The string search algorithm by Knuth, Morris and Pratt

Fabian Hellauer and Peter Lammich

February 23, 2021

Abstract

The Knuth-Morris-Pratt algorithm\cite{1} is often used to show that the problem of finding a string $s$ in a text $t$ can be solved deterministically in $O(|s| + |t|)$ time. We use the Isabelle Refinement Framework\cite{2} to formulate and verify the algorithm. Via refinement, we apply some optimisations and finally use the Sepref tool\cite{3} to obtain executable code in Imperative/HOL.

Contents

1 Specification 3
  1.1 Sublist-predicate with a position check . . . . . . . . . . . . . 3
    1.1.1 Definition . . . . . . . . . . . . . . . . . . . . . . . . 3
    1.1.2 Properties . . . . . . . . . . . . . . . . . . . . . . . . 3
  1.2 Sublist-check algorithms . . . . . . . . . . . . . . . . . . . . 5

2 Naive algorithm 5
  2.1 Invariants . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
  2.2 Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
  2.3 Correctness . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

3 Knuth–Morris–Pratt algorithm 7
  3.1 Preliminaries: Borders of lists . . . . . . . . . . . . . . . . . 7
    3.1.1 Properties . . . . . . . . . . . . . . . . . . . . . . . . 7
    3.1.2 Examples . . . . . . . . . . . . . . . . . . . . . . . . . 9
  3.2 Main routine . . . . . . . . . . . . . . . . . . . . . . . . . . 10
    3.2.1 Invariants . . . . . . . . . . . . . . . . . . . . . . . . 10
    3.2.2 Algorithm . . . . . . . . . . . . . . . . . . . . . . . . 10
    3.2.3 Correctness . . . . . . . . . . . . . . . . . . . . . . . 11
    3.2.4 Storing the $f$-values . . . . . . . . . . . . . . . . . . 13
  3.3 Computing $f$ . . . . . . . . . . . . . . . . . . . . . . . . . . 14
    3.3.1 Invariants . . . . . . . . . . . . . . . . . . . . . . . . 14
1 Specification

1.1 Sublist-predicate with a position check

1.1.1 Definition

One could define

\[ \text{definition sublist-at'} \colon \text{xs y i} \equiv \text{take (length xs) (drop i ys)} = \text{xs} \]

However, this doesn’t handle out-of-bound indexes uniformly:

\[
\begin{align*}
\text{value[nbe] sublist-at'} [\text{a}] [\text{a}] 5 \\
\text{value[nbe] sublist-at'} [\text{a}] [\text{a}] 5 \\
\text{value[nbe] sublist-at'} [\text{a}] [\text{a}] 5
\end{align*}
\]

Instead, we use a recursive definition:

\[
\begin{align*}
\text{fun sublist-at :: 'a list } & \Rightarrow 'a list \Rightarrow \text{nat } \Rightarrow \text{bool where} \\
\text{sublist-at (x#xs) (y#ys) 0 } & \leftarrow x = y \land \text{sublist-at xs ys 0 } \\
\text{sublist-at xs (y#ys) (Suc i) } & \leftarrow \text{sublist-at xs ys i } \\
\text{sublist-at [ ] ys 0 } & \leftarrow \text{True } \\
\text{sublist-at [ ] - } & \leftarrow \text{False }
\end{align*}
\]

In the relevant cases, both definitions agree:

\[
\begin{align*}
\text{lemma i } & \leq \text{length ys } \Rightarrow \text{sublist-at xs ys i } \leftrightarrow \text{sublist-at'} xs ys i \\
\text{unfolding sublist-at'-def by (induction xs ys i rule: sublist-at.induct) auto}
\end{align*}
\]

However, the new definition has some reasonable properties:

1.1.2 Properties

\[
\begin{align*}
\text{lemma sublist-lengths: sublist-at xs ys i } & \Rightarrow i + \text{length xs } \leq \text{length ys } \\
\text{by (induction xs ys i rule: sublist-at.induct) auto}
\end{align*}
\]

\[
\begin{align*}
\text{lemma Nil-is-sublist: sublist-at [ ] :: 'x list } & \Rightarrow i \leftarrow i \leq \text{length ys } \\
\text{by (induction [ ] :: 'x list ys i rule: sublist-at.induct) auto}
\end{align*}
\]

Furthermore, we need:

\[
\begin{align*}
\text{lemma sublist-step[intro]:} \\
\text{[i + length xs } & < \text{length ys; sublist-at xs ys i; y\![i + length xs] = x] } \Rightarrow \text{sublist-at (xs@[x]) ys i}
\end{align*}
\]
apply (induction xs ys i rule: sublist-at.induct)
apply auto
using sublist-at.elims(3) by fastforce

lemma all-positions-sublist:
[ i + length xs ≤ length ys; ∀ jj < length xs. ys!(i+jj) = xs!jj ] => sublist-at xs ys i
proof (induction xs ys i rule: rev-induct)
  case Nil
  then show ?case by (simp add: Nil-is-sublist)
next
  case (snoc x xs)
  from ⟨ i + length (xs @ [x]) ≤ length ys ⟩ have i + length xs ≤ length ys by simp
  moreover have ∀ jj < length xs. ys!(i + jj) = xs!jj
    by (simp add: nth-append snoc.prems(2))
  ultimately have sublist-at xs ys i
    using snoc.IH by simp
  then show ?case
    using snoc.prems by auto
qed

lemma sublist-all-positions:
sublist-at xs ys i =⇒ ∀ jj < length xs. ys!(i+jj) = xs!jj
by (induction xs ys i rule: sublist-at.induct) (auto simp: nth-Cons)

It also connects well to theory HOL-Library.Sublist (compare sublist-def):

lemma sublist-at-altdef:
sublist-at xs ys i ←→ (∃ ps ss. ys = ps@xs@ss ∧ i = length ps)
proof (induction xs ys i rule: sublist-at.induct)
  case (2 ss t ts i)
  show sublist-at ss (t#ts) (Suc i) ←→ (∃ xs ys. t#ts = xs@ss@ys ∧ Suc i = length xs)
    (is ?lhs ←→ ?rhs)
  proof
    assume ?lhs
    then have sublist-at ss ts i by simp
    with 2.IH obtain xs where ∃ ys. ts = xs@ss@ys ∧ i = length xs by auto
    then have ∃ ys. t#ts = (t#xs)@ss@ys ∧ Suc i = length (t#xs) by simp
    then show ?rhs by blast
  next
    assume ?rhs
    then obtain xs where ∃ ys. t#ts = xs@ss@ys ∧ length xs = Suc i
      by (blast dest: sgm)
    then have ∃ ys. ts = (tl xs)@ss@ys ∧ i = length (tl xs)
      by (auto simp add: length-Suc-conv)
    then have ∃ xs ys. ts = xs@ss@ys ∧ i = length xs by blast
    with 2.IH show ?lhs by simp
  qed
  qed auto

corollary sublist-iff-sublist-at: Sublist.sublist xs ys ←→ (∃ i. sublist-at xs ys i)
by (simp add: sublist-at-altdef Sublist.sublist-def)

1.2 Sublist-check algorithms

We use the Isabelle Refinement Framework (Theory Refine-Monadic.Refine-Monadic) to phrase the specification and the algorithm.

