

A Formalization of Knuth–Bendix Orders*

Christian Sternagel and René Thiemann

April 14, 2026

Abstract

We define a generalized version of Knuth–Bendix orders, including subterm coefficient functions. For these orders we formalize several properties such as strong normalization, the subterm property, closure properties under substitutions and contexts, as well as ground totality.

Contents

1	Introduction	2
2	Order Pairs	2
3	Lexicographic Extension	5
4	KBO	27
4.1	Subterm Coefficient Functions	27
4.2	Weight Functions	29
4.3	Definition of KBO	30
4.4	Reflexivity and Irreflexivity	34
4.5	Monotonicity (a.k.a. Closure under Contexts)	34
4.6	The Subterm Property	36
4.7	Least Elements	39
4.8	Stability (a.k.a. Closure under Substitutions)	42
4.9	Transitivity and Compatibility	45
4.10	Strong Normalization (a.k.a. Well-Foundedness)	49
4.11	Ground Totality	52
4.12	Summary	53

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

1 Introduction

In their seminal paper [2], Knuth and Bendix introduced two important concepts: a procedure that allows us to solve certain instances of the word problem – (Knuth–Bendix) completion – as well as a specific order on terms that is useful to orient equations in the aforementioned procedure – the Knuth–Bendix order (or KBO, for short).

This AFP-entry is about the formalization of KBO. Note that there are several variants of KBO [2, 1, 3, 7, 4], e.g., incorporating quasi-precedences, infinite signatures, subterm coefficient functions, and generalized weight functions. In fact, not for all of these variants well-foundedness has been proven. We give the first well-foundedness proof for a variant of KBO that combines infinite signatures, quasi-precedences, and subterm coefficient functions. Our proof is direct, i.e., it does not depend on Kruskal’s tree theorem.

This formalization is used in the `IsaFoR/CeTAproject` [6] for certifying untrusted termination and confluence proofs. For more details we refer to our RTA paper [5].

2 Order Pairs

An order pair consists of two relations S and NS , where S is a strict order and NS a compatible non-strict order, such that the combination of S and NS always results in strict decrease.

theory *Order-Pair*

imports *Abstract–Rewriting.Relative-Rewriting*

begin

named-theorems *order-simps*

declare *O-assoc*[*order-simps*]

locale *pre-order-pair* =

fixes $S :: 'a\ rel$

and $NS :: 'a\ rel$

assumes *refl-NS*: *refl NS*

and *trans-S*: *trans S*

and *trans-NS*: *trans NS*

begin

lemma *refl-NS-point*: $(s, s) \in NS$ **using** *refl-NS* **unfolding** *refl-on-def* **by** *blast*

lemma *NS-O-NS*[*order-simps*]: $NS\ O\ NS = NS\ NS\ O\ NS\ O\ T = NS\ O\ T$

proof –

show $NS\ O\ NS = NS$ **by**(*fact trans-refl-imp-O-id*[*OF trans-NS refl-NS*])

then show $NS\ O\ NS\ O\ T = NS\ O\ T$ **by** *fast*

qed

```

lemma trancl-NS[order-simps]:  $NS^+ = NS$  using trans-NS by simp

lemma rtrancl-NS[order-simps]:  $NS^* = NS$ 
  by (rule trans-refl-imp-rtrancl-id[OF trans-NS refl-NS])

lemma trancl-S[order-simps]:  $S^+ = S$  using trans-S by simp

lemma S-O-S:  $S O S \subseteq S S O S O T \subseteq S O T$ 
proof –
  show  $S O S \subseteq S$  by (fact trans-O-subset[OF trans-S])
  then show  $S O S O T \subseteq S O T$  by blast
qed

lemma trans-S-point:  $\bigwedge x y z. (x, y) \in S \implies (y, z) \in S \implies (x, z) \in S$ 
  using trans-S unfolding trans-def by blast

lemma trans-NS-point:  $\bigwedge x y z. (x, y) \in NS \implies (y, z) \in NS \implies (x, z) \in NS$ 
  using trans-NS unfolding trans-def by blast
end

locale compat-pair =
  fixes  $S NS :: 'a \text{ rel}$ 
  assumes compat-NS-S:  $NS O S \subseteq S$ 
  and compat-S-NS:  $S O NS \subseteq S$ 
begin
lemma compat-NS-S-point:  $\bigwedge x y z. (x, y) \in NS \implies (y, z) \in S \implies (x, z) \in S$ 
  using compat-NS-S by blast

lemma compat-S-NS-point:  $\bigwedge x y z. (x, y) \in S \implies (y, z) \in NS \implies (x, z) \in S$ 
  using compat-S-NS by blast

lemma S-O-rtrancl-NS[order-simps]:  $S O NS^* = S S O NS^* O T = S O T$ 
proof –
  show  $S O NS^* = S$ 
  proof(intro equalityI subrelI)
    fix  $x y$  assume  $(x, y) \in S O NS^*$ 
    then obtain  $n$  where  $(x, y) \in S O NS^{\sim n}$  by blast
    then show  $(x, y) \in S$ 
    proof(induct n arbitrary: y)
      case 0 then show ?case by auto
    next
      case IH: (Suc n)
        then obtain  $z$  where  $xz: (x, z) \in S O NS^{\sim n}$  and  $zy: (z, y) \in NS$  by auto
        from IH.hyps[OF xz] zy have  $(x, y) \in S O NS$  by auto
        with compat-S-NS show ?case by auto
    qed
  qed auto
  then show  $S O NS^* O T = S O T$  by auto

```

qed

lemma *rtrancl-NS-O-S[order-simps]*: $NS^* O S = S NS^* O S O T = S O T$

proof –

show $NS^* O S = S$

proof (*intro equalityI subrelI*)

fix $x y$ **assume** $(x, y) \in NS^* O S$

then obtain n **where** $(x, y) \in NS^{\sim n} O S$ **by** *blast*

then show $(x, y) \in S$

proof (*induct n arbitrary: x*)

case 0 **then show** *?case* **by** *auto*

next

case *IH: (Suc n)*

then obtain z **where** $xz: (x, z) \in NS$ **and** $zy: (z, y) \in NS^{\sim n} O S$ **by**
(*unfold relpow-Suc, auto*)

from xz *IH.hyps[OF zy]* **have** $(x, y) \in NS O S$ **by** *auto*

with *compat-NS-S* **show** *?case* **by** *auto*

qed

qed *auto*

then show $NS^* O S O T = S O T$ **by** *auto*

qed

end

locale *order-pair* = *pre-order-pair S NS* + *compat-pair S NS*

for $S NS :: 'a rel$

begin

lemma *S-O-NS[order-simps]*: $S O NS = S S O NS O T = S O T$ **by** (*fact*
S-O-rtrancl-NS[unfolded rtrancl-NS])**+**

lemma *NS-O-S[order-simps]*: $NS O S = S NS O S O T = S O T$ **by** (*fact*
rtrancl-NS-O-S[unfolded rtrancl-NS])**+**

lemma *compat-rtrancl*:

assumes $ab: (a, b) \in S$

and $bc: (b, c) \in (NS \cup S)^*$

shows $(a, c) \in S$

using bc

proof (*induct*)

case *base*

show *?case* **by** (*rule ab*)

next

case (*step c d*)

from *step(2-3)* **show** *?case* **using** *compat-S-NS-point trans-S unfolding trans-def*
by *blast*

qed

end

```

locale SN-ars =
  fixes S :: 'a rel
  assumes SN: SN S

locale SN-pair = compat-pair S NS + SN-ars S for S NS :: 'a rel

locale SN-order-pair = order-pair S NS + SN-ars S for S NS :: 'a rel

sublocale SN-order-pair  $\subseteq$  SN-pair ..

end

```

3 Lexicographic Extension

theory *Lexicographic-Extension*

imports

Matrix.Utility

Order-Pair

begin

In this theory we define the lexicographic extension of an order pair, so that it generalizes the existing notion ($\langle *lex* \rangle$) which is based on a single order only.

Our main result is that this extension yields again an order pair.

fun *lex-two* :: 'a rel \Rightarrow 'a rel \Rightarrow 'b rel \Rightarrow ('a \times 'b) rel

where

lex-two s ns s2 = $\{((a1, b1), (a2, b2)) . (a1, a2) \in s \vee (a1, a2) \in ns \wedge (b1, b2) \in s2\}$

lemma *lex-two*:

assumes *compat*: *ns O s* \subseteq *s*

and *SN-s*: *SN s*

and *SN-s2*: *SN s2*

shows *SN (lex-two s ns s2)* (**is** *SN ?r*)

proof

fix *f*

assume $\forall i. (f i, f (Suc i)) \in ?r$

then have *steps*: $\bigwedge i. (f i, f (Suc i)) \in ?r$..

let *?a* = $\lambda i. fst (f i)$

let *?b* = $\lambda i. snd (f i)$

{

fix *i*

from *steps*[*of i*]

have $(?a i, ?a (Suc i)) \in s \vee (?a i, ?a (Suc i)) \in ns \wedge (?b i, ?b (Suc i)) \in s2$

by (*cases f i*, *cases f (Suc i)*, *auto*)

}

note *steps* = *this*

have $\exists j. \forall i \geq j. (?a i, ?a (Suc i)) \in ns - s$

by (rule non-strict-ending[OF - compat], insert steps SN-s, unfold SN-on-def, auto)
 with steps obtain j where steps: $\bigwedge i. i \geq j \implies (?b\ i, ?b\ (Suc\ i)) \in s2$ by auto
 obtain g where g: $g = (\lambda i. ?b\ (j + i))$ by auto
 from steps have $\bigwedge i. (g\ i, g\ (Suc\ i)) \in s2$ unfolding g by auto
 with SN-s2 show False unfolding SN-defs by auto
 qed

lemma lex-two-compat:

assumes compat1: $ns1\ O\ s1 \subseteq s1$
 and compat1': $s1\ O\ ns1 \subseteq s1$
 and trans1: $s1\ O\ s1 \subseteq s1$
 and trans1': $ns1\ O\ ns1 \subseteq ns1$
 and compat2: $ns2\ O\ s2 \subseteq s2$
 and ns: $(ab1, ab2) \in lex-two\ s1\ ns1\ ns2$
 and s: $(ab2, ab3) \in lex-two\ s1\ ns1\ s2$
 shows $(ab1, ab3) \in lex-two\ s1\ ns1\ s2$
 proof -
 obtain a1 b1 where ab1: $ab1 = (a1, b1)$ by force
 obtain a2 b2 where ab2: $ab2 = (a2, b2)$ by force
 obtain a3 b3 where ab3: $ab3 = (a3, b3)$ by force
 note id = ab1 ab2 ab3
 show ?thesis
 proof (cases $(a1, a2) \in s1$)
 case s1: True
 show ?thesis
 proof (cases $(a2, a3) \in s1$)
 case s2: True
 from trans1 s1 s2 show ?thesis unfolding id by auto
 next
 case False with s have $(a2, a3) \in ns1$ unfolding id by simp
 from compat1' s1 this show ?thesis unfolding id by auto
 qed
 next
 case False
 with ns have ns: $(a1, a2) \in ns1\ (b1, b2) \in ns2$ unfolding id by auto
 show ?thesis
 proof (cases $(a2, a3) \in s1$)
 case s2: True
 from compat1 ns(1) s2 show ?thesis unfolding id by auto
 next
 case False
 with s have nss: $(a2, a3) \in ns1\ (b2, b3) \in s2$ unfolding id by auto
 from trans1' ns(1) nss(1) compat2 ns(2) nss(2)
 show ?thesis unfolding id by auto
 qed
 qed
 qed
 qed

```

lemma lex-two-compat':
  assumes compat1:  $ns1 \ O \ s1 \subseteq s1$ 
    and compat1':  $s1 \ O \ ns1 \subseteq s1$ 
    and trans1:  $s1 \ O \ s1 \subseteq s1$ 
    and trans1':  $ns1 \ O \ ns1 \subseteq ns1$ 
    and compat2':  $s2 \ O \ ns2 \subseteq s2$ 
    and s:  $(ab1, ab2) \in \text{lex-two } s1 \ ns1 \ s2$ 
    and ns:  $(ab2, ab3) \in \text{lex-two } s1 \ ns1 \ ns2$ 
  shows  $(ab1, ab3) \in \text{lex-two } s1 \ ns1 \ s2$ 
proof –
  obtain a1 b1 where ab1:  $ab1 = (a1, b1)$  by force
  obtain a2 b2 where ab2:  $ab2 = (a2, b2)$  by force
  obtain a3 b3 where ab3:  $ab3 = (a3, b3)$  by force
  note id = ab1 ab2 ab3
  show ?thesis
  proof (cases  $(a1, a2) \in s1$ )
    case s1: True
      show ?thesis
      proof (cases  $(a2, a3) \in s1$ )
        case s2: True
          from trans1 s1 s2 show ?thesis unfolding id by auto
        next
          case False with ns have  $(a2, a3) \in ns1$  unfolding id by simp
          from compat1' s1 this show ?thesis unfolding id by auto
        qed
      next
        case False
          with s have s:  $(a1, a2) \in ns1 \ (b1, b2) \in s2$  unfolding id by auto
          show ?thesis
          proof (cases  $(a2, a3) \in s1$ )
            case s2: True
              from compat1 s(1) s2 show ?thesis unfolding id by auto
            next
              case False
                with ns have nss:  $(a2, a3) \in ns1 \ (b2, b3) \in ns2$  unfolding id by auto
                from trans1' s(1) nss(1) compat2' s(2) nss(2)
                show ?thesis unfolding id by auto
              qed
            qed
          qed
        qed
      qed
    qed
  qed

lemma lex-two-compat2:
  assumes  $ns1 \ O \ s1 \subseteq s1 \ s1 \ O \ ns1 \subseteq s1 \ s1 \ O \ s1 \subseteq s1 \ ns1 \ O \ ns1 \subseteq ns1 \ ns2 \ O \ s2 \subseteq s2$ 
  shows  $\text{lex-two } s1 \ ns1 \ ns2 \ O \ \text{lex-two } s1 \ ns1 \ s2 \subseteq \text{lex-two } s1 \ ns1 \ s2$ 
  using lex-two-compat[OF assms] by (intro subsetI, elim relcompE, fast)

lemma lex-two-compat'2:
  assumes  $ns1 \ O \ s1 \subseteq s1 \ s1 \ O \ ns1 \subseteq s1 \ s1 \ O \ s1 \subseteq s1 \ ns1 \ O \ ns1 \subseteq ns1 \ s2 \ O \ ns2$ 

```

$\subseteq s2$
shows *lex-two s1 ns1 s2 O lex-two s1 ns1 ns2* \subseteq *lex-two s1 ns1 s2*
using *lex-two-compat'[OF assms]* **by** (*intro subsetI, elim relcompE, fast*)

lemma *lex-two-refl*:
assumes *r1: refl ns1 and r2: refl ns2*
shows *refl (lex-two s1 ns1 ns2)*
using *refl-onD[OF r1] and refl-onD[OF r2]* **by** (*intro refl-onI auto*)

lemma *lex-two-order-pair*:
assumes *o1: order-pair s1 ns1 and o2: order-pair s2 ns2*
shows *order-pair (lex-two s1 ns1 s2) (lex-two s1 ns1 ns2)*
proof –
interpret *o1: order-pair s1 ns1 using o1.*
interpret *o2: order-pair s2 ns2 using o2.*
note *o1.trans-S o1.trans-NS o2.trans-S o2.trans-NS*
o1.compat-NS-S o2.compat-NS-S o1.compat-S-NS o2.compat-S-NS
note *this [unfolded trans-O-iff]*
note *o1.refl-NS o2.refl-NS*
show *?thesis*
by (*unfold-locales, intro lex-two-refl, fact+, unfold trans-O-iff*)
(rule lex-two-compat2 lex-two-compat'2;fact)+
qed

lemma *lex-two-SN-order-pair*:
assumes *o1: SN-order-pair s1 ns1 and o2: SN-order-pair s2 ns2*
shows *SN-order-pair (lex-two s1 ns1 s2) (lex-two s1 ns1 ns2)*
proof –
interpret *o1: SN-order-pair s1 ns1 using o1.*
interpret *o2: SN-order-pair s2 ns2 using o2.*
note *o1.trans-S o1.trans-NS o2.trans-S o2.trans-NS o1.SN o2.SN*
o1.compat-NS-S o2.compat-NS-S o1.compat-S-NS o2.compat-S-NS
note *this [unfolded trans-O-iff]*
interpret *order-pair (lex-two s1 ns1 s2) (lex-two s1 ns1 ns2)*
by(*rule lex-two-order-pair, standard*)
show *?thesis by(standard, rule lex-two; fact)*
qed

