TPPmark 2017: 最長共通部分列(LCS)

山本 光晴 (千葉大学)

5種類の解答を作成

- 自然な再帰的定義　tppmark2017-Function.v
  - 列n(≥1)個に拡張
- 再帰による部分列全生成　tppmark2017-finType.v
  - 列n(≥1)個に拡張
- maskによる部分列網羅　tppmark2017-mask.v
  - 列n(≥1)個に拡張
- 動的計画法(DP)　tppmark2017-DP.v
  - 列n(≥1)個に拡張 ただし拡張版の証明は未完成
- 依存型版動的計画法　tppmark2017-DTDP.v

自然な再帰的定義(列2個)

```
Definition argmax (X : Type) f (x y : X) := if f x < f y then y else x.

Lemma argmax_maxn (X : Type) f (x y : X) : f (argmax f x y) = maxn (f x) (f y).
Proof.  by rewrite /argmax fun_if.  Qed.

Variable T : eqType.

Function LCS p {measure (fun p => size p.1 + size p.2) p} : seq T :=
  if p is (x :: s, y :: t) then if x == y then x :: LCS (s, t)
                                           else argmax size (LCS (s, p.2)) (LCS (p.1, t))
                             else [::].
  - by move=> _ _ _ x s _ y t _ _ _; apply: ltP; rewrite addnS.
  - by move=> _ _ _ x s _ y t _ _ _; apply: ltP; rewrite addnS.
  - by move=> _ _ _ x s _ y t _ _ _; apply: ltP.
Defined.
```

- 構造的な再帰ではないため、FixpointではなくFunctionを使用
    - 減少する引数は1つしか指定できないので、引数を対に

部分列関係 subseq
(in Mathematical Components)

```
Fixpoint subseq s1 s2 :=
  if s2 is y :: s2' then
    if s1 is x :: s1' then subseq (if x == y then s1' else s1) s2' else true
  else s1 == [::].
Lemma sub0seq s : subseq [::] s.
Lemma subseq0 s : subseq s [::] = (s == [::]).
Lemma subseq_trans : transitive subseq.
Lemma subseq_cons s x : subseq s (x :: s).

Definition bitseq := seq bool.
Fixpoint mask (m : bitseq) s {struct m} :=
  match m, s with
  | b :: m', x :: s' => if b then x :: mask m' s' else mask m' s'
  | _, _ => [::]
  end.
Lemma subseqP s1 s2 : reflect (exists2 m, size m = size s2 & s1 = mask m s2) (subseq s1 s2).
Lemma mask_subseq m s : subseq (mask m s) s.
```

例:

```
mask [:: true; false; true; false] [:: 4; 1; 2; 6]
 = [:: 4; 2]
```

```
Lemma lcs_subseq p : subseq (LCS p) p.1 ∧ subseq (LCS p) p.2.
Proof.
 (* functional induction (LCS p) *)
 elim/LCS_ind: p / _; last by move⇒ ? ?; rewrite !sub0seq.
 - by move⇒ _ x s _ t _ /eqP ← /=; rewrite eqxx.
 - move⇒ [_ _] x s y t [→ →] [] // _ _; rewrite /argmax.
   case: ifP ⇒ _ [H1 H2] [H3 H4]; split ⇒ // {H2 H3}.
   + by rewrite (subseq_trans H4) // subseq_cons.
   + by rewrite (subseq_trans H1) // subseq_cons.
Qed.

Lemma lcs_longest p u : subseq u p.1 → subseq u p.2 → size u ≤ size (LCS p).
Proof.
 rewrite [subseq]lock; elim/LCS_ind: p / _ u ⇒ /=.
 - move⇒ _ x s _ t _ /eqP ←; rewrite -lock ⇒ IH; case⇒ // z u Hs Ht.
   by move: (IH _ Hs Ht); case: ifP ⇒ _ // /leqW.
 - move⇒ [_ _] x s y t [→ →] [] // Hxy _ /= IHs IHt.
   rewrite -lock in IHs IHt *; case⇒ // z u /=; rewrite argmax_maxn leq_max.
   case: ifP ⇒ [/eqP → | Hzx] Hs.
   + by rewrite Hxy ⇒ Ht; rewrite (IHt (x :: u)) ?orbT //= eqxx.
   + by move⇒ Ht; rewrite (IHs (z :: u)).
 - move⇒ [s t] [_ _] [← ←]; rewrite -lock.
   case: s ⇒ [|x s] /=; first by move⇒ _ _ /eqP →.
   by case: t ⇒ [|y t] //= ⇒ _ _ _ /eqP →.
Qed.
```

