

A Verified Efficient Implementation of the Weighted Path Order*

René Thiemann and Elias Wenninger

University of Innsbruck

February 6, 2026

Abstract

The Weighted Path Order (WPO) of Yamada is a powerful technique for proving termination [3, 4, 5]. In a previous AFP entry [2], the WPO was defined and properties of WPO have been formally verified. However, the implementation of WPO was naive, leading to an exponential runtime in the worst case.

Therefore, in this AFP entry we provide a poly-time implementation of WPO. The implementation is based on memoization. Since WPO generalizes the recursive path order (RPO) [1], we also easily derive an efficient implementation of RPO.

Contents

1	Indexed Terms	2
2	Memoized Functions on Lists	3
2.1	Congruence Rules	6
2.2	Connection to Original Functions	6
3	An Approximation of WPO	7
4	A Memoized Implementation of WPO	9
5	An Unbounded Variant of RPO	11
6	A Memoized Implementation of RPO	12

*This research was supported by the Austrian Science Fund (FWF) project I 5943.

1 Indexed Terms

We provide a method to index all subterms of a term by numbers.

theory *Indexed-Term*

imports

First-Order-Terms.Subterm-and-Context

begin

type-synonym *index* = *int*

type-synonym (*f*, *'v*) *indexed-term* = ((*f* × (*f*,*'v*)*term* × *index*), (*'v* × (*f*,*'v*)*term* × *index*)) *term*

fun *index-term-aux* :: *index* ⇒ (*f*, *'v*) *term* ⇒ *index* × (*f*, *'v*) *indexed-term*

and *index-term-aux-list* :: *index* ⇒ (*f*, *'v*) *term list* ⇒ *index* × (*f*, *'v*) *indexed-term list*

where

index-term-aux *i* (*Var* *v*) = (*i* + 1, *Var* (*v*, *Var* *v*, *i*))

| *index-term-aux* *i* (*Fun* *f* *ts*) = (*case index-term-aux-list* *i* *ts* of (*j*, *ss*) ⇒ (*j* + 1, *Fun* (*f*, *Fun* *f* *ts*, *j*) *ss*))

| *index-term-aux-list* *i* [] = (*i*, [])

| *index-term-aux-list* *i* (*t* # *ts*) = (*case index-term-aux* *i* *t* of (*j*, *s*) ⇒ *map-prod* *id* (*Cons* *s*) (*index-term-aux-list* *j* *ts*))

definition *index-term* :: (*f*, *'v*) *term* ⇒ (*f*, *'v*) *indexed-term*

where

index-term *t* = *snd* (*index-term-aux* 0 *t*)

fun *unindex* :: (*f*, *'v*) *indexed-term* ⇒ (*f*, *'v*) *term*

where

unindex (*Var* (*v*, -)) = *Var* *v*

| *unindex* (*Fun* (*f*, -) *ts*) = *Fun* *f* (*map unindex* *ts*)

fun *stored* :: (*f*, *'v*) *indexed-term* ⇒ (*f*, *'v*) *term*

where

stored (*Var* (*v*, (*s*, -))) = *s*

| *stored* (*Fun* (*f*, (*s*, -) *ts*) = *s*

fun *name-of* :: (*'a* × *'b*) ⇒ *'a*

where

name-of (*a*, -) = *a*

fun *index* :: (*f*, *'v*) *indexed-term* ⇒ *index*

where

index (*Var* (-, (-, *i*))) = *i*

| *index* (*Fun* (-, (-, *i*) -) = *i*

definition *index-term-prop* *f* *s* = (∀ *u*. *s* ⊇ *u* → *f* (*index* *u*) = *Some* (*unindex* *u*) ∧ *stored* *u* = *unindex* *u*)

lemma *index-term-aux*: **fixes** $t :: ('f, 'v)\text{term}$ **and** $ts :: ('f, 'v)\text{term list}$
shows $\text{index-term-aux } i \ t = (j, s) \implies \text{unindex } s = t \wedge i < j \wedge (\exists f. \text{dom } f = \{i \dots j\} \wedge \text{index-term-prop } f \ s)$
and $\text{index-term-aux-list } i \ ts = (j, ss) \implies \text{map unindex } ss = ts \wedge i \leq j \wedge$
 $(\exists f. \text{dom } f = \{i \dots j\} \wedge \text{Ball } (\text{set } ss) (\text{index-term-prop } f))$
 $\langle \text{proof} \rangle$

lemma *index-term-index-unindex*: $\exists f. \forall t. \text{index-term } s \supseteq t \longrightarrow f (\text{index } t) =$
 $\text{unindex } t \wedge \text{stored } t = \text{unindex } t$
 $\langle \text{proof} \rangle$

lemma *unindex-index-term[simp]*: $\text{unindex } (\text{index-term } s) = s$
 $\langle \text{proof} \rangle$

end

2 Memoized Functions on Lists

We define memoized version of lexicographic comparison of lists, multiset comparison of lists, filter on lists, etc.

