

# Formalization of Recursive Path Orders for Lambda-Free Higher-Order Terms

Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand

October 11, 2017

## Abstract

This Isabelle/HOL formalization defines recursive path orders (RPOs) for higher-order terms without  $\lambda$ -abstraction and proves many useful properties about them. The main order fully coincides with the standard RPO on first-order terms also in the presence of currying, distinguishing it from previous work. An optimized variant is formalized as well. It appears promising as the basis of a higher-order superposition calculus.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Utilities for Lambda-Free Orders</b>	<b>2</b>
2.1	Finite Sets . . . . .	2
2.2	Function Power . . . . .	3
2.3	Least Operator . . . . .	3
2.4	Antisymmetric Relations . . . . .	3
2.5	Acyclic Relations . . . . .	3
2.6	Reflexive, Transitive Closure . . . . .	4
2.7	Well-Founded Relations . . . . .	4
2.8	Wellorders . . . . .	6
2.9	Lists . . . . .	6
2.10	Extended Natural Numbers . . . . .	7
2.11	Multisets . . . . .	7
<b>3</b>	<b>Lambda-Free Higher-Order Terms</b>	<b>12</b>
3.1	Precedence on Symbols . . . . .	12
3.2	Heads . . . . .	12
3.3	Terms . . . . .	12
3.4	Substitutions . . . . .	15
3.5	Subterms . . . . .	15
3.6	Maximum Arities . . . . .	16
3.7	Potential Heads of Ground Instances of Variables . . . . .	18
<b>4</b>	<b>Infinite (Non-Well-Founded) Chains</b>	<b>20</b>
<b>5</b>	<b>Extension Orders</b>	<b>23</b>
5.1	Locales . . . . .	23
5.2	Lexicographic Extension . . . . .	27
5.3	Reverse (Right-to-Left) Lexicographic Extension . . . . .	30
5.4	Generic Length Extension . . . . .	32
5.5	Length-Lexicographic Extension . . . . .	32
5.6	Reverse (Right-to-Left) Length-Lexicographic Extension . . . . .	33
5.7	Dershowitz–Manna Multiset Extension . . . . .	35
5.8	Huet–Oppen Multiset Extension . . . . .	40
5.9	Componentwise Extension . . . . .	50

<b>6</b>	<b>The Applicative Recursive Path Order for Lambda-Free Higher-Order Terms</b>	<b>55</b>
<b>7</b>	<b>The Graceful Recursive Path Order for Lambda-Free Higher-Order Terms</b>	<b>56</b>
7.1	Setup	56
7.2	Inductive Definitions	56
7.3	Transitivity	57
7.4	Irreflexivity	59
7.5	Subterm Property	60
7.6	Compatibility with Functions	60
7.7	Compatibility with Arguments	61
7.8	Stability under Substitution	62
7.9	Totality on Ground Terms	63
7.10	Well-foundedness	65
<b>8</b>	<b>The Optimized Graceful Recursive Path Order for Lambda-Free Higher-Order Terms</b>	<b>68</b>
8.1	Setup	68
8.2	Definition of the Order	69
8.3	Transitivity	69
8.4	Conditional Equivalence with Unoptimized Version	71
<b>9</b>	<b>Recursive Path Orders for Lambda-Free Higher-Order Terms</b>	<b>75</b>

## 1 Introduction

This Isabelle/HOL formalization defines recursive path orders (RPOs) for higher-order terms without  $\lambda$ -abstraction and proves many useful properties about them. The main order fully coincides with the standard RPO on first-order terms also in the presence of currying, distinguishing it from previous work. An optimized variant is formalized as well. It appears promising as the basis of a higher-order superposition calculus. We refer to our FoSSaCS 2017 paper for details.<sup>1</sup>

## 2 Utilities for Lambda-Free Orders

```
theory Lambda_Free_Util
imports HOL-Library.Extended_Nat HOL-Library.Multiset_Order
begin
```

This theory gathers various lemmas that likely belong elsewhere in Isabelle or the *Archive of Formal Proofs*. Most (but certainly not all) of them are used to formalize orders on  $\lambda$ -free higher-order terms.

```
hide-const (open) Complex.arg
```

### 2.1 Finite Sets

```
lemma finite_set_fold_singleton[simp]: Finite_Set.fold f z {x} = f x z
proof -
  have fold_graph f z {x} (f x z)
    by (auto intro: fold_graph.intros)
  moreover
  {
    fix X y
    have fold_graph f z X y  $\implies$  (X = {}  $\implies$  y = z)  $\wedge$  (X = {x}  $\implies$  y = f x z)
      by (induct rule: fold_graph.induct) auto
  }
  ultimately have (THE y. fold_graph f z {x} y) = f x z
    by blast
  thus ?thesis
    by (simp add: Finite_Set.fold_def)
qed
```

<sup>1</sup>[https://www21.in.tum.de/~blanchet/lambda\\_free\\_rpo\\_conf.pdf](https://www21.in.tum.de/~blanchet/lambda_free_rpo_conf.pdf)

## 2.2 Function Power

**lemma** *funpow\_lesseq\_iter*:  
**fixes**  $f :: ('a::order) \Rightarrow 'a$   
**assumes**  $mono: \bigwedge k. k \leq f k$  **and**  $m\_le\_n: m \leq n$   
**shows**  $(f \hat{\hat{m}}) k \leq (f \hat{\hat{n}}) k$   
**using**  $m\_le\_n$  **by** (*induct n*) (*fastforce simp: le\_Suc\_eq intro: mono order\_trans*)+

**lemma** *funpow\_less\_iter*:  
**fixes**  $f :: ('a::order) \Rightarrow 'a$   
**assumes**  $mono: \bigwedge k. k < f k$  **and**  $m\_lt\_n: m < n$   
**shows**  $(f \hat{\hat{m}}) k < (f \hat{\hat{n}}) k$   
**using**  $m\_lt\_n$  **by** (*induct n*) (*auto, blast intro: mono less\_trans dest: less\_antisym*)

## 2.3 Least Operator

**lemma** *Least\_eq[simp]*:  $(LEAST y. y = x) = x$  **and**  $(LEAST y. x = y) = x$  **for**  $x :: 'a::order$   
**by** (*blast intro: Least\_equality*)+

**lemma** *Least\_in\_nonempty\_set\_imp\_ex*:  
**fixes**  $f :: 'b \Rightarrow ('a::wellorder)$   
**assumes**  
 $A\_nemp: A \neq \{\}$  **and**  
 $P\_least: P (LEAST y. \exists x \in A. y = f x)$   
**shows**  $\exists x \in A. P (f x)$

**proof** –  
**obtain**  $a$  **where**  $a: a \in A$   
**using**  $A\_nemp$  **by** *fast*  
**have**  $\exists x. x \in A \wedge (LEAST y. \exists x. x \in A \wedge y = f x) = f x$   
**by** (*rule LeastI[of \_ f a]*) (*fast intro: a*)  
**thus** *?thesis*  
**by** (*metis P\_least*)  
**qed**

**lemma** *Least\_eq\_0\_enat*:  $P 0 \implies (LEAST x :: enat. P x) = 0$   
**by** (*simp add: Least\_equality*)

## 2.4 Antisymmetric Relations

**lemma** *irrefl\_trans\_imp\_antisym*:  $irrefl\ r \implies trans\ r \implies antisym\ r$   
**unfolding** *irrefl\_def trans\_def antisym\_def* **by** *fast*

**lemma** *irreflp\_transp\_imp\_antisymP*:  $irreflp\ p \implies transp\ p \implies antisymp\ p$   
**by** (*fact irrefl\_trans\_imp\_antisym [to\_pred]*)

## 2.5 Acyclic Relations

**lemma** *finite\_nonempty\_ex\_succ\_imp\_cyclic*:  
**assumes**  
 $fin: finite\ A$  **and**  
 $nemp: A \neq \{\}$  **and**  
 $ex\_y: \forall x \in A. \exists y \in A. (y, x) \in r$   
**shows**  $\neg acyclic\ r$

**proof** –  
**let**  $?R = \{(x, y). x \in A \wedge y \in A \wedge (x, y) \in r\}$   
  
**have**  $R\_sub\_r: ?R \subseteq r$   
**by** *auto*  
  
**have**  $?R \subseteq A \times A$   
**by** *auto*  
**hence**  $fin\_R: finite\ ?R$   
**by** (*auto intro: fin dest!: infinite\_super*)

```

have  $\neg$  acyclic ?R
  by (rule notI, drule finite_acyclic_wf[OF fin_R], unfold wf_eq_minimal, drule spec[of _ A],
      use ex_y nemp in blast)
thus ?thesis
  using R_sub_r acyclic_subset by auto
qed

```

## 2.6 Reflexive, Transitive Closure

**lemma** *relcomp\_subset\_left\_imp\_relcomp\_trancl\_subset\_left*:

```

assumes sub:  $R \circ S \subseteq R$ 
shows  $R \circ S^* \subseteq R$ 
proof
fix x
assume  $x \in R \circ S^*$ 
then obtain n where  $x \in R \circ S \wedge n$ 
  using rtrancl_imp_relpow by fastforce
thus  $x \in R$ 
proof (induct n)
case (Suc m)
thus ?case
  by (metis (no_types) O_assoc inf_sup_ord(3) le_iff_sup relcomp_distrib2 relpow_simps(2)
      relpow_commute sub_subsetCE)
qed auto
qed

```

**lemma** *f\_chain\_in\_rtrancl*:

```

assumes m_le_n:  $m \leq n$  and f_chain:  $\forall i \in \{m..<n\}. (f\ i, f\ (Suc\ i)) \in R$ 
shows  $(f\ m, f\ n) \in R^*$ 
proof (rule relpow_imp_rtrancl, rule relpow_fun_conv[THEN iffD2], intro exI conjI)
let ?g =  $\lambda i. f\ (m + i)$ 
let ?k =  $n - m$ 

show ?g 0 = f m
  by simp
show ?g ?k = f n
  using m_le_n by force
show  $(\forall i < ?k. (?g\ i, ?g\ (Suc\ i)) \in R)$ 
  by (simp add: f_chain)
qed

```

**lemma** *f\_rev\_chain\_in\_rtrancl*:

```

assumes m_le_n:  $m \leq n$  and f_chain:  $\forall i \in \{m..<n\}. (f\ (Suc\ i), f\ i) \in R$ 
shows  $(f\ n, f\ m) \in R^*$ 
by (rule f_chain_in_rtrancl[OF m_le_n, of  $\lambda i. f\ (n + m - i)$ , simplified])
(metis f_chain le_add_diff Suc_diff_Suc Suc_leI atLeastLessThan_iff diff_Suc_diff_eq1 diff_less
  le_add1 less_le_trans zero_less_Suc)

```

## 2.7 Well-Founded Relations

**lemma** *wf\_app*:  $wf\ r \implies wf\ \{(x, y). (f\ x, f\ y) \in r\}$   
**unfolding** *wf\_eq\_minimal* **by** (intro allI, drule spec[of \_ f ' Q for Q]) auto

**lemma** *wfP\_app*:  $wfP\ p \implies wfP\ (\lambda x\ y. p\ (f\ x)\ (f\ y))$   
**unfolding** *wfP\_def* **by** (rule wf\_app[of  $\{(x, y). p\ x\ y\}$  f, simplified])

**lemma** *wf\_exists\_minimal*:

```

assumes wf: wf r and Q:  $Q\ x$ 
shows  $\exists x. Q\ x \wedge (\forall y. (f\ y, f\ x) \in r \implies \neg Q\ y)$ 
using wf_eq_minimal[THEN iffD1, OF wf_app[OF wf], rule_format, of _  $\{x. Q\ x\}$ , simplified, OF Q]
by blast

```

**lemma** *wfP\_exists\_minimal*:

```

assumes wf: wfP p and Q:  $Q\ x$ 

```

**shows**  $\exists x. Q x \wedge (\forall y. p (f y) (f x) \longrightarrow \neg Q y)$   
**by** (rule *wf\_exists\_minimal*[of  $\{(x, y). p x y\}$   $Q x$ ,  $OF wf[unfolded wfP_def]$   $Q$ , *simplified*])

**lemma** *finite\_irrefl\_trans\_imp\_wf*:  $finite\ r \implies irrefl\ r \implies trans\ r \implies wf\ r$   
**by** (erule *finite\_acyclic\_wf*) (*simp add: acyclic\_irrefl*)

**lemma** *finite\_irreflp\_transp\_imp\_wfp*:  
 $finite\ \{(x, y). p\ x\ y\} \implies irreflp\ p \implies transp\ p \implies wfp\ p$   
**using** *finite\_irrefl\_trans\_imp\_wf*[of  $\{(x, y). p\ x\ y\}$ ]  
**unfolding** *wfP\_def transp\_def irreflp\_def trans\_def irrefl\_def mem\_Collect\_eq prod.case*  
**by** *assumption*

**lemma** *wf\_infinite\_down\_chain\_compatible*:

**assumes**

*wf\_R*:  $wf\ R$  **and**

*inf\_chain\_RS*:  $\forall i. (f\ (Suc\ i), f\ i) \in R \cup S$  **and**

*O\_subset*:  $R\ O\ S \subseteq R$

**shows**  $\exists k. \forall i. (f\ (Suc\ (i + k)), f\ (i + k)) \in S$

**proof** (rule *ccontr*)

**assume**  $\nexists k. \forall i. (f\ (Suc\ (i + k)), f\ (i + k)) \in S$

**hence**  $\forall k. \exists i. (f\ (Suc\ (i + k)), f\ (i + k)) \notin S$

**by** *blast*

**hence**  $\forall k. \exists i > k. (f\ (Suc\ i), f\ i) \notin S$

**by** (*metis add.commute add\_Suc less\_add\_Suc1*)

**hence**  $\forall k. \exists i > k. (f\ (Suc\ i), f\ i) \in R$

**using** *inf\_chain\_RS* **by** *blast*

**hence**  $\exists i > k. (f\ (Suc\ i), f\ i) \in R \wedge (\forall j > k. (f\ (Suc\ j), f\ j) \in R \longrightarrow j \geq i)$  **for**  $k$

**using** *wf\_eq\_minimal*[*THEN iffD1*, *OF wf\_less*, *rule\_format*,  
of  $\{i. i > k \wedge (f\ (Suc\ i), f\ i) \in R\}$ , *simplified*]

**by** (*meson not\_less*)

**then obtain**  $j\_of$  **where**

$j\_of\_gt$ :  $\bigwedge k. j\_of\ k > k$  **and**

$j\_of\_in\_R$ :  $\bigwedge k. (f\ (Suc\ (j\_of\ k)), f\ (j\_of\ k)) \in R$  **and**

$j\_of\_min$ :  $\bigwedge k. \forall j > k. (f\ (Suc\ j), f\ j) \in R \longrightarrow j \geq j\_of\ k$

**by** *moura*

**have**  $j\_of\_min\_s$ :  $\bigwedge k\ j. j > k \implies j < j\_of\ k \implies (f\ (Suc\ j), f\ j) \in S$

**using**  $j\_of\_min$  *inf\_chain\_RS* **by** *fastforce*

**define**  $g :: nat \Rightarrow 'a$  **where**  $\bigwedge k. g\ k = f\ (Suc\ ((j\_of\ \wedge\wedge\ k)\ 0))$

**have** *between\_g*[*simplified*]:  $(f\ ((j\_of\ \wedge\wedge\ (Suc\ i))\ 0), f\ (Suc\ ((j\_of\ \wedge\wedge\ i)\ 0))) \in S^*$  **for**  $i$

**proof** (rule *f\_rev\_chain\_in\_rtrancl*; *clarify?*)

**show**  $Suc\ ((j\_of\ \wedge\wedge\ i)\ 0) \leq (j\_of\ \wedge\wedge\ Suc\ i)\ 0$

**using**  $j\_of\_gt$  **by** (*simp add: Suc\_leI*)

**next**

**fix**  $ia$

**assume**  $ia$ :  $ia \in \{Suc\ ((j\_of\ \wedge\wedge\ i)\ 0)..<(j\_of\ \wedge\wedge\ Suc\ i)\ 0\}$

**have**  $ia\_gt$ :  $ia > (j\_of\ \wedge\wedge\ i)\ 0$

**using**  $ia$  **by** *auto*

**have**  $ia\_lt$ :  $ia < j\_of\ ((j\_of\ \wedge\wedge\ i)\ 0)$

**using**  $ia$  **by** *auto*

**show**  $(f\ (Suc\ ia), f\ ia) \in S$

**by** (rule  $j\_of\_min\_s$ [*OF ia\_gt ia\_lt*])

**qed**

**have**  $\bigwedge i. (g\ (Suc\ i), g\ i) \in R$

**unfolding**  $g\_def$  *funpow.simps comp\_def*

**by** (rule *set\_mp*[*OF relcomp\_subset\_left\_imp\_relcomp\_trancl\_subset\_left*[*OF O\_subset*]])

(rule *relcompI*[*OF j\_of\_in\_R between\_g*])

**moreover have**  $\forall f. \exists i. (f\ (Suc\ i), f\ i) \notin R$

**using**  $wf\_R$ [*unfolded wf\_iff\_no\_infinite\_down\_chain*] **by** *blast*

**ultimately show** *False*

by *blast*  
qed

## 2.8 Wellorders

**lemma** (in *wellorder*) *exists\_minimal*:  
**fixes**  $x :: 'a$   
**assumes**  $P\ x$   
**shows**  $\exists x. P\ x \wedge (\forall y. P\ y \longrightarrow y \geq x)$   
**using** *assms* **by** (*auto intro: LeastI Least\_le*)

## 2.9 Lists

**lemma** *rev\_induct2*[*consumes 1, case\_names Nil snoc*]:  
 $length\ xs = length\ ys \Longrightarrow P\ []\ [] \Longrightarrow$   
 $(\bigwedge x\ xs\ y\ ys. length\ xs = length\ ys \Longrightarrow P\ xs\ ys \Longrightarrow P\ (xs\ @\ [x])\ (ys\ @\ [y])) \Longrightarrow P\ xs\ ys$   
**proof** (*induct xs arbitrary: ys rule: rev\_induct*)  
**case** (*snoc x xs ys*)  
**thus** *?case*  
**by** (*induct ys rule: rev\_induct simp\_all*)  
**qed** *auto*

**lemma** *hd\_in\_set*:  $length\ xs \neq 0 \Longrightarrow hd\ xs \in set\ xs$   
**by** (*cases xs*) *auto*

**lemma** *in\_lists\_iff\_set*:  $xs \in lists\ A \longleftrightarrow set\ xs \subseteq A$   
**by** *fast*

**lemma** *butlast\_append\_Cons*[*simp*]:  $butlast\ (xs\ @\ y\ \#\ ys) = xs\ @\ butlast\ (y\ \#\ ys)$   
**using** *butlast\_append*[*of xs y # ys, simplified*] **by** *simp*

**lemma** *rev\_in\_lists*[*simp*]:  $rev\ xs \in lists\ A \longleftrightarrow xs \in lists\ A$   
**by** *auto*

**lemma** *hd\_le\_sum\_list*:  
**fixes**  $xs :: 'a::ordered\_ab\_semigroup\_monoid\_add\_imp\_le\ list$   
**assumes**  $xs \neq []$  **and**  $\forall i < length\ xs. xs\ !\ i \geq 0$   
**shows**  $hd\ xs \leq sum\_list\ xs$   
**using** *assms*  
**by** (*induct xs rule: rev\_induct, simp\_all,*  
*metis add\_cancel\_right\_left add\_increasing2 hd\_append2 lessI less\_SucI list.sel(1) nth\_append*  
*nth\_append\_length order\_refl self\_append\_conv2 sum\_list.Nil*)

**lemma** *sum\_list\_ge\_length\_times*:  
**fixes**  $a :: 'a::\{ordered\_ab\_semigroup\_add,semiring\_1\}$   
**assumes**  $\forall i < length\ xs. xs\ !\ i \geq a$   
**shows**  $sum\_list\ xs \geq of\_nat\ (length\ xs) * a$   
**using** *assms*  
**proof** (*induct xs*)  
**case** (*Cons x xs*)  
**note**  $ih = this(1)$  **and**  $xxs\_i\_ge\_a = this(2)$   
  
**have**  $xxs\_i\_ge\_a: \forall i < length\ xs. xs\ !\ i \geq a$   
**using**  $xxs\_i\_ge\_a$  **by** *auto*  
  
**have**  $x \geq a$   
**using**  $xxs\_i\_ge\_a$  **by** *auto*  
**thus** *?case*  
**using**  $ih[OF\ xxs\_i\_ge\_a]$  **by** (*simp add: ring\_distrib ordered\_ab\_semigroup\_add\_class.add\_mono*)  
**qed** *auto*

**lemma** *prod\_list\_nonneg*:  
**fixes**  $xs :: ('a :: \{ordered\_semiring\_0,linordered\_nonzero\_semiring\})\ list$   
**assumes**  $\bigwedge x. x \in set\ xs \Longrightarrow x \geq 0$

**shows**  $\text{prod\_list } xs \geq 0$   
**using**  $\text{assms}$  **by**  $(\text{induct } xs)$   $\text{auto}$

**lemma**  $\text{zip\_append\_0\_upt}$ :  
 $\text{zip } (xs @ ys) [0..<\text{length } xs + \text{length } ys] =$   
 $\text{zip } xs [0..<\text{length } xs] @ \text{zip } ys [\text{length } xs..<\text{length } xs + \text{length } ys]$   
**proof**  $(\text{induct } ys \text{ arbitrary}: xs)$   
**case**  $(\text{Cons } y \ ys)$   
**note**  $ih = \text{this}$   
**show**  $?case$   
**using**  $ih[\text{of } xs @ [y]]$  **by**  $(\text{simp}, \text{cases } ys, \text{simp}, \text{simp add: upt\_rec})$   
**qed**  $\text{auto}$

**lemma**  $\text{zip\_eq\_butlast\_last}$ :  
**assumes**  $\text{len\_gt0}: \text{length } xs > 0$  **and**  $\text{len\_eq}: \text{length } xs = \text{length } ys$   
**shows**  $\text{zip } xs \ ys = \text{zip } (\text{butlast } xs) (\text{butlast } ys) @ [(\text{last } xs, \text{last } ys)]$   
**using**  $\text{len\_eq len\_gt0}$  **by**  $(\text{induct rule: list\_induct2}) \text{ auto}$

## 2.10 Extended Natural Numbers

**lemma**  $\text{the\_enat\_0}[\text{simp}]: \text{the\_enat } 0 = 0$   
**by**  $(\text{simp add: zero\_enat\_def})$

**lemma**  $\text{the\_enat\_1}[\text{simp}]: \text{the\_enat } 1 = 1$   
**by**  $(\text{simp add: one\_enat\_def})$

**lemma**  $\text{enat\_le\_minus\_1\_imp\_lt}: m \leq n - 1 \implies n \neq \infty \implies n \neq 0 \implies m < n$  **for**  $m \ n :: \text{enat}$   
**by**  $(\text{cases } m; \text{cases } n; \text{simp add: zero\_enat\_def one\_enat\_def})$

**lemma**  $\text{enat\_diff\_diff\_eq}: k - m - n = k - (m + n)$  **for**  $k \ m \ n :: \text{enat}$   
**by**  $(\text{cases } k; \text{cases } m; \text{cases } n) \text{ auto}$

**lemma**  $\text{enat\_sub\_add\_same}[\text{intro}]: n \leq m \implies m = m - n + n$  **for**  $m \ n :: \text{enat}$   
**by**  $(\text{cases } m; \text{cases } n) \text{ auto}$

**lemma**  $\text{enat\_the\_enat\_iden}[\text{simp}]: n \neq \infty \implies \text{enat } (\text{the\_enat } n) = n$   
**by**  $\text{auto}$

**lemma**  $\text{the\_enat\_minus\_nat}: m \neq \infty \implies \text{the\_enat } (m - \text{enat } n) = \text{the\_enat } m - n$   
**by**  $\text{auto}$

**lemma**  $\text{enat\_the\_enat\_le}: \text{enat } (\text{the\_enat } x) \leq x$   
**by**  $(\text{cases } x; \text{simp})$

**lemma**  $\text{enat\_the\_enat\_minus\_le}: \text{enat } (\text{the\_enat } (x - y)) \leq x$   
**by**  $(\text{cases } x; \text{cases } y; \text{simp})$

**lemma**  $\text{enat\_le\_imp\_minus\_le}: k \leq m \implies k - n \leq m$  **for**  $k \ m \ n :: \text{enat}$   
**by**  $(\text{metis } \text{Groups.add\_ac}(2) \text{ enat\_diff\_diff\_eq } \text{enat\_ord\_simps}(3) \text{ enat\_sub\_add\_same}$   
 $\text{ enat\_the\_enat\_iden } \text{enat\_the\_enat\_minus\_le } \text{idiff\_0\_right } \text{idiff\_infinity } \text{idiff\_infinity\_right}$   
 $\text{ order\_trans\_rules}(23) \text{ plus\_enat\_simps}(3))$

**lemma**  $\text{add\_diff\_assoc2\_enat}: m \geq n \implies m - n + p = m + p - n$  **for**  $m \ n \ p :: \text{enat}$   
**by**  $(\text{cases } m; \text{cases } n; \text{cases } p; \text{auto})$

**lemma**  $\text{enat\_mult\_minus\_distrib}: \text{enat } x * (y - z) = \text{enat } x * y - \text{enat } x * z$   
**by**  $(\text{cases } y; \text{cases } z; \text{auto simp: enat\_0\_right\_diff\_distrib'})$

## 2.11 Multisets

**lemma**  $\text{add\_mset\_lt\_left\_lt}: a < b \implies \text{add\_mset } a \ A < \text{add\_mset } b \ A$   
**unfolding**  $\text{less\_multiset}_{HO}$  **by**  $\text{auto}$

**lemma**  $\text{add\_mset\_le\_left\_le}: a \leq b \implies \text{add\_mset } a \ A \leq \text{add\_mset } b \ A$  **for**  $a :: 'a :: \text{linorder}$

**unfolding** *less\_multiset<sub>HO</sub>* **by** *auto*

**lemma** *add\_mset\_lt\_right\_lt*:  $A < B \implies \text{add\_mset } a A < \text{add\_mset } a B$   
**unfolding** *less\_multiset<sub>HO</sub>* **by** *auto*

**lemma** *add\_mset\_le\_right\_le*:  $A \leq B \implies \text{add\_mset } a A \leq \text{add\_mset } a B$   
**unfolding** *less\_multiset<sub>HO</sub>* **by** *auto*

**lemma** *add\_mset\_lt\_lt\_lt*:  
**assumes** *a lt b*:  $a < b$  **and** *A le B*:  $A < B$   
**shows** *add\_mset a A < add\_mset b B*  
**by** (*rule less\_trans[OF add\_mset\_lt\_left\_lt[OF a lt b] add\_mset\_lt\_right\_lt[OF A le B]]*)

**lemma** *add\_mset\_lt\_lt\_le*:  $a < b \implies A \leq B \implies \text{add\_mset } a A < \text{add\_mset } b B$   
**using** *add\_mset\_lt\_lt\_le\_neq\_trans* **by** *fastforce*

**lemma** *add\_mset\_lt\_le\_lt*:  $a \leq b \implies A < B \implies \text{add\_mset } a A < \text{add\_mset } b B$  **for**  $a :: 'a :: \text{linorder}$   
**using** *add\_mset\_lt\_lt\_le* **by** (*metis add\_mset\_lt\_right\_lt le\_less*)

**lemma** *add\_mset\_le\_le\_le*:  
**fixes**  $a :: 'a :: \text{linorder}$   
**assumes** *a le b*:  $a \leq b$  **and** *A le B*:  $A \leq B$   
**shows** *add\_mset a A \leq add\_mset b B*  
**by** (*rule order.trans[OF add\_mset\_le\_left\_le[OF a le b] add\_mset\_le\_right\_le[OF A le B]]*)

**declare** *filter\_eq\_replicate\_mset [simp] image\_mset\_subseteq\_mono [intro]*

**lemma** *nonempty\_subseteq\_mset\_eq\_singleton*:  $M \neq \{\#\} \implies M \subseteq\# \{\#x\# \implies M = \{\#x\#\}$   
**by** (*cases M*) (*auto dest: subset\_mset.diff\_add*)

**lemma** *nonempty\_subseteq\_mset\_iff\_singleton*:  $(M \neq \{\#\} \wedge M \subseteq\# \{\#x\# \wedge P) \longleftrightarrow M = \{\#x\#\} \wedge P$   
**by** (*cases M*) (*auto dest: subset\_mset.diff\_add*)

**lemma** *count\_gt\_imp\_in\_mset*[*intro*]:  $\text{count } M x > n \implies x \in\# M$   
**using** *count\_greater\_zero\_iff* **by** *fastforce*

**lemma** *size\_lt\_imp\_ex\_count\_lt*:  $\text{size } M < \text{size } N \implies \exists x \in\# N. \text{count } M x < \text{count } N x$   
**by** (*metis count\_eq\_zero\_iff leD not\_le\_imp\_less not\_less\_zero size\_mset\_mono subseteq\_mset\_def*)

**lemma** *filter\_filter\_mset*[*simp*]:  $\{\#x \in\# \{\#x \in\# M. Q x\#. P x\#\} = \{\#x \in\# M. P x \wedge Q x\#\}$   
**by** (*induct M*) *auto*

**lemma** *size\_filter\_unsat\_elem*:  
**assumes**  $x \in\# M$  **and**  $\neg P x$   
**shows**  $\text{size } \{\#x \in\# M. P x\# < \text{size } M$   
**proof** –  
**have**  $\text{size } (\text{filter\_mset } P M) \neq \text{size } M$   
**using** *assms* **by** (*metis add.right\_neutral add\_diff\_cancel\_left' count\_filter\_mset mem\_Collect\_eq multiset\_partition\_nonempty\_has\_size set\_mset\_def size\_union*)  
**then show** *?thesis*  
**by** (*meson leD nat\_neq\_iff size\_filter\_mset\_lesseq*)  
**qed**

**lemma** *size\_filter\_ne\_elem*:  $x \in\# M \implies \text{size } \{\#y \in\# M. y \neq x\# < \text{size } M$   
**by** (*simp add: size\_filter\_unsat\_elem[of x M \lambda y. y \neq x]*)

**lemma** *size\_eq\_ex\_count\_lt*:  
**assumes**  
*sz\_m\_eq\_n*:  $\text{size } M = \text{size } N$  **and**  
*m\_eq\_n*:  $M \neq N$   
**shows**  $\exists x. \text{count } M x < \text{count } N x$   
**proof** –  
**obtain**  $x$  **where**  $\text{count } M x \neq \text{count } N x$



```

    using m_eq_n by (meson multiset_eqI)
moreover
{
  assume count M x < count N x
  hence ?thesis
    by blast
}
moreover
{
  assume cnt_x: count M x > count N x

  have size {#y ∈# M. y = x#} + size {#y ∈# M. y ≠ x#} =
    size {#y ∈# N. y = x#} + size {#y ∈# N. y ≠ x#}
    using sz_m_eq_n multiset_partition by (metis size_union)
  hence sz_m_minus_x: size {#y ∈# M. y ≠ x#} < size {#y ∈# N. y ≠ x#}
    using cnt_x by simp
  then obtain y where count {#y ∈# M. y ≠ x#} y < count {#y ∈# N. y ≠ x#} y
    using size_lt_imp_ex_count_lt[OF sz_m_minus_x] by blast
  hence count M y < count N y
    by (metis count_filter_mset_less_asym)
  hence ?thesis
    by blast
}
ultimately show ?thesis
  by fastforce
qed

lemma count_image_mset_lt_imp_lt_raw:
  assumes
    finite A and
    A = set_mset M ∪ set_mset N and
    count (image_mset f M) b < count (image_mset f N) b
  shows ∃x. f x = b ∧ count M x < count N x
  using assms
proof (induct A arbitrary: M N b rule: finite_induct)
  case (insert x F)
  note fin = this(1) and x_ni_f = this(2) and ih = this(3) and x_f_eq_m_n = this(4) and
    cnt_b = this(5)

  let ?Ma = {#y ∈# M. y ≠ x#}
  let ?Mb = {#y ∈# M. y = x#}
  let ?Na = {#y ∈# N. y ≠ x#}
  let ?Nb = {#y ∈# N. y = x#}

  have m_part: M = ?Mb + ?Ma and n_part: N = ?Nb + ?Na
    using multiset_partition by blast+

  have f_eq_ma_na: F = set_mset ?Ma ∪ set_mset ?Na
    using x_f_eq_m_n x_ni_f by auto

  show ?case
proof (cases count (image_mset f ?Ma) b < count (image_mset f ?Na) b)
  case cnt_ba: True
  obtain xa where f xa = b and count ?Ma xa < count ?Na xa
    using ih[OF f_eq_ma_na cnt_ba] by blast
  thus ?thesis
    by (metis count_filter_mset_not_less0)
next
  case cnt_ba: False
  have fx_eq_b: f x = b
    using cnt_b cnt_ba by (subst (asm) m_part, subst (asm) n_part, auto, presburger)
  moreover have count M x < count N x
    using cnt_b cnt_ba by (subst (asm) m_part, subst (asm) n_part, auto simp: fx_eq_b)

