

# Aho-Corasick String Matching

Arthur Freitas Ramos  
David Barros Hulak  
Ruy J. G. B. de Queiroz

May 6, 2026

## Abstract

This development formalizes the Aho-Corasick multi-pattern string matching algorithm [1] over lists. The verified executable reference automaton uses the canonical Aho-Corasick state: after reading a text prefix, the state is the longest suffix of that prefix that is also a pattern prefix. The development also proves a state-only transition theorem, a fold-based scan invariant, and a recursive failure-link transition refining the canonical transition, together with equivalent state-carrying and failure-link search procedures. The main theorem shows that the search procedure reports exactly the pattern occurrences ending at each text position. AI assistance was used for proof engineering; all final definitions, statements, and proofs are checked by Isabelle.

## Contents

<b>1</b>	<b>Aho-Corasick String Matching</b>	<b>1</b>
1.1	Pattern-prefix states . . . . .	2
1.2	The canonical Aho-Corasick state . . . . .	4
1.3	State-only transition . . . . .	5
1.4	Failure-link transition refinement . . . . .	7
1.5	Search and correctness . . . . .	12
1.6	Executable example . . . . .	14

```
theory Aho-Corasick  
  imports HOL-Library.Sublist  
begin
```

## 1 Aho-Corasick String Matching

The original Aho-Corasick string matching algorithm was introduced by Aho and Corasick [1]. It searches for all occurrences of a finite dictionary of patterns in a text. Its central automaton invariant is that, after reading a

text prefix, the current state is the longest suffix of the processed text that is also a prefix of a dictionary pattern.

This theory formalizes that canonical Aho-Corasick state and an executable reference search procedure over arbitrary lists. For each consumed text prefix, the reference transition recomputes the canonical state by scanning the finite list of pattern prefixes; this is the abstract automaton semantics that the usual failure-link implementation realizes more efficiently. We also derive a state-only transition theorem, a fold-based scan invariant, and a recursive failure-link transition that refines the canonical transition. The main theorem proves that the reports produced by the search are exactly the pattern occurrences ending at each text position.

AI assistance was used for proof engineering. The final definitions, statements, and proofs are checked by Isabelle.

## 1.1 Pattern-prefix states

**definition** *ac-states* :: 'a list list  $\Rightarrow$  'a list list **where**  
*ac-states* pats = remdups ([] # concat (map prefixes pats))

**lemma** *set-ac-states*:

*set (ac-states pats) = insert [] {q.  $\exists p \in \text{set pats. prefix } q \text{ } p$ }*  
**by** (*auto simp: ac-states-def*)

**lemma** *Nil-in-ac-states* [*simp*]: []  $\in$  set (ac-states pats)  
**by** (*simp add: set-ac-states*)

**fun** *longest-list* :: 'a list list  $\Rightarrow$  'a list **where**  
*longest-list* [] = []  
| *longest-list* (x # xs) =  
(*let y = longest-list xs in if length x < length y then y else x*)

**lemma** *longest-list-member*:

**assumes** *xs*  $\neq$  []  
**shows** *longest-list xs*  $\in$  set *xs*  
**using** *assms*

**proof** (*induction xs*)

**case** *Nil*

**then show** *?case* **by** *simp*

**next**

**case** (*Cons x xs*)

**show** *?case*

**proof** (*cases xs = []*)

**case** *True*

**then show** *?thesis* **by** *simp*

**next**

**case** *False*

**let** *?y = longest-list xs*

**from** *False* **have** *y-mem: ?y*  $\in$  set *xs*

```

    using Cons.IH by simp
  show ?thesis
  proof (cases length x < length ?y)
    case True
      then show ?thesis using y-mem by simp
    next
      case False
        then show ?thesis by simp
  qed
qed
qed

lemma longest-list-longest:
  assumes  $x \in \text{set } xs$ 
  shows  $\text{length } x \leq \text{length } (\text{longest-list } xs)$ 
  using assms
  proof (induction xs arbitrary: x)
    case Nil
      then show ?case by simp
  next
    case (Cons y ys)
      show ?case
      proof (cases ys = [])
        case True
          then show ?thesis using Cons.prems by auto
        next
          case False
            let ?z = longest-list ys
            have z-long:  $\bigwedge u. u \in \text{set } ys \implies \text{length } u \leq \text{length } ?z$ 
              using Cons.IH by blast
            show ?thesis
            proof (cases  $\text{length } y < \text{length } ?z$ )
              case True
                then show ?thesis using Cons.prems z-long by auto
              next
                case le: False
                  then have z-le-y:  $\text{length } ?z \leq \text{length } y$  by simp
                  show ?thesis
                  proof (cases  $x = y$ )
                    case True
                      then show ?thesis using le by auto
                    next
                      case neq: False
                        then have  $x \in \text{set } ys$ 
                          using Cons.prems by simp
                        then have  $\text{length } x \leq \text{length } ?z$ 
                          using z-long by simp
                        also have  $\dots \leq \text{length } y$ 
                          using z-le-y .

```

```

    finally show ?thesis using le by auto
  qed
qed
qed
qed

```

## 1.2 The canonical Aho-Corasick state

The canonical state for a word  $w$  is chosen among the finite list of all pattern prefixes. Since the empty list is always a state and always a suffix, the candidate list is nonempty. The selected state is therefore a suffix of  $w$ , a valid pattern-prefix state, and at least as long as any other valid suffix state.

