A Formalization of Weighted Path Orders and Recursive Path Orders*

Christian Sternagel René Thiemann Akihisa Yamada

November 24, 2021

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

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

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 chosing 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* = { (σ :: *'f* × *nat* ⇒ *nat list*). (∀ *f k*. *set* (σ (*f*, *k*)) ⊆ {*0* ..< *k*})}
  morphisms *status Abs-status*
  by (*rule exI*[*of* - λ -. []]) *auto*

setup-lifting *type-definition-status*

lemma *status*: *set* (*status* σ (*f*, *n*)) ⊆ {*0* ..< *n*}
  by (*transfer*) *auto*

lemma *status-aux*[*termination-simp*]: *i* ∈ *set* (*status* σ (*f*, *length ss*)) ⟹ *ss* ! *i* ∈ *set ss*
  using *status*[*of* σ *f length ss*] **unfolding** *set-conv-nth* **by** *force*

lemma *status-termination-simps*[*termination-simp*]:
  assumes *i1*: *i* < *length* (*status* σ (*f*, *length xs*))
  shows *size* (*xs* ! (*status* σ (*f*, *length xs*) ! *i*)) < *Suc* (*size-list size xs*) (**is** *?a* < *?c*)
proof −
  from *i1* **have** *status* σ (*f*, *length xs*) ! *i* ∈ *set* (*status* σ (*f*, *length xs*)) **by** *auto*
  from *status-aux*[*OF this*] **have** *?a* ≤ *size-list size xs* **by** (*auto simp*: *termination-simp*)

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

lemma *status-ne*:
status σ $(f, n) \neq [] \Longrightarrow \exists\, i < n.\ i \in set\ (status\ \sigma\ (f, n))$
using *status* [*of* σ *f n*]
by (*meson atLeastLessThan-iff set-empty subsetCE subsetI subset-empty*)

lemma *set-status-nth*:
length xs $= n \Longrightarrow i \in set\ (status\ \sigma\ (f, n)) \Longrightarrow i < length\ xs \wedge xs\ !\ i \in set\ xs$
using *status* [*of* σ *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* $\Rightarrow$ $'f \times nat \Rightarrow nat \Rightarrow bool$ **where**
simple-arg-pos rel f i $\equiv \forall$ *ts*. $i < snd\ f \longrightarrow length\ ts = snd\ f \longrightarrow (Fun\ (fst\ f)\ ts, ts\ !\ i) \in rel$

lemma *simple-arg-posI*: $[\![\bigwedge$ *ts*. *length ts* $= n \Longrightarrow i < n \Longrightarrow (Fun\ f\ ts, ts\ !\ i) \in rel]\!] \Longrightarrow$ *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 *precedence* =
fixes *prc* :: $'f \Rightarrow 'f \Rightarrow bool \times bool$
and *prl* :: $'f \Rightarrow bool$
assumes *prc-refl*: *prc f f* $= (False, True)$
and *prc-SN*: *SN* $\{(f, g).\ fst\ (prc\ f\ g)\}$

and *prl*: *prl g* $\Longrightarrow$ *snd* (*prc f g*) = *True*
and *prl3*: *prl f* $\Longrightarrow$ *snd* (*prc f g*) $\Longrightarrow$ *prl g*
and *prc-compat*: *prc f g* = (*s1*, *ns1*) $\Longrightarrow$ *prc g h* = (*s2*, *ns2*) $\Longrightarrow$ *prc f h* = (*s*, *ns*) $\Longrightarrow$
(*ns1* $\wedge$ *ns2* $\longrightarrow$ *ns*) $\wedge$ (*ns1* $\wedge$ *s2* $\longrightarrow$ *s*) $\wedge$ (*s1* $\wedge$ *ns2* $\longrightarrow$ *s*)
and *prc-stri-imp-nstri*: *fst* (*prc f g*) $\Longrightarrow$ *snd* (*prc f g*)
begin

lemma *prl2*:
assumes *g*: *prl g* **shows** *fst* (*prc g f*) = *False*
proof (*rule ccontr*)
assume ¬ *?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 *SN-on-irrefl*[*OF prc-SN*] **show** *False* **by** *blast*
qed

abbreviation *pr* ≡ (*prc*, *prl*)

end

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* ⇒ *'a rel* ⇒ *bool* **where**
compatible-l R1 R2 ≡ *R1 O R2* ⊆ *R2*

definition *compatible-r* :: *'a rel* ⇒ *'a rel* ⇒ *bool* **where**
compatible-r R1 R2 ≡ *R2 O R1* ⊆ *R2*

Local reflexivity

definition *locally-refl* :: *'a rel* ⇒ *'a multiset* ⇒ *bool* **where**
locally-refl R A ≡ (∀ *a*. *a* ∈# *A* ⟶ (*a*,*a*) ∈ *R*)

definition *locally-irrefl* :: *'a rel* ⇒ *'a multiset* ⇒ *bool* **where**
locally-irrefl R A ≡ (∀ *t*. *t* ∈# *A* ⟶ (*t*,*t*) ∉ *R*)

Local symmetry

definition *locally-sym* :: *'a rel* ⇒ *'a multiset* ⇒ *bool* **where**

locally-sym R A ≡ (∀ *t u*. *t* ∈# *A* ⟶ *u* ∈# *A* ⟶
(*t*,*u*) ∈ *R* ⟶ (*u*,*t*) ∈ *R*)

definition *locally-antisym* :: *'a rel* ⇒ *'a multiset* ⇒ *bool* **where**
locally-antisym R A ≡ (∀ *t u*. *t* ∈# *A* ⟶ *u* ∈# *A* ⟶
(*t*,*u*) ∈ *R* ⟶ (*u*,*t*) ∈ *R* ⟶ *t* = *u*)

Local transitivity

definition *locally-trans* :: *'a rel* ⇒ *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* ⇒ *bool* **where**
locally-trans R A B C ≡ (∀ *t u v*.
t ∈# *A* ⟶ *u* ∈# *B* ⟶ *v* ∈# *C* ⟶
(*t*,*u*) ∈ *R* ⟶ (*u*,*v*) ∈ *R* ⟶ (*t*,*v*) ∈ *R*)

Local inclusion

definition *locally-included* :: *'a rel* ⇒ *'a rel* ⇒ *'a multiset* ⇒ *'a multiset* ⇒ *bool* **where**
locally-included R1 R2 A B ≡ (∀ *t u*. *t* ∈# *A* ⟶ *u* ∈# *B* ⟶
(*t*,*u*) ∈ *R1* ⟶ (*t*,*u*) ∈ *R2*)

Local transitivity compatibility

definition *locally-compatible-l* :: *'a rel* ⇒ *'a rel* ⇒ *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* ⇒ *bool* **where**
locally-compatible-l R1 R2 A B C ≡ (∀ *t u v*. *t* ∈# *A* ⟶ *u* ∈# *B* ⟶ *v* ∈# *C* ⟶
(*t*,*u*) ∈ *R1* ⟶ (*u*,*v*) ∈ *R2* ⟶ (*t*,*v*) ∈ *R2*)

definition *locally-compatible-r* :: *'a rel* ⇒ *'a rel* ⇒ *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* ⇒ *bool* **where**
locally-compatible-r R1 R2 A B C ≡ (∀ *t u v*. *t* ∈# *A* ⟶ *u* ∈# *B* ⟶ *v* ∈# *C* ⟶
(*t*,*u*) ∈ *R2* ⟶ (*u*,*v*) ∈ *R1* ⟶ (*t*,*v*) ∈ *R2*)

included + compatible ⟶ transitive

lemma *in-cl-tr*:
assumes *R1* ⊆ *R2*
and *compatible-l R2 R1*
shows *trans R1*
proof−
{
fix *x y z*
assume *s-x-y*: (*x*,*y*) ∈ *R1* **and** *s-y-z*: (*y*,*z*) ∈ *R1*
from *assms s-x-y* **have** (*x*,*y*) ∈ *R2* **by** *auto*
with *s-y-z assms*(*2*)[*unfolded compatible-l-def*] **have** (*x*,*z*) ∈ *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** ↾ *80*)

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

lemma *mem-restrictD*[*dest*]: **assumes** $(x,y) \in R \restriction X$ **shows** $x \in X$ $y \in X$ $(x,y) \in R$
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 ⇒ 'a rel ⇒ '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 ⟹ SN-order-pair (s-list S NS) (ns-list S NS)*
and *s-map: ⟦⋀ a b. (a,b) ∈ S ⟹ (f a,f b) ∈ S; ⋀ a b. (a,b) ∈ NS ⟹ (f a,f b) ∈ NS⟧ ⟹ (as,bs) ∈ s-list S NS ⟹ (map f as, map f bs) ∈ s-list S NS*
and *ns-map: ⟦⋀ a b. (a,b) ∈ S ⟹ (f a,f b) ∈ S; ⋀ a b. (a,b) ∈ NS ⟹ (f a,f b) ∈ NS⟧ ⟹ (as,bs) ∈ ns-list S NS ⟹ (map f as, map f bs) ∈ ns-list S NS*
and *all-ns-imp-ns: length as = length bs ⟹ ⟦⋀ i. i < length bs ⟹ (as ! i, bs ! i) ∈ NS⟧ ⟹ (as,bs) ∈ ns-list S NS*

type-synonym *'a list-ext-impl = ('a ⇒ 'a ⇒ bool × bool) ⇒ 'a list ⇒ 'a list ⇒ bool × 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: ⋀ s ns. s-list {(a,b). s a b} {(a,b). ns a b} = {(as,bs). fst (list-ext (λ a b. (s a b, ns a b)) as bs)}*
and *list-ext-ns: ⋀ s ns. ns-list {(a,b). s a b} {(a,b). ns a b} = {(as,bs). snd (list-ext (λ a b. (s a b, ns a b)) as bs)}*
and *s-ext-local-mono: ⋀ s ns s' ns' as bs. (set as × set bs) ∩ ns ⊆ ns' ⟹ (set as × set bs) ∩ s ⊆ s' ⟹ (as,bs) ∈ s-list ns s ⟹ (as,bs) ∈ s-list ns' s'*
and *ns-ext-local-mono: ⋀ s ns s' ns' as bs. (set as × set bs) ∩ ns ⊆ ns' ⟹ (set as × set bs) ∩ s ⊆ s' ⟹ (as,bs) ∈ ns-list ns s ⟹ (as,bs) ∈ ns-list ns' s'*

end

3 Multiset extension of an order pair

Given a well-founded order $\prec$ and a compatible non-strict order $\precsim$, 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 $(X + Z,\ Y + Z) \in mult\ s \longleftrightarrow (X,\ Y) \in mult\ s$ (**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** $*$: *add-mset z X + Z = Z' + X' add-mset z Y + Z = Z' + Y' Y' ≠ {#}*
$\forall x \in set\text{-}mset\ X'.\ \exists y \in set\text{-}mset\ Y'.\ (x,\ y) \in s$
using *mult-implies-one-step*[*OF* ‹*trans s*› *add*(*2*)] **by** *auto*
consider *Z2* **where** *Z' = add-mset z Z2* | *X2 Y2* **where** *X' = add-mset z X2 Y' = 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.prems*(*2*) *add.prems*(*3*) *add-mset-add-single li-trans-l union-mset-add-mset-right*)
next
case *2* **then obtain** *y* **where** $y \in set\text{-}mset\ Y2$ $(z,\ y) \in s$ **using** $*$(*4*) *add*(*3*, *4*)
by (*auto simp*: *locally-irrefl-def*)
moreover from *this transD*[*OF* ‹*trans s*› - *this*(*2*)]
have $x' \in set\text{-}mset\ X2 \Longrightarrow \exists y \in set\text{-}mset\ Y2.\ (x',\ y) \in s$ **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 $Y = I + J\ X = I + K\ J \neq \{\#\}\ \forall k \in set\text{-}mset\ K.\ \exists j \in set\text{-}mset\ J.\ (k,\ j) \in s$
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 $(Z + X,\ Z + Y) \in mult\ s \longleftrightarrow (X,\ Y) \in mult\ s$
using *mult-locally-cancel*[*OF assms*] **by** (*simp only*: *union-commute*)

lemma *mult-cancelL*:

assumes *trans s irrefl s* **shows** $(Z + X, Z + Y) \in mult\ s \longleftrightarrow (X, Y) \in mult\ s$
using *assms mult-locally-cancelL* **by** (*simp add*: *mult-cancel union-commute*)

lemma *wf-trancl-conv*:
shows $wf\ (r^+) \longleftrightarrow wf\ r$
using *wf-subset*[*of* r^+ r] **by** (*force simp*: *wf-trancl*)

3.1 Pointwise multiset order

inductive-set *multpw* :: *'a rel* ⇒ *'a multiset rel* **for** *ns* :: *'a rel* **where**
empty: $(\{\#\}, \{\#\}) \in multpw\ ns$
| *add*: $(x, y) \in ns \Longrightarrow (X, Y) \in multpw\ ns \Longrightarrow (add\text{-}mset\ x\ X, add\text{-}mset\ y\ Y) \in multpw\ ns$

lemma *multpw-emptyL* [*simp*]:
$(\{\#\}, X) \in multpw\ ns \longleftrightarrow X = \{\#\}$
by (*cases X*) (*auto elim*: *multpw.cases intro*: *multpw.intros*)

lemma *multpw-emptyR* [*simp*]:
$(X, \{\#\}) \in multpw\ ns \longleftrightarrow X = \{\#\}$
by (*cases X*) (*auto elim*: *multpw.cases intro*: *multpw.intros*)

lemma *refl-multpw*:
assumes *refl ns* **shows** *refl* (*multpw ns*)
proof −
have $(X, X) \in multpw\ ns$ **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 $(X, Y) \in multpw\ (Id :: 'a\ rel) \Longrightarrow X = Y$ **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** $multpw\ ns \subseteq multpw\ ns'$
proof −
have $(X, Y) \in multpw\ ns \Longrightarrow (X, Y) \in multpw\ ns'$ **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*:
$multpw\ (ns^{-1}) = (multpw\ ns)^{-1}$
proof −

have $(X,\ Y) \in multpw\ (ns^{-1}) \Longrightarrow (X,\ Y) \in (multpw\ ns)^{-1}$ **for** $X\ Y$ **and** $ns :: 'a\ rel$
by (*induct X Y rule*: *multpw.induct*) (*auto intro*: *multpw.intros*)
then show *?thesis* **by** *auto*
qed

lemma *multpw-local*:
$(X,\ Y) \in multpw\ ns \Longrightarrow (X,\ Y) \in multpw\ (ns \cap set\text{-}mset\ X \times set\text{-}mset\ Y)$
proof (*induct X Y rule*: *multpw.induct*)
case (*add x y X Y*) **then show** *?case*
using *mono-multpw*[*of ns* ∩ *set-mset X* × *set-mset Y ns* ∩ *insert x* (*set-mset X*) × *insert y* (*set-mset Y*)]
by (*auto intro*: *multpw.intros*)
qed *auto*

lemma *multpw-split1R*:
assumes (*add-mset x X*, *Y*) ∈ *multpw ns*
obtains *z Z* **where** $Y = add\text{-}mset\ z\ Z$ **and** $(x,\ z) \in ns$ **and** $(X,\ Z) \in multpw\ ns$
using *assms*
proof (*induct 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*]: $X = add\text{-}mset\ x'\ X''$
using *add*(*4*) *False*
by (*metis add-eq-conv-diff*)
have $X' = add\text{-}mset\ x\ X''$ **using** *add*(*4*) **by** (*auto simp*: *add-eq-conv-ex*)
with *add*(*2*) **obtain** *Y'' y* **where** $Y' = add\text{-}mset\ y\ Y''$ $(x,y) \in ns$ $(X'',\ Y'') \in multpw\ ns$
by (*auto intro*: *add*(*3*))
then show *?thesis* **using** *add*(*1*) *add*(*5*)[*of y add-mset y' Y''*]
by (*auto simp*: *ac-simps intro*: *multpw.intros*)
qed *auto*
qed *auto*

lemma *multpw-splitR*:
assumes (*X1* + *X2*, *Y*) ∈ *multpw ns*
obtains *Y1 Y2* **where** *Y* = *Y1* + *Y2* **and** (*X1*, *Y1*) ∈ *multpw ns* **and** (*X2*, *Y2*) ∈ *multpw ns*
using *assms*
proof (*induct X2 arbitrary*: *Y thesis*)
case (*add x2 X2*)
from *add*(*3*) **obtain** *Y' y2* **where** (*X1* + *X2*, *Y'*) ∈ *multpw ns* (*x2*, *y2*) ∈ *ns Y* = *add-mset y2 Y'*
by (*auto elim*: *multpw-split1R simp*: *union-assoc*[*symmetric*])
moreover then obtain *Y1 Y2* **where** (*X1*, *Y1*) ∈ *multpw ns* (*X2*, *Y2*) ∈ *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 $(X,\ add\text{-}mset\ y\ Y) \in multpw\ ns$
obtains *z Z* **where** $X = add\text{-}mset\ z\ Z$ **and** $(z,\ y) \in ns$ **and** $(Z,\ Y) \in multpw\ ns$
using *assms multpw-split1R*[*of y Y X* ns^{-1} *thesis*] **by** (*auto simp*: *multpw-converse*)

