

# A Formalization of Weighted Path Orders and Recursive Path Orders\*

Christian Sternagel      René Thiemann      Akihisa Yamada

April 18, 2024

## Abstract

We define the weighted path order (WPO) and formalize several properties such as strong normalization, the subterm property, and closure properties under substitutions and contexts. Our definition of WPO extends the original definition by also permitting multiset comparisons of arguments instead of just lexicographic extensions. Therefore, our WPO not only subsumes lexicographic path orders (LPO), but also recursive path orders (RPO). We formally prove these subsumptions and therefore all of the mentioned properties of WPO are automatically transferable to LPO and RPO as well. Such a transformation is not required for Knuth–Bendix orders (KBO), since they have already been formalized. Nevertheless, we still provide a proof that WPO subsumes KBO and thereby underline the generality of WPO.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Preliminaries</b>	<b>3</b>
2.1	Status functions . . . . .	3
2.2	Precedence . . . . .	4
2.3	Local versions of relations . . . . .	5
2.4	Interface for extending an order pair on lists . . . . .	11
<b>3</b>	<b>Multiset extension of an order pair</b>	<b>11</b>
3.1	Pointwise multiset order . . . . .	13
3.2	Multiset extension for order pairs via the pointwise order and <i>mult</i> . . . . .	17
3.3	One-step versions of the multiset extensions . . . . .	18
3.4	Cancellation . . . . .	21

---

\*Supported by FWF (Austrian Science Fund) projects P27502 and Y757.

3.5	Implementation friendly versions of <i>mult2-s</i> and <i>mult2-ns</i> . . .	22
3.6	Local well-foundedness: restriction to downward closure of a set . . . . .	23
3.7	Trivial cases . . . . .	27
3.8	Executable version . . . . .	30
<b>4</b>	<b>Multiset extension of order pairs in the other direction</b>	<b>34</b>
4.1	List based characterization of <i>multpw</i> . . . . .	35
4.2	Definition of the multiset extension of $>$ -orders . . . . .	35
4.3	Basic properties . . . . .	36
4.4	Multisets as order on lists . . . . .	53
4.5	Special case: non-strict order is equality . . . . .	55
4.6	Executable version . . . . .	57
<b>5</b>	<b>The Weighted Path Order</b>	<b>60</b>
<b>6</b>	<b>The Recursive Path Order as an instance of WPO</b>	<b>100</b>
<b>7</b>	<b>The Lexicographic Path Order as an instance of WPO</b>	<b>103</b>
<b>8</b>	<b>The Knuth–Bendix Order as an instance of WPO</b>	<b>106</b>
8.1	Aligning least elements . . . . .	106
8.2	A restricted equality between KBO and WPO . . . . .	109
<b>9</b>	<b>Executability of the orders</b>	<b>118</b>

## 1 Introduction

Path orders are well-founded orders on terms that are useful for automated deduction, e.g., for termination proving of term rewrite systems or for completion-based theorem provers. Well-known path orders are the lexicographic path order (LPO) [3], the recursive path order (RPO) [2], and the Knuth–Bendix order (KBO) [4], and all of these orders are presented in a standard textbook on term rewriting [1, Chapter 5].

Whereas the mentioned path orders date back to the last century, the weighted path order (WPO) has only recently been presented [9, 10]. It has two nice properties. First, the search for suitable parameters is feasible and tools like NaTT and TTT2 implement it. Second, WPO is quite powerful and versatile: in fact, KBO and LPO are just instances of WPO. Moreover, with a slight extension of WPO (adding multiset-comparisons) also RPO is covered.

This AFP-entry provides a full formalization of WPO and also the connection to KBO, LPO, and RPO. Here, for the existing formal version of KBO [5, 6] it is just proven that WPO can simulate it by choosing suitable

parameters, whereas LPO and RPO are defined from scratch and many properties of LPO and RPO—such as strong normalization, closure under contexts and substitutions, transitivity, etc.—are derived from the corresponding WPO properties.

Note that most of the WPO formalization is described in [8]. The formal version deviates from the paper version only by the additional possibility to perform multiset-comparisons instead of lexicographic comparisons within WPO. The formal version of LPO and RPO extend their original definitions as well: the RPO definition is taken from [7], and LPO is defined as this extended RPO where always lexicographic comparisons are performed when comparing lists of terms. The formalization of multiset-comparisons (w.r.t. two orders) is described in more detail in [7].

## 2 Preliminaries

### 2.1 Status functions

A status function assigns to each n-ary symbol a list of indices between 0 and n-1. These functions are encapsulated into a separate type, so that recursion on the i-th subterm does not have to perform out-of-bounds checks (e.g., to ensure termination).

**theory** *Status*

**imports**

*First-Order-Terms.Term*

**begin**

**typedef** *'f status* = { ( $\sigma :: 'f \times \text{nat} \Rightarrow \text{nat list}$ ). ( $\forall f k. \text{set } (\sigma (f, k)) \subseteq \{0 ..< k\}$ )}

**morphisms** *status Abs-status*

**by** (*rule exI[of -  $\lambda \_ . []$ ] auto*)

**setup-lifting** *type-definition-status*

**lemma** *status: set (status  $\sigma (f, n)) \subseteq \{0 ..< n\}$*

**by** (*transfer*) *auto*

**lemma** *status-aux[termination-simp]:  $i \in \text{set } (\text{status } \sigma (f, \text{length } ss)) \implies ss ! i \in \text{set } ss$*

**using** *status[of  $\sigma f \text{length } ss$ ] unfolding set-conv-nth by force*

**lemma** *status-termination-simps[termination-simp]:*

**assumes** *i1:  $i < \text{length } (\text{status } \sigma (f, \text{length } xs))$*

**shows** *size (xs ! (status  $\sigma (f, \text{length } xs) ! i)) < \text{Suc } (\text{size-list size } xs)$  (is ?a < ?c)*

**proof** –

**from** *i1 have status  $\sigma (f, \text{length } xs) ! i \in \text{set } (\text{status } \sigma (f, \text{length } xs))$  by auto*

**from** *status-aux[OF this] have ?a  $\leq \text{size-list size } xs$  by (auto simp: termination-simp)*

**then show** *?thesis* **by** *auto*  
**qed**

**lemma** *status-ne*:

*status*  $\sigma (f, n) \neq [] \implies \exists i < n. i \in \text{set} (\text{status } \sigma (f, n))$

**using** *status* [*of*  $\sigma f n$ ]

**by** (*meson atLeastLessThan-iff set-empty subsetCE subsetI subset-empty*)

**lemma** *set-status-nth*:

*length xs = n*  $\implies i \in \text{set} (\text{status } \sigma (f, n)) \implies i < \text{length } xs \wedge xs ! i \in \text{set } xs$

**using** *status* [*of*  $\sigma f n$ ] **by** *force*

**lift-definition** *full-status* :: *'f status is*  $\lambda (f, n). [0 ..< n]$  **by** *auto*

**lemma** *full-status[simp]*: *status full-status*  $(f, n) = [0 ..< n]$

**by** *transfer auto*

An argument position *i* is simple wrt. some term relation, if the *i*-th subterm is in relation to the full term.

**definition** *simple-arg-pos* :: (*'f, 'v*) *term rel*  $\Rightarrow 'f \times \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**

*simple-arg-pos rel f i*  $\equiv \forall ts. i < \text{snd } f \longrightarrow \text{length } ts = \text{snd } f \longrightarrow (\text{Fun } (\text{fst } f) ts, ts ! i) \in \text{rel}$

**lemma** *simple-arg-posI*:  $\llbracket \bigwedge ts. \text{length } ts = n \implies i < n \implies (\text{Fun } f ts, ts ! i) \in \text{rel} \rrbracket \implies \text{simple-arg-pos rel } (f, n) i$

**unfolding** *simple-arg-pos-def* **by** *auto*

**end**

## 2.2 Precedence

A precedence consists of two compatible relations (strict and non-strict) on symbols such that the strict relation is strongly normalizing. In the formalization we model this via a function "prc" (precedence-compare) which returns two Booleans, indicating whether the one symbol is strictly or weakly bigger than the other symbol. Moreover, there also is a function "prl" (precedence-least) which gives quick access to whether a symbol is least in precedence, i.e., without comparing it to all other symbols explicitly.

**theory** *Precedence*

**imports**

*Abstract-Rewriting.Abstract-Rewriting*

**begin**

**locale** *irrefl-precedence* =

**fixes** *prc* :: *'f*  $\Rightarrow 'f \Rightarrow \text{bool} \times \text{bool}$

**and** *prl* :: *'f*  $\Rightarrow \text{bool}$

**assumes** *prc-refl*: *prc f f* = (*False*, *True*)

**and** *prc-stri-imp-nstri*: *fst (prc f g)*  $\implies \text{snd } (\text{prc } f g)$

```

and prl: prl g  $\implies$  snd (prc f g) = True
and prl3: prl f  $\implies$  snd (prc f g)  $\implies$  prl g
and prc-compat: prc f g = (s1, ns1)  $\implies$  prc g h = (s2, ns2)  $\implies$  prc f h = (s,
ns)  $\implies$ 
(ns1  $\wedge$  ns2  $\longrightarrow$  ns)  $\wedge$  (ns1  $\wedge$  s2  $\longrightarrow$  s)  $\wedge$  (s1  $\wedge$  ns2  $\longrightarrow$  s)
begin
lemma prl2:
  assumes g: prl g shows fst (prc g f) = False
proof (rule ccontr)
  assume  $\neg$  ?thesis
  then obtain b where gf: prc g f = (True, b) by (cases prc g f, auto)
  obtain b1 b2 where gg: prc g g = (b1, b2) by force
  obtain b' where fg: prc f g = (b', True) using prl[OF g, of f] by (cases prc f
g, auto)
  from prc-compat[OF gf fg gg] gg have gg: fst (prc g g) by auto
  with prc-refl[of g] show False by auto
qed

abbreviation pr  $\equiv$  (prc, prl)

end

```

```

locale precedence = irrefl-precedence +
  constrains prc :: 'f  $\Rightarrow$  'f  $\Rightarrow$  bool  $\times$  bool
  and prl :: 'f  $\Rightarrow$  bool
  assumes prc-SN: SN {(f, g). fst (prc f g)}

```

**end**

## 2.3 Local versions of relations

**theory** *Relations*

**imports**

*HOL-Library.Multiset*

*Abstract-Rewriting.Abstract-Rewriting*

**begin**

Common predicates on relations

**definition** *compatible-l* :: '*a rel*  $\Rightarrow$  '*a rel*  $\Rightarrow$  *bool* **where**  
*compatible-l R1 R2*  $\equiv$  *R1 O R2*  $\subseteq$  *R2*

**definition** *compatible-r* :: '*a rel*  $\Rightarrow$  '*a rel*  $\Rightarrow$  *bool* **where**  
*compatible-r R1 R2*  $\equiv$  *R2 O R1*  $\subseteq$  *R2*

Local reflexivity

**definition** *locally-refl* :: '*a rel*  $\Rightarrow$  '*a multiset*  $\Rightarrow$  *bool* **where**  
*locally-refl R A*  $\equiv$  ( $\forall$  *a. a*  $\in$   $\#$  *A*  $\longrightarrow$  (*a, a*)  $\in$  *R*)

**definition** *locally-irrefl* :: '*a rel*  $\Rightarrow$  '*a multiset*  $\Rightarrow$  *bool* **where**

*locally-irrefl*  $R A \equiv (\forall t. t \in\# A \longrightarrow (t,t) \notin R)$

Local symmetry

**definition** *locally-sym*  $:: 'a \text{ rel} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$  **where**  
*locally-sym*  $R A \equiv (\forall t u. t \in\# A \longrightarrow u \in\# A \longrightarrow$   
 $(t,u) \in R \longrightarrow (u,t) \in R)$

**definition** *locally-antisym*  $:: 'a \text{ rel} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$  **where**  
*locally-antisym*  $R A \equiv (\forall t u. t \in\# A \longrightarrow u \in\# A \longrightarrow$   
 $(t,u) \in R \longrightarrow (u,t) \in R \longrightarrow t = u)$

Local transitivity

**definition** *locally-trans*  $:: 'a \text{ rel} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$   
**where**

*locally-trans*  $R A B C \equiv (\forall t u v.$   
 $t \in\# A \longrightarrow u \in\# B \longrightarrow v \in\# C \longrightarrow$   
 $(t,u) \in R \longrightarrow (u,v) \in R \longrightarrow (t,v) \in R)$

Local inclusion

**definition** *locally-included*  $:: 'a \text{ rel} \Rightarrow 'a \text{ rel} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$   
**where**

*locally-included*  $R1 R2 A B \equiv (\forall t u. t \in\# A \longrightarrow u \in\# B \longrightarrow$   
 $(t,u) \in R1 \longrightarrow (t,u) \in R2)$

Local transitivity compatibility

**definition** *locally-compatible-l*  $:: 'a \text{ rel} \Rightarrow 'a \text{ rel} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a$   
*multiset*  $\Rightarrow \text{bool}$  **where**

*locally-compatible-l*  $R1 R2 A B C \equiv (\forall t u v. t \in\# A \longrightarrow u \in\# B \longrightarrow v \in\# C$   
 $\longrightarrow$   
 $(t,u) \in R1 \longrightarrow (u,v) \in R2 \longrightarrow (t,v) \in R2)$

**definition** *locally-compatible-r*  $:: 'a \text{ rel} \Rightarrow 'a \text{ rel} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a$   
*multiset*  $\Rightarrow \text{bool}$  **where**

*locally-compatible-r*  $R1 R2 A B C \equiv (\forall t u v. t \in\# A \longrightarrow u \in\# B \longrightarrow v \in\# C$   
 $\longrightarrow$   
 $(t,u) \in R2 \longrightarrow (u,v) \in R1 \longrightarrow (t,v) \in R2)$

included + compatible  $\longrightarrow$  transitive

**lemma** *in-cl-tr*:

**assumes**  $R1 \subseteq R2$

**and** *compatible-l*  $R2 R1$

**shows** *trans*  $R1$

**proof**–

{  
**fix**  $x y z$   
**assume**  $s\text{-}x\text{-}y: (x,y) \in R1$  **and**  $s\text{-}y\text{-}z: (y,z) \in R1$   
**from** *assms*  $s\text{-}x\text{-}y$  **have**  $(x,y) \in R2$  **by** *auto*  
**with**  $s\text{-}y\text{-}z$  *assms*(2)[*unfolded compatible-l-def*] **have**  $(x,z) \in R1$  **by** *blast*  
}

**then show** *?thesis unfolding trans-def by fast*  
**qed**

**lemma** *in-cr-tr*:  
**assumes**  $R1 \subseteq R2$   
**and** *compatible-r R2 R1*  
**shows** *trans R1*  
**proof-**  
**{**  
**fix**  $x\ y\ z$   
**assume**  $s-x-y: (x,y) \in R1$  **and**  $s-y-z: (y,z) \in R1$   
**with** *assms* **have**  $(y,z) \in R2$  **by** *auto*  
**with**  $s-x-y$  *assms*(2)[*unfolded compatible-r-def*] **have**  $(x,z) \in R1$  **by** *blast*  
**}**  
**then show** *?thesis unfolding trans-def by fast*  
**qed**

If a property holds globally, it also holds locally. Obviously.

**lemma** *r-lr*:  
**assumes** *refl R*  
**shows** *locally-refl R A*  
**using** *assms* **unfolding** *refl-on-def locally-refl-def* **by** *blast*

**lemma** *tr-ltr*:  
**assumes** *trans R*  
**shows** *locally-trans R A B C*  
**using** *assms* **unfolding** *trans-def* **and** *locally-trans-def* **by** *fast*

**lemma** *in-lin*:  
**assumes**  $R1 \subseteq R2$   
**shows** *locally-included R1 R2 A B*  
**using** *assms* **unfolding** *locally-included-def* **by** *auto*

**lemma** *cl-lcl*:  
**assumes** *compatible-l R1 R2*  
**shows** *locally-compatible-l R1 R2 A B C*  
**using** *assms* **unfolding** *compatible-l-def* **and** *locally-compatible-l-def* **by** *auto*

**lemma** *cr-lcr*:  
**assumes** *compatible-r R1 R2*  
**shows** *locally-compatible-r R1 R2 A B C*  
**using** *assms* **unfolding** *compatible-r-def* **and** *locally-compatible-r-def* **by** *auto*

If a predicate holds on a set then it holds on all the subsets:

**lemma** *lr-trans-l*:  
**assumes** *locally-refl R (A + B)*  
**shows** *locally-refl R A*  
**using** *assms* **unfolding** *locally-refl-def*  
**by** *auto*

**lemma** *li-trans-l*:  
**assumes** *locally-irrefl*  $R (A + B)$   
**shows** *locally-irrefl*  $R A$   
**using** *assms* **unfolding** *locally-irrefl-def*  
**by** *auto*

**lemma** *ls-trans-l*:  
**assumes** *locally-sym*  $R (A + B)$   
**shows** *locally-sym*  $R A$   
**using** *assms* **unfolding** *locally-sym-def*  
**by** *auto*

**lemma** *las-trans-l*:  
**assumes** *locally-antisym*  $R (A + B)$   
**shows** *locally-antisym*  $R A$   
**using** *assms* **unfolding** *locally-antisym-def*  
**by** *auto*

**lemma** *lt-trans-l*:  
**assumes** *locally-trans*  $R (A + B) (C + D) (E + F)$   
**shows** *locally-trans*  $R A C E$   
**using** *assms*[*unfolded locally-trans-def, rule-format*]  
**unfolding** *locally-trans-def* **by** *auto*

**lemma** *lin-trans-l*:  
**assumes** *locally-included*  $R1 R2 (A + B) (C + D)$   
**shows** *locally-included*  $R1 R2 A C$   
**using** *assms* **unfolding** *locally-included-def* **by** *auto*

**lemma** *lcl-trans-l*:  
**assumes** *locally-compatible-l*  $R1 R2 (A + B) (C + D) (E + F)$   
**shows** *locally-compatible-l*  $R1 R2 A C E$   
**using** *assms*[*unfolded locally-compatible-l-def, rule-format*]  
**unfolding** *locally-compatible-l-def* **by** *auto*

**lemma** *lcr-trans-l*:  
**assumes** *locally-compatible-r*  $R1 R2 (A + B) (C + D) (E + F)$   
**shows** *locally-compatible-r*  $R1 R2 A C E$   
**using** *assms*[*unfolded locally-compatible-r-def, rule-format*]  
**unfolding** *locally-compatible-r-def* **by** *auto*

**lemma** *lr-trans-r*:  
**assumes** *locally-refl*  $R (A + B)$   
**shows** *locally-refl*  $R B$   
**using** *assms* **unfolding** *locally-refl-def*  
**by** *auto*

**lemma** *li-trans-r*:

**assumes** *locally-irrefl*  $R (A + B)$   
**shows** *locally-irrefl*  $R B$   
**using** *assms* **unfolding** *locally-irrefl-def*  
**by** *auto*

**lemma** *ls-trans-r*:  
**assumes** *locally-sym*  $R (A + B)$   
**shows** *locally-sym*  $R B$   
**using** *assms* **unfolding** *locally-sym-def*  
**by** *auto*

**lemma** *las-trans-r*:  
**assumes** *locally-antisym*  $R (A + B)$   
**shows** *locally-antisym*  $R B$   
**using** *assms* **unfolding** *locally-antisym-def*  
**by** *auto*

**lemma** *lt-trans-r*:  
**assumes** *locally-trans*  $R (A + B) (C + D) (E + F)$   
**shows** *locally-trans*  $R B D F$   
**using** *assms*[*unfolded locally-trans-def, rule-format*]  
**unfolding** *locally-trans-def*  
**by** *auto*

**lemma** *lin-trans-r*:  
**assumes** *locally-included*  $R1 R2 (A + B) (C + D)$   
**shows** *locally-included*  $R1 R2 B D$   
**using** *assms* **unfolding** *locally-included-def* **by** *auto*

**lemma** *lcl-trans-r*:  
**assumes** *locally-compatible-l*  $R1 R2 (A + B) (C + D) (E + F)$   
**shows** *locally-compatible-l*  $R1 R2 B D F$   
**using** *assms*[*unfolded locally-compatible-l-def, rule-format*]  
**unfolding** *locally-compatible-l-def* **by** *auto*

**lemma** *lcr-trans-r*:  
**assumes** *locally-compatible-r*  $R1 R2 (A + B) (C + D) (E + F)$   
**shows** *locally-compatible-r*  $R1 R2 B D F$   
**using** *assms*[*unfolded locally-compatible-r-def, rule-format*]  
**unfolding** *locally-compatible-r-def* **by** *auto*

**lemma** *lr-minus*:  
**assumes** *locally-refl*  $R A$   
**shows** *locally-refl*  $R (A - B)$   
**using** *assms* **unfolding** *locally-refl-def* **by** (*meson in-diffD*)

**lemma** *li-minus*:  
**assumes** *locally-irrefl*  $R A$   
**shows** *locally-irrefl*  $R (A - B)$

**using** *assms* **unfolding** *locally-irrefl-def* **by** (*meson in-diffD*)

**lemma** *ls-minus*:

**assumes** *locally-sym*  $R$   $A$   
**shows** *locally-sym*  $R$   $(A - B)$   
**using** *assms* **unfolding** *locally-sym-def* **by** (*meson in-diffD*)

**lemma** *las-minus*:

**assumes** *locally-antisym*  $R$   $A$   
**shows** *locally-antisym*  $R$   $(A - B)$   
**using** *assms* **unfolding** *locally-antisym-def* **by** (*meson in-diffD*)

**lemma** *lt-minus*:

**assumes** *locally-trans*  $R$   $A$   $C$   $E$   
**shows** *locally-trans*  $R$   $(A - B)$   $(C - D)$   $(E - F)$   
**using** *assms*[*unfolded locally-trans-def*, *rule-format*]  
**unfolding** *locally-trans-def* **by** (*meson in-diffD*)

**lemma** *lin-minus*:

**assumes** *locally-included*  $R1$   $R2$   $A$   $C$   
**shows** *locally-included*  $R1$   $R2$   $(A - B)$   $(C - D)$   
**using** *assms* **unfolding** *locally-included-def* **by** (*meson in-diffD*)

**lemma** *lcl-minus*:

**assumes** *locally-compatible-l*  $R1$   $R2$   $A$   $C$   $E$   
**shows** *locally-compatible-l*  $R1$   $R2$   $(A - B)$   $(C - D)$   $(E - F)$   
**using** *assms*[*unfolded locally-compatible-l-def*, *rule-format*]  
**unfolding** *locally-compatible-l-def* **by** (*meson in-diffD*)

**lemma** *lcr-minus*:

**assumes** *locally-compatible-r*  $R1$   $R2$   $A$   $C$   $E$   
**shows** *locally-compatible-r*  $R1$   $R2$   $(A - B)$   $(C - D)$   $(E - F)$   
**using** *assms*[*unfolded locally-compatible-r-def*, *rule-format*]  
**unfolding** *locally-compatible-r-def* **by** (*meson in-diffD*)

Notations

**notation** *restrict* (**infixl**  $\uparrow$  80)

**lemma** *mem-restrictI*[*intro!*]: **assumes**  $x \in X$   $y \in X$   $(x,y) \in R$  **shows**  $(x,y) \in R$   
 $\uparrow$   $X$   
**using** *assms* **unfolding** *restrict-def* **by** *auto*

**lemma** *mem-restrictD*[*dest*]: **assumes**  $(x,y) \in R$   $\uparrow$   $X$  **shows**  $x \in X$   $y \in X$   $(x,y) \in R$   
 $\in R$   
**using** *assms* **unfolding** *restrict-def* **by** *auto*

end

## 2.4 Interface for extending an order pair on lists

**theory** *List-Order*

**imports**

*Knuth-Bendix-Order.Order-Pair*

**begin**

**type-synonym** *'a list-ext* = *'a rel*  $\Rightarrow$  *'a rel*  $\Rightarrow$  *'a list rel*

**locale** *list-order-extension* =

**fixes** *s-list* :: *'a list-ext*

**and** *ns-list* :: *'a list-ext*

**assumes** *extension*: *SN-order-pair S NS*  $\Longrightarrow$  *SN-order-pair (s-list S NS) (ns-list S NS)*

**and** *s-map*:  $\llbracket \bigwedge a b. (a,b) \in S \Longrightarrow (f a, f b) \in S; \bigwedge a b. (a,b) \in NS \Longrightarrow (f a, f b) \in NS \rrbracket \Longrightarrow (as, bs) \in s\text{-list } S NS \Longrightarrow (map f as, map f bs) \in s\text{-list } S NS$

**and** *ns-map*:  $\llbracket \bigwedge a b. (a,b) \in S \Longrightarrow (f a, f b) \in S; \bigwedge a b. (a,b) \in NS \Longrightarrow (f a, f b) \in NS \rrbracket \Longrightarrow (as, bs) \in ns\text{-list } S NS \Longrightarrow (map f as, map f bs) \in ns\text{-list } S NS$

**and** *all-ns-imp-ns*: *length as = length bs*  $\Longrightarrow \llbracket \bigwedge i. i < length bs \Longrightarrow (as ! i, bs ! i) \in NS \rrbracket \Longrightarrow (as, bs) \in ns\text{-list } S NS$

**type-synonym** *'a list-ext-impl* = (*'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *bool*  $\times$  *bool*)  $\Rightarrow$  *'a list*  $\Rightarrow$  *'a list*  $\Rightarrow$  *bool*  $\times$  *bool*

**locale** *list-order-extension-impl* = *list-order-extension s-list ns-list for*

*s-list ns-list* :: *'a list-ext* +

**fixes** *list-ext* :: *'a list-ext-impl*

**assumes** *list-ext-s*:  $\bigwedge s ns. s\text{-list } \{(a,b). s a b\} \{(a,b). ns a b\} = \{(as,bs). fst (list-ext (\lambda a b. (s a b, ns a b))) as bs\}$

**and** *list-ext-ns*:  $\bigwedge s ns. ns\text{-list } \{(a,b). s a b\} \{(a,b). ns a b\} = \{(as,bs). snd (list-ext (\lambda a b. (s a b, ns a b))) as bs\}$

**and** *s-ext-local-mono*:  $\bigwedge s ns s' ns' as bs. (set as \times set bs) \cap ns \subseteq ns' \Longrightarrow (set as \times set bs) \cap s \subseteq s' \Longrightarrow (as,bs) \in s\text{-list } ns s \Longrightarrow (as,bs) \in s\text{-list } ns' s'$

**and** *ns-ext-local-mono*:  $\bigwedge s ns s' ns' as bs. (set as \times set bs) \cap ns \subseteq ns' \Longrightarrow (set as \times set bs) \cap s \subseteq s' \Longrightarrow (as,bs) \in ns\text{-list } ns s \Longrightarrow (as,bs) \in ns\text{-list } ns' s'$

end

## 3 Multiset extension of an order pair

Given a well-founded order  $\prec$  and a compatible non-strict order  $\succsim$ , we define the corresponding multiset-extension of these orders.

**theory** *Multiset-Extension-Pair*

**imports**

*HOL-Library.Multiset*

*Regular-Sets.Regexp-Method*

*Abstract–Rewriting.Abstract-Rewriting  
Relations*

**begin**

**lemma** *mult-locally-cancel:*

**assumes** *trans s and locally-irrefl s (X + Z) and locally-irrefl s (Y + Z)*  
**shows**  $(X + Z, Y + Z) \in \text{mult } s \longleftrightarrow (X, Y) \in \text{mult } s$  (**is**  $?L \longleftrightarrow ?R$ )

**proof**

**assume**  $?L$  **thus**  $?R$  **using** *assms(2, 3)*

**proof** (*induct Z arbitrary: X Y*)

**case** (*add z Z*)

**obtain**  $X' Y' Z'$  **where**  $*$ :  $\text{add-mset } z X + Z = Z' + X'$   $\text{add-mset } z Y + Z = Z' + Y'$   $Y' \neq \{\#\}$

$\forall x \in \text{set-mset } X'. \exists y \in \text{set-mset } Y'. (x, y) \in s$

**using** *mult-implies-one-step[OF <trans s> add(2)]* **by** *auto*

**consider**  $Z2$  **where**  $Z' = \text{add-mset } z Z2 \mid X2 Y2$  **where**  $X' = \text{add-mset } z X2$   
 $Y' = \text{add-mset } z Y2$

**using**  $*(1,2)$  **by** (*metis add-mset-remove-trivial-If insert-iff set-mset-add-mset-insert union-iff*)

**thus**  $?case$

**proof** (*cases*)

**case** 1 **thus**  $?thesis$  **using**  $*$  *one-step-implies-mult[of Y' X' s Z2]*

**by** (*auto simp: add.commute[of - {#-#}] add.assoc intro: add(1)*)

(*metis add.hyps add.prem(2) add.prem(3) add-mset-add-single li-trans-l union-mset-add-mset-right*)

**next**

**case** 2 **then obtain**  $y$  **where**  $y \in \text{set-mset } Y2$   $(z, y) \in s$  **using**  $*(4)$  *add(3, 4)*

**by** (*auto simp: locally-irrefl-def*)

**moreover from** *this transD[OF <trans s> - this(2)]*

**have**  $x' \in \text{set-mset } X2 \implies \exists y \in \text{set-mset } Y2. (x', y) \in s$  **for**  $x'$

**using** 2  $*(4)$  [*rule-format, of x'*] **by** *auto*

**ultimately show**  $?thesis$  **using**  $*$  *one-step-implies-mult[of Y2 X2 s Z']* 2 *add(3, 4)*

**by** (*force simp: locally-irrefl-def add.commute[of {#-#}] add.assoc[symmetric] intro: add(1)*)

**qed**

**qed** *auto*

**next**

**assume**  $?R$  **then obtain**  $I J K$

**where**  $Y = I + J$   $X = I + K$   $J \neq \{\#\}$   $\forall k \in \text{set-mset } K. \exists j \in \text{set-mset } J. (k, j) \in s$

**using** *mult-implies-one-step[OF <trans s>]* **by** *blast*

**thus**  $?L$  **using** *one-step-implies-mult[of J K s I + Z]* **by** (*auto simp: ac-simps*)

**qed**

**lemma** *mult-locally-cancelL:*

**assumes** *trans s locally-irrefl s (X + Z) locally-irrefl s (Y + Z)*  
**shows**  $(Z + X, Z + Y) \in \text{mult } s \longleftrightarrow (X, Y) \in \text{mult } s$   
**using** *mult-locally-cancel[OF assms]* **by** (*simp only: union-commute*)

**lemma** *mult-cancelL*:

**assumes** *trans s irrefl s* **shows**  $(Z + X, Z + Y) \in \text{mult } s \longleftrightarrow (X, Y) \in \text{mult } s$   
**using** *assms*  
**by** (*auto simp: union-commute intro!: mult-cancel elim: irrefl-on-subset*)

**lemma** *wf-trancl-conv*:

**shows**  $wf (r^+) \longleftrightarrow wf r$   
**using** *wf-subset[of r<sup>+</sup> r]* **by** (*force simp: wf-trancl*)

### 3.1 Pointwise multiset order

**inductive-set** *multpw* :: '*a* rel  $\Rightarrow$  '*a* multiset rel **for** *ns* :: '*a* rel **where**

*empty*:  $(\{\#\}, \{\#\}) \in \text{multpw } ns$   
| *add*:  $(x, y) \in ns \Longrightarrow (X, Y) \in \text{multpw } ns \Longrightarrow (\text{add-mset } x X, \text{add-mset } y Y) \in \text{multpw } ns$

**lemma** *multpw-emptyL* [*simp*]:

$(\{\#\}, X) \in \text{multpw } ns \longleftrightarrow X = \{\#\}$   
**by** (*cases X*) (*auto elim: multpw.cases intro: multpw.intros*)

**lemma** *multpw-emptyR* [*simp*]:

$(X, \{\#\}) \in \text{multpw } ns \longleftrightarrow X = \{\#\}$   
**by** (*cases X*) (*auto elim: multpw.cases intro: multpw.intros*)

**lemma** *refl-multpw*:

**assumes** *refl ns* **shows** *refl (multpw ns)*

**proof** –

**have**  $(X, X) \in \text{multpw } ns$  **for** *X* **using** *assms*  
**by** (*induct X*) (*auto intro: multpw.intros simp: refl-on-def*)  
**then show** *?thesis* **by** (*auto simp: refl-on-def*)

**qed**

**lemma** *multpw-Id-Id* [*simp*]:

*multpw Id = Id*

**proof** –

**have**  $(X, Y) \in \text{multpw } (Id :: 'a \text{ rel}) \Longrightarrow X = Y$  **for** *X Y* **by** (*induct X Y rule: multpw.induct*) *auto*

**then show** *?thesis* **using** *refl-multpw[of Id]* **by** (*auto simp: refl-on-def*)

**qed**

**lemma** *mono-multpw*:

**assumes**  $ns \subseteq ns'$  **shows**  $\text{multpw } ns \subseteq \text{multpw } ns'$

**proof** –

**have**  $(X, Y) \in \text{multpw } ns \Longrightarrow (X, Y) \in \text{multpw } ns'$  **for** *X Y*  
**by** (*induct X Y rule: multpw.induct*) (*insert assms, auto intro: multpw.intros*)

**then show** *?thesis* **by** *auto*  
**qed**

**lemma** *multpw-converse*:

$\text{multpw } (ns^{-1}) = (\text{multpw } ns)^{-1}$

**proof** –

**have**  $(X, Y) \in \text{multpw } (ns^{-1}) \implies (X, Y) \in (\text{multpw } ns)^{-1}$  **for**  $X\ Y$  **and**  $ns :: 'a\ \text{rel}$

**by** (*induct*  $X\ Y$  *rule*: *multpw.induct*) (*auto* *intro*: *multpw.intros*)

**then show** *?thesis* **by** *auto*

**qed**

**lemma** *multpw-local*:

$(X, Y) \in \text{multpw } ns \implies (X, Y) \in \text{multpw } (ns \cap \text{set-mset } X \times \text{set-mset } Y)$

**proof** (*induct*  $X\ Y$  *rule*: *multpw.induct*)

**case** (*add*  $x\ y\ X\ Y$ ) **then show** *?case*

**using** *mono-multpw*[*of*  $ns \cap \text{set-mset } X \times \text{set-mset } Y\ ns \cap \text{insert } x\ (\text{set-mset } X) \times \text{insert } y\ (\text{set-mset } Y)$ ]

**by** (*auto* *intro*: *multpw.intros*)

**qed** *auto*

**lemma** *multpw-split1R*:

**assumes**  $(\text{add-mset } x\ X, Y) \in \text{multpw } ns$

**obtains**  $z\ Z$  **where**  $Y = \text{add-mset } z\ Z$  **and**  $(x, z) \in ns$  **and**  $(X, Z) \in \text{multpw } ns$

**using** *assms*

**proof** (*induct*  $\text{add-mset } x\ X\ Y$  *arbitrary*:  $X$  *thesis* *rule*: *multpw.induct*)

**case** (*add*  $x'\ y'\ X'\ Y'$ ) **then show** *?case*

**proof** (*cases*  $x = x'$ )

**case** *False*

**obtain**  $X''$  **where** [*simp*]:  $X = \text{add-mset } x'\ X''$

**using** *add(4)* *False*

**by** (*metis* *add-eq-conv-diff*)

**have**  $X' = \text{add-mset } x\ X''$  **using** *add(4)* **by** (*auto* *simp*: *add-eq-conv-ex*)

**with** *add(2)* **obtain**  $Y''\ y$  **where**  $Y' = \text{add-mset } y\ Y''$   $(x, y) \in ns$   $(X'', Y'')$

$\in \text{multpw } ns$

**by** (*auto* *intro*: *add(3)*)

**then show** *?thesis* **using** *add(1)* *add(5)*[*of*  $y\ \text{add-mset } y'\ Y''$ ]

**by** (*auto* *simp*: *ac-simps* *intro*: *multpw.intros*)

**qed** *auto*

**qed** *auto*

**lemma** *multpw-splitR*:

**assumes**  $(X1 + X2, Y) \in \text{multpw } ns$

**obtains**  $Y1\ Y2$  **where**  $Y = Y1 + Y2$  **and**  $(X1, Y1) \in \text{multpw } ns$  **and**  $(X2, Y2)$

$\in \text{multpw } ns$

**using** *assms*

**proof** (*induct*  $X2$  *arbitrary*:  $Y$  *thesis*)

**case** (*add*  $x2\ X2$ )

**from** *add(3)* **obtain**  $Y'\ y2$  **where**  $(X1 + X2, Y') \in \text{multpw } ns$   $(x2, y2) \in ns$   $Y$

= *add-mset* *y2* *Y'*  
 by (*auto elim: multpw-split1R simp: union-assoc[symmetric]*)  
 moreover then obtain *Y1 Y2* where  $(X1, Y1) \in \text{multpw } ns$   $(X2, Y2) \in \text{multpw } ns$   $Y' = Y1 + Y2$   
 by (*auto elim: add(1)[rotated]*)  
 ultimately show ?*case* by (*intro add(2)*) (*auto simp: union-assoc intro: multpw.intros*)  
 qed *auto*

**lemma** *multpw-split1L*:  
 assumes  $(X, \text{add-mset } y \ Y) \in \text{multpw } ns$   
 obtains *z Z* where  $X = \text{add-mset } z \ Z$  and  $(z, y) \in ns$  and  $(Z, Y) \in \text{multpw } ns$   
 using *assms multpw-split1R[of y Y X ns<sup>-1</sup> thesis]* by (*auto simp: multpw-converse*)

**lemma** *multpw-splitL*:  
 assumes  $(X, Y1 + Y2) \in \text{multpw } ns$   
 obtains *X1 X2* where  $X = X1 + X2$  and  $(X1, Y1) \in \text{multpw } ns$  and  $(X2, Y2) \in \text{multpw } ns$   
 using *assms multpw-splitR[of Y1 Y2 X ns<sup>-1</sup> thesis]* by (*auto simp: multpw-converse*)

**lemma** *locally-trans-multpw*:  
 assumes *locally-trans ns S T U*  
 and  $(S, T) \in \text{multpw } ns$   
 and  $(T, U) \in \text{multpw } ns$   
 shows  $(S, U) \in \text{multpw } ns$   
 using *assms(2,3,1)*  
**proof** (*induct S T arbitrary: U rule: multpw.induct*)  
 case (*add x y X Y*)  
 then show ?*case* **unfolding** *locally-trans-def*  
 by (*auto 0 3 intro: multpw.intros elim: multpw-split1R*)  
 qed *blast*

**lemma** *trans-multpw*:  
 assumes *trans ns* shows *trans (multpw ns)*  
 using *locally-trans-multpw unfolding locally-trans-def trans-def*  
 by (*meson assms locally-trans-multpw tr-ltr*)

**lemma** *multpw-add*:  
 assumes  $(X1, Y1) \in \text{multpw } ns$   $(X2, Y2) \in \text{multpw } ns$  shows  $(X1 + X2, Y1 + Y2) \in \text{multpw } ns$   
 using *assms(2,1)*  
 by (*induct X2 Y2 rule: multpw.induct*) (*auto intro: multpw.intros simp: add.assoc[symmetric]*)

**lemma** *multpw-single*:  
 $(x, y) \in ns \implies (\{ \#x \}, \{ \#y \}) \in \text{multpw } ns$   
 using *multpw.intros(2)[OF - multpw.intros(1)]* .

