

# Landau Symbols

Manuel Eberl

May 20, 2026

## Contents

<b>1</b>	<b>Sorting and grouping factors</b>	<b>1</b>
<b>2</b>	<b>Decision procedure for real functions</b>	<b>8</b>
2.1	Eventual non-negativity/non-zerosness . . . . .	8
2.2	Rewriting Landau symbols . . . . .	13
2.3	Preliminary facts . . . . .	18
2.4	Decision procedure . . . . .	21
2.5	Reification . . . . .	33
<b>3</b>	<b>Simplification procedures</b>	<b>41</b>
3.1	Simplification under Landau symbols . . . . .	41
3.2	Simproc setup . . . . .	41
3.3	Tests . . . . .	43
3.3.1	Product simplification tests . . . . .	44
3.3.2	Real product decision procure tests . . . . .	44
3.3.3	Sum cancelling tests . . . . .	44

## 1 Sorting and grouping factors

```
theory Group-Sort
imports Main 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: mset xs = mset ys ⇒ g xs = g ys
  assumes g-merge: f x = f y ⇒ g [x,y] = g [merge x y]
  assumes g-append-cong: g xs1 = g xs2 ⇒ g ys1 = g ys2 ⇒ g (xs1 @ ys1) =
g (xs2 @ ys2)
begin

```

```

context
begin

```

```

private function part-aux ::
  'b ⇒ 'a list ⇒ ('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, goal-cases)

```

```

  case prems: (1 P p xs ls eq gs)

```

```

  show ?case

```

```

  proof (cases xs)

```

```

    fix x xs' assume xs = x # xs'

```

```

    thus ?thesis using prems by (cases f x p rule: linorder-cases) auto

```

```

  qed (auto intro: prems(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-locale

```

```

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 fst (part-aux p xs (ls, eq, gs)) = rev (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] []] 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:
  assumes ∧x. x ∈# A ⇒ P x ⇒ Q x ⇒ False
  shows filter-mset P A + filter-mset Q A = filter-mset (λx. P x ∨ Q x) A (is ?lhs
= ?rhs)
  using assms by (auto simp add: count-eq-zero-iff intro!: multiset-eqI) blast

```

```

private lemma multiset-of-sort: mset (sort xs) = mset xs
proof (induction xs rule: sort.induct)
  case (2 x xs)
  let ?M = λoper. {#y:# mset xs. oper (f y) (f x)#}
  from 2 have mset (sort (x#xs)) = ?M (<) + ?M (=) + ?M (>) + {#x#}
    by (simp add: part Multiset.union-assoc mset-filter)
  also have ?M (<) + ?M (=) + ?M (>) = mset 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 set-sort: set (sort xs) = set xs
  using arg-cong[OF multiset-of-sort[of xs], of set-mset] by (simp only: set-mset-mset)

```

```

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

```

```

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

```

```

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

```

```

private lemma f-fold-merge: (λy. 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 (2 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 2(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
qed simp

```

```

private lemma sorted-group: sorted (map f xs) => sorted (map f (group xs))
proof (induction xs rule: group.induct)
  case (2 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 2(2) x'' have ... ≤ f x' by (auto)
  finally have f (fold merge [y←xs . f y = f x] x) ≤ f x' .
  }
moreover from 2(2) have sorted (map f (group [xa←xs . f xa ≠ f x]))

```

by (intro 2 sorted-filter) (simp-all add: o-def)  
ultimately show ?case by (simp add: o-def)  
qed simp-all

**private lemma distinct-group:** distinct (map f (group xs))  
**proof** (induction xs rule: group.induct)  
case (2 x xs)  
have distinct (map f (group [xa←xs . f xa ≠ f x])) by (intro 2) (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-merge) (auto dest: f-group)  
ultimately show ?case by (simp add: o-def)  
qed simp

**private lemma g-fold-same:**  
assumes  $\bigwedge z. z \in \text{set } xs \implies f z = f x$   
shows  $g (\text{fold merge } xs \ x \ \# \ ys) = g (x \# xs @ ys)$   
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.prem have  $g \dots = g ([\text{merge } y \ x] @ xs @ ys)$   
by (intro g-append-cong g-merge) auto  
also have  $[\text{merge } y \ x] @ xs @ ys = \text{merge } y \ x \ \# \ xs @ ys$  by simp  
also from Cons.prem have  $g \dots = g (\text{fold merge } xs \ (\text{merge } y \ x) \ \# \ ys)$   
by (intro Cons.IH[symmetric]) (auto simp: f-merge)  
also have  $\dots = g (\text{fold merge } (y \ \# \ xs) \ x \ \# \ ys)$  by simp  
finally show ?case by simp  
qed simp

**private lemma g-group:**  $g (\text{group } xs) = g \ xs$   
**proof** (induction xs rule: group.induct)  
case (2 x xs)  
have  $g (\text{group } (x \# xs)) = g (\text{fold merge } [y \leftarrow xs . f y = f x] \ x \ \# \ \text{group } [xa \leftarrow xs . f xa \neq f x])$   
by (simp add: o-def)  
also have  $\dots = g (x \ \# \ [y \leftarrow xs . f y = f x] @ \text{group } [y \leftarrow xs . f y \neq f x])$   
by (intro g-fold-same) simp-all  
also have  $\dots = g ((x \ \# \ [y \leftarrow xs . f y = f x]) @ \text{group } [y \leftarrow xs . f y \neq f x])$  (is - = ?A) by simp  
also from 2 have  $g (\text{group } [y \leftarrow xs . f y \neq f x]) = g [y \leftarrow xs . f y \neq f x]$  by (simp add: o-def)  
hence ?A =  $g ((x \ \# \ [y \leftarrow xs . f y = f x]) @ [y \leftarrow xs . f y \neq f x])$   
by (intro g-append-cong) simp-all  
also have  $\dots = g (x \ \# \ xs)$  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, goal-cases)  
  case prems: (1 P p xs ls eq gs)  
  show ?case  
  proof (cases xs)  
    fix x xs' assume xs = x # xs'  
    thus ?thesis using prems by (cases f x p rule: linorder-cases) auto  
  qed (auto intro: prems(1))  
qed simp-all  
termination by (relation Wellfounded.measure (size ∘ fst ∘ snd)) simp-all
```

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

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

```
private lemma group-part:  
  group-part p xs = (rev (filter (λx. f x < f p) xs),  
    fold merge (filter (λx. f x = f p) xs) p, rev (filter (λx. f x > f p) xs))  
proof –  
  {  
    fix p xs ls eq gs  
    have fst (group-part-aux p xs (ls, eq, gs)) = rev (filter (λx. f x < p) xs) @ ls  
      by (induction p xs ls eq gs rule: group-part-aux-induct) simp-all  
  } note A = this  
  {  
    fix p xs ls eq gs  
    have snd (snd (group-part-aux p xs (ls, eq, gs))) = rev (filter (λx. f x > p) xs)  
  } @ gs  
  by (induction p xs ls eq gs rule: group-part-aux-induct) simp-all  
  } note B = this  
  {  
    fix p xs ls eq gs  
    have fst (snd (group-part-aux p xs (ls, eq, gs))) =  
      fold merge (filter (λx. f x = p) xs) eq  
    by (induction p xs ls eq gs rule: group-part-aux-induct) auto  
  } note C = this
```

**note**  $ABC = A B C$   
**from**  $ABC[of\ f\ p\ xs\ []\ p\ []]$  **show** *?thesis* **unfolding** *group-part-def*  
**by** (*intro prod-eqI*) *simp-all*  
**qed**

**function** *group-sort* :: 'a list  $\Rightarrow$  'a list **where**  
*group-sort* [] = []  
| *group-sort* (x#xs) = (case *group-part* x xs of (ls, eq, gs)  $\Rightarrow$  *group-sort* ls @ eq #  
*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**  $\bigwedge x\ y. x \in set\ xs \Rightarrow y \in set\ ys \Rightarrow f\ x \neq f\ y$   
**shows**  $group\ (xs\ @\ ys) = group\ xs\ @\ group\ ys$   
**using** *assms*  
**proof** (*induction xs arbitrary: ys rule: length-induct*)  
**case** (1 xs')  
**hence** *IH*:  $\bigwedge x\ xs\ ys. length\ xs < length\ xs' \Rightarrow (\bigwedge x\ y. x \in set\ xs \Rightarrow y \in set\ ys \Rightarrow f\ x \neq f\ y)$   
 $\Rightarrow group\ (xs\ @\ ys) = group\ xs\ @\ group\ ys$  **by** *blast*  
**show** *?case*  
**proof** (*cases xs'*)  
**case** (*Cons x xs*)  
**note** [*simp*] = *this*  
**have**  $group\ (xs'\ @\ ys) = fold\ merge\ [y \leftarrow xs @ ys . f\ y = f\ x]\ x\ \#$   
 $group\ ([xa \leftarrow xs . f\ xa \neq f\ x]\ @\ [xa \leftarrow ys . f\ xa \neq f\ x])$  **by** (*simp add: o-def*)  
**also from** 1(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** 1(2) **have**  $[xa \leftarrow ys . f\ xa \neq f\ x] = ys$  **by** (*force simp: filter-id-conv*)  
**also have**  $group\ ([xa \leftarrow xs . f\ xa \neq f\ x]\ @\ ys) =$   
 $group\ [xa \leftarrow xs . f\ xa \neq f\ x]\ @\ group\ ys$  **using** 1(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*]:  $group\ xs = [] \longleftrightarrow xs = []$   
**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** (2 x xs)  
**have**  $group\ sort\ (x\ #\ xs) =$   
 $group\ sort\ (rev\ [xa \leftarrow xs . f\ xa < f\ x])\ @\ group\ (x\ #\ [xa \leftarrow xs . f\ xa = f\ x])\ @$   
 $group\ sort\ (rev\ [xa \leftarrow xs . f\ x < f\ xa])$  **by** (*simp add: group-part*)  
**also have**  $group\ sort\ (rev\ [xa \leftarrow xs . f\ xa < f\ x]) = group\ (sort\ (rev\ [xa \leftarrow xs . f\ xa$

```

< f x])
  by (rule 2) (simp-all add: group-part)
  also have group-sort (rev [xa←xs . f xa > f x]) = group (sort (rev [xa←xs . f xa
> f x]))
  by (rule 2) (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

```

## 2 Decision procedure for real functions