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

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

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

lemma *multpw-mult1-commute*:
assumes *compat*: *s O ns* $\subseteq$ *s* **and** *reflns*: *refl ns*
shows *mult1 s O multpw ns* $\subseteq$ *multpw ns O mult1 s*
proof −
{ fix *X Y Z* **assume** *1*: $(X,\ Y) \in mult1\ s\ (Y,\ Z) \in multpw\ ns$
then obtain *X′ Y′ y* **where** *2*: $X = Y' + X'\ Y = add\text{-}mset\ y\ Y' \forall x.\ x \in\#$
$X' \longrightarrow (x,\ y) \in s$
by (*auto simp*: *mult1-def*)
moreover obtain *Z′ z* **where** *3*: $Z = add\text{-}mset\ z\ Z'\ (y,\ z) \in ns\ (Y',\ Z') \in$

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 $\exists Y'.\ (X,\ Y') \in$ *multpw ns* $\wedge\ (Y',\ Z) \in$ *mult1 s* **unfolding** *mult1-def*
using *refl-multpw*[*OF 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 refl ns* **shows** *mult s O multpw ns* $\subseteq$ *multpw ns O mult s*
proof −
{ fix *X Y Z* **assume** *1*: $(X,\ Y) \in$ *mult s* $(Y,\ Z) \in$ *multpw ns*
then have $\exists Y'.\ (X,\ Y') \in$ *multpw ns* $\wedge\ (Y',\ Z) \in$ *mult s* **unfolding** *mult-def*
using *multpw-mult1-commute*[*OF 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 refl ns* **shows** *wf* ((*multpw ns*)* *O mult s O* (*multpw ns*)*)
using *assms*(*1*) *multpw-mult-commute*[*OF assms*(*2*,*3*)] **by** (*subst qc-wf-relto-iff*) (*auto simp*: *wf-mult*)

lemma *multpw-cancel1*:
assumes *trans ns* $(y,\ x) \in$ *ns*
shows (*add-mset x X*, *add-mset y Y*) $\in$ *multpw ns* $\Longrightarrow (X,\ Y) \in$ *multpw ns* (**is** *?L* $\Longrightarrow$ *?R*)
proof −
assume *?L* **then obtain** *x' X'* **where** *X*: $(x',\ y) \in$ *ns add-mset x' X' = add-mset x X* $(X',\ Y) \in$ *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' = add-mset x X2 X = 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 = add-mset y' Y'* $(X2,\ Y') \in$ *multpw ns*
using *X*(*3*) **by** (*auto elim*: *multpw-split1R*)
have $(x',\ y') \in$ *ns* **using** *X*(*1*) *Y*(*1*) ‹*trans ns*› *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-multpw*[*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-multpw*[*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-multpw*[*OF assms*(*3*)] *trans-trancl*[*of mult1 s*, *folded mult-def*] *multpw-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-multpw*[*OF assms*(*3*)] *trans-trancl*[*of mult1 s*, *folded mult-def*] *multpw-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-multpw*[*OF assms*(*3*)] *trans-trancl*[*of mult1 s*, *folded mult-def*] *multpw-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-multpw*[*of ns*] *assms* **unfolding** *mult2-s-def refl-on-def* **by** *blast*

lemma *mult-implies-mult2-ns*:
assumes *refl ns* (*X*, *Y*) ∈ (*mult s*)$^=$
shows (*X*, *Y*) ∈ *mult2-ns ns s*
using *refl-multpw*[*of ns*] *assms* **unfolding** *mult2-ns-def refl-on-def* **by** *blast*

lemma *multpw-implies-mult2-ns*:
assumes (*X*, *Y*) ∈ *multpw 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*) ∈ *mult2-s ns s* ⟷ (∃ *X1 X2 Y1 Y2*. *X* = *X1* + *X2* ∧ *Y* = *Y1* + *Y2* ∧
(*X1*, *Y1*) ∈ *multpw ns* ∧ *Y2* ≠ {#} ∧ (∀ *x*. *x* ∈# *X2* ⟶ (∃ *y*. *y* ∈# *Y2* ∧ (*x*, *y*) ∈ *s*))) (**is** *?L* ⟷ *?R*)
proof
assume *?R* **then obtain** *X1 X2 Y1 Y2* **where** ∗: *X* = *X1* + *X2* *Y* = *Y1* + *Y2* (*X1*, *Y1*) ∈ *multpw ns* **and**

```
    Y2 ≠ {#} ∀x. x ∈# X2 ⟶ (∃y. y ∈# Y2 ∧ (x, y) ∈ s) by blast
  then have (Y1 + X2, Y1 + Y2) ∈ mult s
    using ‹trans s› by (auto intro: one-step-implies-mult)
  moreover have (X1 + X2, Y1 + X2) ∈ multpw ns
    using ‹refl ns› 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) ∈ multpw ns
    (X2, Z2) ∈ multpw ns Y2 ≠ {#} ∀x. x ∈# Z2 ⟶ (∃y. y ∈# Y2 ∧ (x, y) ∈
s)
    by (auto 0 3 dest!: mult-implies-one-step[OF ‹trans s›] simp: mult2-s-def elim!:
multpw-splitL) metis
  have ∀x. x ∈# X2 ⟶ (∃y. y ∈# Y2 ∧ (x,y) ∈ s)
  proof (intro allI impI)
    fix x assume x ∈# X2
    then obtain X2' where X2 = add-mset x X2' by (metis multi-member-split)
     then obtain z Z2' where Z2 = add-mset z Z2' (x, z) ∈ ns using ∗(4) by
(auto elim: multpw-split1R)
    then have z ∈# Z2 (x, z) ∈ ns by auto
    then show ∃y. y ∈# Y2 ∧ (x,y) ∈ s using ∗(6) ‹ns O s ⊆ s› by blast
  qed
  then show ?R using ∗ by auto
qed

lemma mult2-ns-one-step:
  assumes ns O s ⊆ s refl ns trans s
  shows (X, Y) ∈ mult2-ns ns s ⟷ (∃X1 X2 Y1 Y2. X = X1 + X2 ∧ Y = Y1
+ Y2 ∧
    (X1, Y1) ∈ multpw ns ∧ (∀x. x ∈# X2 ⟶ (∃y. y ∈# Y2 ∧ (x, y) ∈ s))) (is
?L ⟷ ?R)
proof
  assume ?L then consider (X, Y) ∈ multpw ns | (X, Y) ∈ 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) ∈ multpw ns ∀x. x ∈# X2 ⟶ (∃y. y ∈# Y2 ∧ (x, y) ∈ 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 ⊆ s refl ns locally-irrefl s X locally-irrefl s Y trans s
  shows (X, Y) ∈ mult2-s ns s ⟷ (∃X1 X2 Y1 Y2. X = X1 + X2 ∧ 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-irrefl s* (*X2* $+$ *Y1*) **and** *r*: *locally-irrefl s* (*Y2* $+$ *Y1*)
using *assms*(*3*, *4*) **by** (*auto simp add*: *locally-irrefl-def*)
show *?L* **using** *ns s mult-locally-cancelL*[*OF assms*(*5*) *l r*] *multpw-add*[*OF ns*, *of X2 X2*] *refl-multpw*[*OF* ‹*refl ns*›]
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-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-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* $\Longrightarrow$ *?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* $\Longrightarrow$ *?L* **for** *X1 Y1 Y2*
using *multpw-add*[*of X1 Y1 ns Y2 Y2*] *refl-multpw*[*OF* ‹*refl ns*›] **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*
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-irrefl s* (*add-mset z X*) *locally-irrefl s* (*add-mset z Y*) *trans s*
(*add-mset z X*, *add-mset z Y*) $\in$ *mult2-s ns s*
shows (*X*, *Y*) $\in$ *mult2-s ns s*
proof $-$

obtain *X1 X2 Y1 Y2* **where** ∗: *add-mset z X = X1 + X2 add-mset z Y = Y1 + Y2* (*X1*, *Y1*) ∈ *multpw ns*
(*X2*, *Y2*) ∈ *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-mset z X1′* | *Y1′* **where** *Y1 = add-mset z Y1′* | *X2′ Y2′* **where** *X2 = add-mset z X2′ Y2 = add-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′*) ∈ *multpw ns Y1 = add-mset z′ Y1′* (*z*, *z′*) ∈ *ns*
using ∗ **by** (*auto elim*: *multpw-split1R*)
then have (*X*, *Y1′ + Y2*) ∈ *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*) ∈ *multpw ns*
using *refl-multpw*[*OF* ‹*refl ns*›] ∗ ∗∗ *multpw-cancel1*[*OF* ‹*trans ns*› ∗∗(*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′*) ∈ *multpw ns X1 = add-mset z′ X1′* (*z′*, *z*) ∈ *ns*
using ∗ **by** (*auto elim*: *multpw-split1L*)
then have (*X1′ + X2*, *Y*) ∈ *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*) ∈ *multpw ns*
using *refl-multpw*[*OF* ‹*refl ns*›] ∗ ∗∗ *multpw-cancel1*[*OF* ‹*trans ns*› ∗∗(*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* {#-#}] *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* ⊆ *s ns O s* ⊆ *s refl ns trans ns irrefl s trans s* (*add-mset z X*, *add-mset z Y*) ∈ *mult2-s ns s*

shows $(X,\ Y) \in \mathit{mult2\text{-}s}\ \mathit{ns}\ s$
using *assms mult2-s-locally-cancel1* **by** (*metis irefl-def locally-irefl-def*)

lemma *mult2-s-locally-cancel*:
assumes $s\ O\ \mathit{ns} \subseteq s\ \mathit{ns}\ O\ s \subseteq s$ *refl ns trans ns locally-irefl s* $(X + Z)$ *locally-irefl s* $(Y + Z)$ *trans s*
shows $(X + Z,\ Y + Z) \in \mathit{mult2\text{-}s}\ \mathit{ns}\ s \Longrightarrow (X,\ Y) \in \mathit{mult2\text{-}s}\ \mathit{ns}\ s$
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 $s\ O\ \mathit{ns} \subseteq s\ \mathit{ns}\ O\ s \subseteq s$ *refl ns trans ns irrefl s trans s*
shows $(X + Z,\ Y + Z) \in \mathit{mult2\text{-}s}\ \mathit{ns}\ s \Longrightarrow (X,\ Y) \in \mathit{mult2\text{-}s}\ \mathit{ns}\ s$
using *mult2-s-locally-cancel assms* **by** (*metis irefl-def locally-irefl-def*)

lemma *mult2-ns-cancel*:
assumes $s\ O\ \mathit{ns} \subseteq s\ \mathit{ns}\ O\ s \subseteq s$ *refl ns trans s irrefl s trans ns*
shows $(X + Z,\ Y + Z) \in \mathit{mult2\text{-}s}\ \mathit{ns}\ s \Longrightarrow (X,\ Y) \in \mathit{mult2\text{-}ns}\ \mathit{ns}\ s$
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* ⇒ *'a rel* ⇒ *'a rel* ⇒ *'a multiset rel* **where**
$\mathit{mult2\text{-}alt}\ b\ \mathit{ns}\ s = \{(X,\ Y).\ (\exists\, \mathit{X1}\ \mathit{X2}\ \mathit{Y1}\ \mathit{Y2}.\ X = \mathit{X1} + \mathit{X2} \wedge Y = \mathit{Y1} + \mathit{Y2} \wedge (\mathit{X1},\ \mathit{Y1}) \in \mathit{multpw}\ \mathit{ns} \wedge (b \vee \mathit{Y2} \neq \{\#\}) \wedge (\forall x.\ x \in\# \mathit{X2} \longrightarrow (\exists y.\ y \in\# \mathit{Y2} \wedge (x,\ y) \in s)))\}$

lemma *mult2-altI*:
assumes $X = \mathit{X1} + \mathit{X2}\ Y = \mathit{Y1} + \mathit{Y2}\ (\mathit{X1},\ \mathit{Y1}) \in \mathit{multpw}\ \mathit{ns}$
$b \vee \mathit{Y2} \neq \{\#\}\ \forall x.\ x \in\# \mathit{X2} \longrightarrow (\exists y.\ y \in\# \mathit{Y2} \wedge (x,\ y) \in s)$
shows $(X,\ Y) \in \mathit{mult2\text{-}alt}\ b\ \mathit{ns}\ s$
using *assms* **unfolding** *mult2-alt-def* **by** *blast*

lemma *mult2-altE*:
assumes $(X,\ Y) \in \mathit{mult2\text{-}alt}\ b\ \mathit{ns}\ s$
obtains *X1 X2 Y1 Y2* **where** $X = \mathit{X1} + \mathit{X2}\ Y = \mathit{Y1} + \mathit{Y2}\ (\mathit{X1},\ \mathit{Y1}) \in \mathit{multpw}\ \mathit{ns}$
$b \vee \mathit{Y2} \neq \{\#\}\ \forall x.\ x \in\# \mathit{X2} \longrightarrow (\exists y.\ y \in\# \mathit{Y2} \wedge (x,\ y) \in s)$
using *assms* **unfolding** *mult2-alt-def* **by** *blast*

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

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

abbreviation *mult2-alt-ns* ≡ *mult2-alt True*

lemmas *mult2-alt-s-def* = *mult2-alt-def*[**where** *b* = *False*, *unfolded simp-thms*]
lemmas *mult2-alt-ns-def* = *mult2-alt-def*[**where** *b* = *True*, *unfolded simp-thms*]
lemmas *mult2-alt-sI* = *mult2-altI*[**where** *b* = *False*, *unfolded simp-thms*]
lemmas *mult2-alt-nsI* = *mult2-altI*[**where** *b* = *True*, *unfolded simp-thms True-implies-equals*]
lemmas *mult2-alt-sE* = *mult2-altE*[**where** *b* = *False*, *unfolded simp-thms*]
lemmas *mult2-alt-nsE* = *mult2-altE*[**where** *b* = *True*, *unfolded simp-thms True-implies-equals*]

Equivalence to *mult2-s* **and** *mult2-ns* **lemma** *mult2-s-eq-mult2-s-alt*:
assumes *ns O s* ⊆ *s refl ns trans s*
shows *mult2-alt-s ns s* = *mult2-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* ⊆ *s refl ns trans s*
shows *mult2-alt-ns ns s* = *mult2-ns ns s*
using *mult2-ns-one-step*[*OF assms*] **unfolding** *mult2-alt-ns-def* **by** *blast*

lemma *mult2-alt-local*:
assumes (*X*, *Y*) ∈ *mult2-alt b ns s*
shows (*X*, *Y*) ∈ *mult2-alt b* (*ns* ∩ *set-mset X* × *set-mset Y*) (*s* ∩ *set-mset X* × *set-mset Y*)
proof −
from *assms* **obtain** *X1 X2 Y1 Y2* **where** ∗: *X* = *X1* + *X2 Y* = *Y1* + *Y2* **and**
(*X1*, *Y1*) ∈ *multpw ns* (*b* ∨ *Y2* ≠ {#}) (∀ *x*. *x* ∈# *X2* ⟶ (∃ *y*. *y* ∈# *Y2* ∧ (*x*, *y*) ∈ *s*))
unfolding *mult2-alt-def* **by** *blast*
then have *X* = *X1* + *X2* ∧ *Y* = *Y1* + *Y2* ∧
(*X1*, *Y1*) ∈ *multpw* (*ns* ∩ *set-mset X* × *set-mset Y*) ∧ (*b* ∨ *Y2* ≠ {#}) ∧
(∀ *x*. *x* ∈# *X2* ⟶ (∃ *y*. *y* ∈# *Y2* ∧ (*x*, *y*) ∈ *s* ∩ *set-mset X* × *set-mset Y*))
using *multpw-local*[*of X1 Y1 ns*]
mono-multpw[*of ns* ∩ *set-mset X1* × *set-mset Y1 ns* ∩ *set-mset X* × *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* ⇒ *'a set* ⇒ *bool* **where**
wf-below r A = *wf* (*Restr r* ((*r*∗)$^{-1}$ `` *A*))

lemma *wf-below-UNIV*[*simp*]:
shows *wf-below r UNIV* ⟷ *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-below* r $\{a\})$ (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L* **then show** *?R* **using** *wf-below-mono2*[*of* {-} A r] **by** *blast*
next
have $*$: $(r^*)^{-1}$ `` $A = \bigcup\{(r^*)^{-1}$ `` $\{a\} \mid a.\ a \in A\}$ **unfolding** *Image-def* **by** *blast*
assume *?R*
{ fix x X **assume** $*$: $X \subseteq$ *Restr* r $((r^*)^{-1}$ `` $A)$ `` X $x \in X$
then obtain a **where** $**$: $a \in A$ $(x, a) \in r^*$ **unfolding** *Image-def* **by** *blast*
from $*$ **have** $X \cap ((r^*)^{-1}$ `` $\{a\}) \subseteq$ (*Restr* $r$ $((r^*)^{-1}$ `` $A)$ `` $X) \cap ((r^*)^{-1}$ `` $\{a\})$ **by** *auto*
also have ... $\subseteq$ *Restr* r $((r^*)^{-1}$ `` $\{a\})$ `` $(X \cap ((r^*)^{-1}$ `` $\{a\}))$ **unfolding** *Image-def* **by** *fastforce*
finally have $X \cap ((r^*)^{-1}$ `` $\{a\}) = \{\}$ **using** ‹*?R*› $**$(*1*) **unfolding** *wf-below-def*
by (*intro wfE-pf*[*of Restr* r $((r^*)^{-1}$ `` $\{a\})$]) (*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-on* r $(r^*$ `` $A)$ (**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-below* (r^{-1}) A
proof −
{ fix f
assume $f\ 0 \in r^*$ `` A **and** $**$: $(f\ i, f\ (Suc\ i)) \in r$ **for** i
then have $f\ i \in r^*$ `` 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 $(r^*$ `` $A)$ **for** i **using** $**$ **by** *auto*
}

moreover then have *SN-on r* $(r^*$ `` `` $A) \longleftrightarrow$ *SN-on* $(Restr\ r\ (r^*$ `` `` $A))\ (r^*$ `` `` $A)$
unfolding *SN-on-def* **by** *auto blast*
moreover have $(\bigwedge i.\ (f\ i, f\ (Suc\ i)) \in Restr\ r\ (r^*$ `` `` $A)) \Longrightarrow f\ 0 \in r^*$ `` `` A **for** *f*
by *auto*
then have *SN-on* $(Restr\ r\ (r^*$ `` `` $A))\ (r^*$ `` `` $A) \longleftrightarrow$ *SN-on* $(Restr\ r\ (r^*$ `` `` $A))$
UNIV
unfolding *SN-on-def* **by** *auto*
ultimately show *?thesis* **unfolding** *SN-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 $Restr\ (r^+)\ ((r^*)^{-1}$ `` `` $A) = (Restr\ r\ ((r^*)^{-1}$ `` `` $A))^+$
proof (*intro equalityI subrelI*, *elim IntE conjE*[*OF iffD1*[*OF mem-Sigma-iff*]])
fix *a b* **assume** $*$: $(a, b) \in r^+$ $b \in (r^*)^{-1}$ `` `` A
then have $(a, b) \in (Restr\ r\ ((r^*)^{-1}$ `` `` $A))^+ \wedge a \in (r^*)^{-1}$ `` `` 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 $(a, b) \in (Restr\ r\ ((r^*)^{-1}$ `` `` $A))^+$ **by** *simp*
next
fix *a b* **assume** $(a, b) \in (Restr\ r\ ((r^*)^{-1}$ `` `` $A))^+$
then show $(a, b) : Restr\ (r^+)\ ((r^*)^{-1}$ `` `` $A)$ **by** *induct auto*
qed

lemma *wf-below-trancl*:
shows *wf-below* $(r^+)\ A \longleftrightarrow$ *wf-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-below r* (*set-mset X*) **shows** *wf-below* (*mult r*) $\{X\}$

proof −
have *foo*: *mult r* $\subseteq$ *mult* (r^+) **using** *mono-mult*[*of r* r^+] **by** *force*
have $(Y, X) \in (mult\ (r^+))^* \Longrightarrow$ *set-mset Y* $\subseteq ((r^+)^*)^{-1}$ `` *set-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-mset K*. $\exists j \in$ *set-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** $*(3)$ **by** *auto*
moreover then obtain *x* **where** $x \in\# X$ $(j, x) \in r^*$ **using** *step*(*3*) **by** (*auto*
simp: $*$)
ultimately have $\exists x.\ x \in\# X \wedge (k, x) \in r^*$ **by** *auto*
}

