A Formalization of Knuth–Bendix Orders*

Christian Sternagel and René Thiemann

March 17, 2025

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

*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*: $(s, s) \in NS$ ⟨*proof*⟩

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

lemma *trancl-NS*[*order-simps*]: $NS^+ = NS$ ⟨*proof*⟩

lemma *rtrancl-NS*[*order-simps*]: $NS^* = NS$
  ⟨*proof*⟩

lemma *trancl-S*[*order-simps*]: $S^+ = S$ ⟨*proof*⟩

lemma *S-O-S*: $S\ O\ S \subseteq S\ S\ O\ S\ O\ T \subseteq S\ O\ T$
⟨*proof*⟩

lemma *trans-S-point*: $\bigwedge x\ y\ z.\ (x,\ y) \in S \Longrightarrow (y,\ z) \in S \Longrightarrow (x,\ z) \in S$
  ⟨*proof*⟩

lemma *trans-NS-point*: $\bigwedge x\ y\ z.\ (x,\ y) \in NS \Longrightarrow (y,\ z) \in NS \Longrightarrow (x,\ z) \in NS$
  ⟨*proof*⟩
end

locale *compat-pair* =
  fixes *S NS* :: *'a 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*: $\bigwedge x\ y\ z.\ (x,\ y) \in NS \Longrightarrow (y,\ z) \in S \Longrightarrow (x,\ z) \in S$
  ⟨*proof*⟩

lemma *compat-S-NS-point*: $\bigwedge x\ y\ z.\ (x,\ y) \in S \Longrightarrow (y,\ z) \in NS \Longrightarrow (x,\ z) \in S$
  ⟨*proof*⟩

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

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

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*]: $S\ O\ NS = S\ S\ O\ NS\ O\ T = S\ O\ T$ ⟨*proof*⟩
lemma *NS-O-S*[*order-simps*]: $NS\ O\ S = S\ NS\ O\ S\ O\ T = S\ O\ T$ ⟨*proof*⟩

lemma *compat-rtrancl*:
  assumes *ab*: $(a,\ b) \in S$
    and *bc*: $(b,\ c) \in (NS \cup S)^*$
  shows $(a,\ c) \in S$
  ⟨*proof*⟩

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* ⊆ *SN-pair* ⟨*proof*⟩

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 (<∗*lex*∗>) 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* ⇒ *'a rel* ⇒ *'b rel* ⇒ (*'a* × *'b*) *rel*
  where
    lex-two s ns s2 = {((*a1*, *b1*), (*a2*, *b2*)) . (*a1*, *a2*) ∈ *s* ∨ (*a1*, *a2*) ∈ *ns* ∧ (*b1*, *b2*) ∈ *s2*}

lemma *lex-two*:
  assumes *compat*: *ns O s* ⊆ *s*
    and *SN-s*: *SN s*
    and *SN-s2*: *SN s2*
  shows *SN* (*lex-two s ns s2*) (**is** *SN ?r*)
⟨*proof*⟩

lemma *lex-two-compat*:
  assumes *compat1*: *ns1 O s1* ⊆ *s1*
    and *compat1'*: *s1 O ns1* ⊆ *s1*
    and *trans1*: *s1 O s1* ⊆ *s1*
    and *trans1'*: *ns1 O ns1* ⊆ *ns1*
    and *compat2*: *ns2 O s2* ⊆ *s2*
    and *ns*: (*ab1*, *ab2*) ∈ *lex-two s1 ns1 ns2*
    and *s*: (*ab2*, *ab3*) ∈ *lex-two s1 ns1 s2*
  shows (*ab1*, *ab3*) ∈ *lex-two s1 ns1 s2*
⟨*proof*⟩

lemma *lex-two-compat'*:

assumes *compat1*: *ns1 O s1* ⊆ *s1*
and *compat1'*: *s1 O ns1* ⊆ *s1*
and *trans1*: *s1 O s1* ⊆ *s1*
and *trans1'*: *ns1 O ns1* ⊆ *ns1*
and *compat2'*: *s2 O ns2* ⊆ *s2*
and *s*: (*ab1*, *ab2*) ∈ *lex-two s1 ns1 s2*
and *ns*: (*ab2*, *ab3*) ∈ *lex-two s1 ns1 ns2*
shows (*ab1*, *ab3*) ∈ *lex-two s1 ns1 s2*
⟨*proof*⟩