```

ultimately show ?thesis  
 by blast  
 qed  
 qed auto

lemma count\_image\_mset\_lt\_imp\_lt:  
 assumes cnt\_b: count (image\_mset f M) b < count (image\_mset f N) b  
 shows  $\exists x. f x = b \wedge \text{count } M x < \text{count } N x$   
 by (rule count\_image\_mset\_lt\_imp\_lt\_raw[of set\_mset M  $\cup$  set\_mset N, OF\_refl cnt\_b]) auto

lemma count\_image\_mset\_le\_imp\_lt\_raw:  
 assumes  
 finite A and  
 A = set\_mset M  $\cup$  set\_mset N and  
 count (image\_mset f M) (f a) + count N a < count (image\_mset f N) (f a) + count M a  
 shows  $\exists b. f b = f a \wedge \text{count } M b < \text{count } N b$   
 using assms  
 proof (induct A arbitrary: M N rule: finite\_induct)  
 case (insert x F)  
 note fin = this(1) and x\_ni\_f = this(2) and ih = this(3) and x\_f\_eq\_m\_n = this(4) and  
 cnt\_lt = this(5)

let ?Ma = {#y  $\in$  # M. y  $\neq$  x#}  
 let ?Mb = {#y  $\in$  # M. y = x#}  
 let ?Na = {#y  $\in$  # N. y  $\neq$  x#}  
 let ?Nb = {#y  $\in$  # N. y = x#}

have m\_part: M = ?Mb + ?Ma and n\_part: N = ?Nb + ?Na  
 using multiset\_partition by blast+

have f\_eq\_ma\_na: F = set\_mset ?Ma  $\cup$  set\_mset ?Na  
 using x\_f\_eq\_m\_n x\_ni\_f by auto

show ?case  
 proof (cases f x = f a)  
 case fx\_ne\_fa: False

have cnt\_fma\_fa: count (image\_mset f ?Ma) (f a) = count (image\_mset f M) (f a)  
 using fx\_ne\_fa by (subst (2) m\_part) auto  
 have cnt\_fna\_fa: count (image\_mset f ?Na) (f a) = count (image\_mset f N) (f a)  
 using fx\_ne\_fa by (subst (2) n\_part) auto  
 have cnt\_ma\_a: count ?Ma a = count M a  
 using fx\_ne\_fa by (subst (2) m\_part) auto  
 have cnt\_na\_a: count ?Na a = count N a  
 using fx\_ne\_fa by (subst (2) n\_part) auto

obtain b where fb\_eq\_fa: f b = f a and cnt\_b: count ?Ma b < count ?Na b  
 using ih[OF f\_eq\_ma\_na] cnt\_lt unfolding cnt\_fma\_fa cnt\_fna\_fa cnt\_ma\_a cnt\_na\_a by blast  
 have fx\_ne\_fb: f x  $\neq$  f b  
 using fb\_eq\_fa fx\_ne\_fa by simp

have cnt\_ma\_b: count ?Ma b = count M b  
 using fx\_ne\_fb by (subst (2) m\_part) auto  
 have cnt\_na\_b: count ?Na b = count N b  
 using fx\_ne\_fb by (subst (2) n\_part) auto

show ?thesis  
 using fb\_eq\_fa cnt\_b unfolding cnt\_ma\_b cnt\_na\_b by blast

next  
 case fx\_eq\_fa: True  
 show ?thesis  
 proof (cases x = a)  
 case x\_eq\_a: True

```

have count (image_mset f ?Ma) (f a) + count ?Na a
  < count (image_mset f ?Na) (f a) + count ?Ma a
  using cnt_lt x_eq_a by (subst (asm) (1 2) m_part, subst (asm) (1 2) n_part, auto)
thus ?thesis
  using ih[OF f_eq_ma_na] by (metis count_filter_mset nat_neq_iff)
next
case x_ne_a: False
show ?thesis
proof (cases count M x < count N x)
  case True
  thus ?thesis
  using fx_eq_fa by blast
next
case False
hence cnt_x: count M x ≥ count N x
  by fastforce
have count M x + count (image_mset f ?Ma) (f a) + count ?Na a
  < count N x + count (image_mset f ?Na) (f a) + count ?Ma a
  using cnt_lt x_ne_a fx_eq_fa by (subst (asm) (1 2) m_part, subst (asm) (1 2) n_part, auto)
hence count (image_mset f ?Ma) (f a) + count ?Na a
  < count (image_mset f ?Na) (f a) + count ?Ma a
  using cnt_x by linarith
thus ?thesis
  using ih[OF f_eq_ma_na] by (metis count_filter_mset nat_neq_iff)
qed
qed
qed
qed auto

```

```

lemma count_image_mset_le_imp_lt:
assumes
  count (image_mset f M) (f a) ≤ count (image_mset f N) (f a) and
  count M a > count N a
shows ∃ b. f b = f a ∧ count M b < count N b
using assms by (auto intro: count_image_mset_le_imp_lt_raw[of set_mset M ∪ set_mset N])

```

```

lemma Max_in_mset: M ≠ {#} ⇒ Max_mset M ∈# M
by simp

```

```

lemma Max_lt_imp_lt_mset:
assumes n_nemp: N ≠ {#} and max: Max_mset M < Max_mset N (is ?max_M < ?max_N)
shows M < N
proof (cases M = {#})
case m_nemp: False

```

```

  have max_n_in_n: ?max_N ∈# N
  using n_nemp by simp
  have max_n_nin_m: ?max_N ∉# M
  using max Max_ge leD by auto

```

```

have M ≠ N
using max by auto

```

```

moreover

```

```

{
  fix y
  assume count N y < count M y
  hence y ∈# M
  by blast
  hence ?max_M ≥ y
  by simp
  hence ?max_N > y
  using max by auto
  hence ∃ x > y. count M x < count N x

```

```

    using max_n_nin_m max_n_in_n by fastforce
  }
  ultimately show ?thesis
    unfolding less_multisetHO by blast
qed (auto simp: n_nemp)

lemma fold_mset_singleton[simp]: fold_mset f z {#x#} = f x z
  by (simp add: fold_mset_def)

end

```

### 3 Lambda-Free Higher-Order Terms

```

theory Lambda_Free_Term
imports Lambda_Free_Util
abbrevs
  >s = >s
  >h = >hd
  ≤>h = ≤>hd
begin

```

This theory defines  $\lambda$ -free higher-order terms and related locales.

#### 3.1 Precedence on Symbols

```

locale gt_sym =
  fixes
    gt_sym :: 's ⇒ 's ⇒ bool (infix >s 50)
  assumes
    gt_sym_irrefl: ¬ f >s f and
    gt_sym_trans: h >s g ⇒ g >s f ⇒ h >s f and
    gt_sym_total: f >s g ∨ g >s f ∨ g = f and
    gt_sym_wf: wfP (λf g. g >s f)
begin

lemma gt_sym_antisym: f >s g ⇒ ¬ g >s f
  by (metis gt_sym_irrefl gt_sym_trans)

end

```

#### 3.2 Heads

```

datatype (plugins del: size) (syms_hd: 's, vars_hd: 'v) hd =
  is_Var: Var (var: 'v)
| Sym (sym: 's)

abbreviation is_Sym :: ('s, 'v) hd ⇒ bool where
  is_Sym ζ ≡ ¬ is_Var ζ

lemma finite_vars_hd[simp]: finite (vars_hd ζ)
  by (cases ζ) auto

lemma finite_syms_hd[simp]: finite (syms_hd ζ)
  by (cases ζ) auto

```

#### 3.3 Terms

```

consts head0 :: 'a

datatype (syms: 's, vars: 'v) tm =
  is_Hd: Hd (head: ('s, 'v) hd)
| App (fun: ('s, 'v) tm) (arg: ('s, 'v) tm)
where

```

```

  head (App s _) = head0 s
| fun (Hd ζ) = Hd ζ
| arg (Hd ζ) = Hd ζ

```

**overloading**  $head0 \equiv head0 :: ('s, 'v) tm \Rightarrow ('s, 'v) hd$   
**begin**

```

primrec head0 :: ('s, 'v) tm  $\Rightarrow$  ('s, 'v) hd where
  head0 (Hd ζ) = ζ
| head0 (App s _) = head0 s

```

**end**

**lemma**  $head\_App[simp]$ :  $head (App s t) = head s$   
**by** (cases s) auto

**declare**  $tm.sel(2)[simp del]$

**lemma**  $head\_fun[simp]$ :  $head (fun s) = head s$   
**by** (cases s) auto

**abbreviation**  $ground :: ('s, 'v) tm \Rightarrow bool$  **where**  
 $ground t \equiv vars t = \{\}$

**abbreviation**  $is\_App :: ('s, 'v) tm \Rightarrow bool$  **where**  
 $is\_App s \equiv \neg is\_Hd s$

**lemma**  
 $size\_fun\_lt: is\_App s \Longrightarrow size (fun s) < size s$  **and**  
 $size\_arg\_lt: is\_App s \Longrightarrow size (arg s) < size s$   
**by** (cases s; simp)+

**lemma**  
 $finite\_vars[simp]$ :  $finite (vars s)$  **and**  
 $finite\_syms[simp]$ :  $finite (syms s)$   
**by** (induct s) auto

**lemma**  
 $vars\_head\_subsetq: vars\_hd (head s) \subseteq vars s$  **and**  
 $syms\_head\_subsetq: syms\_hd (head s) \subseteq syms s$   
**by** (induct s) auto

**fun**  $args :: ('s, 'v) tm \Rightarrow ('s, 'v) tm list$  **where**  
 $args (Hd _) = []$   
|  $args (App s t) = args s @ [t]$

**lemma**  $set\_args\_fun: set (args (fun s)) \subseteq set (args s)$   
**by** (cases s) auto

**lemma**  $arg\_in\_args: is\_App s \Longrightarrow arg s \in set (args s)$   
**by** (cases s rule: tm.exhaust) auto

**lemma**  
 $vars\_args\_subsetq: si \in set (args s) \Longrightarrow vars si \subseteq vars s$  **and**  
 $syms\_args\_subsetq: si \in set (args s) \Longrightarrow syms si \subseteq syms s$   
**by** (induct s) auto

**lemma**  $args\_Nil\_iff\_is\_Hd: args s = [] \longleftrightarrow is\_Hd s$   
**by** (cases s) auto

**abbreviation**  $num\_args :: ('s, 'v) tm \Rightarrow nat$  **where**  
 $num\_args s \equiv length (args s)$

**lemma** *size\_ge\_num\_args*:  $size\ s \geq num\_args\ s$   
**by** (*induct s*) *auto*

**lemma** *Hd\_head\_id*:  $num\_args\ s = 0 \implies Hd\ (head\ s) = s$   
**by** (*metis args.cases args.simps(2) length\_0\_conv snoc\_eq\_iff\_butlast tm.collapse(1) tm.disc(1)*)

**lemma** *one\_arg\_imp\_Hd*:  $num\_args\ s = 1 \implies s = App\ t\ u \implies t = Hd\ (head\ t)$   
**by** (*simp add: Hd\_head\_id*)

**lemma** *size\_in\_args*:  $s \in set\ (args\ t) \implies size\ s < size\ t$   
**by** (*induct t*) *auto*

**primrec** *apps* ::  $(\ 's, 'v) tm \Rightarrow (\ 's, 'v) tm\ list \Rightarrow (\ 's, 'v) tm$  **where**  
*apps s [] = s*  
| *apps s (t # ts) = apps (App s t) ts*

**lemma**  
*vars\_apps[simp]*:  $vars\ (apps\ s\ ss) = vars\ s \cup (\bigcup s \in set\ ss.\ vars\ s)$  **and**  
*syms\_apps[simp]*:  $syms\ (apps\ s\ ss) = syms\ s \cup (\bigcup s \in set\ ss.\ syms\ s)$  **and**  
*head\_apps[simp]*:  $head\ (apps\ s\ ss) = head\ s$  **and**  
*args\_apps[simp]*:  $args\ (apps\ s\ ss) = args\ s @ ss$  **and**  
*is\_App\_apps[simp]*:  $is\_App\ (apps\ s\ ss) \longleftrightarrow args\ (apps\ s\ ss) \neq []$  **and**  
*fun\_apps\_Nil[simp]*:  $fun\ (apps\ s\ []) = fun\ s$  **and**  
*fun\_apps\_Cons[simp]*:  $fun\ (apps\ (App\ s\ sa)\ ss) = apps\ s\ (butlast\ (sa\ #\ ss))$  **and**  
*arg\_apps\_Nil[simp]*:  $arg\ (apps\ s\ []) = arg\ s$  **and**  
*arg\_apps\_Cons[simp]*:  $arg\ (apps\ (App\ s\ sa)\ ss) = last\ (sa\ #\ ss)$   
**by** (*induct ss arbitrary: s sa*) (*auto simp: args\_Nil\_iff\_is\_Hd*)

**lemma** *apps\_append[simp]*:  $apps\ s\ (ss @ ts) = apps\ (apps\ s\ ss)\ ts$   
**by** (*induct ss arbitrary: s ts*) *auto*

**lemma** *App\_apps*:  $App\ (apps\ s\ ts)\ t = apps\ s\ (ts @ [t])$   
**by** *simp*

**lemma** *tm\_inject\_apps[iff, induct\_simp]*:  $apps\ (Hd\ \zeta)\ ss = apps\ (Hd\ \xi)\ ts \longleftrightarrow \zeta = \xi \wedge ss = ts$   
**by** (*metis args\_apps head\_apps same\_append\_eq tm.sel(1)*)

**lemma** *tm\_collapse\_apps[simp]*:  $apps\ (Hd\ (head\ s))\ (args\ s) = s$   
**by** (*induct s*) *auto*

**lemma** *tm\_expand\_apps*:  $head\ s = head\ t \implies args\ s = args\ t \implies s = t$   
**by** (*metis tm\_collapse\_apps*)

**lemma** *tm\_exhaust\_apps\_sel[case\_names apps]*:  $(s = apps\ (Hd\ (head\ s))\ (args\ s) \implies P) \implies P$   
**by** (*atomize\_elim, induct s*) *auto*

**lemma** *tm\_exhaust\_apps[case\_names apps]*:  $(\bigwedge \zeta\ ss.\ s = apps\ (Hd\ \zeta)\ ss \implies P) \implies P$   
**by** (*metis tm\_collapse\_apps*)

**lemma** *tm\_induct\_apps[case\_names apps]*:  
**assumes**  $\bigwedge \zeta\ ss.\ (\bigwedge s.\ s \in set\ ss \implies P\ s) \implies P\ (apps\ (Hd\ \zeta)\ ss)$   
**shows**  $P\ s$   
**using** *assms*  
**by** (*induct s taking: size rule: measure\_induct\_rule*) (*metis size\_in\_args tm\_collapse\_apps*)

**lemma**  
*ground\_fun*:  $ground\ s \implies ground\ (fun\ s)$  **and**  
*ground\_arg*:  $ground\ s \implies ground\ (arg\ s)$   
**by** (*induct s*) *auto*

**lemma** *ground\_head*:  $ground\ s \implies is\_Sym\ (head\ s)$   
**by** (*cases s rule: tm\_exhaust\_apps*) (*auto simp: is\_Var\_def*)

**lemma** *ground\_args*:  $t \in \text{set } (\text{args } s) \implies \text{ground } s \implies \text{ground } t$   
**by** (*induct s rule: tm\_induct\_apps*) *auto*

**primrec** *vars\_mset* ::  $(\text{'s}, \text{'v}) \text{tm} \Rightarrow \text{'v} \text{multiset}$  **where**  
*vars\_mset* (*Hd*  $\zeta$ ) = *mset\_set* (*vars\_hd*  $\zeta$ )  
| *vars\_mset* (*App*  $s$   $t$ ) = *vars\_mset*  $s$  + *vars\_mset*  $t$

**lemma** *set\_vars\_mset[simp]*:  $\text{set\_mset } (\text{vars\_mset } t) = \text{vars } t$   
**by** (*induct t*) *auto*

**lemma** *vars\_mset\_empty\_iff*[*iff*]:  $\text{vars\_mset } s = \{\#\} \iff \text{ground } s$   
**by** (*induct s*) (*auto simp: mset\_set\_empty\_iff*)

**lemma** *vars\_mset\_fun*[*intro*]:  $\text{vars\_mset } (\text{fun } t) \subseteq\# \text{vars\_mset } t$   
**by** (*cases t*) *auto*

**lemma** *vars\_mset\_arg*[*intro*]:  $\text{vars\_mset } (\text{arg } t) \subseteq\# \text{vars\_mset } t$   
**by** (*cases t*) *auto*

### 3.4 Substitutions

**primrec** *subst* ::  $(\text{'v} \Rightarrow (\text{'s}, \text{'v}) \text{tm}) \Rightarrow (\text{'s}, \text{'v}) \text{tm} \Rightarrow (\text{'s}, \text{'v}) \text{tm}$  **where**  
*subst*  $\rho$  (*Hd*  $\zeta$ ) = (*case*  $\zeta$  *of* *Var*  $x \Rightarrow \rho$   $x$  | *Sym*  $f \Rightarrow \text{Hd } (\text{Sym } f)$ )  
| *subst*  $\rho$  (*App*  $s$   $t$ ) = *App* (*subst*  $\rho$   $s$ ) (*subst*  $\rho$   $t$ )

**lemma** *subst\_apps[simp]*:  $\text{subst } \rho (\text{apps } s \text{ ts}) = \text{apps } (\text{subst } \rho \text{ s}) (\text{map } (\text{subst } \rho) \text{ ts})$   
**by** (*induct ts arbitrary: s*) *auto*

**lemma** *head\_subst[simp]*:  $\text{head } (\text{subst } \rho \text{ s}) = \text{head } (\text{subst } \rho (\text{Hd } (\text{head } s)))$   
**by** (*cases s rule: tm\_exhaust\_apps*) (*auto split: hd.split*)

**lemma** *args\_subst[simp]*:  
 $\text{args } (\text{subst } \rho \text{ s}) = (\text{case } \text{head } s \text{ of } \text{Var } x \Rightarrow \text{args } (\rho \text{ x}) \mid \text{Sym } f \Rightarrow []) @ \text{map } (\text{subst } \rho) (\text{args } s)$   
**by** (*cases s rule: tm\_exhaust\_apps*) (*auto split: hd.split*)

**lemma** *ground\_imp\_subst\_iden*:  $\text{ground } s \implies \text{subst } \rho \text{ s} = s$   
**by** (*induct s*) (*auto split: hd.split*)

**lemma** *vars\_mset\_subst[simp]*:  $\text{vars\_mset } (\text{subst } \rho \text{ s}) = (\bigcup\# \{\#\text{vars\_mset } (\rho \text{ x}). \text{x} \in\# \text{vars\_mset } s\#\})$

**proof** (*induct s*)  
**case** (*Hd*  $\zeta$ )  
**show** *?case*  
**by** (*cases*  $\zeta$ ) *auto*

**qed** *auto*

**lemma** *vars\_mset\_subst\_subseteq*:  
 $\text{vars\_mset } t \supseteq\# \text{vars\_mset } s \implies \text{vars\_mset } (\text{subst } \rho \text{ t}) \supseteq\# \text{vars\_mset } (\text{subst } \rho \text{ s})$   
**unfolding** *vars\_mset\_subst*  
**by** (*metis* (*no\_types*) *add\_diff\_cancel\_right' diff\_subset\_eq\_self image\_mset\_union sum\_mset.union subset\_mset.add\_diff\_inverse*)

**lemma** *vars\_subst\_subseteq*:  $\text{vars } t \supseteq \text{vars } s \implies \text{vars } (\text{subst } \rho \text{ t}) \supseteq \text{vars } (\text{subst } \rho \text{ s})$   
**unfolding** *set\_vars\_mset[symmetric]* *vars\_mset\_subst* **by** *auto*

### 3.5 Subterms

**inductive** *sub* ::  $(\text{'s}, \text{'v}) \text{tm} \Rightarrow (\text{'s}, \text{'v}) \text{tm} \Rightarrow \text{bool}$  **where**  
*sub\_refl*: *sub*  $s$   $s$   
| *sub\_fun*: *sub*  $s$   $t \implies \text{sub } s$  (*App*  $u$   $t$ )  
| *sub\_arg*: *sub*  $s$   $t \implies \text{sub } s$  (*App*  $t$   $u$ )

**inductive-cases** *sub\_HdE*[*simplified, elim*]: *sub*  $s$  (*Hd*  $\xi$ )  
**inductive-cases** *sub\_AppE*[*simplified, elim*]: *sub*  $s$  (*App*  $t$   $u$ )  
**inductive-cases** *sub\_Hd\_HdE*[*simplified, elim*]: *sub* (*Hd*  $\zeta$ ) (*Hd*  $\xi$ )

**inductive-cases** *sub\_Hd\_AppE*[*simplified, elim*]: *sub (Hd ζ) (App t u)*

**lemma** *in\_vars\_imp\_sub*:  $x \in \text{vars } s \iff \text{sub } (Hd \text{ (Var } x)) \text{ } s$   
**by** *induct (auto intro: sub.intros elim: hd.set\_cases(2))*

**lemma** *sub\_args*:  $s \in \text{set } (args \ t) \implies \text{sub } s \ t$   
**by** (*induct t*) (*auto intro: sub.intros*)

**lemma** *sub\_size*:  $\text{sub } s \ t \implies \text{size } s \leq \text{size } t$   
**by** *induct auto*

**lemma** *sub\_subst*:  $\text{sub } s \ t \implies \text{sub } (\text{subst } \rho \ s) \ (\text{subst } \rho \ t)$

**proof** (*induct t*)

**case** (*Hd ζ*)

**thus** *?case*

**by** (*cases ζ; blast intro: sub.intros*)

**qed** (*auto intro: sub.intros del: sub\_AppE elim!: sub\_AppE*)

**abbreviation** *proper\_sub* ::  $(\ 's, \ 'v) \text{ tm} \Rightarrow (\ 's, \ 'v) \text{ tm} \Rightarrow \text{bool}$  **where**  
*proper\_sub s t*  $\equiv \text{sub } s \ t \wedge s \neq t$

**lemma** *proper\_sub\_Hd[simp]*:  $\neg \text{proper\_sub } s \ (Hd \ \zeta)$   
**using** *sub.cases* **by** *blast*

**lemma** *proper\_sub\_subst*:

**assumes** *psub*: *proper\_sub s t*

**shows** *proper\_sub (subst ρ s) (subst ρ t)*

**proof** (*cases t*)

**case** *Hd*

**thus** *?thesis*

**using** *psub* **by** *simp*

**next**

**case** *t: (App t1 t2)*

**have**  $\text{sub } s \ t1 \vee \text{sub } s \ t2$

**using** *t psub* **by** *blast*

**hence**  $\text{sub } (\text{subst } \rho \ s) \ (\text{subst } \rho \ t1) \vee \text{sub } (\text{subst } \rho \ s) \ (\text{subst } \rho \ t2)$

**using** *sub\_subst* **by** *blast*

**thus** *?thesis*

**unfolding** *t* **by** (*auto intro: sub.intros dest: sub\_size*)

**qed**

### 3.6 Maximum Arities

**locale** *arity* =

**fixes**

*arity\_sym* ::  $\ 's \Rightarrow \text{enat}$  **and**

*arity\_var* ::  $\ 'v \Rightarrow \text{enat}$

**begin**

**primrec** *arity\_hd* ::  $(\ 's, \ 'v) \text{ hd} \Rightarrow \text{enat}$  **where**

*arity\_hd (Var x)* = *arity\_var x*

| *arity\_hd (Sym f)* = *arity\_sym f*

**definition** *arity* ::  $(\ 's, \ 'v) \text{ tm} \Rightarrow \text{enat}$  **where**

*arity s* = *arity\_hd (head s)* - *num\_args s*

**lemma** *arity\_simps[simp]*:

*arity (Hd ζ)* = *arity\_hd ζ*

*arity (App s t)* = *arity s* - 1

**by** (*auto simp: arity\_def enat\_diff\_diff\_eq add commute eSuc\_enat plus\_1\_eSuc(1)*)

**lemma** *arity\_apps[simp]*:  $\text{arity } (\text{apps } s \ ts) = \text{arity } s - \text{length } ts$

**proof** (*induct ts arbitrary: s*)

**case** (*Cons t ts*)



```

thus ?case
  by (case_tac arity s; simp add: one_enat_def)
qed simp

inductive wary :: ('s, 'v) tm  $\Rightarrow$  bool where
  wary_Hd[intro]: wary (Hd  $\zeta$ )
| wary_App[intro]: wary s  $\Longrightarrow$  wary t  $\Longrightarrow$  num_args s < arity_hd (head s)  $\Longrightarrow$  wary (App s t)

inductive-cases wary_HdE: wary (Hd  $\zeta$ )
inductive-cases wary_AppE: wary (App s t)
inductive-cases wary_binaryE[simplified]: wary (App (App s t) u)

lemma wary_fun[intro]: wary t  $\Longrightarrow$  wary (fun t)
  by (cases t) (auto elim: wary.cases)

lemma wary_arg[intro]: wary t  $\Longrightarrow$  wary (arg t)
  by (cases t) (auto elim: wary.cases)

lemma wary_args: s  $\in$  set (args t)  $\Longrightarrow$  wary t  $\Longrightarrow$  wary s
  by (induct t arbitrary: s, simp)
  (metis Un_iff args.simps(2) wary.cases insert_iff length_pos_if_in_set
  less_numeral_extra(3) list.set(2) list.size(3) set_append tm.distinct(1) tm.inject(2))

lemma wary_sub: sub s t  $\Longrightarrow$  wary t  $\Longrightarrow$  wary s
  by (induct rule: sub.induct) (auto elim: wary.cases)

lemma wary_inf_arity: ( $\bigwedge \zeta$ . arity_hd  $\zeta = \infty$ )  $\Longrightarrow$  wary s
  by induct auto

lemma wary_num_args_le_arity_head: wary s  $\Longrightarrow$  num_args s  $\leq$  arity_hd (head s)
  by (induct rule: wary.induct) (auto simp: zero_enat_def[symmetric] Suc_ile_eq)

lemma wary_apps:
  wary s  $\Longrightarrow$  ( $\bigwedge sa$ . sa  $\in$  set ss  $\Longrightarrow$  wary sa)  $\Longrightarrow$  length ss  $\leq$  arity s  $\Longrightarrow$  wary (apps s ss)
proof (induct ss arbitrary: s)
  case (Cons sa ss)
  note ih = this(1) and wary_s = this(2) and wary_ss = this(3) and nargs_s_sa_ss = this(4)
  show ?case
    unfolding apps.simps
  proof (rule ih)
    have wary sa
      using wary_ss by simp
    moreover have enat (num_args s) < arity_hd (head s)
      by (metis (mono_tags) One_nat_def add.comm neutral_arity_def diff_add_zero enat_ord_simps(1)
      idiff_enat_enat less_one list.size(4) nargs_s_sa_ss not_add_less2
      order.not_eq_order_implies_strict wary_num_args_le_arity_head wary_s)
    ultimately show wary (App s sa)
      by (rule wary_App[OF wary_s])
  next
  show  $\bigwedge sa$ . sa  $\in$  set ss  $\Longrightarrow$  wary sa
    using wary_ss by simp
  next
  show length ss  $\leq$  arity (App s sa)
  proof (cases arity s)
    case enat
    thus ?thesis
      using nargs_s_sa_ss by (simp add: one_enat_def)
  qed simp
qed
qed simp

lemma wary_cases_apps[consumes 1, case_names apps]:
  assumes

```

$wary\_t: wary\ t\ \mathbf{and}$   
 $apps: \bigwedge \zeta\ ss.\ t = apps\ (Hd\ \zeta)\ ss \implies (\bigwedge sa.\ sa \in set\ ss \implies wary\ sa) \implies length\ ss \leq arity\_hd\ \zeta \implies P$   
**shows**  $P$   
**using**  $apps$   
**proof** ( $atomize\_elim, cases\ t\ rule: tm\_exhaust\_apps$ )  
**case**  $t: (apps\ \zeta\ ss)$   
**show**  $\exists \zeta\ ss.\ t = apps\ (Hd\ \zeta)\ ss \wedge (\forall sa.\ sa \in set\ ss \implies wary\ sa) \wedge enat\ (length\ ss) \leq arity\_hd\ \zeta$   
**by** ( $rule\ exI[of\_ \zeta], rule\ exI[of\_ ss]$ )  
 $(auto\ simp: t\ wary\_args[OF\_ wary\_t]\ wary\_num\_args\_le\_arity\_head[OF\ wary\_t, unfolded\ t, simplified])$   
**qed**

**lemma**  $arity\_hd\_head: wary\ s \implies arity\_hd\ (head\ s) = arity\ s + num\_args\ s$   
**by** ( $simp\ add: arity\_def\ enat\_sub\_add\_same\ wary\_num\_args\_le\_arity\_head$ )

**lemma**  $arity\_head\_ge: arity\_hd\ (head\ s) \geq arity\ s$   
**by** ( $induct\ s$ ) ( $auto\ intro: enat\_le\_imp\_minus\_le$ )

**inductive**  $wary\_fo :: ('s, 'v)\ tm \Rightarrow bool$  **where**  
 $wary\_foI[intro]: is\_Hd\ s \vee is\_Sym\ (head\ s) \implies length\ (args\ s) = arity\_hd\ (head\ s) \implies$   
 $(\forall t \in set\ (args\ s).\ wary\_fo\ t) \implies wary\_fo\ s$

**lemma**  $wary\_fo\_args: s \in set\ (args\ t) \implies wary\_fo\ t \implies wary\_fo\ s$   
**by** ( $induct\ t\ arbitrary: s\ rule: tm\_induct\_apps, simp$ )  
 $(metis\ args.simps(1)\ args\_apps\ self\_append\_conv2\ wary\_fo.cases)$

**lemma**  $wary\_fo\_arg: wary\_fo\ (App\ s\ t) \implies wary\_fo\ t$   
**by** ( $erule\ wary\_fo.cases$ )  $auto$

**end**

### 3.7 Potential Heads of Ground Instances of Variables

**locale**  $ground\_heads = gt\_sym\ op\ >_s + arity\ arity\_sym\ arity\_var$   
**for**  
 $gt\_sym :: 's \Rightarrow 's \Rightarrow bool$  ( $\mathbf{infix}\ >_s\ 50$ ) **and**  
 $arity\_sym :: 's \Rightarrow enat$  **and**  
 $arity\_var :: 'v \Rightarrow enat +$   
**fixes**  
 $ground\_heads\_var :: 'v \Rightarrow 's\ set$   
**assumes**  
 $ground\_heads\_var\_arity: f \in ground\_heads\_var\ x \implies arity\_sym\ f \geq arity\_var\ x$  **and**  
 $ground\_heads\_var\_nonempty: ground\_heads\_var\ x \neq \{\}$   
**begin**

**primrec**  $ground\_heads :: ('s, 'v)\ hd \Rightarrow 's\ set$  **where**  
 $ground\_heads\ (Var\ x) = ground\_heads\_var\ x$   
 $| ground\_heads\ (Sym\ f) = \{f\}$

**lemma**  $ground\_heads\_arity: f \in ground\_heads\ \zeta \implies arity\_sym\ f \geq arity\_hd\ \zeta$   
**by** ( $cases\ \zeta$ ) ( $auto\ simp: ground\_heads\_var\_arity$ )

**lemma**  $ground\_heads\_nonempty[simp]: ground\_heads\ \zeta \neq \{\}$   
**by** ( $cases\ \zeta$ ) ( $auto\ simp: ground\_heads\_var\_nonempty$ )

**lemma**  $sym\_in\_ground\_heads: is\_Sym\ \zeta \implies sym\ \zeta \in ground\_heads\ \zeta$   
**by** ( $metis\ ground\_heads.simps(2)\ hd.collapse(2)\ hd.set\_sel(1)\ hd.simps(16)$ )

**lemma**  $ground\_hd\_in\_ground\_heads: ground\ s \implies sym\ (head\ s) \in ground\_heads\ (head\ s)$   
**by** ( $simp\ add: ground\_head\ sym\_in\_ground\_heads$ )

**lemma**  $some\_ground\_head\_arity: arity\_sym\ (SOME\ f.\ f \in ground\_heads\ (Var\ x)) \geq arity\_var\ x$   
**by** ( $simp\ add: ground\_heads\_var\_arity\ ground\_heads\_var\_nonempty\ some\_in\_eq$ )

**definition**  $wary\_subst :: ('v \Rightarrow ('s, 'v)\ tm) \Rightarrow bool$  **where**

$wary\_subst \varrho \longleftrightarrow$   
 $(\forall x. wary (\varrho x) \wedge arity (\varrho x) \geq arity\_var x \wedge ground\_heads (head (\varrho x)) \subseteq ground\_heads\_var x)$

**definition**  $strict\_wary\_subst :: ('v \Rightarrow ('s, 'v) tm) \Rightarrow bool$  **where**

$strict\_wary\_subst \varrho \longleftrightarrow$   
 $(\forall x. wary (\varrho x) \wedge arity (\varrho x) \in \{arity\_var x, \infty\} \wedge ground\_heads (head (\varrho x)) \subseteq ground\_heads\_var x)$