theory *List-Memo-Functions*

imports

Indexed-Term

Knuth-Bendix-Order.Lexicographic-Extension

Weighted-Path-Order.Multiset-Extension2-Impl

HOL-Library.Mapping

begin

definition *valid-memory* :: $('a \Rightarrow 'b) \Rightarrow ('i \Rightarrow 'a) \Rightarrow ('i, 'b) \text{ mapping} \Rightarrow \text{bool}$

where

$\text{valid-memory } f \ \text{ind } \text{mem} = (\forall i \ b. \text{Mapping.lookup } \text{mem } i = \text{Some } b \longrightarrow f (\text{ind } i) = b)$

definition *memoize-fun* **where** $\text{memoize-fun } \text{impl } f \ g \ \text{ind } A =$

$((\forall x \ m \ p \ m'. \text{valid-memory } f \ \text{ind } m \longrightarrow \text{impl } m \ x = (p, m') \longrightarrow x \in A \longrightarrow$
 $p = f (g \ x) \wedge \text{valid-memory } f \ \text{ind } m')$

lemma *memoize-funD*: **assumes** $\text{memoize-fun } \text{impl } f \ g \ \text{ind } A$

shows $\text{valid-memory } f \ \text{ind } m \implies \text{impl } m \ x = (p, m') \implies x \in A \implies p = f (g \ x) \wedge \text{valid-memory } f \ \text{ind } m'$

$\langle \text{proof} \rangle$

lemma *memoize-funI*: **assumes** $\bigwedge m \ x \ p \ m'. \text{valid-memory } f \ \text{ind } m \implies \text{impl } m \ x = (p, m') \implies x \in A \implies p = f (g \ x) \wedge \text{valid-memory } f \ \text{ind } m'$

shows *memoize-fun impl f g ind A*
 ⟨proof⟩

lemma *memoize-fun-pairI*: **assumes** $\bigwedge m x y p m'. \text{valid-memory } f \text{ ind } m \implies \text{impl } m (x,y) = (p,m') \implies x \in A \implies y \in B \implies p = f (g x, h y) \wedge \text{valid-memory } f \text{ ind } m'$
shows *memoize-fun impl f (map-prod g h) ind (A × B)*
 ⟨proof⟩

lemma *memoize-fun-mono*: **assumes** *memoize-fun impl f g ind B*
and $A \subseteq B$
shows *memoize-fun impl f g ind A*
 ⟨proof⟩

fun *filter-mem* :: $('a \Rightarrow 'b) \Rightarrow ('m \Rightarrow 'b \Rightarrow 'c \times 'm) \Rightarrow ('c \Rightarrow \text{bool}) \Rightarrow 'm \Rightarrow 'a \text{ list} \Rightarrow ('a \text{ list} \times 'm)$
where
filter-mem pre f post mem [] = ([], mem)
 | *filter-mem pre f post mem (x # xs) = (case f mem (pre x) of (c, mem') \Rightarrow case filter-mem pre f post mem' xs of (ys, mem'') \Rightarrow (if post c then (x # ys, mem'') else (ys, mem''))*

fun *forall-mem* :: $('a \Rightarrow 'b) \Rightarrow ('m \Rightarrow 'b \Rightarrow 'c \times 'm) \Rightarrow ('c \Rightarrow \text{bool}) \Rightarrow 'm \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \times 'm$
where
forall-mem pre f post mem [] = (True, mem)
 | *forall-mem pre f post mem (x # xs) = (case f mem (pre x) of (c, mem') \Rightarrow if post c then forall-mem pre f post mem' xs else (False, mem'))*

