Is Predicate Calculus a Descriptive Language?

15.10.2017 Arve Meisingset

Most parts of this paper are addressed in the paper on Evaluation of Predicate Calculus. The current paper provides better explanations, and may serve as a better introduction to Existence logic. Existence logic is a language for descriptions. Predicate calculus is a language for calculi on truth-values.

According to the Church-Turing Thesis, a Universal Turing automaton can evaluate anything stated in first order Predicate Calculus. However, this paper is not about computability, but about descriptive capability. We show the differences between Predicate Calculus and Existence logic.

We will discard the notions of predicates and truth values, and replace all notions of truth-value calculus. We will show that inscriptions like 'person', 'person' and 'inhabitant' may denote different phenomenon instances, while name tags like 'JOHN' and 'MARY' denote nothing. We will arrive at an isomorphic mapping between phenomena and their description.

Contents

1 Global terms

A simple predicate statement may look as follows:

Loves (JOHN, MARY)

The functor Loves maps from the terms JOHN and MARY – in that sequence – to a Truth-value, eg.

TRUE=Loves (JOHN, MARY)

JOHN is in the above statement assumed to be a globally unique term. This means that some observer is assumed to observe the system of terms from the outside.

If JOHN denotes an entity, the outside observation of the term must imply an outside observation of the entity.

Predicate Calculus provides no means to state who is the observer, and where he is observing from, except you may introduce more terms, like

Thinks-that-loves (ANN, JOHN, MARY)

Here, ANN may be the person that thinks that JOHN Loves MARY. However, here ANN is being included in the system that is being observed from the outside. This is similar to Predicate Calculus and Set Theory do not treat time changes, but you may include time events as entities in their static universes.

Predicate Calculus does not treat context, but you may introduce context with a dot notation for the terms, like this:

Loves (NORWAY.JOHN, NORWAY.MARY)

NORWAY is now the identifier of the place from where JOHN is observed.

The above addition of terms shows attempts of extending Predicate Calculus with means to treat time and contexts without introducing these into the foundation of the calculus.

2 Classification of terms

Both dogs and persons may have the identifier JOHN, and both companies and countries may be named NORWAY. Therefore, we have to qualify the identifier values by their entities:

Loves ((Country (NORWAY, Person (JOHN))), (Country (NORWAY, Person (MARY))))

The dot notation is not satisfactory, because Person is not contained in NORWAY, but in a Country having the identifier NORWAY. Therefore, NORWAY is a sub-branch at the same level as Person. Hence, we introduce parentheses. Also, we introduce commas to indicate next item at the same level. The parentheses and commas modify the foundation of Predicate Calculus.

We may have more identifiers of the entities, eg. country Name and ISO CC, and Person Name and Person Civil Registration Code. Hence, we need to introduce the attribute classes:

Loves ((Country (Name (NORWAY), Person (Name (JOHN)))), (Country (Name (NORWAY),
Person (Name (MARY)))))

We have now put Person at the same level as Country Name. Note that commas are used both to separate the items Name and Person in a list, and are used to separate the arguments of the predicate.

We have introduced quite a lot of parentheses, but in a subsequent section we will do away with most of them.

If our predicate is constrained to both Person-s being contained in the same Country, the last expression may be simplified:

Loves (Country (Name (NORWAY), Person (Name (JOHN))), Person (Name (MARY))))

Here, the Country appears as the superior data item of both Person-s.

3 Improper names

At the end of the previous section, we see that the two arguments of the predicate are identified by the following expressions:

Country (Name (NORWAY), Person (Name (JOHN)))
Country (Name (NORWAY), Person (Name (MARY)))

These expressions are branches that have sub-branches. They are not strict paths within trees, as of the dot notation.

Predicate Calculus uses proper names, like JOHN and MARY. We replace proper names by complex expressions.

In Predicate Calculus, the terms – JOHN and MARY – are used to identify entities. In the new expressions, the entire branches are needed to identify the entities. In other words, the two Person-s are identified locally to the Country in which they belong.

4 Denotations

The name tags Name (NORWAY), Name (JOHN) and Name (MARY) are just attached as sub-branches from the entity labels; they denote nothing. It is the entity labels Country, Person and Person that denote the entities. This is fundamentally different from the assumptions behind Predicate Calculus.