**lemma**  $strict\_imp\_wary\_subst: strict\_wary\_subst \varrho \Longrightarrow wary\_subst \varrho$

**unfolding**  $strict\_wary\_subst\_def wary\_subst\_def$  **using**  $eq\_iff$  **by**  $force$

**lemma**  $wary\_subst\_wary:$

**assumes**  $wary\_var: wary\_subst \varrho$  **and**  $wary\_s: wary s$

**shows**  $wary (subst \varrho s)$

**using**  $wary\_s$

**proof** ( $induct s$   $rule: tm.induct$ )

**case** ( $App s t$ )

**note**  $wary\_st = this(3)$

**from**  $wary\_st$  **have**  $wary\_s: wary s$

**by** ( $rule wary\_AppE$ )

**from**  $wary\_st$  **have**  $wary\_t: wary t$

**by** ( $rule wary\_AppE$ )

**from**  $wary\_st$  **have**  $nargs\_s\_lt: num\_args s < arity\_hd (head s)$

**by** ( $rule wary\_AppE$ )

**note**  $wary\_os = App(1)[OF wary\_s]$

**note**  $wary_ot = App(2)[OF wary\_t]$

**note**  $wary\_ox = wary\_var[unfolded wary\_subst\_def, rule\_format, THEN conjunct1]$

**note**  $ary\_ox = wary\_var[unfolded wary\_subst\_def, rule\_format, THEN conjunct2]$

**have**  $num\_args (\varrho x) + num\_args s < arity\_hd (head (\varrho x))$  **if**  $hd\_s: head s = Var x$  **for**  $x$

**proof** –

**have**  $ary\_hd\_s: arity\_hd (head s) = arity\_var x$

**using**  $hd\_s$   $arity\_hd.simps(1)$  **by**  $presburger$

**hence**  $num\_args s \leq arity (\varrho x)$

**by** ( $metis (no\_types) wary\_num\_args\_le\_arity\_head ary\_ox$   $dual\_order.trans$   $wary\_s$ )

**hence**  $num\_args s + num\_args (\varrho x) \leq arity\_hd (head (\varrho x))$

**by** ( $metis (no\_types) arity\_hd\_head[OF wary\_ox]$   $add\_right\_mono$   $plus\_enat\_simps(1)$ )

**thus**  $?thesis$

**using**  $ary\_hd\_s$  **by** ( $metis (no\_types) add.commute$   $add\_diff\_cancel\_left'$   $ary\_ox$   $arity\_def$   $idiff\_enat\_enat$   $leD$   $nargs\_s\_lt$   $order.not\_eq\_order\_implies\_strict$ )

**qed**

**hence**  $nargs\_os: num\_args (subst \varrho s) < arity\_hd (head (subst \varrho s))$

**using**  $nargs\_s\_lt$  **by** ( $auto$   $split: hd.split$ )

**show**  $?case$

**by**  $simp$  ( $rule wary\_App[OF wary\_os wary_ot nargs\_os]$ )

**qed** ( $auto$   $simp: wary\_var[unfolded wary\_subst\_def]$   $split: hd.split$ )

**lemmas**  $strict\_wary\_subst\_wary = wary\_subst\_wary[OF strict\_imp\_wary\_subst]$

**lemma**  $wary\_subst\_ground\_heads:$

**assumes**  $wary\_var: wary\_subst \varrho$

**shows**  $ground\_heads (head (subst \varrho s)) \subseteq ground\_heads (head s)$

**proof** ( $induct s$   $rule: tm\_induct\_apps$ )

**case** ( $apps \zeta ss$ )

**show**  $?case$

**proof** ( $cases \zeta$ )

**case**  $x: (Var x)$

**thus**  $?thesis$

**using**  $wary\_var$   $wary\_subst\_def$   $x$  **by**  $auto$

**qed**  $auto$

**qed**

**lemmas** *strict\_wary\_subst\_ground\_heads* = *wary\_subst\_ground\_heads*[*OF strict\_imp\_wary\_subst*]

**definition** *grounding\_ρ* :: '*v* ⇒ ('*s*, '*v*) *tm* **where**  
*grounding\_ρ* *x* = (let *s* = Hd (Sym (SOME *f*. *f* ∈ *ground\_heads\_var* *x*)) in  
 apps *s* (replicate (the\_enat (arity *s* - arity\_var *x*)) *s*))

**lemma** *ground\_grounding\_ρ*: *ground* (subst *grounding\_ρ* *s*)  
**by** (induct *s*) (auto simp: Let\_def *grounding\_ρ\_def* elim: hd.set\_cases(2) split: hd.split)

**lemma** *strict\_wary\_grounding\_ρ*: *strict\_wary\_subst* *grounding\_ρ*  
**unfolding** *strict\_wary\_subst\_def*

**proof** (intro allI conjI)  
**fix** *x*

**define** *f* **where** *f* = (SOME *f*. *f* ∈ *ground\_heads\_var* *x*)  
**define** *s* :: ('*s*, '*v*) *tm* **where** *s* = Hd (Sym *f*)

**have** *wary\_s*: *wary* *s*  
**unfolding** *s\_def* **by** (rule *wary\_Hd*)  
**have** *ary\_s\_ge\_x*: *arity* *s* ≥ *arity\_var* *x*  
**unfolding** *s\_def* *f\_def* **using** *some\_ground\_head\_arity* **by** *simp*  
**have** *gr\_ρ\_x*: *grounding\_ρ* *x* = apps *s* (replicate (the\_enat (arity *s* - arity\_var *x*)) *s*)  
**unfolding** *grounding\_ρ\_def* *Let\_def* *f\_def* [symmetric] *s\_def* [symmetric] **by** (rule *refl*)

**show** *wary* (*grounding\_ρ* *x*)  
**unfolding** *gr\_ρ\_x* **by** (auto intro!: *wary\_s* *wary\_apps*[*OF wary\_s*] *enat\_the\_enat\_minus\_le*)  
**show** *arity* (*grounding\_ρ* *x*) ∈ {*arity\_var* *x*, ∞}  
**unfolding** *gr\_ρ\_x* **using** *ary\_s\_ge\_x* **by** (cases *arity* *s*; cases *arity\_var* *x*; *simp*)  
**show** *ground\_heads* (head (*grounding\_ρ* *x*)) ⊆ *ground\_heads\_var* *x*  
**unfolding** *gr\_ρ\_x* *s\_def* *f\_def* **by** (*simp* add: *some\_in\_eq* *ground\_heads\_var\_nonempty*)

**qed**

**lemmas** *wary\_grounding\_ρ* = *strict\_wary\_grounding\_ρ*[*THEN strict\_imp\_wary\_subst*]

**definition** *gt\_hd* :: ('*s*, '*v*) *hd* ⇒ ('*s*, '*v*) *hd* ⇒ bool (**infix** ><sub>*hd*</sub> 50) **where**  
 $\xi >_{hd} \zeta \iff (\forall g \in \text{ground\_heads } \xi. \forall f \in \text{ground\_heads } \zeta. g >_s f)$

**definition** *comp\_hd* :: ('*s*, '*v*) *hd* ⇒ ('*s*, '*v*) *hd* ⇒ bool (**infix** ≤<sub>*hd*</sub> 50) **where**  
 $\xi \leq_{hd} \zeta \iff \xi = \zeta \vee \xi >_{hd} \zeta \vee \zeta >_{hd} \xi$

**lemma** *gt\_hd\_irrefl*: ¬  $\zeta >_{hd} \zeta$   
**unfolding** *gt\_hd\_def* **using** *gt\_sym\_irrefl* **by** (meson *ex\_in\_conv* *ground\_heads\_nonempty*)

**lemma** *gt\_hd\_trans*:  $\chi >_{hd} \xi \implies \xi >_{hd} \zeta \implies \chi >_{hd} \zeta$   
**unfolding** *gt\_hd\_def* **using** *gt\_sym\_trans* **by** (meson *ex\_in\_conv* *ground\_heads\_nonempty*)

**lemma** *gt\_sym\_imp\_hd*:  $g >_s f \implies \text{Sym } g >_{hd} \text{Sym } f$   
**unfolding** *gt\_hd\_def* **by** *simp*

**lemma** *not\_comp\_hd\_imp\_Var*: ¬  $\xi \leq_{hd} \zeta \implies \text{is\_Var } \zeta \vee \text{is\_Var } \xi$   
**using** *gt\_sym\_total* **by** (cases  $\zeta$ ; cases  $\xi$ ; auto simp: *comp\_hd\_def* *gt\_hd\_def*)

**end**

**end**

## 4 Infinite (Non-Well-Founded) Chains

**theory** *Infinite\_Chain*  
**imports** *Lambda\_Free\_Util*  
**begin**

This theory defines the concept of a minimal bad (or non-well-founded) infinite chain, as found in the term rewriting literature to prove the well-foundedness of syntactic term orders.

**context**

**fixes**  $p :: 'a \Rightarrow 'a \Rightarrow \text{bool}$   
**begin**

**definition**  $\text{inf\_chain} :: (\text{nat} \Rightarrow 'a) \Rightarrow \text{bool}$  **where**

$\text{inf\_chain } f \longleftrightarrow (\forall i. p (f i) (f (\text{Suc } i)))$

**lemma**  $\text{wfP\_iff\_no\_inf\_chain}: \text{wfP } (\lambda x y. p y x) \longleftrightarrow (\nexists f. \text{inf\_chain } f)$

**unfolding**  $\text{wfP\_def } \text{wf\_iff\_no\_infinite\_down\_chain } \text{inf\_chain\_def}$  **by**  $\text{simp}$

**lemma**  $\text{inf\_chain\_offset}: \text{inf\_chain } f \Longrightarrow \text{inf\_chain } (\lambda j. f (j + i))$

**unfolding**  $\text{inf\_chain\_def}$  **by**  $\text{simp}$

**definition**  $\text{bad} :: 'a \Rightarrow \text{bool}$  **where**

$\text{bad } x \longleftrightarrow (\exists f. \text{inf\_chain } f \wedge f 0 = x)$

**lemma**  $\text{inf\_chain\_bad}:$

**assumes**  $\text{bad\_f}: \text{inf\_chain } f$

**shows**  $\text{bad } (f i)$

**unfolding**  $\text{bad\_def}$  **by**  $(\text{rule } \text{exI}[\text{of } \_ \lambda j. f (j + i)]) (\text{simp } \text{add}: \text{inf\_chain\_offset}[\text{OF } \text{bad\_f}])$

**context**

**fixes**  $gt :: 'a \Rightarrow 'a \Rightarrow \text{bool}$   
**assumes**  $\text{wf}: \text{wf } \{(x, y). gt y x\}$   
**begin**

**primrec**  $\text{worst\_chain} :: \text{nat} \Rightarrow 'a$  **where**

$\text{worst\_chain } 0 = (\text{SOME } x. \text{bad } x \wedge (\forall y. \text{bad } y \longrightarrow \neg gt x y))$   
 $|\ \text{worst\_chain } (\text{Suc } i) = (\text{SOME } x. \text{bad } x \wedge p (\text{worst\_chain } i) x \wedge$   
 $(\forall y. \text{bad } y \wedge p (\text{worst\_chain } i) y \longrightarrow \neg gt x y))$

**declare**  $\text{worst\_chain.simps}[\text{simp } \text{del}]$

**context**

**fixes**  $x :: 'a$   
**assumes**  $x\_bad: \text{bad } x$   
**begin**

**lemma**

$\text{bad\_worst\_chain\_0}: \text{bad } (\text{worst\_chain } 0)$  **and**

$\text{min\_worst\_chain\_0}: \neg gt (\text{worst\_chain } 0) x$

**proof**  $-$

**obtain**  $y$  **where**  $\text{bad } y \wedge (\forall z. \text{bad } z \longrightarrow \neg gt y z)$

**using**  $\text{wf\_exists\_minimal}[\text{OF } \text{wf}, \text{of } \text{bad}, \text{OF } x\_bad]$  **by**  $\text{force}$

**hence**  $\text{bad } (\text{worst\_chain } 0) \wedge (\forall z. \text{bad } z \longrightarrow \neg gt (\text{worst\_chain } 0) z)$

**unfolding**  $\text{worst\_chain.simps}$  **by**  $(\text{rule } \text{someI})$

**thus**  $\text{bad } (\text{worst\_chain } 0)$  **and**  $\neg gt (\text{worst\_chain } 0) x$

**using**  $x\_bad$  **by**  $\text{blast+}$

**qed**

**lemma**

$\text{bad\_worst\_chain\_Suc}: \text{bad } (\text{worst\_chain } (\text{Suc } i))$  **and**

$\text{worst\_chain\_pred}: p (\text{worst\_chain } i) (\text{worst\_chain } (\text{Suc } i))$  **and**

$\text{min\_worst\_chain\_Suc}: p (\text{worst\_chain } i) x \Longrightarrow \neg gt (\text{worst\_chain } (\text{Suc } i)) x$

**proof**  $(\text{induct } i \text{ rule: less\_induct})$

**case**  $(\text{less } i)$

**have**  $\text{bad } (\text{worst\_chain } i)$

**proof**  $(\text{cases } i)$

**case**  $0$

**thus**  $?thesis$

```

    using bad_worst_chain_0 by simp
next
  case (Suc j)
  thus ?thesis
    using less(1) by blast
qed
then obtain fa where fa_bad: inf_chain fa and fa_0: fa 0 = worst_chain i
  unfolding bad_def by blast

have  $\exists s0. \text{bad } s0 \wedge p (\text{worst\_chain } i) s0$ 
proof (intro exI conjI)
  let ?y0 = fa (Suc 0)

  show bad ?y0
    unfolding bad_def by (auto intro: exI[of _  $\lambda i. \text{fa } (Suc i)$ ] inf_chain_offset[OF fa_bad])
  show p (worst_chain i) ?y0
    using fa_bad[unfolded inf_chain_def] fa_0 by metis
qed
then obtain y0 where y0: bad y0  $\wedge$  p (worst_chain i) y0
  by blast

obtain y1 where
  y1: bad y1  $\wedge$  p (worst_chain i) y1  $\wedge$  ( $\forall z. \text{bad } z \wedge p (\text{worst\_chain } i) z \longrightarrow \neg \text{gt } y1 z$ )
  using wf_exists_minimal[OF wf, of  $\lambda y. \text{bad } y \wedge p (\text{worst\_chain } i) y, OF y0$ ] by force

let ?y = worst_chain (Suc i)

have conj: bad ?y  $\wedge$  p (worst_chain i) ?y  $\wedge$  ( $\forall z. \text{bad } z \wedge p (\text{worst\_chain } i) z \longrightarrow \neg \text{gt } ?y z$ )
  unfolding worst_chain.simps using y1 by (rule someI)

show bad ?y
  by (rule conj[THEN conjunct1])
show p (worst_chain i) ?y
  by (rule conj[THEN conjunct2, THEN conjunct1])
show p (worst_chain i) x  $\implies \neg \text{gt } ?y x$ 
  using x_bad conj[THEN conjunct2, THEN conjunct2, rule_format] by meson
qed

lemma bad_worst_chain: bad (worst_chain i)
  by (cases i) (auto intro: bad_worst_chain_0 bad_worst_chain_Suc)

lemma worst_chain_bad: inf_chain worst_chain
  unfolding inf_chain_def using worst_chain_pred by metis

end

context
  fixes x :: 'a
  assumes
    x_bad: bad x and
    p_trans:  $\bigwedge z y x. p z y \implies p y x \implies p z x$ 
begin

lemma worst_chain_not_gt:  $\neg \text{gt } (\text{worst\_chain } i) (\text{worst\_chain } (Suc i))$  for i
proof (cases i)
  case 0
  show ?thesis
    unfolding 0 by (rule min_worst_chain_0[OF inf_chain_bad[OF worst_chain_bad[OF x_bad]]])
next
  case Suc
  show ?thesis
    unfolding Suc
    by (rule min_worst_chain_Suc[OF inf_chain_bad[OF worst_chain_bad[OF x_bad]]])
end

```

```

(rule p_trans[OF worst_chain_pred[OF x_bad] worst_chain_pred[OF x_bad]])
qed
end
end
end

lemma inf_chain_subset: inf_chain p f  $\implies$  p  $\leq$  q  $\implies$  inf_chain q f
  unfolding inf_chain_def by blast

hide-fact (open) bad_worst_chain_0 bad_worst_chain_Suc
end

```

## 5 Extension Orders

```

theory Extension_Orders
imports Lambda_Free_Util Infinite_Chain HOL-Cardinals.Wellorder_Extension
begin

```

This theory defines locales for categorizing extension orders used for orders on  $\lambda$ -free higher-order terms and defines variants of the lexicographic and multiset orders.

### 5.1 Locales

```

locale ext =
  fixes ext :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool
  assumes
    mono_strong: ( $\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y x \longrightarrow \text{gt}' y x$ )  $\implies$  ext gt ys xs  $\implies$  ext gt' ys xs and
    map: finite A  $\implies$  ys  $\in$  lists A  $\implies$  xs  $\in$  lists A  $\implies$  ( $\forall x \in A. \neg \text{gt } (f x) (f x)$ )  $\implies$ 
      ( $\forall z \in A. \forall y \in A. \forall x \in A. \text{gt } (f z) (f y) \longrightarrow \text{gt } (f y) (f x) \longrightarrow \text{gt } (f z) (f x)$ )  $\implies$ 
      ( $\forall y \in A. \forall x \in A. \text{gt } y x \longrightarrow \text{gt } (f y) (f x)$ )  $\implies$  ext gt ys xs  $\implies$  ext gt (map f ys) (map f xs)
begin

```

```

lemma mono[mono]: gt  $\leq$  gt'  $\implies$  ext gt  $\leq$  ext gt'
  using mono_strong by blast
end

```

```

locale ext_irrefl = ext +
  assumes irrefl: ( $\forall x \in \text{set } xs. \neg \text{gt } x x$ )  $\implies$   $\neg$  ext gt xs xs

```

```

locale ext_trans = ext +
  assumes trans: zs  $\in$  lists A  $\implies$  ys  $\in$  lists A  $\implies$  xs  $\in$  lists A  $\implies$ 
    ( $\forall z \in A. \forall y \in A. \forall x \in A. \text{gt } z y \longrightarrow \text{gt } y x \longrightarrow \text{gt } z x$ )  $\implies$  ext gt zs ys  $\implies$ 
    ext gt zs xs

```

```

locale ext_irrefl_before_trans = ext_irrefl +
  assumes trans_from_irrefl: finite A  $\implies$  zs  $\in$  lists A  $\implies$  ys  $\in$  lists A  $\implies$  xs  $\in$  lists A  $\implies$ 
    ( $\forall x \in A. \neg \text{gt } x x$ )  $\implies$  ( $\forall z \in A. \forall y \in A. \forall x \in A. \text{gt } z y \longrightarrow \text{gt } y x \longrightarrow \text{gt } z x$ )  $\implies$  ext gt zs ys  $\implies$ 
    ext gt ys xs  $\implies$  ext gt zs xs

```

```

locale ext_trans_before_irrefl = ext_trans +
  assumes irrefl_from_trans: ( $\forall z \in \text{set } xs. \forall y \in \text{set } xs. \forall x \in \text{set } xs. \text{gt } z y \longrightarrow \text{gt } y x \longrightarrow \text{gt } z x$ )  $\implies$ 
    ( $\forall x \in \text{set } xs. \neg \text{gt } x x$ )  $\implies$   $\neg$  ext gt xs xs

```

```

locale ext_irrefl_trans_strong = ext_irrefl +
  assumes trans_strong: ( $\forall z \in \text{set } zs. \forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } z y \longrightarrow \text{gt } y x \longrightarrow \text{gt } z x$ )  $\implies$ 
    ext gt zs ys  $\implies$  ext gt ys xs  $\implies$  ext gt zs xs

```

```

sublocale ext_irrefl_trans_strong < ext_irrefl_before_trans

```

```

by standard (erule irrefl, metis in_listsD trans_strong)

sublocale ext_irrefl_trans_strong < ext_trans
  by standard (metis in_listsD trans_strong)

sublocale ext_irrefl_trans_strong < ext_trans_before_irrefl
  by standard (rule irrefl)

locale ext_snoc = ext +
  assumes snoc: ext gt (xs @ [x]) xs

locale ext_compat_cons = ext +
  assumes compat_cons: ext gt ys xs  $\implies$  ext gt (x # ys) (x # xs)
begin

lemma compat_append_left: ext gt ys xs  $\implies$  ext gt (zs @ ys) (zs @ xs)
  by (induct zs) (auto intro: compat_cons)

end

locale ext_compat_snoc = ext +
  assumes compat_snoc: ext gt ys xs  $\implies$  ext gt (ys @ [x]) (xs @ [x])
begin

lemma compat_append_right: ext gt ys xs  $\implies$  ext gt (ys @ zs) (xs @ zs)
  by (induct zs arbitrary: xs ys rule: rev_induct)
  (auto intro: compat_snoc simp del: append_assoc simp: append_assoc[symmetric])

end

locale ext_compat_list = ext +
  assumes compat_list:  $y \neq x \implies$  gt y x  $\implies$  ext gt (xs @ y # xs') (xs @ x # xs')

locale ext_singleton = ext +
  assumes singleton:  $y \neq x \implies$  ext gt [y] [x]  $\longleftrightarrow$  gt y x

locale ext_compat_list_strong = ext_compat_cons + ext_compat_snoc + ext_singleton
begin

lemma compat_list:  $y \neq x \implies$  gt y x  $\implies$  ext gt (xs @ y # xs') (xs @ x # xs')
  using compat_append_left[of gt y # xs' x # xs' xs]
  compat_append_right[of gt, of [y] [x] xs'] singleton[of y x gt]
  by fastforce

end

sublocale ext_compat_list_strong < ext_compat_list
  by standard (fact compat_list)

locale ext_total = ext +
  assumes total:  $(\forall y \in B. \forall x \in A. gt y x \vee gt x y \vee y = x) \implies$   $ys \in lists B \implies xs \in lists A \implies$ 
  ext gt ys xs  $\vee$  ext gt xs ys  $\vee$  ys = xs

locale ext_wf = ext +
  assumes wf: wfP  $(\lambda x y. gt y x) \implies$  wfP  $(\lambda xs ys. ext gt ys xs)$ 

locale ext_hd_or_tl = ext +
  assumes hd_or_tl:  $(\forall z y x. gt z y \longrightarrow gt y x \longrightarrow gt z x) \implies$   $(\forall y x. gt y x \vee gt x y \vee y = x) \implies$ 
  length ys = length xs  $\implies$  ext gt (y # ys) (x # xs)  $\implies$  gt y x  $\vee$  ext gt ys xs

locale ext_wf_bounded = ext_irrefl_before_trans + ext_hd_or_tl
begin

```



```

context
  fixes gt :: 'a ⇒ 'a ⇒ bool
  assumes
    gt_irrefl:  $\bigwedge z. \neg gt\ z\ z$  and
    gt_trans:  $\bigwedge z\ y\ x. gt\ z\ y \implies gt\ y\ x \implies gt\ z\ x$  and
    gt_total:  $\bigwedge y\ x. gt\ y\ x \vee gt\ x\ y \vee y = x$  and
    gt_wf: wfP ( $\lambda x\ y. gt\ y\ x$ )
begin

lemma irrefl_gt:  $\neg ext\ gt\ xs\ xs$ 
  using irrefl_gt_irrefl by simp

lemma trans_gt:  $ext\ gt\ zs\ ys \implies ext\ gt\ ys\ xs \implies ext\ gt\ zs\ xs$ 
  by (rule trans_from_irrefl[of set zs  $\cup$  set ys  $\cup$  set xs zs ys xs gt])
  (auto intro: gt_trans simp: gt_irrefl)

lemma hd_or_tl_gt:  $length\ ys = length\ xs \implies ext\ gt\ (y \# ys)\ (x \# xs) \implies gt\ y\ x \vee ext\ gt\ ys\ xs$ 
  by (rule hd_or_tl) (auto intro: gt_trans simp: gt_total)

lemma wf_same_length_if_total: wfP ( $\lambda xs\ ys. length\ ys = n \wedge length\ xs = n \wedge ext\ gt\ ys\ xs$ )
proof (induct n)
  case 0
  thus ?case
    unfolding wfP_def wf_def using irrefl by auto
next
  case (Suc n)
  note ih = this(1)

  define gt_hd where  $\bigwedge ys\ xs. gt\_hd\ ys\ xs \longleftrightarrow gt\ (hd\ ys)\ (hd\ xs)$ 
  define gt_tl where  $\bigwedge ys\ xs. gt\_tl\ ys\ xs \longleftrightarrow ext\ gt\ (tl\ ys)\ (tl\ xs)$ 

  have hd_tl:  $gt\_hd\ ys\ xs \vee gt\_tl\ ys\ xs$ 
    if len_ys:  $length\ ys = Suc\ n$  and len_xs:  $length\ xs = Suc\ n$  and ys_gt_xs:  $ext\ gt\ ys\ xs$ 
    for n ys xs
    using len_ys len_xs ys_gt_xs unfolding gt_hd_def gt_tl_def
    by (cases xs; cases ys) (auto simp: hd_or_tl_gt)

  show ?case
    unfolding wfP_iff_no_inf_chain
  proof (intro notI)
    let ?gtsn =  $\lambda ys\ xs. length\ ys = n \wedge length\ xs = n \wedge ext\ gt\ ys\ xs$ 
    let ?gtsSn =  $\lambda ys\ xs. length\ ys = Suc\ n \wedge length\ xs = Suc\ n \wedge ext\ gt\ ys\ xs$ 
    let ?gttlSn =  $\lambda ys\ xs. length\ ys = Suc\ n \wedge length\ xs = Suc\ n \wedge gt\_tl\ ys\ xs$ 

    assume  $\exists f. inf\_chain\ ?gtsSn\ f$ 
    then obtain xs where xs_bad: bad ?gtsSn xs
      unfolding inf_chain_def bad_def by blast

    let ?ff = worst_chain ?gtsSn gt_hd

    have wf_hd: wf {(xs, ys). gt_hd ys xs}
      unfolding gt_hd_def by (rule wfP_app[OF gt_wf, of hd, unfolded wfP_def])

    have inf_chain ?gtsSn ?ff
      by (rule worst_chain_bad[OF wf_hd xs_bad])
    moreover have  $\neg gt\_hd\ (?ff\ i)\ (?ff\ (Suc\ i))$  for i
      by (rule worst_chain_not_gt[OF wf_hd xs_bad]) (blast intro: trans_gt)
    ultimately have tl_bad: inf_chain ?gttlSn ?ff
      unfolding inf_chain_def using hd_tl by blast

    have  $\neg inf\_chain\ ?gtsn\ (tl \circ ?ff)$ 
      using wfP_iff_no_inf_chain[THEN iffD1, OF ih] by blast
    hence tl_good:  $\neg inf\_chain\ ?gttlSn\ ?ff$ 

```

```

unfolding inf_chain_def gt_tl_def by force

show False
  using tl_bad tl_good by sat
qed
qed

lemma wf_bounded_if_total: wfP ( $\lambda x s y s. \text{length } y s \leq n \wedge \text{length } x s \leq n \wedge \text{ext } g t y s x s$ )
  unfolding wfP_iff_no_inf_chain
proof (intro notI, induct n rule: less_induct)
  case (less n)
  note ih = this(1) and ex_bad = this(2)

  let ?gtsle =  $\lambda y s x s. \text{length } y s \leq n \wedge \text{length } x s \leq n \wedge \text{ext } g t y s x s$ 

  obtain xs where xs_bad: bad ?gtsle xs
    using ex_bad unfolding inf_chain_def bad_def by blast

  let ?ff = worst_chain ?gtsle ( $\lambda y s x s. \text{length } y s > \text{length } x s$ )

  note wf_len = wf_app[OF wellorder_class.wf, of length, simplified]

  have ff_bad: inf_chain ?gtsle ?ff
    by (rule worst_chain_bad[OF wf_len xs_bad])
  have ffi_bad:  $\bigwedge i. \text{bad } ?gtsle (\text{?ff } i)$ 
    by (rule inf_chain_bad[OF ff_bad])

  have len_le_n:  $\bigwedge i. \text{length } (\text{?ff } i) \leq n$ 
    using worst_chain_pred[OF wf_len xs_bad] by simp
  have len_le_Suc:  $\bigwedge i. \text{length } (\text{?ff } i) \leq \text{length } (\text{?ff } (\text{Suc } i))$ 
    using worst_chain_not_gt[OF wf_len xs_bad] not_le_imp_less by (blast intro: trans_gt)

  show False
proof (cases  $\exists k. \text{length } (\text{?ff } k) = n$ )
  case False
  hence len_lt_n:  $\bigwedge i. \text{length } (\text{?ff } i) < n$ 
    using len_le_n by (blast intro: le_neq_implies_less)
  hence nm1_le:  $n - 1 < n$ 
    by fastforce

  let ?gtslt =  $\lambda y s x s. \text{length } y s \leq n - 1 \wedge \text{length } x s \leq n - 1 \wedge \text{ext } g t y s x s$ 

  have inf_chain ?gtslt ?ff
    using ff_bad len_lt_n unfolding inf_chain_def
    by (metis (no_types, lifting) Suc_diff_1 le_antisym nat_neq_iff not_less0 not_less_eq_eq)
  thus False
    using ih[OF nm1_le] by blast
next
  case True
  then obtain k where len_eq_n:  $\text{length } (\text{?ff } k) = n$ 
    by blast

  let ?gtssl =  $\lambda y s x s. \text{length } y s = n \wedge \text{length } x s = n \wedge \text{ext } g t y s x s$ 

  have len_eq_n:  $\text{length } (\text{?ff } (i + k)) = n$  for i
    by (induct i) (simp add: len_eq_n,
      metis (lifting) len_le_n len_le_Suc add_Suc dual_order.antisym)

  have inf_chain ?gtsle ( $\lambda i. \text{?ff } (i + k)$ )
    by (rule inf_chain_offset[OF ff_bad])
  hence inf_chain ?gtssl ( $\lambda i. \text{?ff } (i + k)$ )
    unfolding inf_chain_def using len_eq_n by presburger
  hence  $\neg \text{wfP } (\lambda x s y s. \text{?gtssl } y s x s)$ 

```

```

    using wfP_iff_no_inf_chain by blast
  thus False
    using wf_same_length_if_total[of n] by sat
qed
qed

end

context
  fixes gt :: 'a ⇒ 'a ⇒ bool
  assumes
    gt_irrefl:  $\bigwedge z. \neg gt\ z\ z$  and
    gt_wf: wfP ( $\lambda x\ y. gt\ y\ x$ )
begin

lemma wf_bounded: wfP ( $\lambda xs\ ys. length\ ys \leq n \wedge length\ xs \leq n \wedge ext\ gt\ ys\ xs$ )
proof -
  obtain Ge' where
    gt_sub_Ge':  $\{(x, y). gt\ y\ x\} \subseteq Ge'$  and
    Ge'_wo: Well_order Ge' and
    Ge'_fld: Field Ge' = UNIV
    using total_well_order_extension[OF gt_wf[unfolded wfP_def]] by blast

  define gt' where  $\bigwedge y\ x. gt'\ y\ x \longleftrightarrow y \neq x \wedge (x, y) \in Ge'$ 

  have gt_imp_gt':  $gt \leq gt'$ 
    by (auto simp: gt'_def gt_irrefl intro: gt_sub_Ge'[THEN set_mp])

  have gt'_irrefl:  $\bigwedge z. \neg gt'\ z\ z$ 
    unfolding gt'_def by simp

  have gt'_trans:  $\bigwedge z\ y\ x. gt'\ z\ y \implies gt'\ y\ x \implies gt'\ z\ x$ 
    using Ge'_wo
    unfolding gt'_def well_order_on_def linear_order_on_def partial_order_on_def preorder_on_def
    trans_def antisym_def
    by blast

  have wf  $\{(x, y). (x, y) \in Ge' \wedge x \neq y\}$ 
    by (rule Ge'_wo[unfolded well_order_on_def set_diff_eq
      case_prod_eta[symmetric, of  $\lambda xy. xy \in Ge' \wedge xy \notin Id$ ] pair_in_Id_conv, THEN conjunct2])
  moreover have  $\bigwedge y\ x. (x, y) \in Ge' \wedge x \neq y \longleftrightarrow y \neq x \wedge (x, y) \in Ge'$ 
    by auto
  ultimately have gt'_wf: wfP ( $\lambda x\ y. gt'\ y\ x$ )
    unfolding wfP_def gt'_def by simp

  have gt'_total:  $\bigwedge x\ y. gt'\ y\ x \vee gt'\ x\ y \vee y = x$ 
    using Ge'_wo unfolding gt'_def well_order_on_def linear_order_on_def total_on_def Ge'_fld
    by blast

  have wfP ( $\lambda xs\ ys. length\ ys \leq n \wedge length\ xs \leq n \wedge ext\ gt'\ ys\ xs$ )
    using wf_bounded_if_total gt'_total gt'_irrefl gt'_trans gt'_wf by blast
  thus ?thesis
    by (rule wfP_subset) (auto intro: mono[OF gt_imp_gt', THEN predicate2D])
qed

end

end

```

## 5.2 Lexicographic Extension

```

inductive lexext :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool for gt where
  lexext_Nil: lexext gt (y # ys) []
| lexext_Cons:  $gt\ y\ x \implies lexext\ gt\ (y\ \# \ ys)\ (x\ \# \ xs)$ 

```

| *lexert\_Cons\_eq*:  $\text{lexert gt ys xs} \implies \text{lexert gt (x \# ys) (x \# xs)}$

**lemma** *lexert\_simps[simp]*:  
 $\text{lexert gt ys []} \longleftrightarrow \text{ys} \neq \text{[]}$   
 $\neg \text{lexert gt [] xs}$   
 $\text{lexert gt (y \# ys) (x \# xs)} \longleftrightarrow \text{gt y x} \vee \text{x} = \text{y} \wedge \text{lexert gt ys xs}$

