Atelier B

Reusable Components

Reference Manual

version 3.6
## Contents

1 Introduction 1

2 Index of Basic Machines 3

3 Index of Library Machines 5

4 Description of Basic Machines 13

4.1 BASIC_ARRAY_VAR: Implanting a one dimensional table 14

4.2 BASIC_ARRAY_RGE: Implementing a Two Dimensional Array 17

4.3 BASIC_IO: vt100 style inputs/outputs 20

5 Description of Library Machines 25

5.1 L_ARITHMETIC1: Extended Integer Operations 26

5.2 L_ARRAY1: One Dimensional Array, with Initialization Loop 29

5.3 L_ARRAY3: Array with Non Ordered Values, Maximum Operations 31

5.4 L_ARRAY5: Array with Ordered Values, Sort Operation 34

5.5 L_PFNC: Partial Function 37

5.6 L_SEQUENCE: Creating a Sequence 41

5.7 L_SET: Creating a Set 45

5.8 L_ARRAY1_RANGE: A Range of Arrays of the Same Size, with Numerical
Indexes 48

5.9 L_ARRAY3_RANGE: A Range of Arrays of the Same Size, with Non Ordered
Values, Maximum Operations 51

5.10 L_ARRAY5_RANGE: Range of Arrays of the Same Size, with Ordered
Value Numerical Indexes, Sort Operation 56

5.11 L_SEQUENCE_RANGE: Range of Sequences 61

5.12 L_ARRAY_COLLECTION: collection of arrays of the same size 67

5.13 L_ARRAY1_COLLECTION: array of the same size, with numerical indexes 69
1 Introduction

The reusable components supplied with Atelier B are basic machines and library machines. Basic machines are the modelisation in B of modules manually coded in C, C++ or ADA. These modules are used to encapsulate the operating system functions that must be used; they must usually be performed in taking into account the specificities of the hardware that the security software will run on. This is why there are few basic machines delivered with "Atelier B".

Library machines are abstract machines written in B language. They generally model a type of mathematical object (sequence, function, etc.) and offer the operations that allow the handling of these objects.

Unlike basic machines, library machines are properly performed using the B method, i.e., using refining and implementation in B along with complete proof of the set. This proof may in principle be executed at any time in order to check its validity (warning: proving methods may depend on the demonstrator version used). Therefore, unlike basic machines, library machines may be numerous and complex while remaining secure as they are proven.

To use basic machines, simply reference them in the appropriate B project, by INCLUDES, IMPORTS or any derived actions. When the final project is translated into a traditional programming language, the translation of the library machine implementations used must be redone if this was not already done at Atelier B installation.

Library machines are implemented on basic machines. As they are performed until the implementation in B language, they provide complete examples of use in the B method. They especially contain examples of proven WHILE loops. For practical advice on proving WHILE loops, refer to the "B Language User Manual".

The user may directly use library machines just like he uses basic machines. Sometimes the implementation of a library machine may use the services of a machine that it does not create an instance for (use by SEES) to avoid duplications. In this case the user will have to create the instance in question (using IMPORTS) by following the indications in the "IMPORTS REQUIRED" section of the description for each library machine.

When the final project C, C++ or ADA compilation is performed, the library compilation is automatically performed if necessary. Performing link editing will then enable incorporation into the final executable program only those object files that correspond to the library machines actually used. All this is performed in the Makefile produced by Atelier B. To integrate a software component produced by Atelier B into a traditional product, use this Makefile as a basis or refer to the “ADA Translator User Manual”.

Warning:
This warning regards the use of reusable components with the Ada, C and C++ translators
supplied with Atelier B. These translators are experimentals. Their goal is to show that it is possible to translate some B0 implementations into classical programming languages. Therefore, their use is not guaranteed. Especially the reusable components use may induce errors when compiling the code produced by the translators. The reusable components must be considered as examples. Each user can develop his own library machines according to his needs.
2 Index of Basic Machines

**BASIC_ARRAY_VAR**: implanting a one dimension table
- VAL_ARRAY: read a table element
- STR_ARRAY: write a table element

**BASIC_ARRAY_RGE**: implementing a two dimensional table
- VAL.ARR_RGE: read a table element
- STR.ARR_RGE: write a table element
- COP.ARR_RGE: copy a table line to another
- CMP.ARR_RGE: compare two table lines

**BASIC_IO**: vt 100 style input/output
- INTERVAL_READ: entry by the operator of a number in mm..nn.
- INT_WRITE: print a number.
- BOOL_READ: entry by a TRUE or FALSE boolean operator
- BOOL_WRITE: print the TRUE or FALSE condition.
- CHAR_READ: entry by a character’s operator.
- CHAR_WRITE: print a character.
- STRING_WRITE: print a message.
3 Index of Library Machines

**L_ARITHMETIC1**: extended integer operations: MIN, MAX, INC, DEC, EXP, SQRT, LOG

- VAL.ARR.RGE read a table element
- STR.ARR.RGE write a table element
- COP.ARR.RGE copy a table line to another
- CMP.ARR.RGE compare two table lines

**BASIC_IO**: vt 100 style input/output

- MIN minimum of two numbers.
- MAX maximum of two numbers.
- INC increment a number.
- DEC decrement a number.
- EXP exponential.
- SQRT integer square root by default.
- LOG_BY_DEFAULT logarithm by default.
- LOG_BY_EXCESS logarithm by excess.

**L_ARRAY1**: one dimensional table with initialization loop

- VAL.ARRAY value of an element (promoted operation)
- STR.ARRAY write an element (promoted operation)
- SET.ARRAY write the same value in a portion of the table

**L_ARRAY3**: table with non-ordered values, maximum operations

- VAL.ARRAY value of an element (promoted operation).
- STR.ARRAY write an element (promoted operation).
- SET.ARRAY write a same value in a table portion (promoted operation).
- SWAP.ARRAY exchange two elements (promoted operation).
- RIGHT_SHIFT.ARRAY shift a portion to the large index (promoted operation).
LEFT_SHIFT_ARRAY shift a portion to the small index (promoted operation).
SEARCH_MAX_EQL_ARRAY search for a value in a portion of the table (promoted operation).
SEARCH_MIN_EQL_ARRAY search for a value in a portion of the table (promoted operation).
REVERSE_ARRAY invert the order of the elements in a portion of the table.

L_ARRAY5: table with ordered values, sort operation

VAL_ARRAY value of an element (promoted operation).
STR_ARRAY write an element (promoted operation).
SET_ARRAY write the same value in a portion of the table (promoted operation).
SWAP_ARRAY exchange two elements (promoted operation).
RIGHT_SHIFT_ARRAY shift a portion to the large index (promoted operation).
LEFT_SHIFT_ARRAY shift a portion to the small index (promoted operation).
SEARCH_MAX_EQL_ARRAY search for a value in a portion of the table (promoted operation).
SEARCH_MIN_EQL_ARRAY search for a value in a portion of the table (promoted operation).
REVERSE_ARRAY invert the order of elements in a portion of the table (promoted operation).
SEARCH_MIN_GEQ_ARRAY search for the first element that exceeds a value (promoted operation).
ASCENDING_SORT_ARRAY sort of a table portion.

L_PFNC: partial function

VAL_PFNC value of the function for an element in its domain
STR_PFNC overloads the partial function with a couple
XST_PFNC tests if an index is in the partial function domain
RMV_PFNC removes a couple from the partial function
SET_PFNC overloads a part of the function with a constant
SWAP_PFNC exchanges the images for two domain indexes
RIGHT_SHIFT_PFNC right shift of a domain part
LEFT_SHIFT_PFNC left shift of a domain part
SEARCH_MAX_EQL_PFNC searches for a value in the partial function
SEARCH_MIN_EQL_PFNC searches for a value in the partial function
REVERSE_PFNC reverses the order of elements for a portion of the domain
ASCENDING_SORT_PFNC sorts in a portion of the domain

**L_SEQUENCE: building a sequence**

LEN_SEQ    returns the current size of the sequence.
IS_FULL_SEQ is used to determine if the sequence is full (size = LS_maxsize).
IS_INDEX_SEQ is used to determine whether ii is a valid index.
VAL_SEQ    value of an element in the sequence.
FIRST_SEQ  returns the first element in the sequence.
LAST_SEQ   returns the last element in the sequence.
PUSH_SEQ   add vv to the end of the sequence.
POP_SEQ    removes the last element from the sequence (its value is lost).
STR_SEQ    changes the value of an element in the sequence.
RMV_SEQ    removes an element from the middle of the sequence.
INS_AFT_SEQ inserts vv right after index ii.
CLR_SEQ    clears the sequence.
TAIL_SEQ   removes the first element from the sequence.
KEEP_SEQ   only keeps the first elements in the sequence.
CUT_SEQ    cuts the nn first elements from the sequence.
PART_SEQ   only retains part ii..jj in the sequence.
REV_SEQ    reverses the order of elements in the sequence.
FIND_FIRST_SEQ finds vv in the sequence, from the start.
FIND_LAST_SEQ finds vv in the sequence, from the end.

**L_SET: creating a set**

CARD_SET    returns the cardinal for the set.
IS_FULL_SET identifies if the set is full (card = LSET_maxsize).
IS_INDEX_SET identifies if a number is a valid index.
VAL_SET    value of a element in the set.
FIND_SET    finds an element in the set.
RMV_SET    removes an element from the set.
INS_SET    inserts an element in the set.
CLR_SET    clears all elements from the set.

**L_ARRAY_1_RANGE: array of tables of the same size with numerical indexes**

VAL_ARR_RGE value of an element (promoted operation).
**Reusable Components—Reference Manual**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR_ARR_RGE</td>
<td>write an element (promoted operation).</td>
</tr>
<tr>
<td>COP_ARR_RGE</td>
<td>copy a table to another (promoted operation).</td>
</tr>
<tr>
<td>CMP_ARR_RGE</td>
<td>compare two tables (promoted operation).</td>
</tr>
<tr>
<td>DUP_ARR_RGE</td>
<td>duplicate the same table into a series of tables.</td>
</tr>
<tr>
<td>SET_ARR_RGE</td>
<td>copy the same value to an index set in one of the tables.</td>
</tr>
<tr>
<td>PCOP_ARR_RGE</td>
<td>copy part of one of the tables to a different table to a given position.</td>
</tr>
<tr>
<td>PCMP_ARR_RGE</td>
<td>find the first element that is different from two parts of two tables. A</td>
</tr>
<tr>
<td></td>
<td>Boolean element indicates if this element was found and, in this case, the</td>
</tr>
<tr>
<td></td>
<td>index of this element is returned.</td>
</tr>
<tr>
<td>SWAP_RGE</td>
<td>swap two elements in a table.</td>
</tr>
<tr>
<td>RIGHT_SHIFT_RGE</td>
<td>shift a table range to the large index.</td>
</tr>
<tr>
<td>LEFT_SHIFT_RGE</td>
<td>shift a table range to the small index.</td>
</tr>
<tr>
<td>SEARCH_MAX_EQL_RGE</td>
<td>find the last element that equals a value in a table range.</td>
</tr>
<tr>
<td>SEARCH_MIN_EQL_RGE</td>
<td>find the first element that equals a value in a table range.</td>
</tr>
<tr>
<td>REVERSE_RGE</td>
<td>reverse the order of the elements of a table part.</td>
</tr>
</tbody>
</table>

**L_ARRAY_3_RANGE**: range of tables of the same size, with numerical indexes, and values that are not ordered, maximum operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAL_ARR_RGE</td>
<td>value of an element (promoted operation).</td>
</tr>
<tr>
<td>STR_ARR_RGE</td>
<td>write an element (promoted operation).</td>
</tr>
<tr>
<td>COP_ARR_RGE</td>
<td>copy a table to another (promoted operation).</td>
</tr>
<tr>
<td>CMP_ARR_RGE</td>
<td>compare two tables (promoted operation).</td>
</tr>
<tr>
<td>DUP_ARR_RGE</td>
<td>duplicate the same table to an array of tables (promoted operation).</td>
</tr>
<tr>
<td>SET_ARR_RGE</td>
<td>copy the same value to a range in one of the tables (promoted operation).</td>
</tr>
<tr>
<td>PCOP_ARR_RGE</td>
<td>copy part of one of the tables to a different table, in a given position</td>
</tr>
<tr>
<td></td>
<td>(promoted operation).</td>
</tr>
<tr>
<td>PCMP_ARR_RGE</td>
<td>find the first different element from two parts in two tables. A Boolean</td>
</tr>
<tr>
<td></td>
<td>element indicates whether this element was found and, in this case, the</td>
</tr>
<tr>
<td></td>
<td>index of this element is returned (promoted operation).</td>
</tr>
<tr>
<td>SWAP_RGE</td>
<td>swap two elements in a table.</td>
</tr>
<tr>
<td>RIGHT_SHIFT_RGE</td>
<td>shift a table range to the large index.</td>
</tr>
<tr>
<td>LEFT_SHIFT_RGE</td>
<td>shift a table range to the small index.</td>
</tr>
<tr>
<td>SEARCH_MAX_EQL_RGE</td>
<td>find the last element that equals a value in a table range.</td>
</tr>
<tr>
<td>SEARCH_MIN_EQL_RGE</td>
<td>find the first element that equals a value in a table range.</td>
</tr>
<tr>
<td>REVERSE_RGE</td>
<td>reverse the order of the elements of a table part.</td>
</tr>
</tbody>
</table>

**L_ARRAY_5_RANGE**: array of tables of the same size, with numerical indexes, with ordered values, sort operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAL_ARR_RGE</td>
<td>value of an element (promoted operation).</td>
</tr>
<tr>
<td>STR_ARR_RGE</td>
<td>write an element (promoted operation).</td>
</tr>
</tbody>
</table>
COP_ARR_RGE  copy a table to another (promoted operation).
CMP_ARR_RGE  compare two tables (promoted operation).
DUP_ARR_RGE  duplicate the same table in a range of tables (promoted operation).
SET_ARR_RGE  copy the same value to an index range in one of the arrays (promoted operation).
PCOP_ARR_RGE copy a range from one of the tables to a different table, at a given position (promoted operation).
PCMP_ARR_RGE find the first different element in two ranges in two tables. A Boolean element indicates that this element was found and, in this case, the index of this element is returned (promoted operation).
SWAP_RGE  swap two elements in a table (promoted operation).
RIGHT_SHIFT_RGE shift a table range to the large index (promoted operation).
LEFT_SHIFT_RGE shift a table range to the small index (promoted operation).
SEARCH_MAX_EQL_RGE search for the last element that equals a value in a table range (promoted operation).
SEARCH_MIN_EQL_RGE search for the first element that equals a value in a table range (promoted operation).
REVERSE_RGE reverses the order of the elements of a part of a table (promoted operation).
SEARCH_MIN_GEQ_RGE search for the first element that exceeds a value in a table range.
ASCENDING_SORT_RGE sort a table range into ascending order.

L_SEQUENCE_RANGE: sequence range

LEN_SEQ_RGE  determines the length of a sequence.
IS_FULL_SEQ_RGE determines whether a sequence is full.
IS_INDEX_SEQ_RGE determines whether an integer is in a sequence range.
VAL_SEQ_RGE  gives the value of a sequence for a valid index.
FIRST_SEQ_RGE gives the first element in a sequence.
LAST_SEQ_RGE  gives the last element in a sequence.
PUSH_SEQ_RGE adds an element to a sequence.
POP_SEQ_RGE  removes the last element from a sequence.
STR_SEQ_RGE  changes the value of a sequence element.
RMV_SEQ_RGE removes an element from a sequence, with a size that decreases by 1.
INS_SEQ_RGE adds an element to a sequence, with a size that increases by 1.
CLR_SEQ_RGE  clears a sequence.
TAIL_SEQ_RGE removes the first element from a sequence.
KEEP_SEQ_RGE      only keeps in a sequence the N first elements.
CUT_SEQ_RGE      cuts the N first elements from a sequence.
PART_SEQ_RGE      only keeps in a sequence the indexes in a range between two limits.
REV_SEQ_RGE      reverses the order of the elements in a sequence.
FIND_FIRST_SEQ_RGE finds a value in a sequence, returns a Boolean element indicating that it was found and if yes returns the smallest corresponding index.
FIND_LAST_SEQ_RGE finds a value in a sequence, returns a Boolean element indicating that it was found and if yes returns the largest corresponding index.
COP_SEQ_RGE      copies from one sequence to another.
CMP_SEQ_RGE      comparison of two sequences.
PCOP_SEQ_RGE      partially copies one of the sequences to another.
PCMP_SEQ_RGE      partial comparison of two sequences.