lemma *lex-two-compat2*:
assumes *ns1 O s1* ⊆ *s1 s1 O ns1* ⊆ *s1 s1 O s1* ⊆ *s1 ns1 O ns1* ⊆ *ns1 ns2 O s2* ⊆ *s2*
shows *lex-two s1 ns1 ns2 O lex-two s1 ns1 s2* ⊆ *lex-two s1 ns1 s2*
⟨*proof*⟩

lemma *lex-two-compat'2*:
assumes *ns1 O s1* ⊆ *s1 s1 O ns1* ⊆ *s1 s1 O s1* ⊆ *s1 ns1 O ns1* ⊆ *ns1 s2 O ns2* ⊆ *s2*
shows *lex-two s1 ns1 s2 O lex-two s1 ns1 ns2* ⊆ *lex-two s1 ns1 s2*
⟨*proof*⟩

lemma *lex-two-refl*:
assumes *r1*: *refl ns1* **and** *r2*: *refl ns2*
shows *refl* (*lex-two s1 ns1 ns2*)
⟨*proof*⟩

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*⟩

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*⟩

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., $[1] > [0,1] > [0,0,1] > \ldots$

fun *lex-ext-unbounded* :: (*'a* ⇒ *'a* ⇒ *bool* × *bool*) ⇒ *'a list* ⇒ *'a list* ⇒ *bool* × *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*) = (
((∃ *i* < *length xs*. *i* < *length ys* ∧ (∀ *j* < *i*. *snd* (*f* (*xs* ! *j*) (*ys* ! *j*))) ∧ *fst* (*f* (*xs* ! *i*) (*ys* !*i*))) ∨
(∀ *i* < *length ys*. *snd* (*f* (*xs* ! *i*) (*ys* ! *i*))) ∧ *length xs* > *length ys*),
((∃ *i* < *length xs*. *i* < *length ys* ∧ (∀ *j* < *i*. *snd* (*f* (*xs* ! *j*) (*ys* ! *j*))) ∧ *fst* (*f* (*xs* ! *i*) (*ys* !*i*))) ∨
(∀ *i* < *length ys*. *snd* (*f* (*xs* ! *i*) (*ys* ! *i*))) ∧ *length xs* ≥ *length ys*))
(**is** *?lex xs ys* = (*?stri xs ys*, *?nstri xs ys*))
⟨*proof*⟩

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* ⇒ *'a* ⇒ *bool* × *bool*) ⇒ *nat* ⇒ *'a list* ⇒ *'a list* ⇒ *bool* × *bool*
where
lex-ext f n ss ts =
(*let lts* = *length ts in*
if (*length ss* = *lts* ∨ *lts* ≤ *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* ∨ *length ys* ≤ *m*) ∧ ((∃ *i* < *length xs*. *i* < *length ys* ∧ (∀ *j* < *i*. *snd* (*f* (*xs* ! *j*) (*ys* ! *j*))) ∧ *fst* (*f* (*xs* ! *i*) (*ys* !*i*))) ∨
(∀ *i* < *length ys*. *snd* (*f* (*xs* ! *i*) (*ys* ! *i*))) ∧ *length xs* > *length ys*),
(*length xs* = *length ys* ∨ *length ys* ≤ *m*) ∧
((∃ *i* < *length xs*. *i* < *length ys* ∧ (∀ *j* < *i*. *snd* (*f* (*xs* ! *j*) (*ys* ! *j*))) ∧ *fst* (*f* (*xs* ! *i*) (*ys* !*i*))) ∨
(∀ *i* < *length ys*. *snd* (*f* (*xs* ! *i*) (*ys* ! *i*))) ∧ *length xs* ≥ *length ys*))