```
    then show ?case using ∗ step(3) by (auto simp: Image-def) metis
  qed auto
  then have q: (Y, X) ∈ (mult (r⁺))* ⟹ y ∈# Y ⟹ y ∈ ((r⁺)*)⁻¹ `` set-mset
X for Y y by force
  have Restr (mult (r⁺)) (((mult (r⁺))*)⁻¹ `` {X}) ⊆ mult (Restr (r⁺) (((r⁺)*)⁻¹
`` set-mset X))
  proof (intro subrelI)
    fix N M assume (N, M) ∈ Restr (mult (r⁺)) (((mult (r⁺))*)⁻¹ `` {X})
    then have ∗∗: (N, X) ∈ (mult (r⁺))* (M, X) ∈ (mult (r⁺))* (N, M) ∈ mult
(r⁺) by auto
    obtain I J K where ∗: M = I + J N = I + K J ≠ {#} ∀ k ∈ set-mset K.
∃ j ∈ set-mset J. (k, j) ∈ r⁺
      using mult-implies-one-step[OF - ‹(N, M) ∈ mult (r⁺)›] by auto
    then show (N, M) ∈ mult (Restr (r⁺) (((r⁺)*)⁻¹ `` set-mset X))
    using q[OF ∗∗(1)] q[OF ∗∗(2)] unfolding ∗ by (auto intro: one-step-implies-mult)
  qed
  note bar = subset-trans[OF Int-mono[OF foo Sigma-mono] this]
  have ((mult r)*)⁻¹ `` {X} ⊆ ((mult (r⁺))*)⁻¹ `` {X} using foo by (simp add:
Image-mono rtrancl-mono)
  then have Restr (mult r) (((mult r)*)⁻¹ `` {X}) ⊆ mult (Restr (r⁺) (((r⁺)*)⁻¹
`` 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 ⊆ (s ∪ ns)* O s 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 `` A
  fix X assume X: X ⊆ Restr (ns* O s O ns*) ?A' `` X
  let ?X' = ((s ∪ ns)* ∩ UNIV × ((s*)⁻¹ `` A)) `` X
  have ∗: s O (s ∪ ns)* ⊆ (s ∪ ns)* O s
  proof −
    { fix x y z assume (y, z) ∈ (s ∪ ns)* and (x, y) ∈ s
      then have (x, z) ∈ (s ∪ 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) ∈ ((s ∪ ns)* O (s ∪ ns)*) O s using assms by blast
        then show ?case by simp
      qed }
    then show ?thesis by auto
  qed
  { fix x assume x ∈ Restr (ns* O s O ns*) ?A' `` X
    then obtain y z where ∗∗: y ∈ X z ∈ A (y, x) ∈ ns* O s O ns* (x, z) ∈ (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*:
$(X,\ \{\#\}) \in$ *mult2-alt b ns s* $\longleftrightarrow$ *b* $\wedge$ $X = \{\#\}$
unfolding *mult2-alt-def* **by** (*auto intro*: *multiset-eqI*)

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

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

lemma *mult2-alt-ns-conv*:
mult2-alt-ns ns s = *mult2-alt-s ns s* $\cup$ *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* $(X1,\ Y1) \in$ *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 $(X,\ Y) \in$ *mult2-alt-s ns s*
shows $(X,\ Y) \in$ *mult2-alt-ns ns s*
using *assms* **by** (*auto intro*: *mult2-alt-nsI elim*: *mult2-alt-sE*)

lemma *mult2-alt-add*:
assumes $(X1,\ Y1) \in$ *mult2-alt b1 ns s* **and** $(X2,\ Y2) \in$ *mult2-alt b2 ns s*
shows $(X1 + X2,\ Y1 + Y2) \in$ *mult2-alt* $(b1 \wedge b2)$ *ns s*
proof −
from *assms* **obtain** *X11 X12 Y11 Y12 X21 X22 Y21 Y22* **where**
X1 = *X11* + *X12 Y1* = *Y11* + *Y12*
$(X11,\ Y11) \in$ *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*
$(X21,\ Y21) \in$ *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 $\bigwedge x\ y.\ x \in\# X \Longrightarrow y \in\# Y \Longrightarrow (x, y) \in ns \Longrightarrow (f\ x, g\ y) \in ns'$
and $(X, Y) \in multpw\ ns$
shows $(image\text{-}mset\ f\ X, image\text{-}mset\ g\ Y) \in multpw\ ns'$
using *assms*(2,1) **by** (*induct X Y rule*: *multpw.induct*) (*auto intro*: *multpw.intros*)

lemma *mult2-alt-map*:
assumes $\bigwedge x\ y.\ x \in\# X \Longrightarrow y \in\# Y \Longrightarrow (x, y) \in ns \Longrightarrow (f\ x, g\ y) \in ns'$
and $\bigwedge x\ y.\ x \in\# X \Longrightarrow y \in\# Y \Longrightarrow (x, y) \in s \Longrightarrow (f\ x, g\ y) \in s'$
and $(X, Y) \in mult2\text{-}alt\ b\ ns\ s$
shows $(image\text{-}mset\ f\ X, image\text{-}mset\ g\ Y) \in mult2\text{-}alt\ b\ ns'\ s'$
proof −
from *assms*(3) **obtain** *X1 X2 Y1 Y2* **where** $X = X1 + X2\ Y = Y1 + Y2\ (X1, Y1) \in multpw\ ns$
$b \vee Y2 \neq \{\#\}\ \forall x.\ x \in\# X2 \longrightarrow (\exists y.\ y \in\# Y2 \wedge (x, y) \in s)$ **by** (*auto elim*: *mult2-altE*)
moreover from *this*(1,2,5) **have** $\forall x.\ x \in\# image\text{-}mset\ f\ X2 \longrightarrow (\exists y.\ y \in\# image\text{-}mset\ g\ Y2 \wedge (x, y) \in s')$
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*: $\bigwedge x\ y\ z.\ x \in\# X \Longrightarrow y \in\# Y \Longrightarrow z \in\# Z \Longrightarrow (x, y) \in s \Longrightarrow (y, z) \in s \Longrightarrow (x, z) \in s$
and *ns*: $\bigwedge x\ y\ z.\ x \in\# X \Longrightarrow y \in\# Y \Longrightarrow z \in\# Z \Longrightarrow (x, y) \in ns \Longrightarrow (y, z) \in s \Longrightarrow (x, z) \in s$
and *sn*: $\bigwedge x\ y\ z.\ x \in\# X \Longrightarrow y \in\# Y \Longrightarrow z \in\# Z \Longrightarrow (x, y) \in s \Longrightarrow (y, z) \in ns \Longrightarrow (x, z) \in s$
and *nn*: $\bigwedge x\ y\ z.\ x \in\# X \Longrightarrow y \in\# Y \Longrightarrow z \in\# Z \Longrightarrow (x, y) \in ns \Longrightarrow (y, z) \in ns \Longrightarrow (x, z) \in ns$
and *xyz*: $(X, Y) \in mult2\text{-}alt\ b1\ ns\ s\ (Y, Z) \in mult2\text{-}alt\ b2\ ns\ s$
shows $(X, Z) \in mult2\text{-}alt\ (b1 \wedge b2)\ ns\ s$
proof −
let $?a1 = Enum.finite\text{-}3.a_1$ **and** $?a2 = Enum.finite\text{-}3.a_2$ **and** $?a3 = Enum.finite\text{-}3.a_3$
let $?t = \{(?a1, ?a2), (?a1, ?a3), (?a2, ?a3)\}$
let $?A = \{(?a1, x)\ |x.\ x \in\# X\} \cup \{(?a2, y)\ |y.\ y \in\# Y\} \cup \{(?a3, z)\ |z.\ z \in\# Z\}$
define s' **where** $s' = Restr\ \{((a, x), (b, y))\ |a\ x\ b\ y.\ (a, b) \in ?t \wedge (x, y) \in s\}\ ?A$
define ns' **where** $ns' = (Restr\ \{((a, x), (b, y))\ |a\ x\ b\ y.\ (a, b) \in ?t \wedge (x, y) \in ns\}\ ?A)^=$
have $*$: *refl ns' trans ns' trans s' s' O ns'* $\subseteq$ *s' ns' O s'* $\subseteq$ *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* ⊆# *Y* (*Y*, *Z*) ∈ *mult2-alt b ns s b* ⟹ *b'*
shows (*X*, *Z*) ∈ *mult2-alt b' ns s*
proof −
from *assms*(*2*) **obtain** *Y1 Y2 Z1 Z2* **where** ∗: *Y* = *Y1* + *Y2 Z* = *Z1* + *Z2* (*Y1*, *Z1*) ∈ *multpw ns b* ∨ *Z2* ≠ {#} ∀ *y*. *y* ∈# *Y2* ⟶ (∃ *z*. *z* ∈# *Z2* ∧ (*y*, *z*) ∈ *s*)
unfolding *mult2-alt-def* **by** *blast*
define *Y11 Y12 X2* **where** *Y11* = *Y1* ∩# *X* **and** *Y12* = *Y1* − *X* **and** *X2* = *X* − *Y11*
have ∗∗: *X* = *Y11* + *X2 X2* ⊆# *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) ∈ *multpw ns*
using ∗(*2*,*3*) ∗∗(*3*) **by** (*auto elim*: *multpw-splitR simp*: *ac-simps*)
moreover have ∀ *y*. *y* ∈# *X2* ⟶ (∃ *z*. *z* ∈# *Z12* + *Z2* ∧ (*y*, *z*) ∈ *s*) *b* ∨ *Z12* + *Z2* ≠ {#}
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* ∈# *A* ⟹ *x* ≠ *y* ⟹ *x* ∈# (*A* − {#*y*#})
by (*metis add-mset-remove-trivial-If diff-single-trivial insert-noteq-member*)

lemma *mult2-alt-addL*: (*add-mset x X*, *Y*) ∈ *mult2-alt b ns s* ⟷
(∃ *y*. *y* ∈# *Y* ∧ (*x*, *y*) ∈ *s* ∧ ({# *x* ∈# *X*. (*x*, *y*) ∉ *s* #}, *Y* − {#*y*#}) ∈ *mult2-alt-ns ns s*) ∨
(∃ *y*. *y* ∈# *Y* ∧ (*x*, *y*) ∈ *ns* ∧ (*x*, *y*) ∉ *s* ∧ (*X*, *Y* − {#*y*#}) ∈ *mult2-alt b ns s*)
(**is** *?L* ⟷ *?R1* ∨ *?R2*)
proof (*intro iffI*; (*elim disjE*)*?*)
assume *?L* **then obtain** *X1 X2 Y1 Y2* **where** ∗: *add-mset x X* = *X1* + *X2 Y* = *Y1* + *Y2*
(*X1*, *Y1*) ∈ *multpw ns b* ∨ *Y2* ≠ {#} ∀ *x*. *x* ∈# *X2* ⟶ (∃ *y*. *y* ∈# *Y2* ∧ (*x*, *y*) ∈ *s*)
unfolding *mult2-alt-def* **by** *blast*
from *union-single-eq-member*[*OF this*(*1*)] *multi-member-split*
consider *X1′* **where** *X1* = *add-mset x X1′ x* ∈# *X1* | *X2′* **where** *X2* = *add-mset x X2′ x* ∈# *X2*
unfolding *set-mset-union Un-iff* **by** *metis*
then show *?R1* ∨ *?R2*
proof *cases*
case *1* **then obtain** *y Y1′* **where** ∗∗: *y* ∈# *Y1 Y1* = *add-mset y Y1′* (*X1′*, *Y1′*) ∈ *multpw ns* (*x*, *y*) ∈ *ns*
using ∗ **by** (*auto elim*: *multpw-split1R*)
show *?thesis*
proof (*cases* (*x*, *y*) ∈ *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* ∈# *X2*. (*x*, *y*) ∉ *s* #}
have *x3*: ∀ *x*. *x* ∈# *X2′* ⟶ (∃ *z*. *z* ∈# *Y2* ∧ (*x*, *z*) ∈ *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* (λ*x*. (*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* ⇒ *bool*, use pairs of type *bool* × *bool*

definition [*simp*]: *or2 a b* = (*fst a* ∨ *fst b*, *snd a* ∨ *snd b*)

