EXPLORATIONS ON "NATURALISTIC FORMALISMS"

Heikki Hyötyniemi

Helsinki University of Technology, Control Engineering Laboratory
Konemiehentie 2, FIN-02150 Espoo, Finland

It seems that knowledge representation formalisms are getting more and more sophisticated. In this paper, however, the opposite extreme is studied: the foundations of the formalisms are based directly on the data. It turns out that this kind of "naturalistic" formalisms may give interesting insight and new possibilities to program analysis and validation. Two formalisms are presented: the expressive power of the first one, having no data structures, is based solely on computation, whereas the other one is based on representation, introducing no procedural machinery whatsoever. By appropriately selecting the basic constructs, nontrivial models can still be realized in both cases. The simplicity of the naturalistic grounding is utilized for showing that the latter formalism can be implemented using the former one; this means that the computational power of pure representations need not be limited.

1. INTRODUCTION

The representation formalisms are getting more and more sophisticated. This means that the compilers/interpreters become more and more complicated; there is more and more data processing happening out of sight. In the AI field — even if the new tools are very welcome — this development has some not so nice consequences: the gap between the conceptual level and the data level is getting wider and wider.

There might be some advantage if the formalism were *transparent* so that the role of the interpreter would become trivial. This paper presents two formalisms that can be used for representing classification problems; the role of these formalisms is to offer means for describing what the classes look like. Both of them have their founding very directly on the data; it can be claimed that the semantics of the formalisms is bound to the data in a "naturalistic" way. These formalisms are not supposed to be applied in real actual programming or modeling applications, but it may be that there is some intuition and understanding to be gained. The transparency of the formalisms, for example, makes it possible to see some close connection between them — this relationship facilitates the analysis of the different approaches.

These two formalisms differ very much from each other — actually, they are from the opposite extremes of the continuum: the first one is based exclusively on *computation,* there are no data structures whatsoever, whereas the other formalism is based exclusively on *representation,* so that no computational primitives are defined. It turns out that despite these shortcomings, the expressive power of both of these formalisms is rather high. This results from the fact that when the data itself has the main role, system theory and mathematics take control. In both cases, the formalisms can directly be interpreted in terms of high-dimensional real-valued vectors having very simple topology.

2. FOCUS ON *COMPUTATION*

First it is assumed that everything that is interesting emerges from *dynamic evolution only.*

It turns out that there are very fruitful connections between theoretical computer science and dynamic systems analysis. To see this, some computability theory is needed (for example, see [2]). According to the basic axiom of computability theory, all computable functions can be implemented using a *Turing machine.* There are various ways to realize a Turing machine; it can be shown that if a function is Turing computable, it can be coded using the following simple language $\mathcal{L}$ that is here presented in the Backus-Naur (BNF) form:

`Program`	::=	`Variables`* `Commands`*	
`Variables`	::=	*LineNumber VarName* = *Constant*	% Initial value
`Commands`	::=	*LineNumber* `CondBranch`	
	::=	*LineNumber* `Operations`	
`CondBranch`	::=	IF `Condition`*	
		THEN `Operations`	
		ELSE `Operations`	
`Condition`	::=	*VarName* > 0	
`Operations`	::=	`Modifications`* `Jump`	
`Modifications`	::=	*VarName* ADD *Constant*	% Increment
	::=	*VarName* SUB *Constant*	% Decrement
`Jump`	::=	GOTO *LineNumber*	

Above, *Constant* $\in \mathcal{N}$ can be any positive integer; *VarName* is a character string. All variables are assumed to have non-negative integer values; if an operation would make the variable value negative, zero value is used instead. To make the "compiled", matrix-form programs (see later) match the program formulation in one-to-one fashion, all lines in the code must have a distinct *LineNumber;* line numbers must be successive integers starting from 1, and one of the program lines has to be the entry point into the program. Note that a conditional branch always exhausts *three lines,* so that the line number counter is incremented by three (the reason for this peculiarity is due to the special coding of the compiled program). The Kleene stars "*" mean that there can be an arbitrary number of corresponding constructs.

