

tphols-2011

By xingyuan

February 3, 2011

Contents

| | | |
|----------|--|----------|
| 1 | Direction <i>regular language</i> \Rightarrow <i>finite partition</i> | 1 |
| 1.1 | The scheme | 1 |
| 1.2 | The proof | 6 |
| 1.2.1 | The base case for <i>NULL</i> | 6 |
| 1.2.2 | The base case for <i>EMPTY</i> | 6 |
| 1.2.3 | The base case for <i>CHAR</i> | 7 |
| 1.2.4 | The inductive case for <i>ALT</i> | 8 |
| 1.2.5 | The inductive case for <i>SEQ</i> | 8 |
| 1.2.6 | The inductive case for <i>STAR</i> | 12 |
| 1.2.7 | The conclusion | 19 |

```
theory Myhill
  imports Myhill-1
begin
```

1 Direction *regular language* \Rightarrow *finite partition*

1.1 The scheme

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

definition

$str\text{-}eq :: string \Rightarrow lang \Rightarrow string \Rightarrow bool$ ($- \approx -$)

where

$x \approx_{Lang} y \equiv (x, y) \in (\approx_{Lang})$

The main lemma (*rexp-imp-finite*) is proved by a structural induction over regular expressions. While base cases (cases for *NULL*, *EMPTY*, *CHAR*) are quite straight forward, the inductive cases are rather involved. What we have when starting to prove these inductive cases is that the partitions induced by the component language are finite. The basic idea to show the finiteness of the partition induced by the composite language is to attach a tag $tag(x)$ to every string x . The tags are made of equivalent classes from

the component partitions. Let tag be the tagging function and $Lang$ be the composite language, it can be proved that if strings with the same tag are equivalent with respect to $Lang$, expressed as:

$$tag(x) = tag(y) \implies x \approx_{Lang} y$$

then the partition induced by $Lang$ must be finite. There are two arguments for this. The first goes as the following:

1. First, the tagging function tag induces an equivalent relation $(=tag=)$ (definition of $f\text{-eq-rel}$ and lemma $equiv\text{-}f\text{-eq-rel}$).
2. It is shown that: if the range of tag (denoted $range(tag)$) is finite, the partition given rise by $(=tag=)$ is finite (lemma $finite\text{-}eq\text{-}f\text{-rel}$). Since tags are made from equivalent classes from component partitions, and the inductive hypothesis ensures the finiteness of these partitions, it is not difficult to prove the finiteness of $range(tag)$.
3. It is proved that if equivalent relation $R1$ is more refined than $R2$ (expressed as $R1 \subseteq R2$), and the partition induced by $R1$ is finite, then the partition induced by $R2$ is finite as well (lemma $refined\text{-}partition\text{-}finite$).
4. The injectivity assumption $tag(x) = tag(y) \implies x \approx_{Lang} y$ implies that $(=tag=)$ is more refined than (\approx_{Lang}) .
5. Combining the points above, we have: the partition induced by language $Lang$ is finite (lemma $tag\text{-}finite\text{-}imageD$).

definition

$f\text{-eq-rel}$ $(=-=)$

where

$(=f=) = \{(x, y) \mid x\ y.\ f\ x = f\ y\}$

lemma $equiv\text{-}f\text{-eq-rel:equiv\ UNIV\ (=f=)$

by $(auto\ simp:equiv\text{-}def\ f\text{-eq-rel}\text{-}def\ refl\text{-}on\text{-}def\ sym\text{-}def\ trans\text{-}def)$

lemma $finite\text{-}range\text{-}image: finite\ (range\ f) \implies finite\ (f\ 'A)$

by $(rule\text{-}tac\ B = \{y.\ \exists x.\ y = f\ x\}\ \mathbf{in}\ finite\text{-}subset,\ auto\ simp:image\text{-}def)$

lemma $finite\text{-}eq\text{-}f\text{-rel}$:

assumes $rng\text{-}fnt: finite\ (range\ tag)$

shows $finite\ (UNIV\ /\ (=\text{tag}=\))$

proof –

let $?f = op\ 'tag$ **and** $?A = (UNIV\ /\ (=\text{tag}=\))$

show $?thesis$

proof $(rule\text{-}tac\ f = ?f\ \mathbf{and}\ A = ?A\ \mathbf{in}\ finite\text{-}imageD)$

– The finiteness of f -image is a simple consequence of assumption $rng\text{-}fnt$:

show $finite\ (?f\ 'A)$

```

proof –
  have  $\forall X. ?f X \in (Pow (range tag))$  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-image is a consequence of the definition of (=tag=):
  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 f-eq-rel-def
          apply simp by blast
          with X-in Y-in show ?thesis
          by (auto simp:quotient-def str-eq-rel-def str-eq-def f-eq-rel-def)
        qed
      } thus ?thesis unfolding inj-on-def by auto
    qed
  qed
qed

```

lemma *finite-image-finite*: $\llbracket \forall x \in A. f x \in B; \text{finite } B \rrbracket \implies \text{finite } (f \text{ ` } A)$
by (*rule finite-subset [of - B], auto*)

lemma *refined-partition-finite*:
fixes *R1 R2 A*
assumes *fnt: finite (A // R1)*
and *refined: R1 ⊆ R2*
and *eq1: equiv A R1* **and** *eq2: equiv A R2*
shows *finite (A // R2)*

```

proof –
  let ?f = λ X. {R1 “ {x} | x. x ∈ X}
    and ?A = (A // R2) and ?B = (A // R1)
  show ?thesis
  proof(rule-tac f = ?f and A = ?A in finite-imageD)
    show finite (?f ` ?A)
    proof(rule finite-subset [of - Pow ?B])
      from fnt show finite (Pow (A // R1)) by simp
    next