⟨*proof*⟩

lemma *lex-ext-to-lex-ext-unbounded*:
assumes *length xs* ≤ *n* **and** *length ys* ≤ *n*
shows *lex-ext f n xs ys* = *lex-ext-unbounded f xs ys*
⟨*proof*⟩

lemma *lex-ext-stri-imp-nstri*:
assumes *fst* (*lex-ext f m xs ys*)
shows *snd* (*lex-ext f m xs ys*)
⟨*proof*⟩

lemma *nstri-lex-ext-map*:

assumes $\bigwedge s\ t.\ s \in set\ ss \Longrightarrow t \in set\ ts \Longrightarrow fst\ (order\ s\ t) \Longrightarrow fst\ (order'\ (f\ s)\ (f\ t))$
and $\bigwedge s\ t.\ s \in set\ ss \Longrightarrow t \in set\ ts \Longrightarrow snd\ (order\ s\ t) \Longrightarrow snd\ (order'\ (f\ s)\ (f\ t))$
and $snd\ (lex\text{-}ext\ order\ n\ ss\ ts)$
shows $snd\ (lex\text{-}ext\ order'\ n\ (map\ f\ ss)\ (map\ f\ ts))$
⟨*proof*⟩

lemma *stri-lex-ext-map*:
assumes $\bigwedge s\ t.\ s \in set\ ss \Longrightarrow t \in set\ ts \Longrightarrow fst\ (order\ s\ t) \Longrightarrow fst\ (order'\ (f\ s)\ (f\ t))$
and $\bigwedge s\ t.\ s \in set\ ss \Longrightarrow t \in set\ ts \Longrightarrow snd\ (order\ s\ t) \Longrightarrow snd\ (order'\ (f\ s)\ (f\ t))$
and $fst\ (lex\text{-}ext\ order\ n\ ss\ ts)$
shows $fst\ (lex\text{-}ext\ order'\ n\ (map\ f\ ss)\ (map\ f\ ts))$
⟨*proof*⟩

lemma *lex-ext-arg-empty*: $snd\ (lex\text{-}ext\ f\ n\ []\ xs) \Longrightarrow xs = []$
⟨*proof*⟩

lemma *lex-ext-co-compat*:
assumes $\bigwedge\ s\ t.\ s \in set\ ss \Longrightarrow t \in set\ ts \Longrightarrow fst\ (order\ s\ t) \Longrightarrow snd\ (order'\ t\ s) \Longrightarrow False$
and $\bigwedge\ s\ t.\ s \in set\ ss \Longrightarrow t \in set\ ts \Longrightarrow snd\ (order\ s\ t) \Longrightarrow fst\ (order'\ t\ s) \Longrightarrow False$
and $\bigwedge\ s\ t.\ fst\ (order\ s\ t) \Longrightarrow snd\ (order\ s\ t)$
and $fst\ (lex\text{-}ext\ order\ n\ ss\ ts)$
and $snd\ (lex\text{-}ext\ order'\ n\ ts\ ss)$
shows *False*
⟨*proof*⟩

lemma *lex-ext-irrefl*: **assumes** $\bigwedge\ x.\ x \in set\ xs \Longrightarrow \neg\ fst\ (rel\ x\ x)$
shows $\neg\ fst\ (lex\text{-}ext\ rel\ n\ xs\ xs)$
⟨*proof*⟩

lemma *lex-ext-unbounded-stri-imp-nstri*:
assumes $fst\ (lex\text{-}ext\text{-}unbounded\ f\ xs\ ys)$
shows $snd\ (lex\text{-}ext\text{-}unbounded\ f\ xs\ ys)$
⟨*proof*⟩

lemma *all-nstri-imp-lex-nstri*: **assumes** $\forall\ i < length\ ys.\ snd\ (f\ (xs\ !\ i)\ (ys\ !\ i))$
and $length\ xs \geq length\ ys$ **and** $length\ xs = length\ ys \vee length\ ys \leq m$
shows $snd\ (lex\text{-}ext\ f\ m\ xs\ ys)$
⟨*proof*⟩

lemma *lex-ext-cong*[*fundef-cong*]: **fixes** $f\ g\ m1\ m2\ xs1\ xs2\ ys1\ ys2$
assumes $length\ xs1 = length\ ys1$ **and** $m1 = m2$ **and** $length\ xs2 = length\ ys2$
and $\bigwedge\ i.\ [\![i < length\ ys1;\ i < length\ ys2]\!] \Longrightarrow f\ (xs1\ !\ i)\ (xs2\ !\ i) = g\ (ys1\ !\ i)$

