# tphols-2011

By xingyuan

January 27, 2011

## Contents

**theory** *Myhill*
  **imports** *Myhill-1*
**begin**

# 1 Direction: *regular language* ⇒*finite partition*

## 1.1 The scheme for this direction

The following convenient notation $x \approx Lang\ y$ means: string $x$ and $y$ are equivalent with respect to language *Lang*.

**definition**
  *str-eq* (- ≈- -)
**where**
  $x \approx Lang\ y \equiv (x,\ y) \in (\approx Lang)$

The very basic scheme to show the finiteness of the partion generated by a language *Lang* is by attaching a tag to every string. The set of tags are carfully choosen to be finite so that the range of tagging function is finite. If it can be proved that strings with the same tag are equivlent with respect *Lang*, then the partition given rise by *Lang* must be finite. The detailed argjument for this is formalized by the following lemma *tag-finite-imageD*. The basic idea is using lemma *finite-imageD* from standard library:

$$[\![finite\ (f\ `\ A);\ inj\text{-}on\ f\ A]\!] \Longrightarrow finite\ A$$

which says: if the image of injective function $f$ over set $A$ is finite, then $A$ must be finte.

**lemma** *finite-range-image*: *finite* (*range f*) ⟹ *finite* (*f* ‘ *A*)
  **by** (*rule-tac B* = {*y*. ∃ *x*. *y* = *f x*} **in** *finite-subset*, *auto simp*:*image-def*)

**lemma** *tag-finite-imageD*:
  **fixes** *tag*
  **assumes** *rng-fnt*: *finite* (*range tag*)

```
  — Suppose the rang of tagging fucntion tag is finite.
  and same-tag-eqvt: ⋀ m n. tag m = tag (n::string) ⟹ m ≈lang n
  — And strings with same tag are equivalent
  shows finite (UNIV // (≈lang))
  — Then the partition generated by (≈lang) is finite.
proof −
  — The particular f and A used in finite-imageD are:
  let ?f = op ‘ tag and ?A = (UNIV // ≈lang)
  show ?thesis
  proof (rule-tac f = ?f and A = ?A in finite-imageD)
    — The finiteness of f-image is a simple consequence of assumption rng-fnt:
    show finite (?f ‘ ?A)
    proof −
      have ∀ X. ?f X ∈ (Pow (range tag)) by (auto simp:image-def Pow-def)
      moreover from rng-fnt have finite (Pow (range tag)) by simp
      ultimately have finite (range ?f)
        by (auto simp only:image-def intro:finite-subset)
      from finite-range-image [OF this] show ?thesis .
    qed
  next
    — The injectivity of f is the consequence of assumption same-tag-eqvt:
    show inj-on ?f ?A
    proof−
      { fix X Y
        assume X-in: X ∈ ?A
          and  Y-in: Y ∈ ?A
          and  tag-eq: ?f X = ?f Y
        have X = Y
        proof −
          from X-in Y-in tag-eq
         obtain x y where x-in: x ∈ X and y-in: y ∈ Y and eq-tg: tag x = tag y
            unfolding quotient-def Image-def str-eq-rel-def str-eq-def image-def
            apply simp by blast
          from same-tag-eqvt [OF eq-tg] have x ≈lang y .
          with X-in Y-in x-in y-in
          show ?thesis by (auto simp:quotient-def str-eq-rel-def str-eq-def)
        qed
      } thus ?thesis unfolding inj-on-def by auto
    qed
  qed
qed
```

## 1.2 Lemmas for basic cases

The the final result of this direction is in *easier-direction*, which is an induction on the structure of regular expressions. There is one case for each regular expression operator. For basic operators such as *NULL*, *EMPTY*, *CHAR c*, the finiteness of their language partition can be established directly with no need of taggiing. This section contains several technical lemma for these base cases.

The inductive cases involve operators *ALT*, *SEQ* and *STAR*. Tagging functions need to be defined individually for each of them. There will be one dedicated section for each of these cases, and each section goes virtually the same way: gives definition of the tagging function and prove that strings with the same tag are equivalent.

