The string search algorithm by Knuth, Morris and Pratt

Fabian Hellauer and Peter Lammich

March 17, 2025

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 *Sepref* tool[3] to obtain executable code in *Imperative/HOL*.

Contents

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* ≡ *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* ⇒ *'a list* ⇒ *nat* ⇒ *bool* **where**
sublist-at (*x*#*xs*) (*y*#*ys*) *0* ⟷ *x*=*y* ∧ *sublist-at xs ys 0* |
sublist-at xs (*y*#*ys*) (*Suc i*) ⟷ *sublist-at xs ys i* |
sublist-at [] *ys 0* ⟷ *True* |
sublist-at - [] - ⟷ *False*

In the relevant cases, both definitions agree:

lemma *i* ≤ *length ys* ⟹ *sublist-at xs ys i* ⟷ *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* ⟹ *i* + *length xs* ≤ *length ys*
⟨*proof*⟩

lemma *Nil-is-sublist*: *sublist-at* ([] :: *'x list*) *ys i* ⟷ *i* ≤ *length ys*
⟨*proof*⟩

Furthermore, we need:

lemma *sublist-step*[*intro*]:
⟦*i* + *length xs* < *length ys*; *sublist-at xs ys i*; *ys*!(*i* + *length xs*) = *x*⟧ ⟹ *sublist-at* (*xs*@[*x*]) *ys i*

⟨*proof*⟩

lemma *all-positions-sublist*:
⟦*i* + *length xs* ≤ *length ys*; ∀ *jj*<*length xs*. *ys*!(*i*+*jj*) = *xs*!*jj*⟧ ⟹ *sublist-at xs ys i*
⟨*proof*⟩

lemma *sublist-all-positions*: *sublist-at xs ys i* ⟹ ∀ *jj*<*length xs*. *ys*!(*i*+*jj*) = *xs*!*jj*
⟨*proof*⟩

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*⟩

corollary *sublist-iff-sublist-at*: *Sublist.sublist xs ys* ⟷ (∃ *i*. *sublist-at xs 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* = *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*)
⟨*proof*⟩

lemma *kmp-result*: *kmp-SPEC s t* =
RETURN (*if sublist s t then Some* (*LEAST i*. *sublist-at s t i*) *else None*)
⟨*proof*⟩

corollary *weak-kmp-SPEC*: *kmp-SPEC s t* ≤ *SPEC* (λ*pos*. *pos*≠*None* ⟷ *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-out-na s t* ≡ 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*. *t*!(*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* ≤ *length t* − *length s* in the first while-condition to *i* + *length s* ≤ *length t* to avoid having to use *int* for list indexes (or the additional precondition *length s* ≤ *length t*)

definition *naive-algorithm s t* ≡ *do* {
  let i=0;
  let j=0;
  let pos=None;
  (-,-,*pos*) ← *WHILEIT* (*I-out-na s t*) (λ(*i*,-,*pos*). *i* + *length s* ≤ *length t* ∧ *pos=None*) (λ(*i*,*j*,*pos*). *do* {
    (-,*pos*) ← *WHILEIT* (*I-in-na s t i*) (λ(*j*,*pos*). *t*!(*i*+*j*) = *s*!*j* ∧ *pos=None*) (λ(*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 $s \neq [] \Longrightarrow$ *naive-algorithm s t* $\leq$ *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 $\mathfrak{f}\ s$ 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*
⟨*proof*⟩
interpretation *border-bot*: *order-bot Nil border strict-border*
⟨*proof*⟩