```

theory Landau-Real-Products
imports
  Main
  HOL-Library.Function-Algebras
  HOL-Library.Set-Algebras
  HOL-Library.Landau-Symbols
  Group-Sort
begin

```

## 2.1 Eventual non-negativity/non-zerosness

For certain transformations of Landau symbols, it is required that the functions involved are eventually non-negative or 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*  $F f \longleftrightarrow \text{eventually } (\lambda x. (f x :: - :: \text{real-normed-field}) \neq 0) F$

**definition** *eventually-nonneg*  $F f \longleftrightarrow \text{eventually } (\lambda x. (f x :: - :: \text{linordered-field}) \geq 0) F$

**named-theorems** *eventually-nonzero-simps*

**lemmas** [*eventually-nonzero-simps*] =  
*eventually-nonzero-def* [*symmetric*] *eventually-nonneg-def* [*symmetric*]

**lemma** *eventually-nonzeroD*: *eventually-nonzero*  $F f \implies \text{eventually } (\lambda x. f x \neq 0) F$

**by** (*simp add: eventually-nonzero-def*)

**lemma** *eventually-nonzero-const* [*eventually-nonzero-simps*]:  
*eventually-nonzero*  $F (\lambda x :: \text{linorder}. c) \longleftrightarrow F = \text{bot} \vee c \neq 0$

**unfolding** *eventually-nonzero-def* **by** (*auto simp add: eventually-False*)

**lemma** *eventually-nonzero-inverse* [*eventually-nonzero-simps*]:  
*eventually-nonzero*  $F (\lambda x. \text{inverse } (f x)) \longleftrightarrow \text{eventually-nonzero } F f$

**unfolding** *eventually-nonzero-def* **by** *simp*

**lemma** *eventually-nonzero-mult* [*eventually-nonzero-simps*]:  
*eventually-nonzero*  $F (\lambda x. f x * g x) \longleftrightarrow \text{eventually-nonzero } F f \wedge \text{eventually-nonzero } F g$

**unfolding** *eventually-nonzero-def* **by** (*simp-all add: eventually-conj-iff* [*symmetric*])

**lemma** *eventually-nonzero-pow* [*eventually-nonzero-simps*]:  
*eventually-nonzero*  $F (\lambda x :: \text{linorder}. f x ^ n) \longleftrightarrow n = 0 \vee \text{eventually-nonzero } F f$

**by** (*induction n*) (*auto simp: eventually-nonzero-simps*)

**lemma** *eventually-nonzero-divide* [*eventually-nonzero-simps*]:  
*eventually-nonzero*  $F (\lambda x. f x / g x) \longleftrightarrow \text{eventually-nonzero } F f \wedge \text{eventually-nonzero } F g$

**unfolding** *eventually-nonzero-def* **by** (*simp-all add: eventually-conj-iff* [*symmetric*])

**lemma** *eventually-nonzero-ident-at-top-linorder* [*eventually-nonzero-simps*]:  
*eventually-nonzero at-top*  $(\lambda x :: 'a :: \{\text{real-normed-field}, \text{linordered-field}\}. x)$

**unfolding** *eventually-nonzero-def* **by** *simp*

**lemma** *eventually-nonzero-ident-nhds* [*eventually-nonzero-simps*]:  
*eventually-nonzero* (*nhds a*)  $(\lambda x. x) \longleftrightarrow a \neq 0$

**using** *eventually-nhds-in-open*[of  $-\{0\}$   $a$ ]  
**by** (*auto elim!*: *eventually-mono simp*: *eventually-nonzero-def open-Compl*  
*dest*: *eventually-nhds-x-imp-x*)

**lemma** *eventually-nonzero-ident-at-within* [*eventually-nonzero-simps*]:  
*eventually-nonzero* (at  $a$  within  $A$ ) ( $\lambda x. x$ )  
**using** *eventually-nonzero-ident-nhds*[of  $a$ ]  
**by** (*cases*  $a = 0$ ) (*auto simp*: *eventually-nonzero-def eventually-at-filter elim!*:  
*eventually-mono*)

**lemma** *eventually-nonzero-ln-at-top* [*eventually-nonzero-simps*]:  
*eventually-nonzero* at-top ( $\lambda x::\text{real}. \ln x$ )  
**unfolding** *eventually-nonzero-def* **by** (*auto intro!*: *eventually-mono*[*OF eventually-gt-at-top*[of  $1$ ]])

**lemma** *eventually-nonzero-ln-const-at-top* [*eventually-nonzero-simps*]:  
 $b > 0 \implies \text{eventually-nonzero at-top } (\lambda x. \ln (b * x :: \text{real}))$   
**unfolding** *eventually-nonzero-def*  
**apply** (*rule eventually-mono* [*OF eventually-gt-at-top*[of  $\max 1$  (*inverse*  $b$ )]])  
**by** (*metis exp-ln exp-minus exp-minus-inverse less-numeral-extra*( $\beta$ ) *ln-gt-zero*  
*max-less-iff-conj mult.commute mult-strict-right-mono*)

**lemma** *eventually-nonzero-ln-const'-at-top* [*eventually-nonzero-simps*]:  
 $b > 0 \implies \text{eventually-nonzero at-top } (\lambda x. \ln (x * b :: \text{real}))$   
**using** *eventually-nonzero-ln-const-at-top*[of  $b$ ] **by** (*simp add*: *mult.commute*)

**lemma** *eventually-nonzero-powr-at-top* [*eventually-nonzero-simps*]:  
*eventually-nonzero* at-top ( $\lambda x::\text{real}. f x \text{ powr } p$ )  $\longleftrightarrow$  *eventually-nonzero at-top*  $f$   
**unfolding** *eventually-nonzero-def* **by** *simp*

**lemma** *eventually-nonneg-const* [*eventually-nonzero-simps*]:  
*eventually-nonneg*  $F$  ( $\lambda \cdot. c$ )  $\longleftrightarrow F = \text{bot} \vee c \geq 0$   
**unfolding** *eventually-nonneg-def* **by** (*auto simp*: *eventually-False*)

**lemma** *eventually-nonneg-inverse* [*eventually-nonzero-simps*]:  
*eventually-nonneg*  $F$  ( $\lambda x. \text{inverse } (f x)$ )  $\longleftrightarrow$  *eventually-nonneg*  $F f$   
**unfolding** *eventually-nonneg-def* **by** (*intro eventually-subst*) (*auto*)

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

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

**lemma** *eventually-nonneg-mult'* [*eventually-nonzero-simps*]:  
**assumes** *eventually-nonneg*  $F$   $(\lambda x. -f x)$  *eventually-nonneg*  $F$   $(\lambda x. -g x)$   
**shows** *eventually-nonneg*  $F$   $(\lambda x. f x * g x)$   
**using** *assms* **unfolding** *eventually-nonneg-def* **by** *eventually-elim* (*auto intro: mult-nonpos-nonpos*)

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

**lemma** *eventually-nonneg-divide'* [*eventually-nonzero-simps*]:  
**assumes** *eventually-nonneg*  $F$   $(\lambda x. -f x)$  *eventually-nonneg*  $F$   $(\lambda x. -g x)$   
**shows** *eventually-nonneg*  $F$   $(\lambda x. f x / g x)$   
**using** *assms* **unfolding** *eventually-nonneg-def* **by** *eventually-elim* (*auto intro: divide-nonpos-nonpos*)

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

**lemma** *eventually-nonneg-ident-nhds* [*eventually-nonzero-simps*]:  
**fixes**  $a :: 'a :: \{\text{linorder-topology, linordered-field}\}$   
**shows**  $a > 0 \implies \text{eventually-nonneg}(\text{nhds } a)$   $(\lambda x. x)$  **unfolding** *eventually-nonneg-def*  
**using** *eventually-nhds-in-open*[*of*  $\{0 < ..\}$   $a$ ]  
**by** (*auto simp: eventually-nonneg-def dest: eventually-nhds-x-imp-x elim!: eventually-mono*)

**lemma** *eventually-nonneg-ident-at-within* [*eventually-nonzero-simps*]:  
**fixes**  $a :: 'a :: \{\text{linorder-topology, linordered-field}\}$   
**shows**  $a > 0 \implies \text{eventually-nonneg}(\text{at } a \text{ within } A)$   $(\lambda x. x)$   
**using** *eventually-nonneg-ident-nhds*[*of*  $a$ ]  
**by** (*auto simp: eventually-nonneg-def eventually-at-filter elim: eventually-mono*)

**lemma** *eventually-nonneg-pow* [*eventually-nonzero-simps*]:  
*eventually-nonneg*  $F$   $f \implies \text{eventually-nonneg}$   $F$   $(\lambda x. f x ^ n)$   
**by** (*induction n*) (*auto simp: eventually-nonzero-simps*)

**lemma** *eventually-nonneg-powr* [*eventually-nonzero-simps*]:  
*eventually-nonneg*  $F$   $(\lambda x. f x \text{ powr } y :: \text{real})$  **by** (*simp add: eventually-nonneg-def*)

**lemma** *eventually-nonneg-ln-at-top* [*eventually-nonzero-simps*]:  
*eventually-nonneg at-top*  $(\lambda x. \ln x :: \text{real})$   
**by** (*auto intro!: eventually-mono*[*OF eventually-gt-at-top*][*of*  $1 :: \text{real}$ ])  
*simp: eventually-nonneg-def*)

**lemma** *eventually-nonneg-ln-const* [*eventually-nonzero-simps*]:  
 $b > 0 \implies \text{eventually-nonneg at-top}$   $(\lambda x. \ln (b*x) :: \text{real})$

**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*]:  
 $b > 0 \implies \text{eventually-nonneg at-top } (\lambda x. \ln (x*b) :: \text{real})$   
**using** *eventually-nonneg-ln-const*[of  $b$ ] **by** (*simp add: mult.commute*)

**lemma** *eventually-nonzero-bigtheta'*:  
 $f \in \Theta[F](g) \implies \text{eventually-nonzero } F f \longleftrightarrow \text{eventually-nonzero } F g$   
**unfolding** *eventually-nonzero-def* **by** (*rule eventually-nonzero-bigtheta*)

**lemma** *eventually-nonneg-at-top*:  
**assumes** *filterlim f at-top F*  
**shows** *eventually-nonneg F f*  
**proof** –  
**from** *assms* **have** *eventually*  $(\lambda x. f x \geq 0) F$   
**by** (*simp add: filterlim-at-top*)  
**thus** *?thesis* **unfolding** *eventually-nonneg-def* **by** *eventually-elim simp*  
**qed**

**lemma** *eventually-nonzero-at-top*:  
**assumes** *filterlim*  $(f :: 'a \Rightarrow 'b :: \{\text{linordered-field, real-normed-field}\}) \text{ at-top } F$   
**shows** *eventually-nonzero F f*  
**proof** –  
**from** *assms* **have** *eventually*  $(\lambda x. f x \geq 1) F$   
**by** (*simp add: filterlim-at-top*)  
**thus** *?thesis* **unfolding** *eventually-nonzero-def* **by** *eventually-elim auto*  
**qed**

**lemma** *eventually-nonneg-at-top-ASSUMPTION* [*eventually-nonzero-simps*]:  
 $\text{ASSUMPTION } (\text{filterlim } f \text{ at-top } F) \implies \text{eventually-nonneg } F f$   
**by** (*simp add: ASSUMPTION-def eventually-nonneg-at-top*)

**lemma** *eventually-nonzero-at-top-ASSUMPTION* [*eventually-nonzero-simps*]:  
 $\text{ASSUMPTION } (\text{filterlim } f \text{ (at-top } :: 'a :: \{\text{linordered-field, real-normed-field}\} \text{ filter) } F) \implies$   
 $\text{eventually-nonzero } F f$   
**using** *eventually-nonzero-at-top*[of  $f F$ ] **by** (*simp add: ASSUMPTION-def*)

**lemma** *filterlim-at-top-iff-smallomega*:  
**fixes**  $f :: - \Rightarrow \text{real}$   
**shows**  $\text{filterlim } f \text{ at-top } F \longleftrightarrow f \in \omega[F](\lambda-. 1) \wedge \text{eventually-nonneg } F f$   
**unfolding** *eventually-nonneg-def*  
**proof** *safe*  
**assume**  $A: \text{filterlim } f \text{ at-top } F$   
**thus**  $B: \text{eventually } (\lambda x. f x \geq 0) F$  **by** (*simp add: eventually-nonzero-simps*)  
{  
**fix**  $c$   
**from**  $A$  **have**  $\text{filterlim } (\lambda x. \text{norm } (f x)) \text{ at-top } F$

by (intro filterlim-at-infinity-imp-norm-at-top filterlim-at-top-imp-at-infinity)  
 hence eventually  $(\lambda x. \text{norm } (f x) \geq c)$   $F$  by (auto simp: filterlim-at-top)  
 }  
 thus  $f \in \omega[F](\lambda-. 1)$  by (rule landau-omega.smallI)  
 next  
 assume  $A: f \in \omega[F](\lambda-. 1)$  and  $B: \text{eventually } (\lambda x. f x \geq 0)$   $F$   
 {  
 fix  $c :: \text{real}$  assume  $c > 0$   
 from landau-omega.smallD[OF A this] B  
 have eventually  $(\lambda x. f x \geq c)$   $F$  by eventually-elim simp  
 }  
 thus filterlim  $f$  at-top  $F$   
 by (subst filterlim-at-top-gt[of - - 0]) simp-all  
 qed

**lemma smallomega-1-iff:**  
*eventually-nonneg*  $F f \implies f \in \omega[F](\lambda-. 1 :: \text{real}) \longleftrightarrow \text{filterlim } f \text{ at-top } F$   
 by (simp add: filterlim-at-top-iff-smallomega)

**lemma smallo-1-iff:**  
*eventually-nonneg*  $F f \implies (\lambda-. 1 :: \text{real}) \in o[F](f) \longleftrightarrow \text{filterlim } f \text{ at-top } F$   
 by (simp add: filterlim-at-top-iff-smallomega smallomega-iff-smallo)

**lemma eventually-nonneg-add1 [eventually-nonzero-simps]:**  
 assumes *eventually-nonneg*  $F f g \in o[F](f)$   
 shows *eventually-nonneg*  $F (\lambda x. f x + g x :: \text{real})$   
 using landau-o.smallD[OF assms(2) zero-less-one] assms(1) unfolding *eventually-nonneg-def*  
 by eventually-elim simp-all

**lemma eventually-nonneg-add2 [eventually-nonzero-simps]:**  
 assumes *eventually-nonneg*  $F g f \in o[F](g)$   
 shows *eventually-nonneg*  $F (\lambda x. f x + g x :: \text{real})$   
 using landau-o.smallD[OF assms(2) zero-less-one] assms(1) unfolding *eventually-nonneg-def*  
 by eventually-elim simp-all

**lemma eventually-nonneg-diff1 [eventually-nonzero-simps]:**  
 assumes *eventually-nonneg*  $F f g \in o[F](f)$   
 shows *eventually-nonneg*  $F (\lambda x. f x - g x :: \text{real})$   
 using landau-o.smallD[OF assms(2) zero-less-one] assms(1) unfolding *eventually-nonneg-def*  
 by eventually-elim simp-all

**lemma eventually-nonneg-diff2 [eventually-nonzero-simps]:**  
 assumes *eventually-nonneg*  $F (\lambda x. - g x) f \in o[F](g)$   
 shows *eventually-nonneg*  $F (\lambda x. f x - g x :: \text{real})$   
 using landau-o.smallD[OF assms(2) zero-less-one] assms(1) unfolding *eventually-nonneg-def*

*tually-nonneg-def*  
**by** *eventually-elim simp-all*

## 2.2 Rewriting Landau symbols

**lemma** *bigheta-mult-eq*:  $\Theta[F](\lambda x. f x * g x) = \Theta[F](f) * \Theta[F](g)$   
**proof** (*intro equalityI subsetI*)  
**fix** *h* **assume**  $h \in \Theta[F](f) * \Theta[F](g)$   
**thus**  $h \in \Theta[F](\lambda x. f x * g x)$   
**by** (*elim set-times-elim, hypsubst, unfold func-times*) (*erule (1) landau-theta.mult*)  
**next**  
**fix** *h* **assume**  $h \in \Theta[F](\lambda x. f x * g x)$   
**then obtain**  $c1\ c2 :: \text{real}$   
**where** *c*:  
 $c1 > 0 \ \forall_F\ x\ \text{in}\ F. \text{norm}\ (h\ x) \leq c1 * \text{norm}\ (f\ x * g\ x)$   
 $c2 > 0 \ \forall_F\ x\ \text{in}\ F. c2 * \text{norm}\ (f\ x * g\ x) \leq \text{norm}\ (h\ x)$   
**unfolding** *bigheta-def* **by** (*blast elim: landau-o.bigE*)  
  
**define** *h1 h2*  
**where**  $h1\ x = (\text{if}\ g\ x = 0\ \text{then}\ \text{if}\ f\ x = 0\ \text{then}\ \text{if}\ h\ x = 0\ \text{then}\ h\ x\ \text{else}\ 1\ \text{else}\ f\ x\ \text{else}\ h\ x / g\ x)$   
**and**  $h2\ x = (\text{if}\ g\ x = 0\ \text{then}\ \text{if}\ f\ x = 0\ \text{then}\ h\ x\ \text{else}\ h\ x / f\ x\ \text{else}\ g\ x)$   
**for** *x*  
  
**have**  $h = h1 * h2$  **by** (*intro ext*) (*auto simp: h1-def h2-def field-simps*)  
**moreover have**  $h1 \in \Theta[F](f)$   
**proof** (*rule bighetaI'*)  
**from** *c(3)* **show**  $\min\ c2\ 1 > 0$  **by** *simp*  
**from** *c(1)* **show**  $\max\ c1\ 1 > 0$  **by** *simp*  
**from** *c(2,4)*  
**show** *eventually*  $(\lambda x. \min\ c2\ 1 * (\text{norm}\ (f\ x)) \leq \text{norm}\ (h1\ x) \wedge \text{norm}\ (h1\ x) \leq \max\ c1\ 1 * (\text{norm}\ (f\ x)))\ F$   
**apply** *eventually-elim*  
**proof** (*rule conjI*)  
**fix** *x* **assume**  $A: (\text{norm}\ (h\ x)) \leq c1 * \text{norm}\ (f\ x * g\ x)$   
**and**  $B: (\text{norm}\ (h\ x)) \geq c2 * \text{norm}\ (f\ x * g\ x)$   
**have**  $m: \min\ c2\ 1 * (\text{norm}\ (f\ x)) \leq 1 * (\text{norm}\ (f\ x))$  **by** (*rule mult-right-mono*)  
*simp-all*  
**have**  $\min\ c2\ 1 * \text{norm}\ (f\ x * g\ x) \leq c2 * \text{norm}\ (f\ x * g\ x)$  **by** (*intro mult-right-mono simp-all*)  
**also note** *B*  
**finally show**  $\text{norm}\ (h1\ x) \geq \min\ c2\ 1 * (\text{norm}\ (f\ x))$  **using** *m A*  
**by** (*cases g x = 0*) (*simp-all add: h1-def norm-mult norm-divide field-simps*)  
  
**have**  $m: 1 * (\text{norm}\ (f\ x)) \leq \max\ c1\ 1 * (\text{norm}\ (f\ x))$  **by** (*rule mult-right-mono*)  
*simp-all*  
**note** *A*  
**also have**  $c1 * \text{norm}\ (f\ x * g\ x) \leq \max\ c1\ 1 * \text{norm}\ (f\ x * g\ x)$   
**by** (*intro mult-right-mono simp-all*)

```

finally show  $\text{norm } (h1\ x) \leq \max\ c1\ 1 * (\text{norm } (f\ x))$  using  $m\ A$ 
by (cases  $g\ x = 0$ ) (simp-all add: h1-def norm-mult norm-divide field-simps)+
qed
qed
moreover have  $h2 \in \Theta[F](g)$ 
proof (rule bighetaI')
  from  $c(3)$  show  $\min\ c2\ 1 > 0$  by simp
  from  $c(1)$  show  $\max\ c1\ 1 > 0$  by simp
  from  $c(2,4)$ 
  show eventually  $(\lambda x. \min\ c2\ 1 * (\text{norm } (g\ x)) \leq \text{norm } (h2\ x) \wedge$ 
     $\text{norm } (h2\ x) \leq \max\ c1\ 1 * (\text{norm } (g\ x)))\ F$ 
  apply eventually-elim
proof (rule conjI)
  fix  $x$  assume  $A: (\text{norm } (h\ x)) \leq c1 * \text{norm } (f\ x * g\ x)$ 
    and  $B: (\text{norm } (h\ x)) \geq c2 * \text{norm } (f\ x * g\ x)$ 
  have  $m: \min\ c2\ 1 * (\text{norm } (f\ x)) \leq 1 * (\text{norm } (f\ x))$  by (rule mult-right-mono)
simp-all
  have  $\min\ c2\ 1 * \text{norm } (f\ x * g\ x) \leq c2 * \text{norm } (f\ x * g\ x)$ 
    by (intro mult-right-mono) simp-all
  also note  $B$ 
  finally show  $\text{norm } (h2\ x) \geq \min\ c2\ 1 * (\text{norm } (g\ x))$  using  $m\ A\ B$ 
    by (cases  $g\ x = 0$ ) (auto simp: h2-def abs-mult field-simps)+