s for "searchword" / "searchlist", t for "text"

**definition** kmp-SPEC s t = SPEC (λ
None ⇒ ∃ i. sublist-at s t i |
Some i ⇒ sublist-at s t i ∧ (∀ ii<i. ¬sublist-at s t ii))

**lemma** is-arg-min-id: is-arg-min id P i ←→ P i ∧ (∀ ii<i. ¬P ii)
    unfolding is-arg-min-def by auto

**lemma** kmp-result: kmp-SPEC s t = RETURN (if sublist s t then Some (LEAST i. sublist-at s t i) else None)
    unfolding kmp-SPEC-def sublist-iff-sublist-at
    apply (auto intro: LeastI dest: not-less-Least split: option.splits)
    by (meson LeastI nat-neq-iff not-less-Least)

**corollary** weak-kmp-SPEC: kmp-SPEC s t ≤ SPEC (λpos. pos≠None ←→ Sublist.sublist s t)
    by (simp add: kmp-result)

**lemmas** kmp-SPEC-altdefs =
    kmp-SPEC-def[folded is-arg-min-id]
    kmp-SPEC-def[folded sublist-iff-sublist-at]
    kmp-result

2 Naive algorithm

Since KMP is a direct advancement of the naive "test-all-starting-positions" approach, we provide it here for comparison:

2.1 Invariants

**definition** I-out-na s t i ≡ λ(i,j,pos).
    (∀ ii<i. ¬sublist-at s t ii) ∧
    (case pos of None ⇒ j = 0
    | Some p ⇒ p=i ∧ sublist-at s t i)

**definition** I-in-na s t i ≡ λ(j,pos).
    case pos of None ⇒ j < length s ∧ (∀ jj<j. tl(i+jj) = s!(jj))
    | Some p ⇒ sublist-at s t i
2.2 Algorithm

The following definition is taken from Helmut Seidl’s lecture on algorithms and data structures[4] except that we

- output the identified position pos instead of just True
- use pos as break-flag to support the abort within the loops
- rewrite \( i \leq \text{length } t - \text{length } s \) in the first while-condition to \( i + \text{length } s \leq \text{length } t \) to avoid having to use \text{int} for list indexes (or the additional precondition \( \text{length } s \leq \text{length } t \))

**definition** naive-algorithm \( s \ t \equiv \text{do} \{ \)

\begin{align*}
&\text{let } i = 0; \\
&\text{let } j = 0; \\
&\text{let } pos = \text{None}; \\
&\text{let } \text{vs} = \text{WHILEIT } (l-out-na \ s \ t) \ (\lambda (i,-, pos). \ i + \text{length } s \leq \text{length } t \land pos = \text{None}) \ (\lambda (i,j,pos). \ \text{do} \{ \\
&\text{let } j = j + 1; \\
&\text{if } j = \text{length } s \text{ then RETURN } (j, \text{Some } i) \ \text{else RETURN } (j, \text{None}) \\
&\}) \ (i,pos); \\
&\text{if } pos = \text{None} \text{ then do } \{ \\
&\text{let } i = i + 1; \\
&\text{let } j = 0; \\
&\text{RETURN } (i,j, \text{None}) \\
&\} \ \text{else RETURN } (i,j, \text{Some } i) \\
&\}) \ (i,j,pos); \\
&\text{RETURN } pos \\
&\}\}
\end{align*}

2.3 Correctness

The basic lemmas on sublist-at from the previous chapter together with Refine-Monadic. Refine-Monadic’s verification condition generator / solver suffice:

**lemma** \( s \neq [] \Rightarrow \text{naive-algorithm } s \ t \leq \text{kmp-SPEC } s \ t \)

**unfolding** naive-algorithm-def kmp-SPEC-def l-out-na-def l-in-na-def

**apply** (refine-vcg

\begin{align*}
&\text{WHILEIT-rule}[\text{where } R = \text{measure } (\lambda (i,-, pos). \ \text{length } t - i + (\text{if } pos = \text{None} \ \text{then } 0 \ \text{else } 0))] \\
&\text{WHILEIT-rule}[\text{where } R = \text{measure } (\lambda (j,-, \text{::nat option}). \ \text{length } s - j)] \\
\end{align*}

**apply** (vc-solve solve: asm-rfl

**subgoal by** (metis add-Suc-right all-positions-sublist less-antisym)
subgoal using \texttt{less-Suc-eq} by \texttt{blast}
subgoal by (\texttt{metis less-SucE sublist-all-positions})
subgoal by (\texttt{auto split: option.splits} (\texttt{metis sublist-lengths add-less-cancel-right leI le-less-trans})
done

Note that the precondition cannot be removed without an extra branch: If $s = []$, the inner while-condition accesses out-of-bound memory. This will apply to KMP, too.

3 Knuth–Morris–Pratt algorithm

Just like our templates[1][4], we first verify the main routine and discuss the computation of the auxiliary values $f_s$ only in a later section.

3.1 Preliminaries: Borders of lists

definition border $xs \ ys$ $\iff$ prefix $xs \ ys$ \&\& suffix $xs \ ys$
definition strict-border $xs \ ys$ $\iff$ border $xs \ ys$ \&\& length $xs$ $<$ length $ys$
definition intrinsic-border $ls$ $\equiv$ \texttt{ARG-MAX length $b$. strict-border $b \ ls$}

3.1.1 Properties

interpretation border-order: order border strict-border 
by standard (auto simp: border-def suffix-def strict-border-def)
interpretation border-bot: order-bot Nil border strict-border 
by standard (simp add: border-def)

lemma borderE[elim]: 
fixes $xs \ ys :: \ 'a \ list$
assumes border $xs \ ys$
obtains prefix $xs \ ys$ \&\& suffix $xs \ ys$
using assms unfolding border-def by blast

lemma strict-borderE[elim]: 
fixes $xs \ ys :: \ 'a \ list$
assumes strict-border $xs \ ys$
obtains border $xs \ ys$ \&\& length $xs$ $<$ length $ys$
using assms unfolding strict-border-def by blast

lemma strict-border-simps[simp]: 
strict-border $xs []$ $\iff$ False 
strict-border $[] (x \ # \ xs)$ $\iff$ True 
by (simp-all add: strict-border-def)

lemma strict-border-prefix: strict-border $xs \ ys$ $\implies$ strict-prefix $xs \ ys$
and strict-border-suffix: strict-border $xs \ ys$ $\implies$ strict-suffix $xs \ ys$
and strict-border-imp-nonempty: strict-border $xs \ ys$ $\implies$ $ys \neq []$
and strict-border-prefix-suffix: strict-border xs ys \leftrightarrow strict-prefix xs ys \land strict-suffix xs ys
by (auto simp: border-order.order.strict-iff-order border-def)

lemma border-length-le: border xs ys \implies length xs \leq length ys
unfolding border-def by (simp add: prefix-length-le)

lemma border-length-r-less : \forall xs. strict-border xs ys \implies length xs < length ys
using strict-borderE by auto

lemma border-positions: border xs ys \implies \forall i<length xs. ys!i = ys!(length ys - length xs + i)
unfolding border-def by (metis diff-add-inverse diff-add-inverse2 length-append not-add-less1 nth-append prefixE suffixE)

