

# The string search algorithm by Knuth, Morris and Pratt

Fabian Hellauer and Peter Lammich

September 23, 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

<b>1</b>	<b>Specification</b>	<b>3</b>
1.1	Sublist-predicate with a position check . . . . .	3
1.1.1	Definition . . . . .	3
1.1.2	Properties . . . . .	3
1.2	Sublist-check algorithms . . . . .	4
<b>2</b>	<b>Naive algorithm</b>	<b>4</b>
2.1	Invariants . . . . .	5
2.2	Algorithm . . . . .	5
2.3	Correctness . . . . .	5
<b>3</b>	<b>Knuth–Morris–Pratt algorithm</b>	<b>6</b>
3.1	Preliminaries: Borders of lists . . . . .	6
3.1.1	Properties . . . . .	6
3.1.2	Examples . . . . .	8
3.2	Main routine . . . . .	8
3.2.1	Invariants . . . . .	8
3.2.2	Algorithm . . . . .	8
3.2.3	Correctness . . . . .	9
3.2.4	Storing the $f$ -values . . . . .	10
3.3	Computing $f$ . . . . .	11
3.3.1	Invariants . . . . .	11

3.3.2	Algorithm . . . . .	11
3.3.3	Correctness . . . . .	11
3.3.4	Index shift . . . . .	13
3.4	Conflation . . . . .	14
<b>4</b>	<b>Refinement to Imperative/HOL</b>	<b>15</b>
4.1	Overall Correctness Theorem . . . . .	15
<b>5</b>	<b>Tests of Generated ML-Code</b>	<b>16</b>

```

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**  $i \leq length\ ys \implies sublist-at\ xs\ ys\ i \longleftrightarrow sublist-at'\ xs\ ys\ i$   
*<proof>*

However, the new definition has some reasonable properties:

#### 1.1.2 Properties

**lemma** *sublist-lengths*:  $sublist-at\ xs\ ys\ i \implies i + length\ xs \leq length\ ys$   
*<proof>*

**lemma** *Nil-is-sublist*:  $sublist-at\ ([] :: 'x\ list)\ ys\ i \longleftrightarrow i \leq length\ ys$   
*<proof>*

Furthermore, we need:

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

*<proof>*

**lemma** *all-positions-sublist*:

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

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

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

**lemma** *sublist-at-altdef*:

$\text{sublist-at } xs \text{ } ys \ i \longleftrightarrow (\exists ps \ ss. ys = ps@xs@ss \wedge i = \text{length } ps)$   
*<proof>*

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

## 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*  $s \ t = \text{SPEC } (\lambda$

$\text{None} \implies \nexists i. \text{sublist-at } s \ t \ i \mid$

$\text{Some } i \implies \text{sublist-at } s \ t \ i \wedge (\forall ii < i. \neg \text{sublist-at } s \ t \ ii)$

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

**lemma** *kmp-result*:  $\text{kmp-SPEC } s \ t =$

$\text{RETURN } (\text{if } \text{sublist } s \ t \ \text{then } \text{Some } (\text{LEAST } i. \text{sublist-at } s \ t \ i) \ \text{else } \text{None})$

*<proof>*

**corollary** *weak-kmp-SPEC*:  $\text{kmp-SPEC } s \ t \leq \text{SPEC } (\lambda pos. pos \neq \text{None} \longleftrightarrow \text{Sublist.sublist } s \ t)$

*<proof>*

**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**  $I\text{-out-na } s \ t \equiv \lambda(i,j,pos).$

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

**definition**  $I\text{-in-na } s \ t \ i \equiv \lambda(j,pos).$

$\text{case } pos \text{ of } None \Rightarrow j < \text{length } s \wedge (\forall jj < j. t!(i+jj) = s!(jj))$   
 $\mid \text{Some } p \Rightarrow \text{sublist-at } s \ t \ i$

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

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

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

## 2.3 Correctness

The basic lemmas on  $\text{sublist-at}$  from the previous chapter together with  $\text{Refine-Monadic.Refine-Monadic}$ 's verification condition generator / solver suffice:

**lemma**  $s \neq [] \implies \text{naive-algorithm } s \ t \leq \text{kmp-SPEC } s \ t$   
 ⟨proof⟩