**lemma** *multpw-mult1-commute*:  
 assumes *compat: s O ns  $\subseteq$  s* and *refl ns*  
 shows *mult1 s O multpw ns  $\subseteq$  multpw ns O mult1 s*

**proof** –  
 { **fix**  $X Y Z$  **assume**  $1: (X, Y) \in \text{mult1 } s (Y, Z) \in \text{multpw } ns$   
**then obtain**  $X' Y' y$  **where**  $2: X = Y' + X' Y = \text{add-mset } y Y' \forall x. x \in \#$   
 $X' \longrightarrow (x, y) \in s$   
**by** (*auto simp: mult1-def*)  
**moreover obtain**  $Z' z$  **where**  $3: Z = \text{add-mset } z Z' (y, z) \in ns (Y', Z') \in$   
 $\text{multpw } ns$   
**using**  $1(2) 2(2)$  **by** (*auto elim: multpw-split1R*)  
**moreover have**  $\forall x. x \in \# X' \longrightarrow (x, z) \in s$  **using**  $2(3) 3(2)$  **compat by blast**  
**ultimately have**  $\exists Y'. (X, Y') \in \text{multpw } ns \wedge (Y', Z) \in \text{mult1 } s$  **unfolding**  
 $\text{mult1-def}$   
**using**  $\text{refl-multpw}[OF \text{ reflns}]$   
**by** (*intro exI[of - Z' + X'] (auto intro: multpw-add simp: refl-on-def)*)  
 }  
**then show** *?thesis* **by fast**  
**qed**

**lemma** *multpw-mult-commute*:  
**assumes**  $s O ns \subseteq s \text{ refl } ns$  **shows**  $\text{mult } s O \text{multpw } ns \subseteq \text{multpw } ns O \text{mult } s$   
**proof** –  
 { **fix**  $X Y Z$  **assume**  $1: (X, Y) \in \text{mult } s (Y, Z) \in \text{multpw } ns$   
**then have**  $\exists Y'. (X, Y') \in \text{multpw } ns \wedge (Y', Z) \in \text{mult } s$  **unfolding** *mult-def*  
**using**  $\text{multpw-mult1-commute}[OF \text{ assms}]$  **by** (*induct rule: converse-trancl-induct*)  
 (*auto 0 3*)  
 }  
**then show** *?thesis* **by fast**  
**qed**

**lemma** *wf-mult-rel-multpw*:  
**assumes**  $wf s s O ns \subseteq s \text{ refl } ns$  **shows**  $wf ((\text{multpw } ns)^* O \text{mult } s O (\text{multpw } ns)^*)$   
**using**  $\text{assms}(1) \text{multpw-mult-commute}[OF \text{ assms}(2,3)]$  **by** (*subst qc-wf-relto-iff*)  
 (*auto simp: wf-mult*)

**lemma** *multpw-cancel1*:  
**assumes**  $\text{trans } ns (y, x) \in ns$   
**shows**  $(\text{add-mset } x X, \text{add-mset } y Y) \in \text{multpw } ns \implies (X, Y) \in \text{multpw } ns$  (**is**  
 $?L \implies ?R$ )  
**proof** –  
**assume**  $?L$  **then obtain**  $x' X'$  **where**  $X: (x', y) \in ns \text{add-mset } x' X' = \text{add-mset}$   
 $x X (X', Y) \in \text{multpw } ns$   
**by** (*auto elim: multpw-split1L simp: union-assoc[symmetric]*)  
**then show**  $?R$   
**proof** (*cases x = x'*)  
**case False** **then obtain**  $X2$  **where**  $X2: X' = \text{add-mset } x X2 X = \text{add-mset } x'$   
 $X2$   
**using**  $X(2)$  **by** (*auto simp: add-eq-conv-ex*)  
**then obtain**  $y' Y'$  **where**  $Y: (x, y') \in ns Y = \text{add-mset } y' Y' (X2, Y') \in$   
 $\text{multpw } ns$

**using**  $X(3)$  **by** (*auto elim: multpw-split1R*)  
**have**  $(x', y') \in ns$  **using**  $X(1)$   $Y(1)$   $\langle trans\ ns \rangle$  *assms(2)* **by** (*metis trans-def*)  
**then show** *?thesis* **using**  $Y$  **by** (*auto intro: multpw.intros simp: X2*)  
**qed** *auto*  
**qed**

**lemma** *multpw-cancel*:  
**assumes** *refl ns trans ns*  
**shows**  $(X + Z, Y + Z) \in multpw\ ns \longleftrightarrow (X, Y) \in multpw\ ns$  (**is**  $?L \longleftrightarrow ?R$ )  
**proof**  
**assume**  $?L$  **then show**  $?R$   
**proof** (*induct Z*)  
**case** (*add z Z*) **then show**  $?case$  **using** *multpw-cancel1*[*of ns z z X + Z Y + Z*] *assms*  
**by** (*auto simp: refl-on-def union-assoc*)  
**qed** *auto*  
**next**  
**assume**  $?R$  **then show**  $?L$  **using** *assms refl-multpw* **by** (*auto intro: multpw-add simp: refl-on-def*)  
**qed**

**lemma** *multpw-cancelL*:  
**assumes** *refl ns trans ns* **shows**  $(Z + X, Z + Y) \in multpw\ ns \longleftrightarrow (X, Y) \in multpw\ ns$   
**using** *multpw-cancel*[*OF assms, of X Z Y*] **by** (*simp only: union-commute*)

### 3.2 Multiset extension for order pairs via the pointwise order and *mult*

**definition** *mult2-s ns s*  $\equiv multpw\ ns\ O\ mult\ s$   
**definition** *mult2-ns ns s*  $\equiv multpw\ ns\ O\ (mult\ s)^=$

**lemma** *mult2-ns-conv*:  
**shows**  $mult2-ns\ ns\ s = mult2-s\ ns\ s \cup multpw\ ns$   
**by** (*auto simp: mult2-s-def mult2-ns-def*)

**lemma** *mono-mult2-s*:  
**assumes**  $ns \subseteq ns' \ s \subseteq s'$  **shows**  $mult2-s\ ns\ s \subseteq mult2-s\ ns'\ s'$   
**using** *mono-multpw*[*OF assms(1)*] *mono-mult*[*OF assms(2)*] **unfolding** *mult2-s-def*  
**by** *auto*

**lemma** *mono-mult2-ns*:  
**assumes**  $ns \subseteq ns' \ s \subseteq s'$  **shows**  $mult2-ns\ ns\ s \subseteq mult2-ns\ ns'\ s'$   
**using** *mono-multpw*[*OF assms(1)*] *mono-mult*[*OF assms(2)*] **unfolding** *mult2-ns-def*  
**by** *auto*

**lemma** *wf-mult2-s*:  
**assumes**  $wf\ s\ s\ O\ ns \subseteq s\ refl\ ns$   
**shows**  $wf\ (mult2-s\ ns\ s)$

**using** *wf-mult-rel-multipw*[*OF assms*] *assms* **by** (*auto simp: mult2-s-def wf-mult intro: wf-subset*)

**lemma** *refl-mult2-ns*:

**assumes** *refl ns* **shows** *refl (mult2-ns ns s)*

**using** *refl-multipw*[*OF assms*] **unfolding** *mult2-ns-def refl-on-def* **by** *fast*

**lemma** *trans-mult2-s*:

**assumes** *s O ns ⊆ s refl ns trans ns*

**shows** *trans (mult2-s ns s)*

**using** *trans-multipw*[*OF assms(3)*] *trans-trancl*[*of mult1 s, folded mult-def*] *multipw-mult-commute*[*OF assms(1,2)*]

**unfolding** *mult2-s-def trans-O-iff* **by** (*blast 8*)

**lemma** *trans-mult2-ns*:

**assumes** *s O ns ⊆ s refl ns trans ns*

**shows** *trans (mult2-ns ns s)*

**using** *trans-multipw*[*OF assms(3)*] *trans-trancl*[*of mult1 s, folded mult-def*] *multipw-mult-commute*[*OF assms(1,2)*]

**unfolding** *mult2-ns-def trans-O-iff* **by** (*blast 8*)

**lemma** *compat-mult2*:

**assumes** *s O ns ⊆ s refl ns trans ns*

**shows** *mult2-ns ns s O mult2-s ns s ⊆ mult2-s ns s mult2-s ns s O mult2-ns ns*

*s ⊆ mult2-s ns s*

**using** *trans-multipw*[*OF assms(3)*] *trans-trancl*[*of mult1 s, folded mult-def*] *multipw-mult-commute*[*OF assms(1,2)*]

**unfolding** *mult2-s-def mult2-ns-def trans-O-iff* **by** (*blast 8*)+

Trivial inclusions

**lemma** *mult-implies-mult2-s*:

**assumes** *refl ns (X, Y) ∈ mult s*

**shows** *(X, Y) ∈ mult2-s ns s*

**using** *refl-multipw*[*of ns*] *assms* **unfolding** *mult2-s-def refl-on-def* **by** *blast*

**lemma** *mult-implies-mult2-ns*:

**assumes** *refl ns (X, Y) ∈ (mult s)<sup>=</sup>*

**shows** *(X, Y) ∈ mult2-ns ns s*

**using** *refl-multipw*[*of ns*] *assms* **unfolding** *mult2-ns-def refl-on-def* **by** *blast*

**lemma** *multipw-implies-mult2-ns*:

**assumes** *(X, Y) ∈ multipw ns*

**shows** *(X, Y) ∈ mult2-ns ns s*

**unfolding** *mult2-ns-def* **using** *assms* **by** *simp*

### 3.3 One-step versions of the multiset extensions

**lemma** *mult2-s-one-step*:

**assumes** *ns O s ⊆ s refl ns trans s*

**shows**  $(X, Y) \in \text{mult2-s ns } s \iff (\exists X1 X2 Y1 Y2. X = X1 + X2 \wedge Y = Y1 + Y2 \wedge$   
 $(X1, Y1) \in \text{multpw ns} \wedge Y2 \neq \{\#\} \wedge (\forall x. x \in\# X2 \longrightarrow (\exists y. y \in\# Y2 \wedge (x, y) \in s)))$  **(is ?L  $\iff$  ?R)**

**proof**  
**assume ?R then obtain**  $X1 X2 Y1 Y2$  **where**  $*$ :  $X = X1 + X2 \ Y = Y1 + Y2$   
 $(X1, Y1) \in \text{multpw ns}$  **and**  
 $Y2 \neq \{\#\} \ \forall x. x \in\# X2 \longrightarrow (\exists y. y \in\# Y2 \wedge (x, y) \in s)$  **by** *blast*  
**then have**  $(Y1 + X2, Y1 + Y2) \in \text{mult } s$   
**using**  $\langle \text{trans } s \rangle$  **by** *(auto intro: one-step-implies-mult)*  
**moreover have**  $(X1 + X2, Y1 + X2) \in \text{multpw ns}$   
**using**  $\langle \text{refl ns} \rangle$  *refl-multpw[of ns]* **by** *(auto intro: multpw-add simp: refl-on-def \*)*  
**ultimately show ?L by** *(auto simp: mult2-s-def \*)*  
**next**  
**assume ?L then obtain**  $X1 X2 Z1 Z2 Y2$  **where**  $*$ :  $X = X1 + X2 \ Y = Z1 + Y2$   
 $(X1, Z1) \in \text{multpw ns}$   
 $(X2, Z2) \in \text{multpw ns} \ Y2 \neq \{\#\} \ \forall x. x \in\# Z2 \longrightarrow (\exists y. y \in\# Y2 \wedge (x, y) \in s)$   
**by** *(auto 0 3 dest!: mult-implies-one-step[OF  $\langle \text{trans } s \rangle$ ] simp: mult2-s-def elim!: multpw-splitL) metis*  
**have**  $\forall x. x \in\# X2 \longrightarrow (\exists y. y \in\# Y2 \wedge (x, y) \in s)$   
**proof** *(intro allI impI)*  
**fix**  $x$  **assume**  $x \in\# X2$   
**then obtain**  $X2'$  **where**  $X2 = \text{add-mset } x \ X2'$  **by** *(metis multi-member-split)*  
**then obtain**  $z \ Z2'$  **where**  $Z2 = \text{add-mset } z \ Z2'$   $(x, z) \in \text{ns}$  **using**  $*(4)$  **by**  
*(auto elim: multpw-split1R)*  
**then have**  $z \in\# Z2 \ (x, z) \in \text{ns}$  **by** *auto*  
**then show**  $\exists y. y \in\# Y2 \wedge (x, y) \in s$  **using**  $*(6)$   $\langle \text{ns } O \ s \subseteq s \rangle$  **by** *blast*  
**qed**  
**then show ?R using**  $*$  **by** *auto*  
**qed**

**lemma** *mult2-ns-one-step*:  
**assumes**  $\text{ns } O \ s \subseteq s \ \text{refl ns trans } s$   
**shows**  $(X, Y) \in \text{mult2-ns ns } s \iff (\exists X1 X2 Y1 Y2. X = X1 + X2 \wedge Y = Y1 + Y2 \wedge$   
 $(X1, Y1) \in \text{multpw ns} \wedge (\forall x. x \in\# X2 \longrightarrow (\exists y. y \in\# Y2 \wedge (x, y) \in s)))$  **(is ?L  $\iff$  ?R)**

**proof**  
**assume ?L then consider**  $(X, Y) \in \text{multpw ns} \mid (X, Y) \in \text{mult2-s ns } s$   
**by** *(auto simp: mult2-s-def mult2-ns-def)*  
**then show ?R using** *mult2-s-one-step[OF assms]*  
**by** *(cases, intro exI[of -  $\{\#\}$ ], THEN exI[of -  $Y$ ], THEN exI[of -  $\{\#\}$ ], THEN exI[of -  $X$ ]]) auto*  
**next**  
**assume ?R then obtain**  $X1 X2 Y1 Y2$  **where**  $X = X1 + X2 \ Y = Y1 + Y2$   
 $(X1, Y1) \in \text{multpw ns} \ \forall x. x \in\# X2 \longrightarrow (\exists y. y \in\# Y2 \wedge (x, y) \in s)$  **by** *blast*  
**then show ?L using** *mult2-s-one-step[OF assms, of X Y] count-inject[of X2*

{#}]  
 by (cases  $Y2 = \{#\}$ ) (auto simp: mult2-s-def mult2-ns-def)  
 qed

**lemma** *mult2-s-locally-one-step'*:

assumes  $ns \ O \ s \subseteq \ s \ refl \ ns \ locally\text{-irrefl} \ s \ X \ locally\text{-irrefl} \ s \ Y \ trans \ s$   
 shows  $(X, Y) \in \ mult2\text{-}s \ ns \ s \longleftrightarrow (\exists X1 \ X2 \ Y1 \ Y2. X = X1 + X2 \wedge Y = Y1 + Y2 \wedge (X1, Y1) \in \ multpw \ ns \wedge (X2, Y2) \in \ mult \ s)$  (is ?L  $\longleftrightarrow$  ?R)

**proof**

assume ?L then show ?R **unfolding** *mult2-s-one-step*[OF *assms*(1,2,5)]  
 using *one-step-implies-mult*[of - - s {#}] **by** *auto metis*

**next**

assume ?R then obtain  $X1 \ X2 \ Y1 \ Y2$  where  $x: X = X1 + X2$  and  $y: Y = Y1 + Y2$  and

$ns: (X1, Y1) \in \ multpw \ ns$  and  $s: (X2, Y2) \in \ mult \ s$  **by** *blast*

then have  $l: locally\text{-irrefl} \ s \ (X2 + Y1)$  and  $r: locally\text{-irrefl} \ s \ (Y2 + Y1)$

using *assms*(3, 4) **by** (auto simp *add: locally-irrefl-def*)

show ?L using  $ns \ s \ mult\text{-}locally\text{-cancel}$ [OF *assms*(5)  $l \ r$ ] *multpw-add*[OF  $ns$ , of  $X2 \ X2$ ] *refl-multpw*[OF  $\langle refl \ ns \rangle$ ]

**unfolding** *mult2-s-def refl-on-def*  $x \ y$  **by** *auto*

qed

**lemma** *mult2-s-one-step'*:

assumes  $ns \ O \ s \subseteq \ s \ refl \ ns \ irrefl \ s \ trans \ s$

shows  $(X, Y) \in \ mult2\text{-}s \ ns \ s \longleftrightarrow (\exists X1 \ X2 \ Y1 \ Y2. X = X1 + X2 \wedge Y = Y1 + Y2 \wedge (X1, Y1) \in \ multpw \ ns \wedge (X2, Y2) \in \ mult \ s)$  (is ?L  $\longleftrightarrow$  ?R)

using *assms* *mult2-s-locally-one-step'* **by** (*simp add: mult2-s-locally-one-step' irrefl-def locally-irrefl-def*)

**lemma** *mult2-ns-one-step'*:

assumes  $ns \ O \ s \subseteq \ s \ refl \ ns \ irrefl \ s \ trans \ s$

shows  $(X, Y) \in \ mult2\text{-}ns \ ns \ s \longleftrightarrow (\exists X1 \ X2 \ Y1 \ Y2. X = X1 + X2 \wedge Y = Y1 + Y2 \wedge (X1, Y1) \in \ multpw \ ns \wedge (X2, Y2) \in (\mult \ s)^=)$  (is ?L  $\longleftrightarrow$  ?R)

**proof** –

have  $(X, Y) \in \ multpw \ ns \implies ?R$   
 by (*intro exI*[of - {#}], *THEN exI*[of -  $Y$ ], *THEN exI*[of - {#}], *THEN exI*[of -  $X$ ]]]) *auto*

moreover have  $X = X1 + Y2 \wedge Y = Y1 + Y2 \wedge (X1, Y1) \in \ multpw \ ns \implies ?L$  for  $X1 \ Y1 \ Y2$

using *multpw-add*[of  $X1 \ Y1 \ ns \ Y2 \ Y2$ ] *refl-multpw*[OF  $\langle refl \ ns \rangle$ ] **by** (auto simp: *mult2-ns-def refl-on-def*)

ultimately show *?thesis* using *mult2-s-one-step'*[OF *assms*] **unfolding** *mult2-ns-conv* **by** *auto blast*

**by** *auto blast*

qed

### 3.4 Cancellation

**lemma** *mult2-s-locally-cancel1*:

**assumes**  $s \ O \ ns \subseteq s \ ns \ O \ s \subseteq s \ refl \ ns \ trans \ ns \ locally\text{-irrefl} \ s \ (add\text{-mset} \ z \ X)$   
*locally-irrefl*  $s \ (add\text{-mset} \ z \ Y) \ trans \ s$

$(add\text{-mset} \ z \ X, add\text{-mset} \ z \ Y) \in \text{mult2-s} \ ns \ s$

**shows**  $(X, Y) \in \text{mult2-s} \ ns \ s$

**proof** –

**obtain**  $X1 \ X2 \ Y1 \ Y2$  **where**  $*$ :  $add\text{-mset} \ z \ X = X1 + X2 \ add\text{-mset} \ z \ Y = Y1 + Y2$   $(X1, Y1) \in \text{multpw} \ ns$

$(X2, Y2) \in \text{mult} \ s$  **using** *assms(8)* **unfolding** *mult2-s-locally-one-step'*[*OF* *assms(2,3,5,6,7)*] **by** *blast*

**from** *union-single-eq-member*[*OF* *this(1)*] *union-single-eq-member*[*OF* *this(2)*] *multi-member-split*

**consider**  $X1'$  **where**  $X1 = add\text{-mset} \ z \ X1' \mid Y1'$  **where**  $Y1 = add\text{-mset} \ z \ Y1' \mid X2' \ Y2'$  **where**  $X2 = add\text{-mset} \ z \ X2' \ Y2 = add\text{-mset} \ z \ Y2'$

**unfolding** *set-mset-union* *Un-iff* **by** *metis*

**then show** *?thesis*

**proof** (*cases*)

**case 1** **then obtain**  $Y1' \ z'$  **where**  $**$ :  $(X1', Y1') \in \text{multpw} \ ns \ Y1 = add\text{-mset} \ z' \ Y1' \ (z, z') \in ns$

**using**  $*$  **by** (*auto elim: multpw-split1R*)

**then have**  $(X, Y1' + Y2) \in \text{mult2-s} \ ns \ s$  **using**  $*$  1

**by** *auto* (*metis add-mset-add-single assms(2 – 7) li-trans-l mult2-s-locally-one-step'*)

**moreover**

**have**  $(Y1' + Y2, Y) \in \text{multpw} \ ns$

**using** *refl-multpw*[*OF*  $\langle refl \ ns \rangle$ ]  $*$   $**$  *multpw-cancel1*[*OF*  $\langle trans \ ns \rangle$ ]  $**$ (3), *of*  $Y1' + Y2 \ Y$

**by** (*auto simp: refl-on-def*)

**ultimately show** *?thesis* **using** *compat-mult2*[*OF* *assms(1,3,4)*] **unfolding** *mult2-ns-conv* **by** *blast*

**next**

**case 2** **then obtain**  $X1' \ z'$  **where**  $**$ :  $(X1', Y1') \in \text{multpw} \ ns \ X1 = add\text{-mset} \ z' \ X1' \ (z', z) \in ns$

**using**  $*$  **by** (*auto elim: multpw-split1L*)

**then have**  $(X1' + X2, Y) \in \text{mult2-s} \ ns \ s$  **using**  $*$  2

**by** *auto* (*metis add-mset-add-single assms(2 – 7) li-trans-l mult2-s-locally-one-step'*)

**moreover**

**have**  $(X, X1' + X2) \in \text{multpw} \ ns$

**using** *refl-multpw*[*OF*  $\langle refl \ ns \rangle$ ]  $*$   $**$  *multpw-cancel1*[*OF*  $\langle trans \ ns \rangle$ ]  $**$ (3), *of*  $X \ X1' + X2$

**by** (*auto simp: refl-on-def*)

**ultimately show** *?thesis* **using** *compat-mult2*[*OF* *assms(1,3,4)*] **unfolding** *mult2-ns-conv* **by** *blast*

**next**

**case 3** **then show** *?thesis* **using** *assms*  $*$

**by** (*auto simp: mult2-s-locally-one-step' union-commute*[*of*  $\{\#\text{-}\#\}$ ] *union-assoc*[*symmetric*] *mult-cancel* *mult-cancel-add-mset*)

(*metis*  $*$ (1)  $*$ (2) *add-mset-add-single li-trans-l li-trans-r mult2-s-locally-one-step'*)

*mult-locally-cancel*)

**qed**  
**qed**

**lemma** *mult2-s-cancel1*:

**assumes**  $s \ O \ ns \subseteq s \ ns \ O \ s \subseteq s \ refl \ ns \ trans \ ns \ irrefl \ s \ trans \ s$  (*add-mset z X*,  
*add-mset z Y*)  $\in \ mult2-s \ ns \ s$

**shows**  $(X, Y) \in \ mult2-s \ ns \ s$

**using** *assms mult2-s-locally-cancel1* **by** (*metis irrefl-def locally-irrefl-def*)

**lemma** *mult2-s-locally-cancel*:

**assumes**  $s \ O \ ns \subseteq s \ ns \ O \ s \subseteq s \ refl \ ns \ trans \ ns \ locally-irrefl \ s \ (X + Z) \ locally-irrefl$   
 $s \ (Y + Z) \ trans \ s$

**shows**  $(X + Z, Y + Z) \in \ mult2-s \ ns \ s \implies (X, Y) \in \ mult2-s \ ns \ s$

**using** *assms(5, 6)*

**proof** (*induct Z*)

**case** (*add z Z*) **then show** *?case*

**using** *mult2-s-locally-cancel1[OF assms(1-4), of z X + Z Y + Z]*

**by** *auto (metis add-mset-add-single assms(7) li-trans-l)*

**qed** *auto*

**lemma** *mult2-s-cancel*:

**assumes**  $s \ O \ ns \subseteq s \ ns \ O \ s \subseteq s \ refl \ ns \ trans \ ns \ irrefl \ s \ trans \ s$

**shows**  $(X + Z, Y + Z) \in \ mult2-s \ ns \ s \implies (X, Y) \in \ mult2-s \ ns \ s$

**using** *mult2-s-locally-cancel assms* **by** (*metis irrefl-def locally-irrefl-def*)

**lemma** *mult2-ns-cancel*:

**assumes**  $s \ O \ ns \subseteq s \ ns \ O \ s \subseteq s \ refl \ ns \ trans \ s \ irrefl \ s \ trans \ ns$

**shows**  $(X + Z, Y + Z) \in \ mult2-s \ ns \ s \implies (X, Y) \in \ mult2-ns \ ns \ s$

**unfolding** *mult2-ns-conv* **using** *assms mult2-s-cancel multpw-cancel* **by** *blast*

### 3.5 Implementation friendly versions of *mult2-s* and *mult2-ns*

**definition** *mult2-alt*  $:: \ bool \implies 'a \ rel \implies 'a \ rel \implies 'a \ multiset \ rel$  **where**

$\mult2-alt \ b \ ns \ s = \{(X, Y). (\exists X1 \ X2 \ Y1 \ Y2. X = X1 + X2 \wedge Y = Y1 + Y2 \wedge$   
 $(X1, Y1) \in \ multpw \ ns \wedge (b \vee Y2 \neq \{\#\}) \wedge (\forall x. x \in\# \ X2 \longrightarrow (\exists y. y \in\# \ Y2$   
 $\wedge (x, y) \in \ s))\}$

**lemma** *mult2-altI*:

**assumes**  $X = X1 + X2 \ Y = Y1 + Y2 \ (X1, Y1) \in \ multpw \ ns$

$b \vee Y2 \neq \{\#\} \ \forall x. x \in\# \ X2 \longrightarrow (\exists y. y \in\# \ Y2 \wedge (x, y) \in \ s)$

**shows**  $(X, Y) \in \ mult2-alt \ b \ ns \ s$

**using** *assms unfolding mult2-alt-def* **by** *blast*

**lemma** *mult2-altE*:

**assumes**  $(X, Y) \in \ mult2-alt \ b \ ns \ s$

**obtains**  $X1 \ X2 \ Y1 \ Y2$  **where**  $X = X1 + X2 \ Y = Y1 + Y2 \ (X1, Y1) \in \ multpw$   
 $ns$

$b \vee Y2 \neq \{\#\} \ \forall x. x \in\# \ X2 \longrightarrow (\exists y. y \in\# \ Y2 \wedge (x, y) \in \ s)$

using *assms* **unfolding** *mult2-alt-def* **by** *blast*

**lemma** *mono-mult2-alt*:

**assumes**  $ns \subseteq ns' \ s \subseteq s'$  **shows**  $mult2\text{-}alt \ b \ ns \ s \subseteq mult2\text{-}alt \ b \ ns' \ s'$   
**unfolding** *mult2-alt-def* **using** *mono-multpw*[*OF assms*(1)] *assms* **by** (*blast 19*)

**abbreviation**  $mult2\text{-}alt\text{-}s \equiv mult2\text{-}alt \ False$

**abbreviation**  $mult2\text{-}alt\text{-}ns \equiv mult2\text{-}alt \ True$

**lemmas**  $mult2\text{-}alt\text{-}s\text{-}def = mult2\text{-}alt\text{-}def$ [**where**  $b = False$ , *unfolded simp-thms*]

**lemmas**  $mult2\text{-}alt\text{-}ns\text{-}def = mult2\text{-}alt\text{-}def$ [**where**  $b = True$ , *unfolded simp-thms*]

**lemmas**  $mult2\text{-}alt\text{-}sI = mult2\text{-}altI$ [**where**  $b = False$ , *unfolded simp-thms*]

**lemmas**  $mult2\text{-}alt\text{-}nsI = mult2\text{-}altI$ [**where**  $b = True$ , *unfolded simp-thms True-implies-equals*]

**lemmas**  $mult2\text{-}alt\text{-}sE = mult2\text{-}altE$ [**where**  $b = False$ , *unfolded simp-thms*]

**lemmas**  $mult2\text{-}alt\text{-}nsE = mult2\text{-}altE$ [**where**  $b = True$ , *unfolded simp-thms True-implies-equals*]

**Equivalence to  $mult2\text{-}s$  and  $mult2\text{-}ns$**  **lemma** *mult2-s-eq-mult2-s-alt*:

**assumes**  $ns \ O \ s \subseteq s \ refl \ ns \ trans \ s$

**shows**  $mult2\text{-}alt\text{-}s \ ns \ s = mult2\text{-}s \ ns \ s$

**using** *mult2-s-one-step*[*OF assms*] **unfolding** *mult2-alt-s-def* **by** *blast*

**lemma** *mult2-ns-eq-mult2-ns-alt*:

**assumes**  $ns \ O \ s \subseteq s \ refl \ ns \ trans \ s$

**shows**  $mult2\text{-}alt\text{-}ns \ ns \ s = mult2\text{-}ns \ ns \ s$

**using** *mult2-ns-one-step*[*OF assms*] **unfolding** *mult2-alt-ns-def* **by** *blast*

**lemma** *mult2-alt-local*:

**assumes**  $(X, Y) \in mult2\text{-}alt \ b \ ns \ s$

**shows**  $(X, Y) \in mult2\text{-}alt \ b \ (ns \cap \text{set-mset } X \times \text{set-mset } Y) \ (s \cap \text{set-mset } X \times \text{set-mset } Y)$

**proof** –

**from** *assms* **obtain**  $X1 \ X2 \ Y1 \ Y2$  **where**  $*$ :  $X = X1 + X2 \ Y = Y1 + Y2$  **and**

$(X1, Y1) \in \text{multpw } ns \ (b \vee Y2 \neq \{\#\}) \ (\forall x. x \in\# \ X2 \longrightarrow (\exists y. y \in\# \ Y2 \wedge (x, y) \in s))$

**unfolding** *mult2-alt-def* **by** *blast*

**then have**  $X = X1 + X2 \wedge Y = Y1 + Y2 \wedge$

$(X1, Y1) \in \text{multpw } (ns \cap \text{set-mset } X \times \text{set-mset } Y) \wedge (b \vee Y2 \neq \{\#\}) \wedge$

$(\forall x. x \in\# \ X2 \longrightarrow (\exists y. y \in\# \ Y2 \wedge (x, y) \in s \cap \text{set-mset } X \times \text{set-mset } Y))$

**using** *multpw-local*[*of X1 Y1 ns*]

*mono-multpw*[*of ns \cap set-mset X1 \times set-mset Y1 ns \cap set-mset X \times set-mset Y*] *assms*

**unfolding**  $*$  *set-mset-union* **unfolding** *set-mset-def* **by** *blast*

**then show** *?thesis* **unfolding** *mult2-alt-def* **by** *blast*

**qed**

### 3.6 Local well-foundedness: restriction to downward closure of a set

**definition** *wf-below* ::  $'a \ rel \Rightarrow 'a \ set \Rightarrow bool$  **where**

$wf\text{-below } r \ A = wf \ (Restr \ r \ ((r^*)^{-1} \ \text{``} \ A))$

**lemma** *wf-below-UNIV*[*simp*]:

**shows**  $wf\text{-below } r \ UNIV \longleftrightarrow wf \ r$

**by** (*auto simp: wf-below-def rtrancl-converse[symmetric] Image-closed-trancl[OF subset-UNIV]*)

**lemma** *wf-below-mono1*:

**assumes**  $r \subseteq r'$  *wf-below*  $r' \ A$  **shows** *wf-below*  $r \ A$

**using** *assms unfolding wf-below-def*

**by** (*intro wf-subset[OF assms(2)[unfolded wf-below-def]] Int-mono Sigma-mono Image-mono*

*iffD2[OF converse-mono] rtrancl-mono subset-refl*)

**lemma** *wf-below-mono2*:

**assumes**  $A \subseteq A'$  *wf-below*  $r \ A'$  **shows** *wf-below*  $r \ A$

**using** *assms unfolding wf-below-def*

**by** (*intro wf-subset[OF assms(2)[unfolded wf-below-def]] blast*)

**lemma** *wf-below-pointwise*:

*wf-below*  $r \ A \longleftrightarrow (\forall a. a \in A \longrightarrow wf\text{-below } r \ \{a\})$  (**is**  $?L \longleftrightarrow ?R$ )

**proof**

**assume**  $?L$  **then show**  $?R$  **using** *wf-below-mono2*[*of*  $\{-\} \ A \ r$ ] **by** *blast*

**next**

**have**  $*$ :  $(r^*)^{-1} \ \text{``} \ A = \bigcup \{(r^*)^{-1} \ \text{``} \ \{a\} \mid a. a \in A\}$  **unfolding** *Image-def* **by** *blast*

**assume**  $?R$

**{** **fix**  $x \ X$  **assume**  $*$ :  $X \subseteq Restr \ r \ ((r^*)^{-1} \ \text{``} \ A)$   $\ \text{``} \ X \ x \in X$

**then obtain**  $a$  **where**  $**$ :  $a \in A \ (x, a) \in r^*$  **unfolding** *Image-def* **by** *blast*

**from**  $*$  **have**  $X \cap ((r^*)^{-1} \ \text{``} \ \{a\}) \subseteq (Restr \ r \ ((r^*)^{-1} \ \text{``} \ A) \ \text{``} \ X) \cap ((r^*)^{-1} \ \text{``} \ \{a\})$  **by** *auto*

**also have**  $\dots \subseteq Restr \ r \ ((r^*)^{-1} \ \text{``} \ \{a\}) \ \text{``} \ (X \cap ((r^*)^{-1} \ \text{``} \ \{a\}))$  **unfolding** *Image-def* **by** *fastforce*

**finally have**  $X \cap ((r^*)^{-1} \ \text{``} \ \{a\}) = \{ \}$  **using**  $\langle ?R \rangle ** (1)$  **unfolding** *wf-below-def*

**by** (*intro wfE-pf*[*of*  $Restr \ r \ ((r^*)^{-1} \ \text{``} \ \{a\})$ ]) (*auto simp: Image-def*)

**then have** *False* **using**  $*(2) **$  **unfolding** *Image-def* **by** *blast*

**}**

**then show**  $?L$  **unfolding** *wf-below-def* **by** (*intro wfI-pf*) *auto*

**qed**

**lemma** *SN-on-Image-rtrancl-conv*:

*SN-on*  $r \ A \longleftrightarrow SN\text{-on } r \ (r^* \ \text{``} \ A)$  (**is**  $?L \longleftrightarrow ?R$ )

**proof**

**assume**  $?L$  **then show**  $?R$  **by** (*auto simp: SN-on-Image-rtrancl*)

**next**

**assume**  $?R$  **then show**  $?L$  **by** (*auto simp: SN-on-def*)

**qed**

**lemma** *SN-on-iff-wf-below*:

*SN-on*  $r \ A \longleftrightarrow wf\text{-below } (r^{-1}) \ A$

**proof** –  
 { **fix**  $f$   
   **assume**  $f \ 0 \in r^* \text{ `` } A$  **and**  $** : (f \ i, f \ (Suc \ i)) \in r$  **for**  $i$   
   **then have**  $f \ i \in r^* \text{ `` } A$  **for**  $i$   
   **by** (*induct*  $i$ ) (*auto simp: Image-def,metis UnCI relcomp.relcompI rtrancl-unfold*)  
   **then have**  $(f \ i, f \ (Suc \ i)) \in Restr \ r \ (r^* \text{ `` } A)$  **for**  $i$  **using**  $**$  **by** *auto*  
 }  
**moreover then have**  $SN\text{-on } r \ (r^* \text{ `` } A) \longleftrightarrow SN\text{-on } (Restr \ r \ (r^* \text{ `` } A)) \ (r^* \text{ `` } A)$   
**unfolding**  $SN\text{-on-def}$  **by** *auto blast*  
**moreover have**  $(\bigwedge i. (f \ i, f \ (Suc \ i)) \in Restr \ r \ (r^* \text{ `` } A)) \implies f \ 0 \in r^* \text{ `` } A$  **for**  $f$   
**by** *auto*  
**then have**  $SN\text{-on } (Restr \ r \ (r^* \text{ `` } A)) \ (r^* \text{ `` } A) \longleftrightarrow SN\text{-on } (Restr \ r \ (r^* \text{ `` } A))$   
 $UNIV$   
**unfolding**  $SN\text{-on-def}$  **by** *auto*  
**ultimately show** *?thesis* **unfolding**  $SN\text{-on-Image-rtrancl-conv}$  [*of* -  $A$ ]  
**by** (*simp add: wf-below-def SN-iff-wf rtrancl-converse converse-Int converse-Times*)  
**qed**

**lemma** *restr-trancl-under*:  
**shows**  $Restr \ (r^+) \ ((r^*)^{-1} \text{ `` } A) = (Restr \ r \ ((r^*)^{-1} \text{ `` } A))^+$   
**proof** (*intro equalityI subrelI, elim IntE conjE[OF iffD1[OF mem-Sigma-iff]]*)  
**fix**  $a \ b$  **assume**  $*$ :  $(a, b) \in r^+ \ b \in (r^*)^{-1} \text{ `` } A$   
**then have**  $(a, b) \in (Restr \ r \ ((r^*)^{-1} \text{ `` } A))^+ \wedge a \in (r^*)^{-1} \text{ `` } A$   
**proof** (*induct rule: trancl-trans-induct[consumes 1]*)  
**case 1** **then show** *?case* **by** (*auto simp: Image-def intro: converse-rtrancl-into-rtrancl*)  
**next**  
**case 2** **then show** *?case* **by** (*auto simp del: Int-iff del: ImageE*)  
**qed**  
**then show**  $(a, b) \in (Restr \ r \ ((r^*)^{-1} \text{ `` } A))^+$  **by** *simp*  
**next**  
**fix**  $a \ b$  **assume**  $(a, b) \in (Restr \ r \ ((r^*)^{-1} \text{ `` } A))^+$   
**then show**  $(a, b) : Restr \ (r^+) \ ((r^*)^{-1} \text{ `` } A)$  **by** *induct auto*  
**qed**

**lemma** *wf-below-trancl*:  
**shows**  $wf\text{-below } (r^+) \ A \longleftrightarrow wf\text{-below } r \ A$   
**using** *restr-trancl-under*[*of*  $r \ A$ ] **by** (*simp add: wf-below-def wf-trancl-conv*)

**lemma** *wf-below-mult-local*:  
**assumes**  $wf\text{-below } r \ (set\text{-mset } X)$  **shows**  $wf\text{-below } (mult \ r) \ \{X\}$

**proof** –  
**have**  $foo : mult \ r \subseteq mult \ (r^+)$  **using** *mono-mult*[*of*  $r \ r^+$ ] **by** *force*  
**have**  $(Y, X) \in (mult \ (r^+))^* \implies set\text{-mset } Y \subseteq ((r^+)^*)^{-1} \text{ `` } set\text{-mset } X$  **for**  $Y$   
**proof** (*induct rule: converse-rtrancl-induct*)  
**case** (*step*  $Z \ Y$ )  
**obtain**  $I \ J \ K$  **where**  $*$ :  $Y = I + J \ Z = I + K \ (\forall k \in set\text{-mset } K. \exists j \in set\text{-mset } J. (k, j) \in r^+)$

```

    using mult-implies-one-step[OF - step(1)] by auto
  { fix k assume  $k \in \# K$ 
    then obtain j where  $j \in \# J$   $(k, j) \in r^+$  using  $*(\beta)$  by auto
    moreover then obtain x where  $x \in \# X$   $(j, x) \in r^*$  using step(β) by (auto simp: *)
    ultimately have  $\exists x. x \in \# X \wedge (k, x) \in r^*$  by auto
  }
  then show ?case using  $* \text{ step}(\beta)$  by (auto simp: Image-def) metis
qed auto
then have  $q: (Y, X) \in (\text{mult } (r^+))^* \implies y \in \# Y \implies y \in ((r^+)^*)^{-1}$  “set-mset X for Y y by force
have Restr  $(\text{mult } (r^+)) (((\text{mult } (r^+))^*)^{-1} \text{ “ } \{X\}) \subseteq \text{mult } (\text{Restr } (r^+)) (((r^+)^*)^{-1} \text{ “ } \text{set-mset } X)$ 
proof (intro subrelI)
  fix N M assume  $(N, M) \in \text{Restr } (\text{mult } (r^+)) (((\text{mult } (r^+))^*)^{-1} \text{ “ } \{X\})$ 
  then have  $** : (N, X) \in (\text{mult } (r^+))^* (M, X) \in (\text{mult } (r^+))^* (N, M) \in \text{mult } (r^+)$  by auto
  obtain I J K where  $* : M = I + J$   $N = I + K$   $J \neq \{\#\}$   $\forall k \in \text{set-mset } K. \exists j \in \text{set-mset } J. (k, j) \in r^+$ 
    using mult-implies-one-step[OF - <(N, M) ∈ mult (r^+)>] by auto
    then show  $(N, M) \in \text{mult } (\text{Restr } (r^+)) (((r^+)^*)^{-1} \text{ “ } \text{set-mset } X)$ 
    using  $q[\text{OF } **(\text{1})]$   $q[\text{OF } **(\text{2})]$  unfolding  $*$  by (auto intro: one-step-implies-mult)
qed
note bar = subset-trans[OF Int-mono[OF foo Sigma-mono] this]
have  $((\text{mult } r^*)^{-1} \text{ “ } \{X\}) \subseteq ((\text{mult } (r^+))^*)^{-1} \text{ “ } \{X\}$  using foo by (simp add: Image-mono rtrancl-mono)
then have Restr  $(\text{mult } r) (((\text{mult } r)^*)^{-1} \text{ “ } \{X\}) \subseteq \text{mult } (\text{Restr } (r^+)) (((r^+)^*)^{-1} \text{ “ } \text{set-mset } X)$ 
  by (intro bar) auto
then show ?thesis using wf-mult assms wf-subset
  unfolding wf-below-trancl[of r, symmetric] unfolding wf-below-def by blast
qed

```

**lemma** *qc-wf-below*:

```

  assumes  $s \ O \ ns \subseteq (s \cup ns)^* \ O \ s \ \text{wf-below } s \ A$ 
  shows wf-below  $(ns^* \ O \ s \ O \ ns^*) \ A$ 
  unfolding wf-below-def
proof (intro wfI-pf)
  let  $?A' = ((ns^* \ O \ s \ O \ ns^*)^*)^{-1} \text{ “ } A$ 
  fix X assume  $X: X \subseteq \text{Restr } (ns^* \ O \ s \ O \ ns^*) \ ?A' \text{ “ } X$ 
  let  $?X' = ((s \cup ns)^* \cap \text{UNIV} \times ((s^*)^{-1} \text{ “ } A)) \text{ “ } X$ 
  have  $* : s \ O \ (s \cup ns)^* \subseteq (s \cup ns)^* \ O \ s$ 
  proof –
    { fix x y z assume  $(y, z) \in (s \cup ns)^*$  and  $(x, y) \in s$ 
      then have  $(x, z) \in (s \cup ns)^* \ O \ s$ 
      proof (induct y z)
        case rtrancl-refl then show ?case by auto
      next
        case (rtrancl-into-rtrancl a b c)

```

**then have**  $(x, c) \in ((s \cup ns)^* O (s \cup ns)^*) O s$  **using** *assms* **by** *blast*  
**then show** *?case by simp*  
**qed** }  
**then show** *?thesis by auto*  
**qed**  
{ **fix**  $x$  **assume**  $x \in Restr (ns^* O s O ns^*)$  *?A' “ X*  
**then obtain**  $y z$  **where**  $** : y \in X z \in A (y, x) \in ns^* O s O ns^* (x, z) \in (ns^*$   
 $O s O ns^*)^*$  **by** *blast*  
**have**  $(ns^* O s O ns^*) O (ns^* O s O ns^*)^* \subseteq (s \cup ns)^*$  **by** *regexp*  
**then have**  $(y, z) \in (s \cup ns)^*$  **using** *?(3,4)* **by** *blast*  
**moreover have** *?X' = {}*  
**proof** (*intro wfE-pf[OF assms(2)[unfolded wf-below-def]] subsetI*)  
**fix**  $x$  **assume**  $x \in ?X'$   
**then have**  $x \in ((s \cup ns)^* \cap UNIV \times ((s^*)^{-1} “ A)) “ Restr (ns^* O s O ns^*)$   
 $?A' “ X$  **using** *X by auto*  
**then obtain**  $x0 y z$  **where**  $** : z \in X x0 \in A (z, y) \in ns^* O s O ns^* (y, x)$   
 $\in (s \cup ns)^* (x, x0) \in s^*$   
**unfolding** *Image-def by blast*  
**have**  $(ns^* O s O ns^*) O (s \cup ns)^* \subseteq ns^* O (s O (s \cup ns)^*)$  **by** *regexp*  
**with** *?(3,4)* **have**  $(z, x) \in ns^* O (s O (s \cup ns)^*)$  **by** *blast*  
**moreover have**  $ns^* O ((s \cup ns)^* O s) \subseteq (s \cup ns)^* O s$  **by** *regexp*  
**ultimately have**  $(z, x) \in (s \cup ns)^* O s$  **using** *\** **by** *blast*  
**then obtain**  $x'$  **where**  $z \in X (z, x') \in (s \cup ns)^* (x', x) \in s (x', x0) \in s^*$   
 $(x, x0) \in s^* x0 \in A$   
**using** *?(1,2,5) converse-rtrancl-into-rtrancl[OF - \*(5)] by blast*  
**then show**  $x \in Restr s ((s^*)^{-1} “ A) “ ?X'$   
**unfolding** *Image-def by blast*  
**qed**  
**ultimately have** *False* **using** *?(1,2)* **unfolding** *Image-def by blast*  
}
**then show**  $X = \{\}$  **using** *X by blast*  
**qed**

**lemma** *wf-below-mult2-s-local*:

**assumes** *wf-below s (set-mset X) s O ns*  $\subseteq s refl ns trans ns$   
**shows** *wf-below (mult2-s ns s) {X}*  
**using** *wf-below-mult-local[of s X] assms multpw-mult-commute[of s ns]*  
*wf-below-mono1[of multpw ns O mult s (multpw ns)^\* O mult s O (multpw ns)^\**  
 $\{X\}$   
*qc-wf-below[of mult s multpw ns {X}]*  
**unfolding** *mult2-s-def by blast*

### 3.7 Trivial cases

**lemma** *mult2-alt-emptyL*:

$(\{\#\}, Y) \in mult2-alt b ns s \longleftrightarrow b \vee Y \neq \{\#\}$   
**unfolding** *mult2-alt-def by auto*

**lemma** *mult2-alt-emptyR*:

$(X, \{\#\}) \in \text{mult2-alt } b \text{ ns } s \iff b \wedge X = \{\#\}$   
**unfolding** *mult2-alt-def* **by** (*auto intro: multiset-eqI*)

**lemma** *mult2-alt-s-single*:

$(a, b) \in s \implies (\{\#a\}, \{\#b\}) \in \text{mult2-alt-s ns } s$   
**using** *mult2-altI[of - {\#} - - {\#} - ns False s]* **by** *auto*

**lemma** *multpw-implies-mult2-alt-ns*:

**assumes**  $(X, Y) \in \text{multpw ns}$   
**shows**  $(X, Y) \in \text{mult2-alt-ns ns } s$   
**using** *assms* **by** (*intro mult2-alt-nsI[of X X {\#} Y Y {\#}]*) *auto*

**lemma** *mult2-alt-ns-conv*:

$\text{mult2-alt-ns ns } s = \text{mult2-alt-s ns } s \cup \text{multpw ns}$  (**is**  $?l = ?r$ )

**proof** (*intro equalityI subrelI*)

**fix**  $X Y$  **assume**  $(X, Y) \in ?l$

**thm** *mult2-alt-nsE*

**then obtain**  $X1 X2 Y1 Y2$  **where**  $X = X1 + X2$   $Y = Y1 + Y2$   $(X1, Y1) \in \text{multpw ns}$

$\forall x. x \in \# X2 \longrightarrow (\exists y. y \in \# Y2 \wedge (x, y) \in s)$  **by** (*auto elim: mult2-alt-nsE*)