自然な再帰的定義(列n(≥1)個)

```
Fixpoint argmaxl (X : Type) f (x : X) ys ≔
  if ys is y :: ys′ then argmaxl f (if f x < f y then y else x) ys else x.

Variable T : eqType.

Definition LCSL_spec s ts (lcsl : seq T) ≔ [∧ subseq lcsl s, all (subseq lcsl) ts &
                                           ∀ u, subseq u s → all (subseq u) ts → size u ≤ size lcsl].

Program Fixpoint LCSL_strong s ts {measure (size s + (∑_(t ← ts) size t))}
                                  : {lcsl : seq T | LCSL_spec s ts lcsl} ≔
  if s is x :: s′ then
    let hds ≔ map ohead ts in
    match None ∈ hds with
    | true ⇒ [::]
    | false ⇒ if all (pred1 (Some x)) hds then x :: LCSL_strong s′ (map behead ts)
              else argmaxl size (LCSL_strong s′ ts)
                          [tuple LCSL_strong s (set_nth [::] ts i (behead (nth [::] ts i))) | i < size ts]
    end
  else [::].
Next Obligation. … (* 5 obligations,  92-line proof script. *)
```

1つ目の列(: seq T)

2つ目以降の列(: seq (seq T))

tsのi番目の要素(それ自身リスト)を、それの先頭を除いたものに置き換えたもの

再帰による部分列全生成
(続きはmask版(後述)とほぼ同様)

```
Variable T : choiceType.
Definition all_subseqs ≔ foldr (fun (x : T) ss ⇒ ss ++ map (cons x) ss) [:: [::]].
Lemma mem_all_subseqs s t : (s ∈ all_subseqs t) = subseq s t.
Proof.
  elim: t ⇒ [| y t IHt] in s *; first by rewrite inE.
  case: s ⇒ [| x s] ⇒ /=; rewrite mem_cat; first by rewrite IHt sub0seq.
  case: ifPn ⇒ [/eqP ← | /eqP Hxy]; rewrite IHt.
  - rewrite mem_map ?IHt ?orb_idl //; last by move⇒ ? ? [].
    by apply: subseq_trans; rewrite subseq_cons.
  - rewrite orb_idr // ⇒ /nthP -/(_ [::]) [n].
    rewrite size_map ⇒ Hsize Hnth. case: Hxy. move: Hnth.
    by rewrite (nth_map [::]) // ⇒ -[→].
Qed.

Lemma nil_in_all_subseqs s : [::] ∈ all_subseqs s.
Proof.  by rewrite mem_all_subseqs sub0seq.  Qed.
Definition LCS s t : seq T ≔ val [arg max_(r > (Sub [::] (nil_in_all_subseqs s)
                                      : seq_sub (all_subseqs s)) | subseq (val r) t)
                               size (val r)].
```

[arg max_(i > $i0$ | P) M] ⋯ Pを満たしMを最大化するiの値 ($i0$はPを満たすものが存在することのwitness)

seq_sub s ⋯ リストsの要素全体からなる型 finTypeインターフェースを持つ

maskによる部分列網羅(列2個)