In the unbounded lexicographic extension, there is no restriction on the lengths of the lists. Therefore it is possible to compare lists of different lengths. This usually results a non-terminating relation, e.g., $[1] > [0, 1] > [0, 0, 1] > \dots$

fun *lex-ext-unbounded* :: (*'a* \Rightarrow *'a* \Rightarrow *bool* \times *bool*) \Rightarrow *'a list* \Rightarrow *'a list* \Rightarrow *bool* \times *bool*
where *lex-ext-unbounded f [] [] = (False, True) |*
lex-ext-unbounded f (- # -) [] = (True, True) |
lex-ext-unbounded f [] (- # -) = (False, False) |
lex-ext-unbounded f (a # as) (b # bs) =
(let (stri, nstri) = f a b in
if stri then (True, True)

else if *nstri* then *lex-ext-unbounded f as bs*
 else (*False, False*)

lemma *lex-ext-unbounded-iff*: (*lex-ext-unbounded f xs ys*) = (
 (($\exists i < \text{length } xs. i < \text{length } ys \wedge (\forall j < i. \text{snd } (f (xs ! j) (ys ! j))) \wedge \text{fst } (f (xs ! i) (ys ! i))$)) \vee
 ($\forall i < \text{length } ys. \text{snd } (f (xs ! i) (ys ! i)) \wedge \text{length } xs > \text{length } ys$),
 (($\exists i < \text{length } xs. i < \text{length } ys \wedge (\forall j < i. \text{snd } (f (xs ! j) (ys ! j))) \wedge \text{fst } (f (xs ! i) (ys ! i))$)) \vee
 ($\forall i < \text{length } ys. \text{snd } (f (xs ! i) (ys ! i)) \wedge \text{length } xs \geq \text{length } ys$))
 (**is** *?lex xs ys = (?stri xs ys, ?nstri xs ys)*)
proof (*induct xs arbitrary: ys*)
 case *Nil* then show *?case* by (*cases ys, auto*)
next
 case (*Cons a as*)
note *oCons = this*
from *oCons* show *?case*
proof (*cases ys, simp*)
 case (*Cons b bs*)
 show *?thesis*
proof (*cases f a b*)
 case (*Pair stri nstri*)
 show *?thesis*
proof (*cases stri*)
 case *True*
 with *Pair Cons* show *?thesis* by *auto*
next
 case *False*
 show *?thesis*
proof (*cases nstri*)
 case *False*
 with $\langle \neg \text{stri} \rangle$ *Pair Cons* show *?thesis* by *force*
next
 case *True*
 with *False Pair* have *f: f a b = (False, True)* by *auto*
 show *?thesis* by (*simp add: all-Suc-conv ex-Suc-conv Cons f oCons*)
 qed
 qed
 qed
 qed
 qed

declare *lex-ext-unbounded.simps[simp del]*

The lexicographic extension of an order pair takes a natural number as maximum bound. A decrease with lists of unequal lengths will never be successful if the length of the second list exceeds this bound. The bound is essential to preserve strong normalization.

definition *lex-ext* :: (*'a* \Rightarrow *'a* \Rightarrow *bool* \times *bool*) \Rightarrow *nat* \Rightarrow *'a list* \Rightarrow *'a list* \Rightarrow *bool* \times

bool

where

lex-ext f n ss ts =
 (*let lts = length ts in*
 if (length ss = lts \vee lts \leq n) then lex-ext-unbounded f ss ts
 else (False, False))

lemma *lex-ext-iff*: (*lex-ext f m xs ys =* (
 (*length xs = length ys \vee length ys \leq m*) \wedge (\exists *i < length xs. i < length ys \wedge (\forall
 j < i. snd (f (xs ! j) (ys ! j))) \wedge *fst (f (xs ! i) (ys ! i))*) \vee
 (\forall *i < length ys. snd (f (xs ! i) (ys ! i))*) \wedge *length xs > length ys*),
 (*length xs = length ys \vee length ys \leq m*) \wedge
 (\exists *i < length xs. i < length ys \wedge (\forall *j < i. snd (f (xs ! j) (ys ! j))*) \wedge *fst (f (xs*
 !*i) (ys !i))*) \vee
 (\forall *i < length ys. snd (f (xs ! i) (ys ! i))*) \wedge *length xs \geq length ys*))**

unfolding *lex-ext-def*

by (*simp only: lex-ext-unbounded-iff Let-def, auto*)

lemma *lex-ext-to-lex-ext-unbounded*:

assumes *length xs \leq n and length ys \leq n*
shows *lex-ext f n xs ys = lex-ext-unbounded f xs ys*
using *assms by (simp add: lex-ext-def)*

lemma *lex-ext-stri-imp-nstri*:

assumes *fst (lex-ext f m xs ys)*
shows *snd (lex-ext f m xs ys)*
using *assms by (auto simp: lex-ext-iff)*

lemma *nstri-lex-ext-map*:

assumes $\bigwedge s t. s \in \text{set } ss \implies t \in \text{set } ts \implies \text{fst } (\text{order } s t) \implies \text{fst } (\text{order}' (f s)$
(f t))
 and $\bigwedge s t. s \in \text{set } ss \implies t \in \text{set } ts \implies \text{snd } (\text{order } s t) \implies \text{snd } (\text{order}' (f s) (f$
t))
 and *snd (lex-ext order n ss ts)*
shows *snd (lex-ext order' n (map f ss) (map f ts))*
using *assms unfolding lex-ext-iff by auto*

lemma *stri-lex-ext-map*:

assumes $\bigwedge s t. s \in \text{set } ss \implies t \in \text{set } ts \implies \text{fst } (\text{order } s t) \implies \text{fst } (\text{order}' (f s)$
(f t))
 and $\bigwedge s t. s \in \text{set } ss \implies t \in \text{set } ts \implies \text{snd } (\text{order } s t) \implies \text{snd } (\text{order}' (f s) (f$
t))
 and *fst (lex-ext order n ss ts)*
shows *fst (lex-ext order' n (map f ss) (map f ts))*
using *assms unfolding lex-ext-iff by auto*

lemma *lex-ext-arg-empty*: *snd (lex-ext f n [] xs) \implies xs = []*

unfolding *lex-ext-iff* **by** *auto*

lemma *lex-ext-co-compat*:

assumes $\bigwedge s t. s \in \text{set } ss \implies t \in \text{set } ts \implies \text{fst } (\text{order } s t) \implies \text{snd } (\text{order}' t s)$
 $\implies \text{False}$

and $\bigwedge s t. s \in \text{set } ss \implies t \in \text{set } ts \implies \text{snd } (\text{order } s t) \implies \text{fst } (\text{order}' t s)$
 $\implies \text{False}$

and $\bigwedge s t. \text{fst } (\text{order } s t) \implies \text{snd } (\text{order } s t)$

and $\text{fst } (\text{lex-ext } \text{order } n ss ts)$

and $\text{snd } (\text{lex-ext } \text{order}' n ts ss)$

shows *False*

proof –

let $?ls = \text{length } ss$

let $?lt = \text{length } ts$

define s **where** $s i = \text{fst } (\text{order } (ss ! i) (ts ! i))$ **for** i

define ns **where** $ns i = \text{snd } (\text{order } (ss ! i) (ts ! i))$ **for** i

define s' **where** $s' i = \text{fst } (\text{order}' (ts ! i) (ss ! i))$ **for** i

define ns' **where** $ns' i = \text{snd } (\text{order}' (ts ! i) (ss ! i))$ **for** i

have $co: i < ?ls \implies i < ?lt \implies s i \implies ns' i \implies \text{False}$ **for** i

using $\text{assms}(1)$ **unfolding** $s\text{-def } ns'\text{-def } \text{set-conv-nth}$ **by** *auto*

have $co': i < ?ls \implies i < ?lt \implies s' i \implies ns i \implies \text{False}$ **for** i

using $\text{assms}(2)$ **unfolding** $s'\text{-def } ns\text{-def } \text{set-conv-nth}$ **by** *auto*

from $\text{assms}(4)$ [*unfolded lex-ext-iff fst-conv, folded s-def ns-def*]

have $ch1: (\exists i. i < ?ls \wedge i < ?lt \wedge (\forall j < i. ns j) \wedge s i) \vee (\forall i < ?lt. ns i) \wedge ?lt < ?ls$ **(is ?A \vee ?B)** **by** *auto*

from $\text{assms}(5)$ [*unfolded lex-ext-iff snd-conv, folded s'-def ns'-def*]

have $ch2: (\exists i. i < ?ls \wedge i < ?lt \wedge (\forall j < i. ns' j) \wedge s' i) \vee (\forall i < ?ls. ns' i) \wedge ?ls \leq ?lt$ **(is ?A' \vee ?B')** **by** *auto*

from $ch1$ **show** *False*

proof

assume $?A$

then obtain i **where** $i: i < ?ls \wedge i < ?lt$ **and** $s: s i$ **and** $ns: \bigwedge j. j < i \implies ns j$ **by** *auto*

note $s = co[OF i s]$

have $ns: j < i \implies s' j \implies \text{False}$ **for** j

using $i ns[of j] co'[of j]$ **by** *auto*

from $ch2$ **show** *False*

proof

assume $?A'$

then obtain i' **where** $i': i' < ?ls \wedge i' < ?lt$ **and** $s': s' i'$ **and** $ns': \bigwedge j'. j' < i' \implies ns' j'$ **by** *auto*

from $s ns'[of i']$ **have** $i \geq i'$ **by** *presburger*

with $ns[OF - s']$ **have** $i': i' = i$ **by** *presburger*

from $\langle s i \rangle$ **have** $ns i$ **using** $\text{assms}(3)$ **unfolding** $s\text{-def } ns\text{-def}$ **by** *auto*

from $co'[OF i s'[unfolding i']]$ **show** *False* .

next

assume $?B'$

with i **have** $ns' i$ **by** *auto*

from $s[OF this]$ **show** *False* .

```

qed
next
  assume B: ?B
  with ch2 have ?A' by auto
  then obtain i where i: i < ?ls i < ?lt and s': s' i and ns':  $\bigwedge j. j < i \implies$ 
ns' j by auto
  from co'[OF i s'] B i show False by auto
qed
qed

```

```

lemma lex-ext-irrefl: assumes  $\bigwedge x. x \in \text{set } xs \implies \neg \text{fst } (\text{rel } x x)$ 
  shows  $\neg \text{fst } (\text{lex-ext rel } n \text{ } xs \text{ } xs)$ 
proof
  assume fst (lex-ext rel n xs xs)
  then obtain i where i < length xs and fst (rel (xs ! i) (xs ! i))
    unfolding lex-ext-iff by auto
  with assms[of xs ! i] show False by auto
qed

```

```

lemma lex-ext-unbounded-stri-imp-nstri:
  assumes fst (lex-ext-unbounded f xs ys)
  shows snd (lex-ext-unbounded f xs ys)
  using assms by (auto simp: lex-ext-unbounded-iff)

```

```

lemma all-nstri-imp-lex-nstri: assumes  $\forall i < \text{length } ys. \text{snd } (f (xs ! i) (ys ! i))$ 
  and  $\text{length } xs \geq \text{length } ys$  and  $\text{length } xs = \text{length } ys \vee \text{length } ys \leq m$ 
  shows  $\text{snd } (\text{lex-ext } f \text{ } m \text{ } xs \text{ } ys)$ 
  using assms by (auto simp: lex-ext-iff)

```

```

lemma lex-ext-cong[fundef-cong]: fixes f g m1 m2 xs1 xs2 ys1 ys2
  assumes  $\text{length } xs1 = \text{length } ys1$  and  $m1 = m2$  and  $\text{length } xs2 = \text{length } ys2$ 
  and  $\bigwedge i. \llbracket i < \text{length } ys1; i < \text{length } ys2 \rrbracket \implies f (xs1 ! i) (xs2 ! i) = g (ys1 ! i) (ys2 ! i)$ 
  shows  $\text{lex-ext } f \text{ } m1 \text{ } xs1 \text{ } xs2 = \text{lex-ext } g \text{ } m2 \text{ } ys1 \text{ } ys2$ 
  using assms by (auto simp: lex-ext-iff)

```

```

lemma lex-ext-unbounded-cong[fundef-cong]: assumes  $as = as'$  and  $bs = bs'$ 
  and  $\bigwedge i. i < \text{length } as' \implies i < \text{length } bs' \implies f (as' ! i) (bs' ! i) = g (as' ! i) (bs' ! i)$ 
  shows  $\text{lex-ext-unbounded } f \text{ } as \text{ } bs = \text{lex-ext-unbounded } g \text{ } as' \text{ } bs'$ 
  unfolding assms lex-ext-unbounded-iff using assms(3) by auto

```

Compatibility is the key property to ensure transitivity of the order.

We prove compatibility locally, i.e., it only has to hold for elements of the argument lists. Locality is essential for being applicable in recursively defined term orders such as KBO.

```

lemma lex-ext-compat:
  assumes compat:  $\bigwedge s \text{ } t \text{ } u. \llbracket s \in \text{set } ss; t \in \text{set } ts; u \in \text{set } us \rrbracket \implies$ 
  ( $\text{snd } (f \text{ } s \text{ } t) \wedge \text{fst } (f \text{ } t \text{ } u) \longrightarrow \text{fst } (f \text{ } s \text{ } u) \wedge$ 

```

$$\begin{aligned} & (fst (f s t) \wedge snd (f t u) \longrightarrow fst (f s u)) \wedge \\ & (snd (f s t) \wedge snd (f t u) \longrightarrow snd (f s u)) \wedge \\ & (fst (f s t) \wedge fst (f t u) \longrightarrow fst (f s u)) \end{aligned}$$

shows

$$\begin{aligned} & (snd (lex-ext f n ss ts) \wedge fst (lex-ext f n ts us) \longrightarrow fst (lex-ext f n ss us)) \wedge \\ & (fst (lex-ext f n ss ts) \wedge snd (lex-ext f n ts us) \longrightarrow fst (lex-ext f n ss us)) \wedge \\ & (snd (lex-ext f n ss ts) \wedge snd (lex-ext f n ts us) \longrightarrow snd (lex-ext f n ss us)) \wedge \\ & (fst (lex-ext f n ss ts) \wedge fst (lex-ext f n ts us) \longrightarrow fst (lex-ext f n ss us)) \end{aligned}$$

proof –

```

let ?ls = length ss
let ?lt = length ts
let ?lu = length us
let ?st = lex-ext f n ss ts
let ?tu = lex-ext f n ts us
let ?su = lex-ext f n ss us
let ?fst =  $\lambda$  ss ts i. fst (f (ss ! i) (ts ! i))
let ?snd =  $\lambda$  ss ts i. snd (f (ss ! i) (ts ! i))
let ?ex =  $\lambda$  ss ts.  $\exists$  i < length ss. i < length ts  $\wedge$  ( $\forall$  j < i. ?snd ss ts j)  $\wedge$  ?fst
ss ts i
let ?all =  $\lambda$  ss ts.  $\forall$  i < length ts. ?snd ss ts i
have lengths: (?ls = ?lt  $\vee$  ?lt  $\leq$  n)  $\wedge$  (?lt = ?lu  $\vee$  ?lu  $\leq$  n)  $\longrightarrow$ 
(?ls = ?lu  $\vee$  ?lu  $\leq$  n) (is ?lst  $\wedge$  ?ltu  $\longrightarrow$  ?lsu) by arith
{
  assume st: snd ?st and tu: fst ?tu
  with lengths have lsu: ?lsu by (simp add: lex-ext-iff)
  from st have st: ?ex ss ts  $\vee$  ?all ss ts  $\wedge$  ?lt  $\leq$  ?ls by (simp add: lex-ext-iff)
  from tu have tu: ?ex ts us  $\vee$  ?all ts us  $\wedge$  ?lu < ?lt by (simp add: lex-ext-iff)
  from st have fst ?su
  proof
    assume st: ?ex ss ts
    then obtain i1 where i1: i1 < ?ls  $\wedge$  i1 < ?lt and fst1: ?fst ss ts i1 and
snd1:  $\forall$  j < i1. ?snd ss ts j by force
    from tu show ?thesis
    proof
      assume tu: ?ex ts us
      then obtain i2 where i2: i2 < ?lt  $\wedge$  i2 < ?lu and fst2: ?fst ts us i2 and
snd2:  $\forall$  j < i2. ?snd ts us j by auto
      let ?i = min i1 i2
      from i1 i2 have i: ?i < ?ls  $\wedge$  ?i < ?lt  $\wedge$  ?i < ?lu by auto
      then have ssi: ss ! ?i  $\in$  set ss and tsi: ts ! ?i  $\in$  set ts and usi: us ! ?i  $\in$ 
set us by auto
      have snd:  $\forall$  j < ?i. ?snd ss us j
      proof (intro allI impI)
        fix j
        assume j: j < ?i
        with snd1 snd2 have snd1: ?snd ss ts j and snd2: ?snd ts us j by auto
        from j i have ssj: ss ! j  $\in$  set ss and tsj: ts ! j  $\in$  set ts and usj: us ! j  $\in$ 
set us by auto