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

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

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

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

lemma *border-length-le*: *border xs ys* $\Longrightarrow$ *length xs* $\leq$ *length ys*
⟨*proof*⟩

lemma *border-length-r-less* : $\forall$ *xs. strict-border xs ys* $\longrightarrow$ *length xs* $<$ *length ys*
⟨*proof*⟩

lemma *border-positions*: *border xs ys* $\Longrightarrow$ $\forall$ *i*$<$*length xs. ys*!*i* = *ys*!(*length ys* − *length xs* + *i*)
⟨*proof*⟩

lemma *all-positions-drop-length-take*: ⟦*i* $\leq$ *length w*; *i* $\leq$ *length x*;
$\forall$ *j*$<$*i. x* ! *j* = *w* ! (*length w* + *j* − *i*)⟧
$\Longrightarrow$ *drop* (*length w* − *i*) *w* = *take i x*
⟨*proof*⟩

lemma *all-positions-suffix-take*: ⟦*i* $\leq$ *length w*; *i* $\leq$ *length x*;
$\forall$ *j*$<$*i. x* ! *j* = *w* ! (*length w* + *j* − *i*)⟧
$\Longrightarrow$ *suffix* (*take i x*) *w*
⟨*proof*⟩

lemma *suffix-butlast*: *suffix xs ys* $\Longrightarrow$ *suffix* (*butlast xs*) (*butlast ys*)
⟨*proof*⟩

lemma *positions-border*: $\forall$ *j*$<$*l. w*!*j* = *w*!(*length w* − *l* + *j*) $\Longrightarrow$ *border* (*take l w*) *w*
⟨*proof*⟩

lemma *positions-strict-border*: *l* $<$ *length w* $\Longrightarrow$ $\forall$ *j*$<$*l. w*!*j* = *w*!(*length w* − *l* + *j*) $\Longrightarrow$ *strict-border* (*take l w*) *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* $\neq$ [] $\Longrightarrow$ *is-arg-max length* (λ*xs. strict-border xs ys*) (*intrinsic-border ys*)
⟨*proof*⟩

lemma *intrinsic-border-less*: *w* $\neq$ [] $\Longrightarrow$ *length* (*intrinsic-border w*) $<$ *length w*
⟨*proof*⟩

lemma *intrinsic-border-take-less*: *j* $>$ *0* $\Longrightarrow$ *w* $\neq$ [] $\Longrightarrow$ *length* (*intrinsic-border*

(*take j w*)) < *length w*
⟨*proof*⟩

3.1.2 Examples

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

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

corollary *intrinsic-border ''aabaabaa''* = *''aabaa''*
⟨*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 ƒ :: *'a list* ⇒ *nat* ⇒ *nat* **where**
ƒ *s 0* = *0* — This increments the compare position while *j* = *0* |
ƒ *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* ≡ λ(*i*,*j*,*pos*).
(∀ *ii*<*i*. ¬*sublist-at s t ii*) ∧
(*case pos of None* ⇒ (∀ *jj*<*j*. *t*!(*i*+*jj*) = *s*!(*jj*)) ∧ *j* < *length s*
| *Some p* ⇒ *p*=*i* ∧ *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 ƒ directly:

definition *kmp s t* ≡ *do* {
ASSERT (*s* ≠ []);
let i=*0*;
let j=*0*;
let pos=*None*;
(-,-,*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 s);
      let i = i + (j - ∱ s j + 1);
      let j = max 0 (∱ 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 ∱-*eq-0-iff-j-eq-0*[*simp*]: ∱ *s j* = *0* ⟷ *j* = *0*
⟨*proof*⟩

lemma *j-le-*∱*-le*: *j* ≤ *length s* ⟹ ∱ *s j* ≤ *j*
⟨*proof*⟩

lemma *j-le-*∱*-le′*: *0* < *j* ⟹ *j* ≤ *length s* ⟹ ∱ *s j* − *1* < *j*
⟨*proof*⟩

lemma ∱-*le*: *s* ≠ [] ⟹ ∱ *s j* − *1* < *length s*
⟨*proof*⟩

lemma *reuse-matches*:
assumes *j-le*: *j* ≤ *length s*
and *old-matches*: ∀ *jj*<*j*. *t* ! (*i* + *jj*) = *s* ! *jj*
shows ∀ *jj*<∱ *s j* − *1*. *t* ! (*i* + (*j* − ∱ *s j* + *1*) + *jj*) = *s* ! *jj*
(**is** ∀ *jj*<*?j′*. *t* ! (*?i′* + *jj*) = *s* ! *jj*)
⟨*proof*⟩

