```
POLYNOM_2
---------
*C polynom_2_begin            ************ POLYNOM_2 ************
*C oalist_com_1
               ==============================
               ORDERED A-LIST TYPE DEFINITION
               ==============================
*D before_df   before{<a:a:*>}(<u:u:*>;<ps:ps:*>)== before{}(<a>; <u>; <ps>)
               before(<u:u:*>;<ps:ps:*>)== before{}(<a>; <u>; <ps>)
*A before                     before(u;ps) ==  null(ps) ∨_b(hd(ps) <_b u)
*T before_wf              1 2 ∀a:DSet. ∀ps:|a| List. ∀u:|a|.  before(u;ps) ∈ 𝔹
*T comb_for_before_wf     0 0
               (λa,ps,u,z.before(u;ps)) ∈ a:DSet → ps:|a| List → u:|a| → ↓True → 𝔹
*M before_eval let before_nilC =
                 MacroC `before_nilC`
                 (RepeatC (UnfoldsC ``before null``) ANDTHENC ReduceC)
                 ⌈before(u;[])⌉
                 IdC ⌈tt⌉;;
               let before_consC =
                 MacroC `befire_consC`
                 (RepeatC (UnfoldsC ``before null hd``) ANDTHENC ReduceC)
                 ⌈before(u;v::vs)⌉
                 IdC  ⌈v <_b u⌉;;
               add_AbReduce_conv `before`
                 (before_nilC ORELSEC before_consC);;
*T before_trans           2 2
               ∀a:LOSet. ∀u,v:|a|. ∀ws:|a| List.  v <a u ⇒ ↑before(v;ws) ⇒ ↑before(u;ws)
*C sd_ordered_com
               sd_ordered = s(tricly) d(escending) ordered
               Design choice here between boolean and prop
               valued definition. Went with bool valued since it
               trival then to prove decidable.
               Decided to define this with an auxiliary predicate
               `before' rather than stripping off two head elements
               and comparing them directly. `before'  makes
               one-cons unrolling of lists OK when reasoning with
               sd_ordered. It also hides the partial function `hd'.
*D sd_ordered_df
               sd_ordered{<s:s:*>}(<as:as:*>)== sd_ordered{}(<s>; <as>)
               sd_ordered(<as:as:*>)== sd_ordered{}(<s>; <as>)
*M sd_ordered_ml
               sd_ordered(as)
               ==r case as of [] => tt | a::bs => before(a;bs) ∧_b sd_ordered(bs) esac
*T sd_ordered_wf          2 0 ∀s:DSet. ∀as:|s| List.  sd_ordered(as) ∈ 𝔹
*T comb_for_sd_ordered_wf 0 0 (λs,as,z.sd_ordered(as)) ∈ s:DSet → as:|s| List → ↓True → 𝔹
*M sd_ordered_eval
               let sd_ordered_nilC =
                 MacroC `sd_ordered_nilC`
                  (RecUnfoldC `sd_ordered` ANDTHENC PrimReduceC) ⌈sd_ordered([])⌉
                  IdC ⌈tt⌉
               ;;
               let sd_ordered_consC =
                 MacroC `sd_ordered_consC`
                  (RecUnfoldC `sd_ordered` ANDTHENC PrimReduceC) ⌈sd_ordered(a::as)⌉
                  IdC ⌈before(a;as) ∧_b sd_ordered(as)⌉
               ;;
               add_AbReduce_conv `sd_ordered`
```

```
                  (sd_ordered_nilC ORELSEC sd_ordered_consC) ;;
*C sd_ordered_char_com
                Alternate characterization of
                stricly-descending order predicate.
                Probably, would have been easiest to use this definition
                from the start.
                The proof of this theorem shows how annoying the
                bool/prop difference is.
*T sd_ordered_char        4 5
                ∀s:QOSet. ∀us:|s| List.
                  sd_ordered(us) = HTFor{<𝔹,∧b>} v::vs ∈ us. ∀bw(:|s|) ∈ vs. w <b v
*T before_all_imp_count_zero 3 4
                ∀s:QOSet. ∀a:|s|. ∀cs:|s| List.  ↑(∀bc(:|s|) ∈ cs. c <b a) ⇒ a #∈ cs = 0
*T sd_ordered_count       3 4 ∀s:QOSet. ∀as:|s| List.  ↑sd_ordered(as) ⇒ (∀c:|s|. c #∈ as ≤ 1)
*C oalist_com  (strictly) o(rdered) a(ssociation) lists
                This is an experimental use of ‘set’
                constructors. Proofs are unlikely to be
                that clean.
                Old Definition:
                oal(a;b) ==  {ps:(a × {z:b↓set| ¬(z = e)}) List| ↑sd_ordered(map(λx.x.1;ps))}
                This gave slower Inclusion proofs because dset a × {z:b↓set| ¬(z = e)}
                exposed when type checking with funs typed over concrete Lists
*D oalist_df                   oal(<a:a:*>;<b:b:*>)== oalist{}(<a>; <b>)
*A oalist      oal(a;b) ==
                  {ps:(a × b↓set) List| ↑sd_ordered(map(λx.x.1;ps)) ∧ ¬↑(e ∈b map(λx.x.2;ps))}
*T oalist_wf              0 3 ∀a:LOSet. ∀b:AbMon.  oal(a;b) ∈ DSet
*M oalist_ml                   note_reduction_strength ‘oalist‘ ‘8‘;;
*T oalist_car_properties  0 0
                ∀a:LOSet. ∀b:AbMon. ∀ws:|oal(a;b)|.
                  ↑sd_ordered(map(λx.x.1;ws)) ∧ ¬↑(e ∈b map(λx.x.2;ws))
*T before_imp_before_all  3 4
                ∀a:LOSet. ∀b:AbMon. ∀k:|a|. ∀ps:|oal(a;b)|.
                  ↑before(k;map(λz.z.1;ps)) ⇒ ↑(∀bx(:|a|) ∈ map(λz.z.1;ps). x <b k)
*T before_all_imp_before  1 3
                ∀a:LOSet. ∀b:AbMon. ∀k:|a|. ∀ps:(|a| × |b|) List.
                  ↑(∀bx(:|a|) ∈ map(λz.z.1;ps). x <b k) ⇒ ↑before(k;map(λz.z.1;ps))
*T nil_in_oalist          2 2 ∀a:LOSet. ∀b:AbMon.  [] ∈ |oal(a;b)|
*T cons_in_oalist         3 5
                ∀a:LOSet. ∀b:AbMon. ∀ws:|oal(a;b)|. ∀x:|a|. ∀y:|b|.