fun *exists-mem* :: $('a \Rightarrow 'b) \Rightarrow ('m \Rightarrow 'b \Rightarrow ('c \times 'm)) \Rightarrow ('c \Rightarrow \text{bool}) \Rightarrow 'm \Rightarrow 'a \text{ list} \Rightarrow (\text{bool} \times 'm)$
where
exists-mem pre f post mem [] = (False, mem)
 | *exists-mem pre f post mem (x # xs) = (case f mem (pre x) of (c, mem') \Rightarrow if post c then (True, mem') else exists-mem pre f post mem' xs)*

type-synonym *term-rel-mem* = $(\text{index} \times \text{index}, \text{bool} \times \text{bool})$ mapping
type-synonym $'a \text{ term-rel-mem-type} = \text{term-rel-mem} \Rightarrow 'a \times 'a \Rightarrow (\text{bool} \times \text{bool}) \times \text{term-rel-mem}$

fun *lex-ext-unbounded-mem* :: $'a \text{ term-rel-mem-type} \Rightarrow \text{term-rel-mem} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow (\text{bool} \times \text{bool}) \times \text{term-rel-mem}$
where *lex-ext-unbounded-mem f mem [] [] = ((False, True), mem) |*
lex-ext-unbounded-mem f mem (- # -) [] = ((True, True), mem) |
lex-ext-unbounded-mem f mem [] (- # -) = ((False, False), mem) |
lex-ext-unbounded-mem f mem (a # as) (b # bs) = (let (sns-res, mem-new) = f mem (a,b) in (case sns-res of

```

      (True, -) ⇒ ((True, True), mem-new)
    | (False, True) ⇒ lex-ext-unbounded-mem f mem-new as bs
    | (False, False) ⇒ ((False, False), mem-new)
  )
)

```

lemma *filter-mem-len*: $filter\ mem\ pre\ f\ post\ mem\ xs = (ys, mem') \implies length\ ys \leq length\ xs$
 ⟨proof⟩

lemma *filter-mem-len2*: $(ys, mem') = filter\ mem\ mem\ pre\ f\ post\ xs \implies length\ ys \leq length\ xs$
 ⟨proof⟩

lemma *filter-mem-set*: $filter\ mem\ pre\ f\ post\ mem\ xs = (ys, mem') \implies set\ ys \subseteq set\ xs$
 ⟨proof⟩

function *mul-ext-mem* :: 'a term-rel-mem-type ⇒ term-rel-mem ⇒ 'a list ⇒ 'a list ⇒ (bool × bool) × term-rel-mem

and *mul-ext-dom-mem* :: 'a term-rel-mem-type ⇒ term-rel-mem ⇒ 'a list ⇒ 'a list ⇒ 'a ⇒ 'a list ⇒ (bool × bool) × term-rel-mem

where

```

  mul-ext-mem f mem [] [] = ((False, True), mem)
| mul-ext-mem f mem [] (v # va) = ((False, False), mem)
| mul-ext-mem f mem (v # va) [] = ((True, True), mem)
| mul-ext-mem f mem (v # va) (y # ys) = mul-ext-dom-mem f mem (v # va) []

```

y ys

```

| mul-ext-dom-mem f mem [] xs y ys = ((False, False), mem)
| mul-ext-dom-mem f mem (x # xsa) xs y ys =
  (case f mem (x,y) of (sns-res, mem-new-1) ⇒
    (case sns-res of
      (True, -) ⇒ (case
        (filter-mem (Pair x) f (λ p. ¬ fst p) mem-new-1 ys)
        of (ys-new, mem-new-2) ⇒ case
          mul-ext-mem f mem-new-2 (xsa @ xs) ys-new of (tmp-res, mem-new-3)

```

⇒

```

  if snd tmp-res
  then ((True, True), mem-new-3)
  else mul-ext-dom-mem f mem-new-3 xsa (x # xs) y ys)
| (False, True) ⇒ (case mul-ext-mem f mem-new-1 (xsa @ xs) ys of
  (sns-res-a, mem-new-2) ⇒
  if sns-res-a = (True, True) then (sns-res-a, mem-new-2) else
  case mul-ext-dom-mem f mem-new-2 xsa (x # xs) y ys of
  (sns-res-b, mem-new-3) ⇒
  (or2 sns-res-a sns-res-b, mem-new-3))
| (False, False) ⇒ mul-ext-dom-mem f mem-new-1 xsa (x # xs) y ys)
⟨proof⟩