context fixes *sns* :: *'a* ⇒ *'a* ⇒ *bool* × *bool*
begin

fun *mult2-impl* :: *'a list* ⇒ *'a list* ⇒ *bool* × *bool*
and *mult2-ex-dom* :: *'a* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* ⇒ *bool* × *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 length xs = length ys X = mset xs Y = mset ys
    ∀ i. i < length ys ⟶ (xs ! i, ys ! i) ∈ ns
```

shows $(X,\ Y) \in \mathit{multpw}\ \mathit{ns}$
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 $\forall\, i.\ i < \mathit{length}\ \mathit{ys} \longrightarrow (\mathit{xs}\ !\ i,\ \mathit{ys}\ !\ i) \in \mathit{ns}$ **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 $(X,\ Y) \in \mathit{multpw}\ \mathit{ns}$
obtains *xs ys* **where** $\mathit{length}\ \mathit{xs} = \mathit{length}\ \mathit{ys}\ X = \mathit{mset}\ \mathit{xs}\ Y = \mathit{mset}\ \mathit{ys}$
$\forall\, i.\ i < \mathit{length}\ \mathit{ys} \longrightarrow (\mathit{xs}\ !\ i,\ \mathit{ys}\ !\ i) \in \mathit{ns}$
using *assms*
proof (*induct X Y arbitrary*: *thesis rule*: *multpw.induct*)
case (*add x y X Y*)
then obtain *xs ys* **where** $\mathit{length}\ \mathit{xs} = \mathit{length}\ \mathit{ys}\ X = \mathit{mset}\ \mathit{xs}$
$Y = \mathit{mset}\ \mathit{ys}\ (\forall\, i.\ i < \mathit{length}\ \mathit{ys} \longrightarrow (\mathit{xs}\ !\ i,\ \mathit{ys}\ !\ i) \in \mathit{ns})$ **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 $(\geqslant, >)$ – usually written as (ns, s) in the sources – by just flipping the directions twice.

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

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

lemma *ns-mul-extE*:
assumes $(A,\ B) \in \mathit{ns\text{-}mul\text{-}ext}\ \mathit{ns}\ s$
obtains *A1 A2 B1 B2* **where** $A = A1 + A2$ **and** $B = B1 + B2$
and $(A1,\ B1) \in \mathit{multpw}\ \mathit{ns}$
and $\bigwedge b.\ b \in\# B2 \Longrightarrow \exists\, a.\ a \in\# A2 \wedge (a,\ b) \in s$
using *assms* **by** (*auto simp*: *ns-mul-ext-def multpw-converse elim*!: *mult2-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* ⇒ *'a rel* ⇒ *'a multiset rel*
where *s-mul-ext ns s* ≡ (*mult2-alt-s* (*ns*$^{-1}$) (*s*$^{-1}$))$^{-1}$

lemma *s-mul-extI*:
assumes *A* = *A1* + *A2* **and** *B* = *B1* + *B2*
and (*A1*, *B1*) ∈ *multpw ns*
and *A2* ≠ {#} **and** ⋀*b*. *b* ∈# *B2* ⟹ ∃ *a*. *a* ∈# *A2* ∧ (*a*, *b*) ∈ *s*
shows (*A*, *B*) ∈ *s-mul-ext ns s*
using *assms* **by** (*auto simp*: *s-mul-ext-def multpw-converse intro*!: *mult2-alt-sI*)

lemma *s-mul-extE*:
assumes (*A*, *B*) ∈ *s-mul-ext ns s*
obtains *A1 A2 B1 B2* **where** *A* = *A1* + *A2* **and** *B* = *B1* + *B2*
and (*A1*, *B1*) ∈ *multpw ns*
and *A2* ≠ {#} **and** ⋀*b*. *b* ∈# *B2* ⟹ ∃ *a*. *a* ∈# *A2* ∧ (*a*, *b*) ∈ *s*
using *assms* **by** (*auto simp*: *s-mul-ext-def multpw-converse elim*!: *mult2-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* ⊆ *ns'* *s* ⊆ *s'* **shows** *s-mul-ext ns s* ⊆ *s-mul-ext ns' s'*
unfolding *s-mul-ext-def* **using** *assms mono-mult2-alt*[*of ns*$^{-1}$ *ns'*$^{-1}$ *s*$^{-1}$ *s'*$^{-1}$] **by** *simp*

lemma *ns-mul-ext-mono*:
assumes *ns* ⊆ *ns'* *s* ⊆ *s'* **shows** *ns-mul-ext ns s* ⊆ *ns-mul-ext ns' s'*
unfolding *ns-mul-ext-def* **using** *assms mono-mult2-alt*[*of ns*$^{-1}$ *ns'*$^{-1}$ *s*$^{-1}$ *s'*$^{-1}$] **by** *simp*

lemma *s-mul-ext-local-mono*:
assumes *sub*: (*set-mset xs* × *set-mset ys*) ∩ *ns* ⊆ *ns'* (*set-mset xs* × *set-mset ys*) ∩ *s* ⊆ *s'*
and *rel*: (*xs*,*ys*) ∈ *s-mul-ext ns s*
shows (*xs*,*ys*) ∈ *s-mul-ext ns' s'*
using *rel s-mul-ext-mono*[*OF sub*] *mult2-alt-local*[*of ys xs False ns*$^{-1}$ *s*$^{-1}$]
by (*auto simp*: *s-mul-ext-def converse-Int ac-simps converse-Times*)

lemma *ns-mul-ext-local-mono*:
assumes *sub*: (*set-mset xs* × *set-mset ys*) ∩ *ns* ⊆ *ns'* (*set-mset xs* × *set-mset ys*) ∩ *s* ⊆ *s'*
and *rel*: (*xs*,*ys*) ∈ *ns-mul-ext ns s*
shows (*xs*,*ys*) ∈ *ns-mul-ext ns' s'*
using *rel ns-mul-ext-mono*[*OF sub*] *mult2-alt-local*[*of ys xs True ns*$^{-1}$ *s*$^{-1}$]
by (*auto simp*: *ns-mul-ext-def converse-Int ac-simps converse-Times*)

The empty multiset is the minimal element for these orders

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

lemma *ns-mul-ext-bottom-uniqueness*:
assumes $(\{\#\},A) \in$ *ns-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-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-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-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 *length as* = *length bs*
and $\forall\, i.\ i <$ *length bs* $\longrightarrow$ (*as* ! *i*, *bs* ! *i*) $\in$ *ns*
shows (*mset as*, *mset bs*) $\in$ *ns-mul-ext ns s*
using *assms* **by** (*auto intro*!: *ns-mul-extI*[*of* - - $\{\#\}$ - - $\{\#\}$] *multpw-listI*)

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-mul-ext ns s*
using *assms* **by** (*auto intro*!: *s-mul-extI*[*of* - $\{\#\}$ - - $\{\#\}$] *multpw-listI*)

Being stricly lesser than implies being lesser than

lemma *s-ns-mul-ext*:
assumes $(A,\ B) \in$ *s-mul-ext ns s*
shows $(A,\ B) \in$ *ns-mul-ext ns s*
using *assms* **by** (*simp add*: *s-mul-ext-def ns-mul-ext-def mult2-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-mset A*) $\subseteq$ *ns* **using** *assms* **by** (*auto simp*: *locally-refl-def*)

from *refl-multpw*[*of Id*] *multpw-local*[*of A A Id*] *mono-multpw*[*OF this*]
show *?thesis* **by** (*auto simp*: *refl-on-def*)
qed

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

lemma *ns-mul-ext-refl*:
assumes *refl ns*
shows $(A,\ A) \in$ *ns-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-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-mul-ext ns s*
using *assms* **unfolding** *ns-mul-ext-def s-mul-ext-def*
by (*auto intro*!: *converseI mult2-alt-ns-s-add mult2-alt-sI*[*of* - {#} - - {#}])

lemma *s-mul-ext-union-compat*:
assumes $(A,\ B) \in$ *s-mul-ext ns s*
and *locally-refl ns C*
shows $(A + C,\ B + C) \in$ *s-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-mul-ext ns s*
and *locally-refl ns C*
shows $(A + C,\ B + C) \in$ *ns-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-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-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* :: $('a \Rightarrow 'a \Rightarrow bool \times bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool \times bool$
where *mul-ext f xs ys* ≡ *let s* = {(*x*,*y*). *fst* (*f x y*)}; *ns* = {(*x*,*y*). *snd* (*f x y*)}
in ((*mset xs*,*mset ys*) ∈ *s-mul-ext ns s*, (*mset xs*, *mset ys*) ∈ *ns-mul-ext ns s*)

definition *smulextp f m n* ⟷ (*m*, *n*) ∈ *s-mul-ext* {(*x*, *y*). *snd* (*f x y*)} {(*x*, *y*). *fst* (*f x y*)}
definition *nsmulextp f m n* ⟷ (*m*, *n*) ∈ *ns-mul-ext* {(*x*, *y*). *snd* (*f x y*)} {(*x*, *y*). *fst* (*f x y*)}

lemma *smulextp-cong*[*fundef-cong*]:
assumes *xs1* = *ys1*
and *xs2* = *ys2*
and ⋀ *x x*′. *x* ∈# *ys1* ⟹ *x*′ ∈# *ys2* ⟹ *f x x*′ = *g x x*′
shows *smulextp f xs1 xs2* = *smulextp g ys1 ys2*
unfolding *smulextp-def*
proof
assume (*xs1*, *xs2*) ∈ *s-mul-ext* {(*x*, *y*). *snd* (*f x y*)} {(*x*, *y*). *fst* (*f x y*)}
from *s-mul-ext-local-mono*[*OF - - this*, *of* {(*x*, *y*). *snd* (*g x y*)} {(*x*, *y*). *fst* (*g x y*)}]
show (*ys1*, *ys2*) ∈ *s-mul-ext* {(*x*, *y*). *snd* (*g x y*)} {(*x*, *y*). *fst* (*g x y*)}
using *assms* **by** *force*
next
assume (*ys1*, *ys2*) ∈ *s-mul-ext* {(*x*, *y*). *snd* (*g x y*)} {(*x*, *y*). *fst* (*g x y*)}
from *s-mul-ext-local-mono*[*OF - - this*, *of* {(*x*, *y*). *snd* (*f x y*)} {(*x*, *y*). *fst* (*f x y*)}]
show (*xs1*, *xs2*) ∈ *s-mul-ext* {(*x*, *y*). *snd* (*f x y*)} {(*x*, *y*). *fst* (*f x y*)}
using *assms* **by** *force*
qed

lemma *nsmulextp-cong*[*fundef-cong*]:
assumes *xs1* = *ys1*
and *xs2* = *ys2*
and ⋀ *x x*′. *x* ∈# *ys1* ⟹ *x*′ ∈# *ys2* ⟹ *f x x*′ = *g x x*′
shows *nsmulextp f xs1 xs2* = *nsmulextp g ys1 ys2*
unfolding *nsmulextp-def*

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

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

lemma *mulextp-cong*[*fundef-cong*]:
assumes *xs1* = *ys1*
and *xs2* = *ys2*
and ⋀ *x x'*. *x* ∈# *ys1* ⟹ *x'* ∈# *ys2* ⟹ *f x x'* = *g x x'*
shows *mulextp f xs1 xs2* = *mulextp g ys1 ys2*
unfolding *mulextp-def* **using** *assms* **by** (*auto cong*: *nsmulextp-cong smulextp-cong*)

lemma *mset-s-mul-ext*:
(*mset xs*, *mset ys*) ∈ *s-mul-ext* {(*x*, *y*). *snd* (*f x y*)} {(*x*, *y*).*fst* (*f x y*)} ⟷
fst (*mul-ext f xs ys*)
by (*auto simp*: *mul-ext-def Let-def*)

lemma *mset-ns-mul-ext*:
(*mset xs*, *mset ys*) ∈ *ns-mul-ext* {(*x*, *y*). *snd* (*f x y*)} {(*x*, *y*).*fst* (*f x y*)} ⟷
snd (*mul-ext f xs ys*)
by (*auto simp*: *mul-ext-def Let-def*)

lemma *smulextp-mset-code*:
smulextp f (*mset xs*) (*mset ys*) ⟷ *fst* (*mul-ext f xs ys*)
unfolding *smulextp-def mset-s-mul-ext* **..**

lemma *nsmulextp-mset-code*:
nsmulextp f (*mset xs*) (*mset ys*) ⟷ *snd* (*mul-ext f xs ys*)
unfolding *nsmulextp-def mset-ns-mul-ext* **..**

lemma *nstri-mul-ext-map*:
assumes ⋀*s t*. *s* ∈ *set ss* ⟹ *t* ∈ *set ts* ⟹ *fst* (*order s t*) ⟹ *fst* (*order'* (*f s*) (*f t*))
and ⋀*s t*. *s* ∈ *set ss* ⟹ *t* ∈ *set ts* ⟹ *snd* (*order s t*) ⟹ *snd* (*order'* (*f s*) (*f t*))
and *snd* (*mul-ext order ss ts*)
shows *snd* (*mul-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 ⋀*s t*. *s* ∈ *set ss* ⟹ *t* ∈ *set ts* ⟹ *fst* (*order s t*) ⟹ *fst* (*order′* (*f s*) (*f t*))
and ⋀*s t*. *s* ∈ *set ss* ⟹ *t* ∈ *set ts* ⟹ *snd* (*order s t*) ⟹ *snd* (*order′* (*f s*) (*f t*))
and *fst* (*mul-ext order ss ts*)
shows *fst* (*mul-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*)

The non-strict order is transitive.

lemma *ns-mul-ext-trans*:
assumes *trans s trans ns compatible-l ns s compatible-r ns s refl ns*
and (*A*, *B*) ∈ *ns-mul-ext ns s*
and (*B*, *C*) ∈ *ns-mul-ext ns s*
shows (*A*, *C*) ∈ *ns-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*) ∈ *s-mul-ext ns s*
and (*B*, *C*) ∈ *s-mul-ext ns s*
shows (*A*, *C*) ∈ *s-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*) ∈ *s-mul-ext ns s*
and (*B*, *C*) ∈ *ns-mul-ext ns s*
shows (*A*, *C*) ∈ *s-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-mul-ext ns s*
and $(B,\ C) \in$ *s-mul-ext ns s*
shows $(A,\ C) \in$ *s-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-on s* $\{y\}$
shows *SN-on* (*s-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-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-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*: $\bigwedge s\ t\ u.\ [\![s \in set\ ss;\ t \in set\ ts;\ u \in set\ us]\!] \Longrightarrow$
$(snd\ (f\ s\ t) \wedge fst\ (f\ t\ u) \longrightarrow fst\ (f\ s\ u)) \wedge$
$(fst\ (f\ s\ t) \wedge snd\ (f\ t\ u) \longrightarrow fst\ (f\ s\ u)) \wedge$
$(snd\ (f\ s\ t) \wedge snd\ (f\ t\ u) \longrightarrow snd\ (f\ s\ u)) \wedge$
$(fst\ (f\ s\ t) \wedge fst\ (f\ t\ u) \longrightarrow fst\ (f\ s\ u))$
shows
$(snd\ (mul\text{-}ext\ f\ ss\ ts) \wedge fst\ (mul\text{-}ext\ f\ ts\ us) \longrightarrow fst\ (mul\text{-}ext\ f\ ss\ us)) \wedge$
$(fst\ (mul\text{-}ext\ f\ ss\ ts) \wedge snd\ (mul\text{-}ext\ f\ ts\ us) \longrightarrow fst\ (mul\text{-}ext\ f\ ss\ us)) \wedge$
$(snd\ (mul\text{-}ext\ f\ ss\ ts) \wedge snd\ (mul\text{-}ext\ f\ ts\ us) \longrightarrow snd\ (mul\text{-}ext\ f\ ss\ us)) \wedge$
$(fst\ (mul\text{-}ext\ f\ ss\ ts) \wedge fst\ (mul\text{-}ext\ f\ ts\ us) \longrightarrow fst\ (mul\text{-}ext\ f\ ss\ us))$
proof −
let *?s* = $\{(x,\ y).\ fst\ (f\ x\ y)\}^{-1}$ **and** *?ns* = $\{(x,\ y).\ snd\ (f\ x\ y)\}^{-1}$
have [*dest*]: $(mset\ ts,\ mset\ ss) \in mult2\text{-}alt\ b2\ ?ns\ ?s \Longrightarrow (mset\ us,\ mset\ ts) \in mult2\text{-}alt\ b1\ ?ns\ ?s \Longrightarrow$
$(mset\ us,\ mset\ ss) \in mult2\text{-}alt\ (b1 \wedge b2)\ ?ns\ ?s$ **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 $\bigwedge x\ x'.\ x \in set\ ys1 \Longrightarrow x' \in set\ ys2 \Longrightarrow f\ x\ x' = g\ x\ x'$
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*:
$(x,y) \in \{(a,b).\ fst\ (mul\text{-}ext\ g\ a\ b)\} \longleftrightarrow (mset\ x,\ mset\ y) \in (s\text{-}mul\text{-}ext\ \{(a,b).\ snd\ (g\ a\ b)\}\ \{(a,b).\ fst\ (g\ a\ b)\})$
unfolding *mul-ext-def* **by** (*simp add*: *Let-def*)

The next lemma is a local version of strong-normalization of the multiset 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 $SN\ \{(ys,\ xs).$
$(\forall y{\in}set\ ys.\ SN\text{-}on\ \{(s\ ,t).\ fst\ (g\ s\ t)\}\ \{y\}) \wedge$
$fst\ (mul\text{-}ext\ g\ ys\ xs)\}$
proof −
let *?R1* $= \lambda xs\ ys.\ \forall y{\in}\ set\ ys.\ SN\text{-}on\ \{(s\ ,t).\ fst\ (g\ s\ t)\}\ \{y\}$
let *?R2* $= \lambda xs\ ys.\ fst\ (mul\text{-}ext\ g\ ys\ xs)$
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* $= SN\text{-}on\ ?R$
{
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*: $\forall y.\ y \in set\ ys \longrightarrow SN\text{-}on\ ?s\ \{y\}$ **by** *simp*
with *in-multiset-in-set*[*of ys*] **have** $\forall y.\ y \in\# ?mys \longrightarrow SN\text{-}on\ ?s\ \{y\}$ **by** *simp*
from *SN-s-mul-ext-strong*[*OF OP this*] **and** *mul-ext-unfold*
have *SN-on* $\{(ys,xs).\ fst\ (mul\text{-}ext\ g\ ys\ xs)\}\ \{ys\}$ **by** *fast*
from *relation-inter*[*of ?R2 ?R1*] **and** *SN-on-weakening*[*OF this*]
have *SN-on ?R* $\{ys\}$ **by** *blast*
}
then have *Hyp*: $\forall\ ys\ xs.\ (ys,xs) \in ?R \longrightarrow SN\text{-}on\ ?R\ \{ys\}$ **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) ∈ ns-mul-ext ns s
    and (C,D) ∈ ns-mul-ext ns s
  shows (A + C, B + D) ∈ 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) ∈ 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}
```