                  ↑before(x;map(λx.x.1;ws)) ⇒ ¬(y = e) ⇒ (<x, y>::ws) ∈ |oal(a;b)|
*T cons_pr_in_oalist      2 5
                ∀a:LOSet. ∀b:AbMon. ∀ws:|oal(a;b)|. ∀x:|a|. ∀y:|b|.
                  ↑before(x;map(λx.x.1;ws)) ⇒ ¬(y = e) ⇒ (<x, y>::ws) ∈ |oal(a;b)|
*C oal_nil_cons_com
                =====================================
                NIL AND CONS CONSTRUCTORS FOR OALISTS
                =====================================
*D oal_nil_df  Parens ::Prec(preop)::  00<a:set:*>,<b:mon:*>== oal_nil{}(<a>; <b>)
                00== oal_nil{}(<a>; <b>)
*A oal_nil                     00 ==  []
*T oal_nil_wf             2 2 ∀a:LOSet. ∀b:AbMon.  00 ∈ |oal(a;b)|
*D oal_cons_pr_df
                oal_cons_pr(<x:x:*>;<y:y:*>;<ws:ws:*>)== oal_cons_pr{}(<x>; <y>; <ws>)
*A oal_cons_pr                 oal_cons_pr(x;y;ws) ==  <x, y>::ws
*T oal_cons_pr_wf         1 0
                ∀a:LOSet. ∀b:AbMon. ∀ws:|oal(a;b)|. ∀x:|a|. ∀y:|b|.
                  ↑before(x;map(λx.x.1;ws)) ⇒ ¬(y = e) ⇒ oal_cons_pr(x;y;ws) ∈ |oal(a;b)|
```

```
*C oal_case_ind
               ======================================
               OALIST CASE SPLIT AND INDUCTION LEMMAS
               ======================================
*C oalist_cases_com
               NB: it helps typechecking here to make Q's domain type
               larger than |oal(a;b)| (otherwise get extra obligations
               to show that subtype predicates are satisfied).
               With hindsight, it might have been cleaner to use oal_nil
               and define an oal_cons_pr constructor for use in the cases
               and induction lemmas. Then, the nil_in_oalist and cons_in_oalist wf lemmas would
                never have to be manually invoked.
               Making such a change would require updating all the MacroC
               conversion involving oalists.
*T oalist_cases           4 6
               ∀a:LOSet. ∀b:AbMon. ∀Q:(|a| × |b|) List → ℙ.
                 Q[[]]
                 ⇒ (∀ws:|oal(a;b)|. ∀x:|a|. ∀y:|b|.
                       ↑before(x;map(λx.x.1;ws)) ⇒ ¬(y = e) ⇒ Q[<x, y>::ws])
                 ⇒ {∀ws:|oal(a;b)|. Q[ws]}
*T oalist_cases_a         4 6
               ∀a:LOSet. ∀b:AbMon. ∀Q:|oal(a;b)| → ℙ.
                 Q[[]]
                 ⇒ (∀ws:|oal(a;b)|. ∀x:|a|. ∀y:|b|.
                       ↑before(x;map(λx.x.1;ws)) ⇒ ¬(y = e) ⇒ Q[<x, y>::ws])
                 ⇒ {∀ws:|oal(a;b)|. Q[ws]}
*T oalist_cases_c         0 0
               ∀a:LOSet. ∀b:AbMon. ∀Q:|oal(a;b)| → ℙ.
                 Q[00]
                 ⇒ (∀ws:|oal(a;b)|. ∀x:|a|. ∀y:|b|.
                       ↑before(x;map(λx.x.1;ws)) ⇒ ¬(y = e) ⇒ Q[oal_cons_pr(x;y;ws)])
                 ⇒ {∀ws:|oal(a;b)|. Q[ws]}
*T oalist_cases_b         2 4
               ∀a:LOSet. ∀b:AbMon. ∀Q:|oal(a;b)| → ℙ.
                 Q[[]]
                 ⇒ (∀ws:|oal(a;b)|. ∀k:|a|. ∀v:|b|.
                       ↑(∀_b x(:|a|) ∈ map(λz.z.1;ws). x <_b k) ⇒ ¬(v = e) ⇒ Q[<k, v>::ws])
                 ⇒ {∀ws:|oal(a;b)|. Q[ws]}
*T oalist_ind             2 6
               ∀a:LOSet. ∀b:AbMon. ∀Q:(|a| × |b|) List → ℙ.
                 Q[[]]
                 ⇒ (∀ws:|oal(a;b)|
                       Q[ws]
                       ⇒ (∀x:|a|. ∀y:|b|.
                             ↑before(x;map(λx.x.1;ws)) ⇒ ¬(y = e) ⇒ Q[<x, y>::ws]))
                 ⇒ {∀ws:|oal(a;b)|. Q[ws]}
*T oalist_ind_a           3 6
               ∀a:LOSet. ∀b:AbMon. ∀Q:|oal(a;b)| → ℙ.
                 Q[[]]
                 ⇒ (∀ws:|oal(a;b)|
                       Q[ws]
                       ⇒ (∀x:|a|. ∀y:|b|.
                             ↑before(x;map(λx.x.1;ws)) ⇒ ¬(y = e) ⇒ Q[<x, y>::ws]))
                 ⇒ {∀ws:|oal(a;b)|. Q[ws]}
*T list_pr_length_ind     3 4
               ∀T:𝕌. ∀Q:T List → T List → ℙ.
                 (∀ps,qs:T List.
```

```
                    (∀us,vs:T List.  ||us|| + ||vs|| < ||ps|| + ||qs|| ⇒ Q[us;vs])
                    ⇒ Q[ps;qs])
                 ⇒ {∀ps,qs:T List.  Q[ps;qs]}
*T oalist_pr_length_ind   3 6
               ∀a:LOSet. ∀b:AbMon. ∀Q:(|a| × |b|) List → (|a| × |b|) List → ℙ.
                 (∀ps,qs:|oal(a;b)|.
                    (∀us,vs:|oal(a;b)|.  ||us|| + ||vs|| < ||ps|| + ||qs|| ⇒ Q[us;vs])
                    ⇒ Q[ps;qs])
                 ⇒ {∀ps,qs:|oal(a;b)|.  Q[ps;qs]}
*C oal_fun_fin_sup_char
               ==============================
               CHARACTERIZATION OF OALISTS AS
               FUNCTIONS OF FINITE SUPPORT
               ==============================
               It seems far more elegant to prove the
               algebraic properties of oalists and functions
               over them by using this characterization, rather
               than by plowing through inductive proofs.
