Aho-Corasick String Matching

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

June 25, 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

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 ⇒ 'a list list where
  ac-states pats = remdups ([] # concat (map prefixes pats))

lemma set-ac-states:
  set (ac-states pats) = insert [] {q. ∃p∈set pats. prefix q p}
  by (auto simp: ac-states-def)

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

fun longest-list :: 'a list list ⇒ '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 ≠ []
  shows longest-list xs ∈ 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 ∈ 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 ∈ set xs
  shows length x ≤ length (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: ⋀u. u ∈ set ys ⟹ length u ≤ length ?z
      using Cons.IH by blast
    show ?thesis
    proof (cases length y < length ?z)
      case True
      then show ?thesis using Cons.prems z-long by auto
    next
      case le: False
      then have z-le-y: length ?z ≤ 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 ∈ set ys
          using Cons.prems by simp
        then have length x ≤ length ?z
          using z-long by simp
        also have ... ≤ 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 ⇒ 'a list ⇒ 'a list list where
  state-candidates pats w = filter (λq. suffix q w) (ac-states pats)

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

lemma state-candidates-nonempty: state-candidates pats w ≠ []
proof −
  have [] ∈ set (state-candidates pats w)
    by (simp add: state-candidates-def)
  then show ?thesis by auto
qed

lemma ac-state-in-states: ac-state pats w ∈ set (ac-states 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 ∈ set (ac-states pats)
  assumes suffix q w
  shows length q ≤ length (ac-state 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 ∈ set pats
  shows p ∈ set (ac-states pats)
  using assms by (auto simp: set-ac-states)

lemma ac-state-snoc-prefix-closed:
  assumes r @ [x] ∈ set (ac-states pats)
  shows r ∈ set (ac-states pats)
  using assms by (auto simp: set-ac-states dest: append-prefixD)
```

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

lemma *ac-state-Nil* [*simp*]: *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 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*])
using *assms* **by** (*auto simp*: *suffix-def*)

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

```
  moreover have suffix q w
    using q ac-state-suffix by simp
  ultimately have suffix r' q
    using suffix-length-suffix[OF r'-suffix] by blast
  with snoc y show ?thesis by simp
qed

theorem ac-state-snoc:
  ac-state pats (w @ [x]) = ac-step pats (ac-state pats w) x
proof −
  let ?q = ac-state pats w
  let ?left = ac-state pats (w @ [x])
  let ?right = ac-state pats (?q @ [x])
  have left-in: ?left ∈ set (ac-states pats)
    by (rule ac-state-in-states)
  have left-suffix-full: suffix ?left (w @ [x])
    by (rule ac-state-suffix)
  have left-suffix-q: suffix ?left (?q @ [x])
    using state-suffix-next-window[
      where q=?q and pats=pats and w=w and r=?left and x=x]
      left-in left-suffix-full by simp
  have right-suffix-q: suffix ?right (?q @ [x])
    by (rule ac-state-suffix)
  have left-le-right: length ?left ≤ length ?right
    using ac-state-longest[OF left-in left-suffix-q] .
  have left-suffix-right: suffix ?left ?right
    using suffix-length-suffix[OF left-suffix-q right-suffix-q left-le-right] .

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

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 = 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.prems 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* ⇒ *′a list* ⇒ *′a list list* **where**
proper-suffix-state-candidates pats q =
filter (λ*r*. *suffix r q* ∧ *r* ≠ *q*) (*ac-states pats*)

definition *ac-fail* :: *′a list list* ⇒ *′a list* ⇒ *′a list* **where**
ac-fail pats q = *longest-list* (*proper-suffix-state-candidates pats q*)

lemma *proper-suffix-state-candidates-nonempty*:
assumes *q* ≠ []
shows *proper-suffix-state-candidates pats q* ≠ []
proof −
have [] ∈ *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-fail pats q* ∈ *set* (*ac-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-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-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 *length-ac-fail-less*:
assumes $q \neq []$
shows *length* (*ac-fail pats q*) < *length q*
proof −
obtain *zs* **where** *q*: *q* = *zs* @ *ac-fail pats q*
using *ac-fail-suffix*[*OF assms*] **by** (*auto simp*: *suffix-def*)
have *zs-ne*: *zs* ≠ []
proof
assume *zs* = []
then have *ac-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-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* ∈ *set* (*ac-states pats*)
assumes *suffix r q*
assumes *r* ≠ *q*
shows *length r* ≤ *length* (*ac-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 $q \in$ *set* (*ac-states pats*)
shows *ac-state pats q* = *q*
proof −
have *q-le-state*: *length q* ≤ *length* (*ac-state pats q*)
using *ac-state-longest*[*OF assms*, *of q*] **by** *simp*
have *state-suffix-q*: *suffix* (*ac-state pats q*) *q*
by (*rule ac-state-suffix*)
have *q-suffix-state*: *suffix q* (*ac-state pats q*)
using *suffix-length-suffix*[*OF - state-suffix-q q-le-state*] **by** *simp*
show *?thesis*
using *state-suffix-q q-suffix-state* **by** (*rule suffix-order.antisym*)
qed

