A formalization of reasoning not derived from standard predicate logic

Gregory S. Tseytin

V.I.Smirnov Res. Inst. for Mathematics and Mechanics, Bibliotechnaya pl., 2, Stary Petergof 198904, Russia

Abstract

A system of formal reasoning, termed object-oriented logic (OOL), is presented which is based on the logical concepts present in modern computer programming and not related directly to predicate logic. It is expected to be efficient in deriving simple conclusions when information is centered around objects and in combining independently designed subsystems. Some examples are discussed, as well as relation to other systems for logic and/or programming.

Key words: non-predicate logic; logic programming; object-oriented programming; open data model.

1 Introduction

This is an extended version of a paper submitted to a special issue of Theoretical Computer Science series of Elsevier Science.

It is a refinement of the previous paper [3] [1] where a formal system with some reasoning capabilities was proposed, which was not a variant of the predicate logic.

This should not be construed as a totally fresh start in formalization of thinking: an independent tradition of "logic" can be traced back both to programming concepts (in the first place, object-oriented programming) and to early work in artificial intelligence systems (pattern matching, backtracking, frames, semantic nets). The fundamental concepts of object-oriented programming, like *object*, *class*, *inheritance*, *encapsulation*, etc., belong to description of thinking rather than to programming techniques. Approaches to modularity

[1] I am grateful to Grigori Mints for some stimulating discussions.

and code reusability, as well as to interoperability of independently designed programs offer another source of insight; the most important part of those seems to be the open model of data, where an entity is described by a set of named properties without specifying the full set of names in advance.

Whereas all those things are of logical nature, it is not very clear what are their counterparts in predicate logic, and it is more appropriate to see this as an alternative approach to formal description of thinking. A system of derivation rules (like the one proposed in this paper) can be called a logic if the objects manipulated in the derivation can be interpreted as facts or assumptions. It can be also regarded as an instrument of knowledge representation, and this is not a contradiction because *any* logic is intended to represent and manipulate knowledge.

On the other hand, as this work starts from ideas developed in programming languages, the proposed system can be also viewed as a programming language (assuming some additional rules for ordering of derivations — this is what Prolog does starting from a form of predicate logic). In fact it can be so used; there is an experimental implementation and some programming techniques can be seen from the examples in section 6. Generally, conclusions obtained by inference can result in actions, and there is no sharp dividing line between reasoning and action except for purely mathematical applications.

The present work is also based on many years of this author's work with programming tools based on some of the abovementioned principles (see paper [2], containing also a short survey of previous applications of semantic, or associative, networks).

The aim of this paper is to present a definition of a logical system, *object-oriented logic* (OOL), and to show its utility by some examples of its application. Its essential feature is that general rules are represented by the same type of network-like structures as individual facts, with small additional information, and, except for that information, the rules look exactly like facts that can be derived from them. Of course, any general rule or any algorithm in any language can be encoded like data, but a LISP function encoded as a list doesn't look like its arguments or results; pattern-matching techniques come closer to this goal.

When a logical derivation system is proposed, some questions naturally arise, regarding its validity and its strength in comparison with other systems. The question of validity is formally irrelevant because OOL has no axioms, and any conclusion deemed invalid can be blamed on the premises. Similarly, there is no obvious mapping of rules of this system into an established logical calculus, which makes any comparisons difficult.

This paper contains no mathematical results regarding OOL; at this stage it seems to be more important to refine the definitions by considering various applications. However one example presents a natural mapping of Prolog into OOL suggesting that it can represent arithmetics and resursive functions (for a formal proof of its algorithmic universality it would be easier to use recursive functions directly). Some possible variations of the definition as well as the relationship to various approaches found in other systems will be discussed in the concluding section 8.

The examples in this paper have been computer-tested; the previous paper [3] had some bugs, see http://www.math.spbu.ru/∼tseytin/assomat3.ps for its "debugged" version.

2 The data model

The data to which OOL is applied are represented as an *association net* — a finite collection of objects and relationships. An object is either a *constant* or a *variable*. In the sequel single capital letters, possibly with subscripts, will be used as meta-variables, while longer words will represent constants.

The relationships can be of one of the following types (A, B, and C stand for objects):

$$A = B, \tag{1}$$

$$A(B) = C, \quad \text{where } A \text{ is a constant},$$

$$A \in B.$$

This set of relationship must contain a relationship

$$A = A \tag{2}$$

for any object A and must be closed under the following inference rule

$$\frac{\begin{array}{c} XSY \\ S = T \end{array}}{XTY}. \tag{3}$$

where X,Y,S, and T are any sequences of symbols generating valid relationships in (3). In the computer implementation closedness with respect to these rules is maintained automatically, i.e., all inferences are performed immediately after introduction of a new relationship.

Informally, the difference between constants and variables is that each constant has its own "identity" and cannot be equated (as in (1)) to a different constant, while variables can be equated both to constants and other variables.

A relationship of the form (1), where A and B are two distinct constants, is called a *contradiction.* Formally, having a contradiction doesn't forbid to continue inference using other rules, but we will never care what happens in such a situation, and detection of a contradiction will be tantamount to a stop (some rules will tell to revert to a previous state of the association net).

The informal meaning of the relationship $A(B) = C$ is that A represents the *name* of a property of the object B, and C is the value of the property. Some nouns in natural languages are most appropriately represented here as property names. E.g., describing a family, we can state:

$$Husband(X) = Y, \tag{4}$$

$$Wife(Y) = X. \tag{5}$$

The meaning of a relationship $A \in B$ is that B is regarded as a set and A is one if its members. Being a set doesn't preclude an object from participating in other relationships. There are no restrictions on membership (e.g., a set can be a member of itself) and two sets with an equal membership do not need to be equal. This type of relationship can be used to represent what is known in other models as "many-to-one" relationships.

To *assert* a relationship will mean to add it to the association net with all relationships that can be obtained by applying the inference rules (2) and (3).

On the whole association nets are similar to "semantic nets" used in some artificial intelligence systems; the difference is that semantic nets usually show relationships between general concepts whereas we are interested in relationships between individual objects. General concepts will be considered as well, as individuals in their own right, and with some additional (reserved) property names to show how they apply to instances of the concepts. Normally a general concept is *not* the set of its instances, and its description is related to its *intension* rather than *extension.*

3 Restrictions on operations

Some operations can change association nets. At every moment only a single association net is considered, and an operation can be applied only to this net and possibly some selected objects in the net. The concept of association net includes some restrictions on what operations may be defined on it. There are two kinds of such restrictions.

3.1 Restrictions imposed by openness

An *extension* of an association net $\mathcal{N}$ is a net obtained from $\mathcal{N}$ by adding some new objects and/or asserting some relationships. As for objects, any number of constants or variables not present in $\mathcal{N}$ can be introduced. A constant or a variable will be termed *new* if it is not present in the association net.

There are some types of extensions that are defined as *admissible*, and every operation shall be *tolerant* of any admissible extensions.

Let $\mathcal{O}$ be an operation on association nets, possibly with parameters (selected objects) $X_1, \ldots, X_n$, and denote the result of its application to an association net $\mathcal{N}$ by $\mathcal{O}(X_1, \ldots, X_n)[\mathcal{N}]$. Then we can state the tolerance restriction as follows:

> for any association net $\mathcal{N}_1$, obtained from $\mathcal{N}$ by an admissible extension, $\mathcal{O}(X_1, \ldots, X_n)[\mathcal{N}_1]$ is an extension of $\mathcal{O}(X_1, \ldots, X_n)[\mathcal{N}]$. (Admissibility of the latter extension is not assumed.)

We are not going to use these notations in the sequel because we will consider at any moment only one association net.

There are two types of admissible extensions:

(a) addition of a single equality where the right member is a new variable;

(b) for any new object X, assertion of any number of relationships of the form $A(X) = B$, $C \in X$, and $X(A) = B$ (provided X is a constant), unless the resulting association net contains a contradiction.

Informally, (a) specifies introduction of a new notation, and (b) addition of irrelevant information. However, tolerance of such extensions is a powerful restriction on operations. Here are two important special cases of this restriction.

No operation may inquire, for some A and B, where A is a constant, whether the property A is "defined" for B, i.e. whether the association net contains a relationship of the form $A(B) = X$. This is because adding such a relationship (with a new variable in place of X) is an admissible extension. However it is all right to inquire whether there is a *constant* X satisfying such a relationship.

No operation may depend on a full listing of properties of a certain object

A, even of those with constant values. This is because adding a relationship $X(A) = B$ for some constant B and a new constant X is an admissible extension.

