Sorting and other functions

15.03.2020, 30.06.2019 Arve Meisingset

There is a need to sort lists. We have the following operations which involves sorting:

- User defined sorting
- Identifiers
- Last first
- Last last
- Reversing
- Adjustments
- Specified sorting
- Significant duplicates
- Cardinality
- Cardinality functions
- Alphanumerical ascending
- Alphanumerical descending

In addition to sorting within a list, there is a need to sort across lists. This may happen when allowing for use of significant duplicates.
Example: At the schema level we have the following structure

House (Car, Dog).

At the population level we may want

House (Car, Car, Dog) and House (Car, Dog, Dog).

Here we have two House-s, three Car-s and three Dog-s. The Car-s and Dog-s appear in particular House-s. All Car-s are listed before any Dog in each House. However, they appear in the same list under each House.

We may want the sorting in the population to be dependent on the sequences in the schema. This is the normal situation in the External/Contents population: For each House, we list all Car-s before the Dog-s. In the External terminology layer, they may not be sorted. Here a sort function needs to be added if sorting is wanted. See a subsequent section.

User defined sorting

Lists are presented at the user-computer interfaces. The end user may expand the lists with blank items, and fill into these. Alternatively, he places the cursor somewhere inside a list of characters, and inserts additional tokens. The inserted items are stored where they are inserted. Hence, the end user decides the sequence in the list.

Such lists may also be found in the meta data, defining the sequence of letters in the alphabet, sequence of digits in a number system etc.

User defined sorting allows the user to insert items anywhere in a list. The same sequence is applied in presentations to the user, in the application logic, and in storage of the list. User defined sorting is the default sorting of any list.

Identifiers

Before continuing, let us first define the identifier function Id, eg. in a meta schema.

```
Id   (S <> '&(& Identifier
     <> '&
```

Id is a function. The affected attribute value is found under the superior attribute of Id. '& refers to a superior data item, within which the attribute value is unique. Example schema:

```
:                          House
    :                      Car
        :                  Registration number
            :
                :          Id
                    S      <> '&(& Identifier
                    <>     Id ': 'Registration number 'Car 'House
```

The colon, in the line having a blank name tag, indicates the value of the Registration number. Id is a contained function. This function is defined in a schema. The name space of the identifier is defined in the condition on the identifier, in the last line.

Any Car instance will have a Registration number value that is unique within the scope of its superior House instance.

The Id function is in this example used to define a Local Distinguished Name of the Id attribute value.

The Id function may as well be applied to the combined value of attribute groups. Example value of attribute group:

```
:                          House
    :                      Car
        :                  Registration number
            :              (Id <> Id ': 'Registration number 'Car 'House
            ;              Letters
            :              Number
```

If entities are tagged, the default identity is within the the superior entity, ie. The House. Hence, the reference needs not be stated.

```
:                House
    :            Car
        :        Registration number
            :    (Id <>
```

Function notation

If we use local function names, for every function we have to refer to its definition, eg S <> '&(& Identifier.

If the function name is global, it may be referenced by the condition symbol only, eg.

```
Id <>
```

The definition of the function is found in a particular register. The parameters of the function may be filled in after the condition symbol.

Last first

We assume that Instruction-s (><) are inserting items at the start of each list. This way, we need not step to any subsequent position before doing insert or delete. This means that the last inserted item comes first in the list of contained items.

The function Last first may be called by

```
Lf    (S <> '&(& Last first
```

Suppose you want to place the last inserted letter first in a Car (Name. This is stated as follows:

```
:                House
    :            Car
        :        Name
            :
                :    (Lf (S <> '&(& Last first
```

The line with the blank name tag is the value. The last line is a letter. It contains the Last first function, which may be written Lf <> only.

The Last first function function may not be needed, as most lists use User defined sorting, while functions in the application logic use Last first.

If data being inserted Last first into a sink list are copied from a source list First first and the first is then deleted in this source list, the sink list will contain its item in reverse order. We will in our working papers use First first as the normal working of the Instruction operator, ><.

Last last

Inserting Last last means that the last inserted item comes last in the list of contained items. This happens typically when a cursor is floating with the last item.

