Landau Symbols

Manuel Eberl

July 15, 2015

Contents

1 Auxiliary lemmas 1
  1.1 Filters 2
  1.2 Miscellaneous 3
  1.3 Real powers 3

2 Definition of Landau symbols 7
  2.1 Eventual non-negativity/non-zeroness 7
  2.2 Definition of Landau symbols 9
  2.3 Landau symbols and limits 29
  2.4 Rewriting Landau symbols 42

3 Sorting and grouping factors 45

4 Decision procedure for real functions 52
  4.1 Decision procedure 53
  4.2 Reification 65

5 Simplification procedures 71
  5.1 Simplification under Landau symbols 72
  5.2 Simproc setup 72
  5.3 Tests 74
    5.3.1 Product simplification tests 74
    5.3.2 Real product decision procue tests 74
    5.3.3 Sum cancelling tests 75

1 Auxiliary lemmas

theory Landau-Library
imports Complex-Main
begin
1.1 Filters

lemma filterlim-abs-real: filterlim (abs::real ⇒ real) at-top at-top
proof (subst filterlim-cong[OF refl refl])
  from eventually-ge-at-top[of 0::real] show eventually (λx::real. |x| = x) at-top
    by eventually-elim simp
qed (simp-all add: filterlim-ident)

lemma eventually-False-at-top-linorder [simp]:
  eventually (λ::-::linorder. False) at-top ←→ False
unfolding eventually-at-top-linorder by force

lemma eventually-not-equal: eventually (λx::'a::linordered-semidom. x ≠ a) at-top
using eventually-ge-at-top[of a+1] by eventually-elim (insert less-add-one[of a], auto)

lemma eventually-subst':
  eventually (λx. f x = g x) F =⇒ eventually (λx. P x (f x)) F = eventually (λx. P x (g x)) F
  by (rule eventually-subst, erule eventually-rev-mp) simp

lemma eventually-nat-real:
  assumes eventually P (at-top :: real filter)
  shows eventually P (at-top :: nat filter)
using assms filterlim-real-sequentially
unfolding filterlim-def le-filter-def eventually-filtermap by auto

lemma filterlim-cong':
  assumes filterlim f F G
  assumes eventually (λx. f x = g x) G
  shows filterlim g F G
using assms by (subst filterlim-cong[OF refl refl, of - f]) (auto elim: eventually-elim1)

lemma tendsto-cong:
  assumes eventually (λx. f x = g x) F
  shows (f −−−> (x::real)) F ←→ (g −−−> x) F
by (subst (1 2) tendsto-iff, subst eventually-subst[OF assms]) (rule refl)

lemma eventually-ln-at-top: eventually (λx. P (ln x :: real)) at-top = eventually P at-top
proof
  fix P assume eventually (λx. P (ln x :: real)) at-top
  then obtain x0 where x0: ∀x. x ≥ x0 =⇒ P (ln x)
    by (subst (asm) eventually-at-top-linorder) auto
  { fix x assume x ≥ ln (max 1 x0)
    hence exp x ≥ max 1 x0 by (subst (2) exp-ln[symmetric], simp, subst exp-le-cancel-iff)
    hence exp x ≥ x0 by simp
    from x0[OF this] have P x by simp
  }

thus eventually $P$ at-top by \(\text{subst eventually-at-top-linorder}\) blast

next

fix $P :: \text{real} \Rightarrow \text{bool}$ assume eventually $P$ at-top

then obtain $x0$ where $x0 \land x \geq x0 \implies P x$

by \(\text{(subst (asm) eventually-at-top-linorder)}\) auto

{ 
    fix $x$ assume $x \geq \exp x0$

    hence $\ln x \geq x0$ by \((\text{subst \ln-exp[symmetric], subst \ln-le-cancel-iff})\)

    (simp-all add: less-le-trans[OF exp-gt-zero])

    from $x0$[OF this] have $P \ (\ln x)$.
}

thus eventually $(\lambda x. \ P \ (\ln x))$ at-top

by \(\text{(subst eventually-at-top-linorder)}\) blast

qed

lemma \text{filtermap-ln-at-top}: \text{filtermap} \ (\ln :: \text{real} \Rightarrow \text{real}) \text{at-top} = \text{at-top}

by \(\text{(simp add: filter-eq-iff eventually-filtermap eventually-ln-at-top)}\)

lemma \text{eventually-ln-not-equal}: eventually $(\lambda x :: \text{real}. \ \ln x \neq a)$ at-top

by \(\text{(subst eventually-ln-at-top)}\) (rule eventually-not-equal)

1.2 Miscellaneous

lemma \text{ln-mono}: $0 < x \implies 0 < y \implies x \leq y \implies \ln (x :: \text{real}) \leq \ln y$

by \(\text{(subst \ln-le-cancel-iff)}\) simp-all

lemma \text{ln-mono-strict}: $0 < x \implies 0 < y \implies x < y \implies \ln (x :: \text{real}) < \ln y$

by \(\text{(subst \ln-less-cancel-iff)}\) simp-all

lemma \text{listprod-pos}: $(\forall x :: \text{-linordered-semidom}. \ x \in \text{set} \ xs \implies x > 0) \implies \text{listprod} \ xs > 0$

by \(\text{(induction \ xs)}\) auto

lemma \text{(in monoid-mult)} \text{fold-plus-listprod-rev}:

\text{fold times \ xs} = \text{times} (\text{listprod} (\text{rev} \ xs))

proof

fix $x$

have \text{fold times \ xs} $x = \text{listprod} (\text{rev} \ xs \ @ \ [x])$

by \(\text{(simp add: foldr-conv-fold listprod.eq-foldr)}\)

also have \ldots = \text{listprod} (\text{rev} \ xs) * $x$

by simp

finally show \text{fold times \ xs} $x = \text{listprod} (\text{rev} \ xs) * x$.

qed

1.3 Real powers

lemma \text{powr-realpow-eventually}:

\text{assumes \ filterlim \ f \ at-top \ F}

\text{shows \ eventually} \ (\lambda x. \ f x \ \text{powr} \ (\text{real} \ n) = f \ x ^ n) \ F

proof—
from assms have eventually \((\lambda x. f x > 0)\) \(F\) using filterlim-at-top-dense by blast
thus \(?thesis\) by eventually-elim (simp add: powr-realpow)
qed

lemma zero-powr [simp]: \((0::real) powr x = 0\)
unfolding powr-def by simp

lemma powr-negD: \((a::real) powr b \leq 0 \Longrightarrow a = 0\)
unfolding powr-def by (simp split: split-if-asm)

lemma inverse-powr [simp]:
assumes \((x::real) \geq 0\)
shows \(inverse x powr y = inverse (x powr y)\)
proof (cases \(x > 0\))
assume \(x\): \(x > 0\)
from \(x\) have \(inverse x powr a \leq inverse x powr b\) using assms
by (simp add: powr-mono)
also have \(ln (inverse x) = -ln x\) by (simp add: x ln-inverse)
also have \(exp (y * -ln x) = inverse (exp (y * ln x))\) by (simp add: exp-minus)
also from \(x\) have \(exp (y * ln x) = x powr y\) by (simp add: powr-def)
finally show \(?thesis\)
qed (insert assms, simp)

lemma powr-mono':
assumes \((x::real) > 0\) \(x \leq 1\) \(a \leq b\)
shows \(x powr b \leq x powr a\)
proof—
have \(inverse x powr a \leq inverse x powr b\) using assms
by (intro powr-mono') (simp-all add: field-simps)

lemma powr-less-mono':
assumes \((x::real) > 0\) \(x < 1\) \(a < b\)
shows \(x powr b < x powr a\)
proof—
have \(inverse x powr a < inverse x powr b\) using assms
by (intro powr-less-mono') (simp-all add: field-simps)

lemma powr-mono2':
assumes \((x::real) \leq 0\) \(0 < x \leq y\)
shows \(x powr z \geq y powr z\)
proof—
from assms have \(x powr -z \leq y powr -z\)
by (intro powr-mono2') simp-all
with assms show ?thesis by (simp add: field_simps powr-minus)
qed

lemma powr-mono2-ex: \(0 \leq (a::real) \implies 0 \leq x \implies x \leq y \implies x^a \leq y^a\)
  by (cases x = 0) (simp-all add: powr-mono2)

lemma powr-lower-bound: \([l::real) > 0; l \leq x; x \leq u \] \implies \(\min (l^z (u^z)) \leq x^z\)
apply (insert assms)
apply (cases z \geq 0)
apply (rule order.trans[OF min.cobounded1 powr-mono2, simp-all])
done

lemma powr-upper-bound: \([l::real) > 0; l \leq x; x \leq u \] \implies \(\max (l^z (u^z)) \geq x^z\)
apply (insert assms)
apply (cases z \geq 0)
apply (rule order.trans[OF powr-mono2 max.cobounded2, simp-all])
done

lemma powr-eventually-exp-ln: \(\text{eventually } (\lambda x. (x::real) powr p = \exp (p*\ln x)) \text{ at-top}\)
  using eventually-gt-at-top[of 0::real] unfolding powr-def by eventually-elim simp-all

lemma powr-eventually-exp-ln':
  assumes \(x > 0\)
  shows \(\text{eventually } (\lambda x. (x::real) powr p = \exp (p*\ln x)) \text{ (nhds x)}\)
proof-
  let \(?A = \{(0::real)<..\}\)
  from assms have \(\text{eventually } (\lambda x > 0 (nhds x) \text{ unfolding eventually-nhds\ by \(\text{intro ext[of - \{(0::real)<..\}] simp-all\)}\)
    thus ?thesis by eventually-elim (simp add: powr-def)
qed

lemma powr-at-top:
  assumes \((p::real) > 0\)
  shows \(\text{filterlim } (\lambda x. x powr p) \text{ at-top at-top}\)
proof-
  have \(\text{LIM x at-top. exp (p * ln x) :> at-top}\)
    by (rule filterlim-compose[OF exp-at-top filterlim-tendsto-pos-mult-at-top[OF tendsto-const]])
      (simp-all add: ln-at-top assms)
    thus ?thesis by (subst filterlim-cong[OF refl refl powr-eventually-exp-ln])
qed
lemma pour-at-top-neg:
assumes (a::real) > 0 a < 1
shows ((λx. a powr x) ----> 0) at-top
proof-
  from assms have LIM x at-top. ln (inverse a) * x :> at-top
    by (intro filterlim-tendsto-pos-mult-at-top[OF tendsto-const])
    (simp-all add: filterlim-ident field-simps)
  with assms have LIM x at-top. ln a * x :> at-bot
    by (subst filterlim-uminus-at-bot) (simp add: ln-inverse)
  hence ((λx. exp (x * ln a)) ----> 0) at-top
    by (intro filterlim-compose[OF exp-at-bot]) (simp-all add: mult.commute)
  with assms show ?thesis unfolding powr-def by simp
qed

lemma pour-at-bot:
assumes (a::real) > 1
shows ((λx. a powr x) ----> 0) at-bot
proof-
  from assms have filterlim (λx. ln a * x) at-bot at-bot
    by (intro filterlim-tendsto-pos-mult-at-top[OF tendsto-const - filterlim-ident])
  auto
  hence ((λx. exp (x * ln a)) ----> 0) at-bot
    by (intro filterlim-compose[OF exp-at-bot]) (simp add: algebra-simps)
  thus ?thesis using assms unfolding powr-def by simp
qed

lemma pour-at-bot-neg:
assumes (a::real) > 0 a < 1
shows filterlim (λx. a powr x) at-top at-bot
proof-
  from assms have LIM x at-bot. ln (inverse a) * -x :> at-top
    by (intro filterlim-tendsto-pos-mult-at-top[OF tendsto-const - filterlim-ident])
    (simp-all add: ln-inverse)
  with assms have LIM x at-bot. x * ln a :> at-top
    by (subst (asm) ln-inverse) (simp-all add: mult.commute)
  hence LIM x at-bot. exp (x * ln a) :> at-top
    by (intro filterlim-compose[OF exp-at-top]) simp
  thus ?thesis using assms unfolding powr-def by simp
qed

lemma DERIV-powr:
assumes x > 0
shows ((λx. x powr p) has-real-derivative p * x powr (p - 1)) (at x)
proof-
  have ((λx. exp (p * ln x)) has-real-derivative
    exp (p * ln x) * (p * inverse x)) (at x)
  unfolding powr-def by (intro DERIV-fun-exp DERIV-cmult DERIV-ln) fact
  also have exp (p * ln x) * (p * inverse x) = p * x powr (p - 1)
unfolding power-def by (simp add: field-simps exp-diff assms)
finally show ?thesis using assms by (subst DERIV-cong-ev[OF refl powr-eventually-exp-ln’ refl])
qed

2 Definition of Landau symbols

theory Landau-Symbols-Definition
imports Complex-Main
~~/src/HOL/Library/Function-Algebras
~~/src/HOL/Library/Set-Algebras
Landau-Library
begin

2.1 Eventual non-negativity/non-zeroness

For certain transformations of Landau symbols, it is required that the functions involved are eventually non-negative of non-zero. In the following, we set up a system to guide the simplifier to discharge these requirements during simplification at least in obvious cases.

definition eventually-nonzero f ←→ eventually (λx. (f x :: :: linordered-field) ≠ 0) at-top
definition eventually-nonneg f ←→ eventually (λx. (f x :: :: linordered-field) ≥ 0) at-top

named-theorems eventually-nonzero-simps

lemmas [eventually-nonzero-simps] = eventually-not-equal eventually-ln-not-equal

lemma eventually-nonzeroD: eventually-nonzero f ⇒ eventually (λx. f x ≠ 0) at-top
  by (simp add: eventually-nonzero-def)

lemma eventually-nonzero-const [eventually-nonzero-simps]:
eventually-nonzero (λ:::::linorder. c) ←→ c ≠ 0
unfolding eventually-nonzero-def
  by (simp add: eventually-False trivial-limit-at-top-linorder)

lemma eventually-nonzero-inverse [eventually-nonzero-simps]:
eventually-nonzero (λx. inverse (f x)) ←→ eventually-nonzero f
unfolding eventually-nonzero-def by simp

lemma eventually-nonzero-mult [eventually-nonzero-simps]:
eventually-nonzero (λx. f x * g x) ←→ eventually-nonzero f ∧ eventually-nonzero g
unfolding eventually-nonzero-def by (simp-all add: eventually-conj-iff[symmetric])

lemma eventually-nonzero-pow [eventually-nonzero-simps]:
eventually-nonzero (λx::linorder. f x ^ n) ←→ n = 0 ∨ eventually-nonzero f
by (induction n) (auto simp: eventually-nonzero-simps)

lemma eventually-nonzero-divide [eventually-nonzero-simps]:
eventually-nonzero (λx. f x / g x) ←→ eventually-nonzero f ∧ eventually-nonzero g
unfolding eventually-nonzero-def by (simp-all add: eventually-conj-iff[symmetric])

lemma eventually-nonzero-ident [eventually-nonzero-simps]:
eventually-nonzero (λx. x)
unfolding eventually-nonzero-def by (simp add: eventually-not-equal)

lemma eventually-nonzero-ln [eventually-nonzero-simps]:
eventually-nonzero (λx::real. ln x)
unfolding eventually-nonzero-def by (subst eventually-ln-at-top) (rule eventually-not-equal)

lemma eventually-nonzero-ln-const [eventually-nonzero-simps]:
b > 0 ⇒ eventually-nonzero (λx. ln (b * x :: real))
unfolding eventually-nonzero-def using eventually-gt-at-top[of max 1 (inverse b)]
by (auto elim!: eventually-elim1 simp: field-simps)

lemma eventually-nonzero-ln-const' [eventually-nonzero-simps]:
b > 0 ⇒ eventually-nonzero (λx. ln (x * b :: real))
using eventually-nonzero-ln-in-const[of b] by (simp add: mult.commute)

lemma eventually-nonzero-powr [eventually-nonzero-simps]:
eventually-nonzero (λx. f x powr p) ←→ eventually-nonzero f
unfolding eventually-nonzero-def by simp

lemma eventually-nonneg-const [eventually-nonzero-simps]:
eventually-nonneg (λx::linorder. c) ←→ c ≥ 0
unfolding eventually-nonneg-def by (auto simp: eventually-at-top-linorder)

lemma eventually-nonneg-inverse [eventually-nonzero-simps]:
eventually-nonneg (λx. inverse (f x)) ←→ eventually-nonneg f
unfolding eventually-nonneg-def by (intro eventually-subst) simp

lemma eventually-nonneg-add [eventually-nonzero-simps]:
assumes eventually-nonneg f eventually-nonneg g
shows eventually-nonneg (λx. f x + g x)
using assms unfolding eventually-nonneg-def by eventually-elim simp

lemma eventually-nonneg-mult [eventually-nonzero-simps]:
assumes eventually-nonneg f eventually-nonneg g
shows eventually-nonneg (λx. f x * g x)
using assms unfolding eventually-nonneg-def by eventually-elim simp

lemma eventually-nonneg-mult [eventually-nonzero-simps]:
assumes eventually-nonneg (λx. −f x) eventually-nonneg (λx. −g x)
shows eventually-nonneg (λx. f x * g x)
using assms unfolding eventually-nonneg-def by eventually-elim simp

lemma eventually-nonneg-divide [eventually-nonzero-simps]:
assumes eventually-nonneg f eventually-nonneg g
shows eventually-nonneg (λx. f x / g x)
using assms unfolding eventually-nonneg-def by eventually-elim simp

lemma eventually-nonneg-divide' [eventually-nonzero-simps]:
assumes eventually-nonneg (λx. −f x) eventually-nonneg (λx. −g x)
shows eventually-nonneg (λx. f x / g x)
using assms unfolding eventually-nonneg-def by eventually-elim simp

lemma eventually-nonneg-ident [eventually-nonzero-simps]:
eventually-nonneg (λx. x) unfolding eventually-nonneg-def by (rule eventually-ge-at-top)

lemma eventually-nonneg-pow [eventually-nonzero-simps]:
eventually-nonneg f \Rightarrow eventually-nonneg (λx::linorder. f x ^ n)
by (induction n) (auto simp: eventually-nonzero-simps)

lemma eventually-nonneg-powr [eventually-nonzero-simps]:
eventually-nonneg (λx. f x powr y :: real) by (simp add: eventually-nonneg-def)

lemma eventually-nonneg-ln [eventually-nonzero-simps]:
eventually-nonneg (λx. ln x :: real)
by (simp add: eventually-nonneg-def eventually-ln-at-top eventually-ge-at-top)

lemma eventually-nonneg-ln-const [eventually-nonzero-simps]:
b > 0 \Rightarrow eventually-nonneg (λx. ln (b*x) :: real)
unfolding eventually-nonneg-def using eventually-ge-at-top[of inverse b]
by eventually-elim (simp-all add: field-simps)

lemma eventually-nonneg-ln-const' [eventually-nonzero-simps]:
b > 0 \Rightarrow eventually-nonneg (λx. ln (x*b) :: real)
using eventually-nonneg-ln-const[of b]
by (simp add: mult.commute)

2.2 Definition of Landau symbols
Our Landau symbols are sign-oblivious, i.e. any function always has
the same growth as its absolute. This has the advantage of making some
cancelling rules for sums nicer, but introduces some problems in other places.
Nevertheless, we found this definition more convenient to work with.

**definition** bigo :: (('a::order) ⇒ ('b :: linordered-field)) ⇒ ('a ⇒ 'b) set ((10Ω(·-))

where \( O(g) = \{ f. (\exists c>0. \text{ eventually } (\lambda x. |f x| \leq c \cdot |g x|) \text{ at-top}) \} \)

**definition** smallo :: (('a::order) ⇒ ('b :: linordered-field)) ⇒ ('a ⇒ 'b) set ((1oΩ(·-))

where \( o(g) = \{ f. (\forall c>0. \text{ eventually } (\lambda x. |f x| \leq c \cdot |g x|) \text{ at-top}) \} \)

**definition** bigomega :: (('a::order) ⇒ ('b :: linordered-field)) ⇒ ('a ⇒ 'b) set ((1Ω(·-))

where \( \Omega(g) = \{ f. (\exists c>0. \text{ eventually } (\lambda x. |f x| \geq c \cdot |g x|) \text{ at-top}) \} \)

**definition** smallomega :: (('a::order) ⇒ ('b :: linordered-field)) ⇒ ('a ⇒ 'b) set ((1ω(·-))

where \( \omega(g) = \{ f. (\forall c>0. \text{ eventually } (\lambda x. |f x| \geq c \cdot |g x|) \text{ at-top}) \} \)

**definition** bitheta :: (('a::order) ⇒ ('b :: linordered-field)) ⇒ ('a ⇒ 'b) set ((1Θ(·-))

where \( \Theta(g) = O(g) \cap \Omega(g) \)

The following is a set of properties that all Landau symbols satisfy.

