

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

1	Indexed Terms	2
2	Memoized Functions on Lists	5
2.1	Congruence Rules	7
2.2	Connection to Original Functions	9
3	An Approximation of WPO	14
4	A Memoized Implementation of WPO	19
5	An Unbounded Variant of RPO	26
6	A Memoized Implementation of RPO	29

*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 $\text{index-term-aux-list } i \ ts = (j, ss) \implies \text{map unindex } ss = ts \wedge i \leq j \wedge (\exists f. \text{dom } f = \{i \dots j\} \wedge \text{Ball } (\text{set } ss) (\text{index-term-prop } f))$
proof (*induct* $i \ t$ **and** $i \ ts$ *arbitrary*: $j \ s$ **and** $j \ ss$ *rule*: *index-term-aux-index-term-aux-list.induct*)
case ($1 \ i \ v$)
then show $?case \text{ by } (\text{auto intro!} : \text{exI}[\text{of } - (\lambda -. \text{None})](i := \text{Some } (\text{Var } v))) \text{ split}$:
if-splits simp: *index-term-prop-def supteq-var-imp-eq*
next
case ($2 \ i \ g \ ts \ j \ s$)
obtain $k \ ss$ **where** $\text{rec} : \text{index-term-aux-list } i \ ts = (k, ss)$ **by force**
from $2(2)[\text{unfolded } \text{index-term-aux.simps } \text{rec } \text{split}]$
have $j : j = k + 1$ **and** $s : s = \text{Fun } (g, \text{Fun } g \ ts, k) \ ss$ **by auto**
from $2(1)[\text{OF } \text{rec}]$ **obtain** f **where** $\text{fss} : \text{map unindex } ss = ts$ **and**
 $ik : i \leq k$ **and** $f : \text{dom } f = \{i \dots k\} \wedge s. s \in \text{set } ss \implies \text{index-term-prop } f \ s$
by auto
have $\text{set} : \{i \dots k + 1\} = \text{insert } k \ \{i \dots k\}$ **using** ik **by auto**
define h **where** $h = f(k := \text{Some } (\text{Fun } g \ ts))$
show $?case \text{ unfolding } s \ \text{unindex.simps } \text{fss } j \ \text{set } \text{index-term-prop-def}$
proof (*intro conjI exI[of - h] refl allI*)
show $i < k + 1$ **using** ik **by simp**
show $\text{dom } h = \text{insert } k \ \{i \dots k\}$ **using** $ik \ f(1)$ **unfolding** $h\text{-def}$ **by auto**
fix u
show $\text{Fun } (g, \text{Fun } g \ ts, k) \ ss \supseteq u \longrightarrow h (\text{index } u) = \text{Some } (\text{unindex } u) \wedge \text{stored } u = \text{unindex } u$
proof (*cases* $u = \text{Fun } (g, \text{Fun } g \ ts, k) \ ss$)
case *True*
thus $?thesis$ **by** (*auto simp*: *fss h-def index-term-prop-def*)
next
case *False*
show $?thesis$
proof (*intro impI*)
assume $\text{Fun } (g, \text{Fun } g \ ts, k) \ ss \supseteq u$
with *False* **obtain** si **where** $si \in \text{set } ss$ **and** $si \supseteq u$
by (*metis Fun-supt suptI*)
from $f(2)[\text{unfolded } \text{index-term-prop-def}, \text{rule-format}, \text{OF this}] \ f(1) \ ik$
show $h (\text{index } u) = \text{Some } (\text{unindex } u) \wedge \text{stored } u = \text{unindex } u$ **unfolding**
 $h\text{-def}$ **by auto**
qed
qed
qed
next
case ($4 \ i \ t \ ts \ j \ sss$)
obtain $k \ s$ **where** $\text{rec1} : \text{index-term-aux } i \ t = (k, s)$ **by force**
with $4(3)$ **obtain** ss **where** $\text{rec2} : \text{index-term-aux-list } k \ ts = (j, ss)$ **and** $sss : sss = s \# ss$
by (*cases* *index-term-aux-list* $k \ ts$, *auto*)

```

from 4(1)[OF rec1] obtain f where fs: unindex s = t and ik: i < k and f: dom
f = {i..<k}
  index-term-prop f s by auto
from 4(2)[unfolded rec1, OF refl rec2] obtain g where fss: map unindex ss =
ts and kj: k ≤ j
  and g: dom g = {k..<j} ∧ si. si ∈ set ss ⇒ index-term-prop g si
  by auto
define h where h = (λ n. if n ∈ {i..<k} then f n else g n)
show ?case unfolding sss list.simps fs fss
proof (intro conjI exI[of - h] refl allI ballI)
  have dom h = {i ..<k} ∪ {k ..<j} unfolding h-def using f(1) g(1) by force
  also have ... = {i ..<j} using ik kj by auto
  finally show dom h = {i..<j} by auto
  show i ≤ j using ik kj by auto
  fix si
  assume si: si ∈ insert s (set ss)
  show index-term-prop h si
  proof (cases si = s)
    case True
      from f show ?thesis unfolding True h-def index-term-prop-def by auto
    next
      case False
      with si have si: si ∈ set ss by auto
      have disj: {i..<k} ∩ {k..<j} = {} by auto
      from g(1) g(2)[OF si]
      show ?thesis unfolding index-term-prop-def h-def using disj
      by (metis disjoint-iff domI)
  qed
qed
qed auto

```

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

```

proof –
  obtain t i where aux: index-term-aux 0 s = (i,t) by force
  from index-term-aux(1)[OF this] show ?thesis unfolding index-term-def aux
index-term-prop-def by force
qed

```

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

```

proof –
  obtain t i where aux: index-term-aux 0 s = (i,t) by force
  from index-term-aux(1)[OF this] show ?thesis unfolding index-term-def aux by
force
qed

```

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'

using *assms* **unfolding** *memoize-fun-def* **by** *auto*

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

shows *memoize-fun impl f g ind A*

using *assms* **unfolding** *memoize-fun-def* **by** *auto*

lemma *memoize-fun-pairI*: **assumes** $\bigwedge m x y p m'. \text{valid-memory } f \text{ ind } m \Rightarrow \text{impl } m (x, y) = (p, m') \Rightarrow x \in A \Rightarrow y \in B \Rightarrow p = f (g x, h y) \wedge \text{valid-memory } f \text{ ind } m'$

shows *memoize-fun impl f (map-prod g h) ind (A × B)*

using *assms* **unfolding** *memoize-fun-def* **by** *auto*

lemma *memoize-fun-mono*: **assumes** *memoize-fun impl f g ind B*

and $A \subseteq B$

shows *memoize-fun impl f g ind A*

using *assms* **unfolding** *memoize-fun-def* **by** *blast*

fun *filter-mem* :: ('a ⇒ 'b) ⇒ ('m ⇒ 'b ⇒ 'c × 'm) ⇒ ('c ⇒ bool) ⇒ 'm ⇒ 'a list ⇒ ('a list × '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') ⇒ case filter-mem pre f post mem' xs of
    (ys, mem'') ⇒ (if post c then (x # ys, mem'') else (ys, mem'')))

fun forall-mem :: ('a ⇒ 'b) ⇒ ('m ⇒ 'b ⇒ 'c × 'm) ⇒ ('c ⇒ bool) ⇒ 'm ⇒ 'a
list ⇒ bool × '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')
  ⇒ if post c then forall-mem pre f post mem' xs else (False, mem'))

fun exists-mem :: ('a ⇒ 'b) ⇒ ('m ⇒ 'b ⇒ ('c × 'm)) ⇒ ('c ⇒ bool) ⇒ 'm ⇒ 'a
list ⇒ (bool × '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')
  ⇒ if post c then (True, mem') else exists-mem pre f post mem' xs)

type-synonym term-rel-mem = (index × index, bool × bool) mapping
type-synonym 'a term-rel-mem-type = term-rel-mem ⇒ 'a × 'a ⇒ (bool × bool)
× term-rel-mem

fun lex-ext-unbounded-mem :: 'a term-rel-mem-type ⇒ term-rel-mem ⇒ 'a list ⇒
'a list ⇒ (bool × bool) × 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
by (induction xs arbitrary: mem ys mem'; force split: prod.splits if-splits)

lemma filter-mem-len2: (ys, mem') = filter-mem mem pre f post xs ⇒ length ys
≤ length xs
using filter-mem-len[of mem pre f post xs ys mem'] by auto

