Association Nets: an Alternative Formalization of Common Thinking

(improved intermediate version, Oct. 25, 1997)

G.S.Tseytin

St.Petersburg University, Russia
e-mail: tseyting@acm.org

Abstract. The development of programming as well as some studies in artificial intelligence have produced their own conceptualizations used in the representation of tasks being solved and methods applied. This paper is an attempt at logic-like representation of a programming system, based on the open data model and object-oriented programming, earlier developed and extensively used by the author for various tasks including authomatic programming and natural language understanding. This is suggested as an alternative to traditional logic systems in description of common thinking.

1 Introduction

The classic mathematical logic, after being so successful in the analysis of standard mathematical thinking, since long tends to embrace neighboring areas like computer programming and commonsense reasoning. The typical approach to commonsense reasoning is to modify the classic techniques by adding features like non-monotonicity, modality, temporal reasoning, actions, causality, fuzziness, etc. The aim of this paper is to present a different approach, not directly relying on classic techniques, but also leading to a mathematical description of some features of common thinking, though possibly not the same as in other approaches.

In fact, computer programming is pursuing the same goal, though not stating it explicitly. At the initial stages programming was too machine-oriented, based on such concepts as storage location or goto and producing entangled spaghetti-like programs that clearly called for some kind of logical verification. However, methods of logical verification didn't develop very fast and clearly lagged behind the practial needs of this rapidly expanding art. What is more, many actual applications of programming didn't yield themselves to rigorous mathematical description, and their specifications were based instead on practical experience and informal thinking. As a result, programming moved, all on its own, from machine-oriented to human-oriented concepts, with symbolic programming languages, databases, modularity and object-oriented programming as important milestones on this road. With program code represented in more humanlike terms, the need for verification with external methods decreased.

Studies in artificial intelligence also developed some forms of knowledge representation as an alternative to the classic approach. Some of those forms converged with the tendencies of programming. One can observe deep analogies between M.Minsky's theory of frames, object-oriented programming and some concepts of Prolog (seen as a programming language rather than a proof-searching technique). The central concepts of each of the three approaches, i.e., frames, classes, and predicates defined by lists of axioms, are clearly related to each other. So it seems that these concepts capture some important elements of human cognition.

One more feature long present in some AI languages is the open data model – property lists with dynamic retrieval of properties by their names with no lists of names specified in advance; another example of this is found in semantic nets. In conventional programming this was avoided until recently because deemed inefficient. But development of Internet put emphasis on interoperability of independently designed applications, which called for exactly this feature.

The system presented in this paper was originally conceived as a programming tool systematically using the open data model in various programming and AI tasks, in the first place in natural language understanding. Very soon it also acquired object-oriented features in a very permissive form. An early version of it is presented in [1], for natural language applications see [2, 3]. For a philosophic background see [4]. This paper is an attempt to present a version of this system to make it look as close as possibly to logic rather than programming, so it can be seen as a kind of alternative logic. No attempt is made to study this system mathematically; instead some examples are given to illustrate the way it can be used. Some analogies with concepts of object-oriented programming (OOP) will be mentioned later.

2 Constants and variables

We assume an unlimited supply of both constants and variables. Constants will have a concrete representation (e.g., words in a predefined alphabet) but we won't consider this representation here.

The difference between constants and variables is that each constant has its own identity and may not be equated to a different constant, while variables may be equated to each other or to constants.

A unique constant (unique variable) is a constant (a variable) created on a specific occasion and distinct from all constants (variables) similarly created on other occasions.

3 Association nets

An association net is a finite collection of formulas of the following forms (A,B,C, etc. stand for constants of variables)

$$A = B$$

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

$$A \in B,$$

closed under the following inference rules

$$\frac{A = B}{A = A}$$

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

(where X, Y, S, and T are any sequences of symbols generating valid formulas in the above rule).

Some other types of formulas and related inference rules will be introduced later, but they are not regarded as part of the association net and are only needed to perform some operations. The last rule applies to those types of formulas as well.

A formula of the form

$$A = B,$$

where A and B are two distinct constants is called a contradiction.

In the sequel, unless explicitly stated otherwise, we will be considering one single association net, and will not mention it directly.

The expression

$$\textbf{assert } F,$$

where F is a list of one or more formulas, will mean a change in the association net resulting from addition of F and all other formulas that can be inferred.