**locale** landau-symbol =

**fixes** \( L L' :: (('a::order) ⇒ ('b :: linordered-field)) ⇒ ('a ⇒ 'b) set \)

**assumes** in-cong: eventually (\( \lambda x. f x = g x \)) at-top ⇒ \( f \in L(h) \) if \( g \in L(h) \)

**assumes** cong: eventually (\( \lambda x. f x = g x \)) at-top ⇒ \( L(f) = L(g) \)

**assumes** cong-bitheta: \( f \in \Theta(g) \) ⇒ \( L(f) = L(g) \)

**assumes** in-cong-bitheta: \( f \in \Theta(g) \) ⇒ \( f \in L(h) \) if \( g \in L(h) \)

**assumes** abs [simp]: \( L(\lambda x. |f x|) = L(f) \)

**assumes** abs-in-if [simp]: (\( \lambda x. |f x| \)) ∈ \( L(g) \) if \( f \in L(g) \)

**assumes** cmult [simp]: \( c \neq 0 \) ⇒ \( L(\lambda x. c \cdot f x) = L(f) \)

**assumes** cmult-in-if [simp]: \( c \neq 0 \) ⇒ \( (\lambda x. c \cdot f x) \in L(g) \) if \( f \in L(g) \)

**assumes** mult-left [simp]: \( f \in L(g) \) ⇒ \( (\lambda x. h x \cdot f x) \in L(\lambda x. h x \cdot g x) \)

**assumes** inverse: eventually (\( \lambda x. f x \neq 0 \)) at-top ⇒ eventually (\( \lambda x. g x \neq 0 \)) at-top ⇒ \( f \in L(g) \) if \( (\lambda x. \text{ inverse } (g x)) \in L(h) \) if \( (\lambda x. \text{ inverse } (f x)) \)

**assumes** subset1: \( f \in L(g) \) if \( f \subseteq L(g) \)

**assumes** plus-subset1: \( f \in o(g) \) if \( L(g) \subseteq L(\lambda x . f x + g x) \)

**assumes** trans: \( f \in L(g) \) if \( g \in L(h) \) if \( f \in L(h) \)

**begin**

**lemma** cong-ex:

eventually (\( \lambda x. f1 x = f2 x \)) at-top ⇒ eventually (\( \lambda x. g1 x = g2 x \)) at-top ⇒ \( f1 \in L(g1) \) if \( f2 \in L(g2) \) by (subst cong, assumption, subst in-cong, assumption, rule refl)

**lemma** cong-ex-bitheta:

\( f1 \in \Theta(f2) \) if \( g1 \in \Theta(g2) \) if \( f1 \in L(g1) \) if \( f2 \in L(g2) \)

by (subst cong-bitheta, assumption, subst in-cong-bitheta, assumption, rule refl)
lemma landau-symbol: landau-symbol L
using cong abs abs-in-iff cmult cmult-in-iff plus-subset1 by unfold-locales

lemma bigtheta-trans1:
f ∈ L(g) ⟹ g ∈ Θ(h) ⟹ f ∈ L(h)
by (subst cong-bigtheta[symmetric])

lemma bigtheta-trans2:
f ∈ Θ(g) ⟹ g ∈ L(h) ⟹ f ∈ L(h)
by (subst in-cong-bigtheta)

lemma cmult'[simp]: c ≠ 0 ⟹ L(λx. f x * c) = L(f)
by (subst mult.commute) (rule cmult)

lemma cmult-in-iff'[simp]: c ≠ 0 ⟹ (λx. f x * c) ∈ L(g) ⟷ f ∈ L(g)
using cmult'[of inverse c f] by (simp add: field-simps)

lemma cdiv [simp]: c ≠ 0 ⟹ (λx. f x / c) ∈ L(g) ⟷ f ∈ L(g)
using cmult-in-iff''[of inverse c f] by (simp add: field-simps)

lemma uminus [simp]: L(λx. - g x) = L(g) using cmult[of -1] by simp

lemma uminus-in-iff [simp]: (λx. - f x) ∈ L(g) ⟷ f ∈ L(g)
using cmult-in-iff'[of -1] by simp

lemma const: c ≠ 0 ⟹ L(λ_. c) = L(λ_. 1)
by (subst (2) cmult[symmetric]) simp-all

lemma const'[simp]: NO-MATCH 1 c ⟹ c ≠ 0 ⟹ L(λ_. c) = L(λ_. 1)
by (rule const)

lemma const-in-iff: c ≠ 0 ⟹ (λ_. c) ∈ L(f) ⟷ (λ_. 1) ∈ L(f)
using cmult-in-iff'[of c λ_. 1] by simp

lemma const-in-iff'[simp]: NO-MATCH 1 c ⟹ c ≠ 0 ⟹ (λ_. c) ∈ L(f) ⟷
(λ_. 1) ∈ L(f)
by (rule const-in-iff)

lemma plus-subset2: g ∈ o(f) ⟹ L(f) ⊆ L(λx. f x + g x)
by (subst add.commute) (rule plus-subset1)

lemma mult-right [simp]: f ∈ L(g) ⟹ (λx. f x * h x) ∈ L(λx. g x * h x)
using mult-left by (simp add: mult.commute)
lemmas mult \colon f1 \in L(g1) \implies f2 \in L(g2) \implies (\lambda x. f1 \cdot f2 \cdot x) \in L(\lambda x. g1 \cdot x \cdot g2 \cdot x)
by (rule trans, erule mult-left, erule mult-right)

lemma inverse-cancel:
assumes eventually \((\lambda x. f \cdot x \neq 0)\) at-top
assumes eventually \((\lambda x. g \cdot x \neq 0)\) at-top
shows \((\lambda x. \text{inverse}(f \cdot x)) \in L(\lambda x. \text{inverse}(g \cdot x)) \iff g \in L(f)\)
proof
assume \((\lambda x. \text{inverse}(f \cdot x)) \in L(\lambda x. \text{inverse}(g \cdot x))\)
from inverse[OF - - this] assms shows \(g \in L(f)\) by simp
qed (intro inverse assms)

lemma divide-right:
assumes eventually \((\lambda x. h \cdot x \neq 0)\) at-top
assumes \(f \in L(g)\)
shows \((\lambda x. f \cdot x / h \cdot x) \in L(\lambda x. g \cdot x / h \cdot x)\)
by (subst (1 2) divide-inverse) (intro mult-right inverse assms)

lemma divide-right-iff :
assumes eventually \((\lambda x. h \cdot x \neq 0)\) at-top
shows \((\lambda x. f \cdot x / h \cdot x) \in L(\lambda x. g \cdot x / h \cdot x) \iff f \in L(g)\)
proof
assume \((\lambda x. f \cdot x / h \cdot x) \in L(\lambda x. g \cdot x / h \cdot x)\)
from mult-right[OF this, of h] assms shows \(f \in L(g)\)
by (subst (asm) cong-ex[of - f - g]) (auto elim!: eventually-elim1)
qed (simp add: divide-right assms)

lemma divide-left:
assumes eventually \((\lambda x. f \cdot x \neq 0)\) at-top
assumes eventually \((\lambda x. g \cdot x \neq 0)\) at-top
assumes \(g \in L(f)\)
shows \((\lambda x. h \cdot x / f \cdot x) \in L(\lambda x. h \cdot x / g \cdot x)\)
by (subst (1 2) divide-inverse) (intro mult-left inverse assms)

lemma divide-left-iff:
assumes eventually \((\lambda x. f \cdot x \neq 0)\) at-top
assumes \((\lambda x. h \cdot x \neq 0)\) at-top
assumes eventually \((\lambda x. h \cdot x \neq 0)\) at-top
shows \((\lambda x. h \cdot x / f \cdot x) \in L(\lambda x. h \cdot x / g \cdot x) \iff g \in L(f)\)
proof
assume \(A\) \((\lambda x. h \cdot x / f \cdot x) \in L(\lambda x. h \cdot x / g \cdot x)\)
from assms have \(B\) \((\lambda x. h \cdot x / f \cdot x / h \cdot x = \text{inverse}(f \cdot x))\) at-top
by eventually-elim (simp add: divide-inverse)
from assms have \(C\) \((\lambda x. h \cdot x / g \cdot x / h \cdot x = \text{inverse}(g \cdot x))\) at-top
by eventually-elim (simp add: divide-inverse)
from divide-right[OF assms(3) A] assms shows \(g \in L(f)\)
by (subst (asm) cong-ex[OF B C]) (simp add: inverse-cancel)
qed (simp add: divide-left assms)
lemma \textit{divide}: 
\begin{itemize}
\item \textbf{assumes} \((\lambda x. \; g1 \; x \neq 0)\) at-top
\item \textbf{assumes} \((\lambda x. \; g2 \; x \neq 0)\) at-top
\item \textbf{assumes} \(f1 \in L(f2)\) \(g2 \in L(g1)\)
\item \textbf{shows} \((\lambda x. \; f1 \; x / g1 \; x) \in L(\lambda x. \; f2 \; x / g2 \; x)\)
\end{itemize}
by (subst (1 2) divide-inverse) (intro mult inverse assms)
\begin{proof}
\item \textbf{have} \(f \in L(\lambda x. \; g \; x / h \; x) \iff (\lambda x. \; f \; x \; * \; h \; x) \in L(\lambda x. \; g \; x / h \; x)\)
\item \textbf{using} \textit{assms} by (intro in-cong) (auto elim: eventually-elim1)
\item \textbf{thus} \(?thesis\) by (simp only: divide-right-iff assms)
\end{proof}

lemma \textit{divide-eq1}: 
\begin{itemize}
\item \textbf{assumes} \((\lambda x. \; h \; x \neq 0)\) at-top
\item \textbf{shows} \((\lambda x. \; f \; x \; / \; h \; x) \iff (\lambda x. \; f \; x \; x \; h \; x) \in L(\lambda x. \; g \; x / h \; x)\)
\end{itemize}
\begin{proof}
\item \textbf{have} \(L(\lambda x. \; g \; x) = L(\lambda x. \; g \; x * h \; x / h \; x)\)
\item \textbf{using} \textit{assms} by (intro cong) (auto elim: eventually-elim1)
\item \textbf{thus} \(?thesis\) by (simp only: divide-right-iff assms)
\end{proof}

lemma \textit{inverse-flip}: 
\begin{itemize}
\item \textbf{assumes} \((\lambda x. \; g \; x \neq 0)\) at-top
\item \textbf{assumes} \((\lambda x. \; h \; x \neq 0)\) at-top
\item \textbf{assumes} \((\lambda x. \; \text{inverse} \; (g \; x)) \in L(h)\)
\item \textbf{shows} \((\lambda x. \; \text{inverse} \; (h \; x)) \in L(g)\)
\item \textbf{using} \textit{assms} by (simp add: divide-eq1 divide-eq2 inverse-eq-divide \textit{mult.commute})
\end{itemize}

lemma \textit{lift-trans}: 
\begin{itemize}
\item \textbf{assumes} \(f \in L(g)\)
\item \textbf{assumes} \((\lambda x. \; t \; x \; (g \; x)) \in L(h)\)
\item \textbf{shows} \((\lambda x. \; t \; (f \; x)) \in L(h)\)
\item \textbf{by} (rule trans[OF assms(\textit{3})[OF assms(\textit{1}) \; assms(\textit{2})]])
\end{itemize}

lemma \textit{lift-trans\:'}: 
\begin{itemize}
\item \textbf{assumes} \(f \in L(\lambda x. \; t \; x \; (g \; x))\)
\item \textbf{assumes} \(g \in L(h)\)
\item \textbf{assumes} \((\lambda h. \; g \; h) \in L(h)\)
\item \textbf{shows} \((\lambda x. \; t \; x \; (g \; x)) \in L(\lambda x. \; t \; x \; (h \; x))\)
\item \textbf{by} (rule trans[OF assms(\textit{1}) \; assms(\textit{3})[OF assms(\textit{2})]])
\end{itemize}

lemma \textit{lift-trans-bigtheta}: 
\begin{itemize}
\item \textbf{assumes} \(f \in L(g)\)
\end{itemize}
assumes \((\lambda x. t x (g x)) \in \Theta(h)\)
assumes \(\bigwedge f. g. f \in L(g) \implies (\lambda x. t x (f x)) \in L(\lambda x. t x (g x))\)
shows \((\lambda x. t x (f x)) \in L(h)\)
using cong-bigtheta\([OF \text{ assms}(2)] \text{ assms}(3)[OF \text{ assms}(1)]\) by simp

lemma lift-trans-bigtheta':
assumes \(f \in L(\lambda x. t x (g x))\)
assumes \(g \in \Theta(h)\)
assumes \(\bigwedge g. h. g \in \Theta(h) \implies (\lambda x. t x (g x)) \in \Theta(\lambda x. t x (h x))\)
shows \(f \in L(\lambda x. t x (h x))\)
using cong-bigtheta\([OF \text{ assms}(3)][OF \text{ assms}(2)]\) \text{ assms}(1) by simp

end

The symbols \(O\) and \(o\) and \(\Omega\) and \(\omega\) are dual, so for many rules, replacing \(O\) with \(\Omega\), \(o\) with \(\omega\), and \(\le\) with \(\ge\) in a theorem yields another valid theorem.

The following locale captures this fact.

locale landau-pair =
fixes \(L\) \(l\) :: \(\langle\langle a :: \text{order} \rangle \Rightarrow \langle b :: \text{linordered-field}\rangle \Rightarrow \langle a \Rightarrow b\rangle\rangle\) set
and \(R\) :: \(\langle b :: \text{linordered-field}\rangle \Rightarrow \langle b \Rightarrow \text{bool}\rangle\rangle\)
assumes \(L\text{-def}\): \(L g = \{ f. \exists c > 0. \text{ eventually} (\lambda x. R |f x| (c * |g x|)) \text{ at-top}\}\)
and \(l\text{-def}\): \(l g = \{ f. \forall c > 0. \text{ eventually} (\lambda x. R |f x| (c * |g x|)) \text{ at-top}\}\)
and \(R\): \(R = \text{ op \ le \ or \ op \ ge}\)
interpretation landau-o!: landau-pair bigo smallo op \(\le\)
by unfold-locales (auto simp: bigo-def smallo-def intro!: ext)

interpretation landau-omega!: landau-pair bigomega smallomega op \(\ge\)
by unfold-locales (auto simp: bigomega-def smallomega-def intro!: ext)

context landau-pair
begin

lemmas \(R-E = \text{ disjE}[OF R]\)

lemma bigI:
\(c > 0 \implies \text{ eventually} (\lambda x. R |f x| (c * |g x|)) \text{ at-top} \implies f \in L(g)\)
unfolding \(L\text{-def}\) by blast

lemma bigE:
assumes \(f \in L(g)\)
obtains \(c\) where \(c > 0 \text{ eventually} (\lambda x. R |f x| (c * |g x|)) \text{ at-top}\)
using \text{ assms unfolding} \(L\text{-def}\) by blast

lemma bigE-nonneg:
assumes \(f \in L(g)\) eventually \((\lambda x. f x \ge 0)\) \text{ at-top}
obtains \(c\) where \(c > 0 \text{ eventually} (\lambda x. R (f x) (c * |g x|)) \text{ at-top}\)

proof –

14
from assms(1) guess c by (rule bigE) note c = this
from c(2) assms(2) have eventually \(\lambda x. R (f x) (c * |g x|)\) at-top
  by eventually-elim simp
from c(1) and this show \(\text{thesis}\) by (rule that)

qed

lemma smallI:
(\(\forall c. c > 0 \Rightarrow \text{eventually} (\lambda x. R (f x) (c * |g x|)) \text{ at-top}\)) \(\Rightarrow f \in l(g)\)
unfolding l-def by blast

lemma smallD:
f \(\in l(g) \Rightarrow c > 0 \Rightarrow \text{eventually} (\lambda x. R (f x) (c * |g x|)) \text{ at-top}\)
using assms unfolding l-def by blast

lemma smallD-nonneg:
assumes \(f \in l(g) \text{ eventually} (\lambda x. f x \geq 0) \text{ at-top} c > 0\)
shows \(\text{eventually} (\lambda x. R (f x) (c * |g x|)) \text{ at-top}\)
using smallD[OF assms(1,3)] assms(2) by eventually-elim simp

lemma small-imp-big: \(f \in l(g) \Rightarrow f \in L(g)\)
by (rule bigI[OF - smallD, of 1]) simp-all

lemma small-subset-big: \(l(g) \subseteq L(g)\)
using small-imp-big by blast

lemma R-refl [simp]: \(R x x\) using R by auto

lemma R-linear: \(\neg R x y \Rightarrow R y x\)
using R by auto

lemma R-trans [trans]: \(R a b \Rightarrow R b c \Rightarrow R a c\)
using R by auto

lemma R-mult-left-mono: \(R a b \Rightarrow c \geq 0 \Rightarrow R (c*a) (c*b)\)
using R by (auto simp: mult-left-mono)

lemma R-mult-right-mono: \(R a b \Rightarrow c \geq 0 \Rightarrow R (a*c) (b*c)\)
using R by (auto simp: mult-right-mono)

lemma big-trans:
assumes \(f \in L(g)\) \(g \in L(h)\)
shows \(f \in L(h)\)

proof -
  from assms(1) guess c by (elim bigE) note c = this
  from assms(2) guess d by (elim bigE) note d = this
  from c(2) d(2) have eventually \(\lambda x. R (f x) (c * d * |h x|)\) at-top
  proof eventually-elim
    fix x assume R \(f x\) (c * |g x|)

  15
also assume $R \mid g \mid (d \ast h \mid x)$
with $c(1)$ have $R \mid (c \ast g \mid x) \mid (c \ast (d \ast h \mid x))$
by (intro $R$-mult-left-mono) simp-all
finally show $R \mid f \mid (c \ast d \ast h \mid x)$ by (simp add: algebra-simps)
qed
with $c(1) \hspace{1em} d(1)$ show thesis by (intro bigI[of $c \ast d$]) simp-all
qed

lemma big-small-trans:
assumes $f \in L(g) \hspace{1em} g \in l(h)$
shows $f \in l(h)$
proof (rule smallI)
  fix $c :: 'b$
  assume $c > 0$
  from assms(1) guess $d$ by (elim $bigE$) note $d = this$
  note $d(2)$
  moreover from $c \hspace{1em} d$ assms(2) have eventually $(\lambda x. R \mid g \mid x) \mid (c \ast inverse \hspace{1em} d \ast h \mid x)$
at-top
  by (intro smallD) simp-all
  ultimately show eventually $(\lambda x. R \mid f \mid x) \mid (c \ast h \mid x))$ at-top
  by eventually-elim (erule $R$-trans, insert $R \hspace{1em} d(1)$, auto simp: field-simps)
qed

lemma small-big-trans:
assumes $f \in l(g) \hspace{1em} g \in L(h)$
shows $f \in L(h)$
proof (rule smallI)
  fix $c :: 'b$
  assume $c > 0$
  from assms(2) guess $d$ by (elim $bigE$) note $d = this$
  note $d(2)$
  moreover from $c \hspace{1em} d$ assms(1) have eventually $(\lambda x. R \mid f \mid x) \mid (c \ast inverse \hspace{1em} d \ast g \mid x))$ at-top
  by (intro smallD) simp-all
  ultimately show eventually $(\lambda x. R \mid f \mid x) \mid (c \ast h \mid x))$ at-top
  by eventually-elim (rotate-tac 2, erule $R$-trans, insert $R \hspace{1em} c \hspace{1em} d(1)$, auto simp: field-simps)
qed

lemma small-trans:
$f \in l(g) \implies g \in l(h) \implies f \in l(h)$
by (rule big-small-trans[of $small-imp-big$])

lemma small-big-trans':
$f \in l(g) \implies g \in L(h) \implies f \in L(h)$
by (rule small-imp-big[of $small-big-trans$])

lemma big-small-trans':
$f \in L(g) \implies g \in l(h) \implies f \in L(h)$
by (rule small-imp-big[of $big-small-trans$])
lemma big-subsetI [intro]: \( f \in L(g) \implies L(f) \subseteq L(g) \)
by (intro subsetI) (drule (1) big-trans)

lemma small-subsetI [intro]: \( f \in L(g) \implies l(f) \subseteq l(g) \)
by (intro subsetI) (drule (1) small-big-trans)

lemma big-refl [simp]: \( f \in L(f) \)
by (rule bigI[of 1]) simp-all

lemma small-refl-iff: \( f \in l(f) \leftrightarrow \text{eventually} (\lambda x. f x = 0) \at-top \)
proof (rule iffI[OF - smallI])
assume \( f : f \in l f \)
have \((1/2::'b) > 0 (2::'b) > 0\) by simp-all
from smallD[OF \( f \) this(1)] smallD[OF \( f \) this(2)]
show eventually \((\lambda x. f x = 0) \at-top\) by eventually-elim (insert \( R \), auto)
next
fix \( c :: 'b \)
assume \( c > 0 \) eventually \((\lambda x. f x = 0) \at-top \)
from this(2) show eventually \((\lambda x. \text{R} \mid f x \mid (c \ast \mid f x \mid)) \at-top \)
  by eventually-elim simp-all
qed

lemma big-small-asymmetric: \( f \in L(g) \implies g \in l(f) \implies \text{eventually} (\lambda x. f x = 0) \at-top \)
by (drule (1) big-small-trans) (simp add: small-refl-iff)

lemma small-big-asymmetric: \( f \in l(g) \implies g \in L(f) \implies \text{eventually} (\lambda x. f x = 0) \at-top \)
by (drule (1) small-big-trans) (simp add: small-refl-iff)

lemma small-asymmetric: \( f \in l(g) \implies g \in l(f) \implies \text{eventually} (\lambda x. f x = 0) \at-top \)
by (drule (1) small-trans) (simp add: small-refl-iff)

lemma plus-aux:
assumes \( f \in o(g) \)
shows \( g \in L(\lambda x. f x + g x) \)
proof (rule R-E)
assume [simp]: \( R = op \leq \)
have \( A : 1/2 > (0::'b) \) by simp
{ 
  fix \( x \) assume \( |f x| \leq 1/2 \ast |g x| \)
  hence \( 1/2 \ast |g x| \leq |g x| - |f x| \) by simp
  also have \( |g x| - |f x| \leq |f x + g x| \) by simp
  finally have \( 1/2 \ast |g x| \leq |f x + g x| \) by simp
} note \( B = this \)
show \( g \in L(\lambda x. f x + g x) \)
  apply (rule bigI[of 2], simp)
using landau-o.smallD[OF assms A]
apply eventually-elim
using B
apply (simp add: algebra-simps)
done
next
assume [simp]; R = (\lambda x y. x \geq y)
show g \in L(\lambda x. f x + g x)
apply (rule bigI[of 1/2], simp)
using landau-o.smallD[OF assms zero-less-one]
apply eventually-elim
apply (auto simp: algebra-simps)
done
qed
end

lemma bigomega-iff-bigo: g \in \Omega(f) \iff f \in O(g)
proof
  assume f \in O(g)
  then guess c by (elim landau-o.bigE)
  thus g \in \Omega(f) by (intro landau-omega.bigI[of inverse c]) (simp-all add: field-simps)
next
  assume g \in \Omega(f)
  then guess c by (elim landau-omega.bigE)
  thus f \in O(g) by (intro landau-o.bigI[of inverse c]) (simp-all add: field-simps)
qed

lemma smallomega-iff-smallo: g \in \omega(f) \iff f \in o(g)
proof
  assume f \in o(g)
  from landau-o.smallD[OF this, of inverse c for c]
  show g \in \omega(f) by (intro landau-omega.smallI) (simp-all add: field-simps)
next
  assume g \in \omega(f)
  from landau-omega.smallD[OF this, of inverse c for c]
  show f \in o(g) by (intro landau-o.smallI) (simp-all add: field-simps)
qed

context landau-pair
begin

lemma big-mono:
  eventually (\lambda x. R \ | f x| | g x|) at-top \Longrightarrow f \in L(g)
by (rule bigI[OF zero-less-one]) simp

lemma big-mult:
  assumes f1 \in L(g1) f2 \in L(g2)
  shows (\lambda x. f1 x \ast f2 x) \in L(\lambda x. g1 x \ast g2 x)

end
proof
  from assms(1) guess c1 by (elim bigE) note c1 = this
from assms(2) guess c2 by (elim bigE) note c2 = this

from c1(1) and c2(1) have c1 * c2 > 0 by simp
moreover from c1(2) and c2(2)
  have eventually (λx. R |f1 x * f2 x| (c1 * c2 * |g1 x * g2 x|)) at-top
    apply (eventually-elim)
    apply (rule R-E)
    apply (hypsubst, drule (1) mult-mono, insert c1(1) c2(1), simp, simp, simp add: abs-mult field-simps)+
  done
ultimately show ?thesis by (rule bigI)
qed

lemma small-big-mult:
  assumes f1 ∈ l(g1) f2 ∈ L(g2)
  shows (λx. f1 x * f2 x) ∈ l(λx. g1 x * g2 x)
proof (rule smallII)
  fix c1 :: 'b assume c1: c1 > 0
  from assms(2) guess c2 by (elim bigE) note c2 = this
  with c1 assms(1) have eventually (λx. R |f1 x| (c1 * inverse c2 * |g1 x|)) at-top
    by (auto intro!: smallD)
  with c2(2) show eventually (λx. R |f1 x * f2 x| (c1 * |g1 x * g2 x|)) at-top
    apply eventually-elim
    apply (rule R-E, insert c1(1) c2(1))
    apply (hypsubst, drule (1) mult-mono, simp, simp, simp add: abs-mult field-simps)+
  done
qed

lemma big-small-mult: f1 ∈ L(g1) ⟹ f2 ∈ l(g2) ⟹ (λx. f1 x * f2 x) ∈ l(λx. g1 x * g2 x)
  by (subst (1 2) mult.commute) (rule small-big-mult)

lemma small-mult: f1 ∈ l(g1) ⟹ f2 ∈ l(g2) ⟹ (λx. f1 x * f2 x) ∈ l(λx. g1 x * g2 x)
  by (rule small-big-mult, assumption, rule small-imp-big)

lemmas mult = big-mult small-big-mult big-small-mult small-mult

sublocale big!: landau-symbol L
proof
  have L: L = bigo ∨ L = bigomega
    apply (rule R-E)
    apply (rule disjI1, rule ext, simp add: bigo-def L-def)
    apply (rule disjI2, rule ext, simp add: bigomega-def L-def)
  done
{
fix c :: 'b and f :: 'a ⇒ 'b assume c ≠ 0

hence (λx. c ∗ f x) ∈ L f by (intro bigI[of ‹c›]) (simp-all add: abs-mult)

} note A = this

{  
fix c :: 'b and f :: 'a ⇒ 'b assume c ≠ 0
from ‹c ≠ 0› and A[of c f] and A[of inverse c λx. c ∗ f x]
  show L (λx. c ∗ f x) = L f by (intro equalityI big-subsetI) (simp-all add: field-simps)

}

{  
fix c :: 'b and f g :: 'a ⇒ 'b assume c ≠ 0
from ‹c ≠ 0› and A[of c f] and A[of inverse c λx. c ∗ f x]
  have (λx. c ∗ f x) ∈ L f f ∈ L (λx. c ∗ f x) by (simp-all add: field-simps)
  thus ((λx. c ∗ f x) ∈ L g) = (f ∈ L g) by (intro iffI) (erule (1) big-trans)+