```

termination $\langle proof \rangle$

2.1 Congruence Rules

lemma *filter-mem-cong*[*fundef-cong*]:

assumes $\bigwedge m x. x \in \text{set } xs \implies f m (\text{pre } x) = g m (\text{pre } x)$

shows $\text{filter-mem } \text{pre } f \text{ post mem } xs = \text{filter-mem } \text{pre } g \text{ post mem } xs$

$\langle proof \rangle$

lemma *forall-mem-cong*[*fundef-cong*]:

assumes $\bigwedge m x. x \in \text{set } xs \implies f m (\text{pre } x) = g m (\text{pre } x)$

shows $\text{forall-mem } \text{pre } f \text{ post mem } xs = \text{forall-mem } \text{pre } g \text{ post mem } xs$

$\langle proof \rangle$

lemma *exists-mem-cong*[*fundef-cong*]:

assumes $\bigwedge m x. x \in \text{set } xs \implies f m (\text{pre } x) = g m (\text{pre } x)$

shows $\text{exists-mem } \text{pre } f \text{ post mem } xs = \text{exists-mem } \text{pre } g \text{ post mem } xs$

$\langle proof \rangle$

lemma *lex-ext-unbounded-mem-cong*[*fundef-cong*]:

assumes $\bigwedge x y m. x \in \text{set } xs \implies y \in \text{set } ys \implies f m (x,y) = g m (x,y)$

shows $\text{lex-ext-unbounded-mem } f m xs ys = \text{lex-ext-unbounded-mem } g m xs ys$

$\langle proof \rangle$

lemma *mul-ext-mem-cong*[*fundef-cong*]:

assumes $\bigwedge x y m. x \in \text{set } xs \implies y \in \text{set } ys \implies f m (x,y) = g m (x,y)$

shows $\text{mul-ext-mem } f m xs ys = \text{mul-ext-mem } g m xs ys$

$\langle proof \rangle$

2.2 Connection to Original Functions

lemma *filter-mem*: **assumes** *valid-memory fun ind mem1*

filter-mem f fun-mem h mem1 xs = (ys, mem2)

memoize-fun fun-mem fun g ind (f ' set xs)

shows $ys = \text{filter } (\lambda y. h (\text{fun } (g (f y)))) xs \wedge \text{valid-memory fun ind mem2}$

$\langle proof \rangle$

lemma *forall-mem*: **assumes** *valid-memory fun ind m*

and *forall-mem f fun-mem h m xs = (b, m')*

and *memoize-fun fun-mem fun g ind (f ' set xs)*

shows $b = \text{Ball } (\text{set } xs) (\lambda s. h (\text{fun } (g (f s)))) \wedge \text{valid-memory fun ind m'}$

$\langle proof \rangle$

lemma *exists-mem*: **assumes** *valid-memory fun ind m*

and *exists-mem f fun-mem h m xs = (b, m')*

and *memoize-fun fun-mem fun g ind (f ' set xs)*

shows $b = \text{Bex } (\text{set } xs) (\lambda s. h (\text{fun } (g (f s)))) \wedge \text{valid-memory fun ind m'}$

$\langle proof \rangle$

lemma *lex-ext-unbounded-mem*: **assumes** $rel\text{-}pair = (\lambda(s, t). rel\ s\ t)$
shows $valid\text{-}memory\ rel\text{-}pair\ ind\ mem \implies lex\text{-}ext\text{-}unbounded\text{-}mem\ rel\text{-}mem\ mem\ xs\ ys = (p, mem')$
 $\implies memoize\text{-}fun\ rel\text{-}mem\ rel\text{-}pair\ (map\text{-}prod\ g\ h)\ ind\ (set\ xs \times set\ ys)$
 $\implies p = lex\text{-}ext\text{-}unbounded\ rel\ (map\ g\ xs)\ (map\ h\ ys) \wedge valid\text{-}memory\ rel\text{-}pair\ ind\ mem'$
 $\langle proof \rangle$

lemma *mul-ext-mem*: **assumes** $rel\text{-}pair = (\lambda(s, t). rel\ s\ t)$
shows $valid\text{-}memory\ rel\text{-}pair\ ind\ mem \implies mul\text{-}ext\text{-}mem\ rel\text{-}mem\ mem\ xs\ ys = (p, mem')$
 $\implies memoize\text{-}fun\ rel\text{-}mem\ rel\text{-}pair\ (map\text{-}prod\ g\ h)\ ind\ (set\ xs \times set\ ys)$
 $\implies p = mul\text{-}ext\text{-}impl\ rel\ (map\ g\ xs)\ (map\ h\ ys) \wedge valid\text{-}memory\ rel\text{-}pair\ ind\ mem' \text{ (is } ?A \implies ?B \implies ?C \implies ?D)$
 $\langle proof \rangle$

end

3 An Approximation of WPO

We define an approximation of WPO.