```

```

from eq2
show ?f ‘ A // R2 ⊆ Pow ?B
  unfolding image-def Pow-def quotient-def
  apply auto
  by (rule-tac x = xb in beXI, simp,
      unfold equiv-def sym-def refl-on-def, blast)
qed
next
show inj-on ?f ?A
proof –
  { fix X Y
    assume X-in: X ∈ ?A and Y-in: Y ∈ ?A
    and eq-f: ?f X = ?f Y (is ?L = ?R)
    have X = Y using X-in
    proof(rule quotientE)
      fix x
      assume X = R2 “ {x} and x ∈ A with eq2
      have x-in: x ∈ X
        unfolding equiv-def quotient-def refl-on-def by auto
      with eq-f have R1 “ {x} ∈ ?R by auto
      then obtain y where
        y-in: y ∈ Y and eq-r: R1 “ {x} = R1 “ {y} by auto
      have (x, y) ∈ R1
      proof –
        from x-in X-in y-in Y-in eq2
        have x ∈ A and y ∈ A
          unfolding equiv-def quotient-def refl-on-def by auto
        from eq-equiv-class-iff [OF eq1 this] and eq-r
        show ?thesis by simp
      qed
      with refined have xy-r2: (x, y) ∈ R2 by auto
      from quotient-eqI [OF eq2 X-in Y-in x-in y-in this]
      show ?thesis .
    } thus ?thesis by (auto simp:inj-on-def)
  } qed
qed
qed
qed

```

lemma equiv-lang-eq: equiv UNIV (\approx Lang)
unfolding equiv-def str-eq-rel-def sym-def refl-on-def trans-def
by blast

lemma tag-finite-imageD:
fixes tag
assumes rng-fnt: finite (range tag)
 — Suppose the rang of tagging fuction tag is finite.
and same-tag-eqv: $\bigwedge m n. \text{tag } m = \text{tag } (n::\text{string}) \implies m \approx \text{Lang } n$
 — And strings with same tag are equivalent

```

shows finite (UNIV // ( $\approx$ Lang))
proof –
  let ?R1 = (=tag=)
  show ?thesis
proof(rule-tac refined-partition-finite [of - ?R1])
  from finite-eq-f-rel [OF rng-fnt]
  show finite (UNIV // =tag=) .
next
  from same-tag-eqv
  show (=tag=)  $\subseteq$  ( $\approx$ Lang)
  by (auto simp:f-eq-rel-def str-eq-def)
next
  from equiv-f-eq-rel
  show equiv UNIV (=tag=) by blast
next
  from equiv-lang-eq
  show equiv UNIV ( $\approx$ Lang) by blast
qed
qed

```

A more concise, but less intelligible argument for *tag-finite-imageD* is given as the following. The basic idea is still using standard library lemma *finite-imageD*:

$$\llbracket \text{finite } (f \text{ ' } A); \text{ inj-on } f \text{ } A \rrbracket \implies \text{finite } A$$

which says: if the image of injective function f over set A is finite, then A must be finite, as we did in the lemmas above.

lemma

fixes *tag*

assumes *rng-fnt*: *finite* (*range tag*)

— Suppose the range of tagging function *tag* is finite.

and *same-tag-eqv*: $\bigwedge m n. \text{tag } m = \text{tag } (n::\text{string}) \implies m \approx \text{Lang } n$

— And strings with same tag are equivalent

shows *finite* (*UNIV* // (\approx *Lang*))

— Then the partition generated by (\approx *Lang*) is finite.

proof –

— The particular f and A used in *finite-imageD* are:

let ? f = *op* ' *tag* **and** ? A = (*UNIV* // \approx *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 $\forall X. ?f X \in (\text{Pow } (\text{range } \text{tag}))$ **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-eqt:
  show inj-on ? $f$  ? $A$ 
  proof –
    { fix  $X Y$ 
      assume  $X\text{-in}: X \in ?A$ 
      and  $Y\text{-in}: Y \in ?A$ 
      and  $\text{tag-eq}: ?f X = ?f Y$ 
      have  $X = Y$ 
      proof –
        from  $X\text{-in} Y\text{-in} \text{tag-eq}$ 
        obtain  $x y$  where  $x\text{-in}: x \in X$  and  $y\text{-in}: y \in Y$  and  $\text{eq-tg}: \text{tag } x = \text{tag } y$ 
        unfolding quotient-def Image-def str-eq-rel-def str-eq-def image-def
        apply simp by blast
        from same-tag-eqt [OF eq-tg] have  $x \approx \text{Lang } y$  .
        with  $X\text{-in} Y\text{-in} x\text{-in} y\text{-in}$ 
        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 The proof

Each case is given in a separate section, as well as the final main lemma. Detailed explanations accompanied by illustrations are given for non-trivial cases.