  have  $m: 1 * (\text{norm } (g\ x)) \leq \max\ c1\ 1 * (\text{norm } (g\ x))$  by (rule mult-right-mono)
simp-all
  note  $A$ 
  also have  $c1 * \text{norm } (f\ x * g\ x) \leq \max\ c1\ 1 * \text{norm } (f\ x * g\ x)$ 
    by (intro mult-right-mono) simp-all
  finally show  $\text{norm } (h2\ x) \leq \max\ c1\ 1 * (\text{norm } (g\ x))$  using  $m\ A$ 
    by (cases  $g\ x = 0$ ) (simp-all add: h2-def abs-mult field-simps)+
qed
qed
ultimately show  $h \in \Theta[F](f) * \Theta[F](g)$  by blast
qed

```

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*  $A\ B = \{\lambda x. f\ x * g\ x \mid f\ g. f \in A \wedge g \in B\}$

**definition** *set-inverse*  $A = \{\lambda x. \text{inverse } (f\ x) \mid f. f \in A\}$

**definition** *set-divide*  $A\ B = \{\lambda x. f\ x / g\ x \mid f\ g. f \in A \wedge g \in B\}$

**definition** *set-pow*  $A\ n = \{\lambda x. f\ x \wedge^n \mid f. f \in A\}$

**definition** *set-powr*  $A\ y = \{\lambda x. f\ x \text{ powr } y \mid f. f \in A\}$

**lemma** *bigheta-mult-eq-set-mult*:

**shows**  $\Theta[F](\lambda x. f\ x * g\ x) = \text{set-mult } (\Theta[F](f))\ (\Theta[F](g))$

**unfolding** *bigheta-mult-eq set-mult-def set-times-def func-times* **by** *blast*

**lemma** *bigheta-inverse-eq-set-inverse*:

**shows**  $\Theta[F](\lambda x. \text{inverse } (f x)) = \text{set-inverse } (\Theta[F](f))$

**proof** (*intro equalityI subsetI*)

**fix**  $g :: 'a \Rightarrow 'b$  **assume**  $g \in \Theta[F](\lambda x. \text{inverse } (f x))$

**hence**  $(\lambda x. \text{inverse } (g x)) \in \Theta[F](\lambda x. \text{inverse } (\text{inverse } (f x)))$  **by** (*subst bigheta-inverse*)

**also have**  $(\lambda x. \text{inverse } (\text{inverse } (f x))) = f$  **by** (*rule ext simp*)

**finally show**  $g \in \text{set-inverse } (\Theta[F](f))$  **unfolding** *set-inverse-def* **by** *force*

**next**

**fix**  $g :: 'a \Rightarrow 'b$  **assume**  $g \in \text{set-inverse } (\Theta[F](f))$

**then obtain**  $g'$  **where**  $g = (\lambda x. \text{inverse } (g' x))$   $g' \in \Theta[F](f)$  **unfolding** *set-inverse-def* **by** *blast*

**hence**  $(\lambda x. \text{inverse } (g' x)) \in \Theta[F](\lambda x. \text{inverse } (f x))$  **by** (*subst bigheta-inverse*)

**also from**  $\langle g = (\lambda x. \text{inverse } (g' x)) \rangle$  **have**  $(\lambda x. \text{inverse } (g' x)) = g$  **by** (*intro ext simp*)

**finally show**  $g \in \Theta[F](\lambda x. \text{inverse } (f x))$  .

**qed**

**lemma** *set-divide-inverse*:

*set-divide*  $(A :: (- \Rightarrow (- :: \text{division-ring})) \text{ set}) B = \text{set-mult } A (\text{set-inverse } B)$

**proof** (*intro equalityI subsetI*)

**fix**  $f$  **assume**  $f \in \text{set-divide } A B$

**then obtain**  $g h$  **where**  $f = (\lambda x. g x / h x)$   $g \in A$   $h \in B$  **unfolding** *set-divide-def* **by** *blast*

**hence**  $f = g * (\lambda x. \text{inverse } (h x))$   $(\lambda x. \text{inverse } (h x)) \in \text{set-inverse } B$

**unfolding** *set-inverse-def* **by** (*auto simp: divide-inverse*)

**with**  $\langle g \in A \rangle$  **show**  $f \in \text{set-mult } A (\text{set-inverse } B)$  **unfolding** *set-mult-def* **by** *force*

**next**

**fix**  $f$  **assume**  $f \in \text{set-mult } A (\text{set-inverse } B)$

**then obtain**  $g h$  **where**  $f = g * (\lambda x. \text{inverse } (h x))$   $g \in A$   $h \in B$

**unfolding** *set-times-def set-inverse-def set-mult-def* **by** *force*

**hence**  $f = (\lambda x. g x / h x)$  **by** (*intro ext*) (*simp add: divide-inverse*)

**with**  $\langle g \in A \rangle \langle h \in B \rangle$  **show**  $f \in \text{set-divide } A B$  **unfolding** *set-divide-def* **by** *blast*

**qed**

**lemma** *bigheta-divide-eq-set-divide*:

**shows**  $\Theta[F](\lambda x. f x / g x) = \text{set-divide } (\Theta[F](f)) (\Theta[F](g))$

**by** (*simp only: set-divide-inverse divide-inverse bigheta-mult-eq-set-mult bigheta-inverse-eq-set-inverse*)

**primrec** *bigheta-pow* **where**

*bigheta-pow*  $F A 0 = \Theta[F](\lambda-. 1)$

| *bigheta-pow*  $F A (\text{Suc } n) = \text{set-mult } A (\text{bigheta-pow } F A n)$

**lemma** *bigheta-pow-eq-set-pow*:  $\Theta[F](\lambda x. f x \hat{=} n) = \text{bigheta-pow } F (\Theta[F](f)) n$

by (induction n) (simp-all add: bigheta-mult-eq-set-mult)

**definition** *bigheta-powr* where

*bigheta-powr*  $F A y =$  (if  $y = 0$  then  $\{f. \exists g \in A. \text{eventually-nonneg } F g \wedge f \in \Theta[F](\lambda x. g x \text{ powr } y)\}$   
else  $\{f. \exists g \in A. \text{eventually-nonneg } F g \wedge (\forall x. (\text{norm } (f x)) = g x \text{ powr } y)\}$ )

**lemma** *bigheta-powr-eq-set-powr*:

assumes *eventually-nonneg*  $F f$

shows  $\Theta[F](\lambda x. f x \text{ powr } (y::\text{real})) = \text{bigheta-powr } F (\Theta[F](f)) y$

**proof** (cases  $y = 0$ )

assume [simp]:  $y = 0$

show ?thesis

**proof** (intro equalityI subsetI)

fix  $h$  assume  $h \in \text{bigheta-powr } F \Theta[F](f) y$

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

unfolding *bigheta-powr-def* by force

note *this(3)*

also have  $(\lambda x. g x \text{ powr } 0) \in \Theta[F](\lambda x. |g x| \text{ powr } 0)$

using *assms unfolding eventually-nonneg-def*

by (intro *bighetaI-cong*) (auto elim!: *eventually-mono*)

also from  $g(1)$  have  $(\lambda x. |g x| \text{ powr } 0) \in \Theta[F](\lambda x. |f x| \text{ powr } 0)$

by (rule *bigheta-powr*)

also from  $g(2)$  have  $(\lambda x. f x \text{ powr } 0) \in \Theta[F](\lambda x. |f x| \text{ powr } 0)$

unfolding *eventually-nonneg-def*

by (intro *bighetaI-cong*) (auto elim!: *eventually-mono*)

finally show  $h \in \Theta[F](\lambda x. f x \text{ powr } y)$  by *simp*

next

fix  $h$  assume  $h \in \Theta[F](\lambda x. f x \text{ powr } y)$

with *assms* have  $\exists g \in \Theta[F](f). \text{eventually-nonneg } F g \wedge h \in \Theta[F](\lambda x. g x \text{ powr } 0)$

by (intro *bexI[of - f] conjI simp-all*)

thus  $h \in \text{bigheta-powr } F \Theta[F](f) y$  unfolding *bigheta-powr-def* by *simp*

qed

next

assume  $y: y \neq 0$

show ?thesis

**proof** (intro equalityI subsetI)

fix  $h$  assume  $h: h \in \Theta[F](\lambda x. f x \text{ powr } y)$

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

from *bigheta-powr[OF h, of inverse y]*  $y$

have  $?h' \in \Theta[F](\lambda x. f x \text{ powr } 1)$  by (*simp add: powr-powr*)

also have  $(\lambda x. f x \text{ powr } 1) \in \Theta[F](f)$  using *assms unfolding eventually-nonneg-def*

by (intro *bighetaI-cong*) (auto elim!: *eventually-mono*)

finally have  $?h' \in \Theta[F](f)$ .

with  $y$  have  $\exists g \in \Theta[F](f). \text{eventually-nonneg } F g \wedge (\forall x. (\text{norm } (h x)) = g x \text{ powr } y)$

**by** (*intro bexI[of - ?h<sup>1</sup>]*) (*simp-all add: powr-powr eventually-nonneg-def*)  
**thus**  $h \in \text{bigheta-powr } F \Theta[F](f) y$  **using**  $y$  **unfolding** *bigheta-powr-def* **by**  
*simp*  
**next**  
**fix**  $h$  **assume**  $h \in \text{bigheta-powr } F (\Theta[F](f)) y$   
**with**  $y$  **obtain**  $g$  **where**  $A: g \in \Theta[F](f) \wedge x. |h x| = g x \text{ powr } y \text{ eventually-nonneg}$   
 $F g$   
**unfolding** *bigheta-powr-def* **by force**  
**from** *this(3)* **have**  $(\lambda x. g x \text{ powr } y) \in \Theta[F](\lambda x. |g x| \text{ powr } y)$  **unfolding**  
*eventually-nonneg-def*  
**by** (*intro bighetaI-cong*) (*auto elim!: eventually-mono*)  
**also from**  $A(1)$  **have**  $(\lambda x. |g x| \text{ powr } y) \in \Theta[F](\lambda x. |f x| \text{ powr } y)$  **by** (*rule*  
*bigheta-powr*)  
**also have**  $(\lambda x. |f x| \text{ powr } y) \in \Theta[F](\lambda x. f x \text{ powr } y)$  **using** *assms* **unfolding**  
*eventually-nonneg-def*  
**by** (*intro bighetaI-cong*) (*auto elim!: eventually-mono*)  
**finally have**  $(\lambda x. |h x|) \in \Theta[F](\lambda x. f x \text{ powr } y)$  **by** (*subst A(2)*)  
**thus**  $(\lambda x. h x) \in \Theta[F](\lambda x. f x \text{ powr } y)$  **by** *simp*  
**qed**  
**qed**

**lemmas** *bigheta-factors-eq =*  
*bigheta-mult-eq-set-mult bigheta-inverse-eq-set-inverse bigheta-divide-eq-set-divide*  
  
*bigheta-pow-eq-set-pow bigheta-powr-eq-set-powr*

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

**lemma** (*in landau-symbol*) *meta-cong-bigheta*:  $\Theta[F](f) \equiv \Theta[F](g) \implies L F (f) \equiv$   
 $L F (g)$   
**using** *bigheta-refl[of f]* **by** (*intro eq-reflection cong-bigheta*) *blast*

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

## 2.3 Preliminary facts

**lemma** *real-powr-at-top*:  
**assumes**  $(p::\text{real}) > 0$   
**shows** *filterlim*  $(\lambda x. x \text{ powr } p) \text{ at-top at-top}$   
**proof** (*subst filterlim-cong[OF refl refl]*)  
**show** *LIM*  $x \text{ at-top. exp } (p * \ln x) :> \text{at-top}$   
**by** (*rule filterlim-compose[OF exp-at-top filterlim-tendsto-pos-mult-at-top[OF*  
*tendsto-const]]*)  
*(simp-all add: ln-at-top assms)*  
**show** *eventually*  $(\lambda x. x \text{ powr } p = \text{exp } (p * \ln x)) \text{ at-top}$   
**using** *eventually-gt-at-top[of 0]* **by** *eventually-elim (simp add: powr-def)*  
**qed**

**lemma** *tendsto-ln-over-powr*:  
**assumes**  $(a::real) > 0$   
**shows**  $((\lambda x. \ln x / x \text{ powr } a) \longrightarrow 0)$  *at-top*  
**proof** (*rule lhospital-at-top-at-top*)  
**from** *assms* **show** *LIM*  $x$  *at-top*.  $x \text{ powr } a :> \text{at-top}$  **by** (*rule real-powr-at-top*)  
**show** *eventually*  $(\lambda x. a * x \text{ powr } (a - 1) \neq 0)$  *at-top*  
**using** *eventually-gt-at-top*[*of 0::real*] **by** *eventually-elim* (*insert assms, simp*)  
**show** *eventually*  $(\lambda x::real. (\ln \text{ has-real-derivative } (\text{inverse } x)) (\text{at } x))$  *at-top*  
**using** *eventually-gt-at-top*[*of 0::real*] *DERIV-ln* **by** (*elim eventually-mono*) *simp*  
**show** *eventually*  $(\lambda x. ((\lambda x. x \text{ powr } a) \text{ has-real-derivative } a * x \text{ powr } (a - 1)))$  (*at*  $x$ ) *at-top*  
**using** *eventually-gt-at-top*[*of 0::real*]  
**by** *eventually-elim* (*auto intro!: derivative-eq-intros*)  
**have** *eventually*  $(\lambda x. \text{inverse } a * x \text{ powr } -a = \text{inverse } x / (a * x \text{ powr } (a-1)))$   
*at-top*  
**using** *eventually-gt-at-top*[*of 0::real*]  
**by** (*elim eventually-mono*) (*simp add: field-simps powr-diff powr-minus*)  
**moreover from** *assms* **have**  $((\lambda x. \text{inverse } a * x \text{ powr } -a) \longrightarrow 0)$  *at-top*  
**by** (*intro tendsto-mult-right-zero tendsto-neg-powr filterlim-ident*) *simp-all*  
**ultimately show**  $((\lambda x. \text{inverse } x / (a * x \text{ powr } (a - 1))) \longrightarrow 0)$  *at-top*  
**by** (*subst (asm) tendsto-cong*) *simp-all*  
**qed**

**lemma** *tendsto-ln-powr-over-powr*:  
**assumes**  $(a::real) > 0$   $b > 0$   
**shows**  $((\lambda x. \ln x \text{ powr } a / x \text{ powr } b) \longrightarrow 0)$  *at-top*  
**proof** –  
**have** *eventually*  $(\lambda x. \ln x \text{ powr } a / x \text{ powr } b = (\ln x / x \text{ powr } (b/a)) \text{ powr } a)$   
*at-top*  
**using** *assms eventually-gt-at-top*[*of 1::real*]  
**by** (*elim eventually-mono*) (*simp add: powr-divide powr-powr*)  
**moreover have** *eventually*  $(\lambda x. 0 < \ln x / x \text{ powr } (b / a))$  *at-top*  
**using** *eventually-gt-at-top*[*of 1::real*] **by** (*elim eventually-mono*) *simp*  
**with** *assms* **have**  $((\lambda x. (\ln x / x \text{ powr } (b/a)) \text{ powr } a) \longrightarrow 0)$  *at-top*  
**by** (*intro tendsto-zero-powrI tendsto-ln-over-powr*) (*simp-all add: eventually-mono*)  
**ultimately show** *?thesis* **by** (*subst tendsto-cong*) *simp-all*  
**qed**

**lemma** *tendsto-ln-powr-over-powr'*:  
**assumes**  $b > 0$   
**shows**  $((\lambda x::real. \ln x \text{ powr } a / x \text{ powr } b) \longrightarrow 0)$  *at-top*  
**proof** (*cases a ≤ 0*)  
**assume**  $a: a \leq 0$   
**show** *?thesis*  
**proof** (*rule tendsto-sandwich*[*of λ::real. 0*])  
**have** *eventually*  $(\lambda x. \ln x \text{ powr } a \leq 1)$  *at-top* **unfolding** *eventually-at-top-linorder*  
**proof** (*intro allI exI impI*)  
**fix**  $x :: real$  **assume**  $x: x \geq \exp 1$   
**have**  $0 < \exp (1::real)$  **by** *simp*

**also have**  $\dots \leq x$  **by fact**  
**finally have**  $\ln x \geq \ln (exp\ 1)$  **using**  $x$  **by** *(subst ln-le-cancel-iff)* *auto*  
**hence**  $\ln x\ powr\ a \leq \ln (exp\ 1)\ powr\ a$  **using**  $a$  **by** *(intro powr-mono2')*  
*simp-all*  
**thus**  $\ln x\ powr\ a \leq 1$  **by** *simp*  
**qed**  
**thus** *eventually*  $(\lambda x. \ln x\ powr\ a / x\ powr\ b \leq x\ powr\ -b)$  *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)*

**lemma** *tendsto-ln-over-ln*:  
**assumes**  $(a::real) > 0\ c > 0$   
**shows**  $((\lambda x. \ln (a*x) / \ln (c*x)) \longrightarrow 1)$  *at-top*  
**proof** *(rule lhospital-at-top-at-top)*  
**show** *LIM*  $x$  *at-top*.  $\ln (c*x) :>$  *at-top*  
**by** *(intro filterlim-compose[OF ln-at-top] filterlim-tendsto-pos-mult-at-top[OF tendsto-const])*  
*filterlim-ident assms(2)*  
**show** *eventually*  $(\lambda x. ((\lambda x. \ln (a*x))\ has-real-derivative\ (inverse\ x))\ (at\ x))$  *at-top*  
**using** *eventually-gt-at-top[of inverse a] assms*  
**by** *(auto elim!: eventually-mono intro!: derivative-eq-intros simp: field-simps)*  
**show** *eventually*  $(\lambda x. ((\lambda x. \ln (c*x))\ has-real-derivative\ (inverse\ x))\ (at\ x))$  *at-top*  
**using** *eventually-gt-at-top[of inverse c] assms*  
**by** *(auto elim!: eventually-mono intro!: derivative-eq-intros simp: field-simps)*  
**show**  $((\lambda x::real. inverse\ x / inverse\ x) \longrightarrow 1)$  *at-top*  
**by** *(subst tendsto-cong[of - λ-. 1]) simp-all*  
**qed** *simp-all*

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

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

**lemma** *tendsto-ln-powr-over-ln-powr''*:  
 $a > 0 \implies ((\lambda x::real. \ln (a*x)\ powr\ d / \ln x\ powr\ d) \longrightarrow 1)$  *at-top*

**using** *tendsto-ln-powr-over-ln-powr*[of - 1] **by** *simp*

**lemma** *bigheta-const-ln-powr* [*simp*]:  $a > 0 \implies (\lambda x::real. \ln (a*x) \text{ powr } d) \in \Theta(\lambda x. \ln x \text{ powr } d)$

**by** (*intro bighetaI-tendsto*[of 1] *tendsto-ln-powr-over-ln-powr'*) *simp*

**lemma** *bigheta-const-ln-pow* [*simp*]:  $a > 0 \implies (\lambda x::real. \ln (a*x) \wedge d) \in \Theta(\lambda x. \ln x \wedge d)$

**proof** –

**assume**  $a > 0$

**have**  $\forall_F x \text{ in at-top. } \ln (a * x) \wedge d = \ln (a * x) \text{ powr } real \ d$

**using** *eventually-gt-at-top*[of 1/a]

**by** *eventually-elim* (*insert a, subst powr-realpow, auto simp: field-simps*)

**hence**  $(\lambda x::real. \ln (a*x) \wedge d) \in \Theta(\lambda x. \ln (a*x) \text{ powr } real \ d)$

**by** (*rule bighetaI-cong*)

**also from**  $a$  **have**  $(\lambda x. \ln (a*x) \text{ powr } real \ d) \in \Theta(\lambda x. \ln x \text{ powr } real \ d)$  **by** *simp*

**also have**  $\forall_F x \text{ in at-top. } \ln x \text{ powr } real \ d = \ln x \wedge d$

**using** *eventually-gt-at-top*[of 1]

**by** *eventually-elim* (*subst powr-realpow, auto simp: field-simps*)

**hence**  $(\lambda x. \ln x \text{ powr } real \ d) \in \Theta(\lambda x. \ln x \wedge d)$

**by** (*rule bighetaI-cong*)

**finally show** *?thesis* .

**qed**

**lemma** *bigheta-const-ln* [*simp*]:  $a > 0 \implies (\lambda x::real. \ln (a*x)) \in \Theta(\lambda x. \ln x)$

**using** *tendsto-ln-over-ln*[of a 1] **by** (*intro bighetaI-tendsto*[of 1]) *simp-all*

If there are two functions  $f$  and  $g$  where any power of  $g$  is asymptotically smaller than  $f$ , propositions like  $(\lambda x. (f x)^{p1} * (g x)^{q1}) \in O(\lambda x. (f x)^{p2} * (g x)^{q2})$  can be decided just by looking at the exponents: the proposition is true iff  $p1 < p2$  or  $p1 = p2 \wedge q1 \leq q2$ .

The functions  $\lambda x. x, \ln, \lambda x. \ln (\ln x), \dots$  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.

## 2.4 Decision procedure

**definition** *powr-closure f*  $\equiv \{\lambda x. f x \text{ powr } p :: real \mid p. \text{ True}\}$

**lemma** *powr-closureI* [*simp*]:  $(\lambda x. f x \text{ powr } p) \in \text{powr-closure } f$

**unfolding** *powr-closure-def* **by** *force*

**lemma** *powr-closureE*:

**assumes**  $g \in \text{powr-closure } f$   
**obtains**  $p$  **where**  $g = (\lambda x. f x \text{ powr } p)$   
**using** *assms* **unfolding** *powr-closure-def* **by** *force*

**locale** *landau-function-family* =  
**fixes**  $F :: 'a \text{ filter}$  **and**  $H :: ('a \Rightarrow \text{real}) \text{ set}$   
**assumes** *F-nontrivial*:  $F \neq \text{bot}$   
**assumes** *pos*:  $h \in H \Longrightarrow \text{eventually } (\lambda x. h x > 0) F$   
**assumes** *linear*:  $h1 \in H \Longrightarrow h2 \in H \Longrightarrow h1 \in o[F](h2) \vee h2 \in o[F](h1) \vee h1 \in \Theta[F](h2)$   
**assumes** *mult*:  $h1 \in H \Longrightarrow h2 \in H \Longrightarrow (\lambda x. h1 x * h2 x) \in H$   
**assumes** *inverse*:  $h \in H \Longrightarrow (\lambda x. \text{inverse } (h x)) \in H$   
**begin**

**lemma** *div*:  $h1 \in H \Longrightarrow h2 \in H \Longrightarrow (\lambda x. h1 x / h2 x) \in H$   
**by** (*subst divide-inverse*) (*intro mult inverse*)

**lemma** *nonzero*:  $h \in H \Longrightarrow \text{eventually } (\lambda x. h x \neq 0) F$   
**by** (*drule pos*) (*auto elim: eventually-mono*)

**lemma** *landau-cases*:  
**assumes**  $h1 \in H$   $h2 \in H$   
**obtains**  $h1 \in o[F](h2) \mid h2 \in o[F](h1) \mid h1 \in \Theta[F](h2)$   
**using** *linear[OF assms]* **by** *blast*

**lemma** *small-big-antisym*:  
**assumes**  $h1 \in H$   $h2 \in H$   $h1 \in o[F](h2)$   $h2 \in O[F](h1)$  **shows** *False*  
**proof** –  
**from** *nonzero[OF assms(1)] nonzero[OF assms(2)] landau-o.small-big-asymmetric[OF assms(3,4)]*  
**have** *eventually*  $(\lambda :: 'a. \text{False}) F$  **by** *eventually-elim simp*  
**thus** *False* **by** (*simp add: eventually-False F-nontrivial*)  
**qed**

**lemma** *small-antisym*:  
**assumes**  $h1 \in H$   $h2 \in H$   $h1 \in o[F](h2)$   $h2 \in o[F](h1)$  **shows** *False*  
**using** *assms* **by** (*blast intro: small-big-antisym landau-o.small-imp-big*)

**end**

**locale** *landau-function-family-pair* =  
 $G: \text{landau-function-family } F$   $G + H: \text{landau-function-family } F$   $H$  **for**  $F$   $G$   $H +$   
**fixes**  $g$   
**assumes** *gs-dominate*:  $g1 \in G \Longrightarrow g2 \in G \Longrightarrow h1 \in H \Longrightarrow h2 \in H \Longrightarrow g1 \in o[F](g2) \Longrightarrow$   
 $(\lambda x. g1 x * h1 x) \in o[F](\lambda x. g2 x * h2 x)$   
**assumes**  $g: g \in G$   
**assumes** *g-dominates*:  $h \in H \Longrightarrow h \in o[F](g)$

**begin**

**sublocale**  $GH$ : *landau-function-family*  $F G * H$

**proof** (*unfold-locales*; (*elim set-times-elim*; *hypsubst*)?)

**fix**  $g h$  **assume**  $g \in G h \in H$

**from**  $G.pos[OF this(1)] H.pos[OF this(2)]$  **show** *eventually*  $(\lambda x. (g*h) x > 0)$   
 $F$

**by** *eventually-elim simp*

**next**

**fix**  $g h$  **assume**  $A: g \in G h \in H$

**have**  $(\lambda x. inverse ((g * h) x)) = (\lambda x. inverse (g x)) * (\lambda x. inverse (h x))$  **by**  
(*rule ext*) *simp*

**also from**  $A$  **have**  $\dots \in G * H$  **by** (*intro G.inverse H.inverse set-times-intro*)

**finally show**  $(\lambda x. inverse ((g * h) x)) \in G * H$  .

**next**

**fix**  $g1 g2 h1 h2$  **assume**  $A: g1 \in G g2 \in G h1 \in H h2 \in H$

**from**  $gs-dominate[OF this] gs-dominate[OF this(2,1,4,3)]$

$G.linear[OF this(1,2)] H.linear[OF this(3,4)]$

**show**  $g1 * h1 \in o[F](g2 * h2) \vee g2 * h2 \in o[F](g1 * h1) \vee g1 * h1 \in \Theta[F](g2 * h2)$

**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**  $B: (\lambda x. (g1 * h1) x * (g2 * h2) x) = (g1 * g2) * (h1 * h2)$

**by** (*rule ext*) (*simp add: func-times mult-ac*)

**from**  $A$  **show**  $(\lambda x. (g1 * h1) x * (g2 * h2) x) \in G * H$

**by** (*subst B, intro set-times-intro*) (*auto intro: G.mult H.mult simp: func-times*)

**qed** (*fact G.F-nontrivial*)

**lemma** *smallo-iff*:

**assumes**  $g1 \in G g2 \in G h1 \in H h2 \in H$

**shows**  $(\lambda x. g1 x * h1 x) \in o[F](\lambda x. g2 x * h2 x) \longleftrightarrow$

$g1 \in o[F](g2) \vee (g1 \in \Theta[F](g2) \wedge h1 \in o[F](h2))$  (**is**  $?P \longleftrightarrow ?Q$ )

**proof** (*rule G.landau-cases[OF assms(1,2)]*)

**assume**  $g1 \in o[F](g2)$

**thus**  $?thesis$  **by** (*auto intro!: gs-dominate assms*)

**next**

**assume**  $A: g1 \in \Theta[F](g2)$

**hence**  $B: g2 \in O[F](g1)$  **by** (*subst (asm) bigtheta-sym*) (*rule bigthetaD1*)

**hence**  $g1 \notin o[F](g2)$  **using** *assms* **by** (*auto dest: G.small-big-antisym*)

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

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

**ultimately show**  $?thesis$  **using**  $G.nonzero[OF assms(1)] A$

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

**next**

**assume**  $A: g2 \in o[F](g1)$

**from**  $gs-dominate[OF assms(2,1,4,3) this]$  **have**  $B: g2 * h2 \in o[F](g1 * h1)$

**by** (*simp add: func-times*)

**have**  $g1 \notin o[F](g2) g1 \notin \Theta[F](g2)$  **using** *assms A*

by (auto dest: G.small-antisym G.small-big-antisym simp: bigomega-iff-bigo)  
 moreover have  $\neg ?P$   
 by (intro notI GH.small-antisym[OF - - B] set-times-intro) (simp-all add:  
 func-times assms)  
 ultimately show ?thesis by blast  
 qed

lemma bigo-iff:

assumes  $g1 \in G$   $g2 \in G$   $h1 \in H$   $h2 \in H$   
 shows  $(\lambda x. g1 x * h1 x) \in O[F](\lambda x. g2 x * h2 x) \longleftrightarrow$   
 $g1 \in o[F](g2) \vee (g1 \in \Theta[F](g2) \wedge h1 \in O[F](h2))$  (is  $?P \longleftrightarrow ?Q$ )  
 proof (rule G.landau-cases[OF assms(1,2)])  
 assume  $g1 \in o[F](g2)$   
 thus ?thesis by (auto intro!: gs-dominate assms landau-o.small-imp-big)  
 next  
 assume A:  $g2 \in o[F](g1)$   
 hence  $g1 \notin O[F](g2)$  using assms by (auto dest: G.small-big-antisym)  
 moreover from gs-dominate[OF assms(2,1,4,3) A] have  $g2 * h2 \in o[F](g1 * h1)$   
 by (simp add: func-times)  
 hence  $g1 * h1 \notin O[F](g2 * h2)$  by (blast intro: GH.small-big-antisym assms)  
 ultimately show ?thesis using A assms  
 by (auto simp: func-times dest: landau-o.small-imp-big)  
 next  
 assume A:  $g1 \in \Theta[F](g2)$   
 hence  $g1 \notin o[F](g2)$  unfolding bigtheta-def using assms  
 by (auto dest: G.small-big-antisym simp: bigomega-iff-bigo)  
 moreover have  $O[F](\lambda x. g2 x * h2 x) = O[F](\lambda x. g1 x * h2 x)$   
 by (subst landau-o.big.cong-bigtheta[OF landau-theta.mult-right[OF A]]) (rule  
 refl)  
 ultimately show ?thesis using A G.nonzero[OF assms(2)]  
 by (auto simp: landau-o.big.mult-cancel-left eventually-nonzero-bigtheta)  
 qed

lemma bigtheta-iff:

$g1 \in G \implies g2 \in G \implies h1 \in H \implies h2 \in H \implies$   
 $(\lambda x. g1 x * h1 x) \in \Theta[F](\lambda x. g2 x * h2 x) \longleftrightarrow g1 \in \Theta[F](g2) \wedge h1 \in \Theta[F](h2)$   
 by (auto simp: bigtheta-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  $F \neq \text{bot}$  filterlim f at-top F  
 shows landau-function-family F (powr-closure f)  
 proof (unfold locales; (elim powr-closureE; hypsubst) ?)  
 from assms have eventually  $(\lambda x. f x \geq 1)$  F using filterlim-at-top by auto  
 hence A: eventually  $(\lambda x. f x \neq 0)$  F by eventually-elim simp  
 {