}

{  
fix f g :: 'a ⇒ 'b assume A: f ∈ L(g)
assume B: eventually (λx. f x ≠ 0) at-top eventually (λx. g x ≠ 0) at-top
from A guess c by (elim bigE) note c = this
from c(2) B have eventually (λx. R |inverse (g x)) (c ∗ |inverse (f x))) at-top by eventually-elim (rule R-E, insert c(1), simp-all add: field-simps)
  with c(1) show (λx. inverse (g x)) ∈ L(λx. inverse (f x)) by (rule bigI)

}

{  
fix f g :: 'a ⇒ 'b assume f ∈ o(g)
with plus-aaux show L g ⊆ L (λx. f x + g x) by (blast intro!: big-subsetI)

}

{  
fix f g :: 'a ⇒ 'b assume A: eventually (λx. f x = g x) at-top
  show L(f) = L(g) unfolding L-def by (subst eventually-subst[of A]) (rule refl)

}

{  
fix f g h :: 'a ⇒ 'b assume A: eventually (λx. f x = g x) at-top
  show f ∈ L(h) ⇔ g ∈ L(h) unfolding L-def mem-Collect-eq
    by (subst (1) eventually-subst[of A]) (rule refl)

}

{  
fix f g :: 'a ⇒ 'b assume f ∈ L g thus L f ⊆ L g by (rule big-subsetI)

}

{  
fix f g :: 'a ⇒ 'b assume A: f ∈ Θ(g)
with A L show L(f) = L(g) unfolding bigtheta-def
  by (intro equalityI big-subsetI) (auto simp: bigomega-iff-bigo)
fix h :: 'a ⇒ 'b
  show f ∈ L(h) ⇔ g ∈ L(h) by (rule disjE[of L])
    (insert A, auto simp: bigtheta-def bigomega-iff-bigo intro: landau-o.big-trans)
\textbf{proof}\ 
\begin{enumerate}
\item \texttt{fix f g h :: 'a ⇒ 'b assume f ∈ L g}
\item \texttt{thus (λx. h x * f x) ∈ L (λx. h x * g x) by (intro big-mult) simp}
\end{enumerate}
\begin{enumerate}
\item \texttt{fix f g h :: 'a ⇒ 'b assume f ∈ L g g ∈ L h}
\item \texttt{thus f ∈ L(h) by (rule big-trans)}
\end{enumerate}
\texttt{qed (auto simp: L-def)}

\textbf{sublocale} \texttt{small!: landau-symbol l}
\begin{enumerate}
\item \texttt{fix c :: 'b and f :: 'a ⇒ 'b assume c ≠ 0}
\item \texttt{hence (λx. c * f x) ∈ L f by (intro bigI[of |c|]) (simp-all add: abs-mult)}
\end{enumerate}
\texttt{note A = this}
\begin{enumerate}
\item \texttt{fix c :: 'b and f :: 'a ⇒ 'b assume c ≠ 0}
\item \texttt{from (c ≠ 0) and A[of c f] and A[of inverse c λx. c * f x]}
\item \texttt{show l (λx. c * f x) = l f by (intro equalityI small-subsetI) (simp-all add: field-simps)}
\end{enumerate}
\begin{enumerate}
\item \texttt{fix c :: 'b and f :: 'a ⇒ 'b assume c ≠ 0}
\item \texttt{from (c ≠ 0) and A[of c f] and A[of inverse c λx. c * f x]}
\item \texttt{have (λx. c * f x) ∈ L f f ∈ L (λx. c * f x) by (simp-all add: field-simps)}
\item \texttt{thus ((λx. c * f x) ∈ l y) = (f ∈ l g) by (intro iffI) (erule (1) big-small-trans)+}
\end{enumerate}
\begin{enumerate}
\item \texttt{fix f g :: 'a ⇒ 'b assume f ∈ o(g)}
\item \texttt{with plus-aux show l g ≺ l (λx. f x + g x) by (blast intro!: small-subsetI)}
\end{enumerate}
\begin{enumerate}
\item \texttt{fix f g :: 'a ⇒ 'b assume A: f ∈ l(g)}
\item \texttt{assume B: eventually (λx. f x ≠ 0) at-top eventually (λx. g x ≠ 0) at-top}
\item \texttt{show (λx. inverse (g x)) (l(λx. inverse (f x)))} 
\item \texttt{proof (rule smallI)}
\item \texttt{fix c :: 'b assume c: c > 0}
\item \texttt{from B smallD[OF A c] show eventually (λx. R of inverse (g x)) (c * |inverse (f x)|)} 
\item \texttt{by eventually-elim (rule R-E, simp-all add: field-simps)}
\item \texttt{qed}
\end{enumerate}
\begin{enumerate}
\item \texttt{fix f g :: 'a ⇒ 'b assume A: eventually (λx. f x = g x) at-top}
\item \texttt{show l(f) = l(g) unfolding l-def by (subst eventually-subst"[OF A]) (rule refl)}
\end{enumerate}
\begin{enumerate}
\item \texttt{fix f g h :: 'a ⇒ 'b assume A: eventually (λx. f x = g x) at-top}
\item \texttt{show f ∈ l(h) ↔ g ∈ l(h) unfolding l-def mem-Collect-eq}
\end{enumerate}
These rules allow chaining of Landau symbol propositions in Isar with "also".

**Lemma** big-mul-1: \( f \in L(g) \Rightarrow (\lambda -. 1) \in L(h) \Rightarrow f \in L(\lambda x. g x \ast h x) \)

**And** big-mul-1": \((\lambda -. 1) \in L(g) \Rightarrow f \in L(h) \Rightarrow f \in L(\lambda x. g x \ast h x) \)

**And** small-mul-1: \( f \in l(g) \Rightarrow (\lambda -. 1) \in L(h) \Rightarrow f \in l(\lambda x. g x \ast h x) \)

**And** small-mul-1": \((\lambda -. 1) \in L(g) \Rightarrow f \in l(h) \Rightarrow f \in l(\lambda x. g x \ast h x) \)

**And** small-mul-1": \((\lambda -. 1) \in l(g) \Rightarrow f \in l(h) \Rightarrow f \in l(\lambda x. g x \ast h x) \)

**By** (drule (1) big-mul big-small-mul small-big-mul, simp)+

**Lemma** big-1-mult: \( f \in L(g) \Rightarrow h \in L(\lambda -. 1) \Rightarrow (\lambda x. f x \ast h x) \in L(g) \)

**And** big-1-mult": \( h \in L(\lambda -. 1) \Rightarrow f \in L(g) \Rightarrow (\lambda x. f x \ast h x) \in L(g) \)

**And** small-1-mult: \( f \in l(g) \Rightarrow h \in L(\lambda -. 1) \Rightarrow (\lambda x. f x \ast h x) \in l(g) \)

**And** small-1-mult": \( h \in L(\lambda -. 1) \Rightarrow f \in l(g) \Rightarrow (\lambda x. f x \ast h x) \in l(g) \)

**And** small-1-mult": \( f \in L(g) \Rightarrow h \in l(\lambda -. 1) \Rightarrow (\lambda x. f x \ast h x) \in l(g) \)
and small-1-mult'': \( h \in l(\lambda \cdot 1) \implies f \in L(g) \implies (\lambda x. f x * h x) \in l(g) \)

by (drule (1) big-mult big-small-mult small-big-mult, simp)+

lemmas mult-1-trans =

big-mult-1 big-mult-1' small-mult-1 small-mult-1' small-mult-1''
big-1-mult big-1-mult' small-1-mult small-1-mult' small-1-mult''

lemma big-equal-iff-bigtheta: \( L(f) = L(g) \iff f \in \Theta(g) \)

proof

have \( L = \text{bigo} \lor L = \text{bigomega} \)

apply (rule R-E)

apply (rule disjI1, rule ext, simp add: bigo-def L-def)

apply (rule disjI2, rule ext, simp add: bigomega-def L-def)

done

fix \( f \cdot g :: \alpha \Rightarrow \beta \)

assume \( L(f) = L(g) \)

with \( \text{big-refl} [\text{of } f] \text{ big-refl} [\text{of } g] \)

have \( f \in L(g) \land g \in L(f) \) by simp-all

thus \( f \in \Theta(g) \) using \( L \) unfolding bigomega Def by (auto simp: bigomega-iff-bigo)

qed (rule big.cong-bigtheta)

end

context landau-symbol

begin

lemma plus-absorb1:

assumes \( f \in o(g) \)

shows \( L(\lambda x. f x + g x) = L(g) \)

proof (intro equalityI)

from plus-subset1 and assms show \( L g \subseteq L (\lambda x. f x + g x) \).

from landau-o.small.plus-subset1[OF assms] and assms have \( (\lambda x. - f x) \in o(\lambda x. f x + g x) \)

by (auto simp: landau-o.small.uminus-in-iff)

from plus-subset1[OF this] show \( L(\lambda x. f x + g x) \subseteq L(g) \) by simp

qed

lemma plus-absorb2: \( g \in o(f) \implies L(\lambda x. f x + g x) = L(f) \)

using plus-absorb1[of g f] by (simp add: add.commute)

lemma diff-absorb1: \( f \in o(g) \implies L(\lambda x. f x - g x) = L(g) \)

by (simp only: diff-conv-add-uminus plus-absorb1 landau-o.small.uminus minus)

lemma diff-absorb2: \( g \in o(f) \implies L(\lambda x. f x - g x) = L(f) \)

by (simp only: diff-conv-add-uminus plus-absorb2 landau-o.small.uminus-in-iff)

lemmas absorb = plus-absorb1 plus-absorb2 diff-absorb1 diff-absorb2

end
lemma bigtheta1 [intro]: \( f \in O(g) \implies f \in \Omega(g) \implies f \in \Theta(g) \)
unfolding bigtheta-def bigomega-def by blast

lemma bigthetaD1 [dest]: \( f \in \Theta(g) \implies f \in O(g) \) and bigthetaD2 [dest]: \( f \in \Theta(g) \implies f \in \Omega(g) \)
unfolding bigtheta-def bigo-def bigomega-def by blast

lemma bigtheta-refl [simp]: \( f \in \Theta(f) \)
unfolding bigtheta-def by simp

lemma bigtheta-sym: \( f \in \Theta(g) \iff g \in \Theta(f) \)
unfolding bigtheta-def by (auto simp: bigomega-iff-bigo)

lemmas landau-flip =
bigomega-iff-bigo[bisymmetric] smallomega-iff-smallo[bisymmetric]
bigomega-iff-bigo smallomega-iff-smallo bigtheta-sym

interpretation landau-theta!: landau-symbol bigtheta
proof
  fix f g :: 'a \Rightarrow 'b
  assume f \in o(g)
  hence \( O(g) \subseteq O(\lambda x. f x + g x) \) \( \Omega(g) \subseteq \Omega(\lambda x. f x + g x) \)
    by (rule landau-o.big.plus-subset1 landau-omega.big.plus-subset1)+
  thus \( \Theta(g) \subseteq \Theta(\lambda x. f x + g x) \) unfolding bigtheta-def by blast
next
  fix f g :: 'a \Rightarrow 'b
  assume f \in \Theta(g)
  thus A: \( \Theta(f) = \Theta(g) \)
    apply (subst (1 2) bigtheta-def)
    apply (subst landau-o.big.cong-bigtheta landau-omega.big.cong-bigtheta, assumption)+
    apply (rule refl)
  done
  thus \( \Theta(f) \subseteq \Theta(g) \) by simp
fix h :: 'a \Rightarrow 'b
show f \in \Theta(h) \iff g \in \Theta(h) by (subst (1 2) bigtheta-sym) (simp add: A)
next
  fix f g h :: 'a \Rightarrow 'b
  assume f \in \Theta(g) g \in \Theta(h)
  thus f \in \Theta(h) unfolding bigtheta-def
    by (blast intro: landau-o.big.trans landau-omega.big.trans)
qed (simp-all add: bigtheta-def landau-o.big.abs landau-omega.big.abs
  landau-o.big.abs-in-iff landau-omega.big.abs-in-iff
  landau-o.big.cmult landau-omega.big.cmult
  landau-o.big.cmult-in-iff landau-omega.big.cmult-in-iff
  landau-o.big.cong landau-omega.big.cong
  landau-o.big.in-cong landau-omega.big.in-cong)
lemmas landau-symbols =
  landau-o.big.landau-symbol landau-o.small.landau-symbol
  landau-omega.big.landau-symbol landau-omega.small.landau-symbol landau-theta.landau-symbol

lemma bigoI [intro]:
  assumes eventually (λx. |f x| ≤ c * |g x|) at-top
  shows f ∈ O(g)
proof (rule landau-o.bigoI)
  show max 1 c > 0 by simp
  note assms
  moreover have \( x \cdot c \cdot |g x| ≤ max 1 c \cdot |g x| \) by (simp add: mult-right-mono)
  ultimately show eventually (λx. |f x| ≤ max 1 c * |g x|) at-top
  by (auto elim!: eventually-elim1 dest: order.trans)
qed

declare landau-o.smallI landau-omega.bigI landau-omega.smallI [intro]
declare landau-o.bigE landau-omega.bigE [elim]
declare landau-o.smallD landau-omega.smallD [dest]

lemma (in landau-symbol) bigtheta-trans1':
  f ∈ L(g) ⇒ h ∈ Θ(g) ⇒ f ∈ L(h)
  by (subst cong-bigtheta[symmetric]) (simp add: bigtheta-sym)

lemma (in landau-symbol) bigtheta-trans2':
  g ∈ Θ(f) ⇒ g ∈ L(h) ⇒ f ∈ L(h)
  by (rule bigtheta-trans2', subst bigtheta-sgm)

lemma bigo-bigomega-trans: f ∈ O(g) ⇒ h ∈ Ω(g) ⇒ f ∈ O(h)
and bigo-smallomega-trans: f ∈ O(g) ⇒ h ∈ ω(g) ⇒ f ∈ o(h)
and smallo-bigomega-trans: f ∈ o(g) ⇒ h ∈ Ω(g) ⇒ f ∈ o(h)
and smallo-smallomega-trans: f ∈ o(g) ⇒ h ∈ ω(g) ⇒ f ∈ o(h)
and bigomega-bigomega-trans: f ∈ Ω(g) ⇒ h ∈ O(g) ⇒ f ∈ Ω(h)
and bigomega-smallomega-trans: f ∈ Ω(g) ⇒ h ∈ o(g) ⇒ f ∈ ω(h)
and smallomega-bigomega-trans: f ∈ ω(g) ⇒ h ∈ O(g) ⇒ f ∈ ω(h)
and smallomega-smallomega-trans: f ∈ ω(g) ⇒ h ∈ o(g) ⇒ f ∈ ω(h)
by (unfold bigomega-iff-big smallomega-iff-small)
  (erule (1) landau-o.big-trans landau-o.big-small-trans landau-o.small-big-trans
landau-o.big-trans landau-o.small-trans)

lemmas landau-trans-lift [trans] =
  landau-symbols[THEN landau-symbol.lift-trans]
  landau-symbols[THEN landau-symbol.lift-trans']
  landau-symbols[THEN landau-symbol.lift-trans-bigtheta]
  landau-symbols[THEN landau-symbol.lift-trans-bigtheta']
lemmas landau-mult-1-trans [trans] =
landau-o.mult-1-trans landau-omega.mult-1-trans

lemmas landau-trans [trans] =
landau-symbols[THEN landau-symbol.bigtheta-trans1]
landau-symbols[THEN landau-symbol.bigtheta-trans2]
landau-symbols[THEN landau-symbol.bigtheta-trans1']
landau-symbols[THEN landau-symbol.bigtheta-trans2']

landau-o.big-trans landau-o.small-trans landau-o.big-small-trans landau-o.big-small-trans
landau-omega.big-trans landau-omega.small-trans
landau-omega.small-big-trans landau-omega.big-small-trans

bigo-bigomega-trans bigo-smallomega-trans smallo-bigomega-trans smallo-smallomega-trans

bigo-bigomega-trans bigo-smallomega-trans smallo-bigomega-trans smallo-smallomega-trans

lemma bigtheta-inverse [simp]:
fixes f g :: ('a :: order) ⇒ ('b :: linordered-field)
serves (λx. inverse (f x)) ∈ Θ(λx. inverse (g x)) ↔ f ∈ Θ(g)
proof
- { fix f g :: 'a ⇒ 'b assume A: f ∈ Θ(g)
  then guess c1 c2 :: 'b unfolding bigtheta-def by (elim landau-o.bigE landau-omega.bigE
  IntE)
  note c = this
  from c(3) have inverse c2 > 0 by simp
  moreover from c(2,4)
  have eventually (λx. inverse (f x)) ≤ inverse c2 * inverse (g x)) at-top
  proof eventually-elim
  fix x assume A: [f x] ≤ c1 * [g x] c2 * [g x] ≤ [f x]
  from A c(1,3) have f x = 0 ↔ g x = 0 by (auto simp: field-simps
  mult-le-0-iff)
  with A c(1,3) show [inverse (f x)] ≤ inverse c2 * inverse (g x)]
  by (force simp: field-simps)
  qed
  ultimately have (λx. inverse (f x)) ∈ O(λx. inverse (g x)) by (rule landau-o.bigI)
  }
  thus ?thesis using assms unfolding bigtheta-def
  by (force simp: bigomega-iff-bigo bigtheta-sym)
  qed

lemma bigtheta-divide:
assumes f1 ∈ Θ(f2) g1 ∈ Θ(g2)
serves (λx. f1 x / g1 x) ∈ Θ(λx. f2 x / g2 x)
by (subst (1 2) divide-inverse, intro landau-theta.mult) (simp-all add: bigtheta-inverse
assms)
lemma bigthetaI':
  assumes c1 > 0 c2 > 0
  assumes eventually (\lambda x. c1 * |g x| \leq |f x| \land |f x| \leq c2 * |g x|) at-top
  shows f \in \Theta(g)
apply (rule bigthetaI)
apply (rule landau-o.bigI[OF assms(2)]) using assms(3) apply (eventually-elim, simp)
apply (rule landau-omega.bigI[OF assms(1)]) using assms(3) apply (eventually-elim, simp)
done

lemma eventually-nonzero-bigtheta [simp]:
  assumes (f :: ('a :: order \Rightarrow 'b :: linordered-field)) \in \Theta(g)
  shows eventually (\lambda x. f x \neq 0) at-top \iff eventually (\lambda x. g x \neq 0) at-top
proof -
  { fix f g :: 'a \Rightarrow 'b assume A: f \in \Theta(g) and B: eventually (\lambda x. f x \neq 0) at-top
    from A guess c1 c2 unfolding bigtheta-def by (elim landau-o.bigE landau-omega.bigE IntE)
    from B this(2,4) have eventually (\lambda x. g x \neq 0) at-top by eventually-elim auto }
with assms show ?thesis by (force simp: bigtheta-sym)
qed

lemma eventually-nonzero-bigtheta':
  f \in \Theta(g) \Rightarrow eventually-nonzero f \iff eventually-nonzero g
unfolding eventually-nonzero-bigtheta-def by (rule eventually-nonzero-bigtheta)

lemma bigthetaI-cong: eventually (\lambda x. f x = g x) at-top \Rightarrow f \in \Theta(g)
by (intro bigthetaI[OF 1 1]) (auto elim!: eventually-elim1)

lemma bigtheta-mult-eq:
  fixes f g :: ('a :: order \Rightarrow 'b :: linordered-field)
  shows \Theta(\lambda x. f x * g x) = \Theta(f) * \Theta(g)
proof (intro equalityI subsetI)
  fix h assume h \in \Theta(f) * \Theta(g)
  thus h \in \Theta(\lambda x. f x * g x)
  by (elim set-times-elim, hypsubst, unfold func-times) (erule (1) landau-theta.mult)
next
  fix h assume h \in \Theta(\lambda x. f x * g x)
  then guess c1 c2 :: 'b unfolding bigtheta-def by (elim landau-o.bigE landau-omega.bigE IntE)
  note c = this

  def h1 \equiv \lambda x. if g x = 0 then if f x = 0 then if h x = 0 then h x else 1 else f x
  else h x / g x
  def h2 \equiv \lambda x. if g x = 0 then if f x = 0 then h x else h x / f x else g x

27
have \( h = h_1 \ast h_2 \) by (intro ext) (auto simp: h1-def h2-def field-simps)
mOREOVER have \( h_1 \in \Theta(f) \)
proof (rule bigthetaI)
  from \( c(3) \) show \( \min c_2 \ast 1 > 0 \) by simp
  from \( c(1) \) show \( \max c_1 \ast 1 > 0 \) by simp
  from \( c(2,4) \)
  show eventually \( (\lambda x. \min c_2 1 \ast |f x| \leq |h_1 x| \land |h_1 x| \leq \max c_1 1 \ast |f x|) \)
at-top
  apply eventually-elim
  proof (rule conjI)
    fix \( x \) assume \( A: \ |h_1 x| \leq \min c_2 1 \ast |f x| \) and \( B: \ |h_1 x| \geq c_2 1 \ast |f x| \)
    have \( m: \ \min c_2 1 \ast |f x| \leq 1 \ast |f x| \) by (rule mult-right-mono) simp-all
    have \( \min c_2 1 \ast |f x \ast g x| \leq c_2 1 \ast |f x \ast g x| \) by (intro mult-right-mono)
    simp-all
  also note \( \text{A} \)
  finally show \( |h_1 x| \geq \min c_2 1 \ast |g x| \) using \( m \ A \)
    by (cases \( g x = 0 \) ) (simp-all add: h1-def abs-mult field-simps)+
  have \( m: \ 1 \ast |f x| \leq \max c_1 1 \ast |f x| \) by (rule mult-right-mono) simp-all
  note \( \text{A} \)
  also have \( c_1 \ast |f x \ast g x| \leq \max c_1 1 \ast |f x \ast g x| \) by (intro mult-right-mono)
  simp-all
  finally show \( |h_1 x| \leq \max c_1 1 \ast |g x| \) using \( m \ A \)
    by (cases \( g x = 0 \) ) (simp-all add: h1-def abs-mult field-simps)+
  qed
  qed
moreover have \( h_2 \in \Theta(g) \)
proof (rule bigthetaI)
  from \( c(3) \) show \( \min c_2 \ast 1 > 0 \) by simp
  from \( c(1) \) show \( \max c_1 \ast 1 > 0 \) by simp
  from \( c(2,4) \)
  show eventually \( (\lambda x. \min c_2 1 \ast |g x| \leq |h_2 x| \land |h_2 x| \leq \max c_1 1 \ast |g x|) \)
at-top
  apply eventually-elim
  proof (rule conjI)
    fix \( x \) assume \( A: \ |h_2 x| \leq \min c_2 1 \ast |g x| \) and \( B: \ |h_2 x| \geq c_2 1 \ast |f x \ast g x| \)
    have \( m: \ \min c_2 1 \ast |f x| \leq 1 \ast |f x| \) by (rule mult-right-mono) simp-all
    have \( \min c_2 1 \ast |f x \ast g x| \leq c_2 1 \ast |f x \ast g x| \) by (intro mult-right-mono)
    simp-all
  also note \( \text{A} \)
  finally show \( |h_2 x| \leq \max c_1 1 \ast |g x| \) using \( m \ A \)
    by (cases \( g x = 0 \) ) (simp-all add: h2-def abs-mult field-simps)+
  qed
  qed
ultimately show $h \in \Theta(f) \ast \Theta(g)$ by blast