It replaces the bounded lexicographic comparison by an unbounded one. Hence, no runtime check on lengths are required anymore, but instead the arities of the inputs have to be bounded via an assumption.

Moreover, instead of checking that terms are strictly or non-strictly decreasing w.r.t. the algebra (i.e., the input reduction pair), we just demand that there are sufficient criteria to ensure a strict- or non-strict decrease.

theory *WPO-Approx*
imports *Weighted-Path-Order.WPO*
begin

definition *compare-bools* :: $bool \times bool \Rightarrow bool \times bool \Rightarrow bool$
where
 $compare\text{-}bools\ p1\ p2 \iff (fst\ p1 \longrightarrow fst\ p2) \wedge (snd\ p1 \longrightarrow snd\ p2)$

notation *compare-bools* ($\langle - / \leq_{cb} - \rangle$) [*51, 51*] *50*)

lemma *lex-ext-unbounded-cb*:
assumes $\bigwedge i. i < length\ xs \implies i < length\ ys \implies f\ (xs\ !\ i)\ (ys\ !\ i) \leq_{cb}\ g\ (xs\ !\ i)\ (ys\ !\ i)$
shows $lex\text{-}ext\text{-}unbounded\ f\ xs\ ys \leq_{cb}\ lex\text{-}ext\text{-}unbounded\ g\ xs\ ys$
 $\langle proof \rangle$

lemma *mul-ext-cb*:
assumes $\bigwedge x\ y. x \in set\ xs \implies y \in set\ ys \implies f\ x\ y \leq_{cb}\ g\ x\ y$
shows $mul\text{-}ext\ f\ xs\ ys \leq_{cb}\ mul\text{-}ext\ g\ xs\ ys$

<proof>

context

```
fixes pr :: ('f × nat ⇒ 'f × nat ⇒ bool × bool)
and prl :: 'f × nat ⇒ bool
and ssimple :: bool
and large :: 'f × nat ⇒ bool
and cS cNS :: ('f,'v)term ⇒ ('f,'v)term ⇒ bool — sufficient criteria
and σ :: 'f status
and c :: 'f × nat ⇒ order-tag
```

begin

```
fun wpo-ub :: ('f, 'v) term ⇒ ('f, 'v) term ⇒ bool × bool
```

where

```
wpo-ub s t = (if cS s t then (True, True) else if cNS s t then (case s of
  Var x ⇒ (False,
    (case t of
      Var y ⇒ x = y
    | Fun g ts ⇒ status σ (g, length ts) = [] ∧ prl (g, length ts)))
  | Fun f ss ⇒
    let ff = (f, length ss); sf = status σ ff in
    if (∃ i ∈ set sf. snd (wpo-ub (ss ! i) t)) then (True, True)
    else
      (case t of
        Var - ⇒ (False, ssimple ∧ large ff)
      | Fun g ts ⇒
        let gg = (g, length ts); sg = status σ gg in
        (case pr ff gg of (prs, prns) ⇒
          if prns ∧ (∀ j ∈ set sg. fst (wpo-ub s (ts ! j))) then
            if prs then (True, True)
          else
            let ss' = map (λ i. ss ! i) sf;
                ts' = map (λ i. ts ! i) sg;
                cf = c ff;
                cg = c gg in
            if cf = Lex ∧ cg = Lex then lex-ext-unbounded wpo-ub ss' ts'
            else if cf = Mul ∧ cg = Mul then mul-ext wpo-ub ss' ts'
            else if ts' = [] then (ss' ≠ [], True) else (False, False)
          else (False, False)))
    ) else (False, False))
```

```
declare wpo-ub.simps [simp del]
```

abbreviation $wpo\text{-orig } n \ S \ NS \equiv wpo.wpo \ n \ S \ NS \ pr \ prl \ \sigma \ c \ ssimple \ large$

soundness of approximation: *local.wpo-ub* can be simulated by *local.wpo-orig* if the arities are small (usually the length of the status of *f* is smaller than the arity of *f*).