$\lambda s\ ns.\ \{(as,\ bs).\ (mset\ as,\ mset\ bs) \in ns\text{-}mul\text{-}ext\ ns\ s\}$
proof −
let *?m* = *mset* :: (*'a list* ⇒ *'a multiset*)
let *?S* = $\lambda s\ ns.\ \{(as,\ bs).\ (?m\ as,\ ?m\ bs) \in s\text{-}mul\text{-}ext\ ns\ s\}$
let *?NS* = $\lambda s\ ns.\ \{(as,\ bs).\ (?m\ as,\ ?m\ bs) \in ns\text{-}mul\text{-}ext\ ns\ s\}$
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*
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 $\bigwedge a\ b.\ (a,\ b) \in S \Longrightarrow (f\ a,\ f\ b) \in S$
$\bigwedge a\ b.\ (a,\ b) \in NS \Longrightarrow (f\ a,\ f\ b) \in NS$
$(as,\ bs) \in ?S\ S\ NS$
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 $\bigwedge a\ b.\ (a,\ b) \in S \Longrightarrow (f\ a,\ f\ b) \in S$
$\bigwedge a\ b.\ (a,\ b) \in NS \Longrightarrow (f\ a,\ f\ b) \in NS$
$(as,\ bs) \in ?NS\ S\ NS$
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*
$\bigwedge i.\ i < length\ bs \Longrightarrow (as\ !\ i,\ bs\ !\ i) \in NS$
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-mul-ext ns s*
using *assms* **by** (*auto simp*: *s-mul-ext-def mult2-alt-s-single*)

lemma *ns-mul-ext-singleton* [*simp*, *intro*]:
$(a,\ b) \in ns \Longrightarrow (\{\#a\#\},\ \{\#b\#\}) \in$ *ns-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*:
$(a,\ b) \in s \Longrightarrow (\{\#a\#\},\ \{\#b\#\}) \in$ *ns-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-mul-ext W S*
proof −
have $(A + B,\ \{\#\} + B) \in$ *s-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-mul-ext W S*
shows $(A + B,\ C) \in$ *s-mul-ext W S*
using *assms*
proof −
have $(A + B,\ \{\#\} + C) \in$ *s-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-mul-ext W S*
shows $(A + B,\ C) \in$ *s-mul-ext W S*
using *assms*
proof −
have $(B + A,\ C + \{\#\}) \in$ *s-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 $\bigwedge x\ y.\ [\![x \in set\ xs;\ y \in set\ ys;\ fst\ (P\ x\ y)]\!] \Longrightarrow fst\ (P'\ x\ y)$
and $\bigwedge x\ y.\ [\![x \in set\ xs;\ y \in set\ ys;\ snd\ (P\ x\ y)]\!] \Longrightarrow snd\ (P'\ x\ y)$
shows
fst (*mul-ext P xs ys*) $\Longrightarrow$ *fst* (*mul-ext P' xs ys*)

snd (*mul-ext P xs ys*) ⟹ *snd* (*mul-ext P′ xs ys*)
unfolding *mul-ext-def Let-def fst-conv snd-conv*
proof −
assume *mem*: (*mset xs*, *mset ys*) ∈ *s-mul-ext* {(*x*, *y*). *snd* (*P x y*)} {(*x*, *y*). *fst* (*P x y*)}
show (*mset xs*, *mset ys*) ∈ *s-mul-ext* {(*x*, *y*). *snd* (*P′ x y*)} {(*x*, *y*). *fst* (*P′ x y*)}
by (*rule s-mul-ext-local-mono*[*OF - - mem*], *insert assms*, *auto*)
next
assume *mem*: (*mset xs*, *mset ys*) ∈ *ns-mul-ext* {(*x*, *y*). *snd* (*P x y*)} {(*x*, *y*). *fst* (*P x y*)}
show (*mset xs*, *mset ys*) ∈ *ns-mul-ext* {(*x*, *y*). *snd* (*P′ x y*)} {(*x*, *y*). *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*) ∈ *ns-mul-ext Id R*
obtains *X* **and** *Y* **and** *Z* **where** *M* = *X* + *Z* **and** *N* = *Y* + *Z*
and ∀ *y* ∈ *set-mset Y*. ∃ *x* ∈ *set-mset X*. (*x*, *y*) ∈ *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** ∀ *y* ∈ *set-mset Y*. ∃ *x* ∈ *set-mset X*. (*x*, *y*) ∈ *R*
shows (*M*, *N*) ∈ *ns-mul-ext Id R*
using *assms mult2-alt-nsI*[*of N Z Y M Z X Id* R^{-1}]
by (*auto simp*: *ns-mul-ext-def*)

lemma *s-mul-ext-IdE*:
assumes (*M*, *N*) ∈ *s-mul-ext Id R*
obtains *X* **and** *Y* **and** *Z* **where** *X* ≠ {#} **and** *M* = *X* + *Z* **and** *N* = *Y* + *Z*
and ∀ *y* ∈ *set-mset Y*. ∃ *x* ∈ *set-mset X*. (*x*, *y*) ∈ *R*
using *assms*
by (*auto simp*: *s-mul-ext-def elim*!: *mult2-alt-sE*) (*metis union-commute*)

lemma *s-mul-ext-IdI*:
assumes *X* ≠ {#} **and** *M* = *X* + *Z* **and** *N* = *Y* + *Z*
and ∀ *y* ∈ *set-mset Y*. ∃ *x* ∈ *set-mset X*. (*x*, *y*) ∈ *R*
shows (*M*, *N*) ∈ *s-mul-ext Id R*
using *assms mult2-alt-sI*[*of N Z Y M Z X Id* R^{-1}]
by (*auto simp*: *s-mul-ext-def ac-simps*)

lemma *mult-s-mul-ext-conv*:
assumes *trans R*
shows (*mult* $(R^{-1})^{-1}$ = *s-mul-ext Id R*
using *mult2-s-eq-mult2-s-alt*[*of Id* R^{-1}] *assms*

by (*auto simp*: *s-mul-ext-def refl-Id mult2-s-def*)

lemma *ns-mul-ext-Id-eq*:
ns-mul-ext Id R = (*s-mul-ext Id R*)$^=$
by (*auto simp add*: *ns-mul-ext-def s-mul-ext-def mult2-alt-ns-conv*)

lemma *subseteq-mset-imp-ns-mul-ext-Id*:
assumes *A* $\subseteq\#$ *B*
shows (*B*, *A*) ∈ *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*) ∈ *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*) ∈ *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*: ∃ *s ns*. *list-order-extension-impl s ns mul-ext*
proof(*intro exI*)
let *?s* = λ *s ns*. {(*as*,*bs*). (*mset as*, *mset bs*) ∈ *s-mul-ext ns s*}
let *?ns* = λ *s ns*. {(*as*,*bs*). (*mset as*, *mset bs*) ∈ *ns-mul-ext ns s*}
let *?m* = *mset*
show *list-order-extension-impl ?s ?ns mul-ext*
proof
fix *s ns*
show *?s* {(*a*,*b*). *s a b*} {(*a*,*b*). *ns a b*} = {(*as*,*bs*). *fst* (*mul-ext* (λ *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} = {(as,bs). snd (mul-ext (λ 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 × set bs ∩ ns ⊆ ns'
           set as × set bs ∩ s ⊆ s'
           (as,bs) ∈ ?s s ns
    then show (as,bs) ∈ ?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 × set bs ∩ ns ⊆ ns'
           set as × set bs ∩ s ⊆ s'
           (as,bs) ∈ ?ns s ns
    then show (as,bs) ∈ ?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 ⇒ 'a ⇒ bool × bool
begin

fun mul-ext-impl :: 'a list ⇒ 'a list ⇒ bool × bool
and mul-ex-dom :: 'a list ⇒ 'a list ⇒ 'a ⇒ 'a list ⇒ bool × 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, -) ⇒ if snd (mul-ext-impl (xs @ xs') (filter (λy. ¬ fst (sns x y)) ys))
then (True, True)
                 else mul-ex-dom xs (x # xs') y ys
    | (False, True) ⇒ or2 (mul-ext-impl (xs @ xs') ys) (mul-ex-dom xs (x # xs') y
ys)
    | - ⇒ mul-ex-dom xs (x # xs') y ys)

end

context
begin
lemma mul-ext-impl-sound0:
```

mul-ext-impl sns xs ys = mult2-impl (λ*x y. sns y x*) *ys xs*
mul-ex-dom sns xs xs′ y ys = mult2-ex-dom (λ*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 ≡
((∃ *b*. *b* ∈ *set bs* ∧ *fst* (*f b y*) ∧ *snd* (*mul-ext f* (*remove1 b xs*) [*y*←*ys* . ¬ *fst* (*f b y*)]))
∨ (∃ *b*. *b* ∈ *set bs* ∧ *snd* (*f b y*) ∧ *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 ≡ (*cond1 f bs y xs ys*
∨ (∃ *b*. *b* ∈ *set bs* ∧ *snd* (*f b y*) ∧ *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 ⋀*x x′*. *x* ∈ *set xs* ⟹ *x′* ∈ *set ys* ⟹ *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*) ∈ *set xs* ⟹ *count* (*mset* (*ass-list-to-single-list xs*)) *x* ≥ *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-list-to-single-list* (*DAList.impl-of xs*))
by (*intro multiset-eqI*, *induct xs*) (*auto simp*: *Alist-inverse*)

lemma *Bag-Alist-Cons*:
x ∉ *fst* ‘ *set xs* ⟹ *distinct* (*map fst xs*) ⟹
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*) ⟹ *mset* (*ass-list-to-single-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*) ∈ *s-mul-ext* {(*x*, *y*). *snd* (*f x y*)} {(*x*, *y*). *fst* (*f x y*)} ⟷
fst (*mul-ext f* (*ass-list-to-single-list* (*DAList.impl-of xs*)) (*ass-list-to-single-list* (*DAList.impl-of ys*)))
by (*auto simp*: *mul-ext-def Let-def Alist-impl-of*)

lemma *Bag-ns-mul-ext*:
(*Bag xs*, *Bag ys*) ∈ *ns-mul-ext* {(*x*, *y*). *snd* (*f x y*)} {(*x*, *y*). *fst* (*f x y*)} ⟷
snd (*mul-ext f* (*ass-list-to-single-list* (*DAList.impl-of xs*)) (*ass-list-to-single-list* (*DAList.impl-of ys*)))
by (*auto simp*: *mul-ext-def Let-def Alist-impl-of*)

lemma *smulextp-code*[*code*]:
smulextp f (*Bag xs*) (*Bag ys*) ⟷ *fst* (*mul-ext f* (*ass-list-to-single-list* (*DAList.impl-of xs*)) (*ass-list-to-single-list* (*DAList.impl-of ys*)))

unfolding *smulextp-def Bag-s-mul-ext* **..**

lemma *nsmulextp-code*[*code*]:
nsmulextp f (*Bag xs*) (*Bag ys*) ⟷ *snd* (*mul-ext f* (*ass-list-to-single-list* (*DAList.impl-of xs*)) (*ass-list-to-single-list* (*DAList.impl-of ys*)))
unfolding *nsmulextp-def Bag-ns-mul-ext* **..**

lemma *mulextp-code*[*code*]:
mulextp f (*Bag xs*) (*Bag ys*) = *mul-ext f* (*ass-list-to-single-list* (*DAList.impl-of xs*)) (*ass-list-to-single-list* (*DAList.impl-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
Knuth-Bendix-Order.Term-Aux
Knuth-Bendix-Order.Order-Pair
Polynomial-Factorization.Missing-List
Status
Precedence
Multiset-Extension2
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 =
(*case s of*
Var x ⇒ (*False*,
(*case t of*

```
      Var y ⇒ x = y
    | Fun g ts ⇒ (s, t) ∈ NS ∧ status σσ (g, length ts) = [] ∧ prl (g, length ts)))
  | Fun f ss ⇒
    if (s, t) ∈ S then (True, True)
    else if (s, t) ∈ NS then
     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 ≻ 50) where s ≻ t ≡ fst (wpo s t)
abbreviation wpo-ns (infix ≽ 50) where s ≽ t ≡ snd (wpo s t)

abbreviation WPO-S ≡ {(s,t). s ≻ t}
abbreviation WPO-NS ≡ {(s,t). s ≽ t}

lemma wpo-s-imp-ns: s ≻ t ⟹ s ≽ 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)

end

declare wpo.wpo.simps[code]


definition strictly-simple-status :: 'f status ⇒ ('f,'v)term rel ⇒ bool where
  strictly-simple-status σ rel =
    (∀ f ts i. i ∈ set (status σ (f,length ts)) ⟶ (Fun f ts, ts ! i) ∈ rel)

locale wpo-with-assms = wpo +
  SN-order-pair + precedence +
```

constrains $S :: ('f, 'v)\ term\ rel$ **and** $NS :: -$
and $prc :: 'f \times nat \Rightarrow 'f \times nat \Rightarrow bool \times bool$
and $prl :: 'f \times nat \Rightarrow bool$
and $ssimple :: bool$
and $large :: 'f \times nat \Rightarrow bool$
and $c :: 'f \times nat \Rightarrow order\text{-}tag$
and $n :: nat$
and $\sigma\sigma :: 'f\ status$
assumes *subst-S*: $(s,t) \in S \Longrightarrow (s \cdot \sigma,\ t \cdot \sigma) \in S$
and *subst-NS*: $(s,t) \in NS \Longrightarrow (s \cdot \sigma,\ t \cdot \sigma) \in NS$
and *ctxt-NS*: $(s,t) \in NS \Longrightarrow (Fun\ f\ (bef\ @\ s\ \#\ aft),\ Fun\ f\ (bef\ @\ t\ \#\ aft)) \in NS$
and *S-imp-NS*: $S \subseteq NS$
and *ws-status*: $i \in set\ (status\ \sigma\sigma\ fn) \Longrightarrow simple\text{-}arg\text{-}pos\ NS\ fn\ i$
and *large*: $ssimple \Longrightarrow large\ fn \Longrightarrow fst\ (prc\ fn\ gm) \vee snd\ (prc\ fn\ gm) \wedge status\ \sigma\sigma\ gm = []$
and *large-trans*: $ssimple \Longrightarrow large\ fn \Longrightarrow snd\ (prc\ gm\ fn) \Longrightarrow large\ gm$
and *ss-status*: $ssimple \Longrightarrow i \in set\ (status\ \sigma\sigma\ fn) \Longrightarrow simple\text{-}arg\text{-}pos\ S\ fn\ i$
and *ss-S-non-empty*: $ssimple \Longrightarrow S \neq \{\}$
begin

lemma *ss-NS-not-UNIV*: $ssimple \Longrightarrow NS \neq UNIV$
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 *SN* **show** *False* **by** *fast*
qed

abbreviation $\sigma \equiv status\ \sigma\sigma$

lemmas $\sigma = status[of\ \sigma\sigma]$

lemma *NS-arg*: **assumes** *i*: $i \in set\ (\sigma\ (f,length\ ts))$
shows $(Fun\ f\ ts,\ ts\ !\ i) \in NS$
by (*rule ws-status*[*OF i*, *unfolded simple-arg-pos-def fst-conv*, *rule-format*],
insert σ[*of f length ts*] *i*, *auto*)

lemma *NS-subterm*: **assumes** *all*: $\bigwedge f\ k.\ set\ (\sigma\ (f,k)) = \{0\ ..<\ k\}$
shows $s \trianglerighteq t \Longrightarrow (s,t) \in NS$
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** $(Fun\ f\ ss,\ s) \in NS$ **by** *auto*

```
  from trans-NS-point[OF this subt(3)] show ?case .
qed


lemma σE: i ∈ set (σ (f, length ss)) ⟹ ss ! i ∈ set ss by (rule status-aux)

lemma wpo-ns-refl:
  shows s ≿ 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 σE[OF si]] have wpo-s (Fun f ss) (ss ! i) unfolding wpo.simps[of
Fun f ss 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 prc-refl wpo-s
    by (cases c (f,length ss), auto simp: wpo.simps[of Fun f ss Fun f ss])
qed (simp add: wpo.simps)

lemma wpo-ns-imp-NS: s ≿ t ⟹ (s,t) ∈ 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 ≻ t ⟹ (s,t) ∈ NS
  by (rule wpo-ns-imp-NS[OF wpo-s-imp-ns])

lemma S-imp-wpo-s: assumes st: (s,t) ∈ S
  shows s ≻ t
proof (cases s)
  case (Fun f ss)
  then show ?thesis using st by (auto simp: wpo.simps)
next
  case (Var x)
  from SN-imp-minimal[OF SN, rule-format, of undefined UNIV]
  obtain s where ⋀ u. (s,u) ∉ S by blast
  with subst-S[OF st[unfolded Var], of λ -. s]
  have False by auto
```

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

lemma *wpo-least-1*: **assumes** *prl* (*f*,*length ss*)
and (*t*, *Fun f ss*) ∈ *NS*
and *σ* (*f*,*length ss*) = []
shows *t* ≽ *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*) ∉ *S*
and *σ* (*f*,*length ss*) = []
shows ¬ *Fun f ss* ≻ *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* ≽ *t*
and *NS*: (*u*, *Fun f ss*) ∈ *NS*
and *ss*: *σ* (*f*,*length ss*) = []
and *S*: ⋀ *x*. (*Fun f ss*, *x*) ∉ *S*
and *u*: *u* = *Var x*
shows *u* ≽ *t*
proof (*cases* (*Fun f ss*, *t*) ∈ *S* ∨ (*u*, *Fun f ss*) ∈ *S* ∨ (*u*, *t*) ∈ *S*)