**definition** *state-candidates* :: 'a list list  $\Rightarrow$  'a list  $\Rightarrow$  'a list list **where**  
*state-candidates* pats  $w = \text{filter } (\lambda q. \text{suffix } q \ w) \ (\text{ac-states } \text{pats})$

**definition** *ac-state* :: 'a list list  $\Rightarrow$  'a list  $\Rightarrow$  'a list **where**  
*ac-state* pats  $w = \text{longest-list } (\text{state-candidates } \text{pats } w)$

**lemma** *state-candidates-nonempty*: *state-candidates* pats  $w \neq []$

**proof** –  
 have  $[] \in \text{set } (\text{state-candidates } \text{pats } w)$   
 by (*simp add: state-candidates-def*)  
 then show ?thesis by auto  
**qed**

**lemma** *ac-state-in-states*: *ac-state* pats  $w \in \text{set } (\text{ac-states } \text{pats})$   
 using *longest-list-member*[*OF state-candidates-nonempty*]  
 by (*auto simp: ac-state-def state-candidates-def*)

**lemma** *ac-state-suffix*: *suffix* (*ac-state* pats  $w$ )  $w$   
 using *longest-list-member*[*OF state-candidates-nonempty*]  
 by (*auto simp: ac-state-def state-candidates-def*)

**lemma** *ac-state-longest*:  
 assumes  $q \in \text{set } (\text{ac-states } \text{pats})$   
 assumes *suffix*  $q \ w$   
 shows  $\text{length } q \leq \text{length } (\text{ac-state } \text{pats } w)$   
 using *assms longest-list-longest*[*of q state-candidates pats w*]  
 by (*auto simp: ac-state-def state-candidates-def*)

**lemma** *pattern-is-state*:  
 assumes  $p \in \text{set } \text{pats}$   
 shows  $p \in \text{set } (\text{ac-states } \text{pats})$   
 using *assms* by (*auto simp: set-ac-states*)

**lemma** *ac-state-snoc-prefix-closed*:  
 assumes  $r @ [x] \in \text{set } (\text{ac-states } \text{pats})$   
 shows  $r \in \text{set } (\text{ac-states } \text{pats})$   
 using *assms* by (*auto simp: set-ac-states dest: append-prefixD*)

**lemma** *pattern-suffix-ac-state-iff*:  
**assumes**  $p \in \text{set pats}$   
**shows**  $\text{suffix } p \text{ (ac-state pats } w) \longleftrightarrow \text{suffix } p \text{ } w$   
**proof**  
**assume**  $\text{suffix } p \text{ (ac-state pats } w)$   
**then show**  $\text{suffix } p \text{ } w$   
**using** *ac-state-suffix suffix-order.order-trans* **by** *blast*  
**next**  
**assume**  $p\text{-suffix: suffix } p \text{ } w$   
**have**  $\text{length } p \leq \text{length (ac-state pats } w)$   
**using** *ac-state-longest[OF pattern-is-state[OF assms] p-suffix]* .  
**then show**  $\text{suffix } p \text{ (ac-state pats } w)$   
**using** *suffix-length-suffix[OF p-suffix ac-state-suffix]* **by** *blast*  
**qed**

**lemma** *ac-state-Nil [simp]*:  $\text{ac-state pats } [] = []$   
**using** *ac-state-suffix[of pats []]* **by** *simp*

### 1.3 State-only transition

The next theorem captures the automaton nature of the construction. Although the canonical state is defined from the whole consumed text prefix, the next canonical state can be computed from only the current state and the next symbol.

**definition** *ac-step* ::  $'a \text{ list list} \Rightarrow 'a \text{ list} \Rightarrow 'a \Rightarrow 'a \text{ list}$  **where**  
 $\text{ac-step pats } q \text{ } x = \text{ac-state pats (} q @ [x])$

**lemma** *suffix-snoc-mono*:  
**assumes**  $\text{suffix } q \text{ } w$   
**shows**  $\text{suffix (} q @ [x]) \text{ (} w @ [x])$   
**using** *assms* **by** (*auto simp: suffix-def*)

**lemma** *state-suffix-next-window*:  
**assumes**  $q: q = \text{ac-state pats } w$   
**assumes**  $r\text{-state: } r \in \text{set (ac-states pats)}$   
**assumes**  $r\text{-suffix: suffix } r \text{ (} w @ [x])$   
**shows**  $\text{suffix } r \text{ (} q @ [x])$   
**proof** (*cases r rule: rev-cases*)  
**case** *Nil*  
**then show** *?thesis* **by** *simp*  
**next**  
**case** (*snoc*  $r' \text{ } y$ )  
**from**  $r\text{-suffix snoc}$  **have**  $y: y = x$  **and**  $r'\text{-suffix: suffix } r' \text{ } w$   
**by** *auto*  
**have**  $r'\text{-state: } r' \in \text{set (ac-states pats)}$   
**using**  $r\text{-state snoc}$  **by** (*auto intro: ac-state-snoc-prefix-closed*)  
**have**  $\text{length } r' \leq \text{length } q$   
**using** *ac-state-longest[OF r'-state r'-suffix]*  $q$  **by** *simp*

**moreover have**  $\text{suffix } q \ w$   
**using**  $q \ \text{ac-state-suffix}$  **by**  $\text{simp}$   
**ultimately have**  $\text{suffix } r' \ q$   
**using**  $\text{suffix-length-suffix}[OF \ r'\text{-suffix}]$  **by**  $\text{blast}$   
**with**  $\text{snoc } y$  **show**  $?thesis$  **by**  $\text{simp}$   
**qed**

