Existence Logic

18.03.2020 Arve Meisingset

Purpose

This paper gives an overview of Existence Logic.

The motivation is as follows:
1. The language defines data as they appear at the end user interface without introduction of artificial terms
2. Headings and values are local to each other, such that each item defines a name space - context - for other items
3. The data allows for use of significant duplicates both of headings and values
4. The same language is used to define data instances as well as their classes, such that they become homomorphic, ie. many-to-one mappings from instances to classes, and classes are copied into instances
5. Data are classes and instances relative to each other, and their schema references may be nested in any way
6. The data may or may not describe phenomena being observed from other phenomena, such that data, phenomena and concepts become isomorphic, ie. one-to-one without being onto, as some data may give overview of other data without describing anything
7. The language is used to define the contents layer and the external terminology layer of the data transformation architecture
8. The language may be used to define both alphanumeric and graphical user interfaces

The language violates the type-token principle. This means that the language handles inscriptions only, and has no notion of strings behind similar inscriptions. Hence, John and John are two different inscriptions. The language (theory) has no notion of a common John denoted by the two inscriptions.
Also, the language does not distinguish constants and variables. It does not use quantifiers and predicates, and no truth value logic.

The Basic Language

All statements about an application domain are expressed as one tree of data, being organized into a three-dimensional tree.

Each data item may have 3*2 associations, each to one other data item in the tree:

Subordinate	Superior
Next	Previous
Conditioning	Conditioned

A data item may have one Subordinate data item only. However, the recursively Next items of the Subordinate item are considered to be Subordinate, as well. This consideration simplify search expressions and navigation, even if the simplification is not strictly needed.

There is no Previous item of the first item of Next items. However, such Previous items may be used for state handling during execution, being invisible to the user of the language.

There is no Superior of the non-first of Next items. However, such Superior items may be used to express negation. These Superior items do not connect to superior lists of Next items.

The Conditioning item may have associations in any direction, expressing navigation through the data tree containing the Conditioned item. The Condition is finally validated when the control is returned to the Conditioned item.

If the control is returned from a Conditioning to a Conditioned item without having ever accessed the Conditioned item, this is interpreted as a write Inscription. The Conditioning item is interpreted as an Instructed item. The Conditioned item is interpreted as an Instructing item. Hence, Conditions and Instructions are expressed by the same dimension depending on how the control arrives at the items.

The data tree has only one root. It may be very long, having millions of Next items. It may be low, seldom having more than ten subordinate items. And it may be thin, having one Condition with two to three sub-Conditions. It may additionally have five dependent Instructions. From these observations, we may call the data tree a hedge.

Short on Execution

The data tree is executed from its root towards its branches, condition first, subordinate next, then Next and finally superior (including negation). The execution returns instruction first, then next towards its branches. Finally, the execution returns via already executed items to the conditioning item.

The instruction may cause several passes through the branches.

The processing orchestrates execution of four kind of branches
- The condition branch
- The control branch (by the condition)
- The instruction branch
- The write branch (by the instruction)

The instructing branch consists of three sub-parts
- The navigation part up to the second instruction

- The fixed insert part up to the third instruction that copies from the instruction branch to the write branch
- The variable insert part up to the fourth instruction that copies from the control branch to the write branch
- The delete part containing the fifth instruction that instructs deletion

We have compared the data tree with a hedge. The execution along the branches resembles DNA strands, where one branch matches with another branch.

Manipulation Commands

The Manipulation Commands are signal that may be sent from outside the data tree or may be exchanged among the branches of the data tree. These Commands have recognizable data structures, but may not be primitives.

The Commands may be attached to any data item, both in schemata and populations. The expressions i9n the data tree may need to refer to the Commands in order to distinguish before and after states of an operation.

The Commands are:

I	-	Insert
D	-	Delete
M	-	Modify
R	-	Read

The Commands may be combined. For example, II on an item in a reference may mean both Insertion in the reference and Insertion of its referenced item.

The Commands are placed subordinate to the item on which they operate. For simplicity, we write them after the item separated by a blank and without parentheses around the Command. In a three-dimensional tree, the Commands may be placed before the first subordinate item, together with Statuses that are used to control the execution. This way, they may be invisible to users.

One-dimensional Notation

The one-dimensional notation is convenient for defining condition and instruction branches. These make up the third dimension of the data tree.
The condition and instruction branches state navigation up and down in the data tree.

The one-dimensional notation uses the following symbols:

:	any item
<>	condition

><	instruction
(	subordinate
‘	superior
)	end of subordinate
,	next
;	previous
&	recursion
‘&	recursion via superiors
(&	recursion via subordinates
R	reversal of a specified list from the first a item to the first b item - R<>a, b

The recursion symbol - & - is replaced by specific navigation through pre-compilation and instantiations. The expression ‘&(& is a wild card that used to make a reference to anywhere in the data tree.

The specified list on which R applies may be navigating up - ‘ - and down - (-, and may not only be next or previous. The target list may be placed previous or above the R, and the reversed list is replacing the R expression.