theorem *shift-safe*:
assumes
∀ *ii*<*i*. ¬*sublist-at s t ii*
t!(*i*+*j*) ≠ *s*!*j* **and**
[*simp*]: *j* < *length s* **and**
matches: ∀ *jj*<*j*. *t*!(*i*+*jj*) = *s*!*jj*
defines
assignment: *i′* ≡ *i* + (*j* − ∱ *s j* + *1*)
shows

$\forall ii{<}i'.\ \neg$*sublist-at s t ii*
⟨*proof*⟩

lemma *kmp-correct*: $s \neq []$
$\Longrightarrow$ *kmp s t* $\leq$ *kmp-SPEC s t*
⟨*proof*⟩

3.2.4 Storing the ƒ-values

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

definition *compute-ƒs-SPEC* :: *'a list* ⇒ *nat list nres* **where**
compute-ƒs-SPEC s ≡ *SPEC* (λƒ*s*. *length* ƒ*s* = *length s* + *1* ∧ (∀ *j*≤*length s*. ƒ*s*!*j* = ƒ *s j*))

definition *kmp1 s t* ≡ *do* {
ASSERT (*s* ≠ []);
let i=0;
let j=0;
let pos=None;
ƒ*s* ← *compute-ƒs-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* ƒ*s*);
let i = *i* + (*j* − ƒ*s*!*j* + *1*);
let j = *max 0* (ƒ*s*!*j* − *1*); — *max* not necessary
RETURN (*i*,*j*,*None*)
} *else RETURN* (*i*,*j*,*Some i*)
}) (*i*,*j*,*pos*);

RETURN pos
}

lemma ƒ-*butlast*[*simp*]: *j* < *length s* ⟹ ƒ (*butlast s*) *j* = ƒ *s j*
⟨*proof*⟩

lemma *kmp1-refine*: *kmp1 s t* ≤ *kmp s t*
⟨*proof*⟩

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

3.3 Computing ƒ

3.3.1 Invariants

definition *I-out-cb s* ≡ λ(ƒ*s*,*i*,*j*).
  length s + 1 = length ƒ*s* ∧
  (∀ *jj*<*j*. ƒ*s*!*jj* = ƒ *s jj*) ∧
  ƒ*s*!(*j*−*1*) = *i* ∧
  0 < *j*
definition *I-in-cb s j* ≡ λ*i*.
  if j=1 then i=0 — first iteration
  else
    strict-border (*take* (*i*−*1*) *s*) (*take* (*j*−*1*) *s*) ∧
    ƒ *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 ƒ*s* ! *1*, we let the first loop-iteration (if there is one) do that for us. This allows us to remove the precondition *s* ≠ [], as the index bounds are respected even in that corner case.

definition *compute-*ƒ*s* :: *'a list* ⇒ *nat list nres* **where**
  *compute-*ƒ*s s* = *do* {
  let ƒ*s*=*replicate* (*length s* + *1*) *0*; — only the first 0 is needed
  let i=0;
  let j=1;
  (ƒ*s*,-,-) ← *WHILEIT* (*I-out-cb s*) (λ(ƒ*s*,-,*j*). *j* < *length* ƒ*s*) (λ(ƒ*s*,*i*,*j*). *do* {
    i ← *WHILEIT* (*I-in-cb s j*) (λ*i*. *i*>*0* ∧ *s*!(*i*−*1*) ≠ *s*!(*j*−*1*)) (λ*i*. *do* {
      ASSERT (*i*−*1* < *length* ƒ*s*);
      let i=ƒ*s*!(*i*−*1*);
      RETURN i
    }) *i*;
    let i=i+1;
    ASSERT (*j* < *length* ƒ*s*);
    let ƒ*s*=ƒ*s*[*j*:=*i*];
    let j=j+1;
    RETURN (ƒ*s*,*i*,*j*)
  }) (ƒ*s*,*i*,*j*);