L_ARRAY_COLLECTION: collection of arrays of the same size

CRE_ARR_COL      returns a Boolean element indicating that there is still an array free in the collection and gives the index of this free array.
DEL_ARR_COL      releases the identified array.
VAL_ARR_COL      reads an element from one of the valid arrays.
STR_ARR_COL      writes an element from one of the valid arrays.
COP_ARR_COL      copies one of the arrays to another.
CMP_ARR_COL      compares two tables.

L_ARRAY1_COLLECTION: collection of arrays of the same size with numerical index

CRE_ARR_COL      returns a Boolean element indicating that there is an array free in the collection and the index of this free array (promoted operation).
DEL_ARR_COL      releases the listed array (promoted operation).
VAL_ARR_COL      read a element from on of the valid arrays (promoted operation).
STR_ARR_COL      write a element from one of the valid arrays (promoted operation).
COP_ARR_COL      copies from one of the arrays to another (promoted operation).
CMP_ARR_COL      compares two tables (promoted operation).
SET_ARR_COL      copies the same value to an index range in one of the arrays.
PCOP_ARR_COL     copies part of one of the arrays to another, to a given position.
PCMP_ARR_COL     find the first different element between the two parts of the two different arrays. A Boolean element indicates if this element was found and in this case, the index of this element is returned.
L_RELATION : complete binary relations

- **op_reset**: The relation becomes the empty relation.
- **op_isFullRelation**: Returns TRUE only if the cardinal of the relation equals $\text{max_nb}_\text{2tuple}$.
- **op_add**: Adds a couple to the relation.
- **op_remove**: Removes a couple to the relation.
- **op_cardinal**: Returns the relation cardinal $[]$.
- **op_belongsTo**: Checks if a couple is present in the relation.

\[\text{i.e. the number of couple present in the relation.}\]
4 Description of Basic Machines

The basic machines supplied with Atelier B allow either the creation of dynamic arrays that cannot be obtained using B0, or producing models using vt100 style inputs/outputs. “dynamics arrays” are arrays which size depends on the machine parameters. Such arrays cannot be realised directly in B0, the safety design of the ADA, C and C++ translators do not allow to treat this case. For example, the following construction is not allowed:

```plaintext
IMPLEMENTATION
  mm(xx)
  ...
CONCRETE_VARIABLES
  mytab
INARIANT
  mytab ∈ (0..xx) → NAT
  ...
END
```

Such an array would have to be realised using BASIC_ARRAY_VAR.

The atelier actual version is composed of three basic machines:
- BASIC_ARRAY_VAR Arrays with dimension 1.
- BASIC_ARRAY_RGE Arrays with dimension 2.
- BASIC_IO Usual inputs/outputs management.

This chapter presents this three machines.

The basic machine BASIC_IO is intended to the model designing. It mustn’t be considered as safe.

**WARNING:** The manual implementations of the basic machines BASIC_ARRAY_VAR and BASIC_ARRAY_RGE destined for the translators supplied with Atelier B are provided as demonstration. They are not safe, and are not appropriated in all the B use contexts.
4.1 BASIC_ARRAY_VAR: Implanting a one dimensional table

OPERATIONS

VAL_ARRAY read a table element
STR_ARRAY write a table element

EXAMPLE

Example of use with listed sets:

```
<table>
<thead>
<tr>
<th>MACHINE array</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETS</td>
</tr>
<tr>
<td>FONTS = {Times, Serif, Courier};</td>
</tr>
<tr>
<td>FTYPE = {fixed, unfixed}</td>
</tr>
<tr>
<td>VARIABLES</td>
</tr>
<tr>
<td>fixedsz</td>
</tr>
<tr>
<td>INVARIANT</td>
</tr>
<tr>
<td>fixedsz ∈ FONTS → FTYPE</td>
</tr>
<tr>
<td>INITIALISATION</td>
</tr>
<tr>
<td>fixedsz:={Times ↦ unfixed,</td>
</tr>
<tr>
<td>Serif ↦ fixed, Courier ↦ fixed}</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>
```

arr_vrb is the name of the table encapsulated by BASIC_ARRAY_VAR

DESCRIPTION

BASIC_ARRAY_VAR models one dimensional arrays. Such arrays cannot be created directly in B0 if their size depends on the machine parameters ("dynamic arrays"). The current design of ADA or C translators does not allow handling this case. The following construction is therefore illegal:

```
| IMPLEMENTATION |
| mm(xx) |
| ... |
| VARIABLES |
| mytab |
| INVARIANT |
| mytab ∈ (0..xx) → NAT |
| ... |
| END |
```

This kind of table should be generated using BASIC_ARRAY_VAR.

MACHINE PARAMETERS

BASIC_ARRAY_VAR (BAV_INDEX, BAV_VALUE): BAV_INDEX is the set of values used to index the table, BAV_VALUE is the set of possible values for table elements.
The B language rule relating to the possible values of the BAV_VALUE parameter ensure that: if a computer variable can contain elements of MININT..MAXINT, then it can contain those of BAV_VALUE. For example, B rules forbid assigning BAV_VALUE the value of MAXINT+1,MAXINT+2

**VAL_ARRAY**

```plaintext
syntax          vv ← VAL_ARRAY(ii)
preconditions   ii must be a BAV_INDEX
outputs         vv is a BAV_VALUE, the value of the array at position ii.
```

**STR_ARRAY**

```plaintext
syntax          STR_ARRAY(ii,vv)
preconditions   ii must be a BAV_INDEX and vv must be a BAV_VALUE
```

The value vv is stored in the array at ii index.

**C++ LANGUAGE**

In C++, the array is realised by an integer array. The accesses to this array are done using method that refuse the index used between 0 and the array size, guaranting an optimal memory use.

The array is dynamically reserved when launching the program. If the size indicated by the formal parameters is too big, the program stops with the following message:

*Virtual memory exceede in ‘‘new’’*

**C LANGUAGE**

The realisation in C is based on the same principles as in C++. The stop message on initial reservation failure is:

*Fatal error: Malloc of X bytes failed*

Execution of current application is aborted

**ADA LANGUAGE**

The use of generic packaging guarantees an optimal memory occupation. No restrictions are made on the instanciation parameters. On initial reservation failure, an exception stops the program.
PROGRAMMING

Example of use with literal sets:

```
MACHINE narr
VARIABLES myvar
INVARIANT myvar ∈ 0..2 → 0..1
INITIALISATION myvar := {0 ↦ 0, 1 ↦ 1, 2 ↦ 1}
END

IMPLEMENTATION narr_1
REFINES narr
IMPORTS BASIC_ARRAY_VAR(0..2,0..1)
INVARIANT arr_vrb = myvar
INITIALISATION
  STR_ARRAY(0,0);
  STR_ARRAY(1,1);
  STR_ARRAY(2,1)
END
```

Another example. Only the implementation is presented. The write of a machine refined by this implementation is an exercise for the reader:

```
IMPLEMENTATION parr_1
REFINES parr
IMPORTS BASIC_ARRAY_VAR(FONTS, FTYPE)
VALUES
  FONTS = 5..7;
  FTYPE = 3..4
INVARIANT arr_vrb = fixedsz
INITIALISATION
  STR_ARRAY(5,3);
  STR_ARRAY(6,4);
  STR_ARRAY(7,5)
END
```

NOTE: The possible values of the BASIC_ARRAY_VAR parameters are given by the B language rules, (refer to section 12.2 page 574 of the BBOOK)
## 4.2 BASIC_ARRAY_RGE: Implementing a Two Dimensional Array

### OPERATIONS
- VAL_ARR_RGE: read an array element
- STR_ARR_RGE: write an array element
- COP_ARR_RGE: copy an array line to another
- CMP_ARR_RGE: compare two array lines

### EXAMPLE
Example of use, two lines and three columns array:

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitab</td>
<td>bitab_1</td>
</tr>
<tr>
<td>SETS</td>
<td></td>
</tr>
<tr>
<td>LGNS = {ll1,ll2}</td>
<td></td>
</tr>
<tr>
<td>VARIABLES</td>
<td></td>
</tr>
<tr>
<td>mytab</td>
<td></td>
</tr>
<tr>
<td>INVARIANT</td>
<td>mytab ∈ LGNS → (1.3 → 0.255)</td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>mytab := {{ll1 ↦ {1 ↦ 7.2 ↦ 8.3 ↦ 9}}, ll2 ↦ {1 ↦ 0,2 ↦ 1,3 ↦ 2}}</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

The variable arr_rge is the name of the encapsulated array par BASIC_ARRAY_RGE

### DESCRIPTION
BASIC_ARRAY_RGE models two dimensional arrays. Such arrays cannot be created directly in B0 if their size depends on the machine parameters (“dynamic array”). The safe design of the ADA, C++ or C translators do not allow to treat this case. The following construction is forbidden:

<table>
<thead>
<tr>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm(xx)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>CONCRETE_VARIABLES</td>
</tr>
<tr>
<td>mytab</td>
</tr>
<tr>
<td>INVARIANT</td>
</tr>
<tr>
<td>mytab ∈ (0..10) → (0..xx)×(0..xx)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>
Such an array must be implemented using BASIC\_ARRAY\_RGE.

**MACHINE PARAMETERS**

BASIC\_ARRAY\_RGE(BAR\_INDEX,BAR\_VALUE,BAR\_RANGE):

- **BAR\_INDEX** represents the column indexes.
- **BAR\_VALUE** is the set of the possible values for the array elements,
- **BAR\_RANGE** represents the line indexes.

The B language rules concerning the possible values of the BAR\_VALUE parameter ensure that a computing variable being able to contain the elements of MININT..MAXINT, then it can contain those of BAR\_VALUE. For example, the B rules do not permit to give to BAR\_VALUE the value MAXINT+1,MAXINT+2.

**VAL\_ARR\_RGE**

**syntax**

\[ v \leftarrow \text{VAL}\_\text{ARR}\_\text{RGE}(rr,ii) \]

**preconditions**

ii must be a BAR\_INDEX, rr must be a BAR\_RANGE

**outputs**

vv is an element of BAR\_VALUE, which value is the array value at position ii, line rr.

**STR\_ARR\_RGE**

**syntax**

\[ \text{STR}\_\text{ARR}\_\text{RGE}(rr,ii,vv) \]

**preconditions**

rr must be an element of BAR\_RANGE, ii an element of BAR\_INDEX and vv an element of BAR\_VALUE

Value vv is stored in the array line rr, index ii.

**COP\_ARR\_RGE**

**syntax**

\[ \text{COP}\_\text{ARR}\_\text{RGE}(\text{dest},\text{src}) \]

**preconditions**

dest and src must be elements of BAR\_RANGE

The src line is copied to the dest line.

**CMP\_ARR\_RGE**

**syntax**

\[ bb \leftarrow \text{CMP}\_\text{ARR}\_\text{RGE}(\text{range1},\text{range2}) \]

**preconditions**

range1 and range2 must be elements of BAR\_RANGE

**outputs**

bb is an element of BOOL, that takes the TRUE value if the two lines are equal.

**C++ LANGUAGE**

In C++, the array is realised by an array of pointers, pointing on integers arrays. The access to these arrays are done using methods that refuse the index used between 0 and the arrays size, guaranting an optimal memory occupation.

The memory is dynamically reserved when lauching the program. If the size indicated by the formal parameters is too big, the program stops with the following message:

*Virtual memory exceeded in ‘new’*
C LANGUAGE
The realisation in C is based on the same principles as in C++. The stop message on the initial reservation failure is: Fatal error: Malloc of X bytes failed Execution of current application is aborted

ADA LANGUAGE
The use of generic packages guarantees an optimal memory occupation. No restriction is made on the instancing parameters. On an initial reservation failure, an exception stops the program.
4.3 BASIC_IO: vt100 style inputs/outputs

OPERATIONS
INTERVAL_READ operator input of an integer in mm..nn.
INT_WRITE print an integer.
BOOL_READ operator input of a Boolean TRUE or FALSE state
BOOL_WRITE print TRUE or FALSE.
CHAR_READ operator input of a character.
CHAR_WRITE print a character.
STRING_WRITE print a message.

SIMPLE EXAMPLE
The following implementation displays “hello” on the terminal:

```
MACHINE
bonj
OPERATIONS
main = skip
END

IMPLEMENTATION
bonj
REFINES
bonj
IMPORTS
BASIC_IO
OPERATIONS
main = begin
  STRING_WRITE("hello\n")
end
END
```

DESCRIPTION
BASIC_IO is used for simple input/output actions on a terminal. This basic machine is
used to build models. Such I/O cannot be considered as safe.

In UNIX, the system devices used are standard input and standard output (stdin and
stdout), they can therefore be redirected.

INTERVAL_READ

```plaintext
syntax          bb ← INTERVAL_READ(mm,nn)
preconditions    mm and nn must be NATs so that mm ≤ nn
outputs          bb integer in mm..nn
```

The operator inputs an integer of the interval mm..nn. The input format forces to type a
succession of number(s) followed by RETURN. The first input character must be a number.
On the opposite case, the input fails “3” is not valid). When a character that is not the
first input is not a number anymore, this character, as all the following ones, are ignored:
“3e2” is a valid input of the integer 3. As long as the input is false, the message “THIS IS
NOT A NUMBER IN mm..nn” is displayed and a new entry is required.
INT_WRITE

.syntax  INT_WRITE(vv)

.preconditions  vv must belong to NAT

Output number vv, with no return.

BOOL_READ

.syntax  bb ← BOOL_READ

.outputs  bb must be Boolean.

The operator enters Boolean TRUE or FALSE conditions, with no character before it (for example: “TRUE” is rejected because of the space before it). As long as the operator has not made a valid entry, the message “THIS IS NOT A BOOL VALUE: type TRUE or FALSE” is displayed and a new entry is required.

BOOL_WRITE

.syntax  BOOL_WRITE(bb)

.preconditions  bb must be Boolean

Output TRUE or FALSE, with no return.

CHAR_READ

.syntax  cc ← CHAR_READ

.outputs  cc must be part of 0..255

Operator entry of a character that is interpreted as a number in 0..255. Type in the character followed by return. If several characters has been typed, only the first one is taken into account (example: “cdef” is understood as “=32). In C, pressing Return only returns 10, ctrl-D (EOF) returns 0. In ADA, only the ‘visible’ characters entries (i.e, no control characters) are accepted.

CHAR_WRITE

.syntax  CHAR_WRITE(vv)

.preconditions  vv must belong to the range 0..255

Displays the cc character on-screen (example: CHAR_WRITE(10) to produce a return). Remember, a single quote means “prime” the language’s notation conventions, and B. CHAR_WRITE(’A’) for example, means nothing. On the contrary, the quoted strings are valid elements in a formula, they serve for STRING_WRITE below.

STRING_WRITE

.syntax  STRING_WRITE(ss)

.preconditions  ss must be an element in the STRING set

Will display a character string on-screen. For ss use quoted strings. A “C type” formatting is used, even for a translation into ADA, i.e,

\t produces a tab
\E produces Escape
\B produces a sound
\" produces a quote
KNOWN PROBLEMS
STRING does not have a coherent definition. The prover proves that any character string belongs to STRING due to an ad hoc rule, that does not derive from the definition STRING = seq(CHAR). In addition, using a STRING type local variable in an implementation is not possible. To be completely rigorous, nothing ensures that the operator performs all the requested entries. Therefore the operations for entering the true data entry module (BASIC_IO.c for example) do not really implant the specifications of the corresponding B operations.