2.3 Landau symbols and limits

lemma `bigoI-tendsto-abs`:
fixes $f \ g :: \mathbb{R} \Rightarrow \mathbb{R}$
assumes $((\lambda x. |f x / g x|) \longrightarrow c)$ at-top
assumes eventually $(\lambda x. g x \neq 0)$ at-top
shows $f \in O(g)$

proof (rule `bigoI`)

using `(rule bigoI)`

thus eventually $(\lambda x. |f x| \leq (|c| + 1) \ast |g x|)$ at-top

unfolding `dist-real-def` using assms

proof eventually-elim

fix $x$ assume $x : \mathbb{R}$

have $|f x| - |c| \ast |g x| \leq |f x| - c \ast |g x|$

by (simp add: `abs-mult`[symmetric])

also from $g$ have $\ldots = |g x| \ast |f x / g x| - c$

by (simp add: algebra-simps)

also from $x$ have $|f x / g x| - c \leq 1$ by simp

hence $|g x| \ast |f x / g x| - c \leq |g x| \ast 1$

by (rule `mult-left-mono`) simp-all

finally show $|f x| \leq (|c| + 1) \ast |g x|$ by (simp add: algebra-simps)

qed
from \( \text{ev}(OF \ (c/2 > 0)) \) show eventually \((\lambda x. |f x| \geq c/2 * |g x|)\) at-top

proof (eventually-elim)
  fix \( x \) assume \( B: ||f x / g x| - c| < c/2 \)
  from \( B \) have \( g: g x \neq 0 \) by auto
  from \( B \) have \(-c/2 < -||f x / g x| - c| \) by simp
  also have \( \ldots \leq ||f x / g x| - c| \) by simp
  finally show \( |f x| \geq c/2 * |g x| \) using \( g \) by (simp add: field-simps abs-mult)
qed

lemma bigomegaI-tendsto:
fixes \( f :: ('a::linorder) \Rightarrow \text{real} \)
assumes c-not-0: \((c :: \text{real}) \neq 0\)
assumes lim: \((\lambda x. f x / g x) \to c)\) at-top
shows \( f \in \Omega(g) \)
using assms by (intro bigomegaI-tendsto-abs[of c] tendsto-rabs) simp-all

lemma smallomegaI-filterlim-at-top:
fixes \( f :: - \Rightarrow \text{real} \)
assumes lim: \( \text{LIM x at-top.} \ |f x / g x| :> \text{at-top} \)
shows \( f \in \omega(g) \)
proof (rule landau-omega.smallI)
  fix \( c :: \text{real} \) assume c-pos: \( c > 0 \)
  from \( \text{lim} \) have ev: eventually \((\lambda x. |f x / g x| \geq c)\) at-top
    by (subst (asm) filterlim-at-top) simp
  thus eventually \((\lambda x. |f x| \geq c * |g x|) \)
proof eventually-elim
  fix \( x \) assume \( A: |f x / g x| \geq c \)
  from \( A \) c-pos have \( g x \neq 0 \) by auto
  with \( A \) show \( |f x| \geq c * |g x| \) by (simp add: field-simps)
qed

lemma smallomegaI-filterlim-at-top':
fixes \( f :: - \Rightarrow \text{real} \)
assumes lim: \( \text{LIM x at-top.} \ f x / g x :> \text{at-top} \)
shows \( f \in \omega(g) \)
proof (rule smallomegaI-filterlim-at-top)
  from \( \text{lim} \) show \( \text{LIM x at-top.} \ |f x / g x| :> \text{at-top} \)
    by (rule filterlim-compose[OF filterlim-abs-real])
qed

lemma smallomegaD-filterlim-at-top:
fixes \( f :: - \Rightarrow \text{real} \)
assumes \( f \in \omega(g) \)
assumes eventually \((\lambda x. g x \neq 0)\) at-top
shows \( \text{LIM x at-top.} \ |f x / g x| :> \text{at-top} \)
proof (subst filterlim-at-top-\geq, clarify)
fix \(c \:: \real\) assume \(c > 0\)
from landau-omega.smallD[OF assms(1) this] assms(2)
  show eventually \((\lambda x. |f x / g x| \geq c)\) at-top by eventually-elim (simp add: field-simps)
qed

lemma smalloI-tendsto:
fixes \(f g : \cdot \Rightarrow \real\)
assumes lim: \(((\lambda x. f x / g x) \dashrightarrow 0)\) at-top
assumes eventually \((\lambda x. g x \neq 0)\) at-top
shows \(f \in o(g)\)
proof (rule landau-o.smallI)
fix \(c \:: \real\) assume \(c > 0\)
from c-pos and lim have ev: eventually \((\lambda x. |f x / g x| < c)\) at-top
  by (subst (asm) tendsto-iff) (simp add: dist-real-def)
with assms(2) show eventually \((\lambda x. |f x| \leq c * |g x|)\) at-top
  by eventually-elim (simp add: field-simps)
qed

lemma smalloD-tendsto:
assumes \(f \in o(g)\)
shows \(((\lambda x. f x / g x :: \real) \dashrightarrow 0)\) at-top
unfolding tendsto-iff
proof clarify
fix \(e :: \real\) assume \(e > 0\)
  hence \(e / 2 > 0\) by simp
from landau-o.smallD[OF assms this] show eventually \((\lambda x. dist (f x / g x) 0 < e)\) at-top
proof eventually-elim
  fix \(x\) assume \(|f x| \leq e / 2 * |g x|\)
    with \(e\) have \(dist (f x / g x) 0 \leq e / 2\)
      by (cases \(g x = 0\)) (simp-all add: dist-real-def field-simps)
    also from \(e\) have \(\ldots < e\) by simp
    finally show \(dist (f x / g x) 0 < e\) by simp
  qed
qed

lemma bigthetaI-tendsto-abs:
fixes \(f g \:: ('a::linorder) \Rightarrow \real\)
assumes c-not-0: \((c::real) \neq 0\)
assumes lim: \(((\lambda x. |f x / g x|) \dashrightarrow c)\) at-top
shows \(f \in \Theta(g)\)
proof (rule bigthetaI)
from c-not-0 have \(|c| > 0\) by simp
  with lim have eventually \((\lambda x. |f x / g x| - c| < |c|)\) at-top
    by (subst (asm) tendsto-iff) (simp add: dist-real-def)
  hence \(g\) eventually \((\lambda x. g x \neq 0)\) at-top by eventually-elim (auto simp add: field-simps)
qed

31
from \( \lim g \) show \( f \in O(g) \) by (rule bigO-tendsto-abs)
from c-not-0 and \( \lim \) show \( f \in \Omega(g) \) by (rule bigomegaI-tendsto-abs)
qed

lemma bigthetaI-tendsto:
fixes \( f, g \) :: ('a::linorder) ⇒ real
assumes c-not-0: \( c :: real \) \( \neq 0 \)
assumes lim: \( (λx. f x / g x) \longrightarrow c \) at-top
shows \( f \in Θ(g) \)
using assms by (intro bigthetaI-tendsto-abs[of \( |c| \)] tendsto-rabs) simp-all

lemma tendsto-add-smallo:
fixes \( f1, f2 \) :: ('a::order) ⇒ real
assumes \( f1 \longrightarrow a \) at-top
assumes \( f2 \in o(f1) \)
shows \( (λx. f1 x + f2 x) \longrightarrow a \) at-top
proof
(subst filterlim-cong[OF refl refl])
from landau-o.smallD[OF assms(2) zero-less-one]
  have eventually \( (λx. |f2 x| \leq |f1 x|) \) at-top by simp
  thus eventually \( (λx. f1 x + f2 x = f1 x * (1 + f2 x / f1 x)) \) at-top
  by eventually-elim (auto simp: field-simps)
next
from assms(1) show \( (λx. f1 x * (1 + f2 x / f1 x)) \longrightarrow a \) at-top
  by (force intro: tendsto-eq-intros smalloD-tendsto[OF assms(2)])
qed

lemma tendsto-diff-smallo:
fixes \( f1, f2 \) :: ('a::order) ⇒ real
shows \( f1 \longrightarrow a \) at-top
  ⇒ \( f2 \in o(f1) \)
  ⇒ \( (λx. f1 x - f2 x) \longrightarrow a \) at-top
using tendsto-add-smallo[of f1 a \( \lambda x. -f2 x \)] by simp

lemma tendsto-diff-smallo-iff:
fixes \( f1, f2 \) :: ('a::order) ⇒ real
assumes \( f2 \in o(f1) \)
shows \( (f1 \longrightarrow a \) at-top \( ↔ \ (\lambda x. f1 x + f2 x \longrightarrow a) \) at-top
proof
assume \( (λx. f1 x + f2 x) \longrightarrow a \) at-top
hence \( (λx. f1 x + f2 x - f2 x) \longrightarrow a \) at-top
  by (rule tendsto-diff-smallo) (simp add: landau-o.small.plus-absorb2 assms)
thus \( (f1 \longrightarrow a \) at-top by simp
qed (rule tendsto-add-smallo[of \( \lim - assms \)])

lemma tendsto-diff-smallo-iff:
fixes \( f1, f2 \) :: ('a::order) ⇒ real
assumes \( f2 \in o(f1) \)
shows \( f2 \in o(f1) \)
  ⇒ \( (f1 \longrightarrow a \) at-top \( ↔ \ (\lambda x. f1 x - f2 x \longrightarrow a) \) at-top
using tendsto-add-smallo-iff[of \( \lambda x. -f2 x f1 a \)] by simp
lemma tendsto-divide-smallo:
fixes f1 f2 g1 g2 :: ('a::order) ⇒ real
assumes ((λx. f1 x / g1 x) ----> a) at-top
assumes f2 ∈ o(f1) g2 ∈ o(g1)
assumes eventually (λx. g1 x ≠ 0) at-top
shows ((λx. (f1 x + f2 x) / (g1 x + g2 x)) ----> a) at-top (is (?f ----> -)
-
)
proof (subst tendsto-cong)
let ?f' = λx. (f1 x / g1 x) * (1 + f2 x / f1 x) / (1 + g2 x / g1 x)

have (?f' ----> a * (1 + 0) / (1 + 0)) at-top
by (rule tendsto-mult tendsto-divide tendsto-add assms tendsto-const
smalloD-tendsto[OF assms(2)] smalloD-tendsto[OF assms(3)]) simp
thus (?f' ----> a) at-top by simp

have (1/2::real) > 0 by simp
from landau-o.smallID[OF assms(2) this] landau-o.smallD[OF assms(3) this]
  have eventually (λx. |f2 x| ≤ |f1 x|/2) at-top
     eventually (λx. |g2 x| ≤ |g1 x|/2) at-top by simp-all
with assms(4) show eventually (λx. ?f x = ?f' x) at-top
proof eventually-elim
fix x assume A: |f2 x| ≤ |f1 x|/2 and B: |g2 x| ≤ |g1 x|/2 and C: g1 x ≠ 0
show "?f x = ?f' x"
proof (cases f1 x = 0)
assume D: f1 x ≠ 0
from D have f1 x + f2 x = f1 x * (1 + f2 x/f1 x) by (simp add: field-simps)
mOREOVER from C have g1 x + g2 x = g1 x * (1 + g2 x/g1 x) by (simp
add: field-simps)
ultimately have ?f x = (f1 x * (1 + f2 x/f1 x)) / (g1 x * (1 + g2 x/g1 x))
by (simp only:)
  also have ... = ?f' x by simp
  finally show ?thesis .
qed (insert A, simp)

qed

lemma bigo-powr:
fixes f :: ('a::order) ⇒ real
assumes f ∈ O(p) p ≥ 0
shows (λx. |f x| powr p) ∈ O(λx. |g x| powr p)
proof –
from assms(1) guess c by (elim landau-o.bigE landau-omega.bigE IntE)
note c = this
from c(2) assms(2) have eventually (λx. |f x| powr p ≤ (c * |g x| powr p)
at-top
by (auto elim!: eventually-elim1 intro!: powr_mono2-ex)
thus (λx. |f x| powr p) ∈ O(λx. |g x| powr p) using c(1)
by (intro bigO [af - c powr p]) (simp-all add: powr-mult)
qed
assume eventually \((\lambda x. f x = 0)\) \at-top

hence \(\forall c > 0.\) eventually \((\lambda x. |f x| \leq c \cdot |0|)\) \at-top \by simp

thus \(f \in o(\lambda x . 0)\) \unfolding \smallo-def \by simp

\qed

lemma \in-bigo-zero-iff [simp]: \(f \in O(\lambda x . 0) \iff\) eventually \((\lambda x . f x = 0)\) \at-top

proof

assume \(f \in O(\lambda x . 0)\)

thus eventually \((\lambda x . f x = 0)\) \at-top \by (elim \landau-o \bigE) simp

next

assume eventually \((\lambda x . f x = 0)\) \at-top

hence eventually \((\lambda x . |f x| \leq 1 \cdot |0|)\) \at-top \by simp

thus \(f \in O(\lambda x . 0)\) \by (intro \landau-o \bigI[of 1]) simp-all

\qed

lemma \zero-in-smallomega-iff [simp]: \((\lambda x . 0) \in \omega(f) \iff\) eventually \((\lambda x . f x = 0)\) \at-top

\by (simp add: smallomega-iff-small)

lemma \zero-in-bigomega-iff [simp]: \((\lambda x . 0) \in \Omega(f) \iff\) eventually \((\lambda x . f x = 0)\) \at-top

\by (simp add: bigomega-iff-bigo)

lemma \zero-in-bigtheta-iff [simp]: \((\lambda x . 0) \in \Theta(f) \iff\) eventually \((\lambda x . f x = 0)\) \at-top

\unfolding bigtheta-def \by simp

lemma \in-bigtheta-zero-iff [simp]: \(f \in \Theta(\lambda x . 0) \iff\) eventually \((\lambda x . f x = 0)\) \at-top

\unfolding bigtheta-def \by simp

lemma \cmult-in-bigo-iff [simp]: \((\lambda x . c \cdot f x) \in O(g) \iff c = 0 \lor f \in O(g)\)

and \cmult-in-bigo-iff' [simp]: \((\lambda x . f x \cdot c) \in O(g) \iff c = 0 \lor f \in O(g)\)

and \cmult-in-smallomega-iff [simp]: \((\lambda x . c \cdot f x) \in o(g) \iff c = 0 \lor f \in o(g)\)

and \cmult-in-smallomega-iff' [simp]: \((\lambda x . f x \cdot c) \in o(g) \iff c = 0 \lor f \in o(g)\)

\by (cases c = 0, simp, simp)+

lemma \bigo-const [simp]: \((\lambda x . c) \in O(\lambda x . 1)\) \by (rule \bigoI[of \cdot |c|]) simp

lemma \bigo-const iff [simp]: \((\lambda x .::::\linorder . c1) \in O(\lambda x . c2) \iff c1 = 0 \lor c2 \neq 0\)

\by (auto simp: eventually-at-top-linorder)

lemma \bigomega-const iff [simp]: \((\lambda x .::::\linorder . c1) \in \Omega(\lambda x . c2) \iff c1 \neq 0 \lor c2 = 0\)

\by (subst \bigomega-iff-bigo, subst \bigo-const iff) blast

35
lemma smallo-real-nat-transfer:
(f :: real ⇒ real) ∈ o(g) ⇒ (λx::nat. f (real x)) ∈ o(λx. g (real x))
by (force intro!: eventually-nat-real landau-o.smallI dest: landau-o.smallD)

lemma bigo-real-nat-transfer:
(f :: real ⇒ real) ∈ O(g) ⇒ (λx::nat. f (real x)) ∈ O(λx. g (real x))
by (elim landau-o.bigE, erule landau-o.bigI, erule eventually-nat-real)

lemma smallomega-real-nat-transfer:
(f :: real ⇒ real) ∈ ω(g) ⇒ (λx::nat. f (real x)) ∈ ω(λx. g (real x))
by (force intro!: eventually-nat-real landau-omega.smallI dest: landau-omega.smallD)

lemma bigomega-real-nat-transfer:
(f :: real ⇒ real) ∈ Ω(g) ⇒ (λx::nat. f (real x)) ∈ Ω(λx. g (real x))
by (elim landau-omega.bigE, erule landau-omega.bigI, erule eventually-nat-real)

lemma bigtheta-real-nat-transfer:
(f :: real ⇒ real) ∈ Θ(g) ⇒ (λx::nat. f (real x)) ∈ Θ(λx. g (real x))
unfolding bigtheta-def using bigo-real-nat-transfer bigomega-real-nat-transfer
by blast

lemmas landau-symbol-if-eq [simp]:
assumes landau-symbol L
shows L(λx::'a::linordered-semidom. if x = a then f x else g x) = L(g)
apply (rule landau-symbol.cong[OF assms])
using eventually-ge-at-top[of a + 1] less-add-one[of a] apply (auto elim!: eventually-elim1)
done

lemmas landau-symbols-if-eq [simp] = landau-symbol[THEN landau-symbol-if-eq]

lemma sum-in-smallo:
fixes f :: 'b :: linordered-field
assumes f ∈ o(h) g ∈ o(h)
shows (λx. f x + g x) ∈ o(h) (λx. f x - g x) ∈ o(h)
proof –
{ fix f g assume fg: f ∈ o(h) g ∈ o(h)
  have (λx. f x + g x) ∈ o(h)
  proof (rule landau-o.smallI)
    fix c :: 'b assume c > 0
    hence c/2 > 0 by simp
    from fg[THEN landau-o.smallD[OF - this]]
show eventually \((\lambda x. \vert f x + g x \vert) \leq c \ast \vert h x \vert)\) \textbf{at-top} by eventually-elim

simp-all

qed

\)

from this[of \(f\) \(g\)] this[of \(f\) \(\lambda x. -g x\)] assms

show \((\lambda x. f x + g x) \in o(h) \land (\lambda x. f x - g x) \in o(h)\) by simp-all

qed

\begin{proof}

- \textbf{lemma} \textit{sum-in-bigo:}

\begin{itemize}
  \item \textbf{fixes} \(f :: - \Rightarrow 'b :: \text{lorderd-field}\)
  \item \textbf{assumes} \(f \in O(h) \land g \in O(h)\)
  \item \textbf{shows} \((\lambda x. f x + g x) \in O(h) \land (\lambda x. f x - g x) \in O(h)\)
\end{itemize}

\begin{proof}

\begin{itemize}
  \item \textbf{fix} \(f\) \(g\) \textbf{assume} \(fg: f \in O(h) \land g \in O(h)\)
  \item \textbf{from} \(fg(1)\) \textbf{guess} \(c1\) by \textit{(elim landau-o-bigE)} \textbf{note} \(c1 = \text{this}\)
  \item \textbf{from} \(fg(2)\) \textbf{guess} \(c2\) by \textit{(elim landau-o-bigE)} \textbf{note} \(c2 = \text{this}\)
  \item \textbf{from} \(c1(2)\) \(c2(2)\) \textbf{have eventually} \((\lambda x. \vert f x + g x \vert) \leq (c1 + c2) \ast \vert h x \vert)\) \textbf{at-top}
    \begin{itemize}
      \item \textbf{by} eventually-elim \text{(simp-all add: algebra-simps)}
      \item \textbf{hence} \((\lambda x. f x + g x) \in O(h)\) \textbf{by} \textit{(rule bigoI)}
    \end{itemize}
  \item \textbf{from} this[of \(f\) \(g\)] this[of \(f\) \(\lambda x. -g x\)] assms
  \item \textbf{show} \((\lambda x. f x + g x) \in O(h) \land (\lambda x. f x - g x) \in O(h)\) by simp-all
\end{itemize}

\end{proof}

\end{proof}

\begin{proof}

\begin{itemize}
  \item \textbf{lemma} \textit{tendsto-ln-over-powr:}
  \item \textbf{assumes} \((a::\text{real}) \geq 0\)
  \item \textbf{shows} \((\lambda x. \ln x / x^a) \rightarrow 0)\) \textbf{at-top}
\end{itemize}

\begin{proof}

\begin{itemize}
  \item \textbf{rule} \textit{lnhospital-at-top-at-top}\)
  \item \textbf{from} \textit{assms} \textbf{show} \textit{LIM x at-top. x powr a} := at-top \textbf{by} \textit{(rule powr-at-top)}
  \item \textbf{show eventually} \((\lambda x. a \ast x powr (a - 1) \neq 0)\) \textbf{at-top}
    \begin{itemize}
      \item \textbf{using} eventually-\textit{gt-at-top[of 0::real]} \textbf{by} eventually-elim \text{(insert \textit{assms, simp})}
      \item \textbf{show eventually} \((\lambda x::\text{real.} (\ln has-real-derivative (inverse x)) (at x))\) \textbf{at-top}
        \begin{itemize}
          \item \textbf{using} eventually-\textit{gt-at-top[of 0::real]} \textit{DERIV-ln by} \textit{(elim eventually-\textit{eltim}) simp}
          \item \textbf{show eventually} \((\lambda x. ((\lambda x. x powr a) has-real-derivative a \ast x powr (a - 1)) (at x))\) \textbf{at-top}
            \begin{itemize}
              \item \textbf{using} eventually-\textit{gt-at-top[of 0::real]} \textit{DERIV-powr by} \textit{(elim eventually-\textit{eltim}) simp}
            \end{itemize}
        \end{itemize}
    \end{itemize}
  \item \textbf{have eventually} \((\lambda x. inverse a \ast x powr -a = inverse x / (a \ast x powr (a - 1)))\) \textbf{at-top}
    \begin{itemize}
      \item \textbf{using} eventually-\textit{gt-at-top[of 0::real]}
          \begin{itemize}
            \item \textbf{by} \textit{(elim eventually-\textit{eltim}) \text{(simp add: field-simps powr-divide2[\text{symmetric}] powr-minus)}}
          \end{itemize}
    \end{itemize}
  \item \textbf{moreover from} \textit{assms} \textbf{have} \((\lambda x. inverse a \ast x powr -a) \rightarrow 0)\) \textbf{at-top}
    \begin{itemize}
      \item \textbf{by} \textit{(intro tendsto-mul-right-zero tendsto-neg-powr filterlim-ident) simp-all}
    \end{itemize}
  \item \textbf{ultimately show} \((\lambda x. inverse x / (a \ast x powr (a - 1))) \rightarrow 0)\) \textbf{at-top}
    \begin{itemize}
      \item \textbf{by} \textit{(subst (asm) tendsto-cong) simp-all}
    \end{itemize}
\end{itemize}

\end{proof}

\end{proof}

37
lemma tendsto-ln-powr-over-powr: 
  assumes (a::real) > 0 b > 0 
  shows ((λx. ln x powr a / x powr b) ---→ 0) at-top 
proof- 
  have eventually (λx. ln x powr a / x powr b = (ln x / x powr (b/a)) powr a) at-top 
    using assms eventually-gt-at-top[of 1::real] 
    by (elim eventually-elim1) (simp add: powr-divide powr-powr) 
  moreover have eventually (λx. 0 < ln x / x powr (b/a)) at-top 
    using eventually-gt-at-top[of 1::real] by (elim eventually-elim1) simp 
  with assms have ((λx. (ln x / x powr (b/a)) powr a) ---→ 0) at-top 
    by (intro tendsto-zero-powrI tendsto-ln-over-powr) simp-all 
ultimately show ?thesis by (subst tendsto-cong) simp-all 
qed 

lemma tendsto-ln-powr-over-powr': 
  assumes b > 0 
  shows ((λx::real. ln x powr a / x powr b) ---→ 0) at-top 
proof (cases a ≤ 0) 
  assume a: a ≤ 0 
  show ?thesis 
  proof (rule tendsto-sandwich[of λ::real. 0]) 
    have eventually (λx. ln x powr a ≤ 1) at-top unfolding eventually-at-top-linorder 
      proof (intro allI exI impI) 
        fix x :: real assume x ≥ exp 1 
        from ln-mono[OF less_le_trans[OF this this]] have ln x ≥ 1 by simp 
        hence ln x powr a ≤ ln (exp 1) powr a using a by (intro powr-mono2') simp-all 
        thus ln x powr a ≤ 1 by simp 
      qed 
      thus eventually (λx. ln x powr a / x powr b ≤ x powr −b) at-top 
        by eventually-elim (insert a, simp add: field-simps powr-minus divide-right-mono) 
      qed (auto intro: filterlim-ident tendsto-neg-powr assms) 
      qed (intro tendsto-ln-powr-over-powr, simp-all add: assms) 
  qed 

lemma tendsto-ln-in-lin: 
  assumes (a::real) > 0 c > 0 
  shows ((λx. ln (a*x) / ln (c*x)) ---→ 1) at-top 
proof (rule lhospital-at-top-at-top) 
  show LIM x at-top. ln (c*x) ---→ 1 at-top 
    by (intro filterlim-compose[OF LIM at-top] filterlim-tendsto-pos-mult-at-top[OF tendsto-const]) 
      filterlim-ident assms(2)) 
  show eventually (λx. ((λx. ln (a*x)) has-real-derivative (inverse x)) (at x)) at-top 
    using eventually-gt-at-top[of inverse a] assms 
    by (auto elim!: eventually-elim1 intro!: derivative-eq-intros simp: field-simps) 
  show eventually (λx. ((λx. ln (c*x)) has-real-derivative (inverse x)) (at x)) at-top 
  qed 