(*ys2* ! *i*)
shows *lex-ext f m1 xs1 xs2* = *lex-ext g m2 ys1 ys2*
⟨*proof*⟩

lemma *lex-ext-unbounded-cong*[*fundef-cong*]: **assumes** *as* = *as'* **and** *bs* = *bs'*
and ⋀ *i*. *i* < *length as'* ⟹ *i* < *length bs'* ⟹ *f* (*as'* ! *i*) (*bs'* ! *i*) = *g* (*as'* ! *i*) (*bs'* ! *i*) **shows** *lex-ext-unbounded f as bs* = *lex-ext-unbounded g as' bs'*
⟨*proof*⟩

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*: ⋀ *s t u*. ⟦*s* ∈ *set ss*; *t* ∈ *set ts*; *u* ∈ *set us*⟧ ⟹
(*snd* (*f s t*) ∧ *fst* (*f t u*) ⟶ *fst* (*f s u*)) ∧
(*fst* (*f s t*) ∧ *snd* (*f t u*) ⟶ *fst* (*f s u*)) ∧
(*snd* (*f s t*) ∧ *snd* (*f t u*) ⟶ *snd* (*f s u*)) ∧
(*fst* (*f s t*) ∧ *fst* (*f t u*) ⟶ *fst* (*f s u*))
shows
(*snd* (*lex-ext f n ss ts*) ∧ *fst* (*lex-ext f n ts us*) ⟶ *fst* (*lex-ext f n ss us*)) ∧
(*fst* (*lex-ext f n ss ts*) ∧ *snd* (*lex-ext f n ts us*) ⟶ *fst* (*lex-ext f n ss us*)) ∧
(*snd* (*lex-ext f n ss ts*) ∧ *snd* (*lex-ext f n ts us*) ⟶ *snd* (*lex-ext f n ss us*)) ∧
(*fst* (*lex-ext f n ss ts*) ∧ *fst* (*lex-ext f n ts us*) ⟶ *fst* (*lex-ext f n ss us*))

⟨*proof*⟩

lemma *lex-ext-unbounded-map*:
assumes *S*: ⋀ *i*. *i* < *length ss* ⟹ *i* < *length ts* ⟹ *fst* (*r* (*ss* ! *i*) (*ts* ! *i*)) ⟹ *fst* (*r* (*map f ss* ! *i*) (*map f ts* ! *i*))
and *NS*: ⋀ *i*. *i* < *length ss* ⟹ *i* < *length ts* ⟹ *snd* (*r* (*ss* ! *i*) (*ts* ! *i*)) ⟹ *snd* (*r* (*map f ss* ! *i*) (*map f ts* ! *i*))
shows (*fst* (*lex-ext-unbounded r ss ts*) ⟶ *fst* (*lex-ext-unbounded r* (*map f ss*) (*map f ts*))) ∧
(*snd* (*lex-ext-unbounded r ss ts*) ⟶ *snd* (*lex-ext-unbounded r* (*map f ss*) (*map f ts*)))
⟨*proof*⟩

lemma *lex-ext-unbounded-map-S*:
assumes *S*: ⋀ *i*. *i* < *length ss* ⟹ *i* < *length ts* ⟹ *fst* (*r* (*ss* ! *i*) (*ts* ! *i*)) ⟹ *fst* (*r* (*map f ss* ! *i*) (*map f ts* ! *i*))
and *NS*: ⋀ *i*. *i* < *length ss* ⟹ *i* < *length ts* ⟹ *snd* (*r* (*ss* ! *i*) (*ts* ! *i*)) ⟹ *snd* (*r* (*map f ss* ! *i*) (*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*))
⟨*proof*⟩