On the other hand, it is all right for an operation to inquire whether a given relationship is present in the association net, or whether a given object is equal to a constant, or whether the association net contains a contradiction. It is also legal to list all members of a given set, i.e., to obtain, for a given A, a list of objects X such that for any X in the list $X \in A$, and any Y such that $Y \in A$ is equal to one of the members of the list.

The intent of the restrictions is to allow independently designed applications to work on the same association net without intervening with each other, or to interact in some predesigned way through shared access to some objects and property names.

3.2 The "logic versus programming" restriction

This principle says that

> no operation shall remove relationships from the association net except for reverting to a previously saved state if the association net contains a contradiction.

This is contrary to standard programming practices where the memory state is allowed to change through assignment of new values to program variables. In the context of this work this would mean that a relationship of the form $A(X) = 1$ could be replaced by $A(X) = 2$ by an appropriate assignment; but the two relationships cannot be both present without a contradiction ($1 = 2$), hence the first of them would have to be removed before introduction of the second. Introducing this restriction is, in a way, an arbitrary choice, but it is this restriction that makes the system closer to logic than to programming languages. Some implications of this decision will be discussed later.

Thus, any changes in the association net reflect only the progress of our reasoning, and not the changing state of the world. If we were to extend this system it to a temporal logic, time would have to be introduced explicitly.

There is another kind of operation removing objects or relationships from the association nets that could be considered: reversing admissible extensions. In fact, if one is sure that some objects or relationships will never be used again on may forget about them (and an implementation might attempt to do so as a sort of "garbage collection"). We prefer not to consider this option here.

3.3 Extended notations

Admissible extensions of the type (a) above allow for introduction of terms, which can simplify the notation. The term $A(B)$, where A is a constant, will *evaluate* to some X such that $A(B) = X$. More complex terms can be built on this basis in the usual way, and then used in equalities or in membership relationships. To reduce such formulas to the original notation one has for every subterm $A(B)$ to find (or build) the respective X, and substitute X for $A(B)$; after a finite number of such steps we'll have a simple equality of type (1) and a number of equalities used to evaluate the intermediate terms.

Some other kinds of extended notation will be used as well.

Const(X) will stand for a true equality $(X = X)$ if there is a constant equal to X, and for a contradiction if there is no such constant (any particular contradiction can be chosen).

$*$ (an asterisk) will stand for a new constant (there will be different constants for different occurrences of $*$).

4 Handling general concepts — a simple example

In the family example in section 2 X and Y were assumed to be individuals (a wife and a husband). We'll show how the same relationships can be treated in more general terms. Let $MarriedWoman$ be an object representing the *class* of married women and let us formulate the generalization of the previous example. Let $HerHusband$ be a new constant, and let us write the following relationships:

$$Husband(MarriedWoman) = HerHusband, \tag{6}$$

$$Wife(HerHusband) = MarriedWoman. \tag{7}$$

The intention is to allow to derive relationships like (4) and (5) for any instance of $MarriedWoman$. Let X be such an instance, and this fact will be written down as

$$X \textbf{ is-a } MarriedWoman \tag{8}$$

(this use of $MarriedWoman$ shows another possibility to represent natural language nouns).

This will trigger a *mapping* of the general concept, $MarriedWoman$, onto this instance. We need a little more than just the two new equalities (6) and (7): we have to describe the property $Husband$ of $MarriedWoman$ and the property $Wife$ of $HerHusband$ as *copied* properties. Then the mapping will proceed as follows:

$$MarriedWoman \text{ is mapped to } X. \tag{9}$$

$MarriedWoman$ has a copied property $Husband$, which instructs us to map $Husband(MarriedWoman)$ to $Husband(X)$, where $Husband(X)$ probably, but not necessarily, evaluates to a new variable, so

$$HerHusband \text{ is mapped to } Husband(X).$$

$HerHusband$ has another copied property, $Wife$, and this will result in one more mapping, of $Wife(HerHusband)$ to $Wife(Husband(X))$, and, due to (7),

$$MarriedWoman \text{ is mapped to } Wife(Husband(X). \tag{10}$$

The mapping is complete now, but from (9) and (10) we obtain:

$$Wife(Husband(X)) = X$$

(this is a piece of new knowledge about X).

Let us extend the example by adding more properties to $MarriedWoman$ and $HerHusband$:

$$Gender(MarriedWoman) = Feminine, \tag{11}$$

$$Gender(HerHusband) = Masculine.$$

Obviously we don't want to map the new properties like the copied properties before, because $Gender(X)$ should be exactly $Feminine$ and not a newly constructed variable. The property $Gender$ of both $MarriedWoman$ and $HerHusband$ will be described as *invariant*. The rules for invariant properties applied to (9) will produce

$$Gender(X) = Gender(MarriedWoman),$$

hence

$$Gender(X) = Feminine;$$

similarly, we get

$$Gender(Husband(X)) = Masculine.$$

Thus we have two different *modes* of properties, *copied* and *invariant*. One more mode is needed if we have another class, $MarriedMan$, and want to derive, for the same X, that

$$Husband(X) \textbf{ is-a } MarriedMan. \tag{12}$$

For this to happen we have to give $HerHusband$ one more property, $Category$, which will have both the mode *invariant* and one more mode, *ascribed*:

$$Category(HerHusband) = MarriedMan. \tag{13}$$

In virtue of being invariant the property will be transferred to $Husband(X)$ literally:

$$Category(Husband(X)) = MarriedMan.$$

Then the value of the property will be ascribed to $Husband(X)$, resulting in (12). If the mode of the property were only *ascribed* but not *invariant*, this would mean that a pre-existing property of $Husband(X)$ would have to be used.

Property modes will be shown as superscripts, with c standing for *copied*, i for *invariant*, and a for *ascribed*. Thus (6), (11) and (13) can be written as follows:

$$Husband^{c}(MarriedWoman) = HerHusband,$$

$$Gender^{i}(MarriedWoman) = Feminine,$$

$$Category^{ai}(HerHusband) = MarriedMan$$

(in the last formula we can write "$*$" instead of $Category$ because we don't need to use this name again, and any other name can be used as well).

The details of the superscript notation are given later, in 5.1; it is not an extension of association nets, just a kind of shorthand.

To illustrate other modes of properties consider a different, though related, example: the concept of $Baby$.

On asserting that

$$Z \textbf{ is-a } Baby$$

we want to infer something about its parents. Of course, we will write

$$Mother^c(Baby) = ItsMother, \tag{14}$$

$$Father^c(Baby) = ItsFather. \tag{15}$$

But are we going to add that

$$Husband^c(ItsMother) = ItsFather? \tag{16}$$

The statement resulting from the mapping,

$$Husband(Mother(Z)) = Father(Z)$$

is normally assumed to be true unless something else is known to reject it. So (16) is a formula that represents a *default* assumption, and this has to be represented in OOL. Rather than mark defeasible properties we'll use a "positive" marker: the properties that have to be transferred unconditionally will receive an additional mode, *mandatory* (abbreviated to m), and this has to be added to (14) and (15) as well as to all similar formulas in the preceding example, but not to (16).

One more kind of mapping step is produced by the formula

$$Baby \in Children^c(ItsMother).$$

The mapping extends to elements of a set in a way analogous to copied properties, and the conclusion will be obviously

$$Z \in Children(Mother(Z)).$$

One more mode is needed to represent the following situation. Suppose we need to know if the baby's mother is married or not. We can use the term $Husband(Mother(Z))$ in any case just like we can speak about someone's husband without knowing if there is a real person satisfying the condition. But we need something else in order to find out whether it is a real person and not just an abstract idea. Here we are going to use the distinction between variables and constants. We might require the husband to be a constant, and in this case *Husband* should be marked as an *expected* property (abbreviated to x), in addition to being a copied property. For an expected property it is required that its value at the target of the mapping is a constant, otherwise the mapping is defined to fail (i.e., to produce a contradiction).

5 Classes — a formal definition

Inference in OOL is driven by formulas with **is-a**, like (8). The term to the right of **is-a** is a *class*, i.e., an object with special properties; the whole formula states that the object to the left of **is-a** is an instance of this class. By processing the formula we build new relationships in the association net as well as new formulas of this type, which can in turn be processed for further conclusions. The inference rules will be stated in this section. There are several kinds of classes distinguished by their $\#type$ properties.

5.1 Superscript notation for property modes

Modes of properties are represented in the net by putting the names of the properties in special sets, one for each mode; the sets are attached to the object as values of its properties with special names (thus the same property can have different modes for different objects).

The following property modes will be used: *ascribed*, *copied*, *expected*, *invariant*, *mandatory*, and *synchronous*. The last mode has no direct effect on the rules defined in this section and will be considered later, in 5.6.5. The reserved names for properties are the same, but prefixed with #. Thus, for an object X, $\#ascribed(X)$ will be the set names of its properties with *ascribed* mode, etc.