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**  $\text{border } xs \ ys \longleftrightarrow \text{prefix } xs \ ys \wedge \text{suffix } xs \ ys$

**definition**  $\text{strict-border } xs \ ys \longleftrightarrow \text{border } xs \ ys \wedge \text{length } xs < \text{length } ys$

**definition**  $\text{intrinsic-border } ls \equiv \text{ARG-MAX length } b. \text{strict-border } b \ ls$

##### 3.1.1 Properties

**interpretation**  $\text{border-order}$ :  $\text{order border strict-border}$   
 ⟨proof⟩

**interpretation**  $\text{border-bot}$ :  $\text{order-bot Nil border strict-border}$   
 ⟨proof⟩

**lemma**  $\text{borderE}[\text{elim}]$ :  
**fixes**  $xs \ ys :: 'a \ \text{list}$   
**assumes**  $\text{border } xs \ ys$   
**obtains**  $\text{prefix } xs \ ys$  **and**  $\text{suffix } xs \ ys$   
 ⟨proof⟩

**lemma**  $\text{strict-borderE}[\text{elim}]$ :  
**fixes**  $xs \ ys :: 'a \ \text{list}$   
**assumes**  $\text{strict-border } xs \ ys$   
**obtains**  $\text{border } xs \ ys$  **and**  $\text{length } xs < \text{length } ys$   
 ⟨proof⟩

**lemma**  $\text{strict-border-simps}[\text{simp}]$ :  
 $\text{strict-border } xs \ [] \longleftrightarrow \text{False}$   
 $\text{strict-border } [] \ (x \ \# \ xs) \longleftrightarrow \text{True}$   
 ⟨proof⟩

**lemma**  $\text{strict-border-prefix}$ :  $\text{strict-border } xs \ ys \implies \text{strict-prefix } xs \ ys$   
**and**  $\text{strict-border-suffix}$ :  $\text{strict-border } xs \ ys \implies \text{strict-suffix } xs \ ys$   
**and**  $\text{strict-border-imp-nonempty}$ :  $\text{strict-border } xs \ ys \implies ys \neq []$   
**and**  $\text{strict-border-prefix-suffix}$ :  $\text{strict-border } xs \ ys \longleftrightarrow \text{strict-prefix } xs \ ys \wedge \text{strict-suffix } xs \ ys$   
 ⟨proof⟩

**lemma** *border-length-le*:  $\text{border } xs \ ys \implies \text{length } xs \leq \text{length } ys$

*<proof>*

**lemma** *border-length-r-less* :  $\forall xs. \text{strict-border } xs \ ys \implies \text{length } xs < \text{length } ys$

*<proof>*

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

*<proof>*

**lemma** *all-positions-drop-length-take*:  $\llbracket i \leq \text{length } w; i \leq \text{length } x;$

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

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

*<proof>*

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

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

$\implies \text{suffix } (\text{take } i \ x) \ w$

*<proof>*

**lemma** *suffix-butlast*:  $\text{suffix } xs \ ys \implies \text{suffix } (\text{butlast } xs) \ (\text{butlast } ys)$

*<proof>*

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

$w$

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *intrinsic-border-less*:  $w \neq [] \implies \text{length } (\text{intrinsic-border } w) < \text{length } w$

*<proof>*

**lemma** *intrinsic-border-take-less*:  $j > 0 \implies w \neq [] \implies \text{length } (\text{intrinsic-border$

$(take\ j\ w) < length\ w$   
 ⟨proof⟩

### 3.1.2 Examples

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

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

**corollary** *intrinsic-border*  $"aabaabaa" = "aaba"$   
 ⟨proof⟩

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

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

### 3.2.2 Algorithm

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

**definition**  $kmp\ s\ t \equiv do\ \{$   
 $ASSERT\ (s \neq []);$   
 $let\ i=0;$   
 $let\ j=0;$   
 $let\ pos=None;$   
 $(-,pos) \leftarrow WHILEIT\ (I-outer\ s\ t)\ (\lambda(i,j,pos). i + length\ s \leq length\ t \wedge pos=None)$   
 $(\lambda(i,j,pos). do\ \{$