For ever inductive case, there are two tasks, the easier one is to show the range finiteness of of the tagging function based on the finiteness of component partitions, the difficult one is to show that strings with the same tag are equivalent with respect to the composite language. Suppose the composite language be *Lang*, tagging function be *tag*, it amounts to show:

$$\text{tag}(x) = \text{tag}(y) \implies x \approx \text{Lang } y$$

expanding the definition of $\approx \text{Lang}$, it amounts to show:

$$\text{tag}(x) = \text{tag}(y) \implies (\forall z. x@z \in \text{Lang} \longleftrightarrow y@z \in \text{Lang})$$

Because the assumed tag equality $\text{tag}(x) = \text{tag}(y)$ is symmetric, it is sufficient to show just one direction:

$$\bigwedge x y z. \llbracket \text{tag}(x) = \text{tag}(y); x@z \in \text{Lang} \rrbracket \implies y@z \in \text{Lang}$$

This is the pattern followed by every inductive case.

1.2.1 The base case for *NULL*

lemma *quot-null-eq*:
shows $(UNIV // \approx\{\}) = (\{UNIV\}::lang\ set)$
unfolding *quotient-def Image-def str-eq-rel-def* **by** *auto*

lemma *quot-null-finiteI* [*intro*]:
shows *finite* $((UNIV // \approx\{\})::lang\ set)$
unfolding *quot-null-eq* **by** *simp*

1.2.2 The base case for *EMPTY*

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-empty-finiteI* [*intro*]:
shows *finite* $(UNIV // (\approx\{\}))$
by (*rule finite-subset[OF quot-empty-subset]*) (*simp*)

1.2.3 The base case for *CHAR*

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

```

```

lemma quot-char-finiteI [intro]:
  shows finite (UNIV // ( $\approx\{[c]\}$ ))
by (rule finite-subset[OF quot-char-subset]) (simp)

```

1.2.4 The inductive case for ALT

definition

```

tag-str-ALT :: lang  $\Rightarrow$  lang  $\Rightarrow$  string  $\Rightarrow$  (lang  $\times$  lang)
where
  tag-str-ALT L1 L2 = ( $\lambda x. (\approx L1 \text{ `` } \{x\}, \approx L2 \text{ `` } \{x\})$ )

```

```

lemma quot-union-finiteI [intro]:

```

```

  fixes L1 L2::lang
  assumes finite1: finite (UNIV //  $\approx L1$ )
  and    finite2: finite (UNIV //  $\approx L2$ )
  shows finite (UNIV //  $\approx (L1 \cup L2)$ )
proof (rule-tac tag = tag-str-ALT L1 L2 in tag-finite-imageD)
  show  $\bigwedge x y. \text{tag-str-ALT } L1 L2 x = \text{tag-str-ALT } L1 L2 y \implies x \approx (L1 \cup L2) y$ 
    unfolding tag-str-ALT-def
    unfolding str-eq-def
    unfolding Image-def
    unfolding str-eq-rel-def
    by auto

```

next