**lemma** *quot-empty-subset*:
  $UNIV\ //\ (\approx\{[]\}) \subseteq \{\{[]\},\ UNIV - \{[]\}\}$
**proof**
  **fix** $x$
  **assume** $x \in UNIV\ //\ \approx\{[]\}$
  **then obtain** $y$ **where** $h$: $x = \{z.\ (y, z) \in \approx\{[]\}\}$
    **unfolding** *quotient-def Image-def* **by** *blast*
  **show** $x \in \{\{[]\},\ UNIV - \{[]\}\}$
  **proof** (*cases* $y = []$)
    **case** *True* **with** $h$
    **have** $x = \{[]\}$ **by** (*auto simp*:*str-eq-rel-def*)
    **thus** *?thesis* **by** *simp*
  **next**
    **case** *False* **with** $h$
    **have** $x = UNIV - \{[]\}$ **by** (*auto simp*:*str-eq-rel-def*)
    **thus** *?thesis* **by** *simp*
  **qed**
**qed**

**lemma** *quot-char-subset*:
  $UNIV\ //\ (\approx\{[c]\}) \subseteq \{\{[]\},\{[c]\},\ UNIV - \{[],\ [c]\}\}$
**proof**
  **fix** $x$
  **assume** $x \in UNIV\ //\ \approx\{[c]\}$
  **then obtain** $y$ **where** $h$: $x = \{z.\ (y, z) \in \approx\{[c]\}\}$
    **unfolding** *quotient-def Image-def* **by** *blast*
  **show** $x \in \{\{[]\},\{[c]\},\ UNIV - \{[],\ [c]\}\}$
  **proof** −
    **{ assume** $y = []$ **hence** $x = \{[]\}$ **using** $h$
        **by** (*auto simp*:*str-eq-rel-def*)
    **} moreover {**
      **assume** $y = [c]$ **hence** $x = \{[c]\}$ **using** $h$
        **by** (*auto dest*!:*spec*[**where** $x = []$] *simp*:*str-eq-rel-def*)
    **} moreover {**
      **assume** $y \neq []$ **and** $y \neq [c]$
      **hence** $\forall\ z.\ (y\ @\ z) \neq [c]$ **by** (*case-tac* $y$, *auto*)
      **moreover have** $\bigwedge p.\ (p \neq [] \wedge p \neq [c]) = (\forall\ q.\ p\ @\ q \neq [c])$
        **by** (*case-tac* $p$, *auto*)
      **ultimately have** $x = UNIV - \{[],[c]\}$ **using** $h$
        **by** (*auto simp add*:*str-eq-rel-def*)
    **} ultimately show** *?thesis* **by** *blast*
  **qed**

**qed**

## 1.3 The case for *SEQ*

**definition**
  *tag-str-SEQ* $L_1$ $L_2$ $x$ ≡
    $((\approx L_1)$ ‘‘ $\{x\}$, $\{(\approx L_2)$ ‘‘ $\{x - xa\}|\ xa.\ xa \le x \wedge xa \in L_1\})$

**lemma** *tag-str-seq-range-finite*:
  ⟦*finite* (*UNIV* // ≈$L_1$); *finite* (*UNIV* // ≈$L_2$)⟧
    ⟹ *finite* (*range* (*tag-str-SEQ* $L_1$ $L_2$))
**apply** (*rule-tac B* = (*UNIV* // ≈$L_1$) × (*Pow* (*UNIV* // ≈$L_2$)) **in** *finite-subset*)
**by** (*auto simp*:*tag-str-SEQ-def Image-def quotient-def split*:*if-splits*)

**lemma** *append-seq-elim*:
  **assumes** $x$ @ $y \in L_1$ ;; $L_2$
  **shows** (∃ $xa \le x$. $xa \in L_1 \wedge (x - xa)$ @ $y \in L_2) \vee$
      (∃ $ya \le y$. ($x$ @ $ya$) $\in L_1 \wedge (y - ya) \in L_2$)
**proof**−
  **from** *assms* **obtain** $s_1$ $s_2$
    **where** $x$ @ $y$ = $s_1$ @ $s_2$
    **and** *in-seq*: $s_1 \in L_1 \wedge s_2 \in L_2$
    **by** (*auto simp*:*Seq-def*)
  **hence** ($x \le s_1 \wedge (s_1 - x)$ @ $s_2 = y) \vee (s_1 \le x \wedge (x - s_1)$ @ $y = s_2$)
    **using** *app-eq-dest* **by** *auto*
  **moreover have** ⟦$x \le s_1$; $(s_1 - x)$ @ $s_2 = y$⟧ ⟹
      ∃ $ya \le y$. ($x$ @ $ya$) $\in L_1 \wedge (y - ya) \in L_2$
    **using** *in-seq* **by** (*rule-tac* $x = s_1 - x$ **in** *exI*, *auto elim*:*prefixE*)
  **moreover have** ⟦$s_1 \le x$; $(x - s_1)$ @ $y = s_2$⟧ ⟹
      ∃ $xa \le x$. $xa \in L_1 \wedge (x - xa)$ @ $y \in L_2$
    **using** *in-seq* **by** (*rule-tac* $x = s_1$ **in** *exI*, *auto*)
  **ultimately show** *?thesis* **by** *blast*
**qed**