38
at-top

using eventually-gt-at-top[of inverse c] assms
by (auto elim!: eventually-elim1 intr!: derivative-eq-intros simp: field-simps)
show ((λx::real. inverse x / inverse x) ----> 1) at-top
by (subst tendsto-cong[of - λ-. 1]) (simp-all add: eventually-not-equal)
qed (simp-all add: eventually-not-equal)

lemma tendsto-ln-powr-over-ln-powr:
assumes (a::real) > 0 c > 0
shows ((λx. ln (a*x) powr d / ln (c*x) powr d) ----> 1) at-top
proof
have eventually (λx. ln (a*x) powr d / ln (c*x) powr d) powr d) at-top
using assms eventually-gt-at-top[of max (inverse a) (inverse c)]
by (auto elim!: eventually-elim1 simp: powr-divide field-simps)
moreover have ((λx. (ln (a*x) / ln (c*x)) powr d) ----> 1) at-top using assms
by (intro tendsto-eq-rhs[OF tendsto-powr[OF tendsto-ln-over-ln tendsto-const]])
ultimately show ?thesis by (subst tendsto-cong)
qed

lemma tendsto-ln-powr-over-ln-powr':
c > 0 --->(λx::real. ln x powr d / ln (c*x) powr d) ----> 1) at-top
using tendsto-ln-powr-over-ln-powr[of 1 c d] by simp

lemma tendsto-ln-powr-over-ln-powr'':
a > 0 --->(λx::real. ln (a*x) powr d / ln x powr d) ----> 1) at-top
using tendsto-ln-powr-over-ln-powr[of - 1] by simp

lemma bigtheta-const-ln-powr [simp]: a > 0 --->(λx::real. ln (a*x) powr d) ∈ Θ(λx. ln x powr d)
by (intro bigthetal-tendsto[of 1] tendsto-ln-powr-over-ln-powr'') simp

lemma bigtheta-const-ln-pow [simp]: a > 0 --->(λx::real. ln (a*x) powr d) ∈ Θ(λx. ln x ^ d)
proof
assume A: a > 0
hence (λx::real. ln (a*x) powr d) ∈ Θ(λx. ln (a*x) powr real d)
by [subst bigtheta-sym, intro bigthetal-cong powr-realpow-eventually
filterlim-compose[OF ln-at-top]
filterlim-tendsto-pos-mult-at-top[OF tendsto-const - filterlim-ident]]
also from A have (λx. ln (a*x) powr real d) ∈ Θ(λx. ln x powr real d) by simp
also have (λx. ln x powr real d) ∈ Θ(λx. ln x ^ d)
by (intro bigthetal-cong powr-realpow-eventually filterlim-compose[OF ln-at-top]
filterlim-ident)
finally show ?thesis .
qed
lemma bigheta-const-ln [simp]: \( a > 0 \implies (\lambda x::\text{real. ln (a*x)}) \in \Theta(\lambda x, \text{ln } x) \)
using tendsto-ln-over-ln[of a 1] by (intro bighetaI-tendsto[of 1]) simp-all

context landau-symbol
begin

lemma mult-cancel-left:
assumes \( f1 \in \Theta(g1) \) and eventually \((\lambda x. g1 x \neq 0)\) at-top
notes [trans] = bigheta-trans1 bigheta-trans2
shows \((\lambda x. f1 x * f2 x) \in L(\lambda x. g1 x * g2 x) \iff f2 \in L(g2)\)
proof
assume A: \((\lambda x. f1 x * f2 x) \in L(\lambda x. g1 x * g2 x)\)
from assms have eventually \((\lambda x. f1 x \neq 0)\) at-top by simp
hence \( f2 \in \Theta(\lambda x. f1 x * f2 x / f1 x) \)
  by (intro bighetaI-cong) (auto elim!: eventually-elim1)
also from A assms have \((\lambda x. f1 x * f2 x / f1 x) \in L(\lambda x. g1 x * g2 x / f1 x)\)
  by (intro divide-right) simp-all
also from assms have \((\lambda x. g1 x * g2 x / f1 x) \in \Theta(\lambda x. g1 x * g2 x / g1 x)\)
  by (intro landau-theta.mult landau-theta.divide) (simp-all add: bigheta-sym)
also from assms have \((\lambda x. g1 x * g2 x / g1 x) \in \Theta(g2)\)
  by (intro bighetaI-cong) (auto elim!: eventually-elim1)
finally show \( f2 \in L(g2) \).
next
assume f2 \( \in L(g2) \)

hence \((\lambda x. f1 x * f2 x) \in L(\lambda x. f1 x * g2 x)\) by (rule mult-left)
also have \((\lambda x. f1 x * g2 x) \in \Theta(\lambda x. g1 x * g2 x)\)
  by (intro landau-theta.mult-right assms)
finally show \((\lambda x. f1 x * f2 x) \in L(\lambda x. g1 x * g2 x)\).
qed

lemma mult-cancel-right:
assumes \( f2 \in \Theta(g2) \) and eventually \((\lambda x. g2 x \neq 0)\) at-top
shows \((\lambda x. g1 x * g2 x) \in L(\lambda x. f1 x * g2 x) \iff f1 \in L(g1)\)
by (subst (1 2) mult-commute) (rule mult-cancel-left[OF assms])

lemma divide-cancel-right:
assumes \( f2 \in \Theta(g2) \) and eventually \((\lambda x. g2 x \neq 0)\) at-top
shows \((\lambda x. f1 x / g2 x) \in L(\lambda x. g1 x / g2 x) \iff f1 \in L(g1)\)
by (subst (1 2) divide-inverse, intro mult-cancel-right bigheta-inverse) (simp-all add: assms)

lemma divide-cancel-left:
assumes \( f1 \in \Theta(g1) \) and eventually \((\lambda x. g1 x \neq 0)\) at-top
shows \((\lambda x. f1 x / f2 x) \in L(\lambda x. g1 x / f2 x) \iff (\lambda x. \text{inverse (f2 x)}) \in L(\lambda x. \text{inverse (g2 x)})\)
by (simp only: divide-inverse mult-cancel-left[OF assms])

40
lemma powr-smallo-iff:
assumes filterlim g at-top at-top
shows \((\lambda x::'a::linorder. g x powr p :: real) \in o(\lambda x. g x powr q) \iff p < q\)
proof
from assms have eventually \((\lambda x. g x \geq 1)\) at-top by (force simp: filterlim-at-top)
hence have B: \((\lambda x. g x powr q) \in O(\lambda x. g x powr p) \implies (\lambda x. g x powr p) \notin o(\lambda x. g x powr q)\)
  proof
  assume \((\lambda x. g x powr q) \in O(\lambda x. g x powr p)\)
  from landau-o.big-small-asymmetric[OF this] have eventually \((\lambda x. g x = 0)\)
at-top by simp
  with A have eventually \((\lambda::'a. False)\) at-top by eventually-elim simp
  thus False by force
qed

lemma powr-bigo-iff:
assumes filterlim g at-top at-top
shows \((\lambda x::'a::linorder. g x powr p :: real) \in O(\lambda x. g x powr q) \iff p \leq q\)
proof (cases p q rule: linorder-cases)
  assume p < q
  hence \((\lambda x. g x powr p) \in o(\lambda x. g x powr q)\)
  by (auto intro!: smalloI-tendsto tendsto-neg-powr simp: powr-divide2)
  with \(p < q\) show \(?thesis\) by auto
next
  assume p = q
  hence \((\lambda x. g x powr q) \in O(\lambda x. g x powr p)\)
  by (auto intro!: bithetaD1)
  with B \(p = q\) show \(?thesis\) by auto
next
  assume p > q
  hence \((\lambda x. g x powr q) \in O(\lambda x. g x powr p)\)
  by (auto intro!: smalloI-tendsto tendsto-neg-powr landau-o.small-imp-big simp: powr-divide2)
  with B \(p > q\) show \(?thesis\) by auto
qed

lemma powr-landau-iff:
assumes filterlim g at-top at-top
shows \((\lambda x::'a::linorder. g x powr p :: real) \in \omega(\lambda x. g x powr q) \iff p \leq q\)
proof
from assms have eventually \((\lambda x. g x \geq 1)\) at-top by (force simp: filterlim-at-top)
hence have B: \((\lambda x. g x powr q) \in \omega(\lambda x. g x powr p) \implies (\lambda x. g x powr p) \notin O(\lambda x. g x powr q)\)
  proof
  assume \((\lambda x. g x powr q) \in \omega(\lambda x. g x powr p)\)
  with \(p \leq q\) show \(?thesis\) by auto
qed
from landau-o.small-big-asymmetric[Of this] have eventually $(\lambda x. \ g \ x = 0)$
at-top by simp
  with $A$ have eventually $(\lambda x::'a. \ False)$ at-top by eventually-elim simp
  thus $False$ by force
qed
show ?thesis
proof (cases $p \ q$ rule: linorder-cases)
  assume $p < q$
  hence $(\lambda x. \ g \ x powr p) \in o((\lambda x. \ g \ x powr q))$ using assms $A$
    by (auto intro!: smalloI-tendsto tendsto-neg-powr simp: powr-divide2)
  with $(p < q)$ show ?thesis by (auto intro!: landau-o.small-imp-big)
next
  assume $p = q$
  hence $(\lambda x. \ g \ x powr q) \in O((\lambda x. \ g \ x powr p))$ by (auto intro!: bigthetaD1)
  with $(p = q)$ show ?thesis by auto
next
  assume $p > q$
  hence $(\lambda x. \ g \ x powr q) \in o((\lambda x. \ g \ x powr p))$ using assms $A$
    by (auto intro!: smalloI-tendsto tendsto-neg-powr simp: powr-divide2)
  with $(p > q)$ show ?thesis by (auto intro!: landau-o.small-imp-big)
qed
qed

lemma pourr-bigtheta-iff:
  assumes $filterlim \ g \ at-top \ at-top$
  shows $(\lambda x::'a::linorder. \ g \ x powr p :: real) \in \Theta((\lambda x. \ g \ x powr q)) \longleftrightarrow p = q$
  using assms unfolding bigtheta-def by (auto simp: bigomega-iff-bigo powr-bigo-iff)

2.4 Rewriting Landau symbols

Since the simplifier does not currently rewriting with relations other than
equality, but we want to rewrite terms like $\Theta(\lambda x. \ log \ 2 \ x * x)$ to $\Theta(\lambda x. \ ln \ x * x)$, we need to bring the term into something that contains $\Theta(log \ 2)$ and
$\Theta(\lambda x. \ x)$, which can then be rewritten individually. For this, we introduce
the following constants and rewrite rules. The rules are mainly used by the
simprocs, but may be useful for manual reasoning occasionally.

definition set-mult $A \ B$ = $(\lambda x. \ f \ x * g \ x | f \ g. \ f \in A \land g \in B)$
definition set-inverse $A$ = $(\lambda x. \ inverse \ (f \ x) | f. \ f \in A)$
definition set-divide $A \ B$ = $(\lambda x. \ f \ x / g \ x | f \ g. \ f \in A \land g \in B)$
definition set-pow $A \ n$ = $(\lambda x. \ f \ x ^ n | f. \ f \in A)$
definition set-powr $A \ y$ = $(\lambda x. \ f \ x powr y | f. \ f \in A)$

lemma bigtheta-mult-eq-set-mult:
  fixes $f \ g :: ('a :: order) \Rightarrow ('b :: linordered-field)$
  shows $\Theta(\lambda x. \ f \ x * g \ x) \Rightarrow set-mult (\Theta(f)) (\Theta(g))$
  unfolding bigtheta-mult-eq set-mult-def set-times-def func-times by blast

lemma bigtheta-inverse-eq-set-inverse:

42
proof (by blast)
  next
  proof
  lemma set-divide-inverse qed ext blast

bigtheta-pow-eq-set-pow
  lemma
  next
  force

bigtheta-divide-eq-set-divide
  lemma
  qed blast

hence (g
  then obtain
  g h
  then obtain
  (bigtheta-pow A

fix
  g
  hence
  g
  fix
  f
  finally show
  (set-divide

also have
  f
  also from
  ⟨

show
  fixes
  f g
  with
  ⟨

also have
  f
  with
  ⟨

shows
  f
  fixes
  f g

show
  ⟩

also from
  (g = (λx. inverse (g′ x))):

have
  (λx. inverse (g′ x)) = g
  by
  (intro ext) simp

finally show
  g ∈ (Θ(λx. inverse (f x)))
.

qed


lemma set-divide-inverse:
  set-divide (A :: (\⇒ (:: division-ring)) set) B = set-mult A (set-inverse B)
proof (intro equality1 subsetI)
  fix f assume f ∈ set-divide A B
  then obtain g h where f = (λx. g x / h x) g ∈ A h ∈ B unfolding set-inverse-def
  by blast
  hence (λx. inverse (h x)) (λx. inverse (h x)) ∈ set-inverse B
  unfolding set-inverse-def by (auto simp: divide-inverse)
  with \( g \in A \) show f ∈ set-mult A (set-inverse B) unfolding set-mult-def by force
next
  fix f assume f ∈ set-mult A (set-inverse B)
  then obtain g h where f = g * (λx. inverse (h x)) g ∈ A h ∈ B
  unfolding set-times-def set-inverse-def set-mult-def by force
  hence f = (λx. g x / h x) by (intro ext) (simp add: divide-inverse)
  with \( g \in A \) \( h \in B \) show f ∈ set-divide A B unfolding set-inverse-def by blast
qed

 lemma bigtheta-divide-eq-set-divide:
  fixes f g :: ('a :: order) ⇒ ('b :: linordered-field)
  shows \( Θ(λx. f x / g x) = set-divide (Θ(f)) (Θ(g)) \)
  by (simp only: set-divide-inverse divide-inverse bigtheta-mult-eq-set-mult
  bigtheta-inverse-eq-set-inverse)

primrec bigtheta-pow where
  bigtheta-pow A 0 = Θ(λx. 1)
| bigtheta-pow A (Suc n) = set-mult A (bigtheta-pow A n)

lemma bigtheta-pow-eq-set-pow: \( \Theta(λx. f x ^ n) = bigtheta-pow \ (Θ(f)) \ n \)
  by (induction n) (simp-all add: bigtheta-mult-eq-set-mult)

43
definition bigtheta-powr where

bigtheta-powr $A$ $y = (\text{if } y = 0 \text{ then } \{f. \exists g \in A. \text{ eventually-nonneg } g \land f \in \Theta(\lambda x. g \ x \ powr y)\}

   \text{else } \{f. \exists g \in A. \text{ eventually-nonneg } g \land (\forall x. |f x| = g \ x \ powr y)\})$

lemma bigtheta-powr-eq-set-powr:

assumes \(\text{eventually-nonneg } f\)

shows \(\Theta(\lambda x. f \ x \ powr (y::\text{real})) = \text{bigtheta-powr } (\Theta(f)) \ y\)

proof (cases \(y = 0\))

assume \([\text{simp}]: y = 0\)

thus \(?\text{thesis}\)

proof

fix \(h\) assume \(h \in \text{bigtheta-powr } \Theta(f) \ y\)

then obtain \(g\) where \(g: g \in \Theta(f) \ \text{eventually-nonneg } g \ h \in \Theta(\lambda x. g \ x \ powr 0)\)

unfolding bigtheta-powr-def by force

note \(\Theta(h)\)

also have \((\lambda x. g \ x \ powr 0) \in \Theta(\lambda x. g \ x \ powr 0)\) using \(\text{assms}\)

unfolding eventually-nonneg-def

by (intro bigthetaI-cong) (auto elim!: eventually-elim1)

also from \((\lambda x. g \ x \ powr 0) \in \Theta(\lambda x. g \ x \ powr 0)\) by (rule bigtheta-powr)

also from \((\lambda x. f \ x \ powr 0) \in \Theta(\lambda x. f \ x \ powr 0)\) unfolding eventually-nonneg-def

by (intro bigthetaI-cong) (auto elim!: eventually-elim1)

finally show \(h \in \Theta(\lambda x. f \ x \ powr y)\) by simp

next

fix \(h\) assume \(h \in \Theta(\lambda x. f \ x \ powr y)\)

with \(\text{assms}\)

have \(\exists g \in \Theta(f). \ \text{eventually-nonneg } g \land h \in \Theta(\lambda x. g \ x \ powr 0)\)

by (intro \text{becl}[of - f] \ conjI) simp-all

thus \(h \in \text{bigtheta-powr } \Theta(f) \ y\) unfolding bigtheta-powr-def by simp

qed

next

assume \(y: y \neq 0\)

show \(?\text{thesis}\)

proof

fix \(h\) assume \(h \in \Theta(\lambda x. f \ x \ powr y)\)

let \(?h' = \lambda x. |h x| \ powr \text{inverse } y\)

from bigtheta-powr[OF \(h\), of inverse \(y\) \(y\)]

have \(?h' \in \Theta(\lambda x. f \ x \ powr 1)\) by (simp add: powr-powr)

also have \((\lambda x. f \ x \ powr 1) \in \Theta(f)\) using \(\text{assms}\)

unfolding eventually-nonneg-def

by (intro bigthetaI-cong) (auto elim!: eventually-elim1)

finally have \(?h' \in \Theta(f)\) .

with \(y\) have \(\exists g \in \Theta(f). \ \text{eventually-nonneg } g \land (\forall x. |h x| = g \ x \ powr y)\)

by (intro becl[of - ?h']) (simp-all add: powr-powr eventually-nonneg-def)

thus \(h \in \text{bigtheta-powr } \Theta(f) \ y\) using \(y\)

unfolding bigtheta-powr-def by simp

next

fix \(h\) assume \(h \in \text{bigtheta-powr } \Theta(f) \ y\)

with \(y\) obtain \(g\) where \(A: g \in \Theta(f) \ \forall x. |h x| = g \ x \ powr y \text{ eventually-nonneg}\)
unfolding bigtheta-powr-def by force
from this(3) have \((\lambda x. g x powr y) \in \Theta(\lambda x. |g x| powr y)\)
unfolding eventually-nonneg-def
by (intro bigthetaI-cong) (auto elim!: eventually-elim1)
also from \(A(1)\) have \((\lambda x. |g x| powr y) \in \Theta(\lambda x. |f x| powr y)\) by (rule bigtheta-powr)
also have \((\lambda x. |f x| powr y) \in \Theta(\lambda x. f x powr y)\)
using assms unfolding eventually-nonneg-def
by (intro bigthetaI-cong) (auto elim!: eventually-elim1)
finally have \((\lambda x. |h x|) \in \Theta(\lambda x. f x powr y)\)
by (subst \(A(2)\))
using simp
qed

lemmas bigtheta-factors-eq =
bigtheta-pow-eq-set-pow bigtheta-powr-eq-set-powr

lemmas landau-bigtheta-congs = landau-symbols[THEN landau-symbol.cong-bigtheta]

lemma \((\text{in landau-symbol})\) meta-cong-bigtheta: \(\Theta(f) \equiv \Theta(g) \Rightarrow L(f) \equiv L(g)\)
using bigtheta-refl[of \(f\)]
by (intro eq-reflection cong-bigtheta) blast

lemmas landau-bigtheta-meta-congs = landau-symbols[THEN landau-symbol.meta-cong-bigtheta]

end

3 Sorting and grouping factors

theory Group-Sort
imports Main ~~/src/HOL/Library/Multiset
begin

For the reification of products of powers of primitive functions such as \(\lambda x. x * (\ln x)^2\)
into a canonical form, we need to be able to sort the factors
according to the growth of the primitive function it contains and merge
terms with the same function by adding their exponents. The following
locale defines such an operation in a general setting; we can then instantiate
it for our setting.

The locale takes as parameters a key function \(f\) that sends list elements
into a linear ordering that determines the sorting order, a \(merge\) function
to merge to equivalent (w.r.t. \(f\)) elements into one, and a list reduction
function \(g\) that reduces a list to a single value. This function must be
invariant w.r.t. the order of list elements and be compatible with merging
of equivalent elements. In our case, this list reduction function will be the product of all list elements.

locale groupsort = 
  fixes \( f \) :: 
  'a ⇒ ('b::linorder)
  fixes merge :: 'a ⇒ 'a ⇒ 'a
  fixes g :: 'a list ⇒ 'c
  assumes f-merge: \( f x = f y \) ⇒ \( f \) (merge \( x \) \( y \)) = \( f x \)
  assumes g-cong: \( \text{multiset-of} \) \( x s \) = \( \text{multiset-of} \) \( y s \) ⇒ \( g \) \( x s \) = \( g \) \( y s \)
  assumes g-merge: \( f x = f y \) ⇒ \( g \) \[ \( x s \), \( y s \) \] = \( g \) \[ \( \text{merge} \) \( x s \) \( y s \) \]
  assumes g-append-cong: \( g \) \( x s \) = \( g \) \( y s \) ⇒ \( g \) \( x s \) \@ \( y s \) = \( g \) \( x s \) \@ \( y s \)

begin

context

private function part-aux ::
  'b ⇒ 'a list ⇒ ('a list) × ('a list) ⇒ ('a list) ⇒ ('a list) ⇒ ('a list)
where
  part-aux p [] (ls, eq, gs) = (ls, eq, gs)
| \( f x < p \) ⇒ part-aux p (x#xs) (ls, eq, gs) = part-aux p xs (x#ls, eq, gs)
| \( f x > p \) ⇒ part-aux p (x#xs) (ls, eq, gs) = part-aux p xs (ls, eq, x#gs)
| \( f x = p \) ⇒ part-aux p (x#xs) (ls, eq, gs) = part-aux p xs (ls, eq@[x], gs)

proof (clarify)
  case (goal1 P p xs ls eq gs)
  show ?case
    proof (cases xs)
      fix x xs' assume xs = x # xs'
      thus ?thesis using goal1 by (cases f x p rule: linorder-cases) auto
    qed (auto intro: goal1(1))
    qed simp-all

termination by (relation Wellfounded.measure (size o fst o snd)) simp-all

private lemma groupsort-locale: groupsort f merge g by unfold-locales

private lemmas part-aux-induct = part-aux.induct[split-format (complete), OF groupsort-locale]

private definition part where
  part p xs = part-aux (f p) xs ([]), [p], []

private lemma part: 
  part p xs = (rev (filter (λx. f x < f p) xs),
    p # filter (λx. f x = f p) xs, rev (filter (λx. f x > f p) xs))

proof−
  { 
    fix p xs ls eq gs
    have \( \text{fst} \) \( \text{part-aux} \) \( p s \) \( (ls, eq, gs) \) = \( \text{rev} \) \( \text{filter} \) \( (λx. f x < p) \) \( xs \) \@ \( ls \)
      by (induction p xs ls eq gs rule: part-aux-induct) simp-all
  } note A = this
{ fix p xs ls eq gs have snd (snd (part-aux p xs (ls, eq, gs))) = rev (filter (λx. f x > p) xs) @ gs by (induction p xs ls eq gs rule: part-aux-induct) simp-all }
} note B = this { fix p xs ls eq gs have fst (snd (part-aux p xs (ls, eq, gs))) = eq @ filter (λx. f x = p) xs by (induction p xs ls eq gs rule: part-aux-induct) auto }
} note C = this note ABC = A B C from ABC[of f p xs [[]] [p] []] assms show ?thesis unfolding part-def by (intro prod-eqI simp-all) qed

private function sort :: 'a list ⇒ 'a list where sort [] = [] | sort (x#xs) = (case part x xs of (ls, eq, gs) ⇒ sort ls @ eq @ sort gs) by pat-completeness simp-all termination by (relation Wellfounded.measure length) (simp-all add: part less-Suc-eq-le)

private lemma filter-mset-union: (⋀x. x ∈# A ⇒ P x ⇒ Q x ⇒ False) ⇒ filter-mset P A + filter-mset Q A = filter-mset (λx. P x ∨ Q x) A by (subst multiset-eq-iff) force