```

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

RETURN  $pos$ 
}

```

### 3.2.3 Correctness

**lemma**  $f\text{-eq-0-iff-}j\text{-eq-0}$ [*simp*]:  $f \ s \ j = 0 \longleftrightarrow j = 0$   
 <proof>

**lemma**  $j\text{-le-}f\text{-le}$ :  $j \leq \text{length } s \implies f \ s \ j \leq j$   
 <proof>

**lemma**  $j\text{-le-}f\text{-le'}$ :  $0 < j \implies j \leq \text{length } s \implies f \ s \ j - 1 < j$   
 <proof>

**lemma**  $f\text{-le}$ :  $s \neq [] \implies f \ s \ j - 1 < \text{length } s$   
 <proof>

**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>

**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**

*matches*:  $\forall jj < j. t!(i+jj) = s!jj$

**defines**

*assignment*:  $i' \equiv i + (j - f \ s \ j + 1)$

**shows**

$\forall ii < i'. \neg \text{sublist-at } s \ t \ ii$   
 <proof>

**lemma** *kmp-correct*:  $s \neq []$   
 $\implies \text{kmp } s \ t \leq \text{kmp-SPEC } s \ t$   
 <proof>

### 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*  $s \equiv \text{SPEC } (\lambda fs. \text{length } fs = \text{length } s + 1 \wedge (\forall j \leq \text{length } s. fs!j = f \ s \ j))$

**definition** *kmp1*  $s \ t \equiv \text{do } \{$   
*ASSERT* ( $s \neq []$ );  
*let*  $i=0$ ;  
*let*  $j=0$ ;  
*let*  $pos=None$ ;  
 $fs \leftarrow \text{compute-fs-SPEC } (\text{butlast } s)$ ; — At the last char, we abort instead.  
 $(-, -, pos) \leftarrow \text{WHILEIT } (I\text{-outer } s \ t) (\lambda(i, j, pos). i + \text{length } s \leq \text{length } t \wedge pos=None)$   
 $(\lambda(i, j, pos). \text{do } \{$   
*ASSERT* ( $i + \text{length } s \leq \text{length } t$ );  
 $(j, pos) \leftarrow \text{WHILEIT } (I\text{-in-na } s \ t \ i) (\lambda(j, pos). t!(i+j) = s!j \wedge pos=None)$   
 $(\lambda(j, pos). \text{do } \{$   
*let*  $j=j+1$ ;  
*if*  $j=\text{length } s$  *then* *RETURN* ( $j, \text{Some } i$ ) *else* *RETURN* ( $j, None$ )  
 $\}) (j, pos)$ ;  
*if*  $pos=None$  *then* *do* {  
*ASSERT* ( $j < \text{length } fs$ );  
*let*  $i = i + (j - fs!j + 1)$ ;  
*let*  $j = \max \ 0 \ (fs!j - 1)$ ; — *max* not necessary  
*RETURN* ( $i, j, None$ )  
 $\}$  *else* *RETURN* ( $i, j, \text{Some } i$ )  
 $\}) (i, j, pos)$ ;  
*RETURN*  $pos$   
 $\}$

**lemma** *f-butlast[simp]*:  $j < \text{length } s \implies f (\text{butlast } s) \ j = f \ s \ j$   
 <proof>

**lemma** *kmp1-refine*:  $\text{kmp1 } s \ t \leq \text{kmp } s \ t$   
 <proof>

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

### 3.3 Computing $f$

#### 3.3.1 Invariants

**definition**  $I\text{-out-cb } s \equiv \lambda(\mathfrak{f}s, i, j).$

$length\ s + 1 = length\ \mathfrak{f}s \wedge$   
 $(\forall jj < j. \mathfrak{f}s!jj = \mathfrak{f}\ s\ jj) \wedge$   
 $\mathfrak{f}s!(j-1) = i \wedge$   
 $0 < j$

**definition**  $I\text{-in-cb } s\ j \equiv \lambda i.$

$if\ j=1\ then\ i=0$  — first iteration  
 $else$   
 $strict\text{-border } (take\ (i-1)\ s)\ (take\ (j-1)\ s) \wedge$   
 $\mathfrak{f}\ s\ j \leq i + 1$

#### 3.3.2 Algorithm

Again, we follow Seidl[4], p.582. Apart from the +1-shift, we make another modification: Instead of directly setting  $\mathfrak{f}s ! 1$ , 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**  $compute\text{-}\mathfrak{f}s :: 'a\ list \Rightarrow nat\ list\ nres$  **where**