lemma filter-mem-set: filter-mem pre f post mem xs = (ys, mem') ⇒ set ys ⊆ set
xs
by (induction xs arbitrary: mem ys mem', auto split: prod.splits if-splits) blast

function mul-ext-mem :: 'a term-rel-mem-type ⇒ term-rel-mem ⇒ 'a list ⇒ 'a

```

$list \Rightarrow (bool \times bool) \times term\text{-}rel\text{-}mem$
and $mul\text{-}ext\text{-}dom\text{-}mem :: 'a\ term\text{-}rel\text{-}mem\text{-}type \Rightarrow term\text{-}rel\text{-}mem \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow (bool \times bool) \times term\text{-}rel\text{-}mem$
where
 $mul\text{-}ext\text{-}mem\ f\ mem\ []\ [] = ((False, True), mem)$
 $| mul\text{-}ext\text{-}mem\ f\ mem\ []\ (v \# va) = ((False, False), mem)$
 $| mul\text{-}ext\text{-}mem\ f\ mem\ (v \# va)\ [] = ((True, True), mem)$
 $| mul\text{-}ext\text{-}mem\ f\ mem\ (v \# va)\ (y \# ys) = mul\text{-}ext\text{-}dom\text{-}mem\ f\ mem\ (v \# va)\ []\ y\ ys$
 $| mul\text{-}ext\text{-}dom\text{-}mem\ f\ mem\ []\ xs\ y\ ys = ((False, False), mem)$
 $| mul\text{-}ext\text{-}dom\text{-}mem\ f\ mem\ (x \# xsa)\ xs\ y\ ys =$
 $(case\ f\ mem\ (x,y)\ of\ (sns\text{-}res, mem\text{-}new\text{-}1) \Rightarrow$
 $(case\ sns\text{-}res\ of$
 $(True, -) \Rightarrow (case$
 $(filter\text{-}mem\ (Pair\ x)\ f\ (\lambda\ p.\ \neg\ fst\ p)\ mem\text{-}new\text{-}1\ ys)$
 $of\ (ys\text{-}new, mem\text{-}new\text{-}2) \Rightarrow case$
 $mul\text{-}ext\text{-}mem\ f\ mem\text{-}new\text{-}2\ (xsa\ @\ xs)\ ys\text{-}new\ of\ (tmp\text{-}res, mem\text{-}new\text{-}3)$
 \Rightarrow
 $if\ snd\ tmp\text{-}res$
 $then\ ((True, True), mem\text{-}new\text{-}3)$
 $else\ mul\text{-}ext\text{-}dom\text{-}mem\ f\ mem\text{-}new\text{-}3\ xsa\ (x \# xs)\ y\ ys)$
 $| (False, True) \Rightarrow (case\ mul\text{-}ext\text{-}mem\ f\ mem\text{-}new\text{-}1\ (xsa\ @\ xs)\ ys\ of$
 $(sns\text{-}res\text{-}a, mem\text{-}new\text{-}2) \Rightarrow case\ mul\text{-}ext\text{-}dom\text{-}mem\ f\ mem\text{-}new\text{-}2\ xsa\ (x$
 $\# xs)\ y\ ys\ of$
 $(sns\text{-}res\text{-}b, mem\text{-}new\text{-}3) \Rightarrow$
 $(or2\ sns\text{-}res\text{-}a\ sns\text{-}res\text{-}b, mem\text{-}new\text{-}3))$
 $| (False, False) \Rightarrow mul\text{-}ext\text{-}dom\text{-}mem\ f\ mem\text{-}new\text{-}1\ xsa\ (x \# xs)\ y\ ys))$
by *pat-completeness auto*
termination by (*relation measures* [
 $(\lambda\ input.\ case\ input\ of\ Inl\ (-, -, xs, ys) \Rightarrow length\ ys\ | Inr\ (-, -, xs, xs', y, ys) \Rightarrow$
 $length\ ys),$
 $(\lambda\ input.\ case\ input\ of\ Inl\ (-, -, xs, ys) \Rightarrow 0\ | Inr\ (-, -, xs, xs', y, ys) \Rightarrow Suc$
 $(length\ xs))$
 $])$
 $(auto\ dest:\ filter\text{-}mem\text{-}len2)$

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 $filter\text{-}mem\ pre\ f\ post\ mem\ xs = filter\text{-}mem\ pre\ g\ post\ mem\ xs$
using *assms by (induct xs arbitrary: mem, auto split: prod.splits)*

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

assumes $\bigwedge m\ x.\ x \in set\ xs \Longrightarrow f\ m\ (pre\ x) = g\ m\ (pre\ x)$
shows $forall\text{-}mem\ pre\ f\ post\ mem\ xs = forall\text{-}mem\ pre\ g\ post\ mem\ xs$
using *assms by (induct xs arbitrary: mem, auto split: prod.splits)*

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 *exists-mem pre f post mem xs = exists-mem pre g post mem xs*
using *assms* **by** (*induct xs arbitrary: mem, auto split: prod.splits*)

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 *lex-ext-unbounded-mem f m xs ys = lex-ext-unbounded-mem g m xs ys*
using *assms*
by (*induct f m xs ys rule: lex-ext-unbounded-mem.induct, auto split: prod.splits bool.splits*)

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 *mul-ext-mem f m xs ys = mul-ext-mem g m xs ys*
proof –
have $(\bigwedge x' y' m. x' \in \text{set } xs \implies y' \in \text{set } ys \implies f m (x',y') = g m (x', y')) \implies$
 $\text{mul-ext-mem } f m xs ys = \text{mul-ext-mem } g m xs ys$
 $(\bigwedge x' y' m. x' \in \text{set } (xs @ xs') \implies y' \in \text{set } (y \# ys) \implies f m (x', y') = g m$
 $(x', y')) \implies$
 $\text{mul-ext-dom-mem } f m xs xs' y ys = \text{mul-ext-dom-mem } g m xs xs' y ys$ **for**
 $xs' y$
proof (*induct g m xs ys and g m xs xs' y ys rule: mul-ext-mem-mul-ext-dom-mem.induct*)
case $(6 g m x xs xs' y ys)$
note $IHs = 6(1-5)$
note $fg = 6(6)$
note $[simp\ del] = \text{mul-ext-mem.simps mul-ext-dom-mem.simps}$
note $[simp] = \text{mul-ext-dom-mem.simps}(2)[of - m x xs xs' y ys]$
from fg **have** $fgx[simp]: f m (x, y) = g m (x, y)$ **by** *simp*
obtain $a1 b1 m1$ **where** $r1[simp]: g m (x, y) = ((a1,b1),m1)$ **by** (*cases g m*
 $(x,y), auto$)
note $IHs = IHs(1-5)[OF r1[symmetric] refl]$
show *?case*
proof (*cases a1*)
case *True*
hence $a1 = True$ **by** *auto*
note $IHs = IHs(1-2)[OF this]$
let $?rec = \text{filter-mem } (Pair x) g (\lambda p. \neg \text{fst } p) m1 ys$
let $?recf = \text{filter-mem } (Pair x) f (\lambda p. \neg \text{fst } p) m1 ys$
have $[simp]: ?recf = ?rec$
by (*rule filter-mem-cong, insert fg, auto*)
obtain $zs m2$ **where** $rec: ?rec = (zs,m2)$ **by** *fastforce*
from *filter-mem-set*[*OF rec*] **have** $sub: \text{set } zs \subseteq \text{set } ys$ **by** *auto*
note $IHs = IHs(1-2)[OF rec[symmetric]]$
have $IH1[simp]: \text{mul-ext-mem } f m2 (xs @ xs') zs = \text{mul-ext-mem } g m2 (xs @$
 $xs') zs$
by (*rule IHs(1), rule fg*) (*insert sub, auto*)
obtain $p3 m3$ **where** $rec2[simp]: \text{mul-ext-mem } g m2 (xs @ xs') zs = (p3,m3)$