```

```

    from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
  qed
  have fst: ?fst ss us ?i
  proof (cases i1 < i2)
    case True
    then have ?i = i1 by simp
    with True fst1 snd2 have ?fst ss ts ?i and ?snd ts us ?i by auto
    with compat[OF ssi tsi usi] show ?fst ss us ?i by auto
  next
    case False
    show ?thesis
    proof (cases i2 < i1)
      case True
      then have ?i = i2 by simp
      with True snd1 fst2 have ?snd ss ts ?i and ?fst ts us ?i by auto
      with compat[OF ssi tsi usi] show ?fst ss us ?i by auto
    next
      case False
      with ⟨¬ i1 < i2⟩ have i1 = i2 by simp
      with fst1 fst2 have ?fst ss ts ?i and ?fst ts us ?i by auto
      with compat[OF ssi tsi usi] show ?fst ss us ?i by auto
    qed
  qed
  show ?thesis by (simp add: lex-ext-iff lsu, rule disjI1, rule exI[of - ?i], simp
  add: fst snd i)
next
  assume tu: ?all ts us ∧ ?lu < ?lt
  show ?thesis
  proof (cases i1 < ?lu)
    case True
    then have usi: us ! i1 ∈ set us by auto
    from i1 have ssi: ss ! i1 ∈ set ss and tsi: ts ! i1 ∈ set ts by auto
    from True tu have ?snd ts us i1 by auto
    with fst1 compat[OF ssi tsi usi] have fst: ?fst ss us i1 by auto
    have snd: ∀ j < i1. ?snd ss us j
    proof (intro allI impI)
      fix j
      assume j < i1
      with i1 True snd1 tu have snd1: ?snd ss ts j and snd2: ?snd ts us j
and
      ssj: ss ! j ∈ set ss and tsj: ts ! j ∈ set ts and usj: us ! j ∈ set us by
auto
      from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
    qed
  with fst lsu True i1 show ?thesis by (auto simp: lex-ext-iff)
next
  case False
  with i1 have lus: ?lu < ?ls by auto
  have snd: ∀ j < ?lu. ?snd ss us j

```

```

    proof (intro allI impI)
      fix j
      assume j < ?lu
      with False i1 snd1 tu have snd1: ?snd ss ts j and snd2: ?snd ts us j
and
      ssj: ss ! j ∈ set ss and tsj: ts ! j ∈ set ts and usj: us ! j ∈ set us by
auto
      from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
      qed
      with lus lsu show ?thesis by (auto simp: lex-ext-iff)
    qed
  qed
next
  assume st: ?all ss ts ∧ ?lt ≤ ?ls
  from tu
  show ?thesis
  proof
    assume tu: ?ex ts us
    with st obtain i2 where i2: i2 < ?lt ∧ i2 < ?lu and fst2: ?fst ts us i2
and snd2: ∀ j < i2. ?snd ts us j by auto
    from st i2 have i2: i2 < ?ls ∧ i2 < ?lt ∧ i2 < ?lu by auto
    then have ssi: ss ! i2 ∈ set ss and tsi: ts ! i2 ∈ set ts and usi: us ! i2 ∈
set us by auto
    from i2 st have ?snd ss ts i2 by auto
    with fst2 compat[OF ssi tsi usi] have fst: ?fst ss us i2 by auto
    have snd: ∀ j < i2. ?snd ss us j
    proof (intro allI impI)
      fix j
      assume j < i2
      with i2 snd2 st have snd1: ?snd ss ts j and snd2: ?snd ts us j and
      ssj: ss ! j ∈ set ss and tsj: ts ! j ∈ set ts and usj: us ! j ∈ set us by auto
      from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
    qed
    with fst lsu i2 show ?thesis by (auto simp: lex-ext-iff)
  next
    assume tu: ?all ts us ∧ ?lu < ?lt
    with st have lus: ?lu < ?ls by auto
    have snd: ∀ j < ?lu. ?snd ss us j
    proof (intro allI impI)
      fix j
      assume j < ?lu
      with st tu have snd1: ?snd ss ts j and snd2: ?snd ts us j and
      ssj: ss ! j ∈ set ss and tsj: ts ! j ∈ set ts and usj: us ! j ∈ set us by auto
      from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
    qed
    with lus lsu show ?thesis by (auto simp: lex-ext-iff)
  qed
  qed
}

```

```

moreover
{
  assume  $st: fst\ ?st$  and  $tu: snd\ ?tu$ 
  with  $lengths$  have  $lsu: ?lsu$  by (simp add: lex-ext-iff)
  from  $st$  have  $st: ?ex\ ss\ ts \vee ?all\ ss\ ts \wedge ?lt < ?ls$  by (simp add: lex-ext-iff)
  from  $tu$  have  $tu: ?ex\ ts\ us \vee ?all\ ts\ us \wedge ?lu \leq ?lt$  by (simp add: lex-ext-iff)
  from  $st$  have  $fst\ ?su$ 
  proof
    assume  $st: ?ex\ ss\ ts$ 
    then obtain  $i1$  where  $i1: i1 < ?ls \wedge i1 < ?lt$  and  $fst1: ?fst\ ss\ ts\ i1$  and
 $snd1: \forall j < i1. ?snd\ ss\ ts\ j$  by force
    from  $tu$  show  $?thesis$ 
    proof
      assume  $tu: ?ex\ ts\ us$ 
      then obtain  $i2$  where  $i2: i2 < ?lt \wedge i2 < ?lu$  and  $fst2: ?fst\ ts\ us\ i2$  and
 $snd2: \forall j < i2. ?snd\ ts\ us\ j$  by auto
      let  $?i = \min\ i1\ i2$ 
      from  $i1\ i2$  have  $i: ?i < ?ls \wedge ?i < ?lt \wedge ?i < ?lu$  by auto
      then have  $ssi: ss ! ?i \in set\ ss$  and  $tsi: ts ! ?i \in set\ ts$  and  $usi: us ! ?i \in$ 
 $set\ us$  by auto
      have  $snd: \forall j < ?i. ?snd\ ss\ us\ j$ 
      proof (intro allI impI)
        fix  $j$ 
        assume  $j: j < ?i$ 
        with  $snd1\ snd2$  have  $snd1: ?snd\ ss\ ts\ j$  and  $snd2: ?snd\ ts\ us\ j$  by auto
        from  $j\ i$  have  $ssj: ss ! j \in set\ ss$  and  $tsj: ts ! j \in set\ ts$  and  $usj: us ! j \in$ 
 $set\ us$  by auto
        from  $compat[OF\ ssj\ tsj\ usj]$   $snd1\ snd2$  show  $?snd\ ss\ us\ j$  by auto
      qed
      have  $fst: ?fst\ ss\ us\ ?i$ 
      proof (cases i1 < i2)
        case True
          then have  $?i = i1$  by simp
          with True  $fst1\ snd2$  have  $?fst\ ss\ ts\ ?i$  and  $?snd\ ts\ us\ ?i$  by auto
          with  $compat[OF\ ssi\ tsi\ usi]$  show  $?fst\ ss\ us\ ?i$  by auto
        next
          case False
            show  $?thesis$ 
            proof (cases i2 < i1)
              case True
                then have  $?i = i2$  by simp
                with True  $snd1\ fst2$  have  $?snd\ ss\ ts\ ?i$  and  $?fst\ ts\ us\ ?i$  by auto
                with  $compat[OF\ ssi\ tsi\ usi]$  show  $?fst\ ss\ us\ ?i$  by auto
              next
                case False
                  with  $\langle \neg i1 < i2 \rangle$  have  $i1 = i2$  by simp
                  with  $fst1\ fst2$  have  $?fst\ ss\ ts\ ?i$  and  $?fst\ ts\ us\ ?i$  by auto
                  with  $compat[OF\ ssi\ tsi\ usi]$  show  $?fst\ ss\ us\ ?i$  by auto
            qed

```

```

    qed
  show ?thesis by (simp add: lex-ext-iff lsu, rule disjI1, rule exI[of - ?i], simp
add: fst snd i)
next
  assume tu: ?all ts us  $\wedge$  ?lu  $\leq$  ?lt
  show ?thesis
  proof (cases i1 < ?lu)
    case True
      then have usi: us ! i1  $\in$  set us by auto
      from i1 have ssi: ss ! i1  $\in$  set ss and tsi: ts ! i1  $\in$  set ts by auto
      from True tu have ?snd ts us i1 by auto
      with fst1 compat[OF ssi tsi usi] have fst: ?fst ss us i1 by auto
      have snd:  $\forall j < i1. ?snd ss us j$ 
      proof (intro allI impI)
        fix j
        assume j < i1
        with i1 True snd1 tu have snd1: ?snd ss ts j and snd2: ?snd ts us j
and
        ssj: ss ! j  $\in$  set ss and tsj: ts ! j  $\in$  set ts and usj: us ! j  $\in$  set us by
auto
        from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
      qed
      with fst lsu True i1 show ?thesis by (auto simp: lex-ext-iff)
    next
      case False
        with i1 have lus: ?lu < ?ls by auto
        have snd:  $\forall j < ?lu. ?snd ss us j$ 
        proof (intro allI impI)
          fix j
          assume j < ?lu
          with False i1 snd1 tu have snd1: ?snd ss ts j and snd2: ?snd ts us j
and
          ssj: ss ! j  $\in$  set ss and tsj: ts ! j  $\in$  set ts and usj: us ! j  $\in$  set us by
auto
          from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
        qed
        with lus lsu show ?thesis by (auto simp: lex-ext-iff)
      qed
    qed
  next
  assume st: ?all ss ts  $\wedge$  ?lt < ?ls
  from tu
  show ?thesis
  proof
    assume tu: ?ex ts us
    with st obtain i2 where i2: i2 < ?lt  $\wedge$  i2 < ?lu and fst2: ?fst ts us i2
and snd2:  $\forall j < i2. ?snd ts us j$  by auto
    from st i2 have i2: i2 < ?ls  $\wedge$  i2 < ?lt  $\wedge$  i2 < ?lu by auto
    then have ssi: ss ! i2  $\in$  set ss and tsi: ts ! i2  $\in$  set ts and usi: us ! i2  $\in$ 

```

```

set us by auto
  from i2 st have ?snd ss ts i2 by auto
  with fst2 compat[OF ssi tsi usi] have fst: ?fst ss us i2 by auto
  have snd:  $\forall j < i2. ?snd ss us j$ 
  proof (intro allI impI)
    fix j
    assume j < i2
    with i2 snd2 st have snd1: ?snd ss ts j and snd2: ?snd ts us j and
      ssj: ss ! j  $\in$  set ss and tsj: ts ! j  $\in$  set ts and usj: us ! j  $\in$  set us by auto
    from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
  qed
  with fst lsu i2 show ?thesis by (auto simp: lex-ext-iff)
next
assume tu: ?all ts us  $\wedge$  ?lu < ?lt
with st have lus: ?lu < ?ls by auto
have snd:  $\forall j < ?lu. ?snd ss us j$ 
proof (intro allI impI)
  fix j
  assume j < ?lu
  with st tu have snd1: ?snd ss ts j and snd2: ?snd ts us j and
    ssj: ss ! j  $\in$  set ss and tsj: ts ! j  $\in$  set ts and usj: us ! j  $\in$  set us by auto
  from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
qed
with lus lsu show ?thesis by (auto simp: lex-ext-iff)
qed
qed
}
moreover
{
  assume st: snd ?st and tu: snd ?tu
  with lengths have lsu: ?lsu by (simp add: lex-ext-iff)
  from st have st: ?ex ss ts  $\vee$  ?all ss ts  $\wedge$  ?lt  $\leq$  ?ls by (simp add: lex-ext-iff)
  from tu have tu: ?ex ts us  $\vee$  ?all ts us  $\wedge$  ?lu  $\leq$  ?lt by (simp add: lex-ext-iff)
  from st have snd ?su
  proof
    assume st: ?ex ss ts
    then obtain i1 where i1: i1 < ?ls  $\wedge$  i1 < ?lt and fst1: ?fst ss ts i1 and
      snd1:  $\forall j < i1. ?snd ss ts j$  by force
    from tu show ?thesis
    proof
      assume tu: ?ex ts us
      then obtain i2 where i2: i2 < ?lt  $\wedge$  i2 < ?lu and fst2: ?fst ts us i2 and
        snd2:  $\forall j < i2. ?snd ts us j$  by auto
      let ?i = min i1 i2
      from i1 i2 have i: ?i < ?ls  $\wedge$  ?i < ?lt  $\wedge$  ?i < ?lu by auto
      then have ssi: ss ! ?i  $\in$  set ss and tsi: ts ! ?i  $\in$  set ts and usi: us ! ?i  $\in$ 
        set us by auto
      have snd:  $\forall j < ?i. ?snd ss us j$ 
      proof (intro allI impI)

```

```

    fix j
    assume j: j < ?i
    with snd1 snd2 have snd1: ?snd ss ts j and snd2: ?snd ts us j by auto
    from j i have ssj: ss ! j ∈ set ss and tsj: ts ! j ∈ set ts and usj: us ! j ∈
set us by auto
    from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
qed
have fst: ?fst ss us ?i
proof (cases i1 < i2)
  case True
  then have ?i = i1 by simp
  with True fst1 snd2 have ?fst ss ts ?i and ?snd ts us ?i by auto
  with compat[OF ssi tsi usi] show ?fst ss us ?i by auto
next
  case False
  show ?thesis
  proof (cases i2 < i1)
    case True
    then have ?i = i2 by simp
    with True snd1 fst2 have ?snd ss ts ?i and ?fst ts us ?i by auto
    with compat[OF ssi tsi usi] show ?fst ss us ?i by auto
  next
    case False
    with ⟨¬ i1 < i2⟩ have i1 = i2 by simp
    with fst1 fst2 have ?fst ss ts ?i and ?fst ts us ?i by auto
    with compat[OF ssi tsi usi] show ?fst ss us ?i by auto
  qed
qed
show ?thesis by (simp add: lex-ext-iff lsu, rule disjI1, rule exI[of - ?i], simp
add: fst snd i)
next
  assume tu: ?all ts us ∧ ?lu ≤ ?lt
  show ?thesis
  proof (cases i1 < ?lu)
    case True
    then have usi: us ! i1 ∈ set us by auto
    from i1 have ssi: ss ! i1 ∈ set ss and tsi: ts ! i1 ∈ set ts by auto
    from True tu have ?snd ts us i1 by auto
    with fst1 compat[OF ssi tsi usi] have fst: ?fst ss us i1 by auto
    have snd: ∀ j < i1. ?snd ss us j
    proof (intro allI impI)
      fix j
      assume j < i1
      with i1 True snd1 tu have snd1: ?snd ss ts j and snd2: ?snd ts us j
and
      ssj: ss ! j ∈ set ss and tsj: ts ! j ∈ set ts and usj: us ! j ∈ set us by
auto
      from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
    qed
  end

```

```

    with fst lsu True i1 show ?thesis by (auto simp: lex-ext-iff)
  next
    case False
    with i1 have lus: ?lu ≤ ?ls by auto
    have snd: ∀ j < ?lu. ?snd ss us j
    proof (intro allI impI)
      fix j
      assume j < ?lu
      with False i1 snd1 tu have snd1: ?snd ss ts j and snd2: ?snd ts us j
and
      ssj: ss ! j ∈ set ss and tsj: ts ! j ∈ set ts and usj: us ! j ∈ set us by
auto
      from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
    qed
    with lus lsu show ?thesis by (auto simp: lex-ext-iff)
  qed
qed
next
assume st: ?all ss ts ∧ ?lt ≤ ?ls
from tu
show ?thesis
proof
  assume tu: ?ex ts us
  with st obtain i2 where i2: i2 < ?lt ∧ i2 < ?lu and fst2: ?fst ts us i2
and snd2: ∀ j < i2. ?snd ts us j by auto
  from st i2 have i2: i2 < ?ls ∧ i2 < ?lt ∧ i2 < ?lu by auto
  then have ssi: ss ! i2 ∈ set ss and tsi: ts ! i2 ∈ set ts and usi: us ! i2 ∈
set us by auto
  from i2 st have ?snd ss ts i2 by auto
  with fst2 compat[OF ssi tsi usi] have fst: ?fst ss us i2 by auto
  have snd: ∀ j < i2. ?snd ss us j
  proof (intro allI impI)
    fix j
    assume j < i2
    with i2 snd2 st have snd1: ?snd ss ts j and snd2: ?snd ts us j and
      ssj: ss ! j ∈ set ss and tsj: ts ! j ∈ set ts and usj: us ! j ∈ set us by auto
    from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto
  qed
  with fst lsu i2 show ?thesis by (auto simp: lex-ext-iff)
next
assume tu: ?all ts us ∧ ?lu ≤ ?lt
with st have lus: ?lu ≤ ?ls by auto
have snd: ∀ j < ?lu. ?snd ss us j
proof (intro allI impI)
  fix j
  assume j < ?lu
  with st tu have snd1: ?snd ss ts j and snd2: ?snd ts us j and
    ssj: ss ! j ∈ set ss and tsj: ts ! j ∈ set ts and usj: us ! j ∈ set us by auto
  from compat[OF ssj tsj usj] snd1 snd2 show ?snd ss us j by auto

```

```

      qed
      with lus lsu show ?thesis by (auto simp: lex-ext-iff)
    qed
  qed
}
ultimately
show ?thesis using lex-ext-stri-imp-nstri by blast
qed

```

lemma *lex-ext-unbounded-map:*

```

  assumes S:  $\bigwedge i. i < \text{length } ss \implies i < \text{length } ts \implies \text{fst } (r (ss ! i) (ts ! i)) \implies \text{fst } (r (\text{map } f ss ! i) (\text{map } f ts ! i))$ 
  and NS:  $\bigwedge i. i < \text{length } ss \implies i < \text{length } ts \implies \text{snd } (r (ss ! i) (ts ! i)) \implies \text{snd } (r (\text{map } f ss ! i) (\text{map } f ts ! i))$ 
  shows (fst (lex-ext-unbounded r ss ts)  $\longrightarrow$  fst (lex-ext-unbounded r (map f ss) (map f ts)))  $\wedge$ 
  (snd (lex-ext-unbounded r ss ts)  $\longrightarrow$  snd (lex-ext-unbounded r (map f ss) (map f ts)))
  using S NS unfolding lex-ext-unbounded-iff by auto