lemma all-positions-drop-length-take: \[
i \leq \text{length } w; i \leq \text{length } x;
\forall j<i. x ! j = w ! (\text{length } w + j - i)\]
\implies drop (\text{length } w - i) w = take i x
by (cases i = length x) (auto intro: nth-equality1)

lemma all-positions-suffix-take: \[
i \leq \text{length } w; i \leq \text{length } x;
\forall j<i. x ! j = w ! (\text{length } w + j - i)\]
\implies suffix (take i x) w
by (metis all-positions-drop-length-take suffix-drop)

lemma suffix-butlast: suffix xs ys \implies suffix (butlast xs) (butlast ys)
unfolding suffix-def by (metis append-Nil2 butlast.simps(1) butlast-append)

lemma positions-border: \forall j<l. w!j = w!(\text{length } w - l + j) \implies border (take l w) w
by (cases l < length w) (simp-all add: border-def all-positions-suffix-take take-is-prefix)

lemma positions-strict-border: l < length w \implies \forall j<l. w!j = w!(\text{length } w - l + j) \implies strict-border (take l w) w
by (simp add: positions-border strict-border-def)

lemmas intrinsic-borderI = arg-max-natI[OF - border-length-r-less, folded intrinsic-border-def]

lemmas intrinsic-borderI' = border-bot.bot.not-eq-extremum[THEN iffD1, THEN intrinsic-borderI]

lemmas intrinsic-border-max = arg-max-nat-le[OF - border-length-r-less, folded intrinsic-border-def]

lemma nonempty-is-arg-max-ib: ys \neq [] \implies is-arg-max length (\lambda xs. strict-border xs ys) (intrinsic-border ys)
by (simp add: intrinsic-borderI' intrinsic-border-max is-arg-max-linorder)
lemma intrinsic-border-less: \( w \neq [] \Rightarrow \text{length}(\text{intrinsic-border}(w)) < \text{length}(w) \)
using intrinsic-borderI[of w] border-length-r-less intrinsic-borderI’ by blast

lemma intrinsic-border-take-less: \( j > 0 \Rightarrow w \neq [] \Rightarrow \text{length}(\text{intrinsic-border}(\text{take}(j)(w))) < \text{length}(w) \)
by (metis intrinsic-border-less length-take less-not-refl2 min-less-iff-conj take-eq-Nil)

### 3.1.2 Examples

**lemma border-example:** \( \{b. \text{border } b''\text{"aabaabaa"}\} = \{"","a","aa","aabaa","aabaabaa"\} \)
(is \( \{b. \text{border } b ?l\} = \{\text{?take0}, \text{?take1}, \text{?take2}, \text{?take5}, ?l\}\) )

**proof**
show \( \{\text{?take0}, \text{?take1}, \text{?take2}, \text{?take5}, ?l\} \subseteq \{b. \text{border } b ?l\} \)
by simp eval
have \( \neg \text{border }"aab" ?l \neg \text{border }"aab" ?l \neg \text{border }"aabaab" ?l \neg \text{border }"aabaaba" ?l \)
by eval+
moreover have \( \{b. \text{border } b ?l\} \subseteq \text{set}(\text{prefixes } ?l) \)
using border-def in-set-prefixes by blast
ultimately show \( \{b. \text{border } b ?l\} \subseteq \{\text{?take0}, \text{?take1}, \text{?take2}, \text{?take5}, ?l\} \)
by auto
qed

**corollary strict-border-example:** \( \{b. \text{strict-border } b''\text{"aabaabaa"}\} = \{"","a","aa","aabaa","aabaabaa"\} \)
(is \( ?l = ?r \) )

**proof**
have \( ?l \subseteq \{b. \text{border } b''\text{"aabaabaa"}\} \)
by auto
also have \( \ldots = \{"","a","aa","aabaa","aabaabaa"\} \)
by (fact border-example)
finally show \( ?l \subseteq ?r \) by auto
show \( ?r \subseteq ?l \) by simp eval
qed

**corollary intrinsic-border "aabaabaa" = "aabaa"**
**proof** — We later obtain a fast algorithm for that.

**proof** — We later obtain a fast algorithm for that.

**have exhaust: strict-border b"aabaabaa" \( \longleftarrow b \in \{"","a","aa","aabaa"\} \) for b**

**using strict-border-example by auto**
then have
\( \neg \text{is-arg-max } \text{length } (\lambda b. \text{strict-border } b"aabaabaa") """
\( \neg \text{is-arg-max } \text{length } (\lambda b. \text{strict-border } b"aabaabaa") "a"
\( \neg \text{is-arg-max } \text{length } (\lambda b. \text{strict-border } b"aabaabaa") "aa"
\( \text{is-arg-max } \text{length } (\lambda b. \text{strict-border } b"aabaabaa") "aabaa"

**unfolding is-arg-max-linorder by auto**
**moreover have strict-border (intrinsic-border "aabaabaa") "aabaabaa"**
using intrinsic-border' by blast
note this[unfolded exhaust]
ultimately show \( ?thesis \)
by simp (metis list.discI nonempty-is-arg-max-ib)

3.2 Main routine

The following is Seidl’s "border"-table\[4\] (values shifted by 1 so we don’t need \( int \)), or equivalently, "f" from Knuth’s, Morris’ and Pratt’s paper\[1\] (with indexes starting at 0).

fun \( j :: 'a list \Rightarrow \text{nat} \Rightarrow \text{nat} \) where
\( j s 0 = 0 \) — This increments the compare position while \( j = 0 \ |
\( j s j = \text{length} (\text{intrinsic-border} (\text{take} j s)) + 1 \)

Note that we use their "next" only implicitly.

3.2.1 Invariants

\begin{definition}
I-outer s t \equiv \lambda (i,j,pos).
(\forall ii<i. \neg\text{sublist-at} s t ii) \land
\begin{cases}
\text{case pos of None \Rightarrow (\forall jj<j. t!(i+jj) = s!j) \land j < \text{length} s} \\
\text{Some p \Rightarrow p=i \land \text{sublist-at} s t i)
\end{cases}
\end{definition}

For the inner loop, we can reuse I-in-na.

3.2.2 Algorithm

First, we use the non-evaluable function \( j \) directly:

\begin{definition}
\( \text{kmp} s t \equiv \lambda (i,j,pos)\).
\end{definition}

\begin{verbatim}
\{ ASSERT (s \neq []); let i=0; let j=0; let pos=\text{None};
(j,pos) \leftarrow \text{WHILEIT} (\text{I-outer} s t) \ (\lambda (i,j,pos). \ i + \text{length} s \leq \text{length} t \land \text{pos}=\text{None})
(\lambda(i,j,pos). \ do \{ 
  \text{ASSERT} (i + \text{length} s \leq \text{length} t); 
  (j,pos) \leftarrow \text{WHILEIT} (\text{I-in-na} s t i) \ (\lambda(j,pos). \ t!(i+j) = s!j \land \text{pos}=\text{None})
(\lambda(j,pos). \ do \{ 
    \text{let j=j+1;}
    \text{if j=\text{length} s then RETURN} (j,\text{Some} i) \ \text{else RETURN} (j,\text{None})
  \}); (j,pos);
  \text{if pos=\text{None} then do} \{ 
    \text{ASSERT} (j < \text{length} s); 
    \text{let i = i + (j - f j + 1)};
    \text{let j = max 0 (f s j + 1)}; \quad \text{— max not necessary}
\}) \ (j,pos); \}
\}
\end{verbatim}
3.2.3 Correctness