  RETURN ƒ*s*
}

3.3.3 Correctness

lemma *take-length-ib*[*simp*]:
  assumes *0* < *j j* ≤ *length s*
    shows *take* (*length* (*intrinsic-border* (*take j s*))) *s* = *intrinsic-border* (*take j s*)
⟨*proof*⟩

lemma *ib-singleton*[*simp*]: *intrinsic-border* [*z*] = []
⟨*proof*⟩

lemma *border-butlast*: *border xs ys* ⟹ *border* (*butlast xs*) (*butlast ys*)
⟨*proof*⟩

corollary *strict-border-butlast*: *xs* ≠ [] ⟹ *strict-border xs ys* ⟹ *strict-border* (*butlast xs*) (*butlast ys*)
⟨*proof*⟩

lemma *border-take-lengths*: *i* ≤ *length s* ⟹ *border* (*take i s*) (*take j s*) ⟹ *i* ≤ *j*
⟨*proof*⟩

lemma *border-step*: *border xs ys* ⟷ *border* (*xs*@[*ys*!*length xs*]) (*ys*@[*ys*!*length xs*])
⟨*proof*⟩

corollary *strict-border-step*: *strict-border xs ys* ⟷ *strict-border* (*xs*@[*ys*!*length xs*]) (*ys*@[*ys*!*length xs*])
⟨*proof*⟩

lemma *ib-butlast*: *length w* ≥ *2* ⟹ *length* (*intrinsic-border w*) ≤ *length* (*intrinsic-border* (*butlast w*)) + *1*
⟨*proof*⟩

corollary ϱ-*Suc*: *Suc i* ≤ *length w* ⟹ ϱ *w* (*Suc i*) ≤ ϱ *w i* + *1*
⟨*proof*⟩

lemma ϱ-*step-bound*:
assumes *j* ≤ *length w*
shows ϱ *w j* ≤ ϱ *w* (*j*−*1*) + *1*
⟨*proof*⟩

lemma *border-take*-ϱ: *border* (*take* (ϱ *s i* − *1*) *s*) (*take i s*)
⟨*proof*⟩

corollary ϱ-*strict-borderI*: *y* = ϱ *s* (*i*−*1*) ⟹ *strict-border* (*take* (*i*−*1*) *s*) (*take* (*j*−*1*) *s*) ⟹ *strict-border* (*take* (*y*−*1*) *s*) (*take* (*j*−*1*) *s*)
⟨*proof*⟩

corollary *strict-border-take*-ϱ: *0* < *i* ⟹ *i* ≤ *length s* ⟹ *strict-border* (*take* (ϱ *s i* − *1*) *s*) (*take i s*)
⟨*proof*⟩

lemma ϱ-*is-max*: *j* ≤ *length s* ⟹ *strict-border b* (*take j s*) ⟹ ϱ *s j* ≥ *length b* + *1*
⟨*proof*⟩

theorem *skipping-ok*:
assumes *j-bounds*[*simp*]: *1* < *j j* ≤ *length s*

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

lemma *extend-border*:
assumes $j \leq$ *length* s
assumes $s!(i-1) = s!(j-1)$
assumes *strict-border* (*take* $(i-1)$ s) (*take* $(j-1)$ s)
assumes $\mathfrak{f}\ s\ j \leq i + 1$
shows $\mathfrak{f}\ s\ j = i + 1$
⟨*proof*⟩

lemma *compute-*$\mathfrak{f}s$*-correct*: *compute-*$\mathfrak{f}s$ $s \leq$ *compute-*$\mathfrak{f}s$*-SPEC* s
⟨*proof*⟩

3.3.4 Index shift

To avoid inefficiencies, we refine *compute-*$\mathfrak{f}s$ to take s instead of *butlast* s (it still only uses *butlast* s).