```

by fastforce
  note IHs(2)[OF rec2[symmetric] - fg]
  thus ?thesis using True by (simp add: rec)
next
  case False
  hence a1 = False by simp
  note IHs = IHs(3-)[OF this]
  show ?thesis
  proof (cases b1)
    case True
    hence b1 = True by simp
    note IHs = IHs(1-2)[OF this]
    have [simp]: mul-ext-mem f m1 (xs @ xs') ys = mul-ext-mem g m1 (xs @
xs') ys
      by (rule IHs(1)[OF fg], auto)
    obtain p2 m2 where rec1[simp]: mul-ext-mem g m1 (xs @ xs') ys = (p2,m2)
  by fastforce
    have [simp]: mul-ext-dom-mem f m2 xs (x # xs') y ys = mul-ext-dom-mem
g m2 xs (x # xs') y ys
      by (rule IHs(2)[OF rec1[symmetric] fg], auto)
    show ?thesis using False True by simp
  next
    case b1: False
    hence b1 = False by simp
    note IHs = IHs(3)[OF this fg]
    have [simp]: mul-ext-dom-mem f m1 xs (x # xs') y ys = mul-ext-dom-mem
g m1 xs (x # xs') y ys
      by (rule IHs, auto)
    show ?thesis using False b1 by auto
  qed
qed
qed auto
with assms show ?thesis by auto
qed

```

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 (λy. h (fun (g (f y)))) xs ∧ valid-memory fun ind mem2*

using *assms*

proof (*induct xs arbitrary: mem1 ys mem2*)

case (*Cons x xs mem1 ys mem'*)

note *res = Cons(3)*

note *mem1 = Cons(2)*

note *fun-mems = Cons(4)*

obtain *p mem2* **where** *fm: fun-mem mem1 (f x) = (p, mem2)* **by force**

from *memoize-funD[OF fun-mems mem1 fm]*

have $p: p = \text{fun } (g \ (f \ x))$ **and** $\text{mem2}: \text{valid-memory fun ind mem2}$ **by** *auto*
note $\text{res} = \text{res}[\text{unfolded filter-mem.simps fm split}]$
obtain $zs \ \text{mem3}$ **where** $\text{rec}: \text{filter-mem } f \ \text{fun-mem } h \ \text{mem2} \ xs = (zs, \ \text{mem3})$ **by**
force
note $\text{res} = \text{res}[\text{unfolded rec split}]$
from $\text{Cons}(1)[\text{OF mem2 rec memoize-fun-mono}[\text{OF fun-mems}]]$
have $\text{mem3}: \text{valid-memory fun ind mem3}$ **and** $zs: zs = \text{filter } (\lambda y. h \ (\text{fun } (g \ (f \ y)))) \ xs$ **by** *auto*
from mem3 res
show *?case unfolding* $zs \ p$ **by** *auto*
qed *auto*

lemma forall-mem: assumes $\text{valid-memory fun ind } m$
and $\text{forall-mem } f \ \text{fun-mem } h \ m \ xs = (b, \ m')$
and $\text{memoize-fun fun-mem fun } g \ \text{ind } (f \ ' \ \text{set } xs)$
shows $b = \text{Ball } (\text{set } xs) \ (\lambda s. h \ (\text{fun } (g \ (f \ s)))) \wedge \text{valid-memory fun ind } m'$
using *assms*
proof (*induct xs arbitrary: m b m'*)
case ($\text{Cons } x \ xs \ m \ b \ m'$)
obtain $b1 \ m1$ **where** $x: \text{fun-mem } m \ (f \ x) = (b1, m1)$ **by** *force*
note $\text{res} = \text{Cons}(3)[\text{unfolded forall-mem.simps } x \ \text{map-prod-simp split}]$
note $\text{mem} = \text{Cons}(2)$
from $\text{memoize-funD}[\text{OF Cons}(4) \ \text{mem } x]$
have $b1: b1 = \text{fun } (g \ (f \ x))$ **and** $m1: \text{valid-memory fun ind } m1$ **by** *auto*
obtain $b2 \ m2$ **where** $\text{rec}: \text{forall-mem } f \ \text{fun-mem } h \ m1 \ xs = (b2, m2)$ **by** *fastforce*
from $\text{Cons}(1)[\text{OF } m1 \ \text{rec memoize-fun-mono}[\text{OF Cons}(4)]]$
have $\text{IH}: b2 = \text{Ball } (\text{set } xs) \ (\lambda s. h \ (\text{fun } (g \ (f \ s))))$ **and** $m2: \text{valid-memory fun ind } m2$ **by** *auto*
show *?case using* $\text{res rec IH } m2 \ b1 \ m1$ **by** (*auto split: if-splits*)
qed *auto*

lemma exists-mem: assumes $\text{valid-memory fun ind } m$
and $\text{exists-mem } f \ \text{fun-mem } h \ m \ xs = (b, \ m')$
and $\text{memoize-fun fun-mem fun } g \ \text{ind } (f \ ' \ \text{set } xs)$
shows $b = \text{Bex } (\text{set } xs) \ (\lambda s. h \ (\text{fun } (g \ (f \ s)))) \wedge \text{valid-memory fun ind } m'$
using *assms*
proof (*induct xs arbitrary: m b m'*)
case ($\text{Cons } x \ xs \ m \ b \ m'$)
obtain $b1 \ m1$ **where** $x: \text{fun-mem } m \ (f \ x) = (b1, m1)$ **by** *force*
note $\text{res} = \text{Cons}(3)[\text{unfolded exists-mem.simps } x \ \text{map-prod-simp split}]$
note $\text{mem} = \text{Cons}(2)$
from $\text{memoize-funD}[\text{OF Cons}(4) \ \text{mem } x]$
have $b1: b1 = \text{fun } (g \ (f \ x))$ **and** $m1: \text{valid-memory fun ind } m1$ **by** *auto*
obtain $b2 \ m2$ **where** $\text{rec}: \text{exists-mem } f \ \text{fun-mem } h \ m1 \ xs = (b2, m2)$ **by** *fastforce*
from $\text{Cons}(1)[\text{OF } m1 \ \text{rec memoize-fun-mono}[\text{OF Cons}(4)]]$
have $\text{IH}: b2 = \text{Bex } (\text{set } xs) \ (\lambda s. h \ (\text{fun } (g \ (f \ s))))$ **and** $m2: \text{valid-memory fun ind } m2$ **by** *auto*
show *?case using* $\text{res rec IH } m2 \ b1 \ m1$ **by** (*auto split: if-splits*)
qed *auto*

lemma *lex-ext-unbounded-mem*: **assumes** $rel\text{-}pair = (\lambda(s, t). rel\ s\ t)$
shows $valid\text{-}memory\ rel\text{-}pair\ ind\ mem \implies lex\text{-}ext\text{-}unbounded\text{-}mem\ rel\text{-}mem\ mem\ xs\ ys = (p, mem')$
 $\implies memoize\text{-}fun\ rel\text{-}mem\ rel\text{-}pair\ (map\text{-}prod\ g\ h)\ ind\ (set\ xs \times set\ ys)$
 $\implies p = lex\text{-}ext\text{-}unbounded\ rel\ (map\ g\ xs)\ (map\ h\ ys) \wedge valid\text{-}memory\ rel\text{-}pair\ ind\ mem'$
proof (*induct rel-mem mem xs ys arbitrary: p mem' rule: lex-ext-unbounded-mem.induct*)
case $(\lambda rel\text{-}mem\ mem\ x\ xs\ y\ ys)$
note *lex-ext-unbounded.simps[simp]*
note $IH = \lambda(1)[OF\ refl - refl]$
obtain $s\ ns\ mem1$ **where** *impl: rel-mem mem (x, y) = ((s, ns), mem1)* **by** (*cases rel-mem mem (x, y), auto*)
have $rel: rel\ (g\ x)\ (h\ y) = (s, ns)$ **and** $mem1: valid\text{-}memory\ rel\text{-}pair\ ind\ mem1$
using *memoize-funD[OF $\lambda(4,2)$ impl]* **assms** *impl* **unfolding** *assms o-def* **by** *auto*
note $res = \lambda(3)[unfolding\ lex\text{-}ext\text{-}unbounded\text{-}mem.simps\ Let\text{-}def\ impl\ split]$
have $rel\text{-}pair: lex\text{-}ext\text{-}unbounded\ rel\ (map\ g\ (x\ \#\ xs))\ (map\ h\ (y\ \#\ ys)) = ($
 $if\ s\ then\ (True, True)\ else\ if\ ns\ then\ lex\text{-}ext\text{-}unbounded\ rel\ (map\ g\ xs)\ (map\ h\ ys)\ else\ (False, False))$
unfolding *lex-ext-unbounded.simps list.simps Let-def split rel* **by** *simp*
show *?case*
proof (*cases s \vee \neg ns*)
case *True*
thus *?thesis* **using** *res rel-pair mem1* **by** *auto*
next
case *False*
obtain $p2\ mem2$ **where** *rec: lex-ext-unbounded-mem rel-mem mem1 xs ys = (p2, mem2)* **by** *fastforce*
from *False* **have** $s = False\ ns = True$ **by** *auto*
from $IH[unfolding\ impl, OF\ refl\ this\ mem1\ rec\ memoize\text{-}fun\ mono[OF\ \lambda(4)]]$
have $mem2: valid\text{-}memory\ rel\text{-}pair\ ind\ mem2$ **and** $p2: p2 = lex\text{-}ext\text{-}unbounded\ rel\ (map\ g\ xs)\ (map\ h\ ys)$ **by** *auto*
show *?thesis* **unfolding** *rel-pair* **using** *res rec False mem2 p2* **by** (*auto split: if-splits*)
qed
qed (*auto simp: lex-ext-unbounded.simps*)

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