[tuple **of** nseq n b] = [tuple b; b; … ; b] : n.-tuple bool

n.-tuple bool から seq bool へのCoercionが自動的に入る

```
Variable T : eqType.

Definition LCS s t : seq T ≔ mask [arg max_(m > [tuple of nseq (size s) false] |
                                       subseq (mask m s) t)
                                 size (mask m s)] s.

Lemma lcs_subseq s t : subseq (LCS s t) s ∧ subseq (LCS s t) t.
Proof.
 rewrite /LCS; split; first by apply: mask_subseq.
 by case: arg_maxP ⇒ //; rewrite mask_false sub0seq.
Qed.

Lemma lcs_longest s t u : subseq u s → subseq u t → size u ≤ size (LCS s t).
Proof.
 rewrite /LCS ⇒ /subseqP [m [← Hus]] Hut.
 case: arg_maxP ⇒ /=; first by rewrite mask_false sub0seq.
 by move⇒ ? _ /(_ (in_tuple m)); rewrite -Hus ⇒ →.
Qed.
```

maskによる部分列網羅
(列n(≥1)個)

```
Variable T : eqType.

Definition LCSL s ts : seq T ≔ mask [arg max_(m > [tuple of nseq (size s) false] |
                                              all (subseq (mask m s)) ts)
                                         size (mask m s)] s.

Lemma lcsl_subseq s ts : subseq (LCSL s ts) s ∧ all (subseq (LCSL s ts)) ts.
Proof.
  rewrite /LCSL; split; first by apply: mask_subseq.
  by case: arg_maxP ⇒ //; apply/allP ⇒ *; rewrite mask_false sub0seq.
Qed.

Lemma lcsl_longest s ts u : subseq u s → all (subseq u) ts → size u ≤ size (LCSL s ts).
Proof.
  rewrite /LCSL ⇒ /subseqP [m [← Hus]] Hut.
  case: arg_maxP ⇒ /=; first by apply/allP ⇒ *; rewrite mask_false sub0seq.
  by move⇒ ? _ /(_ (in_tuple m)); rewrite -Hus ⇒ →.
Qed.
```

DP版(直接定義, 列2個)

```
Fixpoint LCS2 x us t : seq (seq T) ≔ if t is y :: t′ then let p ≔ LCS2 x (behead us) t′ in
                                         (if x == y then x :: head [::] (behead us)
                                          else argmax size (head [::] us) (head [::] p)) :: p
                                     else [::].
Fixpoint LCS1 s t : seq (seq T) ≔ if s is x :: s′ then LCS2 x (LCS1 s′ t) t else [::].
Definition LCS s t ≔ head [::] (LCS1 s t).

Lemma lcs_nil_l s : LCS [::] s = [::].
Proof. by []. Qed.
Lemma lcs_nil_r s : LCS s [::] = [::].
Proof. by rewrite /LCS; case: s ⇒ [| x s] //=; case: LCS1. Qed.
Lemma lcs_cons_aux x s y t : LCS1 (x :: s) (y :: t) = (if x == y then x :: (LCS s t)
                                          else argmax size (LCS s (y :: t)) (LCS (x :: s) t))
                                         :: LCS1 (x :: s) t.
Proof. by rewrite /LCS; elim: s ⇒ [| x′ s /= /(_ x′) [→ →]] // in x *. Qed.
Lemma lcs_cons x s y t : LCS (x :: s) (y :: t) = if x == y then x :: LCS s t
                                   else argmax size (LCS s (y :: t)) (LCS (x :: s) t).
Proof. by rewrite /LCS lcs_cons_aux. Qed.
```

```
Lemma lcs_subseq s t : subseq (LCS s t) s ∧ subseq (LCS s t) t.
Proof.
  elim: s ⇒ [| x s IHs] in t *; first by rewrite lcs_nil_l !sub0seq.
  elim: t ⇒ [| y t [IHts IHtt]]; first by rewrite lcs_nil_r sub0seq.
  rewrite lcs_cons; case: ifP ⇒ [/eqP → | Hxy]; first by rewrite /= eqxx.
  case/IHs: (y :: t) ⇒ IHss IHst; rewrite /argmax; split; case: ifP ⇒ _ //.
  - by rewrite (subseq_trans IHss) // subseq_cons.
  - by rewrite (subseq_trans IHtt) // subseq_cons.
Qed.

Lemma lcs_longest s t u : subseq u s → subseq u t → size u ≤ size (LCS s t).
Proof.
  elim: s ⇒ [| x s IHs] in u t *; first by move⇒ /eqP →.
  elim: t ⇒ [| y t IHt] in u *; first by move⇒ _ /eqP →.
  case: u ⇒ [| z u] //; rewrite lcs_cons; case: ifP ⇒ [/eqP ←{y} | Hxy] /=.
  - by move⇒ Hs Ht; move: (IHs _ _ Hs Ht); case: ifP ⇒ _ // /leqW.
  - rewrite argmax_maxn leq_max; case: ifP ⇒ [/eqP → | _] Hs.
    + by rewrite Hxy ⇒ Ht; rewrite (IHt (x :: u)) ?orbT //= eqxx.
    + by move⇒ Ht; rewrite (IHs (z :: u)).
Qed.
```

