Sequence and string types

Contents:

What is a sequence?

A sequence is an ordered collection of objects that allows multiple occurrences of the same object. As with sets, the objects are known as elements of the sequence.

Examples:

(1) [5, 15, 15, 5, 35]

(2) ['u', 'n', 'i', 'v', 'e', 'r', 's', 'i', 't', 'y']

(3) [20.5, 40.5, 85.5]

The “string” type

A string is a special sequence in the sense that its all elements are only characters. In other words, a string is a sequence of characters.

We use the keyword string to denote the type that contains all the possible strings.

Examples of strings:

"university"
"sofl@yahoo.ac.jp"
"Formal Engineering Methods"

Sequence type declarations

A sequence type A is declared based on an element type T in the following format:

A = **seq of** T

Example:

Ages = seq of nat

Then, we can declare a variable of type Ages:

student_ages: Ages

Constructors and operators on sequences

A sequence can be created using either sequence constructors or operators.

1. Constructors

There are two constructors: sequence enumeration and sequence comprehension.

(2.1) Sequence enumeration

A sequence enumeration has the format:

[a_1, a_2, ..., a_n]

where a_i (i=1..n) are the elements of the sequence.

Example:

[5, 9, 8, 9, 5]

The order and the occurrences of the elements are significant. Thus:

[5, 9] <> [9, 5] and

[5, 9, 5] <> [5, 9]

(2.2) Sequence comprehension

A sequence comprehension takes the format:

$[e(x\_1, x\_2, ..., x\_n) \mid x\_1: T\_1, x\_2: T\_2, ..., x\_n: T\_n \;\&\; P(x\_1, x\_2, ..., x\_n)]$

The sequence comprehension defines a sequence whose elements are derived from the evaluation of expression $e(x\_1, x\_2, ..., x\_n)$ under the condition that x_1 takes values from type T_1, x_2 from T_2, ..., x_n from T_n, and all of these values satisfy property $P(x\_1, x\_2, ..., x\_n)$.

Note that all the types T_i $(i = 1,...,n)$ are countable numeric types and the elements of the sequence must occur in the ascending order. For example,

$[i * j \mid i: nat, j: nat \;\&\; 1 <= i + j <= 3] = [1, 2, 2]$

As with the set notation, we also use the following special notation to represent a sequence of integer interval from i to j:

[i, ..., j] = [x | x: int & i <= x <= j]

Thus:

[3, ..., 6] = [3, 4, 5, 6]
[-2, ..., 2] = [-2, -1, 0, 1, 2]
[0, ..., 4] = [0, 1, 2, 3, 4]

However, if index j is smaller than i, [i, ..., j] will represents the empty sequence [ ]. For example,

[9, ..., 2] = [ ]

2. Operators

All the operators are applicable to variables of string type as well.

2.1 Length (len)

The length of a sequence means the number of its elements.

len: seq of T --> nat0

len(s) == the number of its elements.

Examples: let s1 = [4, 9, 10], s2 = [{3, 9}, {6}],
s3 = [10, 9, 4, 25], and s4 = "university".

Then:

len(s1) = 3

len(s2) = 2

len(s3) = 4

len(s4) = 10

2.2 Sequence application

A sequence can apply to an index, a natural number, to yield the element occurring at the position indicated by the index.

Let s be a sequence of type seq of T. Then, s can be regarded as a function from nat to T:

seq of T * nat --> T
s(i) == the ith element of sequence s

The precondition for applying s to an index i (i.e., s(i)) is that index i is within the range of 1 to len(s). Otherwise, if i is beyond this range, the sequence application s(i) is undefined.

Examples: let s1 = [4, 9, 10], s2 = [{3, 9}, {6}],

s3 = [10, 9, 4, 25], and s4 = "university".

Then:

s1(1) = 4

s1(2) = 9

s2(1) = {3, 9}

s3(4) = 25

s4(5) = 'e'

2.3 Subsequence

A subsequence of a sequence is part of the sequence.

Let s be a sequence of type seq of T, and i and j are two indexes. Then the subsequence of s that keeps the elements in the same order as they are in s is denoted as:

```
seq of T * nat * nat --> seq of T
s(i, j) == [s(i), s(i + 1), ..., s(j - 1), s(j)]
```

Examples:

```
s1(2, 3) = [9, 10]     s1(1, 3) = s1
s3(2, 4) = [9, 4, 25]     s4(2, 8) = "niversi"
```

2.4 Head (hd)

The head of a non-empty sequence is its first element.

```
hd(s: seq of T) he: T
pre   s <> []
post  he = s(1)
```

If s is the empty sequence, hd(s) is undefined.