```

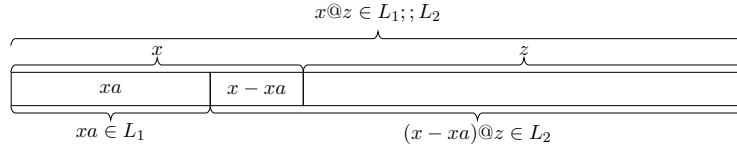
  have *: finite ((UNIV //  $\approx L1$ )  $\times$  (UNIV //  $\approx L2$ ))
    using finite1 finite2 by auto
  show finite (range (tag-str-ALT L1 L2))
    unfolding tag-str-ALT-def
    apply (rule finite-subset[OF - *])
    unfolding quotient-def
    by auto
qed

```

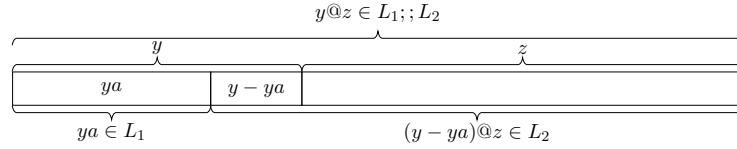
1.2.5 The inductive case for SEQ

For case SEQ, the language L is $L_1 ;; L_2$. Given $x @ z \in L_1 ;; L_2$, according to the definition of $L_1 ;; L_2$, string $x @ z$ can be splitted with the prefix in

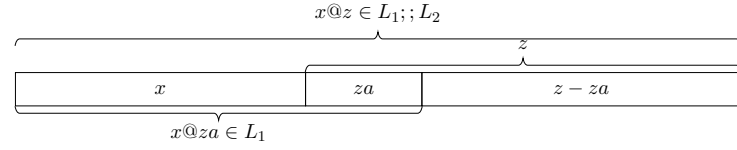
L_1 and suffix in L_2 . The split point can either be in x (as shown in Fig. 1(a)), or in z (as shown in Fig. 1(c)). Whichever way it goes, the structure on $x @ z$ can be transferred faithfully onto $y @ z$ (as shown in Fig. 1(b) and 1(d)) with the help of the assumed tag equality. The following tag function *tag-str-SEQ* is such designed to facilitate such transfers and lemma *tag-str-SEQ-injI* formalizes the informal argument above. The details of structure transfer will be given their.



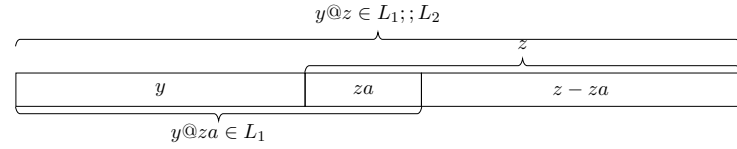
(a) First possible way to split $x@z$



(b) Transferred structure corresponding to the first way of splitting



(c) The second possible way to split $x@z$



(d) Transferred structure corresponding to the second way of splitting

Figure 1: The case for *SEQ*

definition

$tag\text{-}str\text{-}SEQ :: lang \Rightarrow lang \Rightarrow string \Rightarrow (lang \times lang\ set)$

where

$tag\text{-}str\text{-}SEQ\ L1\ L2 =$

$(\lambda x. (\approx L1\ \{\! \{x\}\!, \{(\approx L2\ \{\! \{x - xa\}\} \mid xa. xa \leq x \wedge xa \in L1)\})$)

The following is a techical lemma which helps to split the $x @ z \in L_1 ;; L_2$ mentioned above.

lemma *append-seq-elim*:

assumes $x @ y \in L_1 ;; L_2$

shows $(\exists xa \leq x. xa \in L_1 \wedge (x - xa) @ y \in L_2) \vee$
 $(\exists ya \leq y. (x @ ya) \in L_1 \wedge (y - ya) \in L_2)$
proof –
from *assms* **obtain** $s_1 s_2$
where *eq-xy*: $x @ y = s_1 @ s_2$
and *in-seq*: $s_1 \in L_1 \wedge s_2 \in L_2$
by (*auto simp:Seq-def*)
from *app-eq-dest* [*OF eq-xy*]
have
 $(x \leq s_1 \wedge (s_1 - x) @ s_2 = y) \vee (s_1 \leq x \wedge (x - s_1) @ y = s_2)$
 $(\text{is } ?Split1 \vee ?Split2) .$
moreover have $?Split1 \implies \exists ya \leq y. (x @ ya) \in L_1 \wedge (y - ya) \in L_2$
using *in-seq* **by** (*rule-tac x = s_1 - x in exI, auto elim:prefixE*)
moreover have $?Split2 \implies \exists xa \leq x. xa \in L_1 \wedge (x - xa) @ y \in L_2$
using *in-seq* **by** (*rule-tac x = s_1 in exI, auto*)
ultimately show *?thesis* **by** *blast*
qed

lemma *tag-str-SEQ-injI*:

fixes $v w$
assumes *eq-tag*: $tag\text{-str-SEQ } L_1 L_2 v = tag\text{-str-SEQ } L_1 L_2 w$
shows $v \approx (L_1 ;; L_2) w$

proof –

– As explained before, a pattern for just one direction needs to be dealt with:

{ fix $x y z$
assume *xz-in-seq*: $x @ z \in L_1 ;; L_2$