**theorem**  $\text{ac-state-snoc}$ :

$\text{ac-state pats } (w \ @ \ [x]) = \text{ac-step pats } (\text{ac-state pats } w) \ x$

**proof** –

**let**  $?q = \text{ac-state pats } w$   
**let**  $?left = \text{ac-state pats } (w \ @ \ [x])$   
**let**  $?right = \text{ac-state pats } (?q \ @ \ [x])$   
**have**  $\text{left-in}: ?left \in \text{set } (\text{ac-states pats})$   
**by**  $(\text{rule } \text{ac-state-in-states})$   
**have**  $\text{left-suffix-full}: \text{suffix } ?left \ (w \ @ \ [x])$   
**by**  $(\text{rule } \text{ac-state-suffix})$   
**have**  $\text{left-suffix-q}: \text{suffix } ?left \ (?q \ @ \ [x])$   
**using**  $\text{state-suffix-next-window}$   
**where**  $q=?q$  **and**  $\text{pats}=\text{pats}$  **and**  $w=w$  **and**  $r=?left$  **and**  $x=x$   
 $\text{left-in } \text{left-suffix-full}$  **by**  $\text{simp}$   
**have**  $\text{right-suffix-q}: \text{suffix } ?right \ (?q \ @ \ [x])$   
**by**  $(\text{rule } \text{ac-state-suffix})$   
**have**  $\text{left-le-right}: \text{length } ?left \leq \text{length } ?right$   
**using**  $\text{ac-state-longest}[OF \ \text{left-in } \text{left-suffix-q}]$  .  
**have**  $\text{left-suffix-right}: \text{suffix } ?left \ ?right$   
**using**  $\text{suffix-length-suffix}[OF \ \text{left-suffix-q } \text{right-suffix-q } \text{left-le-right}]$  .

**have**  $q\text{-suffix}: \text{suffix } ?q \ w$   
**by**  $(\text{rule } \text{ac-state-suffix})$   
**have**  $\text{right-in}: ?right \in \text{set } (\text{ac-states pats})$   
**by**  $(\text{rule } \text{ac-state-in-states})$   
**have**  $\text{right-suffix-w}: \text{suffix } ?right \ (w \ @ \ [x])$   
**using**  $\text{suffix-snoc-mono}[OF \ q\text{-suffix}, \text{of } x] \ \text{right-suffix-q}$   
 $\text{suffix-order.order-trans}$  **by**  $\text{blast}$   
**have**  $\text{right-le-left}: \text{length } ?right \leq \text{length } ?left$   
**using**  $\text{ac-state-longest}[OF \ \text{right-in } \text{right-suffix-w}]$  .  
**have**  $\text{right-suffix-left}: \text{suffix } ?right \ ?left$   
**using**  $\text{suffix-length-suffix}[OF \ \text{right-suffix-w } \text{ac-state-suffix } \text{right-le-left}]$  .  
**have**  $?left = ?right$   
**using**  $\text{left-suffix-right } \text{right-suffix-left}$  **by**  $(\text{rule } \text{suffix-order.antisym})$   
**then show**  $?thesis$   
**by**  $(\text{simp add: } \text{ac-step-def})$   
**qed**

Iterating the state transition over a text fragment therefore tracks the canonical state for the whole consumed prefix.

**theorem**  $\text{ac-state-foldl-from}$ :

**assumes**  $q = \text{ac-state pats } w$

```

shows foldl (ac-step pats) q xs = ac-state pats (w @ xs)
using assms
proof (induction xs arbitrary: w q)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  let ?w' = w @ [x]
  have step: ac-step pats q x = ac-state pats ?w'
    using Cons.prem1 ac-state-snoc[of pats w x] by simp
  have foldl (ac-step pats) (ac-state pats ?w') xs =
    ac-state pats (?w' @ xs)
    using Cons.IH[where w=?w' and q=ac-state pats ?w'] by simp
  then show ?case
    using step by simp
qed

```

```

theorem ac-state-foldl:
  foldl (ac-step pats) [] xs = ac-state pats xs
  using ac-state-foldl-from[where pats=pats and w=[] and q=[] and xs=xs]
  by simp

```

```

theorem ac-state-foldl-take:
  foldl (ac-step pats) [] (take i text) = ac-state pats (take i text)
  by (simp add: ac-state-foldl)

```

## 1.4 Failure-link transition refinement

The failure link of a state is the longest proper suffix that is also a pattern-prefix state. The recursive failure-link transition follows these links until it finds a state that can consume the next symbol, or returns to the empty state. The main theorem of this subsection proves that this recognizable Aho-Corasick transition refines the canonical transition above.

```

definition proper-suffix-state-candidates :: 'a list list  $\Rightarrow$  'a list  $\Rightarrow$  'a list list where
  proper-suffix-state-candidates pats q =
    filter ( $\lambda r$ . suffix r q  $\wedge$  r  $\neq$  q) (ac-states pats)

```

```

definition ac-fail :: 'a list list  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  ac-fail pats q = longest-list (proper-suffix-state-candidates pats q)

```

```

lemma proper-suffix-state-candidates-nonempty:
  assumes q  $\neq$  []
  shows proper-suffix-state-candidates pats q  $\neq$  []
proof -
  have []  $\in$  set (proper-suffix-state-candidates pats q)
    using assms by (simp add: proper-suffix-state-candidates-def)
  then show ?thesis by auto
qed