*D oal_dom_df  dom{<a:oset:*>,<b:mon:*>}(<ps:oal:*>)== oal_dom{}(<a>; <b>; <ps>)
               dom(<ps:oal:*>)== oal_dom{}(<a>; <b>; <ps>)
*C oal_dom_com The type arguments are needed here to help with
               type checking (e.g. consider dom([])
               and provide arguments for rhs of eval rw rules.
*A oal_dom                    dom(ps) ==  mk_mset(map(λz.z.1;ps))
*T oal_dom_wf              0 1 ∀a:LOSet. ∀b:AbMon. ∀ps:(|a| × |b|) List.  dom(ps) ∈ MSet{a}
*T oal_dom_wf2             2 3 ∀a:LOSet. ∀b:AbMon. ∀ps:|oal(a;b)|.  dom(ps) ∈ FSet{a}
*M oal_dom_eval
               let oal_dom_nilC =
                 MacroC ‘oal_dom_nilC‘
                   (EvalC ‘‘oal_dom mk_mset‘‘)
                   ⌜dom([])⌝
                   (UnfoldC ‘null_mset‘)
                   ⌜0{a}⌝
               ;;
               let oal_dom_cons_prC =
                 MacroC ‘oal_dom_cons_prC‘
                   (EvalC ‘‘oal_dom mk_mset‘‘)
                   ⌜dom(<k, v>::ps)⌝
                   (EvalC ‘‘oal_dom mk_mset mset_inj mset_sum‘‘)
                   ⌜mset_inj{a}(k) + dom(ps)⌝
               ;;
               add_AbReduce_conv ‘oal_dom‘
                 (oal_dom_nilC ORELSEC oal_dom_cons_prC)
               ;;
*M oal_dom_eval2              DCL: oal_dom_nil:  dom(00) ∼ 0{s}
*C lookup_com  The lookup function defines the ‘function of
               finite support’ that oalists represent. Most
               properties of functions on oalists are most easily
               proven with the help of this function.
*D lookup_df   <as:as:E>[<k:k:*>]{<s:s:*>,<z:z:*>}== lookup{}(<s>; <z>; <k>; <as>)
               Parens ::Prec(postop)::  <as:as:E>[<k:k:*>]== lookup{}(<s>; <z>; <k>; <as>)
*M lookup_ml   as[k]
               ==r case as of
                     [] => z
                     b::bs => let <bk,bv> = b in if bk =_b k then bv else bs[k] fi
                   esac
*T lookup_wf              1 1 ∀a:PosetSig. ∀B:𝕌. ∀z:B. ∀k:|a|. ∀xs:(|a| × B) List.  xs[k] ∈ B
```

```
*T comb_for_lookup_wf     0 0
              (λa,B,z,k,xs,z1.xs[k]) ∈ a:PosetSig
                                         → B:𝕌
                                         → z:B
                                         → k:|a|
                                         → xs:(|a| × B) List
                                         → ↓True
                                         → B
*M lookup_eval let lookup_nilC =
                MacroC `lookup_nilC`
                (RecUnfoldTopC `lookup` ANDTHENC ReduceC)
                ⌈[][k]⌉
                IdC ⌈z⌉
              ;;
              let lookup_cons_prC =
                MacroC `lookup_cons_prC`
                (RecUnfoldTopC `lookup` ANDTHENC ReduceC)
                 ⌈(<a, b>::cs)[k]⌉
                 IdC ⌈if a =_b k then b else cs[k] fi ⌉
              ;;
              add_AbReduce_conv `lookup`
                (lookup_cons_prC ORELSEC lookup_nilC) ;;
              % would be interesting to see if ifthenelse makes this work
                any faster than if matching used directly.
              %
              let lookup_evalC e t =
                if is_term `lookup` t then
                  (lookup_nilC
                   ORELSEC
                   (ITECondC lookup_cons_prC (RelRST THEN Auto))
                  ) e t
                 else
                   failwith `lookup_evalC`
              ;;
*M lookup_oal_eval            DCL: lookup_oal_nil:  00[k] ∼ z
*T lookup_fails           3 4
              ∀a:DSet. ∀B:𝕌. ∀z:B. ∀k:|a|. ∀ps:(|a| × B) List.
                ¬↑(k ∈_b map(λx.x.1;ps)) ⇒ ps[k] = z
*T lookup_non_zero        4 5
              ∀a:LOSet. ∀b:AbMon. ∀k:|a|. ∀ps:|oal(a;b)|.  ¬(ps[k] = e) ⟺ ↑(k ∈_b dom(ps))
*T lookup_oal_eq_id       1 2
              ∀a:LOSet. ∀b:AbMon. ∀k:|a|. ∀ps:|oal(a;b)|.  ¬↑(k ∈_b dom(ps)) ⇒ ps[k] = e
*T lookup_oal_cons        2 4
              ∀a:LOSet. ∀b:OCMon. ∀k,kp:|a|. ∀vp:|b|. ∀ps:|oal(a;b)|.
                ↑before(kp;map(λz.z.1;ps)) ⇒ (<kp, vp>::ps)[k] = (when kp =_b k. vp) * ps[k]
*C lookup_before_start_com
              lookup_before_start_a was
              a lot easier to prove. Shows the benefit
              of using a 'stronger' notion of beforeness.
*T lookup_before_start    3 6
              ∀a:LOSet. ∀b:AbMon. ∀k:|a|. ∀ps:|oal(a;b)|.
                ↑before(k;map(λz.z.1;ps)) ⇒ ps[k] = e
*T lookup_before_start_a  2 4
              ∀a:QOSet. ∀b:AbMon. ∀k:|a|. ∀ps:(|a| × |b|) List.
                ↑(∀_bk'(:|a|) ∈ map(λz.z.1;ps). k' <_b k) ⇒ ps[k] = e
*T lookups_same           5 7
              ∀a:LOSet. ∀b:AbMon. ∀ps,qs:|oal(a;b)|.  (∀u:|a|. ps[u] = qs[u]) ⇒ ps = qs
```

```
*T lookups_same_a         2 3
              ∀a:LOSet. ∀b:AbMon. ∀ps,qs:|oal(a;b)|.  (∀u:|a|. ps[u] = qs[u]) ⇒ ps = qs
*T oal_equal_char         1 3
              ∀a:LOSet. ∀b:AbMon. ∀ps,qs:|oal(a;b)|.  ps = qs ⟺ (∀u:|a|. ps[u] = qs[u])
*C oal_merge_com_1
              =====================
              OALIST MERGE FUNCTION
              =====================
*C oal_merge_com
              Comments on this definition and the wf goal:
              1. AbMonoid typing not necessary. However, the
                RepSplitITE tactic calls bool_to_propC which in
                turn invokes the lemma `assert_of_mon_eq` which has
                the monoid assumption. Would be messier to try to disable
                this action.