```

fix p q :: real
show ( $\lambda x. f\ x\ \text{powr}\ p$ )  $\in o[F](\lambda x. f\ x\ \text{powr}\ q) \vee$ 
      ( $\lambda x. f\ x\ \text{powr}\ q$ )  $\in o[F](\lambda x. f\ x\ \text{powr}\ p) \vee$ 
      ( $\lambda x. f\ x\ \text{powr}\ p$ )  $\in \Theta[F](\lambda x. f\ x\ \text{powr}\ q)$ 
by (cases p q rule: linorder-cases)
      (force intro!: smalloI-tendsto tendsto-neg-powr simp: powr-diff [symmetric]
assms A)+
}
fix p
show eventually ( $\lambda x. f\ x\ \text{powr}\ p > 0$ ) F using A by simp
qed (auto simp: powr-add[symmetric] powr-minus[symmetric]  $\langle F \neq \text{bot} \rangle$  intro:
powr-closureI)

lemma landau-function-family-pair-trans:
  assumes landau-function-family-pair Ftr F G f
  assumes landau-function-family-pair Ftr G H g
  shows landau-function-family-pair Ftr F (G*H) f
proof -
  interpret FG: landau-function-family-pair Ftr F G f by fact
  interpret GH: landau-function-family-pair Ftr G H g by fact
  show ?thesis
  proof (unfold-locales; (elim set-times-elim)?; (clarify)?;
      (unfold func-times mult.assoc[symmetric])?)
  fix f1 f2 g1 g2 h1 h2
  assume A: f1  $\in F$  f2  $\in F$  g1  $\in G$  g2  $\in G$  h1  $\in H$  h2  $\in H$  f1  $\in o[Ftr](f2)$ 