More formally, when a term of the form $A^{\text{superscripts}}(B)$ occurs in a formula, its value is the same as for just $A(B)$, and additionally the following relationships are added to the association net (if they are not already present):

$A \in \#ascribed(B)$, if a occurs in the superscripts,
$A \in \#copied(B)$, if c occurs in the superscripts,
$A \in \#expected(B)$, if x occurs in the superscripts,
$A \in \#invariant(B)$, if i occurs in the superscripts,
$A \in \#mandatory(B)$, if m occurs in the superscripts,
$A \in \#synchronous(B)$, if s occurs in the superscripts.

Hence, if several occurrences of $A^{\text{superscripts}}(B)$ with the same values of A and B are found, the effect will be as if the union of the superscript sets were used each time.

Some formulas consist only of one term, without an equality or "$\in$" relationship; then the property modes shown in superscripts are the only information carried by such a formula.

5.2 Frames

A class like those discussed in the previous section is called a *frame*. The fact that a class F is a frame is shown by the relationship

$$\#type(F) = \#frame$$

which can be abbreviated to

frame F.

A formula

$$X \textbf{ is-a } F$$

is processed as follows: F is mapped to X using a new constant M (which will never be referred to after the map is complete and hence can be discarded by an implementation).

The mapping is defined by a number of inference rules using auxiliary formulas starting with Map or Map_1. We start with the formula

$$\text{Map}(M, F, X) \tag{17}$$

and apply the following rules.

$$\frac{\text{Map}(M, F, X)}{M(F) = X} \tag{18}$$

(i.e., part of the mapping is that the map value for F is set to X, like in the examples in the preceding section).

$$\frac{\begin{array}{c}\mathrm{Map}(M, F, X)\\ P \in \#mandatory(F)\end{array}}{\mathrm{Map}_1(M, F, X, P)}$$

(with mandatory properties we proceed unconditionally).

$$\frac{\begin{array}{c}\mathrm{Map}(M, F, X)\\ \mathrm{Map}_1(M, F, X, P) \text{ doesn't produce a contradiction}\end{array}}{\mathrm{Map}_1(M, F, X, P)} \tag{19}$$

(this happens if the property is not mandatory; the meaning of "produce a contradiction" will be discussed later in 5.6.1).

$$\frac{\begin{array}{c}\mathrm{Map}_1(M, F, X, P)\\ P \in \#expected(F)\end{array}}{\mathbf{Const}(P(X))} \tag{20}$$

(if the expected property of the target doesn't have a constant value the mapping will lead to a contradiction).

$$\frac{\begin{array}{c}\mathrm{Map}_1(M, F, X, P)\\ P \in \#copied(F)\end{array}}{\mathrm{Map}(M, P(F), P(X))} \tag{21}$$

(a copied property requires propagation of the mapping).

$$\frac{\begin{array}{c}\mathrm{Map}(M, F, X)\\ E \in F\end{array}}{\begin{array}{c}M(E) \in X\\ \mathrm{Map}(M, E, M(E))\end{array}} \tag{22}$$

(elements of sets map and propagate like copied properties).

$$\frac{\begin{array}{c}\mathrm{Map}_1(M, F, X, P)\\ P \in \#invariant(F)\end{array}}{P(X) = P(F)}$$

(the value of an invariant property is simply transferred to the target).

$$\frac{\begin{array}{c}\mathrm{Map}_1(M, F, X, P)\\ P \in \#ascribed(F)\end{array}}{X \ \textbf{is-a}\ P(X)} \tag{23}$$

(this is how new **is-a** formulas are generated).

5.3 Frames for sets

A special variation of frames are frames for sets which define a mapping of both a set and its selected element. Such frames can be of two types, depending of whether all of the elements of the set have to be selected in turn or only one element, but not leading to a contradiction (so this is analogous to quantifiers over elements of finite sets). The frame F is mapped to the set and a "sample element" S is mapped to one of its elements. This is encoded in the association net as either

$$\#type(F) = \#frame\text{-}all \tag{24}$$

or

$$\#type(F) = \#frame\text{-}one, \tag{25}$$

and, in both cases,

$$\#sample\text{-}element(F) = S. \tag{26}$$

A shortened notation for (24) and (26) is

$$S \ \textbf{all-in}\ F. \tag{27}$$

A shortened notation for (25) and (26) is

$$S \ \textbf{one-in}\ F. \tag{28}$$

If X is a set and $Y \in X$, simultaneous mapping of F and S to X and Y is like in the preceding subsection 5.2 and uses the same inference rules, but instead of a single starting formula (17) two formulas are used:

$$\mathrm{Map}(M, F, X),$$

$$\mathrm{Map}(M, \#\mathit{sample\text{-}element}(F), Y)$$

(see (26)).

To process a formula

$$X \textbf{ is-a } F$$

in the case (27) one has to perform the mapping as described for each Y such that $Y \in X$ (taking each time a new constant M).

In the case of (28) each Y such that $Y \in X$ is tried with a similar process, but the state of the association net before this process is saved and in case of a contradiction is restored; this ends when either an Y is found for which no contradiction results or if all elements of X are exhausted, in which case a contradiction is added to the association net anyway.

In [3] a simpler approach to set operations is proposed, but it requires a more complex representation of data.

5.4 Ordered conjunctions and disjunctions

Ordered conjunctions and disjunctions are one more kind of classes. If F is a conjunction of F_1 and F_2 a formula

$$X \textbf{ is-a } F$$

is processed by processing sequentially (in this order)

$$X \textbf{ is-a } F_1$$

and

$$X \textbf{ is-a } F_2.$$

If F is a disjunction of F_1 and F_2 a formula

$$X \textbf{ is-a } F$$

is processed as follows. First we save the current state of the association net, then process

$$X \textbf{ is-a } F_1.$$

If we get no contradiction the processing is complete (F_2 is never considered). In case of contradiction we revert to the saved state and process

$$X \textbf{ is-a } F_2.$$

Such conjunctions (disjunctions) are encoded in the association net by the relationship

$$\#type(F) = \#conjunction$$

or, respectively,

$$\#type(F) = \#disjunction$$

and the following relationships

$$\#left(F) = F_1,$$

$$\#right(F) = F_2.$$

The shortened notation to define conjunctions and disjunctions (with any number of members) is as follows:

$$\textbf{conj } F\langle F_1, \ldots, F_n\rangle \tag{29}$$

or, correspondingly,

$$\textbf{disj } F\langle F_1, \ldots, F_n\rangle. \tag{30}$$

Both (29) and (30) can be used as terms within other conjunctions and disjunctions: in such cases it is actually F that is included in the sequence, and its definition as a conjunction or a disjunction is included separately in the association net.

When $n > 2$, n-member conjunction (29) is defined as

$$\textbf{conj } F\langle F_1, \textbf{conj } *\langle F_2 \ldots, F_n\rangle\rangle$$

(the asterisk is used to name the intermediate conjunction), and similarly for disjunctions.

5.5 Built-in pseudoclasses

Some functions needed for communication of this system with the outside world are represented as class-like objects. For example, processing a formula

$$X \textbf{ is-a } \_term\_to\_print$$

will result in whatever constant X stands for being printed out.

A more complex use of pseudoclasses is needed to print formulas looking like sums and products of unknown number of terms, like in [3]; to know if any terms of the sum/product have been printed before the current term, special external objects (with changing state) are attached to some OOL objects.

5.6 Timing considerations and synchronous properties

Timing of inference steps should be irrelevant for pure logic, as well as possible repetition of inference (which shouldn't bring new conclusions anyway). However, there are some reasons why this should be discussed for OOL.

5.6.1 Non-monotonic inference — default properties in frames

The rule (19) for mapping says that it is applicable only if a certain assumption doesn't lead to a contradiction. The problem is how far should we trace the consequences before we decide that it is safe. If we were to trace them "forever", i.e., to be able to revoke the assumption with all its consequences at any future step, we would face a problem of deciding which of a number of such assumptions made earlier should be revoked, without any information to help in such a decision. Such approach seems to cancel out all benefits default reasoning might bring.

If we are to use default reasoning, *a point must be defined for every default assumption beyond which it becomes fully accepted*, even if we'll have to revise it later in view of new facts. This is where timing becomes essential.

The intended timing policy for mapping is that it is done like in recursive calls in programming languages. It means that every time the mapping propagates according to rules (21) and (22) derivation of consequences of the Map formula just obtained must be completed before the processing of other properties of the object (or other elements of the set) that has generated the new Map formula.