```

ys))
 $\implies p = \text{mul-ex-dom rel (map g xs) (map g xs') (h y) (map h ys) \wedge \text{valid-memory rel-pair ind mem'}$ 
for  $xs' y$ 
proof (induct rel-mem mem xs ys and rel-mem mem xs xs' y ys arbitrary: p mem' and p mem' rule: mul-ext-mem-mul-ext-dom-mem.induct)
  case (6 sns mem x xs ys d zs pair mem')
  note  $IHs = 6(1-5)$ 
  note  $mem = 6(6)$ 
  note  $res = 6(7)$ 
  note  $memo = 6(8)$ 
  let  $?Sns = \lambda x. \text{rel-pair (map-prod g h x)}$ 
  let  $?xd = \text{rel-pair (g x, h d)}$ 
  obtain  $p1 mem1$  where  $sns: sns mem (x, d) = (p1, mem1)$  by fastforce
  note  $IHs = IHs[OF sns[symmetric]]$ 
  from memoize-funD[OF memo mem sns]
  have  $p1: p1 = ?xd$  and  $mem1: \text{valid-memory rel-pair ind mem1}$  by auto
  note  $sns = sns[unfolded p1]$ 
  note  $res = res[unfolded mul-ext-dom-mem.simps sns split]$ 
  have  $rel: rel (g x) (h d) = ?xd$  unfolding assms by auto
  define  $wp$  where  $wp = \text{mul-ex-dom rel (map g (x \# xs)) (map g ys) (h d) (map h zs)}$ 
  note  $wp = wp\text{-def}[unfolded list.simps, unfolded mul-ex-dom.simps rel]$ 
  consider (1)  $b$  where  $?xd = (True, b) \mid (2) ?xd = (False, True) \mid (3) ?xd = (False, False)$ 
  by (cases ?xd, auto)
  hence  $\text{valid-memory rel-pair ind mem'} \wedge \text{pair} = wp$ 
proof cases
  case (1  $b$ )
  let  $?pre = \text{Pair } x$ 
  let  $?post = (\lambda p. \neg \text{fst } p)$ 
  from 1  $p1$  have  $(True, b) = p1$  by auto
  note  $IHs = IHs(1-2)[OF \text{this}, OF \text{refl}]$ 
  obtain  $p2 mem2$  where  $\text{filter: filter-mem } ?pre \text{ sns } ?post \text{ mem1 } zs = (p2, mem2)$  by force
  obtain  $p3 mem3$  where  $\text{rec1: mul-ext-mem sns mem2 (xs @ ys) } p2 = (p3, mem3)$  by fastforce
  obtain  $p4 mem4$  where  $\text{rec2: mul-ext-dom-mem sns mem3 xs (x \# ys) } d \text{ zs} = (p4, mem4)$  by fastforce
  note  $res = res[unfolded 1 \text{ split}[of - - mem1], unfolded Let-def \text{ split}, simplified, unfolded \text{ filter } \text{rec1} \text{ split } \text{rec2}]$ 
  note  $wp = wp[unfolded 1 \text{ split } \text{bool.simps}]$ 
  {
  fix  $z$ 
  assume  $z \in \text{set } zs$ 
  hence  $(x, z) \in \text{set } ((x \# xs) @ ys) \times \text{set } (d \# zs)$  by auto
  from memoize-funD[OF memo - - this]
  have  $\text{valid-memory rel-pair ind } m \implies \text{sns } m (x, z) = (p, m') \implies p = \text{rel-pair (map-prod g h (x, z)) \wedge \text{valid-memory rel-pair ind } m'$ 

```

```

    for m p m' by auto
  }
  hence memoize-fun sns rel-pair (map-prod g h) ind (Pair x ' set zs)
    by (intro memoize-funI, blast)
  from filter-mem[OF mem1 filter, of map-prod g h,
    OF this]
  have mem2: valid-memory rel-pair ind mem2 and p2: p2 = filter (λy. ¬ fst
    (rel-pair (g x, h y))) zs
    by auto
  have filter (λy. ¬ fst (rel (g x) y)) (map h zs) = map h p2 unfolding p2
    asms split
    by (induct zs, auto)
  note wp = wp[unfolded this]
  note IHs = IHs[OF filter[symmetric]]
  from IHs(1)[OF mem2 rec1 memoize-fun-mono[OF memo]] p2
  have mem3: valid-memory rel-pair ind mem3
    and p3: p3 = mul-ext-impl rel (map g xs @ map g ys) (map h p2)
    by auto
  note wp = wp[folded p3]
  show ?thesis
  proof (cases snd p3)
    case True
    thus ?thesis using res wp mem3 by auto
  next
    case False
    with IHs(2)[OF rec1[symmetric] False mem3 rec2 memoize-fun-mono[OF
      memo]] wp res
    show ?thesis by auto
  qed
  next
    case 2
    note wp = wp[unfolded 2 split bool.simps]
    obtain p2 mem2 where rec2: mul-ext-mem sns mem1 (xs @ ys) zs = (p2,
      mem2) by fastforce
    obtain p3 mem3 where rec3: mul-ext-dom-mem sns mem2 xs (x # ys) d zs
      = (p3, mem3) by fastforce
    from 2 p1 have (False, True) = p1 by auto
    note IHs = IHs(3-4)[OF this refl refl, unfolded rec2]
    from IHs(1)[OF mem1 refl memoize-fun-mono[OF memo]]
    have mem2: valid-memory rel-pair ind mem2 and p2: p2 = mul-ext-impl rel
      (map g (xs @ ys)) (map h zs)
      by auto
    from IHs(2)[OF refl mem2 rec3 memoize-fun-mono[OF memo]]
    have mem3: valid-memory rel-pair ind mem3 and p3: p3 = mul-ex-dom rel
      (map g xs) (map g (x # ys)) (h d) (map h zs) by auto
    from wp res[unfolded Let-def split 2 bool.simps rec2 rec3]
    show ?thesis using mem3 p2 p3 by auto
  next
    case 3

```

```

obtain p2 mem2 where rec2: mul-ext-dom-mem sns mem1 xs (x # ys) d zs
= (p2,mem2) by fastforce
from  $\exists$  p1 have (False, False) = p1 by auto
from IHs(5)[OF this refl refl mem1 rec2 memoize-fun-mono[OF memo]]
have mem2: valid-memory rel-pair ind mem2 and p2: p2 = mul-ext-dom rel
(map g xs) (map g (x # ys)) (h d) (map h zs)
by auto
have wp = p2 unfolding wp  $\exists$  using p2 by simp
with mem2 show ?thesis using p2 res  $\exists$  rec2 by auto
qed
thus ?case unfolding wp-def by blast
qed auto
thus ?A  $\implies$  ?B  $\implies$  ?C  $\implies$  ?D by blast
qed

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  $\implies$  bool  $\times$  bool  $\implies$  bool
```

```
where
```

```
  compare-bools p1 p2  $\iff$  (fst p1  $\longrightarrow$  fst p2)  $\wedge$  (snd p1  $\longrightarrow$  snd p2)
```

```
notation compare-bools ((-/  $\leq_{cb}$  -) [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-ext-unbounded f xs ys  $\leq_{cb}$  lex-ext-unbounded g xs ys
```

```
  unfolding compare-bools-def
```

```
  by (rule lex-ext-unbounded-mono,
```

```
  insert assms[unfolded compare-bools-def], auto)
```

```
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 $mul-ext\ f\ xs\ ys \leq_{cb}\ mul-ext\ g\ xs\ ys$
unfolding *compare-bools-def*
by (*intro conjI impI; rule mul-ext-mono*) (*insert assms, auto simp: compare-bools-def*)