```

compute- $\mathfrak{f}s\ s = do\ \{
  let\ \mathfrak{f}s = replicate\ (length\ s + 1)\ 0; \text{--- only the first 0 is needed}
  let\ i = 0;
  let\ j = 1;
  ( $\mathfrak{f}s, -, -$ )  $\leftarrow$  WHILEIT (I-out-cb s) ( $\lambda(\mathfrak{f}s, -, j). j < length\ \mathfrak{f}s$ ) ( $\lambda(\mathfrak{f}s, i, j). do\ \{$ 
     $i \leftarrow$  WHILEIT (I-in-cb s j) ( $\lambda i. i > 0 \wedge s!(i-1) \neq s!(j-1)$ ) ( $\lambda i. do\ \{$ 
      ASSERT ( $i-1 < length\ \mathfrak{f}s$ );
      let  $i = \mathfrak{f}s!(i-1)$ ;
      RETURN  $i$ 
    })  $i$ ;
    let  $i = i + 1$ ;
    ASSERT ( $j < length\ \mathfrak{f}s$ );
    let  $\mathfrak{f}s = \mathfrak{f}s[j := i]$ ;
    let  $j = j + 1$ ;
    RETURN ( $\mathfrak{f}s, i, j$ )
  }) ( $\mathfrak{f}s, i, j$ );