and *tag-xy*: $tag\text{-str-SEQ } L_1 L_2 x = tag\text{-str-SEQ } L_1 L_2 y$
have $y @ z \in L_1 ;; L_2$

proof –

– There are two ways to split $x@z$:

from *append-seq-elim* [*OF xz-in-seq*]
have $(\exists xa \leq x. xa \in L_1 \wedge (x - xa) @ z \in L_2) \vee$
 $(\exists za \leq z. (x @ za) \in L_1 \wedge (z - za) \in L_2) .$

– It can be shown that *?thesis* holds in either case:

moreover {

– The case for the first split:

fix xa
assume $h1: xa \leq x$ **and** $h2: xa \in L_1$ **and** $h3: (x - xa) @ z \in L_2$

– The following subgoal implements the structure transfer:

obtain ya
where $ya \leq y$
and $ya \in L_1$
and $(y - ya) @ z \in L_2$

proof –

By expanding the definition of

– $tag\text{-str-SEQ } L_1 L_2 x = tag\text{-str-SEQ } L_1 L_2 y$

and extracting the second component, we get:

have $\{\approx_{L_2} \text{ “ } \{x - xa\} \mid xa. xa \leq x \wedge xa \in L_1 \} =$
 $\{\approx_{L_2} \text{ “ } \{y - ya\} \mid ya. ya \leq y \wedge ya \in L_1 \}$ (**is** $?Left = ?Right$)
using *tag-xy unfolding tag-str-SEQ-def* **by** *simp*
— Since $xa \leq x$ and $xa \in L_1$ hold, it is not difficult to show:
moreover have $\approx_{L_2} \text{ “ } \{x - xa\} \in ?Left$ **using** *h1 h2* **by** *auto*
— Through tag equality, equivalent class $\approx_{L_2} \text{ “ } \{x - xa\}$
— also belongs to the $?Right$:
ultimately have $\approx_{L_2} \text{ “ } \{x - xa\} \in ?Right$ **by** *simp*
— From this, the counterpart of xa in y is obtained:
then obtain ya
where *eq-xya*: $\approx_{L_2} \text{ “ } \{x - xa\} = \approx_{L_2} \text{ “ } \{y - ya\}$
and *pref-ya*: $ya \leq y$ **and** *ya-in*: $ya \in L_1$
by *simp blast*
— It can be proved that ya has the desired property:
have $(y - ya)@z \in L_2$
proof —
from *eq-xya* **have** $(x - xa) \approx_{L_2} (y - ya)$
unfolding *Image-def str-eq-rel-def str-eq-def* **by** *auto*
with *h3* **show** $?thesis$ **unfolding** *str-eq-rel-def str-eq-def* **by** *simp*
qed
— Now, ya has all properties to be a qualified candidate:
with *pref-ya ya-in*
show $?thesis$ **using** *that* **by** *blast*
qed
— From the properties of ya , $y @ z \in L_1 ; ; L_2$ is derived easily.
hence $y @ z \in L_1 ; ; L_2$ **by** (*erule-tac prefixE, auto simp:Seq-def*)
} moreover {
— The other case is even more simpler:
fix za
assume *h1*: $za \leq z$ **and** *h2*: $(x @ za) \in L_1$ **and** *h3*: $z - za \in L_2$
have $y @ za \in L_1$
proof —
have $\approx_{L_1} \text{ “ } \{x\} = \approx_{L_1} \text{ “ } \{y\}$
using *tag-xy unfolding tag-str-SEQ-def* **by** *simp*
with *h2* **show** $?thesis$
unfolding *Image-def str-eq-rel-def str-eq-def* **by** *auto*
qed
with *h1 h3* **have** $y @ z \in L_1 ; ; L_2$
by (*drule-tac A = L_1 in seq-intro, auto elim:prefixE*)
}
ultimately show $?thesis$ **by** *blast*
qed
}
— $?thesis$ is proved by exploiting the symmetry of *eq-tag*:
from *this [OF - eq-tag]* **and** *this [OF - eq-tag [THEN sym]]*
show $?thesis$ **unfolding** *str-eq-def str-eq-rel-def* **by** *blast*
qed

lemma *quot-seq-finiteI* [*intro*]:

```

fixes  $L1\ L2::lang$ 
assumes  $fin1: finite\ (UNIV\ //\ \approx L1)$ 
and  $fin2: finite\ (UNIV\ //\ \approx L2)$ 
shows  $finite\ (UNIV\ //\ \approx(L1\ ;;\ L2))$ 
proof ( $rule-tac\ tag = tag-str-SEQ\ L1\ L2\ in\ tag-finite-imageD$ )
  show  $\bigwedge x\ y.\ tag-str-SEQ\ L1\ L2\ x = tag-str-SEQ\ L1\ L2\ y \implies x \approx(L1\ ;;\ L2)\ y$ 
    by ( $rule\ tag-str-SEQ-injI$ )
next
  have  $*$ :  $finite\ ((UNIV\ //\ \approx L1) \times (Pow\ (UNIV\ //\ \approx L2)))$ 
    using  $fin1\ fin2$  by  $auto$ 
  show  $finite\ (range\ (tag-str-SEQ\ L1\ L2))$ 
    unfolding  $tag-str-SEQ-def$ 
    apply( $rule\ finite-subset[OF\ -\ *]$ )
    unfolding  $quotient-def$ 
    by  $auto$ 
qed

```

1.2.6 The inductive case for $STAR$

This turned out to be the trickiest case. The essential goal is to prove $y @ z \in L_1^*$ under the assumptions that $x @ z \in L_1^*$ and that x and y have the same tag. The reasoning goes as the following:

1. Since $x @ z \in L_1^*$ holds, a prefix xa of x can be found such that $xa \in L_1^*$ and $(x - xa) @ z \in L_1^*$, as shown in Fig. 2(a). Such a prefix always exists, $xa = []$, for example, is one.
2. There could be many but finite many of such xa , from which we can find the longest and name it $xa-max$, as shown in Fig. 2(b).
3. The next step is to split z into za and zb such that $(x - xa-max) @ za \in L_1$ and $zb \in L_1^*$ as shown in Fig. 2(e). Such a split always exists because:
 - (a) Because $(x - xa-max) @ z \in L_1^*$, it can always be splitted into prefix a and suffix b , such that $a \in L_1$ and $b \in L_1^*$, as shown in Fig. 2(c).
 - (b) But the prefix a CANNOT be shorter than $x - xa-max$ (as shown in Fig. 2(d)), because otherwise, $xa-max @ a$ would be in the same kind as $xa-max$ but with a larger size, conflicting with the fact that $xa-max$ is the longest.
4. By the assumption that x and y have the same tag, the structure on $x @ z$ can be transferred to $y @ z$ as shown in Fig. 2(f). The detailed steps are:
 - (a) A y -prefix ya corresponding to xa can be found, which satisfies conditions: $ya \in L_1^*$ and $(y - ya) @ za \in L_1$.

- (b) Since we already know $zb \in L_1^*$, we get $(y - ya)@za@zb \in L_1^*$, and this is just $(y - ya)@z \in L_1^*$.
- (c) With fact $ya \in L_1^*$, we finally get $y@z \in L_1^*$.

The formal proof of lemma *tag-str-STAR-injI* faithfully follows this informal argument while the tagging function *tag-str-STAR* is defined to make the transfer in step ?? feasible.

definition

tag-str-STAR :: lang \Rightarrow string \Rightarrow lang set

where

tag-str-STAR L1 = ($\lambda x. \{\approx L1 \text{ “ } \{x - xa\} \mid xa. xa < x \wedge xa \in L1^*\}$)

A technical lemma.

lemma *finite-set-has-max*: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \Longrightarrow$

$(\exists \text{max} \in A. \forall a \in A. f a \leq (f \text{max} :: \text{nat}))$

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 *insertI.hyps* **and** *False*

obtain *max*

where *h1*: *max* \in A

and *h2*: $\forall a \in A. f a \leq f \text{max}$ **by** *blast*

show ?*thesis*

proof (*cases* f a \leq f *max*)

assume f a \leq f *max*

with *h1* *h2* **show** ?*thesis* **by** (*rule-tac* x = *max* **in** *beXI*, *auto*)

next

assume $\neg (f a \leq f \text{max})$

thus ?*thesis* **using** *h2* **by** (*rule-tac* x = a **in** *beXI*, *auto*)

qed

qed

qed

The following is a technical lemma. which helps to show the range finiteness of tag function.

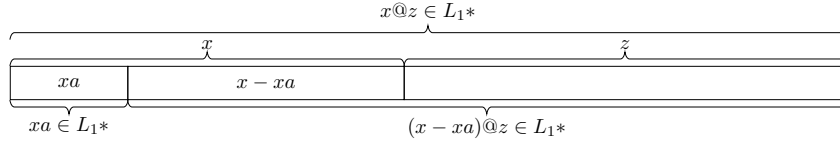
lemma *finite-strict-prefix-set*: *finite* {*xa*. *xa* < (*x*::*string*)}

apply (*induct* x *rule:rev-induct*, *simp*)

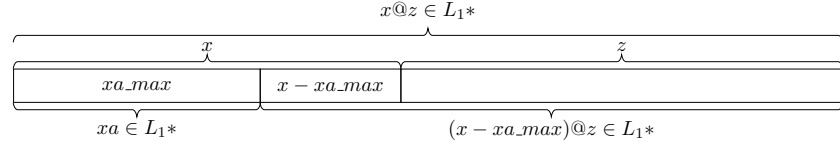
apply (*subgoal-tac* {*xa*. *xa* < *xs* @ [*x*]} = {*xa*. *xa* < *xs*} \cup {*xs*})

by (*auto* *simp:strict-prefix-def*)

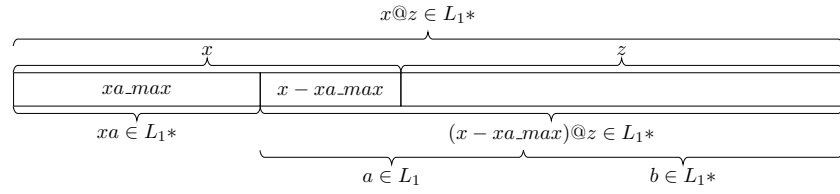
lemma *tag-str-STAR-injI*:



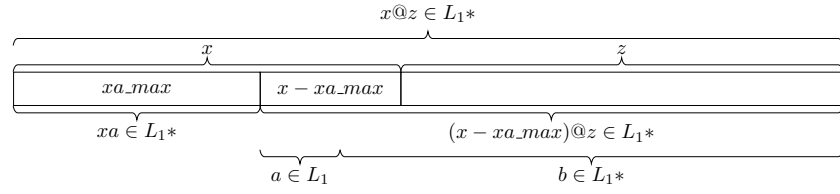
(a) First split



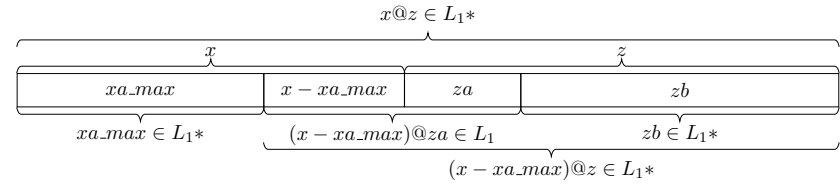
(b) Max split



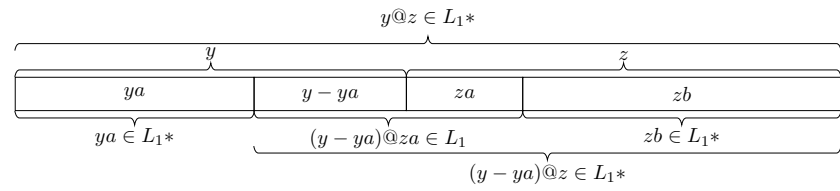
(c) Max split with a and b (the right situation)



(d) Max split with a and b (the wrong situation)



(e) Last split



(f) Structure transferred to y

Figure 2: The case for $STAR$

fixes $v w$
assumes $eq\text{-}tag: tag\text{-}str\text{-}STAR L_1 v = tag\text{-}str\text{-}STAR L_1 w$
shows $(v::string) \approx_{(L_1\star)} w$
proof –
 — As explained before, a pattern for just one direction needs to be dealt with:
{ fix $x y z$
assume $xz\text{-}in\text{-}star: x @ z \in L_1\star$