```
  case True
  with wpo-ns-imp-NS[OF ns] NS compat-NS-S-point compat-S-NS-point have (u,
t) ∈ 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) ∈ 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) ∈ NS .
      from False subst-NS[OF this, of λ z. if z = x then v else w for v w]
      have (v,w) ∈ 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 show wpo-s s u by (cases u, auto)
      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› have t = u by auto
          with gr show ?thesis by auto
        next
          case (Fun h us)
          let ?h = (h,length us)
          from Fun ge Var have us: σ ?h = [] and pri: prl ?h by auto
```

```
        from gr Var obtain f ss where s: s = Fun f ss by (cases s, auto)
        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 auto
        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› have t = u by auto
        with ge1 show ?thesis by auto
      next
        case (Fun h us)
        let ?h = (h,length us)
        from Fun ge2 Var have us: σ ?h = [] and pri: prl ?h by auto
        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 show wpo-s s u by (auto simp: wpo.simps)
      next
        assume ge: wpo-ns s t and gr: wpo-s t u
        with Var Fun have pri: prl ?g and σ ?g = [] by (auto simp: wpo.simps)
       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 have pri: prl ?g and empty: σ ?g = [] by (auto simp:
wpo.simps)
        from wpo-ns-imp-NS[OF ge1] Var Fun empty have ns: (Var x, Fun g ts)
```

$\in$ *NS* **by** *simp*
show *wpo-ns s u*
proof (*rule wpo-least-3*[*OF pri ge2*[*unfolded Fun empty*]
wpo-ns-imp-NS[*OF ge1*[*unfolded Fun empty*]] *empty - Var*], *rule*)
fix *v*
assume *S*: (*Fun g ts*, *v*) $\in$ *S*
from *SN-imp-minimal*[*OF SN*, *rule-format*, *of undefined UNIV*]
obtain *s* **where** $\bigwedge$ *u*. (*s*,*u*) $\notin$ *S* **by** *blast*
with *compat-NS-S-point*[*OF subst-NS*[*OF ns*, *of* λ *-. s*] *subst-S*[*OF S*, *of* λ *-. s*]]
show *False* **by** *auto*
qed
qed
next
case (*Fun f ss*)
let *?s* = *Fun f ss*
let *?f* = (*f*,*length ss*)
let *?ss* = *set* (σ *?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* = λ *a b c*.
((*wpo-ns a b*) $\wedge$ (*wpo-s b c*) $\longrightarrow$ (*wpo-s a c*)) $\wedge$
((*wpo-s a b*) $\wedge$ (*wpo-ns b c*) $\longrightarrow$ (*wpo-s a c*)) $\wedge$
((*wpo-ns a b*) $\wedge$ (*wpo-ns b c*) $\longrightarrow$ (*wpo-ns a c*)) $\wedge$
((*wpo-s a b*) $\wedge$ (*wpo-s b c*) $\longrightarrow$ (*wpo-s a c*))
from *s* **have** $\forall$ *s'* $\in$ *set ss*. *size s'* < *size s* **by** (*auto simp*: *size-simps*)
with *ind* **have** *ind2*: $\bigwedge$ *s' t' u'*. $[\![$*s'* $\in$ *set ss*$]\!] \Longrightarrow$ *?tran s' t' u'* **by** *blast*
with *wpo-s-imp-ns* **have** *ind3*: $\bigwedge$ *us s' t' u'*. $[\![$*s'* $\in$ *set ss*; *t'* $\in$ *set ts*$]\!] \Longrightarrow$ *?tran2 s' t' u'* **by** *blast*
let *?mss* = *map* (λ *i*. *ss* ! *i*) (σ *?f*)
let *?mts* = *map* (λ *j*. *ts* ! *j*) (σ *?g*)
have *ind3'*: $\bigwedge$ *us s' t' u'*. $[\![$*s'* $\in$ *set ?mss*; *t'* $\in$ *set ?mts*$]\!] \Longrightarrow$ *?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*) $\in$ *NS* **.**
from *wpo-s-imp-NS*[*OF ge2*] **have** *tuA*: (*t*,*u*) $\in$ *NS* **.**
from *trans-NS-point*[*OF stA tuA*] **have** *suA*: (*s*,*u*) $\in$ *NS* **.**
have *s* $\succ$ *u*
proof (*cases ?s1*)
case *True*
from *this* **obtain** *i* **where** *i*: *i* $\in$ *?ss* **and** *ges*: *ss* ! *i* $\succeq$ *t* **by** *auto*
from σ*E*[*OF i*] **have** *s'*: *ss* ! *i* $\in$ *set ss* **.**
with *i s s' ind2*[*of ss* ! *i t u*, *simplified*] *ges ge2* **have** *ss* ! *i* $\succ$ *u* **by** *auto*
then have *ss* ! *i* $\succeq$ *u* **by** (*rule wpo-s-imp-ns*)

```
      with i s suA show ?thesis by (cases u, auto simp: wpo.simps)
    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 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 (σ ?h)
        let ?mus = map (λ k. us ! k) (σ ?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 ‹¬ ?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 $\sigma E[OF\ this]$ **have** $size\ (us\ !\ k) < size\ u$ **unfolding** u **using** *size-simps* **by** *auto*
with $tu'[folded\ t]\ \langle s \succeq t\rangle$
$ind[rule\text{-}format,\ of\ s\ t\ us\ !\ k]\ k$ **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 $ge1 = ge1[unfolded * if\text{-}False]$
note $ge2 = ge2[unfolded * if\text{-}False]$
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 $\bigwedge f.\ snd\ (mul\text{-}ext\ f\ []\ ?mts) \Longrightarrow ?mts = []$ **unfolding** *mul-ext-def* **by** (*simp add*: *Let-def*)
with *ts-ne fg* ⟨¬ *?s1*⟩ *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*
qed
next
case *Lex* **note** $cg = this$
from *fg* ⟨¬ *?s1*⟩ *t ge1 st' prc nfg cf cg* **have** *ts-e*: *?mts* = [] **by** *simp*
with *gh* ⟨¬ *?t1*⟩ *u ge2 tu' prc2 ngh cg* **show** *?thesis*

```
            by (cases c ?h) (simp-all add: lex-ext-least-2)
          qed
        next
          case Lex note cf = this
          show ?thesis
          proof (cases c ?g)
            case Mul note cg = this
          from fg ‹¬ ?s1› t ge1 st' prc nfg cf cg have ts-e: ?mts = [] by simp
            with gh ‹¬ ?t1› u ge2 tu' prc2 ngh cg show ?thesis
            by (cases c ?h) (auto simp: Let-def s-mul-ext-def s-mul-ext-bottom
mul-ext-def elim: mult2-alt-sE)
          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
              from gh u ge2 tu' ngh cg ch have ts-ne: ?mts ≠ [] by simp
              from lex-ext-iff[of - - [] ?mts]
              have ⋀f. snd (lex-ext f n [] ?mts) ⟹ ?mts = [] by simp
              with ts-ne fg t ge1 st' nfg cf cg have ss-ne: ?mss ≠ [] by 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
            next
              case Lex note ch = this
              from fg t ge1 st' nfg cf cg
              have lex1: snd (lex-ext wpo n ?mss ?mts) by auto
              from gh u ge2 tu' ngh cg ch
              have lex2: fst (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
    qed
  }
  moreover
  {
    assume ge1: s ≻ t and ge2: t ≽ u
    from wpo-s-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 .
    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)
    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 σE(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 ⋀f. snd (lex-ext f n [] ?mus) ⟹ ?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 ⋀f. fst (lex-ext f n ?mss ?mts) ⟹ ?mss ≠ []
                    by (cases ?mss) (simp-all add: lex-ext-iff)
                  with fg t ge1 st' prc nfg cf cg have ss-ne: ?mss ≠ [] 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 ‹¬ ?t1› 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) ∨ snd (prc ?g ?f) ∧ σ ?f = [] by auto
         obtain fgs fgns where prc-fg: prc ?f ?g = (fgs,fgns) by (cases prc ?f
?g, auto)
           from ge1 ‹¬ ?s1› stS have weak-fg: snd (prc ?f ?g) unfolding s t
using prc-fg
            by (auto simp: wpo.simps split: if-splits)
          from refl-not-SN[of ?f] prc-SN have prc-irrefl: ¬ fst (prc ?f ?f) by
auto
          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) ∧ σ ?f = []
            let ?mss = map (λ i. ss ! i) (σ ?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 ⪰ ?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]
from s t u ngt1 ngt2 have ngt1: ¬ ?s ≻ ?t and ngt2: ¬ ?t ≻ ?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 ‹¬ ?s1› t ge1 have st': ∀ j ∈ ?ts. ?s ≻ ts ! j by (cases ?thesis,
auto)
from ‹¬ ?t1› u ge2 have tu': ∀ k ∈ ?us. ?t ≻ 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 ‹¬ ?s1› t ge1 st' have fg: pns by (cases ?thesis, auto simp: prc)
from ‹¬ ?t1› u ge2 tu' have gh: pns2 by (cases ?thesis, auto simp: prc2)
note compat = prc-compat[OF prc prc2 prc3]
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]
from compat fg gh have fh: pns3 by blast
{
  fix k
  assume k: k ∈ ?us
    from σE[OF this] have size (us ! k) < size u unfolding u using
size-simps by auto
  with tu'[folded t] ‹s ≽ t›
    ind[rule-format, of s t us ! k] k have s ≻ us ! k by blast
} note su' = this
from ‹¬ ?s1› st' ge1 ngt1 s t have nfg: ¬ ps
  by (simp, cases ?thesis, simp, cases ps, auto simp: prc fg)
from ‹¬ ?t1› tu' ge2 ngt2 t u have ngh: ¬ ps2
  by (simp, cases ?thesis, simp, cases ps2, auto simp: prc2 gh)
show s ≽ 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 mul1: 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 ⋀f. snd (lex-ext f n [] ?mus) ⟹ ?mus = [] 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
          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
      qed
    qed
   }
   ultimately
   show ?thesis using wpo-s-imp-ns by auto
  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)
qed

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-ns-pre-mono: fixes f and bef aft :: ('f,'v)term list
 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 ∀ j∈set ?σ. wpo-s ?s ((bef @ t # aft) ! j) (is ?one)
  proof
    fix j
    assume j: j ∈ set ?σ
    then have j ∈ set (σ (f,length ?ss)) by simp
    from subterm-wpo-s-arg[OF this]
    have s: wpo-s ?s (?ss ! j) .
    show wpo-s ?s (?ts ! j)
    proof (cases j = length 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 wpo-s ?s s by simp
      with rel wpo-compat have wpo-s ?s t by blast
      with True show ?thesis by simp
    qed
  qed
  with ‹?three› ctxt-NS[OF stA] show ?thesis unfolding σf-def by auto
qed

lemma wpo-ns-mono:
  assumes rel: s ≿ t
  shows Fun f (bef @ s # aft) ≿ Fun f (bef @ t # aft)
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-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*
have *snd* (*lex-ext wpo n* (*map* ((!) *?ss*) *?σ*) (*map* ((!) *?ts*) *?σ*))
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

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 δ y]
by auto
      next
        case (Fun g ts) note t = this
        let ?g = (g,length ts)
        show ?thesis
        proof (cases δ 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 (σ ?f)
      {
        fix i
        assume i: i ∈ ?ss and ns: wpo-ns (ss ! i) t
        from IH[of ss ! i t] σE[OF i] ns have wpo-ns (ss ! i · δ) ?t using s
          by (auto simp: size-simps)
        then have wpo-s ?s ?t using i sstsA σ[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 ∃ i∈?ss. wpo-ns (ss ! i) t)
        case True
        then obtain i
          where si: i ∈ ?ss and ns: wpo-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 ¬ wpo-s s t unfolding wpo.simps[of s t] by auto
        moreover
        have wpo-ns ?s ?t
        proof (cases δ y)
          case (Var z)
          show ?thesis unfolding wpo.simps[of ?s ?t] not id
            unfolding s t using Var ‹ssimple› 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 ⟹ pns by auto
          {
            fix j
            assume j ∈ set (σ ?g)
            with set-status-nth[OF refl this] ss-status[OF ‹ssimple› this] t Fun
            have (t · δ, ts ! j) ∈ S by (auto simp: simple-arg-pos-def)
            with sstsA have S: (s · δ, ts ! j) ∈ S by (metis compat-NS-S-point)
            hence wpo-s (s · δ) (ts ! j) by (rule S-imp-wpo-s)
          } note ssimple = this
          from large[OF ‹ssimple› largef, of ?g, unfolded prc]
          have ps ∨ pns ∧  σ ?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 (σ ?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* ∃ *i* ∈ *?ss*. *wpo-ns* (*ss* ! *i*) *t*)
case *True*
with *si-arg* **show** *?thesis* **by** *blast*
next
case *False*
then have *id*: (∃ *i* ∈ *?ss*. *wpo-ns* (*ss* ! *i*) (*Fun g ts*)) = *False* **unfolding** *t* **by** *auto*
note *ns* = *ns*[*unfolded this if-False*]
let *?mss* = *map* (λ *s* . *s* · *δ*) *ss*
let *?mts* = *map* (λ *t* . *t* · *δ*) *ts*
from *ns* **have** *prns* **and** *s-tj*: ⋀ *j*. *j* ∈ *?ts* ⟹ *wpo-s* (*Fun f ss*) (*ts* ! *j*)
by (*auto split*: *if-splits*)
{
fix *j*
assume *j*: *j* ∈ *?ts*
from *σE*[*OF this*]
have *size s* + *size* (*ts* ! *j*) < *size s* + *size t* **unfolding** *t* **by** (*auto simp*: *size-simps*)
from *IH*[*OF this*] *s-tj*[*OF j*, *folded s*] **have** *wpo*: *wpo-s ?s* (*ts* ! *j* · *δ*) **by** *auto*
from *j σ*[*of g length ts*] **have** *j* < *length ts* **by** *auto*
with *wpo* **have** *wpo-s ?s* (*?mts* ! *j*) **by** *auto*
} **note** *ss-ts* = *this*
note *σE* = *σE*[*of - f ss*] *σE*[*of - g ts*]
show *?thesis*
proof (*cases prs*)
case *True*
with *ss-ts sstsA p* ‹*prns*› **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
case *False*
let *?mmss* = *map* ((!) *ss*) (*σ ?f*)
let *?mmts* = *map* ((!) *ts*) (*σ ?g*)
let *?Mmss* = *map* ((!) *?mss*) (*σ ?f*)
let *?Mmts* = *map* ((!) *?mts*) (*σ ?g*)
have *id-map*: *?Mmss* = *map* (λ *t*. *t* · *δ*) *?mmss ?Mmts* = *map* (λ *t*. *t* · *δ*) *?mmts*
unfolding *map-map o-def* **by** (*auto simp*: *set-status-nth*)
let *?ls* = *length* (*σ ?f*)
let *?lt* = *length* (*σ ?g*)
{
fix *si tj*
assume ∗: *si* ∈ *set ?mmss tj* ∈ *set ?mmts*
have (*wpo-s si tj* ⟶ *wpo-s* (*si* · *δ*) (*tj* · *δ*)) ∧ (*wpo-ns si tj* ⟶ *wpo-ns*

```
(si · δ) (tj · δ))
            proof (intro IH add-strict-mono)
              from ∗(1) have si ∈ set ss using set-status-nth[of - - - σσ] by auto
            then show size si < size s unfolding s by (auto simp: termination-simp)
              from ∗(2) have tj ∈ set ts using set-status-nth[of - - - σσ] by auto
            then show size tj < size t unfolding t by (auto simp: termination-simp)
            qed
            hence wpo-s si tj ⟹ wpo-s (si · δ) (tj · δ)
              wpo-ns si tj ⟹ wpo-ns (si · δ) (tj · δ) by blast+
           } note IH' = this
           {
            fix i
            assume i < ?ls i < ?lt
            then have i-f: i < length (σ ?f) and i-g: i < length (σ ?g) by auto
           with σ[of f length ss] σ[of g length ts] have i: σ ?f ! i < length ss σ ?g
! i < length ts
              unfolding set-conv-nth by auto
            then have size (ss ! (σ ?f ! i)) < size s size (ts ! (σ ?g ! i)) < size t
unfolding s t by (auto simp: size-simps)
          then have size (ss ! (σ ?f ! i)) + size (ts ! (σ ?g ! i)) < size s + size t
by simp
            from IH[OF this] i i-f i-g
            have (wpo-s (?mmss ! i) (?mmts ! i) ⟹
                  wpo-s (?mss ! (σ ?f ! i)) (?mts ! (σ ?g ! i)))
              (wpo-ns (?mmss ! i) (?mmts ! i) ⟹
                  wpo-ns (?mss ! (σ ?f ! i)) (?mts ! (σ ?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 ≠ 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[unfolded lex-ext-iff snd-conv]
            have len: (?ls = ?lt ∨ ?lt ≤ n)
              and choice: (∃ i< ?ls.
              i < ?lt ∧ (∀ j<i. wpo-ns (?mmss ! j) (?mmts ! j)) ∧ wpo-s (?mmss !
i) (?mmts ! i)) ∨
                (∀ i< ?lt. wpo-ns (?mmss ! i) (?mmts ! i)) ∧ ?lt ≤ ?ls (is ?stri ∨
?nstri) by auto
            from choice have ?stri ∨ (¬ ?stri ∧ ?nstri) by blast
            then show ?thesis
            proof
              assume ?stri
              then obtain i where i: i < ?ls i < ?lt
                  and NS: (∀ j<i. wpo-ns (?mmss ! j) (?mmts ! j)) and S: wpo-s
(?mmss ! i) (?mmts ! i) by auto
```