PROGRAMMING
A more complete example:

MACHINE bio
OPERATIONS
main = skip
END

IMPLEMENTATION
bio_1
REFINES
bio
IMPORTS
BASIC_ARITHMETIC,BASIC_IO
OPERATIONS
main = var zz,bb,cc in
    zz ← INTERVAL_READ(0,100);
    STRING_WRITE("this is the value : ");
    INT_WRITE(zz);
    CHAR_WRITE(10);
    bb ← BOOL_READ;
    STRING_WRITE("this is the value : ");
    BOOL_WRITE(bb);
    CHAR_WRITE(10);
    cc ← CHAR_READ;
    STRING_WRITE("this is the value : ");
    INT_WRITE(cc);
    STRING_WRITE(" = ");
    CHAR_WRITE(cc);
    CHAR_WRITE(10)
END
END

Execution example:

ATELIER-B% bio
sdfsdf
THIS IS NOT A NUMBER IN 0..100
20
this is the value: 20
CRUE
THIS IS NOT A BOOL VALUE: type TRUE or FALSE
TRUE
this is the value: TRUE
cvf
NOTE: To be completely rigorous, nothing ensures that the operator performs all the entries requested. The entry loops of the concrete module (BASIC_IO.c for example) do not really implant the specifications of the corresponding operations.

Possible evolutions:

It should be possible to define in the machine BASIC_IO., abstract variables modeling the inputs/outputs; it should then be possible to specify the required interactions of the external system. The abstract machine that needs to handle inputs/outputs will use BASIC_IO notions (by SEES or INCLUDES) to represent the required interactions.
5 Description of Library Machines

The library machines are all intended for creating mathematical objects, except machine L_ARITHMETIC1 that provides certain arithmetical functions. The modeled mathematical objects are:

- **total functions**: these are machines contain “ARR” (array) in their name;
- **partial functions**: machines with the “PFNC” (partial function) in their name;
- **sets**: these are machines with the “SET” (set) in their name;
- **sequences**: these are machines with the “SEQ” (sequence) in their name.

For each mathematical object, it is possible to realize either a variable representing the object, or a variable representing several objects of this type. For each type of object, it is therefore possible to realize:

- The object itself;
- An array of objects with the same type, same size, these are machines with a name containing the “RGE” (range) radical;
- A partial function of objects with the same size and same type, these are machines with a name containing the “COL” (collection) radical;
- A partial function of objects with the same type, but with various sizes (“OBJ” radical).

The “RGE” and “COL” type machines produce objects that consume the memory necessary for the maximum number of required objects. For example, if we create a range or a collection of three sequences of at least ten elements, we will always require 30 memory spaces; but the use of a collection avoids the user program to manage the sequences available/occupied. Object machines reserve a memory space that may be freely distributed depending on the created objects and their size. Mathematical objects listed above are not all available on the different types of machines, refer to library machines table of contents for the list that corresponds to the current version.

**WARNING**: Most of the library machines are based on the basic machines BASIC_ARRAY_VAR and BASIC_ARRAY_RGE. The manual implementations of the basic machines BASIC_ARRAY_VAR and BASIC_ARRAY_RGE destined to the translators supplied with Atelier B are provided as a demonstration. They are not safe, and not appropriate in all the B use context. In the case of a more complete use, the user would have to realize these basic machines.
5.1 L_ARITHMETIC1: Extended Integer Operations

The “integer” term refers to the elements of NAT, NAT that is the set of the natural integers between 0 and MAXINT.

OPERATIONS

- **MIN** minimum of two integers.
- **MAX** maximum of two integers.
- **INC** increment an integer strictly inferior to MAXINT.
- **DEC** decrement a literal integer.
- **EXP** exponentiation.
- **SQRT** default integer square root.
- **LOG** _by default_ default logarithm.
- **LOG** _by excess_ logarithm by excess.

EXAMPLE

The example below shows a machine that uses a certain number of functionalities of the machine L_ARITHMETIC1.

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>m1_1</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>REFINES</td>
</tr>
<tr>
<td>xx ← op1 = ANY tt WHERE tt ∈ NAT ∧ tt×tt = 16 THEN xx:=tt END;</td>
<td>m1</td>
</tr>
<tr>
<td>xx ← op2 = ANY tt WHERE tt ∈ NAT ∧ 3^tt = 27 THEN xx:=tt END</td>
<td>L_ARITHMETIC1, IMPORTS</td>
</tr>
<tr>
<td>END</td>
<td>OPERATIONS</td>
</tr>
<tr>
<td>xx ← op1 = BEGIN xx ← SQRT(16) END;</td>
<td>OPERATIONS</td>
</tr>
<tr>
<td>xx ← op2 = VAR tt IN xx,tt ← LOG_BY_DEFAULT (3, 27) END</td>
<td>END</td>
</tr>
</tbody>
</table>

DESCRIPTION

L_ARITHMETIC1 offers arithmetical operations such as roots and logarithms, operations on the elements NAT and dedicated to calculatory applications. Calculus being integers values, the search operation for the logarithm and the square root return the _best approaching_ value in NAT. The used algorithms are optimized.

MACHINE PARAMETERS

None.

---

1 The NAT element immediately inferior or superior whether the calcul is performed by inferior value or superior value.
MIN

syntax \( uu ← \text{MIN}(vv,ww) \)

preconditions \( vv \) and \( ww \) must be in NAT.

outputs \( uu = \min \{vv,ww\} \)

MAX

syntax \( uu ← \text{MAX}(vv,ww) \)

preconditions \( vv \) and \( ww \) must be in NAT.

outputs \( uu \) receives \( \max\{vv,ww\} \)

INC

syntax \( uu ← \text{INC}(vv) \)

preconditions \( vv \) must be in \( 0..\text{MAXINT}-1 \).

outputs \( uu = vv+1 \)

DEC

syntax \( uu ← \text{DEC}(vv) \)

preconditions \( vv \) must be in \( 1..\text{MAXINT} \).

outputs \( uu = vv-1 \)

EXP

syntax \( rr ← \text{EXP}(xx,nn) \)

preconditions \( xx \) and \( nn \) must be in NAT. \( xx \) and \( nn \) must not both be nil. \( xx^{nn} \) must be less than or equal to \( \text{MAXINT} \).

outputs \( rr \) receives \( xx^{nn} \)

EXP returns \( xx \) to the power of \( nn \). Calculating \( 0^0 \) is illegal (\( 0^0 \) is not defined). The implementation uses a fast algorithm based on breaking down into base 2 of \( nn \) (\( \log_2(nn) \)) iterations).

SQRT

syntax \( nn ← \text{SQRT}(pp) \)

preconditions \( pp \) must be in NAT.

outputs \( nn \) so that \( nn \times nn \leq pp < (nn+1) \times (nn+1) \)

SQRT returns the largest \( nn \) so that \( nn \times nn \leq pp \). The implementation uses an algorithm that performs \( \text{SQRT}(nn) \) iterations, where each iteration costs two additions and a subtraction.

LOG_BY_DEFAULT

syntax \( uu,rr ← \text{LOG_BY_DEFAULT}(vv,ww) \)

preconditions \( ww \) and \( vv \) are two natural integers and \( vv \) is between 2 and \( \text{MAXINT} \).

outputs \( uu \) is the smallest natural so that \( vv^{(uu+1)} \) is strictly greater than \( ww \). By definition, \( uu \) is a natural integer. \( rr \) takes the value \( vv^{uu} \).
LOG_BY_DEFAULT in base vv of ww: returns the smallest uu value so that \( w < v^u(v+1) \).
This gives \( v^u \leq w \), except if \( w < v \) (example: \( w = 0 \)). Does not work for \( v = 0 \) or 1 as 0 and 1 are constants. rr receives the value of \( v^u \), which easily allows judging the error made.

LOG_BY_EXCESS

\[ uu,bb \leftarrow \text{LOG_BY_EXCESS}(vv,ww) \]

**syntax**

\( uu,bb \leftarrow \text{LOG_BY_EXCESS}(vv,ww) \)

**preconditions**

\( ww \) belongs to NAT and vv is an element of the intervall 2..MAXINT.

**outputs**

\( uu \) receives the smallest natural so that \( v^u \) is greater than or equal to \( w \). uu must be in NAT. bb is an element of BOOL, it indicates whether the logarithm is an exact one.

LOG_BY_EXCESS in base vv in ww: returns the smallest uu so that \( w \leq v^u \). WARNING: \( v^u \) may exceed MAXINT! Does not work for \( v = 0 \) or 1 as 0 and 1 are constants. bb equals TRUE if \( w = v^u \).

**IMPORTS REQUIRED**

None.
5.2 L_ARRAY1: One Dimensional Array, with Initialization Loop

OPERATIONS

VAL_ARRAY  value of an element (promoted operation)
STR_ARRAY  write an element (promoted operation)
SET_ARRAY  write the same value in a portion of the array

EXAMPLE

Use SET_ARRAY to initialize an array:

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>m1</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>IMPORTS</td>
</tr>
<tr>
<td>vv</td>
<td>i1.L_ARRAY1(0..255,10)</td>
</tr>
<tr>
<td>INARIANT</td>
<td>(arr_varb is the variable in L_ARRAY1)</td>
</tr>
<tr>
<td>vv ∈ 0..10 → 0..255</td>
<td>i1.arr_varb = vv</td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>i1.SET_ARRAY(0,10,5)</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
</tr>
</tbody>
</table>

DESCRIPTION

As it is possible, L_ARRAY1 is used instead of BASIC_ARRAY_VAR. L_ARRAY1 realizes, using an array, an abstract variable representing a function. It is then possible to have an initialization operation of the entire function or of a part of it (initialization loop).

The starting part of the function performed is an interval: if not, it would not be possible to indicate a portion of this set without mentioning all elements involved.

MACHINE PARAMETERS

L_ARRAY1(LAU_VALUE, LAU_maxidx): LAU_VALUE is the set of possible values for the array elements, 0..LAU_maxidx is the set of array indexes.

VAL_ARRAY

syntax       vv ← VAL_ARRAY(ii)
preconditions ii must be in 0..LAU_maxidx
outputs      vv is an element of LAU_VALUE, the array value at position ii.
**STR ARRAY**

*Syntax*  
STR ARRAY(ii,vv)

*Preconditions*  
ii and vv must belong to the 0..LAU_maxidx and LAU_VALUE respectively.

vv value is stored in the array at ii index.  

**SET ARRAY**

*Syntax*  
SET ARRAY (ii,jj,vv)

*Preconditions*  
ii..jj is a sub-set of 0..LAU_maxidx and vv an element of LAU_VALUE.
For implementation reasons, jj and MAXINT must be different.

The value vv is stored in the array for all the indexes between ii to jj. If ii>jj, the array does not change.

Note that it would not have been advisable to set ii≤jj as a precondition of this operation, as this would have limited its use. Let us consider the case of a call to SET ARRAY in a loop. The last iteration fo the loop contains a call with the form SET ARRAY (ii, jj, vv) with ii=jj+1. The presence of a precondition in the definition of the operation SET ARRAY would force us to “guard” all the calls to SET ARRAY by an IF. More generally, the precondition must be selected as minimal to protect us from a code of “defensive” aspect.

**IMPORTS REQUIRED**

None.

**Warning:** The implementation of this machine creates the default instance for the BASIC ARRAY_VAR machine (IMPORTS BASIC ARRAY_VAR(...)). The addition of an instance of the machine BASIC ARRAY_VAR requires choosing a new instance name, as, for example: i1.BASIC ARRAY_VAR).

---

2Indeed, the loops used make a pre-incrementation, that does not produce literal exceedent)
5.3 L_ARRAY3: Array with Non Ordered Values, Maximum Operations

OPERATIONS

VAL_ARRAY value of an element (promoted operation).
STR_ARRAY write an element (promoted operation).
SET_ARRAY write the same value in an array portion (promoted operation).
SWAP_ARRAY exchange two elements (promoted operation).
RIGHT_SHIFT_ARRAY shift a portion to the main index (promoted operation).
LEFT_SHIFT_ARRAY shift a portion to the small index (promoted operation).
SEARCH_MAX_EQL_ARRAY search for a value in an array (promoted operation).
SEARCH_MIN_EQL_ARRAY search for a value in an array portion (promoted operation).
REVERSE_ARRAY reverse the order of elements in an array portion.

EXAMPLE
The example below is a machine that represents the color assigned to 101 points, this color may be red, green or blue for each point. An operation is used to find a red dot.

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>m1_l</td>
</tr>
<tr>
<td>SETS</td>
<td></td>
</tr>
<tr>
<td>COLOR = {red, green, blue}</td>
<td>IMPORTS</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>i1.L_ARRAY3(COLOR,100)</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>color</td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>color :=(0..100) ×{red}</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>OPERATIONS</td>
</tr>
<tr>
<td>ii,bb ← trouve_red = PRE rouge ∈ ran(color) THEN END</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

DESCRIPTION
L_ARRAY3 is the most complete of the one dimensional array machines that do not require that the output set be part of an interval. L_ARRAY5 has been constrained. It is therefore possible to create arrays with values that are elements of a listed set while having access to complete operations such as element order reversal. The operation that
is not available is the one that would require an order relationship on the array elements: sort.

**MACHINE PARAMETERS**

L_ARRAY3(LAT_VALUE,LAT_maxidx): LAT_VALUE is the set of possible values for array elements, 0..LAT_maxidx is the set of array indexes.

**VAL_ARRAY**

*Syntax*  
vv ← VAL_ARRAY(ii)

*Preconditions*  
ii must be in 0..LAT_maxidx

*Outputs*  
vv is a LAT_VALUE, it is the value of the array at position ii.

**STR_ARRAY**

*Syntax*  
STR_ARRAY(ii,vv)

*Preconditions*  
ii must be in 0..LAT_maxidx and vv must belong to LAT_VALUE

The vv value is stored in the array at index ii.

**SET_ARRAY**

*Syntax*  
SET_ARRAY(ii,jj,vv)

*Preconditions*  
ii..jj must be a subset of 0..LAT_maxidx and vv belong to LAT_VALUE. For implementation reasons it is also necessary that jj be different from MAXINT.

The vv value is stored in the array for all indexes between ii and jj. If ii>jj, the array will not change.

**SWAP_ARRAY**

*Syntax*  
SWAP_ARRAY(ii,jj)

*Preconditions*  
ii,jj must be in 0..LAT_maxidx.

The ii and jj elements in the array are exchanged.

**RIGHT_SHIFT_ARRAY**

*Syntax*  
RIGHT_SHIFT_ARRAY(ii,jj,nn)

*Preconditions*  
ii,jj,nn must be in 0..LAT_maxidx, with ii≤jj and jj+nn≤LAT_maxidx to make possible the possible the shift to the right by nn spaces.

Part ii+nn..jj+nn receives a copy of part ii..jj of the array (shift nn spaces to the right).

**LEFT_SHIFT_ARRAY**

*Syntax*  
LEFT_SHIFT_ARRAY(ii,jj,nn)

*Preconditions*  
ii,jj must be in 0..LAT_maxidx, with ii≤jj. nn must be NAT with nn≤ii to make possible the shift to the left by nn places. For implementation reasons, jj must be not equal MAXINT.

The ii-nn..jj-nn part receives a copy of part ii..jj from the array (shift nn spaces to the left).
SEARCH_MAX_EQL_ARRAY
syntax \( rr, bb \leftarrow \text{SEARCH\_MAX\_EQL\_ARRAY}(ii, jj, vv) \)
preconditions \( ii \) and \( jj \) must be in \( 0..\text{LAT\_maxidx} \), \( ii \leq jj \) and \( vv \) belong to \( \text{LAT\_VALUE} \).
outputs TRUE if \( vv \) was found, FALSE if not. \( rr \) is a NAT, if \( bb = \text{TRUE} \) then \( rr \) is the largest index in the array worth \( vv \).

Search for an array element equal to \( vv \), by scanning the \( ii..jj \) part starting from \( jj \).

SEARCH_MIN_EQL_ARRAY
syntax \( rr, bb \leftarrow \text{SEARCH\_MIN\_EQL\_ARRAY}(ii, jj, vv) \)
preconditions \( ii \) and \( jj \) must be in \( 0..\text{LAT\_maxidx} \), \( ii \leq jj \) and \( vv \) belong to \( \text{LAT\_VALUE} \).
outputs TRUE if \( vv \) was found, FALSE if not. \( rr \) is a NAT, if \( bb = \text{TRUE} \), then \( rr \) is the smallest index in the array worth \( vv \).

