

The string search algorithm by Knuth, Morris and Pratt

Fabian Hellauer and Peter Lammich

December 14, 2021

Abstract

The Knuth-Morris-Pratt algorithm[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[2] to formulate and verify the algorithm. Via refinement, we apply some optimisations and finally use the *Seppref* tool[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

3.3.2	Algorithm	15
3.3.3	Correctness	15
3.3.4	Index shift	20
3.4	Conflation	21
4	Refinement to Imperative/HOL	22
4.1	Overall Correctness Theorem	23
5	Tests of Generated ML-Code	23

```

theory KMP
  imports Refine-Imperative-HOL.IICF
           HOL-Library.Sublist
begin

declare len-greater-imp-nonempty[simp del] min-absorb2[simp]
no-notation Ref.update (- := - 62)

```

1 Specification

1.1 Sublist-predicate with a position check

1.1.1 Definition

One could define

definition *sublist-at'* $xs\ ys\ i \equiv take\ (length\ xs)\ (drop\ i\ ys) = xs$

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

```

value[nbe] sublist-at' [] [a] 5
value[nbe] sublist-at' [a] [a] 5
value[nbe] sublist-at' [] [] 5

```

Instead, we use a recursive definition:

```

fun sublist-at :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  nat  $\Rightarrow$  bool where
  sublist-at (x#xs) (y#ys) 0  $\longleftrightarrow$  x=y  $\wedge$  sublist-at xs ys 0 |
  sublist-at xs (y#ys) (Suc i)  $\longleftrightarrow$  sublist-at xs ys i |
  sublist-at [] ys 0  $\longleftrightarrow$  True |
  sublist-at - [] -  $\longleftrightarrow$  False

```

In the relevant cases, both definitions agree:

lemma $i \leq length\ ys \implies$ *sublist-at* xs ys i \longleftrightarrow *sublist-at'* xs ys i
unfolding *sublist-at'-def*
by (induction xs ys i rule: *sublist-at.induct*) auto

However, the new definition has some reasonable properties:

1.1.2 Properties

lemma *sublist-lengths*: *sublist-at* xs ys i \implies $i + length\ xs \leq length\ ys$
by (induction xs ys i rule: *sublist-at.induct*) auto

lemma *Nil-is-sublist*: *sublist-at* ([] :: 'x list) ys i \longleftrightarrow $i \leq length\ ys$
by (induction [] :: 'x list ys i rule: *sublist-at.induct*) auto

Furthermore, we need:

lemma *sublist-step*[intro]:
 $\llbracket i + length\ xs < length\ ys; \textit{sublist-at}\ xs\ ys\ i; ys!(i + length\ xs) = x \rrbracket \implies \textit{sublist-at}\ (xs@[x])\ ys\ i$

apply (*induction xs ys i rule: sublist-at.induct*)
apply *auto*
using *sublist-at.elims(3)* **by** *fastforce*

lemma *all-positions-sublist:*

$\llbracket i + \text{length } xs \leq \text{length } ys; \forall jj < \text{length } xs. \text{ys}!(i+jj) = \text{xs}!jj \rrbracket \implies \text{sublist-at } xs \text{ } ys \text{ } i$

proof (*induction xs rule: rev-induct*)

case *Nil*

then show *?case* **by** (*simp add: Nil-is-sublist*)

next

case (*snoc x xs*)

from $\langle i + \text{length } (xs @ [x]) \leq \text{length } ys \rangle$ **have** $i + \text{length } xs \leq \text{length } ys$ **by** *simp*

moreover have $\forall jj < \text{length } xs. \text{ys}!(i+jj) = \text{xs}!jj$

by (*simp add: nth-append snoc.prem(2)*)

ultimately have *sublist-at xs ys i*

using *snoc.IH* **by** *simp*

then show *?case*

using *snoc.prem(1)* **by** *auto*

qed

lemma *sublist-all-positions: sublist-at xs ys i $\implies \forall jj < \text{length } xs. \text{ys}!(i+jj) = \text{xs}!jj$*

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:*

$\text{sublist-at } xs \text{ } ys \text{ } i \iff (\exists ps \text{ } ss. \text{ys} = \text{ps}@xs@ss \wedge i = \text{length } ps)$

proof (*induction xs ys i rule: sublist-at.induct*)

case (*2 ss t ts i*)

show $\text{sublist-at } ss \text{ } (t\#ts) \text{ } (\text{Suc } i) \iff (\exists xs \text{ } ys. t\#ts = xs@ss@ys \wedge \text{Suc } i = \text{length } xs)$