Example condition branch:

<> Owner ‘Car ‘Producer ‘Country (Person

This expression states a reference from the Owner of a Car up via Producer to its Country and down to a Person in the same Country. The navigation - here up to Country - is used to state the scope of the reference.

Some condition branches may go downwards only, or a branch may contain several sub-branches in various directions.

Example instruction sub-branch:

<> Owner ‘Car ‘Producer ‘Country (Person >< (#owners >< (:

This expression inserts a value (: of #owners under Person - to be supplied by the user after the last instruction, as there may be several Owner-s of the same Car.

Two-dimensional Notation

The two-dimensional notation is most convenient to state the data tree.

```
:                    is used to state an item of the data tree
    :                indentation is used to state a subordinate item
    :                line shift is used to state next item
        <>           is used to state a condition on the superior item
        ><           is used to state an instruction
```

```
:   :               an indented colon in the same line identifies a name tag
:          Car      most often name tags are put into a separate column
                    without a colon to identify it
    S<>             is used to refer to a schema containing the classes
                    of instances contained in the superior item of the S
    P<>             is used to refer from a schema to its population
                    of instances
```

References from a Condition to its referenced item is most conveniently stated in the one-dimensional notation, where the reference is stated in the name tag column. References of instructions are stated in the same way.

Example

```
:                       System
    :                   Schema
        :               Producer
            :           Name
            :           Car
                :       Registration number
                :       Owner
                    <>  Owner 'Car 'Producer ': (Person
        :               Person
            :           Name
    :                   Population
        S<>             S 'Population 'System (Schema
        :               Producer
            :           Name (A
            :           Car
                :       Registration number (1
                :       Owner
                    <>  Owner 'Car 'Producer ': (Person (Name (a
        :               Producer
            :           Name (B
        :               Person
            :           Name (a
```

Note that

- Each item in the Population is identical to its class in the schema.
- There is an explicit reference from the Population to its Schema.
- Both Producer-s and Person-s have Name attributes, which may or may not have similar value sets.
- We have written values at the attribute lines, eg. (1, but could have put them on separate lines with indentation, as well.
- The references in the Schema are made via the Schema item (:) without referencing its name tag. This allows the references to be copied into the Population.

- References in the Schema are most often stated to the entity class, eg. Person; we have added identifier attributes and values in the Population, eg (Name (a.
- Value types could have been defined in the Schema, as well. The permissible values are either listed under each attribute, or the attribute contains a schema reference to the value set definition in a meta-schema.

Condition branches may contain sub-Conditions.
Write instructions may be stated on a conditioned item or on any item in a condition branch. Instruction branches may contain sub-Condition specifying the targets for insertions. The Instruction symbol is used to separate the various parts of a instruction branch.

Meta-functions

Meta-functions are fundamental functions that are used to define other functions. Meta-functions are not strictly needed, but are useful in most applications.

The distinction between classes and instances is not strictly needed, but the distinction is very useful. Schema and Population references are used to define this distinction. The basic language may not need Negation, but it is very useful.

The meta-functions are as follows:

S<>	Reference from a population to (one of) its schema(ta) containing classes that act as templates for the instances in the population. Schema references may be recursive, such that a class may contain a schema reference to another class, or to an instance, eg to a table, in the population.
P<>	Reference from a schema to (one of) its population(s) containing instances that are copies of the classes contained in the schema(ta). The Population reference is the reverse of the Schema reference; it is not needed for execution, but may be helpful for documentation.
!	Negation is a suffix on a data item (in a condition branch), telling that the item shall not appear among the controlled data. If x! appears in a condition branch and x! appears in the corresponding control branch, the condition is satisfied. Also, negations may be copied from an instruction or control branch to a write branch.

Function applications

A function is a data item that contains a many-to-one mapping between arguments and function values.

Every application of a function has an explicit or implicit reference to its definition in a schema.
The location of the arguments of the function are referenced by its condition branch.
The location of its function values are referenced by its instruction branch.

Note that the recursion (&) and reversal (R) operators are replaced by concrete lists during the execution.

The name tag of a function is local to its superior data item, and may be different at different places of the same functionality. The function name tag is only fixed in cases where the schema reference is left out. Also, references to arguments and/or function values may be left out when their placements are default.

The notation for a function is as follows:

F	function name tag
S	reference to the schema of the function
<>	reference to arguments
><><	reference to function value

A function may have an empty condition. The short hand notation for a function is

F<>

Pre-defined functions

E<>	Entity function defines the data item to be an entity
Id<>	Identifier function, defining the value to be an identifier within the scope of the superior of the current entity, or within the scope of some other referenced superior entity.
Lf<>	Last first is a condition on a component of a value, eg a letter, to be inserted first
Ll<>	Last last is a condition on a component of a value, eg a letter, to be inserted last
Lh (x)<>	Length defines the maximum length x of a value
C (n, m)<>	Cardinality constraint, minimum n and maximum m, on values, attributes or entities
So<>	Sorting is a constraint on the values to be sorted according to the sequence in their definition and within the scope of the referenced item
+<>	Increment function under an inserted entity or value, that adds 1 to the value under the write Instruction
÷<>	Decrement function under a deleted entity or value, that subtracts 1 from the value under the write Instruction