```

lemma *lex-ext-unbounded-map-S:*

```

  assumes S:  $\bigwedge i. i < \text{length } ss \implies i < \text{length } ts \implies \text{fst } (r (ss ! i) (ts ! i)) \implies \text{fst } (r (\text{map } f ss ! i) (\text{map } f ts ! i))$ 
  and NS:  $\bigwedge i. i < \text{length } ss \implies i < \text{length } ts \implies \text{snd } (r (ss ! i) (ts ! i)) \implies \text{snd } (r (\text{map } f ss ! i) (\text{map } f ts ! i))$ 
  and stri: fst (lex-ext-unbounded r ss ts)
  shows fst (lex-ext-unbounded r (map f ss) (map f ts))
  using lex-ext-unbounded-map[of ss ts r f, OF S NS] stri by blast

```

lemma *lex-ext-unbounded-map-NS:*

```

  assumes S:  $\bigwedge i. i < \text{length } ss \implies i < \text{length } ts \implies \text{fst } (r (ss ! i) (ts ! i)) \implies \text{fst } (r (\text{map } f ss ! i) (\text{map } f ts ! i))$ 
  and NS:  $\bigwedge i. i < \text{length } ss \implies i < \text{length } ts \implies \text{snd } (r (ss ! i) (ts ! i)) \implies \text{snd } (r (\text{map } f ss ! i) (\text{map } f ts ! i))$ 
  and nstri: snd (lex-ext-unbounded r ss ts)
  shows snd (lex-ext-unbounded r (map f ss) (map f ts))
  using lex-ext-unbounded-map[of ss ts r f, OF S NS] nstri by blast

```

Strong normalization with local SN assumption

lemma *lex-ext-SN:*

```

  assumes compat:  $\bigwedge x y z. \llbracket \text{snd } (g x y); \text{fst } (g y z) \rrbracket \implies \text{fst } (g x z)$ 
  shows SN { (ys, xs).  $(\forall y \in \text{set } ys. \text{SN-on } \{ (s, t). \text{fst } (g s t) \} \{y\}) \wedge \text{fst } (lex-ext g m ys xs)$  }
  (is SN { (ys, xs). ?cond ys xs } )

```

proof *(rule ccontr)*

assume $\neg ?thesis$

from *this obtain f where f: $\bigwedge n :: \text{nat}. ?cond (f n) (f (Suc n))$* **unfolding** *SN-defs by auto*

have *m-imp-m: $\bigwedge n. \text{length } (f n) \leq m \implies \text{length } (f (Suc n)) \leq m$*

```

proof –
  fix  $n$ 
  assume  $\text{length } (f\ n) \leq m$ 
  then show  $\text{length } (f\ (\text{Suc } n)) \leq m$ 
    using  $f[\text{of } n]$  by (auto simp: lex-ext-iff)
qed
have lm-imp-m-or-eq:  $\bigwedge n. \text{length } (f\ n) > m \implies \text{length } (f\ (\text{Suc } n)) \leq m \vee$ 
 $\text{length } (f\ n) = \text{length } (f\ (\text{Suc } n))$ 
proof –
  fix  $n$ 
  assume  $\text{length } (f\ n) > m$ 
  then have  $\neg \text{length } (f\ n) \leq m$  by auto
  then show  $\text{length } (f\ (\text{Suc } n)) \leq m \vee \text{length } (f\ n) = \text{length } (f\ (\text{Suc } n))$ 
    using  $f[\text{of } n]$  by (simp add: lex-ext-iff, blast)
qed
let  $?l0 = \max (\text{length } (f\ 0))\ m$ 
have  $\bigwedge n. \text{length } (f\ n) \leq ?l0$ 
proof –
  fix  $n$ 
  show  $\text{length } (f\ n) \leq ?l0$ 
  proof (induct n, simp)
    case (Suc n)
    show  $?case$ 
    proof (cases length (f n) ≤ m)
      case True
      with m-imp-m[of n] show  $?thesis$  by auto
    next
    case False
    then have  $\text{length } (f\ n) > m$  by auto
    with lm-imp-m-or-eq[of n]
    have  $\text{length } (f\ n) = \text{length } (f\ (\text{Suc } n)) \vee \text{length } (f\ (\text{Suc } n)) \leq m$  by auto
    with Suc show  $?thesis$  by auto
  qed
qed
qed
from this obtain  $m'$  where  $\text{len}: \bigwedge n. \text{length } (f\ n) \leq m'$  by auto
let  $?lexgr = \lambda\ ys\ xs. \text{fst } (\text{lex-ext } g\ m\ ys\ xs)$ 
let  $?lexge = \lambda\ ys\ xs. \text{snd } (\text{lex-ext } g\ m\ ys\ xs)$ 
let  $?gr = \lambda\ t\ s. \text{fst } (g\ t\ s)$ 
let  $?ge = \lambda\ t\ s. \text{snd } (g\ t\ s)$ 
let  $?S = \{ (y, x). \text{fst } (g\ y\ x) \}$ 
let  $?NS = \{ (y, x). \text{snd } (g\ y\ x) \}$ 
let  $?baseSN = \lambda\ ys. \forall y \in \text{set } ys. \text{SN-on } ?S\ \{y\}$ 
let  $?con = \lambda\ ys\ xs\ m'. ?baseSN\ ys \wedge \text{length } ys \leq m' \wedge ?lexgr\ ys\ xs$ 
let  $?confn = \lambda\ m'\ f\ n. ?con\ (f\ n)\ (f\ (\text{Suc } n))\ m'$ 
from compat have compat2:  $?NS\ O\ ?S \subseteq ?S$  by auto
from  $f\ \text{len}$  have  $\exists f. \forall n. ?confn\ m'\ f\ n$  by auto
then show False
proof (induct m')

```

```

case 0
from this obtain f where ?confn 0 f 0 by auto
then have ?lexgr [] (f (Suc 0)) by force
then show False by (simp add: lex-ext-iff)
next
case (Suc m')
from this obtain f where confn:  $\bigwedge n. ?confn (Suc m') f n$  by auto
have ne:  $\bigwedge n. f n \neq []$ 
proof -
  fix n
  show  $f n \neq []$ 
  proof (cases f n)
    case (Cons a b) then show ?thesis by auto
  next
  case Nil
  with confn[of n] show ?thesis by (simp add: lex-ext-iff)
  qed
qed
let ?hf =  $\lambda n. hd (f n)$ 
have ge:  $\bigwedge n. ?ge (?hf n) (?hf (Suc n)) \vee ?gr (?hf n) (?hf (Suc n))$ 
proof -
  fix n
  from ne[of n] obtain a as where n:  $f n = a \# as$  by (cases f n, auto)
  from ne[of Suc n] obtain b bs where sn:  $f (Suc n) = b \# bs$  by (cases f
(Suc n), auto)
  from n sn have ?ge a b  $\vee ?gr a b$ 
  proof (cases ?gr a b, simp, cases ?ge a b, simp)
    assume  $\neg ?gr a b$  and  $\neg ?ge a b$ 
    then have  $g: g a b = (False, False)$  by (cases g a b, auto)
    from confn[of n] have fst (lex-ext g m (f n) (f (Suc n))) (is ?fst) by simp
    have ?fst = False by (simp add: n sn lex-ext-def g lex-ext-unbounded.simps)
    with <?fst> show ?ge a b  $\vee ?gr a b$  by simp
  qed
  with n sn show ?ge (?hf n) (?hf (Suc n))  $\vee ?gr (?hf n) (?hf (Suc n))$  by
simp
qed
from ge have GE:  $\forall n. (?hf n, ?hf (Suc n)) \in ?NS \cup ?S$  by auto
from confn[of 0] ne[of 0] have SN-0: SN-on ?S {?hf 0} by (cases f 0, auto)
from non-strict-ending[of ?hf, OF GE compat2 SN-0]
obtain j where j:  $\forall i \geq j. (?hf i, ?hf (Suc i)) \in ?NS - ?S$  by auto
let ?h =  $\lambda n. tl (f (j + n))$ 
obtain h where h:  $h = ?h$  by auto
have  $\bigwedge n. ?confn m' h n$ 
proof -
  fix n
  let ?nj =  $j + n$ 
  from spec[OF j, of ?nj]
  have ge-not-gr:  $(?hf ?nj, ?hf (Suc ?nj)) \in ?NS - ?S$  by simp
  from confn[of ?nj] have old: ?confn (Suc m') f ?nj by simp

```

from $ne[of \ ?nj]$ **obtain** a **as where** $n: f \ ?nj = a \ \# \ as$ **by** ($cases \ f \ ?nj, \ auto$)
from $ne[of \ Suc \ ?nj]$ **obtain** $b \ bs$ **where** $sn: f \ (Suc \ ?nj) = b \ \# \ bs$ **by** ($cases \ f \ (Suc \ ?nj), \ auto$)
from old **have** $one: \forall \ y \in \ set \ (h \ n). \ SN\text{-on} \ ?S \ \{y\}$
by ($simp \ add: \ h \ n$)
from old **have** $two: \ length \ (h \ n) \leq \ m'$ **by** ($simp \ add: \ j \ n \ h$)
from $ge\text{-not}\text{-gr}$ **have** $ge\text{-not}\text{-gr}2: \ g \ a \ b = (False, \ True)$ **by** ($simp \ add: \ n \ sn, \ cases \ g \ a \ b, \ auto$)
from old **have** $fst \ (lex\text{-ext} \ g \ m \ (f \ (j+ \ n)) \ (f \ (Suc \ (j+n))))$ (**is** $\ ?fst$) **by** $simp$
then **have** $\ length \ as = \ length \ bs \vee \ length \ bs \leq \ m$ (**is** $\ ?len$)
by ($simp \ add: \ lex\text{-ext}\text{-def} \ n \ sn, \ cases \ ?len, \ auto$)
from $\langle \ ?fst \rangle[simplified \ n \ sn]$ **have** $fst \ (lex\text{-ext}\text{-unbounded} \ g \ as \ bs)$ (**is** $\ ?fst$)
by ($simp \ add: \ lex\text{-ext}\text{-def}, \ cases \ length \ as = \ length \ bs \vee \ Suc \ (length \ bs) \leq \ m, \ simp\text{-all} \ add: \ ge\text{-not}\text{-gr}2 \ lex\text{-ext}\text{-unbounded}.simps$)
then **have** $fst \ (lex\text{-ext}\text{-unbounded} \ g \ as \ bs)$ (**is** $\ ?fst$)
by ($simp \ add: \ lex\text{-ext}\text{-unbounded}\text{-iff}$)
have $three: \ ?lexgr \ (h \ n) \ (h \ (Suc \ n))$
by ($simp \ add: \ lex\text{-ext}\text{-def} \ h \ n \ sn \ ge\text{-not}\text{-gr}2 \ lex\text{-ext}\text{-unbounded}.simps, \ simp \ only: \ Let\text{-def}, \ simp \ add: \ \langle \ ?len \rangle \ \langle \ ?fst \rangle$)
from $one \ two \ three$ **show** $\ ?confn \ m' \ h \ n$ **by** $blast$
qed
with Suc **show** $\ ?thesis$ **by** $blast$
qed
qed

Strong normalization with global SN assumption is immediate consequence.

lemma $lex\text{-ext}\text{-SN}\text{-2}$:

assumes $compat: \bigwedge \ x \ y \ z. \ \llbracket \ snd \ (g \ x \ y); \ fst \ (g \ y \ z) \rrbracket \implies \ fst \ (g \ x \ z)$
and $SN: \ SN \ \{(s, \ t). \ fst \ (g \ s \ t)\}$
shows $SN \ \{(ys, \ xs). \ fst \ (lex\text{-ext} \ g \ m \ ys \ xs) \}$
proof –
from $lex\text{-ext}\text{-SN}[OF \ compat]$
have $SN \ \{(ys, \ xs). \ (\forall \ y \in \ set \ ys. \ SN\text{-on} \ \{(s, \ t). \ fst \ (g \ s \ t)\} \ \{y\}) \wedge \ fst \ (lex\text{-ext} \ g \ m \ ys \ xs) \}$.
then **show** $\ ?thesis$ **using** $SN \ unfolding \ SN\text{-on}\text{-def}$ **by** $fastforce$
qed

The empty list is the least element in the lexicographic extension.

lemma $lex\text{-ext}\text{-least}\text{-1}$: $snd \ (lex\text{-ext} \ f \ m \ xs \ [])$
by ($simp \ add: \ lex\text{-ext}\text{-iff}$)

lemma $lex\text{-ext}\text{-least}\text{-2}$: $\neg \ fst \ (lex\text{-ext} \ f \ m \ [] \ ys)$
by ($simp \ add: \ lex\text{-ext}\text{-iff}$)

Preservation of totality on lists of same length.

lemma $lex\text{-ext}\text{-unbounded}\text{-total}$:

assumes $\forall \ (s, \ t) \in \ set \ (zip \ ss \ ts). \ s = t \vee \ fst \ (f \ s \ t) \vee \ fst \ (f \ t \ s)$
and $refl: \bigwedge \ t. \ snd \ (f \ t \ t)$

```

    and length ss = length ts
  shows ss = ts  $\vee$  fst (lex-ext-unbounded f ss ts)  $\vee$  fst (lex-ext-unbounded f ts ss)
  using assms(3, 1)
proof (induct ss ts rule: list-induct2)
  case (Cons s ss t ts)
  from Cons(3) have s = t  $\vee$  (fst (f s t)  $\vee$  fst (f t s)) by auto
  then show ?case
  proof
    assume st: s = t
    then show ?thesis using Cons(2-3) refl[of t] by (cases f t t, auto simp:
lex-ext-unbounded.simps)
  qed (auto simp: lex-ext-unbounded.simps split: prod.splits)
qed simp

```

```

lemma lex-ext-total:
  assumes  $\forall (s, t) \in \text{set } (\text{zip } ss \ ts). s = t \vee \text{fst } (f \ s \ t) \vee \text{fst } (f \ t \ s)$ 
    and  $\bigwedge t. \text{snd } (f \ t \ t)$ 
    and len: length ss = length ts
  shows ss = ts  $\vee$  fst (lex-ext f n ss ts)  $\vee$  fst (lex-ext f n ts ss)
  using lex-ext-unbounded-total[OF assms] unfolding lex-ext-def Let-def len by
  auto

```

Monotonicity of the lexicographic extension.

```

lemma lex-ext-unbounded-mono:
  assumes  $\bigwedge i. \llbracket i < \text{length } xs; i < \text{length } ys; \text{fst } (P \ (xs \ ! \ i) \ (ys \ ! \ i)) \rrbracket \implies \text{fst } (P' \ (xs \ ! \ i) \ (ys \ ! \ i))$ 
    and  $\bigwedge i. \llbracket i < \text{length } xs; i < \text{length } ys; \text{snd } (P \ (xs \ ! \ i) \ (ys \ ! \ i)) \rrbracket \implies \text{snd } (P' \ (xs \ ! \ i) \ (ys \ ! \ i))$ 
  shows
    (fst (lex-ext-unbounded P xs ys)  $\longrightarrow$  fst (lex-ext-unbounded P' xs ys))  $\wedge$ 
    (snd (lex-ext-unbounded P xs ys)  $\longrightarrow$  snd (lex-ext-unbounded P' xs ys))
    (is (?l1 xs ys  $\longrightarrow$  ?r1 xs ys)  $\wedge$  (?l2 xs ys  $\longrightarrow$  ?r2 xs ys))
  using assms
proof (induct  $x \equiv P \ xs \ ys$  rule: lex-ext-unbounded.induct)
  note [simp] = lex-ext-unbounded.simps
  case (4 x xs y ys)
  consider (TT) P x y = (True, True)
    | (TF) P x y = (True, False)
    | (FT) P x y = (False, True)
    | (FF) P x y = (False, False) by (cases P x y, auto)
  thus ?case
proof cases
  case TT
  moreover
  with 4(2) [of 0] and 4(3) [of 0]
  have P' x y = (True, True)
    by (auto) (metis (full-types) prod.collapse)
  ultimately
  show ?thesis by simp

```

```

next
  case TF
  show ?thesis
  proof (cases snd (P' x y))
    case False
    moreover
    with 4(2) [of 0] and TF
    have P' x y = (True, False)
      by (cases P' x y, auto)
    ultimately
    show ?thesis by simp
  next
  case True
  with 4(2) [of 0] and TF
  have P' x y = (True, True)
    by (auto) (metis (full-types) fst-conv snd-conv surj-pair)
  then show ?thesis by simp
qed
next
  case FF then show ?thesis by simp
next
  case FT
  show ?thesis
  proof (cases fst (P' x y))
    case True
    with 4(3) [of 0] and FT
    have *: P' x y = (True, True)
      by (auto) (metis (full-types) prod.collapse)
    have ?l1 (x#xs) (y#ys)  $\longrightarrow$  ?r1 (x#xs) (y#ys)
      by (simp add: FT *)
    moreover
    have ?l2 (x#xs) (y#ys)  $\longrightarrow$  ?r2 (x#xs) (y#ys)
      by (simp add: *)
    ultimately show ?thesis by blast
  next
  case False
  with 4(3) [of 0] and FT
  have *: P' x y = (False, True)
    by (cases P' x y, auto)
  show ?thesis
    using 4(1) [OF refl FT [symmetric]] and 4(2) and 4(3)
    using FT *
    by (auto) (metis Suc-less-eq nth-Cons-Suc+)
  qed
qed
qed (simp add: lex-ext-unbounded.simps)+
```

lemma *lex-ext-local-mono* [*mono*]:
 assumes $\bigwedge s t. s \in \text{set } ts \implies t \in \text{set } ss \implies \text{ord } s t \implies \text{ord}' s t$

```

shows fst (lex-ext (λ x y. (ord x y, (x, y) ∈ ns-rel)) (length ts) ts ss) →
      fst (lex-ext (λ x y. (ord' x y, (x, y) ∈ ns-rel)) (length ts) ts ss)
proof
  assume ass: fst (lex-ext (λ x y. (ord x y, (x, y) ∈ ns-rel)) (length ts) ts ss)
  from assms have mono: (∧ i. i < length ts ⇒ i < length ss ⇒ ord (ts ! i) (ss
! i) ⇒ ord' (ts ! i) (ss ! i))
    using nth-mem by blast
  let ?P = (λ x y. (ord x y, (x, y) ∈ ns-rel))
  let ?P' = (λ x y. (ord' x y, (x, y) ∈ ns-rel))
  from ass have lex: fst (lex-ext-unbounded ?P ts ss) unfolding lex-ext-def Let-def
if-distrib
    by (auto split: if-splits)
  have fst (lex-ext-unbounded ?P' ts ss)
    by (rule lex-ext-unbounded-mono[THEN conjunct1, rule-format, OF - - lex],
insert mono, auto)
  thus fst (lex-ext (λ x y. (ord' x y, (x, y) ∈ ns-rel)) (length ts) ts ss)
    using ass unfolding lex-ext-def by (auto simp: Let-def)
qed

lemma lex-ext-mono [mono]:
  assumes ∧ s t. ord s t → ord' s t
  shows fst (lex-ext (λ x y. (ord x y, (x, y) ∈ ns)) (length ts) ts ss) →
      fst (lex-ext (λ x y. (ord' x y, (x, y) ∈ ns)) (length ts) ts ss)
  using assms lex-ext-local-mono[of ts ss ord ord' ns] by blast

```

end

4 KBO

Below, we formalize a variant of KBO that includes subterm coefficient functions.