              2. The destructor style definition is necessary here. The
                wf goal cannot be proven if a constructor style
                definition is used. (The problem is that the left list
                decomp rule doesn't do substitutions in the hyp list.)
              3. Arith reasoning needs patching up to get rid of
                 clumsy need for `pos_length` lemmas.
              4. Would HO matching take care of cases in wf lemma
                 where IH has to be explicitly instantiated?
*D oal_merge_df
              Parens ::Prec(inop)::
                <ps:ps:L> ++<a:a:*>,<b:b:L> <qs:qs:E>
                == oal_merge{}(<a>; <b>; <ps>; <qs>)
              Parens ::Prec(inop)::
                <ps:ps:L> ++ <qs:qs:E>
                == oal_merge{}(<a>; <b>; <ps>; <qs>)
*M oal_merge_ml
              ps ++ qs
              ==r if null(ps) then qs
                  if null(qs) then ps
                  if hd(qs).1 <_b hd(ps).1 then hd(ps)::(tl(ps) ++ qs)
                  if hd(ps).1 <_b hd(qs).1 then hd(qs)::(ps ++ tl(qs))
                  if (hd(ps).2 * hd(qs).2) =_b e then tl(ps) ++ tl(qs)
                  else <hd(ps).1, hd(ps).2 * hd(qs).2>::(tl(ps) ++ tl(qs))
                  fi
*M oal_merge_eval
              let oal_merge_left_nilC =
                MacroC `oal_merge_left_nilC`
                (RecUnfoldC `oal_merge` ANDTHENC ReduceC)
                ⌜[] ++ qs⌝
                IdC
                 ⌜qs⌝ ;;
              let oal_merge_right_nilC =
                MacroC `oal_merge_right_nilC`
                (RecUnfoldC `oal_merge` ANDTHENC ReduceC)
                ⌜(p::ps) ++ []⌝
                IdC
                 ⌜p::ps⌝ ;;
              let oal_merge_consesC =
                MacroC `oal_merge_consesC`
                (RecUnfoldC `oal_merge` ANDTHENC ReduceC)
                ⌜(<kp, vp>::ps) ++ (<kq, vq>::qs)⌝
                IdC
```

```
                ⌈if kq <_b kp then <kp, vp>::(ps ++ (<kq, vq>::qs))
                 if kp <_b kq then <kq, vq>::((<kp, vp>::ps) ++ qs)
                 if (vp * vq) =_b e then ps ++ qs
                 else <kp, vp * vq>::(ps ++ qs)
                 fi ⌉ ;;
              add_AbReduce_conv `oal_merge`
                (FirstC [oal_merge_left_nilC
                        ;oal_merge_right_nilC
                        ;oal_merge_consesC
                        ]) ;;
              % tries to make as much headway as poss on ifthenelses %
              let oal_merge_evalC e t =
                if is_term `oal_merge` t then
                 (FirstC
                   [oal_merge_left_nilC
                   ;oal_merge_right_nilC
                   ;oal_merge_consesC
                    ANDTHENC RepeatC (ITECondC IdC (RelRST THEN Auto))
                   ]) e t
                else
                  failwith `oal_merge_evalC` ;;
*T oal_merge_wf           4 7
              ∀a:LOSet. ∀b:AbMon. ∀ps,qs:(|a| × |b|) List.  ps ++ qs ∈ (|a| × |b|) List
*T oal_merge_dom_pred     5 7
              ∀a:LOSet. ∀b:AbMon. ∀Q:|a| → 𝔹. ∀ps,qs:(|a| × |b|) List.
                ↑(∀_b x(:|a|) ∈ map(λx.x.1;ps). Q[x])
                ⇒ ↑(∀_b x(:|a|) ∈ map(λx.x.1;qs). Q[x])
                ⇒ ↑(∀_b x(:|a|) ∈ map(λx.x.1;ps ++ qs). Q[x])
*T oal_dom_merge          3 5
              ∀a:LOSet. ∀b:AbMon. ∀ps,qs:|oal(a;b)|.  ↑(dom(ps ++ qs) ⊆_b dom(ps) ∪ dom(qs))
*C oal_merge_sd_ordered_com
              Notes on proof:
              1. Includes a couple of examples of monotonicity reasoning.
              2. Should assert(ball...) be taken care of by bool_to_propC?
              3. Induction delicate, because have to keep around some unreduced
                 sd_ordered predicates.
                 If unrolling of recursive definitions were to be driven by
                 destructors (such as of oal_merge by hd and tl) then automation
                 would be more straightforward. I guess this happens in NQTHM.
*T oal_merge_sd_ordered   5 7
              ∀a:LOSet. ∀b:AbMon. ∀ps,qs:(|a| × |b|) List.
                ↑sd_ordered(map(λx.x.1;ps))
                ⇒ ↑sd_ordered(map(λx.x.1;qs))
                ⇒ ↑sd_ordered(map(λx.x.1;ps ++ qs))
*T oal_merge_non_id_vals  5 8
              ∀a:LOSet. ∀b:AbMon. ∀ps,qs:(|a| × |b|) List.