Search for an array element that equals \( vv \), by scanning the \( ii..jj \) part starting from \( ii \).

REVERSE_ARRAY
syntax \( \text{REVERSE\_ARRAY}(ii, jj) \)
preconditions \( ii \) and \( jj \) must be in \( 0..\text{LAT\_maxidx} \).
Reverse the order of elements in the \( ii..jj \) portion of the array.

IMPORTS REQUIRED
(instances to import as the implementation tree for this library machine sees them with SEES)
BASIC_ARITHMETIC; BASIC_BOOL.

WARNING: The implementation of this machine creates the default instance for the BASIC\_ARRAY\_VAR machine (clause IMPORTS BASIC\_ARRAY\_VAR(...)). Therefore if another instance is necessary, it must be given a different instance name (for example: i1.BASIC\_ARRAY\_VAR).
## 5.4 L ARRAY5: Array with Ordered Values, Sort Operation

### OPERATIONS

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAL ARRAY</td>
<td>value of an element (promoted operation).</td>
</tr>
<tr>
<td>STR ARRAY</td>
<td>write an element (promoted operation).</td>
</tr>
<tr>
<td>SET ARRAY</td>
<td>write the same value to a portion of an array (promoted operation).</td>
</tr>
<tr>
<td>SWAP ARRAY</td>
<td>exchange two elements (promoted operation).</td>
</tr>
<tr>
<td>RIGHT SHIFT ARRAY</td>
<td>shift a portion to the large index (promoted operation).</td>
</tr>
<tr>
<td>LEFT SHIFT ARRAY</td>
<td>shift a portion to the small index (promoted operation).</td>
</tr>
<tr>
<td>SEARCH MAX EQL ARRAY</td>
<td>search for a value in a portion of the array (promoted operation).</td>
</tr>
<tr>
<td>SEARCH MIN EQL ARRAY</td>
<td>search for a value in a portion of the array (promoted operation).</td>
</tr>
<tr>
<td>REVERSE ARRAY</td>
<td>reverse the order of the elements in a portion of the array (promoted operation).</td>
</tr>
<tr>
<td>SEARCH MIN GEQ ARRAY</td>
<td>search for the first element that exceeds a value (promoted operation).</td>
</tr>
<tr>
<td>ASCENDING SORT ARRAY</td>
<td>sort a portion of the array.</td>
</tr>
</tbody>
</table>

### EXAMPLE

#### MACHINE

```plaintext
MACHINE m1
VARIABLES vv
INVARIANT
   vv ∈ 0..4 → 0..255 ∧
   ∀xx.(xx ∈ 0..3 ⇒ vv(xx) ≥ vv(xx+1))
INITIALISATION
   vv : (vv ∈ 0..4 → 0..255 ∧
   ∀xx.(xx ∈ 0..3 ⇒ vv(xx) ≥ vv(xx+1)))
END
```

#### IMPLEMENTATION

```plaintext
IMPLEMENTATION m1_1
REFINES m1
IMPORTS L ARRAY5(0,255,4)
INVARIANT
   arr_vrb = vv
INITIALISATION
   SET ARRAY(0,4,50);
   STR ARRAY(2,10);
   STR ARRAY(4,30);
   ASCENDING SORT ARRAY(0,4);
   REVERSE ARRAY(0,4)
END
```

### DESCRIPTION

L ARRAY5 is the most complete of the one dimensional array machines. It especially comprises a sort operation implanted using a shift sort (fast algorithm).
MACHINE PARAMETERS

L_ARRAY5(LAC_minval,LAC_maxval,LAC_maxidx): LAC_minval..LAC_maxval is the set of possible values for the elements in the array, 0..LAC_maxidx is the set of index values for the array. LAC_minval, LAC_maxval, LAC_maxidx must be NATs: this machine does not allow negative values. It is also necessary for LAC_minval ≤ LAC_maxval and 1 ≤ LAC_maxidx.

VAL_ARRAY

syntax vv ← VAL_ARRAY(ii)
preconditions ii must be in 0..LAC_maxidx
outputs vv is in LAC_minval..LAC_maxval, is the array value at position ii.

STR_ARRAY

syntax STR_ARRAY(ii,vv)
preconditions ii must be in 0..LAC_maxidx and vv in LAC_minval..LAC_maxval and LAC_VALUE.
The vv value is stored in the array at index ii.

SET_ARRAY

syntax SET_ARRAY(ii,jj,vv)
preconditions ii..jj must be included in 0..LAC_maxidx and vv must be in LAC_VALUE. For implementation, it is also necessary that jj be different from the MAXINT constant.
The vv value is stored in the array for all indexes from ii to jj. If ii > jj, the array does not change.

SWAP_ARRAY

syntax SWAP_ARRAY(ii,jj)
preconditions ii,jj must be in 0..LAC_maxidx.
The ii and jj elements in the array are exchanged.

RIGHT_SHIFT_ARRAY

syntax RIGHT_SHIFT_ARRAY(ii,jj,nn)
preconditions ii,jj,nn must be in 0..LAC_maxidx, with ii ≤ jj and jj + nn ≤ LAC_maxidx to make possible the right shift by nn spaces.
The ii + nn..jj + nn part receives a copy of the ii..jj part of the array (shift right by nn spaces).

LEFT_SHIFT_ARRAY

syntax LEFT_SHIFT_ARRAY(ii,jj,nn)
preconditions ii,jj must be in 0..LAC_maxidx, with ii ≤ jj. nn must be a NAT with nn ≤ ii to allow the left shift by nn spaces. For implementation reasons, jj cannot equal MAXINT.
The ii-nn..jj-nn part receives a copy of the ii..jj part of the array (shift left by nn spaces).
SEARCH_MAX_EQL_ARRAY

**syntax**

\[
rr, bb \leftarrow \text{SEARCH\_MAX\_EQL\_ARRAY}(ii, jj, vv)
\]

**preconditions**

\(ii\) and \(jj\) must be in \(0..\text{LAC\_maxidx}\), \(ii \leq jj\) and \(vv\) be in \(\text{LAC\_VALUE}\).

**outputs**

RULE if \(vv\) was found, FALSE if not. \(rr\) is a NAT, if \(bb = \text{TRUE}\), then \(rr\) is the highest index in the array worth \(vv\).

Search for an array element equal to \(vv\), by scanning the \(ii..jj\) part starting from \(jj\).

SEARCH_MIN_EQL_ARRAY

**syntax**

\[
rr, bb \leftarrow \text{SEARCH\_MIN\_EQL\_ARRAY}(ii, jj, vv)
\]

**preconditions**

\(ii\) and \(jj\) must be in \(0..\text{LAC\_maxidx}\), \(ii \leq jj\) and \(vv\) be in \(\text{LAC\_VALUE}\).

**outputs**

RULE if \(vv\) was found, FALSE if not. \(rr\) is a NAT, if \(bb = \text{TRUE}\) then \(rr\) is the smallest index in the array worth \(vv\).

Search for an array element equal to \(vv\), by scanning the \(ii..jj\) part starting from \(ii\).

REVERSE_ARRAY

**syntax**

\[
\text{REVERSE\_ARRAY}(ii, jj)
\]

**preconditions**

\(ii\) and \(jj\) must be in \(0..\text{LAC\_maxidx}\).

Reverse the order of the elements in the \(ii..jj\) portion of the array.

SEARCH_MIN_GEQ_ARRAY

**syntax**

\[
ii, bb \leftarrow \text{SEARCH\_MIN\_GEQ\_ARRAY}(jj, kk, vv)
\]

**preconditions**

\(jj\) and \(kk\) must be in \(0..\text{LAC\_maxidx}\), \(jj \leq kk\) and \(vv\) be in \(\text{LAC\_minval..LAC\_maxval}\). For implementation location reasons, \(kk\) must not equal the \(\text{MAXINT}\) constant.

**outputs**

RULE if an element that is greater or equal to \(vv\) was found, FALSE if not. \(ii\) is a NAT, if \(bb = \text{TRUE}\), then \(ii\) is the smallest index in the image array that is greater than or equal to \(vv\).

Search for an element that is greater than or equal to \(vv\) in \(jj..kk\) starting from \(jj\).

ASCENDING_SORT_ARRAY

**syntax**

\[
\text{ASCENDING\_SORT\_ARRAY}(ii, jj)
\]

**preconditions**

\(ii\) and \(jj\) must be in \(0..\text{LAC\_maxidx}\). For implementation reasons, \(ii\) and \(jj\) must not equal \(\text{MAXINT}\).

Shift sort, in ascending order (the smallest first) on the \(ii..jj\) portion.

IMPORTS REQUIRED

(Instances to import as the implementation tree for this library machine sees them with SEES)

\text{BASIC\_ARITHMETIC; BASIC\_BOOL}.

WARNING: The implementation of this machine creates the default instance for the \text{BASIC\_ARRAY\_VAR} machine (clause IMPORTS \text{BASIC\_ARRAY\_VAR}(...)). Therefore if another instance is required it must be given a different instance name (for example: \text{i1.BASIC\_ARRAY\_VAR}).
5.5 L_PFNC: Partial Function

OPERATIONS

VAL_PFNC value of the function for an element in its domain
STR_PFNC overloads the partial function with a pair
XST_PFNC test that an index is in the partial function domain
RMV_PFNC removes a pair from the partial function
SET_PFNC overloads a part of the function with a constant
SWAP_PFNC exchanges the images for two domain indexes
RIGHT_SHIFT_PFNC right shift part of the domain
LEFT_SHIFT_PFNC left shift part of the domain
SEARCH_MAX_EQL_PFNC search for a value in the partial function
SEARCH_MIN_EQL_PFNC search for a value in the partial function
REVERSE_PFNC reverse the order of elements in a portion of the domain
ASCENDING_SORT_PFNC sort in a portion of the domain

EXAMPLE

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>m1_1</td>
</tr>
<tr>
<td>VARIABLES</td>
<td></td>
</tr>
<tr>
<td>pf</td>
<td></td>
</tr>
<tr>
<td>INVARIANT</td>
<td></td>
</tr>
<tr>
<td>pf ∈ 0..10 → 0..255</td>
<td>pfnc_vrb = pf</td>
</tr>
<tr>
<td>INITIALISATION</td>
<td></td>
</tr>
<tr>
<td>pf := {4 → 6}</td>
<td>STR_PFNC(4,6)</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
</tr>
</tbody>
</table>

DESCRIPTION

L_PFNC implements a partial function with almost all of the operations available in L_ARRAY5 (In fact only SEARCH_MIN_GEQ is not used). The practical usefulness of partial functions is that they dispense with the need to add a "non existent" or "unused" element in the input sets in order to implant them as total functions. The implementation of L_PFNC performs these elements by using the seldom used MAXINT value.

MACHINE PARAMETERS

L_PFNC(LPF_minval,LPF_maxval,LPF_maxidx): LPF_minval,LPF_maxval is the input set of the function, 0..LPF_maxidx is the source set. LPF_minval, LPF_maxval,
LPF_maxidx must be NATs: this machine does not allow negative values. Moreover, LPF_minval \leq LPF_maxval and 1 \leq LPF_maxidx; as well as LPF_maxval < MAXINT: This is because MAXINT is used to indicate that the corresponding index is not part of the partial function. Again to simplify implementation, it is also illegal to have LPF_maxidx = MAXINT.

**VAL_PFNC**
- **syntax**  
  \( vv \leftarrow \text{VAL\_PFNC}(ii) \)
- **preconditions**  
  ii must be in the partial function domain
- **outputs**  
  vv is in LPF_minval..LPF_maxval, it is the value of the array at position ii.

**STR_PFNC**
- **syntax**  
  \( \text{STR\_PFNC}(ii,vv) \)
- **preconditions**  
  ii must be in 0..LPF_maxidx and vv be in LPF_minval..LPF_maxval.

The partial function is overloaded by \( \{ii \mapsto vv\} \).

**XST_PFNC**
- **syntax**  
  \( bb \leftarrow \text{XST\_PFNC}(ii) \)
- **outputs**  
  bb is TRUE if ii is in the domain of the function, FALSE if not.

**RMV_PFNC**
- **syntax**  
  \( \text{RMV\_PFNC}(ii) \)
- **preconditions**  
  ii must be in the domain of the partial function.

The \( \{ii\mapsto\text{pfnc\_vrb}(ii)\} \) pair is removed from the partial function pfnc_vrb.

**SET_PFNC**
- **syntax**  
  \( \text{SET\_PFNC}(ii,jj,vv) \)
- **preconditions**  
  ii..jj must be included in 0..LPF_maxidx and vv be in LPF_minval..LPF_maxval. ii and jj must be NATs.

The partial function is overloaded by \( (ii..jj)\times vv \). If ii > jj, ii..jj is blank and the partial function is not modified, but it is still necessary for ii and jj to be NATs.

**SWAP_PFNC**
- **syntax**  
  \( \text{SWAP\_PFNC}(ii,jj) \)
- **preconditions**  
  ii..jj must be in the domain of the partial function.

The ii and jj elements in the array are exchanged.
RIGHT_SHIFT_PFNC

**syntax**
RIGHT_SHIFT_PFNC(ii,jj,nn)

**preconditions**
ii, jj, nn must be in 0..LPF_maxidx, with ii≤jj and jj+nn≤LPF_maxidx to allow the right shift by nn spaces. It is also necessary for ii..jj to be included in the domain of the partial function.

The ii+nn..jj+nn part is overloaded by a copy of the ii..jj part in the partial function (shift by nn spaces to the right).

LEFT_SHIFT_PFNC

**syntax**
LEFT_SHIFT_PFNC(ii,jj,nn)

**preconditions**
ii, jj must be in 0..LPF_maxidx, with ii≤jj. nn must be a NAT with nn ≤ ii to allow the left shift by nn spaces. In addition it is necessary for ii..jj to be included in the domain of the partial function.

The ii-nn..jj-nn part is overloaded by a copy of the ii..jj part in the partial function (shift left by nn spaces).

SEARCH_MAX_EQL_PFNC

**syntax**
rr, bb ← SEARCH_MAX_EQL_PFNC(ii,jj,vv)

**preconditions**
ii and jj must be in 0..LPF_maxidx, ii≤jj and vv be in LPF_minval..LPF_maxval.

**outputs**
TRUE if vv was found, FALSE if not, rr is a NAT, if bb = TRUE, then rr is the largest index, the image of which by the partial function is vv.

Search for an array element that equals vv, by scanning the ii..jj part, starting from jj.

SEARCH_MIN_EQL_PFNC

**syntax**
rr, bb ← SEARCH_MIN_EQL_PFNC(ii,jj,vv)

**preconditions**
ii and jj must be in 0..LPF_maxidx, ii≤jj and vv be in LPF_minval..LPF_maxval.

**outputs**
TRUE if vv was found, FALSE if not, rr is a NAT, if bb = TRUE, then rr is the smallest index, the image of which by the partial function is vv.

Search for an array element that equals vv, by scanning the ii..jj part starting from ii.

REVERSE_PFNC

**syntax**
REVERSE_PFNC(ii,jj)

**preconditions**
ii and jj must be in 0..LPF_maxidx, and ii..jj must be included in the domain of the partial function.

Reverse the order of the elements in the ii..jj portion of the partial function.

ASCENDING_SORT_PFNC

**syntax**
ASCENDING_SORT_PFNC(ii,jj)

**preconditions**
ii and jj must be in 0..LPF_maxidx, and ii..jj must be included in the domain of the partial function.

Shift sort, in ascending order (the smallest first) in the ii..jj portion.
IMPORTS REQUIRED

(Instances to import as the implementation tree for this library machine
sees them with SEES) BASIC_ARITHMETIC; BASIC_BOOL.