and $tag\text{-}xy: tag\text{-}str\text{-}STAR L_1 x = tag\text{-}str\text{-}STAR L_1 y$
have $y @ z \in L_1\star$
proof($cases\ x = []$)
 — The degenerated case when x is a null string is easy to prove:
case $True$
with $tag\text{-}xy$ **have** $y = []$
by ($auto\ simp\ add: tag\text{-}str\text{-}STAR\text{-}def\ strict\text{-}prefix\text{-}def$)
thus $?thesis$ **using** $xz\text{-}in\text{-}star\ True$ **by** $simp$
next
 — The nontrivial case:
case $False$
 Since $x @ z \in L_1\star$, x can always be splitted by a prefix xa together
 with its suffix $x - xa$, such that both xa and $(x - xa) @ z$ are
 in $L_1\star$, and there could be many such splittings. Therefore, the
 following set $?S$ is nonempty, and finite as well:
let $?S = \{xa. xa < x \wedge xa \in L_1\star \wedge (x - xa) @ z \in L_1\star\}$
have $finite\ ?S$
by ($rule\text{-}tac\ B = \{xa. xa < x\}$ **in** $finite\text{-}subset,$
 $auto\ simp: finite\text{-}strict\text{-}prefix\text{-}set$)
moreover **have** $?S \neq \{\}$ **using** $False\ xz\text{-}in\text{-}star$
by ($simp, rule\text{-}tac\ x = []$ **in** $exI, auto\ simp: strict\text{-}prefix\text{-}def$)
 — Since $?S$ is finite, we can always single out the longest and
 name it $xa\text{-}max$:
ultimately **have** $\exists xa\text{-}max \in ?S. \forall xa \in ?S. length\ xa \leq length\ xa\text{-}max$
using $finite\text{-}set\text{-}has\text{-}max$ **by** $blast$
then **obtain** $xa\text{-}max$
where $h1: xa\text{-}max < x$
and $h2: xa\text{-}max \in L_1\star$
and $h3: (x - xa\text{-}max) @ z \in L_1\star$
and $h4: \forall xa < x. xa \in L_1\star \wedge (x - xa) @ z \in L_1\star$
 $\longrightarrow length\ xa \leq length\ xa\text{-}max$
by $blast$
 — By the equality of tags, the counterpart of $xa\text{-}max$ among y -
 prefixes, named ya , can be found:
obtain ya
where $h5: ya < y$ **and** $h6: ya \in L_1\star$
and $eq\text{-}xya: (x - xa\text{-}max) \approx_{L_1} (y - ya)$
proof –
from $tag\text{-}xy$ **have** $\{\approx_{L_1} \{x - xa\} \mid xa. xa < x \wedge xa \in L_1\star\} =$
 $\{\approx_{L_1} \{y - xa\} \mid xa. xa < y \wedge xa \in L_1\star\}$ (**is** $?left = ?right$)
by ($auto\ simp: tag\text{-}str\text{-}STAR\text{-}def$)
moreover **have** $\approx_{L_1} \{x - xa\text{-}max\} \in ?left$ **using** $h1\ h2$ **by** $auto$
ultimately **have** $\approx_{L_1} \{x - xa\text{-}max\} \in ?right$ **by** $simp$
thus $?thesis$ **using** $that$

apply (*simp add:Image-def str-eq-rel-def str-eq-def*) **by blast**
qed
— The *?thesis*, $y @ z \in L_1\star$, is a simple consequence of the following proposition:
have $(y - ya) @ z \in L_1\star$
proof —
— The idea is to split the suffix z into za and zb , such that:
obtain $za\ zb$ **where** *eq-zab*: $z = za @ zb$
and *l-za*: $(y - ya)@za \in L_1$ **and** *ls-zb*: $zb \in L_1\star$
proof —
— Since $xa-max < x$, x can be splitted into a and b such that:
from *h1* **have** $(x - xa-max) @ z \neq []$
by (*auto simp:strict-prefix-def elim:prefixE*)
from *star-decom* [*OF h3 this*]
obtain $a\ b$ **where** *a-in*: $a \in L_1$
and *a-neg*: $a \neq []$ **and** *b-in*: $b \in L_1\star$
and *ab-max*: $(x - xa-max) @ z = a @ b$ **by blast**
— Now the candidates for za and zb are found:
let *?za* = $a - (x - xa-max)$ **and** *?zb* = b
have *pfx*: $(x - xa-max) \leq a$ (**is** *?P1*)
and *eq-z*: $z = ?za @ ?zb$ (**is** *?P2*)
proof —
— Since $(x - xa-max) @ z = a @ b$, string $(x - xa-max) @ z$ can be splitted in two ways:
have $((x - xa-max) \leq a \wedge (a - (x - xa-max)) @ b = z) \vee$
 $(a < (x - xa-max) \wedge ((x - xa-max) - a) @ z = b)$
using *app-eq-dest*[*OF ab-max*] **by** (*auto simp:strict-prefix-def*)
moreover {
— However, the undesired way can be refuted by absurdity:
assume *np*: $a < (x - xa-max)$
and *b-eqs*: $((x - xa-max) - a) @ z = b$
have *False*
proof —
let *?xa-max'* = $xa-max @ a$
have $?xa-max' < x$
using *np h1* **by** (*clarsimp simp:strict-prefix-def diff-prefix*)
moreover **have** $?xa-max' \in L_1\star$
using *a-in h2* **by** (*simp add:star-intro3*)
moreover **have** $(x - ?xa-max') @ z \in L_1\star$
using *b-eqs b-in np h1* **by** (*simp add:diff-diff-appd*)
moreover **have** $\neg (\text{length } ?xa-max' \leq \text{length } xa-max)$
using *a-neg* **by** *simp*
ultimately show *?thesis* **using** *h4* **by blast**
qed }
— Now it can be shown that the splitting goes the way we desired.
ultimately show *?P1* **and** *?P2* **by auto**
qed
hence $(x - xa-max)@?za \in L_1$ **using** *a-in* **by** (*auto elim:prefixE*)
— Now candidates *?za* and *?zb* have all the required properteis.
with *eq-xya* **have** $(y - ya) @ ?za \in L_1$