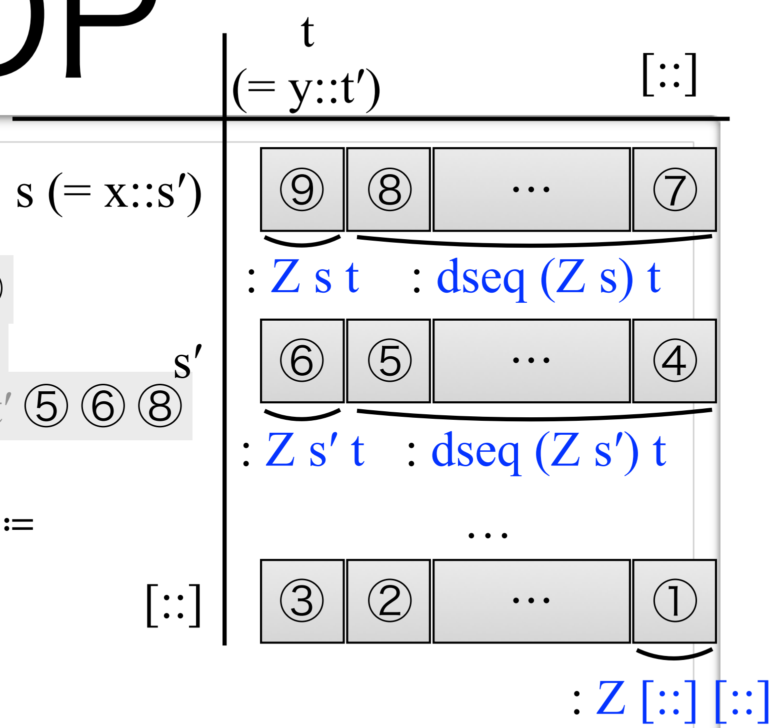
DP部分のくくり出し

```
Variables X Y Z : Type.
Variable z00 : Z.
Variable zc0 : X → seq X → Z → Z.
Variable z0c : Y → seq Y → Z → Z.
Variable zcc : X → seq X → Y → seq Y → Z → Z → Z → Z.

Fixpoint dp0t t ≔ if t is y :: t′ then let: (z, zs) ≔ dp0t t′ in (z0c y t′ z, z :: zs) else (z00, [::]).
Fixpoint dpt x s t zs ≔ match t, zs with | y :: t′, (z, z′ :: zs′) ⇒ let: (zy, zt′) ≔ dpt x s t′ (z′, zs′) in
                                                                      (zcc x s y t′ z′ z zy, zy :: zt′)
                                          | _, (z, _) ⇒ (zc0 x s z, [::]) end.
Fixpoint dps s t ≔ if s is x :: s′ then dpt x s′ t (dps s′ t) else dp0t t.
Definition dp s t ≔ (dps s t).1.

Lemma dp00 : dp [::] [::] = z00.
Lemma dp0c y t : dp [::] (y :: t) = z0c y t (dp [::] t).
Lemma dpc0 x s : dp (x :: s) [::] = zc0 x s (dp s [::]).
Lemma dpscc x y s t : dps (x :: s) (y :: t) = (zcc x s y t (dp s t) (dp s (y :: t)) (dp (x :: s) t),
                                               dp (x :: s) t :: (dps (x :: s) t).2).
Lemma dpcc x y s t : dp (x :: s) (y :: t) = zcc x s y t (dp s t) (dp s (y :: t)) (dp (x :: s) t).
```