2.1 The "Turing system"

It can be shown that a special dynamic system structure is in one-to-one correspondence with the above language $\mathcal{L}$. This is an autonomous, discrete-time nonlinear system (see, for example, [1]) of the form

$$s(k+1) = f(As(k)). \tag{1}$$

Here A is a real-valued square matrix compatible with s, and the "generalized cut" function $f : \mathcal{R}^{\dim(s)} \to \mathcal{R}^{\dim(s)}$ is now defined elementwise as

$$f_i(s) = \begin{cases} s_i, & \text{if } s_i > 0, \text{ and} \\ 0, & \text{otherwise,} \end{cases} \tag{2}$$

for all $i = 1, \ldots, \dim(s)$. The state transition matrix A in (1) can be defined so that any program in language $\mathcal{L}$ can be presented as a discrete-time process. The dimension of A

is dependent of the program complexity, so that there is a separate entry in the "snapshot vector" s corresponding to all program lines. The original state vector $s(0)$ consists of the initial values of the variables. Additionally, if the entry point of the program is on line i, then $s_i(0) = 1$. All other elements of $s(0)$ are zeros.

In [3] it is shown that the internal state of the dynamic system can be identified with the program states, and the succession of the states corresponds to the program flow. Running a program now means that the process (1) is iterated until the state no more changes, or until a fixed state is found — the final variable values, or the calculation results, can be seen in the resulting state vector.

The language $\mathcal{L}$ is a rather straightforward extension (and simplification[1]) of the formalism that is presented in [3], and the construction of the matrix A is now skipped. However, an example is presented.

2.3 Program example

Assume that the *parity function* is to be realized, so that $y(x) = 0$ if $x \in \mathcal{N}$ is even, and $y(x) = 1$ if x is odd. Using language $\mathcal{L}$ this can be coded as

```
1   VAR X = x
2   VAR Y = 0
3   IF X > 0      % Entry point
    THEN X SUB 1 Y ADD 1 GOTO 6
    ELSE GOTO END
6   IF X > 0
    THEN X SUB 1 Y SUB 1 GOTO 3
    ELSE GOTO END
```

Above, the mnemonic "END" has been used to denote jump "outside" the program, that is, the program counter vanishes altogether. In a matrix form this algorithm can be implemented as shown below. Here the '+' signs stand for number 1, '−' signs stand for −1, and all other elements are zeros. The initial snapshot, or the state $s(0)$ before iteration, is also shown. After the iteration (1) has converged to a fixed point $s(\infty)$, the final result y is obtained as the second element of the state vector (because Y is declared on the second program line). It turns out that regardless of the value $x \in \mathcal{N}$, this system remains always stable; the time it takes to converge to a fixed state is linearly dependent of the value of x.

$$A_{\text{parity}} = \begin{pmatrix} + & & & - & + & & - & + \\ & + & & + & - & & - & + \\ & & & & & & + & - \\ & & + & & & & & \\ - & & + & & & & & \\ & & & + & - & & & \\ & & & & & + & & \\ - & & & & & + & & \end{pmatrix} \quad \text{with} \quad s(0) = \begin{pmatrix} x \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

For example, if $x = 3$, the following state sequence results (note that the second row starts with the state that corresponds to the problem "What is y when $x = 1$?"; the more

[1] For example, the 19-dimensional "multiplier system" in [3] can now be implemented using only a 13-dimensional system

complex problem with $x = 3$ has also been reduced into a simpler form having the same answer). The process seems to freeze in a state where $y = s_2 = 1$, so that "3 is odd".

$$\begin{pmatrix} 3 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 3 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 2 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 2 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \cdots$$

2.4 Analysis

As a "programming language", the matrix formulation is an interesting extension of the language $\mathcal{L}$ — on the other hand, it is powerful (extensions towards parallel processing are simple, etc.), but at the same time it is extremely simple: only one basic construct is available, spanning a continuum between pure variables ("eigenvalues" in 1) to program statements with relayed "program counter" (eigenvalues in 0). It can be argued that the process formulation (1) is *the simplest way to realize the Turing machine.*