(*is ?lhs \iff ?rhs*)

proof

assume *?lhs*

then have *sublist-at ss ts i* **by** *simp*

with *2.IH* **obtain** *xs* **where** $\exists \text{ys}. \text{ts} = \text{xs}@ss@ys \wedge i = \text{length } xs$ **by** *auto*

then have $\exists \text{ys}. t\#ts = (t\#xs)@ss@ys \wedge \text{Suc } i = \text{length } (t\#xs)$ **by** *simp*

then show *?rhs* **by** *blast*

next

assume *?rhs*

then obtain *xs* **where** $\exists \text{ys}. t\#ts = \text{xs}@ss@ys \wedge \text{length } xs = \text{Suc } i$

by (*blast dest: sym*)

then have $\exists \text{ys}. \text{ts} = (\text{tl } xs)@ss@ys \wedge i = \text{length } (\text{tl } xs)$

by (*auto simp add: length-Suc-conv*)

then have $\exists xs \text{ } ys. \text{ts} = \text{xs}@ss@ys \wedge i = \text{length } xs$ **by** *blast*

with *2.IH* **show** *?lhs* **by** *simp*

qed

qed *auto*

corollary *sublist-iff-sublist-at: Sublist.sublist xs ys $\iff (\exists i. \text{sublist-at } xs \text{ } ys \text{ } i)$*

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* $s\ t = SPEC\ (\lambda$
 $None \Rightarrow \nexists i. sublist\text{-}at\ s\ t\ i \mid$
 $Some\ i \Rightarrow sublist\text{-}at\ s\ t\ i \wedge (\forall ii < i. \neg sublist\text{-}at\ s\ t\ ii))$

lemma *is-arg-min-id*: $is\text{-}arg\text{-}min\ id\ P\ i \longleftrightarrow P\ i \wedge (\forall ii < i. \neg P\ ii)$
unfolding *is-arg-min-def* **by** *auto*

lemma *kmp-result*: $kmp\text{-}SPEC\ s\ t =$
 $RETURN\ (if\ sublist\ s\ t\ then\ Some\ (LEAST\ i. sublist\text{-}at\ s\ t\ i)\ else\ None)$
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*: $kmp\text{-}SPEC\ s\ t \leq SPEC\ (\lambda pos. pos \neq None \longleftrightarrow Sublist.sublist\ s\ t)$
by (*simp* *add*: *kmp-result*)

lemmas *kmp-SPEC-altdefs* =
 $kmp\text{-}SPEC\text{-}def[folded\ is\text{-}arg\text{-}min\text{-}id]$
 $kmp\text{-}SPEC\text{-}def[folded\ sublist\text{-}iff\text{-}sublist\text{-}at]$
 $kmp\text{-}result$

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* $s\ t \equiv \lambda(i,j,pos).$
 $(\forall ii < i. \neg sublist\text{-}at\ s\ t\ ii) \wedge$
 $(case\ pos\ of\ None \Rightarrow j = 0$
 $\mid Some\ p \Rightarrow p=i \wedge sublist\text{-}at\ s\ t\ i)$

definition *I-in-na* $s\ t\ i \equiv \lambda(j,pos).$
 $case\ pos\ of\ None \Rightarrow j < length\ s \wedge (\forall jj < j. t!(i+jj) = s!(jj))$
 $\mid Some\ p \Rightarrow sublist\text{-}at\ s\ t\ i$

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 int for list indexes (or the additional precondition $\text{length } s \leq \text{length } t$)

definition *naive-algorithm* $s\ t \equiv$ do {
 $let\ i=0;$
 $let\ j=0;$
 $let\ pos=None;$
 $(-,pos) \leftarrow WHILEIT\ (I-out-na\ s\ t)\ (\lambda(i,-,pos).\ i + \text{length } s \leq \text{length } t \wedge pos=None)\ (\lambda(i,j,pos).\ do\ \{$
 $(-,pos) \leftarrow WHILEIT\ (I-in-na\ s\ t\ i)\ (\lambda(j,pos).\ t!(i+j) = s!j \wedge pos=None)\ (\lambda(j,-).$
 $do\ \{$
 $let\ j=j+1;$
 $if\ j=length\ s\ then\ RETURN\ (j,Some\ i)\ else\ RETURN\ (j,None)$
 $\})\ (j,pos);$
 $if\ pos=None\ then\ do\ \{$
 $let\ i = i + 1;$
 $let\ j = 0;$
 $RETURN\ (i,j,None)$
 $\}\ else\ RETURN\ (i,j,Some\ i)$
 $\})\ (i,j,pos);$
 $RETURN\ pos$
}