lemma \( \text{f-eq-0-iff-j-eq-0}[\text{simp}]: \) \( j \leq 0 \iff j = 0 \)
  
  by (cases \( j \)) simp-all

lemma \( j-le-f-le: j \leq \text{length } s \implies \text{f } s j \leq j \)
  
  apply (cases \( j \))
  apply simp-all
  
  by (metis Suc-leI intrinsic-border-less length-take list.size(\( j \)) min.absorb2 nat.simps(\( j \)) not-less)

lemma \( j-le-f-le': 0 < j \implies j \leq \text{length } s \implies \text{f } s j - 1 < j \)
  
  by (metis diff-less j-le-f-le le-eq-less-or-eq less-imp-diff-less less-one)

lemma \( f-le: s \neq [] \implies \text{f } s j - 1 < \text{length } s \)
  
  by (cases \( j \)) (simp-all add: intrinsic-border-take-less)

lemma reuse-matches:
  
  assumes \( j-le: j \leq \text{length } s \)
  and old-matches: \( \forall jj < j. \, t ! (i + jj) = s ! jj \)
  
  shows \( \forall jj < \text{f } s j - 1. \, t ! (i + (j - \text{f } s j + 1) + jj) = s ! jj \)
  
  proof (cases \( j > 0 \))
    
    assume \( j > 0 \)
    
    have \( \text{f-le}: \, \text{f } s j \leq j \)
      
      by (simp add: j-le-j-le-f-le)
      
      with old-matches have \( I: \forall jj < \text{f } j'. \, t ! (?i' + jj) = s ! (j - \text{f } s j + 1 + jj) \)
        
        by (metis ab-semigroup-add-class.add.commute add.assoc diff-diff-cancel less-diff-cone)
      
      have \( \text{simp}: \, \text{length } (\text{take } j s) = j \, \text{length } (\text{intrinsic-border } (\text{take } j s)) = ?j' \)
        
        by (simp add: j-le) (metis \( 0 < \text{f } j \) diff-add-inverse2 \( j \).clims nat.neq-iff)
      
      then have \( \forall jj < ?j', \, \text{take } j s ! jj = \text{take } j s ! (j - (\text{f } s j - 1) + jj) \)
        
        by (metis intrinsic-borderI' \( 0 < j \) border-positions length-greater-0-conv strict-border-def)
      
      then have \( \forall jj < ?j', \, \text{take } j s ! jj = \text{take } j s ! (j - \text{f } s j + 1 + jj) \)
        
        by (simp add: \( f-le \))
      
      then have \( 2: \forall jj < ?j'. \, s ! (j - \text{f } s j + 1 + jj) = s ! jj \)
        
        using \( f-le \) by simp
      
      from 1 2 show \( \text{thesis by simp} \)
    
    qed simp

theorem shift-safe:
assumes
\( \forall \, ii < i. \neg \text{sublist-at } s \, t \, ii \)
\( t!(i+j) \neq \$j \) and
\[ \text{[simp]}: j < \text{length } s \quad \text{and} \quad \text{matches}: \forall jj < j. \, t!(i+jj) = s!jj \]
defines
\( \text{assignment: } i' \equiv i + (j - \lfloor s \, j \rfloor + 1) \)
shows
\( \forall \, ii < i'. \neg \text{sublist-at } s \, t \, ii \)
proof (standard, standard)
fix \( ii \)
assume \( ii < i' \)
then consider — The position falls into one of three categories:
(\( \text{old} \) \( ii < i \))
(\( \text{current} \) \( ii = i \))
(\( \text{skipped} \) \( ii > i \))
by linarith
then show \( \neg \text{sublist-at } s \, t \, ii \)
proof cases
  case \( \text{old} \) — Old position, use invariant.
    with \( \forall \, ii < i. \neg \text{sublist-at } s \, t \, ii \) show ?thesis by simp
next
case \( \text{current} \) — The mismatch occurred while testing this alignment.
  with \( t!(i+j) \neq s!j \) show ?thesis
    using sublist-all-positions[\( \text{of } s \, t \, i \)] by auto
next
case \( \text{skipped} \) — The skipped positions.
then have \( \theta < j \)
  using \( \theta < i' \) assignment by linarith
then have \( \text{less-fj} \) \( j + i - ii < j \) and \( \text{le-s: } j + i - ii \leq \text{length } s \)
  using \( \theta < i' \) assms(\( j \)) skipped by linarith+
note \( \text{f-le} \) \( j \leq j \leq f \) \( \theta \)
  [OF assms(\( j \)) [THEN less-imp-le]]
have \( \theta < j \)
  using \( \theta < j \) \( \lfloor \text{eq-0} \rfloor \text{-eq-0} \) \( \text{eq-0} \) \( \text{neq0-conv} \) by blast
then have \( j + i - ii > j \)
  using \( \theta < i' \) assignment \( \lfloor \text{le} \) by linarith
then have \( \text{contradiction-goal: } j + i - ii > \text{length } \) (\( \text{intrinsic-border: } \text{take } j \))
  by (metis \( \text{f.elims: } \theta < j \) \( \text{add-diff-cancel-right' not-gr-zero} \))
show ?thesis
proof
assume \( \text{sublist-at } s \, t \, ii \)
note sublist-all-positions[OF this]
with \( \text{le-s} \) have \( \alpha: \forall jj < j + i - ii. \, t!(ii+jj) = s!jj \)
  by simp
have \( ffl: \neg \, ii < i \)
  by (metis \( \text{not-less-iff-gr-or-eq} \) \( \text{skipped} \))
then have \( i + (ii - i + jj) = ii + jj \) for \( jj \)
  by (metis \( \text{add.assoc add-diff-inverse-nat} \))
then have \( \neg \, jj < j + i - ii \lor t! \) (\( ii + jj \)) = \( s! (ii - i + jj) \)
  if \( ii - i + jj \)

12
< j for jj
  using that ff1 by (metis matches)
then have ¬ jj < j + i − ii ∨ t! (ii + jj) = s! (ii − i + jj) for jj
  using ff1 by auto
with matches have ∀ jj < j+i−ii. t!(ii+jj) = s!(ii−i+jj) by metis
then have ∀ jj < j+i−ii. s!jj = s!(ii−i+jj)
  using a by auto
then have ∀ jj < j+i−ii. t!(ii+jj) = s!(ii−i+jj)
  using ⟨i<ii⟩ by auto
with positions-strict-border[of j+i−ii take j s, simplified]
have strict-border (take (j+i−ii) s) (take j s),
note intrinsic-border-max[OF this]
also note contradiction-goal
also have j+i−ii ≤ length s by (fact le-s)
ultimately
show False by simp
qed
qed

lemma kmp-correct: s ≠ []
  ⇒ kmp s t ≤ kmp-SPEC s t
unfolding kmp-def kmp-SPEC-def I-outer-def I-in-na-def
apply (refine-vcg
   WHILEIT-rule[where R=measure (λ(i,-,pos). length t − i + (if pos = None then 1 else 0))]
   WHILEIT-rule[where R=measure (λ(j,-::nat option). length s − j)]
   )
  apply (vc-solve solve: asm-rl)
subgoal by (metis add-Suc-right all-positions-sublist less-antisym)
subgoal using less-antisym by blast
subgoal for i jout j using shift-safe[of i s t j] by fastforce
subgoal for i jout j using reuse-matches[of j s t i] j-le by simp
subgoal by (auto split: option.splits) (metis sublist-lengths add-less-cancel-right leI le-less-trans)
done