**then show**  $(X, Y) \in ?r$  **using** *count-inject[of X2 {\#}]*

**by** (*cases Y2 = {\#}*) (*auto intro: mult2-alt-sI elim: mult2-alt-nsE mult2-alt-sE*)

**next**

**fix**  $X Y$  **assume**  $(X, Y) \in ?r$  **then show**  $(X, Y) \in ?l$

**by** (*auto intro: mult2-alt-nsI multpw-implies-mult2-alt-ns elim: mult2-alt-sE*)

**qed**

**lemma** *mult2-alt-s-implies-mult2-alt-ns*:

**assumes**  $(X, Y) \in \text{mult2-alt-s ns } s$   
**shows**  $(X, Y) \in \text{mult2-alt-ns ns } s$   
**using** *assms* **by** (*auto intro: mult2-alt-nsI elim: mult2-alt-sE*)

**lemma** *mult2-alt-add*:

**assumes**  $(X1, Y1) \in \text{mult2-alt } b1 \text{ ns } s$  **and**  $(X2, Y2) \in \text{mult2-alt } b2 \text{ ns } s$

**shows**  $(X1 + X2, Y1 + Y2) \in \text{mult2-alt } (b1 \wedge b2) \text{ ns } s$

**proof** –

**from** *assms* **obtain**  $X11 X12 Y11 Y12 X21 X22 Y21 Y22$  **where**

$X1 = X11 + X12$   $Y1 = Y11 + Y12$

$(X11, Y11) \in \text{multpw ns}$   $(b1 \vee Y12 \neq \{\#\})$   $(\forall x. x \in \# X12 \longrightarrow (\exists y. y \in \# Y12 \wedge (x, y) \in s))$

$X2 = X21 + X22$   $Y2 = Y21 + Y22$

$(X21, Y21) \in \text{multpw ns}$   $(b2 \vee Y22 \neq \{\#\})$   $(\forall x. x \in \# X22 \longrightarrow (\exists y. y \in \# Y22 \wedge (x, y) \in s))$

**unfolding** *mult2-alt-def* **by** (*blast 9*)

**then show** *?thesis*

**by** (*intro mult2-altI[of - X11 + X21 X12 + X22 - Y11 + Y21 Y12 + Y22]*)

(*auto intro: multpw-add simp: ac-simps*)

**qed**

**lemmas** *mult2-alt-s-s-add* = *mult2-alt-add*[of - - *False* - - - *False*, *unfolded simp-thms*]

**lemmas** *mult2-alt-ns-s-add* = *mult2-alt-add*[of - - *True* - - - *False*, *unfolded simp-thms*]

**lemmas** *mult2-alt-s-ns-add* = *mult2-alt-add*[of - - *False* - - - *True*, *unfolded simp-thms*]

**lemmas** *mult2-alt-ns-ns-add* = *mult2-alt-add*[of - - *True* - - - *True*, *unfolded simp-thms*]

**lemma** *multpw-map*:

**assumes**  $\bigwedge x y. x \in\# X \implies y \in\# Y \implies (x, y) \in ns \implies (f x, g y) \in ns'$

**and**  $(X, Y) \in \text{multpw } ns$

**shows**  $(\text{image-mset } f X, \text{image-mset } g Y) \in \text{multpw } ns'$

**using** *assms(2,1)* **by** (*induct X Y rule: multpw.induct*) (*auto intro: multpw.intros*)

**lemma** *mult2-alt-map*:

**assumes**  $\bigwedge x y. x \in\# X \implies y \in\# Y \implies (x, y) \in ns \implies (f x, g y) \in ns'$

**and**  $\bigwedge x y. x \in\# X \implies y \in\# Y \implies (x, y) \in s \implies (f x, g y) \in s'$

**and**  $(X, Y) \in \text{mult2-alt } b \ ns \ s$

**shows**  $(\text{image-mset } f X, \text{image-mset } g Y) \in \text{mult2-alt } b \ ns' \ s'$

**proof** -

**from** *assms(3)* **obtain** *X1 X2 Y1 Y2* **where**  $X = X1 + X2 \ Y = Y1 + Y2 \ (X1, Y1) \in \text{multpw } ns$

$b \vee Y2 \neq \{\#\} \ \forall x. x \in\# X2 \longrightarrow (\exists y. y \in\# Y2 \wedge (x, y) \in s)$  **by** (*auto elim: mult2-altE*)

**moreover from** *this(1,2,5)* **have**  $\forall x. x \in\# \text{image-mset } f X2 \longrightarrow (\exists y. y \in\# \text{image-mset } g Y2 \wedge (x, y) \in s')$

**using** *assms(2)* **by** (*simp add: in-image-mset image-iff*) *blast*

**ultimately show** *?thesis* **using** *assms multpw-map*[of *X1 Y1 ns f g*]

**by** (*intro mult2-altI*[of - *image-mset f X1 image-mset f X2 - image-mset g Y1 image-mset g Y2*]) *auto*

**qed**

Local transitivity of *mult2-alt*

**lemma** *trans-mult2-alt-local*:

**assumes** *ss*:  $\bigwedge x y z. x \in\# X \implies y \in\# Y \implies z \in\# Z \implies (x, y) \in s \implies (y, z) \in s \implies (x, z) \in s$

**and** *ns*:  $\bigwedge x y z. x \in\# X \implies y \in\# Y \implies z \in\# Z \implies (x, y) \in ns \implies (y, z) \in ns \implies (x, z) \in ns$

**and** *sn*:  $\bigwedge x y z. x \in\# X \implies y \in\# Y \implies z \in\# Z \implies (x, y) \in s \implies (y, z) \in ns \implies (x, z) \in ns$

**and** *nn*:  $\bigwedge x y z. x \in\# X \implies y \in\# Y \implies z \in\# Z \implies (x, y) \in ns \implies (y, z) \in ns \implies (x, z) \in ns$

**and** *xyz*:  $(X, Y) \in \text{mult2-alt } b1 \ ns \ s \ (Y, Z) \in \text{mult2-alt } b2 \ ns \ s$

**shows**  $(X, Z) \in \text{mult2-alt } (b1 \wedge b2) \ ns \ s$

**proof** -

**let** *?a1* = *Enum.finite-3.a1* **and** *?a2* = *Enum.finite-3.a2* **and** *?a3* = *Enum.finite-3.a3*

**let** *?t* =  $\{(?a1, ?a2), (?a1, ?a3), (?a2, ?a3)\}$

```

let ?A = {(?a1, x) | x. x ∈# X} ∪ {(?a2, y) | y. y ∈# Y} ∪ {(?a3, z) | z. z ∈# Z}
define s' where s' = Restr {(a, x), (b, y) | a x b y. (a, b) ∈ ?t ∧ (x, y) ∈ s}
?A
define ns' where ns' = (Restr {(a, x), (b, y) | a x b y. (a, b) ∈ ?t ∧ (x, y) ∈ ns} ?A)=
have *: refl ns' trans ns' trans s' s' O ns' ⊆ s' ns' O s' ⊆ s'
by (force simp: trans-def ss ns sn nn s'-def ns'-def)+
have ({#(?a1, x). x ∈# X#}, {#(?a2, y). y ∈# Y#}) ∈ mult2-alt b1 ns' s'
by (auto intro: mult2-alt-map[OF - - xyz(1)] simp: s'-def ns'-def)
moreover have ({#(?a2, y). y ∈# Y#}, {#(?a3, z). z ∈# Z#}) ∈ mult2-alt
b2 ns' s'
by (auto intro: mult2-alt-map[OF - - xyz(2)] simp: s'-def ns'-def)
ultimately have ({#(?a1, x). x ∈# X#}, {#(?a3, z). z ∈# Z#}) ∈ mult2-alt
(b1 ∧ b2) ns' s'
using mult2-s-eq-mult2-s-alt[OF *(5,1,3)] mult2-ns-eq-mult2-ns-alt[OF *(5,1,3)]
trans-mult2-s[OF *(4,1,2)] trans-mult2-ns[OF *(4,1,2)] compat-mult2[OF
*(4,1,2)]
by (cases b1; cases b2) (auto simp: trans-O-iff)
from mult2-alt-map[OF - - this, of snd snd ns s]
show ?thesis by (auto simp: s'-def ns'-def image-mset.compositionality comp-def
in-image-mset image-iff)
qed

```

**lemmas** trans-mult2-alt-s-s-local = trans-mult2-alt-local[of - - - - False False, unfolded simp-thms]

**lemmas** trans-mult2-alt-ns-s-local = trans-mult2-alt-local[of - - - - True False, unfolded simp-thms]

**lemmas** trans-mult2-alt-s-ns-local = trans-mult2-alt-local[of - - - - False True, unfolded simp-thms]

**lemmas** trans-mult2-alt-ns-ns-local = trans-mult2-alt-local[of - - - - True True, unfolded simp-thms]

**end**

### 3.8 Executable version

**theory** Multiset-Extension-Pair-Impl

**imports**

Multiset-Extension-Pair

**begin**

**lemma** subset-mult2-alt:

**assumes**  $X \subseteq\# Y$   $(Y, Z) \in \text{mult2-alt } b \text{ ns } s \text{ b} \implies b'$

**shows**  $(X, Z) \in \text{mult2-alt } b' \text{ ns } s$

**proof** –

**from** *assms*(2) **obtain**  $Y1 \ Y2 \ Z1 \ Z2$  **where** \*:  $Y = Y1 + Y2 \ Z = Z1 + Z2$

$(Y1, Z1) \in \text{multpw } ns \ b \vee Z2 \neq \{\#\} \forall y. y \in\# Y2 \longrightarrow (\exists z. z \in\# Z2 \wedge (y, z) \in s)$

**unfolding** *mult2-alt-def* **by** *blast*  
**define**  $Y11\ Y12\ X2$  **where**  $Y11 = Y1 \cap\# X$  **and**  $Y12 = Y1 - X$  **and**  $X2 = X - Y11$   
**have**  $**$ :  $X = Y11 + X2$   $X2 \subseteq\# Y2$   $Y1 = Y11 + Y12$  **using**  $*(1)$   
**by** (*auto simp: Y11-def Y12-def X2-def multiset-eq-iff subseteq-mset-def*)  
(*metis add.commute assms(1) le-diff-conv subseteq-mset-def*)  
**obtain**  $Z11\ Z12$  **where**  $***$ :  $Z = Z11 + (Z12 + Z2)$   $Z1 = Z11 + Z12$  ( $Y11, Z11$ )  $\in$  *multpw ns*  
**using**  $*(2,3)\ ***(3)$  **by** (*auto elim: multpw-splitR simp: ac-simps*)  
**moreover have**  $\forall y. y \in\# X2 \longrightarrow (\exists z. z \in\# Z12 + Z2 \wedge (y, z) \in s) \vee Z12 + Z2 \neq \{\#\}$   
**using**  $*(4,5)\ ***(2)$  **by** (*auto dest!: mset-subset-eqD*)  
**ultimately show** *?thesis* **using**  $*(2)\ ***(1)\ assms(3)$  **unfolding** *mult2-alt-def*  
**by** *blast*  
**qed**

Case distinction for recursion on left argument

**lemma** *mem-multiset-diff*:  $x \in\# A \implies x \neq y \implies x \in\# (A - \{\#y\#\})$   
**by** (*metis add-mset-remove-trivial-If diff-single-trivial insert-noteq-member*)

**lemma** *mult2-alt-addL*:  $(\text{add-mset } x\ X, Y) \in \text{mult2-alt } b\ ns\ s \longleftrightarrow$   
 $(\exists y. y \in\# Y \wedge (x, y) \in s \wedge (\{\#\ x \in\# X. (x, y) \notin s \#\}, Y - \{\#y\#\}) \in$   
 $\text{mult2-alt-ns } ns\ s) \vee$   
 $(\exists y. y \in\# Y \wedge (x, y) \in ns \wedge (x, y) \notin s \wedge (X, Y - \{\#y\#\}) \in \text{mult2-alt } b\ ns\ s)$   
**(is** *?L*  $\longleftrightarrow$  *?R1*  $\vee$  *?R2*)  
**proof** (*intro iffI; (elim disjE)?*)

**assume** *?L* **then obtain**  $X1\ X2\ Y1\ Y2$  **where**  $*$ :  $\text{add-mset } x\ X = X1 + X2$   $Y = Y1 + Y2$   
 $(X1, Y1) \in \text{multpw } ns\ b \vee Y2 \neq \{\#\} \forall x. x \in\# X2 \longrightarrow (\exists y. y \in\# Y2 \wedge (x, y) \in s)$

**unfolding** *mult2-alt-def* **by** *blast*

**from** *union-single-eq-member[OF this(1)] multi-member-split*

**consider**  $X1'$  **where**  $X1 = \text{add-mset } x\ X1'$   $x \in\# X1 \mid X2'$  **where**  $X2 = \text{add-mset } x\ X2'$   $x \in\# X2$

**unfolding** *set-mset-union Un-iff* **by** *metis*

**then show** *?R1*  $\vee$  *?R2*

**proof** *cases*

**case** *1* **then obtain**  $y\ Y1'$  **where**  $**$ :  $y \in\# Y1$   $Y1 = \text{add-mset } y\ Y1'$  ( $X1', Y1'$ )  $\in$  *multpw ns*  $(x, y) \in ns$

**using**  $*$  **by** (*auto elim: multpw-split1R*)

**show** *?thesis*

**proof** (*cases*  $(x, y) \in s$ )

**case** *False* **then show** *?thesis* **using** *mult2-altI[OF refl refl \*\*\*(3) \*(4,5)] \**

**by** (*auto simp: 1 \*\* intro: exI[of - y]*)

**next**

**case** *True*

**define**  $X2'$  **where**  $X2' = \{\#\ x \in\# X2. (x, y) \notin s \#\}$

**have**  $x3$ :  $\forall x. x \in\# X2' \longrightarrow (\exists z. z \in\# Y2 \wedge (x, z) \in s)$  **using**  $*(5)\ ***(1,2)$

**by** (*auto simp: X2'-def*)

```

    have x4: {# x ∈# X. (x, y) ∉ s#} ⊆# X1' + X2' using *(1) 1
    by (auto simp: X2'-def multiset-eq-iff intro!: mset-subset-eqI split: if-splits
elim!: in-countE) (metis le-refl)
    show ?thesis using mult2-alt-nsI[OF refl refl ***(3) x3, THEN subset-mult2-alt[OF
x4]]
      ***(2) *(2) True by (auto intro: exI[of - y])
  qed
next
  case 2 then obtain y where **: y ∈# Y2 (x, y) ∈ s using * by blast
  define X2' where X2' = {# x ∈# X2. (x, y) ∉ s#}
  have x3: ∀ x. x ∈# X2' → (∃ z. z ∈# Y2 - {#y#} ∧ (x, z) ∈ s)
    using *(5) ***(1,2) by (auto simp: X2'-def) (metis mem-multiset-diff)
  have x4: {# x ∈# X. (x, y) ∉ s#} ⊆# X1 + X2'
    using *(1) ***(2) by (auto simp: X2'-def multiset-eq-iff intro!: mset-subset-eqI
split: if-splits)
  show ?thesis
    using mult2-alt-nsI[OF refl refl *(3) x3, THEN subset-mult2-alt[OF x4], of
True] ***(1,2) *(2)
    by (auto simp: diff-union-single-conv[symmetric])
  qed
next
  assume ?R1
  then obtain y where *: y ∈# Y (x, y) ∈ s ({# x ∈# X. (x, y) ∉ s#}, Y -
{#y#}) ∈ mult2-alt-ns ns s
    by blast
  then have **: ({# x ∈# X. (x, y) ∈ s#} + {#x#}, {#y#}) ∈ mult2-alt b ns
s
    {# x ∈# X. (x, y) ∉ s#} + {# x ∈# X. (x, y) ∈ s#} = X
    by (auto intro: mult2-altI[of - {#} - - {#}] multiset-eqI split: if-splits)
  show ?L using mult2-alt-add[OF ***(3) ***(1)] *** by (auto simp: union-assoc[symmetric])
next
  assume ?R2
  then obtain y where *: y ∈# Y (x, y) ∈ ns (X, Y - {#y#}) ∈ mult2-alt b
ns s by blast
  then show ?L using mult2-alt-add[OF *(3) multpw-implies-mult2-alt-ns, of
{#x#} {#y#}]
    by (auto intro: multpw-single)
  qed

```

Auxiliary version with an extra *bool* argument for distinguishing between the non-strict and the strict orders

```

context fixes nss :: 'a ⇒ 'a ⇒ bool ⇒ bool
begin

```

```

fun mult2-impl0 :: 'a list ⇒ 'a list ⇒ bool ⇒ bool
and mult2-ex-dom0 :: 'a ⇒ 'a list ⇒ 'a list ⇒ 'a list ⇒ bool ⇒ bool
where
  mult2-impl0 [] [] b ←→ b
| mult2-impl0 xs [] b ←→ False

```

```

| mult2-impl0 [] ys b <=> True
| mult2-impl0 (x # xs) ys b <=> mult2-ex-dom0 x xs ys [] b

| mult2-ex-dom0 x xs [] ys' b <=> False
| mult2-ex-dom0 x xs (y # ys) ys' b <=>
  nss x y False & mult2-impl0 (filter (λx. ¬ nss x y False) xs) (ys @ ys') True ∨
  nss x y True & ¬ nss x y False & mult2-impl0 xs (ys @ ys') b ∨
  mult2-ex-dom0 x xs ys (y # ys') b

```

**end**

**lemma** *mult2-impl0-sound*:

```

fixes nss
defines ns ≡ {(x, y). nss x y True} and s ≡ {(x, y). nss x y False}
shows mult2-impl0 nss xs ys b <=> (mset xs, mset ys) ∈ mult2-alt b ns s
  mult2-ex-dom0 nss x xs ys ys' b <=>
    (∃ y. y ∈ # mset ys & (x, y) ∈ s & (mset (filter (λx. (x, y) ∉ s) xs), mset (ys
@ ys') - {#y#}) ∈ mult2-alt True ns s) ∨
    (∃ y. y ∈ # mset ys & (x, y) ∈ ns & (x, y) ∉ s & (mset xs, mset (ys @ ys') -
{#y#}) ∈ mult2-alt b ns s)
proof (induct xs ys b and x xs ys ys' b taking: nss rule: mult2-impl0-mult2-ex-dom0.induct)
  case (4 x xs y ys b) show ?case unfolding mult2-impl0.simps 4
    using mult2-alt-addL[of x mset xs mset (y # ys) b ns s] by (simp add: mset-filter)
next
  case (6 x xs y ys ys' b) show ?case unfolding mult2-ex-dom0.simps 6
    using subset-mult2-alt[of mset [x←xs . (x, y) ∉ s] mset xs mset (ys @ ys') b ns
s True]
    apply (intro iffI; elim disjE conjE exE; simp add: mset-filter ns-def s-def; (elim
disjE)?)
    subgoal by (intro disjI1 exI[of - y]) auto
    subgoal by (intro disjI2 exI[of - y]) auto
    subgoal for y' by (intro disjI1 exI[of - y']) auto
    subgoal for y' by (intro disjI2 exI[of - y']) auto
    subgoal for y' by simp
    subgoal for y' by (rule disjI2, rule disjI2, rule disjI1, rule exI[of - y']) simp
    subgoal for y' by simp
    subgoal for y' by (rule disjI2, rule disjI2, rule disjI2, rule exI[of - y']) simp
  done
qed (auto simp: mult2-alt-emptyL mult2-alt-emptyR)

```

Now, instead of functions of type  $bool \Rightarrow bool$ , use pairs of type  $bool \times bool$

**definition** [*simp*]:  $or2\ a\ b = (fst\ a \vee\ fst\ b,\ snd\ a \vee\ snd\ b)$

**context** **fixes**  $sns :: 'a \Rightarrow 'a \Rightarrow bool \times bool$   
**begin**

**fun** *mult2-impl* ::  $'a\ list \Rightarrow 'a\ list \Rightarrow bool \times bool$   
**and** *mult2-ex-dom* ::  $'a \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool \times bool$

```

where
  mult2-impl [] [] = (False, True)
| mult2-impl xs [] = (False, False)
| mult2-impl [] ys = (True, True)
| mult2-impl (x # xs) ys = mult2-ex-dom x xs ys []

| mult2-ex-dom x xs [] ys' = (False, False)
| mult2-ex-dom x xs (y # ys) ys' =
  (case sns x y of
    (True, -) => if snd (mult2-impl (filter (λx. ¬ fst (sns x y)) xs) (ys @ ys')) then
      (True, True)
    else mult2-ex-dom x xs ys (y # ys'))
  | (False, True) => or2 (mult2-impl xs (ys @ ys')) (mult2-ex-dom x xs ys (y #
ys'))
  | - => mult2-ex-dom x xs ys (y # ys'))
end

```

**lemma** *mult2-impl-sound0*:

```

defines pair ≡ λf. (f False, f True) and fun ≡ λp b. if b then snd p else fst p
shows mult2-impl sns xs ys = pair (mult2-impl0 (λx y. fun (sns x y)) xs ys) (is
?P)

```

```

  mult2-ex-dom sns x xs ys ys' = pair (mult2-ex-dom0 (λx y. fun (sns x y)) x xs
ys ys') (is ?Q)

```

**proof** –

```

show ?P ?Q

```

```

proof (induct xs ys and x xs ys ys' taking: sns rule: mult2-impl-mult2-ex-dom.induct)

```

```

  case (6 x xs y ys ys')

```

```

  show ?case unfolding mult2-ex-dom.simps mult2-ex-dom0.simps

```

```

  by (fastforce simp: pair-def fun-def 6 if-bool-eq-conj split: prod.splits bool.splits)

```

```

qed (auto simp: pair-def fun-def if-bool-eq-conj)

```

**qed**

```

lemmas mult2-impl-sound = mult2-impl-sound0(1)[unfolded mult2-impl0-sound if-True
if-False]

```

**end**

## 4 Multiset extension of order pairs in the other direction

Many term orders are formulated in the other direction, i.e., they use strong normalization of  $>$  instead of well-foundedness of  $<$ . Here, we flip the direction of the multiset extension of two orders, connect it to existing interfaces, and prove some further properties of the multiset extension.

**theory** *Multiset-Extension2*

**imports**

*List-Order*

*Multiset-Extension-Pair*

begin

#### 4.1 List based characterization of *multpw*

**lemma** *multpw-listI*:

**assumes**  $\text{length } xs = \text{length } ys \ X = \text{mset } xs \ Y = \text{mset } ys$

$\forall i. i < \text{length } ys \longrightarrow (xs ! i, ys ! i) \in ns$

**shows**  $(X, Y) \in \text{multpw } ns$

**using** *assms*

**proof** (*induct xs arbitrary: ys X Y*)

**case** (*Nil ys*) **then show** *?case* **by** (*cases ys*) (*auto intro: multpw.intros*)

**next**

**case** *Cons1: (Cons x xs ys' X Y)* **then show** *?case*

**proof** (*cases ys'*)

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

**then have**  $\forall i. i < \text{length } ys \longrightarrow (xs ! i, ys ! i) \in ns$  **using** *Cons1(5)* **by** *fastforce*

**then show** *?thesis* **using** *Cons1(2,5)* **by** (*auto intro!: multpw.intros simp: Cons(1) Cons1*)

**qed** *auto*

**qed**

**lemma** *multpw-listE*:

**assumes**  $(X, Y) \in \text{multpw } ns$

**obtains**  $xs \ ys$  **where**  $\text{length } xs = \text{length } ys \ X = \text{mset } xs \ Y = \text{mset } ys$

$\forall i. i < \text{length } ys \longrightarrow (xs ! i, ys ! i) \in ns$

**using** *assms*

**proof** (*induct X Y arbitrary: thesis rule: multpw.induct*)

**case** (*add x y X Y*)

**then obtain**  $xs \ ys$  **where**  $\text{length } xs = \text{length } ys \ X = \text{mset } xs$

$Y = \text{mset } ys \ (\forall i. i < \text{length } ys \longrightarrow (xs ! i, ys ! i) \in ns)$  **by** *blast*

**then show** *?case* **using** *add(1)* **by** (*intro add(4)[of x # xs y # ys]*) (*auto, case-tac i, auto*)

**qed** *auto*

#### 4.2 Definition of the multiset extension of $>$ -orders

We define here the non-strict extension of the order pair  $(\geq, >)$  – usually written as  $(ns, s)$  in the sources – by just flipping the directions twice.

**definition** *ns-mul-ext*  $:: 'a \text{ rel} \Rightarrow 'a \text{ rel} \Rightarrow 'a \text{ multiset rel}$

**where**  $ns\text{-mul-ext } ns \ s \equiv (\text{mult2-alt-ns } (ns^{-1}) \ (s^{-1}))^{-1}$

**lemma** *ns-mul-extI*:

**assumes**  $A = A1 + A2$  **and**  $B = B1 + B2$

**and**  $(A1, B1) \in \text{multpw } ns$

**and**  $\bigwedge b. b \in \# B2 \Longrightarrow \exists a. a \in \# A2 \wedge (a, b) \in s$

**shows**  $(A, B) \in ns\text{-mul-ext } ns \ s$

**using** *assms* **by** (*auto simp: ns-mul-ext-def multpw-converse intro!: mult2-alt-nsI*)

**lemma** *ns-mul-extE*:

**assumes**  $(A, B) \in ns\text{-mul-ext } ns \ s$   
**obtains**  $A1 \ A2 \ B1 \ B2$  **where**  $A = A1 + A2$  **and**  $B = B1 + B2$   
**and**  $(A1, B1) \in multpw \ ns$   
**and**  $\bigwedge b. b \in \# B2 \implies \exists a. a \in \# A2 \wedge (a, b) \in s$   
**using** *assms* **by**  $(auto \ simp: ns\text{-mul-ext-def } multpw\text{-converse } elim!: mult2\text{-alt-nsE})$

**lemmas** *ns-mul-extI-old* = *ns-mul-extI*[*OF* - - *multpw-listI*[*OF* - *refl refl*], *rule-format*]

Same for the "greater than" order on multisets.

**definition** *s-mul-ext* :: '*a rel*  $\Rightarrow$  '*a rel*  $\Rightarrow$  '*a multiset rel*  
**where** *s-mul-ext ns s*  $\equiv (mult2\text{-alt-s } (ns^{-1}) (s^{-1}))^{-1}$

**lemma** *s-mul-extI*:

**assumes**  $A = A1 + A2$  **and**  $B = B1 + B2$   
**and**  $(A1, B1) \in multpw \ ns$   
**and**  $A2 \neq \{\#\}$  **and**  $\bigwedge b. b \in \# B2 \implies \exists a. a \in \# A2 \wedge (a, b) \in s$   
**shows**  $(A, B) \in s\text{-mul-ext } ns \ s$   
**using** *assms* **by**  $(auto \ simp: s\text{-mul-ext-def } multpw\text{-converse } intro!: mult2\text{-alt-sI})$

**lemma** *s-mul-extE*:

**assumes**  $(A, B) \in s\text{-mul-ext } ns \ s$   
**obtains**  $A1 \ A2 \ B1 \ B2$  **where**  $A = A1 + A2$  **and**  $B = B1 + B2$   
**and**  $(A1, B1) \in multpw \ ns$   
**and**  $A2 \neq \{\#\}$  **and**  $\bigwedge b. b \in \# B2 \implies \exists a. a \in \# A2 \wedge (a, b) \in s$   
**using** *assms* **by**  $(auto \ simp: s\text{-mul-ext-def } multpw\text{-converse } elim!: mult2\text{-alt-sE})$

**lemmas** *s-mul-extI-old* = *s-mul-extI*[*OF* - - *multpw-listI*[*OF* - *refl refl*], *rule-format*]

### 4.3 Basic properties

**lemma** *s-mul-ext-mono*:

**assumes**  $ns \subseteq ns' \ s \subseteq s'$  **shows**  $s\text{-mul-ext } ns \ s \subseteq s\text{-mul-ext } ns' \ s'$   
**unfolding** *s-mul-ext-def* **using** *assms mono-mult2-alt*[*of ns<sup>-1</sup> ns'<sup>-1</sup> s<sup>-1</sup> s'<sup>-1</sup>*] **by**  
*simp*

**lemma** *ns-mul-ext-mono*:

**assumes**  $ns \subseteq ns' \ s \subseteq s'$  **shows**  $ns\text{-mul-ext } ns \ s \subseteq ns\text{-mul-ext } ns' \ s'$   
**unfolding** *ns-mul-ext-def* **using** *assms mono-mult2-alt*[*of ns<sup>-1</sup> ns'<sup>-1</sup> s<sup>-1</sup> s'<sup>-1</sup>*]  
**by** *simp*

**lemma** *s-mul-ext-local-mono*:

**assumes** *sub*:  $(set\text{-mset } xs \times set\text{-mset } ys) \cap ns \subseteq ns' (set\text{-mset } xs \times set\text{-mset } ys)$   
 $\cap s \subseteq s'$   
**and** *rel*:  $(xs, ys) \in s\text{-mul-ext } ns \ s$   
**shows**  $(xs, ys) \in s\text{-mul-ext } ns' \ s'$   
**using** *rel s-mul-ext-mono*[*OF sub*] *mult2-alt-local*[*of ys xs False ns<sup>-1</sup> s<sup>-1</sup>*]  
**by**  $(auto \ simp: s\text{-mul-ext-def } converse\text{-Int } ac\text{-simps } converse\text{-Times})$

**lemma** *ns-mul-ext-local-mono*:

**assumes** *sub*:  $(\text{set-mset } xs \times \text{set-mset } ys) \cap ns \subseteq ns' (\text{set-mset } xs \times \text{set-mset } ys)$   
 $\cap s \subseteq s'$   
**and** *rel*:  $(xs, ys) \in ns\text{-mul-ext } ns \ s$   
**shows**  $(xs, ys) \in ns\text{-mul-ext } ns' \ s'$   
**using** *rel ns-mul-ext-mono*[*OF sub*] *mult2-alt-local*[*of ys xs True ns<sup>-1</sup> s<sup>-1</sup>*]  
**by** (*auto simp: ns-mul-ext-def converse-Int ac-simps converse-Times*)

**lemma** *s-mul-ext-ord-s* [*mono*]:  
**assumes**  $\bigwedge s \ t. \text{ord } s \ t \longrightarrow \text{ord}' \ s \ t$   
**shows**  $(s, t) \in s\text{-mul-ext } ns \ \{(s, t). \text{ord } s \ t\} \longrightarrow (s, t) \in s\text{-mul-ext } ns \ \{(s, t). \text{ord}' \ s \ t\}$   
**using** *assms s-mul-ext-mono* **by** (*metis (mono-tags) case-prod-conv mem-Collect-eq old.prod.exhaust subset-eq*)

**lemma** *ns-mul-ext-ord-s* [*mono*]:  
**assumes**  $\bigwedge s \ t. \text{ord } s \ t \longrightarrow \text{ord}' \ s \ t$   
**shows**  $(s, t) \in ns\text{-mul-ext } ns \ \{(s, t). \text{ord } s \ t\} \longrightarrow (s, t) \in ns\text{-mul-ext } ns \ \{(s, t). \text{ord}' \ s \ t\}$   
**using** *assms ns-mul-ext-mono* **by** (*metis (mono-tags) case-prod-conv mem-Collect-eq old.prod.exhaust subset-eq*)

The empty multiset is the minimal element for these orders

**lemma** *ns-mul-ext-bottom*:  $(A, \{\#\}) \in ns\text{-mul-ext } ns \ s$   
**by** (*auto intro!: ns-mul-extI*)

**lemma** *ns-mul-ext-bottom-uniqueness*:  
**assumes**  $(\{\#\}, A) \in ns\text{-mul-ext } ns \ s$   
**shows**  $A = \{\#\}$   
**using** *assms* **by** (*auto simp: ns-mul-ext-def mult2-alt-ns-def*)

**lemma** *ns-mul-ext-bottom2*:  
**assumes**  $(A, B) \in ns\text{-mul-ext } ns \ s$   
**and**  $B \neq \{\#\}$   
**shows**  $A \neq \{\#\}$   
**using** *assms* **by** (*auto simp: ns-mul-ext-def mult2-alt-ns-def*)

**lemma** *s-mul-ext-bottom*:  
**assumes**  $A \neq \{\#\}$   
**shows**  $(A, \{\#\}) \in s\text{-mul-ext } ns \ s$   
**using** *assms* **by** (*auto simp: s-mul-ext-def mult2-alt-s-def*)

**lemma** *s-mul-ext-bottom-strict*:  
 $(\{\#\}, A) \notin s\text{-mul-ext } ns \ s$   
**by** (*auto simp: s-mul-ext-def mult2-alt-s-def*)

Obvious introduction rules.

**lemma** *all-ns-ns-mul-ext*:  
**assumes**  $\text{length } as = \text{length } bs$   
**and**  $\forall i. i < \text{length } bs \longrightarrow (as ! i, bs ! i) \in ns$

**shows**  $(mset\ as, mset\ bs) \in ns\text{-mul-ext}\ ns\ s$   
**using** *assms* **by**  $(auto\ intro!: ns\text{-mul-ext}I[of\ -\ -\ \{\#\}\ -\ -\ \{\#\}]\ multpw\text{-list}I)$

**lemma** *all-s-s-mul-ext*:

**assumes**  $A \neq \{\#\}$   
**and**  $\forall b. b \in\# B \longrightarrow (\exists a. a \in\# A \wedge (a,b) \in s)$   
**shows**  $(A, B) \in s\text{-mul-ext}\ ns\ s$   
**using** *assms* **by**  $(auto\ intro!: s\text{-mul-ext}I[of\ -\ \{\#\}\ -\ -\ \{\#\}]\ multpw\text{-list}I)$

Being strictly lesser than implies being lesser than

**lemma** *s-ns-mul-ext*:

**assumes**  $(A, B) \in s\text{-mul-ext}\ ns\ s$   
**shows**  $(A, B) \in ns\text{-mul-ext}\ ns\ s$   
**using** *assms* **by**  $(simp\ add: s\text{-mul-ext-def}\ ns\text{-mul-ext-def}\ mult2\text{-alt-s-implies-mult2-alt-ns})$

The non-strict order is reflexive.

**lemma** *multpw-refl'*:

**assumes** *locally-refl*  $ns\ A$   
**shows**  $(A, A) \in multpw\ ns$

**proof** –

**have**  $Restr\ Id\ (set\text{-}mset\ A) \subseteq ns$  **using** *assms* **by**  $(auto\ simp: locally\text{-refl-def})$   
**from** *refl-multpw*[*of*  $Id$ ] *multpw-local*[*of*  $A\ A\ Id$ ] *mono-multpw*[*OF* *this*]  
**show** *?thesis* **by**  $(auto\ simp: refl\text{-on-def})$

**qed**

**lemma** *ns-mul-ext-refl-local*:

**assumes** *locally-refl*  $ns\ A$   
**shows**  $(A, A) \in ns\text{-mul-ext}\ ns\ s$   
**using** *assms* **by**  $(auto\ intro!: ns\text{-mul-ext}I[of\ A\ A\ \{\#\}\ A\ A\ \{\#\}]\ ns\ s]\ multpw\text{-refl}')$

**lemma** *ns-mul-ext-refl*:

**assumes** *refl*  $ns$   
**shows**  $(A, A) \in ns\text{-mul-ext}\ ns\ s$   
**using** *assms* *ns-mul-ext-refl-local*[*of*  $ns\ A\ s$ ] **unfolding** *refl-on-def* *locally-refl-def*  
**by** *auto*

The orders are union-compatible

**lemma** *ns-s-mul-ext-union-multiset-l*:

**assumes**  $(A, B) \in ns\text{-mul-ext}\ ns\ s$   
**and**  $C \neq \{\#\}$   
**and**  $\forall d. d \in\# D \longrightarrow (\exists c. c \in\# C \wedge (c,d) \in s)$   
**shows**  $(A + C, B + D) \in s\text{-mul-ext}\ ns\ s$   
**using** *assms* **unfolding** *ns-mul-ext-def* *s-mul-ext-def*  
**by**  $(auto\ intro!: converseI\ mult2\text{-alt-ns-s-add}\ mult2\text{-alt-s}I[of\ -\ \{\#\}\ -\ -\ \{\#\}])$

**lemma** *s-mul-ext-union-compat*:

**assumes**  $(A, B) \in s\text{-mul-ext}\ ns\ s$   
**and** *locally-refl*  $ns\ C$   
**shows**  $(A + C, B + C) \in s\text{-mul-ext}\ ns\ s$

**using** *assms ns-mul-ext-refl-local*[*OF assms(2)*] **unfolding** *ns-mul-ext-def s-mul-ext-def*  
**by** (*auto intro!: converseI mult2-alt-s-ns-add*)

**lemma** *ns-mul-ext-union-compat*:

**assumes**  $(A, B) \in ns\text{-mul-ext } ns \ s$

**and** *locally-refl ns C*

**shows**  $(A + C, B + C) \in ns\text{-mul-ext } ns \ s$

**using** *assms ns-mul-ext-refl-local*[*OF assms(2)*] **unfolding** *ns-mul-ext-def s-mul-ext-def*  
**by** (*auto intro!: converseI mult2-alt-ns-ns-add*)

**context**

**fixes** *NS :: 'a rel*

**assumes** *NS: refl NS*

**begin**

**lemma** *refl-imp-locally-refl*: *locally-refl NS A using NS unfolding refl-on-def locally-refl-def by auto*

**lemma** *supseteq-imp-ns-mul-ext*:

**assumes**  $A \supseteq\# B$

**shows**  $(A, B) \in ns\text{-mul-ext } NS \ S$

**using** *assms*

**by** (*auto intro!: ns-mul-extI*[*of A B A - B B B {#}*] *multpw-refl' refl-imp-locally-refl*  
*simp: subset-mset.add-diff-inverse*)

**lemma** *supset-imp-s-mul-ext*:

**assumes**  $A \supset\# B$

**shows**  $(A, B) \in s\text{-mul-ext } NS \ S$

**using** *assms subset-mset.add-diff-inverse*[*of B A*]

**by** (*auto intro!: s-mul-extI*[*of A B A - B B B {#}*] *multpw-refl' refl-imp-locally-refl*  
*simp: Diff-eq-empty-iff-mset*)

**end**

**definition** *mul-ext* ::  $('a \Rightarrow 'a \Rightarrow \text{bool} \times \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \times \text{bool}$

**where** *mul-ext* *f xs ys*  $\equiv \text{let } s = \{(x,y). \text{fst } (f \ x \ y)\}; ns = \{(x,y). \text{snd } (f \ x \ y)\}$

*in*  $((\text{mset } xs, \text{mset } ys) \in s\text{-mul-ext } ns \ s, (\text{mset } xs, \text{mset } ys) \in ns\text{-mul-ext } ns \ s)$

**definition** *smulextp* *f m n*  $\longleftrightarrow (m, n) \in s\text{-mul-ext } \{(x, y). \text{snd } (f \ x \ y)\} \{(x, y). \text{fst } (f \ x \ y)\}$

**definition** *nsmulextp* *f m n*  $\longleftrightarrow (m, n) \in ns\text{-mul-ext } \{(x, y). \text{snd } (f \ x \ y)\} \{(x, y). \text{fst } (f \ x \ y)\}$

**lemma** *smulextp-cong*[*fundef-cong*]:

**assumes**  $xs1 = ys1$

**and**  $xs2 = ys2$

**and**  $\bigwedge x \ x'. x \in\# \ ys1 \Longrightarrow x' \in\# \ ys2 \Longrightarrow f \ x \ x' = g \ x \ x'$

**shows** *smulextp* *f xs1 xs2 = smulextp* *g ys1 ys2*

**unfolding** *smulextp-def*

**proof**  
**assume**  $(xs1, xs2) \in s\text{-mul-ext } \{(x, y). \text{snd } (f x y)\} \{(x, y). \text{fst } (f x y)\}$   
**from**  $s\text{-mul-ext-local-mono}[OF - - \text{this, of } \{(x, y). \text{snd } (g x y)\} \{(x, y). \text{fst } (g x y)\}]$   
**show**  $(ys1, ys2) \in s\text{-mul-ext } \{(x, y). \text{snd } (g x y)\} \{(x, y). \text{fst } (g x y)\}$   
**using** *assms* **by force**  
**next**  
**assume**  $(ys1, ys2) \in s\text{-mul-ext } \{(x, y). \text{snd } (g x y)\} \{(x, y). \text{fst } (g x y)\}$   
**from**  $s\text{-mul-ext-local-mono}[OF - - \text{this, of } \{(x, y). \text{snd } (f x y)\} \{(x, y). \text{fst } (f x y)\}]$   
**show**  $(xs1, xs2) \in s\text{-mul-ext } \{(x, y). \text{snd } (f x y)\} \{(x, y). \text{fst } (f x y)\}$   
**using** *assms* **by force**  
**qed**

**lemma**  $nsmulextp\text{-cong}[fundef\text{-cong}]$ :

**assumes**  $xs1 = ys1$   
**and**  $xs2 = ys2$   
**and**  $\bigwedge x x'. x \in \# ys1 \implies x' \in \# ys2 \implies f x x' = g x x'$   
**shows**  $nsmulextp f xs1 xs2 = nsmulextp g ys1 ys2$   
**unfolding**  $nsmulextp\text{-def}$

**proof**

**assume**  $(xs1, xs2) \in ns\text{-mul-ext } \{(x, y). \text{snd } (f x y)\} \{(x, y). \text{fst } (f x y)\}$   
**from**  $ns\text{-mul-ext-local-mono}[OF - - \text{this, of } \{(x, y). \text{snd } (g x y)\} \{(x, y). \text{fst } (g x y)\}]$   
**show**  $(ys1, ys2) \in ns\text{-mul-ext } \{(x, y). \text{snd } (g x y)\} \{(x, y). \text{fst } (g x y)\}$   
**using** *assms* **by force**  
**next**  
**assume**  $(ys1, ys2) \in ns\text{-mul-ext } \{(x, y). \text{snd } (g x y)\} \{(x, y). \text{fst } (g x y)\}$   
**from**  $ns\text{-mul-ext-local-mono}[OF - - \text{this, of } \{(x, y). \text{snd } (f x y)\} \{(x, y). \text{fst } (f x y)\}]$   
**show**  $(xs1, xs2) \in ns\text{-mul-ext } \{(x, y). \text{snd } (f x y)\} \{(x, y). \text{fst } (f x y)\}$   
**using** *assms* **by force**  
**qed**

**definition**  $mulextp f m n = (smulextp f m n, nsmulextp f m n)$

**lemma**  $mulextp\text{-cong}[fundef\text{-cong}]$ :

**assumes**  $xs1 = ys1$   
**and**  $xs2 = ys2$   
**and**  $\bigwedge x x'. x \in \# ys1 \implies x' \in \# ys2 \implies f x x' = g x x'$   
**shows**  $mulextp f xs1 xs2 = mulextp g ys1 ys2$   
**unfolding**  $mulextp\text{-def}$  **using** *assms* **by** (*auto cong: nsmulextp-cong smulextp-cong*)

**lemma**  $mset\text{-s-mul-ext}$ :

$(mset xs, mset ys) \in s\text{-mul-ext } \{(x, y). \text{snd } (f x y)\} \{(x, y). \text{fst } (f x y)\} \longleftrightarrow$   
 $\text{fst } (mul\text{-ext } f xs ys)$   
**by** (*auto simp: mul-ext-def Let-def*)

