdvd-bounds*, *auto*)
done

ultimately show *?thesis* **by** *auto*
qed

lemma (**in** *PHI*) *n-zgcd-range*: [| *1* <= *x*; *x* <= *n* |] ==> *1* <= *zgcd* (*x*, *n*) & *zgcd* (*x*, *n*) <= *n*
apply (*insert zgcd-zdvd2* [*of x n*])
apply (*rule n-dvd-range*, *auto simp add*: *zgcd-def*)
done

lemma (**in** *PHI*) *zgcd-elem-S*: [| *1* <= *x*; *x* <= *n* |] ==> *zgcd* (*x*, *n*):*S*
by (*auto simp add*: *S-def n-zgcd-range*)

lemma (**in** *PHI*) *S-finite*: *finite S*
apply (*subgoal-tac S* <= {*1*..*n*})
apply (*erule finite-subset*)
apply *simp*
apply (*unfold S-def*)
apply *auto*
done

lemma (**in** *PHI*) *I-sub-S*: *I* <= *S*
by (*auto simp add*: *S-def I-def n-dvd-range*)

lemma (**in** *PHI*) *I-finite*: *finite I*
by (*insert I-sub-S S-finite*, *auto simp add*: *finite-subset*)

lemma (**in** *PHI*) *I-elem-prop*: *d*:*I* ==> *1* <= *d*
apply (*subgoal-tac d* ~= *0*)
apply (*insert n-ge-1*, *auto simp add*: *I-def*)
done

lemma (**in** *PHI*) *I-eq-I2*: *I* = *I2*
by (*auto simp add*: *n-ge-1 I-def I2-def n-dvd-range*)

lemma (**in** *PHI*) *A-sub-S*: *A*(*d*) <= *S*
by (*auto simp add*: *S-def A-def*)

lemma (**in** *PHI*) *A-finite*: *finite*(*A*(*d*))
by (*insert A-sub-S* [*of d*] *S-finite*, *auto simp add*: *finite-subset*)

lemma (**in** *PHI*) *B-sub-S*: *d*:*I* ==> *B*(*d*) <= *S*
proof
fix *x*
assume *d*:*I* **and** *x*:*B*(*d*)
then have *x* <= *n*

```
  proof−
    from prems have x <= n div d by auto
    also have n div d <= n div 1
    proof−
      from PHI-def prems have 1 <= n by auto
      then have 0 <= n by auto
      moreover have (0::int) < 1 by auto
        moreover from prems have 1 <= d by (insert prems, auto simp add:
I-elem-prop)
      ultimately show n div d <= n div 1 by (rule zdiv-mono2)
    qed
    also have n div 1 = n by auto
    finally show x <= n.
  qed
  moreover from prems have 1 <= x by auto
  ultimately show x:S by (auto simp add: S-def)
qed

lemma (in PHI) B-finite: d:I ==> finite(B(d))
  apply (drule B-sub-S, insert S-finite)
  apply (auto simp add: finite-subset)
done


lemma (in PHI) S-eq-UNION-A: S = UNION I A
proof −
  have UNION I A <= S by (auto simp add: S-def A-def)
  moreover have S <= UNION I A
  proof −
    have S <= UNION I2 A
      by (insert n-zgcd-range, auto simp add: S-def A-def I2-def)
    also have I2 = I by (auto simp add: I-eq-I2)
    finally show S <= UNION I A.
  qed
  ultimately show ?thesis by auto
qed


lemma (in PHI) A-disj-prop: ALL d1:I. ALL d2:I. d1 ~= d2 −−> A(d1) Int
A(d2) = {}
  by (auto simp add: A-def)


lemma (in PHI) n-eq-Sum-card-A: n = setsum (%x. int(card (A(x)))) I
proof −
```

```
  have n = int(card S)
    apply (insert n-ge-1, auto simp add: S-def)
    apply (subgoal-tac {x. 1 <= x & x <= n} = {x. 0 < x & x <= n})
    apply (rule ssubst, force)
    apply (subgoal-tac {x. 0 < x & x <= n} = {}0..n})
    apply (erule ssubst)back
    apply simp
    apply force
    apply auto
    done
  also have S = UNION I A by (rule S-eq-UNION-A)
  also have int (card (UNION I A)) = setsum (%x. int (card (A(x)))) I
    apply (subst card-UN-disjoint)
    apply (auto simp add: I-finite A-finite A-disj-prop)
    apply (subst int-setsum)
    apply (simp add: o-def)
    done
  finally show ?thesis .
qed


lemma (in PHI) zgcd-div-prop: d:S ==> (zgcd(k, n) = d) =
  (d dvd k & d dvd n & zgcd(k div d, n div d) = 1)
proof -
  assume d: S
  with S-def have 0 < d by auto
  thus ?thesis
    by (rule zgcd-equiv)
qed


lemma (in PHI) A-inj-prop: d:I ==> inj-on (%(k::int). k div d) (A(d))
proof (auto simp add: inj-on-def)
  fix x and y
  assume d:I and x div d = y div d and
        p1: x:A(d) and p2: y:A(d)
  then have (x div d) * d = (y div d) * d by auto
  also have (x div d) * d = x
  proof -
    have d dvd x by (insert p1, auto simp add: A-def)
    then show (x div d) * d = x by (auto simp add: dvd-def)
  qed
  also have (y div d) * d = y
  proof -
    have d dvd y by (insert p2, auto simp add: A-def)
```

```
    then show (y div d) * d = y by (auto simp add: dvd-def)
  qed
  finally show x = y .
qed


lemma (in PHI) image-A-eq-B: d:I ==> (%(k::int). k div d) ` A(d) = B(d)
proof (auto)
  fix k
  assume d:I and k:A(d)
  then show (k div d):B(d)
  proof −
    from prems have 1 <= k div d
    proof (auto simp add: A-def I-def S-def)
      assume 1 <= k
       then have 0 < k & 0 < zgcd(k,n) & zgcd(k,n) dvd k by (auto simp add:
zgcd-gr-zero1)
      then have 0 < k div zgcd(k,n) by (auto simp add: zdiv-gr-zero)
      thus 1 <= k div zgcd(k,n) by arith
    qed
    moreover from prems have k div d <= n div d
    proof (auto simp add: A-def I-def S-def)
      assume 1 <= k
      then have 0 < zgcd(k,n) by (auto simp add: zgcd-gr-zero1)
      thus k div zgcd (k, n) <= n div zgcd (k, n)
        by (insert prems, auto simp add: zdiv-mono1)
    qed
    moreover from prems have zgcd(k div d, n div d) = 1
    proof (auto simp add: A-def I-def S-def)
      assume 1 <= k and k <= n
      then have p1:zgcd (k, n):S by (auto simp add: zgcd-elem-S)
      thus zgcd (k div zgcd (k, n), n div zgcd (k, n)) = 1
        apply (insert p1)
        apply (frule zgcd-div-prop, auto)
      done
    qed
    ultimately show (k div d):B(d) by (auto simp add: B-def)
  qed
next
  fix y
  assume q: d:I and y:B(d)
  then show y:(%k. k div d) ` A d
  proof (auto simp add: image-def)
    have a1:(y * d):A(d)
    proof−
```

```
    have 1 <= y * d
    proof−
      from prems have 0 < (y * d)
        by (insert I-elem-prop [of d], auto simp add: mult-pos)
      thus 1 <= (y * d) by arith
    qed
    moreover have y * d <= n
    proof−
      from prems have y <= n div d by auto
      with prems have y * d <= (n div d) * d
        by (auto simp add: mult-right-mono)
      also have ... = n
        proof −
          from prems have d dvd n by auto
          thus (n div d) * d = n by (auto simp add: dvd-def)
        qed
      finally show y * d <= n.
    qed
    moreover have zgcd(y * d, n) = d
    proof−
     from prems have p1: d dvd (y * d) & d dvd n & zgcd( (y * d) div d, n div
d) = 1
        by (auto simp add: B-def)
      moreover from prems have p2: d:S by (insert prems I-sub-S, auto)
      ultimately show zgcd(y * d, n) = d
        by (insert p1 p2 zgcd-div-prop [of d y * d], auto)
    qed
    ultimately show (y * d):A(d) by (auto simp add: A-def S-def)
  qed
   moreover have a2: y = (y * d) div d by (insert prems, auto simp add:
dvd-def)
  ultimately show EX x:A d. y = x div d
    apply (insert a1 a2)
    apply (rule bexI, auto)
  done
 qed
qed


lemma (in PHI) card-A-eq-phi: d:I ==> int(card (A(d))) = phi(n div d)
proof−
 assume d:I
 have card(A(d)) = card((%(k::int). k div d) ‘ A(d))
 proof−
  have finite( A(d)) by (auto simp add: A-finite)
```

```
    moreover have inj-on (%(k::int). k div d) (A(d))
      by (rule A-inj-prop, insert prems, auto)
    ultimately show card(A(d)) = card((%(k::int). k div d) ‘ A(d))
      by (auto simp add: card-image)
  qed
  also have (%(k::int). k div d) ‘ A(d) = B(d)
    by (rule image-A-eq-B, insert prems, auto)
  finally have int(card (A(d)) ) = int (card (B(d))) by (auto)
  also have ... = phi (n div d)
    by (auto simp add: B-def phi-def)
  finally show ?thesis .
qed


lemma (in PHI) n-eq-setsum-phi: n = setsum (phi ∘ f) I
proof−
  have n = setsum (%x. int(card (A(x)))) I
    by (auto simp add: n-eq-Sum-card-A)
  also have ... = setsum (%d. phi (n div d)) I
  proof−
    have finite I by (auto simp add: I-finite)
    moreover have ALL x:I. (%x. int(card (A(x)))) x =  (%d. phi (n div d)) x
      by (auto simp add: card-A-eq-phi)
    ultimately show setsum (%x. int(card (A(x)))) I = setsum (%d. phi (n div
d)) I
      apply (intro setsum-cong)
      apply auto
      done
  qed
  also have ... = setsum (phi ∘ f) I
  proof−
    have finite I by (auto simp add: I-finite)
    moreover have ALL x:I. (%d. phi (n div d)) x = (phi ∘ f) x
      by (auto simp add: f-def)
    ultimately show setsum (%d. phi (n div d)) I = setsum (phi ∘ f) I
      by (intro setsum-cong, auto)
  qed
  finally show ?thesis .
qed

lemma (in PHI) I-inj-prop: inj-on f I
proof (auto simp add: inj-on-def f-def I-def)
  fix x and y
  assume x1:x dvd n and x2: 0 <= x and
         y1:y dvd n and y2: 0 <= y and n div x = n div y
```

```
  then have p1: y * (n div x) * x = y * (n div y) * x
    by (auto)
  also have y * (n div x) * x = y * n
    by (insert x1 x2 p1, auto simp add: dvd-def)
  also have y * (n div y) * x = n * x
    by (insert y1 y2 p1, auto simp add: dvd-def)
  finally have n * x = n * y by auto
  thus x = y by (insert n-ge-1, auto)
qed

lemma (in PHI) f-image-I-eq-I: f ‘ I = I
proof (auto)
  fix x
  assume x:I
  thus (f x):I
  proof−
    have 0 <= n div x
    proof−
      have 0 <= n & 0 < x by (insert prems, frule I-elem-prop,
        auto simp add: PHI-def)
      then have 0 div x <= n div x
        apply (clarify)
        apply (rule zdiv-mono1, auto)
      done
      also have 0 div x = 0 by auto
      finally show 0 <= n div x.
    qed
   moreover have (n div x) dvd n by (insert prems, auto simp add: I-def dvd-def)
    ultimately show (f x):I by (auto simp add: f-def I-def)
  qed
next
  fix x
  assume x:I
  thus x : f ‘ I
  proof (auto simp add: image-def f-def)
    have a1: (n div x):I
    proof−
      have 0 <= n div x
      proof−
        have 0 <= n & 0 < x by (insert prems, frule I-elem-prop,
          auto simp add: PHI-def)
        then have 0 div x <= n div x
          apply (clarify)
          apply (rule zdiv-mono1, auto)
        done
```

```
      also have 0 div x = 0 by auto
      finally show 0 <= n div x.
    qed
      moreover have (n div x) dvd n by (insert prems, auto simp add: I-def
dvd-def)
    ultimately show (n div x):I by (auto simp add: I-def)
  qed
  moreover have a2: x = n div (n div x)
  proof -
    from prems and PHI-def have 0 < n by auto
    moreover have 0 < x
      apply (insert prems)
      apply (frule I-elem-prop, auto)
    done
    moreover from prems have x dvd n by auto
    ultimately have n div (n div x) = x by (auto simp add: zdiv-zdiv-prop)
    thus x = n div (n div x) by auto
  qed
  ultimately show EX y:I. x = n div y
    apply (insert a1 a2)
    apply (rule bexI, auto)
  done
 qed
qed


theorem (in PHI) big-prop1: n = setsum (%d. phi(d)) I
proof-
 have n = setsum (phi o f) I by (auto simp add: n-eq-setsum-phi)
 also have ... = setsum (%d. phi(d)) (f ` I)
 proof -
  have finite I by (auto simp add: I-finite)
  moreover have inj-on f I by (auto simp add: I-inj-prop)
  ultimately have setsum (%d. phi(d)) (f ` I) = setsum (phi o f) I
    by (rule setsum-reindex)
  thus setsum (phi o f) I = setsum (%d. phi(d)) (f ` I) by auto
 qed
 also have ... = setsum (%d. phi(d)) I
   by (auto simp add: f-image-I-eq-I)
 finally show ?thesis .
qed

lemma card-gcd-set: [| p:zprime; 0 < k |] ==>
                int(card({x. 1 <= x & x <= p^k & zgcd(x,p^k) ~= 1})) = p^(k
- 1)
```

```
proof−
  assume p:zprime and 0 < k
  then have {x. 1 <= x & x <= p^k & zgcd(x,p^k) ~= 1} =
          {x. 1 <= x & x <= p^k & p dvd x}
    by (auto simp add: gcd-prime-power-iff-zdvd-prop)
  also have ... = {x. EX y.(1 <= y & y <= p^(k − 1) & x = p ∗ y)}
  proof
    show {x. 1 <= x & x <= p ^ k & p dvd x} <=
        {x. EX y. 1 <= y & y <= p ^ (k − 1) & x = p ∗ y}
      proof (auto, rule-tac x = x div p in exI, auto)
        fix x
        assume 1 <= x and x <= p ^ k and p dvd x
        thus 1 <= x div p
        proof −
          have 0 < x & 0 < p & 0 < x div p
            by (insert prems, auto simp add: zprime-def zdiv-gr-zero)
          thus 1 <= x div p by auto
        qed
      next
        fix x
        assume 1 <= x and x <= p ^ k and p dvd x
        thus x div p <= p ^ (k − Suc 0)
        proof −
          have x div p <= (p ^ k) div p
            by (insert prems, auto simp add: zprime-def zdiv-mono1)
          also have (p ^ k) div p = p ^ (k − 1)
            by (insert prems, auto simp add: zpower-minus-one zprime-def)
          finally show x div p <= p ^ (k − Suc 0) by auto
        qed
      next
        fix x
        assume 1 <= x and x <= p ^ k and p dvd x
        thus x = p ∗ (x div p) by (auto simp add: dvd-def)
      qed
    next
    show {x. EX y. 1 <= y & y <= p ^ (k − 1) & x = p ∗ y} <= {x. 1 <= x
& x <= p ^ k & p dvd x}
    proof (auto)
      fix y
      assume 1 <= y and y <= p ^ (k − Suc 0)
      thus 1 <= p ∗ y
      proof −
        have 0 < y & 0 < p by (insert prems, auto simp add: zprime-def)
        then have 0 < p ∗ y by (auto simp add: mult-pos)
        thus 1 <= p ∗ y by auto
```

```
    qed
  next
    fix y
    assume 1 <= y and y <= p ^ (k - Suc 0)
    thus p * y <= p ^ k
    proof -
      have y * p <= (p ^ (k - 1)) * p
        by (insert prems, auto simp add: zprime-def mult-right-mono)
      also have (p ^ (k - 1)) * p = p ^ k
        by (insert prems, auto simp add: zpower-zmult zprime-def)
      finally show p * y <= p ^ k by (auto simp add: zmult-ac)
    qed
  qed
 qed
 finally have card {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1} =
            card {x. EX y. 1 <= y & y <= p ^ (k - 1) & x = p * y} by auto
 also have ... = card {y. 1 <= y & y <= p^(k - 1)}
 proof-
  have finite({y. 1 <= y & y <= p ^ (k - 1)})
  proof-
    have finite({y. 0 < y & y <= p ^ (k - 1)})
      apply (subgoal-tac {y. 0 < y ∧ y ≤ p ^ (k - 1)} =
          {)0..p ^ (k - 1)})
      apply (erule ssubst)
      apply auto
      done
    also have {y. 0 < y & y <= p ^ (k - 1)} =
              {y. 1 <= y & y <= p ^ (k - 1)} by auto
    finally show finite({y. 1 <= y & y <= p ^ (k - 1)}).
  qed
  moreover have inj-on (%y.(p::int)*y) {y. 1 <= y & y <= p ^ (k - 1)}
    by (insert prems, auto simp add: inj-on-def zprime-def)
  ultimately have card ((%y.(p::int)*y) ' {y. 1 <= y & y <= p ^ (k - 1)})
=
                  card {y. 1 <= y & y <= p ^ (k - 1)}
    by (rule-tac A = {y. 1 <= y & y <= p ^ (k - 1)} in card-image)
  also have (%y.(p::int)*y) ' {y. 1 <= y & y <= p ^ (k - 1)} =
            {x. EX y. 1 <= y & y <= p ^ (k - 1) & x = p * y} by (auto)
  finally show card {x. EX y. 1 <= y & y <= p ^ (k - 1) & x = p * y} =
               card {y. 1 <= y & y <= p ^ (k - 1)}.
 qed
 also have {y. 1 <= y & y <= p ^ (k - 1)} = {y. 0 < y & y <= p ^ (k -
1)} by auto
 finally have int (card({x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1})) =
            int (card({y. 0 < y & y <= p ^ (k - 1)}))
```

```
    by auto
  also have ... = p ^ (k - 1)
    apply (subgoal-tac {y. 0 < y ∧ y ≤ p ^ (k - 1)} = {}0.. p^(k - 1)})
    apply (erule ssubst)
    apply (subgoal-tac 0 <= p ^ (k - 1))
    apply simp
    apply (rule zero-le-power)
    apply (rule order-less-imp-le)
    apply (insert prems, auto simp add: zprime-def)
    done
  finally show int (card {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1}) =
p ^ (k - 1).
qed

theorem big-prop2:  [| p : zprime; 0 < k |] ==> phi(p^k) = (p^k) - (p^(k -
1))
proof -
  assume p : zprime and 0 < k
  then have p:0 <= p ^ k
  proof-
    have 0 < p by (insert prems, auto simp add: zprime-def)
    then have 0 < p ^ k by (rule zpower-gr-0)
    thus 0 <= p ^ k by auto
  qed
  then have p^k = int (card {x. 0 < x & x <= p^k})
     apply (subgoal-tac {x. 0 < x & x <= p^k} = {}0..p^k})
     apply auto
     done
  also have {x. 0 < x & x <= p^k} = {x. 1 <= x & x <= p^k} by auto
  also have ... = {x. 1 <= x & x <= p^k & zgcd(x,p^k) = 1} Un
                  {x. 1 <= x & x <= p^k & zgcd(x,p^k) ~= 1} by auto
  finally have p ^ k = int (card ({x. 1 <= x & x <= p^k & zgcd(x,p^k) = 1}
Un
                                  {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1})).
  also have card ({x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) = 1} Un
                  {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1}) =
            card {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) = 1} +
            card {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1}
  proof-
    have finite {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) = 1}
    proof-
      have finite {x. 1 <= x & x <= p ^ k}
      proof-
        from p have finite {x. 0 < x & x <= p ^ k}
          apply (subgoal-tac {x. 0 < x & x <= p ^ k} = {}0..p^k})
```

```
      apply auto
      done
    also have {x. 0 < x & x <= p ^ k} = {x. 1 <= x & x <= p ^ k}
      by (auto)
    finally show finite {x. 1 <= x & x <= p ^ k}.
  qed
  moreover have {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) = 1} <=
                {x. 1 <= x & x <= p ^ k}
    by auto
  ultimately show finite {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) = 1}
    by (auto simp add: finite-subset)
qed
moreover have finite {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1}
proof-
  have finite {x. 1 <= x & x <= p ^ k}
  proof-
    from p have finite {x. 0 < x & x <= p ^ k}
      apply (subgoal-tac {x. 0 < x & x <= p ^ k} = {}0..p^k})
      apply auto
      done
    also have {x. 0 < x & x <= p ^ k} = {x. 1 <= x & x <= p ^ k}
      by (auto)
    finally show finite {x. 1 <= x & x <= p ^ k}.
  qed
  moreover have {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1} <=
                {x. 1 <= x & x <= p ^ k}
    by auto
  ultimately show finite {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1}
    by (auto simp add: finite-subset)
qed
moreover have {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1} Int
              {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) = 1} = {}
  by auto
ultimately show card ({x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) = 1} Un
                      {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1}) =
                card {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) = 1} +
                card {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1}
  by (rule-tac A = {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) = 1}
    in card-Un-disjoint, auto)
qed
also have int (card {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) = 1} +
               card {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1}) =
          int (card {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) = 1}) +
          int (card {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1})
  by (auto)
```

```
  also have int( card {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) ~= 1}) =
      p ^ (k - 1)
    by (insert prems, auto simp add: card-gcd-set)
  also have int( card {x. 1 <= x & x <= p ^ k & zgcd (x, p ^ k) = 1})
      = phi(p^k)
    by (auto simp add: phi-def)
  finally have p ^ k = phi (p ^ k) + p ^ (k - 1).
  thus phi (p ^ k) = p ^ k - p ^ (k - 1)
    by (auto)
qed


locale PHI2 =
  fixes n        :: int
  fixes m        :: int
  fixes f        :: int => (int * int)
  fixes SR       :: int => int set
  fixes ResUnits :: int => int set

  assumes n-ge-1:  1 <= n
  assumes m-ge-1:  1 <= m
  assumes m-n-gcd: zgcd(m, n) = 1

  defines f-def:         f x == (x mod m, x mod n)
  defines SR-def:        SR y == {x. (0 <= x) & (x < y)}
  defines ResUnits-def: ResUnits y == {x. (x:(SR(y))) & zgcd(x,y) = 1}

lemma (in PHI2) SR-m-mod-prop: a:SR(m) ==> a mod m = a
  apply (insert m-ge-1)
  apply (auto simp add: SR-def mod-pos-pos-trivial)
done

lemma (in PHI2) SR-n-mod-prop: a:SR(n) ==> a mod n = a
  apply (insert n-ge-1)
  apply (auto simp add: SR-def mod-pos-pos-trivial)
done

lemma (in PHI2) SR-mn-mod-prop: a:SR(m*n) ==> a mod (m*n) = a
  apply (insert n-ge-1 m-ge-1)
  apply (auto simp add: SR-def mod-pos-pos-trivial)
done

lemma (in PHI2) SR-m-finite: finite(SR(m))
  apply (insert m-ge-1)
  apply (auto simp add: SR-def)
```

apply (*subgoal-tac* $\{x.\ 0 \leq x \wedge x < m\} = \{0..m(\}$)
apply *auto*
done

lemma (**in** *PHI2*) *SR-m-card*: $int(card(SR(m))) = m$
apply (*insert m-ge-1*)
apply (*auto simp add*: *SR-def*)
apply (*subgoal-tac* $\{x.\ 0 \leq x \wedge x < m\} = \{0..m(\}$)
apply *auto*
done

lemma (**in** *PHI2*) *SR-n-finite*: *finite*$(SR(n))$
apply (*insert n-ge-1*)
apply (*unfold SR-def*)
apply (*subgoal-tac* $\{x.\ 0 \leq x \wedge x < n\} = \{0..n(\}$)
apply *auto*
done

lemma (**in** *PHI2*) *SR-n-card*: $int(card(SR(n))) = n$
apply (*insert n-ge-1*)
apply (*auto simp add*: *SR-def*)
apply (*subgoal-tac* $\{x.\ 0 \leq x \wedge x < n\} = \{0..n(\}$)
apply *auto*
done

lemma (**in** *PHI2*) *SR-mn-finite*: *finite*$(SR(m*n))$
apply (*insert m-ge-1 n-ge-1*)
apply (*auto simp add*: *SR-def*)
apply (*subgoal-tac* $\{x.\ 0 \leq x \wedge x < m * n\} = \{0..m*n(\}$)
apply *auto*
done

lemma (**in** *PHI2*) *SR-mn-card*: $int(card(SR(m*n))) = m * n$
apply (*insert m-ge-1 n-ge-1*)
apply (*auto simp add*: *SR-def*)
apply (*subgoal-tac* $\{x.\ 0 \leq x \wedge x < m * n\} = \{0..m*n(\}$)
apply (*erule ssubst*)
apply *auto*
apply (*subgoal-tac* $0 < m*n$)
apply *auto*
apply (*rule mult-pos*)
apply *auto*
done

lemma (**in** *PHI2*) *ResUnits-m-finite*: *finite*$(ResUnits(m))$

proof−
have *ResUnits*(*m*) <= *SR*(*m*) **by** (*auto simp add*: *ResUnits-def*)
with *SR-m-finite* **show** *?thesis* **by** (*auto simp add*: *finite-subset*)
qed

lemma (**in** *PHI2*) *ResUnits-n-finite*: *finite*(*ResUnits*(*n*))
proof−
have *ResUnits*(*n*) <= *SR*(*n*) **by** (*auto simp add*: *ResUnits-def*)
with *SR-n-finite* **show** *?thesis* **by** (*auto simp add*: *finite-subset*)
qed

lemma (**in** *PHI2*) *ResUnits-mn-finite*: *finite*(*ResUnits*(*m* ∗ *n*))
proof−
have *ResUnits*(*m* ∗ *n*) <= *SR*(*m* ∗ *n*) **by** (*auto simp add*: *ResUnits-def*)
with *SR-mn-finite* **show** *?thesis* **by** (*auto simp add*: *finite-subset*)
qed

lemma (**in** *PHI2*) *zgcd-zmult-prop*: (*zgcd*(*m* ∗ *n*, *x*) = *1*) = ((*zgcd*(*m*, *x*) = *1*) & (*zgcd*(*n*, *x*) = *1*))
proof
assume *p1*: *zgcd* (*m* ∗ *n*, *x*) = *1*
have *zgcd*(*n*, *x*) *dvd* *zgcd*(*m* ∗ *n*, *x*) & *zgcd*(*m*, *x*) *dvd* *zgcd*(*m* ∗ *n*, *x*)
apply (*insert zgcd-zdvd-zgcd-zmult* [*of m x n*])
apply (*auto simp add*: *zgcd-zdvd-zgcd-zmult zmult-ac*)
done
thus *zgcd*(*m*, *x*) = *1* & *zgcd*(*n* , *x*) = *1*
apply (*insert p1*, *insert PHI2-def*, *auto*)
apply (*drule-tac n* = *zgcd*(*m*,*x*) **in** *zdvd-bounds*, *auto*)
apply (*subgoal-tac 0* < *m*)
apply (*drule-tac b* = *x* **in** *zgcd-gr-zero1*, *auto*) **defer**
apply (*drule zdvd-bounds*, *auto*)
apply (*subgoal-tac 0* < *n*)
apply (*drule-tac b* = *x* **in** *zgcd-gr-zero1*)
apply (*insert prems*, *auto*)
done
next
assume *zgcd* (*m*, *x*) = *1* & *zgcd* (*n*, *x*) = *1*
thus *zgcd* (*m* ∗ *n*, *x*) = *1*
by (*auto simp add*: *zgcd-zgcd-zmult*)
qed

lemma (**in** *PHI2*) *specialized-chinese*: *ALL a*. *ALL b*. *EX x*. ([*x* = *a*] (*mod m*) & [*x* = *b*] (*mod n*))
proof (*auto*)
fix *a b*

have *0 < n* **by** (*insert prems*, *auto simp add*: *PHI2-def*)
with *m-n-gcd* **obtain** *u* **where** *p1*:[*m*u = 1*] (*mod n*) **using** *zcong-lineq-ex* **by** *auto*
have *zgcd*(*n*, *m*) = *1* **by** (*insert prems*, *auto simp add*: *zgcd-commute PHI2-def*)
moreover have *0 < m* **by** (*insert prems*, *auto simp add*: *PHI2-def*)
ultimately obtain *v* **where** *p2*:[*n * v = 1*] (*mod m*) **using** *zcong-lineq-ex* **by** *auto*
from *p1* **have** [*m*u*b = 1*b*] (*mod n*) **by** (*rule zcong-scalar*)
then have [*m*u*b = b*] (*mod n*) **by** *auto*
then have [*m*u*b + n*(v*a) = b*] (*mod n*) **by** (*rule zcong-m-scalar-prop*)
then have *p3*: [*m*u*b + n*v*a = b*] (*mod n*) **by** (*auto simp add*: *zmult-ac*)
from *p2* **have** [*n*v*a = 1*a*] (*mod m*) **by** (*rule zcong-scalar*)
then have [*n*v*a = a*] (*mod m*) **by** *auto*
then have [*n*v*a + m*(u*b) = a*] (*mod m*) **by** (*rule zcong-m-scalar-prop*)
then have [*n*v*a + m*u*b = a*] (*mod m*) **by** (*auto simp add*: *zmult-ac*)
then have *p4*: [*m*u*b + n*v*a = a*] (*mod m*) **by** (*auto simp add*: *zadd-ac*)
have [*m*u*b + n*v*a = a*] (*mod m*) & [*m*u*b + n*v*a = b*] (*mod n*)
by (*insert p3 p4*, *auto*)
thus *EX x*. [*x = a*] (*mod m*) & [*x = b*] (*mod n*) **by** (*auto*)
qed

lemma (**in** *PHI2*) *SR-inj-on-f*: *inj-on f* (*SR*(*m * n*))
apply (*auto simp add*: *inj-on-def f-def*)
proof−
fix *x y*
assume *x* : *SR* (*m * n*) **and** *y* : *SR* (*m * n*) **and**
x mod m = y mod m **and** *x mod n = y mod n*
have *p1*: *0 < m* & *0 < n* **by** (*insert m-ge-1 n-ge-1*, *auto*)
then have [*x = y*] (*mod m*) & [*x = y*] (*mod n*)
by (*insert prems*, *auto simp add*: *zcong-zmod-eq*)
then have *p2*: [*x = y*] (*mod m * n*)
apply (*auto*)
apply (*insert prems*)
apply (*rule zcong-zgcd-zmult-zmod*, *auto simp add*: *PHI2-def*)
done
from *p1* **have** *p3*: *0 < m * n* **by** (*auto simp add*: *mult-pos*)
have *p4*: *x mod* (*m*n*) = *y mod* (*m*n*)
by (*insert p2 p3 zcong-zmod-eq* [*of m*n x y*], *auto*)
have *0 <= x* & *0 <= y* & *x <* (*m*n*) & *y <* (*m*n*) **by** (*insert prems*, *auto simp add*: *SR-def*)
thus *x = y*
apply (*auto*)
apply (*insert p1 p4*, *auto*)
apply (*drule mod-pos-pos-trivial*, *auto*)+
done

qed

lemma (**in** *PHI2*) *f-SR-card*: *card* (*f* ‘ (*SR*(*m*∗*n*))) = *card*(*SR*(*m*) <∗> *SR*(*n*))
proof−
have *card* (*f* ‘ (*SR*(*m*∗*n*))) = *card*(*SR*(*m*∗*n*))
apply (*insert SR-mn-finite SR-inj-on-f*)
apply (*auto simp add*: *card-image*)
done
also have ... = *card*(*SR*(*m*) <∗> *SR*(*n*))
proof−
have *int*(*card*(*SR*(*m* ∗ *n*))) = *int*(*card*(*SR*(*m*) <∗> *SR*(*n*)))
apply (*insert SR-m-finite SR-n-finite SR-mn-finite*)
apply (*insert SR-m-card SR-n-card SR-mn-card*)
apply (*auto simp add*: *zmult-int* [*THEN sym*])
done
thus *?thesis* **by** *auto*
qed
finally show *?thesis* **by** *auto*
qed

lemma (**in** *PHI2*) *SR-image-prop*: *f* ‘ *SR*(*m* ∗ *n*) <= *SR*(*m*) <∗> *SR*(*n*)
apply (*insert m-ge-1 n-ge-1*)
apply (*auto simp add*: *SR-def f-def pos-mod-sign pos-mod-bound*)
done

lemma (**in** *PHI2*) *SR-prod-finite*: *finite*(*SR*(*m*) <∗> *SR*(*n*))
by (*auto simp add*: *SR-m-finite SR-n-finite*)

lemma (**in** *PHI2*) *image-SR-prop*: *f* ‘ *SR*(*m* ∗ *n*) = *SR*(*m*) <∗> *SR*(*n*)
apply (*insert SR-prod-finite SR-image-prop f-SR-card*)
apply (*auto simp add*: *card-seteq*)
done

lemma (**in** *PHI2*) *ResUnits-inj-on-f*: *inj-on f* (*ResUnits*(*m* ∗ *n*))
proof−
have (*ResUnits*(*m* ∗ *n*)) <= *SR*(*m* ∗ *n*)
by (*auto simp add*: *ResUnits-def SR-def*)
with *SR-inj-on-f* **show** *?thesis* **by** (*auto simp add*: *subset-inj-on*)
qed

lemma (**in** *PHI2*) *ResUnits-prod-finite*: *finite*(*ResUnits*(*m*) <∗> *ResUnits*(*n*))
by (*auto simp add*: *ResUnits-m-finite ResUnits-n-finite*)

lemma (**in** *PHI2*) *aux1-ResUnits-image-prop*: *f* ‘ *ResUnits*(*m* ∗ *n*) <= *ResUnits*(*m*)
<∗> *ResUnits*(*n*)

```
  apply (auto simp add: image-def f-def)
proof −
  fix x
  assume x : ResUnits (m * n)
  then have zgcd(m*n, x) = 1 by (auto simp add: ResUnits-def zgcd-commute)
  then have zgcd(m,x) = 1 by (auto simp add: zgcd-zmult-prop)
  also have zgcd(m,x) = zgcd(m,x mod m) by (insert zgcd-eq2 [of m x], auto)
  finally have zgcd (m, x mod m) = 1 .
  thus x mod m : ResUnits m
    apply (auto simp add: ResUnits-def zgcd-commute SR-def)
    apply (insert m-ge-1, auto simp add: pos-mod-sign pos-mod-bound)
  done
next
  fix x
  assume x : ResUnits (m * n)
  then have zgcd(m*n, x) = 1 by (auto simp add: ResUnits-def zgcd-commute)
  then have zgcd(n,x) = 1 by (auto simp add: zgcd-zmult-prop)
  also have zgcd(n,x) = zgcd(n,x mod n) by (insert zgcd-eq2 [of n x], auto)
  finally have zgcd (n, x mod n) = 1 .
  thus x mod n : ResUnits n
    apply (auto simp add: ResUnits-def zgcd-commute SR-def)
    apply (insert n-ge-1, auto simp add: pos-mod-sign pos-mod-bound)
  done
qed

lemma (in PHI2) aux2-ResUnits-image-prop: ResUnits(m) <*> ResUnits(n) <=
f ‘ ResUnits(m * n)
  apply (auto simp add: image-def f-def)
  apply (auto simp add: ResUnits-def)
  apply (insert specialized-chinese)
  apply (drule-tac x = a in allE, auto)
  apply (drule-tac x = b in allE, auto)
  apply (rule-tac x = x mod (m*n) in exI)
  apply (auto)
proof−
  fix a b x
  have 0 < m * n by (insert m-ge-1 n-ge-1, auto simp add: mult-pos)
 thus x mod (m*n) : SR(m*n) by (auto simp add: SR-def pos-mod-sign pos-mod-bound)
next
  fix a b x
  assume p1: a : SR m and p2: b : SR n and
         p3: zgcd (a, m) = 1 and p4: zgcd (b, n) = 1 and
         [x = a] (mod m) and [x = b] (mod n)
  moreover have 0 < m & 0 < n by (insert n-ge-1 m-ge-1, auto)
  ultimately have x mod m = a mod m & x mod n = b mod n
```

```
    by (auto simp add: zcong-zmod-eq)
  also have a mod m = a by (insert p1, auto simp add: SR-m-mod-prop)
  also have b mod n = b by (insert p2, auto simp add: SR-n-mod-prop)
  finally have x mod m = a & x mod n = b .
  with p3 p4 have zgcd (x mod m, m) = 1 & zgcd (x mod n, n) = 1 by auto
  with zgcd-ac have  zgcd (m, x mod m) = 1 & zgcd (n, x mod n) = 1 by auto
  with zgcd-eq2 have zgcd (m, x) = 1 & zgcd(n, x) = 1 by auto
  with zgcd-zmult-prop have zgcd(m * n, x) = 1 by auto
  with zgcd-eq2 have zgcd(m*n, x mod (m*n)) = 1 by auto
  thus zgcd (x mod (m * n), m * n) = 1 by (auto simp add: zgcd-ac)
next
  fix a b x
  assume p1: a : SR m and p2: zgcd (a, m) = 1 and p3: [x = a] (mod m)
  then have a = a mod m by (auto simp add: SR-m-mod-prop)
  also have ... = x mod m
  proof-
    have 0 < m by (insert m-ge-1, auto)
    with p3 zcong-sym show ?thesis
      by (auto simp add:  zcong-zmod-eq [THEN sym])
  qed
  also have ... = x mod (m*n) mod m
    by (insert m-ge-1, auto simp add: zmod-zmult-zmod)
  finally show a = x mod (m * n) mod m .
next
  fix a b x
  assume p1: b : SR n and p2: zgcd (b, n) = 1 and p3: [x = b] (mod n)
  then have b = b mod n by (auto simp add: SR-n-mod-prop)
  also have ... = x mod n
  proof-
    have 0 < n by (insert n-ge-1, auto)
    with p3 zcong-sym show ?thesis
      by (auto simp add:  zcong-zmod-eq [THEN sym])
  qed
  also have ... = x mod (m*n) mod n
    by (insert n-ge-1, auto simp add: zmod-zmult-zmod2)
  finally show b = x mod (m * n) mod n .
qed

lemma (in PHI2) ResUnits-image-prop: f ' ResUnits(m * n) = ResUnits(m) <*>
ResUnits(n)
  by (insert aux1-ResUnits-image-prop aux2-ResUnits-image-prop, auto)

lemma (in PHI2) mn-prop: m ~= 1 | n ~= 1 ==> 1 < m * n
proof-
  assume m ~= 1 | n ~= 1
```

```
  then have 1 ~= m * n
     by (auto, insert n-ge-1 m-ge-1 zmult-eq-1-iff [of m n], auto)
  moreover from n-ge-1 m-ge-1 have 0 < m * n
    by (auto simp add: mult-pos)
  ultimately show ?thesis by arith
qed

lemma (in PHI2) aux1-big-prop3: m ~= 1 | n ~= 1 ==>
    {x. 1 <= x & x <= m * n & zgcd (x, m * n) = 1} = ResUnits(m*n)
  apply (auto simp add: PHI2-def ResUnits-def SR-def)
  apply (case-tac x = m * n)
  apply (insert mn-prop)
  apply (auto)
  apply (case-tac x = m * n)
  apply (insert mn-prop)
  apply (auto)
  apply (case-tac x = 0, auto)
  apply (case-tac x = 0, auto)
done

lemma (in PHI2) aux2-big-prop3: m ~= 1 ==>
                {x. 1 <= x & x <= m & zgcd (x, m) = 1} = ResUnits(m)
  apply (auto simp add: ResUnits-def SR-def)
  apply (case-tac x = m, auto)
  apply (case-tac x = 0, auto)
done

lemma (in PHI2) aux3-big-prop3: n ~= 1 ==>
                {x. 1 <= x & x <= n & zgcd (x, n) = 1} = ResUnits(n)
  apply (auto simp add: ResUnits-def SR-def)
  apply (case-tac x = n, auto)
  apply (case-tac x = 0, auto)
done



theorem (in PHI2) big-prop3: phi(m * n) = phi(m) * phi(n)
proof-
  have phi(m*n) = int(card(ResUnits(m*n)))
    apply (auto simp add: phi-def)
    apply (case-tac m = 1)
    apply (case-tac n = 1)deferdefer
    apply (case-tac n = 1)
  proof-
    assume m ~= 1
```

then show *card* {*x*. *1* <= *x* & *x* <= *m* * *n* & *zgcd* (*x*, *m* * *n*) = *1*} = *card* (*ResUnits* (*m* * *n*))
by (*auto simp add*: *aux1-big-prop3*)
next
assume *m* ~= *1*
then show *card* {*x*. *1* <= *x* & *x* <= *m* * *n* & *zgcd* (*x*, *m* * *n*) = *1*} = *card* (*ResUnits* (*m* * *n*))
by (*auto simp add*: *aux1-big-prop3*)
next
assume *m* = *1* **and** *n* = *1*
then have *card* {*x*. *1* <= *x* & *x* <= *m* * *n* & *zgcd* (*x*, *m* * *n*) = *1*} = *card* {*x*. (*1*::*int*) <= *x* & *x* <= *1*}
by(*auto*)
also have {*x*. (*1*::*int*) <= *x* & *x* <= *1*} = {*x*. (*0*::*int*) < *x* & *x* <= *1*}
by (*auto*)
also have *card* {*x*. (*0*::*int*) < *x* & *x* <= *1*} = *nat 1*
apply (*subgoal-tac* {*x*. (*0*::*int*) < *x* & *x* <= *1*} = {}*0*..*1*})
apply *auto*
done
finally have *p1*: *card* {*x*. *1* <= *x* & *x* <= *m* * *n* & *zgcd* (*x*, *m* * *n*) = *1*} = *nat 1* **.**
from *prems* **have** *card* (*ResUnits* (*m* * *n*)) = *card* {*x*. (*0*::*int*) <= *x* & *x* < *1*}
by (*auto*)
also have ... = *nat 1*
apply (*subgoal-tac* {*x*. (*0*::*int*) <= *x* & *x* < *1*} = {*0*..*1*(}})
apply *auto*
done
finally have *card* (*ResUnits* (*m* * *n*)) = *nat 1* **.**
with *p1* **show** *card* {*x*. *1* <= *x* & *x* <= *m* * *n* & *zgcd* (*x*, *m* * *n*) = *1*} = *card* (*ResUnits* (*m* * *n*))
by (*auto*)
next
assume *n* ~= *1*
then show *card* {*x*. *1* <= *x* & *x* <= *m* * *n* & *zgcd* (*x*, *m* * *n*) = *1*} = *card* (*ResUnits* (*m* * *n*))
by (*auto simp add*: *aux1-big-prop3*)
qed
also have *card* (*ResUnits* (*m* * *n*)) = *card* (*f* ‘ *ResUnits* (*m* * *n*))
apply (*insert ResUnits-mn-finite ResUnits-inj-on-f*)
apply (*auto simp add*: *card-image*)
done
also have *f* ‘ *ResUnits* (*m* * *n*) = *ResUnits*(*m*) <*> *ResUnits*(*n*)
by (*auto simp add*: *ResUnits-image-prop*)
also have *card* (*ResUnits m* <*> *ResUnits n*) = *card* (*ResUnits m*) * *card*

```
(ResUnits n)
    apply (insert ResUnits-m-finite ResUnits-n-finite)
    by (simp add: setsum-constant)
  also have int (card (ResUnits m) * card (ResUnits n)) = int (card (ResUnits
m)) * int (card (ResUnits n))
    by (auto simp add: zmult-int)
  also have int (card (ResUnits m)) = phi (m)
    apply (auto simp add: phi-def)
    apply (case-tac m = 1)
  proof−
    assume m = 1
   then have card {x. 1 <= x & x <= m & zgcd (x, m) = 1} = card {x. (1::int)
<= x & x <= 1}
      by(auto)
    also have {x. (1::int) <= x & x <= 1} = {x. (0::int) < x & x <= 1}
      by (auto)
    also have card {x. (0::int) < x & x <= 1} = nat 1
      by (subgoal-tac {x. (0::int) < x & x <= 1} = {}0..1}, auto)
    finally have p1: card {x. 1 <= x & x <= m & zgcd (x, m) = 1} = nat 1 .
    from prems have card (ResUnits (m)) = card {x. (0::int) <= x & x < 1}
      by (auto)
    also have ... = nat 1
      apply (subgoal-tac {x. 0 ≤ x ∧ x < 1} = {0..1(})
      apply (erule ssubst)
      apply auto
      done
    finally have card (ResUnits (m)) = nat 1 .
     with p1 have card {x. 1 <= x & x <= m & zgcd (x, m) = 1} = card
(ResUnits (m))
      by (auto)
    then show card (ResUnits m) = card {x. 1 <= x & x <= m & zgcd (x, m)
= 1}
      by auto
  next
    assume m ~= 1
    then show card (ResUnits m) = card {x. 1 <= x & x <= m & zgcd (x, m)
= 1}
      by(auto simp add: aux2-big-prop3)
  qed
  also have int (card (ResUnits n)) = phi (n)
    apply (auto simp add: phi-def)
    apply (case-tac n = 1)
  proof−
    assume n = 1
   then have card {x. 1 <= x & x <= n & zgcd (x, n) = 1} = card {x. (1::int)
```

```
<= x & x <= 1}
      by(auto)
    also have {x. (1::int) <= x & x <= 1} = {x. (0::int) < x & x <= 1}
      by (auto)
    also have card {x. (0::int) < x & x <= 1} = nat 1
      apply (subgoal-tac {x. (0::int) < x & x <= 1} = {}0..1})
      apply (erule ssubst)
      apply auto
      done
    finally have p1: card {x. 1 <= x & x <= n & zgcd (x, n) = 1} = nat 1 .
    from prems have card (ResUnits (n)) = card {x. (0::int) <= x & x < 1}
      by (auto)
    also have ... = nat 1
      apply (subgoal-tac {x. 0 ≤ x ∧ x < 1} = {0..1(})
      apply (erule ssubst, auto)
      done
    finally have card (ResUnits (n)) = nat 1 .
   with p1 have card {x. 1 <= x & x <= n & zgcd (x, n) = 1} = card (ResUnits
(n))
      by (auto)
    then show card (ResUnits n) = card {x. 1 <= x & x <= n & zgcd (x, n) =
1}
      by auto
  next
    assume n ~= 1
    then show card (ResUnits n) = card {x. 1 <= x & x <= n & zgcd (x, n) =
1}
      by(auto simp add: aux3-big-prop3)
  qed
  finally show ?thesis .
qed

end
```

23 The radical function

```
theory Radical = PrimeFactorsList:

consts
  pdivisors  :: int => int list
  rad        :: int => int
  squarefree :: int => bool
  subplist   :: int list => int list => bool
```

defs
pdivisors-def: *pdivisors n == remdups(pfactors n)*
rad-def: *rad n == zprod(pdivisors n)*
squarefree-def: *squarefree n ==*
(if (n <= 1) then True
else (distinct (pfactors n)))

23.1 Some Initial Properties Involving distinct

lemma *distinct-numoccurs-le-1*: *distinct list = (ALL p. (numoccurs p list) <= 1)*
proof−
have *distinct list −−> (ALL p. (numoccurs p list) <= 1)*
by (*induct list*, *auto simp add*: *not-mem-numoccurs-eq-0 set-mem-eq*)
moreover have ~(*distinct list*) −−> ~(*ALL p. (numoccurs p list <= 1)*)
apply (*induct list*)
apply (*auto simp add*: *mem-numoccurs-gr-0 set-mem-eq*)
apply (*rule-tac x = p* **in** *exI*, *auto*)
done
ultimately show *?thesis* **by** *auto*
qed

23.2 Some Initial Properties Involving remdups

lemma *zprimel-remdups-prop* [*rule-format*]: *zprimel l −−> zprimel (remdups l)*
by (*auto simp add*: *zprimel-def*)

lemma *aux* [*rule-format*]: *znondec (remdups list) −−> znondec (a # list) −−>*
znondec (remdups (a # list))
apply (*induct-tac list*, *auto*)
apply (*case-tac list*, *auto*)
done

lemma *znondec-remdups-prop* [*rule-format*]: *znondec l −−> znondec (remdups l)*
apply (*induct-tac l*, *force*, *clarify*)
apply (*subgoal-tac znondec*(*remdups list*))
apply (*drule aux*, *force*, *force*)
apply (*drule znondec-distrib*, *force*)
done

lemma *not-empty-remdups*: (*l* ~= []) = (*remdups*(*l*) ~= [])
by (*induct l*, *auto*)

lemma *zprimel-zprod-ge-1* [*rule-format*]: *zprimel pl −−> 1 <= zprod* (*remdups* (*pl*))
apply (*induct pl*, *auto simp add*: *zprimel-def zprime-def*)
proof−
fix *a* **and** *list*
assume *1 <* (*a*::*int*) **and** *1 <= zprod* (*remdups list*)
then have *b*: *1 <= a* **by** *auto*
from *b* **have** *c*: *0 <= a* **by** *arith*
have *1 * 1 <= a * zprod* (*remdups list*)
apply (*rule mult-mono*)
by (*auto simp add*: *prems b c*)
thus *1 <= a * zprod* (*remdups list*) **by** *auto*
qed

lemma *pfactors-zprod-ge-1*: *1 <= zprod* (*remdups* (*pfactors n*))
apply (*rule-tac zprimel-zprod-ge-1*)
apply (*auto simp add*: *pfactors-ac*)
done

lemma *zprimel-zprod-le* [*rule-format*]: *zprimel pl −−> zprod* (*remdups* (*pl*)) <= *zprod* (*pl*)
proof (*induct pl*, *auto*)
fix *a* **and** *list*
assume ~ *zprimel list* **and** *a* : *set list* **and** *zprimel* (*a # list*)
thus *zprod* (*remdups list*) <= *a * zprod list* **by** (*auto simp add*: *zprimel-def*)
next
fix *a* **and** *list*
assume ~ *zprimel list* **and** *a* ~: *set list* **and** *zprimel* (*a # list*)
thus *a * zprod* (*remdups list*) <= *a * zprod list* **by** (*auto simp add*: *zprimel-def*)
next
fix *a* **and** *list*
assume *zprod* (*remdups list*) <= *zprod list* **and** *zprimel* (*a # list*)
then have *zprod* (*remdups list*) * *1 <= zprod list * a*
apply (*intro mult-mono*)
apply (*auto simp add*: *zprimel-def zprime-def zprodl-zprimel-pos*)
done
also have *zprod* (*remdups list*) * *1* = *zprod* (*remdups list*) **by** *auto*
also have *zprod list * a = a * zprod list* **by** *auto*
finally show *zprod* (*remdups list*) <= *a * zprod list*.
next
fix *a* **and** *list*
assume *zprod* (*remdups list*) <= *zprod list* **and** *a* ~: *set list* **and** *zprimel* (*a # list*)
thus *a * zprod* (*remdups list*) <= *a * zprod list*

by (*auto simp add*: *zprimel-def zprime-def mult-left-mono*)
qed

lemma *pfactors-zprod-le*: *zprod* (*remdups* (*pfactors n*)) <= *zprod* (*pfactors n*)
apply (*rule-tac zprimel-zprod-le*)
apply (*auto simp add*: *pfactors-ac*)
done

lemma *numoccurs-remdups*: *ALL p. numoccurs p* (*remdups list*) <= *numoccurs p list*
apply (*induct list*, *auto*)
apply (*drule-tac x = a* **in** *spec*, *auto*)
done

23.3 Properties about pdivisors

lemma *pdivisors-zprimel*: *zprimel* (*pdivisors n*)
apply (*insert pfactors-zprimel* [*of n*])
apply (*drule zprimel-remdups-prop*)
apply (*auto simp add*: *pdivisors-def*)
done

lemma *pdivisors-znondec*: *znondec* (*pdivisors n*)
apply (*insert pfactors-znondec* [*of n*])
apply (*drule znondec-remdups-prop*)
apply (*auto simp add*: *pdivisors-def*)
done

lemma *pdivisors-le-1*: *n <= 1 ==> pdivisors n = []*
by (*auto simp add*: *pdivisors-def pfactors-def*)

lemma *pdivisors-gr-1*: *1 < n ==> pdivisors n ~= []*
apply (*drule pfactors-gr-1*)
apply (*auto simp add*: *pdivisors-def*)
apply (*auto simp add*: *not-empty-remdups*)
done

lemmas *pdivisors-ac = pdivisors-le-1 pdivisors-gr-1*
pdivisors-znondec pdivisors-zprimel

23.4 Properties about rad

lemma *le-1-rad-prop*: *n <= 1 ==> rad* (*n*) = *1*
by (*auto simp add*: *rad-def pfactors-def pdivisors-def*)

```
lemma rad-max: 1 < n ==> rad n <= n
proof−
  assume 1 < n
  note pfactors-zprod-le
  also have zprod (remdups (pfactors n)) = rad n
    by (auto simp add: rad-def pdivisors-def)
  also have zprod (pfactors n) = n
    by (insert prems, auto simp add: pfactors-ac)
  finally show ?thesis .
qed

lemma rad-min-gre-1:1 <= n ==> 1 <= rad n
proof−
  assume 1 <= n
  note pfactors-zprod-ge-1
  also have zprod (remdups (pfactors n)) = rad n
    by (auto simp add: rad-def pdivisors-def)
  finally show 1 <= rad n by auto
qed

lemma rad-min-gr-1:1 < n ==> 1 < rad n
proof−
  assume 1 < n
  then have 1 <= rad n by (auto simp add: rad-min-gre-1)
  moreover have 1 ~= rad n
  proof
    assume 1 = rad n
    then have zprod( pdivisors n) = 1
      by (auto simp add: rad-def)
    moreover have zprimel (pdivisors n)
      by (auto simp add: pdivisors-ac)
    ultimately have pdivisors n = []
      by (auto simp add: zprimel-zprod-eq-1-impl-empty)
    moreover have pdivisors n ~= []
      by (insert prems, auto simp add: pdivisors-ac)
    ultimately show False by auto
  qed
  ultimately show ?thesis by auto
qed

lemma rad-prime: p:zprime ==> rad (p) = p
  by (auto simp add: rad-def pfactors-zprime pdivisors-def)

lemma pfactors-rad-eq-pdivisors: pfactors (rad n) = pdivisors n
  by (rule pfactors-simp-prop, auto simp add: pdivisors-ac rad-def)
```

lemma *mem-pfactors-mem-pdivisors*: *p mem* (*pfactors n*) = *p mem* (*pdivisors n*)
by (*auto simp add*: *pdivisors-def set-mem-eq*)

lemma *zprod-remdups-dvd* [*rule-format*]: *x mem list* −−> *x dvd* (*zprod*(*remdups list*))
by (*induct list*, *auto simp add*: *dvd-def set-mem-eq*)

lemma *mem-pfactors-zdvd-rad*: [| *1* <= *n*; *p mem* (*pfactors n*) |] ==> *p dvd* (*rad n*)
proof−
assume *1* <= *n* **and** *p mem* (*pfactors n*)
then have *p dvd* (*zprod*(*remdups*(*pfactors n*)))
by (*auto simp add*: *zprod-remdups-dvd*)
also have *zprod* (*remdups* (*pfactors n*)) = *rad n*
by (*auto simp add*: *rad-def pdivisors-def*)
finally show *?thesis* **.**
qed

lemma *mem-pdivisors-dvd-rad*: [| *1* <= *n*; *p mem* (*pdivisors n*) |] ==> *p dvd rad*(*n*)
proof−
assume *1* <= *n* **and** *p mem* (*pdivisors n*)
then have *p dvd* (*zprod* (*pdivisors n*))
by (*auto simp add*: *zprod-zdvd*)
also have *zprod* (*pdivisors n*) = *rad n*
by (*auto simp add*: *prems rad-def*)
finally show *?thesis***.**
qed

lemma *dvd-rad-mem-pdivisors*: [| *1* <= *n*; *p*:*zprime*; *p dvd rad*(*n*) |] ==>
p mem (*pdivisors n*)
proof−
assume *1* <= *n* **and** *p dvd rad*(*n*)
then have *p dvd zprod* (*pdivisors n*)
by (*auto simp add*: *rad-def*)
moreover note *pdivisors-ac*
moreover assume *p*:*zprime*
ultimately show *p mem* (*pdivisors n*)
by (*auto simp add*: *zprod-zprime-prop*)
qed

lemma *zprime-zdvd-rad*: [| *p*:*zprime*; *1* <= *n* |] ==> (*p dvd n*) = (*p dvd rad* (*n*))
proof
assume *p*:*zprime* **and** *1* <= *n* **and** *p dvd n*

then have *p mem* (*pfactors n*) **by** (*auto simp add*: *zdvd-imp-mem-pfactors*)
thus *p dvd rad* (*n*) **by** (*auto simp add*: *prems mem-pfactors-zdvd-rad*)
next
assume *p*:*zprime* **and** *1 <= n* **and** *p dvd rad n*
then have *p mem pdivisors n*
by (*auto simp add*: *dvd-rad-mem-pdivisors*)
then have *p mem pfactors n*
by (*auto simp add*: *mem-pfactors-mem-pdivisors*)
thus *p dvd n*
by (*insert prems*, *auto simp add*: *mem-pfactors-imp-zdvd*)
qed

lemma *rad-squarefree*: *1 <= n ==> squarefree* (*rad* (*n*))
by (*auto simp add*: *squarefree-def pfactors-rad-eq-pdivisors pdivisors-def*)

lemma *rad-multiplicity-le*: *ALL p. multiplicity p* (*rad n*) *<= multiplicity p n*
by (*auto simp add*: *multiplicity-def pfactors-rad-eq-pdivisors*
pdivisors-def numoccurs-remdups)

lemma *rad-zdvd*: *1 <= n ==> rad* (*n*) *dvd n*
proof−
assume *1 <= n*
have *ALL p. multiplicity p* (*rad n*) *<= multiplicity p n*
by (*auto simp add*: *rad-multiplicity-le*)
moreover have *1 <= rad n*
by (*auto simp add*: *prems rad-min-gre-1*)
moreover note *multiplicity-le-imp-zdvd*
ultimately show *?thesis* **by** (*insert prems*, *auto*)
qed

23.5 Properties about squarefree

lemma *squarefree-prop*: *squarefree*(*n*) = (*ALL p.* (*multiplicity p n*) *<= 1*)
apply (*simp add*: *squarefree-def multiplicity-def*)
apply (*auto simp add*: *distinct-numoccurs-le-1*)
apply (*auto simp add*: *pfactors-def*)
done

lemma *squarefree-zdvd-imp-zdvd-rad*: [| *1 <= m*; *1 <= n*; *m dvd n*; *squarefree m*
|] ==> *m dvd rad*(*n*)
proof−
assume *1 <= m* **and** *1 <= n*
then have *p1*: *1 <= rad n* **by** (*auto simp add*: *rad-min-gre-1*)
assume *m dvd n* **and** *squarefree m*
then have *ALL p.* (*multiplicity p m <= multiplicity p* (*rad n*))

```
    apply (auto simp add: squarefree-prop)
    apply (drule-tac x = p in spec)
    apply (case-tac multiplicity p m = 0, auto)
    apply (subgoal-tac multiplicity p m = 1, auto)
  proof−
    fix p
    assume m dvd n
    assume multiplicity p m = Suc 0
    then have multiplicity p m = 1 by auto
    then have p dvd m by (auto simp add: multiplicity-eq-1-imp-zdvd)
    then have p dvd n by (insert prems zdvd-trans [of p m n], auto)
    moreover have p2: p : zprime
      by (insert prems not-zprime-multiplicity-eq-0 [of p m], auto)
    ultimately have p dvd rad(n)
      by (insert prems zprime-zdvd-rad [of p n], auto)
    then have 1 <= multiplicity p (rad n)
      by (insert prems p1 p2 zdvd-zprime-imp-multiplicity-ge-1 [of rad n p], auto)
    thus Suc 0 <= multiplicity p (rad n) by auto
  qed
  thus ?thesis by (insert prems p1, auto simp add: multiplicity-le-imp-zdvd)
qed

lemma squarefree-zdvd-impl-squarefree: [| 0 < n; squarefree n; m dvd n |] ==>
squarefree m
proof−
  assume squarefree n and 0 < n and m dvd n
  then have ALL p. (multiplicity p n) <= 1 by (auto simp add: squarefree-prop)
  moreover have ALL p. (multiplicity p m <= multiplicity p n)
    by (insert prems, auto simp add:  zdvd-imp-multiplicity-le)
  ultimately have ALL p. (multiplicity p m <= 1)
    apply (clarify)
    apply (drule-tac x = p in spec)+
    apply (arith)
  done
  thus squarefree m by (auto simp add: squarefree-prop)
qed

lemma squarefree-distrib1: [| 0 < x; 0 < y; squarefree(x * y) |] ==>
                           squarefree (x)
proof−
  assume 0 < x and 0 < y
  then have 0 < x * y by (auto simp add: mult-pos)
  moreover have x dvd x * y by auto
  moreover assume squarefree(x * y)
  ultimately show ?thesis
```

by (*insert prems squarefree-zdvd-impl-squarefree* [*of x∗y x*], *auto*)
qed

lemma *squarefree-distrib2*: [| *0 < x*; *0 < y*; *squarefree*(*x ∗ y*) |] ==>
squarefree (*y*)
proof−
assume *0 < x* **and** *0 < y*
then have *0 < x ∗ y* **by** (*auto simp add*: *mult-pos*)
moreover have *y dvd x ∗ y* **by** *auto*
moreover assume *squarefree*(*x ∗ y*)
ultimately show *?thesis*
by (*insert prems squarefree-zdvd-impl-squarefree* [*of x∗y y*], *auto*)
qed

end

24 Properties of the mu function

theory *Mu* = *Radical*:

lemma *zdvd-zmult-not-zdvd-impl-zdvd*: [| *p*:*zprime*; *0 < d*; *0 < a*; (*d*::*int*) *dvd a ∗ p* ; ~ *p dvd d* |] ==> *d dvd a*
apply (*rule-tac multiplicity-le-imp-zdvd*, *auto*)
apply (*case-tac p = pa*, *auto*)
proof−
assume ~ *p dvd d* **and** *0 < d*
then have ~ *p mem pfactors d*
by (*insert mem-pfactors-imp-zdvd* [*of d p*], *auto*)
then have *multiplicity p d = 0*
by (*auto simp add*: *multiplicity-def not-mem-numoccurs-eq-0*)
thus *multiplicity p d <= multiplicity p a*
by *auto*
next
fix *pa*
assume *p* ~= *pa*
assume *p*:*zprime* **and** *0 < a*
then have *0 < a ∗ p*

by (*insert prems*, *auto simp add*: *mult-pos zprime-def*)
moreover assume *d dvd a * p*
ultimately have *ALL pq*. (*multiplicity pq d <= multiplicity pq (a * p)*)
by (*drule-tac zdvd-imp-multiplicity-le*, *auto*)
then have *multiplicity pa d <= multiplicity pa (a * p)*
by *auto*
also have *multiplicity pa (a * p) = multiplicity pa a + multiplicity pa p*
by (*insert prems*, *auto simp add*: *multiplicity-zmult-distrib zprime-def*)
also have *multiplicity pa p = 0*
by (*insert prems*, *auto simp add*: *multiplicity-def pfactors-zprime*)
finally show *multiplicity pa d <= multiplicity pa a* **by** *auto*
qed

consts
mu :: *int => int*

defs
mu-def: *mu n ==*
(if (n <= 1) then 1
else if (squarefree n) then
(if (even(int(length(pfactors n)))) then 1
else −1)
else 0)

24.1 Properties about mu

lemma *mu-eq-1*: *n <= 1 ==> mu (n) = 1*
by (*auto simp add*: *mu-def*)

lemma *mu-zprime*: *p:zprime ==> mu(p) = −1*
by (*auto simp add*: *mu-def squarefree-def pfactors-zprime zprime-def*)

lemma *mu-squarefree1*: [| *1 <= n*; ~*squarefree(n)* |] ==> *mu(n) = 0*
apply (*case-tac* ~*squarefree n*)
apply (*auto simp add*: *mu-def*)
apply (*simp add*: *squarefree-def*)
done

lemma *mu-squarefree2*: [| *1 <= n*; *mu(n)* ~*= 0* |] ==> *squarefree(n)*
proof−
assume *1 <= n* **and** *mu(n)* ~*= 0*

moreover note *mu-squarefree1*
ultimately show *?thesis* **by** *auto*
qed

lemma *aux1*: [| *p*:*zprime*; *1* < *d*; *squarefree*(*d* * *p*) |] ==> *mu*(*d* * *p*) = *-1* * *mu*(*d*)
apply (*subgoal-tac 0* < *d*) **defer**
apply (*force*)
apply (*subgoal-tac 1* < *p*) **defer**
apply (*simp add*: *zprime-def*)
apply (*subgoal-tac 1* < *d* * *p*)**defer**
apply (*insert zmult-eq-1-iff* [*of d p*] *zero-less-mult-iff* [*of d p*], *force*)
apply (*subgoal-tac length* (*pfactors* (*d* * *p*)) = *length*(*pfactors d*) + *1*)**defer**
apply (*simp add*: *pfactors-zprime-zmult-length*)
apply (*frule-tac x* = *d* **and** *y* = *p* **in** *squarefree-distrib1*, *force*, *force*)
apply (*frule-tac x* = *d* **and** *y* = *p* **in** *squarefree-distrib2*, *force*, *force*)
apply (*auto simp add*: *mu-def*)
done

lemma *aux2*: [| *p*:*zprime*; *1* = *d*; *squarefree*(*d* * *p*) |] ==> *mu*(*d* * *p*) = *-1* * *mu*(*d*)
apply (*auto simp add*: *mu-def length-pfactors-zprime zprime-def*)
done

lemma *squarefree-mu-prop*: [| *p*:*zprime*; *1* <= *d*; *squarefree*(*d* * *p*) |] ==> *mu*(*d* * *p*) = *−mu*(*d*)
apply (*case-tac 1* = *d*)
apply (*drule aux2*, *auto*)
apply (*subgoal-tac 1* < *d*)
apply (*auto simp add*: *aux1*)
done

lemma *pos-pos-zdvd-finite*: (*0*::*int*) < *m* ==> *finite* {*d*. *0* < *d* & *d dvd m*}
proof−
assume *0* < *m*
then have {*d*. *0* < *d* & *d dvd m*} <= {*d*. *0* < *d* & *d* <= *m*}
apply (*auto*)
apply (*drule zdvd-bounds*, *auto*)

```
  done
  moreover have finite {d. 0 < d & d <= m}
    apply (subgoal-tac {d. 0 < d & d <= m} = {}0..m})
    apply (erule ssubst, auto)
    done
 ultimately show ?thesis by (auto simp add: finite-subset)
qed
```

```
lemma zprime-zdvd-existence-rad: 1 < n ==> (hd (pdivisors n)):zprime & (hd
(pdivisors n)) dvd (rad n)
  apply (case-tac pdivisors n)
  apply (frule pdivisors-gr-1, force)
  apply (insert pdivisors-zprimel [of n])
  apply (subgoal-tac a mem (pdivisors n))
  apply (auto simp add:  mem-pdivisors-dvd-rad zprimel-def)
done
```

```
lemma squarefree-subsets-prop1: {d. (0::int) < d & d dvd n} =
                                {d. 0 < d & d dvd n & squarefree d} Un {d. 0 < d &
d dvd n & ~squarefree d}
  by (auto)

lemma squarefree-subsets-prop2: {d. 0 < d & d dvd n & squarefree d} Int {d. 0
< d & d dvd n & ~squarefree d} = {}
  by (auto)

lemma squarefree-subset1-finite: (0::int) < n ==> finite {d. 0 < d & d dvd n &
squarefree d}
proof-
  assume 0 < n
  then have finite {d. 0 < d & d dvd n}
    by (auto simp add: pos-pos-zdvd-finite)
  moreover have {d. 0 < d & d dvd n & squarefree d} <= {d. 0 < d & d dvd
n}
    by (auto)
 ultimately show ?thesis by (auto simp add: finite-subset)
```

qed

lemma *squarefree-subset2-finite*: $(0::int) < n ==>$ *finite* $\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ \sim squarefree\ d\}$
proof−
assume $0 < n$
then have *finite* $\{d.\ 0 < d \ \&\ d\ dvd\ n\}$
by (*auto simp add*: *pos-pos-zdvd-finite*)
moreover have $\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ \sim squarefree\ d\} <= \{d.\ 0 < d \ \&\ d\ dvd\ n\}$
by (*auto*)
ultimately show *?thesis* **by** (*auto simp add*: *finite-subset*)
qed

lemma *key-step1*: $(0::int) < n ==>$
setsum mu $\{d.\ 0 < d \ \&\ d\ dvd\ n\} =$
setsum mu $\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ squarefree\ d\}$ + *setsum mu* $\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ \sim\ squarefree\ d\}$
proof−
assume $0 < n$
then have *finite* $\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ squarefree\ d\}$ **and**
finite $\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ \sim\ squarefree\ d\}$
by (*insert squarefree-subset1-finite* [*of n*] *squarefree-subset2-finite* [*of n*], *auto*)
moreover have $\{d.\ (0::int) < d \ \&\ d\ dvd\ n \ \&\ squarefree\ d\}$ *Int* $\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ \sim squarefree\ d\} = \{\}$
by (*auto simp add*: *squarefree-subsets-prop2*)
ultimately have *setsum mu* $(\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ squarefree\ d\}$ *Un* $\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ \sim\ squarefree\ d\}) =$
setsum mu $\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ squarefree\ d\}$ + *setsum mu* $\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ \sim\ squarefree\ d\}$
by (*rule setsum-Un-disjoint*)
also have $\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ squarefree\ d\}$ *Un* $\{d.\ 0 < d \ \&\ d\ dvd\ n \ \&\ \sim squarefree\ d\} =$
$\{d.\ 0 < d \ \&\ d\ dvd\ n\}$
by (*auto simp add*: *squarefree-subsets-prop1*)
finally show *?thesis* **.**
qed

lemma *not-squarefree-setsum-prop*: $0 < n ==>$ *setsum mu* $\{d.\ (0::int) < d \ \&\ d\ dvd\ n \ \&\ \sim squarefree\ d\} = 0$
proof−

assume $0 < n$
then have $ALL\ x{:}\{d.\ (0{::}int) < d\ \&\ d\ dvd\ n\ \&\ \sim squarefree\ d\}.\ mu\ x = (\%x.\ 0)\ x$
by (*auto simp add*: *mu-squarefree1*)
moreover have *p1*: *finite* $\{d.\ (0{::}int) < d\ \&\ d\ dvd\ n\ \&\ \sim squarefree\ d\}$
by (*insert prems*, *auto simp add*: *squarefree-subset2-finite*)
ultimately have $setsum\ mu\ \{d.\ (0{::}int) < d\ \&\ d\ dvd\ n\ \&\ \sim squarefree\ d\} =$
$setsum\ (\%x.\ 0)\ \{d.\ (0{::}int) < d\ \&\ d\ dvd\ n\ \&\ \sim squarefree\ d\}$
by (*rule-tac setsum-cong*, *auto*)
also have $... = 0$
by (*rule finite-induct*, *auto simp add*: *p1*)
finally show *?thesis* **.**
qed

lemma *zdvd-squarefree-eq-zdvd-rad*: $0 < n ==> \{d.\ (0{::}int) < d\ \&\ d\ dvd\ n\ \&\ squarefree\ d\} =$
$\{d.\ 0 < d\ \&\ d\ dvd\ rad(n)\}$
apply (*auto*)
apply (*rule squarefree-zdvd-imp-zdvd-rad*, *auto*)
apply (*subgoal-tac rad n dvd n*)
apply (*frule zdvd-trans*, *auto*)
apply (*auto simp add*: *rad-zdvd*)
apply (*subgoal-tac* $0 < rad\ n$)
apply (*subgoal-tac squarefree* $(rad\ n)$)
apply (*frule-tac* $n = rad\ n$ **in** *squarefree-zdvd-impl-squarefree*)
apply (*auto simp add*: *rad-squarefree*)
apply (*insert rad-min-gre-1* [*of n*], *auto*)
done

lemma *key-step2*: $(0{::}int) < n ==>$
$setsum\ mu\ \{d.\ 0 < d\ \&\ d\ dvd\ n\ \&\ squarefree\ d\} + setsum\ mu\ \{d.\ 0 < d\ \&\ d\ dvd\ n\ \&\ \sim\ squarefree\ d\} =$
$setsum\ mu\ \{d.\ 0 < d\ \&\ d\ dvd\ (rad\ n)\}$
proof−
assume $0 < n$
then have $setsum\ mu\ \{d.\ 0 < d\ \&\ d\ dvd\ n\ \&\ squarefree\ d\} + setsum\ mu\ \{d.\ 0 < d\ \&\ d\ dvd\ n\ \&\ \sim\ squarefree\ d\} =$
$setsum\ mu\ \{d.\ 0 < d\ \&\ d\ dvd\ (rad\ n)\} + 0$
by (*auto simp add*: *not-squarefree-setsum-prop zdvd-squarefree-eq-zdvd-rad*)
thus *?thesis* **by** *auto*
qed

lemma *zdvd-subsets-prop1*: $\{d.\ (0::int) < d\ \&\ d\ dvd\ m\} =$
$\{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ p\ dvd\ d\}\ Un\ \{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ \sim p\ dvd\ d\}$
by (*auto*)

lemma *zdvd-subsets-prop2*: $\{d.\ (0::int) < d\ \&\ d\ dvd\ m\ \&\ p\ dvd\ d\}\ Int\ \{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ \sim p\ dvd\ d\} = \{\}$
by (*auto*)

lemma *subset1-finite*: $(0::int) < m \Longrightarrow finite\ \{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ p\ dvd\ d\}$
proof−
assume $0 < m$
then have *finite* $\{d.\ 0 < d\ \&\ d\ dvd\ m\}$
by (*auto simp add*: *pos-pos-zdvd-finite*)
moreover have $\{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ p\ dvd\ d\} <= \{d.\ 0 < d\ \&\ d\ dvd\ m\}$
by (*auto*)
ultimately show *?thesis* **by** (*auto simp add*: *finite-subset*)
qed

lemma *subset2-finite*: $(0::int) < m \Longrightarrow finite\ \{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ \sim p\ dvd\ d\}$
proof−
assume $0 < m$
then have *finite* $\{d.\ 0 < d\ \&\ d\ dvd\ m\}$
by (*auto simp add*: *pos-pos-zdvd-finite*)
moreover have $\{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ \sim p\ dvd\ d\} <= \{d.\ 0 < d\ \&\ d\ dvd\ m\}$
by (*auto*)
ultimately show *?thesis* **by** (*auto simp add*: *finite-subset*)
qed

lemma *key-step3*: $(0::int) < m \Longrightarrow$
$setsum\ mu\ \{d.\ 0 < d\ \&\ d\ dvd\ m\} =$
$setsum\ mu\ \{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ p\ dvd\ d\} + setsum\ mu\ \{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ \sim p\ dvd\ d\}$
proof−
assume $0 < m$
then have *finite* $\{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ p\ dvd\ d\}$ **and**
finite $\{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ \sim p\ dvd\ d\}$
by (*insert subset1-finite* [*of m p*] *subset2-finite* [*of m p*], *auto*)
moreover have $\{d.\ (0::int) < d\ \&\ d\ dvd\ m\ \&\ p\ dvd\ d\}\ Int\ \{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ \sim p\ dvd\ d\} = \{\}$
by (*auto simp add*: *zdvd-subsets-prop2*)
ultimately have $setsum\ mu\ (\{d.\ 0 < d\ \&\ d\ dvd\ m\ \&\ p\ dvd\ d\}\ Un\ \{d.\ 0 < d$

```
& d dvd m & ~ p dvd d}) =
                setsum mu {d. 0 < d & d dvd m & p dvd d} + setsum mu {d. 0
< d & d dvd m & ~ p dvd d}
   by (rule setsum-Un-disjoint)
  also have {d. 0 < d & d dvd m & p dvd d} Un {d. 0 < d & d dvd m & ~ p
dvd d} =
          {d. (0::int) < d & d dvd m}
   by (auto simp add: zdvd-subsets-prop1)
  finally show ?thesis .
qed
```

```
lemma aux-inject-prop: p:zprime ==> inj-on (%x. (x::int) * p)  {d. (0::int) <
d & d dvd (m div p)}
  by (auto simp add: inj-on-def, auto simp add: zprime-def)

lemma aux-finite-prop: [| p:zprime; 0 < m; p dvd m |] ==> finite {d. (0::int) <
d & d dvd (m div p)}
proof-
  assume p : zprime and 0 < m and p dvd m
  then have 0 < m div p
  proof-
    have p <= m by (insert prems zdvd-bounds [of p m], auto)
    then have p div p <= m div p
      by (insert prems, rule zdiv-mono1, auto simp add: zprime-def)
    also have p div p = 1 by (insert prems, auto simp add: zprime-def)
    finally show 0 < m div p by auto
  qed
  thus ?thesis by (auto simp add: pos-pos-zdvd-finite)
qed

lemma aux-image-prop: [| p:zprime; p dvd m |] ==>
                    ((%x. (x::int) * p) ` {d. (0::int) < d & d dvd (m div p)}) =
                    {d. 0 < d & d dvd m & p dvd d }
  apply (auto simp add: image-def)
  apply (clarsimp simp add: zprime-def zero-less-mult-iff) defer
  apply (rule-tac x = x div p in exI, auto)
proof-
  fix x
  assume p:zprime and p dvd x and 0 < x
  then have p <= x by (insert zdvd-bounds [of p x], auto)
  then have p div p <= x div p
```

```
    by (insert prems, rule zdiv-mono1, auto simp add: zprime-def)
  also have p div p = 1 by (insert prems, auto simp add: zprime-def)
  finally show 0 < x div p by auto
next
  fix x
  assume p:zprime
  assume p dvd x and p1: x dvd m
  then have p2: p dvd m by (rule zdvd-trans)
  note p1
  also have m = p * (m div p)
    by (insert p2 zmod-zdiv-equality [of m p] zdvd-iff-zmod-eq-0 [of p m], auto)
  also have x = p * (x div p)
    by (insert prems zmod-zdiv-equality [of x p] zdvd-iff-zmod-eq-0 [of p x], auto)
  finally have p * (x div p) dvd p * (m div p).
  thus (x div p) dvd (m div p)
    apply (rule-tac a = p in ne-0-zdvd-prop)
    apply (insert prems, auto simp add: zprime-def)
  done
next
  fix x
  assume p dvd x
  thus x = (x div p) * p
   by (insert zmod-zdiv-equality [of x p] zdvd-iff-zmod-eq-0 [of p x], auto)
next
  fix xa
  assume p dvd m
  then have p1 :p * (m div p) = m
    by (insert zmod-zdiv-equality [of m p] zdvd-iff-zmod-eq-0 [of p m], auto)
  assume xa dvd m div p
  then have p * xa dvd p * (m div p)
    by (insert zdvd-refl [of p], auto simp add: zdvd-zmult-mono)
  also note p1
  finally show xa * p dvd m by (auto simp add: zmult-ac)
qed

lemma aux3: [| p : zprime; 0 < m; p dvd m|] ==>
                setsum mu {d. 0 < d & d dvd m & p dvd d} =
                 setsum (mu ∘ (%x. (x::int) * p)) {d. (0::int) < d & d dvd (m div
p)}
proof−
  assume p:zprime and 0 < m and p dvd m
  then have setsum mu ((%x. (x::int) * p) ‘ {d. (0::int) < d & d dvd (m div
p)}) =
           setsum (mu ∘ (%x. (x::int) * p)) {d. (0::int) < d & d dvd (m div p)}
    apply (insert prems aux-finite-prop [of p m] aux-inject-prop [of p m])
```

```
  apply (rule setsum-reindex, auto)
 done
 also have ((%x. (x::int) * p) ` {d. (0::int) < d & d dvd (m div p)}) =
         {d. 0 < d & d dvd m & p dvd d }
  by (insert prems, auto simp add: aux-image-prop)
 finally show ?thesis .
qed

lemma aux4: [| p:zprime; 0 < m; p dvd m; squarefree m|] ==>
          ALL x:{d. (0::int) < d & d dvd (m div p)}.
            (mu o (%x. (x::int) * p))x = ((%x. -(x::int)) o mu)x
 apply (auto, rule squarefree-mu-prop, auto)
proof-
 fix x
 assume x dvd m div p and p dvd m
 then have x * p dvd (m div p) * p
  by (insert zdvd-refl [of p], auto simp add: zdvd-zmult-mono)
 also have (m div p) * p = m
  by (insert prems zmod-zdiv-equality [of m p] zdvd-iff-zmod-eq-0 [of p m], auto)
 finally have x * p dvd m.
 moreover assume squarefree m and 0 < m
 ultimately show squarefree(x * p)
  by (rule-tac squarefree-zdvd-impl-squarefree)
qed

lemma aux5: [| p : zprime; 0 < m; p dvd m; squarefree m |] ==>
              setsum (mu o (%x. (x::int) * p)) {d. (0::int) < d & d dvd (m div
p)} =
             setsum ((%x. -(x::int)) o mu) {d. (0::int) < d & d dvd (m div p)}
proof-
 assume p : zprime and 0 < m and p dvd m and squarefree m
 with aux4 have ALL x:{d. (0::int) < d & d dvd (m div p)}.
           (mu o (%x. (x::int) * p))x = ((%x. -(x::int)) o mu)x
  by (auto)
 moreover from aux-finite-prop have finite {d. (0::int) < d & d dvd (m div p)}
  by (insert prems, auto)
 ultimately show ?thesis
  apply (intro setsum-cong)
  apply auto
  done
qed

lemma aux6: [| p:zprime; 0 < m; p dvd m; squarefree m |] ==>
          {d. (0::int) < d & d dvd m & ~p dvd d} = {d. 0 < d & d dvd (m div
p)}
```

```
  apply (auto, rule zdvd-zmult-not-zdvd-impl-zdvd, auto)
proof−
  fix x
  assume p :zprime and 0 < m and p dvd m
  then have p <= m by (insert zdvd-bounds [of p m], auto)
  then have p div p <= m div p
    by (insert prems, rule zdiv-mono1, auto simp add: zprime-def)
  also have p div p = 1 by (insert prems, auto simp add: zprime-def)
  finally show 0 < m div p by auto
next
  fix x
  assume p dvd m
  assume x dvd m
  also have m = (m div p) * p
    by (insert prems zmod-zdiv-equality [of m p] zdvd-iff-zmod-eq-0 [of p m], auto)
  finally show x dvd m div p * p .
next
  fix x
  assume p dvd m
  assume x dvd m div p
  then have x dvd (m div p) * p
    by (auto simp add: zdvd-zmult2)
  also have (m div p) * p = m
    by (insert prems zmod-zdiv-equality [of m p] zdvd-iff-zmod-eq-0 [of p m], auto)
  finally show x dvd m .
next
  fix x
  assume p dvd m and p:zprime and 0 < m
  assume x dvd m div p and p dvd x
  then have p dvd m div p by (rule-tac zdvd-trans)
  then have p * p dvd (m div p) * p
    by (insert zdvd-refl [of p], auto simp add: zdvd-zmult-mono)
  also have p * p = p ^ Suc(Suc(0))
    by (auto simp add: power-Suc)
  also have Suc(Suc(0))= 2 by auto
  also have m div p * p = m
    by (insert prems zmod-zdiv-equality [of m p] zdvd-iff-zmod-eq-0 [of p m], auto)
  finally have p ^ 2 dvd m .
  then have 2 <= multiplicity p m
    by (insert prems, auto simp add: multiplicity-zpower-zdvd)
  moreover assume squarefree m
  ultimately show False
    apply (auto simp add: squarefree-prop)
    apply (drule-tac x = p in spec, auto)
  done
```

qed

lemma *aux7*: [| *p* : *zprime*; *0* < *m*; *p dvd m*; *squarefree m* |] ==>
setsum mu {*d*. *0* < *d* & *d dvd m* & ~*p dvd d*} =
setsum mu {*d*. *0* < *d* & *d dvd* (*m div p*)}
by (*auto simp add*: *aux6*)

lemma *key-step4*: [| *p* : *zprime*; *0* < *m*; *p dvd m*; *squarefree m* |] ==>
setsum mu {*d*. *0* < *d* & *d dvd m* & *p dvd d*} + *setsum mu* {*d*. *0* < *d* & *d dvd m* & ~ *p dvd d*} =
setsum ((%*x*. −(*x*::*int*)) ∘ *mu*) {*d*. (*0*::*int*) < *d* & *d dvd* (*m div p*)} +
setsum mu {*d*. *0* < *d* & *d dvd* (*m div p*)}
proof−
assume *p* : *zprime* **and** *0* < *m* **and** *p dvd m* **and** *squarefree m*
then have *setsum mu* {*d*. *0* < *d* & *d dvd m* & *p dvd d*} + *setsum mu* {*d*. *0* < *d* & *d dvd m* & ~ *p dvd d*} =
setsum (*mu* ∘ (%*x*. (*x*::*int*) ∗ *p*)) {*d*. (*0*::*int*) < *d* & *d dvd* (*m div p*)} +
setsum mu {*d*. *0* < *d* & *d dvd* (*m div p*)}
by (*auto simp add*: *aux3 aux7*)
also have *setsum* (*mu* ∘ (%*x*. (*x*::*int*) ∗ *p*)) {*d*. (*0*::*int*) < *d* & *d dvd* (*m div p*)} =
setsum ((%*x*. −(*x*::*int*)) ∘ *mu*) {*d*. (*0*::*int*) < *d* & *d dvd* (*m div p*)}
by (*insert prems*, *auto simp add*: *aux5*)
finally show *?thesis* **.**
qed

lemma *key-step5*: [| *p* : *zprime*; *0* < *m*; *p dvd m* |] ==>
setsum ((%*x*. −(*x*::*int*)) ∘ *mu*) {*d*. (*0*::*int*) < *d* & *d dvd* (*m div p*)} +
setsum mu {*d*. *0* < *d* & *d dvd* (*m div p*)} =
0
proof−
assume *p* : *zprime* **and** *0* < *m* **and** *p dvd m*
have *setsum* ((%*x*. −(*x*::*int*)) ∘ *mu*) {*d*. (*0*::*int*) < *d* & *d dvd* (*m div p*)} +
setsum mu {*d*. *0* < *d* & *d dvd* (*m div p*)} =
setsum (%*x*. ((%*x*. −(*x*::*int*)) ∘ *mu*)(*x*) + (*mu x*)) {*d*. (*0*::*int*) < *d* & *d dvd* (*m div p*)}
by (*auto simp add*: *setsum-addf* [*THEN sym*])
also have *setsum* (%*x*. ((%*x*. −(*x*::*int*)) ∘ *mu*)(*x*) + (*mu x*)) {*d*. (*0*::*int*) < *d* & *d dvd* (*m div p*)} = *0*
apply (*insert prems aux-finite-prop* [*of p m*])
apply (*auto simp add*: *setsum-constant*)
done

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

theorem *moebius-prop*: [| *1 <= n* |] ==> *setsum mu* {*d*. *0* < *d* & *d dvd n*} = (*if n = 1 then 1 else 0*)
apply (*case-tac n = 1*)
apply (*auto*)
apply (*subgoal-tac* {*d*. (*0*::*int*) < *d* & *d dvd 1*} = {*1*})
apply (*simp*)
apply (*simp add*: *mu-def*)
apply (*insert ge-0-zdvd-1*)
apply (*force*)
apply (*subgoal-tac 1 < n*)
apply (*auto*)
proof−
assume *1 < n*
then have *p1*: *1 < rad n* **by** (*auto simp add*: *rad-min-gr-1*)
have *p2*: (*hd* (*pdivisors n*)):*zprime* **and** *p3*: (*hd* (*pdivisors n*)) *dvd* (*rad n*)
by (*insert prems*, *auto simp add*: *zprime-zdvd-existence-rad*)
have *p4*: *squarefree* (*rad n*)
by (*insert prems*, *auto simp add*: *rad-squarefree*)
have *setsum mu* {*d*. *0* < *d* & *d dvd n*} =
setsum mu {*d*. *0* < *d* & *d dvd n* & *squarefree d*} + *setsum mu* {*d*. *0* < *d* & *d dvd n* & ~ *squarefree d*}
by (*insert prems*, *auto simp add*: *key-step1*)
also have ... = *setsum mu* {*d*. *0* < *d* & *d dvd* (*rad n*)}
by (*insert prems*, *auto simp add*: *key-step2*)
also have ... = *setsum mu* {*d*. *0* < *d* & *d dvd* (*rad n*) & (*hd* (*pdivisors n*)) *dvd d*} +
setsum mu {*d*. *0* < *d* & *d dvd* (*rad n*) & ~ (*hd* (*pdivisors n*)) *dvd d*}
apply (*subgoal-tac 0 < rad n*)
apply (*drule-tac p* = (*hd* (*pdivisors n*)) **in** *key-step3*, *force*)
apply (*insert p1*, *auto*)
done
also have ... = *setsum* ((%*x*. −(*x*::*int*)) ∘ *mu*) {*d*. (*0*::*int*) < *d* & *d dvd* ((*rad n*) *div* (*hd* (*pdivisors n*)))} +
setsum mu {*d*. *0* < *d* & *d dvd* ((*rad n*) *div* (*hd* (*pdivisors n*)))}
apply (*rule-tac key-step4*)
apply (*insert p1 p2 p3 p4*, *auto*)
done
also have ... = *0*
apply (*rule-tac key-step5*)
apply (*insert prems p1 p2 p3*, *auto*)
done

finally show *setsum mu* {*d*. *0* < *d* & *d dvd n*} = *0*.
qed

end

25 Moebius inversion and variants

theory *Inversion* = *NatIntLib* + *FiniteLib* + *Mu*:

25.1 The central lemma

locale *MITEMP2* =
fixes *n* :: *int*
fixes *f* :: (*int* * *int*) => ('*b*::*plus-ac0*)
fixes *S* :: (*int* * *int*) *set*
fixes *R* :: *int* => (*int* * *int*) *set*
fixes *U* :: *int* => *int set*
fixes *T* :: *int* => (*int* **int*) *set*
fixes *V* :: *int* => *int set*

assumes *n-gr-0*: *0* < *n*

defines *S-def*: *S* == { (*d*,*d'*). *d*: {}*0*..*n*} & *d'*:{}*0*..*n*} & (*d* * *d'*) <= *n* }
defines *R-def*: *R d* == { (*c*,*d'*). *c* = *d* & *d'*:{}*0*..*n*} & (*d* * *d'*) <= *n* }
defines *U-def*: *U d* == { *d'*. *d'*:{}*0*..*n*} & (*d* * *d'*) <= *n* }
defines *T-def*: *T c* == { (*d*,*d'*). (*d*,*d'*):*S* & (*d* * *d'*) = *c* }
defines *V-def*: *V c* == { *d'*. *d'*:{}*0*..*n*} & *d' dvd c* }

lemma (**in** *MITEMP2*) *calc1*: (∑ *x*:*S*. *f*(*snd x*, *fst x*)) =
(∑ *d*:{}*0*..*n*}. (∑ *x*:(*R d*). *f*(*snd x*, *fst x*)))
proof−
have *finite* {}*0*..*n*} **by** *auto*
moreover have *ALL d*:{}*0*..*n*}. *finite* (*R d*)
proof
fix *d*
have *finite* ({*d*::*int*} <*> {}*0*..*n*}) **by** *auto*
moreover have (*R d*) <= {*d*} <*> {}*0*..*n*} **by** (*auto simp add*: *R-def*)
ultimately show *finite* (*R d*) **by** (*auto simp add*: *finite-subset*)
qed
moreover have *ALL x*:{}*0*..*n*}. *ALL y*:{}*0*..*n*}. *x* ~= *y* --> (*R x*) *Int* (*R y*)
= {}
by (*auto simp add*: *R-def*)
ultimately have *setsum* (%*x*. *f*(*snd x*, *fst x*)) (*UNION* {}*0*..*n*} *R*) =

```
                (∑ d:{}0..n}. setsum (%x. f(snd x, fst x)) (R d))
    by (auto simp add: setsum-UN-disjoint)
  moreover have S = UNION {}0..n} R
    by (auto simp add: S-def R-def)
  ultimately show setsum (%x. f(snd x, fst x)) S =
          (∑ d:{}0..n}. setsum (%x. f(snd x, fst x)) (R d))
    by auto
qed

lemma (in MITEMP2) calc2: (∑ d:{}0..n}. (∑ x:(R d). f(snd x, fst x))) =
                          (∑ d:{}0..n}. (∑ x:(R d). f(snd x, d)))
  apply (rule setsum-cong, auto)
  apply (rule setsum-cong)
  apply (rule refl)
  apply (subgoal-tac fst xa = x)
  apply force
  apply (unfold R-def)
  apply auto
  done

lemma (in MITEMP2) calc3: (∑ d:{}0..n}. (∑ x:(R d). f(snd x, d))) =
                          (∑ d:{}0..n}. (∑ x:(U d). f(x, d)))
  apply (rule setsum-cong, auto)
proof−
  fix x
  have finite (R x)
  proof−
    have finite ({x} <*> {}0..n}) by auto
    moreover have (R x) <= {x} <*> {}0..n} by (auto simp add: R-def)
    ultimately show ?thesis by (auto simp add: finite-subset)
  qed
  moreover have inj-on snd (R x) by (auto simp add: inj-on-def R-def)
  ultimately have setsum ((%xa. f(xa, x)) ∘ snd) (R x) =
                  setsum (%xa. f(xa, x)) (snd ‘ R x)
    by (auto simp add: setsum-reindex [THEN sym])
  moreover have snd ‘ (R x) = U x by (auto simp add: image-def R-def U-def)
  ultimately have setsum (%xa. f(snd xa, x)) (R x) =
                  setsum (%xa. f(xa, x)) (U x)
    by (auto simp add: comp-def)
  thus (∑ xa:R x. f (snd xa, x)) = (∑ xa:U x. f (xa, x)).
qed

lemma (in MITEMP2) calc4: (∑ x:S. f(snd x, fst x)) =
                          (∑ c:{}0..n}. (∑ x:(T c). f(snd x, fst x)))
proof−
```

```
  have finite {}0..n} by auto
  moreover have ALL c:{}0..n}. finite (T c)
  proof
    fix c
    have finite ({}0..n} <*> {}0..n}) by auto
    moreover have (T c) <= {}0..n} <*> {}0..n}
      by (auto simp add: T-def S-def)
    ultimately show finite (T c) by (auto simp add: finite-subset)
  qed
  moreover have ALL x:{}0..n}. ALL y:{}0..n}. x ~= y -->
      (T x) Int (T y) = {}
    by (auto simp add: T-def)
  ultimately have setsum (%x. f(snd x, fst x)) (UNION {}0..n} T) =
                  (∑ d:{}0..n}. setsum (%x. f(snd x, fst x)) (T d))
    by (auto simp add: setsum-UN-disjoint)
  moreover have S = UNION {}0..n} T
    by (auto simp add: S-def T-def zero-less-mult-iff)
  ultimately show setsum (%x. f(snd x, fst x)) S =
        (∑ d:{}0..n}. setsum (%x. f(snd x, fst x)) (T d))
    by (auto)
qed

lemma aux1: [| 0 < (x::int); x <= n; 0 < y; y <= n; y dvd x |] ==>
    x < n * y + y
proof-
  assume 0 < x and x <= n and 0 < y and y <= n and y dvd x
  then have x * 1 <= n * y
    apply (intro mult-mono)
    apply (insert prems, auto)
  done
  thus ?thesis by (insert prems, auto simp add: zadd-zless-mono)
qed

lemma (in MITEMP2) calc5:
  (∑ c:{}0..n}. (∑ x:(T c). f(snd x, fst x))) =
     (∑ c:{}0..n}. (∑ d:(V c). f(snd(d, c div d), fst(d, c div d))))
  apply (rule setsum-cong, auto)
proof-
  fix x
  assume p1:0 < x and p2:x <= n
  have f-V: finite (V x)
    proof-
      have finite {}0..n} by auto
      moreover have (V x) <= {}0..n} by (auto simp add: V-def)
      ultimately show ?thesis by (auto simp add: finite-subset)
```

```
  qed
 moreover have inj-on (%d. (d, x div d)) (V x)
  by (auto simp add: V-def inj-on-def)
 ultimately have setsum ((%y. f(snd y, fst y)) ∘ (%d. (d, x div d))) (V x) =
                 setsum (%y. f(snd y, fst y)) ((%d. (d, x div d)) ‘ V x)
  by (auto simp add: setsum-reindex [THEN sym])
 moreover have (%d. (d, x div d)) ‘ (V x) = (T x)
  apply (auto simp add: V-def T-def S-def image-def)
  apply (rule zdiv-gr-zero, auto simp add: p1 p2)
  apply (rule div-prop2, auto simp add: p1 p2 aux1)
  apply (insert prems, auto simp add: int-dvd-times-div-cancel)
 done
 ultimately have setsum (%y. f(snd(y, x div y), fst(y, x div y))) (V x) =
          setsum (%y. f(snd y, fst y)) (T x)
  by (auto simp add: comp-def)
 then have setsum (%y. f(x div y, y)) (V x) =
          setsum (%y. f(snd y, fst y)) (T x)
  by (auto)
 thus (∑ x:T x. f (snd x, fst x)) = (∑ d:V x. f (x div d, d)) by auto
qed

lemma (in MITEMP2) calc6:
   (∑ c:{}0..n}. (∑ d:(V c). f(snd(d, c div d), fst(d, c div d)))) =
     (∑ c:{}0..n}. (∑ d:(V c). f(c div d, d)))
 by auto

lemma (in MITEMP2) calc7:
   (∑ d:{}0..n}. (∑ d':(U d). f(d', d))) =
     (∑ d:{}0..n}. (∑ d':{}0..(n div d)}. f(d', d)))
 apply (rule setsum-cong, auto)
 apply (subgoal-tac (U x) = {}0..n div x}, auto simp add: U-def)
proof−
 fix x and xa
 assume 0 < x and x * xa <= n
 then have (x * xa) div x <= n div x
  by (insert zdiv-mono1 [of x*xa n x], auto)
 also have (x * xa) div x = xa
  by (insert prems, rule zdiv-zmult-self2, auto)
 finally show xa <= n div x.
next
 fix x and xa
 assume 0 < x and xa <= n div x
 then have n div x <= n div 1
  by (rule-tac a = n in zdiv-mono2, insert n-gr-0, auto)
 then have n div x <= n by auto
```

```
  thus xa <= n by (insert prems, auto)
next
  fix x and xa
  assume 0 < x and xa <= n div x
  moreover have 0 <= x by (insert prems, auto)
  ultimately have x * xa <= x * (n div x)
    by (auto simp add: mult-left-mono)
  moreover have x * (n div x) <= n
    by (insert prems, rule zdiv-leq-prop, auto)
  ultimately show x * xa <= n by auto
qed

lemma (in MITEMP2) inv-short:
  (∑ d:{}0..n}. (∑ d':{}0..(n div d)}. f(d', d))) =
    (∑ c:{}0..n}. (∑ d:(V c). f(c div d, d)))
  by (insert calc1 calc2 calc3 calc4 calc5 calc6 calc7, auto)

theorem general-inversion-aux: 0 < (n::int) ==>
    (∑ d:{}0..n}. (∑ d':{}0..(n div d)}. f(d', d))) =
     (∑ c:{}0..n}. (∑ d: {d'. d':{}0..n} & d' dvd c }. f(c div d, d)))
  by (auto simp add: MITEMP2-def MITEMP2.inv-short)
```

25.2 General inversion laws

```
lemma general-inversion-int1: 0 < (n::int) ⟹
    (∑ x:{x. 0 < x & x dvd n}. f x) = (∑ x:{x. 0 < x & x dvd n}. f (n div x))
  apply (rule setsum-reindex-cong')
  apply (erule finite-int-dvd-set)
  apply (rule inj-onI)
  apply (erule int-inj-div)
  apply auto
  apply (unfold image-def)
  apply clarify
  apply (rule-tac x = n div x in bexI)
  apply (rule int-div-div [THEN sym])
  apply auto
  apply (erule zdiv-gr-zero)
  apply auto
  apply (auto simp add: dvd-def)
  apply (erule zero-less-mult-pos)
  apply assumption
done

lemma general-inversion-int2: 0 < (n::int) ==>
  (∑ d:{}0..n}. (∑ d':{}0..(n div d)}. (f d d'))) =
```

$(\sum c{:}\{)0..n\}.\ (\sum d{:}\ \{d.\ 0 < d\ \&\ d\ dvd\ c\}.\ (f\ d\ (c\ div\ d))))$
apply (*insert general-inversion-aux* [*of n %x.* (*f* (*snd x*) (*fst x*))])
apply *simp*
apply (*rule setsum-cong*)
apply *auto*
apply (*rule setsum-cong*)
apply *auto*
apply (*frule zdvd-imp-le*)
apply *assumption*
apply (*erule order-trans*)
apply *assumption*
done

lemma *general-inversion-int2-cor1*: $0 < (n{::}int) ==>$
$(\sum d{:}\{)0..n\}.\ (\sum d'{:}\{)0..(n\ div\ d)\}.\ (f\ d\ d'))) =$
$(\sum d'{:}\{)0..n\}.\ (\sum d{:}\{)0..(n\ div\ d')\}.\ (f\ d\ d')))$
apply (*subst general-inversion-int2*)
apply *assumption*
apply (*subst general-inversion-int2*)
apply *assumption*
apply (*rule setsum-cong2*)
apply (*subst general-inversion-int1*)
apply *force*
apply (*rule setsum-cong2*)
apply (*subst int-div-div*)
apply *auto*
done

lemma *general-inversion-int2-cor2*: $0 < (n{::}int) ==>$
$(\sum d{:}\{)0..n\}.\ (\sum d'{:}\{)0..(n\ div\ d)\}.\ f\ d')) =$
$(\sum c{:}\{)0..n\}.\ (\sum d{:}\ \{d.\ 0 < d\ \&\ d\ dvd\ c\}.\ f\ (c\ div\ d)))$
by (*rule general-inversion-int2*)

lemma *general-inversion-int2-cor3*: $0 < (n{::}int) ==>$
$(\sum d{:}\{)0..n\}.\ (\sum d'{:}\{)0..(n\ div\ d)\}.\ f\ d')) =$
$(\sum d{:}\{)0..n\}.\ of\text{-}int\ (n\ div\ d) * f\ d)$
apply (*subst general-inversion-int2-cor1*)
apply *assumption*
apply (*rule setsum-cong2*)
apply (*subst setsum-constant*)
apply *simp*
apply *simp*
apply (*subgoal-tac of-int*$(n\ div\ x) =$ *of-int*(*of-nat*(*nat*$(n\ div\ x)$)))
apply (*erule ssubst*)
apply (*subst of-int-of-nat-eq*)

```
  apply (rule refl)
  apply (subst int-eq-of-nat [THEN sym])
  apply (subst nat-0-le)
  apply (rule int-nonneg-div-nonneg)
  apply auto
done

lemma general-inversion-int3: 0 < (n::int) ==>
  (∑ d:{d. 0 < d & d dvd n}.
    (∑ d':{d'.0 < d' & d' dvd (n div d)}. (f d' d))) =
  (∑ c:{c. 0 < c & c dvd n}.
    (∑ d: {d. 0 < d & d dvd c}. (f (c div d) d)))
proof -
  assume 0 < n
  let ?f1 = %d' d. if (d dvd n & d' dvd (n div d)) then f d' d else 0
  have (∑ d:{}0..n}. ∑ d':{}0..n div d}. ?f1 d' d) =
      (∑ c:{}0..n}. ∑ d:{d. 0 < d & d dvd c}. ?f1 (c div d) d)
      (is ?LHS = ?RHS)
    by (intro general-inversion-int2)
  also have ?LHS = (∑ d:{d. 0 < d & d dvd n}.
      (∑ d':{d'.0 < d' & d' dvd (n div d)}. (f d' d))) (is ?LHS = ?RHS2)
  proof -
    have ?LHS = (∑ d:{d. 0 < d & d dvd n}. ∑ d':{}0..n div d}. ?f1 d' d) +
      (∑ d:{d. d:{}0..n} & ~ (d dvd n)}. ∑ d':{}0..n div d}. ?f1 d' d)
        (is ?LHS = ?term1 + ?term2)
      apply (subst setsum-Un-disjoint [THEN sym])
      apply (rule finite-int-dvd-set, rule prems)
      apply (rule finite-subset-GreaterThan0AtMost-int)
      apply force
      apply force
      apply (rule setsum-cong)
      apply auto
      apply (erule zdvd-imp-le, rule prems)
      done
    also have ?term2 = 0
      apply (rule setsum-0')
      apply auto
      apply (rule setsum-0')
      apply auto
      done
    also have ?term1 + 0 = ?term1
      by simp
    also have ?term1 = ?RHS2
      apply (rule setsum-cong2)
      apply (subgoal-tac {}0..n div x} = {d'. 0 < d' & d' dvd n div x} Un
```

```
      {d'. 0 < d' & ~(d' dvd n div x) & d' <= n div x})
    apply (erule ssubst)
    apply (subst setsum-Un-disjoint)
    apply (rule finite-int-dvd-set)
    apply (clarsimp)
    apply (rule zdiv-gr-zero)
    apply assumption
    apply (rule prems)
    apply assumption
    apply (rule finite-subset-GreaterThan0AtMost-int)
    apply force
    apply force
    apply (subgoal-tac (∑ d':{d'. 0 < d' & ~ d' dvd n div x &
      d' <= n div x}.
        if x dvd n & d' dvd n div x then f d' x else 0) = 0)
    apply (erule ssubst)
    apply simp
    apply (rule setsum-cong2)
    apply (clarsimp)
    apply (rule setsum-0')
    apply clarsimp
    apply auto
    apply (erule zdvd-imp-le)
    apply (rule zdiv-gr-zero)
    apply (assumption, rule prems, assumption)
    done
  finally show ?thesis.
qed
also have ?RHS = (∑ c:{c. 0 < c & c dvd n}.
    (∑ d: {d. 0 < d & d dvd c}. (f (c div d) d))) (is ?RHS = ?LHS2)
proof -
  have ?RHS = (∑ c:{c. 0 < c & c dvd n}.
      ∑ d:{d. 0 < d & d dvd c}. ?f1 (c div d) d) +
    (∑ c:{c. 0 < c & c <= n & ~(c dvd n)}.
       ∑ d:{d. 0 < d & d dvd c}. ?f1 (c div d) d)
     (is ?RHS = ?term1 + ?term2)
    apply (subst setsum-Un-disjoint [THEN sym])
    apply (rule finite-int-dvd-set)
    apply (rule prems)
    apply (rule finite-subset-GreaterThan0AtMost-int)
    apply force
    apply force
    apply (rule setsum-cong)
    apply auto
    apply (rule zdvd-imp-le)
```

apply (*assumption*, *rule prems*)
done
also have *?term2 = 0*
apply (*rule setsum-0'*)
apply *clarsimp*
apply (*rule setsum-0'*)
apply *clarsimp*
apply (*subgoal-tac a dvd n*)
apply *force*
apply (*subgoal-tac* (*a dvd n*) = (*aa* ∗ (*a div aa*) *dvd aa* ∗ (*n div aa*)))
apply (*erule ssubst*)
apply (*rule zdvd-zmult-mono*)
apply *simp*
apply *assumption*
apply (*subst int-dvd-times-div-cancel2*)
apply *assumption+*
apply (*subst int-dvd-times-div-cancel2*)
apply (*assumption+*, *rule refl*)
done
also have *?term1 + 0 = ?term1*
by *simp*
also have ... = *?LHS2*
apply (*rule setsum-cong2*)
apply (*rule setsum-cong2*)
apply *auto*
apply (*erule notE*)
apply (*erule zdvd-trans*)
apply *assumption*
apply (*erule notE*)
apply (*subgoal-tac xa* ∗ (*x div xa*) *dvd xa* ∗ (*n div xa*))
apply (*rule ne-0-zdvd-prop*)
prefer *2*
apply *assumption*
apply *force*
apply (*subst int-dvd-times-div-cancel2*)
apply *assumption+*
apply (*erule zdvd-trans*, *assumption*)
apply (*subst int-dvd-times-div-cancel2*)
apply *assumption+*
done
finally show *?thesis*.
qed
finally show *?thesis*.
qed

lemma *general-inversion-nat1*: $0 < (n::nat) \Longrightarrow$
$(\sum x{:}\{x.\ x\ dvd\ n\}.\ f\ x) = (\sum x{:}\{x.\ x\ dvd\ n\}.\ f\ (n\ div\ x))$
proof −
assume $0 < (n::nat)$
have $(\sum x{:}\{x.\ 0 < x\ \&\ x\ dvd\ (int\ n)\}.\ (f\ o\ nat)\ x) =$
$(\sum x{:}\{x.\ 0 < x\ \&\ x\ dvd\ (int\ n)\}.\ (f\ o\ nat)\ ((int\ n)\ div\ x))$
(**is** *?LHS = ?RHS*)
apply (*rule general-inversion-int1*)
apply (*simp add*: *prems*)
done
also have *?LHS* $= (\sum x{:}\{x.\ x\ dvd\ n\}.\ ((f\ o\ nat)\ o\ int)\ x)$
apply (*rule setsum-reindex-cong*)
apply (*rule finite-nat-dvd-set*)
apply (*rule prems*)
apply (*force simp add*: *inj-on-def*)
apply (*subst image-int-dvd-set*)
apply (*rule prems*)
apply (*rule refl*)+
done
also have $((f\ o\ nat)\ o\ int) = f$
by (*simp add*: *o-def*)
also have *?RHS* =
$(\sum x{:}\{x.\ x\ dvd\ n\}.\ ((f\ o\ nat)\ o\ int)\ (n\ div\ x))$
apply (*rule setsum-reindex-cong*)
apply (*rule finite-nat-dvd-set*)
apply (*rule prems*)
apply (*subgoal-tac inj-on int* $\{x.\ x\ dvd\ n\}$)
apply *assumption*
apply (*force simp add*: *inj-on-def*)
apply (*subst image-int-dvd-set*)
apply (*rule prems*)
apply (*rule refl*)
apply (*simp add*: *o-def int-div* [*THEN sym*])
done
also have $((f\ o\ nat)\ o\ int) = f$
by (*simp add*: *o-def*)
finally show *?thesis*.
qed

lemma *general-inversion-nat2*: $0 < (n::nat) ==>$
$(\sum d{:}\{)0..n\}.\ (\sum d'{:}\{)0..(n\ div\ d)\}.\ (f\ d\ d'))) =$
$(\sum c{:}\{)0..n\}.\ (\sum d{:}\ \{d.\ d\ dvd\ c\}.\ (f\ d\ (c\ div\ d))))$
proof −
assume $0 < (n::nat)$
have $(\sum d{:}\{)0..int\ n\}.\ (\sum d'{:}\{)0..((int\ n)\ div\ d)\}.\ (f\ (nat\ d)\ (nat\ d')))) =$

$(\sum c{:}\{)0..(int\ n)\}.\ (\sum d{:}\ \{d.\ 0 < d\ \&\ d\ dvd\ c\}.$
$(f\ (nat\ d)\ (nat\ (c\ div\ d)))))\ (\textbf{is}\ ?LHS = ?RHS)$
apply (*rule general-inversion-int2*)
apply (*simp add*: *prems*)
done
also have $?LHS = (\sum d{:}\{)0..n\}.\ (\sum d'{:}\{)0..(n\ div\ d)\}.\ (f\ d\ d')))$
apply (*rule setsum-reindex-cong'*)
apply *simp*
apply (*subgoal-tac inj-on int* $\{)0..n\}$)
apply *assumption*
apply (*simp add*: *inj-on-def*)
apply (*subst image-int-GreaterThanAtMost*)
apply *simp*
apply (*rule ext*)
apply (*rule sym*)
apply (*rule setsum-reindex-cong*)
apply *simp*
apply (*subgoal-tac inj-on int* $\{)0..n\ div\ d\}$)
apply *assumption*
apply (*force simp add*: *inj-on-def*)
apply (*subst image-int-GreaterThanAtMost*)
apply (*subst int-div*)
apply *simp*
apply (*simp add*: *o-def*)
done
also have $?RHS = (\sum c{:}\{)0..n\}.\ (\sum d{:}\ \{d.\ d\ dvd\ c\}.\ (f\ d\ (c\ div\ d))))$
apply (*rule setsum-reindex-cong''*)
apply *simp*
apply (*subgoal-tac inj-on int* $\{)0..n\}$)
apply *assumption*
apply (*simp add*: *inj-on-def*)
apply (*subst image-int-GreaterThanAtMost*)
apply *simp*
apply *clarsimp*
apply (*rule sym*)
apply (*rule setsum-reindex-cong'*)
apply (*rule finite-nat-dvd-set*)
apply *assumption*
apply (*subgoal-tac inj-on int* $\{d.\ d\ dvd\ x\}$)
apply *assumption*
apply (*force simp add*: *inj-on-def*)
apply (*subst image-int-dvd-set*)
apply (*assumption*, *rule refl*)
apply (*simp add*: *int-div* [*THEN sym*])
done

finally show *?thesis*.
qed

lemma *general-inversion-nat2-cor1*: $0 < (n::nat) ==>$
$(\sum d{:}\{)0..n\}.\ (\sum d'{:}\{)0..(n\ div\ d)\}.\ (f\ d\ d'))) =$
$(\sum d'{:}\{)0..n\}.\ (\sum d{:}\{)0..(n\ div\ d')\}.\ (f\ d\ d')))$
apply (*subst general-inversion-nat2*)
apply *assumption*
apply (*subst general-inversion-nat2*)
apply *assumption*
apply (*rule setsum-cong2*)
apply (*subst general-inversion-nat1*)
apply *force*
apply (*rule setsum-cong2*)
apply *simp*
apply (*subst nat-div-div*)
apply *auto*
done

lemma *general-inversion-nat2-cor2*: $0 < (n::nat) ==>$
$(\sum d{:}\{)0..n\}.\ (\sum d'{:}\{)0..(n\ div\ d)\}.\ f\ d')) =$
$(\sum c{:}\{)0..n\}.\ (\sum d{:}\ \{d.\ d\ dvd\ c\}.\ f\ (c\ div\ d)))$
by (*rule general-inversion-nat2*)

lemma *general-inversion-nat2-cor3*: $0 < (n::nat) ==>$
$(\sum d{:}\{)0..n\}.\ (\sum d'{:}\{)0..(n\ div\ d)\}.\ f\ d')) =$
$(\sum d{:}\{)0..n\}.\ of\text{-}nat\ (n\ div\ d) * f\ d)$
apply (*subst general-inversion-nat2-cor1*)
apply *assumption*
apply (*rule setsum-cong2*)
apply (*subst setsum-constant*)
apply *simp*
apply *simp*
done

lemma *general-inversion-nat3*: $0 < (n::nat) ==>$
$(\sum d{:}\{d.\ d\ dvd\ n\}.$
$(\sum d'{:}\{d'.\ d'\ dvd\ (n\ div\ d)\}.\ (f\ d'\ d)))) =$
$(\sum c{:}\{c.\ c\ dvd\ n\}.$
$(\sum d{:}\ \{d.\ d\ dvd\ c\}.\ (f\ (c\ div\ d)\ d)))$
(**is** $0 < n \Longrightarrow$ *?LHS* = *?RHS*)
proof −
assume $0 < (n::nat)$
have $(\sum d{:}\{d.\ 0 < d\ \&\ d\ dvd\ (int\ n)\}.$
$(\sum d'{:}\{d'.0 < d'\ \&\ d'\ dvd\ ((int\ n)\ div\ d)\}.\ (f\ (nat\ d')\ (nat\ d)))) =$

$(\sum c{:}\{c.\ 0 < c\ \&\ c\ dvd\ (int\ n)\}.$
$(\sum d{:}\ \{d.\ 0 < d\ \&\ d\ dvd\ c\}.\ (f\ (nat\ (c\ div\ d))\ (nat\ d))))$
(**is** *?LHS2 = ?RHS2*)
apply (*rule general-inversion-int3*)
apply (*simp add*: *prems*)
done
also have *?LHS2 = ?LHS*
apply (*rule setsum-reindex-cong''*)
apply (*rule finite-nat-dvd-set*)
apply (*rule prems*)
apply (*subgoal-tac inj-on int* $\{d.\ d\ dvd\ n\}$)
apply *assumption*
apply (*simp add*: *inj-on-def*)
apply (*subst image-int-dvd-set*)
apply (*rule prems*)
apply (*rule refl*)
apply *clarify*
apply (*rule sym*)
apply (*rule setsum-reindex-cong*)
apply (*rule finite-nat-dvd-set*)
apply (*rule nat-pos-div-dvd-gr-0*)
apply (*rule prems*)
apply (*assumption*)
apply (*subgoal-tac inj-on int* $\{d'.\ d'\ dvd\ n\ div\ x\}$)
apply *assumption*
apply (*simp add*: *inj-on-def*)
apply (*subst image-int-dvd-set*)
apply (*rule nat-pos-div-dvd-gr-0*, *rule prems*, *assumption*)
apply (*subst int-div*)
apply (*rule refl*)
apply (*simp add*: *o-def*)
done
also have *?RHS2 = ?RHS*
apply (*rule setsum-reindex-cong''*)
apply (*rule finite-nat-dvd-set*)
apply (*rule prems*)
apply (*subgoal-tac inj-on int* $\{c.\ c\ dvd\ n\}$)
apply *assumption*
apply (*simp add*: *inj-on-def*)
apply (*subst image-int-dvd-set*)
apply (*rule prems*, *rule refl*)
apply *clarsimp*
apply (*rule sym*)
apply (*rule setsum-reindex-cong'*)
apply (*rule finite-nat-dvd-set*)

apply (*erule nat-pos-dvd-pos*)
apply (*rule prems*)
apply (*subgoal-tac inj-on int* {*d*. *d dvd x*})
apply *assumption*
apply (*simp add*: *inj-on-def*)
apply (*subst image-int-dvd-set*)
apply (*erule nat-pos-dvd-pos*, *rule prems*)
apply (*rule refl*)
apply (*simp add*: *int-div* [*THEN sym*])
done
finally show *?thesis*.
qed

25.3 Moebius inversion

lemma *moebius-prop2*: *0 < n* ⟹ *setsum mu* {*d*. *0 < d* ∧ *d dvd n*} =
(*if n = 1 then 1 else 0*)
apply (*rule moebius-prop*, *force*)
done

lemma *moebius-prop-nat*: *0 < n* ⟹ *setsum* (%*x*. *mu*(*int x*))
{*d*. *d dvd n*} = (*if n = 1 then 1 else 0*)
(**is** *0 < n* ⟹ *?LHS = ?RHS*)
proof −
assume *0 < n*
have *setsum mu* {*d*. *0 < d* ∧ *d dvd* (*int n*)} =
(*if* (*int n*) = *1 then 1 else 0*) (**is** *?LHS2 = ?RHS2*)
apply (*rule moebius-prop2*)
apply (*simp add*: *prems*)
done
also have *?LHS2 = ?LHS*
apply (*rule setsum-reindex-cong′*)
apply (*rule finite-nat-dvd-set*)
apply (*rule prems*)
apply (*subgoal-tac inj-on int* {*d*. *d dvd n*})
apply *assumption*
apply (*simp add*: *inj-on-def*)
apply (*subst image-int-dvd-set*)
apply (*rule prems*)
apply (*rule refl*)+
done
also have *?RHS2 = ?RHS*
by *simp*
finally show *?thesis*.
qed

lemma *moebius-prop-nat-general*: *0 < n ⟹*
(∑ x∈{d. d dvd n}. of-int(mu (int x))) = (if n = 1 then 1 else 0)
apply *(subst setsum-of-int' [THEN sym])*
apply *(subst moebius-prop-nat)*
apply *assumption*
apply *simp*
done

lemma *moebius-prop-int-general*: *0 < n ⟹*
(∑ x∈{d. 0 < d & d dvd n}. of-int(mu x)) = (if n = 1 then 1 else 0)
apply *(subst setsum-of-int' [THEN sym])*
apply *(subst moebius-prop2)*
apply *assumption*
apply *simp*
done

lemma *mu-aux*: *0 < (x::nat) ⟹*
*(setsum (%x. f x * (if x = 1 then 1 else 0))) {d. d dvd x} =*
((f 1)::'a::semiring)
apply *(subgoal-tac {d. d dvd x} = {1} Un {d. d dvd x & d ~= 1})*
apply *(erule ssubst)*
apply *(subst setsum-Un-disjoint)*
apply *simp*
apply *(rule finite-subset)*
apply *(subgoal-tac {d. d dvd x & d ~= 1} <= {d. d dvd x})*
apply *assumption*
apply *force*
apply *(rule finite-nat-dvd-set)*
apply *assumption*
apply *simp-all*
apply *(subgoal-tac (∑ x∈{d. d dvd x & d ~= Suc 0}. f x *
(if x = Suc 0 then 1 else 0)) = 0)*
apply *simp*
apply *(rule setsum-0')*
apply *auto*
done

lemma *mu-aux2*: *0 < x ⟹*
*(setsum (%x. f x * (if x = 1 then 1 else 0))) {)0..(x::nat)} =*
((f 1)::'a::semiring)
apply *(subgoal-tac {)0..x} = {1} Un {)1..x})*
apply *(erule ssubst)*
apply *(subst setsum-Un-disjoint)*
apply *simp-all*

apply (*subgoal-tac* ($\sum x{\in}\{)Suc\ 0..x\}$. $f\ x$ $*$
  (*if* $x = Suc\ 0$ *then* 1 *else* 0)) $= 0$)
apply *simp*
apply (*rule setsum-0'*)
apply *auto*
done

lemma *mu-inversion-nat1*: $0 < (n{::}nat) \Longrightarrow g\ n =$
 ($\sum d{:}\{d.\ d\ dvd\ n\}$. ($\sum d'{:}\{d'.\ d'\ dvd\ (n\ div\ d)\}$.
  of-int(*mu*(*int d*)) $*$ *g*((*n div d*) *div d'*)))
  (**is** $0 < n \Longrightarrow g\ n = ?sum$)
proof −
 assume $0 < n$
 then have $?sum = (\sum c{\in}\{c.\ c\ dvd\ n\}.\ \sum d{\in}\{d.\ d\ dvd\ c\}.$
  of-int (*mu* (*int d*)) $*$ *g* (*n div d div* (*c div d*)))
  by (*subst general-inversion-nat3*, *assumption*, *simp*)
 also have $\ldots = (\sum c{\in}\{c.\ c\ dvd\ n\}.\ \sum d{\in}\{d.\ d\ dvd\ c\}.$
  g(*n div c*) $*$ *of-int* (*mu* (*int d*)))
  apply (*rule setsum-cong2*)
  apply (*rule setsum-cong2*)
  apply (*subst nat-div-div-eq-div*)
  apply *force*
  apply *clarify*
  apply (*erule nat-pos-dvd-pos*)
  apply (*rule prems*)
  apply (*simp add*: *mult-ac*)
  done
 also have $\ldots = (\sum c{\in}\{c.\ c\ dvd\ n\}.\ g(n\ div\ c) * (\sum d{\in}\{d.\ d\ dvd\ c\}.$
  of-int (*mu* (*int d*))))
  apply (*rule setsum-cong2*)
  apply (*rule setsum-const-times*)
  done
 also have $\ldots = (\sum c{\in}\{c.\ c\ dvd\ n\}.\ g(n\ div\ c) * (if\ c = 1\ then\ 1$
  else 0))
  apply (*rule setsum-cong2*)
  apply (*subst moebius-prop-nat-general*)
  apply *auto*
  apply (*rule nat-pos-dvd-pos*)
  apply (*assumption*, *rule prems*)
  done
 also have $\ldots = g\ n$
  apply (*subst mu-aux*)
  apply (*rule prems*)
  apply *simp*
  done

finally show *?thesis* **by** (*rule sym*)
qed

lemma *mu-inversion-nat1a*: *ALL n.* (*0* < *n* ⟶
f n = (∑ *d*:{*d. d dvd n*}. *g*(*n div d*))) ⟹ *0* < (*n*::*nat*) ⟹
g n = (∑ *d*:{*d. d dvd n*}. *of-int*(*mu*(*int*(*d*))) ∗ *f* (*n div d*))
proof −
assume *ALL n.* (*0* < *n* ⟶ *f n* = (∑ *d*:{*d. d dvd n*}. *g*(*n div d*)))
assume *0* < *n*
show *g n* = (∑ *d*:{*d. d dvd n*}. *of-int*(*mu*(*int*(*d*))) ∗ *f* (*n div d*))
(**is** *g n* = *?sum*)
proof −
have *?sum* = (∑ *d*:{*d. d dvd n*}. *of-int*(*mu*(*int*(*d*))) ∗
(∑ *d′*:{*d′. d′ dvd* (*n div d*)}. *g*((*n div d*) *div d′*)))
apply (*rule setsum-cong2*)
apply (*insert prems*)
apply (*drule-tac x* = *n div x* **in** *spec*)
apply (*subgoal-tac 0* < *n div x*)
apply *force*
apply (*rule nat-pos-div-dvd-gr-0*)
apply *auto*
done
also have ... = (∑ *d*:{*d. d dvd n*}. (∑ *d′*:{*d′. d′ dvd* (*n div d*)}.
of-int(*mu*(*int*(*d*))) ∗ *g*((*n div d*) *div d′*)))
apply (*rule setsum-cong2*)
apply (*subst setsum-const-times*)
apply (*rule refl*)
done
also have ... = *g n*
apply (*subst mu-inversion-nat1*)
apply (*rule prems*)
apply (*rule refl*)
done
finally show *?thesis* **by** (*rule sym*)
qed
qed

lemma *mu-inversion-nat2*: *0* < (*n*::*nat*) ⟹ *f n* =
(∑ *d*:{*d. d dvd n*}. (∑ *d′*:{*d′. d′ dvd* (*n div d*)}.
of-int(*mu*(*int d′*)) ∗ *f*((*n div d*) *div d′*)))
(**is** *0* < *n* ⟹ *f n* = *?sum*)
proof −
assume *0* < *n*
then have *?sum* = (∑ *c*∈{*c. c dvd n*}. ∑ *d*∈{*d. d dvd c*}.
of-int (*mu* (*int* (*c div d*))) ∗ *f* (*n div d div* (*c div d*)))

by (*subst general-inversion-nat3*, *assumption*, *simp*)
also have ... = $(\sum c{\in}\{c.\ c\ dvd\ n\}.\ \sum d{\in}\{d.\ d\ dvd\ c\}.$
$(of\text{-}int\ (mu\ (int\ d))) * f\ (n\ div\ (c\ div\ d)\ div\ d)))$
apply (*rule setsum-cong2*)
apply (*subst general-inversion-nat1*)
apply (*insert prems*)
apply *clarify*
apply (*erule nat-pos-dvd-pos*)
apply *assumption*
apply (*rule setsum-cong2*)
apply (*subst nat-div-div*)
apply *auto*
apply (*erule nat-pos-dvd-pos*)
apply *assumption*
done
also have ... = $(\sum c{\in}\{c.\ c\ dvd\ n\}.\ \sum d{\in}\{d.\ d\ dvd\ c\}.$
$(f\ (n\ div\ c) * of\text{-}int\ (mu\ (int\ d))))$
apply (*rule setsum-cong2*)
apply (*rule setsum-cong2*)
apply (*subst div-mult2-eq* [*THEN sym*])
apply (*subst mult-commute*)**backback**
apply (*subst nat-dvd-mult-div*)
apply *clarsimp*
apply (*erule nat-pos-dvd-pos*)
apply (*erule nat-pos-dvd-pos*)
apply (*rule prems*)
apply *force*
apply (*simp add*: *mult-ac*)
done
also have ... = $(\sum c{\in}\{c.\ c\ dvd\ n\}.\ f(n\ div\ c) * (\sum d{\in}\{d.\ d\ dvd\ c\}.$
$of\text{-}int\ (mu\ (int\ d))))$
apply (*rule setsum-cong2*)
apply (*rule setsum-const-times*)
done
also have ... = $(\sum c{\in}\{c.\ c\ dvd\ n\}.\ f(n\ div\ c) * (if\ c = 1\ then\ 1$
$else\ 0))$
apply (*rule setsum-cong2*)
apply (*subst moebius-prop-nat-general*)
apply *auto*
apply (*rule nat-pos-dvd-pos*)
apply (*assumption*, *rule prems*)
done
also have ... = $f\ n$
apply (*subst mu-aux*)
apply (*rule prems*)

```
    apply simp
    done
  finally show ?thesis by (rule sym)
qed

lemma mu-inversion-nat2a: ALL n. (0 < n ⟶
  g n = (∑ d:{d. d dvd n}. of-int(mu(int(d))) * f(n div d))) ⟹
  0 < (n::nat) ⟹ f n = (∑ d:{d. d dvd n}. g (n div d))
proof −
  assume ALL n. (0 < n ⟶
      g n = (∑ d:{d. d dvd n}. of-int(mu(int(d))) * f(n div d)))
  assume 0 < n
  show f n = (∑ d:{d. d dvd n}. g (n div d)) (is f n = ?sum)
  proof −
    have ?sum = (∑ d:{d. d dvd n}. (∑ d':{d'. d' dvd (n div d)}.
        of-int(mu(int(d'))) * f((n div d) div d')))
      apply (rule setsum-cong2)
      apply (insert prems)
      apply (drule-tac x = n div x in spec)
      apply (subgoal-tac 0 < n div x)
      apply force
      apply (rule nat-pos-div-dvd-gr-0)
      apply auto
      done
    also have ... = f n
      apply (subst mu-inversion-nat2)
      apply (rule prems)
      apply (rule refl)
      done
    finally show ?thesis by (rule sym)
  qed
qed

lemma mu-inversion-nat3: 0 < (n::nat) ⟹ g n =
  (∑ d:{}0..n}. (∑ d':{}0..(n div d)}.
    of-int(mu(int d)) * g((n div d) div d')))
    (is 0 < n ⟹ g n = ?sum)
proof −
  assume 0 < n
  then have ?sum = (∑ c∈{}0..n}. ∑ d∈{d. d dvd c}.
    of-int (mu (int d)) * g (n div d div (c div d)))
    by (subst general-inversion-nat2, assumption, simp)
  also have ... = (∑ c∈{}0..n}. ∑ d∈{d. d dvd c}.
    g(n div c) * of-int (mu (int d)))
    apply (rule setsum-cong2)
```

apply (*rule setsum-cong2*)
apply (*subst nat-div-div-eq-div*)
apply *force*
apply *force*
apply (*simp add*: *mult-ac*)
done
also have ... = ($\sum$ c∈{}0..n}. g(n *div* c) * ($\sum$ d∈{d. d *dvd* c}.
of-int (*mu* (*int* d))))
apply (*rule setsum-cong2*)
apply (*rule setsum-const-times*)
done
also have ... = ($\sum$ c∈{}0..n}. g(n *div* c) * (*if* c = 1 *then* 1
else 0))
apply (*rule setsum-cong2*)
apply (*subst moebius-prop-nat-general*)
apply *auto*
done
also have ... = g n
apply (*subst mu-aux2*)
apply (*rule prems*)
apply *simp*
done
finally show *?thesis* **by** (*rule sym*)
qed

lemma *mu-inversion-nat4*: 0 < (n::*nat*) ⟹ f n =
($\sum$ d:{}0..n}. ($\sum$ d′:{}0..n *div* d}.
of-int(*mu*(*int* d′)) * f((n *div* d) *div* d′)))
(**is** 0 < n ⟹ f n = *?sum*)
proof −
assume 0 < n
then have *?sum* = ($\sum$ c:{}0..n}. $\sum$ d∈{d. d *dvd* c}.
of-int (*mu* (*int* (c *div* d))) * f (n *div* d *div* (c *div* d)))
by (*subst general-inversion-nat2*, *assumption*, *simp*)
also have ... = ($\sum$ c∈{}0..n}. $\sum$ d∈{d. d *dvd* c}.
(*of-int* (*mu* (*int* d)) * f (n *div* (c *div* d) *div* d)))
apply (*rule setsum-cong2*)
apply (*subst general-inversion-nat1*)
apply *clarsimp*
apply (*rule setsum-cong2*)
apply (*subst nat-div-div*)
apply *auto*
done
also have ... = ($\sum$ c∈{}0..n}. $\sum$ d∈{d. d *dvd* c}.
(f (n *div* c) * *of-int* (*mu* (*int* d))))

apply (*rule setsum-cong2*)
apply (*rule setsum-cong2*)
apply (*subst div-mult2-eq* [*THEN sym*])
apply (*subst mult-commute*)**backback**
apply (*subst nat-dvd-mult-div*)
apply *clarsimp*
apply (*erule nat-pos-dvd-pos*)
apply *assumption*
apply *simp*
apply (*simp add*: *mult-ac*)
done
also have $\ldots = (\sum c{\in}\{)0..n\}.\ f(n\ div\ c) * (\sum d{\in}\{d.\ d\ dvd\ c\}.$
of-int (*mu* (*int d*))))
apply (*rule setsum-cong2*)
apply (*rule setsum-const-times*)
done
also have $\ldots = (\sum c{\in}\{)0..n\}.\ f(n\ div\ c) * (if\ c = 1\ then\ 1$
else 0))
apply (*rule setsum-cong2*)
apply (*subst moebius-prop-nat-general*)
apply *auto*
done
also have $\ldots = f\ n$
apply (*subst mu-aux2*)
apply (*rule prems*)
apply *simp*
done
finally show *?thesis* **by** (*rule sym*)
qed

end

26 Operations on sets and functions

theory *SetsAndFunctions* = *Complex*:

26.1 Basic definitions

instance *set* :: (*times*)*times*
by *intro-classes*

instance *fun* :: (*type*,*times*)*times*
by *intro-classes*

defs (**overloaded**)
func-times: $f * g == (\lambda x.\ f\ x * g\ x)$
set-times:$A * B == \{c.\ \exists\, a{\in}A.\ \exists\, b{\in}B.\ c = a * b\}$

instance *set* :: (*plus*)*plus*
by *intro-classes*

instance *fun* :: (*type*,*plus*)*plus*
by *intro-classes*

defs (**overloaded**)
func-plus: $f + g == (\lambda x.\ f\ x + g\ x)$
set-plus: $A + B == \{c.\ \exists\, a{\in}A.\ \exists\, b{\in}B.\ c = a + b\}$

instance *fun* :: (*type*,*minus*)*minus*
by *intro-classes*

defs (**overloaded**)
func-minus: $-\ f == (\lambda x.\ -\ f\ x)$
func-diff-minus: $(f - g)\ x == f\ x - g\ x$

theorem *func-diff-minus2*: $((f{::}'a \Rightarrow 'b{::}ring) - g) = (f + -g)$
apply(*rule ext*)
by(*auto simp add*: *func-minus func-diff-minus func-plus diff-minus*)

instance *fun* :: (*type*,*zero*)*zero*
by *intro-classes*

instance *set* :: (*zero*)*zero*
by(*intro-classes*)

defs (**overloaded**)
func-zero: $0{::}(('a{::}type) \Rightarrow ('b{::}zero)) == \lambda x.\ 0$
set-zero: $0{::}('a{::}zero)set == \{0\}$

instance *fun* :: (*type*,*one*)*one*
by *intro-classes*

instance *set* :: (*one*)*one*
by *intro-classes*

defs (**overloaded**)

func-one: *1*::((*'a*::*type*) ⇒ (*'b*::*one*)) == λ*x*. *1*
set-one: *1*::(*'a*::*one*)*set* == {*1*}

constdefs
elt-set-plus :: *'a*::*plus* ⇒ *'a set* ⇒ *'a set* (**infixl** +*o* *70*)
a +*o* *B* == {*c*. ∃ *b*∈*B*. *c* = *a* + *b*}

elt-set-times :: *'a*::*times* ⇒ *'a set* ⇒ *'a set* (**infixl** ∗*o* *80*)
a ∗*o* *B* == {*c*. ∃ *b*∈*B*. *c* = *a* ∗ *b*}

syntax
elt-set-eq :: *'a* ⇒ *'a set* ⇒ *bool* (**infix** =*o* *50*)
set-set-eq :: *'a set* ⇒ *'a set* ⇒ *bool* (**infix** =*s* *50*)

translations
x =*o* *A* => *x* ∈ *A*
A =*s* *B* => *A* ⊆ *B*

syntax
elt-set-eq :: *'a* ⇒ *'a set* ⇒ *bool* (**infix** = *50*)
set-set-eq :: *'a set* ⇒ *'a set* ⇒ *bool* (**infix** = *50*)
elt-set-plus :: *'a* ⇒ *'a set* ⇒ *'a set* (**infix** + *70*)
elt-set-times :: *'a* ⇒ *'a set* ⇒ *'a set* (**infix** × *80*)

syntax (**output**)
elt-set-plus :: *'a* ⇒ *'a set* ⇒ *'a set* (**infix** + *70*)
elt-set-times :: *'a* ⇒ *'a set* ⇒ *'a set* (**infix** ∗ *80*)

instance *fun* :: (*type*,*plus-ac0*)*plus-ac0*
apply *intro-classes*
apply(*auto simp add*: *func-zero func-plus*)
apply(*rule ext*)
apply(*auto simp add*: *plus-ac0.axioms*)
apply(*rule ext*)
by(*auto simp add*: *plus-ac0.axioms plus-ac0-left-commute*)

instance *set* :: (*plus-ac0*)*plus-ac0*
apply *intro-classes*
apply (*auto simp add*: *func-plus set-plus plus-ac0*)
apply(*rule-tac x* = *b* **in** *bexI*)
apply(*rule-tac x* = *a* **in** *bexI*)
apply(*auto simp add*: *plus-ac0.axioms*)
apply(*rule-tac x* = *b* **in** *bexI*)
apply(*rule-tac x* = *a* **in** *bexI*)
apply(*auto simp add*: *plus-ac0.axioms*)

apply(*rule-tac* $x = aa$ **in** *bexI*)
apply(*rule-tac* $x = b + ba$ **in** *exI*)
apply(*auto simp add*: *plus-ac0.axioms*)
apply(*rule-tac* $x = ba$ **in** *bexI*)
apply(*rule-tac* $x = b$ **in** *bexI*)
apply(*auto simp add*: *plus-ac0*)
apply(*rule-tac* $x = a + aa$ **in** *exI*)
by(*auto simp add*: *plus-ac0 func-zero func-plus set-zero*)

instance *fun* :: (*type*,*ring*)*ring*
apply *intro-classes*
apply(*auto simp add*: *func-plus func-times func-minus func-diff-minus ext*
func-one func-zero ring.axioms)
apply(*rule ext*)
apply(*simp add*: *add-ac*)
apply(*rule ext*)
apply(*simp add*: *add-ac*)
apply(*rule ext*)
apply(*simp add*: *mult-ac*)
apply(*rule ext*)
apply(*simp add*: *mult-ac*)
apply(*simp add*: *ring-distrib ring.axioms*)
apply(*frule fun-cong*)
apply *auto*
apply (*rule ext*)
apply (*simp add*: *func-diff-minus2 func-plus func-minus*)
done

lemma *func-diff*: $(\%x.\ (f\ x)::'a::ring) - (\%x.\ g\ x) = (\%x.\ f\ x - g\ x)$
by (*simp add*: *diff-minus func-minus func-plus*)

26.2 Basic properties

lemma *set-plus-intro* [*intro*!]: $[\![\ a \in C;\ b \in D]\!] \Longrightarrow a + b \in C + D$
by (*auto simp add*: *set-plus*)

lemma *set-plus-intro2* [*intro*!]: $b \in C \Longrightarrow a + b \in a + C$
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-plus-rearrange*: $((a::'a::plus\text{-}ac0) + C) +$
$(b + D) = (a + b) + (C + D)$
apply (*auto simp add*: *elt-set-plus-def set-plus plus-ac0*)
apply (*rule-tac* $x = ba + bb$ **in** *exI*)
apply (*auto simp add*: *plus-ac0*)
apply (*rule-tac* $x = aa + a$ **in** *exI*)

by (*auto simp add*: *plus-ac0*)

lemma *set-plus-rearrange2*: (*a*::'*a*::*plus-ac0*) + (*b* + *C*) =
(*a* + *b*) + *C*
by (*auto simp add*: *elt-set-plus-def plus-ac0*)

lemma *set-plus-rearrange3*: ((*a*::'*a*::*plus-ac0*) + *B*) + *C* =
a + (*B* + *C*)
apply (*auto simp add*: *elt-set-plus-def set-plus*)
apply (*blast intro*: *plus-ac0*)
apply (*rule-tac x* = *a* + *aa* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac x* = *aa* **in** *bexI*)
apply *auto*
apply (*rule-tac x* = *ba* **in** *bexI*)
by (*auto simp add*: *plus-ac0*)

theorem *set-plus-rearrange4*: *C* + ((*a*::'*a*::*plus-ac0*) +*o* *D*) = *a* +*o* (*C* + *D*)
apply (*auto intro*!: *subsetI simp add*: *elt-set-plus-def set-plus plus-ac0*)
apply (*rule-tac x* = *aa* + *ba* **in** *exI*)
by (*auto simp add*: *plus-ac0*)

theorems *set-plus-rearranges* = *set-plus-rearrange set-plus-rearrange2*
set-plus-rearrange3 set-plus-rearrange4

lemma *set-plus-mono* [*intro*!]: *C* ⊆ *D* ⟹ *a* + *C* ⊆ *a* + *D*
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-plus-mono2* [*intro*]: *C* ⊆ *D* ⟹ *E* ⊆ *F* ⟹ *C* + *E* ⊆ *D* + *F*
by (*auto simp add*: *set-plus*)

lemma *set-plus-mono3* [*intro*]: *a* ∈ *C* ⟹ *a* + *D* ⊆ *C* + *D*
by (*auto simp add*: *elt-set-plus-def set-plus*)

lemma *set-plus-mono4* [*intro*]: (*a*::'*a*::*plus-ac0*) ∈ *C* ⟹
a + *D* ⊆ *D* + *C*
by (*auto simp add*: *elt-set-plus-def set-plus plus-ac0*)

lemma *set-plus-mono5*: *a* : *C* ==> *B* <= *D* ==> *a* +*o* *B* <= *C* + *D*
apply (*subgoal-tac a* +*o* *B* <= *a* +*o* *D*)
apply (*erule order-trans*)
apply (*erule set-plus-mono3*)
apply (*erule set-plus-mono*)
done

lemma *set-plus-mono-b*: $C \subseteq D \Longrightarrow x \in a + C$
$\Longrightarrow x \in a + D$
apply (*frule set-plus-mono*)
apply *auto*
done

lemma *set-plus-mono2-b*: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow x \in C + E \Longrightarrow$
$x \in D + F$
apply (*frule set-plus-mono2*)
prefer *2*
apply *force*
apply *assumption*
done

lemma *set-plus-mono3-b*: $a \in C \Longrightarrow x \in a + D \Longrightarrow x \in C + D$
apply (*frule set-plus-mono3*)
apply *auto*
done

lemma *set-plus-mono4-b*: $(a::'a::plus\text{-}ac0) \in C \Longrightarrow$
$x \in a + D \Longrightarrow x \in D + C$
apply (*frule set-plus-mono4*)
apply *auto*
done

lemma *set-zero-plus* [*simp*]: $(0::'a::plus\text{-}ac0) + C = C$
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-zero-plus2*: $(0::'a::plus\text{-}ac0) \in A \Longrightarrow B \subseteq A + B$
apply (*auto intro*!: *subsetI simp add*: *set-plus*)
apply (*rule-tac* $x = 0$ **in** *bexI*)
apply (*rule-tac* $x = x$ **in** *bexI*)
by (*auto simp add*: *plus-ac0*)

lemma *set-plus-imp-minus*: $(a::'a::ring) \in b +o C \Longrightarrow (a - b) \in C$
by (*auto simp add*: *elt-set-plus-def add-ac diff-minus*)

lemma *set-minus-imp-plus*: $(a::'a::ring) - b \in C \Longrightarrow a \in b +o C$
apply (*auto simp add*: *elt-set-plus-def add-ac diff-minus*)
apply (*subgoal-tac* $a = (a + - b) + b$)
apply (*rule bexI*, *assumption*, *assumption*)
by (*auto simp add*: *add-ac*)

lemma *set-minus-plus*: $((a::'a::ring) - b \in C) = (a \in b +o C)$
by (*rule iffI*, *rule set-minus-imp-plus*, *assumption*,

rule set-plus-imp-minus, *assumption*)

lemma *set-times-intro* [*intro*!]: $[\![a \in C; b \in D]\!] \Longrightarrow a * b \in C * D$
by (*auto simp add*: *set-times*)

lemma *set-times-intro2* [*intro*!]: $b \in C \Longrightarrow a * b \in a \times C$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-rearrange*: $((a::'a::ring) \times C) *$
$(b \times D) = (a * b) \times (C * D)$
apply (*auto simp add*: *elt-set-times-def set-times*)
apply (*rule-tac x* = *ba* $*$ *bb* **in** *exI*)
apply (*auto simp add*: *mult-ac*)
apply (*rule-tac x* = *aa* $*$ *a* **in** *exI*)
by (*auto simp add*: *mult-ac*)

lemma *set-times-rearrange2*: $(a::'a::ring) \times (b \times C) =$
$(a * b) \times C$
by (*auto simp add*: *elt-set-times-def mult-ac*)

lemma *set-times-rearrange3*: $((a::'a::ring) \times B) * C =$
$a \times (B * C)$
apply (*auto simp add*: *elt-set-times-def set-times*)
apply (*blast intro*: *mult-ac*)
apply (*rule-tac x* = *a* $*$ *aa* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac x* = *aa* **in** *bexI*)
apply *auto*
apply (*rule-tac x* = *ba* **in** *bexI*)
by (*auto simp add*: *mult-ac*)

theorem *set-times-rearrange4*: $C * ((a::'a::ring) *o D) = a *o (C * D)$
apply (*auto intro*!: *subsetI simp add*: *elt-set-times-def set-times*
mult-ac)
apply (*rule-tac x* = *aa* $*$ *ba* **in** *exI*)
by (*auto simp add*: *mult-ac*)

theorems *set-times-rearranges* = *set-times-rearrange set-times-rearrange2*
set-times-rearrange3 set-times-rearrange4

lemma *set-times-mono* [*intro*!]: $C \subseteq D \Longrightarrow a \times C \subseteq a \times D$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-mono2* [*intro*]: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow C * E \subseteq D * F$
by (*auto simp add*: *set-times*)

lemma *set-times-mono3* [*intro*]: $a \in C \Longrightarrow a \times D \subseteq C * D$
by (*auto simp add*: *elt-set-times-def set-times*)

lemma *set-times-mono4* [*intro*]: $(a::'a::times\text{-}ac1) \in C \Longrightarrow$
$a \times D \subseteq D * C$
by (*auto simp add*: *elt-set-times-def set-times times-ac1*)

lemma *set-times-mono5*: $a : C ==> B <= D ==> a *o B <= C * D$
apply (*subgoal-tac* $a *o B <= a *o D$)
apply (*erule order-trans*)
apply (*erule set-times-mono3*)
apply (*erule set-times-mono*)
done

lemma *set-times-mono-b*: $C \subseteq D \Longrightarrow x \in a \times C$
$\Longrightarrow x \in a \times D$
apply (*frule set-times-mono*)
apply *auto*
done

lemma *set-times-mono2-b*: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow x \in C * E \Longrightarrow$
$x \in D * F$
apply (*frule set-times-mono2*)
prefer *2*
apply *force*
apply *assumption*
done

lemma *set-times-mono3-b*: $a \in C \Longrightarrow x \in a \times D \Longrightarrow x \in C * D$
apply (*frule set-times-mono3*)
apply *auto*
done

lemma *set-times-mono4-b*: $(a::'a::times\text{-}ac1) \in C \Longrightarrow$
$x \in a \times D \Longrightarrow x \in D * C$
apply (*frule set-times-mono4*)
apply *auto*
done

lemma *set-one-times* [*simp*]: $(1::'a::ring) \times C = C$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-plus-distrib*: $(a::'a::ring) \times (b + C) =$
$(a * b) + (a \times C)$

by (*auto simp add*: *elt-set-plus-def elt-set-times-def ring-distrib*)

lemma *set-times-plus-distrib2*: (*a*::'*a*::*ring*) × (*B* + *C*) =
(*a* × *B*) + (*a* × *C*)
apply (*auto simp add*: *set-plus elt-set-times-def ring-distrib*)
apply *blast*
apply (*rule-tac x* = *b* + *bb* **in** *exI*)
by (*auto simp add*: *ring-distrib*)

lemma *set-times-plus-distrib3*: ((*a*::'*a*::*ring*) + *C*) ∗ *D* ⊆
a × *D* + *C* ∗ *D*
apply (*auto intro*!: *subsetI simp add*:
elt-set-plus-def elt-set-times-def set-times
set-plus ring-distrib)
by *auto*

theorems *set-times-plus-distribs* = *set-times-plus-distrib*
set-times-plus-distrib2

lemma *set-neg-intro*: (*a*::'*a*::*ring*) : (− *1*) ∗*o* *C* ==>
− *a* : *C*
by (*auto simp add*: *elt-set-times-def*)

lemma *set-neg-intro2*: (*a*::'*a*::*ring*) : *C* ==>
− *a* : (− *1*) ∗*o* *C*
by (*auto simp add*: *elt-set-times-def*)

end

27 Facts about the real numbers

theory *RealLib* = *Complex* + *RingLib* + *FiniteLib* + *SetsAndFunctions*:
— Clean this up!
— Eliminate duplicates
— Generalize theorems about reals to rings

27.1 Casting to reals

lemma *real-eq-of-nat*: *real* = *of-nat*
apply (*rule ext*)
apply (*unfold real-of-nat-def*)
apply (*rule refl*)

done

lemma *real-eq-of-int*: *real* = *of-int*
apply (*rule ext*)
apply (*unfold real-of-int-def*)
apply (*rule refl*)
done

lemma *real-nat-zero* [*simp*]: *real* (*0*::*nat*) = *0*
by (*simp add*: *real-of-nat-def*)

lemma *real-nat-one* [*simp*]: *real* (*1*::*nat*) = *1*
by *simp*

lemma *le-imp-real-of-nat-le*: (*x*::*nat*) <= *y* ==>
real x <= *real y*
by *auto*

lemma *divide-div-aux*: *0* < *d* ⟹ (*real* (*x*::*nat*)) / (*real d*) =
real (*x div d*) + (*real* (*x mod d*)) / (*real d*)
proof −
assume *0* < *d*
have *x* = (*x div d*) ∗ *d* + *x mod d*
by *auto*
then have *real x* = *real* (*x div d*) ∗ *real d* + *real*(*x mod d*)
by (*simp only*: *real-of-nat-mult* [*THEN sym*] *real-of-nat-add* [*THEN sym*])
then have *real x* / *real d* = ... / *real d*
by *simp*
then show *?thesis*
by (*auto simp add*: *add-divide-distrib ring-eq-simps prems*)
qed

lemma *nat-dvd-real-div*: *0* < (*d*::*nat*) ==> *d dvd n* ==>
real(*n div d*) = *real n* / *real d*
apply (*frule divide-div-aux* [*of d n*])
apply *simp*
apply (*subst dvd-eq-mod-eq-0* [*THEN sym*])
apply *assumption*
done

27.2 Misc theorems about limits and infinite sums

lemma *LIMSEQ-diff-const*: *f* −−−−> *a* ==> (%*n*.(*f n* − *b*)) −−−−> *a* − *b*
apply (*subgoal-tac* %*n*. (*f n* − *b*) == %*n*. (*f n* − (%*n*. *b*) *n*))
apply (*subgoal-tac* (%*n*. *b*) −−−−> *b*)

by (*auto simp add*: *LIMSEQ-diff LIMSEQ-const*)

lemma *LIMSEQ-ignore-initial-segment*: *f* −−−−> *a*
==> (%*n*. *f*(*n* + *k*)) −−−−> *a*
apply (*unfold LIMSEQ-def*)
apply (*clarify*)
apply (*drule-tac x* = *r* **in** *spec*)
apply (*clarify*)
apply (*rule-tac x* = *no* + *k* **in** *exI*)
by *auto*

lemma *LIMSEQ-offset*: (%*x*. *f* (*x* + *k*)) −−−−> *a* ==>
f −−−−> *a*
apply (*unfold LIMSEQ-def*)
apply *clarsimp*
apply (*drule-tac x* = *r* **in** *spec*)
apply *clarsimp*
apply (*rule-tac x* = *no* + *k* **in** *exI*)
apply *clarsimp*
apply (*drule-tac x* = *n* − *k* **in** *spec*)
apply (*frule mp*)
apply *arith*
apply *simp*
done

lemma *LIMSEQ-diff-approach-zero*:
g −−−−> *L* ==> (%*x*. *f x* − *g x*) −−−−> *0* ==>
f −−−−> *L*
apply (*drule LIMSEQ-add*)
apply *assumption*
apply *simp*
done

lemma *LIMSEQ-diff-approach-zero2*:
f −−−−> *L* ==> (%*x*. *f x* − *g x*) −−−−> *0* ==>
g −−−−> *L*
apply (*drule LIMSEQ-diff*)
apply *assumption*
apply *simp*
done

lemma *sums-split-initial-segment*: *f sums s* ==>
(%*n*. *f*(*n* + *k*)) *sums* (*s* − *sumr 0 k f*)
apply (*unfold sums-def*)
apply (*simp add*: *sumr-offset*)

apply (*rule LIMSEQ-diff-const*)
apply (*rule LIMSEQ-ignore-initial-segment*)
by *assumption*

lemma *summable-ignore-initial-segment*: *summable f ==>*
summable (%*n*. *f*(*n* + *k*))
apply (*unfold summable-def*)
by (*auto intro*: *sums-split-initial-segment*)

lemma *suminf-minus-initial-segment*: *summable f ==>*
suminf f = *s ==> suminf* (%*n*. *f*(*n* + *k*)) = *s* − *sumr 0 k f*
apply (*frule summable-ignore-initial-segment*)
apply (*rule sums-unique* [*THEN sym*])
apply (*frule summable-sums*)
apply (*rule sums-split-initial-segment*)
by *auto*

lemma *suminf-split-initial-segment*: *summable f ==>*
suminf f = *sumr 0 k f* + *suminf* (%*n*. *f*(*n* + *k*))
by (*auto simp add*: *suminf-minus-initial-segment*)

lemma *sums-const-times*: *f sums a* ⟹ (%*x*. *c* ∗ *f x*) *sums* (*c* ∗ *a*)
apply (*unfold sums-def*)
apply (*subgoal-tac* (λ*n*. *sumr 0 n* (λ*x*. *c* ∗ *f x*)) = (%*n*. *c* ∗ *sumr 0 n f*))
apply (*erule ssubst*)
apply (*rule LIMSEQ-mult*)
apply (*rule LIMSEQ-const*)
apply *assumption*
apply (*rule ext*)
apply (*rule sumr-mult* [*THEN sym*])
done

lemma *summable-const-times*: *summable f* ⟹ *summable* (%*x*. *c* ∗ *f x*)
apply (*unfold summable-def*)
apply (*auto intro*: *sums-const-times*)
done

lemma *suminf-const-times*: *summable f* ⟹ *suminf* (%*x*. *c* ∗ *f x*) = *c* ∗ *suminf f*
apply (*rule sym*)
apply (*rule sums-unique*)
apply (*rule sums-const-times*)
apply (*erule summable-sums*)
done

27.3 Facts about sumr, setsum, and setprod

lemma *sumr-cong* [*rule-format*]: (*ALL y.* (*y* < *x* −−> *f y* = *g y*)) −−>
sumr 0 x f = *sumr 0 x g*
by (*induct x*, *simp-all*)

lemma *sumr-cong* [*rule-format*]: (*ALL y.* (*y* < *x* −−> *f y* = *g y*)) −−>
sumr 0 x f = *sumr 0 x g*
by (*induct x*, *simp-all*)

lemma *sumr-le-cong* [*rule-format*]: (*ALL y.* (*y* < *x* −−> *f y* <= *g y*)) −−>
sumr 0 x f <= *sumr 0 x g*
apply (*induct x*)
apply *simp*
apply *auto*
apply (*erule add-mono*)
apply *auto*
done

lemma *sumr-ge-zero-cong* [*rule-format*]: (*ALL y.* (*y* < *x* −−> *0* <= *g y*))
−−> *0* <= *sumr 0 x g*
apply (*subgoal-tac sumr 0 x* (%*x.* *0*) = *0*)
apply (*erule subst*)
apply (*clarify*)
apply (*rule sumr-le-cong*)
apply *auto*
done

lemma *sumr-split-add-le*: *n* <= *p* ==> *sumr 0 n f* + *sumr n p f* = *sumr 0 p f*
apply(*case-tac n* < *p*)
apply(*simp only*: *sumr-split-add*)
apply(*subgoal-tac n* = *p*)
by(*auto*)

lemma *setsum-sumr*: *setsum f* {*0..x*::*nat*(} = *sumr 0 x f*
apply (*induct-tac x*)
apply (*auto*)
apply (*simp add*: *atLeastLessThan-def*)
apply (*simp add*: *atLeastLessThan-def*)
apply (*simp add*: *lessThan-Suc*)
done

lemma *setsum-sumr2*: *setsum f* {*0..x*} = *sumr 0* (*x*+*1*) *f*
apply (*subgoal-tac* {*0..x*} = {*0..x*+*1*(})
apply (*erule ssubst*)

```
  apply (rule setsum-sumr)
  apply auto
done

lemma setsum-sumr3: setsum f {)0..x} = sumr 0 x (%n. f(n + 1))
  apply (subgoal-tac setsum f {)0..x} = setsum (%n. f(n+1)) {0..x(})
  apply (erule ssubst)
  apply (rule setsum-sumr)
  apply (rule setsum-reindex-cong')
  prefer 4
  apply simp
  apply simp
  apply (simp add: inj-on-def)
  apply (auto simp add: image-def)
  apply (rule-tac x = xa - 1 in bexI)
  apply auto
  apply arith
done

lemma setsum-sumr4: (∑ i=1..n. f i) = sumr 0 n (%i. f(i + 1))
  apply (subst setsum-sumr3 [THEN sym])
  apply (rule setsum-cong)
  apply auto
done

lemma setsum-sumr5-aux: (∑ i=a..a+c. f i) = sumr a (a+c+1) f
  apply (induct-tac c)
  apply simp
  apply simp
  apply (subgoal-tac {a..Suc(a+n)} = {a..a+n} Un {Suc(a+n)})
  apply (erule ssubst)back
  apply (subst setsum-Un-disjoint)
  apply auto
done

lemma setsum-sumr5: (∑ i=a..b. f i) = sumr a (b+1) f
  apply (case-tac b < a)
  apply (subgoal-tac {a..b} = {})
  apply simp
  apply simp
  apply (subgoal-tac b = a + (b - a))
  apply (erule ssubst)
  apply (rule setsum-sumr5-aux)
  apply arith
done
```

lemma *setsum-singleton*: *setsum f {n} = f n*
apply *auto*
done

lemma *setsum-bound*: *finite A ==> ALL x:A. (f(x) <= c) ==>*
*setsum f A <= c * card A*
apply (*induct set*: *Finites*, *auto*)
done

lemma *real-of-nat-setsum*: *real (setsum (f::'a => nat) A) =*
setsum (real o f) A
apply (*subst real-eq-of-nat*)
apply (*rule setsum-of-nat*)
done

lemma *real-card-eq-setsum*: *finite A ==> real (card A) = setsum (%x.1) A*
apply (*subst card-eq-setsum*)
apply *assumption*
apply (*subst real-of-nat-setsum*)
apply (*unfold o-def*)
apply *simp*
done

lemma *setprod-real-of-int*: *real (setprod (f::'a ⇒ int) A) =*
setprod (%x. real(f x)) A
apply (*subst real-eq-of-int*)
apply (*simp add*: *setprod-of-int o-def*)
done

lemma *sumr-shift*: *sumr 0 n (%n. f(m + n)) = sumr m (m + n) f*
by (*induct-tac n*, *simp-all*)

lemma *sumr-zero-to-two*: *sumr 0 2 f = f 0 + f 1*
proof −
have *2 = Suc (Suc 0)*
by *simp*
hence *sumr 0 2 f = sumr 0 (Suc (Suc 0)) f*
by (*rule subst*, *simp*)
thus *?thesis*
by *simp*
qed

lemma *sums-zero*: (%*x*. *0*) *sums 0*
apply (*unfold sums-def*)

apply *simp*
apply (*rule LIMSEQ-const*)
done

lemma *suminf-zero*: *suminf* (%*x*. *0*) = *0*
apply (*rule sym*)
apply (*rule sums-unique*)
apply (*rule sums-zero*)
done

lemma *summable-zero*: *summable* (%*x*. *0*)
apply (*rule sums-summable*)
apply (*rule sums-zero*)
done

lemma *sums-neg*: *f sums s* ⟹ (− *f*) *sums* (− *s*)
by (*simp add*: *sums-def func-minus sumr-minus LIMSEQ-minus*)

lemma *summable-neg*: *summable f* ⟹ *summable* (− *f*)
by (*auto simp add*: *summable-def intro*: *sums-neg*)

lemma *summable-neg2*: *summable f* ⟹ *summable* (%*x*. − *f x*)
by (*frule summable-neg*, *unfold func-minus*)

lemma *suminf-neg*: *summable f* ==> *suminf* (− *f*) = − (*suminf f*)
apply (*rule sym*)
apply (*rule sums-unique*)
apply (*rule sums-neg*)
apply (*erule summable-sums*)
done

27.4 Help for calculations with reals (a mess!)

lemma *realpow-two2*: (*x*::*real*) ∗ *x* = *x*^*2*
proof −
have (*x*::*real*) ∗ *x* = *x*^(*Suc* (*Suc 0*))
by (*rule realpow-two* [*THEN sym*])
also have ... = *x*^*2*
proof −
have *Suc* (*Suc 0*) = *2* **by** *simp*
thus *?thesis* **by** (*rule ssubst*, *simp*)
qed
finally show *?thesis* **.**
qed

lemma *real-mult-le-one-le*: **assumes** *0* <= (*x*::*real*) **and** *0* <= *y* **and** *y* <= *1*
  shows *x* * *y* <= *x*
proof −
 from *prems* **have** *x* * *y* <= *x* * *1*
  by (*intro mult-left-mono*)
 thus *?thesis* **by** *auto*
qed

lemma *real-le-one-mult-le*: **assumes** *0* <= (*x*::*real*) **and** *0* <= *y* **and** *y* <= *1*
  shows *y* * *x* <= *x*
proof −
 from *prems* **have** *y* * *x* <= *1* * *x*
  by (*intro mult-right-mono*)
 thus *?thesis* **by** *auto*
qed

lemma *real-mult-less-one-less*:
  (*0*::*real*) < *x* ==> *x* < *1* ==> *0* < *y* ==> *y* * *x* < *y*
proof −
 assume (*0*::*real*) < *x* **and** *x* < *1* **and** *0* < *y*
 have *y* * *x* < *y* * *1*
  by (*auto simp only*: *real-mult-less-mono2 prems*)
 thus *?thesis*
  by *simp*
qed

lemma *real-less-one-mult-less*:
  (*0*::*real*) < *x* ==> *x* < *1* ==> *0* < *y* ==> *x* * *y* < *y*
proof −
 assume (*0*::*real*) < *x* **and** *x* < *1* **and** *0* < *y*
 have *x* * *y* < *1* * *y*
  by (*auto simp only*: *mult-strict-right-mono prems*)
 thus *?thesis*
  by *simp*
qed

lemma *real-inverse-le-swap*: *0* < (*r*::*real*) ==> *r* <= *x* ==>
  inverse x <= *inverse r*
proof −
 assume *0* < (*r*::*real*)
 assume *r* <= *x*
 then have *r* < *x* | *r* = *x*
  by *auto*
 with *prems* **show** *inverse x* <= *inverse r*
 proof (*elim disjE*)

```
    assume 0 < r and r < x
    then have inverse x < inverse r
      by (intro less-imp-inverse-less)
    then show ?thesis by auto
  next assume r = x
     thus inverse x <= inverse r by auto
  qed
qed

lemma real-inverse-nat-le-one: 1 <= (n::nat) ==> inverse (real n) <= 1
proof −
  assume 1 <= (n::nat)
  then have real (1::nat) <= real n
    by (auto simp only: real-of-nat-le-iff)
  then have 1 <= real n by simp
  then have inverse (real n) <= inverse 1
    by (simp only: real-inverse-le-swap)
  thus ?thesis by auto
qed

lemma real-le-mult-imp-div-pos-le: assumes a: 0 < y and
    b: (x::real) <= y * z shows x / y <= z
proof −
  from b have x <= y * z .
  also from a have x = y * (x / y)
    by auto
  finally have y * (x / y) <= y * z .
  with a show ?thesis
    by (simp only: real-mult-le-cancel-iff2)
qed

lemma real-mult-le-imp-le-div-pos: assumes a: 0 < y and
    b: (x::real) * y <= z shows x <= z / y
proof −
  from b a have x * y  <= (z / y) * y by auto
  with a show ?thesis
    by (simp only: real-mult-le-cancel-iff1)
qed

lemma real-le-mult-imp-le-div-pos2: assumes a: 0 < y and
    b: y * (x::real) <= z shows x <= z / y
proof −
  from b a have y * x  <= y * (z / y) by auto
  with a show ?thesis
    by (simp only: real-mult-le-cancel-iff2)
```

qed

lemma *real-mult-less-imp-less-div-pos*: **assumes** *a*: *0 < y* **and**
b: (*x*::*real*) * *y < z* **shows** *x < z / y*
proof −
from *b a* **have** *c*: *x * y < (z / y) * y*
by *simp*
show *?thesis*
apply (*rule mult-right-less-imp-less*)
apply (*rule c*)
apply (*rule order-less-imp-le*, *rule a*)
done
qed

lemma *real-div-pos-le-mono*: (*x*::*real*) <= *y* ==> *0 < z* ==> *x / z <= y / z*
by (*unfold real-divide-def*, *auto*)

lemma *real-div-neg-le-anti-mono*: (*x*::*real*) <= *y* ==> *z < 0* ==>
y / z <= x / z
by (*simp add*: *neg-divide-le-eq*)

lemma *real-div-pos-less-mono*: (*x*::*real*) *< y* ==> *0 < z* ==> *x / z < y / z*
by (*auto simp add*: *real-divide-def*)

lemma *real-pos-div-less-anti-mono*: *0 <* (*x*::*real*) ==> *x < y* ==> *0 < z*
==> *z / y < z / x*
apply (*unfold real-divide-def*)
apply (*rule real-mult-inverse-cancel2*)
apply *auto*
done

lemma *real-pos-div-le-mono*: *0 <* (*x*::*real*) ==> *x <= y* ==> *0 <= z*
==> *z / y <= z / x*
apply (*unfold real-divide-def*)
apply (*rule mult-left-mono*)
apply (*rule le-imp-inverse-le*)
by (*assumption+*)

lemma *real-one-div-le-anti-mono*: *0 <* (*x*::*real*) ==> *x <= y* ==>
1 / y <= 1 / x
by (*rule real-pos-div-le-mono* [*of x y 1*], *auto*)

lemma *real-ge-zero-plus-gt-zero-is-gt-zero*: (*0*::*real*) <= *x* ==>
0 < y ==> *0 < x + y*
proof −

assume *a*: (*0*::*real*) <= *x* **and** *b*: (*0*::*real*) < *y*
have (*0*::*real*) = *0* + *0* **by** *simp*
also from *a b* **have** ... < *x* + *y*
by (*rule real-add-le-less-mono*)
finally show *?thesis* **.**
qed

lemma *real-gt-zero-plus-ge-zero-is-gt-zero*: (*0*::*real*) < *x* ==>
0 <= *y* ==> *0* < *x* + *y*
proof −
assume *a*: (*0*::*real*) < *x* **and** *b*: (*0*::*real*) <= *y*
have (*0*::*real*) = *0* + *0* **by** *simp*
also from *a b* **have** ... < *x* + *y*
by (*rule real-add-less-le-mono*)
finally show *?thesis* **.**
qed

lemma *real-ge-zero-div-gt-zero* [*simp*]: (*0*::*real*) <= *x* ==> *0* < *y* ==>
0 <= *x*/*y*
by (*rule real-mult-le-imp-le-div-pos*, *auto*)

lemma *real-div-mult-simp*: (*z*::*real*) ~= *0*
==> (*x* / *z* = *y*) = (*x* = *z* * *y*)
by *auto*

lemma *real-div-mult-simp2*: (*z*::*real*) ~= *0*
==> (*y* = *x* / *z*) = (*x* = *z* * *y*)
by *auto*

lemma *real-minus-divide-distrib*: ((*x*::*real*) − *y*) / *z* = *x* / *z* − *y* / *z*
proof −
have *x* − *y* = *x* + −*y*
by *auto*
then have (*x* − *y*) / *z* = (*x* + −*y*) / *z*
by (*rule ssubst*, *simp*)
also have ... = *x* / *z* + (−*y*) / *z*
by (*rule add-divide-distrib*)
also have ... = *x* / *z* − *y* / *z*
by *simp*
finally show *?thesis* **.**
qed

lemma *real-mult-div-cancel2*: (*k*::*real*) ~= *0* ==> (*m* * *k*) / (*n* * *k*) = *m* / *n*
proof −
assume *k* ~= *0*

have $m * k = k * m$ **by** *auto*
moreover have $n * k = k * n$ **by** *auto*
ultimately have $(m * k) / (n * k) = (k * m) / (k * n)$
by (*auto simp only*:)
also have $... = m / n$
by (*rule mult-divide-cancel-left*)
finally show *?thesis* **.**
qed

lemma *unused1*: $((x::real) / y) / z = (x / z) / y$
by *auto*

lemma *unused2*: $0 < x ==> ((x::real) / y) / x = 1 / y$
by *auto*

lemma *real-neg-squared-eq-pos-squared*: $(- (x::real))\hat{}2 = x\hat{}2$
by (*auto simp add*: *realpow-two2* [*THEN sym*])

lemma *real-x-times-x-squared*: $(x::real) * x\hat{}2 = x\hat{}3$
proof −
have $3 = Suc\ 2$ **by** *auto*
hence $x\hat{}3 = x\hat{}(Suc\ 2)$ **by** *simp*
also have $... = x * x\hat{}2$ **by** (*rule realpow-Suc*)
finally show *?thesis* **by** *simp*
qed

lemma *real-one-over-pow*: $(x::real) \sim= 0 \Longrightarrow 1 / x\hat{}n = (1 / x)\hat{}n$
apply (*simp add*: *real-divide-def*)
by (*rule power-inverse*)

lemma *real-nat-ge-zero* [*simp*]: $0 \leq real\ (n::nat)$
by *auto*

lemma *real-nat-plus-one-gt-zero* [*simp*]: $0 < real\ (n::nat) + 1$
apply (*rule real-ge-zero-plus-gt-zero-is-gt-zero*)
by *auto*

lemma *real-one-over-nat-plus-one-gt-zero* [*simp*]: $0 < 1 / (real\ (n::nat) + 1)$
apply (*rule real-mult-less-imp-less-div-pos*)
apply (*rule real-nat-plus-one-gt-zero*)
by *simp*

lemma *real-one-over-nat-plus-one-ge-zero* [*simp*]:
$0 \leq 1 / (real\ (n::nat) + 1)$
apply (*rule order-less-imp-le*)

by (*rule real-one-over-nat-plus-one-gt-zero*)

lemma *real-nat-plus-one*: *real* ((*n*::*nat*) + *1*) = *real n* + *1*
by (*auto intro*: *real-of-nat-Suc*)

lemma *real-frac-add*: [|*b* ~= *0*;*d* ~= *0*|] ==> *a*/*b* + *c*/(*d*::*real*) = ((*a*∗*d* + *c*∗*b*) / (*d*∗*b*))
by(*auto simp add*: *add-divide-distrib*)

lemma *div-ge-1*: [|*0* < *a*;*a* <= (*b*::*real*)|] ==> *1* <= *b* / *a*
apply(*auto simp add*: *real-divide-def*)
apply(*rotate-tac 1*)
apply(*rule order-trans*[*of 1 a* ∗ *inverse a*])
apply(*simp add*: *Ring-and-Field.right-inverse*)
apply (*rule mult-right-mono*)
apply *auto*
done

lemma *real-one-over-pos*: *0* < (*x*::*real*) ==> *0* < *1* / *x*
by (*rule real-mult-less-imp-less-div-pos*, *auto*)

lemma *real-fraction-le*: *0* < (*w*::*real*) ==> *0* < *z* ==> *x* ∗ *w* <= *y* ∗ *z* ==>
x / *z* <= *y* / *w*
apply (*rule real-le-mult-imp-div-pos-le*)
apply *assumption*
apply *simp*
apply (*rule real-mult-le-imp-le-div-pos*)
apply *assumption*
apply (*simp add*: *mult-commute*)
done

lemma *real-fraction-le2*: *0* <= *x* ==> (*x*::*real*) <= *y* ==> *0* < *w* ==> *w* <= *z* ==>
x / *z* <= *y* / *w*
apply (*rule real-fraction-le*)
apply *auto*
apply (*rule mult-mono*)
apply *auto*
done

27.5 Casting quotients to real

theorem *nat-div-real-div*[*rule-format*]:
0 <= *real* (*n*::*nat*) / *real* (*x*) − *real* (*n div x*) &
real (*n*) / *real* (*x*) − *real* (*n div x*) <= *1*

apply(*auto*)
apply(*case-tac x = 0*)
apply(*force simp add*: *DIVISION-BY-ZERO-DIV real-divide-def INVERSE-ZERO*)
apply(*rule real-mult-le-imp-le-div-pos*)
apply *force*
apply *simp*
apply (*subst real-of-nat-mult* [*THEN sym*])
apply (*subst real-of-nat-le-iff*)
apply (*subst mult-commute*)
apply (*subst mult-div-cancel*)
apply *force*
apply(*case-tac x = 0*)
apply *simp*
apply (*simp add*: *compare-rls*)
apply (*rule real-le-mult-imp-div-pos-le*)
apply *force*
apply (*simp add*: *ring-distrib*)
apply (*subst real-of-nat-mult* [*THEN sym*])
apply (*subst real-of-nat-add* [*THEN sym*])
apply (*subst real-of-nat-le-iff*)
apply (*subst mult-div-cancel*)
apply (*subgoal-tac n mod x <= x*)
apply *arith*
apply (*rule order-less-imp-le*)
apply (*erule mod-less-divisor*)
done

theorems *nat-div-real-div1 = nat-div-real-div* [*THEN conjunct1*]
theorems *nat-div-real-div2 = nat-div-real-div* [*THEN conjunct2*]

lemma *real-nat-div-le-real-div*: *real* (*n div x*) *<= real* (*n*::*nat*) */ real x*
apply (*insert nat-div-real-div1* [*of n x*])
apply *simp*
done

27.6 Floor and ceiling

lemma *floor-bound*: *real*(*x*::*nat*) *<= k ==> x <= nat*(*floor*(*k*))
apply (*drule floor-le2*)
apply (*simp only*: *floor-real-of-nat*)
apply (*subgoal-tac nat* (*int x*) *<= nat* (*floor k*))
apply (*simp add*: *nat-int*)
apply (*case-tac x = 0*)
apply (*subgoal-tac 0 <= k*)
apply (*simp add*: *nat-le-eq-zle*)

apply (*subgoal-tac int x = 0*)
apply (*simp add*: *real-of-int-le-iff* [*THEN sym*])
apply (*auto simp add*: *real-of-int-floor-le*)
apply (*subgoal-tac real*(*floor k*) <= *k*)
apply (*erule order-trans*)
apply *simp*
apply (*simp add*: *real-of-int-floor-le*)
apply (*subgoal-tac nat*(*int x*) <= *nat*(*floor k*))
apply (*simp add*: *nat-int*)
apply (*subgoal-tac 0 < int x | 0 <= floor k*)
apply (*frule nat-le-eq-zle* [*THEN sym*])
apply *force*
apply *force*
done

lemma *real-int-nat1*: *real*(*int*(*nat*(*x*))) = *real*(*nat*(*x*))
apply (*simp only*: *real-of-int-real-of-nat*)
done

lemma *real-int-nat*: *0 <= x ==> real*(*x*) = *real*(*nat*(*x*))
apply (*subgoal-tac real*(*int*(*nat*(*x*))) = *real*(*nat*(*x*)))
apply *force*
apply (*rule real-int-nat1*)
done

lemma *real-nat-floor*: *0 <= x ==> real*(*nat*(*floor*(*x*))) <= *x*
apply (*subgoal-tac 0 <= floor*(*x*))
apply (*subgoal-tac real*(*nat*(*floor*(*x*))) = *real*(*floor*(*x*)))
apply (*erule ssubst*)
apply (*auto simp add*: *real-of-int-floor-le*)
apply (*simp add*: *real-int-nat*)
apply (*subgoal-tac floor 0 <= floor x*)
apply (*simp add*: *floor-zero*)
apply (*rule floor-le2*)
by *simp*

lemma *real-upper-bound*: *0 < r ⟹ r <= real*(*nat*(*ceiling*(*r*)))
apply (*subgoal-tac real*(*nat*(*ceiling*(*r*))) = *real*(*ceiling*(*r*)))
apply *simp*
apply (*subgoal-tac real*(*ceiling*(*r*)) = *real*(*int*(*nat*(*ceiling*(*r*)))))
apply (*erule ssubst*)
apply (*subst real-of-int-real-of-nat*)
apply (*rule refl*)
apply *simp*
apply (*insert real-of-int-ceiling-ge* [*of r*])

apply (*subgoal-tac 0 <= real(ceiling(r))*)
apply *simp*
apply (*subgoal-tac 0 < real(ceiling(r))*)
apply *force*
apply (*erule order-less-le-trans*)
apply *assumption*
done

constdefs
natfloor :: *real* ⇒ *nat*
natfloor x == *nat*(*floor x*)

lemma *real-of-nat-eq-real-of-int*: *0 <= x* ⟹ *real*(*nat x*) = *real x*
apply (*subgoal-tac real x = real*(*int*(*nat*(*x*))))
apply (*erule ssubst*)
apply (*rule real-of-int-real-of-nat* [*THEN sym*])
apply *simp*
done

lemma *floor-ge-zero-ge-zero*: *0 <= x* ⟹ *0 <= floor*(*x*)
apply (*subgoal-tac 0 = floor*(*0*))
apply (*erule ssubst*)
apply (*erule floor-le2*)
apply *simp*
done

lemma *real-natfloor-le*: *0 <= x* ⟹ *real*(*natfloor*(*x*)) <= *x*
apply (*unfold natfloor-def*)
apply (*subst real-of-nat-eq-real-of-int*)
apply (*erule floor-ge-zero-ge-zero*)
apply *simp*
done

lemma *real-natfloor-plus-one-ge*: *x <= real*(*natfloor x*) + *1*
apply (*case-tac 0 <= x*)
apply (*unfold natfloor-def*)
apply (*subst real-of-nat-eq-real-of-int*)
apply (*erule floor-ge-zero-ge-zero*)
apply (*rule real-of-int-floor-add-one-ge*)
apply *simp*
done

lemma *real-of-int-floor-gt-diff-one* [*simp*]: *r* − *1 < real*(*floor r*)
apply (*simp add*: *floor-def Least-def*)
apply (*insert real-lb-ub-int* [*of r*], *safe*)

apply (*rule theI2*)
apply (*auto intro*: *lemma-floor*)
done

lemma *real-of-int-floor-add-one-gt* [*simp*]: *r* < *real*(*floor r*) + *1*
apply (*insert real-of-int-floor-gt-diff-one* [*of r*])
apply (*auto simp del*: *real-of-int-floor-gt-diff-one*)
done

lemma *real-of-nat-floor-add-one-gt* [*simp*]: *r* < *real* (*natfloor r*) + *1*
apply (*unfold natfloor-def*)
apply (*case-tac 0* <= *r*)
apply (*subst real-of-nat-eq-real-of-int*)
apply (*erule floor-ge-zero-ge-zero*)
apply *simp*
apply *simp*
done

lemma *floor-add* [*simp*]: *floor* (*x* + *real a*) = *floor x* + *a*
apply (*subgoal-tac floor*(*x* + *real a*) <= *floor x* + *a*)
apply (*subgoal-tac floor x* + *a* <= *floor*(*x* + *real a*))
apply *force*
apply (*subgoal-tac floor x* + *a* < *floor* (*x* + *real a*) + *1*)
apply *arith*
apply (*subst real-of-int-less-iff* [*THEN sym*])
apply *simp*
apply (*subgoal-tac x* + *real a* < *real*(*floor*(*x* + *real a*)) + *1*)
apply (*subgoal-tac real* (*floor x*) <= *x*)
apply *arith*
apply (*rule real-of-int-floor-le*)
apply (*rule real-of-int-floor-add-one-gt*)
apply (*subgoal-tac floor* (*x* + *real a*) < *floor x* + *a* + *1*)
apply *arith*
apply (*subst real-of-int-less-iff* [*THEN sym*])
apply *simp*
apply (*subgoal-tac real*(*floor*(*x* + *real a*)) <= *x* + *real a*)
apply (*subgoal-tac x* < *real*(*floor x*) + *1*)
apply *arith*
apply (*rule real-of-int-floor-add-one-gt*)
apply (*rule real-of-int-floor-le*)
done

lemma *floor-subtract* [*simp*]: *floor* (*x* − *real a*) = *floor x* − *a*
apply (*subst diff-minus*)+
apply (*subst real-of-int-minus*)

apply (*rule floor-add*)
done

lemma *real-le-floor* [*intro*]: *real a <= x ==> a <= floor x*
apply (*subgoal-tac a < floor x + 1*)
apply *arith*
apply (*subst real-of-int-less-iff* [*THEN sym*])
apply *simp*
apply (*insert real-of-int-floor-add-one-gt* [*of x*])
apply *arith*
done

lemma *real-le-natfloor* [*intro*]: *real a <= x ==> a <= natfloor x*
apply (*unfold natfloor-def*)
apply (*subgoal-tac nat* (*int a*) *<= nat* (*floor x*))
apply *simp*
apply (*subst nat-le-eq-zle*)
apply (*subgoal-tac int 0 <= floor x*)
apply *force*
apply (*rule real-le-floor*)
apply *force*
apply (*rule real-le-floor*)
apply *simp*
done

lemma *natfloor-add* [*simp*]: *0 <= x ==> natfloor*(*x + real a*) *= natfloor x + a*
apply (*unfold natfloor-def*)
apply (*subgoal-tac real a = real* (*int a*))
apply (*erule ssubst*)
apply (*subst floor-add*)
apply (*subst nat-add-distrib*)
apply *auto*
done

lemma *natfloor-subtract* [*simp*]: *real a <= x ==>*
natfloor(*x − real a*) *= natfloor x − a*
apply (*unfold natfloor-def*)
apply (*subgoal-tac real a = real* (*int a*))
apply (*erule ssubst*)
apply (*subst floor-subtract*)
apply (*subst nat-diff-distrib*)
apply *auto*
done

lemma *natfloor-ge-zero-lt-one*: *0 <= x ==> x < 1 ==> natfloor x = 0*

```
  apply (subgoal-tac 0 <= natfloor x)
  apply (subgoal-tac natfloor x < 1)
  apply arith
  apply (unfold natfloor-def)
  apply simp
  apply (rule floor-eq4)
  apply auto
done

lemma natfloor-div-nat: 1 <= x ==> 0 < y ==>
  natfloor (x / real y) = natfloor x div y
proof -
  assume 1 <= (x::real) and 0 < (y::nat)
  have natfloor x = (natfloor x) div y * y + (natfloor x) mod y
    by simp
  then have a: real(natfloor x) = real ((natfloor x) div y) * real y +
    real((natfloor x) mod y)
    by (simp only: real-of-nat-add [THEN sym] real-of-nat-mult [THEN sym])
  have x = real(natfloor x) + (x - real(natfloor x))
    by simp
  then have x = real ((natfloor x) div y) * real y +
      real((natfloor x) mod y) + (x - real(natfloor x))
    by (simp add: a)
  then have x / real y = ... / real y
    by simp
  also have ... = real((natfloor x) div y) + real((natfloor x) mod y) /
    real y + (x - real(natfloor x)) / real y
    by (auto simp add: ring-distrib ring-eq-simps add-divide-distrib
      diff-divide-distrib prems)
  finally have natfloor (x / real y) = natfloor(...) by simp
  also have ... = natfloor(real((natfloor x) mod y) /
    real y + (x - real(natfloor x)) / real y + real((natfloor x) div y))
    by (simp add: add-ac)
  also have ... = natfloor(real((natfloor x) mod y) /
    real y + (x - real(natfloor x)) / real y) + (natfloor x) div y
    apply (rule natfloor-add)
    apply (rule nonneg-plus-nonneg)
    apply (rule real-ge-zero-div-gt-zero)
    apply force
    apply (simp add: prems)
    apply (rule real-ge-zero-div-gt-zero)
    apply simp
    apply (rule real-natfloor-le)
    apply (auto simp add: prems)
    apply (insert prems, arith)
```

```
    done
  also have natfloor(real((natfloor x) mod y) /
    real y + (x - real(natfloor x)) / real y) = 0
    apply (rule natfloor-ge-zero-lt-one)
    apply (rule nonneg-plus-nonneg)
    apply (rule real-ge-zero-div-gt-zero)
    apply force
    apply (force simp add: prems)
    apply (rule real-ge-zero-div-gt-zero)
    apply simp
    apply (rule real-natfloor-le)
    apply (auto simp add: prems)
    apply (insert prems, arith)
    apply (simp add: add-divide-distrib [THEN sym])
    apply (subst pos-divide-less-eq)
    apply (force simp add: prems)
    apply simp
    apply (subgoal-tac real y = real y - 1 + 1)
    apply (erule ssubst)
    apply (rule add-le-less-mono)
    apply (simp add: compare-rls)
    apply (subgoal-tac real(natfloor x mod y) + 1 =
      real(natfloor x mod y + 1))
    apply (erule ssubst)
    apply (subst real-of-nat-le-iff)
    apply (subgoal-tac natfloor x mod y < y)
    apply arith
    apply (rule mod-less-divisor)
    apply assumption
    apply auto
    apply (simp add: compare-rls)
    apply (subst add-commute)
    apply (rule real-of-nat-floor-add-one-gt)
    done
  finally show ?thesis
    by simp
qed

lemma nat-le-natfloor: 0 <= x ==> y <= natfloor x ==> real y <= x
  apply (rule order-trans)
  apply (subgoal-tac real y <= real (natfloor x))
  apply assumption
  apply force
  apply (rule real-natfloor-le)
  apply assumption
```

done

lemma *natfloor-real-id* [*simp*]: *natfloor* (*real n*) = *n*
by (*unfold natfloor-def*, *simp*)

lemma *natfloor-neg*: *x* <= *0* ==> *natfloor x* = *0*
apply (*unfold natfloor-def*)
apply (*subgoal-tac floor x* <= *0*)
apply *simp*
apply (*subgoal-tac floor x* < *1*)
apply *arith*
apply (*subgoal-tac real* (*floor x*) < *real 1*)
apply *force*
apply (*rule order-le-less-trans*)
apply (*rule real-of-int-floor-le*)
apply *simp*
done

lemma *ge-natfloor-plus-one-imp-gt*: *natfloor z* + *1* <= *n* ==> *z* < *real n*
apply (*subgoal-tac z* < *real*(*natfloor z*) + *1*)
apply *arith*
apply (*rule real-of-nat-floor-add-one-gt*)
done

lemma *natfloor-zero* [*simp*]: *natfloor 0* = *0*
apply (*subgoal-tac 0* = *real* (*0*::*nat*))
apply (*erule ssubst*)
apply (*rule natfloor-real-id*)
apply *simp*
done

lemma *natfloor-one* [*simp*]: *natfloor 1* = *1*
apply (*subgoal-tac 1* = *real* (*1*::*nat*))
apply (*erule ssubst*)
apply (*rule natfloor-real-id*)
apply *simp*
done

lemma *natfloor-mono*: *x* <= *y* ==> *natfloor x* <= *natfloor y*
apply (*case-tac 0* <= *x*)
apply (*subst natfloor-def*)+
apply (*subst nat-le-eq-zle*)
apply *force*
apply (*erule floor-le2*)
apply (*subst natfloor-neg*)**back**

apply *arith*
apply *force*
done

lemma *natfloor-plus-one*: *0 <= x ==> natfloor(x + 1) = natfloor x + 1*
apply (*subgoal-tac natfloor (x + 1) = natfloor(x + real (1::nat))*)
apply (*erule ssubst*)
apply (*erule natfloor-add*)
apply *simp*
done

27.7 powr and ln

lemma *zero-le-powr* [*iff*]: *0 <= x powr y*
apply (*rule order-less-imp-le*)
apply *simp*
done

lemma *powr-realpow*: *0 < x ==> x powr (real n) = x^n*
apply (*induct n*)
apply *simp*
apply (*subgoal-tac real(Suc n) = real n + 1*)
apply (*erule ssubst*)
apply (*subst powr-add*)
apply *simp*
apply *simp*
done

lemma *powr-realpow2*: *0 <= x ==> 0 < n ==> x^n = (if (x = 0) then 0*
else x powr (real n))
apply (*case-tac x = 0*)
apply *simp*
apply *simp*
apply (*rule powr-realpow* [*THEN sym*])
apply *simp*
done

lemma *ln-pwr*: *0 < x ==> 0 < y ==> ln(x powr y) = y * ln x*
apply (*unfold powr-def*)
apply *simp*
done

lemma *ln-bound*: *1 <= x ==> ln x <= x*
apply (*subgoal-tac ln(1 + (x - 1)) <= x - 1*)
apply *simp*

apply (*rule ln-add-one-self-le-self*)
apply *simp*
done

lemma *powr-mono*: *a <= b ==> 1 <= x ==> x powr a <= x powr b*
apply (*case-tac x = 1*)
apply *simp*
apply (*case-tac a = b*)
apply *simp*
apply (*rule order-less-imp-le*)
apply (*rule powr-less-mono*)
apply *auto*
done

lemma *ge-one-powr-ge-zero*: *1 <= x ==> 0 <= a ==> 1 <= x powr a*
apply (*subst powr-zero-eq-one* [*THEN sym*])
apply (*rule powr-mono*)
apply *assumption+*
done

lemma *power-less-mono2*: *0 < a ==> 0 < x ==> x < y ==> x powr a <*
y powr a
apply (*unfold powr-def*)
apply (*rule exp-less-mono*)
apply (*rule mult-strict-left-mono*)
apply (*subst ln-less-cancel-iff*)
apply *assumption*
apply (*rule order-less-trans*)
prefer *2*
apply *assumption+*
done

lemma *power-mono2*: *0 <= a ==> 0 < x ==> x <= y ==> x powr a <=*
y powr a
apply (*case-tac a = 0*)
apply *simp*
apply (*case-tac x = y*)
apply *simp*
apply (*rule order-less-imp-le*)
apply (*rule power-less-mono2*)
apply *auto*
done

lemma *ln-powr-bound*: *1 <= x ==> 0 < a ==> ln x <= (x powr a) / a*
apply (*rule real-le-mult-imp-le-div-pos2*)

apply *assumption*
apply (*subst ln-pwr* [*THEN sym*])
apply *auto*
apply (*rule ln-bound*)
apply (*erule ge-one-powr-ge-zero*)
apply (*erule order-less-imp-le*)
done

lemma *ln-powr-bound2*: *1 < x ==> 0 < a ==> (ln x) powr a <= (a powr a) * x*
proof −
assume *1 < x* **and** *0 < a*
then have *ln x <= (x powr (1 / a)) / (1 / a)*
apply (*intro ln-powr-bound*)
apply (*erule order-less-imp-le*)
apply (*erule real-one-over-pos*)
done
also have *... = a * (x powr (1 / a))*
by *simp*
finally have *(ln x) powr a <= (a * (x powr (1 / a))) powr a*
apply (*intro power-mono2*)
apply (*rule order-less-imp-le*, *rule prems*)
apply (*rule ln-gt-zero*)
apply (*rule prems*)
apply *assumption*
done
also have *... = (a powr a) * ((x powr (1 / a)) powr a)*
apply (*rule powr-mult*)
apply (*rule prems*)
apply (*rule powr-gt-zero*)
done
also have *(x powr (1 / a)) powr a = x powr ((1 / a) * a)*
by (*rule powr-powr*)
also have *... = x*
apply *simp*
apply (*subgoal-tac a ~= 0*)
apply (*insert prems*, *auto*)
done
finally show *?thesis*.
qed

lemma *power-less-mono2-neg*: *a < 0 ==> 0 < x ==> x < y ==> y powr a < x powr a*
apply (*unfold powr-def*)
apply (*rule exp-less-mono*)

```
  apply (rule mult-strict-left-mono-neg)
  apply (subst ln-less-cancel-iff)
  apply assumption
  apply (rule order-less-trans)
  prefer 2
  apply assumption+
done

lemma LIMSEQ-neg-powr: 0 < s ==> (%x. (real x) powr - s) ----> 0
  apply (unfold LIMSEQ-def)
  apply clarsimp
  apply (rule-tac x = natfloor(r powr (1 / - s)) + 1 in exI)
  apply clarify
  proof -
    fix r fix n
    assume 0 < s and 0 < r and natfloor (r powr (1 / - s)) + 1 <= n
    have r powr (1 / - s) < real(natfloor(r powr (1 / - s))) + 1
      by (rule real-of-nat-floor-add-one-gt)
    also have ... = real(natfloor(r powr (1 / -s)) + 1)
      by simp
    also have ... <= real n
      apply (subst real-of-nat-le-iff)
      apply (rule prems)
      done
    finally have r powr (1 / - s) < real n.
    then have real n powr (- s) < (r powr (1 / - s)) powr - s
      apply (intro power-less-mono2-neg)
      apply (auto simp add: prems)
      done
    also have ... = r
      by (simp add: powr-powr prems less-imp-neq [THEN not-sym])
    finally show real n powr - s < r.
  qed

lemma powr-divide2: x powr a / x powr b = x powr (a - b)
  apply (simp add: powr-def)
  apply (subst exp-diff [THEN sym])
  apply (simp add: left-diff-distrib)
done

lemma ln-x-over-x-mono: exp 1 <= x ==> x <= y ==> (ln y / y) <= (ln x /
x)
proof -
  assume exp 1 <= x and x <= y
  have a: 0 < x and b: 0 < y
```

apply (*insert prems*)
apply (*subgoal-tac 0 < exp 1*)
apply *arith*
apply *auto*
apply (*subgoal-tac 0 < exp 1*)
apply *arith*
apply *auto*
done
have $x * ln\ y - x * ln\ x = x * (ln\ y - ln\ x)$
by (*simp add*: *ring-eq-simps*)
also have $... = x * ln(y\ /\ x)$
apply (*subst ln-div*)
apply (*rule b*, *rule a*, *rule refl*)
done
also have $y\ /\ x = (x + (y - x))\ /\ x$
by *simp*
also have $... = 1 + (y - x)\ /\ x$
apply (*simp only*: *add-divide-distrib*)
apply (*simp add*: *prems*)
apply (*insert a*, *arith*)
done
also have $x * ln(1 + (y - x)\ /\ x) <= x * ((y - x)\ /\ x)$
apply (*rule mult-left-mono*)
apply (*rule ln-add-one-self-le-self*)
apply (*rule real-ge-zero-div-gt-zero*)
apply (*simp-all add*: *prems a*)
apply (*rule order-less-imp-le*, *rule a*)
done
also have $... = y - x$
apply *simp*
apply (*insert a*, *arith*)
done
also have $... = (y - x) * ln\ (exp\ 1)$
by *simp*
also have $... <= (y - x) * ln\ x$
apply (*rule mult-left-mono*)
apply (*subst ln-le-cancel-iff*)
apply *force*
apply (*rule a*)
apply (*rule prems*)
apply (*simp add*: *prems*)
done
also have $... = y * ln\ x - x * ln\ x$
by (*rule left-diff-distrib*)
finally have $x * ln\ y <= y * ln\ x$

by *arith*
then have *ln y <= (y * ln x) / x*
apply (*subst pos-le-divide-eq*)
apply (*rule a*)
apply (*simp add*: *mult-ac*)
done
also have *... = y * (ln x / x)*
by *simp*
finally show *?thesis*
apply (*subst pos-divide-le-eq*)
apply (*rule b*)
apply (*simp add*: *mult-ac*)
done
qed

end

28 Big O notation

theory *BigO = SetsAndFunctions + RingLib + FiniteLib + RealLib*:

28.1 Preliminaries

Note: since the Big O library includes rules that demonstrate set inclusion, to use the automated reasoners with the library one should redeclare subsetI as an intro rule, instead of an intro! rule

Missing transitivity rule!

lemma *eq-in-trans* [*trans*]: $a = b \Longrightarrow b \in C \Longrightarrow a \in C$
by *auto*

28.2 Definitions

constdefs

bigo :: $('a \Rightarrow 'b::ordered\text{-}ring) => ('a => 'b)\ set$ ((*1O'*(-')))
$O(f::('a => 'b::ordered\text{-}ring)) == \{h.\ \exists c.\ \forall x.\ abs\ (h\ x) \leq c * abs\ (f\ x)\}$

bigoset :: $('a \Rightarrow 'b::ordered\text{-}ring)\ set \Rightarrow ('a \Rightarrow 'b)\ set$ ((*1O'*(-')))
$O(S::('a \Rightarrow 'b::ordered\text{-}ring)\ set) ==$
$\{h.\ \exists f \in S.\ \exists c.\ \forall x.\ abs(h\ x) \leq (c * abs(f\ x))\}$

28.3 Basic properties

lemma *bigo-pos-const*: $(\exists\, c.\ \forall\, x.\ (abs\ (h\ x)) \leq (c * (abs\ (f\ x))))$
$= (\exists\, c.\ 0 < (c::'a::ordered\text{-}ring) \wedge (\forall\, x.\ (abs(h\ x)) \leq (c * (abs\ (f\ x)))))$
apply(*auto*)
apply(*case-tac* $c = 0$)
apply(*simp*)
apply(*rule-tac* $x = 1$ **in** *exI*)
apply(*auto simp add*: *zero-less-one*)
apply(*rule-tac* $x = abs\ c$ **in** *exI*)
apply *auto*
apply (*subgoal-tac* $c * abs(f\ x) \leq abs\ c * abs\ (f\ x)$)
apply (*erule-tac* $x = x$ **in** *allE*)
apply *force*
apply (*rule mult-right-mono*)
apply (*rule abs-ge-self*)
apply (*rule abs-ge-zero*)
done

lemma *bigo-alt-def*: $O(f::('a \Rightarrow 'b::ordered\text{-}ring)) =$
$\{h.\ \exists\, c.\ (0 < c \wedge (\forall\, x.\ abs\ (h\ x) \leq c * abs\ (f\ x)))\}$
by (*auto simp add*: *bigo-def bigo-pos-const*)

lemma *bigoset-alt-def*: $O(S::('a \Rightarrow 'b::ordered\text{-}ring)\ set) =$
$\{h.\ \exists f{\in}S.\ \exists\, c.\ (0 < c\ \&\ (\forall\, x.\ abs(h\ x) \leq c * abs(f\ x)))\}$
by (*auto simp add*: *bigoset-def bigo-pos-const*)

lemma *bigoset-alt-def2*: $O(A::('a \Rightarrow 'b::ordered\text{-}ring)\ set) =$
$\{\ h.\ \exists f{\in}A.\ h \in O(f)\ \}$
by (*simp add*: *bigoset-def bigo-def*)

lemma *bigo-elt-subset* [*intro*]: $f \in O(g::('a \Rightarrow 'b::ordered\text{-}ring))$
$\Longrightarrow O(f) \subseteq O(g)$
apply (*auto simp add*: *bigo-alt-def*)
apply (*rule-tac* $x = ca * c$ **in** *exI*)
apply (*rule conjI*)
apply (*rule mult-pos*)
apply (*assumption*)+
apply (*rule allI*)
apply (*drule-tac* $x = xa$ **in** *spec*)+
apply (*subgoal-tac* $ca * abs(f\ xa) \leq ca * (c * abs(g\ xa))$)
apply (*erule order-trans*)
apply (*simp add*: *mult-ac*)
apply (*rule mult-left-mono*, *assumption*)
by (*rule order-less-imp-le*, *assumption*)

lemma *bigoset-elt-subset*: $f \in O(A::('a \Rightarrow 'b::ordered\text{-}ring)\ set)$
$\Longrightarrow O(f) \subseteq O(A)$
apply (*auto simp add*: *bigo-alt-def bigoset-alt-def*)
apply (*rule-tac* $x = fa$ **in** *bexI*)
apply (*rule-tac* $x = ca * c$ **in** *exI*)
apply (*rule conjI*)
apply (*rule mult-pos*)
apply (*assumption*)+
apply (*rule allI*)
apply (*drule-tac* $x = xa$ **in** *spec*)+
apply (*subgoal-tac* $ca * abs(f\ xa) \leq ca * (c * abs(fa\ xa))$)
apply (*erule order-trans*)
apply (*simp add*: *mult-ac*)
apply (*rule mult-left-mono*, *assumption*)
by (*rule order-less-imp-le*, *assumption*, *assumption*)

lemma *bigoset-elt-subset2*: $f \in A \Longrightarrow f \in O(A)$
apply (*auto simp add*: *bigoset-def*)
apply (*rule-tac* $x = f$ **in** *bexI*)
apply (*rule-tac* $x = 1$ **in** *exI*)
by *auto*

lemma *bigoset-mono*: $A \subseteq B \Longrightarrow O(A) \subseteq O(B)$
by (*auto simp add*: *bigoset-def*)

lemma *bigo-refl* [*intro*]: $f \in O(f)$
apply(*auto simp add*: *bigo-def*)
apply(*rule-tac* $x = 1$ **in** *exI*)
by *simp*

lemma *bigoset-refl*: $A \subseteq O(A)$
apply (*auto simp add*: *bigoset-def*)
apply (*rule-tac* $x = x$ **in** *bexI*)
apply (*rule-tac* $x = 1$ **in** *exI*)
by *auto*

lemma *bigo-bigo-refl*: $f \in O(O(f))$
apply (*insert bigo-refl* [*of f*])
apply (*insert bigoset-refl* [*of* $O(f)$])
by (*erule set-rev-mp*, *assumption*)

lemma *bigo-bigo-eq*: $O(O(f)) = O(f::('a => 'b::ordered\text{-}ring))$
apply(*subgoal-tac* $(O(O(f)) = O(f)) = (O(f) = O(O(f)))$,*erule ssubst*)
apply(*simp add*: *bigo-def bigoset-def bigo-refl bigo-pos-const*)

apply(*auto*)
apply(*simp-all only*: *bigo-pos-const*[*THEN sym*])
apply(*rule-tac* $x = f$ **in** *exI*)
apply(*auto*)
apply(*rule-tac* $x = 1$ **in** *exI*)
apply(*auto*)
apply(*rule-tac* $x = ca * c$ **in** *exI*)
apply(*auto*)
apply(*erule-tac* $x = xa$ **in** *allE*)
apply(*erule-tac* $x = xa$ **in** *allE*)
apply(*subgoal-tac* $ca * abs(fa\ xa) \leq ca * c * abs(f\ xa)$)
apply *force*
apply(*subgoal-tac* $ca * abs\ (fa\ xa) \leq ca * (c * abs\ (f\ xa))$)
apply(*auto simp add*: *mult-assoc mult-left-mono*)
apply(*rule mult-left-mono*)
by(*auto simp add*: *order-less-imp-le*)

lemma *bigo-zero*: $0 \in O(g::('a \Rightarrow 'b::ordered\text{-}ring))$
apply (*auto simp add*: *bigo-def func-zero*)
apply (*rule-tac* $x = 0$ **in** *exI*)
by *auto*

lemma *bigo-zero2*: $O(\lambda x.0) = \{\lambda x.0\}$
apply (*auto simp add*: *bigo-def*)
apply (*rule ext*)
apply *auto*
done

lemma *bigo-plus-self-subset* [*intro*]:
  $O((f::'a \Rightarrow 'b::ordered\text{-}ring)) + O(f) \subseteq O(f)$
apply (*auto simp add*: *bigo-alt-def bigoset-alt-def set-plus*)
apply (*rule-tac* $x = c + ca$ **in** *exI*)
apply *auto*
apply (*elim pos-plus-pos*)
apply *assumption*
apply (*simp add*: *func-plus*)
apply (*drule-tac* $x = x$ **in** *spec*)+
apply (*subgoal-tac abs* $(a\ x + b\ x) \leq abs\ (a\ x) + abs\ (b\ x)$)
apply (*erule order-trans*)
apply (*simp add*: *ring-distrib*)
apply (*erule add-mono*)
apply (*assumption*)
by (*rule abs-triangle-ineq*)

lemma *bigo-plus-idemp* [*simp*]: $O(f::('a \Rightarrow 'b::ordered\text{-}ring)) + O(f) = O(f)$

apply (*rule equalityI*)
apply (*rule bigo-plus-self-subset*)
apply (*rule set-zero-plus2*)
by (*rule bigo-zero*)

lemma *bigo-plus-subset-lemma*: $x \in O(f + g) \Longrightarrow x \in O(f) +$
$O(g::'a=>'b::ordered\text{-}ring)$
apply(*auto simp add*: *bigo-def bigo-pos-const func-plus set-plus*)
apply(*simp only*: *bigo-pos-const*[*THEN sym*])
apply(*rule-tac x* = %*n*. *if abs* (*g n*) ≤ (*abs* (*f n*)) *then x n else 0* **in** *exI*)
apply(*rule conjI*)
apply(*rule-tac x* = *c* + *c* **in** *exI*)
apply(*clarsimp*)
apply(*auto*)
apply(*subgoal-tac c* ∗ *abs* (*f xa* + *g xa*) ≤ (*c* + *c*) ∗ *abs* (*f xa*))
apply(*erule-tac x* = *xa* **in** *allE*)
apply(*erule order-trans*)
apply(*simp*)
apply(*subgoal-tac c* ∗ *abs* (*f xa* + *g xa*) ≤ *c* ∗ (*abs* (*f xa*) + *abs* (*g xa*)))
apply(*erule order-trans*)
apply(*simp add*: *ring-distrib*)
apply(*rule mult-left-mono*)
apply *assumption*
apply(*simp add*: *order-less-le*)
apply(*rule mult-left-mono*)
apply(*simp add*: *abs-triangle-ineq*)
apply(*simp add*: *order-less-le*)
apply(*subgoal-tac abs* (*0::'b*) = *0*)
apply(*erule ssubst*)
apply (*rule nonneg-times-nonneg*)
apply (*rule nonneg-plus-nonneg*)
apply *auto*
apply(*rule-tac x* = %*n*. *if* (*abs* (*f n*)) < *abs* (*g n*) *then x n else 0* **in** *exI*)
apply(*rule conjI*)
apply(*rule-tac x* = *c* + *c* **in** *exI*)
apply(*clarsimp*)
apply(*auto*)
apply(*subgoal-tac c* ∗ *abs* (*f xa* + *g xa*) ≤ (*c* + *c*) ∗ *abs* (*g xa*))
apply(*erule-tac x* = *xa* **in** *allE*)
apply(*erule order-trans*)
apply(*simp*)
apply(*subgoal-tac c* ∗ *abs* (*f xa* + *g xa*) ≤ *c* ∗ (*abs* (*f xa*) + *abs* (*g xa*)))
apply(*erule order-trans*)
apply(*simp add*: *ring-distrib*)
apply (*rule mult-left-mono*)

apply(*simp add*: *order-less-le*)
apply(*simp add*: *order-less-le*)
apply (*rule mult-left-mono*)
apply (*rule abs-triangle-ineq*)
apply(*simp add*: *order-less-le*)
apply (*rule nonneg-times-nonneg*)
apply (*rule nonneg-plus-nonneg*)
apply (*erule order-less-imp-le*)+
apply *simp*
apply (*rule ext*)
apply (*auto simp add*: *if-splits linorder-not-le*)
done

lemma *bigo-plus-subset* [*intro*]: $O(a + b) \subseteq O(a) + O(b::'a=>'b::ordered\text{-}ring)$
by (*rule subsetI*, *rule bigo-plus-subset-lemma*, *assumption*)

lemma *bigo-plus-subset2*: $O(f + A) \subseteq O(f) + O(A)$
apply (*auto simp add*: *bigoset-alt-def2 elt-set-plus-def set-plus*)
apply (*frule bigo-plus-subset-lemma*)
by (*auto simp add*: *set-plus*)

lemma *bigo-plus-subset3*: $O((A::('a \Rightarrow 'b::ordered\text{-}ring)\ set) + B) \subseteq$
$O(A) + O(B)$
apply (*auto simp add*: *bigoset-alt-def2 set-plus*)
apply (*frule bigo-plus-subset-lemma*)
by (*auto simp add*: *set-plus*)

lemma *bigo-plus-subset4* [*intro*]: $[\![A \subseteq O(f::('a \Rightarrow 'b::ordered\text{-}ring));$
$B \subseteq O(f)]\!] \Longrightarrow A + B \subseteq O(f)$
proof −
assume $A \subseteq O(f::('a \Rightarrow 'b::ordered\text{-}ring))$ **and** $B \subseteq O(f)$
then have $A + B \subseteq O(f) + O(f)$
by (*auto del*: *subsetI simp del*: *bigo-plus-idemp*)
also have $\ldots \subseteq O(f)$
by (*auto del*: *subsetI*)
finally show *?thesis*.
qed

lemma *bigo-plus-subset5*: $O((f::('a \Rightarrow 'b::ordered\text{-}ring)) +$
$O(g::('a \Rightarrow 'b))) \subseteq O(f) + O(g)$
proof −
have $O((f::('a \Rightarrow 'b::ordered\text{-}ring)) +$
$O(g::('a \Rightarrow 'b))) \subseteq O(f) + O(O(g))$
by (*rule bigo-plus-subset2*)
thus *?thesis* **by** (*simp add*: *bigo-bigo-eq*)

qed

lemma *bigo-plus-subset6*: $[\![\forall x.\ 0 \leq f\,x; \forall x.\ 0 \leq g\,x\ ]\!] \Longrightarrow$
$O(f + g) = O(f) + O(g)$
apply (*rule equalityI*)
apply (*rule bigo-plus-subset*)
apply (*simp add*: *bigo-alt-def set-plus func-plus*)
apply *clarify*
apply (*rule-tac* $x = max\ c\ ca$ **in** *exI*)
apply (*rule conjI*)
apply (*subgoal-tac* $c \leq max\ c\ ca$)
apply (*erule order-less-le-trans*)
apply *assumption*
apply (*rule le-maxI1*)
apply *clarify*
apply (*drule-tac* $x = xa$ **in** *spec*)+
apply (*subgoal-tac* $0 \leq f\,xa + g\,xa$)
apply (*simp add*: *ring-distrib abs-nonneg*)
apply (*subgoal-tac* $abs(a\ xa + b\ xa) \leq abs(a\ xa) + abs(b\ xa)$)
apply (*subgoal-tac* $abs(a\ xa) + abs(b\ xa) \leq$
$max\ c\ ca * f\,xa + max\ c\ ca * g\,xa$)
apply (*force*)
apply (*rule add-mono*)
apply (*subgoal-tac* $c * f\,xa \leq max\ c\ ca * f\,xa$)
apply (*force*)
apply (*rule mult-right-mono*)
apply (*rule le-maxI1*)
apply *assumption*
apply (*subgoal-tac* $ca * g\,xa \leq max\ c\ ca * g\,xa$)
apply (*force*)
apply (*rule mult-right-mono*)
apply (*rule le-maxI2*)
apply *assumption*
apply (*rule abs-triangle-ineq*)
apply (*rule nonneg-plus-nonneg*)
apply *assumption*+
done

lemma *bigo-elt-subset2* [*intro*]: $f \in g + O(h::('a \Rightarrow 'b::ordered\text{-}ring)) \Longrightarrow$
$O(f) \subseteq O(g) + O(h)$
proof −
assume $f \in g + O(h::('a \Rightarrow 'b::ordered\text{-}ring))$
then have $O(f) \subseteq O(g + O(h))$
apply (*intro bigoset-elt-subset*)
by (*rule bigoset-elt-subset2*)

also have $\ldots \subseteq O(g) + O(h)$
by (*rule bigo-plus-subset5*)
finally show *?thesis*.
qed

lemma *bigo-mult* [*intro*]: $O(f)*O(g) \subseteq O(f * g::('a => 'b::ordered\text{-}ring))$
apply(*auto simp add*: *bigo-def bigo-pos-const set-times func-times*)
apply(*rule-tac* $x = c * ca$ **in** *exI*)
apply(*rule allI*)
apply(*erule-tac* $x = x$ **in** *allE*)
apply(*erule-tac* $x = x$ **in** *allE*)
apply(*subgoal-tac* $c * ca * (abs(f\ x) * abs(g\ x)) = (c * abs(f\ x)) * (ca * abs(g\ x))$)
apply(*erule ssubst*)
apply (*rule mult-mono*)
apply *assumption+*
apply (*rule nonneg-times-nonneg*)
apply *auto*
apply (*simp add*: *mult-ac*)
done

For reals, can make this =

lemma *bigo-mult2* [*intro*]: $f \times O(g::('a \Rightarrow 'b::ordered\text{-}ring)) \subseteq$
$O(f * g)$
apply (*auto simp add*: *bigo-def elt-set-times-def func-times abs-mult*)
apply (*rule-tac* $x = c$ **in** *exI*)
apply *auto*
apply (*drule-tac* $x = x$ **in** *spec*)
apply (*subgoal-tac* $abs(f\ x) * abs(b\ x) \leq abs(f\ x) * (c * abs(g\ x))$)
apply (*force simp add*: *mult-ac*)
apply (*rule mult-left-mono*, *assumption*)
by (*rule abs-ge-zero*)

lemma *bigo-mult3*: [| $f : O(h::'a=>'b::ordered\text{-}ring)$; $g : O(j)$ |] ==> $f * g : O(h * j)$
apply(*subgoal-tac* $f * g : O(h) * O(j)$)
apply(*insert bigo-mult*[*of h j*])
apply(*simp only*: *subsetD*)
by(*auto simp add*: *bigo-def func-times set-times*)

lemma *bigo-mult4*[*intro*]:[| $f : k +o O(h::'a=>'b::ordered\text{-}ring)$ |] ==> $g * f : (g * k) +o O(g * h)$
apply(*simp add*: *bigo-def elt-set-plus-def func-times*)
apply(*clarsimp*)
apply(*rule-tac* $x = g * b$ **in** *exI*)

apply(*rule conjI*)
apply(*rule-tac* $x = c$ **in** *exI*)
apply(*clarsimp*)
apply(*simp add*: *func-times abs-mult*)
apply(*erule-tac* $x = x$ **in** *allE*)
apply(*subgoal-tac* $c * (abs\ (g\ x) * abs\ (h\ x)) = abs\ (g\ x) * (c * abs\ (h\ x))$)
apply(*erule ssubst*)
apply(*rule mult-left-mono*)
apply(*simp-all add*: *abs-ge-zero*)
apply(*simp add*: *times-ac1*)
by(*simp add*: *func-times func-plus ring-distrib*)

lemma *bigo-minus* [*intro*]: $f : O(g::'a => 'b::ordered\text{-}ring) ==> -f : O(g)$
apply(*auto simp add*: *bigo-def*)
apply(*rule-tac* $x = c$ **in** *exI*)
apply(*rule allI*)
apply(*erule-tac* $x = x$ **in** *allE*)
apply(*subgoal-tac abs* $((-f)\ x) = abs\ (f\ x)$)
apply(*erule ssubst*)
apply(*simp*)
apply(*auto simp add*: *func-minus*)
done

lemma *bigo-mult5*: *ALL x. f x ~= 0 ==>*
$O(f * g) <= (f::'a => ('b::ordered\text{-}field)) *o\ O(g)$
proof −
assume *ALL x. f x ~= 0*
show $O(f * g) <= f *o\ O(g)$
proof
fix *h*
assume $h : O(f * g)$
then have $(\%x.\ 1\ /\ (f\ x)) * h : (\%x.\ 1\ /\ f\ x) *o\ O(f * g)$
by *auto*
also have $... <= O((\%x.\ 1\ /\ f\ x) * (f * g))$
by (*rule bigo-mult2*)
also have $(\%x.\ 1\ /\ f\ x) * (f * g) = g$
apply (*simp add*: *func-times*)
apply (*rule ext*)
apply (*simp add*: *prems nonzero-divide-eq-eq mult-ac*)
done
finally have $(\%x.\ (1::'b)\ /\ f\ x) * h : O(g)$**.**
then have $f * ((\%x.\ (1::'b)\ /\ f\ x) * h) : f *o\ O(g)$
by *auto*
also have $f * ((\%x.\ (1::'b)\ /\ f\ x) * h) = h$
apply (*simp add*: *func-times*)

```
    apply (rule ext)
    apply (simp add: prems nonzero-divide-eq-eq mult-ac)
    done
  finally show h : f *o O(g).
 qed
qed

lemma bigo-mult6: ALL x. f x ~= 0 ==>
    O(f * g) = (f::'a => ('b::ordered-field)) *o O(g)
apply (rule equalityI)
apply (erule bigo-mult5)
apply (rule bigo-mult2)
done

lemma bigo-mult7: ALL x. f x ~= 0 ==>
    O(f * g) <= O(f::'a => ('b::ordered-field)) * O(g)
 apply (subst bigo-mult6)
 apply assumption
 apply (rule set-times-mono3)
 apply (rule bigo-refl)
done

lemma bigo-mult8: ALL x. f x ~= 0 ==>
    O(f * g) = O(f::'a => ('b::ordered-field)) * O(g)
apply (rule equalityI)
apply (erule bigo-mult7)
apply (rule bigo-mult)
done

lemma bigo-minus2: f : g + O(h::('a => ('b::ordered-ring))) ==>
-f : -g + O(h)
proof -
 assume f : g + O(h::('a => ('b::ordered-ring)))
 then have f - g : O(h)
   by (rule set-plus-imp-minus)
 then have -(f - g) : O(h)
   by (rule bigo-minus)
 also have -(f - g) = -f - -g
   by (simp add: diff-minus plus-ac0)
 finally have -f - -g : O(h).
 then show -f : -g + O(h)
   by (rule set-minus-imp-plus)
qed

lemma bigo-minus3: O(-f) = O(f::('a => 'b::ordered-ring))
```

by (*auto simp add*: *bigo-def func-minus abs-minus-cancel*)

declare *subsetI* [*rule del*, *intro*]

lemma *bigo-plus-absorb-lemma1* [*intro*]: $f \in O(g::('a \Rightarrow 'b::ordered\text{-}ring)) \Longrightarrow$
$f + O(g) \subseteq O(g)$
proof −
assume *a*: $f \in O(g)$
show $f + O(g) \subseteq O(g)$
proof −
have $f \in O(f)$ **by** *auto*
then have $f + O(g) \subseteq O(f) + O(g)$
by *auto*
also have $\ldots \subseteq O(g) + O(g)$
proof −
from *a* **have** $O(f) \subseteq O(g)$ **by** *auto*
thus *?thesis* **by** *auto*
qed
also have $\ldots \subseteq O(g)$ **by** (*simp add*: *bigo-plus-idemp*)
finally show *?thesis***.**
qed
qed

lemma *bigo-plus-absorb-lemma2* [*intro*]: $f \in O(g::('a \Rightarrow 'b::ordered\text{-}ring)) \Longrightarrow$
$O(g) \subseteq f + O(g)$
proof −
assume *a*: $f \in O(g::('a \Rightarrow 'b::ordered\text{-}ring))$
show $O(g) \subseteq f + O(g)$
proof −
from *a* **have** $-f \in O(g)$ **by** *auto*
then have $-f + O(g) \subseteq O(g)$ **by** *auto*
then have $f + (-f + O(g)) \subseteq f + O(g)$ **by** *auto*
also have $f + (-f + O(g)) = O(g)$
by (*simp add*: *set-plus-rearranges*)
finally show *?thesis***.**
qed
qed

lemma *bigo-plus-absorb* [*simp*]: $f \in O(g::('a \Rightarrow 'b::ordered\text{-}ring)) \Longrightarrow$
$f + O(g) = O(g)$
apply (*rule equalityI*)
apply (*rule bigo-plus-absorb-lemma1*, *assumption*)
by (*rule bigo-plus-absorb-lemma2*)

lemma *bigo-plus-absorb2* [*intro*]: $f \in O(g::('a::type \Rightarrow 'b::ordered\text{-}ring)) \Longrightarrow$

$A \subseteq O(g) \Longrightarrow f + A \subseteq O(g)$
apply (*subgoal-tac* $f + A \subseteq f + O(g)$)
by *force*+

lemma *bigo-add-commute-imp*: $(f : g +o O(h::'a=>'b::ordered\text{-}ring)) \Longrightarrow$
$(g : f +o O(h))$
apply (*subst set-minus-plus* [*THEN sym*])
apply (*subgoal-tac* $g - f = - (f - g)$)
apply (*erule ssubst*)
apply (*rule bigo-minus*)
apply (*subst set-minus-plus*)
apply *assumption*
by (*simp add*: *diff-minus add-ac*)

lemma *bigo-add-commute*: $(f : g +o O(h::'a=>'b::ordered\text{-}ring)) =$
$(g : f +o O(h))$
apply (*rule iffI*)
apply (*rule bigo-add-commute-imp*, *assumption*)
by (*rule bigo-add-commute-imp*, *assumption*)

lemma *bigo-bounded*: $[\![\forall x.\ 0 \le f\,x; \forall x.\ f\,x \le g\,x]\!] \Longrightarrow f \in O(g)$
apply (*auto simp add*: *bigo-def*)
apply (*rule-tac* $x = 1$ **in** *exI*)
apply *auto*
apply (*drule-tac* $x = x$ **in** *spec*)+
apply (*subgoal-tac* $0 \le g\,x$)
apply (*simp add*: *abs-nonneg*)
apply (*erule order-trans*, *assumption*)
done

lemma *bigo-bounded-alt*: $[\![\forall x.\ 0 \le f\,x; \forall x.\ f\,x \le c * g\,x]\!] \Longrightarrow f \in O(g)$
apply (*auto simp add*: *bigo-def*)
apply (*rule-tac* $x = abs\ c$ **in** *exI*)
apply *auto*
apply (*drule-tac* $x = x$ **in** *spec*)+
apply (*subgoal-tac* $abs(c) * abs(g\,x) = abs(c * g\,x)$)
apply (*erule ssubst*)
apply (*subgoal-tac* $0 \le c * g\,x$)
apply (*simp only*: *abs-nonneg*)
apply (*erule order-trans*, *assumption*)
apply *simp*
done

lemma *bigo-bounded2*: $[\![\forall x.\ lb\ x \le f\,x; \forall x.\ f\,x \le lb\ x + g\,x]\!] \Longrightarrow$
$f \in lb +o O(g)$

apply (*rule set-minus-imp-plus*)
apply (*rule bigo-bounded*)
apply (*auto simp add*: *diff-minus func-minus func-plus*)
apply (*drule-tac* $x = x$ **in** *spec*)+
apply (*subgoal-tac* $-lb\ x + lb\ x \leq -\ lb\ x + f\ x$)
apply (*simp add*: *plus-ac0*)
apply (*erule add-left-mono*)
apply (*drule-tac* $x = x$ **in** *spec*)+
apply (*subgoal-tac* $-lb\ x + f\ x \leq -lb\ x + (lb\ x + g\ x)$)
apply (*simp add*: *plus-ac0*)
by (*erule add-left-mono*)

lemma *bigo-bounded3*: $[|\ \forall x.\ lb\ x \leq f\ x\ \&\ f\ x \leq lb\ x + g\ x\ |] ==>$
$f \in lb +o O(g)$
by (*rule bigo-bounded2*, *auto*)

lemma *bigo-abs*: $(\%x.\ abs(f\ x)) =o O(f)$
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac* $x = 1$ **in** *exI*)
apply *auto*
done

lemma *bigo-abs2*: $f =o O(\%x.\ abs(f\ x))$
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac* $x = 1$ **in** *exI*)
apply *auto*
done

lemma *bigo-abs3*: $O(f) = O(\%x.\ abs(f\ x))$
apply (*rule equalityI*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-abs2*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-abs*)
done

lemma *bigo-abs4*: $f =o g +o O(h::'a=>'b::ordered\text{-}ring) ==>$
$(\%x.\ abs\ (f\ x)) =o (\%x.\ abs\ (g\ x)) +o O(h)$
apply (*drule set-plus-imp-minus*)
apply (*rule set-minus-imp-plus*)
apply (*subst func-diff*)
proof −
assume *a*: $f - g : O(h)$

```
  have (%x. abs (f x) - abs (g x)) =o O(%x. abs(abs (f x) - abs (g x)))
    by (rule bigo-abs2)
  also have ... <= O(%x. abs (f x - g x))
    apply (rule bigo-elt-subset)
    apply (rule bigo-bounded)
    apply force
    apply (rule allI)
    apply (rule abs-triangle-ineq3)
    done
  also have ... <= O(f - g)
    apply (rule bigo-elt-subset)
    apply (subst func-diff)
    apply (rule bigo-abs)
    done
  also have ... <= O(h)
    by (rule bigo-elt-subset)
  finally show (%x. abs (f x) - abs (g x)) : O(h).
qed


— generalize these beyond reals

lemma bigo-const1: (λx. c) ∈ O(λx. 1)
by (auto simp add: bigo-def mult-ac)

lemma bigo-const2 [intro]: O(λx. c) ⊆ O(λx. 1)
apply (rule bigo-elt-subset)
by (rule bigo-const1)

lemma bigo-const3: (c::real) ~= 0 ⟹ (λx. 1) ∈ O(λx. c)
apply (simp add: bigo-def)
apply (rule-tac x = abs(inverse c) in exI)
by (simp)

lemma bigo-const4: (c::real) ~= 0 ⟹ O(λx. 1) ⊆ O(λx. c)
by (rule bigo-elt-subset, rule bigo-const3, assumption)

lemma bigo-const [intro]: (c::real) ~= 0 ⟹ O(λx. c) = O(λx. 1)
by (rule equalityI, rule bigo-const2, rule bigo-const4, assumption)

lemma bigo-const-mult1: (λx. c * f x) ∈ O(f)
apply (simp add: bigo-def)
apply (rule-tac x = abs(c) in exI)
by auto
```

lemma *bigo-const-mult2* [*intro*]: $O(\lambda x.\ c * f\ x) \subseteq O(f)$
by (*rule bigo-elt-subset*, *rule bigo-const-mult1*)

lemma *bigo-const-mult3*: (c::*real*) $\neq 0 \Longrightarrow f \in O(\lambda x.\ c * f\ x)$
apply (*simp add*: *bigo-def*)
apply (*rule-tac* $x = abs(inverse\ c)$ **in** *exI*)
by (*simp add*: *mult-assoc* [*THEN sym*])

lemma *bigo-const-mult4*: (c::*real*) $\neq 0 \Longrightarrow O(f) \subseteq O(\lambda x.\ c * f\ x)$
by (*rule bigo-elt-subset*, *rule bigo-const-mult3*, *assumption*)

lemma *bigo-const-mult* [*simp*]: (c::*real*) $\neq 0 \Longrightarrow O(\lambda x.\ c * f\ x) = O(f)$
apply (*rule equalityI*, *rule bigo-const-mult2*, *rule bigo-const-mult4*)
by *assumption*

lemma *bigo-const-mult5* [*simp*]: (c::*real*) $\neq 0 \Longrightarrow$
$(\lambda x.\ c) \times O(f::('a \Rightarrow real)) = O(f)$
apply (*auto intro*!: *subsetI simp add*: *bigo-def elt-set-times-def func-times*)
apply (*rule-tac* $x = abs(c) * ca$ **in** *exI*)
apply (*auto*)
apply (*subgoal-tac* $abs(c) * ca * abs(f\ x) = abs(c) * (ca * abs(f\ x))$)
apply (*erule ssubst*)
apply (*rule mult-mono*)
apply *auto*
apply (*rule-tac* $x = \lambda\ n.\ (1\ /\ c) * x\ n$ **in** *exI*)
apply *auto*
apply (*rule-tac* $x = ca\ /\ abs(c)$ **in** *exI*)
apply *auto*
apply (*subgoal-tac* $abs(x\ xa\ /\ c) = abs(x\ xa)\ /\ abs(c)$)
apply (*erule ssubst*)
apply (*rule divide-right-mono*)
apply (*erule spec*)
apply *force*
apply (*simp add*: *real-divide-def*)
done

lemma *bigo-const-mult6* [*intro*]: $((\%x.\ c) *o\ O(f::('b => 'a::ordered\text{-}ring))) \leq$
$O(f)$
apply(*auto intro*!: *subsetI*
simp add: *bigo-def elt-set-times-def func-times*)
apply(*rule-tac* $x = ca * (abs\ c)$ **in** *exI*)
apply(*rule allI*)
apply (*subgoal-tac* $ca * abs(c) * abs(f\ x) = abs(c) * (ca * abs(f\ x))$)
apply (*erule ssubst*)
apply (*rule mult-left-mono*)

apply (*erule spec*)
apply *simp*
apply(*simp add*: *mult-ac*)
done

lemma *bigo-const-mult7* [*intro*]: $f =o O(g::('a \Rightarrow 'b::ordered\text{-}ring)) \Longrightarrow$
$(\%x.\ c * f\ x) =o O(g)$
proof −
assume $f =o O(g)$
then have $(\%x.\ c) * f =o (\%x.\ c) *o O(g)$
by *auto*
also have $(\%x.\ c) * f = (\%x.\ c * f\ x)$
by (*simp add*: *func-times*)
also have $(\%x.\ c) *o O(g) =s O(g)$
by *auto*
finally show *?thesis*.
qed

lemma *bigo-sumr-pos*: [| $\forall x.\ 0 \leq h\ x$; $f \in O(h)$ |] ==>
$(\%x.\ sumr\ 0\ x\ f) : O(\%x.\ sumr\ 0\ x\ h)$
apply(*auto simp add*: *bigo-def bigo-pos-const*)
apply(*simp add*: *bigo-pos-const*[*THEN sym*])
apply(*rule-tac x* = *c* **in** *exI*)
apply(*rule allI*)
apply(*induct-tac x*)
apply(*simp*)
apply(*simp*)
apply(*subgoal-tac abs* (*sumr 0 n h* + *h n*) = *abs* (*sumr 0 n h*) + *abs*(*h n*))
apply(*erule ssubst*)
apply(*simp add*: *ring-distrib*)
apply(*subgoal-tac abs* (*sumr 0 n f* + *f n*) <= *abs* (*sumr 0 n f*) + *abs*(*f n*))
apply(*erule order-trans*)
apply(*rule add-mono*)
apply(*simp*)
apply(*erule-tac x* = *n* **in** *allE*)
apply(*erule-tac x* = *n* **in** *allE*)
apply(*simp*)
apply(*simp add*: *abs-triangle-ineq*)
apply(*simp add*: *sumr-ge-zero real-abs-def*)
apply(*subgoal-tac* − *sumr 0 n h* <= − *0*)
apply(*erule-tac x* = *n* **in** *allE*)
apply(*simp*)
apply(*frule-tac x* = *n* **in** *allE*)
apply(*simp*)
by(*simp add*: *sumr-ge-zero*)

declare *abs-nonneg* [*simp del*]
declare *abs-nonpos* [*simp del*]

lemma *bigo-sumr-pos2*: ⟦ ∀ *x*. *0* ≤ *h x*; *f* ∈ *g* + *O*(*h*) ⟧
⟹ (λ*x*. *sumr 0 x f*) ∈ (λ*x*. *sumr 0 x g*) + *O*(λ*x*. *sumr 0 x h*)
apply(*rule set-minus-imp-plus*)
apply(*insert set-plus-imp-minus*[*of f g O*(*h*)])
apply(*simp add*: *bigo-def bigo-pos-const elt-set-plus-def func-plus func-diff-minus func-minus*)
apply(*erule exE*)
apply(*erule exE*)
apply(*simp only*: *bigo-pos-const*[*THEN sym*])
apply(*rule-tac x* = *c* **in** *exI*)
apply(*rule allI*)
apply(*rotate-tac 1*)
apply(*simp add*: *sumr-diff*)
apply(*subgoal-tac c* ∗ *abs* (*sumr 0 x h*) = *sumr 0 x* (%*x*. *c* ∗ *abs*(*h x*)))
apply(*erule ssubst*)
apply(*subgoal-tac abs* (*sumr 0 x b*) <= *sumr 0 x* (%*x*. *abs*(*b x*)))
apply(*erule order-trans*)
apply(*rule sumr-le2*)
apply(*rule allI*)
apply(*clarify*)
apply(*rotate-tac 2*)
apply(*erule-tac x* = *r* **in** *allE*)
apply(*simp*)
apply(*simp add*: *sumr-rabs*)
apply(*induct-tac x*)
apply(*simp*)
apply(*simp*)
apply(*subgoal-tac abs* (*sumr 0 n h* + *h n*) = *abs* (*sumr 0 n h*) + *abs* (*h n*))
apply(*rotate-tac* −*1*)
apply(*erule ssubst*)
apply(*simp add*: *ring-distrib*)
apply(*erule conjE*)
apply(*subgoal-tac 0* <= *sumr 0 n h*)
apply(*erule-tac x* = *n* **in** *allE*)
apply(*simp add*: *real-abs-def*)
apply *auto*
apply (*simp add*: *sumr-ge-zero*)
done

declare *abs-nonneg* [*simp*]
declare *abs-nonpos* [*simp*]

lemma *bigo-sumr3*: $f : O(h) ==>$
$(\%x.\ sumr\ 0\ x\ f) : O(\%x.\ sumr\ 0\ x\ (\%y.\ abs(h\ y)))$
apply (*rule bigo-sumr-pos*)
apply *force*
apply (*subst bigo-abs3* [*THEN sym*])
apply *assumption*
done

lemma *bigo-sumr4*: $f =o\ g +o\ O(h) ==>$
$(\%x.\ sumr\ 0\ x\ f) =o\ (\%x.\ sumr\ 0\ x\ g) +o\ O(\%x.\ sumr\ 0\ x\ (\%y.\ abs(h\ y)))$
apply (*rule bigo-sumr-pos2*)
apply *force*
apply (*subst bigo-abs3* [*THEN sym*])
apply *assumption*
done

lemma *bigo-sumr5*: [| $ALL\ x\ y.\ 0 <= h\ x\ y$;
$EX\ c.\ ALL\ x\ y.\ abs(f\ x\ y) <= c * (h\ x\ y)$ |] ==>
$(\%x.\ sumr\ 0\ (j\ x)\ (f\ x)) : O(\%x.\ sumr\ 0\ (j\ x)\ (h\ x))$
apply(*auto simp add*: *bigo-def*)
apply(*rule-tac* $x = c$ **in** *exI*)
apply(*auto*)
apply(*subgoal-tac* $ALL\ y.\ abs\ (sumr\ 0\ y\ (f\ x)) <=$
$c * abs\ (sumr\ 0\ y\ (h\ x))$)
apply(*erule spec*)
apply(*rule allI*)
apply(*subgoal-tac* $0 <= sumr\ 0\ y\ (h\ x)$)
apply(*auto simp add*: *sumr-mult sumr-le2 sumr-ge-zero*)
apply(*subgoal-tac* $abs\ (sumr\ 0\ y\ (f\ x)) <= sumr\ 0\ y\ (\%n.\ abs\ (f\ x\ n))$)
apply(*subgoal-tac* $sumr\ 0\ y\ (\%n.\ abs\ (f\ x\ n)) <= sumr\ 0\ y\ (\%n.\ c * h\ x\ n)$)
apply(*simp only*: *le-trans*)
by(*auto simp add*: *sumr-le2 sumr-rabs*)

lemma *bigo-sumr6*: [| $ALL\ x\ y.\ 0 <= h\ x\ y$;
$EX\ c.\ ALL\ x\ y.\ abs((f\ x\ y) - (g\ x\ y)) <= (c * (h\ x\ y))$ |] ==>
$(\%x.\ sumr\ 0\ (j\ x)\ (f\ x)) : (\%x.\ sumr\ 0\ (j\ x)\ (g\ x))$
$+o\ O(\%x.\ sumr\ 0\ (j\ x)\ (h\ x))$
apply(*simp only*: *set-minus-plus*[*THEN sym*] *func-diff-minus2 func-plus*
sumr-minus[*THEN sym*] *sumr-add func-minus*)
apply(*simp only*: *diff-minus*[*THEN sym*])
apply(*rule bigo-sumr5*)
by(*auto*)

lemma *bigo-sumr7*: $f =o\ O(h) ==>$

```
  (%x. sumr 0 (j x) (%y. (l x y) * (f (k x y)))) =o
    O(%x. sumr 0 (j x) (%y. abs((l x y) * (h (k x y)))))
  apply (rule bigo-sumr5)
  apply (simp del: abs-mult)
  apply (unfold bigo-def)
  apply auto
  apply (rule-tac x = c in exI)
  apply (rule allI)+
  apply (subst mult-left-commute)
  apply (rule mult-left-mono)
  apply (erule spec)
  apply force
done

lemma bigo-sumr8: f =o g +o O(h) ==>
  (%x. sumr 0 (j x) (%y. (l x y) * (f (k x y)))) =o
  (%x. sumr 0 (j x) (%y. (l x y) * (g (k x y)))) +o
    O(%x. sumr 0 (j x) (%y. abs((l x y) * (h (k x y)))))
  apply (simp only: set-minus-plus [THEN sym] func-diff sumr-diff
    right-diff-distrib [THEN sym])
  apply (erule bigo-sumr7)
done

lemma bigo-sumr9: f =o g +o O(%x. 1) ==>
    ALL x y. 0 <= l x y ==>
    (%x. sumr 0 (j x) (%y. (l x y) * f(k x y))):
      (%x. sumr 0 (j x) (%y. (l x y) * g(k x y))) +o
        O(%x. sumr 0 (j x) (l x))
  by (drule bigo-sumr8 [of f g %x. 1 j l k], simp)

lemma bigo-compose1: f =o O(g) ==> (%x. f(k x)) =o O(%x. g(k x))
  by (unfold bigo-def, auto)

lemma bigo-compose2: f =o g +o O(h) ==> (%x. f(k x)) =o (%x. g(k x)) +o
    O(%x. h(k x))
  apply (simp only: set-minus-plus [THEN sym] diff-minus func-minus
    func-plus)
  apply (erule bigo-compose1)
  done
```

28.4 Setsum

```
lemma bigo-setsum1: [| ALL x y. (0::'a::ordered-ring) <= h x y;
    EX c. ALL x y. abs(f x y) <= c * (h x y) |] ==>
      (%x. setsum (f x) (A x)) : O(%x. setsum (h x) (A x))
```

apply (*auto simp add*: *bigo-def*)
apply (*rule-tac* $x = abs\ c$ **in** *exI*)
apply (*auto*)
apply (*case-tac finite* $(A\ x)$)
apply (*subst abs-nonneg*)
apply (*rule setsum-nonneg*)
apply *assumption*
apply *force*
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule order-trans*)
apply (*rule abs-setsum*)
apply (*rule setsum-le-cong*)
apply (*rule order-trans*)
apply (*subgoal-tac abs* $(f\ x\ xa) <= c * h\ x\ xa$)
apply (*assumption*)
apply *force*
apply (*rule mult-right-mono*)
apply (*rule abs-ge-self*)
apply *force*
apply (*simp add*: *setsum-def*)
done

lemma *bigo-setsum3*: $f =o\ O(h) ==>$
$(\%x.\ setsum\ (\%y.\ ((l\ x\ y)::'a::ordered\text{-}ring) * f(k\ x\ y))\ (A\ x)) =o$
$O(\%x.\ setsum\ (\%y.\ abs((l\ x\ y) * h(k\ x\ y)))\ (A\ x))$
apply (*rule bigo-setsum1*)
apply (*rule allI*)+
apply (*rule abs-ge-zero*)
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac* $x = c$ **in** *exI*)
apply (*rule allI*)+
apply (*subst mult-left-commute*)
apply (*rule mult-left-mono*)
apply (*erule spec*)
apply *force*
done

lemma *setsum-subtractf'*: $(\sum x{:}A.\ ((f\ x)::'a::ordered\text{-}ring) - g\ x) =$
$setsum\ f\ A - setsum\ g\ A$
apply (*case-tac finite A*)
apply (*erule setsum-subtractf*)
apply (*simp add*: *setsum-def*)
done

lemma *bigo-setsum4*: f =o g +o $O(h)$ ==>
    (%x. *setsum* (%y. ((l x y)::'a::*ordered-ring*) * f(k x y)) (A x)) =o
    (%x. *setsum* (%y. ((l x y)::'a::*ordered-ring*) * g(k x y)) (A x)) +o
    O(%x. *setsum* (%y. *abs*((l x y) * h(k x y))) (A x))
  apply (*rule set-minus-imp-plus*)
  apply (*subst func-diff*)
  apply (*subgoal-tac* (%u. ($\sum$ y:A u. l u y * f (k u y)) −
     ($\sum$ y:A u. l u y * g (k u y))) =
    (%u. $\sum$ y:A u. l u y * (f − g)(k u y)))
  apply (*erule ssubst*)
  apply (*rule bigo-setsum3*)
  apply (*erule set-plus-imp-minus*)
  apply (*rule ext*)
  apply (*subst setsum-subtractf'* [*THEN sym*])
  apply (*case-tac finite* (A u))
  apply (*rule setsum-cong2*)
  apply (*subst func-diff*)
  apply (*simp add*: *ring-eq-simps*)
  apply (*simp add*: *setsum-def*)
done

lemma *bigo-setsum5*: f =o $O(h)$ ==> *ALL* x y. 0 <= l x y ==>
    ALL x. 0 <= h x ==>
      (%x. *setsum* (%y. ((l x y)::'a::*ordered-ring*) * f(k x y)) (A x)) =o
      O(%x. *setsum* (%y. (l x y) * h(k x y)) (A x))
  apply (*subgoal-tac* (%x. *setsum* (%y. (l x y) * h(k x y)) (A x))
    = (%x. *setsum* (%y. *abs*((l x y) * h(k x y))) (A x)))
  apply (*erule ssubst*)
  apply (*erule bigo-setsum3*)
  apply (*rule ext*)
  apply (*rule setsum-cong2*)
  apply (*subst abs-nonneg*)
  apply (*rule nonneg-times-nonneg*)
  apply *auto*
done

lemma *bigo-setsum6*: f =o g +o $O(h)$ ==> *ALL* x y. 0 <= l x y ==>
    ALL x. 0 <= h x ==>
      (%x. *setsum* (%y. ((l x y)::'a::*ordered-ring*) * f(k x y)) (A x)) =o
      (%x. *setsum* (%y. ((l x y)::'a::*ordered-ring*) * g(k x y)) (A x)) +o
      O(%x. *setsum* (%y. (l x y) * h(k x y)) (A x))
  apply (*rule set-minus-imp-plus*)
  apply (*subst func-diff*)
  apply (*subgoal-tac* (%u. ($\sum$ y:A u. l u y * f (k u y)) −
     ($\sum$ y:A u. l u y * g (k u y))) =

(%u. $\sum$ y:A u. l u y * (f − g)(k u y)))
apply (*erule ssubst*)
apply (*rule bigo-setsum5*)
apply (*erule set-plus-imp-minus*)
apply *assumption*+
apply (*rule ext*)
apply (*subst setsum-subtractf'* [*THEN sym*])
apply (*case-tac finite* (*A u*))
apply (*rule setsum-cong2*)
apply (*subst func-diff*)
apply (*simp add*: *ring-eq-simps*)
apply (*simp add*: *setsum-def*)
done

28.5 Misc useful stuff

lemma *bigo-useful-intro*: *A* <= *O*(*f*) ==> *B* <= *O*(*f*) ==>
A + *B* <= *O*(*f*::'*a*=>('*b*::*ordered-ring*))
apply (*subst bigo-plus-idemp* [*THEN sym*])
apply (*rule set-plus-mono2*)
apply *assumption*+
done

lemma *bigo-add-useful*: *f* =*o* *O*(*k*::'*a*=>'*b*::*ordered-ring*) ==>
g =*o* *O*(*k*) ==> *f* + *g* =*o* *O*(*k*)
apply (*subst bigo-plus-idemp* [*THEN sym*])
apply (*rule set-plus-intro*)
apply *assumption*+
done

lemma *bigo-useful-const-mult*: (*c*::'*a*::*ordered-field*) ~= *0* ==>
(%*x*. *c*) * *f* =*o* *O*(*h*) ==> *f* =*o* *O*(*h*::'*b*=>'*a*)
apply (*rule subsetD*)
apply (*subgoal-tac* (%*x*. *1* / *c*) *o* *O*(*h*) <= *O*(*h*))
apply *assumption*
apply (*rule bigo-const-mult6*)
apply (*subgoal-tac f* = (%*x*. *1* / *c*) * ((%*x*. *c*) * *f*))
apply (*erule ssubst*)
apply (*erule set-times-intro2*)
apply (*simp add*: *func-times*)
apply (*rule ext*)
apply (*subst times-divide-eq-left* [*THEN sym*])
apply *simp*
done

lemma *bigo-fix2*: $(\%x.\ f\ ((x::nat) + 1)) =o\ O(\%x.\ h(x + 1)) ==> f\ 0 = 0 ==>$
$f =o\ O(h)$
apply (*simp add*: *bigo-alt-def*)
apply *auto*
apply (*rule-tac* $x = c$ **in** *exI*)
apply *auto*
apply (*case-tac* $x = 0$)
apply *simp*
apply (*rule nonneg-times-nonneg*)
apply *force*
apply *force*
apply (*subgoal-tac* $x = Suc\ (x - 1)$)
apply (*erule ssubst*)**back**
apply (*erule spec*)
apply *simp*
done

lemma *bigo-fix3*:
$(\%x.\ f\ ((x::nat) + 1)) =o\ (\%x.\ g(x + 1)) +o\ O(\%x.\ h(x + 1)) ==>$
$f\ 0 = g\ 0 ==> f =o\ g +o\ O(h)$
apply (*rule set-minus-imp-plus*)
apply (*rule bigo-fix2*)
apply (*subst func-diff*)
apply (*subst func-diff* [*THEN sym*])**back**
apply (*erule set-plus-imp-minus*)
apply (*subst func-diff*)
apply *simp*
done

lemma *bigo-LIMSEQ1*: $f =o\ O(g) ==> g ----> 0 ==> f ----> 0$
apply (*simp add*: *LIMSEQ-def bigo-alt-def*)
apply *clarify*
apply (*drule-tac* $x = r\ /\ c$ **in** *spec*)
apply (*drule mp*)
apply (*erule pos-div-pos*)
apply *assumption*
apply *clarify*
apply (*rule-tac* $x = no$ **in** *exI*)
apply (*rule allI*)
apply (*drule-tac* $x = n$ **in** *spec*)+
apply (*rule impI*)
apply (*drule mp*)
apply *assumption*
apply (*rule order-le-less-trans*)
apply *assumption*

apply (*rule order-less-le-trans*)
apply (*subgoal-tac* $c * abs(g\ n) < c * (r\ /\ c)$)
apply *assumption*
apply (*erule mult-strict-left-mono*)
apply *assumption*
apply *simp*
done

lemma *bigo-LIMSEQ2*: $f =o\ g +o\ O(h) ==>\ h \longrightarrow 0 ==>\ f \longrightarrow a$
$==>\ g \longrightarrow a$
apply (*drule set-plus-imp-minus*)
apply (*drule bigo-LIMSEQ1*)
apply *assumption*
apply (*simp only*: *func-diff*)
apply (*erule LIMSEQ-diff-approach-zero2*)
apply *assumption*
done

28.6 Older stuff

— Are these superceded by the above?

lemma *bigo-plus-cong*: [| $(f::'a=>'b::ordered\text{-}ring) = g + h$; $h : O(j::'a=>'b::ordered\text{-}ring)$
|] ==>
$f : g +o\ O(j)$
by(*auto simp add*: *bigo-def bigo-pos-const elt-set-plus-def func-plus*)

lemma *bigo-plus-cong2*: [| $(f::'a=>'b::ordered\text{-}ring) = g + h$; $h : i +o\ O(j::'a=>'b::ordered\text{-}ring)$
|] ==>
$f : (g + i) +o\ O(j)$
apply(*auto simp add*: *bigo-def bigo-pos-const elt-set-plus-def func-plus*)
apply(*rule-tac* $x = b$ **in** *exI*)
apply(*simp only*: *bigo-pos-const*[*THEN sym*])
by(*auto simp add*: *add-assoc*)

lemma *bigo-minus-cong*: [| $(f::'a=>'b::ordered\text{-}ring) = g - h$; $h : O(j::'a=>'b::ordered\text{-}ring)$
|] ==>
$f : g +o\ O(j)$
apply(*auto simp add*: *bigo-def bigo-pos-const elt-set-plus-def func-plus*)
apply(*rule-tac* $x = -\ h$ **in** *exI*)
apply(*simp only*: *bigo-pos-const*[*THEN sym*])
apply(*auto simp add*: *func-minus func-plus func-diff-minus diff-minus*)
done

lemma *bigo-minus-cong2*: [| $(f::'a=>'b::ordered\text{-}ring) = g - h$; $h : i +o\ O(j::'a=>'b::ordered\text{-}ring)$

|] ==>
  $f : (g - i) +o O(j)$
apply(*simp*)
apply(*auto simp add*: *bigo-def bigo-pos-const elt-set-plus-def func-plus*)
apply(*rule-tac x = - b* **in** *exI*)
apply(*auto*)
apply(*simp only*: *bigo-pos-const*[*THEN sym*])
apply(*rule-tac x = c* **in** *exI*)
apply(*rule allI*)
apply(*simp add*: *func-minus*)
apply(*simp add*: *func-minus diff-minus func-plus ext plus-ac0*)
done

28.7 Less than or equal to

constdefs
  lesso :: ('*a* => '*b*::*ordered-ring*) => ('*a* => '*b*) => ('*a* => '*b*)
    (**infixl** <*o* 70)
  f <*o* *g* == (%*x*. *max* (*f x* - *g x*) 0)

lemma *bigo-lesseq1*: *f* =*o* *O*((*h*::'*a*=>'*b*::*ordered-ring*)) ==>
    ALL x. *abs* (*g x*) <= *abs* (*f x*) ==>
      g =*o* *O*(*h*)
  apply (*unfold bigo-def*)
  apply *clarsimp*
  apply (*rule-tac x = c* **in** *exI*)
  apply (*rule allI*)
  apply (*rule order-trans*)
  apply (*erule spec*)+
done

lemma *bigo-lesseq2*: *f* =*o* *O*((*h*::'*a*=>'*b*::*ordered-ring*)) ==>
    ALL x. *abs* (*g x*) <= *f x* ==>
      g =*o* *O*(*h*)
  apply (*erule bigo-lesseq1*)
  apply (*rule allI*)
  apply (*drule-tac x = x* **in** *spec*)
  apply (*rule order-trans*)
  apply *assumption*
  apply (*rule abs-ge-self*)
done

lemma *bigo-lesseq3*: *f* =*o* *O*((*h*::'*a*=>'*b*::*ordered-ring*)) ==>
    ALL x. 0 <= *g x* ==> *ALL x*. *g x* <= *f x* ==>
      g =*o* *O*(*h*)

apply (*erule bigo-lesseq2*)
apply (*rule allI*)
apply (*subst abs-nonneg*)
apply (*erule spec*)+
done

lemma *bigo-lesseq4*: *f* =*o* *O*((*h*::'*a*=>'*b*::*ordered-ring*)) ==>
ALL x. *0* <= *g x* ==> *ALL x*. *g x* <= *abs* (*f x*) ==>
g =*o* *O*(*h*)
apply (*erule bigo-lesseq1*)
apply (*rule allI*)
apply (*subst abs-nonneg*)**back**
apply (*erule spec*)+
done

lemma *bigo-lesso1*:
ALL x. *f x* <= *g x* ==> *f* <*o* *g* =*o* *O*(*h*::'*a*=>'*b*::*ordered-ring*)
apply (*unfold lesso-def*)
apply (*subgoal-tac* (%*x*. *max* (*f x* − *g x*) *0*) = *0*)
apply (*erule ssubst*)
apply (*rule bigo-zero*)
apply (*unfold func-zero*)
apply (*rule ext*)
apply (*simp split*: *split-max*)
done

lemma *bigo-lesso2*: *f* =*o* *g* +*o* *O*((*h*::'*a*=>'*b*::*ordered-ring*)) ==>
ALL x. *0* <= *k x* ==> *ALL x*. *k x* <= *f x* ==>
k <*o* *g* =*o* *O*(*h*)
apply (*unfold lesso-def*)
apply (*rule bigo-lesseq4*)
apply (*erule set-plus-imp-minus*)
apply (*rule allI*)
apply (*rule le-maxI2*)
apply (*rule allI*)
apply (*subst func-diff*)
apply (*case-tac* *0* <= *k x* − *g x*)
apply *simp*
apply (*subst abs-nonneg*)
apply (*drule-tac x* = *x* **in** *spec*)**back**
apply (*simp add*: *compare-rls*)
apply (*subst diff-minus*)+
apply (*rule add-right-mono*)
apply (*erule spec*)
apply (*rule order-trans*)

prefer *2*
apply (*rule abs-ge-zero*)
apply (*simp add*: *compare-rls*)
done

lemma *bigo-lesso3*: *f* =*o* *g* +*o* *O*((*h*::'*a*=>'*b*::*ordered-ring*)) ==>
ALL x. *0* <= *k x* ==> *ALL x*. *g x* <= *k x* ==>
f <*o* *k* =*o* *O*(*h*)
apply (*unfold lesso-def*)
apply (*rule bigo-lesseq4*)
apply (*erule set-plus-imp-minus*)
apply (*rule allI*)
apply (*rule le-maxI2*)
apply (*rule allI*)
apply (*subst func-diff*)
apply (*case-tac* *0* <= *f x* − *k x*)
apply *simp*
apply (*subst abs-nonneg*)
apply (*drule-tac x* = *x* **in** *spec*)**back**
apply (*simp add*: *compare-rls*)
apply (*subst diff-minus*)+
apply (*rule add-left-mono*)
apply (*rule le-imp-neg-le*)
apply (*erule spec*)
apply (*rule order-trans*)
prefer *2*
apply (*rule abs-ge-zero*)
apply (*simp add*: *compare-rls*)
done

lemma *bigo-lesso4*: *f* <*o* *g* =*o* *O*(*k*::'*a*=>*real*) ==>
g =*o* *h* +*o* *O*(*k*) ==> *f* <*o* *h* =*o* *O*(*k*)
apply (*unfold lesso-def*)
apply (*drule set-plus-imp-minus*)
apply (*subgoal-tac* (%*x*. *abs* ((*g* − *h*) *x*)) : *O*(*k*))
apply (*frule bigo-add-useful*)
apply *assumption***backback**
apply (*erule bigo-lesseq2*)**backbackback**
apply (*rule allI*)
apply (*simp add*: *func-plus func-diff*)
apply (*subst abs-nonneg*)**back**
apply (*rule le-maxI2*)
apply (*auto simp add*: *func-plus func-diff compare-rls*

split: *split-max abs-split*)
apply (*rule subsetD*)
prefer *2*
apply (*rule bigo-abs*)
apply (*erule bigo-elt-subset*)
done

lemma *bigo-lesso5*: *f* <*o* *g* =*o* *O*(*h*::'*a*=>'*b*::*ordered-ring*) ==>
EX C. *ALL x*. *f x* <= *g x* + *C* ∗ *abs*(*h x*)
apply (*simp only*: *lesso-def bigo-alt-def*)
apply *clarsimp*
apply (*rule-tac x* = *c* **in** *exI*)
apply (*rule allI*)
apply (*drule-tac x* = *x* **in** *spec*)
apply (*subgoal-tac abs*(*max* (*f x* − *g x*) *0*) = *max* (*f x* − *g x*) *0*)
apply (*clarsimp simp add*: *compare-rls add-ac*)
apply (*rule abs-nonneg*)
apply (*rule le-maxI2*)
done

lemma *lesso-add*: *f* <*o* *g* =*o* *O*(*h*::'*a*=>*real*) ==>
k <*o* *l* =*o* *O*(*h*) ==> (*f* + *k*) <*o* (*g* + *l*) =*o* *O*(*h*)
apply (*unfold lesso-def*)
apply (*rule bigo-lesseq3*)
apply (*erule bigo-add-useful*)
apply *assumption*
apply (*rule allI*)
apply (*simp split*: *split-max*)
apply (*rule allI*)
apply (*simp split*: *split-max add*: *func-plus*)
done

end

29 The derivative of ln

theory *Ln* = *RealLib*:

29.1 Lower bound for ln (1 + x), for x positive and small

lemma *exp-first-two-terms*: *exp x* = *1* + *x* + *suminf* (%*n*.
inverse(*real* (*fact* (*n*+*2*))) ∗ (*x* ^ (*n*+*2*)))
proof −

```
  fix x
  have exp x = suminf(%n. inverse(real (fact n)) * (x ^ n))
    by (unfold exp-def, simp)
  also from summable-exp have ... = sumr 0 2
      (%n. inverse(real (fact n)) * (x ^ n)) + suminf (%n.
      inverse(real (fact (n+2))) * (x ^ (n+2)))
    by (rule suminf-split-initial-segment)
  also have sumr 0 2 (%n. inverse(real (fact n)) * (x ^ n)) =
      1 + x
    by (simp add: sumr-zero-to-two)
  finally show exp x = 1 + x +
    suminf (%n. inverse (real (fact (n + 2))) * x ^ (n + 2)) .
qed

lemma exp-tail-after-first-two-terms-summable:
  summable (%n. inverse(real (fact (n+2))) * (x ^ (n+2)))
proof -
  note summable-exp
  thus ?thesis
    by (frule summable-ignore-initial-segment)
qed

lemma aux1: assumes a: 0 <= x and b: x <= 1
  shows inverse (real (fact (n + 2))) * x ^ (n + 2) <= (x^2/2) * ((1/2) ^n)
proof (induct n)
  show inverse (real (fact (0 + 2))) * x ^ (0 + 2) <= x ^ 2 / 2 * (1 / 2) ^ 0
  proof -
    have real (fact (0 + 2)) = 2
      by simp
    moreover have x ^ (0 + 2) = x ^ 2
    proof -
      have 0 + 2 = (2::nat) by simp
      thus ?thesis by (rule ssubst, simp)
    qed
    ultimately have inverse (real (fact (0 + 2))) * x ^ (0 + 2) =
        inverse 2 * x ^ 2
      by (simp only:)
    also have ... <= inverse 2 * x ^ 2
      by auto
    finally show ?thesis
      by auto
  qed
next
  fix n
  assume c: inverse (real (fact (n + 2))) * x ^ (n + 2)
```

```
    <= x ^ 2 / 2 * (1 / 2) ^ n
show inverse (real (fact (Suc n + 2))) * x ^ (Suc n + 2)
       <= x ^ 2 / 2 * (1 / 2) ^ Suc n
proof −
  have inverse(real (fact (Suc n + 2))) <=
     (1/ 2) *inverse (real (fact (n+2)))
  proof −
    have Suc n + 2 = Suc (n + 2) by simp
    then have fact (Suc n + 2) = Suc (n + 2) * fact (n + 2)
      by simp
    then have real(fact (Suc n + 2)) = real(Suc (n + 2) * fact (n + 2))
      apply (rule subst) by (rule refl)
    also have ... = real(Suc (n + 2)) * real(fact (n + 2))
      by (rule real-of-nat-mult)
    finally have real (fact (Suc n + 2)) =
       real (Suc (n + 2)) * real (fact (n + 2)) .
    then have inverse(real (fact (Suc n + 2))) =
       inverse(real (Suc (n + 2))) * inverse(real (fact (n + 2)))
     apply (rule ssubst)
     apply (rule inverse-mult-distrib)
     done
    also have ... <= (1/2) * inverse(real (fact (n + 2)))
     proof (intro mult-right-mono)
       show 0 <= inverse (real (fact (n + 2)))
         by (rule inv-real-of-nat-fact-ge-zero)
     next
       show inverse(real (Suc (n + 2))) <= 1/2
       proof −
         have 2 <= Suc (n + 2)  by arith
         then have real (2::nat) <= real(Suc (n + 2))
           by (auto simp only: real-of-nat-le-iff)
         then have inverse (real(Suc (n + 2))) <=
             inverse (real (2:: nat))
           by (auto simp only: intro: real-inverse-le-swap)
         thus ?thesis by simp
       qed
     qed
     finally show ?thesis .
  qed
  moreover have x ^ (Suc n + 2) <= x ^ (n + 2)
  proof −
    have Suc n + 2 = Suc (n + 2) by auto
    then have x ^ (Suc n + 2) = x ^ Suc (n + 2) by simp
    also have ... = x ^ (n + 2) * x by simp
    also have ... <= x ^ (n + 2)
```

```
      apply (rule real-mult-le-one-le)
      by (auto simp only: a b zero-le-power)
    finally show ?thesis .
  qed
  ultimately have inverse (real (fact (Suc n + 2))) *  x ^ (Suc n + 2) <=
      (1 / 2 * inverse (real (fact (n + 2)))) * x ^ (n + 2)
    apply (rule mult-mono)
    apply (rule nonneg-times-nonneg)
    apply simp
    apply (subst inverse-nonnegative-iff-nonnegative)
    apply (rule real-of-nat-fact-ge-zero)
    apply (rule zero-le-power)
    apply assumption
    done
  also have ... = 1 / 2 * (inverse (real (fact (n + 2))) * x ^ (n + 2))
    by simp
  also have ... <= 1 / 2 * (x ^ 2 / 2 * (1 / 2) ^ n)
    apply (rule mult-left-mono)
    apply (rule prems)
    apply simp
    done
  also have ... = x ^ 2 / 2 * (1 / 2 * (1 / 2) ^ n)
    by auto
  also have (1::real) / 2 * (1 / 2) ^ n = (1 / 2) ^ (Suc n)
    by (rule realpow-Suc [THEN sym])
  finally show ?thesis .
 qed
qed

lemma aux2: (%n. x ^ 2 / 2 * (1 / 2) ^ n) sums x^2
proof -
 have (%n. (1 / 2)^n) sums (1 / (1 - (1/2)))
   apply (rule geometric-sums)
   by (simp add: abs-interval-iff)
 also have (1::real) / (1 - 1/2) = 2
   by simp
 finally have (%n. (1 / 2)^n) sums 2 .
 then have (%n. x ^ 2 / 2 * (1 / 2) ^ n) sums (x^2 / 2 * 2)
   by (rule sums-mult)
 also have x^2 / 2 * 2 = x^2
   by simp
 finally show ?thesis .
qed

lemma exp-bound: 0 <= x ==> x <= 1 ==> exp x <= 1 + x + x^2
```

```
proof −
  assume a: 0 <= x
  assume b: x <= 1
  have c: exp x = 1 + x + suminf (%n. inverse(real (fact (n+2))) *
      (x ^ (n+2)))
    by (rule exp-first-two-terms)
  moreover have suminf (%n. inverse(real (fact (n+2))) * (x ^ (n+2))) <= x^2
  proof −
    have suminf (%n. inverse(real (fact (n+2))) * (x ^ (n+2))) <=
        suminf (%n. (x^2/2) * ((1/2) ^n))
      apply (rule summable-le)
      apply (auto simp only: aux1 prems)
      apply (rule exp-tail-after-first-two-terms-summable)
      by (rule sums-summable, rule aux2)
    also have ... = x^2
      by (rule sums-unique [THEN sym], rule aux2)
    finally show ?thesis .
  qed
  ultimately show ?thesis
    by auto
qed

lemma aux3: assumes a: (0::real) <= x and b: x <= 1
    shows (1 + x + x^2)/(1 + x^2) <= 1 + x
proof −
  from a have 1 + x + x^2 <= 1 + x + x^2 + x^3
    by (auto simp add: zero-le-power)
  also have ... = (1 + x^2) * (1 + x)
    apply (auto simp add: left-distrib right-distrib)
    proof −
      have x ^ 2 * x = x * x ^ 2
        by auto
      moreover have x ^ 3 = x ^ (Suc 2)
        by auto
      ultimately show x ^ 3 = x ^ 2 * x
        by (auto simp only: realpow-Suc)
    qed
  finally have 1 + x + x^2 <= (1 + x^2) * (1 + x) .
  moreover have 0 < 1 + x^2
  proof −
    have (0::real) = 0 + 0 by simp
    also have ... < 1 + x^2
      by (rule real-add-less-le-mono, auto)
    finally show ?thesis .
  qed
```

```
  ultimately show ?thesis
    by (auto simp only: real-le-mult-imp-div-pos-le)
qed

theorem aux4: 0 <= x ==> x <= 1 ==> exp (x - x^2) <= 1 + x
proof -
  assume a: 0 <= x and b: x <= 1
  have exp (x - x^2) = exp x / exp (x^2)
    by (rule exp-diff)
  also have ... <= (1 + x + x^2) / exp (x ^2)
  proof (intro real-div-pos-le-mono)
    from a b show exp x <= (1::real) + x + x ^ 2
      by (rule exp-bound)
    then show 0 < exp (x^2)
      by auto
  qed
  also have ... <= (1 + x + x^2) / (1 + x^2)
  proof (intro real-pos-div-le-mono)
    show (0::real) < (1::real) + x^2
      by (rule real-gt-zero-plus-ge-zero-is-gt-zero, auto simp add: a)
    next show (1::real) + x ^ 2 <= exp (x ^ 2)
      by auto
    next show (0::real) <= (1::real) + x + x ^ 2
      apply (rule real-le-add-order)+
      by (auto simp add: a)
  qed
  also from a b have ... <= 1 + x
    by (rule aux3)
  finally show ?thesis .
qed

theorem ln-one-plus-pos-lower-bound: 0 <= x ==> x <= 1 ==>
    x - x^2 <= ln (1 + x)
proof -
  assume a: 0 <= x and b: x <= 1
  then have exp (x - x^2) <= 1 + x
    by (rule aux4)
  also have ... = exp (ln (1 + x))
  proof -
    from a have 0 < 1 + x by auto
    thus ?thesis
      by (auto simp only: exp-ln-iff [THEN sym])
  qed
  finally have exp (x - x ^ 2) <= exp (ln (1 + x)) .
  thus ?thesis by (auto simp only: exp-le-cancel-iff)
```

qed

29.2 Bounds for ln (1 + x), for x negative and small

lemma *ln-one-plus-neg-upper-bound*: *0 <= x ==> x < 1 ==> ln (1 − x) <= − x*
proof −
  assume *a*: *0 <= (x::real)* **and** *b*: *x < 1*
  have *(1 − x) * (1 + x + x^2) = (1 − x^3)*
    by (*simp add*: *right-distrib left-distrib diff-minus realpow-two2 real-x-times-x-squared*)
  also have *... <= 1*
    by (*auto intro*: *zero-le-power simp add*: *a*)
  finally have *(1 − x) * (1 + x + x ^ 2) <= 1* **.**
  moreover have *0 < 1 + x + x^2*
    apply (*rule real-gt-zero-plus-ge-zero-is-gt-zero*)+
    by (*auto simp add*: *a*)
  ultimately have *1 − x <= 1 / (1 + x + x^2)*
    by (*intro real-mult-le-imp-le-div-pos*)
  also have *... <= 1 / exp x*
    apply (*rule real-one-div-le-anti-mono*)
    apply *auto*
    apply (*rule exp-bound*, *rule a*)
    by (*rule order-less-imp-le*, *rule b*)
  also have *... = exp (−x)*
    by (*auto simp add*: *exp-minus real-divide-def*)
  finally have *1 − x <= exp (− x)* **.**
  also have *1 − x = exp (ln (1 − x))*
  proof −
    have *0 < 1 − x*
      by (*auto simp add*: *b*)
    thus *?thesis*
      by (*auto simp only*: *exp-ln-iff* [*THEN sym*])
  qed
  finally have *exp (ln (1 − x)) <= exp (− x)* **.**
  thus *?thesis* **by** (*auto simp only*: *exp-le-cancel-iff*)
qed

lemma *aux5*: *x < 1 ==> ln(1 − x) = −ln(1 + x / (1 − x))*
proof −
  assume *a*: *x < 1*
  have *ln(1 − x) = − ln(1 / (1 − x))*
  proof −
    have *ln(1 − x) = − (− ln (1 − x))*
      by *auto*

```
    also have − ln(1 − x) = ln 1 − ln(1 − x)
      by simp
    also have ... = ln(1 / (1 − x))
      apply (rule ln-div [THEN sym])
      by (auto simp add: a)
    finally show ?thesis .
  qed
  also have 1/ (1 − x) = 1 + x / (1 − x)
  proof −
    have 1 = 1 − x + x
      by simp
    then have 1/ (1 − x) = (1 − x + x) / (1 − x)
      by auto
    also have ... = (1 − x) / (1 − x) + x / (1 − x)
      by (rule add-divide-distrib)
    also have ... = 1 + x / (1−x)
      apply (subst add-right-cancel)
      apply (simp)
      apply (rule less-imp-neq, assumption)
      done
    finally show ?thesis .
  qed
  finally show ?thesis .
qed

lemma ln-one-plus-neg-lower-bound: 0 <= x ==> x <= (1 / 2) ==>
    x < 1 ==> − x − 2 ∗ x^2 <= ln (1 − x)
proof −
  assume a: 0 <= x and b: x <= (1 / 2)
  from b have c: x < 1
    by auto
  then have ln (1 − x) = − ln (1 + x / (1 − x))
    by (rule aux5)
  also have − (x / (1 − x)) <= ...
  proof −
    have ln (1 + x / (1 − x)) <= x / (1 − x)
      apply (rule ln-add-one-self-le-self)
      apply (rule real-ge-zero-div-gt-zero)
      by (auto simp add: a c)
    thus ?thesis
      by auto
  qed
  also have − (x / (1 − x)) = −x / (1 − x)
    by auto
  finally have d: − x / (1 − x) <= ln (1 − x) .
```

```
  have e: −x − 2∗ x^2 <= − x / (1 − x)
  proof −
    have (1 − x)∗ (−x − 2∗ x^2) <= −x
      apply (simp add: right-distrib left-distrib realpow-two2
        real-x-times-x-squared diff-minus)
      proof −
        have x ∗ (2 ∗ x^2) = (x ∗ 2) ∗ x^2
          by auto
        also have ... <= x^2
        proof −
          from b have x ∗ 2 <= (1 / 2) ∗ 2
            by auto
          also have ... = 1
            by auto
          finally have x ∗ 2 <= 1 .
          hence x ∗ 2 ∗ x^2 <= 1 ∗ x^2
            apply (rule mult-right-mono)
            by (rule zero-le-power, rule a)
          thus x ∗ 2 ∗ x^2 <= x^2
            by simp
        qed
        finally show x ∗ (2 ∗ x ^ 2) <= x ^ 2 .
      qed
    thus ?thesis
      by (intro real-le-mult-imp-le-div-pos2, auto simp add: c)
  qed
  from e d show − x − 2 ∗ x^2 <= ln (1 − x)
    by (rule real-le-trans)
qed
```

29.3 The derivative of ln

```
lemma aux6: assumes a: (h::real) ~= 0 and b: x ~= 0 shows
    (h / x − (h / x)^2) / h − 1 / x = − h / x^2
proof −
  from b have c: h / x = (h ∗ x) / (x ∗ x)
    by (simp add: real-mult-div-cancel2)
  have d: (h / x)^2 = (h ∗ h) / (x ∗ x)
    by (simp add: realpow-two2 [THEN sym])
  from c d have h / x − (h / x)^2 = (h ∗ x − h ∗ h) / (x ∗ x)
    by (simp add: real-minus-divide-distrib)
  also have ... / h = ((h ∗ x − h ∗ h) / h) / (x ∗ x)
    by simp
  also have h ∗ x − h ∗ h = h ∗ (x − h)
    by (simp add: right-diff-distrib)
```

```
  also have h * (x - h) / h = x - h
    by (simp add: a)
  finally have (h / x - (h / x) ^ 2) / h = (x - h) / (x * x) .
  then have (h / x - (h / x) ^ 2) / h - 1 / x = (x - h) / (x * x) - 1 / x
    by simp
  also have ... = (x - h) / (x * x) - x / (x * x)
    by (simp add: b)
  also have ... = ((x - h) - x) / (x * x)
    by (rule diff-divide-distrib [THEN sym])
  also have (x - h) - x = -h
    by simp
  finally show ?thesis
    by (simp add: realpow-two2)
qed

lemma aux7: assumes a: (h::real) ~= 0 and b: x ~= 0 shows
    (h / x - 2 * (h / x)^2) / h - 1 / x =  - 2 * h / x^2
proof -
  from b have c: h / x = (h * x) / (x * x)
    by (simp add: real-mult-div-cancel2)
  have (h/x)^2 = (h * h) / (x * x)
    by (simp add: realpow-two2 [THEN sym])
  then have 2 * (h / x)^2 = 2 * (h * h) / (x * x)
    by simp
  also have ... = (2 * h * h) / (x * x)
    by (simp add: real-divide-def)
  finally have d: 2 * (h / x) ^ 2 = 2 * h * h / (x * x) .
  from c d have h / x - 2 * (h / x)^2 = (h * x - 2 * h * h) / (x * x)
    by (simp add: real-minus-divide-distrib)
  also have ... / h = ((h * x - 2* h * h) / h) / (x * x)
    by simp
  also have h * x - 2 * h * h = h * (x - 2 * h)
    by (simp add: right-diff-distrib)
  also have ... / h = x - 2 * h
    by (simp add: a)
  finally have (h / x - 2 * (h / x) ^ 2) / h = (x - 2 * h) / (x * x) .
  then have (h / x - 2 * (h / x) ^ 2) / h - 1 / x =
      (x - 2 * h) / (x * x) - 1 / x
    by simp
  also have ... = (x - 2 * h) / (x * x) - x / (x * x)
    by (simp add: b)
  also have ... = (x - 2 * h - x) / (x * x)
    by (rule real-minus-divide-distrib [THEN sym])
  also have x - 2 * h - x = (- 2 * h)
    by simp
```

```
  also have x * x = x^2 by (simp add: realpow-two2)
  finally show ?thesis .
qed



lemma DERIV-ln: 0 < x ==> DERIV ln x :> 1 / x
apply (unfold deriv-def, unfold LIM-def, clarsimp)
apply (rename-tac epsilon)
apply (rule-tac x =
    min (min (epsilon * x^2 / 2) (x / 2)) (1/2) in exI)
apply (rule conjI)
apply (force simp add: real-mult-order)
apply (clarify)
apply (rename-tac epsilon h)
apply (simp only: abs-interval-iff min-less-iff-conj)
apply (subgoal-tac ln(x + h) + - ln x = ln(1 + h / x))
apply (clarsimp)

apply (case-tac 0 < (h::real))
proof -

  fix epsilon fix h
  assume 0 < (epsilon::real) and
    u1: 0 < (h::real) and
    u2: h * 2 < x and
    u3: h * 2 < epsilon * x^2 and
    u4: h * 2 < 1
  show - epsilon < ln (1 + h / x) / h + - (1 / x) &
        ln (1 + h / x) / h + - (1 / x) < epsilon
  proof
    from u1 have v1: h ~= 0 by auto
    from u1 have v2: 0 <= h by auto
    from u1 u2 have v3: h < x
      by auto
    from v3 have v4: h <= x by auto
    from u1 v3 have v5: 0 < x by auto
    then have v6: 0 <= x by auto
    from v5 have v7: x ~= 0 by auto
    from v2 v5  have v8: 0 <= (h / x)
      by (rule real-ge-zero-div-gt-zero)
    from v4 v5 have v9: (h / x) <= 1
      by (auto simp add: pos-divide-le-eq)
    from u3 v2 have v10: h < epsilon * x^2
      by auto
```

```
show - epsilon < ln (1 + h / x) / h + - (1 / x)
proof -
  have (h / x) - (h / x) ^2 <= ln (1 + h / x)
    apply (rule ln-one-plus-pos-lower-bound)
    by (auto simp add: real-0-le-divide-iff v6 v2 v9)
  then have ((h / x) - (h / x) ^2) / h <= (ln (1 + h / x)) / h
    apply (rule real-div-pos-le-mono)
    by (rule u1)
  then have ((h / x) - (h / x) ^2) / h - 1 / x <=
      (ln (1 + h / x)) / h - 1 / x
    by auto
  also have ((h / x) - (h / x) ^2) / h - 1 / x = - h / x^2
    by (rule aux6, rule v1, rule v7)
  also have ln (1 + h / x) / h - 1 / x =
      ln (1 + h / x) / h + - (1 / x)
    by simp
  finally have v11: - h / x ^ 2 <= ln (1 + h / x) / h + - (1 / x) .
  have h / x^2 < epsilon
  proof -
    from v5 have 0 < x^2
      by (rule zero-less-power)
    thus ?thesis
      by (simp only: pos-divide-less-eq v10)
  qed
  then have v12: -epsilon < - h / x^2
    by auto
  from v12 v11 show ?thesis
    by (rule order-less-le-trans)
qed

show ln (1 + h / x) / h + - (1 / x) < epsilon
proof -
  have ln (1 + h / x) <= h / x
    by (rule ln-add-one-self-le-self, rule v8)
  then have ln (1 + h / x) / h <= (h / x) / h
    by (intro real-div-pos-le-mono, assumption)
  also have ... = 1 / x
    by (auto simp add: v1)
  finally have ln (1 + h / x) / h <= 1 / x .
  then have ln (1 + h / x) / h + - (1 / x) <= 1 / x + - (1 / x)
    by auto
  also have ... = 0
    by auto
  also have 0 < epsilon
```

```
      by assumption
    finally show ?thesis .
  qed
 qed
next

 fix epsilon fix h
 assume b1: 0 < epsilon and
   b2: h ~= 0 and
   b3: ~ 0 < h and
   b4: - (1/2) < h and
   b5: - (epsilon * x ^ 2 / 2) < h and
   b6: - (x / 2) < h
 show - epsilon < ln (1 + h / x) / h + - (1 / x) &
       ln (1 + h / x) / h + - (1 / x) < epsilon
 proof
   from b1 have b22: -epsilon < 0
     by auto
   from b2 b3 have b7: h < 0
     by auto
   from b7 have b8: h <= 0 by auto
   from b7 have b16: h ~= 0 by auto
   from b6 b7 have b9: 0 < x
     by auto
   then have b18: 0 <= x by auto
   from b9 have b15: x ~= 0 by auto
   from b6 b7 have b10: - x < h
     by auto
   from b7 b9 have h / x < 0
     by (simp add: divide-less-eq)
   then have b20: h / x <= 0 by simp
   have b11: - 1 < h * 2 / x
   proof -
     from b6 have - (x / 2) * 2 < h * 2
       by auto
     also have - (x / 2) * 2 = - 1 * x
       by auto
     finally have - 1 * x < h * 2 .
     then show ?thesis
       apply (intro real-mult-less-imp-less-div-pos)
       by (rule b9)
   qed
   then have b12: -1 <= h * 2 / x
     by (simp add: order-less-imp-le)
   from b10 have -1 * x < h
```

```
  by simp
then have b13: − 1 < h / x
  apply (intro real-mult-less-imp-less-div-pos)
  by (rule b9, simp)
then have b14: − 1 <= h / x
  by (simp add: order-less-imp-le)
from b13 have b13b: −1 < h / x by simp
from b13 have b21: − h / x < 1
  by auto
have b17: − 2 * h / x^2 < epsilon
proof −
  from b5 have − 2 * h < − 2 * − (epsilon * x ^ 2 / 2)
    by auto
  also have ... = epsilon * x^2 by simp
  finally have − 2 * h < epsilon * x ^ 2 .
  then show ?thesis
    apply (subst pos-divide-less-eq)
    apply (rule zero-less-power, rule b9, assumption)
    done
qed

show − epsilon < ln (1 + h / x) / h + − (1 / x)
proof −
  have ln (1 − (− h / x)) <= − (− h / x)
    apply (rule ln-one-plus-neg-upper-bound)
    by (simp add: b20, rule b21)
  also have ln (1 − (− h / x)) = ln (1 + h / x)
    by simp
  also have − (− h / x) = h / x
    by simp
  finally have ln (1 + h / x) <= h / x .
  then have (h / x) / h <= ln (1 + h / x) / h
    apply (rule real-div-neg-le-anti-mono)
    by (rule b7)
  also have (h / x) / h = 1 / x
    by (simp add: b16)
  finally have 1 / x <= ln (1 + h / x) / h .
  then have 1 / x + − (1 / x) <= ln (1 + h / x) / h + − (1 / x)
    by auto
  then have 0 <= ln (1 + h / x) / h + − (1 / x)
    by simp
  with b22 show ?thesis
    by (rule order-less-le-trans)
qed
```

```
    show ln (1 + h / x) / h + - (1 / x) < epsilon
    proof -
      have - (- h / x) - 2 * (- h / x) ^2 <= ln (1 -  (- h / x))
        apply (rule ln-one-plus-neg-lower-bound)
        by (auto simp add: b18 b4 b6 b8 b9 divide-le-0-iff b12 b13b)
      also have - (- h / x) - 2 * (- h / x) ^2 =
         ((h / x) - 2 * (h / x) ^2)
        by (simp add: real-neg-squared-eq-pos-squared)
      also have ln (1 - - h / x) = ln (1 + h / x) by simp
      finally have h / x - 2 * (h / x) ^ 2 <= ln (1 + h / x) .
      then have ln (1 + h / x) / h <= (h / x - 2 * (h / x) ^ 2) / h
        apply (intro real-div-neg-le-anti-mono)
        by (assumption, rule b7)
      then have ln (1 + h / x) / h - 1 / x <=
          (h / x - 2 * (h / x) ^ 2) / h - 1 / x
        by auto
      also have ... = - 2 * h / x^2
        by (rule aux7, rule b16, rule b15)
      finally have ln (1 + h / x) / h - 1 / x <= - 2 * h / x ^ 2 .
      also have ... < epsilon
        by (rule b17)
      finally show ?thesis
        by simp
    qed
  qed
next

  fix epsilon fix h
  assume c1: (0::real) < x
  assume c2: h ~= (0::real)

  assume c3: ((- (epsilon * x ^ 2 / 2) < h & h < epsilon * x ^ 2 / 2) &
          - (x / 2) < h & h < x / 2) &
          - (1 / 2) < h & h < 1 / 2
  show ln (x + h) + - ln x = ln (1 + h / x)
  proof -
    from c1 have c4: x ~= 0 by simp
    have ln (x + h) + - ln x = ln (x + h) - ln x
      by simp
    also have ... = ln ((x + h) / x)
      apply (rule ln-div [THEN sym])
      apply (auto simp add: prems)
      proof -
        have - x < -(x / 2)
          by auto
```

```
      also from c3 have - (x / 2) < h
        by auto
      finally show - x < h.
    qed
  also have (x + h) / x = x / x + h / x
    by (rule add-divide-distrib)
  also have ... = 1 + h / x by (simp add: c4)
  finally show ?thesis .
 qed
qed

lemma abs-ln-one-plus-pos-minus-x-bound:
   0 <= x ==> x <= 1 ==> abs(ln (1 + x) - x) <= abs(x^2)
proof -
 assume 0 <= x
 assume x <= 1
 have ln (1 + x) <= x
   by (rule ln-add-one-self-le-self)
 then have ln (1 + x) - x <= 0
   by simp
 then have abs(ln(1 + x) - x) = - (ln(1 + x) - x)
   by (rule abs-nonpos)
 also have ... = x - ln (1 + x)
   by simp
 also have ... <= x^2
 proof -
   from prems have x - x^2 <= ln (1 + x)
     by (intro ln-one-plus-pos-lower-bound)
   thus ?thesis
     by simp
 qed
 also have ... = abs(x^2)
   by simp
 finally show ?thesis.
qed

end
```

30 Partial Summation

theory *PartialSummation* = *Series* + *FiniteLib* + *RealLib*:

lemma *partial-sum*: !*x*. *F*(*x*) = *sumr* 0 *x* (%*n*. *f*(*n*+1)) ==>
sumr a (*a* + *c* + 1) (%*n*. *f*(*n* + 1) * *G*(*n* + 1)) = *F*(*a* + *c* + 1) * *G*(*a* + *c* + 1) −
sumr a (*a* + *c*) (%*n*. *F*(*n*+1) * (*G*(*n*+2) − *G*(*n*+1))) − *F*(*a*) * *G*(*a* + 1)
apply(*induct-tac c*)
apply(*auto simp add*: *ring-eq-simps*)
done

lemma *partial-sum0*: !*x*. *F*(*x*) = *sumr* 0 *x* (%*n*. *f*(*n*+1)) ==>
sumr 0 (*c* + 1) (%*n*. *f*(*n* + 1) * *G*(*n* + 1)) = *F*(*c* + 1) * *G*(*c* + 1) −
sumr 0 (*c*) (%*n*. *F*(*n*+1) * (*G*(*n*+2) − *G*(*n*+1)))
apply(*insert partial-sum*[*of F f* 0])
by(*force*)

30.1 Added later

lemma *partial-sum-b*: *ALL x*. (*F*::*nat*=>*real*)(*x*) = (∑ *n*=1..*x*. *f n*) ==>
1 <= *a* ==>
(∑ *n*=*a*..*a* + *c*. *f*(*n* + 1) * *G*(*n* + 1)) = *F*(*a* + *c* + 1) * *G*(*a* + *c* + 1) −
(∑ *n*=*a*..*a* + *c* − 1. *F*(*n*+1) * (*G*(*n*+2) − *G*(*n*+1))) − *F*(*a*) * *G*(*a* + 1)
apply (*subst setsum-sumr5*)+
apply (*subst partial-sum*)
apply (*rule allI*)
apply (*subst setsum-sumr4* [*THEN sym*])
apply (*erule spec*)
apply *simp*
done

lemma *partial-sum-b0*: *ALL x*. 1 <= *x* −−>
(*F*::*nat*=>*real*)(*x*) = (∑ *n*=1..*x*. *f n*) ==>
(∑ *n*=1..*x* + 1. *f n* * *G n*) = *F*(*x* + 1) * *G*(*x* + 1) −
(∑ *n*=1..*x*. *F n* * (*G* (*n* + 1) − *G n*))
apply (*subst setsum-sumr4*)+
apply (*subst partial-sum0*)
apply (*rule allI*)
apply (*subst setsum-sumr4* [*THEN sym*])
apply (*subgoal-tac* (*if x* = 0 *then* 0 *else F x*) = *?s*)
apply *assumption*
apply *force*
apply *simp*
done

declare *One-nat-def* [*simp del*]

lemma *another-partial-sum*:

$(\sum n{=}1..x{+}1.\ (F{::}nat{=>}'b{::}ordered\text{-}ring)(n) * (G\ n - G\ (n - 1))) =$
$F(x + 1) * G(x + 1) - F\ 1 * G\ 0 +$
$(\sum n{=}1..x.\ G\ n * (F\ n - F\ (n{+}1)))$
apply (*induct x*)
apply (*simp add*: *ring-eq-simps*)
apply (*subst Suc-plus1*)+
apply (*subst setsum-range-plus-one-nat'*)**back**
apply *force*
apply (*erule ssubst*)
apply (*simp add*: *ring-eq-simps compare-rls*)
apply (*subst setsum-range-plus-one-nat'*)
apply *force*
apply (*simp add*: *ring-eq-simps compare-rls*)
done

lemma *another-partial-sum2*: $(\sum n{=}1..x.\ G\ n * (F\ n - F\ (n{+}1))) =$
$(\sum n{=}1..x{+}1.\ (F{::}nat{=>}'b{::}ordered\text{-}ring)(n) * (G\ n - G\ (n - 1))) -$
$F(x + 1) * G(x + 1) + F\ 1 * G\ 0$
apply (*subst another-partial-sum*)
apply (*simp add*: *ring-eq-simps*)
done

declare *One-nat-def* [*simp add*]

end

31 Identities involving sums and ln, part 1

theory *LnSum1* = *BigO* + *Ln*:

lemma *ln-sum-minus*: *sumr 0 m* (%*m*. (*ln* (*real* (*Suc* (*Suc m*))) − *ln* (*real* (*Suc m*)))) = *ln* (*real* (*Suc m*))
apply (*induct m*)
by (*simp*+)

lemma *ln-approx-upper*: [| *0* <= *x*; *x* <= *1* |] ==> *ln*((*1*::*real*) + *x*) <= *x*
by(*auto simp add*: *ln-add-one-self-le-self*)

lemma *ln-approx-lower*: [| *0* <= *x*; *x* <= *1* |] ==> (*x* − (*x* ∗ *x*)) <= *ln*((*1*::*real*) + *x*)
apply(*insert ln-one-plus-pos-lower-bound*[*of x*])
apply(*subgoal-tac 2* = *Suc* (*Suc 0*))
by(*auto*)

lemma *ln-sum-div*: *sumr 0 m* (%*m*. (*ln* ((*real* (*Suc* (*Suc m*))) / (*real* (*Suc m*))))) = *ln* (*real* (*Suc m*))
by(*auto simp add*: *ln-sum-minus ln-div*)

lemma *ln-sum-plus*: *sumr 0 m* (%*m*. (*ln* (*1 + 1 / *(*real* (*Suc m*))))) = *ln* (*real* (*Suc m*))
apply(*subgoal-tac* (%*m*. *ln* (*1 + 1 / real* (*Suc m*))) = (%*m*. *ln* ((*real* (*Suc m*)) / (*real* (*Suc m*)) + *1 / real* (*Suc m*))))
apply(*erule ssubst*)
apply(*simp only*: *add-divide-distrib* [*THEN sym*])
apply(*insert ln-sum-div*[*of m*])
apply(*force simp add*: *real-of-nat-Suc*)
by(*simp*)

lemma *ln-sum-upper*: *ln* (*real* (*Suc m*)) <= *sumr 0 m* (%*m*. *1 / *(*real* (*Suc m*)))
apply(*subst ln-sum-plus*[*THEN sym*])
apply(*rule sumr-le2*)
by(*auto*)

lemma *ln-sum-lower*: *sumr 0 m* (%*m*. *1 / *(*real* (*Suc m*))) − *sumr 0 m* (%*m*. *1 / *(*real* (*Suc m*) ∗ *real* (*Suc m*))) <= *ln* (*real* (*Suc m*))
apply(*subst ln-sum-plus*[*THEN sym*])
apply(*simp only*: *sumr-diff*)
apply(*rule sumr-le2*)
apply(*rule allI*)
apply(*rule impI*)
apply(*subgoal-tac 1 / *(*real* (*Suc r*) ∗ *real* (*Suc r*)) = *1 / *(*real* (*Suc r*)) ∗ (*1 / real* (*Suc r*)), *erule ssubst*)
apply(*rule ln-approx-lower*)
apply(*auto*)
apply(*auto simp add*: *real-divide-def*)
apply(*subst inverse-1*[*THEN sym*])
by(*auto simp add*: *real-of-nat-Suc less-imp-inverse-less order-le-less simp del*: *inverse-1*)

lemma *real-inverse-mult-suc*: *0 < k ==> 1 / *(*k* ∗ (*k* + (*1*::*real*))) = (*1 / k − 1 / *(*k + 1*))
apply(*auto simp add*: *diff-minus*)
apply(*simp only*: *minus-divide-left*)
apply (*simp only*: *real-frac-add*)
apply *simp*
done

lemma *lnsum-inv-sq-2*: *sumr 0 n* (%*i*. *1 / *(*real* (*Suc i*)) ∗ *1 / *(*real* (*Suc i*))) <= (*2 − *(*1 / *(*real n*)))

```
apply(induct-tac n)
apply(simp)
apply(case-tac n <= 0)
apply(subgoal-tac n = 0)
apply(simp)
apply(simp)
apply(simp only: linorder-not-le)
apply(simp)
proof −
  fix n
  assume 0 < (n::nat)
  assume sumr 0 n (%i. 1 / (real (Suc i) * real (Suc i))) <= 2 − 1 / real n
  have 2 − 1 / real n <= (2 − 1 / real (Suc n))
    by(auto simp add: real-divide-def real-inverse-le-swap prems)
  then have sumr 0 n (%i. 1 / (real (Suc i) * real (Suc i))) + 1 / (real (Suc n)
* real (Suc n)) <=
 2 − 1 / real n + 1 / (real (Suc n) * real (Suc n))
    by(auto simp add: prems simp del: sumr.simps)
  also have ... <= 2 − 1 / (real n) + 1 / ((real n) * real (Suc n))
    by(auto simp add: real-divide-def real-inverse-le-swap prems)
  also have ... = 2 − 1 / (real n) + 1 / (real n) − 1 / (real (Suc n))
    by(auto simp add: real-of-nat-Suc prems real-inverse-mult-suc)
  finally show sumr 0 n (%i. 1 / (real (Suc i) * real (Suc i))) + 1 / (real (Suc
n) * real (Suc n)) <=
    2 − 1 / (real (Suc n))
    by(simp add: prems)
qed

lemma lnsum-inv-sq-2b: sumr 0 (Suc n) (%i. 1 / (real (Suc i)) * 1 / (real (Suc
i))) <= (2 − (1 / (real (Suc n))))
apply(induct-tac n)
apply auto
proof −
  fix n
  assume sumr 0 n (%i. 1 / (real (Suc i) * real (Suc i))) +
         1 / (real (Suc n) * real (Suc n))
         <= 2 − 1 / real (Suc n)  (is ?lhs <= ?rhs)
  show ?lhs + 1 / (real (Suc (Suc n)) * real (Suc (Suc n)))
            <= 2 − 1 / real (Suc (Suc n))
  proof −
    from prems have ?lhs + 1 / (real (Suc (Suc n)) * real (Suc (Suc n)))
      <= ?rhs + 1 / (real (Suc (Suc n)) * real (Suc (Suc n)))
      by auto
    also have ... <= 2 − 1 / real (Suc (Suc n))
      proof −
```

```
    have 1 / (real (Suc (Suc n)) * real (Suc (Suc n))) <=
       1 / (real (Suc n) * real (Suc (Suc n)))
     by (auto simp add: real-divide-def real-inverse-le-swap
       prems real-mult-order)
    also have ... = 1 / (real (Suc n)) - 1 / (real (Suc (Suc n)))
     apply (simp only: real-of-nat-Suc real-inverse-mult-suc)
     done
    finally have (2 - (1 / real (Suc n)))
       + (1 / (real(Suc (Suc n)) * real(Suc (Suc n)))) <=
     (2 - (1 / real (Suc n))) + (1 / real (Suc n) - 1 / real (Suc (Suc n)))
     by (rule add-left-mono)
    thus ?thesis by simp
   qed
  finally show ?thesis.
 qed
qed

lemma lnsum-upper-bigo-1: (%m. sumr 0 m (%i. 1 / (real (Suc i) * (real (Suc
i))))) : O(%x. 1)
apply(auto simp add: bigo-def simp del: sumr.simps)
apply(rule-tac x = 2 in exI)
apply(rule allI)
apply(auto simp add: abs-if)
apply(subgoal-tac 0 <= sumr 0 x (%i. 1 / (real (Suc i) * real (Suc i))))
apply(simp)
apply(rule sumr-ge-zero2)
apply(rule allI, rule impI)
apply(simp add: real-divide-def)
proof -
 fix x
 have sumr 0 x (%i. 1 / (real (Suc i) * real (Suc i))) <= 2 - 1 / real x
  apply (insert lnsum-inv-sq-2)
  by(force)
 then show sumr 0 x (%i. 1 / (real (Suc i) * real (Suc i))) <= 2
  apply(rule order-trans)
  by(auto simp add: real-divide-def)
qed

lemma lnsum-sumr-bounds: ALL m. (%m. sumr 0 m (%m. 1 / (real (Suc m)) -
1 / (real (Suc m) * real (Suc m)))) m <= (%m. ln (real (Suc m))) m & (%m. ln
(real (Suc m))) m <=  (%m. sumr 0 m (%m. 1 / (real (Suc m)))) m
apply(rule allI)
by(auto simp add: ln-sum-lower ln-sum-upper sumr-diff[THEN sym])

lemma lnsum-sumr-bounds2: ALL m. (%m. sumr 0 m (%m. 1 / (real (Suc m))
```

$- 1 / (real\ (Suc\ m) * real\ (Suc\ m)))) \ m <= (\%m.\ ln\ (real\ (Suc\ m)))\ m\ \&\ (\%m.$
$ln\ (real\ (Suc\ m)))\ m <= (\%m.\ sumr\ 0\ m\ (\%m.\ 1 / (real\ (Suc\ m)) - 1 / (real$
$(Suc\ m) * real\ (Suc\ m)))) \ m + (\%m.\ sumr\ 0\ m\ (\%i.\ 1 / (real\ (Suc\ i) * (real\ (Suc$
$i))))) \ m$
apply(*rule allI*)
by(*auto simp add*: *ln-sum-lower ln-sum-upper sumr-diff* [*THEN sym*])

lemma *ln-eq-sum-inverse-bigo-1*: $(\%m.\ ln\ (real\ (Suc\ m))) : (\%\ x.\ sumr\ 0\ x\ (\%m.$
$1 / (real\ (Suc\ m)))) +o\ O(\%\ x.\ 1)$
proof −
have $(\%m.\ ln\ (real\ (Suc\ m))) =o\ (\%m.\ sumr\ 0\ m\ (\%m.\ 1 / (real\ (Suc\ m)) -$
$1 / (real\ (Suc\ m) * real\ (Suc\ m))))$
$+o\ O(\%m.\ sumr\ 0\ m\ (\%i.\ 1 / (real\ (Suc\ i) * (real\ (Suc\ i)))))$
apply (*insert lnsum-sumr-bounds2*)
apply (*rule bigo-bounded2*)
apply *auto*
done
also have $(\%m.\ sumr\ 0\ m\ (\%m.\ 1 / (real\ (Suc\ m)) -$
$1 / (real\ (Suc\ m) * real\ (Suc\ m)))) =$
$(\%m.\ sumr\ 0\ m\ (\%m.\ 1 / (real\ (Suc\ m)))) + (- (\%m.\ sumr\ 0\ m\ (\%m.$
$1 / (real\ (Suc\ m) * real\ (Suc\ m)))))$
by (*simp add*: *diff-minus func-plus func-minus ext sumr-add*
sumr-minus [*THEN sym*])
also have $((\%m.\ sumr\ 0\ m\ (\%m.\ 1 / real\ (Suc\ m))) +$
$- (\%m.\ sumr\ 0\ m\ (\%m.\ 1 / (real\ (Suc\ m) * real\ (Suc\ m))))) +o$
$O(\%m.\ sumr\ 0\ m\ (\%i.\ 1 / (real\ (Suc\ i) * real\ (Suc\ i)))) =$
$(\%m.\ sumr\ 0\ m\ (\%m.\ 1 / real\ (Suc\ m))) +o$
$(- (\%m.\ sumr\ 0\ m\ (\%m.\ 1 / (real\ (Suc\ m) * real\ (Suc\ m))))) +o$
$O(\%m.\ sumr\ 0\ m\ (\%i.\ 1 / (real\ (Suc\ i) * real\ (Suc\ i)))))$
by (*simp add*: *set-plus-rearranges*)
also have $(\%m.\ sumr\ 0\ m\ (\%m.\ 1 / real\ (Suc\ m))) +o$
$(- (\%m.\ sumr\ 0\ m\ (\%m.\ 1 / (real\ (Suc\ m) * real\ (Suc\ m))))) +o$
$O(\%m.\ sumr\ 0\ m\ (\%i.\ 1 / (real\ (Suc\ i) * real\ (Suc\ i))))) <=$
$(\%m.\ sumr\ 0\ m\ (\%m.\ 1 / real\ (Suc\ m))) +o\ O(\%m.1)$
proof (*rule set-plus-mono*)
have $- (\%m.\ sumr\ 0\ m\ (\%m.\ 1 / (real\ (Suc\ m) * real\ (Suc\ m)))) +o$
$O(\%m.\ sumr\ 0\ m\ (\%i.\ 1 / (real\ (Suc\ i) * real\ (Suc\ i)))) =$
$O(\%m.\ sumr\ 0\ m\ (\%i.\ 1 / (real\ (Suc\ i) * real\ (Suc\ i))))$
apply (*subst bigo-plus-absorb*)
apply *auto*
done
also have ... $<= O(\%m.1)$
apply (*rule bigo-elt-subset*)
apply (*rule lnsum-upper-bigo-1*)
done

```
    finally show − (%m. sumr 0 m (%m. 1 / (real (Suc m) * real (Suc m))))
      +o O(%m. sumr 0 m (%i. 1 / (real (Suc i) * real (Suc i))))
      <= O(%m. 1).
  qed
 finally show ?thesis.
qed

lemma sum-inverse-eq-ln-1: (λx. sumr 0 x (λi. 1 / (real (Suc i))))
 = (λn. ln (real (Suc n))) + O(λx. 1)
by(simp only: ln-eq-sum-inverse-bigo-1 bigo-add-commute-imp)

lemma sum-inverse-bigo-ln: (λx. sumr 0 x (λi. 1 / (real (Suc i)))) =
 O(λn. ln (real (Suc n)))
apply(insert sum-inverse-eq-ln-1)
apply(auto simp add: bigo-def bigo-pos-const elt-set-plus-def func-plus)
apply(rule-tac x = 1 + (c / (ln 2)) in exI)
apply(rule allI)
apply (case-tac x)
apply force
apply (erule ssubst)
apply (simp add: ring-distrib)
apply (rule order-trans)
apply (rule abs-triangle-ineq)
apply simp
apply (rule order-trans)
apply (erule spec)
apply (rule real-mult-le-imp-le-div-pos)
apply force
apply (rule mult-left-mono)
apply (subst ln-le-cancel-iff)
apply auto
apply (rule order-trans)
prefer 2
apply (erule spec)
apply force
done

end
```

32 Stronger versions of identities in LnSum1

theory *LnSum1a* = *BigO* + *RealLib* + *Ln*:

```
lemma real-inverse-mult-suc: 0 < k ==> 1 / (k * (k + (1::real))) = (1 / k - 1
/ (k + 1))
  apply(auto simp add: diff-minus)
  apply(simp only: minus-divide-left)
  apply (simp only: real-frac-add)
  apply simp
done

lemma telescoping-sumr: sumr x (x + y) (%n. 1 / ((real n + 1) * (real n + 2)))
=
    1 / (real (x + 1)) - 1 / (real (x + y + 1)) (is ?P y)
proof (induct-tac y)
  show ?P(0) by simp
  next fix n assume ih: ?P(n) show ?P(Suc n)
  proof -
    have sumr x (x + Suc n) (%n. 1 / ((real n + 1) * (real n + 2))) =
       sumr x (Suc (x + n)) (%n. 1 / ((real n + 1) * (real n + 2)))
      by simp
    also have ... =  sumr x (x + n) (%n. 1 / ((real n + 1) * (real n + 2))) +
       (1 / ((real (x + n) + 1) * (real (x + n) + 2)))
      by simp
    also have ... = (1 / real (x + 1) - 1 / real (x + n + 1)) +
       (1 / ((real (x + n) + 1) * (real (x + n) + 2)))
      by (simp add: ih)
    also have ... = 1 / (real (x + 1)) - 1 / (real (x + Suc n + 1))
    proof -
      have 1 / ((real (x + n) + 1) * (real (x + n) + 2)) =
        1 / ((real (x + n) + 1) * ((real (x + n) + 1) + 1))
        by simp
      also have ... = 1 / (real (x + n) + 1) - 1 / ((real (x + n) + 1) + 1)
        by (rule real-inverse-mult-suc, auto)
      finally have 1 / ((real (x + n) + 1) * (real (x + n) + 2)) =
          1 / (real (x + n) + 1) - 1 / (real (x + n) + 1 + 1).
      then have 1 / real (x + 1) - 1 / real (x + n + 1) +
          1 / ((real (x + n) + 1) * (real (x + n) + 2)) =
          1 / real (x + 1) - 1 / real (x + n + 1) +
          (1 / (real (x + n) + 1) - 1 / (real (x + n) + 1 + 1))
        by simp
      also have ... =  1 / real (x + 1) + (1 / (real (x + n) + 1) - (1 / (real (x
+ n) + 1))
          - 1 / (real (x + n) + 1 + 1))
        by simp
      also have ... =  1 / real (x + 1) - 1 / real (x + Suc n + 1)
        by simp
      finally show ?thesis.
```

```
    qed
  finally show ?thesis.
  qed
qed

lemma sums-one-over-n-n-plus-one:
  (%n. 1 / ((real (x + n) + 1) * (real (x + n) + 2))) sums (1 / (real x + 1))
proof (unfold sums-def)
  have (%n. sumr 0 n (%n. 1 / ((real (x + n) + 1) * (real (x + n) + 2)))) =
      (%y. sumr x (x + y) (%n. 1 / ((real n + 1) * (real n + 2))))
    apply (rule ext)
    apply (rule sumr-shift)
    done
  also have ... =  (%y. 1 / (real (x + 1)) - 1 / (real (x + y + 1)))
    apply (rule ext)
    apply (subst telescoping-sumr)
    apply (rule refl)
    done
  finally have a:
    (%n. sumr 0 n (%n. 1 / ((real (x + n) + 1) * (real (x + n) + 2)))) =
      (%y. 1 / (real (x + 1)) - 1 / (real (x + y + 1))).
  have (%y. 1 / (real (x + 1))) ----> 1 / (real (x + 1))
    by (rule LIMSEQ-const)
  moreover have (%y. 1 / (real (x + y + 1))) ----> 0
    apply (unfold LIMSEQ-def)
    apply auto
    apply (rule-tac x = nat(ceiling(1 / r)) in exI)
    apply auto
    proof -
      fix r fix n
      assume a: 0 < r and b: nat(ceiling(1/r)) <= n
      show 1 / real(Suc (x + n)) < r
      proof -
        have 1 / r <= real(ceiling (1 / r))
          by auto
        also have real(ceiling (1 / r)) = real(int(nat(ceiling(1/r))))
          apply auto
          apply (subgoal-tac 0 <= 1 / r)
          apply (frule ceiling-le2)
          apply simp
          apply simp
          apply (rule order-less-imp-le)
          apply (rule prems)
          done
        also have ... = real(nat(ceiling(1/r)))
```

```
        apply (subst real-of-int-real-of-nat)
        apply (rule refl)
        done
      also have ... <= real n
        by (auto simp add: prems)
      also have real n < real (Suc (x + n))
        by auto
      finally have 1 / r < real(Suc (x + n)).
      then show ?thesis
        apply (subst pos-divide-less-eq)
        apply force
        apply (subst mult-commute)
        apply (subst pos-divide-less-eq [THEN sym])
        apply (rule prems)
        apply (assumption)
        done
    qed
  qed
 ultimately have (%y. 1 / (real (x + 1)) - 1 / (real (x + y + 1))) ----->
    1 / (real (x + 1)) - 0
  by (intro LIMSEQ-diff)
 thus (λn. sumr 0 n (λn. 1 / ((real (x + n) + 1) * (real (x + n) + 2))))
    ----> 1 / (real x + 1)
  apply (subst a)
  apply (subgoal-tac 1 / real (x + 1) - 0 = 1 / (real x + 1))
  apply (erule subst, assumption)
  apply simp
  done
qed

lemma summable-one-over-n-n-plus-one:
   summable (%n. 1 / ((real (x + n) + 1) * (real (x + n) + 2)))
 apply (unfold summable-def)
 apply (rule exI)
 apply (rule sums-one-over-n-n-plus-one)
done

lemma suminf-one-over-n-n-plus-one: suminf
   (%n. 1 / ((real (x + n) + 1) * (real (x + n) + 2))) = 1 / (real x + 1)
 apply (rule sym)
 apply (rule sums-unique)
 apply (rule sums-one-over-n-n-plus-one)
 done

lemma summable-one-over-n-squared: summable (%n. 1 / (real n+1) ^2)
```

```
proof −
  have summable (λn. 1 / ((real n + 1) * (real n + 2)))
    by (insert summable-one-over-n-n-plus-one [of 0], simp)
  then have summable (λn. 2 * (1 / ((real n + 1) * (real n + 2))))
    by (intro summable-const-times)
  also have (λn. 2 * (1 / ((real n + 1) * (real n + 2)))) =
      (λn. 2 / ((real n + 1) * (real n + 2)))
    by simp
  finally have a: summable (λn. 2 / ((real n + 1) * (real n + 2))).
  show ?thesis
    apply (rule summable-comparison-test)
    prefer 2 apply (rule a)
    apply (rule-tac x = 0 in exI)
    apply auto
    apply (rule real-le-mult-imp-div-pos-le)
    apply auto
    apply (rule real-mult-le-imp-le-div-pos)
    apply (rule mult-pos)
    apply auto
    apply (subst realpow-two2 [THEN sym])
    apply (subgoal-tac (real n + 1) * (real n + 1) * 2 = (real n + 1) *
      ((real n + 1) * 2))
    apply (erule ssubst)
    apply (rule mult-left-mono)
    apply auto
    done
qed

lemma abs-ln-one-plus-pos-minus-x-bound2:
    0 <= x ==> x <= 1 ==> abs(ln (1 + x) − x) <= x^2
proof −
  assume 0 <= x
  assume x <= 1
  have ln (1 + x) <= x
    by (rule ln-add-one-self-le-self)
  then have ln (1 + x) − x <= 0
    by simp
  then have abs(ln(1 + x) − x) = − (ln(1 + x) − x)
    by (rule abs-nonpos)
  also have ... = x − ln (1 + x)
    by simp
  also have ... <= x^2
  proof −
    from prems have x − x^2 <= ln (1 + x)
      by (intro ln-one-plus-pos-lower-bound)
```

```
    thus ?thesis
      by simp
  qed
  finally show ?thesis.
qed

lemma ln-successor-diff: ln (real (n::nat) + 2) - ln (real n + 1) =
      ln (1 + 1 / (real n + 1))
  apply (subst ln-div [THEN sym])
  apply auto
  apply (rule arg-cong)back
  apply (simp add: nonzero-divide-eq-eq ring-distrib)
  apply (subst add-divide-distrib [THEN sym])
  apply auto
done

lemma gamma-lemma: summable (%n. ln(real n+2) - ln(real n+1) - 1 / (real
n + 1))
  apply (rule summable-comparison-test)
  prefer 2
  apply (rule summable-one-over-n-squared)
  apply (subgoal-tac (%n. ln (real n + 2) - ln (real n + 1)) =
      (%n. ln (1 + 1 / (real n + 1))))
  apply (erule ssubst)
  apply (rule-tac x = 0 in exI)
  apply clarify
  apply (subgoal-tac 1 / (real n + 1) ^2 = (1 / (real n + 1)) ^2)
  apply (erule ssubst)
  apply (rule abs-ln-one-plus-pos-minus-x-bound2)
  apply force
  apply (rule real-le-mult-imp-div-pos-le)
  apply force
  apply force
  apply (rule real-one-over-pow)
  apply force
  apply (rule ext)
  apply (rule ln-successor-diff)
done

constdefs
  gamma :: real
  gamma == suminf (%n. ln (real n + 2) - ln (real n + 1) - 1 / (real n + 1))

lemma gamma-def2: (%n. ln (real n + 2) - ln (real n + 1) - 1 / (real n + 1))
    sums gamma
```

apply (*unfold gamma-def*)
apply (*rule summable-sums*)
apply (*rule gamma-lemma*)
done

lemma *bigo-minus4*: $(f =o O(g::'a \Rightarrow 'b::ordered\text{-}ring)) = (- f =o O(g))$
apply (*rule iffI*)
apply (*erule bigo-minus*)
apply (*subgoal-tac* $f = - (- f)$)
apply (*erule ssubst*)
apply (*erule bigo-minus*)
apply *simp*
done

lemma *one-over-n-squared-bound-old*: $1 / (real\ (n::nat) + 1)\ \hat{}\ 2 <=$
$2 * (1 / (real\ n + 1) - 1 / (real\ n + 2))$
apply (*subgoal-tac real* $n + 2 = (real\ n + 1) + 1$)
apply (*erule ssubst*)
apply (*subst real-inverse-mult-suc* [*THEN sym*])
apply *force*
apply *simp*
apply (*rule real-le-mult-imp-div-pos-le*)
apply *auto*
apply (*rule real-mult-le-imp-le-div-pos*)
apply (*rule mult-pos*)
apply *auto*
apply (*subst realpow-two2* [*THEN sym*])
apply (*subgoal-tac* $(real\ n + 1) * (real\ n + 1) * 2 = (real\ n + 1) *$
$((real\ n + 1) * 2)$)
apply (*erule ssubst*)
apply (*rule mult-left-mono*)
apply *auto*
done

lemma *one-over-n-squared-bound*: $1 / (real\ (n::nat) + 1)\ \hat{}\ 2 <=$
$2 * (1 / ((real\ n + 1) * (real\ n + 2)))$
apply (*subgoal-tac real* $n + 2 = (real\ n + 1) + 1$)
apply (*erule ssubst*)
apply *simp*
apply (*rule real-le-mult-imp-div-pos-le*)
apply *auto*
apply (*rule real-mult-le-imp-le-div-pos*)
apply (*rule mult-pos*)
apply *auto*
apply (*subst realpow-two2* [*THEN sym*])

```
  apply (subgoal-tac (real n + 1) * (real n + 1) * 2 = (real n + 1) *
    ((real n + 1) * 2))
  apply (erule ssubst)
  apply (rule mult-left-mono)
  apply auto
done

lemma gamma-diff-bigo-one-over-x: (%x. (gamma -
  sumr 0 x (%n. ln (real n + 2) - ln (real n + 1) - 1 / (real n + 1)))) =o
    O(%x. 1 / (real x + 1)) (is (%x. (gamma - sumr 0 x ?f)) =o ?RHS)
  proof -
    have (%x. (gamma - sumr 0 x ?f)) = (%x. suminf (%n. ?f(n + x)))
      apply (rule ext)
      apply (subgoal-tac gamma = sumr 0 x ?f + suminf (%n. ?f(n + x)))
      apply (erule ssubst)
      apply simp
      apply (unfold gamma-def)
      apply (rule suminf-split-initial-segment)
      apply (rule gamma-lemma)
      done
    also have ... =o O(%x. 2 / (real x + 1))
      apply (subst bigo-minus4)
      apply (unfold func-minus)
      apply (rule bigo-bounded)
      apply (rule allI)
      apply (subst suminf-neg [THEN sym])
      apply (rule summable-ignore-initial-segment)
      apply (rule gamma-lemma)
      apply (subst suminf-zero [THEN sym])
      apply (rule summable-le)
      apply (rule allI)
      apply (unfold func-minus)
      apply (subst ln-successor-diff)
      apply simp
      apply (rule summable-zero)
      apply (rule summable-neg2)
      apply (rule summable-ignore-initial-segment)
      apply (rule gamma-lemma)
      apply (rule allI)
      apply (subst suminf-neg [THEN sym])
      apply (rule summable-ignore-initial-segment)
      apply (rule gamma-lemma)
      apply (unfold func-minus)
      proof -
        fix x
```

show *suminf* (λ*n*. − (*ln* (*real* (*n* + *x*) + *2*) − *ln* (*real* (*n* + *x*) + *1*) −
1 / (*real* (*n* + *x*) + *1*))) ≤ *2* / (*real x* + *1*) (**is** *?LHS2* <= *?RHS2*)
proof −
have *?LHS2* <= *suminf*
(%*n*. *2* ∗ (*1* / ((*real* (*x* + *n*) + *1*) ∗ (*real* (*x* + *n*) + *2*))))
apply (*rule summable-le*)
apply (*rule allI*)
apply (*subst ln-successor-diff*)
apply (*subgoal-tac* − (*ln* (*1* + *1* / (*real* (*n* + *x*) + *1*)) −
1 / (*real* (*n* + *x*) + *1*)) <= (*1* / (*real* (*n* + *x*) + *1*)) ^ *2*)
prefer *2*
apply (*subst minus-le-iff*)
apply (*subst le-diff-eq*)
apply (*subgoal-tac* − (*1* / (*real* (*n* + *x*) + *1*))^2 + *1* / (*real* (*n* + *x*) +
1) =
1 / (*real* (*n* + *x*) + *1*) − (*1* / (*real* (*n* + *x*) + *1*))^2)
apply (*erule ssubst*)
apply (*rule ln-one-plus-pos-lower-bound*)
apply *force*
apply (*rule real-le-mult-imp-div-pos-le*)
apply *force*
apply *force*
apply *force*
apply (*erule order-trans*)
apply (*subst real-one-over-pow* [*THEN sym*])
apply *force*
apply (*subgoal-tac 1* / (*real* (*n* + *x*) + *1*)^2 =
1 / (*real* (*x* + *n*) + *1*)^2)
apply (*erule ssubst*)
apply (*rule one-over-n-squared-bound*)
apply (*simp add*: *add-ac*)
apply (*rule summable-ignore-initial-segment*)
apply (*rule summable-neg2*)
apply (*rule gamma-lemma*)
apply (*rule summable-const-times*)
apply (*rule summable-one-over-n-n-plus-one*)
done
also have . . . = *2* ∗
suminf (λ*n*. (*1* / ((*real* (*x* + *n*) + *1*) ∗ (*real* (*x* + *n*) + *2*))))
apply (*rule suminf-const-times*)
apply (*rule summable-one-over-n-n-plus-one*)
done
also have *suminf* (λ*n*. (*1* / ((*real* (*x* + *n*) + *1*) ∗ (*real* (*x* + *n*) + *2*))))
= *1* / (*real x* + *1*)
by (*rule suminf-one-over-n-n-plus-one*)

```
      finally show ?thesis by simp
    qed
  qed
  also have O(%x. 2 / (real x + 1)) = O(%x. 2 * (1 / (real x + 1)))
    by simp
  also have ... = O(%x. 1 / (real x + 1))
    apply (rule bigo-const-mult)
    apply force
    done
  finally show ?thesis.
qed

lemma ln-sum-minus2: sumr 0 x (%n. ln (real n + 2) - ln (real n + 1)) =
  ln (real x + 1)
  apply (induct x)
  apply simp+
done

lemma better-ln-theorem: (%x. ln(real x + 1)) =o
  ((%x. sumr 0 x (%n. 1 / (real n + 1))) + (%x. gamma)) +o O(%x. 1 / (real x
+ 1))
proof -
  have - (%x. (gamma - sumr 0 x (%n. ln (real n + 2) -
      ln (real n + 1) - 1 / (real n + 1)))) =o
      O(%x. 1 / (real x + 1)) (is ?LHS =o ?RHS)
    apply (rule bigo-minus)
    by (rule gamma-diff-bigo-one-over-x)
  also have ?LHS = (%x. sumr 0 x (%n. ln (real n + 2) -
      ln (real n + 1) - 1 / (real n + 1))) - (%x. gamma)
    by (simp add: func-minus diff-minus func-plus add-ac)
  also have (%x. sumr 0 x (%n. ln (real n + 2) -
      ln (real n + 1) - 1 / (real n + 1))) =
      (%x. sumr 0 x (%n. ln (real n + 2) -
      ln (real n + 1))) - (%x. sumr 0 x (%n. 1 / (real n + 1)))
    by (simp add: diff-minus func-minus func-plus add-ac sumr-add [THEN sym]
      sumr-minus)
  also have (%x. sumr 0 x (%n. ln (real n + 2) -
      ln (real n + 1))) = (%x. ln(real x + 1))
    apply (rule ext)
    apply (subst ln-sum-minus2)
    apply (rule refl)
    done
  finally have (λx. ln (real x + 1)) - (λx. sumr 0 x (λn. 1 / (real n + 1))) -
      (λx. gamma) ∈ O(λx. 1 / (real x + 1)) (is ?a ∈ ?B).
  then have ((λx. sumr 0 x (λn. 1 / (real n + 1))) + (λx. gamma)) + ?a :
```

```
    ((λx. sumr 0 x (λn. 1 / (real n + 1))) + (λx. gamma)) +o ?B
   by (rule set-plus-intro2)
 thus ?thesis
   by (simp add: set-plus-rearranges add-ac compare-rls)
qed

lemma better-ln-theorem2: (%x. ln(real x + 1)) =o
 ((%x. sumr 0 (x+1) (%n. 1 / (real n + 1))) + (%x. gamma)) +o
   O(%x. 1 / (real x + 1)) (is ?LHS =o ?RHS)
proof −
 note better-ln-theorem
 also have ((λx. sumr 0 x (λn. 1 / (real n + 1))) + (λx. gamma)) +o
     O(λx. 1 / (real x + 1)) = ?RHS (is ?LHS2 = ?RHS)
 proof −
   have (%x. sumr 0 (x+1) (%n. 1 / (real n + 1))) =
     (%x. sumr 0 x (%n. 1 / (real n + 1))) + (%x. 1 / (real x + 1))
     by (simp add: func-plus)
   then have ?RHS = ((%x. sumr 0 x (%n. 1 / (real n + 1))) + (%x. gamma))

     +o ((%x. (1 / (real x + 1))) +o O(%x. 1 / (real x + 1)))
     by (elim ssubst, simp add: set-plus-rearranges add-ac)
   also have (%x. (1 / (real x + 1))) +o O(%x. 1 / (real x + 1)) =
     O(%x. 1 / (real x + 1))
     by (rule bigo-plus-absorb, rule bigo-refl)
   finally show ?thesis by (rule sym)
 qed
 finally show ?thesis.
qed

end
```

33 Identities involving sums and ln, part 2

```
theory LnSum2 = LnSum1:

lemma inverse-func-pos: ALL x::nat. 0 <= 1 / (real (Suc x))
apply auto
done



lemma abs-ln-one-plus-pos-minus-x-bound2: [| 0 <= x; x <= 1 |] ==>
```

```
  abs (ln (1 + x) − x) <= x ^ 2
  apply (subgoal-tac x ^ 2 = abs(x ^ 2))
  apply (erule ssubst)
  apply (rule abs-ln-one-plus-pos-minus-x-bound)
  apply auto
  done

lemma ln-1-plus-small: (%n::nat. ln (1 + (1 / (real (n + 1))))) ∈ (%n::nat. 1 /
(real (n + 1))) +o
  O(%n::nat. 1 / ((real (n + 1)) * (real (n + 1))))
apply simp
apply (rule set-minus-imp-plus)
apply (unfold bigo-def)
apply simp
apply (rule-tac x = 1 in exI)
apply (clarsimp)
apply (subgoal-tac 1 / (real (Suc x) * real (Suc x)) = (1 / real (Suc x)) ^2)
apply (simp only: func-diff-minus)
apply (rule abs-ln-one-plus-pos-minus-x-bound2)
apply auto
proof −
  fix x
  show 1 / real(Suc x) <= 1
  proof −
    have 1 <= Suc x
      by auto
    then have inverse(real(Suc x)) <= 1
      by (rule real-inverse-nat-le-one)
    thus 1 / real(Suc x) <= 1
      by (simp add: real-divide-def)
  qed
next
  fix x
  show 1 / (real (Suc x) * real (Suc x)) = (1 / real (Suc x)) ^ 2
  proof −
    have 1 / (real(Suc x) * real (Suc x)) = (1 / real(Suc x)) * (1 / real(Suc x))
      by simp
    also have ... = (1 / real(Suc x)) ^2
      by (rule realpow-two2)
    finally show ?thesis.
  qed
qed
```

lemma *xlnx-sum*: *sumr 0 m (%n. (real (n + 2)) * (ln (real (n + 2))) −*
*(real (n + 1)) * (ln (real (n + 1)))) = ((real (m + 1)) * (ln (real (m + 1))))*
apply(*induct-tac m*)
by(*simp-all*)

lemma *xlnx-sum-plus*: *sumr 0 m (%n. (ln (real (n + 2))) +*
*(real (n + 1)) * ((ln (real (n + 2))) − (ln (real (n + 1))))) = ((real (m + 1))*
** (ln (real (m + 1))))*
apply(*subgoal-tac (%n. ln (real ((n::nat) + 2)) + real (n + 1) * (ln (real (n +*
2)) − ln (real (n + 1)))) =
*(%n. (real (n + 2)) * (ln (real (n + 2))) − (real (n + 1)) * (ln (real (n +*
1)))))
apply(*erule ssubst*)
apply(*simp only*: *xlnx-sum*)
apply(*rule ext*)
apply(*auto simp add*: *plus-ac0 diff-minus right-distrib left-distrib add-assoc*)
apply(*subgoal-tac ln (real (Suc (Suc n))) + real (Suc n) * ln (real (Suc (Suc n)))*
=
*1 * ln (real (Suc (Suc n))) + real (Suc n) * ln (real (Suc (Suc n)))*
apply(*erule ssubst*)
apply(*simp only*: *real-add-mult-distrib*[*THEN sym*])
apply(*simp add*: *real-of-nat-Suc*)
by(*simp*)

lemma *xlnx-sum-split1*: *sumr 0 m (%n. (ln (real (n + 2)))) + sumr 0 m (%n.*
*(real (n + 1)) * (ln (1 + 1 / (real (n + 1))))) = ((real (m + 1)) * (ln (real (m*
+ 1))))
apply(*subgoal-tac sumr 0 m (%n. ln (real (n + 2))) +*
*sumr 0 m (%n. real (n + 1) * ln (1 + 1 / real (n + 1))) =*
sumr 0 m (%n. (ln (real (n + 2))) +
*(real (n + 1)) * ((ln (real (n + 2))) − (ln (real (n + 1))))))*
apply(*erule ssubst*)
apply(*simp only*: *xlnx-sum-plus*)
apply(*simp add*: *sumr-add*)
apply(*subgoal-tac (%n. ln (real (Suc (Suc n))) + real (Suc n) * ln (1 + 1 / real*
(Suc n))) =
(%n. ln (real (Suc (Suc n))) +
*real (Suc n) * (ln (real (Suc (Suc n))) − ln (real (Suc n)))))*
apply(*erule ssubst*, *simp*)
apply(*rule ext*,*simp*)
apply(*subgoal-tac ln (1 + 1 / real (Suc n)) = (ln (real (Suc (Suc n))) − ln (real*
(Suc n))))
apply(*erule ssubst*, *simp*)
apply(*subgoal-tac ln (real (Suc (Suc n))) − ln (real (Suc n)) = ln (real (Suc (Suc*
n)) / (real (Suc n))))

apply(*simp*)
apply(*subgoal-tac real (Suc (Suc n)) = real (Suc n) + 1*)
apply(*simp add*: *add-divide-distrib*)
apply(*simp only*: *real-of-nat-Suc*)
by(*auto simp add*: *ln-div*)

lemma *xlnx-sum-split2*: *sumr 0 m (%n. (ln (real (n + 2)))) + sumr 0 m (%n. (real (n + 1)) * (1 / (real (n + 1)) + (ln (1 + 1 / (real (n + 1)))) − 1 / (real (n + 1)))) = ((real (m + 1)) * (ln (real (m + 1))))*
apply(*subgoal-tac sumr 0 m (%n. (ln (real (n + 2)))) + sumr 0 m (%n. (real (n + 1)) * (1 / (real (n + 1)) + (ln (1 + 1 / (real (n + 1)))) − 1 / (real (n + 1)))) =*
*sumr 0 m (%n. (ln (real (n + 2)))) + sumr 0 m (%n. (real (n + 1)) * (ln (1 + 1 / (real (n + 1)))))*)
apply(*erule ssubst*, *simp only*: *xlnx-sum-split1*)
by(*simp*)

lemma *xlnx-sum-split3*: *sumr 0 m (%n. (ln (real (n + 2)))) + (real m) + sumr 0 m (%n. (real (n + 1)) * (ln (1 + 1 / (real (n + 1))) − 1 / (real (n + 1)))) = ((real (m + 1)) * (ln (real (m + 1))))*
apply(*subgoal-tac sumr 0 m (%n. ln (real (n + 2))) + real m +*
*sumr 0 m (%n. real (n + 1) * (ln (1 + 1 / real (n + 1)) − 1 / real (n + 1))) =*
*sumr 0 m (%n. (ln (real (n + 2)))) + sumr 0 m (%n. (real (n + 1)) * (1 / (real (n + 1)) + (ln (1 + 1 / (real (n + 1)))) − 1 / (real (n + 1)))))*)
apply(*erule ssubst*, *simp only*: *xlnx-sum-split2*)
apply(*simp add*: *diff-minus right-distrib*)
apply(*simp only*: *sumr-add*[*THEN sym*])
by(*simp*)

lemma *ln-1-plus-small-minus*: (%*n*::*nat*. *ln (1 + (1 / (real (n + 1)))) − 1 / (real (n + 1))) ∈*
O(%*n*::*nat*. *1 / ((real (n + 1)) * (real (n + 1))))*
apply(*insert ln-1-plus-small*)
apply(*simp add*: *bigo-def elt-set-plus-def*)
apply(*auto*)
apply(*rule-tac x = c* **in** *exI*)
apply(*rule allI*)
apply(*erule-tac x = x* **in** *allE*)
apply(*erule ssubst*)
apply (*auto simp add*: *func-plus*)
apply (*subgoal-tac − b x <= abs (b x)*)
apply (*erule order-trans*)
apply *assumption*
apply (*rule abs-ge-minus-self*)

done

lemma *xln-1-plus-small-minus*: (%*n*::*nat*. (*real* (*n* + *1*)) ∗ (*ln* (*1* + (*1* / (*real* (*n* + *1*)))) −
1 / (*real* (*n* + *1*)))) = *O*(%*n*::*nat*. *1* / (*real* (*n* + *1*)))
apply(*subgoal-tac* (%*n*::*nat*. (*real* (*n* + *1*)) ∗ (*ln* (*1* + (*1* / (*real* (*n* + *1*)))) −
1 / (*real* (*n* + *1*)))) = (%*n*::*nat*. (*real* (*n* + *1*))) ∗ (%*n*. (*ln* (*1* + (*1* / (*real* (*n* + *1*)))) −
1 / (*real* (*n* + *1*)))))
apply(*erule ssubst*)
proof −
have *1*: (%*n*::*nat*. *real* (*n* + *1*)) = *O*(%*n*::*nat*. *real* (*n* + *1*)) **by** (*simp add*: *bigo-refl*)
also have (%*n*::*nat*. *ln* (*1* + *1* / *real* (*n* + *1*)) − *1* / *real* (*n* + *1*)) = *O*(%*n*::*nat*. *1* / ((*real* (*n* + *1*)) ∗ (*real* (*n* + *1*)))) **by** (*simp only*: *ln-1-plus-small-minus*)
ultimately have (%*n*. *real* (*n* + *1*)) ∗
(%*n*. *ln* (*1* + *1* / *real* (*n* + *1*)) − *1* / *real* (*n* + *1*)) = *O*((%*n*. *real* (*n* + *1*)) ∗ (%*n*::*nat*. *1* / ((*real* (*n* + *1*)) ∗ (*real* (*n* + *1*))))))
by(*auto simp add*: *bigo-mult3*)
then show (%*n*::*nat*. *real* (*n* + *1*)) ∗
(%*n*. *ln* (*1* + *1* / *real* (*n* + *1*)) − *1* / *real* (*n* + *1*)) = *O*(%*n*. *1* / *real* (*n* + *1*))
by(*simp add*: *func-times*)
next
show (%*n*. *real* (*n* + *1*) ∗ (*ln* (*1* + *1* / *real* (*n* + *1*)) − *1* / *real* (*n* + *1*))) =
(%*n*. *real* (*n* + *1*)) ∗ (%*n*. *ln* (*1* + *1* / *real* (*n* + *1*)) − *1* / *real* (*n* + *1*))
by (*simp add*: *func-times*)
qed

lemma *xlnx-sum-snd-bigo*: (%*m*. *sumr 0 m* (%*n*. (*real* (*n* + *1*)) ∗ (*ln* (*1* + *1* / (*real* (*n* + *1*))) − *1* / (*real* (*n* + *1*))))) = *O*(%*m*. *sumr 0 m* (%*n*::*nat*. *1* / (*real* (*n* + *1*))))
apply(*simp*)
apply(*rule bigo-sumr-pos*)
apply(*simp add*: *inverse-func-pos*)
by(*insert xln-1-plus-small-minus*,*simp*)

lemma *fn-xlnx-sum-split3*: (%*m*. *sumr 0 m* (%*n*. (*ln* (*real* (*n* + *2*))))) + (%*m*. (*real m*)) + (%*m*. *sumr 0 m* (%*n*. (*real* (*n* + *1*)) ∗ (*ln* (*1* + *1* / (*real* (*n* + *1*))) − *1* / (*real* (*n* + *1*))))) = (%*m*. ((*real* (*m* + *1*)) ∗ (*ln* (*real* (*m* + *1*))))))
apply(*simp only*: *func-plus*)
by(*rule ext*, *simp only*: *xlnx-sum-split3*)

lemma *fn-lnx-sum*: (%*m*. *sumr 0 m* (%*n*. (*ln* (*real* (*n* + *2*))))) = (%*m*. ((*real* (*m* + *1*)) ∗ (*ln* (*real* (*m* + *1*))))) − (%*m*. (*real m*)) − (%*m*. *sumr 0 m* (%*n*. (*real* (*n*

```
+ 1)) * (ln (1 + 1 / (real (n + 1))) - 1 / (real (n + 1)))))
apply(insert fn-xlnx-sum-split3)
proof -
  assume (%m. sumr 0 m (%n. ln (real (n + 2)))) + real +
    (%m. sumr 0 m (%n. real (n + 1) * (ln (1 + 1 / real (n + 1)) - 1 / real (n
+ 1)))) =
    (%m. real (m + 1) * ln (real (m + 1)))
  have (%m. sumr 0 m (%n. ln (real (n + 2)))) + real +
    (%m. sumr 0 m (%n. real (n + 1) * (ln (1 + 1 / real (n + 1)) - 1 / real (n
+ 1)))) + - real =
    (%m. real (m + 1) * ln (real (m + 1))) + - real
    apply(subst prems)
    by(simp)
  then have (%m. sumr 0 m (%n. ln (real (n + 2)))) + real +
    (%m. sumr 0 m (%n. real (n + 1) * (ln (1 + 1 / real (n + 1)) - 1 / real
(n + 1)))) + - real + - (%m. sumr 0 m (%n. real (n + 1) * (ln (1 + 1 / real
(n + 1)) - 1 / real (n + 1)))) = (%m. real (m + 1) * ln (real (m + 1))) + -
real + - (%m. sumr 0 m (%n. real (n + 1) * (ln (1 + 1 / real (n + 1)) - 1 /
real (n + 1))))
    by(simp)
  then show ?thesis
    apply(simp)
    apply(simp add: func-plus func-diff-minus)
    apply(rule ext)
    apply(simp only: expand-fun-eq func-diff-minus)
    apply(erule-tac x = m in allE)
    by(simp)
qed

lemma fn-lnx-sum-bigo: (%m. sumr 0 m  (%n. (ln (real (n + 2))))) = ((%m.
((real (m + 1)) * (ln (real (m + 1))))) - (%m. (real m))) +o O(%m. sumr 0 m
(%n::nat. 1 / (real (n + 1))))
apply(subst fn-lnx-sum)
apply(auto simp add: elt-set-plus-def)
apply(rule-tac x = - (%m. sumr 0 m (%n. real (Suc n) * (ln (1 + 1 / real (Suc
n)) - 1 / real (Suc n)))) in bexI)
apply(rule ext)
apply(simp add: func-diff-minus func-plus func-minus)
apply(insert xlnx-sum-snd-bigo)
by auto

lemma xlnx-sum-snd-bigo-ln: (%m. sumr 0 m (%n. (real (n + 1)) * (ln (1 + 1
/ (real (n + 1))) - 1 / (real (n + 1))))) = O(%n::nat. ln (real (n + 1)))
apply(insert xlnx-sum-snd-bigo)
apply(insert sum-inverse-bigo-ln)
```

apply(*simp only*: *bigo-def bigo-pos-const elt-set-plus-def*)
apply(*auto simp add*: *bigo-def bigo-pos-const elt-set-plus-def*)

apply(*rule-tac x* = *c* ∗ *ca* **in** *exI*)
apply *auto*
apply (*elim mult-pos*)
apply *assumption*
apply(*erule-tac x* = *x* **in** *allE*)
apply(*erule order-trans*)
apply(*erule-tac x* = *x* **in** *allE*)
apply(*simp only*: *mult-assoc*)
by(*simp*)

lemma *fn-lnx-sum-bigo-ln*: (%*m*. *sumr 0 m* (%*n*. (*ln* (*real* (*n* + *2*))))) = ((%*m*. ((*real* (*m* + *1*)) ∗ (*ln* (*real* (*m* + *1*))))) − (%*m*. (*real m*))) +*o* *O*(%*n*::*nat*. *ln* (*real* (*n* + *1*)))
apply(*subst fn-lnx-sum*)
apply(*auto simp add*: *elt-set-plus-def*)
apply(*rule-tac x* = − (%*m*. *sumr 0 m* (%*n*. *real* (*Suc n*) ∗ (*ln* (*1* + *1* / *real* (*Suc n*)) − *1* / *real* (*Suc n*)))) **in** *bexI*)
apply(*rule ext*)
apply(*simp add*: *func-diff-minus func-plus func-minus*)
apply(*insert xlnx-sum-snd-bigo-ln*)
by *auto*

lemma *fn-lnx-sum-bigo-ln2*: (λ*m*. *sumr 0* (*Suc m*) (λ*n*. *ln* (*real* (*Suc n*))))
  = ((λ*m*. ((*real* (*m* + *1*)) ∗ (*ln* (*real* (*m* + *1*))))) − (λ*m*. (*real m*)))
  + *O*(λ*n*::*nat*. *ln* (*real* (*n* + *1*)))
proof −
  have (%*m*. *sumr 0* (*Suc m*) (%*n*. (*ln* (*real* (*Suc n*))))) =
      (%*m*. *sumr 0 m* (%*n*. (*ln* (*real* (*n* + *2*)))))
    apply(*rule ext*)
    apply(*induct-tac m*)
    by(*auto*)
  then show *?thesis* **by** (*auto simp only*: *fn-lnx-sum-bigo-ln*)
qed

end

34 Identities involving sums and ln, part 3

theory *LnSum3* = *LnSum2*:

lemma *lndivxsum-minus*: *sumr 0 (Suc n) (%m. ln ((real (Suc n)) / (real (Suc m))))*
= sumr 0 (Suc n) (%m. ln (real (Suc n))) − sumr 0 (Suc n) (%m. ln (real (Suc m)))
proof −
have *0 < real (Suc n)*
by (*simp add*: *real-of-nat-ge-zero*)
moreover have *0 < real (Suc m)*
by (*simp add*: *real-of-nat-ge-zero*)
ultimately have *(%m. ln (real (Suc n) / real (Suc m))) =*
(%m. ln (real (Suc n)) − ln(real (Suc m)))
by(*simp add*: *ext ln-div*)
then have *sumr 0 (Suc n) (%m. ln ((real (Suc n)) / (real (Suc m)))) =*
sumr 0 (Suc n) (%m. ln (real (Suc n)) − ln(real (Suc m)))
by *simp*
also have *sumr 0 (Suc n) (%m. ln (real (Suc n)) − ln(real (Suc m))) =*
sumr 0 (Suc n) (%m. ln (real (Suc n))) − sumr 0 (Suc n) (%m. ln (real (Suc m)))
by(*simp only*: *sumr-diff*[*THEN sym*])
finally show *?thesis*.
qed

lemma *lndivxsum-minus2*: *sumr 0 (Suc n) (%m. ln ((real (Suc n)) / (real (Suc m)))) =*
(real (Suc n)) ∗ (ln (real (Suc n))) − (sumr 0 (Suc n) (%m. ln (real (Suc m))))
proof −
have *sumr 0 (Suc n) (%m. ln (real (Suc n))) = real (Suc n) ∗ ln (real (Suc n))*

by (*rule sumr-const*)
then show *sumr 0 (Suc n) (%m. ln ((real (Suc n)) / (real (Suc m)))) =*
(real (Suc n)) ∗ (ln (real (Suc n))) − (sumr 0 (Suc n) (%m. ln (real (Suc m))))
apply(*insert lndivxsum-minus*[*of n*])
by(*auto simp add*: *lndivxsum-minus simp del*: *sumr.simps*)
qed

lemma *lndivxsum-eq-x-bigo-ln*: *(%n. sumr 0 (Suc n) (%m. ln ((real (Suc n)) / (real (Suc m))))) :*
(%n. real n) +o O(%n. ln (real (Suc n)))
proof −
have *1*: *(%n. sumr 0 (Suc n) (%m. ln (real (Suc n) / real (Suc m)))) =*
(%n. (real (Suc n)) ∗ (ln (real (Suc n)))) − (%n. (sumr 0 (Suc n) (%m. ln (real (Suc m)))))
apply(*simp add*: *lndivxsum-minus2 del*: *sumr.simps*)

```
    by(simp add: ext func-diff-minus)
  moreover have 2: (%m. sumr 0 (Suc m)  (%n. (ln (real (Suc n))))) =
  ((%m. ((real (m + 1)) * (ln (real (m + 1))))) - (%m. (real m))) +o
  O(%n::nat. ln (real (n + 1))) by(rule fn-lnx-sum-bigo-ln2)
  ultimately have (%n. sumr 0 (Suc n) (%m. ln (real (Suc n)) / real (Suc m))))
:
       ((%n. (real (Suc n)) * (ln (real (Suc n)))) - (((%m. ((real (m + 1)) * (ln
(real (m + 1))))) - (%m. (real m))))) +o O(%n::nat. ln (real (n + 1)))
    apply(simp (no-asm-use))
    apply(erule bigo-minus-cong2)
    by(simp)
 moreover have ((%n. (real (Suc n)) * (ln (real (Suc n)))) - (((%m. ((real (m +
1)) * (ln (real (m + 1))))) - (%m. (real m))))) = (%m. (real m))
    apply(rule ext)
    by(simp add: func-diff-minus func-plus)
  ultimately show ?thesis
    by(auto)
qed

lemma ln-sq-sum: ln (real (Suc n)) * (ln (real (Suc n))) = sumr 0 n
  (%i. ln (real (Suc (Suc i))) * (ln (real (Suc (Suc i)))) -
  ln (real (Suc i)) * (ln (real (Suc i))))
apply(induct-tac n)
by(auto)

lemma ln-sq-diff-factor: ln (real (Suc (Suc i))) * (ln (real (Suc (Suc i)))) -
  ln (real (Suc i)) * (ln (real (Suc i))) =
  (ln (real (Suc (Suc i))) - ln (real (Suc i))) *
  (ln (real (Suc (Suc i))) + ln (real (Suc i)))
  by (auto simp add: ring-eq-simps)

lemma ln-factor-minus-1: ln (real (Suc (Suc i))) - ln (real (Suc i)) =
  ln (1 + 1 / (real (Suc i)))
proof -
  have ln (real (Suc (Suc i))) - ln (real (Suc i)) = ln (real (Suc (Suc i)) /
    real (Suc i))
    by(auto simp add: ln-div)
  also have ... = ln ((real (Suc i)) / (real (Suc i)) + 1 / real (Suc i))
    apply (subgoal-tac real(Suc (Suc i)) = real(Suc i) + 1)
    apply (erule ssubst)
    apply (simp only: add-divide-distrib [THEN sym])
    apply simp
    done
  also have ... = ln (1 + 1 / (real (Suc i))) by simp
  finally show ?thesis.
```

qed

lemma *ln-factor-minus-lbound*: *1 / (real (Suc i)) − 1 / (real (Suc i)) ∗ (1 / (real (Suc i))) <= ln (real (Suc (Suc i))) − ln (real (Suc i))*
apply(*subst ln-factor-minus-1*)
apply(*rule ln-approx-lower*)
apply(*auto simp add*: *real-divide-def*)
apply(*subgoal-tac 1 = inverse 1*,*erule ssubst*)
apply(*rule real-inverse-le-swap*)
by(*auto simp add*: *real-of-nat-Suc*)

lemma *ln-factor-minus-ubound*: *ln (real (Suc (Suc i))) − ln (real (Suc i)) <= 1 / (real (Suc i))*
apply(*subst ln-factor-minus-1*)
apply(*rule ln-approx-upper*)
apply(*auto simp add*: *real-divide-def*)
apply(*subgoal-tac 1 = inverse 1*,*erule ssubst*)
apply(*rule real-inverse-le-swap*)
by(*auto simp add*: *real-of-nat-Suc*)

lemma *ln-factor-plus-lbound*: *2 ∗ ln (real (Suc i)) <= (ln (real (Suc (Suc i))) + ln (real (Suc i)))*
by(*auto*)

lemma *ln-factor-plus-ubound*: *(ln (real (Suc (Suc i))) + ln (real (Suc i))) <= 2 ∗ ln (real (Suc i)) + 1 / (real (Suc i))*
proof −
  have *ln (real (Suc (Suc i))) + ln (real (Suc i)) = 2 ∗ ln (real (Suc i)) + (ln (real (Suc (Suc i))) − ln (real (Suc i)))*
    by(*auto*)
  also have *2 ∗ ln (real (Suc i)) + (ln (real (Suc (Suc i))) − ln (real (Suc i))) <=*
    2 ∗ ln (real (Suc i)) + 1 / (real (Suc i))
    apply (*rule add-left-mono*)
    by(*simp only*: *ln-factor-minus-ubound*)
  finally show *?thesis*.
qed

lemma *ln-sq-diff-lbound*: *2 ∗ ln (real (Suc i)) / (real (Suc i)) −*
  2 ∗ ln (real (Suc i)) / (real (Suc i)) ∗ (1 / (real (Suc i))) <=
  ln (real (Suc (Suc i))) ∗ (ln (real (Suc (Suc i)))) −
  ln (real (Suc i)) ∗ (ln (real (Suc i)))
proof −
  have *2 ∗ ln (real (Suc i)) / real (Suc i) −*
    2 ∗ ln (real (Suc i)) / real (Suc i) ∗ (1 / real (Suc i)) =

*2 * ln (real (Suc i)) * (1 / (real (Suc i)) − 1 / (real (Suc i)) * (1 / (real (Suc i))))* **by**(*auto simp add*: *ring-eq-simps*)
also have ... <= (*ln* (*real* (*Suc* (*Suc i*))) + *ln* (*real* (*Suc i*))) *
(*ln* (*real* (*Suc* (*Suc i*))) − *ln* (*real* (*Suc i*)))
apply (*rule mult-mono*)
apply *simp*
apply (*rule ln-factor-minus-lbound*)
apply (*rule nonneg-plus-nonneg*)
apply *auto*
apply (*rule real-one-div-le-anti-mono*)
apply *auto*
apply (*subgoal-tac 1 * real*(*Suc i*) <= *real*(*Suc i*) * *real*(*Suc i*))
apply *simp*
apply (*rule mult-right-mono*)
apply *simp*
apply *simp*
done
also have ... = (*ln* (*real* (*Suc* (*Suc i*))) − *ln* (*real* (*Suc i*))) *
(*ln* (*real* (*Suc* (*Suc i*))) + *ln* (*real* (*Suc i*))) **by** *simp*
finally show *?thesis*
apply(*subst ln-sq-diff-factor*)**.**
qed

lemma *ln-sq-diff-ubound*: *ln* (*real* (*Suc* (*Suc i*))) * (*ln* (*real* (*Suc* (*Suc i*)))) −
ln (*real* (*Suc i*)) * (*ln* (*real* (*Suc i*))) <= *2 * ln* (*real* (*Suc i*)) / (*real* (*Suc i*)) +
1 / (real (*Suc i*) * (*real* (*Suc i*)))
proof −
have *ln* (*real* (*Suc* (*Suc i*))) * *ln* (*real* (*Suc* (*Suc i*))) −
ln (*real* (*Suc i*)) * *ln* (*real* (*Suc i*)) =
(*ln* (*real* (*Suc* (*Suc i*))) − *ln* (*real* (*Suc i*))) *
(*ln* (*real* (*Suc* (*Suc i*))) + *ln* (*real* (*Suc i*)))
by(*simp add*: *ln-sq-diff-factor*)
also have ... <= *1 / (real* (*Suc i*)) * (*2 * ln* (*real* (*Suc i*)) + *1 / (real* (*Suc i*)))
apply(*rule mult-mono*)
apply (*rule ln-factor-minus-ubound*)
apply (*rule ln-factor-plus-ubound*)
apply *force*
apply (*subst ln-mult* [*THEN sym*])
apply *auto*
apply (*rule ln-ge-zero*)
apply(*subgoal-tac 1 = 1 * (1*::*real*), *erule ssubst*)
apply(*rule mult-mono*)
apply *auto*
done
also have ... = *2 * ln* (*real* (*Suc i*)) / (*real* (*Suc i*)) + *1 / (real* (*Suc i*) * (*real*

(Suc i)))
  by(*simp add*: *ring-eq-simps*)
 finally show *?thesis*.
qed

lemma *ln-sq-diff-all-bounds*: *ALL n. (%i. 2 * ln (real (Suc i)) / (real (Suc i)) −*

*2 * ((ln (real (Suc i)) + 1) / (real (Suc i)) * (1 / (real (Suc i))))) n <=*
*(%i. ln (real (Suc (Suc i))) * (ln (real (Suc (Suc i)))) −*
*ln (real (Suc i)) * (ln (real (Suc i)))) n & (%i. ln (real (Suc (Suc i))) * (ln (real*
(Suc (Suc i)))) −
*ln (real (Suc i)) * (ln (real (Suc i)))) n <= (%i. 2 * ln (real (Suc i)) / (real*
(Suc i)) −
*2 * ((ln (real (Suc i)) + 1) / (real (Suc i)) * (1 / (real (Suc i))))) n + (%i. 3*
*((ln (real (Suc i)) + 1) / (real (Suc i)) * (1 / (real (Suc i))))) n*
apply(*rule allI*)
apply(*rule conjI*)
apply(*subgoal-tac 2 * ln (real (Suc n)) / real (Suc n) −*
*2 * ((ln (real (Suc n)) + 1)/ real (Suc n) * (1 / real (Suc n)))*
*<= 2 * ln (real (Suc n)) / real (Suc n) −*
*2 * ln (real (Suc n)) / real (Suc n) * (1 / real (Suc n)))*
apply(*erule order-trans*)
apply(*simp only*: *ln-sq-diff-lbound*)
apply(*simp add*: *real-divide-def*)
apply(*subgoal-tac 2 * ln (real (Suc n)) / real (Suc n) −*
*2 * ((ln (real (Suc n)) + 1)/ real (Suc n) * (1 / real (Suc n))) +*
*3 * ((ln (real (Suc n)) + 1)/ real (Suc n) * (1 / real (Suc n))) =*
*2 * ln (real (Suc n)) / real (Suc n) + ((ln (real (Suc n)) + 1)/ real (Suc n)*
** (1 / real (Suc n))), erule ssubst*)
apply(*subgoal-tac ln (real (Suc (Suc n))) * (ln (real (Suc (Suc n)))) −*
*ln (real (Suc n)) * (ln (real (Suc n))) <= 2 * ln (real (Suc n)) / (real (Suc n))*
*+ 1 / (real (Suc n) * (real (Suc n))))*
apply(*erule order-trans*)
apply(*simp*)
apply(*rule real-div-pos-le-mono*)
apply *force*
apply (*rule mult-pos*)
apply *force*
apply *force*
apply (*rule ln-sq-diff-ubound*)
apply (*auto simp add*: *ring-eq-simps add-divide-distrib* [*THEN sym*])
done

lemma *ln-sq-diff-bigo*: (%*i*. *ln* (*real* (*Suc* (*Suc i*))) * (*ln* (*real* (*Suc* (*Suc i*)))) − *ln* (*real* (*Suc i*)) * (*ln* (*real* (*Suc i*)))) : (%*i*. *2* * *ln* (*real* (*Suc i*)) / (*real* (*Suc i*)) − *2* * ((*ln* (*real* (*Suc i*)) + *1*) / (*real* (*Suc i*)) * (*1* / (*real* (*Suc i*))))) +*o* *O*(%*i*. (*ln* (*real* (*Suc i*)) + *1*)/ (*real* (*Suc i*) * (*real* (*Suc i*))))
proof −
have (%*i*. *ln* (*real* (*Suc* (*Suc i*))) * *ln* (*real* (*Suc* (*Suc i*))) − *ln* (*real* (*Suc i*)) * *ln* (*real* (*Suc i*))) : (%*i*. *2* * *ln* (*real* (*Suc i*)) / *real* (*Suc i*) − *2* * ((*ln* (*real* (*Suc i*)) + *1*) / *real* (*Suc i*) * (*1* / *real* (*Suc i*)))) +*o* *O*(%*i*. *3* * ((*ln* (*real* (*Suc i*)) + *1*) / *real* (*Suc i*) * (*1* / *real* (*Suc i*))))
apply (*rule bigo-bounded3*)
apply (*rule ln-sq-diff-all-bounds*)
done
also have *O*(%*i*. *3* * ((*ln* (*real* (*Suc i*)) + *1*) / *real* (*Suc i*) * (*1* / *real* (*Suc i*)))) = *O*(%*i*. (*ln* (*real* (*Suc i*)) + *1*) / *real* (*Suc i*) * (*1* / *real* (*Suc i*)))
apply (*rule bigo-const-mult*)
apply *simp*
done
also have ... = *O*(%*i*. (*ln* (*real* (*Suc i*)) + *1*)/ (*real* (*Suc i*) * (*real* (*Suc i*))))
by *simp*
finally show *?thesis*.
qed

lemma *ln-sq-diff-bigo-subset*: (%*i*. *2* * *ln* (*real* (*Suc i*)) / (*real* (*Suc i*)) − *2* * ((*ln* (*real* (*Suc i*)) + *1*) / (*real* (*Suc i*)) * (*1* / (*real* (*Suc i*)))) +*o* *O*(%*i*. (*ln* (*real* (*Suc i*)) + *1*)/ (*real* (*Suc i*) * (*real* (*Suc i*)))) <= (%*i*. *2* * *ln* (*real* (*Suc i*)) / (*real* (*Suc i*))) +*o* *O*(%*i*. (*ln* (*real* (*Suc i*)) + *1*)/ (*real* (*Suc i*) * (*real* (*Suc i*))))
apply(*simp add*: *bigo-def elt-set-plus-def func-plus*)
apply(*clarify*)
apply(*rule-tac x* = (%*x*. (− *2*) * (*ln* (*real* (*Suc x*)) + *1*) / (*real* (*Suc x*) * *real* (*Suc x*)) + *b x*) **in** *exI*)
apply(*rule conjI*)
apply(*rule-tac x* = *2* + *c* **in** *exI*)
apply(*rule allI*)
apply(*simp only*: *abs-mult left-distrib*)
apply(*subgoal-tac abs* ((− *2*) * (*ln* (*real* (*Suc xa*)) + *1*) / (*real* (*Suc xa*) * *real* (*Suc xa*)) + *b xa*) <= *abs* ((− *2*) * (*ln* (*real* (*Suc xa*)) + *1*) / (*real* (*Suc xa*) * *real* (*Suc xa*))) + *abs* (*b xa*))

apply(*erule order-trans*)
apply(*rule add-mono*)
apply(*simp only*: *abs-mult mult-assoc real-divide-def*)
apply(*force*)
apply(*erule-tac x = xa* **in** *allE*)
apply(*simp*)
apply(*simp add*: *abs-triangle-ineq*)
apply(*rule ext*)
apply(*simp add*: *real-divide-def*)
apply(*simp only*: *minus-mult-left minus-add-distrib*)
apply *simp*
done

lemma *ln-sq-diff-bigo2*: (%*i*. *ln* (*real* (*Suc* (*Suc i*))) ∗ (*ln* (*real* (*Suc* (*Suc i*)))) −
ln (*real* (*Suc i*)) ∗ (*ln* (*real* (*Suc i*)))) : (%*i*. *2* ∗ *ln* (*real* (*Suc i*)) / (*real* (*Suc i*))) +*o* *O*(%*i*. (*ln* (*real* (*Suc i*)) + *1*)/ (*real* (*Suc i*) ∗ (*real* (*Suc i*))))
apply(*insert ln-sq-diff-bigo-subset*)
apply(*erule subsetD*)
by(*simp only*: *ln-sq-diff-bigo*)

lemma *ln-sq-sum-bigo*: (%*n*. *ln* (*real* (*Suc n*)) ∗ (*ln* (*real* (*Suc n*)))) :
(%*n*. *2* ∗ *sumr 0 n* (%*i*. *ln* (*real* (*Suc i*)) / (*real* (*Suc i*)))) +*o*
O(%*n*. *sumr 0 n* (%*i*. (*ln* (*real* (*Suc i*)) + *1*)/ (*real* (*Suc i*) ∗ (*real* (*Suc i*)))))
apply(*subgoal-tac* (%*n*. *2* ∗
sumr 0 n (%*i*. *ln* (*real* (*Suc i*)) / *real* (*Suc i*))) +*o*
O(%*n*. *sumr 0 n*
(%*i*. (*ln* (*real* (*Suc i*)) + *1*) /
(*real* (*Suc i*) ∗ *real* (*Suc i*)))) =
(%*n*. *sumr 0 n* (%*i*. *2* ∗ (*ln* (*real* (*Suc i*)) / *real* (*Suc i*)))) +*o*
O(%*n*. *sumr 0 n*
(%*i*. (*ln* (*real* (*Suc i*)) + *1*) /
(*real* (*Suc i*) ∗ *real* (*Suc i*))))))
apply(*erule ssubst*)
apply(*subgoal-tac* (%*n*. *ln* (*real* (*Suc n*)) ∗ *ln* (*real* (*Suc n*))) =
(%*n*. *sumr 0 n*
(%*i*. *ln* (*real* (*Suc* (*Suc i*))) ∗ *ln* (*real* (*Suc* (*Suc i*))) −
ln (*real* (*Suc i*)) ∗ *ln* (*real* (*Suc i*)))))
apply(*erule ssubst*)
apply(*rule bigo-sumr-pos2*)
apply(*rule allI*)
apply(*simp add*: *real-divide-def*)
apply(*simp only*: *order-le-less*)
apply(*clarify*)
apply(*rule real-mult-order*)
apply(*subgoal-tac 0* <= *ln* (*real* (*Suc n*)))

apply *arith*
apply (*rule ln-ge-zero*)
apply *force*
apply (*rule mult-pos*)
apply *force*
apply *force*
apply(*insert ln-sq-diff-bigo2*)
apply(*simp*)
apply(*rule ext*)
apply(*subst ln-sq-sum*)
apply(*simp*)
by(*simp only*: *sumr-mult*)

lemma *calc2*: **assumes** *a*: *0* < *x*
  shows *ln x* − *ln* (*x* + *1*) = *ln* (*1* − *1* / (*x* + *1*))
proof −
 have *b*: *0* < *x* + *1*
  by (*auto simp only*: *real-add-order a*)
 from *a* **have** *c*: *x* ~= *0* **by** *auto*
 from *b* **have** *d*: *x* + *1* ~= *0* **by** *auto*
 have *ln x* − *ln* (*x* + *1*) = *ln* (*x* / (*x* + *1*))
  by (*rule ln-div* [*THEN sym*], *rule a*, *rule b*)
 also have *x* / (*x* + *1*) = *1* − *1*/(*x* + *1*)
 proof −
  have *x* = *x* + *1* − *1*
   by *auto*
  then have *x* / (*x* + *1*) = (*x* + *1* − *1*) / (*x* + *1*)
   by *simp*
  also have ... = (*x* + *1*) / (*x* + *1*) − *1* / (*x* + *1*)
   by (*rule real-minus-divide-distrib*)
  also have (*x* + *1*) / (*x* + *1*) = *1*
   by (*rule divide-self*, *rule d*)
  finally show *?thesis*.
 qed
 finally show *?thesis*.
qed

lemma *calc3*: **assumes** *a*: *0* < *x*
  shows (*ln x* / *x*) − (*ln* (*x* + *1*) / (*x* + *1*)) =
    ln (*1* − (*1*/(*x*+*1*))) / (*x* + *1*) + *ln x* / (*x* ∗ (*x* + *1*))
proof −
 have *b*: *0* < *x* + *1*
  by (*auto simp only*: *real-add-order a*)
 from *a* **have** *c*: *x* ~= *0* **by** *auto*
 from *b* **have** *d*: *x* + *1* ~= *0* **by** *auto*

```
  have ln x / x = ((x + 1) * ln x) / ((x + 1) * x)
    by (subst mult-divide-cancel-left, rule d, rule refl)
  also have (x + 1) * x = x * (x + 1)
    by auto
  finally have ln x / x = (x + 1) * ln x / (x * (x + 1)).
  then have ln x / x - (ln (x + 1) / (x + 1)) =
      (x + 1) * ln x / (x * (x + 1)) - (ln (x + 1) / (x + 1))
    by simp
  also have ... = (x + 1) * ln x / (x * (x + 1)) -
      (x * ln (x + 1) / (x *(x + 1)))
    by (subst mult-divide-cancel-left, rule c, rule refl)
  also have ... = ((x + 1) * ln x - x * ln(x + 1)) / (x * (x + 1))
    by (rule real-minus-divide-distrib [THEN sym])
  also have (x + 1) * ln x = x * ln x + ln x
    by (auto simp add: real-add-mult-distrib)
  also have x * ln x + ln x - x * ln (x + 1) =
      x * ln x - x * ln (x + 1) + ln x
    by simp
  also have x * ln x - x * ln (x + 1) = x * (ln x - ln (x + 1))
    by (rule right-diff-distrib [THEN sym])
  also have ln x - ln (x + 1) = ln (1 - 1/(x + 1))
    by (rule calc2, rule a)
  finally have ln x / x - ln (x + 1) / (x + 1) =
      (x * ln (1 - 1 / (x + 1)) + ln x) / (x * (x + 1)).
  also have ... = x * ln (1 - 1 / (x + 1)) / (x * (x + 1)) +
      ln x / (x * (x + 1))
    by (rule add-divide-distrib)
  also have x * ln (1 - 1 / (x + 1)) / (x * (x + 1)) =
      ln (1 - 1 / (x + 1)) / (x + 1)
    by (subst mult-divide-cancel-left, rule c, rule refl)
  finally show ?thesis.
qed

lemma ln-one-minus-small-pos-lower-bound2:
  0 <= x ==> x <= (1 / 2) ==> - 2 * x <= ln (1 - x)
proof -
  assume a: 0 <= (x::real)
  assume b: x <= (1 / 2)
  then have c: x < 1
    by auto
  have d: 2 * x^2 <= x
  proof -
    have x^2 = x * x
      by (rule realpow-two2 [THEN sym])
    also have ... <= (1 / 2) * x
```

```
    by (rule mult-right-mono)
   finally have x^2 <= (1 / 2) * x.
   then have 2 * x^2 <= 2 * ((1 / 2) * x)
    by auto
   also have ... = x
    by simp
   finally show ?thesis.
  qed
  then have - x - x <= - x - 2 * x^2
   by auto
  also have ... <= ln(1 - x)
   apply (rule ln-one-plus-neg-lower-bound)
   by (rule a, rule b, rule c)
  also have -x - x = - 2 * x
   by simp
  finally show ?thesis.
qed


lemma aux-calc4: 1 <= x ==> - (2 / ((x + 1) * (x + 1))) <= ln (1 - 1 / (x
+ 1)) / (x + 1)
apply(insert ln-one-minus-small-pos-lower-bound2[of 1 / (x + 1)])
apply(simp)
apply(subgoal-tac 2 / (x + 1) <= 1)
apply(simp)
apply(subgoal-tac - (2 / ((x + 1) * (x + 1))) = -2 / (x + 1) / (x + 1))
apply(erule ssubst)
apply(simp only: real-divide-def)
apply(force)
apply (simp add: divide-divide-eq-left nonzero-minus-divide-left)
apply (rule real-le-mult-imp-div-pos-le)
apply arith
apply simp
done

lemma aux-calc4-neg: 1 <= x ==> - (ln (1 - 1 / (x + 1)) / (x + 1)) <= 2
/ ((x + 1) * (x + 1))
  apply(insert aux-calc4[of x])
  apply simp
  done

lemma calc4: ln (real (Suc n)) / (real (Suc n) * (real (Suc n))) <= 2 * (ln (real
(Suc n)) / (real (Suc n)) - (ln (real (Suc (Suc n))) / (real (Suc (Suc n))))) + 4
/ (real (Suc (Suc n)) * (real (Suc (Suc n))))
proof -
```

have *ln (real (Suc n)) / (real (Suc n) * (real (Suc n))) <= 2 * ln (real (Suc n)) / (real (Suc n) * (real (Suc (Suc n))))*

apply(*subgoal-tac (ln (real (Suc n)) / (real (Suc n) * real (Suc n)) <= 2 * ln (real (Suc n)) / (real (Suc n) * real (Suc (Suc n)))) = ((ln (real (Suc n)) / (real (Suc n) * real (Suc n))) / 2 <= ln (real (Suc n)) / (real (Suc n) * real (Suc (Suc n))))*)

apply(*erule ssubst*)

apply(*simp*)

apply(*subgoal-tac real (Suc (Suc n)) = real (Suc n) + 1*)

apply(*erule ssubst*)

apply(*simp only*: *left-distrib real-divide-def*)

apply(*rule mult-left-mono*)

apply(*rule real-inverse-le-swap*)

apply(*simp only*: *real-of-nat-ge-zero real-of-nat-Suc*)

apply (*rule mult-pos*)

apply *force*

apply *force*

apply (*subst mult-assoc*)

apply (*rule mult-left-mono*)

apply (*simp-all*)

apply (*subgoal-tac ln (real (Suc n)) / (real (Suc n) * real (Suc n)) = 2 * ((ln (real (Suc n)) / (real (Suc n) * real (Suc n) * 2))))*

apply (*erule ssubst*)

apply (*subgoal-tac 2 * ln (real (Suc n)) / (real (Suc n) * real (Suc (Suc n))) = 2 * ((ln (real (Suc n)) / (real (Suc n) * real (Suc (Suc n)))))*)

apply (*erule ssubst*)

apply (*subst mult-le-cancel-left*)

apply *auto*

done

also have *... = 2 * (ln (real (Suc n))/(real (Suc n)) − (ln (real (Suc (Suc n))) / (real (Suc (Suc n))))) − 2 * (ln (1 − 1 / (real (Suc (Suc n)))) / (real (Suc (Suc n))))*

apply(*subgoal-tac ln (real (Suc (Suc n))) / real (Suc (Suc n)) = ln (real (Suc n) + 1) / (real (Suc n) + 1)*)

apply(*erule ssubst*)

apply(*subgoal-tac ln (real (Suc n)) / real (Suc n) − ln (real (Suc n) + 1) / (real (Suc n) + 1) = ln (1 − (1/((real (Suc n))+1))) / ((real (Suc n)) + 1) + ln (real (Suc n)) / ((real (Suc n)) * ((real (Suc n)) + 1)))*

apply(*erule ssubst*)

apply(*simp add*: *real-of-nat-Suc*)

apply(*rule calc3*)

apply(*simp add*: *real-of-nat-ge-zero*)

by(*simp add*: *real-of-nat-Suc*)

also have ... <= *2 * (ln (real (Suc n))/(real (Suc n)) − (ln (real (Suc (Suc n))) / (real (Suc (Suc n))))) + 4 / (real (Suc (Suc n)) * (real (Suc (Suc n))))*
apply(*simp*)
apply(*subgoal-tac − (2 * ln (1 − 1 / real (Suc (Suc n))) / real (Suc (Suc n))) =*
*2 * (− (ln (1 − 1 / (real (Suc n) + 1)) / (real (Suc n) + 1))))*
apply(*erule ssubst*)
apply(*subgoal-tac 4 / (real (Suc (Suc n)) * real (Suc (Suc n))) =*
*2 * (2 / ((real (Suc n) + 1) * (real (Suc n) + 1))))*
apply(*erule ssubst*)
apply(*simp only*: *real-mult-le-cancel-iff2*)
apply(*rule aux-calc4-neg*)
apply(*simp add*: *real-of-nat-Suc*)
apply(*simp add*: *real-of-nat-Suc*)
by(*simp add*: *real-of-nat-Suc*)
finallyshow *?thesis*.
qed

lemma *aux-bigo-calc*[*rule-format*]: *sumr 0 x (%i. ln (real (Suc i)) / (real (Suc i) * real (Suc i))) <= 8*
proof −
have *sumr 0 x (%i. ln (real (Suc i)) / (real (Suc i) * real (Suc i))) <=*
*sumr 0 x (%n. 2 * (ln (real (Suc n)) / (real (Suc n)) − (ln (real (Suc (Suc n))) / (real (Suc (Suc n))))) + 4 / (real (Suc (Suc n)) * (real (Suc (Suc n)))))*
apply(*rule sumr-le2*)
apply(*rule allI*)
apply(*simp only*: *calc4*)
by(*simp*)
also have ... <= *sumr 0 x (%n. 2 * (ln (real (Suc n)) / (real (Suc n)) − (ln (real (Suc (Suc n))) / (real (Suc (Suc n)))))) + sumr 0 x (%n. 4 / (real (Suc (Suc n)) * (real (Suc (Suc n)))))*
apply (*subst sumr-add*)
apply (*rule order-refl*)
done
also have *sumr 0 x*
*(%n. 2 * (ln (real (Suc n)) / real (Suc n) −*
ln (real (Suc (Suc n))) / real (Suc (Suc n)))) +
*sumr 0 x (%n. 4 / (real (Suc (Suc n)) * real (Suc (Suc n)))) <= sumr 0 x (%n. 4 / (real (Suc (Suc n)) * real (Suc (Suc n))))*
apply(*simp*)

apply(*subgoal-tac sumr 0 x*
*(%n. 2 * ln (real (Suc n)) / real (Suc n) −*
*2 * ln (real (Suc (Suc n))) / real (Suc (Suc n))) =*
*2 * (ln (real (Suc 0)) / real (Suc 0) − ln (real (Suc x))/ (real (Suc x))))*

```
    apply(simp add: real-divide-def)
    apply (rule nonneg-times-nonneg)
    apply auto
    apply(induct-tac x)
    apply(simp)
    by(simp (no-asm-simp))
  also have sumr 0 x (%n. 4 / (real (Suc (Suc n)) * real (Suc (Suc n)))) =
    4 * sumr 0 x (%n. 1 / (real (Suc (Suc n))) * (1 / real (Suc (Suc n))))
    by(simp add: sumr-mult)
  also have ... <= 4 * sumr 0 x (%n. 1 / (real (Suc n)) * (1 / real (Suc n)))
    apply(simp)
    apply(rule sumr-le2)
    apply(rule allI)
    apply(auto simp add: real-divide-def)
    apply (rule mult-mono)
    apply auto
    done
  also have ... <= 8
    apply(insert lnsum-inv-sq-2[of x])
    apply(simp)
    apply(erule order-trans)
    by(simp add: real-divide-def)
  finally show sumr 0 x (%i. ln (real (Suc i)) / (real (Suc i) * real (Suc i))) <=
8.
qed

lemma aux2-bigo-calc[rule-format]: sumr 0 x (%i. 1 / (real (Suc i) * real (Suc
i))) <= 2
apply(insert lnsum-inv-sq-2[of x])
apply(simp)
apply(erule order-trans)
by(simp add: real-divide-def)

lemma ln-sq-diff-part-bigo-1: (%n. sumr 0 n (%i. (ln (real (Suc i)) + 1)/ (real
(Suc i) * (real (Suc i))))) : O(%x. 1)
apply(auto simp add: bigo-def)
apply(rule-tac x = 8 + 2 in exI)
apply(rule allI)
apply(subgoal-tac abs (sumr 0 x
              (%i. (ln (real (Suc i)) + 1) /
                   (real (Suc i) * real (Suc i)))) <=
  sumr 0 x (%i. (ln (real (Suc i)))/ (real (Suc i) * real (Suc i))) +
  sumr 0 x (%i. 1 / (real (Suc i) * real (Suc i))))
apply(erule order-trans)
```

apply(*subgoal-tac sumr 0 x (%i. 1 / (real (Suc i) * real (Suc i))) <= 2*)
apply(*rule add-mono*)
apply(*rule aux-bigo-calc*)
apply(*simp*)
apply(*rule aux2-bigo-calc*)
apply(*subgoal-tac 0 <= sumr 0 x (%i. (ln (real (Suc i)) + 1) / (real (Suc i) * real (Suc i)))*)
apply(*auto simp add*: *real-abs-def*)
apply(*simp add*: *sumr-add real-abs-def add-divide-distrib*)
apply(*rule sumr-ge-zero*)
apply(*rule allI*)
apply(*simp add*: *real-divide-def*)
apply(*rule nonneg-times-nonneg*)
apply(*rule nonneg-plus-nonneg*)
apply *auto*
done

declare *subsetI* [*rule del*, *intro*]

lemma *ln-sq-diff-bigo3*: *(%x. sumr 0 x (%i. ln (real (Suc (Suc i))) * (ln (real (Suc (Suc i)))) −*
*ln (real (Suc i)) * (ln (real (Suc i))))) : (%x. sumr 0 x (%i. 2 * ln (real (Suc i)) / (real (Suc i)))) +o O(%x. 1)*
proof −
have *(%x. sumr 0 x*
*(%i. ln (real (Suc (Suc i))) * ln (real (Suc (Suc i))) −*
*ln (real (Suc i)) * ln (real (Suc i))))*
*: (%x. sumr 0 x (%i. 2 * ln (real (Suc i)) / real (Suc i))) +o O(%x. sumr 0 x*
*(%i. (ln (real (Suc i)) + 1)/ (real (Suc i) * (real (Suc i)))))*
apply(*rule bigo-sumr-pos2*)
apply(*rule allI*)
apply(*simp add*: *real-divide-def*)
apply (*rule nonneg-times-nonneg*)
apply (*rule nonneg-plus-nonneg*)
apply *force* **apply** *force* **apply** *force*
by(*simp only*: *ln-sq-diff-bigo2*)
moreover have *O((%x. sumr 0 x*
*(%i. (ln (real (Suc i)) + 1)/ (real (Suc i) * (real (Suc i))))))*
<= O(%x. 1)
apply (*rule bigo-elt-subset*)
by(*simp only*: *ln-sq-diff-part-bigo-1*)
ultimately have *(%x. sumr 0 x*
*(%i. ln (real (Suc (Suc i))) * ln (real (Suc (Suc i))) −*
*ln (real (Suc i)) * ln (real (Suc i))))*
*: (%x. sumr 0 x (%i. 2 * ln (real (Suc i)) / real (Suc i))) +o O(%x. 1)*

```
   by (elim set-plus-mono-b)
 then show ?thesis.
qed

lemma lnxdivx-bigo: (%x. ln (real (Suc x)) * (ln (real (Suc x)))) :
  (%x. sumr 0 x (%i. 2 * ln (real (Suc i)) / (real (Suc i)))) +o O(%x. 1)
apply(insert ln-sq-diff-bigo3)
by(simp only: ln-sq-sum[THEN sym])

lemma lnxdivx2-bigo: (%x. ln (real (Suc x)) * (ln (real (Suc x))) / 2) :
  (%x. sumr 0 x (%i. ln (real (Suc i)) / (real (Suc i)))) +o O(%x. 1)
apply(insert lnxdivx-bigo)
apply(simp add: bigo-def elt-set-plus-def func-plus)
apply(erule exE)
apply(rule-tac x = (%x. 1 / 2 * b x) in exI)
apply(clarify)
apply(rule conjI)
apply(rule-tac x = c in exI)
apply(simp)
apply(rule allI)
apply(erule-tac x = x in allE)
apply(subgoal-tac abs(b x / 2) <= abs(b x))
apply(erule order-trans)
apply(simp)
apply(simp add: real-abs-def)
apply(rule ext)
apply(simp add: ext)
apply(subgoal-tac (%x. ln (real (Suc x)) * ln (real (Suc x))) x =
        (%x. sumr 0 x (%i. 2 * ln (real (Suc i)) / real (Suc i)) + b x) x)
apply(simp)
apply(simp only: mult-commute sumr-mult)
apply(rule sumr-fun-eq)
apply(clarify)
apply(simp add: mult-commute)
by(erule ssubst,simp)

lemma lnxdivx-bigo-final: (%x. sumr 0 x (%i. 2 * ln (real (Suc i)) / (real (Suc
i)))) :
 (%x. ln (real (Suc x)) * (ln (real (Suc x)))) +o O(%x. 1)
by(simp add: bigo-add-commute lnxdivx-bigo)

end
```

35 Identities involving sums and ln, part 4

theory *LnSum4* = *PartialSummation* + *LnSum3*:

declare *subsetI* [*rule del*, *intro*]

35.1 Previous identities, rewritten

lemma *identity-one*: (λn::*nat*. *ln* (*1* + *1* / (*real n* + *1*))) =
(λn. *1* / (*real n* + *1*)) + *O*(λn. *1* / ((*real n*) + *1*) ^ *2*)
apply (*rule set-minus-imp-plus*)
apply (*simp add*: *bigo-def func-minus diff-minus func-plus del*: *abs-nonneg abs-nonpos*)
apply (*rule-tac x* = *1* **in** *exI*)
apply (*simp del*: *abs-nonneg abs-nonpos add*: *diff-minus* [*THEN sym*])
apply (*rule allI*)
apply (*subgoal-tac 1* / (*real x* + *1*) ^ *2* = (*1* / (*real x* + *1*)) ^ *2*)
apply (*erule ssubst*)
apply (*rule abs-ln-one-plus-pos-minus-x-bound2*)
apply *force*
apply (*rule real-le-mult-imp-div-pos-le*)
apply *auto*
apply (*rule real-one-over-pow*)
apply *auto*
done

lemma *identity-two*: (λn. *sumr 0* (*n*+*1*) (λi. *1*/(*real i* + *1*))) =
(λn. *ln*(*real n* + *1*)) + *O*(λn. *1*)
apply (*insert sum-inverse-eq-ln-1*)
apply (*simp add*: *real-of-nat-Suc*)
proof −
assume (λx. *sumr 0 x* (λi. *1* / (*real i* + *1*)))
∈ (λn. *ln* (*real n* + *1*)) + *O*(λx. *1*)
then have (λn. *1* / (*real* (*n*::*nat*) + *1*)) +
(λx. *sumr 0 x* (λi. *1* / (*real i* + *1*))) ∈
(λn. *1* / (*real n* + *1*)) + ((λn. *ln* (*real n* + *1*)) +
O(λx. *1*))
by (*intro set-plus-intro2*)
also have ... = ((λn. *1* / (*real n* + *1*)) + (λn. *ln* (*real n* + *1*)))
+ *O*(λx. *1*)
by (*simp add*: *set-plus-rearranges*)
also have ... = ((λn. *ln* (*real n* + *1*)) + (λn. *1* / (*real n* + *1*))) +
O(λx. *1*)
by (*simp add*: *plus-ac0*)
also have ... = (λn. *ln* (*real n* + *1*)) + ((λn. *1* / (*real n* + *1*))

```
    + O(λx. 1))
   by (simp add: set-plus-rearranges)
 also have ... ⊆ (λn. ln (real n + 1)) + (O(λx. 1) + O(λx. 1))
   apply (rule set-plus-mono)
   apply (rule set-plus-mono4)
   apply (simp add: bigo-def)
   apply (rule-tac x = 1 in exI)
   apply (rule allI)
   apply (rule real-le-mult-imp-div-pos-le)
   by auto
 also have ... ⊆ (λn. ln (real n + 1)) + O(λx. 1)
   apply (rule set-plus-mono)
   by (rule bigo-plus-self-subset)
 also have (λn. 1 / (real n + 1)) +
   (λx. sumr 0 x (λi. 1 / (real i + 1))) =
   (λn. sumr 0 n (λi. 1 / (real i + 1)) + 1 / (real n + 1))
   by (simp add: func-plus plus-ac0)
 finally show (λu. sumr 0 u (λi. 1 / (real i + 1)) +
    1 / (real u + 1)) ∈ (λn. ln (real n + 1)) + O(λx. 1).
qed

lemma identity-three:
 (λn. sumr 0 (n+1) (λi. ln(real i + 1))) =
 ((λn. (real n + 1) * ln(real n + 1)) − (λn. real n)) +
 O(λn. ln (real n + 1))
apply (insert fn-lnx-sum-bigo-ln2)
by (simp add: real-of-nat-Suc)

lemma identity-four:
 (λn. sumr 0 n (λi. ln(real i+1)/(real i + 1))) =
   (λn. (ln (real n+1)) ^2 / 2) + O(λn. 1)
apply (insert lnxdivx-bigo-final)
apply (simp add: real-of-nat-Suc realpow-two2)
apply (drule-tac a = λn. 1 / 2 in set-times-intro2)
apply (simp add: func-times)
apply (subgoal-tac (λx. sumr 0 x (λi. 2 * ln (real i + 1) / (real i + 1)) / 2) =
(λn. sumr 0 n (λi. ln (real i + 1) / (real i + 1))))
prefer 2
apply (rule ext)
apply (induct-tac x)
apply simp
apply (simp add: mult-ac)
apply (simp)
apply (rule set-mp)
prefer 2
```

apply *assumption*
apply (*subgoal-tac* $(\lambda n.\ 1\ /\ 2) \times ((\lambda x.\ (ln\ (real\ x + 1))^2) + O(\lambda x.\ 1)) \subseteq (\lambda n.\ (ln\ (real\ n + 1))^2\ /\ 2) + O(\lambda n.\ 1)$)
apply *force*
apply (*subgoal-tac* $(\lambda n.\ 1\ /\ 2) \times ((\lambda x.\ (ln\ (real\ x + 1))^2)$
$+\ O(\lambda x.\ 1)) \subseteq ((\lambda n.\ 1\ /\ 2) * (\lambda x.\ (ln\ (real\ x + 1))\ \hat{}\ 2)) +$
$((\lambda n.\ 1\ /\ 2) \times O(\lambda x.\ 1))$)
apply (*erule subset-trans*)
apply (*simp add*: *func-times*)
by (*simp add*: *set-times-plus-distrib*)

lemma *identity-five*: $(\lambda n.\ (ln(real\ (n::nat) + 2))\ \hat{}\ 2 -$
$(ln(real\ n + 1))\ \hat{}\ 2) =$
$(\lambda n.\ 2 * ln(real\ n + 1)\ /\ (real\ n + 1)) +$
$O(\lambda n.\ (ln(real\ n + 1) + 1)\ /\ (real\ n + 1)\ \hat{}\ 2)$
apply (*insert ln-sq-diff-bigo2*)
by (*simp add*: *real-of-nat-Suc realpow-two2 plus-ac0*)

theorem *ln-sq-partial-sum*: $sumr\ 0\ (n+1)\ (\%i.\ (ln\ (real\ (i+1)))\ \hat{}\ 2) =$
$(real\ (n + 1))*(ln\ (real\ (n + 1)))\ \hat{}\ 2 - sumr\ 0\ n\ (\%i.\ (real\ (i + 1)) *$
$((ln\ (real\ (i + 2)))\ \hat{}\ 2 - (ln\ (real\ (i+1)))\ \hat{}\ 2))$
apply(*insert partial-sum0*[*of* $\%n.\ (real\ n)\ \%n.\ (1::real)\ n\ \%i.\ (ln\ (real\ i))\ \hat{}\ 2$])
apply(*subgoal-tac ALL n. real n = sumr 0 n* $(\%i.\ 1)$)
apply(*rotate-tac 1*)
apply(*simp* (*no-asm-use*) *del*: *sumr.simps*)
by(*simp*)

lemma *real-nat-plus-two*: $real\ ((n::nat) + 2) = real\ n + 2$
by *auto*

theorem *ln-sq-partial-sum2*: $sumr\ 0\ (n+1)\ (\%i.\ (ln\ ((real\ i)+1))\ \hat{}\ 2) =$
$((real\ n) + 1)*(ln\ ((real\ n) + 1))\ \hat{}\ 2 - sumr\ 0\ n\ (\%i.\ ((real\ i) + 1) *$
$((ln\ ((real\ i) + 2))\ \hat{}\ 2 - (ln\ ((real\ i)+1))\ \hat{}\ 2))$
apply (*insert ln-sq-partial-sum* [*of n*])
apply (*simp only*: *real-nat-plus-one* [*THEN sym*] *real-nat-plus-two* [*THEN sym*])
done

lemma *identity-nine*: $(\lambda n.\ sumr\ 0\ n\ (\lambda i.\ (real\ i + 1) *$
$((ln\ (real\ i + 2))\ \hat{}\ 2 - (ln\ (real\ i + 1))\ \hat{}\ 2))) =$
$(\lambda n.\ 2 * sumr\ 0\ n\ (\lambda i.\ ln(real\ i+1))) +$
$O(\lambda n.\ sumr\ 0\ n\ (\lambda i.\ ln(real\ i+1)\ /\ (real\ i + 1))) +$
$O(\lambda n.\ sumr\ 0\ n\ (\lambda i.\ 1\ /\ (real\ i + 1)))$
proof −
from *identity-five* **have**
$(\lambda n.\ (real\ n + 1)) * (\lambda n.\ (ln(real\ (n::nat) + 2))\ \hat{}\ 2 -$