3.2.4 Storing the f-values

We refine the algorithm to compute the f-values only once at the start:

definition compute-fs-SPEC :: 'a list ⇒ nat list nres where
  compute-fs-SPEC s ≡ SPEC (λs. length s = length s + 1 ∧ (∀ j≤length s. fs!j
  = f s j))

definition kmp1 s t ≡ do {
  ASSERT (s ≠ []);
  let i=0;
  let j=0;
  let pos= None;
  
13
\( f \leftarrow \text{compute-\( f \)-SPEC} \; (\text{butlast} \; s); \; - \; \text{At the last char, we abort instead.} \\
(\gamma, \text{pos}) \leftarrow \text{WHILEIT} \; (I-outer \; s \; t) \; (\lambda (i, j, \text{pos}). \; i + \text{length} \; s \leq \text{length} \; t \land \text{pos} = \text{None}) \\
(\lambda (i, j, \text{pos}). \; \text{do} \{ \\
\quad \text{ASSERT} \; (i + \text{length} \; s \leq \text{length} \; t); \\
\quad (j, \text{pos}) \leftarrow \text{WHILEIT} \; (I-in-na \; s \; t \; i) \; (\lambda (j, \text{pos}). \; t!(i+j) = s!j \land \text{pos} = \text{None}) \\
\} \} (i, j, \text{pos}); \\
\text{if} \; \text{pos} = \text{None} \; \text{then} \; \text{do} \{ \\
\quad \text{ASSERT} \; (j < \text{length} \; f \; s); \\
\quad \text{let} \; i = i + (j - f \; s!j + 1); \\
\quad \text{let} \; j = \text{max} \; 0 \; (f \; s!j - 1); \; - \; \text{max not necessary} \\
\; \} \else \text{return} \; (i, j, \text{Some} \; i) \\
\} \} (i, j, \text{pos}); \\
\text{return} \; \text{pos} \\
\} \\
\text{lemma} \; f\!-\!\text{butlast}[\text{simp}]: \; j < \text{length} \; s \implies f\! (\text{butlast} \; s) \; j = f \; s \; j \\
\text{by} \; (\text{cases} \; j) \; (\text{simp-all} \; \text{add}: \; \text{take-butlast}) \\
\text{lemma} \; kmp1\!-\!\text{refine}: \; kmp1 \; s \; t \leq \; kmp \; s \; t \\
\text{apply} \; (\text{rule refine-IdD}) \\
\text{unfolding} \; kmp1\!-\!\text{def} \; kmp\!-\!\text{def} \; \text{Let-def compute-\( f \)-SPEC-def nres-monad-laws} \\
\text{apply} \; (\text{intro} \; \text{ASSERT-refine-right} \; \text{ASSERT-refine-left}) \\
\text{apply} \; \text{simp} \\
\text{apply} \; (\text{rule Refine-Basic.intro-spec-refine}) \\
\text{apply} \; \text{refine-rcg} \\
\quad \text{apply} \; \text{refine-dref-type} \\
\quad \text{apply} \; \text{vc-solve} \\
\text{done} \\
\text{Next, an algorithm that satisfies compute-\( f \)-SPEC:} \\
\text{3.3 Computing} \; f \text{3.3.1 Invariants} \\
\text{definition} \; I\!-\!\text{out-cb} \; s \equiv \lambda (f \; s, i, j). \\
\; \text{length} \; s + 1 = \text{length} \; f \; s \land \\
\; (\forall \; jj < j. \; f \! s!jj = f \! s \; jj) \land \\
\; f \! s!(j-1) = i \land \\
\; 0 < j \\
\text{definition} \; I\!-\!\text{in-cb} \; s \; j \equiv \lambda i. \\
\; \text{if} \; j = 1 \; \text{then} \; i = 0 \; - \; \text{first iteration} \\
\; \text{else} \\
\; \quad \text{strict-border} \; (\text{take} \; (i-1) \; s) \; (\text{take} \; (j-1) \; s) \land \\
\; \quad f \; s \; j \leq i + 1
3.3.2 Algorithm

Again, we follow Seidl[4], p.582. Apart from the +1-shift, we make another modification: Instead of directly setting $f_s ! 1$, we let the first loop-iteration (if there is one) do that for us. This allows us to remove the precondition $s \neq []$, as the index bounds are respected even in that corner case.

**Definition** compute-$f_s :: \text{'a list } \Rightarrow \text{ nat list nres where}

\[
\begin{align*}
\text{compute-}f_s \text{ do } & \{ \\
& \text{let } f_s = \text{replicate (length } s + 1) 0; \text{ -- only the first 0 is needed} \\
& \text{let } i=0; \\
& \text{let } j=1; \\
& (f_s,-,-) \leftarrow \text{WHILEIT (I-out-cb s) (λ(f_s,-,j). } j < \text{ length } f_s) \text{ (λ(f_s,i,j). do } \{ \\
& \hspace{1em} \text{ASSERT (i-1 < length } f_s); \\
& \hspace{1em} \text{let } i=\text{sl}(i-1); \\
& \hspace{1em} \text{RETURN i} \\
& \}) \} i; \\
& \text{let } i=i+1; \\
& \text{ASSERT (j < length } f_s); \\
& \text{let } f_s[f_s[j]=i]; \\
& \text{let } j=j+1; \\
& \text{RETURN (f_s,i,j) } \} (f_s,i,j); \\
\text{RETURN } f_s \\
\}
\end{align*}
\]

3.3.3 Correctness

**Lemma** take-length-ib[simp];

\[
\text{assumes } 0 < j \leq \text{ length } s \\
\text{shows } \text{take (length (intrinsic-border (take } j s))) s = \text{ intrinsic-border (take } j s) \\
\]

**Proof** –

\[
\begin{align*}
\text{from assms have prefix (intrinsic-border (take } j s)) (\text{take } j s) \\
\text{by (metis intrinsic-borderI' border-def list.size(3) neq0-conv not-less strict-border-def take-eq-Nil)} \\
& \text{also have prefix (take } j s) s \\
& \text{by (simp add: } j \leq \text{ length } s \text{ take-is-prefix)} \\
& \text{finally show } \text{thesis} \\
& \text{by (metis append-eq-cone-conj prefixE)}
\end{align*}
\]

qed

**Lemma** ib-singleton[simp]: intrinsic-border [z] = []

\[
\text{by (metis intrinsic-border-less length-Cons length-greater-0-conv less-Suc0 list.size(3))}
\]

**Lemma** border-butlast; border xs ys = border (butlast xs) (butlast ys)

\[
\text{apply (auto simp: border-def)} \\
\text{apply (metis butlast-append prefixE prefix-order.eq_refl prefix-root prefixeq-butchoplast)} \\
\text{apply (metis Sublist.suffix-eq append.right-neutral butlast.simps(1) butlast-append)}
\]