The denoted entities are organized in a tree corresponding to the data tree Country (Person, Person). We will come back to this in a subsequent section.

We observe that there is an isomorphic mapping between the data labels and the denoted phenomena.

We additionally observe that we will use significant duplicates, like Person and Person, in lists of lists. This is also fundamentally different from the assumptions behind Predicate Calculus. However, in some cases we will add name tags like Name (NORWAY), Name (JOHN) and Name (MARY).

5 Roles

Now that we know that the Person-s are Inhabitant-s in one and the same Country, we may change the entity labels to the following:

Country (Name (NORWAY), Inhabitant (Name (JOHN)))
Country (Name (NORWAY), Inhabitant (Name (MARY)))

In the original predicate, Loves, the two terms have different positions. The two terms may have identical values if JOHN loves himself. However, if we use variables in the two positions, then these have to be different.

The combination of the predicate label and the position of a term tells the role of the term.

We may replace the predicate label and the term by a new label, Loved-person. Then we get the following expression:

Country (Name (NORWAY), Inhabitant (Name (JOHN), (Loved-person (Name (MARY))))

Here, Inhabitant is contained in Country, and Loved-person is contained in Inhabitant, who has this Loved-person. We have replaced the comma, which separated the two original arguments, with a left hand parenthesis in front of the Loved-person.

The original predicate formulation maps to truth-values. This allows for constructing a calculus on truth-values. However, this mapping and calculus have nothing to do with description. Truth-value calculus is made in a meta-language of the descriptions. Predicates map to these truth-values, which we have removed.

The new formulation structures data such that they can describe phenomena. We will come back to this in a subsequent section.

6 Domains

The last expression shows Loved-person-s being contained in Inhabitant-s. However, the Loved-person is an Inhabitant, as well. Hence, we have to list two inhabitants and the association between these. However, now the name labels are no more global. Hence, the association will be stated by a condition on the Loved-person plus a navigation to the right Inhabitant. The navigation consists of ascending – ‘ - to a common superior followed by descending – (- to the Inhabitant.

Country (Name (NORWAY), Inhabitant (Name (JOHN), (Loved-person (Name (MARY),
<> Loved-person ‘Inhabitant ‘Country (Inhabitant (Name (MARY)))),
Inhabitant (Name (MARY)))

The second line shows the reference from the Loved-person to the Inhabitant, who is listed in the third line. <> is a condition, telling that for the Loved-person to exist, the referenced Inhabitant has to exist, as well. The condition does not produce explicit truth-values. If the condition is satisfied, the execution just continues. If the condition fails, the conditioned item, ie. the Loved-person and everything contained in it, is deleted.

The hyphens – ‘ – indicate ascending one level up in the data tree.

Since, the data form a tree, we do not need to write the name labels when ascending. Hence, the expression may be reduced to the following:

Country (Name (NORWAY), Inhabitant (Name (JOHN), (Loved-person (Name (MARY),
<> ‘ ‘ (Inhabitant (Name (MARY)))), Inhabitant (Name (MARY)))

We realize that the role Loved-person is an entity by itself. It may, or may not refer to another Inhabitant. It may have its owned attributes, eg. data on when it became a Loved-person. Also, the role may have its own identifier attribute, or it may have no identifier.

Country (Name (NORWAY), Inhabitant (Name (JOHN), Loved-person (No (1)),
<> ‘ ‘ (Inhabitant (Name (MARY)))), Inhabitant (Name (MARY)))

Here, the Loved-person has become the first loved one, ie. No (1).

7 Removing parentheses

When arranging the expressions in two dimensions, line shifts will replace the commas. We may replace the opening parentheses by indentations, and remove all closing parentheses.

```
Country
        Name
                NORWAY
        Inhabitant
                Name
                        JOHN
                Loved-person
                        No
                                1
                        <>      ‘ ‘ (Inhabitant (Name (MARY
        Inhabitant
                Name
                        MARY
```

Here, the condition with navigation is written in the one-dimensional notation – without the closing parentheses. The expression may be made even more compact by putting the values at the same line as of the attributes:

```
Country
        Name                    NORWAY
        Inhabitant
            Name                JOHN
            Loved-person
                No              1
                <>              ‘ ‘ (Inhabitant (Name (MARY
      Inhabitant
            Name                MARY
```