**lemma** *mset-ns-mul-ext*:

$(mset\ xs, mset\ ys) \in ns\text{-}mul\text{-}ext\ \{(x, y).\ snd\ (f\ x\ y)\}\ \{(x, y).\ fst\ (f\ x\ y)\} \longleftrightarrow$   
 $snd\ (mul\text{-}ext\ f\ xs\ ys)$   
**by** (*auto simp: mul-ext-def Let-def*)

**lemma** *smulextp-mset-code*:

$smulextp\ f\ (mset\ xs)\ (mset\ ys) \longleftrightarrow fst\ (mul\text{-}ext\ f\ xs\ ys)$   
**unfolding** *smulextp-def mset-s-mul-ext ..*

**lemma** *nsmulextp-mset-code*:

$nsmulextp\ f\ (mset\ xs)\ (mset\ ys) \longleftrightarrow snd\ (mul\text{-}ext\ f\ xs\ ys)$   
**unfolding** *nsmulextp-def mset-ns-mul-ext ..*

**lemma** *nstri-mul-ext-map*:

**assumes**  $\bigwedge s\ t. s \in set\ ss \implies t \in set\ ts \implies fst\ (order\ s\ t) \implies fst\ (order'\ (f\ s)$   
 $(f\ t))$   
**and**  $\bigwedge s\ t. s \in set\ ss \implies t \in set\ ts \implies snd\ (order\ s\ t) \implies snd\ (order'\ (f\ s)\ (f$   
 $t))$   
**and**  $snd\ (mul\text{-}ext\ order\ ss\ ts)$   
**shows**  $snd\ (mul\text{-}ext\ order'\ (map\ f\ ss)\ (map\ f\ ts))$   
**using** *assms mult2-alt-map[of mset ts mset ss {(t, s). snd (order s t)} f f*  
*{(t, s). snd (order' s t)} {(t, s). fst (order s t)} {(t, s). fst (order' s t)} True]*  
**by** (*auto simp: mul-ext-def ns-mul-ext-def converse-unfold*)

**lemma** *stri-mul-ext-map*:

**assumes**  $\bigwedge s\ t. s \in set\ ss \implies t \in set\ ts \implies fst\ (order\ s\ t) \implies fst\ (order'\ (f\ s)$   
 $(f\ t))$   
**and**  $\bigwedge s\ t. s \in set\ ss \implies t \in set\ ts \implies snd\ (order\ s\ t) \implies snd\ (order'\ (f\ s)\ (f$   
 $t))$   
**and**  $fst\ (mul\text{-}ext\ order\ ss\ ts)$   
**shows**  $fst\ (mul\text{-}ext\ order'\ (map\ f\ ss)\ (map\ f\ ts))$   
**using** *assms mult2-alt-map[of mset ts mset ss {(t,s). snd (order s t)} f f*  
*{(t, s). snd (order' s t)} {(t, s). fst (order s t)} {(t, s). fst (order' s t)} False]*  
**by** (*auto simp: mul-ext-def s-mul-ext-def converse-unfold*)

**lemma** *mul-ext-arg-empty*:  $snd\ (mul\text{-}ext\ f\ []\ xs) \implies xs = []$

**unfolding** *mul-ext-def Let-def* **by** (*auto simp: ns-mul-ext-def mult2-alt-def*)

The non-strict order is irreflexive

**lemma** *s-mul-ext-irrefl*: **assumes** *irr: irrefl-on (set-mset A) S*

**and** *S-NS: S ⊆ NS*

**and** *compat: S O NS ⊆ S*

**shows**  $(A, A) \notin s\text{-}mul\text{-}ext\ NS\ S$  **using** *irr*

**proof** (*induct A rule: wf-induct[OF wf-measure[of size]]*)

**case**  $(1\ A)$

**show** *?case*

**proof**

**assume**  $(A, A) \in s\text{-}mul\text{-}ext\ NS\ S$

```

from s-mul-extE[OF this]
obtain A1 A2 B1 B2 where
  A:  $A = A1 + A2$ 
  and B:  $A = B1 + B2$ 
  and AB1:  $(A1, B1) \in \text{multpw } NS$ 
  and ne:  $A2 \neq \{\#\}$ 
  and S:  $\bigwedge b. b \in \# B2 \implies \exists a. a \in \# A2 \wedge (a, b) \in S$ 
  by blast
from multpw-listE[OF AB1] obtain as1 bs1 where
  l1:  $\text{length } as1 = \text{length } bs1$ 
  and A1:  $A1 = \text{mset } as1$ 
  and B1:  $B1 = \text{mset } bs1$ 
  and NS:  $\bigwedge i. i < \text{length } bs1 \implies (as1 ! i, bs1 ! i) \in NS$  by blast

note NSS = NS
note SS = S

obtain as2 where A2:  $A2 = \text{mset } as2$  by (metis ex-mset)
obtain bs2 where B2:  $B2 = \text{mset } bs2$  by (metis ex-mset)
define as where  $as = as1 @ as2$ 
define bs where  $bs = bs1 @ bs2$ 
have as:  $A = \text{mset } as$  unfolding A A1 A2 as-def by simp
have bs:  $A = \text{mset } bs$  unfolding B B1 B2 bs-def by simp
from as bs have abs:  $\text{mset } as = \text{mset } bs$  by simp
hence set-ab:  $\text{set } as = \text{set } bs$  by (rule mset-eq-setD)
let ?n =  $\text{length } bs$ 
have las:  $\text{length } as = ?n$ 
  using mset-eq-length abs by fastforce
let ?m =  $\text{length } bs1$ 
define decr where  $decr j i \equiv$ 
   $(as ! j, bs ! i) \in NS \wedge (i < ?m \longrightarrow j = i) \wedge (?m \leq i \longrightarrow ?m \leq j \wedge (as ! j,$ 
bs ! i) \in S) for i j
define step where  $step i j k =$ 
   $(i < ?n \wedge j < ?n \wedge k < ?n \wedge bs ! k = as ! j \wedge decr j i)$ 
for i j k
{
  fix i
  assume i:  $i < ?n$ 
  let ?b =  $bs ! i$ 
  have  $\exists j. j < ?n \wedge decr j i$ 
  proof (cases i < ?m)
    case False
    with i have ?b  $\in \text{set } bs2$  unfolding bs-def
    by (auto simp: nth-append)
    hence ?b  $\in \# B2$  unfolding B2 by auto
    from S[OF this, unfolded A2] obtain a where  $a \in \text{set } as2$  and S:  $(a,$ 
?b) \in S
    by auto
    from a obtain k where  $a = as2 ! k$  and k:  $k < \text{length } as2$  unfolding

```

```

set-conv-nth by auto
  have a = as ! (?m + k) unfolding a as-def l1[symmetric] by simp
  from S[unfolded this] S-NS False k
  show ?thesis unfolding decr-def
  by (intro exI[of - ?m + k], auto simp: las[symmetric] l1[symmetric] as-def)
next
  case True
  from NS[OF this] i True show ?thesis unfolding decr-def
  by (auto simp: as-def bs-def l1 nth-append)
qed (insert i NS)
from this[unfolded set-conv-nth] las
obtain j where j: j < ?n and decr: decr j i by auto
let ?a = as ! j
from j las have ?a ∈ set as by auto
from this[unfolded set-ab, unfolded set-conv-nth] obtain k where
  k: k < ?n and id: ?a = bs ! k by auto
have ∃ j k. step i j k
  using j k decr id i unfolding step-def by metis
}
hence ∀ i. ∃ j k. i < ?n → step i j k by blast
from choice[OF this] obtain J' where ∀ i. ∃ k. i < ?n → step i (J' i) k by
blast
from choice[OF this] obtain K' where
  step: ∧ i. i < ?n ⇒ step i (J' i) (K' i) by blast
define I where I i = (K' i) 0 for i
define J where J i = J' (I i) for i
define K where K i = K' (I i) for i
from ne have A ≠ {} unfolding A by auto
hence set as ≠ {} unfolding as by auto
hence length as ≠ 0 by simp
hence n0: 0 < ?n using las by auto
{
  fix n
  have step (I n) (J n) (K n)
  proof (induct n)
    case 0
    from step[OF n0] show ?case unfolding I-def J-def K-def by auto
  next
    case (Suc n)
    from Suc have K n < ?n unfolding step-def by auto
    from step[OF this] show ?case unfolding J-def K-def I-def by auto
  qed
}
note step = this
have I n ∈ {..<?n} for n using step[of n] unfolding step-def by auto
hence I ' UNIV ⊆ {..<?n} by auto
from finite-subset[OF this] have finite (I ' UNIV) by simp
from pigeonhole-infinite[OF - this] obtain m where
  infinite {i. I i = I m} by auto

```

```

hence  $\exists m'. m' > m \wedge I m' = I m$ 
  by (simp add: infinite-nat-iff-unbounded)
then obtain  $m'$  where  $*$ :  $m < m' \wedge I m' = I m$  by auto
let  $?P = \lambda n. \exists m. n \neq 0 \wedge I (n + m) = I m$ 
define  $n$  where  $n = (\text{LEAST } n. ?P n)$ 
have  $\exists n. ?P n$ 
  by (rule exI[of -  $m' - m$ ], rule exI[of -  $m$ ], insert *, auto)
from LeastI-ex[of  $?P$ , OF this, folded n-def]
obtain  $m$  where  $n: n \neq 0$  and  $Im: I (n + m) = I m$  by auto
let  $?M = \{m..<m+n\}$ 
{
  fix  $i j$ 
  assume  $*$ :  $m \leq i < j < n + m$ 
  define  $k$  where  $k = j - i$ 
  have  $k0: k \neq 0$  and  $j: j = k + i$  and  $kn: k < n$  using * unfolding k-def
by auto
  from not-less-Least[of -  $?P$ , folded n-def, OF kn] k0
  have  $I i \neq I j$  unfolding j by metis
}
hence inj: inj-on  $I$   $?M$  unfolding inj-on-def
  by (metis add.commute atLeastLessThan-iff linorder-neqE-nat)
define  $b$  where  $b i = bs ! I i$  for  $i$ 
have  $bnm: b (n + m) = b m$  unfolding b-def Im ..
{
  fix  $i$ 
  from step[of  $i$ , unfolded step-def]
  have  $id: bs ! K i = as ! J i$  and  $decr: decr (J i) (I i)$  by auto
  from id decr[unfolded decr-def] have  $(bs ! K i, bs ! I i) \in NS$  by auto
  also have  $K i = I (Suc i)$  unfolding I-def K-def by auto
  finally have  $(b (Suc i), b i) \in NS$  unfolding b-def by auto
} note NS = this
{
  fix  $i j :: nat$ 
  assume  $i \leq j$ 
  then obtain  $k$  where  $j: j = i + k$  by (rule less-eqE)
  have  $(b j, b i) \in NS^*$  unfolding j
  proof (induct k)
    case (Suc k)
    thus ?case using NS[of  $i + k$ ] by auto
  qed auto
} note NSstar = this
{
  assume  $\exists i \in ?M. I i \geq ?m$ 
  then obtain  $k$  where  $k: k \in ?M$  and  $I: I k \geq ?m$  by auto
  from step[of  $k$ , unfolded step-def]
  have  $id: bs ! K k = as ! J k$  and  $decr: decr (J k) (I k)$  by auto
  from id decr[unfolded decr-def] I have  $(bs ! K k, bs ! I k) \in S$  by auto
  also have  $K k = I (Suc k)$  unfolding I-def K-def by auto
  finally have  $S: (b (Suc k), b k) \in S$  unfolding b-def by auto
}

```

```

from  $k$  have  $m \leq k$  by auto
from  $NSstar[OF\ this]$  have  $NS1: (b\ k, b\ m) \in NS^*$  .
from  $k$  have  $Suc\ k \leq n + m$  by auto
from  $NSstar[OF\ this, unfolded\ bnm]$  have  $NS2: (b\ m, b\ (Suc\ k)) \in NS^*$  .
from  $NS1\ NS2$  have  $(b\ k, b\ (Suc\ k)) \in NS^*$  by simp
with  $S$  have  $(b\ (Suc\ k), b\ (Suc\ k)) \in S\ O\ NS^*$  by auto
also have  $\dots \subseteq S$  using compat
by (metis\ compat-tr-compat\ converse-inward(1)\ converse-mono\ converse-relcomp)
  finally have contradiction:  $b\ (Suc\ k) \notin set-mset\ A$  using  $1$  unfolding
irrefl-on-def by auto
  have  $b\ (Suc\ k) \in set\ bs$  unfolding b-def using step[of\ Suc\ k] unfolding
step-def
  by auto
  also have  $set\ bs = set-mset\ A$  unfolding bs by auto
  finally have False using contradiction by auto
}
hence only-NS:  $i \in ?M \implies I\ i < ?m$  for  $i$  by force
{
  fix  $i$ 
  assume  $i: i \in ?M$ 
  from step[of\ i, unfolded\ step-def] have  $*$ :  $I\ i < ?n\ K\ i < ?n$ 
    and  $id: bs\ !\ K\ i = as\ !\ J\ i$  and  $decr: decr\ (J\ i)\ (I\ i)$  by auto
  from decr[unfolded\ decr-def] only-NS[OF\ i] have  $J\ i = I\ i$  by auto
  with  $id$  have  $id: bs\ !\ K\ i = as\ !\ I\ i$  by auto
  note only-NS[OF\ i]  $id$ 
} note pre-result = this
{
  fix  $i$ 
  assume  $i: i \in ?M$ 
  have  $*$ :  $I\ i < ?m\ K\ i < ?m$ 
  proof (rule\ pre-result[OF\ i])
    have  $\exists j \in ?M. K\ i = I\ j$ 
    proof (cases\ Suc\ i \in ?M)
      case True
        show ?thesis by (rule\ bexI[OF\ -\ True], auto\ simp: K-def\ I-def)
      next
        case False
          with  $i$  have  $id: n + m = Suc\ i$  by auto
          hence  $id: K\ i = I\ m$  by (subst\ Im[symmetric], unfold\ id, auto\ simp: K-def
I-def)
          with  $i$  show ?thesis by (intro\ bexI[of\ -\ m], auto\ simp: K-def\ I-def)
    qed
  with pre-result show  $K\ i < ?m$  by auto
  qed
from pre-result(2)[OF\ i]  $*\ l1$  have  $bs1\ !\ K\ i = as1\ !\ I\ i\ K\ i = I\ (Suc\ i)$ 
  unfolding as-def\ bs-def by (auto\ simp: nth-append\ K-def\ I-def)
with  $*$  have  $bs1\ !\ I\ (Suc\ i) = as1\ !\ I\ i\ I\ i < ?m\ I\ (Suc\ i) < ?m$ 
  by auto
} note pre-identities = this

```

```

define M where M = ?M
note inj = inj[folded M-def]
define next where next i = (if Suc i = n + m then m else Suc i) for i
define prev where prev i = (if i = m then n + m - 1 else i - 1) for i
{
  fix i
  assume i ∈ M
  hence i: i ∈ ?M unfolding M-def by auto
  from i n have inM: next i ∈ M prev i ∈ M next (prev i) = i prev (next i) = i
    unfolding next-def prev-def by (auto simp: M-def)
  from i pre-identities[OF i] pre-identities[of m] Im n
  have next: bs1 ! I (next i) = as1 ! I i
    unfolding next-def prev-def by (auto simp: M-def)
  note next inM
} note identities = this

```

```

note identities = identities[folded M-def]
define Drop where Drop = I ' M

```

```

define rem-idx where rem-idx = filter (λ i. i ∉ Drop) [0..<?m]
define drop-idx where drop-idx = filter (λ i. i ∈ Drop) [0..<?m]
define as1' where as1' = map (!) as1 rem-idx
define bs1' where bs1' = map (!) bs1 rem-idx
define as1'' where as1'' = map (!) as1 drop-idx
define bs1'' where bs1'' = map (!) bs1 drop-idx
{
  fix as1 :: 'a list and D :: nat set
  define I where I = [0..< length as1]
  have mset as1 = mset (map (!) as1 I) unfolding I-def
    by (rule arg-cong[of - - mset], intro nth-equalityI, auto)
  also have ... = mset (map (!) as1 (filter (λ i. i ∈ D) I))
    + mset (map (!) as1 (filter (λ i. i ∉ D) I))
    by (induct I, auto)
  also have I = [0..< length as1] by fact
  finally have mset as1 = mset (map (!) as1 (filter (λ i. i ∈ D) [0..<length
as1])) + mset (map (!) as1 (filter (λ i. i ∉ D) [0..<length as1])) .
} note split = this
from split[of bs1 Drop, folded rem-idx-def drop-idx-def, folded bs1'-def bs1''-def]
have bs1: mset bs1 = mset bs1'' + mset bs1' .
from split[of as1 Drop, unfolded l1, folded rem-idx-def drop-idx-def, folded
as1'-def as1''-def]
have as1: mset as1 = mset as1'' + mset as1' .

```

```

define I' where I' = the-inv-into M I
have bij: bij-betw I M Drop using inj unfolding Drop-def by (rule inj-on-imp-bij-betw)
from the-inv-into-f-f[OF inj, folded I'-def] have I'I: i ∈ M ⇒ I' (I i) = i

```

```

for  $i$  by auto
  from  $\text{bij } I'I$  have  $II': i \in \text{Drop} \implies I (I' i) = i$  for  $i$ 
    by (simp add: I'-def f-the-inv-into-f-bij-betw)
  from  $II' I'I$  identities  $\text{bij}$  have  $\text{Drop-M}: i \in \text{Drop} \implies I' i \in M$  for  $i$ 
    using Drop-def by force
  have  $M\text{-Drop}: i \in M \implies I i \in \text{Drop}$  for  $i$  unfolding Drop-def by auto
  {
    fix  $x$ 
    assume  $x \in \text{Drop}$ 
    then obtain  $i$  where  $i: i \in M$  and  $x: x = I i$  unfolding Drop-def by auto
    have  $x < ?m$  unfolding  $x$  using  $i$  pre-identities[of i] unfolding M-def by
auto
  } note  $\text{Drop-m} = \text{this}$ 
  hence  $\text{drop-idx}: \text{set drop-idx} = \text{Drop}$  unfolding M-def drop-idx-def set-filter
set-upt by auto
  have  $\text{mset as1}'' = \text{mset} (\text{map} (!) \text{as1}) \text{drop-idx}$  unfolding  $\text{as1}''\text{-def}$  mset-map
by auto
  also have  $\text{drop-idx} = \text{map} (I \circ I') \text{drop-idx}$  using  $\text{drop-idx}$  by (intro nth-equalityI,
auto intro!: II'[symmetric])
  also have  $\text{map} (!) \text{as1} \dots = \text{map} (\lambda i. \text{as1} ! I i) (\text{map } I' \text{drop-idx})$  by auto
  also have  $\dots = \text{map} (\lambda i. \text{bs1} ! I (\text{next } i)) (\text{map } I' \text{drop-idx})$ 
    by (rule map-cong[OF refl], rule identities(1)[symmetric], insert drop-idx
Drop-M, auto)
  also have  $\dots = \text{map} (!) \text{bs1} (\text{map} (I \circ \text{next} \circ I') \text{drop-idx})$ 
    by auto
  also have  $\text{mset} \dots = \text{image-mset} (!) \text{bs1} (\text{image-mset} (I \circ \text{next} \circ I') (\text{mset}$ 
drop-idx)) unfolding mset-map ..
  also have  $\text{image-mset} (I \circ \text{next} \circ I') (\text{mset } \text{drop-idx}) = \text{image-mset } I (\text{image-mset}$ 
next (image-mset } I' (\text{mset } \text{drop-idx})))
    by (metis multiset.map-comp)
  also have  $\text{image-mset } \text{next} (\text{image-mset } I' (\text{mset } \text{drop-idx})) = \text{image-mset } I'$ 
(mset drop-idx)
  proof –
    have  $\text{dist}: \text{distinct } \text{drop-idx}$  unfolding drop-idx-def by auto
    have  $\text{injI}': \text{inj-on } I' \text{Drop}$  using  $II'$  by (rule inj-on-inverseI)
    have  $\text{mset } \text{drop-idx} = \text{mset-set } \text{Drop}$  unfolding  $\text{drop-idx}$ [symmetric]
      by (rule mset-set-set[symmetric, OF dist])
    from  $\text{image-mset-mset-set}[OF \text{injI}', \text{folded this}]$ 
    have  $\text{image-mset } I' (\text{mset } \text{drop-idx}) = \text{mset-set} (I' \text{' Drop})$  by auto
    also have  $I' \text{' Drop} = M$  using  $II' I'I M\text{-Drop } \text{Drop-M}$  by force
    finally have  $\text{id}: \text{image-mset } I' (\text{mset } \text{drop-idx}) = \text{mset-set } M$  .
    have  $\text{inj-next}: \text{inj-on } \text{next } M$  using  $\text{identities}$  by (intro inj-on-inverseI)
    have  $\text{next}: \text{next } \text{' } M = M$  using  $\text{identities}$  by force
    show  $?thesis$  unfolding  $\text{id}$   $\text{image-mset-mset-set}[OF \text{inj-next}] \text{next} ..$ 
  qed
  also have  $\text{image-mset } I \dots = \text{mset } \text{drop-idx}$  unfolding multiset.map-comp
using  $II'$ 
    by (intro multiset.map-ident-strong, auto simp: drop-idx)
  also have  $\text{image-mset} (!) \text{bs1} \dots = \text{mset } \text{bs1}''$  unfolding  $\text{bs1}''\text{-def}$  mset-map

```

..

**finally have**  $bs1''$ :  $mset\ bs1'' = mset\ as1''$  ..

**let**  $?A = mset\ as1' + mset\ as2$   
**let**  $?B = mset\ bs1' + mset\ bs2$   
**from**  $as1\ bs1''$  **have**  $as1$ :  $mset\ as1 = mset\ bs1'' + mset\ as1'$  **by** *auto*  
**have**  $A$ :  $A = mset\ bs1'' + ?A$  **unfolding**  $A\ A1\ A2\ as1$  **by** *auto*  
**have**  $B$ :  $A = mset\ bs1'' + ?B$  **unfolding**  $B\ B1\ B2\ bs1$  **by** *auto*  
**from**  $A$ [*unfolded B*] **have**  $AB$ :  $?A = ?B$  **by** *simp*

**have**  $l1'$ :  $length\ as1' = length\ bs1'$  **unfolding**  $as1'$ -*def*  $bs1'$ -*def* **by** *auto*  
**have**  $NS$ :  $(mset\ as1', mset\ bs1') \in multpw\ NS$   
**proof** (*rule multpw-listI[OF l1' refl refl]*, *intro allI impI*)

**fix**  $i$

**assume**  $i$ :  $i < length\ bs1'$

**hence**  $rem-idx\ !\ i \in set\ rem-idx$  **unfolding**  $bs1'$ -*def* **by** (*auto simp: nth-append*)

**hence**  $ri$ :  $rem-idx\ !\ i < ?m$  **unfolding**  $rem-idx$ -*def* **by** *auto*

**from**  $NSS$ [*OF this*]  $i$

**show**  $(as1'\ !\ i, bs1'\ !\ i) \in NS$  **unfolding**  $as1'$ -*def*  $bs1'$ -*def* **by** (*auto simp: nth-append*)

**qed**

**have**  $S$ :  $(mset\ as1' + mset\ as2, ?B) \in s\text{-mul-ext}\ NS\ S$

**by** (*intro s-mul-extI[OF refl refl NS]*, *unfold A2[symmetric] B2[symmetric]*, *rule ne, rule S*)

**have**  $irr$ : *irrefl-on* (*set-mset*  $?B$ )  $S$  **using**  $1(2)\ B$  **unfolding** *irrefl-on-def* **by** *simp*

**have**  $M \neq \{\}$  **unfolding**  $M$ -*def* **using**  $n$  **by** *auto*

**hence**  $Drop \neq \{\}$  **unfolding**  $Drop$ -*def* **by** *auto*

**with**  $drop-idx$  **have**  $drop-idx \neq []$  **by** *auto*

**hence**  $bs1'' \neq []$  **unfolding**  $bs1''$ -*def* **by** *auto*

**hence**  $?B \subset\# A$  **unfolding**  $B$  **by** (*simp add: subset-mset.less-le*)

**hence**  $size\ ?B < size\ A$  **by** (*rule mset-subset-size*)

**thus** *False* **using**  $1(1)\ AB\ S\ irr$  **by** *auto*

**qed**

**qed**

**lemma** *mul-ext-irrefl*: **assumes**  $\bigwedge x. x \in set\ xs \implies \neg fst\ (rel\ x\ x)$

**and**  $\bigwedge x\ y\ z. fst\ (rel\ x\ y) \implies snd\ (rel\ y\ z) \implies fst\ (rel\ x\ z)$

**and**  $\bigwedge x\ y. fst\ (rel\ x\ y) \implies snd\ (rel\ x\ y)$

**shows**  $\neg fst\ (mul\text{-ext}\ rel\ xs\ xs)$

**unfolding**  $mul\text{-ext}\text{-def}$  *Let-def* *fst-conv*

**by** (*rule s-mul-ext-irrefl, insert assms, auto simp: irrefl-on-def*)

The non-strict order is transitive.

**lemma** *ns-mul-ext-trans*:

**assumes**  $trans\ s\ trans\ ns\ compatible\text{-l}\ ns\ s\ compatible\text{-r}\ ns\ s\ refl\ ns$

**and**  $(A, B) \in ns\text{-mul-ext}\ ns\ s$

**and**  $(B, C) \in ns\text{-mul-ext}\ ns\ s$

**shows**  $(A, C) \in ns\text{-mul-ext } ns \ s$   
**using** *assms* **unfolding** *compatible-l-def compatible-r-def ns-mul-ext-def*  
**using** *trans-mult2-ns*[of  $s^{-1} \ ns^{-1}$ ]  
**by** (*auto simp: mult2-ns-eq-mult2-ns-alt converse-relcomp*[*symmetric*]) (*metis trans-def*)

The strict order is trans.

**lemma** *s-mul-ext-trans*:

**assumes** *trans s trans ns compatible-l ns s compatible-r ns s refl ns*  
**and**  $(A, B) \in s\text{-mul-ext } ns \ s$   
**and**  $(B, C) \in s\text{-mul-ext } ns \ s$   
**shows**  $(A, C) \in s\text{-mul-ext } ns \ s$   
**using** *assms* **unfolding** *compatible-l-def compatible-r-def s-mul-ext-def*  
**using** *trans-mult2-s*[of  $s^{-1} \ ns^{-1}$ ]  
**by** (*auto simp: mult2-s-eq-mult2-s-alt converse-relcomp*[*symmetric*]) (*metis trans-def*)

The strict order is compatible on the left with the non strict one

**lemma** *s-ns-mul-ext-trans*:

**assumes** *trans s trans ns compatible-l ns s compatible-r ns s refl ns*  
**and**  $(A, B) \in s\text{-mul-ext } ns \ s$   
**and**  $(B, C) \in ns\text{-mul-ext } ns \ s$   
**shows**  $(A, C) \in s\text{-mul-ext } ns \ s$   
**using** *assms* **unfolding** *compatible-l-def compatible-r-def s-mul-ext-def ns-mul-ext-def*  
**using** *compat-mult2(1)*[of  $s^{-1} \ ns^{-1}$ ]  
**by** (*auto simp: mult2-s-eq-mult2-s-alt mult2-ns-eq-mult2-ns-alt converse-relcomp*[*symmetric*])

The strict order is compatible on the right with the non-strict one.

**lemma** *ns-s-mul-ext-trans*:

**assumes** *trans s trans ns compatible-l ns s compatible-r ns s refl ns*  
**and**  $(A, B) \in ns\text{-mul-ext } ns \ s$   
**and**  $(B, C) \in s\text{-mul-ext } ns \ s$   
**shows**  $(A, C) \in s\text{-mul-ext } ns \ s$   
**using** *assms* **unfolding** *compatible-l-def compatible-r-def s-mul-ext-def ns-mul-ext-def*  
**using** *compat-mult2(2)*[of  $s^{-1} \ ns^{-1}$ ]  
**by** (*auto simp: mult2-s-eq-mult2-s-alt mult2-ns-eq-mult2-ns-alt converse-relcomp*[*symmetric*])

*s-mul-ext* is strongly normalizing

**lemma** *SN-s-mul-ext-strong*:

**assumes** *order-pair s ns*  
**and**  $\forall y. y \in \# M \longrightarrow SN\text{-on } s \ \{y\}$   
**shows**  $SN\text{-on } (s\text{-mul-ext } ns \ s) \ \{M\}$   
**using** *mult2-s-eq-mult2-s-alt*[of  $ns^{-1} \ s^{-1}$ ] *assms wf-below-pointwise*[of  $s^{-1} \ set\text{-mset } M$ ]  
**unfolding** *SN-on-iff-wf-below s-mul-ext-def order-pair-def compat-pair-def pre-order-pair-def*  
**by** (*auto intro!: wf-below-mult2-s-local simp: converse-relcomp*[*symmetric*])

**lemma** *SN-s-mul-ext*:

**assumes** *order-pair s ns SN s*  
**shows**  $SN \ (s\text{-mul-ext } ns \ s)$

**using** *SN-s-mul-ext-strong*[*OF assms(1)*] *assms(2)*  
**by** (*auto simp: SN-on-def*)

**lemma** (**in** *order-pair*) *mul-ext-order-pair*:

*order-pair* (*s-mul-ext NS S*) (*ns-mul-ext NS S*) (**is** *order-pair* *?S ?NS*)

**proof**

**from** *s-mul-ext-trans trans-S trans-NS compat-NS-S compat-S-NS refl-NS*

**show** *trans ?S unfolding trans-def compatible-l-def compatible-r-def* **by** *blast*

**next**

**from** *ns-mul-ext-trans trans-S trans-NS compat-NS-S compat-S-NS refl-NS*

**show** *trans ?NS unfolding trans-def compatible-l-def compatible-r-def* **by** *blast*

**next**

**from** *ns-s-mul-ext-trans trans-S trans-NS compat-NS-S compat-S-NS refl-NS*

**show** *?NS O ?S  $\subseteq$  ?S unfolding trans-def compatible-l-def compatible-r-def* **by**

*blast*

**next**

**from** *s-ns-mul-ext-trans trans-S trans-NS compat-NS-S compat-S-NS refl-NS*

**show** *?S O ?NS  $\subseteq$  ?S unfolding trans-def compatible-l-def compatible-r-def* **by**

*blast*

**next**

**from** *ns-mul-ext-refl*[*OF refl-NS, of - S*]

**show** *refl ?NS unfolding refl-on-def* **by** *fast*

**qed**

**lemma** (**in** *SN-order-pair*) *mul-ext-SN-order-pair*: *SN-order-pair* (*s-mul-ext NS S*)  
(*ns-mul-ext NS S*)

(**is** *SN-order-pair* *?S ?NS*)

**proof** –

**from** *mul-ext-order-pair*

**interpret** *order-pair* *?S ?NS* .

**have** *order-pair S NS* **by** *unfold-locales*

**then interpret** *SN-ars ?S* **using** *SN-s-mul-ext*[*of S NS*] *SN* **by** *unfold-locales*

**show** *?thesis* **by** *unfold-locales*

**qed**

**lemma** *mul-ext-compat*:

**assumes** *compat*:  $\bigwedge s t u. \llbracket s \in \text{set } ss; t \in \text{set } ts; u \in \text{set } us \rrbracket \implies$

$(\text{snd } (f s t) \wedge \text{fst } (f t u) \longrightarrow \text{fst } (f s u)) \wedge$

$(\text{fst } (f s t) \wedge \text{snd } (f t u) \longrightarrow \text{fst } (f s u)) \wedge$

$(\text{snd } (f s t) \wedge \text{snd } (f t u) \longrightarrow \text{snd } (f s u)) \wedge$

$(\text{fst } (f s t) \wedge \text{fst } (f t u) \longrightarrow \text{fst } (f s u))$

**shows**

$(\text{snd } (\text{mul-ext } f ss ts) \wedge \text{fst } (\text{mul-ext } f ts us) \longrightarrow \text{fst } (\text{mul-ext } f ss us)) \wedge$

$(\text{fst } (\text{mul-ext } f ss ts) \wedge \text{snd } (\text{mul-ext } f ts us) \longrightarrow \text{fst } (\text{mul-ext } f ss us)) \wedge$

$(\text{snd } (\text{mul-ext } f ss ts) \wedge \text{snd } (\text{mul-ext } f ts us) \longrightarrow \text{snd } (\text{mul-ext } f ss us)) \wedge$

$(\text{fst } (\text{mul-ext } f ss ts) \wedge \text{fst } (\text{mul-ext } f ts us) \longrightarrow \text{fst } (\text{mul-ext } f ss us))$

**proof** –

**let**  $?s = \{(x, y). \text{fst } (f x y)\}^{-1}$  **and**  $?ns = \{(x, y). \text{snd } (f x y)\}^{-1}$

**have** [*dest*]:  $(\text{mset } ts, \text{mset } ss) \in \text{mult2-alt } b2 ?ns ?s \implies (\text{mset } us, \text{mset } ts) \in$

*mult2-alt*  $b1 \text{ ?ns ?s} \implies$   
 $(mset\ us, mset\ ss) \in \text{mult2-alt } (b1 \wedge b2) \text{ ?ns ?s}$  **for**  $b1\ b2$   
**using** *assms* **by** (*auto intro!*: *trans-mult2-alt-local*[*of - mset ts*] *simp: in-multiset-in-set*)  
**show** *?thesis* **by** (*auto simp: mul-ext-def s-mul-ext-def ns-mul-ext-def Let-def*)  
**qed**

**lemma** *mul-ext-cong*[*fundef-cong*]:  
**assumes**  $mset\ xs1 = mset\ ys1$   
**and**  $mset\ xs2 = mset\ ys2$   
**and**  $\bigwedge x\ x'. x \in set\ ys1 \implies x' \in set\ ys2 \implies f\ x\ x' = g\ x\ x'$   
**shows**  $mul-ext\ f\ xs1\ xs2 = mul-ext\ g\ ys1\ ys2$   
**using** *assms*  
 $mult2-alt-map[of\ mset\ xs2\ mset\ xs1\ \{(x, y). snd\ (f\ x\ y)\}^{-1}\ id\ id\ \{(x, y). snd\ (g\ x\ y)\}^{-1}]$   
 $\{(x, y). fst\ (f\ x\ y)\}^{-1}\ \{(x, y). fst\ (g\ x\ y)\}^{-1}$   
 $mult2-alt-map[of\ mset\ ys2\ mset\ ys1\ \{(x, y). snd\ (g\ x\ y)\}^{-1}\ id\ id\ \{(x, y). snd\ (f\ x\ y)\}^{-1}]$   
 $\{(x, y). fst\ (g\ x\ y)\}^{-1}\ \{(x, y). fst\ (f\ x\ y)\}^{-1}$   
**by** (*auto simp: mul-ext-def s-mul-ext-def ns-mul-ext-def Let-def in-multiset-in-set*)

**lemma** *all-nstri-imp-mul-nstri*:  
**assumes**  $\forall i < length\ ys. snd\ (f\ (xs\ !\ i)\ (ys\ !\ i))$   
**and**  $length\ xs = length\ ys$   
**shows**  $snd\ (mul-ext\ f\ xs\ ys)$   
**proof**–  
**from** *assms*(1) **have**  $\forall i. i < length\ ys \longrightarrow (xs\ !\ i, ys\ !\ i) \in \{(x, y). snd\ (f\ x\ y)\}$   
**by** *simp*  
**from** *all-ns-ns-mul-ext*[*OF assms*(2) *this*] **show** *?thesis* **by** (*simp add: mul-ext-def*)  
**qed**

**lemma** *relation-inter*:  
**shows**  $\{(x, y). P\ x\ y\} \cap \{(x, y). Q\ x\ y\} = \{(x, y). P\ x\ y \wedge Q\ x\ y\}$   
**by** *blast*

**lemma** *mul-ext-unfold*:  
 $(x, y) \in \{(a, b). fst\ (mul-ext\ g\ a\ b)\} \longleftrightarrow (mset\ x, mset\ y) \in (s-mul-ext\ \{(a, b). snd\ (g\ a\ b)\}\ \{(a, b). fst\ (g\ a\ b)\})$   
**unfolding** *mul-ext-def* **by** (*simp add: Let-def*)

The next lemma is a local version of strong-normalization of the multi-set extension, where the base-order only has to be strongly normalizing on elements of the multisets. This will be crucial for orders that are defined recursively on terms, such as RPO or WPO.

**lemma** *mul-ext-SN*:  
**assumes**  $\forall x. snd\ (g\ x\ x)$   
**and**  $\forall x\ y\ z. fst\ (g\ x\ y) \longrightarrow snd\ (g\ y\ z) \longrightarrow fst\ (g\ x\ z)$   
**and**  $\forall x\ y\ z. snd\ (g\ x\ y) \longrightarrow fst\ (g\ y\ z) \longrightarrow fst\ (g\ x\ z)$   
**and**  $\forall x\ y\ z. snd\ (g\ x\ y) \longrightarrow snd\ (g\ y\ z) \longrightarrow snd\ (g\ x\ z)$   
**and**  $\forall x\ y\ z. fst\ (g\ x\ y) \longrightarrow fst\ (g\ y\ z) \longrightarrow fst\ (g\ x\ z)$

**shows**  $SN \{(ys, xs).$   
 $(\forall y \in set \ ys. SN-on \{(s, t). fst (g \ s \ t)\} \{y\}) \wedge$   
 $fst (mul-ext \ g \ ys \ xs)\}$   
**proof** –  
**let**  $?R1 = \lambda xs \ ys. \forall y \in set \ ys. SN-on \{(s, t). fst (g \ s \ t)\} \{y\}$   
**let**  $?R2 = \lambda xs \ ys. fst (mul-ext \ g \ ys \ xs)$   
**let**  $?s = \{(x, y). fst (g \ x \ y)\}$  **and**  $?ns = \{(x, y). snd (g \ x \ y)\}$   
**have**  $OP$ : *order-pair*  $?s \ ?ns$  **using**  $assms(1-5)$   
**by** *unfold-locales*  $((unfold \ refl-on-def \ trans-def)?, blast)+$   
**let**  $?R = \{(ys, xs). ?R1 \ xs \ ys \wedge ?R2 \ xs \ ys\}$   
**let**  $?Sn = SN-on \ ?R$   
**{**  
**fix**  $ys \ xs$   
**assume**  $R-ys-xs: (ys, xs) \in ?R$   
**let**  $?mys = mset \ ys$   
**let**  $?mxs = mset \ xs$   
**from**  $R-ys-xs$  **have**  $HSN-ys: \forall y. y \in set \ ys \longrightarrow SN-on \ ?s \ \{y\}$  **by** *simp*  
**with** *in-multiset-in-set*[*of*  $ys$ ] **have**  $\forall y. y \in \# \ ?mys \longrightarrow SN-on \ ?s \ \{y\}$  **by** *simp*  
**from** *SN-s-mul-ext-strong*[*OF*  $OP$  *this*] **and** *mul-ext-unfold*  
**have**  $SN-on \{(ys, xs). fst (mul-ext \ g \ ys \ xs)\} \{ys\}$  **by** *fast*  
**from** *relation-inter*[*of*  $?R2 \ ?R1$ ] **and** *SN-on-weakening*[*OF* *this*]  
**have**  $SN-on \ ?R \ \{ys\}$  **by** *blast*  
**}**  
**then** **have**  $Hyp: \forall ys \ xs. (ys, xs) \in ?R \longrightarrow SN-on \ ?R \ \{ys\}$  **by** *auto*  
**{**  
**fix**  $ys$   
**have**  $SN-on \ ?R \ \{ys\}$   
**proof** (*cases*  $\exists \ xs. (ys, xs) \in ?R$ )  
**case** *True* **with**  $Hyp$  **show**  $?thesis$  **by** *simp*  
**next**  
**case** *False* **then** **show**  $?thesis$  **by** *auto*  
**qed**  
**}**  
**then** **show**  $?thesis$  **unfolding** *SN-on-def* **by** *simp*  
**qed**

**lemma** *mul-ext-stri-imp-nstri*:  
**assumes**  $fst (mul-ext \ f \ as \ bs)$   
**shows**  $snd (mul-ext \ f \ as \ bs)$   
**using**  $assms$  **and** *s-ns-mul-ext* **unfolding** *mul-ext-def* **by** (*auto simp: Let-def*)

**lemma** *ns-ns-mul-ext-union-compat*:  
**assumes**  $(A, B) \in ns-mul-ext \ ns \ s$   
**and**  $(C, D) \in ns-mul-ext \ ns \ s$   
**shows**  $(A + C, B + D) \in ns-mul-ext \ ns \ s$   
**using**  $assms$  **by** (*auto simp: ns-mul-ext-def intro: mult2-alt-ns-ns-add*)

**lemma** *s-ns-mul-ext-union-compat*:  
**assumes**  $(A, B) \in s-mul-ext \ ns \ s$

```

    and (C,D) ∈ ns-mul-ext ns s
  shows (A + C, B + D) ∈ s-mul-ext ns s
  using assms by (auto simp: s-mul-ext-def ns-mul-ext-def intro: mult2-alt-s-ns-add)

lemma ns-ns-mul-ext-union-compat-rtrancl: assumes refl: refl ns
  and AB: (A, B) ∈ (ns-mul-ext ns s)*
  and CD: (C, D) ∈ (ns-mul-ext ns s)*
shows (A + C, B + D) ∈ (ns-mul-ext ns s)*
proof -
  {
    fix A B C
    assume (A, B) ∈ (ns-mul-ext ns s)*
    then have (A + C, B + C) ∈ (ns-mul-ext ns s)*
    proof (induct rule: rtrancl-induct)
      case (step B D)
      have (C, C) ∈ ns-mul-ext ns s
        by (rule ns-mul-ext-refl, insert refl, auto simp: locally-refl-def refl-on-def)
      from ns-ns-mul-ext-union-compat[OF step(2) this] step(3)
      show ?case by auto
    qed auto
  }
  from this[OF AB, of C] this[OF CD, of B]
  show ?thesis by (auto simp: ac-simps)
qed

```

#### 4.4 Multisets as order on lists

```

interpretation mul-ext-list: list-order-extension
  λs ns. {(as, bs). (mset as, mset bs) ∈ s-mul-ext ns s}
  λs ns. {(as, bs). (mset as, mset bs) ∈ ns-mul-ext ns s}
proof -
  let ?m = mset :: ('a list ⇒ 'a multiset)
  let ?S = λs ns. {(as, bs). (?m as, ?m bs) ∈ s-mul-ext ns s}
  let ?NS = λs ns. {(as, bs). (?m as, ?m bs) ∈ ns-mul-ext ns s}
  show list-order-extension ?S ?NS
  proof (rule list-order-extension.intro)
    fix s ns
    let ?s = ?S s ns
    let ?ns = ?NS s ns
    assume SN-order-pair s ns
    then interpret SN-order-pair s ns .
    interpret SN-order-pair (s-mul-ext ns s) (ns-mul-ext ns s)
      by (rule mul-ext-SN-order-pair)
    show SN-order-pair ?s ?ns
  proof
    show refl ?ns using refl-NS unfolding refl-on-def by blast
    show ?ns O ?s ⊆ ?s using compat-NS-S by blast
    show ?s O ?ns ⊆ ?s using compat-S-NS by blast
    show trans ?ns using trans-NS unfolding trans-def by blast
  end
end