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 $cS\ cNS :: ('f, 'v)term \Rightarrow ('f, 'v)term \Rightarrow bool$ — sufficient criteria
and $\sigma :: 'f\ status$
and $c :: 'f \times nat \Rightarrow order-tag$

begin

fun $wpo-ub :: ('f, 'v) term \Rightarrow ('f, 'v) term \Rightarrow bool \times bool$

where

$wpo-ub\ s\ t = (if\ cS\ s\ t\ then\ (True,\ True)\ else\ if\ cNS\ s\ t\ then\ (case\ s\ of$
 $\quad Var\ x \Rightarrow (False,$
 $\quad (case\ t\ of$
 $\quad\quad Var\ y \Rightarrow x = y$
 $\quad | Fun\ g\ ts \Rightarrow status\ \sigma\ (g,\ length\ ts) = [] \wedge prl\ (g,\ length\ ts)))$
 $| Fun\ f\ ss \Rightarrow$
 $\quad let\ ff = (f,\ length\ ss); sf = status\ \sigma\ ff\ in$
 $\quad if\ (\exists\ i \in set\ sf.\ snd\ (wpo-ub\ (ss\ !\ i)\ t))\ then\ (True,\ True)$
 $\quad else$
 $\quad (case\ t\ of$
 $\quad\quad Var\ - \Rightarrow (False,\ ssimple \wedge large\ ff)$
 $\quad | Fun\ g\ ts \Rightarrow$
 $\quad\quad let\ gg = (g,\ length\ ts); sg = status\ \sigma\ gg\ in$
 $\quad\quad (case\ pr\ ff\ gg\ of\ (prs,\ prns) \Rightarrow$
 $\quad\quad\quad if\ prns \wedge (\forall\ j \in set\ sg.\ fst\ (wpo-ub\ s\ (ts\ !\ j)))\ then$
 $\quad\quad\quad\quad if\ prs\ then\ (True,\ True)$
 $\quad\quad\quad\quad else$
 $\quad\quad\quad\quad\quad let\ ss' = map\ (\lambda\ i.\ ss\ !\ i)\ sf;$
 $\quad\quad\quad\quad\quad ts' = map\ (\lambda\ i.\ ts\ !\ i)\ sg;$
 $\quad\quad\quad\quad\quad cf = c\ ff;$
 $\quad\quad\quad\quad\quad cg = c\ gg\ in$
 $\quad\quad\quad\quad\quad if\ cf = Lex \wedge cg = Lex\ then\ lex-ext-unbounded\ wpo-ub\ ss'\ ts'$
 $\quad\quad\quad\quad\quad else\ if\ cf = Mul \wedge cg = Mul\ then\ mul-ext\ wpo-ub\ ss'\ ts'$
 $\quad\quad\quad\quad\quad else\ if\ ts' = []\ then\ (ss' \neq [],\ True)\ else\ (False,\ False)$
 $\quad\quad\quad\quad\quad else\ (False,\ False)))$
 $\quad) else\ (False,\ False))$

declare $wpo-ub.simps$ [*simp del*]

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

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  $wpo\text{-ub } s\ t \leq_{cb} wpo\text{-orig } n\ S\ NS\ s\ t$ 
  using assms
proof (induct s t rule: wpo.wpo.induct [of S NS  $\sigma$  - n pr prl c simple large])
  case (1 s t)
  note  $IH = 1(1-4)$ 
  note  $cb = 1(5)$ 
  note  $n = 1(6)$ 
  note  $cbd = \text{compare-bools-def}$ 
  note  $\text{simps} = wpo\text{-ub.simps}[of\ s\ t]\ wpo.wpo.simps[of\ n\ S\ NS\ pr\ prl\ \sigma\ c\ simple$ 
 $large\ s\ t]$ 
  let  $?wpo = wpo\text{-orig } n\ S\ NS$ 
  let  $?cb = \lambda\ s\ t. (cS\ s\ t, cNS\ s\ t) \leq_{cb} ((s, t) \in S, (s, t) \in NS)$ 
  let  $?goal = \lambda\ s\ t. wpo\text{-ub } s\ t \leq_{cb} ?wpo\ s\ t$ 
  from  $cb[of\ s\ t]$  have  $cb\text{-st}: ?cb\ s\ t$  by auto
  show  $?case$ 
  proof ( $\text{cases } (s, t) \in S \vee \neg cNS\ s\ t$ )
    case True
    with  $cb\text{-st}$  show  $?thesis$  unfolding simps unfolding cbd by auto
  next
  case False
  with  $cb\text{-st}$  have  $*$ :  $(s, t) \notin S, (s, t) \in NS \implies ((s, t) \notin S) = \text{True} \implies ((s, t) \in S) = \text{False}$ 
 $((s, t) \in NS) = \text{True} \implies cS\ s\ t = \text{False} \implies cNS\ s\ t = \text{True}$ 
    unfolding cbd by auto
    note  $\text{simps} = \text{simps}[unfolded\ * \text{ if-False if-True}]$ 
    note  $IH = IH[OF\ *(1-2)]$ 
    show  $?thesis$ 
    proof ( $\text{cases } s$ )
      case (Var x) note  $s = \text{this}$ 
      show  $?thesis$ 
      proof ( $\text{cases } t$ )
        case (Var y) note  $t = \text{this}$ 
        show  $?thesis$  unfolding simps unfolding  $s\ t$  cbd by simp
      next
      case (Fun g ts) note  $t = \text{this}$ 
      show  $?thesis$  unfolding simps unfolding  $s\ t$  cbd by auto
    qed
  next
  case  $s$ : (Fun f ss)
  let  $?f = (f, \text{length } ss)$ 
  let  $?sf = \text{status } \sigma\ ?f$ 
  let  $?s = \text{Fun } f\ ss$ 
  note  $IH = IH[OF\ s]$ 
  show  $?thesis$ 
  proof ( $\text{cases } (\exists i \in \text{set } ?sf. \text{snd } (?wpo\ (ss\ !\ i)\ t))$ )

```



```

    case True
  then show ?thesis unfolding_simps using True * unfolding s cbd by auto
next
case False
{
  fix i
  assume i: i ∈ set ?sf
  from status-aux[OF i]
  have ?goal (ss ! i) t
    by (intro IH(1)[OF i cb n], auto simp: s)
}
  with False have sub: (∃ i ∈ set ?sf. snd (wpo-ub (ss ! i) t)) = False
unfolding cbd by auto
note IH = IH(2-4)[OF False]
show ?thesis
proof (cases wpo-ub s t = (False, False))
  case True
  then show ?thesis unfolding cbd by auto
next
case False note noFF = this
  note False = False[unfolded_simps * Let-def, unfolded s term_simps sub,
simplified]
  show ?thesis
  proof (cases t)
    case t: (Var y)
    from False[unfolded t, simplified]
    show ?thesis unfolding s t unfolding cbd
      using * s_simps sub t by auto
  next
  case t: (Fun g ts)
  let ?g = (g, length ts)
  let ?sg = status σ ?g
  let ?t = Fun g ts
  obtain ps pns where p: pr ?f ?g = (ps, pns) by force
  note IH = IH[OF t p[symmetric]]
  note False = False[unfolded t p split term_simps]
  from False have pns: pns = True by (cases pns, auto)
  {
    fix j
    assume j: j ∈ set ?sg
    from status-aux[OF j]
    have cb: ?goal s (ts ! j)
      by (intro IH(1)[OF j cb n], auto simp: t)
    from j False have fst (wpo-ub s (ts ! j)) unfolding s by (auto split:
if-splits)
  }
  with cb have fst (?wpo s (ts ! j)) unfolding cbd by auto
}
  then have cond: pns ∧ (∀ j ∈ set ?sg. fst (?wpo s (ts ! j))) using pns
by auto

```