We may further clarify the expression by indicating each item by a colon, and put all name labels in a separate column. We have chosen to remove the identifier of the Loved-person.

```
:                                       Country
        :                               Name (NORWAY
                :                       Inhabitant
                        :               Name (JOHN
                        :               Loved-person
                                <>      ‘ ‘ (Inhabitant (Name (MARY
        :                               Inhabitant
                :                       Name (MARY
```

The colon is a special character, like characters for line shifts in text documents. Also, the colon may be understood as a pixel. Any item, including the name labels and condition may be defined by organizing these pixels in three dimensions. The condition indicates the third dimension.

The structure of colons shows the organization of the data. We will come back to the isomorphic organization of phenomena.

8 Removing quantifiers

In Predicate Calculus, we may state that all Inhabitants in NORWAY Loves minimum one Inhabitant – not excluding himself – in NORWAY.

$\forall x \in NORWAY \exists y \in NORWAY\ Loves(x, y) \wedge x \in Inhabitant \wedge y \in Inhabitant$

Here we have combined Set theory and Predicate Calculus, and we have been informal about use of parentheses.

A similar expression in the new language is

```
:                                   Country
        :                           Name (NORWAY
        :                           Inhabitant
                <>                  Inhabitant (Loved-person
                :                   Loved-person
                        <>          ‘ ‘ (Inhabitant
```

Here, we have put in a condition on any Inhabitant that it contains a Loved-person, and may contain more than one Loved-person. This corresponds to the existential quantifier - $\exists$ -. The containment replaces the universal quantifier - $\forall$ -, ie. the expression applies for all Inhabitants in NORWAY.

Note that we replace bound quantifiers only. We have no notion of unbound quantifiers.

9 Classes

In the last expression of the previous section, we have used no variable term, like x and y in the Predicate Calculus formulation. Rather we use the notions of classes and instances as shown in the next expression:

```
:                                   SCHEMA 1
        :                           Country
                :                   Inhabitant
                        :           Loved-person
                                <>  ‘ ‘ (Inhabitant
:                                   POPULATION 1
        S
                <>                  S ‘ ‘ (SCHEMA 1
        :                           Country
                :                   Inhabitant
                        :           Loved-person
                                <>  ‘ ‘ (Inhabitant, Inhabitant
                        :           Loved-person
                                <>  ‘ ‘ (Inhabitant, Inhabitant, Inhabitant
                :                   Inhabitant
                :                   Inhabitant
```

Here, POPULATION 1 contains a role S that refers to SCHEMA 1. S is a schema reference. A reference P in the opposite direction is called a population reference.

The schema contains classes. The population contains instances. The instances are copies of their corresponding class. The classes act as templates for their instances. There is a homomorphic mapping from instances to classes.

In this example, we have shown no attribute and no value. These will be shown in the subsequent section.

The example shows a schema for the management of Country-s, their Inhabitant-s and their Loved-persons. The population shows one Country, three Inhabitant-s, and one Inhabitant has two Loved-person-s. The referenced Inhabitant-s are second and third in the list of Inhabitant-s.

Note that we have no notion of axioms. We have only the notion of classes that work as templates for the instances. We may have constraints, like the conditions, which are copied into the instances, and are executed among the instances when they are created or modified. Functions will be dealt with in a subsequent section.

Since we create instances by copying, we allow for having significant duplicates. Duplicates are the normal thing. We do not assume that similar inscriptions are taken from the same string, and we do not assume that they denote the same entity. Hence, we have abandoned the Type-token principle.

In the above example, there are ten inhabitant inscriptions; they are not taken from the same string, and they do not denote the same entity.

11 Meta-classes

The schema references may be stated recursively, eg. that any item in a schema may have a reference to another schema.

```
:                                       SCHEMA 2
      :                                 Letter (A, B, C, ….

:                                       SCHEMA 1
      :                                 Country
            :                           Name
                  :
                        S <>            ‘ ‘ ‘ ‘ ‘ (SCHEMA 2 (Letter
:                                       POPULATION 1
      S
            <>                          ‘ ‘ (SCHEMA 1
      :                                 Country
            :                           Name (NORWAY
```

Here, SCHEMA 2 contains a list of Letter-s. This is the value type. SCHEMA 1 (Country (Name is an attribute containing a value - (: - that has a schema reference to Letter. We distinguish between the value and the letters in it. The value may - according to its schema reference - contain any sequence of letters. Hence, we can write NORWAY as a value of Country (Name in POPULATION 1.