A more standard definition is obtained by setting all subterm coefficients to 1. For this special case it would be possible to define more efficient code-equations that do not have to evaluate subterm coefficients at all.

theory KBO

imports

Lexicographic-Extension

First-Order-Terms.Subterm-and-Context

Polynomial-Factorization.Missing-List

begin

4.1 Subterm Coefficient Functions

Given a function *scf*, associating positions with subterm coefficients, and a list *xs*, the function *scf-list* yields an expanded list where each element of *xs* is replicated a number of times according to its subterm coefficient.

definition *scf-list* :: (nat ⇒ nat) ⇒ 'a list ⇒ 'a list

where
 $scf\text{-list } scf\ xs = concat\ (map\ (\lambda(x, i). replicate\ (scf\ i)\ x)\ (zip\ xs\ [0\ ..<\ length\ xs]))$

lemma *set-scf-list* [*simp*]:
assumes $\forall i < length\ xs. scf\ i > 0$
shows $set\ (scf\text{-list } scf\ xs) = set\ xs$
using *assms* **by** (*auto simp: scf-list-def set-zip set-conv-nth[of xs]*)

lemma *scf-list-subset*: $set\ (scf\text{-list } scf\ xs) \subseteq set\ xs$
by (*auto simp: scf-list-def set-zip*)

lemma *scf-list-empty* [*simp*]:
 $scf\text{-list } scf\ [] = []$ **by** (*auto simp: scf-list-def*)

lemma *scf-list-bef-i-aft* [*simp*]:
 $scf\text{-list } scf\ (bef\ @\ i\ \# \text{aft}) =$
 $scf\text{-list } scf\ bef\ @\ replicate\ (scf\ (length\ bef))\ i\ @$
 $scf\text{-list } (\lambda i. scf\ (Suc\ (length\ bef + i)))\ \text{aft}$
unfolding *scf-list-def*

proof (*induct aft rule: List.rev-induct*)

case (*snoc a aft*)

define *bia* **where** $bia = bef\ @\ i\ \# \text{aft}$

have *bia*: $bef\ @\ i\ \# \text{aft}\ @\ [a] = bia\ @\ [a]$ **by** (*simp add: bia-def*)

have *zip*: $zip\ (bia\ @\ [a])\ [0..<length\ (bia\ @\ [a])]$

$= zip\ bia\ [0..<length\ bia]\ @\ [(a, length\ bia)]$ **by** *simp*

have *concat*:

$concat\ (map\ (\lambda(x, i). replicate\ (scf\ i)\ x)\ (zip\ bia\ [0..<length\ bia]\ @\ [(a, length\ bia)])) =$

$concat\ (map\ (\lambda(x, i). replicate\ (scf\ i)\ x)\ (zip\ bia\ [0..<length\ bia]))\ @$

$replicate\ (scf\ (length\ bia))\ a$ **by** *simp*

show *?case*

unfolding *bia zip concat*

unfolding *bia-def snoc*

by *simp*

qed *simp*

lemma *scf-list-map* [*simp*]:
 $scf\text{-list } scf\ (map\ f\ xs) = map\ f\ (scf\text{-list } scf\ xs)$
by (*induct xs rule: List.rev-induct*) (*auto simp: scf-list-def*)

The function *scf-term* replicates each argument a number of times according to its subterm coefficient function.

fun *scf-term* :: $(f \times nat \Rightarrow nat \Rightarrow nat) \Rightarrow (f, 'v)\ term \Rightarrow (f, 'v)\ term$

where

$scf\text{-term } scf\ (Var\ x) = (Var\ x) |$

$scf\text{-term } scf\ (Fun\ f\ ts) = Fun\ f\ (scf\text{-list } (scf\ (f, length\ ts))\ (map\ (scf\text{-term } scf)\ ts))$

```

lemma vars-term-scf-subset:
  vars-term (scf-term scf s)  $\subseteq$  vars-term s
proof (induct s)
  case (Fun f ss)
  have vars-term (scf-term scf (Fun f ss)) =
    ( $\bigcup_{x \in \text{set } (scf\text{-list } (scf (f, \text{length } ss)) ss). \text{vars-term } (scf\text{-term } scf x)}$ ) by auto
  also have ...  $\subseteq$  ( $\bigcup_{x \in \text{set } ss. \text{vars-term } (scf\text{-term } scf x)}$ )
    using scf-list-subset [of - ss] by blast
  also have ...  $\subseteq$  ( $\bigcup_{x \in \text{set } ss. \text{vars-term } x}$ ) using Fun by auto
  finally show ?case by auto
qed auto

```

```

lemma scf-term-subst:
  scf-term scf (t  $\cdot$   $\sigma$ ) = scf-term scf t  $\cdot$  ( $\lambda x. \text{scf-term } scf (\sigma x)$ )
proof (induct t)
  case (Fun f ts)
  { fix t
    assume t  $\in$  set (scf-list (scf (f, length ts)) ts)
    with scf-list-subset [of - ts] have t  $\in$  set ts by auto
    then have scf-term scf (t  $\cdot$   $\sigma$ ) = scf-term scf t  $\cdot$  ( $\lambda x. \text{scf-term } scf (\sigma x)$ ) by
    (rule Fun) }
  then show ?case by auto
qed auto

```

4.2 Weight Functions

```

locale weight-fun =
  fixes w :: 'f  $\times$  nat  $\Rightarrow$  nat
  and w0 :: nat
  and scf :: 'f  $\times$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat
begin

```

The *weight* of a term is computed recursively, where variables have weight $w0$ and the weight of a compound term is computed by adding the weight of its root symbol $w (f, n)$ to the weighted sum where weights of arguments are multiplied according to their subterm coefficients.

```

fun weight :: ('f, 'v) term  $\Rightarrow$  nat
  where
    weight (Var x) = w0 |
    weight (Fun f ts) =
      (let n = length ts; scff = scf (f, n) in
       w (f, n) + sum-list (map ( $\lambda (ti, i). \text{weight } ti * \text{scff } i$ ) (zip ts [0 ..< n])))

```

Alternatively, we can replicate arguments via *scf-list*. The advantage is that then both *weight* and *scf-term* are defined via *scf-list*.

```

lemma weight-simp [simp]:
  weight (Fun f ts) = w (f, length ts) + sum-list (map weight (scf-list (scf (f,
  length ts)) ts))
proof -

```

```

define scff where scff = (scf (f, length ts) :: nat ⇒ nat)
have ( $\sum (ti, i) \leftarrow \text{zip } ts [0..<\text{length } ts]. \text{weight } ti * \text{scff } i =$ 
  sum-list (map weight (scf-list scff ts)))
proof (induct ts rule: List.rev-induct)
  case (snoc t ts)
  moreover
  { fix n
    have sum-list (replicate n (weight t)) = n * weight t by (induct n) auto }
  ultimately show ?case by (simp add: scf-list-def)
qed simp
then show ?thesis by (simp add: Let-def scff-def)
qed

declare weight.simps(2)[simp del]

abbreviation SCF ≡ scf-term scf

lemma sum-list-scf-list:
  assumes  $\bigwedge i. i < \text{length } ts \implies f i > 0$ 
  shows sum-list (map weight ts) ≤ sum-list (map weight (scf-list f ts))
  using assms unfolding scf-list-def
proof (induct ts rule: List.rev-induct)
  case (snoc t ts)
  have sum-list (map weight ts) ≤
    sum-list (map weight (concat (map ( $\lambda(x, i). \text{replicate } (f i) x$ ) (zip ts [0..<length
ts])))))
  by (auto intro!: snoc)
  moreover
  from snoc(2) [of length ts] obtain n where f (length ts) = Suc n by (auto elim:
lessE)
  ultimately show ?case by simp
qed simp

end

```

4.3 Definition of KBO

The precedence is given by three parameters:

- a predicate *pr-strict* for strict decrease between two function symbols,
- a predicate *pr-weak* for weak decrease between two function symbols,
and
- a function indicating whether a symbol is least in the precedence.

```

locale kbo = weight-fun w w0 scf
for w w0 and scf :: 'f × nat ⇒ nat ⇒ nat +
fixes least :: 'f ⇒ bool

```

```

and pr-strict :: 'f × nat ⇒ 'f × nat ⇒ bool
and pr-weak :: 'f × nat ⇒ 'f × nat ⇒ bool
begin

```

The result of *kbo* is a pair of Booleans encoding strict/weak decrease.

Interestingly, the bound on the lengths of the lists in the lexicographic extension is not required for KBO.

```

fun kbo :: ('f, 'v) term ⇒ ('f, 'v) term ⇒ bool × bool
where
  kbo s t = (if (vars-term-ms (SCF t) ⊆# vars-term-ms (SCF s) ∧ weight t ≤
weight s)
  then if (weight t < weight 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))

```

Abbreviations for strict (S) and nonstrict (NS) KBO.

```

abbreviation S ≡ λ s t. fst (kbo s t)
abbreviation NS ≡ λ s t. snd (kbo s t)

```

For code-generation we do not compute the weights of *s* and *t* repeatedly.

```

lemma kbo-code: kbo s t = (let wt = weight t; ws = weight s in
if (vars-term-ms (SCF t) ⊆# vars-term-ms (SCF s) ∧ wt ≤ ws)
then if wt < ws
  then (True, True)
  else (case s of
    Var y ⇒ (False, (case t of Var x ⇒ True | Fun g ts ⇒ ts = [] ∧ least g))
  | Fun f ss ⇒ (case t of
    Var x ⇒ (True, True)
  | Fun g ts ⇒ let ff = (f, length ss); gg = (g, length ts) in
    if pr-strict ff gg
      then (True, True)
      else if pr-weak ff gg
      then lex-ext-unbounded kbo ss ts
      else (False, False)))
  else (False, False))
unfolding kbo.simps[of s t] Let-def
by (auto simp del: kbo.simps split: term.splits)

```

end

```

declare kbo.kbo-code[code]
declare weight-fun.weight.simps[code]

```

lemma *mset-replicate-mono*:

```

assumes m1  $\subseteq\#$  m2
shows  $\sum\#$  (mset (replicate n m1))  $\subseteq\#$   $\sum\#$  (mset (replicate n m2))
proof (induct n)
  case (Suc n)
  have  $\sum\#$  (mset (replicate (Suc n) m1)) =
     $\sum\#$  (mset (replicate n m1)) + m1 by simp
  also have ...  $\subseteq\#$   $\sum\#$  (mset (replicate n m1)) + m2 using  $\langle m1 \subseteq\# m2 \rangle$  by
    auto
  also have ...  $\subseteq\#$   $\sum\#$  (mset (replicate n m2)) + m2 using Suc by auto
  finally show ?case by (simp add: union-commute)
qed simp

```

While the locale *kbo* only fixes its parameters, we now demand that these parameters are sensible, e.g., encoding a well-founded precedence, etc.

locale *admissible-kbo* =

```

  kbo w w0 scf least pr-strict pr-weak
  for w w0 pr-strict pr-weak and least :: 'f  $\Rightarrow$  bool and scf +
  assumes w0: w (f, 0)  $\geq$  w0 w0 > 0
    and adm: w (f, 1) = 0  $\implies$  pr-weak (f, 1) (g, n)
    and least: least f = (w (f, 0) = w0  $\wedge$  ( $\forall$  g. w (g, 0) = w0  $\longrightarrow$  pr-weak (g, 0)
      (f, 0)))
    and scf: i < n  $\implies$  scf (f, n) i > 0
    and pr-weak-refl [simp]: pr-weak fn fn
    and pr-weak-trans: pr-weak fn gm  $\implies$  pr-weak gm hk  $\implies$  pr-weak fn hk
    and pr-strict: pr-strict fn gm  $\longleftrightarrow$  pr-weak fn gm  $\wedge$   $\neg$  pr-weak gm fn
    and pr-SN: SN {(fn, gm). pr-strict fn gm}
begin

```

lemma *weight-w0*: weight t \geq w0

```

proof (induct t)
  case (Fun f ts)
  show ?case
  proof (cases ts)
    case Nil
    with w0(1) have w0  $\leq$  w (f, length ts) by auto
    then show ?thesis by auto
  next
    case (Cons s ss)
    then obtain i where i: i < length ts by auto
    from scf[OF this] have scf: 0 < scf (f, length ts) i by auto
    then obtain n where scf: scf (f, length ts) i = Suc n by (auto elim: lessE)
    from id-take-nth-drop[OF i] obtain bef aft where ts: ts = bef @ ts ! i # aft
  and ii: length bef = i by auto
  define tsi where tsi = ts ! i
  note ts = ts[folded tsi-def]

```

```

from  $i$  have  $tsi$ :  $tsi \in \text{set } ts$  unfolding  $tsi\text{-def}$  by  $auto$ 
from  $Fun[OF \text{this}]$  have  $w0$ :  $w0 \leq \text{weight } tsi$  .
show  $?thesis$  using  $scf \ ii \ w0$  unfolding  $ts$ 
  by  $simp$ 
qed
qed  $simp$ 

lemma  $\text{weight-gt-0}$ :  $\text{weight } t > 0$ 
  using  $\text{weight-w0}$   $[of \ t]$  and  $w0$  by  $arith$ 

lemma  $\text{weight-0}$   $[iff]$ :  $\text{weight } t = 0 \longleftrightarrow False$ 
  using  $\text{weight-gt-0}$   $[of \ t]$  by  $auto$ 

lemma  $\text{not-S-Var}$ :  $\neg S (\text{Var } x) \ t$ 
  using  $\text{weight-w0}$   $[of \ t]$  by  $(\text{cases } t, \text{auto})$ 

lemma  $S\text{-imp-NS}$ :  $S \ s \ t \implies NS \ s \ t$ 
proof  $(\text{induct } s \ t \text{ rule: } kbo.\text{induct})$ 
  case  $(1 \ s \ t)$ 
  from  $1(2)$  have  $S$ :  $S \ s \ t$  .
  from  $S$  have  $w$ :  $\text{vars-term-ms} (SCF \ t) \subseteq \# \ \text{vars-term-ms} (SCF \ s) \wedge \text{weight } t \leq$ 
 $\text{weight } s$ 
  by  $(\text{auto split: if-splits})$ 
  note  $S = S \ w$ 
  note  $IH = 1(1)[OF \ w]$ 
  show  $?case$ 
  proof  $(\text{cases } \text{weight } t < \text{weight } s)$ 
  case  $True$ 
  with  $S$  show  $?thesis$  by  $simp$ 
  next
  case  $False$ 
  note  $IH = IH[OF \ False]$ 
  note  $S = S \ False$ 
  from  $\text{not-S-Var}[of \ - \ t] \ S$ 
  obtain  $f \ ss$  where  $s$ :  $s = Fun \ f \ ss$  by  $(\text{cases } s, \text{auto})$ 
  note  $IH = IH[OF \ s]$ 
  show  $?thesis$ 
  proof  $(\text{cases } t)$ 
  case  $(\text{Var } x)$ 
  from  $S$  show  $?thesis$  by  $(\text{auto, insert } \text{Var } s, \text{auto})$ 
  next
  case  $(Fun \ g \ ts)$ 
  note  $IH = IH[OF \ Fun]$ 
  let  $?f = (f, \text{length } ss)$ 
  let  $?g = (g, \text{length } ts)$ 
  let  $?lex = \text{lex-ext-unbounded } kbo \ ss \ ts$ 
  from  $S[\text{simplified, unfolded } s \ Fun]$  have  $\text{disj}$ :  $\text{pr-strict } ?f \ ?g \vee \text{pr-weak } ?f \ ?g$ 
 $\wedge \text{fst } ?lex$  by  $(\text{auto split: if-splits})$ 
  show  $?thesis$ 