```

    note IH = IH(2-3)[OF cond]
    from cond have cond: (pns ∧ (∀ j ∈ set ?sg. fst (?wpo ?s (ts ! j)))) =
True unfolding s by simp
    note_simps =_simps[unfolded * Let-def, unfolded s t term.simps if-False
if-True sub[unfolded t] p split cond]
    show ?thesis
    proof (cases ps)
      case True
      then show ?thesis unfolding s t unfolding_simps cbd by auto
    next
      case False
      note IH = IH[OF this refl refl refl refl]
      let ?msf = map (!) ss ?sf
      let ?msg = map (!) ts ?sg
      have set-msf: set ?msf ⊆ set ss using status[of σ f length ss]
      unfolding set-conv-nth by force
      have set-msg: set ?msg ⊆ set ts using status[of σ g length ts]
      unfolding set-conv-nth by force
      {
        fix i
        assume i < length ?msf
        then have ?msf ! i ∈ set ?msf unfolding set-conv-nth by blast
        with set-msf have ?msf ! i ∈ set ss by auto
      } note msf = this
      {
        fix i
        assume i < length ?msg
        then have ?msg ! i ∈ set ?msg unfolding set-conv-nth by blast
        with set-msg have ?msg ! i ∈ set ts by auto
      } note msg = this
    show ?thesis
    proof (cases c ?f = Lex ∧ c ?g = Lex)
      case Lex: True
      note IH = IH(1)[OF Lex - - cb n, unfolded s t length-map]
      from n[of ?g, unfolded t] have length (?msg) ≤ n by auto
      then have ub: lex-ext-unbounded ?wpo ?msf ?msg =
      lex-ext ?wpo n ?msf ?msg
      unfolding lex-ext-unbounded-iff lex-ext-iff by auto
      from Lex False_simps noFF
      have wpo-ub: wpo-ub s t = lex-ext-unbounded wpo-ub ?msf ?msg
      unfolding s t using False by (auto split: if-splits)
      also have ... ≤cb lex-ext-unbounded ?wpo ?msf ?msg
      by (rule lex-ext-unbounded-cb, rule IH) (insert msf msg, auto)
      finally show ?thesis unfolding ub s t_simps(2) cbd using Lex by
auto
    next
      case nLex: False
      show ?thesis
      proof (cases c ?f = Mul ∧ c ?g = Mul)

```

```

    case Mul: True
    note IH = IH(2)[OF nLex Mul - - cb n, unfolded s t]
    from Mul nLex False_simps noFF
    have wpo-ub: wpo-ub s t = mul-ext wpo-ub ?msf ?msg
      unfolding s t using False by (auto split: if-splits)
    also have ...  $\leq_{cb}$  mul-ext ?wpo ?msf ?msg
      by (rule mul-ext-cb, rule IH) (insert set-msf set-msg, auto)
    finally show ?thesis unfolding s t_simps(2) cbd using nLex Mul
  by auto
  next
  case nMul: False
  thus ?thesis unfolding s t_simps cbd using nLex nMul noFF False
    by auto
  qed
  qed
  qed
  qed
  qed
  qed
  qed
  qed
  qed
  qed
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* \Rightarrow (*f*, *v*) *term* \Rightarrow (*bool* \times *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* \Rightarrow (*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 \bigwedge *si*. *fst* *st* \supseteq *si* \implies *rli* (*index* *si*) = *unindex* *si* \wedge *stored* *si* = *unindex* *si*

and \bigwedge *ti*. *snd* *st* \supseteq *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'*

using *assms*(4-)

proof (*induct* *m st* **and** *m st* *arbitrary*: *p m'* **and** *p m'* *rule*: *wpo-mem-wpo-main.induct*)

case (*1 m s t*)

note *IH* = *1*(1)

note *revi* = *1*(3,4)[*unfolded fst-conv snd-conv*]

note *mem* = *1*(5)

note *res* = *1*(2)[*unfolded wpo wpo-mem.simps Let-def*]

have *ri*: *ri* (*index* *s*, *index* *t*) = (*unindex* *s*, *unindex* *t*)

unfolding *ri* **using** *revi*(1)[*of s*] *revi*(2)[*of t*] **by** *auto*

show ?*case*

proof (*cases* *Mapping.lookup* *m* (*index* *s*, *index* *t*))

case (*Some* *q*)

note *res* = *res*[*unfolded Some option.simps*]

from *res* **have** *id*: *p* = *q* *m'* = *m* **by** *auto*

from *mem*[*unfolded valid-memory-def, rule-format, OF Some*]

have *wpo* (*ri* (*index* *s*, *index* *t*)) = *q* **by** *auto*

with *ri* **show** ?*thesis* **unfolding** *id* **using** *mem* **by** *auto*

next

case *None*

note *res* = *res*[*unfolded None option.simps*]

obtain *res2* *mem2* **where** *rec*: *wpo-main* *m* (*s*, *t*) = (*res2*, *mem2*) **by** *fastforce*

have *res2*: *res2* = *wpo* (*unindex* *s*, *unindex* *t*) **and** *mem*: *valid-memory* *wpo ri* *mem2*

using *IH*[*OF refl refl None rec rev* *mem*] **by** *auto*

from *res*[*unfolded rec split*]

have *p*: *p* = *res2* **and** *m'*: *m'* = *Mapping.update* (*index* *s*, *index* *t*) *res2* *mem2* **by** *auto*

```

    show ?thesis unfolding p res2 m' using mem ri
      by (auto simp add: valid-memory-def lookup-update')
qed
next
case (2 m s t)
let ?s = unindex s
let ?t = unindex t
note rev1 = 2(6,7)[unfolded fst-conv snd-conv]
from rev1(1)[of s] rev1(2)[of t]
have stored: stored s = unindex s stored t = unindex t by auto
note IHs = 2(1-4)[OF stored[symmetric]]
note mem = 2(8)
note res = 2(5)[unfolded wpo-main.simps Let-def stored]
have wpo-st: wpo (unindex s, unindex t) = wpoub (unindex s) (unindex t) for s
t
  unfolding wpo by simp
  note wpo = this[of s t, unfolded wpoub wpo-ub.simps[of - - - - - ?s ?t], folded
wpoub]
  show ?case
  proof (cases s)
    case (Var xi)
    then obtain x i where s: s = Var (x,i) by (cases xi, auto)
    thus ?thesis using res mem wpo by (cases t, auto)
  next
  case (Fun fi ss)
  then obtain f i where s: s = Fun (f,i) ss by (cases fi, auto)
  let ?Sta = status  $\sigma$  (f, length ss)
  note res = res[unfolded s term.simps name-of.simps, folded s]
  note wpo = wpo[unfolded s unindex.simps term.simps, folded unindex.simps[of
- i], folded s,
    unfolded length-map Let-def]
  show ?thesis
  proof (rule ccontr)
    assume neg:  $\neg$  ?thesis
    from neg res mem wpo s have ncS:  $\neg$  cS ?s ?t by auto
    from neg res mem wpo s ncS have cNS: cNS ?s ?t by (auto split: if-splits)
    have id: map-prod unindex unindex (s,t) = (unindex s, unindex t) for s t ::
(f,v)indexed-term by auto
    define sss where sss = map (!) ss ?Sta
    note IHs = IHs[OF ncS cNS s refl refl refl, unfolded name-of.simps, unfolded
id fst-conv snd-conv, folded sss-def]
    from ncS cNS have id: cS ?s ?t = False cNS ?s ?t = True by auto
    note res = res[unfolded id if-True if-False, folded sss-def]
    have sss: (map (!) (map unindex ss)) ?Sta = map unindex sss
      unfolding sss-def by (auto dest: set-status-nth[OF refl])
    note wpo = wpo[unfolded id if-True if-False]
    have sss-sub: set sss  $\subseteq$  set ss unfolding sss-def by (auto dest: set-status-nth[OF
refl])
    let ?cond1' = Bex (set sss) ( $\lambda$ s. snd (wpoub (unindex s) (unindex t)))

```

```

let ?cond1'' = Bex (set ?Sta) (λi. snd (wpoub (map unindex ss ! i) (unindex
t)))
have ?cond1'' = ?cond1' unfolding sss-def
  using set-status-nth[OF refl, of - σ f ss] by simp
note wpo = wpo[unfolded this sss]
let ?cond1 = exists-mem (λ s'. (s',t)) wpo-mem snd m sss
obtain b1 m1 where cond1: ?cond1 = (b1,m1) by fastforce
{
  fix si
  assume si: si ∈ set sss
  have wpo-mem m (si, t) = (p, m') ⇒
    valid-memory wpo ri m ⇒ p = wpo (unindex si, unindex t) ∧ valid-memory
wpo ri m'
  for m p m'
  by (intro IHs(1)[OF si - rev1, of m p m'], insert sss-sub s si, auto)
}
hence memoize-fun wpo-mem wpo (map-prod unindex unindex) ri ((λs'. (s',
t)) ' set sss)
  by (intro memoize-funI, auto)
from exists-mem[OF mem cond1 this]
have cond1': ?cond1' = b1 and mem1: valid-memory wpo ri m1
  unfolding wpo-st[symmetric] by auto
note IHs = IHs(2-)[OF cond1[symmetric]]
note res = res[unfolded cond1 split]
note wpo = wpo[unfolded cond1']
from neg res wpo mem1 have b1: ¬ b1 by auto
note IHs = IHs[OF this]
from b1 have b1: b1 = False by simp
note res = res[unfolded b1 if-False]
note wpo = wpo[unfolded b1 if-False]
show False
proof (cases t)
  case (Var yj)
  with neg res wpo mem1 show ?thesis by (cases yj, auto)
next
  case (Fun gj ts)
  then obtain g j where t: t = Fun (g,j) ts by (cases gj, auto)
  let ?f = (f, length ss) let ?g = (g, length ts)
  obtain prs prns where pr: pr ?f ?g = (prs, prns) by force
  let ?sta = (status σ (g, length ts))
  define tss where tss = map (!) ts ?sta
  have tss: (map (!) (map unindex ts)) ?sta = map unindex tss
  unfolding tss-def by (auto dest: set-status-nth[OF refl])
  have tss-sub: set tss ⊆ set ts unfolding tss-def by (auto dest: set-status-nth[OF
refl])
  note res = res[unfolded t term.simps name-of.simps pr split, folded tss-def]
  note wpo = wpo[unfolded t unindex.simps term.simps length-map pr split,
    folded unindex.simps[of - j], folded t, unfolded tss]
  from neg res mem1 wpo have prns: prns by (auto split: if-splits)