This sorting may be implemented by having a two-way list in a ring of contained items, having a pointer from the superior item to the first contained item, and another pointer to the last contained item. This ring may appear in the internal layers of the data transformation architecture, and may not appear in the application layer, ie. not in the External Terminology Layer.

The function Last last is called by

Ll (S <> '&(& Last last

'&(& First last refers to the Last last definition. The function may be called from the Letter level. The short hand notation is Ll <>.

When inserting data from the user interface, Last last is the normal mode. This mode is often applied on any sub-string after placement of the cursor.

Reversing

If you copy one item at a time by First first from a list and delete it from the original list while inserting the copy into a new list by Last first, then the new list will become the reverse of the original list.

If you edit the new list by Last first, this editing will become similar to editing the tail of the original list.

Thereafter, you may reverse the new edited list into the original list, which will now be edited at the tail.

See the previous section on a more efficient implementation.

Adjustments

Attributes may have fixed length fields, into which values may be put. The length of the attribute may be specified by the Lh function at the value level:

Lh (S <> '&(& Length
<> , x

where x gives the max number of characters in the value.

The short hand notation is Lh <> , x.

The value of an attribute may be adjusted to the left, often being used for texts. This is specified by attaching the Le function to the value. Example:

```
:           House
    :       Explanation
        :   (Le  (S <> '&(& Left), <>
```

The short hand notation is Le <>.

Numbers are typically adjusted to the right. This is specified by the Ri function at the value level. Example:

```
:           House
    :       Number
        :   (Ri  (S <> '&(& Right), <>
```

The short hand notation is Ri <>.

Specified sorting

Example

```
:           House
    :       Car
    :       Dog
```

When instantiated, some Car-s and Dog-s may appear under the same House. The instances are to be sorted under the each House in the sequence they are stated in the schema. Hence, for each House, all Car-s are listed first, then all Dog-s.
All Car-s and Dog-s may appear in the same list under a House. However, it may be inconvenient to search through all Car-s before you insert a Dog. Due to the sorting, we may consider the list of Car-s to be separate from the list of Dog-s.

Significant duplicates

This is the default case. Hence, no specification is needed.
In the above Example, House, Car and Dog may all have significant duplicates. Hence, when instantiated, there may be multiple House-s, Car-s, Dog-s etc.

At the entity level under a House there will by significant duplicates of Car-s and Dog-s. However, we may or may not add the Id function to the value of the Name of Car-s (and Dog-s), making them unique or not.

Cardinality

Car may have an attribute Registration number, which uniquely identifies the entity within the superior entity House. Dog may have an attribute Name, which uniquely

identifies the entity within the superior entity House. This means that each entity will have a Local distinguished name.
Example

```
:           House
    :       Name
    :       Car
        :   Registration number
    :       Dog
        :   Name
```

There are three requirements on an attribute acting as a distinguished local name:

a) The attribute will contain one value only, ie. single-valued-ness
b) There shall be just one instance of this attribute within the entity, eg. Car
c) There shall be only one instance within this class with this identifier value within the scope of its superior entity, eg. House

We write these requirements for the Car (Registration number attribute:

a) The attribute will contain one value only, ie. single-valued-ness.

   This is a cardinality constraint on the value, which can be stated as:
   Car (Registration number (: <> :, ;!

   The Condition on this value, ie on the (:, states that the value has no (!) previous value (;). This means that Registration number contains zero or one value only. Zero means no value.

   Also, to state that any value has no next value may do:
   Car (Registration number (: <> :, :!

   However, this is not a constraint on the value being inserted, but on its previous. Therefore, this constraint needs to be transformed prior to execution.

b) There shall be just one instance of this attribute within the entity.

   This is a cardinality constraint on the attribute Registration number, which can be stated as:
   Car (Registration number <> Registration number, ;Registration number !

   The Condition on this Registration number states that the it shall have no previous Registration number attribute. This means that the entity contains one or zero Registration number attribute only, ie there shall be no repeating Registration number attribute. If not explicitly constrained, there may be multiple attribute instances of the same class within the superior data item.