private lemma multiset-of-sort: multiset (sort xs) = multiset-of xs proof (induction xs rule: sort.induct) case [] x xs let ?M = λoper. {#y:# multiset-of xs. oper (f y) (f x)#} from goal2 have multiset-of (sort (x#xs)) = ?M (op <) + ?M (op =) + ?M (op >) + {#x#} by (simp add: part Multiset.union-assoc multiset-of-filter) also have ?M (op <) + ?M (op =) + ?M (op >) = multiset-of xs by ((subst filter-mset-union, force)+, subst multiset-eq-iff, force) finally show ?case by simp qed simp

private lemma g-sort: g (sort xs) = g xs by (intro g-cong multiset-of-sort)


private lemma sorted-all-equal: (⋀x. x ∈ set xs ⇒ x = y) ⇒ sorted xs by (induction xs) (auto simp: sorted-Cons)

private lemma sorted-sort: sorted (map f (sort xs)) apply (induction xs rule: sort.induct)
apply simp
apply (simp only: sorted-append sorted-Cons sort.simps part map-append split)
apply (intro conjI TrueI)
using sorted-map-same by (auto simp: set-sort sorted-Cons)

private fun group where
  group [] = []
| group (x#xs) = (case partition (\y. f y = f x) xs of (xs', xs'') ⇒
  fold merge xs' x # group xs'')

private lemma f-fold-merge: (\y. y ∈ set xs ⇒ f y = f x) ⇒ f (fold merge xs x) = f x
  by (induction xs rule: rev-induct) (auto simp: f-merge)

private lemma f-group: x ∈ set (group xs) ⇒ ∃x'∈set xs. f x = f x'
proof (induction xs rule: group.induct)
case (goal2 x xs)
  hence x = fold merge [y←xs. f y = f x'] x' ∨ x ∈ set (group [xa←xs. f xa ≠ f x'])
  by (auto simp: o-def)
thus ?case
proof
  assume x = fold merge [y←xs. f y = f x'] x'
  also have f ... = f x' by (rule f-fold-merge) simp
  finally show ?thesis by simp
next
  assume x ∈ set (group [xa←xs. f xa ≠ f x'])
  from goal2(1)(OF - this) have ∃x'∈set [xa←xs. f xa ≠ f x']. f x = f x' by (simp add: o-def)
  thus ?thesis by force
qed simp

private lemma sorted-group: sorted (map f xs) ⇒ sorted (map f (group xs))
proof (induction xs rule: group.induct)
case (goal2 x xs)
  { fix x' assume x': x' ∈ set (group [y←xs. f y ≠ f x])
    with f-group obtain x'' where x'': x'' ∈ set xs f x' = f x'' by force
    have f (fold merge [y←xs. f y = f x] x) = f x
      by (subst f-fold-merge) simp-all
    also from goal2(2) have ... ≤ f x' by (auto simp: sorted-Cons)
    finally have f (fold merge [y←xs. f y = f x] x) ≤ f x'.
  }
moreover from goal2(2) have sorted (map f (group [xa←xs. f xa ≠ f x]))
  by (intro goal2 sorted-filter) (simp add: sorted-Cons o-def)
ultimately show ?case by (simp add: o-def sorted-Cons)
proof (using assms)

private lemma distinct-group: distinct (map f (group xs))

proof (induction xs rule: group.induct)

case (goal2 x xs)

have distinct (map f (group [xa→ xs . f xa ≠ f x])) by (intro goal2) (simp-all add: o-def)

moreover have f (fold merge [y← xs . f y = f x] x) ∈ set (map f (group [xa← xs . f xa ≠ f x]))

by (rule notI, subst (asm) f-fold-cong) (auto dest: f-group)

ultimately show ?case by (simp add: o-def)

qed simp

private lemma g-fold-same:

assumes f z ∈ set xs ⇒ f z = f x

shows g (fold merge xs x # ys) = g (x # xs @ ys)

using assms

proof (induction xs arbitrary: x)

case (Cons y xs)

have g (x # y # xs @ ys) = g (y # x # xs @ ys) by (intro g-cong) (auto simp: add-ac)

also have y # x # xs @ ys = [y,x] @ xs @ ys by simp

also from Cons.prems have g ... = g ([merge y x] @ xs @ ys)

by (intro g-append-cong g-merge) auto

also have [merge y x] @ xs @ ys = merge y x # xs @ ys by simp

also from Cons.prems have g ... = g (fold merge xs (merge y x) # ys)

by (intro Cons.IL[symmetric]) (auto simp: f-merge)

also have ... = g (fold merge (y # xs) # ys) by simp

finally show ?case by simp

qed simp

private lemma g-group: g (group xs) = g xs

proof (induction xs rule: group.induct)

case (goal2 x xs)

have g (group (x # xs)) = g (fold merge [y← xs . f y = f x] x # group [xa← xs . f xa ≠ f x])

by (simp add: o-def)

also have ... = g (x # [y← xs . f y = f x] @ group [ya← xs . f y # f x])

by (intro g-fold-same) simp-all

also have ... = g ((x # [y← xs . f y = f x]) @ group [ya← xs . f y ≠ f x]) (is - = ?A)

by simp

also from goal2 have g (group [y← xs . f y ≠ f x]) = g [ya← xs . f y ≠ f x] by (simp add: o-def)

hence ?A = g ((x # [y← xs . f y = f x]) @ [ya← xs . f y # f x])

by (intro g-append-cong) simp-all

also have ... = g (x # xs) by (intro g-cong) simp-all

finally show ?case.

qed simp
function group-part-aux ::
'b ⇒ 'a list ⇒ ('a list) × 'a × ('a list) ⇒ ('a list) × 'a × ('a list)
where
  group-part-aux p [] (ls, eq, gs) = (ls, eq, gs)
| f x < p ⇒ group-part-aux p (x#xs) (ls, eq, gs) = group-part-aux p xs (x#ls, eq, gs)
| f x > p ⇒ group-part-aux p (x#xs) (ls, eq, gs) = group-part-aux p xs (ls, eq, x#gs)
| f x = p ⇒ group-part-aux p (x#xs) (ls, eq, gs) = group-part-aux p xs (ls, merge x eq, gs)

proof
clarify
case (goal1 P p xs ls eq gs)
show ?case
proof
  cases xs
  fix x xs'
  assume xs' = x # xs'
  thus ?thesis using goal1 by (cases f x p rule: linorder-cases) auto
qed (auto intro: goal1 (1))
qed simp-all

termination by (relation Wellfounded.measure (size o fst o snd)) simp-all
function group-sort :: 'a list ⇒ 'a list where

\[\text{group-sort} [\ ] = [\ ]\]
\[\text{group-sort} (x \# xs) = (\text{case group-part } x \ x s \text{ of } (ls, eq, gs) \Rightarrow \text{group-sort } ls @ eq \ # \text{group-sort } gs)\]

by pat-completeness simp-all
termination by (relation Wellfounded.measure length) (simp-all add: group-part less-Suc-eq-le)

private lemma group-append:

assumes \(\forall x y. x \in \text{set } xs \implies y \in \text{set } ys \implies f x \neq f y\)

shows \(\text{group } (xs @ ys) = \text{group } xs @ \text{group } ys\)

using assms

proof (induction xs arbitrary: ys rule: length-induct)
case (goal1 \(xs')\)
  hence IH: \(\forall x y. x \in \text{set } xs < length xs' \implies (\forall x y. x \in \text{set } xs \implies y \in \text{set } ys \implies f x \neq f y)\)
  \(\Rightarrow \text{group } (xs @ ys) = \text{group } xs @ \text{group } ys\) by blast

show ?case

proof (cases xs')
case (Cons x xs)
  note [simp] = this
  have \(\text{group } (xs' @ ys) = \text{fold merge } [y \leftarrow xs@ys . f y = f x] x \#
  \text{group } ([x @ xs . f x \neq f x] @ [x @ ys . f x \neq f x])\) by (simp add: o-def)
  also from goal1(2) have \([y \leftarrow xs@ys . f y = f x] = [y \leftarrow xs . f y = f x]\)
  by (force simp: filter-empty-conv)
  also from goal1(2) have \([x @ ys . f x \neq f x] = ys\) by (force simp: filter-id-conv)
  also have \(\text{group } ([x @ xs . f x \neq f x] @ ys) =
  \text{group } [x @ xs . f x \neq f x] @ \text{group } ys\) using goal1(2)
  by (intro IH) (simp-all add: less-Suc-eq-le)
  finally show ?thesis by (simp add: o-def)

qed simp

qed

private lemma group-empty-iff [simp]: group xs = [] ←→ xs = []

by (induction xs rule: group.induct) auto

lemma group-sort-correct: group-sort xs = group (sort xs)

proof (induction xs rule: group-sort.induct)
case (goal2 \(x\) \(xs\))
  have \(\text{group-sort } (x \# xs) =
  \text{group-sort } (\text{rev } [x @ xs . f x < f x]) @ \text{group } (x \# [x @ xs . f x = f x]) @
  \text{group-sort } (\text{rev } [x @ xs . f x \neq f x])\) by (simp add: group-part)
  also have \(\text{group-sort } (\text{rev } [x @ xs . f x < f x]) = \text{group } (\text{sort } (\text{rev } [x @ xs . f
  x < f x]))\)
  by (rule goal2) (simp-all add: group-part)

51
also have group-sort (rev [xa←xs . f xa > f x]) = group (sort (rev [xa←xs . f xa > f x]))
by (rule goal2) (simp-all add: group-part)
also have group (x[#][xa←xs . f xa = f x]) @ group (sort (rev [xa←xs . f xa > f x])) =
group (((x[#][xa←xs . f xa = f x]) @ sort (rev [xa←xs . f xa > f x])))
by (intro group-append[symmetric]) (auto simp: set-sort)
also have group (sort (rev [xa←xs . f xa < f x])) @ ... =
group (sort (rev [xa←xs . f xa < f x]) @ (x[#][xa←xs . f xa = f x]) @
sort (rev [xa←xs . f xa > f x]))
by (intro group-append[symmetric]) (auto simp: set-sort)
also have sort (rev [xa←xs . f xa < f x]) @ (x[#][xa←xs . f xa = f x]) @
sort (rev [xa←xs . f xa > f x]) = sort (x # xs) by (simp add: part)
finally show ?case.
qed simp

lemma sorted-group-sort: sorted (map f (group-sort xs))
by (auto simp: group-sort-correct intro!: sorted-group sorted-sort)

lemma distinct-group-sort: distinct (map f (group-sort xs))
by (simp add: group-sort-correct distinct-group)

lemma g-group-sort: g (group-sort xs) = g xs
by (simp add: group-sort-correct g-group g-sort)

lemmas [simp del] = group-sort.simps group-part-aux.simps

end
end
end

4 Decision procedure for real functions

theory Landau-Real-Products
imports
  Main Group-Sort Landau-Symbols-Definition ~~/src/HOL/Library/Function-Algebras
begin
If there are two functions f and g where any power of g is asymptotically
smaller than f, propositions like (λx. (f x)^p1 * (g x)^q1) ∈ O(λx. (f x)^p2 *
(g x)^q2) can be decided just by looking at the exponents: the proposition is
true iff p1 < p2 or p1 = p2 ∧ q1 ≤ q2.
The functions λx. x, ln, λx. ln (ln x), ... form a chain in which every
function dominates all succeeding functions in the above sense, allowing
to decide propositions involving Landau symbols and functions that are
products of powers of functions from this chain by reducing the proposition to a statement involving only logical connectives and comparisons on the exponents.

We will now give the mathematical background for this and implement reification to bring functions from this class into a canonical form, allowing the decision procedure to be implemented in a simproc.

4.1 Decision procedure

definition powr-closure f ≡ \{λx. f x powr p :: real | p. True\}

lemma powr-closureI [simp]: (λx. f x powr p) ∈ powr-closure f
unfolding powr-closure-def by force

lemma powr-closureE: assumes g ∈ powr-closure f obtains p where g = (λx. f x powr p)
using assms unfolding powr-closure-def by force

locale landau-function-family =
fixes H :: (′a :: linorder ⇒ ′b :: linordered-field) set
assumes pos: h ∈ H ⇒ eventually (λx. h x > 0) at-top
assumes linear: h1 ∈ H ⇒ h2 ∈ H ⇒ h1 ∈ o(h2) ∨ h2 ∈ o(h1) ∨ h1 ∈ Θ(h2)
assumes mult: h1 ∈ H ⇒ h2 ∈ H ⇒ (λx. h1 x * h2 x) ∈ H
assumes inverse: h ∈ H ⇒ (λx. inverse (h x)) ∈ H

begin

lemma div: h1 ∈ H ⇒ h2 ∈ H ⇒ (λx. h1 x / h2 x) ∈ H
by (subst divide-inverse) (intro mult inverse)

lemma nonzero: h ∈ H ⇒ eventually (λx. h x ≠ 0) at-top
by (drule pos) (auto elim: eventually-elim1)

lemma landau-cases:
assumes h1 ∈ H h2 ∈ H
obtains h1 ∈ o(h2) | h2 ∈ o(h1) | h1 ∈ Θ(h2)
using linear[OF assms] by blast

lemma small-big-antisym:
assumes h1 ∈ H h2 ∈ H h1 ∈ o(h2) h2 ∈ O(h1) shows False
proof –
from nonzero[OF assms(1)] nonzero[OF assms(2)] landau-o.small-big-asymmetric[OF assms(3,4)]
  have eventually (λ-::′a. False) at-top by eventually-elim simp
thus False by force
qed
lemma small-antisym:
  assumes $h_1 \in H \land h_2 \in H \land h_1 \in o(h_2) \land h_2 \in o(h_1)$ shows False
  using assms by (blast intro: small-big-antisym landau-o.small-imp-big)
end

locale landau-function-family-pair =
  G!: landau-function-family $G + H$: landau-function-family $H$ for $G \land H$
  fixes $g$
  assumes gs-dominate: $g_1 \in G \Rightarrow g_2 \in G \Rightarrow h_1 \in H \Rightarrow h_2 \in H \Rightarrow g_1 \in o(g_2) \Rightarrow$
    $(\lambda x. \ g_1 \ x \ \ast \ h_1 \ x) \in o(\lambda x. \ g_2 \ x \ \ast \ h_2 \ x)$
  assumes $g$: $g \in G$
  assumes g-dominates: $h \in H \Rightarrow h \in o(g)$
begin

sublocale GH!: landau-function-family $G \ast H$
proof (unfold-locale; elim set-times-elim; hypsubst)
  fix $g \ h$ assume $g \in G \land h \in H$
  from $G.\ pos[OF \ this(1)] \land H.\ pos[OF \ this(2)]$ show eventually $(\lambda x. \ (g \ast h) \ x) > 0$
  at-top
  by eventually-elim simp
next
  fix $g \ h \ \ast \ h_2$ assume $A$: $g \in G \land g_2 \in G \land h_1 \in H \land h_2 \in H$
  from gs-dominate[OF this] gs-dominate[OF this[2,1,4,3]]
    $G.\ linear[OF \ this(1,2)] \land H.\ linear[OF \ this(3,4)]$
    show $g_1 \ast h_1 \in o(g_2 \ast h_2) \lor g_2 \ast h_2 \in o(g_1 \ast h_1) \lor g_1 \ast h_1 \in \Theta(g_2 \ast h_2)$
    by (elim disjE) (force simp: func-times bigomega-iff-bigo intro: landau-theta.mult
      landau-o.small.mult landau-o.small-big-mult landau-o.big-small-mult+)
    have $B$: $(\lambda x. \ (g_1 \ast h_1) \ x \ \ast \ (g_2 \ast h_2) \ x) = (g_1 \ast g_2) \ast (h_1 \ast h_2)$
    by (rule ext) (simp add: func-times mult-ac)
    from $A$ show $(\lambda x. \ (g_1 \ast h_1) \ x \ \ast \ (g_2 \ast h_2) \ x) \in G \ast H$
      by (sub B, intro set-times-intro) (auto intro: G.mult H.mult simp: func-times)
  qed

lemma smallo-iff:
  assumes $g_1 \in G \land g_2 \in G \land h_1 \in H \land h_2 \in H$
  shows $(\lambda x. \ g_1 \ x \ \ast \ h_1 \ x) \in o(\lambda x. \ g_2 \ x \ \ast \ h_2 \ x)$ if $g_1 \in o(g_2) \lor (g_1 \in \Theta(g_2) \land h_1 \in o(h_2))$
proof (rule G.landau-cases[OF assms[1,2]])
  assume $g_1 \in o(g_2)$

54
thus \( \text{thesis by (auto intro!: gs-dominates assms)} \)

next

assume \( A \): \( g1 \in \Theta(g2) \)
	hence \( B \): \( g2 \in O(g1) \) by (subst (asm) bigtheta-sym) (rule bigthetaD1)
	hence \( g1 \notin o(g2) \) using assms by (auto dest: G.small-big-antisym)

moreover from \( A \) have \( o(\lambda x. g2 x * h2 x) = o(\lambda x. g1 x * h2 x) \)

by (intro landau-o.small.cong-bigtheta landau-theta.mult-right subst bigtheta-sym)

ultimately show \( \text{thesis using } G\.nonzero[O \text{ assms}(1)] \) \( A \)

by (auto simp add: landau-o.small.mult-cancel-left)

next

assume \( A \): \( g2 \in o(g1) \)

from gs-dominates\(O \text{ assms}(2,1,4,3)\) this] have \( B \): \( g2 * h2 \in o(g1 * h1) \) by (simp add: func-times)

have \( g1 \notin o(g2) \) \( g1 \notin \Theta(g2) \) using assms

by (auto dest: G.small-antisym G.small-big-antisym simp: bigomega-iff-bigo)

moreover have \( \neg P \)

by (intro notI G.H.small-antisym\(O \text{ - - B}\) set-times-intro) (simp-all add: func-times assms)

ultimately show \( \text{thesis by blast} \)

next

assume \( A \): \( g1 \in G \) \( g2 \in G \) \( h1 \in H \) \( h2 \in H \)

shows \( (\lambda x. g1 x * h1 x) \in O(\lambda x. g2 x * h2 x) \)
	hus \( g1 \in o(g2) \lor (g1 \in \Theta(g2) \land h1 \in O(h2)) \) (is \( \neg P \) \( \neg Q \))

proof (rule G.\( \text{landau-cases}(O \text{ assms}(1,2)) \))

assume \( g1 \in o(g2) \)

thus \( \text{thesis by (auto intro!: gs-dominates assms landau-o.small-imp-big)} \)

next

assume \( A \): \( g2 \in o(g1) \)

hence \( g1 \notin O(g2) \) using assms by (auto dest: G.small-big-antisym)

moreover from gs-dominates\(O \text{ assms}(2,1,4,3) \) \( A \) have \( g2 * h2 \in o(g1 * h1) \)

by (simp add: func-times)

hence \( g1*h1 \notin O(g2*h2) \) by (blast intro: G.H.small-big-antisym assms)

ultimately show \( \text{thesis using } A \) assms

by (auto simp: func-times dest: landau-o.small-imp-big)

next

assume \( A \): \( g1 \in \Theta(g2) \)

hence \( g1 \notin o(g2) \) unfolding bigtheta-def using assms

by (auto dest: G.small-big-antisym simp: bigomega-iff-bigo)

moreover have \( O(\lambda x. g2 x * h2 x) = O(\lambda x. g1 x * h2 x) \)

by (subst landau-o.big.cong-bigtheta\(O \text{ landau-theta.mult-right}[O \text{ A}] \)) (rule refl)

ultimately show \( \text{thesis using } A \) G.nonzero\(O \text{ assms}(2) \)

by (auto simp: landau-o.big.mult-cancel-left)

next

assume \( A \): \( g1 \in G \Rightarrow g2 \in G \Rightarrow h1 \in H \Rightarrow h2 \in H \Rightarrow \)

lemma bigomega-iff:

\( g1 \in G \Rightarrow g2 \in G \Rightarrow h1 \in H \Rightarrow h2 \in H \Rightarrow \)
\[(\lambda x. g_1 x * h_1 x) \in \Theta(\lambda x. g_2 x * h_2 x) \iff q_1 \in \Theta(g_2) \land h_1 \in \Theta(h_2)\]

by (auto simp: bigheta-def bigo-iff bigomega-iff-bigo intro: landau-o.small-imp-big
  dest: G.small-antisym G.small-big-antisym)
end

lemma landau-function-family-powr-closure:
  assumes filterlim f at-top at-top
  shows  landau-function-family (powr-closure f)
proof (unfold-locales; elim powr-closureE; hypsubst)
  from assms have eventually \((\lambda x. f x \geq 1)\) at-top using filterlim-at-top by auto
  hence A: eventually \((\lambda x. f x \neq 0)\) at-top by eventually-elim simp
  
  fix \(p, q\) :: real
  show \((\lambda x. f x powr p) \in o(\lambda x. f x powr q)\) \lor
        \((\lambda x. f x powr q) \in o(\lambda x. f x powr p)\) \lor
        \((\lambda x. f x powr p) \in \Omega(\lambda x. f x powr q)\)
  by (cases p q rule: linorder-cases)
  (force intro!: smallof-tendsto tendsto-neg-powr simp: powr-divide2 assms A)+

  fix \(p\)
  show eventually \((\lambda x. f x powr p > 0)\) at-top using A by simp

lemma landau-function-family-pair-trans:
  assumes landau-function-family-pair F G f
  assumes landau-function-family-pair G H g
  shows  landau-function-family-pair F (G \times H) f
proof
  interpret FG: landau-function-family-pair F G f by fact
  interpret GH: landau-function-family-pair G H g by fact
  show \(?thesis\)
  proof (unfold-locales; (elim set-times-elims)?; (clarify)?; (unfold func-times mult.assoc[symmetric])?)
    fix \(f_1 f_2 g_1 g_2 h_1 h_2\)
    assume A: \(f_1 \in F \land f_2 \in F \land g_1 \in G \land g_2 \in G \land h_1 \in H \land h_2 \in H \land f_1 \in o(f_2)\)

    from A have \((\lambda x. f_1 x * g_1 x * h_1 x) \in o(\lambda x. f_1 x * g_1 x * g x)\)
      by (intro landau-o.small.mult-left GH.g-dominates)
    also have \((\lambda x. f_1 x * g_1 x * g x) = (\lambda x. f_1 x * (g_1 x * g x))\) by (simp only: mult.assoc)
    also from A have \(... \in o(\lambda x. f_2 x * (g_2 x / g x))\)
      by (intro FG.gs-dominate FG.H.mult FG.H.div GH.g)
    also from A have \((\lambda x. inverse (h_2 x)) \in o(g)\) by (intro GH.g-dominates GH.H.inverse)
    with GH.g A have \((\lambda x. f_2 x * (g_2 x / g x)) \in o(\lambda x. f_2 x * (g_2 x * h_2 x))\)
      by (auto simp: FG.H.nonzero GH.H.nonzero divide-inverse
        intro!: landau-o.small.mult-left intro: landau-o.small.inverse-flip)
also have ... = o(\lambda x. f_2 x \ast g_2 x \ast h_2 x) by (simp only: mult.assoc)
finally show (\lambda x. f_1 x \ast g_1 x \ast h_1 x) \in o(\lambda x. f_2 x \ast g_2 x \ast h_2 x) .

next
fix g_1 h_1 assume A: g_1 \in G h_1 \in H
hence (\lambda x. g_1 x \ast h_1 x) \in o(\lambda x. g_1 x \ast g x)
by (intro landau-o.small.mult-left GH.g-dominates)
also from A have (\lambda x. g_1 x \ast g x) \in o(f) by (intro FG.g-dominates FG.H.mult GH.g)
finally show (\lambda x. g_1 x \ast h_1 x) \in o(f) .
qed (simp-all add: FG.g)

lemma landau-function-family-pair-trans-powr:
assumes landau-function-family-pair (powr-closure g) H (\lambda x. g x powr 1)
assumes filterlim f at-top at-top
assumes \forall p. (\lambda x. g x powr p) \in o(f)
shows landau-function-family-pair (powr-closure f) (powr-closure g \ast H) (\lambda x. f x powr 1)

proof (rule landau-function-family-pair-trans[OF - assms(1)])
interpret GH!: landau-function-family-pair powr-closure g H \lambda x. g x powr 1 by fact
interpret F!: landau-function-family powr-closure f
by (rule landau-function-family-powr-closure) fact+
show landau-function-family-pair (powr-closure f) (powr-closure g) (\lambda x. f x powr 1)

proof (unfold-locales; (elim powr-closureE; hypsubst)?)
show (\lambda x. f x powr 1) \in powr-closure f by (rule powr-closureI)
next
fix p ::real
note assms(3)[of p]
also from assms(2) have eventually (\lambda x. f x \geq 1) at-top by (force simp: filterlim-at-top)
hence f \in \Theta(\lambda x. f x powr 1) by (auto intro!: bigthetaI-cong elim!: eventually-elim1)
finally show (\lambda x. g x powr p) \in o(\lambda x. f x powr 1) .

