A Verified Efficient Implementation of the Weighted Path Order[*]

René Thiemann and Elias Wenninger

University of Innsbruck

September 13, 2023

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

---

[*] 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) in-
dexed-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*)*term* **and** *ts* :: (*'f*,*'v*)*term list*
shows *index-term-aux i t* = (*j*,*s*) ⟹ *unindex s* = *t* ∧ *i* < *j* ∧ (∃ *f*. *dom f* = {*i* ..< *j*} ∧ *index-term-prop f s*)
and *index-term-aux-list i ts* = (*j*,*ss*) ⟹ *map unindex ss* = *ts* ∧ *i* ≤ *j* ∧
(∃ *f*. *dom f* = {*i* ..< *j*} ∧ *Ball* (*set ss*) (*index-term-prop f*))
⟨*proof*⟩

lemma *index-term-index-unindex*: ∃ *f*. ∀ *t*. *index-term s* ⊵ *t* ⟶ *f* (*index t*) = *unindex t* ∧ *stored t* = *unindex t*
⟨*proof*⟩

lemma *unindex-index-term*[*simp*]: *unindex* (*index-term s*) = *s*
⟨*proof*⟩

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* ⇒ *'b*) ⇒ (*'i* ⇒ *'a*) ⇒ (*'i*, *'b*) *mapping* ⇒ *bool*
where
valid-memory f ind mem = (∀ *i b*. *Mapping.lookup mem i* = *Some b* ⟶ *f* (*ind i*) = *b*)

definition *memoize-fun* **where** *memoize-fun impl f g ind A* =
((∀ *x m p m'*. *valid-memory f ind m* ⟶ *impl m x* = (*p*,*m'*) ⟶ *x* ∈ *A* ⟶
p = *f* (*g x*) ∧ *valid-memory f ind m'*))

lemma *memoize-funD*: **assumes** *memoize-fun impl f g ind A*
shows *valid-memory f ind m* ⟹ *impl m x* = (*p*,*m'*) ⟹ *x* ∈ *A* ⟹ *p* = *f* (*g x*) ∧ *valid-memory f ind m'*
⟨*proof*⟩

lemma *memoize-funI*: **assumes** ⋀ *m x p m'*. *valid-memory f ind m* ⟹ *impl m x* = (*p*,*m'*) ⟹ *x* ∈ *A* ⟹ *p* = *f* (*g x*) ∧ *valid-memory f ind m'*

shows *memoize-fun impl f g ind A*
$\langle$*proof*$\rangle$

lemma *memoize-fun-pairI*: **assumes** $\bigwedge$ *m x y p m'*. *valid-memory f ind m* $\Longrightarrow$ *impl m* (*x*,*y*) = (*p*,*m'*) $\Longrightarrow$ *x* $\in$ *A* $\Longrightarrow$ *y* $\in$ *B* $\Longrightarrow$ *p* = *f* (*g x*, *h y*) $\wedge$ *valid-memory f ind m'*
shows *memoize-fun impl f* (*map-prod g h*) *ind* (*A* $\times$ *B*)
$\langle$*proof*$\rangle$

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

fun *filter-mem* :: ('*a* $\Rightarrow$ '*b*) $\Rightarrow$ ('*m* $\Rightarrow$ '*b* $\Rightarrow$ '*c* $\times$ '*m*) $\Rightarrow$ ('*c* $\Rightarrow$ *bool*) $\Rightarrow$ '*m* $\Rightarrow$ '*a list* $\Rightarrow$ ('*a 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* $\Rightarrow$ *bool*) $\Rightarrow$ '*m* $\Rightarrow$ '*a list* $\Rightarrow$ *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* $\Rightarrow$ *bool*) $\Rightarrow$ '*m* $\Rightarrow$ '*a list* $\Rightarrow$ (*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* = (*index* $\times$ *index*, *bool* $\times$ *bool*) *mapping*
type-synonym '*a term-rel-mem-type* = *term-rel-mem* $\Rightarrow$ '*a* $\times$ '*a* $\Rightarrow$ (*bool* $\times$ *bool*) $\times$ *term-rel-mem*

fun *lex-ext-unbounded-mem* :: '*a term-rel-mem-type* $\Rightarrow$ *term-rel-mem* $\Rightarrow$ '*a list* $\Rightarrow$ '*a list* $\Rightarrow$ (*bool* $\times$ *bool*) $\times$ *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′) ⟹ length ys ≤ length xs
⟨proof⟩

lemma filter-mem-len2: (ys,mem′) = filter-mem mem pre f post xs ⟹ length ys ≤ length xs
⟨proof⟩

lemma filter-mem-set: filter-mem pre f post mem xs = (ys,mem′) ⟹ set ys ⊆ 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) ⇒ 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 ⟨proof⟩