```

```

proof (cases pr-strict ?f ?g)
  case True
  then show ?thesis using S s Fun by auto
next
  case False
  with disj have fg: pr-weak ?f ?g and lex: fst ?lex by auto
  note IH = IH[OF False fg]
  from lex have fst (lex-ext kbo (length ss + length ts) ss ts)
    unfolding lex-ext-def Let-def by auto
  from lex-ext-stri-imp-nstri[OF this] have lex: snd ?lex
    unfolding lex-ext-def Let-def by auto
  with False fg S s Fun show ?thesis by auto
qed
qed
qed
qed

```

4.4 Reflexivity and Irreflexivity

lemma NS-refl: NS s s

```

proof (induct s)
  case (Fun f ss)
  have snd (lex-ext kbo (length ss) ss ss)
    by (rule all-nstri-imp-lex-nstri, insert Fun[unfolded set-conv-nth], auto)
  then have snd (lex-ext-unbounded kbo ss ss) unfolding lex-ext-def Let-def by
simp
  then show ?case by auto
qed simp

```

lemma pr-strict-irrefl: \neg pr-strict fn fn
unfolding pr-strict **by** auto

lemma S-irrefl: \neg S t t

```

proof (induct t)
  case (Var x) then show ?case by (rule not-S-Var)
next
  case (Fun f ts)
  from pr-strict-irrefl have  $\neg$  pr-strict (f, length ts) (f, length ts) .
  moreover
  { assume fst (lex-ext-unbounded kbo ts ts)
    then obtain i where i < length ts and S (ts ! i) (ts ! i)
      unfolding lex-ext-unbounded-iff by auto
    with Fun have False by auto }
  ultimately show ?case by auto
qed

```

4.5 Monotonicity (a.k.a. Closure under Contexts)

lemma S-mono-one:

assumes S: S s t

shows $S (Fun f (ss1 @ s \# ss2)) (Fun f (ss1 @ t \# ss2))$
proof –
let $?ss = ss1 @ s \# ss2$
let $?ts = ss1 @ t \# ss2$
let $?s = Fun f ?ss$
let $?t = Fun f ?ts$
from S **have** $w: weight t \leq weight s$ **and** $v: vars-term-ms (SCF t) \subseteq\# vars-term-ms (SCF s)$
by (*auto split: if-splits*)
have $v': vars-term-ms (SCF ?t) \subseteq\# vars-term-ms (SCF ?s)$ **using** *mset-replicate-mono[OF v]* **by** *simp*
have $w': weight ?t \leq weight ?s$ **using** *sum-list-replicate-mono[OF w]* **by** *simp*
have $lex: fst (lex-ext-unbounded kbo ?ss ?ts)$
unfolding *lex-ext-unbounded-iff fst-conv*
by (*rule disjI1, rule exI[of - length ss1], insert S NS-refl, auto simp del: kbo.simps simp: nth-append*)
show *?thesis* **using** $v' w' lex$ **by** *simp*
qed

lemma *S-ctxt*: $S s t \implies S (C\langle s \rangle) (C\langle t \rangle)$
by (*induct C, auto simp del: kbo.simps intro: S-mono-one*)

lemma *NS-mono-one*:

assumes $NS: NS s t$ **shows** $NS (Fun f (ss1 @ s \# ss2)) (Fun f (ss1 @ t \# ss2))$
proof –
let $?ss = ss1 @ s \# ss2$
let $?ts = ss1 @ t \# ss2$
let $?s = Fun f ?ss$
let $?t = Fun f ?ts$
from NS **have** $w: weight t \leq weight s$ **and** $v: vars-term-ms (SCF t) \subseteq\# vars-term-ms (SCF s)$
by (*auto split: if-splits*)
have $v': vars-term-ms (SCF ?t) \subseteq\# vars-term-ms (SCF ?s)$ **using** *mset-replicate-mono[OF v]* **by** *simp*
have $w': weight ?t \leq weight ?s$ **using** *sum-list-replicate-mono[OF w]* **by** *simp*
have $lex: snd (lex-ext-unbounded kbo ?ss ?ts)$
unfolding *lex-ext-unbounded-iff snd-conv*
proof (*intro disjI2 conjI allI impI*)
fix i
assume $i < length (ss1 @ t \# ss2)$
then show $NS (?ss ! i) (?ts ! i)$ **using** $NS NS-refl$
by (*cases i = length ss1, auto simp del: kbo.simps simp: nth-append*)
qed *simp*
show *?thesis* **using** $v' w' lex$ **by** *simp*
qed

lemma *NS-ctxt*: $NS s t \implies NS (C\langle s \rangle) (C\langle t \rangle)$
by (*induct C, auto simp del: kbo.simps intro: NS-mono-one*)

4.6 The Subterm Property

lemma *NS-Var-imp-eq-least*: $NS (Var\ x)\ t \implies t = Var\ x \vee (\exists\ f.\ t = Fun\ f\ [] \wedge\ least\ f)$

by (*cases* t , *insert weight-w0*[*of* t], *auto split*: *if-splits*)

lemma *kbo-supt-one*: $NS\ s\ (t :: ('f, 'v)\ term) \implies S\ (Fun\ f\ (bef\ @\ s\ \#\ aft))\ t$

proof (*induct* t *arbitrary*: $f\ s\ bef\ aft$)

case ($Var\ x$)

note $NS = this$

let $?ss = bef\ @\ s\ \#\ aft$

let $?t = Var\ x$

have $length\ bef < length\ ?ss$ **by** *auto*

from $scf[OF\ this,\ of\ f]$ **obtain** n **where** $scf:scf\ (f,\ length\ ?ss)\ (length\ bef) = Suc\ n$ **by** (*auto elim*: *lessE*)

obtain X **where** $vars-term-ms\ (SCF\ (Fun\ f\ ?ss)) = vars-term-ms\ (SCF\ s) + X$

by (*simp add*: *o-def scf[simplified]*)

then have $vs: vars-term-ms\ (SCF\ s) \subseteq\# vars-term-ms\ (SCF\ (Fun\ f\ ?ss))$ **by** *simp*

from NS **have** $vt: vars-term-ms\ (SCF\ ?t) \subseteq\# vars-term-ms\ (SCF\ s)$ **by** (*auto split*: *if-splits*)

from $vt\ vs$ **have** $v: vars-term-ms\ (SCF\ ?t) \subseteq\# vars-term-ms\ (SCF\ (Fun\ f\ ?ss))$

by (*rule subset-mset.order-trans*)

from $weight-w0[of\ Fun\ f\ ?ss]\ v$ **show** $?case$ **by** *simp*

next

case ($Fun\ g\ ts\ f\ s\ bef\ aft$)

let $?t = Fun\ g\ ts$

let $?ss = bef\ @\ s\ \#\ aft$

note $NS = Fun(2)$

note $IH = Fun(1)$

have $length\ bef < length\ ?ss$ **by** *auto*

from $scf[OF\ this,\ of\ f]$ **obtain** n **where** $scff:scf\ (f,\ length\ ?ss)\ (length\ bef) = Suc\ n$ **by** (*auto elim*: *lessE*)

note $scff = scff[simplified]$

obtain X **where** $vars-term-ms\ (SCF\ (Fun\ f\ ?ss)) = vars-term-ms\ (SCF\ s) + X$

by (*simp add*: *o-def scff*)

then have $vs: vars-term-ms\ (SCF\ s) \subseteq\# vars-term-ms\ (SCF\ (Fun\ f\ ?ss))$ **by** *simp*

have $ws: weight\ s \leq sum-list\ (map\ weight\ (scf-list\ (scf\ (f,\ length\ ?ss))\ ?ss))$

by (*simp add*: *scff*)

from NS **have** $wt: weight\ ?t \leq weight\ s$ **and**

$vt: vars-term-ms\ (SCF\ ?t) \subseteq\# vars-term-ms\ (SCF\ s)$ **by** (*auto split*: *if-splits*)

from $ws\ wt$ **have** $w: weight\ ?t \leq sum-list\ (map\ weight\ (scf-list\ (scf\ (f,\ length\ ?ss))\ ?ss))$ **by** *simp*

from $vt\ vs$ **have** $v: vars-term-ms\ (SCF\ ?t) \subseteq\# vars-term-ms\ (SCF\ (Fun\ f\ ?ss))$

by *auto*

then have $v': (vars-term-ms\ (SCF\ ?t) \subseteq\# vars-term-ms\ (SCF\ (Fun\ f\ ?ss))) =$

$True$ **by** *simp*

show $?case$

proof (*cases* $weight\ ?t = weight\ (Fun\ f\ ?ss)$)

```

case False
with w v show ?thesis by auto
next
case True
from wt[unfolded True] weight-gt-0[of s]
have wf: w (f, length ?ss) = 0
  and lsum: sum-list (map weight (scf-list (scf (f, length ?ss) bef))) = 0
  sum-list (map weight (scf-list (λ i. (scf (f, length ?ss) (Suc (length bef) +
i)))) aft)) = 0
  and n: n = 0
  by (auto simp: scff)
have sum-list (map weight bef) ≤ sum-list (map weight (scf-list (scf (f, length
?ss) bef))
  by (rule sum-list-scf-list, rule scf, auto)
with lsum(1) have sum-list (map weight bef) = 0 by arith
then have bef: bef = [] using weight-gt-0[of hd bef] by (cases bef, auto)
have sum-list (map weight aft) ≤ sum-list (map weight (scf-list (λ i. (scf (f,
length ?ss) (Suc (length bef) + i)))) aft)
  by (rule sum-list-scf-list, rule scf, auto)
with lsum(2) have sum-list (map weight aft) = 0 by arith
then have aft: aft = [] using weight-gt-0[of hd aft] by (cases aft, auto)
note scff = scff[unfolded bef aft n, simplified]
from bef aft
have ba: bef @ s # aft = [s] by simp
with wf have wf: w (f, 1) = 0 by auto
from wf have wst: weight s = weight ?t using scff unfolding True[unfolded
ba]
  by (simp add: scf-list-def)
let ?g = (g, length ts)
let ?f = (f, 1)
show ?thesis
proof (cases pr-strict ?f ?g)
  case True
  with w v show ?thesis unfolding ba by simp
next
case False
note admf = adm[OF wf]
from admf have pg: pr-weak ?f ?g .
from pg False[unfolded pr-strict] have pr-weak ?g ?f by auto
from pr-weak-trans[OF this admf] have g: ∧ h k. pr-weak ?g (h, k) .
show ?thesis
proof (cases ts)
  case Nil
  have fst (lex-ext-unbounded kbo [s] ts)
  unfolding Nil lex-ext-unbounded-iff by auto
  with pg w v show ?thesis unfolding ba by simp
next
case (Cons t tts)
  {

```

```

    fix x
    assume s: s = Var x
    from NS-Var-imp-eq-least[OF NS[unfolded s Cons]] have False by auto
  }
  then obtain h ss where s: s = Fun h ss by (cases s, auto)
  from NS wst g[of h length ss] pr-strict[of (h, length ss) (g, length ts)] have
lex: snd (lex-ext-unbounded kbo ss ts)
    unfolding s by (auto split: if-splits)
    from lex obtain s0 sss where ss: ss = s0 # sss unfolding Cons
lex-ext-unbounded-iff snd-conv by (cases ss, auto)
    from lex[unfolded ss Cons] have S s0 t ∨ NS s0 t
    by (cases kbo s0 t, simp add: lex-ext-unbounded.simps del: kbo.simps split:
if-splits)
    with S-imp-NS[of s0 t] have NS s0 t by blast
  from IH[OF - this, of h Nil sss] have S: S s t unfolding Cons s ss by simp
  have fst (lex-ext-unbounded kbo [s] ts) unfolding Cons
    unfolding lex-ext-unbounded-iff fst-conv
    by (rule disjI1[OF exI[of - 0]], insert S, auto simp del: kbo.simps)
  then have lex: fst (lex-ext-unbounded kbo [s] ts) = True by simp
  note all = lex wst[symmetric] S pg scff v'
  note all = all[unfolded ba, unfolded s ss Cons]
  have w: weight (Fun f [t]) = weight (t :: ('f, 'v) term) for t
    using wf scff by (simp add: scf-list-def)
  show ?thesis unfolding ba unfolding s ss Cons
    unfolding kbo.simps[of Fun f [Fun h (s0 # sss)]]
    unfolding all w using all by simp
qed
qed
qed
qed

lemma S-supt:
  assumes supt: s ▷ t
  shows S s t
proof -
  from supt obtain C where s: s = C⟨t⟩ and C: C ≠ □ by auto
  show ?thesis unfolding s using C
proof (induct C arbitrary: t)
  case (More f bef C aft t)
  show ?case
proof (cases C = □)
  case True
    from kbo-supt-one[OF NS-refl, of f bef t aft] show ?thesis unfolding True
by simp
  next
  case False
    from kbo-supt-one[OF S-imp-NS[OF More(1)[OF False]], of f bef t aft]
show ?thesis by simp
qed

```

qed *simp*
qed

lemma *NS-supteq*:
assumes $s \supseteq t$
shows $NS\ s\ t$
using *S-imp-NS*[*OF S-supt*[*of s t*]] *NS-refl*[*of s*] **using** *assms*[*unfolded subterm.le-less*]
by *blast*

4.7 Least Elements

lemma *NS-all-least*:
assumes l : *least f*
shows $NS\ t\ (Fun\ f\ [])$
proof (*induct t*)
case (*Var x*)
show *?case* **using** l [*unfolded least*] l
by *auto*
next
case (*Fun g ts*)
show *?case*
proof (*cases ts*)
case (*Cons s ss*)
with Fun [*of s*] **have** $NS\ s\ (Fun\ f\ [])$ **by** *auto*
from *S-imp-NS*[*OF kbo-supt-one*[*OF this, of g Nil ss*]] **show** *?thesis* **unfolding**
Cons **by** *simp*
next
case *Nil*
from $weight-w0$ [*of Fun g []*] **have** w : $weight\ (Fun\ g\ []) \geq weight\ (Fun\ f\ [])$
using l [*unfolded least*] **by** *auto*
from *lex-ext-least-1*
have $snd\ (lex-ext\ kbo\ 0\ [])$.
then **have** lex : $snd\ (lex-ext-unbounded\ kbo\ [])$ **unfolding** *lex-ext-def Let-def*
by *simp*
then **show** *?thesis* **using** w l [*unfolded least*] **unfolding** *Fun Nil* **by** (*auto simp: empty-le*)
qed
qed

lemma *not-S-least*:
assumes l : *least f*
shows $\neg S\ (Fun\ f\ [])\ t$
proof (*cases t*)
case (*Fun g ts*)
show *?thesis* **unfolding** *Fun*
proof
assume S : $S\ (Fun\ f\ [])\ (Fun\ g\ ts)$
from S [*unfolded Fun, simplified*]
have w : $w\ (g,\ length\ ts) + sum-list\ (map\ weight\ (scf-list\ (scf\ (g,\ length\ ts)))$

```

ts)) ≤ weight (Fun f [])
  by (auto split: if-splits)
show False
proof (cases ts)
  case Nil
  with w have w (g, 0) ≤ weight (Fun f []) by simp
  also have weight (Fun f []) ≤ w0 using l[unfolding least] by simp
  finally have g: w (g, 0) = w0 using w0(1)[of g] by auto
  with w Nil l[unfolding least] have gf: w (g, 0) = w (f, 0) by simp
  with S have p: pr-weak (f, 0) (g, 0) unfolding Nil
    by (simp split: if-splits add: pr-strict)
  with l[unfolding least, THEN conjunct2, rule-format, OF g] have p2: pr-weak
(g, 0) (f, 0) by auto
  from p p2 gf S have fst (lex-ext-unbounded kbo [] ts) unfolding Nil
    by (auto simp: pr-strict)
  then show False unfolding lex-ext-unbounded-iff by auto
next
  case (Cons s ss)
  then have ts: ts = [] @ s # ss by auto
  from scf[of 0 length ts g] obtain n where scff: scf (g, length ts) 0 = Suc n
unfolding Cons by (auto elim: lessE)
  let ?e = sum-list (map weight (
    scf-list (λi. scf (g, Suc (length ss)) (Suc i)) ss
  ))
  have w0 + sum-list (map weight (replicate n s)) ≤ weight s + sum-list (map
weight (replicate n s))
    using weight-w0[of s] by auto
  also have ... = sum-list (map weight (replicate (scf (g, length ts) 0) s))
unfolding scff by simp
  also have w (g, length ts) + ... + ?e ≤ w0 using w l[unfolding least] unfolding
ts scf-list-bef-i-aft by auto
  finally have w0 + sum-list (map weight (replicate n s)) + w (g, length ts) +
?e ≤ w0 by arith
  then have wg: w (g, length ts) = 0 and null: ?e = 0 sum-list (map weight
(replicate n s)) = 0 by auto
  from null(2) weight-gt-0[of s] have n: n = 0 by (cases n, auto)
  have sum-list (map weight ss) ≤ ?e
    by (rule sum-list-scf-list, rule scf, auto)
  from this[unfolding null] weight-gt-0[of hd ss] have ss: ss = [] by (cases ss,
auto)
  with Cons have ts: ts = [s] by simp
  note scff = scff[unfolding ts n, simplified]
  from wg ts have wg: w (g, 1) = 0 by auto
  from adm[OF wg, rule-format, of f] have pr-weak (g, 1) (f, 0) by auto
  with S[unfolding Fun ts] l[unfolding least] weight-w0[of s] scff
  have fst (lex-ext-unbounded kbo [] [s])
    by (auto split: if-splits simp: scf-list-def pr-strict)
  then show ?thesis unfolding lex-ext-unbounded-iff by auto
qed