lemma *wpo-ub*:

```

assumes  $\bigwedge si\ tj. s \supseteq si \implies t \supseteq tj \implies (cS\ si\ tj, cNS\ si\ tj) \leq_{cb} ((si, tj) \in S,$ 
 $(si, tj) \in NS)$ 
and  $\bigwedge f. f \in \text{funas-term } t \implies \text{length } (\text{status } \sigma f) \leq n$ 
shows  $wpo\text{-ub } s\ t \leq_{cb} wpo\text{-orig } n\ S\ NS\ s\ t$ 
 $\langle \text{proof} \rangle$ 

end
end

```

4 A Memoized Implementation of WPO

```

theory WPO-Mem-Impl

```

```

imports

```

```

  WPO-Approx

```

```

  Indexed-Term

```

```

  List-Memo-Functions

```

```

begin

```

```

context

```

```

  fixes  $pr :: ('f \times nat \Rightarrow 'f \times nat \Rightarrow bool \times bool)$ 

```

```

    and  $prl :: 'f \times nat \Rightarrow bool$ 

```

```

    and  $ssimple :: bool$ 

```

```

    and  $large :: 'f \times nat \Rightarrow bool$ 

```

```

    and  $cS\ cNS :: ('f, 'v)\text{term} \Rightarrow ('f, 'v)\text{term} \Rightarrow bool$ 

```

```

    and  $\sigma :: 'f\ \text{status}$ 

```

```

    and  $c :: 'f \times nat \Rightarrow \text{order-tag}$ 

```

```

begin

```

The main implementation working on indexed terms

```

fun

```

```

   $wpo\text{-mem} :: ((f, 'v)\ \text{indexed-term})\ \text{term-rel-mem-type}$  and

```

```

   $wpo\text{-main} :: ((f, 'v)\ \text{indexed-term})\ \text{term-rel-mem-type}$ 

```

```

where

```

```

   $wpo\text{-mem}\ \text{mem}\ (s, t) =$ 

```

```

    (let

```

```

       $i = \text{index } s;$ 

```

```

       $j = \text{index } t$ 

```

```

    in

```

```

      (case Mapping.lookup mem (i, j) of

```

```

        Some res  $\Rightarrow (res, mem)$ 

```

```

        | None  $\Rightarrow \text{case } wpo\text{-main}\ \text{mem}\ (s, t)$ 

```

```

        of (res, mem-new)  $\Rightarrow (res, \text{Mapping.update } (i, j)\ res\ mem\text{-new}))$ )

```

```

|  $wpo\text{-main}\ \text{mem}\ (s, t) = (\text{let } fs = \text{stored } s; ft = \text{stored } t\ \text{in}$ 

```

```

  if  $cS\ fs\ ft$  then ((True, True), mem)

```

```

  else if  $cNS\ fs\ ft$  then (

```

```

    case  $s$  of

```

```

    Var x  $\Rightarrow ((\text{False},$ 

```

```

      (case  $t$  of

```

```

        Var y  $\Rightarrow \text{name-of } x = \text{name-of } y$ 

```

```

    | Fun g ts ⇒ status σ (name-of g, length ts) = [] ∧ prl (name-of g, length
ts))), mem)
  | Fun f ss ⇒
    let ff = (name-of f, length ss); sf = status σ ff; ss' = map (λ i. ss ! i) sf in
    (case exists-mem (λ s'. (s',t)) wpo-mem snd mem ss' of
    (wpo-result, mem-out-1) ⇒
    if wpo-result then ((True, True), mem-out-1)
    else
    (case t of
    Var - ⇒ ((False, ssimple ∧ large ff), mem-out-1)
    | Fun g ts ⇒
    let gg = (name-of g, length ts); sg = status σ gg; ts' = map (λ i. ts !
i) sg in
    (case pr ff gg of (prs, prns) ⇒
    if prns then
    (case forall-mem (λ t'. (s,t')) wpo-mem fst mem-out-1 ts' of
    (wpo-result, mem-out-2) ⇒
    if wpo-result then
    if prs then ((True, True), mem-out-2)
    else
    let cf = c ff; cg = c gg in
    if cf = Lex ∧ cg = Lex then lex-ext-unbounded-mem wpo-mem
mem-out-2 ss' ts'
    else if cf = Mul ∧ cg = Mul then mul-ext-mem wpo-mem
mem-out-2 ss' ts'
    else if ts' = [] then ((ss' ≠ [], True), mem-out-2)
    else ((False, False), mem-out-2)
    else ((False, False), mem-out-2)) else ((False,False), mem-out-1))
    )
    ) else ((False, False), mem))