4 Interpretation

In applications, both constants and variables are interpreted as real-world entities.

A formula $A = B$ is a statement that A and B refer to one same entity.

A formula $A(B) = C$ is interpreted as a statement about a property of the entity referred to by B; properties are distinguished by names (which can be, e.g., words of a commonly used language), A is the name of the property, and C is its value, referring to another (or possibly the same) entity. Examples:

$Husband(Alla) = Philip$
$Mother(Christine) = Alla$

A formula $A \in B$ means that the entity referred to by B, apart from any other properties it may have, can be interpreted as a finite set, and A refers to one of its elements.

The collection of formulas is meant to reflect some part of one's knowledge of the world; in applications, an essential step will be to combine two nets and establish additional relationships between them (e.g., to combine a net representing some facts with another net representing previous knowledge of the world – "axioms"). Some restrictions stated in the next section are intended to preserve the integrity of the parts brought together.

Any changes in the association net are meant to reflect the progress of reasoning and/or decision-making. They are not interpreted as changes over time (for a time logic, special entities for moments of time and special relationships including such moments will have to be introduced).

Details of representation of constants are not considered here.

5 Principles of self-discipline

An extension of an association net N is a net obtained from N by one or more **assert** operations. If N_1 is an extension of N, an operation on N, resulting in N', is said to be *tolerant* of this extension if its application to N_1 results in an extension of N'.

All operations we are going to define shall be tolerant of the following types of extensions:

(a) **assert** *a single equality where the right member is a unique variable* (introducing a new notation);

(b) *for any constant or variable X not occurring in the net* **assert** *any number of formulas of the types $A(X) = B$, $C \in X$, and $X(A) = B$ (when X is a constant), unless this results in a contradiction* (this is an accessibility restriction: no operation shall be able to access X from these types of formulas).

The second rule requires tolerance of irrelevant information. No definition of an operation shall rely on the complete listing of the association net, except for some specially introduced primitives. This is needed because different applications can use partially overlapping data without interfering with each other, but possibly cooperating through shared data. A special case of this is the third type of added formula, $X(A) = B$, because it implies that there shall be no way of listing all properties defined for an item. Normally different applications use different, possibly overlapping, sets of properties.

Another principle of self-discipline requires that

no operation shall remove formulas from an association net except for return to a specially saved previous state in case of contradiction.

In practice, one can remove formulas from the net by reversing extensions (a) and (b) above ("garbage collection"), but, in virtue of the tolerance principle, this will not affect the results, so one can think that this operation, violating the last restriction, is never used.

These principles don't forbid to inquire whether the net contains any given formula, or to look for a formula of the form $A(B) = X$ with given A and B, or to list all formulas $X \in A$ for a given A.

As seen from these rules, there is no requirement of monotonicity (tolerance is promised only for some types of extensions but not for all). But all sources of non-monotonicity will be explicitly identified; they are two, and both related to the primitives allowed to scan the whole association net.

One of them is the test whether the association net contains a contradiction (which is not forbidden by (b)); the other is discussed in connection with procedures.

6 Terms and evaluation

Terms can be built out of constants and variables using the constructor $T_1(T_2)$, T_1 and T_2 being terms.

To evaluate a term of the form $A(B)$, where A is a constant and B a constant or a variable, one has to look for a formula $A(B) = C$; if there is no such formula extension (a) of the preceding section is used to add one. C is called the value of $A(B)$ in the given association net.

More complex terms can be evaluated by first evaluating innermost subterms. But sometimes this process will stop without a result. In evaluating a term like $A(B)(C)$ one has to evaluate $A(B)$, but if the value isn't equal to any constant there will be no result.

In the sequel terms sometimes will be used for brevity instead of their values with an implicit precondition that the values can be built.

7 Procedures

A procedure can be associated with a constant. A procedure can be applied to a constant or variable, changing the net and possibly performing some external actions (like i/o). For simplicity we'll use the constant to refer to the associated procedure without further explanations.

An application of a procedure to an entity should be construed as an attempt to ascribe to the entity a certain conceptual category represented by this procedure. If the attempt succeeds, the knowledge is extended. If it fails (i.e., produces a contradiction), one can try something else.

In OOP, the closest analogy to these procedures are classes. But here we don't generate a new instance of a class; instead, we ascribe a class to an existing object.