**lemma** *tag-str-SEQ-injI*:
  *tag-str-SEQ* $L_1$ $L_2$ $m$ = *tag-str-SEQ* $L_1$ $L_2$ $n$ ⟹ $m \approx(L_1$ ;; $L_2)$ $n$
**proof**−
  **{ fix** $x$ $y$ $z$
    **assume** *xz-in-seq*: $x$ @ $z \in L_1$ ;; $L_2$
    **and** *tag-xy*: *tag-str-SEQ* $L_1$ $L_2$ $x$ = *tag-str-SEQ* $L_1$ $L_2$ $y$
    **have**$y$ @ $z \in L_1$ ;; $L_2$
    **proof**−
      **have** (∃ $xa \le x$. $xa \in L_1 \wedge (x - xa)$ @ $z \in L_2) \vee$
          (∃ $za \le z$. ($x$ @ $za$) $\in L_1 \wedge (z - za) \in L_2$)
        **using** *xz-in-seq append-seq-elim* **by** *simp*
      **moreover {**
        **fix** $xa$
        **assume** *h1*: $xa \le x$ **and** *h2*: $xa \in L_1$ **and** *h3*: $(x - xa)$ @ $z \in L_2$
        **obtain** $ya$ **where** $ya \le y$ **and** $ya \in L_1$ **and** $(y - ya)$ @ $z \in L_2$

```
    proof -
      have ∃ ya. ya ≤ y ∧ ya ∈ L1 ∧ (x - xa) ≈L2 (y - ya)
      proof -
        have {≈L2 `` {x - xa} |xa. xa ≤ x ∧ xa ∈ L1} =
              {≈L2 `` {y - xa} |xa. xa ≤ y ∧ xa ∈ L1}
                  (is ?Left = ?Right)
          using h1 tag-xy by (auto simp:tag-str-SEQ-def)
        moreover have ≈L2 `` {x - xa} ∈ ?Left using h1 h2 by auto
        ultimately have ≈L2 `` {x - xa} ∈ ?Right by simp
        thus ?thesis by (auto simp:Image-def str-eq-rel-def str-eq-def)
      qed
      with prems show ?thesis by (auto simp:str-eq-rel-def str-eq-def)
    qed
    hence y @ z ∈ L1 ;; L2 by (erule-tac prefixE, auto simp:Seq-def)
  } moreover {
    fix za
    assume h1: za ≤ z and h2: (x @ za) ∈ L1 and h3: z - za ∈ L2
    hence y @ za ∈ L1
    proof-
      have ≈L1 `` {x} = ≈L1 `` {y}
        using h1 tag-xy by (auto simp:tag-str-SEQ-def)
      with h2 show ?thesis
        by (auto simp:Image-def str-eq-rel-def str-eq-def)
    qed
    with h1 h3 have y @ z ∈ L1 ;; L2
      by (drule-tac A = L1 in seq-intro, auto elim:prefixE)
  }
  ultimately show ?thesis by blast
 qed
} thus tag-str-SEQ L1 L2 m = tag-str-SEQ L1 L2 n ⟹ m ≈(L1 ;; L2) n
  by (auto simp add: str-eq-def str-eq-rel-def)
qed
```

```
lemma quot-seq-finiteI:
  ⟦finite (UNIV // ≈L1); finite (UNIV // ≈L2)⟧
  ⟹ finite (UNIV // ≈(L1 ;; L2))
  apply (rule-tac tag = tag-str-SEQ L1 L2 in tag-finite-imageD)
  by (auto intro:tag-str-SEQ-injI elim:tag-str-seq-range-finite)
```

## 1.4 The case for *ALT*

```
definition
  tag-str-ALT L1 L2 (x::string) ≡ ((≈L1) `` {x}, (≈L2) `` {x})

lemma quot-union-finiteI:
  assumes finite1: finite (UNIV // ≈(L1::string set))
  and finite2: finite (UNIV // ≈L2)
  shows finite (UNIV // ≈(L1 ∪ L2))
proof (rule-tac tag = tag-str-ALT L1 L2 in tag-finite-imageD)
```

```
  show ⋀m n. tag-str-ALT L1 L2 m = tag-str-ALT L1 L2 n ⟹ m ≈(L1 ∪ L2) n
    unfolding tag-str-ALT-def str-eq-def Image-def str-eq-rel-def by auto
next
  show finite (range (tag-str-ALT L1 L2)) using finite1 finite2
    apply (rule-tac B = (UNIV // ≈L1) × (UNIV // ≈L2) in finite-subset)
    by (auto simp:tag-str-ALT-def Image-def quotient-def)
qed
```