```

```

    show trans ?s using trans-S unfolding trans-def by blast
    show SN ?s using SN-inv-image[OF SN, of ?m, unfolded inv-image-def] .
qed
next
fix S f NS as bs
assume  $\bigwedge a b. (a, b) \in S \implies (f a, f b) \in S$ 
     $\bigwedge a b. (a, b) \in NS \implies (f a, f b) \in NS$ 
     $(as, bs) \in ?S S NS$ 
then show  $(map f as, map f bs) \in ?S S NS$ 
    using mult2-alt-map[of - - NS-1 f f NS-1 S-1 S-1] by (auto simp: mset-map
s-mul-ext-def)
next
fix S f NS as bs
assume  $\bigwedge a b. (a, b) \in S \implies (f a, f b) \in S$ 
     $\bigwedge a b. (a, b) \in NS \implies (f a, f b) \in NS$ 
     $(as, bs) \in ?NS S NS$ 
then show  $(map f as, map f bs) \in ?NS S NS$ 
    using mult2-alt-map[of - - NS-1 f f NS-1 S-1 S-1] by (auto simp: mset-map
ns-mul-ext-def)
next
fix as bs :: 'a list and NS S :: 'a rel
assume ass: length as = length bs
     $\bigwedge i. i < length bs \implies (as ! i, bs ! i) \in NS$ 
show  $(as, bs) \in ?NS S NS$ 
    by (rule, unfold split, rule all-ns-ns-mul-ext, insert ass, auto)
qed
qed

lemma s-mul-ext-singleton [simp, intro]:
  assumes  $(a, b) \in s$ 
  shows  $(\{\#a\}, \{\#b\}) \in s\text{-mul-ext } ns \ s$ 
  using assms by (auto simp: s-mul-ext-def mult2-alt-s-single)

lemma ns-mul-ext-singleton [simp, intro]:
   $(a, b) \in ns \implies (\{\#a\}, \{\#b\}) \in ns\text{-mul-ext } ns \ s$ 
  by (auto simp: ns-mul-ext-def multpw-converse intro: multpw-implies-mult2-alt-ns
multpw-single)

lemma ns-mul-ext-singleton2:
   $(a, b) \in s \implies (\{\#a\}, \{\#b\}) \in ns\text{-mul-ext } ns \ s$ 
  by (intro s-ns-mul-ext s-mul-ext-singleton)

lemma s-mul-ext-self-extend-left:
  assumes  $A \neq \{\#\}$  and locally-refl W B
  shows  $(A + B, B) \in s\text{-mul-ext } W \ S$ 
proof -
  have  $(A + B, \{\#\} + B) \in s\text{-mul-ext } W \ S$ 
    using assms by (intro s-mul-ext-union-compat) (auto dest: s-mul-ext-bottom)
  then show ?thesis by simp

```

qed

**lemma** *s-mul-ext-ne-extend-left*:

**assumes**  $A \neq \{\#\}$  **and**  $(B, C) \in ns\text{-mul-ext } W S$

**shows**  $(A + B, C) \in s\text{-mul-ext } W S$

**using** *assms*

**proof** –

**have**  $(A + B, \{\#\} + C) \in s\text{-mul-ext } W S$

**using** *assms* **by** (*intro s-ns-mul-ext-union-compat*)

(*auto simp: s-mul-ext-bottom dest: s-ns-mul-ext*)

**then show** *?thesis* **by** (*simp add: ac-simps*)

qed

**lemma** *s-mul-ext-extend-left*:

**assumes**  $(B, C) \in s\text{-mul-ext } W S$

**shows**  $(A + B, C) \in s\text{-mul-ext } W S$

**using** *assms*

**proof** –

**have**  $(B + A, C + \{\#\}) \in s\text{-mul-ext } W S$

**using** *assms* **by** (*intro s-ns-mul-ext-union-compat*)

(*auto simp: ns-mul-ext-bottom dest: s-ns-mul-ext*)

**then show** *?thesis* **by** (*simp add: ac-simps*)

qed

**lemma** *mul-ext-mono*:

**assumes**  $\bigwedge x y. \llbracket x \in \text{set } xs; y \in \text{set } ys; \text{fst } (P x y) \rrbracket \implies \text{fst } (P' x y)$

**and**  $\bigwedge x y. \llbracket x \in \text{set } xs; y \in \text{set } ys; \text{snd } (P x y) \rrbracket \implies \text{snd } (P' x y)$

**shows**

$\text{fst } (mul\text{-ext } P xs ys) \implies \text{fst } (mul\text{-ext } P' xs ys)$

$\text{snd } (mul\text{-ext } P xs ys) \implies \text{snd } (mul\text{-ext } P' xs ys)$

**unfolding** *mul-ext-def Let-def fst-conv snd-conv*

**proof** –

**assume** *mem*:  $(mset xs, mset ys) \in s\text{-mul-ext } \{(x, y). \text{snd } (P x y)\} \{(x, y). \text{fst } (P x y)\}$

**show**  $(mset xs, mset ys) \in s\text{-mul-ext } \{(x, y). \text{snd } (P' x y)\} \{(x, y). \text{fst } (P' x y)\}$

**by** (*rule s-mul-ext-local-mono[OF - - mem], insert assms, auto*)

**next**

**assume** *mem*:  $(mset xs, mset ys) \in ns\text{-mul-ext } \{(x, y). \text{snd } (P x y)\} \{(x, y). \text{fst } (P x y)\}$

**show**  $(mset xs, mset ys) \in ns\text{-mul-ext } \{(x, y). \text{snd } (P' x y)\} \{(x, y). \text{fst } (P' x y)\}$

**by** (*rule ns-mul-ext-local-mono[OF - - mem], insert assms, auto*)

qed

## 4.5 Special case: non-strict order is equality

**lemma** *ns-mul-ext-IdE*:

**assumes**  $(M, N) \in ns\text{-mul-ext } Id R$

**obtains**  $X$  **and**  $Y$  **and**  $Z$  **where**  $M = X + Z$  **and**  $N = Y + Z$

**and**  $\forall y \in \text{set-mset } Y. \exists x \in \text{set-mset } X. (x, y) \in R$   
**using** *assms*  
**by** (*auto simp: ns-mul-ext-def elim!: mult2-alt-nsE*) (*insert union-commute, blast*)

**lemma** *ns-mul-ext-IdI*:

**assumes**  $M = X + Z$  **and**  $N = Y + Z$  **and**  $\forall y \in \text{set-mset } Y. \exists x \in \text{set-mset } X. (x, y) \in R$   
**shows**  $(M, N) \in \text{ns-mul-ext Id } R$   
**using** *assms mult2-alt-nsI*[*of N Z Y M Z X Id R<sup>-1</sup>*]  
**by** (*auto simp: ns-mul-ext-def*)

**lemma** *s-mul-ext-IdE*:

**assumes**  $(M, N) \in \text{s-mul-ext Id } R$   
**obtains**  $X$  **and**  $Y$  **and**  $Z$  **where**  $X \neq \{\#\}$  **and**  $M = X + Z$  **and**  $N = Y + Z$   
**and**  $\forall y \in \text{set-mset } Y. \exists x \in \text{set-mset } X. (x, y) \in R$   
**using** *assms*  
**by** (*auto simp: s-mul-ext-def elim!: mult2-alt-sE*) (*metis union-commute*)

**lemma** *s-mul-ext-IdI*:

**assumes**  $X \neq \{\#\}$  **and**  $M = X + Z$  **and**  $N = Y + Z$   
**and**  $\forall y \in \text{set-mset } Y. \exists x \in \text{set-mset } X. (x, y) \in R$   
**shows**  $(M, N) \in \text{s-mul-ext Id } R$   
**using** *assms mult2-alt-sI*[*of N Z Y M Z X Id R<sup>-1</sup>*]  
**by** (*auto simp: s-mul-ext-def ac-simps*)

**lemma** *mult-s-mul-ext-conv*:

**assumes** *trans*  $R$   
**shows**  $(\text{mult } (R^{-1}))^{-1} = \text{s-mul-ext Id } R$   
**using** *mult2-s-eq-mult2-s-alt*[*of Id R<sup>-1</sup>*] *assms*  
**by** (*auto simp: s-mul-ext-def refl-Id mult2-s-def*)

**lemma** *ns-mul-ext-Id-eq*:

$\text{ns-mul-ext Id } R = (\text{s-mul-ext Id } R)^{=}$   
**by** (*auto simp add: ns-mul-ext-def s-mul-ext-def mult2-alt-ns-conv*)

**lemma** *subsetq-mset-imp-ns-mul-ext-Id*:

**assumes**  $A \subseteq_{\#} B$   
**shows**  $(B, A) \in \text{ns-mul-ext Id } R$

**proof** –

**obtain**  $C$  **where** [*simp*]:  $B = C + A$  **using** *assms* **by** (*auto simp: mset-subset-eq-exists-conv ac-simps*)

**have**  $(C + A, \{\#\} + A) \in \text{ns-mul-ext Id } R$

**by** (*intro ns-mul-ext-IdI* [*of - C A - \{\#\}*]) *auto*

**then show** *?thesis* **by** *simp*

**qed**

**lemma** *subset-mset-imp-s-mul-ext-Id*:

**assumes**  $A \subset_{\#} B$   
**shows**  $(B, A) \in \text{s-mul-ext Id } R$

using *assms* by (intro *supset-imp-s-mul-ext*) (auto simp: *refl-Id*)

end

## 4.6 Executable version

theory *Multiset-Extension2-Impl*

imports

*HOL-Library.DAList-Multiset*

*List-Order*

*Multiset-Extension2*

*Multiset-Extension-Pair-Impl*

begin

lemma *mul-ext-list-ext*:  $\exists s ns. \text{list-order-extension-impl } s ns \text{ mul-ext}$

proof(intro *exI*)

let  $?s = \lambda s ns. \{(a,b). (mset\ as, mset\ bs) \in s\text{-mul-ext}\ ns\ s\}$

let  $?ns = \lambda s ns. \{(a,b). (mset\ as, mset\ bs) \in ns\text{-mul-ext}\ ns\ s\}$

let  $?m = mset$

show *list-order-extension-impl*  $?s\ ?ns\ \text{mul-ext}$

proof

fix  $s\ ns$

show  $?s\ \{(a,b). s\ a\ b\}\ \{(a,b). ns\ a\ b\} = \{(a,b). fst\ (mul\text{-ext}\ (\lambda\ a\ b. (s\ a\ b,\ ns\ a\ b))\ as\ bs)\}$

unfolding *mul-ext-def* *Let-def* by auto

next

fix  $s\ ns$

show  $?ns\ \{(a,b). s\ a\ b\}\ \{(a,b). ns\ a\ b\} = \{(a,b). snd\ (mul\text{-ext}\ (\lambda\ a\ b. (s\ a\ b,\ ns\ a\ b))\ as\ bs)\}$

unfolding *mul-ext-def* *Let-def* by auto

next

fix  $s\ ns\ s'\ ns'\ as\ bs$

assume  $set\ as \times set\ bs \cap ns \subseteq ns'$

$set\ as \times set\ bs \cap s \subseteq s'$

$(a,b) \in ?s\ s\ ns$

then show  $(a,b) \in ?s\ s'\ ns'$

using *s-mul-ext-local-mono*[of  $?m\ as\ ?m\ bs\ ns\ ns'\ s\ s'$ ]

unfolding *set-mset-mset* by auto

next

fix  $s\ ns\ s'\ ns'\ as\ bs$

assume  $set\ as \times set\ bs \cap ns \subseteq ns'$

$set\ as \times set\ bs \cap s \subseteq s'$

$(a,b) \in ?ns\ s\ ns$

then show  $(a,b) \in ?ns\ s'\ ns'$

using *ns-mul-ext-local-mono*[of  $?m\ as\ ?m\ bs\ ns\ ns'\ s\ s'$ ]

unfolding *set-mset-mset* by auto

**qed**  
**qed**

**context fixes**  $sns :: 'a \Rightarrow 'a \Rightarrow bool \times bool$   
**begin**

**fun**  $mul-ext-impl :: 'a list \Rightarrow 'a list \Rightarrow bool \times bool$   
**and**  $mul-ex-dom :: 'a list \Rightarrow 'a list \Rightarrow 'a \Rightarrow 'a list \Rightarrow bool \times bool$   
**where**

$mul-ext-impl [] [] = (False, True)$   
 $| mul-ext-impl [] ys = (False, False)$   
 $| mul-ext-impl xs [] = (True, True)$   
 $| mul-ext-impl xs (y \# ys) = mul-ex-dom xs [] y ys$   
  
 $| mul-ex-dom [] xs' y ys = (False, False)$   
 $| mul-ex-dom (x \# xs) xs' y ys =$   
     $(case\ sns\ x\ y\ of$   
       $(True, -) \Rightarrow if\ snd\ (mul-ext-impl\ (xs\ @\ xs')\ (filter\ (\lambda y. \neg fst\ (sns\ x\ y))\ ys))$   
       $then\ (True, True)$   
       $else\ mul-ex-dom\ xs\ (x \# xs')\ y\ ys$   
     $| (False, True) \Rightarrow or2\ (mul-ext-impl\ (xs\ @\ xs')\ ys)\ (mul-ex-dom\ xs\ (x \# xs')\ y$   
     $ys)$   
     $| - \Rightarrow mul-ex-dom\ xs\ (x \# xs')\ y\ ys)$

**end**

**context**  
**begin**

**lemma**  $mul-ext-impl-sound0$ :

$mul-ext-impl\ sns\ xs\ ys = mult2-impl\ (\lambda x\ y. sns\ y\ x)\ ys\ xs$

$mul-ex-dom\ sns\ xs\ xs'\ y\ ys = mult2-ex-dom\ (\lambda x\ y. sns\ y\ x)\ y\ ys\ xs\ xs'$

**by** ( $induct\ xs\ ys$  **and**  $xs\ xs'\ y\ ys$  *taking: sns rule: mul-ext-impl-mul-ex-dom.induct*)  
*(auto split: prod.splits bool.splits)*

**private definition**  $cond1$  **where**

$cond1\ f\ bs\ y\ xs\ ys \equiv$   
 $((\exists b. b \in set\ bs \wedge fst\ (f\ b\ y) \wedge snd\ (mul-ext\ f\ (remove1\ b\ xs)\ [y \leftarrow ys . \neg fst\ (f\ b$   
 $y])))$   
 $\vee (\exists b. b \in set\ bs \wedge snd\ (f\ b\ y) \wedge fst\ (mul-ext\ f\ (remove1\ b\ xs)\ ys))$

**private lemma**  $cond1-propagate$ :

**assumes**  $cond1\ f\ bs\ y\ xs\ ys$

**shows**  $cond1\ f\ (b \# bs)\ y\ xs\ ys$

**using**  $assms$  **unfolding**  $cond1-def$  **by**  $auto$

**private definition**  $cond2$  **where**

$cond2\ f\ bs\ y\ xs\ ys \equiv (cond1\ f\ bs\ y\ xs\ ys$

$\vee (\exists b. b \in set\ bs \wedge snd\ (f\ b\ y) \wedge snd\ (mul-ext\ f\ (remove1\ b\ xs)\ ys))$

```

private lemma cond2-propagate:
  assumes cond2 f bs y xs ys
  shows cond2 f (b # bs) y xs ys
using assms and cond1-propagate[of f bs y xs ys]
unfolding cond2-def by auto

private lemma cond1-cond2:
  assumes cond1 f bs y xs ys
  shows cond2 f bs y xs ys
using assms unfolding cond2-def by simp

lemma mul-ext-impl-sound:
  shows mul-ext-impl f xs ys = mul-ext f xs ys
unfolding mul-ext-def s-mul-ext-def ns-mul-ext-def
by (auto simp: Let-def converse-def mul-ext-impl-sound0 mult2-impl-sound)

lemma mul-ext-code [code]: mul-ext = mul-ext-impl
  by (intro ext, unfold mul-ext-impl-sound, auto)

lemma mul-ext-impl-cong[fundef-cong]:
  assumes  $\bigwedge x x'. x \in \text{set } xs \implies x' \in \text{set } ys \implies f x x' = g x x'$ 
  shows mul-ext-impl f xs ys = mul-ext-impl g xs ys
using assms
  stri-mul-ext-map[of xs ys g f id] nstri-mul-ext-map[of xs ys g f id]
  stri-mul-ext-map[of xs ys f g id] nstri-mul-ext-map[of xs ys f g id]
  by (auto simp: mul-ext-impl-sound mul-ext-def Let-def)
end

fun ass-list-to-single-list :: ('a × nat) list ⇒ 'a list
  where
    ass-list-to-single-list [] = []
    | ass-list-to-single-list ((x, n) # xs) = replicate n x @ ass-list-to-single-list xs

lemma set-ass-list-to-single-list [simp]:
  set (ass-list-to-single-list xs) = {x. ∃ n. (x, n) ∈ set xs ∧ n > 0}
  by (induct xs rule: ass-list-to-single-list.induct) auto

lemma count-mset-replicate [simp]:
  count (mset (replicate n x)) x = n
  by (induct n) (auto)

lemma count-mset-lal-ge:
   $(x, n) \in \text{set } xs \implies \text{count (mset (ass-list-to-single-list xs)) } x \geq n$ 
  by (induct xs) auto

lemma count-of-count-mset-lal [simp]:
  distinct (map fst y) ⟹ count-of y x = count (mset (ass-list-to-single-list y)) x
  by (induct y) (auto simp: count-mset-lal-ge count-of-empty)

```

**lemma** *Bag-mset*:  $Bag\ xs = mset\ (ass\text{-}list\text{-}to\text{-}single\text{-}list\ (DAList.impl\text{-}of\ xs))$   
**by** (*intro multiset-eqI, induct xs*) (*auto simp: Alist-inverse*)

**lemma** *Bag-Alist-Cons*:

$x \notin fst\ 'set\ xs \implies distinct\ (map\ fst\ xs) \implies$   
 $Bag\ (Alist\ ((x, n) \# xs)) = mset\ (replicate\ n\ x) + Bag\ (Alist\ xs)$   
**by** (*induct xs*) (*auto simp: Bag-mset Alist-inverse*)

**lemma** *mset-lal* [*simp*]:

$distinct\ (map\ fst\ xs) \implies mset\ (ass\text{-}list\text{-}to\text{-}single\text{-}list\ xs) = Bag\ (Alist\ xs)$   
**apply** (*induct xs*) **apply** (*auto simp: Bag-Alist-Cons*)  
**apply** (*simp add: Mempty-Bag empty.abs-eq*)  
**done**

**lemma** *Bag-s-mul-ext*:

$(Bag\ xs, Bag\ ys) \in s\text{-}mul\text{-}ext\ \{(x, y). snd\ (f\ x\ y)\} \{(x, y). fst\ (f\ x\ y)\} \longleftrightarrow$   
 $fst\ (mul\text{-}ext\ f\ (ass\text{-}list\text{-}to\text{-}single\text{-}list\ (DAList.impl\text{-}of\ xs))\ (ass\text{-}list\text{-}to\text{-}single\text{-}list$   
 $(DAList.impl\text{-}of\ ys)))$   
**by** (*auto simp: mul-ext-def Let-def Alist-impl-of*)

**lemma** *Bag-ns-mul-ext*:

$(Bag\ xs, Bag\ ys) \in ns\text{-}mul\text{-}ext\ \{(x, y). snd\ (f\ x\ y)\} \{(x, y). fst\ (f\ x\ y)\} \longleftrightarrow$   
 $snd\ (mul\text{-}ext\ f\ (ass\text{-}list\text{-}to\text{-}single\text{-}list\ (DAList.impl\text{-}of\ xs))\ (ass\text{-}list\text{-}to\text{-}single\text{-}list$   
 $(DAList.impl\text{-}of\ ys)))$   
**by** (*auto simp: mul-ext-def Let-def Alist-impl-of*)

**lemma** *smulextp-code*[*code*]:

$smulextp\ f\ (Bag\ xs)\ (Bag\ ys) \longleftrightarrow fst\ (mul\text{-}ext\ f\ (ass\text{-}list\text{-}to\text{-}single\text{-}list\ (DAList.impl\text{-}of$   
 $xs))\ (ass\text{-}list\text{-}to\text{-}single\text{-}list\ (DAList.impl\text{-}of\ ys)))$   
**unfolding** *smulextp-def Bag-s-mul-ext ..*

**lemma** *nsmulextp-code*[*code*]:

$nsmulextp\ f\ (Bag\ xs)\ (Bag\ ys) \longleftrightarrow snd\ (mul\text{-}ext\ f\ (ass\text{-}list\text{-}to\text{-}single\text{-}list\ (DAList.impl\text{-}of$   
 $xs))\ (ass\text{-}list\text{-}to\text{-}single\text{-}list\ (DAList.impl\text{-}of\ ys)))$   
**unfolding** *nsmulextp-def Bag-ns-mul-ext ..*

**lemma** *mulextp-code*[*code*]:

$mulextp\ f\ (Bag\ xs)\ (Bag\ ys) = mul\text{-}ext\ f\ (ass\text{-}list\text{-}to\text{-}single\text{-}list\ (DAList.impl\text{-}of$   
 $xs))\ (ass\text{-}list\text{-}to\text{-}single\text{-}list\ (DAList.impl\text{-}of\ ys))$   
**unfolding** *mulextp-def* **by** (*simp add: nsmulextp-code smulextp-code*)

**end**

## 5 The Weighted Path Order

This is a version of WPO that also permits multiset comparisons of lists of terms. It therefore generalizes RPO.

**theory** *WPO*

```

imports
  Knuth-Bendix-Order.Lexicographic-Extension
  First-Order-Terms.Subterm-and-Context
  Knuth-Bendix-Order.Order-Pair
  Polynomial-Factorization.Missing-List
  Status
  Precedence
  Multiset-Extension2
  HOL.Zorn
begin

datatype order-tag = Lex | Mul

locale wpo =
  fixes n :: nat
  and S NS :: ('f, 'v) term rel
  and prc :: ('f × nat ⇒ 'f × nat ⇒ bool × bool)
  and prl :: 'f × nat ⇒ bool
  and σσ :: 'f status
  and c :: 'f × nat ⇒ order-tag
  and ssimple :: bool
  and large :: 'f × nat ⇒ bool
begin

fun wpo :: ('f, 'v) term ⇒ ('f, 'v) term ⇒ bool × bool
  where
    wpo s t = (if (s,t) ∈ S then (True, True) else
      if (s,t) ∈ NS then (case s of
        Var x ⇒ (False,
          (case t of
            Var y ⇒ x = y
          | Fun g ts ⇒ status σσ (g, length ts) = [] ∧ prl (g, length ts)))
        | Fun f ss ⇒
          if ∃ i ∈ set (status σσ (f, length ss)). snd (wpo (ss ! i) t) then (True, True)
          else
            (case t of
              Var - ⇒ (False, ssimple ∧ large (f, length ss))
            | Fun g ts ⇒
              (case prc (f, length ss) (g, length ts) of (prs, prns) ⇒
                if prns ∧ (∀ j ∈ set (status σσ (g, length ts)). fst (wpo s (ts ! j))) then
                  if prs then (True, True)
                  else let ss' = map (λ i. ss ! i) (status σσ (f, length ss));
                      ts' = map (λ i. ts ! i) (status σσ (g, length ts));
                      cf = c (f, length ss);
                      cg = c (g, length ts)
                      in if cf = Lex ∧ cg = Lex
                        then lex-ext wpo n ss' ts'
                      else if cf = Mul ∧ cg = Mul
                        then mul-ext wpo ss' ts'

```

```

else (length ss' ≠ 0 ∧ length ts' = 0, length ts' = 0)
  else (False, False)))
else (False, False))

```

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

**abbreviation** *wpo-s* (**infix**  $\succ$  50) **where**  $s \succ t \equiv \text{fst } (wpo \ s \ t)$

**abbreviation** *wpo-ns* (**infix**  $\succeq$  50) **where**  $s \succeq t \equiv \text{snd } (wpo \ s \ t)$

**abbreviation** *WPO-S*  $\equiv \{(s,t). s \succ t\}$

**abbreviation** *WPO-NS*  $\equiv \{(s,t). s \succeq t\}$

**lemma** *wpo-s-imp-ns*:  $s \succ t \implies s \succeq t$

**using** *lex-ext-stri-imp-nstri*

**unfolding** *wpo.simps*[*of s t*]

**by** (*auto simp: Let-def mul-ext-stri-imp-nstri split: term.splits if-splits prod.splits*)

**lemma** *S-imp-wpo-s*:  $(s,t) \in S \implies s \succ t$  **by** (*simp add: wpo.simps*)

**end**

**declare** *wpo.wpo.simps*[*code*]

**definition** *strictly-simple-status* ::  $'f \ \text{status} \Rightarrow ('f, 'v) \ \text{term \ rel} \Rightarrow \text{bool}$  **where**

*strictly-simple-status*  $\sigma \ \text{rel} =$

$(\forall f \ ts \ i. i \in \text{set } (\text{status } \sigma \ (f, \text{length } ts)) \longrightarrow (\text{Fun } f \ ts, ts \ ! \ i) \in \text{rel})$

**definition** *trans-precedence* **where** *trans-precedence*  $\text{prc} = (\forall f \ g \ h.$

$(\text{fst } (\text{prc } f \ g) \longrightarrow \text{snd } (\text{prc } g \ h) \longrightarrow \text{fst } (\text{prc } f \ h)) \wedge$

$(\text{snd } (\text{prc } f \ g) \longrightarrow \text{fst } (\text{prc } g \ h) \longrightarrow \text{fst } (\text{prc } f \ h)) \wedge$

$(\text{snd } (\text{prc } f \ g) \longrightarrow \text{snd } (\text{prc } g \ h) \longrightarrow \text{snd } (\text{prc } f \ h)))$

**locale** *wpo-with-basic-assms* = *wpo* +

*order-pair* + *irrefl-precedence* +

**constrains** *S* ::  $('f, 'v) \ \text{term \ rel}$  **and** *NS* :: -

**and** *prc* ::  $'f \times \text{nat} \Rightarrow 'f \times \text{nat} \Rightarrow \text{bool} \times \text{bool}$

**and** *prl* ::  $'f \times \text{nat} \Rightarrow \text{bool}$

**and** *ssimple* ::  $\text{bool}$

**and** *large* ::  $'f \times \text{nat} \Rightarrow \text{bool}$

**and** *c* ::  $'f \times \text{nat} \Rightarrow \text{order-tag}$

**and** *n* ::  $\text{nat}$

**and**  $\sigma \sigma$  ::  $'f \ \text{status}$

**assumes** *subst-S*:  $(s,t) \in S \implies (s \cdot \sigma, t \cdot \sigma) \in S$

**and** *subst-NS*:  $(s,t) \in NS \implies (s \cdot \sigma, t \cdot \sigma) \in NS$

**and** *irrefl-S*: *irrefl S*

**and** *S-imp-NS*:  $S \subseteq NS$

**and** *ss-status*:  $ssimple \implies i \in set (status \ \sigma \ \sigma \ fn) \implies simple\text{-arg-pos } S \ fn \ i$   
**and** *large*:  $ssimple \implies large \ fn \implies fst (prc \ fn \ gm) \vee snd (prc \ fn \ gm) \wedge status$   
 $\sigma \ \sigma \ gm = []$   
**and** *large-trans*:  $ssimple \implies large \ fn \implies snd (prc \ gm \ fn) \implies large \ gm$   
**and** *ss-S-non-empty*:  $ssimple \implies S \neq \{\}$   
**begin**  
**abbreviation**  $\sigma \equiv status \ \sigma \ \sigma$

**lemma** *ss-NS-not-UNIV*:  $ssimple \implies NS \neq UNIV$

**proof**

**assume**  $ssimple \ NS = UNIV$

**with** *ss-S-non-empty* **obtain**  $a \ b$  **where**  $(a,b) \in S \ (b,a) \in NS$  **by** *auto*

**from** *compat-S-NS-point*[*OF this*] **have**  $(a,a) \in S$ .

**with** *irrefl-S* **show** *False* **unfolding** *irrefl-def* **by** *auto*

**qed**

**lemmas**  $\sigma = status[of \ \sigma \ \sigma]$

**lemma**  $\sigma E$ :  $i \in set (\sigma (f, length \ ss)) \implies ss ! i \in set \ ss$  **by** (*rule status-aux*)

**lemma** *wpo-ns-imp-NS*:  $s \succeq t \implies (s,t) \in NS$

**using** *S-imp-NS*

**by** (*cases s*, *auto simp: wpo.simps[of - t]*, *cases t*,

*auto simp: refl-NS-point split: if-splits*)

**lemma** *wpo-s-imp-NS*:  $s \succ t \implies (s,t) \in NS$

**by** (*rule wpo-ns-imp-NS*[*OF wpo-s-imp-ns*])

**lemma** *wpo-least-1*: **assumes**  $prl (f, length \ ss)$

**and**  $(t, Fun \ f \ ss) \in NS$

**and**  $\sigma (f, length \ ss) = []$

**shows**  $t \succeq Fun \ f \ ss$

**proof** (*cases t*)

**case** (*Var x*)

**with** *assms* **show** *?thesis* **by** (*simp add: wpo.simps*)

**next**

**case** (*Fun g ts*)

**let**  $?f = (f, length \ ss)$

**let**  $?g = (g, length \ ts)$

**obtain**  $s \ ns$  **where**  $prc \ ?g \ ?f = (s, ns)$  **by** *force*

**with**  $prl[OF \ assms(1), of \ ?g]$  **have**  $prc: prc \ ?g \ ?f = (s, True)$  **by** *auto*

**show** *?thesis* **using** *assms(2)*

**unfolding** *Fun*

**unfolding**  $wpo.simps[of \ Fun \ g \ ts \ Fun \ f \ ss]$  *term.simps assms(3)*

**by** (*auto simp: prc lex-ext-least-1 mul-ext-def ns-mul-ext-bottom Let-def*)

**qed**

**lemma** *wpo-least-2*: **assumes**  $prl (f, length \ ss)$  (**is**  $prl \ ?f$ )

**and**  $(Fun \ f \ ss, t) \notin S$

**and**  $\sigma (f, length \ ss) = []$

```

shows  $\neg \text{Fun } f \text{ ss} \succ t$ 
proof (cases t)
  case (Var x)
    with Var show ?thesis using assms(2-3) by (auto simp: wpo.simps split:
if-splits)
  next
    case (Fun g ts)
    let ?g = (g,length ts)
    obtain s ns where prc ?f ?g = (s,ns) by force
    with prl2[OF assms(1), of ?g] have prc: prc ?f ?g = (False,ns) by auto
    show ?thesis using assms(2) assms(3) unfolding Fun
      by (simp add: wpo.simps[of - Fun g ts] lex-ext-least-2 prc
mul-ext-def s-mul-ext-bottom-strict Let-def)
qed

```

```

lemma wpo-least-3: assumes prl (f,length ss) (is prl ?f)
  and ns: Fun f ss  $\succeq t$ 
  and NS: (u, Fun f ss)  $\in NS$ 
  and ss:  $\sigma$  (f,length ss) = []
  and S:  $\bigwedge x. (\text{Fun } f \text{ ss}, x) \notin S$ 
  and u: u = Var x
shows u  $\succeq t$ 
proof (cases (Fun f ss, t)  $\in S \vee (u, \text{Fun } f \text{ ss}) \in S \vee (u, t) \in S$ )
  case True
    with wpo-ns-imp-NS[OF ns] NS compat-NS-S-point compat-S-NS-point have (u,
t)  $\in S$  by blast
    from wpo-s-imp-ns[OF S-imp-wpo-s[OF this]] show ?thesis .
  next
    case False
    from trans-NS-point[OF NS wpo-ns-imp-NS[OF ns]] have utA: (u, t)  $\in NS$  .
    show ?thesis
    proof (cases t)
      case t: (Var y)
        with ns False ss have *: ssimple large (f,length ss)
          by (auto simp: wpo.simps split: if-splits)
        show ?thesis
        proof (cases x = y)
          case True
            thus ?thesis using ns * False utA ss
              unfolding wpo.simps[of u t] wpo.simps[of Fun f ss t]
              unfolding t u term.simps
              by (auto split: if-splits)
          next
            case False
            from utA[unfolded t u]
            have (Var x, Var y)  $\in NS$  .
            from False subst-NS[OF this, of  $\lambda z. \text{if } z = x \text{ then } v \text{ else } w$  for v w]
            have (v,w)  $\in NS$  for v w by auto
            hence NS = UNIV by auto

```

```

    with ss-NS-not-UNIV[OF ‹ssimple›]
    have False by auto
    thus ?thesis ..
  qed
next
  case (Fun g ts)
  let ?g = (g,length ts)
  obtain s ns where prc ?f ?g = (s,ns) by force
  with prl2[OF ‹prl ?f›, of ?g] have prc: prc ?f ?g = (False,ns) by auto
  show ?thesis
  proof (cases ‹σ ?g›)
    case Nil
    with False Fun assms prc have prc ?f ?g = (False,True)
    by (auto simp: wpo.simps split: if-splits)
    with prl3[OF ‹prl ?f›, of ?g] have prl ?g by auto
    show ?thesis using utA unfolding Fun by (rule wpo-least-1[OF ‹prl ?g›],
simp add: Nil)
  next
  case (Cons t1 tts)
  have ¬ wpo-s (Fun f ss) (ts ! t1) by (rule wpo-least-2[OF ‹prl ?f› S ss])
  with ‹wpo-ns (Fun f ss) t› False Fun Cons
  have False by (simp add: ss wpo.simps split: if-splits)
  then show ?thesis ..
  qed
qed
qed

```

```

lemma wpo-compat: (s ≥ t ∧ t > u → s > u) ∧
(s > t ∧ t ≥ u → s > u) ∧
(s ≥ t ∧ t ≥ u → s ≥ u) (is ?tran s t u)
proof (induct (s,t,u) arbitrary: s t u rule: wf-induct[OF wf-measures[of [λ (s,t,u).
size s, λ (s,t,u). size t, λ (s,t,u). size u]]])
  case 1
  note ind = 1[simplified]
  show ?tran s t u
  proof (cases (s,t) ∈ S ∨ (t,u) ∈ S ∨ (s,u) ∈ S)
    case True
    {
      assume st: wpo-ns s t and tu: wpo-ns t u
      from wpo-ns-imp-NS[OF st] wpo-ns-imp-NS[OF tu]
      True compat-NS-S-point compat-S-NS-point have (s,u) ∈ S by blast
      from S-imp-wpo-s[OF this] have wpo-s s u .
    }
  with wpo-s-imp-ns show ?thesis by blast
  next
  case False
  then have stS: (s,t) ∉ S and tuS: (t,u) ∉ S and suS: (s,u) ∉ S by auto
  show ?thesis

```

```

proof (cases t)
  note [simp] = wpo.simps[of s u] wpo.simps[of s t] wpo.simps[of t u]
  case (Var x)
  note wpo.simps[simp]
  show ?thesis
  proof safe
    assume wpo-s t u
    with Var tuS show wpo-s s u by (auto split: if-splits)
  next
    assume gr: wpo-s s t and ge: wpo-ns t u
    from wpo-s-imp-NS[OF gr] have stA: (s,t) ∈ NS .
    from wpo-ns-imp-NS[OF ge] have tuA: (t,u) ∈ NS .
    from trans-NS-point[OF stA tuA] have suA: (s,u) ∈ NS .
    show wpo-s s u
    proof (cases u)
      case (Var y)
        with ge ⟨t = Var x⟩ tuS have t = u by (auto split: if-splits)
        with gr show ?thesis by auto
      next
        case (Fun h us)
        let ?h = (h,length us)
        from Fun ge Var tuS have us: σ ?h = [] and pri: prl ?h by (auto split:
if-splits)
        from gr Var tuS ge stS obtain f ss where s: s = Fun f ss by (cases s,
auto split: if-splits)
        let ?f = (f,length ss)
        from s gr Var False obtain i where i: i ∈ set (σ ?f) and sit: ss ! i ≥ t
by (auto split: if-splits)
        from trans-NS-point[OF wpo-ns-imp-NS[OF sit] tuA] have siu: (ss ! i,u)
∈ NS .
        from wpo-least-1[OF pri siu[unfolded Fun us] us]
        have ss ! i ≥ u unfolding Fun us .
        with i have ∃ i ∈ set (σ ?f). ss ! i ≥ u by blast
        with s suA show ?thesis by simp
      qed
    next
      assume ge1: wpo-ns s t and ge2: wpo-ns t u
      show wpo-ns s u
      proof (cases u)
        case (Var y)
          with ge2 ⟨t = Var x⟩ tuS have t = u by (auto split: if-splits)
          with ge1 show ?thesis by auto
        next
          case (Fun h us)
          let ?h = (h,length us)
          from Fun ge2 Var tuS have us: σ ?h = [] and pri: prl ?h by (auto split:
if-splits)
          show ?thesis unfolding Fun us
          by (rule wpo-least-1[OF pri trans-NS-point[OF wpo-ns-imp-NS[OF ge1]

```

```

      wpo-ns-imp-NS[OF ge2[unfolded Fun us]] us)
    qed
  qed
next
  case (Fun g ts)
  let ?g = (g,length ts)
  let ?ts = set (σ ?g)
  let ?t = Fun g ts
  from Fun have t: t = ?t .
  show ?thesis
  proof (cases s)
    case (Var x)
    show ?thesis
    proof safe
      assume gr: wpo-s s t
      with Var Fun stS show wpo-s s u by (auto simp: wpo.simps split: if-splits)
    next
      assume ge: wpo-ns s t and gr: wpo-s t u
      with Var Fun stS have pri: prl ?g and σ ?g = [] by (auto simp: wpo.simps
split: if-splits)
      with gr Fun show wpo-s s u using wpo-least-2[OF pri, of u] False by auto
    next
      assume ge1: wpo-ns s t and ge2: wpo-ns t u
      with Var Fun stS have pri: prl ?g and empty: σ ?g = [] by (auto simp:
wpo.simps split: if-splits)
      from wpo-ns-imp-NS[OF ge1] Var Fun empty have ns: (Var x, Fun g ts)
∈ NS by simp
      from wpo-ns-imp-NS[OF ge1] wpo-ns-imp-NS[OF ge2]
      have suA: (s,u) ∈ NS by (rule trans-NS-point)
      note wpo-simp = wpo.simps[of t u]
      show wpo-ns s u
      proof (cases u)
        case (Fun h us)
        let ?h = (h,length us)
        obtain pns where prc: prc ?g ?h = (False,pns) using prl2[OF pri, of
?h] by (cases prc ?g ?h, auto)
        from prc wpo-ns-imp-NS[OF ge2] tuS ge2 Fun u empty have pns
unfolding wpo-simp
          by (auto split: if-splits simp: Let-def)
        with prc have prc: prc ?g ?h = (False, True) by auto
        from prl3[OF pri, of ?h] prc have pri': prl ?h by auto
        from prc wpo-ns-imp-NS[OF ge2] tuS ge2 Fun u empty have empty': σ
?h = [] unfolding wpo-simp
          by (auto split: if-splits simp: Let-def dest: lex-ext-arg-empty mul-ext-arg-empty)
        from pri' empty' suA show ?thesis unfolding Var u by (auto simp:
wpo.simps)
      next
        case u: (Var z)
        from wpo-ns-imp-NS[OF ge2] tuS ge2 Fun u empty wpo-simp

```

```

have ssimple large ?g by auto
show ?thesis
proof (cases  $x = z$ )
  case True
  thus ?thesis using suA Var u by (simp add: wpo.simps)
next
  case False
  from suA[unfolded Var u] have ns: (Var x, Var z) ∈ NS by auto
  have (a,b) ∈ NS for a b using subst-NS[OF ns, of  $\lambda z. \text{if } z = x \text{ then } a \text{ else } b$ ] False by auto
  hence NS = UNIV by auto
  from ss-S-non-empty[OF  $\langle \text{ssimple} \rangle$ ] this compat-S-NS obtain a where
  (a,a) ∈ S by auto
  with irrefl-S show ?thesis unfolding irrefl-def by auto
qed
qed
qed
next
  case (Fun f ss)
  let ?s = Fun f ss
  let ?f = (f,length ss)
  let ?ss = set ( $\sigma$  ?f)
  from Fun have s:  $s = ?s$  .
  let ?s1 =  $\exists i \in ?ss. ss ! i \succeq t$ 
  let ?t1 =  $\exists j \in ?ts. ts ! j \succeq u$ 
  let ?ls = length ss
  let ?lt = length ts
  obtain ps pns where prc: prc ?f ?g = (ps,pns) by force
  let ?tran2 =  $\lambda a b c.$ 
    ((wpo-ns a b) ∧ (wpo-s b c) → (wpo-s a c)) ∧
    ((wpo-s a b) ∧ (wpo-ns b c) → (wpo-s a c)) ∧
    ((wpo-ns a b) ∧ (wpo-ns b c) → (wpo-ns a c)) ∧
    ((wpo-s a b) ∧ (wpo-s b c) → (wpo-s a c))
  from s have  $\forall s' \in \text{set } ss. \text{size } s' < \text{size } s$  by (auto simp: size-simps)
  with ind have ind2:  $\bigwedge s' t' u'. \llbracket s' \in \text{set } ss \rrbracket \implies ?tran s' t' u'$  by blast
  with wpo-s-imp-ns have ind3:  $\bigwedge us s' t' u'. \llbracket s' \in \text{set } ss; t' \in \text{set } ts \rrbracket \implies$ 
?tran2 s' t' u' by blast
  let ?mss = map ( $\lambda i. ss ! i$ ) ( $\sigma$  ?f)
  let ?mts = map ( $\lambda j. ts ! j$ ) ( $\sigma$  ?g)
  have ind3':  $\bigwedge us s' t' u'. \llbracket s' \in \text{set } ?mss; t' \in \text{set } ?mts \rrbracket \implies ?tran2 s' t' u'$ 
  by (rule ind3, auto simp: status-aux)
  {
  assume ge1:  $s \succeq t$  and ge2:  $t \succ u$ 
  from wpo-ns-imp-NS[OF ge1] have stA: (s,t) ∈ NS .
  from wpo-s-imp-NS[OF ge2] have tuA: (t,u) ∈ NS .
  from trans-NS-point[OF stA tuA] have suA: (s,u) ∈ NS .
  have  $s \succ u$ 
  proof (cases ?s1)
    case True

```

```

from this obtain i where i: i ∈ ?ss and ges: ss ! i ⋮ t by auto
from  $\sigma E[OF\ i]$  have s': ss ! i ∈ set ss .
with i s s' ind2[of ss ! i t u, simplified] ges ge2 have ss ! i > u by auto
then have ss ! i ⋮ u by (rule wpo-s-imp-ns)
  with i s suA show ?thesis by (cases u, auto simp: wpo.simps split:
if-splits)
next
  case False
  show ?thesis
  proof (cases ?t1)
    case True
      from this obtain j where j: j ∈ ?ts and ges: ts ! j ⋮ u by auto
      from  $\sigma E[OF\ j]$  have t': ts ! j ∈ set ts by auto
      from j t' t stS False ge1 s have ge1': s > ts ! j unfolding wpo.simps[of
s t]
        by (auto split: if-splits prod.splits)
        from t' s t ge1' ges ind[rule-format, of s ts ! j u, simplified]
        show s > u
        using suA size-simps supt.intros unfolding wpo.simps[of s u]
        by (auto split: if-splits)
      next
        case False
        from t this ge2 tuS obtain h us where u: u = Fun h us
          by (cases u, auto simp: wpo.simps split: if-splits)
        let ?u = Fun h us
        let ?h = (h, length us)
        let ?us = set ( $\sigma$  ?h)
        let ?mus = map ( $\lambda k. us ! k$ ) ( $\sigma$  ?h)
        from s t u ge1 ge2 have ge1: ?s ⋮ ?t and ge2: ?t > ?u by auto
        from stA stS s t have stAS: ((?s, ?t) ∈ S) = False ((?s, ?t) ∈ NS) =
True by auto
        from tuA tuS t u have tuAS: ((?t, ?u) ∈ S) = False ((?t, ?u) ∈ NS) =
True by auto
        note ge1 = ge1[unfolded wpo.simps[of ?s ?t] stAS, simplified]
        note ge2 = ge2[unfolded wpo.simps[of ?t ?u] tuAS, simplified]
        obtain ps2 pns2 where prc2: prc ?g ?h = (ps2, pns2) by force
        obtain ps3 pns3 where prc3: prc ?f ?h = (ps3, pns3) by force
        from  $\langle \neg ?s1 \rangle t ge1$  have st':  $\forall j \in ?ts. ?s > ts ! j$  by (auto split:
if-splits prod.splits)
        from  $\langle \neg ?t1 \rangle t u ge2 tuS$  have tu':  $\forall k \in ?us. ?t > us ! k$  by (auto
split: if-splits prod.splits)
        from  $\langle \neg ?s1 \rangle s t ge1 stS st'$  have fg: pns by (cases ?thesis, auto simp:
prc)
        from  $\langle \neg ?t1 \rangle u ge2 tu'$  have gh: pns2 by (cases ?thesis, auto simp:
prc2)
        from  $\langle \neg ?s1 \rangle$  have ?s1 = False by simp
        note ge1 = ge1[unfolded this[unfolded t] if-False term.simps prc split]
        from  $\langle \neg ?t1 \rangle$  have ?t1 = False by simp
        note ge2 = ge2[unfolded this[unfolded u] if-False term.simps prc2 split]