We introduce a new type of formula:

$$A \ \textbf{is-a} \ B, \tag{1}$$

representing an intermediate step in the process of reasoning. The only way to use such a formula is to apply to A the procedure associated with B. However this transition from the formula to action can be deferred (see below).

Some procedures are primitives, not defined here (but tolerant to the extensions described above).

Other procedures are defined here in terms of properties of the constants with which they are associated. Some particular constants essential in their definition will be denoted by symbols starting with $\#$; the constants are chosen to be distinct from any unique (2) constants or variables. (The choice of representation is as arbitrary as any decisions about Gödel numbering.)

Some auxiliary types of formulas and inference rules will be needed to apply procedures. One type of such formula is

$$\mathbf{Const}(X)$$

with the following inference rule:

if the association net contains no formula equating X to a constant, a contradiction is inferred from $\mathbf{Const}(X)$. (For a contradiction in such cases one fixed equality is chosen, no matter which one.)

This is the second operation requiring a complete scan of the net and thus producing non-monotonicity. Other types of auxiliary formulas and rules will be shown later.

We define five types of procedures, distinguished by means of the property $\#type$. The types are frame, frame-all, frame-one, conjunction and disjunction. Let C be a constant, with which we associate a procedure.

7.1 Frames

If $\#type(C) = \#frame$ the procedure is a frame.

The application of a frame to a target consists in mapping the frame to the target. More formally, the following steps are taken:

- let M be a unique constant;
- map C to X with M;
- optionally, discard M by reversing the appropriate extension.

(Mapping corresponds to frame instantiation and Prolog unification.)

Mapping. In the simplest case mapping C to X consists in building around X a copy of the structure of properties already existing around C (while some of them might have existed at X before), and possibly imposing some equalities among properties of C on the respective properties of X (this can propagate recursively).

Five sets of property names (probably overlapping and possibly empty) are needed to map a frame C:

- $\#copied(C)$, the set of properties for which new copies are built;
- $\#invariant(C)$, the set of "invariant" (not copied) properties;
- $\#expected(C)$, the set of properties whose values are sought from the target;
- $\#ascribed(C)$, the set of properties whose values must be "ascribed" to the target (in addition to C) by means of **is-a** statements;
- $\#mandatory(C)$, the set of properties not marked as defaults that can be overridden by other information.

(In OOP, copied properties correspond to instance variables, and invariant properties to class variables. "Ascribed" values are a wide generalization of the OOP concepts of derived classes and of method calls.)

Formally, mapping is done with the use of some auxiliary types of formulas and inference rules. The auxiliary formulas start with Map or Map_1, and the rules are

$$\frac{Map(M,C,X)}{M(C)=X}$$

$$\frac{\begin{array}{c}Map(M,C,X)\\ E\in C\end{array}}{\begin{array}{c}M(E)\in X\\ Map(M,E,M(E))\end{array}} \tag{2}$$

$$\frac{\begin{array}{c}Map(M,C,X)\\ P\in \#mandatory(C)\end{array}}{Map_1(M,C,X,P)}$$

$$\frac{\begin{array}{c}Map(M,C,X)\\ Map_1(M,C,X,P)\quad \text{doesn't produce a contradiction}\end{array}}{Map_1(M,C,X,P)} \tag{3}$$

$$\frac{\begin{array}{c}Map_1(M,C,X,P)\\ P\in \#expected(C)\end{array}}{\mathbf{Const}(P(X))}$$

$$\frac{\begin{array}{c}Map_1(M,C,X,P)\\ P\in \#copied(C)\end{array}}{Map(M,P(C),P(X))}$$

$$\frac{\begin{array}{c}Map_1(M,C,X,P)\\ P\in \#invariant(C)\end{array}}{P(X)=P(C)}$$

$$\frac{\begin{array}{c}Map_1(M,C,X,P)\\ P\in \#ascribed(C)\end{array}}{X\ \textbf{is-a}\ P(X)}$$

To map C onto X with M, **assert** $Map(M,C,X)$.