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 $s \neq [] \implies \text{naive-algorithm } s\ t \leq \text{kmp-SPEC } s\ t$
unfolding *naive-algorithm-def kmp-SPEC-def I-out-na-def I-in-na-def*
apply (*refine-vcg*
 $WHILEIT-rule[\mathbf{where}\ R=measure\ (\lambda(i,-,pos). \text{length } t - i + (if\ pos = None\ then\ 1\ else\ 0))]$
 $WHILEIT-rule[\mathbf{where}\ R=measure\ (\lambda(j,-::nat\ option). \text{length } s - j)]$
)
apply (*vc-solve solve: asm-rl*)
subgoal by (*metis add-Suc-right all-positions-sublist less-antisym*)

```

subgoal using less-Suc-eq by blast
subgoal by (metis less-SucE sublist-all-positions)
subgoal by (auto split: option.splits) (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  $xs\ ys \longleftrightarrow prefix\ xs\ ys \wedge suffix\ xs\ ys$ 
definition strict-border  $xs\ ys \longleftrightarrow border\ xs\ ys \wedge length\ xs < length\ ys$ 
definition intrinsic-border  $ls \equiv ARG-MAX\ length\ b.\ strict-border\ b\ ls$ 

```

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  $xs\ ys$  and 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  $xs\ ys$  and  $length\ xs < length\ ys$ 
using assms unfolding strict-border-def by blast

```

```

lemma strict-border-simps[simp]:
strict-border  $xs\ [] \longleftrightarrow False$ 
strict-border  $[]\ (x \# xs) \longleftrightarrow True$ 
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*: $strict\text{-}border\ xs\ ys \longleftrightarrow strict\text{-}prefix\ xs\ ys \wedge strict\text{-}suffix\ xs\ ys$

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* : $\forall xs. strict\text{-}border\ xs\ ys \longrightarrow length\ xs < length\ ys$

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*: $\llbracket i \leq length\ w; i \leq length\ x;$

$\forall j < i. x!j = w!(length\ w + j - i)\rrbracket$

$\implies drop\ (length\ w - i)\ w = take\ i\ x$

by (*cases i = length x*) (*auto intro: nth-equalityI*)

lemma *all-positions-suffix-take*: $\llbracket i \leq length\ w; i \leq length\ x;$

$\forall j < i. x!j = w!(length\ w + j - i)\rrbracket$

$\implies suffix\ (take\ i\ x)\ w$

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*: $\forall j < l. w!j = w!(length\ w - l + j) \implies border\ (take\ l\ w)$

w

by (*cases l < length w*) (*simp-all add: border-def all-positions-suffix-take take-is-prefix*)

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

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*: $ys \neq [] \implies is\text{-}arg\text{-}max\ length\ (\lambda xs. strict\text{-}border\ xs\ ys)$ (*intrinsic-border ys*)

by (*simp add: intrinsic-borderI' intrinsic-border-max is-arg-max-linorder*)

lemma *intrinsic-border-less*: $w \neq [] \implies \text{length } (\text{intrinsic-border } w) < \text{length } w$
using *intrinsic-borderI*[of w] *border-length-r-less* *intrinsic-borderI* **by** *blast*

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

3.1.2 Examples

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

proof

show $\{?take0, ?take1, ?take2, ?take5, ?l\} \subseteq \{b. \text{border } b \ ?l\}$

by *simp eval*

have $\neg \text{border "aab" ?l} \neg \text{border "aba" ?l} \neg \text{border "aabaab" ?l} \neg \text{border "aabaaba" ?l}$

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 $\{b. \text{border } b \ ?l\} \subseteq \{?take0, ?take1, ?take2, ?take5, ?l\}$

by *auto*

qed

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

proof

have $?l \subseteq \{b. \text{border } b \ \text{"aabaabaa"}\}$

by *auto*

also have $\dots = \{''', 'a', 'aa', 'aaba', 'aabaabaa'\}$

by (*fact* *border-example*)

finally show $?l \subseteq ?r$ **by** *auto*

show $?r \subseteq ?l$ **by** *simp eval*

qed

corollary *intrinsic-border "aabaabaa" = "aaba"*

proof — We later obtain a fast algorithm for that.

have *exhaust*: *strict-border* $b \ \text{"aabaabaa"} \iff b \in \{''', 'a', 'aa', 'aaba'\}$ **for** b

using *strict-border-example* **by** *auto*

then have

$\neg \text{is-arg-max length } (\lambda b. \text{strict-border } b \ \text{"aabaabaa"}) \ \text{'''}$