```

**lemma** *ac-fail-in-states*:  
**assumes**  $q \neq []$   
**shows**  $ac\text{-fail pats } q \in set (ac\text{-states pats})$   
**using** *longest-list-member*[*OF proper-suffix-state-candidates-nonempty* [*OF assms*]]  
**by** (*auto simp: ac-fail-def proper-suffix-state-candidates-def*)

**lemma** *ac-fail-suffix*:  
**assumes**  $q \neq []$   
**shows**  $suffix (ac\text{-fail pats } q) q$   
**using** *longest-list-member*[*OF proper-suffix-state-candidates-nonempty* [*OF assms*]]  
**by** (*auto simp: ac-fail-def proper-suffix-state-candidates-def*)

**lemma** *ac-fail-proper*:  
**assumes**  $q \neq []$   
**shows**  $ac\text{-fail pats } q \neq q$   
**using** *longest-list-member*[*OF proper-suffix-state-candidates-nonempty* [*OF assms*]]  
**by** (*auto simp: ac-fail-def proper-suffix-state-candidates-def*)

**lemma** *length-ac-fail-less*:  
**assumes**  $q \neq []$   
**shows**  $length (ac\text{-fail pats } q) < length q$   
**proof** –  
**obtain** *zs* **where**  $q = zs @ ac\text{-fail pats } q$   
**using** *ac-fail-suffix*[*OF assms*] **by** (*auto simp: suffix-def*)  
**have** *zs-ne*:  $zs \neq []$   
**proof**  
**assume**  $zs = []$   
**then have**  $ac\text{-fail pats } q = q$   
**using** *q* **by** *simp*  
**then show** *False*  
**using** *ac-fail-proper*[*OF assms*] **by** *simp*  
**qed**  
**have**  $length q = length zs + length (ac\text{-fail pats } q)$   
**by** (*subst q, simp*)  
**moreover have**  $0 < length zs$   
**using** *zs-ne* **by** *simp*  
**ultimately show** *?thesis*  
**by** *simp*  
**qed**

**lemma** *ac-fail-longest*:  
**assumes**  $r \in set (ac\text{-states pats})$   
**assumes**  $suffix r q$   
**assumes**  $r \neq q$   
**shows**  $length r \leq length (ac\text{-fail pats } q)$   
**using** *assms longest-list-longest*[*of r proper-suffix-state-candidates pats q*]  
**by** (*auto simp: ac-fail-def proper-suffix-state-candidates-def*)

**lemma** *ac-state-self-if-state*:  
**assumes**  $q \in \text{set } (\text{ac-states pats})$   
**shows**  $\text{ac-state pats } q = q$   
**proof** –  
**have**  $q\text{-le-state: length } q \leq \text{length } (\text{ac-state pats } q)$   
**using**  $\text{ac-state-longest}[OF \text{ assms, of } q]$  **by** *simp*  
**have**  $\text{state-suffix-q: suffix } (\text{ac-state pats } q) q$   
**by**  $(\text{rule ac-state-suffix})$   
**have**  $q\text{-suffix-state: suffix } q (\text{ac-state pats } q)$   
**using**  $\text{suffix-length-suffix}[OF - \text{state-suffix-q } q\text{-le-state}]$  **by** *simp*  
**show** *?thesis*  
**using**  $\text{state-suffix-q } q\text{-suffix-state}$  **by**  $(\text{rule suffix-order.antisym})$   
**qed**

**lemma** *ac-state-singleton-if-no-state*:  
**assumes**  $[x] \notin \text{set } (\text{ac-states pats})$   
**shows**  $\text{ac-state pats } [x] = []$   
**proof** –  
**have**  $\text{state: ac-state pats } [x] \in \text{set } (\text{ac-states pats})$   
**by**  $(\text{rule ac-state-in-states})$   
**have**  $\text{suffix: suffix } (\text{ac-state pats } [x]) [x]$   
**by**  $(\text{rule ac-state-suffix})$   
**have**  $\text{ac-state pats } [x] = [] \vee \text{ac-state pats } [x] = [x]$   
**proof** –  
**obtain**  $zs$  **where**  $[x] = zs @ \text{ac-state pats } [x]$   
**using**  $\text{suffix}$  **by**  $(\text{auto simp: suffix-def})$   
**then show** *?thesis*  
**by**  $(\text{cases } zs; \text{cases ac-state pats } [x]) \text{ auto}$   
**qed**  
**then show** *?thesis*  
**using**  $\text{assms state}$  **by** *auto*  
**qed**

**lemma** *state-suffix-fail-next*:  
**assumes**  $q\text{-ne: } q \neq []$   
**assumes**  $qx\text{-not-state: } q @ [x] \notin \text{set } (\text{ac-states pats})$   
**assumes**  $r\text{-state: } r \in \text{set } (\text{ac-states pats})$   
**assumes**  $r\text{-suffix: suffix } r (q @ [x])$   
**shows**  $\text{suffix } r (\text{ac-fail pats } q @ [x])$   
**proof**  $(\text{cases } r \text{ rule: rev-cases})$   
**case** *Nil*  
**then show** *?thesis* **by** *simp*  
**next**  
**case**  $(\text{snoc } r' y)$   
**from**  $r\text{-suffix snoc}$  **have**  $y = x$  **and**  $r'\text{-suffix: suffix } r' q$   
**by** *auto*  
**have**  $r'\text{-state: } r' \in \text{set } (\text{ac-states pats})$   
**using**  $r\text{-state snoc}$  **by**  $(\text{auto intro: ac-state-snoc-prefix-closed})$   
**have**  $r'\text{-proper: } r' \neq q$