WARNING: The implementation of this machine creates the default instance for the
BASIC_ARRAY_VAR machine (clause IMPORTS BASIC_ARRAY_VAR(...)). Therefore
if another instance is necessary, it must be given a different instance name (for example:
i1.BASIC_ARRAY_VAR).
5.6 L_SEQUENCE: Creating a Sequence

OPERATIONS

LEN_SEQ returns the current size of the sequence.
IS_FULL_SEQ shows whether the sequence is full (size = LS_maxsize).
IS_INDEX_SEQ shows whether ii is a valid index.
VAL_SEQ value of an element in the sequence.
FIRST_SEQ returns the first element in the sequence.
LAST_SEQ returns the last element in the sequence.
PUSH_SEQ adds vv to the end of the sequence.
POP_SEQ removes the last element from the sequence (its value is lost).
STR_SEQ changes the value of an element in the sequence.
RMV_SEQ removes an element from the middle of the sequence.
INS_AFT_SEQ inserts vv right after index ii.
CLR_SEQ clears the sequence.
TAIL_SEQ removes the first element from the sequence.
KEEP_SEQ only keeps the nn first elements in the sequence.
CUT_SEQ cuts the nn first elements from the sequence.
PART_SEQ only keeps the ii..jj portion in the sequence.
REV_SEQ reverses the order of the elements in the sequence.
FIND_FIRST_SEQ searches for vv in the sequence, starting from the beginning.
FIND_LAST_SEQ searches for vv in the sequence, starting from the end.

EXAMPLE
The example below shows the use of L_SEQUENCE for a listed set.

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>m1_1</td>
</tr>
<tr>
<td>SETS</td>
<td></td>
</tr>
<tr>
<td>ST = {classic, baroque}</td>
<td>L_SEQUENCE(10,ST)</td>
</tr>
<tr>
<td>VARIABLES</td>
<td></td>
</tr>
<tr>
<td>vv</td>
<td>seq_vrb := vv</td>
</tr>
<tr>
<td>INARIANT</td>
<td>(seq_vrb is the variable in L_SEQUENCE)</td>
</tr>
<tr>
<td>vv ∈ seq(ST)</td>
<td>seq_vrb = vv</td>
</tr>
<tr>
<td>size(vv) ≤ 10</td>
<td>INITIALISATION</td>
</tr>
<tr>
<td>vv := [baroque, baroque]</td>
<td>PUSH_SEQ(baroque) (L_SEQUENCE guarantees that the sequence is empty at the start)</td>
</tr>
</tbody>
</table>

END
DESCRIPTION
L_SEQUENCE provides a sequence type variable, the maximum size of which is a machine parameter. Conventional search and shift functions are provided for the practical use of this sequence. This answers the frequent problem in programming applications which is to maintain a list with no blanks.

MACHINE PARAMETERS
L_SEQUENCE(LS_maxsize,LS_VALUE): the variable is a sequence of LS_VALUE elements, with a maximum size that is LS_maxsize.

LEN_SEQ
  syntax  nn ← LEN_SEQ
  outputs 0..LS_maxsize
Returns the current size of the sequence.

IS_FULL_SEQ
  syntax  bb ← IS_FULL_SEQ
  outputs bb is TRUE if the sequence is full, FALSE if not.
Specifies whether the sequence is full (size = LS_maxsize).

IS_INDEX_SEQ
  syntax  bb ← IS_INDEX_SEQ(ii)
  preconditions ii must be a NAT.
  outputs bb is TRUE if ii is an index in the sequence, FALSE if not.
Specifies whether ii is a valid index.

VAL_SEQ
  syntax  vv ← VAL_SEQ(ii)
  preconditions ii must be an index in the sequence (ii ∈ 1..size(seq_vrb)).
  outputs vv is the value of the ii-ith element (vv ∈ VALUE).
Value of an element in the sequence.

FIRST_SEQ
  syntax  vv ← FIRST_SEQ
  preconditions the sequence must not be empty.
  outputs vv is the value of the first element (vv ∈ VALUE).
Returns the first element in the sequence.

LAST_SEQ
  syntax  vv ← LAST_SEQ
  preconditions the sequence must not be empty.
  outputs vv is the value of the last element (vv ∈ VALUE).
Returns the last element in the sequence.
**DESCRIPTION OF LIBRARY MACHINES**

**PUSH_SEQ**

*Syntax:* PUSH_SEQ(vv)

*Preconditions:* vv must be in VALUE and the sequence must not be full.

Add vv at the end of the sequence.

**POP_SEQ**

*Syntax:* POP_SEQ

*Preconditions:* the sequence must not be empty.

Removes the last element from the sequence (its value is lost).

**STR_SEQ**

*Syntax:* STR_SEQ(ii,vv)

*Preconditions:* vv must be in VALUE and ii must be a valid index for the sequence.

Changes the value of an existing element in the sequence.

**RMV_SEQ**

*Syntax:* RMV_SEQ(ii)

*Preconditions:* ii must be a valid index in the sequence.

Removes an element from the middle of the sequence.

**INS_AFT_SEQ**

*Syntax:* INS_AFT_SEQ(ii,vv)

*Preconditions:* vv must be in VALUE and ii must be a valid index for the sequence. The sequence must not be full.

Inserts vv right after index ii.

**CLR_SEQ**

*Syntax:* CLR_SEQ

Clears the sequence.

**TAIL_SEQ**

*Syntax:* TAIL_SEQ

*Preconditions:* the sequence must not be empty.

Removes the first element from the sequence.

**KEEP_SEQ**

*Syntax:* KEEP_SEQ(nn)

*Preconditions:* nn must be a NAT.

Only retains the nn first elements in the sequence. For nn = size(seq_vrb), this operation does not take action.
CUT_SEQ

**syntax**  
CUT_SEQ(nn)

**preconditions**  
nn must be a NAT.

Deletes the nn first elements from the sequence. For nn = size(seq_vrb), this operation is equivalent to CLR_SEQ.

PART_SEQ

**syntax**  
PART_SEQ(ii, jj)

**preconditions**  
ii and jj must be non null NATs, with ii \leq jj.

Only retains the ii..jj portion in the sequence. ii..jj may not be included in the domain of the sequence.

REV_SEQ

**syntax**  
REV_SEQ

Reverses the order of the elements in the sequence. Applies even for sequences that are empty or of size 1.

FIND_FIRST_SEQ

**syntax**  
bb, ii ← FIND_FIRST_SEQ(vv)

**preconditions**  
vv must be in VALUE.

**outputs**  
bb is TRUE if vv is in the sequence, FALSE if not. ii belongs to the range 1..LS_maxsize, if bb = TRUE, then it indicates the first position equal to vv.

Search for vv in the sequence, starting from the start.

FIND_LAST_SEQ

**syntax**  
bb, ii ← FIND_LAST_SEQ(vv)

**preconditions**  
vv must be in VALUE.

**outputs**  
bb is TRUE if vv is in the sequence, FALSE if not. If bb = TRUE, ii belongs to the range 1..LS_maxsize and indicates the last position equal to vv.

Search for vv in the sequence, starting from the end.

IMPORTS REQUIRED

(instances to import as the implementation tree for this library machine
sees them with SEES) BASIC_ARITHMETIC; BASIC_BOOL.

WARNING: The implementation of this machine creates the default instance for the BASIC ARRAY VAR machine (clause IMPORTS BASIC ARRAY VAR(...)). Therefore if another instance is required, it must be given a non blank instance name (for example: i1.BASIC ARRAY VAR).
5.7 L_SET: Creating a Set

OPERATIONS

CARD_SET returns the cardinal for the set.
IS_FULL_SET identifies whether the set is full (card = LSET_maxsize).
FIND_SET finds an element in the set.
RMV_SET removes an element from the set.
INS_SET inserts an element in the set.
CLR_SET removes all of the elements from the set.
IS_INDEX_SET identifies whether a number is a valid index.
VAL_SET value of an element in the set.

EXAMPLE

The example below shows the use of L_SET on a listed set.

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>ml_1</td>
</tr>
<tr>
<td>SETS</td>
<td>REFINES</td>
</tr>
<tr>
<td>ST = {cat, dog, bird}</td>
<td>ml</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>IMPORTS</td>
</tr>
<tr>
<td>vv</td>
<td>L_SET(3,ST)</td>
</tr>
<tr>
<td>INVARINT</td>
<td>INVARINT</td>
</tr>
<tr>
<td>vv ⊆ ST</td>
<td>(set_vrb is the variable in L_SET)</td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>INITIALISATION</td>
</tr>
<tr>
<td>vv := {cat,bird}</td>
<td>(L_SET ensures that the set is empty at the start)</td>
</tr>
<tr>
<td>END</td>
<td>INS_SET(cat);</td>
</tr>
<tr>
<td></td>
<td>INS_SET(bird)</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>

DESCRIPTION

L_SET creates a set that is modeled by an injective sequence type variable, set_vrb the maximum size of which is a machine parameter. It offers functions to search for, add and delete elements.

The use of an injective sequence type variable enables easy access to each element of the set via an index. The user can therefore create loops by using the CARD_SET and VAL_SET functions. This would not have been possible if the variable directly represented the set.

WARNING: The user must add the gluing invariant ran(set_vrb) = var_locale to his machine in order to link his set variable with the L_SET machine state.
MACHINE PARAMETERS

L_Set (LSET_maxsize, LSET_VALUE): the variable is an injective sequence of elements from LSET_VALUE, with a maximum size LSET_maxsize.

CARD_SET

\[ \text{syntax} \quad \text{nn} \leftarrow \text{CARD_SET} \]
\[ \text{output} \quad \text{nn is the size of the set (the cardinal of ran (set_vrb))}. \] Therefore, \text{nn} belongs to 0.. LSET_maxsize

Returns the size of the set.

IS_FULL_SET

\[ \text{syntax} \quad \text{bb} \leftarrow \text{IS_FULL_SET} \]
\[ \text{output} \quad \text{bb is TRUE if the set is full, FALSE if not}. \]
States whether the set is full (size = LSET_maxsize).

IS_INDEX_SET

\[ \text{syntax} \quad \text{bb} \leftarrow \text{IS_INDEX_SET}(\text{ii}) \]
\[ \text{preconditions} \quad \text{ii must be a NAT}. \]
\[ \text{outputs} \quad \text{bb is TRUE if ii is an index of the set, FALSE if not}. \]
States whether \text{ii} is a valid index.

VAL_SET

\[ \text{syntax} \quad \text{vv} \leftarrow \text{VAL_SET}(\text{ii}) \]
\[ \text{preconditions} \quad \text{ii must be an index of the set (ii } \in \text{1..size(seq_vrb))}. \]
\[ \text{outputs} \quad \text{vv is the value of the ii-the element (vv } \in \text{LSET_VALUE}). \]
Value of an element of the set.

FIND_SET

\[ \text{syntax} \quad \text{bb, ii} \leftarrow \text{FIND_SET(vv)} \]
\[ \text{preconditions} \quad \text{vv must be in LSET_VALUE}. \]
\[ \text{outputs} \quad \text{bb is TRUE if vv is in the set, FALSE if not. ii is a NAT, if bb = TRUE, then it indicates the position of element vv}. \]
Search for \text{vv} in the set.

RMV_SET

\[ \text{syntax} \quad \text{RMV_SET(vv)} \]
\[ \text{preconditions} \quad \text{vv must be in the set}. \]
Removes an element from the set.
INS_SET

syntax INS_SET(vv)

preconditions vv must be in LSET_VALUE.

Adds an element to the end of the set, if it is not already in it, if not it does nothing.

CLR_SET

syntax CLR_SET

Clears the set.
5.8 L_ARRAY1_RANGE: A Range of Arrays of the Same Size, with Numerical Indexes

OPERATIONS

- **VAL_ARR_RGE**: value of an element (promoted operation).
- **STR_ARR_RGE**: write an element (promoted operation).
- **COP_ARR_RGE**: copy an array to another (promoted operation).
- **CMP_ARR_RGE**: compare two arrays (promoted operation).
- **DUP_ARR_RGE**: duplicate the same array to a series of arrays.
- **SET_ARR_RGE**: copy the same value to an index interval in one of the arrays.
- **PCOP_ARR_RGE**: copy part of one array to a different array, to a given position.
- **PCMP_ARR_RGE**: search for the first element that is different between two parts of two arrays. A Boolean element indicates whether this element was found and, in this case, the index of this element in returned.

EXAMPLE

Using SET_ARR_RGE and DUP_ARR_RGE to initialize a set of arrays:

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>m1_1</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>REFINES</td>
</tr>
<tr>
<td>vv</td>
<td>m1</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>IMPORTS</td>
</tr>
<tr>
<td>vv ∈ 0..20 → 0..10 → 0..255</td>
<td>i1.L_ARRAY1_RANGE(0,20,10,0..255)</td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>INVARIANT</td>
</tr>
<tr>
<td>vv := (0..20)×{(0..10)×{5}}</td>
<td>i1.arr_rge = vv</td>
</tr>
<tr>
<td>END</td>
<td>INITIALISATION</td>
</tr>
<tr>
<td></td>
<td>i1.SET_ARR_RGE(0,0,10,5);</td>
</tr>
<tr>
<td></td>
<td>i1.DUP_ARR_RGE (1,20,0)</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>

DESCRIPTION

L_ARRAY1_RANGE is used in place of BASIC_ARRAY_RANGE, so that a range of arrays may create a set of function type abstract variables when operations are required to perform complete array initialization.

It also allows performing operations that use parts of two different arrays.

The index and range sets are intervals so that it is possible to indicate only portions of these sets without listing all elements involved.
MACHINE PARAMETERS

L_ARRAY1_RANGE (LAUR_minrge, LAUR_maxrge, LAUR_maxidx, LAUR_VALUE):
The range interval is the LAUR_minrge..LAUR_maxrge interval, the index interval is
0..LAUR_maxidx and LAUR_VALUE is the set of possible values.

VAL.ARR.RGE
syntax vv ← VAL.ARR.RGE (range, index)
preconditions range must belong to LAUR_minrge..LAUR_maxrge and index belong to
0..LAUR_maxidx.
outputs vv is a LAUR_VALUE, it is the value of the array range at the index
position.

STR.ARR.RGE
syntax STR.ARR.RGE (range, index, value)
preconditions range must belong to LAUR_minrge..LAUR_maxrge, index belong to
0..LAUR_maxidx and value belong to LAUR_VALUE.
The value data value is stored in the indexed array range.

COP.ARR.RGE
syntax COP.ARR.RGE (dest, src)
preconditions dest and src are in LAUR_minrge..LAUR_maxrge
The src array is copied to the dest array.

CMP.ARR.RGE
syntax bb ← CMP.ARR.RGE (range1, range2)
preconditions range1 and range2 are in LAUR_minrge..LAUR_maxrge
outputs bb is a BOOL element that is TRUE if the two arrays are equal and
FALSE if not.

SET.ARR.RGE
syntax SET.ARR.RGE (range, ii, jj, vv)
preconditions range must belong to LAUR_minrge..LAUR_maxrge, ii..jj be included in
0..LAUR_maxidx and vv belong to LAUR_VALUE. For implementation
reasons, jj must also be different from MAXINT.
The vv value is stored in the array range for all index values between ii and jj. If ii>jj,
the array remains unchanged.

DUP.ARR.RGE
syntax DUP.ARR.RGE (dest1, dest2, src)
preconditions dest1, dest2, src are in LAUR_minrge..LAUR_maxrge. For implementa-
tion reasons, dest2 must also be different from MAXINT.
The src array is duplicated in all of the arrays of the dest1..dest2 interval.
**PCOP_ARR_RGE**

*Syntax*

PCOP_ARR_RGE (dest, idx_dst, src, ii, jj)

*Preconditions*

`dest` and `src` must be different elements of `LAUR_minrge..LAUR_maxrge`, `ii..jj` be a non empty interval of `0..LAUR_maxidx`, `idx_dst` belong to `0..LAUR_maxidx`, `jj` be different from `MAXINT` and `idx_dst + jj - ii` belong to `0..LAUR_maxidx` (condition necessary to ensure that the copy does not overflow).

The `ii..jj` part of the `src` array is copied to the `dest` array, from the `idx_dst` index.

**PCMP_ARR_RGE**

*Syntax*

`idx, bb ← PCMP_ARR_RGE (rng2, idx2, rng1, ii, jj)`

*Preconditions*

`rng1` and `rng2` must belong to `LAUR_minrge..LAUR_maxrge`, `ii..jj` be a non empty interval of `0..LAUR_maxidx`, `idx2` and `idx2 + jj - ii` are in `0..LAUR_maxidx`.