$\neg \text{is-arg-max length } (\lambda b. \text{strict-border } b \ \text{"aabaabaa"}) \ 'a'$

$\neg \text{is-arg-max length } (\lambda b. \text{strict-border } b \ \text{"aabaabaa"}) \ 'aa'$

$\text{is-arg-max length } (\lambda b. \text{strict-border } b \ \text{"aabaabaa"}) \ \text{"aaba"}$

unfolding *is-arg-max-linorder* **by** *auto*

moreover have *strict-border* (*intrinsic-border* "aabaabaa") "aabaabaa"

```

    using intrinsic-borderI' by blast
    note this[unfolding exhaust]
    ultimately show ?thesis
    by simp (metis list.discI nonempty-is-arg-max-ib)
qed

```

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

```

Note that we use their "next" only implicitly.

3.2.1 Invariants

```

definition I-outer s t  $\equiv$   $\lambda(i,j,pos)$ .
  ( $\forall ii < i. \neg$ sublist-at s t ii)  $\wedge$ 
  (case pos of None  $\Rightarrow$  ( $\forall jj < j. t!(i+jj) = s!(jj)$ )  $\wedge$  j < length s
   | Some p  $\Rightarrow$  p=i  $\wedge$  sublist-at s t i)

```

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

3.2.2 Algorithm

First, we use the non-evaluable function *f* directly:

```

definition kmp s t  $\equiv$  do {
  ASSERT (s  $\neq$  []);
  let i=0;
  let j=0;
  let pos=None;
  ( $\cdot, \cdot, pos$ )  $\leftarrow$  WHILEIT (I-outer s t) ( $\lambda(i,j,pos)$ . i + length s  $\leq$  length t  $\wedge$  pos=None)
  ( $\lambda(i,j,pos)$ . do {
    ASSERT (i + length s  $\leq$  length t);
    ( $j, pos$ )  $\leftarrow$  WHILEIT (I-in-na s t i) ( $\lambda(j,pos)$ . t!(i+j) = s!j  $\wedge$  pos=None)
  }) ( $\lambda(j,pos)$ . do {
    let j=j+1;
    if j=length s then RETURN (j,Some i) else RETURN (j,None)
  }) (j,pos);
  if pos=None then do {
    ASSERT (j < length s);
    let i = i + (j - f s j + 1);
    let j = max 0 (f s j - 1); — max not necessary
    RETURN (i,j,None)
  }
}

```

```

    } else RETURN (i,j,Some i)
  }) (i,j,pos);

  RETURN pos
}

```

3.2.3 Correctness

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

lemma $j\text{-le-f-le}$: $j \leq \text{length } s \implies f\ s\ j \leq j$
apply (cases j)
apply simp-all
by (metis Suc-leI intrinsic-border-less length-take list.size(3) min.absorb2 nat.simps(3) not-less)

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

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

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

theorem shift-safe:

assumes
 $\forall ii < i. \neg \text{sublist-at } s \ t \ ii$
 $t!(i+j) \neq s!j$ **and**
 $[simp]: j < \text{length } s$ **and**
 $\text{matches}: \forall jj < j. t!(i+jj) = s!jj$

defines
 $\text{assignment}: i' \equiv i + (j - f \ s \ j + 1)$

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:
(old) $ii < i$ |
(current) $ii = i$ |
(skipped) $ii > i$
by *linarith*
then show $\neg \text{sublist-at } s \ t \ ii$
proof *cases*
case *old* — Old position, use invariant.
with $\langle \forall ii < i. \neg \text{sublist-at } s \ t \ ii \rangle$ **show** *?thesis by simp*
next
case *current* — The mismatch occurred while testing this alignment.
with $\langle t!(i+j) \neq s!j \rangle$ **show** *?thesis*
using *sublist-all-positions[of s t i]* **by** *auto*
next
case *skipped* — The skipped positions.
then have $0 < j$
using $\langle ii < i' \rangle$ *assignment by linarith*
then have $\text{less-}j[simp]: j + i - ii < j$ **and** $\text{le-}s: j + i - ii \leq \text{length } s$
using $\langle ii < i' \rangle$ *assms(3) skipped by linarith+*
note $f\text{-le}[simp] = j\text{-le-}f\text{-le}[OF \ \text{assms}(3)[THEN \ \text{less-imp-le}]$
have $0 < f \ s \ j$
using $\langle 0 < j \rangle$ *f-eq-0-iff-j-eq-0 neq0-conv by blast*
then have $j + i - ii > f \ s \ j - 1$
using $\langle ii < i' \rangle$ *assignment f-le by linarith*
then have *contradiction-goal: j + i - ii > length (intrinsic-border (take j s))*
by (*metis f.elims <0 < j> add-diff-cancel-right' not-gr-zero*)
show *?thesis*
proof
assume *sublist-at s t ii*
note *sublist-all-positions[OF this]*
with $\text{le-}s$ **have** $a: \forall jj < j+i-ii. t!(ii+jj) = s!jj$
by *simp*
have $ff1: \neg ii < i$
by (*metis not-less-iff-gr-or-eq skipped*)
then have $i + (ii - i + jj) = ii + jj$ **for** jj
by (*metis add.assoc add-diff-inverse-nat*)
then have $\neg jj < j + i - ii \vee t! (ii + jj) = s! (ii - i + jj)$ **if** $ii - i + jj$