## 1.5 The case for *STAR*

This turned out to be the trickiest case.

```
definition
  tag-str-STAR L1 x ≡ {(≈L1) `` {x − xa} | xa. xa < x ∧ xa ∈ L1⋆}

lemma finite-set-has-max: ⟦finite A; A ≠ {}⟧ ⟹
          (∃ max ∈ A. ∀ a ∈ A. f a <= (f max :: nat))
proof (induct rule:finite.induct)
  case emptyI thus ?case by simp
next
  case (insertI A a)
  show ?case
  proof (cases A = {})
    case True thus ?thesis by (rule-tac x = a in bexI, auto)
  next
    case False
    with prems obtain max
      where h1: max ∈ A
      and h2: ∀ a∈A. f a ≤ f max by blast
    show ?thesis
    proof (cases f a ≤ f max)
      assume f a ≤ f max
      with h1 h2 show ?thesis by (rule-tac x = max in bexI, auto)
    next
      assume ¬ (f a ≤ f max)
      thus ?thesis using h2 by (rule-tac x = a in bexI, auto)
    qed
  qed
qed

lemma finite-strict-prefix-set: finite {xa. xa < (x::string)}
apply (induct x rule:rev-induct, simp)
apply (subgoal-tac {xa. xa < xs @ [x]} = {xa. xa < xs} ∪ {xs})
by (auto simp:strict-prefix-def)


lemma tag-str-star-range-finite:
  finite (UNIV // ≈L1) ⟹ finite (range (tag-str-STAR L1))
apply (rule-tac B = Pow (UNIV // ≈L1) in finite-subset)
by (auto simp:tag-str-STAR-def Image-def
```

```
                    quotient-def split:if-splits)

lemma tag-str-STAR-injI:
  tag-str-STAR L₁ m = tag-str-STAR L₁ n ⟹ m ≈(L₁⋆) n
proof−
  { fix x y z
    assume xz-in-star: x @ z ∈ L₁⋆
    and tag-xy: tag-str-STAR L₁ x = tag-str-STAR L₁ y
    have y @ z ∈ L₁⋆
    proof(cases x = [])
      case True
      with tag-xy have y = []
        by (auto simp:tag-str-STAR-def strict-prefix-def)
      thus ?thesis using xz-in-star True by simp
    next
      case False
      obtain x-max
        where h1: x-max < x
        and h2: x-max ∈ L₁⋆
        and h3: (x − x-max) @ z ∈ L₁⋆
        and h4:∀ xa < x. xa ∈ L₁⋆ ∧ (x − xa) @ z ∈ L₁⋆
                              ⟶ length xa ≤ length x-max
      proof−
        let ?S = {xa. xa < x ∧ xa ∈ L₁⋆ ∧ (x − xa) @ z ∈ L₁⋆}
        have finite ?S
          by (rule-tac B = {xa. xa < x} in finite-subset,
                      auto simp:finite-strict-prefix-set)
        moreover have ?S ≠ {} using False xz-in-star
          by (simp, rule-tac x = [] in exI, auto simp:strict-prefix-def)
        ultimately have ∃ max ∈ ?S. ∀ a ∈ ?S. length a ≤ length max
          using finite-set-has-max by blast
        with prems show ?thesis by blast
      qed
      obtain ya
        where h5: ya < y and h6: ya ∈ L₁⋆ and h7: (x − x-max) ≈L₁ (y − ya)
      proof−
        from tag-xy have {≈L₁ `` {x − xa} |xa. xa < x ∧ xa ∈ L₁⋆} =
          {≈L₁ `` {y − xa} |xa. xa < y ∧ xa ∈ L₁⋆} (is ?left = ?right)
          by (auto simp:tag-str-STAR-def)
        moreover have ≈L₁ `` {x − x-max} ∈ ?left using h1 h2 by auto
        ultimately have ≈L₁ `` {x − x-max} ∈ ?right by simp
        with prems show ?thesis apply
          (simp add:Image-def str-eq-rel-def str-eq-def) by blast
      qed
      have (y − ya) @ z ∈ L₁⋆
      proof−
        from h3 h1 obtain a b where a-in: a ∈ L₁
          and a-neq: a ≠ [] and b-in: b ∈ L₁⋆
          and ab-max: (x − x-max) @ z = a @ b
```

```
        by (drule-tac star-decom, auto simp:strict-prefix-def elim:prefixE)
      have (x − x-max) ≤ a ∧ (a − (x − x-max)) @ b = z
      proof −
        have ((x − x-max) ≤ a ∧ (a − (x − x-max)) @ b = z) ∨
                 (a < (x − x-max) ∧ ((x − x-max) − a) @ z = b)
          using app-eq-dest[OF ab-max] by (auto simp:strict-prefix-def)
        moreover {
          assume np: a < (x − x-max) and b-eqs: ((x − x-max) − a) @ z = b
          have False
          proof −
            let ?x-max' = x-max @ a
            have ?x-max' < x
              using np h1 by (clarsimp simp:strict-prefix-def diff-prefix)
            moreover have ?x-max' ∈ L₁⋆
              using a-in h2 by (simp add:star-intro3)
            moreover have (x − ?x-max') @ z ∈ L₁⋆
              using b-eqs b-in np h1 by (simp add:diff-diff-appd)
            moreover have ¬ (length ?x-max' ≤ length x-max)
              using a-neq by simp
            ultimately show ?thesis using h4 by blast
          qed
        } ultimately show ?thesis by blast
      qed
      then obtain za where z-decom: z = za @ b
        and x-za: (x − x-max) @ za ∈ L₁
        using a-in by (auto elim:prefixE)
      from x-za h7 have (y − ya) @ za ∈ L₁
        by (auto simp:str-eq-def str-eq-rel-def)
      with z-decom b-in show ?thesis by (auto dest!:step[of (y − ya) @ za])
    qed
    with h5 h6 show ?thesis
      by (drule-tac star-intro1, auto simp:strict-prefix-def elim:prefixE)
  qed
} thus tag-str-STAR L₁ m = tag-str-STAR L₁ n ⟹ m ≈(L₁⋆) n
  by (auto simp add:str-eq-def str-eq-rel-def)
qed

lemma quot-star-finiteI:
  finite (UNIV // ≈L₁) ⟹ finite (UNIV // ≈(L₁⋆))
  apply (rule-tac tag = tag-str-STAR L₁ in tag-finite-imageD)
  by (auto intro:tag-str-STAR-injI elim:tag-str-star-range-finite)
```