For example, let s1 = [4, 9, 10], s2 = [{3, 9}, {6}],
s3 = [10, 9, 4, 25], and s4 = "university".

hd(s1) = 4,   hd(s2) = {3, 9},   hd(s3) = 10,
hd(s4) = 'u'

2.5 Tail (tl)

The tail of a non-empty sequence is its subsequence resulting from eliminating its head.

tl(s: seq of T) ts: seq of T
pre s <> []
post ts = s(2, len(s))

The application of the operator tl to the empty sequence is undefined, that is, tl([]) = nil.

For example: let s1 = [4, 9, 10], s2 = [{3, 9}, {6}],
s3 = [10, 9, 4, 25], and s4 = "university".

tl(s1) = [9, 10], tl(s2) = [{6}],
tl(s3) = [9, 4, 25], tl(s4) = "niversity"

2.6 Elements (elems)

The operator for obtaining the set of all the elements of a sequence is elems that is defined as:

```
elems: seq of T --> set of T
elems(s) == {x | x: T &
                  (exists[i: {1, ..., len(s)}] | x = s(i)})
```

Since the result of elems(s) is a set, not a sequence, no duplication of elements of s is allowed.

For example, let s1 = [4, 9, 10], s2 = [{3, 9}, {6}],
s3 = [10, 9, 4, 25], and s4 = "university".

Then,

elems(s1) = {4, 9, 10}   elems(s2) = {{3, 9}, {6}}
elems(s3) = {10, 9, 4, 25}
elems([5, 10, 5, 10, 15]) = {5, 10, 15}
elems(s4) = {'u', 'n', 'i', 'v', 'e', 'r', 's', 't', 'y'}
elems([ ]) = { }.

2.7 Indexes (inds)

A sequence corresponds to a set of natural numbers that indicates the positions of the elements of the sequence. Such a set is known as index set.

```
inds: seq of T --> set of nat
inds(s) == {i | i: nat & exists[ x: elems(s)] | s(i) = x}
```

It is obvious that the index set of the empty sequence is the empty set.

Furthermore, the cardinality of inds(s) is equal to the length of sequence s, but may be greater than the number of the elements of set elems(s) due to the possibility of having duplicated elements in s.

For example, let s1 = [4, 9, 10],
s2 = [{3, 9}, {6}],
s3 = [10, 9, 4, 25],
s4 = "university".

Then:

inds(s1) = {1, 2, 3}
inds(s2) = {1, 2}
inds(s3) = {1, 2, 3, 4}
inds(s4) = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

The index set is often used when describing a property of a sequence. Consider the example:

exists[i: inds(s)] | s(i) > 5

This quantified expression describes a property of sequence s, requiring that s has at least one element greater than 5.

2.8 Concatenation (conc)

Sequences can be concatenated to form another sequence.

```
conc(s_1: seq of T, s_2: seq of T) cs: seq of T
post (forall[i: inds(s_1)] | cs(i) = s_1(i)) and
        (forall[j: inds(s_2)] | cs(j + len(s_1)) = s_2(j)) and
        len(cs) = len(s_1) + len(s_2)
```

The concatenation of sequences s_1 and s_2 is formed by appending s_2 to the end of s_1.

Examples:

conc(s1, s3) = [4, 9, 10, 10, 9, 4, 25]

conc(s4, s4) = "universityuniversity"

The concatenation of sequences is not commutative. Thus:

conc(s1, s3) <> conc(s3, s1)

The concatenation operator conc can be extended to deal with more than two sequences. Thus:

conc(s_1, s_2, ..., s_n) = conc(s_1, conc(s_2, conc(s_3, ...)))

For example, let s1 = [5, 15, 25],

s2 = [10, 20, 30, 40],

s3 = [2, 4, 6, 8, 10].