lemma *ac-state-singleton-if-no-state*:
assumes [*x*] ∉ *set* (*ac-states pats*)
shows *ac-state pats* [*x*] = []
proof −
have *state*: *ac-state pats* [*x*] ∈ *set* (*ac-states pats*)
by (*rule ac-state-in-states*)
have *suffix*: *suffix* (*ac-state pats* [*x*]) [*x*]
by (*rule ac-state-suffix*)
have *ac-state pats* [*x*] = [] ∨ *ac-state pats* [*x*] = [*x*]
proof −
obtain *zs* **where** [*x*] = *zs* @ *ac-state pats* [*x*]
using *suffix* **by** (*auto simp*: *suffix-def*)
then show *?thesis*
by (*cases zs*; *cases ac-state pats* [*x*]) *auto*
qed
then show *?thesis*
using *assms state* **by** *auto*
qed

lemma *state-suffix-fail-next*:
assumes *q-ne*: *q* ≠ []
assumes *qx-not-state*: *q* @ [*x*] ∉ *set* (*ac-states pats*)
assumes *r-state*: *r* ∈ *set* (*ac-states pats*)
assumes *r-suffix*: *suffix r* (*q* @ [*x*])
shows *suffix r* (*ac-fail pats q* @ [*x*])
proof (*cases r rule*: *rev-cases*)
case *Nil*
then show *?thesis* **by** *simp*
next
case (*snoc r′ y*)
from *r-suffix snoc* **have** *y*: *y* = *x* **and** *r′-suffix*: *suffix r′ q*
by *auto*
have *r′-state*: *r′* ∈ *set* (*ac-states pats*)
using *r-state snoc* **by** (*auto intro*: *ac-state-snoc-prefix-closed*)
have *r′-proper*: *r′* ≠ *q*

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

lemma ac-step-fail-eq:
  assumes q-state: q ∈ set (ac-states pats)
  assumes q-ne: q ≠ []
  assumes qx-not-state: q @ [x] ∉ set (ac-states pats)
  shows ac-step pats (ac-fail pats q) x = ac-step pats q x
proof −
  let ?fail = ac-fail pats q
  let ?left = ac-state pats (q @ [x])
  let ?right = ac-state pats (?fail @ [x])
  have left-in: ?left ∈ set (ac-states pats)
    by (rule ac-state-in-states)
  have left-suffix-q: suffix ?left (q @ [x])
    by (rule ac-state-suffix)
  have left-suffix-fail: suffix ?left (?fail @ [x])
    using state-suffix-fail-next[OF q-ne qx-not-state left-in left-suffix-q] .
  have right-suffix-fail: suffix ?right (?fail @ [x])
    by (rule ac-state-suffix)
  have left-le-right: length ?left ≤ length ?right
    using ac-state-longest[OF left-in left-suffix-fail] .
  have left-suffix-right: suffix ?left ?right
    using suffix-length-suffix[OF left-suffix-fail right-suffix-fail left-le-right] .

  have fail-suffix-q: suffix ?fail q
    using ac-fail-suffix[OF q-ne] .
  have right-in: ?right ∈ set (ac-states pats)
    by (rule ac-state-in-states)
  have right-suffix-q: suffix ?right (q @ [x])
    using suffix-snoc-mono[OF fail-suffix-q, of x] right-suffix-fail
      suffix-order.order-trans by blast
  have right-le-left: length ?right ≤ length ?left
    using ac-state-longest[OF right-in right-suffix-q] .
  have right-suffix-left: suffix ?right ?left
    using suffix-length-suffix[OF right-suffix-q ac-state-suffix right-le-left] .
  have ?left = ?right
    using left-suffix-right right-suffix-left by (rule suffix-order.antisym)
  then show ?thesis
    by (simp add: ac-step-def)
qed
```

```
function ac-fail-step :: 'a list list ⇒ 'a list ⇒ 'a ⇒ 'a list where
  ac-fail-step pats q x =
    (if q @ [x] ∈ 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 (λ(pats, q, x). length q))
    (auto simp: length-ac-fail-less)

theorem ac-fail-step-refines:
  assumes q ∈ 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] ∈ set (ac-states pats)
    | (root) q @ [x] ∉ set (ac-states pats) q = []
    | (fail) q @ [x] ∉ set (ac-states pats) q ≠ []
      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 ∈ 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.prems fail(2) fail(1)] .
      finally show ?thesis .
    qed
  qed
qed