**proof**

**show**  $\text{lexert gt ys []} \implies (\text{ys} \neq \text{[]})$   
**by** (*metis lexert.cases list.distinct(1)*)

**next**

**show**  $\text{ys} \neq \text{[]} \implies \text{lexert gt ys []}$   
**by** (*metis lexert\_Nil list.exhaust*)

**next**

**show**  $\neg \text{lexert gt [] xs}$   
**using** *lexert.cases* **by** *auto*

**next**

**show**  $\text{lexert gt (y \# ys) (x \# xs)} = (\text{gt y x} \vee \text{x} = \text{y} \wedge \text{lexert gt ys xs})$

**proof** -

**have** *fudd*:  $\text{lexert gt (y \# ys) (x \# xs)} \longrightarrow \text{gt y x} \vee \text{x} = \text{y} \wedge \text{lexert gt ys xs}$

**proof**

**assume**  $\text{lexert gt (y \# ys) (x \# xs)}$

**thus**  $\text{gt y x} \vee \text{x} = \text{y} \wedge \text{lexert gt ys xs}$

**using** *lexert.cases* **by** *blast*

**qed**

**have** *backd*:  $\text{gt y x} \vee \text{x} = \text{y} \wedge \text{lexert gt ys xs} \longrightarrow \text{lexert gt (y \# ys) (x \# xs)}$

**by** (*simp add: lexert\_Cons lexert\_Cons\_eq*)

**show**  $\text{lexert gt (y \# ys) (x \# xs)} = (\text{gt y x} \vee \text{x} = \text{y} \wedge \text{lexert gt ys xs})$

**using** *fudd backd* **by** *blast*

**qed**

**qed**

**lemma** *lexert\_mono\_strong*:

**assumes**

$\forall y \in \text{set ys.} \forall x \in \text{set xs.} \text{gt y x} \longrightarrow \text{gt}' y x$  **and**

$\text{lexert gt ys xs}$

**shows**  $\text{lexert gt}' \text{ys xs}$

**using** *assms* **by** (*induct ys xs rule: list\_induct2'*) *auto*

**lemma** *lexert\_map\_strong*:

$(\forall y \in \text{set ys.} \forall x \in \text{set xs.} \text{gt y x} \longrightarrow \text{gt (f y) (f x)}) \implies \text{lexert gt ys xs} \implies$

$\text{lexert gt (map f ys) (map f xs)}$

**by** (*induct ys xs rule: list\_induct2'*) *auto*

**lemma** *lexert\_irrefl*:

**assumes**  $\forall x \in \text{set xs.} \neg \text{gt x x}$

**shows**  $\neg \text{lexert gt xs xs}$

**using** *assms* **by** (*induct xs*) *auto*

**lemma** *lexert\_trans\_strong*:

**assumes**

$\forall z \in \text{set zs.} \forall y \in \text{set ys.} \forall x \in \text{set xs.} \text{gt z y} \longrightarrow \text{gt y x} \longrightarrow \text{gt z x}$  **and**

$\text{lexert gt zs ys}$  **and**  $\text{lexert gt ys xs}$

**shows**  $\text{lexert gt zs xs}$

**using** *assms*

**proof** (*induct zs arbitrary: ys xs*)

**case** (*Cons z zs*)

**note** *zs\_trans* = *this(1)*

**show** *?case*

**using** *Cons(2-4)*

**proof** (*induct ys arbitrary: xs rule: list.induct*)

**case** (*Cons y ys*)

**note** *ys\_trans* = *this(1)* **and** *gt\_trans* = *this(2)* **and** *zys\_gt\_yys* = *this(3)* **and** *yys\_gt\_xs* = *this(4)*

**show** *?case*

```

proof (cases xs)
  case xs: (Cons x xs)
  thus ?thesis
proof (unfold xs)
  note yys_gt_xxs = yys_gt_xs[unfolded xs]
  note gt_trans = gt_trans[unfolded xs]

  let ?case = lext gt (z # zs) (x # xs)

  {
    assume gt z y and gt y x
    hence ?case
    using gt_trans by simp
  }
moreover
  {
    assume gt z y and x = y
    hence ?case
    by simp
  }
moreover
  {
    assume y = z and gt y x
    hence ?case
    by simp
  }
moreover
  {
    assume
      y_eq_z: y = z and
      zs_gt_ys: lext gt zs ys and
      x_eq_y: x = y and
      ys_gt_xs: lext gt ys xs

    have lext gt zs xs
      by (rule zs_trans[OF zs_gt_ys ys_gt_xs]) (meson gt_trans[simplified])
    hence ?case
      by (simp add: x_eq_y y_eq_z)
  }
  ultimately show ?case
  using zzs_gt_yys yys_gt_xxs by force
qed
qed auto
qed auto
qed auto

lemma lext_snoc: lext gt (xs @ [x]) xs
  by (induct xs) auto

lemmas lext_compat_cons = lext_Cons_eq

lemma lext_compat_snoc_if_same_length:
  assumes length ys = length xs and lext gt ys xs
  shows lext gt (ys @ [x]) (xs @ [x])
  using assms(2,1) by (induct rule: lext.induct) auto

lemma lext_compat_list: gt y x  $\implies$  lext gt (xs @ y # xs') (xs @ x # xs')
  by (induct xs) auto

lemma lext_singleton: lext gt [y] [x]  $\longleftrightarrow$  gt y x
  by simp

lemma lext_total:  $(\forall y \in B. \forall x \in A. gt y x \vee gt x y \vee y = x) \implies ys \in lists B \implies xs \in lists A \implies$ 

```

*lexext gt ys xs*  $\vee$  *lexext gt xs ys*  $\vee$  *ys* = *xs*  
**by** (*induct ys xs rule: list\_induct2'*) *auto*

**lemma** *lexext\_hd\_or\_tl*: *lexext gt (y # ys) (x # xs)  $\implies$  gt y x  $\vee$  lexext gt ys xs*  
**by** *auto*

**interpretation** *lexext*: *ext lexext*  
**by** *standard (fact lexext\_mono\_strong, rule lexext\_map\_strong, metis in\_listsD)*

**interpretation** *lexext*: *ext\_irrefl\_trans\_strong lexext*  
**by** *standard (fact lexext\_irrefl, fact lexext\_trans\_strong)*

**interpretation** *lexext*: *ext\_snoc lexext*  
**by** *standard (fact lexext\_snoc)*

**interpretation** *lexext*: *ext\_compat\_cons lexext*  
**by** *standard (fact lexext\_compat\_cons)*

**interpretation** *lexext*: *ext\_compat\_list lexext*  
**by** *standard (rule lexext\_compat\_list)*

**interpretation** *lexext*: *ext\_singleton lexext*  
**by** *standard (rule lexext\_singleton)*

**interpretation** *lexext*: *ext\_total lexext*  
**by** *standard (fact lexext\_total)*

**interpretation** *lexext*: *ext\_hd\_or\_tl lexext*  
**by** *standard (rule lexext\_hd\_or\_tl)*

**interpretation** *lexext*: *ext\_wf\_bounded lexext*  
**by** *standard*

### 5.3 Reverse (Right-to-Left) Lexicographic Extension

**abbreviation** *lexext\_rev* :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool **where**  
*lexext\_rev gt ys xs*  $\equiv$  *lexext gt (rev ys) (rev xs)*

**lemma** *lexext\_rev\_simps[simp]*:  
*lexext\_rev gt ys []  $\longleftrightarrow$  ys  $\neq$  []*  
 $\neg$  *lexext\_rev gt [] xs*  
*lexext\_rev gt (ys @ [y]) (xs @ [x])  $\longleftrightarrow$  gt y x  $\vee$  x = y  $\wedge$  lexext\_rev gt ys xs*  
**by** *simp+*

**lemma** *lexext\_rev\_cons\_cons*:  
**assumes** *length ys = length xs*  
**shows** *lexext\_rev gt (y # ys) (x # xs)  $\longleftrightarrow$  lexext\_rev gt ys xs  $\vee$  ys = xs  $\wedge$  gt y x*  
**using** *assms*

**proof** (*induct arbitrary: y x rule: rev\_induct2*)

**case** *Nil*  
**thus** *?case*  
**by** *simp*

**next**

**case** (*snoc y' ys x' xs*)  
**show** *?case*  
**using** *snoc(2)* **by** *auto*

**qed**

**lemma** *lexext\_rev\_mono\_strong*:  
**assumes**  
 $\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y \ x \longrightarrow \text{gt}' \ y \ x$  **and**  
*lexext\_rev gt ys xs*  
**shows** *lexext\_rev gt' ys xs*  
**using** *assms* **by** (*simp add: lexext\_mono\_strong*)

**lemma** *lexext\_rev\_map\_strong*:  
 $(\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y \ x \longrightarrow \text{gt } (f \ y) \ (f \ x)) \Longrightarrow \text{lexext\_rev } \text{gt } ys \ xs \Longrightarrow$   
 $\text{lexext\_rev } \text{gt } (\text{map } f \ ys) \ (\text{map } f \ xs)$   
**by** (*simp add: lexext\_map\_strong rev\_map*)

**lemma** *lexext\_rev\_irrefl*:  
**assumes**  $\forall x \in \text{set } xs. \neg \text{gt } x \ x$   
**shows**  $\neg \text{lexext\_rev } \text{gt } xs \ xs$   
**using** *assms* **by** (*simp add: lexext\_irrefl*)

**lemma** *lexext\_rev\_trans\_strong*:  
**assumes**  
 $\forall z \in \text{set } zs. \forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } z \ y \longrightarrow \text{gt } y \ x \longrightarrow \text{gt } z \ x$  **and**  
 $\text{lexext\_rev } \text{gt } zs \ ys$  **and**  $\text{lexext\_rev } \text{gt } ys \ xs$   
**shows**  $\text{lexext\_rev } \text{gt } zs \ xs$   
**using** *assms*(1) *lexext\_trans\_strong[OF \_ assms(2,3), unfolded set\_rev]* **by** *sat*

**lemma** *lexext\_rev\_compat\_cons\_if\_same\_length*:  
**assumes**  $\text{length } ys = \text{length } xs$  **and**  $\text{lexext\_rev } \text{gt } ys \ xs$   
**shows**  $\text{lexext\_rev } \text{gt } (x \ \# \ ys) \ (x \ \# \ xs)$   
**using** *assms* **by** (*simp add: lexext\_compat\_snoc\_if\_same\_length*)

**lemma** *lexext\_rev\_compat\_snoc*:  $\text{lexext\_rev } \text{gt } ys \ xs \Longrightarrow \text{lexext\_rev } \text{gt } (ys \ @ \ [x]) \ (xs \ @ \ [x])$   
**by** (*simp add: lexext\_compat\_cons*)

**lemma** *lexext\_rev\_compat\_list*:  $\text{gt } y \ x \Longrightarrow \text{lexext\_rev } \text{gt } (xs \ @ \ y \ \# \ xs') \ (xs \ @ \ x \ \# \ xs')$   
**by** (*induct xs' rule: rev\_induct*) *auto*

**lemma** *lexext\_rev\_singleton*:  $\text{lexext\_rev } \text{gt } [y] \ [x] \longleftrightarrow \text{gt } y \ x$   
**by** *simp*

**lemma** *lexext\_rev\_total*:  
 $(\forall y \in B. \forall x \in A. \text{gt } y \ x \vee \text{gt } x \ y \vee y = x) \Longrightarrow ys \in \text{lists } B \Longrightarrow xs \in \text{lists } A \Longrightarrow$   
 $\text{lexext\_rev } \text{gt } ys \ xs \vee \text{lexext\_rev } \text{gt } xs \ ys \vee ys = xs$   
**by** (*rule lexext\_total[of \_ \_ \_ rev ys rev xs, simplified]*)

**lemma** *lexext\_rev\_hd\_or\_tl*:  
**assumes**  
 $\text{length } ys = \text{length } xs$  **and**  
 $\text{lexext\_rev } \text{gt } (y \ \# \ ys) \ (x \ \# \ xs)$   
**shows**  $\text{gt } y \ x \vee \text{lexext\_rev } \text{gt } ys \ xs$   
**using** *assms* *lexext\_rev\_cons\_cons* **by** *fastforce*

**interpretation** *lexext\_rev*: *ext lexext\_rev*  
**by** *standard* (*fact lexext\_rev\_mono\_strong, rule lexext\_rev\_map\_strong, metis in\_listsD*)

**interpretation** *lexext\_rev*: *ext\_irrefl\_trans\_strong lexext\_rev*  
**by** *standard* (*fact lexext\_rev\_irrefl, fact lexext\_rev\_trans\_strong*)

**interpretation** *lexext\_rev*: *ext\_compat\_snoc lexext\_rev*  
**by** *standard* (*fact lexext\_rev\_compat\_snoc*)

**interpretation** *lexext\_rev*: *ext\_compat\_list lexext\_rev*  
**by** *standard* (*rule lexext\_rev\_compat\_list*)

**interpretation** *lexext\_rev*: *ext\_singleton lexext\_rev*  
**by** *standard* (*rule lexext\_rev\_singleton*)

**interpretation** *lexext\_rev*: *ext\_total lexext\_rev*  
**by** *standard* (*fact lexext\_rev\_total*)

**interpretation** *lexext\_rev*: *ext\_hd\_or\_tl lexext\_rev*

by standard (rule lenext\_rev\_hd\_or\_tl)

**interpretation** lenext\_rev: ext\_wf\_bounded lenext\_rev  
by standard

## 5.4 Generic Length Extension

**definition** lenext :: ('a list  $\Rightarrow$  'a list  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool **where**  
lenext gts ys xs  $\longleftrightarrow$  length ys > length xs  $\vee$  length ys = length xs  $\wedge$  gts ys xs

**lemma**

lenext\_mono\_strong: (gts ys xs  $\Longrightarrow$  gts' ys xs)  $\Longrightarrow$  lenext gts ys xs  $\Longrightarrow$  lenext gts' ys xs **and**  
lenext\_map\_strong: (length ys = length xs  $\Longrightarrow$  gts ys xs  $\Longrightarrow$  gts (map f ys) (map f xs))  $\Longrightarrow$   
lenext gts ys xs  $\Longrightarrow$  lenext gts (map f ys) (map f xs) **and**  
lenext\_irrefl:  $\neg$  gts xs xs  $\Longrightarrow$   $\neg$  lenext gts xs xs **and**  
lenext\_trans: (gts zs ys  $\Longrightarrow$  gts ys xs  $\Longrightarrow$  gts zs xs)  $\Longrightarrow$  lenext gts zs ys  $\Longrightarrow$  lenext gts ys xs  $\Longrightarrow$   
lenext gts zs xs **and**  
lenext\_snoc: lenext gts (xs @ [x]) xs **and**  
lenext\_compat\_cons: (length ys = length xs  $\Longrightarrow$  gts ys xs  $\Longrightarrow$  gts (x # ys) (x # xs))  $\Longrightarrow$   
lenext gts ys xs  $\Longrightarrow$  lenext gts (x # ys) (x # xs) **and**  
lenext\_compat\_snoc: (length ys = length xs  $\Longrightarrow$  gts ys xs  $\Longrightarrow$  gts (ys @ [x]) (xs @ [x]))  $\Longrightarrow$   
lenext gts ys xs  $\Longrightarrow$  lenext gts (ys @ [x]) (xs @ [x]) **and**  
lenext\_compat\_list: gts (xs @ y # xs') (xs @ x # xs')  $\Longrightarrow$   
lenext gts (xs @ y # xs') (xs @ x # xs') **and**  
lenext\_singleton: lenext gts [y] [x]  $\longleftrightarrow$  gts [y] [x] **and**  
lenext\_total: (gts ys xs  $\vee$  gts xs ys  $\vee$  ys = xs)  $\Longrightarrow$   
lenext gts ys xs  $\vee$  lenext gts xs ys  $\vee$  ys = xs **and**  
lenext\_hd\_or\_tl: (length ys = length xs  $\Longrightarrow$  gts (y # ys) (x # xs))  $\Longrightarrow$  gt y x  $\vee$  gts ys xs  $\Longrightarrow$   
lenext gts (y # ys) (x # xs)  $\Longrightarrow$  gt y x  $\vee$  lenext gts ys xs  
**unfolding** lenext\_def by auto

## 5.5 Length-Lexicographic Extension

**abbreviation** len\_lexext :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool **where**  
len\_lexext gt  $\equiv$  lenext (lexext gt)

**lemma** len\_lexext\_mono\_strong:

( $\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y \ x \longrightarrow \text{gt}' \ y \ x$ )  $\Longrightarrow$  len\_lexext gt ys xs  $\Longrightarrow$  len\_lexext gt' ys xs  
by (rule lenext\_mono\_strong[OF lexext\_mono\_strong])

**lemma** len\_lexext\_map\_strong:

( $\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y \ x \longrightarrow \text{gt } (f \ y) \ (f \ x)$ )  $\Longrightarrow$  len\_lexext gt ys xs  $\Longrightarrow$   
len\_lexext gt (map f ys) (map f xs)  
by (rule lenext\_map\_strong) (metis lexext\_map\_strong)

**lemma** len\_lexext\_irrefl: ( $\forall x \in \text{set } xs. \neg \text{gt } x \ x$ )  $\Longrightarrow$   $\neg$  len\_lexext gt xs xs

by (rule lenext\_irrefl[OF lexext\_irrefl])

**lemma** len\_lexext\_trans\_strong:

( $\forall z \in \text{set } zs. \forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } z \ y \longrightarrow \text{gt } y \ x \longrightarrow \text{gt } z \ x$ )  $\Longrightarrow$  len\_lexext gt zs ys  $\Longrightarrow$   
len\_lexext gt ys xs  $\Longrightarrow$  len\_lexext gt zs xs  
by (rule lenext\_trans[OF lexext\_trans\_strong])

**lemma** len\_lexext\_snoc: len\_lexext gt (xs @ [x]) xs

by (rule lenext\_snoc)

**lemma** len\_lexext\_compat\_cons: len\_lexext gt ys xs  $\Longrightarrow$  len\_lexext gt (x # ys) (x # xs)

by (intro lenext\_compat\_cons lexext\_compat\_cons)

**lemma** len\_lexext\_compat\_snoc: len\_lexext gt ys xs  $\Longrightarrow$  len\_lexext gt (ys @ [x]) (xs @ [x])

by (intro lenext\_compat\_snoc lexext\_compat\_snoc\_if\_same\_length)

**lemma** len\_lexext\_compat\_list: gt y x  $\Longrightarrow$  len\_lexext gt (xs @ y # xs') (xs @ x # xs')

by (intro lenext\_compat\_list lexext\_compat\_list)



**lemma** *len\_lexext\_singleton[simp]*:  $\text{len\_lexext } \text{gt } [y] [x] \longleftrightarrow \text{gt } y \ x$   
**by** (*simp only: lenext\_singleton lexext\_singleton*)

**lemma** *len\_lexext\_total*:  $(\forall y \in B. \forall x \in A. \text{gt } y \ x \vee \text{gt } x \ y \vee y = x) \implies ys \in \text{lists } B \implies xs \in \text{lists } A \implies$   
 $\text{len\_lexext } \text{gt } ys \ xs \vee \text{len\_lexext } \text{gt } xs \ ys \vee ys = xs$   
**by** (*rule lenext\_total[OF lexext\_total]*)

**lemma** *len\_lexext\_iff\_lenlex*:  $\text{len\_lexext } \text{gt } ys \ xs \longleftrightarrow (xs, ys) \in \text{lenlex } \{(x, y). \text{gt } y \ x\}$   
**proof** –  
{  
  **assume** *length xs = length ys*  
  **hence**  $\text{lexext } \text{gt } ys \ xs \longleftrightarrow (xs, ys) \in \text{lex } \{(x, y). \text{gt } y \ x\}$   
  **by** (*induct xs ys rule: list\_induct2*) *auto*  
}  
**thus** *?thesis*  
  **unfolding** *lenext\_def lenlex\_conv* **by** *auto*  
**qed**

**lemma** *len\_lexext\_wf*:  $\text{wfP } (\lambda x \ y. \text{gt } y \ x) \implies \text{wfP } (\lambda xs \ ys. \text{len\_lexext } \text{gt } ys \ xs)$   
**unfolding** *wfP\_def len\_lexext\_iff\_lenlex* **by** (*simp add: wf\_lenlex*)

**lemma** *len\_lexext\_hd\_or\_tl*:  $\text{len\_lexext } \text{gt } (y \ \# \ ys) \ (x \ \# \ xs) \implies \text{gt } y \ x \vee \text{len\_lexext } \text{gt } ys \ xs$   
**using** *lenext\_hd\_or\_tl lexext\_hd\_or\_tl* **by** *metis*

**interpretation** *len\_lexext*: *ext len\_lexext*  
**by** *standard (fact len\_lexext\_mono\_strong, rule len\_lexext\_map\_strong, metis in\_listsD)*

**interpretation** *len\_lexext*: *ext\_irrefl\_trans\_strong len\_lexext*  
**by** *standard (fact len\_lexext\_irrefl, fact len\_lexext\_trans\_strong)*

**interpretation** *len\_lexext*: *ext\_snoc len\_lexext*  
**by** *standard (fact len\_lexext\_snoc)*

**interpretation** *len\_lexext*: *ext\_compat\_cons len\_lexext*  
**by** *standard (fact len\_lexext\_compat\_cons)*

**interpretation** *len\_lexext*: *ext\_compat\_snoc len\_lexext*  
**by** *standard (fact len\_lexext\_compat\_snoc)*

**interpretation** *len\_lexext*: *ext\_compat\_list len\_lexext*  
**by** *standard (rule len\_lexext\_compat\_list)*

**interpretation** *len\_lexext*: *ext\_singleton len\_lexext*  
**by** *standard (rule len\_lexext\_singleton)*

**interpretation** *len\_lexext*: *ext\_total len\_lexext*  
**by** *standard (fact len\_lexext\_total)*

**interpretation** *len\_lexext*: *ext\_wf len\_lexext*  
**by** *standard (fact len\_lexext\_wf)*

**interpretation** *len\_lexext*: *ext\_hd\_or\_tl len\_lexext*  
**by** *standard (rule len\_lexext\_hd\_or\_tl)*

**interpretation** *len\_lexext*: *ext\_wf\_bounded len\_lexext*  
**by** *standard*

## 5.6 Reverse (Right-to-Left) Length-Lexicographic Extension

**abbreviation** *len\_lexext\_rev* ::  $(\ 'a \Rightarrow \ 'a \Rightarrow \ \text{bool}) \Rightarrow \ 'a \ \text{list} \Rightarrow \ 'a \ \text{list} \Rightarrow \ \text{bool}$  **where**  
 $\text{len\_lexext\_rev } \text{gt} \equiv \text{lenext } (\text{lexext\_rev } \text{gt})$

**lemma** *len\_lexext\_rev\_mono\_strong*:

$(\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y \ x \longrightarrow \text{gt}' \ y \ x) \Longrightarrow \text{len\_lexext\_rev } \text{gt } ys \ xs \Longrightarrow \text{len\_lexext\_rev } \text{gt}' \ ys \ xs$   
**by** (rule `lenext_mono_strong`) (rule `lexext_rev_mono_strong`)

**lemma** `len_lexext_rev_map_strong`:

$(\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y \ x \longrightarrow \text{gt } (f \ y) \ (f \ x)) \Longrightarrow \text{len\_lexext\_rev } \text{gt } ys \ xs \Longrightarrow$   
 $\text{len\_lexext\_rev } \text{gt } (\text{map } f \ ys) \ (\text{map } f \ xs)$   
**by** (rule `lenext_map_strong`) (rule `lexext_rev_map_strong`)

**lemma** `len_lexext_rev_irrefl`:  $(\forall x \in \text{set } xs. \neg \text{gt } x \ x) \Longrightarrow \neg \text{len\_lexext\_rev } \text{gt } xs \ xs$

**by** (rule `lenext_irrefl`) (rule `lexext_rev_irrefl`)

**lemma** `len_lexext_rev_trans_strong`:

$(\forall z \in \text{set } zs. \forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } z \ y \longrightarrow \text{gt } y \ x \longrightarrow \text{gt } z \ x) \Longrightarrow \text{len\_lexext\_rev } \text{gt } zs \ ys \Longrightarrow$   
 $\text{len\_lexext\_rev } \text{gt } ys \ xs \Longrightarrow \text{len\_lexext\_rev } \text{gt } zs \ xs$   
**by** (rule `lenext_trans`) (rule `lexext_rev_trans_strong`)

**lemma** `len_lexext_rev_snoc`:  $\text{len\_lexext\_rev } \text{gt } (xs \ @ \ [x]) \ xs$

**by** (rule `lenext_snoc`)

**lemma** `len_lexext_rev_compat_cons`:  $\text{len\_lexext\_rev } \text{gt } ys \ xs \Longrightarrow \text{len\_lexext\_rev } \text{gt } (x \ # \ ys) \ (x \ # \ xs)$

**by** (intro `lenext_compat_cons` `lexext_rev_compat_cons_if_same_length`)

**lemma** `len_lexext_rev_compat_snoc`:  $\text{len\_lexext\_rev } \text{gt } ys \ xs \Longrightarrow \text{len\_lexext\_rev } \text{gt } (ys \ @ \ [x]) \ (xs \ @ \ [x])$

**by** (intro `lenext_compat_snoc` `lexext_rev_compat_snoc`)

**lemma** `len_lexext_rev_compat_list`:  $\text{gt } y \ x \Longrightarrow \text{len\_lexext\_rev } \text{gt } (xs \ @ \ y \ # \ xs') \ (xs \ @ \ x \ # \ xs')$

**by** (intro `lenext_compat_list` `lexext_rev_compat_list`)

**lemma** `len_lexext_rev_singleton[simp]`:  $\text{len\_lexext\_rev } \text{gt } [y] \ [x] \longleftrightarrow \text{gt } y \ x$

**by** (simp only: `lenext_singleton` `lexext_rev_singleton`)

**lemma** `len_lexext_rev_total`:  $(\forall y \in B. \forall x \in A. \text{gt } y \ x \vee \text{gt } x \ y \vee y = x) \Longrightarrow ys \in \text{lists } B \Longrightarrow$

$xs \in \text{lists } A \Longrightarrow \text{len\_lexext\_rev } \text{gt } ys \ xs \vee \text{len\_lexext\_rev } \text{gt } xs \ ys \vee ys = xs$

**by** (rule `lenext_total[OF lexext_rev_total]`)

**lemma** `len_lexext_rev_iff_len_lexext`:  $\text{len\_lexext\_rev } \text{gt } ys \ xs \longleftrightarrow \text{len\_lexext } \text{gt } (\text{rev } ys) \ (\text{rev } xs)$

**unfolding** `lenext_def` **by** `simp`

**lemma** `len_lexext_rev_wf`:  $\text{wfP } (\lambda x \ y. \text{gt } y \ x) \Longrightarrow \text{wfP } (\lambda xs \ ys. \text{len\_lexext\_rev } \text{gt } ys \ xs)$

**unfolding** `len_lexext_rev_iff_len_lexext`

**by** (rule `wfP_app[OF lambda xs ys. len_lexext gt ys xs rev, simplified]`) (rule `len_lexext_wf`)

**lemma** `len_lexext_rev_hd_or_tl`:

$\text{len\_lexext\_rev } \text{gt } (y \ # \ ys) \ (x \ # \ xs) \Longrightarrow \text{gt } y \ x \vee \text{len\_lexext\_rev } \text{gt } ys \ xs$

**using** `lenext_hd_or_tl` `lexext_rev_hd_or_tl` **by** `metis`

**interpretation** `len_lexext_rev`: `ext len_lexext_rev`

**by** `standard` (fact `len_lexext_rev_mono_strong`, rule `len_lexext_rev_map_strong`, `metis in_listsD`)

**interpretation** `len_lexext_rev`: `ext_irrefl_trans_strong len_lexext_rev`

**by** `standard` (fact `len_lexext_rev_irrefl`, fact `len_lexext_rev_trans_strong`)

**interpretation** `len_lexext_rev`: `ext_snoc len_lexext_rev`

**by** `standard` (fact `len_lexext_rev_snoc`)

**interpretation** `len_lexext_rev`: `ext_compat_cons len_lexext_rev`

**by** `standard` (fact `len_lexext_rev_compat_cons`)

**interpretation** `len_lexext_rev`: `ext_compat_snoc len_lexext_rev`

**by** `standard` (fact `len_lexext_rev_compat_snoc`)

**interpretation** `len_lexext_rev`: `ext_compat_list len_lexext_rev`

**by** `standard` (rule `len_lexext_rev_compat_list`)

**interpretation**  $len\_lexext\_rev$ :  $ext\_singleton\ len\_lexext\_rev$   
 by *standard* (*rule*  $len\_lexext\_rev\_singleton$ )

**interpretation**  $len\_lexext\_rev$ :  $ext\_total\ len\_lexext\_rev$   
 by *standard* (*fact*  $len\_lexext\_rev\_total$ )

**interpretation**  $len\_lexext\_rev$ :  $ext\_wf\ len\_lexext\_rev$   
 by *standard* (*fact*  $len\_lexext\_rev\_wf$ )

**interpretation**  $len\_lexext\_rev$ :  $ext\_hd\_or\_tl\ len\_lexext\_rev$   
 by *standard* (*rule*  $len\_lexext\_rev\_hd\_or\_tl$ )

**interpretation**  $len\_lexext\_rev$ :  $ext\_wf\_bounded\ len\_lexext\_rev$   
 by *standard*

## 5.7 Dershowitz–Manna Multiset Extension

**definition**  $msetext\_dersh$  **where**

$msetext\_dersh\ gt\ ys\ xs = (let\ N = mset\ ys;\ M = mset\ xs\ in$   
 $(\exists Y\ X. Y \neq \{\#\} \wedge Y \subseteq\# N \wedge M = (N - Y) + X \wedge (\forall x. x \in\# X \longrightarrow (\exists y. y \in\# Y \wedge gt\ y\ x))))$

The following proof is based on that of  $less\_multiset_{DM}\ imp\_mult$ .