Then:

conc(s1, s2, s3) = [5, 15, 25,

10, 20, 30, 40,

2, 4, 6, 8, 10]

2.9 Distributed concatenation (dconc)

Let S be a sequence of sequences:

S = [s_1, s_2, ..., s_n]

where each s_i (i =1,…,n) is a sequence.

dconc: seq of seq of T --> seq of T

dconc(S) == conc(s_1, s_2, ..., s_n)

Example: let S1 = [[5, 15, 25], [10, 20, 30, 40],
[2, 4, 6, 8, 10]]

Then

dconc(S1) = [5, 15, 25, 10, 20, 30, 40,
2, 4, 6, 8, 10]

2.10 Equality and inequality (= and <>)

Sequences can be compared to determine whether they are identical or not.

s_1 = s_2 <=> (len(s_1) = len(s_2) and
forall[i: inds(s_1)] | s_1(i) = s_2(i))

s_1 <> s_2 <=> not s_1 = s_2

Examples: let s1 = [5, 15, 25], s2 = [10, 20, 30, 40],
s3 = [2, 4, 6, 8, 10].

Then:

s1 = s1 <=> true   s1 <> s2 <=> true
s2 = s3 <=> false

Specifications using sequences

1. Sorting of an integer sequence

Example: ~list = [3, 5, 8, 5, 9, 3, 10, 5]
list = [3, 3, 5, 5, 5, 8, 9,10]

```
module SortingOfIntegerSequence;
 var
   list: seq of int;
 process Sort()
 ext wr list: seq of int
 pre   true
 post  Is_Permutation(~list, list) and Is_Ordered(list)
 comment
 After the sorting, the final list must maintain the same
 elements including their all occurrences and keep their
 elements in the ascending order.
 end_process;
```

```
function Is_Permutation(l1, l2: seq of int): bool

== forall[e: union(elems(l1), elems(l2))] |

          card({i | i: inds(l1) & l1(i) = e}) =

          card({i | i: inds(l2) & l2(i) = e})

end_function;



function Is_Ordered(l: seq of int): bool
== forall[i,j: inds(l)] | i < j => l(i) <= l(j)
end_function;
```

The characteristic of this specification is that it says nothing about how to sort the integer sequence list, but focuses on the relation between the initial list (i.e., ~list) and the final list.

2. Membership Management System

```
module MembershipManagementSystem;
  type
   Member = string;  /* A member is denoted by its name
                         which is a string of characters */
  var
    all_members: seq of Member;

  process Register(m: Member)
  ext wr all_members
  post  all_members = conc(~all_members, [m])
  comment
    The function of recording the member m in the member list
  all_members is specified by defining the all_members after
  the process as the concatenation of the all_members before
  the process and the sequence composed of member m.
  end_process;
```

```
process Search(m: Member) pos: set of nat
ext rd all_members
post  pos = {i | i: nat & all_members(i) = m}
comment
  Finding all the positions of member m in the
  member list all_members is modeled by a set
  comprehension.
end_process;
```

```
process Exchange(pos1, pos2: nat)
ext wr all_members
pre pos1 inset inds(all_members) and pos2
      inset inds(all_members)
post  all_members(pos1) = ~all_members(pos2) and
        all_members(pos2) = ~all_members(pos1) and
        forall[i | i: inds(all_members)] &
                    i <> pos1 and i <> pos2 =>
                     all_members(i) = ~all_members(i)
comment    This process only exchanges the
members at position pos1 and pos2, and keeps the
rest of the members unchanged in the list.
end_process;
end_module.
```

Class exercise 6

1. Given a set T = {1, 2, 5}, declare a sequence type based on T, and list up to 5 possible sequence values in the type.

2. Evaluate the sequence comprehensions:

   a. [x | x: nat & 3 < x < 8]

   b. [y | y: nat0 & y <= 3]

   c. [x * x | x: nat, y: nat & 1 <= x <= 3]

3. Let s1 = [5, 15, 25], s2 = [15, 30, 50], s3 = [30, 2, 8], and s = [s1, s2, s3].
   Evaluate the expressions:

   a. hd(s1)

   b. hd(s)

   c. len(tl(s1)) + len(tl(s2)) + len(tl(s3))

   d. len(s1) + len(s2) - len(s3)

   e. union(elems(s1), elems(s2))

   f. inter(union({hd(s2)}, elems(s3)), elems(s1))

   g. union(inds(s1), inds(s2), inds(s3))

   h. elems(conc(s1, s2, s3))

   i. dconc(s)

4. Construct a module to model a queue of integers with the processes: Append, Eliminate, Read, and Count. The process Append adds a new element to the queue; Eliminate deletes the top element of the queue; Read tells what is the top element; and Count yields the number of the elements in the queue.