Hence the "testing phase" of the rule (19) for default properties extends only to the end of processing of the respective Map formula. After that, the conclusions are irreversible (except for a failure of a previous assumption or of a previous choice of member of a disjunction or sample element of a set, like in 5.4 or 5.3).

5.6.2 Non-monotonic inference due to expected properties

Expected properties, according to the rule (20), are one more source of non-monotonicity and a concern for timing. The rule produces a contradiction in case of absence of the required information. If the information can be supplied by applying other rules, the result, once again, can depend on relative timing of inferences.

5.6.3 Repeated inference may affect sets

The mapping rule (22) generates new set members. If, for any reason, the inference is repeated, a new member will be generated in the same set. The newly generated members are variables, which can be possibly equated to each other at a later stage. But the identifying information is not always available.

Suppose we have an application where a arithmetical formula has to be derived to calculate someone's taxes, and every taxation rule produces a separate member of the sum. What happens if some rule gets processed twice? Double taxation can be avoided if every member of the sum is additionally identified as the taxpayer's property named after a specific type of tax. This will result in equating the two copies of the deduction.

Another example is from [3]. We are deriving differential equations for numbers of two populations of animals from the descriptions of the populations. If one population preys on the other, this will bring additional terms to the death rate of the prey and to the fertility of the predator. If the same population suffers from more than one predator there will be more addends in the death rate. But how to distinguish it from the case when the same preying relationship is processed twice? Although some solution can be probably found, it is noteworthy that this difficulty is related to another device of non-monotonic reasoning, *circumscription*, by which we assume that there are no other predators but those explicitly listed. So there seems to be a relationship between circumscription and the use of sets in OOL.

5.6.4 Interaction with the outside world

Taking actions in the outside world on the basis of inference conducted within OOL is one more reason for taking time into account. The only instance of external action considered in this paper is printing. Printing a complex expression consists of printing elementary symbols, and we don't want them scrambled or unnecessarily repeated due to inappropriate timing or repetitions. The approach adopted in this work is borrowed from Prolog, where some entities looking like predicates in fact specify external (or internal) actions. Another approach would be to exclude such actions from OOL, and instead to allow an external device to observe the state of the association net and to take actions accordingly.

5.6.5 The synchronous mode of properties in frames

No reference to this mode was present in the processing rules given above. This mode is used exactly to control timing and is, in a way, implementation dependent, just like the conventions of 5.6.1.

The *synchronous* mode only makes sense in connection with *ascribed* mode referred to in the rule (23). If, in the context of the rule, we additionally have

$$P \in \#synchronous(F),$$

the resulting formula

$$X \ \mathbf{is\text{-}a}\ P(X) \tag{31}$$

is processed immediately, otherwise its processing is delayed and will start only after the current "**is-a**" formula has been completely processed. (The implementation processes all *ascribed* properties after all other property modes for the same object, so one can expect some information to be available before the processing of (31) begins.)

6 Example — serial/parallel circuits

Consider electric circuits built up of resistors by means of serial and parallel connections. We want to derive numerical relationships between their parameters from the description of the structure of the circuit. This is one of the earliest examples of automatic generation of programs due to Enn Tõugu (also known as E.H.Tyugu), see [1].

Consider a circuit described in terms of its subcircuits — a subcircuit is either a single resistor or a serial or parallel combination of two other subcircuits. The following description seems to be self-explanatory.

$circuit$ **is-a** $serial$
$r1(circuit) = circ1$
$circ1$ **is-a** $resistor$
$value(r(circ1)) = R1$
$r2(circuit) = circx$
$circx$ **is-a** $parallel$
$r1(circx) = circ2$
$r2(circx) = circ3$
$circ2$ **is-a** $resistor$
$value(r(circ2)) = R2$
$circ3$ **is-a** $resistor$
$value(r(circ3)) = R3$
$value(i(circ2)) = I$

(i stands for the current, u for the voltage applied, r for the resistance, $r1$ and $r2$ denote the two subcircuits of a circuit; the whole circuit consists of a resistor $circ1$ connected serially with a subcircuit $circx$, which consists of two parallelly connected resistors $circ2$ and $circ3$).

6.1 Deriving numeric relationships

To pass from this to numeric relationships we introduce some classes representing Ohm's and Kirchhoff's laws. Classes $op3add$ and $op3mult$, described later, represent the relationships between a sum or product and the two operands.

frame $resistor$
$*^{cm}(resistor) = ohm$
$oper1^{cm}(ohm) = i^{cm}(resistor)$
$oper2^{cm}(ohm) = r^{cm}(resistor)$
$res^{cm}(ohm) = u^{cm}(resistor)$
$*^{aim}(ohm) = op3mult$

frame $serial$
$*^{aim}(serial) = resistor$

$$*^{cm}(serial) = kirchhoff\text{-}s$$
$$oper1^{cm}(kirchhoff\text{-}s) = u^{cm}(r1^{cm}(serial))$$
$$oper2^{cm}(kirchhoff\text{-}s) = u^{cm}(r2^{cm}(serial))$$
$$res^{cm}(kirchhoff\text{-}s) = u^{cm}(serial)$$
$$i^{cm}(serial) = i^{cm}(r1^{cm}(serial))$$
$$i^{cm}(serial) = i^{cm}(r2^{cm}(serial))$$
$$*^{aim}(kirchhoff\text{-}s) = op3add$$

frame *parallel*
$$*^{aim}(parallel) = resistor$$
$$*^{cm}(parallel) = kirchhoff\text{-}p$$
$$oper1^{cm}(kirchhoff\text{-}p) = i^{cm}(r1^{cm}(parallel))$$
$$oper2^{cm}(kirchhoff\text{-}p) = i^{cm}(r2^{cm}(parallel))$$
$$res^{cm}(kirchhoff\text{-}p) = i^{cm}(parallel)$$
$$u^{cm}(parallel) = u^{cm}(r1^{cm}(parallel))$$
$$u^{cm}(parallel) = u^{cm}(r2^{cm}(parallel))$$
$$*^{aim}(kirchhoff\text{-}p) = op3add$$

In addition to giving numeric relationships these rules also state that every serial and parallel combination is a resistor, and hence follows Ohm's law as well.

Let us build an association net from all these statements and process all "**is-a**" formulas (we'll remove for the moment the references to $op3add$ and $op3mult$ which are not yet defined).

Following is a part of computer tracing (mapping details have been edited out for brevity). The names starting with ## are generated constants.

```
##2(circuit) = resistor;
u(circ1) = oper1(##3(circuit));
u(circx) = oper2(##3(circuit));
i(circx) = i(circ1);
u(circuit) = res(##3(circuit));
i(circuit) = i(circ1);
circuit is-a resistor;
i(circ1) = oper1(##1(circ1));
r(circ1) = oper2(##1(circ1));
oper1(##3(circuit)) = res(##1(circ1));
##4(circx) = resistor;
i(circ2) = oper1(##5(circx));
i(circ3) = oper2(##5(circx));
```

```
u(circ3) = u(circ2);
oper1(##1(circ1)) = res(##5(circx));
oper2(##3(circuit)) = u(circ2);
circx is-a resistor;
oper1(##5(circx)) = oper1(##1(circ2));
r(circ2) = oper2(##1(circ2));
u(circ2) = res(##1(circ2));
oper2(##5(circx)) = oper1(##1(circ3));
r(circ3) = oper2(##1(circ3));
res(##1(circ2)) = res(##1(circ3));
res(##5(circx)) = oper1(##1(circuit));
r(circuit) = oper2(##1(circuit));
res(##3(circuit)) = res(##1(circuit));
oper1(##1(circuit)) = oper1(##1(circx));
r(circx) = oper2(##1(circx));
res(##1(circ3)) = res(##1(circx));
```

Now define *op3add* and *op3mult*. The important part of this is that they will provide information to compute some quantities in terms of others. Every quantity that can be computed in terms of others will get a *comp-methods* property whose value will be a set of "methods". Each method has a target (the quantity itself), *operand*1, *operand*2, and *operator*, the last being just one of the symbols $\underline{+}$, $\underline{-}$, $\underline{\times}$, and $\underline{/}$ (the symbols are underlined here to distinguish them from ordinary syntactic marks). Both *op3add* and *op3mult* are defined as special cases of a more general class, *op3rel*, differing only in symbols for their direct and inverse operations.