```
          with IH have (∀ 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 ‹prns› 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 ¬ ?stri ∧ ?nstri
         then have ?nstri and nstri: ¬ ?stri by blast+
          with IH have (∀ i< ?lt. wpo-ns (?Mmss ! i) (?Mmts ! i)) ∧ ?lt ≤
?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 (∀ i< ?lt. wpo-ns (?mmss ! i) (?mmts ! i)) ∧ ?lt < ?ls by auto
         with IH have (∀ i< ?lt. wpo-ns (?Mmss ! i) (?Mmts ! i)) ∧ ?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 subst-apply-term.simps length-map
Mul
            by (simp add: id-map id)
        qed
      qed
    qed
  qed
 qed
qed

lemma WPO-S-SN: SN WPO-S
proof −
  {
   fix t :: ('f,'v)term
   let ?S = λx. 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) ∈ WPO-S
     then have False by (cases s, auto simp: wpo.simps)
     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 = λ f ys. ∀ i ∈ set (σ f). ?S (ys ! i)
    let ?r3 = {((f,ab), (g,ab')). ((c f = c g) ⟶ (?Slist f ab ∧
        (c f = Mul ⟶ fst (mul-ext wpo (map ((!) ab) (σ f)) (map ((!) ab') (σ
g)))) ∧
        (c f = Lex ⟶ fst (lex-ext wpo n (map ((!) ab) (σ f)) (map ((!) ab') (σ
g))))))
     ∧ ((c f ≠ c g) ⟶ (map ((!) ab) (σ f) ≠ [] ∧ (map ((!) ab') (σ g)) = [])))}
    let ?r0 = lex-two {(f,g). fst (prc f g)} {(f,g). snd (prc f g)} ?r3
    {
     fix ab
     {
      assume ∃ S. S 0 = ab ∧ (∀ i. (S i, S (Suc i)) ∈ ?r3)
```

```
      then obtain S where
        S0: S 0 = ab and
        SS: ∀ i. (S i, S (Suc i)) ∈ ?r3
        by auto
      let ?Sf = λi. fst (fst (S i))
      let ?Sn = λi. snd (fst (S i))
      let ?Sfn = λ i. fst (S i)
      let ?Sts = λi. snd (S i)
      let ?Stsσ = λi. map ((!) (?Sts i)) (σ (?Sfn i))
      have False
      proof (cases ∀ i. c (?Sfn i) = 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 ∃ i. c (?Sfn i) ≠ c (?Sfn (Suc i))
  proof (cases ?thesis, simp)
    case False
    then have T: ∀ 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) ≠ c (?Sfn (Suc i))
    by auto
  from Hdiff have Hf: ?Stsσ (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 (?Stsσ (Suc i)) (?Stsσ (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 (?Stsσ (Suc i)) (?Stsσ (Suc (Suc i))))
              by auto
            with Hf show ?thesis by (simp add: lex-ext-iff)
          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 ∃ i ∈ set (σ ?f). ts ! i ≽ ?s)
      case True
     then obtain i where i ∈ set (σ ?f) and ge: ts ! i ≽ ?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) ∈ WPO-S
        with wpo-compat ge have u: ts ! i ≻ u by blast
        with ‹?S (ts ! i)›[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) ∈ S)
        case True
        then have ((f,ts),(g,ss)) ∈ ?rrr unfolding rr-def by auto
        with ind have ind: ?ind (g,ss) by auto
        {
          fix i
          assume i: i ∈ set (σ ?g)
          have ?s ≽ ss ! i by (rule subterm-wpo-ns-arg[OF i])
          with t-gr-s have ts: ?t ≻ ss ! i using wpo-compat by blast
          have ?S (ss ! i) using IH(1)[OF σE[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 = length ss
        let ?lt = 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*: ∀ *i* ∈ *set* (*σ ?g*). *?t* ≻ *ss* ! *i*
by (*cases ?thesis*, *auto*)
with *Fun σE*[*of - g ss*] **have** *ss-S*: *?Slist ?g ss* **by** *auto*
show *?thesis*
proof (*cases s1*)
case *True*
then 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 *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
 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
 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 ⊆ WPO-S using wpo-compat by blast
 show WPO-S O WPO-NS ⊆ WPO-S using wpo-compat by blast
 show SN WPO-S using WPO-S-SN .
qed

theorem WPO-S-subst: (s,t) ∈ WPO-S ⟹ (s · σ, t · σ) ∈ WPO-S for σ
 using wpo-stable by auto

theorem WPO-NS-subst: (s,t) ∈ WPO-NS ⟹ (s · σ, t · σ) ∈ WPO-NS for σ
 using wpo-stable by auto

theorem WPO-NS-ctxt: (s,t) ∈ WPO-NS ⟹ (Fun f (bef @ s # aft), Fun f (bef
@ t # aft)) ∈ WPO-NS
 using wpo-ns-mono by blast

theorem WPO-S-subset-WPO-NS: WPO-S ⊆ WPO-NS
 using wpo-s-imp-ns by blast


context
 assumes σ-full: ⋀ f k. set (σ (f,k)) = {0 ..< k}
begin

lemma subterm-wpo-s: s ▷ t ⟹ s ≻ t
proof (induct s t rule: supt.induct)
 case (arg s ss f)
```

from *arg*[*unfolded set-conv-nth*] **obtain** *i* **where** *i*: $i < length\ ss$ **and** *s*: $s = ss\ !$ *i* **by** *auto*
from σ-*full*[*of f length ss*] *i* **have** *ii*: $i \in set\ (\sigma\ (f, length\ ss))$ **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** $\exists\ s \in set\ ss.\ wpo\text{-}ns\ s\ t$ **by** *blast*
from *this*[*unfolded set-conv-nth*] **obtain** *i* **where** *ns*: $ss\ !\ i \succeq t$ **and** *i*: $i < length\ ss$ **by** *auto*
from σ-*full*[*of f length ss*] *i* **have** *ii*: $i \in set\ (\sigma\ (f, length\ ss))$ **by** *auto*
from *subt* **have** $Fun\ f\ ss \trianglerighteq t$ **by** *auto*
from *NS-subterm*[*OF* σ-*full this*] *ns ii*
show *?case* **by** (*auto simp*: *wpo.simps*)
qed

lemma *subterm-wpo-ns*: **assumes** *supteq*: $s \trianglerighteq t$ **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 $Fun\ f\ (bef\ @\ s\ \#\ aft) \succ Fun\ f\ (bef\ @\ t\ \#\ aft)$
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 $?\sigma = \sigma$ *?f*
from *wpo-s-imp-ns*[*OF rels*] **have** *rel*: *wpo-ns s t* **.**
from *wpo-ns-pre-mono*[*OF rel*]
have *id*: $(\forall j{\in}set\ ?\sigma.\ wpo\text{-}s\ ?s\ ((bef\ @\ t\ \#\ aft)\ !\ j)) = True$
$((?s, ?t) \in NS) = True$
$length\ ?ss = ?len\ length\ ?ts = ?len$
by *auto*
let *?lb* = *length bef*
from σ-*full*[*of f ?len*] **have** *lb-mem*: *?lb* $\in$ *set* $?\sigma$ **by** *auto*
then obtain *i* **where** σi: $?\sigma\ !\ i = ?lb$ **and** *i*: $i < length\ ?\sigma$
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 ≠ ?lb by auto
      hence ?ss ! j = ?ts ! j by (auto simp: nth-append)
     thus (?mnlbs ! i, ?mnlbt ! i) ∈ WPO-NS unfolding id using wpo-ns-refl by
auto
    }
    fix u
    assume u ∈# mset ?mlbt
    hence u = t using part by auto
    moreover have s ∈# mset ?mlbs using lb by force
     ultimately show ∃ v. v ∈#  mset ?mlbs ∧ (v,u) ∈ WPO-S using rels by
force
  qed 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

theorem *WPO-S-ctxt*: $(s,t) \in$ *WPO-S* $\Longrightarrow$ (*Fun f* (*bef* @ *s* # *aft*), *Fun f* (*bef* @ *t* # *aft*)) $\in$ *WPO-S*
using *wpo-s-mono* **by** *blast*

theorem *supt-subset-WPO-S*: $\{\rhd\} \subseteq$ *WPO-S*
using *subterm-wpo-s* **by** *blast*

theorem *supteq-subset-WPO-NS*: $\{\trianglerighteq\} \subseteq$ *WPO-NS*
using *subterm-wpo-ns* **by** *blast*

end

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* :: $'f \times nat \Rightarrow 'f \times nat \Rightarrow bool \times bool$
and *prl* :: $'f \times nat \Rightarrow bool$
and *c* :: $'f \times nat \Rightarrow$ *order-tag*
and *n* :: *nat*
begin

fun *rpo* :: $('f, 'v)$ *term* $\Rightarrow$ $('f, 'v)$ *term* $\Rightarrow$ *bool* $\times$ *bool*
where
rpo (*Var x*) (*Var y*) = (*False*, *x* = *y*) |
rpo (*Var x*) (*Fun g ts*) = (*False*, *ts* = [] $\wedge$ *prl* (*g*,*0*)) |
rpo (*Fun f ss*) (*Var y*) = (*let con* = ($\exists$ *s* $\in$ *set ss*. *snd* (*rpo s* (*Var y*))) *in* (*con*,*con*)) |
rpo (*Fun f ss*) (*Fun g ts*) = (
if ($\exists$ *s* $\in$ *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 $\wedge$ ($\forall$ *t* $\in$ *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 ∧ c (g,length ts) = Mul
                      then mul-ext rpo ss ts
                      else (length ss ≠ 0 ∧ length ts = 0, length ts = 0)
         else (False,False)))
end

locale rpo-with-assms = precedence prc prl
  for prc :: 'f × nat ⇒ 'f × nat ⇒ bool × bool
    and prl :: 'f × nat ⇒ bool
    and c :: 'f × nat ⇒ order-tag
    and n :: nat
begin

sublocale wpo-with-assms n {} UNIV prc prl full-status c False λ -. False
  by (unfold-locales, auto simp: refl-on-def trans-def simple-arg-pos-def)

abbreviation rpo-pr ≡ rpo prc prl c n
abbreviation rpo-s ≡ λ s t. fst (rpo-pr s t)
abbreviation rpo-ns ≡ λ 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: ⋀ 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. 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 ¬ (∃ *si*∈*set ss*. *rpo-ns si* (*Fun g ts*))
note *IH* = *IH*(*2*−)[*OF this prc*[*symmetric*] *refl*]
from *IH*(*1*) **show** (∀ *j*∈*set* (*σ* (*g*, *length ts*)). *wpo-s* (*Fun f ss*) (*ts* ! *j*)) = (∀ *t*∈*set ts*. *rpo-s* (*Fun f ss*) *t*)
unfolding *set-conv-nth* **by** *auto*
assume *prns* ∧ (∀ *t*∈*set ts*. *rpo-s* (*Fun f ss*) *t*) ¬ *prs*
note *IH* = *IH*(*2*−)[*OF this*]
{
assume *c* (*f*, *length ss*) = *Lex* ∧ *c* (*g*, *length ts*) = *Lex*
from *IH*(*1*)[*OF this*]
show *lex-ext wpo n ss ts* = *lex-ext rpo-pr n ss ts*
by (*intro lex-ext-cong*, *auto*)
}
{
assume ¬ (*c* (*f*, *length ss*) = *Lex* ∧ *c* (*g*, *length ts*) = *Lex*) *c* (*f*, *length ss*) = *Mul* ∧ *c* (*g*, *length ts*) = *Mul*
from *IH*(*2*)[*OF this*]
show *mul-ext wpo ss ts* = *mul-ext rpo-pr ss ts*
by (*intro mul-ext-cong*, *auto*)
}
qed *auto*
qed
qed

abbreviation *RPO-S* ≡ {(*s*,*t*). *rpo-s s t*}
abbreviation *RPO-NS* ≡ {(*s*,*t*). *rpo-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*) ∈ *RPO-S* ⟹ (*s* · *σ*, *t* · *σ*) ∈ *RPO-S* **for** *σ* :: (′*f*,′*a*)*subst*
using *WPO-S-subst* **unfolding** *rpo-eq-wpo* **.**

theorem *RPO-NS-subst*: (*s*,*t*) ∈ *RPO-NS* ⟹ (*s* · *σ*, *t* · *σ*) ∈ *RPO-NS* **for** *σ* :: (′*f*,′*a*)*subst*
using *WPO-NS-subst* **unfolding** *rpo-eq-wpo* **.**

theorem *RPO-NS-ctxt*: (*s*,*t*) ∈ *RPO-NS* ⟹ (*Fun f* (*bef* @ *s* # *aft*), *Fun f* (*bef* @ *t* # *aft*)) ∈ *RPO-NS*
using *WPO-NS-ctxt* **unfolding** *rpo-eq-wpo* **.**

theorem *RPO-S-ctxt*: (*s*,*t*) ∈ *RPO-S* ⟹ (*Fun f* (*bef* @ *s* # *aft*), *Fun f* (*bef* @ *t*

```
# aft)) ∈ RPO-S
  using WPO-S-ctxt unfolding rpo-eq-wpo by auto

theorem RPO-S-subset-RPO-NS: RPO-S ⊆ RPO-NS
  using WPO-S-subset-WPO-NS unfolding rpo-eq-wpo .

theorem supt-subset-RPO-S: {▷} ⊆ RPO-S
  using supt-subset-WPO-S unfolding rpo-eq-wpo by auto

theorem supteq-subset-RPO-NS: {▷=} ⊆ 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 (∅, UNIV) and the status is the full one, i.e., taking parameters [0,..,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-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)