The rule (3) needs a special discussion. Its obvious meaning is to provide for defaults: a default can be used unless this leads to a contradiction. But if the derivation of new formulas proceeds indefinitely it can be an unsolvable problem to determine whether there ever will be a contradiction. Without expansion of "is-a" formulas (1) some obvious restrictions will guarantee that the process is finite (if X is not "accessible" from C none of the inference steps will increase the set of potential arguments for the M-mapping). If "is-a" formulas are used immediately, resulting in new applications of procedures, it is easy to build an infinite process, and even to simulate a Turing machine. If the application is deferred, there is a danger that a default conclusion accepted as not leading to a contradiction will produce a contradiction later. Thus the rule for default is not completely defined. For the time being we'll accept the interpretation that the test for contradiction is performed before expansion of any "is-a" formulas obtained during the mapping process.

Both default (non-mandatory) and "expected" properties of a frame generate non-monotonicity. The intuitive meaning of "expected" properties is that there is no point applying a procedure in the absence of required information.

The rule (2) could be replaced by somewhat more elaborate rules distinguishing between "copied" and "invariant" elements of C, but for simplicity it can be left as it stands now.

7.2 Conjunctions and disjunctions

If $\#type(C) = \#conjunction$ (or $\#type(C) = \#disjunction$) the procedure associated with C is a conjunction (respectively, a disjunction), defined by means of a sequence of other procedures.

To apply a conjunction to a target one applies all elements of the sequence (to the same target) in succession as long as the association net doesn't contain a contradiction.

To apply a disjunction to a target one attempts to apply all elements of the sequence in succession until a successful attempt, if any. If no attempt was successful, a contradiction is **assert**ed.

To attempt to apply a procedure (to a target) means to save the current state of the association net, then to apply the procedure, and in case of a contradiction to revert to the saved state.

For a more formal definition we will use constants $\#first$, $\#next$, $\#nil$, and $\#body$.

To apply a conjunction C to X, and-apply $\#first(C)$ to X.

To and-apply F to X,

- if $F = \#nil$ stop;
- apply $\#body(F)$ to X;
- if there is a contradiction, stop;
- and-apply $\#next(F)$ to X.

To apply a disjunction C to X, or-apply $\#first(C)$ to X.

To or-apply F to X,

- if $F = \#nil$ **assert** a contradiction and stop;
- save the current state of the association net;
- apply $\#body(F)$ to X;
- if there is no contradiction stop;
- revert to the saved state;
- or-apply $\#next(F)$ to X.

7.3 Frames for sets

If $\#type(C) = \#frame\text{-}all$ (or $\#type(C) = \#frame\text{-}one$) the procedure associated with C is like a conjunction (respectively, a disjunction) of subprocedures over all members of the set represented by the target.

In applying C to a target X, suppose a Y is selected such that

$$Y \in X.$$

Similarly to the section 7.1, we'll use mapping as follows (note the second item added for this case):

- let M be a unique constant;
- map $\#sample\text{-}element(C)$ to Y with M (see 7.1);
- map C to X with M;
- optionally, discard M by reversing the appropriate extension.

In the case of $\#frame\text{-}all$ the above process is repeated for all possible choices of Y (i.e., for all elements of the set X).

In the case of $\#frame\text{-}one$ it suffices to complete the process for at least one choice of Y; we attempt, like in 7.2, to do it for various Y, until a value is found that doesn't lead to a contradiction, or until all values have been tried, which will mean a contradiction on the whole application of C to X.

8 Example 1: family

Consider a frame associated with the item X (in this and subsequent examples frames will be built up of constants only):

$\#type(X) = \#frame$
$Mother(X) = Y$
$Father(X) = Z$
$Husband(Y) = Z$
$Gender(Y) = Feminine$
$Gender(Z) = Masculine$
$Mother \in \#copied(X)$
$Mother \in \#mandatory(X)$
$Father \in \#copied(X)$
$Husband \in \#copied(Y)$
$Husband \in \#mandatory(Y)$
$Gender \in \#invariant(Y)$
$Gender \in \#mandatory(Y)$
$Gender \in \#invariant(Z)$
$Gender \in \#mandatory(Z)$

Try apply X to $Christine$ of the section 4. X will map to $Christine$, Y to $Alla$, and Z to $Philip$. Since $Father$ is not in $\#mandatory(X)$ the conclusion

$Father(Christine) = Philip$
will not be made in the presence of incompatible information.

9 Example 2: simulation

Consider computer simulation of a group of two populations which we'll call unicorns and griffins (this will relieve us of the need of giving realistic relationships and data).