**lemma**  $msetext\_dersh\_imp\_mult\_rel$ :

**assumes**

$ys\_a$ :  $ys \in lists\ A$  **and**  $xs\_a$ :  $xs \in lists\ A$  **and**

$ys\_gt\_xs$ :  $msetext\_dersh\ gt\ ys\ xs$

**shows**  $(mset\ xs, mset\ ys) \in mult\ \{(x, y). x \in A \wedge y \in A \wedge gt\ y\ x\}$

**proof** –

**obtain**  $Y\ X$  **where**  $y\_nemp$ :  $Y \neq \{\#\}$  **and**  $y\_sub\_ys$ :  $Y \subseteq\# mset\ ys$  **and**

$xs\_eq$ :  $mset\ xs = mset\ ys - Y + X$  **and**  $ex\_y$ :  $\forall x. x \in\# X \longrightarrow (\exists y. y \in\# Y \wedge gt\ y\ x)$

**using**  $ys\_gt\_xs$ [*unfolded*  $msetext\_dersh\_def\ Let\_def$ ] **by** *blast*

**have**  $ex\_y'$ :  $\forall x. x \in\# X \longrightarrow (\exists y. y \in\# Y \wedge x \in A \wedge y \in A \wedge gt\ y\ x)$

**using**  $ex\_y\ y\_sub\_ys\ xs\_eq\ ys\_a\ xs\_a$  **by** (*metis* *in\_listsD\ mset\_subset\_eqD\ set\_mset\_mset\_union\_iff*)

**hence**  $(mset\ ys - Y + X, mset\ ys - Y + Y) \in mult\ \{(x, y). x \in A \wedge y \in A \wedge gt\ y\ x\}$

**using**  $y\_nemp\ y\_sub\_ys$  **by** (*intro* *one\_step\_implies\_mult*) (*auto* *simp*:  $Be_x\_def\ trans\_def$ )

**thus** *?thesis*

**using**  $xs\_eq\ y\_sub\_ys$  **by** (*simp* *add*:  $subset\_mset.diff\_add$ )

**qed**

**lemma**  $msetext\_dersh\_imp\_mult$ :  $msetext\_dersh\ gt\ ys\ xs \implies (mset\ xs, mset\ ys) \in mult\ \{(x, y). gt\ y\ x\}$

**using**  $msetext\_dersh\_imp\_mult\_rel$ [*of*  $\_ UNIV$ ] **by** *auto*

**lemma**  $mult\_imp\_msetext\_dersh\_rel$ :

**assumes**

$ys\_a$ :  $set\_mset\ (mset\ ys) \subseteq A$  **and**  $xs\_a$ :  $set\_mset\ (mset\ xs) \subseteq A$  **and**

$in\_mult$ :  $(mset\ xs, mset\ ys) \in mult\ \{(x, y). x \in A \wedge y \in A \wedge gt\ y\ x\}$  **and**

$trans$ :  $\forall z \in A. \forall y \in A. \forall x \in A. gt\ z\ y \longrightarrow gt\ y\ x \longrightarrow gt\ z\ x$

**shows**  $msetext\_dersh\ gt\ ys\ xs$

**using**  $in\_mult\ ys\_a\ xs\_a$  **unfolding**  $mult\_def\ msetext\_dersh\_def\ Let\_def$

**proof** *induct*

**case** (*base*  $Ys$ )

**then obtain**  $y\ M0\ X$  **where**  $Ys = M0 + \{\#y\#\}$  **and**  $mset\ xs = M0 + X$  **and**  $\forall a. a \in\# X \longrightarrow gt\ y\ a$

**unfolding**  $mult1\_def$  **by** *auto*

**thus** *?case*

**by** (*auto* *intro*:  $exI$ [*of*  $\_ \{\#y\#\}$ ]  $exI$ [*of*  $\_ X$ ])

**next**

**case** (*step*  $Ys\ Zs$ )

**note**  $ys\_zs\_in\_mult1 = this(2)$  **and**  $ih = this(3)$  **and**  $zs\_a = this(4)$  **and**  $xs\_a = this(5)$

**have**  $Ys\_a$ :  $set\_mset\ Ys \subseteq A$

**using**  $ys\_zs\_in\_mult1\ zs\_a$  **unfolding**  $mult1\_def$  **by** *auto*

**obtain**  $Y\ X$  **where**  $y\_nemp$ :  $Y \neq \{\#\}$  **and**  $y\_sub\_ys$ :  $Y \subseteq\# Ys$  **and**  $xs\_eq$ :  $mset\ xs = Ys - Y + X$  **and**

```

ex_y:  $\forall x. x \in\# X \longrightarrow (\exists y. y \in\# Y \wedge gt\ y\ x)$ 
using ih[OF Ys_a xs_a] by blast

obtain z M0 Ya where zs_eq:  $Zs = M0 + \{\#z\}$  and ys_eq:  $Ys = M0 + Ya$  and
z_gt:  $\forall y. y \in\# Ya \longrightarrow y \in A \wedge z \in A \wedge gt\ z\ y$ 
using ys_zs_in_mult1[unfolded mult1_def] by auto

let ?Za =  $Y - Ya + \{\#z\}$ 
let ?Xa =  $X + Ya + (Y - Ya) - Y$ 

have xa_sub_x_ya:  $set\_mset\ ?Xa \subseteq set\_mset\ (X + Ya)$ 
by (metis diff_subset_eq_self in_diffD subsetI subset_mset.diff_diff_right)

have x_a:  $set\_mset\ X \subseteq A$ 
using xs_a xs_eq by auto
have ya_a:  $set\_mset\ Ya \subseteq A$ 
by (simp add: subsetI z_gt)

have ex_y':  $\exists y. y \in\# Y - Ya + \{\#z\} \wedge gt\ y\ x$  if x_in:  $x \in\# X + Ya$  for x
proof (cases  $x \in\# X$ )
case True
then obtain y where y_in:  $y \in\# Y$  and y_gt_x:  $gt\ y\ x$ 
using ex_y by blast
show ?thesis
proof (cases  $y \in\# Ya$ )
case False
hence  $y \in\# Y - Ya + \{\#z\}$ 
using y_in by fastforce
thus ?thesis
using y_gt_x by blast
next
case True
hence  $y \in A$  and  $z \in A$  and  $gt\ z\ y$ 
using z_gt by blast+
hence  $gt\ z\ x$ 
using trans y_gt_x x_a ya_a x_in by (meson subsetCE union_iff)
thus ?thesis
by auto
qed
next
case False
hence  $x \in\# Ya$ 
using x_in by auto
hence  $x \in A$  and  $z \in A$  and  $gt\ z\ x$ 
using z_gt by blast+
thus ?thesis
by auto
qed

show ?case
proof (rule exI[of _ ?Za], rule exI[of _ ?Xa], intro conjI)
show  $Y - Ya + \{\#z\} \subseteq\# Zs$ 
using mset_subset_eq_mono_add subset_eq_diff_conv y_sub_ys ys_eq zs_eq by fastforce
next
show  $mset\ xs = Zs - (Y - Ya + \{\#z\}) + (X + Ya + (Y - Ya) - Y)$ 
unfolding xs_eq ys_eq zs_eq by (auto simp: multiset_eq_iff)
next
show  $\forall x. x \in\# X + Ya + (Y - Ya) - Y \longrightarrow (\exists y. y \in\# Y - Ya + \{\#z\} \wedge gt\ y\ x)$ 
using ex_y' xa_sub_x_ya by blast
qed auto
qed

lemma msetext_dersh_mono_strong:

```

$(\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y \ x \longrightarrow \text{gt}' \ y \ x) \implies \text{msetext\_dersh } \text{gt } \text{ys } \text{xs} \implies$   
 $\text{msetext\_dersh } \text{gt}' \ \text{ys } \text{xs}$   
**unfolding**  $\text{msetext\_dersh\_def } \text{Let\_def}$   
**by** ( $\text{metis } \text{mset\_subset\_eqD } \text{mset\_subset\_eq\_add\_right } \text{set\_mset\_mset}$ )

**lemma**  $\text{msetext\_dersh\_map\_strong}$ :  
**assumes**  
 $\text{compat\_f}: \forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y \ x \longrightarrow \text{gt } (f \ y) (f \ x)$  **and**  
 $\text{ys\_gt\_xs}: \text{msetext\_dersh } \text{gt } \text{ys } \text{xs}$   
**shows**  $\text{msetext\_dersh } \text{gt } (\text{map } f \ \text{ys}) (\text{map } f \ \text{xs})$   
**proof** –  
**obtain**  $Y \ X$  **where**  
 $y\_nemp: Y \neq \{\#\}$  **and**  $y\_sub\_ys: Y \subseteq\# \text{mset } \text{ys}$  **and**  $xs\_eq: \text{mset } \text{xs} = \text{mset } \text{ys} - Y + X$  **and**  
 $ex\_y: \forall x. x \in\# X \longrightarrow (\exists y. y \in\# Y \wedge \text{gt } y \ x)$   
**using**  $\text{ys\_gt\_xs}[\text{unfolded } \text{msetext\_dersh\_def } \text{Let\_def } \text{mset\_map}]$  **by**  $\text{blast}$

**have**  $x\_sub\_xs: X \subseteq\# \text{mset } \text{xs}$   
**using**  $xs\_eq$  **by**  $\text{simp}$

**let**  $?fY = \text{image\_mset } f \ Y$   
**let**  $?fX = \text{image\_mset } f \ X$

**show**  $?thesis$   
**unfolding**  $\text{msetext\_dersh\_def } \text{Let\_def } \text{mset\_map}$   
**proof** ( $\text{intro } exI \ conjI$ )  
**show**  $\text{image\_mset } f (\text{mset } \text{xs}) = \text{image\_mset } f (\text{mset } \text{ys}) - ?fY + ?fX$   
**using**  $xs\_eq[\text{THEN } \text{arg\_cong}, \text{of } \text{image\_mset } f] \ y\_sub\_ys$  **by** ( $\text{metis } \text{image\_mset\_Diff } \text{image\_mset\_union}$ )  
**next**  
**obtain**  $y$  **where**  $y: \forall x. x \in\# X \longrightarrow y \ x \in\# Y \wedge \text{gt } (y \ x) \ x$   
**using**  $ex\_y$  **by**  $\text{moura}$

**show**  $\forall fx. fx \in\# ?fX \longrightarrow (\exists fy. fy \in\# ?fY \wedge \text{gt } fy \ fx)$   
**proof** ( $\text{intro } allI \ impI$ )  
**fix**  $fx$   
**assume**  $fx \in\# ?fX$   
**then obtain**  $x$  **where**  $fx: fx = f \ x$  **and**  $x\_in: x \in\# X$   
**by**  $\text{auto}$   
**hence**  $y\_in: y \ x \in\# Y$  **and**  $y\_gt: \text{gt } (y \ x) \ x$   
**using**  $y[\text{rule\_format}, \text{OF } x\_in]$  **by**  $\text{blast+}$   
**hence**  $f \ (y \ x) \in\# ?fY \wedge \text{gt } (f \ (y \ x)) (f \ x)$   
**using**  $\text{compat\_f } y\_sub\_ys \ x\_sub\_xs \ x\_in$   
**by** ( $\text{metis } \text{image\_eqI } \text{in\_image\_mset } \text{mset\_subset\_eqD } \text{set\_mset\_mset}$ )  
**thus**  $\exists fy. fy \in\# ?fY \wedge \text{gt } fy \ fx$   
**unfolding**  $fx$  **by**  $\text{auto}$   
**qed**

**qed** ( $\text{auto } \text{simp}: y\_nemp \ y\_sub\_ys \ \text{image\_mset\_subsetq\_mono}$ )  
**qed**

**lemma**  $\text{msetext\_dersh\_trans}$ :  
**assumes**  
 $zs\_a: zs \in \text{lists } A$  **and**  
 $ys\_a: ys \in \text{lists } A$  **and**  
 $xs\_a: xs \in \text{lists } A$  **and**  
 $\text{trans}: \forall z \in A. \forall y \in A. \forall x \in A. \text{gt } z \ y \longrightarrow \text{gt } y \ x \longrightarrow \text{gt } z \ x$  **and**  
 $zs\_gt\_ys: \text{msetext\_dersh } \text{gt } zs \ ys$  **and**  
 $ys\_gt\_xs: \text{msetext\_dersh } \text{gt } ys \ xs$   
**shows**  $\text{msetext\_dersh } \text{gt } zs \ xs$   
**proof** ( $\text{rule } \text{mult\_imp\_msetext\_dersh\_rel}[\text{OF } \_ \_ \_ \text{trans}]$ )  
**show**  $\text{set\_mset } (\text{mset } zs) \subseteq A$   
**using**  $zs\_a$  **by**  $\text{auto}$   
**next**  
**show**  $\text{set\_mset } (\text{mset } xs) \subseteq A$   
**using**  $xs\_a$  **by**  $\text{auto}$

```

next
  let ?Gt = {(x, y). x ∈ A ∧ y ∈ A ∧ gt y x}

  have (mset xs, mset ys) ∈ mult ?Gt
    by (rule msetext_dersh_imp_mult_rel[OF ys_a xs_a ys_gt_xs])
  moreover have (mset ys, mset zs) ∈ mult ?Gt
    by (rule msetext_dersh_imp_mult_rel[OF zs_a ys_a zs_gt_ys])
  ultimately show (mset xs, mset zs) ∈ mult ?Gt
    unfolding mult_def by simp
qed

lemma msetext_dersh_irrefl_from_trans:
  assumes
    trans: ∀z ∈ set xs. ∀y ∈ set xs. ∀x ∈ set xs. gt z y → gt y x → gt z x and
    irrefl: ∀x ∈ set xs. ¬ gt x x
  shows ¬ msetext_dersh gt xs xs
  unfolding msetext_dersh_def Let_def
proof clarify
  fix Y X
  assume y_nemp: Y ≠ {#} and y_sub_xs: Y ⊆# mset xs and xs_eq: mset xs = mset xs - Y + X and
    ex_y: ∀x. x ∈# X → (∃y. y ∈# Y ∧ gt y x)

  have x_eq_y: X = Y
    using y_sub_xs xs_eq by (metis diff_union_cancelL subset_mset.diff_add)

  let ?Gt = {(y, x). y ∈# Y ∧ x ∈# Y ∧ gt y x}

  have ?Gt ⊆ set_mset Y × set_mset Y
    by auto
  hence fin: finite ?Gt
    by (auto dest!: infinite_super)
  moreover have irrefl ?Gt
    unfolding irrefl_def using irrefl y_sub_xs by (fastforce dest!: set_mset_mono)
  moreover have trans ?Gt
    unfolding trans_def using trans y_sub_xs by (fastforce dest!: set_mset_mono)
  ultimately have acyc: acyclic ?Gt
    by (rule finite_irrefl_trans_imp_wf[THEN wf_acyclic])

  have fin_y: finite (set_mset Y)
    using y_sub_xs by simp
  hence cyc: ¬ acyclic ?Gt
    proof (rule finite_nonempty_ex_succ_imp_cyclic)
      show ∀x ∈# Y. ∃y ∈# Y. (y, x) ∈ ?Gt
        using ex_y[unfolded x_eq_y] by auto
      qed (auto simp: y_nemp)

  show False
    using acyc cyc by sat
qed

lemma msetext_dersh_snoc: msetext_dersh gt (xs @ [x]) xs
  unfolding msetext_dersh_def Let_def
proof (intro exI conjI)
  show mset xs = mset (xs @ [x]) - {#x#} + {#}
    by simp
qed auto

lemma msetext_dersh_compat_cons:
  assumes ys_gt_xs: msetext_dersh gt ys xs
  shows msetext_dersh gt (x # ys) (x # xs)
proof -
  obtain Y X where
    y_nemp: Y ≠ {#} and y_sub_ys: Y ⊆# mset ys and xs_eq: mset xs = mset ys - Y + X and

```

```

ex_y:  $\forall x. x \in\# X \longrightarrow (\exists y. y \in\# Y \wedge gt\ y\ x)$ 
using ys_gt_xs[unfolded msetext_dersh_def Let_def mset_map] by blast

show ?thesis
  unfolding msetext_dersh_def Let_def
proof (intro exI conjI)
  show  $Y \subseteq\# mset\ (x \# ys)$ 
    using y_sub_ys
    by (metis add_mset_add_single mset.simps(2) mset_subset_eq_add_left
        subset_mset.add_increasing2)
next
  show  $mset\ (x \# xs) = mset\ (x \# ys) - Y + X$ 
  proof -
    have  $X + (mset\ ys - Y) = mset\ xs$ 
      by (simp add: union_commute xs_eq)
    hence  $mset\ (x \# xs) = X + (mset\ (x \# ys) - Y)$ 
      by (metis add_mset_add_single mset.simps(2) mset_subset_eq_multiset_union_diff_commute
          union_mset_add_mset_right y_sub_ys)
    thus ?thesis
      by (simp add: union_commute)
  qed
qed (auto simp: y_nemp ex_y)
qed

lemma msetext_dersh_compat_snoc: msetext_dersh gt ys xs  $\implies$  msetext_dersh gt (ys @ [x]) (xs @ [x])
  using msetext_dersh_compat_cons[of gt ys xs x] unfolding msetext_dersh_def by simp

lemma msetext_dersh_compat_list:
  assumes y_gt_x: gt y x
  shows msetext_dersh gt (xs @ y # xs') (xs @ x # xs')
  unfolding msetext_dersh_def Let_def
proof (intro exI conjI)
  show  $mset\ (xs @ x \# xs') = mset\ (xs @ y \# xs') - \{y\} + \{x\}$ 
    by auto
  qed (auto intro: y_gt_x)

lemma msetext_dersh_singleton: msetext_dersh gt [y] [x]  $\longleftrightarrow$  gt y x
  unfolding msetext_dersh_def Let_def
  by (auto dest: nonempty_subseteq_mset_eq_singleton simp: nonempty_subseteq_mset_iff_singleton)

lemma msetext_dersh_wf:
  assumes wf_gt: wfP ( $\lambda x\ y. gt\ y\ x$ )
  shows wfP ( $\lambda xs\ ys. msetext_dersh\ gt\ ys\ xs$ )
proof (rule wfP_subset, rule wfP_app[of  $\lambda xs\ ys. (xs, ys) \in mult\ \{(x, y). gt\ y\ x\}\ mset$ ])
  show wfP ( $\lambda xs\ ys. (xs, ys) \in mult\ \{(x, y). gt\ y\ x\}$ )
    using wf_gt unfolding wfP_def by (auto intro: wf_mult)
next
  show ( $\lambda xs\ ys. msetext_dersh\ gt\ ys\ xs$ )  $\leq$  ( $\lambda x\ y. (mset\ x, mset\ y) \in mult\ \{(x, y). gt\ y\ x\}$ )
    using msetext_dersh_imp_mult by blast
qed

interpretation msetext_dersh: ext msetext_dersh
  by standard (fact msetext_dersh_mono_strong, rule msetext_dersh_map_strong, metis in_listsD)

interpretation msetext_dersh: ext_trans_before_irrefl msetext_dersh
  by standard (fact msetext_dersh_trans, fact msetext_dersh_irrefl_from_trans)

interpretation msetext_dersh: ext_snoc msetext_dersh
  by standard (fact msetext_dersh_snoc)

interpretation msetext_dersh: ext_compat_cons msetext_dersh
  by standard (fact msetext_dersh_compat_cons)

```

**interpretation** *msetext\_dersh*: *ext\_compat\_snoc msetext\_dersh*  
 by *standard (fact msetext\_dersh\_compat\_snoc)*

**interpretation** *msetext\_dersh*: *ext\_compat\_list msetext\_dersh*  
 by *standard (rule msetext\_dersh\_compat\_list)*

**interpretation** *msetext\_dersh*: *ext\_singleton msetext\_dersh*  
 by *standard (rule msetext\_dersh\_singleton)*

**interpretation** *msetext\_dersh*: *ext\_wf msetext\_dersh*  
 by *standard (fact msetext\_dersh\_wf)*

## 5.8 Huet–Oppen Multiset Extension

**definition** *msetext\_huet* **where**

*msetext\_huet gt ys xs = (let N = mset ys; M = mset xs in*  
*M ≠ N ∧ (∀ x. count M x > count N x → (∃ y. gt y x ∧ count N y > count M y)))*

**lemma** *msetext\_huet\_imp\_count\_gt*:

**assumes** *ys\_gt\_xs*: *msetext\_huet gt ys xs*

**shows**  $\exists x. \text{count } (mset \text{ ys}) \ x > \text{count } (mset \text{ xs}) \ x$

**proof** –

**obtain** *x* **where** *count (mset ys) x ≠ count (mset xs) x*

**using** *ys\_gt\_xs*[*unfolded msetext\_huet\_def Let\_def*] **by** (*fastforce intro: multiset\_eqI*)

**moreover**

{  
**assume** *count (mset ys) x < count (mset xs) x*  
**hence** *?thesis*

**using** *ys\_gt\_xs*[*unfolded msetext\_huet\_def Let\_def*] **by** *blast*

}

**moreover**

{  
**assume** *count (mset ys) x > count (mset xs) x*  
**hence** *?thesis*

**by** *fast*

}

**ultimately show** *?thesis*

**by** *fastforce*

**qed**

**lemma** *msetext\_huet\_imp\_dersh*:

**assumes** *huet*: *msetext\_huet gt ys xs*

**shows** *msetext\_dersh gt ys xs*

**proof** (*unfold msetext\_dersh\_def Let\_def, intro exI conjI*)

**let** *?X = mset xs - mset ys*

**let** *?Y = mset ys - mset xs*

**show** *?Y ≠ {#}*

**by** (*metis msetext\_huet\_imp\_count\_gt[OF huet] empty\_iff\_in\_diff\_count set\_mset\_empty*)

**show** *?Y ⊆# mset ys*

**by** *auto*

**show** *mset xs = mset ys - ?Y + ?X*

**by** (*metis add.commute diff\_intersect\_right\_idem multiset\_inter\_def subset\_mset.inf.cobounded2 subset\_mset.le\_imp\_diff\_is\_add*)

**show**  $\forall x. x \in \# \ ?X \rightarrow (\exists y. y \in \# \ ?Y \wedge gt \ y \ x)$

**using** *huet*[*unfolded msetext\_huet\_def Let\_def, THEN conjunct2*] **by** (*meson in\_diff\_count*)

**qed**

The following proof is based on that of *mult\_imp\_less\_multiset<sub>HO</sub>*.

**lemma** *mult\_imp\_msetext\_huet*:

**assumes**

*irrefl*: *irreflp gt* **and** *trans*: *transp gt* **and**

*in\_mult*:  $(mset \ xs, mset \ ys) \in \text{mult } \{(x, y). gt \ y \ x\}$

**shows** *msetext\_huet gt ys xs*



```

using in_mult unfolding mult_def msetext_huet_def Let_def
proof (induct rule: trancl_induct)
case (base Ys)
thus ?case
  using irrefl unfolding irreflp_def msetext_huet_def Let_def mult1_def
  by (auto 0 3 split: if_splits)
next
case (step Ys Zs)

have asym[unfolded antisym_def, simplified]: antisymp gt
  by (rule irreflp_transp_imp_antisymP[OF irrefl trans])

from step(3) have mset xs ≠ Ys and
  **:  $\bigwedge x. \text{count } Ys\ x < \text{count } (mset\ xs)\ x \implies (\exists y. gt\ y\ x \wedge \text{count } (mset\ xs)\ y < \text{count } Ys\ y)$ 
  by blast+
from step(2) obtain M0 a K where
  *:  $Zs = M0 + \{\#a\# \} Ys = M0 + K\ a \notin\# K \wedge b. b \in\# K \implies gt\ a\ b$ 
  using irrefl unfolding mult1_def irreflp_def by force
have mset xs ≠ Zs
proof (cases K = {\#})
case True
thus ?thesis
  using  $\langle mset\ xs \neq Ys \rangle ** *(1,2)$  irrefl[unfolded irreflp_def]
  by (metis One_nat_def add.comm_neutral count_single diff_union_cancelL lessI
    minus_multiset.rep_eq not_add_less2 plus_multiset.rep_eq union_commute zero_less_diff)
next
case False
thus ?thesis
proof -
obtain aa :: 'a  $\Rightarrow$  'a where
  f1:  $\forall a. \neg \text{count } Ys\ a < \text{count } (mset\ xs)\ a \vee gt\ (aa\ a)\ a \wedge$ 
     $\text{count } (mset\ xs)\ (aa\ a) < \text{count } Ys\ (aa\ a)$ 
  using ** by moura
have f2:  $K + M0 = Ys$ 
  using *(2) union_ac(2) by blast
have f3:  $\bigwedge aa. \text{count } Zs\ aa = \text{count } M0\ aa + \text{count } \{\#a\#\}\ aa$ 
  by (simp add: *(1))
have f4:  $\bigwedge a. \text{count } Ys\ a = \text{count } K\ a + \text{count } M0\ a$ 
  using f2 by auto
have f5:  $\text{count } K\ a = 0$ 
  by (meson *(3) count_inI)
have  $Zs - M0 = \{\#a\#\}$ 
  using *(1) add_diff_cancel_left' by blast
then have f6:  $\text{count } M0\ a < \text{count } Zs\ a$ 
  by (metis in_diff_count union_single_eq_member)
have  $\bigwedge m. \text{count } m\ a = 0 + \text{count } m\ a$ 
  by simp
moreover
{ assume aa a ≠ a
  then have  $mset\ xs = Zs \wedge \text{count } Zs\ (aa\ a) < \text{count } K\ (aa\ a) + \text{count } M0\ (aa\ a) \longrightarrow$ 
     $\text{count } K\ (aa\ a) + \text{count } M0\ (aa\ a) < \text{count } Zs\ (aa\ a)$ 
    using f5 f3 f2 f1 *(4) asym by (auto dest!: antisympD) }
ultimately show ?thesis
  using f6 f5 f4 f1 by (metis less_imp_not_less)
qed
qed
moreover
{
assume  $\text{count } Zs\ a \leq \text{count } (mset\ xs)\ a$ 
with  $\langle a \notin\# K \rangle$  have  $\text{count } Ys\ a < \text{count } (mset\ xs)\ a$  unfolding *(1,2)
  by (auto simp add: not_in_iff)
with ** obtain z where  $z: gt\ z\ a \wedge \text{count } (mset\ xs)\ z < \text{count } Ys\ z$ 
  by blast
}

```

```

with * have count Ys z ≤ count Zs z
  using asym
  by (auto simp: intro: count_inI dest: antisymD)
with z have ∃z. gt z a ∧ count (mset xs) z < count Zs z by auto
}
note count_a = this
{
  fix y
  assume count_y: count Zs y < count (mset xs) y
  have ∃x. gt x y ∧ count (mset xs) x < count Zs x
  proof (cases y = a)
    case True
    with count_y count_a show ?thesis by auto
  next
  case False
  show ?thesis
  proof (cases y ∈# K)
    case True
    with *(4) have gt a y by simp
    then show ?thesis
    by (cases count Zs a ≤ count (mset xs) a,
      blast dest: count_a trans[unfolded transp_def, rule_format], auto dest: count_a)
  next
  case False
  with ⟨y ≠ a⟩ have count Zs y = count Ys y unfolding *(1,2)
  by (simp add: not_in_iff)
  with count_y ** obtain z where z: gt z y count (mset xs) z < count Ys z by auto
  show ?thesis
  proof (cases z ∈# K)
    case True
    with *(4) have gt a z by simp
    with z(1) show ?thesis
    by (cases count Zs a ≤ count (mset xs) a)
    (blast dest: count_a not_le_imp_less trans[unfolded transp_def, rule_format])+
  next
  case False
  with ⟨a ∉# K⟩ have count Ys z ≤ count Zs z unfolding *
  by (auto simp add: not_in_iff)
  with z show ?thesis by auto
  qed
  qed
  qed
}
ultimately show ?case
  unfolding msetext_huet_def Let_def by blast
qed

```

**theorem** *msetext\_huet\_eq\_dersh*:  $\text{irreflp } gt \implies \text{transp } gt \implies \text{msetext\_dersh } gt = \text{msetext\_huet } gt$   
**using** *msetext\_huet\_imp\_dersh msetext\_dersh\_imp\_mult mult\_imp\_msetext\_huet* **by** *fast*

**lemma** *msetext\_huet\_mono\_strong*:  
 $(\forall y \in \text{set } ys. \forall x \in \text{set } xs. gt \ y \ x \longrightarrow gt' \ y \ x) \implies \text{msetext\_huet } gt \ ys \ xs \implies \text{msetext\_huet } gt' \ ys \ xs$   
**unfolding** *msetext\_huet\_def*  
**by** (*metis less\_le\_trans mem\_Collect\_eq not\_le not\_less0 set\_mset\_mset[unfolded set\_mset\_def]*)

**lemma** *msetext\_huet\_map*:  
**assumes**  
*fin*: *finite A* **and**  
*ys\_a*:  $ys \in \text{lists } A$  **and** *xs\_a*:  $xs \in \text{lists } A$  **and**  
*irrefl\_f*:  $\forall x \in A. \neg gt (f x) (f x)$  **and**  
*trans\_f*:  $\forall z \in A. \forall y \in A. \forall x \in A. gt (f z) (f y) \longrightarrow gt (f y) (f x) \longrightarrow gt (f z) (f x)$  **and**  
*compat\_f*:  $\forall y \in A. \forall x \in A. gt \ y \ x \longrightarrow gt (f y) (f x)$  **and**  
*ys\_gt\_xs*: *msetext\_huet* *gt* *ys xs*

```

shows msetext_huet gt (map f ys) (map f xs) (is msetext_huet _ ?fys ?fxs)
proof -
  have irrefl:  $\forall x \in A. \neg gt\ x\ x$ 
    using irrefl_f compat_f by blast

  have
    ms_xs_ne_ys: mset xs  $\neq$  mset ys and
    ex_gt:  $\forall x. count\ (mset\ ys)\ x < count\ (mset\ xs)\ x \longrightarrow$ 
      ( $\exists y. gt\ y\ x \wedge count\ (mset\ xs)\ y < count\ (mset\ ys)\ y$ )
    using ys_gt_xs[unfolded msetext_huet_def Let_def] by blast+

  have ex_y:  $\exists y. gt\ (f\ y)\ (f\ x) \wedge count\ (mset\ ?fxs)\ (f\ y) < count\ (mset\ (map\ f\ ys))\ (f\ y)$ 
    if cnt_x:  $count\ (mset\ xs)\ x > count\ (mset\ ys)\ x$  for x
  proof -
    have x_in_a:  $x \in A$ 
      using cnt_x xs_a dual_order.strict_trans2 by fastforce

    obtain y where y_gt_x:  $gt\ y\ x$  and cnt_y:  $count\ (mset\ ys)\ y > count\ (mset\ xs)\ y$ 
      using cnt_x ex_gt by blast
    have y_in_a:  $y \in A$ 
      using cnt_y ys_a dual_order.strict_trans2 by fastforce

    have wf_gt_f:  $wfP\ (\lambda y\ x. y \in A \wedge x \in A \wedge gt\ (f\ y)\ (f\ x))$ 
      by (rule finite_irreflp_trans_imp_wfp)
      (auto elim: trans_f[rule_format] simp: fin_irrefl_f Collect_case_prod_Sigma irreflp_def
        trans_def)

    obtain yy where
      fyy_gt_fx:  $gt\ (f\ yy)\ (f\ x)$  and
      cnt_yy:  $count\ (mset\ ys)\ yy > count\ (mset\ xs)\ yy$  and
      max_yy:  $\forall y \in A. yy \in A \longrightarrow gt\ (f\ y)\ (f\ yy) \longrightarrow gt\ (f\ y)\ (f\ x) \longrightarrow$ 
         $count\ (mset\ xs)\ y \geq count\ (mset\ ys)\ y$ 
      using wfP_eq_minimal[THEN iffD1, OF wf_gt_f, rule_format,
        of y {y. gt (f y) (f x)  $\wedge$  count (mset xs) y < count (mset ys) y}, simplified]
      y_gt_x cnt_y
      by (metis compat_f not_less x_in_a y_in_a)
    have yy_in_a:  $yy \in A$ 
      using cnt_yy ys_a dual_order.strict_trans2 by fastforce

    {
      assume count (mset ?fxs) (f yy)  $\geq$  count (mset ?fys) (f yy)
      then obtain u where fu_eq_fyy:  $f\ u = f\ yy$  and cnt_u:  $count\ (mset\ xs)\ u > count\ (mset\ ys)\ u$ 
        using count_image_mset_le_imp_lt cnt_yy mset_map by (metis (mono_tags))
      have u_in_a:  $u \in A$ 
        using cnt_u xs_a dual_order.strict_trans2 by fastforce

      obtain v where v_gt_u:  $gt\ v\ u$  and cnt_v:  $count\ (mset\ ys)\ v > count\ (mset\ xs)\ v$ 
        using cnt_u ex_gt by blast
      have v_in_a:  $v \in A$ 
        using cnt_v ys_a dual_order.strict_trans2 by fastforce

      have fv_gt_fu:  $gt\ (f\ v)\ (f\ u)$ 
        using v_gt_u compat_f v_in_a u_in_a by blast
      hence fv_gt_fyy:  $gt\ (f\ v)\ (f\ yy)$ 
        by (simp only: fu_eq_fyy)

      have gt (f v) (f x)
        using fv_gt_fyy fyy_gt_fx v_in_a yy_in_a x_in_a trans_f by blast
      hence False
        using max_yy[rule_format, of v] fv_gt_fyy v_in_a yy_in_a cnt_v by linarith
    }
  thus ?thesis
    using fyy_gt_fx leI by blast

```

qed

show *?thesis*

unfolding *msetext\_huet\_def Let\_def*

proof (intro conjI allI impI)

```
{
  assume len_eq: length xs = length ys
  obtain x where cnt_x: count (mset xs) x > count (mset ys) x
  using len_eq ms_xs_ne_ys by (metis size_eq_ex_count_lt size_mset)
  hence mset ?fxs ≠ mset ?fys
  using ex_y by fastforce
}
thus mset ?fxs ≠ mset (map f ys)
  by (metis length_map size_mset)
```

next

fix *fx*

assume *cnt\_fx*: count (mset *?fxs*) *fx* > count (mset *?fys*) *fx*

then obtain *x* where *fx*: *fx* = *f x* and *cnt\_x*: count (mset *xs*) *x* > count (mset *ys*) *x*

using *count\_image\_mset\_lt\_imp\_lt mset\_map* by (metis (*mono\_tags*))

thus ∃*fy*. *gt fy fx* ∧ count (mset *?fxs*) *fy* < count (mset (map *f ys*)) *fy*

using *ex\_y[OF cnt\_x]* by blast

qed

qed

lemma *msetext\_huet\_irrefl*: (∀*x* ∈ *set xs*. ¬ *gt x x*) ⇒ ¬ *msetext\_huet gt xs xs*

unfolding *msetext\_huet\_def* by simp

lemma *msetext\_huet\_trans\_from\_irrefl*:

assumes

*fin*: finite *A* and

*zs\_a*: *zs* ∈ *lists A* and *ys\_a*: *ys* ∈ *lists A* and *xs\_a*: *xs* ∈ *lists A* and

*irrefl*: ∀*x* ∈ *A*. ¬ *gt x x* and

*trans*: ∀*z* ∈ *A*. ∀*y* ∈ *A*. ∀*x* ∈ *A*. *gt z y* ⇒ *gt y x* ⇒ *gt z x* and

*zs\_gt\_ys*: *msetext\_huet gt zs ys* and

*ys\_gt\_xs*: *msetext\_huet gt ys xs*

shows *msetext\_huet gt zs xs*

proof –

have *wf\_gt*: *wfP* (λ*y x*. *y* ∈ *A* ∧ *x* ∈ *A* ∧ *gt y x*)

by (rule *finite\_irreflp\_transp\_imp\_wfp*)

(auto elim: *trans[rule\_format] simp: fin irrefl Collect\_case\_prod\_Sigma irreflp\_def transp\_def*)

show *?thesis*

unfolding *msetext\_huet\_def Let\_def*

proof (intro conjI allI impI)

obtain *x* where *cnt\_x*: count (mset *zs*) *x* > count (mset *ys*) *x*

using *msetext\_huet\_imp\_count\_gt[OF zs\_gt\_ys]* by blast

have *x\_in\_a*: *x* ∈ *A*

using *cnt\_x zs\_a dual\_order.strict\_trans2* by fastforce

obtain *xx* where

*cnt\_xx*: count (mset *zs*) *xx* > count (mset *ys*) *xx* and

*max\_xx*: ∀*y* ∈ *A*. *xx* ∈ *A* ⇒ *gt y xx* ⇒ count (mset *ys*) *y* ≥ count (mset *zs*) *y*

using *wfP\_eq\_minimal[THEN iffD1, OF wf\_gt, rule\_format, of x {y. count (mset *ys*) *y* < count (mset *zs*) *y*}, simplified]*