## 1.6 The main lemma

```
lemma easier-directioν:
  Lang = L (r::rexp) ⟹ finite (UNIV // (≈Lang))
proof (induct arbitrary:Lang rule:rexp.induct)
  case NULL
  have UNIV // (≈{}) ⊆ {UNIV}
```

**by** (*auto simp*:*quotient-def str-eq-rel-def str-eq-def*)
**with** *prems* **show** *?case* **by** (*auto intro*:*finite-subset*)
**next**
**case** *EMPTY*
**have** $UNIV\ //\ (\approx\{[]\}) \subseteq \{\{[]\},\ UNIV - \{[]\}\}$
**by** (*rule quot-empty-subset*)
**with** *prems* **show** *?case* **by** (*auto intro*:*finite-subset*)
**next**
**case** (*CHAR c*)
**have** $UNIV\ //\ (\approx\{[c]\}) \subseteq \{\{[]\},\{[c]\},\ UNIV - \{[],\ [c]\}\}$
**by** (*rule quot-char-subset*)
**with** *prems* **show** *?case* **by** (*auto intro*:*finite-subset*)
**next**
**case** (*SEQ* $r_1$ $r_2$)
**have** $[\![\mathit{finite}\ (UNIV\ //\ \approx(L\ r_1));\ \mathit{finite}\ (UNIV\ //\ \approx(L\ r_2))]\!]$
$\Longrightarrow \mathit{finite}\ (UNIV\ //\ \approx(L\ r_1\ ;;\ L\ r_2))$
**by** (*erule quot-seq-finiteI*, *simp*)
**with** *prems* **show** *?case* **by** *simp*
**next**
**case** (*ALT* $r_1$ $r_2$)
**have** $[\![\mathit{finite}\ (UNIV\ //\ \approx(L\ r_1));\ \mathit{finite}\ (UNIV\ //\ \approx(L\ r_2))]\!]$
$\Longrightarrow \mathit{finite}\ (UNIV\ //\ \approx(L\ r_1 \cup L\ r_2))$
**by** (*erule quot-union-finiteI*, *simp*)
**with** *prems* **show** *?case* **by** *simp*
**next**
**case** (*STAR r*)
**have** $\mathit{finite}\ (UNIV\ //\ \approx(L\ r))$
$\Longrightarrow \mathit{finite}\ (UNIV\ //\ \approx((L\ r)\star))$
**by** (*erule quot-star-finiteI*)
**with** *prems* **show** *?case* **by** *simp*
**qed**

**end**