  RETURN  $\mathfrak{f}s$ 
}$ 
```

#### 3.3.3 Correctness

**lemma**  $take\text{-}length\text{-}ib[simp]:$

**assumes**  $0 < j \leq length\ s$

**shows**  $take\ (length\ (intrinsic\text{-}border\ (take\ j\ s)))\ s = intrinsic\text{-}border\ (take\ j\ s)$

$\langle proof \rangle$

**lemma** *ib-singleton[simp]*:  $\text{intrinsic-border } [z] = []$   
*<proof>*

**lemma** *border-butlast*:  $\text{border } xs \ ys \implies \text{border } (\text{butlast } xs) (\text{butlast } ys)$   
*<proof>*

**corollary** *strict-border-butlast*:  $xs \neq [] \implies \text{strict-border } xs \ ys \implies \text{strict-border } (\text{butlast } xs) (\text{butlast } ys)$   
*<proof>*

**lemma** *border-take-lengths*:  $i \leq \text{length } s \implies \text{border } (\text{take } i \ s) (\text{take } j \ s) \implies i \leq j$   
*<proof>*

**lemma** *border-step*:  $\text{border } xs \ ys \longleftrightarrow \text{border } (xs@[ys!\text{length } xs]) (ys@[ys!\text{length } xs])$   
*<proof>*

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

**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>*

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

**lemma** *f-step-bound*:  
  **assumes**  $j \leq \text{length } w$   
  **shows**  $\mathbf{f} \ w \ j \leq \mathbf{f} \ w \ (j-1) + 1$   
*<proof>*

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

**corollary** *f-strict-borderI*:  $y = \mathbf{f} \ s \ (i-1) \implies \text{strict-border } (\text{take } (i-1) \ s) (\text{take } (j-1) \ s) \implies \text{strict-border } (\text{take } (y-1) \ s) (\text{take } (j-1) \ s)$   
*<proof>*

**corollary** *strict-border-take-f*:  $0 < i \implies i \leq \text{length } s \implies \text{strict-border } (\text{take } (\mathbf{f} \ s \ i - 1) \ s) (\text{take } i \ s)$   
*<proof>*

**lemma** *f-is-max*:  $j \leq \text{length } s \implies \text{strict-border } b \ (\text{take } j \ s) \implies \mathbf{f} \ s \ j \geq \text{length } b + 1$   
*<proof>*

**theorem** *skipping-ok*:  
  **assumes** *j-bounds[simp]*:  $1 < j \ j \leq \text{length } s$

**and mismatch:**  $s!(i-1) \neq s!(j-1)$   
**and greater-checked:**  $\text{f } s \ j \leq i + 1$   
**and strict-border**  $(\text{take } (i-1) \ s) \ (\text{take } (j-1) \ s)$   
**shows**  $\text{f } s \ j \leq \text{f } s \ (i-1) + 1$   
 $\langle \text{proof} \rangle$

**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**  $\text{f } s \ j \leq i + 1$   
**shows**  $\text{f } s \ j = i + 1$   
 $\langle \text{proof} \rangle$

**lemma compute-fs-correct:**  $\text{compute-fs } s \leq \text{compute-fs-SPEC } s$   
 $\langle \text{proof} \rangle$

### 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  $\Rightarrow$  nat list nres where**  
 $\text{compute-butlast-fs } s = \text{do } \{$   
 $\text{let } \text{fs} = \text{replicate } (\text{length } s) \ 0;$   
 $\text{let } i = 0;$   
 $\text{let } j = 1;$   
 $(\text{fs}, -, -) \leftarrow \text{WHILEIT } (I\text{-out-cb } (\text{butlast } s)) \ (\lambda(b, i, j). j < \text{length } b) \ (\lambda(\text{fs}, i, j). \text{do } \{$   
 $\text{ASSERT } (j < \text{length } \text{fs});$   
 $i \leftarrow \text{WHILEIT } (I\text{-in-cb } (\text{butlast } s) \ j) \ (\lambda i. i > 0 \wedge s!(i-1) \neq s!(j-1)) \ (\lambda i. \text{do } \{$   
 $\text{ASSERT } (i-1 < \text{length } \text{fs});$   
 $\text{let } i = \text{fs}!(i-1);$   
 $\text{RETURN } i$   
 $\}) \ i;$   
 $\text{let } i = i + 1;$   
 $\text{ASSERT } (j < \text{length } \text{fs});$   
 $\text{let } \text{fs} = \text{fs}[j := i];$   
 $\text{let } j = j + 1;$   
 $\text{RETURN } (\text{fs}, i, j)$   
 $\}) \ (\text{fs}, i, j);$   
 $\text{RETURN } \text{fs}$   
 $\}$