One nice feature about the presented language is that it is *continuous* in all parameters; normally languages, natural or formal, are based on symbols that are mutually incommeasurable, while now all constructs are structureless and their mutual distances can be calculated using some metric defined in the vector space. Whereas the operation of normal programming languages is "crisp", that is, minor changes in the program code may collapse the whole system, now any of the parameters can be modified with no sudden catastrophic effects. It is easy to show that if any of the state elements is disturbed by an amount ϵ, the variables of the next state vector will differ at most by an amount of $\pm\epsilon$ from the nominal values. Analogous continuity holds for changes in the elements of A. Additionally, it is easy to prove that qualitatively the behavior of the system is not changed if all variable values and program counters are scaled by the same (positive) factor: if $\sigma(k) = \alpha \cdot s(k)$, there holds $\sigma(k+1) = f(A\sigma(k)) = f(A \cdot \alpha s(k)) = \alpha \cdot f(As(k)) = \alpha \cdot s(k+1)$ (see Fig. 1).

3. FOCUS ON *REPRESENTATION*

Next we take the opposite view: assume that everything that is interesting is *hidden in the static data structure.*

3.1 About formalisms

New representation formalisms try to capture the fundamental structure of the world around us; the better the formalism matches the existing entities, the shorter codes are

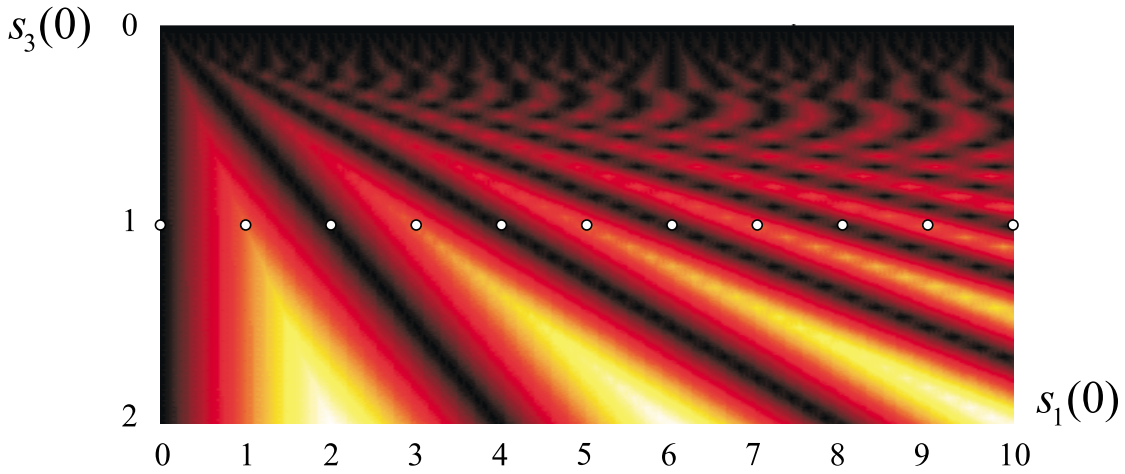

Figure 1: "Generalized parity function": the output y (entry $s_2(\infty)$) of $s(k+1) = f(A_{\texttt{parity}} \cdot s(k))$ has been plotted as the initial values $s_1(0)$ and $s_3(0)$, or the "input" x and the "program counter", respectively, are continuousaly varied (lighter color denoting higher value; black stands for zero level). Note that only in discrete points (dotted points in the figure) the parity value is defined in the traditional sense, the value being either "1" or "0"

additionally needed. This way the evolution of the modern programming languages has started from no structure whatsoever (`Basic`, etc.), gone through the era of procedural programming (`Pascal`, `C`, etc.), reaching the current period of object-oriented programming (`C++`, `Java`, etc.). The object-oriented formalisms nicely implement the Aristotelian view of the categories: there are strict hierarchies between classes, and the class boundaries are strict.