**using**  $qx\text{-not-state } r\text{-state } snoc\ y$  **by** *auto*  
**have**  $length\ r' \leq length\ (ac\text{-fail } pats\ q)$   
**using**  $ac\text{-fail-longest}[OF\ r'\text{-state } r'\text{-suffix } r'\text{-proper}]$  .  
**moreover** **have**  $suffix\ (ac\text{-fail } pats\ q)\ q$   
**using**  $ac\text{-fail-suffix}[OF\ q\text{-ne}]$  .  
**ultimately** **have**  $suffix\ r'\ (ac\text{-fail } pats\ q)$   
**using**  $suffix\text{-length-suffix}[OF\ r'\text{-suffix}]$  **by** *blast*  
**with**  $snoc\ y$  **show**  $?thesis$  **by** *simp*  
**qed**

**lemma** *ac-step-fail-eq*:

**assumes**  $q\text{-state}: q \in set\ (ac\text{-states } pats)$   
**assumes**  $q\text{-ne}: q \neq []$   
**assumes**  $qx\text{-not-state}: q @ [x] \notin set\ (ac\text{-states } pats)$   
**shows**  $ac\text{-step } pats\ (ac\text{-fail } pats\ q)\ x = ac\text{-step } pats\ q\ x$

**proof** –

**let**  $?fail = ac\text{-fail } pats\ q$   
**let**  $?left = ac\text{-state } pats\ (q @ [x])$   
**let**  $?right = ac\text{-state } pats\ (?fail @ [x])$   
**have**  $left\text{-in}: ?left \in set\ (ac\text{-states } pats)$   
**by** *(rule ac-state-in-states)*  
**have**  $left\text{-suffix-q}: suffix\ ?left\ (q @ [x])$   
**by** *(rule ac-state-suffix)*  
**have**  $left\text{-suffix-fail}: suffix\ ?left\ (?fail @ [x])$   
**using**  $state\text{-suffix-fail-next}[OF\ q\text{-ne } qx\text{-not-state } left\text{-in } left\text{-suffix-q}]$  .  
**have**  $right\text{-suffix-fail}: suffix\ ?right\ (?fail @ [x])$   
**by** *(rule ac-state-suffix)*  
**have**  $left\text{-le-right}: length\ ?left \leq length\ ?right$   
**using**  $ac\text{-state-longest}[OF\ left\text{-in } left\text{-suffix-fail}]$  .  
**have**  $left\text{-suffix-right}: suffix\ ?left\ ?right$   
**using**  $suffix\text{-length-suffix}[OF\ left\text{-suffix-fail } right\text{-suffix-fail } left\text{-le-right}]$  .  
  
**have**  $fail\text{-suffix-q}: suffix\ ?fail\ q$   
**using**  $ac\text{-fail-suffix}[OF\ q\text{-ne}]$  .  
**have**  $right\text{-in}: ?right \in set\ (ac\text{-states } pats)$   
**by** *(rule ac-state-in-states)*  
**have**  $right\text{-suffix-q}: suffix\ ?right\ (q @ [x])$   
**using**  $suffix\text{-snoc-mono}[OF\ fail\text{-suffix-q},\ of\ x]\ right\text{-suffix-fail}$   
 $suffix\text{-order.order-trans}$  **by** *blast*  
**have**  $right\text{-le-left}: length\ ?right \leq length\ ?left$   
**using**  $ac\text{-state-longest}[OF\ right\text{-in } right\text{-suffix-q}]$  .  
**have**  $right\text{-suffix-left}: suffix\ ?right\ ?left$   
**using**  $suffix\text{-length-suffix}[OF\ right\text{-suffix-q } ac\text{-state-suffix } right\text{-le-left}]$  .  
**have**  $?left = ?right$   
**using**  $left\text{-suffix-right } right\text{-suffix-left}$  **by** *(rule suffix-order.antisym)*  
**then** **show**  $?thesis$   
**by** *(simp add: ac-step-def)*  
**qed**

```

function ac-fail-step :: 'a list list  $\Rightarrow$  'a list  $\Rightarrow$  'a  $\Rightarrow$  'a list where
  ac-fail-step pats q x =
    (if q @ [x]  $\in$  set (ac-states pats) then q @ [x]
     else if q = [] then []
     else ac-fail-step pats (ac-fail pats q) x)
by pat-completeness auto
termination
  by (relation measure ( $\lambda$ (pats, q, x). length q))
    (auto simp: length-ac-fail-less)

theorem ac-fail-step-refines:
  assumes q  $\in$  set (ac-states pats)
  shows ac-fail-step pats q x = ac-step pats q x
  using assms
proof (induction pats q x rule: ac-fail-step.induct)
  case (1 pats q x)
  show ?case
  proof –
    consider
      (next-state) q @ [x]  $\in$  set (ac-states pats)
    | (root) q @ [x]  $\notin$  set (ac-states pats) q = []
    | (fail) q @ [x]  $\notin$  set (ac-states pats) q  $\neq$  []
    by blast
  then show ?thesis
  proof cases
    case next-state
    then show ?thesis
      by (simp add: ac-step-def ac-state-self-if-state)
  next
    case root
    then show ?thesis
      using ac-state-singleton-if-no-state[of x pats]
      by (simp add: ac-step-def)
  next
    case fail
    have fail-state: ac-fail pats q  $\in$  set (ac-states pats)
      using ac-fail-in-states[OF fail(2)] .
    have ac-fail-step pats q x = ac-fail-step pats (ac-fail pats q) x
      using fail by simp
    also have ... = ac-step pats (ac-fail pats q) x
      using 1.IH[OF fail fail-state] .
    also have ... = ac-step pats q x
      using ac-step-fail-eq[OF 1.prem1 fail(2) fail(1)] .
    finally show ?thesis .
  qed
qed
qed