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

assumes q = *ac-state pats w*
shows *foldl* (*ac-fail-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 *q-state*: $q \in$ *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*: *foldl* (*ac-fail-step pats*) (*ac-state pats ?w′*) *xs* =
ac-state pats (*?w′* @ *xs*)
by (*rule Cons.IH*) *simp*
have *foldl* (*ac-fail-step pats*) q (x # *xs*) =
foldl (*ac-fail-step pats*) (*ac-state pats ?w′*) *xs*
by (*simp only*: *foldl.simps step*)
also have ... = *ac-state pats* (*?w′* @ *xs*)
using *rec* **.**
also have ... = *ac-state pats* (w @ x # *xs*)
by *simp*
finally show *?case* **.**
qed

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

theorem *ac-fail-state-foldl-take*:
foldl (*ac-fail-step pats*) [] (*take i text*) = *ac-state pats* (*take i text*)
by (*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 *ac-output* :: *'a list list* ⇒ *'a list* ⇒ *'a list list* **where**
ac-output pats q = *filter* (λ*p*. *suffix p q*) *pats*

theorem *ac-output-state*:
set (*ac-output pats* (*ac-state pats w*)) =
{*p*. *p* ∈ *set pats* ∧ *suffix p w*}

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

fun *ac-search-from* ::
  'a list list ⇒ *'a list* ⇒ *nat* ⇒ *'a list* ⇒ (*'a list* × *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 (λ*p*. (*p*, *i*)) (*ac-output pats q*) @
       ac-search-from pats w' (*Suc i*) *xs*)

definition *ac-search* :: *'a list list* ⇒ *'a list* ⇒ (*'a list* × *nat*) *list* **where**
  ac-search pats text = *ac-search-from pats* [] *0 text*

fun *ac-search-states-from* ::
  'a list list ⇒ *'a list* ⇒ *nat* ⇒ *'a list* ⇒ (*'a list* × *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 (λ*p*. (*p*, *i*)) (*ac-output pats q'*) @
      ac-search-states-from pats q' (*Suc i*) *xs*)

definition *ac-search-states* :: *'a list list* ⇒ *'a list* ⇒ (*'a list* × *nat*) *list* **where**
  ac-search-states pats text = *ac-search-states-from pats* [] *0 text*

fun *ac-search-fail-from* ::
  'a list list ⇒ *'a list* ⇒ *nat* ⇒ *'a list* ⇒ (*'a list* × *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 (λ*p*. (*p*, *i*)) (*ac-output pats q'*) @
      ac-search-fail-from pats q' (*Suc i*) *xs*)

definition *ac-search-fail* :: *'a list list* ⇒ *'a list* ⇒ (*'a list* × *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 ⇒ 'a list ⇒ nat ⇒ bool where
  occurs-ending-at pat text i ⟷
    i < length text ∧ suffix pat (take (Suc i) text)

lemma in-ac-search-from-iff:
  (p, k) ∈ set (ac-search-from pats w i xs) ⟷
    (∃ j < length xs.
      k = i + j ∧
      p ∈ set pats ∧
      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 ⟷ ?rhs)
  proof
    assume ?lhs
    then have mem:
      (p, k) ∈
        set (map (λp. (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 ∈ set (ac-output pats (ac-state pats ?w')) ∧ k = i ∨
       (p, k) ∈ set (ac-search-from pats ?w' (Suc i) xs)
      using mem by auto
    then show ?rhs
    proof
      assume p ∈ set (ac-output pats (ac-state pats ?w')) ∧ k = i
      then have p: p ∈ 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 - 0]) simp
```

```
    next
      assume rec: (p, k) ∈ set (ac-search-from pats ?w' (Suc i) xs)
      then obtain j where j: j < length xs
        and k: k = Suc i + j
        and p: p ∈ set pats
        and s: suffix p (ac-state pats (?w' @ take (Suc j) xs))
        using Cons.IH[where w=?w' and i=Suc i and k=k] by auto
      have ?w' @ take (Suc j) xs = w @ take (Suc (Suc j)) (x # xs)
        by simp
      moreover have k = i + 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 < length (x # xs)
      and k: k = i + j
      and p: p ∈ set pats
      and s: suffix p (ac-state pats (w @ take (Suc j) (x # xs)))
      by auto
    show ?lhs
    proof (cases j)
      case 0
      then have p ∈ set (ac-output pats (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' < length xs
        using j by simp
      have ?w' @ take (Suc j') xs = w @ take (Suc j) (x # xs)
        using Suc by simp
      then have rec-suffix:
        suffix p (ac-state pats (?w' @ take (Suc j') xs))
        using s by simp
      have (p, k) ∈ set (ac-search-from pats ?w' (Suc i) xs)
        using Cons.IH[where w=?w' and i=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:
  set (ac-search pats text) =
    {(p, i). p ∈ set pats ∧ occurs-ending-at p text i}
proof
  show set (ac-search pats text) ⊆
```

```
    {(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.