```

declare *wpo-mem.simps*[simp del]
declare *wpo-main.simps*[simp del]

And the wrapper that computes the indexed terms and initializes the memory.

definition *wpo-mem-impl* :: ('f, 'v) term ⇒ ('f, 'v) term ⇒ (bool × bool)

where

wpo-mem-impl s t = fst (wpo-mem Mapping.empty (index-term s, index-term t))

Soundness of the implementation

lemma *wpo-mem*: **fixes** rli rri :: index ⇒ ('f,'v)term

assumes

wpoub: *wpoub* = *wpo-ub* pr prl ssimple large cS cNS σ c

and *wpo*: *wpo* = (λ (s,t). *wpoub* s t)

and *ri*: *ri* = map-prod rli rri

and ∧ *si*. fst st ⊇ si ⇒ rli (index si) = unindex si ∧ stored si = unindex si

```

and  $\wedge$  ti. snd st  $\succeq$  ti  $\implies$  rri (index ti) = unindex ti  $\wedge$  stored ti = unindex ti
and valid-memory wpo ri m
shows wpo-mem m st = (p,m')  $\implies$  p = wpo (map-prod unindex unindex st)  $\wedge$ 
valid-memory wpo ri m'
      wpo-main m st = (p,m')  $\implies$  p = wpo (map-prod unindex unindex st)  $\wedge$ 
valid-memory wpo ri m'
      <proof>

```

```

lemma wpo-ub-memoized-code[code]:
  wpo-ub pr prl ssimple large cS cNS  $\sigma$  c s t = wpo-mem-impl s t
  <proof>
end
end

```

5 An Unbounded Variant of RPO

We define an unbounded version of RPO in the sense that lexicographic comparisons do not require a length check. This unbounded version of RPO is equivalent to the original RPO provided that the arities of the function symbols are below the bound that is used for lexicographic comparisons.

```

theory RPO-Unbounded

```

```

  imports

```

```

    Weighted-Path-Order.RPO

```

```

begin

```

```

fun rpo-unbounded :: (f  $\times$  nat  $\Rightarrow$  f  $\times$  nat  $\Rightarrow$  bool  $\times$  bool)  $\times$  (f  $\times$  nat  $\Rightarrow$  bool)
   $\Rightarrow$  (f  $\times$  nat  $\Rightarrow$  order-tag)  $\Rightarrow$  (f,v)term  $\Rightarrow$  (f,v)term  $\Rightarrow$  bool  $\times$  bool where
  rpo-unbounded - - (Var x) (Var y) = (False, x = y)
| rpo-unbounded pr - (Var x) (Fun g ts) = (False, ts = []  $\wedge$  snd pr (g,0))
| rpo-unbounded pr c (Fun f ss) (Var y) =
  (let con =  $\exists s \in$  set ss. snd (rpo-unbounded pr c s (Var y)) in (con,con))
| rpo-unbounded pr c (Fun f ss) (Fun g ts) = (
  if  $\exists s \in$  set ss. snd (rpo-unbounded pr c s (Fun g ts))
  then (True, True)
  else (case (fst pr) (f,length ss) (g,length ts) of (prs,prns)  $\Rightarrow$ 
    if prns  $\wedge$  ( $\forall t \in$  set ts. fst (rpo-unbounded pr c (Fun f ss) t))
    then if prs
      then (True, True)
      else if c (f,length ss) = c (g,length ts)
        then if c (f,length ss) = Mul
          then mul-ext (rpo-unbounded pr c) ss ts
          else lex-ext-unbounded (rpo-unbounded pr c) ss ts
          else (length ss  $\neq$  0  $\wedge$  length ts = 0, length ts = 0)
        else (False, False)))

```

```

lemma rpo-to-rpo-unbounded:

```

```

  assumes  $\forall f i$ . (f, i)  $\in$  funas-term s  $\cup$  funas-term t  $\longrightarrow$  i  $\leq$  n (is ?b s t)

```

```

  shows rpo pr prl c n s t = rpo-unbounded (pr,prl) c s t (is ?e s t)