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

assumes *S*: ⋀ *i*. *i* < *length ss* ⟹ *i* < *length ts* ⟹ *fst* (*r* (*ss* ! *i*) (*ts* ! *i*)) ⟹ *fst* (*r* (*map f ss* ! *i*) (*map f ts* ! *i*))
and *NS*: ⋀ *i*. *i* < *length ss* ⟹ *i* < *length ts* ⟹ *snd* (*r* (*ss* ! *i*) (*ts* ! *i*)) ⟹ *snd* (*r* (*map f ss* ! *i*) (*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*))
⟨*proof*⟩

Strong normalization with local SN assumption

lemma *lex-ext-SN*:
assumes *compat*: ⋀ *x y z*. ⟦*snd* (*g x y*); *fst* (*g y z*)⟧ ⟹ *fst* (*g x z*)
shows *SN* { (*ys*, *xs*). (∀ *y* ∈ *set ys*. *SN-on* { (*s*, *t*). *fst* (*g s t*) } {*y*}) ∧ *fst* (*lex-ext g m ys xs*) }
(**is** *SN* { (*ys*, *xs*). *?cond ys xs* })
⟨*proof*⟩

Strong normalization with global SN assumption is immediate consequence.

lemma *lex-ext-SN-2*:
assumes *compat*: ⋀ *x y z*. ⟦*snd* (*g x y*); *fst* (*g y z*)⟧ ⟹ *fst* (*g x z*)
and *SN*: *SN* {(*s*, *t*). *fst* (*g s t*)}
shows *SN* { (*ys*, *xs*). *fst* (*lex-ext g m ys xs*) }
⟨*proof*⟩

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

lemma *lex-ext-least-1*: *snd* (*lex-ext f m xs* [])
⟨*proof*⟩

lemma *lex-ext-least-2*: ¬ *fst* (*lex-ext f m* [] *ys*)
⟨*proof*⟩

Preservation of totality on lists of same length.

lemma *lex-ext-unbounded-total*:
assumes ∀(*s*, *t*)∈*set* (*zip ss ts*). *s* = *t* ∨ *fst* (*f s t*) ∨ *fst* (*f t s*)
and *refl*: ⋀ *t*. *snd* (*f t t*)
and *length ss* = *length ts*
shows *ss* = *ts* ∨ *fst* (*lex-ext-unbounded f ss ts*) ∨ *fst* (*lex-ext-unbounded f ts ss*)
⟨*proof*⟩

lemma *lex-ext-total*:
assumes ∀(*s*, *t*)∈*set* (*zip ss ts*). *s* = *t* ∨ *fst* (*f s t*) ∨ *fst* (*f t s*)
and ⋀ *t*. *snd* (*f t t*)
and *len*: *length ss* = *length ts*
shows *ss* = *ts* ∨ *fst* (*lex-ext f n ss ts*) ∨ *fst* (*lex-ext f n ts ss*)
⟨*proof*⟩

Monotonicity of the lexicographic extension.

lemma *lex-ext-unbounded-mono*:

assumes $\bigwedge i.\ [\![i < \mathit{length}\ \mathit{xs};\ i < \mathit{length}\ \mathit{ys};\ \mathit{fst}\ (P\ (\mathit{xs}\ !\ i)\ (\mathit{ys}\ !\ i))]\!] \Longrightarrow \mathit{fst}\ (P'\ (\mathit{xs}\ !\ i)\ (\mathit{ys}\ !\ i))$
and $\bigwedge i.\ [\![i < \mathit{length}\ \mathit{xs};\ i < \mathit{length}\ \mathit{ys};\ \mathit{snd}\ (P\ (\mathit{xs}\ !\ i)\ (\mathit{ys}\ !\ i))]\!] \Longrightarrow \mathit{snd}\ (P'\ (\mathit{xs}\ !\ i)\ (\mathit{ys}\ !\ i))$
shows
$(\mathit{fst}\ (\mathit{lex\text{-}ext\text{-}unbounded}\ P\ \mathit{xs}\ \mathit{ys}) \longrightarrow \mathit{fst}\ (\mathit{lex\text{-}ext\text{-}unbounded}\ P'\ \mathit{xs}\ \mathit{ys})) \wedge$
$(\mathit{snd}\ (\mathit{lex\text{-}ext\text{-}unbounded}\ P\ \mathit{xs}\ \mathit{ys}) \longrightarrow \mathit{snd}\ (\mathit{lex\text{-}ext\text{-}unbounded}\ P'\ \mathit{xs}\ \mathit{ys}))$
(**is** $(?l1\ \mathit{xs}\ \mathit{ys} \longrightarrow ?r1\ \mathit{xs}\ \mathit{ys}) \wedge (?l2\ \mathit{xs}\ \mathit{ys} \longrightarrow ?r2\ \mathit{xs}\ \mathit{ys})$)
⟨*proof*⟩

