

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

1	Aho-Corasick String Matching	1
1.1	Pattern-prefix states	2
1.2	The canonical Aho-Corasick state	2
1.3	State-only transition	3
1.4	Failure-link transition refinement	4
1.5	Search and correctness	6
1.6	Executable example	8

```
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. $\exists p \in \text{set pats. prefix } q \text{ } p$ }
<proof>

lemma *Nil-in-ac-states [simp]*: [] \in *set (ac-states pats)*
<proof>

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*
<proof>

lemma *longest-list-longest*:

assumes *x* \in *set xs*
shows *length x* \leq *length (longest-list xs)*
<proof>

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 = filter (λq . suffix q w) (ac-states pats)

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

lemma *state-candidates-nonempty*: state-candidates pats w \neq []
 <proof>

lemma *ac-state-in-states*: ac-state pats w \in set (ac-states pats)
 <proof>

lemma *ac-state-suffix*: suffix (ac-state pats w) w
 <proof>

lemma *ac-state-longest*:
 assumes q \in set (ac-states pats)
 assumes suffix q w
 shows length q \leq length (ac-state pats w)
 <proof>

lemma *pattern-is-state*:
 assumes p \in set pats
 shows p \in set (ac-states pats)
 <proof>

lemma *ac-state-snoc-prefix-closed*:
 assumes r @ [x] \in set (ac-states pats)
 shows r \in set (ac-states pats)
 <proof>

lemma *pattern-suffix-ac-state-iff*:
 assumes p \in set pats
 shows suffix p (ac-state pats w) \longleftrightarrow suffix p w
 <proof>

lemma *ac-state-Nil [simp]*: ac-state pats [] = []
 <proof>

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 list list \Rightarrow 'a list \Rightarrow 'a \Rightarrow 'a list **where**
ac-step pats q x = ac-state pats (q @ [x])

lemma *suffix-snoc-mono*:

assumes *suffix* $q\ w$
shows *suffix* $(q\ @\ [x])\ (w\ @\ [x])$
 \langle *proof* \rangle

lemma *state-suffix-next-window*:
assumes $q: q = \text{ac-state pats } w$
assumes *r-state*: $r \in \text{set } (\text{ac-states pats})$
assumes *r-suffix*: *suffix* $r\ (w\ @\ [x])$
shows *suffix* $r\ (q\ @\ [x])$
 \langle *proof* \rangle

theorem *ac-state-snoc*:
 $\text{ac-state pats } (w\ @\ [x]) = \text{ac-step pats } (\text{ac-state pats } w)\ x$
 \langle *proof* \rangle

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

theorem *ac-state-foldl-from*:
assumes $q = \text{ac-state pats } w$
shows *foldl* $(\text{ac-step pats})\ q\ xs = \text{ac-state pats } (w\ @\ xs)$
 \langle *proof* \rangle

theorem *ac-state-foldl*:
 $\text{foldl } (\text{ac-step pats})\ []\ xs = \text{ac-state pats } xs$
 \langle *proof* \rangle

theorem *ac-state-foldl-take*:
 $\text{foldl } (\text{ac-step pats})\ []\ (\text{take } i\ \text{text}) = \text{ac-state pats } (\text{take } i\ \text{text})$
 \langle *proof* \rangle

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

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

lemma *proper-suffix-state-candidates-nonempty*:
assumes $q \neq []$
shows *proper-suffix-state-candidates* $pats\ q \neq []$
 \langle *proof* \rangle

lemma *ac-fail-in-states*:
assumes $q \neq []$
shows $ac\text{-fail pats } q \in set (ac\text{-states pats})$
 $\langle proof \rangle$

lemma *ac-fail-suffix*:
assumes $q \neq []$
shows $suffix (ac\text{-fail pats } q) q$
 $\langle proof \rangle$

lemma *ac-fail-proper*:
assumes $q \neq []$
shows $ac\text{-fail pats } q \neq q$
 $\langle proof \rangle$

lemma *length-ac-fail-less*:
assumes $q \neq []$
shows $length (ac\text{-fail pats } q) < length q$
 $\langle proof \rangle$

lemma *ac-fail-longest*:
assumes $r \in set (ac\text{-states pats})$
assumes $suffix r q$
assumes $r \neq q$
shows $length r \leq length (ac\text{-fail pats } q)$
 $\langle proof \rangle$

lemma *ac-state-self-if-state*:
assumes $q \in set (ac\text{-states pats})$
shows $ac\text{-state pats } q = q$
 $\langle proof \rangle$

lemma *ac-state-singleton-if-no-state*:
assumes $[x] \notin set (ac\text{-states pats})$
shows $ac\text{-state pats } [x] = []$
 $\langle proof \rangle$

lemma *state-suffix-fail-next*:
assumes $q\text{-ne: } q \neq []$
assumes $qx\text{-not-state: } q @ [x] \notin set (ac\text{-states pats})$
assumes $r\text{-state: } r \in set (ac\text{-states pats})$
assumes $r\text{-suffix: } suffix r (q @ [x])$
shows $suffix r (ac\text{-fail pats } q @ [x])$
 $\langle proof \rangle$

lemma *ac-step-fail-eq*:
assumes $q\text{-state: } q \in set (ac\text{-states pats})$
assumes $q\text{-ne: } q \neq []$

assumes $qx\text{-not-state}: q @ [x] \notin \text{set } (ac\text{-states } pats)$
shows $ac\text{-step } pats (ac\text{-fail } pats q) x = ac\text{-step } pats q x$
 $\langle \text{proof} \rangle$

function $ac\text{-fail-step} :: 'a \text{ list } list \Rightarrow 'a \text{ list} \Rightarrow 'a \Rightarrow 'a \text{ list}$ **where**
 $ac\text{-fail-step } pats q x =$
 (if $q @ [x] \in \text{set } (ac\text{-states } pats)$ then $q @ [x]$
 else if $q = []$ then $[]$
 else $ac\text{-fail-step } pats (ac\text{-fail } pats q) x$)
 $\langle \text{proof} \rangle$

termination
 $\langle \text{proof} \rangle$

theorem $ac\text{-fail-step-refines}$:
assumes $q \in \text{set } (ac\text{-states } pats)$
shows $ac\text{-fail-step } pats q x = ac\text{-step } pats q x$
 $\langle \text{proof} \rangle$

theorem $ac\text{-fail-state-foldl-from}$:
assumes $q = ac\text{-state } pats w$
shows $foldl (ac\text{-fail-step } pats) q xs = ac\text{-state } pats (w @ xs)$
 $\langle \text{proof} \rangle$

theorem $ac\text{-fail-state-foldl}$:
 $foldl (ac\text{-fail-step } pats) [] xs = ac\text{-state } pats xs$
 $\langle \text{proof} \rangle$

theorem $ac\text{-fail-state-foldl-take}$:
 $foldl (ac\text{-fail-step } pats) [] (take i text) = ac\text{-state } pats (take i text)$
 $\langle \text{proof} \rangle$

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 $ac\text{-output} :: 'a \text{ list } list \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list } list$ **where**
 $ac\text{-output } pats q = \text{filter } (\lambda p. \text{suffix } p q) pats$

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

fun $ac\text{-search-from} ::$
 $'a \text{ list } list \Rightarrow 'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'a \text{ list} \Rightarrow ('a \text{ list} \times \text{nat}) \text{ list}$
where
 $ac\text{-search-from } pats w i [] = []$

| *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*
 <proof>

theorem *ac-search-states-eq*:
ac-search-states pats *text* = *ac-search* pats *text*
 <proof>

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*
 <proof>

theorem *ac-search-fail-eq*:
ac-search-fail pats *text* = *ac-search* pats *text*

$\langle proof \rangle$

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 $[[0], [0,1], [1,0,1], [1,2], [1,2,0], [2], [2,0,0]]$
 $[0,1,2,2,0,1::nat] =$
 $[[([0], 0), ([0, 1], 1), ([1, 2], 2), ([2], 2), ([2], 3), ([0], 4), ([0, 1], 5)]$
 $\langle proof \rangle$

lemma *ac-search-fail-example*:

ac-search-fail $[[0], [0,1], [1,0,1], [1,2], [1,2,0], [2], [2,0,0]]$
 $[0,1,2,2,0,1::nat] =$
 $[[([0], 0), ([0, 1], 1), ([1, 2], 2), ([2], 2), ([2], 3), ([0], 4), ([0, 1], 5)]$
 $\langle proof \rangle$

definition *occurs-ending-at* :: 'a list \Rightarrow 'a list \Rightarrow nat \Rightarrow bool **where**
occurs-ending-at pat text $i \longleftrightarrow$
 $i < \text{length text} \wedge \text{suffix pat (take (Suc } i) \text{ text)}$

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

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

theorem *ac-search-correct*:

$\text{set (ac-search pats text)} =$
 $\{(p, i). p \in \text{set pats} \wedge \text{occurs-ending-at } p \text{ text } i\}$
 $\langle proof \rangle$

theorem *ac-search-states-correct*:

$\text{set (ac-search-states pats text)} =$
 $\{(p, i). p \in \text{set pats} \wedge \text{occurs-ending-at } p \text{ text } i\}$
 $\langle proof \rangle$

theorem *ac-search-fail-correct*:

$\text{set (ac-search-fail pats text)} =$
 $\{(p, i). p \in \text{set pats} \wedge \text{occurs-ending-at } p \text{ text } i\}$
 $\langle proof \rangle$

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.