frame *op3rel*
$oper1^{cm}(op3rel) = op1$
$oper2^{cm}(op3rel) = op2$
$res^{cm}(op3rel) = result$
$compop1 \in comp\text{-}methods^{cm}(op1)$
$compop2 \in comp\text{-}methods^{cm}(op2)$
$compres \in comp\text{-}methods^{cm}(result)$
$target^{cm}(compop1) = op1$
$target^{cm}(compop2) = op2$
$target^{cm}(compres) = result$
$operand1^{cm}(compres) = op1$
$operand2^{cm}(compres) = op2$
$operator^{cm}(compres) = op\text{-}direct^{cm}(op3rel)$
$operand1^{cm}(compop1) = result$
$operand2^{cm}(compop1) = op2$
$operator^{cm}(compop1) = op\text{-}inverse^{cm}(op3rel)$

$operand1^{cm}(compop2) = result$
$operand2^{cm}(compop2) = op1$
$operator^{cm}(compop2) = op\text{-}inverse^{cm}(op3rel)$

frame *op3add*
$op\text{-}direct^{im}(op3add) = \underline{+}$
$op\text{-}inverse^{im}(op3add) = \underline{-}$
$*^{aim}(op3add) = op3rel$

frame *op3mult*
$op\text{-}direct^{im}(op3mult) = \underline{\times}$
$op\text{-}inverse^{im}(op3mult) = \underline{/}$
$*^{aim}(op3mult) = op3rel$

6.2 Solving equations — asynchronous approach

The relationships obtained from these classes allow for solving the equations for a certain quantity. We will try to solve the equations to obtain $r(circuit)$, the resistance of the whole circuit from quantities for which values are specified. Unlike the preceding part, this is not a straightforward task. And it does not need to be solvable, though it happens to be in this case — due to the redundant value for $i(circ2)$ which could be cancelled out from the final result if we added a capability for algebraic transformations.

The approach adopted in this subsection is the same as originally used by Tõugu and his colleagues — to extend the set of known quantities by applying whatever rules are available until it includes the desired quantity.

First extend the *op3rel* class by ascribing a new class *a-rule* to *compop*1, *compop*2, and *compop*3.

$*^{aim}(compop1) = a\text{-}rule$
$*^{aim}(compop2) = a\text{-}rule$
$*^{aim}(compres) = a\text{-}rule$

The class *a-rule* is defined as a disjunction of four “subclasses”:

disj $a\text{-}rule\langle get\text{-}target\text{-}value, get\text{-}target\text{-}solved, try\text{-}rule, reschedule\rangle$

An attempt to apply it to an instance of a rule will thus be in four stages, in this order.

First we try to find an existing *value* of the quantity (the rule's *target*), and in case of success the quantity will be marked as *solved* (i.e., will get a *solved* property with a constant value). Expected properties are used here (the corresponding line contains no explicit relationship and is used only to specify property modes).

frame *get-target-value*
$value^{xm}(target^{cm}(get\text{-}target\text{-}value))$
$solved^{im}(target^{cm}(get\text{-}target\text{-}value)) = *$

The next attempt is to see whether the quantity is already *solved*.

frame *get-target-solved*
$solved^{xm}(target^{cm}(get\text{-}target\text{-}solved))$

If both previous attempts fail, we try to make sure that both operands of the rule have been *solved* (it may look strange that the respective relationships are in the end of the list, but the order doesn't matter here). If this is the case this rule is marked as a *good-expression* for its *target* (which may have several candidates for this role), and it is marked as *solved*.

frame *try-rule*
$solved^{im}(target^{cm}(try\text{-}rule)) = *$
$good\text{-}expression^{cm}(target^{cm}(try\text{-}rule)) = try\text{-}rule$
$solved^{xm}(operand1^{cm}(try\text{-}rule))$
$solved^{xm}(operand2^{cm}(try\text{-}rule))$

In case of failure the last resort is *reschedule*. It depends on the way the current implementation works. The implementation puts all "**is-a**" formulas on a queue in the order they are obtained, and processes each of them only once. Here the failed rule is scheduled again in the hope that some time later it will succeed due to new information (note that information here is only added and never changed, and the only reason for a failure is the absence of information at a certain point).

frame *reschedule*
$*^{aim}(reschedule) = a\text{-}rule$

This could proceed indefinitely if there were no solution (no loop detection is implemented now). But it happens to exist, and the process ends. Then, using the *good-expression* properties we can obtain the algebraic expression. This is the result (but we still have to figure out how to print it):

```
((((I+((I*R2)/R3))*R1)+(I*R2))/(I+((I*R2)/R3)))
```

6.3 Solving equations — synchronous approach

With the *try-rule* frame in the preceding subsection we had to test for a prospective expression whether the operands have been solved, and to fail (and try other rules) if that was not the case. Another approach requires on seeing an operand not yet solved we try to solve it immediately, in a recursive fashion, using synchronous mode defined in 5.6.5. The problem, however, will be to avoid infinite recursions which can arise when the recursive quest returns to the quantity for which we were originally seeking a solution.

Unlike the previous approach where numeric relationships initiated the search on their own accord, here we need something to initiate the search for a specific quantity. Thus we start by asserting:

$$r(circuit)\ \textbf{is-a}\ get\text{-}expression,$$

showing the initial goal.

The class *get-expression* is defined as a combination of disjunctions and conjunctions.

$\mathbf{disj}\ get\text{-}expression\langle$
  $get\text{-}value,$
  $get\text{-}solved,$
  $\mathbf{conj}\ *\langle$
    $\mathbf{disj}\ *\langle$
      $get\text{-}started,$
      $\mathbf{conj}\ *\langle set\text{-}started, find\text{-}expression, set\text{-}solved\rangle$
      $\rangle,$
    $get\text{-}solved$
    $\rangle$
  $\rangle$

The first two members of the disjunction represent the attempts to see if the quantity in question already has a value or has been marked as *solved*. Each of them fails if the expected property is not found.

frame *get-value*
$value^{xm}(get\text{-}value)$

frame *get-solved*
$solved^{xm}(get\text{-}solved)$

The last member of the disjunction is a conjunction of two steps, of which the second, *get-solved*, tests whether the first has resulted in a solution. The essential work is done by the first step, which is itself a disjunction. The disjunction first tests, using *get-started*, if the quantity has been marked as started; if this is the case, we have a recursive call for the same quantity, and the disjunction returns without setting the *solved* flag, which will cause the containing conjunction to fail at the second step. This is what should happen for a recursive loop.

If the last disjunction doesn't succeed at the first step (i.e., the call is not a loop), we proceed to the innermost conjunction,

$$\textbf{conj} \ * \langle set\text{-}started, find\text{-}expression, set\text{-}solved\rangle. \tag{32}$$

Following are the auxiliary frames used to set or test flags.

frame *get-started*
$started^{xm}(get\text{-}started)$

frame *set-solved*
$solved^{im}(set\text{-}solved) = *$

frame *set-started*
$started^{im}(set\text{-}started) = *$

So the conjunction (32) first sets the *started* flag, then does the productive work with *find-expression*, and finally, in case of success, sets the *solved* flag.

frame *find-expression*
$*^{asim}(comp\text{-}methods^{cm}(find\text{-}expression)) = find\text{-}one$

What *find-expression* does is to delegate the work from the quantity to the set of its computation methods, constructed by *op3rel*. It requests an immediate (*synchronous*) application of the following frame, *find-one*, to the set. This one is of *#frame-one* type, defined in 5.3.

$$
\begin{aligned}
&\textit{an-expression}\ \textbf{one-in}\ \textit{find-one}\\
&\textit{good-expression}^{cm}(\textit{target}^{cm}(\textit{an-expression})) = \textit{an-expression}\\
&*^{asim}(\textit{operand1}^{cm}(\textit{an-expression})) = \textit{get-expression}\\
&*^{asim}(\textit{operand2}^{cm}(\textit{an-expression})) = \textit{get-expression}
\end{aligned}
$$

It will try in succession each computational method in the set, represented in the frame by *an-expression*, and will either stop successfully, or fail. For each expression in the set it does the following:

- assigns it to the quantity (the *target* of the method) as its *good-expression* (this will be retracted in case of failure);
- recursively (and synchronously) applies *get-expression* (the original big disjunction) to each of its operands.

This eventually finds the solution (the same as by the method of the preceding subsection). In this particular example this approach turns out to be more efficient than the previous one, and, moreover, in case of the absence of a solution it fails explicitly rather than looping indefinitely.

6.4 Printing — synchronous approach

To print the expression obtained by one of the preceding methods the synchronous approach seems to be most appropriate due to the recursive structure of the expression. And in fact, it is done in a rather straightforward fashion.