  from A have ( $\lambda x. f1\ x * g1\ x * h1\ x$ )  $\in o[Ftr](\lambda x. f1\ x * g1\ x * g\ x)$ 
  by (intro landau-o.small.mult-left GH.g-dominates)
  also have ( $\lambda x. f1\ x * g1\ x * g\ x$ ) = ( $\lambda x. f1\ x * (g1\ x * g\ x)$ ) by (simp only:
mult.assoc)
  also from A have ...  $\in o[Ftr](\lambda x. f2\ x * (g2\ x / g\ x))$ 
  by (intro FG.gs-dominate FG.H.mult FG.H.div GH.g)
  also from A have ( $\lambda x. \text{inverse}\ (h2\ x)$ )  $\in o[Ftr](g)$  by (intro GH.g-dominates
GH.H.inverse)
  with GH.g A have ( $\lambda x. f2\ x * (g2\ x / g\ x)$ )  $\in o[Ftr](\lambda x. f2\ x * (g2\ x * h2\ x))$ 
  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 ... =  $o[Ftr](\lambda x. f2\ x * g2\ x * h2\ x)$  by (simp only: mult.assoc)
  finally show ( $\lambda x. f1\ x * g1\ x * h1\ x$ )  $\in o[Ftr](\lambda x. f2\ x * g2\ x * h2\ x)$  .
next
  fix g1 h1 assume A: g1  $\in G$  h1  $\in H$ 
  hence ( $\lambda x. g1\ x * h1\ x$ )  $\in o[Ftr](\lambda x. g1\ x * g\ x)$ 
  by (intro landau-o.small.mult-left GH.g-dominates)
  also from A have ( $\lambda x. g1\ x * g\ x$ )  $\in o[Ftr](f)$  by (intro FG.g-dominates
FG.H.mult GH.g)
  finally show ( $\lambda x. g1\ x * h1\ x$ )  $\in o[Ftr](f)$  .
  qed (simp-all add: FG.g)
qed

```

**lemma** *landau-function-family-pair-trans-powr*:  
**assumes** *landau-function-family-pair*  $F$  (*powr-closure*  $g$ )  $H$  ( $\lambda x. g\ x\ \text{powr}\ 1$ )  
**assumes** *filterlim*  $f$  *at-top*  $F$   
**assumes**  $\bigwedge p. (\lambda x. g\ x\ \text{powr}\ p) \in o[F](f)$   
**shows** *landau-function-family-pair*  $F$  (*powr-closure*  $f$ ) (*powr-closure*  $g * H$ ) ( $\lambda x. f\ x\ \text{powr}\ 1$ )  
**proof** (*rule* *landau-function-family-pair-trans*[*OF - assms(1)*])  
**interpret**  $GH$ : *landau-function-family-pair*  $F$  *powr-closure*  $g$   $H$   $\lambda x. g\ x\ \text{powr}\ 1$   
**by** *fact*  
**interpret**  $F$ : *landau-function-family*  $F$  *powr-closure*  $f$   
**by** (*rule* *landau-function-family-powr-closure*) (*rule*  $GH.G.F\text{-nontrivial}$ , *rule* *assms*)  
**show** *landau-function-family-pair*  $F$  (*powr-closure*  $f$ ) (*powr-closure*  $g$ ) ( $\lambda x. f\ x\ \text{powr}\ 1$ )  
**proof** (*unfold-locales*; (*elim* *powr-closureE*; *hypsubst*)?)  
**show** ( $\lambda x. f\ x\ \text{powr}\ 1$ )  $\in$  *powr-closure*  $f$  **by** (*rule* *powr-closureI*)  
**next**  
**fix**  $p :: \text{real}$   
**note** *assms(3)*[*of*  $p$ ]  
**also from** *assms(2)* **have** *eventually* ( $\lambda x. f\ x \geq 1$ )  $F$  **by** (*force simp*: *filterlim-at-top*)  
**hence**  $f \in \Theta[F](\lambda x. f\ x\ \text{powr}\ 1)$  **by** (*auto intro!*: *bighetaI-cong* *elim!*: *eventually-mono*)  
**finally show** ( $\lambda x. g\ x\ \text{powr}\ p$ )  $\in$   $o[F](\lambda x. f\ x\ \text{powr}\ 1)$  .  
**next**  
**fix**  $p\ p1\ p2\ p3 :: \text{real}$   
**assume**  $A$ : ( $\lambda x. f\ x\ \text{powr}\ p$ )  $\in$   $o[F](\lambda x. f\ x\ \text{powr}\ p1)$   
**have**  $p$ :  $p < p1$   
**proof** (*cases*  $p\ p1$  *rule*: *linorder-cases*)  
**assume**  $p > p1$   
**moreover from** *assms(2)* **have** *eventually* ( $\lambda x. f\ x \geq 1$ )  $F$   
**by** (*force simp*: *filterlim-at-top*)  
**hence** *eventually* ( $\lambda x. f\ x \neq 0$ )  $F$  **by** *eventually-elim simp*  
**ultimately have** ( $\lambda x. f\ x\ \text{powr}\ p1$ )  $\in$   $o[F](\lambda x. f\ x\ \text{powr}\ p)$  **using** *assms*  
**by** (*auto intro!*: *smalloI-tendsto* *tendsto-neg-powr* *simp*: *powr-diff* [*symmetric*])  
**from**  $F$ .*small-antisym*[*OF - - this*  $A$ ] **show** *?thesis* **by** (*auto simp*: *powr-closureI*)  
**next**  
**assume**  $p = p1$   
**hence** ( $\lambda x. f\ x\ \text{powr}\ p1$ )  $\in$   $O[F](\lambda x. f\ x\ \text{powr}\ p)$  **by** (*intro bighetaD1*) *simp*  
**with**  $F$ .*small-big-antisym*[*OF - - A this*] **show** *?thesis* **by** (*auto simp*: *powr-closureI*)  
**qed**

**from** *assms(2)* **have** *f-pos*: *eventually* ( $\lambda x. f\ x \geq 1$ )  $F$  **by** (*force simp*: *filterlim-at-top*)  
**from** *assms* **have** ( $\lambda x. g\ x\ \text{powr}\ ((p2 - p3)/(p1 - p))$ )  $\in$   $o[F](f)$  **by** *simp*  
**from** *smallo-powr*[*OF this*, *of*  $p1 - p$ ]  $p$   
**have** ( $\lambda x. g\ x\ \text{powr}\ (p2 - p3)$ )  $\in$   $o[F](\lambda x. |f\ x|\ \text{powr}\ (p1 - p))$  **by** (*simp*)

*add: powr-powr*  
**hence**  $(\lambda x. |f x| \text{ powr } p * g x \text{ powr } p2) \in o[F](\lambda x. |f x| \text{ powr } p1 * g x \text{ powr } p3)$   
*(is ?P)*  
**using** *GH.G.nonzero[OF GH.g] F.nonzero[OF powr-closureI]*  
**by** *(simp add: powr-diff landau-o.small.divide-eq1*  
*landau-o.small.divide-eq2 mult.commute)*  
**also have**  $?P \longleftrightarrow (\lambda x. f x \text{ powr } p * g x \text{ powr } p2) \in o[F](\lambda x. f x \text{ powr } p1 * g$   
 $x \text{ powr } p3)$   
**using** *f-pos* **by** *(intro landau-o.small.cong-ex) (auto elim!: eventually-mono)*  
**finally show**  $(\lambda x. f x \text{ powr } p * g x \text{ powr } p2) \in o[F](\lambda x. f x \text{ powr } p1 * g x \text{ powr } p3)$  .  
**qed**  
**qed**

**definition** *dominates* :: *'a filter*  $\Rightarrow$  *('a  $\Rightarrow$  real)*  $\Rightarrow$  *('a  $\Rightarrow$  real)*  $\Rightarrow$  *bool* **where**  
*dominates*  $F f g = (\forall p. (\lambda x. g x \text{ powr } p) \in o[F](f))$

**lemma** *dominates-trans:*

**assumes** *eventually*  $(\lambda x. g x > 0)$   $F$   
**assumes** *dominates*  $F f g$  *dominates*  $F g h$   
**shows** *dominates*  $F f h$   
**unfolding** *dominates-def*  
**proof**  
**fix**  $p :: \text{real}$   
**from** *assms(3)* **have**  $(\lambda x. h x \text{ powr } p) \in o[F](g)$  **unfolding** *dominates-def* **by**  
*simp*  
**also from** *assms(1)* **have**  $g \in \Theta[F](\lambda x. g x \text{ powr } 1)$   
**by** *(intro bighetaI-cong) (auto elim!: eventually-mono)*  
**also from** *assms(2)* **have**  $(\lambda x. g x \text{ powr } 1) \in o[F](f)$   
**using** *dominates-def* **by** *blast*  
**finally show**  $(\lambda x. h x \text{ powr } p) \in o[F](f)$  .  
**qed**

**fun** *landau-dominating-chain* **where**

*landau-dominating-chain*  $F (f \# g \# gs) \longleftrightarrow$   
*dominates*  $F f g \wedge \text{landau-dominating-chain } F (g \# gs)$   
 $| \text{landau-dominating-chain } F [f] \longleftrightarrow (\lambda x. 1) \in o[F](f)$   
 $| \text{landau-dominating-chain } F [] \longleftrightarrow \text{True}$

**primrec** *landau-dominating-chain'* **where**

*landau-dominating-chain'*  $F [] \longleftrightarrow \text{True}$   
 $| \text{landau-dominating-chain}' F (f \# gs) \longleftrightarrow$   
*landau-function-family-pair*  $F (\text{powr-closure } f) (\text{prod-list } (\text{map } \text{powr-closure } gs))$   
 $(\lambda x. f x \text{ powr } 1) \wedge$   
*landau-dominating-chain'*  $F gs$

**primrec nonneg-list where**  
*nonneg-list* []  $\longleftrightarrow$  *True*  
| *nonneg-list* (*x#xs*)  $\longleftrightarrow$   $x > 0 \vee (x = 0 \wedge \text{nonneg-list } xs)$

**primrec pos-list where**  
*pos-list* []  $\longleftrightarrow$  *False*  
| *pos-list* (*x#xs*)  $\longleftrightarrow$   $x > 0 \vee (x = 0 \wedge \text{pos-list } xs)$

**lemma dominating-chain-imp-dominating-chain':**

$Ftr \neq bot \implies (\bigwedge g. g \in \text{set } gs \implies \text{filterlim } g \text{ at-top } Ftr) \implies$   
 $\text{landau-dominating-chain } Ftr \text{ } gs \implies \text{landau-dominating-chain}' Ftr \text{ } gs$

**proof** (*induction gs rule: landau-dominating-chain.induct*)

**case** (*1 F f g gs*)

**from** *1 show ?case*

**by** (*auto intro!: landau-function-family-pair-trans-powr simp add: dominates-def simp flip: powr-one'*)

**next**

**case** (*2 F f*)

**then interpret** *F: landau-function-family F powr-closure f*

**by** (*intro landau-function-family-powr-closure simp-all*)

**from** *2 have eventually* ( $\lambda x. f \ x \geq 1$ ) *F by* (*force simp: filterlim-at-top*)

**hence**  $o[F](\lambda x. f \ x \ \text{powr } 1) = o[F](\lambda x. f \ x)$

**by** (*intro landau-o.small.cong*) (*auto elim!: eventually-mono*)

**with** *2 have landau-function-family-pair F (powr-closure f) {λ-. 1} (λx. f x powr 1)*

**by** *unfold-locales (auto intro: powr-closureI simp flip: powr-one')*

**thus** *?case by (simp add: one-fun-def)*

**next**

**case** *3*

**then show** *?case by simp*

**qed**

**locale** *landau-function-family-chain =*

**fixes** *F :: 'b filter*

**fixes** *gs :: 'a list*

**fixes** *get-param :: 'a  $\Rightarrow$  real*

**fixes** *get-fun :: 'a  $\Rightarrow$  ('b  $\Rightarrow$  real)*

**assumes** *F-nontrivial: F  $\neq$  bot*

**assumes** *gs-pos: g  $\in$  set (map get-fun gs)  $\implies$  filterlim g at-top F*

**assumes** *dominating-chain: landau-dominating-chain F (map get-fun gs)*

**begin**

**lemma** *dominating-chain': landau-dominating-chain' F (map get-fun gs)*

**by** (*intro dominating-chain-imp-dominating-chain' gs-pos dominating-chain F-nontrivial*)

**lemma** *gs-powr-0-eq-one:*

*eventually* ( $\lambda x. (\prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } 0) = 1$ ) *F*

**using** *gs-pos*  
**proof** (*induction gs*)  
    **case** (*Cons g gs*)  
        **from** *Cons* **have** *eventually* ( $\lambda x. \text{get-fun } g \ x \ > \ 0$ ) *F* **by** (*auto simp: filter-lim-at-top-dense*)  
        **moreover from** *Cons* **have** *eventually* ( $\lambda x. (\prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } 0) = 1$ )  
        *F* **by** *simp*  
        **ultimately show** *?case* **by** *eventually-elim simp*  
**qed** *simp-all*

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

**lemma** *smallo-iff*:  
 $(\lambda \cdot. 1) \in o[F](\lambda x. \prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } \text{get-param } g) \longleftrightarrow \text{pos-list } (\text{map } \text{get-param } gs)$   
**proof** –  
    **have**  $((\lambda \cdot. 1) \in o[F](\lambda x. \prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } \text{get-param } g)) \longleftrightarrow$   
     $((\lambda x. \prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } 0) \in o[F](\lambda x. \prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } \text{get-param } g))$   
    **by** (*rule sym, intro landau-o.small.in-cong gs-powr-0-eq-one*)  
    **also from** *gs-pos dominating-chain'* **have** ...  $\longleftrightarrow \text{pos-list } (\text{map } \text{get-param } gs)$   
    **proof** (*induction gs*)  
    **case** *Nil*  
    **have**  $(\lambda x :: 'b. 1 :: \text{real}) \notin o[F](\lambda x. 1)$  **using** *F-nontrivial*  
    **by** (*auto dest!: landau-o.small-big-asymmetric*)  
    **thus** *?case* **by** *simp*  
    **next**  
    **case** (*Cons g gs*)  
    **then interpret** *G: landau-function-family-pair F powr-closure (get-fun g)*  
     $\text{prod-list } (\text{map } \text{powr-closure } (\text{map } \text{get-fun } gs)) \ \lambda x. \text{get-fun } g \ x \ \text{powr } 1$  **by** *simp*  
    **from** *Cons* **show** *?case* **using** *listmap-gs-in-listmap[of get-fun - gs] F-nontrivial*  
    **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* .  
**qed**

**lemma** *bigO-iff*:  
 $(\lambda-. 1) \in O[F](\lambda x. \prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } \text{get-param } g) \longleftrightarrow \text{nonneg-list } (\text{map } \text{get-param } gs)$

**proof** –  
**have**  $((\lambda-. 1) \in O[F](\lambda x. \prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } \text{get-param } g)) \longleftrightarrow$   
 $((\lambda x. \prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } 0) \in O[F](\lambda x. \prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } \text{get-param } g))$   
**by** (*rule sym, intro landau-o.big.in-cong gs-powr-0-eq-one*)  
**also from** *gs-pos dominating-chain'* **have**  $\dots \longleftrightarrow \text{nonneg-list } (\text{map } \text{get-param } gs)$   
**proof** (*induction gs*)  
**case** *Nil*  
**then show** *?case* **by** (*simp add: func-one*)  
**next**  
**case** (*Cons g gs*)  
**then interpret** *G: landau-function-family-pair F powr-closure (get-fun g)*  
*prod-list (map powr-closure (map get-fun gs))*  $\lambda x. \text{get-fun } g \ x \ \text{powr } 1$  **by** *simp*  
**from** *Cons* **show** *?case* **using** *listmap-gs-in-listmap[of get-fun - gs] F-nontrivial*  
**by** (*simp-all add: G.bigO-iff listmap-gs-in-listmap powr-smallo-iff powr-bigtheta-iff*  
*del: powr-zero-eq-one*)  
**qed**  
**finally show** *?thesis* .  
**qed**

**lemma** *bigtheta-iff*:  
 $(\lambda-. 1) \in \Theta[F](\lambda x. \prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } \text{get-param } g) \longleftrightarrow \text{list-all } ((=) 0)$   
 $(\text{map } \text{get-param } gs)$

**proof** –  
**have**  $((\lambda-. 1) \in \Theta[F](\lambda x. \prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } \text{get-param } g)) \longleftrightarrow$   
 $((\lambda x. \prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } 0) \in \Theta[F](\lambda x. \prod g \leftarrow gs. \text{get-fun } g \ x \ \text{powr } \text{get-param } g))$   
**by** (*rule sym, intro landau-theta.in-cong gs-powr-0-eq-one*)  
**also from** *gs-pos dominating-chain'* **have**  $\dots \longleftrightarrow \text{list-all } ((=) 0)$   $(\text{map } \text{get-param } gs)$   
**proof** (*induction gs*)  
**case** *Nil*  
**then show** *?case* **by** (*simp add: func-one*)  
**next**  
**case** (*Cons g gs*)  
**then interpret** *G: landau-function-family-pair F powr-closure (get-fun g)*  
*prod-list (map powr-closure (map get-fun gs))*  $\lambda x. \text{get-fun } g \ x \ \text{powr } 1$  **by** *simp*  
**from** *Cons* **show** *?case* **using** *listmap-gs-in-listmap[of get-fun - gs] F-nontrivial*  
**by** (*simp-all add: G.bigtheta-iff listmap-gs-in-listmap powr-smallo-iff powr-bigtheta-iff*  
*del: powr-zero-eq-one*)  
**qed**  
**finally show** *?thesis* .  
**qed**

**end**

**lemma** *fun-chain-at-top-at-top*:  
**assumes** *filterlim* ( $f :: ('a::order) \Rightarrow 'a$ ) *at-top at-top*  
**shows** *filterlim* ( $f \rightsquigarrow n$ ) *at-top at-top*  
**by** (*induction n*) (*auto intro: filterlim-ident filterlim-compose[OF assms]*)

**lemma** *const-smallo-ln-chain*:  $(\lambda-. 1) \in o((\text{ln}::\text{real} \Rightarrow \text{real}) \rightsquigarrow n)$   
**proof** (*intro smalloI-tendsto*)  
**show**  $((\lambda x::\text{real}. 1 / (\text{ln} \rightsquigarrow n) x) \longrightarrow 0)$  *at-top*  
**by** (*rule tendsto-divide-0 tendsto-const filterlim-at-top-imp-at-infinity*  
*fun-chain-at-top-at-top ln-at-top*)  
**next**  
**from** *fun-chain-at-top-at-top*[*OF ln-at-top, of n*]  
**have** *eventually*  $(\lambda x::\text{real}. (\text{ln} \rightsquigarrow n) x > 0)$  *at-top* **by** (*simp add: filterlim-at-top-dense*)  
**thus** *eventually*  $(\lambda x::\text{real}. (\text{ln} \rightsquigarrow n) x \neq 0)$  *at-top* **by** *eventually-elim simp-all*  
**qed**

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

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

```

datatype primfun = LnChain nat

instantiation primfun :: linorder
begin

fun less-eq-primfun :: primfun  $\Rightarrow$  primfun  $\Rightarrow$  bool where
  LnChain x  $\leq$  LnChain y  $\longleftrightarrow$  x  $\leq$  y

fun less-primfun :: primfun  $\Rightarrow$  primfun  $\Rightarrow$  bool where
  LnChain x < LnChain y  $\longleftrightarrow$  x < y

instance
proof (standard, goal-cases)
  case (1 x y) show ?case by (induction x y rule: less-eq-primfun.induct) auto
next
  case (2 x) show ?case by (cases x) auto
next
  case (3 x y z) thus ?case
  by (induction x y rule: less-eq-primfun.induct, cases z) auto
next
  case (4 x y) thus ?case by (induction x y rule: less-eq-primfun.induct) auto
next
  case (5 x y) thus ?case by (induction x y rule: less-eq-primfun.induct) auto
qed

end

fun eval-primfun' :: -  $\Rightarrow$  -  $\Rightarrow$  real where
  eval-primfun' (LnChain n) = ( $\lambda$ x. (ln  $\widehat{\sim}$  n) x)

fun eval-primfun :: -  $\Rightarrow$  -  $\Rightarrow$  real where
  eval-primfun (f, e) = ( $\lambda$ x. 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 ( $\bigwedge n e. P (LnChain\ n, e)$ )
  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 intro!: fun-chain-at-top-at-top ln-at-top)

lemma primfun-dominates:
   $f < g \implies \text{dominates at-top (eval-primfun' f) (eval-primfun' g)}$ 
  by (elim less-primfun.elims; hypsubst) (simp-all add: ln-chain-dominates)

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

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

lemma eval-primfun-nonzero: eventually ( $\lambda x. \text{eval-primfun } f\ x \neq 0$ ) at-top
  using eval-primfun-pos[of f] by eventually-elim simp

lemma eval-merge-primfun:
   $\text{fst } f = \text{fst } g \implies$ 
   $\text{eval-primfun } (\text{merge-primfun } f\ g)\ x = \text{eval-primfun } f\ x * \text{eval-primfun } g\ x$ 
  by (induction f g rule: merge-primfun.induct) (simp-all add: powr-add)