*cnt\_x*

by force

have *xx\_in\_a*: *xx* ∈ *A*

using *cnt\_xx zs\_a dual\_order.strict\_trans2* by fastforce

show *mset xs* ≠ *mset zs*

proof (cases count (mset *ys*) *xx* ≥ count (mset *xs*) *xx*)

case True

```

thus ?thesis
  using cnt_xx by fastforce
next
case False
hence count (mset ys) xx < count (mset xs) xx
  by fastforce
then obtain z where z_gt_xx: gt z xx and cnt_z: count (mset ys) z > count (mset xs) z
  using ys_gt_xs[unfolded msetext_huet_def Let_def] by blast
have z_in_a: z ∈ A
  using cnt_z ys_a dual_order.strict_trans2 by fastforce

have count (mset zs) z ≤ count (mset ys) z
  using max_xx[rule_format, of z] z_in_a xx_in_a z_gt_xx by blast
moreover
{
  assume count (mset zs) z < count (mset ys) z
  then obtain u where u_gt_z: gt u z and cnt_u: count (mset ys) u < count (mset zs) u
  using zs_gt_ys[unfolded msetext_huet_def Let_def] by blast
  have u_in_a: u ∈ A
  using cnt_u zs_a dual_order.strict_trans2 by fastforce
  have u_gt_xx: gt u xx
  using trans u_in_a z_in_a xx_in_a u_gt_z z_gt_xx by blast
  have False
  using max_xx[rule_format, of u] u_in_a xx_in_a u_gt_xx cnt_u by fastforce
}
ultimately have count (mset zs) z = count (mset ys) z
  by fastforce
thus ?thesis
  using cnt_z by fastforce
qed
next
fix x
assume cnt_x_xz: count (mset zs) x < count (mset xs) x
have x_in_a: x ∈ A
  using cnt_x_xz zs_a dual_order.strict_trans2 by fastforce

let ?case = ∃ y. gt y x ∧ count (mset zs) y > count (mset xs) y

{
  assume cnt_x: count (mset zs) x < count (mset ys) x
  then obtain y where y_gt_x: gt y x and cnt_y: count (mset zs) y > count (mset ys) y
  using zs_gt_ys[unfolded msetext_huet_def Let_def] by blast
  have y_in_a: y ∈ A
  using cnt_y zs_a dual_order.strict_trans2 by fastforce

obtain yy where
  yy_gt_x: gt yy x and
  cnt_yy: count (mset zs) yy > count (mset ys) yy and
  max_yy: ∀ y ∈ A. yy ∈ A → gt y yy → gt y x → count (mset ys) y ≥ count (mset zs) y
  using wfP_eq_minimal[THEN iffD1, OF wf_gt, rule_format,
    of y {y. gt y x ∧ count (mset ys) y < count (mset zs) y}, simplified]
  y_gt_x cnt_y
  by force
have yy_in_a: yy ∈ A
  using cnt_yy zs_a dual_order.strict_trans2 by fastforce

have ?case
proof (cases count (mset ys) yy ≥ count (mset xs) yy)
case True
  thus ?thesis
    using yy_gt_x cnt_yy by fastforce
next
case False

```

```

hence count (mset ys) yy < count (mset xs) yy
  by fastforce
then obtain z where z_gt_yy: gt z yy and cnt_z: count (mset ys) z > count (mset xs) z
  using ys_gt_xs[unfolded msetext_huet_def Let_def] by blast
have z_in_a: z ∈ A
  using cnt_z ys_a dual_order.strict_trans2 by fastforce
have z_gt_x: gt z x
  using trans z_in_a yy_in_a x_in_a z_gt_yy yy_gt_x by blast

have count (mset zs) z ≤ count (mset ys) z
  using max_yy[rule_format, of z] z_in_a yy_in_a z_gt_yy z_gt_x by blast
moreover
{
  assume count (mset zs) z < count (mset ys) z
  then obtain u where u_gt_z: gt u z and cnt_u: count (mset ys) u < count (mset zs) u
  using zs_gt_ys[unfolded msetext_huet_def Let_def] by blast
  have u_in_a: u ∈ A
  using cnt_u zs_a dual_order.strict_trans2 by fastforce
  have u_gt_yy: gt u yy
  using trans u_in_a z_in_a yy_in_a u_gt_z z_gt_yy by blast
  have u_gt_x: gt u x
  using trans u_in_a z_in_a x_in_a u_gt_z z_gt_x by blast
  have False
  using max_yy[rule_format, of u] u_in_a yy_in_a u_gt_yy u_gt_x cnt_u by fastforce
}
ultimately have count (mset zs) z = count (mset ys) z
  by fastforce
thus ?thesis
  using z_gt_x cnt_z by fastforce
qed
}
moreover
{
  assume count (mset zs) x ≥ count (mset ys) x
  hence count (mset ys) x < count (mset xs) x
  using cnt_x_xz by fastforce
  then obtain y where y_gt_x: gt y x and cnt_y: count (mset ys) y > count (mset xs) y
  using ys_gt_xs[unfolded msetext_huet_def Let_def] by blast
  have y_in_a: y ∈ A
  using cnt_y ys_a dual_order.strict_trans2 by fastforce

obtain yy where
  yy_gt_x: gt yy x and
  cnt_yy: count (mset ys) yy > count (mset xs) yy and
  max_yy: ∀ y ∈ A. yy ∈ A → gt y yy → gt y x → count (mset xs) y ≥ count (mset ys) y
  using wfP_eq_minimal[THEN iffD1, OF wf_gt, rule_format,
    of y {y. gt y x ∧ count (mset xs) y < count (mset ys) y}, simplified]
  y_gt_x cnt_y
  by force
have yy_in_a: yy ∈ A
  using cnt_yy ys_a dual_order.strict_trans2 by fastforce

have ?case
proof (cases count (mset zs) yy ≥ count (mset ys) yy)
  case True
  thus ?thesis
  using yy_gt_x cnt_yy by fastforce
next
  case False
  hence count (mset zs) yy < count (mset ys) yy
  by fastforce
  then obtain z where z_gt_yy: gt z yy and cnt_z: count (mset zs) z > count (mset ys) z
  using zs_gt_ys[unfolded msetext_huet_def Let_def] by blast

```

```

have z_in_a: z ∈ A
  using cnt_z zs_a dual_order.strict_trans2 by fastforce
have z_gt_x: gt z x
  using trans z_in_a yy_in_a x_in_a z_gt_yy yy_gt_x by blast

have count (mset ys) z ≤ count (mset xs) z
  using max_yy[rule_format, of z] z_in_a yy_in_a z_gt_yy z_gt_x by blast
moreover
{
  assume count (mset ys) z < count (mset xs) z
  then obtain u where u_gt_z: gt u z and cnt_u: count (mset xs) u < count (mset ys) u
    using ys_gt_xs[unfolded msetext_huet_def Let_def] by blast
  have u_in_a: u ∈ A
    using cnt_u ys_a dual_order.strict_trans2 by fastforce
  have u_gt_yy: gt u yy
    using trans u_in_a z_in_a yy_in_a u_gt_z z_gt_yy by blast
  have u_gt_x: gt u x
    using trans u_in_a z_in_a x_in_a u_gt_z z_gt_x by blast
  have False
    using max_yy[rule_format, of u] u_in_a yy_in_a u_gt_yy u_gt_x cnt_u by fastforce
}
ultimately have count (mset ys) z = count (mset xs) z
  by fastforce
thus ?thesis
  using z_gt_x cnt_z by fastforce
qed
}
ultimately show ∃y. gt y x ∧ count (mset xs) y < count (mset zs) y
  by fastforce
qed
qed

```

**lemma** *msetext\_huet\_snoc*: *msetext\_huet gt (xs @ [x]) xs*  
*unfolding msetext\_huet\_def Let\_def by simp*

**lemma** *msetext\_huet\_compat\_cons*: *msetext\_huet gt ys xs ⇒ msetext\_huet gt (x # ys) (x # xs)*  
*unfolding msetext\_huet\_def Let\_def by auto*

**lemma** *msetext\_huet\_compat\_snoc*: *msetext\_huet gt ys xs ⇒ msetext\_huet gt (ys @ [x]) (xs @ [x])*  
*unfolding msetext\_huet\_def Let\_def by auto*

**lemma** *msetext\_huet\_compat\_list*: *y ≠ x ⇒ gt y x ⇒ msetext\_huet gt (xs @ y # xs') (xs @ x # xs')*  
*unfolding msetext\_huet\_def Let\_def by auto*

**lemma** *msetext\_huet\_singleton*: *y ≠ x ⇒ msetext\_huet gt [y] [x] ↔ gt y x*  
*unfolding msetext\_huet\_def by simp*

**lemma** *msetext\_huet\_wf*: *wfP (λx y. gt y x) ⇒ wfP (λxs ys. msetext\_huet gt ys xs)*  
*by (erule wfP\_subset[OF msetext\_dersh\_wf]) (auto intro: msetext\_huet\_imp\_dersh)*

**lemma** *msetext\_huet\_hd\_or\_tl*:  
**assumes**  
*trans*:  $\forall z y x. gt z y \longrightarrow gt y x \longrightarrow gt z x$  **and**  
*total*:  $\forall y x. gt y x \vee gt x y \vee y = x$  **and**  
*len\_eq*: *length ys = length xs* **and**  
*sys\_gt\_xs*: *msetext\_huet gt (y # ys) (x # xs)*  
**shows** *gt y x ∨ msetext\_huet gt ys xs*

**proof** –  
**let** ?Y = *mset (y # ys)*  
**let** ?X = *mset (x # xs)*

**let** ?Ya = *mset ys*  
**let** ?Xa = *mset xs*

```

have Y_ne_X: ?Y ≠ ?X and
  ex_gt_Y:  $\bigwedge xa. \text{count } ?X \text{ } xa > \text{count } ?Y \text{ } xa \implies \exists ya. \text{gt } ya \text{ } xa \wedge \text{count } ?Y \text{ } ya > \text{count } ?X \text{ } ya$ 
  using yys_gt_xs[unfolded msetext_huet_def Let_def] by auto
obtain yy where
  yy:  $\bigwedge xa. \text{count } ?X \text{ } xa > \text{count } ?Y \text{ } xa \implies \text{gt } (yy \text{ } xa) \text{ } xa \wedge \text{count } ?Y \text{ } (yy \text{ } xa) > \text{count } ?X \text{ } (yy \text{ } xa)$ 
  using ex_gt_Y by metis

have cnt_Y_pres:  $\text{count } ?Y a \text{ } xa > \text{count } ?X a \text{ } xa$  if  $\text{count } ?Y \text{ } xa > \text{count } ?X \text{ } xa$  and  $xa \neq y$  for  $xa$ 
  using that by (auto split: if_splits)
have cnt_X_pres:  $\text{count } ?X a \text{ } xa > \text{count } ?Y a \text{ } xa$  if  $\text{count } ?X \text{ } xa > \text{count } ?Y \text{ } xa$  and  $xa \neq x$  for  $xa$ 
  using that by (auto split: if_splits)

{
  assume y_eq_x:  $y = x$ 
  have ?Xa ≠ ?Ya
    using y_eq_x Y_ne_X by simp
  moreover have  $\bigwedge xa. \text{count } ?X a \text{ } xa > \text{count } ?Y a \text{ } xa \implies \exists ya. \text{gt } ya \text{ } xa \wedge \text{count } ?Y a \text{ } ya > \text{count } ?X a \text{ } ya$ 
  proof -
    fix xa :: 'a
    assume a1:  $\text{count } (mset \text{ } ys) \text{ } xa < \text{count } (mset \text{ } xs) \text{ } xa$ 
    from ex_gt_Y obtain aa :: 'a  $\Rightarrow$  'a where
      f3:  $\forall a. \neg \text{count } (mset \text{ } (y \# ys)) \text{ } a < \text{count } (mset \text{ } (x \# xs)) \text{ } a \vee \text{gt } (aa \text{ } a) \text{ } a \wedge$ 
         $\text{count } (mset \text{ } (x \# xs)) \text{ } (aa \text{ } a) < \text{count } (mset \text{ } (y \# ys)) \text{ } (aa \text{ } a)$ 
      by (metis (full_types))
    then have f4:  $\bigwedge a. \text{count } (mset \text{ } (x \# xs)) \text{ } (aa \text{ } a) < \text{count } (mset \text{ } (x \# ys)) \text{ } (aa \text{ } a) \vee$ 
       $\neg \text{count } (mset \text{ } (x \# ys)) \text{ } a < \text{count } (mset \text{ } (x \# xs)) \text{ } a$ 
      using y_eq_x by meson
    have  $\bigwedge a \text{ as } aa. \text{count } (mset \text{ } ((a::'a) \# as)) \text{ } aa = \text{count } (mset \text{ } as) \text{ } aa \vee aa = a$ 
      by fastforce
    then have  $xa = x \vee \text{count } (mset \text{ } (x \# xs)) \text{ } (aa \text{ } xa) < \text{count } (mset \text{ } (x \# ys)) \text{ } (aa \text{ } xa)$ 
      using f4 a1 by (metis (no_types))
    then show  $\exists a. \text{gt } a \text{ } xa \wedge \text{count } (mset \text{ } xs) \text{ } a < \text{count } (mset \text{ } ys) \text{ } a$ 
      using f3 y_eq_x a1 by (metis (no_types) Suc_less_eq count_add_mset mset.simps(2))
  qed
  ultimately have msetext_huet_gt_ys_xs
    unfolding msetext_huet_def Let_def by simp
}
moreover
{
  assume x_gt_y:  $\text{gt } x \text{ } y$  and y_ngt_x:  $\neg \text{gt } y \text{ } x$ 
  hence y_ne_x:  $y \neq x$ 
  by fast
}

obtain z where z_cnt:  $\text{count } ?X \text{ } z > \text{count } ?Y \text{ } z$ 
  using size_eq_ex_count_lt[of ?Y ?X] size_mset size_mset len_eq Y_ne_X by auto

have Xa_ne_Ya:  $?X a \neq ?Y a$ 
proof (cases  $z = x$ )
  case True
  hence yy z  $\neq$  y
    using y_ngt_x yy z_cnt by blast
  hence  $\text{count } ?Y a \text{ } (yy \text{ } z) > \text{count } ?X a \text{ } (yy \text{ } z)$ 
    using cnt_Y_pres yy z_cnt by blast
  thus ?thesis
    by auto
  next
  case False
  hence  $\text{count } ?X a \text{ } z > \text{count } ?Y a \text{ } z$ 
    using z_cnt cnt_X_pres by blast
  thus ?thesis
    by auto
qed

```



```

have  $\exists ya. gt\ ya\ xa \wedge count\ ?Ya\ ya > count\ ?Xa\ ya$ 
  if  $xa\_cnt: count\ ?Xa\ xa > count\ ?Ya\ xa$  for  $xa$ 
proof (cases  $xa = y$ )
  case  $xa\_eq\_y: True$ 

  {
    assume  $count\ ?Ya\ x > count\ ?Xa\ x$ 
    moreover have  $gt\ x\ xa$ 
      unfolding  $xa\_eq\_y$  by (rule  $x\_gt\_y$ )
    ultimately have  $?thesis$ 
      by fast
  }
moreover
  {
    assume  $count\ ?Xa\ x \geq count\ ?Ya\ x$ 
    hence  $x\_cnt: count\ ?X\ x > count\ ?Y\ x$ 
      by (simp add: y\_ne\_x)
    hence  $yyx\_gt\_x: gt\ (yy\ x)\ x$  and  $yyx\_cnt: count\ ?Y\ (yy\ x) > count\ ?X\ (yy\ x)$ 
      using  $yy$  by blast+

    have  $yyx\_ne\_y: yy\ x \neq y$ 
      using  $y\_ngt\_x\ yyx\_gt\_x$  by auto

    have  $gt\ (yy\ x)\ xa$ 
      unfolding  $xa\_eq\_y$  using  $trans\ yyx\_gt\_x\ x\_gt\_y$  by blast
    moreover have  $count\ ?Ya\ (yy\ x) > count\ ?Xa\ (yy\ x)$ 
      using  $cnt\_Y\_pres\ yyx\_cnt\ yyx\_ne\_y$  by blast
    ultimately have  $?thesis$ 
      by blast
  }
ultimately show  $?thesis$ 
  by fastforce
next
case False
hence  $xa\_cnt: count\ ?X\ xa > count\ ?Y\ xa$ 
  using  $xa\_cnta$  by fastforce

show  $?thesis$ 
proof (cases  $yy\ xa = y \wedge count\ ?Ya\ y \leq count\ ?Xa\ y$ )
  case  $yyxa\_ne\_y\_or: False$ 

  have  $yyxa\_gt\_xa: gt\ (yy\ xa)\ xa$  and  $yyxa\_cnt: count\ ?Y\ (yy\ xa) > count\ ?X\ (yy\ xa)$ 
    using  $yy[OF\ xa\_cnt]$  by blast+

  have  $count\ ?Ya\ (yy\ xa) > count\ ?Xa\ (yy\ xa)$ 
    using  $cnt\_Y\_pres\ yyxa\_cnt\ yyxa\_ne\_y\_or$  by fastforce
  thus  $?thesis$ 
    using  $yyxa\_gt\_xa$  by blast
next
case True
note  $yyxa\_eq\_y = this[THEN\ conjunct1]$  and  $y\_cnt = this[THEN\ conjunct2]$ 

  {
    assume  $count\ ?Ya\ x > count\ ?Xa\ x$ 
    moreover have  $gt\ x\ xa$ 
      using  $trans\ x\_gt\_y\ xa\_cnt\ yy\ yyxa\_eq\_y$  by blast
    ultimately have  $?thesis$ 
      by fast
  }
moreover
  {
    assume  $count\ ?Xa\ x \geq count\ ?Ya\ x$ 

```

```

hence  $x\_cnt$ : count ?X  $x > count ?Y x$ 
  by (simp add:  $y\_ne\_x$ )
hence  $yyx\_gt\_x$ :  $gt (yy x) x$  and  $yyx\_cnt$ : count ?Y  $(yy x) > count ?X (yy x)$ 
  using  $yy$  by blast+

have  $yyx\_ne\_y$ :  $yy x \neq y$ 
  using  $y\_ngt\_x$   $yyx\_gt\_x$  by auto

have  $gt (yy x) xa$ 
  using  $trans x\_gt\_y xa\_cnt yy yyx\_gt\_x yyxa\_eq\_y$  by blast
moreover have  $count ?Ya (yy x) > count ?Xa (yy x)$ 
  using  $cnt\_Y\_pres yyx\_cnt yyx\_ne\_y$  by blast
ultimately have ?thesis
  by blast
}
ultimately show ?thesis
  by fastforce
qed
qed
hence  $msetext\_huet$   $gt$   $ys$   $xs$ 
  unfolding  $msetext\_huet\_def$   $Let\_def$  using  $Xa\_ne\_Ya$  by fast
}
ultimately show ?thesis
  using  $total$  by blast
qed

```

```

interpretation  $msetext\_huet$ : ext  $msetext\_huet$ 
  by standard (fact  $msetext\_huet\_mono\_strong$ , fact  $msetext\_huet\_map$ )

```

```

interpretation  $msetext\_huet$ : ext_irrefl_before_trans  $msetext\_huet$ 
  by standard (fact  $msetext\_huet\_irrefl$ , fact  $msetext\_huet\_trans\_from\_irrefl$ )

```

```

interpretation  $msetext\_huet$ : ext_snoc  $msetext\_huet$ 
  by standard (fact  $msetext\_huet\_snoc$ )

```

```

interpretation  $msetext\_huet$ : ext_compat_cons  $msetext\_huet$ 
  by standard (fact  $msetext\_huet\_compat\_cons$ )

```

```

interpretation  $msetext\_huet$ : ext_compat_snoc  $msetext\_huet$ 
  by standard (fact  $msetext\_huet\_compat\_snoc$ )

```

```

interpretation  $msetext\_huet$ : ext_compat_list  $msetext\_huet$ 
  by standard (fact  $msetext\_huet\_compat\_list$ )

```

```

interpretation  $msetext\_huet$ : ext_singleton  $msetext\_huet$ 
  by standard (fact  $msetext\_huet\_singleton$ )

```

```

interpretation  $msetext\_huet$ : ext_wf  $msetext\_huet$ 
  by standard (fact  $msetext\_huet\_wf$ )

```

```

interpretation  $msetext\_huet$ : ext_hd_or_tl  $msetext\_huet$ 
  by standard (rule  $msetext\_huet\_hd\_or\_tl$ )

```

```

interpretation  $msetext\_huet$ : ext_wf_bounded  $msetext\_huet$ 
  by standard

```

## 5.9 Componentwise Extension

```

definition  $cwiseext$  :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool where
   $cwiseext$   $gt$   $ys$   $xs$   $\longleftrightarrow$   $length$   $ys = length$   $xs$ 
   $\wedge (\forall i < length$   $ys$ .  $gt$   $(ys ! i)$   $(xs ! i) \vee ys ! i = xs ! i)$ 
   $\wedge (\exists i < length$   $ys$ .  $gt$   $(ys ! i)$   $(xs ! i))$ 

```

```

lemma  $cwiseext\_imp\_len\_lexext$ :

```

```

assumes cw: wiseext gt ys xs
shows len_lexext gt ys xs
proof –
  have len_eq: length ys = length xs
    using cw[unfolded wiseext_def] by sat
  moreover have lexext gt ys xs
  proof –
    obtain j where
      j_len: j < length ys and
      j_gt: gt (ys ! j) (xs ! j)
      using cw[unfolded wiseext_def] by blast
    then obtain j0 where
      j0_len: j0 < length ys and
      j0_gt: gt (ys ! j0) (xs ! j0) and
      j0_min:  $\bigwedge i. i < j0 \implies \neg \text{gt } (ys ! i) (xs ! i)$ 
      using wf_eq_minimal[THEN iffD1, OF wf_less, rule_format, of _ {i. gt (ys ! i) (xs ! i)},
        simplified, OF j_gt]
      by (metis less_trans nat_neq_iff)

  have j0_eq:  $\bigwedge i. i < j0 \implies ys ! i = xs ! i$ 
    using cw[unfolded wiseext_def] by (metis j0_len j0_min less_trans)

  have lexext_gt (drop j0 ys) (drop j0 xs)
    using lexext_Cons[of gt _ _ drop (Suc j0) ys drop (Suc j0) xs, OF j0_gt]
    by (metis Cons_nth_drop_Suc j0_len len_eq)
  thus ?thesis
    using cw len_eq j0_len j0_min
  proof (induct j0 arbitrary: ys xs)
    case (Suc k)
    note ih0 = this(1) and gts_dropSk = this(2) and cw = this(3) and len_eq = this(4) and
      Sk_len = this(5) and Sk_min = this(6)

    have Sk_eq:  $\bigwedge i. i < \text{Suc } k \implies ys ! i = xs ! i$ 
      using cw[unfolded wiseext_def] by (metis Sk_len Sk_min less_trans)

    have k_len: k < length ys
      using Sk_len by simp
    have k_min:  $\bigwedge i. i < k \implies \neg \text{gt } (ys ! i) (xs ! i)$ 
      using Sk_min by simp

    have k_eq:  $\bigwedge i. i < k \implies ys ! i = xs ! i$ 
      using Sk_eq by simp

    note ih = ih0[OF _ cw len_eq k_len k_min]

    show ?case
    proof (cases k < length ys)
      case k_lt_ys: True
      note k_lt_xs = k_lt_ys[unfolded len_eq]

      obtain x where x: x = xs ! k
        by simp
      hence y: x = ys ! k
        using Sk_eq[of k] by simp

      have dropk_xs: drop k xs = x # drop (Suc k) xs
        using k_lt_xs x by (simp add: Cons_nth_drop_Suc)
      have dropk_ys: drop k ys = x # drop (Suc k) ys
        using k_lt_ys y by (simp add: Cons_nth_drop_Suc)

      show ?thesis
        by (rule ih, unfold dropk_xs dropk_ys, rule lexext_Cons_eq[OF gts_dropSk])
    next

```

```

    case False
    hence drop k xs = [] and drop k ys = []
      using len_eq by simp_all
    hence lexext gt [] []
      using gts_dropSk by simp
    hence lexext gt (drop k ys) (drop k xs)
      by simp
    thus ?thesis
      by (rule ih)
  qed
  qed simp
  qed
  ultimately show ?thesis
    unfolding lenext_def by sat
  qed

```

```

lemma wiseext_mono_strong:
  ( $\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y \ x \longrightarrow \text{gt}' \ y \ x$ )  $\implies$  wiseext gt ys xs  $\implies$  wiseext gt' ys xs
  unfolding wiseext_def by (induct, force, fast)

```

```

lemma wiseext_map_strong:
  ( $\forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } y \ x \longrightarrow \text{gt } (f \ y) \ (f \ x)$ )  $\implies$  wiseext gt ys xs  $\implies$ 
  wiseext gt (map f ys) (map f xs)
  unfolding wiseext_def by auto

```

```

lemma wiseext_irrefl: ( $\forall x \in \text{set } xs. \neg \text{gt } x \ x$ )  $\implies$   $\neg$  wiseext gt xs xs
  unfolding wiseext_def by (blast intro: nth_mem)

```

```

lemma wiseext_trans_strong:
  assumes
     $\forall z \in \text{set } zs. \forall y \in \text{set } ys. \forall x \in \text{set } xs. \text{gt } z \ y \longrightarrow \text{gt } y \ x \longrightarrow \text{gt } z \ x$  and
    wiseext gt zs ys and wiseext gt ys xs
  shows wiseext gt zs xs
  using assms unfolding wiseext_def by (metis (mono_tags) nth_mem)

```

```

lemma wiseext_compat_cons: wiseext gt ys xs  $\implies$  wiseext gt (x # ys) (x # xs)
  unfolding wiseext_def

```

```

proof (elim conjE, intro conjI)
  assume
    length ys = length xs and
     $\forall i < \text{length } ys. \text{gt } (ys \ ! \ i) \ (xs \ ! \ i) \vee \text{ys} \ ! \ i = \text{xs} \ ! \ i$ 
  thus  $\forall i < \text{length } (x \ \# \ ys). \text{gt } ((x \ \# \ ys) \ ! \ i) \ ((x \ \# \ xs) \ ! \ i) \vee (x \ \# \ ys) \ ! \ i = (x \ \# \ xs) \ ! \ i$ 
    by (simp add: nth_Cons')
  next
  assume  $\exists i < \text{length } ys. \text{gt } (ys \ ! \ i) \ (xs \ ! \ i)$ 
  thus  $\exists i < \text{length } (x \ \# \ ys). \text{gt } ((x \ \# \ ys) \ ! \ i) \ ((x \ \# \ xs) \ ! \ i)$ 
    by fastforce
  qed auto

```

```

lemma wiseext_compat_snoc: wiseext gt ys xs  $\implies$  wiseext gt (ys @ [x]) (xs @ [x])
  unfolding wiseext_def

```

```

proof (elim conjE, intro conjI)
  assume
    length ys = length xs and
     $\forall i < \text{length } ys. \text{gt } (ys \ ! \ i) \ (xs \ ! \ i) \vee \text{ys} \ ! \ i = \text{xs} \ ! \ i$ 
  thus  $\forall i < \text{length } (ys \ @ \ [x]).$ 
     $\text{gt } ((ys \ @ \ [x]) \ ! \ i) \ ((xs \ @ \ [x]) \ ! \ i) \vee (ys \ @ \ [x]) \ ! \ i = (xs \ @ \ [x]) \ ! \ i$ 
    by (simp add: nth_append)
  next
  assume
    length ys = length xs and
     $\exists i < \text{length } ys. \text{gt } (ys \ ! \ i) \ (xs \ ! \ i)$ 
  thus  $\exists i < \text{length } (ys \ @ \ [x]). \text{gt } ((ys \ @ \ [x]) \ ! \ i) \ ((xs \ @ \ [x]) \ ! \ i)$ 

```

by (metis length\_append\_singleton less\_Suc\_eq nth\_append)  
qed auto

lemma wiseext\_compat\_list:

assumes  $y\_gt\_x: gt\ y\ x$   
shows  $wiseext\ gt\ (xs\ @\ y\ \# \ xs')\ (xs\ @\ x\ \# \ xs')$   
unfolding wiseext\_def

proof (intro conjI)

show  $\forall i < length\ (xs\ @\ y\ \# \ xs').\ gt\ ((xs\ @\ y\ \# \ xs')\ !\ i)\ ((xs\ @\ x\ \# \ xs')\ !\ i)$   
 $\vee\ (xs\ @\ y\ \# \ xs')\ !\ i = (xs\ @\ x\ \# \ xs')\ !\ i$   
using  $y\_gt\_x$  by (simp add: nth\_Cons' nth\_append)

next

show  $\exists i < length\ (xs\ @\ y\ \# \ xs').\ gt\ ((xs\ @\ y\ \# \ xs')\ !\ i)\ ((xs\ @\ x\ \# \ xs')\ !\ i)$   
using  $y\_gt\_x$  by (metis add\_diff\_cancel\_right' append\_is\_Nil\_conv diff\_less length\_append  
length\_greater\_0\_conv list.simps(3) nth\_append\_length)

qed auto

lemma wiseext\_singleton:  $wiseext\ gt\ [y]\ [x] \longleftrightarrow gt\ y\ x$

unfolding wiseext\_def by auto

lemma wiseext\_wf:  $wfP\ (\lambda x\ y.\ gt\ y\ x) \implies wfP\ (\lambda xs\ ys.\ wiseext\ gt\ ys\ xs)$

by (auto intro: wiseext\_imp\_len\_lexext wfP\_subset[OF len\_lexext\_wf])

lemma wiseext\_hd\_or\_tl:  $wiseext\ gt\ (y\ \# \ ys)\ (x\ \# \ xs) \implies gt\ y\ x \vee wiseext\ gt\ ys\ xs$

unfolding wiseext\_def

proof (elim conjE, intro disj\_imp[THEN iffD2, rule\_format] conjI)

assume

$\exists i < length\ (y\ \# \ ys).\ gt\ ((y\ \# \ ys)\ !\ i)\ ((x\ \# \ xs)\ !\ i)$  and  
 $\neg gt\ y\ x$

thus  $\exists i < length\ ys.\ gt\ (ys\ !\ i)\ (xs\ !\ i)$

by (metis (no\_types) One\_nat\_def diff\_le\_self diff\_less dual\_order.strict\_trans2  
length\_Cons less\_Suc\_eq linorder\_neqE\_nat not\_less0 nth\_Cons')

qed auto

locale ext\_wiseext = ext\_compat\_list + ext\_compat\_cons

begin

context

fixes  $gt :: 'a \Rightarrow 'a \Rightarrow bool$

assumes

$gt\_irrefl: \neg gt\ x\ x$  and

$trans\_gt: ext\ gt\ zs\ ys \implies ext\ gt\ ys\ xs \implies ext\ gt\ zs\ xs$

begin

lemma

assumes  $ys\_gtcw\_xs: wiseext\ gt\ ys\ xs$

shows  $ext\ gt\ ys\ xs$

proof -

have  $length\ ys = length\ xs$

by (rule  $ys\_gtcw\_xs$ [unfolded wiseext\_def, THEN conjunct1])

thus ?thesis

using  $ys\_gtcw\_xs$

proof (induct rule: list\_induct2)

case Nil

thus ?case

unfolding wiseext\_def by simp

next

case (Cons y ys x xs)

note  $len\_ys\_eq\_xs = this(1)$  and  $ih = this(2)$  and  $ys\_gtcw\_xs = this(3)$

have  $xs\_gts\_xxs: ext\ gt\ (x\ \# \ ys)\ (x\ \# \ xs)$  if  $ys\_ne\_xs: ys \neq xs$

proof -

have  $ys\_gtcw\_xs: wiseext\ gt\ ys\ xs$

```

    using yys_gtcw_xxs unfolding cwiseext_def
  proof (elim conjE, intro conjI)
    assume
       $\forall i < \text{length } (y \# ys). \text{gt } ((y \# ys) ! i) ((x \# xs) ! i) \vee (y \# ys) ! i = (x \# xs) ! i$ 
    hence ge:  $\forall i < \text{length } ys. \text{gt } (ys ! i) (xs ! i) \vee ys ! i = xs ! i$ 
      by auto
    thus  $\exists i < \text{length } ys. \text{gt } (ys ! i) (xs ! i)$ 
      using ys_ne_xs len_ys_eq_xs nth_equalityI by blast
  qed auto
  hence ext gt ys xs
    by (rule ih)
  thus ext gt (x # ys) (x # xs)
    by (rule compat_cons)
qed

have gt y x  $\vee y = x$ 
  using yys_gtcw_xxs unfolding cwiseext_def by fastforce
moreover
{
  assume y_eq_x:  $y = x$ 
  have ?case
  proof (cases ys = xs)
    case True
      hence False
        using y_eq_x gt_irrefl yys_gtcw_xxs unfolding cwiseext_def by presburger
      thus ?thesis
        by sat
    next
      case False
      thus ?thesis
        using y_eq_x xys_gts_xxs by simp
  qed
}
moreover
{
  assume  $y \neq x$  and gt y x
  hence yys_gts_xys: ext gt (y # ys) (x # ys)
    using compat_list[of _ _ gt []] by simp

  have ?case
  proof (cases ys = xs)
    case ys_eq_xs: True
      thus ?thesis
        using yys_gts_xys by simp
    next
      case False
      thus ?thesis
        using yys_gts_xys xys_gts_xxs trans_gt by blast
  qed
}
ultimately show ?case
  by sat
qed
qed

end

end

interpretation cwiseext: ext cwiseext
  by standard (fact cwiseext_mono_strong, rule cwiseext_map_strong, metis in_listsD)

interpretation cwiseext: ext_irrefl_trans_strong cwiseext

```

```

by standard (fact wiseext_irrefl, fact wiseext_trans_strong)

interpretation wiseext: ext_compat_cons wiseext
  by standard (fact wiseext_compat_cons)

interpretation wiseext: ext_compat_snoc wiseext
  by standard (fact wiseext_compat_snoc)

interpretation wiseext: ext_compat_list wiseext
  by standard (rule wiseext_compat_list)

interpretation wiseext: ext_singleton wiseext
  by standard (rule wiseext_singleton)

interpretation wiseext: ext_wf wiseext
  by standard (rule wiseext_wf)

interpretation wiseext: ext_hd_or_tl wiseext
  by standard (rule wiseext_hd_or_tl)

interpretation wiseext: ext_wf_bounded wiseext
  by standard

end

```

## 6 The Applicative Recursive Path Order for Lambda-Free Higher-Order Terms

```

theory Lambda_Free_RPO_App
imports Lambda_Free_Term_Extension_Orders
abbrevs
  >t = >t
  ≥t = ≥t
begin

```

This theory defines the applicative recursive path order (RPO), a variant of RPO for  $\lambda$ -free higher-order terms. It corresponds to the order obtained by applying the standard first-order RPO on the applicative encoding of higher-order terms and assigning the lowest precedence to the application symbol.

```

locale rpo_app = gt_sym op >s
  for gt_sym :: 's ⇒ 's ⇒ bool (infix >s 50) +
  fixes ext :: (('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool) ⇒ ('s, 'v) tm list ⇒ ('s, 'v) tm list ⇒ bool
  assumes
    ext_ext_trans_before_irrefl: ext_trans_before_irrefl ext and
    ext_ext_compat_list: ext_compat_list ext
begin

```

```

lemma ext_mono[mono]: gt ≤ gt' ⇒ ext gt ≤ ext gt'
  by (simp add: ext_mono ext_ext_compat_list[unfolded ext_compat_list_def, THEN conjunct1])

```

```

inductive gt :: ('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool (infix >t 50) where
  | gt_sub: is_App t ⇒ (fun t >t s ∨ fun t = s) ∨ (arg t >t s ∨ arg t = s) ⇒ t >t s
  | gt_sym_sym: g >s f ⇒ Hd (Sym g) >t Hd (Sym f)
  | gt_sym_app: Hd (Sym g) >t s1 ⇒ Hd (Sym g) >t s2 ⇒ Hd (Sym g) >t App s1 s2
  | gt_app_app: ext (op >t) [t1, t2] [s1, s2] ⇒ App t1 t2 >t s1 ⇒ App t1 t2 >t s2 ⇒
    App t1 t2 >t App s1 s2

```

```

abbreviation ge :: ('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool (infix ≥t 50) where
  t ≥t s ≡ t >t s ∨ t = s

```

```
end
```

```
end
```

## 7 The Graceful Recursive Path Order for Lambda-Free Higher-Order Terms

```

theory Lambda_Free_RPO_Std
imports Lambda_Free_Term_Extension_Orders
abbrevs
  >t = >t
  ≥t = ≥t
begin

```

This theory defines the graceful recursive path order (RPO) for  $\lambda$ -free higher-order terms.