Let *print2* be the name of the class we are going to use to print the expression. To initiate printing we state

$$r(\textit{circuit})\ \textbf{is-a}\ \textit{print2}.$$

The same class will be applied to subexpressions which have to be treated differently depending on whether it is a plain value of another expression. Hence two “subclasses”, *printv* for plain values and *printx* for expressions.

$$\textbf{disj}\ \textit{print2}\langle \textit{printv}, \textit{printx}\rangle.$$

Printing a value is simple; a built-in pseudoclass $\_print$ is used, a synonym of $\_term\_to\_print$ introduced in 5.5 (this one looks more like a verb than a noun though noun-like names for classes seem to be more appropriate).

frame $printv$
$*^{asim}(value^{mcx}(printv)) = \_print$

Printing an expression consists, as one might expect, of five stages: printing a left parenthesis, then the first operand, then the operator, the second operand, and finally the right parenthesis. Parentheses and operators are special printable objects, shown here with an underline.

conj $printx\langle printlpar, printop1, printopr, printop2, printrpar\rangle$

frame $printlpar$
$reprl^{cim}(printlpar) = \underline{(}$
$*^{asim}(reprl(printlpar)) = \_print$

frame $printrpar$
$reprr^{cim}(printrpar) = \underline{)}$
$*^{asim}(reprr(printrpar)) = \_print$

Both operands and the operator are retrieved from *good-expression* properties set by the solver. This is how they are printed. The operands are recursively processed by $print2$.

frame $printop1$
$*^{asim}(operand1^{cm}(good\text{-}expression^{cm}(printop1))) = print2$

frame $printop2$
$*^{asim}(operand2^{cm}(good\text{-}expression^{cm}(printop2))) = print2$

frame $printopr$
$*^{asim}(operator^{cxm}(good\text{-}expression^{cm}(printopr))) = \_print$

The synchronous approach to printing relies heavily on synchronous properties in frames. However, one can do the same in an asynchronous fashion, except that the physical call to *_print* must be synchronized anyway, or else the symbols might be printed in an unpredictable order.

This is the alternative version of $print2$. The idea is that an expression gets printed when it is marked ready for printing. In fact it is not the expression but a specially constructed object that bears the mark, and the expression to be printed is pointed to by the *target* property of the object. This is essential because an expression cannot be marked for printing forever, it should be ready only in the current instance of printing, which is represented by the constructed object. Likewise, this object is marked as done when the actual printing starts. If the object is not ready for printing it can be retried later.

This is how the initial instance of printing is built.

$$\begin{array}{l}
\textbf{frame}\ print2 \\
x \in print2 \\
target^{cm}(x) = print2 \\
ready^{im}(x) = * \\
*^{aim}(x) = print2l
\end{array}$$

A new object is created by mapping of x, it receives the target (the expression to print) and a *ready* flag is set. This object is then submitted to $print2l$.

$$\begin{array}{l}
\textbf{disj}\ print2l\langle \\
\quad \textbf{conj}\ *\langle \\
\quad\quad get\text{-}ready, \\
\quad\quad \textbf{disj}\ *\langle get\text{-}done, print\text{-}value, propagate\text{-}print\rangle \\
\quad\quad \rangle, \\
\quad reschedule\text{-}print \\
\quad \rangle
\end{array}$$

This disjunction consists of an inner conjunction and *reschedule-print* called (like in the subsection 6.2) in case of failure. The first step of the conjunction is to check for the *ready* flag and to fail (and drop out to rescheduling) in case of failure. If the *ready* flag has been set, we proceed to the innermost disjunction.

The first step of the disjunction is to test if it is *done* and then just to exit without printing. The next attempt is to print the target expression as a plain value, and if it is not a plain value, to call *propagate-print*.

Both *print-value* and *propagate-print* set the *done* flag which will persist in case of success. *print-value* also sets the *ready* flag to the next printable item, pointed to by the *print-next* property (we will see later how this property is set).

print-value contains the only occurrence of a synchronous property, related to physical printing.

frame *print-value*
$done^{im}(print\text{-}value) = *$
$ready^{im}(print\text{-}next^{cm}(print\text{-}value)) = *$
$*^{asim}(value^{mcx}(target^{cm}(print\text{-}value))) = \_print$

frame *get-ready*
$ready^{xm}(get\text{-}ready)$

frame *get-done*
$done^{xm}(get\text{-}done)$

Rescheduling is done like in the asynchronous solver.

frame *reschedule-print*
$*^{aim}(reschedule\text{-}print) = print2l$

The following frame, *propagate-print*, splits printing of an expression into five steps, *pp-lpar*, *pp-op*1, *pp-operator*, *pp-op*2, and *pp-lpar*. The first of them immediately gets the *ready* flag. Successors are appointed via *print-next* properties, so that the successor of *pp-lpar* (printing left parenthesis) is *pp-op*1 (printing the first operand), etc. The successor of the whole expression becomes the successor of *pp-lpar*, the last constituent. Parenthesis and operators are converted to "values" and printed with *print-value*, propagating the *ready* flag to the successor.

frame *propagate-print*
$done^{im}(propagate\text{-}print) = *$
$*^{cm}(propagate\text{-}print) = pp\text{-}lpar$

$*^{aim}(\textit{pp-lpar}) = print2l$
$ready^{im}(\textit{pp-lpar}) = *$
$value^{im}(target^{cm}(\textit{pp-lpar})) = \underline{(}$
$\textit{print-next}^{cm}(\textit{pp-lpar}) = \textit{pp-op}1$
$*^{aim}(\textit{pp-op}1) = print2l$
$target^{cm}(\textit{pp-op}1) =$
$\quad operand1^{cm}(\textit{good-expression}^{cm}(target^{cm}(\textit{propagate-print})))$
$\textit{print-next}^{cm}(\textit{pp-op}1) = \textit{pp-operator}$
$*^{aim}(\textit{pp-operator}) = print2l$
$value^{cm}(target^{cm}(\textit{pp-operator})) =$
$\quad operator^{cxm}(\textit{good-expression}^{cm}(target^{cm}(\textit{propagate-print})))$
$\textit{print-next}^{cm}(\textit{pp-operator}) = \textit{pp-op}2$
$*^{aim}(\textit{pp-op}2) = print2l$
$target^{cm}(\textit{pp-op}2) =$
$\quad operand2^{cm}(\textit{good-expression}^{cm}(target^{cm}(\textit{propagate-print})))$
$\textit{print-next}^{cm}(\textit{pp-op}2) = \textit{pp-rpar}$
$*^{aim}(\textit{pp-rpar}) = print2l$
$value^{im}(target^{cm}(\textit{pp-rpar})) = \underline{)}$
$\textit{print-next}^{cm}(\textit{pp-rpar}) = \textit{print-next}^{cm}(\textit{propagate-print})$

This is how expressions can be printed almost without synchronization.

7 Emulating Prolog

We define a simple mapping of Prolog into OOL. Only basic features of Prolog will be emulated: no cuts, no predefined predicates (numbers should be represented using 0 and the successor function), no axioms added or deleted at run time, etc. And even this mapping is not absolutely accurate, due to the treatment of self-embedding in unification. However the sublanguage of Prolog reproduced here is sufficient to program any recursive function. A more accurate mapping might be built but at this point our aim is to show a simple relationship.

7.1 Representation of terms and clauses

We need some predefined constants:

- property names $Arity$ and $Name$ (predicate or function symbol arity and name);
- numerals $(0, 1, 2, \ldots)$ to represent arities of predicate and function symbols;
- property names $Arg_1, Arg_2, \ldots$ to retrieve arguments in terms and atomic goals;
- property names $Subgoal_1, Subgoal_2, \ldots$ to retrieve subgoals in clauses.

Prolog predicate names and function names will be also considered OOL constants. Prolog constants will be treated as function symbols of zero arity.

Given a collection of Prolog clauses and/or query goals, we select for every term or atomic goal T occurring therein an OOL variable[2] $\mathcal{O}[T]$, and then assert the following relationships.

If $T = f(T_1, \ldots, T_n), \quad n \geq 0$, and T occurs in a clause

$$\begin{aligned} Name^{im}(\mathcal{O}[T]) &= f, \\ Arity^{im}(\mathcal{O}[T]) &= n, \\ Arg_i^{cm}(\mathcal{O}[T]) &= \mathcal{O}[T_i], \quad 1 \leq i \leq n. \end{aligned} \tag{33}$$

If T occurs in a goal rather than a clause the relationships will be the same except that the property modes (the superscripts) are omitted.