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

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

```

**definition** *eval-primfun*s where

*eval-primfun*s  $fs\ x = (\prod f \leftarrow fs. \text{eval-primfun } f\ x)$

**lemma** *eval-primfun*s-pos: eventually  $(\lambda x. \text{eval-primfuns  $fs\ x > 0)$  at-top$

**proof** –

**have** *prod-list*-pos:  $(\bigwedge x :: \text{linordered-semidom. } x \in \text{set } xs \implies x > 0) \implies \text{prod-list } xs > 0$

**for**  $xs :: \text{real list}$  **by** (*induction*  $xs$ ) *auto*

**have** eventually  $(\lambda x. \forall f \in \text{set } fs. \text{eval-primfun } f\ x > 0)$  at-top

**by** (*intro eventually-ball-finite ballI eval-primfun-pos finite-set*)

**thus** *?thesis unfolding eval-primfun*s-def **by** *eventually-elim (rule prod-list-pos, auto)*

**qed**

**lemma** *eval-primfun*s-nonzero: eventually  $(\lambda x. \text{eval-primfuns  $fs\ x \neq 0)$  at-top$

**using** *eval-primfun*s-pos[*of fs*] **by** *eventually-elim simp*

## 2.5 Reification

**definition** *LANDAU-PROD'* where

*LANDAU-PROD'*  $L\ c\ f = L(\lambda x. c * f\ x)$

**definition** *LANDAU-PROD* where

*LANDAU-PROD*  $L\ c1\ c2\ fs \longleftrightarrow (\lambda-. c1) \in L(\lambda x. c2 * \text{eval-primfuns  $fs\ x)$$

**definition** *BIGTHETA-CONST'* where *BIGTHETA-CONST'*  $c = \Theta(\lambda x. c)$

**definition** *BIGTHETA-CONST* where *BIGTHETA-CONST*  $c\ A = \text{set-mult } \Theta(\lambda-. c)\ A$

**definition** *BIGTHETA-FUN* where *BIGTHETA-FUN*  $f = \Theta(f)$

**lemma** *BIGTHETA-CONST'*-tag:  $\Theta(\lambda x. c) = \text{BIGTHETA-CONST}'\ c$  **using** *BIGTHETA-CONST'*-def ..

**lemma** *BIGTHETA-CONST*-tag:  $\Theta(f) = \text{BIGTHETA-CONST}\ 1\ \Theta(f)$

**by** (*simp add: BIGTHETA-CONST-def bigheta-mult-eq-set-mult[symmetric]*)

**lemma** *BIGTHETA-FUN*-tag:  $\Theta(f) = \text{BIGTHETA-FUN}\ f$

**by** (*simp add: BIGTHETA-FUN-def*)

**lemma** *set-mult-is-times*:  $\text{set-mult } A\ B = A * B$

**unfolding** *set-mult-def set-times-def func-times* **by** *blast*

**lemma** *set-powr-mult*:

**assumes** *eventually-nonneg*  $F\ f$  **and** *eventually-nonneg*  $F\ g$

**shows**  $\Theta[F](\lambda x. (f\ x * g\ x :: \text{real})\ \text{powr } p) = \text{set-mult } (\Theta[F](\lambda x. f\ x\ \text{powr } p))$   
 $(\Theta[F](\lambda x. g\ x\ \text{powr } p))$

**proof** –

**from** *assms* **have** eventually  $(\lambda x. f\ x \geq 0)$   $F$  eventually  $(\lambda x. g\ x \geq 0)$   $F$

**by** (*simp-all add: eventually-nonneg-def*)

**hence** eventually  $(\lambda x. (f\ x * g\ x :: \text{real})\ \text{powr } p = f\ x\ \text{powr } p * g\ x\ \text{powr } p)$   $F$

**by** *eventually-elim (simp add: powr-mult)*

**hence**  $\Theta[F](\lambda x. (f x * g x :: \text{real}) \text{ powr } p) = \Theta[F](\lambda x. f x \text{ powr } p * g x \text{ powr } p)$   
**by** (*rule landau-theta.cong*)  
**also have**  $\dots = \text{set-mult } (\Theta[F](\lambda x. f x \text{ powr } p)) (\Theta[F](\lambda x. g x \text{ powr } p))$   
**by** (*simp add: bigheta-mult-eq-set-mult*)  
**finally show** *?thesis* .  
**qed**

**lemma** *eventually-nonneg-bigheta-pow-realpow*:  
 $\Theta(\lambda x. \text{eval-primfun } f x \hat{=} e) = \Theta(\lambda x. \text{eval-primfun } f x \text{ powr } \text{real } e)$   
**using** *eval-primfun-pos[of f]*  
**by** (*auto intro!: landau-theta.cong elim!: eventually-mono simp: powr-realpow*)

**lemma** *BIGTHETA-CONST-fold*:  
 $\text{BIGTHETA-CONST } (c::\text{real}) (\text{BIGTHETA-CONST } d A) = \text{BIGTHETA-CONST } (c*d) A$   
 $\text{bigheta-pow at-top } (\text{BIGTHETA-CONST } c \Theta(\text{eval-primfun } pf)) k =$   
 $\text{BIGTHETA-CONST } (c \hat{=} k) \Theta(\lambda x. \text{eval-primfun } pf x \text{ powr } k)$   
 $\text{set-inverse } (\text{BIGTHETA-CONST } c \Theta(f)) = \text{BIGTHETA-CONST } (\text{inverse } c)$   
 $\Theta(\lambda x. \text{inverse } (f x))$   
 $\text{set-mult } (\text{BIGTHETA-CONST } c \Theta(f)) (\text{BIGTHETA-CONST } d \Theta(g)) =$   
 $\text{BIGTHETA-CONST } (c*d) \Theta(\lambda x. f x * g x)$   
 $\text{BIGTHETA-CONST}' (c::\text{real}) = \text{BIGTHETA-CONST } c \Theta(\lambda -. 1)$   
 $\text{BIGTHETA-FUN } (f::\text{real} \Rightarrow \text{real}) = \text{BIGTHETA-CONST } 1 \Theta(f)$   
**apply** (*simp add: BIGTHETA-CONST-def set-mult-is-times bigheta-mult-eq-set-mult mult-ac*)  
**apply** (*simp only: BIGTHETA-CONST-def bigheta-mult-eq-set-mult[symmetric] bigheta-pow-eq-set-pow[symmetric] power-mult-distrib mult-ac*)  
**apply** (*simp add: bigheta-mult-eq-set-mult eventually-nonneg-bigheta-pow-realpow*)  
**by** (*simp-all add: BIGTHETA-CONST-def BIGTHETA-CONST'-def BIGTHETA-FUN-def*  
 $\text{bigheta-mult-eq-set-mult[symmetric] set-mult-is-times[symmetric]$   
 $\text{bigheta-pow-eq-set-pow[symmetric] bigheta-inverse-eq-set-inverse[symmetric]$   
 $\text{mult-ac power-mult-distrib}$ )

**lemma** *fold-fun-chain*:  
 $g x = (g \hat{=} 1) x (g \hat{=} m) ((g \hat{=} n) x) = (g \hat{=} (m+n)) x$   
**by** (*simp-all add: funpow-add*)

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

**lemma** *reify-monom-1*:

$\Theta(\lambda x::\text{real}. x) = \Theta(\text{eval-primfun } (\text{LnChain } 0, 1))$

**proof** (*intro landau-theta.cong*)

**from** *eventually-gt-at-top*[of  $0::\text{real}$ ]

**show** *eventually*  $(\lambda x. x = \text{eval-primfun } (\text{LnChain } 0, 1) x)$  *at-top*

**by** *eventually-elim simp*

**qed**

**lemma** *reify-monom-pow*:

$\Theta(\lambda x::\text{real}. x \hat{=} e) = \Theta(\text{eval-primfun } (\text{LnChain } 0, \text{real } e))$

**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** *eventually*  $(\lambda x. x \text{ powr } (\text{real } e) = x \hat{=} e)$  *at-top*

**using** *eventually-gt-at-top*[of  $0$ ] **by** *eventually-elim (simp add: powr-realpow)*

**hence**  $\Theta(\lambda x. x \text{ powr } (\text{real } e)) = \Theta(\lambda x. x \hat{=} e)$

**by** (*rule landau-theta.cong*)

**finally show** *?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*:

$\Theta(\lambda x. (\text{ln } \hat{=} n) x \hat{=} e) = \Theta(\text{eval-primfun } (\text{LnChain } n, \text{real } e))$

**proof** –

**have**  $\Theta(\text{eval-primfun } (\text{LnChain } n, \text{real } e)) = \Theta(\lambda x. (\text{ln } \hat{=} n) x \text{ powr } (\text{real } e))$

**by** *simp*

**also have** *eventually*  $(\lambda x::\text{real}. (\text{ln } \hat{=} n) x > 0)$  *at-top*

**using** *fun-chain-at-top-at-top*[OF *ln-at-top*] **unfolding** *filterlim-at-top-dense* **by**

*blast*

**hence** *eventually*  $(\lambda x. (\text{ln } \hat{=} n) x \text{ powr } (\text{real } e) = (\text{ln } \hat{=} n) x \hat{=} e)$  *at-top*

**by** *eventually-elim (subst powr-realpow, auto)*

**hence**  $\Theta(\lambda x. (\text{ln } \hat{=} n) x \text{ powr } (\text{real } e)) = \Theta(\lambda x. (\text{ln } \hat{=} n) x \hat{=} e)$

**by** (*rule landau-theta.cong*)

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

**qed**

**lemma** *reify-ln-chain-powr*:

$\Theta(\lambda x. (\text{ln } \hat{=} n) x \text{ powr } e) = \Theta(\text{eval-primfun } (\text{LnChain } n, e))$

**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*:  $\text{numeral } n \hat{=} \text{Suc } a = \text{numeral } n * \text{numeral } n \hat{=} a$

**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*:  
*BIGTHETA-CONST e*  $\Theta(\lambda-. d) = \text{BIGTHETA-CONST } (e*d) \Theta(\text{eval-primfun}$   
 $\text{[]})$   
*LANDAU-PROD' c*  $(\lambda-. 1) = \text{LANDAU-PROD' } c (\text{eval-primfun}$   
 $\text{[]})$   
*eval-primfun f* = *eval-primfun* [f]  
*eval-primfun* fs x \* *eval-primfun* gs x = *eval-primfun* (fs @ gs) x  
**apply** (*simp only: BIGTHETA-CONST-def set-mult-is-times eval-primfun-def[abs-def]*  
*bigheta-mult-eq*)  
**apply** (*simp add: bigheta-mult-eq[symmetric]*)  
**by** (*simp-all add: eval-primfun-def[abs-def] BIGTHETA-CONST-def*)

**lemma** *inverse-prod-list-field*:  
*prod-list* (*map*  $(\lambda x. \text{inverse } (f x)) xs$ ) = *inverse* (*prod-list* (*map* f xs :: - :: *field*  
*list*))  
**by** (*induction xs simp-all*)

**lemma** *landau-prod-meta-cong*:  
**assumes** *landau-symbol L L' Lr*  
**assumes**  $\Theta(f) \equiv \text{BIGTHETA-CONST } c1 (\Theta(\text{eval-primfun } fs))$   
**assumes**  $\Theta(g) \equiv \text{BIGTHETA-CONST } c2 (\Theta(\text{eval-primfun } gs))$   
**shows**  $f \in L \text{ at-top } (g) \equiv \text{LANDAU-PROD } (L \text{ at-top}) c1 c2 (\text{map } \text{inverse-primfun}$   
 $fs @ gs)$   
**proof** –  
**interpret** *landau-symbol L L' Lr by fact*  
**have**  $f \in L \text{ at-top } (g) \longleftrightarrow (\lambda x. c1 * \text{eval-primfun } fs x) \in L \text{ at-top } (\lambda x. c2 * \text{eval-primfun } gs x)$   
**using** *assms(2,3)[symmetric] unfolding BIGTHETA-CONST-def*  
**by** (*intro cong-ex-bigheta*) (*simp-all add: bigheta-mult-eq-set-mult[symmetric]*)  
**also have** ...  $\longleftrightarrow (\lambda x. c1) \in L \text{ at-top } (\lambda x. c2 * \text{eval-primfun } gs x / \text{eval-primfun } fs x)$   
**by** (*simp-all add: eval-primfun-nonzero divide-eq1*)  
**finally show**  $f \in L \text{ at-top } (g) \equiv \text{LANDAU-PROD } (L \text{ at-top}) c1 c2 (\text{map } \text{inverse-primfun } fs @ gs)$   
**by** (*simp add: LANDAU-PROD-def eval-primfun-def eval-inverse-primfun*  
*divide-inverse o-def inverse-prod-list-field mult-ac*)

**qed**

**fun** *pos-primfun-list where*  
*pos-primfun-list []*  $\longleftrightarrow \text{False}$   
| *pos-primfun-list ((-,x)#xs)*  $\longleftrightarrow x > 0 \vee (x = 0 \wedge \text{pos-primfun-list } xs)$

```

fun nonneg-primfun-list where
  nonneg-primfun-list []  $\longleftrightarrow$  True
| nonneg-primfun-list ((-,x)#xs)  $\longleftrightarrow$   $x > 0 \vee (x = 0 \wedge \text{nonneg-primfun-list } xs)$ 

```

```

fun iszero-primfun-list where
  iszero-primfun-list []  $\longleftrightarrow$  True
| iszero-primfun-list ((-,x)#xs)  $\longleftrightarrow$   $x = 0 \wedge \text{iszero-primfun-list } xs$ 

```

**definition** *group-primfun*s  $\equiv$  *groupsort.group-sort fst merge-primfun*

```

lemma list-ConsCons-induct:
  assumes  $P [] \wedge x. P [x] \wedge x y xs. P (y\#xs) \implies P (x\#y\#xs)$ 
  shows  $P xs$ 
proof (induction xs rule: length-induct)
  case (1 xs)
  show ?case
  proof (cases xs)
  case (Cons x xs')
  note  $A = \text{this}$ 
  from assms 1 show ?thesis
  proof (cases xs')
  case (Cons y xs'')
  with 1 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-primfun:
  assumes sorted (map fst fs)
  assumes distinct (map fst fs)
  shows landau-function-family-chain at-top fs (eval-primfun' o fst)
proof (standard, goal-cases)
  case 3
  from assms show ?case
  proof (induction fs rule: list-ConsCons-induct)
  case (2 g)
  from eval-primfun'-at-top[of fst g]
  have  $\text{eval-primfun}' (fst g) \in \omega(\lambda-. 1)$ 
  by (intro smallomegaI-filterlim-at-infinity filterlim-at-top-imp-at-infinity) simp
  thus ?case by (simp add: smallomega-iff-smallo)
  next
  case (3 f g gs)
  thus ?case by (auto simp: primfun-dominates)
  qed simp
qed (auto simp: eval-primfun'-at-top)

```

**lemma** (in monoid-mult) fold-plus-prod-list-rev:  
*fold times xs = times (prod-list (rev xs))*

**proof**  
**fix**  $x$   
**have** *fold times xs x = prod-list (rev xs @ [x])*  
**by** (*simp add: foldr-conv-fold prod-list.eq-foldr*)  
**also have**  $\dots = \text{prod-list (rev xs) * x}$   
**by** *simp*  
**finally show** *fold times xs x = prod-list (rev xs) \* x .*  
**qed**

**interpretation** groupsort-primfun: groupsort fst merge-primfun eval-primfuns  
**proof** (*standard, goal-cases*)  
**case** ( $1\ x\ y$ )  
**thus** *?case* **by** (*induction x y rule: merge-primfun.induct*) *simp-all*  
**next**  
**case** ( $2\ fs\ gs$ )  
**show** *?case*  
**proof**  
**fix**  $x$   
**have** *eval-primfuns fs x = fold (\*) (map ( $\lambda f$ . eval-primfun f x) fs) 1*  
**unfolding** *eval-primfuns-def* **by** (*simp add: fold-plus-prod-list-rev*)  
**also have** *fold (\*) (map ( $\lambda f$ . eval-primfun f x) fs) = fold (\*) (map ( $\lambda f$ .  
*eval-primfun f x) gs)*  
**using**  $2$  **by** (*intro fold-multiset-equiv ext*) *auto*  
**also have**  $\dots 1 = \text{eval-primfuns gs } x$   
**unfolding** *eval-primfuns-def* **by** (*simp add: fold-plus-prod-list-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 ((=) 0) (map snd fs)*  
**by** (*induction fs rule: iszero-primfun-list.induct*) *simp-all*

**lemma** *landau-primfuns-iff*:  
 $((\lambda-. 1) \in O(\text{eval-primfuns } fs)) = \text{nonneg-primfun-list (group-primfuns } fs)$  (**is** *?A*)  
 $((\lambda-. 1) \in o(\text{eval-primfuns } fs)) = \text{pos-primfun-list (group-primfuns } fs)$  (**is** *?B*)  
 $((\lambda-. 1) \in \Theta(\text{eval-primfuns } fs)) = \text{iszero-primfun-list (group-primfuns } fs)$  (**is** *?C*)  
**proof** –  
**interpret** *landau-function-family-chain at-top group-primfuns fs snd eval-primfun'*

$o$  *fst*  
**by** (*rule landau-function-family-chain-primfun*)  
(*simp-all add: group-primfun-def groupsort-primfun.sorted-group-sort*  
*groupsort-primfun.distinct-group-sort*)

**have**  $(\lambda-. 1) \in O(\text{eval-primfun } fs) \longleftrightarrow (\lambda-. 1) \in O(\text{eval-primfun } (\text{group-primfun } fs))$   
**by** (*simp-all add: groupsort-primfun.g-group-sort group-primfun-def*)  
**also have**  $\dots \longleftrightarrow \text{nonneg-list } (\text{map snd } (\text{group-primfun } fs))$  **using** *bigo-iff*  
**by** (*simp add: eval-primfun-def[abs-def] eval-primfun-altdef*)  
**finally show**  $?A$  **by** (*simp add: nonneg-primfun-list-iff*)

**have**  $(\lambda-. 1) \in o(\text{eval-primfun } fs) \longleftrightarrow (\lambda-. 1) \in o(\text{eval-primfun } (\text{group-primfun } fs))$   
**by** (*simp-all add: groupsort-primfun.g-group-sort group-primfun-def*)  
**also have**  $\dots \longleftrightarrow \text{pos-list } (\text{map snd } (\text{group-primfun } fs))$  **using** *smallo-iff*  
**by** (*simp add: eval-primfun-def[abs-def] eval-primfun-altdef*)  
**finally show**  $?B$  **by** (*simp add: pos-primfun-list-iff*)

**have**  $(\lambda-. 1) \in \Theta(\text{eval-primfun } fs) \longleftrightarrow (\lambda-. 1) \in \Theta(\text{eval-primfun } (\text{group-primfun } fs))$   
**by** (*simp-all add: groupsort-primfun.g-group-sort group-primfun-def*)  
**also have**  $\dots \longleftrightarrow \text{list-all } ((=) 0) (\text{map snd } (\text{group-primfun } fs))$  **using** *bigtheta-iff*  
**by** (*simp add: eval-primfun-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 at-top) c1 c2 fs*  $\longleftrightarrow c1 = 0 \vee (c2 \neq 0 \wedge \text{nonneg-primfun-list } (\text{group-primfun } fs))$   
**unfolding** *LANDAU-PROD-def*  
**by** (*cases c1 = 0, simp, cases c2 = 0, simp*) (*simp-all add: landau-primfun-iff*)

**lemma** *LANDAU-PROD-smallo-iff*:  
*LANDAU-PROD (smallo at-top) c1 c2 fs*  $\longleftrightarrow c1 = 0 \vee (c2 \neq 0 \wedge \text{pos-primfun-list } (\text{group-primfun } fs))$   
**unfolding** *LANDAU-PROD-def*  
**by** (*cases c1 = 0, simp, cases c2 = 0, simp*) (*simp-all add: landau-primfun-iff*)

**lemma** *LANDAU-PROD-bigtheta-iff*:  
*LANDAU-PROD (bigtheta at-top) c1 c2 fs*  $\longleftrightarrow (c1 = 0 \wedge c2 = 0) \vee (c1 \neq 0 \wedge c2 \neq 0 \wedge \text{iszero-primfun-list } (\text{group-primfun } fs))$

**proof** –

**have**  $A: \bigwedge P x. (x = 0 \implies P) \implies (x \neq 0 \implies P) \implies P$  **by** *blast*

{

**assume** *eventually*  $(\lambda x. \text{eval-primfun } fs x = 0)$  *at-top*

**with** *eval-primfun-nonzero*[of *fs*] **have** *eventually*  $(\lambda x::\text{real}. \text{False})$  *at-top*