15
done

corollary strict-border-butlast: \( xS \neq [] \implies \text{strict-border} \; xS \; yS \implies \text{strict-border} \; (\text{butlast} \; xS) \; (\text{butlast} \; yS) \)

  unfolding strict-border-def by (simp add: border-butlast less-diff-conv)

lemma border-take-lengths: \( i \leq \text{length} \; s \implies \text{border} \; (\text{take} \; i \; s) \; (\text{take} \; j \; s) \implies i \leq j \)

  using border-length-le by fastforce

lemma border-step: \( \text{border} \; xS \; yS \iff \text{border} \; (xS@[\text{length} \; xS]) \; (yS@[\text{length} \; xS]) \)

  apply (auto simp: border-def suffix-def)

  using append-one-prefix prefixE apply fastforce

  using append-prefixD apply blast

  done

corollary strict-border-step: \( \text{strict-border} \; xS \; yS \iff \text{strict-border} \; (xS@[\text{length} \; xS]) \; (yS@[\text{length} \; xS]) \)

  unfolding strict-border-def using border-step by auto

lemma ib-butlast: \( \text{length} \; w \geq 2 \implies \text{length} \; (\text{intrinsic-border} \; w) \leq \text{length} \; (\text{intrinsic-border} \; (\text{butlast} \; w)) + 1 \)

  proof –

  assume \( \text{length} \; w \geq 2 \)

  then have \( w \neq [] \) by auto

  then have \( \text{strict-border} \; (\text{intrinsic-border} \; w) \)

  by (fact intrinsic-borderI)

  with \( \langle 2 \leq \text{length} \; w \rangle \) have \( \text{strict-border} \; (\text{butlast} \; (\text{intrinsic-border} \; w)) \; (\text{butlast} \; w) \)

  by (metis One-nat-def border-bot.bot.not-eq-extremum butlast.simps(1) len-greater-imp-nonempty

  length-butlast lessl less-le-trans numerals(2) strict-border-butlast zero-less-diff)

  then have \( \text{length} \; (\text{butlast} \; (\text{intrinsic-border} \; w)) \leq \text{length} \; (\text{intrinsic-border} \; (\text{butlast} \; w)) \)

  using intrinsic-border-max by blast

  then show \( \text{thesis} \)

  by simp

qed

corollary \( \text{f-Suc}: \text{Suc} \; i \leq \text{length} \; w \implies \text{f} \; w \; (\text{Suc} \; i) \leq \text{f} \; w \; i + 1 \)

  apply (cases \( i \))

  apply (simp-all add: take-Suc)

  by (metis One-nat-def Suc-ep-plus1 Suc-to-right butlast-take diff-is-0-ep ib-butlast

  length-take min.absorb2 nat.simps(3) not-less-ep-ep numerals(2))

lemma \( \text{f-step-bound}: \)

  assumes \( j \leq \text{length} \; w \)

  shows \( \text{f} \; w \; j \leq \text{f} \; w \; (j - 1) + 1 \)

  using assms[THEN f-le-f-le] f-Suc assms

  by (metis One-nat-def Suc-pred le-Suc1 not-gr-zero trans-le-add2)

lemma border-take-f: \( \text{border} \; (\text{take} \; (\text{f} \; s \; i - 1) \; s) \; (\text{take} \; i \; s) \)

16
apply (cases i, simp-all)
by (metis intrinsic-borderI' border-order.eq_iff border-order.less_imp_le border-positions
| nat.simps(3) nat.le_linear positions-border take-all take-eq.Nil take-length-ib zero_less-Suc)

**corollary** f-strict-borderI: y = f s (i-1) \implies strict-border (take (i-1) s) (take
(j-1) s) \implies strict-border (take (y-1) s) (take (j-1) s)

**using** border-order.less_le_not_le border-order.order.trans border-take-f by blast

**corollary** strict-border-take-f: 0 < i \implies i \leq length s \implies strict-border (take (f s i
- 1) s) (take i s)

by (meson border-order.less_le_not_le border-take-f border-take-lengths j-le-f-le' leD)

**lemma** f-is-max: j \leq length s \implies strict-border b (take j s) \implies f s j \geq length b + 1

by (metis f.elims add-le-cancel-right add-less_same-cancel2 border-length-r-less
intrinsic-border-max length-take min-absorb2 not_add_less2)

**theorem** skipping-ok:
assumes f-bounds[simp]: 1 < j \leq length s
and mismatch: s'(i-1) \neq s'(j-1)
and greater-checked: f s j \leq i + 1
and strict-border (take (i-1) s) (take (j-1) s)
shows f s j \leq f s (i-1) + 1

**proof** (rule contr)
assume \neg f s j \leq f s (i-1) + 1
then have i-bounds: 0 < i i \leq length s
using greater-checked assms(5) take-nil by fastforce+
then have i-less-j: i < j
using assms(5) border-length-r-less nz_le_conv_less by auto
from \neg f s j \leq f s (i-1) + 1) greater-checked consider
(tested) f s j = i + 1 — This contradicts s ! (i-1) \neq s !(j-1) |
(skipped) f s (i-1) + 1 < f s j \leq s i
— This contradicts [i - 1 \leq length s; strict-border ?b (take (i - 1) s)] \implies
length ?b + 1 \leq f s (i - 1)

by linarith
then show False

**proof** cases

**case** tested
then have \neg f s j - 1 = i by simp
moreover note border-positions[of s j, unfolded this]
ultimately have take j s ! (i-1) = s!(j-1) using i-bounds i-less-j by simp
with \neg i < j have s'(i-1) = s'(j-1)

by (simp add: less_imp_diff_less)
with mismatch show False..

next
**case** skipped
let ?border = take (i-1) s
— This border of take (j - 1) s could not be extended to a border of take j s
due to the mismatch.

let \( ?\text{impossible} = \text{take} (j \ s \ j - 2) \ s \)
— A strict border longer than \( \text{intrinsic-border} \ (\text{take} \ (i - 1) \ s) \), a contradiction.

have length (\( \text{take} \ j \ s \)) = \( j \)
    by simp
have \( j \ s \ j - 2 < i - 1 \)
    using skipped by linarith
then have less-s: \( j \ s \ j - 2 < \text{length} \ s \ i - 1 < \text{length} \ s \)
    using \( i < j \); j-bounds(2) by linarith+
then have strict: length \( ?\text{impossible} < \text{length} \ ?\text{border} \)
    using \( i < j \); linarith

moreover {
    have prefix \( ?\text{impossible} \ (\text{take} \ j \ s) \)
        using prefix-length-prefix take-is-prefix
    by (metis (no-types, lifting) \( \text{length} \ (\text{take} \ j \ s) = j \) j-bounds(2) diff-le-self
                  j-le-\( j \)-le length-take less-s(1) min-simps(2) order-trans)
    moreover have prefix \( ?\text{border} \ (\text{take} \ j \ s) \)
        by (metis (no-types, lifting) \( \text{length} \ (\text{take} \ j \ s) = j \) diff-le-self i-less-j le-trans
                  length-take less-or-eq-imp-le less-s(2) min-simps(2) prefix-length-prefix take-is-prefix)
    ultimately have prefix \( ?\text{impossible} \ ?\text{border} \)
        using strict less-imp-le-nat prefix-length-prefix
    by blast
}