Additionally, for a clause

$$A : -A_1, \ldots, A_n, \quad n > 0,$$

we assert

$$Subgoal_i(\mathcal{O}[A]) = \mathcal{O}[A_i], \quad 1 \leq i \leq n$$

and

$$*^{asim}(\mathcal{O}[A]) = ClauseClass_n. \tag{34}$$

Here $ClauseClass_n$ is an OOL class, depending only on n and constructed as follows:

$$\textbf{conj } ClauseClass_n\langle SubgoalClass_1, \ldots, SubgoalClass_n\rangle, \quad n > 1,$$

[2] This paper doesn't provide a special notation for OOL variables, but any new term will evaluate to a variable.

or (for $n = 1$)

$$\begin{array}{l} \textbf{frame } ClauseClass_1 \\ *^{asim}(ClauseClass_1) = SubgoalClass_1 \end{array}$$

where the definition of $SubgoalClass_i$ (depending only on i) is

$$\begin{array}{l} \textbf{frame } SubgoalClass_i \\ Name^{asm}(Subgoal_i^{cm}(SubgoalClass_i)) \end{array} \tag{35}$$

(the second line specifies no relationship and is used only to show property modes).

Finally, for every predicate symbol P, whose alternatives are $A_1, \ldots, A_n$ we define

$$\textbf{disj } P\langle \mathcal{O}[A_1], \ldots, \mathcal{O}[A_n] \rangle, \quad n > 1, \tag{36}$$

or (for $n = 1$)

$$\begin{array}{l} \textbf{frame } P \\ *^{asim}(P) = \mathcal{O}[A_1]. \end{array} \tag{37}$$

We also define an inverse mapping for $\mathcal{O}$, denoted by $\overset{-1}{\mathcal{O}}$. A newly generated OOL variable v will be mapped to a new Prolog variable. If relationships analogous to (33) are asserted for v

$$\begin{array}{l} Name^{im}(v) = f, \\ Arity^{im}(v) = n, \end{array}$$

(probably with some values for $Arg_i(v)$), they will translate into the following equality

$$\overset{-1}{\mathcal{O}}[v] = f(\overset{-1}{\mathcal{O}}[Arg_1(v)], \ldots, \overset{-1}{\mathcal{O}}[Arg_n(v)]). \tag{38}$$

7.2 Unification

Let G be an atomic goal, and A the head of a clause and let the formula

$$\mathcal{O}[G] \textbf{ is-a } \mathcal{O}[A] \tag{39}$$

be processed in OOL.

It initiates a mapping of $\mathcal{O}[A]$ onto $\mathcal{O}[G]$. Using the notation of 5.2, including M for the mapping constant, we write

$$\mathcal{M}(T, U) \tag{40}$$

for

$$\text{Map}(M, \mathcal{O}[T], \mathcal{O}[U]).$$

So we start with $\mathcal{M}(A, G)$, and the process can recursively propagate to subterms. If we get at $\mathcal{M}(T, U)$, where

$$\begin{aligned} T &= r(T_1, \ldots, T_m), \\ U &= s(U_1, \ldots, U_n), \end{aligned}$$

mapping of $Name^{im}$ and $Arity^{im}$ properties will force $r = s$ and $m = n$, which will result in a contradiction in case the equalities do not hold. Then, mapping of Arg_i^{cm} properties will produce

$$\mathcal{M}(T_i, U_i), \quad 1 \leq i \leq m. \tag{41}$$

If in (40) T is not a variable and U is a variable, the mapping rules will result in creating new relationships

$$\begin{aligned} Name(\mathcal{O}[U]) &= r, \\ Arity(\mathcal{O}[U]) &= m, \\ Arg_i(\mathcal{O}[U]) &= v_i, \quad 1 \leq i \leq m, \end{aligned} \tag{42}$$

where v_i are newly created OOL variables.

Correspondingly, $\overset{-1}{\mathcal{O}}[v_i]$ will be new Prolog variables, and we will have

$$\mathcal{M}(T_i, \overset{-1}{\mathcal{O}}[v_i]), \quad 1 \leq i \leq m$$

(a variant of (41) which, if T_i is not a variable, will cause further recursion), and, in virtue of (38), a new equality which will be called a *proto-unifier*:

$$U = f(\overset{-1}{\mathcal{O}}[v_1], \ldots, \overset{-1}{\mathcal{O}}[v_m]).$$

We replace the current occurrence of U in G by the right member of the proto-unifier (but leave $\mathcal{O}[U]$ intact), and proceed further.

After the mapping recursion is complete consider one more source of inference. If T is a variable and occurs in A more than once the map will result in additional equalities: if

$$\mathcal{M}(T, U),$$
$$\mathcal{M}(T, V),$$

we get, using (18),

$$M(\mathcal{O}[T]) = \mathcal{O}[U],$$
$$M(\mathcal{O}[T]) = \mathcal{O}[V],$$

hence

$$\mathcal{O}[U] = \mathcal{O}[V]. \tag{43}$$

This equality will propagate similarly to (40) but using the basic inference rule (3) instead of mapping rules. If

$$U = s(U_1, \ldots, U_n),$$
$$V = t(V_1, \ldots, V_k),$$

we get

$$Name(\mathcal{O}[U]) = Name(\mathcal{O}[V]),$$
$$Arity(\mathcal{O}[U]) = Arity(\mathcal{O}[V]).$$

This will produce $s = t$ and $n = k$, which may result in a contradiction if the names or the arities do not match. If they do we further have

$$Arg_i(\mathcal{O}[U]) = Arg_i(\mathcal{O}[V]), \quad 1 \leq i \leq n,$$

hence

$$\mathcal{O}[U_i] = \mathcal{O}[V_i], \quad 1 \leq i \leq n$$

(similarly to (41)).

If in (43) U is a variable and V is not, or the other way round, we proceed like in the similar case in (40), obtain proto-unifiers (we will call them *induced equalities* instead), but don't replace the variable by the corresponding term.

Note that no similar derivation is needed if U is a variable and occurs in G more than once. In this case the same object $\mathcal{O}(U)$ will receive mapping several times, but after U is replaced by the term from its proto-unifier it will be no longer considered a variable.

The whole process ends in a finite number of steps because if a step is an application of (40) it is done only once for each subterm occurrence in A, and each of the new equalities of (43) type will not produce more propagation steps than the number of subterms in its members. The "net" result of the process is a number of proto-unifiers and induced equalities.

If A and G unify this process will not produce a contradiction. Indeed, if $A = G$, all (40) relationships obtained during the process will have $T = U$, and there will be no contradiction. If A and G can be made equal by a unifying substitution, this will only result in adding more relationships of (33) type. Comparing the process for the unified and original versions we will see that the only new OOL relationships for the original version that are not derived for the unified version are either (42) relationships that cannot produce a contradiction or (43) relationships which correspond to tautologic equalities in the unified version.

Though unification of A and G is a sufficient condition for a successful completion of the mapping the converse is not true. Let A and G be $p(X, X)$ and $p(X, f(X))$. Processing (39) will not produce a contradiction, it will just generate an induced equality $X = f(X)$, turning X into a sort of infinitely recursive term $f(f(\ldots))$.

However, if the mapping completes without a contradiction we can attempt to build a unifying substitution. For variables in G we will use proto-unifiers obtained during the mapping (occurrences of variables inside the substituted terms should also be replaced by the corresponding terms from their proto-unifiers, if any). Then for each occurrence of a variable T in A substitute U from the corresponding (40) equation with T. The resulting atomic goals will be equal, but this is not a true substitution because different occurrences of T are replaced by different terms, though equal in virtue of the induced equalities. We may attempt a consistent replacement of the left members of all induced equalities by the corresponding right members to eliminate the equalities. If this process ends, we actually have a unification. But if the right member of an induced equality contains, of will contain after some substitutions, the variable in the left member, the process will never end.

This is the limitation of the current emulation of Prolog by OOL, and for the time being we have to rely on the fact that all recursive functions can be represented in Prolog in a way that never leads to such situations.

7.3 Derivation of new goals

Let C be a clause

$$A:-A_1,\ldots,A_n,\quad n>0.$$

Process the OOL formula

$$\mathcal{O}[G]\ \textbf{is-a}\ \mathcal{O}[C]. \tag{44}$$

It will result in a mapping if $\mathcal{O}[C]$ to $\mathcal{O}[G]$. Part of the process will be the mapping of $\mathcal{O}[A]$ which we have just considered. Suppose it hasn't produced a contradiction, and we accept this as a proof of successful unification (we can proceed even if there is no unification but then it is not clear if we get what Prolog semantics prescribes). By the same mapping $\mathcal{O}[G]$ will get properties $Subgoal_i$ for $1 \leq i \leq n$. $Subgoal_i(\mathcal{O}[G])$ will be equal to a copy of $\mathcal{O}[A_i]$, with additional relationships based on the unifying substitution. Then (34) will result in immediate processing of

$$\mathcal{O}[G]\ \textbf{is-a}\ ClauseClass_n,$$

which in turn will resolve into a sequence of formulas

$$\mathcal{O}[G]\ \textbf{is-a}\ SubgoalClass_i,\quad 1\leq i\leq n.$$

Processing of this formula will map $SubgoalClass_i$ to $\mathcal{O}[G]$, and this, in virtue of (35), will to map $Subgoal_i(SubgoalClass_i)$ to $Subgoal_i(\mathcal{O}[G])$, where the latter is a copy of $\mathcal{O}[A_i]$ with additional relationships. Hence

$$Name(Subgoal_i(\mathcal{O}[G]))$$

is the predicate name of A_i, we denote it by P_i. Further, again from (35), we'll have to process synchronously the formula

$$Subgoal_i(\mathcal{O}[G])\ \textbf{is-a}\ P_i.$$

This, in view of (36) or (37) will lead to alternatives for P_i which will be tried on the subgoal in succession.