lemma *lex-ext-local-mono* [*mono*]:
assumes $\bigwedge s\ t.\ s \in \mathit{set}\ \mathit{ts} \Longrightarrow t \in \mathit{set}\ \mathit{ss} \Longrightarrow \mathit{ord}\ s\ t \Longrightarrow \mathit{ord}'\ s\ t$
shows $\mathit{fst}\ (\mathit{lex\text{-}ext}\ (\lambda\ x\ y.\ (\mathit{ord}\ x\ y,\ (x,\ y) \in \mathit{ns\text{-}rel}))\ (\mathit{length}\ \mathit{ts})\ \mathit{ts}\ \mathit{ss}) \longrightarrow$
$\mathit{fst}\ (\mathit{lex\text{-}ext}\ (\lambda\ x\ y.\ (\mathit{ord}'\ x\ y,\ (x,\ y) \in \mathit{ns\text{-}rel}))\ (\mathit{length}\ \mathit{ts})\ \mathit{ts}\ \mathit{ss})$
⟨*proof*⟩

lemma *lex-ext-mono* [*mono*]:
assumes $\bigwedge s\ t.\ \mathit{ord}\ s\ t \longrightarrow \mathit{ord}'\ s\ t$
shows $\mathit{fst}\ (\mathit{lex\text{-}ext}\ (\lambda\ x\ y.\ (\mathit{ord}\ x\ y,\ (x,\ y) \in \mathit{ns}))\ (\mathit{length}\ \mathit{ts})\ \mathit{ts}\ \mathit{ss}) \longrightarrow$
$\mathit{fst}\ (\mathit{lex\text{-}ext}\ (\lambda\ x\ y.\ (\mathit{ord}'\ x\ y,\ (x,\ y) \in \mathit{ns}))\ (\mathit{length}\ \mathit{ts})\ \mathit{ts}\ \mathit{ss})$
⟨*proof*⟩

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

lemma *set-scf-list* [*simp*]:
assumes ∀ *i* < *length xs. scf i* > *0*
shows *set* (*scf-list scf xs*) = *set xs*
⟨*proof*⟩

lemma *scf-list-subset*: *set* (*scf-list scf xs*) ⊆ *set xs*
⟨*proof*⟩

lemma *scf-list-empty* [*simp*]:
scf-list scf [] = [] ⟨*proof*⟩

lemma *scf-list-bef-i-aft* [*simp*]:
scf-list scf (*bef* @ *i* # *aft*) =
scf-list scf bef @ *replicate* (*scf* (*length bef*)) *i* @
scf-list (λ *i. scf* (*Suc* (*length bef* + *i*))) *aft*
⟨*proof*⟩

lemma *scf-list-map* [*simp*]:
scf-list scf (*map f xs*) = *map f* (*scf-list scf xs*)
⟨*proof*⟩