```

    by eventually-elim simp
  hence False by simp
} note B = this
show ?thesis by (rule A[of c1, case-product A[of c2]])
                (insert B, auto simp: LANDAU-PROD-def landau-primfun-iff)
qed

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

```

```

lemmas landau-real-prod-simps [simp] =
  group-sort-primfun.group-part-def
  group-primfun-def group-sort-primfun.group-sort.simps
  group-sort-primfun.group-part-aux.simps pos-primfun-list.simps
  nonneg-primfun-list.simps iszero-primfun-list.simps

```

end

### 3 Simplification procedures

```

theory Landau-Simprocs
imports Landau-Real-Products
begin

```

#### 3.1 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 <
  let
    val eq-thms = @{thms landau-theta.cong-bigtheta}
    fun eq-rule thm = get-first (try (fn eq-thm => eq-thm OF [thm])) eq-thms
  in
    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
>

```

```

lemma bigtheta-const [landau-simp]:

```

*NO-MATCH*  $1\ c \implies c \neq 0 \implies (\lambda x. c) \in \Theta(\lambda x. 1)$  **by** *simp*

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

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

$0 < a \implies (\lambda x::\text{real}. \ln (x * a)) \in \Theta(\ln)$

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

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

$0 < a \implies (\lambda x::\text{real}. \ln (x * a) \text{ powr } p) \in \Theta(\lambda x. \ln x \text{ powr } p)$

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

**lemma** *bigheta-const-ln-pow'* [*landau-simp*]:

$0 < a \implies (\lambda x::\text{real}. \ln (x * a) \hat{\ } p) \in \Theta(\lambda x. \ln x \hat{\ } p)$

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

### 3.2 Simproc setup

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

*landau-symbol*  $L\ L'\ Lr \implies (\bigwedge x::\text{real}. x > 1 \implies f\ x = g\ x) \implies L\ \text{at-top}\ (f) = L\ \text{at-top}\ (g)$

**by** (*auto intro: eventually-mono [OF eventually-gt-at-top[of 1]] elim!: landau-symbol.cong*)

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

*landau-symbol*  $L\ L'\ Lr \implies (\bigwedge x::\text{real}. x > 1 \implies f\ x = g\ x) \implies f \in L\ \text{at-top}\ (h) \longleftrightarrow g \in L\ \text{at-top}\ (h)$

**by** (*auto intro: eventually-mono [OF eventually-gt-at-top[of 1]] elim!: landau-symbol.in-cong*)

**lemma** *landau-prop-equalsI*:

*landau-symbol*  $L\ L'\ Lr \implies (\bigwedge x::\text{real}. x > 1 \implies f1\ x = f2\ x) \implies (\bigwedge x. x > 1 \implies g1\ x = g2\ x) \implies$

$f1 \in L\ \text{at-top}\ (g1) \longleftrightarrow f2 \in L\ \text{at-top}\ (g2)$

**apply** (*subst landau-gt-1-cong, assumption+*)

**apply** (*subst landau-gt-1-in-cong, assumption+*)

**apply** (*rule refl*)

**done**

**lemma** *ab-diff-conv-add-uminus'*:  $(a:::\text{ab-group-add}) - b = -b + a$  **by** *simp*

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

**lemma** *divide-inverse'*:  $(a:::\{\text{division-ring, ab-semigroup-mult}\}) / b = \text{inverse } b * a$

**by** (*simp add: divide-inverse mult.commute*)

**lemma** *extract-divide-middle*:  $(a:::\{\text{field}\}) / (x * b) = \text{inverse } x * (a / b)$

**by** (*simp add: divide-inverse algebra-simps*)

**lemmas** *landau-cancel* = *landau-symbol.mult-cancel-left*

**lemmas** *mult-cancel-left'* = *landau-symbol.mult-cancel-left*[*OF - bitheta-refl eventually-nonzeroD*]

**lemma** *mult-cancel-left-1*:

**assumes** *landau-symbol L L' Lr eventually-nonzero F f*

**shows**  $f \in L F (\lambda x. f x * g2 x) \longleftrightarrow (\lambda-. 1) \in L F (g2)$

$(\lambda x. f x * f2 x) \in L F (f) \longleftrightarrow f2 \in L F (\lambda-. 1)$

$f \in L F (f) \longleftrightarrow (\lambda-. 1) \in L F (\lambda-. 1)$

**using** *mult-cancel-left'*[*OF assms, of λ-. 1*] *mult-cancel-left'*[*OF assms, of - λ-. 1*]

*mult-cancel-left'*[*OF assms, of λ-. 1 λ-. 1*] **by** *simp-all*

**lemmas** *landau-mult-cancel-simps* = *mult-cancel-left'* *mult-cancel-left-1*

**ML-file** *⟨landau-simprocs.ML⟩*

**lemmas** *bitheta-simps* =

*landau-theta.cong-bitheta*[*OF bitheta-const-ln*]

*landau-theta.cong-bitheta*[*OF bitheta-const-ln-powr*]

The following simproc attempts to cancel common factors in Landau symbols, i. e. in a goal like  $f(x)h(x) \in L(g(x)h(x))$ , the common factor  $h(x)$  will be cancelled. This only works if the simproc can prove that  $h(x)$  is eventually non-zero, for which it uses some heuristics.

**simproc-setup** *landau-cancel-factor* (

$f \in o[F](g) \mid f \in O[F](g) \mid f \in \omega[F](g) \mid f \in \Omega[F](g) \mid f \in \Theta[F](g)$

) = *⟨K Landau.cancel-factor-simproc⟩*

The next simproc attempts to cancel dominated summands from Landau symbols; e. g.  $O(x + \ln x)$  is simplified to  $O(x)$ , since  $\ln x \in o(x)$ . This can be very slow on large terms, so it is not enabled by default.

**simproc-setup** *simplify-landau-sum* (

$o[F](\lambda x. f x) \mid O[F](\lambda x. f x) \mid \omega[F](\lambda x. f x) \mid \Omega[F](\lambda x. f x) \mid \Theta[F](\lambda x. f x) \mid$

$f \in o[F](g) \mid f \in O[F](g) \mid f \in \omega[F](g) \mid f \in \Omega[F](g) \mid f \in \Theta[F](g)$

) = *⟨K (Landau.lift-landau-simproc Landau.simplify-landau-sum-simproc)⟩*

This simproc attempts to simplify factors of an expression in a Landau symbol statement independently from another, i. e. in something like  $O(f(x)g(x))$ , a simp rule that rewrites  $O(f(x))$  to  $O(f'(x))$  will also rewrite  $O(f(x)g(x))$  to  $O(f'(x)g(x))$  without any further setup.

**simproc-setup** *simplify-landau-product* (

$o[F](\lambda x. f x) \mid O[F](\lambda x. f x) \mid \omega[F](\lambda x. f x) \mid \Omega[F](\lambda x. f x) \mid \Theta[F](\lambda x. f x) \mid$

$f \in o[F](g) \mid f \in O[F](g) \mid f \in \omega[F](g) \mid f \in \Omega[F](g) \mid f \in \Theta[F](g)$

) = *⟨K (Landau.lift-landau-simproc Landau.simplify-landau-product-simproc)⟩*

Lastly, the next very specialised simproc can solve goals of the form  $f(x) \in L(g(x))$  where  $f$  and  $g$  are real-valued functions consisting only of multipli-

cations, powers of  $x$ , and powers of iterated logarithms of  $x$ . This is done by rewriting both sides into the form  $x^a(\ln x)^b(\ln \ln x)^c$  etc. and then comparing the exponents lexicographically.

Note that for historic reasons, this only works for  $x \rightarrow \infty$ .

```
simproc-setup landau-real-prod (
  (f :: real => real) ∈ o(g) | (f :: real => real) ∈ O(g) |
  (f :: real => real) ∈ ω(g) | (f :: real => real) ∈ Ω(g) |
  (f :: real => real) ∈ Θ(g)
) = <K Landau.simplify-landau-real-prod-prop-simproc>
```

### 3.3 Tests

```
lemma asymp-equiv-plus-const-left: (λn. c + real n) ~[at-top] (λn. real n)
by (subst asymp-equiv-add-left) (auto intro!: asymp-equiv-intros eventually-gt-at-top)
```

```
lemma asymp-equiv-plus-const-right: (λn. real n + c) ~[at-top] (λn. real n)
using asymp-equiv-plus-const-left[of c] by (simp add: add.commute)
```

#### 3.3.1 Product simplification tests

```
lemma (λx::real. f x * x) ∈ O(λx. g x / (h x / x)) ↔ f ∈ O(λx. g x / h x)
by simp
```

```
lemma (λx::real. x) ∈ ω(λx. g x / (h x / x)) ↔ (λx. 1) ∈ ω(λx. g x / h x)
by simp
```

#### 3.3.2 Real product decision procure tests

```
lemma (λx. x powr 1) ∈ O(λx. x powr 2 :: real)
by simp
```

```
lemma Θ(λx::real. 2*x powr 3 - 4*x powr 2) = Θ(λx::real. x powr 3)
by (simp add: landau-theta.absorb)
```

```
lemma p < q ⇒ (λx::real. c * x powr p * ln x powr r) ∈ o(λx::real. x powr q)
by simp
```

```
lemma c ≠ 0 ⇒ p > q ⇒ (λx::real. c * x powr p * ln x powr r) ∈ ω(λx::real.
x powr q)
by simp
```

```
lemma b > 0 ⇒ (λx::real. x / ln (2*b*x) * 2) ∈ o(λx. x * ln (b*x))
by simp
```

```
lemma o(λx::real. x * ln (3*x)) = o(λx. ln x * x)
by (simp add: mult.commute)
```

```
lemma (λx::real. x) ∈ o(λx. x * ln (3*x)) by simp
```

**ML-val** <

```
Landau.simplify-landau-real-prod-prop-conv @{context}
```

@{cterm ( $\lambda x::real. 5 * \ln (\ln x) ^ 2 / (2*x) \text{ powr } 1.5 * \text{ inverse } 2$ )  $\in$   
 $\omega(\lambda x. 3 * \ln x * \ln x / x * \ln (\ln (\ln (ln x))))$ )}  
 >

**lemma** ( $\lambda x. 3 * \ln x * \ln x / x * \ln (\ln (\ln (ln x)))$ )  $\in$   
 $\omega(\lambda x::real. 5 * \ln (\ln x) ^ 2 / (2*x) \text{ powr } 1.5 * \text{ inverse } 2)$   
**by** *simp*

### 3.3.3 Sum cancelling tests

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

**lemma**  $\Theta(\lambda x::real. 2 * x \text{ powr } 3 + x * x^2 / \ln x + 42 * x \text{ powr } 9 + 213 * x \text{ powr } 5 - 4 * x \text{ powr } 7) =$   
 $\Theta(\lambda x::real. x ^ 3 + x / \ln x * x \text{ powr } (3/2) - 2*x \text{ powr } 9)$   
**using**  $[[\text{landau-sum-limit} = 5]]$  **by** *simp*

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

**end**

**theory** *Landau-More*

**imports**

*HOL-Library.Landau-Symbols*

*Landau-Simprocs*

**begin**

**lemma** *bigo-const-inverse* [*simp*]:

**assumes** *filterlim f at-top F F  $\neq$  bot*

**shows**  $(\lambda-. c) \in O[F](\lambda x. \text{ inverse } (f x) :: real) \longleftrightarrow c = 0$

**proof** –

{

**assume** *A*:  $(\lambda-. 1) \in O[F](\lambda x. \text{ inverse } (f x))$

**from** *assms* **have**  $(\lambda-. 1) \in o[F](f)$

**by** (*simp add: eventually-nonzero-simps smallomega-iff-smallo filterlim-at-top-iff-smallomega*)

**also from** *assms A* **have**  $f \in O[F](\lambda-. 1)$

**by** (*simp add: eventually-nonzero-simps landau-divide-simps*)

**finally have** *False* **using** *assms* **by** (*simp add: landau-o.small-refl-iff*)

}

**thus** *?thesis* **by** (*cases c = 0*) *auto*

**qed**

**lemma** *smallo-const-inverse* [*simp*]:

*filterlim f at-top F  $\implies F \neq bot \implies (\lambda-. c :: real) \in o[F](\lambda x. \text{ inverse } (f x)) \longleftrightarrow c = 0$*

**by** (*auto dest: landau-o.small-imp-big*)

**lemma** *const-in-smallo-const* [*simp*]:  $(\lambda-. b) \in o(\lambda-. - :: \text{linorder}. c) \longleftrightarrow b = 0$   
(**is** *?lhs*  $\longleftrightarrow$  *?rhs*)

**by** (*cases b = 0; cases c = 0*) (*simp-all add: landau-o.small-refl-iff*)

**lemma** *smallomega-1-conv-filterlim*:  $f \in \omega[F](\lambda-. 1) \longleftrightarrow \text{filterlim } f \text{ at-infinity } F$   
**by** (*auto intro: smallomegaI-filterlim-at-infinity dest: smallomegaD-filterlim-at-infinity*)

**lemma** *bigheta-powr-1* [*landau-simp*]:  
*eventually*  $(\lambda x. (f x :: \text{real}) \geq 0) F \implies (\lambda x. f x \text{ powr } 1) \in \Theta[F](f)$   
**by** (*intro bighetaI-cong*) (*auto elim!: eventually-mono*)

**lemma** *bigheta-powr-0* [*landau-simp*]:  
*eventually*  $(\lambda x. (f x :: \text{real}) \neq 0) F \implies (\lambda x. f x \text{ powr } 0) \in \Theta[F](\lambda-. 1)$   
**by** (*intro bighetaI-cong*) (*auto elim!: eventually-mono*)

**lemma** *bigheta-powr-nonzero* [*landau-simp*]:  
*eventually*  $(\lambda x. (f x :: \text{real}) \neq 0) F \implies (\lambda x. \text{if } f x = 0 \text{ then } g x \text{ else } h x) \in \Theta[F](h)$   
**by** (*intro bighetaI-cong*) (*auto elim!: eventually-mono*)

**lemma** *bigheta-powr-nonzero'* [*landau-simp*]:  
*eventually*  $(\lambda x. (f x :: \text{real}) \neq 0) F \implies (\lambda x. \text{if } f x \neq 0 \text{ then } g x \text{ else } h x) \in \Theta[F](g)$   
**by** (*intro bighetaI-cong*) (*auto elim!: eventually-mono*)

**lemma** *bigheta-powr-nonneg* [*landau-simp*]:  
*eventually*  $(\lambda x. (f x :: \text{real}) \geq 0) F \implies (\lambda x. \text{if } f x \geq 0 \text{ then } g x \text{ else } h x) \in \Theta[F](g)$   
**by** (*intro bighetaI-cong*) (*auto elim!: eventually-mono*)

**lemma** *bigheta-powr-nonneg'* [*landau-simp*]:  
*eventually*  $(\lambda x. (f x :: \text{real}) \geq 0) F \implies (\lambda x. \text{if } f x < 0 \text{ then } g x \text{ else } h x) \in \Theta[F](h)$   
**by** (*intro bighetaI-cong*) (*auto elim!: eventually-mono*)

**lemma** *bigo-powr-iff*:

**assumes**  $0 < p$  *eventually*  $(\lambda x. f x \geq 0) F$  *eventually*  $(\lambda x. g x \geq 0) F$

**shows**  $(\lambda x. (f x :: \text{real}) \text{ powr } p) \in O[F](\lambda x. g x \text{ powr } p) \longleftrightarrow f \in O[F](g)$  (**is** *?lhs*  $\longleftrightarrow$  *?rhs*)