### 7.1 Setup

```

locale rpo_basis = ground_heads op >s arity_sym arity_var
  for
    gt_sym :: 's ⇒ 's ⇒ bool (infix >s 50) and
    arity_sym :: 's ⇒ enat and
    arity_var :: 'v ⇒ enat +
  fixes
    extf :: 's ⇒ (('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool) ⇒ ('s, 'v) tm list ⇒ ('s, 'v) tm list ⇒ bool
  assumes
    extf_ext_trans_before_irrefl: ext_trans_before_irrefl (extf) and
    extf_ext_compat_cons: ext_compat_cons (extf) and
    extf_ext_compat_list: ext_compat_list (extf)
begin

lemma extf_ext_trans: ext_trans (extf)
  by (rule ext_trans_before_irrefl.axioms(1)[OF extf_ext_trans_before_irrefl])

lemma extf_ext: ext (extf)
  by (rule ext_trans.axioms(1)[OF extf_ext_trans])

lemmas extf_mono_strong = ext_mono_strong[OF extf_ext]
lemmas extf_mono = ext_mono[OF extf_ext, mono]
lemmas extf_map = ext_map[OF extf_ext]
lemmas extf_trans = ext_trans.trans[OF extf_ext_trans]
lemmas extf_irrefl_from_trans =
  ext_trans_before_irrefl.irrefl_from_trans[OF extf_ext_trans_before_irrefl]
lemmas extf_compat_append_left = ext_compat_cons.compat_append_left[OF extf_ext_compat_cons]
lemmas extf_compat_list = ext_compat_list.compat_list[OF extf_ext_compat_list]

definition chkvar :: ('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool where
  [simp]: chkvar t s ⇔ vars_hd (head s) ⊆ vars t

end

locale rpo = rpo_basis _ _ arity_sym arity_var
  for
    arity_sym :: 's ⇒ enat and
    arity_var :: 'v ⇒ enat
begin

```

### 7.2 Inductive Definitions

```

definition
  chksubs :: (('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool) ⇒ ('s, 'v) tm ⇒ ('s, 'v) tm ⇒ bool
where
  [simp]: chksubs gt t s ⇔ (case s of App s1 s2 ⇒ gt t s1 ∧ gt t s2 | _ ⇒ True)

lemma chksubs_mono[mono]: gt ≤ gt' ⇒ chksubs gt ≤ chksubs gt'
  by (auto simp: tm.case_eq_if) force+
```



**inductive**  $gt :: ('s, 'v) tm \Rightarrow ('s, 'v) tm \Rightarrow bool$  (**infix**  $>_t$  50) **where**  
 $gt\_sub: is\_App\ t \Longrightarrow (fun\ t\ >_t\ s \vee fun\ t = s) \vee (arg\ t\ >_t\ s \vee arg\ t = s) \Longrightarrow t\ >_t\ s$   
 $gt\_diff: head\ t\ >_{hd}\ head\ s \Longrightarrow chkvar\ t\ s \Longrightarrow chksubs\ (op\ >_t)\ t\ s \Longrightarrow t\ >_t\ s$   
 $gt\_same: head\ t = head\ s \Longrightarrow chksubs\ (op\ >_t)\ t\ s \Longrightarrow$   
 $(\forall f \in ground\_heads\ (head\ t).\ extf\ f\ (op\ >_t)\ (args\ t)\ (args\ s)) \Longrightarrow t\ >_t\ s$

**abbreviation**  $ge :: ('s, 'v) tm \Rightarrow ('s, 'v) tm \Rightarrow bool$  (**infix**  $\geq_t$  50) **where**  
 $t\ \geq_t\ s \equiv t\ >_t\ s \vee t = s$

**inductive**  $gt\_sub :: ('s, 'v) tm \Rightarrow ('s, 'v) tm \Rightarrow bool$  **where**  
 $gt\_subI: is\_App\ t \Longrightarrow fun\ t\ \geq_t\ s \vee arg\ t\ \geq_t\ s \Longrightarrow gt\_sub\ t\ s$

**inductive**  $gt\_diff :: ('s, 'v) tm \Rightarrow ('s, 'v) tm \Rightarrow bool$  **where**  
 $gt\_diffI: head\ t\ >_{hd}\ head\ s \Longrightarrow chkvar\ t\ s \Longrightarrow chksubs\ (op\ >_t)\ t\ s \Longrightarrow gt\_diff\ t\ s$

**inductive**  $gt\_same :: ('s, 'v) tm \Rightarrow ('s, 'v) tm \Rightarrow bool$  **where**  
 $gt\_sameI: head\ t = head\ s \Longrightarrow chksubs\ (op\ >_t)\ t\ s \Longrightarrow$   
 $(\forall f \in ground\_heads\ (head\ t).\ extf\ f\ (op\ >_t)\ (args\ t)\ (args\ s)) \Longrightarrow gt\_same\ t\ s$

**lemma**  $gt\_iff\_sub\_diff\_same: t\ >_t\ s \longleftrightarrow gt\_sub\ t\ s \vee gt\_diff\ t\ s \vee gt\_same\ t\ s$   
**by** ( $subst\ gt.simps$ ) ( $auto\ simp: gt\_sub.simps\ gt\_diff.simps\ gt\_same.simps$ )

### 7.3 Transitivity

**lemma**  $gt\_fun\_imp: fun\ t\ >_t\ s \Longrightarrow t\ >_t\ s$   
**by** ( $cases\ t$ ) ( $auto\ intro: gt\_sub$ )

**lemma**  $gt\_arg\_imp: arg\ t\ >_t\ s \Longrightarrow t\ >_t\ s$   
**by** ( $cases\ t$ ) ( $auto\ intro: gt\_sub$ )

**lemma**  $gt\_imp\_vars: t\ >_t\ s \Longrightarrow vars\ t \supseteq vars\ s$

**proof** ( $simp\ only: atomize\_imp,$   
 $rule\ measure\_induct\_rule[of\ \lambda(t, s). size\ t + size\ s,$   
 $\lambda(t, s). t\ >_t\ s \longrightarrow vars\ t \supseteq vars\ s\ (t, s), simplified\ prod.case],$   
 $simp\ only: split\_paired\_all\ prod.case\ atomize\_imp[symmetric])$   
**fix**  $t\ s$   
**assume**  
 $ih: \bigwedge ta\ sa. size\ ta + size\ sa < size\ t + size\ s \Longrightarrow ta\ >_t\ sa \Longrightarrow vars\ ta \supseteq vars\ sa$  **and**  
 $t\_gt\_s: t\ >_t\ s$   
**show**  $vars\ t \supseteq vars\ s$   
**using**  $t\_gt\_s$   
**proof**  $cases$   
**case**  $gt\_sub$   
**thus**  $?thesis$   
**using**  $ih[of\ fun\ t\ s]\ ih[of\ arg\ t\ s]$   
**by** ( $meson\ add\_less\_cancel\_right\ subsetD\ size\_arg\_lt\ size\_fun\_lt\ subsetI\ tm.set\_sel(5,6)$ )  
**next**  
**case**  $gt\_diff$   
**show**  $?thesis$   
**proof** ( $cases\ s$ )  
**case**  $Hd$   
**thus**  $?thesis$   
**using**  $gt\_diff(2)$  **by** ( $auto\ elim: hd.set\_cases(2)$ )  
**next**  
**case** ( $App\ s1\ s2$ )  
**thus**  $?thesis$   
**using**  $gt\_diff(3)$   $ih[of\ t\ s1]\ ih[of\ t\ s2]$  **by**  $simp$   
**qed**  
**next**  
**case**  $gt\_same$   
**show**  $?thesis$   
**proof** ( $cases\ s$ )  
**case**  $Hd$   
**thus**  $?thesis$

```

    using gt_same(1) vars_head_subseteq by fastforce
  next
  case (App s1 s2)
  thus ?thesis
    using gt_same(2) ih[of t s1] ih[of t s2] by simp
qed
qed
qed

theorem gt_trans:  $u >_t t \implies t >_t s \implies u >_t s$ 
proof (simp only: atomize_imp,
  rule measure_induct_rule[of  $\lambda(u, t, s). \{\#size\ u, size\ t, size\ s\}$ 
     $\lambda(u, t, s). u >_t t \longrightarrow t >_t s \longrightarrow u >_t s (u, t, s)$ ,
    simplified prod.case],
  simp only: split_paired_all prod.case atomize_imp[symmetric])
fix u t s
assume
  ih:  $\bigwedge ua\ ta\ sa. \{\#size\ ua, size\ ta, size\ sa\} < \{\#size\ u, size\ t, size\ s\} \implies$ 
     $ua >_t ta \implies ta >_t sa \implies ua >_t sa$  and
  u_gt_t:  $u >_t t$  and t_gt_s:  $t >_t s$ 

have chkvar: chkvar u s
  by clarsimp (meson u_gt_t t_gt_s gt_imp_vars hd.set_sel(2) vars_head_subseteq subsetCE)

have chk_u_s_if: chksubs (op >_t) u s if chk_t_s: chksubs (op >_t) t s
proof (cases s)
  case (App s1 s2)
  thus ?thesis
    using chk_t_s by (auto intro: ih[of _ _ s1, OF _ u_gt_t] ih[of _ _ s2, OF _ u_gt_t])
qed auto

have
  fun_u_lt_etc:  $is\_App\ u \implies \{\#size\ (fun\ u), size\ t, size\ s\} < \{\#size\ u, size\ t, size\ s\}$  and
  arg_u_lt_etc:  $is\_App\ u \implies \{\#size\ (arg\ u), size\ t, size\ s\} < \{\#size\ u, size\ t, size\ s\}$ 
  by (simp_all add: size_fun_lt size_arg_lt)

have u_gt_s_if_ui:  $is\_App\ u \implies fun\ u \geq_t t \vee arg\ u \geq_t t \implies u >_t s$ 
  using ih[of fun u t s, OF fun_u_lt_etc] ih[of arg u t s, OF arg_u_lt_etc] gt_arg_imp
  gt_fun_imp t_gt_s by blast

show  $u >_t s$ 
  using t_gt_s
proof cases
  case gt_sub_t_s: gt_sub
  have u_gt_s_if_chk_u_t: ?thesis if chk_u_t: chksubs (op >_t) u t
    using gt_sub_t_s(1)
  proof (cases t)
    case t: (App t1 t2)
    show ?thesis
      using ih[of u t1 s] ih[of u t2 s] gt_sub_t_s(2) chk_u_t unfolding t by auto
  qed auto

show ?thesis
  using u_gt_t by cases (auto intro: u_gt_s_if_ui u_gt_s_if_chk_u_t)
next
case gt_diff_t_s: gt_diff
show ?thesis
  using u_gt_t
proof cases
  case gt_diff_u_t: gt_diff
  have head  $u >_{hd}$  head s
    using gt_diff_u_t(1) gt_diff_t_s(1) by (auto intro: gt_hd_trans)

```

```

thus ?thesis
  by (rule gt_diff[OF _ chkvar chk_u_s_if[OF gt_diff_t_s(3)]])
next
  case gt_same_u_t: gt_same
  have head u >hd head s
    using gt_diff_t_s(1) gt_same_u_t(1) by auto
  thus ?thesis
    by (rule gt_diff[OF _ chkvar chk_u_s_if[OF gt_diff_t_s(3)]])
  qed (auto intro: u_gt_s_if_ui)
next
  case gt_same_t_s: gt_same
  show ?thesis
    using u_gt_t
  proof cases
    case gt_diff_u_t: gt_diff
    have head u >hd head s
      using gt_diff_u_t(1) gt_same_t_s(1) by simp
    thus ?thesis
      by (rule gt_diff[OF _ chkvar chk_u_s_if[OF gt_same_t_s(2)]])
  next
  case gt_same_u_t: gt_same
  have hd_u_s: head u = head s
    using gt_same_u_t(1) gt_same_t_s(1) by simp

  let ?S = set (args u) ∪ set (args t) ∪ set (args s)

  have gt_trans_args: ∀ua ∈ ?S. ∀ta ∈ ?S. ∀sa ∈ ?S. ua >t ta → ta >t sa → ua >t sa
  proof clarify
    fix sa ta ua
    assume
      ua_in: ua ∈ ?S and ta_in: ta ∈ ?S and sa_in: sa ∈ ?S and
      ua_gt_ta: ua >t ta and ta_gt_sa: ta >t sa
    show ua >t sa
      by (auto intro!: ih[OF Max_lt_imp_lt_mset ua_gt_ta ta_gt_sa])
      (meson ua_in ta_in sa_in Un_iff max.strict_coboundedI1 max.strict_coboundedI2
        size_in_args)+
  qed

  have ∀f ∈ ground_heads (head u). extf f (op >t) (args u) (args s)
  proof (clarify, rule extf_trans[OF _ _ _ gt_trans_args])
    fix f
    assume f_in_grounds: f ∈ ground_heads (head u)
    show extf f (op >t) (args u) (args t)
      using f_in_grounds gt_same_u_t(3) by blast
    show extf f (op >t) (args t) (args s)
      using f_in_grounds gt_same_t_s(3) unfolding gt_same_u_t(1) by blast
  qed auto
  thus ?thesis
    by (rule gt_same[OF hd_u_s chk_u_s_if[OF gt_same_t_s(2)]])
  qed (auto intro: u_gt_s_if_ui)
qed

```

## 7.4 Irreflexivity

```

theorem gt_irrefl: ¬ s >t s
proof (standard, induct s rule: measure_induct_rule[of size])
  case (less s)
  note ih = this(1) and s_gt_s = this(2)

  show False
    using s_gt_s
  proof cases
    case _: gt_sub

```

```

note is_app = this(1) and si_ge_s = this(2)
have s_gt_fun_s: s >_t fun s and s_gt_arg_s: s >_t arg s
  using is_app by (simp_all add: gt_sub)

have fun_s >_t s ∨ arg_s >_t s
  using si_ge_s is_app s_gt_arg_s s_gt_fun_s by auto
moreover
{
  assume fun_s_gt_s: fun_s >_t s
  have fun_s >_t fun_s
    by (rule gt_trans[OF fun_s_gt_s s_gt_fun_s])
  hence False
    using ih[of fun_s] is_app size_fun_lt by blast
}
moreover
{
  assume arg_s_gt_s: arg_s >_t s
  have arg_s >_t arg_s
    by (rule gt_trans[OF arg_s_gt_s s_gt_arg_s])
  hence False
    using ih[of arg_s] is_app size_arg_lt by blast
}
ultimately show False
  by sat
next
case gt_diff
thus False
  by (cases head s) (auto simp: gt_hd_irrefl)
next
case gt_same
note in_grounds = this(3)

obtain si where si_in_args: si ∈ set (args s) and si_gt_si: si >_t si
  using in_grounds
  by (metis (full_types) all_not_in_conv extf_irrefl_from_trans ground_heads_nonempty gt_trans)
have size_si < size s
  by (rule size_in_args[OF si_in_args])
thus False
  by (rule ih[OF _ si_gt_si])
qed
qed

```

```

lemma gt_antisym: t >_t s ⟹ ¬ s >_t t
  using gt_irrefl gt_trans by blast

```

## 7.5 Subterm Property

```

lemma
  gt_sub_fun: App s t >_t s and
  gt_sub_arg: App s t >_t t
  by (auto intro: gt_sub)

```

```

theorem gt_proper_sub: proper_sub s t ⟹ t >_t s
  by (induct t) (auto intro: gt_sub_fun gt_sub_arg gt_trans)

```

## 7.6 Compatibility with Functions

```

lemma gt_compat_fun:
  assumes t'_gt_t: t' >_t t
  shows App s t' >_t App s t

```

proof –

```

  have t'_ne_t: t' ≠ t
    using gt_antisym t'_gt_t by blast
  have extf_args_single: ∀ f ∈ ground_heads (head s). extf (op >_t) (args s @ [t']) (args s @ [t])

```

```

  by (simp add: extf_compat_list t'_gt_t t'_ne_t)
show ?thesis
  by (rule gt_same) (auto simp: gt_sub gt_sub_fun t'_gt_t intro!: extf_args_single)
qed

```

```

theorem gt_compat_fun_strong:
  assumes t'_gt_t: t' >_t t
  shows apps s (t' # us) >_t apps s (t # us)
proof (induct us rule: rev_induct)
  case Nil
  show ?case
    using t'_gt_t by (auto intro!: gt_compat_fun)
next
case (snoc u us)
note ih = snoc

let ?v' = apps s (t' # us @ [u])
let ?v = apps s (t # us @ [u])

show ?case
proof (rule gt_same)
  show chksubs (op >_t) ?v' ?v
    using ih by (auto intro: gt_sub gt_sub_arg)
next
show  $\forall f \in \text{ground\_heads} (\text{head } ?v'). \text{extf } f (\text{op } >_t) (\text{args } ?v') (\text{args } ?v)$ 
  by (metis args_apps extf_compat_list gt_irrefl t'_gt_t)
qed simp
qed

```

## 7.7 Compatibility with Arguments

```

theorem gt_diff_same_compat_arg:
  assumes
    extf_compat_snoc:  $\bigwedge f. \text{ext\_compat\_snoc } (\text{extf } f)$  and
    diff_same:  $\text{gt\_diff } s' s \vee \text{gt\_same } s' s$ 
  shows  $\text{App } s' t >_t \text{App } s t$ 
proof -
  {
    assume s' >_t s
    hence  $\text{App } s' t >_t s$ 
      using gt_sub_fun gt_trans by blast
    moreover have  $\text{App } s' t >_t t$ 
      by (simp add: gt_sub_arg)
    ultimately have  $\text{chksubs } (\text{op } >_t) (\text{App } s' t) (\text{App } s t)$ 
      by auto
  }
note chk_s't_st = this

show ?thesis
  using diff_same
proof
  assume gt_diff s' s
  hence
    s'_gt_s:  $s' >_t s$  and
    hd_s'_gt_s:  $\text{head } s' >_{hd} \text{head } s$  and
    chkvar_s'_s:  $\text{chkvar } s' s$  and
    chk_s'_s:  $\text{chksubs } (\text{op } >_t) s' s$ 
    using gt_diff.cases by (auto simp: gt_iff_sub_diff_same)

  have  $\text{chkvar } s't\_st: \text{chkvar } (\text{App } s' t) (\text{App } s t)$ 
    using  $\text{chkvar } s'_s$  by auto
  show ?thesis
    by (rule gt_diff[OF chkvar_s't_st chk_s't_st[OF s'_gt_s]])
      (simp add: hd_s'_gt_s[simplified])

```

```

next
  assume  $gt\_same\ s'\ s$ 
  hence
     $s'\_gt\ s: s' >_t s$  and
     $hd\_s'\_eq\ s: head\ s' = head\ s$  and
     $chk\_s'\_s: chksubs\ (op\ >_t)\ s'\ s$  and
     $gts\_args: \forall f \in ground\_heads\ (head\ s').\ extf\ f\ (op\ >_t)\ (args\ s')\ (args\ s)$ 
    using  $gt\_same.cases$  by (auto simp:  $gt\_iff\_sub\_diff\_same, metis$ )

  have  $gts\_args\ t:$ 
     $\forall f \in ground\_heads\ (head\ (App\ s'\ t)).\ extf\ f\ op\ >_t\ (args\ (App\ s'\ t))\ (args\ (App\ s\ t))$ 
    using  $gts\_args\ ext\_compat\_snoc.compat\_append\_right[OF\ extf\_compat\_snoc]$  by simp

  show ?thesis
    by (rule  $gt\_same[OF\ \_chk\_s't\_st[OF\ s'\_gt\_s]\ gts\_args\_t]$ ) (simp add:  $hd\_s'\_eq\_s$ )
  qed
qed

```

## 7.8 Stability under Substitution

```

lemma  $gt\_imp\_chksubs\_gt:$ 
  assumes  $t\_gt\ s: t >_t s$ 
  shows  $chksubs\ (op\ >_t)\ t\ s$ 
proof -
  have  $is\_App\ s \implies t >_t fun\ s \wedge t >_t arg\ s$ 
    using  $t\_gt\ s$  by (meson  $gt\_sub\ gt\_trans$ )
  thus ?thesis
    by (simp add:  $tm.case\_eq\_if$ )
qed

theorem  $gt\_subst:$ 
  assumes  $wary\_q: wary\_subst\ q$ 
  shows  $t >_t s \implies subst\ q\ t >_t subst\ q\ s$ 
proof (simp only:  $atomize\_imp,$ 
  rule  $measure\_induct\_rule[of\ \lambda(t, s). \{\#size\ t, size\ s\#}$ 
   $\lambda(t, s). t >_t s \longrightarrow subst\ q\ t >_t subst\ q\ s\ (t, s),$ 
   $simplified\ prod.case],$ 
  simp only:  $split\_paired\_all\ prod.case\ atomize\_imp[symmetric]$ )
  fix  $t\ s$ 
  assume
     $ih: \bigwedge ta\ sa.\ \{\#size\ ta, size\ sa\# \} < \{\#size\ t, size\ s\# \} \implies ta >_t sa \implies$ 
     $subst\ q\ ta >_t subst\ q\ sa$  and
     $t\_gt\ s: t >_t s$ 

  {
  assume  $chk\_t\_s: chksubs\ (op\ >_t)\ t\ s$ 
  have  $chksubs\ (op\ >_t)\ (subst\ q\ t)\ (subst\ q\ s)$ 
  proof (cases  $s$ )
  case  $s: (Hd\ \zeta)$ 
  show ?thesis
  proof (cases  $\zeta$ )
  case  $\zeta: (Var\ x)$ 
  have  $psub\_x\_t: proper\_sub\ (Hd\ (Var\ x))\ t$ 
    using  $\zeta\ s\ t\_gt\ s\ gt\_imp\_vars\ gt\_irrefl\ in\_vars\_imp\_sub$  by fastforce
  show ?thesis
    unfolding  $\zeta\ s$ 
    by (rule  $gt\_imp\_chksubs\_gt[OF\ gt\_proper\_sub[OF\ proper\_sub\_subst]]$ ) (rule  $psub\_x\_t$ )
  qed (auto simp:  $s$ )
  next
  case  $s: (App\ s1\ s2)$ 
  have  $t >_t s1$  and  $t >_t s2$ 
    using  $s\ chk\_t\_s$  by auto
  thus ?thesis
    using  $s$  by (auto intro!:  $ih[of\ t\ s1]\ ih[of\ t\ s2]$ )
  }

```

```

qed
}
note chk_ϱt_ϱs_if = this

show subst ϱ t >t subst ϱ s
  using t_gt_s
proof cases
  case gt_sub_t_s: gt_sub
  obtain t1 t2 where t: t = App t1 t2
    using gt_sub_t_s(1) by (metis tm.collapse(2))
  show ?thesis
    using gt_sub ih[of t1 s] ih[of t2 s] gt_sub_t_s(2) t by auto
next
  case gt_diff_t_s: gt_diff
  have head (subst ϱ t) >hd head (subst ϱ s)
    by (meson wary_subst_ground_heads gt_diff_t_s(1) gt_hd_def subsetCE wary_ϱ)
  moreover have chkvar (subst ϱ t) (subst ϱ s)
    unfolding chkvar_def using vars_subst_subseteq[OF gt_imp_vars[OF t_gt_s]] vars_head_subseteq
    by force
  ultimately show ?thesis
    by (rule gt_diff[OF _ _ chk_ϱt_ϱs_if[OF gt_diff_t_s(3)]])
next
  case gt_same_t_s: gt_same

  have hd_ϱt_eq_ϱs: head (subst ϱ t) = head (subst ϱ s)
    using gt_same_t_s(1) by simp

  {
    fix f
    assume f_in_grounds: f ∈ ground_heads (head (subst ϱ t))

    let ?S = set (args t) ∪ set (args s)

    have extf_args_s_t: extf f (op >t) (args t) (args s)
      using gt_same_t_s(3) f_in_grounds wary_ϱ wary_subst_ground_heads by blast
    have extf f (op >t) (map (subst ϱ) (args t)) (map (subst ϱ) (args s))
    proof (rule extf_map[of ?S, OF _ _ _ _ _ extf_args_s_t])
      have sz_a: ∀ ta ∈ ?S. ∀ sa ∈ ?S. {#size ta, size sa#} < {#size t, size s#}
        by (fastforce intro: Max_lt_imp_lt_mset dest: size_in_args)
      show ∀ ta ∈ ?S. ∀ sa ∈ ?S. ta >t sa ⟶ subst ϱ ta >t subst ϱ sa
        using ih sz_a size_in_args by fastforce
    qed (auto intro!: gt_irrefl elim!: gt_trans)
    hence extf f (op >t) (args (subst ϱ t)) (args (subst ϱ s))
      by (auto simp: gt_same_t_s(1) intro: extf_compat_append_left)
  }
  hence ∀ f ∈ ground_heads (head (subst ϱ t)).
    extf f (op >t) (args (subst ϱ t)) (args (subst ϱ s))
    by blast
  thus ?thesis
    by (rule gt_same[OF hd_ϱt_eq_ϱs chk_ϱt_ϱs_if[OF gt_same_t_s(2)]])
qed
qed

```

## 7.9 Totality on Ground Terms

**theorem** *gt\_total\_ground*:

**assumes** *extf\_total*:  $\bigwedge f. \text{ext\_total } (extf\ f)$

**shows**  $\text{ground } t \implies \text{ground } s \implies t ><sub>t</sub> s \vee s ><sub>t</sub> t \vee t = s$

**proof** (*simp only*: *atomize\_imp*,

*rule measure\_induct\_rule*[of  $\lambda(t, s). \text{size } t + \text{size } s$

$\lambda(t, s). \text{ground } t \implies \text{ground } s \implies t ><sub>t</sub> s \vee s ><sub>t</sub> t \vee t = s$  (*t, s*), *simplified prod.case*],

*simp only*: *split\_paired\_all prod.case atomize\_imp*[*symmetric*])

**fix** *t s* :: (*'s, 'v*) *tm*

**assume**

```

ih:  $\bigwedge ta\ sa. \text{size } ta + \text{size } sa < \text{size } t + \text{size } s \implies \text{ground } ta \implies \text{ground } sa \implies$ 
 $ta >_t sa \vee sa >_t ta \vee ta = sa$  and
 $gr\_t: \text{ground } t$  and  $gr\_s: \text{ground } s$ 

let  $?case = t >_t s \vee s >_t t \vee t = s$ 

have  $chksubs\ (op\ >_t)\ t\ s \vee s >_t\ t$ 
  unfolding  $chksubs\_def\ tm.case\_eq\_if$  using  $ih[of\ t\ fun\ s]$   $ih[of\ t\ arg\ s]$ 
  by  $(metis\ gt\_sub\ add\_less\_cancel\_left\ gr\_s\ gr\_t\ ground\_arg\ ground\_fun\ size\_arg\_lt\ size\_fun\_lt)$ 
moreover have  $chksubs\ (op\ >_t)\ s\ t \vee t >_t\ s$ 
  unfolding  $chksubs\_def\ tm.case\_eq\_if$  using  $ih[of\ fun\ t\ s]$   $ih[of\ arg\ t\ s]$ 
  by  $(metis\ gt\_sub\ add\_less\_cancel\_right\ gr\_s\ gr\_t\ ground\_arg\ ground\_fun\ size\_arg\_lt\ size\_fun\_lt)$ 
moreover
{
  assume
     $chksubs\_t\_s: chksubs\ (op\ >_t)\ t\ s$  and
     $chksubs\_s\_t: chksubs\ (op\ >_t)\ s\ t$ 

  obtain  $g$  where  $g: \text{head } t = \text{Sym } g$ 
    using  $gr\_t$  by  $(metis\ ground\_head\ hd.collapse(2))$ 
  obtain  $f$  where  $f: \text{head } s = \text{Sym } f$ 
    using  $gr\_s$  by  $(metis\ ground\_head\ hd.collapse(2))$ 

  have  $chkvar\_t\_s: chkvar\ t\ s$  and  $chkvar\_s\_t: chkvar\ s\ t$ 
    using  $g\ f$  by  $simp\_all$ 

  {
    assume  $g\_gt\_f: g >_s f$ 
    have  $t >_t s$ 
      by  $(rule\ gt\_diff[OF\_chkvar\_t\_s\ chksubs\_t\_s])\ (simp\ add: g\ f\ gt\_sym\_imp\_hd[OF\ g\_gt\_f])$ 
  }
  moreover
  {
    assume  $f\_gt\_g: f >_s g$ 
    have  $s >_t t$ 
      by  $(rule\ gt\_diff[OF\_chkvar\_s\_t\ chksubs\_s\_t])\ (simp\ add: g\ f\ gt\_sym\_imp\_hd[OF\ f\_gt\_g])$ 
  }
  moreover
  {
    assume  $g\_eq\_f: g = f$ 
    hence  $hd\_t: \text{head } t = \text{head } s$ 
      using  $g\ f$  by  $auto$ 

    let  $?ts = args\ t$ 
    let  $?ss = args\ s$ 

    have  $gr\_ts: \forall ta \in set\ ?ts. \text{ground } ta$ 
      using  $ground\_args[OF\_gr\_t]$  by  $blast$ 
    have  $gr\_ss: \forall sa \in set\ ?ss. \text{ground } sa$ 
      using  $ground\_args[OF\_gr\_s]$  by  $blast$ 

    {
      assume  $ts\_eq\_ss: ?ts = ?ss$ 
      have  $t = s$ 
        using  $hd\_t\ ts\_eq\_ss$  by  $(rule\ tm\_expand\_apps)$ 
    }
  }
  moreover
  {
    assume  $ts\_gt\_ss: extf\ g\ (op\ >_t)\ ?ts\ ?ss$ 
    have  $t >_t s$ 
      by  $(rule\ gt\_same[OF\ hd\_t\ chksubs\_t\_s])\ (auto\ simp: g\ ts\_gt\_ss)$ 
  }
  moreover