The `ii..jj` part of array `rng1` is compared to the part with the same size in the `rng2` array. The `idx2 + jj - ii ∈ 0..LAUR_maxidx` condition guarantees that this comparison is possible. `bb` is a Boolean element that is `FALSE` if the two parts are equal and `TRUE` if they are different. In the latter case, `idx` and `index` are the first element that is different from `ii..jj`.

**IMPORTS REQUIRED**

(Instances to import as the implementation tree for this library machine sees them with SEES) `BASIC_ARITHMETIC`, `BASIC_BOOL`.

**WARNING:** The implementation of this machine creates the default instance for the `BASIC_ARRAY_RANGE` machine (IMPORTS `BASIC_ARRAY_RANGE(...)` clause). Therefore if another instance is necessary, it must be given the name of a non empty instance (for example: `i1.BASIC_ARRAY_RANGE`).
5.9 L_ARRAY3_RANGE: A Range of Arrays of the Same Size, with Non Ordered Values, Maximum Operations

OPERATIONS

VAL_ARR_RGE  value of an element (promoted operation).
STR_ARR_RGE  write an element (promoted operation).
COP_ARR_RGE  copy an array to another (promoted operation).
CMP_ARR_RGE  compare two arrays (promoted operation).
DUP_ARR_RGE  duplicate the same array to a set of arrays (promoted operation).
SET_ARR_RGE  copy the same value to an index interval in one of the arrays (promoted operation).
PCOP_ARR_RGE  copy part of one of the arrays to a different array, at a given position (promoted operation).
PCMP_ARR_RGE  search for the first element that is different between two parts of two arrays. A Boolean element indicates whether this element was found and, in this case, the index of this element is returned (promoted operation).
SWAP_RGE  exchange two array elements.
RIGHT SHIFT_RGE  shift part of an array to the large index.
LEFT SHIFT_RGE  shift part of an array to the small index.
SEARCH_MAX_EQL_RGE  search for the last element that equals a value in part of an array.
SEARCH_MIN_EQL_RGE  search for the first element that equals a value in part of an array.
REVERSE_RGE  reverse the order of the elements in part of an array.

EXAMPLE

The following example is a machine that represents the color assigned to 101 dots for each array in a range; this color may be red, green or blue for each dot. A operation enables finding a red dot in an array.
MACHINE  
m3  
SETS  
COLOR = \{red, green, blue\}  
VARIABLES  
color  
INVARIANT  
color \in [0..10] \rightarrow (0..100 \rightarrow \text{COLOR})  
INITIALISATION  
color := (0..10) \times \{(0..100) \times \{\text{red}\}\}  
OPERATIONS  
ii, bb \leftarrow \text{find_red}(\text{rng}) = \text{PRE}  
\text{rng} \in 0..10 \land  
\text{rouge} \in \text{ran}(\text{color}(\text{rng}))  
\text{THEN}  
ii := \text{color}(\text{rng})^{-1}[\{\text{red}\}]  
bb := \text{BOOL}  
\text{END}  
END

IMPLEMENTATION  
m3_1  
REFINES  
m3  
IMPORTS  
i1.L ARRAY3 RANGE(0,10,100,\text{COLOR})  
INVARIANT  
i1.arr_rge = \text{color}  
INITIALISATION  
i1.arr_rge := \text{color}  
OPERATIONS  
ii, bb \leftarrow \text{find}(\text{rng}) = \text{VAR} \ bb \text{ IN}  
ii, bb \leftarrow  
\ i1.\text{SEARCH}_\text{MAX}_\text{EQL}_\text{RGE}(\text{rng},0,100,\text{red})  
\text{END}  
\text{END}

DESCRIPTION
L ARRAY3 RANGE is the most complete of the two dimensional array machines with no constraint[1]. This makes it possible to create arrays with values that are the elements of an enumerated set, while retaining access to complete operations such as reversing the order of elements.

The operation that is not available is the one that would require an order relation on the elements in the array: \text{sort}.

MACHINE PARAMETERS
L ARRAY3 RANGE (LATR_minrge, LATR_maxrge, LATR_maxidx, LATR_VALUE): The range interval is LATR_minrge..LATR_maxrge, the index interval 0..LATR_maxidx and LATR_VALUE is the set of possible values.

VAL_ARR_RGE

syntax  
vv \leftarrow \text{VAL}_\text{ARR}_\text{RGE} (\text{range}, \text{index})  
preconditions  
\text{range} \text{ must belong to LATR_minrge..LATR_maxrge, index belong to 0..LATR_maxidx}  
outputs  
vv \text{ is a LATR_VALUE, it is the value of the array range at the index position.}

STR_ARR_RGE

syntax  
\text{STR}_\text{ARR}_\text{RGE} (\text{range}, \text{index}, \text{value})  
preconditions  
\text{range} \text{ must belong to LATR_minrge..LATR_maxrge, index belong to 0..LATR_maxidx and value belong to LATR_VALUE.}  
The LATR_VALUE value is stored in the array range in the index.

\text{L ARRAY5 RANGE} can only have a finite integer set as range.
**COP** _ARR_ RGE

**syntax**  
COP ARR RGE (dest, src)

**preconditions**  
dest and src are in LATR_minrge..LATR_maxrge

The src array is copied to the dest array.

**CMP** _ARR_ RGE

**syntax**  
bb ← CMP ARR RGE (range1, range2)

**preconditions**  
range1 and range2 are in LATR_minrge..LATR_maxrge

**outputs**  
bb is an BOOL that equals TRUE if the two arrays are equal and FALSE if not.

**SET** _ARR_ RGE

**syntax**  
SET ARR RGE (range,ii,jj,vv)

**preconditions**  
range must belong to LATR_minrge..LATR_maxrge, ii..jj be included in 0..LATR_maxidx and vv belong to LATR_VALUE. For implementation reasons, jj must also be different to MAXINT.

Value vv is stored in the array range for all indexes in the range from ii to jj. If ii>jj, the array remains unchanged.

**DUP** _ARR_ RGE

**syntax**  
DUP ARR RGE (dest1, dest2, src)

**preconditions**  
dest1, dest2, src are in LATR_minrge..LATR_maxrge. For implementation reasons, dest2 must also be different to MAXINT.

The src array is duplicated in all of the arrays of the dest1..dest2 interval.

**PCOP** _ARR_ RGE

**syntax**  
PCOP ARR RGE (dest, idx_dst, src,ii,jj)

**preconditions**  
dest and src must belong to LATR_minrge..LATR_maxrge and be different, ii..jj be a non empty interval of 0..LATR_maxidx, idx_dst belong to 0..LATR_maxidx, jj be different from MAXINT and idx_dst + jj - ii belong to 0..LATR_maxidx (necessary condition to avoid copy overflow).

The ii..jj part in the src array is copied to the dest array, from the idx_dst index.

**PCMP** _ARR_ RGE

**syntax**  
idx, bb ← PCMP ARR RGE (rng2,idx2,rng1,ii,jj)

**preconditions**  
rng1 and rng2 are in LATR_minrge..LATR_maxrge, ii..jj is a non empty interval of 0..LATR_maxidx idx2 and idx2 + jj-ii are in 0..LATR_maxidx.

The ii..jj part of array rng1 is compared with the part with the same size in array rng2. The idx2 + jj-ii ∈ 0..LATR_maxidx condition guarantees that this comparison is possible.

bb is a Boolean element that is FALSE if the two parts are equal and TRUE if they are different. In the latter case, idx is the index of the first element that is different to ii..jj.
**SWAP_RGE**

*Syntax:* \( \text{SWAP}_\text{RGE} (\text{rng}, \text{ii}, \text{jj}) \)

*Preconditions:* \( \text{rng} \) is in \( \text{LATR}_{\text{minrge}}..\text{LATR}_{\text{maxrge}} \), \( \text{ii} \) and \( \text{jj} \) in \( 0..\text{LATR}_{\text{maxidx}} \).

The \( \text{ii} \) and \( \text{jj} \) elements in the array are exchanged.

**RIGHT_SHIFT_RGE**

*Syntax:* \( \text{RIGHT}_\text{SHIFT}_\text{RGE} (\text{rng}, \text{ii}, \text{jj}, \text{nn}) \)

*Preconditions:* \( \text{rng} \) must belong to \( \text{LATR}_{\text{minrge}}..\text{LATR}_{\text{maxrge}} \), \( \text{ii} \), \( \text{jj} \) and \( \text{nn} \) belong to \( 0..\text{LATR}_{\text{maxidx}} \), with \( \text{ii} \leq \text{jj} \) and \( \text{jj} + \text{nn} \leq \text{LATR}_{\text{maxidx}} \) to allow a right shift by \( \text{nn} \) spaces.

The \( \text{ii}+\text{nn}..\text{jj}+\text{nn} \) part in the \( \text{rng} \) array receives a copy of the \( \text{ii}..\text{jj} \) part of this same array (shift right by \( \text{nn} \) spaces).

**LEFT_SHIFT_RGE**

*Syntax:* \( \text{LEFT}_\text{SHIFT}_\text{RGE} (\text{rng}, \text{ii}, \text{jj}, \text{nn}) \)

*Preconditions:* \( \text{rng} \) is in \( \text{LATR}_{\text{minrge}}..\text{LATR}_{\text{maxrge}} \), \( \text{ii} \) and \( \text{jj} \) must be in \( 0..\text{LATR}_{\text{maxidx}} \), with \( \text{ii} \leq \text{jj} \). \( \text{nn} \) must be a NAT with \( \text{nn} \leq \text{ii} \) to allow the left shift by \( \text{nn} \) spaces. For implementation reasons, \( \text{jj} \) must be equal to \( \text{MAXINT} \).

The \( \text{ii}+\text{nn}..\text{jj}+\text{nn} \) part of the \( \text{rng} \) array receives a copy of the \( \text{ii}..\text{jj} \) part of this same array (shift left by \( \text{nn} \) spaces).

**SEARCH_MAX_EQL_RGE**

*Syntax:* \( \text{rr}, \text{bb} \leftarrow \text{SEARCH}_\text{MAX}_\text{EQL}_\text{RGE} (\text{rng}, \text{ii}, \text{jj}, \text{vv}) \)

*Preconditions:* \( \text{rng} \) must be in \( \text{LATR}_{\text{minrge}}..\text{LATR}_{\text{maxrge}} \), \( \text{ii} \) and \( \text{jj} \) must be in \( 0..\text{LATR}_{\text{maxidx}} \), \( \text{ii} \leq \text{jj} \) and \( \text{vv} \) must belong to \( \text{LATR}_{\text{VALUE}} \).

*Outputs:* TRUE if \( \text{vv} \) was found, FALSE if not. \( \text{rr} \) is a NAT, if \( \text{bb} = \text{TRUE} \) then \( \text{rr} \) is the largest index in the \( \text{rng} \) array equal to \( \text{vv} \).

Search for an element in an array equal to \( \text{vv} \), by scanning the \( \text{ii}..\text{jj} \) part starting from \( \text{jj} \).

**SEARCH_MIN_EQL_RGE**

*Syntax:* \( \text{rr}, \text{bb} \leftarrow \text{SEARCH}_\text{MIN}_\text{EQL}_\text{RGE} (\text{rng}, \text{ii}, \text{jj}, \text{vv}) \)

*Preconditions:* \( \text{rng} \) must belong to \( \text{LATR}_{\text{minrge}}..\text{LATR}_{\text{maxrge}} \), \( \text{ii} \) and \( \text{jj} \) belong to \( 0..\text{LATR}_{\text{maxidx}} \), \( \text{ii} \leq \text{jj} \) and \( \text{vv} \) belong to \( \text{LATR}_{\text{VALUE}} \).

*Outputs:* TRUE if \( \text{vv} \) was found, FALSE if not. \( \text{rr} \) is a NAT, if \( \text{bb} = \text{TRUE} \), then \( \text{rr} \) is the smallest index in the \( \text{rng} \) array equal to \( \text{vv} \).

Search for an element in an array that is equal to \( \text{vv} \), by scanning the \( \text{ii}..\text{jj} \) part starting from \( \text{ii} \).

**REVERSE_RGE**

*Syntax:* \( \text{REVERSE}_\text{RGE}(\text{rng}, \text{ii}, \text{jj}) \)

*Preconditions:* \( \text{rng} \) must belong to \( \text{LATR}_{\text{minrge}}..\text{LATR}_{\text{maxrge}} \), \( \text{ii} \) and \( \text{jj} \) belong to \( 0..\text{LATR}_{\text{maxidx}} \).

Reversing the order of elements in the \( \text{ii}..\text{jj} \) part of the \( \text{rng} \) array.
IMPORTS REQUIRED

(instances to import as the implementation tree for this library machine
sees them with SEES) BASIC_ARITHMETIC; BASIC_BOOL.

WARNING: The implementation of this machine creates the default instance for the
BASIC_ARRAY_RANGE machine (IMPORTS BASIC_ARRAY_RANGE(...) clause).
Therefore if another instance is necessary, it must be given a non empty instance name
(for example: i1.BASIC_ARRAY_RANGE)
5.10 L_ARRAY5_RGE: Range of Arrays of the Same Size, with Ordered Value Numerical Indexes, Sort Operation

OPERATIONS

- **VAL_ARR_RGE**: value of an element (promoted operation).
- **STR_ARR_RGE**: write an element (promoted operation).
- **COP_ARR_RGE**: copy an array to another (promoted operation).
- **CMP_ARR_RGE**: compare two arrays (promoted operation).
- **DUP_ARR_RGE**: duplicate the same array to a set of arrays (promoted operation).
- **SET_ARR_RGE**: copy the same value to an index range in one of the arrays (promoted operation).
- **PCOP_ARR_RGE**: copy part of one of the arrays to a different array, to a given position (promoted operation).
- **PCMP_ARR_RGE**: search for the first different element between two parts of two arrays. A Boolean element indicates whether this element was found and, in this case, the index of this element is returned (promoted operation).
- **SWAP_RGE**: exchange two elements in an array (promoted operation).
- **RIGHT_SHIFT_RGE**: shift a part of an array to the large index (promoted operation).
- **LEFT_SHIFT_RGE**: shift a part of an array to the small index (promoted operation).
- **SEARCH_MAX_EQL_RGE**: search for the last element that is equal to a value in an array range (promoted operation).
- **SEARCH_MIN_EQL_RGE**: search for the first element that equals a value in an array range (promoted operation).
- **REVERSE_RGE**: reverse the order of the elements of a part of an array (promoted operation).
- **SEARCH_MIN_GEQ_RGE**: search for the first element that exceeds a value in an array range.
- **ASCENDING_SORT_RGE**: sort part of an array and arrange in ascending order.
## Example

### Machine

**m5**

**Variables**

$v_v$

**Invariant**

$v_v \in 0..20 \rightarrow (0..4 \rightarrow 0..255) \land 
\forall (x, y). (x \in 0..20 \land y \in 0..3 
\Rightarrow v_v(y)(x) >= v_v(y)(x+1))$

**Initialisation**

$v_v : (v_v \in 0..20 \rightarrow (0..4 \rightarrow 0..255) \land 
\forall (x, y). (x \in 0..20 \land y \in 0..3 
\Rightarrow v_v(y)(x) >= v_v(y)(x+1)))$

**End**

### Implementation

$m5_1$

**Refines**

$m5$

**Imports**

L\_ARRAY5\_RANGE(0,20,4,0,255)

**Invariant**

\[ arr\_rge = v_v \]

**Initialisation**

SET\_ARR\_RGE(0,0,4,50);
STR\_ARR\_RGE(0,2,10);
STR\_ARR\_RGE(0,4,30);
ASCENDING\_SORT\_RGE(0,0,4);
REVERSE\_RGE(0,0,4);
DUP\_ARR\_RGE(1,20,0)

**End**

## Description

L\_ARRAY5\_RANGE is the most complete two dimensional array machines. It especially contains a sort operation, implanted by a shift sort (fast algorithm).