```

<proof>

end

6 A Memoized Implementation of RPO

We derive a memoized RPO implementation from the memoized WPO implementation

theory *RPO-Mem-Impl*

imports

RPO-Unbounded

WPO-Mem-Impl

begin

definition *rpo-mem* :: $('f \times nat \Rightarrow 'f \times nat \Rightarrow bool \times bool) \times ('f \times nat \Rightarrow bool) \Rightarrow ('f \times nat \Rightarrow order-tag) \Rightarrow -$ **where**
[*code del*]: *rpo-mem pr c mem st* =
wpo-mem (fst pr) (snd pr) False ($\lambda - . False$) ($\lambda - . False$) ($\lambda - . True$) *full-status*
c mem st

definition *rpo-main* :: $('f \times nat \Rightarrow 'f \times nat \Rightarrow bool \times bool) \times ('f \times nat \Rightarrow bool) \Rightarrow ('f \times nat \Rightarrow order-tag) \Rightarrow -$ **where**
[*code del*]: *rpo-main pr c mem st* =
wpo-main (fst pr) (snd pr) False ($\lambda - . False$) ($\lambda - . False$) ($\lambda - . True$) *full-status*
c mem st

lemma *rpo-mem-code*[*code*]: *rpo-mem pr c mem (s,t)* =
(*let*
 i = index s;
 j = index t
in
 (*case Mapping.lookup mem (i,j) of*
 Some res \Rightarrow (*res, mem*)
 | *None* \Rightarrow *case rpo-main pr c mem (s,t)*
 of (res, mem-new) \Rightarrow (res, Mapping.update (i,j) res mem-new)))
<proof>

lemma *rpo-main-code*[*code*]: *rpo-main pr c mem (s,t)* = (*case s of*
 Var x \Rightarrow (*False,*
 (*case t of*
 Var y \Rightarrow *name-of x = name-of y*
 | *Fun g ts* \Rightarrow *ts = [] \wedge snd pr (name-of g, 0)*), *mem*)
 | *Fun f ss* \Rightarrow
 let ff = (name-of f, length ss) in
 (*case exists-mem* ($\lambda s'. (s',t)$) (*rpo-mem pr c*) *snd mem ss of*
 (*sub-result, mem-out-1*) \Rightarrow
 if sub-result then ((True, True), mem-out-1))

```

else
  (case t of
    Var - => ((False, False), mem-out-1)
  | Fun g ts =>
    let gg = (name-of g, length ts) in
    (case fst pr ff gg of (prs, prns) =>
      if prns then
        (case forall-mem ( $\lambda t'. (s, t')$ ) (rpo-mem pr c) fst mem-out-1 ts of
          (sub-result, mem-out-2) =>
            if sub-result then
              if prs then ((True, True), mem-out-2)
            else
              let cf = c ff; cg = c gg in
              if cf = Lex  $\wedge$  cg = Lex then lex-ext-unbounded-mem (rpo-mem
pr c) mem-out-2 ss ts
            else if cf = Mul  $\wedge$  cg = Mul then mul-ext-mem (rpo-mem pr
c) mem-out-2 ss ts
            else if ts = [] then ((ss  $\neq$  []), True), mem-out-2)
          else ((False, False), mem-out-2)
        else ((False, False), mem-out-2)) else ((False, False), mem-out-1))
    )
  )
)
<proof>

```

lemma *rpo-unbounded-memoized-code*[code]: *rpo-unbounded pr c s t = fst (rpo-mem pr c Mapping.empty (index-term s, index-term t))*
 <proof>

end

References

- [1] N. Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1/2):69–116, 1987.
- [2] C. Sternagel, R. Thiemann, and A. Yamada. A formalization of weighted path orders and recursive path orders. *Archive of Formal Proofs*, September 2021. https://isa-afp.org/entries/Weighted_Path_Order.html, Formal proof development.
- [3] R. Thiemann, J. Schöpf, C. Sternagel, and A. Yamada. Certifying the weighted path order (invited talk). In Z. M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29–July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPICs*, pages 4:1–4:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

- [4] A. Yamada, K. Kusakari, and T. Sakabe. Unifying the Knuth-Bendix, recursive path and polynomial orders. In R. Peña and T. Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 181–192. ACM, 2013.
- [5] A. Yamada, K. Kusakari, and T. Sakabe. A unified ordering for termination proving. *Sci. Comput. Program.*, 111:110–134, 2015.