$(ln(real\ n\ +\ 1))\ \hat{}\ 2) =$
$(\lambda n.\ (real\ n\ +\ 1)) \times$
$((\lambda n.\ 2 * ln(real\ n\ +\ 1)\ /\ (real\ n\ +\ 1)) +$
$O(\lambda n.\ (ln(real\ n\ +\ 1)\ +\ 1)\ /\ (real\ n\ +\ 1)\ \hat{}\ 2))$
by *auto*
also have $(\lambda n.\ (real\ n\ +\ 1)) * (\lambda n.\ (ln(real\ (n::nat)\ +\ 2))\ \hat{}\ 2\ -$
$(ln(real\ n\ +\ 1))\ \hat{}\ 2) = (\lambda n.\ (real\ n\ +\ 1) * ((ln(real\ (n::nat)\ +\ 2))\ \hat{}\ 2\ -$
$(ln(real\ n\ +\ 1))\ \hat{}\ 2))$
by (*simp add*: *func-times*)
also have $(\lambda n.\ (real\ n\ +\ 1)) \times$
$((\lambda n.\ 2 * ln(real\ n\ +\ 1)\ /\ (real\ n\ +\ 1)) +$
$O(\lambda n.\ (ln(real\ n\ +\ 1)\ +\ 1)\ /\ (real\ n\ +\ 1)\ \hat{}\ 2)) =$
$((\lambda n.\ (real\ n{+}1)) * (\lambda n.\ 2 * ln(real\ n\ +\ 1)\ /\ (real\ n\ +\ 1)))$
$+$
$((\lambda n.\ (real\ n\ +\ 1)) \times$
$O(\lambda n.\ (ln(real\ n\ +\ 1)\ +\ 1)\ /\ (real\ n\ +\ 1)\ \hat{}\ 2))$
by (*rule set-times-plus-distrib*)
also have $(\lambda n{::}nat.\ (real\ n{+}1)) * (\lambda n.\ 2 * ln(real\ n\ +\ 1)\ /\ (real\ n\ +\ 1))$
$= (\lambda n.\ 2 * ln(real\ n\ +\ 1))$
apply (*auto simp add*: *func-times*)
done
also have $(\lambda u.\ 2 * ln\ (real\ u\ +\ 1)) +$
$(\lambda n.\ real\ n\ +\ 1) \times O(\lambda n.\ (ln\ (real\ n\ +\ 1)\ +\ 1)\ /\ (real\ n\ +\ 1)^2)$
$\subseteq (\lambda u.\ 2 * ln\ (real\ u\ +\ 1)) +$
$O((\lambda n.\ real\ n\ +\ 1) * (\lambda n.\ (ln\ (real\ n\ +\ 1)\ +\ 1)\ /\ (real\ n\ +\ 1)^2))$
proof $-$
have $(\lambda n.\ real\ n\ +\ 1) \times$
$O(\lambda n.\ (ln\ (real\ n\ +\ 1)\ +\ 1)\ /\ (real\ n\ +\ 1)^2) \subseteq$
$O((\lambda n.\ real\ n\ +\ 1) * (\lambda n.\ (ln\ (real\ n\ +\ 1)\ +\ 1)\ /\ (real\ n\ +\ 1)^2))$
by *auto*
thus *?thesis* **by** *auto*
qed
also have $(\lambda n.\ real\ n\ +\ 1) *$
$(\lambda n.\ (ln\ (real\ n\ +\ 1)\ +\ 1)\ /\ (real\ n\ +\ 1)^2) =$
$(\lambda n.\ (ln\ (real\ n\ +\ 1)\ +\ 1)\ /\ (real\ n\ +\ 1))$
by (*auto simp add*: *func-times realpow-two2* [*THEN sym*])
finally have $(\lambda i{::}nat.\ (real\ i\ +\ 1) *$
$((ln\ (real\ i\ +\ 2))^2\ -\ (ln\ (real\ i\ +\ 1))^2)) =$
$(\lambda i.\ 2 * ln\ (real\ i\ +\ 1)) +$
$O(\lambda i.\ (ln\ (real\ i\ +\ 1)\ +\ 1)\ /\ (real\ i\ +\ 1))$**.**
then have $(\lambda n.\ sumr\ 0\ n\ (\lambda i{::}nat.\ (real\ i\ +\ 1) *$
$((ln\ (real\ i\ +\ 2))^2\ -\ (ln\ (real\ i\ +\ 1))^2))) =$
$(\lambda n.\ sumr\ 0\ n\ (\lambda i.\ 2 * ln\ (real\ i\ +\ 1))) +$
$O(\lambda n.\ sumr\ 0\ n\ (\lambda i.\ (ln\ (real\ i\ +\ 1)\ +\ 1)\ /\ (real\ i\ +\ 1)))$
apply (*intro bigo-sumr-pos2*)

```
    apply auto
    apply (rule real-ge-zero-div-gt-zero)
    apply (rule real-le-add-order)
    apply force+
    done
  also have (λn. sumr 0 n (λi. 2 * ln (real i + 1))) =
      (λn. 2 * sumr 0 n (λi. ln(real i + 1)))
    apply (rule ext)
    by (simp add: sumr-mult)
  also have (λn. 2 * sumr 0 n (λi. ln (real i + 1))) +
      O(λn. sumr 0 n (λi. (ln (real i + 1) + 1) / (real i + 1))) =
      (λn. 2 * sumr 0 n (λi. ln (real i + 1))) +
      (O(λn. sumr 0 n (λi. (ln (real i + 1)) / (real i + 1))) +
       O(λn. sumr 0 n (λi. 1 / (real i + 1))))
    apply (auto)
    apply (rule subset-trans)
    prefer 2
    apply (rule bigo-plus-subset)
    apply (subgoal-tac (λn. sumr 0 n (λi. (ln (real i + 1) + 1) /
        (real i + 1))) = (λn. sumr 0 n (λi. ln (real i + 1) / (real i + 1))) +
        (λn. sumr 0 n (λi. 1 / (real i + 1))))
    apply simp
    apply (simp add: func-plus, rule ext)
    apply (subgoal-tac (λi. (ln (real i + 1) + 1) / (real i + 1)) =
        (λi. (ln (real i + 1) / (real i + 1)) + 1 / (real i + 1)))
    apply (erule ssubst)
    apply (rule sumr-add [THEN sym])
    apply (rule ext)
    by (rule add-divide-distrib)
  finally show ?thesis
    by (simp add: set-plus-rearrange3)
qed

lemma aux1: (λn. 1) ∈ O(λn. (ln(real n + 1) ^2 + 1))
  apply (rule bigo-bounded)
  by auto

lemma aux2: (n::nat) ~= 0 ⟹ ln 2 ≤ ln(real n + 1)
proof -
  assume (n::nat) ~= 0
  then have 0 < n by auto
  then have 2 ≤ n + 1 by auto
  then have real (2::nat) ≤ real (n + 1)
    by (simp add: real-of-nat-le-iff [THEN sym])
  then have real (2::nat) ≤ real n + 1
```

```
    by (simp only: real-nat-plus-one)
  then have ln (real (2::nat)) ≤ ln (real n + 1)
    by auto
  thus ?thesis
    by simp
qed

lemma aux3: (λn::nat. ln(real n + 1)) ∈ O(λn. (ln(real n + 1) ^ 2 + 1))
  apply (rule bigo-bounded-alt)
  apply auto
  apply (subgoal-tac ln (real x + 1) ≤ (1 / ln 2) *
    (ln (real x + 1) ^ 2 + 1))
  apply assumption
  apply (simp add: ring-distrib)
  apply (subgoal-tac ln (real x + 1) ≤ (ln (real x + 1)) ^ 2 / ln 2)
  apply (erule order-trans)
  apply auto
  apply (rule real-mult-le-imp-le-div-pos)
  apply auto
  apply (subst realpow-two2 [THEN sym])
  apply (case-tac x = 0)
  apply simp
  apply (rule mult-mono)
  apply auto
  done

lemma first-term-a: (λn. sumr 0 n (λi. ln(real i + 1))) =
  ((λn. (real n + 1) * ln (real n + 1)) -
    (λn. real n)) + O(λn. (ln(real n + 1) ^ 2 + 1))
proof -
  have (λn. sumr 0 n (λi. ln(real i + 1))) =
    (λn. sumr 0 (n + 1) (λi. ln(real i + 1))) - (λn. ln (real n + 1))
    by (simp add: diff-minus func-minus func-plus)
  also have ... = - (λn. ln (real n + 1)) +
    (λn. sumr 0 (n + 1) (λi. ln(real i + 1)))
    by (simp add: diff-minus add-ac)
  also from identity-three have ... = - (λn. ln (real n + 1)) +
    (((λn. (real n + 1) * ln(real n + 1)) - (λn. (real n))) +
      O(λn. ln (real n + 1)))
    by (rule set-plus-intro2)
  also have ... = ((λn. (real n + 1) * ln (real n + 1)) - (λn. real n))
        +
        ( -(λn. ln (real n + 1)) + O(λn. ln (real n + 1)))
     by (simp add: ring-eq-simps set-plus-rearranges)
  also from aux3 have ... = ((λn::nat. (real n + 1) * ln (real n + 1))
```

```
    − (λn. real n)) +
    O(λn. (ln (real n + 1)) ^ 2 + 1)
    by auto
  finally show ?thesis.
qed

lemma first-term: (λn. 2 ∗ sumr 0 n (λi. ln(real i + 1))) =
  ((λn. 2∗ (real n + 1) ∗ ln (real n + 1)) −
    (λn. 2∗ (real n))) + O(λn. (ln(real n + 1) ^ 2 + 1))
proof −
  note first-term-a
  then have (λn. 2) ∗ (λn. sumr 0 n (λi. ln (real i + 1))) =
    (λn. 2) × (((λn. (real n + 1) ∗ ln (real n + 1)) −
      (λn. (real n))) +
      O(λn. (ln (real n + 1))^2 + 1)) (is ?LHS ∈ ?RHS)
    by (intro set-times-intro2)
  also have ?LHS = (λn. 2 ∗ sumr 0 n (λi. ln(real i + 1)))
    by (simp add: func-times)
  also have ?RHS = ((λn. 2∗ (real n + 1) ∗ ln (real n + 1)) −
    (λn. 2∗ (real n))) + O(λn. (ln(real n + 1) ^ 2 + 1))
    by (simp add: set-times-plus-distribs func-times ring-distrib
      diff-minus func-minus func-plus mult-ac)
  finally show ?thesis.
qed

lemma second-term: O(λn. sumr 0 n (λi. ln(real i + 1) / (real i + 1)))
  = O(λn. (ln(real n + 1)) ^ 2 + 1)
proof −
  from identity-four have
    O(λn. sumr 0 n (λi. ln(real i + 1) / (real i + 1))) ⊆
      O(λn. (ln(real n + 1)) ^ 2/2) + O(λn. 1)
    by auto
  also have ... ⊆ O(λn. (ln(real n + 1)) ^ 2 + 1)
    proof −
      have O(λn. (ln(real n + 1)) ^ 2/2) ⊆ O(λn. (ln(real n + 1)) ^ 2 + 1)
        apply (rule bigo-elt-subset)
        apply (rule bigo-bounded)
        apply force
        apply auto
        apply (subgoal-tac −2 ≤ 0)
        apply (erule order-trans)
        by auto
      with aux1 show ?thesis
        by auto
    qed
```

```
  finally show ?thesis.
qed

lemma third-term: O(λn. sumr 0 n (λi. 1 / (real i + 1))) =
    O(λn. (ln(real n + 1)) ^2 + 1)
proof −
  have (λn. sumr 0 n (λi. 1 / (real i + 1))) =
      −(λn. 1 / (real n + 1)) + (λn. sumr 0 (n+1) (λi. 1 / (real i + 1)))
    by (simp add: func-plus func-minus)
  also from identity-two have ... ∈ −(λn. 1 / (real n + 1)) +
    ((λn. ln(real n + 1)) + O(λn. 1))
    by auto
  also have ... ⊆ O(λn. (ln(real n + 1)) ^2 + 1)
    proof −
      have −(λn::nat. 1 / (real n + 1)) ∈ O(λn. (ln(real n + 1)) ^2 + 1)
        apply (rule bigo-minus)
        apply (rule bigo-bounded)
        apply auto
        apply (subgoal-tac 1 / (real x + 1) ≤ 1)
        apply (erule order-trans)
        apply auto
        apply (rule real-le-mult-imp-div-pos-le)
        apply (rule real-nat-plus-one-gt-zero)
        by auto
      with aux1 aux3 show ?thesis by auto
    qed
  finally show ?thesis by auto
qed

lemma penultimate-equation: (λn. sumr 0 n (λi. (real i + 1) *
  ((ln(real i + 2)) ^2 − (ln(real i + 1)) ^2))) =
  ((λn. 2 * (real n + 1) * ln(real n + 1)) − (λn. 2 * (real n))) +
  O(λn. (ln (real n + 1)) ^2 + 1) (is ?LHS = ?RHS)
proof −
  from identity-nine have
      ?LHS =
      (λn. 2 * sumr 0 n (λi. ln(real i+1))) +
        (O(λn. sumr 0 n (λi. ln(real i+1) / (real i + 1))) +
        O(λn. sumr 0 n (λi. 1 / (real i + 1))))
    by (simp add: set-plus-rearranges)
  also from second-term third-term have ... ⊆
    (λn. 2 * sumr 0 n (λi. ln(real i+1))) +
    O(λn. (ln (real n + 1)) ^2 + 1)
    by auto
  also from first-term have ... ⊆
```

```
    (((λn. 2* (real n + 1) * ln (real n + 1)) −
    (λn. 2* (real n))) + O(λn. (ln(real n + 1) ^2 + 1))) +
    O(λn. (ln (real n + 1)) ^2 + 1)
    by auto
  also have ... =
    ((λn. 2* (real n + 1) * ln (real n + 1)) −
    (λn. 2* (real n))) + (O(λn. (ln(real n + 1) ^2 + 1)) +
    O(λn. (ln (real n + 1)) ^2 + 1))
    by (simp add: set-plus-rearranges)
  also have ... ⊆ ((λn. 2* (real n + 1) * ln (real n + 1)) −
    (λn. 2* (real n))) + O(λn. (ln(real n + 1) ^2 + 1))
    by auto
  finally show ?thesis.
qed

lemma almost-there: (λn. sumr 0 (n+1) (λi. (ln (real i + 1)) ^2)) =
  ((λn. (real n + 1) * (ln(real n + 1)) ^2) −
    (λn. 2 * (real n + 1) * ln(real n + 1)) + (λn. 2 * (real n)))
      + O(λn. (ln (real n + 1)) ^2 + 1)
proof −
  from penultimate-equation have −(λn. sumr 0 n (λi. (real i + 1) *
  ((ln(real i + 2)) ^2 − (ln(real i + 1)) ^2))) =
  −((λn. 2 * (real n + 1) * ln(real n + 1)) − (λn. 2 * (real n))) +
  O(λn. (ln (real n + 1)) ^2 + 1) (is ?LHS = ?RHS)
    by (rule bigo-minus2)
  then have (λn. ((real n) + 1)*(ln ((real n) + 1)) ^2) + ?LHS
      = (λn. ((real n) + 1)*(ln ((real n) + 1)) ^2) + ?RHS
        (is ?LHS2 = ?RHS2)
    by auto
  also from ln-sq-partial-sum2 have ?LHS2 =
      (λn. sumr 0 (n+1) (%i. (ln ((real i)+1)) ^2))
    by (simp add: diff-minus func-minus func-plus ext)
  finally show ?thesis
    by (simp add: diff-minus set-plus-rearranges ring-eq-simps)
qed

lemma aux5: f ∈ O(λn::nat. (ln (real n + 1)) ^2 + 1) ⟹
  f(0) = 0 ⟹ f ∈ O(λn::nat. (ln (real n + 1)) ^2)
  apply (auto simp add: bigo-alt-def)
  apply (rule-tac x = c + c / (ln 2) ^2 in exI)
  apply auto
  apply (subgoal-tac − c < 0)
  apply (subgoal-tac 0 < c / (ln 2) ^2)
  apply (erule order-less-trans)
  apply (assumption)
```

apply (*rule real-mult-less-imp-less-div-pos*)
apply (*auto*)
apply (*case-tac* $x = 0$)
apply (*simp add*: *realpow-two2* [*THEN sym*])
apply (*drule-tac* $x = x$ **in** *spec*)
apply (*subgoal-tac abs*$((ln\ (real\ x + 1))^2 + 1) = (ln\ (real\ x + 1))^2 + 1$)
apply (*simp add*: *left-distrib right-distrib*)
apply (*erule order-trans*)
apply *auto*
apply (*subgoal-tac* $c * 1 \leq c * ((ln\ (real\ x + 1))\ \hat{}\ 2\ /\ (ln\ 2)\ \hat{}\ 2)$)
apply *simp*
apply (*rule mult-left-mono*)
apply (*rule div-ge-1*)
apply *force*
apply (*rule power-mono*)
apply (*rule aux2*)
apply *force*
apply *force*
apply *force*
apply (*rule abs-nonneg*)
apply (*rule nonneg-plus-nonneg*)
apply *auto*
done

lemma *aux6*: $f \in g + O(\lambda n{::}nat.\ (ln\ (real\ n + 1)\ \hat{}\ 2 + 1)) \Longrightarrow$
$f\ 0 = g\ 0 \Longrightarrow f \in g + O(\lambda n.\ (ln\ (real\ n + 1))\ \hat{}\ 2)$
proof −
assume $f \in g + O(\lambda n.\ (ln\ (real\ n + 1)\ \hat{}\ 2 + 1))$
then have *a*: $f - g \in O(\lambda n.\ (ln\ (real\ n + 1)\ \hat{}\ 2 + 1))$
by (*rule set-plus-imp-minus*)
assume $f\ 0 = g\ 0$
then have *b*: $(f - g)0 = 0$
by (*simp add*: *diff-minus func-minus func-plus*)
from *a b* **have** $f - g \in O(\lambda n.\ (ln\ (real\ n + 1)\ \hat{}\ 2))$
by (*intro aux5*)
then show $f \in g + O(\lambda n.\ (ln\ (real\ n + 1)\ \hat{}\ 2))$
by (*rule set-minus-imp-plus*)
qed

theorem *identity-six*: $(\lambda n.\ sumr\ 0\ (n+1)\ (\lambda i.\ (ln\ (real\ i + 1))\ \hat{}\ 2)) =$
$((\lambda n.\ (real\ n + 1) * (ln(real\ n + 1))\ \hat{}\ 2) -$
$(\lambda n.\ 2 * (real\ n + 1) * ln(real\ n + 1)) + (\lambda n.\ 2 * (real\ n)))$
$+ O(\lambda n.\ (ln\ (real\ n + 1))\ \hat{}\ 2)$
apply (*rule aux6*)
apply (*rule almost-there*)

by (*simp add*: *realpow-two2* [*THEN sym*] *diff-minus func-minus func-plus*)

end

36 Identities involving sums and ln, part 5

theory *LnSum5* = *LnSum4*:

lemma *aux1*: (%*n*. *ln*(*real n* + *1*) ∗ *sumr 0* (*n* + *1*) (%*i*. *ln* (*real i* + *1*))) =*o*
((%*n*. (*real n* + *1*) ∗ (*ln* (*real n* + *1*)) ^*2*) − (%*n*. (*real n*) ∗ *ln* (*real n* + *1*)))
+*o* *O*(%*n*. (*ln* (*real n* + *1*)) ^*2*)
proof −
have (%*n*. *ln*(*real n* + *1*) ∗ *sumr 0* (*n* + *1*) (%*i*. *ln* (*real i* + *1*))) =
(%*n*. *ln*(*real n* + *1*)) ∗ (%*n*. *sumr 0* (*n* + *1*) (%*i*. *ln* (*real i* + *1*)))
by (*simp add*: *func-times*)
also from *identity-three* **have** ... =*o* (%*n*. *ln*(*real n* + *1*)) ∗*o*
(((%*n*. (*real n* + *1*) ∗ *ln*(*real n* + *1*)) − (λ*n*. *real n*)) +*o*
O(λ*n*. *ln* (*real n* + *1*)))
by *auto*
also have ... = (%*n*. *ln*(*real n* + *1*)) ∗
((%*n*. (*real n* + *1*) ∗ *ln*(*real n* + *1*)) − (λ*n*. *real n*)) +*o*
(%*n*. *ln*(*real n* + *1*)) ∗*o* *O*(λ*n*. *ln* (*real n* + *1*))
by (*simp add*: *set-times-plus-distribs*)
also have (%*n*. *ln*(*real n* + *1*)) ∗
((%*n*. (*real n* + *1*) ∗ *ln*(*real n* + *1*)) − (λ*n*. *real n*)) =
((%*n*. (*real n* + *1*) ∗ (*ln*(*real n* + *1*)) ^*2*) −
(λ*n*. (*real n*) ∗ *ln*(*real n* + *1*)))
by (*simp add*: *func-times diff-minus func-minus func-plus ring-distrib*
realpow-two2 [*THEN sym*] *mult-ac*)
also have ((%*n*. (*real n* + *1*) ∗ *ln* (*real n* + *1*) ^ *2*) −
(%*n*. *real n* ∗ *ln* (*real n* + *1*))) +*o*
(%*n*. *ln* (*real n* + *1*)) ∗*o* *O*(%*n*. *ln* (*real n* + *1*)) =*s*
((%*n*. (*real n* + *1*) ∗ *ln* (*real n* + *1*) ^ *2*) −
(%*n*. *real n* ∗ *ln* (*real n* + *1*))) +*o* *O*(%*n*. (*ln* (*real n* + *1*)) ^*2*)
apply (*auto simp add*: *func-times realpow-two2* [*THEN sym*])
apply (*subgoal-tac* (%*n*. *ln* (*real n* + *1*)) ∗*o* *O*(%*n*. *ln* (*real n* + *1*))
=*s* *O*((%*n*. *ln* (*real n* + *1*)) ∗ (%*n*. *ln* (*real n* + *1*))))
apply (*simp add*: *func-times*)
by *auto*
finally show *?thesis*.
qed

lemma *aux2*: (%*n*. 2 * *ln*(*real n* + 1) * *sumr* 0 (*n* + 1) (%*i*. *ln* (*real i* + 1)))
=*o*
((%*n*. 2 * (*real n* + 1) * (*ln* (*real n* + 1)) ^2) − (%*n*. 2 * (*real n*) *
ln (*real n* + 1))) +*o* *O*(%*n*. (*ln* (*real n* + 1)) ^2)
(**is** *?LHS* =*o* *?RHS*)
proof −
from *aux1* **have**
(%*n*. 2) * (%*n*. *ln*(*real n* + 1) * *sumr* 0 (*n* + 1) (%*i*. *ln* (*real i* + 1))) =*o*
(%*n*. 2) *o (((%*n*. (*real n* + 1) * (*ln* (*real n* + 1)) ^2) −
(%*n*. (*real n*) * *ln* (*real n* + 1))) +*o* *O*(%*n*. (*ln* (*real n* + 1)) ^2))
(**is** *?LHS1* =*o* *?RHS1*)
by *auto*
also have *?LHS1* = *?LHS*
by (*simp add*: *func-times mult-ac*)
also have *?RHS1* = *?RHS*
by (*simp add*: *func-times diff-minus func-minus func-plus*
ring-distrib set-times-plus-distribs mult-ac)
finally show *?thesis*.
qed

lemma *aux2a*: −(%*n*. 2 * *ln*(*real n* + 1) * *sumr* 0 (*n* + 1)
(%*i*. *ln* (*real i* + 1))) =*o*
(− (%*n*. 2 * (*real n* + 1) * (*ln* (*real n* + 1)) ^2) + (%*n*. 2 * (*real n*) *
ln (*real n* + 1))) +*o* *O*(%*n*. (*ln* (*real n* + 1)) ^2)
proof −
from *aux2* **have** −(%*n*. 2 * *ln*(*real n* + 1) * *sumr* 0 (*n* + 1)
(%*i*. *ln* (*real i* + 1))) =*o*
− ((%*n*. 2 * (*real n* + 1) * (*ln* (*real n* + 1)) ^2) − (%*n*. 2 * (*real n*) *
ln (*real n* + 1))) +*o* *O*(%*n*. (*ln* (*real n* + 1)) ^2)
by (*rule bigo-minus2*)
then show *?thesis*
by (*simp add*: *minus-add-distrib diff-minus*)
qed

lemma *aux3*: ((*x*::*real*) − *y*) ^2 = *x* ^2 − 2 * *x* * *y* + *y* ^2
by (*simp add*: *realpow-two2* [*THEN sym*] *ring-distrib mult-ac*
diff-minus)

lemma *aux4*: (%*n*. *sumr* 0 (*n* + 1) (%*i*. (*ln* ((*real n* + 1) / (*real i* + 1))) ^2)) =
(%*n*. (*real n*+1) * *ln*(*real n* + 1) ^2) −
(%*n*. 2 * *ln*(*real n* + 1) * *sumr* 0 (*n*+1) (%*i*. *ln*(*real i* + 1))) +
(%*n*. *sumr* 0 (*n* + 1) (%*i*. (*ln* (*real i* + 1)) ^2))
apply (*simp only*: *diff-minus func-minus func-plus*)
apply (*rule ext*)
proof −

```
  fix n
  show sumr 0 (n + 1) (%i. ln ((real n + 1) / (real i + 1)) ^ 2) =
        (real n + 1) * ln (real n + 1) ^ 2 +
        - (2 * ln (real n + 1) * sumr 0 (n + 1) (%i. ln (real i + 1))) +
        sumr 0 (n + 1) (%i. (ln (real i + 1)) ^ 2)
  proof -
    have (%i::nat. (ln ((real n + 1) / (real i + 1)))^2) =
     (%i. (ln (real n + 1) - ln(real i + 1))^2)
      apply (rule ext)
      apply (subst ln-div)
      apply (rule real-nat-plus-one-gt-zero)+
      by (rule refl)
    also have ... = (%i. (ln (real n + 1))^2 -
      2 * ln (real n + 1) * ln (real i + 1) + ln(real i + 1)^2)
      apply (rule ext)
      by (rule aux3)
    finally have (%i::nat.
        ln ((real (n::nat) + (1::real)) / (real i + (1::real))) ^ 2) =
      (%i::nat. ln (real n + (1::real)) ^ 2 -
        (2::real) * ln (real n + (1::real)) * ln (real i + (1::real)) +
        ln (real i + (1::real)) ^ 2) (is ?LHS = ?RHS).
    then have sumr 0 (n + 1) (?LHS) = sumr 0 (n + 1) (?RHS)
      by auto
    also have ... = sumr 0 (n+1) (%i. ln (real n + 1) ^ 2) -
      sumr 0 (n+1) (%i. 2 * ln (real n + 1) * ln (real i + 1)) +
      sumr 0 (n+1) (%i. ln (real i + 1) ^ 2)
      by (simp only: sumr-add diff-minus func-minus func-plus
        sumr-minus [THEN sym])
    also have sumr 0 (n+1) (%i. ln (real n + 1) ^ 2) =
      real (n + 1) * ln (real n + 1) ^ 2
      by (rule sumr-const)
    also have real (n + 1) = real n + 1
      by (rule real-nat-plus-one)
    also have sumr 0 (n + 1) (%i. 2 * ln (real n + 1) * ln (real i + 1))
      = 2 * ln (real n + 1) * sumr 0 (n + 1) (%i. ln (real i + 1))
      by (subst sumr-mult, rule refl)
    finally show ?thesis
      by (simp only: diff-minus)
  qed
qed

lemma aux5: (%n. (real n+1) * ln(real n + 1)^2) -
    (%n. 2 * ln(real n + 1) * sumr 0 (n+1) (%i. ln(real i + 1))) =o
      ((%n. - (real n + 1) * ln (real n + 1)^2) +
      (%n. 2 * (real n) * ln(real n + 1))) +o O(%n. ln(real n + 1)^2)
```

proof −
have (%*n*. (*real n*+*1*) ∗ *ln*(*real n* + *1*) ^ *2*) −
(%*n*. *2* ∗ *ln*(*real n* + *1*) ∗ *sumr 0* (*n*+*1*) (%*i*. *ln*(*real i* + *1*))) =
(%*n*. (*real n*+*1*) ∗ *ln*(*real n* + *1*) ^ *2*) + −
(%*n*. *2* ∗ *ln*(*real n* + *1*) ∗ *sumr 0* (*n*+*1*) (%*i*. *ln*(*real i* + *1*)))
by (*simp only*: *diff-minus*)
also from *aux2a* **have** ... =*o* (%*n*. (*real n*+*1*) ∗ *ln*(*real n* + *1*) ^ *2*) +*o*
((− (%*n*. *2* ∗ (*real n* + *1*) ∗ (*ln* (*real n* + *1*)) ^ *2*) + (%*n*. *2* ∗ (*real n*) ∗
ln (*real n* + *1*))) +*o* *O*(%*n*. (*ln* (*real n* + *1*)) ^ *2*))
by *auto*
also have ... = ((%*n*. (*real n*+*1*) ∗ *ln*(*real n* + *1*) ^ *2*) +
(− (%*n*. *2* ∗ (*real n* + *1*) ∗ (*ln* (*real n* + *1*)) ^ *2*)) + (%*n*. *2* ∗ (*real n*) ∗
ln (*real n* + *1*))) +*o* *O*(%*n*. (*ln* (*real n* + *1*)) ^ *2*)
by (*simp add*: *set-plus-rearranges plus-ac0*)
also have (%*n*. (*real n*+*1*) ∗ *ln*(*real n* + *1*) ^ *2*) +
(− (%*n*. *2* ∗ (*real n* + *1*) ∗ (*ln* (*real n* + *1*)) ^ *2*)) + (%*n*. *2* ∗ (*real n*) ∗
ln (*real n* + *1*)) = (%*n*. − (*real n* + *1*) ∗ *ln* (*real n* + *1*) ^ *2*) +
(%*n*. *2* ∗ (*real n*) ∗ *ln*(*real n* + *1*))
apply (*subgoal-tac* (%*n*. (*real n*+*1*) ∗ *ln*(*real n* + *1*) ^ *2*) +
(− (%*n*. *2* ∗ (*real n* + *1*) ∗ (*ln* (*real n* + *1*)) ^ *2*)) =
(%*n*. − (*real n* + *1*) ∗ *ln* (*real n* + *1*) ^ *2*))
apply (*erule ssubst*, *rule refl*)
apply (*auto simp add*: *func-plus func-minus*)
apply (*rule ext*)
apply (*subgoal-tac* (*real x* + *1*) ∗ *ln* (*real x* + *1*) ^ *2* +
− ((*2* ∗ *real x* + *2*) ∗ *ln* (*real x* + *1*) ^ *2*) =
((*real x* + *1*) + − (*2* ∗ *real x* + *2*)) ∗ *ln* (*real x* + *1*) ^ *2*)
apply *force*
by (*simp add*: *ring-distrib*)
finally show *?thesis*.
qed

lemma *aux6*: (λ*n*::*nat*. *ln*(*real n* + *1*)) ∈ *O*(λ*n*. (*ln*(*real n* + *1*) ^ *2*))
apply (*rule bigo-bounded-alt*)
apply *auto*
apply (*subgoal-tac ln* (*real x* + *1*) ≤ (*1* / *ln 2*) ∗ (*ln* (*real x* + *1*)) ^ *2*)
apply *assumption*
apply (*simp add*: *realpow-two2* [*THEN sym*])
apply (*rule real-mult-le-imp-le-div-pos*)
apply *auto*
apply (*case-tac x* = *0*)
apply *auto*
done

theorem *almost-there*:

```
  (%n. sumr 0 (n+1) (%i. ln((real n + 1)/(real i + 1)) ^ 2)) =o
  (%n. 2 * (real n)) +o O(%n. ln(real n + 1) ^ 2)
proof −
  from aux4 identity-six have
    (%n. sumr 0 (n + 1) (%i. (ln ((real n + 1) / (real i + 1))) ^ 2)) =o
        ((%n. (real n+1) * ln(real n + 1) ^ 2) −
        (%n. 2 * ln(real n + 1) * sumr 0 (n+1) (%i. ln(real i + 1)))) +o
        (((λn. (real n + 1) * (ln(real n + 1)) ^ 2) −
         (λn. 2 * (real n + 1) * ln(real n + 1)) + (λn. 2 * (real n)))
         +o O(λn. (ln (real n + 1)) ^ 2))
    by auto
  also have ... = ((%n. (real n + 1) * (ln(real n + 1)) ^ 2) −
    (%n. 2 * (real n + 1) * ln(real n + 1)) + (%n. 2 * (real n))) +o
    (((%n. (real n+1) * ln(real n + 1) ^ 2) −
      (%n. 2 * ln(real n + 1) * sumr 0 (n+1) (%i. ln(real i + 1)))) +o
      O(%n. (ln (real n + 1)) ^ 2))
    by (simp add: set-plus-rearranges plus-ac0)
  also from aux5 have ... =s ((%n. (real n + 1) * (ln(real n + 1)) ^ 2) −
    (%n. 2 * (real n + 1) * ln(real n + 1)) + (%n. 2 * (real n))) +o
    (((%n. − (real n + 1) * ln (real n + 1) ^ 2) +
      (%n. 2 * (real n) * ln(real n + 1))) +o O(%n. ln(real n + 1) ^ 2)
      + O(%n. ln(real n + 1) ^ 2))
    by auto
  also have ... = ((%n. (real n + 1) * (ln(real n + 1)) ^ 2) −
    (%n. 2 * (real n + 1) * ln(real n + 1)) + (%n. 2 * (real n)) +
    (%n. − (real n + 1) * ln (real n + 1) ^ 2) +
    (%n. 2 * (real n) * ln(real n + 1))) +o O(%n. ln(real n + 1) ^ 2)
    by (simp add: diff-minus plus-ac0 set-plus-rearranges
      set-plus-rearrange4)
  also have ((%n. (real n + 1) * (ln(real n + 1)) ^ 2) −
    (%n. 2 * (real n + 1) * ln(real n + 1)) + (%n. 2 * (real n)) +
    (%n. − (real n + 1) * ln (real n + 1) ^ 2) +
    (%n. 2 * (real n) * ln(real n + 1))) = (%n. 2 * (real n)) −
    (%n. 2 * ln(real n + 1))
    apply (simp add: diff-minus func-minus func-plus)
    apply (rule ext)
    by (auto simp add: plus-ac0 ring-distrib)
  also have ((%n. 2 * real n) − (%n. 2 * ln (real n + 1))) +o
    O(%n. (ln (real n + 1)) ^ 2) = (%n. 2 * real n) +o
      (− (%n. 2 * ln (real n + 1)) +o O(%n. (ln (real n + 1)) ^ 2))
    by (auto simp add: diff-minus set-plus-rearranges)
  also have
    ... =s (%n. 2 * real (n::nat)) +o O(%n. (ln (real n + 1)) ^ 2)
    proof −
      from aux6 have − (%n::nat. 2 * ln (real n + 1)) =o
```

```
      O(%n. (ln (real n + 1)) ^ 2)
      by auto
    thus ?thesis by auto
  qed
 finally show ?thesis.
qed

end
```

37 Transfering asymptotic functions from nats to reals

theory *RealLnSum* = *NatIntLib* + *LnSum1a* + *LnSum5*:

37.1 Sum of one over n

```
lemma bigo-real-inverse-nat-inverse: O(%x. 1 / (abs(x) + 1)) =
    O(%x. 1 / (real(natfloor(abs(x))) + 1))
proof (rule equalityI)
 show O(%x. 1 / (abs x +1)) <= O(%x. 1 / (real (natfloor(abs x)) + 1))
  apply (rule bigo-elt-subset)
  apply (rule bigo-bounded)
  apply (rule allI)
  apply (rule order-less-imp-le)
  apply (rule real-one-over-pos)
  apply arith
  apply (rule allI)
  apply (rule real-one-div-le-anti-mono)
  apply force
  apply (rule add-right-mono)
  apply (rule real-natfloor-le)
  apply force
  done
 next show O(%x. 1 / (real (natfloor (abs x)) + 1)) <=
     O(%x. 1 / (abs x + 1))
  apply (rule bigo-elt-subset)
  apply (rule bigo-bounded-alt)
  apply (rule allI)
  apply (rule order-less-imp-le)
  apply (rule real-one-over-pos)
  apply arith
  apply (rule allI)
```

```
    apply simp
    apply (rule real-fraction-le)
    apply arith
    apply force
    apply (simp add: ring-distrib)
    apply (subgoal-tac abs x + 1 <= 2 * real(natfloor(abs x)) + 2)
    apply assumption
    apply auto
    apply (subgoal-tac abs x <= real(natfloor (abs x)) + 1)
    apply force
    apply (rule real-natfloor-plus-one-ge)
    done
qed

lemma ln-real-approx-ln-nat: (%x. ln(abs(x) + 1)) =o
    (%x. ln(real(natfloor(abs(x))) + 1)) +o O(%x. 1 / (abs(x) + 1))
proof (rule set-minus-imp-plus)
  have (%x. ln (abs(x) + 1)) - (%x. ln (real (natfloor(abs(x))) + 1)) =
      (%x. ln((abs(x) + 1) / (real(natfloor(abs(x))) + 1)))
    apply (simp add: func-diff)
    apply (rule ext)
    apply (subst ln-div)
    apply arith
    apply force
    apply (rule refl)
    done
  also have (%x. ln ((abs(x) + 1) / (real(natfloor(abs(x))) + 1))) =
      (%x. ln (1 + (abs(x) - real(natfloor(abs(x))))) /
        (real(natfloor(abs(x))) + 1)))
    apply (rule ext)
    apply (rule arg-cong)back
    apply (simp add: add-ac mult-ac nonzero-eq-divide-eq
      nonzero-divide-eq-eq right-distrib)
    done
  also have ... =o O(%x. 1 / (real(natfloor(abs(x))) + 1))
    apply (rule bigo-bounded)
    apply (rule allI)
    apply (rule ln-ge-zero)
    apply simp
    apply (subst pos-le-divide-eq)
    apply force
    apply simp
    apply (rule real-natfloor-le)
    apply force
    apply (rule allI)
```

```
    apply (subst add-commute)
    apply (rule order-trans)
    apply (rule ln-add-one-self-le-self)
    apply (subst pos-le-divide-eq)
    apply force
    apply simp
    apply (rule real-natfloor-le)
    apply force
    apply (rule divide-right-mono)
    apply (simp add: compare-rls)
    apply (subst add-commute)
    apply (rule real-natfloor-plus-one-ge)
    apply force
    done
  also have ... = O(%x. 1 / (abs x + 1))
    by (rule bigo-real-inverse-nat-inverse [THEN sym])
  finally show (%x. ln (abs x + 1)) - (%x. ln (real (natfloor (abs x)) + 1))
  : O(%x. 1 / (abs x + 1)).
qed

lemma better-ln-theorem2-real: (%x. ln(abs x + 1)) =o
  ((%x. sumr 0 (natfloor(abs x)+1) (%n. 1 / (real n + 1))) + (%x. gamma)) +o

    O(%x. 1 / (abs x + 1)) (is ?LHS =o ?RHS)
proof -
  have ?LHS =o (%x. ln(real(natfloor(abs x)) + 1)) +o
      O(%x. 1 / (abs x + 1))
    by (rule ln-real-approx-ln-nat)
  also have ... <=
    ((%x. sumr 0 ((natfloor(abs x)) + 1) (%n. 1 / (real n + 1))) +
    (%x. gamma)) +o O(%x. 1 / (real (natfloor(abs x)) + 1))
      + O(%x. 1 / (abs(x) + 1))
    apply (rule set-plus-mono3)
    apply (insert better-ln-theorem2)
    apply (simp only: func-plus)
    apply (erule bigo-compose2)
    done
  also have ... =
    (((%x. sumr 0 ((natfloor(abs x)) + 1) (%n. 1 / (real n + 1))) +
    (%x. gamma))) +o (O(%x. 1 / (real (natfloor(abs x)) + 1)) +
      O(%x. 1 / (abs(x) + 1)))
    by (simp add: set-plus-rearranges add-ac)
  also have O(%x. 1 / (real (natfloor(abs x)) + 1)) +
      O(%x. 1 / (abs(x) + 1)) = O(%x. 1 / (abs(x) + 1))
    apply (subst bigo-real-inverse-nat-inverse [THEN sym])
```

apply *simp*
done
finally show *?thesis*.
qed

37.2 Sum of ln

lemma *identity-three-cor*: (%*n*. *sumr 0* (*n* + *1*) (%*i*. *ln* (*real i* + *1*)))
=*o* ((%*n*. (*real n* + *1*) ∗ *ln* (*real n* + *1*)) +*o* *O*(%*n*. *real n*))
proof −
note *identity-three*
also have ((%*n*. (*real n* + *1*) ∗ *ln* (*real n* + *1*)) − *real*) +*o*
O(%*n*. *ln* (*real n* + *1*)) = (%*n*. (*real n* + *1*) ∗ *ln* (*real n* + *1*)) +*o*
(− *real* +*o* *O*(%*n*. *ln* (*real n* + *1*)))
by (*simp add*: *set-plus-rearranges add-ac diff-minus*)
also have ... <= ((%*n*. (*real* (*n*::*nat*) + *1*) ∗ *ln* (*real n* + *1*)) +*o*
O(%*n*. *real n*))
apply (*rule set-plus-mono*)
apply (*subst bigo-plus-idemp* [*THEN sym*])
apply (*rule set-plus-mono5*)
apply (*rule bigo-minus*)
apply (*rule bigo-refl*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply (*rule allI*)
apply (*rule ln-ge-zero*)
apply *force*
apply (*rule allI*)
apply (*subst add-commute*)
apply (*rule ln-add-one-self-le-self*)
apply *force*
done
finally show *?thesis*.
qed

lemma *natfloor-bigo*: (%*x*. *real* (*natfloor* (*abs x*)) + *1*) : *O*(%*x*. *abs x* + *1*)
apply (*rule bigo-bounded*)
apply *force*
apply *auto*
apply (*rule real-natfloor-le*)
apply *force*
done

lemma *identity-three-cor-real*: (%*x*. *sumr 0* (*natfloor*(*abs x*) + *1*)
(%*i*. *ln* (*real i* + *1*)))

```
  =o (%x. (real (natfloor (abs x) + 1)) *
    ln (abs x + 1)) +o O(%x. (abs x) + 1)
proof −
note bigo-compose2 [OF identity-three-cor, of %x. natfloor(abs x)]
also have (%x. (real (natfloor (abs x)) + 1) *
  ln (real (natfloor (abs x)) + 1)) = (%x. (real (natfloor(abs x)) + 1) *
  ln (abs x + 1)) + (%x. (real (natfloor(abs x)) + 1) *
    (ln (real (natfloor (abs x))+1) − ln(abs x + 1)))
  by (simp add: func-plus diff-minus ring-eq-simps)
also have ... +o O(%x. real(natfloor(abs x))) =
  (%x. (real (natfloor(abs x)) + 1) *
  ln (abs x + 1)) +o ((%x. (real (natfloor(abs x)) + 1) *
    (ln (real (natfloor (abs x))+1) − ln(abs x + 1))) +o
    O(%x. real(natfloor(abs x))))
  by (simp add: set-plus-rearranges)
also have ... <= (%x. (real (natfloor(abs x)) + 1) *
  ln (abs x + 1)) +o O(%x. abs x + 1)
  apply (rule set-plus-mono)
  apply (subst bigo-plus-idemp [THEN sym])
  apply (rule set-plus-mono5)
  apply (subgoal-tac O(%x. abs x + (1::real)) =
    O(%x. abs x + 1) * O(%x. 1))
  apply (erule ssubst)
  apply (subgoal-tac (%x. (real (natfloor (abs x)) + 1) *
        (ln (real (natfloor (abs x)) + 1) − ln (abs x + 1))) =
    (%x. (real (natfloor (abs x)) + 1)) *
     (%x. (ln (real (natfloor (abs x)) + 1) − ln (abs x + 1))))
  apply (erule ssubst)
  apply (rule set-times-intro)
  apply (rule natfloor-bigo)
  apply (subgoal-tac O(%x. 1 / (abs x + 1)) <= O(%x. 1))
  apply (erule subsetD)
  apply (subst func-diff [THEN sym])back
  apply (rule set-plus-imp-minus)
  apply (rule bigo-add-commute-imp)
  apply (rule ln-real-approx-ln-nat)
  apply (rule bigo-elt-subset)
  apply (rule bigo-bounded)
  apply (rule allI)
  apply simp
  apply arith
  apply (rule allI)
  apply (rule real-le-mult-imp-div-pos-le)
  apply arith
  apply simp
```

apply (*simp add*: *func-times*)
apply (*subst bigo-mult8* [*THEN sym*])
apply *arith*
apply (*simp add*: *func-times*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply (*rule allI*)
apply (*rule order-trans*)
apply (*rule real-natfloor-le*)
apply *force*
apply *force*
done
also have (%*x*. (*real* (*natfloor* (*abs x*)) + *1*) ∗ *ln* (*abs x* + *1*)) =
(%*x*. (*real* (*natfloor* (*abs x*) + *1*)) ∗ *ln* (*abs x* + *1*))
by (*rule ext*, *simp*)
finally show *?thesis*.
qed

37.3 Misc bigo calculations

lemma *natfloor-bigo*: (%*x*. *abs x* − *real* (*natfloor* (*abs x*))) : *O*(%*x*. *1*)
apply (*rule bigo-bounded*)
apply *auto*
apply (*rule real-natfloor-le*)
apply *simp*
apply (*subgoal-tac abs x* <= *real*(*natfloor* (*abs x*)) + *1*)
apply *force*
apply (*rule real-natfloor-plus-one-ge*)
done

lemma *x-times-ln-x-real-nat-approx*: (%*x*. (*abs x* + *1*) ∗ *ln* (*abs x* + *1*)) =*o*
(%*x*. (*real* (*natfloor* (*abs x*)) + *1*) ∗ *ln*(*real* (*natfloor* (*abs x*)) + *1*)) +*o*
O(%*x*. *ln*(*abs x* + *1*) + *1*)
proof (*rule set-minus-imp-plus*)
have (%*x*. (*abs x* + *1*) ∗ *ln* (*abs x* + *1*)) −
(%*x*. (*real* (*natfloor* (*abs x*)) + *1*) ∗ *ln* (*real* (*natfloor* (*abs x*)) + *1*)) =
(%*x*. (*abs x* + *1*) − (*real* (*natfloor* (*abs x*)) + *1*)) ∗
(%*x*. *ln*(*abs x* + *1*)) + (%*x*. (*real* (*natfloor* (*abs x*))) + *1*) ∗
(%*x*. *ln*(*abs x* + *1*) − *ln*(*real* (*natfloor* (*abs x*)) + *1*))
apply (*simp add*: *func-plus func-diff func-times*)
apply (*rule ext*)
apply (*simp add*: *ring-eq-simps*)
done
also have ... =*o* (%*x*. *ln* (*abs x* + *1*)) ∗*o* *O*(%*x*. *1*) + *O*(%*x*. *abs x* + *1*) ∗

```
    O(%x. 1 / (abs x + 1))
  apply (rule set-plus-intro)
  apply (simp only: mult-commute)
  apply (rule set-times-intro2)
  apply (simp add: diff-minus [THEN sym])
  apply (rule natfloor-bigo)
  apply (rule set-times-intro)
  apply (rule bigo-bounded)
  apply force
  apply auto
  apply (rule real-natfloor-le)
  apply force
  apply (subst func-diff [THEN sym])back
  apply (rule set-plus-imp-minus)
  apply (rule ln-real-approx-ln-nat)
  done
also have ... <= O(%x. ln(abs x + 1)) + O(%x. 1)
  apply (rule set-plus-mono2)
  apply (rule order-trans)
  apply (rule bigo-mult2)
  apply (simp add: func-times)
  apply (rule order-trans)
  apply (rule bigo-mult)
  apply (simp add: func-times)
  apply (subgoal-tac (%x. (abs x + 1) / (abs x + 1)) = (%x. 1))
  apply (erule ssubst)
  apply simp
  apply (rule ext)
  apply simp
  apply arith
  done
also have ... <= O(%x. ln(abs x + 1) + 1)
  apply (subst bigo-plus-subset6 [THEN sym])
  apply force
  apply force
  apply (simp add: func-plus)
  done
finally show (%x. (abs x + 1) * ln (abs x + 1)) -
    (%x. (real (natfloor (abs x)) + 1) * ln (real (natfloor (abs x)) + 1))
    : O(%x. ln (abs x + 1) + 1).
qed

lemma ln-x-squared-real-nat-approx: (%x. (ln(abs x + 1)) ^2) =o
  (%x. (ln(real (natfloor (abs x)) + 1)) ^2) +o
    O(%x. (ln (abs x + 1)) / (abs x + 1))
```

proof (*rule set-minus-imp-plus*)
  have (%*x*. *ln* (*abs x* + *1*) ^ *2*) − (%*x*. *ln* (*real* (*natfloor* (*abs x*)) + *1*) ^ *2*)
      = (%*x*. *ln*(*abs x* + *1*) − *ln*(*real* (*natfloor* (*abs x*))+*1*)) ∗
        (%*x*. *ln*(*abs x* + *1*) + *ln*(*real* (*natfloor* (*abs x*))+*1*))
    by (*simp add*: *func-diff func-times realpow-two2* [*THEN sym*]
      ring-eq-simps diff-minus)
  also have ... =*o* *O*(%*x*. *1* / (*abs x* + *1*)) ∗ *O*(%*x*. *ln*(*abs x* + *1*))
    apply (*rule set-times-intro*)
    apply (*subst func-diff* [*THEN sym*])**back**
    apply (*rule set-plus-imp-minus*)
    apply (*rule ln-real-approx-ln-nat*)
    apply (*rule-tac c* = *2* **in** *bigo-bounded-alt*)
    apply (*rule allI*)
    apply (*rule nonneg-plus-nonneg*)
    apply (*rule ln-ge-zero*)
    apply *force*
    apply (*rule ln-ge-zero*)
    apply *force*
    apply (*rule allI*)
    apply (*subgoal-tac ln*(*real* (*natfloor* (*abs x*)) + *1*) <=
       ln(*abs x* + *1*))
    apply *force*
    apply (*subst ln-le-cancel-iff*)
    apply *auto*
    apply *arith*
    apply (*rule real-natfloor-le*)
    apply *force*
    done
  also have ... <= *O*(%*x*. *ln*(*abs x* + *1*) / (*abs x* + *1*))
    apply (*rule order-trans*)
    apply (*rule bigo-mult*)
    apply (*simp add*: *func-times*)
    done
  finally show (%*x*. *ln* (*abs x* + *1*) ^ *2*) −
      (%*x*. *ln* (*real* (*natfloor* (*abs x*)) + *1*) ^ *2*)
       : *O*(%*x*. *ln* (*abs x* + *1*) / (*abs x* + *1*)).
qed

lemma *x-times-ln-x-squared-real-nat-approx*:
    (%*x*. (*abs x* + *1*) ∗ (*ln*(*abs x* + *1*))^*2*) =*o*
      (%*x*.(*real*(*natfloor*(*abs x*)) + *1*) ∗
        (*ln*(*real* (*natfloor* (*abs x*)) + *1*))^*2*) +*o*
          O(%*x*. *ln*(*abs x* + *1*) + (*ln*(*abs x* + *1*))^*2*)
proof (*rule set-minus-imp-plus*)
  have (%*x*. (*abs x* + *1*) ∗ *ln* (*abs x* + *1*) ^ *2*) −

```
    (%x. (real (natfloor (abs x)) + 1) *
      ln (real (natfloor (abs x)) + 1) ^ 2) =
   (%x. (abs x + 1)) *
    (%x. ln (abs x + 1) ^2 - ln(real (natfloor (abs x)) + 1) ^2) +
   (%x. (abs x - real (natfloor (abs x)))) *
    (%x. ln(real (natfloor (abs x)) + 1) ^2)
  by (simp add: func-diff func-times func-plus ring-eq-simps
    add-ac diff-minus)
also have ... =o (%x. abs x + 1) *o O(%x. ln (abs x + 1) / (abs x + 1)) +
    O(%x. 1) * O(%x. ln (abs x + 1) ^2)
  apply (rule set-plus-intro)
  apply (rule set-times-intro2)
  apply (subst func-diff [THEN sym])back
  apply (rule set-plus-imp-minus)
  apply (rule ln-x-squared-real-nat-approx)
  apply (rule set-times-intro)
  apply (rule bigo-bounded)
  apply auto
  apply (rule real-natfloor-le)
  apply force
  apply (subgoal-tac abs x <= real (natfloor (abs x)) + 1)
  apply force
  apply (rule real-natfloor-plus-one-ge)
  apply (rule bigo-bounded)
  apply auto
  apply (rule power-mono)
  apply (subst ln-le-cancel-iff)
  apply auto
  apply arith
  apply (rule real-natfloor-le)
  apply force
  done
also have ... <= O(%x. ln(abs x + 1)) + O(%x. ln(abs x + 1) ^2)
  apply (rule set-plus-mono2)
  apply (rule order-trans)
  apply (rule bigo-mult2)
  apply (subst func-times)
  apply (subgoal-tac (%u. (abs u + 1) * (ln (abs u + 1) / (abs u + 1))) =
    (%u. ln (abs u + 1)))
  apply simp
  apply (rule ext)
  apply simp
  apply arith
  apply (rule order-trans)
  apply (rule bigo-mult)
```

```
    apply (simp add: func-times)
    done
  also have ... <= O(%x. ln(abs x + 1) + ln(abs x + 1) ^2)
    apply (subst bigo-plus-subset6 [THEN sym])
    apply force
    apply force
    apply (simp add: func-plus)
    done
  finally show (%x. (abs x + 1) * ln (abs x + 1) ^ 2) -
      (%x. (real (natfloor (abs x)) + 1) *
        ln (real (natfloor (abs x)) + 1) ^ 2)
          =o O(%x. ln (abs x + 1) + ln (abs x + 1) ^ 2).
qed

lemma bigo-fix: f =o O(g) ==> ALL x. 0 <= g x ==> ALL x. 0 <= h x ==>

  ALL x. x < (a::'a::linorder) --> f x = 0 ==> 0 < c  ==>
    ALL x. a <= x --> g x <= c * (h x) ==>
      f =o O(h)
  apply (subst bigo-def)
  apply (simp only: bigo-alt-def)
  apply auto
  apply (rule-tac x = c * ca in exI)
  apply (rule allI)
  apply (case-tac x < a)
  apply (subgoal-tac f x = 0)
  apply simp
  apply (rule nonneg-times-nonneg)
  apply (rule nonneg-times-nonneg)
  apply (erule order-less-imp-le)+
  apply (erule spec)
  apply force
  apply (rule order-trans)
  apply (subgoal-tac abs (f x) <= ca * g x)
  apply assumption
  apply (erule spec)
  apply (subgoal-tac c * ca * h x = ca * (c * h x))
  apply (erule ssubst)
  apply (rule mult-left-mono)
  apply (subgoal-tac a <= x)
  apply blast
  apply (subst linorder-not-less [THEN sym])
  apply assumption
  apply (rule order-less-imp-le)
  apply assumption
```

apply (*simp add*: *mult-ac*)
done

lemma *id-real-nat-approx-bigo*: (%*x*. *abs x*) =*o*
(%*x*. *real* (*natfloor* (*abs x*))) +*o* *O*(%*x*. *1*)
apply (*rule set-minus-imp-plus*)
apply (*subst func-diff*)
apply (*rule bigo-bounded*)
apply *auto*
apply (*rule real-natfloor-le*)
apply *force*
apply (*subgoal-tac abs x* <= *real*(*natfloor* (*abs x*)) + *1*)
apply *force*
apply (*rule real-natfloor-plus-one-ge*)
done

lemma *one-plus-ln-plus-ln-squared-bigo*:
(%*x*. *1* + *ln* (*abs x* + *1*) + *ln*(*abs x* + *1*) ^*2*) =*o*
O(%*x*. *1* + *ln* (*abs x* + *1*) ^*2*)
apply (*rule-tac c* = *2* + *ln* (*exp 1* + *1*) **in** *bigo-bounded-alt*)
apply (*rule allI*)
apply (*subgoal-tac 0* <= *ln*(*abs x* + *1*))
apply (*subgoal-tac 0* <= *ln*(*abs x* + *1*) ^*2*)
apply *arith*
apply (*rule zero-le-power*)
apply (*rule ln-ge-zero*)
apply *force*
apply (*rule ln-ge-zero*)
apply *force*
apply (*rule allI*)
apply (*simp add*: *ring-distrib*)
apply (*subst realpow-two2* [*THEN sym*])
apply (*case-tac exp 1* <= *abs x*)
apply (*subgoal-tac ln* (*abs x* + *1*) <= *ln*(*abs x* + *1*) ∗ *ln*(*abs x* + *1*))
apply (*subgoal-tac 0* <= (*1* + (*ln* (*exp 1* + *1*) +
ln (*exp 1* + *1*) ∗ (*ln* (*abs x* + *1*) ∗ *ln* (*abs x* + *1*)))))
apply *arith*
apply (*rule nonneg-plus-nonneg*)
apply *force*
apply (*rule nonneg-plus-nonneg*)
apply *force*
apply (*rule nonneg-times-nonneg*)
apply *force*
apply *force*
apply (*subgoal-tac ln* (*abs x* + *1*) ∗ *1*<= *ln* (*abs x* + *1*) ∗ *ln* (*abs x* + *1*))

```
  apply simp
  apply (rule mult-left-mono)
  apply (subgoal-tac ln (exp 1) <= ln (abs x + 1))
  apply simp
  apply (subst ln-le-cancel-iff)
  apply auto
  apply arith
  apply (subgoal-tac 0 + ln(abs x + 1) <= ln (abs x + 1) * ln (abs x + 1) +
    (1 + (ln (exp 1 + 1) +
      ln (exp 1 + 1) * (ln (abs x + 1) * ln (abs x + 1)))))
  apply simp
  apply (rule add-mono)
  apply force
  apply (subgoal-tac 0 + ln(abs x + 1) <= 1 + (ln (exp 1 + 1) +
              ln (exp 1 + 1) * (ln (abs x + 1) * ln (abs x + 1))))
  apply simp
  apply (rule add-mono)
  apply force
  apply (subgoal-tac ln (abs x + 1) + 0 <= ln (exp 1 + 1) +
             ln (exp 1 + 1) * (ln (abs x + 1) * ln (abs x + 1)))
  apply simp
  apply (rule add-mono)
  apply (subst ln-le-cancel-iff)
  apply arith
  apply arith
  apply arith
  apply (rule nonneg-times-nonneg)
  apply force
  apply (rule nonneg-times-nonneg)
  apply auto
done

lemma identity-six-real: (%x. sumr 0 (natfloor (abs x) + 1)
    (%i. ln (real i + 1) ^ 2)) =o
   ((%x. (abs x + 1) * ln (abs x + 1) ^ 2) -
     (%x. 2 * (abs x + 1) * ln (abs x + 1)) +
     (%x. 2 * (abs x + 1))) +o O(%x. 1 + ln (abs x + 1) ^ 2)
proof -
  have (%x. sumr 0 (natfloor (abs x) + 1)
    (%i. ln (real i + 1) ^ 2)) =o
      ((%x. (real (natfloor (abs x)) + 1) *
        ln (real (natfloor (abs x)) + 1) ^ 2) -
      (%x. 2) * (%x. (real (natfloor (abs x)) + 1) *
        ln (real (natfloor (abs x)) + 1)) +
      (%x. 2) * (%x. real (natfloor (abs x)))) +o
```

```
  O(%x. ln (real (natfloor (abs x)) + 1) ^ 2)
  apply (insert bigo-compose2 [OF identity-six, of %x. natfloor(abs x)])
  apply (simp add: func-plus func-diff func-times func-minus ring-eq-simps)
  done
also have ... <=
    (((%x. (abs x + 1) * ln(abs x + 1) ^2) +o O(%x. ln(abs x + 1) +
      ln(abs x + 1) ^2)) +
    ((%x. 2) *o (-(%x. (abs x + 1) * ln(abs x + 1)) +o
      O(%x. ln (abs x + 1) + 1))) +
    ((%x. 2) *o ((%x. abs x + 1) +o O(%x. 1)))) + O(%x. ln(abs x + 1) ^2)
  apply (rule set-plus-mono5)
  apply (rule set-plus-intro)
  apply (simp add: diff-minus)
  apply (rule set-plus-intro)
  apply (rule bigo-add-commute-imp)
  apply (rule x-times-ln-x-squared-real-nat-approx)
  apply (subgoal-tac - ((%x. 2) *
     (%x. (real (natfloor (abs x)) + 1) *
          ln (real (natfloor (abs x)) + 1))) =
    (%x. 2) * -(%x. (real (natfloor (abs x)) + 1) *
          ln (real (natfloor (abs x)) + 1)))
  apply (erule ssubst)
  apply (rule set-times-intro2)
  apply (rule bigo-minus2)
  apply (rule bigo-add-commute-imp)
  apply (rule x-times-ln-x-real-nat-approx)
  apply simp
  apply (rule set-times-intro2)
  apply (rule bigo-add-commute-imp)
  apply (rule set-minus-imp-plus)
  apply (subst func-diff)
  apply (rule-tac c = 2 in bigo-bounded-alt)
  apply (rule allI)
  apply (subgoal-tac real( natfloor (abs x)) <= abs x)
  apply arith
  apply (rule real-natfloor-le)
  apply force
  apply (rule allI)
  apply (subgoal-tac abs x <= real(natfloor (abs x)) + 1)
  apply simp
  apply (rule real-natfloor-plus-one-ge)
  apply (rule bigo-elt-subset)
  apply (rule bigo-bounded)
  apply force
  apply (rule allI)
```

apply (*rule power-mono*)
apply (*subst ln-le-cancel-iff*)
apply *force*
apply *arith*
apply *simp*
apply (*rule real-natfloor-le*)
apply *simp*
apply *force*
done
also have ... = ((%*x*. (*abs x* + *1*) * *ln*(*abs x* + *1*) ^ *2*) −
(%*x*. *2* * (*abs x* + *1*) * *ln*(*abs x* + *1*)) +
(%*x*. *2* * (*abs x* + *1*))) +*o* (
O(%*x*. *1*) +
O(%*x*. *ln* (*abs x* + *1*) + *1*) +
O(%*x*. *ln*(*abs x* + *1*) + *ln*(*abs x* + *1*) ^ *2*) +
O(%*x*. *ln*(*abs x* + *1*) ^ *2*))
by (*simp add*: *set-plus-rearranges ring-distrib plus-ac0 mult-ac*
func-times func-minus set-times-plus-distribs diff-minus func-plus)
also have ... <= ((%*x*. (*abs x* + *1*) * *ln*(*abs x* + *1*) ^ *2*) −
(%*x*. *2* * (*abs x* + *1*) * *ln*(*abs x* + *1*)) +
(%*x*. *2* * (*abs x* + *1*))) +*o* *O*(%*x*. *1* + *ln*(*abs x* + *1*) + *ln*(*abs x* + *1*) ^ *2*)
apply (*rule set-plus-mono*)
apply (*rule bigo-useful-intro*)+
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply (*rule allI*)
apply (*subgoal-tac 0* <= *ln*(*abs x* + *1*))
apply (*subgoal-tac 0* <= *ln*(*abs x* + *1*) ^ *2*)
apply *arith*
apply (*rule zero-le-power*)
apply (*rule ln-ge-zero*)
apply *force*
apply (*rule ln-ge-zero*)
apply *force*
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply (*rule allI*)
apply (*subgoal-tac 0* <= *ln*(*abs x* + *1*))
apply *arith*
apply (*rule ln-ge-zero*)
apply *force*
apply (*rule allI*)
apply *force*
apply (*rule bigo-elt-subset*)

```
    apply (rule bigo-bounded)
    apply (rule allI)
    apply (subgoal-tac 0 <= ln(abs x + 1))
    apply (subgoal-tac 0 <= ln(abs x + 1)^2)
    apply arith
    apply (rule zero-le-power)
    apply (rule ln-ge-zero)
    apply force
    apply (rule ln-ge-zero)
    apply force
    apply force
    apply (rule bigo-elt-subset)
    apply (rule bigo-bounded)
    apply force
    apply (rule allI)
    apply (subgoal-tac 0 <= ln(abs x + 1))
    apply arith
    apply (rule ln-ge-zero)
    apply arith
    done
  also have ... <= ((%x. (abs x + 1) * ln (abs x + 1) ^ 2) -
     (%x. 2 * (abs x + 1) * ln (abs x + 1)) +
     (%x. 2 * (abs x + 1))) +o O(%x. 1 + ln(abs x + 1)^2)
    apply (rule set-plus-mono)
    apply (rule bigo-elt-subset)
    apply (rule one-plus-ln-plus-ln-squared-bigo)
    done
  finally show ?thesis.
qed

lemma better-ln-theorem2-real-cor: (%x. ln(abs x + 1)) =o
  (%x. sumr 0 (natfloor(abs x)+1) (%n. 1 / (real n + 1))) +o O(%x. 1)
proof -
  note better-ln-theorem2-real
  also have ((%x. sumr 0 (natfloor (abs x) + 1) (%n. 1 / (real n + 1))) +
       (%x. gamma)) +o O(%x. 1 / (abs x + 1)) =
     (%x. sumr 0 (natfloor (abs x) + 1) (%n. 1 / (real n + 1))) +o
     ((%x. gamma) +o O(%x. 1 / (abs x + 1)))
    by (simp add: set-plus-rearranges)
  also have ... <=
    (%x. sumr 0 (natfloor (abs x) + 1) (%n. 1 / (real n + 1))) +o
     (O(%x. 1) + O(%x. 1))
    apply (rule set-plus-mono)
    apply (rule set-plus-mono5)
    apply (rule bigo-const1)
```

```
    apply (rule bigo-elt-subset)
    apply (rule bigo-bounded)
    apply (rule allI)
    apply (rule real-ge-zero-div-gt-zero)
    apply force
    apply arith
    apply (rule allI)
    apply (rule real-le-mult-imp-div-pos-le)
    apply arith
    apply simp
    done
  also have O(%x. 1) + O(%x. 1) = O(%x. 1)
    by (rule bigo-plus-idemp)
  finally show ?thesis.
qed

lemma identity-three-real: (%x. sumr 0 (natfloor(abs x) + 1)
    (%i. ln (real i + 1))) =o
    ((%x. (abs x + 1) * ln (abs x + 1)) - (%x. abs x + 1)) +o
    O(%x. 1 + ln (abs x + 1))
proof -
  have (%x. sumr 0 (natfloor (abs x) + 1) (%i. ln (real i + 1))) =o
      ((%x. (real (natfloor(abs x)) + 1) *
         (ln(real (natfloor (abs x)) + 1))) + -
       (%x. (real (natfloor(abs x))))) +o
      O(%x. ln(real(natfloor(abs x)) + 1))
    apply (insert bigo-compose2 [OF identity-three, of %x. natfloor(abs x)])
    apply (simp add: func-plus func-minus diff-minus ring-eq-simps)
    done
  also have ... <=
      ((%x. (abs x + 1) * ln (abs x + 1)) +o O(%x. ln(abs x + 1) + 1)) +
      ((-(%x. abs x + 1)) +o O(%x. 1)) +
      O(%x. ln(abs x + 1))
    apply (rule set-plus-mono5)
    apply (rule set-plus-intro)
    apply (rule bigo-add-commute-imp)
    apply (rule x-times-ln-x-real-nat-approx)
    apply (rule bigo-minus2)
    apply (rule bigo-add-commute-imp)
    apply (rule set-minus-imp-plus)
    apply (subst func-diff)
    apply (rule-tac c = 2 in bigo-bounded-alt)
    apply clarsimp
    apply (subgoal-tac real(natfloor (abs x)) <= abs x)
    apply simp
```

```
    apply (rule real-natfloor-le)
    apply force
    apply clarsimp
    apply (subgoal-tac abs x <= real(natfloor (abs x)) + 1)
    apply arith
    apply (rule real-natfloor-plus-one-ge)
    apply (rule bigo-elt-subset)
    apply (rule bigo-bounded)
    apply force
    apply (rule allI)
    apply (subst ln-le-cancel-iff)
    apply auto
    apply arith
    apply (rule real-natfloor-le)
    apply force
    done
  also have ... = ((%x. (abs x + 1) * ln (abs x + 1)) - (%x. abs x + 1)) +o
      (O(%x. ln(abs x + 1) + 1) + (O(%x. ln(abs x + 1)) + O(%x. 1)))
    by (simp add: diff-minus set-plus-rearranges plus-ac0)
  also have ... <= ((%x. (abs x + 1) * ln (abs x + 1)) - (%x. abs x + 1)) +o
      O(%x. ln(abs x + 1) + 1)
    apply (rule set-plus-mono)
    apply (rule bigo-useful-intro)
    apply simp
    apply (subst bigo-plus-subset6 [THEN sym])
    apply force
    apply force
    apply (simp add: func-plus)
    done
  also have (%x. ln(abs x + 1) + 1) = (%x. 1 + ln(abs x + 1))
    by (simp add: add-ac)
  finally show ?thesis.
qed

lemma sum-ln-x-div-x-squared-real-bigo: (%x. sumr 0 (natfloor (abs x) + 1)
      (%i. ln ((abs x + 1) / (real i + 1)) ^ 2)) =o (%x. 2 * (abs x + 1))
  +o O(%x. 1 + ln(abs x + 1) + ln(abs x + 1) ^ 2)
proof -
  have (%x. sumr 0 (natfloor (abs x) + 1)
      (%i. ln ((abs x + 1) / (real i + 1)) ^ 2))
    = (%x. sumr 0 (natfloor (abs x) + 1)
      (%i. ln (abs x + 1) ^ 2 - 2 * ln(abs x + 1) * ln(real i + 1) +
          ln (real i + 1) ^ 2))
    apply (rule ext)
    apply (rule sumr-cong)
```

apply (*subst ln-div*)
apply *arith*
apply *force*
apply (*simp add*: *realpow-two2* [*THEN sym*] *ring-eq-simps*
ring-distrib)
done
also have ... = (%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*i*. *ln*(*abs x* + *1*) ^ *2*)) +
(−(%*x*. *2* ∗ *ln*(*abs x* + *1*))) ∗ (%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*i*. *ln*(*real i* + *1*))) +
(%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*i*. *ln*(*real i* + *1*) ^ *2*))
apply (*simp only*: *func-plus func-minus func-times*)
apply (*rule ext*)
apply (*subst sumr-mult*)
apply (*subst sumr-add*)
apply (*subst sumr-add*)
apply (*rule sumr-cong*)
apply (*simp add*: *diff-minus*)
done
also have ... =*o*
((%*x*. (*abs x* + *1*) ∗ *ln*(*abs x* + *1*) ^ *2*) +*o* *O*(%*x*. *ln*(*abs x* + *1*) ^ *2*)) +
((−(%*x*. *2* ∗ *ln*(*abs x* + *1*))) ∗*o* (((%*x*. (*abs x* + *1*) ∗ *ln* (*abs x* + *1*)) −
(%*x*. *abs x* + *1*)) +*o* *O*(%*x*. *1* + *ln* (*abs x* + *1*)))) +
(((%*x*. (*abs x* + *1*) ∗ *ln* (*abs x* + *1*) ^ *2*) −
(%*x*. *2* ∗ (*abs x* + *1*) ∗ *ln* (*abs x* + *1*)) +
(%*x*. *2* ∗ (*abs x* + *1*))) +*o* *O*(%*x*. *1* + *ln* (*abs x* + *1*) ^ *2*))
apply (*rule set-plus-intro*)
apply (*rule set-plus-intro*)
apply (*rule bigo-add-commute-imp*)
apply (*rule set-minus-imp-plus*)
apply (*subst func-diff*)
apply (*simp only*: *sumr-const*)
apply (*simp only*: *left-diff-distrib* [*THEN sym*])
apply (*rule bigo-bounded*)
apply (*rule allI*)
apply (*rule nonneg-times-nonneg*)
apply (*subgoal-tac real*(*natfloor* (*abs x*)) <= *abs x*)
apply *arith*
apply (*rule real-natfloor-le*)
apply *force*
apply *force*
apply (*rule allI*)
apply (*subgoal-tac ?t* <= *1* ∗ *ln*(*abs x* + *1*) ^ *2*)
apply *simp*

apply (*rule mult-right-mono*)
apply *simp*
apply (*subgoal-tac abs x <= real(natfloor(abs x)) + 1*)
apply *force*
apply (*rule real-natfloor-plus-one-ge*)
apply *force*
apply (*rule set-times-intro2*)
apply (*rule identity-three-real*)
apply (*rule identity-six-real*)
done
also have ... = (%*x*. *2* * (*abs x* + *1*)) +*o*
(*O*(%*x*. *ln* (*1* + *abs x*) ^ *2*) + *O*(%*x*. *1* + *ln* (*1* + *abs x*) ^ *2*) +
(((%*x*. −*2*) * (%*x*. (*ln* (*1* + *abs x*)))) ∗*o*
O(%*x*. *1* + *ln* (*1* + *abs x*))))
apply (*simp add*: *set-plus-rearranges set-times-plus-distribs plus-ac0 func-plus func-times func-diff func-minus ring-eq-simps ring-distrib mult-ac realpow-two2* [*THEN sym*])
apply (*subgoal-tac* (%*x*. − (*ln* (*1* + *abs x*) * *2*)) =
(%*x*. −*2* * *ln* (*1* + *abs x*)))
apply (*erule ssubst*, *rule refl*)
apply (*rule ext*)
apply *simp*
done
also have ... <= (%*x*. *2* * (*abs x* + *1*)) +*o* *O*(%*x*. *1* + *ln*(*1* + *abs x*)
+ *ln*(*1* + *abs x*) ^ *2*)
apply (*rule set-plus-mono*)
apply (*rule bigo-useful-intro*)
apply (*rule bigo-useful-intro*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply (*rule allI*)
apply *clarsimp*
apply (*subgoal-tac 0 <= ln*(*1* + *abs x*))
apply *arith*
apply (*rule ln-ge-zero*)
apply *arith*
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply (*rule allI*)
apply (*rule nonneg-plus-nonneg*)
apply *force*
apply *force*
apply (*rule allI*)
apply *clarsimp*

```
    apply (subgoal-tac ((%x. -2) * (%x. ln (1 + abs x))) *o
      O(%x. 1 + ln (1 + abs x)) =
      (%x. -2) *o ((%x. ln (1 + abs x)) *o O(%x. 1 + ln (1 + abs x))))
    apply (erule ssubst)
    apply (rule order-trans)
    prefer 2
    apply (subgoal-tac (%x. -2) *o ?t <= ?t)
    apply assumption
    apply (rule bigo-const-mult6)
    apply (rule set-times-mono)
    apply (rule order-trans)
    apply (rule bigo-mult2)
    apply (simp add: func-times ring-distrib realpow-two2 [THEN sym])
    apply (rule bigo-elt-subset)
    apply (rule bigo-bounded)
    apply (rule allI)
    apply (rule nonneg-plus-nonneg)
    apply (rule ln-ge-zero)
    apply arith
    apply (rule nonneg-times-nonneg)
    apply (rule ln-ge-zero, arith)+
    apply arith
    apply (simp add: set-times-rearranges times-ac1)
    done
  finally show ?thesis
    by (simp add: add-ac)
qed

lemma power-ln-bigo: (%x. ln(abs x + 1) ^ n) =o
    O(%x. abs x + 1)
  apply (case-tac n = 0)
  apply (erule ssubst)
  apply (rule bigo-bounded)
  apply force
  apply force
  apply (rule-tac c = (real n) powr (real n) in bigo-bounded-alt)
  apply (rule allI)
  apply (rule zero-le-power)
  apply (rule ln-ge-zero)
  apply arith
  apply (rule allI)
  apply (subst powr-realpow2)
  apply (rule ln-ge-zero)
  apply arith
  apply force
```

```
  apply (case-tac x = 0)
  apply simp
  apply (subgoal-tac ln(abs x + 1) ~= 0)
  apply simp
  apply (rule ln-powr-bound2)
  apply arith
  apply force
  apply (subgoal-tac 0 < ln(abs x + 1))
  apply force
  apply (rule ln-gt-zero)
  apply arith
done

lemma sum-ln-x-div-x-squared-real-bigo-cor:
  (%x. sumr 0 (natfloor (abs x) + 1)
       (%i. ln ((abs x + 1) / (real i + 1)) ^2)) =o O(%x. abs x + 1)
proof -
  note sum-ln-x-div-x-squared-real-bigo
  also have (%x. 2 * (abs x + 1)) +o O(%x. 1 + ln (abs x + 1) +
      ln (abs x + 1) ^ 2) <=
    O(%x. (abs x + 1)) + (O(%x. 1) + O(%x. ln(abs x + 1)) +
      O(%x. ln (abs x + 1) ^2))
    apply (rule set-plus-mono5)
    apply (rule-tac c = 2 in bigo-bounded-alt)
    apply (rule allI)
    apply simp
    apply arith
    apply force
    apply (rule order-trans)
    prefer 2
    apply (rule set-plus-mono2)
    apply (rule bigo-plus-subset)
    apply force
    apply (subst func-plus)
    apply (rule order-trans)
    prefer 2
    apply (rule bigo-plus-subset)
    apply (subst func-plus)
    apply simp
    done
  also have ... <= O(%x. abs x + 1)
    apply (rule bigo-useful-intro)
    apply simp
    apply (rule bigo-useful-intro)
    apply (rule bigo-useful-intro)
```

apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply *arith*
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply (*rule allI*)
apply (*rule ln-bound*)
apply *arith*
apply (*rule bigo-elt-subset*)
apply (*rule power-ln-bigo*)
done
finally show *?thesis*.
qed

lemma *telescoping-sum-aux*: $(\sum n = 1..x+1.\ f\ n - f\ (n - 1)) =$
$f\ (x+1) - (f::(nat => 'a::ordered\text{-}ring))\ 0$
apply (*induct x*)
apply *simp*
apply (*subgoal-tac* $\{1..Suc\ n + 1\} = \{1..n + 1\}\ Un\ \{Suc\ n + 1\}$)
apply (*erule ssubst*)**back**
apply (*subst setsum-Un-disjoint*)
apply *force*
apply *force*
apply *force*
apply (*erule ssubst*)
apply (*simp add*: *compare-rls plus-ac0*)
apply *auto*
done

lemma *telescoping-sum*: $1 <= x ==> (\sum n = 1..x.\ f\ n - f\ (n - 1)) =$
$f\ x - (f::(nat => 'a::ordered\text{-}ring))\ 0$
apply (*subgoal-tac* $x = (x - 1) + 1$)
apply (*erule ssubst*)
apply (*rule telescoping-sum-aux*)
apply *simp*
done

lemma *ln-one-plus-one-over-x-bigo*:
$(\%x.\ ln\ (1 + 1\ /\ (real\ x + 1))) =o\ O(\%x.\ 1\ /\ (real\ (x::nat) + 1))$
apply (*rule bigo-bounded*)
apply *force*
apply (*rule allI*)
apply (*rule ln-add-one-self-le-self*)

```
  apply force
done

lemma identity-four-real: (%x. sumr 0 (natfloor(abs x))
    (%i. ln (real i + 1) / (real i + 1))) =o
      (%x. ln(abs x + 1) ^ 2 / 2) +o O(%x. 1)
        (is ?LHS =o ?RHS)
proof -
  have ?LHS =o (%x. ln (real(natfloor(abs x)) + 1) ^ 2 / 2) +o O(%x. 1)
    by (rule bigo-compose2 [OF identity-four])
  also have ... <= ?RHS + O(%x. 1)
    apply (rule set-plus-mono3)
    apply (rule set-minus-imp-plus)
    apply (subgoal-tac (%x. ln (real (natfloor (abs x)) + 1) ^ 2 / 2) -
      (%x. ln (abs x + 1) ^ 2 / 2) = (%x. 1 / 2) *
        ((%x. ln (real (natfloor (abs x)) + 1) ^ 2) -
          (%x. ln (abs x + 1) ^ 2)))
    apply (erule ssubst)
    apply (subgoal-tac O(%x. 1) = (%x. 1 / 2) *o O(%x. 1))
    apply (erule ssubst)
    apply (rule set-times-intro2)
    apply (rule set-plus-imp-minus)
    apply (rule bigo-add-commute-imp)
    apply (rule subsetD)
    prefer 2
    apply (rule ln-x-squared-real-nat-approx)
    apply (rule set-plus-mono)
    apply (rule bigo-elt-subset)
    apply (rule bigo-bounded)
    apply (rule allI)
    apply (rule real-ge-zero-div-gt-zero)
    apply force
    apply arith
    apply (rule allI)
    apply (rule real-le-mult-imp-div-pos-le)
    apply arith
    apply simp
    apply (rule ln-bound)
    apply force
    apply (rule bigo-const-mult5 [THEN sym])
    apply force
    apply (simp add: func-times func-diff)
    apply (rule ext)
    apply simp
    done
```

```
  also have ... <= ?RHS
    by (simp add: set-plus-rearranges)
  finally show ?thesis.
qed

lemma identity-four-real-b: (%x. ∑ i=1..natfloor(abs x).
     ln (real i) / (real i)) =o
       (%x. ln(abs x + 1) ^ 2 / 2) +o O(%x. 1)
  apply (subgoal-tac (%x. ∑ i=1..natfloor(abs x).
     ln (real i) / (real i)) = ?temp)
  apply (erule ssubst)
  apply (rule identity-four-real)
  apply (rule ext)
  apply (subst setsum-sumr4)
  apply (rule sumr-cong)
  apply (subgoal-tac real(y + 1) = real y + 1)
  apply (erule ssubst)
  apply (rule refl)
  apply simp
done

lemma identity-four-real-b-cor: (%x. ∑ i=1..natfloor(abs x)+1.
     ln (real i) / (real i)) =o
       (%x. ln(abs x + 1) ^ 2 / 2) +o O(%x. 1)
proof -
  have (%x. ∑ i=1..natfloor(abs x)+1.
     ln (real i) / (real i)) = (%x. ∑ i=1..natfloor(abs x).
     ln (real i) / (real i)) + (%x. ln (real (natfloor(abs x)+1)) /
       (real (natfloor(abs x)+1)))
    apply (subst func-plus)
    apply (rule ext)
    apply (case-tac natfloor(abs x) = 0)
    apply simp
    apply (rule setsum-range-plus-one-nat)
    apply force
    done
  also have ... =o ((%x. ln(abs x + 1) ^ 2 / 2) +o O(%x. 1)) + O(%x. 1)
    apply (rule set-plus-intro)
    apply (rule identity-four-real-b)
    apply (rule bigo-bounded)
    apply force
    apply (rule allI)
    apply (rule real-le-mult-imp-div-pos-le)
    apply auto
    apply (rule ln-bound)
```

```
    apply auto
    done
  finally show ?thesis by (simp add: set-plus-rearranges)
qed

lemma ln-sum-real2: (%x. ln(abs x + 1)) =o
    (%x. ∑ i=1..natfloor(abs x)+1. 1 / (real i)) +o
      O(%x. 1)
proof −
  have (%x. ln (abs x + 1)) =o
    ((%x. sumr 0 (natfloor (abs x) + 1) (%n. 1 / (real n + 1))) +
     (%x. gamma)) +o O(%x. 1 / (abs x + 1))
    by (rule better-ln-theorem2-real)
  also have (%x. sumr 0 (natfloor (abs x) + 1) (%n. 1 / (real n + 1)))
      = (%x. ∑ i=1..natfloor(abs x)+1. 1 / (real i))
    apply (rule ext)
    apply (subst setsum-sumr4)
    apply (rule sumr-cong)
    apply simp
    done
  also have ((%x. ∑ i = 1..natfloor (abs x) + 1. 1 / real i) +
      (%x. gamma)) +o O(%x. 1 / (abs x + 1)) =
    (%x. ∑ i = 1..natfloor (abs x) + 1. 1 / real i) +o
      ((%x. gamma) +o O(%x. 1 / (abs x + 1)))
    by (simp add: set-plus-rearranges)
  also have ... <= (%x. ∑ i = 1..natfloor (abs x) + 1. 1 / real i) +o
      O(%x. 1)
    apply (rule set-plus-mono)
    apply (rule bigo-plus-absorb2)
    apply (rule bigo-const1)
    apply (rule bigo-elt-subset)
    apply (rule bigo-bounded)
    apply (rule allI)
    apply (rule real-ge-zero-div-gt-zero)
    apply force
    apply arith
    apply (rule allI)
    apply (rule real-le-mult-imp-div-pos-le)
    apply arith
    apply auto
    done
  finally show ?thesis.
qed

lemma bigo-one-subset-bigo-one-plus-ln:
```

$O(\%x.\ 1) <= O(\%x.\ 1 + ln\ (abs\ x + 1))$
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *auto*
done

lemma *bigo-ln-subset-bigo-one-plus-ln*:
$O(\%x.\ ln\ (abs\ x + 1)) <= O(\%x.\ 1 + ln\ (abs\ x + 1))$
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *auto*
done

declare *One-nat-def* [*simp del*]

lemma *one-over-natfloor-one-over-bigo*:
$(\%x.\ 1\ /\ real\ (natfloor\ (abs\ x) + 1)) =o\ O(\%x.\ 1\ /\ (abs\ x + 1))$
apply (*rule-tac c = 2* **in** *bigo-bounded-alt*)
apply *force*
apply (*rule allI*)
apply *simp*
apply (*rule real-fraction-le*)
apply *arith*
apply *force*
apply *simp*
apply (*subgoal-tac abs x <= real(natfloor(abs x)) + 1*)
apply *arith*
apply (*rule real-natfloor-plus-one-ge*)
done

declare *One-nat-def* [*simp add*]

37.4 Not needed?

lemma *divides-div-bigo*: $(\%d.\ real\ x\ /\ real\ (d + (1::nat))) =o$
$(\%d.\ real(x\ div\ (d + 1))) +o\ O(\%d.\ 1)$
apply (*rule set-minus-imp-plus*)
apply (*subst func-diff*)
apply (*rule bigo-bounded*)
apply *clarify*
apply (*rule real-mult-le-imp-le-div-pos*)
apply *force*
apply (*subst real-of-nat-mult* [*THEN sym*])
apply (*rule le-imp-real-of-nat-le*)
apply (*rule nat-div-times-le*)

apply (*rule allI*)
apply (*subst divide-div-aux*)
apply *force*
apply *simp*
apply (*rule real-le-mult-imp-div-pos-le*)
apply *auto*
apply (*rule order-less-imp-le*)
apply *auto*
done

end

38 Chebyshev's functions

theory *Chebyshev1 = Complex + PrimeFactorsList + RealLib*:

consts
lprime :: *nat => real*
Lambda :: *nat => real*
theta :: *nat => real*
psi :: *nat => real*
char-prime :: *nat => real*
pi :: *nat => real*
nmult :: *nat => nat => nat*

defs
lprime-def: *lprime*(*x*) == (*if* (*x* : *prime*)
then (*ln*(*real*(*x*)))
else 0)
theta-def: *theta*(*x*) == *sumr 0* (*x+1*) *lprime*

Lambda-def: *Lambda*(*x*) == *if* (*EX p a*. (*p*:*prime*) & (*p*^*a* = *x*) & *0* < *a*)
then ln(*real*(*THE p*. (*EX a*. (*p*:*prime*) & (*p*^*a* = *x*))))
else 0
psi-def: *psi*(*x*) == *sumr 0* (*x+1*) *Lambda*

char-prime-def: *char-prime*(*x*) == *if* (*x*:*prime*)
then 1
else 0
pi-def: *pi*(*x*) == *real*(*card*({*y*. *y*<=*x* & *y*:*prime*}))

nmult-def: *nmult x y == multiplicity (int x) (int y)*

locale *l* =
fixes *x*::*nat*
and *A*::*nat set*
and *B*::*nat set*
and *C*::(*nat*∗*nat*) *set*
and *D*::(*nat*∗*nat*) *set*
and *E*::(*nat*∗*nat*) *set*
and *F*::*nat set*
and *f*::*nat*∗*nat* => *nat*
and *g*:: *nat* => (*nat*∗*nat*)
and *G*::*nat set*
and *H*::*nat* => *nat set*
and *J*::*nat set*

defines
A-def:*A* == {*y*. (*y*:{*0*..*x*} & (*EX p a*. (*p*:*prime* & *0*<*a* & *p* ^ *a*=*y*)))}
and *B-def*:*B* == {*y*. (*y*:{*0*..*x*} & ~(*EX p a*. (*p*:*prime* & *0*<*a* & *p* ^ *a*=*y*)))}
and *C-def*:*C* == {(*p*,*a*). (*p*:*prime* & *0*<*a* & *p* ^ *a*:{*0*..*x*})}
and *D-def*:*D* == {(*p*,*a*). (*p*:*prime* & *a* = *1* & *p* ^ *a*:{*0*..*x*})}
and *E-def*:*E* == {(*p*,*a*). (*p*:*prime* & *1* < *a* & *p* ^ *a*:{*0*..*x*})}
and *F-def*:*F* == {*p*. *p*:*prime* & *p*:{*0*..*x*}}
and *f-def*:*f y* == (*fst y*) ^ (*snd y*)
and *g-def*:*g z* == (*z*,*1*)
and *G-def*:*G* == {*d*. *0* < *d* & *d dvd x*}
and *H-def*:*H*(*p*) == {*y*. *y*:{*0*..*x*} & (*EX a*. (*0* < *a* & *p* ^ *a* = *y*))}
and *J-def*:*J* == {*p*. *p* : *prime* & *p dvd x*}

38.1 Miscellaneaous

lemma *prime-prop2*: *a*:*prime* ==> *b*:*prime* ==> (*0*<*n*) ==> *a dvd b* ^ *n* ==> *a*=*b*
apply (*frule prime-dvd-power*)
apply (*assumption*)
by (*auto simp add*: *prime-def*)

lemma *dvd-self-exp* [*rule-format*]: *0* < *n* --> (*m*::*nat*) *dvd m* ^ *n*
apply (*induct-tac n*)
by *auto*

lemma *prime-prop*: *a*:*prime* ==> *b*:*prime* ==> *0*<*c* ==> *0*<*d* ==> *a* ^ *c* = *b* ^ *d* ==> *a*=*b*
apply (*subgoal-tac a dvd b* ^ *d*)
apply (*erule prime-prop2*)

prefer *3*
apply (*assumption*)+
apply (*subgoal-tac a dvd a^c*)
apply *simp*
by (*erule dvd-self-exp*)

lemma *power-lemma* [*rule-format*]: (*1*::*nat*) < *a* −−> (*0*::*nat*) < *c* −−> *1* < *a^c*
apply (*induct-tac c*)
apply *force*
apply *auto*
apply (*subgoal-tac 1 < a * a^n*)
apply *force*
apply (*subgoal-tac 1 < a*)
apply (*simp only*: *power-gt1-lemma*)
apply *auto*
done

lemma *prime-prop-lzero*: *a*:*prime* ==> *b*:*prime* ==> *0*<*c* ==> *a^c* = *b^d* ==> *a*=*b*
apply (*case-tac 0=d*)
apply *auto*
apply (*subgoal-tac 1 < a*)
apply (*subgoal-tac 1 < a^c*)
apply (*force*)
apply (*rule power-lemma*)
apply *force*
apply *force*
apply (*subgoal-tac 2 <= a*)
apply *force*
apply (*rule prime-ge-2*)
apply *force*
apply (*rule prime-prop*)
apply *auto*
done

lemma *prime-prop-rzero*: *a*:*prime* ==> *b*:*prime* ==> *0*<*d* ==> *a^c* = *b^d* ==> *a*=*b*
apply (*subgoal-tac b^d = a^c*)
apply (*subgoal-tac b=a*)
apply *force*
apply (*rule prime-prop-lzero*)
apply *auto*
done

lemma *prime-prop2*: *a*:*prime* ==> *b*:*prime* ==> (*0*<*c*) ==> (*0*<*d*) ==>
a ^ *c* = *b* ^ *d* ==> *c*=*d*
apply (*frule prime-prop*)
prefer *4*
apply (*assumption*)+
apply *simp*
by (*auto simp add*:*prime-def*)

lemma *prime-prop-pair*: (*fst x*):*prime* ==> (*fst y*):*prime* ==> *0*<(*snd x*) ==>
0<(*snd y*) ==> (*fst x*) ^ (*snd x*) = (*fst y*) ^ (*snd y*) ==> *x*=*y*
apply (*frule prime-prop2*)
apply *auto*
apply (*subgoal-tac fst x* = *fst y*)
apply (*subgoal-tac snd x* = *snd y*)
apply (*auto simp add*: *prime-prop prime-prop2*)
apply (*simp add*: *prod-eqI*)
done

lemma *real-addition*: (*c*::*real*) ∗ *real*(*Suc x*) = *c* + *c* ∗ *real*(*x*)
by (*simp add*: *real-of-nat-Suc right-distrib*)

lemma *setsum-bound-real*: *finite A* ==> *ALL x*:*A*. (*f*(*x*) <= (*c*::*real*)) ==> *setsum f A* <= *c* ∗ *real*(*card A*)
apply (*induct set*: *Finites*, *auto*)
apply (*simp add*: *real-addition*)
done

lemma *sumr-suc*: *sumr 0 x f* + *f x* = *sumr 0* (*x*+*1*) *f*
apply *auto*
done

lemma *real-power*: ((*a*::*nat*) ^ *b* <= *x*) = ((*real a*) ^ *b* <= *real*(*x*))
apply (*simp only*: *real-of-nat-le-iff* [*THEN sym*])
apply (*simp add*: *realpow-real-of-nat*)
done

lemma *zprime-pos*: *x* : *zprime* ==> *0* < *x*
apply (*auto simp add*: *zprime-def*)
done

lemma *int-nat-inj*: *nat x* = *nat y* ==> *0* < *x* ==> *0* < *y* ==> *x* = *y*
apply (*subgoal-tac int* (*nat x*) = *int* (*nat y*))
apply *auto*
done

lemma *setprod-multiplicity-real*: $0 < n \Longrightarrow real\ n =$
$setprod\ (\%p.\ (real\ p)\ \hat{}\ (multiplicity\ p\ n))$
$\{p.\ p : zprime\ \&\ p\ dvd\ n\}$
apply (*frule n-eq-setprod-multiplicity*)
apply (*subgoal-tac real* $(\prod p{\in}\{p.\ p \in zprime \wedge p\ dvd\ n\}.$
$p\ \hat{}\ multiplicity\ p\ n) = (\prod p{\in}\{p.\ p \in zprime \wedge p\ dvd\ n\}.$
$(real\ p)\ \hat{}\ multiplicity\ p\ n)$)
apply *force*
apply (*subst setprod-real-of-int*)
apply (*rule setprod-cong*)
apply (*rule refl*)
apply (*subst real-of-int-power*)
apply (*rule refl*)
done

lemma *multiplicity-nmult-eq*: *multiplicity x y = nmult* (*nat x*) (*nat y*)
apply (*auto simp add*: *nmult-def*)
apply (*simp add*: *multiplicity-def pfactors-le-1*)
apply (*subgoal-tac x* ~: *zprime*)
apply (*subgoal-tac 0* ~: *zprime*)
apply (*auto simp add*: *not-zprime-multiplicity-eq-0 zprime-def*)
done

38.2 Basic properties

lemma *Lambda-eq-aux*: $[\![p \in prime;\ 0 < a]\!]$
$\Longrightarrow (THE\ pa.\ pa \in prime \wedge (\exists\, aa.\ pa\ \hat{}\ aa = p\ \hat{}\ a)) = p$
apply (*rule the-equality*)
apply *auto*
apply (*erule prime-prop*)
apply *assumption*
prefer *3*
apply *assumption*
apply (*subgoal-tac aa* ~= *0*)
apply *force*
apply (*subgoal-tac p^a* ~= *1*)
apply (*rule notI2*)
apply *assumption*
apply *simp*
apply (*subgoal-tac 1 < p^a*)
apply *force*
apply (*rule power-lemma*)
apply (*force simp add*: *prime-def*)
apply *assumption+*
done

lemma *Lambda-eq*: *p:prime* $\Longrightarrow$ *0 < a* $\Longrightarrow$ *Lambda(p ^ a) = ln(real p)*
apply (*unfold Lambda-def*)
apply *auto*
apply (*subst Lambda-eq-aux*)
apply *auto*
done

lemma *Lambda-eq2*: ~ (*EX p a. p : prime & p ^ a = x & 0 < a*) ==>
Lambda x = 0
apply (*unfold Lambda-def*)
apply *auto*
done

lemma *Lambda-zero*: *Lambda 0 = 0*
apply (*subst Lambda-eq2*)
apply (*auto simp add*: *prime-def*)
done

lemma *Lambda-ge-zero*: *0 <= Lambda x*
apply (*case-tac EX p a. p : prime & p ^ a = x & 0 < a*)
apply (*auto simp add*: *Lambda-eq Lambda-eq2*)
apply (*rule ln-ge-zero*)
apply (*auto simp add*: *prime-def*)
done

lemma *psi-diff1*: *psi (x + 1) - psi x = Lambda (x+1)*
by (*simp add*: *psi-def*)

lemma *psi-diff2*: *1 <= x ==> Lambda x = psi x - psi (x - 1)*
apply (*subgoal-tac x = (x - 1) + 1*)
apply (*erule ssubst*)
apply (*subst psi-diff1* [*THEN sym*])
apply *simp*
apply *simp*
done

lemma *psi-zero*: *psi 0 = 0*
by (*simp add*: *psi-def Lambda-zero*)

lemma *psi-plus-one*: *psi(x + 1) = psi x + Lambda(x + 1)*
apply (*unfold psi-def*)
apply *auto*
done

lemma *psi-def-alt*: *psi x* = (∑ *i*=*1*..*x*. *Lambda i*)
apply (*induct x*)
apply (*simp add*: *psi-zero*)
apply (*subgoal-tac Suc n* = *n* + *1*)
apply (*erule ssubst*)**back**
apply (*subst psi-plus-one*)
apply (*case-tac n* = *0*)
apply *simp*
apply (*subst setsum-range-plus-one-nat*)
apply *auto*
done

lemma *psi-mono*: *x* <= *y* ==> *psi x* <= *psi y*
apply (*unfold psi-def*)
apply (*case-tac x* = *y*)
apply *simp*
apply (*rule sumr-le*)
apply *clarify*
apply (*rule Lambda-ge-zero*)
apply *simp*
done

lemma *psi-ge-zero*: *0* <= *psi x*
apply (*unfold psi-def*)
apply (*rule sumr-ge-zero*)
apply (*rule allI*)
apply (*rule Lambda-ge-zero*)
done

lemma *theta-geq-zero*: *0* <= *theta n*
apply (*unfold theta-def*)
apply (*rule sumr-ge-zero*)
apply (*unfold lprime-def*)
apply *auto*
apply (*rule ln-ge-zero*)
apply (*auto simp add*: *prime-def*)
done

lemma *theta-zero*: *theta 0* = *0*
apply (*unfold theta-def*)
apply *simp*
apply (*unfold lprime-def*)
apply (*simp add*: *prime-def*)
done

lemma *theta-1-is-0*: *theta*(*1*) = *0*
by (*simp add*: *theta-def lprime-def prime-def*)

lemma *big-lemma1*: *pi*(*2*) = *1*
apply (*simp add*: *pi-def*)
apply (*subgoal-tac* {*y*::*nat*. *y* <= (*2*::*nat*) & *y* : *prime*} = {*2*})
apply (*erule ssubst*)
apply (*simp add*: *real-of-nat-inject*)
apply *auto*
apply (*case-tac x=0*)
apply *simp*
apply (*simp add*: *zero-not-prime*)
apply (*case-tac x=1*)
apply (*simp add*: *one-not-prime*)
apply *auto*
apply (*simp add*: *two-is-prime*)
done

38.3 Comparing psi and theta

lemma *theta-setsum-eq1*: *theta*(*x*) = *setsum lprime* {*0*..*x*}
apply (*simp only*: *theta-def*)
apply (*auto simp add*: *setsum-sumr2*)
done

lemma *prime-partition*: {*p*. *p*<*x*} = {*p*. *p*<*x* & *p*:*prime*} *Un* {*p*. *p*<*x* & *p*~:*prime*}
apply *auto*
done

lemma *prime-partition-le*: {*p*. *p*<=*x*} = {*p*. *p*<=*x* & *p*:*prime*} *Un*
{*p*. *p*<=*x* & *p*~:*prime*}
by *auto*

lemma *all-nonprime-set-l*: [| *finite A*; *ALL x*:*A*. *x* ~: *prime* |] ==>
setsum lprime A = *0*
apply (*rule setsum-0'*)
apply (*auto simp add*: *lprime-def*)
done

lemma (**in** *l*) *finite-A*: *finite A*
apply (*auto simp add*: *A-def*)
apply (*rule finite-subset* [*of* {*y*. *y* <= *x* &
(*EX p*. *p* : *prime* & (*EX a*. *0* < *a* & *p* ^ *a* = *y*))} {..*x*}])
apply *auto*
done

lemma (**in** *l*) *finite-B*: *finite B*
apply (*auto simp add*: *B-def*)
apply (*rule finite-subset* [*of* {*y*. *y* <= *x* & (
ALL p. *p* : *prime* −−> (*ALL a*. *a* = *0* | *p* ^ *a* ~= *y*))} {..*x*}])
apply *auto*
done

lemma (**in** *l*) *A-B-disjoint*: *A Int B* = {}
apply (*unfold A-def B-def*)
apply *blast*
done

lemma (**in** *l*) *A-B-all*: *A Un B* = {*y*. *y*<=*x*}
apply (*unfold A-def B-def*)
apply *auto*
done

lemma (**in** *l*) *cor-psi-sum*: *psi*(*x*) = *setsum Lambda A* + *setsum Lambda B*
apply (*unfold psi-def*)
apply (*subst setsum-sumr2* [*THEN sym*])
apply (*subst setsum-Un-disjoint* [*THEN sym*])
apply (*rule finite-A*, *rule finite-B*, *rule A-B-disjoint*)
apply (*subgoal-tac* {*0*..*x*} = *A Un B*)
apply *force*
apply (*unfold A-def B-def*)
apply *blast*
done

lemma (**in** *l*) *B-kernel-for-Lambda*: *y*:*B* ==> *Lambda*(*y*) = *0*
apply (*auto simp add*: *B-def Lambda-def*)
apply (*drule-tac x* = *p* **in** *spec*)
apply (*clarify*)
apply (*drule-tac x* = *a* **in** *spec*)
by *auto*

lemma (**in** *l*) *sum-over-B-of-Lambda-zero*: *setsum Lambda B* = *0*
apply (*rule setsum-0'*)
apply (*rule ballI*)
apply (*rule B-kernel-for-Lambda*)
apply *assumption*
done

lemma (**in** *l*) *inj-on-C*: *inj-on f C*
apply (*rule inj-onI*)

apply (*unfold C-def f-def*)
apply (*subgoal-tac* (*fst xa*):*prime*)
apply (*subgoal-tac* (*fst y*):*prime* 0<(*snd xa*) 0<*snd y*)
apply (*auto simp add*: *prime-prop-pair*)
done

lemma (**in** *l*) *range-of-C-is-A*: *f*‘*C* = *A*
apply (*auto simp add*: *C-def A-def f-def image-def*)
done

lemma (**in** *l*) *finite-C*: *finite C*
apply (*rule finite-imageD*)
apply (*subst range-of-C-is-A*)
apply (*rule finite-A*, *rule inj-on-C*)
done

lemma (**in** *l*) *D-subset-C*: *D* <= *C*
by (*auto simp add*: *D-def C-def*)

lemma (**in** *l*) *E-subset-C*: *E* <= *C*
by (*auto simp add*: *E-def C-def*)

lemma (**in** *l*) *Lambda-reindex-1*: *setsum Lambda* (*f*‘*C*) = *setsum* (*Lambda o f*) *C*
apply (*rule setsum-reindex*)
apply (*simp add*: *finite-C*)
apply (*simp add*: *inj-on-C*)
done

lemma (**in** *l*) *psi-Lambda-eq-over-C*: *psi*(*x*) = *setsum* (*Lambda o f*) *C*
apply (*simp add*: *Lambda-reindex-1* [*THEN sym*])
apply (*subst range-of-C-is-A*)
apply (*subst cor-psi-sum*)
apply (*subst sum-over-B-of-Lambda-zero*)
apply *simp*
done

lemma *psi-alt-def*: *psi*(*x*) = (∑ *u*:{(*p*,*a*). (*p*:*prime* & 0<*a* & *p* ^ *a*:{0..*x*})}.
Lambda((*fst u*) ^ (*snd u*)))
apply (*subst l.psi-Lambda-eq-over-C*)
apply (*unfold o-def*)
apply (*rule refl*)
done

lemma (**in** *l*) *C-eq-D-Un-E*: *C* = *D Un E*
apply (*auto simp add*: *C-def D-def E-def*)

done

lemma (**in** *l*) *D-Int-E-empty*: *D Int E* = {}
by (*auto simp add*: *D-def E-def*)

lemma (**in** *l*) *finite-D*: *finite D*
apply (*rule finite-subset*)
apply (*rule D-subset-C*)
apply (*rule finite-C*)
done

lemma (**in** *l*) *finite-E*: *finite E*
apply (*rule finite-subset*)
apply (*rule E-subset-C*)
apply (*rule finite-C*)
done

lemma (**in** *l*) *setsum-C-D-E*: *setsum* (*Lambda o f*) *C* = *setsum* (*Lambda o f*) *D*
+
setsum (*Lambda o f*) *E*
apply (*subgoal-tac C* = *D Un E*)
apply *simp*
apply (*rule setsum-Un-disjoint*)
apply (*rule finite-D*, *rule finite-E*, *rule D-Int-E-empty*, *rule C-eq-D-Un-E*)
done

lemma (**in** *l*) *psi-Lambda-eq*: *psi*(*x*) = *setsum* (*Lambda o f*) *D* +
setsum (*Lambda o f*) *E*
apply (*subst setsum-C-D-E* [*THEN sym*])
apply (*rule psi-Lambda-eq-over-C*)
done

lemma (**in** *l*) *inj-on-g-F*: *inj-on g F*
apply (*auto simp add*: *inj-on-def F-def g-def*)
done

lemma (**in** *l*) *g-image-F-is-D*: *g*‘*F* = *D*
apply (*auto simp add*: *g-def F-def D-def*)
done

lemma (**in** *l*) *finite-F*: *finite F*
apply (*unfold F-def*)
apply (*subgoal-tac finite* {..*x*})
apply (*rule finite-subset*)
prefer *2*

apply *assumption*
apply *auto*
done

lemma (**in** *l*) *reindex-Lambda-f-g*: *setsum* (*Lambda o f*) *D* =
setsum (*Lambda o f o g*) *F*
apply (*insert g-image-F-is-D* [*THEN sym*] *inj-on-g-F finite-F*)
apply (*simp add*: *setsum-reindex*)
done

lemma *aux1* [*rule-format*]: *1* < (*a*::*nat*) −−> *0* < *b* −−> *Suc 0* < *a*^*b*
apply (*induct-tac b*)
apply *force*
apply *clarsimp*
apply (*case-tac n* = *0*)
apply *auto*
apply (*rule one-less-mult*)
apply *auto*
done

lemma *aux2* [*rule-format*]: *1* < (*a*::*nat*) & *1* < *b* −−> *a* < *a*^*b*
apply (*induct-tac b*)
apply *force*
apply *clarify*
apply *simp*
apply (*rule aux1*)
apply *auto*
done

lemma *aux3*: *p*:*prime* ==> *1* < *a* ==> ~ (*p*^*a* : *prime*)
apply (*unfold prime-def*)
apply (*clarsimp*)
apply (*rule-tac x* = *p* **in** *exI*)
apply (*rule conjI*)
apply (*rule dvd-self-exp*)
apply *force*
apply (*rule conjI*)
apply *force*
apply (*subgoal-tac p* < *p*^*a*)
apply *force*
apply (*rule aux2*)
by *auto*

lemma *prime-power-must-be-one*: *p*:*prime* ==> *p*^*a*:*prime* ==> *a*=*1*
apply *auto*

apply (*case-tac a=0*)
apply *simp*
apply (*simp add*: *prime-def*)
apply (*case-tac a=1*)
apply (*simp-all add*: *aux3*)
done

lemma (**in** *l*) *F-prop*: *p*:*F* −−> (*Lambda*(*f*(*g*(*p*)))) = *ln*(*real*(*p*))
apply (*unfold F-def f-def g-def*)
apply *clarify*
apply *simp*
apply (*unfold Lambda-def*)
apply *auto*
apply (*subgoal-tac* (*THE pa*::*nat*. *pa* : *prime* &
(*EX aa*::*nat*. *pa* ^ *aa* = *p* ^ *a*)) = *p*^*a*)
apply (*erule ssubst*)
apply (*simp*)
apply (*rule the-equality*)
apply (*auto simp add*: *if-splits*)
apply (*rule-tac x* = *1* **in** *exI*)
apply *simp*
apply (*subgoal-tac a=1*)
apply *simp*
apply (*subgoal-tac aa=1*)
apply *simp*
apply (*subgoal-tac pa*^*aa* : *prime*)
prefer *2*
apply *simp*
apply (*simp only*: *prime-power-must-be-one*)
apply (*simp only*: *prime-power-must-be-one*)
apply (*drule-tac x* = *p* **in** *spec*)
apply *simp*
apply (*drule-tac x* = *1* **in** *spec*)
apply *auto*
done

lemma (**in** *l*) *sum-over-F*: *setsum* (*Lambda o f o g*) *F* = *setsum* (*ln o real*) *F*
apply (*rule setsum-cong*)
apply (*auto simp add*: *F-prop*)
done

lemma (**in** *l*) *sum-over-F2-lemma1*: *setsum lprime* {*0*..*x*} =
setsum lprime ({*p*. *p*<=*x* & *p*:*prime*} *Un* {*p*. *p*<=*x* & *p*~:*prime*})
apply (*subgoal-tac* {*0*..*x*} = {*p*. *p*<=*x*})
apply (*erule ssubst*)

apply (*subgoal-tac* {*p*. *p*<=*x*} =
({*p*::*nat*. *p* <= *x* & *p* : *prime*} *Un* {*p*::*nat*. *p* <= *x* & *p* ~: *prime*}))
apply (*erule ssubst*)
apply *simp*
apply (*auto simp add*: *prime-partition*)
done

lemma (**in** *l*) *sum-over-F2-lemma2*:
setsum lprime ({*p*. *p*<=*x* & *p*:*prime*} *Un* {*p*. *p*<=*x* & *p*~:*prime*}) =
setsum lprime {*p*. *p*<=*x* & *p*:*prime*} + *setsum lprime* {*p*. *p*<=*x* & *p*~:*prime*}
apply (*rule setsum-Un-disjoint*)
apply *auto*
apply (*rule finite-subset-AtMost-nat*)
apply *force*
apply (*rule finite-subset-AtMost-nat*)
apply *force*
done

lemma (**in** *l*) *l-set-of-primes*: *ALL x*:*P*. *x*:*prime* ==>
setsum lprime P = *setsum* (*ln o real*) *P*
apply (*rule setsum-cong*)
apply (*auto simp add*: *lprime-def*)
done

lemma (**in** *l*) *sum-over-F2*: *setsum* (*ln o real*) *F* = *theta*(*x*)
apply (*unfold F-def theta-def*)
apply (*subst setsum-sumr2* [*THEN sym*])
apply (*subst l-set-of-primes* [*THEN sym*])
apply *force*
apply (*subgoal-tac setsum lprime* {*0..x*} =
setsum lprime ({*p*. *p*:*prime* & *p*<=*x*} *Un* {*p*. *p*~:*prime* & *p*<=*x*}))
apply (*erule ssubst*)
apply (*subst setsum-Un-disjoint*)
apply (*rule finite-subset-AtMost-nat*)
apply *auto*
apply (*rule finite-subset-AtMost-nat*)
apply *auto*
apply (*subst all-nonprime-set-l*)
apply (*rule finite-subset-AtMost-nat*)
apply *auto*
apply (*subgoal-tac* {*0..x*} =
{*p*. *p*:*prime* & *p*<=*x*} *Un* {*p*. *p*~:*prime* & *p*<=*x*})
apply *auto*
done

lemma (**in** *l*) *psi-theta-sum*: *psi*(*x*) = *theta*(*x*) + *setsum* (*Lambda o f*) *E*
apply (*subst sum-over-F2* [*THEN sym*])
apply (*subst sum-over-F* [*THEN sym*])
apply (*subst reindex-Lambda-f-g* [*THEN sym*])
apply (*rule psi-Lambda-eq*)
done

lemma *exponent-eq-0-iff*: *2* <= *p* ==> *Suc 0* = *p* ^ *a* ==> *a* = *0*
apply (*case-tac a*)
apply (*auto simp add*: *power-Suc*)
done

lemma (**in** *l*) *Lambda-positive*: *ALL x*:*E*. *0* <= *Lambda*(*f*(*x*))
apply (*auto simp add*: *E-def Lambda-def f-def*)
apply (*subgoal-tac 0* = *ln 1*)
apply (*erule ssubst* [*of* (*0*::*real*) *ln* (*1*::*real*) -])
apply (*subgoal-tac 0* < *real*((*THE p*::*nat*. *p* : *prime* &
(*EX aa*::*nat*. *p* ^ *aa* = *a* ^ *b*))))
apply (*simp only*: *ln-le-cancel-iff*)
apply (*subgoal-tac* (*1* :: *real*) = *real*(*1*::*nat*))
apply (*erule ssubst* [*of* (*1*::*real*) *real* (*1*::*nat*) -])
apply (*simp only*: *real-of-nat-le-iff*)
apply (*auto*)
apply (*rule Suc-leI*)
apply *simp*
apply (*rule theI2*)
apply *auto*
prefer *2*
apply (*simp add*: *prime-pos*)
apply (*case-tac aaa* = *0*)
prefer *2*
apply (*subgoal-tac 0* < *aaa*)
prefer *2*
apply *simp*
apply (*subgoal-tac p* ^ *aa* = *pa* ^ *aaa*)
apply (*simp only*: *prime-prop*)
apply *simp*
apply *auto*
apply (*subgoal-tac Suc 0* = *p* ^ *aa*)
prefer *2*
apply *simp*
apply (*subgoal-tac 2* <= *p*)
apply (*subgoal-tac 0* = *aa*)
prefer *2*
apply (*simp add*: *exponent-eq-0-iff*)

prefer *2*
apply (*simp add*: *prime-ge-2*)
apply (*simp only*: *order-less-imp-not-eq* [*of 0 aa*])
done

lemma *real-power-ln*: *1 < a ==> 0<x ==> ((a::nat) ^ b <= x) =*
(real(b) <= ln(real x)/ln(real a))
apply (*simp only*: *real-power*)
apply (*subgoal-tac 0 < real a ^ b*)
apply (*subgoal-tac 0 < real(x)*)
apply (*simp only*: *ln-le-cancel-iff* [*THEN sym*])
apply (*simp add*: *ln-realpow*)
apply (*subgoal-tac 0 < ln (real a)*)
apply (*simp add*: *pos-le-divide-eq*)
apply (*subgoal-tac 1 < real a*)
apply (*simp add*: *ln-gt-zero*)
apply *force*
apply *force*
apply (*subgoal-tac 0 < a*)
apply (*subgoal-tac 0 < a^b*)
apply (*simp add*: *real-of-nat-less-iff*)
apply (*simp add*: *zero-less-power*)
apply *force*
done

lemma (**in** *l*) *extent-of-E2*: *E <= {1..nat(floor(real(x) powr (1/2)))} <*>*
{1..nat(floor(ln(real x)/ln(2)))}
apply (*auto simp add*: *E-def*)
apply (*subgoal-tac 2 <=a*)
apply *force*
apply (*simp add*: *prime-ge-2*)
apply (*subgoal-tac a <= natfloor(real x powr (1/2))*)
apply (*simp add*: *natfloor-def*)
apply (*subgoal-tac a^2 <= x*)
apply (*subgoal-tac (real a) powr (real (2::nat)) <= real x*)
apply (*subgoal-tac ((real a) powr (real (2::nat))) powr (1/2) <= (real x) powr*
(1/2))
apply (*simp add*: *powr-powr*)
apply *auto*
apply (*subgoal-tac real a <= real x powr (1/2)*)
apply (*rule real-le-natfloor*)
apply *auto*
apply (*subgoal-tac real a powr 1 = real a*)

apply *force*
apply (*subst powr-one-gt-zero-iff*)
apply (*subgoal-tac 2 <= a*)
apply *arith*
apply (*simp add*: *prime-ge-2*)
apply (*rule power-mono2*)
apply *force+*
apply (*subgoal-tac* (*2*::*real*) = *real* (*2*::*nat*))
apply (*erule ssubst*)
apply (*subst powr-realpow*)
apply (*subgoal-tac 2 <= a*)
apply *arith*
apply (*simp add*: *prime-ge-2*)
apply (*subgoal-tac real a ^ 2 = real* (*a^2*))
apply *force*
apply (*rule realpow-real-of-nat*)
apply *force+*
apply (*subgoal-tac a^2 <= a^b*)
apply *force*
apply (*rule power-increasing*)
apply *force*
apply (*frule prime-ge-2*)
apply *arith*

apply (*subgoal-tac 2^b <= x*)
apply (*subgoal-tac 0 < x*)
apply (*subgoal-tac 1 < a*)
apply (*simp only*: *real-power-ln*)
apply (*subgoal-tac real*(*2*::*nat*) = *2*)
apply (*erule ssubst*)
apply (*auto simp add*: *floor-bound*)
apply (*subgoal-tac 2 <= a*)
apply *force*
apply (*simp add*: *prime-ge-2*)
apply (*subgoal-tac 0 < a*)
apply (*subgoal-tac 0 < a^b*)
apply (*simp only*: *less-le-trans* [*of 0 a^b x*])
apply (*simp add*: *zero-less-power*)
apply (*subgoal-tac 2 <= a*)
apply *force*
apply (*simp add*: *prime-ge-2*)
apply (*subgoal-tac 2 <= a*)
apply (*subgoal-tac 2^b <= a^b*)
apply *force*
apply (*auto simp add*: *power-mono prime-ge-2*)

done

lemma (**in** *l*) *card-E2-lemma*: *card E* <=
card ({*1..nat*(*floor*(*real*(*x*) *powr* (*1*/*2*)))} <*> {*1..nat*(*floor*(*ln*(*real x*)/*ln*(*2*)))})
apply (*rule card-mono*)
prefer *2*
apply (*rule extent-of-E2*)
apply *simp*
done

lemma (**in** *l*) *card-E2-lemma2*:
card ({*1..nat*(*floor*(*real*(*x*) *powr* (*1*/*2*)))} <*> {*1..nat*(*floor*(*ln*(*real x*)/*ln*(*2*)))})
=
card {*1..nat*(*floor*(*real*(*x*) *powr* (*1*/*2*)))} * *card* {*1..nat*(*floor*(*ln*(*real x*)/*ln*(*2*)))}
apply *simp*
done

lemma (**in** *l*) *card-E2-lemma3*:
card ({*1..nat*(*floor*(*real*(*x*) *powr* (*1*/*2*)))} <*> {*1..nat*(*floor*(*ln*(*real x*)/*ln*(*2*)))})
<=
card {*1..nat*(*floor*(*real*(*x*) *powr* (*1*/*2*)))} * *card* {*1..nat*(*floor*(*ln*(*real x*)/*ln*(*2*)))}
apply (*auto simp add*: *card-E2-lemma2*)
done

lemma (**in** *l*) *card-E2*:
card E <= *card* {*1..nat*(*floor*(*real*(*x*) *powr* (*1*/*2*)))} * *card* {*1..nat*(*floor*(*ln*(*real x*)/*ln*(*2*)))}
apply (*insert card-E2-lemma3*)
apply (*insert card-E2-lemma*)
apply (*erule le-trans*)
apply *assumption*
done

lemma *card-E2-real-lemma4*: *real*(*floor*(*ln*(*real x*)/*ln*(*2*))) <= *ln*(*real x*)/*ln*(*2*)
apply (*simp add*: *real-of-int-floor-le*)

done

lemma (**in** *l*) *real-card-E2*: *real*(*card E*) <=
real(*card* {*1..nat*(*floor*(*real*(*x*) *powr* (*1/2*)))} * *card* {*1..nat*(*floor*(*ln*(*real x*)/*ln*(*2*)))})
apply (*simp only*: *real-of-nat-le-iff*)
apply (*rule card-E2*)
done

lemma (**in** *l*) *card-E2-real-lemma6*: *0* < *x* ==>
real(*card* {*1..nat*(*floor*(*ln*(*real x*)/*ln*(*2*)))}) <= *ln*(*real x*)/*ln*(*2*)
apply *simp*
apply (*subgoal-tac 0* <= *ln*(*real x*) / *ln 2*)
apply (*rule real-nat-floor*)
apply *auto*
done

lemma (**in** *l*) *card-E2-real*: *0* < *x* ==> *real*(*card E*) <=
real(*nat*(*floor*(*real*(*x*) *powr* (*1/2*)))) * *ln*(*real x*)/*ln*(*2*)
proof−
assume *pos*: *0* < *x*
have *step1*: *real*(*card E*) <=
real(*card* {*1..nat*(*floor*(*real*(*x*) *powr* (*1/2*)))} * *card* {*1..nat*(*floor*(*ln*(*real*
x)/*ln*(*2*)))})
by (*rule real-card-E2*)
also have *step2*: ... =
real(*card* {*1..nat*(*floor*(*real*(*x*) *powr* (*1/2*)))}) * *real*(*card* {*1..nat*(*floor*(*ln*(*real*
x)/*ln*(*2*)))})
by (*rule real-of-nat-mult*)
also have *step3*: ... <= *real*(*nat*(*floor*(*real*(*x*) *powr* (*1/2*)))) * *ln*(*real x*)/*ln*(*2*)
apply (*simp add*: *card-E2-real-lemma6*)
apply (*subgoal-tac real* (*nat*(*floor*(*real*(*x*) *powr* (*1/2*)))) * *ln* (*real x*) / *ln 2*
=
real (*nat*(*floor*(*real*(*x*) *powr* (*1/2*)))) * (*ln* (*real x*) / *ln 2*))
apply (*erule ssubst*)
apply (*rule mult-left-mono*)
apply (*subgoal-tac 0*<= *ln*(*real x*)/*ln 2*)
apply (*rule real-nat-floor*)
apply *force*
apply (*subgoal-tac 1*<= *real x*)

```
    apply (drule ln-ge-zero)
    apply (subgoal-tac 0 < ln 2)
    apply (simp add: real-ge-zero-div-gt-zero)
    apply auto
    apply (subgoal-tac 1<=x)
    apply (auto simp add: pos)
    apply (simp add: pos Suc-leI)
    done
  finally show ?thesis.
qed

lemma (in l) E-bound-with-f: y:E --> f(y) <= x
 apply (auto simp add: E-def)
 apply (auto simp add: f-def)
done

lemma Lambda-prop: ALL x. (0 < x --> Lambda(x) <= ln(real(x)))
 apply (auto simp add: Lambda-def)
 apply (rule theI2)
 apply auto
 apply (rule prime-prop-rzero)
 apply auto
 apply (case-tac aa=0)
 apply auto
 apply (subgoal-tac Suc 0 < p^1)
 apply (subgoal-tac p^1 <= p^a)
 apply force
 apply (rule power-increasing)
 apply force
 apply force
 apply (subgoal-tac 2 <= p^1)
 apply force
 apply (subgoal-tac 2 <= p)
 apply (subgoal-tac p = p^1)
 apply force
 apply (rule power-one-right [THEN sym])
 apply (simp add: prime-ge-2)
 apply (subgoal-tac x = p)
 apply (subgoal-tac real(x) = real(p))
 apply auto
 apply (subgoal-tac p^1 <= p^a)
 apply force
 apply (rule power-increasing)
 apply force
 apply force
```

apply (*rule prime-prop-rzero*)
apply *auto*
done

lemma (**in** *l*) *E-bound*: *0 < x ==> y:E ==> (Lambda o f)(y) <= ln(real(x))*
apply (*auto simp add*: *E-def*)
apply (*auto simp add*: *f-def*)
apply (*subgoal-tac 0 < xa^y*)
apply (*subgoal-tac Lambda(xa^y) <= ln(real(xa^y))*)
apply (*subgoal-tac ln(real(xa^y)) <= ln(real(x))*)
apply *force*
apply (*simp only*: *ln-le-cancel-iff*)
apply (*simp add*: *Lambda-prop*)
apply (*subgoal-tac xa^1 <= xa^y*)
apply (*subgoal-tac 0 < xa*)
apply *force*
apply (*subgoal-tac 2 <= xa*)
apply *force*
apply (*simp add*: *prime-ge-2*)
apply (*subgoal-tac 1 <= y*)
apply (*rule power-increasing*)
apply *auto*
apply (*subgoal-tac 2 <= xa*)
apply *force*
apply (*auto simp add*: *prime-ge-2*)
done

lemma (**in** *l*) *sum-over-E2-of-Lambda-o-f*: *0 < x ==>*
*setsum (Lambda o f) E <= real(nat(floor(real(x) powr (1/2)))) * ln(real(x)) * ln(real(x)) / ln 2*
apply (*subgoal-tac setsum (Lambda o f) E <= real(card E) * ln(real(x))*)
apply (*subgoal-tac real(card E) <= real(nat(floor(real(x) powr (1/2)))) * ln(real x)/ln(2)*)
apply (*subgoal-tac real(card E) * ln(real(x)) <=*
*real(nat(floor(real(x) powr (1/2)))) * ln(real x)/ln(2) * ln(real(x))*)
apply (*subgoal-tac setsum (Lambda o f) E <=*
*real(nat(floor(real(x) powr (1/2)))) * ln(real x)/ln(2) * ln(real(x))*)
apply *force*
apply *force*
apply (*subgoal-tac 0 <= ln(real(x))*)
apply (*rule mult-right-mono*)
apply *force*
apply (*subgoal-tac 1 <= x*)

apply (*simp add*: *ln-ge-zero*)
apply *force*
apply (*subgoal-tac 1 <= x*)
apply (*simp add*: *ln-ge-zero*)
apply *force*
apply (*simp add*: *card-E2-real*)
apply (*subgoal-tac real* (*card E*) * *ln* (*real x*) =
ln (*real x*) * *real* (*card E*))
apply (*erule ssubst*)
apply (*rule setsum-bound-real*)
apply (*simp add*: *finite-E*)
apply (*rule ballI*)
apply (*rule E-bound*)
apply *auto*
done

lemma (**in** *l*) *psi-theta-bound-for-real-aux*: *0 < x ==> psi*(*x*) <=
theta(*x*) + *real*(*nat*(*floor*(*real*(*x*) *powr* (*1/2*)))) * *ln*(*real*(*x*)) * *ln*(*real*(*x*)) / *ln* *2*
apply (*subgoal-tac psi*(*x*) = *theta*(*x*) + *setsum* (*Lambda o f*) *E*)
apply (*subgoal-tac setsum* (*Lambda o f*) *E* <=
real(*nat*(*floor*(*real*(*x*) *powr* (*1/2*)))) * *ln*(*real*(*x*)) * *ln*(*real*(*x*)) / *ln* *2*)
apply *auto*
apply (*auto simp add*: *psi-theta-sum sum-over-E2-of-Lambda-o-f*)
done

lemma *psi-theta-bound-for-real*: *0 < x ==> psi*(*x*) <=
theta(*x*) + *real*(*x*) *powr* (*1/2*) * *ln*(*real*(*x*)) * *ln*(*real*(*x*)) / *ln* *2*
apply (*rule order-trans*)
apply (*erule l.psi-theta-bound-for-real-aux*)
apply (*rule add-left-mono*)
apply (*rule divide-right-mono*)
apply (*rule mult-right-mono*)+
apply (*subgoal-tac nat*(*floor* (*?t*)) = *natfloor*(*?t*))
apply (*erule ssubst*)
apply (*rule real-natfloor-le*)
apply (*rule order-less-imp-le*)
apply (*rule powr-gt-zero*)
apply (*simp add*: *natfloor-def*)
apply *auto*
done