                ¬↑(e ∈_b map(λx.x.2;ps))
                ⇒ ¬↑(e ∈_b map(λx.x.2;qs))
                ⇒ ¬↑(e ∈_b map(λx.x.2;ps ++ qs))
*T lookup_merge           5 8
              ∀a:LOSet. ∀b:AbMon. ∀k:|a|. ∀ps,qs:|oal(a;b)|.  (ps ++ qs)[k] = ps[k] * qs[k]
*T oal_merge_wf2          2 3 ∀a:LOSet. ∀b:AbMon. ∀ps,qs:|oal(a;b)|.  ps ++ qs ∈ |oal(a;b)|
*C oal_mon_com =======================
              OALIST MONOID DEFINITION
              ========================
*T oal_nil_ident_r        3 4 ∀a:LOSet. ∀b:AbMon. ∀ps:|oal(a;b)|.  ps ++ 00 = ps
*T oal_nil_ident_l        1 2 ∀a:LOSet. ∀b:AbMon. ∀ps:|oal(a;b)|.  00 ++ ps = ps
```

```
*T oal_merge_comm         2 3 ∀a:LOSet. ∀b:AbMon. ∀ps,qs:|oal(a;b)|.  ps ++ qs = qs ++ ps
*T oal_merge_assoc        2 3
              ∀a:LOSet. ∀b:AbMon. ∀ps,qs,rs:|oal(a;b)|.  ps ++ qs ++ rs = (ps ++ qs) ++ rs
*D oal_mon_df                 oal_mon(<a:a:*>;<b:b:*>)== oal_mon{}(<a>; <b>)
*A oal_mon                    oal_mon(a;b) ==  <|oal(a;b)|, =_b, λx,y.tt, λx,y.x ++ y, 00, λx.x>
*T oal_mon_wf             3 2 ∀a:LOSet. ∀b:AbMon.  oal_mon(a;b) ∈ AbMon
*M oal_mon_ml  % Conversions for folding up oal_mon components %
              let oal_monC,rem_oal_monC =
                let cprs =
                 map (\t,t'. DoubleMacroC `oal_monC` IdC t (ForceReduceC `5`) t')
               [⌈|oal(s;g)|⌉,⌈|oal_mon(s;g)|⌉
               ;⌈ps ++ qs⌉,⌈ps * qs⌉
               ;⌈00⌉,⌈e⌉
               ]
                in
                  FirstC (map fst cprs),FirstC (map snd cprs)
              ;;
*C oal_inj_com ======================
              INJECTION INTO OALISTS
              ======================
*D oal_inj_df  inj{<a:a:*>,<b:b:*>}(<k:k:*>,<v:v:*>)== oal_inj{}(<a>; <b>; <k>; <v>)
              inj(<k:k:*>,<v:v:*>)== oal_inj{}(<a>; <b>; <k>; <v>)
*A oal_inj                    inj(k,v) ==  if v =_b e then [] else <k, v>::[] fi
*T oal_inj_wf             3 4 ∀a:LOSet. ∀b:AbMon. ∀k:|a|. ∀v:|b|.  inj(k,v) ∈ |oal(a;b)|
*T comb_for_oal_inj_wf    0 0
              (λa,b,k,v,z.inj(k,v)) ∈ a:LOSet
                                      → b:AbMon
                                      → k:|a|
                                      → v:|b|
                                      → ↓True
                                      → |oal(a;b)|
*T lookup_oal_inj         2 4
              ∀a:LOSet. ∀b:AbMon. ∀k,k':|a|. ∀v:|b|.  inj(k,v)[k'] = when k =_b k'. v
*T oal_dom_inj            2 3
              ∀a:LOSet. ∀b:AbMon. ∀k:|a|. ∀v:|b|.
                dom(inj(k,v)) = if v =_b e then 0{a} else mset_inj{a}(k) fi
*T oalist_fact            4 6
              ∀a:LOSet. ∀b:AbMon. ∀ps:|oal(a;b)|.
                ps = msFor{oal_mon(a;b)} k' ∈ dom(ps). inj(k',ps[k'])
*C oal_neg_com ===============
              OALIST NEGATION
              ===============
              Definition & characterization of
              inverse function for oalist group.
*D oal_neg_df  Parens ::Prec(preop)::  --<a:a:*>,<b:b:*> <ps:ps:E>== oal_neg{}(<a>; <b>; <ps>)
              Parens ::Prec(preop)::  --<ps:ps:E>== oal_neg{}(<a>; <b>; <ps>)
*A oal_neg                    --ps ==  map(λkv.<kv.1, ~ kv.2>;ps)
*T oal_neg_wf             0 0
              ∀a:PosetSig. ∀b:GrpSig. ∀ps:(|a| × |b|) List.  --ps ∈ (|a| × |b|) List
*M oal_neg_eval
              let oal_neg_nilC =
                MacroC `oal_neg_nilC`
                (EvalC ``oal_neg``)
                ⌈--[]⌉
                IdC
                 ⌈[]⌉
              ;;
```

```
               let oal_neg_cons_prC =
                 MacroC `oal_neg_cons_prC`
                  (EvalC ``oal_neg``)
                  ⌜--(<k, v>::ps)⌝
                  (UnfoldC `oal_neg`)
                  ⌜<k, ∼ v>::(--ps)⌝
               ;;
               add_AbReduce_conv `oal_neg`
                 (oal_neg_nilC ORELSEC oal_neg_cons_prC)
               ;;
*T oal_neg_keys_invar     1 1
               ∀a:PosetSig. ∀b:GrpSig. ∀ps:(|a| × |b|) List.  map(λz.z.1;--ps) = map(λz.z.1;ps)
*T oal_neg_sd_ordered     1 2
               ∀a:LOSet. ∀b:AbMon. ∀ps:(|a| × |b|) List.
                 ↑sd_ordered(map(λx.x.1;ps)) ⇒ ↑sd_ordered(map(λx.x.1;--ps))
*T oal_neg_non_id_vals    3 4
               ∀a:LOSet. ∀b:AbGrp. ∀ps:(|a| × |b|) List.
                 ¬↑(e ∈b map(λx.x.2;ps)) ⇒ ¬↑(e ∈b map(λx.x.2;--ps))
*T oal_neg_wf2            2 3 ∀a:LOSet. ∀b:AbGrp. ∀ps:|oal(a;b)|.  --ps ∈ |oal(a;b)|
*T lookup_oal_neg         2 3
               ∀a:DSet. ∀b:IGroup. ∀k:|a|. ∀ps:(|a| × |b|) List.  (--ps)[k] = ∼ ps[k]
*T oal_dom_neg            3 3 ∀a:LOSet. ∀b:AbGrp. ∀ps:|oal(a;b)|.  dom(--ps) = dom(ps)
*T oal_neg_left_inv       2 3 ∀a:LOSet. ∀b:AbGrp. ∀ps:|oal(a;b)|.  --ps ++ ps = 00
*T oal_neg_right_inv      1 2 ∀a:LOSet. ∀b:AbGrp. ∀ps:|oal(a;b)|.  ps ++ --ps = 00
*T oal_neg_eq_nil         2 2 ∀a:LOSet. ∀b:AbGrp. ∀ps:|oal(a;b)|.  --ps = 00 ⟺ ps = 00
*C oal_lv_and_lk_funs
               ==========================================
               LEADING KEY AND VALUE FUNCTIONS FOR OALISTS
               ==========================================
*C oal_null_com
               With most ps can infer s and g, but
               put args in, just in case get [] list
               before get chance for reduction.