$Unicorns$ **is-a** $Population$
$Griffins$ **is-a** $Population$

We build a model of population that will allow to build arithmetic relationships between the number of the population, its fertility (birth rate) and death reate; the death rate will consist of the basic death rate to which something can be added for reasons not shown in this model. Auxiliary procedures for $PlainValue$, Sum, and $Product$ will be defined later. We'll assume for simplicity that all copied and invariant properties are also mandatory, and will not write out the respective formulas.

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

$Number(Population) = vNumber$
$Number \in \#copied(Population)$
$Exptype \in \#ascribed(vNumber)$
$Exptype \in \#invariant(vNumber)$
$Exptype(vNumber) = PlainValue$

$Fertility(Population) = vFertility$
$Fertility \in \#copied(Population)$

$BasicDeathRate(Population) = vBasicDeathRate$
$BasicDeathRate \in \#copied(Population)$

$DeathRate(Population) = vDeathRate$
$DeathRate \in \#copied(Population)$
$Exptype \in \#ascribed(vDeathRate)$
$Exptype \in \#invariant(vDeathRate)$
$Exptype(vDeathRate) = Sum$
$vBasicDeathRate \in vDeathRate$

Before we continue let us apply this to one of the populations, just to see how it works. Newly introduced unique variables will be prefixed with an underscore.

$Number(Unicorns) = \_uNumber$
$\_uNumber$ **is-a** $PlainValue$
$Fertility(Unicorns) = \_uFertility$
$BasicDeathRate(Unicorns) = \_uBasicDeathRate$
$DeathRate(Unicorns) = \_uDeathRate$
$\_uDeathRate$ **is-a** Sum
$\_uBasicDeathRate \in \_uDeathRate$

A few words on the representation of arithmetic expressions. Members of sums and products are represented simply as set members. Here $\_uBasicDeathRate$

will be one member of $\_uDeathRate$, and other members will be introduces from other models (preying, see below). However, a caution is needed with this approach because if a relationship is stated more than once, this may result in repetition of the same member in the sum or product. This can be partly prevented by defining the same members as named properties, e.g., by adding to the population model statements like the following.

$Member1(vDeathRate) = vBasicDeathRate$
$Member1 \in \#copied(vBasicDeathRate)$

This will be instantiated for $Unicorns$ producing the relationship

$Member1(\_uDeathRate) = \_uBasicDeathRate$

Any other repetition of derivation, producing a new variable like $\_uBasicDeathRate$ as a member of the sum, will thus result in an equality between $\_uBasicDeathRate$ and the new variable. Additional property names like $Member1$ will be omitted for simplicity.

Continue the population model with the construction of the differential equation for the values. The left member of the equation for a value is its derivative with respect to time. Thus we only have to build the right member using sums, products and a special value for -1, represented here as $MinusOne$.

$RightMember(vNumber) = vRightMember$
$RightMember \in \#copied(vNumber)$
$Exptype \in \#ascribed(vRightMember)$
$Exptype \in \#invariant(vRightMember)$
$Exptype(vRightMember) = Product$
$vNumber \in vRightMember$
$vBirthMinusDeath \in vRightMember$
$Exptype \in \#ascribed(vBirthMinusDeath)$
$Exptype \in \#invariant(vBirthMinusDeath)$
$Exptype(vBirthMinusDeath) = Sum$
$vFertility \in vBirthMinusDeath$
$vMinusDeath \in vBirthMinusDeath$
$Exptype \in \#ascribed(vMinusDeath)$
$Exptype \in \#invariant(vMinusDeath)$
$Exptype(vMinusDeath) = Product$
$MinusOne \in vMinusDeath$
$vDeathRate \in vMinusDeath$

The simplified rule (2) of 7.1 will result in unnecessary copying of $MinusOne$. This can be eliminated with a simple trick, but instead we will consistently treat $MinusOne$ as part of this frame (as well as of some others) and take care of its printing properties (including print representation as (-1), not shown here).

$Exptype \in \#ascribed(MinusOne)$
$Exptype \in \#invariant(MinusOne)$
$Exptype(MinusOne) = PlainValue$

Let griffins prey on unicorns. We define a preying relationship $PrGU$ which will be an instance of a special model Pr, by which both the fertility and the death rate of the predator depend on the availability of prey, and the basic death rate of the prey is further increased by the predator.