```

**theorem** *ac-fail-state-foldl-from*:

```

assumes  $q = \text{ac-state pats } w$ 
shows  $\text{foldl } (\text{ac-fail-step pats}) \ q \ xs = \text{ac-state pats } (w @ xs)$ 
using  $\text{assms}$ 
proof ( $\text{induction } xs \text{ arbitrary: } w \ q$ )
  case  $\text{Nil}$ 
  then show  $?case$  by  $\text{simp}$ 
next
  case ( $\text{Cons } x \ xs$ )
  let  $?w' = w @ [x]$ 
  have  $q\text{-state: } q \in \text{set } (\text{ac-states pats})$ 
    using  $\text{Cons.prem } \text{ac-state-in-states}$  by  $\text{simp}$ 
  have  $\text{step: } \text{ac-fail-step pats } q \ x = \text{ac-state pats } ?w'$ 
    using  $\text{ac-fail-step-refines}[OF \ q\text{-state, of } x] \ \text{Cons.prem } \text{ac-state-snoc}[of \ \text{pats } w$ 
 $x]$ 
    by  $\text{simp}$ 
  have  $\text{rec: } \text{foldl } (\text{ac-fail-step pats}) \ (\text{ac-state pats } ?w') \ xs =$ 
 $\text{ac-state pats } (?w' @ xs)$ 
    by ( $\text{rule } \text{Cons.IH}$ )  $\text{simp}$ 
  have  $\text{foldl } (\text{ac-fail-step pats}) \ q \ (x \# xs) =$ 
 $\text{foldl } (\text{ac-fail-step pats}) \ (\text{ac-state pats } ?w') \ xs$ 
    by ( $\text{simp only: foldl.simps step}$ )
  also have  $\dots = \text{ac-state pats } (?w' @ xs)$ 
    using  $\text{rec}$  .
  also have  $\dots = \text{ac-state pats } (w @ x \# xs)$ 
    by  $\text{simp}$ 
  finally show  $?case$  .
qed

```

**theorem**  $\text{ac-fail-state-foldl}$ :

```

 $\text{foldl } (\text{ac-fail-step pats}) \ [] \ xs = \text{ac-state pats } xs$ 
using  $\text{ac-fail-state-foldl-from}[\text{where } \text{pats}=\text{pats} \ \text{and } w=[] \ \text{and } q=[] \ \text{and } xs=xs]$ 
by  $\text{simp}$ 

```

**theorem**  $\text{ac-fail-state-foldl-take}$ :

```

 $\text{foldl } (\text{ac-fail-step pats}) \ [] \ (\text{take } i \ \text{text}) = \text{ac-state pats } (\text{take } i \ \text{text})$ 
by ( $\text{simp add: ac-fail-state-foldl}$ )

```

## 1.5 Search and correctness

The output function filters the dictionary to the patterns that are suffixes of the current state. By the canonical-state invariant, this is equivalent to filtering the dictionary to the patterns that occur ending at the current text position.

**definition**  $\text{ac-output} :: 'a \ \text{list} \ \text{list} \Rightarrow 'a \ \text{list} \Rightarrow 'a \ \text{list} \ \text{list}$  **where**

```

 $\text{ac-output pats } q = \text{filter } (\lambda p. \text{suffix } p \ q) \ \text{pats}$ 

```

**theorem**  $\text{ac-output-state}$ :

```

 $\text{set } (\text{ac-output pats } (\text{ac-state pats } w)) =$ 
 $\{p. p \in \text{set pats} \wedge \text{suffix } p \ w\}$ 

```

by (auto simp: ac-output-def pattern-suffix-ac-state-iff)

```
fun ac-search-from ::  
  'a list list  $\Rightarrow$  'a list  $\Rightarrow$  nat  $\Rightarrow$  'a list  $\Rightarrow$  ('a list  $\times$  nat) list  
where  
  ac-search-from pats w i [] = []  
| ac-search-from pats w i (x # xs) =  
  (let w' = w @ [x];  
    q = ac-state pats w'  
    in map ( $\lambda p. (p, i)$ ) (ac-output pats q) @  
    ac-search-from pats w' (Suc i) xs)
```

**definition** ac-search :: 'a list list  $\Rightarrow$  'a list  $\Rightarrow$  ('a list  $\times$  nat) list **where**  
 ac-search pats text = ac-search-from pats [] 0 text

```
fun ac-search-states-from ::  
  'a list list  $\Rightarrow$  'a list  $\Rightarrow$  nat  $\Rightarrow$  'a list  $\Rightarrow$  ('a list  $\times$  nat) list  
where  
  ac-search-states-from pats q i [] = []  
| ac-search-states-from pats q i (x # xs) =  
  (let q' = ac-step pats q x in  
    map ( $\lambda p. (p, i)$ ) (ac-output pats q') @  
    ac-search-states-from pats q' (Suc i) xs)
```