               (e.g. from oal_cases)
*D oal_null_df null{<s:s:*>,<g:g:*>}(<ps:ps:*>)== oal_null{}(<s>; <g>; <ps>)
               null(<ps:ps:*>)== oal_null{}(<s>; <g>; <ps>)
*A oal_null                   null(ps) ==  null(ps)
*T oal_null_wf            0 2 ∀s:LOSet. ∀g:AbMon. ∀ps:|oal(s;g)|.  null(ps) ∈ 𝔹
*T assert_of_oal_null     3 3 ∀s:LOSet. ∀g:AbMon. ∀ps:|oal(s;g)|.  ↑null(ps) ⟺ ps = 00
*M oal_null_ml add_reducible_ab `oal_null` ;;
               update_assert_elim_lemmas ``assert_of_oal_null``;;
*C oal_lk_com  lk = l(eading) k(ey)
               lv = l(eading) v(alue)
*D oal_lk_df                  lk(<ps:ps:*>)== oal_lk{}(<ps>)
*A oal_lk                     lk(ps) ==  hd(ps).1
*T oal_lk_wf              2 4 ∀s:LOSet. ∀g:AbMon. ∀ps:|oal(s;g)|.  ¬(ps = 00) ⇒ lk(ps) ∈ |s|
*T oal_lk_in_dom          3 4
               ∀s:LOSet. ∀g:AbMon. ∀ps:|oal(s;g)|.  ¬(ps = 00) ⇒ ↑(lk(ps) ∈b dom(ps))
*T oal_lk_bounds_dom      3 5
               ∀s:LOSet. ∀g:AbMon. ∀k:|s|. ∀ps:|oal(s;g)|.
                 ¬(ps = 00) ⇒ ↑(k ∈b dom(ps)) ⇒ k ≤ lk(ps)
*M oal_lk_eval let oal_lk_cons_prC =
                 MacroC `oal_lk_cons_prC`
                   (EvalC ``oal_lk``)
                   ⌜lk(<k, v>::ps)⌝
                   IdC ⌜k⌝
               ;;
```

```
              add_AbReduce_conv `oal_lk`
                oal_lk_cons_prC ;;
*D oal_lv_df                 lv(<ps:ps:*>)== oal_lv{}(<ps>)
*A oal_lv                    lv(ps) ==  hd(ps).2
*T oal_lv_wf             2 4 ∀s:LOSet. ∀g:AbMon. ∀ps:|oal(s;g)|.  ¬(ps = 00) ⇒ lv(ps) ∈ |g|
*T oal_lv_nid            2 3 ∀s:LOSet. ∀g:AbMon. ∀ps:|oal(s;g)|.  ¬(ps = 00) ⇒ ¬(lv(ps) = e)
*M oal_lv_eval let oal_lv_cons_prC =
                MacroC `oal_lv_cons_prC`
                  (EvalC ``oal_lv``)
                   ⌈lv(<k, v>::ps)⌉
                  IdC ⌈v⌉
              ;;
              add_AbReduce_conv `oal_lv`
                oal_lv_cons_prC ;;
*T oal_lk_merge_1        4 6
              ∀s:LOSet. ∀g:AbMon. ∀ps,qs:|oal(s;g)|.
                ¬(ps = 00)
                ⇒ ¬(qs = 00)
                ⇒ ¬(ps ++ qs = 00)
                ⇒ lk(ps) <s lk(qs)
                ⇒ lk(ps ++ qs) = lk(qs)
*T oal_lk_merge_2        4 6
              ∀s:LOSet. ∀g:AbMon. ∀ps,qs:|oal(s;g)|.
                ¬(ps = 00)
                ⇒ ¬(qs = 00)
                ⇒ ¬(ps ++ qs = 00)
                ⇒ lk(ps) = lk(qs)
                ⇒ ¬(lv(ps) * lv(qs) = e)
                ⇒ lk(ps ++ qs) = lk(qs)
*C oal_lk_neg_com
              With partial functions there is always the question
              of how explicit to make all the totality guaranteeiing
              conditions. Here at one point, have to check that
              oal_neg(ps) is not nil because it occurs as arg to oal_lk.
*T oal_lk_neg            3 4
              ∀s:LOSet. ∀g:AbGrp. ∀ps:|oal(s;g)|.  ¬(ps = 00) ⇒ lk(--ps) = lk(ps)
*T lookup_oal_lk         2 4
              ∀s:LOSet. ∀g:AbMon. ∀ps:|oal(s;g)|.  ¬(ps = 00) ⇒ ps[lk(ps)] = lv(ps)
*T oal_lv_neg            2 4
              ∀s:LOSet. ∀g:AbGrp. ∀ps:|oal(s;g)|.  ¬(ps = 00) ⇒ lv(--ps) = ∼ lv(ps)
*C ocgrp_constr
              ====================================
              CONSTRUCTION OF ORDER REL ON OALISTS
              ====================================
              The simplest order (a lexicographic order)
              is induced on oalists by orders on the
              key and value domains of oalists.
              For the purposes of defining this order,
              it is convenient to assume that the value
              domain is an abelian group rather than an
              abelian monoid.
*D oal_bpos_df pos{<s:s:*>,<g:g:*>}(<ps:ps:*>)== oal_bpos{}(<s>; <g>; <ps>)
              pos(<ps:ps:*>)== oal_bpos{}(<s>; <g>; <ps>)
*A oal_bpos                  pos(ps) ==  ¬_b null(ps) ∧_b e <_b lv(ps)
*T oal_bpos_wf           2 3 ∀s:LOSet. ∀g:AbMon. ∀ps:|oal(s;g)|.  pos(ps) ∈ 𝔹
*T comb_for_oal_bpos_wf   0 0
              (λs,g,ps,z.pos(ps)) ∈ s:LOSet → g:AbMon → ps:|oal(s;g)| → ↓True → 𝔹
```

```
*D oal_blt_df  Parens ::Prec(inop)::
                 <ps:ps:L> <<b<s:s:L>,<g:g:L> <qs:qs:L>
                 == oal_blt{}(<s>; <g>; <ps>; <qs>)
               Parens ::Prec(inop)::  <ps:ps:L> <<b <qs:qs:L>== oal_blt{}(<s>; <g>; <ps>; <qs>)
*A oal_blt                    ps <<b qs ==  pos(qs ++ --ps)
*T oal_blt_wf             0 2 ∀s:LOSet. ∀g:AbGrp. ∀ps,qs:|oal(s;g)|.  ps <<b qs ∈ 𝔹
*D oal_ble_df  <ps:ps:L> ≤≤b<s:s:L>,<g:g:L> <qs:qs:L>== oal_ble{}(<s>; <g>; <ps>; <qs>)
               <ps:ps:L> ≤≤b <qs:qs:L>== oal_ble{}(<s>; <g>; <ps>; <qs>)
*A oal_ble                    ps ≤≤b qs ==  (ps =b qs) ∨b(ps <<b qs)
*T oal_ble_wf             0 2 ∀s:LOSet. ∀g:AbGrp. ∀ps,qs:|oal(s;g)|.  ps ≤≤b qs ∈ 𝔹
*D oal_le_df   Parens ::Prec(atomrel)::
                 <ps:ps:L> ≤{<s:s:*>,<g:g:*>} <qs:qs:L>
                 == oal_le{}(<s>; <g>; <ps>; <qs>)
*A oal_le                     ps ≤{s,g} qs ==  ↑ps ≤≤b qs
*T oal_le_wf              0 2 ∀s:LOSet. ∀g:AbGrp. ∀ps,qs:|oal(s;g)|.  (ps ≤{s,g} qs) ∈ ℙ1
*C oal_grp_com ============================
               INTRODUCTION OF OALIST GROUP
               ============================
               This is not ideally placed. Definition had to wait
               for introduction of order function.