```

    by (auto simp:str-eq-def str-eq-rel-def)
    with eq-z and b-in
    show ?thesis using that by blast
qed
— ?thesis can easily be shown using properties of za and zb:
have ((y - ya) @ za) @ zb ∈ L1★ using l-za ls-zb by blast
with eq-zab show ?thesis by simp
qed
with h5 h6 show ?thesis
by (drule-tac star-intro1, auto simp:strict-prefix-def elim:prefixE)
qed
}
— By instantiating the reasoning pattern just derived for both directions:
from this [OF - eq-tag] and this [OF - eq-tag [THEN sym]]
— The thesis is proved as a trival consequence:
show ?thesis unfolding str-eq-def str-eq-rel-def by blast
qed

```

lemma — The original version with less explicit details.

```

fixes v w
assumes eq-tag: tag-str-STAR L1 v = tag-str-STAR L1 w
shows (v::string) ≈(L1★) w

```

proof—

According to the definition of \approx_{Lang} , proving $v \approx_{(L_1\star)} w$ amounts to showing: for any string u , if $v @ u \in (L_1\star)$ then $w @ u \in (L_1\star)$ and vice versa. The reasoning pattern for both directions are the same, as derived in the following:

```

{ 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 = [])

```

— The degenerated case when x is a null string is easy to prove:

```

  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

— The case when x is not null, and $x @ z$ is in $L_1\star$,

```

  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 $?S \neq \{\}$ using *False xz-in-star*
 by (simp, rule-tac $x = []$ in *exI*, *auto simp:strict-prefix-def*)
 ultimately have $\exists \text{max} \in ?S. \forall a \in ?S. \text{length } a \leq \text{length } \text{max}$
 using *finite-set-has-max* by blast
 thus *?thesis* using *that* by blast
 qed
 obtain *ya*
 where *h5*: $ya < y$ and *h6*: $ya \in L_1^*$ and *h7*: $(x - x\text{-max}) \approx_{L_1} (y - ya)$
 proof-
 from *tag-xy* have $\{\approx_{L_1} \text{“ } \{x - xa\} \mid xa. xa < x \wedge xa \in L_1^* \} =$
 $\{\approx_{L_1} \text{“ } \{y - xa\} \mid xa. xa < y \wedge xa \in L_1^* \}$ (is *?left = ?right*)
 by (*auto simp:tag-str-STAR-def*)
 moreover have $\approx_{L_1} \text{“ } \{x - x\text{-max}\} \in ?left$ using *h1 h2* by *auto*
 ultimately have $\approx_{L_1} \text{“ } \{x - x\text{-max}\} \in ?right$ by *simp*
 with *that* show *?thesis* apply
 (*simp add:Image-def str-eq-rel-def str-eq-def*) by blast
 qed
 have $(y - ya) @ z \in L_1^*$
 proof-
 from *h3 h1* obtain *a b* where *a-in*: $a \in L_1$
 and *a-neq*: $a \neq []$ and *b-in*: $b \in L_1^*$
 and *ab-max*: $(x - x\text{-max}) @ z = a @ b$
 by (*drule-tac star-decom*, *auto simp:strict-prefix-def elim:prefixE*)
 have $(x - x\text{-max}) \leq a \wedge (a - (x - x\text{-max})) @ b = z$
 proof -
 have $((x - x\text{-max}) \leq a \wedge (a - (x - x\text{-max})) @ b = z) \vee$
 $(a < (x - x\text{-max}) \wedge ((x - x\text{-max}) - a) @ z = b)$
 using *app-eq-dest[OF ab-max]* by (*auto simp:strict-prefix-def*)
 moreover {
 assume *np*: $a < (x - x\text{-max})$ and *b-eqs*: $((x - x\text{-max}) - a) @ z = b$
 have *False*
 proof -
 let $?x\text{-max}' = x\text{-max} @ a$
 have $?x\text{-max}' < x$
 using *np h1* by (*clarsimp simp:strict-prefix-def diff-prefix*)
 moreover have $?x\text{-max}' \in L_1^*$
 using *a-in h2* by (*simp add:star-intro3*)
 moreover have $(x - ?x\text{-max}') @ z \in L_1^*$
 using *b-eqs b-in np h1* by (*simp add:diff-diff-appd*)
 moreover have $\neg (\text{length } ?x\text{-max}' \leq \text{length } x\text{-max})$
 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*: $(x - x\text{-max}) @ za \in L_1$

```

    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 b-in have ((y - ya) @ za) @ b ∈ L1★ by blast
  with z-decom show ?thesis by auto
qed
with h5 h6 show ?thesis
  by (drule-tac star-intro1, auto simp:strict-prefix-def elim:prefixE)
qed
}
— By instantiating the reasoning pattern just derived for both directions:
from this [OF - eq-tag] and this [OF - eq-tag [THEN sym]]
— The thesis is proved as a trival consequence:
show ?thesis unfolding str-eq-def str-eq-rel-def by blast
qed

```

```

lemma quot-star-finiteI [intro]:
  fixes L1::lang
  assumes finite1: finite (UNIV // ≈L1)
  shows finite (UNIV // ≈(L1★))
proof (rule-tac tag = tag-str-STAR L1 in tag-finite-imageD)
  show ∧x y. tag-str-STAR L1 x = tag-str-STAR L1 y ⇒ x ≈(L1★) y
    by (rule tag-str-STAR-injI)
next
  have *: finite (Pow (UNIV // ≈L1))
    using finite1 by auto
  show finite (range (tag-str-STAR L1))
    unfolding tag-str-STAR-def
    apply(rule finite-subset[OF - *])
    unfolding quotient-def
    by auto
qed

```

1.2.7 The conclusion

```

lemma rexp-imp-finite:
  fixes r::rexp
  shows finite (UNIV // ≈(L r))
by (induct r) (auto)

end

```