2.1 Congruence Rules

lemma *filter-mem-cong*[*fundef-cong*]:
assumes $\bigwedge$ *m x. x* $\in$ *set xs* $\Longrightarrow$ *f m* (*pre x*) = *g m* (*pre x*)
shows *filter-mem pre f post mem xs* = *filter-mem pre g post mem xs*
⟨*proof*⟩

lemma *forall-mem-cong*[*fundef-cong*]:
assumes $\bigwedge$ *m x. x* $\in$ *set xs* $\Longrightarrow$ *f m* (*pre x*) = *g m* (*pre x*)
shows *forall-mem pre f post mem xs* = *forall-mem pre g post mem xs*
⟨*proof*⟩

lemma *exists-mem-cong*[*fundef-cong*]:
assumes $\bigwedge$ *m x. x* $\in$ *set xs* $\Longrightarrow$ *f m* (*pre x*) = *g m* (*pre x*)
shows *exists-mem pre f post mem xs* = *exists-mem pre g post mem xs*
⟨*proof*⟩

lemma *lex-ext-unbounded-mem-cong*[*fundef-cong*]:
assumes $\bigwedge$*x y m. x* $\in$ *set xs* $\Longrightarrow$ *y* $\in$ *set ys* $\Longrightarrow$ *f m* (*x*,*y*) = *g m* (*x*,*y*)
shows *lex-ext-unbounded-mem f m xs ys* = *lex-ext-unbounded-mem g m xs ys*
⟨*proof*⟩

lemma *mul-ext-mem-cong*[*fundef-cong*]:
assumes $\bigwedge$*x y m. x* $\in$ *set xs* $\Longrightarrow$ *y* $\in$ *set ys* $\Longrightarrow$ *f m* (*x*,*y*) = *g m* (*x*,*y*)
shows *mul-ext-mem f m xs ys* = *mul-ext-mem g m xs ys*
⟨*proof*⟩

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* = *filter* (λ*y. h* (*fun* (*g* (*f y*)))) *xs* $\wedge$ *valid-memory fun ind mem2*
⟨*proof*⟩

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* = *Ball* (*set xs*) (λ*s. h* (*fun* (*g* (*f s*)))) $\wedge$ *valid-memory fun ind m*′
⟨*proof*⟩

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* = *Bex* (*set xs*) (λ*s. h* (*fun* (*g* (*f s*)))) $\wedge$ *valid-memory fun ind m*′
⟨*proof*⟩

lemma *lex-ext-unbounded-mem*: **assumes** *rel-pair* = (λ(*s*, *t*). *rel s t*)
shows *valid-memory rel-pair ind mem* $\Longrightarrow$ *lex-ext-unbounded-mem rel-mem mem*

xs ys = (*p*, *mem′*)
  ⟹ *memoize-fun rel-mem rel-pair* (*map-prod g h*) *ind* (*set xs* × *set ys*)
  ⟹ *p* = *lex-ext-unbounded rel* (*map g xs*) (*map h ys*) ∧ *valid-memory rel-pair ind mem′*
⟨*proof*⟩

lemma *mul-ext-mem*: **assumes** *rel-pair* = (λ(*s*, *t*). *rel s t*)
  shows *valid-memory rel-pair ind mem* ⟹ *mul-ext-mem rel-mem mem xs ys* = (*p*, *mem′*)
  ⟹ *memoize-fun rel-mem rel-pair* (*map-prod g h*) *ind* (*set xs* × *set ys*)
  ⟹ *p* = *mul-ext-impl rel* (*map g xs*) (*map h ys*) ∧ *valid-memory rel-pair ind mem′* (**is** *?A* ⟹ *?B* ⟹ *?C* ⟹ *?D*)
⟨*proof*⟩

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 lenghts 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* × *bool* ⇒ *bool* × *bool* ⇒ *bool*
  where
    compare-bools p1 p2 ⟷ (*fst p1* ⟶ *fst p2*) ∧ (*snd p1* ⟶ *snd p2*)

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

lemma *lex-ext-unbounded-cb*:
  assumes ⋀ *i*. *i* < *length xs* ⟹ *i* < *length ys* ⟹ *f* (*xs* ! *i*) (*ys* ! *i*) $\le_{cb}$ *g* (*xs* ! *i*) (*ys* ! *i*)
  shows *lex-ext-unbounded f xs ys* $\le_{cb}$ *lex-ext-unbounded g xs ys*
  ⟨*proof*⟩

lemma *mul-ext-cb*:
  assumes ⋀ *x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set ys* ⟹ *f x y* $\le_{cb}$ *g x y*
  shows *mul-ext f xs ys* $\le_{cb}$ *mul-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-orig n S NS ≡ wpo.wpo n S NS pr prl σ 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 ⋀ si tj. s ⊵ si ⟹ t ⊵ tj ⟹ (cS si tj, cNS si tj) ≤cb ((si, tj) ∈ S,
(si, tj) ∈ NS)
```