               However, characterization as ordered group comes later.
*D oal_grp_df                 oal_grp(<s:s:*>;<g:g:*>)== oal_grp{}(<s>; <g>)
*A oal_grp     oal_grp(s;g) ==  <|oal(s;g)|, =b, λx,y.x ≤≤b y, λx,y.x ++ y, 00, λx.--x>
*T oal_grp_wf             3 4 ∀s:LOSet. ∀g:AbGrp.  oal_grp(s;g) ∈ AbGrp
*M oal_grp_ml  % Conversions for folding up oal_grp components %
               let oal_grpC,rem_oal_grpC =
                 let cprs =
                  map (\t,t'. DoubleMacroC `oal_grpC` IdC t (ForceReduceC `5`) t')
                [⌈|oal(s;g)|⌉,⌈|oal_grp(s;g)|⌉
                ;⌈ps ++ qs⌉,⌈ps * qs⌉
                ;⌈--ps⌉,⌈∼ ps⌉
                ;⌈00⌉,⌈e⌉
                ]
                 in
                   FirstC (map fst cprs),FirstC (map snd cprs)
               ;;
               let oal_grp_leC =
                 MacroC `oal_grp_leC`
                   (UnfoldC `oal_le`) ⌈ps ≤{s,g} qs⌉
                   (EvalC ``grp_leq``) ⌈ps ≤ qs⌉
               ;;
               let WithOalAsGrp T i =
                 RWD oal_grpC i
                 THENM T i
                 THENM RWD rem_oal_grpC i
               ;;
*D oal_lt_df   Parens ::Prec(atomrel)::
                 <ps:ps:L> <<<s:s:*>,<g:g:*> <qs:qs:L>
                 == oal_lt{}(<s>; <g>; <ps>; <qs>)
               Parens ::Prec(atomrel)::
                 <ps:ps:L> << <qs:qs:L>
                 == oal_lt{}(<s>; <g>; <ps>; <qs>)
*A oal_lt      ps << qs ==  ∃k:|s|. (∀k':|s|. k <s k' ⇒ ps[k'] = qs[k']) ∧ ps[k] < qs[k]
*T oal_lt_wf              0 3 ∀s:LOSet. ∀g:OCMon. ∀ps,qs:|oal(s;g)|.  (ps << qs) ∈ ℙ
*T decidable__oal_lt      0 2 ∀s:LOSet. ∀g:OGrp. ∀ps,qs:|oal(s;g)|.  Dec(ps << qs)
*T assert_of_oal_blt      4 7 ∀s:LOSet. ∀g:OGrp. ∀ps,qs:|oal(s;g)|.  ↑(ps <<b qs) ⟺ ps << qs
*M assert_of_oal_blt_ml       update_assert_elim_lemmas ``assert_of_oal_blt`` ;;
```

```
*T oal_lt_irrefl          2 2 ∀s:LOSet. ∀g:OCMon.  Irrefl(|oal(s;g)|;ps,qs.ps << qs)
*T oal_lt_trans           3 6 ∀s:LOSet. ∀g:OCMon.  Trans(|oal(s;g)|;ps,qs.ps << qs)
*T oal_bpos_trichot       3 5
              ∀s:LOSet. ∀g:OGrp. ∀rs:|oal(s;g)|.  ↑pos(rs) ∨ rs = 00 ∨ ↑pos(--rs)
*T oal_lt_trichot         3 5
              ∀s:LOSet. ∀g:OGrp. ∀ps,qs:|oal(s;g)|.  ps << qs ∨ ps = qs ∨ qs << ps
#T oal_lt_trichot_a       3 5
              ∀s:LOSet. ∀g:OGrp. ∀ps,qs:|oal(s;g)|.  ps << qs ∨ ps = qs ∨ qs << ps
*T oal_merge_preserves_lt 2 5
              ∀s:LOSet. ∀g:OCMon. ∀ps,qs,rs:|oal(s;g)|.  qs << rs ⇒ ps ++ qs << ps ++ rs
*M oal_le_order_decl
              Relation Family
               <: ps << qs
               ≤: ps ≤{s,g} qs
               ≡: ps = qs
               ≥: ?
               >: ?
              add_lin_order_check_fun `oal_lt` (\p r.true) ;;
*T oal_le_char            2 3
              ∀s:LOSet. ∀g:OGrp.
                (ps,qs:|oal(s;g)|. ps ≤{s,g} qs) <≡>{|oal(s;g)|} (ps,qs:|oal(s;g)|. ps << qs)⁰
*T oal_lt_char            3 4
              ∀s:LOSet. ∀g:OGrp.
                (ps,qs:|oal(s;g)|. ps << qs) <≡>{|oal(s;g)|} (ps,qs:|oal(s;g)|. ps ≤{s,g} qs)\
*T oal_le_is_order        3 3 ∀s:LOSet. ∀g:OGrp.  Order(|oal(s;g)|;ps,qs.ps ≤{s,g} qs)
*T oal_le_connex          2 3 ∀s:LOSet. ∀g:OGrp.  Connex(|oal(s;g)|;ps,qs.ps ≤{s,g} qs)
*T oal_grp_wf1            2 2 ∀s:LOSet. ∀g:OGrp.  oal_grp(s;g) ∈ OMon
*T oal_lt_iff_grp_lt      2 4 ∀s:LOSet. ∀g:OGrp. ∀ps,qs:|oal(s;g)|.  ps << qs ⟺ ps < qs
*C oal_grp_wf2_com
              subtype reasoning needs looking at here.
              The identification of the extra properties of
              oal_grp that have to be proven should not
              depend on the particular inheritance structure
              adopted for the algebraic classes. It should also
              be automatic.
*T oal_grp_wf2            4 4 ∀s:LOSet. ∀g:OGrp.  oal_grp(s;g) ∈ OGrp
*T oal_merge_preserves_le 1 4
              ∀s:LOSet. ∀g:OGrp. ∀ps,qs,rs:|oal(s;g)|.
                qs ≤{s,g} rs ⇒ ps ++ qs ≤{s,g} ps ++ rs
*C oal_hgp_com ====================
              ORDERED 'HALF' GROUP
              ====================
              A 'half group' is the ordered cancellation monoid
              of the non-negative elements of an ordered group.