However, as has been motivated by the fuzzy theorists, this "all or nothing" approach is becoming obsolete. But what do they have to offer to substitute the old tools? Despite the enthusiasm, the fuzzy formalisms seem to become extremely involved in more complex applications.

Next, a formalism is presented that implements a special ontological view. Even though this view is rather simple, some rather complicated problems can be formulated in this framework. It turns out that many nontrivial modeling problems can be implemented with *no additional code at all.* The emphasis is on a specialized representations, but the structural framework is again extremely simple, based on linear algebra.

3.2 Representation formalism

Assume that there are classes that are being defined in terms of other classes, constituting a semantically more or less integral entity:

$$\texttt{class}_i \;\rightarrow\; \sum_{j\neq i}^{N_a} a_{ji} \cdot \texttt{class}_j \quad + \quad \sum_{j}^{N_b} b_{ji} \cdot \texttt{input}_j$$

This means that if an entity belongs to `class`i, it also belongs to class `class`j, but *only with limited membership,* the weight being defined by a_{ji}. The "inputs" are lower level categories that define the naturalistic grounding for the classes. The parameters N_a and N_b stand for the number of categories and inputs in the model, respectively. As an example, study the following simple "world model":

$$
\begin{array}{lcl}
\texttt{animal} & \rightarrow & 0.1{\cdot}\texttt{bird} + 0.1{\cdot}\texttt{dog} \\
\texttt{bird} & \rightarrow & 1{\cdot}\texttt{animal} + 0.1{\cdot}\texttt{canary} + 1{\cdot}\texttt{flying} \\
\texttt{canary} & \rightarrow & 1{\cdot}\texttt{bird} + 1{\cdot}\texttt{yellow} \\
\texttt{dog} & \rightarrow & 1{\cdot}\texttt{animal} + 0.8{\cdot}\texttt{brown} - 1{\cdot}\texttt{flying}
\end{array}
$$

There are now four interconnected categories (`animal`, `bird`, `canary`, and `dog`) and three independent inputs (`flying`, `brown`, and `yellow`) in this extremely simple world model. The declarations say, for example, that the concept of "animal" contributes when a "dog" is being defined; not so much, though — compare to the definition of "dog", where the weight of "animal" equals 1, meaning that actually dogs are all animals (see Fig. 2). For a closer analysis on consistency issues, etc., see [5]. The concepts define each other introducing some kind of contextual semantics. There are no separate subclasses or superclasses; what is more, there does not exist any distinction between classes and objects as their instantiations. An individual (normally defined as an object in object-oriented formalisms) just is strictly inside its superclass and has minimal effect on it. What is important is that categories and their attributes have identical representation: concepts within a "small world" may add flavor to each other. The classes need not be nouns, and attributes need not be adjectives.

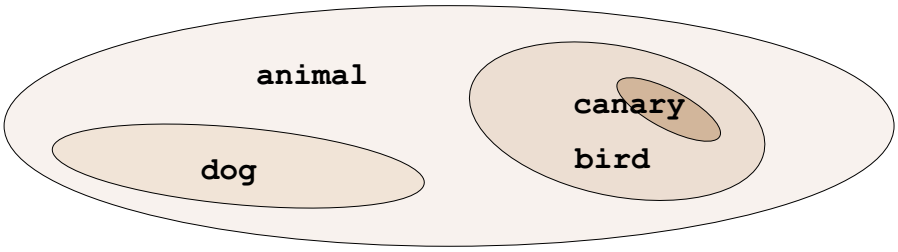

Figure 2: Visualization of the set hierarchy in the world model. Dogs are members of the animal class (membership 1.0), whereas animals belong to dogs with fuzzy membership 0.1. Note that "negative memberships" can also be declared in case of mutually exclusive categories