38.4 Comparing pi and theta

lemma *pi-set-zero*: *finite A ==> ALL x:A. x~:prime ==>*
setsum char-prime A = 0
apply (*rule setsum-0'*)
apply (*unfold char-prime-def*)
apply *auto*
done

lemma *setsum-prime-split*: *setsum f {p. p<=x} =*
setsum f {p. p<=x & p:prime} + setsum f {p. p<=x & p~:prime}
apply (*simp add*: *prime-partition-le*)
apply (*rule setsum-Un-disjoint*)
apply *auto*
apply (*auto intro*: *finite-subset-AtMost-nat*)
done

lemma *setsum-char-prime-zero*: *setsum char-prime {p. p<=x & p~:prime} = 0*
apply (*rule pi-set-zero*)
by (*auto intro*: *finite-subset-AtMost-nat*)

lemma *pi-setsum-equiv*: *pi(x) = setsum char-prime {p. p<=x}*
apply (*auto simp add*: *pi-def char-prime-def*)
apply (*simp add*: *setsum-prime-split*)
apply (*simp add*: *setsum-char-prime-zero*)
apply (*subgoal-tac setsum char-prime {p::nat. p <= x & p : prime} =*
setsum (%x. 1) {p::nat. p <= x & p : prime})
apply (*erule ssubst*)
apply (*subst real-card-eq-setsum*)
apply (*force intro*: *finite-subset-AtMost-nat*)
apply *simp*
apply (*rule setsum-cong* [*of* {*p::nat. p <= x & p : prime*}
{*p::nat. p <= x & p : prime*} *char-prime %x. 1*])
apply (*auto simp add*: *char-prime-def*)
done

lemma *pi-sumr-equiv*: *pi(x) = sumr 0 (x+1) char-prime*
apply (*simp only*: *pi-setsum-equiv*)
apply (*subgoal-tac {p::nat. p <= x} = {0..x}*)
apply (*erule ssubst*)
apply (*auto simp add*: *setsum-sumr2*)
done

lemma *pi-Suc*: *pi(Suc x) = char-prime(Suc x) + pi(x)*
apply (*case-tac (Suc x):prime*)

apply (*simp add*: *pi-def char-prime-def*)
apply (*subgoal-tac* {*y*::*nat*. *y* <= *Suc x* & *y* : *prime*} =
{*y*::*nat*. *y* <= *x* & *y* : *prime*} *Un* {*Suc x*})
apply (*erule ssubst*)
apply (*subst card-Un-disjoint*)
apply (*force intro*: *finite-subset-AtMost-nat*)
apply *simp*
apply *force*
apply *simp*
apply *auto*
apply (*simp add*: *pi-def char-prime-def*)
apply (*rule arg-cong*)**back**
apply *auto*
apply (*case-tac xa* = *Suc x*)
apply *auto*
done

lemma *char-prime-def-eq*: *1* <= *n* ==>
(*theta*(*n*+*1*) - *theta*(*n*)) / *ln*(*real*(*n*+*1*)) = *char-prime*(*n*+*1*)
apply (*auto simp add*: *theta-def*)
apply (*case-tac Suc n*:*prime*)
apply (*auto simp add*: *lprime-def char-prime-def real-divide-def*)
done

lemma *nat-int-inj-on*: *inj-on* (%*x*. *nat x*) {*p*. *p*:*zprime* & (*p dvd x*)}
apply (*rule inj-onI*)
apply *auto*
apply (*rule int-nat-inj*)
apply (*auto simp add*: *zprime-pos*)
done

lemma *int-nat-inj-on*: *inj-on int* {*p*. *p*:*prime* & *p dvd x*}
apply (*rule inj-onI*)
apply *auto*
done

38.5 Expressing ln in terms of Lambda

lemma *ln-product-sum*: *finite A* ==> *ALL x*:*A*. (*0* < *f*(*x*)) ==>
ln (*setprod f A*) = *setsum* (*ln o f*) *A*
apply (*induct set*: *Finites*)
apply (*simp add*: *setprod-def setsum-def*)
apply (*simp add*: *setprod-insert setsum-insert*)
apply (*subst ln-mult*)
apply (*auto simp add*: *setprod-pos o-def*)

done

lemma *multiplicity-eq-1*: *1 < n ==> ln(real n) =*
ln(setprod (%p. (real p) ^ (multiplicity p n)) {p. 0 < p & p : zprime &
p dvd n})
apply (*subst ln-inj-iff*)
apply *force*
apply (*rule setprod-pos*)
apply *auto*
apply (*subgoal-tac {p. 0 < p ∧ p ∈ zprime ∧ p dvd n} =*
{p. p ∈ zprime ∧ p dvd n})
apply (*erule ssubst*)
apply (*rule setprod-multiplicity-real*)
apply (*auto simp add*: *zprime-def*)
done

lemma *divisor-set-fact*: *1 < n ==> {p::int. 0 < p & p : zprime & p dvd n} <=*

{−n..n}
apply *auto*
apply (*subgoal-tac (−n <= x) = (−x <= n)*)
apply (*erule ssubst*)
apply (*simp add*: *zdvd-imp-le*)
apply *auto*
done

lemma *multiplicity-eq-2*: *1 < n ==> ln(real n) =*
setsum (%p. ln ((real p) ^ (multiplicity p n))) {p. 0 < p & p : zprime &
p dvd n}
apply (*subst multiplicity-eq-1*)
apply (*assumption*)
apply (*subst ln-product-sum*)
apply (*force intro*: *finite-subset-GreaterThan0AtMost-int zdvd-imp-le*)
apply *force*
apply (*simp add*: *o-def*)
done

lemma *multiplicity-eq-3*: *1 < n ==> ln(real n) =*
*setsum (%p. real(multiplicity p n) * ln(real p)) {p. 0 < p & p : zprime &*
p dvd n}
apply (*subst multiplicity-eq-2*, *assumption*)
apply (*rule setsum-cong*)
apply (*rule refl*)
apply (*subst ln-realpow*)
apply *force*

apply (*rule refl*)
done

lemma (**in** *l*) *finite-G*: *0 < x ==> finite G*
apply (*unfold G-def*)
apply (*rule finite-subset-GreaterThan0AtMost-nat*)
apply (*auto intro*: *dvd-imp-le*)
done

lemma (**in** *l*) *G-fact*: *0 < x ==> G = (G Int A) Un (G Int B)*
by (*auto simp add*: *G-def A-def B-def dvd-def*)

lemma (**in** *l*) *Lambda-over-G-lemma1*: *0 < x ==>*
setsum Lambda G = setsum Lambda ((G Int A) Un (G Int B))
apply (*rule setsum-cong*)
apply (*simp only*: *G-fact* [*THEN sym*])
by *auto*

lemma (**in** *l*) *Lambda-over-G-lemma2*: *0 < x ==>*
setsum Lambda G = setsum Lambda (G Int A) + setsum Lambda (G Int B)
apply (*subgoal-tac setsum Lambda G =*
setsum Lambda ((G Int A) Un (G Int B)))
apply (*erule ssubst*)
apply (*subst setsum-Un-disjoint*)
apply (*simp add*: *finite-subset finite-G*)
apply (*simp add*: *finite-subset finite-G*)
apply (*subgoal-tac G Int A Int (G Int B) = G Int (A Int B)*)
apply (*erule ssubst*)
apply (*simp add*: *A-B-disjoint*)
apply *force*
apply (*rule refl*)
apply (*subst G-fact* [*THEN sym*])
apply *assumption*
apply (*rule refl*)
done

lemma (**in** *l*) *Lambda-over-G-lemma3*: *0 < x ==> setsum Lambda (G Int B) = 0*
apply (*rule setsum-0'*)
apply (*insert B-kernel-for-Lambda*, *blast*)
done

lemma (**in** *l*) *Lambda-over-G-lemma4*: *0 < x ==>*
setsum Lambda G = setsum Lambda (G Int A)
apply (*auto simp add*: *Lambda-over-G-lemma2 Lambda-over-G-lemma3*)

done

lemma (**in** *l*) *G-Int-A-Un-over-J*: *G Int A = UNION J (%p. H(p) Int G)*
apply (*auto simp add*: *G-def A-def J-def H-def*)
apply (*rule-tac x = p* **in** *exI*)
apply *auto*
apply (*subgoal-tac p dvd p^a*)
prefer *2*
apply (*case-tac a*)
apply *auto*
apply (*rule dvd-trans*)
apply *auto*
done

lemma (**in** *l*) *finite-J*: *0 < x ==> finite J*
apply (*simp add*: *J-def*)
apply (*rule finite-subset-GreaterThan0AtMost-nat*)
apply *auto*
apply (*erule prime-pos*)
apply (*erule dvd-imp-le*)
apply *assumption*
done

lemma (**in** *l*) *finite-H-p*: *0 < x ==> finite (H(p))*
apply (*simp add*: *H-def*)
apply (*rule finite-subset-AtMost-nat*)
apply *auto*
done

lemma (**in** *l*) *different-H*: *0 < x ==> p1:prime ==> p2:prime ==>*
p1 ~= p2 ==> H(p1) Int H(p2) = {}
apply (*auto simp add*: *H-def*)
apply (*simp add*: *prime-prop*)
done

lemma (**in** *l*) *setsum-Lambda-G-lemma1*: *0 < x ==> setsum Lambda G =*
setsum (%p. setsum Lambda (H(p) Int G)) J
apply (*simp add*: *Lambda-over-G-lemma4*)
apply (*simp only*: *G-Int-A-Un-over-J*)
apply (*rule setsum-UN-disjoint* [*of J (%p. H(p) Int G) Lambda*])
apply (*rule finite-J*)
apply (*assumption*)
apply (*force intro*: *finite-H-p*)
apply *auto*
apply (*subgoal-tac xa*: *H p Int H j*)

apply (*subgoal-tac H p Int H j* = {})
apply *simp*
apply (*simp add*: *J-def different-H*)
apply (*rule IntI*)
apply *auto*
done

lemma (**in** *l*) *inj-on-prime-power*: *inj-on* (%*x*. (*p*::*nat*) ^ *x*) {*1*..*nmult p x*}
apply (*auto*)
apply (*rule inj-onI*)
apply (*case-tac p*:*prime*)
apply (*auto simp add*: *prime-prop2*)
apply (*simp add*: *nmult-def*)
apply (*subgoal-tac multiplicity* (*int p*) (*int x*) = *0*)
apply *simp*
apply (*subgoal-tac* (*int p*)~:*zprime*)
apply (*rule not-zprime-multiplicity-eq-0* [*of int p int x*])
apply *auto*
apply (*simp add*: *int-prime-prop*)
done

lemma *nat-int-dvd*: (*int*(*x*) *dvd int*(*y*)) = (*x dvd y*)
apply (*simp only*: *zdvd-iff-zmod-eq-0*)
apply (*simp only*: *dvd-eq-mod-eq-0*)
apply *auto*
apply (*auto simp add*: *mod-eq-0-iff*)
apply (*case-tac 0* <= *q*)
apply (*rule-tac x* = *nat q* **in** *exI*)
apply (*auto simp add*: *int-eq-iff*)
apply (*auto simp add*: *nat-mult-distrib*)
apply (*auto simp add*: *zmod-eq-0-iff*)
apply (*rule-tac x* = *int q* **in** *exI*)
apply (*auto simp add*: *zmult-int*)
done

lemma *nat-zmult-multiplicity-lemma1*:
(*int p*) ^ *multiplicity* (*int p*) (*int n*) *dvd* (*int n*)
apply (*auto simp add*: *multiplicity-zdvd*)
done

lemma *nat-zmult-multiplicity-lemma2*:
int (*p* ^ *multiplicity* (*int p*) (*int n*)) *dvd* (*int n*)
apply (*auto simp add*: *zpow-int nat-zmult-multiplicity-lemma1*)
done

lemma *nat-zmult-multiplicity*: *p ^ multiplicity (int p) (int n) dvd n*
apply (*auto simp add*: *nat-int-dvd* [*THEN sym*])
apply (*auto simp add*: *nat-zmult-multiplicity-lemma2*)
done

lemma *power-dvd-prop*: *p^b dvd (x::nat) ==> a <= b ==> p^a dvd x*
apply (*auto simp add*: *dvd-def*)
apply (*rule-tac x = k * (p ^ (b-a))* **in** *exI*)
apply *auto*
apply (*auto simp add*: *power-add* [*THEN sym*])
done

lemma *power-le-prop1* [*rule-format*]: *1 <= (p::nat) ==> a <= b --> p^a <= p^b*
apply (*induct-tac b*)
apply *auto*
apply (*subgoal-tac a = Suc n*)
apply *auto*
apply (*subgoal-tac p^n <= p*p^n*)
apply (*simp only*: *le-trans* [*of p^a p^n p*p^n*])
apply *auto*
done

lemma *power-le-prop*: *1 < p ==> p^b <= (x::nat) ==> a <= b ==> p^a <= x*
apply (*subgoal-tac p^a <= p^b*)
apply *auto*
apply (*auto simp add*: *power-le-prop1*)
done

lemma *nat-zmult-multiplicity-le*: *0<n ==>*
p ^ multiplicity (int p) (int n) <= n
apply (*subgoal-tac p^multiplicity (int p) (int n) dvd n*)
apply (*simp add*: *dvd-imp-le*)
apply (*simp add*: *nat-zmult-multiplicity*)
done

lemma *multiplicity-power-dvd*: *0<n ==> p:prime ==>*
(k <= multiplicity (int p) (int n)) = (p^k dvd n)
apply *auto*
apply (*subgoal-tac p^multiplicity (int p) (int n) dvd n*)
apply (*rule power-dvd-prop*)
apply *assumption+*
apply (*simp add*: *nat-zmult-multiplicity*)
apply (*subgoal-tac (int p):zprime*)

apply (*subgoal-tac* (*int p*) ^*k dvd* (*int n*))
apply (*simp add*: *aux9*)
apply (*subgoal-tac int p* ^ *k* = *int*(*p*^*k*))
apply (*erule ssubst*)
apply (*simp only*: *nat-int-dvd* [*THEN sym*])
apply (*simp only*: *zpow-int* [*THEN sym*])
apply (*auto simp add*: *int-prime-prop*)
done

lemma *multiplicity-power-dvd-imp1*: *0* < *n* ==> *p* : *prime* ==>
p ^ *k dvd n* ==> *k* <= *multiplicity* (*int p*) (*int n*)
apply (*subst multiplicity-power-dvd*)
apply *assumption*+
done

lemma (**in** *l*) *G-Int-Hp-eq*: *0* < *x* ==> *p*:*prime* ==>
(%*x*. *p*^*x*) ‘ {*1*..*nmult p x*} = *G Int H*(*p*)
apply *auto*
apply (*simp add*: *G-def*)
apply (*simp add*: *nmult-def*)
apply *auto*
apply (*subgoal-tac* 2 <= *p*)
apply (*simp*)
apply (*simp add*: *prime-ge-2*)
apply (*subgoal-tac* 1 < *p*)
apply (*subgoal-tac p* ^ *multiplicity* (*int p*) (*int x*) *dvd x*)
apply (*rule power-dvd-prop*)
apply *assumption*+
apply (*simp add*: *nat-zmult-multiplicity*)
apply (*simp add*: *prime-ge-2*)
apply (*subgoal-tac* 2 <= *p*)
apply *simp*
apply (*rule prime-ge-2*,*assumption*)
apply (*unfold H-def nmult-def*)
apply *clarsimp*
apply (*rule conjI*)
apply (*subgoal-tac* 1 < *p*)
apply (*subgoal-tac p* ^ *multiplicity* (*int p*) (*int x*) <= *x*)
apply (*rule power-le-prop*)
apply *assumption*+
apply (*subgoal-tac p* ^ *multiplicity* (*int p*) (*int x*) <= *x*)
apply (*subgoal-tac* 1 < *p*)
apply (*rule power-le-prop*)
apply *assumption*+
apply *force*

```
  apply assumption
  apply (simp add: nat-zmult-multiplicity-le)
  apply (subgoal-tac 2 <= p)
  apply simp
  apply (erule prime-ge-2)
  apply (rule-tac x = xa in exI)
  apply simp
  apply (unfold G-def image-def)
  apply simp
  apply clarify
  apply (rule-tac x = a in bexI)
  apply (rule refl)
  apply auto
  apply (subst multiplicity-power-dvd)
  apply auto
  apply (subgoal-tac 0 < p ^ a)
  apply (erule order-less-le-trans)backback
  apply assumption
  apply auto
done

lemma (in l) card-multiplicity-eq: 0 < x ==> p:prime ==>
    card(G Int H(p)) = multiplicity (int p) (int x)
  apply (subst G-Int-Hp-eq [THEN sym])
  apply assumption+
  apply (subst card-image)
  apply simp
  apply (rule inj-on-prime-power)
  apply (simp add: nmult-def)
done

lemma (in l) setsum-Lambda-G-int-Hp: 0 < x ==> p:prime ==>
  setsum Lambda (G Int H(p)) = ln (real p) *
    real (multiplicity (int p) (int x))
proof -
  assume 0 < x and p:prime
  have ALL x:(G Int H(p)). Lambda x = ln (real p)
    by (auto simp add: G-def H-def prems Lambda-eq)
  then have setsum Lambda (G Int H(p)) =
      setsum (%x. ln (real p)) (G Int H(p))
    apply (intro setsum-cong)
    apply (rule refl)
    apply (erule bspec)
    apply assumption
    done
```

```
  also have ... = real(card (G Int H(p))) * ln (real p)
    apply (subst setsum-constant)
    apply (rule finite-subset)
    prefer 2
    apply (rule finite-G)
    apply (rule prems)
    apply force
    apply (simp add: real-eq-of-nat)
    done
  also have card (G ∩ H p) = multiplicity (int p) (int x)
    by (rule card-multiplicity-eq)
  finally show ?thesis
    by simp
qed

lemma (in l) setsum-Lambda-G-lemma2: 0 < x ==> setsum Lambda G =
    setsum (%p. ln (real p) * real (multiplicity (int p) (int x))) J
  apply (subst setsum-Lambda-G-lemma1)
  apply assumption
  apply (rule setsum-cong)
  apply (rule refl)
  apply (subgoal-tac H xa Int G = G Int H xa)
  apply (erule ssubst)
  apply (erule setsum-Lambda-G-int-Hp)
  apply (simp add: J-def)
  apply auto
done

lemma multiplicity-eq-4: 0 < n ==> ln(real n) =
    setsum (%p. real(multiplicity p n) * ln(real p)) {p. 0 < p & p : zprime &
      p dvd n}
  apply (subgoal-tac n = 1 | 1 < n)
  apply (erule disjE)
  apply (simp add: multiplicity-p-1-eq-0 setsum-0)
  apply (elim multiplicity-eq-3)
  apply force
done

lemma multiplicity-eq-5: 0 < n ==> ln(real n) =
    setsum (%p. real(multiplicity (int p) (int n)) * ln(real p))
      {p. 0 < p & p : prime & p dvd n}
  apply (subgoal-tac real n = real (int n))
  apply (erule ssubst)
  apply (subst multiplicity-eq-4)
  apply force
```

apply (*rule setsum-reindex-cong*)
apply (*rule finite-subset-AtMost-nat*)
apply *clarify*
apply (*erule dvd-imp-le*)
apply *assumption*
apply (*subgoal-tac inj-on int* {*p*. *0* < *p* ∧ *p* ∈ *prime* ∧ *p dvd n*})
apply *assumption*
apply (*force simp add*: *inj-on-def*)
apply (*unfold image-def*)
apply *auto*
apply (*rule-tac x* = *nat x* **in** *exI*)
apply (*auto simp add*: *nat-prime-prop*)
apply (*subgoal-tac nat x dvd nat* (*int n*))
apply *simp*
apply (*rule nat-int-dvd-prop*)
apply *assumption*
apply *assumption*
apply (*simp add*: *int-nat-dvd-prop*)
apply (*rule ext*)
apply (*unfold o-def*)
apply (*subgoal-tac real p* = *real* (*int p*))
apply (*erule subst*)
apply (*rule refl*)
apply *simp*
done

lemma *ln-eq-setsum-Lambda*: *0* < (*n*::*nat*) ==> *ln* (*real n*) =
setsum Lambda {*d*. *d dvd n*}
apply (*subgoal-tac* {*d*. *d dvd n*} = {*d*. *0* < *d* & *d dvd n*})
apply (*erule ssubst*)
apply (*subst l.setsum-Lambda-G-lemma2*)
apply *assumption*
apply (*subst multiplicity-eq-5*)
apply *assumption*
apply (*rule setsum-cong*)
apply *auto*
apply (*rule prime-pos*)
apply *assumption*
apply (*erule dvd-pos-pos*)
apply *assumption*
done

lemma *ln-eq-setsum-Lambda2*: *0* < *n* ==> *ln* (*real n*) =
(∑ *x*:{(*p*, *a*). *0* < *a* & *p* : *prime* & *p* ^ *a dvd n*}.
ln (*real* (*fst x*)))

proof −
assume *0 < (n::nat)*
have *ln (real n) = (∑ d:{d. d dvd n}. Lambda d)*
by (*rule ln-eq-setsum-Lambda*)
also have *... =*
(∑ d:{d. d dvd n & (EX p a. 0 < a & p : prime & d = p ^ a)}.
Lambda d) +
(∑ d:{d. d dvd n & ~(EX p a. 0 < a & p : prime & d = p ^ a)}.
Lambda d) (**is** *... = ?term1 + ?term2*)
apply (*subst setsum-Un-disjoint* [*THEN sym*])
apply (*rule finite-subset*)
prefer *2*
apply (*rule finite-nat-dvd-set*)
apply (*rule prems*)
apply *force*
apply (*rule finite-subset*)
prefer *2*
apply (*rule finite-nat-dvd-set*)
apply (*rule prems*)
apply *force*
apply *blast*
apply (*rule setsum-cong*)
apply *blast*
apply *force*
done
also have *?term2 = 0*
apply (*rule setsum-0'*)
apply (*rule ballI*)
apply (*rule Lambda-eq2*)
apply *auto*
done
also have *?term1 + 0 = ?term1*
by *simp*
also have *... = (∑ x:{(p,a). 0 < a & p : prime & p^a dvd n}.*
Lambda((%x. (fst x) ^(snd x))x))
apply (*subst setsum-reindex'* [*THEN sym*])**back**
apply (*rule finite-subset*)
prefer *2*
apply (*rule l.finite-C* [*of n*])
apply *auto*
apply (*erule dvd-imp-le*)
apply (*rule prems*)
apply (*rule subset-inj-on*)
prefer *2*
apply (*rule l.inj-on-C*)

apply *auto*
apply (*erule dvd-imp-le*)
apply (*rule prems*)
apply (*rule setsum-cong*)
apply (*auto simp add*: *image-def*)
done
finally show *?thesis*
apply (*elim ssubst*)
apply (*rule setsum-cong2*)
apply (*rule Lambda-eq*)
apply *auto*
done
qed

end

theory *Chebyshev2* = *Chebyshev1*:

declare *binomial-Suc-Suc* [*simp del*]

38.6 General facts – move some of these into libraries!

lemma *setsum-multiplier*: *finite A* ==> (*k*::*nat*)∗(*setsum f A*) = *setsum* (%*i*. *k*∗*f*(*i*)) *A*
apply (*induct set*: *Finites*)
apply *force*
apply (*simp add*: *setsum-insert*)
apply (*simp add*: *add-mult-distrib2*)
done

lemma *setsum-multiplier-real*: *finite A* ==> (*k*::*real*)∗(*setsum f A*) = *setsum* (%*i*. *k* ∗ *f*(*i*)) *A*
apply (*induct set*: *Finites*)
apply *force*
apply (*simp add*: *setsum-insert*)
apply (*subst mult-commute*)**back**
apply (*subst real-add-mult-distrib*)
apply *auto*
apply (*subst mult-commute*)
apply *assumption*
done

lemma *empty-interval* [*simp*]: $\{Suc\ b..b\} = \{\}$
apply *auto*
done

lemma *interval1* [*simp*]: $\{(a::nat)..a\} = \{a\}$
by *auto*

lemma *interval2* [*simp*]: $\{(a::nat)..a(\} = \{\}$
by *auto*

lemma *interval3* [*simp*]: $\{)(a::nat)..a\} = \{\}$
by *auto*

lemma *insert-interval*: $\{(0::nat)..Suc\ n(\} = insert\ n\ \{..n(\}$
apply *auto*
done

lemma *right-open-prop*: $n\ \sim: \{(0::nat)..n(\}$
apply *auto*
done

lemma *right-closed-prop*: $Suc\ n\ \sim: \{(0::nat)..n\}$
by *auto*

lemma *insert-interval-closed*: $\{(0::nat)..Suc\ n\} = insert\ (Suc\ n)\ \{..n\}$
by *auto*

lemma *setsum-interval-insert*: $setsum\ f\ \{0..Suc\ n(\} = setsum\ f\ \{0..n(\} + f\ n$
apply (*auto simp add:insert-interval right-open-prop setsum-insert*)
apply (*subgoal-tac* $\{..n(\} = \{0..n(\}$)
apply (*erule ssubst*)
apply *auto*
apply (*simp only*: *plus-ac0*)
done

lemma *setsum-interval-insert-closed*: $setsum\ f\ \{0..Suc\ n\} = setsum\ f\ \{0..n\} + f\ (Suc\ n)$
apply (*auto simp add*: *insert-interval-closed right-closed-prop setsum-insert*)
apply (*subgoal-tac* $\{..n\} = \{0..n\}$)
apply (*erule ssubst*)
apply (*auto simp add*: *plus-ac0*)
done

lemma *setsum-index-shift-right*: $setsum\ f\ \{(0::nat)..b\} = setsum\ (\%i.\ f(i-k))\ \{k..(b+k)\}$

apply (*induct-tac b*)
apply *force*
apply (*simp only*: *setsum-interval-insert-closed*)
apply (*subgoal-tac* (*Suc n* + *k*) ~: {*k*..*n*+*k*})
apply (*subgoal-tac* {*k*..*Suc n* + *k*} = *insert* (*Suc n* + *k*) {*k*..*n*+*k*})
apply (*simp add*: *setsum-insert*)
apply (*auto simp add*: *plus-ac0*)
done

lemma *setsum-index-shift-left*: *a*<= *b* ==> *setsum f* {(*a*::*nat*)..*b*} = *setsum* (%*i*. *f*(*a*+*i*)) {*0*..(*b*−*a*)}
apply (*simp add*: *setsum-index-shift-right* [*of* (%*i*. *f*(*a*+*i*)) *b*−*a a*])
apply (*subgoal-tac* (∑ *i*::*nat*:{*a*..*b*}. *f* (*a* + (*i* − *a*))) = (∑ *i*::*nat*:{*a*..*b*}. *f*(*i*)))
apply (*erule ssubst*)
apply *force*
apply (*rule setsum-cong*)
apply *auto*
done

lemma *real-nat-division*: *0* < *b* ==> *a mod b* = (*0*::*nat*) ==> *real*(*a div b*) = *real*(*a*)/*real*(*b*)
apply (*subgoal-tac 0* < *real*(*b*))
apply (*simp add*: *nonzero-eq-divide-eq*)
apply (*simp only*: *times-ac1 real-of-nat-mult* [*THEN sym*])
apply (*simp only*: *mult-div-cancel*)
apply *auto*
done

lemma *div-combine*: *a mod b* = (*0*::*nat*) ==> (*c* ∗ (*a div b*)) = ((*c*∗*a*) *div b*)
apply *auto*
done

lemma *div-combine-2*: *c mod b* = (*0*::*nat*) ==> (*a* ∗ (*c div b*)) = ((*c*∗*a*) *div b*)
apply *auto*
done

lemma *nat-le-diff*: (*x*::*nat*) <= *y* − *k* ==> *x* <= *y*
apply (*case-tac y* <= *k*)
apply *force*
apply (*case-tac y* = *k*)
apply *force*
apply (*subgoal-tac k* < *y*)
apply *auto*
apply (*subgoal-tac x* + *k* <= *y* − *k* + *k*)
apply (*subgoal-tac k* <= *y*)

```
  apply (simp add: le-add-diff-inverse2)
  apply force+
  apply (simp only: add-le-mono1 [of - - k])
done

lemma div-add: a mod c = (0::nat) ==> b mod c = 0 ==> (a div c + b div c)
= ((a+b) div c)
  apply (simp add: div-add1-eq)
done

lemma factorial-never-0 [simp]: fact(n) ~= 0
  by simp

lemma factorial-gt-0 [simp]: 0 < fact n
  apply simp
  done

lemma product-eq: a ~= 0 ==> b ~= 0 ==> d ~= 0 ==> (a::nat)*b*c = d
==> a*b*e = d ==> c = e
  apply auto
done

lemma factorial-pos: 1 <= k ==> fact k = k * fact(k - 1)
  apply (case-tac k)
  apply auto
done

lemma product-div: 0 < (b::nat) ==> a * b = c ==> a = c div b
  apply auto
  done

lemma lt-gt-product: 0 < a ==> 0 < b ==>  0 < d ==> (a::nat) * b = c * d
==> a < c ==> (d < b)
  proof-
    assume a: 0 < a
    assume b: 0 < b
    assume d: 0 < d
    assume eq: a * b = c * d
    assume ineq1: a < c
    have step1: a * b * d < c * b * d
      apply (auto)
      apply (auto simp add: d b ineq1)
      done
    from step1 have step2: c * d * d < c * b * d
      apply (simp add: eq)
```

```
    done
  from step2 have step3: d * d < b * d
    apply auto
    done
  also from step3 show ?thesis
    apply auto
    done
  qed

lemma gt-lt-product: 0 < a ==> 0 < c ==> 0 < d ==> (a::nat) * b = c * d
==> d < b ==> (a < c)
  proof−
    assume a: 0<a
    assume c: 0<c
    assume d: 0<d
    assume eq: a*b = c*d
    assume ineq: d < b
    have step1: a * d * c < a * b * c
      apply auto
      apply (auto simp add: a c ineq)
      done
    from step1 have step2: a * d * c < c * d * c
      apply (simp only: eq)
      done
    from step2 have step3: a * d < c * d
      apply auto
      done
    from step3 show ?thesis
      by auto
    qed

lemma gt-lt-product-iff: 0 < a ==> 0 < b ==> 0 < c ==> 0 < d ==> (a::nat)
* b = c * d ==> (d < b) = (a < c)
  apply (auto)
  apply (force intro: gt-lt-product, force intro: lt-gt-product)
done

lemma eq-product-iff: 0 < a ==> 0 < b ==> 0 < c ==> 0 < d ==> (a::nat)
* b = c * d ==> (d = b) = (a = c)
  apply auto
done

lemma nat-less-diff: (a < b − (c::nat)) = (a + c < b)
  apply (induct b c rule: diff-induct)
  apply auto
```

done

lemma *sum-greater-than-part*: *finite A ==> a : A ==>*
ALL x:A. (0 < (f(x)::nat)) ==> (A - {a}) ~= {} ==>
f(a) < setsum f A
apply (*subgoal-tac A = insert a (A - {a})*)
apply (*erule ssubst*)
apply (*subst setsum-insert*)
apply *force+*
done

lemma *setsum-constant-bound*: *finite A ==> ALL x:A. (f(x) <= k) ==> setsum f A <= card(A)*k*
apply (*induct set*: *Finites*)
apply *force*
apply (*auto simp add*: *setsum-insert*)
done

lemma *chained-inequalities-up* [*rule-format*]: (*ALL i < n. ((f(i)::nat) < f(Suc i))) --> k < n --> f(k) < f(n)*
apply (*induct-tac n*)
apply *force*
apply (*simp only*: *le-simps*)
apply (*rule impI*)
apply (*rule impI*)
apply *auto*
apply (*subgoal-tac k = n*)
apply (*frule-tac x = n* **in** *spec*)
apply (*subgoal-tac n <= n*)
apply (*frule mp*)
apply *auto*
done

lemma *chained-inequalities-down* [*rule-format*]: (*ALL i < n. f(Suc i) < (f(i)::nat)) --> k < n --> f(Suc k) < f(0)*
apply (*induct-tac k*)
apply *force*
apply *force*
done

lemma *sum-is-greater-than-part*: *finite A ==> B <= A ==> ALL x:A. (0 < (f(x)::nat)) ==> A - B ~= {} ==> setsum f B < setsum f A*
apply (*subgoal-tac A = B Un (A - B)*)
apply (*erule ssubst*)

apply (*subgoal-tac finite B*)
apply (*subgoal-tac finite* (*A − B*))
apply (*frule setsum-Un-disjoint* [*of B A − B f*])
apply *force+*
apply (*simp add*: *finite-subset*)
apply *force*
done

lemma *setprod-dvd*: *finite A ==> (x::nat):A ==> x dvd setprod id A*
apply (*induct set*: *Finites*)
apply *force*
apply (*simp add*: *setprod-insert*)
apply *auto*
apply (*simp add*: *dvd-mult*)
done

lemma *prime-divides-product*: *m+2 <= p ==> p <= Suc(2∗m) ==> p dvd setprod id {m+2..Suc(2∗m)}*
by (*rule setprod-dvd*, *auto*)

lemma *factorial-Suc*: *fact n + n ∗ fact n = fact* (*Suc n*)
apply *auto*
done

lemma *primes-dvd-one-or-other-left*: *a : prime ==> (a::nat) dvd b∗c ==> ~ (a dvd b) ==> a dvd c*
apply (*frule prime-dvd-mult*)
apply *auto*
done

lemma *primes-dvd-one-or-other-right*: *a : prime ==> (a::nat) dvd b∗c ==> ~ (a dvd c) ==> a dvd b*
apply (*frule prime-dvd-mult*, *auto*)
done

lemma *prime-dividing-factorial* [*rule-format*]: *p : prime −−> p dvd fact n −−> p <= n*
apply (*induct-tac n*)
apply (*simp add*: *prime-def*)
apply *auto*
apply (*simp only*: *factorial-Suc*)
apply (*simp only*: *fact-Suc*)
apply (*subgoal-tac p dvd Suc n*)
apply (*rule dvd-imp-le*)
apply *force*

apply *force*
apply (*rule primes-dvd-one-or-other-right*)
apply *auto*
done

lemma *prime-not-divide-less*: *p* : *prime* ==> *m*+*2* <= *p* ==> *p* <= *Suc*(*2*∗*m*) ==> ~ *p dvd fact*(*m*)
apply (*rule contrapos-pn*)
apply *auto*
apply (*frule prime-dividing-factorial*)
apply *auto*
done

lemma *factorial-setprod-def*: *fact n* = *setprod id* {*1*..*n*}
apply (*induct-tac n*)
apply (*simp only*: *fact-0*)
apply (*subgoal-tac* {*1*..*0*} = {})
apply (*erule ssubst*)
apply *force*
apply *force*
apply (*simp only*: *fact-Suc*)
apply (*subgoal-tac* {*1*..*Suc n*} = {*1*..*n*} *Un* {*Suc n*})
apply (*subgoal-tac finite* {*1*..*n*})
apply (*subgoal-tac finite* {*Suc n*})
apply (*subgoal-tac* {*1*..*n*} *Int* {*Suc n*} = {})
apply (*simp only*: *setprod-Un-disjoint*)
apply (*subgoal-tac setprod id* {*1*..*n*} = *fact n*)
apply (*erule ssubst*)**backbackback**
apply *simp*
apply (*simp only*: *nat-mult-commute*)
apply *force*+
done

lemma *setprod-interval-insert-1*: *setprod f* {*1*..*Suc n*} = *setprod f* {*1*..*n*} ∗ (*f*(*Suc n*)::*'a*::*semiring*)
apply (*subgoal-tac* {*1*..*Suc n*} = {*1*..*n*} *Un* {*Suc n*})
apply (*erule ssubst*)
apply (*subgoal-tac finite* {*1*..*n*})
apply (*subgoal-tac finite* {*Suc n*})
apply (*simp add*: *setprod-Un-disjoint*)
apply *auto*
apply (*simp add*: *mult-ac*)
done

lemma *setprod-interval-insert*: *b* <= *Suc a* ==> *setprod f* {*b*..*Suc a*} = *setprod*

f {*b*..*a*} * (*f*(*Suc a*)::'*a*::*semiring*)
apply (*subgoal-tac* {*b*..*Suc a*} = {*b*..*a*} *Un* {*Suc a*})
apply (*erule ssubst*)
apply (*subgoal-tac finite* {*b*..*a*})
apply (*subgoal-tac finite* {*Suc a*})
apply (*simp add*: *setprod-Un-disjoint*)
apply (*auto simp add*: *mult-ac*)
done

lemma *setprod-interval-gt-0* [*rule-format*]: *0* < *b* ==> *b* <= *a* --> *0* < *setprod id* {*b*..(*a*::*nat*)}
apply (*induct-tac a*)
apply *force*
apply *auto*
apply (*subgoal-tac b* = *Suc n*)
apply (*erule ssubst*)
apply *force*+
apply (*subst setprod-interval-insert*)
apply *force*+
done

lemma *setprod-interval-1-gt-0* [*simp*]: *0* < *setprod id* {*1*..(*n*::*nat*)}
apply (*case-tac n*)
apply *force*
apply (*rule setprod-interval-gt-0*)
apply *force*+
done

lemma *setprod-interval-Suc0-gt-0* [*simp*]: *0* < *setprod id* {*Suc 0*..(*n*::*nat*)}
apply (*subgoal-tac Suc 0* = *1*)
apply (*erule ssubst*)
apply (*simp only*: *setprod-interval-1-gt-0*)
by *force*

lemma *setprod-interval-multiply* [*rule-format*]: *a* <= *b* --> *setprod id* {*1*..*b*} = *setprod id* {*1*..*a*} * *setprod id* {*Suc a*..*b*}
apply (*induct-tac b*)
apply (*rule impI*)
apply (*subgoal-tac a* = *0*)
apply (*erule ssubst*)
apply (*subgoal-tac* {*1*..*0*} = {})
apply (*erule ssubst*)
apply (*subgoal-tac* {*Suc*(*0*)..*0*} = {})
apply (*erule ssubst*)
apply *force*+

apply (*rule impI*)
apply (*case-tac a <= n*)
apply (*frule mp*)
apply *force*
apply (*subgoal-tac setprod id {1::nat..Suc n} = setprod id {1::nat..n} * id (Suc n)*)
apply (*erule ssubst*)**back**
apply (*erule ssubst*)
apply (*subgoal-tac setprod id {Suc a..n} * id (Suc n) = setprod id {Suc a..Suc n}*)
apply (*subst nat-mult-assoc*)
apply (*erule ssubst*)
apply (*force*)
apply (*subgoal-tac setprod id {Suc a..Suc n} = setprod id {Suc a..n} * id (Suc n)*)
apply *force*
apply (*subgoal-tac Suc a <= Suc n*)
apply (*simp only*: *setprod-interval-insert*)
apply *force+*
apply (*simp only*: *setprod-interval-insert*)
apply (*subgoal-tac a = Suc n*)
apply (*erule ssubst*)
apply (*subgoal-tac {Suc (Suc n)..Suc n} = {}*)
apply (*erule ssubst*)
apply *force+*
done

lemma *setprod-interval-mod*: *a <= b ==> setprod id {1..b} mod setprod id {1..(a::nat)} = 0*
apply (*simp only*: *mod-eq-0-iff*)
apply (*rule-tac x = setprod id {Suc a..b}* **in** *exI*)
apply (*simp only*: *setprod-interval-multiply*)
done

lemma *div-multiply*: *0 < b ==> 0 < c ==> ((a::nat) = b*c) ==> (b = a div c)*
apply *auto*
done

lemma *setprod-interval-div*: *a <= b ==> (setprod id {1..b}) div (setprod id {1..a}) = (setprod id {Suc a..b})*
apply (*case-tac b = 0*)
apply (*subgoal-tac a = 0*)
apply (*erule ssubst*)+
apply *force+*

apply (*case-tac a = 0*)
apply (*erule ssubst*)
apply *force*
apply (*rule sym*)
apply (*rule div-multiply*)
apply (*case-tac a = b*)
apply (*erule ssubst*)
apply *force*
apply (*rule setprod-interval-gt-0*)
apply *force+*
apply (*subst nat-mult-commute*)
apply (*simp only*: *setprod-interval-multiply*)
done

lemma *factorial-div*: *b <= a ==> (fact a) div (fact b) = setprod id {(Suc b)..a}*
apply (*simp only*: *factorial-setprod-def setprod-interval-div*)
done

lemma *primes-always-ge-2* [*simp*]: *p : prime ==> 2 <= p*
apply (*auto simp add*: *prime-def*)
done

lemma *relprime-dvd-prod-dvd*: *gcd(a,b) = 1 ==> a dvd m ==> b dvd m ==> (a*b) dvd (m::nat)*
apply (*unfold dvd-def*)
apply *clarify*
apply (*subgoal-tac a dvd ka*)
apply (*force simp add*: *dvd-def*)
apply (*subst relprime-dvd-mult-iff* [*THEN sym*])
apply *assumption*
apply (*auto simp add*: *mult-commute dvd-def*)
apply (*rule exI*)
apply (*erule sym*)
done

lemma *distinct-primes-gcd-1*: *p : prime ==> q : prime ==> p ~= q ==> gcd(p,q) = 1*
apply (*rule prime-imp-relprime*)
apply (*auto simp add*: *prime-def*)
done

lemma *all-relprime-prod-relprime-nat*: *finite A ==> ALL x:A. gcd(x,y) = 1 ==> gcd(setprod id A, y) = 1*
apply (*induct set*: *Finites*)
apply (*auto simp add*: *gcd-mult-cancel*)

done

lemma *prime-product-dvd*: *finite A ==> ALL x:A. (x : prime & x dvd M) ==> setprod id A dvd M*
apply (*induct set*: *Finites*)
apply *auto*
apply (*rule relprime-dvd-prod-dvd*)
apply (*subst gcd-commute*)
apply (*rule all-relprime-prod-relprime-nat*)
apply *assumption*
apply (*rule ballI*)
apply (*rule distinct-primes-gcd-1*)
apply *auto*
done

lemma *pi-setsum-def*: *pi (x) = real (setsum (%x. (1::nat)) {y::nat. y <= x & y : prime})*
proof−
have *pi (x) = real(card {y::nat. y <= x & y : prime})*
by (*auto simp add*: *pi-def*)
also have *... = real(setsum (%x. (1::nat)) {y::nat. y <= x & y : prime})*
apply (*subst real-of-nat-inject*)
apply (*rule card-eq-setsum*)
apply (*rule finite-subset* [*of* - {*0..x*}])
apply *auto*
done
finally show *?thesis*.
qed

lemma *set-difference*: *{x::nat. x <= y & P(x)} − {x::nat. x <= z & P(x)} = {x::nat. z < x & x <= y & P(x)}*
by *auto*

lemma *pi-mono*: *y <= x ==> pi(y) <= pi(x)*
apply (*auto simp add*: *pi-def*)
apply (*rule card-mono*)
apply (*rule finite-subset* [*of* - {*0..x*}])
apply *force*+
done

lemma *finite-simp* [*simp*]: *finite {z::nat. z <= x & P(x)}*
apply (*rule finite-subset* [*of* - {*0..x*}])
apply *force*+
done

lemma *pi-diff*: $y <= x ==> pi(x) - pi(y) = real\ (setsum\ (\%x.\ (1::nat))\ \{z::nat.\ y < z\ \&\ z <= x\ \&\ z : prime\})$
proof−
assume $y <= x$
have *mono*: $pi\ (y) <= pi(x)$
by (*insert prems*, *erule pi-mono*)
have $pi(x) - pi(y) = real(card\ \{z::nat.\ z <= x\ \&\ z : prime\}) - real(card\ \{z::nat.\ z <= y\ \&\ z : prime\})$
by (*simp add*: *pi-def*)
also have ... = $real(card\ \{z::nat.\ z <= x\ \&\ z : prime\} - card\ \{z::nat.\ z <= y\ \&\ z : prime\})$
apply (*rule real-of-nat-diff* [*THEN sym*])
apply (*rule card-mono*)
apply (*rule finite-subset* [*of* - $\{0..x\}$])
apply (*insert prems*, *auto*)
done
also have ... = $real(card\ \{z::nat.\ y < z\ \&\ z <= x\ \&\ z : prime\})$
apply (*subst real-of-nat-inject*)
apply (*subst card-Diff-subset*)
apply (*rule finite-subset* [*of* - $\{0..x\}$])
apply *force*+
apply *auto*
apply (*insert prems*, *force*)
apply (*subst set-difference*)
apply *auto*
done
finally show *?thesis*
apply (*subst card-eq-setsum* [*THEN sym*])
apply (*rule finite-subset* [*of* - $\{0..x\}$])
apply *force*+
done
qed

lemma *pi-less* [*simp*]: $pi(x) <= real\ x$
apply (*auto simp add*: *pi-def*)
apply (*subgoal-tac* $x = card\ \{1..x\}$)
apply (*erule ssubst*)
apply (*rule card-mono*)
apply *force*+
apply *auto*
apply (*subgoal-tac* $2 <= xa$)
apply *arith*
apply *simp*
done

lemma *mono-setsum-le*: *finite A ==> ALL x:A. f x <= (y::'a::ordered-semiring) ==> setsum f A <= setsum (%x. y) A*
apply (*induct set*: *Finites*)
apply *auto*
apply (*rule add-mono*)
apply *force+*
done

lemma *setsum-real*: *finite A ==> real (setsum (f::nat => nat) (A::nat set)) = setsum (real o f) A*
apply (*induct set*: *Finites*)
apply *auto*
done

lemma *primes-always-ge-1* [*simp*]: *p : prime ==> 1 <= p*
apply (*frule primes-always-ge-2*)
apply *arith*
done

lemma *powr-divide-denom*: *x powr a / x powr b = 1 / (x powr (b - a))*
apply (*simp add*: *powr-divide2*)
apply (*subst powr-minus-divide* [*THEN sym*])
apply *simp*
done

lemma *setprod-pos*: *finite A ==> ALL x:A. ((0::'a::ordered-semiring) < x) ==> 0 < setprod id A*
apply (*induct set*: *Finites*)
apply *force*
apply (*simp add*: *setprod-insert*)
apply *clarify*
apply (*simp add*: *mult-pos*)
done

lemma *setsum-setprod-ln*: *finite A ==> ALL x:A. (0 < x) ==> setsum ln A = ln (setprod id A)*
apply (*induct set*: *Finites*)
apply *force*
apply (*simp add*: *setsum-insert*)
apply (*subgoal-tac 0 < setprod id F*)
apply (*simp add*: *ln-mult*)
apply (*rule setprod-pos*)
apply *auto*
done

lemma *setsum-setprod-ln-real*: *finite A ==> ALL x:A. ((0::nat) < x) ==> setsum (ln o real) A = ln(real(setprod id A))*
apply (*induct set*: *Finites*)
apply *force*
apply *auto*
apply (*subgoal-tac 0 < real(setprod id F)*)
apply (*subgoal-tac 0 < real x*)
apply (*simp add*: *ln-mult*)
apply *force*
apply (*subgoal-tac 0 < setprod id F*)
apply *force*
apply (*rule setprod-pos*)
apply *auto*
done

lemma *theta-setsum-pos-def*: *theta x = setsum lprime {2::nat..x}*
proof−
have *step1*: *theta x = setsum lprime {0::nat..x}*
by (*simp add*: *theta-setsum-eq1*)
also have *... = setsum lprime ({0::nat} Un {1::nat..x})*
apply (*rule setsum-cong*)
apply *force+*
done
also have *... = setsum lprime {0::nat} + setsum lprime {1::nat..x}*
apply *force*
done
also have *... = setsum lprime {1::nat..x}*
apply *auto*
apply (*auto simp add*: *lprime-def*)
apply (*frule primes-always-ge-2*)
apply *force*
done
also have *... = setsum lprime ({1::nat} Un {2::nat..x})*
apply (*case-tac x = 0*)
apply (*simp add*: *lprime-def*)
apply (*rule setsum-cong*)
apply *auto*
done
finally show *?thesis*
apply (*auto simp add*: *lprime-def primes-always-ge-2*)
done
qed

lemma *theta-def-2*: *theta x = ln (real (setprod id {p. p:prime & p <= x}))*
apply (*subst l.sum-over-F2* [*THEN sym*])

apply (*subgoal-tac finite* {*p*. *p*:*prime* & *p* <= *x*})
apply (*subgoal-tac ALL x*:{*p*. *p*:*prime* & *p* <= *x*}. (*0* < *x*))
apply (*simp only*: *setsum-setprod-ln-real* [*THEN sym*])
apply (*rule setsum-cong*)
apply *auto*
apply (*subgoal-tac 2* <= *xa*)
apply (*arith*)
apply (*simp add*: *primes-always-ge-2*)
apply (*rule finite-subset* [*of* - {*0*..*x*}])
apply *auto*
done

lemma *lprime-lt-ln*: *finite A* ==> *0* ~: *A* ==> *setsum lprime A* <= *setsum* (*ln o real*) *A*
apply (*rule setsum-le-cong*)
apply (*auto simp add*: *lprime-def*)
apply (*subgoal-tac 1* <= *x*)
apply (*subgoal-tac 1* <= *real*(*x*))
apply (*simp add*: *ln-ge-zero*)
apply *force*+
apply (*subgoal-tac x* ~= *0*)
apply *force*+
apply (*case-tac x* = *0*)
apply *force*
apply *force*
done

lemma *setprod-gt-0-iff*: *finite B* ==> ((*0*::*nat*) < *setprod id B*) = (*0* ~: *B*)
apply *auto*
apply (*subgoal-tac setprod id B* = *0*)
apply *force*
apply (*subgoal-tac B* = *insert 0* (*B* − {*0*}))
apply (*erule ssubst*)
apply (*subst setprod-insert*)
apply *force*+
apply (*induct set*: *Finites*)
apply *force*
apply *auto*
done

lemma *setsum-subset*: *finite B* ==> *A* <= *B* ==> *ALL x*:*B*. *0* <= *f x* ==> ((*setsum f A*)::'*a*::*ordered-semiring*) <= *setsum f B*
proof−
assume *finite B* **and** *A* <= *B* **and** *ALL x*:*B*. *0* <= *f x*
have *setsum f A* = *setsum f A* + *0*

```
    apply auto
    done
  also have ...<= setsum f A + setsum f (B - A)
    apply (simp only: add-le-cancel-left)
    apply (rule setsum-nonneg)
    apply (insert prems, auto)
    done
  also have ... = setsum f (A Un (B - A))
    apply (rule setsum-Un-disjoint [THEN sym])
    apply (insert prems, auto)
    apply (auto simp add: finite-subset)
    done
  also have ... = setsum f B
    apply (insert prems, auto)
    apply (rule setsum-cong)
    apply auto
    done
  finally show ?thesis.
  qed

lemma setprod-lt [rule-format]: finite B ==> 0 ~: B ==>
      ALL A. (A <= B -->((setprod id A)::nat) <= setprod id B)
 proof (clarify)
  assume finite B and 0 ~: B
  fix A
  assume A <= B
  show setprod id A <= setprod id B
    proof-
  have setprod id B = setprod id (A Un (B - A))
    apply (rule setprod-cong)
    apply (insert prems)
    apply force+
    done
  also have ... = (setprod id A) * (setprod id (B - A))
    apply (rule setprod-Un-disjoint)
    apply (insert prems, auto)
    apply (auto simp add: finite-subset)
    done
  also have setprod id A <= ...
    apply auto
    apply (rule Suc-leI)
    apply (subst setprod-gt-0-iff)
    apply (insert prems, auto simp add: finite-subset)
    done
  finally show ?thesis.
```

qed
qed

lemma *lt-one-minus*: $x <= n ==> x \sim= n ==> x <= (n - Suc\ 0)$
apply (*subst Suc-le-mono* [*THEN sym*])
apply *simp*
done

lemma *only-even-prime*: $p : prime ==> even\ p ==> p = 2$
apply (*case-tac* $p = 2$)
apply *force*
apply (*simp add*: *even-mult-two-ex*)
apply (*erule exE*)
apply (*auto simp add*: *prime-def*)
done

lemma *dvd-div*: $a : prime ==> a\ dvd\ b ==> \sim(a\ dvd\ c) ==> b\ mod\ c = 0 ==> a\ dvd\ (b\ div\ c)$
apply *auto*
apply (*frule prime-dvd-mult*)
apply *auto*
done

38.7 Binomial coefficients – break this out!

lemma *binomial-factorial-def-lemma1*: $fact\ (Suc\ y) * fact(Suc\ x - Suc\ y) * (x\ choose\ y) = (Suc\ y) * fact\ y * fact(x-y)*(x\ choose\ y)$
apply *simp*
done

lemma *binomial-factorial-def-lemma2*: $fact\ (Suc\ y) * fact(Suc\ x - Suc\ y) * (x\ choose\ Suc\ y) = (x-y)*fact\ (Suc\ y) * fact(x - Suc\ y) * (x\ choose\ Suc\ y)$
apply *simp*
apply (*case-tac* $x <= y$)
apply *auto*
apply (*simp add*: *binomial-eq-0-iff*)
apply (*case-tac* $x = Suc\ y$)
apply *force*
apply (*subgoal-tac* $y < x$)
apply *auto*
apply (*subgoal-tac* $x-y = Suc\ (x - Suc\ y)$)
apply (*erule ssubst*)
apply *force*
apply *arith*
done

lemma *binomial-factorial-def* [*rule-format*]: *ALL n k. k<=n --> fact(k) * fact(n−k) * (n choose k) = fact(n)*
apply (*rule allI*)
apply (*induct-tac n*)
apply *force*
apply *auto*
apply (*case-tac k*)
apply (*erule ssubst*)
apply *force*
apply (*subgoal-tac Suc nat <= Suc na*)
apply (*erule ssubst*)
apply (*simp only*: *binomial-Suc-Suc*)
apply (*simp only*: *ring-distrib*)
apply (*simp only*: *binomial-factorial-def-lemma1 binomial-factorial-def-lemma2*)
apply (*case-tac Suc nat = Suc na*)
apply *force*
apply (*subgoal-tac Suc nat <= na*)
apply (*frule-tac x = Suc nat* **in** *spec*)
apply (*frule-tac x = nat* **in** *spec*)
apply (*subgoal-tac nat <= na*)
apply (*drule mp*)
apply *force*
apply (*drule mp*)
apply *force*
apply (*subgoal-tac Suc nat * fact nat * fact (na − nat) * (na choose nat) = Suc nat * (fact nat * fact (na − nat) * (na choose nat))*)
apply (*erule ssubst*)
back
back
apply (*erule ssubst*)
back
apply (*subgoal-tac (na − nat) * fact (Suc nat) * fact (na − Suc nat) * (na choose Suc nat) = (na − nat) * (fact (Suc nat) * fact (na − Suc nat) * (na choose Suc nat))*)
apply (*erule ssubst*)
back
apply (*erule ssubst*)
apply (*simp add*: *ring-distrib* [*THEN sym*])
apply *force*
apply (*simp only*: *nat-mult-assoc*)
apply *auto*
done

lemma *binomial-mod*: *k <= n ==> fact(n) mod (fact(k) * fact(n−k)) = 0*
apply (*simp add*: *mod-eq-0-iff*)
apply (*rule-tac x = n choose k* **in** *exI*)
apply (*simp add*: *binomial-factorial-def*)
done

lemma *binomial-dvd*: *k <= n ==> (fact(k) * fact(n−k)) dvd fact(n)*
apply (*simp add*: *binomial-mod dvd-eq-mod-eq-0*)
done

lemma *binomial-factorial-def-div*: *k <= n ==> n choose k = fact(n) div (fact(k) * fact (n−k))*
apply (*frule binomial-dvd*)
apply (*drule dvd-mult-div-cancel*)
apply (*frule binomial-factorial-def*)
apply (*rule product-div*)
apply (*rule mult-pos*)
apply *auto*
apply (*auto simp add*: *mult-commute*)
done

lemma *binomial-step*: *1 <= k ==> k <= n ==> k * (n choose k) = (n − (k − 1))*(n choose (k − 1))*
apply (*subgoal-tac k − 1 <= n*)
apply (*simp only*: *binomial-factorial-def-div*)
apply (*frule binomial-mod* [*of k n*])
apply (*frule binomial-mod* [*of k − 1 n*])
apply (*simp only*: *div-combine*)
apply (*simp only*: *factorial-pos* [*of k*])
apply (*subgoal-tac 1 <= n − (k − 1)*)
apply (*simp only*: *factorial-pos* [*of n − (k − 1)*])
apply (*subgoal-tac 0 < k*)
apply (*subgoal-tac 0 < (n − (k − 1))*)
apply (*subgoal-tac* (*fact* (*k* − (*1*::*nat*)) * ((*n* − (*k* − (*1*::*nat*))) * *fact* (*n* − (*k* − (*1*::*nat*)) − (*1*::*nat*)))) = ((*n* − (*k* − *1*)) * (*fact* (*k* − (*1*::*nat*)) * *fact* (*n* − (*k* − (*1*::*nat*)) − (*1*::*nat*)))))
apply (*erule ssubst*)
back
back
apply (*simp add*: *div-mult-mult1* [*of k - -*] *div-mult-mult1* [*of* (*n* − (*k* − *1*)) - -])
apply *force*
apply *force*
apply *force*

apply *simp*
apply (*subgoal-tac 0 <= n−k*)
apply (*simp add*: *Suc-le-mono Suc-diff-le*)
apply *force*
apply (*rule Suc-leD*)
apply (*subgoal-tac* (*1*::*nat*) = *Suc 0*)
apply (*erule ssubst*)
apply (*subgoal-tac 0 < k*)
apply (*simp add*: *Suc-pred*)
apply *auto*
done

lemma *binomial-n-n*: *n choose n = 1*
apply (*simp add*: *binomial-factorial-def*)
done

lemma *binomial-n-0*: *n choose 0 = 1*
apply (*simp add*: *binomial-0*)
done

lemma *binomial-sym*: *k <= n ==> n choose k = n choose* (*n−k*)
apply (*simp add*: *binomial-factorial-def-div*)
apply (*subgoal-tac n − (n − k) = k*)
apply (*erule ssubst*)
apply (*subgoal-tac fact k ∗ fact*(*n−k*) = *fact*(*n−k*) ∗ *fact k*)
apply (*erule ssubst*)
apply *auto*
done

lemma *binomial-minus*: *Suc k <= n ==> fact*(*n−k*) = (*n−k*) ∗ *fact*(*n − Suc k*)
apply (*subgoal-tac n−k = Suc* (*n − Suc k*))
apply (*erule ssubst*)
apply (*simp only*: *fact-Suc*)
apply (*simp add*: *Suc-diff-le* [*THEN sym*])
done

lemma *divide-mod-equality*: *a mod b* = (*0*::*nat*) ==> (*a div b*) ∗ *b = a*
apply *auto*
done

lemma *binomial-def*: *k<=n ==> 0 < k ==>* (*n choose k*) + (*n choose* (*k −* *Suc 0*)) = (*Suc n choose k*)
apply (*simp add*: *binomial-Suc*)
done

lemma *binomial-0*: *n choose Suc n = 0*
apply (*simp add*: *binomial-eq-0-iff*)
done

lemma *binomial-gt-0* [*simp*]: *k <= n ==> 0 < n choose k*
apply (*rule contrapos-pp*)
apply *auto*
apply (*simp add*: *binomial-eq-0-iff*)
done

lemma *binomial-theorem*: *(x+y)^n = setsum (%i. (n choose i)*x^i*y^(n−i)) {0..n}*
apply (*induct-tac n*)
apply *force*
apply (*simp add*: *power-Suc*)
apply (*simp add*: *add-mult-distrib*)
apply (*simp add*: *setsum-multiplier*)
apply (*subgoal-tac* (∑ *i*::*nat*:{*0*::*nat*..*n*}. *x * ((n choose i) * x ^ i * y ^ (n − i))) = (∑ *i*::*nat*:{*0*::*nat*..*n*}. *((n choose i) * x ^ Suc i * y ^ (n − i))))*
apply (*erule ssubst*)
apply (*erule ssubst*)
apply (*subgoal-tac* (∑ *i*::*nat*:{*0*::*nat*..*n*}. *y * ((n choose i) * x ^ i * y ^ (n − i))) = (∑ *i*::*nat*:{*0*::*nat*..*n*}. *((n choose i) * x ^ i * y ^ (Suc n − i))))*
apply (*erule ssubst*)
apply (*subgoal-tac* (∑ *i*::*nat*:{*0*::*nat*..*n*}. *(n choose i) * x ^ Suc i * y ^ (n − i)) = (∑ *i*::*nat*:{*1*::*nat*..*(n+1)*}. *(n choose (i−(1*::*nat))) * x ^ ((Suc i) − (1*::*nat)) * y ^ (Suc n − i)))*
apply (*erule ssubst*)
apply (*subgoal-tac* (∑ *i*::*nat*:{*0*::*nat*..*n*}. *(n choose i) * x ^ i * y ^ (Suc n − i)) = ((*n choose 0)*x^0*y^Suc n) + (∑ *i*::*nat*:{*1*::*nat*..*n+1*}. *(n choose i) * x ^ i * y ^ (Suc n − i)))*
apply (*erule ssubst*)
apply (*simp only*:*plus-ac0*)
apply (*simp add*: *setsum-addf* [*THEN sym*])
apply (*simp add*: *add-mult-distrib* [*THEN sym*])
apply (*subgoal-tac* (∑ *xa*::*nat*:{*Suc* (*0*::*nat*)..*Suc n*}. *(n choose xa + (n choose (xa − Suc (0*::*nat*)))) * x ^ xa * y ^ (Suc n − xa)) = (∑ *xa*::*nat*:{*Suc* (*0*::*nat*)..*Suc n*}. *(Suc n choose xa) * x ^ xa * y ^ (Suc n − xa)))*
apply (*erule ssubst*)
apply (*subgoal-tac y * y^n = (Suc n choose 0) * x^0 * y^(Suc n − 0)*)
apply (*erule ssubst*)
apply (*subgoal-tac 0 ~*: {*Suc 0..Suc n*})
apply (*subgoal-tac finite* {*Suc 0..Suc n*})
apply (*subgoal-tac* {*0..Suc n*} = *insert 0* {*Suc 0..Suc n*})

apply (*erule ssubst*)
apply (*subgoal-tac* ($\sum$ *i*::*nat*:*insert* (*0*::*nat*) {*Suc* (*0*::*nat*)..*Suc n*}. (*Suc n choose i*) * *x* ^ *i* * *y* ^ (*Suc n* − *i*)) = (*Suc n choose 0*) * *x* ^ *0* * *y* ^ (*Suc n* − *0*) + ($\sum$ *i*::*nat*:{*Suc* (*0*::*nat*)..*Suc n*}. (*Suc n choose i*) * *x* ^ *i* * *y* ^ (*Suc n* − *i*)))
apply (*erule ssubst*)
apply *force*
apply (*simp add*: *setsum-insert*)
apply *force*
apply *force*
apply *force*
apply *force*
apply (*rule setsum-cong*)
apply *force*
apply (*case-tac xa* = *Suc n*)
apply (*erule ssubst*)
apply (*simp add*: *binomial-0*)
apply (*subgoal-tac xa* <= *n*)
apply (*subgoal-tac 0* < *xa*)
apply (*simp add*: *binomial-def*)
apply *force*
apply *force*
apply (*subgoal-tac y* ^ *Suc n* = *y* ^ (*Suc n* − *0*))
apply (*erule ssubst*)
apply (*subgoal-tac 0* ~: {*1*::*nat*..*n*+*1*})
apply (*subgoal-tac finite* {*1*::*nat*..*n*+*1*})
apply (*subgoal-tac* (*n choose* (*0*::*nat*)) * *x* ^ (*0*::*nat*) * *y* ^ (*Suc n* − (*0*::*nat*)) + ($\sum$ *i*::*nat*:{*1*::*nat*..*n* + (*1*::*nat*)}. (*n choose i*) * *x* ^ *i* * *y* ^ (*Suc n* − *i*)) = ($\sum$ *i*::*nat*:{*0*::*nat*..*n* + (*1*::*nat*)}. (*n choose i*) * *x* ^ *i* * *y* ^ (*Suc n* − *i*)))
apply (*erule ssubst*)
apply (*subgoal-tac* {*0*::*nat*..*n*+*1*} = *insert* (*n*+*1*) {*0*..*n*})
apply (*erule ssubst*)
apply (*subgoal-tac* (*n*+*1*) ~: {*0*::*nat*..*n*})
apply (*subgoal-tac finite* {*0*::*nat*..*n*})
apply (*simp add*: *setsum-insert*)
apply (*simp add*: *binomial-0*)
apply *force*
apply *force*
apply *force*
apply (*subgoal-tac* {*0*::*nat*..*n* + (*1*::*nat*)} = *insert 0* {*1*::*nat*..*n*+*1*})
apply (*erule ssubst*)
apply (*simp add*: *setsum-insert*)
apply *force*
apply *force*
apply *force*
apply *force*

apply (*subgoal-tac* ($\sum$ *i*::*nat*:{*1*::*nat*..*n* + (*1*::*nat*)}. (*n choose* (*i* − (*1*::*nat*))) ∗ *x* ^ (*Suc i* − (*1*::*nat*)) ∗ *y* ^ (*Suc n* − *i*)) = ($\sum$ *i*::*nat*:{*1*::*nat*..*n* + (*1*::*nat*)}. (*n choose* (*i* − (*1*::*nat*))) ∗ *x* ^ (*Suc* (*i* − *1*::*nat*)) ∗ *y* ^ (*n* − (*i*−(*1*::*nat*)))))
apply (*erule ssubst*)
apply (*rule setsum-index-shift-right*)
apply (*rule setsum-cong*)
apply *force*
apply (*subgoal-tac Suc xa* − *1* = *xa*)
apply (*erule ssubst*)
apply (*subgoal-tac Suc* (*xa* − (*1*::*nat*)) = *xa*)
apply (*erule ssubst*)
apply (*subgoal-tac Suc n* − *xa* = *n* − (*xa* − (*1*::*nat*)))
apply (*erule ssubst*)
apply *simp*
apply *simp*
apply *simp*
apply (*rule Suc-pred*)
apply *force*
apply *force*
apply (*rule setsum-cong*)
apply *force*
apply *simp*
apply (*simp only*: *power-Suc* [*THEN sym*])
apply (*simp add*: *Suc-diff-le*)
apply (*rule setsum-cong*)
apply *force*
apply *simp*
done

38.8 Beginning of Chebyshev's theorem

lemma *binomial-unimodal-1*: *1* <= *n* ==> *1* <= *k* ==> *k* <= *n* ==> (*n choose* (*k* − *1*) < *n choose k*) = (*2*∗*k* < *n*+*1*)
proof−
assume *npos*: *1* <= *n*
assume *kpos*: *1* <= *k*
assume *kn*: *k* <= *n*
have *step1*: *k* ∗ (*n choose k*) = (*n* − (*k* − *1*))∗(*n choose* (*k* − *1*))
apply (*rule binomial-step*)
apply (*auto simp only*: *kpos kn*)
done
have *a0*: *0* < *k*
apply (*subgoal-tac 1* <= *k*)
apply *force*
apply (*simp only*: *kpos*)

```
    done
  have a1: 0 < (n choose k)
    apply (case-tac n choose k)
    apply (simp add: binomial-eq-0-iff)
    apply auto
    apply (subgoal-tac k <= n)
    apply force
    apply (simp add: kn)
    done
  have a2: 0 < n - (k - 1)
    apply auto
    apply (rule Suc-le-lessD)
    apply (subgoal-tac 0 < k)
    apply (simp add: Suc-pred)
    apply (auto simp add: kn a0)
    done
  have a3: 0 < (n choose (k - 1))
    apply (case-tac n choose (k - 1))
    apply (simp add: binomial-eq-0-iff)
    apply auto
    apply (subgoal-tac k <= n)
    apply (simp only: Suc-less-eq [of n k - Suc (0::nat), THEN sym])
    apply (subgoal-tac 0 < k)
    apply (simp add: Suc-pred)
    apply (simp add: a0)
    apply (simp add: kn)
    done
  from step1 a0 a1 a2 a3 show ?thesis
    apply (simp only: gt-lt-product-iff [of k n choose k n - (k - 1) n choose (k
- 1)])
    apply simp
    apply (simp add: nat-less-diff [of k Suc n k])
    apply auto
    done
  qed

lemma binomial-unimodal-Suc-1: 1 <= n ==> k < n ==> (n choose k) < (n
choose Suc k) = (2 * Suc k < n+1)
  apply (subgoal-tac 1 <= Suc k)
  apply (subgoal-tac Suc k <= n)
  apply (subgoal-tac n choose k = n choose (Suc k - 1))
  apply (erule ssubst)
  apply (simp only: binomial-unimodal-1 [of n Suc k])
  apply auto
done
```

lemma *binomial-unimodal-2*: *1 <= n ==> 1 <= k ==> k <= n ==> (n choose k < n choose (k − 1)) = (n+1 < 2∗k)*
proof−
assume *n*: *1 <= n*
assume *k*: *1 <= k*
assume *kn*: *k <= n*
have *step1*: *(n − (k − 1))∗(n choose (k − 1)) = k ∗ (n choose k)*
apply (*simp only*: *eq-sym-conv*)
apply (*rule binomial-step*)
apply (*auto simp only*: *k kn*)
done
have *a0*: *0 < k*
apply (*subgoal-tac 1 <= k*)
apply *force*
apply (*simp only*: *k*)
done
have *a1*: *0 < (n choose k)*
apply (*case-tac n choose k*)
apply (*simp add*: *binomial-eq-0-iff*)
apply *auto*
apply (*subgoal-tac k <= n*)
apply *force*
apply (*simp add*: *kn*)
done
have *a2*: *0 < n − (k − 1)*
apply *auto*
apply (*rule Suc-le-lessD*)
apply (*subgoal-tac 0 < k*)
apply (*simp add*: *Suc-pred*)
apply (*auto simp add*: *kn a0*)
done
have *a3*: *0 < (n choose (k − 1))*
apply (*case-tac n choose (k − 1)*)
apply (*simp add*: *binomial-eq-0-iff*)
apply *auto*
apply (*subgoal-tac k <= n*)
apply (*simp only*: *Suc-less-eq* [*of n k − Suc* (*0*::*nat*), *THEN sym*])
apply (*subgoal-tac 0 < k*)
apply (*simp add*: *Suc-pred*)
apply (*simp add*: *a0*)
apply (*simp add*: *kn*)
done
from *step1 a0 a1 a2 a3* **show** *?thesis*
apply (*simp only*: *gt-lt-product-iff* [*of n − (k − 1) n choose (k − 1) k n*

choose k])
apply *simp*
apply (*subgoal-tac* (*Suc n − k < k*) = (*k* + (*Suc n − k*) < *k* + *k*))
apply (*erule ssubst*)**back**
apply *force*
apply (*simp only*: *nat-add-left-cancel-less* [*THEN sym*])
done
qed

lemma *binomial-unimodal-Suc-2*: *1 <= n ==> k < n ==>* (*n choose Suc k*) < (*n choose k*) = (*n*+*1 < 2 ∗ Suc k*)
apply (*subgoal-tac 1 <= Suc k*)
apply (*subgoal-tac Suc k <= n*)
apply (*subgoal-tac n choose k = n choose* (*Suc k − 1*))
apply (*erule ssubst*)
apply (*simp only*: *binomial-unimodal-2* [*of n Suc k*])
apply *auto*
done

lemma *binomial-unimodal-3*: *1 <= n ==> 1 <= k ==> k <= n ==>* (*n choose k = n choose* (*k − 1*)) = (*2∗k = n*+*1*)
proof−
assume *n*: *1 <= n*
assume *k*: *1 <= k*
assume *kn*: *k <= n*
have *step1*: (*n − k − 1*))∗(*n choose* (*k − 1*)) = *k ∗* (*n choose k*)
apply (*simp only*: *eq-sym-conv*)
apply (*rule binomial-step*)
apply (*auto simp only*: *k kn*)
done
have *a0*: *0 < k*
apply (*subgoal-tac 1 <= k*)
apply *force*
apply (*simp only*: *k*)
done
have *a1*: *0 <* (*n choose k*)
apply (*case-tac n choose k*)
apply (*simp add*: *binomial-eq-0-iff*)
apply *auto*
apply (*subgoal-tac k <= n*)
apply *force*
apply (*simp add*: *kn*)
done
have *a2*: *0 < n − k − 1*)
apply *auto*

```
    apply (rule Suc-le-lessD)
    apply (subgoal-tac 0 < k)
    apply (simp add: Suc-pred)
    apply (auto simp add: kn a0)
    done
  have a3: 0 < (n choose (k - 1))
    apply (case-tac n choose (k - 1))
    apply (simp add: binomial-eq-0-iff)
    apply auto
    apply (subgoal-tac k <= n)
    apply (simp only: Suc-less-eq [of n k - Suc (0::nat), THEN sym])
    apply (subgoal-tac 0 < k)
    apply (simp add: Suc-pred)
    apply (simp add: a0)
    apply (simp add: kn)
    done
  from step1 a0 a1 a2 a3 show ?thesis
   apply (simp only: eq-product-iff [of n - (k - 1) n choose (k - 1) k n choose
k])
    apply simp
    apply (subst eq-sym-conv [of Suc n - k k])
    apply (simp only: nat-add-right-cancel [of k k Suc n - k, THEN sym])
    apply auto
    done
  qed

lemma binomial-unimodal-Suc-3: 1 <= n ==> k < n ==> (n choose Suc k) =
(n choose k) = (2 * Suc k = n+1)
 apply (subgoal-tac 1 <= Suc k)
 apply (subgoal-tac Suc k <= n)
 apply (subgoal-tac n choose k = n choose (Suc k - 1))
 apply (erule ssubst)
 apply (simp only: binomial-unimodal-3 [of n Suc k])
 apply auto
done



lemma binomial-two-level-equality: 1 <= n ==> (2*n + 2) choose (n+1) =
(2*n choose (n - 1)) + 2*(2*n choose n) + (2*n choose (n+1))
 apply (simp add: binomial-Suc)
done

lemma binomial-two-level-inequality: 1 <= n ==> 2*(2*n choose n) <= (2*n+2)
choose (n+1)
```

apply (*subgoal-tac* (*2∗n + 2*) *choose* (*n+1*) = (*2∗n choose* (*n − 1*)) + *2∗*(*2∗n choose n*) + (*2∗n choose* (*n+1*)))
apply *force*
apply (*simp only*: *binomial-two-level-equality*)
done

lemma *binomial-two-level-inequality-Suc*: *1 <= n ==> 2∗*(*2∗n choose n*) <= (*2∗Suc n choose Suc n*)
apply (*subgoal-tac 2∗Suc n = 2∗n+2*)
apply (*erule ssubst*)
apply (*subgoal-tac Suc n = n+1*)
apply (*erule ssubst*)
apply (*simp only*: *binomial-two-level-inequality*)
apply *auto*
done

lemma *binomial-always-ge-2* [*rule-format*]: *0 < n −−> 2 <=* (*2∗n choose n*)
apply (*induct-tac n*)
apply *auto*
apply (*simp add*: *binomial-Suc-Suc*)
done

lemma *extended-unimodality-lemma*: *1 <= n ==> ALL i < n.* (*2∗n choose i*) < (*2∗n choose Suc i*)
apply *auto*
apply (*subgoal-tac 2∗Suc i < 2∗n+1*)
apply (*simp add*: *binomial-unimodal-Suc-1*)
apply *auto*
done

lemma *extended-unimodality-1*: *1 <= n ==> k < n ==>* (*2∗n choose k*) < (*2∗n choose n*)
apply (*rule chained-inequalities-up* [*of n* (%*i.* (*2∗n choose i*)) *k*])
apply (*simp add*: *extended-unimodality-lemma*)
by *auto*

lemma *extended-unimodality-lemma-2*: *1 <= n ==> ALL i < 2∗n.* (*n <= i −−>* (*2∗n choose Suc i*) < (*2∗n choose i*))
apply *auto*
apply (*subgoal-tac 2∗n + 1 < 2 ∗ Suc i*)
apply (*simp add*: *binomial-unimodal-Suc-2*)
apply *force*
done

lemma *extended-unimodality-2*: *1 <= n ==> n < k ==>* (*2∗n choose k*) <

```
(2*n choose n)
  apply (case-tac k <= 2*n)
  apply (frule less-imp-Suc-add)
  apply auto
  apply (simp only: add-Suc-right [THEN sym])
  apply (subgoal-tac 2*n choose (n + Suc k) = 2*n choose (n - Suc k))
  apply (erule ssubst)
  apply (subgoal-tac (n - Suc k) < n)
  apply (simp add: extended-unimodality-1)
  apply force
  apply (simp add: binomial-sym)
  apply (subgoal-tac 2*n choose k = 0)
  apply (erule ssubst)
  apply (subgoal-tac 2 <= 2*n choose n)
  apply force
  apply (auto simp add: binomial-always-ge-2 binomial-eq-0-iff)
done

lemma middle-binomial-is-max: k <= 2*n ==> (2*n choose k) <= (2*n choose
n)
  apply (case-tac n = 0)
  apply force
  apply (subgoal-tac 1 <= n)
  apply (case-tac k < n)
  apply (frule extended-unimodality-1)
  apply force+
  apply (case-tac k = n)
  apply force+
  apply (subgoal-tac n < k)
  apply (frule extended-unimodality-2)
  apply force+
done

lemma binomial-theorem-for-2: (1+1)^n = setsum (%i. (n choose i)) {0..n}
  apply (simp only: binomial-theorem)
  apply (auto simp only: power-one)
  apply simp
done

lemma middle-binomial-bound-high: 0 < n ==> (2*n choose n) < 2^(2*n)
  proof-
    assume pos: 0 < n
    have step1: (2*n choose n) < setsum (%i. 2*n choose i) {0..2*n}
      apply (rule sum-greater-than-part)
      apply force+
```

```
    apply (subgoal-tac 2*n : {0..2*n})
    apply (subgoal-tac 2*n ~= n)
    apply force
    apply (auto simp add: pos)
    done
  also have ... = (1+1) ^(2*n)
    apply (simp only: binomial-theorem-for-2)
    done
  also have ... = 2^(2*n)
    apply (subgoal-tac 1+1 = 2)
    apply (erule ssubst)
    apply force
    apply force
    done
  finally show ?thesis.
  qed

lemma middle-binomial-bound-low: 1 <= n ==> 2^(2*n) <= 2*n*(2*n choose
n)
 proof-
  assume pos: 1 <= n
  have (2::nat)^(2*n) = (1+1)^(2*n)
    apply (subgoal-tac 1+1 = 2)
    apply (erule ssubst)
    apply auto
    done
  also have ... = setsum (%i. 2*n choose i) {0..2*n}
    apply (rule binomial-theorem-for-2)
    done
  also have ... = 1 + setsum (%i. 2*n choose i) {1..2*n}
    apply (subgoal-tac {0..2*n} = insert 0 {1..2*n})
    apply (simp add: setsum-insert)
    apply force
    done
  also have ... = 1 + 1 + setsum (%i. 2*n choose i) {1..(2*n - 1)}
    apply (subgoal-tac {1..2*n} = insert (2*n) {1..(2*n - 1)})
    apply (erule ssubst)
    apply (subgoal-tac finite {1::nat..(2::nat) * n - (1::nat)})
    apply (subgoal-tac (2*n) ~: {1..(2*n - 1)})
    apply (simp add: setsum-insert)
    apply (simp add: pos)
    apply auto
    apply (simp add: Suc-le-mono [of 2*n 2*n - Suc 0, THEN sym])
    apply (simp add: not-le-iff-less)
    apply (subgoal-tac Suc (2*n - Suc 0) < Suc x)
```

apply (*simp only*: *Suc-pred*)
apply (*simp only*: *Suc-less-eq*)
apply (*subgoal-tac 1 <= n*)
apply *force*
apply (*simp only*: *pos*)
apply (*subgoal-tac Suc x <= Suc (2*n − Suc 0)*)
apply (*subgoal-tac 0 < 2*n*)
apply (*simp add*: *Suc-pred*)
apply (*simp add*: *pos*)
apply (*subgoal-tac 1 <= n*)
apply *force*
apply (*rule pos*)
apply *simp*
done
also have ... = *2 + setsum (%i. 2*n choose i) {1..(2*n − 1)}*
by *auto*
also have ... <= *2 + (2*n − 1)*(2*n choose n)*
apply *simp*
apply (*subgoal-tac ((2::nat) * n − Suc (0::nat))*(2*n choose n) = card({Suc (0::nat)..(2::nat) * n − Suc (0::nat)}) *(2*n choose n)*)
apply (*erule ssubst*)
apply (*rule setsum-constant-bound* [*of* {*Suc (0::nat)..(2::nat) * n − Suc (0::nat)*} (*%i. 2*n choose i*) *2*n choose n*])
apply *force*
apply (*auto simp add*: *middle-binomial-is-max*)
apply (*subgoal-tac x <= 2*n*)
apply (*auto simp add*: *middle-binomial-is-max*)
apply (*rule nat-le-diff*)
apply *force*
done
also have ... <= *(2*n choose n) + (2*n − 1)*(2*n choose n)*
apply (*auto*)
apply (*rule binomial-always-ge-2*)
apply (*subgoal-tac 1 <= n*)
apply *force*
apply (*intro pos*)
done
also have ... = *1*(2*n choose n) + (2*n − 1) * (2*n choose n)*
apply *auto*
done
also have ... = *(2*n)*(2*n choose n)*
apply (*simp only*: *add-mult-distrib* [*THEN sym*])
apply (*subgoal-tac 1 + (2*n − 1) = 2*n*)
apply (*erule ssubst*)
apply *simp*

apply (*rule le-add-diff-inverse*)
apply (*subgoal-tac 1 <= n*)
apply *force*
apply (*simp only*: *pos*)
done
finally show *?thesis*.
qed

lemma *extended-unimodality-lemma-odd*: *1 <= n ==> ALL i < n. (Suc(2*n) choose i) < (Suc(2*n) choose Suc i)*
apply (*auto*)
apply (*subgoal-tac 2*Suc i < Suc(2*n) + 1*)
apply (*simp only*: *binomial-unimodal-Suc-1* [*THEN sym*])
apply *auto*
done

lemma *extended-unimodality-odd-1*: *1 <= n ==> k < n ==> (Suc(2*n) choose k) < (Suc(2*n) choose n)*
apply (*rule chained-inequalities-up* [*of n* (%*i*. *Suc(2*n) choose i*) *k*])
apply (*simp add*: *extended-unimodality-lemma-odd*)
apply *force*
done

lemma *extended-unimodality-odd-2*: *1 <= n ==> Suc n < k ==> (Suc(2*n) choose k) < (Suc(2*n) choose Suc n)*
apply (*case-tac Suc(2*n) < k*)
apply (*subgoal-tac Suc(2*n) choose k = 0*)
apply (*erule ssubst*)
apply (*simp add*: *binomial-Suc-Suc*)
apply (*subgoal-tac 2 <= 2*n choose n*)
apply (*simp add*: *binomial-eq-0-iff*)
apply (*simp add*: *binomial-always-ge-2*)
apply (*subgoal-tac k <= Suc(2*n)*)
apply (*subgoal-tac (Suc(2*n) choose k) = (Suc(2*n) choose (Suc(2*n) - k))*)
apply (*erule ssubst*)
apply (*subgoal-tac (Suc(2*n) choose Suc n) = (Suc(2*n) choose n)*)
apply (*erule ssubst*)
prefer *4*
apply *force*
apply (*subgoal-tac (Suc(2*n) - k) < n*)
apply (*simp add*: *extended-unimodality-odd-1*)
apply (*simp only*: *nat-add-left-cancel-less* [*of k Suc(2*n) - k n*, *THEN sym*])
apply (*simp add*: *le-add-diff-inverse*)
apply (*subgoal-tac Suc(2*n) choose n = Suc(2*n) choose (Suc(2*n) - Suc n)*)
apply (*erule ssubst*)

apply (*subst binomial-sym*)
apply *force+*
apply (*simp only*: *binomial-sym*)
done

lemma *middle-binomial-is-max-odd*: *1 <= n ==> (Suc(2∗n) choose k) <= (Suc(2∗n) choose n)*
apply (*case-tac Suc(2∗n) < k*)
apply (*subgoal-tac Suc(2∗n) choose k = 0*)
apply (*subgoal-tac Suc(2∗n) choose n = Suc(2∗n) choose Suc n*)
apply (*erule ssubst*)+
apply (*simp add*: *binomial-Suc-Suc*)
apply (*subgoal-tac 2 <= 2∗n choose n*)
apply (*simp only*: *binomial-sym*)
apply *simp*
apply (*simp add*: *binomial-always-ge-2*)
apply (*simp add*: *binomial-eq-0-iff*)
apply (*case-tac k < n*)
apply (*subgoal-tac Suc ((2*::nat*) ∗ n) choose k < Suc ((2*::nat*) ∗ n) choose n*)
apply *simp*
apply (*simp add*: *extended-unimodality-odd-1*)
apply (*case-tac Suc n < k*)
apply (*subgoal-tac Suc (2∗n) choose n = Suc (2∗n) choose (Suc n)*)
apply (*erule ssubst*)
apply (*subgoal-tac Suc ((2*::nat*) ∗ n) choose k < Suc ((2*::nat*) ∗ n) choose Suc n*)
apply *simp*
apply (*force intro*: *extended-unimodality-odd-2*)
apply (*simp only*: *binomial-sym*, *simp*)
apply *auto*
apply (*case-tac k = n*)
apply (*erule ssubst*)
apply *force*
apply (*case-tac k = Suc n*)
apply (*erule ssubst*)
apply (*subgoal-tac Suc ((2*::nat*) ∗ n) choose Suc n = Suc ((2*::nat*) ∗ n) choose n*)
apply *force*
apply (*simp only*: *binomial-sym*, *simp*)
apply *force*
done

lemma *middle-odd-binomial*: *(Suc(2∗m) choose m) = (Suc(2∗m) choose Suc m)*
apply (*subgoal-tac Suc(2∗m) − m = Suc m*)
apply (*simp only*: *binomial-sym*)

apply *auto*
done

lemma *binomial-sum-lemma-1*: *1 <= m ==> 2*(Suc(2*m) choose m) < 2^(Suc(2*m))*
proof−
assume *m*: *1 <= m*
have *2*(Suc(2*m) choose m) = (Suc(2*m) choose m) + (Suc(2*m) choose Suc m)*
apply (*auto simp add*: *middle-odd-binomial*)
done
also have ... = *setsum* (%*i*. (*Suc*(*2*∗*m*) *choose i*)) {*m*,*Suc m*}
apply *auto*
done
also have ... < *setsum* (%*i*. (*Suc*(*2*∗*m*) *choose i*)) {*0*..*Suc*(*2*∗*m*)}
apply (*rule sum-is-greater-than-part*)
apply *force*
apply *force*
apply *clarsimp*
apply (*case-tac 0 = Suc*(*2*∗*m*) *choose x*)
apply (*simp add*: *binomial-eq-0-iff*)
apply *auto*
apply (*subgoal-tac 0* : {*0*..*Suc*(*2*∗*m*)})
apply (*subgoal-tac 0* ~: {*m*,*Suc m*})
apply *force*
apply (*simp*)
apply (*subgoal-tac 1 <= m*)
apply *force*
apply (*simp only*: *m*)
apply *force*
apply (*subgoal-tac 0* : {*0*..*Suc*(*2*∗*m*)})
apply (*subgoal-tac 0* ~: {*m*,*Suc m*})
apply *force*
apply *simp*
apply (*subgoal-tac 1 <= m*)
apply *force*
apply (*simp only*: *m*)
apply *force*
done
also have ... = (*1*+*1*)^(*Suc*(*2*∗*m*))
apply (*simp only*: *binomial-theorem-for-2*)
done
also have ... = *2*^(*Suc*(*2*∗*m*))
apply (*subgoal-tac 1*+*1* = (*2*::*nat*))
apply (*erule ssubst*)
apply *auto*

```
    done
  finally show ?thesis.
  qed

lemma binomial-sum-lemma-2: 1 <= m ==> (Suc(2*m) choose m) < 2^(2*m)
 proof−
  assume m: 1 <= m
  have step1: 2*(Suc(2*m) choose m) < 2^(Suc(2*m))
   apply (rule binomial-sum-lemma-1)
   apply (rule m)
   done
  then have (Suc(2*m) choose m) < 2^(2*m)
   apply auto
   done
  then show ?thesis.
  qed

lemma binomial-sum-lemma-3: 1 <= m ==> Suc(2*m) choose Suc m < 2^(2*m)
 apply (subgoal-tac Suc(2*m) choose Suc m = Suc(2*m) choose m)
 apply (erule ssubst)
 apply (simp only: binomial-sum-lemma-2)
 apply (simp only: middle-odd-binomial)
 done

lemma binomial-sum-lemma-4: 1 <= m ==> Suc(2*m) choose m < 4^m
 proof−
  assume m: 1 <= m
  have Suc(2*m) choose m < 2^(2*m)
   apply (rule binomial-sum-lemma-2)
   apply (rule m)
   done
  also have ... = 4^m
   apply (simp add: power-mult)
   done
  finally show ?thesis.
  qed


lemma binomial-factorial-simplified: k <= n ==> (n choose k) = setprod id {(Suc
k)..n} div setprod id {1..(n−k)}
 apply (subst binomial-factorial-def-div)
 apply force
 apply (subst div-mult2-eq)
 apply (subst factorial-div)
 apply force
```

apply (*simp only*: *factorial-setprod-def*)
done

lemma *binomial-odd-def*: *1 <= m ==> (Suc(2∗m) choose m) = setprod id {m+2..Suc(2∗m)} div setprod id {1..m}*
apply (*subgoal-tac (Suc(2∗m) choose m) = (Suc(2∗m) choose Suc m)*)
apply (*erule ssubst*)
apply (*subst binomial-factorial-simplified*)
apply *force*
apply (*subgoal-tac m + 2 = Suc (Suc m)*)
apply (*erule ssubst*)
apply (*subgoal-tac Suc(2∗m) − Suc m = m*)
apply (*erule ssubst*)
apply *force+*
apply (*simp only*: *binomial-sym*)
apply *force*
done

lemma *binomial-setprod-multiply*: *k <= n ==> (setprod id {1..k})∗(setprod id {1..(n−k)})∗(n choose k) = setprod id {1..n}*
apply (*simp only*: *factorial-setprod-def* [*THEN sym*])
apply (*simp only*: *binomial-factorial-def*)
done

lemma *binomial-setprod-multiply-reduced*: *k <= n ==> (setprod id {1..(n−k)})∗(n choose k) = setprod id {Suc(k)..n}*
apply (*subgoal-tac setprod id {1..k} ∗ (setprod id {1::nat..n − k} ∗ (n choose k)) = setprod id {1..k} ∗ setprod id {Suc k..n}*)
apply (*simp only*: *mult-cancel1*)
apply (*subgoal-tac 0 < setprod id {Suc 0..k}*)
apply *simp*
apply *simp*
apply (*subst nat-mult-assoc* [*THEN sym*])
apply (*simp only*: *binomial-setprod-multiply*)
apply (*simp only*: *setprod-interval-multiply*)
done

lemma *binomial-setprod-mod* [*simp*]: *setprod id {Suc k..n} mod setprod id {1..(n−k)} = 0*
apply (*case-tac n < k*)
apply *simp*
apply (*subgoal-tac k <= n*)
apply (*simp only*: *mod-eq-0-iff*)
apply (*rule-tac x = (n choose k)* **in** *exI*)
apply (*rule sym*)

apply (*simp only*: *binomial-setprod-multiply-reduced*)
apply *force*
done

lemma *binomial-setprod-def*: *k* <= *n* ==> (*n choose k*) = *setprod id* {*1*..*n*} *div* ((*setprod id* {*1*..*k*})∗(*setprod id* {*1*..(*n*−*k*)}))
apply (*rule div-multiply*)
apply *force*
apply *force*
apply (*rule sym*)
apply (*subgoal-tac* (*n choose k*) ∗ (*setprod id* {*1*::*nat*..*k*} ∗ *setprod id* {*1*::*nat*..*n* − *k*}) = (*setprod id* {*1*..*k*})∗(*setprod id* {*1*..(*n*−*k*)})∗(*n choose k*))
apply (*erule ssubst*)
apply (*simp only*: *binomial-setprod-multiply div-multiply*)
apply *force*
done

lemma *binomial-eq*: *a* + *b* = *n* ==> (*n choose a*) = (*n choose b*)
apply (*subgoal-tac b* = *n* − *a*)
apply (*erule ssubst*)**back**
apply (*rule binomial-sym*)
apply *force*+
done

end

theory *Chebyshev3* = *Chebyshev2* + *BigO*:

38.9 theta(x) = O(x) and psi(x) = O(x)

lemma *prime-dvd-binomial*: *m* + *2* <= *p* ==> *p* <= *Suc*(*2*∗*m*) ==> *1* <= *m* ==> *p* : *prime* ==> *p dvd* (*Suc*(*2*∗*m*) *choose m*)
apply (*simp only*: *binomial-odd-def*)
apply (*rule dvd-div*)
apply *force*
apply (*simp only*: *prime-divides-product*)
apply (*subst factorial-setprod-def* [*THEN sym*])
apply (*simp add*: *prime-not-divide-less*)
apply (*subgoal-tac m*+*2* = *Suc*(*m*+*1*))
apply (*erule ssubst*)
apply (*subgoal-tac setprod id* {*1*..*m*} = *setprod id* {*1*..(*Suc*(*2*∗*m*) − (*m*+*1*))})
apply (*erule ssubst*)
apply (*simp only*: *binomial-setprod-mod*)

apply *force+*
done

lemma *product-dvd-M*: *1 <= m ==> setprod id {p. p: prime & m+2 <= p & p <= Suc(2*m)} dvd (Suc(2*m) choose m)*
apply (*rule prime-product-dvd*)
apply (*subgoal-tac {p::nat. p : prime & m + (2::nat) <= p & p <= Suc ((2::nat) * m)} <= {1..Suc(2*m)}*)
apply (*simp add*: *finite-subset*)
apply (*auto simp add*: *prime-dvd-binomial*)
done

lemma *product-lt-M*: *1 <= m ==> setprod id {p. p: prime & m+2 <= p & p <= Suc(2*m)} <= (Suc(2*m) choose m)*
apply (*rule dvd-imp-le*)
apply (*simp only*: *product-dvd-M*)
apply *simp*
done

lemma *product-lt-4-to-m*: *1 <= m ==> setprod id {p. p: prime & m+2 <= p & p <= Suc(2*m)} < (4::nat)^m*
apply (*rule le-less-trans* [*of setprod id {p. p: prime & m+2 <= p & p <= Suc(2*m)} (Suc(2*m) choose m) 4^m*])
apply (*simp only*: *product-lt-M*)
apply (*simp only*: *binomial-sum-lemma-4*)
done

lemma *product-primes-induction-step*: *ALL m < n. (setprod id {p. p : prime & p <= (Suc m)} < 4^(Suc m))*
==> setprod id {p. p : prime & p <= Suc n} < 4^(Suc n)
apply (*case-tac n = 0*)
apply (*erule ssubst*)
apply (*subgoal-tac {p::nat. p : prime & p <= Suc 0} = {}*)
apply (*erule ssubst*)
apply *force+*
apply *clarsimp*
apply (*subgoal-tac 2 <= x*)
apply *force*
apply (*simp add*: *primes-always-ge-2*)
apply (*case-tac n = 1*)
apply (*erule ssubst*)
apply (*subgoal-tac {p::nat. p : prime & p <= Suc 1} = {2}*)
apply (*erule ssubst*)
apply *force*
apply *clarsimp*

apply *auto*
apply (*subgoal-tac 2 <= x*)
apply (*force intro*: *le-anti-sym*)
apply (*force intro*: *primes-always-ge-2*)
apply (*simp add*: *two-is-prime*)
apply (*subgoal-tac 2 <= n*)
apply *auto*
apply (*case-tac even* (*Suc n*))
apply (*subgoal-tac Suc n ~*: *prime*)
apply (*subgoal-tac* {*p*::*nat*. *p* : *prime* & *p <= Suc n*} = {*p*::*nat*. *p* : *prime* & *p <= Suc* (*n − 1*)})
apply (*erule ssubst*)
apply (*frule-tac x = n − 1* **in** *spec*)
apply (*frule mp*)
apply *force*
apply (*subgoal-tac 4 ∗ 4^(n − 1) <= 4 ∗ 4 ^ n*)
apply (*erule order-less-le-trans*)
apply *assumption+*
apply (*rule mult-left-mono*)
apply (*rule power-increasing*)
apply *force+*
apply *auto*
apply (*case-tac x = Suc n*)
apply (*simp add*: *lt-one-minus*)
apply (*simp add*: *lt-one-minus*)
apply (*frule only-even-prime*)
apply *simp*
apply *arith*

apply (*subgoal-tac EX y. n = Suc*(*Suc 0*) ∗ *y*)
apply (*erule exE*)
apply (*subgoal-tac Suc y <= n*)
apply (*subgoal-tac 1 <= y*)
apply (*erule ssubst*)

apply (*subgoal-tac* {*p*::*nat*. *p* : *prime* & *p <= Suc*(*Suc* (*Suc 0*) ∗ *y*)} =
{*p*::*nat*. *p* : *prime* & *p <= Suc y*} *Un* {*p*::*nat*. *p* : *prime* & (*y*+*2*) <= *p* & *p <= Suc*(*2*∗*y*)})
apply (*erule ssubst*)
apply (*subst setprod-Un-disjoint*)
apply (*subgoal-tac* {*p*::*nat*. *p* : *prime* & *p <= Suc y*} <= {*0*..*Suc y*})
apply (*simp add*: *finite-subset*)
apply *force+*
apply (*subgoal-tac* {*p*::*nat*. *p* : *prime* & *y* + (*2*::*nat*) <= *p* & *p <= Suc* ((*2*::*nat*)

```
* y)} <= {0..Suc(2*y)})
  apply (simp add: finite-subset)
  apply force+
  apply (frule-tac x = y in spec)
  apply (frule mp)
  apply force
  apply (subgoal-tac setprod id {p::nat. p : prime & y + (2::nat) <= p & p <=
Suc ((2::nat) * y)} < 4^y)
  apply (subgoal-tac 0 < (4::nat) ^ Suc y)
  apply (subgoal-tac 0 <= setprod id {p::nat. p : prime & y + (2::nat) <= p &
p <= Suc ((2::nat) * y)})
  apply (subgoal-tac 4 * 4^(Suc (Suc 0) * y) = 4^Suc y * 4^y)
  apply (erule ssubst)
  apply (rule mult-strict-mono)
  apply (subst power-Suc)
  apply assumption+
  apply auto
  apply (subst power-add [THEN sym])
  apply force
  apply (subgoal-tac Suc (Suc y) = y+2)
  apply (erule ssubst)
  apply (simp only: product-lt-4-to-m)
  apply force+
  apply (simp only: even-nat-equiv-def2)
  apply auto
  done

lemma product-primes-lt-4-to-n-aux: setprod id {p. p: prime & p <= Suc n} <
4^(Suc n)
  apply (rule nat-less-induct [of - n])
  apply (erule product-primes-induction-step)
done

lemma product-primes-lt-4-to-n [rule-format]: 1 <= n ==> setprod id {p. p :
prime & p <= n} < 4^n
  apply (subgoal-tac n = Suc (n - 1))
  apply (erule ssubst)
  apply (rule product-primes-lt-4-to-n-aux)
  apply simp
done

lemma theta-bound: 1 <= n ==> theta n < real(n) * ln(4)
  proof-
    assume n: 1 <= n
    have theta n = ln (real (setprod id {p::nat. p : prime & p <= n}))
```

```
    apply (rule theta-def-2)
    done
  also have step1: ... < ln (real ((4::nat) ^ n))
    apply (subgoal-tac 0 < real (setprod id {p::nat. p : prime & p <= n}))
    apply (subgoal-tac 0 < real ((4::nat) ^ n))
    apply (simp only:  ln-less-cancel-iff)
    apply (simp only: real-of-nat-less-iff)
    apply (rule product-primes-lt-4-to-n)
    apply (rule n)
    apply (subgoal-tac 0 < (4::nat) ^ n)
    apply (simp only: real-of-nat-less-iff [THEN sym])
    apply force
    apply (subgoal-tac 0 < setprod id {p::nat. p : prime & p <= n})
    apply (simp only: real-of-nat-less-iff [THEN sym])
    apply (case-tac n = 0)
    apply (subgoal-tac 1 <= n)
    apply force
    apply (rule n)
    apply (case-tac n = 1)
    apply (simp)
    apply (subgoal-tac {p::nat. p : prime & p <= Suc (0::nat)} = {})
    apply (erule ssubst)back
    apply force
    apply auto
    apply (subgoal-tac 2 <= x)
    apply force
    apply (erule primes-always-ge-2)
    apply (subgoal-tac 2 <= n)
    apply (subst setprod-gt-0-iff)
    apply (auto simp add: finite-subset)
    apply (rule finite-subset [of - {0..n}])
    apply force+
    apply (auto simp add: prime-def)
    done
  also have ... = real n * ln(4::real)
    apply (subst realpow-real-of-nat [THEN sym])
    apply (subst ln-realpow)
    apply force
    apply force
    done
  finally show ?thesis.
  qed

lemma theta-bound2: theta n <= ln 4 * real n
  apply (case-tac n = 0)
```

apply (*simp add*: *theta-zero*)
apply (*subst mult-commute*)
apply (*rule order-less-imp-le*)
apply (*rule theta-bound*)
apply *auto*
done

lemma *theta-bigo*: *theta* =*o* *O*(%*x*. *real x*)
apply (*rule bigo-bounded-alt*)
apply (*rule allI*)
apply (*rule theta-geq-zero*)
apply (*rule allI*)
apply (*rule theta-bound2*)
done

lemma *psi-bigo-aux*:
(%*x*. *real*((*x*::*nat*) + *1*) *powr* (*1*/*2*) ∗ *ln*(*real*(*x* + *1*)) ∗ *ln*(*real*(*x* + *1*))) =*o*
O(%*x*. *real* (*x* + *1*))
apply (*rule bigo-bounded-alt*)
apply (*rule allI*)
apply (*rule nonneg-times-nonneg*)+
apply (*rule order-less-imp-le*)
apply *force*+
apply (*rule allI*)
apply (*subgoal-tac real* (*x* + *1*) *powr* (*1* / *2*) ∗ *ln* (*real* (*x* + *1*)) ∗
ln (*real* (*x* + *1*)) <=
(*real* (*x* + *1*) *powr* (*1* / *2*)) ∗ ((*real* (*x* + *1*) *powr* (*1* / *4*)) / (*1* / *4*)) ∗
((*real* (*x* + *1*) *powr* (*1* / *4*)) / (*1* / *4*)))
apply (*simp add*: *powr-add* [*THEN sym*])
apply (*rule mult-mono*)+
apply *force*
apply (*rule ln-powr-bound*)
apply *force*+
apply (*rule ln-powr-bound*)
apply *force*+
apply (*rule nonneg-times-nonneg*)
apply *force*+
done

lemma *psi-bigo-aux2*: (%*x*. *psi* (*x* + *1*)) =*o* *O*(%*x*. *real* (*x* + *1*))
proof −
have (%*x*. *psi* (*x* + *1*)) =*o* *O*((%*x*. *theta*(*x* + *1*)) +
(%*x*. *real*(*x* + *1*) *powr* (*1*/*2*) ∗ *ln*(*real*(*x* + *1*)) ∗ *ln*(*real*(*x* + *1*)) / *ln* *2*))
apply (*rule bigo-bounded*)
apply (*rule allI*)

```
    apply (rule psi-ge-zero)
    apply (rule allI)
    apply (simp add: func-plus)
    apply (rule psi-theta-bound-for-real)
    apply force
    done
  also have ... <= O(%x. real(x + 1)) + O(%x. real(x + 1))
    apply (rule subset-trans)
    apply (rule bigo-plus-subset)
    apply (rule set-plus-mono2)
    apply (rule bigo-elt-subset)
    apply (rule bigo-compose1 [OF theta-bigo])
    apply (rule bigo-elt-subset)
    apply (rule subsetD)
    apply (subgoal-tac (%x. 1 / ln 2) *o O(%x. real(x + 1)) <=
      O(%x. real(x + 1)))
    apply assumption
    apply (rule bigo-const-mult6)
    apply (subgoal-tac (%x. real (x + 1) powr (1 / 2) *
      ln (real (x + 1)) * ln (real (x + 1)) / ln 2) = (%x. 1 / ln 2) *
       (%x. real (x + 1) powr (1 / 2) * ln (real (x + 1)) *
      ln (real (x + 1))))
    apply (erule ssubst)
    apply (rule set-times-intro2)
    apply (rule psi-bigo-aux)
    apply (simp add: func-times)
    done
  also have ... = O(%x. real(x + 1))
    by simp
  finally show ?thesis.
qed

lemma psi-bigo: psi =o O(%x. real x)
  apply (rule bigo-fix2)
  apply (rule psi-bigo-aux2)
  apply (rule psi-zero)
done

lemma theta-le-psi: theta x <= psi x
  apply (unfold theta-def)
  apply (unfold psi-def)
  apply (rule sumr-le-cong)
  apply (unfold lprime-def)
  apply (case-tac y : prime)
  apply simp
```

apply (*subgoal-tac Lambda y = Lambda (y^1)*)
apply (*erule ssubst*)
apply (*subst Lambda-eq*)
apply *auto*
apply (*rule Lambda-ge-zero*)
done

lemma *psi-theta-lim1*: (%*x*. *psi* (*x* + *1*)) =*o* (%*x*. *theta*(*x* + *1*)) +*o*
O(%*x*. *real*(*x* + *1*) *powr* (*1*/*2*) ∗ *ln*(*real*(*x* + *1*)) ∗ *ln*(*real*(*x* + *1*)) / *ln 2*)
apply (*rule set-minus-imp-plus*)
apply (*subst func-diff*)
apply (*rule bigo-bounded*)
apply (*rule allI*)
apply *simp*
apply (*rule theta-le-psi*)
apply (*rule allI*)
apply (*simp only*: *compare-rls*)
apply (*subst add-commute*)
apply (*rule psi-theta-bound-for-real*)
apply *force*
done

lemma *psi-theta-lim2*:
O(%*x*::*nat*. *real*(*x* + *1*) *powr* (*1*/*2*) ∗ *ln*(*real*(*x* + *1*)) ∗ *ln*(*real*(*x* + *1*))
/ *ln 2*) <= *O*(%*x*. *real*(*x* + *1*) *powr* (*3* / *4*))
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded-alt*)
apply (*rule allI*)
apply (*rule real-ge-zero-div-gt-zero*)
apply (*rule nonneg-times-nonneg*)+
apply (*rule order-less-imp-le*)
apply *force*+
apply (*rule allI*)
apply (*subgoal-tac real* (*x* + *1*) *powr* (*1* / *2*) ∗ *ln* (*real* (*x* + *1*)) ∗
ln (*real* (*x* + *1*)) / *ln 2* <=
(*real* (*x* + *1*) *powr* (*1* / *2*)) ∗ ((*real* (*x* + *1*) *powr* (*1* / *8*)) / (*1* / *8*)) ∗
((*real* (*x* + *1*) *powr* (*1* / *8*)) / (*1* / *8*)) / *ln 2*)
apply (*simp add*: *powr-add* [*THEN sym*])
apply (*subgoal-tac 64* ∗ *real* (*Suc x*) *powr* (*1* / *2* + *1* / *4*) / *ln 2* =
(*64* / *ln 2*) ∗ *real* (*Suc x*) *powr* (*3* / *4*))
apply (*erule subst*)
apply *assumption*
apply (*subgoal-tac* (*1*::*real*) / *2* + *1* / *4* = *3* / *4*)
apply (*erule ssubst*)
apply *simp*

```
  apply simp
  apply (rule divide-right-mono)
  apply (rule mult-mono)+
  apply force
  apply (rule ln-powr-bound)
  apply force+
  apply (rule ln-powr-bound)
  apply force+
  apply (rule nonneg-times-nonneg)
  apply force+
done

lemma psi-theta-lim3: (%x. psi (x + 1) / real (x + 1)) =o
  (%x. theta (x + 1) / real(x + 1)) +o O(%x. real(x + 1) powr (- (1 / 4)))
proof -
  have (%x. psi(x + 1) / real (x + 1)) =
      (%x. 1 / (real (x + 1))) * (%x. psi(x + 1))
    by (simp add: func-times)
  also have ... =o (%x. 1 / (real (x + 1))) *o ((%x. theta (x + 1)) +o
      O(%x. real(x + 1) powr (3 / 4)))
    apply (rule set-times-intro2)
    apply (rule subsetD)
    prefer 2
    apply (rule psi-theta-lim1)
    apply (rule set-plus-mono)
    apply (rule psi-theta-lim2)
    done
  also have ... = (%x. theta (x + 1) / real (x + 1)) +o
      (%x. 1 / (real (x + 1))) *o O(%x. real(x + 1) powr (3 / 4))
    by (simp add: set-times-plus-distribs func-times)
  also have ... <= (%x. theta (x + 1) / real (x + 1)) +o
      O(%x. real(x + 1) powr (- (1 / 4)))
    apply (rule set-plus-mono)
    apply (rule subset-trans)
    apply (rule bigo-mult2)
    apply (simp add: func-times)
    apply (subgoal-tac (%x. real (Suc x) powr (3 / 4) / real (Suc x))
      = (%x. real (Suc x) powr - (1 / 4)))
    apply (erule ssubst)
    apply (rule order-refl)
    apply (rule ext)
    apply (subst nonzero-divide-eq-eq)
    apply force
    apply (subgoal-tac 3 / 4 = -(1 / 4) + 1)
    apply (erule ssubst)
```

apply (*subst powr-add*)
apply *simp*
apply *simp*
done
finally show *?thesis*.
qed

lemma *psi-theta-lim4*: (%*x*. *psi x* / *real x*) ----> *1* ==>
(%*x*. *theta x* / *real x*) ----> *1*
apply (*subgoal-tac* (%*x*. *theta* (*x* + *1*) / (*real* (*x* + *1*))) ----> *1*)
apply (*erule LIMSEQ-offset*)
apply (*rule bigo-LIMSEQ2*)
apply (*rule psi-theta-lim3*)
apply (*rule LIMSEQ-ignore-initial-segment*)
apply (*rule LIMSEQ-neg-powr*)
apply *force*
apply (*erule LIMSEQ-ignore-initial-segment*)
done

38.10 theta and pi

lemma *theta-pi-rel1*: *1* <= *x* ==> *0* < (*d*::*real*) ==> *d* < *1* ==>
(*1* − *d*)∗*pi*(*x*) ∗ *ln*(*real*(*x*)) − *real*(*x*) *powr* (*1* − *d*) ∗ *ln*(*real*(*x*)) <= *theta x*
proof−
assume *1* <= *x* **and** *0* < *d* **and** *d* < *1*
have (*1* − *d*)∗*pi*(*x*) ∗ *ln*(*real*(*x*)) − *real*(*x*) *powr* (*1* − *d*) ∗ *ln*(*real*(*x*)) <= (*1* − *d*)∗*pi*(*x*) ∗ *ln*(*real*(*x*)) − (*1* − *d*)∗*real*(*x*) *powr* (*1* − *d*) ∗ *ln*(*real*(*x*))
apply *auto*
apply (*subst times-ac1-one-right* [*THEN sym*])
apply (*subst mult-commute*)
apply (*subst mult-assoc*)
apply (*rule mult-mono*)
apply (*insert prems*, *arith*)
apply *force*+
apply (*subst zero-le-mult-iff*)
apply (*rule disjI1*)
apply *auto*
apply (*insert powr-gt-zero* [*of real x* ((*1*::*real*) − *d*)])
done
also have ... <= (*1* − *d*)∗*pi*(*x*) ∗ *ln*(*real*(*x*)) − (*1* − *d*)∗*real*(*nat*(*floor*(*real*(*x*) *powr* (*1* − *d*)))) ∗ *ln*(*real*(*x*))
apply *auto*
apply (*rule mult-right-mono*)
apply (*rule mult-left-mono*)

apply (*rule real-nat-floor*)
apply (*rule order-less-imp-le*)
apply (*insert prems*, *auto*)
done
also have ... <= ((*1*::*real*) − *d*) ∗ *pi x* ∗ *ln* (*real x*) −
((*1*::*real*) − *d*) ∗ *pi* (*nat* (*floor* (*real x powr* ((*1*::*real*) − *d*)))) ∗ *ln* (*real x*)
apply *auto*
apply (*rule mult-right-mono*)
apply (*rule mult-left-mono*)
apply (*rule pi-less*)
apply (*insert prems*, *auto*)
done
also have ... = (*1* − *d*)∗(*pi*(*x*) − *pi*(*nat*(*floor*(*real x powr* (*1* − *d*)))))∗*ln* (*real x*)
apply (*subst right-diff-distrib*)
apply (*subst left-diff-distrib*)
apply *simp*
done
also have ... = *setsum* (%*y*. (*1* − *d*)∗*ln* (*real x*)) {*z*::*nat*. *nat*(*floor*(*real x powr* (*1* − *d*))) < *z* & *z* <= *x* & *z* : *prime*}
apply (*subst pi-diff*)
apply *auto*
apply (*subst real-of-nat-le-iff* [*THEN sym*])
apply (*rule real-le-trans* [*of real*(*nat*(*floor*(*real x powr* (*1*−*d*)))) *real x powr* (*1* − *d*) *real x*])
apply (*rule real-nat-floor*)
apply (*rule order-less-imp-le*)
apply (*rule powr-gt-zero*)
apply (*subgoal-tac real x* = *real x powr 1*)
apply (*erule ssubst*)**back**
apply (*case-tac x* = *1*)
apply (*erule ssubst*)
apply *force*
apply (*subst powr-le-cancel-iff*)
apply (*insert prems*, *force*+)
apply (*subst mult-assoc*)
apply (*subst mult-commute*)**backback**
apply (*subst mult-assoc* [*THEN sym*])
apply (*subst setsum-real*)
apply (*rule finite-subset* [*of* - {*0*..*x*}])
apply *force*
apply *force*
apply *simp*
apply (*subst setsum-multiplier-real*)
apply (*rule finite-subset* [*of* - {*0*..*x*}])

apply *force+*
done
also have ... = ($\sum$ *y*:{*z*. *nat* (*floor* (*real x powr* (*1* − *d*))) < *z* & *z* <= *x* & *z* : *prime*}. *ln* (*real x powr* (*1* − *d*)))
apply (*rule setsum-cong*)
apply *force*
apply (*subst powr-def*)
apply *simp*
done
also have ... <= ($\sum$ *y*:{*z*. *nat* (*floor* (*real x powr* (*1* − *d*))) < *z* & *z* <= *x* & *z* : *prime*}. *ln* (*real y*))
apply (*rule setsum-le-cong*)
apply *auto*
apply (*rule order-less-imp-le*)
apply (*rule order-less-le-trans* [*of* - *real* (*natfloor* (*real x powr* (*1* − *d*))) + *1* -])
apply (*rule real-of-nat-floor-add-one-gt*)
apply (*simp only*: *natfloor-def*)
done
also have ... <= *theta x*
apply (*simp only*: *theta-setsum-pos-def*)
apply (*rule order-trans* [*of* - *setsum* (%*y*. *ln* (*real y*)) {*z*. *2* <= *z* & *z* <= *x* & *z* : *prime*}])
apply (*rule setsum-subset*)
apply (*rule finite-subset* [*of* - {*0*..*x*}])
apply *force+*
apply *auto*
apply (*subgoal-tac 1* <= *real xa*)
apply *force*
apply (*frule primes-always-ge-2*)
apply *arith*
apply (*subgoal-tac* ($\sum$ *y*:{*z*. *2* <= *z* & *z* <= *x* & *z* : *prime*}. *ln* (*real y*)) = ($\sum$ *y*:{*z*. *2* <= *z* & *z* <= *x* & *z* : *prime*}. (*lprime y*)))
apply (*erule ssubst*)
apply (*rule setsum-subset*)
apply (*blast*)
apply *force+*
apply (*simp add*: *lprime-def*)
apply *auto*
apply (*frule primes-always-ge-1*)
apply *force*
apply (*rule setsum-cong*)
apply *force*
apply *auto*
apply (*simp add*: *lprime-def*)

done
finally show *?thesis*.
qed

lemma *theta-pi-rel2*: *1 <= x ==> 0 < d ==> d < 1 ==> pi(x) * ln(real x) / (real x) <=*
*theta(x) / ((1 − d) * (real x)) + ln(real x) / ((1 − d) * (real x) powr d)*
proof−
assume *1 <= x* **and** *0 < d* **and** *d < 1*
have *step1*: *(1 − d)*pi(x) * ln(real(x)) − real(x) powr (1 − d) * ln(real(x)) <= theta x*
apply (*rule theta-pi-rel1*)
apply (*insert prems*, *auto*)
done
then have *step2*: *(1 − d)*pi(x) * ln(real(x)) <= theta x + real(x) powr (1 − d) * ln(real(x))*
apply *auto*
done
then have *step3*: *(1 − d)*pi(x) * ln(real(x)) / ((1 − d) * real(x)) <= (theta x + real(x) powr (1 − d) * ln(real(x))) / ((1 − d) * real(x))*
apply (*rule divide-right-mono*)
apply (*rule nonneg-times-nonneg*)
apply (*insert prems*, *auto*)
done
then have *step4*: *pi(x) * ln(real(x)) / real(x) <= (theta x + real(x) powr (1 − d) * ln(real(x))) / ((1 − d) * real(x))*
apply (*insert prems*, *auto*)
done
then have *step5*: *(1 − d)*pi(x) * ln(real(x)) / ((1 − d) * real(x)) <= theta x / ((1 − d) * real(x)) + real(x) powr (1 − d) * ln(real(x)) / ((1 − d) * real(x))*
apply (*subst add-divide-distrib* [*THEN sym*])
apply *auto*
done
then have *step6*: *(1 − d)*pi(x) * ln(real(x)) / ((1 − d) * real(x)) <= theta x / ((1 − d) * real(x)) + real(x) powr (1 − d) * ln(real(x)) / ((1 − d) * (real(x) powr 1))*
apply (*insert prems*, *auto*)
done
then have *step7*: *(1 − d)*pi(x) * ln(real(x)) / ((1 − d) * real(x)) <= theta x / ((1 − d) * real(x)) + ln(real(x)) / ((1 − d) * (real(x) powr d))*
apply (*subgoal-tac ln (real x) / ((1 − d) * real x powr d) = (ln (real x) / (1 − d)) * (real x powr (1 − d) / real x powr 1)*)
apply (*erule ssubst*)
apply (*insert prems*, *simp add*: *mult-commute*)

```
    apply (simp only: powr-divide-denom)
    apply simp
    done
  then show ?thesis
    apply (insert prems, simp)
    done
  qed

lemma theta-pi-rel3: 1 <= x ==> 0 < d ==> d < 1 ==> pi x * ln (real x) /
real x - theta(x) / (real x) <= (theta x / (real x))*(d / (1 - d)) + ln (real x) /
((1 - d) * real x powr d)
 proof-
  assume 1 <= x and 0 < d and d < 1
  have pi(x) * ln(real x) / (real x) <=
                        theta(x) / ((1 - d) * (real x)) + ln(real x) / ((1 - d) *
(real x) powr d)
    by (insert prems, simp only: theta-pi-rel2)
  then have step1: pi(x) * ln(real x) / (real x) - theta(x) / (real x) <=
                          theta(x) / ((1 - d) * (real x)) - theta(x) / (real x) +
ln(real x) / ((1 - d) * (real x) powr d)
    apply (insert prems, arith)
    done
  then have step2: pi(x) * ln(real x) / (real x) - theta(x) / (real x) <=
                        (theta(x) - ((1 - d) * theta(x))) / ((1 - d) * (real x)) +
ln(real x) / ((1 - d) * (real x) powr d)
    apply (subst diff-divide-distrib)
    apply (subst mult-divide-cancel-left)
    apply (insert prems, force)
    apply simp
    done
  then have step3: pi(x) * ln(real x) / (real x) - theta(x) / (real x) <=
                         (theta(x)*(1 - (1 - d))) / ((1 - d) * (real x)) + ln(real
x) / ((1 - d) * (real x) powr d)
    apply (subst right-diff-distrib)
    apply (subst times-ac1-one-right)
    apply (simp only: mult-commute)
    done
  then have step4: pi(x) * ln(real x) / (real x) - theta(x) / (real x) <=
                        (theta(x)*d) / ((1 - d) * (real x)) + ln(real x) / ((1 - d)
* (real x) powr d)
    apply (auto)
    done
  then have step5: pi(x) * ln(real x) / (real x) - theta(x) / (real x) <=
                        (theta(x) / real x)*(d / (1 - d)) + ln(real x) / ((1 - d) *
(real x) powr d)
```

apply (*auto simp add*: *mult-commute*)
done
then show *?thesis*.
qed

lemma *theta-pi-rel4*: *1 <= x ==> 0 < d ==> d < 1 ==> pi x * ln (real x) / real x − theta(x) / (real x) <= (theta x / (real x))*(d / (1 − d)) + (2::real) / ((1 − d)*d) * (real x) powr (− d / 2)*
proof−
assume *1 <= x* **and** *0 < d* **and** *d < 1*
have *pi x * ln (real x) / real x − theta(x) / (real x) <= (theta x / (real x))*(d / (1 − d)) + ln (real x) / ((1 − d) * real x powr d)*
by (*rule theta-pi-rel3*)
also have *... <= (theta x / (real x))*(d / (1 − d)) + ((real x) powr (d / 2)/(d/2)) / ((1 − d) * real x powr d)*
apply (*rule add-left-mono*)
apply (*rule divide-right-mono*)
apply (*rule ln-powr-bound*)
apply (*insert prems*, *auto*)
apply (*rule nonneg-times-nonneg*)
apply (*auto*)
done
also have *... = (theta x / (real x))*(d / (1 − d)) + ((2::real) * (real x) powr (d / 2)) / ((1 − d) * d * real x powr d)*
apply (*subst add-left-cancel*)
apply *auto*
apply (*subgoal-tac d * ((1 − d) * real x powr d) = (1 − d) * d * real x powr d*)
apply (*erule ssubst*)
apply (*rule fun-cong*)**back**
apply *auto*
apply (*simp add*: *mult-commute*)
done
also have *... = (theta x / (real x))*(d / (1 − d)) + (2::real) / ((1 − d) * d) * (real x) powr (− d / 2)*
apply (*subst add-left-cancel*)
apply (*insert prems*, *auto*)
apply (*subgoal-tac 2 * real x powr (d / 2) / ((1 − d) * d * real x powr d) = (2 / ((1 − d) * d)) * (real x powr (d/2) / real x powr d)*)
apply (*erule ssubst*)
apply (*subst powr-divide2*)
apply *clarsimp*
apply (*subgoal-tac d / 2 − d = − (d / 2)*)
apply (*erule ssubst*)
apply *force+*

done
finally show *?thesis*.
qed

lemma *theta-pi-rel5*: *1 <= x ==> 0 < d ==> d < 1 ==> abs (pi x * ln (real x) / real x − theta(x) / (real x)) <= (theta x / (real x))*(d / (1 − d)) + (2::real) / ((1 − d)*d) * (real x) powr (− d / 2)*
apply (*subst abs-eqI1*)
apply *simp*
apply (*rule divide-right-mono*)
apply (*subst l.sum-over-F2* [*THEN sym*])
apply (*subst pi-setsum-def*)
apply (*subst setsum-real*)
apply (*rule finite-subset* [*of* - {*0..x*}])
apply *force+*
apply (*subst mult-commute*)
apply (*subst setsum-multiplier-real*)
apply (*rule finite-subset* [*of* - {*0..x*}])
apply *force+*
apply *simp*
apply (*subgoal-tac* {*p*. *p* : *prime* & *p <= x*} = {*y*. *y <= x* & *y* : *prime*})
apply (*erule ssubst*)
apply (*rule setsum-le-cong*)
apply (*subst ln-le-cancel-iff*)
apply *clarify*
apply (*frule primes-always-ge-2*)
apply *arith*
apply *clarify*
apply (*frule primes-always-ge-2*)
apply *arith*
apply *clarify*
apply *blast*
apply *arith*
apply (*simp only*: *theta-pi-rel4*)
done

lemma *theta-pi-rel6-aux1*: *0 < (d::real) ==> d < 1/2 ==> d < 1 − d*
apply *arith*
done

lemma *theta-pi-rel6-aux2*: *0 < (d::real) ==> d < 1/2 ==> (d / (1 − d)) < 1*
apply (*subgoal-tac 0 < (1 − d)*)
apply (*frule theta-pi-rel6-aux1*)
apply *simp*
apply (*frule divide-strict-right-mono* [*of d 1 − d 1 − d*])

apply *force+*
done

lemma *theta-pi-rel6-aux3*: *0 < (d::real) ==> d < 1/2 ==> 2 / (d * (1 − d)) < 4 / d*
apply (*subst pos-divide-less-eq*)
apply (*rule real-mult-order*)
apply *force+*
done

lemma *theta-pi-aux1*: *0 < (d::real) ==> d < 1/2 ==> 1 / (1 − d) < 2*
apply (*subst pos-divide-less-eq*)
apply *force+*
done

lemma *theta-pi-rel6*: *1 <= x ==> 0 < d ==> d < 1/2 ==> abs (pi x * ln (real x) / real x − theta(x) / (real x)) <= theta(x) / real x + 4 / d * (real x) powr (− d / 2)*
proof−
assume *1 <= x* **and** *0 < d* **and** *d < 1/2*
have *lt*: *d < 1*
apply (*insert prems*, *auto*)
done
have *abs (pi x * ln (real x) / real x − theta(x) / (real x)) <= (theta x / (real x))*(d / (1 − d)) + (2::real) / ((1 − d)*d) * (real x) powr (− d / 2)*
apply (*insert prems lt*)
by (*simp only*: *theta-pi-rel5*)
also have ... *<= (theta x / (real x)) + (2::real) / ((1 − d)*d) * (real x) powr (− d / 2)*
apply *simp*
apply (*subgoal-tac theta x * d / (real x * (1 − d)) = (theta x / real x) * (d / (1 − d))*)
apply (*erule ssubst*)
apply (*subgoal-tac theta x / real x = (theta x / real x) * 1*)
apply (*erule ssubst*)**back**
apply (*rule mult-left-mono*)
apply (*rule order-less-imp-le*)
apply (*insert prems*)
apply (*simp add*: *theta-pi-rel6-aux2*)
apply (*rule real-ge-zero-div-gt-zero*)
apply (*auto simp add*: *theta-geq-zero*)
done
also have ... *<= (theta x / (real x)) + (4 / d) * (real x) powr (− d / 2)*
apply *simp*
apply (*subgoal-tac 2 * real x powr − (d / 2) / ((1 − d) * d) = (2 / (d * (1*

```
− d))) * real x powr − (d / 2))
    apply (erule ssubst)
    apply (subgoal-tac 4 * real x powr − (d / 2) / d = (4 / d) * real x powr −
(d / 2))
    apply (erule ssubst)
    apply (insert prems)
    apply (rule mult-right-mono)
    apply (rule order-less-imp-le)
    apply (simp only: theta-pi-rel6-aux3)
    apply (rule order-less-imp-le)
    apply force+
    done
  finally show ?thesis.
  qed


lemma theta-pi-rel7: 1 <= x ==> 0 < d ==> d < 1/2 ==> abs (pi x * ln
(real x) / real x − theta(x) / (real x)) <= ln 4 + 4 / d * (real x) powr (− d / 2)
 proof−
  assume 1 <= x and 0 < d and d < 1/2
  have abs (pi x * ln (real x) / real x − theta(x) / (real x)) <= theta(x) / real
x + 4 / d * (real x) powr (− d / 2)
    apply (insert prems)
    apply (rule theta-pi-rel6)
    apply auto
    done
  also have ... <= ln 4 + 4 / d * (real x) powr (− d / 2)
    apply simp
    apply (insert prems, auto)
    apply (subst pos-divide-le-eq)
    apply force
    apply (subst mult-commute)
    apply (rule order-less-imp-le)
    apply (rule theta-bound)
    apply auto
    done
  finally show ?thesis.
  qed

lemma theta-pi-rel8: 1 <= x ==> 0 < d ==> d < 1/2 ==> abs (pi x * ln
(real x) / real x − theta(x) / (real x)) <= 2 * d * theta(x) / real x + 4 / d *
(real x) powr (− d / 2)
 proof−
  assume 1 <= x and 0 < d and d < 1/2
  have lt: d < 1
```

apply (*insert prems*, *auto*)
done
have *abs* (*pi x* ∗ *ln* (*real x*) / *real x* − *theta*(*x*) / (*real x*)) <= (*theta x* / (*real x*))∗(*d* / (*1* − *d*)) + (*2*::*real*) / ((*1* − *d*)∗*d*) ∗ (*real x*) *powr* (− *d* / *2*)
apply (*insert prems lt*)
by (*simp only*: *theta-pi-rel5*)
also have ... <= *2* ∗ *d* ∗ (*theta x* / (*real x*)) + (*2*::*real*) / ((*1* − *d*)∗*d*) ∗ (*real x*) *powr* (− *d* / *2*)
apply *simp*
apply (*subgoal-tac theta x* ∗ *d* / (*real x* ∗ (*1* − *d*)) = (*1* / (*1* − *d*)) ∗ *d* ∗ (*theta x* / *real x*))
apply (*erule ssubst*)
apply (*subgoal-tac 2* ∗ *d* ∗ *theta x* / *real x* = *2* ∗ *d* ∗ (*theta x* / *real x*))
apply (*erule ssubst*)
apply (*rule mult-right-mono*)
apply (*rule order-less-imp-le*)
apply (*insert prems*)
apply *auto*
apply (*subgoal-tac d* / (*1* − *d*) = (*1* / (*1* − *d*)) ∗ *d*)
apply (*erule ssubst*)
apply (*rule mult-strict-right-mono*)
apply (*simp add*: *theta-pi-aux1*)
apply *force*+
apply (*rule real-ge-zero-div-gt-zero*)
apply (*auto simp add*: *theta-geq-zero*)
apply (*auto simp add*: *times-ac1*)
done
also have ... <= *2* ∗ *d* ∗ (*theta x* / (*real x*)) + (*4* / *d*) ∗ (*real x*) *powr* (− *d* / *2*)
apply *simp*
apply (*subgoal-tac 2* ∗ *real x powr* − (*d* / *2*) / ((*1* − *d*) ∗ *d*) = (*2* / (*d* ∗ (*1* − *d*))) ∗ *real x powr* − (*d* / *2*))
apply (*erule ssubst*)
apply (*subgoal-tac 4* ∗ *real x powr* − (*d* / *2*) / *d* = (*4* / *d*) ∗ *real x powr* − (*d* / *2*))
apply (*erule ssubst*)
apply (*insert prems*)
apply (*rule mult-right-mono*)
apply (*rule order-less-imp-le*)
apply (*simp only*: *theta-pi-rel6-aux3*)
apply (*rule order-less-imp-le*)
apply *force*+
done
finally show *?thesis*
apply *simp*

```
    done
  qed

lemma theta-pi-rel9: 1 <= x ==> 0 < d ==> d < 1 / 2 ==>
  abs (pi x * ln (real x) / real x - theta(x) / (real x)) <=
    2 * d * ln 4 + 4 / d * (real x) powr (- d / 2)
  proof-
    assume 1 <= x and 0 < d and d < 1/2
   have abs (pi x * ln (real x) / real x - theta(x) / (real x)) <= 2 * d * theta(x)
/ real x + 4 / d * (real x) powr (- d / 2)
      apply (insert prems, simp only: theta-pi-rel8)
      done
    also have ... <= 2 * d * ln 4 + 4 / d * (real x) powr (- d / 2)
      apply auto
      apply (subgoal-tac 2 * d * theta x / real x = 2 * d * (theta x / real x))
      apply (erule ssubst)
      apply (insert prems)
      apply (rule mult-left-mono)+
      apply (subst pos-divide-le-eq)
      apply force
      apply (subst mult-commute)
      apply (rule order-less-imp-le)
      apply (rule theta-bound)
      apply auto
      done
    finally show ?thesis.
    qed



lemma aux: 0 < d ==> 0 < r ==> EX (no::nat). ALL n. (no <= n -->
    (4::real) / d * (real n) powr (- d / 2) < r)
proof -
  assume 0 < d and 0 < r
  have (%x. (real x) powr (- (d / 2))) ----> 0
    apply (rule LIMSEQ-neg-powr)
    apply (simp add: prems)
    done
  then have (%x. (4::real) / d * (real x) powr (- d / 2)) ---->
      4 / d * 0
    apply (intro LIMSEQ-mult)
    apply (rule LIMSEQ-const)
    apply simp
    done
  also have 4 / d * 0 = 0
```

```
  by simp
 finally show ?thesis
  apply (unfold LIMSEQ-def)
  apply (drule-tac x = r in spec)
  apply (drule mp)
  apply (rule prems)
  apply clarsimp
  apply (rule-tac x = no in exI)
  apply clarify
  apply (drule-tac x = n in spec)
  apply clarify
  apply (subgoal-tac abs (4 * real n powr - (d / 2) / d)
     = 4 * real n powr - (d / 2) / d)
  apply (erule subst)
  apply assumption
  apply (subst abs-nonneg)
  apply (rule real-ge-zero-div-gt-zero)
  apply (rule nonneg-times-nonneg)
  apply force
  apply (rule order-less-imp-le)
  apply force
  apply (rule prems)
  apply (rule refl)
  done
 qed

lemma aux2: (%x. (pi x * ln(real x) / (real x)) -
   (theta x / (real x))) ----> 0
 apply (unfold LIMSEQ-def)
 apply clarsimp
 apply (subgoal-tac  EX (no::nat). ALL n. (no <= n -->
   (4::real) / (min (min (1 / 3) (r / 2)) (r / (4 * ln 4))) *
     (real n) powr (- (min (min (1 / 3) (r / 2)) (r / (4 * ln 4))) / 2)
       < r / 2))
 apply (erule exE)
 apply (rule-tac x = max no 1 in exI)
 apply clarify
 apply (drule-tac x = n in spec)
 apply (drule mp)
 apply simp
 apply (rule order-le-less-trans)
 apply (rule theta-pi-rel9)
 apply simp
 prefer 4
 apply (rule aux)
```

apply *simp*
apply (*rule real-mult-less-imp-less-div-pos*)
apply *simp*
apply *simp*
apply *simp*
prefer *3*
apply (*rule order-less-le-trans*)
prefer *2*
apply (*subgoal-tac r / 2 + r / 2 <= r*)
apply *assumption*
apply *simp*
apply (*rule add-le-less-mono*)
prefer *2*
apply *assumption*
apply *simp*
apply (*rule order-trans*)
prefer *2*
apply (*subgoal-tac 4 * (ln 4 * (r / (4 * ln 4))) <= r*)
apply *assumption*
apply *simp*
apply (*rule mult-left-mono*)
apply (*subst mult-commute*)
apply (*rule mult-right-mono*)
apply (*simp split*: *split-min*)
apply *force*
apply *force*
apply (*simp split*: *split-min*)
apply (*rule real-mult-less-imp-less-div-pos*)
apply *force*
apply *force*
apply (*rule order-le-less-trans*)
prefer *2*
apply (*subgoal-tac 1 / 3 < 1 / 2*)
apply *assumption*
apply *arith*
apply (*subst min-le-iff-disj*)+
apply *simp*
done

lemma *theta-limit-imp-pi-limit*: (%*x*. *theta x / (real x*)) ----> *1* ==>
(%*x*. *pi x * ln (real x*) / (*real x*)) ----> *1*
apply (*erule LIMSEQ-diff-approach-zero*)
apply (*rule aux2*)
done

end

39 Sums involving mu

theory *MuSum = Inversion + RealLnSum + Chebyshev3*:

39.1 Variants of inversion laws

constdefs *mu2* :: (*nat* => *real*)
mu2 d == *real*(*mu*(*int*(*d*)))

lemma *mu-inversion-nat1-real*: *ALL n*. (*0* < *n* −−>
f n = (∑ *d*:{*d*. *d dvd n*}. *g*(*n div d*))) ==> *0* < (*n*::*nat*) ==>
g n = (∑ *d*:{*d*. *d dvd n*}. (*mu2 d*) ∗ *f* (*n div d*))
apply (*unfold mu2-def*)
apply (*subst mu-inversion-nat1a* [*of f g*])
apply *assumption*+
apply (*subst real-eq-of-int* [*THEN sym*])
apply (*rule refl*)
done

lemma *mu-inversion-nat2-real*: *ALL n*. (*0* < *n* −−>
g n = (∑ *d*:{*d*. *d dvd n*}. (*mu2 d*) ∗ *f*(*n div d*))) ==>
0 < (*n*::*nat*) ==> *f n* = (∑ *d*:{*d*. *d dvd n*}. *g* (*n div d*))
apply (*unfold mu2-def*)
apply (*subst mu-inversion-nat2a* [*of g f*])
apply (*subst real-eq-of-int* [*THEN sym*])
apply *assumption*+
apply (*rule refl*)
done

lemma *mu-inversion-nat3-real*: *0* < (*n*::*nat*) ==> *g n* =
(∑ *d*:{}*0*..*n*}. (∑ *d′*:{}*0*..(*n div d*)}.
(*mu2 d*) ∗ *g*((*n div d*) *div d′*)))
apply (*unfold mu2-def*)
apply (*subst mu-inversion-nat3* [*of n g*])
apply *assumption*
apply (*subst real-eq-of-int* [*THEN sym*])
apply (*rule refl*)
done

lemma *abs-mu2-leq-1*: *abs*(*mu2 x*) <= *1*

apply (*unfold mu2-def mu-def*)
apply *simp*
done

lemma *mu-inversion-nat1a'*:
[| *ALL n. 0 < n −−> f n* = (∑ *d* | *d dvd n. g d*); *0 < n* |]
==> *g n* = (∑ *d* | *d dvd n. of-int* (*mu* (*int d*)) ∗ *f* (*n div d*))
apply (*rule mu-inversion-nat1a*)
apply (*clarify*)
apply (*subst general-inversion-nat1*)
apply *assumption*
apply (*drule-tac x* = *na* **in** *spec*)
apply *simp*
apply (*rule setsum-cong2*)
apply (*subst nat-div-div*)
apply *auto*
done

lemma *mu-inversion-nat1a'-real*:
[| *ALL n. 0 < n −−> f n* = (∑ *d* | *d dvd n. g d*); *0 < n* |]
==> *g n* = (∑ *d* | *d dvd n. mu2 d* ∗ *f* (*n div d*))
apply (*unfold mu2-def*)
apply (*subst mu-inversion-nat1a'* [*of f g*])
apply *assumption+*
apply (*subst real-eq-of-int* [*THEN sym*])
apply (*rule refl*)
done

lemma *aux3*: {}*0..*(*n*::*nat*)} = {*1..n*}
by *auto*

lemma *general-inversion-nat2-modified*: *0* < (*n*::*nat*) ==>
(∑ *d*=*1..n.* ∑ *d'*=*1..n div d. f d d'*) =
(∑ *c*=*1..n.* ∑ *d* | *d dvd c. f d* (*c div d*))
apply (*subst aux3* [*THEN sym*])
apply (*subst general-inversion-nat2* [*THEN sym*])
apply *assumption*
apply (*rule setsum-cong2*)
apply (*subst aux3*)
apply (*rule refl*)
done

lemma *general-inversion-nat2-cor1-modified*: *0* < (*n*::*nat*)
==> (∑ *d*=*1..n.* ∑ *d'*=*1..n div d. f d d'*) =
(∑ *d'*=*1..n.* ∑ *d*=*1..n div d'. f d d'*)

apply (*subgoal-tac* ($\sum d{=}1..n.\ \sum d'{=}1..n\ div\ d.\ f\ d\ d'$) =
($\sum d{:}\{)0..n\}.\ \sum d'{:}\{)0..n\ div\ d\}.\ f\ d\ d'$))
apply (*erule ssubst*)
apply (*subst general-inversion-nat2-cor1*)
apply *assumption*
apply (*subst aux3*)
apply (*rule setsum-cong2*)
apply (*subst aux3*)
apply (*rule refl*)
apply (*subst aux3*)
apply (*rule setsum-cong2*)
apply (*subst aux3*)
apply (*rule refl*)
done

39.2 Sum of mu div n

lemma *sumr-mu-div-n-bigo*: (%x. *sumr 0* ($x{+}1$) (%n. *mu2*($n{+}1$) / *real*($n{+}1$)))
=o
O(%x. *1*)
proof −
have (%x. *1*) = (%x. $\sum d{:}\ \{)0..x{+}1\}.\ \sum d'{:}\{)0..(x{+}1)\ div\ d\}.\ mu2\ d$)
proof (*rule ext*)
fix x
show *1* = ($\sum d{:}\ \{)0..x{+}1\}.\ \sum d'{:}\{)0..(x{+}1)\ div\ d\}.\ mu2\ d$)
proof −
have ($\sum d{\in}\{)0..x{+}1\}.\ \sum d'{:}\{)0..(x{+}1)\ div\ d\}.\ mu2\ d$) =
($\sum d{\in}\{)0..x{+}1\}.\ \sum d'{:}\{)0..(x{+}1)\ div\ d\}.\ mu2\ d\ *$
1)
by *simp*
also have ... = *1*
apply (*rule mu-inversion-nat3-real* [*of* $x{+}1$ %x. *1*, *THEN sym*])
apply *force*
done
finally show *?thesis* **by** (*rule sym*)
qed
qed
also have ... = (%x. $\sum d{:}\ \{)0..x{+}1\}.\ real((x{+}1)\ div\ d) * mu2\ d$)
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst real-eq-of-nat*)
apply (*subst setsum-constant*)
apply *simp*
apply *simp*
done

also have ... =
(%*x*. *sumr 0* (*x*+*1*) (%*d*. *real*((*x*+*1*) *div* (*d*+*1*)) ∗ *mu2* (*d*+*1*)))
apply (*rule ext*)
apply (*rule setsum-sumr3*)
done
also have ... =*o*
(%*x*. *sumr 0* (*x*+*1*) (%*d*. *real* (*x*+*1*) / *real*(*d*+*1*) ∗ *mu2*(*d*+*1*)))
+*o* *O*(%*x*. *sumr 0* (*x*+*1*) (%*d*. *1*)) (**is** *?LHS* =*o* *?term1* +*o* *?term2*)
apply (*rule bigo-compose2*)
apply (*rule bigo-sumr6*)
apply *force*
apply (*rule-tac x* = *1* **in** *exI*)
apply (*rule allI*)+
apply (*subgoal-tac abs*(*real* (*x div* (*y* + *1*)) ∗ *mu2* (*y* + *1*) −
real x / *real* (*y* + *1*) ∗ *mu2* (*y* + *1*)) =
abs(*real*(*x div* (*y* + *1*)) − *real x* / *real* (*y* + *1*)) ∗ *abs*(*mu2*(*y*+*1*)))
apply (*erule ssubst*)
apply (*rule mult-mono*)
apply (*subgoal-tac real* (*x div* (*y* + *1*)) − *real x* / *real* (*y* + *1*) <= *0*)
apply *simp*
apply (*rule nat-div-real-div* [*THEN conjunct2*])
apply (*subgoal-tac 0* <= *real x* / *real* (*y* + *1*) − *real*(*x div* (*y* + *1*)))
apply *simp*
apply (*rule nat-div-real-div* [*THEN conjunct1*])
apply *simp*
apply (*rule abs-mu2-leq-1*)
apply *force*
apply *force*
apply (*simp del*: *abs-mult add*: *abs-mult* [*THEN sym*] *ring-eq-simps*)
done
also have (%*x*. *sumr 0* (*x* + *1*) (%*d*. *1*)) = (%*x*. *real*(*x* + *1*))
apply (*rule ext*)
apply (*subst sumr-const*)
apply *simp*
done
also have *?term1* = (%*x*. *real* (*x* + *1*) ∗ *sumr 0* (*x* + *1*)
(%*n*. *mu2*(*n*+*1*) / *real*(*n*+*1*)))
apply (*rule ext*)
apply (*subst sumr-mult*)
apply *simp*
done
finally have (%*x*. *real* (*x* + *1*) ∗
sumr 0 (*x* + *1*) (%*n*. *mu2* (*n* + *1*) / *real* (*n* + *1*))) =*o*
(%*x*. *1*) +*o* *O*(%*x*. *real*(*x* + *1*))
by (*rule bigo-add-commute-imp*)

```
  also have ... <= O(%x. real(x+1))
    apply (rule bigo-plus-absorb-lemma1)
    apply (rule bigo-bounded)
    apply auto
    done
  finally have (%x. real (x + 1) *
      sumr 0 (x + 1) (%n. mu2 (n + 1) / real (n + 1))) =o O(%x. real(x+1))
        (is ?term3 =o ?term4).
  then have (%x. 1 / real (x + 1)) * ?term3 =o
      (%x. 1 / real (x + 1)) *o ?term4
    by (rule set-times-intro2)
  also have ... <= O((%x. (1 / real (x + 1))) * (%x. real(x + 1)))
    by (rule bigo-mult2)
  finally show ?thesis
    by (simp add: func-times)
qed
```

39.3 Sum of mu div n times ln n over n

```
lemma aux-nat-diff: (a::nat) <= x ==> x - a + a = x
  by auto

lemma aux: (%x. sumr 0 (natfloor(abs x) + 1)
      (%y. mu2 (y + 1) / real (y + 1) *
          ln ((abs x + 1) / (real (y + 1)))))
  =o (%x. sumr 0 (natfloor(abs x) + 1)
        (%y. mu2 (y + 1) / (real (y + 1)) *
            (sumr 0 (natfloor((abs x + 1) / real (y + 1)))
              (%n. 1 / (real n + 1)) +
                gamma))) +o O(%x. 1)
    (is ?LHS =o ?RH1 +o O(%x. 1))
proof -
  note bigo-sumr8 [OF better-ln-theorem2-real, of
       %x. natfloor(abs x) + 1
       %x y. mu2(y+1) / real(y+1)
       %x y. ((abs x) + 1) / (real (y+1)) - 1]
  also have (%x. sumr 0 (natfloor (abs x) + 1)
       (%y. mu2 (y + 1) / real (y + 1) *
           ln (abs ((abs x + 1) / real (y + 1) - 1) + 1))) = ?LHS
    apply (rule ext)
    apply (rule sumr-cong)
    apply (subst abs-nonneg)backback
    apply (simp del: One-nat-def)
    apply (rule div-ge-1)
    apply auto
```

apply (*subgoal-tac real*(*natfloor*(*abs x*)) <= *abs x*)
apply *force*
apply (*rule real-natfloor-le*)
apply *force*
done
also have (%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*y*. *mu2* (*y* + *1*) / *real* (*y* + *1*) *
((%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*) (%*n*. *1* / (*real n* + *1*))) +
(%*x*. *gamma*))
((*abs x* + *1*) / *real* (*y* + *1*) − *1*))) = *?RH1*
apply (*rule ext*)
apply (*rule sumr-cong*)
apply (*simp only*: *func-plus*)
apply (*subgoal-tac natfloor*(*abs* ((*abs x* + *1*) / *real* (*y* + *1*) − *1*)) + *1* =
natfloor((*abs x* + *1*) / *real* (*y* + *1*)))
apply (*erule ssubst*)
apply (*rule refl*)
apply (*subst abs-nonneg*)**backback**
apply (*simp del*: *One-nat-def*)
apply (*rule div-ge-1*)
apply *force*
apply *simp*
apply (*subgoal-tac real y* <= *real* (*natfloor* (*abs x*)))
apply (*erule order-trans*)
apply (*rule real-natfloor-le*)
apply *force*
apply *simp*
apply (*subgoal-tac* 1 = *real* 1)
apply (*erule ssubst*)
apply (*subst natfloor-subtract*)
apply *simp*
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real*(*Suc y*) = *real y* + *1*)
apply (*erule ssubst*)
apply *simp*
apply (*subgoal-tac real y* <= *real* (*natfloor* (*abs x*)))
apply (*erule order-trans*)
apply (*rule real-natfloor-le*)
apply *force*
apply *force*
apply *simp*
apply (*rule aux-nat-diff*)
apply (*rule real-le-natfloor*)
apply *simp*

apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real*(*Suc y*) = *real y* + *1*)
apply (*erule ssubst*)
apply *simp*
apply (*subgoal-tac real y* <= *real* (*natfloor* (*abs x*)))
apply (*erule order-trans*)
apply (*rule real-natfloor-le*)
apply *force*
apply *force*
apply *simp*
apply *simp*
done
also have (%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*y*. *abs* (*mu2* (*y* + *1*) / *real* (*y* + *1*) *
(*1* / (*abs* ((*abs x* + *1*) / *real* (*y* + *1*) − *1*) + *1*))))) =
(%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*) (%*y*. *abs*(*mu2* (*y* + *1*)) / (*abs x* + *1*)))
apply (*rule ext*)
apply (*rule sumr-cong*)
proof −
fix *x* **fix** *y*
assume *y* < *natfloor* (*abs x*) + *1*
show *abs* (*mu2* (*y* + *1*) / *real* (*y* + *1*) *
(*1* / (*abs* ((*abs x* + *1*) / *real* (*y* + *1*) − *1*) + *1*))) =
abs (*mu2* (*y* + *1*)) / (*abs x* + *1*)
proof −
have *abs* ((*abs x* + *1*) / *real* (*y* + *1*) − *1*) =
(*abs x* + *1*) / *real* (*y* + (*1*::*nat*)) − *1* (**is** *?one* = *?two*)
apply (*rule abs-nonneg*)
apply *simp*
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real*(*Suc y*) = *real y* + *1*)
apply (*erule ssubst*)
apply *simp*
apply (*subgoal-tac real*(*natfloor*(*abs x*)) <= *abs x*)
apply (*rule order-trans*)
prefer *2*
apply *assumption*
apply (*insert prems*, *arith*)
apply (*rule real-natfloor-le*)
apply *auto*
done
then have *?one* + *1* = (*abs x* + *1*) / *real* (*y* + *1*) (**is** *?two* = *?three*)
by *simp*

```
    then have 1 / ?two = 1 / ?three
      by simp
    also have ... = real(y + 1) / (abs x + 1) (is ... = ?four)
      by simp
    finally have 1 / ?two = ?four.
    then have mu2 (y + 1) / real (y + 1) * (1 / ?two) =
      mu2 (y + 1) / real (y + 1) * ?four
      by simp
    also have ...= mu2(y + 1) / (abs x + 1)
      by simp
    finally have abs(mu2 (y + 1) / real (y + 1) * (1 / ?two)) =
      abs(mu2(y + 1) / (abs x + 1))
      by force
    also have ... = abs(mu2(y+1)) / (abs(abs x + 1))
      by (simp add: abs-divide)
    also have abs (abs x + 1) = abs x + 1
      apply (rule abs-nonneg)
      apply arith
      done
    finally show ?thesis.
  qed
  qed
finally have ?LHS =o ?RH1 +o O(%x. sumr 0 (natfloor (abs x) + 1)
        (%y. abs (mu2 (y + 1)) / (abs x + 1))).
also have ... <= ?RH1 +o O(%x. 1)
  apply (rule set-plus-mono)
  apply (rule bigo-elt-subset)
  apply (rule bigo-bounded)
  apply (rule allI)
  apply (rule sumr-ge-zero-cong)
  apply (rule real-ge-zero-div-gt-zero)
  apply force
  apply arith
  apply (rule allI)
  apply (subgoal-tac sumr 0 (natfloor (abs x) + 1)
    (%y. abs (mu2 (y + 1)) / (abs x + 1)) <= sumr 0 (natfloor (abs x) + 1)
    (%y. 1 / (abs x + 1)))
  apply (erule order-trans)
  apply (subst sumr-const)
  apply (subst natfloor-add [THEN sym])
  apply force
  apply simp
  apply (rule real-le-mult-imp-div-pos-le)
  apply arith
  apply simp
```

```
    apply (rule real-natfloor-le)
    apply arith
    apply (rule sumr-le-cong)
    apply (rule divide-right-mono)
    apply (rule abs-mu2-leq-1)
    apply arith
    done
  finally show ?thesis.
qed

lemma sumr-mu-div-n-times-ln-div-nbigo:
  (%x. sumr 0 (natfloor(abs x)+1) (%n. (mu2(n+1) / real(n+1)) *
    ln((abs x+1) / (real (n+1))))) =o O(%x. 1) (is ?LHS =o ?RHS)
proof -
  note aux
  then show ?thesis
    apply (elim rev-subsetD)
    apply (rule bigo-plus-absorb-lemma1)
    proof -
      show (%x. sumr 0 (natfloor (abs x) + 1)
          (%y. mu2 (y + 1) / real (y + 1) *
              (sumr 0 (natfloor ((abs x + 1) / real (y + 1)))
                (%n. 1 / (real n + 1)) +
               gamma))) =o O(%x. 1) (is ?LHS =o ?RHS)
      proof -
        have ?LHS = (%x. sumr 0 (natfloor(abs x) + 1)
          (%y. mu2 (y + 1) / real (y + 1) *
              (sumr 0 (natfloor((abs x + 1) / real (y + 1)))
                (%n. 1 / (real n + 1))))) +
           (%x. sumr 0 (natfloor(abs x) + 1)
              (%y. mu2 (y + 1) / real (y + 1) * gamma))
           (is ?LHS = ?term1 + ?term2)
          by (simp add: sumr-add func-plus ring-eq-simps
            del: One-nat-def)
        also have ?term1 + ?term2 =o O(%x. 1)
        proof -
          have ?term1 = (%x.(∑ y:{}0..natfloor(abs x)+1}.
            ∑ n:{}0..(natfloor(abs x)+1) div y}.
            ((mu2 y) / (real (y * n)))))
            apply (rule ext)
            apply (subst setsum-sumr3 [THEN sym])
            apply (rule setsum-cong2)
            apply (subst setsum-sumr3)
            apply (subst sumr-mult)
            apply (subst natfloor-div-nat)
```

apply *force*
apply *force*
apply (*subgoal-tac natfloor* (*abs x* + *1*) = *natfloor* (*abs x*) + *1*)
apply (*erule ssubst*)
apply (*rule sumr-cong*)
apply (*simp add*: *ring-eq-simps*)
apply (*subst natfloor-add* [*THEN sym*])
apply *simp*
apply *simp*
done
also have ... = (%*x*.($\sum$ *c*:{}*0*..*natfloor* (*abs x*) + *1*}.
$\sum$ *y*:{*y*. *y dvd c*}. ((*mu2 y*) / (*real* (*y* ∗ (*c div y*))))))
apply (*rule ext*)
apply (*rule general-inversion-nat2*)
apply *force*
done
also have ... = (%*x*.($\sum$ *c*:{}*0*..*natfloor* (*abs x*) + *1*}.
$\sum$ *y*:{*y*. *y dvd c*}. ((*mu2 y*) / (*real c*))))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*rule setsum-cong2*)
apply (*subst dvd-mult-div-cancel*)
apply *auto*
done
also have ... = (%*x*.($\sum$ *c*:{}*0*..*natfloor* (*abs x*) + *1*}.
(*1* / (*real c*)) ∗ ($\sum$ *y*:{*y*. *y dvd c*}. *mu2 y*)))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst setsum-const-times* [*THEN sym*])
apply *simp*
done
also have ... = (%*x*.($\sum$ *c*:{}*0*..*natfloor* (*abs x*) + *1*}.
(*1* / (*real c*)) ∗ (*if c* = *1 then 1 else 0*)))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*unfold mu2-def*)
apply (*subst real-eq-of-int*)
apply (*subst moebius-prop-nat-general*)
apply *auto*
done
also have ... = (%*x*. (*1* / *real* (*1*::*nat*)))
apply (*rule ext*)
apply (*rule mu-aux2*)
apply *force*
done

```
      also have ... =o O(%x. 1)
        by auto
      finally have a: ?term1 =o O(%x. 1).
      moreover have ?term2 =o O(%x. 1)
      proof −
        have ?term2 = (%x. gamma *
          sumr 0 (natfloor (abs x) + 1) (%y. mu2 (y + 1) / real (y + 1)))
          apply (rule ext)
          apply (subst sumr-mult)
          apply (simp add: ring-eq-simps)
          done
        also have ... =o O(%x. 1)
          apply (rule bigo-const-mult7)
          apply (rule  bigo-compose1 [OF sumr-mu-div-n-bigo,
            of %x. natfloor(abs x)])
          done
        finally show ?thesis.
      qed
      ultimately have ?term1 + ?term2 =o O(%x. 1) + O(%x. 1)
        by (rule set-plus-intro)
      also have ... <= O(%x. 1) by auto
      finally show ?thesis.
    qed
    finally show ?thesis.
  qed
qed
```

39.4 Mertens theorem

```
lemma Mertens-theorem-aux:
  (%x. sumr 0 (x + 1) (%d. Lambda (d + 1) / real (d + 1))) =o
    (%x. ln (real x + 1)) +o (O(%x. 1) + O(%x. psi (x + 1) / (real x + 1)))
proof −
  have (%x. (real (x + 1)) * (sumr 0 (x + 1) (%d.
    (Lambda (d + 1)) / (real (d + 1))))) = (%x. (sumr 0 (x + 1)
      (%d. (Lambda (d + 1)) * ((real (x + 1)) / (real (d + 1))))))
    apply (rule ext)
    apply (subst sumr-mult)
    apply (rule sumr-cong)
    apply simp
    done
  also have ... =o (%x. sumr 0 (x + 1) (%d. Lambda (d + 1) *
    real ((x + 1) div (d + 1)))) +o O(%x. sumr 0 (x + 1) (%d. Lambda(d +
1)))
```

```
    apply (rule bigo-sumr6)
    apply (rule allI)+
    apply (rule Lambda-ge-zero)
    apply (rule-tac x = 1 in exI)
    apply clarsimp
    apply (subst times-divide-eq-right [THEN sym])
    apply (subst right-diff-distrib [THEN sym])
    apply (subst abs-nonneg)
    apply (rule nonneg-times-nonneg)
    apply (rule Lambda-ge-zero)
    apply (rule nat-div-real-div1)
    apply (rule real-mult-le-one-le)
    apply (rule Lambda-ge-zero)
    apply (rule nat-div-real-div1)
    apply (rule nat-div-real-div2)
    done
  also have (%x. sumr 0 (x + 1) (%d. Lambda (d + 1))) = (%x. psi (x + 1))
    apply (rule ext)
    apply (unfold psi-def)
    apply (induct-tac x)
    apply (simp add: Lambda-zero)
    apply simp
    done
  finally have (%x. real (x + 1) * sumr 0 (x + 1)
    (%d. Lambda (d + 1) / real (d + 1))) =o
      (%x. sumr 0 (x + 1) (%d. Lambda (d + 1) * real ((x + 1) div (d + 1))))
+o
      O(%x. psi (x + 1)).
  also have ... <= ((%x. (real x + 1) * ln(real x + 1)) +o
      O(%x. real x + 1)) + O(%x. psi(x + 1))
  proof (rule set-plus-mono3)
    have (%x. sumr 0 (x + 1)
      (%d. Lambda (d + 1) * real ((x + 1) div (d + 1)))) =
        (%x. (∑ d:{}0..x+1}. Lambda d * (real((x + 1) div d))))
      apply (rule ext)
      apply (subst setsum-sumr3)
      apply (rule refl)
      done
    also have ... = (%x. (∑ d:{}0..x+1}. (∑ d':{}0..((x+1) div d)}.
        Lambda d)))
      apply (rule ext)
      apply (rule setsum-cong2)
      apply (subst setsum-constant)
      apply force
      apply (simp add: real-eq-of-nat mult-ac)
```

done
also have ... = (%x. ($\sum$ c:{}0..x+1}. ($\sum$ d:{d. d *dvd* c}.
Lambda d)))
apply (*rule ext*)
apply (*subst general-inversion-nat2*)
apply *auto*
done
also have ... = (%x. ($\sum$ c:{}0..x+1}. *ln* (*real* c)))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst ln-eq-setsum-Lambda*)
apply *auto*
done
also have ... = (%x. *sumr* 0 (x + 1) (%n. *ln*(*real* n + 1)))
apply (*rule ext*)
apply (*subst setsum-sumr3*)
apply (*rule sumr-cong*)
apply *simp*
done
also have ... =o ((%n. (*real* n + 1) $*$ *ln*(*real* n + 1)) $-$ (%n. *real* n))
+o O(%n. *ln* (*real* n + 1))
by (*rule identity-three*)
also have ... = (%n. (*real* n + 1) $*$ *ln*(*real* n + 1)) +o
($-$(%n. *real* n) +o O(%n. *ln*(*real* n + 1)))
by (*simp add*: *diff-minus set-plus-rearranges add-ac*)
also have ... <= (%n. (*real* n + 1) $*$ *ln*(*real* n + 1)) +o
($-$(%n. *real* n) +o O(%n. *real* n + 1))
apply (*rule set-plus-mono*)
apply (*rule set-plus-mono*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply (*rule allI*)
apply (*rule ln-ge-zero*)
apply *force*
apply (*rule allI*)
apply (*subst add-commute*)
apply (*rule order-trans*)
apply (*rule ln-add-one-self-le-self*)
apply *auto*
done
also have ... <= (%n. (*real* n + 1) $*$ *ln* (*real* n + 1)) +o
O(%n. *real* n + 1)
apply (*rule set-plus-mono*)
apply (*rule bigo-plus-absorb-lemma1*)
apply (*rule bigo-minus*)

```
    apply (rule bigo-bounded)
    apply auto
    done
  finally show (%x. sumr 0 (x + 1)
    (%d. Lambda (d + 1) * real ((x + 1) div (d + 1)))) =o
      (%n. (real n + 1) * ln (real n + 1)) +o O(%n. real n + 1).
qed
finally have (%x. real (x + 1) *
  sumr 0 (x + 1) (%d. Lambda (d + 1) / real (d + 1))) =o
    (%x. (real x + 1) * ln (real x + 1)) +o (O(%x. real x + 1) +
    O(%x. psi (x + 1))) (is ?LHS =o ?RHS)
  by (simp add: set-plus-rearranges)
then have (%x. 1 / (real x + 1)) * ?LHS =o (%x. 1 / (real x + 1)) *o
    ?RHS
  by (rule set-times-intro2)
also have (%x. 1 / (real x + 1)) * ?LHS =
  (%x. sumr 0 (x + 1) (%d. Lambda (d + 1) / real (d + 1)))
  apply (simp add: func-times)
  apply (rule ext)
  apply (subgoal-tac real x + 1 = real (Suc x))
  apply (erule ssubst)
  apply simp
  apply simp
  done
also have (%x. 1 / (real x + 1)) *o ?RHS =
    (%x. ln (real x + 1)) +o ((%x. 1 / (real x + 1)) *o O(%x. real x + 1)
      + (%x. 1 / (real x + 1)) *o O(%x. psi (x + 1)))
  by (simp add: set-times-plus-distribs func-times)
also have ... <= (%x. ln (real x + 1)) +o
    (O(%x. 1) + O(%x. psi (x + 1) / (real x + 1)))
  apply (rule set-plus-mono)
  apply (rule set-plus-mono2)
  apply (rule order-trans)
  apply (rule bigo-mult2)
  apply (simp add: func-times)
  apply (rule order-trans)
  apply (rule bigo-mult2)
  apply (simp add: func-times)
  done
finally show ?thesis.
qed

lemma Mertens-theorem:
  (%x. sumr 0 (x + 1) (%d. Lambda (d + 1) / real (d + 1))) =o
    (%x. ln (real x + 1)) +o O(%x. 1)
```

proof −
have (%*x*. *psi*(*x* + *1*)) =*o* *O*(%*x*. *real* (*x* + *1*))
by (*rule bigo-compose1* [*OF psi-bigo*])
also have (%*x*. *real* (*x* + (*1*::*nat*))) = (%*x*. *real x* + *1*)
by (*rule ext*, *simp*)
finally have (%*x*. *psi*(*x* + *1*)) =*o* *O*(%*x*. *real x* + *1*).
then have (%*x*. *1* / (*real x* + *1*)) ∗ (%*x*. *psi*(*x*+*1*)) =*o*
(%*x*. *1* / (*real x* + *1*)) ∗*o* *O*(%*x*. *real x* + *1*)
by (*rule set-times-intro2*)
also have (%*x*. *1* / (*real x* + *1*)) ∗ (%*x*. *psi*(*x*+*1*)) =
(%*x*. *psi*(*x*+*1*) / (*real x* + *1*)) **by** (*simp add*: *func-times*)
also have (%*x*. *1* / (*real x* + *1*)) ∗*o* *O*(%*x*. *real x* + *1*) <=
O((%*x*. *1* / (*real x* + *1*)) ∗ (%*x*. *real x* + *1*))
by (*rule bigo-mult2*)
also have ((%*x*. *1* / (*real* (*x*::*nat*) + *1*)) ∗ (%*x*. *real x* + *1*)) = (%*x*. *1*)
by (*simp add*: *func-times*)
finally have *a*: *O*(%*x*. *psi*(*x* + *1*) / (*real x* + *1*)) <= *O*(%*x*. *1*)
by (*rule bigo-elt-subset*)

note *Mertens-theorem-aux*
also have (%*x*. *ln* (*real x* + *1*)) +*o*
(*O*(%*x*. *1*) + *O*(%*x*. *psi* (*x* + *1*) / (*real x* + *1*))) <=
(%*x*. *ln* (*real x* + *1*)) +*o* *O*(%*x*. *1*)
apply (*rule set-plus-mono*)
apply (*rule bigo-plus-subset4*)
apply *simp*
apply (*rule a*)
done
finally show *?thesis*.
qed

lemma *Mertens-theorem-real*:
(%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*) (%*d*. *Lambda* (*d* + *1*) / *real* (*d* + *1*))) =*o*
(%*x*. *ln* (*abs x* + *1*)) +*o* *O*(%*x*. *1*) (**is** *?LHS* =*o* *?RHS*)
proof −
have *?LHS* =*o* (%*x*. *ln*(*real*(*natfloor*(*abs x*)) + *1*)) +*o* *O*(%*x*. *1*)
by (*rule bigo-compose2* [*OF Mertens-theorem*])
also have ... <= ((%*x*. *ln*(*abs x* + *1*)) +*o* *O*(%*x*. *1* / (*abs x* + *1*))) +
O(%*x*. *1*)
apply (*rule set-plus-mono3*)
apply (*rule bigo-add-commute-imp*)
apply (*rule ln-real-approx-ln-nat*)
done
also have ... <= (%*x*. *ln*(*abs x* + *1*)) +*o* *O*(%*x*. *1*)
apply (*simp add*: *set-plus-rearranges add-ac*)

```
    apply (rule set-plus-mono)
    apply (rule bigo-plus-subset4)
    apply (rule bigo-elt-subset)
    apply (rule bigo-bounded)
    apply auto
    apply arith
    apply (rule real-le-mult-imp-div-pos-le)
    apply arith
    apply auto
    done
  finally show ?thesis.
qed

lemma Mertens-theorem-real2: (%x. ∑ m = 1..natfloor (abs x) + 1.
    Lambda m / real m) =o (%x. ln (abs x + 1)) +o O(%x. 1)
      (is ?left =o ?right)
  apply (subgoal-tac ?left =
    (%x. sumr 0 (natfloor (abs x) + 1) (%d. Lambda (d + 1) / real (d + 1))))
  apply (erule ssubst)
  apply (rule Mertens-theorem-real)
  apply (rule ext)
  apply (subst setsum-sumr4)
  apply (rule refl)
  done
```

39.5 Sum of mu n over n times (ln n over n) squared

```
lemma aux2a: (y::nat) <= natfloor (abs x) ==>
        abs((abs(x) + 1) / real (y + 1) - 1) + 1 =
          ((abs(x) + 1) / real (y + 1))
  apply (subst abs-nonneg)backback
  apply (simp del: One-nat-def)
  apply (rule div-ge-1)
  apply force
  apply simp
  apply (subgoal-tac real y <= real (natfloor (abs x)))
  apply (erule order-trans)
  apply (rule real-natfloor-le)
  apply force
  apply simp
  apply simp
done

lemma sumr-ln-div-bigo:
  (%n. sumr 0 (n+1) (%i. ln((real n + 1) / (real i + 1)))) =o
```

```
    (%n. real n) +o O(%n. ln (real n + 1))
proof −
  have (%n. sumr 0 (n+1) (%i. ln((real n + 1) / (real i + 1)))) =
    (%n. sumr 0 (n+1) (%i. ln (real n + 1))) −
    (%n. sumr 0 (n+1) (%i. ln (real i + 1)))
    apply (subst func-diff)
    apply (rule ext)
    apply (subst sumr-diff)
    apply (rule sumr-cong)
    apply (subst ln-div)
    apply auto
    done
  also have (%n. sumr 0 (n+1) (%i. ln (real n + 1))) = (%n. (real n + 1) *
    ln(real n + 1))
    apply (rule ext)
    apply (simp add: sumr-const ring-distrib)
    done
  finally have (%n. sumr 0 (n+1) (%i. ln((real n + 1) / (real i + 1)))) =
    (%n. (real n + 1) * ln(real n + 1)) +
      −(%n. sumr 0 (n+1) (%i. ln (real i + 1)))
    by (simp add: diff-minus)
  also have ... =o (%n. (real n + 1) * ln(real n + 1)) +o
    (−((%n. (real n + 1) * ln(real n + 1)) − (%n. real n)) +o
      O(%n. ln (real n + 1)))
    apply (rule set-plus-intro2)
    apply (rule bigo-minus2)
    apply (rule identity-three)
    done
  also have ... = real +o O(%n. ln(real n + 1))
    by (simp add: set-plus-rearranges add-ac)
  finally show ?thesis.
qed

lemma sumr-ln-div-bigo-real:
  (%x. sumr 0 (natfloor(abs x)+1) (%i. ln((abs x + 1) / (real i + 1)))) =o
    O(%x. abs x + 1)
proof −
  have (%x. sumr 0 (natfloor(abs x)+1)
     (%i. ln((abs x + 1) / (real i + 1)))) =
       (%x. sumr 0 (natfloor(abs x)+1) (%i. ln (abs x + 1))) −
       (%x. sumr 0 (natfloor(abs x)+1) (%i. ln (real i + 1)))
    apply (subst func-diff)
    apply (rule ext)
    apply (subst sumr-diff)
    apply (rule sumr-cong)
```

```
    apply (subst ln-div)
    apply auto
    apply arith
    done
  also have (%x. sumr 0 (natfloor(abs x)+1) (%i. ln (abs x + 1))) =
      (%x. (real (natfloor(abs x) + 1) * ln(abs x + 1)))
    apply (rule ext)
    apply (simp only: sumr-const ring-distrib)
    done
  finally have (%x. sumr 0 (natfloor(abs x)+1)
        (%i. ln((abs x + 1) / (real i + 1)))) =
      (%x. (real (natfloor(abs x) + 1) * ln(abs x + 1))) +
      -(%x. sumr 0 (natfloor(abs x)+1) (%i. ln (real i + 1)))
    by (simp add: diff-minus)
  also have ... =o (%x. real (natfloor (abs x) + 1) * ln (abs x + 1)) +o
      (-(%x. (real (natfloor (abs x) + 1)) *
        ln (abs x + 1)) +o O(%x. (abs x) + 1))
    apply (rule set-plus-intro2)
    apply (rule bigo-minus2)
    apply (rule identity-three-cor-real)
    done
  also have ... = O(%x. (abs x) + 1)
    by (simp add: set-plus-rearranges add-ac)
  finally show ?thesis.
qed

lemma aux2: (%x. sumr 0 (natfloor(abs x) + 1)
      (%y. mu2 (y + 1) / real (y + 1) *
          (ln ((abs x + 1) / (real (y + 1)))) ^2))
  =o (%x. sumr 0 (natfloor(abs x) + 1)
        (%y. mu2 (y + 1) / (real (y + 1)) *
          ln ((abs x + 1) / (real (y + 1))) *
            (sumr 0 (natfloor((abs x + 1) / real (y + 1)))
              (%n. 1 / (real n + 1)) +
                gamma))) +o O(%x. 1)
    (is ?LHS =o ?RH1 +o O(%x. 1))
proof -
  note bigo-sumr8 [OF better-ln-theorem2-real, of
      %x. natfloor(abs x) + 1
      %x y. mu2(y+1) / real(y+1) * ln ((abs x + 1) / (real (y + 1)))
      %x y. ((abs x) + 1) / (real (y+1)) - 1]
  also have (%x. sumr 0 (natfloor (abs x) + 1)
      (%y. mu2 (y + 1) / real (y + 1) * ln ((abs x + 1) / real (y + 1)) *
          ln (abs ((abs x + 1) / real (y + 1) - 1) + 1))) = ?LHS
    apply (rule ext)
```

apply (*rule sumr-cong*)
apply (*subst aux2a*)
apply *force*
apply (*subst realpow-two2* [*THEN sym*])
apply *simp*
done
also have (%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*y*. *mu2* (*y* + *1*) / *real* (*y* + *1*) ∗ *ln* ((*abs x* + *1*) / *real* (*y* + *1*)) ∗
((%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*) (%*n*. *1* / (*real n* + *1*))) +
(%*x*. *gamma*))
((*abs x* + *1*) / *real* (*y* + *1*) − *1*))) = *?RH1*
apply (*rule ext*)
apply (*rule sumr-cong*)
apply (*simp only*: *func-plus*)
apply (*subst natfloor-add* [*THEN sym*])
apply *force*
apply (*subst real-nat-one*)
apply (*subst aux2a*)
apply *force*
apply (*rule refl*)
done
also have (%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*y*. *abs* (*mu2* (*y* + *1*) / *real* (*y* + *1*) ∗
ln ((*abs x* + *1*) / *real* (*y* + *1*)) ∗
(*1* / (*abs* ((*abs x* + *1*) / *real* (*y* + *1*) − *1*) + *1*))))) =
(%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*y*. *abs*(*mu2* (*y* + *1*)) / (*abs x* + *1*) ∗
ln ((*abs x* + *1*) / *real* (*y* + *1*))))
apply (*rule ext*)
apply (*rule sumr-cong*)
apply (*subst aux2a*)
apply *force*
apply (*simp add*: *abs-divide* [*THEN sym*] *abs-mult* [*THEN sym*])
apply (*simp add*: *abs-mult abs-divide*)
apply (*subgoal-tac abs*(*ln*((*abs x* + *1*) / *real* (*Suc y*))) =
ln((*abs x* + *1*) / *real* (*Suc y*)))
apply (*subgoal-tac abs*(*abs x* + *1*) = *abs x* + *1*)
apply *simp*
apply (*rule abs-nonneg*)
apply *arith*
apply (*rule abs-nonneg*)
apply (*rule ln-ge-zero*)
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real*(*Suc y*) = *real y* + *1*)

apply (*erule ssubst*)
apply *simp*
apply (*subgoal-tac real*(*natfloor*(*abs x*)) <= *abs x*)
apply (*rule order-trans*)
prefer *2*
apply *assumption*
apply *arith*
apply (*rule real-natfloor-le*)
apply *auto*
done
also have (%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*y*. *abs* (*mu2* (*y* + *1*)) / (*abs x* + *1*) ∗
ln ((*abs x* + *1*) / *real* (*y* + *1*)))) =
(%*x*. *1* / (*abs x* + *1*)) ∗ (%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*y*. *abs* (*mu2* (*y* + *1*)) ∗
ln ((*abs x* + *1*) / *real* (*y* + *1*))))
by (*simp add*: *func-times sumr-mult ring-eq-simps*)
also have *O*(...) =
(%*x*. *1* / (*abs x* + *1*)) ∗*o* *O*(%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*y*. *abs* (*mu2* (*y* + *1*)) ∗ *ln* ((*abs x* + *1*) / *real* (*y* + *1*))))
apply (*subst bigo-mult6* [*THEN sym*])
apply (*rule allI*)
apply *clarsimp*
apply *arith*
apply (*rule refl*)
done
finally have *?LHS* =*o* *?RH1* +*o* ....
also have ... <= *?RH1* +*o* (%*x*. *1* / (*abs x* + *1*)) ∗*o*
O(%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*y*. *ln* ((*abs x* + *1*) / *real* (*y* + *1*))))
apply (*rule set-plus-mono*)
apply (*rule set-times-mono*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply (*rule allI*)
apply (*rule sumr-ge-zero-cong*)
apply (*rule nonneg-times-nonneg*)
apply *force*
apply (*rule ln-ge-zero*)
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real* (*y* + *1*) = *real y* + *1*)
apply (*erule ssubst*)
apply *simp*
apply (*rule nat-le-natfloor*)

apply *force*
apply *force*
apply *force*
apply (*rule allI*)
apply (*rule sumr-le-cong*)
apply (*rule real-le-one-mult-le*)
apply (*rule ln-ge-zero*)
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real* (*y* + *1*) = *real y* + *1*)
apply (*erule ssubst*)
apply *simp*
apply (*rule nat-le-natfloor*)
apply *force*
apply *force*
apply *force*
apply *force*
apply (*rule abs-mu2-leq-1*)
done
also have ... <= *?RH1* +*o* (%*x*. *1* / (*abs x* + *1*)) ∗*o* *O*(%*x*. (*abs x* + *1*))
apply (*rule set-plus-mono*)
apply (*rule set-times-mono*)
apply (*rule bigo-elt-subset*)
apply (*subgoal-tac* (%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*y*. *ln* ((*abs x* + *1*) / *real* (*y* + *1*)))) =
(%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*)
(%*y*. *ln* ((*abs x* + *1*) / (*real y* + *1*)))))
apply (*erule ssubst*)
apply (*rule sumr-ln-div-bigo-real*)
apply (*rule ext*)
apply (*rule sumr-cong*)
apply (*subgoal-tac real* (*y* + *1*) = *real y* + *1*)
apply (*erule subst*)
apply *simp*
apply *simp*
done
also have (%*x*. *1* / (*abs x* + *1*)) ∗*o* *O*(%*x*. (*abs x* + *1*)) =
O(%*x*. *1*::*real*)
apply (*subst bigo-mult6* [*THEN sym*])
apply (*rule allI*)
apply *simp*
apply *arith*
apply (*simp add*: *func-times*)
apply (*subgoal-tac* (%*x*. (*abs x* + *1*) / (*abs x* + *1*)) = (%*x*. *1*))
apply (*erule ssubst*)

apply (*rule refl*)
apply (*rule ext*)
apply *simp*
apply *arith*
done
finally show *?thesis*.
qed

lemma *Lambda-sum-mu-ln*: *1 <= x ==>*
Lambda x = −(∑ d:{d. d dvd x}. mu2(d) ∗ ln(real d))
proof −
assume *1 <= (x::nat)*
then have *Lambda x = (∑ d:{d. d dvd x}.*
∑ d′:{d′. d′ dvd (x div d)}.
mu2(d) ∗ Lambda((x div d) div d′))
apply (*unfold mu2-def*)
apply (*subst real-eq-of-int*)
apply (*rule mu-inversion-nat1*)
apply *force*
done
also have *... = (∑ d:{d. d dvd x}. mu2(d) ∗*
(∑ d′: {d′. d′ dvd (x div d)}.
Lambda((x div d) div d′)))
apply (*rule setsum-cong2*)
apply (*rule setsum-const-times*)
done
also have *... = (∑ d:{d. d dvd x}. mu2(d) ∗*
(∑ d′: {d′. d′ dvd (x div d)}.
Lambda d′))
apply (*rule setsum-cong2*)
apply (*subst general-inversion-nat1* [*THEN sym*])
apply *clarsimp*
apply (*rule nat-pos-div-dvd-gr-0*)
apply (*insert prems*, *arith*)
apply *assumption*
apply (*rule refl*)
done
also have *... = (∑ d:{d. d dvd x}. mu2(d) ∗ ln(real (x div d)))*
apply (*rule setsum-cong2*)
apply (*subst ln-eq-setsum-Lambda*)
apply *clarsimp*
apply (*rule nat-pos-div-dvd-gr-0*)
apply (*insert prems*, *arith*)
apply *assumption*
apply *simp*

done
also have ... = $(\sum d{:}\{d.\ d\ dvd\ x\}.\ mu2(d) * ln(real\ x\ /\ real\ d))$
apply (*rule setsum-cong2*)
apply (*subst nat-dvd-real-div*)
apply *auto*
apply (*rule dvd-pos-pos*)
prefer *2*
apply *assumption*
apply (*insert prems*, *arith*)
done
also have ... = $(\sum d{:}\{d.\ d\ dvd\ x\}.\ mu2\ d * ln\ (real\ x))$
$-\ (\sum d{:}\{d.\ d\ dvd\ x\}.\ mu2\ d * ln\ (real\ d))$
apply (*subst setsum-subtractf* [*THEN sym*])
apply (*rule finite-nat-dvd-set*)
apply (*insert prems*, *arith*)
apply (*rule setsum-cong2*)
apply (*subst ln-div*)
apply *force*
apply *clarsimp*
apply (*rule dvd-pos-pos*)
prefer *2*
apply *assumption*
apply (*insert prems*, *arith*)
apply (*simp add*: *ring-eq-simps*)
done
also have $(\sum d{:}\{d.\ d\ dvd\ x\}.\ mu2\ d * ln\ (real\ x)) =$
$ln\ (real\ x) * (\sum d{:}\{d.\ d\ dvd\ x\}.\ mu2\ d)$
apply (*subst setsum-const-times* [*THEN sym*])
apply (*simp add*: *mult-ac*)
done
also have ... = $ln(real\ x) * (if\ x = 1\ then\ 1\ else\ 0)$
apply (*unfold mu2-def*)
apply (*subst real-eq-of-int*)
apply (*subst moebius-prop-nat-general*)
apply (*insert prems*, *arith*)
apply (*rule refl*)
done
also have ... = *0*
apply (*case-tac x = 1*)
apply *simp*
apply *simp*
done
finally show *?thesis*
by *simp*
qed

```
lemma sumr-mu-div-n-times-ln-squared-div-nbigo:
  (%x. sumr 0 (natfloor(abs x)+1) (%n. (mu2(n+1) / real(n+1)) *
    (ln((abs x+1) / (real (n+1)))^2)) =o
      (%x. 2 * ln (abs x + 1)) +o O(%x. 1)
        (is ?LHS =o ?RHS)
proof −
  note aux2
  also have (%x. sumr 0 (natfloor (abs x) + 1)
        (%y. mu2 (y + 1) / real (y + 1) *
             ln ((abs x + 1) / real (y + 1)) *
             (sumr 0 (natfloor ((abs x + 1) / real (y + 1)))
               (%n. 1 / (real n + 1)) +
              gamma))) =
    (%x. sumr 0 (natfloor (abs x) + 1)
        ((%y. mu2 (y + 1) / real (y + 1) *
             ln ((abs x + 1) / real (y + 1)) *
             (sumr 0 (natfloor ((abs x + 1) / real (y + 1)))
               (%n. 1 / (real n + 1)))) +
        (%y. mu2 (y + 1) / real (y + 1) *
             ln ((abs x + 1) / real (y + 1)) * gamma)))
    apply (rule ext)
    apply (rule sumr-cong)
    apply (subst func-plus)
    apply (simp add: ring-distrib)
    done
  also have ... = (%x. sumr 0 (natfloor (abs x) + 1)
        (%y. mu2 (y + 1) / real (y + 1) *
             ln ((abs x + 1) / real (y + 1)) *
             (sumr 0 (natfloor ((abs x + 1) / real (y + 1)))
               (%n. 1 / (real n + 1))))) +
       (%x. sumr 0 (natfloor (abs x) + 1)
         (%y. mu2 (y + 1) / real (y + 1) *
             ln ((abs x + 1) / real (y + 1)) * gamma))
    apply (subst func-plus)
    apply (rule ext)
    apply (subst sumr-add)
    apply (rule sumr-cong)
    apply (simp add: func-plus)
    done
  finally have (%x. sumr 0 (natfloor (abs x) + 1)
       (%y. mu2 (y + 1) / real (y + 1) *
            ln ((abs x + 1) / real (y + 1)) ^ 2)) =o
     ((%x. sumr 0 (natfloor (abs x) + 1)
         (%y. mu2 (y + 1) / real (y + 1) *
```

```
            ln ((abs x + 1) / real (y + 1)) *
            sumr 0 (natfloor ((abs x + 1) / real (y + 1)))
             (%n. 1 / (real n + 1)))) +
      (%x. sumr 0 (natfloor (abs x) + 1)
         (%y. mu2 (y + 1) / real (y + 1) *
            ln ((abs x + 1) / real (y + 1)) *
            gamma))) +o
       O(%x. 1) (is ?LHS =o (?term1 + ?term2) +o O(%x. 1)).
also have ... = ?term1 +o (?term2 +o O(%x. 1))
  by (simp only: set-plus-rearranges add-ac)
also have ... <= ?term1 +o O(%x. 1)
  apply (rule set-plus-mono)
  apply (subst bigo-plus-idemp [THEN sym])back
  apply (rule set-plus-mono3)
  apply (subgoal-tac ?term2 = (%x. gamma) *
     (%x. sumr 0 (natfloor (abs x) + 1)
      (%y. mu2 (y + 1) / real (y + 1) *
          ln ((abs x + 1) / real (y + 1)))))
  apply (erule ssubst)
  apply (rule rev-subsetD)
  prefer 2
  apply (subgoal-tac (%x. gamma) *o O(%x. 1) <= O(%x. 1))
  apply assumption
  apply (rule bigo-const-mult6)
  apply (rule set-times-intro2)
  apply (rule sumr-mu-div-n-times-ln-div-nbigo)
  apply (simp only: func-times)
  apply (rule ext)
  apply (subst sumr-mult)
  apply (rule sumr-cong)
  apply simp
  done
also have ... <= ((%x. 2 * ln (abs x + 1)) +o O(%x. 1)) + O(%x.1)
  proof (rule set-plus-mono3)
    have ?term1 = (%x. (∑ y:{}0..natfloor (abs x)+1}.
      (mu2 y / real y *
          ln ((abs x + 1) / real y) *
          sumr 0 (natfloor ((abs x + 1) / real y))
           (%n. 1 / (real n + 1)))))
      apply (rule ext)
      apply (rule setsum-sumr3 [THEN sym])
      done
   also have ... = (%x. (∑ y:{}0..natfloor (abs x)+1}.
       ∑ n:{}0..natfloor ((abs x + 1) / real y)}.
          (mu2 y / real y * ln ((abs x + 1) / real y) *
```

1 / (real n))))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst sumr-mult*)
apply (*subst setsum-sumr3*)
apply (*rule sumr-cong*)
apply *simp*
done
also have ... = (%*x*. (∑ *y*:{}*0..natfloor* (*abs x*)+*1*}.
∑ *n*:{}*0..*(*natfloor* (*abs x*)+*1*) *div y*}.
(*mu2 y / real* (*y* ∗ *n*) ∗ *ln* ((*abs x* + *1*) / *real y*))))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst natfloor-div-nat*)
apply *force*
apply *force*
apply (*subst natfloor-add* [*THEN sym*])
apply *force*
apply *simp*
done
also have ... = (%*x*. (∑ *c*:{}*0..natfloor* (*abs x*) + *1*}.
∑ *y*:{*y*. *y dvd c*}. (*mu2 y / real* (*y* ∗ (*c div y*)) ∗
ln ((*abs x* + *1*) / *real y*))))
apply (*rule ext*)
apply (*rule general-inversion-nat2*)
apply *force*
done
also have ... = (%*x*. (∑ *c*:{}*0..natfloor* (*abs x*) + *1*}.
∑ *y*:{*y*. *y dvd c*}. (*mu2 y / real c*) ∗ *ln* (*abs x* + *1*) −
(*mu2 y / real c*) ∗ *ln* (*real y*)))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*rule setsum-cong2*)
apply (*subst dvd-mult-div-cancel*)
apply *force*
apply (*subst ln-div*)
apply *arith*
apply *clarsimp*
apply (*rule dvd-pos-pos*)
apply *assumption*+
apply (*simp add*: *right-diff-distrib*)
done
also have ... = (%*x*. (∑ *c*:{}*0..natfloor* (*abs x*) + *1*}.
∑ *y*:{*y*. *y dvd c*}. (*mu2 y / real c*) ∗ *ln* (*abs x* + *1*))) −
(%*x*. (∑ *c*:{}*0..natfloor* (*abs x*) + *1*}.

$\sum y{:}\{y.\ y\ dvd\ c\}.\ (mu2\ y\ /\ real\ c) * ln\ (real\ y)))$
(**is** *?temp = ?term2 − ?term3*)
apply (*subst func-diff*)
apply (*rule ext*)
apply (*subst setsum-subtractf* [*THEN sym*])
apply *force*
apply (*rule setsum-cong2*)
apply (*subst setsum-subtractf* [*THEN sym*])
apply (*rule finite-nat-dvd-set*)
apply *force*
apply (*rule refl*)
done
also have *... = ?term2 + −(?term3)*
by (*simp only*: *diff-minus*)
also have *?term2* = $(\%x.\ (\sum c{:}\{)0..natfloor\ (abs\ x) + 1\}.$
$(ln\ (abs\ x + 1)\ /\ real\ c) * (\sum y{:}\{y.\ y\ dvd\ c\}.\ mu2\ y)))$
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
done
also have *... =* $(\%x.\ (\sum c{:}\{)0..natfloor\ (abs\ x) + 1\}.$
$(ln\ (abs\ x + 1)\ /\ real\ c) * (if\ c = 1\ then\ 1\ else\ 0)))$
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*unfold mu2-def*)
apply (*subst real-eq-of-int*)
apply (*subst moebius-prop-nat-general*)
apply *force*
apply (*rule refl*)
done
also have *... =* $(\%x.\ ln\ (abs\ x + 1))$
apply (*rule ext*)
apply (*subst mu-aux2*)
apply *force*
apply *simp*
done
also have *?term3* = $(\%x.\ \sum c{:}\{)0..natfloor\ (abs\ x) + 1\}.$
$1\ /\ real\ c * (\sum y{:}\{y.\ y\ dvd\ c\}.\ mu2\ y * ln\ (real\ y)))$
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst setsum-const-times* [*THEN sym*])
apply *simp*
done

```
    also have ... = - (%x. ∑ c:{}0..natfloor (abs x) + 1}.
       Lambda c / real c)
      apply (subst func-minus)
      apply (rule ext)
      apply (subst setsum-negf [THEN sym])
      apply force
      apply (rule setsum-cong2)
      apply (subst Lambda-sum-mu-ln)
      apply force
      apply simp
      done
    also have - (...) = (%x. ∑ c:{}0..natfloor (abs x) + 1}.
       Lambda c / real c)
      by simp
    also have ... = (%x. sumr 0 (natfloor (abs x) + 1)
       (%d. Lambda (d + 1) / real (d + 1))) (is ?temp = ?term4)
      apply (rule ext)
      apply (rule setsum-sumr3)
      done
    also have (%x. ln (abs x + 1)) + ?term4 =o
       (%x. ln (abs x + 1)) +o ((%x. ln(abs x + 1)) +o O(%x. 1))
      apply (rule set-plus-intro2)
      apply (rule Mertens-theorem-real)
      done
    also have ... = (%x. 2 * ln (abs x + 1)) +o O(%x. 1)
      apply (simp only: set-plus-rearranges)
      apply (simp add: func-plus)
      done
    finally show ?term1 =o ....
  qed
  also have ... = (%x. 2 * ln (abs x + 1)) +o (O(%x. 1) + O(%x.1))
    by (simp only: set-plus-rearranges)
  also have ... = (%x. 2 * ln (abs x + 1)) +o O(%x. 1)
    by simp
  finally show ?thesis.
qed

end
```

40 The Selberg symmetry formula

```
theory Selberg = MuSum:
```

lemma *Selberg1*: $0 < n \Longrightarrow ln(real\ n)\ \hat{}\ 2 = (\sum\ d \mid d\ dvd\ n.$
$Lambda\ d * ln\ (real\ d) +$
$(\sum\ u \mid u\ dvd\ d.\ Lambda\ u * Lambda\ (d\ div\ u)))$
proof −
assume $0 < (n::nat)$
have $(ln(real\ n))\hat{}2 = (\sum x{:}\{(p,\ a).\ 0 < a\ \&\ p : prime\ \&\ p\ \hat{}\ a\ dvd\ n\}.$
$ln\ (real\ (fst\ x)))\hat{}2$ (**is** $?temp = (?term1)\hat{}2$)
apply (*subst ln-eq-setsum-Lambda2*)
apply (*rule prems*)
apply (*rule refl*)
done
also have ... = $?term1 * ?term1$
by (*rule realpow-two2* [*THEN sym*])
also have ... = $(\sum x{:}\{(p,\ a).\ 0 < a\ \&\ p : prime\ \&\ p\ \hat{}\ a\ dvd\ n\}.$
$(\sum y{:}\{(q,\ b).\ 0 < b\ \&\ q : prime\ \&\ q\ \hat{}\ b\ dvd\ n\}.$
$ln\ (real(fst\ x)) * ln\ (real(fst\ y))))$
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply (*subst mult-commute*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule refl*)
done
also have ... = $(\sum x{:}\ \{((p,\ a),\ (q,\ b)).$
$0 < a\ \&\ p : prime\ \&\ p\ \hat{}\ a\ dvd\ n\ \&$
$0 < b\ \&\ q : prime\ \&\ q\ \hat{}\ b\ dvd\ n\}.$
$ln\ (real\ (fst\ (fst\ x))) * ln\ (real\ (fst\ (snd\ x))))$
(**is** $?temp = (\sum x{:}\ \{((p,\ a),\ (q,\ b)).\ (?P\ p\ a\ q\ b)\}.\ ?t\ x)$)
apply (*subst setsum-cartesian-product*)
apply (*rule finite-subset*)
prefer *2*
apply (*rule l.finite-C* [*of n*])
apply (*force intro*: *dvd-imp-le prems*)
apply (*rule finite-subset*)
prefer *2*
apply (*rule l.finite-C* [*of n*])
apply (*force intro*: *dvd-imp-le prems*)
apply (*rule setsum-cong*)
apply *auto*
done
also have ... = $(\sum x{:}\ \{((p,\ a),\ (q,\ b)).\ (?P\ p\ a\ q\ b)\ \&\ (p\ \sim{=}\ q \mid$
$(a + b) <= multiplicity\ (int\ p)\ (int\ n))\}.\ ?t\ x) +$
$(\sum x{:}\ \{((p,\ a),\ (q,\ b)).\ (?P\ p\ a\ q\ b)\ \&\ (p = q\ \&$
$multiplicity\ (int\ p)\ (int\ n) < a + b)\}.\ ?t\ x)$
(**is** $?temp = ?term2 + ?term3$)
apply (*subst setsum-Un-disjoint* [*THEN sym*])

apply (*rule finite-subset*)
prefer *2*
apply (*rule finite-cartesian-product*)
apply (*rule l.finite-C* [*of n*])
apply (*rule l.finite-C* [*of n*])
apply (*force intro*: *dvd-imp-le prems*)
apply (*rule finite-subset*)
prefer *2*
apply (*rule finite-cartesian-product*)
apply (*rule l.finite-C* [*of n*])
apply (*rule l.finite-C* [*of n*])
apply (*force intro*: *dvd-imp-le prems*)
apply *force*
apply (*rule setsum-cong*)
apply *simp*
apply *auto*
done
also have *?term2* = ($\sum$ *x*:{((*p*,*a*),(*q*,*b*)). (*?P p a q b*) &
(($p \hat{} a * q \hat{} b$) *dvd n*)}. *?t x*)
apply (*rule setsum-cong*)
apply *auto*
apply (*rule relprime-dvd-prod-dvd*)
apply (*erule distinct-primes-power-gcd-1*)
apply *assumption*+
apply (*case-tac a* ~= *aa*)
apply (*rule relprime-dvd-prod-dvd*)
apply (*erule distinct-primes-power-gcd-1*)
apply *assumption*+
apply *simp*
apply (*subst power-add* [*THEN sym*])
apply (*subst multiplicity-power-dvd* [*THEN sym*])
apply (*rule prems*)
apply *assumption*+
apply (*subst multiplicity-power-dvd*)
apply (*rule prems*)
apply *assumption*
apply (*subst power-add*)
apply *assumption*
done
also have ... = ($\sum$ *x*:{(*p*,*a*). *0* < *a* & *p* : *prime* & $p \hat{} a$ *dvd n*}.
$\sum$ *y*:{(*q*,*b*). *0* < *b* & *q* : *prime* & $q \hat{} b$ *dvd n* &
(*fst x*) ^(*snd x*) * $q \hat{} b$ *dvd n*}. *ln*(*real*(*fst x*)) * *ln*(*real* (*fst y*)))
apply (*subst setsum-Sigma*)
apply (*rule finite-subset*)
prefer *2*

apply (*rule l.finite-C* [*of n*])
apply (*force intro*: *dvd-imp-le prems*)
apply (*rule ballI*)
apply (*rule finite-subset*)
prefer *2*
apply (*rule l.finite-C* [*of n*])
apply (*force intro*: *dvd-imp-le prems*)
apply (*rule setsum-cong*)
apply *auto*
done
also have ... = ($\sum$ *x*:{(*p*,*a*). *0* < *a* & *p* : *prime* & *p* ^ *a dvd n*}.
ln(*real* (*fst x*)) * ($\sum$ *y*:{(*q*,*b*). *0* < *b* & *q* : *prime* & *q* ^ *b dvd n* &
(*fst x*) ^ (*snd x*) * *q* ^ *b dvd n*}. *ln*(*real* (*fst y*))))
apply (*rule setsum-cong2*)
apply (*rule setsum-const-times*)
done
also have ... = ($\sum$ *x*:{(*p*,*a*). *0* < *a* & *p* : *prime* & *p* ^ *a dvd n*}.
Lambda((*fst x*) ^ (*snd x*)) *
($\sum$ *y*:{(*q*,*b*). *0* < *b* & *q* : *prime* & *q* ^ *b dvd n* &
(*fst x*) ^ (*snd x*) * *q* ^ *b dvd n*}. *Lambda*((*fst y*) ^ (*snd y*))))
apply (*rule setsum-cong2*)
apply (*subst Lambda-eq* [*THEN sym*])
apply *force*
apply (*subgoal-tac* *0* < *snd x*)
apply *assumption*
apply *force*
apply (*rule arg-cong*)**back**
apply (*rule setsum-cong2*)
apply (*rule Lambda-eq* [*THEN sym*])
apply *auto*
done
also have ... = ($\sum$ *u* | (*EX p a*. *0* < *a* & *p* : *prime* & *p* ^ *a* = *u* &
u dvd n). *Lambda u* *
($\sum$ *y*:{(*q*,*b*). *0* < *b* & *q* : *prime* & *q* ^ *b dvd n* &
u * *q* ^ *b dvd n*}. *Lambda*((*fst y*) ^ (*snd y*))))
apply (*rule setsum-reindex-cong'* [*THEN sym*])
apply (*rule finite-subset*)
prefer *2*
apply (*rule l.finite-C* [*of n*])
apply (*force intro*: *prems dvd-imp-le*)
apply (*subgoal-tac inj-on* (%*x*. (*fst x*) ^ (*snd x*)) *?Q*)
apply (*assumption*)
apply (*clarsimp simp add*: *inj-on-def*)
apply (*rule conjI*)
apply (*rule prime-prop-rzero*)

apply *assumption*+**back**
apply (*rule prime-prop2*)
prefer *5*
apply *assumption*+
apply (*auto simp add*: *image-def*)
done
also have ... = ... + ($\sum$ *u* | ~(*EX p a*. *0* < *a* & *p* : *prime* & *p* ^ *a* = *u*) &
u dvd n. *Lambda u* * ($\sum$ *y*:{(*q*, *b*). *0* < *b* & *q* : *prime* &
q ^ *b dvd n* & *u* * *q* ^ *b dvd n*}. *Lambda* (*fst y* ^ *snd y*)))
(**is** *?temp* = *?temp* + *?zeroterm*)
apply (*subgoal-tac ?zeroterm* = *0*)
apply *simp*
apply (*rule setsum-0'*)
apply (*rule ballI*)
apply (*subst Lambda-eq2*)
apply *blast*
apply *simp*
done
also have ... = ($\sum$ *u* | *u dvd n*. *Lambda u* *
($\sum$ *y*:{(*q*, *b*). *0* < *b* & *q* : *prime* &
q ^ *b dvd n* & *u* * *q* ^ *b dvd n*}. *Lambda* (*fst y* ^ *snd y*)))
apply (*subst setsum-Un-disjoint* [*THEN sym*])
apply (*rule finite-subset*)
prefer *2*
apply (*rule finite-nat-dvd-set*)
apply (*rule prems*)
apply *force*
apply (*rule finite-subset*)
prefer *2*
apply (*rule finite-nat-dvd-set*)
apply (*rule prems*)
apply *force*
apply *blast*
apply (*rule setsum-cong*)
apply *auto*
done
also have ... = ($\sum$ *u* | *u dvd n*. *Lambda u* *
($\sum$ *y*:{(*q*, *b*). *0* < *b* & *q* : *prime* & *u* * *q* ^ *b dvd n*}.
Lambda (*fst y* ^ *snd y*)))
apply (*rule setsum-cong2*)
apply (*rule arg-cong*)**back**
apply (*rule setsum-cong*)
apply *auto*
apply (*auto simp add*: *dvd-def*)
done

also have ... = ($\sum$ *u* | *u dvd n*. *Lambda u* ∗
($\sum$ *v* | (*EX q b*. *0* < *b* & *q* : *prime* & *v* = *q*^*b*) & *u* ∗ *v dvd n*.
Lambda v))
apply (*rule setsum-cong2*)
apply (*rule arg-cong*)**back**
apply (*rule setsum-reindex-cong*′ [*THEN sym*])
apply (*rule finite-subset*)
prefer *2*
apply (*rule l.finite-C* [*of n*])
apply *clarsimp*
apply (*rule dvd-imp-le*)
apply (*force simp add*: *dvd-def*)
apply (*rule prems*)
apply (*subgoal-tac inj-on* (%*x*. (*fst x*)^(*snd x*)) *?Q*)
apply (*assumption*)
apply (*clarsimp simp add*: *inj-on-def*)
apply (*rule conjI*)
apply (*rule prime-prop-rzero*)
apply *assumption*+**back**
apply (*rule prime-prop2*)
prefer *5*
apply *assumption*+
apply (*auto simp add*: *image-def*)
done
also have ... = ($\sum$ *u* | *u dvd n*. *Lambda u* ∗
($\sum$ *v* | (*u* ∗ *v*) *dvd n*. *Lambda v*))
apply (*rule setsum-cong2*)
apply (*rule arg-cong*)**back**
proof −
fix *x*
have ($\sum$ *v* | (*EX q b*. *0* < *b* & *q* : *prime* & *v* = *q* ^ *b*) & *x* ∗ *v dvd n*.
Lambda v) =
($\sum$ *v* | (*EX q b*. *0* < *b* & *q* : *prime* & *v* = *q* ^ *b*) & *x* ∗ *v dvd n*.
Lambda v) +
($\sum$ *v* | ~(*EX q b*. *0* < *b* & *q* : *prime* & *v* = *q* ^ *b*) & *x* ∗ *v dvd n*.
Lambda v) (**is** *?temp* = *?temp2* + *?zeroterm*)
apply (*subgoal-tac ?zeroterm* = *0*)
apply *simp*
apply (*rule setsum-0*′)
apply (*rule ballI*)
apply (*subst Lambda-eq2*)
apply *auto*
done
also have ... = ($\sum$ *v* | *x* ∗ *v dvd n*. *Lambda v*)
apply (*subst setsum-Un-disjoint* [*THEN sym*])

apply (*rule finite-subset*)
prefer *2*
apply (*rule finite-nat-dvd-set*)
apply (*rule prems*)
apply (*force simp add*: *dvd-def*)
apply (*rule finite-subset*)
prefer *2*
apply (*rule finite-nat-dvd-set*)
apply (*rule prems*)
apply (*force simp add*: *dvd-def*)
apply *blast*
apply (*rule setsum-cong*)
apply *blast*
apply (*rule refl*)
done
finally show ($\sum$ *v* | (*EX q b*. *0* < *b* & *q* : *prime* & *v* = *q* ^ *b*) &
x * *v dvd n*. *Lambda v*) = ($\sum$ *v* | *x* * *v dvd n*. *Lambda v*)**.**
qed
also have ... = ($\sum$ *u* | *u dvd n*. ($\sum$ *v* | (*u* * *v*) *dvd n*.
Lambda u * *Lambda v*))
apply (*rule setsum-cong2*)
apply (*subst setsum-const-times*)
apply (*rule refl*)
done
also have ... = ($\sum$ *u* | *u dvd n*.
($\sum$ *v* | *v dvd* (*n div u*). *Lambda u* * *Lambda v*))
apply (*rule setsum-cong2*)
apply (*rule setsum-cong*)
apply *auto*
apply (*force simp add*: *prems dvd-def*)
apply (*force simp add*: *prems dvd-def*)
done
also have ... = ($\sum$ *d* | *d dvd n*.
($\sum$ *u* | *u dvd d*. *Lambda u* * *Lambda* (*d div u*)))
apply (*rule general-inversion-nat3*)
apply (*rule prems*)
done
also have *?term3* = ($\sum x$: {((*p*,*a*),*c*). *p* : *prime* & *p* ^ *a dvd n* &
0 < *a* & *c* < *a*}. *ln*(*real* (*fst* (*fst x*))) * *ln*(*real* (*fst* (*fst x*))))
apply (*rule setsum-reindex-cong'*)
apply (*rule finite-subset*)
prefer *2*
apply (*rule finite-SigmaI*)
apply (*rule l.finite-C* [*of n*])
apply (*subgoal-tac finite* {..*snd a*(})

apply *assumption*
apply *simp*
apply *clarsimp*
apply (*erule dvd-imp-le*)
apply (*rule prems*)
apply (*subgoal-tac inj-on* (%*x*. ((*fst x*), ((*fst* (*fst x*)),
multiplicity (*int* (*fst* (*fst x*))) (*int n*) − (*snd x*)))) *?Q*)
apply *assumption*
apply (*clarsimp simp add*: *inj-on-def*)
apply (*simp only*: *prems multiplicity-power-dvd* [*THEN sym*])
apply *arith*
prefer *2*
apply *simp*
apply (*unfold image-def*)
apply (*insert prems*)
apply (*rule set-ext*)
apply (*rule iffI*)
apply *clarsimp*
apply (*rule-tac x* = *aa* **in** *exI*)
apply *clarify*
apply (*rule-tac x* = *b* **in** *exI*)
apply *clarify*
apply (*rule-tac x* = *multiplicity* (*int aa*) (*int n*) − *ba* **in** *exI*)
apply (*clarsimp simp add*: *prems multiplicity-power-dvd* [*THEN sym*])
apply *arith*
apply (*clarsimp simp add*: *prems multiplicity-power-dvd* [*THEN sym*])
apply *arith*
done
also have ... = (∑ *x*:{(*p*, *a*). *p* : *prime* & *p* ^ *a dvd n* & *0* < *a*}.
∑ *c*:{*0*..*snd x*(}. *ln* (*real* (*fst x*)) ∗ *ln* (*real* (*fst x*)))
apply (*subst setsum-Sigma*)
apply (*rule finite-subset*)
prefer *2*
apply (*rule l.finite-C* [*of n*])
apply (*force intro*: *prems dvd-imp-le*)
apply (*rule ballI*)
apply *simp*
apply (*rule setsum-cong*)
apply *auto*
done
also have ... = (∑ *x*:{(*p*, *a*). *p* : *prime* & *p* ^ *a dvd n* & *0* < *a*}.
ln (*real* ((*fst x*) ^(*snd x*))) ∗ *ln* (*real* (*fst x*)))
apply (*rule setsum-cong2*)
apply (*subst setsum-constant*)
apply *simp*

apply (*simp add*: *real-eq-of-nat* [*THEN sym*] *realpow-real-of-nat* [*THEN sym*])
apply (*subst ln-realpow*)
apply *auto*
apply (*auto simp add*: *prime-def*)
done
also have ... = ($\sum$ *x*:{(*p*, *a*). *p* : *prime* & *p* ^ *a dvd n* & *0* < *a*}.
ln (*real*((*fst x*) ^(*snd x*))) * *Lambda*((*fst x*) ^(*snd x*)))
apply (*rule setsum-cong2*)
apply (*subst Lambda-eq*)
apply *auto*
done
also have ... = ($\sum$ *d* | *d dvd n* & (*EX p a*. *p* : *prime* & *0* < *a* & *d* = *p*^*a*).
ln (*real d*) * *Lambda d*)
apply (*rule setsum-reindex-cong'* [*THEN sym*])
apply (*rule finite-subset*)
prefer *2*
apply (*rule l.finite-C* [*of n*])
apply (*force intro*: *prems dvd-imp-le*)
apply (*subgoal-tac inj-on* (%*x*. (*fst x*) ^(*snd x*)) *?Q*)
apply *assumption*
apply (*clarsimp simp add*: *inj-on-def*)
apply (*rule conjI*)
apply (*rule prime-prop-rzero*)
apply *assumption*+**back**
apply (*rule prime-prop2*)
prefer *5*
apply *assumption*+
apply (*auto simp add*: *image-def*)
done
also have ... =
($\sum$ *d* | *d dvd n* & (*EX p a*. *p* : *prime* & *0* < *a* & *d* = *p* ^ *a*).
ln (*real d*) * *Lambda d*) +
($\sum$ *d* | *d dvd n* & ~(*EX p a*. *p* : *prime* & *0* < *a* & *d* = *p* ^ *a*).
ln (*real d*) * *Lambda d*) (**is** *?temp* = *?temp'* + *?zeroterm*)
apply (*subgoal-tac ?zeroterm* = *0*)
apply *simp*
apply (*rule setsum-0'*)
apply (*rule ballI*)
apply (*subst Lambda-eq2*)
apply *auto*
done
also have ... = ($\sum$ *d* | *d dvd n*. *Lambda d* * *ln* (*real d*))
apply (*subst setsum-Un-disjoint* [*THEN sym*])
apply (*rule finite-subset*)
prefer *2*

```
    apply (rule finite-nat-dvd-set)
    apply (rule prems)
    apply force
    apply (rule finite-subset)
    prefer 2
    apply (rule finite-nat-dvd-set)
    apply (rule prems)
    apply force
    apply blast
    apply (rule setsum-cong)
    apply blast
    apply (rule mult-commute)
    done
  finally have ln(real n) ^ 2 =
    (∑ d | d dvd n. Lambda d * ln (real d)) +
    (∑ d | d dvd n. ∑ u | u dvd d. Lambda u * Lambda (d div u))
    by (subst add-commute)
  thus ?thesis
    by (subst setsum-addf)
qed

lemma Selberg2: (%x. ∑n = 1..natfloor (abs x) + 1.
    Lambda n * ln (real n) +
     (∑ u | u dvd n. Lambda u * Lambda (n div u)))
  =o (%x. 2 * (abs x + 1) * ln (abs x + 1)) +o O(%x. abs x + 1)
proof −
  have (%x. ∑n=1..natfloor(abs x) + 1.
         Lambda n * ln(real n) +
           (∑ u | u dvd n. Lambda u * Lambda (n div u))) =
       (%x. ∑n:{}0..natfloor(abs x) + 1}.
           (∑d | d dvd n. mu2(d) * (ln(real (n div d))) ^2))
    apply (rule ext)
    apply (rule setsum-cong)
    apply force
    apply (rule mu-inversion-nat1a'-real)
    apply clarify
    apply (rule Selberg1)
    apply assumption+
    apply force
    done
  also have ... = (%x. ∑d:{}0..natfloor(abs x)+1}.
     ∑k:{}0..(natfloor (abs x) + 1) div d}. mu2(d) * (ln(real k) ^2))
    apply (rule ext)
    apply (rule general-inversion-nat2 [THEN sym])
    apply force
```

done
also have ... = (%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*) (%*d*.
mu2(*d*+*1*) ∗ (*sumr 0* (*natfloor* ((*abs x* + *1*) / (*real d*+*1*)))
(%*k*. *ln*(*real k* + *1*) ^ *2*))))
apply (*rule ext*)
apply (*subst setsum-sumr3*)
apply (*rule sumr-cong*)
apply (*subst setsum-const-times*)
apply (*rule arg-cong*)**back**
apply (*subst setsum-sumr3*)
apply (*subgoal-tac* (*natfloor* (*abs x*) + *1*) *div* (*y* + *1*) =
natfloor ((*abs x* + *1*) / (*real y* + *1*)))
apply (*erule ssubst*)
apply (*rule sumr-cong*)
apply (*rule arg-cong*)**back**
apply (*rule arg-cong*)**back**
apply *simp*
apply (*subst natfloor-add* [*THEN sym*])
apply *force*
apply (*subst natfloor-div-nat* [*THEN sym*])
apply *force*
apply *force*
apply (*subst real-nat-plus-one*)
apply *simp*
done
also have ... = (%*x*. *sumr 0* (*natfloor* (*abs x*) + *1*) (%*d*.
mu2(*d*+*1*) ∗ (*sumr 0* (*natfloor* (*abs*((*abs x* + *1*) / (*real d*+*1*) − *1*)) + *1*)
(%*k*. *ln*(*real k* + *1*) ^ *2*))))
apply (*rule ext*)
apply (*rule sumr-cong*)
apply (*rule arg-cong*)**back**
apply (*subgoal-tac natfloor* ((*abs x* + *1*) / (*real y* + *1*)) =
natfloor (*abs* ((*abs x* + *1*) / (*real y* + *1*) − *1*)) + *1*)
apply (*erule ssubst*, *rule refl*)
apply (*subst natfloor-add* [*THEN sym*])
apply *force*
apply (*rule arg-cong*)**back**
apply (*subst abs-nonneg*)
apply *simp*
apply (*rule div-ge-1*)
apply *force*
apply *auto*
apply (*rule nat-le-natfloor*)
apply *force*
apply *force*

```
  done
also have ... =o (%x. sumr 0 (natfloor (abs x) + 1) (%d. mu2(d+1) *
    ((%y. (abs y + 1) * ln (abs y + 1) ^ 2) -
     (%y. 2 * (abs y + 1) * ln (abs y + 1)) +
     (%y. 2 * (abs y + 1))) ((abs x + 1) / (real d + 1) - 1))) +o
    O(%x. sumr 0 (natfloor (abs x) + 1) (%d. abs( mu2(d+1) *
      (1 + ln(abs ((abs x + 1) / (real d + 1) - 1) + 1)^2))))
    (is ?temp =o ?term1 +o ?oterm)
  by (rule bigo-sumr8 [OF identity-six-real])
also have ?term1 = (%x. sumr 0 (natfloor (abs x) + 1) (%d.
      (mu2(d+1) * (abs x + 1) / (real d + 1) *
        ln((abs x + 1) / (real d + 1))^2) -
      (mu2(d+1) * 2 * ((abs x + 1) / (real d + 1)) *
        ln((abs x + 1) / (real d + 1))) +
      (mu2(d+1) * 2 * ((abs x + 1) / (real d + 1)))))
  apply (simp only: func-plus func-diff)
  apply (rule ext)
  apply (rule sumr-cong)
  apply (subgoal-tac abs ((abs x + 1) / (real y + 1) - 1) + 1 =
     (abs x + 1) / (real y + 1))
  apply (simp add: ring-eq-simps)
  apply (subst abs-nonneg)backback
  apply simp
  apply (rule div-ge-1)
  apply auto
  apply (rule nat-le-natfloor)
  apply auto
  done
also have ... = (%x. (abs x + 1) * sumr 0 (natfloor(abs x) + 1)
      (%d. mu2(d+1) / (real (d + 1)) *
         ln((abs x + 1) / (real (d + 1)))^2)) + -
    (%x. 2 * (abs x + 1) * sumr 0 (natfloor(abs x) + 1)
      (%d. mu2(d+1) / (real (d + 1)) * ln((abs x + 1) / (real (d + 1))))) +
    (%x. 2 * (abs x + 1) * sumr 0 (natfloor(abs x) + 1)
      (%d. mu2(d+1) / (real (d + 1))))
  apply (rule ext)
  apply (simp only: func-plus func-minus)
  apply (simp only: sumr-mult sumr-add sumr-minus [THEN sym])
  apply (rule sumr-cong)
  apply (subst real-nat-plus-one)
  apply (simp add: ring-eq-simps ring-distrib add-divide-distrib)
  done
also have ... = (%x. (abs x + 1)) *
    ((%x. sumr 0 (natfloor(abs x) + 1)
      (%d. mu2(d+1) / (real (d + 1)) *
```

```
        ln((abs x + 1) / (real (d + 1))) ^ 2)) + -
    (%x. 2) * (%x. sumr 0 (natfloor(abs x) + 1)
      (%d. mu2(d+1) / (real (d + 1)) * ln((abs x + 1) / (real (d + 1))))) +
    (%x. 2) * (%x. sumr 0 (natfloor(abs x) + 1)
      (%d. mu2(d+1) / (real (d + 1))))) (is ?temp = ?term1a)
  apply (simp only: func-times func-minus func-plus)
  apply (rule ext)
  apply (simp only: right-distrib right-diff-distrib mult-ac)
  done
also
have (%x. sumr 0 (natfloor (abs x) + 1) (%d. abs( mu2(d + 1) *
      (1 + ln(abs ((abs x + 1) / (real d + 1) - 1) + 1) ^ 2)))) =
    (%x. sumr 0 (natfloor (abs x) + 1) (%d. abs( mu2(d + 1) *
      (1 + ln(((abs x + 1) / (real d + 1))) ^ 2))))
  apply (rule ext)
  apply (rule sumr-cong)
  apply (rule arg-cong)back
  apply (rule arg-cong)back
  apply (rule arg-cong)back
  apply (rule arg-cong)back
  apply (rule arg-cong)back
  apply (subst abs-nonneg)backback
  apply simp
  apply (rule div-ge-1)
  apply force
  apply simp
  apply (rule nat-le-natfloor)
  apply auto
  done
also have ?term1a +o O(...) <= (%x. abs x + 1) *o
    (((%x. 2 * ln(abs x + 1)) +o O(%x. 1)) +
     ((-(%x. 2)) *o O(%x. 1)) +
     ((%x. 2) *o O(%x. 1))) +
    O(%x. sumr 0 (natfloor (abs x) + 1) (%d.
      (1 + ln((abs x + 1) / (real d + 1)) ^ 2)))
  apply (rule set-plus-mono5)
  apply (rule set-times-intro2)
  apply (rule set-plus-intro)
  apply (rule set-plus-intro)
  apply (rule sumr-mu-div-n-times-ln-squared-div-nbigo)
  apply (rule set-times-intro2)
  apply (rule sumr-mu-div-n-times-ln-div-nbigo)
  apply (rule set-times-intro2)
  apply (rule bigo-compose1 [OF sumr-mu-div-n-bigo])
  apply (rule bigo-elt-subset)
```

apply (*rule bigo-bounded*)
apply (*rule allI*)
apply (*rule sumr-ge-zero*)
apply *arith*
apply (*rule allI*)
apply (*rule sumr-le-cong*)
apply (*subst abs-mult*)
apply (*subst abs-nonneg*)**backback**
apply (*rule nonneg-plus-nonneg*)
apply *force*
apply *force*
apply (*rule real-mult-le-lemma*)
apply (*rule abs-mu2-leq-1*)
apply (*rule nonneg-plus-nonneg*)
apply *auto*
done
also have $(-\ (\%x.\ 2))\ *o\ O(\%x.\ 1) = O(\%x.\ (1::real))$
apply (*subst func-minus*)
apply (*rule bigo-const-mult5*)
apply *simp*
done
also have $(\%x.\ 2)\ *o\ O(\%x.\ 1) = O(\%x.\ (1::real))$
by (*rule bigo-const-mult5*, *simp*)
also have $(\%x.\ 2 * ln\ (abs\ x + 1))\ +o\ O(\%x.\ 1) + O(\%x.\ 1) + O(\%x.\ 1) =$
$(\%x.\ 2 * ln\ (abs\ x + 1))\ +o\ (O(\%x.\ 1) + O(\%x.\ 1) + O(\%x.\ 1))$
by (*simp only*: *set-plus-rearranges*)
also have $O(\%x.\ 1) + O(\%x.\ 1) + O(\%x.\ 1) = O(\%x.\ (1::real))$
apply (*subst bigo-plus-idemp*)+
apply (*rule refl*)
done
also have $(\%x.\ abs\ x + 1)\ *o\ ((\%x.\ 2 * ln\ (abs\ x + 1))\ +o\ O(\%x.\ 1)) =$
$(\%x.\ 2 * (abs\ x + 1) * ln\ (abs\ x + 1))\ +o\ (\%x.\ abs\ x + 1)\ *o\ O(\%x.\ 1)$
by (*simp only*: *set-times-plus-distribs func-times mult-ac*)
also have ... + $O(\%x.\ sumr\ 0\ (natfloor\ (abs\ x) + 1)$
$(\%d.\ 1 + ln\ ((abs\ x + 1)\ /\ (real\ d + 1))\ \hat{}\ 2)) <=$
$(\%x.\ 2 * (abs\ x + 1) * ln\ (abs\ x + 1))\ +o\ O(\%x.\ abs\ x + 1)$
apply (*simp only*: *set-plus-rearranges*)
apply (*rule set-plus-mono*)
apply (*rule bigo-useful-intro*)
apply (*rule order-trans*)
apply (*rule bigo-mult2*)
apply (*simp add*: *func-times*)
apply (*subgoal-tac* $(\%x.\ sumr\ 0\ (natfloor\ (abs\ x) + 1)$
$(\%d.\ 1 + ln\ ((abs\ x + 1)\ /\ (real\ d + 1))\ \hat{}\ 2)) =$
$(\%x.\ sumr\ 0\ (natfloor\ (abs\ x) + 1)\ (\%d.\ 1)) +$

```
      (%x. sumr 0 (natfloor (abs x) + 1)
        (%d. ln ((abs x + 1) / (real d + 1)) ^ 2)))
   apply (erule ssubst)
   apply (rule order-trans)
   apply (rule bigo-plus-subset)
   apply (rule bigo-useful-intro)
   apply (rule bigo-elt-subset)
   apply simp
   apply (rule bigo-bounded)
   apply force
   apply (rule allI)
   apply simp
   apply (rule real-natfloor-le)
   apply force
   apply (rule bigo-elt-subset)
   apply (rule sum-ln-x-div-x-squared-real-bigo-cor)
   apply (simp only: func-plus sumr-add)
   done
 finally show ?thesis.
qed

lemma Selberg3: (%x. ∑ n = 1..natfloor (abs x) + 1.
    Lambda n * ln (real n))+ (%x. ∑ n=1..natfloor (abs x) + 1.
     (∑ u | u dvd n. Lambda u * Lambda (n div u)))
      =o (%x. 2 * (abs x + 1) * ln (abs x + 1)) +o O(%x. abs x + 1)
       (is ?LHS =o ?RHS)
 apply (subgoal-tac ?LHS = ?temp)
 apply (erule ssubst)
 apply (rule Selberg2)
 apply (subst func-plus)
 apply (rule ext)
 apply (subst setsum-addf)
 apply (rule refl)
done

lemma sum-lambda-ln-bigo:
    (%x. ∑ n = 1..x + 1. Lambda n * ln (real n)) =o
      (%x. psi (x + 1) * ln (real (x + 1))) +o O(%x. real x)
proof -
 have (%x. ∑ n=1..x+1. Lambda n * ln (real n)) =
      (%x. ∑ n=1..x+1. (psi n - psi (n - 1)) * ln (real n))
   apply (rule ext)
   apply (rule setsum-cong2)
   apply (simp add: psi-diff2)
   done
```

also have ... = (%x. *psi* (x + 1) * *ln* (*real* (x + 1))) + −
(%x. $\sum$ n = 1..x. *psi* n * (*ln* (*real* (n + 1)) − *ln* (*real* n)))
apply (*simp only*: *func-minus func-plus*)
apply (*rule ext*)
apply (*subst partial-sum-b0*)
apply *clarify*
apply (*erule telescoping-sum* [*THEN sym*])
apply (*simp add*: *psi-zero*)
done
also have ... =*o* (%x. *psi*(x + 1) * *ln*(*real* (x + 1))) +*o*
O(%x. *sumr 0 x* (%i. *1*))
apply (*rule set-plus-intro2*)
apply (*rule bigo-minus*)
apply (*subgoal-tac* (%x. $\sum$ n = 1..x. *psi* n *
(*ln* (*real* (n + 1)) − *ln* (*real* n))) = (%x. *?s* x))
prefer *2*
apply (*rule ext*)
apply (*rule setsum-sumr4*)
apply (*erule ssubst*)
apply (*rule bigo-sumr-pos*)
apply *simp*
apply (*subgoal-tac* (%i. *psi* (i + 1) *
(*ln* (*real* (i + 1 + 1)) − *ln* (*real* (i + 1)))) =
(%i. *psi* (i + 1)) * (%i. (*ln* (*real* (i + 1 + 1)) − *ln* (*real* (i + 1))))))
apply (*erule ssubst*)
apply (*subgoal-tac O*(%i. *real* (i + 1)) * *O*(%i. *1* / (*real* i + 1)) =
O(%i. *1*))
apply (*erule subst*)
apply (*rule set-times-intro*)
apply (*rule bigo-compose1*)
apply (*rule psi-bigo*)
apply (*subgoal-tac* (%i. *ln* (*real* (i + 1 + 1)) − *ln* (*real* (i + 1))) =
(%i. *ln*(*1* + *1* / (*real* i + *1*))))
apply (*erule ssubst*)
apply (*rule ln-one-plus-one-over-x-bigo*)
apply (*rule ext*)
apply (*subst ln-div* [*THEN sym*])
apply *force*
apply *force*
apply (*rule arg-cong*)**back**
apply (*subgoal-tac real* (i + 1 + 1) = *real* (i + 1) + 1)
apply (*erule ssubst*)
apply (*subst add-divide-distrib*)
apply *simp*
apply *simp*

```
    apply (subst bigo-mult8 [THEN sym])
    apply force
    apply (subst func-times)
    apply (subgoal-tac (%u. real (u + 1) * (1 / (real u + 1))) = (%i. 1))
    apply (erule ssubst)
    apply simp
    apply (rule ext)
    apply (subgoal-tac real (u + 1) = real u + 1)
    apply (erule ssubst)
    apply simp
    apply simp
    apply (subst func-times, rule refl)
    done
  also have ... <= (%x. psi(x + 1) * ln(real (x + 1))) +o O(%x. real x)
    by simp
  finally show ?thesis.
qed

lemma sum-lambda-ln-bigo-real:
    (%x. ∑ n = 1..natfloor (abs x) + 1. Lambda n * ln (real n)) =o
      (%x. psi (natfloor (abs x) + 1) * ln (abs x + 1)) +o O(%x. abs x + 1)
proof -
  have (%x. ∑ n = 1..natfloor (abs x) + 1. Lambda n * ln (real n)) =o
      (%x. psi (natfloor (abs x) + 1) * ln (real (natfloor (abs x) + 1))) +o
        O(%x. real (natfloor (abs x)))
    by (rule bigo-compose2 [OF sum-lambda-ln-bigo])
  also have (%x. psi (natfloor (abs x) + 1) *
    ln (real (natfloor (abs x) + 1))) =
      (%x. psi (natfloor (abs x) + 1) * ln(abs x + 1)) +
      (%x. psi (natfloor (abs x) + 1)) * (%x. (ln(real(natfloor (abs x) + 1)) -
        (ln(abs x + 1))))
    apply (simp only: func-plus func-times)
    apply (rule ext)
    apply (subst right-diff-distrib)
    apply simp
    done
  also have ((%x. psi (natfloor (abs x) + 1) * ln (abs x + 1)) +
    (%x. psi (natfloor (abs x) + 1)) *
    (%x. ln (real (natfloor (abs x) + 1)) - ln (abs x + 1))) +o
      O(%x. real (natfloor (abs x))) =
    (%x. psi (natfloor (abs x) + 1) * ln (abs x + 1)) +o
     (((%x. psi (natfloor (abs x) + 1)) *
    (%x. ln (real (natfloor (abs x) + 1)) - ln (abs x + 1))) +o
      O(%x. real (natfloor (abs x))))
    by (simp add: set-plus-rearranges)
```

also have ... <= (%x. *psi* (*natfloor* (*abs x*) + *1*) * *ln* (*abs x* + *1*)) +*o*
(*O*(%x. *real* (*natfloor*(*abs x*) + *1*)) * *O*(%x. *1* / (*abs x* + *1*)) +
O(%x. *abs x* + *1*))
apply (*rule set-plus-mono*)
apply (*rule set-plus-mono5*)
apply (*rule set-times-intro*)
apply (*rule bigo-compose1* [*OF psi-bigo*])
apply (*subst func-diff* [*THEN sym*])**back**
apply (*rule set-plus-imp-minus*)
apply (*rule bigo-add-commute-imp*)
apply (*subgoal-tac* (%x. *ln* (*real* (*natfloor* (*abs x*) + *1*))) =
(%x. *ln* (*real* (*natfloor* (*abs x*)) + *1*)))
apply (*erule ssubst*)
apply (*rule ln-real-approx-ln-nat*)
apply (*rule ext*)
apply *simp*
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply (*rule allI*)
apply (*rule order-trans*)
apply (*rule real-natfloor-le*)
apply *auto*
done
also have ... <= (%x. *psi* (*natfloor* (*abs x*) + *1*) * *ln* (*abs x* + *1*)) +*o*
(*O*(%x. (*abs x* + *1*)) * *O*(%x. *1*) + *O*(%x. *abs x* + *1*))
apply (*rule set-plus-mono*)
apply (*rule set-plus-mono2*)
apply (*rule set-times-mono2*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply (*rule allI*)
apply (*subgoal-tac real*(*natfloor*(*abs x*)) <= *abs x*)
apply *force*
apply (*rule real-natfloor-le*)
apply *force*
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply (*rule allI*)
apply (*rule real-ge-zero-div-gt-zero*)
apply *force*
apply *arith*
apply (*rule allI*)
apply (*rule real-le-mult-imp-div-pos-le*)

```
    apply arith
    apply force
    apply force
    done
  also have ... <= (%x. psi (natfloor (abs x) + 1) * ln (abs x + 1)) +o
      O(%x. (abs x + 1))
    apply (rule set-plus-mono)
    apply (rule bigo-useful-intro)
    apply (subst bigo-mult8 [THEN sym])
    apply (rule allI)
    apply arith
    apply (simp add: func-times)
    apply force
    done
  finally show ?thesis.
qed

lemma Selberg4: (%x. psi (natfloor (abs x) + 1) * ln (abs x + 1)) +
    (%x. ∑ n = 1..natfloor (abs x) + 1.
       ∑ u | u dvd n. Lambda u * Lambda (n div u))
    =o (%x. 2 * (abs x + 1) * ln (abs x + 1)) +o O(%x. abs x + 1)
     (is ?LHS1 + ?LHS2 =o ?RHS)
proof -
  have (%x. ∑ n = 1..natfloor (abs x) + 1. Lambda n * ln (real n)) +
    (%x. ∑ n = 1..natfloor (abs x) + 1.
      ∑ u | u dvd n. Lambda u * Lambda (n div u)) =o ?RHS
       (is ?LHS0 + ?LHS2 =o ?RHS)
    by (rule Selberg3)
  then have (?LHS1 - ?LHS0) + (?LHS0 + ?LHS2) =o O(%x. abs x + 1) +
?RHS
    apply (intro set-plus-intro)
    apply (rule set-plus-imp-minus)
    apply (rule bigo-add-commute-imp)
    apply (rule sum-lambda-ln-bigo-real)
    apply assumption
    done
  also have ... <= ?RHS
    by (simp add: set-plus-rearranges)
  also have (?LHS1 - ?LHS0) + (?LHS0 + ?LHS2) = ?LHS1 + ?LHS2
    by (simp add: ring-eq-simps)
  finally show ?thesis.
qed

lemma Selberg4a: (%x. psi (natfloor (abs x) + 1) * ln (abs x + 1)) +
    (%x. ∑ d = 1..natfloor (abs x) + 1.
```

```
      Lambda d * psi((natfloor(abs x) + 1) div d))
   =o (%x. 2 * (abs x + 1) * ln (abs x + 1)) +o O(%x. abs x + 1)
proof -
  note Selberg4
  also have (%x. ∑ n = 1..natfloor (abs x) + 1.
      ∑ u | u dvd n. Lambda u * Lambda (n div u)) =
     (%x. ∑ d = 1..natfloor (abs x) + 1. Lambda d *
        psi((natfloor(abs x) + 1) div d))
    apply (rule ext)
    apply (subst general-inversion-nat2-modified [THEN sym])
    apply force
    apply (rule setsum-cong2)
    apply (subst psi-def-alt)
    apply (subst setsum-const-times [THEN sym])
    apply (rule setsum-cong2)
    apply simp
    done
  finally show ?thesis.
qed


lemma aux: 1 <= (z::real) ==> natfloor(abs(z - 1)) + 1 = natfloor z
  apply (subst natfloor-add [THEN sym])
  apply force
  apply simp
done

lemma aux2: (1::real) <= z ==> abs(z - 1) + 1 = z
  by simp

lemma sum-lambda-m-over-m-ln-x-over-m-bigo:
  (%x. ∑ m = 1..natfloor (abs x) + 1.
      Lambda m / real m * ln ((abs x + 1) / real m)) =o
    (%x. ln (abs x + 1) ^ 2 / 2) +o O(%x. 1 + ln (abs x + 1))
proof -
  have (%x. ∑ m = 1..natfloor (abs x) + 1.
        Lambda m / real m * ln ((abs x + 1) / real m)) =
    (%x. ∑ m = 1..natfloor (abs x) + 1.
        Lambda m / real m * ln (abs((abs x + 1) / real m - 1) + 1))
    apply (rule ext)
    apply (rule setsum-cong2)
    apply (subst aux2)
    apply (rule div-ge-1)
    apply force
    apply (subgoal-tac real(natfloor(abs x)) <= abs x)
```

apply *force*
apply (*rule real-natfloor-le*)
apply *auto*
done
also have ... =*o* (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m / *real m* ∗
(∑ *i*=*1*..*natfloor*(*abs*((*abs x* + *1*) / *real m* − *1*))+*1*.
1 / (*real i*))) +*o*
O(%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
abs(*Lambda m* / *real m* ∗ *1*))
by (*rule bigo-setsum4* [*OF ln-sum-real2*])
also have (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m / *real m* ∗
(∑ *i*=*1*..*natfloor*(*abs*((*abs x* + *1*) / *real m* − *1*))+*1*.
1 / (*real i*))) =
(%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
(∑ *i*=*1*..*natfloor*((*abs x* + *1*) / *real m*).
Lambda m / (*real* (*m* ∗ *i*))))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*subgoal-tac natfloor*(*abs* ((*abs x* + *1*) / *real xa* − *1*)) + *1*
= *natfloor*((*abs x* + *1*) / *real xa*))
apply (*erule ssubst*)
apply (*rule setsum-cong2*)
apply *force*
apply (*subst natfloor-add* [*THEN sym*])
apply *force*
apply (*subst real-nat-one*)
apply (*subst aux2*)
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real*(*natfloor*(*abs x*)) <= *abs x*)
apply *force*
apply (*rule real-natfloor-le*)
apply *force*
apply (*rule refl*)
done
also have ... = (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
$\sum$ *i* = *1*..(*natfloor* (*abs x*) + *1*) *div m*.
Lambda m / *real* (*m* ∗ *i*))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst natfloor-div-nat*)
apply *force*

apply *force*
apply (*subst natfloor-add* [*THEN sym*])
apply *force*
apply *simp*
done
also have ... = (%*x*. $\sum$ *c*=*1*..*natfloor*(*abs x*) + *1*.
$\sum$ *m* | *m dvd c*. *Lambda m* / *real* (*m* ∗ (*c div m*)))
apply (*rule ext*)
apply (*rule general-inversion-nat2-modified*)
apply *force*
done
also have ... = (%*x*. $\sum$ *c*=*1*..*natfloor*(*abs x*) + *1*.
$\sum$ *m* | *m dvd c*. *Lambda m* / (*real c*))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*rule setsum-cong2*)
apply (*subst dvd-mult-div-cancel*)
apply *auto*
done
also have ... = (%*x*. $\sum$ *c*=*1*..*natfloor*(*abs x*) + *1*.
1 / (*real c*) ∗ ($\sum$ *m* | *m dvd c*. *Lambda m*))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
done
also have ... = (%*x*. $\sum$ *c*=*1*..*natfloor*(*abs x*) + *1*.
ln (*real c*) / (*real c*))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst ln-eq-setsum-Lambda* [*THEN sym*])
apply *force*
apply *simp*
done
also have (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
abs (*Lambda m* / *real m* ∗ *1*)) =
(%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m / *real m*)
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst abs-nonneg*)
apply *simp*
apply (*rule real-ge-zero-div-gt-zero*)
apply (*rule Lambda-ge-zero*)

```
    apply force
    apply simp
    done
  finally have   (%x. ∑ m = 1..natfloor (abs x) + 1.
      Lambda m / real m * ln ((abs x + 1) / real m)) =o
    (%x. ∑ c = 1..natfloor (abs x) + 1. ln (real c) / real c) +o
      O(%x. ∑ m = 1..natfloor (abs x) + 1.
        Lambda m / real m).
  also have ... <= ((%x. ln(abs x + 1) ^ 2 / 2) +o O(%x. 1)) +
      (O(%x. ln (abs x + 1)) + O(%x. 1))
    apply (rule set-plus-mono5)
    apply (rule identity-four-real-b-cor)
    apply (rule bigo-elt-subset2)
    apply (rule Mertens-theorem-real2)
    done
  also have ... <= (%x. ln(abs x + 1) ^ 2 / 2) +o O(%x. 1 + ln (abs x + 1))
    apply (simp add: set-plus-rearranges)
    apply (rule set-plus-mono)
    apply (rule bigo-useful-intro)
    apply (rule bigo-one-subset-bigo-one-plus-ln)
    apply (rule bigo-useful-intro)
    apply (rule bigo-ln-subset-bigo-one-plus-ln)
    apply (rule bigo-one-subset-bigo-one-plus-ln)
    done
  finally show ?thesis.
qed

lemma bigo-pos-mono: ALL x. 0 <= f x ==>
    ALL x. 0 <= g x ==> f =o O(f + g)
  apply (erule bigo-bounded)
  apply (auto simp add: func-plus)
  apply (subgoal-tac f x + 0 <= f x + g x)
  apply simp
  apply (rule add-left-mono)
  apply (erule spec)
done

lemma Mertens-real-cor: (%x. ∑ m = 1..natfloor (abs x) + 1.
    Lambda m / real m) =o O(%x. 1 + ln (abs x + 1))
  apply (rule subsetD)
  prefer 2
  apply (rule Mertens-theorem-real2)
  apply (rule bigo-plus-absorb2)
  apply (rule subsetD)
  prefer 2
```

```
  apply (rule bigo-pos-mono)
  apply force
  apply (subgoal-tac ALL x. 0 <= (%x. 1) x)
  apply assumption
  apply force
  apply (simp add: func-plus)
  prefer 2
  apply (rule bigo-elt-subset)
  apply (rule bigo-bounded)
  apply force
  apply force
  apply (subgoal-tac (%x. ln (abs x + 1) + 1) = (%x. 1 + ln (abs x + 1)))
  apply simp
  apply (rule ext)
  apply simp
done

lemma Selberg5: (%x. ∑ m = 1..natfloor (abs x) + 1.
    ∑ n = 1..(natfloor (abs x) + 1) div m.
      Lambda m * Lambda n * ln (real n)) +
    (%x. ∑ m = 1..natfloor (abs x) + 1.
       ∑ n = 1..(natfloor (abs x) + 1) div m.
         ∑ u | u dvd n. Lambda m * Lambda u * Lambda (n div u)) =o
    (%x. (abs x + 1) * ln (abs x + 1) ^ 2) +o
      O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
proof -
  have (%x. ∑m=1..natfloor(abs x) + 1.
    ∑n=1..(natfloor(abs x) + 1) div m.
      (Lambda m) * (Lambda n) * ln (real n)) +
        (%x. ∑m=1..natfloor(abs x) + 1.
          ∑n=1..(natfloor(abs x) + 1) div m.
            ∑u | u dvd n. (Lambda m) * (Lambda u) * (Lambda (n div u))) =
        (%x. ∑m=1..natfloor(abs x) + 1. Lambda m *
          (∑n=1..natfloor(abs (
                                (abs x + 1) / (real m) - 1)) + 1.
             Lambda n * ln (real n) +
               (∑ u | u dvd n. Lambda u * Lambda (n div u))))
    apply (subst func-plus)
    apply (rule ext)
    apply (subst setsum-addf [THEN sym])
    apply (rule setsum-cong2)
    apply (subst setsum-addf [THEN sym])
    apply (subst setsum-const-times [THEN sym])
    apply (subst aux)
    apply (rule div-ge-1)
```

apply *force*
apply (*subgoal-tac real* (*natfloor* (*abs x*)) <= *abs x*)
apply *force*
apply (*rule real-natfloor-le*)
apply *force*
apply (*subst natfloor-div-nat*)
apply *force*
apply *force*
apply (*subst natfloor-add* [*THEN sym*])
apply *simp*
apply (*rule setsum-cong*)
apply *simp*
apply (*simp add*: *ring-eq-simps*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
done
also have ... =*o* (%*x*. $\sum$ *m*=*1*..*natfloor* (*abs x*) + *1*.
Lambda m ∗ (*2* ∗ (*abs*(((*abs x* + *1*) / *real m*) − *1*) + *1*) ∗
ln (*abs*(((*abs x* + *1*) / *real m*) − *1*) + *1*))) +*o*
O(%*x*. $\sum$ *m*=*1*..*natfloor* (*abs x*) + *1*. *Lambda m* ∗
(*abs*(((*abs x* + *1*) / *real m*) − *1*) + *1*))
(**is** *?temp* =*o* *?term1* +*o* *O*(*?term2*::*real*=>*real*))
apply (*rule bigo-setsum6* [*OF Selberg2*])
apply (*rule allI*)+
apply (*rule Lambda-ge-zero*)
apply (*rule allI*)
apply *arith*
done
also have *?term1* = (%*x*. $\sum$ *m*=*1*..*natfloor* (*abs x*) + *1*.
Lambda m ∗ (*2* ∗ (((*abs x* + *1*) / *real m*)) ∗
ln (((*abs x* + *1*) / *real m*))))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst aux2*)
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real* (*natfloor*(*abs x*)) <= *abs x*)
apply *force*
apply (*rule real-natfloor-le*)
apply *force*
apply (*rule refl*)
done
also have *?term2* = (%*x*. $\sum$ *m*=*1*..*natfloor* (*abs x*) + *1*. *Lambda m* ∗
(((*abs x* + *1*) / *real m*)))

apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst aux2*)
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real* (*natfloor*(*abs x*)) <= *abs x*)
apply *force*
apply (*rule real-natfloor-le*)
apply *force*
apply (*rule refl*)
done
also have (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*. *Lambda m* *
(*2* * ((*abs x* + *1*) / *real m*) * *ln* ((*abs x* + *1*) / *real m*))) =
(%*x*. *2* * (*abs x* + *1*)) * (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
(*Lambda m* / *real m*) * *ln* ((*abs x* + *1*) / *real m*))
(**is** *?temp* = *?term3*)
apply (*subst func-times*)
apply (*rule ext*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply (*simp only*: *times-divide-eq-left times-divide-eq-right mult-ac*)
done
also have (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m * ((*abs x* + *1*) / *real m*)) =
(%*x*. (*abs x* + *1*)) * (%*x*. ($\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m / *real m*))
(**is** *?temp* = *?term4*)
apply (*subst func-times*)
apply (*rule ext*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply (*simp only*: *times-divide-eq-right mult-ac*)
done
also have *?term3* +*o* *O*(*?term4*) <= ((%*x*. *2* * (*abs x* + *1*)) *o
((%*x*. (*ln* (*abs x* + *1*)) ^ *2* / *2*) +*o* *O*(%*x*. *1* + *ln*(*abs x* + *1*)))) +
(*O*(%*x*. *abs x* + *1*) * *O*((%*x*. *1* + *ln*(*abs x* + *1*))))
apply (*rule set-plus-mono5*)
apply (*rule set-times-intro2*)
prefer *2*
apply (*rule order-trans*)
apply (*rule bigo-mult7*)
apply *arith*
apply (*rule set-times-mono2*)
apply *force*
apply (*rule bigo-elt-subset*)

```
    apply (rule Mertens-real-cor)
    apply (rule sum-lambda-m-over-m-ln-x-over-m-bigo)
    done
  also have ... <= (%x. (abs x + 1) * (ln (abs x + 1)) ^2) +o
      O(%x. (abs x + 1) * (1 + ln(abs x + 1)))
    apply (simp only: func-times set-times-plus-distribs
      set-plus-rearranges plus-ac0 mult-ac)
    apply (subgoal-tac (%x. (1 + abs x) * (2 * (ln (1 + abs x) ^ 2 / 2))) =
        (%x. (1 + abs x) * (ln (1 + abs x) ^ 2)))
    apply (erule ssubst)
    apply (rule set-plus-mono)
    apply (rule bigo-useful-intro)
    apply (rule subset-trans)
    apply (rule bigo-mult)
    apply (subst func-times)
    apply simp
    apply (subgoal-tac (%x. (1 + abs x) * 2) *o O(%x. 1 + ln (1 + abs x)) =
      (%x. 2) *o ((%x. (1 + abs x)) *o O(%x. 1 + ln (1 + abs x))))
    apply (erule ssubst)
    apply (subgoal-tac O(%x. (1 + abs x) * (1 + ln (1 + abs x))) =
      (%x. 2) *o O(%x. (1 + abs x) * (1 + ln (1 + abs x))) )
    apply (erule ssubst)
    apply (rule set-times-mono)
    apply (rule subset-trans)
    apply (rule bigo-mult2)
    apply (simp add: func-times)
    apply (rule bigo-const-mult5 [THEN sym])
    apply force
    apply (simp add: set-times-rearranges func-times mult-ac)
    apply (rule ext)
    apply simp
    done
  finally show ?thesis.
qed

lemma Selberg6: (%x. ∑ m = 1..natfloor (abs x) + 1.
    ∑ n = 1..(natfloor (abs x) + 1) div m.
      Lambda m * Lambda n * ln (real n)) =o
  (%x. ln (abs x + 1) / 2 *
    (∑ m = 1..natfloor (abs x) + 1.
      Lambda m * psi (natfloor ((abs x + 1) / real m)))) +o
  O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
proof -
  have (%x. ∑ m = 1..natfloor (abs x) + 1.
    ∑ n = 1..(natfloor (abs x) + 1) div m.
```