} moreover {
    have suffix (\( \text{take} (j \ s \ j - 1) \ s \)) (\( \text{take} \ j \ s \)) using border-take-f
        by (auto simp: border-def)
    note suffix-butlast[\( \text{OF this} \)]
    then have suffix \( ?\text{impossible} \ (\text{take} (j - 1) \ s) \)
        by (metis One-nat-def j-bounds(2) butlast-take diff-diff-left \( j \)-le \( j \)-greater \( j \)-imp nonempty
                  length-take less-or-eq-imp-le less-s(2) one-add-one)
    then have suffix \( ?\text{impossible} \ (\text{take} (j - 1) \ s) \) suffix \( ?\text{border} \ (\text{take} (j - 1) \ s) \)
        using assms(5) by auto
    from suffix-length-suffix[\( \text{OF this strict[THEN less-imp-le]} \)]
    have suffix \( ?\text{impossible} \ ?\text{border} \).
}

} ultimately have strict-border \( ?\text{impossible} \ ?\text{border} \)
    unfolding strict-border-def[unfolded border-def] by blast
note \( j\text{-is-max}[\text{OF i-1 s, OF - this}] \)
then have length (\( \text{take} (j \ s \ j - 2) \ s \)) \( + 1 \leq \text{length} \ s \ (i - 1) \)
    using less-imp-le-nat less-s(2) by blast
then have \( \text{simp add: less-s(1)} \)
    by simp
then have \( j \ s \ j - 1 \leq \text{length} \ s \ (i - 1) + 1 \)
    using le-diff-conv by blast
with skipped(1) show False
    by linarith
qed

lemma extend-border:
    assumes \( j \leq \text{length} \ s \)
assumes $s(i-1) = s(j-1)$
assumes strict-border $(\text{take} \ (i-1) \ s) \ (\text{take} \ (j-1) \ s)$
assumes $f \ s j \leq i+1$
shows $f \ s j = i+1$

proof –
from assm(3) have pos-in-range: $i - 1 < \text{length} \ s \ \text{length} \ (\text{take} \ (i-1) \ s) = i - 1$
using border-length-r-less min-less-iff-conj by auto
with strict-border-step[THEN iffD1, OF assms(3)] have strict-border $(\text{take} \ (i-1) \ s \ \oplus \ [s(i-1)]) \ (\text{take} \ (j-1) \ s \ \oplus \ [s(i-1)])$
  by (metis assms(3) border-length-r-less length-take min-less-iff-conj nth-take)
with pos-in-range have strict-border $(\text{take} \ i \ s) \ (\text{take} \ (j-1) \ s \ \oplus \ [s(i-1)])$
  by (simp only: ⟨$s(i-1) = s(j-1)$⟩)
then have strict-border $(\text{take} \ i \ s) \ (\text{take} \ (j-1) \ s \ \oplus \ [s(j-1)])$
  by (simp only: $s(i-1) = s(j-1)$)
then have strict-border $(\text{take} \ i \ s) \ (\text{take} \ j \ s \ \oplus \ [s(j-1)])$
  using Suc-leI by fastforce
with $f$ is max[OF assms(1) this] have $f \ s j \geq i + 1$
using Suc-le1 by fastforce

lemma compute-ť-s-correct: compute-ť $s \leq$ compute-ť-SPEC $s$
unfolding compute-ť-SPEC-def compute-ť-def I-out-cb-def I-in-cb-def
apply (simp, refine-vcg)
WHILEIT-rule[where $R=\text{measure} \ (\lambda (f,s,i,j) \ . \ \text{length} \ s + 1 - j)]$
WHILEIT-rule[where $R=\text{measure} \ \text{id}$] — $i$ decreases with every iteration.
apply (vc-solve, fold One-nat-def)
subgoal for $b \ j$ by (rule strict-border-take-ť, auto)
subgoal by (metis Suc-eq-plus1 ĭ-step-bound less-Suc-eq-le)
subgoal by fastforce
subgoal by (metis (no-types, lifting) One-nat-def Suc-lessD Suc-pred border-length-r-less ĭ-strict-borderI length-take less-Suc-eq less-Suc-eq-le min.absorb2)
subgoal for $b \ j \ i$
  by (metis (no-types, lifting) One-nat-def Suc-diff-1 Suc-eq-plus1 Suc-le1 border-take-lengths less-Suc-eq-le less-antisym skipping-ok strict-border-def)
subgoal by (metis Suc-diff-1 border-take-lengths ĭ-le-ť-le less-Suc-eq-le strict-border-def)
subgoal for $b \ j \ i \ j$
  by (metis Suc-eq-plus1 Suc-eq-plus1-left add.right-neutral extend-border ĭ-eq-0-iff ĭ-eq-0 ĭ-le-ť-le le-zero-eq less-Suc-eq less-Suc-eq-le nth-list-update-eq nth-list-update-neq)
subgoal by linarith
done
3.3.4 Index shift

To avoid inefficiencies, we refine compute-fs to take s instead of butlast s (it still only uses butlast s).

definition compute-butlast-fs :: 'a list ⇒ nat list nres where
compute-butlast-fs s = do 
  let fs=replicate (length s) 0;
  let i=0;
  let j=1;
  (fs,-,-) ← WHILEIT (I-out-cb (butlast s)) (λ(b,i,j). j < length b) (λ(fs,i,j). do 
    ASSERT (j < length fs);
    i ← WHILEIT (I-in-cb (butlast s) j) (λi. i>0 ∧ s!(i−1) ≠ s!(j−1)) (λi. do 
      ASSERT (i−1 < length fs);
      let i=s!(i−1);
      RETURN i)
  ) i;
  let i=i+1;
  ASSERT (j < length fs);
  let fs=fs[j:=i];
  let j=j+1;
  RETURN (fs,i,j);
 ) (fs,i,j);

RETURN fs

lemma compute-fs-inner-bounds:
assumes I-out-cb s (fs,ix,j)
assumes j < length fs
assumes I-in-cb s j i
shows i−1 < length s j−1 < length s
using assms
  by (auto simp: I-out-cb-def I-in-cb-def split: if-splits)

lemma compute-butlast-fs-refine[refine]:
assumes (s,s') ∈ br butlast ((≠) [])
shows compute-butlast-fs s ≤ ⇓ Id (compute-fs-SPEC s')
proof –
have compute-butlast-fs s ≤ ⇓ Id (compute-fs s')
  unfolding compute-butlast-fs-def compute-fs-def
  apply (refine-rcg)
  apply (refine-dref-type)
  using assms apply (vc-solve simp: in-br-conv)
  apply (metis Suc-pred length-greater-0-conv replicate-Suc)
  by (metis One-nat-def compute-fs-inner-bounds nth-butlast)
also note compute-fs-correct
finally show ?thesis by simp
qed
3.4 Conflation

We replace compute-\(fs\)-SPEC with compute-butlast-\(fs\).