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

fun *scf-term* :: (′*f* × *nat* ⇒ *nat* ⇒ *nat*) ⇒ (′*f*, ′*v*) *term* ⇒ (′*f*, ′*v*) *term*
where
scf-term scf (*Var x*) = (*Var x*) |
scf-term scf (*Fun f ts*) = *Fun f* (*scf-list* (*scf* (*f*, *length ts*)) (*map* (*scf-term scf*) *ts*))

lemma *vars-term-scf-subset*:
vars-term (*scf-term scf s*) ⊆ *vars-term s*
⟨*proof*⟩

lemma *scf-term-subst*:
scf-term scf (*t* · σ) = *scf-term scf t* · (λ *x. scf-term scf* (σ *x*))
⟨*proof*⟩

4.2 Weight Functions

locale *weight-fun* =
fixes *w* :: ′*f* × *nat* ⇒ *nat*
and *w0* :: *nat*
and *scf* :: ′*f* × *nat* ⇒ *nat* ⇒ *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* ⇒ *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* (λ (*ti*, *i*). *weight ti* ∗ *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*⟩

declare *weight.simps*(*2*)[*simp del*]

abbreviation *SCF* ≡ *scf-term scf*

lemma *sum-list-scf-list*:
  assumes ⋀ *i*. *i* < *length ts* ⟹ *f i* > *0*
  shows *sum-list* (*map weight ts*) ≤ *sum-list* (*map weight* (*scf-list f ts*))
  ⟨*proof*⟩

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*))
⟨*proof*⟩

end

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

lemma *mset-replicate-mono*:
assumes *m1* ⊆# *m2*
shows $\sum_\#$ (*mset* (*replicate n m1*)) ⊆# $\sum_\#$ (*mset* (*replicate n m2*))
⟨*proof*⟩

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* ⇒ *bool* **and** *scf* +
assumes *w0*: $w\ (f,\ 0) \geq w0\ w0 > 0$
and *adm*: $w\ (f,\ 1) = 0 \Longrightarrow$ *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*: $i < n \Longrightarrow$ *scf* $(f,\ n)\ i > 0$
and *pr-weak-refl* [*simp*]: *pr-weak fn fn*
and *pr-weak-trans*: *pr-weak fn gm* ⟹ *pr-weak gm hk* ⟹ *pr-weak fn hk*
and *pr-strict*: *pr-strict fn gm* ⟷ *pr-weak fn gm* ∧ ¬ *pr-weak gm fn*
and *pr-SN*: *SN* {(*fn*, *gm*). *pr-strict fn gm*}
begin

lemma *weight-w0*: *weight t* ≥ *w0*
⟨*proof*⟩

lemma *weight-gt-0*: *weight t* > *0*
⟨*proof*⟩

lemma *weight-0* [*iff*]: *weight t* = *0* ⟷ *False*
⟨*proof*⟩

lemma *not-S-Var*: ¬ *S* (*Var x*) *t*
⟨*proof*⟩

lemma *S-imp-NS*: *S s t* ⟹ *NS s t*
⟨*proof*⟩

4.4 Reflexivity and Irreflexivity

lemma *NS-refl*: *NS s s*
⟨*proof*⟩

lemma *pr-strict-irrefl*: ¬ *pr-strict fn fn*
⟨*proof*⟩

lemma *S-irrefl*: ¬ *S t t*
⟨*proof*⟩

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*⟩

lemma *S-ctxt*: *S s t* ⟹ *S* (*C*⟨*s*⟩) (*C*⟨*t*⟩)
⟨*proof*⟩

lemma *NS-mono-one*:

assumes *NS*: *NS s t* **shows** *NS* (*Fun f* (*ss1* @ *s* # *ss2*)) (*Fun f* (*ss1* @ *t* # *ss2*))
⟨*proof*⟩

lemma *NS-ctxt*: *NS s t* ⟹ *NS* (*C*⟨*s*⟩) (*C*⟨*t*⟩)
⟨*proof*⟩

4.6 The Subterm Property

lemma *NS-Var-imp-eq-least*: *NS* (*Var x*) *t* ⟹ *t* = *Var x* ∨ (∃ *f*. *t* = *Fun f* [] ∧ *least f*)
⟨*proof*⟩

lemma *kbo-supt-one*: *NS s* (*t* :: ('*f*, '*v*) *term*) ⟹ *S* (*Fun f* (*bef* @ *s* # *aft*)) *t*
⟨*proof*⟩