依存型版DP

```
Variables (X Y : Type) (Z : seq X → seq Y → Type).
Variable z00 : Z [::] [::].
Variable zc0 : ∀ x s, Z s [::] → Z (x :: s) [::].
Variable z0c : ∀ y t, Z [::] t → Z [::] (y :: t).
Variable zcc : ∀ x s y t, Z s t → Z s (y :: t) →
                   Z (x :: s) t → Z (x :: s) (y :: t).
```

① = z00
⑦ = zc0 x s′ ④
③ = z0c y t′ ②
⑨ = zcc x s′ y t′ ⑤ ⑥ ⑧




```
Inductive dseq (X : Type) (T : seq X → Type) : seq X → Type ≔
| DNil : dseq T [::]
| DCons (x : X) (s : seq X) of T s & dseq T s : dseq T (x :: s).
Fixpoint dp0t t : Z [::] t * dseq (Z [::]) t ≔
  if t is y :: t′ then let: (z, zs) ≔ dp0t t′ in (z0c y z, DCons z zs) else (z00, DNil).
Fixpoint dpt x s t : Z s t * dseq (Z s) t → Z (x :: s) t * dseq (Z (x :: s)) t ≔
  if t is y :: t′ then fun zs ⇒ (if zs.2 is DCons y0 t′0 z′ zs′ in dseq _ t0 return t0 = y :: t′ → _ then
                          fun E ⇒ let: Et′ ≔ congr1 behead E : t′0 = t′ in
                                  let: (zy, zt′) ≔ dpt x (ecast _ _ Et′ (z′, zs′)) in
                                  (zcc (ecast _ _ Et′ z′) zs.1 zy, DCons zy zt′)
                     else fun E ⇒ match List.nil_cons E with end) (erefl (y :: t′))
           else fun zs ⇒ (zc0 x zs.1, DNil).
Fixpoint dps s t : Z s t * dseq (Z s) t ≔ if s is x :: s′ then dpt x (dps s′ t) else dp0t t.
Definition dp s t : Z s t ≔ (dps s t).1.
```

依存型版DPによるLCS

```
Definition LCS_strong s t : {lcs | LCS_spec s t lcs}.
  apply: dp (exist _ [::] _) (fun _ _ _ ⇒ exist _ [::] _) (fun _ _ _ ⇒ exist _ [::] _)
                 (fun x _ y _ st syt xst ⇒ exist _ (if x == y then x :: sval st
                                                     else argmax size (sval syt) (sval xst)) _) s t.
  - by split⇒ //= ? /eqP →.
  - by move⇒ ? ? ?; split⇒ // ? ? /eqP →.
  - by move⇒ ? ? ?; split⇒ // ? /eqP →.
  - move⇒ x s y t [st [Hst1 Hst2 Hst3]] [syt [Hsyt1 Hsyt2 Hsyt3]] [xst [Hxst1 Hxst2 Hxst3]].
    rewrite ![sval _]/=; split.
    + case: ifP ⇒ [_ | Hxy]; first by rewrite /= eqxx.
      by rewrite /argmax; case: ifP ⇒ _ //; rewrite (subseq_trans Hsyt1) // subseq_cons.
    + case: ifP ⇒ [/eqP → | Hxy]; first by rewrite /= eqxx.
      by rewrite /argmax; case: ifP ⇒ _ //; rewrite (subseq_trans Hxst2) // subseq_cons.
    + case⇒ [| z u] //; case: ifP ⇒ [/eqP ← | Hxy] /=.
      * by move⇒ Hs Ht; move: (Hst3 _ Hs Ht); case: ifP ⇒ _ // /leqW.
      * { rewrite argmax_maxn leq_max; case: ifP ⇒ [/eqP → | _] Hs.
          - by rewrite Hxy ⇒ Ht; rewrite (Hxst3 (x :: u)) ?orbT //= eqxx.
          - by move ⇒ Ht; rewrite (Hsyt3 (z :: u)). }
Defined.
```