*C set_car_inc_tcom
              This inclusion lemma is needed in proof of oal_hgp_wf.
              Need better way of finding appropriate inclusion lemmas.
              Ideally, this lemma's proof should be completely automatic.
              Problem is that type information makes clauses look different,
              even though they eventually normalize to the same thing.
*T set_car_inc            3 4 ∀s:LOSet. ∀g:OGrp.  |oal(s;g↓hgrp)| ⊆ |oal(s;g)|
*D oal_hgp_df                  oal_hgp(<s:s:*>;<g:g:*>)== oal_hgp{}(<s>; <g>)
*A oal_hgp    oal_hgp(s;g) ==  <|oal(s;g↓hgrp)|, =b, λx,y.x ≤≤b y, λx,y.x ++ y, 00, λx.x>
*T oal_hgp_wf             0 3 ∀s:LOSet. ∀g:OGrp.  oal_hgp(s;g) ∈ GrpSig
*M oal_hgp_ml  let oal_hgpC =
                FirstC
                (map (\t1,t2.MacroC `oal_hgrpC` IdC t1 (ForceReduceC `5`) t2)
```

```
                  [⌈|oal(s;g↓hgrp)|⌉,⌈|oal_hgp(s;g)|⌉
                  ;⌈=b⌉,⌈=b⌉
                  ;⌈x ++ y⌉,⌈x * y⌉
                  ;⌈00⌉,⌈e⌉
                  ])
               ;;
               let oal_add_hgrp_of_ocgrpC,oal_rem_hgrp_of_ocgrpC =
                 let C = RepeatC (EvalC
                   ‘‘oalist eq_list eq_pair
                     oal_merge oal_nil infix_ap
                  ‘‘) in
                 (FirstC # FirstC )
                 (unzip
                   (map (\t1,t2.
                           DoubleMacroC ‘add_oal_hgp‘ C t1 C t2)
                 [⌈=b⌉,⌈=b⌉
                 ;⌈ps ++ qs⌉,⌈ps ++ qs⌉
                 ;⌈00⌉,⌈00⌉
                 ]))
               ;;
*M oal_hgp_ml2 let oal_hgp_to_monC,oal_mon_to_hgpC =
                 let C1 = AbRedexC in
                 let C2 = AbRedexC in
                 (FirstC # FirstC )
                 (unzip
                   (map (\t1,t2. DoubleMacroC ‘oal_hgp_to_monC‘ C1 t1 C2 t2)
                 [⌈|oal_hgp(s;g)|⌉,⌈|oal_mon(s;g↓hgrp)|⌉
                 ;⌈=b⌉,⌈=b⌉
                 ;⌈*⌉,⌈*⌉
                 ;⌈e⌉,⌈e⌉
                 ]))
               ;;
*C oalist_hgrp_eqs_com
               This lemma proves exactly that property
               that should be true of subtypes, but that
               isn’t with the current definition of
               the subtype ‘predicate’ (‘’ because it
               isn’t a fully fledged predicate well-formed
               for any pair of types.
               It also demonstrates proof patterns that could
               be pulled out into ‘template proofs’.
*T oalist_hgrp_eqs        4 5 ∀s:LOSet. ∀g:OGrp. ∀a1,a2:|oal(s;g↓hgrp)|.  a1 = a2 ⇒ a1 = a2
*T oal_hgp_wf2            3 5 ∀s:LOSet. ∀g:OGrp.  oal_hgp(s;g) ∈ OCMon
*C oal_omcp_com
               ==========================
               ASSEMBLY OF MONOID COPOWER
               ==========================
*T oal_inj_mon_hom        3 3
               ∀a:LOSet. ∀b:AbMon. ∀k:|a|.  IsMonHom{b,oal_mon(a;b)}(λv.inj(k,v))
*C oal_umap_char_com
               Desparately need here to throw together
               automatic tactics for solving rewrite rule
               antecedents
*T oal_umap_char          5 7
               ∀s:LOSet. ∀g,h:AbMon. ∀f:|s| → MonHom(g,h).
                 (λps:|oal(s;g)|. msFor{h} k ∈ dom(ps)
```

```
                                              f k ps[k]) = !v:|oal(s;g)| → |h|
                                                             IsMonHom{oal_mon(s;g),h}(v)
                                                             ∧ (∀j:|s|. f j = v o (λw.inj(j,w)))
*D oal_umap_df umap{<s:s:*>,<g:g:*>}(<h:h:*>,<f:f:*>)== oal_umap{}(<s>; <g>; <h>; <f>)
               umap(<h:h:*>,<f:f:*>)== oal_umap{}(<s>; <g>; <h>; <f>)
*A oal_umap                   umap(h,f) ==  λps:|oal(s;g)|. msFor{h} k ∈ dom(ps). f k ps[k]
*T oal_umap_wf           0 3
               ∀s:LOSet. ∀g,h:AbMon. ∀f:|s| → |g| → |h|.  umap(h,f) ∈ |oal(s;g)| → |h|
*T oal_umap_char_a       0 0
               ∀s:LOSet. ∀g,h:AbMon. ∀f:|s| → MonHom(g,h).
                 umap(h,f) = !v:|oal(s;g)| → |h|
                               IsMonHom{oal_mon(s;g),h}(v) ∧ (∀j:|s|. f j = v o (λw.inj(j,w)))
*D oal_mcp_df                 oal_mcp(<s:s:*>;<g:g:*>)== oal_mcp{}(<s>; <g>)
*A oal_mcp     oal_mcp(s;g) ==
                 <oal_mon(s;g), λk,v.inj(k,v), λh,f,ps.msFor{h} k ∈ dom(ps). f k ps[k]>
*T oal_mcp_wf            2 5 ∀s:LOSet. ∀g:AbMon.  oal_mcp(s;g) ∈ MCopower(s;g)
*D oal_omcp_df                oal_omcp{<s:s:*>,<g:g:*>}== oal_omcp{}(<s>; <g>)
*A oal_omcp                   oal_omcp{s,g} ==  <oal_hgp(s;g), λk,v.inj(k,v), λh,f.umap(h,f)>
*T oal_omcp_wf           3 5 ∀s:LOSet. ∀g:OGrp.  oal_omcp{s,g} ∈ MCopower(s;g↓hgrp)
*C polynom_2_end              ************************************
Thm stats: <log2 (# pscript lines)> <log2 (expansion time in sec)>
```