### Machine Parameters

L\_ARRAY5\_RANGE (LACR\_minrge, LACR\_maxrge, LACR\_maxidx, LACR\_minval, LACR\_maxval):

LACR\_minrge..LACR\_maxrge is the set of ranges, 0..LACR\_maxidx is the set of indexes and LACR\_minval..LACR\_maxval, the set of possible values. All of the parameters must be NATs: this machine does not allow negative values.

In addition, LACR\_minrge \leq LACR\_maxrge, 1 \leq LACR\_maxidx and LACR\_minval \leq LACR\_maxval.

### VAL\_ARR\_RGE

**Syntax**

$v_v \leftarrow \text{VAL\_ARR\_RGE} (\text{range}, \text{index})$

**Preconditions**

- range must belong to LACR\_minrge..LACR\_maxrge
- index belong to 0..LACR\_maxidx.

**Outputs**

$v_v$ is a LACR\_VALUE, it is the value of the array range at the index position.

### STR\_ARR\_RGE

**Syntax**

\text{STR\_ARR\_RGE} (\text{range}, \text{index}, \text{value})

**Preconditions**

- range must be in LACR\_minrge..LACR\_maxrge
- index must be in 0..LACR\_maxidx value must belong to LACR\_VALUE.

The value of the value element is stored in the array range as an index.
**COP_ARR_RGE**

*syntax*  
COP_ARR_RGE (dest, src)

*preconditions*  
dest and src are in LACR_minrge..LACR_maxrge

The src array is copied to the dest array.

**CMP_ARR_RGE**

*syntax*  
bb ← CMP_ARR_RGE (range1, range2)

*preconditions*  
range1 and range2 are in LACR_minrge..LACR_maxrge

*outputs*  
bb is a BOOL element that is TRUE if the two arrays are equal and FALSE if not.

**SET_ARR_RGE**

*syntax*  
SET_ARR_RGE (range,ii,jj,vv)

*preconditions*  
range must belong to LACR_minrge..LACR_maxrge, ii..jj be included in 0..LACR_maxidx and vv belong to LACR_VALUE. For implementation reasons, it is also necessary that jj be different from MAXINT.

The vv value is stored in the array range for all indexes between ii and jj. If ii>jj, the array remains unchanged.

**DUP_ARR_RGE**

*syntax*  
DUP_ARR_RGE (dest1, dest2, src)

*preconditions*  
dest1, dest2, src are in LACR_minrge..LACR_maxrge. For implementation reasons, it is also necessary for dest2 to be different from MAXINT.

The src array is duplicated to all arrays for the dest1..dest2 range.

**PCOP_ARR_RGE**

*syntax*  
PCOP_ARR_RGE (dest, idx_dst, src,ii,jj)

*preconditions*  
dest and src must be different elements of LACR_minrge..LACR_maxrge, ii..jj be a non empty subset of 0..LACR_maxidx and idx_dst belong to 0..LACR_maxidx; jj is different from MAXINT and idx_dst + jj - ii belong to 0..LACR_maxidx (condition to avoid copy overflow).

The ii..jj range in the src array is copied to the dest array, for the idx_dst index.

**PCMP_ARR_RGE**

*syntax*  
idx, bb ← PCMP_ARR_RGE (rng2,idx2,rng1,ii,jj)

*preconditions*  
rng1 and rng2 are in LACR_minrge..LACR_maxrge, ii..jj is a non empty range 0..LACR_maxidx, idx2 and idx2 + jj-ii are in 0..LACR_maxidx.

The ii..jj part of the rng1 array is compared with the part of the same size in the rng2 array. The idx2 + jj-ii ∈ 0..LACR_maxidx condition guarantees that this comparison is possible. bb is a Boolean element that is FALSE if the two parts are equal and TRUE if they are different. In the latter case, idx is the index of the first element that is different from ii..jj.
SWAP_RGE

**syntax**

\texttt{SWAP\_RGE (rng,ii,jj)}

**preconditions**

rng is in LACR\_minrge..LACR\_maxrge, ii and jj in 0..LACR\_maxidx.

The ii and jj elements in the array are exchanged.

RIGHT SHIFT RGE

**syntax**

\texttt{RIGHT\_SHIFT\_RGE (rng,ii,jj,nn)}

**preconditions**

rng must belong to LACR\_minrge..LACR\_maxrge. ii, jj and nn belong to 0..LACR\_maxidx, with ii\leq jj and jj+nn\leq LACR\_maxidx to allow the shift right by nn spaces.

The ii+nn..jj+nn part of the rng array receives a copy of the ii..jj part from this same array (shift nn spaces to the right).

LEFT SHIFT RGE

**syntax**

\texttt{LEFT\_SHIFT\_RGE (rng,ii,jj,nn)}

**preconditions**

rng must belong to LACR\_minrge..LACR\_maxrge, ii and jj belong to 0..LACR\_maxidx, with ii\leq jj. nn must belong to NAT with nn \leq ii to make possible the left shift by nn spaces. For implementation reasons, jj cannot equal MAXINT.

The ii-nn..jj-nn part of the rng array receives a copy of the ii..jj part of this same array (shift nn spaces to the left).

SEARCH MAX EQL RGE

**syntax**

\texttt{rr,bb \leftarrow SEARCH\_MAX\_EQL\_RGE (rng,ii,jj,vv)}

**preconditions**

rng must belong to LACR\_minrge..LACR\_maxrge, ii and jj belong to 0..LACR\_maxidx. ii\leq jj and vv must belong to LACR\_VALUE.

**outputs**

TRUE if vv was found, FALSE if not. rr is a NAT, if bb = TRUE then rr is the largest index in the array that equals vv.

Search for an array element that equals vv, by scanning the ii..jj part starting from jj.

SEARCH MIN EQL RGE

**syntax**

\texttt{rr,bb \leftarrow SEARCH\_MIN\_EQL\_RGE (rng,ii,jj,vv)}

**preconditions**

rng must belong to LACR\_minrge..LACR\_maxrge, ii and jj belong to 0..LACR\_maxidx, ii\leq jj and vv must belong to VALUE.

**outputs**

TRUE if vv was found, FALSE if not. rr is a NAT, if bb = TRUE, then rr is the smallest index in the rng array equal to vv.

Search for an element in an array equal to vv, by scanning the ii..jj part starting from ii.

REVERSE_RGE

**syntax**

\texttt{REVERSE\_RGE(rng,ii,jj)}

**preconditions**

rng must belong to LACR\_minrge..LACR\_maxrge, ii and jj belong to 0..LACR\_maxidx.

Reverse the order of elements in the ii..jj range of the rng array.
SEARCH_MIN_EQG_RGE

**syntax**

\[ i, bb \leftarrow \text{SEARCH\_MIN\_EQG\_RGE}(\text{rng, jj, kk, vv}) \]

**preconditions**

- \text{rng} must belong to LACR\_minrge..LACR\_maxrge.
- \text{jj} and \text{kk} belong to 0..LACR\_maxidx.
- \text{jj} \leq \text{kk} and \text{vv} belong to LACR\_minval..LACR\_maxval.
- For implementation reasons, \text{kk} must be different from MAXINT.

**outputs**

- \text{bb} is a Boolean element, \text{TRUE} is an element that exceeds or is equal to the \text{vv} value found, \text{FALSE} if not.
- \text{ii} is a NAT, if \text{bb} = \text{TRUE}, then \text{ii} is the smallest index in the image array that exceeds or is equal to \text{vv}.

Search for an element that exceeds or is equal to \text{vv} in the \text{jj}..\text{kk} range, starting from \text{jj}.

ASCENDING_SORT_RGE

**syntax**

\[ \text{ASCENDING\_SORT\_RGE}(\text{rng, ii, jj}) \]

**preconditions**

- \text{rng} must belong to LACR\_minrge..LACR\_maxrge.
- \text{ii} and \text{jj} belong to 0..LACR\_maxidx.
- For implementation reasons, \text{ii} and \text{jj} must not be different from MAXINT.

Shift sort, in ascending order (starting with the smallest) on the \text{ii}..\text{jj} range in an array.

IMPORTS REQUIRED

(instances to import as the implementation tree for this library machine
sees them with SEES)

- BASIC_ARITHMETIC; BASIC_BOOL.

WARNING: The implementation of this machine creates the default instance for the BASIC\_ARRAY\_RANGE machine (IMPORTS BASIC\_ARRAY\_RANGE(...) clause).

Therefore if another instance is necessary, it must be given a non empty instance name
(for example: i1.BASIC\_ARRAY\_RANGE)
5.11 L_SEQUENCE_RANGE: Range of Sequences

OPERATIONS

LEN_SEQ_RGE gives the size of a sequence.
IS_FULL_SEQ_RGE indicates whether a sequence is full.
IS_INDEX_SEQ_RGE indicates whether an integer is in the sequence domain.
VAL_SEQ_RGE gives the value of a sequence for a valid index.
FIRST_SEQ_RGE gives the first element of a sequence.
LAST_SEQ_RGE gives the last element of a sequence.
PUSH_SEQ_RGE adds an element to a sequence.
POP_SEQ_RGE removes the last element from a sequence.
STR_SEQ_RGE changes the value of an element in a sequence.
RMV_SEQ_RGE removes an element from a sequence, the size of which is reduced by 1.
INS_SEQ_RGE adds an element to a sequence, the size of which increases by 1.
CLR_SEQ_RGE empties a sequence.
TAIL_SEQ_RGE removes the first element from a sequence.
KEEP_SEQ_RGE only retains the first N in a sequence elements.
CUT_SEQ_RGE cuts the N first elements from a sequence.
PART_SEQ_RGE only retains in a sequence the indexes between the two limit values.
REV_SEQ_RGE reverses the order of the elements in a sequence.
FIND_FIRST_SEQ_RGE searches for a value in a sequence, returns a Boolean element indicating whether it was found, and if yes, returns the smallest corresponding index.
FIND_LAST_SEQ_RGE searches for a value in a sequence, returns a Boolean element indicating whether it was found and if yes, returns the largest corresponding index.
COP_SEQ_RGE copies one of the sequences to another.
CMP_SEQ_RGE compares two sequences.
PCOP_SEQ_RGE partial copy from one sequence to another.
PCMP_SEQ_RGE partial comparison of two sequences.
EXAMPLE
The example below shows the use of L_SEQUENCE_RANGE on a numbered set.

```
MACHINE sr
SETS ST = {classical, baroque, rock, rap, funk}
VARIABLES vv
INVARIANT vv ∈ 1..5 → seq(ST) \∧
∀rr.(rr ∈ 1..5 ⇒ size(vv(rr)) ≤ 10)
INITIALISATION vv := (1..5) × {[baroque, rock, rap]}
OPERATIONS ii, bb ← trouve_rap(rng) = PRE
rng ∈ 1..5
THEN
ii := vv(rng)⁻¹[{{rap}}] ||
bb := BOOL
END
END

IMPLEMENTATION
sr_1
REFINES sr
IMPORTS s1.L_SEQUENCE_RANGE(1, 5, 10, ST)
INVARIANT
s1.seq_rge = vv
INITIALISATION
s1.CLR_SEQ_RGE(1);
s1.PUSH_SEQ_RGE(1, baroque);
s1.PUSH_SEQ_RGE(1, rock);
s1.PUSH_SEQ_RGE(1, rap);
s1.COP_SEQ_RGE(2, 1);
s1.COP_SEQ_RGE(3, 1);
s1.COP_SEQ_RGE(4, 1);
s1.COP_SEQ_RGE(5, 1)
OPERATIONS
ii, bb ← trouve_rap(rng) = BEGIN
bb, ii := s1.FIND_FIRST_SEQ_RGE(rng, rap)
END
END
```

DESCRIPTION
L_SEQUENCE_RANGE enables implementing and using a set number of sequences with a fixed maximum size. The sequence number evolves in a range that is a machine parameter, the maximum size of all of the sequences is also a machine parameter. The purpose is to be able to make comparisons and copies between these sequences directly, using an additional operation to the traditional operations on each of the sequences.

MACHINE PARAMETERS
L_SEQUENCE_RANGE (LSR_minrge, LSR_maxrge, LSR_maxsize, LSR_VALUE): the variable is a total function of LSR_minrge..LSR_maxrge in the set of VALUE sequences with a maximum size of LSR_maxsize.

LEN_SEQ_RGE
```
syntax nn ← LEN_SEQ_RGE (range)
preconditions range must belong to the LSR_minrge..LSR_maxrge range.
outputs nn is the size of the range position , nn ∈ 0..LSR_maxsize.
```
Gives the size of a sequence.
IS_FULL_SEQ_RGE

**syntax**  
\[ bb \leftarrow \text{IS_FULL_SEQ_RGE} \text{ (range)} \]

**preconditions** range must belong to the range LSR_minrge..LSR_maxrge.

**outputs** \( bb \) is TRUE if the range position sequence is full, FALSE if not.

Indicates whether a sequence is full.

IS_INDEX_SEQ_RGE

**syntax**  
\[ bb \leftarrow \text{IS_INDEX_SEQ_RGE} \text{ (range, ii)} \]

**preconditions** range must belong to the LSR_minrge..LSR_maxrge range, \( ii \) must be a NAT.

**outputs** \( bb \) is TRUE if \( ii \) is an index in the range position sequence, FALSE if not.

Identifies whether an integer is in a sequence domain.

VAL_SEQ_RGE

**syntax**  
\[ vv \leftarrow \text{VAL_SEQ_RGE} \text{ (range, ii)} \]

**preconditions** range must belong to the LSR_minrge..LSR_maxrge range, \( ii \) must be an index in the range position sequence (\( ii \in 1..\text{size (seq_rge (range))})\).

**outputs** \( vv \) is the value of the \( ii \)-th element in the range position sequence (\( vv \in \text{VALUE} \)).

Gives the value of a sequence for a valid index.

FIRST_SEQ_RGE

**syntax**  
\[ vv \leftarrow \text{FIRST_SEQ_RGE} \text{ (range)} \]

**preconditions** range must belong to the LSR_minrge..LSR_maxrge range, the range position sequence must not be empty.

**outputs** \( vv \) is the value of the first element in the range position sequence (\( vv \in \text{VALUE} \)).

Gives the first element in a sequence.

LAST_SEQ_RGE

**syntax**  
\[ vv \leftarrow \text{LAST_SEQ_RGE} \text{ (range)} \]

**preconditions** range must be in the LSR_minrge..LSR_maxrge range, the range position sequence must not be empty.

**outputs** \( vv \) is the value of the last element in the range position sequence (\( vv \in \text{VALUE} \)).

Gives the last element of a sequence.

PUSH_SEQ_RGE

**syntax**  
\[ \text{PUSH_SEQ_RGE} \text{ (range, vv)} \]

**preconditions** range must belong to the LSR_minrge..LSR_maxrge range, \( vv \) must be in LSR\_VALUE and the range position sequence cannot be full.

Adds an element to a sequence.
POP_SEQ_RGE

syntax: POP_SEQ_RGE (range)

preconditions: range must belong to the LSR_minrge..LSR_maxrge range, the range position sequence must not be empty.

Removes the last element in a sequence.

STR_SEQ_RGE

syntax: STR_SEQ_RGE (range, ii, vv)

preconditions: range must belong to LSR_minrge..LSR_maxrge, ii be a valid index in the range position sequence and vv belong to LSR_VALUE.

Change the value of an element in a sequence.

RMV_SEQ_RGE

syntax: RMV_SEQ_RGE (range, ii)

preconditions: range must belong to the LSR_minrge..LSR_maxrge range, ii must be a valid index in the range sequence.

Removes an element from a sequence, the size of which decreases by 1.

INS_AFT_SEQ_RGE

syntax: INS_AFT_SEQ_RGE (range, ii, vv)

preconditions: range must belong to the LSR_minrge..LSR_maxrge range, ii must be a valid index in the range position sequence, vv must be in LSR_VALUE, the range position sequence must not be full.

Adds an element to a sequence, the size of which increases by 1.

CLR_SEQ_RANGE

syntax: CLR_SEQ_RANGE (range)

preconditions: range must belong to the LSR_minrge..LSR_maxrge range.

Clears a sequence.