```

```

qed
qed simp

lemma NS-least-least:
  assumes l: least f
    and NS: NS (Fun f []) t
  shows  $\exists g. t = \text{Fun } g [] \wedge \text{least } g$ 
proof (cases t)
  case (Var x)
  show ?thesis using NS unfolding Var by simp
next
  case (Fun g ts)
  from NS[unfolded Fun, simplified]
  have w:  $w(g, \text{length } ts) + \text{sum-list } (\text{map weight } (\text{scf-list } (\text{scf } (g, \text{length } ts)) ts))$ 
 $\leq \text{weight } (\text{Fun } f [])$ 
  by (auto split: if-splits)
  show ?thesis
proof (cases ts)
  case Nil
  with w have  $w(g, 0) \leq \text{weight } (\text{Fun } f [])$  by simp
  also have  $\text{weight } (\text{Fun } f []) \leq w0$  using l[unfolded least] by simp
  finally have  $g: w(g, 0) = w0$  using w0(1)[of g] by auto
  with w Nil l[unfolded least] have  $gf: w(g, 0) = w(f, 0)$  by simp
  with NS[unfolded Fun] have  $p: \text{pr-weak } (f, 0) (g, 0)$  unfolding Nil
  by (simp split: if-splits add: pr-strict)
  have least: least g unfolding least
  proof (rule conjI[OF g], intro allI)
    fix h
    from l[unfolded least] have  $w(h, 0) = w0 \longrightarrow \text{pr-weak } (h, 0) (f, 0)$  by blast
    with pr-weak-trans p show  $w(h, 0) = w0 \longrightarrow \text{pr-weak } (h, 0) (g, 0)$  by blast
  qed
  show ?thesis
  by (rule exI[of - g], unfold Fun Nil, insert least, auto)
next
  case (Cons s ss)
  then have  $ts: ts = [] @ s \# ss$  by auto
  from scf[of 0 length ts g] obtain n where  $\text{scff}: \text{scf } (g, \text{length } ts) 0 = \text{Suc } n$ 
  unfolding Cons by (auto elim: lessE)
  let ?e =  $\text{sum-list } (\text{map weight } (\text{scf-list } (\lambda i. \text{scf } (g, \text{Suc } (\text{length } ss)) (\text{Suc } i)) ss))$ 
  ))
  have  $w0 + \text{sum-list } (\text{map weight } (\text{replicate } n s)) \leq \text{weight } s + \text{sum-list } (\text{map weight } (\text{replicate } n s))$ 
  using weight-w0[of s] by auto
  also have  $\dots = \text{sum-list } (\text{map weight } (\text{replicate } (\text{scf } (g, \text{length } ts) 0) s))$ 
  unfolding scff by simp
  also have  $w(g, \text{length } ts) + \dots + ?e \leq w0$  using w l[unfolded least] unfolding
  ts scf-list-bef-i-aft by auto
  finally have  $w0 + \text{sum-list } (\text{map weight } (\text{replicate } n s)) + w(g, \text{length } ts) +$ 

```

$?e \leq w0$ **by** *arith*
then have $wg: w (g, \text{length } ts) = 0$ **and** $null: ?e = 0$ *sum-list (map weight (replicate n s)) = 0* **by** *auto*
from $null(2)$ *weight-gt-0[of s]* **have** $n: n = 0$ **by** (*cases n, auto*)
have *sum-list (map weight ss) \leq ?e*
by (*rule sum-list-scf-list, rule scf, auto*)
from *this[unfolded null] weight-gt-0[of hd ss]* **have** $ss: ss = []$ **by** (*cases ss, auto*)
with *Cons* **have** $ts: ts = [s]$ **by** *simp*
note $scff = scff[unfolded ts n, simplified]$
from wg ts **have** $wg: w (g, 1) = 0$ **by** *auto*
from *adm[OF wg, rule-format, of f]* **have** *pr-weak (g, 1) (f, 0)* **by** *auto*
with *NS[unfolded Fun ts] l[unfolded least] weight-w0[of s] scff*
have $snd (lex-ext-unbounded kbo [] [s])$
by (*auto split: if-splits simp: scf-list-def pr-strict*)
then show *?thesis unfolding lex-ext-unbounded-iff snd-conv* **by** *auto*
qed
qed

4.8 Stability (a.k.a. Closure under Substitutions)

lemma *weight-subst: weight (t · σ) =*
weight t + sum-mset (image-mset ($\lambda x. weight (\sigma x) - w0$) (vars-term-ms (SCF t)))
proof (*induct t*)
case (*Var x*)
show *?case using weight-w0[of σx]* **by** *auto*
next
case (*Fun f ts*)
let $?ts = scf-list (scf (f, \text{length } ts)) ts$
define sts **where** $sts = ?ts$
have $id: map (\lambda t. weight (t \cdot \sigma)) ?ts = map (\lambda t. weight t + sum-mset (image-mset (\lambda x. weight (\sigma x) - w0) (vars-term-ms (scf-term scf t)))) ?ts$
by (*rule map-cong[OF refl Fun], insert scf-list-subset[of - ts], auto*)
show *?case*
by (*simp add: o-def id, unfold sts-def[symmetric], induct sts, auto*)
qed

lemma *weight-stable-le:*
assumes $ws: weight s \leq weight t$
and $vs: vars-term-ms (SCF s) \subseteq\# vars-term-ms (SCF t)$
shows $weight (s \cdot \sigma) \leq weight (t \cdot \sigma)$
proof –
from $vs[unfolded mset-subset-eq-exists-conv]$ **obtain** u **where** $vt: vars-term-ms (SCF t) = vars-term-ms (SCF s) + u ..$
show *?thesis unfolding weight-subst vt using ws* **by** *auto*
qed

lemma *weight-stable-lt:*

assumes ws : $\text{weight } s < \text{weight } t$
and vs : $\text{vars-term-ms } (SCF\ s) \subseteq\# \text{ vars-term-ms } (SCF\ t)$
shows $\text{weight } (s \cdot \sigma) < \text{weight } (t \cdot \sigma)$
proof –
from $vs[\text{unfolded mset-subset-eq-exists-conv}]$ **obtain** u **where** vt : $\text{vars-term-ms } (SCF\ t) = \text{vars-term-ms } (SCF\ s) + u$..
show $?thesis$ **unfolding** $\text{weight-subst } vt$ **using** ws **by** $auto$
qed

KBO is stable, i.e., closed under substitutions.

lemma $kbo\text{-stable}$:

fixes $\sigma :: ('f, 'v)\ \text{subst}$

assumes $NS\ s\ t$

shows $(S\ s\ t \longrightarrow S\ (s \cdot \sigma)\ (t \cdot \sigma)) \wedge NS\ (s \cdot \sigma)\ (t \cdot \sigma)$ **(is** $?P\ s\ t$)

using $assms$

proof ($\text{induct } s\ \text{arbitrary: } t$)

case $(Var\ y\ t)$

then have $\text{not: } \neg S\ (Var\ y)\ t$ **using** $\text{not-S-Var}[of\ y\ t]$ **by** $auto$

from $NS\text{-Var-imp-eq-least}[OF\ Var]$

have $t = Var\ y \vee (\exists f. t = Fun\ f\ [] \wedge \text{least } f)$ **by** simp

then obtain f **where** $t = Var\ y \vee t = Fun\ f\ [] \wedge \text{least } f$ **by** $auto$

then have $NS\ (Var\ y \cdot \sigma)\ (t \cdot \sigma)$

proof

assume $t = Var\ y$

then show $?thesis$ **using** $NS\text{-refl}[of\ t \cdot \sigma]$ **by** $auto$

next

assume $t = Fun\ f\ [] \wedge \text{least } f$

with $NS\text{-all-least}[of\ f\ Var\ y \cdot \sigma]$ **show** $?thesis$ **by** $auto$

qed

with $\text{not show } ?case$ **by** blast

next

case $(Fun\ f\ ss\ t)$

note $NS = Fun(2)$

note $IH = Fun(1)$

let $?s = Fun\ f\ ss$

define s **where** $s = ?s$

let $?ss = \text{map } (\lambda s. s \cdot \sigma)\ ss$

from NS **have** v : $\text{vars-term-ms } (SCF\ t) \subseteq\# \text{ vars-term-ms } (SCF\ ?s)$ **and** w : $\text{weight } t \leq \text{weight } ?s$

by ($\text{auto split: if-splits}$)

from $\text{weight-stable-le}[OF\ w\ v]$ **have** $w\sigma$: $\text{weight } (t \cdot \sigma) \leq \text{weight } (?s \cdot \sigma)$ **by** $auto$

from $\text{vars-term-ms-subst-mono}[OF\ v, of\ \lambda x. SCF\ (\sigma\ x)]$ **have** $v\sigma$: $\text{vars-term-ms } (SCF\ (t \cdot \sigma)) \subseteq\# \text{ vars-term-ms } (SCF\ (?s \cdot \sigma))$

unfolding scf-term-subst .

show $?case$

proof ($\text{cases } \text{weight } (t \cdot \sigma) < \text{weight } (?s \cdot \sigma)$)

case $True$

with $v\sigma$ **show** $?thesis$ **by** $auto$

next

```

case False
with weight-stable-lt[OF - v, of  $\sigma$ ] w have w: weight t = weight ?s by arith
show ?thesis
proof (cases t)
  case (Var y)
    from set-mset-mono[OF v, folded s-def]
    have  $y \in \text{vars-term}$  (SCF s) unfolding Var by (auto simp: o-def)
    also have  $\dots \subseteq \text{vars-term}$  s by (rule vars-term-scf-subset)
    finally have  $y \in \text{vars-term}$  s by auto
    from supteq-Var[OF this] have  $?s \triangleright \text{Var } y$  unfolding s-def Fun by auto
    from S-supt[OF supt-subst[OF this]] have  $S: S (?s \cdot \sigma) (t \cdot \sigma)$  unfolding
Var .
    from S-imp-NS[OF S] S show ?thesis by auto
  next
    case (Fun g ts) note t = this
    let ?f = (f, length ss)
    let ?g = (g, length ts)
    let ?ts = map ( $\lambda s. s \cdot \sigma$ ) ts
    show ?thesis
    proof (cases pr-strict ?f ?g)
      case True
        then have  $S: S (?s \cdot \sigma) (t \cdot \sigma)$  using  $w\sigma$   $v\sigma$  unfolding t by simp
        from S S-imp-NS[OF S] show ?thesis by simp
      next
        case False note prec = this
        show ?thesis
        proof (cases pr-weak ?f ?g)
          case False
            with v w prec have  $\neg NS$  ?s t unfolding t by (auto simp del:
vars-term-ms.simps)
            with NS show ?thesis by blast
          next
            case True
              from v w have  $\text{vars-term-ms}$  (SCF t)  $\subseteq\#$   $\text{vars-term-ms}$  (SCF ?s)  $\wedge$  weight
t  $\leq$  weight ?s  $\neg$  weight t < weight ?s by auto
              {
                fix i
                assume  $i: i < \text{length } ss$   $i < \text{length } ts$ 
                and  $S: S (ss ! i) (ts ! i)$ 
                have  $S$  (map ( $\lambda s. s \cdot \sigma$ )  $ss ! i$ ) (map ( $\lambda s. s \cdot \sigma$ )  $ts ! i$ )
                using IH[OF - S-imp-NS[OF S]] S i unfolding set-conv-nth by (force
simp del: kbo.simps)
              } note IH-S = this
              {
                fix i
                assume  $i: i < \text{length } ss$   $i < \text{length } ts$ 
                and  $NS: NS (ss ! i) (ts ! i)$ 
                have  $NS$  (map ( $\lambda s. s \cdot \sigma$ )  $ss ! i$ ) (map ( $\lambda s. s \cdot \sigma$ )  $ts ! i$ )
                using IH[OF - NS] i unfolding set-conv-nth by (force simp del:

```

```

kbo.simps)
  } note IH-NS = this
  {
    assume S ?s t
    with prec v w True have lex: fst (lex-ext-unbounded kbo ss ts)
      unfolding s-def t by simp
    have fst (lex-ext-unbounded kbo ?ss ?ts)
      by (rule lex-ext-unbounded-map-S[OF - - lex], insert IH-NS IH-S,
blast+)
    with vσ wσ prec True have S (?s · σ) (t · σ)
      unfolding t by auto
  }
  moreover
  {
    from NS prec v w True have lex: snd (lex-ext-unbounded kbo ss ts)
      unfolding t by simp
    have snd (lex-ext-unbounded kbo ?ss ?ts)
      by (rule lex-ext-unbounded-map-NS[OF - - lex], insert IH-S IH-NS,
blast)
    with vσ wσ prec True have NS (?s · σ) (t · σ)
      unfolding t by auto
  }
  ultimately show ?thesis by auto
qed
qed
qed
qed
qed

```

lemma *S-subst*:

$S s t \implies S (s \cdot (\sigma :: ('f, 'v) \text{subst})) (t \cdot \sigma)$
using *kbo-stable*[*OF S-imp-NS, of s t σ*] **by** *auto*

lemma *NS-subst*: $NS s t \implies NS (s \cdot (\sigma :: ('f, 'v) \text{subst})) (t \cdot \sigma)$ **using** *kbo-stable*[*of s t σ*] **by** *auto*

4.9 Transitivity and Compatibility

lemma *kbo-trans*: $(S s t \longrightarrow NS t u \longrightarrow S s u) \wedge$
 $(NS s t \longrightarrow S t u \longrightarrow S s u) \wedge$
 $(NS s t \longrightarrow NS t u \longrightarrow NS s u)$
(is *?P s t u***)**

proof (*induct s arbitrary: t u*)

case (*Var x t u*)

from *not-S-Var*[*of x t*] **have** *nS*: $\neg S (\text{Var } x) t$.

show *?case*

proof (*cases NS (Var x) t*)

case *False*

with *nS* **show** *?thesis* **by** *blast*

```

next
  case True
  from NS-Var-imp-eq-least[OF this] obtain f where
    t = Var x  $\vee$  t = Fun f []  $\wedge$  least f by blast
  then show ?thesis
  proof
    assume t = Var x
    then show ?thesis using nS by blast
  next
    assume t = Fun f []  $\wedge$  least f
    then have t: t = Fun f [] and least: least f by auto
    from not-S-least[OF least] have nS':  $\neg$  S t u unfolding t .
    show ?thesis
    proof (cases NS t u)
      case True
      with NS-least-least[OF least, of u] t obtain h where
        u: u = Fun h [] and least: least h by auto
      from NS-all-least[OF least] have NS: NS (Var x) u unfolding u .
      with nS nS' show ?thesis by blast
    next
      case False
      with S-imp-NS[of t u] show ?thesis by blast
    qed
  qed
qed
next
case (Fun f ss t u) note IH = this
let ?s = Fun f ss
show ?case
proof (cases NS ?s t)
  case False
  with S-imp-NS[of ?s t] show ?thesis by blast
next
case True note st = this
then have vst: vars-term-ms (SCF t)  $\subseteq$ # vars-term-ms (SCF ?s) and wst:
weight t  $\leq$  weight ?s
  by (auto split: if-splits)
show ?thesis
proof (cases NS t u)
  case False
  with S-imp-NS[of t u] show ?thesis by blast
next
case True note tu = this
then have vtu: vars-term-ms (SCF u)  $\subseteq$ # vars-term-ms (SCF t) and wtu:
weight u  $\leq$  weight t
  by (auto split: if-splits)
from vst vtu have v: vars-term-ms (SCF u)  $\subseteq$ # vars-term-ms (SCF ?s) by
simp
from wst wtu have w: weight u  $\leq$  weight ?s by simp