**lemma compute-fs-inner-bounds:**  
**assumes**  $I\text{-out-cb } s \ (\text{fs}, ix, j)$   
**assumes**  $j < \text{length } \text{fs}$   
**assumes**  $I\text{-in-cb } s \ j \ i$   
**shows**  $i-1 < \text{length } s \ j-1 < \text{length } s$   
 $\langle \text{proof} \rangle$

**lemma** *compute-butlast-fs-refine*[*refine*]:  
**assumes**  $(s,s') \in br\ butlast\ ((\neq)\ \square)$   
**shows**  $compute-butlast-fs\ s \leq \Downarrow Id\ (compute-fs-SPEC\ s')$   
 $\langle proof \rangle$

### 3.4 Conflation

We replace *compute-fs-SPEC* with *compute-butlast-fs*

**definition**  $kmp2\ s\ t \equiv do\ \{$   
 $ASSERT\ (s \neq \square);$   
 $let\ i=0;$   
 $let\ j=0;$   
 $let\ pos=None;$   
 $fs \leftarrow compute-butlast-fs\ s;$   
 $(-,pos) \leftarrow WHILEIT\ (I-outer\ s\ t)\ (\lambda(i,j,pos).\ i + length\ s \leq length\ t \wedge pos=None)$   
 $(\lambda(i,j,pos).\ do\ \{$   
 $ASSERT\ (i + length\ s \leq length\ t \wedge pos=None);$   
 $(j,pos) \leftarrow WHILEIT\ (I-in-na\ s\ t\ i)\ (\lambda(j,pos).\ t!(i+j) = s!j \wedge pos=None)$   
 $(\lambda(j,pos).\ do\ \{$   
 $let\ j=j+1;$   
 $if\ j=length\ s\ then\ RETURN\ (j,Some\ i)\ else\ RETURN\ (j,None)$   
 $\})\ (j,pos);$   
 $if\ pos=None\ then\ do\ \{$   
 $ASSERT\ (j < length\ fs);$   
 $let\ i = i + (j - fs!j + 1);$   
 $let\ j = max\ 0\ (fs!j - 1);$  — *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*:  $kmp2\ s\ t \leq kmp1\ s\ t$   
 $\langle proof \rangle$

**lemma** *kmp2-correct*:  $s \neq \square$   
 $\implies kmp2\ s\ t \leq kmp-SPEC\ s\ t$   
 $\langle proof \rangle$

For convenience, we also remove the precondition:

**definition**  $kmp3\ s\ t \equiv do\ \{$   
 $if\ s=\square\ then\ RETURN\ (Some\ 0)\ else\ kmp2\ s\ t$   
 $\}$

**lemma** *kmp3-correct*:  $kmp3\ s\ t \leq kmp-SPEC\ s\ t$   
 $\langle proof \rangle$

## 4 Refinement to Imperative/HOL

**lemma** *eq-id-param*:  $((=), (=)) \in Id \rightarrow Id \rightarrow Id$  *<proof>*

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

**sepref-definition** *compute-butlast-fs-impl* is *compute-butlast-fs* ::  $(arl-assn\ id-assn)^k \rightarrow_a\ array-assn\ nat-assn$   
*<proof>*

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

**sepref-register** *compute-fs*

**lemma** *kmp-inner-in-bound*:  
**assumes**  $i + length\ s \leq length\ t$   
**assumes**  $I-in-na\ s\ t\ i\ (j, None)$   
**shows**  $i + j < length\ t\ j < length\ s$   
*<proof>*

**sepref-definition** *kmp-impl* is *uncurry kmp3* ::  $(arl-assn\ id-assn)^k *_{a}\ (arl-assn\ id-assn)^k \rightarrow_a\ option-assn\ nat-assn$   
*<proof>*

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

**lemma** *kmp3-correct'*:  
 $(uncurry\ kmp3,\ uncurry\ kmp-SPEC) \in Id \times_r\ Id \rightarrow_f\ (Id)\ nres-rel$   
*<proof>*

**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*:  
 $\langle arl-assn\ id-assn\ s\ si * arl-assn\ id-assn\ t\ ti \rangle$   
 $kmp-impl\ si\ ti$

$\langle \lambda r. \text{arl-assn id-assn } s \text{ si} * \text{arl-assn id-assn } t \text{ ti} * \uparrow(\text{case } r \text{ of } \text{None} \Rightarrow \nexists i. \text{sublist-at } s \text{ t } i$   
 $\quad | \text{Some } i \Rightarrow \text{sublist-at } s \text{ t } i \wedge (\forall ii < i. \neg \text{sublist-at } s \text{ t } ii)$   
 $\quad \rangle >_t$   
 $\langle \text{proof} \rangle$

**definition**  $\text{kmp-string-impl} \equiv \text{kmp-impl} :: (\text{char array} \times \text{nat}) \Rightarrow -$

## 5 Tests of Generated ML-Code

$\langle \text{ML} \rangle$

**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](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](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?fid=1347>, German lecture notes, 2016.