**definition** ac-search-states :: 'a list list  $\Rightarrow$  'a list  $\Rightarrow$  ('a list  $\times$  nat) list **where**  
 ac-search-states pats text = ac-search-states-from pats [] 0 text

```
fun ac-search-fail-from ::  
  'a list list  $\Rightarrow$  'a list  $\Rightarrow$  nat  $\Rightarrow$  'a list  $\Rightarrow$  ('a list  $\times$  nat) list  
where  
  ac-search-fail-from pats q i [] = []  
| ac-search-fail-from pats q i (x # xs) =  
  (let q' = ac-fail-step pats q x in  
    map ( $\lambda p. (p, i)$ ) (ac-output pats q') @  
    ac-search-fail-from pats q' (Suc i) xs)
```

**definition** ac-search-fail :: 'a list list  $\Rightarrow$  'a list  $\Rightarrow$  ('a list  $\times$  nat) list **where**  
 ac-search-fail pats text = ac-search-fail-from pats [] 0 text

```
lemma ac-search-states-from-eq:  
  assumes q = ac-state pats w  
  shows ac-search-states-from pats q i xs = ac-search-from pats w i xs  
  using assms  
proof (induction xs arbitrary: w q i)  
  case Nil  
  then show ?case by simp  
next  
  case (Cons x xs)  
  let ?w' = w @ [x]
```

```

have step: ac-step pats q x = ac-state pats ?w'
  using Cons.prems ac-state-snoc[of pats w x] by simp
have rec:
  ac-search-states-from pats (ac-state pats ?w') (Suc i) xs =
    ac-search-from pats ?w' (Suc i) xs
  using Cons.IH[where w=?w' and q=ac-state pats ?w' and i=Suc i] by simp
show ?case
  using step rec by (simp add: Let-def)
qed

```

```

theorem ac-search-states-eq:
  ac-search-states pats text = ac-search pats text
  unfolding ac-search-states-def ac-search-def
  by (rule ac-search-states-from-eq) simp

```

```

lemma ac-search-fail-from-eq:
  assumes q = ac-state pats w
  shows ac-search-fail-from pats q i xs = ac-search-from pats w i xs
  using assms
proof (induction xs arbitrary: w q i)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  let ?w' = w @ [x]
  have q-state: q ∈ set (ac-states pats)
    using Cons.prems ac-state-in-states by simp
  have step: ac-fail-step pats q x = ac-state pats ?w'
    using ac-fail-step-refines[OF q-state, of x] Cons.prems ac-state-snoc[of pats w
x]
    by simp
  have rec:
    ac-search-fail-from pats (ac-state pats ?w') (Suc i) xs =
      ac-search-from pats ?w' (Suc i) xs
    using Cons.IH[where w=?w' and q=ac-state pats ?w' and i=Suc i] by simp
  show ?case
    using step rec by (simp add: Let-def)
qed

```

```

theorem ac-search-fail-eq:
  ac-search-fail pats text = ac-search pats text
  unfolding ac-search-fail-def ac-search-def
  by (rule ac-search-fail-from-eq) simp

```

## 1.6 Executable example

The following concrete input exercises overlapping dictionary patterns for both executable search procedures. The *eval* proof method compiles the definitions to ML, evaluates them, and checks the resulting equations.

**lemma** *ac-search-states-example*:

```
ac-search-states [[0], [0,1], [1,0,1], [1,2], [1,2,0], [2], [2,0,0]]
  [0,1,2,2,0,1::nat] =
  [[0], 0], ([0, 1], 1), ([1, 2], 2), ([2], 2), ([2], 3), ([0], 4), ([0, 1], 5)]
by eval
```

**lemma** *ac-search-fail-example*:

```
ac-search-fail [[0], [0,1], [1,0,1], [1,2], [1,2,0], [2], [2,0,0]]
  [0,1,2,2,0,1::nat] =
  [[0], 0], ([0, 1], 1), ([1, 2], 2), ([2], 2), ([2], 3), ([0], 4), ([0, 1], 5)]
by eval
```

**definition** *occurs-ending-at* :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  nat  $\Rightarrow$  bool **where**

```
occurs-ending-at pat text i  $\longleftrightarrow$ 
  i < length text  $\wedge$  suffix pat (take (Suc i) text)
```

**lemma** *in-ac-search-from-iff*:

```
(p, k)  $\in$  set (ac-search-from pats w i xs)  $\longleftrightarrow$ 
  ( $\exists$  j < length xs.
   k = i + j  $\wedge$ 
   p  $\in$  set pats  $\wedge$ 
   suffix p (ac-state pats (w @ take (Suc j) xs)))
```

**proof** (*induction xs arbitrary: w i k*)

**case** *Nil*

**then show** ?case **by** *simp*

**next**

**case** (*Cons x xs*)

**let** ?w' = w @ [x]

**show** ?case (**is** ?lhs  $\longleftrightarrow$  ?rhs)

**proof**

**assume** ?lhs

**then have** *mem*:

```
(p, k)  $\in$ 
  set (map ( $\lambda$ p. (p, i)) (ac-output pats (ac-state pats ?w'))) @
  ac-search-from pats ?w' (Suc i) xs
```

**by** (*simp add: Let-def*)

**have** *out-or-rec*:

```
p  $\in$  set (ac-output pats (ac-state pats ?w'))  $\wedge$  k = i  $\vee$ 
  (p, k)  $\in$  set (ac-search-from pats ?w' (Suc i) xs)
```