```

< j for jj
  using that ff1 by (metis matches)
  then have  $\neg jj < j + i - ii \vee t! (ii + jj) = s! (ii - i + jj)$  for jj
  using ff1 by auto
  with matches have  $\forall jj < j+i-ii. t!(ii+jj) = s!(ii-i+jj)$  by metis
  then have  $\forall jj < j+i-ii. s!jj = s!(ii-i+jj)$ 
  using a by auto
  then have  $\forall jj < j+i-ii. (take\ j\ s)!jj = (take\ j\ s)!(ii-i+jj)$ 
  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  $j+i-ii \leq length\ s$  by (fact le-s)
  ultimately
  show False by simp
qed
qed
qed

lemma kmp-correct:  $s \neq []$ 
 $\implies kmp\ s\ t \leq kmp-SPEC\ s\ t$ 
  unfolding kmp-def kmp-SPEC-def I-outer-def I-in-na-def
  apply (refine-vcg
    WHILEIT-rule[where R=measure ( $\lambda(i,-,pos). length\ t - i + (if\ pos = None$ 
then 1 else 0))])
    WHILEIT-rule[where R=measure ( $\lambda(j,-::nat\ option). length\ s - j$ )
  ])
    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] f-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 \Rightarrow nat list nres **where**
 $compute-fs-SPEC\ s \equiv SPEC\ (\lambda fs. length\ fs = length\ s + 1 \wedge (\forall j \leq length\ s. fs!j = f\ s\ j))$

definition *kmp1* $s\ t \equiv do\ \{$
 ASSERT ($s \neq []$);
 let $i=0$;
 let $j=0$;
 let $pos=None$;

```

  fs ← compute-fs-SPEC (butlast s); — At the last char, we abort instead.
  (-, pos) ← WHILEIT (I-outer s t) (λ(i, j, pos). i + length s ≤ length t ∧ pos=None)
(λ(i, j, pos). do {
  ASSERT (i + length s ≤ length t);
  (j, pos) ← WHILEIT (I-in-na s t i) (λ(j, pos). t!(i+j) = s!j ∧ pos=None)
(λ(j, pos). do {
  let j=j+1;
  if j=length s then RETURN (j, Some i) else RETURN (j, None)
}) (j, pos);
if pos=None then do {
  ASSERT (j < length fs);
  let i = i + (j - fs!j + 1);
  let j = max 0 (fs!j - 1); — max not necessary
  RETURN (i, j, None)
} else RETURN (i, j, Some i)
}) (i, j, pos);

RETURN pos
}

```

lemma f-butlast[simp]: $j < \text{length } s \implies \text{f } (\text{butlast } s) \text{ } j = \text{f } s \text{ } j$
by (cases j) (simp-all add: take-butlast)

lemma kmp1-refine: $\text{kmp1 } s \text{ } t \leq \text{kmp } s \text{ } t$
apply (rule refine-IdD)
unfolding kmp1-def kmp-def Let-def compute-fs-SPEC-def nres-monad-laws
apply (intro ASSERT-refine-right ASSERT-refine-left)
apply simp
apply (rule Refine-Basic.intro-spec-refine)
apply refine-req
 apply refine-dref-type
 apply vc-solve
done

Next, an algorithm that satisfies *compute-fs-SPEC*:

3.3 Computing f

3.3.1 Invariants

definition I-out-cb s ≡ λ(fs, i, j).

length s + 1 = length fs ∧
 (∀ jj < j. fs!jj = f s jj) ∧
 fs!(j-1) = i ∧
 0 < j

definition I-in-cb s j ≡ λi.