```

```

note IHs = IHs[OF t refl refl, unfolded name-of.simps, OF refl pr[symmetric],
folded tss-def, OF prns]
have prns: (prns ∧ b) = b prns = True for b using prns by auto
note res = res[unfolded prns if-True]
note wpo = wpo[unfolded prns(1)]
let ?cond2 = forall-mem (λ t'. (s,t')) wpo-mem fst m1 tss
let ?cond2'' = Ball (set ?sta) (λj. fst (wpoub ?s (map unindex ts ! j)))
let ?cond2' = Ball (set tss) (λt. fst (wpoub ?s (unindex t)))
have ?cond2'' = ?cond2' unfolding tss-def
using set-status-nth[OF refl, of - σ g ts] by simp
note wpo = wpo[unfolded this]
obtain b2 m2 where cond2: ?cond2 = (b2,m2) by force
{
  fix ti
  assume ti: ti ∈ set tss
  have wpo-mem m (s, ti) = (p, m') ⇒
  valid-memory wpo ri m ⇒ p = wpo (unindex s, unindex ti) ∧ valid-memory
wpo ri m'
  for m p m'
  by (intro IHs(1)[OF ti - rev1, of m p m'], insert tss-sub t ti, auto)
}
hence memoize-fun wpo-mem wpo (map-prod unindex unindex) ri (Pair s '
set tss)
by (intro memoize-funI, auto)
from forall-mem[OF mem1 cond2 this]
have cond2': ?cond2' = b2 and mem2: valid-memory wpo ri m2
unfolding wpo-st[symmetric] by auto
note wpo = wpo[unfolded cond2']
note res = res[unfolded cond2 split]
from neg res wpo mem2 have b2: b2 by (auto split: if-splits)
with neg res wpo mem2 have prs: ¬ prs by (auto split: if-splits)
note IHs = IHs(2-)[OF cond2[symmetric] b2 prs refl refl]
from b2 prs have id: b2 = True prs = False by auto
note res = res[unfolded id if-True if-False, folded sss-def tss-def]
note wpo = wpo[unfolded id if-True if-False]
let ?is-lex = c ?f = Lex ∧ c ?g = Lex
show False
proof (cases ?is-lex)
case True
note IH = IHs(1)[OF True]
from True have lex: ?is-lex = True by auto
note res = res[unfolded lex if-True]
note wpo = wpo[unfolded lex if-True]
have memo: memoize-fun wpo-mem wpo (map-prod unindex unindex) ri
(set sss × set tss)
apply (rule memoize-fun-pairI)
apply (rule IH)
apply force
apply force

```



```

      apply force
      subgoal by (rule rev_i, insert sss-sub, auto simp: s)
      subgoal by (rule rev_i, insert tss-sub, auto simp: t)
      by auto
      have p = lex-ext-unbounded wpoub (map unindex sss) (map unindex tss)
     $\wedge$  valid-memory wpo ri m'
      by (rule lex-ext-unbounded-mem[OF assms(2) mem2 res memo])
      with res wpo neg
      show ?thesis by auto
    next
      case False
      note IH = IHs(2)[OF False]
      from False have lex: ?is-lex = False by auto
      note res = res[unfolded lex if-False]
      note wpo = wpo[unfolded lex if-False]
      let ?is-mul = c (f, length ss) = Mul  $\wedge$  c (g, length ts) = Mul
      show False
      proof (cases ?is-mul)
        case True
        note IH = IH[OF True]
        from True have mul: ?is-mul = True by auto
        note res = res[unfolded mul if-True]
        note wpo = wpo[unfolded mul if-True]
        have memo: memoize-fun wpo-mem wpo (map-prod unindex unindex) ri
      (set sss  $\times$  set tss)
          apply (rule memoize-fun-pairI)
          apply (rule IH)
          apply force
          apply force
          apply force
          subgoal by (rule rev_i, insert sss-sub, auto simp: s)
          subgoal by (rule rev_i, insert tss-sub, auto simp: t)
          by auto
          have p = mul-ext-impl wpoub (map unindex sss) (map unindex tss)  $\wedge$ 
      valid-memory wpo ri m'
          using mul-ext-mem(1)[OF assms(2) mem2 res memo] by auto
          with res wpo neg
          show ?thesis unfolding mul-ext-code by auto
      next
        case False
        from False have mul: ?is-mul = False by auto
        note res = res[unfolded mul if-False]
        note wpo = wpo[unfolded mul if-False]
        from res wpo neg mem2 show False by (auto split: if-splits)
      qed
    qed
  qed
qed

```

qed

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 -

let ?s = index-term s

let ?t = index-term t

let ?m = Mapping.empty :: term-rel-mem

have m: valid-memory ($\lambda(s, t). wpo-ub pr prl ssimple large cS cNS \sigma c s t$)
(map-prod rl rr) ?m for rl rr

unfolding valid-memory-def by auto

from index-term-index-unindex[of s] obtain f where f: $\forall t \triangleleft index-term s. f$
(index t) = unindex t \wedge stored t = unindex t by auto

from index-term-index-unindex[of t] obtain g where g: $\forall s \triangleleft index-term t. g$
(index s) = unindex s \wedge stored s = unindex s by auto

obtain p m where res: wpo-mem ?m (?s, ?t) = (p, m) by fastforce

hence impl: wpo-mem-impl s t = p unfolding wpo-mem-impl-def by simp

also have ... = wpo-ub pr prl ssimple large cS cNS σ c (unindex (index-term
s)) (unindex (index-term t))

by (rule wpo-mem(1)[THEN conjunct1, OF refl refl refl - - m res, unfolded
map-prod-simp split fst-conv snd-conv, of f g])

(insert f g, auto)

finally show ?thesis by simp

qed

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) =>
  if prns & (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 != 0 & length ts = 0, length ts = 0)
    else (False, False)))

```

lemma *rpo-to-rpo-unbounded*:

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

shows $\text{rpo } pr \ prl \ c \ n \ s \ t = \text{rpo-unbounded } (pr, prl) \ c \ s \ t$ (**is** $?e \ s \ t$)

proof –

let $?p = \lambda \ s \ t. ?b \ s \ t \longrightarrow ?e \ s \ t$

let $?pr = (pr, prl)$

{

have $?p \ s \ t$

proof (*induct rule: rpo.induct[of - pr prl c n]*)

case $(\exists \ f \ ss \ y)$

show $?case$

proof (*intro impI*)

assume $?b \ (Fun \ f \ ss) \ (Var \ y)$

then have $\bigwedge \ s. s \in \text{set } ss \implies ?b \ s \ (Var \ y)$ **by** *auto*

with \exists **show** $?e \ (Fun \ f \ ss) \ (Var \ y)$ **by** *simp*

qed

next

case $(\lambda \ f \ ss \ g \ ts)$ **note** *IH = this*

show $?case$

proof (*intro impI*)

assume $?b \ (Fun \ f \ ss) \ (Fun \ g \ ts)$

then have $bs: \bigwedge \ s. s \in \text{set } ss \implies ?b \ s \ (Fun \ g \ ts)$

and $bt: \bigwedge \ t. t \in \text{set } ts \implies ?b \ (Fun \ f \ ss) \ t$

and $ss: \text{length } ss \leq n$ **and** $ts: \text{length } ts \leq n$ **by** *auto*

with $\lambda(1)$ **have** $s: \bigwedge \ s. s \in \text{set } ss \implies ?e \ s \ (Fun \ g \ ts)$ **by** *simp*

show $?e \ (Fun \ f \ ss) \ (Fun \ g \ ts)$

proof (*cases* $\exists \ s \in \text{set } ss. \text{snd } (rpo \ pr \ prl \ c \ n \ s \ (Fun \ g \ ts))$)

case *True* **with** s **show** $?thesis$ **by** *simp*

next

case *False* **note** $oFalse = \text{this}$

with s **have** $oFalse2: \neg (\exists \ s \in \text{set } ss. \text{snd } (rpo \ \text{unbounded } ?pr \ c \ s \ (Fun \ g \ ts)))$

by *simp*

obtain $prns \ prs$ **where** $Hsns: pr \ (f, \text{length } ss) \ (g, \text{length } ts) = (prns, prns)$

by *force*

with $bt \ \lambda(2)[OF \ oFalse]$

have $t: \bigwedge \ t. t \in \text{set } ts \implies ?e \ (Fun \ f \ ss) \ t$ **by** *force*