We observe in the above example that data are not classes and instances in an absolute sense. They are only so relative to each other, and some data may take both roles. One data item may even have several schema references, and they may refer to schemata recursively, as shown.

In the next section, we will learn that schemata have the role of a disjunctions.

12 Replacing Logical Connectives

A list of items in a population states what is the case. Hence, they state a conjunction. However, this is not a logical conjunction, producing truth-values. The conjunction is a collection of data branches from the root of a population, to the leaves of the data tree.

The schema reference states that whatever is found in the schema may be copied into the population. This is a disjunction of data branches, which may be instantiated. This is not a logical disjunction.

If you want to emphasise that the schema reference is a disjunction, S <> may be replaced by v <>.

We may also tag an item in a schema or any other condition by a negation tag - ! -.

```
:                                       SCHEMA 2
        :                               Letter (A, B, C, ….

:                                       SCHEMA 1
        :                               Country
                :                       Name
                        :               <> USA !
                                S <>    ‘ ‘ ‘ ‘ ‘ (SCHEMA 2 (Letter
:                                       POPULATION 1
        S <>                            ‘ ‘ (SCHEMA 1
        :                               Country
                :                       Name (NORWAY
```

In this example, we have stated that USA is not a permissible value according to SCHEMA 1 (Country (Name, even if it is a possible value according to SCHEMA 2 (Letter.

Note that we provide no calculus on truth-values. We may reference other conditions and negations, eg. when stating copy or search. These references have no logical effect on the original expression. There are many other aspects of the data architecture that we will not explain in this paper.

The lists in the population are conjunction. The schema references provide disjunctions. Hence, they organize the data in a kind of conjunctive normal form. However, any list may serve both the role of being a disjunction and a conjunction relative to other data.

Note that for a logical disjunction to be True, minimum one of the constituents must be True. This does not apply for the schema reference; none of the schema options needs to be instantiated.

13 Recursion

We may allow for recursion by allowing a superior data item become the schema.

```
:                                       SCHEMA 3
        :                               Group
                :                       Group
                        S <>            ‘ ‘Group
:                                       POPULATION 3
        :                               Group
                :                       Group
                        :               Group
                :                       Group
        :                               Group
```

Note that the schema reference contains no parenthesis. This means that it refers to a strictly superior item, ie. to the upper Group inscription.

Note that the recursion does not generate the contents of POPULATION 3. The recursion only allows for inserting recursive contents into the population, eg. Group-s containing Group-s.

It is inconvenient to use schema references to state recursive use of operators. Therefore, we allow for recursion by stating & after the operator, eg. ‘& for superior recursively, (& for subordinate recursively, ,& for next recursively, and ;& for previous recursively.

We use the expression ‘& (& Xxxx to refer an arbitrary number of levels up from anywhere, and then an arbitrary number of levels down to the name label Xxxx. This replaces use of global names, which is a notion of non-locality- being claimed with entanglement in quantum mechanics.

The notation in the new formalism allows for local references and local effects only. The references to other locations has to be made explicit. Most often, the expression ‘& (& Xxxx in a schema will be replaced by an explicit reference in the population.

14 Functions

This section has become more complicated than intended, but is necessary for proving the claims in the Conclusion section. If you do not want to go into the details, you may skip this section.

A function is a mapping from a set of arguments, x1, x2, …, to a function value, y, eg.

y=f(x1, x2, …)

We will explain this idea using the new notation.

In a meta-schema, SCHEMA 5, we define a Digit, and the signature of a function IncOp. The function takes Arg as arguments and gives Func as a function value. Both the arguments and the function value takes Digit as their domain.