if j=1 then i=0 — first iteration
 else
 strict-border (take (i-1) s) (take (j-1) s) ∧
 f s j ≤ 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 $\text{fs} ! 1$, 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-fs* :: 'a list \Rightarrow nat list nres **where**
compute-fs $s = \text{do}$ {
 let $\text{fs} = \text{replicate}$ (length $s + 1$) 0; — only the first 0 is needed
 let $i = 0$;
 let $j = 1$;
 ($\text{fs}, -, -$) \leftarrow WHILEIT (I-out-cb s) ($\lambda(\text{fs}, -, j). j < \text{length } \text{fs}$) ($\lambda(\text{fs}, i, j). \text{do}$ {
 $i \leftarrow$ WHILEIT (I-in-cb s j) ($\lambda i. i > 0 \wedge s!(i-1) \neq s!(j-1)$) ($\lambda i. \text{do}$ {
 ASSERT ($i-1 < \text{length } \text{fs}$);
 let $i = \text{fs}!(i-1)$;
 RETURN i
 }) i ;
 let $i = i + 1$;
 ASSERT ($j < \text{length } \text{fs}$);
 let $\text{fs} = \text{fs}[j := i]$;
 let $j = j + 1$;
 RETURN (fs, i, j)
}) (fs, i, j);
 RETURN fs
}

3.3.3 Correctness

lemma *take-length-ib[simp]*:
 assumes $0 < j \leq \text{length } s$
 shows $\text{take} (\text{length} (\text{intrinsic-border} (\text{take } j s))) s = \text{intrinsic-border} (\text{take } j s)$
proof —
 from *assms* have *prefix* ($\text{intrinsic-border} (\text{take } j s)$) ($\text{take } j s$)
 by (*metis intrinsic-borderI' border-def list.size(3) neq0-conv not-less strict-border-def take-eq-Nil*)
 also have *prefix* ($\text{take } j s$) s
 by (*simp add: <j ≤ length s> take-is-prefix*)
 finally show *?thesis*
 by (*metis append-eq-conv-conj prefixE*)
qed

lemma *ib-singleton[simp]*: $\text{intrinsic-border } [z] = []$
 by (*metis intrinsic-border-less length-Cons length-greater-0-conv less-Suc0 list.size(3)*)

lemma *border-butlast*: $\text{border } xs \ ys \Longrightarrow \text{border} (\text{butlast } xs) (\text{butlast } ys)$
 apply (*auto simp: border-def*)
 apply (*metis butlast-append prefixE prefix-order.eq-refl prefix-prefix prefixeq-butlast*)
 apply (*metis Sublist.suffix-def append.right-neutral butlast.simps(1) butlast-append*)

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*: $\text{border } xs \ ys \longleftrightarrow \text{border } (xs@[ys!\text{length } xs]) \ (ys@[ys!\text{length } xs])$

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

using *append-one-prefix prefixE* **apply** *fastforce*

using *append-prefixD* **apply** *blast*

done

corollary *strict-border-step*: $\text{strict-border } xs \ ys \longleftrightarrow \text{strict-border } (xs@[ys!\text{length } xs]) \ (ys@[ys!\text{length } xs])$

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 $\text{strict-border } (\text{intrinsic-border } w) \ w$

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 lessI 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 *?thesis*

by *simp*

qed

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

apply (*cases i*)

apply (*simp-all add: take-Suc0*)

by (*metis One-nat-def Suc-eq-plus1 Suc-to-right butlast-take diff-is-0-eq ib-butlast length-take min.absorb2 nat.simps(3) not-less-eq-eq numerals(2)*)

lemma *f-step-bound*:

assumes $j \leq \text{length } w$

shows $f \ w \ j \leq f \ w \ (j-1) + 1$

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

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

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

apply (*cases i, simp-all*)
by (*metis intrinsic-borderI' border-order.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*: $y = f\ s\ (i-1) \implies \text{strict-border } (take\ (i-1)\ s)\ (take\ (j-1)\ s) \implies \text{strict-border } (take\ (y-1)\ s)\ (take\ (j-1)\ s)$
using *border-order.less-le-not-le border-order.order.trans border-take-f by blast*

corollary *strict-border-take-f*: $0 < i \implies i \leq \text{length } s \implies \text{strict-border } (take\ (f\ s\ i - 1)\ s)\ (take\ i\ s)$
by (*meson border-order.less-le-not-le border-take-f border-take-lengths j-le-f-le' leD*)

lemma *f-is-max*: $j \leq \text{length } s \implies \text{strict-border } b\ (take\ j\ s) \implies f\ s\ j \geq \text{length } b + 1$
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 *j-bounds[simp]*: $1 < j \leq \text{length } s$
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 ccontr*)

assume $\neg f\ s\ j \leq f\ s\ (i-1) + 1$

then have *i-bounds*: $0 < i \leq \text{length } s$

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 $\langle \neg f\ s\ j \leq f\ s\ (i-1) + 1 \rangle$ *greater-checked consider*