```

show ?thesis
proof (cases prns  $\wedge$  ( $\forall t \in \text{set } ts. \text{fst } (rpo \text{ pr prl } c \ n \ (\text{Fun } f \ ss) \ t)$ ))
  case False
    show ?thesis
    proof (cases prns)
      case False then show ?thesis by (simp add: oFalse oFalse2 Hsns)
    next
      case True
        with False have Hf1:  $\neg (\forall t \in \text{set } ts. \text{fst } (rpo \text{ pr prl } c \ n \ (\text{Fun } f \ ss) \ t))$ 
by simp
        with t have Hf2:  $\neg (\forall t \in \text{set } ts. \text{fst } (rpo\text{-unbounded } ?pr \ c \ (\text{Fun } f \ ss) \ t))$ 
by auto
        show ?thesis by (simp add: oFalse oFalse2 Hf1 Hf2)
      qed
    next
      case True
        then have prns: prns and Hts:  $\forall t \in \text{set } ts. \text{fst } (rpo \text{ pr prl } c \ n \ (\text{Fun } f \ ss)$ 
t) by auto
        from Hts and t have Hts2:  $\forall t \in \text{set } ts. \text{fst } (rpo\text{-unbounded } ?pr \ c \ (\text{Fun } f$ 
ss) t) by auto
        show ?thesis
        proof (cases prs)
          case True then show ?thesis by (simp add: oFalse oFalse2 Hsns prns
Hts Hts2)
        next
          case False note prs = this
          show ?thesis
          proof (cases c (f,length ss) = c (g,length ts))
            case False then show ?thesis
            by (cases c (f,length ss), simp-all add: oFalse oFalse2 Hsns prns Hts
Hts2)
          next
            case True note cfg = this
            show ?thesis
            proof (cases c (f,length ss))
              case Mul note cf = this
              with cfg have cg:  $c (g, \text{length } ts) = \text{Mul}$  by simp
              {
                fix x y
                assume x-in-ss:  $x \in \text{set } ss$  and y-in-ts:  $y \in \text{set } ts$ 
                have rpo pr prl c n x y = rpo-unbounded ?pr c x y
                by (rule 4(4)[OF oFalse Hsns[symmetric] refl - prs - conjI[OF cf
cg] x-in-ss y-in-ts, rule-format],
                  insert prns Hts bs[OF x-in-ss] bt[OF y-in-ts] cf cg, auto)
              }
              with mul-ext-cong[of ss ss ts ts]
              have mul-ext (rpo pr prl c n) ss ts = mul-ext (rpo-unbounded ?pr
c) ss ts
              by metis

```

```

    then show ?thesis
      by (simp add: oFalse oFalse2 Hsns prns Hts Hts2 cf cg)
next
case Lex note cf = this
then have ncf: c (f,length ss) ≠ Mul by simp
from cf cfg have cg: c (g,length ts) = Lex by simp
{
  fix i
  assume iss: i < length ss and its: i < length ts
  from nth-mem-mset[OF iss] and in-multiset-in-set
  have in-ss: ss ! i ∈ set ss by force
  from nth-mem-mset[OF its] and in-multiset-in-set
  have in-ts: ts ! i ∈ set ts by force
  from 4(3)[OF oFalse Hsns[symmetric] refl - prs conjI[OF cf cg]
iss its]
    prns Hts bs[OF in-ss] bt[OF in-ts]
  have rpo pr prl c n (ss ! i) (ts ! i) = rpo-unbounded ?pr c (ss ! i)
(ts ! i)
    by simp
}
with lex-ext-cong[of ss ss n n ts ts]
have lex-ext (rpo pr prl c n) n ss ts
= lex-ext (rpo-unbounded ?pr c) n ss ts by metis
then have lex-ext (rpo pr prl c n) n ss ts = lex-ext-unbounded
(rpo-unbounded ?pr c) ss ts
by (simp add: lex-ext-to-lex-ext-unbounded[OF ss ts, of rpo-unbounded
?pr c])

    then show ?thesis
      by (simp add: oFalse oFalse2 Hsns prns prs Hts Hts2 cf cg)
qed
qed
qed
qed
qed
qed auto
}
then show ?thesis using assms by simp
qed
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)))
unfolding *rpo-mem-def* *rpo-main-def* *wpo-mem.simps* ..

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))
    )
  )
)
unfolding rpo-main-def rpo-mem-def wpo-main.simps Let-def if-False if-True
unfolding rpo-main-def[symmetric] rpo-mem-def[symmetric]
by (cases s; cases t, auto simp: map-nth split: prod.splits)

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))
  unfolding rpo-mem-def wpo-mem-impl-def[symmetric] wpo-ub-memoized-code[symmetric]
proof (induct pr c s t rule: rpo-unbounded.induct)
  case (1 pr c x y)
  then show ?case unfolding rpo-unbounded.simps wpo-ub.simps[of - - - - -
Var x Var y]
  by simp
next
  case (2 pr c x g ts)
  then show ?case unfolding rpo-unbounded.simps wpo-ub.simps[of - - - - -
Var x Fun g ts] term.simps
  by auto
next
  case (3 pr c f ss y)
  then show ?case unfolding rpo-unbounded.simps wpo-ub.simps[of - - - - -
Fun f ss Var y] term.simps
  Let-def by (auto simp: set-conv-nth)
next
  case (4 pr c f ss g ts)
  obtain prs prns where pr: fst pr (f, length ss) (g, length ts) = (prs,prns) by
force
  show ?case unfolding rpo-unbounded.simps wpo-ub.simps[of - - - - - Fun f
ss Fun g ts] term.simps
  if-False Let-def if-True pr split
  proof (rule sym, intro if-cong[OF - refl if-cong[OF - if-cong[OF refl refl] refl]],
goal-cases)
  case 1
  show ?case using 4(1) by (auto simp: set-conv-nth)
next
  case 2
  show ?case using 4(2)[unfolded pr, OF 2 refl] by (auto simp: set-conv-nth)

```

```

next
  case 3
  note IH = 4(3-)[unfolded pr, OF 3(1) refl 3(2-3)]
  let ?cf = c (f, length ss)
  let ?cg = c (g, length ts)
  consider (Lex) ?cf = Lex ?cg = Lex | (Mul) ?cf = Mul ?cg = Mul | (Diff)
?cf ≠ ?cg
  by (cases ?cf; cases ?cg, auto)
  thus ?case
  proof cases
  case Lex
  hence ?cf = ?cg and ?cf ≠ Mul by auto
  note IH = IH(2)[OF this]
  from Lex have id: (?cf = Lex ∧ ?cg = Lex) = True (?cf = ?cg) = True (?cf
= Mul) = False by auto
  show ?thesis unfolding id if-True if-False using IH
  by (intro lex-ext-unbounded-cong, auto intro: nth-equalityI)
  next
  case Mul
  hence ?cf = ?cg and ?cf = Mul by auto
  note IH = IH(1)[OF this]
  from Mul have id: (?cf = Lex ∧ ?cg = Lex) = False (?cf = Mul ∧ ?cg =
Mul) = True
  (?cf = ?cg) = True (?cf = Mul) = True by auto
  show ?thesis unfolding id(1-3) if-True if-False unfolding id(4) if-True
using IH
  by (intro mul-ext-cong[OF arg-cong[of - - mset] arg-cong[of - - mset]])
  (auto intro: nth-equalityI)
  qed auto
  qed
qed
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 Confer-*

ence), 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.