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 ¬ (∃ *si*∈*set ss*. *lpo-ns si* (*Fun g ts*))
note *IH* = *IH*(*2*−)[*OF this prc*[*symmetric*] *refl*]
from *IH*(*1*) **show** (∀ *j*∈*set* (*σ* (*g*, *length ts*)). *wpo-s* (*Fun f ss*) (*ts* ! *j*)) = (∀ *t*∈*set ts*. *lpo-s* (*Fun f ss*) *t*)
unfolding *set-conv-nth* **by** *auto*
assume *prns* ∧ (∀ *t*∈*set ts*. *lpo-s* (*Fun f ss*) *t*) ¬ *prns*
note *IH* = *IH*(*2*−)[*OF this*]
show *lex-ext wpo n ss ts* = *lex-ext lpo-pr n ss ts*
using *IH* **by** (*intro lex-ext-cong*, *auto*)
qed
qed
qed

abbreviation *LPO-S* ≡ {(*s*,*t*). *lpo-s s t*}
abbreviation *LPO-NS* ≡ {(*s*,*t*). *lpo-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*) ∈ *LPO-S* ⟹ (*s* · *σ*, *t* · *σ*) ∈ *LPO-S* **for** *σ* :: (′*f*,′*a*)*subst*
using *WPO-S-subst* **unfolding** *lpo-eq-wpo* **.**

theorem *LPO-NS-subst*: (*s*,*t*) ∈ *LPO-NS* ⟹ (*s* · *σ*, *t* · *σ*) ∈ *LPO-NS* **for** *σ* :: (′*f*,′*a*)*subst*
using *WPO-NS-subst* **unfolding** *lpo-eq-wpo* **.**

theorem *LPO-NS-ctxt*: (*s*,*t*) ∈ *LPO-NS* ⟹ (*Fun f* (*bef* @ *s* # *aft*), *Fun f* (*bef* @ *t* # *aft*)) ∈ *LPO-NS*
using *WPO-NS-ctxt* **unfolding** *lpo-eq-wpo* **.**

theorem *LPO-S-ctxt*: (*s*,*t*) ∈ *LPO-S* ⟹ (*Fun f* (*bef* @ *s* # *aft*), *Fun f* (*bef* @ *t* # *aft*)) ∈ *LPO-S*
using *WPO-S-ctxt* **unfolding** *lpo-eq-wpo* **by** *auto*

theorem *LPO-S-subset-LPO-NS*: *LPO-S* ⊆ *LPO-NS*
using *WPO-S-subset-WPO-NS* **unfolding** *lpo-eq-wpo* **.**

theorem *supt-subset-LPO-S*: {▷} ⊆ *LPO-S*
using *supt-subset-WPO-S* **unfolding** *lpo-eq-wpo* **by** *auto*

theorem *supteq-subset-LPO-NS*: {⊵} ⊆ *LPO-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 precende *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 precendence 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 ∗: *weight t = w0 t = Fun f ts ts = t1 # ts′*
shows *ts′ = [] w (f,1) = 0*
proof −
have *w0 + sum-list (map weight ts′) ≤ weight t1 + sum-list (map weight ts′)*
by (*rule add-right-mono*, *rule weight-w0*)
also have *... = sum-list (map weight ts)* **unfolding** ∗ **by** *simp*
also have *... ≤ sum-list (map weight (scf-list (scf (f, length ts)) ts))*
by (*rule sum-list-scf-list*, *insert scf*, *auto*)
finally have *w (f,length ts) + w0 + sum-list (map weight ts′) ≤ weight t*
unfolding ∗ **by** *simp*
with ∗(*1*) **have** *sum*: *sum-list (map weight ts′) = 0* **and** *wf*: *w (f,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 × nat*)*set* **where** *lConsts = { (f,0) | f. least f}*
definition *pr-strict′* **where** *pr-strict′ f g = (f ∉ lConsts ∧ (pr-strict f g ∨ g ∈ lConsts))*
definition *pr-weak′* **where** *pr-weak′ f g = ((f ∉ lConsts ∧ pr-weak f g) ∨ g ∈ 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** *adm*[*of f g n*] **unfolding** *pr-weak'-def* **by** (*auto simp*: *lConsts-def*)
subgoal for *f* **using** *least*[*of f*] **unfolding** *pr-weak'-def lConsts-def* **by** *auto*
subgoal by (*rule scf*)
subgoal for *f* **using** *pr-weak-refl*[*of f*] **unfolding** *pr-weak'-def* **by** *auto*
subgoal for *f g h* **using** *pr-weak-trans*[*of f g h*] **unfolding** *pr-weak'-def* **by** *auto*
subgoal for *f g* **using** *pr-strict*[*of f g*] **unfolding** *pr-strict'-def pr-weak'-def* **by** *auto*
proof −
show *SN* {(*x*, *y*). *pr-strict' x y*} (**is** *SN ?R*)
proof
fix *f*
assume ∀ *i*. (*f i*, *f* (*Suc i*)) ∈ *?R*
hence *steps*: ⋀ *i*. (*f i*, *f* (*Suc i*)) ∈ *?R* **by** *blast*
have *f i* ∉ *lConsts* **for** *i* **using** *steps*[*of i*] **unfolding** *pr-strict'-def* **by** *auto*
hence *pr-strict* (*f i*) (*f* (*Suc i*)) **for** *i* **using** *steps*[*of i*] **unfolding** *pr-strict'-def* **by** *auto*
with *pr-SN* **show** *False* **by** *auto*
qed
qed

lemma *least-pr-weak'*: *least f* ⟹ *pr-weak' g* (*f*,*0*) **unfolding** *lConsts-def pr-weak'-def* **by** *auto*

lemma *least-pr-weak'-trans*: *least f* ⟹ *pr-weak'* (*f*,*0*) *g* ⟹ *least* (*fst g*) ∧ *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*: *kbo'.kbo s t* = *kbo s t*
proof (*induct s t rule*: *kbo.induct*)
case (*1 s t*)
note *simps* = *kbo.simps*[*of s t*] *kbo'.kbo.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 ∗: *vars-term-ms* (*SCF t*) ⊆# *vars-term-ms* (*SCF s*) ∧ *weight t* ≤ *weight s*
¬ *weight t* < *weight s*
and *s*: *s* = *Fun f ss*
and *t*: *t* = *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 ⇒ ('f × nat)set ⇒ bool where
  bounded-arity b F = (∀ (f,n) ∈ F. n ≤ b)

lemma finite-funas-term[simp,intro]: finite (funas-term t)
  by (induct t, auto)

context weight-fun begin

definition weight-le s t ≡
  (vars-term-ms (SCF s) ⊆# vars-term-ms (SCF t) ∧ weight s ≤ weight t)

definition weight-less s t ≡
  (vars-term-ms (SCF s) ⊆# vars-term-ms (SCF t) ∧ weight s < weight t)

lemma weight-le-less-iff: weight-le s t ⟹ weight-less s t ⟷ weight s < weight t
  by (auto simp: weight-le-def weight-less-def)
```

lemma *weight-less-iff*: *weight-less s t* $\Longrightarrow$ *weight-le s t* $\wedge$ *weight s* $<$ *weight t*
by (*auto simp*: *weight-le-def weight-less-def*)

abbreviation *weight-NS* $\equiv$ {(*t*,*s*). *weight-le s t*}

abbreviation *weight-S* $\equiv$ {(*t*,*s*). *weight-less s t*}

lemma *weight-le-mono-one*:
assumes *S*: *weight-le s t*
shows *weight-le* (*Fun f* (*ss1* @ *s* # *ss2*)) (*Fun f* (*ss1* @ *t* # *ss2*)) (**is** *weight-le ?s ?t*)
proof −
from *S* **have** *w*: *weight s* $\leq$ *weight t* **and** *v*: *vars-term-ms* (*SCF s*) $\subseteq\#$ *vars-term-ms* (*SCF t*)
by (*auto simp*: *weight-le-def*)
have *v′*: *vars-term-ms* (*SCF ?s*) $\subseteq\#$ *vars-term-ms* (*SCF ?t*)
using *mset-replicate-mono*[*OF v*] **by** *simp*
have *w′*: *weight ?s* $\leq$ *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*: *weight-le s t* $\Longrightarrow$ *weight-le* (*C*⟨*s*⟩) (*C*⟨*t*⟩)
by (*induct C*, *auto intro*: *weight-le-mono-one*)

lemma *SCF-stable*:
assumes *vars-term-ms* (*SCF s*) $\subseteq\#$ *vars-term-ms* (*SCF t*)
shows *vars-term-ms* (*SCF* (*s* · σ)) $\subseteq\#$ *vars-term-ms* (*SCF* (*t* · σ))
unfolding *scf-term-subst*
using *vars-term-ms-subst-mono*[*OF assms*]**.**

lemma *SN-weight-S*: *SN weight-S*
proof−
from *wf-inv-image*[*OF wf-less*]
have *wf*: *wf* {(*s*,*t*). *weight s* $<$ *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*: *weight-less s t* $\Longrightarrow$ *weight-le s t* **by** (*simp add*: *weight-less-def weight-le-def*)

lemma *weight-le-Var-Var*: *weight-le* (*Var x*) (*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*: $\bigwedge f\ g.\ least\ f \Longrightarrow pr\text{-}weak\ g\ (f,0)$
and *least-trans*: $\bigwedge f\ g.\ least\ f \Longrightarrow pr\text{-}weak\ (f,0)\ g \Longrightarrow least\ (fst\ g) \wedge snd\ g = 0$
fixes $n :: nat$
begin

lemma *kbo-instance-of-wpo-with-assms*: *wpo-with-assms*
weight-S weight-NS $(\lambda f\ g.\ (pr\text{-}strict\ f\ g,\ pr\text{-}weak\ f\ g))$
$(\lambda(f,\ n).\ n = 0 \wedge least\ f)$ *full-status False* $(\lambda f.\ False)$
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*)+
apply (*auto simp*: *pr-strict least dest*:*pr-weak-trans*)
done

interpretation *wpo*: *wpo-with-assms*
where $S = weight\text{-}S$ **and** $NS = weight\text{-}NS$
and $prc = \lambda f\ g.\ (pr\text{-}strict\ f\ g,\ pr\text{-}weak\ f\ g)$ **and** $prl = \lambda(f,n).\ n = 0 \wedge least\ f$
and $c = \lambda\text{-}.\ Lex$
and $ssimple = False$ **and** $large = \lambda f.\ False$ **and** $\sigma\sigma = full\text{-}status$
and $n = n$
by (*rule kbo-instance-of-wpo-with-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* $\Longrightarrow$ *bounded-arity n* (*funas-term ti*) $\Longrightarrow$ *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 ¬ *weight-less t* (*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*, *simp add*: *weight-le-def*)

```
    next
      case s: (Fun f ss)
      show ?thesis
      proof (cases t)
        case t: (Var y)
        { assume weight-le t s
          then have ∃ s' ∈ set ss. 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' ∈ 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 ∃ s'' ∈ set ss'. weight-le t s''
              apply (auto simp: t weight-le-def)
              by (metis scf set-scf-list weight-w0)
            then obtain s'' where s'' ∈ 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 ∃ s' ∈ set ss. 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 ∈ set ts by auto
          hence funas-term (ts ! j) ⊆ 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 ¬ weight-le t s ∨ 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) ∈ weight-NS ⟷ True (s, t) ∈ weight-S ⟷ False by auto
  have lex: (Lex = Lex ∧ Lex = Lex) = True by simp
  have sig: set (wpo.σ ?f) = {..<length ss}
    set (wpo.σ ?g) = {..<length ts} by auto
  have map: map ((!) ss) (wpo.σ ?f) = ss
    map ((!) ts) (wpo.σ ?g) = ts
    by (auto simp: map-nth)
  have sizes: i < length ss ⟹ size (ss ! i) < size s for i unfolding s
    by (simp add: size-simp1)
  have sizet: i < length ts ⟹ size (ts ! i) < size t for i unfolding t
    by (simp add: size-simp1)
  have wpo: wpo.wpo s t =
     (if ∃ i∈{..<length ss}. wpo.wpo-ns (ss ! i) t then (True, True)
      else if pr-weak ?f ?g ∧ (∀ j∈{..<length ts}. wpo.wpo-s s (ts ! j))
      then if pr-strict ?f ?g then (True, True) else lex-ext wpo.wpo n ss ts
      else (False, 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 (True, True)
       else if pr-weak ?f ?g then lex-ext-unbounded kbo ss ts
       else (False, 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 ... = lex-ext wpo.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 (True, True)
      else if pr-weak ?f ?g then lex-ext wpo.wpo n ss ts
      else (False, False)) .
  show ?thesis
  proof (cases ∃ i∈{..<length ss}. wpo.wpo-ns (ss ! i) t)
    case True
   then obtain i where i: i < length ss and wpo.wpo-ns (ss ! i) t by auto
    then obtain b where wpo.wpo (ss ! i) t = (b, True) by (cases wpo.wpo
(ss ! i) t, auto)
  also have wpo.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 (Fun f (take i ss @ ss ! i # drop (Suc i) ss)) t .
    also have (take i ss @ ss ! i # drop (Suc i) ss) = ss using i by (metis
id-take-nth-drop)
```

```
          also have Fun f ss = s unfolding s by simp
          finally have S s t .
          with S-imp-NS[OF this]
          have kbo s t = (True,True) by (cases kbo s t, auto)
          with True show ?thesis unfolding wpo by auto
        next
          case False
          hence False: (∃ i∈{..<length ss}. wpo.wpo-ns (ss ! i) t) = False by simp
          {
            fix j
            assume NS: NS s t
            assume j: j < length ts

            from kbo-supt-one[OF NS-refl, of g take j ts ts ! j drop (Suc j) ts]
            have S: S t (ts ! j) using id-take-nth-drop[OF j] unfolding t by auto
            from kbo-trans[of s t ts ! j] NS S have S s (ts ! j) by auto
            with S S-imp-NS[OF this]
            have kbo s (ts ! j) = (True, True) by (cases kbo s (ts ! j), auto)
            hence wpo.wpo-s s (ts ! j)
              by (subst IH-tj[symmetric], insert sizet[OF j] j, auto)
          }
        thus ?thesis unfolding wpo kbo False if-False using lex-ext-stri-imp-nstri[of
wpo.wpo n ss ts]
            by (cases lex-ext wpo.wpo n ss ts, auto simp: pr-strict split: 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 prec ≡ ((λf g. (pr-strict' f g, pr-weak' f g)))
  and prl ≡ (λ(f, n). n = 0 ∧ least f)
  shows
   kbo-encoding-is-valid-wpo: wpo-with-assms weight-S weight-NS prec prl full-status
False (λf. False)
  and
    kbo-as-wpo: bounded-arity n (funas-term t) ⟹ kbo s t = wpo.wpo n weight-S
weight-NS prec prl full-status (λ-. Lex) False (λf. False) s t
  unfolding prec-def prl-def
   subgoal by (intro admissible-kbo.kbo-instance-of-wpo-with-assms[OF admissi-
ble-kbo']
       least-pr-weak' least-pr-weak'-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*)[*3*])
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 $t_1 > t_2 > t_3 > ...$ 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*: $S\ s\ t \Longrightarrow S\ (s \cdot (\sigma :: ('f,'a)\ subst))\ (t \cdot \sigma)$
proof −
let *?terms* = $\{t,\ t \cdot \sigma\}$
let *?prec* = $((\lambda f\ g.\ (pr\text{-}strict'\ f\ g,\ pr\text{-}weak'\ f\ g)))$
let *?prl* = $(\lambda(f,\ n).\ n = 0 \wedge least\ f)$
have *finite* $(\bigcup\ (funas\text{-}term\ `\ ?terms))$
by *auto*
from *finite-list*[*OF this*] **obtain** *F* **where** *F*: $set\ F = \bigcup\ (funas\text{-}term\ `\ ?terms)$
by *auto*

define n **where** $n = max\text{-}list\ (map\ snd\ F)$

interpret *wpo*: *wpo-with-assms*
where $S = weight\text{-}S$ **and** $NS = weight\text{-}NS$
and $prc = ?prec$ **and** $prl = ?prl$
and $c = \lambda\text{-}.\ Lex$
and $ssimple = False$ **and** $large = \lambda f.\ False$ **and** $\sigma\sigma = full\text{-}status$
and $n = n$
by (*rule kbo-encoding-is-valid-wpo*)

{
fix t
assume $t \in ?terms$
hence $funas\text{-}term\ t \subseteq set\ F$ **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 $S\ s\ t \Longrightarrow S\ (s \cdot \sigma)\ (t \cdot \sigma)$
using *kbo-as-wpo* **by** *auto*
qed

lemma *weight-is-arity-bound*: *weight t* $\leq$ *b* $\Longrightarrow$ *bounded-arity b* (*funas-term t*)
proof (*induct t*)
case (*Fun f ts*)
have *sum-list* (*map weight ts*) $\leq$ *weight* (*Fun f ts*)
using *sum-list-scf-list*[*of ts scf* (*f*,*length ts*), *OF scf*] **by** *auto*
also have ... $\leq$ *b* **using** *Fun* **by** *auto*
finally have *sum-b*: *sum-list* (*map weight ts*) $\leq$ *b* **.**
{
fix *t*
assume *t*: *t* $\in$ *set ts*
from *split-list*[*OF this*] **have** *weight t* $\leq$ *sum-list* (*map weight ts*) **by** *auto*
with *sum-b* **have** *bounded-arity b* (*funas-term t*) **using** *t Fun* **by** *auto*
} note *IH* = *this*
have *length ts* = *sum-list* (*map* (λ -. *1*) *ts*) **by** (*induct ts*, *auto*)
also have ... $\leq$ *sum-list* (*map weight ts*)
apply (*rule sum-list-mono*)
subgoal for *t* **using** *weight-gt-0*[*of t*] **by** *auto*
done
also have ... $\leq$ *b* **by** *fact*
finally have *len*: *length ts* $\leq$ *b* **by** *auto*
from *IH len* **show** *?case* **unfolding** *bounded-arity-def* **by** *auto*
qed (*auto simp*: *bounded-arity-def*)

lemma *KBO-SN-via-WPO*: *SN* {(*s*,*t*). *S s t*}
proof
fix *f* :: *nat* $\Rightarrow$ (*'f*,*'a*)*term*
assume $\forall$ *i*. (*f i*, *f* (*Suc i*)) $\in$ {(*s*, *t*). *S s t*}
hence *steps*: *S* (*f i*) (*f* (*Suc i*)) **for** *i* **by** *auto*
define *n* **where** *n* = *weight* (*f 0*)

have *w-bound*: *weight* (*f i*) $\leq$ *n* **for** *i*
proof (*induct i*)
case (*Suc i*)
from *steps*[*of i*] **have** *weight* (*f* (*Suc i*)) $\leq$ *weight* (*f i*)
unfolding *kbo.simps*[*of f i*] **by** (*auto split*: *if-splits*)
with *Suc* **show** *?case* **by** *simp*
qed (*auto simp*: *n-def*)

let *?prec* = ((λ*f g*. (*pr-strict' f g*, *pr-weak' f g*)))
let *?prl* = (λ(*f*, *n*). *n* = *0* $\wedge$ *least f*)

interpret *wpo*: *wpo-with-assms*
where *S* = *weight-S* **and** *NS* = *weight-NS*
and *prc* = *?prec* **and** *prl* = *?prl*
and *c* = λ-. *Lex*
and *ssimple* = *False* **and** *large* = λ*f*. *False* **and** $\sigma\sigma$ = *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, 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.