*Lambda m * Lambda n * ln (real n)) =*
*(%x. ∑ m = 1..natfloor (abs x) + 1. Lambda m **
(∑ n = 1..(natfloor (abs x) + 1) div m.
*Lambda n * ln (real n)))*
apply *(rule ext)*
apply *(rule setsum-cong2)*
apply *(subst setsum-const-times [THEN sym])*
apply *(rule setsum-cong2)*
apply *simp*
done
also have *... =*
*(%x. ∑ m = 1..natfloor(abs x) + 1. Lambda m **
(∑ n = 1..natfloor(abs((abs x + 1) / (real m) − 1)) + 1.
*Lambda n * ln (real n)))*
apply *(rule ext)*
apply *(rule setsum-cong2)*
apply *(subst aux)*
apply *(rule div-ge-1)*
apply *force*
apply *(subgoal-tac real(natfloor(abs x)) <= abs x)*
apply *force*
apply *(rule real-natfloor-le)*
apply *force*
apply *(subst natfloor-div-nat)*
apply *force*
apply *force*
apply *(subst natfloor-add [THEN sym])*
apply *force*
apply *simp*
done
also have *... =o (%x. ∑ m = 1..natfloor (abs x) + 1.*
*Lambda m * (psi(natfloor(abs ((abs x + 1) / real m − 1)) + 1) **
ln (abs ((abs x + 1) / real m − 1) + 1))) +o
*O(%x. ∑ m = 1..natfloor (abs x) + 1. Lambda m **
((abs ((abs x + 1) / real m − 1)) + 1))
*(***is** *?temp =o ?term1 +o O(?term2::real=>real))*
apply *(rule bigo-setsum6 [OF sum-lambda-ln-bigo-real])*
apply *(rule allI)+*
apply *(rule Lambda-ge-zero)*
apply *(rule allI)*
apply *arith*
done
also have *?term1 = (%x. ∑ m = 1..natfloor (abs x) + 1.*
*Lambda m * (psi(natfloor((abs x + 1) / real m)) **
ln ((abs x + 1) / real m)))

apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst aux*)
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real*(*natfloor*(*abs x*)) <= *abs x*)
apply *force*
apply (*rule real-natfloor-le*)
apply *force*
apply (*subst aux2*)
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real*(*natfloor*(*abs x*)) <= *abs x*)
apply *force*
apply (*rule real-natfloor-le*)
apply *force*
apply (*rule refl*)
done
also have *?term2* = (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*. *Lambda m* ∗
((*abs x* + *1*) / *real m*))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst aux2*)
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real*(*natfloor*(*abs x*)) <= *abs x*)
apply *force*
apply (*rule real-natfloor-le*)
apply *force*
apply (*rule refl*)
done
also have (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m ∗ (*psi* (*natfloor* ((*abs x* + *1*) / *real m*)) ∗
ln ((*abs x* + *1*) / *real m*))) =
(%*x*. *ln* (*abs x* + *1*) ∗ ($\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m ∗ *psi* (*natfloor* ((*abs x* + *1*) / *real m*)))) −
(%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m ∗ *ln*(*real m*) ∗ *psi* (*natfloor* ((*abs x* + *1*) / *real m*)))
apply (*subst func-diff*)
apply (*rule ext*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*subst setsum-subtractf'* [*THEN sym*])
apply (*rule setsum-cong2*)
apply (*subst ln-div*)
apply *arith*

apply *force*
apply (*simp add*: *ring-eq-simps*)
done
also have (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m ∗ *ln*(*real m*) ∗ *psi* (*natfloor* ((*abs x* + *1*) / *real m*))) =
(%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m ∗ *ln*(*real m*) ∗ ($\sum$ *n* = *1*..(*natfloor* (*abs x*) + *1*) *div m*.
Lambda n))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst psi-def-alt*)
apply (*subst natfloor-div-nat*)
apply *force*
apply *force*
apply (*subst natfloor-add* [*THEN sym*])
apply *force*
apply *simp*
done
also have ... = (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
$\sum$ *n* = *1*..(*natfloor* (*abs x*) + *1*) *div m*.
Lambda m ∗ *Lambda n* ∗ *ln* (*real n*))
apply (*rule ext*)
apply (*subst general-inversion-nat2-cor1-modified*)
apply *force*
apply (*rule setsum-cong2*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
done
also have (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m ∗ ((*abs x* + *1*) / *real m*)) =
(%*x*. *abs x* + *1*) ∗ (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m / *real m*)
apply (*subst func-times*)
apply (*rule ext*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
done
finally have (%*x*. $\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
$\sum$ *n* = *1*..(*natfloor* (*abs x*) + *1*) *div m*.
Lambda m ∗ *Lambda n* ∗ *ln* (*real n*)) =*o*
((%*x*. *ln* (*abs x* + *1*) ∗
($\sum$ *m* = *1*..*natfloor* (*abs x*) + *1*.
Lambda m ∗ *psi* (*natfloor* ((*abs x* + *1*) / *real m*)))) −

```
  (%x. ∑ m = 1..natfloor (abs x) + 1.
      ∑ n = 1..(natfloor (abs x) + 1) div m.
      Lambda m * Lambda n * ln (real n))) +o
  O((%x. abs x + 1) * (%x. ∑ m = 1..natfloor (abs x) + 1.
      Lambda m / real m))
    (is ?LHS =o ?RHS +o ?temp).
also have ... <= ?RHS +o ((%x. abs x + 1) *o
    O(%x. ∑ m = 1..natfloor (abs x) + 1. Lambda m / real m))
  apply (rule set-plus-mono)
  apply (rule bigo-mult5)
  apply arith
  done
also have ... <= ?RHS +o ((%x. abs x + 1) *o O(%x. 1 + ln (abs x + 1)))
  apply (rule set-plus-mono)
  apply (rule set-times-mono)
  apply (rule bigo-elt-subset)
  apply (rule Mertens-real-cor)
  done
also have ... <= ?RHS +o O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
    (is ?temp <= ?RHS +o ?Oterm)
  apply (rule set-plus-mono)
  apply (rule subset-trans)
  apply (rule bigo-mult2)
  apply (simp add: func-times)
  done
finally have ?LHS =o ?RHS +o ?Oterm.
then have ?LHS + ?LHS =o ?LHS +o (?RHS +o ?Oterm)
  by (rule set-plus-intro2)
also have ?LHS + ?LHS = (%x. 2) * ?LHS
  by (simp add: func-plus func-times)
also have ?LHS +o (?RHS +o ?Oterm) =
    (%x. ln (abs x + 1) *
        (∑ m = 1..natfloor (abs x) + 1.
        Lambda m * psi (natfloor ((abs x + 1) / real m)))) +o
    O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
      (is ?temp = ?RHS2)
  by (simp add: set-plus-rearranges ring-eq-simps)
finally have (%x. 1 / 2) * ((%x. 2) * ?LHS) =o (%x. 1 / 2) *o ?RHS2
  by (rule set-times-intro2)
also have (%x. 1 / 2) * ((%x. 2) * ?LHS) = ?LHS
  by (simp add: func-times)
also have (%x. 1 / 2) *o ?RHS2 = (%x. ln (abs x + 1) / 2 *
        (∑ m = 1..natfloor (abs x) + 1.
        Lambda m * psi (natfloor ((abs x + 1) / real m)))) +o
    (%x. 1 / 2) *o O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
```

```
    by (simp add: set-times-plus-distribs func-times)
  also have ... <= (%x. ln (abs x + 1) / 2 *
          (∑ m = 1..natfloor (abs x) + 1.
           Lambda m * psi (natfloor ((abs x + 1) / real m)))) +o
      O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
    apply (rule set-plus-mono)
    apply (rule bigo-const-mult6)
    done
  finally show ?thesis.
qed

lemma Selberg6a: (%x. ∑ m = 1..natfloor (abs x) + 1.
    ∑ n = 1..(natfloor (abs x) + 1) div m.
      Lambda m * Lambda n * ln (real n)) =o
    (%x. ln (abs x + 1) / 2 *
     (∑ n = 1..natfloor (abs x) + 1.
      ∑ u | u dvd n. Lambda u * Lambda (n div u))) +o
  O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
proof -
  note Selberg6
  also have (%x. ln (abs x + 1) / 2 *
    (∑ m = 1..natfloor (abs x) + 1.
      Lambda m * psi (natfloor ((abs x + 1) / real m)))) =
    (%x. ln (abs x + 1) / 2 *
     (∑ n = 1..natfloor (abs x) + 1.
      ∑ u | u dvd n. Lambda u * Lambda (n div u)))
    apply (rule ext)
    apply (rule arg-cong)back
    apply (subst general-inversion-nat2-modified [THEN sym])
    apply force
    apply (rule setsum-cong2)
    apply (subst setsum-const-times)
    apply (rule arg-cong)back
    apply (subst psi-def-alt)
    apply (subst natfloor-div-nat)
    apply force
    apply force
    apply (subst natfloor-add [THEN sym])
    apply simp
    apply simp
    done
  finally show ?thesis.
qed

lemma aux4: f + g =o h +o O(k::'a=>('b::ordered-ring)) ==>
```

```
    f =o l +o O(k) ==> l + g =o h +o O(k)
  apply (rule set-minus-imp-plus)
  apply (drule set-plus-imp-minus)+
  apply (drule bigo-minus)
  apply (subgoal-tac (f - l) + - (f + g - h) =o O(k) + O(k))
  apply simp
  apply (subgoal-tac l + g - h = -(f - l + (h - (f + g))))
  apply (erule ssubst)
  apply (erule bigo-minus)
  apply (simp add: ring-eq-simps)
  apply (erule set-plus-intro)
  apply assumption
done

lemma Selberg7: (%x. ln (abs x + 1) / 2 *
    (∑ m = 1..natfloor (abs x) + 1.
      Lambda m * psi (natfloor ((abs x + 1) / real m)))) +
    (%x. ∑ m = 1..natfloor (abs x) + 1.
       ∑ n = 1..(natfloor (abs x) + 1) div m.
         ∑ u | u dvd n. Lambda m * Lambda u * Lambda (n div u)) =o
    (%x. (abs x + 1) * ln (abs x + 1) ^ 2) +o
      O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
  by (rule aux4 [OF Selberg5, OF Selberg6])

lemma Selberg7a: (%x. ln (abs x + 1) / 2 *
     (∑ n = 1..natfloor (abs x) + 1.
      ∑ u | u dvd n. Lambda u * Lambda (n div u))) +
    (%x. ∑ m = 1..natfloor (abs x) + 1.
       ∑ n = 1..(natfloor (abs x) + 1) div m.
         ∑ u | u dvd n. Lambda m * Lambda u * Lambda (n div u)) =o
    (%x. (abs x + 1) * ln (abs x + 1) ^ 2) +o
      O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
  by (rule aux4 [OF Selberg5, OF Selberg6a])

lemma aux5: f + g =o h +o O(k::'a=>('b::ordered-ring)) ==>
  g + l =o h +o O(k) ==> f =o l +o O(k)
  apply (rule set-minus-imp-plus)
  apply (drule set-plus-imp-minus)+
  apply (subgoal-tac f - l = (f + g - h) + -(g + l - h))
  apply (erule ssubst)
  apply (subgoal-tac O(k) = O(k) + O(k))
  apply (erule ssubst)
  apply (rule set-plus-intro)
  apply assumption
  apply (erule bigo-minus)
```

```
  apply simp
  apply (simp add: ring-eq-simps)
done

lemma Selberg8: (%x. psi (natfloor (abs x) + 1) * ln (abs x + 1) ^ 2) =o
      (%x. 2 * (∑ m = 1..natfloor (abs x) + 1.
        ∑ n = 1..(natfloor (abs x) + 1) div m.
          ∑ u | u dvd n. Lambda m * Lambda u * Lambda (n div u))) +o
      O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
proof -
  note Selberg4
  then have (%x. ln(abs x + 1) / 2) * (
    (%x. psi (natfloor (abs x) + 1) * ln (abs x + 1)) +
    (%x. ∑ n = 1..natfloor (abs x) + 1.
      ∑ u | u dvd n. Lambda u * Lambda (n div u))) =o
    (%x. ln(abs x + 1) / 2) *o
      ((%x. 2 * (abs x + 1) * ln (abs x + 1)) +o O(%x. abs x + 1))
        (is ?LHS =o ?RHS)
    by (rule set-times-intro2)
  also have ?LHS = (%x. psi (natfloor (abs x) + 1) * ln (abs x + 1) ^2 / 2)
    + (%x. ln(abs x + 1) / 2 * (∑ n = 1..natfloor (abs x) + 1.
      ∑ u | u dvd n. Lambda u * Lambda (n div u)))
    apply (simp add: set-times-plus-distribs func-times func-plus)
    apply (rule ext)
    apply (simp add: ring-eq-simps realpow-two2 [THEN sym])
    done
  also have ?RHS = (%x. (abs x + 1) * ln (abs x + 1) ^2) +o
      (%x. ln(abs x + 1) / 2) *o O(%x. abs x + 1)
    apply (simp add: set-times-plus-distribs func-times
      realpow-two2 [THEN sym])
    apply (subgoal-tac (%x. ln (abs x + 1) * ((2 * abs x + 2) *
        ln (abs x + 1)) / 2) =
        (%x. (abs x + 1) * (ln (abs x + 1) * ln (abs x + 1))))
    apply (erule ssubst, rule refl)
    apply (rule ext)
    apply (simp add: ring-eq-simps)
    done
  also have ... <= (%x. (abs x + 1) * ln (abs x + 1) ^2) +o
      (O(%x. 1 + ln(abs x + 1)) * O(%x. abs x + 1))
    apply (rule set-plus-mono)
    apply (rule set-times-mono3)
    apply (rule bigo-bounded)
    apply force
    apply (rule allI)
    apply (rule real-le-mult-imp-div-pos-le)
```

apply *force*
apply *simp*
apply (*subgoal-tac 0 <= ln(abs x + 1)*)
apply *arith*
apply *auto*
done
also have ... <= (%*x*. (*abs x + 1*) * *ln* (*abs x + 1*) ^ *2*) +*o*
O((%*x*. *1* + *ln*(*abs x + 1*)) * (%*x*. *abs x + 1*))
apply (*rule set-plus-mono*)
apply (*rule bigo-mult*)
done
also have (%*x*. *1* + *ln*(*abs x + 1*)) * (%*x*. *abs x + 1*) =
(%*x*. (*abs x + 1*) * (*1* + *ln*(*abs x + 1*)))
apply (*rule ext*)
apply (*subst mult-commute*)
apply (*subst func-times*)
apply (*rule refl*)
done
finally have *a*: (%*x*. *psi* (*natfloor* (*abs x*) + *1*) * *ln* (*abs x + 1*) ^ *2* / *2*) +
(%*x*. *ln* (*abs x + 1*) / *2* *
(∑ *n* = *1*..*natfloor* (*abs x*) + *1*.
∑ *u* | *u dvd n*. *Lambda u* * *Lambda* (*n div u*))) =*o*
(%*x*. (*abs x + 1*) * *ln* (*abs x + 1*) ^ *2*) +*o*
O(%*x*. (*abs x + 1*) * (*1* + *ln* (*abs x + 1*))).
have (%*x*. *psi* (*natfloor* (*abs x*) + *1*) * *ln* (*abs x + 1*) ^ *2* / *2*) =*o*
(%*x*. ∑ *m* = *1*..*natfloor* (*abs x*) + *1*.
∑ *n* = *1*..(*natfloor* (*abs x*) + *1*) *div m*.
∑ *u* | *u dvd n*. *Lambda m* * *Lambda u* * *Lambda* (*n div u*)) +*o*
O(%*x*. (*abs x + 1*) * (*1* + *ln* (*abs x + 1*)))
(**is** *?LHS =o ?RHS1 +o ?RHS2*)
by (*rule aux5* [*OF a*, *OF Selberg7a*])
then have (%*x*. *2*) * *?LHS* =*o* (%*x*. *2*) *o *?RHS1* +*o* *?RHS2*)
by (*rule set-times-intro2*)
also have ... = (%*x*. *2*) * *?RHS1* +*o* (%*x*. *2*) *o *?RHS2*
by (*rule set-times-plus-distrib*)
also have (%*x*. *2*) * *?LHS* =
(%*x*. *psi* (*natfloor* (*abs x*) + *1*) * *ln* (*abs x + 1*) ^ *2*)
by (*simp add*: *func-times*)
also have (%*x*. *2*) * *?RHS1* =
(%*x*. *2* * (∑ *m* = *1*..*natfloor* (*abs x*) + *1*.
∑ *n* = *1*..(*natfloor* (*abs x*) + *1*) *div m*.
∑ *u* | *u dvd n*. *Lambda m* * *Lambda u* * *Lambda* (*n div u*)))
by (*simp add*: *func-times*)
also have (%*x*. *2*) *o *?RHS2* = *?RHS2*
by (*rule bigo-const-mult5*, *force*)

finally show *?thesis*.
qed

lemma *Selberg8a*: $(\%x.\ psi\ (natfloor\ (abs\ x) + 1) * ln\ (abs\ x + 1)\ \hat{}\ 2) =o$
$(\%x.\ 2 * (\sum c = 1..natfloor\ (abs\ x) + 1.$
$\sum v \mid v\ dvd\ c.\ Lambda\ v * Lambda\ (c\ div\ v) *$
$psi((natfloor\ (abs\ x) + 1)\ div\ c))) +o$
$O(\%x.\ (abs\ x + 1) * (1 + ln\ (abs\ x + 1)))$
proof −
note *Selberg8*
also have $(\%x.\ 2 * (\sum\ m = 1..natfloor\ (abs\ x) + 1.$
$\sum\ n = 1..(natfloor\ (abs\ x) + 1)\ div\ m.$
$\sum\ u \mid u\ dvd\ n.\ Lambda\ m * Lambda\ u * Lambda\ (n\ div\ u))) =$
$(\%x.\ 2 * (\sum\ m = 1..natfloor\ (abs\ x) + 1.$
$\sum\ c = 1..(natfloor\ (abs\ x) + 1)\ div\ m.$
$\sum\ u = 1..((natfloor\ (abs\ x) + 1)\ div\ m)\ div\ c.$
$Lambda\ m * Lambda\ u * Lambda\ c))$
apply (*rule ext*)
apply (*rule arg-cong*)**back**
apply (*rule setsum-cong2*)
apply (*rule sym*)
apply (*subst general-inversion-nat2-modified*)
apply (*rule nat-div-gr-0*)
apply *force*
apply *force*
apply (*rule setsum-cong2*)+
apply *simp*
done
also have $... = (\%x.\ 2 * (\sum c = 1..natfloor\ (abs\ x) + 1.$
$\sum v \mid v\ dvd\ c.\ Lambda\ v * Lambda\ (c\ div\ v) *$
$psi((natfloor\ (abs\ x) + 1)\ div\ ((c\ div\ v) * v))))$
apply (*rule ext*)
apply (*rule arg-cong*)**back**
apply (*subst general-inversion-nat2-cor1-modified*)
apply *force*
apply (*subst general-inversion-nat2-modified* [*THEN sym*])
apply *force*
apply (*rule setsum-cong2*)
apply (*rule setsum-cong2*)
apply (*subst psi-def-alt*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong*)
apply (*subst div-mult2-eq*)
apply (*rule refl*)
apply *simp*

```
  done
also have (%x. 2 * (∑ c = 1..natfloor (abs x) + 1.
        ∑ v | v dvd c.
          Lambda v * Lambda (c div v) *
          psi ((natfloor (abs x) + 1) div (c div v * v)))) =
    (%x. 2 * (∑ c = 1..natfloor (abs x) + 1.
        ∑ v | v dvd c.
          Lambda v * Lambda (c div v) *
          psi ((natfloor (abs x) + 1) div c)))
  apply (rule ext)
  apply (rule arg-cong)back
  apply (rule setsum-cong2)+
  apply (subst mult-commute)backbackbackback
  apply (subst nat-dvd-mult-div)
  apply clarsimp
  apply (rule dvd-pos-pos)
  prefer 2
  apply assumption
  apply auto
  done
finally show ?thesis.
qed

end
```

41 Estimates on the error term

```
theory Error = Selberg:

declare One-nat-def [simp del]

constdefs
  R :: real => real
  R x == psi (natfloor x) - x

lemma R-alt-def: psi (natfloor x) = x + R x
  by (simp add: R-def)

lemma R-alt-def2: psi (natfloor x) = R x + x
  by (simp add: R-def)

lemma R-alt-def3: psi n = real n + R (real n)
  by (simp add: R-def)
```

lemma *R-alt-def4*: *psi n = R (real n) + real n*
by (*simp add*: *R-def*)

lemma *R-bound-imp-PNT*: *ALL epsilon. (0 < epsilon −−>*
*(EX z. ALL x. (z <= x −−> abs (R x) < epsilon * x))) ==>*
(%x. psi x / (real x)) −−−−> 1
apply (*unfold LIMSEQ-def*)
apply *clarsimp*
apply (*drule-tac x = r* **in** *spec*)
apply *clarsimp*
apply (*rule-tac x = natfloor z + 1* **in** *exI*)
apply *clarify*
apply (*drule-tac x = real n* **in** *spec*)
apply (*unfold R-def*)
apply (*subgoal-tac z <= real n*)
apply (*clarsimp simp del*: *One-nat-def*)
apply (*subgoal-tac abs (psi n − real n) / real n < r*)
prefer *2*
apply (*subst pos-divide-less-eq*)
apply *arith*
apply *assumption*
apply (*subgoal-tac abs (psi n − real n) / real n =*
abs(psi n / real n + −1))
apply (*erule subst*)
apply *assumption*
apply (*subst abs-div-pos*)
apply *force*
apply (*rule arg-cong*)**back**
apply (*simp add*: *ring-eq-simps diff-divide-distrib*)
apply (*rule order-less-imp-le*)
apply (*erule ge-natfloor-plus-one-imp-gt*)
done

lemma *aux*: *f + g =o h +o O(k::'a=>'b::ordered-ring) ==>*
g =o l +o O(k) ==> f + l =o h +o O(k)
apply (*rule set-minus-imp-plus*)
apply (*drule set-plus-imp-minus*)+
apply (*drule bigo-minus*)**back**
apply (*drule bigo-add-useful*)
apply *assumption*
apply (*subgoal-tac f + g − h + − (g − l) = f + l − h*)
apply (*erule subst*)
apply *assumption*
apply (*simp add*: *ring-eq-simps compare-rls minus-diff-eq*)

done

lemma *error0*: $R =o O(\%x.\ abs\ x)$
apply (*subgoal-tac* $R = (\%x.\ psi\ (natfloor\ x)) + (\%x.\ -\ x)$)
apply (*erule ssubst*)
apply (*subst bigo-plus-idemp* [*THEN sym*])
apply (*rule set-plus-intro*)
apply (*rule subsetD*)
apply (*subgoal-tac* $O(\%x.\ real(natfloor\ x)) <= O(\%x.\ abs\ x)$)
apply *assumption*
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply (*rule allI*)
apply (*case-tac* $0 <= x$)
apply *simp*
apply (*erule real-natfloor-le*)
apply (*rule order-trans*)
prefer *2*
apply (*rule abs-ge-zero*)
apply (*subst natfloor-neg*)
apply *simp*
apply *simp*
apply (*rule bigo-compose1*)
apply (*rule psi-bigo*)
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac* $x = 1$ **in** *exI*)
apply *auto*
apply (*rule ext*)
apply (*subst func-plus*)
apply (*subst diff-minus* [*THEN sym*])
apply (*simp add*: *R-def*)
done

lemma *R-zero* [*simp*]: $R\ 0 = 0$
apply (*unfold R-def*)
apply (*simp add*: *psi-zero*)
done

lemma *error1*: $(\%x.\ \sum\ n = 1..natfloor\ (abs\ x).$
$R\ (real\ n)\ /\ (real\ n * real\ (n + 1))) =o O(\%x.\ 1)$
proof −
have $(\%x.\ \sum n = 1..natfloor\ (abs\ x) + 1.\ Lambda\ n\ /\ (real\ n)) =$
$(\%x. \sum n=1..natfloor(abs\ x) + 1.\ (1\ /\ real\ n) * (psi\ n - psi\ (n - 1)))$

apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst psi-diff2*)
apply *force*
apply *simp*
done
also have ... = (%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*. *1* / *real n*) +
(%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*. *1* / *real n* ∗
(*R*(*real n*) − *R*(*real* (*n* − *1*))))
apply (*subst func-plus*)
apply (*rule ext*)
apply (*subst setsum-addf* [*THEN sym*])
apply (*rule setsum-cong2*)
apply (*subst R-alt-def3*)+
apply (*subst real-of-nat-diff*)
apply *force*
apply (*simp add*: *ring-eq-simps*)
done
also have (%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*.
1 / *real n* ∗ (*R* (*real n*) − *R* (*real* (*n* − *1*))))
= (%*x*. *R* (*real* (*natfloor* (*abs x*) + *1*)) / *real* (*natfloor* (*abs x*) + *1*)) +
(%*x*. ($\sum$ *n* = *1*..*natfloor* (*abs x*).
R (*real n*) ∗ (*1* / *real n* − *1* / *real* (*n* + *1*))))
apply (*subst func-plus*)
apply (*rule ext*)
apply (*subst another-partial-sum*)
apply *simp*
done
also have (%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*).
R (*real n*) ∗ (*1* / *real n* − *1* / *real* (*n* + *1*))) =
(%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*).
R (*real n*) / ((*real n*) ∗ (*real* (*n* + *1*))))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*simp add*: *right-diff-distrib*)
apply (*subst diff-frac-eq*)
apply (*auto simp add*: *ring-eq-simps*)
done
also have
(%*x*. *R* (*real* (*natfloor* (*abs x*) + *1*)) / *real* (*natfloor* (*abs x*) + *1*)) =
(%*x*. *1* / (*real* (*natfloor* (*abs x*) + *1*))) ∗
(%*x*. *R*(*real*(*natfloor*(*abs x*) + *1*)))
by (*simp add*: *func-times*)
finally have (%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*).
R (*real n*) / (*real n* ∗ *real* (*n* + *1*))) =

(%x. $\sum$ n = 1..natfloor (abs x) + 1. Lambda n / real n) +
− (%x. $\sum$ n = 1..natfloor (abs x) + 1. 1 / real n) +
− ((%x. 1 / (real (natfloor (abs x) + 1))) *
(%x. R(real(natfloor(abs x) + 1))))
by (*simp add*: *compare-rls*)
also have ... =*o*
((%x. ln(abs x + 1)) +*o* O(%x. 1)) +
(− 1) *o* ((%x. ln(abs x + 1)) +*o* O(%x. 1)) +
(− 1) *o* (O(%x. 1 / (abs x + 1)) * (O(%x. abs x + 1) + O(%x. abs x + 1)))
apply (*rule set-plus-intro*)
apply (*rule set-plus-intro*)
apply (*rule Mertens-theorem-real2*)
apply (*rule set-neg-intro2*)
apply (*rule bigo-add-commute-imp*)
apply (*rule ln-sum-real2*)
apply (*rule set-neg-intro2*)
apply (*rule set-times-intro*)
apply (*rule one-over-natfloor-one-over-bigo*)
apply (*unfold R-def*)
apply (*subgoal-tac* (%x. *psi* (*natfloor* (*real* (*natfloor* (*abs* x) + 1))) −
real (*natfloor* (*abs* x) + 1)) = (%x. *psi*(*natfloor* (*abs* x) + 1)) +
− (%x. *real*(*natfloor*(*abs* x) + 1)))
apply (*erule ssubst*)
apply (*rule set-plus-intro*)
apply (*rule set-mp*)
prefer *2*
apply (*rule bigo-compose1*)**back**
apply (*rule psi-bigo*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply *clarsimp*
apply (*rule real-natfloor-le*)
apply *force*
apply (*rule bigo-minus*)
apply (*rule bigo-bounded*)
apply *force*
apply *clarsimp*
apply (*rule real-natfloor-le*)
apply *force*
apply (*simp add*: *func-minus func-plus*)
apply (*rule ext*)
apply (*simp add*: *ring-eq-simps compare-rls*)
done

```
  also have ... <= O(%x. 1)
    apply (simp add: set-times-plus-distribs set-plus-rearranges)
    apply (rule subset-trans)
    prefer 2
    apply (subgoal-tac O(%x. 1) + O(%x. 1) + O(%x. 1) <= O(%x. 1))
    apply assumption
    apply simp
    apply (rule set-plus-mono2)+
    apply simp
    apply (simp add: func-one func-minus)
    apply (rule subset-trans)
    apply (subgoal-tac ?t <= (- 1) *o O(%x. 1))
    apply assumption
    apply (rule set-times-mono)
    apply (rule subset-trans)
    apply (rule bigo-mult)
    apply (simp add: func-times)
    apply (rule bigo-elt-subset)
    apply (subgoal-tac (%x. (abs x + 1) / (abs x + 1)) = (%x. 1))
    apply (erule ssubst)
    apply (rule bigo-refl)
    apply (rule ext)
    apply simp
    apply arith
    apply (simp add: func-one func-minus)
    done
  finally show ?thesis.
qed

lemma sum-lambda-n-ln-n-over-n-bigo:
    (%x. ∑ n = 1..natfloor (abs x) + 1.
        Lambda n * ln (real n) / real n) =o
     (%x. ln (abs x + 1) ^ 2 / 2) +o O(%x. 1 + ln (abs x + 1))
proof -
  have (%x. ∑ n = 1..natfloor (abs x) + 1.
        Lambda n * ln (real n) / real n) =
      ((%x. (ln (abs x + 1))) * (%x. (∑ n = 1..natfloor (abs x) + 1.
          Lambda n / real n))) +
      ((%x. - 1) * (%x. ∑ n = 1..natfloor (abs x) + 1.
          Lambda n / real n * ln((abs x + 1) / real n)))
    apply (simp add: func-plus func-times)
    apply (rule ext)
    apply (subst setsum-const-times [THEN sym])
    apply (subst setsum-negf' [THEN sym])
    apply (subst setsum-addf [THEN sym])
```

```
    apply (rule setsum-cong2)
    apply (subst ln-div)
    apply arith
    apply force
    apply (simp add: ring-eq-simps diff-divide-distrib)
    done
  also have ... =o
      ((%x. ln(abs x + 1)) *o ((%x. ln (abs x + 1)) +o O(%x. 1))) +
      ((%x. - 1) *o
        ((%x. ln (abs x + 1) ^ 2 / 2) +o O(%x. 1 + ln (abs x + 1))))
    apply (rule set-plus-intro)
    apply (rule set-times-intro2)
    apply (rule Mertens-theorem-real2)
    apply (rule set-times-intro2)
    apply (rule sum-lambda-m-over-m-ln-x-over-m-bigo)
    done
  also have ... =
      (%x. ln(abs x + 1)^2 / 2) +o ((%x. ln(abs x + 1)) *o O(%x. 1) +
        O(%x. 1 + ln(abs x + 1)))
    by (simp add: set-plus-rearranges plus-ac0 set-times-plus-distribs
      func-times func-plus power2-eq-square ring-eq-simps
      compare-rls)
  also have ... <= (%x. ln(abs x + 1)^2 / 2) +o O(%x. 1 + ln(abs x + 1))
    apply (rule set-plus-mono)
    apply (subst bigo-plus-idemp [THEN sym])back
    apply (rule set-plus-mono2)
    apply (rule order-trans)
    apply (rule bigo-mult2)
    apply (simp add: func-times)
    apply (rule bigo-elt-subset)
    apply (rule bigo-bounded)
    apply auto
    done
  finally show ?thesis.
qed

lemma error2:  (%x. R (abs x + 1) * ln (abs x + 1) ^ 2) +
  (%x. 2 * (sum n = 1..natfloor (abs x) + 1.
    Lambda n * ln (real n) * R ((abs x + 1) / real n))) =o
  O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
proof -
  have (%x. psi (natfloor (abs x) + 1) * ln (abs x + 1)) +
    (%x. sum n = 1..natfloor (abs x) + 1.
      sum u | u dvd n. Lambda u * Lambda (n div u))
    =o (%x. 2 * (abs x + 1) * ln (abs x + 1)) +o O(%x. abs x + 1)
```

(**is** *?LHS1 + ?LHS2 =o ?RHS1 +o ?RHS2*)
by (*rule Selberg4*)
then have (%*x*. *ln* (*abs x + 1*)) * (*?LHS1 + ?LHS2*) =*o*
(%*x*. *ln*(*abs x + 1*)) *o (*?RHS1 +o ?RHS2*)
by (*rule set-times-intro2*)
then have (%*x*. *ln* (*abs x + 1*)) * *?LHS1 +*
(%*x*. *ln*(*abs x + 1*)) * *?LHS2 =o*
(%*x*. *ln* (*abs x + 1*)) * *?RHS1 +o*
(%*x*. *ln*(*abs x + 1*)) *o ?RHS2*
by (*simp add*: *ring-distrib set-times-plus-distribs*)
also have ... <= (%*x*. *ln* (*abs x + 1*)) * *?RHS1 +o*
O(%*x*. *ln*(*abs x + 1*) * (*abs x + 1*))
apply (*rule set-plus-mono*)
apply (*rule subset-trans*)
apply (*rule bigo-mult2*)
apply (*simp add*: *func-times*)
done
also have ... <= (%*x*. *ln* (*abs x + 1*)) * *?RHS1 +o*
O(%*x*. (*abs x + 1*) * (*1 + ln* (*abs x + 1*)))
apply (*rule set-plus-mono*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply (*rule allI*)
apply (*rule nonneg-times-nonneg*)
apply *force*
apply *arith*
apply (*rule allI*)
apply (*simp add*: *ring-eq-simps compare-rls*)
apply *arith*
done
also have (%*x*. *ln* (*abs x + 1*)) * *?LHS1* = (%*x*. *psi*(*natfloor*(*abs x*) + *1*) *
(*ln* (*abs x + 1*))^*2*)
by (*simp add*: *func-times mult-ac power2-eq-square*)
also have (%*x*. *ln* (*abs x + 1*)) * *?LHS2* = (%*x*. *ln*(*abs x + 1*) *
(∑ *n = 1..natfloor* (*abs x*) + *1*.
∑ *u* | *u dvd n*. *Lambda u * Lambda* (*n div u*)))
by (*simp add*: *func-times*)
also have (%*x*. *ln* (*abs x + 1*)) * *?RHS1* = (%*x*. *2* * (*abs x + 1*) *
ln(*abs x + 1*)^*2*)
by (*simp add*: *func-times mult-ac power2-eq-square*)
finally have *a*: (%*x*. *psi* (*natfloor* (*abs x*) + *1*) * *ln* (*abs x + 1*) ^ *2*) +
(%*x*. *ln* (*abs x + 1*) *
(∑ *n = 1..natfloor* (*abs x*) + *1*.
∑ *u* | *u dvd n*. *Lambda u * Lambda* (*n div u*))) =*o*
(%*x*. *2* * (*abs x + 1*) * *ln* (*abs x + 1*) ^ *2*) +*o*

```
    O(%x. (abs x + 1) * (1 + ln (abs x + 1))).
note Selberg6a
then have (%x. 2) * (%x. ln (abs x + 1) / 2 *
      (∑ n = 1..natfloor (abs x) + 1.
       ∑ u | u dvd n. Lambda u * Lambda (n div u))) =o
    (%x. 2) *o ((%x. ∑ m = 1..natfloor (abs x) + 1.
      ∑ n = 1..(natfloor (abs x) + 1) div m.
      Lambda m * Lambda n * ln (real n)) +o
    O(%x. (abs x + 1) * (1 + ln (abs x + 1))))
     (is ?LHS3 =o ?RHS3)
  apply (intro set-times-intro2)
  apply (erule bigo-add-commute-imp)
  done
also have ?LHS3 = (%x. ln (abs x + 1) *
      (∑ n = 1..natfloor (abs x) + 1.
       ∑ u | u dvd n. Lambda u * Lambda (n div u)))
  by (simp add: func-times)
also have ?RHS3 =  (%x. 2 * (∑ m = 1..natfloor (abs x) + 1.
      ∑ n = 1..(natfloor (abs x) + 1) div m.
      Lambda m * Lambda n * ln (real n))) +o
    (%x. 2) *o O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
     (is ?RHS3 = ?RHS4 +o ?RHS5)
  by (simp add: set-times-plus-distribs func-times)
also have ... <= ?RHS4 +o O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
  apply (rule set-plus-mono)
  apply (rule bigo-const-mult6)
  done
finally have b: (%x. ln (abs x + 1) *
      (∑ n = 1..natfloor (abs x) + 1.
      ∑ u | u dvd n. Lambda u * Lambda (n div u))) =o
    (%x. 2 *
      (∑ m = 1..natfloor (abs x) + 1.
      ∑ n = 1..(natfloor (abs x) + 1) div m.
      Lambda m * Lambda n * ln (real n))) +o
    O(%x. (abs x + 1) * (1 + ln (abs x + 1))).
have (%x. psi (natfloor (abs x) + 1) * ln (abs x + 1) ^ 2) +
    (%x. 2 *
      (∑ m = 1..natfloor (abs x) + 1.
      ∑ n = 1..(natfloor (abs x) + 1) div m.
      Lambda m * Lambda n * ln (real n))) =o
    (%x. 2 * (abs x + 1) * ln (abs x + 1) ^ 2) +o
    O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
  by (rule aux [OF a, OF b])
also have (%x. 2 * (∑ m = 1..natfloor (abs x) + 1.
      ∑ n = 1..(natfloor (abs x) + 1) div m.
```

Lambda m ∗ Lambda n ∗ ln (real n))) =
(%x. 2 ∗ (∑ n = 1..natfloor (abs x) + 1.
Lambda n ∗ ln (real n) ∗ psi((natfloor (abs x) + 1) div n)))
apply (*rule ext*)
apply (*rule arg-cong*)**back**
apply (*subst general-inversion-nat2-cor1-modified*)
apply *force*
apply (*rule setsum-cong2*)
apply (*subst psi-def-alt*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
done
also have ... =
(%x. 2 ∗ (∑ n = 1..natfloor (abs x) + 1.
Lambda n ∗ ln (real n) ∗ psi(natfloor ((abs x + 1) / real n))))
apply (*rule ext*)
apply (*rule arg-cong*)**back**
apply (*rule setsum-cong2*)
apply (*subst natfloor-div-nat*)
apply *simp*
apply *force*
apply (*subst natfloor-add* [*THEN sym*])
apply *simp*
apply *simp*
done
also have ... = (*%x. 2 ∗ (∑ n = 1..natfloor (abs x) + 1.*
Lambda n ∗ ln (real n) ∗ R((abs x + 1) / real n))) +
(%x. 2 ∗ (∑ n = 1..natfloor (abs x) + 1.
Lambda n ∗ ln (real n) ∗ ((abs x + 1) / real n)))
apply (*subst func-plus*)
apply (*rule ext*)
apply (*subst right-distrib* [*THEN sym*])
apply (*rule arg-cong*)**back**
apply (*subst setsum-addf* [*THEN sym*])
apply (*rule setsum-cong2*)
apply (*subst R-alt-def*)
apply (*simp add*: *ring-eq-simps*)
done
also have (*%x. 2 ∗ (∑ n = 1..natfloor (abs x) + 1.*
Lambda n ∗ ln (real n) ∗ ((abs x + 1) / real n))) =
(%x. 2 ∗ (abs x + 1)) ∗
(%x. ∑ n = 1..natfloor (abs x) + 1.
Lambda n ∗ ln (real n) / real n)
apply (*subst func-times*)

apply (*rule ext*)
apply (*subst mult-assoc*)
apply (*rule arg-cong*)**back**
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply (*simp add*: *ring-eq-simps*)
done
also have (%*x*. *psi* (*natfloor* (*abs x*) + *1*) ∗ *ln* (*abs x* + *1*) ^ *2*) =
(%*x*. *R*(*abs x* + *1*) ∗ *ln* (*abs x* + *1*) ^ *2*) +
(%*x*. (*abs x* + *1*) ∗ *ln* (*abs x* + *1*) ^ *2*)
apply (*subst func-plus*)
apply (*rule ext*)
apply (*subst natfloor-add* [*THEN sym*])
apply *simp*
apply *simp*
apply (*subst R-alt-def*)
apply (*simp add*: *ring-eq-simps*)
done
finally have (%*x*. *R* (*abs x* + *1*) ∗ *ln* (*abs x* + *1*) ^ *2*) +
(%*x*. (*abs x* + *1*) ∗ *ln* (*abs x* + *1*) ^ *2*) +
((%*x*. *2* ∗
(∑ *n* = *1*..*natfloor* (*abs x*) + *1*.
Lambda n ∗ *ln* (*real n*) ∗ *R* ((*abs x* + *1*) / *real n*))) +
(%*x*. *2* ∗ (*abs x* + *1*)) ∗
(%*x*. ∑ *n* = *1*..*natfloor* (*abs x*) + *1*.
Lambda n ∗ *ln* (*real n*) / *real n*)) =*o*
(%*x*. *2* ∗ (*abs x* + *1*) ∗ *ln* (*abs x* + *1*) ^ *2*) +*o*
O(%*x*. (*abs x* + *1*) ∗ (*1* + *ln* (*abs x* + *1*)))
(**is** *?LHS* =*o* *?RHS*)**.**
then have − (%*x*. (*abs x* + *1*) ∗ *ln* (*abs x* + *1*) ^ *2*) +
− (%*x*. *2* ∗ (*abs x* + *1*)) ∗
(%*x*. ∑ *n* = *1*..*natfloor* (*abs x*) + *1*.
Lambda n ∗ *ln* (*real n*) / *real n*) + *?LHS* =*o*
(− (%*x*. (*abs x* + *1*) ∗ *ln* (*abs x* + *1*) ^ *2*) +
(− (%*x*. *2* ∗ (*abs x* + *1*)) ∗
(%*x*. ∑ *n* = *1*..*natfloor* (*abs x*) + *1*.
Lambda n ∗ *ln* (*real n*) / *real n*))) +*o* *?RHS*
(**is** *?LHS2* =*o* *?RHS2*)
by (*rule set-plus-intro2*)
also have *?LHS2* = (%*x*. *R* (*abs x* + *1*) ∗ *ln* (*abs x* + *1*) ^ *2*) +
(%*x*. *2* ∗ (∑ *n* = *1*..*natfloor* (*abs x*) + *1*.
Lambda n ∗ *ln* (*real n*) ∗ *R* ((*abs x* + *1*) / *real n*)))
by (*simp add*: *func-plus func-minus func-times ring-eq-simps*
compare-rls)
also have *?RHS2* = (%*x*. (*abs x* + *1*) ∗ *ln* (*abs x* + *1*) ^ *2*) +*o*

```
    ((− (%x. 2 * (abs x + 1)) *
        (%x. ∑ n = 1..natfloor (abs x) + 1.
        Lambda n * ln (real n) / real n)) +o
      O(%x. (abs x + 1) * (1 + ln (abs x + 1))))
  by (simp add: set-plus-rearranges func-minus func-plus
    func-times ring-eq-simps compare-rls)
finally have (%x. R (abs x + 1) * ln (abs x + 1) ^ 2) +
    (%x. 2 * (∑ n = 1..natfloor (abs x) + 1.
      Lambda n * ln (real n) * R ((abs x + 1) / real n))) =o
    (%x. (abs x + 1) * ln (abs x + 1) ^ 2) +o
      ((− (%x. 2 * (abs x + 1)) *
        (%x. ∑ n = 1..natfloor (abs x) + 1.
          Lambda n * ln (real n) / real n)) +o
      O(%x. (abs x + 1) * (1 + ln (abs x + 1)))).
also have ... <= (%x. (abs x + 1) * ln (abs x + 1) ^ 2) +o
    (((− (%x. 2 * (abs x + 1))) *o
      ((%x. ln(abs x + 1)^2 / 2) +o O(%x. 1 + ln (abs x + 1)))) +
    O(%x. (abs x + 1) * (1 + ln (abs x + 1))))
  apply (rule set-plus-mono)
  apply (rule set-plus-mono5)
  apply (rule set-times-intro2)
  apply (rule sum-lambda-n-ln-n-over-n-bigo)
  apply simp
  done
also have ... = O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
  apply (simp only: set-plus-rearranges set-times-plus-distribs
    func-minus func-plus func-times)
  apply (subgoal-tac (%x. (abs x + 1) * ln (abs x + 1) ^ 2 +
      − (2 * (abs x + 1)) * (ln (abs x + 1) ^ 2 / 2)) = 0)
  apply (erule ssubst)
  apply simp
  apply (subgoal-tac (%x. − (2 * abs x) + −2) *o O(%x. 1 + ln (abs x + 1))
    = O(%x. (abs x + 1) * (1 + ln (abs x + 1))))
  apply (erule ssubst)
  apply simp
  apply (subgoal-tac (%x. − (2 * abs x) + −2) *o O(%x. 1 + ln (abs x + 1))
    = (%x. (abs x + 1)) *o ((%x. −2) *o
      O(%x. (1 + ln (abs x + 1)))))
  apply (erule ssubst)
  apply (subst bigo-const-mult5)
  apply simp
  apply (subst bigo-mult6 [THEN sym])
  apply (rule allI)
  apply arith
  apply (simp add: func-times)
```

```
    apply (simp only: set-times-rearranges func-times ring-eq-simps)
    apply (subgoal-tac (%x. -2 + - (abs x * 2)) = (%x. 1 * -2 + abs x * -2))
    apply (erule ssubst)
    apply (rule refl)
    apply (rule ext)
    apply simp
    apply (simp add: func-zero)
    apply (rule ext)
    apply (simp add: ring-eq-simps add-divide-distrib)
    done
  finally show ?thesis.
qed

lemma error3: (%x. abs(R (abs x + 1)) * ln (abs x + 1) ^ 2) <o
  (%x. 2 * (∑ n = 1..natfloor (abs x) + 1.
    Lambda n * ln (real n) * abs(R ((abs x + 1) / real n)))) =o
  O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
proof -
  note error2
  then have (%x. R (abs x + 1) * ln (abs x + 1) ^ 2) =o
      - (%x. 2 * (∑ n = 1..natfloor (abs x) + 1.
        Lambda n * ln (real n) * R ((abs x + 1) / real n))) +o
      O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
      (is ?LHS =o ?RHS1 +o ?RHS2)
    by (intro set-minus-imp-plus, simp)
  then have (%x. abs(?LHS x)) =o (%x. abs(?RHS1 x)) +o ?RHS2
    by (rule bigo-abs4)
  also have (%x. abs(?LHS x)) =
      (%x. abs(R (abs x + 1)) * ln (abs x + 1) ^ 2)
    apply (rule ext)
    apply (subst abs-times-pos)
    apply force
    apply (rule refl)
    done
  also have (%x. abs(?RHS1 x)) = (%x. 2 *
      abs (∑ n = 1..natfloor (abs x) + 1.
        Lambda n * ln (real n) * R ((abs x + 1) / real n)))
    apply (rule ext)
    apply (simp add: func-minus)
    done
  finally have a: (%x. abs (R (abs x + 1)) * ln (abs x + 1) ^ 2) =o
      (%x. 2 *
        abs (∑ n = 1..natfloor (abs x) + 1.
            Lambda n * ln (real n) * R ((abs x + 1) / real n))) +o
      O(%x. (abs x + 1) * (1 + ln (abs x + 1))).
```

```
  show (%x. abs (R (abs x + 1)) * ln (abs x + 1) ^ 2) <o
     (%x. 2 *
       (∑ n = 1..natfloor (abs x) + 1.
           Lambda n * ln (real n) * abs(R ((abs x + 1) / real n)))) =o
     O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
    apply (rule bigo-lesso3 [OF a])
    apply (rule allI)
    apply (rule nonneg-times-nonneg)
    apply force
    apply (rule setsum-nonneg)
    apply force
    apply clarsimp
    apply (rule nonneg-times-nonneg)
    apply (rule nonneg-times-nonneg)
    apply (rule Lambda-ge-zero)
    apply force
    apply force
    apply (rule allI)
    apply (rule mult-left-mono)
    apply (rule order-trans)
    apply (rule abs-setsum)
    apply (rule setsum-le-cong)
    apply simp
    apply (rule mult-right-mono)
    apply (subst abs-nonneg)
    apply force
    apply (subst abs-nonneg)
    apply (rule Lambda-ge-zero)
    apply simp
    apply auto
    done
qed

lemma error4-aux: (%x. ∑ n = 1..natfloor (abs x) + 1.
    Lambda n / real n *
      (∑ m = 1..(natfloor (abs x) + 1) div n. Lambda m / real m)) =o
    (%x. ln (abs x + 1) ^ 2 / 2) +o O(%x. 1 + ln (abs x + 1))
proof −
thm bigo-setsum6 [OF Mertens-theorem-real2]
  have (%x. ∑ n = 1..natfloor (abs x) + 1. Lambda n / real n *
       (∑ m = 1..(natfloor (abs x) + 1) div n. Lambda m / real m)) =
    (%x. ∑ n = 1..natfloor (abs x) + 1. Lambda n / real n *
       (∑ m=1..natfloor(abs ((abs x + 1) / (real n) − 1)) + 1.
         Lambda m / real m))
    apply (rule ext)
```

apply (*rule setsum-cong2*)
apply (*rule arg-cong*)**back**
apply (*subst Selberg.aux*)
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real* (*natfloor* (*abs x*)) <= *abs x*)
apply *force*
apply (*rule real-natfloor-le*)
apply *force*
apply (*subst natfloor-div-nat*)
apply *force*
apply *force*
apply (*subst natfloor-add* [*THEN sym*])
apply *force*
apply *simp*
done
also have ... =*o*
(%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*. *Lambda n* / *real n* *
ln (*abs* ((*abs x* + *1*) / *real n* − *1*) + *1*)) +*o*
O(%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*. *Lambda n* / *real n* * *1*)
apply (*rule bigo-setsum6* [*OF Mertens-theorem-real2*])
apply (*rule allI*)+
apply (*case-tac n* = *0*)
apply *simp*
apply (*rule real-ge-zero-div-gt-zero*)
apply (*rule Lambda-ge-zero*)
apply *force*
apply *force*
done
also have (%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*. *Lambda n* / *real n* *
ln (*abs* ((*abs x* + *1*) / *real n* − *1*) + *1*)) =
(%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*. *Lambda n* / *real n* *
ln ((*abs x* + *1*) / *real n*))
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst Selberg.aux2*)
apply (*rule div-ge-1*)
apply *force*
apply (*subgoal-tac real*(*natfloor*(*abs x*)) <= *abs x*)
apply *force*
apply (*rule real-natfloor-le*)
apply *simp*
apply (*rule refl*)
done
also have (%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*.

```
      Lambda n / real n * ln ((abs x + 1) / real n)) +o
    O(%x. ∑ n = 1..natfloor (abs x) + 1. Lambda n / real n * 1) <=
  (%x. ln (abs x + 1) ^ 2 / 2) +o
    O(%x. 1 + ln (abs x + 1)) + O(%x. 1 + ln (abs x + 1))
  apply (rule set-plus-mono5)
  apply (rule sum-lambda-m-over-m-ln-x-over-m-bigo)
  apply (rule bigo-elt-subset)
  apply (rule subsetD)
  prefer 2
  apply simp
  apply (rule Mertens-theorem-real2)
  apply (rule bigo-plus-absorb2)
  apply (rule bigo-bounded)
  apply force
  apply force
  apply (rule bigo-elt-subset)
  apply (rule bigo-bounded)
  apply force
  apply force
  done
 also have ... = (%x. ln (abs x + 1)^2 / 2) +o O(%x. 1 + ln(abs x + 1))
  by (simp add: set-plus-rearranges)
 finally show ?thesis.
qed

lemma error4: (%x. R (abs x + 1) * ln (abs x + 1) ^ 2) =o
    (%x. 2 * (∑ c = 1..natfloor (abs x) + 1. ∑ v | v dvd c.
      Lambda v * Lambda (c div v) * R ((abs x + 1) / real c))) +o
    O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
proof −
 note Selberg8a
 also have (%x. psi (natfloor (abs x) + 1) * ln (abs x + 1) ^ 2) =
  (%x. R(abs x + 1) * ln(abs x + 1)^2) +
    (%x. (abs x + 1) * ln (abs x + 1)^2)
  apply (subst func-plus)
  apply (rule ext)
  apply (subst R-def)
  apply (simp add: ring-eq-simps)
  apply (subst add-commute)backback
  apply (subst natfloor-add [THEN sym])
  apply simp
  apply (simp add: add-ac)
  done
 also have (%x. 2 * (∑ c = 1..natfloor (abs x) + 1.
    ∑ v | v dvd c. Lambda v * Lambda (c div v) *
```

psi ((*natfloor* (*abs x*) + *1*) *div c*))) =
(%*x*. *2* * (∑ *c* = *1*..*natfloor* (*abs x*) + *1*.
∑ *v* | *v dvd c*. *Lambda v* * *Lambda* (*c div v*) *
R ((*abs x* + *1*) / *real c*))) +
(%*x*. *2* * (∑ *c* = *1*..*natfloor* (*abs x*) + *1*.
∑ *v* | *v dvd c*. *Lambda v* * *Lambda* (*c div v*) *
(*abs x* + *1*) / *real c*))
apply (*simp add*: *func-plus*)
apply (*rule ext*)
apply (*subst right-distrib* [*THEN sym*])
apply (*rule arg-cong*)**back**
apply (*subst setsum-addf* [*THEN sym*])
apply (*rule setsum-cong2*)
apply (*subst setsum-addf* [*THEN sym*])
apply (*rule setsum-cong2*)
apply (*subst R-def*)
apply (*subst natfloor-div-nat*)
apply *force*
apply *force*
apply (*simp add*: *ring-eq-simps*)
apply (*subst add-commute*)**back**
apply (*subst natfloor-add* [*THEN sym*])
apply *force*
apply (*simp add*: *add-commute*)
done
also have (%*x*. *2* * (∑ *c* = *1*..*natfloor* (*abs x*) + *1*.
∑ *v* | *v dvd c*. *Lambda v* * *Lambda* (*c div v*) *
(*abs x* + *1*) / *real c*)) =
(%*x*. *2*) * (%*x*. ∑ *c* = *1*..*natfloor* (*abs x*) + *1*.
∑ *v* | *v dvd c*.
Lambda v * *Lambda* (*c div v*) * (*abs x* + *1*) / *real* ((*c div v*) * *v*))
apply (*subst func-times*)
apply (*rule ext*)
apply (*rule arg-cong*)**back**
apply (*rule setsum-cong2*)+
apply (*subst mult-commute*)
apply (*subst nat-dvd-mult-div*)
apply *clarsimp*
apply (*rule dvd-pos-pos*)
prefer *2*
apply *assumption*
apply *force*
apply *force*
apply (*rule refl*)
done

also have $(\%x.\ \sum\ c = 1..natfloor\ (abs\ x) + 1.\ \sum\ v \mid v\ dvd\ c.$
$\quad Lambda\ v * Lambda\ (c\ div\ v) * (abs\ x + 1)\ /\ real\ (c\ div\ v * v)) =$
$(\%x.\ abs\ x + 1) * (\%x.\ \sum\ n = 1..natfloor\ (abs\ x) + 1.$
$\quad\quad Lambda\ n\ /\ (real\ n) * (\sum\ m = 1..(natfloor\ (abs\ x) + 1)\ div\ n.$
$\quad\quad\quad Lambda\ m\ /\ (real\ m)))$
apply $(subst\ func\text{-}times)$
apply $(rule\ ext)$
apply $(subst\ setsum\text{-}const\text{-}times\ [THEN\ sym])$
apply $(subst\ general\text{-}inversion\text{-}nat2\text{-}modified\ [THEN\ sym])$
apply $force$
apply $(rule\ setsum\text{-}cong2)$
apply $(subst\ setsum\text{-}const\text{-}times\ [THEN\ sym])+$
apply $(rule\ setsum\text{-}cong2)$
apply $(simp\ add{:}\ mult\text{-}ac)$
done
also have $((\%x.\ 2 * (\sum\ c = 1..natfloor\ (abs\ x) + 1.$
$\quad \sum\ v \mid v\ dvd\ c.$
$\quad\quad Lambda\ v * Lambda\ (c\ div\ v) * R\ ((abs\ x + 1)\ /\ real\ c))) +$
$\quad (\%x.\ 2) * ((\%x.\ abs\ x + 1) * (\%x.\ \sum\ n = 1..natfloor\ (abs\ x) + 1.$
$\quad\quad Lambda\ n\ /\ real\ n *$
$\quad\quad\quad (\sum\ m = 1..(natfloor\ (abs\ x) + 1)\ div\ n.$
$\quad\quad\quad\quad Lambda\ m\ /\ real\ m)))) +o$
$\quad O(\%x.\ (abs\ x + 1) * (1 + ln\ (abs\ x + 1))) <=$
$((\%x.\ 2 * (\sum\ c = 1..natfloor\ (abs\ x) + 1.$
$\quad \sum\ v \mid v\ dvd\ c.$
$\quad\quad Lambda\ v * Lambda\ (c\ div\ v) * R\ ((abs\ x + 1)\ /\ real\ c))) +o$
$\quad (\%x.\ 2) *o ((\%x.\ abs\ x + 1) *o$
$\quad\quad ((\%x.\ ln\ (abs\ x + 1)\ \hat{}\ 2\ /\ 2) +o\ O(\%x.\ 1 + ln\ (abs\ x + 1)))\ )) +$
$\quad O(\%x.\ (abs\ x + 1) * (1 + ln\ (abs\ x + 1)))$
apply $(rule\ set\text{-}plus\text{-}mono3)$
apply $(rule\ set\text{-}plus\text{-}intro2)$
apply $(rule\ set\text{-}times\text{-}intro2)$
apply $(rule\ set\text{-}times\text{-}intro2)$
apply $(rule\ error4\text{-}aux)$
done
also have $...\ <=\ ((\%x.\ 2 * (\sum\ c = 1..natfloor\ (abs\ x) + 1.$
$\quad \sum\ v \mid v\ dvd\ c.$
$\quad\quad Lambda\ v * Lambda\ (c\ div\ v) * R\ ((abs\ x + 1)\ /\ real\ c))) +$
$\quad (\%x.\ (abs\ x + 1) * ln\ (abs\ x + 1)\ \hat{}\ 2)) +o$
$\quad O(\%x.\ (abs\ x + 1) * (1 + ln\ (abs\ x + 1)))$
apply $(simp\ add{:}\ set\text{-}times\text{-}plus\text{-}distribs\ set\text{-}plus\text{-}rearranges$
$\quad plus\text{-}ac0\ func\text{-}times\ func\text{-}plus)$
apply $(rule\ set\text{-}plus\text{-}mono)$
apply $(rule\ bigo\text{-}plus\text{-}subset4)$
apply $simp$

```
    apply (rule subset-trans)
    apply (rule set-times-mono)
    apply (rule bigo-mult2)
    apply (simp add: func-times)
    done
  finally have (%x. R (abs x + 1) * ln (abs x + 1) ^ 2) +
      (%x. (abs x + 1) * ln (abs x + 1) ^ 2) =o
    ((%x. 2 * (∑ c = 1..natfloor (abs x) + 1.
          ∑ v | v dvd c.
            Lambda v * Lambda (c div v) * R ((abs x + 1) / real c))) +
     (%x. (abs x + 1) * ln (abs x + 1) ^ 2)) +o
    O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
      (is ?LHS =o ?RHS) .
  then have − (%x. (abs x + 1) * ln (abs x + 1) ^ 2) + ?LHS =o
      − (%x. (abs x + 1) * ln (abs x + 1) ^ 2) +o ?RHS
    by (rule set-plus-intro2)
  thus ?thesis
    by (simp add: set-plus-rearranges func-plus func-minus plus-ac0)
qed

lemma error5: (%x. abs(R (abs x + 1)) * ln (abs x + 1) ^ 2) <o
  (%x. 2 * (∑ c = 1..natfloor (abs x) + 1.
     ∑ v | v dvd c.
       Lambda v * Lambda (c div v) * abs(R ((abs x + 1) / real c)))) =o
  O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
proof −
  have (%x. R (abs x + 1) * ln (abs x + 1) ^ 2) =o
      (%x. 2 * (∑ c = 1..natfloor (abs x) + 1.
        ∑ v | v dvd c.
          Lambda v * Lambda (c div v) * R ((abs x + 1) / real c))) +o
    O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
      (is ?LHS =o ?RHS1 +o ?RHS2)
    by (rule error4)
  then have (%x. abs(?LHS x)) =o (%x. abs(?RHS1 x)) +o ?RHS2
    by (rule bigo-abs4)
  also have (%x. abs(?LHS x)) =
      (%x. abs(R (abs x + 1)) * ln (abs x + 1) ^ 2)
    apply (rule ext)
    apply (subst abs-times-pos)
    apply force
    apply (rule refl)
    done
  also have (%x. abs(?RHS1 x)) = (%x. 2 *
      abs (∑ c = 1..natfloor (abs x) + 1.
        ∑ v | v dvd c.
```

```
      Lambda v * Lambda (c div v) * R ((abs x + 1) / real c)))
  apply (rule ext)
  apply (simp add: func-minus)
  done
 finally have a:  (%x. abs (R (abs x + 1)) * ln (abs x + 1) ^ 2) =o
    (%x. 2 * abs (∑ c = 1..natfloor (abs x) + 1.
      ∑ v | v dvd c.
        Lambda v * Lambda (c div v) * R ((abs x + 1) / real c))) +o
    O(%x. (abs x + 1) * (1 + ln (abs x + 1))).
 show ?thesis
  apply (rule bigo-lesso3 [OF a])
  apply (rule allI)
  apply (rule nonneg-times-nonneg)
  apply force
  apply (rule setsum-nonneg)
  apply force
  apply clarsimp
  apply (rule setsum-nonneg)
  apply (rule finite-nat-dvd-set)
  apply force
  apply clarsimp
  apply (rule nonneg-times-nonneg)
  apply (rule nonneg-times-nonneg)
  apply (rule Lambda-ge-zero)
  apply (rule Lambda-ge-zero)
  apply force
  apply (rule allI)
  apply (rule mult-left-mono)
  apply (rule order-trans)
  apply (rule abs-setsum)
  apply (rule setsum-le-cong)
  apply simp
  apply (rule order-trans)
  apply (rule abs-setsum)
  apply (rule setsum-le-cong)
  apply simp
  apply (subst abs-nonneg)backbackbackbackbackback
  apply (rule Lambda-ge-zero)
  apply (subst abs-nonneg)backbackbackbackbackback
  apply (rule Lambda-ge-zero)
  apply (rule mult-right-mono)
  apply auto
  done
qed
```

lemma *max-zero-add*: *max (a + b) 0 <= max (a::real) 0 + max b 0*
by (*simp split*: *split-max*)

lemma *max-zero-times*: *(0::real) <= c ==> max (c * a) 0 =*
*c * (max a 0)*
by (*auto simp add*: *max-def mult-le-0-iff*)

lemma *aux2*: *(f + f) <o ((%x. 2 * g x) + (%x. 2 * h x)) =o O(k::'a=>real)*
==> f <o (g + h) =o O(k)
apply (*simp add*: *func-plus*)
apply (*unfold lesso-def*)
apply (*subgoal-tac (%x. max (2 * f x − (2 * g x + 2 * h x)) 0) =*
*(%x. 2) * (%x. max (f x − (g x + h x)) 0)*)
apply *simp*
apply (*rule bigo-useful-const-mult*)
apply (*auto simp add*: *func-times*)
apply (*rule ext*)
apply (*subgoal-tac 2 * f x − (2 * g x + 2 * h x) =*
*2 * (f x − (g x + h x))*)
apply (*erule ssubst*)
apply (*rule max-zero-times*)
apply *force*
apply *simp*
done

lemma *error-cor1*: *(%x. abs (R (abs x + 1)) * ln (abs x + 1) ^ 2) <o*
((%x. ∑ n = 1..natfloor (abs x) + 1.
*Lambda n * ln (real n) * abs (R ((abs x + 1) / real n))) +*
(%x. ∑ c = 1..natfloor (abs x) + 1.
∑ v | v dvd c.
*Lambda v * Lambda (c div v) * abs (R ((abs x + 1) / real c)))) =o*
*O(%x. (abs x + 1) * (1 + ln (abs x + 1)))*
by (*rule aux2* [*OF lesso-add* [*OF error3*, *OF error5*]])

constdefs
rho :: nat => real
rho x == ∑ n = 1..x.
*Lambda n * ln (real n) +*
*(∑ u | u dvd n. Lambda u * Lambda (n div u))*

lemma *aux2*: *1 <= (n::nat) ==> (∑ i=1..n. f i) −*
(∑ i=1..(n − 1). f i) = ((f n)::'a::ordered-ring)
apply (*subgoal-tac {1..n} = {1..(n − 1) + 1}*)

```
  apply (erule ssubst)
  apply (subst setsum-range-plus-one-nat')
  apply auto
done

lemma rho-zero: rho 0 = 0
  by (simp add: rho-def)

lemma aux3: 1 <= n ==> rho n - rho (n - 1) =
    Lambda n * ln (real n) +
      (∑ u | u dvd n. Lambda u * Lambda (n div u))
  apply (unfold rho-def)
  apply (erule aux2)
done

lemma rho-bigo: rho =o (%x. 2 * real x * ln (real x)) +o O(%x. real x)
proof -
  have (%x. rho(x + 1)) = (%x. (∑ n = 1..natfloor (abs (real x)) + 1.
        Lambda n * ln (real n) +
        (∑ u | u dvd n. Lambda u * Lambda (n div u))))
    apply (rule ext)
    apply (unfold rho-def)
    apply (subgoal-tac x + 1 = natfloor(abs (real x)) + 1)
    apply (erule ssubst)
    apply (rule refl)
    apply simp
    done
  also have ... =o (%x. 2 * (abs (real x) + 1) * ln (abs (real x) + 1)) +o
      O(%x. abs (real x) + 1)
    by (rule bigo-compose2 [OF Selberg2])
  also have (%x::nat. 2 * (abs (real x) + 1) * ln (abs (real x) + 1)) =
      (%x. 2 * (real (x + 1)) * ln (real (x + 1)))
    apply (rule ext)
    apply (subst abs-nonneg)
    apply force
    apply simp
    done
  also have (%x::nat. abs (real x) + 1) = (%x. real(x + 1))
    apply (rule ext)
    apply (subst abs-nonneg)
    apply simp-all
    done
  finally have (%x. rho (x + 1)) =o
      (%x. 2 * real (x + 1) * ln (real (x + 1))) +o O(%x. real (x + 1)).
  thus ?thesis
```

```
    apply (rule bigo-fix3)
    apply (simp add: rho-zero)
    done
qed

lemma R-bigo: (%x. R(abs x)) =o O(%x. abs x)
  apply (subgoal-tac (%x. R(abs x)) = (%x. psi(natfloor(abs x))) +
      - (%x. abs x))
  apply (erule ssubst)
  apply (rule subsetD)
  apply (subgoal-tac O(%x. real(natfloor (abs x))) + O(%x. abs x) <=
      O(%x. abs x))
  apply assumption
  apply (rule bigo-plus-subset4)
  apply (rule bigo-elt-subset)
  apply (rule bigo-bounded)
  apply force
  apply (rule allI)
  apply (rule real-natfloor-le)
  apply force
  apply simp
  apply (rule set-plus-intro)
  apply (rule bigo-compose1 [OF psi-bigo])
  apply (rule bigo-minus)
  apply (rule bigo-refl)
  apply (rule ext)
  apply (unfold R-def)
  apply (simp add: func-plus func-minus)
done

lemma error6-aux: (%x. abs (R ((abs x + 1) / (real (natfloor (abs x)) + 1))))
    =o O(%x. 1)
  apply (rule subsetD)
  prefer 2
  apply (rule bigo-abs)
  apply (rule bigo-elt-subset)
  apply (subgoal-tac
    (%x. R ((abs x + 1) / (real (natfloor (abs x)) + 1))) =
    (%x. R (abs((abs x + 1) / (real (natfloor (abs x)) + 1)))))
  apply (erule ssubst)
  apply (rule subsetD)
  prefer 2
  apply (rule bigo-compose1 [OF R-bigo])
  apply (rule bigo-elt-subset)
  apply (rule subsetD)
```

prefer *2*
apply (*rule bigo-abs*)
apply (*rule bigo-elt-subset*)
apply (*rule-tac c = 2* **in** *bigo-bounded-alt*)
apply (*rule allI*)
apply (*rule real-ge-zero-div-gt-zero*)
apply *arith*
apply *force*
apply (*rule allI*)
apply *simp*
apply (*rule real-le-mult-imp-div-pos-le*)
apply *force*
apply *simp*
apply (*subst mult-commute*)
apply (*subgoal-tac abs x <= real(natfloor(abs x)) + 1*)
apply *force*
apply (*rule real-natfloor-plus-one-ge*)
apply (*rule ext*)
apply (*subst abs-nonneg*)
apply (*rule real-ge-zero-div-gt-zero*)
apply *arith*
apply *force*
apply (*rule refl*)
done

lemma *error6-aux2*: (%*n*::*nat*. *real* (*n* + *1*) * *ln* (*real* (*n* + *1*)) −
(*real n*) * *ln* (*real n*)) =*o* (%*n*. *ln* (*real* (*n* + *1*))) +*o* *O*(%*n*. *1*)
apply (*rule set-minus-imp-plus*)
apply (*subst func-diff*)
apply (*rule bigo-bounded*)
apply (*rule allI*)
apply (*case-tac x = 0*)
apply *simp*
apply (*simp add*: *left-distrib*)
apply (*rule allI*)
apply (*case-tac x = 0*)
apply *simp*
apply (*simp add*: *left-distrib*)
apply (*subgoal-tac real x * (ln*(*real x* + *1*) − *ln*(*real x*)) <= *1*)
apply (*simp add*: *diff-minus right-distrib*)
apply (*subst ln-div* [*THEN sym*])
apply *force*
apply *force*
apply (*subst mult-commute*)
apply (*subst pos-le-divide-eq* [*THEN sym*])

```
  apply force
  apply (subst add-divide-distrib)
  apply simp
done

lemma error6: (%x. ∑ n = 1..natfloor (abs x) + 1.
      Lambda n * ln (real n) * abs (R ((abs x + 1) / real n))) +
    (%x. ∑ c = 1..natfloor (abs x) + 1. ∑ v | v dvd c.
      Lambda v * Lambda (c div v) * abs (R ((abs x + 1) / real c))) =o
    (%x. 2 * (∑ n = 1..natfloor (abs x) + 1.
      abs (R ((abs x + 1) / real n)) * ln (real (n - 1 + 1)))) +o
    O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
proof -
  have ((%x. ∑ n = 1..natfloor (abs x) + 1.
     Lambda n * ln (real n) * abs (R ((abs x + 1) / real n))) +
    (%x. ∑ c = 1..natfloor (abs x) + 1.
     ∑ v | v dvd c.
       Lambda v * Lambda (c div v) * abs (R ((abs x + 1) / real c)))) =
    (%x. ∑n = 1..natfloor (abs x) + 1.
     abs(R ((abs x + 1) / real n)) * (rho n - rho (n - 1)))
    apply (subst func-plus)
    apply (rule ext)
    apply (subst setsum-addf [THEN sym])
    apply (rule setsum-cong2)
    apply (subst aux3)
    apply force
    apply (simp add: ring-eq-simps)
    apply (subst setsum-const-times [THEN sym])
    apply (rule setsum-cong2)
    apply simp
    done
  also have ... =
    (%x. abs (R ((abs x + 1) / (real (natfloor (abs x)) + 1))) *
        rho (natfloor (abs x) + 1) +
      (∑ n = 1..natfloor (abs x). rho n *
        (abs (R ((abs x + 1) / real n)) -
          abs (R ((abs x + 1) / (real n + 1))))))
    apply (rule ext)
    apply (subst another-partial-sum)
    apply (simp add: rho-zero)
    done
  also have ... =
      (%x. abs (R ((abs x + 1) / (real (natfloor (abs x)) + 1)))) *
          (%x. rho (natfloor (abs x) + 1)) +
      (%x. ∑ n = 1..natfloor (abs x).
```

```
      (abs (R ((abs x + 1) / real n)) −
        abs (R ((abs x + 1) / (real n + 1)))) ∗ rho n)
  apply (subst func-times)
  apply (subst func-plus)
  apply (rule ext)
  apply (simp add: mult-ac)
  done
also have ... =o
  ((%x. abs (R ((abs x + 1) / (real (natfloor (abs x)) + 1)))) ∗o
    ((%x. 2 ∗ real (natfloor (abs x) + 1) ∗
      ln (real (natfloor (abs x) + 1))) +o
       O(%x. real (natfloor(abs x) + 1)))) +
  ((%x. ∑ n = 1..natfloor (abs x).
      (abs (R ((abs x + 1) / real n)) −
        abs (R ((abs x + 1) / (real n + 1)))) ∗
          (2 ∗ (real n) ∗ ln(real n))) +o
     O(%x. ∑ n = 1..natfloor (abs x).
      abs((abs (R ((abs x + 1) / real n)) −
        abs (R ((abs x + 1) / (real n + 1)))) ∗ real n)))
    (is ?LHS =o ?term1 + (?term2 +o ?term3))
  apply (rule set-plus-intro)
  apply (rule set-times-intro2)
  apply (rule bigo-compose2 [OF rho-bigo])
  apply (rule bigo-setsum4 [OF rho-bigo])
  done
also have ... <= (O(%x. 1) ∗ O(%x. (abs x + 1) ∗ (1 + ln (abs x + 1)))) +
    (?term2 +o ?term3)
  apply (rule set-plus-mono2)
  apply (rule set-times-mono5)
  apply (rule error6-aux)
  apply (rule bigo-plus-absorb2)
  apply (rule-tac c = 2 in bigo-bounded-alt)
  apply (rule allI)
  apply (rule nonneg-times-nonneg)
  apply force
  apply (rule ln-ge-zero)
  apply force
  apply (rule allI)
  apply (subst mult-assoc)
  apply (rule mult-left-mono)
  apply (rule mult-mono)
  apply simp
  apply (rule real-natfloor-le)
  apply force
  apply (subgoal-tac
```

ln (real (natfloor (abs x) + 1)) <= ln (abs x + 1))
apply *arith*
apply (*subst ln-le-cancel-iff*)
apply *force*
apply *arith*
apply *simp*
apply (*rule real-natfloor-le*)
apply *force*
apply *arith*
apply *force*
apply *force*
apply (*rule subset-trans*)
apply (*subgoal-tac ?t <= O(%x. abs x + 1)*)
apply *assumption*
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply *clarsimp*
apply (*rule real-natfloor-le*)
apply *force*
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply (*rule allI*)
apply *arith*
apply (*rule allI*)
apply (*subst right-distrib*)
apply *simp*
apply (*rule nonneg-times-nonneg*)
apply *arith*
apply *force*
apply (*rule order-refl*)
done
also have *... <= O(%x. (abs x + 1) * (1 + ln (abs x + 1))) +*
(?term2 +o ?term3)
apply (*rule set-plus-mono2*)
apply (*rule subset-trans*)
apply (*rule bigo-mult*)
apply (*simp add*: *func-times*)
apply (*rule order-refl*)
done
also have *... = ?term2 +o (?term3 +*
*O(%x. (abs x + 1) * (1 + ln (abs x + 1))))*
(**is** *?temp = ?term2 +o (?term3 + ?term4)*)
by (*simp only*: *set-plus-rearranges plus-ac0*)
also have *?term2 = (%x. 2) * (*

(− (%x. (abs (R ((abs x + 1) / (real (natfloor (abs x)) + 1)))))) *
(%x. ((real (natfloor (abs x)) + 1) *
ln (real (natfloor (abs x)) + 1))) +
(%x. ∑ n=1..natfloor (abs x) + 1.
abs (R ((abs x + 1) / real n)) * ((real n) * ln (real n) −
(real (n − 1) * ln (real (n − 1))))))
(**is** *?term2* = (%x. 2) * *?term2a*)
apply (*simp only*: *func-minus func-plus func-times*)
apply (*rule ext*)
apply (*subst another-partial-sum*)
apply *simp*
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
done
also have *?term2a* =
(− (%x. (abs (R ((abs x + 1) / (real (natfloor (abs x)) + 1)))))) *
(%x. ((real (natfloor (abs x)) + 1) *
ln (real (natfloor (abs x)) + 1))) +
(%x. ∑ n=1..natfloor (abs x) + 1.
abs (R ((abs x + 1) / real n)) * ((real ((n − 1) + 1)) *
ln (real ((n − 1)+1)) −
(real (n − 1) * ln (real (n − 1))))))
(**is** *?term2a* = *?term2b*)
apply (*simp only*: *func-times func-minus func-plus*)
apply (*rule ext*)
apply (*rule arg-cong*)**back**
apply (*rule setsum-cong2*)
apply (*subgoal-tac xa − 1 + 1 = xa*)
apply (*erule ssubst*)
apply (*rule refl*)
apply *simp*
done
also have (%x. 2) * *?term2b* +o (*?term3* + *?term4*) <=
(%x. 2) *o (O(%x. 1) * *?term4* +
((%x. ∑ n = 1..natfloor (abs x) + 1.
abs (R ((abs x + 1) / real n)) * ln (real (n − 1 + 1)))
+o O(%x. ∑ n = 1..natfloor (abs x) + 1.
abs (R ((abs x + 1) / real n)) * 1))) + (*?term3* + *?term4*)
apply (*rule set-plus-mono3*)
apply (*rule set-times-intro2*)
apply (*rule set-plus-intro*)
apply (*rule set-times-intro*)
apply (*rule bigo-minus*)
apply (*rule error6-aux*)

apply (*rule bigo-bounded*)
apply (*rule allI*)
apply (*rule nonneg-times-nonneg*)
apply *force*
apply *force*
apply (*rule allI*)
apply (*rule mult-mono*)
apply (*force intro*: *real-natfloor-le*)
apply (*subgoal-tac ln* (*real* (*natfloor* (*abs x*)) + *1*) <= *ln* (*abs x* + *1*))
apply *arith*
apply (*subst ln-le-cancel-iff*)
apply *force*
apply *arith*
apply (*force intro*: *real-natfloor-le*)
apply *arith*
apply *force*
apply (*rule bigo-setsum6*)
apply (*rule bigo-compose2* [*OF error6-aux2*])
apply *force*
apply *force*
done
also have ... <= (%*x*. *2* ∗ (∑ *n* = *1*..*natfloor* (*abs x*) + *1*.
abs (*R* ((*abs x* + *1*) / *real n*)) ∗ *ln* (*real* (*n* − *1* + *1*)))) +*o*
O(%*x*. (*abs x* + *1*) ∗ (*1* + *ln* (*abs x* + *1*))) +
(*O*(%*x*. (*abs x* + *1*) ∗ (*1* + *ln* (*abs x* + *1*))) +
(*O*(%*x*. (*abs x* + *1*) ∗ (*1* + *ln* (*abs x* + *1*))) + *?term3*))
apply (*simp only*: *set-times-plus-distribs func-times*
set-plus-rearranges plus-ac0)
apply (*rule set-plus-mono*)
apply (*rule set-plus-mono2*)
apply (*rule order-refl*)
apply (*rule set-plus-mono2*)
apply (*rule subset-trans*)
prefer *2*
apply (*subgoal-tac* (%*x*. *2*) ∗*o* *?t* <= *?t*)
apply *assumption*
apply (*rule bigo-const-mult6*)
apply (*rule set-times-mono*)
apply (*rule order-trans*)
apply (*rule bigo-mult*)
apply (*simp add*: *func-times*)
apply (*subst bigo-const-mult5*)
apply *force*
apply (*rule set-plus-mono2*)
apply *simp*

apply (*rule bigo-elt-subset*)
apply (*subgoal-tac* (%*x*. $\sum$ *n* = *1..1* + *natfloor* (*abs x*).
abs (*R* ((*1* + *abs x*) / *real n*))) =
(%*x*. $\sum$ *n* = *1..natfloor* (*abs x*) + *1*.
1 ∗ *abs* (*R* (*abs*((*abs x* + *1*) / *real n*)))))
apply (*erule ssubst*)
apply (*rule subsetD*)
prefer *2*
apply (*subgoal-tac* (%*x*. $\sum$ *n* = *1..natfloor* (*abs x*) + *1*.
1 ∗ *abs* (*R* (*abs*((*abs x* + *1*) / *real n*)))) =*o*
O(%*x*. $\sum$ *n* = *1..natfloor* (*abs x*) + *1*.
1 ∗ *abs*((*abs x* + *1*) / *real n*)))
apply *assumption*
apply (*rule bigo-setsum5*)**back**
apply (*rule subsetD*)
prefer *2*
apply (*rule bigo-abs*)
apply (*rule bigo-elt-subset*)
apply (*rule R-bigo*)
apply *force*
apply *force*
apply (*rule bigo-elt-subset*)
apply (*subgoal-tac* (%*x*. $\sum$ *n* = *1..natfloor* (*abs x*) + *1*.
1 ∗ *abs* ((*abs x* + *1*) / *real n*)) =
(%*x*. *abs x* + *1*) ∗
(%*x*. $\sum$ *n* = *1..natfloor* (*abs x*) + *1*. *1* / *real n*))
apply (*erule ssubst*)
apply (*rule subsetD*)
apply (*subgoal-tac* (%*x*. *abs x* + *1*) ∗*o* *O*(%*x*. *1* + *ln* (*1* + *abs x*)) <=
O(%*x*. (*abs x* + *1*) ∗ (*1* + *ln* (*1* + *abs x*))))
apply (*simp add*: *add-ac*)
apply (*rule subset-trans*)
apply (*rule bigo-mult2*)
apply (*simp add*: *func-times*)
apply (*simp add*: *add-ac*)
apply (*rule set-times-intro2*)
apply (*subgoal-tac* (%*x*. $\sum$ *n* = *1..1* + *natfloor* (*abs x*). *1* / *real n*)
= (%*x*. $\sum$ *n* = *1..natfloor* (*abs x*) + *1*. *1* / *real n*))
apply (*erule ssubst*)
apply (*rule subsetD*)
prefer *2*
apply (*rule bigo-add-commute-imp* [*OF ln-sum-real2*])
apply (*rule bigo-plus-absorb2*)
apply (*rule bigo-bounded*)
apply *force*

apply (*rule allI*)
apply (*simp add*: *add-commute*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply *force*
apply (*simp add*: *add-ac*)
apply (*rule ext*)
apply (*subst func-times*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
apply (*rule abs-nonneg*)
apply (*rule real-ge-zero-div-gt-zero*)
apply *arith*
apply *force*
apply (*rule ext*)
apply (*simp add*: *add-ac*)
apply (*rule setsum-cong2*)
apply (*subst abs-nonneg*)**back**
apply (*rule real-ge-zero-div-gt-zero*)
apply *arith*
apply *force*
apply (*rule refl*)
apply (*rule order-refl*)
done
also have ... = (%*x*. *2* ∗ (∑ *n* = *1*..*natfloor* (*abs x*) + *1*.
abs (*R* ((*abs x* + *1*) / *real n*)) ∗ *ln* (*real* (*n* − *1* + *1*)))) +*o*
(*O*(%*x*. (*abs x* + *1*) ∗ (*1* + *ln* (*abs x* + *1*))) + *?term3*)
apply (*simp only*: *set-plus-rearranges*)
apply (*subst plus-ac0.assoc* [*THEN sym*])
apply (*subst bigo-plus-idemp*)
apply (*subst plus-ac0.assoc* [*THEN sym*])
apply (*subst bigo-plus-idemp*)
apply (*rule refl*)
done
also have ... <= (%*x*. *2* ∗ (∑ *n* = *1*..*natfloor* (*abs x*) + *1*.
abs (*R* ((*abs x* + *1*) / *real n*)) ∗ *ln* (*real* (*n* − *1* + *1*)))) +*o*
O(%*x*. (*abs x* + *1*) ∗ (*1* + *ln* (*abs x* + *1*)))
apply (*rule set-plus-mono*)
apply (*rule bigo-plus-subset4*)
apply (*rule order-refl*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-lesseq3*)
prefer *2*

apply (*rule allI*)
apply (*rule setsum-nonneg*)
apply *force*
apply *clarify*
apply (*rule abs-ge-zero*)
prefer *2*
proof −
show *ALL x.* (∑ *n = 1..natfloor* (*abs x*).
  abs ((*abs* (*R* ((*abs x* + *1*) / *real n*)) −
    abs (*R* ((*abs x* + *1*) / (*real n* + *1*)))) ∗ *real n*)) <=
 (*abs x* + *1*) ∗
   (∑ *n = 1..natfloor* (*abs x*) + *1*. *1* / *real n*) +
  (∑ *n = 1..natfloor* (*abs x*) + *1*.
    psi (*natfloor* ((*abs x* + *1*) / *real n*)))
proof
 fix *x*
 have (∑ *n = 1..natfloor* (*abs x*).
   abs ((*abs* (*R* ((*abs x* + *1*) / *real n*)) −
     abs (*R* ((*abs x* + *1*) / (*real n* + *1*)))) ∗ *real n*)) <=
   (∑ *n = 1..natfloor* (*abs x*).
   abs ((*R* ((*abs x* + *1*) / *real n*) −
     R ((*abs x* + *1*) / (*real n* + *1*)))) ∗ *real n*)
  apply (*rule setsum-le-cong*)
  apply (*subst abs-mult*)+
  apply (*subgoal-tac abs*(*real x*) = *real x*)
  apply (*erule ssubst*)
  apply (*rule mult-right-mono*)
  apply (*rule abs-triangle-ineq3*)
  apply *force*
  apply *force*
  done
 also have ... = (∑ *n = 1..natfloor* (*abs x*).
   abs (
    (((*abs x* + *1*) / *real* (*n* + *1*)) − ((*abs x* + *1*) / (*real n*))) +
    (*psi*(*natfloor*((*abs x* + *1*) / *real n*)) −
     psi(*natfloor*((*abs x* + *1*) / (*real* (*n* + *1*)))))) ∗ *real n*)
  apply (*rule setsum-cong2*)
  apply (*unfold R-def*)
  apply (*simp add*: *ring-eq-simps compare-rls*)
  done
 also have ... <= (∑ *n = 1..natfloor* (*abs x*). *abs* (
   (((*abs x* + *1*) / *real* (*n* + *1*)) − ((*abs x* + *1*) / (*real n*))) ∗
    real n) +
  (∑ *n = 1..natfloor* (*abs x*). *abs* (
    (*psi*(*natfloor*((*abs x* + *1*) / *real n*)) −

```
      psi(natfloor((abs x + 1) / (real (n + 1)))))) * real n)
  apply (subst setsum-addf [THEN sym])
  apply (rule setsum-le-cong)
  apply (subst left-distrib [THEN sym])
  apply (rule mult-right-mono)
  apply (rule abs-triangle-ineq)
  apply force
  done
also have ... = (∑ n = 1..natfloor (abs x). (real n) *
    (((abs x + 1) / real n) - (abs x + 1) / real (n + 1))) +
  (∑ n = 1..natfloor (abs x). (real n) *
     (psi(natfloor((abs x + 1) / real n)) -
       psi(natfloor((abs x + 1) / (real (n + 1)))))
  apply (simp only: setsum-addf [THEN sym])
  apply (rule setsum-cong2)
  apply (subgoal-tac
    abs ((abs ?y + 1) / real (x + 1) - (abs ?y + 1) / real x) =
    -((abs ?y + 1) / real (x + 1) - (abs ?y + 1) / real x))
  apply (erule ssubst)
  apply (subgoal-tac abs (psi (natfloor ((abs ?y + 1) / real x)) -
          psi (natfloor ((abs ?y + 1) / real (x + 1)))) =
       (psi (natfloor ((abs ?y + 1) / real x)) -
          psi (natfloor ((abs ?y + 1) / real (x + 1)))))
  apply (erule ssubst)
  apply (simp add: ring-eq-simps)
  apply (rule abs-nonneg)
  apply (simp add: compare-rls)
  apply (rule psi-mono)
  apply (rule natfloor-mono)
  apply (rule divide-left-mono)
  apply force
  apply arith
  apply (rule mult-pos)
  apply force
  apply force
  apply (rule abs-nonpos)
  apply (simp add: compare-rls)
  apply (rule divide-left-mono)
  apply force
  apply arith
  apply (rule mult-pos)
  apply force
  apply force
  done
also have ... = (∑ n = 1..natfloor (abs x) + 1.
```

```
    (abs x + 1) * (real n - real (n - 1)) / real n) -
      (abs x + 1) +
    ((∑ n = 1..natfloor (abs x) + 1.
      psi (natfloor ((abs x + 1) / real n)) * (real n - real (n - 1))) -
      psi (natfloor ((abs x + 1) / (real (natfloor (abs x)) + 1))) *
      (real (natfloor (abs x)) + 1))
   apply (subst another-partial-sum2)
   apply (subst another-partial-sum2)
   apply simp
   done
  also have ... = (abs x + 1) *
      (∑ n = 1..natfloor (abs x) + 1. 1 / real n) +
    (∑ n = 1..natfloor (abs x) + 1.
      psi (natfloor ((abs x + 1) / real n))) -
    ((abs x + 1) +
      psi (natfloor ((abs x + 1) / (real (natfloor (abs x)) + 1))) *
        (real (natfloor (abs x)) + 1))
   apply (simp add: compare-rls)
    apply (simp add: setsum-addf [THEN sym] setsum-const-times [THEN
sym])
   apply (rule setsum-cong2)
   apply (subgoal-tac real x - real (x - 1) = 1)
   apply (erule ssubst)
   apply simp
   apply (subgoal-tac real (x - 1) = real x - real (1::nat))
   apply simp
   apply (rule real-of-nat-diff)
   apply force
   done
  also have ... <= (abs x + 1) *
      (∑ n = 1..natfloor (abs x) + 1. 1 / real n) +
    (∑ n = 1..natfloor (abs x) + 1.
      psi (natfloor ((abs x + 1) / real n)))
   apply (simp only: compare-rls)
   apply (subst add-commute)
   apply (rule add-increasing)
   apply (rule nonneg-plus-nonneg)
   apply arith
   apply (rule nonneg-times-nonneg)
   apply (rule psi-ge-zero)
   apply force
   apply (rule order-refl)
   done
  finally show (∑ n = 1..natfloor (abs x).
    abs ((abs (R ((abs x + 1) / real n)) -
```

```
      abs (R ((abs x + 1) / (real n + 1)))) * real n))
    <= (abs x + 1) * (∑ n = 1..natfloor (abs x) + 1. 1 / real n) +
      (∑ n = 1..natfloor (abs x) + 1.
        psi (natfloor ((abs x + 1) / real n))).
 qed
next show (%x. (abs x + 1) *
   (∑ n = 1..natfloor (abs x) + 1. 1 / real n) +
     (∑ n = 1..natfloor (abs x) + 1.
      psi (natfloor ((abs x + 1) / real n)))) =o
   O(%x. (abs x + 1) * (1 + ln (abs x + 1)))
 apply (subst bigo-plus-idemp [THEN sym])
 apply (subgoal-tac
   (%x. (abs x + 1) * (∑ n = 1..natfloor (abs x) + 1. 1 / real n) +
     (∑ n = 1..natfloor (abs x) + 1.
       psi (natfloor ((abs x + 1) / real n)))) =
    (%x. (abs x + 1) * (∑ n = 1..natfloor (abs x) + 1. 1 / real n)) +
    (%x. (∑ n = 1..natfloor (abs x) + 1.
      psi (natfloor ((abs x + 1) / real n)))))
 apply (erule ssubst)
 apply (rule set-plus-intro)
 apply (rule subsetD)
 apply (subgoal-tac (%x. (abs x + 1)) *o O(%x. 1 + ln(abs x + 1)) <= ?t)
 apply assumption
 apply (rule subset-trans)
 apply (rule bigo-mult2)
 apply (simp add: func-times)
 apply (subgoal-tac
   (%x. (abs x + 1) * (∑ n = 1..natfloor (abs x) + 1. 1 / real n)) =
    (%x. (abs x + 1)) *
     (%x. ∑ n = 1..natfloor (abs x) + 1. 1 / real n))
 apply (erule ssubst)
 apply (rule set-times-intro2)
 apply (rule subsetD)
 prefer 2
 apply (rule bigo-add-commute-imp [OF ln-sum-real2])
 apply (rule bigo-plus-absorb2)
 apply (rule bigo-bounded)
 apply force
 apply (rule allI)
 apply (simp add: add-commute)
 apply (rule bigo-elt-subset)
 apply (rule bigo-bounded)
 apply force
 apply force
 apply (simp add: add-ac)
```

apply (*rule ext*)
apply (*subst func-times*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
apply (*rule subsetD*)
prefer *2*
apply (*subgoal-tac* (%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*.
    1 ∗ *psi* (*natfloor* ((*abs x* + *1*) / *real n*))) =*o*
  O(%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*.
     abs(*1* ∗ *real*(*natfloor* ((*abs x* + *1*) / *real n*)))))
apply *simp*
apply (*rule bigo-setsum3* [*OF psi-bigo*])
apply (*rule bigo-elt-subset*)
apply (*subgoal-tac* (%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*.
    real (*natfloor* ((*abs x* + *1*) / *real n*))) =
  (%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*.
    real (*natfloor* (*abs x* + *1*) *div n*)))
apply (*erule ssubst*)
apply (*rule bigo-lesseq3*)
apply (*subgoal-tac* (%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*.
    real (*natfloor* (*abs x* + *1*)) / (*real n*)) =*o* *?t*)
apply *assumption*
apply (*rule subsetD*)
apply (*subgoal-tac* *O*(%*x*. (*abs x* + *1*)) ∗ *O*(%*x*. (*1* + *ln* (*abs x* + *1*))) <=
  ?t)
apply *assumption*
apply (*rule subset-trans*)
apply (*rule bigo-mult*)
apply (*simp add*: *func-times*)
apply (*subgoal-tac* (%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*.
    real (*natfloor* (*abs x* + *1*)) / *real n*) =
  (%*x*. *real* (*natfloor* (*abs x* + *1*))) ∗
   (%*x*. $\sum$ *n* = *1*..*natfloor* (*abs x*) + *1*. *1* / *real n*))
apply (*erule ssubst*)
apply (*rule set-times-intro*)
apply (*rule bigo-bounded*)
apply *force*
apply (*rule allI*)
apply (*rule real-natfloor-le*)
apply *arith*
apply (*rule subsetD*)
prefer *2*
apply (*rule bigo-add-commute-imp* [*OF ln-sum-real2*])
apply (*rule bigo-plus-absorb2*)

apply (*rule bigo-bounded*)
apply *force*
apply (*rule allI*)
apply (*simp add*: *add-commute*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply *force*
apply (*subst func-times*)
apply (*rule ext*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
apply (*rule allI*)
apply (*rule setsum-nonneg'*)
apply *force*
apply (*rule allI*)
apply (*rule setsum-le-cong*)
apply (*rule real-nat-div-le-real-div*)
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst natfloor-div-nat*)
apply *force+*
apply (*subst func-plus*)
apply (*rule refl*)
done
qed
finally show *?thesis*.
qed

lemma *error7*: (%*x*. *abs* (*R* (*abs x* + *1*)) ∗ *ln* (*abs x* + *1*) ^ *2*) <*o*
(%*x*. *2* ∗ (∑ *n* = *1*..*natfloor* (*abs x*) + *1*.
abs (*R* ((*abs x* + *1*) / *real n*)) ∗ *ln* (*real n*))) =*o*
O(%*x*. (*abs x* + *1*) ∗ (*1* + *ln* (*abs x* + *1*)))
proof −
note *bigo-lesso4* [*OF error-cor1*, *OF error6*]
also have (%*x*. *2* ∗ (∑ *n* = *1*..*natfloor* (*abs x*) + *1*.
abs (*R* ((*abs x* + *1*) / *real n*)) ∗ *ln* (*real* (*n* − *1* + *1*)))) =
(%*x*. *2* ∗ (∑ *n* = *1*..*natfloor* (*abs x*) + *1*.
abs (*R* ((*abs x* + *1*) / *real n*)) ∗ *ln* (*real n*)))
apply (*rule ext*)
apply (*rule arg-cong*)**back**
apply (*rule setsum-cong2*)
apply (*subgoal-tac xa* − *1* + *1* = *xa*)
apply (*erule ssubst*)

```
    apply (rule refl)
    apply simp
    done
  finally show ?thesis.
qed

end
```

42 The prime number theorem

```
theory PrimeNumberTheorem = Error:

lemma PNT1-aux1: 0 < C ==> 0 <= D ==>
  EX C0. ALL eps. 0 < eps --> eps < 1 -->
    (ALL K. exp (C0 / eps) < K -->
      2 < K & D < ln K & C / (ln K - D) < eps)
proof
  assume a: 0 < C and b: 0 <= D
  let ?C0 = 3 * max (max C D) (ln 2)
  show ALL eps. 0 < eps --> eps < 1 -->
    (ALL K. exp (?C0 / eps) < K -->
      2 < K & D < ln K & C / (ln K - D) < eps)
  proof (clarify)
    fix eps::real
    fix K::real
    assume 0 < eps and eps < 1
    assume exp (?C0 / eps) < K
    show 2 < K & D < ln K & C / (ln K - D) < eps
      apply (rule conjI)
      apply (rule order-le-less-trans)
      prefer 2
      apply (rule prems)
      apply (subgoal-tac 2 = exp (ln 2))
      apply (erule ssubst)
      apply (subst exp-le-cancel-iff)
      apply (subst pos-le-divide-eq)
      apply (rule prems)
      apply (rule order-trans)
      apply (subgoal-tac ln 2 * eps <= ln 2 * 1)
      apply assumption
      apply (rule mult-left-mono)
      apply (rule order-less-imp-le)
```

apply (*rule prems*)
apply *force*
apply (*subst mult-commute*)
apply (*rule mult-mono*)
apply *simp*
apply (*rule le-maxI2*)
apply *force*
apply *simp*
apply (*rule order-trans*)
prefer *2*
apply (*rule le-maxI2*)
apply *auto*
apply (*rule order-le-less-trans*)
prefer *2*
apply (*subgoal-tac ln* (*exp* (*?C0* / *eps*)) < *ln K*)
apply *assumption*
apply (*subst ln-less-cancel-iff*)
apply *force*
apply (*rule order-less-trans*)
prefer *2*
apply (*rule prems*)
apply *force*
apply (*rule prems*)
apply *simp*
apply (*rule order-trans*)
apply (*subgoal-tac D* <= *3* ∗ *D*)
apply *assumption*
apply *simp*
apply (*rule prems*)+
apply (*subst pos-le-divide-eq*)
apply (*rule prems*)
apply *simp*
apply (*rule order-trans*)
apply (*subgoal-tac D* ∗ *eps* <= *D* ∗ *1*)
apply *assumption*
apply (*rule mult-left-mono*)
apply (*rule order-less-imp-le*)
apply (*rule prems*)
apply (*rule prems*)
apply *simp*
apply (*rule order-trans*)
apply (*rule le-maxI2*)
apply (*rule le-maxI1*)
apply (*subst pos-divide-less-eq*)
apply *simp*

apply (*rule order-le-less-trans*)
prefer *2*
apply (*subgoal-tac ln* (*exp* (*?C0* / *eps*)) < *ln K*)
apply *assumption*
apply (*subst ln-less-cancel-iff*)
apply *force*
apply (*rule order-less-trans*)
prefer *2*
apply (*rule prems*)
apply *force*
apply (*rule prems*)
apply *simp*
apply (*rule order-trans*)
apply (*subgoal-tac D <= 3 * D*)
apply *assumption*
apply *simp*
apply (*rule prems*)+
apply (*subst pos-le-divide-eq*)
apply (*rule prems*)
apply *simp*
apply (*rule order-trans*)
apply (*subgoal-tac D * eps <= D * 1*)
apply *assumption*
apply (*rule mult-left-mono*)
apply (*rule order-less-imp-le*)
apply (*rule prems*)
apply (*rule prems*)
apply *simp*
apply (*rule order-trans*)
apply (*rule le-maxI2*)
apply (*rule le-maxI1*)
apply (*subgoal-tac C = eps * (C / eps)*)
apply (*erule ssubst*)
apply (*rule mult-strict-left-mono*)
apply (*simp add*: *compare-rls*)
apply (*rule order-less-le-trans*)
prefer *2*
apply (*subst ln-le-cancel-iff*)
prefer *3*
apply (*rule order-less-imp-le*)
apply (*rule prems*)
apply *force*
apply (*rule order-less-trans*)
prefer *2*
apply (*rule prems*)

apply *force*
apply *simp*
apply (*subgoal-tac C / eps <= max (max C D) (ln 2) / eps*)
apply (*subgoal-tac D <= max (max C D) (ln 2) / eps*)
apply (*subgoal-tac 0 < max (max C D) (ln 2) / eps*)
apply (*subgoal-tac 3 * max (max C D) (ln 2) / eps =*
*2 * (max (max C D) (ln 2) / eps) + max (max C D) (ln 2) / eps*)
apply (*erule ssubst*)
apply *arith*
apply (*subgoal-tac 3 = 2 + 1*)
apply (*erule ssubst*)
apply (*simp only*: *ring-eq-simps add-divide-distrib*)
apply *simp+*
apply (*subst pos-less-divide-eq*)
apply (*rule prems*)
apply *simp*
apply (*rule order-less-le-trans*)
apply (*rule a*)
apply (*rule order-trans*)
apply (*rule le-maxI1*)
apply (*rule le-maxI1*)
apply (*subst pos-le-divide-eq*)
apply (*rule prems*)
apply (*rule order-trans*)
apply (*subgoal-tac D * eps <= D * 1*)
apply *assumption*
apply (*rule mult-left-mono*)
apply (*rule order-less-imp-le*)
apply (*rule prems*)*+*
apply *simp*
apply (*rule order-trans*)
apply (*rule le-maxI2*)
apply (*rule le-maxI1*)
apply (*rule divide-right-mono*)
apply (*rule order-trans*)
apply (*rule le-maxI1*)
apply (*rule le-maxI1*)
apply (*rule order-less-imp-le*)
apply (*rule prems*)*+*
apply (*subgoal-tac eps ~= 0*)
apply *simp*
apply (*rule less-imp-neq* [*THEN not-sym*])
apply (*rule prems*)
done
qed

qed

lemma *PNT1-aux2*: *EX C. ALL x. abs(ln (abs x + 1) −*
(∑ i = 1..natfloor (abs x) + 1. 1 / real i)) < C
apply (*insert ln-sum-real2*)
apply (*drule set-plus-imp-minus*)
apply (*simp only*: *func-diff*)
apply (*simp only*: *bigo-alt-def*)
apply *clarsimp*
apply (*rule-tac x = c + c* **in** *exI*)
apply (*rule allI*)
apply (*drule-tac x = x* **in** *spec*)
apply (*erule order-le-less-trans*)
apply *arith*
done

lemma *PNT1-aux3*: *EX C. ALL x. (1 <= x −−> abs(ln x −*
(∑ i = 1..natfloor x. 1 / real i)) < C)
apply (*insert PNT1-aux2*)
apply *clarsimp*
apply (*rule-tac x = C* **in** *exI*)
apply (*rule allI*)
apply (*rule impI*)
apply (*subgoal-tac ln x = ln((x − 1) + 1)*)
apply (*erule ssubst*)
apply (*subgoal-tac natfloor x = natfloor (x − 1) + 1*)
apply (*erule ssubst*)
apply (*subgoal-tac x − 1 = abs(x − 1)*)
apply (*erule ssubst*)
apply (*erule spec*)
apply (*subst abs-nonneg*)
apply *auto*
apply (*subst natfloor-add* [*THEN sym*])
apply *auto*
done

lemma *PNT1-aux4*: *EX C. ALL x. (1 <= x −−> abs(ln x −*
(∑ i = 1..natfloor x. 1 / real (i + 1))) < C)
apply (*insert PNT1-aux3*)
apply *clarify*
apply (*rule-tac x = C + 1* **in** *exI*)
apply *clarify*
apply (*drule-tac x = x* **in** *spec*)
apply (*subgoal-tac*
(∑ i = 2..natfloor x + 1. 1 / real i) =

$(\sum\ i = 1..natfloor\ x.\ 1\ /\ real\ (i + 1)))$
prefer *2*
apply (*rule setsum-reindex-cong'*)
apply *force*
apply (*subgoal-tac inj-on* (%*x*. *x* + *1*) *?t*)
apply *assumption*
apply (*simp add*: *inj-on-def*)
apply (*clarsimp simp add*: *image-def*)
apply (*rule equalityI*)
apply *clarify*
apply (*rule-tac x* = *xa* − *1* **in** *bexI*)
apply *force*
apply *clarsimp*
apply (*rule conjI*)
apply *arith*
apply *arith*
apply *clarsimp*
apply (*rule refl*)
apply (*erule subst*)
apply (*subgoal-tac ln x* − $(\sum\ i = 2..natfloor\ x + 1.\ 1\ /\ real\ i)$ =
$(ln\ x - (\sum\ i = 1..natfloor\ x.\ 1\ /\ real\ i)) +$
$(1 - 1\ /\ real\ (natfloor\ x + 1)))$
apply (*erule ssubst*)
apply (*rule order-le-less-trans*)
apply (*rule abs-triangle-ineq*)
apply (*rule real-add-less-le-mono*)
apply (*erule mp*)
apply *assumption*
apply (*subst abs-nonneg*)
apply *simp*
apply (*subst pos-divide-le-eq*)
apply *force*
apply *force*
apply *simp*
apply (*simp add*: *compare-rls*)
apply (*subgoal-tac* $(\sum\ i = 2..natfloor\ x + 1.\ 1\ /\ real\ i)$ =
$(\sum\ i = 1..natfloor\ x.\ 1\ /\ real\ i) -$
$(\sum\ i : \{1::nat\}.\ 1\ /\ real\ i) +$
$(\sum\ i : \{natfloor\ x + 1\}.\ 1\ /\ real\ i))$
apply (*erule ssubst*)
apply *simp*
apply (*simp only*: *compare-rls*)
apply (*subst setsum-Un-disjoint* [*THEN sym*])
apply *force*
apply *force*

```
  apply force
  apply (subst setsum-Un-disjoint [THEN sym])
  apply force
  apply force
  apply force
  apply (rule setsum-cong)
  apply force
  apply (rule refl)
done

lemma PNT1-aux5: EX D. ALL x. ALL K. 1 <= x --> 1 <= K -->
  abs (ln K -
    (∑ n | natfloor x < n & n <= natfloor (K * x). 1 / real (n + 1)))
      <= D
proof -
  obtain C where ALL x. (1 <= x --> abs(ln x -
    (∑ i = 1..natfloor x. 1 / real (i + 1))) < C)
    by (insert PNT1-aux4, auto)
  show ?thesis
  proof
  show ALL x. ALL K. 1 <= x --> 1 <= K -->
      abs (ln K -
        (∑ n | natfloor x < n & n <= natfloor (K * x). 1 / real (n + 1)))
          <= C + C
    proof (clarify)
      fix x::real fix K::real
      assume 1 <= x and 1 <= K
    have a: (∑ n | natfloor x < n & n <= natfloor (K * x).
      1 / real (n + 1)) =
      (∑n=1..natfloor (K * x). 1 / real (n + 1)) -
      (∑n=1..natfloor x. 1 / real (n + 1))
        (is ?LHS1 = ?RHS1a - ?RHS1b)
      apply (simp add: compare-rls)
      apply (subst setsum-Un-disjoint [THEN sym])
      apply (rule bounded-nat-set-is-finite)
      apply clarsimp
      apply (subgoal-tac i < natfloor (K * x) + 1)
      apply assumption
      apply arith
      apply force
      apply force
      apply (rule setsum-cong)
      apply auto
      apply (rule order-trans)
      prefer 2
```

```
    apply (subgoal-tac natfloor (1 * ?x) <= natfloor (K * ?x))
    apply assumption
    apply (rule natfloor-mono)
    apply (rule mult-right-mono)
    apply (rule prems)
    apply (insert prems, arith)
    apply simp
    done
  have b: ln K = ln (K * x) - ln x
    apply (subst ln-mult)
    apply (insert prems)
    apply auto
    done
  have ln K - ?LHS1 = (ln (K * x) - ?RHS1a) + (?RHS1b - ln x)
    by (simp only: a b compare-rls)
  then have abs(ln K - ?LHS1) = abs(...)
    by simp
  also have ... <= abs(ln (K * x) - ?RHS1a) + abs(?RHS1b - ln x)
    by (rule abs-triangle-ineq)
  also have ... <= C + C
    apply (rule add-mono)
    apply (insert prems)
    apply (drule-tac x = K * x in spec)
    apply (subgoal-tac 1 <= K * x)
    apply force
    apply (subgoal-tac 1 * 1 <= K * x)
    apply simp
    apply (rule mult-mono)
    apply assumption+
    apply arith
    apply force
    apply (subst abs-diff)
    apply (drule-tac x = x in spec)
    apply force
    done
  finally show abs (ln K -
    (∑ n | natfloor x < n & n <= natfloor (K * x). 1 / real (n + 1)))
    <= C + C.
  qed
 qed
qed

lemma PNT1-aux6: EX D. ALL x. ALL K. 1 <= x --> 1 <= K --> ln K
- D <=
  (∑ n | natfloor x < n & n <= natfloor (K * x). 1 / real (n + 1))
```

```
  apply (insert PNT1-aux5)
  apply (erule exE)
  apply (rule-tac x = D in exI)
  apply (clarsimp)
  apply (drule-tac x = x in spec)
  apply clarsimp
  apply (drule-tac x = K in spec)
  apply clarsimp
  apply (frule abs-le-D1)
  apply auto
done

lemma PNT1-aux7: 0 < C ==> EX C0. ALL eps. 0 < eps --> eps < 1 -->
    (ALL K. ALL x. 1 <= x --> exp (C0 / eps) < K -->
      2 < K &
      C / (∑ n | natfloor x < n & n <= natfloor (K * x).
        1 / real (n + 1)) < eps)
proof -
  assume a: 0 < C
  obtain D where d: ALL x. ALL K. 1 <= K --> 1 <= x --> ln K - D
<=
    (∑ n | natfloor x < n & n <= natfloor (K * x). 1 / real (n + 1))
    by (insert PNT1-aux6, auto)
  let ?D = max D 0
  have EX C0. ALL eps. 0 < eps --> eps < 1 -->
      (ALL K. exp (C0 / eps) < K -->
        2 < K & ?D < ln K & C / (ln K - ?D) < eps)
    apply (rule PNT1-aux1)
    apply (rule a)
    apply (rule le-maxI2)
    done
  then obtain C0 where C0: ALL eps. 0 < eps --> eps < 1 -->
      (ALL K. exp (C0 / eps) < K -->
        2 < K & ?D < ln K & C / (ln K - ?D) < eps)..
  show ?thesis
  proof
  show ALL eps. 0 < eps --> eps < 1 -->
    (ALL K. ALL x. 1 <= x --> exp (C0 / eps) < K --> 2 < K &
      C / (∑ n | natfloor x < n & n <= natfloor (K * x).
        1 / real (n + 1)) < eps)
  proof (clarify)
    fix eps::real
    fix K::real
    fix x::real
    assume 0 < eps
```

assume *eps* < *1*
assume *1* <= *x*
assume *exp* (*C0* / *eps*) < *K*
have *c0*: *ALL K*. *exp* (*C0* / *eps*) < *K* −−> *2* < *K* & *?D* < *ln K* &
C / (*ln K* − *?D*) < *eps*
by (*insert C0 prems*, *blast*)
show *2* < *K* &
C / (∑ *n* | *natfloor x* < *n* & *n* <= *natfloor* (*K* ∗ *x*).
1 / *real* (*n* + *1*)) < *eps*
apply (*rule conjI*)
apply (*insert c0*)
apply (*drule-tac x* = *K* **in** *spec*)
apply (*force simp add*: *prems*)
apply (*case-tac*
(∑ *n* | *natfloor x* < *n* & *n* <= *natfloor* (*K* ∗ *x*).
1 / *real* (*n* + *1*)) = *0*)
apply (*simp add*: *prems*)
apply (*subst pos-divide-less-eq*)
apply (*subgoal-tac 0* <=
(∑ *n* | *natfloor x* < *n* & *n* <= *natfloor* (*K* ∗ *x*).
1 / *real* (*n* + *1*)))
apply *arith*
apply (*rule setsum-nonneg'*)
apply *force*
apply (*rule order-less-le-trans*)
apply (*subgoal-tac C* < *eps* ∗ (*ln K* − *max D 0*))
apply *assumption*
apply (*subst pos-divide-less-eq* [*THEN sym*])
apply (*simp add*: *compare-rls*)
apply (*insert d*)
apply (*drule-tac x* = *x* **in** *spec*)
apply (*drule-tac x* = *K* **in** *spec*)
apply (*clarsimp simp add*: *prems*)
apply (*insert c0*)
apply (*drule-tac x* = *K* **in** *spec*)
apply (*clarsimp simp add*: *prems*)
apply (*drule-tac x* = *K* **in** *spec*)
apply (*clarsimp simp add*: *prems*)
apply (*rule mult-left-mono*)
apply (*rule order-trans*)
apply (*subgoal-tac ln K* − *max D 0* <= *ln K* − *D*)
apply *assumption*
apply *simp*
apply (*rule le-maxI1*)
apply (*drule-tac x* = *x* **in** *spec*)

apply (*drule-tac x = K* **in** *spec*)**back**
apply (*subgoal-tac 1 <= K*)
apply (*clarsimp simp add*: *prems*)
apply (*subgoal-tac 2 < K*)
apply *simp*
apply (*force simp add*: *prems*)
apply (*rule order-less-imp-le*)
apply (*rule prems*)
done
qed
qed
qed

lemma *PNT1-aux8*: ~((*0 <= (x*::*real*)) = (*0 <= y*)) ==> *abs x <= abs* (*y* − *x*)
by *auto*

lemma *PNT1-aux9*:
(*ALL k < (j*::*nat*). ((*0 <= (f*(*i* + *k*)::*real*)) = (*0 <= f*(*i* + (*k* + *1*))))) ==>
(*ALL k <= j*. (*0 <= f*(*i* + *k*))) | (*ALL k <= j*. (*f*(*i* + *k*) < *0*))
apply (*case-tac 0 <= f i*)
apply (*rule disjI1*)
apply (*rule allI*)
apply (*induct-tac k*)
apply *force*
apply *clarify*
apply (*subgoal-tac n <= j*)
apply *clarify*
apply (*subgoal-tac n < j*)
apply (*drule-tac x = n* **in** *spec*)
apply (*frule mp*)
apply *assumption*
apply (*subgoal-tac i + Suc n = i + (n + 1)*)
apply (*erule ssubst*)**back**
apply *force*
apply *simp*
apply *arith*
apply *arith*
apply (*rule disjI2*)
apply (*rule allI*)
apply (*induct-tac k*)
apply *force*
apply *clarify*
apply (*subgoal-tac n <= j*)
apply *clarify*

```
  apply (subgoal-tac n < j)
  apply (drule-tac x = n in spec)
  apply (frule mp)
  apply assumption
  apply (subgoal-tac i + Suc n = i + (n + 1))
  apply (erule ssubst)back
  apply force
  apply simp
  apply arith
  apply arith
done

lemma PNT1-aux10: (i::nat) <= j ==> (ALL n. i < n & n <= j -->
    (((0::real) <= f(n)) = (0 <= f(n+1)))) ==>
      (ALL n. i < n & n <= j --> 0 <= f n) |
      (ALL n. i < n & n <= j --> f n < 0)
  apply (subgoal-tac
    (ALL k <= j - i. (0 <= f((i + 1) + k))) |
    (ALL k <= j - i. (f((i + 1) + k) < 0)))
  apply (erule disjE)
  apply (rule disjI1)
  apply clarify
  apply (drule-tac x = n - (i + 1) in spec)back
  apply (subgoal-tac n - (i + 1) <= j - i)
  apply clarify
  apply (subgoal-tac (i + 1) + (n - (i + 1)) = n)
  apply simp
  apply arith+
  apply (rule disjI2)
  apply clarify
  apply (drule-tac x = n - (i + 1) in spec)back
  apply (subgoal-tac n - (i + 1) <= j - i)
  apply clarify
  apply (subgoal-tac (i + 1) + (n - (i + 1)) = n)
  apply simp
  apply arith+
  apply (rule PNT1-aux9)
  apply clarify
  apply (drule-tac x = i + 1 + k in spec)
  apply (drule mp)
  apply arith
  apply (simp add: add-ac)
done

lemma PNT1-aux11: EX C. ALL i. (1::nat) <= i -->
```

```
    abs(∑ n=1..i. R(real n) / ((real n) * (real (n + 1)))) <= C
proof −
  have EX C. ALL x. abs(∑ n=1..natfloor (abs x).
      R(real n) / ((real n) * (real (n + 1)))) <= C
    apply (insert error1)
    apply (unfold bigo-def)
    apply clarsimp
    done
  thus ?thesis
    apply clarsimp
    apply (rule-tac x = C in exI)
    apply (rule allI)
    apply (rule impI)
    apply (drule-tac x = real i in spec)
    apply (subgoal-tac abs(real i) = real i)
    apply simp-all
    done
qed

lemma PNT1-aux12: EX C. ALL (i::nat). ALL j. 1 <= i --> i <= j -->
    abs (∑ n | i < n & n <= j. R(real n) / ((real n) * (real (n +
      1)))) < C
  apply (insert PNT1-aux11)
  apply (erule exE)
  apply (rule-tac x = C + C + 1 in exI)
  apply clarify
  apply (rule order-le-less-trans)
  prefer 2
  apply (subgoal-tac C + C < C + C + 1)
  apply assumption
  apply arith
  apply (subgoal-tac (∑ n | i < n & n <= j.
      R (real n) / (real n * real (n + 1))) =
    (∑ n=1..j. R(real n) / ((real n) * (real (n + 1)))) −
    (∑ n=1..i. R(real n) / ((real n) * (real (n + 1)))))
  apply (erule ssubst)
  apply (rule order-trans)
  apply (rule abs-triangle-ineq4)
  apply (rule add-mono)
  apply force
  apply force
  apply (simp add: compare-rls)
  apply (subst setsum-Un-disjoint [THEN sym])
  apply (rule bounded-nat-set-is-finite)
  apply clarify
```

```
  apply (subgoal-tac ia < j + 1)
  apply assumption
  apply arith
  apply simp
  apply force
  apply (rule setsum-cong)
  apply (auto simp only: atLeastAtMost-iff)
done

lemma PNT1: EX C0. ALL eps. 0 < eps --> eps < 1 --> (EX x0. ALL K
x.
    exp (C0 / eps) < K --> x0 < x --> (EX n.
      natfloor x < n & n <= natfloor (K * x) &
        abs(R(real n) / (real n)) < eps))
proof -
  from PNT1-aux12 obtain C where C: ALL (i::nat). ALL j.
    1 <= i --> i <= j -->
      abs (sum n | i < n & n <= j. R(real n) / ((real n) * (real (n +
        1)))) < C..
  have EX C0. ALL eps. 0 < eps --> eps < 1 -->
    (ALL K. ALL x. 1 <= x --> exp (C0 / eps) < K -->
      2 < K & C / (sum n | natfloor x < n & n <= natfloor (K * x).
        1 / real (n + 1)) < eps)
    apply (rule PNT1-aux7)
    apply (insert prems)
    apply (drule-tac x = 1 in spec)+
    apply (drule mp, simp)+
    apply arith
    done
  then obtain C0 where C0temp: ALL eps. 0 < eps --> eps < 1 -->
    (ALL K. ALL x. 1 <= x --> exp (C0 / eps) < K -->
      2 < K & C / (sum n | natfloor x < n & n <= natfloor (K * x).
        1 / real (n + 1)) < eps)..
  show ?thesis
  proof ((rule exI)+, clarify)
    fix eps::real
    assume 0 < eps
    assume eps < 1assume eps < 1
    show EX x0. ALL K x. exp (C0 / eps) < K --> x0 < x -->
      (EX n. natfloor x < n & n <= natfloor (K * x) &
        abs (R (real n) / real n) < eps)
  proof
    let ?x0 = max (exp 2 + 1) (exp (4 / eps) + 1)
    show ALL K x. exp (C0 / eps) < K --> ?x0 < x -->
      (EX n. natfloor x < n & n <= natfloor (K * x) &
```

```
    abs (R (real n) / real n) < eps)
proof (clarify)
 fix K::real fix x::real
 assume exp (C0 / eps) < K
 assume ?x0 < x
 have C0: ALL K. ALL x. 1 <= x --> exp (C0 / eps) < K -->
   2 < K &
   C / (∑ n | natfloor x < n & n <= natfloor (K * x).
     1 / real (n + 1)) < eps
   by (insert C0temp prems, blast)
 have a: exp 2 + 1 <= x
   apply (intro order-less-imp-le)
   apply (rule order-le-less-trans)
   prefer 2
   apply assumption
   apply (rule le-maxI1)
   done
 have b: exp (4 / eps) + 1 <= x
   apply (intro order-less-imp-le)
   apply (rule order-le-less-trans)
   prefer 2
   apply (rule prems)
   apply (rule le-maxI2)
   done
 have d: exp (4 / eps) <= real(natfloor x)
   apply (rule order-less-imp-le)
   apply (rule ge-natfloor-plus-one-imp-gt)
   apply (subst natfloor-add [THEN sym])
   apply (subgoal-tac 0 < exp (4 / eps))
   apply arith
   apply force
   apply simp
   apply (rule natfloor-mono)
   apply (rule b)
   done
 show EX n. natfloor x < n &
   n <= natfloor (K * x) & abs (R (real n) / real n) < eps
 proof (cases)
   assume EX n. natfloor x < n & n <= natfloor (K * x) &
     ~((0 <= R(real n)) = (0 <= R(real (n + 1))))
   then obtain n where c: natfloor x < n & n <= natfloor (K * x) &
     ~((0 <= R(real n)) = (0 <= R(real (n + 1))))..
   have c: exp 2 < real n
     apply (rule ge-natfloor-plus-one-imp-gt)
     apply (subgoal-tac natfloor x < n)
```

apply (*subgoal-tac natfloor* (*exp 2*) + *1* <= *natfloor x*)
apply *arith*
apply (*subst natfloor-add* [*THEN sym*])
apply *force*
apply (*rule natfloor-mono*)
apply *simp*
apply (*rule a*)
apply (*insert prems*, *auto*)
done
have *abs* (*R*(*real n*)) <= *abs*(*R*(*real* (*n* + *1*)) − *R*(*real n*))
by (*insert prems*, *intro PNT1-aux8*, *simp*)
also have *R*(*real*(*n* + *1*)) − *R*(*real n*) = *psi*(*n* + *1*) − *psi*(*n*) − *1*
apply (*unfold R-def*)
apply *simp*
apply (*subgoal-tac real n* + *1* = *real*(*n* + *1*))
apply (*erule ssubst*)
apply (*subst natfloor-real-id*)
apply *simp-all*
done
also have *abs*(...) < *ln* (*real n*)
apply (*subst psi-plus-one*)
apply *simp*
apply (*case-tac EX p a*. *p* : *prime* & *p* ^ *a* = (*n* + *1*) & *0* < *a*)
apply (*clarsimp*)
apply (*subgoal-tac n* + *1* = *p* ^ *a*)
apply (*erule ssubst*)**back**
apply (*subst Lambda-eq*)
apply *assumption*+
apply (*case-tac real p* < *exp 1*)
apply (*rule order-le-less-trans*)
apply (*rule abs-triangle-ineq4*)
apply *simp*
apply (*subst abs-nonneg*)
apply (*rule ln-ge-zero*)
apply (*subgoal-tac real* (*1*::*nat*) <= *real p*)
apply *simp*
apply (*rule le-imp-real-of-nat-le*)
apply (*erule primes-always-ge-1*)
apply (*rule order-less-le-trans*)
apply (*subgoal-tac ln*(*real p*) + *1* < *ln* (*exp 1*) + *1*)
apply *assumption*
apply (*rule add-strict-right-mono*)
apply (*subst ln-less-cancel-iff*)
apply (*subgoal-tac 1* <= *p*)
apply *arith*

apply (*erule primes-always-ge-1*)
apply *force*
apply *assumption*
apply *simp*
apply (*subgoal-tac ln* (*exp 2*) <= *ln* (*real n*))
apply *simp*
apply (*subst ln-le-cancel-iff*)
apply *force*
apply (*rule order-le-less-trans*)
prefer *2*
apply (*rule c*)
apply *force*
apply (*rule order-less-imp-le*)
apply (*rule c*)
apply (*subst abs-nonneg*)
apply *simp*
apply (*subgoal-tac ln* (*exp 1*) <= *ln* (*real p*))
apply *simp*
apply (*subst ln-le-cancel-iff*)
apply *force*
apply (*frule primes-always-ge-1*)
apply *arith*
apply *arith*
apply (*simp add*: *compare-rls*)
apply (*rule order-le-less-trans*)
apply (*subgoal-tac ln* (*real p*) <= *ln*(*real* (*n* + *1*)))
apply *assumption*
apply (*subst ln-le-cancel-iff*)
apply (*frule primes-always-ge-1*)
apply *arith*
apply *arith*
apply (*rule le-imp-real-of-nat-le*)
apply (*erule subst*)
apply (*subgoal-tac p^1* <= *p^a*)
apply *simp*
apply (*rule power-increasing*)
apply *arith*
apply (*erule primes-always-ge-1*)
apply *simp*
apply (*subgoal-tac ln* (*real n* + *1*) − *ln* (*real n*) < *1*)
apply *simp*
apply (*subst ln-div* [*THEN sym*])
apply *force*
apply (*rule order-le-less-trans*)
prefer *2*

apply (*rule c*)
apply *force*
apply (*subgoal-tac* (*real n* + *1*) / *real n* = *1* + *1* / *real n*)
apply (*erule ssubst*)**back**
apply (*rule order-le-less-trans*)
apply (*rule ln-add-one-self-le-self*)
apply *force*
apply (*subst pos-divide-less-eq*)
apply (*rule order-le-less-trans*)
prefer *2*
apply (*rule c*, *force*)
apply *simp*
apply (*rule order-le-less-trans*)
prefer *2*
apply (*rule c*)
apply (*subgoal-tac exp 0* <= *exp 2*)
apply *simp*
apply (*subst exp-le-cancel-iff*)
apply *force*
apply (*simp add*: *add-divide-distrib*)
apply (*subgoal-tac 0* < *real n*)
apply *arith*
apply (*rule order-le-less-trans*)
prefer *2*
apply (*rule c*)
apply *force*
apply *simp*
apply (*subst Lambda-eq2*)
apply *assumption*
apply *simp*
apply (*subgoal-tac ln*(*exp 1*) < *ln*(*real n*))
apply *simp*
apply (*subst ln-less-cancel-iff*)
apply *force*
apply (*rule order-le-less-trans*)
prefer *2*
apply (*rule c*)
apply *force*
apply (*rule order-le-less-trans*)
prefer *2*
apply (*rule c*)
apply *simp*
done
finally have *abs* (*R*(*real n*)) < *ln* (*real n*)**.**
then have *abs* (*R*(*real n*)) / (*real n*) < *ln* (*real n*) / (*real n*)

apply (*rule real-div-pos-less-mono*)
apply (*rule order-le-less-trans*)
prefer *2*
apply (*rule c*)
apply *force*
done
also have ... <= *ln* (*real* (*natfloor x*)) / (*real* (*natfloor x*))
apply (*rule ln-x-over-x-mono*)
apply (*rule order-less-imp-le*)
apply (*rule ge-natfloor-plus-one-imp-gt*)
apply (*subst natfloor-add* [*THEN sym*])
apply *force*
apply *simp*
apply (*rule natfloor-mono*)
apply (*rule order-trans*)
prefer *2*
apply (*rule a*)
apply *simp*
apply *simp*
apply (*rule order-less-imp-le*)
apply (*simp add*: *prems*)
done
also have ... <= ((*real* (*natfloor x*) *powr* (*1 / 2*)) / (*1 / 2*)) /
(*real* (*natfloor x*))
apply (*rule divide-right-mono*)
apply (*rule ln-powr-bound*)
apply (*rule order-less-imp-le*)
apply (*rule ge-natfloor-plus-one-imp-gt*)
apply (*subst natfloor-add* [*THEN sym*])
apply *force*
apply *simp*
apply (*rule natfloor-mono*)
apply (*rule order-trans*)
prefer *2*
apply (*rule a*)
apply *simp*
apply (*subgoal-tac exp 0 <= exp 2*)
apply *simp*
apply *force*
apply *force*
apply *simp*
done
also have ... = *2* ∗ ((*real* (*natfloor x*) *powr* (*1 / 2*)) /
real (*natfloor x*) *powr 1*)
apply (*simp add*: *mult-ac*)

apply (*rule sym*)
apply (*subst powr-one-gt-zero-iff*)
apply *simp*
apply (*rule order-less-le-trans*)
prefer *2*
apply (*rule real-le-natfloor*)
apply (*subgoal-tac real 1 <= x*)
apply *assumption*
apply *simp*
apply (*rule order-trans*)
prefer *2*
apply (*rule a*)
apply *simp*
apply *simp*
done
also have *... = 2 * (real (natfloor x) powr (1 / 2 - 1))*
apply (*rule arg-cong*)**back**
apply (*rule powr-divide2*)
done
also have *(1 / 2 - (1::real)) = - (1 / 2)*
by *simp*
also have *2 * (real (natfloor x) powr - (1 / 2)) =*
2 / (real (natfloor x) powr (1 / 2))
apply (*subst powr-minus-divide*)
apply *simp*
done
also have *... <= 2 / (exp (4 / eps) powr (1 / 2))*
apply (*rule divide-left-mono*)
apply (*rule power-mono2*)
apply *force*
apply *force*
apply (*rule d*)
apply *force*
apply (*rule mult-pos*)
apply *force*
apply *force*
done
also have *... = 2 / (exp (2 / eps))*
by (*simp add*: *powr-def*)
also have *... < 2 / (2 / eps)*
apply (*rule divide-strict-left-mono*)
apply (*rule order-less-le-trans*)
prefer *2*
apply (*rule exp-ge-add-one-self*)
apply (*rule real-ge-zero-div-gt-zero*)

apply *force*
apply (*rule prems*)
apply *arith*
apply *auto*
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)
apply *auto*
apply (*rule prems*)
done
also have *... = eps*
by *simp*
finally show *?thesis*
apply (*rule-tac x = n* **in** *exI*)
apply (*auto simp add*: *prems*)
apply (*subst abs-div* [*THEN sym*])
apply (*subgoal-tac 0 < real n*)
apply *arith*
apply (*rule order-le-less-trans*)
prefer *2*
apply (*rule c*)
apply *force*
apply *simp*
done
next assume *no-sign-change*:~ (*EX n. natfloor x < n &*
*n <= natfloor (K * x) &*
~((*0 <= R(real n)*) = (*0 <= R(real (n + 1)*))))
then have
(*ALL n. natfloor x < n & n <= natfloor (K * x) --> 0 <= R(real n)*) |
(*ALL n. natfloor x < n & n <= natfloor (K * x) --> R(real n) < 0*)
apply (*intro PNT1-aux10*)
apply (*rule natfloor-mono*)
apply (*subgoal-tac 1 * x <= K * x*)
apply *simp*
apply (*rule mult-right-mono*)
apply (*rule order-less-imp-le*)
apply (*insert C0*)
apply (*drule-tac x = K* **in** *spec*)
apply (*drule-tac x = 1* **in** *spec*)
apply (*force simp add*: *prems*)
apply (*rule order-trans*)
prefer *2*
apply (*rule a*)
apply (*rule nonneg-plus-nonneg*)
apply *force*
apply *force*

apply *force*
done
then have $(\sum\ n \mid natfloor\ x < n\ \&\ n <= natfloor\ (K * x).$
$abs\ (R(real\ n))\ /\ ((real\ n) * (real\ (n + 1)))) =$
$abs(\sum\ n \mid natfloor\ x < n\ \&\ n <= natfloor\ (K * x).$
$R(real\ n)\ /\ ((real\ n) * (real\ (n + 1))))$
apply (*elim disjE*)
apply (*subst abs-nonneg*)
apply (*rule setsum-nonneg'*)
apply *clarify*
apply (*rule real-ge-zero-div-gt-zero*)
apply *force*
apply (*rule mult-pos*)
apply (*insert a*)
apply (*subgoal-tac 1 < exp 2 + 1*)
apply *arith*
apply *force*
apply (*insert a*)
apply (*subgoal-tac 1 < exp 2 + 1*)
apply *arith*
apply *force*
apply (*rule setsum-cong2*)
apply (*subst abs-nonneg*)
apply *force*
apply (*rule refl*)
apply (*subst abs-nonpos*)
apply (*rule setsum-nonpos'*)
apply *clarify*
apply (*rule order-less-imp-le*)
apply (*subst pos-divide-less-eq*)
apply (*rule mult-pos*)
apply (*insert a*)
apply (*subgoal-tac 1 < exp 2 + 1*)
apply *arith*
apply *force*
apply (*insert a*)
apply (*subgoal-tac 1 < exp 2 + 1*)
apply *arith*
apply *force*
apply *force*
apply (*subst setsum-negf' [THEN sym]*)
apply (*rule setsum-cong2*)
apply (*subst abs-nonpos*)
apply *force*
apply *simp*

```
  done
also have ... < C
  apply (insert C)
  apply (drule spec)+
  apply (drule mp)
  prefer 2
  apply (drule mp)
  prefer 2
  apply assumption
  apply (rule natfloor-mono)
  apply (subgoal-tac 1 * x < K * x)
  apply force
  apply (rule mult-strict-right-mono)
  apply (insert C0)
  apply (drule-tac x = K in spec)
  apply (drule-tac x = 1 in spec)
  apply (force simp add: prems)
  apply (insert a)
  apply (subgoal-tac 0 < exp 2)
  apply arith
  apply force
  apply (rule real-le-natfloor)
  apply (insert a)
  apply (subgoal-tac 0 < exp 2)
  apply arith
  apply force
  done
finally have e: (∑ n | natfloor x < n & n <= natfloor (K * x).
    abs (R (real n)) / (real n * real (n + 1))) < C.
have ~ (ALL n. natfloor x < n & n <= natfloor (K * x) -->
    C / (∑ n | natfloor x < n & n <= natfloor (K * x).
    1 / (real (n + 1))) <=  abs (R (real n)) / (real n))
proof
  assume f: ALL n. natfloor x < n & n <= natfloor (K * x) -->
    C / (∑ n | natfloor x < n & n <= natfloor (K * x).
    1 / real (n + 1)) <= abs (R (real n)) / real n
  have (∑ n | natfloor x < n & n <= natfloor (K * x).
      abs (R (real n)) / (real n * real (n + 1))) =
    (∑ n | natfloor x < n & n <= natfloor (K * x).
      abs (R (real n)) / (real n)  * (1 / real (n + 1)))
    apply (rule setsum-cong2)
    apply simp
    done
  also have (∑ n | natfloor x < n & n <= natfloor (K * x).
    (C / (∑ n | natfloor x < n & n <= natfloor (K * x).
```