next
fix p p1 p2 p3 :: real
assume A: (\lambda x. f x powr p) \in o(\lambda x. f x powr p1)
have p: p < p1
proof (cases p p1 rule: linorder-cases)
assume p > p1
moreover from assms(2) have eventually (\lambda x. f x \geq 1) at-top
by (force simp: filterlim-at-top)
hence eventually (\lambda x. f x \neq 0) at-top by eventually-elim simp
ultimately have (\lambda x. f x powr p1) \in o(\lambda x. f x powr p) using assms
by (auto intro!: smalloI-tendsto tendsto-neg-powr simp: powr-divide2)
from F.small-antisym[OF - this A] show \thetathesis by (auto simp: powr-closureI)
next
assume p = p1
hence (\lambda x. f x powr p1) \in O(\lambda x. f x powr p) by (intro bigthetaD1) simp
with F.small-big-antisym[OF - - A this] show ?thesis by (auto simp: powr-closureI)
qed

from assms(2) have f-pos: eventually (\(\lambda\, f\, x \geq 1\)) at-top by (force simp: filterlim-at-top)
from assms have (\(\lambda\, g\, x\) powr \(((p2 - p3)/(p1 - p))\)) \(\in\) o(f) by simp
have (\(\lambda\, g\, x\) powr \((p2 - p3)\)) \(\in\) o(\(\lambda\, |f|\) powr \((p1 - p)\)) by (simp add: powr-powr)
hence (\(\lambda\, |f|\) powr \(p\) * \(g\) x powr \(p2\)) \(\in\) o(\(\lambda\, |f|\) powr \(p1\) * \(g\) x powr \(p3\))

using GH.G.zero[OF GH.g] F.\texttt{nonzero}[OF powr-closureI]
by (simp add: powr-divide2[symmetric] landau-o.small.divide-eq1 landau-o.small.divide-eq2 mult.commute)
also have (?P \(\iiff\) (\(\lambda\, f\, x\) powr \(p\) * \(g\) x powr \(p2\)) \(\in\) o(\(\lambda\, f\, x\) powr \(p1\) * \(g\) x powr \(p3\)))

using f-pos by (intro landau-o.small.cong-ex) (auto elim!: eventually-\texttt{elim1})
finally show (\(\lambda\, f\, x\) powr \(p\) * \(g\) x powr \(p2\)) \(\in\) o(\(\lambda\, f\, x\) powr \(p1\) * \(g\) x powr \(p3\))
qed

definition dominates :: ('a :: linorder \(\Rightarrow\) real) \(\Rightarrow\) (('a \(\Rightarrow\) real) \(\Rightarrow\) bool) where
dominates \(f\, g\) \(=\) (\(\forall\, p\,\) (\(\lambda\, g\, x\) powr \(p\)) \(\in\) o(\(f\)))

lemma dominates-trans:
assumes eventually (\(\lambda\, g\, x > 0\)) at-top
assumes dominates \(f\) \(g\) dominates \(g\) \(h\)
shows dominates \(f\) \(h\)
unfolding dominates-def
proof
  fix \(p\, :\, real\)
  from assms(3) have (\(\lambda\, h\, x\) powr \(p\)) \(\in\) o(\(g\)) unfolding dominates-def by simp
  also from assms(1) have \(g\) \(\in\) \(\Theta\)(\(\lambda\, g\, x\) powr \(1\))
  by (intro bigthetaI-cong) (auto elim!: eventually-\texttt{elim1})
  also from assms(2) have \(\lambda\, g\, x\) powr \(1\) \(\in\) o(\(f\)) unfolding dominates-def by simp
  finally show \((\lambda\, h\, x\) powr \(p\)) \(\in\) o(\(f\))
qed

fun landau-dominating-chain where
landau-dominating-chain (\(f\ #\ g\ #\ gs\)) \(\leftarrow\)
  dominates \(f\) \(g\) \(\land\) landau-dominating-chain \((g\ #\ gs)\)
| landau-dominating-chain \([f]\) \(\leftarrow\) \((\lambda\, 1)\) \(\in\) o(\(f\))
| landau-dominating-chain \([]\) \(\leftarrow\) True

primrec landau-dominating-chain' where
\[
\begin{align*}
\text{landau-dominating-chain}': [] & \mapsto \text{True} \\
| \text{landau-dominating-chain}': (f \# gs) & \mapsto \\
\text{landau-function-family-pair} (\text{powr-closure } f) (\text{listprod} (\text{map} \text{powr-closure } gs)) \\
(\lambda x. f x \text{ powr } 1) & \land \\
\text{landau-dominating-chain}' gs
\end{align*}
\]

\textbf{primrec nonneg-list where}
\[
\text{nonneg-list} [] \mapsto \text{True} \\
| \text{nonneg-list} (x\#xs) & \mapsto x > 0 \lor (x = 0 \land \text{nonneg-list} \ xs)
\]

\textbf{primrec pos-list where}
\[
\text{pos-list} [] \mapsto \text{False} \\
| \text{pos-list} (x\#xs) & \mapsto x > 0 \lor (x = 0 \land \text{pos-list} \ xs)
\]

\textbf{lemma dominating-chain-imp-dominating-chain':}
\[
(\forall g. g \in \text{set} \ gs \Rightarrow \text{filterlim } g \text{ at-top } \text{at-top}) \Rightarrow \\
\text{landau-dominating-chain } gs \Rightarrow \text{landau-dominating-chain}' gs
\]

\textbf{proof (induction gs rule: landau-dominating-chain.induct)}
\textbf{case (goal2 f)}
\textbf{then interpret } F: \text{landau-function-family powr-closure } f
\textbf{by (intro landau-function-family-powr-closure) simp-all}
\textbf{from goal2 have eventually (\lambda x. f x \geq 1) at-top by (force simp: filterlim-at-top)}
\textbf{hence } o(\lambda x. f x \text{ powr } 1) = o(\lambda x. f x)
\textbf{by (intro landau-o.small.cong) (auto elim!: eventually-elim1)}
\textbf{with goal2 have landau-function-family-pair (powr-closure } f) \{\lambda x. 1\} (\lambda x. f x \text{ powr } 1)
\textbf{by unfold-locales (auto intro: powr-closureI)}
\textbf{thus } ?\text{case by (simp add: one-fun-def)}
\textbf{next}
\textbf{case (goal1 f g gs)}
\textbf{from goal1 show } ?\text{case}
\textbf{by (auto intro!: landau-function-family-pair-trans-powr simp add: dominates-def)}
\textbf{qed simp}

\textbf{locale landau-function-family-chain =}
\textbf{fixes gs :: 'a list}
\textbf{fixes get-param :: 'a } \Rightarrow \text{real}
\textbf{fixes get-fun :: 'a } \Rightarrow \text{('b::linorder } \Rightarrow \text{real)}
\textbf{assumes gs-pos: } g \in \text{set} (\text{map} \text{get-fun } gs) \Rightarrow \text{filterlim } g \text{ at-top } \text{at-top}
\textbf{assumes dominating-chain: } \text{landau-dominating-chain} (\text{map} \text{get-fun } gs)
\textbf{begin}

\textbf{lemma dominating-chain': } \text{landau-dominating-chain}' (\text{map} \text{get-fun } gs)

\textbf{by (intro dominating-chain-imp-dominating-chain' gs-pos dominating-chain)
lemma gs-powr-0-eq-one:
  eventually $(\lambda x. (\prod g \leftarrow gs. \text{get-fun } g \text{ powr } 0)) = 1$ at-top
using gs-pos
proof (induction gs)
case (Cons g gs)
  from Cons have eventually $(\lambda x. \text{get-fun } g \text{ powr } 0 > 0)$ at-top by (auto simp: filterlim-at-top-dense)
moreover from Cons have eventually $(\lambda x. (\prod g \leftarrow gs. \text{get-fun } g \text{ powr } 0)) = 1$ at-top by simp
ultimately show ?case by eventually-elim simp
qed simp-all

lemma listmap-gs-in-listmap:
  $(\lambda x. (\prod g \leftarrow fs. \text{h } g \text{ powr } p g)) \in \text{listprod} (\text{map powr-closure (map h fs)})$
proof
  have $(\prod g \leftarrow fs. g x \text{ powr } p g) = (\prod g \leftarrow fs. (\lambda x. h g x \text{ powr } p g))$
  by (rule ext, induction fs)
simp-all
also have ... \in listprod (\text{map powr-closure (map h fs)})
  apply (induction fs)
  apply (simp add: fun-eq-iff)
  apply (simp only: list.map listprod.Cons, rule set-times-intro)
  apply simp-all
  done
finally show ?thesis .
qed simp-all

lemma smallo-iff:
  $(\lambda x. 1) \in o(\lambda x. (\prod g \leftarrow gs. \text{get-fun } g x \text{ powr } \text{get-param } g)) \iff \text{pos-list (map get-param gs)}$
proof
  have $(\lambda x. 1) \in o(\lambda x. (\prod g \leftarrow gs. \text{get-fun } g x \text{ powr } \text{get-param } g)) \iff ((\lambda x. (\prod g \leftarrow gs. \text{get-fun } g x \text{ powr } 0))) \in o(\lambda x. (\prod g \leftarrow gs. \text{get-fun } g x \text{ powr } \text{get-param } g))$
  by (rule sym, intro landau-o.small.in-cong gs-powr-0-eq-one)
also from assms gs-pos dominating-chain have ... \iff pos-list (map get-param gs)
proof (induction gs)
case goal1
  have $(\lambda x::'b. 1::real) \notin o(\lambda x. 1)$ by (auto dest!: landau-o.small-big-asymmetric)
  thus ?case by simp
next
case (goal2 g gs)
then interpret G: landau-function-family-pair powr-closure (get-fun g)
listprod (map powr-closure (map get-fun gs)) \lambda x. get-fun g x powr 1 by simp
from goal2 show ?case using listmap-gs-in-listmap[of get-fun - gs]
  by (simp-all add: G.smallo-iff listmap-gs-in-listmap powr-smallo-iff powr-bigtheta-iff
    del: powr-zero-eq-one)
qed
finally show ?thesis .
lemma bigo-iff:

\[ (\lambda \cdot 1) \in O(\lambda x. \prod_{g \leftarrow gs} \text{get-fun } g x \text{ powr get-param } g) \iff \text{nonnull } (\text{map get-param } gs) \]

proof

have \((\lambda \cdot 1) \in O(\lambda x. \prod_{g \leftarrow gs} \text{get-fun } g x \text{ powr get-param } g)) \iff \\
\((\lambda x. \prod_{g \leftarrow gs} \text{get-fun } g x \text{ powr } 0) \in O(\lambda x. \prod_{g \leftarrow gs} \text{get-fun } g x \text{ powr get-param } g)) \)

by (rule sym, intro landau-o.big-in-cong gs-powr-0-eq-one)

also from assms gs-pos dominating-chain' have \(\ldots \iff \text{nonnull } (\text{map get-param } gs) \)

proof (induction gs)

case\((\text{goal2 } g \text{ gs})\)

then interpret \(G\): landau-function-family-pair powr-closure \((\text{get-fun } g)\)

listprod \((\text{map powr-closure } (\text{map get-fun } gs)) \lambda x. \text{get-fun } g x \text{ powr } 1 \) by simp

from \(\text{goal2}\) show \(\text{?case}\) using listmap-gs-in-listmap[of get-fun - gs]

by (simp-all add: \(G\).bigo-iff listmap-gs-in-listmap powr-smallo-iff powr-bigtheta-iff)

by (simp-all add: \(G\).bigo-iff listmap-gs-in-listmap powr-smallo-iff powr-bigtheta-iff)

\(\text{del: powr-zero-eq-one}\)

qed (simp add: func-one)

finally show \(\text{?thesis}\).

qed

lemma bigtheta-iff:

\[ (\lambda \cdot 1) \in \Theta(\lambda x. \prod_{g \leftarrow gs} \text{get-fun } g x \text{ powr get-param } g) \iff \text{list-all } (\text{op=} \ 0) \]

\((\text{map get-param } gs)\)

proof

have \((\lambda \cdot 1) \in \Theta(\lambda x. \prod_{g \leftarrow gs} \text{get-fun } g x \text{ powr get-param } g)) \iff \\
\((\lambda x. \prod_{g \leftarrow gs} \text{get-fun } g x \text{ powr } 0) \in \Theta(\lambda x. \prod_{g \leftarrow gs} \text{get-fun } g x \text{ powr get-param } g)) \)

by (rule sym, intro landau-theta.in-cong gs-powr-0-eq-one)

also from assms gs-pos dominating-chain' have \(\ldots \iff \text{list-all } (\text{op=} \ 0) \) (map get-param gs)

proof (induction gs)

case\((\text{goal2 } g \text{ gs})\)

then interpret \(G\): landau-function-family-pair powr-closure \((\text{get-fun } g)\)

listprod \((\text{map powr-closure } (\text{map get-fun } gs)) \lambda x. \text{get-fun } g x \text{ powr } 1 \) by simp

from \(\text{goal2}\) show \(\text{?case}\) using listmap-gs-in-listmap[of get-fun - gs]

by (simp-all add: \(G\).bigo-iff listmap-gs-in-listmap powr-smallo-iff powr-bigtheta-iff)

\(\text{del: powr-zero-eq-one}\)

qed (simp add: func-one)

finally show \(\text{?thesis}\).

qed

end
lemma \textit{\text{fun-chain-at-top-at-top}}:
assumes \texttt{filterlim (f :: (\'a::order) \Rightarrow \'a) at-top at-top}
shows \texttt{filterlim (f ^^ n) at-top at-top}
by (induction \(n\)) (\text{auto intro: filterlim-ident filterlim-compose[\text{OF \texttt{assms}}]}\)

lemma \textit{\text{const-smallo-ln-chain}}: \((\lambda - . 1) \in o((\ln :: real \Rightarrow real) ^^ n)\)
proof (intro \texttt{smalloI-tendsto})
show \((\lambda x :: real. 1 / (\ln ^^ n) x) ----> 0) at-top
by (rule \texttt{tendsto-divide-0 \texttt{tendsto-const \texttt{filterlim-at-top-imp-at-infinity \texttt{fun-chain-at-top-at-top \texttt{ln-at-top}}}})\)

next
from \texttt{fun-chain-at-top-at-top[\text{OF \texttt{ln-at-top}, of \(n\)]}}
have eventually \((\lambda x :: real. (\ln ^^ n) x > 0) \texttt{at-top})\)
by \texttt{eventually-elim \texttt{simp-all}}
thus eventually \((\lambda x :: real. (\ln ^^ n) x \neq 0) \texttt{at-top})\)
qed

lemma \textit{\text{ln-fun-in-smallo-fun}}:
assumes \texttt{filterlim f at-top at-top}
shows \((\lambda x. \ln (f x) powr p :: real) \in o(f)\)
proof (rule \texttt{smalloI-tendsto})
have \((\lambda x. \ln x powr p / x powr 1 ----> 0) at-top\)
\texttt{simp}
moreover have eventually \((\lambda x. \ln x powr p / x powr 1 = \ln x powr p / x) at-top\)
using eventually-gt-at-top[of 0::real] by \texttt{eventually-elim \texttt{simp\texttt{cong}}}\)
ultimately have \((\lambda x. \ln x powr p / x) ----> 0) at-top\)
\texttt{by (subst (asm) \texttt{tendsto-conv})}\)
from this \texttt{assms} show \((\lambda x. \ln (f x) powr p / f x) ----> 0) at-top\)
\texttt{by (rule \texttt{filterlim-compose})}\)
from \texttt{assms} have eventually \((\lambda x. f x \geq 1) \texttt{at-top})\)
\texttt{by \texttt{simp add: \texttt{filterlim-at-top}})}\)
thus eventually \((\lambda x. f x \neq 0) \texttt{at-top})\)
\texttt{by \texttt{eventually-elim \texttt{simp}})}\)
qed

lemma \textit{\text{ln-chain-dominates}}: \(m > n \Longrightarrow \texttt{dominates ((\ln :: real \Rightarrow real) ^^ n) (\ln ^^ m)}\)
proof (rule \texttt{less-Suc-induct})
fix \(n\) show \(\texttt{dominates ((\ln :: real \Rightarrow real) ^^ n) (\ln ^^ (Suc \(n)) \texttt{unfolding \texttt{dominates-def}})}\)
\texttt{by (force intro: \texttt{ln-fun-in-smallo-fun \texttt{fun-chain-at-top-at-top \texttt{ln-at-top})}}\)
next
fix \(k m n\)
assume \texttt{A: \texttt{dominates ((\ln :: real \Rightarrow real) \^ k) (\ln ^^ m \texttt{dominates ((\ln :: real \Rightarrow real) \^ m) (\ln ^^ n))}}\)
from \texttt{fun-chain-at-top-at-top[\texttt{OF \texttt{ln-at-top}, of \(m\)]}}
have eventually \((\lambda x :: real. (\ln ^^ m) x > 0) \texttt{at-top})\)
\texttt{by \texttt{simp add: \texttt{filterlim-at-top-dense}}}\)
from this \texttt{A} show \(\texttt{dominates ((\ln :: real \Rightarrow real) \^ k) ((\ln :: real \Rightarrow real) \^ n)}\)
\texttt{by (rule \texttt{dominates-trans})}\)
qed

62
datatype primfun = LnChain nat

instantiation primfun :: linorder
begin
fun less-eq-primfun :: primfun ⇒ primfun ⇒ bool where
LnChain x ≤ LnChain y ⇔ x ≤ y
fun less-primfun :: primfun ⇒ primfun ⇒ bool where
LnChain x < LnChain y ⇔ x < y
instance
proof
  case (goal1 x y) show ?case by (induction x y rule: less-eq-primfun.induct) auto
next
  case (goal2 x) show ?case by (cases x) auto
next
  case (goal3 x) thus ?case
    by (induction x y rule: less-eq-primfun.induct, cases z) auto
next
  case (goal4 x y) thus ?case by (induction x y rule: less-eq-primfun.induct) auto
next
  case (goal5 x y) thus ?case by (induction x y rule: less-eq-primfun.induct) auto
qed
end

fun eval-primfun' :: - ⇒ - ⇒ real where
eval-primfun' (LnChain n) = (λx. (ln^n) x)

fun eval-primfun :: - ⇒ - ⇒ real where
eval-primfun (f, e) = (λx. eval-primfun' f x powr e)

lemma eval-primfun-altdef: eval-primfun f x = eval-primfun' (fst f) x powr snd f
  by (cases f) simp

fun merge-primfun where
merge-primfun (x::primfun, a) (y, b) = (x, a + b)

fun inverse-primfun where
inverse-primfun (x::primfun, a) = (x, −a)

fun powr-primfun where
powr-primfun (x::primfun, a) e = (x, e*a)

lemma primfun-cases:
  assumes (∀n e. P ( LnChain n, e))
  shows P x
  proof (cases x, hypsubst)
fix a b show P (a, b) by (cases a; hypsubst, rule assms)
qed

lemma eval-primfun'-at-top: filterlim (eval-primfun' f) at-top at-top
by (cases f) (auto intr!: fun-chain-at-top-at-top ln-at-top)

lemma primfun-dominates:
  f < g ⇒ dominates (eval-primfun' f) (eval-primfun' g)
by (elim less-primfun.elims; hypsubst) (simp-all add: ln-chain-dominates)

lemma eval-primfun-pos: eventually (λx::real. eval-primfun f x > 0) at-top
proof (cases f, hypsubst)
  fix f e
  from eval-primfun'-at-top have eventually (λx. eval-primfun f x > 0) at-top
      by (auto simp: filterlim-at-top-dense)
  thus eventually (λx::real. eval-primfun (f, e) x > 0) at-top by eventually-elim simp
qed

lemma eventually-nonneg-primfun: eventually-nonneg (eval-primfun f)
unfolding eventually-nonneg-def using eval-primfun-pos[of f] by eventually-elim simp

lemma eval-primfun-nonzero: eventually (λx. eval-primfun f x ≠ 0) at-top
using eval-primfun-pos[of f] by eventually-elim simp

lemma eval-merge-primfun:
  fst f = fst g ⇒
  eval-primfun (merge-primfun f g) x = eval-primfun f x * eval-primfun g x
by (induction f g rule: merge-primfun.induct) (simp-all add: powr-add)

lemma eval-inverse-primfun:
  eval-primfun (inverse-primfun f) x = inverse (eval-primfun f x)
by (induction f rule: inverse-primfun.induct) (simp-all add: powr-minus)

lemma eval-powr-primfun:
  eval-primfun (powr-primfun f e) x = eval-primfun f x powr e
by (induction f e rule: powr-primfun.induct) (simp-all add: powr-powr mult.commute)

definition eval-primfuns where eval-primfuns fs x = (∏ f∈fs. eval-primfun f x)

lemma eval-primfuns-pos: eventually (λx. eval-primfuns fs x > 0) at-top
proof
  have eventually (λx. ∀ f∈set fs. eval-primfun f x > 0) at-top
      by (intro eventually-ball-finite ballI eval-primfun-pos finite-set)
  thus
thus ?thesis unfolding eval-primfuns-def by eventually-elim (rule listprod-pos, auto)
qed

lemma eval-primfuns-nonzero: eventually (λx. eval-primfuns fs x ≠ 0) at-top
using eval-primfuns-pos[of fs] by eventually-elim simp

4.2 Reification

definition LANDAU-PROD' where
LANDAU-PROD' L c f = L(λx. c * f x)

definition LANDAU-PROD where
LANDAU-PROD L c1 c2 fs ←→ (λ-. c1) ∈ L(λx. c2 * eval-primfuns fs x)

definition BIGTHETA-CONST' where BIGTHETA-CONST' c = Θ(λx. c)

definition BIGTHETA-CONST where BIGTHETA-CONST c A = set-mult Θ(λ-. c) A

definition BIGTHETA-FUN where
BIGTHETA-FUN f = Θ(f)

lemma BIGTHETA-CONST'-tag: Θ(λx. c) = BIGTHETA-CONST' c using BIGTHETA-CONST'-def

lemma BIGTHETA-CONST-tag: Θ(f) = BIGTHETA-CONST 1 Θ(f)
by (simp add: BIGTHETA-CONST-def bigtheta-mult-eq-set-mult[symmetric])

lemma BIGTHETA-FUN-tag: Θ(f) = BIGTHETA-FUN f
by (simp add: BIGTHETA-FUN-def)

lemma set-mult-is-times: set-mult A B = A * B
unfolding set-mult-def set-times-def func-times by blast

lemma set-powr-mult:
assumes eventually-nonneg f and eventually-nonneg g
shows Θ(λx. (f x * g x :: real) powr p) = set-mult (Θ(λx. f x powr p)) (Θ(λx. g x powr p))

proof−
from assms have eventually (λx. f x ≥ 0) at-top eventually (λx. g x ≥ 0) at-top

by (simp-all add: eventually-nonneg-def)
hence eventually (λx. (f x * g x :: real) powr p = f x powr p * g x powr p) at-top
by eventually-elim (simp add: powr-mult)
hence Θ(λx. (f x * g x :: real) powr p) = Θ(λx. f x powr p * g x powr p)
by (rule landau-theta.cong)
also have ... = set-mult (Θ(λx. f x powr p)) (Θ(λx. g x powr p))
by (simp add: bigtheta-mult-eq-set-mult)
finally show ?thesis .
qed

lemma eventually-nonneg-bigtheta-pow-realpow:
Θ(λx. eval-primfun f x ^ e) = Θ(λx. eval-primfun f x powr real e)