```

```

note compat = prc-compat[OF prc prc2 prc3]
from fg gh compat have fh: pns3 by simp
{
  fix k
  assume k: k ∈ ?us
  from  $\sigma E$ [OF this] have  $\text{size } (us ! k) < \text{size } u$  unfolding u using
size-simps by auto
  with  $tu'$ [folded t]  $\langle s \succeq t \rangle$ 
    ind[rule-format, of s t us ! k] k have  $s \succ us ! k$  by blast
} note su' = this
show ?thesis
proof (cases ps3)
  case True
    with  $su' s u fh prc3 suA$  show ?thesis by (auto simp: wpo.simps)
  next
    case False
      from False fg gh compat have nfg: ¬ ps and ngh: ¬ ps2 and  $*$ : ps =
False ps2 = False by blast+
      note ge1 = ge1[unfolded * if-False]
      note ge2 = ge2[unfolded * if-False]
      show ?thesis
      proof (cases c ?f)
        case Mul note cf = this
          show ?thesis
          proof (cases c ?g)
            case Mul note cg = this
              show ?thesis
              proof (cases c ?h)
                case Mul note ch = this
                  from ge1[unfolded cf cg]
                  have mul1: snd (mul-ext wpo ?mss ?mts) by (auto split: if-splits)
                  from ge2[unfolded cg ch]
                  have mul2: fst (mul-ext wpo ?mts ?mus) by (auto split: if-splits)
                  from mul1 mul2 mul-ext-compat[OF ind3', of ?mss ?mts ?mus]
                  have fst (mul-ext wpo ?mss ?mus) by auto
                  with  $s u fh su' prc3 cf ch suA$  show ?thesis unfolding wpo.simps[of
s u] by simp
                next
                  case Lex note ch = this
                    from  $gh u ge2 tu' prc2 ngh cg ch$  have us-e: ?mus = [] by simp
                    from  $gh u ge2 tu' prc2 ngh cg ch$  have ts-ne: ?mts ≠ [] by (auto
split: if-splits)
                    from ns-mul-ext-bottom-uniqueness[of mset ?mts]
                    have  $\bigwedge f. \text{snd } (mul-ext f [] ?mts) \implies ?mts = []$  unfolding
mul-ext-def by (simp add: Let-def)
                    with  $ts-ne fg \langle \neg ?s1 \rangle t ge1 st' prc nfg cf cg$  have ss-ne: ?mss ≠ []
by (cases ss) auto
                    from  $us-e ss-ne s u fh su' prc3 cf cg ch suA$  show ?thesis
unfolding wpo.simps[of s u] by simp

```



```

from wpo-ns-imp-NS[OF ge2] have tuA: (t,u) ∈ NS .
from trans-NS-point[OF stA tuA] have suA: (s,u) ∈ NS .
have s > u
proof (cases ?s1)
  case True
    from True obtain i where i: i ∈ ?ss and ges: ss ! i ≥ t by auto
    from σE[OF i] have s': ss ! i ∈ set ss by auto
    with s s' ind2[of ss ! i t u, simplified] ges ge2 have ss ! i ≥ u by auto
    with i s' s suA show ?thesis by (cases u, auto simp: wpo.simps split:
if-splits)
  next
    case False
    show ?thesis
    proof (cases ?t1)
      case True
        from this obtain j where j: j ∈ ?ts and ges: ts ! j ≥ u by auto
        from σE[OF j] have t': ts ! j ∈ set ts .
        from j t' t stS False ge1 s have ge1': s > ts ! j unfolding wpo.simps[of
s t]
          by (auto split: if-splits prod.splits)
        from t' s t ge1' ges ind[rule-format, of s ts ! j u, simplified]
        show s > u
          using suA size-simps supt.intros unfolding wpo.simps[of s u]
          by (auto split: if-splits)
      next
        case False
        show ?thesis
        proof (cases u)
          case u: (Fun h us)
            let ?u = Fun h us
            let ?h = (h,length us)
            let ?us = set (σ ?h)
            let ?mss = map (λ i. ss ! i) (σ ?f)
            let ?mts = map (λ j. ts ! j) (σ ?g)
            let ?mus = map (λ k. us ! k) (σ ?h)
            note σE = σE[of - f ss] σE[of - g ts] σE[of - h us]
            from s t u ge1 ge2 have ge1: ?s > ?t and ge2: ?t ≥ ?u by auto
            from stA stS s t have stAS: ((?s,?t) ∈ S) = False ((?s,?t) ∈ NS) =
True by auto
            from tuA tuS t u have tuAS: ((?t,?u) ∈ S) = False ((?t,?u) ∈ NS)
= True by auto
            note ge1 = ge1[unfolded wpo.simps[of ?s ?t] stAS, simplified]
            note ge2 = ge2[unfolded wpo.simps[of ?t ?u] tuAS, simplified]
            let ?lu = length us
            obtain ps2 pns2 where prc2: prc ?g ?h = (ps2,pns2) by force
            obtain ps3 pns3 where prc3: prc ?f ?h = (ps3,pns3) by force
            from <¬ ?s1> t ge1 have st': ∀ j ∈ ?ts. ?s > ts ! j by (auto split:
if-splits prod.splits)
            from <¬ ?t1> t u ge2 tuS have tu': ∀ k ∈ ?us. ?t > us ! k by (auto

```

```

split: if-splits prod.splits)
  from <¬ ?s1> s t ge1 stS st' have fg: pns by (cases ?thesis, auto simp:
prc)
  from <¬ ?t1> u ge2 tu' have gh: pns2 by (cases ?thesis, auto simp:
prc2)
  from <¬ ?s1> have ?s1 = False by simp
  note ge1 = ge1[unfolded this[unfolded t] if-False term.simps prc split]
  from <¬ ?t1> have ?t1 = False by simp
  note ge2 = ge2[unfolded this[unfolded u] if-False term.simps prc2 split]
  note compat = prc-compat[OF prc prc2 prc3]
  from fg gh compat have fh: pns3 by simp
  {
    fix k
    assume k: k ∈ ?us
    from  $\sigma E(3)$ [OF this] have size (us ! k) < size u unfolding u using
size-simps by auto
    with tu'[folded t] wpo-s-imp-ns[OF <s > t>]
      ind[rule-format, of s t us ! k] k have s > us ! k by blast
  } note su' = this
  show ?thesis
  proof (cases ps3)
    case True
      with su' s u fh prc3 suA show ?thesis by (auto simp: wpo.simps)
    next
      case False
        from False fg gh compat have nfg: ¬ ps and ngh: ¬ ps2 and *: ps
= False ps2 = False by blast+
        note ge1 = ge1[unfolded * if-False]
        note ge2 = ge2[unfolded * if-False]
        show ?thesis
        proof (cases c ?f)
          case Mul note cf = this
            show ?thesis
            proof (cases c ?g)
              case Mul note cg = this
                show ?thesis
                proof (cases c ?h)
                  case Mul note ch = this
                    from fg t ge1 st' nfg cf cg
                    have mul1: fst (mul-ext wpo ?mss ?mts) by auto
                    from gh u ge2 tu' ngh cg ch
                    have mul2: snd (mul-ext wpo ?mts ?mus) by auto
                    from mul1 mul2 mul-ext-compat[OF ind3', of ?mss ?mts ?mus]
                    have fst (mul-ext wpo ?mss ?mus) by auto
                    with s u fh su' prc3 cf ch suA show ?thesis unfolding
wpo.simps[of s u] by simp
                next
                  case Lex note ch = this
                    from gh u ge2 tu' ngh cg ch have us-e: ?mus = [] by simp

```

```

    from fg t ge1 st' nfg cf cg s-mul-ext-bottom-strict
    have ss-ne: ?mss ≠ [] by (cases ?mss) (auto simp: Let-def
mul-ext-def)
    from us-e ss-ne s u fh su' prc3 cf cg ch suA show ?thesis
unfolding wpo.simps[of s u] by simp
qed
next
case Lex note cg = this
from fg t ge1 st' prc nfg cf cg s-mul-ext-bottom-strict
have ss-ne: ?mss ≠ [] by (auto simp: mul-ext-def)
from fg t ge1 st' nfg cf cg have ts-e: ?mts = [] by simp
show ?thesis
proof (cases c ?h)
  case Mul note ch = this
  with gh u ge2 tu' ngh cg ch ns-mul-ext-bottom-uniqueness
  have ?mus = [] by simp
  with ss-ne s u fh su' prc3 cf cg ch s-mul-ext-bottom suA
  show ?thesis unfolding wpo.simps[of s u] by (simp add: Let-def
mul-ext-def s-mul-ext-def mult2-alt-s-def)
next
case Lex note ch = this
from lex-ext-iff[of - - [] ?mus]
have  $\bigwedge f. \text{snd } (lex\text{-ext } f \ n \ []) \ ?mus \implies \ ?mus = []$  by simp
with ts-e gh u ge2 tu' ngh cg ch
have ?mus = [] by simp
with ss-ne s u fh su' prc3 cf cg ch s-mul-ext-bottom suA
show ?thesis unfolding wpo.simps[of s u] by (simp add:
mul-ext-def)
qed
qed
next
case Lex note cf = this
show ?thesis
proof (cases c ?g)
  case Mul note cg = this
  from fg t ge1 st' nfg cf cg have ss-ne: ?mss ≠ [] by simp
  from fg t ge1 st' nfg cf cg have ts-e: ?mts = [] by simp
  show ?thesis
  proof (cases c ?h)
    case Mul note ch = this
    from ts-e gh u ge2 tu' ngh cg ch
    ns-mul-ext-bottom-uniqueness[of mset ?mus]
    have ?mus = [] by (simp add: mul-ext-def Let-def)
    with ss-ne s u fh su' prc3 cf cg ch s-mul-ext-bottom suA
    show ?thesis unfolding wpo.simps[of s u] by (simp add:
mul-ext-def)
  next
  case Lex note ch = this
  from gh u ge2 tu' prc2 ngh cg ch have ?mus = [] by simp

```

```

      with ss-ne s u fh su' prc3 cf cg ch suA
      show ?thesis unfolding wpo.simps[of s u] by (simp add:
lex-ext-iff)
    qed
  next
  case Lex note cg = this
  show ?thesis
  proof (cases c ?h)
    case Mul note ch = this
    from gh u ge2 tu' ngh cg ch have us-e: ?mus = [] by simp
    have  $\bigwedge f. \text{fst} (\text{lex-ext } f \ n \ ?mss \ ?mts) \implies ?mss \neq []$ 
      by (cases ?mss) (simp-all add: lex-ext-iff)
    with fg t ge1 st' prc nfg cf cg have ss-ne: ?mss  $\neq$  [] by simp
    with us-e s u fh su' prc3 cf cg ch suA show ?thesis unfolding
wpo.simps[of s u] by simp
  next
  case Lex note ch = this
  from fg t ge1 st' nfg cf cg
  have lex1: fst (lex-ext wpo n ?mss ?mts) by auto
  from gh u ge2 tu' ngh cg ch
  have lex2: snd (lex-ext wpo n ?mts ?mus) by auto
  from lex1 lex2 lex-ext-compat[OF ind3', of ?mss ?mts ?mus]
  have fst (lex-ext wpo n ?mss ?mus) by auto
  with s u fh su' prc3 cf cg ch suA show ?thesis unfolding
wpo.simps[of s u] by simp
  qed
  qed
  qed
  qed
  next
  case (Var z)
  from ge2  $\langle \neg ?t1 \rangle$  tuS have ssimple large ?g unfolding Var t
  by (auto simp: wpo.simps split: if-splits)
  from large[OF this, of ?f]
  have large: fst (prc ?g ?f)  $\vee$  snd (prc ?g ?f)  $\wedge$   $\sigma$  ?f = [] by auto
  obtain fgs fgns where prc-fg: prc ?f ?g = (fgs, fgns) by (cases prc ?f
?g, auto)
  from ge1  $\langle \neg ?s1 \rangle$  stS have weak-fg: snd (prc ?f ?g) unfolding s t
using prc-fg
  by (auto simp: wpo.simps split: if-splits)
  have prc-irrefl:  $\neg$  fst (prc ?f ?f) using prc-refl by simp
  from large have False
  proof
    assume fst (prc ?g ?f)
    with weak-fg have fst (prc ?f ?f) by (metis prc-compat prod.collapse)
    with prc-irrefl show False by auto
  next
  assume weak: snd (prc ?g ?f)  $\wedge$   $\sigma$  ?f = []
  let ?mss = map ( $\lambda i. \text{ss } ! i$ ) ( $\sigma$  ?f)

```

```

let ?mts = map (λ j. ts ! j) (σ ?g)
{
  assume fst (prc ?f ?g)
  with weak have fst (prc ?f ?f) by (metis prc-compat prod.collapse)
  with prc-irrefl have False by auto
}
hence ¬ fst (prc ?f ?g) by auto
with ge1 ⟨¬ ?s1⟩ stS prc-fg
have fst (lex-ext wpo n ?mss ?mts) ∨ fst (mul-ext wpo ?mss ?mts)
∨ ?mss ≠ []
  unfolding wpo.simps[of s t] unfolding s t
  by (auto simp: Let-def split: if-splits)
  with weak have fst (lex-ext wpo n [] ?mts) ∨ fst (mul-ext wpo []
?mts) by auto
  thus False using lex-ext-least-2 by (auto simp: mul-ext-def Let-def
s-mul-ext-bottom-strict)
qed
thus ?thesis ..
qed
qed
qed
}
moreover
{
  assume ge1: s ≥ t and ge2: t ≥ u and ngt1: ¬ s > t and ngt2: ¬ t > u
  from wpo-ns-imp-NS[OF ge1] have stA: (s,t) ∈ NS .
  from wpo-ns-imp-NS[OF ge2] have tuA: (t,u) ∈ NS .
  from trans-NS-point[OF stA tuA] have suA: (s,u) ∈ NS .
  from ngt1 stA have ¬ ?s1 unfolding s t by (auto simp: wpo.simps split:
if-splits)
  from ngt2 tuA have ¬ ?t1 unfolding t by (cases u, auto simp: wpo.simps
split: if-splits)
  have s ≥ u
  proof (cases u)
  case u: (Var x)
  from t ⟨¬ ?t1⟩ ge2 tuA ngt2 have large: ssimple large ?g unfolding u
  by (auto simp: wpo.simps split: if-splits)
  from s t ngt1 ge1 have snd (prc ?f ?g)
  by (auto simp: wpo.simps split: if-splits prod.splits)
  from large-trans[OF large this] suA large
  show ?thesis unfolding wpo.simps[of s u] using s u by auto
  next
  case u: (Fun h us)
  let ?u = Fun h us
  let ?h = (h,length us)
  let ?us = set (σ ?h)
  let ?mss = map (λ i. ss ! i) (σ ?f)
  let ?mts = map (λ j. ts ! j) (σ ?g)
  let ?mus = map (λ k. us ! k) (σ ?h)

```

```

from  $s\ t\ u\ ge1\ ge2$  have  $ge1: ?s \succeq ?t$  and  $ge2: ?t \succeq ?u$  by auto
from  $stA\ stS\ s\ t$  have  $stAS: ((?s, ?t) \in S) = False\ ((?s, ?t) \in NS) =$ 
True by auto
from  $tuA\ tuS\ t\ u$  have  $tuAS: ((?t, ?u) \in S) = False\ ((?t, ?u) \in NS) =$ 
True by auto
note  $ge1 = ge1[unfolded\ wpo.simps[of\ ?s\ ?t]\ stAS,\ simplified]$ 
note  $ge2 = ge2[unfolded\ wpo.simps[of\ ?t\ ?u]\ tuAS,\ simplified]$ 
from  $s\ t\ u\ ngt1\ ngt2$  have  $ngt1: \neg ?s \succ ?t$  and  $ngt2: \neg ?t \succ ?u$  by auto
note  $ngt1 = ngt1[unfolded\ wpo.simps[of\ ?s\ ?t]\ stAS,\ simplified]$ 
note  $ngt2 = ngt2[unfolded\ wpo.simps[of\ ?t\ ?u]\ tuAS,\ simplified]$ 
from  $\langle \neg ?s1 \rangle\ t\ ge1$  have  $st': \forall j \in ?ts.\ ?s \succ ts ! j$  by (cases ?thesis,
auto)
from  $\langle \neg ?t1 \rangle\ u\ ge2$  have  $tu': \forall k \in ?us.\ ?t \succ us ! k$  by (cases ?thesis,
auto)
let  $?lu = length\ us$ 
obtain  $ps2\ pns2$  where  $prc2: prc\ ?g\ ?h = (ps2, pns2)$  by force
obtain  $ps3\ pns3$  where  $prc3: prc\ ?f\ ?h = (ps3, pns3)$  by force
from  $\langle \neg ?s1 \rangle\ t\ ge1\ st'$  have  $fg: pns$  by (cases ?thesis, auto simp: prc)
from  $\langle \neg ?t1 \rangle\ u\ ge2\ tu'$  have  $gh: pns2$  by (cases ?thesis, auto simp: prc2)
note  $compat = prc-compat[OF\ prc\ prc2\ prc3]$ 
from  $\langle \neg ?s1 \rangle$  have  $?s1 = False$  by simp
note  $ge1 = ge1[unfolded\ this[unfolded\ t]\ if-False\ term.simps\ prc\ split]$ 
from  $\langle \neg ?t1 \rangle$  have  $?t1 = False$  by simp
note  $ge2 = ge2[unfolded\ this[unfolded\ u]\ if-False\ term.simps\ prc2\ split]$ 
from  $compat\ fg\ gh$  have  $fh: pns3$  by blast
{
  fix  $k$ 
  assume  $k: k \in ?us$ 
  from  $\sigma E[OF\ this]$  have  $size\ (us\ !\ k) < size\ u$  unfolding  $u$  using
size-simps by auto
  with  $tu'[folded\ t]\ \langle s \succeq t \rangle$ 
   $ind[rule-format,\ of\ s\ t\ us\ !\ k]\ k$  have  $s \succ us\ !\ k$  by blast
} note  $su' = this$ 
from  $\langle \neg ?s1 \rangle\ st'\ ge1\ ngt1\ s\ t$  have  $nfg: \neg ps$ 
by (simp, cases ?thesis, simp, cases ps, auto simp: prc fg)
from  $\langle \neg ?t1 \rangle\ tu'\ ge2\ ngt2\ t\ u$  have  $ngh: \neg ps2$ 
by (simp, cases ?thesis, simp, cases ps2, auto simp: prc2 gh)
show  $s \succeq u$ 
proof (cases c ?f)
  case Mul note  $cf = this$ 
  show ?thesis
  proof (cases c ?g)
    case Mul note  $cg = this$ 
    show ?thesis
    proof (cases c ?h)
      case Mul note  $ch = this$ 
      from  $fg\ t\ ge1\ st'\ nfg\ cf\ cg$ 
      have  $mull: snd\ (mul-ext\ wpo\ ?mss\ ?mts)$  by auto
      from  $gh\ u\ ge2\ tu'\ ngh\ cg\ ch$ 

```

```

      have mul2: snd (mul-ext wpo ?mts ?mus) by auto
      from mul1 mul2 mul-ext-compat[OF ind3', of ?mss ?mts ?mus]
      have snd (mul-ext wpo ?mss ?mus) by auto
      with s u fh su' prc3 cf ch suA show ?thesis unfolding wpo.simps[of
s u] by simp
    next
      case Lex note ch = this
      from gh u ge2 tu' ngh cg ch have us-e: ?mus = [] by simp
      with s u fh su' prc3 cf cg ch suA show ?thesis unfolding wpo.simps[of
s u] by simp
    qed
  next
    case Lex note cg = this
    from fg t ge1 st' nfg cf cg have ts-e: ?mts = [] by simp
    show ?thesis
    proof (cases c ?h)
      case Mul note ch = this
      with gh u ge2 tu' ngh cg ch have ?mus = [] by simp
      with s u fh su' prc3 cf cg ch ns-mul-ext-bottom suA
      show ?thesis unfolding wpo.simps[of s u] by (simp add: ns-mul-ext-def
mul-ext-def Let-def mult2-alt-ns-def)
    next
      case Lex note ch = this
      have  $\bigwedge f. \text{snd} (\text{lex-ext } f \ n \ \square \ ?mus) \implies ?mus = \square$  by (simp-all add:
lex-ext-iff)
      with ts-e gh u ge2 tu' ngh cg ch have ?mus = [] by simp
      with s u fh su' prc3 cf cg ch suA show ?thesis unfolding wpo.simps[of
s u] by simp
    qed
  qed
next
  case Lex note cf = this
  show ?thesis
  proof (cases c ?g)
    case Mul note cg = this
    from fg t ge1 st' prc nfg cf cg have ts-e: ?mts = [] by simp
    show ?thesis
    proof (cases c ?h)
      case Mul note ch = this
      with ts-e gh u ge2 tu' ngh cg ch
      ns-mul-ext-bottom-uniqueness[of mset ?mus]
      have ?mus = [] by (simp add: Let-def mul-ext-def)
      with s u fh su' prc3 cf cg ch suA show ?thesis unfolding wpo.simps[of
s u] by simp
    next
      case Lex note ch = this
      with gh u ge2 tu' prc2 ngh cg ch have ?mus = [] by simp
      with s u fh su' prc3 cf cg ch suA show ?thesis unfolding wpo.simps[of
s u] by (simp add: lex-ext-least-1)
    qed
  qed

```

```

      qed
    next
      case Lex note cg = this
      show ?thesis
      proof (cases c ?h)
        case Mul note ch = this
          with gh u ge2 tu' ngh cg ch have ?mus = [] by simp
          with s u fh su' prc3 cf cg ch suA show ?thesis unfolding wpo.simps[of
s u] by (simp add: lex-ext-least-1)
        next
          case Lex note ch = this
          from st' ge1 s t fg nfg cf cg
          have lex1: snd (lex-ext wpo n ?mss ?mts) by (auto simp: prc)
          from tu' ge2 t u gh ngh cg ch
          have lex2: snd (lex-ext wpo n ?mts ?mus) by (auto simp: prc2)
          from lex1 lex2 lex-ext-compat[OF ind3', of ?mss ?mts ?mus]
          have snd (lex-ext wpo n ?mss ?mus) by auto
          with fg gh su' s u fh cf cg ch suA show ?thesis unfolding
wpo.simps[of s u] by (auto simp: prc3)
      qed
    qed
  }
  ultimately
  show ?thesis using wpo-s-imp-ns by auto
  qed
  qed
  qed
  qed

```

context

```

  assumes ssimple: strictly-simple-status  $\sigma \sigma NS$ 
begin
lemma NS-arg':
  assumes i: i ∈ set (σ (f,length ts))
  shows (Fun f ts, ts ! i) ∈ NS
  using assms ssimple unfolding simple-arg-pos-def strictly-simple-status-def by
simp

```

lemma *wpo-ns-refl'*:

```

  shows  $s \succeq s$ 
proof (induct s)
case (Fun f ss)
{
  fix i
  assume si: i ∈ set (σ (f,length ss))
  from NS-arg'[OF this] have (Fun f ss, ss ! i) ∈ NS .

```

```

    with si Fun[OF status-aux[OF si]] have wpo-s (Fun f ss) (ss ! i) unfolding
wpo.simps[of Fun f ss ! i]
    by auto
  } note wpo-s = this
let ?ss = map (λ i. ss ! i) (σ (f,length ss))
have rec11: snd (lex-ext wpo n ?ss ?ss)
  by (rule all-nstri-imp-lex-nstri, insert σE[of - f ss], auto simp: Fun)
have rec12: snd (mul-ext wpo ?ss ?ss)
  unfolding mul-ext-def Let-def snd-conv
  by (intro ns-mul-ext-refl-local,
      unfold locally-refl-def, auto simp: in-multiset-in-set[of ?ss] intro!: Fun
status-aux)
from rec11 rec12 show ?case using refl-NS-point wpo-s
  by (cases c (f,length ss), auto simp: wpo.simps[of Fun f ss Fun f ss] prc-refl)
qed (simp add: wpo.simps refl-NS-point)

```

```

lemma wpo-stable': fixes δ :: ('f,'v)subst
  shows (s > t → s · δ > t · δ) ∧ (s ≥ t → s · δ ≥ t · δ)
  (is ?p s t)
proof (induct (s,t) arbitrary:s t rule: wf-induct[OF wf-measure[of λ (s,t). size s +
size t]])
  case (1 s t)
  from 1
  have ∀ s' t'. size s' + size t' < size s + size t → ?p s' t' by auto
  note IH = this[rule-format]
  let ?s = s · δ
  let ?t = t · δ
  note_simps = wpo.simps[of s t] wpo.simps[of ?s ?t]
  show ?case
  proof (cases ((s,t) ∈ S ∨ (?s,?t) ∈ S) ∨ ((s,t) ∉ NS ∨ ¬ wpo-ns s t))
    case True
    then show ?thesis
    proof
      assume (s,t) ∈ S ∨ (?s,?t) ∈ S
      with subst-S[of s t δ] have (?s,?t) ∈ S by blast
      from S-imp-wpo-s[OF this] have wpo-s ?s ?t .
      with wpo-s-imp-ns[OF this] show ?thesis by blast
    next
      assume (s,t) ∉ NS ∨ ¬ wpo-ns s t
      with wpo-ns-imp-NS have st: ¬ wpo-ns s t by auto
      with wpo-s-imp-ns have ¬ wpo-s s t by auto
      with st show ?thesis by blast
    qed
  next
  case False
  then have not: ((s,t) ∈ S) = False ((?s,?t) ∈ S) = False
  and stA: (s,t) ∈ NS and ns: wpo-ns s t by auto
  from subst-NS[OF stA] have sstsA: (?s,?t) ∈ NS by auto
  from stA sstsA have id: ((s,t) ∈ NS) = True ((?s,?t) ∈ NS) = True by auto

```

```

note simps = simps[unfolded id not if-False if-True]
show ?thesis
proof (cases s)
  case (Var x) note s = this
  show ?thesis
  proof (cases t)
    case (Var y) note t = this
    show ?thesis unfolding simps(1) unfolding s t using wpo-ns-refl'[of  $\delta$  y]
by auto
  next
    case (Fun g ts) note t = this
    let ?g = (g, length ts)
    show ?thesis
    proof (cases  $\delta$  x)
      case (Var y)
        then show ?thesis unfolding simps unfolding s t by simp
    next
      case (Fun f ss)
      let ?f = (f, length ss)
      show ?thesis
      proof (cases prl ?g)
        case False then show ?thesis unfolding simps unfolding s t Fun by
auto
      next
        case True
        obtain s ns where prc ?f ?g = (s, ns) by force
        with prl[OF True, of ?f] have prc: prc ?f ?g = (s, True) by auto
        show ?thesis unfolding simps unfolding s t Fun
        by (auto simp: Fun prc mul-ext-def ns-mul-ext-bottom Let-def intro!:
all-nstri-imp-lex-nstri[of [], simplified])
      qed
    qed
  qed
next
  case (Fun f ss) note s = this
  let ?f = (f, length ss)
  let ?ss = set ( $\sigma$  ?f)
  {
    fix i
    assume i: i  $\in$  ?ss and ns: wpo-ns (ss ! i) t
    from IH[of ss ! i t]  $\sigma E$ [OF i] ns have wpo-ns (ss ! i  $\cdot$   $\delta$ ) ?t using s
    by (auto simp: size-simps)
    then have wpo-s ?s ?t using i sstsA  $\sigma$ [of f length ss] unfolding simps
unfolding s by force
    with wpo-s-imp-ns[OF this] have ?thesis by blast
  } note si-arg = this
show ?thesis
proof (cases t)
  case t: (Var y)

```

```

show ?thesis
proof (cases  $\exists i \in ?ss. wpo\text{-}ns (ss ! i) t$ )
  case True
  then obtain i
    where si:  $i \in ?ss$  and ns:  $wpo\text{-}ns (ss ! i) t$ 
    unfolding s t by auto
    from si-arg[OF this] show ?thesis .
  next
  case False
  with ns[unfolded_simps] s t
  have ssimple and largef: large ?f by (auto split: if-splits)
  from False s t not
  have  $\neg wpo\text{-}s s t$  unfolding wpo.simps[of s t] by auto
  moreover
  have wpo-ns ?s ?t
  proof (cases  $\delta y$ )
    case (Var z)
    show ?thesis unfolding wpo.simps[of ?s ?t] not id
      unfolding s t using Var  $\langle ssimple \rangle$  largef by auto
  next
  case (Fun g ts)
  let ?g = (g, length ts)
  obtain ps pns where prc:  $prc ?f ?g = (ps, pns)$  by (cases prc ?f ?g, auto)
  from prc-stri-imp-nstri[of ?f ?g] prc have ps:  $ps \implies pns$  by auto
  {
    fix j
    assume j  $\in set (\sigma ?g)$ 
    with set-status-nth[OF refl this] ss-status[OF  $\langle ssimple \rangle$  this] t Fun
    have  $(t \cdot \delta, ts ! j) \in S$  by (auto simp: simple-arg-pos-def)
    with sstsA have S:  $(s \cdot \delta, ts ! j) \in S$  by (metis compat-NS-S-point)
    hence wpo-s  $(s \cdot \delta) (ts ! j)$  by (rule S-imp-wpo-s)
  } note ssimple = this
  from large[OF  $\langle ssimple \rangle$  largef, of ?g, unfolded prc]
  have ps  $\vee$  pns  $\wedge$   $\sigma ?g = []$  by auto
  thus ?thesis using ssimple unfolding wpo.simps[of ?s ?t] not id
  unfolding s t using Fun prc ps by (auto simp: lex-ext-least-1 mul-ext-def
Let-def ns-mul-ext-bottom)
  qed
  ultimately show ?thesis by blast
  qed
next
case (Fun g ts) note t = this
let ?g = (g, length ts)
let ?ts = set ( $\sigma ?g$ )
obtain prs prns where p:  $prc ?f ?g = (prs, prns)$  by force
note ns = ns[unfolded_simps, unfolded s t p term.simps split]
show ?thesis
proof (cases  $\exists i \in ?ss. wpo\text{-}ns (ss ! i) t$ )
  case True

```

```

    with si-arg show ?thesis by blast
  next
    case False
    then have id: ( $\exists i \in ?ss. \text{wpo-ns } (ss ! i) (\text{Fun } g \text{ } ts) = \text{False}$ ) unfolding t
  by auto
    note ns = ns[unfolded this if-False]
    let ?mss = map ( $\lambda s . s \cdot \delta$ ) ss
    let ?mts = map ( $\lambda t . t \cdot \delta$ ) ts
    from ns have prns and s-tj:  $\bigwedge j. j \in ?ts \implies \text{wpo-s } (\text{Fun } f \text{ } ss) (ts ! j)$ 
      by (auto split: if-splits)
    {
      fix j
      assume j:  $j \in ?ts$ 
      from  $\sigma E$ [OF this]
      have  $\text{size } s + \text{size } (ts ! j) < \text{size } s + \text{size } t$  unfolding t by (auto simp:
size-simps)
      from IH[OF this] s-tj[OF j, folded s] have wpo:  $\text{wpo-s } ?s (ts ! j \cdot \delta)$  by
auto
      from j  $\sigma$ [of g length ts] have  $j < \text{length } ts$  by auto
      with wpo have  $\text{wpo-s } ?s (?mts ! j)$  by auto
    } note ss-ts = this
    note  $\sigma E = \sigma E$ [of - f ss]  $\sigma E$ [of - g ts]
    show ?thesis
    proof (cases prns)
      case True
      with ss-ts sstsA p ⟨prns⟩ have  $\text{wpo-s } ?s ?t$  unfolding simps unfolding
s t
      by (auto split: if-splits)
      with  $\text{wpo-s-imp-ns}$ [OF this] show ?thesis by blast
    next
      case False
      let ?mmss = map (!) ss ( $\sigma ?f$ )
      let ?mmts = map (!) ts ( $\sigma ?g$ )
      let ?Mmss = map (!) ?mss ( $\sigma ?f$ )
      let ?Mmts = map (!) ?mts ( $\sigma ?g$ )
      have id-map: ?Mmss = map ( $\lambda t. t \cdot \delta$ ) ?mmss ?Mmts = map ( $\lambda t. t \cdot$ 
 $\delta$ ) ?mmts
      unfolding map-map o-def by (auto simp: set-status-nth)
      let ?ls = length ( $\sigma ?f$ )
      let ?lt = length ( $\sigma ?g$ )
      {
        fix si tj
        assume *:  $si \in \text{set } ?mmss \quad tj \in \text{set } ?mmts$ 
        have ( $\text{wpo-s } si \quad tj \implies \text{wpo-s } (si \cdot \delta) (tj \cdot \delta)$ )  $\wedge$  ( $\text{wpo-ns } si \quad tj \implies \text{wpo-ns } (si \cdot \delta) (tj \cdot \delta)$ )
        proof (intro IH add-strict-mono)
          from *(1) have  $si \in \text{set } ss$  using set-status-nth[of - - - sigma] by auto
          then show  $\text{size } si < \text{size } s$  unfolding s by (auto simp: termination-simp)
          from *(2) have  $tj \in \text{set } ts$  using set-status-nth[of - - - sigma] by auto

```

**then show**  $size\ tj < size\ t$  **unfolding**  $t$  **by** (*auto simp: termination-simp*)  
**qed**  
**hence**  $wpo-s\ si\ tj \implies wpo-s\ (si \cdot \delta)\ (tj \cdot \delta)$   
 $wpo-ns\ si\ tj \implies wpo-ns\ (si \cdot \delta)\ (tj \cdot \delta)$  **by** *blast+*  
**} note**  $IH' = this$   
**{**  
**fix**  $i$   
**assume**  $i < ?ls\ i < ?lt$   
**then have**  $i-f: i < length\ (\sigma\ ?f)$  **and**  $i-g: i < length\ (\sigma\ ?g)$  **by** *auto*  
**with**  $\sigma[of\ f\ length\ ss]\ \sigma[of\ g\ length\ ts]$  **have**  $i: \sigma\ ?f!\ i < length\ ss\ \sigma\ ?g$   
 $!\ i < length\ ts$   
**unfolding** *set-conv-nth* **by** *auto*  
**then have**  $size\ (ss!\ (\sigma\ ?f!\ i)) < size\ s\ size\ (ts!\ (\sigma\ ?g!\ i)) < size\ t$   
**unfolding**  $s\ t$  **by** (*auto simp: size-simps*)  
**then have**  $size\ (ss!\ (\sigma\ ?f!\ i)) + size\ (ts!\ (\sigma\ ?g!\ i)) < size\ s + size\ t$   
**by** *simp*  
**from**  $IH[OF\ this]$   $i\ i-f\ i-g$   
**have**  $(wpo-s\ (?mmss!\ i)\ (?mmts!\ i)) \implies$   
 $wpo-s\ (?mss!\ (\sigma\ ?f!\ i))\ (?mts!\ (\sigma\ ?g!\ i))$   
 $(wpo-ns\ (?mmss!\ i)\ (?mmts!\ i)) \implies$   
 $wpo-ns\ (?mss!\ (\sigma\ ?f!\ i))\ (?mts!\ (\sigma\ ?g!\ i))$  **by** *auto*  
**} note**  $IH = this$   
**consider**  $(Lex)\ c\ ?f = Lex\ c\ ?g = Lex\ | (Mul)\ c\ ?f = Mul\ c\ ?g = Mul\ |$   
 $(Diff)\ c\ ?f \neq c\ ?g$   
**by** (*cases\ c\ ?f; cases\ c\ ?g, auto*)  
**thus**  $?thesis$   
**proof** *cases*  
**case**  $Lex$   
**from**  $Lex\ False\ ns$  **have**  $snd\ (lex-ext\ wpo\ n\ ?mmss\ ?mmts)$  **by** (*auto*  
*split: if-splits*)  
**from**  $this[unfolding\ lex-ext-iff\ snd-conv]$   
**have**  $len: (?ls = ?lt \vee ?lt \leq n)$   
**and** *choice:  $(\exists\ i < ?ls.$*   
 $i < ?lt \wedge (\forall\ j < i. wpo-ns\ (?mmss!\ j)\ (?mmts!\ j)) \wedge wpo-s\ (?mmss!\$   
 $i)\ (?mmts!\ i) \vee$   
 $(\forall\ i < ?lt. wpo-ns\ (?mmss!\ i)\ (?mmts!\ i)) \wedge ?lt \leq ?ls$  **(is**  $?stri \vee$   
 $?nstri)$  **by** *auto*  
**from** *choice* **have**  $?stri \vee (\neg\ ?stri \wedge ?nstri)$  **by** *blast*  
**then show**  $?thesis$   
**proof**  
**assume**  $?stri$   
**then obtain**  $i$  **where**  $i: i < ?ls\ i < ?lt$   
**and**  $NS: (\forall\ j < i. wpo-ns\ (?mmss!\ j)\ (?mmts!\ j))$  **and**  $S: wpo-s$   
 $(?mmss!\ i)\ (?mmts!\ i)$  **by** *auto*  
**with**  $IH$  **have**  $(\forall\ j < i. wpo-ns\ (?Mmss!\ j)\ (?Mmts!\ j))\ wpo-s\ (?Mmss$   
 $!\ i)\ (?Mmts!\ i)$  **by** *auto*  
**with**  $i\ len$  **have**  $fst\ (lex-ext\ wpo\ n\ ?Mmss\ ?Mmts)$  **unfolding** *lex-ext-iff*  
**by** *auto*  
**with**  $Lex\ ss-ts\ sstsA\ p\ \langle prms \rangle$  **have**  $wpo-s\ ?s\ ?t$  **unfolding** *simps*

```

unfolding s t
  by (auto split: if-splits)
  with wpo-s-imp-ns[OF this] show ?thesis by blast
next
  assume  $\neg ?stri \wedge ?nstri$ 
  then have ?nstri and nstri:  $\neg ?stri$  by blast+
  with IH have  $(\forall i < ?lt. wpo-ns (?Mmss ! i) (?Mmts ! i)) \wedge ?lt \leq$ 
?ls by auto
  with len have snd (lex-ext wpo n ?Mmss ?Mmts) unfolding lex-ext-iff
  by auto
  with Lex ss-ts sstsA p ⟨prns⟩ have ns: wpo-ns ?s ?t unfolding_simps
unfolding s t
  by (auto split: if-splits)
  {
  assume wpo-s s t
  from Lex this[unfolded_simps, unfolded s t term_simps p split id]
False
  have fst (lex-ext wpo n ?mmss ?mmts) by (auto split: if-splits)
  from this[unfolded_lex-ext-iff fst-conv] nstri
  have  $(\forall i < ?lt. wpo-ns (?mmss ! i) (?mmts ! i)) \wedge ?lt < ?ls$  by auto
  with IH have  $(\forall i < ?lt. wpo-ns (?Mmss ! i) (?Mmts ! i)) \wedge ?lt <$ 
?ls by auto
  then have fst (lex-ext wpo n ?Mmss ?Mmts) using len unfolding
lex-ext-iff by auto
  with Lex ss-ts sstsA p ⟨prns⟩ have ns: wpo-s ?s ?t unfolding_simps
unfolding s t
  by (auto split: if-splits)
  }
  with ns show ?thesis by blast
qed
next
case Diff
  thus ?thesis using ns ss-ts sstsA p ⟨prns⟩ unfolding_simps unfolding
s t
  by (auto simp: Let-def split: if-splits)
next
case Mul
  from Mul False ns have ge: snd (mul-ext wpo ?mmss ?mmts) by (auto
split: if-splits)
  have ge: snd (mul-ext wpo ?Mmss ?Mmts) unfolding id-map
  by (rule nstri-mul-ext-map[OF - - ge], (intro IH', auto)+)
  {
  assume gr: fst (mul-ext wpo ?mmss ?mmts)
  have grσ: fst (mul-ext wpo ?Mmss ?Mmts) unfolding id-map
  by (rule stri-mul-ext-map[OF - - gr], (intro IH', auto)+)
  }
  note gr = this
  from ge gr
  show ?thesis
  using ss-ts ⟨prns⟩ unfolding_simps

```

```

      unfolding s t term.simps p split eval-term.simps length-map Mul
      by (simp add: id-map id)
    qed
  qed
  qed
  qed
  qed
  qed
  qed
  qed

lemma subterm-wpo-s-arg': assumes i: i ∈ set (σ (f,length ss))
  shows Fun f ss > ss ! i
proof -
  have refl: ss ! i ≥ ss ! i by (rule wpo-ns-refl)
  with i have ∃ t ∈ set (σ (f,length ss)). ss ! i ≥ ss ! i by auto
  with NS-arg'[OF i] i
  show ?thesis by (auto simp: wpo.simps split: if-splits)
qed

context
  fixes f s t bef aft
  assumes ctxt-NS: (s,t) ∈ NS ⇒ (Fun f (bef @ s # aft), Fun f (bef @ t # aft))
  ∈ NS
begin

lemma wpo-ns-pre-mono':
  defines σf ≡ σ (f, Suc (length bef + length aft))
  assumes rel: (wpo-ns s t)
  shows (∀ j ∈ set σf. Fun f (bef @ s # aft) > (bef @ t # aft) ! j)
    ∧ (Fun f (bef @ s # aft), (Fun f (bef @ t # aft))) ∈ NS
    ∧ (∀ i < length σf. ((map (!) (bef @ s # aft)) σf) ! i) ≥ ((map (!) (bef @ t
# aft)) σf) ! i)
    (is - ∧ - ∧ ?three)
proof -
  let ?ss = bef @ s # aft
  let ?ts = bef @ t # aft
  let ?s = Fun f ?ss
  let ?t = Fun f ?ts
  let ?len = Suc (length bef + length aft)
  let ?f = (f, ?len)
  let ?σ = σ ?f
  from wpo-ns-imp-NS[OF rel] have stA: (s,t) ∈ NS .
  have ?three unfolding σf-def
  proof (intro allI impI)
    fix i
    assume i < length ?σ
    then have id: ∧ ss. (map (!) ss) ?σ ! i = ss ! (?σ ! i) by auto
    show wpo-ns ((map (!) ?ss) ?σ) ! i ((map (!) ?ts) ?σ) ! i
    proof (cases ?σ ! i = length bef)

```

```

    case True
    then show ?thesis unfolding id using rel by auto
  next
  case False
  from append-Cons-nth-not-middle[OF this, of s aft t] wpo-ns-refl'
  show ?thesis unfolding id by auto
qed
qed
have  $\forall j \in \text{set } ?\sigma. \text{wpo-}s \ ?s \ ((\text{bef} \ @ \ t \ \# \ \text{aft}) \ ! \ j) \ (\text{is} \ ?one)$ 
proof
  fix j
  assume j:  $j \in \text{set } ?\sigma$ 
  then have  $j \in \text{set} \ (\sigma \ (f, \text{length } ?ss))$  by simp
  from subterm-wpo-s-arg'[OF this]
  have  $s: \text{wpo-}s \ ?s \ (?ss \ ! \ j)$  .
  show  $\text{wpo-}s \ ?s \ (?ts \ ! \ j)$ 
  proof (cases  $j = \text{length } \text{bef}$ )
    case False
    then have  $?ss \ ! \ j = ?ts \ ! \ j$  by (rule append-Cons-nth-not-middle)
    with s show ?thesis by simp
  next
  case True
  with s have  $\text{wpo-}s \ ?s \ s$  by simp
  with rel wpo-compat have  $\text{wpo-}s \ ?s \ t$  by fast
  with True show ?thesis by simp
qed
qed
with  $\langle ?three \rangle \text{ctxt-NS}[OF \ \text{stA}]$  show ?thesis unfolding  $\sigma f\text{-def}$  by auto
qed

```

**lemma** *wpo-ns-mono'*:

```

  assumes rel:  $s \succeq t$ 
  shows  $\text{Fun } f \ (\text{bef} \ @ \ s \ \# \ \text{aft}) \succeq \text{Fun } f \ (\text{bef} \ @ \ t \ \# \ \text{aft})$ 
proof -
  let ?ss =  $\text{bef} \ @ \ s \ \# \ \text{aft}$ 
  let ?ts =  $\text{bef} \ @ \ t \ \# \ \text{aft}$ 
  let ?s =  $\text{Fun } f \ ?ss$ 
  let ?t =  $\text{Fun } f \ ?ts$ 
  let ?len =  $\text{Suc} \ (\text{length } \text{bef} \ + \ \text{length } \ \text{aft})$ 
  let ?f =  $(f, \ ?len)$ 
  let ? $\sigma$  =  $\sigma \ ?f$ 
  from wpo-ns-pre-mono'[OF rel]
  have id:  $(\forall j \in \text{set } ?\sigma. \text{wpo-}s \ ?s \ ((\text{bef} \ @ \ t \ \# \ \text{aft}) \ ! \ j)) = \text{True}$ 
     $((?s, ?t) \in \text{NS}) = \text{True}$ 
     $\text{length } ?ss = ?len \ \text{length } ?ts = ?len$ 
  by auto
  have snd  $(\text{lex-ext } \text{wpo } n \ (\text{map} \ (!) \ ?ss) \ ?\sigma) \ (\text{map} \ (!) \ ?ts) \ ?\sigma)$ 
  by (rule all-nstri-imp-lex-nstri, intro allI impI, insert wpo-ns-pre-mono'[OF
rel], auto)

```

```

moreover have snd (mul-ext wpo (map (!) ?ss) ?σ) (map (!) ?ts) ?σ)
  by (rule all-nstri-imp-mul-nstri, intro allI impI, insert wpo-ns-pre-mono[OF
rel], auto)
  ultimately show ?thesis unfolding wpo.simps[of ?s ?t] term.simps id prc-refl
    using order-tag.exhaust by (auto simp: Let-def)
qed

```

```

end
end
end

```

```

locale wpo-with-assms = wpo-with-basic-assms + order-pair +
  constrains S :: ('f, 'v) term rel and NS :: -
  and prc :: 'f × nat ⇒ 'f × nat ⇒ bool × bool
  and prl :: 'f × nat ⇒ bool
  and ssimple :: bool
  and large :: 'f × nat ⇒ bool
  and c :: 'f × nat ⇒ order-tag
  and n :: nat
  and σσ :: 'f status
  assumes ctxt-NS: (s,t) ∈ NS ⇒ (Fun f (bef @ s # aft), Fun f (bef @ t # aft))
  ∈ NS
  and ws-status: i ∈ set (status σσ fn) ⇒ simple-arg-pos NS fn i
begin

```

```

lemma ssimple: strictly-simple-status σσ NS
  using ws-status set-status-nth unfolding strictly-simple-status-def simple-arg-pos-def
by fastforce

```

```

lemma trans-prc: trans-precedence prc
  unfolding trans-precedence-def
proof (intro allI, goal-cases)
  case (1 f g h)
  show ?case using prc-compat[of f g - - h] by (cases prc f g; cases prc g h; cases
prc f h, auto)
qed

```

```

lemma NS-arg: assumes i: i ∈ set (σ (f, length ts))
  shows (Fun f ts, ts ! i) ∈ NS
  using NS-arg'[OF ssimple i] .