**proof**

**assume** *?lhs*

**with** *assms bigo-powr[OF this, of inverse p]* **show** *?rhs*

**by** (*simp add: powr-powr landau-simps*)

**qed** (*insert assms, simp-all add: bigo-powr-nonneg*)

**lemma** *inverse-powr* [*simp*]:  
**assumes**  $(x :: \text{real}) \geq 0$

**shows**  $\text{inverse } x \text{ powr } y = \text{inverse } (x \text{ powr } y)$   
**proof** (*cases*  $x > 0$ )  
**assume**  $x: x > 0$   
**from**  $x$  **have**  $\text{inverse } x \text{ powr } y = \exp (y * \ln (\text{inverse } x))$  **by** (*simp add: powr-def*)  
**also have**  $\ln (\text{inverse } x) = -\ln x$  **by** (*simp add: x ln-inverse*)  
**also have**  $\exp (y * -\ln x) = \text{inverse } (\exp (y * \ln x))$  **by** (*simp add: exp-minus*)  
**also from**  $x$  **have**  $\exp (y * \ln x) = x \text{ powr } y$  **by** (*simp add: powr-def*)  
**finally show** *?thesis* .  
**qed** (*insert assms, simp*)

**lemma** *big-neg-powr-iff*:

**assumes**  $p < 0$  *eventually*  $(\lambda x. f x \geq 0)$   $F$  *eventually*  $(\lambda x. g x \geq 0)$   $F$   
*eventually*  $(\lambda x. f x \neq 0)$   $F$  *eventually*  $(\lambda x. g x \neq 0)$   $F$   
**shows**  $(\lambda x. (f x :: \text{real}) \text{ powr } p) \in O[F](\lambda x. g x \text{ powr } p) \longleftrightarrow g \in O[F](f)$  (**is** *?lhs*  
 $\longleftrightarrow$  *?rhs*)  
**proof** –  
**have**  $(\lambda x. f x \text{ powr } p) \in O[F](\lambda x. g x \text{ powr } p) \longleftrightarrow$   
 $(\lambda x. (\text{inverse } (f x)) \text{ powr } -p) \in O[F](\lambda x. (\text{inverse } (g x)) \text{ powr } -p)$   
**using** *assms* **by** (*intro landau-o.big.cong-ex*) (*auto simp: powr-minus elim: eventually-mono*)  
**also from** *assms* **have**  $\dots \longleftrightarrow ((\lambda x. \text{inverse } (f x)) \in O[F](\lambda x. \text{inverse } (g x)))$   
**by** (*subst bigo-powr-iff*) *simp-all*  
**also from** *assms* **have**  $\dots \longleftrightarrow g \in O[F](f)$  **by** (*simp add: landau-o.big.inverse-cancel*)  
**finally show** *?thesis* .  
**qed**

**lemma** *smallo-powr-iff*:

**assumes**  $0 < p$  *eventually*  $(\lambda x. f x \geq 0)$   $F$  *eventually*  $(\lambda x. g x \geq 0)$   $F$   
**shows**  $(\lambda x. (f x :: \text{real}) \text{ powr } p) \in o[F](\lambda x. g x \text{ powr } p) \longleftrightarrow f \in o[F](g)$  (**is** *?lhs*  
 $\longleftrightarrow$  *?rhs*)  
**proof**  
**assume** *?lhs*  
**with** *assms* *smallo-powr*[*OF this, of inverse p*] **show** *?rhs*  
**by** (*simp add: powr-powr landau-simps*)  
**qed** (*insert assms, simp-all add: smallo-powr-nonneg*)

**lemma** *smallo-neg-powr-iff*:

**assumes**  $p < 0$  *eventually*  $(\lambda x. f x \geq 0)$   $F$  *eventually*  $(\lambda x. g x \geq 0)$   $F$   
*eventually*  $(\lambda x. f x \neq 0)$   $F$  *eventually*  $(\lambda x. g x \neq 0)$   $F$   
**shows**  $(\lambda x. (f x :: \text{real}) \text{ powr } p) \in o[F](\lambda x. g x \text{ powr } p) \longleftrightarrow g \in o[F](f)$  (**is** *?lhs*  
 $\longleftrightarrow$  *?rhs*)  
**proof** –  
**have**  $(\lambda x. f x \text{ powr } p) \in o[F](\lambda x. g x \text{ powr } p) \longleftrightarrow$   
 $(\lambda x. (\text{inverse } (f x)) \text{ powr } -p) \in o[F](\lambda x. (\text{inverse } (g x)) \text{ powr } -p)$   
**using** *assms* **by** (*intro landau-o.small.cong-ex*) (*auto simp: powr-minus elim: eventually-mono*)  
**also from** *assms* **have**  $\dots \longleftrightarrow ((\lambda x. \text{inverse } (f x)) \in o[F](\lambda x. \text{inverse } (g x)))$   
**by** (*subst smallo-powr-iff*) *simp-all*  
**also from** *assms* **have**  $\dots \longleftrightarrow g \in o[F](f)$  **by** (*simp add: landau-o.small.inverse-cancel*)

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

**lemma** *const-smallo-powr*:  
**assumes** *filterlim f at-top F F ≠ bot*  
**shows**  $(\lambda-. c :: \text{real}) \in o[F](\lambda x. f x \text{ powr } p) \longleftrightarrow p > 0 \vee c = 0$   
**by** (*rule linorder-cases*[of  $p \ 0$ ]; *cases*  $c = 0$ )  
*(insert assms smallo-powr-iff*[of  $p \ \lambda-. \ 1 \ F \ f$ ] *smallo-neg-powr-iff*[of  $p \ f \ F \ \lambda-. \ 1$ ],  
*auto simp: landau-simps eventually-nonzero-simps smallo-1-iff*[of  $F \ f$ ] *not-less*  
*dest: landau-o.small-asymmetric simp: eventually-False landau-o.small-refl-iff)*

**lemma** *bigconst-powr*:  
**assumes** *filterlim f at-top F F ≠ bot*  
**shows**  $(\lambda-. c :: \text{real}) \in O[F](\lambda x. f x \text{ powr } p) \longleftrightarrow p \geq 0 \vee c = 0$   
**proof** –  
**from** *assms* **have**  $A: (\lambda-. \ 1) \in o[F](f)$   
**by** (*simp add: filterlim-at-top-iff-smallomega smallomega-iff-smallo landau-o.small-imp-big*)  
**hence**  $B: (\lambda-. \ 1) \in O[F](f) \ f \notin O[F](\lambda-. \ 1)$  **using** *assms*  
**by** (*auto simp: landau-o.small-imp-big dest: landau-o.small-big-asymmetric*)  
**show** *?thesis*  
**by** (*rule linorder-cases*[of  $p \ 0$ ]; *cases*  $c = 0$ )  
*(insert insert assms A B bigconst-powr-iff*[of  $p \ \lambda-. \ 1 \ F \ f$ ] *bigconst-neg-powr-iff*[of  $p \ \lambda-. \ 1 \ F \ f$ ],  
*auto simp: landau-simps eventually-nonzero-simps not-less dest: landau-o.small-asymmetric)*  
**qed**

**lemma** *filterlim-powr-at-top*:  
 $(b :: \text{real}) > 1 \implies \text{filterlim } (\lambda x. b \text{ powr } x) \text{ at-top at-top}$   
**unfolding** *powr-def mult.commute*[of  $- \ \ln \ b$ ]  
**by** (*auto intro!: filterlim-compose*[ $OF \ \text{exp-at-top}$ ]  
*filterlim-tendsto-pos-mult-at-top filterlim-ident*)

**lemma** *power-smallo-exponential*:  
**fixes**  $b :: \text{real}$   
**assumes**  $b: b > 1$   
**shows**  $(\lambda x. x \text{ powr } n) \in o(\lambda x. b \text{ powr } x)$   
**proof** (*rule smalloI-tendsto*)  
**from** *assms* **have** *filterlim*  $(\lambda x. x * \ln b - n * \ln x) \text{ at-top at-top}$   
**using** [*simproc add: simplify-landau-sum*]  
**by** (*simp add: filterlim-at-top-iff-smallomega eventually-nonzero-simps*)  
**hence**  $((\lambda x. \text{exp } (-(x * \ln b - n * \ln x))) \longrightarrow 0) \text{ at-top (is } ?A)$   
**by** (*intro filterlim-compose*[ $OF \ \text{exp-at-bot}$ ]  
*filterlim-compose*[ $OF \ \text{filterlim-uminus-at-bot-at-top}$ ])  
**also have**  $?A \longleftrightarrow ((\lambda x. x \text{ powr } n / b \text{ powr } x) \longrightarrow 0) \text{ at-top}$   
**using** *b eventually-gt-at-top*[of  $0$ ]  
**by** (*intro tendsto-cong*)  
*(auto simp: exp-diff powr-def field-simps exp-of-nat-mult elim: eventually-mono)*

**finally show**  $((\lambda x. x \text{ powr } n / b \text{ powr } x) \longrightarrow 0) \text{ at-top}$  .  
**qed** (*insert assms, simp-all add: eventually-nonzero-simps*)

**lemma** *powr-fast-growth-tendsto*:

**assumes**  $gf: g \in O[F](f)$   
**and**  $n: n \geq 0$   
**and**  $k: k > 1$   
**and**  $f: \text{filterlim } f \text{ at-top } F$   
**and**  $g: \text{eventually } (\lambda x. g \ x \geq 0) \ F$   
**shows**  $(\lambda x. g \ x \text{ powr } n) \in o[F](\lambda x. k \text{ powr } f \ x :: \text{real})$   
**proof** –  
**from**  $f$  **have**  $f': \text{eventually } (\lambda x. f \ x \geq 0) \ F$  **by** (*simp add: eventually-nonzero-simps*)  
**from**  $gf$  **obtain**  $c$  **where**  $c: c > 0$  **eventually**  $(\lambda x. \text{norm } (g \ x) \leq c * \text{norm } (f \ x))$   
 $F$   
**by** (*elim landau-o.bigE*)  
**from**  $c(2)$   $g \ f'$  **have** **eventually**  $(\lambda x. g \ x \leq c * f \ x) \ F$  **by** *eventually-elim simp*  
**from**  $c(2)$   $g \ f'$  **have** **eventually**  $(\lambda x. \text{norm } (g \ x \text{ powr } n) \leq \text{norm } (c \text{ powr } n * f \ x \text{ powr } n)) \ F$   
**by** *eventually-elim (insert n c(1), auto simp: powr-mult [symmetric] intro!: powr-mono2)*  
**from** *landau-o.big-mono[OF this] c(1)*  
**have**  $(\lambda x. g \ x \text{ powr } n) \in O[F](\lambda x. f \ x \text{ powr } n)$  **by** *simp*  
**also from** *power-smallo-exponential f*  
**have**  $(\lambda x. f \ x \text{ powr } n) \in o[F](\lambda x. k \text{ powr } f \ x)$  **by** (*rule landau-o.small.compose*)  
 $\text{fact+}$   
**finally show** *?thesis* .  
**qed**

**lemma** *bigO-abs-powr-iff [simp]*:

$0 < p \implies (\lambda x. |f \ x| \text{ powr } p) \in O[F](\lambda x. |g \ x| \text{ powr } p) \iff f \in O[F](g)$   
**by**(*subst bigO-powr-iff; simp*)

**lemma** *smallo-abs-powr-iff [simp]*:

$0 < p \implies (\lambda x. |f \ x| \text{ powr } p) \in o[F](\lambda x. |g \ x| \text{ powr } p) \iff f \in o[F](g)$   
**by**(*subst smallo-powr-iff; simp*)

**lemma** *const-smallo-inverse-powr*:

**assumes** *filterlim f at-top at-top*  
**shows**  $(\lambda - :: - :: \text{linorder}. c :: \text{real}) \in o(\lambda x. \text{inverse } (f \ x \text{ powr } p)) \iff (p \geq 0 \implies c = 0)$   
**proof**(*cases p 0 :: real rule: linorder-cases*)  
**case**  $p: \text{greater}$   
**have**  $(\lambda -. c) \in o(\lambda x. \text{inverse } (f \ x \text{ powr } p)) \iff (\lambda -. |c|) \in o(\lambda x. \text{inverse } (f \ x \text{ powr } p))$  **by** *simp*  
**also have**  $|c| = (|c| \text{ powr } (\text{inverse } p)) \text{ powr } p$  **using**  $p$  **by**(*simp add: powr-powr*)  
**also** **{** **have** **eventually**  $(\lambda x. f \ x \geq 0) \text{ at-top}$  **using** *assms* **by**(*simp add: filterlim-at-top*)  
**then have**  $o(\lambda x. \text{inverse } (f \ x \text{ powr } p)) = o(\lambda x. |\text{inverse } (f \ x)| \text{ powr } p)$

```

    by(intro landau-o.small.cong)(auto elim!: eventually-rev-mp)
    also have (λ-. |(|c| powr inverse p)| powr p) ∈ ... ↔ (λ-. |c| powr (inverse
p)) ∈ o(λx. inverse (f x))
    using p by(rule smallo-abs-powr-iff)
    also note calculation }
    also have (λ-. |c| powr (inverse p)) ∈ o(λx. inverse (f x)) ↔ c = 0 using
assms by simp
    finally show ?thesis using p by simp
next
  case equal
    from assms have eventually (λx. f x ≥ 1) at-top using assms by(simp add:
filterlim-at-top)
    then have o(λx. inverse (f x powr p)) = o(λx. 1)
      by(intro landau-o.small.cong)(auto simp add: equal elim!: eventually-rev-mp)
    then show ?thesis using equal by simp
next
  case less
    from assms have nonneg: ∀F x in at-top. 0 ≤ f x by(simp add: filterlim-at-top)
    with assms have ∀F x in at-top. ||c| powr (1 / - p)| / d ≤ |f x| (is ∀F x in -.
?c ≤ -) if d > 0 for d
      by(fastforce dest!: spec[where x=?c] simp add: filterlim-at-top elim: eventu-
ally-rev-mp)
    then have (λ-. |c| powr (1 / - p)) ∈ o(f) by(intro landau-o.smallI)(simp add:
field-simps)
    then have (λ-. ||c| powr (1 / - p)| powr - p) ∈ o(λx. |f x| powr - p)
      using less by(subst smallo-powr-iff) simp-all
    also have (λ-. ||c| powr (1 / - p)| powr - p) = (λ-. |c|) using less by(simp
add: powr-powr)
    also have o(λx. |f x| powr - p) = o(λx. f x powr - p) using nonneg
      by(auto intro!: landau-o.small.cong elim: eventually-rev-mp)
    finally have (λ-. c) ∈ o(λx. f x powr - p) by simp
    with less show ?thesis by(simp add: powr-minus[symmetric])
qed

```

**lemma** *bigo-const-inverse-powr*:

```

  assumes filterlim f at-top at-top
  shows (λ- :: - :: linorder. c :: real) ∈ O(λx. inverse (f x powr p)) ↔ c = 0 ∨
p ≤ 0
proof(cases p 0 :: real rule: linorder-cases)
  case p-pos: greater
    have (λ-. c) ∈ O(λx. inverse (f x powr p)) ↔ (λ-. |c|) ∈ O(λx. inverse (f x
powr p)) by simp
    also have |c| = |(|c| powr inverse p)| powr p using p-pos by(simp add: powr-powr)
    also { have eventually (λx. f x ≥ 0) at-top using assms by(simp add: filter-
lim-at-top)
      then have O(λx. inverse (f x powr p)) = O(λx. |inverse (f x)| powr p)
        by(intro landau-o.big.cong)(auto elim!: eventually-rev-mp)
      also have (λ-. |(|c| powr inverse p)| powr p) ∈ ... ↔
        (λ-. |c| powr (inverse p)) ∈ O(λx. inverse (f x))
    }

```

```

    using p-pos by (rule bigo-abs-powr-iff)
    also note calculation }
    also have  $(\lambda-. |c| \text{ powr } (\text{inverse } p)) \in O(\lambda x. \text{inverse } (f x)) \longleftrightarrow c = 0$  using
    assms by simp
    finally show ?thesis using p-pos by simp
next
  case equal
    from assms have eventually  $(\lambda x. f x \geq 1)$  at-top using assms by(simp add:
filterlim-at-top)
    then have  $O(\lambda x. \text{inverse } (f x \text{ powr } p)) = O(\lambda x. 1)$ 
      by(intro landau-o.big.cong) (auto simp add: equal elim!: eventually-rev-mp)
    then show ?thesis using equal by simp
next
  case less
    from assms have *:  $\forall_F x$  in at-top.  $1 \leq f x$  by(simp add: filterlim-at-top)
    then have  $(\lambda-. |c| \text{ powr } (1 / - p)) \in O(f)$ 
      by(intro bigoI[where  $c = |c| \text{ powr } (1 / - p)$ ])
      (auto intro: order-trans[OF - mult-left-mono, rotated] elim!: eventually-rev-mp[OF
- always-eventually])
    then have  $(\lambda-. ||c| \text{ powr } (1 / - p)| \text{ powr } - p) \in O(\lambda x. |f x| \text{ powr } - p)$ 
      using less by (subst bigo-powr-iff) simp-all
    also have  $(\lambda-. ||c| \text{ powr } (1 / - p)| \text{ powr } - p) = (\lambda-. |c|)$  using less by(simp
add: powr-powr)
    also have  $O(\lambda x. |f x| \text{ powr } - p) = O(\lambda x. f x \text{ powr } - p)$  using *
      by (auto intro!: landau-o.big.cong elim: eventually-rev-mp)
    finally have  $(\lambda-. c) \in O(\lambda x. f x \text{ powr } - p)$  by simp
    thus ?thesis using less by (simp add: powr-minus[symmetric])
qed
end

```