65
using eval-primfun-pos[of $f$]
by (auto intro!: landau-theta.cong elim!: eventually-elim1 simp: powr-realpow)

lemma BIGTHETA-CONST-fold:
BIGTHETA-CONST ($c$::real) (BIGTHETA-CONST $d$ $A$) = BIGTHETA-CONST ($c$*$d$) $A$
bigtheta-pow (BIGTHETA-CONST $c$ $\Theta$(eval-primfun $pf$)) $k$ =
BIGTHETA-CONST ($c$ ^ $k$) $\Theta$(\lambda x. eval-primfun $pf$ $x$ powr $k$)
set-inverse (BIGTHETA-CONST $c$ $\Theta$(f)) = BIGTHETA-CONST (inverse $c$)

\(\Theta(\lambda x. inverse (f x))\)
set-mult (BIGTHETA-CONST $c$ $\Theta$(f)) (BIGTHETA-CONST $d$ $\Theta$(g)) =
BIGTHETA-CONST ($c$*$d$) $\Theta$(\lambda x. $f$ $x$*$g$ $x$)
BIGTHETA-CONST' ($c$::real) = BIGTHETA-CONST $c$ $\Theta$(\lambda_. $1$)
BIGTHETA-FUN ($f$::real$\Rightarrow$real) = BIGTHETA-CONST $1$ $\Theta$(f)
apply (simp add: BIGTHETA-CONST-def set-mult-is-times bigtheta-mult-eq-set-mult mult-ac)
apply (simp only: BIGTHETA-CONST-def bigtheta-mult-eq-set-mult symmetric)

lemma fold-fun-chain:
$g$ $x$ = ($g$ ^ 1 $x$) ($g$ ^ $m$) (($g$ ^ $n$) $x$) = ($g$ ^ ($m$+$n$)) $x$
by (simp-all add: funpow-add)

lemma reify-ln-chain-1:
$\Theta$(\lambda x. (ln ^ $n$) $x$) = $\Theta$(eval-primfun (LnChain $n$, $1$))
proof (intro landau-theta.cong)
have filterlim ((ln :: real $\Rightarrow$ real) ^ $n$) at-top at-top
  by (intro fun-chain-at-top-at-top ln-at-top)
  hence eventually (\lambda x::real. (ln ^ $n$) $x$ > 0) at-top using filterlim-at-top-dense
by auto
thus eventually (\lambda x. (ln ^ $n$) $x$ = eval-primfun (LnChain $n$, $1$) $x$) at-top
  by eventually-elim simp
qed

lemma reify-monom-1:
$\Theta$(\lambda x::real. $x$) = $\Theta$(eval-primfun (LnChain $0$, $1$))
proof (intro landau-theta.cong)
from eventually-gt-at-top[of $\theta$::real]
show eventually (\lambda x. $x$ = eval-primfun (LnChain $0$, $1$) $x$) at-top
  by eventually-elim simp
qed
lemma reify-monom-pow:
\[ \Theta(\lambda x::\text{real}. \ x \ ^ \ e) = \Theta(\text{eval-primfun} \ (\text{LnChain 0}, \ \text{real e})) \]
proof
  - have \[ \Theta(\text{eval-primfun} \ (\text{LnChain 0}, \ \text{real e})) = \Theta(\lambda x. \ x \ \text{powr} \ (\text{real e})) \] by simp
  also have \[ \ldots = \Theta(\lambda x. \ x \ ^ e) \] by (intro landau-theta.cong powr-realpow-eventually filterlim-ident)
  finally show \[ \text{thesis} \] ..
qed

lemma reify-monom-powr:
\[ \Theta(\lambda x::\text{real}. \ x \ \text{powr} \ e) = \Theta(\text{eval-primfun} \ (\text{LnChain 0}, \ e)) \]
by (rule landau-theta.cong simp)

lemmas reify-monom = reify-monom-1 reify-monom-pow reify-monom-powr

lemma reify-ln-chain-pow:
\[ \Theta(\lambda x. \ (\ln ^ ^ n \ x \ ^ e) = \Theta(\text{eval-primfun} \ (\text{LnChain n}, \ \text{real e})) \]
proof
  - have \[ \Theta(\text{eval-primfun} \ (\text{LnChain n}, \ \text{real e})) = \Theta(\lambda x. \ (\ln ^ ^ n \ x \ \text{powr} \ (\text{real e})) \] by simp
  also have \[ \ldots = \Theta(\lambda x. \ (\ln ^ ^ n \ x \ ^ e) \] by (intro landau-theta.cong powr-realpow-eventually fun-chain-at-top-at-top ln-at-top)
  finally show \[ \text{thesis} \] ..
qed

lemma reify-ln-chain-powr:
\[ \Theta(\lambda x. \ (\ln ^ ^ n \ x \ \text{powr} \ e) = \Theta(\text{eval-primfun} \ (\text{LnChain n}, \ e)) \]
by (intro landau-theta.cong simp)

lemmas reify-ln-chain = reify-ln-chain-1 reify-ln-chain-pow reify-ln-chain-powr

lemma numeral-power-Suc: \[ \text{numeral n} \ ^ \ Suc a = \text{numeral n} \ * \ \text{numeral n} \ ^ a \]
by (rule power.simps)

lemmas landau-product-preprocess =
one-add-one one-plus-numeral numeral-plus-one arith-simps numeral-power-Suc power-0
fold-fun-chain[where \ g = \ln] reify-ln-chain reify-monom

lemma LANDAU-PROD'-fold:
\[ \text{BIGTHETA-CONST} \ e \ \Theta(\lambda \ d) = \text{BIGTHETA-CONST} \ (e \ * \ d) \ \Theta(\text{eval-primfuns } []) \]
\[ \text{LANDAU-PROD'} \ c \ (\lambda \ . \ \lambda) = \text{LANDAU-PROD'} \ c \ (\text{eval-primfuns } []) \]
eval-primfun f = eval-primfuns \ [f]
eval-primfuns fs x * eval-primfuns gs x = eval-primfuns \ (fs \ @ \ gs) \ x
apply (simp only: \text{BIGTHETA-CONST-def set-mult-is-times eval-primfuns-def[abs-def]}

67
lemma inverse-listprod-field:
listprod (map (λx. inverse (f x)) xs) = inverse (listprod (map f xs :: - :: field list))
by (induction xs) simp-all

lemma landau-prod-meta-cong:
assumes landau-symbol L
assumes Θ(f) ≡ BIGTHETA-CONST c1 (Θ(eval-primfuns fs))
assumes Θ(g) ≡ BIGTHETA-CONST c2 (Θ(eval-primfuns gs))
sows f ∈ L(g) ≡ LANDAU-PROD L c1 c2 (map inverse-primfun fs @ gs)
proof −
interpret landau-symbol L by fact
have f ∈ L(g) ←→ (λx. c1 * eval-primfuns fs x) ∈ L(λx. c2 * eval-primfuns gs x)
using assms[2,3][symmetric] unfolding BIGTHETA-CONST-def
by (intro cong-ex-bigtheta) (simp-all add: bigtheta-mult-eq-set-mult[symmetric])
also have ... ←→ (λx. c1) ∈ L(λx. c2 * eval-primfuns gs x / eval-primfuns fs x)
by (simp-all add: eval-primfuns-nonzero divide-eq1)
finally show f ∈ L(g) ≡ LANDAU-PROD L c1 c2 (map inverse-primfun fs @ gs)
by (simp add: LANDAU-PROD-def eval-primfuns-def eval-inverse-primfun divide-inverse o-def inverse-listprod-field mult-ac)
qed

fun pos-primfun-list where
pos-primfun-list [] ←→ False
| pos-primfun-list ((-,x)#xs) ←→ x > 0 ∨ (x = 0 ∧ pos-primfun-list xs)

fun nonneg-primfun-list where
nonneg-primfun-list [] ←→ True
| nonneg-primfun-list ((-,x)#xs) ←→ x > 0 ∨ (x = 0 ∧ nonneg-primfun-list xs)

fun iszero-primfun-list where
iszero-primfun-list [] ←→ True
| iszero-primfun-list ((-,x)#xs) ←→ x = 0 ∧ iszero-primfun-list xs

definition group-primfuns ≡ groupsort.group-sort fst merge-primfun

lemma list-ConsCons-induct:
assumes P [] ∨ x. P [x] ∨ x y xs. P (y#xs) ⇒ P (x#y#xs)
sows P xs
proof (induction xs rule: length-induct)
case (goal1 xs)
show ?case
proof (cases xs)
  case (Cons x xs')
  note A = this
  from assms goal1 show ?thesis
proof (cases xs')
  case (Cons y xs'')
  with goal1 A have P (y#xs'') by simp
  with Cons A assms show ?thesis by simp
qed (simp add: assms A)
qed (simp add: assms)
qed

lemma landau-function-family-chain-primfuns:
assumes sorted (map fst fs)
assumes distinct (map fst fs)
shows landau-function-family-chain fs (eval-primfun' o fst)
proof
  case goal2
  from assms show ?case
proof (induction fs rule: list-ConsCons-induct)
  case (goal2 g)
  from eval-primfun' -at-top [of fst g]
  have eval-primfun' (fst g) ∈ ω(λ-. 1) by (intro smallomegaI-filterlim-at-top')
  simp
  thus ?case by (simp add: smallomega-iff-smallo)
next
  case (goal3 f g gs)
  thus ?case by (auto simp: primfun-dominates sorted-Cons)
qed simp
qed (auto simp: eval-primfun' -at-top)

interpretation groupsort-primfun!: groupsort fst merge-primfun eval-primfuns
proof
  case (goal1 x y)
  thus ?case by (induction x y rule: merge-primfun.induct) simp-all
next
  case (goal2 fs gs)
  show ?case
proof
    fix x
    have eval-primfuns fs x = fold op* (map (λf. eval-primfun f x) fs) 1
    unfolding eval-primfuns-def by (simp add: fold-plus-listprod-rev)
    also have fold op* (map (λf. eval-primfun f x) fs) = fold op* (map (λf. eval-primfun f x) gs)
    using goal2 by (intro fold-multiset-equiv ext) (auto simp: multiset-of-map)
    also have ... 1 = eval-primfuns gs x
  qed
unfolding eval-primfuns-def by (simp add: fold-plus-listprod-rev)
finally show eval-primfuns fs x = eval-primfuns gs x.
qed

qed (auto simp: fun-eq-iff eval-merge-primfun eval-primfuns-def)

lemma nonneg-primfun-list-iff: nonneg-primfun-list fs = nonneg-list (map snd fs)
by (induction fs rule: nonneg-primfun-list.induct) simp-all

lemma pos-primfun-list-iff: pos-primfun-list fs = pos-list (map snd fs)
by (induction fs rule: pos-primfun-list.induct) simp-all

lemma iszero-primfun-list-iff: iszero-primfun-list fs = list-all (op = 0) (map snd fs)
by (induction fs rule: iszero-primfun-list.induct) simp-all

lemma landau-primfuns-iff:
((λ-. 1) ∈ O(eval-primfuns fs)) = nonneg-primfun-list (group-primfuns fs) (is ?A)
((λ-. 1) ∈ o(eval-primfuns fs)) = pos-primfun-list (group-primfuns fs) (is ?B)
((λ-. 1) ∈ Θ(eval-primfuns fs)) = iszero-primfun-list (group-primfuns fs) (is ?C)
proof -
interpret landau-function-family-chain group-primfuns fs snd eval-primfun' o fst
by (rule landau-function-family-chain-primfuns)
(simp-all add: groupsort-primfun.sorted-group-sort
groupsort-primfun.distinct-group-sort)

have (λ-. 1) ∈ O(eval-primfuns fs) ↔ (λ-. 1) ∈ O(eval-primfuns (group-primfuns fs))
by (simp-all add: groupsort-primfun.g-group-sort group-primfuns-def)
also have ... ↔ nonneg-list (map snd (group-primfuns fs)) using bigo iff
by (simp add: eval-primfuns-def[abs-def] eval-primfun-altdef)
finally show ?A by (simp add: nonneg-primfun-list-iff)

have (λ-. 1) ∈ o(eval-primfuns fs) ↔ (λ-. 1) ∈ o(eval-primfuns (group-primfuns fs))
by (simp-all add: groupsort-primfun.g-group-sort group-primfuns-def)
also have ... ↔ pos-list (map snd (group-primfuns fs)) using smallo iff
by (simp add: eval-primfuns-def[abs-def] eval-primfun-altdef)
finally show ?B by (simp add: pos-primfun-list-iff)

have (λ-. 1) ∈ Θ(eval-primfuns fs) ↔ (λ-. 1) ∈ Θ(eval-primfuns (group-primfuns fs))
by (simp-all add: groupsort-primfun.g-group-sort group-primfuns-def)
also have ... ↔ list-all (op = 0) (map snd (group-primfuns fs)) using bigtheta iff
by (simp add: eval-primfuns-def[abs-def] eval-primfun-altdef)
finally show ?C by (simp add: iszero-primfun-list-iff)
qed
lemma LANDAU-PROD-bigo-iff:
LANDAU-PROD bigo c1 c2 fs \leftrightarrow c1 = 0 \lor (c2 \neq 0 \land \text{nonneg-primfun-list (group-primfuns fs)})

unfolding LANDAU-PROD-def
by (cases c1 = 0, simp, cases c2 = 0, simp) (simp-all add: landau-primfuns-iff)

lemma LANDAU-PROD-smallo-iff:
LANDAU-PROD smallo c1 c2 fs \leftrightarrow c1 = 0 \lor (c2 \neq 0 \land \text{pos-primfun-list (group-primfuns fs)})

unfolding LANDAU-PROD-def
by (cases c1 = 0, simp, cases c2 = 0, simp) (simp-all add: landau-primfuns-iff)

lemma LANDAU-PROD-bigtheta-iff:
LANDAU-PROD bigtheta c1 c2 fs \leftrightarrow (c1 = 0 \land c2 = 0) \lor (c1 \neq 0 \land c2 \neq 0 \land \text{iszero-primfun-list (group-primfuns fs)})

proof
  have A: \land P x. (x = 0 \Rightarrow P) \Rightarrow (x \neq 0 \Rightarrow P) \Rightarrow P by blast
  
  { assume eventually (\lambda x. eval-primfuns fs x = 0) at-top
    with eval-primfuns-nonzero[of fs] have eventually (\lambda x::real. False) at-top
      by eventually-elim simp
    hence False by simp
  }
  note B = this
    (insert B, auto simp: LANDAU-PROD-def landau-primfuns-iff)
  qed

lemmas LANDAU-PROD-iff = LANDAU-PROD-bigo-iff LANDAU-PROD-smallo-iff
LENDAU-PROD-bigtheta-iff

lemmas landau-real-prod-simps [simp] =
groupprims-primfun.group-part-deB
group-primfuns-def growsort-primfun.group-sort.simps
growsort-primfun.group-part-primfun.group-sort.simps
nonneg-primfun-list.simp
  iszero-primfun-list.simps

end

5 Simplification procedures

theory Landau-Simprocs
imports Landau-Symbols-Definition Landau-Real-Products
begin
5.1 Simplification under Landau symbols

The following can be seen as simpset for terms under Landau symbols. When given a rule \( f \in \Theta(g) \), the simproc will attempt to rewrite any occurrence of \( f \) under a Landau symbol to \( g \).

**named-theorems** landau-simp

**BigTheta** rules for simplification of Landau symbols

**setup**

\[
\begin{align*}
\text{let} & \quad \text{val eq-thms} = \{ \text{thms landau-theta.cong-bigtheta} \} \\
\text{fun eq-rule thm} & \quad = \text{get-first} (\text{try} (\text{fn eq-thm} => \text{eq-thm OF [thm]}) \text{eq-thms}) \\
\text{in} & \quad \text{Global-Theory.add-thms-dynamic (@\{binding landau-simps\}, fn context => Named-Theorems.get (Context.proof-of context) @\{named-theorems landau-simp\} |> map-filter eq-rule)}
\end{align*}
\]

**lemma** bigtheta-const [landau-simp]:

\[
\begin{align*}
\text{NO-MATCH} & \quad 1 \ c \Rightarrow c \neq 0 \Rightarrow (\lambda x. \ c) \in \Theta(\lambda x. \ 1) \text{ by simp}
\end{align*}
\]

**lemmas** [landau-simp] = bigtheta-const-ln bigtheta-const-ln-powr bigtheta-const-ln-pow

**lemma** bigtheta-const-ln' [landau-simp]:

\[
\begin{align*}
\theta < a & \Rightarrow (\lambda x::real. \ln (x * a)) \in \Theta(\ln)
\end{align*}
\]

**by** (subst mult.commute) (rule bigtheta-const-ln)

**lemma** bigtheta-const-ln-powr' [landau-simp]:

\[
\begin{align*}
\theta < a & \Rightarrow (\lambda x::real. \ln (x * a) \powr p) \in \Theta(\lambda x. \ln x \powr p)
\end{align*}
\]

**by** (subst mult.commute) (rule bigtheta-const-ln-powr)

**lemma** bigtheta-const-ln-pow [landau-simp]:

\[
\begin{align*}
\theta < a & \Rightarrow (\lambda x::real. \ln (x * a) \pow p) \in \Theta(\lambda x. \ln x \pow p)
\end{align*}
\]

**by** (subst mult.commute) (rule bigtheta-const-ln-pow)

5.2 Simproc setup

**lemma** landau-gt-1-cong:

\[
\begin{align*}
\text{landau-symbol } L & \Rightarrow (\lambda x::real. \ x > 1 \Rightarrow f x = g x) \Rightarrow L(f) = L(g)
\end{align*}
\]

**using** eventually-gt-at-top[of 1::real] by (auto elim!: eventually-elim1 landau-symbol.cong)

**lemma** landau-gt-1-in-cong:

\[
\begin{align*}
\text{landau-symbol } L & \Rightarrow (\lambda x::real. \ x > 1 \Rightarrow f x = g x) \Rightarrow f \in L(h) \iff g \in L(h)
\end{align*}
\]

**using** eventually-gt-at-top[of 1::real] by (auto elim!: eventually-elim1 landau-symbol.in-cong)

**lemma** landau-prop-equalsI:
\textbf{lemma} \textbf{ab-diff-conv-add-uminus}': (a::\textcdot ab-group-add) - b = -b + a \textbf{by simp}

\textbf{lemma} \textbf{extract-diff-middle}: (a::\textcdot ab-group-add) - (x + b) = -x + (a - b) \textbf{by simp}

\textbf{lemma} \textbf{divide-inverse}': (a::\textcdot \{division-ring,ab-semigroup-mult\}) / b = inverse b * a 
\textbf{by (simp add: divide-inverse mult.commute)}

\textbf{lemma} \textbf{extract-divide-middle}: (a::\textcdot \{field\}) / (x * b) = inverse x * (a / b) 
\textbf{by (simp add: divide-inverse algebra-simps)}

\textbf{lemmas} \textbf{landau-cancel} = landau-symbol.mult-cancel-left

\textbf{lemmas} \textbf{mult-cancel-left'} = landau-symbol.mult-cancel-left[OF - bigtheta-refl eventually-nonzeroD]

\textbf{lemma} \textbf{mult-cancel-left'-1}: 
\textbf{assumes} landau-symbol L eventually-nonzero f
\textbf{shows} f \in L(\lambda x. f x * g2 x) \iff (\lambda -. 1) \in L(g2) 
(\lambda x. f x * f2 x) \in L(f) \iff f2 \in L(\lambda -. 1) 
f \in L(f) \iff (\lambda -. 1) \in L(\lambda -. 1)
\textbf{using} mult-cancel-left'[OF assms, of \lambda -. 1] mult-cancel-left'[OF assms, of - \lambda -. 1]
\textbf{by simp-all}

\textbf{lemmas} \textbf{landau-cancel-simps} = mult-cancel-left' mult-cancel-left-1

\textbf{ML-file} landau-simprocs.ML

\textbf{lemmas} \textbf{bigtheta-simps} = landau-theta.cong-bigtheta[OF bigtheta-const-ln]
landau-theta.cong-bigtheta[OF bigtheta-const-ln-powr]

\textbf{simproc-setup} \textbf{landau-cancel-factor} ( 
  f \in o(g) | f \in O(g) | f \in \omega(g) | f \in \Omega(g) | f \in \Theta(g) 
) = \langle K \text{Landau.cancel-factor-simproc} \rangle

\textbf{simproc-setup} \textbf{simplify-landau-sum} ( 
  o(\lambda x. f x) | O(\lambda x. f x) | \omega(\lambda x. f x) | \Omega(\lambda x. f x) | \Theta(\lambda x. f x) | 
  f \in o(g) | f \in O(g) | f \in \omega(g) | f \in \Omega(g) | f \in \Theta(g) 
) = \langle K \text{(Landau.lift-landau-simproc Landau.simplify-landau-sum-simproc)} \rangle
5.3 Tests

5.3.1 Product simplification tests

\textbf{lemma} \((\lambda x::\text{field}. \; f \times x) \in O(\lambda x. \; g \times (h \times x)) \iff f \in O(\lambda x. \; g \times h)

\textbf{by simp}

\textbf{lemma} \((\lambda x::\text{field}. \; x) \in \omega(\lambda x. \; g \times (h \times x)) \iff (\lambda. \; 1) \in \omega(\lambda x. \; g \times h)

\textbf{by simp}

5.3.2 Real product decision procure tests

\textbf{lemma} \((\lambda x. \; x \text{ powr } 1) \in O(\lambda x. \; x \text{ powr } 2 :: \text{ real})

\textbf{by simp}

\textbf{lemma} \(\Theta(\lambda x::\text{real}. \; 2 \text{ powr } 3 - 4 \times \text{ powr } 2) = \Theta(\lambda x::\text{real}. \; x \text{ powr } 3)

\textbf{by (simp add: landau-theta.absorb)}

\textbf{lemma} \(p < q \implies (\lambda x::\text{real}. \; c \times \text{ powr } p \times \ln x \times \text{ powr } r) \in o(\lambda x::\text{real}. \; x \text{ powr } q)

\textbf{by simp}

\textbf{lemma} \(c \neq 0 \implies p > q \implies (\lambda x::\text{real}. \; c \times \text{ powr } p \times \ln x \times \text{ powr } r) \in \omega(\lambda x::\text{real}. \; x \text{ powr } q)

\textbf{by simp}

\textbf{lemma} \(b > 0 \implies (\lambda x::\text{real}. \; x / \ln (2 \times b \times x) \times 2) \in o(\lambda x. \; x \times \ln (b \times x))

\textbf{by simp}

\textbf{lemma} \(o(\lambda x::\text{real}. \; x \times \ln (3 \times x)) = o(\lambda x. \; \ln x \times x)

\textbf{by (simp add: mult.commute)}

\textbf{lemma} \((\lambda x::\text{real}. \; x) \in o(\lambda x. \; x \times \ln (3 \times x)) \text{ by simp}

\textbf{ML-val} \((\langle Landau.simplify-landau-real-prod-prop-simproc Landau.simplify-landau-product-simproc \rangle)

\textbf{lemma} \((\lambda x. \; 3 \times \ln x \times x / x \times \ln (\ln (\ln x)))) \in
\[\omega(\lambda x::\text{real}. \ 5 \ast \ln (\ln x) \ast 2 / (2\ast x) \text{ powr } 1.5 \ast \text{ inverse } 2)\]
by simp

5.3.3 Sum cancelling tests

lemma \(\Theta(\lambda x::\text{real}. \ 2 \ast x \text{ powr } 3 + x \ast x^2 / \ln x) = \Theta(\lambda x::\text{real}. x \text{ powr } 3)\)
by simp

lemma \(\Theta(\lambda x::\text{real}. \ 2 \ast x \text{ powr } 3 + x \ast x^2 / \ln x + 42 \ast x \text{ powr } 9 + 213 \ast x \text{ powr } 5 - 4 \ast x \text{ powr } 7) = \Theta(\lambda x::\text{real}. x \ast 3 + x / \ln x \ast x \text{ powr } (3/2) - 2\ast x \text{ powr } 9)\)
by simp

lemma \((\lambda x::\text{real}. x + x \ast \ln (3\ast x)) \in o(\lambda x::\text{real}. x^2 + \ln (2\ast x) \text{ powr } 3)\) by simp

end

theory Landau-Symbols
imports
  Landau-Symbols-Definition Landau-Simprocs
begin
end