(tested) $f\ s\ j = i + 1$ — This contradicts $s!(i-1) \neq s!(j-1)$ |

(skipped) $f\ s\ (i-1) + 1 < f\ s\ j$ $f\ s\ j \leq i$

— This contradicts $\llbracket i - 1 \leq \text{length } s; \text{strict-border } ?b\ (take\ (i - 1)\ s) \rrbracket \implies \text{length } ?b + 1 \leq f\ s\ (i - 1)$

by *linarith*

then show *False*

proof *cases*

case *tested*

then have $f\ s\ j - 1 = i$ **by** *simp*

moreover note *border-positions[OF border-take-f[of s j, unfolded this]]*

ultimately have $take\ j\ s!(i-1) = s!(j-1)$ **using** *i-bounds i-less-j by simp*

with $\langle i < j \rangle$ **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 $\text{take } (j - 1) s$ could not be extended to a border of $\text{take } j s$ due to the mismatch.

```

let ?impossible = take (f s j - 2) s
— A strict border longer than intrinsic-border ( $\text{take } (i - 1) s$ ), a contradiction.
have length (take j s) = j
  by simp
have f s j - 2 < i - 1
  using skipped by linarith
then have less-s: f s j - 2 < length s i - 1 < length s
  using <i < j> j-bounds(2) by linarith+
then have strict: length ?impossible < length ?border
  using <f s j - 2 < i - 1> by auto
moreover {
  have prefix ?impossible (take j s)
    using prefix-length-prefix take-is-prefix
    by (metis (no-types, lifting) <length (take j s) = j> j-bounds(2) diff-le-self
j-le-f-le length-take less-s(1) min-simps(2) order-trans)
  moreover have prefix ?border (take j s)
    by (metis (no-types, lifting) <length (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 ?impossible ?border
    using strict less-imp-le-nat prefix-length-prefix by blast
} moreover {
  have suffix (take (f s j - 1) s) (take j s) using border-take-f
    by (auto simp: border-def)
  note suffix-butlast[OF this]
  then have suffix ?impossible (take (j-1) s)
    by (metis One-nat-def j-bounds(2) butlast-take diff-diff-left f-le len-greater-imp-nonempty
less-or-eq-imp-le less-s(2) one-add-one)
  then have suffix ?impossible (take (j-1) s) suffix ?border (take (j-1) s)
    using assms(5) by auto
  from suffix-length-suffix[OF this strict[THEN less-imp-le]]
  have suffix ?impossible ?border.
}
ultimately have strict-border ?impossible ?border
  unfolding strict-border-def[unfolded border-def] by blast
note f-is-max[of i-1 s, OF - this]
then have length (take (f s j - 2) s) + 1 ≤ f s (i-1)
  using less-imp-le-nat less-s(2) by blast
then have f s j - 1 ≤ f s (i-1)
  by (simp add: less-s(1))
then have f s j ≤ f s (i-1) + 1
  using le-diff-conv by blast
with skipped(1) show False
  by linarith
qed
qed

```

lemma *extend-border*:

assumes $j \leq \text{length } s$
assumes $s!(i-1) = s!(j-1)$
assumes *strict-border* (take (i-1) s) (take (j-1) s)
assumes $\text{f } s \ j \leq i + 1$
shows $\text{f } s \ j = i + 1$
proof –
from *assms*(β) **have** *pos-in-range*: $i - 1 < \text{length } s$ *length* (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*(β)] **have** *strict-border* (take (i-1) s @ [s!(i-1)]) (take (j-1) s @ [s!(i-1)])
by (*metis assms*(β) *border-length-r-less length-take min-less-iff-conj nth-take*)
with *pos-in-range* **have** *strict-border* (take i s) (take (j-1) s @ [s!(i-1)])
by (*metis Suc-eq-plus1 Suc-pred add.left-neutral border-bot.bot.not-eq-extremum border-order.less-asym neq0-conv take-0 take-Suc-conv-app-nth*)
then **have** *strict-border* (take i s) (take (j-1) s @ [s!(j-1)])
by (*simp only*: $\langle s!(i-1) = s!(j-1) \rangle$)
then **have** *strict-border* (take i s) (take j s)
by (*metis One-nat-def Suc-pred assms*(1, β) *diff-le-self less-le-trans neq0-conv nz-le-conv-less strict-border-imp-nonempty take-Suc-conv-app-nth take-eq-Nil*)
with *f-is-max*[*OF assms*(1) *this*] **have** $\text{f } s \ j \geq i + 1$
using *Suc-leI* **by** *fastforce*
with $\langle \text{f } s \ j \leq i + 1 \rangle$ **show** *?thesis*
using *le-antisym* **by** *presburger*
qed