definition *compute-butlast-*$\mathfrak{f}s$ :: $'a$ *list* ⇒ *nat list nres* **where**
compute-butlast-$\mathfrak{f}s$ s = *do* {
let $\mathfrak{f}s$=*replicate* (*length* s) *0*;
let i=*0*;
let j=*1*;
($\mathfrak{f}s$,-,-) ← *WHILEIT* (*I-out-cb* (*butlast* s)) (λ(b,i,j). $j <$ *length* b) (λ($\mathfrak{f}s$,i,j). *do* {
ASSERT ($j <$ *length* $\mathfrak{f}s$);
i ← *WHILEIT* (*I-in-cb* (*butlast* s) j) (λi. $i>0 \wedge s!(i-1) \neq s!(j-1)$) (λ$i$. *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 $\mathfrak{f}s$
}

lemma *compute-*$\mathfrak{f}s$*-inner-bounds*:
assumes *I-out-cb* s ($\mathfrak{f}s$,ix,j)
assumes $j <$ *length* $\mathfrak{f}s$
assumes *I-in-cb* s j i
shows $i-1 <$ *length* s $j-1 <$ *length* s
⟨*proof*⟩

lemma *compute-butlast-ʃs-refine*[*refine*]:
  assumes $(s,s') \in br\ butlast\ ((\neq)\ [])$
  shows *compute-butlast-ʃs* $s \leq \Downarrow\ Id\ (compute\text{-}ʃs\text{-}SPEC\ s')$
⟨*proof*⟩

3.4 Conflation

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

definition *kmp2 s t* ≡ *do* {
  ASSERT (*s* ≠ []);
  let i=0;
  let j=0;
  let pos=None;
  ʃs ← *compute-butlast-ʃs s*;
 (-,-,*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* ∧ *pos=None*);
     (*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 ʃs*);
      let i = *i* + (*j* − *ʃs*!*j* + *1*);
      let j = *max 0* (*ʃs*!*j* − *1*); — *max* not necessary
      RETURN (*i*,*j*,*None*)
    } *else RETURN* (*i*,*j*,*Some i*)
  }) (*i*,*j*,*pos*);

  RETURN pos
}

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

lemma *kmp2-refine*: *kmp2 s t* ≤ *kmp1 s t*
  ⟨*proof*⟩

lemma *kmp2-correct*: *s* ≠ []
  ⟹ *kmp2 s t* ≤ *kmp-SPEC s t*
⟨*proof*⟩

For convenience, we also remove the precondition:

definition *kmp3 s t* ≡ *do* {
  if s=[] *then RETURN* (*Some 0*) *else kmp2 s t*
}

lemma *kmp3-correct*: *kmp3 s t* ≤ *kmp-SPEC s t*
  ⟨*proof*⟩

4 Refinement to Imperative/HOL

lemma *eq-id-param*: $((=), (=)) \in Id \to Id \to Id$ ⟨*proof*⟩

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

sepref-definition *compute-butlast-∫s-impl* **is** *compute-butlast-∫s* :: $(arl\text{-}assn\ id\text{-}assn)^k$ $\to_a$ *array-assn nat-assn*
⟨*proof*⟩

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

sepref-register *compute-∫s*

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* :: $(arl\text{-}assn\ id\text{-}assn)^k *_a (arl\text{-}assn\ id\text{-}assn)^k \to_a option\text{-}assn\ nat\text{-}assn$
⟨*proof*⟩

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

lemma *kmp3-correct′*:
(*uncurry kmp3*, *uncurry kmp-SPEC*) ∈ $Id \times_r Id \to_f \langle Id\rangle nres\text{-}rel$
⟨*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*:
< *arl-assn id-assn s si* ∗ *arl-assn id-assn t ti* >
kmp-impl si ti

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

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

5 Tests of Generated ML-Code

⟨*ML*⟩

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?fid=1347, German lecture notes, 2016.