For assignment and placement of the function value, we use a write operator - >< - together with a navigation path. The write operator is the opposite of the condition operator - <> -.

```
:                                   SCHEMA 5
    :                               Digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    :                               IncOp
        :       <>                  Arg (S <> S ‘Arg ‘<> ‘IncOp ‘SCHEMA 5 (Digit
                ><                  Func (S <> S ‘Func ‘>< ‘IncOp ‘SCHEMA 5 (Digit
```

We have an extra Condition within the condition branch, and use ‘<> to backtrack over this Condition. We have an extra Instruction within the instruction branch, and use ‘>< to backtrack over this Instruction.

In the above schema, the arguments Arg are found in a condition branch, and the function Func value is found in an instruction branch. This is not what we want. We want the arguments to be found somewhere in the main data tree, and the function value to be delivered somewhere else, as well. Furthermore, for the Condition to be satisfied, the Arg has to be found under the conditioned item in the main data tree. Likewise for the Func. Hence, we would have to write Conditions and Instructions to take and deliver the values to these data items.

The reason for the above difficulties, is that Arg and Func are constants having local name tags, and we do not support variables with global name tags that may reference by names across the data tree. Hence, we replace Arg and Func by arbitrary navigations, ‘& (&. This include navigation over Conditions and Instructions.

```
:                                   SCHEMA 6
    :                               Digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    :                               IncOp
        :       <>                  '& (& (S <> S  '& (& '<> 'IncOp 'SCHEMA 6 (Digit
                ><                  '& (& (S <> S  '& (& '>< 'IncOp 'SCHEMA 6 (Digit
```

We use '& to backtrack over (& and (& to backtrack over '&. '& (& is a wildcard that may reference anywhere in the data tree. The other '& (& refers back over the wildcard.

Let us first use SCHEMA 5 when writing instances in SCHEMA 7. In a second meta-schema, SCHEMA 7, we instantiate the IncOp function, where the values are taken from Digit in SCHEMA 5. The ': in the schema reference refers to the common root above SCHEMA 5 and SCHEMA 7.

```
:                                   SCHEMA 7
    :                               IncOp
        S       <>                  S 'IncOp 'SCHEMA 7 ': (SCHEMA 5 (IncOp
        :       <>                  Arg (0
                ><                  Func (1
        :       <>                  Arg (1
                ><                  Func (2
        :       <>                  Arg (2
                ><                  Func (3
        ....
        :       <>                  Arg (8
                ><                  Func (9
        :       <>                  Arg (9
                ><                  Func (0
```

Note that Arg (0 is the value of Arg. We may replace 0 by empty, ie. Arg (.

Now, we want to refer to SCHEMA 6 rather than SCHEMA 5.

```
:                                   SCHEMA 8
    :                               IncOp
        S       <>                  S 'IncOp 'SCHEMA 8 ': (SCHEMA 6 (IncOp
        :       <>                  '& (& (0
                ><                  '& (& (1
        :       <>                  '& (& (1
                ><                  '& (& (2
        :       <>                  '& (& (2
                ><                  '& (& (3
        ....
        :       <>                  '& (& (8
                ><                  '& (& (9
        :       <>                  '& (& (9
                ><                  '& (& (0
```

In the schema of the application, ie. SCHEMA 9, we define a Product. If the Product is inserted (by the end user), this is represented as a Condition with the insertion command I on the Product. The Condition is on the Product; hence, the Product and everything in it is executed only when the Product is inserted.

The same schema has a global attribute, # products, which counts the number of Products in the application. This number is incremented by one, whenever a new Product instance is inserted. This update is performed by the Inc function.

The Inc function takes SCHEMA 9 (IncOp as its schema. Note that we write ‘: for backtracking over SCHEMA 9, because this backtracking will be instantiated into its population, which will have another name tag.

The Inc function takes its value via the Condition from # products and gives its function value via an Instruction to # products.

The Inc function looks up in its schema, and finds the matching component of IncOp, ie. the matching argument value. It looks up the assigned function value, and instantiates this into the function value of Inc, which refers to # products.

products takes Digit as its domain. See the schema reference.

```
:                                       SCHEMA 9
    :                                   Product
        <>      I                       (nserted)
        :                               Inc
                S       <>              S ‘Inc ‘Product ‘: ‘: (SCHEMA 8 (IncOp
                :       <>              : ‘Inc ‘Product ‘: (# products
                        ><              : ‘Inc ‘Product ‘: (# products
    :                                   # products
        S                               <> S ‘# products ‘: ‘: (SCHEMA 6 (DIgit
```

We observe that despite having had some clutter with stating the meta-schemata, the schema of the application becomes simple.