lemma *S-supt*:
assumes *supt*: *s* ▷ *t*
shows *S s t*
⟨*proof*⟩

lemma *NS-supteq*:
assumes *s* ⊵ *t*
shows *NS s t*
⟨*proof*⟩

4.7 Least Elements

lemma *NS-all-least*:
assumes *l*: *least f*
shows *NS t* (*Fun f* [])
⟨*proof*⟩

lemma *not-S-least*:
assumes *l*: *least f*
shows ¬ *S* (*Fun f* []) *t*
⟨*proof*⟩

lemma *NS-least-least*:
assumes *l*: *least f*
and *NS*: *NS* (*Fun f* []) *t*
shows ∃ *g*. *t* = *Fun g* [] ∧ *least g*
⟨*proof*⟩

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

lemma *weight-subst*: *weight* (*t* · σ) =
weight t + *sum-mset* (*image-mset* (λ *x*. *weight* (σ *x*) − *w0*) (*vars-term-ms* (*SCF t*)))
⟨*proof*⟩

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*⟩

lemma *weight-stable-lt*:
assumes *ws*: *weight s* $<$ *weight t*
and *vs*: *vars-term-ms* (*SCF s*) $\subseteq\#$ *vars-term-ms* (*SCF t*)
shows *weight* ($s \cdot \sigma$) $<$ *weight* ($t \cdot \sigma$)
⟨*proof*⟩

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

lemma *kbo-stable*:
fixes σ :: ('*f*, '*v*) *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*)
⟨*proof*⟩

lemma *S-subst*:
$S\ s\ t \Longrightarrow S\ (s \cdot (\sigma :: ('f, 'v)\ subst))\ (t \cdot \sigma)$
⟨*proof*⟩

lemma *NS-subst*: $NS\ s\ t \Longrightarrow NS\ (s \cdot (\sigma :: ('f, 'v)\ subst))\ (t \cdot \sigma)$ ⟨*proof*⟩

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*⟩

lemma *S-trans*: $S\ s\ t \Longrightarrow S\ t\ u \Longrightarrow S\ s\ u$ ⟨*proof*⟩
lemma *NS-trans*: $NS\ s\ t \Longrightarrow NS\ t\ u \Longrightarrow NS\ s\ u$ ⟨*proof*⟩
lemma *NS-S-compat*: $NS\ s\ t \Longrightarrow S\ t\ u \Longrightarrow S\ s\ u$ ⟨*proof*⟩
lemma *S-NS-compat*: $S\ s\ t \Longrightarrow NS\ t\ u \Longrightarrow S\ s\ u$ ⟨*proof*⟩

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*⟩

lemma *S-SN*: *SN* $\{(x, y).\ S\ x\ y\}$
⟨*proof*⟩

4.11 Ground Totality

lemma *ground-SCF* [*simp*]:
  $ground\ (SCF\ t) = ground\ t$
⟨*proof*⟩

declare *kbo.simps*[*simp del*]

lemma *ground-vars-term-ms*: $ground\ t \Longrightarrow vars\text{-}term\text{-}ms\ t = \{\#\}$
  ⟨*proof*⟩

context
  fixes $F :: ('f \times nat)\ set$
  assumes *pr-weak*: $pr\text{-}weak = pr\text{-}strict^{==}$
    and *pr-gtotal*: $\bigwedge f\ g.\ f \in F \Longrightarrow g \in F \Longrightarrow f = g \vee pr\text{-}strict\ f\ g \vee pr\text{-}strict\ g\ f$
begin

lemma *S-ground-total*:
  assumes $funas\text{-}term\ s \subseteq F$ **and** $ground\ s$ **and** $funas\text{-}term\ t \subseteq F$ **and** $ground\ t$
  shows $s = t \vee S\ s\ t \vee S\ t\ s$
  ⟨*proof*⟩
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* $\{(x,\ y).\ S\ x\ y\}\ \{(x,\ y).\ NS\ x\ y\}$
  ⟨*proof*⟩
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 Intel-*

ligence, 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.