$PrGU$ **is-a** Pr
$Predator(PrGU) = Griffins$
$Prey(PrGU) = Unicorns$

The Pr frame might look like this.

$\#type(Pr) = \#frame$
$Predator(Pr) = vPredator$
$Predator \in \#copied(Pr)$
$Prey(Pr) = vPrey$
$Prey \in \#copied(Pr)$

In principle, this frame must prescribe that both $vPredator$ and $vPrey$ (or, more exactly, the repective instances to which they will be mapped) must be also instances of $Population$. This is redundant in this particular case because both $Griffins$ and $Unicorns$ already have this "superclass", but for the general case this can can be specified as follows.

$Superclass \in \#ascribed(vPredator)$
$Superclass \in \#invariant(vPredator)$
$Superclass(vPredator) = Population$
$Superclass \in \#ascribed(vPrey)$
$Superclass \in \#invariant(vPrey)$
$Superclass(vPrey) = Population$

Now introduce an additional member in the prey's death rate (defined as a sum).

$vPreyDeath \in DeathRate(vPrey)$
$DeathRate \in \#copied(vPrey)$
$Exptype \in \#ascribed(vPreyDeath)$
$Exptype \in \#invariant(vPreyDeath)$
$Exptype(vPreyDeath) = Product$
$Number(vPredator) \in vPreyDeath$
$Number \in \#copied(vPredator)$
$vPreyFactor \in vPreyDeath$

So we assume the additional death rate to be proportional to the number of the predator. We need now to link $vPreyDeath$ and $vPreyFactor$ to the model to make sure the values can be accessed by other relationships and to be used in mapping.

$PreyFactor(Pr) = vPreyFactor$
$PreyFactor \in \#copied(Pr)$
$PreyDeath(Pr) = vPreyDeath$

$PreyDeath \in \#copied(Pr)$

To secure copying, $vPreyDeath$ could be also linked to $vPrey$ instead of Pr; but this would lead to a confusion if the same population suffered from two different predators. On the other hand, $vPreyFactor$ being a member of $vPreyDeath$ set, it would be copied without the link, but the link will be needed to provide a numerical value.

Now we define fertility and death rate of the predator as depending on the number of prey.

$Fertility(vPredator) = vPredFertility$
$Fertility \in \#copied(vPredator)$
$Exptype \in \#ascribed(vPredFertility)$
$Exptype \in \#invariant(vPredFertility)$
$Exptype(vPredFertility) = Product$
$Number(vPrey) \in vPredFertility$
$Number \in \#copied(vPrey)$
$vFertilityFactor \in vPredFertility$
$FertilityFactor(Pr) = vFertilityFactor$
$FertilityFactor \in \#copied(Pr)$

$vDeathDecrease \in DeathRate(vPredator)$
$DeathRate \in \#copied(vPredator)$
$Exptype \in \#ascribed(vDeathDecrease)$
$Exptype \in \#invariant(vDeathDecrease)$
$Exptype(vDeathDecrease) = Product$
$Number(vPrey) \in vDeathDecrease$
$vDeathDecreaseFactor \in vDeathDecrease$
$MinusOne \in vDeathDecrease$
$DeathDecrease(Pr) = vDeathDecrease$
$DeathDecrease \in \#copied(Pr)$
$DeathDecreaseFactor(Pr) = vDeathDecreaseFactor$
$DeathDecreaseFactor \in \#copied(Pr)$

The next thing to do is to provide values for some parameters and to print out the differential equations (which can be appropriately defined as special entities attached to the quantities. We'll only discuss printing of expressions in the right members of the differential equations, because printing is an example of an action other than just inference, and we will use steps not defined in the preceding exposition.

Some expressions have been described as sums or products; others have to be described as plain values. Each of them will have a "method" for printing. A plain value will have its print representation, a constant (number or symbolic name).

$\#type(PlainValue) = \#frame$
$Representation \in \#expected(PlainValue)$
$PrintingMethod \in \#invariant(PlainValue)$

$PrintingMethod(PlainValue) = PrintPlain$

$PrintPlain$ is a procedure that will be applied to an argument, which will be an instance of $PlainValue$; the procedure must find and print the $Representation$ of its argument.