We may now define a POPULATION at the same level as SCHEMA 9, and state a schema reference to SCHEMA 9. The users may insert a list of Product instances, and the above calculations will be executed on each.

We have shown the principles of specification and execution. In practical applications, we will defer the specification of logic – of the Inc function - to the Distribution layer, which will not be addressed here.

15 Mapping to Phenomena

So far, we have defined data and operations on data. These data may or may not describe anything. The data describe anything only if the mapping to phenomena is explicitly stated.

In most applications, we do not define this mapping. We are completely happy with having just the data. In some cases, we have only informal mappings to phenomena. Quite often, the existence of phenomena for every data item is a pure superstition.

The mapping between data and their phenomena shall be isomorphic, but need not be onto either way. This means that there may be data items that correspond to no phenomenon, and there may be phenomena that are not described by any data.

```
:                                       EXTERNAL TERMINOLOGY POPULATION
        D <>                            ‘ ‘ (PHENOMENA

        :                               Country
                :                       Name (NORWAY
                D <>                    ‘ ‘ ‘ (PHENOMENA (:
                :                       Inhabitant
                        :               Name (JOHN
                        D <>            ‘ ‘ ‘ ‘ (PHENOMENA (: (:
                        :               Loved-person
                                D <>    ‘ ‘ ‘ ‘ ‘ (PHENOMENA (: (:, :
                                <>      ‘ ‘Country (Inhabitant (Name (MARY
                        :               Loved-person
                                D <>    ‘ ‘ ‘ ‘ ‘ (PHENOMENA (: (:, :, :
                                <>      ‘ ‘Country (Inhabitant (Name (LILL
                :                       Inhabitant
                        :               Name (MARY
                        D <>            ‘ ‘ ‘ ‘ (PHENOMENA (: (:, :
                :                       Inhabitant
                        :               Name (LILL
                        D <>            ‘ ‘ ‘ ‘ (PHENOMENA (: (:, :, :

:                                       PHENOMENA
        :
                :
                        :
                                <>      ‘ ‘: (:, :
                        :
                                <>      ‘ ‘: (:, :, :
                :
                :
```

We note that denotation mappings – with the role D - from data to phenomena must be explicitly stated, as shown.

In the above Figure, we have not shown the schemata. For the instances to have denotations, their classes – in schemata - will have to have denotations, as well. This observation is contradictory to many philosophical discussions on what exist.

In the above Figure, we have represented each phenomenon by one pixel. We may show more details by adding more pixels. Note that the references are made by pixels only, as well. We may additionally add name tags to the phenomena, but have not done so.

The above Figure has a lot of philosophical implications, which we chose not to discuss here. Look into the book on *Formal Description of Anything in the Universe* Part 1 on these issues.

We observe that the structure of phenomena is isomorphic to the structure of data. However, the mapping may not be onto either way. There may be data, eg. the Name-s that do not correspond to any phenomenon. And, there may phenomena that are not described by any data.

When using Predicate Calculus, most often, set structures are used to represent the phenomena. Set structures are very different from the structure of phenomena shown in the above Figure, and sets have no behaviour. The phenomena in the above Figure may have even richer behaviours than their data.

In this paper, we will not give a deep discussion on Set Theory. We just note that Set Theory is not an appropriate theory for representing phenomena. See the book again, Part 2.

16 Conclusion

We have shown that Predicate Calculus does not provide the required structure, shown in the previous section, for describing anything.

Predicate Calculus assumes a flat/global enumeration of all phenomena, and provides a truth-value calculus on the associations between these. This approach has nothing to do with descriptions, as shown.

We have replaced Predicate Calculus with a proper language for descriptions. We have replaced predicates, terms, truth-values, logical connectives, quantifiers, variables and the Type-token principle. The new language is more applicable and more fundamental than Predicate Calculus.

We call the new language for Existence logic. Existence logic provides a very different world view from Predicate Calculus and Set Theory. The book *Formal Description of Anything in the Universe* gives a more complete discussion of the topics touched upon in this paper.

Bibliography

See http://movinpics.com/publications.html