非依存型版であった補題(dp (x :: s) (y :: t) = zcc (dp s t) (dp s (y :: t)) (dp (x :: s) t)など)は不要

DP版(直接定義, 列n(≥1)個)

```
Fixpoint LCSLs s ys diag neighbors : seq (seq T) ≔
  if s is x :: s′ then let prev ≔ LCSLs s′ ys (omap behead diag) (map behead neighbors) in
                       (if all (pred1 x) ys then x :: head [::] (oapp behead prev diag)
                        else argmaxl size (head [::] prev) (map (head [::]) neighbors)) :: prev
  else [::].
Fixpoint LCSLts s ts ys : option (iter (size ts).+1 seq (seq T)) → (* diag *)
                                 seq (iter (size ts).+1 seq (seq T)) → (* neighbors *)
                                 iter (size ts).+1 seq (seq T) ≔
  if ts is t :: ts′ then fun diag neighbors ⇒
                          let fix LCSLt t neighbors ≔
                            if t is y :: t′ then
                              let prev ≔ LCSLt t′ (map behead neighbors) in
                              (@LCSLts s ts′ (y :: ys) (Some (head [::] (oapp behead prev diag)))
                                         (head [::] prev :: map (head [::]) neighbors)) :: prev
                            else [::] in LCSLt t neighbors
                  else fun diag neighbors ⇒ LCSLs s ys diag neighbors.
Fixpoint nhead (X : Type) (d : X) n : iter n seq X → X ≔
  if n is n′.+1 then fun s ⇒ if s is x :: s′ then nhead d x else d else id.
Definition LCSL s ts ≔ nhead [::] _ (@LCSLts s ts [::] None [::]).
```

DP版(直接定義, 列n(≥1)個)

```
Fixpoint LCSLs s ys diag neighbors : seq (seq T) ≔
  if s is x :: s′ then let prev ≔ LCSLs s′ ys (omap behead diag) (map behead neighbors) in
                              (if all (pred1 x) ys then x :: head [::] (oapp behead prev diag)
                               else argmaxl size (head [::] prev) (map (head [::]) neighbors)) :: prev
  else [::].

Lemma LCSLts_nil s ys diag neighbors:
  LCSLts s [::] ys diag neighbors = LCSLs s ys diag neighbors.
Lemma LCSLts_cons_nil s ts ys diag neighbors:
  LCSLts s ([::] :: ts) ys diag neighbors = [::].
Lemma LCSLts_cons_cons s y t ts ys diag neighbors:
  LCSLts s ((y :: t) :: ts) ys diag neighbors =
  let prev ≔ LCSLts s (t :: ts) ys diag (map behead neighbors) in
  (LCSLts s ts (y :: ys) (Some (head [::] (oapp behead prev diag)))
            (head [::] prev :: map (head [::]) neighbors)) :: prev.
Definition LCSLtsE ≔ (LCSLts_nil, LCSLts_cons_nil, LCSLts_cons_cons).

Fixpoint nhead (X : Type) (d : X) n : iter n seq X → X ≔
  if n is n′.+1 then fun s ⇒ if s is x :: s′ then nhead d x else d else id.
Definition LCSL s ts ≔ nhead [::] _ (@LCSLts s ts [::] None [::]).
```

発表後の追記

- 依存型版DP(13ページ)は、dseqをInductiveではなくFixpointで定義するようにするとdptの定義がより簡単になります。(thanks to 坂口さん)

- 最長性の証明スクリプトが他の方のものに比べて私のが短いのは、使用したsubseqの定義(4ページ)の仕方によるところが大きいです。(thanks to 中野先生)

- 最新版はhttps://bitbucket.org/mituharu/tppmark17に置いておきます。