For sums and products, other printing procedures will be defined. We consider only sums because the other case is absolutely analogous.

$\#type(Sum) = \#frame$
$PrintingMethod \in \#invariant(Sum)$
$PrintingMethod(Sum) = PrintSum$

$PrintSum$ will print the addends in succession, but it has also to keep track of whether any addends were printed before the current addend. We'll assume this is done by providing a special "object" (in the standard sense of OOP) called "sum separator", with a constructor that sets it to the state of "nothing printed", a "separate" method which makes it print a left parenthesis at the first call and a plus sign at subsequent calls, changing the state to "something printed", and a "finalize" method that prints either a right parenthesis, or zero (if nothing was printed).

Thus $PrintSum$ will be a conjunction (7.2) of three smaller procedures:

- build a sum separator,
- print addends,
- finalize the separator.

The second step will, in its turn, be a "frame-all" procedure, ascribing to all addends in turn an internal procedure defined as a conjunction of two steps: a call to the "separate" method of the separator and printing of the addend. We consider the latter in more detail because at this moment we don't know what kind of expression is the addend. The procedure $PrintExpression$ will be defined as follows.

$\#type(PrintExpression) = \#frame$
$PrintingMethod \in \#expected(PrintExpression)$
$PrintingMethod \in \#ascribed(PrintExpression)$

Let us see how $PrintExpression$ applies to an expression X. At the mapping stage we "expect" $PrintingMethod(X)$, and if it was not previously defined as a constant, we get a contradiction. Otherwise we get the specific printing method defined for this X, which might be $PrintPlain$, $PrintSum$, $PrintProduct$, or something else. Then the method is "ascribed" to the expression, and an is-a formula is generated, e.g.,

$$X \textbf{ is-a } PrintPlain,$$

which will result in X being printed out.

Now to obtain the system of differential equations we still need to describe some parameters as plain values and to provide print representations. Then we get something like this (an actual computer printout, except for the layout):

```
   d(uNumber)/dt=(uNumber*(uFertility+((-1)*(uBasicDeathRate+
(gNumber*pPreyFactor)))));
   d(gNumber)/dt=(gNumber*((uNumber*pFertilityFactor)+
((-1)*(gBasicDeathRate+(uNumber*pDeathDecreaseFactor*(-1))))));
```

Let us summarize what has been done in this example. We use two sources of knowledge. One is information about specific populations and their relationships, it is just a few lines (including the information about numeric values or symbolic names of some parameters, not given explicitly in this example). The other source is the supply of models. It should be probably expanded by introducing models for other types of propagation, dying, preying, migration, etc. – we don't impose any particular pattern of numerical relationships. But different models for different populations will be able to interact properly in a single system producing a set of simulating equations, provided that we use the same names (like *Number* or *DeathRate*) for the concepts shared by different models. Few programming tools provide this flexibility, except possibly for latest object-oriented systems.

References

1. Цейтин, Г.С.: Программирование на ассоциативных сетях. In: ЭВМ в проектировании и производстве, вып. 2, под ред. Г.В.Орловского, 16-48. "Машиностроение", Ленингр. отделение, Ленинград, 1985 (G.S.Tseytin. Programming with association nets. English version at http://gamma.niimm.spb.su/~tseytin/netslong.html)
2. Железняков, М.М., Невлева, Т.Н., Новицкая, И.М., Смирнова, Л.Н., Цейтин, Г.С.: Опыт построения модели "текст ⇒ действительность" с использованием ассоциативных сетей. In: Машинный фонд русского языка: предпроектные исследования, 140-167. Институт русского языка, Москва, 1988 (M.M.Zheleznyakov, T.N.Nevleva, I.M.Novitskaya, L.N.Smirnova, G.S.Tseytin. An attempt of building a "text ⇒ reality" model using association nets. English version at http://gamma.niimm.spb.su/~tseytin/article.html)
3. Tseytin, G.S.: On some mechanisms of representation of meaning in natural languages. The Prague Bulletin of Mathematical Linguistics, **65-66** (1996) 5-12 (See also at http://gamma.niimm.spb.su/~tseytin/mytalke.html)
4. Tseytin, G.S.: From Logicism to Proceduralism (an autobiographical account). In Algorithms in Modern Mathematics and Computer Science, Lecture Notes in Computer Science, **122** (1981) 390-396