```

```

show ?thesis
proof (cases weight u < weight ?s)
  case True
  with v show ?thesis by auto
next
  case False
  with wst wtu have wst: weight t = weight ?s and wtu: weight u = weight t
and w: weight u = weight ?s by arith+
  show ?thesis
  proof (cases u)
    case (Var z)
    with v w show ?thesis by auto
  next
    case (Fun h us) note u = this
    show ?thesis
    proof (cases t)
      case (Fun g ts) note t = this
      let ?f = (f, length ss)
      let ?g = (g, length ts)
      let ?h = (h, length us)
      from st t wst have fg: pr-weak ?f ?g by (simp split: if-splits add: pr-strict)
      from tu t u wtu have gh: pr-weak ?g ?h by (simp split: if-splits add:
pr-strict)
      from pr-weak-trans[OF fg gh] have fh: pr-weak ?f ?h .
      show ?thesis
      proof (cases pr-strict ?f ?h)
        case True
        with w v u show ?thesis by auto
      next
        case False
        let ?lex = lex-ext-unbounded kbo
        from False fh have hf: pr-weak ?h ?f unfolding pr-strict by auto
        from pr-weak-trans[OF hf fg] have hg: pr-weak ?h ?g .
        from hg have gh2: ¬ pr-strict ?g ?h unfolding pr-strict by auto
        from pr-weak-trans[OF gh hf] have gf: pr-weak ?g ?f .
        from gf have fg2: ¬ pr-strict ?f ?g unfolding pr-strict by auto
        from st t wst fg2 have st: snd (?lex ss ts)
        by (auto split: if-splits)
        from tu t u wtu gh2 have tu: snd (?lex ts us)
        by (auto split: if-splits)
        {
          fix s t u
          assume s ∈ set ss
          from IH[OF this, of t u]
          have (NS s t ∧ S t u → S s u) ∧
            (S s t ∧ NS t u → S s u) ∧
            (NS s t ∧ NS t u → NS s u) ∧
            (S s t ∧ S t u → S s u)
          using S-imp-NS[of s t] by blast
        }
      }
    }
  }

```

```

} note IH = this
let ?b = length ss + length ts + length us
note lex = lex-ext-compat[of ss ts us kbo ?b, OF IH]
let ?lexb = lex-ext kbo ?b
note conv = lex-ext-def Let-def
from st have st: snd (?lexb ss ts) unfolding conv by simp
from tu have tu: snd (?lexb ts us) unfolding conv by simp
from lex st tu have su: snd (?lexb ss us) by blast
then have su: snd (?lex ss us) unfolding conv by simp
from w v u su fh have NS: NS ?s u by simp
{
  assume st: S ?s t
  with t wst fg fg2 have st: fst (?lex ss ts)
    by (auto split: if-splits)
  then have st: fst (?lexb ss ts) unfolding conv by simp
  from lex st tu have su: fst (?lexb ss us) by blast
  then have su: fst (?lex ss us) unfolding conv by simp
  from w v u su fh have S: S ?s u by simp
} note S-left = this
{
  assume tu: S t u
  with t u wtu gh2 have tu: fst (?lex ts us)
    by (auto split: if-splits)
  then have tu: fst (?lexb ts us) unfolding conv by simp
  from lex st tu have su: fst (?lexb ss us) by blast
  then have su: fst (?lex ss us) unfolding conv by simp
  from w v u su fh have S: S ?s u by simp
} note S-right = this
from NS S-left S-right show ?thesis by blast
qed
next
case (Var x) note t = this
from tu weight-w0[of u] have least: least h and u: u = Fun h [] unfolding
t u
  by (auto split: if-splits)
from NS-all-least[OF least] have NS: NS ?s u unfolding u .
from not-S-Var have nS': ¬ S t u unfolding t .
show ?thesis
proof (cases S ?s t)
  case False
  with nS' NS show ?thesis by blast
next
  case True
  then have vars-term-ms (SCF t) ⊆# vars-term-ms (SCF ?s)
    by (auto split: if-splits)
  from set-mset-mono[OF this, unfolded set-mset-vars-term-ms t]
  have x ∈ vars-term (SCF ?s) by simp
  also have ... ⊆ vars-term ?s by (rule vars-term-scf-subset)
  finally obtain s sss where ss: ss = s # sss by (cases ss, auto)

```

```

      from kbo-supt-one[OF NS-all-least[OF least, of s], of f Nil sss]
      have S ?s u unfolding ss u by simp
      with NS show ?thesis by blast
    qed
  qed
  qed
  qed
  qed
  qed
  qed

```

lemma *S-trans*: $S\ s\ t \implies S\ t\ u \implies S\ s\ u$ **using** *S-imp-NS*[*of s t*] *kbo-trans*[*of s t u*] **by** *blast*

lemma *NS-trans*: $NS\ s\ t \implies NS\ t\ u \implies NS\ s\ u$ **using** *kbo-trans*[*of s t u*] **by** *blast*

lemma *NS-S-compat*: $NS\ s\ t \implies S\ t\ u \implies S\ s\ u$ **using** *kbo-trans*[*of s t u*] **by** *blast*

lemma *S-NS-compat*: $S\ s\ t \implies NS\ t\ u \implies S\ s\ u$ **using** *kbo-trans*[*of s t u*] **by** *blast*

4.10 Strong Normalization (a.k.a. Well-Foundedness)

lemma *kbo-strongly-normalizing*:

fixes $s :: ('f, 'v)$ *term*

shows *SN-on* $\{(s, t). S\ s\ t\}$ $\{s\}$

proof –

let $?SN = \lambda t :: ('f, 'v)$ *term*. *SN-on* $\{(s, t). S\ s\ t\}$ $\{t\}$

let $?m1 = \lambda (f, ss)$. *weight* (*Fun* $f\ ss$)

let $?m2 = \lambda (f, ss)$. (f , *length* ss)

let $?rel' = \text{lex-two } \{(fss, gts). ?m1\ fss > ?m1\ gts\} \{(fss, gts). ?m1\ fss \geq ?m1\ gts\}$ $\{(fss, gts). \text{pr-strict } (?m2\ fss) (?m2\ gts)\}$

let $?rel = \text{inv-image } ?rel' (\lambda x. (x, x))$

have *SN-rel*: *SN* $?rel$

by (*rule* *SN-inv-image*, *rule* *lex-two*, *insert* *SN-inv-image*[*OF pr-SN*, *of ?m2*] *SN-inv-image*[*OF SN-nat-gt*, *of ?m1*],

auto *simp*: *inv-image-def*)

note *conv* = *SN-on-all-reducts-SN-on-conv*

show $?SN\ s$

proof (*induct* s)

case (*Var* x)

show $?case$ **unfolding** *conv*[*of - Var x*] **using** *not-S-Var*[*of x*] **by** *auto*

next

case (*Fun* $f\ ss$)

then **have** *subset*: $\text{set } ss \subseteq \{s. ?SN\ s\}$ **by** *blast*

let $?P = \lambda (f, ss)$. $\text{set } ss \subseteq \{s. ?SN\ s\} \longrightarrow ?SN\ (Fun\ f\ ss)$

{

fix fss

have $?P\ fss$

proof (*induct* fss *rule*: *SN-induct*[*OF SN-rel*])

case ($1\ fss$)

obtain $f\ ss$ **where** $fss: fss = (f, ss)$ **by** *force*

{

```

fix  $g\ ts$ 
assume  $?m1\ (f,\ ss) > ?m1\ (g,\ ts) \vee ?m1\ (f,\ ss) \geq ?m1\ (g,\ ts) \wedge pr\text{-strict}$ 
 $(?m2\ (f,\ ss))\ (?m2\ (g,\ ts))$ 
and  $set\ ts \subseteq \{s.\ ?SN\ s\}$ 
then have  $?SN\ (Fun\ g\ ts)$ 
using  $1[rule\text{-format},\ of\ (g,\ ts),\ unfolded\ fss\ split]$  by auto
} note  $IH = this[unfolded\ split]$ 
show  $?case\ unfolding\ fss\ split$ 
proof
assume  $SN\text{-}s: set\ ss \subseteq \{s.\ ?SN\ s\}$ 
let  $?f = (f,\ length\ ss)$ 
let  $?s = Fun\ f\ ss$ 
let  $?SNT = \lambda\ g\ ts.\ ?SN\ (Fun\ g\ ts)$ 
let  $?sym = \lambda\ g\ ts.\ (g,\ length\ ts)$ 
let  $?lex = lex\text{-ext}\ kbo\ (weight\ ?s)$ 
let  $?lexu = lex\text{-ext}\text{-unbounded}\ kbo$ 
let  $?lex\text{-}SN = \{(ys,\ xs).\ (\forall\ y \in set\ ys.\ ?SN\ y) \wedge fst\ (?lex\ ys\ xs)\}$ 
from  $lex\text{-ext}\text{-}SN[of\ kbo\ weight\ ?s,\ OF\ NS\text{-}S\text{-}compat]$ 
have  $SN: SN\ ?lex\text{-}SN$  .
{
fix  $g$  and  $ts :: ('f,\ 'v)\ term\ list$ 
assume  $pr\text{-weak}\ ?f\ (?sym\ g\ ts) \wedge weight\ (Fun\ g\ ts) \leq weight\ ?s \wedge set\ ts$ 
 $\subseteq \{s.\ ?SN\ s\}$ 
then have  $?SNT\ g\ ts$ 
proof  $(induct\ ts\ arbitrary: g\ rule: SN\text{-}induct[OF\ SN])$ 
case  $(1\ ts\ g)$ 
note  $inner\text{-}IH = 1(1)$ 
let  $?g = (g,\ length\ ts)$ 
let  $?t = Fun\ g\ ts$ 
from  $1(2)$  have  $fg: pr\text{-weak}\ ?f\ ?g$  and  $w: weight\ ?t \leq weight\ ?s$  and
 $SN: set\ ts \subseteq \{s.\ ?SN\ s\}$  by auto
show  $?SNT\ g\ ts\ unfolding\ conv[of\ -\ ?t]$ 
proof  $(intro\ allI\ impI)$ 
fix  $u$ 
assume  $(?t,\ u) \in \{(s,\ t).\ S\ s\ t\}$ 
then have  $tu: S\ ?t\ u$  by auto
then show  $?SN\ u$ 
proof  $(induct\ u)$ 
case  $(Var\ x)$ 
then show  $?case\ using\ not\text{-}S\text{-}Var[of\ x]\ unfolding\ conv[of\ -\ Var$ 
 $x]$  by auto
next
case  $(Fun\ h\ us)$ 
let  $?h = (h,\ length\ us)$ 
let  $?u = Fun\ h\ us$ 
note  $tu = Fun(2)$ 
{
fix  $u$ 
assume  $u: u \in set\ us$ 

```

```

then have ?u > u by auto
from S-trans[OF tu S-supt[OF this]] have S ?t u by auto
from Fun(1)[OF u this] have ?SN u .
} then have SNu: set us ⊆ {s . ?SN s} by blast
note IH = IH[OF - this]
from tu have wut: weight ?u ≤ weight ?t by (simp split: if-splits)
show ?case
proof (cases ?m1 (f, ss) > ?m1 (h, us) ∨ ?m1 (f, ss) ≥ ?m1 (h,
us) ∧ pr-strict (?m2 (f, ss)) (?m2 (h, us)))
  case True
  from IH[OF True[unfolded split]] show ?thesis by simp
next
  case False
  with wut w have wut: weight ?t = weight ?u weight ?s = weight
?u by auto
  note False = False[unfolded split wut]
note tu = tu[unfolded kbo.simps[of ?t] wut, unfolded Fun term.simps,
simplified]
  from tu have gh: pr-weak ?g ?h unfolding pr-strict by (auto
split: if-splits)
  from pr-weak-trans[OF fg gh] have fh: pr-weak ?f ?h .
  from False wut fh have ¬ pr-strict ?f ?h unfolding pr-strict by
auto
  with fh have hf: pr-weak ?h ?f unfolding pr-strict by auto
  from pr-weak-trans[OF hf fg] have hg: pr-weak ?h ?g .
  from hg have gh2: ¬ pr-strict ?g ?h unfolding pr-strict by auto
  from tu gh2 have lex: fst (?lexu ts us) by (auto split: if-splits)
  from fh wut SNu have pr-weak ?f ?h ∧ weight ?u ≤ weight ?s ∧
set us ⊆ {s. ?SN s}
  by auto
  note inner-IH = inner-IH[OF - this]
  show ?thesis
  proof (rule inner-IH, rule, unfold split, intro conjI ballI)
  have fst (?lexu ts us) by (rule lex)
  moreover have length us ≤ weight ?s
  proof –
  have length us ≤ sum-list (map weight us)
  proof (induct us)
  case (Cons u us)
  from Cons have length (u # us) ≤ Suc (sum-list (map weight
us)) by auto
  also have ... ≤ sum-list (map weight (u # us)) using
weight-gt-0[of u]
  by auto
  finally show ?case .
qed simp
  also have ... ≤ sum-list (map weight (scf-list (scf (h, length
us) us))
  by (rule sum-list-scf-list[OF scf])

```

```

    also have ... ≤ weight ?s using wut by simp
    finally show ?thesis .
  qed
  ultimately show fst (?lex ts us) unfolding lex-ext-def Let-def
by auto
  qed (insert SN, blast)
  qed
  qed
  qed
  }
  from this[of f ss] SN-s show ?SN ?s by auto
  qed
  qed
  }
  from this[of (f, ss), unfolded split]
  show ?case using Fun by blast
  qed
qed

```

```

lemma S-SN: SN {(x, y). S x y}
  using kbo-strongly-normalizing unfolding SN-defs by blast

```

4.11 Ground Totality

```

lemma ground-SCF [simp]:
  ground (SCF t) = ground t
proof -
  have *: ∀ i < length xs. scf (f, length xs) i > 0
    for f :: 'f and xs :: ('f, 'v) term list using scf by simp
  show ?thesis by (induct t) (auto simp: set-scf-list [OF *])
qed

```

```

declare kbo.simps[simp del]

```

```

lemma ground-vars-term-ms: ground t ⇒ vars-term-ms t = {#}
  by (induct t) auto

```

```

context
  fixes F :: ('f × nat) set
  assumes pr-weak: pr-weak = pr-strict==
  and pr-gtotal: ∧ f g. f ∈ F ⇒ g ∈ F ⇒ f = g ∨ pr-strict f g ∨ pr-strict g f
begin

```

```

lemma S-ground-total:
  assumes funas-term s ⊆ F and ground s and funas-term t ⊆ F and ground t
  shows s = t ∨ S s t ∨ S t s
  using assms
proof (induct s arbitrary: t)

```

```

case IH: (Fun f ss)
note [simp] = ground-vars-term-ms
let ?s = Fun f ss
have *: (vars-term-ms (SCF t) ⊆# vars-term-ms (SCF ?s)) = True
      (vars-term-ms (SCF ?s) ⊆# vars-term-ms (SCF t)) = True
      using ⟨ground ?s⟩ and ⟨ground t⟩ by (auto simp: scf)
from IH(5) obtain g ts where t[simp]: t = Fun g ts by (cases t, auto)
let ?t = Fun g ts
let ?f = (f, length ss)
let ?g = (g, length ts)
from IH have f: ?f ∈ F and g: ?g ∈ F by auto
{
  assume ¬ ?case
  note contra = this[unfolded kbo.simps[of ?s] kbo.simps[of t] *, unfolded t
term.simps]
  from pr-gtotal[OF f g] contra have fg: ?f = ?g by (auto split: if-splits)
  have IH: ∀ (s, t) ∈ set (zip ss ts). s = t ∨ S s t ∨ S t s
    using IH by (auto elim!: in-set-zipE) blast
  from fg have len: length ss = length ts by auto
  from lex-ext-unbounded-total[OF IH NS-refl len] contra fg
  have False by (auto split: if-splits)
}
then show ?case by blast
qed auto
end

```

4.12 Summary

At this point we have shown well-foundedness S - SN , transitivity and compatibility S - $trans$ NS - $trans$ NS - S - $compat$ S - NS - $compat$, closure under substitutions S - $subst$ NS - $subst$, closure under contexts S - $ctxt$ NS - $ctxt$, the subterm property S - $supt$ NS - $supteq$, reflexivity of the weak NS - $refl$ and irreflexivity of the strict part S - $irrefl$, and ground-totality S - $ground-total$.

In particular, this allows us to show that KBO is an instance of strongly normalizing order pairs (SN - $order-pair$).

```

sublocale SN-order-pair {(x, y). S x y} {(x, y). NS x y}
by (unfold-locales, insert NS-refl NS-trans S-trans S-SN NS-S-compat S-NS-compat)
      (auto simp: refl-on-def trans-def, blast+)
end

end

```

References

- [1] J. Dick, J. Kalmus, and U. Martin. Automating the Knuth-Bendix ordering. *Acta Informatica*, 28(2):95–119, 1990.

- [2] D. E. Knuth and P. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. 1970.
- [3] M. Ludwig and U. Waldmann. An extension of the Knuth–Bendix ordering with LPO-like properties. In *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR’07*, volume 4790 of *LNCS*, pages 348–362, 2007.
- [4] J. Steinbach. Extensions and comparison of simplification orders. In *Rewriting Techniques and Applications, RTA’89*, volume 355 of *LNCS*, pages 434–448, 1989.
- [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] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Theorem Proving in Higher Order Logics, TPHOLs’09*, volume 5674 of *LNCS*, pages 452–468, 2009.
- [7] H. Zankl, N. Hirokawa, and A. Middeldorp. KBO orientability. *Journal of Automated Reasoning*, 43(2):173–201, 2009.