```



```

{
  assume  $ss\_gt\_ts: extf\ g\ (op\ >_t)\ ?ss\ ?ts$ 
  have  $s\ >_t\ t$ 
    by (rule  $gt\_same[OF\ hd\_t[symmetric]\ chksubs\_s\_t]$ ) (auto simp:  $f[folded\ g\_eq\_f]\ ss\_gt\_ts$ )
}
ultimately have  $?case$ 
  using  $ih\ gr\_ss\ gr\_ts$ 
  ext_total.total[OF  $extf\_total, rule\_format, of\ set\ ?ts\ set\ ?ss\ op\ >_t\ ?ts\ ?ss\ g]$ 
  by (metis  $add\_strict\_mono\ in\_listsI\ size\_in\_args$ )
}
ultimately have  $?case$ 
  using  $gt\_sym\_total$  by blast
}
ultimately show  $?case$ 
  by fast
qed

```

## 7.10 Well-foundedness

**abbreviation**  $gtg :: ('s, 'v)\ tm \Rightarrow ('s, 'v)\ tm \Rightarrow bool$  (**infix**  $>_{tg}$  50) **where**  
 $op\ >_{tg} \equiv \lambda t\ s.\ ground\ t \wedge t\ >_t\ s$

**theorem**  $gt\_wf$ :

**assumes**  $extf\_wf: \bigwedge f.\ ext\_wf\ (extf\ f)$   
**shows**  $wfP\ (\lambda s\ t.\ t\ >_t\ s)$

**proof** –

**have**  $ground\_wfP: wfP\ (\lambda s\ t.\ t\ >_{tg}\ s)$   
**unfolding**  $wfP\_iff\_no\_inf\_chain$

**proof**

**assume**  $\exists f.\ inf\_chain\ (op\ >_{tg})\ f$   
**then obtain**  $t$  **where**  $t\_bad: bad\ (op\ >_{tg})\ t$   
**unfolding**  $inf\_chain\_def\ bad\_def$  **by**  $blast$

**let**  $?ff = worst\_chain\ (op\ >_{tg})\ (\lambda t\ s.\ size\ t\ >\ size\ s)$

**let**  $?U\_of = \lambda i.\ if\ is\_App\ (?ff\ i)\ then\ \{fun\ (?ff\ i)\} \cup set\ (args\ (?ff\ i))\ else\ \{\}$

**note**  $wf\_sz = wf\_app[OF\ wellorder\_class.wf, of\ size, simplified]$

**define**  $U$  **where**  $U = (\bigcup i.\ ?U\_of\ i)$

**have**  $gr: \bigwedge i.\ ground\ (?ff\ i)$   
**using**  $worst\_chain\_bad[OF\ wf\_sz\ t\_bad, unfolded\ inf\_chain\_def]$  **by**  $fast$

**have**  $gr\_fun: \bigwedge i.\ ground\ (fun\ (?ff\ i))$   
**by** (rule  $ground\_fun[OF\ gr]$ )

**have**  $gr\_args: \bigwedge i\ s.\ s \in set\ (args\ (?ff\ i)) \implies ground\ s$   
**by** (rule  $ground\_args[OF\ _\ gr]$ )

**have**  $gr\_u: \bigwedge u.\ u \in U \implies ground\ u$   
**unfolding**  $U\_def$  **by** (auto dest:  $gr\_args$ ) (metis (lifting)  $empty\_iff\ gr\_fun$ )

**have**  $\neg\ bad\ (op\ >_{tg})\ u$  **if**  $u\_in: u \in ?U\_of\ i$  **for**  $u\ i$

**proof**

**let**  $?ti = ?ff\ i$

**assume**  $u\_bad: bad\ (op\ >_{tg})\ u$

**have**  $sz\_u: size\ u < size\ ?ti$

**proof** (cases  $?ff\ i$ )

**case**  $Hd$

**thus**  $?thesis$

**using**  $u\_in\ size\_in\_args$  **by**  $fastforce$

**next**

**case**  $App$

**thus**  $?thesis$

**using**  $u\_in\ size\_in\_args\ insert\_iff\ size\_fun\_lt$  **by**  $fastforce$

**qed**

```

show False
proof (cases i)
  case 0
  thus False
  using sz_u min_worst_chain_0[OF wf_sz u_bad] by simp
next
case Suc
hence ?ff (i - 1) >t ?ff i
  using worst_chain_pred[OF wf_sz t_bad] by simp
moreover have ?ff i >t u
proof -
  have u_in: u ∈ ?U_of i
  using u_in by blast
  have ffi_ne_u: ?ff i ≠ u
  using sz_u by fastforce
  hence is_App (?ff i) ⇒ ¬ sub u (?ff i) ⇒ ?ff i >t u
  using u_in gt_sub sub_args by auto
  thus ?ff i >t u
  using ffi_ne_u u_in gt_proper_sub sub_args by fastforce
qed
ultimately have ?ff (i - 1) >t u
  by (rule gt_trans)
thus False
  using Suc sz_u min_worst_chain_Suc[OF wf_sz u_bad] gr by fastforce
qed
hence u_good: ∧u. u ∈ U ⇒ ¬ bad (op >tg) u
  unfolding U_def by blast

have bad_diff_same: inf_chain (λt s. ground t ∧ (gt_diff t s ∨ gt_same t s)) ?ff
  unfolding inf_chain_def
proof (intro allI conjI)
  fix i

  show ground (?ff i)
  by (rule gr)

  have gt: ?ff i >t ?ff (Suc i)
  using worst_chain_pred[OF wf_sz t_bad] by blast

  have ¬ gt_sub (?ff i) (?ff (Suc i))
  proof
    assume a: gt_sub (?ff i) (?ff (Suc i))
    hence fi_app: is_App (?ff i) and
      fun_or_arg_fi_ge: fun (?ff i) ≥t ?ff (Suc i) ∨ arg (?ff i) ≥t ?ff (Suc i)
    unfolding gt_sub.simps by blast+
    have fun (?ff i) ∈ ?U_of i
    unfolding U_def using fi_app by auto
    moreover have arg (?ff i) ∈ ?U_of i
    unfolding U_def using fi_app arg_in_args by force
    ultimately obtain uij where uij_in: uij ∈ U and uij_cases: uij ≥t ?ff (Suc i)
    unfolding U_def using fun_or_arg_fi_ge by blast

  have ∧n. ?ff n >t ?ff (Suc n)
  by (rule worst_chain_pred[OF wf_sz t_bad, THEN conjunct2])
  hence uij_gt_i_plus_3: uij >t ?ff (Suc (Suc i))
  using gt_trans uij_cases by blast

  have inf_chain (op >tg) (λj. if j = 0 then uij else ?ff (Suc (i + j)))
  unfolding inf_chain_def
  by (auto intro!: gr gr_u[OF uij_in] uij_gt_i_plus_3 worst_chain_pred[OF wf_sz t_bad])
  hence bad (op >tg) uij

```

```

    unfolding bad_def by fastforce
  thus False
    using u_good[OF uij_in] by sat
qed
thus gt_diff (?ff i) (?ff (Suc i))  $\vee$  gt_same (?ff i) (?ff (Suc i))
  using gt_unfolding gt_iff_sub_diff_same by sat
qed

have wf {(s, t). ground s  $\wedge$  ground t  $\wedge$  sym (head t)  $>_s$  sym (head s)}
  using gt_sym_wf unfolding wfP_def wf_iff_no_infinite_down_chain by fast
moreover have {(s, t). ground t  $\wedge$  gt_diff t s}
 $\subseteq$  {(s, t). ground s  $\wedge$  ground t  $\wedge$  sym (head t)  $>_s$  sym (head s)}
proof (clarsimp, intro conjI)
  fix s t
  assume gr_t: ground t and gt_diff_t_s: gt_diff t s
  thus gr_s: ground s
    using gt_iff_sub_diff_same gt_imp_vars by fastforce

  show sym (head t)  $>_s$  sym (head s)
    using gt_diff_t_s ground_head[OF gr_s] ground_head[OF gr_t]
    by (cases; cases head s; cases head t) (auto simp: gt_hd_def)
qed
ultimately have wf_diff: wf {(s, t). ground t  $\wedge$  gt_diff t s}
  by (rule wf_subset)

have diff_O_same: {(s, t). ground t  $\wedge$  gt_diff t s}  $O$  {(s, t). ground t  $\wedge$  gt_same t s}
 $\subseteq$  {(s, t). ground t  $\wedge$  gt_diff t s}
  unfolding gt_diff.simps gt_same.simps
  by clarsimp (metis chksubs_def empty_subsetI gt_diff[unfolded chkvar_def] gt_imp_chksubs_gt
    gt_same_gt_trans)

have diff_same_as_union: {(s, t). ground t  $\wedge$  (gt_diff t s  $\vee$  gt_same t s)} =
  {(s, t). ground t  $\wedge$  gt_diff t s}  $\cup$  {(s, t). ground t  $\wedge$  gt_same t s}
  by auto

obtain k where bad_same: inf_chain ( $\lambda t s$ . ground t  $\wedge$  gt_same t s) ( $\lambda i$ . ?ff (i + k))
  using wf_infinite_down_chain_compatible[OF wf_diff _ diff_O_same, of ?ff] bad_diff_same
  unfolding inf_chain_def diff_same_as_union[symmetric] by auto
hence hd_sym:  $\bigwedge i$ . is_Sym (head (?ff (i + k)))
  unfolding inf_chain_def by (simp add: ground_head)

define f where f = sym (head (?ff k))

have hd_eq_f: head (?ff (i + k)) = Sym f for i
proof (induct i)
  case 0
  thus ?case
    by (auto simp: f_def hd.collapse(2)[OF hd_sym, of 0, simplified])
next
  case (Suc ia)
  thus ?case
    using bad_same unfolding inf_chain_def gt_same.simps by simp
qed

let ?gtu =  $\lambda t s$ . t  $\in U \wedge t >_t s$ 

have t  $\in$  set (args (?ff i))  $\implies$  t  $\in$  ?U_of i for t i
  unfolding U_def
  by (cases is_App (?ff i), simp_all,
    metis (lifting) neq_iff_size_in_args sub.cases sub_args tm.discI(2))
moreover have  $\bigwedge i$ . extf f (op  $>_t$ ) (args (?ff (i + k))) (args (?ff (Suc i + k)))
  using bad_same hd_eq_f unfolding inf_chain_def gt_same.simps by auto
ultimately have  $\bigwedge i$ . extf f ?gtu (args (?ff (i + k))) (args (?ff (Suc i + k)))

```

```

    using extf_mono_strong[of _ _ op >t λt s. t ∈ U ∧ t >t s] unfolding U_def by blast
  hence inf_chain (extf f ?gtu) (λi. args (?ff (i + k)))
    unfolding inf_chain_def by blast
  hence nwf_ext: ¬ wfP (λxs ys. extf f ?gtu ys xs)
    unfolding wfP_iff_no_inf_chain by fast

  have gt_u_le_gtg: ?gtu ≤ op >tg
    by (auto intro!: gr_u)

  have wfP (λs t. ?gtu t s)
    unfolding wfP_iff_no_inf_chain
  proof (intro notI, elim exE)
    fix f
    assume bad_f: inf_chain ?gtu f
    hence bad_f0: bad ?gtu (f 0)
      by (rule inf_chain_bad)

    have f 0 ∈ U
      using bad_f unfolding inf_chain_def by blast
    hence good_f0: ¬ bad ?gtu (f 0)
      using u_good bad_f inf_chain_bad inf_chain_subset[OF _ gt_u_le_gtg] by blast

    show False
      using bad_f0 good_f0 by sat
  qed
  hence wf_ext: wfP (λxs ys. extf f ?gtu ys xs)
    by (rule ext_wf.wf[OF extf_wf, rule_format])

  show False
    using nwf_ext wf_ext by blast
  qed

  let ?subst = subst grounding_ϱ

  have wfP (λs t. ?subst t >tg ?subst s)
    by (rule wfP_app[OF ground_wfP])
  hence wfP (λs t. ?subst t >t ?subst s)
    by (simp add: ground_grounding_ϱ)
  thus ?thesis
    by (auto intro: wfP_subset gt_subst[OF wary_grounding_ϱ])
  qed

end

end

```

## 8 The Optimized Graceful Recursive Path Order for Lambda-Free Higher-Order Terms

```

theory Lambda_Free_RPO_Optim
imports Lambda_Free_RPO_Std
begin

```

This theory defines the optimized variant of the graceful recursive path order (RPO) for  $\lambda$ -free higher-order terms.

### 8.1 Setup

```

locale rpo_optim = rpo_basis _ _ arity_sym arity_var
  for
    arity_sym :: 's ⇒ enat and
    arity_var :: 'v ⇒ enat +

```

assumes *extf\_ext\_snoc*: *ext\_snoc* (*extf*)  
begin

lemmas *extf\_snoc* = *ext\_snoc.snoc*[*OF extf\_ext\_snoc*]

## 8.2 Definition of the Order

**definition**

*chkargs* :: ((*'s, 'v*) *tm* ⇒ (*'s, 'v*) *tm* ⇒ *bool*) ⇒ (*'s, 'v*) *tm* ⇒ (*'s, 'v*) *tm* ⇒ *bool*

where

[*simp*]: *chkargs gt t s* ⇔ (∀ *s' ∈ set (args s)*. *gt t s'*)

lemma *chkargs\_mono*[*mono*]: *gt* ≤ *gt'* ⇒ *chkargs gt* ≤ *chkargs gt'*  
by *force*

**inductive** *gt* :: (*'s, 'v*) *tm* ⇒ (*'s, 'v*) *tm* ⇒ *bool* (**infix** ><sub>*t*</sub> 50) **where**

*gt\_arg*: *ti ∈ set (args t)* ⇒ *ti* ><sub>*t*</sub> *s* ∨ *ti* = *s* ⇒ *t* ><sub>*t*</sub> *s*

| *gt\_diff*: *head t* ><sub>*hd*</sub> *head s* ⇒ *chkvar t s* ⇒ *chkargs (op ><sub>*t*</sub>) t s* ⇒ *t* ><sub>*t*</sub> *s*

| *gt\_same*: *head t* = *head s* ⇒ *chkargs (op ><sub>*t*</sub>) t s* ⇒

(∀ *f ∈ ground\_heads (head t)*. *extf f (op ><sub>*t*</sub>) (args t) (args s)*) ⇒ *t* ><sub>*t*</sub> *s*

**abbreviation** *ge* :: (*'s, 'v*) *tm* ⇒ (*'s, 'v*) *tm* ⇒ *bool* (**infix** ≥<sub>*t*</sub> 50) **where**  
*t* ≥<sub>*t*</sub> *s* ≡ *t* ><sub>*t*</sub> *s* ∨ *t* = *s*

## 8.3 Transitivity

lemma *gt\_in\_args\_imp*: *ti ∈ set (args t)* ⇒ *ti* ><sub>*t*</sub> *s* ⇒ *t* ><sub>*t*</sub> *s*  
by (*cases t*) (*auto intro: gt\_arg*)

lemma *gt\_imp\_vars*: *t* ><sub>*t*</sub> *s* ⇒ *vars t* ⊇ *vars s*

**proof** (*simp only: atomize\_imp*,

*rule measure\_induct\_rule*[of λ(*t, s*). *size t* + *size s*

λ(*t, s*). *t* ><sub>*t*</sub> *s* ⇒ *vars t* ⊇ *vars s* (*t, s*), *simplified prod.case*],

*simp only: split\_paired\_all prod.case atomize\_imp*[*symmetric*])

**fix** *t s*

**assume**

*ih*: ∧*ta sa*. *size ta* + *size sa* < *size t* + *size s* ⇒ *ta* ><sub>*t*</sub> *sa* ⇒ *vars ta* ⊇ *vars sa* **and**

*t* ><sub>*t*</sub> *s*

**show** *vars t* ⊇ *vars s*

**using** *t* ><sub>*t*</sub> *s*

**proof** *cases*

**case** (*gt\_arg ti*)

**thus** ?*thesis*

**using** *ih*[of *ti s*]

**by** (*metis size\_in\_args vars\_args\_subseteq add\_mono\_thms\_linordered\_field*(1) *order\_trans*)

**next**

**case** *gt\_diff*

**show** ?*thesis*

**proof** (*cases s*)

**case** *Hd*

**thus** ?*thesis*

**using** *gt\_diff*(2) **by** (*auto elim: hd.set\_cases*(2))

**next**

**case** (*App s1 s2*)

**thus** ?*thesis*

**using** *gt\_diff ih*

**by** *simp* (*metis* (*no\_types*) *add.assoc gt.simps*[*unfolded chkargs\_def chkvar\_def*] *less\_add\_Suc1*)

**qed**

**next**

**case** *gt\_same*

**thus** ?*thesis*

**proof** (*cases s rule: tm\_exhaust\_apps*)

**case** *s*: (*apps* ⊆ *ss*)

**thus** ?*thesis*

```

    using gt_same unfolding chkargs_def s
    by (auto intro!: vars_head_subseteq)
      (metis ih[of t] insert_absorb insert_subset nat_add_left_cancel_less s size_in_args
        tm_collapse_apps tm_inject_apps)
  qed
qed
qed

lemma gt_trans: u >_t t  $\implies$  t >_t s  $\implies$  u >_t s
proof (simp only: atomize_imp,
  rule measure_induct_rule[of  $\lambda(u, t, s). \{\#size\ u, size\ t, size\ s\}$ ]
   $\lambda(u, t, s). u >_t t \implies t >_t s \implies u >_t s (u, t, s)$ ,
  simplified prod.case],
  simp only: split_paired_all prod.case atomize_imp[symmetric])
fix u t s
assume
  ih:  $\bigwedge ua\ ta\ sa. \{\#size\ ua, size\ ta, size\ sa\} < \{\#size\ u, size\ t, size\ s\} \implies$ 
     $ua >_t ta \implies ta >_t sa \implies ua >_t sa$  and
  u_gt_t:  $u >_t t$  and t_gt_s:  $t >_t s$ 

have chkvar: chkvar u s
  by clarsimp (meson u_gt_t t_gt_s gt_imp_vars hd.set_sel(2) vars_head_subseteq subsetCE)

have chk_u_s_if: chkargs (op >_t) u s if chk_t_s: chkargs (op >_t) t s
proof (clarsimp simp only: chkargs_def)
  fix s'
  assume s'  $\in$  set (args s)
  thus  $u >_t s'$ 
    using chk_t_s by (auto intro!: ih[of _ _ s', OF _ u_gt_t] size_in_args)
qed

have u_gt_s_if_ui:  $ui \geq_t t \implies u >_t s$  if ui in:  $ui \in$  set (args u) for ui
  using ih[of ui t s, simplified, OF size_in_args[OF ui in] _ t_gt_s]
  gt_in_args_imp[OF ui in, of s] t_gt_s by blast

show  $u >_t s$ 
  using t_gt_s
proof cases
  case gt_arg_t_s: (gt_arg ti)
  have u_gt_s_if_chk_u_t: ?thesis if chk_u_t: chkargs (op >_t) u t
    using ih[of u ti s] gt_arg_t_s chk_u_t size_in_args by force
  show ?thesis
    using u_gt_t by cases (auto intro: u_gt_s_if_ui u_gt_s_if_chk_u_t)
next
  case gt_diff_t_s: gt_diff
  show ?thesis
    using u_gt_t
  proof cases
    case gt_diff_u_t: gt_diff
    have head u >_hd head s
      using gt_diff_u_t(1) gt_diff_t_s(1) by (auto intro: gt_hd_trans)
    thus ?thesis
      by (rule gt_diff[OF _ chkvar chk_u_s_if[OF gt_diff_t_s(3)]])
  next
  case gt_same_u_t: gt_same
  have head u >_hd head s
    using gt_diff_t_s(1) gt_same_u_t(1) by auto
  thus ?thesis
    by (rule gt_diff[OF _ chkvar chk_u_s_if[OF gt_diff_t_s(3)]])
  qed (auto intro: u_gt_s_if_ui)
next
  case gt_same_t_s: gt_same
  show ?thesis

```

```

    using u_gt_t
  proof cases
    case gt_diff_u_t: gt_diff
    have head_u >_hd head_s
      using gt_diff_u_t(1) gt_same_t_s(1) by simp
    thus ?thesis
      by (rule gt_diff[OF _ chkvar chk_u_s_if[OF gt_same_t_s(2)]])
  next
    case gt_same_u_t: gt_same
    have hd_u_s: head u = head s
      using gt_same_u_t(1) gt_same_t_s(1) by simp

    let ?S = set (args u) ∪ set (args t) ∪ set (args s)

    have gt_trans_args: ∀ua ∈ ?S. ∀ta ∈ ?S. ∀sa ∈ ?S. ua >_t ta → ta >_t sa → ua >_t sa
  proof clarify
    fix sa ta ua
    assume
      ua_in: ua ∈ ?S and ta_in: ta ∈ ?S and sa_in: sa ∈ ?S and
      ua_gt_ta: ua >_t ta and ta_gt_sa: ta >_t sa
    show ua >_t sa
      by (auto intro!: ih[OF Max_lt_imp_lt_mset ua_gt_ta ta_gt_sa])
      (meson ua_in ta_in sa_in Un_iff max.strict_coboundedI1 max.strict_coboundedI2
        size_in_args)+
  qed

  have ∀f ∈ ground_heads (head u). extf f (op >_t) (args u) (args s)
  proof (clarify, rule extf_trans[OF _ _ _ gt_trans_args])
    fix f
    assume f_in_grounds: f ∈ ground_heads (head u)
    show extf f op >_t (args u) (args t)
      using f_in_grounds gt_same_u_t(3) by blast
    show extf f op >_t (args t) (args s)
      using f_in_grounds gt_same_t_s(3) unfolding gt_same_u_t(1) by blast
  qed auto
  thus ?thesis
    by (rule gt_same[OF hd_u_s chk_u_s_if[OF gt_same_t_s(2)]])
  qed (auto intro: u_gt_s_if_ui)
  qed
  qed

```

end

## 8.4 Conditional Equivalence with Unoptimized Version

context rpo

begin

context

assumes extf\_ext\_snoc:  $\bigwedge f. \text{ext\_snoc } (extf\ f)$

begin

lemma rpo\_optim: rpo\_optim ground\_heads\_var (op >\_s) extf arity\_sym arity\_var

unfolding rpo\_optim\_def rpo\_optim\_axioms\_def using rpo\_basis\_axioms extf\_ext\_snoc by auto

abbreviation

chkargs ::  $((s, v) \text{ tm} \Rightarrow (s, v) \text{ tm} \Rightarrow \text{bool}) \Rightarrow (s, v) \text{ tm} \Rightarrow (s, v) \text{ tm} \Rightarrow \text{bool}$

where

chkargs  $\equiv$  rpo\_optim.chkargs

abbreviation gt\_optim ::  $(s, v) \text{ tm} \Rightarrow (s, v) \text{ tm} \Rightarrow \text{bool}$  (infix >\_to 50) where

$op >_{t_o} \equiv rpo\_optim.gt \text{ ground\_heads\_var } (op >_s) \text{ extf}$

**abbreviation**  $ge\_optim :: ('s, 'v) tm \Rightarrow ('s, 'v) tm \Rightarrow bool$  (**infix**  $\geq_{t_o}$  50) **where**  
 $op \geq_{t_o} \equiv rpo\_optim.ge \text{ ground\_heads\_var } (op >_s) \text{ extf}$

**theorem**  $gt\_iff\_optim: t >_t s \longleftrightarrow t >_{t_o} s$

**proof** (*rule measure\_induct\_rule*[of  $\lambda(t, s). \text{size } t + \text{size } s$

$\lambda(t, s). t >_t s \longleftrightarrow t >_{t_o} s (t, s), \text{ simplified prod.case}$ ],

*simp only: split\_paired\_all prod.case*)

**fix**  $t s :: ('s, 'v) tm$

**assume**  $ih: \bigwedge ta sa. \text{size } ta + \text{size } sa < \text{size } t + \text{size } s \implies ta >_t sa \longleftrightarrow ta >_{t_o} sa$

**show**  $t >_t s \longleftrightarrow t >_{t_o} s$

**proof**

**assume**  $t\_gt\_s: t >_t s$

**have**  $chkargs\_if\_chksubs: chkargs (op >_{t_o}) t s$  **if**  $chksubs: chksubs (op >_t) t s$

**unfolding**  $rpo\_optim.chkargs\_def$ [*OF*  $rpo\_optim$ ]

**proof** (*cases*  $s$ , *simp\_all*, *intro conjI ballI*)

**fix**  $s1 s2$

**assume**  $s: s = \text{App } s1 s2$

**have**  $t\_gt\_s2: t >_t s2$

**using**  $chksubs s$  **by** *simp*

**show**  $t >_{t_o} s2$

**by** (*rule*  $ih$ [*THEN iffD1*, *OF*  $t\_gt\_s2$ ]) (*simp add: s*)

{

**fix**  $s1i$

**assume**  $s1i\_in: s1i \in \text{set } (args s1)$

**have**  $t >_t s1$

**using**  $chksubs s$  **by** *simp*

**moreover** **have**  $s1 >_t s1i$

**using**  $s1i\_in$  *gt\_proper\_sub size\_in\_args sub\_args* **by** *fastforce*

**ultimately** **have**  $t\_gt\_s1i: t >_t s1i$

**by** (*rule*  $gt\_trans$ )

**have**  $sz\_s1i: \text{size } s1i < \text{size } s$

**using**  $size\_in\_args$ [*OF*  $s1i\_in$ ]  $s$  **by** *simp*

**show**  $t >_{t_o} s1i$

**by** (*rule*  $ih$ [*THEN iffD1*, *OF*  $t\_gt\_s1i$ ]) (*simp add: sz\_s1i*)

}

**qed**

**show**  $t >_{t_o} s$

**using**  $t\_gt\_s$

**proof** *cases*

**case**  $gt\_sub$

**note**  $t\_app = \text{this}(1)$  **and**  $ti\_geo\_s = \text{this}(2)$

**obtain**  $t1 t2$  **where**  $t: t = \text{App } t1 t2$

**using**  $t\_app$  **by** (*metis*  $tm.collapse(2)$ )

**have**  $t\_gto\_t1: t >_{t_o} t1$

**unfolding**  $t$  **by** (*rule*  $rpo\_optim.gt\_sub\_fun$ [*OF*  $rpo\_optim$ ])

**have**  $t\_gto\_t2: t >_{t_o} t2$

**unfolding**  $t$  **by** (*rule*  $rpo\_optim.gt\_arg$ [*OF*  $rpo\_optim$ , of  $t2$ ]) *simp+*

{

**assume**  $t1\_gt\_s: t1 >_t s$

**have**  $t1 >_{t_o} s$



```

    by (rule ih[THEN iffD1, OF _ t1_gt_s]) (simp add: t)
  hence ?thesis
    by (rule rpo_optim.gt_trans[OF rpo_optim t_gto_t1])
}
moreover
{
  assume t2_gt_s: t2 >_t s
  have t2 >_{t_o} s
    by (rule ih[THEN iffD1, OF _ t2_gt_s]) (simp add: t)
  hence ?thesis
    by (rule rpo_optim.gt_trans[OF rpo_optim t_gto_t2])
}
ultimately show ?thesis
  using t ti_geo_s t_gto_t1 t_gto_t2 by auto
next
case gt_diff
note hd_t_gt_s = this(1) and chkvar = this(2) and chksubs = this(3)
show ?thesis
  by (rule rpo_optim.gt_diff[OF rpo_optim hd_t_gt_s chkvar chkargs_if_chksubs[OF chksubs]])
next
case gt_same
note hd_t_eq_s = this(1) and chksubs = this(2) and extf = this(3)

have extf_gto:  $\forall f \in \text{ground\_heads } (\text{head } t). \text{extf } f \text{ (op } >_{t_o}) (\text{args } t) (\text{args } s)$ 
proof (rule ballI, rule extf_mono_strong[of _ _ op >_t, rule_format])
  fix f
  assume f_in_ground:  $f \in \text{ground\_heads } (\text{head } t)$ 

  {
    fix ta sa
    assume ta_in:  $ta \in \text{set } (\text{args } t)$  and sa_in:  $sa \in \text{set } (\text{args } s)$  and ta_gt_sa:  $ta >_t sa$ 

    show ta >_{t_o} sa
      by (rule ih[THEN iffD1, OF _ ta_gt_sa])
        (simp add: ta_in sa_in add_less_mono size_in_args)
  }

  show extf f (op >_t) (args t) (args s)
    using f_in_ground extf by simp
qed

show ?thesis
  by (rule rpo_optim.gt_same[OF rpo_optim hd_t_eq_s chkargs_if_chksubs[OF chksubs] extf_gto])
qed
next
assume t_gto_s:  $t >_{t_o} s$ 

have chksubs_if_chkargs:  $\text{chksubs } (\text{op } >_t) t s \text{ if } \text{chkargs: } \text{chkargs } (\text{op } >_{t_o}) t s$ 
  unfolding chksubs_def
proof (cases s, simp_all, rule conjI)
  fix s1 s2
  assume s:  $s = \text{App } s1 s2$ 

  have s >_{t_o} s1
    unfolding s by (rule rpo_optim.gt_sub_fun[OF rpo_optim])
  hence t_gto_s1:  $t >_{t_o} s1$ 
  by (rule rpo_optim.gt_trans[OF rpo_optim t_gto_s])
  show t >_t s1
    by (rule ih[THEN iffD2, OF _ t_gto_s1]) (simp add: s)

  have t_gto_s2:  $t >_{t_o} s2$ 
    using chkargs unfolding rpo_optim.chkargs_def[OF rpo_optim] s by simp
  show t >_t s2

```

```

  by (rule ih[THEN iffD2, OF _ t_gto_s2]) (simp add: s)
qed

show  $t >_t s$ 
proof (cases rule: rpo_optim.gt.cases[OF rpo_optim t_gto_s,
  case_names gto_arg gto_diff gto_same])
  case (gto_arg ti)
  hence  $ti\_in: ti \in set (args t)$  and  $ti\_geo\_s: ti \geq_{t_o} s$ 
  by auto
  obtain  $\zeta ts$  where  $t: t = apps (Hd \zeta) ts$ 
  by (fact tm_exhaust_apps)

  {
  assume  $ti\_gto\_s: ti >_{t_o} s$ 
  hence  $ti\_gt\_s: ti >_t s$ 
  using ih[of ti s] size_in_args ti_in by auto
  moreover have  $t >_t ti$ 
  using sub_args[OF ti_in] gt_proper_sub size_in_args[OF ti_in] by blast
  ultimately have ?thesis
  using gt_trans by blast
  }
  moreover
  {
  assume  $ti = s$ 
  hence ?thesis
  using sub_args[OF ti_in] gt_proper_sub size_in_args[OF ti_in] by blast
  }
  ultimately show ?thesis
  using ti_geo_s by blast
next
  case gto_diff
  hence  $hd\_t\_gt\_s: head t >_{hd} head s$  and  $chkvar: chkvar t s$  and
   $chkargs: chkargs (op >_{t_o}) t s$ 
  by blast+

  have  $chksubs (op >_t) t s$ 
  by (rule chksubs_if_chkargs[OF chkargs])
  thus ?thesis
  by (rule gt_diff[OF hd_t_gt_s chkvar])
next
  case gto_same
  hence  $hd\_t\_eq\_s: head t = head s$  and  $chkargs: chkargs (op >_{t_o}) t s$  and
   $extf\_gto: \forall f \in ground\_heads (head t). extf f (op >_{t_o}) (args t) (args s)$ 
  by blast+

  have  $chksubs: chksubs (op >_t) t s$ 
  by (rule chksubs_if_chkargs[OF chkargs])

  have  $extf: \forall f \in ground\_heads (head t). extf f (op >_t) (args t) (args s)$ 
  proof (rule ballI, rule extf_mono_strong[of _ _ op >_{t_o}, rule_format])
  fix f
  assume  $f\_in\_ground: f \in ground\_heads (head t)$ 

  {
  fix  $ta sa$ 
  assume  $ta\_in: ta \in set (args t)$  and  $sa\_in: sa \in set (args s)$  and  $ta\_gto\_sa: ta >_{t_o} sa$ 

  show  $ta >_t sa$ 
  by (rule ih[THEN iffD2, OF _ ta_gto_sa])
  (simp add: ta_in sa_in add_less_mono size_in_args)
  }

  show  $extf f (op >_{t_o}) (args t) (args s)$ 

```

```

    using f_in_ground extf_gto by simp
qed

show ?thesis
  by (rule gt_same[OF hd_t_eq_s chksubs extf])
qed
qed
qed

end

end

end

```

## 9 Recursive Path Orders for Lambda-Free Higher-Order Terms

```

theory Lambda_Free_RPOs
imports Lambda_Free_RPO_App Lambda_Free_RPO_Optim
begin

locale simple_rpo_instances
begin

definition arity_sym :: nat  $\Rightarrow$  enat where
  arity_sym n =  $\infty$ 

definition arity_var :: nat  $\Rightarrow$  enat where
  arity_var n =  $\infty$ 

definition ground_head_var :: nat  $\Rightarrow$  nat set where
  ground_head_var x = UNIV

definition gt_sym :: nat  $\Rightarrow$  nat  $\Rightarrow$  bool where
  gt_sym g f  $\longleftrightarrow$  g > f

sublocale app: rpo_app gt_sym len_lexext
  by unfold_locales (auto simp: gt_sym_def intro: wf_less[folded wfP_def])

sublocale std: rpo ground_head_var gt_sym  $\lambda$ f. len_lexext arity_sym arity_var
  by unfold_locales (auto simp: arity_var_def arity_sym_def ground_head_var_def)

sublocale optim: rpo_optim ground_head_var gt_sym  $\lambda$ f. len_lexext arity_sym arity_var
  by unfold_locales

end

end

```