TAIL_SEQ_RGE

syntax: TAIL_SEQ_RGE (range)

preconditions: range must belong to the LSR_minrge..LSR_maxrge range and the range position sequence cannot be empty.

Removes the first element in a sequence.

KEEP_SEQ_RGE

syntax: KEEP_SEQ_RGE (range, nn)

preconditions: range must belong to the LSR_minrge..LSR_maxrge range, nn must be a NAT.

Only retains the nn first elements in a sequence. For nn = size (seq_rge(range)); this operation has no effect.
**CUT_SEQ_RGE**

**syntax**  
CUT_SEQ_RGE (range, nn)

**preconditions**  
range must belong to the LSR_minrange..LSR_maxrange range, nn must be in NAT.

Clears the sequence of its first nn elements. For nn = size(seq_rge(range)), this operation is equivalent to CLR_SEQ_RGE.

**PART_SEQ_RGE**

**syntax**  
PART_SEQ_RGE (range, ii, jj)

**preconditions**  
range must belong to the LSR_minrange..LSR_maxrange range, ii and jj must be NATs that are not null, with ii ≤ jj.

In a sequence, only retains the indexes between two limits. ii..jj may not be in the sequence domain.

**REV_SEQ_RGE**

**syntax**  
REV_SEQ_RGE (range)

**preconditions**  
range must belong to the LSR_minrange..LSR_maxrange range.

Reverses the order of the elements in a sequence.

**FIND_FIRST_SEQ_RGE**

**syntax**  
bb, ii ← FIND_FIRST_SEQ_RGE (range, vv)

**preconditions**  
range must belong to the LSR_minrange..LSR_maxrange range, vv must be in LSR_VALUE.

**outputs**  
bb is TRUE if vv is in the range position sequence, FALSE if not. ii is a NAT, if bb = TRUE, it indicates the first position that equals vv in the sequence.

Searches for a value in a sequence starting from the beginning.

**FIND_LAST_SEQ_RGE**

**syntax**  
bb, ii ← FIND_LAST_SEQ_RGE (range, vv)

**preconditions**  
range must belong to the LSR_minrange..LSR_maxrange range, vv must be in LSR_VALUE.

**outputs**  
bb is TRUE if vv is in the range position sequence, FALSE if not. ii is a NAT; if bb = TRUE, this indicates the last position that equals vv in the sequence.

Searches for a value in a sequence, starting from the end.

**COP_SEQ_RGE**

**syntax**  
COP_SEQ_RGE (dst, src)

**preconditions**  
dst and src must belong to the LSR_minrange..LSR_maxrange range.

Copy the seq_rge(src) sequence to the seq_rge(dst) sequence.
**CMP_SEQ_RGE**

*Syntax*

\[ bb \leftarrow \text{CMP\_SEQ\_RGE} (\text{rng1}, \text{rng2}) \]

*Preconditions*

\[ \text{rng1 and rng2 must belong to the LSR\_minrge..LSR\_maxrge range.} \]

*Outputs*

\[ bb \text{ is TRUE if the two rng1 and rng2 position sequences are equal, FALSE if not.} \]

Compare two sequences.

**PCOP_SEQ_RGE**

*Syntax*

\[ \text{PCOP\_SEQ\_RGE} (\text{dst}, \text{idx}, \text{src}, \text{ii}, \text{jj}) \]

*Preconditions*

\[ \text{dst and src must belong to the LSR\_minrge..LSR\_maxrge range, dst must be different from src, ii and jj must be valid indexes in the src position sequence, with } ii \leq jj \text{ and } jj \leq \text{MAXINT-1}\]

\[ \text{idx must be a valid index for the dst sequence or where the size of this sequence } +1, \text{ idx } + jj - ii \text{ belongs to the 1..LSR\_maxsize range.} \]

Copy the ii..jj part of the src position sequence to the dst position from the idx index.

**PCMP_SEQ_RGE**

*Syntax*

\[ \text{idx, bb } \leftarrow \text{PCMP\_SEQ\_RGE} (\text{rng1}, \text{ii}, \text{jj}, \text{rng2}, \text{kk}) \]

*Preconditions*

\[ \text{rng1 and rng2 must be in the LSR\_minrge..LSR\_maxrge range, ii and jj must be valid indexes in the rng1 and ii } \leq jj \text{ position sequences, kk must be a valid index in the rng2 position sequence, (kk } + jj - ii \text{) must be a valid index in the rng2 position sequence.} \]

*Output*

\[ bb \text{ is TRUE if there is an element of the ii..jj part in the seq\_rge (rng1) sequence that is different to the kk.. (kk } + jj - ii \text{) part of the seq\_rge (rng2) sequence, FALSE if not. idx is a NAT if bb is TRUE, the idx represents the index of the first element that is different in the seq\_rge (rng1 } \in ii..jj \text{) sequence.} \]

Partial comparison of two sequences.

**IMPORTS REQUIRED**

(instances to import as the implementation tree for this library machine sees them with SEES)

**BASIC_ARITHMETIC**

**BASIC_BOOL**.

WARNING: The implementation of this machine creates the default instance for the **BASIC\_ARRAY\_RANGE** and **BASIC\_ARRAY\_VAR** machines. Therefore, if other instances are required they must be given a name that is not blank.

(example: i1.BASIC\_ARRAY\_RANGE).
5.12 L_ARRAY_COLLECTION: collection of arrays of the same size

OPERATIONS

CRE_ARR_COL  returns a Boolean element that indicates that there remains an array available in the collection and gives the index of this available array.

DEL_ARR_COL  releases the specified array.

VAL_ARR_COL  read an element from one of the valid arrays.

STR_ARR_COL  write an element from one of the valid arrays.

COP_ARR_COL  copy one of the arrays to another.

CMP_ARR_COL  compare two arrays.

EXAMPLE

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>M1_1</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>REFINES</td>
</tr>
<tr>
<td>ii1,ii2 ← initialise_array(x) = PRE vv ∈ 1..10</td>
<td>M1</td>
</tr>
<tr>
<td>THEN</td>
<td>IMPORTS</td>
</tr>
<tr>
<td>ii1:∈ NAT</td>
<td></td>
</tr>
<tr>
<td>ii2:∈ NAT</td>
<td>OPERATIONS</td>
</tr>
<tr>
<td>END</td>
<td>ii1,ii2 ← initialise_array(x) = BEGIN</td>
</tr>
<tr>
<td>END</td>
<td>var b1,b2 in</td>
</tr>
<tr>
<td>END</td>
<td>ii1,b1 ← CRE_ARR_COL;</td>
</tr>
<tr>
<td></td>
<td>ii2,b2 ← CRE_ARR_COL;</td>
</tr>
<tr>
<td></td>
<td>STR_ARR_COL(ii1,1,vv);</td>
</tr>
<tr>
<td></td>
<td>COP_ARR_COL(ii2,ii1)</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>

DESCRIPTION

L_ARRAY_COLLECTION is used to handle identically sized one dimensional arrays.

It contains basic operations (create, delete, read, write, compare).

MACHINE PARAMETERS

L_ARRAY_COLLECTION (LACOLL_maxobj, LACOLL_INDEX, LACOLL.VALUE): LACOLL_maxobj is the maximum number of arrays in the collection. LACOLL_INDEX is the set of array indexes, LACOLL.VALUE is the set of possible values of array elements.
CRE_ARR_COL
Syntax ii, bb ← CRE_ARR_COL
Outputs bb is a Boolean element indicating whether any available arrays are left in the collection, ii is the index of this available array.

Assigning an array in the collection.

DER_ARR_COL
Syntax DEL_ARR_COL (ii)
Preconditions ii must belong to 1..LACOLL_maxobj

The array of index ii in the collection is released. It may once again be assigned using CRE_ARR_COL.

VAL_ARR_COL
Syntax vv ← VAL_ARR_COL (ii, jj)
Preconditions ii must belong to 1..LACOLL_maxobj and jj belong to LACOLL_INDEX.

Output vv contains the jj number value of array ii.

Use vv to store the value of number jj in array ii.

STR_ARR_COL
Syntax STR_ARR_COL (ii, jj, vv)
Preconditions ii must belong to 1..LACOLL_maxobj, jj belong to LACOLL_INDEX and vv belong to LACOLL_VALUE.

Write the value of vv to cell number jj in array ii.

COP_ARR_COL
Syntax COP_ARR_COL (dest, src)
Preconditions dest and src must belong to 1..LACOLL_maxobj.

Copy the contents of the src array to the dest array.

CMP_ARR_COL
Syntax bb ← CMP_ARR_COL (range 1, range 2)
Preconditions range 1 and range 2 must belong to 1..LACOLL_maxobj.

Output bb is a Boolean element indicating whether array range 1 and range 2 are identical.

Comparison between the two 2 arrays.

IMPORTS REQUIRED
(instances to import as the implementation tree for this library machine sees them with SEES).

BASIC_ARITHMETIC BASIC_BOOL
5.13 L_ARRAY1_COLLECTION: array of the same size, with numerical indexes

OPERATIONS

- **CRE_ARR_COL** returns a Boolean element indicating whether an array remains available in the collection and the index of this available array (promoted operation).
- **DEL_ARR_COL** releases the array mentioned (promoted operation).
- **VAL_ARR_COL** read an element from one of the valid arrays (promoted operation).
- **STR_ARR_COL** write an element from one of the valid arrays (promoted operation).
- **COP_ARR_COL** copy one of the arrays to another (promoted operation).
- **CMP_ARR_COL** compare two arrays (promoted operation).
- **SET_ARR_COL** copy the same value to an index range in one of the arrays.
- **PCOP_ARR_COL** copy part of one of the arrays to another in a given position.
- **PCMP_ARR_COL** search for the first different element between two parts of two different arrays. A Boolean element indicates whether the element was found and in this case, the index of this element is returned.

EXAMPLE

Using SET_ARR_COL to fill-in two arrays and PCOP_ARR_COL to define a third one. Note the need to test the Boolean output elements from CRE_ARR_COL in order to use the arrays created.

The example is as follows:

```plaintext
MACHINE
  M1
OPERATIONS
  op = skip
END
```
IMPLEMENTATION
M1_1
REFINES
M1
IMPORTS
L_ARRAY1_COLLECTION(3,3,1,10)

OPERATIONS
op = var i1,i2,i3,b1,b2,b3 in
   i1,b1 ← CRE.ARR.COL;
   i2,b2 ← CRE.ARR.COL;
   i3,b3 ← CRE.ARR.COL;
   if b1 = TRUE ∧
      b2 = TRUE ∧
      b3 = TRUE
      then
         SET.ARR.COL(i1,0,3,1);
         SET.ARR.COL(i2,0,3,2);
         PCOP.ARR.COL(i3,0,i1,0,1);
         PCOP.ARR.COL(i3,2,i2,2,3)
      end
   end
END
END

DESCRIPTION

L_ARRAY1_COLLECTION enables the use of a collection of arrays without the need to
code loops to position a set of elements or arrays. This was not possible with the previous
machine L_ARRAY_COLLECTION where index sets are normally unordered.

MACHINE PARAMETERS

L_ARRAY1_COLLECTION (LAUC_maxobj, LAUC_maxidx, LAUC_minval,
LAUC_maxval): The variable is a partial function of 1..LAUC_maxobj in the set of to-
tal functions of 0..LAUC_maxidx to LAUC_minval..LAUC_maxval. LAUC_maxobj is a
NAT1 that is different from MAXINT. LAUC_maxidx, LAUC_minval and LAUC_maxval
are NATs and LAUC_minval ≤ LAUC_maxval.

CRE.ARR.COL

Syntax     ii, bb ← CRE.ARR.COL
Outputs    bb is a Boolean element indicating whether any available arrays remain
            in the collection, ii is the index of this available array.

Allocate an array in the collection.

DEL.ARR.COL

Syntax     DEL.ARR.COL (ii)
Preconditions ii must belong to 1..LAUC_maxobj

Array ii in the collection is released. It may once again be assigned using CRE.ARR.COL.
VAL_ARR_COL
Syntax \( vv \leftarrow \text{VAL	extunderscore ARR	extunderscore COL} (ii, jj) \)
Preconditions \( ii \) must belong to \( 1..\text{LAUC	extunderscore maxobj} \) \( jj \) must belong to \( 1..\text{LAUC	extunderscore maxidx} \).
Output \( vv \) contains the value of number \( jj \) in array \( ii \). Store in \( vv \) the value of number \( jj \) in array \( ii \).

STR_ARR_COL
Syntax \( \text{STR	extunderscore ARR	extunderscore COL} (ii, jj, vv) \)
Preconditions \( ii \) must belong to \( 1..\text{LAUC	extunderscore maxobj} \) \( jj \) must belong to \( 1..\text{LAUC	extunderscore maxidx} \) \( vv \) must belong to \( \text{LAUC\textunderscore VALUE} \).
Write value \( vv \) to the \( jj \)th cell in array \( ii \).

COP_ARR_COL
Syntax \( \text{COP	extunderscore ARR	extunderscore COL} (\text{dest}, \text{src}) \)
Preconditions \( \text{dest} \) and \( \text{src} \) must belong to \( 1..\text{LAUC	extunderscore maxobj} \).
Copy the contents of the \( \text{src} \) array to the \( \text{dest} \) array.

CMP_ARR_COL
Syntax \( bb \leftarrow \text{CMP	extunderscore ARR	extunderscore COL} (\text{range \( 1 \)}, \text{range \( 2 \)}) \)
Preconditions \( \text{range \( 1 \)} \) and \( \text{range \( 2 \)} \) must belong to \( 1..\text{LAUC	extunderscore maxobj} \).
Output \( bb \) is a Boolean element that indicates whether array ranges \( 1 \) and \( 2 \) are identical.
Comparison between the two arrays.

SET_ARR_COL
Syntax \( \text{SET	extunderscore ARR	extunderscore COL} (\text{range}, ii, jj, vv) \)
Preconditions \( \text{range} \) belonging to \( \text{dom(arr	extunderscore col)} \), i.e. it corresponds to the index of a previously created array. \( ii \) and \( jj \) are in \( 1..\text{LAUC	extunderscore maxidx} \) \( jj \) must be different from \( \text{MAXINT} \) \( vv \) is in \( \text{LAUC\textunderscore minval..LAUC\textunderscore maxval} \).
The value \( vv \) is copied to the range array for all indexes between \( ii \) and \( jj \). If \( ii > jj \), the array remains unchanged.

PCOP_ARR_COL
Syntax \( \text{PCOP	extunderscore ARR	extunderscore COL} (\text{dest}, \text{idx	extunderscore dst}, \text{src}, ii, jj) \)
Preconditions \( \text{dest} \) and \( \text{src} \) are elements that are different from \( 1..\text{LAUC	extunderscore maxobj} \) corresponding to arrays already created. \( ii..jj \) is a non blank interval of \( 0..\text{LAUC	extunderscore maxidx} \) and \( jj /\!\!\!\!\!\!\!\!\!= \text{MAXINT} \) \( \text{idx	extunderscore dst}..\text{idx	extunderscore dst} + jj - ii \) is an interval of \( 0..\text{LAUC	extunderscore maxidx} \).
The \( ii..jj \) part in the \( \text{src} \) array is copied to the \( \text{idx	extunderscore dst}..\text{idx	extunderscore dst} + jj - ii \) part of the \( \text{dst} \) array.
PCMP_ARR_COL

Syntax idx, bb ← PCMP_ARR_COL (nn2, idx2, nn1, ii, jj)

Preconditions nn1 and nn2 are elements that are different from 1..LAUC_maxobj and correspond to arrays already created. ii..jj is a non blank interval of 0..LAUC_maxidx. idx2..idx2 + jj - ii is an interval of 0..LAUC_maxidx.

Outputs bb is a BOOL. idx is in ii..jj.

The ii..jj part in array nn1 is compared to part idx2..idx2 + jj - ii in array nn2. bb is FALSE if the two parts are identical, TRUE if not. In this case, idx is the index of the first element that is different from ii..jj.

IMPORTS REQUIRED

(instances to import as the implementation tree for this library machine sees them with SEES) BASIC_ARITHMETIC, BASIC_BOOL.