**using** *mem* **by** *auto*

**then show** ?rhs

**proof**

**assume** p  $\in$  set (ac-output pats (ac-state pats ?w'))  $\wedge$  k = i

**then have** p: p  $\in$  set pats

**and** k: k = i

**and** s: suffix p (ac-state pats ?w')

**by** (*auto simp: ac-output-def*)

**then show** ?rhs

**using** p k s **by** (*intro exI[of -] simp*)

```

next
  assume rec:  $(p, k) \in \text{set } (\text{ac-search-from pats } ?w' (\text{Suc } i) \text{ xs})$ 
  then obtain j where  $j: j < \text{length } \text{xs}$ 
    and  $k: k = \text{Suc } i + j$ 
    and  $p: p \in \text{set pats}$ 
    and  $s: \text{suffix } p (\text{ac-state pats } (?w' @ \text{take } (\text{Suc } j) \text{ xs}))$ 
    using Cons.IH[where  $w=?w'$  and  $i=\text{Suc } i$  and  $k=k$ ] by auto
  have  $?w' @ \text{take } (\text{Suc } j) \text{ xs} = w @ \text{take } (\text{Suc } (\text{Suc } j)) (x \# \text{xs})$ 
    by simp
  moreover have  $k = i + \text{Suc } j$ 
    using k by simp
  ultimately show ?rhs
    using j p s by (intro exI[of - Suc j]) auto
qed
next
  assume ?rhs
  then obtain j where  $j: j < \text{length } (x \# \text{xs})$ 
    and  $k: k = i + j$ 
    and  $p: p \in \text{set pats}$ 
    and  $s: \text{suffix } p (\text{ac-state pats } (w @ \text{take } (\text{Suc } j) (x \# \text{xs})))$ 
    by auto
  show ?lhs
  proof (cases j)
    case 0
      then have  $p \in \text{set } (\text{ac-output pats } (\text{ac-state pats } ?w'))$ 
        using p s by (simp add: ac-output-def)
      with 0 k show ?thesis by (simp add: Let-def)
    next
      case (Suc j')
        then have  $j': j' < \text{length } \text{xs}$ 
          using j by simp
        have  $?w' @ \text{take } (\text{Suc } j') \text{ xs} = w @ \text{take } (\text{Suc } j) (x \# \text{xs})$ 
          using Suc by simp
        then have rec-suffix:
           $\text{suffix } p (\text{ac-state pats } (?w' @ \text{take } (\text{Suc } j') \text{ xs}))$ 
          using s by simp
        have  $(p, k) \in \text{set } (\text{ac-search-from pats } ?w' (\text{Suc } i) \text{ xs})$ 
          using Cons.IH[where  $w=?w'$  and  $i=\text{Suc } i$  and  $k=k$ ] j' k p rec-suffix Suc
  by auto
  then show ?thesis by (simp add: Let-def)
qed
qed
qed

theorem ac-search-correct:
   $\text{set } (\text{ac-search pats text}) =$ 
   $\{(p, i). p \in \text{set pats} \wedge \text{occurs-ending-at } p \text{ text } i\}$ 
proof
  show  $\text{set } (\text{ac-search pats text}) \subseteq$ 

```

```

    {(p, i). p ∈ set pats ∧ occurs-ending-at p text i}
  proof
    fix x
    assume xin: x ∈ set (ac-search pats text)
    obtain p i where x: x = (p, i)
      by (cases x)
    have pin: (p, i) ∈ set (ac-search pats text)
      using xin by (simp add: x)
    then have pin': (p, i) ∈ set (ac-search-from pats [] 0 text)
      by (simp add: ac-search-def)
    then obtain j where j: j < length text
      and i: i = j
      and p: p ∈ set pats
      and s: suffix p (ac-state pats (take (Suc j) text))
      by (auto simp: in-ac-search-from-iff)
    have suffix p (take (Suc j) text)
      using pattern-suffix-ac-state-iff[OF p, of take (Suc j) text] s by simp
    then have suffix p (take (Suc i) text)
      using i by simp
    with j i p show x ∈ {(p, i). p ∈ set pats ∧ occurs-ending-at p text i}
      by (auto simp: x occurs-ending-at-def)
  qed
next
show {(p, i). p ∈ set pats ∧ occurs-ending-at p text i} ⊆
  set (ac-search pats text)
proof
  fix x
  assume xin: x ∈ {(p, i). p ∈ set pats ∧ occurs-ending-at p text i}
  obtain p i where x: x = (p, i)
    by (cases x)
  from xin have p: p ∈ set pats
    and i: i < length text
    and s: suffix p (take (Suc i) text)
    by (auto simp: x occurs-ending-at-def)
  have suffix p (ac-state pats (take (Suc i) text))
    using pattern-suffix-ac-state-iff[OF p, of take (Suc i) text] s by simp
  with p i have (p, i) ∈ set (ac-search-from pats [] 0 text)
    by (auto simp: in-ac-search-from-iff)
  then show x ∈ set (ac-search pats text)
    by (simp add: x ac-search-def)
qed
qed

theorem ac-search-states-correct:
  set (ac-search-states pats text) =
    {(p, i). p ∈ set pats ∧ occurs-ending-at p text i}
  by (simp add: ac-search-states-eq ac-search-correct)

theorem ac-search-fail-correct:

```

```
set (ac-search-fail pats text) =  
  {(p, i). p ∈ set pats ∧ occurs-ending-at p text i}  
by (simp add: ac-search-fail-eq ac-search-correct)
```

**end**

## References

- [1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. DOI: 10.1145/360825.360855.