```
    and ⋀ f. f ∈ funas-term t ⟹ length (status σ f) ≤ n
  shows wpo-ub s t ≤_cb wpo-orig n S NS s t
  ⟨proof⟩

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 × 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
    and σ :: 'f status
    and c :: 'f × nat ⇒ order-tag
begin
```

The main implementation working on indexed terms

```
fun
  wpo-mem :: (('f, 'v) indexed-term) term-rel-mem-type and
  wpo-main :: (('f, 'v) indexed-term) term-rel-mem-type
  where
    wpo-mem mem (s,t) =
      (let
        i = index s;
        j = index t
      in
        (case Mapping.lookup mem (i,j) of
          Some res ⇒ (res, mem)
        | None ⇒ case wpo-main mem (s,t)
     of (res, mem-new) ⇒ (res, Mapping.update (i,j) res mem-new)))
  | wpo-main mem (s,t) = (let fs = stored s; ft = stored t in
      if cS fs ft then ((True, True), mem)
      else if cNS fs ft then (
      case s of
      Var x ⇒ ((False,
        (case t of
          Var y ⇒ name-of x = 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 ⋀ ti. snd st ⊵ ti ⟹ rri (index ti) = unindex ti ∧ stored ti = unindex ti
    and valid-memory wpo ri m
```

```
  shows wpo-mem m st = (p,m′) ⟹ p = wpo (map-prod unindex unindex st) ∧
valid-memory wpo ri m′
     wpo-main m st = (p,m′) ⟹ p = wpo (map-prod unindex unindex st) ∧
valid-memory wpo ri m′
  ⟨proof⟩

declare [[code drop: wpo-ub]]

lemma wpo-ub-memoized-code[code]:
  wpo-ub pr prl ssimple large cS cNS σ 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 × nat ⇒ 'f × nat ⇒ bool × bool) × ('f × nat ⇒ bool)
  ⇒ ('f × nat ⇒ order-tag) ⇒ ('f,'v)term ⇒ ('f,'v)term ⇒ bool × bool where
  rpo-unbounded - - (Var x) (Var y) = (False, x = y)
| rpo-unbounded pr - (Var x) (Fun g ts) = (False, ts = [] ∧ snd pr (g,0))
| rpo-unbounded pr c (Fun f ss) (Var y) =
    (let con = ∃ s ∈ set ss. snd (rpo-unbounded pr c s (Var y)) in (con,con))
| rpo-unbounded pr c (Fun f ss) (Fun g ts) = (
    if ∃ s ∈ 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) ⇒
         if prns ∧ (∀ t ∈ 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 ≠ 0 ∧ length ts = 0, length ts = 0)
         else (False,False)))

lemma rpo-to-rpo-unbounded:
  assumes ∀ f i. (f, i) ∈ funas-term s ∪ funas-term t ⟶ i ≤ 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 × nat ⇒ 'f × nat ⇒ bool × bool) × ('f × nat ⇒ bool)
  ⇒ ('f × nat ⇒ order-tag) ⇒ - where
  [code del]: rpo-mem pr c mem st =
  wpo-mem (fst pr) (snd pr) False (λ -. False) (λ - -. False) (λ - -. True) full-status
c mem st
```

```
definition rpo-main :: ('f × nat ⇒ 'f × nat ⇒ bool × bool) × ('f × nat ⇒ bool)
  ⇒ ('f × nat ⇒ order-tag) ⇒ - where
  [code del]: rpo-main pr c mem st =
  wpo-main (fst pr) (snd pr) False (λ -. False) (λ - -. False) (λ - -. 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 ⇒ (res, mem)
        | None ⇒ case rpo-main pr c mem (s,t)
     of (res, mem-new) ⇒ (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 ⇒ ((False,
        (case t of
          Var y ⇒ name-of x = name-of y
        | Fun g ts ⇒ ts = [] ∧ snd pr (name-of g, 0))), mem)
    | Fun f ss ⇒
        let ff = (name-of f, length ss) in
          (case exists-mem (λ s'. (s',t)) (rpo-mem pr c) snd mem ss of
         (sub-result, mem-out-1) ⇒
            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 (λ 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 ∧ cg = Lex then lex-ext-unbounded-mem (rpo-mem
pr c) mem-out-2 ss ts
                 else if cf = Mul ∧ cg = Mul then mul-ext-mem (rpo-mem pr
c) 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))
          )
        )
      )
  ⟨proof⟩
```

declare [[*code drop*: *rpo-unbounded*]]

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.