```

```

lemma NS-subterm: assumes all:  $\bigwedge f k. \text{set } (\sigma (f, k)) = \{0 ..< k\}$ 
  shows  $s \supseteq t \implies (s, t) \in NS$ 
proof (induct s t rule: supteq.induct)
  case (refl)
  from refl-NS show ?case unfolding refl-on-def by blast
next
  case (subt s ss t f)
  from subt(1) obtain i where i: i < length ss and s: s = ss ! i unfolding

```

```

set-conv-nth by auto
  from NS-arg[of i f ss, unfolded all] s i have (Fun f ss, s) ∈ NS by auto
  from trans-NS-point[OF this subt(3)] show ?case .
qed

lemma wpo-ns-refl: s ≥ s
  using wpo-ns-refl'[OF ssimple] .

lemma subterm-wpo-s-arg: assumes i: i ∈ set (σ (f,length ss))
  shows Fun f ss > ss ! i
  by (rule subterm-wpo-s-arg'[OF ssimple i])

lemma subterm-wpo-ns-arg: assumes i: i ∈ set (σ (f,length ss))
  shows Fun f ss ≥ ss ! i
  by (rule wpo-s-imp-ns[OF subterm-wpo-s-arg[OF i]])

lemma wpo-irrefl: ¬ (s > s)
proof
  assume s > s
  thus False
proof (induct s)
  case Var
  thus False using irrefl-S by (auto simp: wpo.simps irrefl-def split: if-splits)
next
  case (Fun f ss)
  let ?s = Fun f ss
  let ?n = length ss
  let ?f = (f,length ss)
  let ?sub = ∃ i ∈ set (σ ?f). ss ! i ≥ ?s
  {
    fix i
    assume i: i ∈ set (σ ?f) and ge: ss ! i ≥ ?s
    with status[of σ σ f ?n] have i < ?n by auto
    hence ss ! i ∈ set ss by auto
    from Fun(1)[OF this] have not: ¬ (ss ! i > ss ! i) by auto
    from ge subterm-wpo-s-arg[OF i] have ss ! i > ss ! i
      using wpo-compat by blast
    with not have False ..
  }
  hence id0: ?sub = False by auto
  from irrefl-S refl-NS have id1: ((?s, ?s) ∈ S) = False ((?s, ?s) ∈ NS) = True
  unfolding irrefl-def refl-on-def by auto
  let ?ss = map (!) ss (σ ?f)
  define ss' where ss' = ?ss
  have set ss' ⊆ set ss using status[of σ σ f ?n] by (auto simp: ss'-def)
  note IH = Fun(1)[OF set-mp[OF this]]
  from Fun(2)[unfolded wpo.simps[of ?s ?s] id1 id0 if-False if-True term.simps
pre-refl split Let-def]

```

```

have fst (lex-ext wpo n ss' ss')  $\vee$  fst (mul-ext wpo ss' ss')
  by (auto split: if-splits simp: ss'-def)
thus False
proof
  assume fst (lex-ext wpo n ss' ss')
  with lex-ext-irrefl[of ss' wpo n] IH show False by auto
next
  assume fst (mul-ext wpo ss' ss')
  with mul-ext-irrefl[of ss' wpo, OF - - wpo-s-imp-ns] IH wpo-compat
  show False by blast
qed
qed
qed

```

```

lemma wpo-ns-mono:
  assumes rel:  $s \succeq t$ 
  shows Fun f (bef @ s # aft)  $\succeq$  Fun f (bef @ t # aft)
  by (rule wpo-ns-mono'[OF ssimple ctxt-NS rel])

```

```

lemma wpo-ns-pre-mono: fixes f and bef aft :: ('f,'v)term list
  defines  $\sigma f \equiv \sigma (f, \text{Suc} (\text{length bef} + \text{length aft}))$ 
  assumes rel: (wpo-ns s t)
  shows ( $\forall j \in \text{set } \sigma f. \text{Fun } f (\text{bef @ } s \# \text{aft}) \succ (\text{bef @ } t \# \text{aft}) ! j$ )
     $\wedge$  (Fun f (bef @ s # aft), (Fun f (bef @ t # aft)))  $\in$  NS
     $\wedge$  ( $\forall i < \text{length } \sigma f. ((\text{map } (!) (\text{bef @ } s \# \text{aft})) \sigma f) ! i \succeq ((\text{map } (!) (\text{bef @ } t \# \text{aft})) \sigma f) ! i$ )
  unfolding  $\sigma f$ -def
  by (rule wpo-ns-pre-mono'[OF ssimple ctxt-NS rel])

```

```

lemma wpo-stable: fixes  $\delta :: ('f,'v)\text{subst}$ 
  shows ( $s \succ t \longrightarrow s \cdot \delta \succ t \cdot \delta$ )  $\wedge$  ( $s \succeq t \longrightarrow s \cdot \delta \succeq t \cdot \delta$ )
  by (rule wpo-stable'[OF ssimple])

```

```

theorem wpo-order-pair: order-pair WPO-S WPO-NS
proof
  show refl WPO-NS using wpo-ns-refl unfolding refl-on-def by auto
  show trans WPO-NS using wpo-compat unfolding trans-def by blast
  show trans WPO-S using wpo-compat wpo-s-imp-ns unfolding trans-def by
    blast
  show WPO-NS  $O$  WPO-S  $\subseteq$  WPO-S using wpo-compat by blast
  show WPO-S  $O$  WPO-NS  $\subseteq$  WPO-S using wpo-compat by blast
qed

```

```

theorem WPO-S-subst:  $(s,t) \in \text{WPO-S} \implies (s \cdot \sigma, t \cdot \sigma) \in \text{WPO-S}$  for  $\sigma$ 
  using wpo-stable by auto

```

```

theorem WPO-NS-subst:  $(s,t) \in \text{WPO-NS} \implies (s \cdot \sigma, t \cdot \sigma) \in \text{WPO-NS}$  for  $\sigma$ 
  using wpo-stable by auto

```

**theorem** *WPO-NS-ctxt*:  $(s,t) \in \text{WPO-NS} \implies (\text{Fun } f (\text{bef } @ s \# \text{aft}), \text{Fun } f (\text{bef } @ t \# \text{aft})) \in \text{WPO-NS}$

using *wpo-ns-mono* by *blast*

**theorem** *WPO-S-subset-WPO-NS*:  $\text{WPO-S} \subseteq \text{WPO-NS}$

using *wpo-s-imp-ns* by *blast*

**context**

assumes  $\sigma\text{-full}$ :  $\bigwedge f k. \text{set } (\sigma (f,k)) = \{0 ..< k\}$

**begin**

**lemma** *subterm-wpo-s*:  $s \triangleright t \implies s \succ t$

**proof** (*induct s t rule: supt.induct*)

case (*arg s ss f*)

from *arg*[*unfolded set-conv-nth*] **obtain** *i* **where**  $i: i < \text{length } ss$  **and**  $s: s = ss !$   
*i* **by** *auto*

from  $\sigma\text{-full}$ [*of f length ss*] *i* **have**  $ii: i \in \text{set } (\sigma (f, \text{length } ss))$  **by** *auto*

from *subterm-wpo-s-arg*[*OF ii*] *s* **show** *?case* **by** *auto*

**next**

case (*subt s ss t f*)

from *subt wpo-s-imp-ns* **have**  $\exists s \in \text{set } ss. \text{wpo-ns } s t$  **by** *blast*

from *this*[*unfolded set-conv-nth*] **obtain** *i* **where**  $ns: ss ! i \succeq t$  **and**  $i: i < \text{length } ss$  **by** *auto*

from  $\sigma\text{-full}$ [*of f length ss*] *i* **have**  $ii: i \in \text{set } (\sigma (f, \text{length } ss))$  **by** *auto*

from *subt* **have**  $\text{Fun } f ss \triangleright t$  **by** *auto*

from *NS-subterm*[*OF sigma-full this*] *ns ii*

**show** *?case* **by** (*auto simp: wpo.simps split: if-splits*)

**qed**

**lemma** *subterm-wpo-ns*: **assumes** *supteq*:  $s \triangleright t$  **shows**  $s \succeq t$

**proof** –

from *supteq* **have**  $s = t \vee s \triangleright t$  **by** *auto*

**then** **show** *?thesis*

**proof**

assume  $s = t$  **then** **show** *?thesis* **using** *wpo-ns-refl* **by** *blast*

**next**

assume  $s \triangleright t$

from *wpo-s-imp-ns*[*OF subterm-wpo-s* [*OF this*]]

**show** *?thesis* .

**qed**

**qed**

**lemma** *wpo-s-mono*: **assumes** *rels*:  $s \succ t$

**shows**  $\text{Fun } f (\text{bef } @ s \# \text{aft}) \succ \text{Fun } f (\text{bef } @ t \# \text{aft})$

**proof** –

**let** *?ss* =  $\text{bef } @ s \# \text{aft}$

**let** *?ts* =  $\text{bef } @ t \# \text{aft}$

```

let ?s = Fun f ?ss
let ?t = Fun f ?ts
let ?len = Suc (length bef + length aft)
let ?f = (f, ?len)
let ?σ = σ ?f
from wpo-s-imp-ns[OF rels] have rel: wpo-ns s t .
from wpo-ns-pre-mono[OF rel]
have id: (∀ j ∈ set ?σ. wpo-s ?s ((bef @ t # aft) ! j)) = True
  ((?s, ?t) ∈ NS) = True
  length ?ss = ?len length ?ts = ?len
  by auto
let ?lb = length bef
from σ-full[of f ?len] have lb-mem: ?lb ∈ set ?σ by auto
then obtain i where σi: ?σ ! i = ?lb and i: i < length ?σ
  unfolding set-conv-nth by force
let ?mss = map (!) ?ss ?σ
let ?mts = map (!) ?ts ?σ
have fst (lex-ext wpo n ?mss ?mts)
  unfolding lex-ext-iff fst-conv
proof (intro conjI, force, rule disjI1, unfold length-map id, intro exI conjI, rule
i, rule i,
  intro allI impI)
  show wpo-s (?mss ! i) (?mts ! i) using σi i rels by simp
next
fix j
assume j < i
with i have j: j < length ?σ by auto
with wpo-ns-pre-mono[OF rel]
show ?mss ! j ⋃ ?mts ! j by blast
qed
moreover
obtain lb nlb where part: partition ((=) ?lb) ?σ = (lb, nlb) by force
hence mset-σ: mset ?σ = mset lb + mset nlb
  by (induct ?σ, auto)
let ?mlbs = map (!) ?ss lb
let ?mnlbs = map (!) ?ss nlb
let ?mlbt = map (!) ?ts lb
let ?mnlbt = map (!) ?ts nlb
have id1: mset ?mss = mset ?mnlbs + mset ?mlbs using mset-σ by auto
have id2: mset ?mts = mset ?mnlbt + mset ?mlbt using mset-σ by auto
from part lb-mem have lb: ?lb ∈ set lb by auto
have fst (mul-ext wpo ?mss ?mts)
  unfolding mul-ext-def Let-def fst-conv
proof (intro s-mul-extI-old, rule id1, rule id2)
from lb show mset ?mlbs ≠ {#} by auto
  {
    fix i
    assume i < length ?mnlbt
    then obtain j where id: ?mnlbs ! i = ?ss ! j ?mnlbt ! i = ?ts ! j j ∈ set nlb
  }

```

```

by auto
  with part have  $j \neq ?lb$  by auto
  hence  $?ss ! j = ?ts ! j$  by (auto simp: nth-append)
  thus  $(?mnlbs ! i, ?mnlbt ! i) \in WPO-NS$  unfolding id using wpo-ns-refl by
auto
}
fix u
assume  $u \in\# \text{mset } ?mlbt$ 
hence  $u = t$  using part by auto
moreover have  $s \in\# \text{mset } ?mlbs$  using lb by force
ultimately show  $\exists v. v \in\# \text{mset } ?mlbs \wedge (v,u) \in WPO-S$  using rels by
force
qed auto
ultimately show ?thesis unfolding wpo.simps[of ?s ?t] term.simps id prec-refl
using order-tag.exhaust by (auto simp: Let-def)
qed

theorem WPO-S-ctxt:  $(s,t) \in WPO-S \implies (\text{Fun } f (\text{bef } @ s \# \text{aft}), \text{Fun } f (\text{bef } @
t \# \text{aft})) \in WPO-S$ 
using wpo-s-mono by blast

theorem supt-subset-WPO-S:  $\{\triangleright\} \subseteq WPO-S$ 
using subterm-wpo-s by blast

theorem supreq-subset-WPO-NS:  $\{\trianglerighteq\} \subseteq WPO-NS$ 
using subterm-wpo-ns by blast

end
end

```

If we demand strong normalization of the underlying order and the precedence, then also WPO is strongly normalizing.

```

locale wpo-with-SN-assms = wpo-with-assms + SN-order-pair + precedence +
  constrains S :: ('f, 'v) term rel and NS :: -
  and prec :: 'f × nat ⇒ 'f × nat ⇒ bool × bool
  and prl :: 'f × nat ⇒ bool
  and ssimple :: bool
  and large :: 'f × nat ⇒ bool
  and c :: 'f × nat ⇒ order-tag
  and n :: nat
  and σσ :: 'f status
begin

```

**lemma** *Var-not-S[simp]*:  $(\text{Var } x, t) \notin S$

**proof**

```

  assume st:  $(\text{Var } x, t) \in S$ 
  from SN-imp-minimal[OF SN, rule-format, of undefined UNIV]
  obtain s where  $\bigwedge u. (s,u) \notin S$  by blast
  with subst-S[OF st, of λ -. s]

```

**show** *False* **by** *auto*  
**qed**

**lemma** *WPO-S-SN: SN WPO-S*

**proof** –

```

{
  fix t :: (f,v)term
  let ?S =  $\lambda x. \text{SN-on WPO-S } \{x\}$ 
  note iff = SN-on-all-reducts-SN-on-conv[of WPO-S]
  {
    fix x
    have ?S (Var x) unfolding iff[of Var x]
    proof (intro allI impI)
      fix s
      assume (Var x, s)  $\in$  WPO-S
      then have False by (cases s, auto simp: wpo.simps split: if-splits)
      then show ?S s ..
    qed
  } note var-SN = this
  have ?S t
  proof (induct t)
    case (Var x) show ?case by (rule var-SN)
  next
  case (Fun f ts)
  let ?Slist =  $\lambda f \text{ ys}. \forall i \in \text{set } (\sigma f). ?S (\text{ys } ! i)$ 
  let ?r3 =  $\{((f,ab), (g,ab')). ((c f = c g) \longrightarrow (?Slist f ab \wedge$ 
    (c f = Mul  $\longrightarrow$  fst (mul-ext wpo (map (!) ab) ( $\sigma f$ )) (map (!) ab') ( $\sigma$ 
g)))) \wedge
    (c f = Lex  $\longrightarrow$  fst (lex-ext wpo n (map (!) ab) ( $\sigma f$ )) (map (!) ab') ( $\sigma$ 
g)))))) \wedge
    ( $(c f \neq c g) \longrightarrow (\text{map } (!) \text{ ab } (\sigma f) \neq [] \wedge (\text{map } (!) \text{ ab}' (\sigma g)) = [])$ )}
  let ?r0 = lex-two  $\{(f,g). \text{fst } (\text{prc } f g)\} \{(f,g). \text{snd } (\text{prc } f g)\} ?r3$ 
  {
    fix ab
    {
      assume  $\exists S. S \ 0 = ab \wedge (\forall i. (S \ i, S \ (\text{Suc } i)) \in ?r3)$ 
      then obtain S where
        S0: S 0 = ab and
        SS:  $\forall i. (S \ i, S \ (\text{Suc } i)) \in ?r3$ 
      by auto
      let ?Sf =  $\lambda i. \text{fst } (\text{fst } (S \ i))$ 
      let ?Sn =  $\lambda i. \text{snd } (\text{fst } (S \ i))$ 
      let ?Sfn =  $\lambda i. \text{fst } (S \ i)$ 
      let ?Sts =  $\lambda i. \text{snd } (S \ i)$ 
      let ?Sts $\sigma$  =  $\lambda i. \text{map } (!) (\text{?Sts } i) (\sigma (\text{?Sfn } i))$ 
      have False
      proof (cases  $\forall i. c (\text{?Sfn } i) = \text{Mul}$ )
        case True
        let ?r' =  $\{((f,ys), (g,xs)).$ 

```

```

      (∀ yi ∈ set ((map (!) ys) (σ f))). SN-on WPO-S {yi}
      ∧ fst (mul-ext wpo (map (!) ys) (σ f)) (map (!) xs) (σ g)))}
    {
      fix i
      from True[rule-format, of i] and True[rule-format, of Suc i]
      and SS[rule-format, of i]
      have (S i, S (Suc i)) ∈ ?r' by auto
    }
  then have Hf: ¬ SN-on ?r' {S 0}
    unfolding SN-on-def by auto
  from mul-ext-SN[of wpo, rule-format, OF wpo-ns-refl]
  and wpo-compat wpo-s-imp-ns
  have tmp: SN {(ys, xs). (∀ y ∈ set ys. SN-on {(s, t). wpo-s s t} {y})} ∧ fst
(mul-ext wpo ys xs)}
    (is SN ?R) by blast
  have id: ?r' = inv-image ?R (λ (f,ys). map (!) ys) (σ f) by auto
  from SN-inv-image[OF tmp]
  have SN ?r' unfolding id .
  from SN-on-subset2[OF subset-UNIV[of {S 0}], OF this]
  have SN-on ?r' {(S 0)} .
  with Hf show ?thesis ..
next
case False note HMul = this
show ?thesis
proof (cases ∀ i. c (?Sfn i) = Lex)
  case True
  let ?r' = {(f,ys), (g,xs)}.
  (∀ yi ∈ set ((map (!) ys) (σ f))). SN-on WPO-S {yi}
  ∧ fst (lex-ext wpo n (map (!) ys) (σ f)) (map (!) xs) (σ g)))}
  {
    fix i
    from SS[rule-format, of i] True[rule-format, of i] True[rule-format,
of Suc i]
    have (S i, S (Suc i)) ∈ ?r' by auto
  }
  then have Hf: ¬ SN-on ?r' {S 0}
    unfolding SN-on-def by auto
  from wpo-compat have ∧ x y z. wpo-ns x y ⇒ wpo-s y z ⇒ wpo-s x
z by blast
  from lex-ext-SN[of wpo n, OF this]
  have tmp: SN {(ys, xs). (∀ y ∈ set ys. SN-on WPO-S {y})} ∧ fst (lex-ext
wpo n ys xs)}
    (is SN ?R) .
  have id: ?r' = inv-image ?R (λ (f,ys). map (!) ys) (σ f) by auto
  from SN-inv-image[OF tmp]
  have SN ?r' unfolding id .
  then have SN-on ?r' {(S 0)} unfolding SN-defs by blast
  with Hf show False ..
next

```

```

case False note HLex = this
from HMul and HLex
have  $\exists i. c (?Sfn\ i) \neq c (?Sfn\ (Suc\ i))$ 
proof (cases ?thesis, simp)
  case False
  then have  $T: \forall i. c (?Sfn\ i) = c (?Sfn\ (Suc\ i))$  by simp
  {
    fix i
    have  $c (?Sfn\ i) = c (?Sfn\ 0)$ 
    proof (induct i)
      case (Suc\ j) then show ?case by (simp add: T[rule-format, of\ j])
    qed simp
  }
  then show ?thesis using HMul HLex
  by (cases  $c (?Sfn\ 0)$ ) auto
qed
then obtain i where
  Hdiff:  $c (?Sfn\ i) \neq c (?Sfn\ (Suc\ i))$ 
  by auto
from Hdiff have Hf:  $?Sts\sigma\ (Suc\ i) = []$ 
  using SS[rule-format, of\ i] by auto
show ?thesis
proof (cases  $c (?Sfn\ (Suc\ i)) = c (?Sfn\ (Suc\ (Suc\ i)))$ )
  case False then show ?thesis using Hf and SS[rule-format, of\ Suc
i] by auto
next
  case True
  show ?thesis
  proof (cases  $c (?Sfn\ (Suc\ i))$ )
    case Mul
    with True and SS[rule-format, of\ Suc\ i]
    have  $fst\ (mul-ext\ wpo\ (?Sts\sigma\ (Suc\ i))\ (?Sts\sigma\ (Suc\ (Suc\ i))))$ 
      by auto
    with Hf and s-mul-ext-bottom-strict show ?thesis
      by (simp add: Let-def\ mul-ext-def\ s-mul-ext-bottom-strict)
  next
    case Lex
    with True and SS[rule-format, of\ Suc\ i]
    have  $fst\ (lex-ext\ wpo\ n\ (?Sts\sigma\ (Suc\ i))\ (?Sts\sigma\ (Suc\ (Suc\ i))))$ 
      by auto
    with Hf show ?thesis by (simp add: lex-ext-iff)
  qed
qed
qed
qed
qed
}
}
then have SN ?r3 unfolding SN-on-def by blast
have SN1:SN ?r0

```

```

proof (rule lex-two[OF - prc-SN ‹SN ?r3›])
  let ?r' = {(f,g). fst (prc f g)}
  let ?r = {(f,g). snd (prc f g)}
  {
    fix a1 a2 a3
    assume (a1,a2) ∈ ?r (a2,a3) ∈ ?r'
    then have (a1,a3) ∈ ?r'
      by (cases prc a1 a2, cases prc a2 a3, cases prc a1 a3,
        insert prc-compat[of a1 a2 - - a3], force)
  }
  then show ?r O ?r' ⊆ ?r' by auto
qed
let ?m = λ (f,ts). ((f,length ts), ((f, length ts), ts))
let ?r = {(a,b). (?m a, ?m b) ∈ ?r0}
have SN-r: SN ?r using SN-inv-image[OF SN1, of ?m] unfolding inv-image-def
by fast
  let ?SA = (λ x y. (x,y) ∈ S)
  let ?NSA = (λ x y. (x,y) ∈ NS)
  let ?rr = lex-two S NS ?r
  define rr where rr = ?rr
  from lex-two[OF compat-NS-S SN SN-r]
  have SN-rr: SN rr unfolding rr-def by auto
  let ?rrr = inv-image rr (λ (f,ts). (Fun f ts, (f,ts)))
  have SN-rrr: SN ?rrr
    by (rule SN-inv-image[OF SN-rr])
  let ?ind = λ (f,ts). ?Slist (f,length ts) ts → ?S (Fun f ts)
  have ?ind (f,ts)
proof (rule SN-induct[OF SN-rrr, of ?ind])
  fix fts
  assume ind: ∧ gss. (fts,gss) ∈ ?rrr ⇒ ?ind gss
  obtain f ts where Pair: fts = (f,ts) by force
  let ?f = (f,length ts)
  note ind = ind[unfolded Pair]
  show ?ind fts unfolding Pair split
proof (intro impI)
  assume ts: ?Slist ?f ts
  let ?t = Fun f ts
  show ?S ?t
proof (simp only: iff[of ?t], intro allI impI)
  fix s
  assume (?t,s) ∈ WPO-S
  then have ?t > s by simp
  then show ?S s
proof (induct s, simp add: var-SN)
  case (Fun g ss) note IH = this
  let ?s = Fun g ss
  let ?g = (g,length ss)
  from Fun have t-gr-s: ?t > ?s by auto
  show ?S ?s

```

```

proof (cases  $\exists i \in \text{set } (\sigma \ ?f). \ ts \ ! \ i \succeq \ ?s$ )
  case True
  then obtain i where  $i \in \text{set } (\sigma \ ?f)$  and  $ge: \ ts \ ! \ i \succeq \ ?s$  by auto
  with ts have  $\ ?S \ (ts \ ! \ i)$  by auto
  show  $\ ?S \ ?s$ 
  proof (unfold iff[of ?s], intro allI impI)
    fix u
    assume  $(\ ?s, u) \in \text{WPO-S}$ 
    with wpo-compat ge have  $u: \ ts \ ! \ i \succ \ u$  by blast
    with  $\langle \ ?S \ (ts \ ! \ i) \rangle[\text{unfolded iff[of } ts \ ! \ i]]$ 
    show  $\ ?S \ u$  by simp
  qed
next
  case False note oFalse = this
  from wpo-s-imp-NS[OF t-gr-s]
  have t-NS-s: (?t, ?s) ∈ NS .
  show ?thesis
  proof (cases  $(\ ?t, ?s) \in S$ )
    case True
    then have  $((f, ts), (g, ss)) \in \ ?rrr$  unfolding rr-def by auto
    with ind have ind: ?ind (g, ss) by auto
    {
      fix i
      assume  $i: \ i \in \text{set } (\sigma \ ?g)$ 
      have  $\ ?s \succeq \ ss \ ! \ i$  by (rule subterm-wpo-ns-arg[OF i])
      with t-gr-s have  $ts: \ ?t \succ \ ss \ ! \ i$  using wpo-compat by blast
      have  $\ ?S \ (ss \ ! \ i)$  using IH(1)[OF  $\sigma E$ [OF i] ts] by auto
    }
    note SN-ss = this
    from ind SN-ss show ?thesis by auto
  next
  case False
  with t-NS-s oFalse
  have id: (?t, ?s) ∈ S = False (?t, ?s) ∈ NS = True by simp-all
  let  $\ ?ls = \text{length } ss$ 
  let  $\ ?lt = \text{length } ts$ 
  let  $\ ?f = (f, ?lt)$ 
  let  $\ ?g = (g, ?ls)$ 
  obtain s1 ns1 where prc1: prc ?f ?g = (s1, ns1) by force
  note t-gr-s = t-gr-s[unfolded wpo.simps[of ?t ?s],
    unfolded term.simps id if-True if-False prc1 split]
  from oFalse t-gr-s have f-ge-g: ns1
  by (cases ?thesis, auto)
  from oFalse t-gr-s f-ge-g have small-ss:  $\forall i \in \text{set } (\sigma \ ?g). \ ?t \succ \ ss \ ! \ i$ 
  by (cases ?thesis, auto)
  with Fun  $\sigma E$ [of - g ss] have ss-S: ?Slist ?g ss by auto
  show ?thesis
  proof (cases s1)
    case True
    then have  $((f, ts), (g, ss)) \in \ ?r$  by (simp add: prc1)

```

```

with t-NS-s have ((f,ts),(g,ss)) ∈ ?rrr unfolding rr-def by auto
with ind have ?ind (g,ss) by auto
with ss-S show ?thesis by auto
next
case False
consider (Diff) c ?f ≠ c ?g | (Lex) c ?f = Lex c ?g = Lex | (Mul)
c ?f = Mul c ?g = Mul
by (cases c ?f; cases c ?g, auto)
thus ?thesis
proof cases
case Diff
with False oFalse f-ge-g t-gr-s small-ss prc1 t-NS-s
have ((f,ts),(g,ss)) ∈ ?rrr unfolding rr-def by (cases c ?f; cases
c ?g, auto)
with ind have ?ind (g,ss) using Pair by auto
with ss-S show ?thesis by simp
next
case Lex
from False oFalse t-gr-s small-ss f-ge-g Lex
have lex: fst (lex-ext wpo n (map (!) ts) (σ ?f)) (map (!) ss)
(σ ?g))
by auto
from False lex ts f-ge-g Lex have ((f,ts),(g,ss)) ∈ ?r
by (simp add: prc1)
with t-NS-s have ((f,ts),(g,ss)) ∈ ?rrr unfolding rr-def by auto
with ind have ?ind (g,ss) by auto
with ss-S show ?thesis by auto
next
case Mul
from False oFalse t-gr-s small-ss f-ge-g Mul
have mul: fst (mul-ext wpo (map (!) ts) (σ ?f)) (map (!) ss) (σ
?g))
by auto
from False mul ts f-ge-g Mul have ((f,ts),(g,ss)) ∈ ?r
by (simp add: prc1)
with t-NS-s have ((f,ts),(g,ss)) ∈ ?rrr unfolding rr-def by auto
with ind have ?ind (g,ss) by auto
with ss-S show ?thesis by auto
qed
qed
qed
qed
qed
qed
qed
qed
with Fun show ?case using σE[of - f ts] by simp
qed
}

```

```

from SN-I[OF this]
show SN {(s::('f, 'v)term, t). fst (wpo s t)} .
qed

theorem wpo-SN-order-pair: SN-order-pair WPO-S WPO-NS
proof –
  interpret order-pair WPO-S WPO-NS by (rule wpo-order-pair)
  show ?thesis
  proof
    show SN WPO-S using WPO-S-SN .
  qed
qed

end
end

```

## 6 The Recursive Path Order as an instance of WPO

This theory defines the recursive path order (RPO) that given two terms provides two Booleans, whether the terms can be strictly or non-strictly oriented. It is proven that RPO is an instance of WPO, and hence, carries over all the nice properties of WPO immediately.

```

theory RPO
  imports
    WPO
  begin

  context
    fixes pr :: 'f × nat ⇒ 'f × nat ⇒ bool × bool
      and prl :: 'f × nat ⇒ bool
      and c :: 'f × nat ⇒ order-tag
      and n :: nat
  begin

  fun rpo :: ('f, 'v) term ⇒ ('f, 'v) term ⇒ bool × bool
    where
      rpo (Var x) (Var y) = (False, x = y) |
      rpo (Var x) (Fun g ts) = (False, ts = [] ∧ prl (g,0)) |
      rpo (Fun f ss) (Var y) = (let con = (∃ s ∈ set ss. snd (rpo s (Var y))) in
      (con,con)) |
      rpo (Fun f ss) (Fun g ts) = (
      if (∃ s ∈ set ss. snd (rpo s (Fun g ts)))
      then (True, True)
      else (let (prs,prns) = pr (f,length ss) (g,length ts) in
      if prns ∧ (∀ t ∈ set ts. fst (rpo (Fun f ss) t))
      then if prs
        then (True, True)
        else if c (f,length ss) = Lex ∧ c (g,length ts) = Lex

```

```

      then lex-ext rpo n ss ts
      else if c (f,length ss) = Mul  $\wedge$  c (g,length ts) = Mul
        then mul-ext rpo ss ts
        else (length ss  $\neq$  0  $\wedge$  length ts = 0, length ts = 0)
    else (False,False))
end

locale rpo-with-assms = precedence prc prl
  for prc :: 'f  $\times$  nat  $\Rightarrow$  'f  $\times$  nat  $\Rightarrow$  bool  $\times$  bool
  and prl :: 'f  $\times$  nat  $\Rightarrow$  bool
  and c :: 'f  $\times$  nat  $\Rightarrow$  order-tag
  and n :: nat
begin

sublocale wpo-with-SN-assms n {} UNIV prc prl full-status c False  $\lambda$  -. False
  by (unfold-locales, auto simp: refl-on-def trans-def simple-arg-pos-def irrefl-def)

abbreviation rpo-pr  $\equiv$  rpo prc prl c n
abbreviation rpo-s  $\equiv$   $\lambda$  s t. fst (rpo-pr s t)
abbreviation rpo-ns  $\equiv$   $\lambda$  s t. snd (rpo-pr s t)

lemma rpo-eq-wpo: rpo-pr s t = wpo s t
proof –
  note_simps = wpo_simps
  show ?thesis
  proof (induct s t rule: rpo.induct[of - prc prl c n])
    case (1 x y)
    then show ?case by (simp add:_simps)
  next
    case (2 x g ts)
    then show ?case by (auto simp:_simps)
  next
    case (3 f ss y)
    then show ?case by (auto simp:_simps[of Fun f ss Var y] Let-def set-conv-nth)
  next
    case IH: (4 f ss g ts)
    have id:  $\bigwedge$  s. (s  $\in$  {}) = False  $\wedge$  s. (s  $\in$  UNIV) = True
    and ( $\exists$  i  $\in$  {0.. $\text{length}$  ss}. wpo-ns (ss ! i) t) = ( $\exists$  si  $\in$  set ss. wpo-ns si t)
    by (auto, force simp: set-conv-nth)
    have id': map (!) ss ( $\sigma$  (f, length ss)) = ss for f ss by (intro nth-equalityI, auto)
    have ex: ( $\exists$  i  $\in$  set ( $\sigma$  (f, length ss)). wpo-ns (ss ! i) (Fun g ts)) = ( $\exists$  si  $\in$  set ss. rpo-ns si (Fun g ts))
    using IH(1) unfolding set-conv-nth by auto
    obtain prs prns where prc: prc (f, length ss) (g, length ts) = (prs, prns) by force
    show ?case
    unfolding rpo_simps_simps[of Fun f ss Fun g ts] term_simps id id' if-False if-True

```

*Let-def ex prc split*

**proof** (*rule sym, rule if-cong[OF refl refl], rule if-cong[OF conj-cong[OF refl] if-cong[OF refl refl if-cong[OF refl - if-cong]] refl]*)  
**assume**  $\neg (\exists si \in set\ ss. rpo\text{-}ns\ si\ (Fun\ g\ ts))$   
**note**  $IH = IH(2-)[OF\ this\ prc[symmetric]\ refl]$   
**from**  $IH(1)$  **show**  $(\forall j \in set\ (\sigma\ (g, length\ ts)). wpo\text{-}s\ (Fun\ f\ ss)\ (ts\ !\ j)) =$   
 $(\forall t \in set\ ts. rpo\text{-}s\ (Fun\ f\ ss)\ t)$   
**unfolding** *set-conv-nth* **by** *auto*  
**assume**  $prns \wedge (\forall t \in set\ ts. rpo\text{-}s\ (Fun\ f\ ss)\ t) \neg prs$   
**note**  $IH = IH(2-)[OF\ this]$   
{  
**assume**  $c\ (f, length\ ss) = Lex \wedge c\ (g, length\ ts) = Lex$   
**from**  $IH(1)[OF\ this]$   
**show**  $lex\text{-}ext\ wpo\ n\ ss\ ts = lex\text{-}ext\ rpo\text{-}pr\ n\ ss\ ts$   
**by** (*intro lex-ext-cong, auto*)  
}  
}  
**assume**  $\neg (c\ (f, length\ ss) = Lex \wedge c\ (g, length\ ts) = Lex) \ c\ (f, length\ ss)$   
 $= Mul \wedge c\ (g, length\ ts) = Mul$   
**from**  $IH(2)[OF\ this]$   
**show**  $mul\text{-}ext\ wpo\ ss\ ts = mul\text{-}ext\ rpo\text{-}pr\ ss\ ts$   
**by** (*intro mul-ext-cong, auto*)  
}  
**qed** *auto*  
**qed**  
**qed**

**abbreviation**  $RPO\text{-}S \equiv \{(s,t). rpo\text{-}s\ s\ t\}$   
**abbreviation**  $RPO\text{-}NS \equiv \{(s,t). rpo\text{-}ns\ s\ t\}$

**theorem** *RPO-SN-order-pair: SN-order-pair RPO-S RPO-NS*  
**unfolding** *rpo-eq-wpo* **by** (*rule wpo-SN-order-pair*)

**theorem** *RPO-S-subst:  $(s,t) \in RPO\text{-}S \implies (s \cdot \sigma, t \cdot \sigma) \in RPO\text{-}S$  for  $\sigma :: (f, 'a)subst$*   
**using** *WPO-S-subst* **unfolding** *rpo-eq-wpo* .

**theorem** *RPO-NS-subst:  $(s,t) \in RPO\text{-}NS \implies (s \cdot \sigma, t \cdot \sigma) \in RPO\text{-}NS$  for  $\sigma :: (f, 'a)subst$*   
**using** *WPO-NS-subst* **unfolding** *rpo-eq-wpo* .

**theorem** *RPO-NS-ctxt:  $(s,t) \in RPO\text{-}NS \implies (Fun\ f\ (bef\ @\ s\ \#\ aft), Fun\ f\ (bef\ @\ t\ \#\ aft)) \in RPO\text{-}NS$*   
**using** *WPO-NS-ctxt* **unfolding** *rpo-eq-wpo* .