*1 / real (n + 1))) * (1 / real (n + 1))) <= ...*
(**is** *?LHS <= ?RHS*)
apply (*rule setsum-le-cong*)
apply (*rule mult-right-mono*)
apply (*insert f*)
apply *auto*
done
also have *?LHS = C*
apply (*subst setsum-const-times*)
apply *simp*
apply (*subgoal-tac 0 < (∑ n | natfloor x < n &*
*n <= natfloor (K * x). 1 / (real n + 1))*)
apply *force*
apply (*subgoal-tac (∑ n | natfloor x < n &*
*n <= natfloor (K * x). 1 / (real n + 1)) =*
(∑ n : {natfloor x + 1}. 1 / (real n + 1)) +
(∑ n | natfloor x + 1 < n &
*n <= natfloor (K * x). 1 / (real n + 1))*)
apply (*erule ssubst*)
apply *simp*
apply (*rule order-less-le-trans*)
prefer *2*
apply (*rule setsum-nonneg'*)
apply *force*
apply *force*
apply (*subst setsum-Un-disjoint* [*THEN sym*])
apply *force*
apply (*rule bounded-nat-set-is-finite*)
apply *clarify*
apply (*subgoal-tac i < natfloor (K * x) + 1*)
apply *assumption*
apply *arith*
apply *force*
apply (*rule setsum-cong*)
apply *auto*
apply (*subst natfloor-add* [*THEN sym*])
apply (*rule order-trans*)
prefer *2*
apply (*rule a*)
apply (*subgoal-tac 0 < exp 2*)
apply *arith*
apply *force*
apply *simp*
apply (*rule natfloor-mono*)
apply (*rule order-trans*)

```
    apply (subgoal-tac x + 1 <= 2 * x)
    apply assumption
    apply (insert a)
    apply (subgoal-tac 0 < exp 2)
    apply arith
    apply force
    apply (rule mult-right-mono)
    apply (rule order-less-imp-le)
    apply (insert C0)
    apply (drule-tac x = K in spec)
    apply (drule-tac x = 1 in spec)
    apply (force simp add: prems)
    apply (insert a)
    apply (subgoal-tac 0 < exp 2)
    apply arith
    apply force
    done
  finally show False
    by (insert e, auto)
 qed
then have EX n. natfloor x < n & n <= natfloor (K * x) &
     abs (R (real n)) / real n < C /
   (∑ n | natfloor x < n & n <= natfloor (K * x).
      1 / real (n + 1))
   by auto
then show EX n. natfloor x < n & n <= natfloor (K * x) &
     abs (R (real n) / real n) < eps
 apply auto
 apply (rule-tac x = n in exI)
 apply auto
 apply (subst abs-div [THEN sym])
 apply force
 apply (subst abs-nonneg)
 apply force
 apply (erule order-less-le-trans)
 apply (insert C0)
 apply (drule-tac x = K in spec)
 apply (drule-tac x = x in spec)
 apply (rule order-less-imp-le)
 apply (subgoal-tac 1 <= x)
 apply (force simp add: prems)
 apply (insert a)
 apply (subgoal-tac 0 < exp 2)
 apply arith
 apply force
```

```
    done
  qed
 qed
qed qed qed

lemma PNT1a: EX C0. ALL eps. 0 < eps --> eps < 1 --> (EX x0. ALL K
x.
    exp (C0 / eps) < K --> x0 < x --> (EX n::nat.
      x < real n & real n <= K * x &
        abs(R(real n) / (real n)) < eps))
 apply (insert PNT1)
 apply clarify
 apply (rule-tac x = C0 in exI)
 apply clarify
 apply (drule-tac x = eps in spec)
 apply clarify
 apply (rule-tac x = max x0 0 in exI)
 apply clarify
 apply (drule-tac x = K in spec)
 apply (drule-tac x = x in spec)
 apply clarify
 apply (subgoal-tac x0 < x)
 apply clarify
 apply (rule-tac x = n in exI)
 apply (rule conjI)
 apply (rule ge-natfloor-plus-one-imp-gt)
 apply arith
 apply (rule conjI)
 apply (rule nat-le-natfloor)
 apply (rule nonneg-times-nonneg)
 apply (rule order-less-imp-le)
 apply (rule order-le-less-trans)
 prefer 2
 apply assumption
 apply force
 apply arith
 apply assumption
 apply assumption
 apply simp
done

lemma PNT2-aux1: EX C. ALL x. 1 <= x -->
   abs (psi (natfloor x) * ln x + (SUM d = 1..natfloor x.
     Lambda d * psi ((natfloor x) div d)) -
   2 * x * ln x) <= C * x
```

```
  apply (insert Selberg4a)
  apply (drule set-plus-imp-minus)
  apply (simp add: func-plus func-diff)
  apply (unfold bigo-def)
  apply clarsimp
  apply (rule-tac x = c in exI)
  apply clarify
  apply (drule-tac x = x - 1 in spec)
  apply (subgoal-tac natfloor (abs(x - 1)) + 1 = natfloor (abs(x - 1) + 1))
  apply simp
  apply (subst natfloor-add [THEN sym])
  apply force
  apply simp
done

lemma PNT2-aux2: EX C. ALL x y. 1 <= y --> y <= x -->
    (psi (natfloor x) * ln x - psi (natfloor y) * ln y) <=
    (2 * x * ln x - 2 * y * ln y) + C * x
proof -
  from PNT2-aux1 obtain C where ALL x. 1 <= x -->
    abs (psi (natfloor x) * ln x + (∑ d = 1..natfloor x.
      Lambda d * psi ((natfloor x) div d)) -
     2 * x * ln x) <= C * x..
  show ?thesis
  proof ((rule exI)+, clarify)
    fix x fix y
    assume (1::real) <= y
    assume y <= x
    show (psi (natfloor x) * ln x - psi (natfloor y) * ln y) <=
      (2 * x * ln x - 2 * y * ln y) + (C + C) * x
    proof -
      have a: 0 <= (∑ d = 1..natfloor x.
          Lambda d * psi ((natfloor x) div d)) -
        (∑ d = 1..natfloor y.
          Lambda d * psi ((natfloor y) div d))
          (is 0 <= ?term1)
        apply (subgoal-tac
          (∑ d = 1..natfloor x. Lambda d * psi (natfloor x div d)) =
          (∑ d = 1..natfloor y.
            Lambda d * psi (natfloor x div d)) +
          (∑ d = natfloor y + 1..natfloor x.
            Lambda d * psi (natfloor x div d)))
        apply (erule ssubst)
        apply (subgoal-tac (∑ d = 1..natfloor y.
            Lambda d * psi (natfloor y div d)) <=
```

```
      (∑ d = 1..natfloor y.
     Lambda d * psi (natfloor x div d)))
  apply (subgoal-tac 0 <= (∑ d = natfloor y + 1..natfloor x.
     Lambda d * psi (natfloor x div d)))
  apply arith
  apply (rule setsum-nonneg')
  apply clarsimp
  apply (rule nonneg-times-nonneg)
  apply (rule Lambda-ge-zero)
  apply (rule psi-ge-zero)
  apply (rule setsum-le-cong)
  apply (rule mult-left-mono)
  apply (rule psi-mono)
  apply (rule div-le-mono)
  apply (rule natfloor-mono)
  apply (rule prems)
  apply (rule Lambda-ge-zero)
  apply (subst setsum-Un-disjoint [THEN sym])
  apply force
  apply (rule bounded-nat-set-is-finite)
  apply (rule ballI)
  apply (subgoal-tac i < natfloor x + 1)
  apply assumption
  apply force
  apply force
  apply (rule setsum-cong)
  apply auto
  apply (erule order-trans)
  apply (rule natfloor-mono)
  apply (rule prems)
  done
have abs((psi (natfloor x) * ln x - psi (natfloor y) * ln y) + ?term1) -
    abs(2 * x * ln x - 2 * y * ln y) <=
  abs((psi (natfloor x) * ln x - psi (natfloor y) * ln y) + ?term1 -
    (2 * x * ln x - 2 * y * ln y))
  by (rule abs-triangle-ineq2)
also have ... = abs((psi(natfloor x) * ln x +
    (∑ d = 1..natfloor x. Lambda d * psi ((natfloor x) div d))
     - 2 * x * ln x) -
   (psi(natfloor y) * ln y +
    (∑ d = 1..natfloor y. Lambda d * psi ((natfloor y) div d))
     - 2 * y * ln y))
  apply (rule arg-cong) back
  apply (simp add: compare-rls)
  done
```

also have ... <= $abs(psi(natfloor\ x) * ln\ x +$
$(\sum\ d = 1..natfloor\ x.\ Lambda\ d * psi\ ((natfloor\ x)\ div\ d))$
$- 2 * x * ln\ x) +$
$abs(psi(natfloor\ y) * ln\ y +$
$(\sum\ d = 1..natfloor\ y.\ Lambda\ d * psi\ ((natfloor\ y)\ div\ d))$
$- 2 * y * ln\ y)$
by (*rule abs-triangle-ineq4*)
also have ... <= $C * x + C * y$
apply (*rule add-mono*)
apply (*insert prems*)
apply *auto*
done
also have ... <= $C * x + C * x$
apply *simp*
apply (*rule mult-left-mono*)
apply (*rule prems*)
apply (*insert prems*)
apply (*drule-tac x = 1* **in** *spec*)
apply *simp*
apply (*rule order-trans*)
prefer *2*
apply *assumption*
apply (*rule nonneg-times-nonneg*)
apply *auto*
done
also have ... = $(C + C) * x$
by (*simp add*: *ring-eq-simps*)
also have $abs\ (psi\ (natfloor\ x) * ln\ x - psi\ (natfloor\ y) * ln\ y +$
$((\sum\ d = 1..natfloor\ x.\ Lambda\ d * psi\ (natfloor\ x\ div\ d)) -$
$(\sum\ d = 1..natfloor\ y.\ Lambda\ d * psi\ (natfloor\ y\ div\ d)))) =$
$psi\ (natfloor\ x) * ln\ x - psi\ (natfloor\ y) * ln\ y +$
$((\sum\ d = 1..natfloor\ x.\ Lambda\ d * psi\ (natfloor\ x\ div\ d)) -$
$(\sum\ d = 1..natfloor\ y.\ Lambda\ d * psi\ (natfloor\ y\ div\ d)))$
apply (*rule abs-nonneg*)
apply (*rule nonneg-plus-nonneg*)
apply *simp*
apply (*rule mult-mono*)
apply (*rule psi-mono*)
apply (*rule natfloor-mono*)
apply (*rule prems*)
apply (*subst ln-le-cancel-iff*)
apply (*insert prems*, *arith*)
apply *arith*
apply *assumption*
apply (*rule psi-ge-zero*)

```
      apply force
      apply (rule a)
      done
    also have abs (2 * x * ln x - 2 * y * ln y) =
      2 * x * ln x - 2 * y * ln y
      apply (rule abs-nonneg)
      apply simp
      apply (rule mult-mono)
      apply (rule prems)
      apply (subst ln-le-cancel-iff)
      apply (insert prems)
      apply auto
      done
    finally show ?thesis
      by (insert a, arith)
  qed
 qed
qed

lemma PNT2-aux3: EX E. ALL x y. 1 <= y --> y < x --> x <= D * y
-->
  (psi(natfloor x) - psi(natfloor y)) <= 2 * (x - y) + E * (x / ln x)
proof -
 from PNT2-aux2 obtain C where C: ALL x y. 1 <= y --> y <= x -->
  (psi (natfloor x) * ln x - psi (natfloor y) * ln y) <=
  (2 * x * ln x - 2 * y * ln y) + C * x..
 show ?thesis
 proof ((rule exI), clarify)
  fix x::real
  fix y::real
  assume 1 <= y
  assume y < x
  assume x <= D * y
  have (psi (natfloor x) * ln x - psi (natfloor y) * ln y) <=
    (2 * x * ln x - 2 * y * ln y) + C * x
   by (insert C prems, auto)
  also have psi(natfloor x) * ln x - psi(natfloor y) * ln y =
    ln x * (psi(natfloor x) - psi(natfloor y)) +
    psi (natfloor y) * (ln x - ln y)
   by (simp add: ring-eq-simps)
  also have (2 * x * ln x - 2 * y * ln y) =
    2 * (x - y) * ln x + 2 * y * (ln x - ln y)
   by (simp add: ring-eq-simps)
  also have ln x - ln y = ln (x / y)
   apply (rule ln-div [THEN sym])
```

apply (*insert prems*)
apply *arith*+
done
finally have *ln x* ∗ (*psi* (*natfloor x*) − *psi* (*natfloor y*))
<= *2* ∗ (*x* − *y*) ∗ *ln x* + *2* ∗ *y* ∗ *ln* (*x* / *y*) + *C* ∗ *x*
apply (*subgoal-tac 0* <= *psi*(*natfloor y*) ∗ *ln*(*x* / *y*))
apply *arith*
apply (*rule nonneg-times-nonneg*)
apply (*rule psi-ge-zero*)
apply (*rule ln-ge-zero*)
apply (*subst pos-le-divide-eq*)
apply (*insert prems*, *arith*)
apply *simp*
done
also have ... <= *2* ∗ (*x* − *y*) ∗ *ln x* + *2* ∗ *x* ∗ *ln D* + *C* ∗ *x*
apply (*rule add-right-mono*)
apply (*rule add-left-mono*)
apply (*rule mult-mono*)
apply (*rule mult-left-mono*)
apply (*rule order-less-imp-le*, *rule prems*)
apply *force*
apply (*subst ln-le-cancel-iff*)
apply (*rule pos-div-pos*)
apply (*insert prems*, *arith*)
apply *arith*
prefer *2*
apply (*subst pos-divide-le-eq*)
apply *arith*
apply *assumption*
apply (*rule order-less-le-trans*)
apply (*subgoal-tac 0* < *x* / *y*)
apply *assumption*
apply (*rule pos-div-pos*)
apply *arith*
apply *arith*
apply (*subst pos-divide-le-eq*)
apply *arith*
apply *assumption*
apply *arith*
apply (*rule ln-ge-zero*)
apply (*subst pos-le-divide-eq*)
apply *arith*
apply *simp*
done
also have ... = *2* ∗ (*x* − *y*) ∗ *ln x* + (*2* ∗ *ln D* + *C*) ∗ *x*

```
    by (simp add: ring-eq-simps)
  finally have ln x * (psi (natfloor x) - psi (natfloor y))
      <= 2 * (x - y) * ln x + (2 * ln D + C) * x (is ?LHS <= ?RHS).
  then have ?LHS / ln x <= ?RHS / ln x
    apply (rule divide-right-mono)
    apply (rule ln-ge-zero)
    apply (insert prems, arith)
    done
  also have ?LHS / ln x = (psi (natfloor x) - psi (natfloor y))
    apply (subgoal-tac 0 < ln x)
    apply simp
    apply (rule ln-gt-zero)
    apply (insert prems, arith)
    done
  also have ?RHS / ln x = 2 * (x - y) + (2 * ln D + C) * (x / ln x)
    apply (subgoal-tac 0 < ln x)
    apply (subst add-divide-distrib)
    apply simp
    apply (rule ln-gt-zero)
    apply (insert prems, arith)
    done
  finally show psi (natfloor x) - psi (natfloor y) <= 2 * (x - y) +
      (2 * ln D + C) * (x / ln x).
 qed
qed

lemma PNT2-aux4: EX C. ALL x. abs (R(x) / x) < C
 apply (insert error0)
 apply (simp only: bigo-alt-def)
 apply clarsimp
 apply (rule-tac x = c + 1 in exI)
 apply (rule allI)
 apply (drule-tac x = x in spec)
 apply (case-tac x = 0)
 apply simp
 apply (subst abs-divide)
 apply (subst pos-divide-less-eq)
 apply arith
 apply (simp add: ring-eq-simps)
 apply arith
done

lemma PNT2-aux5: 0 < eps ==> EX x. ALL y. (x < y --> 1 / ln y < eps)
 apply (rule-tac x = max 1 (exp (1 / eps)) in exI)
 apply clarify
```

```
  apply (subst pos-divide-less-eq)
  apply (rule ln-gt-zero)
  apply force
  apply (subst mult-commute)
  apply (subst pos-divide-less-eq [THEN sym])
  apply assumption
  apply (subgoal-tac 1 / eps = ln (exp (1 / eps)))
  apply (erule ssubst)
  apply (subst ln-less-cancel-iff)
  apply auto
done

lemma PNT2: EX lambda C1. 0 < lambda & lambda < 1 &
    (ALL eps. 0 < eps --> eps < 1 --> (EX x1. ALL K x.
      exp (C1 / eps) < K --> x1 < x --> (EX xstar.
        x < xstar & (1 + lambda * eps) * xstar < K * x & (ALL u.
          xstar <= u & u <= (1 + lambda * eps) * xstar -->
            abs(R u) < eps * u)))) (is EX lambda C1. ?P lambda C1)
proof -
  from PNT2-aux4 obtain Cstar where Cstar: ALL x. abs (R(x) / x) < Cstar..
  have Cstar0: 0 < Cstar
    apply (insert Cstar)
    apply (drule-tac x = 0 in spec)
    apply simp
    done
  let ?lambda = 1 / (3 * (Cstar + 3))
  have lambda0: 0 < ?lambda
    apply (rule pos-div-pos)
    apply force
    apply (rule mult-pos)
    apply force
    apply (rule pos-plus-pos)
    apply (rule Cstar0)
    apply force
    done
  have lambda1: ?lambda < 1
    apply (subst pos-divide-less-eq)
    apply (rule mult-pos)
    apply force
    apply (insert Cstar0, arith)
    apply simp
    done
  from PNT1a obtain C0 where C0temp: ALL eps. 0 < eps -->
    eps < 1 --> (EX x0. ALL K x. exp (C0 / eps) < K -->
      x0 < x --> (EX n::nat. x < real n &
```

```
    real n <= K * x & abs (R (real n) / real n) < eps))..
let ?C1 = 3 * C0 + ln 2
show ?thesis
proof (rule exI)+
show ?P ?lambda ?C1
  apply (rule conjI)
  apply (rule lambda0)
  apply (rule conjI)
  apply (rule lambda1)
  apply clarify
proof -
  fix eps::real
  assume eps0: 0 < eps
  assume eps1: eps < 1
  have EX x0. ALL K x.
    exp (C0 / (eps / 3)) < K --> x0 < x --> (EX (n::nat).
      x < real n & real n <= K * x &
        abs(R(real n) / (real n)) < eps / 3)
    apply (insert eps0 eps1 C0temp)
    apply (drule-tac x = eps / 3 in spec)
    apply (drule mp)
    apply arith
    apply (drule mp)
    apply arith
    apply assumption
    done
  then obtain x0 where C0x0: ALL K x.
    exp (C0 / (eps / 3)) < K --> x0 < x -->
      (EX (n::nat). x < real n & real n <= K * x &
        abs(R(real n) / (real n)) < eps / 3) by auto
  have C1: !!K. exp (?C1 / eps) < K ==> 2 * exp (C0 / (eps / 3)) < K
    apply (rule order-le-less-trans)
    prefer 2
    apply assumption
    apply (subgoal-tac (3 * C0 + ln 2) / eps =
       ln 2 / eps + (3 * C0 / eps))
    apply (erule ssubst)
    apply (subst exp-add)
    apply simp
    apply (rule mult-mono)
    apply (subgoal-tac 2 = exp (ln 2))
    apply (erule ssubst)
    apply (subst exp-le-cancel-iff)
    apply simp
    apply (subst pos-le-divide-eq)
```

apply (*rule eps0*)
apply (*subgoal-tac ln 2 * eps <= ln 2 * 1*)
apply *simp*
apply (*rule mult-left-mono*)
apply (*rule order-less-imp-le*, *rule eps1*)
apply *force*
apply *simp*
apply (*simp add*: *mult-ac*)
apply *force*
apply *force*
apply (*simp add*: *ring-eq-simps add-divide-distrib*)
done
from *PNT2-aux3* **obtain** *E* **where** *E*: *ALL x y. 1 <= y --> y < x -->*
*x <= 2 * y --> (psi(natfloor x) - psi(natfloor y)) <=*
*2 * (x - y) + E * (x / ln x)* **..**
let *?E2 = max E 1*
have *E2a*: *0 < ?E2*
apply (*subgoal-tac 1 <= ?E2*)
apply *arith*
apply (*rule le-maxI2*)
done
have *E2b*: *ALL x y. 1 <= y --> y < x -->*
*x <= 2 * y --> (psi(natfloor x) - psi(natfloor y)) <=*
*2 * (x - y) + ?E2 * (x / ln x)*
apply (*clarify*)
apply (*insert E*)
apply (*drule-tac x = x* **in** *spec*)
apply (*drule-tac x = y* **in** *spec*)
apply *clarify*
apply (*subgoal-tac E * (x / ln x) <= ?E2 * (x / ln x)*)
apply *arith*
apply (*rule mult-right-mono*)
apply (*rule le-maxI1*)
apply (*rule order-less-imp-le*)
apply (*rule pos-div-pos*)
apply *arith*
apply (*rule ln-gt-zero*)
apply *arith*
done
have *EX x. ALL y. (x < y --> 1 / ln y <*
*eps / (3 * ?E2 * (1 + ?lambda * eps)))*
apply (*rule PNT2-aux5*)
apply (*rule pos-div-pos*)
apply (*rule eps0*)
apply (*rule mult-pos*)

apply (*rule mult-pos*)
apply *force*
apply (*rule E2a*)
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule mult-pos*)
apply (*rule lambda0*)
apply (*rule eps0*)
done
then obtain *xtemp* **where** *xtemp*:
ALL y. (*xtemp* < *y* −−> *1* / *ln y* <
eps / (*3* ∗ *?E2* ∗ (*1* + *?lambda* ∗ *eps*)))**..**
let *?x1* = *max* (*max xtemp x0*) *1*
have *x1a*: *1* <= *?x1* **by** (*rule le-maxI2*)
have *x1b*: *x0* <= *?x1*
apply (*rule order-trans*)
apply (*rule le-maxI2*)
apply (*rule le-maxI1*)
done
have *x1c*: *xtemp* <= *?x1*
apply (*rule order-trans*)
apply (*rule le-maxI1*)
apply (*rule le-maxI1*)
done
show *EX x1*. *ALL K x*. *exp* (*?C1* / *eps*) < *K* −−> *x1* < *x* −−>
(*EX xstar*. *x* < *xstar* &
(*1* + *1* / (*3* ∗ (*Cstar* + *3*)) ∗ *eps*) ∗ *xstar* < *K* ∗ *x* &
(*ALL u*. *xstar* <= *u* &
u <= (*1* + *1* / (*3* ∗ (*Cstar* + *3*)) ∗ *eps*) ∗ *xstar* −−>
abs (*R u*) < *eps* ∗ *u*)) (**is** *EX x1*. *?P x1*)
proof ((*rule exI*)+, *clarify*)
fix *K*
fix *x*
assume *K*: *exp* (*?C1* / *eps*) < *K*
assume *x*: *?x1* < *x*
have *xa*: *1* < *x*
apply (*rule order-le-less-trans*)
prefer *2*
apply (*rule x*)
apply (*rule x1a*)
done
have *xb*: *x0* < *x*
apply (*rule order-le-less-trans*)
prefer *2*
apply (*rule x*)

```
  apply (rule x1b)
  done
have xc: xtemp < x
  apply (rule order-le-less-trans)
  prefer 2
  apply (rule x)
  apply (rule x1c)
  done
have EX (n::nat). x < real n & real n <= (K / 2) * x &
    abs(R(real n) / (real n)) < eps / 3
  apply (insert C0x0)
  apply (drule-tac x = K / 2 in spec)
  apply (drule-tac x = x in spec)
  apply (drule mp)
  apply (subst pos-less-divide-eq)
  apply force
  apply (subst mult-commute)
  apply (rule C1)
  apply (rule K)
  apply (drule mp)
  apply (rule xb)
  apply assumption
  done
then obtain n::nat where n: x < real n &
      real n <= K / 2 * x & abs (R (real n) / real n) < eps / 3..
from n have n1: x < real n..
from n have n2: real n <= (K / 2) * x by auto
from n have n3: abs (R (real n) / real n) < eps / 3 by auto
show EX xstar. x < xstar &
  (1 + 1 / (3 * (Cstar + 3)) * eps) * xstar < K * x &
    (ALL u. xstar <= u &
      u <= (1 + 1 / (3 * (Cstar + 3)) * eps) * xstar -->
        abs (R u) < eps * u) (is EX xstar. ?P xstar)
proof
let ?xstar = real n
show ?P ?xstar
  apply (rule conjI)
  apply (rule n1)
  apply (rule conjI)
  apply (rule order-le-less-trans)
  apply (subgoal-tac (1 + ?lambda * eps) * real n <=
    (1 + ?lambda * eps) * ((K / 2) * x))
  apply assumption
  apply (rule mult-left-mono)
  apply (rule n2)
```

```
    apply (rule nonneg-plus-nonneg)
    apply force
    apply (rule nonneg-times-nonneg)
    apply (rule order-less-imp-le)
    apply (rule pos-div-pos)
    apply force
    apply (rule mult-pos)
    apply force
    apply (rule pos-plus-pos)
    apply (rule Cstar0)
    apply force
    apply (rule order-less-imp-le)
    apply (rule eps0)
    apply (rule order-less-le-trans)
    apply (subgoal-tac (1 + ?lambda * eps) * (K / 2 * x) <
       2 * (K / 2 * x))
    apply assumption
    apply (rule mult-strict-right-mono)
    apply simp
    apply (subst pos-divide-less-eq)
    apply (insert Cstar0, arith)
    apply (insert eps1, simp)
    apply (rule mult-pos)
    apply (subgoal-tac 0 < K)
    apply arith
    apply (rule order-le-less-trans)
    prefer 2
    apply (rule prems)
    apply force
    apply (insert xa, arith)
    apply simp
    apply clarify
    proof -
      fix u
      assume xstar1: ?xstar <= u
      assume xstar2: u <= (1 + ?lambda * eps) * ?xstar
      show abs (R u) < eps * u
      proof -


let ?n = ?xstar
have abs (R(u) / u - R(?n) / ?n) = abs(R(u) * (1 / u - 1 / ?n) +
   (R(u) - R(?n)) / ?n)
  apply (simp add: ring-eq-simps diff-divide-distrib)
  done
```

also have ... <= *abs* (*R*(*u*) * (*1* / *u* − *1* / *?n*)) + *abs* ((*R*(*u*) − *R*(*?n*)) / *?n*)
by (*rule abs-triangle-ineq*)
also have *abs* (*R*(*u*) * (*1* / *u* − *1* / *?n*)) =
abs (*R*(*u*) / *u*) * *abs* (*1* − *u* / *?n*)
apply (*subst abs-mult* [*THEN sym*])
apply (*rule arg-cong*)**back**
apply (*simp add*: *ring-eq-simps*)
apply (*rule nonzero-mult-divide-cancel-left*)
apply (*insert n1 xa xstar1*)
apply *arith*
apply *arith*
done
also have *abs* (*1* − *u* / *?n*) = *u* / *?n* − *1*
apply (*subst abs-nonpos*)
apply *simp*
apply (*subst pos-le-divide-eq*)
apply (*insert n1 xa*, *arith*)
apply (*simp add*: *xstar1*)
apply (*simp add*: *ring-eq-simps*)
done
also have *abs* ((*R*(*u*) − *R*(*?n*)) / *?n*) = *1* / *?n* * *abs*(*R*(*u*) − *R*(*?n*))
apply (*subst abs-divide*)
apply *simp*
done
also have *R*(*u*) − *R*(*?n*) = *psi*(*natfloor*(*u*)) − *psi*(*natfloor*(*?n*)) −
(*u* − *?n*)
by (*simp add*: *R-def*)
finally have *abs* (*R u* / *u* − *R ?n* / *?n*) <=
abs (*R u* / *u*) * (*u* / *?n* − *1*) +
1 / *?n* * *abs* (*psi* (*natfloor u*) − *psi* (*natfloor ?n*) − (*u* − *?n*)).
also have ... <= *abs* (*R u* / *u*) * (*u* / *?n* − *1*) +
1 / *?n* * (*abs* (*psi* (*natfloor u*) − *psi* (*natfloor ?n*)) + *abs*(*u* − *?n*))
apply (*rule add-left-mono*)
apply (*rule mult-left-mono*)
apply (*rule abs-triangle-ineq4*)
apply (*subst pos-le-divide-eq*)
apply (*insert n1 xa*, *arith*)
apply *simp*
done
also have *1* / *?n* * (*abs* (*psi* (*natfloor u*) − *psi* (*natfloor ?n*)) +
abs(*u* − *?n*))
= (*psi* (*natfloor u*) − *psi*(*natfloor ?n*)) / *?n* + (*u* / *?n* − *1*)
apply (*subst abs-nonneg*)
apply (*simp add*: *xstar1*)
apply (*subst abs-nonneg*)

apply (*simp del*: *natfloor-real-id*)
apply (*rule psi-mono*)
apply (*rule natfloor-mono*)
apply (*rule prems*)
apply (*simp add*: *ring-eq-simps diff-divide-distrib*)
apply (*insert prems*, *arith*)
done
also have *abs* (*R u* / *u*) ∗ (*u* / *?n* − *1*) +
((*psi* (*natfloor u*) − *psi* (*natfloor ?n*)) / *?n* + (*u* / *?n* − *1*)) =
(*u* / *?n* − *1*) ∗ (*abs* (*R u* / *u*) + *1*) +
(*psi* (*natfloor u*) − *psi* (*natfloor ?n*)) / *?n*
by (*simp add*: *ring-eq-simps*)
finally have *abs* (*R u* / *u* − *R ?n* / *?n*) <=
(*u* / *?n* − *1*) ∗ (*abs* (*R u* / *u*) + *1*) +
(*psi* (*natfloor u*) − *psi* (*natfloor ?n*)) / *?n*.
also have ... <= (*u* / *?n* − *1*) ∗ (*abs* (*R u* / *u*) + *1*) +
(*2* ∗ (*u* − *?n*) + *?E2* ∗ (*u* / *ln u*)) / *?n*
apply (*rule add-left-mono*)
apply (*rule divide-right-mono*)
apply (*case-tac u* = *?n*)
apply *simp*
apply (*rule order-less-imp-le*)
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)
apply (*rule E2a*)
apply (*insert n1 xa*, *arith*)
apply (*rule ln-gt-zero*)
apply *arith*
apply (*insert E2b*)
apply (*drule-tac x* = *u* **in** *spec*)
apply (*drule-tac x* = *?n* **in** *spec*)
apply (*drule-tac mp*)
apply (*insert n1 xa*, *arith*)
apply (*drule-tac mp*)
apply (*insert xstar1*, *arith*)
apply (*drule-tac mp*)
apply (*rule order-trans*)
apply (*rule xstar2*)
apply (*rule mult-right-mono*)
apply *simp*
apply (*subst pos-divide-le-eq*)
apply (*insert Cstar0*, *arith*)
apply *simp*
apply (*insert eps1*, *arith*)
apply *arith*

apply *assumption*
apply *arith*
done
also have ... = (*u* / *?n* − *1*) ∗ (*abs* (*R u* / *u*) + *3*) +
(*?E2* ∗ *u* / *?n*) ∗ (*1* / *ln u*) (**is** *?temp* = *?term1* + *?term2*)
apply (*simp add*: *ring-eq-simps add-divide-distrib diff-divide-distrib*)
apply *auto*
apply (*insert n1 xa*, *arith*)
apply (*subst add-divide-distrib* [*THEN sym*])
apply *simp*
done
also have ... <= *?term1* + (*?E2* ∗ (*1* + *?lambda* ∗ *eps*) ∗ *x* / *x*) ∗
(*1* / *ln x*)
apply (*rule add-left-mono*)
apply (*rule mult-mono*)
apply (*subst times-divide-eq-right* [*THEN sym*])**back**
apply (*subst mult-assoc*)
apply (*subst times-divide-eq-right* [*THEN sym*])
apply (*rule mult-left-mono*)
apply (*rule real-fraction-le*)
apply (*insert xa*, *arith*)
apply (*insert n*, *arith*)
apply (*simp only*: *times-ac1*)
apply (*subst mult-commute*)**backbackbackback**
apply (*rule mult-left-mono*)
apply (*subst mult-commute*)
apply (*subst mult-commute*)**back**
apply (*rule xstar2*)
apply *arith*
apply (*rule order-less-imp-le*, *rule E2a*)
apply (*rule real-one-div-le-anti-mono*)
apply (*rule ln-gt-zero*)
apply *assumption*
apply (*subst ln-le-cancel-iff*)
apply *arith*
apply (*insert xstar1 n1*, *arith*)
apply *arith*
apply (*rule order-less-imp-le*)
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)+
apply (*rule E2a*)
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule mult-pos*)
apply (*rule pos-div-pos*)

apply *force*
apply *simp*
apply (*insert Cstar0*, *arith*)
apply (*rule eps0*)
apply *arith*
apply *arith*
apply (*rule order-less-imp-le*)
apply (*rule pos-div-pos*)
apply *force*
apply (*rule ln-gt-zero*)
apply *arith*
done
also have ... = *?term1* + (*?E2* ∗ (*1* + *?lambda* ∗ *eps*)) ∗ (*1* / *ln x*)
apply (*subgoal-tac x* ~= *0*)
apply *simp*
apply (*insert xa*, *arith*)
done
finally have *abs* (*R u* / *u* − *R ?n* / *?n*)
<= (*u* / *?n* − *1*) ∗ (*abs* (*R u* / *u*) + *3*) +
max E 1 ∗ (*1* + *1* / (*3* ∗ (*Cstar* + *3*)) ∗ *eps*) ∗ (*1* / *ln x*)
(**is** *?LHS* <= *?RHS*).
then have *a*: *?LHS* + *abs* (*R ?n* / *?n*) <= *?RHS* + *abs*(*R ?n* / *?n*)
by (*intro add-right-mono*)
have *abs* (*R u* / *u*) = *abs* (*R u* / *u* − *R ?n* / *?n* + *R ?n* / *?n*)
by *simp*
also have ... <= *?LHS* + *abs* (*R ?n* / *?n*)
by (*rule abs-triangle-ineq*)
also note *a*
also have *?RHS* + *abs*(*R ?n* / *?n*) < *eps* / *3* + *eps* / *3* + *eps* / *3*
apply (*rule add-le-less-mono*)
apply (*rule add-mono*)
apply (*rule order-trans*)
apply (*subgoal-tac* (*u* / *real n* − *1*) ∗ (*abs* (*R u* / *u*) + *3*) <=
(*?lambda* ∗ *eps*) ∗ (*Cstar* + *3*))
apply *assumption*
apply (*rule mult-mono*)
apply (*simp only*: *compare-rls*)
apply (*subst add-commute*)
apply (*subst pos-divide-le-eq*)
apply (*insert n1 xa*, *arith*)
apply (*rule prems*)
apply *simp*
apply (*rule order-less-imp-le*)
apply (*insert Cstar*)
apply (*erule spec*)

apply (*rule nonneg-times-nonneg*)
apply (*rule order-less-imp-le*)
apply (*rule pos-div-pos*)
apply *force*
apply *simp*
apply (*insert Cstar0*, *arith*)
apply (*insert eps0*, *arith*)
apply *arith*
apply *simp*
apply (*subst pos-divide-le-eq*)
apply *arith*
apply (*simp add*: *ring-eq-simps*)
apply (*rule order-trans*)
apply (*insert xtemp*)
apply (*drule-tac x* = *x* **in** *spec*)**back**
apply (*drule mp*)
apply (*rule xc*)
apply (*rule mult-left-mono*)
apply (*rule order-less-imp-le*)
apply *assumption*
apply (*rule nonneg-times-nonneg*)
apply *arith*
apply (*rule nonneg-plus-nonneg*)
apply *force*
apply (*rule nonneg-times-nonneg*)
apply *simp*
apply *simp*
apply (*subgoal-tac ?E2* ~= *0*)
apply *simp*
apply (*subst pos-divide-le-eq*)
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule pos-div-pos*)
apply *arith*
apply *arith*
apply (*simp add*: *ring-eq-simps*)
apply (*insert E2a*)
apply *arith*
apply (*rule n3*)
done
also have ... = *eps*
apply *simp*
apply (*subst add-divide-distrib* [*THEN sym*])
apply *simp*
done

```
  finally show ?thesis
    apply (simp add: abs-divide)
    apply (subgoal-tac abs u = u)
    apply simp
    apply (subst pos-divide-less-eq [THEN sym])
    apply (insert xstar1 n1 xa, arith)
    apply assumption
    apply (rule abs-nonneg)
    apply arith
    done
qedqedqedqedqedqedqed

lemma PNT2a: EX lambda C1. 0 < lambda & lambda < 1 & 1 <= C1 &
    (ALL eps. 0 < eps --> eps < 1 --> (EX x1. 1 <= x1 & (ALL K x.
      exp (C1 / eps) <= K --> x1 <= x --> (EX xstar.
        x < xstar & (1 + lambda * eps) * xstar < K * x & (ALL u.
          xstar <= u & u <= (1 + lambda * eps) * xstar -->
            abs(R u) < eps * u)))))
  apply (insert PNT2)
  apply (erule exE)
  apply (erule exE)
  apply (rule-tac x = lambda in exI)
  apply (rule-tac x = max C1 1 + 1 in exI)
  apply clarify
  apply (rule conjI)
  apply simp
  apply (rule order-trans)
  prefer 2
  apply (rule le-maxI2)
  apply force
  apply clarify
  apply (drule-tac x = eps in spec)
  apply clarify
  apply (rule-tac x = max (x1 + 1) 1 in exI)
  apply (rule conjI)
  apply (rule le-maxI2)
  apply clarify
  apply (drule-tac x = K in spec)
  apply (drule-tac x = x in spec)
  apply (subgoal-tac exp (C1 / eps) < K)
  apply (subgoal-tac x1 < x)
  apply clarify
  apply (subgoal-tac x1 + 1 <= x)
  apply arith
  apply (rule order-trans)
```

```
  prefer 2
  apply assumption
  apply (rule le-maxI1)
  apply (rule order-less-le-trans)
  prefer 2
  apply assumption
  apply (subst exp-less-cancel-iff)
  apply (rule divide-strict-right-mono)
  apply simp
  apply (rule order-le-less-trans)
  apply (subgoal-tac C1 <= max C1 1)
  apply assumption
  apply (rule le-maxI1)
  apply arith
  apply assumption
done

lemma PNT3-aux1: EX C. ALL x. 1 <= x -->
    abs (R (x)) * (ln x) ^ 2 <=
      2 * (∑ n = 1..natfloor x. abs (R (x / real n)) * ln (real n)) +
            C * abs (x * (1 + ln x))
  apply (insert bigo-lesso5 [OF error7])
  apply clarsimp
  apply (rule-tac x = C in exI)
  apply (rule allI)
  apply (drule-tac x = x - 1 in spec)
  apply clarsimp
  apply (subgoal-tac natfloor (x - 1) + 1 = natfloor x)
  apply simp
  apply (subst natfloor-add [THEN sym])
  apply simp
  apply simp
done

lemma PNT3-aux1a: EX C. 0 <= C & (ALL x. 2 <= x -->
    abs (R (x)) * (ln x) ^ 2 <=
      2 * (∑ n = 1..natfloor x. abs (R (x / real n)) * ln (real n)) +
            C * x * ln x)
  apply (insert PNT3-aux1)
  apply clarsimp
  apply (subgoal-tac 0 <= C)
  prefer 2
  apply (drule-tac x = 1 in spec)
  apply simp
  apply (rule-tac x = C + C / ln 2 in exI)
```

apply (*rule conjI*)
apply (*rule nonneg-plus-nonneg*)
apply *assumption*
apply (*subst pos-le-divide-eq*)
apply *force*
apply *simp*
apply *clarify*
apply (*drule-tac x = x* **in** *spec*)
apply *clarsimp*
apply (*erule order-trans*)
apply (*rule add-left-mono*)
apply (*subst abs-nonneg*)
apply (*subgoal-tac 0 <= ln x*)
apply *arith*
apply *force*
apply (*simp add*: *mult-ac*)
apply (*subst mult-left-commute*)**back**
apply (*rule mult-left-mono*)
apply (*simp add*: *ring-eq-simps*)
apply (*subst pos-le-divide-eq*)
apply *force*
apply (*rule mult-left-mono*)
apply *auto*
done

lemma *PNT3-aux2*: *EX C. ALL x. 0 <= x --> abs (R x) <= C * x*
apply (*insert R-bigo*)
apply (*clarsimp simp add*: *bigo-alt-def*)
apply (*rule-tac x = c* **in** *exI*)
apply *clarify*
apply (*drule-tac x = x* **in** *spec*)
apply *simp*
done

lemma *PNT3-aux2a*: *EX C. 0 < C & (ALL x. 0 <= x --> abs (R x) <= C * x*)
apply (*insert PNT3-aux2*)
apply *clarify*
apply (*rule-tac x = C + 1* **in** *exI*)
apply (*rule conjI*)
apply (*subgoal-tac 0 <= C*)
apply *simp*
apply (*drule-tac x = 1* **in** *spec*)
apply *clarsimp*
apply *arith*

```
  apply clarify
  apply (drule-tac x = x in spec)
  apply (auto simp add: ring-eq-simps)
done

lemma PNT3-aux3: EX C. ALL x. 1 <= x -->
    abs((∑ i = 1..natfloor x. ln (real i) / real i) -
      (ln x) ^ 2 / 2) <= C
  apply (insert identity-four-real-b-cor)
  apply (drule set-plus-imp-minus)
  apply (simp only: func-diff)
  apply (simp only: bigo-alt-def)
  apply clarsimp
  apply (rule-tac x = c in exI)
  apply (rule allI)
  apply (drule-tac x = (x - 1) in spec)
  apply clarify
  apply (subgoal-tac natfloor (abs (x - 1)) + 1 = natfloor x)
  apply (subgoal-tac abs (x - 1) + 1 = x)
  apply (simp only:)
  apply (erule Selberg.aux2)
  apply (erule Selberg.aux)
done

lemma PNT3-aux4: EX C. ALL x x2. 1 <= x2 --> x2 <= x -->
    (∑ n = natfloor (x / x2) + 1..natfloor x.
      ln (real n) / real n) <=
        ln x2 * ln x + C
proof -
  from PNT3-aux3 obtain C where C: ALL x. 1 <= x -->
    abs((∑ i = 1..natfloor x. ln (real i) / real i) -
      (ln x) ^ 2 / 2) <= C..
  show ?thesis
  proof (rule exI, clarify)
  fix x::real
  fix x2::real
  assume a: 1 <= x2
  assume b: x2 <= x
  show (∑ n = natfloor (x / x2) + 1..natfloor x.
      ln (real n) / real n) <=
    ln x2 * ln x + (C + C)
  proof -
  let ?term1 = (((ln x) ^ 2 / 2) - (ln (x / x2)) ^2 / 2)
  have (∑ n:{natfloor (x / x2) + 1..natfloor x}.
          ln (real n) / real n) =
```

```
    (∑ n:{1..natfloor x}. ln (real n) / real n) −
    (∑ n:{1..natfloor (x / x2)}. ln (real n) / real n)
  apply (simp add: compare-rls)
  apply (subst setsum-Un-disjoint [THEN sym])
  apply simp
  apply simp
  apply force
  apply (rule setsum-cong)
  apply auto
  apply (erule order-trans)
  apply (rule natfloor-mono)
  apply (subst pos-divide-le-eq)
  apply (insert a, arith)
  apply (subgoal-tac ?x * 1 <= ?x * x2)
  apply simp
  apply (rule mult-left-mono)
  apply assumption
  apply (insert b, arith)
  done
then have abs( ?term1 − (∑ n:{natfloor (x / x2) + 1..natfloor x}.
       ln (real n) / real n)) =
    (abs (?term1 − ...)) by simp
also have ... = abs(
  (ln x ^ 2 / 2 −
   (∑ n = 1..natfloor x. ln(real n) / real n)) +
  ((∑ n = 1..natfloor (x / x2). ln (real n) / real n) −
   ln (x / x2) ^ 2 / 2))
  by (simp add: ring-eq-simps)
also have ... <= abs((ln x ^ 2 / 2 −
   (∑ n = 1..natfloor x. ln(real n) / real n))) +
  abs((∑ n = 1..natfloor (x / x2). ln (real n) / real n) −
   ln (x / x2) ^ 2 / 2)
  by (rule abs-triangle-ineq)
also have ... <= C + C
  apply (rule add-mono)
  apply (subst abs-diff)
  apply (insert prems, force)
  apply (subgoal-tac 1 <= x / x2)
  apply force
  apply (subst pos-le-divide-eq)
  apply arith
  apply simp
  done
also have ?term1 = (2 * ln x * ln x2 − ln x2 ^ 2) / 2
  apply (subst ln-div)
```

```
    apply (insert a b, arith)
    apply arith
    apply (simp add: power2-eq-square diff-divide-distrib
      ring-eq-simps)
    done
  also have ... = ln x2 * (2 * ln x - ln x2) / 2
    by (simp add: ring-eq-simps power2-eq-square)
  finally have x:
    abs((∑ n = natfloor (x / x2) + 1..natfloor x. ln (real n) / real n) -
      ln x2 * (2 * ln x - ln x2) / 2) <=
    C + C
    apply (subst abs-diff)
    apply assumption
    done
  have (∑ n = natfloor (x / x2) + 1..natfloor x.
      ln (real n) / real n) - ln x2 * (2 * ln x - ln x2) / 2 <=
    C + C
    apply (rule order-trans)
    prefer 2
    apply (rule x)
    apply (rule abs-ge-self)
    done
  then have (∑ n = natfloor (x / x2) + 1..natfloor x.
      ln (real n) / real n) <=
    ln x2 * (2 * ln x - ln x2) / 2 + (C + C)
    by arith
  also have ln x2 * (2 * ln x - ln x2) / 2 <= ln x2 * ln x
    apply (subst pos-divide-le-eq)
    apply force
    apply (subst mult-assoc)
    apply (rule mult-left-mono)
    apply simp
    apply (insert a b, auto)
    done
  finally show ?thesis by auto
qedqedqed

lemma PNT3-aux5: EX C. 0 <= C & (ALL x. 1 <= x -->
    (∑ i = 1..natfloor x. ln (real i) / real i) <=
      (ln x) ^ 2 / 2 + C)
  apply (insert PNT3-aux3)
  apply clarify
  apply (rule-tac x = C in exI)
  apply (rule conjI)
  apply (drule-tac x = 1 in spec)
```

```
  apply simp
  apply clarify
  apply (drule-tac x = x in spec)
  apply clarify
  apply (subgoal-tac (∑ i = 1..natfloor x. ln (real i) / real i) −
            ln x ^ 2 / 2 <= C)
  apply arith
  apply (rule order-trans)
  prefer 2
  apply assumption
  apply (rule abs-ge-self)
  done

lemma PNT3: EX G. ALL x x2 S eps alpha.
  0 < eps --> 0 < alpha --> alpha < c -->
    2 <= x2 --> exp 1 <= x2 --> x2 <= x -->
      (ALL x. x2 <= x --> abs(R x) <= alpha * x) -->
        S <= {1..natfloor x} -->
          (ALL n:S. abs(R(x / real n)) <= eps * (x / real n)) -->
  abs (R x)
    <= alpha * x −
      2 * (alpha − eps) * x * (∑n:S. ln (real n) / real n) / ln x ^ 2 +
       G * x * ln x2 / ln x
proof −
  from PNT3-aux1a obtain C where C: 0 <= C & (ALL x. 2 <= x -->
    abs (R (x)) * (ln x) ^ 2 <=
      2 * (∑ n = 1..natfloor x. abs (R (x / real n)) * ln (real n)) +
            C * x * ln x)..
  then have C1: ALL x. 2 <= x -->
    abs (R (x)) * (ln x) ^ 2 <=
      2 * (∑ n = 1..natfloor x. abs (R (x / real n)) * ln (real n)) +
            C * x * ln x by blast
  from C have C2: 0 <= C by blast
  from PNT3-aux2 obtain D where D: ALL x. 0 <= x --> abs (R x) <= D
* x..
  then have D': 0 <= D
    apply (drule-tac x = 1 in spec)
    apply simp
    apply arith
    done
  from PNT3-aux4 obtain E where E: ALL x x2. 1 <= x2 --> x2 <= x -->
    (∑ n = natfloor (x / x2) + 1..natfloor x. ln (real n) / real n) <=
    ln x2 * ln x + E..
  then have E': 0 <= E
    apply (drule-tac x = 1 in spec)
```

apply (*drule-tac x = 1* **in** *spec*)
apply *simp*
done
from *PNT3-aux5* **obtain** *F* **where** *F*: *0 <= F & (ALL x. 1 <= x −−>*
(∑ i = 1..natfloor x. ln (real i) / real i) <=
(ln x) ^ 2 / 2 + F)..
then have *F1*: *ALL x. 1 <= x −−>*
(∑ i = 1..natfloor x. ln (real i) / real i) <=
(ln x) ^ 2 / 2 + F **by** *blast*
from *F* **have** *F2*: *0 <= F* **by** *blast*
show *?thesis*
proof ((*rule exI*)+, *clarify*)
fix *eps*::*real*
fix *alpha*::*real*
fix *x*::*real*
fix *x2*::*real*
fix *S*::*nat set*
assume *a*: *0 < eps*
assume *aa*: *0 < alpha*
assume *aaa*: *alpha < c*
assume *b*: *2 <= x2*
assume *bb*: *exp 1 <= x2*
assume *c*: *x2 <= x*
assume *d*: *ALL x. x2 <= x −−> abs(R x) <= alpha * x*
assume *e*: *S <= {1..natfloor x}*
assume *f*: *ALL n:S. abs(R(x / real n)) <= eps * (x / real n)*
have *(∑ n = 1..natfloor x. abs (R (x / real n)) * ln (real n)) =*
*(∑ n:{1..natfloor x} Int S. abs (R (x / real n)) * ln (real n)) +*
*(∑ n:{1..natfloor x} − S. abs (R (x / real n)) * ln (real n))*
apply (*subst setsum-Un-disjoint* [*THEN sym*])
apply (*rule finite-subset*)
apply (*subgoal-tac ?t <= {1..natfloor x}*)
apply *assumption*
apply *force*
apply *force*
apply (*rule finite-subset*)
apply (*subgoal-tac ?t <= {1..natfloor x}*)
apply *assumption*
apply *force*
apply *force*
apply *force*
apply (*rule setsum-cong*)
apply *force*
apply (*rule refl*)
done

also have
$(\sum n{:}\{1..natfloor\ x\} - S.\ abs\ (R\ (x\ /\ real\ n)) * ln\ (real\ n)) =$
$(\sum n{:}\{1..natfloor\ (x\ /\ x2)\} - S.$
$abs\ (R\ (x\ /\ real\ n)) * ln\ (real\ n)) +$
$(\sum n{:}\{natfloor\ (x\ /\ x2) + 1..natfloor\ x\} - S.$
$abs\ (R\ (x\ /\ real\ n)) * ln\ (real\ n))$
apply (*subst setsum-Un-disjoint* [*THEN sym*])
apply (*rule finite-subset*)
apply (*subgoal-tac ?t <= {1..natfloor (x / x2)}*)
apply *assumption*
apply *force*
apply *force*
apply (*rule finite-subset*)
apply (*subgoal-tac ?t <= {natfloor (x / x2)..natfloor x}*)
apply *assumption*
apply *force*
apply *force*
apply *force*
apply (*rule setsum-cong*)
apply *auto*
apply (*erule order-trans*)
apply (*rule natfloor-mono*)
apply (*subst pos-divide-le-eq*)
apply (*insert b, arith*)
apply (*subgoal-tac ?x * 1 <= ?x * x2*)
apply *simp*
apply (*rule mult-left-mono*)
apply (*insert b, arith*)
apply (*insert c, arith*)
done
also have
$(\sum\ n{:}\{1..natfloor\ x\}\ Int\ S.\ abs\ (R\ (x\ /\ real\ n)) * ln\ (real\ n)) <=$
$(\sum\ n{:}\{1..natfloor\ x\}\ Int\ S.\ eps * (x\ /\ real\ n) * ln\ (real\ n))$
apply (*rule setsum-le-cong*)
apply (*rule mult-right-mono*)
apply (*insert f, force*)
apply *force*
done
also have ... $= eps * x *$
$(\sum\ n{:}\{1..natfloor\ x\}\ Int\ S.\ ln\ (real\ n)\ /\ real\ n)$
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
done
also have $(\sum n{:}\{1..natfloor\ (x\ /\ x2)\} - S.$

abs (*R* (*x* / *real n*)) ∗ *ln* (*real n*)) <=
(∑ *n*:{*1*..*natfloor* (*x* / *x2*)} − *S*.
alpha ∗ (*x* / *real n*) ∗ *ln* (*real n*))
apply (*rule setsum-le-cong*)
apply (*rule mult-right-mono*)
apply *clarsimp*
apply (*insert d*)
apply (*drule spec*)
apply (*drule mp*)
prefer *2*
apply (*rule order-trans*)
apply *assumption*
apply *simp*
apply (*subst pos-le-divide-eq*)
apply *force*
apply (*subst mult-commute*)
apply (*subst pos-le-divide-eq* [*THEN sym*])
apply (*insert b*, *arith*)
apply (*rule nat-le-natfloor*)
apply (*rule order-less-imp-le*)
apply (*rule pos-div-pos*)
apply (*insert c*, *arith*)
apply *arith*
apply *assumption*
apply *simp*
apply *force*
done
also have ... = *alpha* ∗ *x* ∗ (∑ *n*:{*1*..*natfloor* (*x* / *x2*)} − *S*.
ln (*real n*) / *real n*)
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
done
also have (∑ *n*:{*natfloor* (*x* / *x2*) + *1*..*natfloor x*} − *S*.
abs (*R* (*x* / *real n*)) ∗ *ln* (*real n*)) <=
(∑ *n*:{*natfloor* (*x* / *x2*) + *1*..*natfloor x*}.
abs (*R* (*x* / *real n*)) ∗ *ln* (*real n*))
apply (*rule setsum-le-cong2*)
apply *simp*
apply *force*
apply (*rule nonneg-times-nonneg*)
apply *auto*
done
also have *alpha* ∗ *x* ∗
(∑ *n*:{*1*..*natfloor* (*x* / *x2*)} − *S*. *ln* (*real n*) / *real n*) <=

alpha $*$ *x* $*$
($\sum$ *n*:{*1*..*natfloor x*} $-$ *S*. *ln* (*real n*) / *real n*)
apply (*rule mult-left-mono*)
apply (*rule setsum-le-cong2*)
apply (*rule finite-subset*)
apply (*subgoal-tac* {*1*..*natfloor x*} $-$ *S* <= {*1*..*natfloor x*})
apply *assumption*
apply *force*
apply *simp*
apply *clarsimp*
apply (*erule order-trans*)
apply (*rule natfloor-mono*)
apply (*subst pos-divide-le-eq*)
apply (*insert b c*, *arith*)
apply (*subgoal-tac ?x* $*$ *1* <= *?x* $*$ *x2*)
apply *simp*
apply (*rule mult-left-mono*)
apply *arith*
apply *simp*
apply (*rule real-ge-zero-div-gt-zero*)
apply *force*
apply *force*
apply (*rule nonneg-times-nonneg*)
apply (*insert d*)
apply (*drule-tac x* = *x* **in** *spec*)
apply *clarsimp*
apply (*rule order-trans*)
apply (*subgoal-tac 0* <= *abs*(*R x*) / *x*)
apply *assumption*
apply (*rule real-ge-zero-div-gt-zero*)
apply *force*
apply *arith*
apply (*subst pos-divide-le-eq*)
apply *arith*
apply *assumption*
apply *arith*
done
finally have
($\sum$ *n* = *1*..*natfloor x*. *abs* (*R* (*x* / *real n*)) $*$ *ln* (*real n*))
<= *eps* $*$ *x* $*$ ($\sum$ *n*:{*1*..*natfloor x*} *Int S*. *ln* (*real n*) / *real n*) +
(*alpha* $*$ *x* $*$ ($\sum$ *n*:{*1*..*natfloor x*} $-$ *S*. *ln* (*real n*) / *real n*) +
($\sum$ *n* = *natfloor* (*x* / *x2*) + *1*..*natfloor x*.
abs (*R* (*x* / *real n*)) $*$ *ln* (*real n*)))
by *arith*
also have ... =

```
  alpha * x * (∑ n:{1..natfloor x}. ln (real n) / real n)
    - ((alpha - eps) * x *
      (∑ n:{1..natfloor x} Int S. ln (real n) / real n)) +
        (∑ n = natfloor (x / x2) + 1..natfloor x.
          abs (R (x / real n)) * ln (real n))
  apply (simp add: ring-eq-simps)
  apply (subst right-distrib [THEN sym])
  apply (rule arg-cong)back
  apply (subst right-distrib [THEN sym])
  apply (rule arg-cong)back
  apply (subst setsum-Un-disjoint [THEN sym])
  apply (rule finite-subset)
  apply (subgoal-tac ?t <= {1..natfloor x})
  apply assumption
  apply force
  apply simp
  apply (rule finite-subset)
  apply (subgoal-tac ?t <= {1..natfloor x})
  apply assumption
  apply force
  apply simp
  apply force
  apply (rule setsum-cong)
  apply force
  apply (rule refl)
  done
finally have g:
    (∑ n = 1..natfloor x. abs (R (x / real n)) * ln (real n))
  <= alpha * x * (∑ n = 1..natfloor x. ln (real n) / real n) -
    (alpha - eps) * x *
      (∑ n:{1..natfloor x} Int S. ln (real n) / real n) +
    (∑ n = natfloor (x / x2) + 1..natfloor x.
      abs (R (x / real n)) * ln (real n)).
have abs (R (x)) * (ln x) ^ 2 <=
    2 * (∑ n = 1..natfloor x. abs (R (x / real n)) * ln (real n)) +
      C * x * ln x
  by (insert C b c, auto)
also note g
also have (∑ n = natfloor (x / x2) + 1..natfloor x.
    abs (R (x / real n)) * ln (real n)) <=
  (∑ n = natfloor (x / x2) + 1..natfloor x.
        D * (x / real n) * ln (real n))
  apply (rule setsum-le-cong)
  apply (rule mult-right-mono)
  apply (insert D)
```

apply (*subgoal-tac 0 <= ?x / real x*)
apply *force*
apply (*subst pos-le-divide-eq*)
apply *simp*
apply *arith*
apply (*insert b c*)
apply *simp*
apply *force*
done
also have ... = $D * x * (\sum\ n = natfloor\ (x\ /\ x2) + 1..natfloor\ x.$
$(ln\ (real\ n)\ /\ real\ n))$
apply (*subst setsum-const-times* [*THEN sym*])
apply (*rule setsum-cong2*)
apply *simp*
done
also have ... <= $D * x * (ln\ x2 * ln\ x + E)$
apply (*rule mult-left-mono*)
apply (*insert E b c*, *force*)
apply (*rule nonneg-times-nonneg*)
apply (*rule D'*)
apply *arith*
done
also have $alpha * x * (\sum\ n = 1..natfloor\ x.\ ln\ (real\ n)\ /\ real\ n) <=$
$alpha * x * ((ln\ x)\hat{}2\ /\ 2 + F)$
apply (*rule mult-left-mono*)
apply (*insert F b c*, *force*)
apply (*rule nonneg-times-nonneg*)
apply (*insert aa*, *auto*)
done
also have $\{1..natfloor\ x\}\ Int\ S = S$
apply (*insert e*)
apply *blast*
done
finally have $abs\ (R\ x) * ln\ x\ \hat{}\ 2$
$<= 2 *$
$(alpha * x * (ln\ x\ \hat{}\ 2\ /\ 2 + F) -$
$(alpha - eps) * x *$
$(\sum n{:}S.\ ln\ (real\ n)\ /\ real\ n) +$
$D * x * (ln\ x2 * ln\ x + E)) +$
$C * x * ln\ x$ **by** *auto*
also have ... = $alpha * x * ln\ x\ \hat{}\ 2 -$
$2 * (alpha - eps) * x * (\sum n{:}S.\ ln\ (real\ n)\ /\ real\ n) +$
$(x * ln\ x * ((2 * D) * ln\ x2 + C) + x * ((2 * F) * alpha + 2 * D * E))$
by (*simp add*: *ring-eq-simps*)
also have ... <=

alpha $*$ *x* $*$ *ln x* ^ *2* $-$
2 $*$ (*alpha* $-$ *eps*) $*$ *x* $*$ ($\sum$ *n*:*S*. *ln* (*real n*) / *real n*) $+$
(*x* $*$ *ln x* $*$ ((*2* $*$ *D*) $*$ *ln x2* $+$ *C* $*$ *ln x2*) $+$
(*x* $*$ *ln x* $*$ *ln x2*) $*$ ((*2* $*$ *F*) $*$ *c* $+$ *2* $*$ *D* $*$ *E*))
apply (*rule add-left-mono*)
apply (*rule add-mono*)
apply (*rule mult-left-mono*)
apply (*rule add-left-mono*)
apply (*subgoal-tac C* $*$ *ln* (*exp 1*) $<=$ *C* $*$ *ln x2*)
apply *simp*
apply (*rule mult-left-mono*)
apply (*subst ln-le-cancel-iff*)
apply *force*
apply (*insert a aa aaa b bb c*)
apply *arith*
apply (*rule bb*)
apply (*rule C2*)
apply (*rule nonneg-times-nonneg*)
apply *arith*
apply (*rule ln-ge-zero*)
apply *arith*
apply (*rule mult-mono*)
apply (*subgoal-tac x* $*$ *ln* (*exp 1*) $*$ *ln* (*exp 1*) $<=$ *x* $*$ *ln x* $*$ *ln x2*)
apply *simp*
apply (*rule mult-mono*)
apply (*rule mult-left-mono*)
apply (*subst ln-le-cancel-iff*)
apply (*force*, *force*, *force*, *force*)
apply (*subst ln-le-cancel-iff*)
apply (*force*, *force*, *force*)
apply (*rule nonneg-times-nonneg*)
apply *force*
apply (*rule ln-ge-zero*)
apply *force*
apply *force*
apply (*rule add-right-mono*)
apply (*rule mult-left-mono*)
apply (*erule order-less-imp-le*)
apply (*rule nonneg-times-nonneg*)
apply *force*
apply (*rule F2*)
apply (*rule nonneg-times-nonneg*)+
apply *force*
apply (*rule ln-ge-zero*)
apply *arith*

apply (*rule ln-ge-zero*)
apply *arith*
apply (*rule nonneg-plus-nonneg*)
apply (*rule nonneg-times-nonneg*)
apply (*rule nonneg-times-nonneg*)
apply *force*
apply (*rule F2*)
apply *arith*
apply (*rule nonneg-times-nonneg*)+
apply *force*
apply (*rule D'*)
apply (*rule E'*)
done
finally have *x*: *abs* (*R x*) * *ln x* ^ *2*
<= *alpha* * *x* * *ln x* ^ *2* −
2 * (*alpha* − *eps*) * *x* * (∑ *n*:*S*. *ln* (*real n*) / *real n*) +
x * *ln x* * *ln x2* * (*2* * *D* + *C* + *2* * *F* * *c* + *2* * *D* * *E*)
(**is** *?term1* <= *?term2* − *?term3* + *?term4* * *?term5*)
by (*simp add*: *ring-eq-simps*)
have *y*: *ln x* ~= *0*
apply (*rule less-imp-neq* [*THEN not-sym*])
apply (*rule ln-gt-zero*)
apply (*insert b c*, *arith*)
done
have *z*: (*ln x* ^ *2*) ~= *0*
apply (*rule less-imp-neq* [*THEN not-sym*])
apply (*subst power2-eq-square*)
apply (*rule mult-pos*)
apply (*rule ln-gt-zero*)
apply (*insert b c*, *arith*)
apply (*rule ln-gt-zero*)
apply (*insert b c*, *arith*)
done
from *x* **have** *?term1* / (*ln x* ^ *2*) <=
(*?term2* − *?term3* + *?term4* * *?term5*) / (*ln x* ^ *2*)
apply (*rule divide-right-mono*)
apply *force*
done
also have *?term1* / (*ln x*^*2*) = *abs* (*R x*)
by (*insert y*, *simp*)
also have (*?term2* − *?term3* + *?term4* * *?term5*) / (*ln x* ^ *2*) =
?term2 / (*ln x* ^ *2*) − *?term3* / (*ln x* ^ *2*) + *?term4* / (*ln x* ^ *2*) * *?term5*
by (*simp add*: *ring-eq-simps add-divide-distrib diff-divide-distrib*)
also have *?term2* / (*ln x* ^ *2*) = *alpha* * *x*
by (*insert z*, *simp*)

```
  also have ?term4 / (ln x ^ 2) * ?term5 = ?term5 * x * ln x2 / ln x
    by (simp add: power2-eq-square y)
  finally show abs (R x)
    <= alpha * x -
      2 * (alpha - eps) * x * (∑ n:S. ln (real n) / real n) / ln x ^ 2 +
        ?term5 * x * ln x2 / ln x.
qedqed

lemma PNT4-aux1: 1 < c2 ==> EX C. 0 < C & (ALL alpha.
  0 < alpha --> alpha < c2 -->
    (EX xlarge. ALL x. xlarge <= x -->
      (EX S. S <= {1..natfloor x} &
        (ALL n:S. abs(R(x / real n)) <= (alpha / c2) * (x / real n)) &
          C * alpha^2 * (ln x)^2
            <= (∑ n:S. ln(real n) / (real n)))))
proof -
  assume c2-gt-1: 1 < c2
  from PNT2a obtain lambda c1 where lambdac1: 0 < lambda &
    lambda < 1 & 1 <= c1 & (ALL eps. 0 < eps --> eps < 1 -->
      (EX x1. 1 <= x1 & (ALL K x. exp (c1 / eps) <= K --> x1 <= x -->
        (EX xstar. x < xstar &
          (1 + lambda * eps) * xstar < K * x &
            (ALL u. xstar <= u & u <= (1 + lambda * eps) * xstar -->
              abs (R u) < eps * u))))) by blast
  from lambdac1 have lambda-gt-0: 0 < lambda by blast
  from lambdac1 have lambda-le-1: lambda < 1 by blast
  from lambdac1 have c1-gt-1: 1 <= c1 by blast
  show ?thesis
proof
  let ?C = lambda / (32 * c1 * c2 ^ 2)
  show 0 < ?C & (ALL alpha.
    0 < alpha --> alpha < c2 -->
      (EX xlarge. ALL x. xlarge <= x -->
        (EX S. S <= {1..natfloor x} &
          (ALL n:S. abs(R(x / real n)) <= (alpha / c2) * (x / real n)) &
            ?C * alpha^2 * (ln x)^2
              <= (∑ n:S. ln(real n) / (real n)))))
proof
  show 0 < ?C
    apply (rule pos-div-pos)
    apply (rule lambda-gt-0)
    apply (rule mult-pos)
    apply (rule mult-pos)
    apply force
    apply (insert c1-gt-1, arith)
```

```
    apply (subst power2-eq-square)
    apply (rule mult-pos)
    apply (insert c2-gt-1, arith, arith)
    done
next show ALL alpha.
    0 < alpha --> alpha < c2 -->
     (EX xlarge. ALL x. xlarge <= x -->
       (EX S. S <= {1..natfloor x} &
         (ALL n:S. abs(R(x / real n)) <= (alpha / c2) * (x / real n)) &
           ?C * alpha^2 * (ln x)^2
             <= (SUM n:S. ln(real n) / (real n))))
proof (clarify)
  fix alpha::real
  assume alpha-gt-0: 0 < (alpha::real)
  assume alpha-lt-c2: alpha < c2
  let ?eps = alpha / c2
  have eps-gt-0: 0 < ?eps
    apply (rule pos-div-pos)
    apply (rule alpha-gt-0)
    apply (insert c2-gt-1, arith)
    done
  have eps-lt-1: ?eps < 1
    apply (subst pos-divide-less-eq)
    apply (insert alpha-gt-0 alpha-lt-c2, arith)
    apply simp
    done
  from lambdac1 eps-gt-0 eps-lt-1 have EX x1. 1 <= x1 & (ALL K x.
      exp (c1 / ?eps) <= K --> x1 <= x --> (EX xstar.
       x < xstar & (1 + lambda * ?eps) * xstar < K * x & (ALL u.
         xstar <= u & u <= (1 + lambda * ?eps) * xstar -->
           abs(R u) < ?eps * u))) by blast
  then obtain x1 where x1c1: 1 <= x1 & (ALL K x.
      exp (c1 / ?eps) <= K --> x1 <= x --> (EX xstar.
       x < xstar & (1 + lambda * ?eps) * xstar < K * x & (ALL u.
         xstar <= u & u <= (1 + lambda * ?eps) * xstar -->
           abs(R u) < ?eps * u)))..
  then have x1-ge-1: 1 <= x1 by blast
  let ?K = exp (c1 / ?eps)
  have K-gt-1: 1 < ?K
    apply (subgoal-tac exp 0 < ?K)
    apply simp
    apply (subst exp-less-cancel-iff)
    apply (rule pos-div-pos)
    apply (insert c1-gt-1, arith)
    apply (insert eps-gt-0, arith)
```

```
  done
 from x1c1 have x1K: ALL x. x1 <= x --> (EX xstar.
      x < xstar & (1 + lambda * ?eps) * xstar < ?K * x & (ALL u.
       xstar <= u & u <= (1 + lambda * ?eps) * xstar -->
        abs(R u) < ?eps * u))
  by auto
 let ?xlarge = max (max (max (max (max (max
    2 (?K powr 2)) (exp 2 + 1))
     ((2 / (1 - 1 / (1 + lambda * ?eps))) powr 2))
      (exp ((alpha * (ln x1 * 2) + c1 * (c2 * 4)) / alpha)))
       (exp (2 * (c1 * c2) / alpha)))
        (x1 powr 8))
         (exp (3 / (alpha / (8 * (c1 * c2))))))
 show EX xlarge. ALL x. xlarge <= x --> (EX S<={1..natfloor x}.
    (ALL n:S. abs (R (x / real n)) <= alpha / c2 * (x / real n)) &
      ?C * alpha ^ 2 * ln x ^ 2
       <= (sum n:S. ln (real n) / real n))
proof (rule exI, clarify)
 fix x::real
 assume xlarge: ?xlarge <= x
 then have x-gt-1: 1 < (x::real) and
   x-ge-K-squared: ?K powr 2 <= x and
   x-gt-exp2: exp 2 < x and
   x-ge-blah: (2 / (1 - 1 / (1 + lambda * ?eps))) powr 2 <= x and
   x-ge-blah2: exp ((alpha * (ln x1 * 2) + c1 * (c2 * 4)) / alpha)
    <= x and
   x-ge-blah3: exp (2 * (c1 * c2) / alpha) <= x and
   x-ge-blah4: x1 powr 8 <= x and
   x-ge-blah5: exp (3 / (alpha / (8 * (c1 * c2)))) <= x
    by auto

 let ?jstar = natfloor (ln x1 / ln ?K) + 1
 have jstar: !!j. ?jstar < j ==> x1 <= ?K powr (real j)
  apply (subst ln-le-cancel-iff [THEN sym])
  apply simp
  apply (insert x1-ge-1, arith)
  apply force
  apply (subst ln-pwr)
  apply force
  apply arith
  apply (rule order-less-imp-le)
  apply (subst pos-divide-less-eq [THEN sym])
  apply (rule ln-gt-zero)
  apply (rule exp-gt-one)
  apply (rule pos-div-pos)
```

apply (*insert c1-gt-1*, *arith*)
apply (*rule eps-gt-0*)
apply (*rule ge-natfloor-plus-one-imp-gt*)
apply (*erule order-less-imp-le*)
done

let *?jhat* = *natfloor* (*ln x* / (*2* ∗ *ln ?K*)) − *1*
have *jhat*: !!*j*. *j* <= *?jhat* ==> *?K powr* ((*real j*) + *1*) <= *x powr* (*1* / *2*)
apply (*subst ln-le-cancel-iff* [*THEN sym*])
apply *force*
apply (*rule powr-gt-zero*)
apply (*subst ln-pwr*)
apply (*insert x-gt-1*, *arith*)
apply *simp*
apply (*subst ln-pwr*)
apply *simp*
apply *simp*
apply *simp*
apply (*subst pos-divide-le-eq*)
apply (*rule alpha-gt-0*)
apply (*subgoal-tac real* (*j* + *1*) ∗ (*c1* ∗ (*c2* ∗ *2*)) <= *alpha* ∗ *ln x*)
apply (*simp add*: *ring-eq-simps compare-rls*)
apply (*subst pos-le-divide-eq* [*THEN sym*])
apply (*rule mult-pos*)
apply (*insert c1-gt-1*, *arith*)
apply (*rule mult-pos*)
apply (*insert c2-gt-1*, *arith*)
apply *force*
apply (*rule nat-le-natfloor*)
apply (*rule real-ge-zero-div-gt-zero*)
apply (*rule nonneg-times-nonneg*)
apply (*rule order-less-imp-le*)
apply (*rule alpha-gt-0*)
apply (*rule ln-ge-zero*)
apply (*rule order-less-imp-le*)
apply (*rule x-gt-1*)
apply (*rule mult-pos*)
apply (*insert c1-gt-1*, *arith*)
apply *arith*
apply (*simp add*: *mult-ac*)
apply (*subgoal-tac natfloor 1* <=
natfloor (*alpha* ∗ *ln x* / (*c1* ∗ (*c2* ∗ *2*))))
apply *simp*
apply *arith*
apply (*rule natfloor-mono*)

apply (*subst pos-le-divide-eq*)
apply (*simp add*: *pos-le-divide-eq mult-ac*)
apply (*rule mult-pos*)
apply *arith*
apply *arith*
apply *simp*
apply (*insert x-ge-K-squared*)
apply (*subgoal-tac 2 * ln (exp (c1 / (alpha / c2))) <= ln x*)
apply *simp*
apply (*subst mult-commute*)
apply (*subst pos-divide-le-eq [THEN sym]*)
apply (*rule alpha-gt-0*)
apply (*simp add*: *mult-ac*)
apply (*subst ln-pwr [THEN sym]*)
apply *auto*
done
have *xstar-aux*: *!!j. ?jstar < j ==>*
(*EX xstar.*
*?K powr (real j) < xstar & (1 + lambda * ?eps) * xstar <*
*?K * (?K powr (real j)) & (ALL u.*
*xstar <= u & u <= (1 + lambda * ?eps) * xstar -->*
*abs(R u) < ?eps * u))*
apply (*insert x1K*)
apply (*drule-tac x = ?K powr (real j)* **in** *spec*)
apply (*drule mp*)
apply (*erule jstar*)
apply *assumption*
done
have *EX xstar. ALL j. ?jstar < j -->*
*?K powr (real j) < xstar j & (1 + lambda * ?eps) **
*xstar j < ?K * (?K powr (real j)) & (ALL u.*
*xstar j <= u & u <= (1 + lambda * ?eps) * xstar j -->*
*abs(R u) < ?eps * u)*
apply (*rule exI*)
apply (*clarify*)
apply (*drule xstar-aux*)
apply (*erule someI-ex*)
done
then obtain *xstar* **where** *xstar*: *ALL j. ?jstar < j -->*
*?K powr (real j) < xstar j & (1 + lambda * ?eps) **
*xstar j < ?K * (?K powr (real j)) & (ALL u.*
*xstar j <= u & u <= (1 + lambda * ?eps) * xstar j -->*
*abs(R u) < ?eps * u)*..
then have *xstar0*: *!!j u. ?jstar < j ==> xstar j <= u ==>*
*u <= (1 + lambda * ?eps) * xstar j ==>*

```
    abs (R u) < ?eps * u by blast
from xstar have xstar1: !!j. ?jstar < j ==> ?K powr (real j) < xstar j
 by blast
have xstar1a: !!j. ?jstar < j ==> 0 < xstar j
 apply (rule order-le-less-trans)
 prefer 2
 apply (erule xstar1)
 apply force
 done
from xstar have xstar2: !!j. ?jstar < j ==>
   (1 + lambda * ?eps) * xstar j < ?K * (?K powr (real j)) by blast
have xstar2a: !!j. ?jstar < j ==>
   xstar j < ?K powr (real j + 1)
 apply (rule order-less-trans)
 prefer 2
 apply (subgoal-tac ?K powr (real j + 1) =
   ?K * (?K powr (real j)))
 apply (erule ssubst)
 apply (erule xstar2)
 apply (simp add: powr-add)
 apply (subgoal-tac 1 * xstar j < ?t)
 apply simp
 apply (rule mult-strict-right-mono)
 apply simp
 apply (rule pos-div-pos)
 apply (rule mult-pos)
 apply (rule lambda-gt-0)
 apply (rule alpha-gt-0)
 apply (insert c2-gt-1, arith)
 apply (erule xstar1a)
 done
have xstar2b: !!j. ?jstar < j ==> j <= ?jhat ==>
   xstar j < x powr (1 / 2)
 apply (rule order-less-le-trans)
 apply (erule xstar2a)
 apply (erule jhat)
 done
have xstar2c: !!j. ?jstar < j ==> j <= ?jhat ==>
   x powr (1 / 2) < x / xstar j
 apply (subst pos-less-divide-eq)
 apply (erule xstar1a)
 apply (subst mult-commute)
 apply (subst pos-less-divide-eq [THEN sym])
 apply force
 apply (subgoal-tac x = x powr 1)
```

apply (*erule ssubst*) **backback**
apply (*subst powr-divide2*)
apply *simp*
apply (*rule xstar2b*)
apply *force*
apply *force*
apply (*rule sym*)
apply (*subst powr-one-gt-zero-iff*)
apply (*insert x-gt-1*, *arith*)
done
from *xstar* **have** *xstar3*: !!*j*. *?jstar* < *j* ==> *xstar j* <= *u* ==>
u <= (*1* + *lambda* * *?eps*) * *xstar j* ==> *abs*(*R u*) < *?eps* * *u* **by** *blast*
have *xstar4-aux*: !!*j n*. *?jstar* < *j* ==> *xstar j* <= *x* / *real* (*n*::*nat*) ==>
0 < *real n*
apply (*subgoal-tac* ~(*n* = *0*))
apply *arith*
apply (*rule notI*)
apply (*frule xstar1a*)
apply *simp*
done
have *xstar4*: !!*j n*. *?jstar* < *j* ==> *xstar j* <= *x* / *real* (*n*::*nat*) ==>
real n <= *x* / *xstar j*
apply (*subst pos-le-divide-eq*)
apply (*erule xstar1a*)
apply (*subst mult-commute*)
apply (*subst pos-le-divide-eq* [*THEN sym*])
apply (*erule xstar4-aux*)
apply *assumption*+
done
have *xstar5*: !!*j n*. *?jstar* < *j* ==> *xstar j* <= *x* / *real* (*n*::*nat*) ==>
real n < *x* / (*?K powr* (*real j*))
apply (*rule order-le-less-trans*)
apply (*erule xstar4*)
apply *assumption*
apply (*rule divide-strict-left-mono*)
apply (*erule xstar1*)
apply (*insert x-gt-1*, *arith*)
apply (*rule mult-pos*)
apply (*erule xstar1a*)
apply *force*
done
have *xstar6*: !!*j n*. *?jstar* < *j* ==>
xstar j <= *x* / *real* (*n*::*nat*) ==>
x / *real n* <= (*1* + *lambda* * *?eps*) * *xstar j* ==>
x / ((*1* + *lambda* * *?eps*) * *xstar j*) <= *real n*

apply (*subst pos-divide-le-eq*)
apply (*rule mult-pos*)
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule eps-gt-0*)
apply (*erule xstar1a*)
apply (*subst mult-commute*)
apply (*subst pos-divide-le-eq* [*THEN sym*])
apply (*elim xstar4-aux*)
apply *assumption*+
done
have *xstar7*: !!*j n*. *?jstar* < *j* ==>
xstar j <= *x* / *real* (*n*::*nat*) ==>
x / *real n* <= (*1* + *lambda* ∗ *?eps*) ∗ *xstar j* ==>
x / (*?K* ∗ (*?K powr* (*real j*))) < *real n*
apply (*rule order-less-le-trans*)
prefer *2*
apply (*erule xstar6*)
apply (*force*, *force*)
apply (*rule divide-strict-left-mono*)
apply (*erule xstar2*)
apply (*insert x-gt-1*, *arith*)
apply (*rule mult-pos*)
apply (*rule mult-pos*)
apply *force*
apply *force*
apply (*rule mult-pos*)
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule eps-gt-0*)
apply (*erule xstar1a*)
done
have *xstar8*: !!*i j n*. *?jstar* < *i* ==> *i* < *j* ==>
xstar i <= *x* / *real* (*n*::*nat*) ==>
x / *real n* <= (*1* + *lambda* ∗ *?eps*) ∗ *xstar i* ==>
xstar j <= *x* / *real* (*n*::*nat*) ==> *False*
apply (*subgoal-tac real n* < *x* / (*?K powr* (*real j*)))
apply (*subgoal-tac x* / (*?K* ∗ (*?K powr* (*real i*))) < *real n*)
apply (*subgoal-tac x* / (*?K powr* (*real j*)) <=
x / (*?K* ∗ (*?K powr* (*real i*))))
apply *arith*

```
    apply (subgoal-tac ?K * (?K powr (real i)) = ?K powr (real (i + 1)))
    apply (erule ssubst)
    apply (rule real-pos-div-le-mono)
    apply (rule powr-gt-zero)
    apply (rule powr-mono)
    apply arith
    apply (insert K-gt-1, simp)
    apply (insert x-gt-1, arith)
    apply (simp add: powr-add)
    apply (rule xstar7)
    apply force
    apply force
    apply force
    apply (rule xstar5)
    apply force
    apply force
    done

  let ?S = UN j:{?jstar + 1..?jhat}.
      {n::nat. xstar j <= x / real n &
            x / real n <= (1 + lambda * ?eps) * xstar j}
  show EX S<={1..natfloor x}.
      (ALL n:S. abs (R (x / real n)) <= alpha / c2 * (x / real n)) &
        ?C * alpha ^ 2 * ln x ^ 2
         <= (∑ n:S. ln (real n) / real n)
proof
  show ?S <= {1..natfloor x} &
    (ALL n:?S. abs (R (x / real n)) <= alpha / c2 * (x / real n)) &
    ?C * alpha ^ 2 * ln x ^ 2 <= (∑ n:?S. ln (real n) / real n)
proof
  show ?S <= {1..natfloor x}
    apply clarsimp
    apply (rule conjI)
    apply (subgoal-tac 0 < real x)
    apply arith
    apply (rule xstar4-aux)
    prefer 2
    apply assumption
    apply force
    apply (rule real-le-natfloor)
    apply (rule order-trans)
    apply (rule order-less-imp-le)
    apply (rule xstar5)
    prefer 2
    apply assumption
```

apply *force*
apply (*subst pos-divide-le-eq*)
apply *force*
apply (*subgoal-tac ?x * 1 <= ?x * exp (c1 / (alpha / c2)) powr real j*)
apply *simp*
apply (*rule mult-left-mono*)
apply (*rule ge-one-powr-ge-zero*)
apply (*subgoal-tac exp 0 <= exp (c1 / ?eps)*)
apply *simp*
apply (*subst exp-le-cancel-iff*)
apply (*rule real-ge-zero-div-gt-zero*)
apply (*insert c1-gt-1, arith*)
apply (*rule eps-gt-0*)
apply *simp*
apply (*insert x-gt-1, arith*)
done
next show (*ALL n:?S. abs (R (x / real n)) <= alpha / c2 * (x / real n)*) &
*?C * alpha ^ 2 * ln x ^ 2 <= (∑ n:?S. ln (real n) / real n)*
proof
show (*ALL n:?S. abs (R (x / real n)) <= alpha / c2 * (x / real n)*)
apply *clarify*
apply (*rule order-less-imp-le*)
apply (*rule xstar0*)
prefer *2*
apply *assumption*
apply *force*
apply *assumption*
done
next show *?C * alpha ^ 2 * ln x ^ 2 <= (∑ n:?S. ln (real n) / real n)*
proof −
have (*∑ n: ?S. ln (real n) / real n*) =
(*∑ j:{?jstar + 1..?jhat}.*
(*∑ n::nat | xstar j <= x / real n &*
*x / real n <= (1 + lambda * ?eps) * xstar j.*
ln (real n) / real n))
apply (*rule setsum-UN-disjoint*)
apply *simp*
apply *clarsimp*
apply (*rule bounded-nat-set-is-finite*)
apply *clarsimp*
apply (*subgoal-tac i < natfloor (x / xstar j) + 1*)
apply *assumption*
apply (*subgoal-tac i <= natfloor (x / xstar j)*)
apply *arith*
apply (*rule real-le-natfloor*)

apply (*rule xstar4*)
apply *force*
apply *force*
apply (*clarsimp simp add*: *Int-def*)
apply (*subgoal-tac ja < j | j < ja*)
apply (*erule disjE*)
apply (*rule xstar8*)
prefer *2*
apply *assumption*
apply *force*
apply *force*
apply *force*
apply *force*
apply (*rule xstar8*)
prefer *2*
apply *assumption*
apply *auto*
done
also have ($\sum$ *j*:{*?jstar + 1..?jhat*}.
(($\sum$ *n*::*nat* | *xstar j <= x / real n &*
*x / real n <= (1 + lambda * ?eps) * xstar j*.
ln (x / xstar j) / (x / xstar j)))) <= ... (**is** *?term1 <= ?temp*)
apply (*rule setsum-le-cong*)
apply (*rule setsum-le-cong*)
apply (*rule ln-x-over-x-mono*)
apply *clarsimp*
apply (*rule order-trans*)
prefer *2*
apply (*rule order-less-imp-le*)
apply (*rule xstar7*)
prefer *2*
apply *force*
apply *force*
apply *force*
apply (*rule order-trans*)
apply (*subgoal-tac exp 1 <= ?x powr (1 / 2)*)
apply *assumption*
apply (*subst ln-le-cancel-iff* [*THEN sym*])
apply *simp*
apply *simp*
apply (*simp add*: *powr-def*)
apply (*subst exp-le-cancel-iff* [*THEN sym*])
apply (*subgoal-tac exp* (*ln ?x*) = *?x*)
apply (*erule ssubst*)
apply (*rule order-less-imp-le*)

apply (*rule x-gt-exp2*)
apply (*subst exp-ln-iff*)
apply (*insert x-gt-1*, *arith*)
apply (*subst pos-le-divide-eq*)
apply (*rule mult-pos*)
apply (*force*, *force*)
apply (*subst mult-commute*)
apply (*subst pos-le-divide-eq* [*THEN sym*])
apply *force*
apply (*subgoal-tac ?x / ?x powr (1 / 2) = ?x powr (1 / 2)*)
apply (*erule ssubst*)
apply (*rule order-trans*)
prefer *2*
apply (*rule jhat*)
apply (*subgoal-tac x <= ?t*)
apply *assumption*
apply *force*
apply (*simp add*: *powr-add*)
apply (*subst nonzero-divide-eq-eq*)
apply *simp*
apply (*simp add*: *powr-add* [*THEN sym*])
apply *clarsimp*
apply (*rule xstar4*)
apply *auto*
done
also have *?term1 = (∑ j:{?jstar + 1..?jhat}. real (card*
{n::nat. xstar j <= x / real n &
*x / real n <= (1 + lambda * ?eps) * xstar j}) **
(ln (x / xstar j) / (x / xstar j)))
apply (*rule setsum-cong2*)
apply (*subst setsum-constant*)
apply (*rule bounded-nat-set-is-finite*)
apply *clarsimp*
apply (*subgoal-tac i <= natfloor (?x / xstar x)*)
prefer *2*
apply (*rule real-le-natfloor*)
apply (*rule xstar4*)
apply *force*
apply *arith*
apply (*subgoal-tac i < Suc (natfloor (?x / xstar x))*)
apply *assumption*
apply (*erule le-imp-less-Suc*)
apply (*subst real-eq-of-nat* [*THEN sym*])
apply (*rule refl*)
done

also have ($\sum$ *j*:{*?jstar* + *1*..*?jhat*}. *real* (*card*
{*natfloor* (*x* / ((*1* + *lambda* ∗ *?eps*) ∗ *xstar j*))+*1*..
natfloor (*x* / *xstar j*)}) ∗
(*ln* (*x* / *xstar j*) / (*x* / *xstar j*))) <= ...
(**is** *?term2* <= *?temp*)
apply (*rule setsum-le-cong*)
apply (*rule mult-right-mono*)
apply (*rule le-imp-real-of-nat-le*)
apply (*rule card-mono*)
apply (*rule bounded-nat-set-is-finite*)
apply *clarsimp*
apply (*subgoal-tac i* <= *natfloor* (*?x* / *xstar x*))
prefer *2*
apply (*rule real-le-natfloor*)
apply (*rule xstar4*)
apply *force*
apply *arith*
apply (*subgoal-tac i* < *Suc* (*natfloor* (*?x* / *xstar x*)))
apply *assumption*
apply (*erule le-imp-less-Suc*)
apply *auto*
apply (*subst pos-le-divide-eq*)
apply *simp*
apply (*subst mult-commute*)
apply (*subst pos-le-divide-eq* [*THEN sym*])
apply (*rule xstar1a*)
apply *force*
apply (*rule nat-le-natfloor*)
apply (*rule order-less-imp-le*)
apply (*rule pos-div-pos*)
apply (*insert x-gt-1*, *arith*)
apply (*rule xstar1a*)
apply *force*
apply *assumption*
apply (*subst pos-divide-le-eq*)
apply *simp*
apply (*subst mult-commute*)
apply (*subst pos-divide-le-eq* [*THEN sym*])
apply (*rule mult-pos*)
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule alpha-gt-0*)

apply (*insert c2-gt-1*, *arith*)
apply (*rule xstar1a*)
apply *force*
apply (*rule order-less-imp-le*)
apply (*erule ge-natfloor-plus-one-imp-gt*)
apply (*rule real-ge-zero-div-gt-zero*)
apply (*rule nonneg-times-nonneg*)
apply (*rule ln-ge-zero*)
apply (*rule order-trans*)
prefer *2*
apply (*rule order-less-imp-le*)
apply (*rule xstar2c*)
apply (*force*, *force*)
apply (*subgoal-tac 1 powr (1 / 2) <= ?x powr (1 / 2)*)
apply (*simp add*: *powr-one-eq-one*)
apply (*rule power-mono2*)
apply (*force*, *force*)
apply (*rule order-less-imp-le*)
apply (*rule x-gt-1*)
apply (*rule order-less-imp-le*)
apply (*rule xstar1a*)
apply *force*
apply *force*
done
also (*order-trans2*) **have** *?term2* = (∑ *j*:{*?jstar + 1..?jhat*}.
(real (natfloor (x / xstar j)) −
*real (natfloor (x / ((1 + lambda * alpha / c2) * xstar j)))) *
(ln (x / xstar j) / (x / xstar j)))
apply (*rule setsum-cong2*)
apply (*simp add*: *natfloor-plus-one*)
apply (*subgoal-tac Suc (natfloor (?x / xstar x)) −*
*(natfloor (?x / ((1 + lambda * alpha / c2) * xstar x)) +*
1) = natfloor (?x / xstar x) −
*natfloor (?x / ((1 + lambda * alpha / c2) * xstar x))*)
apply (*erule ssubst*)
apply (*subst real-of-nat-diff*)
apply (*rule natfloor-mono*)
apply (*rule real-pos-div-le-mono*)
apply (*rule xstar1a*)
apply *force*
apply (*simp add*: *ring-eq-simps*)
apply (*rule order-less-imp-le*)
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)
apply (*rule alpha-gt-0*)

apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule xstar1a*)
apply (*simp add*: *mult-ac*)
apply *arith*
apply (*insert c2-gt-1*, *simp*)
apply (*insert x-gt-1*, *simp*)
apply *simp*
apply *arith*
done
also have ($\sum$ *j*:{*?jstar* + *1*..*?jhat*}.
((*x* / *xstar j* − *1*) −
(*x* / ((*1* + *lambda* ∗ *?eps*) ∗ *xstar j*))) ∗
(*ln* (*x* / *xstar j*) / (*x* / *xstar j*))) <= ...
apply (*rule setsum-le-cong*)
apply (*rule mult-right-mono*)
apply (*rename-tac xa*)
apply (*subgoal-tac real* (*natfloor* (*x* / ((*1* + *lambda* ∗ *alpha* / *c2*) ∗
xstar xa))) <= *x* / ((*1* + *lambda* ∗ *?eps*) ∗ *xstar xa*))
apply (*subgoal-tac x* / *xstar xa* <= *real* (*natfloor* (*x* / *xstar xa*)) + *1*)
apply *arith*
apply (*rule real-natfloor-plus-one-ge*)
apply (*rule nat-le-natfloor*)
apply (*rule order-less-imp-le*)
apply (*rule pos-div-pos*)
apply (*insert x-gt-1*, *simp*)
apply (*rule mult-pos*)
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule eps-gt-0*)
apply (*rule xstar1a*)
apply *force*
apply *simp*
apply (*rule real-ge-zero-div-gt-zero*)
apply (*rule ln-ge-zero*)
apply (*rule order-trans*)
prefer *2*
apply (*rule order-less-imp-le*)
apply (*rule xstar2c*)
apply (*force*, *force*)
apply (*rename-tac xa*)
apply (*subgoal-tac 1 powr* (*1* / *2*) <= *x powr* (*1* / *2*))
apply (*simp add*: *powr-one-eq-one*)

apply (*rule power-mono2*)
apply (*force*, *force*)
apply (*rule order-less-imp-le*)
apply (*rule x-gt-1*)
apply (*rule pos-div-pos*)
apply *force*
apply (*rule xstar1a*)
apply *force*
done
also (*order-trans2*) **have** ($\sum$ *j*:{*?jstar + 1..?jhat*}. ((*x / xstar j − 1*) −
(*x / ((1 + lambda * ?eps) * xstar j))) *
(*ln (x / xstar j) / (x / xstar j))) =
($\sum$ *j*:{*?jstar + 1..?jhat*}.
((*1 − 1 / (1 + lambda * ?eps)) * (x / xstar j) − 1) *
(*ln (x / xstar j) / (x / xstar j)))
apply (*rule setsum-cong2*)
apply (*simp add*: *ring-eq-simps*)
done
also have ($\sum$ *j*:{*?jstar + 1..?jhat*}.
(*1 / 2*) * (*1 − 1 / (1 + lambda * ?eps)) * (x / xstar j) *
(*ln (x / xstar j) / (x / xstar j))) <= ...
apply (*rule setsum-le-cong*)
apply (*rule mult-right-mono*)
apply (*subst mult-assoc*)
apply (*subst times-divide-eq-left*)
apply (*subst pos-divide-le-eq*)
apply *simp*
apply (*subst mult-commute*)
apply (*subst right-diff-distrib*)
apply (*rename-tac xa*)
apply (*subgoal-tac 2 * 1 <=*
*2 * ((1 − 1 / (1 + lambda * (alpha / c2))) * (x / xstar xa)) −*
*1 * ((1 − 1 / (1 + lambda * (alpha / c2))) * (x / xstar xa))*)
apply (*simp only*: *compare-rls*)
apply (*subst left-diff-distrib* [*THEN sym*])
apply *simp*
apply (*subst times-divide-eq-right* [*THEN sym*])
apply (*subst mult-commute*)
apply (*subst pos-divide-le-eq* [*THEN sym*])
apply *simp*
apply (*subst pos-divide-less-eq*)
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)

apply (*rule lambda-gt-0*)
apply (*rule alpha-gt-0*)
apply (*insert c2-gt-1*, *arith*)
apply *simp*
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule alpha-gt-0*)
apply (*insert c2-gt-1*, *arith*)
apply (*rule order-trans*)
prefer *2*
apply (*rule order-less-imp-le*)
apply (*rule xstar2c*)
apply *force*
apply *force*
apply (*subgoal-tac 2 / (1 − 1 / (1 + lambda ∗ alpha / c2)) =*
((2 / (1 − 1 / (1 + lambda ∗ alpha / c2))) powr 2) powr (1 / 2))
apply (*erule ssubst*)
apply (*rule power-mono2*)
apply *force*
apply *force*
apply (*subst times-divide-eq-right* [*THEN sym*])
apply (*rule x-ge-blah*)
apply (*simp add*: *powr-powr*)
apply (*rule sym*)
apply (*subst powr-one-gt-zero-iff*)
apply (*rule pos-div-pos*)
apply *force*
apply *simp*
apply (*subst pos-divide-less-eq*)
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule alpha-gt-0*)
apply (*insert c2-gt-1*, *arith*)
apply *simp*
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule alpha-gt-0*)
apply (*insert c2-gt-1*, *arith*)
apply (*rule real-ge-zero-div-gt-zero*)
apply (*rule ln-ge-zero*)

apply (*rule order-trans*)
prefer *2*
apply (*rule order-less-imp-le*)
apply (*rule xstar2c*)
apply (*force*, *force*)
apply (*subgoal-tac 1 powr (1 / 2) <= ?x powr (1 / 2)*)
apply (*simp add*: *powr-one-eq-one*)
apply (*rule power-mono2*)
apply (*force*, *force*)
apply (*rule order-less-imp-le*)
apply (*rule x-gt-1*)
apply (*rule pos-div-pos*)
apply (*insert x-gt-1*, *arith*)
apply (*rule xstar1a*)
apply *auto*
done
also (*order-trans2*) **have** ($\sum$ *j*:{*?jstar + 1..?jhat*}.
*(1 / 2) * (1 - 1 / (1 + lambda * ?eps)) * (x / xstar j) *
(ln (x / xstar j) / (x / xstar j))) =
($\sum$ *j*:{*?jstar + 1..?jhat*}.
*(1 / 2) * (1 - 1 / (1 + lambda * ?eps)) * (ln (x / xstar j))*)
apply (*rule setsum-cong2*)
apply (*rename-tac xa*)
apply (*subgoal-tac x / xstar xa ~= 0*)
apply *simp*
apply (*subgoal-tac 0 < x / xstar xa*)
apply *arith*
apply (*rule pos-div-pos*)
apply (*insert x-gt-1*, *arith*)
apply (*rule xstar1a*)
apply *auto*
done
also have ($\sum$ *j*:{*?jstar + 1..?jhat*}.
*(1 / 2) * (1 - 1 / (1 + lambda * ?eps)) * (ln (x powr (1 / 2)))) <= ...*
apply (*rule setsum-le-cong*)
apply (*rule mult-left-mono*)
apply (*subst ln-le-cancel-iff*)
apply *force*
apply (*rule pos-div-pos*)
apply (*insert x-gt-1*, *arith*)
apply (*rule xstar1a*)
apply *force*
apply (*rule order-less-imp-le*)
apply (*rule xstar2c*)
apply *force*

apply *force*
apply (*rule nonneg-times-nonneg*)
apply *force*
apply *simp*
apply (*subst pos-divide-le-eq*)
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule alpha-gt-0*)
apply (*insert c2-gt-1*, *arith*)
apply *simp*
apply (*rule order-less-imp-le*)
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule alpha-gt-0*)
apply (*insert c2-gt-1*, *arith*)
done
also (*order-trans2*) **have** ($\sum$ *j*:{*?jstar* + *1*..*?jhat*}.
(*1* / *2*) * (*1* − *1* / (*1* + *lambda* * *?eps*)) * (*ln* (*x powr* (*1* / *2*)))) =
($\sum$ *j*:{*?jstar* + *1*..*?jhat*}.
(*1* / *4*) * (*1* − *1* / (*1* + *lambda* * *?eps*)) * (*ln x*))
apply (*rule setsum-cong2*)
apply (*subst ln-pwr*)
apply (*insert x-gt-1*, *arith*)
apply *force*
apply *simp*
done
also have ... = (*1* / *4*) * (*1* − *1* / (*1* + *lambda* * *?eps*)) * (*ln x*) *
real (*card*({*?jstar* + *1*..*?jhat*}))
apply (*subst setsum-constant*)
apply *force*
apply (*subst real-eq-of-nat* [*THEN sym*])
apply (*rule mult-commute*)
done
also have ... = (*1* / *4*) * (*1* − *1* / (*1* + *lambda* * *?eps*)) * (*ln x*) *
(*real* (*natfloor* (*ln x* / (*2* * *ln ?K*))) −
real (*natfloor* (*ln x1* / *ln ?K*)) − *2*)
apply (*rule arg-cong*) **back**
apply *simp*
apply (*subgoal-tac Suc* (*natfloor* (*ln x* * *alpha* / (*2* * (*c1* * *c2*))) − *1*)
= *natfloor* (*ln x* * *alpha* / (*2* * (*c1* * *c2*))))
apply (*erule ssubst*)

apply (*subst real-of-nat-diff*)
apply (*subst add-assoc*)
apply (*subst natfloor-add* [*THEN sym*])
apply (*rule real-ge-zero-div-gt-zero*)
apply (*rule nonneg-times-nonneg*)
apply (*rule ln-ge-zero*)
apply (*rule x1-ge-1*)
apply (*rule order-less-imp-le*, *rule alpha-gt-0*)
apply (*rule mult-pos*)
apply (*insert c1-gt-1*, *arith*)
apply (*insert c2-gt-1*, *arith*)
apply (*rule natfloor-mono*)
apply *simp*
apply (*subst pos-le-divide-eq*)
apply (*rule mult-pos*)
apply *force*
apply (*rule mult-pos*)
apply *arith*
apply *arith*
apply (*simp add*: *ring-eq-simps*)
apply (*subgoal-tac alpha* $*$ *ln x* $=$ *ln x* $*$ *alpha*)
apply (*erule ssubst*)
apply (*subst pos-divide-le-eq* [*THEN sym*])
apply (*rule alpha-gt-0*)
apply (*subst exp-le-cancel-iff* [*THEN sym*])
apply (*subgoal-tac exp* (*ln x*) $=$ *x*)
apply (*erule ssubst*)
apply (*rule x-ge-blah2*)
apply (*subst exp-ln-iff*)
apply (*insert x-gt-1*, *arith*)
apply (*rule mult-commute*)
apply *arith*
apply (*subgoal-tac 1* $<=$ *natfloor* (*ln x* $*$ *alpha* / (*2* $*$ (*c1* $*$ *c2*))))
apply *arith*
apply (*subgoal-tac 1* $=$ *natfloor 1*)
apply (*erule ssubst*)
apply (*rule natfloor-mono*)
apply (*subst pos-le-divide-eq*)
apply *simp*
apply (*rule mult-pos*)
apply (*insert c1-gt-1*, *arith*)
apply (*insert c2-gt-1*, *arith*)
apply *simp*
apply (*subst pos-divide-le-eq* [*THEN sym*])
apply (*rule alpha-gt-0*)

apply (*subst exp-le-cancel-iff* [*THEN sym*])
apply (*subgoal-tac exp* (*ln x*) = *x*)
apply (*erule ssubst*)
apply (*rule x-ge-blah3*)
apply (*subst exp-ln-iff*)
apply (*insert x-gt-1*, *arith*)
apply *simp*
done
also have (*1 / 4*) * (*lambda * ?eps / 2*) * (*ln x*) *
((*1 / 4*) * (*ln x / ln ?K*)) <= ...
apply (*rule mult-mono*)
apply (*rule mult-right-mono*)
apply (*rule mult-left-mono*)
apply (*subgoal-tac 1 - 1 / (1 + lambda * ?eps*) = *lambda * ?eps /*
(*1 + lambda * ?eps*))
apply (*erule ssubst*)
apply (*rule divide-left-mono*)
apply (*subgoal-tac lambda * ?eps <= 1 * 1*)
apply *simp*
apply (*rule mult-mono*)
apply (*rule order-less-imp-le*, *rule lambda-le-1*)
apply (*rule order-less-imp-le*, *rule eps-lt-1*)
apply *force*
apply (*rule order-less-imp-le*, *rule eps-gt-0*)
apply (*rule order-less-imp-le*)
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule eps-gt-0*)
apply (*rule mult-pos*)
apply *force*
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule eps-gt-0*)
apply (*rule lemma-divide-rearrange*)
apply (*rule less-imp-neq* [*THEN not-sym*])
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule eps-gt-0*)
apply *simp*
apply *force*
apply (*rule ln-ge-zero*)

apply (*insert x-gt-1*, *arith*)
apply (*rule order-trans*)
apply (*subgoal-tac 1 / 4 * (ln x / ln (exp (c1 / (alpha / c2)))) <=*
*((ln x / (2 * ln ?K)) − (ln x1 / ln ?K) − 3)*)
apply *assumption*
prefer *2*
apply (*subgoal-tac (ln x / (2 * ln ?K)) <=*
*real (natfloor (ln x / (2 * ln ?K))) + 1*)
apply (*subgoal-tac real (natfloor (ln x1 / ln ?K)) <=*
(ln x1 / ln ?K))
apply *arith*
apply (*rule real-natfloor-le*)
apply *simp*
apply (*rule real-ge-zero-div-gt-zero*)
apply (*rule nonneg-times-nonneg*)
apply (*rule ln-ge-zero*)
apply (*rule x1-ge-1*)
apply (*rule order-less-imp-le*)
apply (*rule alpha-gt-0*)
apply (*rule mult-pos*)
apply (*insert c1-gt-1*, *arith*)
apply (*insert c2-gt-1*, *arith*)
apply (*rule real-natfloor-plus-one-ge*)
prefer *2*
apply (*rule nonneg-times-nonneg*)
apply (*rule nonneg-times-nonneg*)
apply *force*
apply *simp*
apply (*subst pos-divide-le-eq*)
apply (*rule pos-plus-pos*)
apply *force*
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule alpha-gt-0*)
apply (*insert c2-gt-1*, *arith*)
apply *simp*
apply (*rule order-less-imp-le*)
apply (*rule pos-div-pos*)
apply (*rule mult-pos*)
apply (*rule lambda-gt-0*)
apply (*rule alpha-gt-0*)
apply *arith*
apply (*rule ln-ge-zero*)
apply (*rule order-less-imp-le*)

apply (*rule x-gt-1*)
apply (*simp only*: *diff-minus*)
apply (*subgoal-tac 1 / 4 = 1 / 2 + - (1 / 8) + - (1 / 8)*)
apply (*erule ssubst*)
apply (*subst left-distrib*)
apply (*subst left-distrib*)
apply (*rule add-mono*)
apply (*rule add-mono*)
apply *simp*
apply (*simp add*: *compare-rls*)
apply (*subst times-divide-eq-right* [*THEN sym*])
apply (*subst pos-divide-le-eq* [*THEN sym*])
apply (*rule pos-div-pos*)
apply (*rule alpha-gt-0*)
apply (*rule mult-pos*)
apply *force*
apply (*rule mult-pos*)
apply *arith*
apply *arith*
apply (*subgoal-tac alpha ~= 0*)
apply *simp*
apply (*subst mult-commute*)
apply (*subst ln-pwr* [*THEN sym*])
apply (*insert x1-ge-1*, *arith*)
apply *force*
apply (*subst ln-le-cancel-iff*)
apply *force*
apply (*insert x-gt-1*, *force*)
apply (*rule x-ge-blah4*)
apply (*rule less-imp-neq* [*THEN not-sym*])
apply (*rule alpha-gt-0*)
apply (*simp add*: *compare-rls*)
apply (*subst times-divide-eq-right* [*THEN sym*])
apply (*subst pos-divide-le-eq* [*THEN sym*])
apply (*rule pos-div-pos*)
apply (*rule alpha-gt-0*)
apply (*rule mult-pos*)
apply *force*
apply (*rule mult-pos*)
apply *arith*
apply *arith*
apply (*subst exp-le-cancel-iff* [*THEN sym*])
apply (*subgoal-tac exp* (*ln x*) = *x*)
apply (*erule ssubst*)
apply (*rule x-ge-blah5*)

```
    apply (subst exp-ln-iff)
    apply arith
    apply simp
    apply simp
    apply (rule real-ge-zero-div-gt-zero)
    apply (rule nonneg-times-nonneg)
    apply (rule ln-ge-zero)
    apply arith
    apply (rule order-less-imp-le)
    apply (rule alpha-gt-0)
    apply (rule mult-pos)
    apply force
    apply (rule mult-pos)
    apply arith
    apply arith
    done
  also (order-trans2) have (1 / 4) * (lambda * ?eps / 2) * (ln x) *
      ((1 / 4) * (ln x / ln ?K)) =
    (lambda * alpha^2 * (ln x)^2) / (32 * c1 * c2^2)
    by (simp add: mult-ac power2-eq-square)
  finally show lambda / (32 * c1 * c2 ^ 2) * alpha ^ 2 * ln x ^ 2 <=
      (∑ n:?S. ln (real n) / real n)
    by (simp add: ring-eq-simps)
qedqedqedqedqedqedqedqedqed

lemma PNT4: 1 < (c2::real) ==> EX C. 0 < C & (ALL alpha1 x1. 0 < alpha1
-->
    alpha1 < c2 --> (ALL x. x1 <= x --> abs (R x) <= alpha1 * x) -->
    (EX x2. (ALL x. x2 <= x --> abs (R x) <= (alpha1 - C * alpha1^3) *
x)))
proof -
  assume c2-gt-1: 1 < (c2::real)
  have EX C. 0 < C & (ALL alpha.
    0 < alpha --> alpha < c2 -->
      (EX xlarge. ALL x. xlarge <= x -->
        (EX S. S <= {1..natfloor x} &
          (ALL n:S. abs(R(x / real n)) <= (alpha / c2) * (x / real n)) &
            C * alpha^2 * (ln x)^2
              <= (∑ n:S. ln(real n) / (real n)))))
    apply (rule PNT4-aux1)
    apply (rule prems)
    done
  then obtain C where C-gt-0: 0 < C and C': (ALL alpha.
    0 < alpha --> alpha < c2 -->
      (EX xlarge. ALL x. xlarge <= x -->
```

```
      (EX S. S <= {1..natfloor x} &
        (ALL n:S. abs(R(x / real n)) <= (alpha / c2) * (x / real n)) &
          C * alpha^2 * (ln x)^2
            <= (∑ n:S. ln(real n) / (real n))))) by blast
  from PNT3 obtain G where G: ALL x x2 S eps alpha.
    0 < eps --> 0 < alpha --> alpha < c2 -->
      2 <= x2 --> exp 1 <= x2 --> x2 <= x -->
        (ALL x. x2 <= x --> abs(R x) <= alpha * x) -->
          S <= {1..natfloor x} -->
            (ALL n:S. abs(R(x / real n)) <= eps * (x / real n)) -->
        abs (R x)
          <= alpha * x -
            2 * (alpha - eps) * x *
              (∑n:S. ln (real n) / real n) / ln x ^ 2 +
                G * x * ln x2 / ln x..
  let ?C = (1 - 1 / c2) * C
  show ?thesis
proof
  show 0 < ?C & (ALL alpha1 x1. 0 < alpha1 -->
    alpha1 < c2 --> (ALL x. x1 <= x --> abs (R x) <= alpha1 * x) -->
    (EX x2. (ALL x. x2 <= x --> abs (R x) <= (alpha1 - ?C * alpha1^3) *
x)))
proof
  show 0 < ?C
    apply (rule mult-pos)
    apply simp
    apply (subst pos-divide-less-eq)
    apply (insert c2-gt-1, arith)
    apply simp
    apply (rule C-gt-0)
    done
  next show (ALL alpha1 x1. 0 < alpha1 -->
    alpha1 < c2 --> (ALL x. x1 <= x --> abs (R x) <= alpha1 * x) -->
    (EX x2. (ALL x. x2 <= x --> abs (R x) <= (alpha1 - ?C * alpha1^3) *
x)))
proof (clarify)
  fix alpha1::real
  fix x1::real
  assume alpha1-gt-0: 0 < alpha1
  assume alpha1-lt-c2: alpha1 < c2
  assume x1-good: ALL x. x1 <= x --> abs (R x) <= alpha1 * x
  have EX xlarge. ALL x. xlarge <= x -->
    (EX S. S <= {1..natfloor x} &
      (ALL n:S. abs(R(x / real n)) <= (alpha1 / c2) * (x / real n)) &
        C * alpha1^2 * (ln x)^2
```

```
        <= (∑ n:S. ln(real n) / (real n)))
    by (insert C' alpha1-gt-0 alpha1-lt-c2, blast)
  then obtain xlarge where xlarge: ALL x. xlarge <= x -->
    (EX S. S <= {1..natfloor x} &
      (ALL n:S. abs(R(x / real n)) <= (alpha1 / c2) * (x / real n)) &
        C * alpha1^2 * (ln x)^2
          <= (∑ n:S. ln(real n) / (real n)))..
  let ?x1' = max x1 (max 2 (exp 1))
  have x1prime-ge-x1: x1 <= ?x1' by arith
  have x1prime-ge-2: 2 <= ?x1' by arith
  have x1prime-ge-exp1: exp 1 <= ?x1' by arith
  let ?messy-term = exp (G * ln ?x1' /
          (C * ((1 - 1 / c2) * alpha1 ^ 3)))
  let ?x2 = max xlarge (max (max (max 2 (exp 1)) ?x1') ?messy-term)
  show EX x2. ALL x. x2 <= x --> abs (R x) <= (alpha1 - ?C * alpha1 ^
3) * x
proof
  show ALL x. ?x2 <= x --> abs (R x) <= (alpha1 - ?C * alpha1 ^ 3) * x
proof (clarify)
  fix x::real
  assume x-ge-x2: ?x2 <= x
  have x-ge-xlarge: xlarge <= x
    apply (rule order-trans)
    prefer 2
    apply (rule x-ge-x2)
    apply (rule le-maxI1)
    done
  have x2-ge-2: 2 <= ?x2
    by arith
  have x2-ge-exp1: exp 1 <= ?x2
    by arith
  have x2-ge-x1: x1 <= ?x2
    by arith
  have x2-ge-x1prime: ?x1' <= ?x2
    apply auto
    apply arith
    apply arith
    apply arith
    done
  have messy-term: ?messy-term <= ?x2
    apply arith
    done
  have EX S. S <= {1..natfloor x} &
      (ALL n:S. abs(R(x / real n)) <= (alpha1 / c2) * (x / real n)) &
        C * alpha1^2 * (ln x)^2
```

```
      <= (∑ n:S. ln(real n) / (real n))
  apply (insert xlarge)
  apply (drule-tac x = x in spec)
  apply (insert x-ge-xlarge)
  apply clarify
  done
then obtain S where S1: S <= {1..natfloor x} and
  S2: (ALL n:S. abs(R(x / real n)) <= (alpha1 / c2) * (x / real n)) and
  S3: C * alpha1^2 * (ln x)^2 <= (∑ n:S. ln(real n) / (real n))
    by blast
have abs (R x) <= alpha1 * x -
  2 * (alpha1 - alpha1 / c2) * x *
    (∑n:S. ln (real n) / real n) / ln x ^ 2 +
      G * x * ln ?x1' / ln x
  apply (insert G)
  apply (drule-tac x = x in spec)
  apply (drule-tac x = ?x1' in spec)
  apply (drule-tac x = S in spec)
  apply (drule-tac x = (alpha1 / c2) in spec)
  apply (drule-tac x = alpha1 in spec)
  apply (drule mp)
  apply (rule pos-div-pos)
  apply (rule alpha1-gt-0)
  apply (insert c2-gt-1, arith)
  apply (drule mp)
  apply (rule alpha1-gt-0)
  apply (drule mp)
  apply (rule alpha1-lt-c2)
  apply (drule mp)
  apply (rule x1prime-ge-2)
  apply (drule mp)
  apply (rule x1prime-ge-exp1)
  apply (drule mp)
  apply (rule order-trans)
  apply (rule x2-ge-x1prime)
  apply (rule x-ge-x2)
  apply (drule mp)
  apply (rule allI)
  apply (rename-tac xa)
  apply (rule impI)
  apply (insert x1-good)
  apply (drule-tac x = xa in spec)
  apply (drule mp)
  apply arith
  apply assumption
```

apply (*drule mp*)
apply (*rule S1*)
apply (*drule mp*)
apply (*rule S2*)
apply *assumption*
done
also have ... <= *alpha1* * *x* −
2 * (*alpha1* − *alpha1* / *c2*) * *x* *
(*C* * *alpha1* ^ *2* * *ln x* ^ *2*) / *ln x* ^ *2* +
G * *x* * *ln* (*?x1'*) / *ln x*
apply (*subst diff-minus*)+
apply (*rule add-right-mono*)
apply (*rule add-left-mono*)
apply (*rule le-imp-neg-le*)
apply (*rule real-div-pos-le-mono*)
apply (*rule mult-left-mono*)
apply (*rule S3*)
apply (*rule nonneg-times-nonneg*)
apply (*rule nonneg-times-nonneg*)
apply *force*
apply *simp*
apply (*subst pos-divide-le-eq*)
apply (*insert c2-gt-1*, *arith*)
apply (*subgoal-tac alpha1* * *1* <= *alpha1* * *c2*)
apply *simp*
apply (*rule mult-left-mono*)
apply (*erule order-less-imp-le*)
apply (*insert alpha1-gt-0*, *arith*)
apply (*insert x-ge-x2 x2-ge-2*, *arith*)
apply *force*
done
also have ... = *x* * (*alpha1* − *2* * *alpha1^3* * (*1* − *1* / *c2*) * *C* +
(*G* * *ln ?x1'* / *ln x*))
apply (*subgoal-tac ln x* ~= *0*)
apply (*simp add*: *ring-eq-simps power3-eq-cube power2-eq-square*
add-divide-distrib diff-divide-distrib)
apply (*rule less-imp-neq* [*THEN not-sym*])
apply (*rule ln-gt-zero*)
apply (*insert x-ge-x2 x2-ge-2*, *arith*)
done
also have ... <= *x* * (*alpha1* − *2* * *alpha1^3* * (*1* − *1* / *c2*) * *C* +
alpha1^3 * (*1* − *1* / *c2*) * *C*)
apply (*rule mult-left-mono*)
apply *simp*
apply (*subst pos-divide-le-eq*)

```
    apply (rule ln-gt-zero)
    apply (insert x-ge-x2 x2-ge-2, arith)
    apply (subst mult-commute)
    apply (subst pos-divide-le-eq [THEN sym])
    apply (rule mult-pos)
    apply (rule C-gt-0)
    apply (rule mult-pos)
    apply simp
    apply (subst pos-divide-less-eq)
    apply (insert c2-gt-1, arith)
    apply simp
    apply (subst power3-eq-cube)
    apply (rule mult-pos)
    apply (rule mult-pos)
    apply (rule alpha1-gt-0)+
    apply (subst exp-le-cancel-iff [THEN sym])
    apply (subgoal-tac exp (ln x) = x)
    apply (erule ssubst)
    apply (rule order-trans)
    apply (rule messy-term)
    apply (rule x-ge-x2)
    apply (subst exp-ln-iff)
    apply (insert x-ge-x2 x2-ge-2, arith)
    apply arith
    done
  also have ... = x * (alpha1 - alpha1 ^ 3 * (1 - 1 / c2) * C)
    by simp
  finally show abs (R x) <= (alpha1 - (1 - 1 / c2) * C * alpha1 ^ 3) * x
    by (simp add: ring-eq-simps)
qedqedqedqedqedqed

constdefs
  PNT-alpha0::real
  PNT-alpha0 == (SOME C. 0 < C & (ALL x. 0 <= x --> abs (R x) <= C
* x))

lemma PNT-alpha0: 0 < PNT-alpha0 &
    (ALL x. 0 <= x --> abs (R x) <= PNT-alpha0 * x)
  apply (unfold PNT-alpha0-def)
  apply (rule someI-ex)
  apply (rule PNT3-aux2a)
done

constdefs
  PNT-C::real
```

```
  PNT-C == SOME C. 0 < C & (ALL alpha1 x1. 0 < alpha1 -->
    alpha1 < max (PNT-alpha0 + 1) 2 -->
    (ALL x. x1 <= x --> abs (R x) <= alpha1 * x) -->
    (EX x2. (ALL x. x2 <= x --> abs (R x) <= (alpha1 - C * alpha1^3) *
x)))

lemma PNT-C: 0 < PNT-C & (ALL alpha1 x1. 0 < alpha1 -->
    alpha1 < max (PNT-alpha0 + 1) 2 -->
    (ALL x. x1 <= x --> abs (R x) <= alpha1 * x) -->
    (EX x2. (ALL x. x2 <= x --> abs (R x) <=
      (alpha1 - PNT-C * alpha1^3) * x)))
  apply (unfold PNT-C-def)
  apply (rule someI-ex)
  apply (rule PNT4)
  apply arith
done

consts
  PNT-alpha :: nat => real

primrec
  PNT-alpha-0: PNT-alpha 0 = PNT-alpha0
  PNT-alpha-Suc: PNT-alpha (Suc n) = PNT-alpha n - PNT-C * (PNT-alpha
n)^3

lemma PNT5-aux1: 0 <= PNT-alpha n & PNT-alpha (Suc n) <= PNT-alpha n
&
    PNT-alpha n < max (PNT-alpha0 + 1) 2 &
    (EX x2. (ALL x. x2 <= x --> abs (R x) <=
      PNT-alpha n * x))
  apply (induct-tac n)
  apply (rule conjI)
  apply (rule order-less-imp-le)
  apply (force simp add: PNT-alpha0)
  apply (rule conjI)
  apply simp
  apply (rule nonneg-times-nonneg)
  apply (rule order-less-imp-le)
  apply (force simp add: PNT-C)
  apply (subgoal-tac 0 <= PNT-alpha0)
  apply (subst power3-eq-cube)
  apply (rule nonneg-times-nonneg)+
  apply assumption+
  apply (rule order-less-imp-le)
  apply (force simp add: PNT-alpha0)
```

apply (*rule conjI*)
apply (*rule order-less-le-trans*)
prefer *2*
apply (*rule le-maxI1*)
apply *simp*
apply (*rule-tac x = 1* **in** *exI*)
apply *simp*
apply (*force simp add*: *PNT-alpha0*)
apply (*subgoal-tac* (*EX x2*. *ALL x*. *x2* <= *x* −−>
abs (*R x*) <= *PNT-alpha* (*Suc n*) ∗ *x*))
apply (*subgoal-tac 0* <= *PNT-alpha* (*Suc n*))
apply (*rule conjI*)
apply *assumption*
apply (*rule conjI*)
apply (*subst PNT-alpha-Suc*)**back**
apply (*simp del*: *PNT-alpha-Suc*)
apply (*rule nonneg-times-nonneg*)
apply (*rule order-less-imp-le*)
apply (*force simp add*: *PNT-C*)
apply (*subst power3-eq-cube*)
apply (*rule nonneg-times-nonneg*)+
apply *assumption*+
apply (*rule conjI*)
apply (*rule order-le-less-trans*)
apply *clarify*
apply *assumption*
apply *force*
apply *assumption*
apply *clarify*
apply (*drule-tac x = max x2 1* **in** *spec*)
apply (*drule mp*)
apply *arith*
apply (*rule order-trans*)
apply (*subgoal-tac 0* <= *abs* (*R* (*max x2 1*)) / (*max x2 1*))
apply *assumption***back**
apply (*rule real-ge-zero-div-gt-zero*)
apply *force*
apply *arith*
apply (*subst pos-divide-le-eq*)
apply *arith*
apply *assumption*
apply *simp*
apply (*case-tac PNT-alpha n = 0*)
apply *clarify*
apply (*simp add*: *power3-eq-cube*)

apply (*rule-tac x = x2* **in** *exI*)
apply *assumption*
apply (*insert PNT-C*)
apply *clarify*
apply (*drule-tac x = PNT-alpha n* **in** *spec*)
apply (*drule-tac x = x2* **in** *spec*)**back**
apply (*subgoal-tac 0 < PNT-alpha n*)
apply *clarify*
apply *arith*
done

lemma *PNT5-aux2*: *convergent PNT-alpha*
apply (*rule Bseq-monoseq-convergent*)
apply (*unfold Bseq-def*)
apply (*insert PNT5-aux1*)
apply (*rule-tac x = max (PNT-alpha0 + 1) 2* **in** *exI*)
apply (*rule conjI*)
apply *arith*
apply (*rule allI*)
apply (*subst abs-nonneg*)
apply *force*
apply (*rule order-less-imp-le*)
apply *force*
apply (*subst monoseq-Suc*)
apply (*rule disjI2*)
apply *force*
done

lemma *PNT5-aux3*: *EX L. PNT-alpha ----> L*
apply (*insert PNT5-aux2*)
apply (*unfold convergent-def*)
apply *assumption*
done

lemma *PNT5-aux4*: *PNT-alpha ----> L ==> L = 0*
proof −
assume *a*: *PNT-alpha ----> L*
then have *b*: (%*n*. *PNT-alpha* (*Suc n*)) ----> *L*
by (*rule LIMSEQ-Suc*)
have *c*: (%*n*. *PNT-alpha n* −
PNT-C ∗ *PNT-alpha n* ∗ *PNT-alpha n* ∗ *PNT-alpha n*) ---->
L − *PNT-C* ∗ *L* ∗ *L* ∗ *L*
apply (*rule LIMSEQ-diff*)
apply (*rule a*)
apply (*rule LIMSEQ-mult*)+

```
    apply (rule LIMSEQ-const)
    apply (rule a)+
    done
  also have (%n. PNT-alpha n −
    PNT-C ∗ PNT-alpha n ∗ PNT-alpha n ∗ PNT-alpha n) =
    (%n. PNT-alpha (Suc n))
    by (rule ext, simp add: power3-eq-cube mult-ac)
  finally have c: (%n. PNT-alpha (Suc n)) −−−−> L − PNT-C ∗ L ∗ L ∗ L.
  then have L − PNT-C ∗ L ∗ L ∗ L = L
    apply (rule LIMSEQ-unique)
    apply (rule b)
    done
  then show L = 0
    apply (subgoal-tac PNT-C ~= 0)
    apply simp
    apply (rule less-imp-neq [THEN not-sym])
    apply (insert PNT-C, force)
    done
qed

lemma PNT5-aux5: PNT-alpha −−−−> 0
  by (insert PNT5-aux3 PNT5-aux4, auto)

lemma PNT5: ALL eps. (0 < eps −−>
    (EX z. ALL x. (z <= x −−> abs (R x) < eps ∗ x)))
proof (clarify)
  fix eps::real
  assume 0 < eps
  have EX n. PNT-alpha n < eps
    apply (insert PNT5-aux5)
    apply (unfold LIMSEQ-def)
    apply (drule-tac x = eps in spec)
    apply (insert prems, clarify)
    apply (rule-tac x = no in exI)
    apply (drule-tac x = no in spec)
    apply (subgoal-tac abs(PNT-alpha no) = PNT-alpha no)
    apply auto
    apply (rule abs-nonneg)
    apply (insert PNT5-aux1, force)
    done
  then obtain n where n: PNT-alpha n < eps..
  have EX z. (ALL x. z <= x −−> abs (R x) <=
     PNT-alpha n ∗ x)
    by (insert PNT5-aux1, force)
  thus EX z. ALL x. (z <= x −−> abs (R x) < eps ∗ x)
```

```
    apply clarify
    apply (rule-tac x = max z 1 in exI)
    apply clarify
    apply (drule-tac x = x in spec)
    apply (subgoal-tac z <= x)
    apply clarify
    apply (erule order-le-less-trans)
    apply (rule mult-strict-right-mono)
    apply (rule n)
    apply arith
    apply arith
    done
qed

lemma PrimeNumberTheorem1: (%x. psi x / (real x)) ----> 1
  apply (rule R-bound-imp-PNT)
  apply (rule PNT5)
done

lemma PrimeNumberTheorem2: (%x. theta x / (real x)) ----> 1
  apply (rule psi-theta-lim4)
  apply (rule PrimeNumberTheorem1)
done

lemma PrimeNumberTheorem: (%x. pi x * ln (real x) / (real x)) ----> 1
  apply (rule theta-limit-imp-pi-limit)
  apply (rule PrimeNumberTheorem2)
done

end
```