   Note that the constraint will not prohibit that the attribute will have a previous Name, Colour etc., like what the following constraint would claim:
   Car (Registration number <> Registration number, ; !

c) There shall be only one instance within this class with this identifier value within the the scope of its superior entity.

This is a uniqueness constraint on the value of an attribute of an entity within a superior entity. The constraint is executed before insertion, and can be stated as follows:
House (<> Car (Registration number (XXX !))), (>< Car (Registration number (XXX)))

Note that the outer parentheses are used to indicate that both the Condition and Instruction are within the scope of the particular House. XXX is an example value to be inserted by the user, and is not a variable.

The negation (!) in the Condition tells that the Condition will only be Satisfied when this value does not appear within the given scope. Hence, the value (XXX) will only be inserted if it does not already appear.

Constraint c only contains the Insertion Instruction. Therefore, it has to be executed first. Constraint b is on the attribute, while constraint a is on the value. Hence, we state the constraints in this reversed order of the above presentation, ie in the sequence entity, attribute, value.

Using the one-dimensional notation, the total constraint will look as this:
House (<> Car (Registration number (XXX !))), (>< Car (Registration number (XXX))), Car (Registration number (<> Registration number, ;Registration number !, : <> :, ;!))

The expression is simpler to read in a two+one-dimensional notation:

```
:                   House
    <>              Car (Registration number (XXX !,
    ><              Car (Registration number (XXX
    :               Car
        :           Registration number
            <>      Registration number, ;Registration number !
            :
                <>  :, ;!
```

Cardinality function

As an alternative to the previous section, cardinality may be expressed by a cardinality function
We may tell that an item is an entity by adding a function E<> to each entity class. A shorthand notation is to underline the class label.

: House E <>

: Car E <>
: Registration number C (n, m) <>
: C (n, m) <>

Here we have used the short hand notation for functions.

The Registration number has minimum cardinality n and maximum cardinality m, eg n=1, and m, eg m=1. Any Car will have only one Registration number, which has only one value.

Specified sorting
A list in a schema will act as a sorting instruction for the corresponding population.

Example:
Schema
: House
: Car
: Dog
Population
: House
: Car
. Car
: Dog
: House
: Car
: Dog
: Dog

The Schema contains Car and Dog in the same list under House. Hence, Car-s and Dog-s - if any - will appear in the same list under each House in the population.

If the value set of a multi-valued attribute is Red, Blue, Yellow, then they will be listed in this sequence under each attribute instance, eg Red, Red, Yellow, Yellow, Yellow.

Example:
: House
: Wall
: Colour
: (So (<> ‘ ‘House), S <> ‘&(& Sort

If the Walls under a House entity shall be sorted on Colour values, the So function is stated on the Colour values under the parameter House.

If the attribute is multi-valued, all the purely Red Wall-s are listed first; then the Red and Blue, then Red and Blue and Yellow, then only Blue, then Blue and Yellow, then Yellow.

If the attribute is repeating, and single-valued, the sorting is as in the previous paragraph.

If the attribute is multi-valued, and the sorting is only on values within the attribute, the constraint will be

(So (<> 'Colour), S <> '&(& Sort

This sorting may be default.

If no So is stated, then the the Wall-s are listed by User defined sorting.

Alphanumerical ascending

A Letter data type may be defined in the following meta-schema:

```
:          System
  :        Types
    :      Letter
      :    A, B, C, D, ... X, Y, Z
```

Here all permissible Letter-s are listed, and they are listed in a sequence defining the ascending order.

An application schema and example population may look like this:

```
:                    System
  :                  House schema
    :                House
      :              Colour
        :
          S <>       S 'Colour 'House 'House schema 'System (Types (Letter
          So <>      So ': 'Colour 'House

  :                  House register population
    S <>             S 'House register population 'System (House schema
    :                House
      :              Colour
        :            BLUE
        :            BROWN
    :                House
      :              Colour
        :            BLACK
```

: BROWN
: RED

In the above example, a House may contain Colour attributes. The Colour attributes may contain any alphabetical name values, ie. a series of Letters. The Colour name values are sorted, within each House.

Alphanumerical descending

Alphanumerical descending is achieved by Reversing Alphanumerical ascending.