lemma *compute-fs-correct*: $\text{compute-fs } s \leq \text{compute-fs-SPEC } s$
unfolding *compute-fs-SPEC-def compute-fs-def I-out-cb-def I-in-cb-def*
apply (*simp, refine-vcg*
WHILEIT-rule[**where** *R=measure* ($\lambda(\text{fs}, i, j). \text{length } s + 1 - j$)]
WHILEIT-rule[**where** *R=measure id*] — i decreases with every iteration.
))
apply (*vc-solve, fold One-nat-def*)
subgoal for $b \ j$ **by** (*rule strict-border-take-f, auto*)
subgoal by (*metis Suc-eq-plus1 f-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 f-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-leI border-take-lengths less-Suc-eq-le less-antisym skipping-ok strict-border-def*)
subgoal by (*metis Suc-diff-1 border-take-lengths j-le-f-le less-Suc-eq-le strict-border-def*)
subgoal for $b \ j \ i \ jj$
by (*metis Suc-eq-plus1 Suc-eq-plus1-left add.right-neutral extend-border f-eq-0-iff-j-eq-0 j-le-f-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=fs!(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*

```

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

  RETURN pos
}

```

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

```

lemma kmp2-refine: kmp2 s t  $\leq$  kmp1 s t
apply (rule refine-IdD)
unfolding kmp2-def kmp1-def
apply refine-rcg
      apply refine-dref-type
      apply (vc-solve simp: in-br-conv)
done

```

```

lemma kmp2-correct: s  $\neq$  []
 $\implies$  kmp2 s t  $\leq$  kmp-SPEC s t
proof —
  assume s  $\neq$  []
  have kmp2 s t  $\leq$  kmp1 s t by (fact kmp2-refine)
  also have ...  $\leq$  kmp s t by (fact kmp1-refine)
  also have ...  $\leq$  kmp-SPEC s t by (fact kmp-correct[OF ‹s ≠ []›])
  finally show ?thesis.
qed

```

For convenience, we also remove the precondition:

```

definition kmp3 s t  $\equiv$  do {

```

```

  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 uncurry 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':
  (uncurry kmp3, uncurry kmp-SPEC) ∈ Id ×r Id →f ⟨Id⟩nres-rel
  apply (intro freI nres-relI; clarsimp)
  apply (fact kmp3-correct)

```

done

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.

theorem *kmp-impl-correct*:

```
< arl-assn id-assn s si * arl-assn id-assn t ti >
  kmp-impl si ti
< λr. arl-assn id-assn s si * arl-assn id-assn t ti * ↑(
  case r of None ⇒ ∄ i. sublist-at s t i
    | Some i ⇒ sublist-at s t i ∧ (∀ ii < i. ¬ sublist-at s t ii)
  ) >_t
by (sep-auto
  simp: pure-def kmp-SPEC-def
  split: option.split
  heap: kmp-impl-correct'[THEN hhrefD, THEN hn-refineD, of (s,t) (si,ti), sim-
  plified])
```

definition *kmp-string-impl* ≡ *kmp-impl* :: (char array × nat) ⇒ -

5 Tests of Generated ML-Code

ML-val <

```
fun str2arl s = (Array.fromList (@{code String.explode} s), @{code nat-of-integer}
(String.size s))
fun kmp s t = map-option @{code integer-of-nat} (@{code kmp-string-impl}
(str2arl s) (str2arl t) ())
```

```
val test1 = kmp anas bananas
val test2 = kmp bananas
val test3 = kmp hide-fact (File.read @{file <~~/src/HOL/Main.thy>})
val test4 = kmp sorry (File.read @{file <~~/src/HOL/HOL.thy>})
>
```

end

References

- [1] D. E. Knuth, J. James H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
- [2] P. Lammich. Refinement for monadic programs. *Archive of Formal Proofs*, Jan. 2012. http://isa-afp.org/entries/Refine_Monadic.html, Formal proof development.
- [3] P. Lammich. The imperative refinement framework. *Archive of Formal Proofs*, Aug. 2016. http://isa-afp.org/entries/Refine_Imperative_HOL.html, Formal proof development.
- [4] H. Seidl. Grundlagen: Algorithmen und Datenstrukturen. <http://www2.in.tum.de/hp/file?fd=1347>, German lecture notes, 2016.