\textbf{definition} \(kmp2 s t \equiv \{\)
\begin{verbatim}
ASSERT \((s \neq [])\);
let i=0;
let j=0;
let pos=None;
\(fs \leftarrow \) compute-butlast-\(fs\) \(s\);
\((-,-,pos) \leftarrow \text{WHILEIT (I-outer } t \) \(\lambda (i,j,pos). \not \in (i+length \ s \leq length \ t \land pos=\) None\) \(\lambda (i,j,pos)\). \) do \{
  \(assert \ (i + \text{length } s \leq \text{length } t \land pos=\) None\);
  \(j,pos) \leftarrow \text{WHILEIT (I-in-na } s \ t \ i \) \(\lambda (j,pos). \not \in \) \(t!(i+j) = s!j \land pos=\) None\) \(\lambda (j,pos)\). \) do \{
    \(let j=j+1\);
    if \(j=\) length \(s\) then RETURN \((j,\text{Some } i)\) else RETURN \((j,\text{None})\)
  \} \(j,pos)\);
if pos=None then do \{
  \(assert \ (j < \text{length } fs)\);
  let i = i + \((j - fs!j + 1)\);
  let j = \max \(0 \) \(fs!j - 1\)\); \(\not \in \) max not necessary
  RETURN \((i,j,\text{None})\)
} else RETURN \((i,j,\text{Some } i)\)
\} \(i,j,pos)\);

RETURN pos
\}
\end{verbatim}

Using compute-butlast-\(fs\)-refine (it has attribute refine), the proof is trivial:

\textbf{lemma} \(kmp2\)-refine: \(kmp2 s t \leq kmp1 s t\)

\textbf{apply} (rule refine-IdD)
\textbf{unfolding} \(kmp2\)-def \(kmp1\)-def
\textbf{apply} refine-reg
  \textbf{apply} refine-dref-type
  \textbf{apply} (vc-solve simp: in-br-cone)
\textbf{done}

\textbf{lemma} \(kmp2\)-correct: \(s \neq \) []
\(\Rightarrow \) \(kmp2 s t \leq kmp\)-SPEC \(s t\)

\textbf{proof} 
\assumption \(s \neq \) []
\have \(kmp2 s t \leq kmp1 s t\) \(by\) (fact \(kmp2\)-refine)
\also \have \(\not \in \) \(kmp s t\) \(by\) (fact \(kmp1\)-refine)
\also \have \(\not \in \) \(kmp\)-SPEC \(s t\) \(by\) (fact \(kmp\)-correct\(\langle s \neq \) []\)\)
\finally \show \(\)\thesis.
\textbf{qed}

For convenience, we also remove the precondition:

\textbf{definition} \(kmp3 s t \equiv \{\)
\begin{verbatim}
\end{verbatim}

21
if s=[] then RETURN (Some 0) else kmp2 s t
}

lemma kmp3-correct: kmp3 s t ≤ kmp-SPEC s t
  unfolding kmp3-def by (simp add: kmp2-correct) (simp add: kmp-SPEC-def)

4 Refinement to Imperative/HOL

lemma eq-id-param: ((=), (=)) ∈ Id → Id → Id by simp

lemmas in-bounds-aux = compute-fs-inner-bounds[of butlast s for s, simplified]

sepref-definition compute-butlast-fs-impl is compute-butlast-fs :: (arl-assn id-assn)k
  →a array-assn nat-assn
  unfolding compute-butlast-fs-def
  supply in-bounds-aux[dest]
  supply eq-id-param[where 'a='a, sepref-import-param]
  apply (rewrite array-fold-custom-replicate)
  by sepref

declare compute-butlast-fs-impl.refine[sepref-fr-rules]

sepref-register compute-fs

lemma kmp-inner-in-bound:
  assumes i + length s ≤ length t
  assumes I-in-na s t i (j, None)
  shows i + j < length t j < length s
  using assms
  by (auto simp: I-in-na-def)

sepref-definition kmp-impl is uncurl kmp3 :: (arl-assn id-assn)k *a (arl-assn id-assn)k →a option-assn nat-assn
  unfolding kmp3-def kmp2-def
  apply (simp only: max-0L) — Avoid the unneeded max
  apply (rewrite in WHILEIT (I-in-na - - -) ⊢ conj-commute)
  apply (rewrite in WHILEIT (I-in-na - - -) ⊢ short-circuit-conv)
  supply kmp-inner-in-bound[dest]
  supply option.splits[split]
  supply eq-id-param[where 'a='a, sepref-import-param]
  by sepref

export-code kmp-impl in SML-imp module-name KMP

lemma kmp3-correct':
  (uncurl kmp3, uncurl kmp-SPEC) ∈ Id ×, Id → f (Id)nres-rel
  apply (intro frefI nres-relI; clarsimp)
  apply (fact kmp3-correct)
lemmas kmp-impl-correct' = kmp-impl.refine[FCOMP kmp3-correct']

4.1 Overall Correctness Theorem

The following theorem relates the final Imperative HOL algorithm to its specification, using, beyond basic HOL concepts

- Hoare triples for Imperative/HOL, provided by the Separation Logic Framework for Imperative/HOL (Theory Separation-Logic-Imperative-HOL.Sep-Main);
- The assertion arl-assn to specify array-lists, which we use to represent the input strings of the algorithm;
- The sublist-at function that we defined in section 1.

\[
\text{theorem} \quad kmp-impl-correct:
\begin{align*}
\langle \text{arl-assn id-assn } & s \textit{ si} * \text{arl-assn id-assn } t \textit{ ti} > \\
\text{ kmp-impl si ti} \\
\langle \lambda r. \text{arl-assn id-assn } s \textit{ si} * \text{arl-assn id-assn } t \textit{ ti} * \uparrow( \\
\text{ case } r \text{ of None } \Rightarrow \exists i. \text{sublist-at } s \textit{ t i} \\
\text{ case Some } i \Rightarrow \text{sublist-at } s \textit{ t i} \land (\forall ii < i. \neg \text{sublist-at } s \textit{ t ii}) \\
\rangle > t
\end{align*}
\]

by \((sep-auto)\)

simp: pure-def kmp-SPEC-def
split: option.split
heap: kmp-impl-correct'[THEN hrefD, THEN hn-refineD, of \((s,t)\) \((si,ti)\), simplified]]

\[
\text{definition} \quad kmp-string-impl \equiv \text{kmp-impl} :: (\text{char array } \times \text{ nat}) \Rightarrow \text{-}
\]

5 Tests of Generated ML-Code

\[
\text{ML-val} \quad \text{fun} \quad \text{str2arl } s = \text{(Array.fromList \(\{\text{code String.explode}\} \ s), \(\{\text{code nat-of-integer}\} \ (\text{String.size } s)\)}
\]

\[
\text{fun kmp } s \ t = \text{map-option \(\{\text{code integer-of-nat}\} \ (\{\text{code kmp-string-impl} \ (\text{str2arl } s) \ (\text{str2arl } t) \})\)}
\]

\[
\text{val test1} = \text{kmp anas bananas}
\text{val test2} = \text{kmp bananas}
\text{val test3} = \text{kmp hide-fact \(\text{File.read \(\{\text{file \('\sim\)/src/HOL/Main.thy}\}\})\)}
\text{val test4} = \text{kmp sorry \(\text{File.read \(\{\text{file \('\sim\)/src/HOL/HOL.thy}\}\})\)}
\]

end
References