**theorem** *RPO-S-ctxt:  $(s,t) \in RPO\text{-}S \implies (Fun\ f\ (bef\ @\ s\ \#\ aft), Fun\ f\ (bef\ @\ t\ \#\ aft)) \in RPO\text{-}S$*   
**using** *WPO-S-ctxt* **unfolding** *rpo-eq-wpo* **by** *auto*

**theorem** *RPO-S-subset-RPO-NS*:  $RPO-S \subseteq RPO-NS$   
**using** *WPO-S-subset-WPO-NS unfolding rpo-eq-wpo* .

**theorem** *supt-subset-RPO-S*:  $\{\triangleright\} \subseteq RPO-S$   
**using** *supt-subset-WPO-S unfolding rpo-eq-wpo by auto*

**theorem** *supteq-subset-RPO-NS*:  $\{\triangleright\} \subseteq RPO-NS$   
**using** *supteq-subset-WPO-NS unfolding rpo-eq-wpo by auto*

**end**  
**end**

## 7 The Lexicographic Path Order as an instance of WPO

We first directly define the strict- and non-strict lexicographic path orders (LPO) w.r.t. some precedence, and then show that it is an instance of WPO. For this instance we use the trivial reduction pair in WPO ( $\emptyset$ , UNIV) and the status is the full one, i.e., taking parameters  $[0, \dots, n-1]$  for each n-ary symbol.

```

theory LPO
  imports
    WPO
  begin

  context
    fixes pr :: ('f × nat ⇒ 'f × nat ⇒ bool × bool)
      and prl :: 'f × nat ⇒ bool
      and n :: nat
  begin
  fun lpo :: ('f, 'v) term ⇒ ('f, 'v) term ⇒ bool × bool
    where
      lpo (Var x) (Var y) = (False, x = y) |
      lpo (Var x) (Fun g ts) = (False, ts = [] ∧ prl (g, 0)) |
      lpo (Fun f ss) (Var y) = (let con = (∃ s ∈ set ss. snd (lpo s (Var y))) in
      (con, con)) |
      lpo (Fun f ss) (Fun g ts) = (
      if (∃ s ∈ set ss. snd (lpo s (Fun g ts)))
      then (True, True)
      else (let (prs, prns) = pr (f, length ss) (g, length ts) in
      if prns ∧ (∀ t ∈ set ts. fst (lpo (Fun f ss) t))
      then if prs
        then (True, True)
        else lex-ext lpo n ss ts
      else (False, False)))
  
```

**end**

**locale** *lpo-with-assms* = *precedence prc prl*  
  **for** *prc* :: 'f × nat ⇒ 'f × nat ⇒ bool × bool  
  **and** *prl* :: 'f × nat ⇒ bool  
  **and** *n* :: nat  
**begin**

**sublocale** *wpo-with-SN-assms* *n* {} *UNIV prc prl full-status λ -. Lex False λ -. False*  
  **by** (*unfold-locales, auto simp: refl-on-def trans-def simple-arg-pos-def irrefl-def*)

**abbreviation** *lpo-pr* ≡ *lpo prc prl n*  
**abbreviation** *lpo-s* ≡ λ *s t*. *fst (lpo-pr s t)*  
**abbreviation** *lpo-ns* ≡ λ *s t*. *snd (lpo-pr s t)*

**lemma** *lpo-eq-wpo*: *lpo-pr s t = wpo s t*

**proof** –

**note** *simps = wpo.simps*  
  **show** *?thesis*  
  **proof** (*induct s t rule: lpo.induct[of - prc prl n]*)  
    **case** (1 *x y*)  
    **then show** *?case by (simp add: simps)*  
  **next**  
    **case** (2 *x g ts*)  
    **then show** *?case by (auto simp: simps)*  
  **next**  
    **case** (3 *f ss y*)  
    **then show** *?case by (auto simp: simps[of Fun f ss Var y] Let-def set-conv-nth)*  
  **next**  
    **case** *IH: (4 f ss g ts)*  
    **have** *id: ∧ s. (s ∈ {}) = False ∧ s. (s ∈ UNIV) = True*  
      **and** (∃ *i* ∈ {0..<length *ss*}. *wpo-ns (ss ! i) t*) = (∃ *si* ∈ *set ss*. *wpo-ns si t*)  
      **by** (*auto, force simp: set-conv-nth*)  
    **have** *id': map (!) ss (σ (f, length *ss*)) = ss for f ss by (intro nth-equalityI, auto)*  
    **have** *ex: (∃ i ∈ set (σ (f, length *ss*)). wpo-ns (ss ! i) (Fun g ts)) = (∃ si ∈ set ss. lpo-ns si (Fun g ts))*  
      **using** *IH(1) unfolding set-conv-nth by auto*  
    **obtain** *prs prns where prc: prc (f, length *ss*) (g, length *ts*) = (prs, prns) by force*  
    **have** *lex: (Lex = Lex ∧ Lex = Lex) = True by simp*  
    **show** *?case*  
    **unfolding** *lpo.simps simps[of Fun f ss Fun g ts] term.simps id id' if-False if-True lex*  
      *Let-def ex prc split*  
    **proof** (*rule sym, rule if-cong[OF refl refl], rule if-cong[OF conj-cong[OF refl] if-cong[OF refl refl] refl]*)

**assume**  $\neg (\exists si \in set\ ss. lpo\text{-}ns\ si\ (Fun\ g\ ts))$   
**note**  $IH = IH(2-)[OF\ this\ prc[symmetric]\ refl]$   
**from**  $IH(1)$  **show**  $(\forall j \in set\ (\sigma\ (g,\ length\ ts)). wpo\text{-}s\ (Fun\ f\ ss)\ (ts\ !\ j)) =$   
 $(\forall t \in set\ ts. lpo\text{-}s\ (Fun\ f\ ss)\ t)$   
**unfolding** *set-conv-nth* **by** *auto*  
**assume**  $prns \wedge (\forall t \in set\ ts. lpo\text{-}s\ (Fun\ f\ ss)\ t) \neg prs$   
**note**  $IH = IH(2-)[OF\ this]$   
**show**  $lex\text{-}ext\ wpo\ n\ ss\ ts = lex\text{-}ext\ lpo\text{-}pr\ n\ ss\ ts$   
**using**  $IH$  **by** (*intro lex-ext-cong, auto*)  
**qed**  
**qed**  
**qed**

**abbreviation**  $LPO\text{-}S \equiv \{(s,t). lpo\text{-}s\ s\ t\}$   
**abbreviation**  $LPO\text{-}NS \equiv \{(s,t). lpo\text{-}ns\ s\ t\}$

**theorem** *LPO-SN-order-pair*: *SN-order-pair LPO-S LPO-NS*  
**unfolding** *lpo-eq-wpo* **by** (*rule wpo-SN-order-pair*)

**theorem** *LPO-S-subst*:  $(s,t) \in LPO\text{-}S \implies (s \cdot \sigma, t \cdot \sigma) \in LPO\text{-}S$  **for**  $\sigma ::$   
 $(f, a)subst$   
**using** *WPO-S-subst* **unfolding** *lpo-eq-wpo* .

**theorem** *LPO-NS-subst*:  $(s,t) \in LPO\text{-}NS \implies (s \cdot \sigma, t \cdot \sigma) \in LPO\text{-}NS$  **for**  $\sigma ::$   
 $(f, a)subst$   
**using** *WPO-NS-subst* **unfolding** *lpo-eq-wpo* .

**theorem** *LPO-NS-ctxt*:  $(s,t) \in LPO\text{-}NS \implies (Fun\ f\ (bef\ @\ s\ \#\ aft), Fun\ f\ (bef$   
 $@\ t\ \#\ aft)) \in LPO\text{-}NS$   
**using** *WPO-NS-ctxt* **unfolding** *lpo-eq-wpo* .

**theorem** *LPO-S-ctxt*:  $(s,t) \in LPO\text{-}S \implies (Fun\ f\ (bef\ @\ s\ \#\ aft), Fun\ f\ (bef\ @\ t$   
 $\#\ aft)) \in LPO\text{-}S$   
**using** *WPO-S-ctxt* **unfolding** *lpo-eq-wpo* **by** *auto*

**theorem** *LPO-S-subset-LPO-NS*:  $LPO\text{-}S \subseteq LPO\text{-}NS$   
**using** *WPO-S-subset-WPO-NS* **unfolding** *lpo-eq-wpo* .

**theorem** *supt-subset-LPO-S*:  $\{\triangleright\} \subseteq LPO\text{-}S$   
**using** *supt-subset-WPO-S* **unfolding** *lpo-eq-wpo* **by** *auto*

**theorem** *supteq-subset-LPO-NS*:  $\{\triangleright\} \subseteq LPO\text{-}NS$   
**using** *supteq-subset-WPO-NS* **unfolding** *lpo-eq-wpo* **by** *auto*

**end**

**end**

## 8 The Knuth–Bendix Order as an instance of WPO

Making the Knuth–Bendix an instance of WPO is more complicated than in the case of RPO and LPO, because of syntactic and semantic differences. We face the two main challenges in two different theories and sub-sections.

### 8.1 Aligning least elements

In all of RPO, LPO and WPO there is the concept of a minimal term, e.g., a constant term  $c$  where  $c$  is least in precedence among *all function symbols*. By contrast, in KBO a constant  $c$  is minimal if it has minimal weight and has least precedence *among all constants of minimal weight*.

In this theory we prove that for any KBO one can modify the precedence in a way that least constants  $c$  also have least precedence among *all function symbols*, without changing the defined order. Hence, afterwards it will be simpler to relate such a KBO to WPO.

**theory** *KBO-Transformation*

**imports** *WPO Knuth-Bendix-Order.KBO*

**begin**

**context** *admissible-kbo*

**begin**

**lemma** *weight-w0-unary:*

**assumes**  $*$ :  $\text{weight } t = w0 \ t = \text{Fun } f \ ts \ ts = t1 \ \# \ ts'$

**shows**  $ts' = [] \ w \ (f,1) = 0$

**proof** –

**have**  $w0 + \text{sum-list } (\text{map } \text{weight } ts') \leq \text{weight } t1 + \text{sum-list } (\text{map } \text{weight } ts')$

**by** (*rule add-right-mono, rule weight-w0*)

**also have**  $\dots = \text{sum-list } (\text{map } \text{weight } ts)$  **unfolding**  $*$  **by** *simp*

**also have**  $\dots \leq \text{sum-list } (\text{map } \text{weight } (\text{scf-list } (\text{scf } (f, \text{length } ts)) \ ts))$

**by** (*rule sum-list-scf-list, insert scf, auto*)

**finally have**  $w \ (f, \text{length } ts) + w0 + \text{sum-list } (\text{map } \text{weight } ts') \leq \text{weight } t$

**unfolding**  $*$  **by** *simp*

**with**  $*(1)$  **have**  $\text{sum-list } (\text{map } \text{weight } ts') = 0$  **and**  $wf: w \ (f, \text{length } ts) = 0$

**by** *auto*

**with** *weight-gt-0* **show**  $ts': ts' = []$  **by** (*cases ts', auto*)

**with**  $wf$  **show**  $w \ (f,1) = 0$  **using**  $*$  **by** *auto*

**qed**

**definition** *lConsts* ::  $(f \times \text{nat})\text{set}$  **where**  $lConsts = \{ (f,0) \mid f. \text{least } f \}$

**definition** *pr-strict'* **where**  $\text{pr-strict}' \ f \ g = (f \notin lConsts \wedge (\text{pr-strict } f \ g \vee g \in lConsts))$

**definition** *pr-weak'* **where**  $\text{pr-weak}' \ f \ g = ((f \notin lConsts \wedge \text{pr-weak } f \ g) \vee g \in lConsts)$

**lemma** *admissible-kbo'*: *admissible-kbo*  $w \ w0$  *pr-strict'* *pr-weak'* *least scf*

```

apply (unfold-locales)
subgoal by (rule w0)
subgoal by (rule w0)
subgoal for  $f\ g\ n$  using  $\text{adm}[of\ f\ g\ n]$  unfolding pr-weak'-def by (auto simp:
lConsts-def)
subgoal for  $f$  using  $\text{least}[of\ f]$  unfolding pr-weak'-def lConsts-def by auto
subgoal by (rule scf)
subgoal for  $f$  using  $\text{pr-weak-refl}[of\ f]$  unfolding pr-weak'-def by auto
subgoal for  $f\ g\ h$  using  $\text{pr-weak-trans}[of\ f\ g\ h]$  unfolding pr-weak'-def by auto
subgoal for  $f\ g$  using  $\text{pr-strict}[of\ f\ g]$  unfolding pr-strict'-def pr-weak'-def by
auto
proof –
  show  $SN\ \{(x, y).\ \text{pr-strict}'\ x\ y\}$  (is  $SN\ ?R$ )
  proof
    fix  $f$ 
    assume  $\forall i. (f\ i, f\ (Suc\ i)) \in ?R$ 
    hence  $\text{steps}:\ \bigwedge i. (f\ i, f\ (Suc\ i)) \in ?R$  by blast
    have  $f\ i \notin \text{lConsts}$  for  $i$  using  $\text{steps}[of\ i]$  unfolding pr-strict'-def by auto
    hence  $\text{pr-strict}\ (f\ i)\ (f\ (Suc\ i))$  for  $i$  using  $\text{steps}[of\ i]$  unfolding pr-strict'-def
by auto
    with pr-SN show False by auto
  qed
qed

lemma least-pr-weak':  $\text{least}\ f \implies \text{pr-weak}'\ g\ (f, 0)$  unfolding lConsts-def pr-weak'-def
by auto

lemma least-pr-weak'-trans:  $\text{least}\ f \implies \text{pr-weak}'\ (f, 0)\ g \implies \text{least}\ (fst\ g) \wedge \text{snd}\ g = 0$ 
unfolding lConsts-def pr-weak'-def by auto

context
begin
interpretation kbo': admissible-kbo w w0 pr-strict' pr-weak' least scf
by (rule admissible-kbo')

lemma kbo'-eq-kbo:  $\text{kbo}'.\text{kbo}\ s\ t = \text{kbo}\ s\ t$ 
proof (induct s t rule: kbo.induct)
  case ( $1\ s\ t$ )
  note  $\text{simps} = \text{kbo}.\text{simps}[of\ s\ t]\ \text{kbo}'.\text{kbo}.\text{simps}[of\ s\ t]$ 
  show  $?case$  unfolding simps
    apply (intro if-cong refl, intro term.case-cong refl)
  proof –
    fix  $f\ ss\ g\ ts$ 
    assume  $*$ :  $\text{vars-term-ms}\ (SCF\ t) \subseteq \# \text{vars-term-ms}\ (SCF\ s) \wedge \text{weight}\ t \leq \text{weight}\ s$ 
     $\neg \text{weight}\ t < \text{weight}\ s$ 
    and  $s$ :  $s = \text{Fun}\ f\ ss$ 
    and  $t$ :  $t = \text{Fun}\ g\ ts$ 

```

```

let ?g = (g,length ts)
let ?f = (f,length ss)
have IH: (if pr-strict ?f ?g then (True, True)
  else if pr-weak ?f ?g then lex-ext-unbounded kbo ss ts else (False, False))
  = (if pr-strict ?f ?g then (True, True)
    else if pr-weak ?f ?g then lex-ext-unbounded kbo'.kbo ss ts else (False, False))
  by (intro if-cong refl lex-ext-unbounded-cong, insert 1[OF * s t], auto)
let ?P = pr-strict' ?f ?g = pr-strict ?f ?g ∧ (¬ pr-strict ?f ?g → pr-weak' ?f
?g = pr-weak ?f ?g)
show (if pr-strict' ?f ?g then (True, True)
  else if pr-weak' ?f ?g then lex-ext-unbounded kbo'.kbo ss ts else (False, False))
=
  (if pr-strict ?f ?g then (True, True)
  else if pr-weak ?f ?g then lex-ext-unbounded kbo ss ts else (False, False))
proof (cases ?P)
  case True
  thus ?thesis unfolding IH by auto
next
  case notP: False
  hence fgC: ?f ∈ lConsts ∨ ?g ∈ lConsts unfolding pr-strict'-def pr-weak'-def
by auto
  hence weight: weight s = w0 weight t = w0 using * unfolding lConsts-def
least s t by auto
  show ?thesis
  proof (cases ss = [] ∧ ts = [])
    case empty: True
    with weight have w ?f = w0 w ?g = w0 unfolding s t by auto
    with empty have ?P unfolding pr-strict'-def pr-weak'-def using pr-weak-trans[of
- (g,0) (f,0)]
      pr-weak-trans[of - (f,0) (g,0)]
      by (auto simp: lConsts-def pr-strict least)
    with notP show ?thesis by blast
  next
  case False
  {
  fix f and t :: ('f,'a)term and t1 ts' ts and g
  assume *: weight t = w0 t = Fun f ts ts = t1 # ts'
  from weight-w0-unary[OF this]
  have ts': ts' = [] and w: w (f,1) = 0 .
  from adm[OF w] ts'
  have pr-weak (f, Suc (length ts')) g by (cases g, auto)
  } note unary = this
  from fgC have ss = [] ∨ ts = [] unfolding lConsts-def least by auto
  thus ?thesis
proof
  assume ss: ss = []
  with False obtain t1 ts' where ts: ts = t1 # ts' by (cases ts, auto)
  show ?thesis unfolding ss ts using unary[OF weight(2) t ts]
  by (simp add: lex-ext-unbounded.simps pr-strict'-def lConsts-def pr-strict)

```

```

next
  assume ts: ts = []
  with False obtain s1 ss' where ss: ss = s1 # ss' by (cases ss, auto)
  show ?thesis unfolding ss ts using unary[OF weight(1) s ss]
    by (simp add: lex-ext-unbounded.simps pr-strict'-def pr-weak'-def
lConsts-def pr-strict)
  qed
qed
qed
qed
qed
end
end
end

```

## 8.2 A restricted equality between KBO and WPO

The remaining difficulty to make KBO an instance of WPO is the different treatment of lexicographic comparisons, which is unrestricted in KBO, but there is a length-restriction in WPO. Therefore we will only show that KBO is an instance of WPO if we compare terms with bounded arity.

This restriction does however not prohibit us from lifting properties of WPO to KBO. For instance, for several properties one can choose a large-enough bound restriction of WPO, since there are only finitely many arities occurring in a property.

**theory** *KBO-as-WPO*

**imports**

*WPO*

*KBO-Transformation*

**begin**

**definition** *bounded-arity* :: *nat*  $\Rightarrow$  (*f*  $\times$  *nat*)*set*  $\Rightarrow$  *bool* **where**  
*bounded-arity b F* = ( $\forall$  (*f,n*)  $\in$  *F*. *n*  $\leq$  *b*)

**lemma** *finite-funas-term[simp,intro]*: *finite (funas-term t)*  
**by** (*induct t, auto*)

**context** *weight-fun* **begin**

**definition** *weight-le s t*  $\equiv$   
*(vars-term-ms (SCF s)  $\subseteq$ # vars-term-ms (SCF t)  $\wedge$  weight s  $\leq$  weight t)*

**definition** *weight-less s t*  $\equiv$   
*(vars-term-ms (SCF s)  $\subseteq$ # vars-term-ms (SCF t)  $\wedge$  weight s  $<$  weight t)*

**lemma** *weight-le-less-iff*: *weight-le s t  $\implies$  weight-less s t  $\longleftrightarrow$  weight s  $<$  weight t*  
**by** (*auto simp: weight-le-def weight-less-def*)

**lemma** *weight-less-iff*:  $\text{weight-less } s \ t \implies \text{weight-le } s \ t \wedge \text{weight } s < \text{weight } t$   
**by** (*auto simp: weight-le-def weight-less-def*)

**abbreviation** *weight-NS*  $\equiv \{(t,s). \text{weight-le } s \ t\}$

**abbreviation** *weight-S*  $\equiv \{(t,s). \text{weight-less } s \ t\}$

**lemma** *weight-le-mono-one*:

**assumes** *S*:  $\text{weight-le } s \ t$

**shows**  $\text{weight-le } (\text{Fun } f \ (ss1 \ @ \ s \ \# \ ss2)) \ (\text{Fun } f \ (ss1 \ @ \ t \ \# \ ss2))$  (**is**  $\text{weight-le } ?s \ ?t$ )

**proof** –

**from** *S* **have**  $w: \text{weight } s \leq \text{weight } t$  **and**  $v: \text{vars-term-ms } (\text{SCF } s) \subseteq\# \text{vars-term-ms } (\text{SCF } t)$

**by** (*auto simp: weight-le-def*)

**have**  $v': \text{vars-term-ms } (\text{SCF } ?s) \subseteq\# \text{vars-term-ms } (\text{SCF } ?t)$

**using** *mset-replicate-mono[OF v]* **by** *simp*

**have**  $w': \text{weight } ?s \leq \text{weight } ?t$  **using** *sum-list-replicate-mono[OF w]* **by** *simp*

**from**  $v' \ w'$  **show** *?thesis* **by** (*auto simp: weight-le-def*)

**qed**

**lemma** *weight-le-ctxt*:  $\text{weight-le } s \ t \implies \text{weight-le } (C \langle s \rangle) \ (C \langle t \rangle)$

**by** (*induct C, auto intro: weight-le-mono-one*)

**lemma** *SCF-stable*:

**assumes**  $\text{vars-term-ms } (\text{SCF } s) \subseteq\# \text{vars-term-ms } (\text{SCF } t)$

**shows**  $\text{vars-term-ms } (\text{SCF } (s \cdot \sigma)) \subseteq\# \text{vars-term-ms } (\text{SCF } (t \cdot \sigma))$

**unfolding** *scf-term-subst*

**using** *vars-term-ms-subst-mono[OF assms]*.

**lemma** *SN-weight-S*:  $\text{SN } \text{weight-S}$

**proof**–

**from** *wf-inv-image[OF wf-less]*

**have**  $wf: wf \ \{(s,t). \text{weight } s < \text{weight } t\}$  **by** (*auto simp: inv-image-def*)

**show** *?thesis*

**by** (*unfold SN-iff-wf, rule wf-subset[OF wf], auto simp: weight-less-def*)

**qed**

**lemma** *weight-less-imp-le*:  $\text{weight-less } s \ t \implies \text{weight-le } s \ t$  **by** (*simp add: weight-less-def weight-le-def*)

**lemma** *weight-le-Var-Var*:  $\text{weight-le } (\text{Var } x) \ (\text{Var } y) \longleftrightarrow x = y$

**by** (*auto simp: weight-le-def*)

**end**

**context** *kbo* **begin**

**lemma** *kbo-altdef*:

```

kbo s t = (if weight-le t s
then if weight-less t s
then (True, True)
else (case s of
  Var y => (False, (case t of Var x => x = y | Fun g ts => ts = [] ^ least g))
| Fun f ss => (case t of
  Var x => (True, True)
| Fun g ts => if pr-strict (f, length ss) (g, length ts)
then (True, True)
else if pr-weak (f, length ss) (g, length ts)
then lex-ext-unbounded kbo ss ts
else (False, False)))
else (False, False))
by (simp add: weight-le-less-iff weight-le-def)

```

**end**

**context** *admissible-kbo* **begin**

**lemma** *weight-le-stable*:

**assumes** *weight-le s t*

**shows** *weight-le (s · σ) (t · σ)*

**using** *assms weight-stable-le SCF-stable* **by** (*auto simp: weight-le-def*)

**lemma** *weight-less-stable*:

**assumes** *weight-less s t*

**shows** *weight-less (s · σ) (t · σ)*

**using** *assms weight-stable-lt SCF-stable* **by** (*auto simp: weight-less-def*)

**lemma** *simple-arg-pos-weight: simple-arg-pos weight-NS (f,n) i*

**unfolding** *simple-arg-pos-def*

**proof** (*intro allI impI, unfold snd-conv fst-conv*)

**fix** *ts :: ('f,'a)term list*

**assume** *i: i < n and len: length ts = n*

**from** *id-take-nth-drop[OF i[folded len]] i[folded len]*

**obtain** *us vs where id: Fun f ts = Fun f (us @ ts ! i # vs)*

**and** *us: us = take i ts*

**and** *len: length us = i by auto*

**have** *length us < Suc (length us + length vs) by auto*

**from** *scf[OF this, of f] obtain j where [simp]: scf (f, Suc (length us + length vs)) (length us) = Suc j*

**by** (*rule lessE*)

**show** (*Fun f ts, ts ! i*) ∈ *weight-NS*

**unfolding** *weight-le-def id* **by** (*auto simp: o-def*)

**qed**

**lemma** *weight-lemmas*:

**shows** *refl weight-NS and trans weight-NS and trans weight-S*

**and** *weight-NS O weight-S ⊆ weight-S and weight-S O weight-NS ⊆ weight-S*

by (auto intro!: refl-onI transI simp: weight-le-def weight-less-def)

**interpretation** kbo': *admissible-kbo w w0 pr-strict' pr-weak' least scf*  
by (rule *admissible-kbo'*)

**context**  
**assumes** *least-global*:  $\bigwedge f g. \text{least } f \implies \text{pr-weak } g (f,0)$   
**and** *least-trans*:  $\bigwedge f g. \text{least } f \implies \text{pr-weak } (f,0) g \implies \text{least } (\text{fst } g) \wedge \text{snd } g = 0$   
**fixes** *n* :: nat  
**begin**

**lemma** *kbo-instance-of-wpo-with-SN-assms: wpo-with-SN-assms*  
*weight-S weight-NS* ( $\lambda f g. (\text{pr-strict } f g, \text{pr-weak } f g)$ )  
( $\lambda(f, n). n = 0 \wedge \text{least } f$ ) *full-status* *False* ( $\lambda f. \text{False}$ )  
**apply** (*unfold-locales*)  
**apply** (*auto simp: weight-lemmas SN-weight-S pr-SN pr-strict-irrefl*  
*weight-less-stable weight-le-stable weight-le-mono-one weight-less-imp-le*  
*simple-arg-pos-weight*)  
**apply** (*force dest: least-global least-trans simp: pr-strict*)+  
**using** *SN-on-irrefl[OF SN-weight-S]*  
**apply** (*auto simp: pr-strict least irrefl-def dest:pr-weak-trans*)  
**done**

**interpretation** wpo: *wpo-with-SN-assms*  
**where** *S* = *weight-S* **and** *NS* = *weight-NS*  
**and** *prc* =  $\lambda f g. (\text{pr-strict } f g, \text{pr-weak } f g)$  **and** *prl* =  $\lambda(f,n). n = 0 \wedge \text{least } f$   
**and** *c* =  $\lambda-. \text{Lex}$   
**and** *ssimple* = *False* **and** *large* =  $\lambda f. \text{False}$  **and**  $\sigma\sigma$  = *full-status*  
**and** *n* = *n*  
by (rule *kbo-instance-of-wpo-with-SN-assms*)

**lemma** *kbo-as-wpo-with-assms: assumes bounded-arity n (funas-term t)*  
**shows** *kbo s t = wpo.wpo s t*  
**proof** –  
**define** *m* **where** *m* = *size s* + *size t*  
**from** *m-def* *assms* **show** *?thesis*  
**proof** (*induct m arbitrary: s t rule: less-induct*)  
**case** (*less m s t*)  
**hence** *IH*: *size si* + *size ti* < *size s* + *size t*  $\implies$  *bounded-arity n (funas-term*  
*ti)*  $\implies$  *kbo si ti = wpo.wpo si ti* **for** *si ti* :: (*f, 'a*)*term* **by** *auto*  
**note** *wpo-sI* = *arg-cong[OF wpo.wpo.simps, of fst, THEN iffD2]*  
**note** *wpo-nsI* = *arg-cong[OF wpo.wpo.simps, of snd, THEN iffD2]*  
**note** *bounded* = *less(3)*  
**show** *?case*  
**proof** (*cases s*)  
**case** *s*: (*Var x*)  
**have**  $\neg \text{weight-less } t (\text{Var } x)$   
**by** (*metis leD weight.simps(1) weight-le-less-iff weight-less-imp-le weight-w0*)  
**thus** *?thesis*

```

    by (cases t, auto simp add: s kbo-altdef wpo.wpo.simps)
next
case s: (Fun f ss)
show ?thesis
proof (cases t)
  case t: (Var y)
  { assume weight-le t s
    then have  $\exists s' \in \text{set } ss. \text{weight-le } t s'$ 
      apply (auto simp: s t weight-le-def)
      by (metis scf set-scf-list weight-w0)
    then obtain s' where s':  $s' \in \text{set } ss$  and weight-le t s' by auto
    from this(2) have wpo.wpo-ns s' t
    proof (induct s')
      case (Var x)
      then show ?case by (auto intro!: wpo-nsI simp: t weight-le-Var-Var)
    next
      case (Fun f' ss')
      from this(2) have  $\exists s'' \in \text{set } ss'. \text{weight-le } t s''$ 
        apply (auto simp: t weight-le-def)
        by (metis scf set-scf-list weight-w0)
      then obtain s'' where s''  $\in \text{set } ss'$  and weight-le t s'' by auto
      with Fun(1)[OF this] Fun(2)
      show ?case by (auto intro!: wpo-nsI simp: t in-set-conv-nth)
    qed
    with s' have  $\exists s' \in \text{set } ss. \text{wpo.wpo-ns } s' t$  by auto
  }
then
show ?thesis unfolding wpo.wpo.simps[of s t] kbo-altdef[of s t]
  by (auto simp add: s t weight-less-iff set-conv-nth, auto)
next
case t: (Fun g ts)
{
  fix j
  assume j < length ts
  hence ts ! j  $\in \text{set } ts$  by auto
  hence funas-term (ts ! j)  $\subseteq \text{funas-term } t$  unfolding t by auto
  with bounded have bounded-arity n (funas-term (ts ! j)) unfolding
bounded-arity-def by auto
} note bounded-tj = this
note IH-tj = IH[OF - this]
show ?thesis
proof (cases  $\neg \text{weight-le } t s \vee \text{weight-less } t s$ )
  case True
  thus ?thesis unfolding wpo.wpo.simps[of s t] kbo-altdef[of s t]
    unfolding s t by (auto simp: weight-less-iff)
  next
  case False
  let ?f = (f, length ss)
  let ?g = (g, length ts)

```

**from** *False* **have** *wle*: *weight-le t s = True weight-less t s = False*  
 $(s, t) \in \text{weight-NS} \iff \text{True}$   $(s, t) \in \text{weight-S} \iff \text{False}$  **by** *auto*  
**have** *lex*:  $(\text{Lex} = \text{Lex} \wedge \text{Lex} = \text{Lex}) = \text{True}$  **by** *simp*  
**have** *sig*:  $\text{set } (\text{wpo}.\sigma \ ?f) = \{..<\text{length } ss\}$   
 $\text{set } (\text{wpo}.\sigma \ ?g) = \{..<\text{length } ts\}$  **by** *auto*  
**have** *map*:  $\text{map } (!) \ ss \ (\text{wpo}.\sigma \ ?f) = ss$   
 $\text{map } (!) \ ts \ (\text{wpo}.\sigma \ ?g) = ts$   
**by** *(auto simp: map-nth)*  
**have** *sizes*:  $i < \text{length } ss \implies \text{size } (ss ! i) < \text{size } s$  **for** *i* **unfolding** *s*  
**by** *(simp add: size-simp1)*  
**have** *sizet*:  $i < \text{length } ts \implies \text{size } (ts ! i) < \text{size } t$  **for** *i* **unfolding** *t*  
**by** *(simp add: size-simp1)*  
**have** *wpo*:  $\text{wpo}.\text{wpo } s \ t =$   
*(if*  $\exists i \in \{..<\text{length } ss\}.$  *wpo.wpo-ns*  $(ss ! i) \ t$  *then*  $(\text{True}, \text{True})$   
*else if* *pr-weak*  $?f \ ?g \wedge (\forall j \in \{..<\text{length } ts\}.$  *wpo.wpo-s*  $s \ (ts ! j))$   
*then if* *pr-strict*  $?f \ ?g$  *then*  $(\text{True}, \text{True})$  *else* *lex-ext* *wpo.wpo n ss ts*  
*else*  $(\text{False}, \text{False}))$   
**unfolding** *wpo.wpo.simps[of s t]*  
**unfolding** *s t term.simps split Let-def lex if-True sig map*  
**unfolding** *s[symmetric] t[symmetric] wle if-True weight-less-iff if-False*  
*False snd-conv* **by** *auto*  
**have** *kbo*  $s \ t =$  *(if* *pr-strict*  $?f \ ?g$  *then*  $(\text{True}, \text{True})$   
*else if* *pr-weak*  $?f \ ?g$  *then* *lex-ext-unbounded* *kbo ss ts*  
*else*  $(\text{False}, \text{False}))$   
**unfolding** *kbo-altdef[of s t]*  
**unfolding** *s t term.simps split Let-def if-True*  
**unfolding** *s[symmetric] t[symmetric] wle if-True weight-less-iff if-False*  
**by** *auto*  
**also** **have** *lex-ext-unbounded kbo ss ts = lex-ext kbo n ss ts*  
**using** *bounded[unfolded t]* **unfolding** *bounded-arity-def lex-ext-def* **by**  
*auto*  
**also** **have**  $\dots = \text{lex-ext } \text{wpo}.\text{wpo } n \ ss \ ts$   
**by** *(rule lex-ext-cong[OF refl refl refl], rule IH-tj, auto dest!: sizes sizet)*  
**finally** **have** *kbo*:  $kbo \ s \ t =$   
*(if* *pr-strict*  $?f \ ?g$  *then*  $(\text{True}, \text{True})$   
*else if* *pr-weak*  $?f \ ?g$  *then* *lex-ext* *wpo.wpo n ss ts*  
*else*  $(\text{False}, \text{False}))$  .  
**show** *?thesis*  
**proof** *(cases*  $\exists i \in \{..<\text{length } ss\}.$  *wpo.wpo-ns*  $(ss ! i) \ t$   
**case** *True*  
**then** **obtain** *i* **where**  $i: i < \text{length } ss$  **and** *wpo.wpo-ns*  $(ss ! i) \ t$  **by** *auto*  
**then** **obtain** *b* **where**  $\text{wpo}.\text{wpo } (ss ! i) \ t = (b, \text{True})$  **by** *(cases wpo.wpo*  
 $(ss ! i) \ t,$  *auto)*  
**also** **have**  $\text{wpo}.\text{wpo } (ss ! i) \ t = kbo \ (ss ! i) \ t$  **using** *i* **by** *(intro IH[symmetric,*  
*OF - bounded], auto dest: sizes)*  
**finally** **have** *NS*  $(ss ! i) \ t$  **by** *simp*  
**from** *kbo-supt-one[OF this]*  
**have** *S*  $(\text{Fun } f \ (\text{take } i \ ss \ @ \ ss ! i \ \# \ \text{drop } (\text{Suc } i) \ ss)) \ t$  .  
**also** **have**  $(\text{take } i \ ss \ @ \ ss ! i \ \# \ \text{drop } (\text{Suc } i) \ ss) = ss$  **using** *i* **by** *(metis*

```

id-take-nth-drop)
  also have  $\text{Fun } f \text{ ss} = s$  unfolding  $s$  by  $\text{simp}$ 
  finally have  $S \ s \ t$  .
  with  $S\text{-imp-NS}$ [ $OF$   $\text{this}$ ]
  have  $kbo \ s \ t = (True, True)$  by ( $\text{cases } kbo \ s \ t, \text{auto}$ )
  with  $True$  show  $?thesis$  unfolding  $wpo$  by  $\text{auto}$ 
next
case  $False$ 
hence  $False: (\exists i \in \{..<\text{length } ss\}. wpo.wpo\text{-ns} (ss \ ! \ i) \ t) = False$  by  $\text{simp}$ 
{
  fix  $j$ 
  assume  $NS: NS \ s \ t$ 
  assume  $j: j < \text{length } ts$ 

  from  $kbo\text{-supt-one}$ [ $OF$   $NS\text{-refl}$ , of  $g$  take  $j$   $ts \ ! \ j$  drop ( $Suc \ j$ )  $ts$ ]
  have  $S: S \ t \ (ts \ ! \ j)$  using  $\text{id-take-nth-drop}$ [ $OF$   $j$ ] unfolding  $t$  by  $\text{auto}$ 
  from  $kbo\text{-trans}$ [of  $s \ t \ ts \ ! \ j$ ]  $NS \ S$  have  $S \ s \ (ts \ ! \ j)$  by  $\text{auto}$ 
  with  $S \ S\text{-imp-NS}$ [ $OF$   $\text{this}$ ]
  have  $kbo \ s \ (ts \ ! \ j) = (True, True)$  by ( $\text{cases } kbo \ s \ (ts \ ! \ j), \text{auto}$ )
  hence  $wpo.wpo\text{-s} \ s \ (ts \ ! \ j)$ 
    by ( $\text{subst } IH\text{-tj}$ [ $\text{symmetric}$ ],  $\text{insert } \text{size}$ [ $OF$   $j$ ]  $j, \text{auto}$ )
}
thus  $?thesis$  unfolding  $wpo \ kbo \ False \ \text{if-False}$  using  $\text{lex-ext-stri-imp-nstri}$ [of
 $wpo.wpo \ n \ ss \ ts$ ]
by ( $\text{cases } \text{lex-ext } wpo.wpo \ n \ ss \ ts, \text{auto } \text{simp: } \text{pr-strict split: } \text{if-splits}$ )
qed
qed
qed
qed
qed
qed
end

```

This is the main theorem. It tells us that KBO can be seen as an instance of WPO, under mild preconditions: the parameter  $n$  for the lexicographic extension has to be chosen high enough to cover the arities of all terms that should be compared.

```

lemma defines  $\text{prec} \equiv ((\lambda f \ g. (\text{pr-strict}' \ f \ g, \text{pr-weak}' \ f \ g)))$ 
and  $\text{prl} \equiv (\lambda(f, n). n = 0 \wedge \text{least } f)$ 
shows
   $kbo\text{-encoding-is-valid-wpo: } wpo\text{-with-SN-assms } \text{weight-S } \text{weight-NS } \text{prec } \text{prl } \text{full-status}$ 
   $False \ (\lambda f. False)$ 
and
   $kbo\text{-as-wpo: } \text{bounded-arity } n \ (\text{funas-term } t) \implies kbo \ s \ t = wpo.wpo \ n \ \text{weight-S}$ 
   $\text{weight-NS } \text{prec } \text{prl } \text{full-status} \ (\lambda-. \text{Lex}) \ False \ (\lambda f. False) \ s \ t$ 
unfolding  $\text{prec-def } \text{prl-def}$ 
subgoal by ( $\text{intro } \text{admissible-kbo.kbo-instance-of-wpo-with-SN-assms}$ [ $OF$   $\text{admissible-kbo}$ ]
   $\text{least-pr-weak}' \ \text{least-pr-weak}'\text{-trans}$ )

```

```

apply (subst kbo'-eq-kbo[symmetric])
apply (subst admissible-kbo.kbo-as-wpo-with-assms[OF admissible-kbo' least-pr-weak'
least-pr-weak'-trans, symmetric], (auto)[?])
by auto

```

As a proof-of-concept we show that now properties of WPO can be used to prove these properties for KBO. Here, as example we consider closure under substitutions and strong normalization, but the following idea can be applied for several more properties: if the property involves only terms where the arities are bounded, then just choose the parameter  $n$  large enough. This even works for strong normalization, since in an infinite chain of KBO-decreases  $t_1 > t_2 > t_3 > \dots$  all terms have a weight of at most the weight of  $t_1$ , and this weight is also a bound on the arities.

**lemma** *KBO-stable-via-WPO*:  $S \ s \ t \implies S \ (s \cdot (\sigma :: ('f, 'a) \text{subst})) \ (t \cdot \sigma)$

**proof** –

```

let ?terms = {t, t · σ}
let ?prec = ((λf g. (pr-strict' f g, pr-weak' f g)))
let ?prl = (λ(f, n). n = 0 ∧ least f)
have finite (∪ (funas-term ' ?terms))
by auto
from finite-list[OF this] obtain F where F: set F = ∪ (funas-term ' ?terms)
by auto

```

```

define n where n = max-list (map snd F)

```

```

interpret wpo: wpo-with-SN-assms
where S = weight-S and NS = weight-NS
and prc = ?prec and prl = ?prl
and c = λ-. Lex
and ssimple = False and large = λf. False and σσ = full-status
and n = n
by (rule kbo-encoding-is-valid-wpo)

```

```

{
fix t
assume t ∈ ?terms
hence funas-term t ⊆ set F unfolding F by auto
hence bounded-arity n (funas-term t) unfolding bounded-arity-def
using max-list[of - map snd F, folded n-def] by fastforce
}

```

```

note kbo-as-wpo = kbo-as-wpo[OF this]

```

```

from wpo.WPO-S-subst[of s t σ]
show S s t ⟹ S (s · σ) (t · σ)
using kbo-as-wpo by auto
qed

```

**lemma** *weight-is-arity-bound*:  $\text{weight } t \leq b \implies \text{bounded-arity } b \text{ (funas-term } t)$   
**proof** (*induct t*)  
  **case** (*Fun f ts*)  
  **have**  $\text{sum-list (map weight ts)} \leq \text{weight (Fun f ts)}$   
  **using**  $\text{sum-list-scf-list[of ts scf (f,length ts), OF scf]}$  **by** *auto*  
  **also have**  $\dots \leq b$  **using** *Fun* **by** *auto*  
  **finally have**  $\text{sum-b: sum-list (map weight ts)} \leq b$  .  
  {  
    **fix** *t*  
    **assume**  $t: t \in \text{set ts}$   
    **from**  $\text{split-list[OF this]}$  **have**  $\text{weight } t \leq \text{sum-list (map weight ts)}$  **by** *auto*  
    **with**  $\text{sum-b}$  **have**  $\text{bounded-arity } b \text{ (funas-term } t)$  **using** *t Fun* **by** *auto*  
  } **note**  $IH = \text{this}$   
  **have**  $\text{length ts} = \text{sum-list (map (\lambda -. 1) ts)}$  **by** (*induct ts, auto*)  
  **also have**  $\dots \leq \text{sum-list (map weight ts)}$   
  **apply** (*rule sum-list-mono*)  
  **subgoal for** *t* **using**  $\text{weight-gt-0[of t]}$  **by** *auto*  
  **done**  
  **also have**  $\dots \leq b$  **by** *fact*  
  **finally have**  $\text{len: length ts} \leq b$  **by** *auto*  
  **from**  $IH \text{ len}$  **show**  $?case$  **unfolding**  $\text{bounded-arity-def}$  **by** *auto*  
**qed** (*auto simp: bounded-arity-def*)

**lemma** *KBO-SN-via-WPO*:  $\text{SN } \{(s,t). S \ s \ t\}$

**proof**  
  **fix**  $f :: \text{nat} \Rightarrow (f, 'a)\text{term}$   
  **assume**  $\forall i. (f \ i, f \ (\text{Suc } i)) \in \{(s, t). S \ s \ t\}$   
  **hence**  $\text{steps: } S \ (f \ i) \ (f \ (\text{Suc } i))$  **for** *i* **by** *auto*  
  **define** *n* **where**  $n = \text{weight } (f \ 0)$

**have**  $w\text{-bound: weight } (f \ i) \leq n$  **for** *i*  
**proof** (*induct i*)  
  **case** (*Suc i*)  
  **from**  $\text{steps[of } i]$  **have**  $\text{weight } (f \ (\text{Suc } i)) \leq \text{weight } (f \ i)$   
  **unfolding**  $\text{kbo.simps[of } f \ i]$  **by** (*auto split: if-splits*)  
  **with** *Suc* **show**  $?case$  **by** *simp*  
**qed** (*auto simp: n-def*)

**let**  $?prec = ((\lambda f \ g. (\text{pr-strict}' \ f \ g, \text{pr-weak}' \ f \ g)))$   
**let**  $?prl = (\lambda(f, n). n = 0 \wedge \text{least } f)$

**interpret** *wpo*: *wpo-with-SN-assms*  
**where**  $S = \text{weight-S}$  **and**  $NS = \text{weight-NS}$   
  **and**  $\text{prc} = ?prec$  **and**  $\text{prl} = ?prl$   
  **and**  $c = \lambda -. \text{Lex}$   
  **and**  $\text{ssimple} = \text{False}$  **and**  $\text{large} = \lambda f. \text{False}$  **and**  $\sigma\sigma = \text{full-status}$   
  **and**  $n = n$

```

    by (rule kbo-encoding-is-valid-wpo)

  have kbo (f i) (f (Suc i)) = wpo.wpo (f i) (f (Suc i)) for i
    by (rule kbo-as-wpo[OF weight-is-arity-bound[OF w-bound]])

  from steps[unfolded this] wpo.WPO-S-SN show False by auto
qed

end

end

```

## 9 Executability of the orders

```

theory Executable-Orders
  imports
    WPO
    RPO
    LPO
    Multiset-Extension2-Impl
begin

```

If one loads the implementation of multiset orders (in particular for *mul-ext*), then all orders defined in this AFP-entry (WPO, RPO, LPO, multiset extension of order pairs) are executable.

```

export-code
  lpo
  rpo
  wpo.wpo
  mul-ext
  mult2-impl
in Haskell

```

```
end
```

## References

- [1] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.
- [2] N. Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1/2):69–116, 1987.
- [3] S. Kamin and J. J. Lévy. Two generalizations of the recursive path ordering. Unpublished Manuscript, University of Illinois, IL, USA, 1980.

- [4] D. E. Knuth and P. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. 1970.
- [5] C. Sternagel and R. Thiemann. Formalizing Knuth–Bendix orders and Knuth–Bendix completion. In *Rewriting Techniques and Applications, RTA '13*, volume 2 of *Leibniz International Proceedings in Informatics*, pages 287–302, 2013.
- [6] C. Sternagel and R. Thiemann. A formalization of Knuth–Bendix orders. *Archive of Formal Proofs*, May 2020. [https://isa-afp.org/entries/Knuth\\_Bendix\\_Order.html](https://isa-afp.org/entries/Knuth_Bendix_Order.html), Formal proof development.
- [7] R. Thiemann, G. Allais, and J. Nagele. On the formalization of termination techniques based on multiset orderings. In A. Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA '12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, volume 15 of *LIPICs*, pages 339–354. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.
- [8] R. Thiemann, J. Schöpf, C. Sternagel, and A. Yamada. Certifying the weighted path order (invited talk). In Z. M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPICs*, pages 4:1–4:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- [9] A. Yamada, K. Kusakari, and T. Sakabe. Unifying the Knuth-Bendix, recursive path and polynomial orders. In R. Peña and T. Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 181–192. ACM, 2013.
- [10] A. Yamada, K. Kusakari, and T. Sakabe. A unified ordering for termination proving. *Sci. Comput. Program.*, 111:110–134, 2015.