Note that all calls to subgoals are synchronous, and hence within the scope of the original call (44) which means that a unification failure in any of those will backtrack the whole processing of (44). Of course, within the disjunction a failure of any alternative but the last will reverse only the failing alternative.

8 Relation to other approaches

The principal intended use of OOL is in applications where general rules are applied to particular cases, including derivation of relationships between different levels of representation of a complex system (like in the example of section 6). Probably it can also handle symbolic transformations or computations and generation of program code. It is not intended to handle traditional applications like numerical analysis or text or image processing, but it should be able to interact with them. Sometimes it will encroach on the realms of other systems.

8.1 OOL and predicate logic

It is not very clear how OOL can be embedded into the traditional predicate logic; probably high-order logic will be needed to handle *ascribed* properties when this mode is used without *invariant*, like in (35). It is still harder to imagine how to embed in OOL a complex predicate formula with alternating embedded quantifiers. However OOL can be used as a metalanguage to describe derivations.

Logical inference with OOL is much simpler and efficient than with traditional predicate logic and its variations. This is because in OOL information is organized around objects and normally only a small neighborhood of an object has to be considered for an inference step; this is what should be expected in simple applications. Of course, this does not mean that OOL claims to solve complex problems of proof-searching: real difficulty cannot be removed by simple change of formalism, it can only be shifted to another stage of the work. But simple problems will be handled simply in OOL.

Non-monotonic enhancements to the predicate logic seem to fit well in OOL. Defaults and, partly, circumscription were discussed in 5.6.1 and 5.6.3. A new type of non-monotonicity is added in OOL: failure due to lack of information, as with *expected* properties.

It is not very clear how to extend OOL to comprise temporal and causal logic.

8.2 OOL and object-oriented programming

As said before, OOL uses many ideas from object-oriented programming, and we will discuss the correspondence between some features of both.

Classes in object-oriented programming approximately correspond to OOL frames. The principal difference is that in object-oriented programming usually an object is created as an instance of a class, while in this system classes are ascribed to objects dynamically, and possibly more than once.

The distinction between *copied* and *invariant* properties corresponds to the distinction between instance members and class, or static, members.

Inheritance relationships are represented by the combined use of *invariant* and *ascribed* modes in frames; the first causes a predefined class C to be added to an object K as a value of some property, the second causes this value to be ascribed as a new class to the object. The object K can be the "root" (i.e., the frame itself), and then it is the closest analogue to inheritance: whenever the class K is ascribed to an object, class C will be ascribed as well, i.e., K is derived from C (see, e.g., (37)).

Another situation is when K is not the root object, like in

frame F
$P^{cm}(F) = K$
$*^{aim}(K) = C,$

or, shorter,

frame F
$*^{aim}(P^{cm}(F)) = C.$

In the notation of C++ this will look like

class F {
  $C\ P$;
  ...
};

Still another use of *ascribed* is without *invariant*; in such cases one expects that the target of the mapping already has the property to ascribe; use of *expected* mode might be in order in such cases. Consider the following example.

> **frame** A
> $N^{axm}(A)$;

let

$$N(B) = C;$$

then

$$B \textbf{ is-a } A \tag{45}$$

will result in

$$B \textbf{ is-a } C.$$

C was attached to B before (45), and the effect of (45) is analogous to calling the *method* N on B.

Encapsulation in object-oriented languages consists in restricting access to some members of classes to external code. In OOL, names of properties are global and serve as the only interface between independently written pieces of code. Whoever knows the name of a certain property of an object, will be able to access its value. Hence encapsulation can be achieved by using names of properties unknown to other programs. Generation of unique names is one device that can be used for the purpose. Implicit prefixing of some constants pertaining to particular frames might also be a good idea.

This seems enough to justify the use of the name "object-oriented logic".

OOL can also be used as a programming language roughtly equivalent to other object-oriented languages; the most salient distinction is that OOL doesn't use assignment statements, but this is known not to be critical, and the examples in section 6 show how one can do without assignments (essentially it is like replacing them with initializations).

For those interested in logical foundations of object-oriented programming OOL can serve as a simple instance of an object-oriented system.

8.3 OOL and conventional programming

The examples in section 6, except for the 6.1 which can be considered an example of OOL "native" style, all show how usual programming tricks can be reproduced in OOL environment, circumventing the limitations imposed by 3.2. Both tasks solved in those examples, unraveling a system of equations and recursive printing of terms, are not peculiar to OOL, and the OOL code doesn't look very nice: the meaningful part wrapped in several layers of tinkering with various flags and/or reschedulings.

However this shows how with conjunctions and disjunctions OOL can handle serial execution and conditionals. Boolean values are best represented by absence or presence of a contradiction. A conditional statement with only **then** part can be represented as an ordered conjunction with the condition as its first member. A conditional with both **then** and **else** parts is represented as an ordered disjunction, in which the second member doesn't need to repeat the test because it will be never executed if the condition is true. [3] Curiously, logical negation can be expressed in terms of conditionals, hence in terms of conjunction and disjunction! But this only means that the disjunction used here is not a "true" disjunction.

OOL, conceived as a logical system opposed in style to conventional programming, actually has some constructs prescribing ordering of actions; but each of them has an unordered counterpart. There are synchronous calls, but asynchronous calls are possible too. There are ordered conjunctions and disjunctions, but frames with **all-in** and **one-in** provide similar devices without ordering. Ordering seems indispensable in interaction with the outside world, but it has to be seen what is its role in "pure" OOL.

OOL seems to lack many features of the mechanism of procedure calling present in standard procedural languages (like Pascal). A "procedure" (an OOL class) receives only one "parameter" and has no access to the global environment and the identifiers defined in the scope of the procedure declaration. But this depends on the way the only parameter is used. The general philosophy of OOL is that what we pass to a "call" is not a specific parameter but rather an environment and it is up to the "procedure" to look for the parameters it needs by navigating from objects to their properties; in an application,

[3] I partly owe this treatment of conditionals to Andrei Akopiants (personal communication), but he didn't use ordered disjunctions.

two procedures with nearly the same functionality might use different information to accomplish the same task. The usual procedural mechanisms can be emulated in OOL by building and passing objects organized like environments.

Finally, it should be noted that OOL lacks some safety provided by identifier checking and type checking in conventional languages; neither spelling errors nor type errors can be detected early, and this is due to the deliberate freedom in using names and absence of static typing.

8.4 OOL and Prolog

OOL looks similar to Prolog in that both are derived from restricted fragments of logic and both are intended for use as programming tools. OOL's use of pseudoclasses for procedural actions is a direct borrowing from Prolog. However the non-logical part of OOL is obviously much less elaborate than in Prolog, and it is not clear what will be its relative role in the future. The approach to backtracking adopted in OOL is also similar to that of Prolog.

However Prolog never served as a source of ideas in designing this system (because, as it was mentioned before, it is derived from ideas in programming). Even though some kinship between the two systems was vaguely perceived by the author, the formal relationship between them was not known before the embedding described in section 7 was found.

References

[1] K.A.Tinn, E.H.Tyugu, M.I.Unt, Sistema modul'nogo programmirovaniya dlya TsVM Minsk-22, in VKP-2. Trudy Vsesoyuznoy konferentsii po programmirovaniyu, Zasedanie G (Novosibirsk, 1970) 23–29 (in Russian).

[2] G.S.Tseytin, Programmirovanie na assotsiativnyh setyah, in: G.V.Orlovskiy, ed., *EVM v proektirovanii i proizvodstve, vyp. 2.* ("Mashinostroenie", Leningr. otdelenie, Leningrad, 1985) 16–48 (in Russian, English version at http://www.math.spbu.ru/~tseytin/netslong.html).

[3] G.S. Tseytin, Association Nets: An Alternative Formalization of Common Thinking, in: S.Adian and A.Nerode, eds., *Logical Foundations of Computer Science. Proceedings 4th International Symposium, LFCS'97.* Lecture Notes in Computer Science 1234 (Springer, Berlin, 1997) 385–398.