The above definitions can be written in a compact form as

$$
\begin{pmatrix} x_{\texttt{animal}} \\ x_{\texttt{bird}} \\ x_{\texttt{canary}} \\ x_{\texttt{dog}} \end{pmatrix} = \begin{pmatrix} - & 1 & & 1 \\ 0.1 & - & 1 & \\ & 0.1 & - & \\ 0.1 & & & - \end{pmatrix} \cdot \begin{pmatrix} x_{\texttt{animal}} \\ x_{\texttt{bird}} \\ x_{\texttt{canary}} \\ x_{\texttt{dog}} \end{pmatrix} + \begin{pmatrix} 1 & & \\ & & 1 \\ -1 & 1 & \end{pmatrix} \cdot \begin{pmatrix} u_{\texttt{flying}} \\ u_{\texttt{brown}} \\ u_{\texttt{yellow}} \end{pmatrix}.
$$

This is an implicit declaration, and can be written as

$$x = A \cdot x + B \cdot u,$$

where, in this world model, one has

$$
A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0.1 & 0 & 1 & 0 \\ 0 & 0.1 & 0 & 0 \\ 0.1 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ -1 & 1 & 0 \end{pmatrix}.
$$

Given the inputs, the categories can be solved as

$$x = F \cdot u = (I_{N_a} - A)^{-1} B \cdot u.$$

Vector x consists now of the *a posteriori* "probabilities" of the classes; however, note that the elements of x do not necessarily remain between 0 and 1. Not all definitions result in reasonable classifications; for example, the probability distribution between the classes is highly dependent of the numerical values in the matrices. A simple method of automatically compensating for the probabilities is to divide the obtained values x_i by the *a priori* probabilities v_i that can be solved as an eigenproblem $Av = \lambda v$ (see [5]):

$$x_i \leftarrow x_i / v_i.$$

In the above world model, the eigenvector becomes (without normalization)

$$v = \begin{pmatrix} 0.94 & 0.30 & 0.06 & 0.18 \end{pmatrix}^T.$$

This means that without probability compensation the class "animal" would be almost always the most prominent.

3.3 Ontological assumption

What can be modeled using the above syntax is of course rather limited; however, it can be claimed that our *recognition machinery is based on the same data structure,* making these formulations much more universal. As discussed in [4], the "natural" complex data has often a rather peculiar distribution: the observations occur in chunks, data clusters, and within these clusters the data is distributed in a subspace spanned by a set of non-orthogonal basis vectors. This approach has been applied in many applications; see [4]. Whereas the categories in the applications have been automatically constructed, based on the statistical correlations between data elements, the models can also be explicitly defined, using the above formulations, and thereafter be compiled into numerical form; this way the set of categories can be controlled manually. The formal approach can be interpreted as defining "virtual data" by explicitly determining the (non-orthogonal) main axes of the data distributions. Using philosophical terminology, it could be said that this virtual data is only *potential,* not *actual.* The axis prototypes can be solved from the matrix formulation by applying linear algebra.

To find the outlook of the virtual data distribution, the mapping F should be inverted somehow. Assume that the actual model between u and x contains some error e, so that actually there holds $x = Fu + e$. Let us define the criterion to be minimized:

$$e^T W e + u^T U u,$$

that is, the errors in different elements of x can be weighted by using the (diagonal, non-negative) matrix W; additionally, the size of the data vectors can be limited using the matrix U. A straightforward minimization results in the estimate for u when x is given:

$$\hat{u} = \left(U + F^T W F\right)^{-1} F^T W \cdot x.$$

Using this expression, the category prototype vectors can be solved: for example, when solving for $\bar{u}_{\texttt{bird}}$, the matrices can be selected as

$$x = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \qquad W = \begin{pmatrix} 0 & & & \\ & 1 & & \\ & & 0 & \\ & & & 0 \end{pmatrix} \qquad \text{and} \qquad U = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix},$$

the non-diagonal entries in the matrices being zeros. When using this kind of weightings, the prototype vectors become

$$\left(\bar{u}_{\texttt{animal}} \,\middle|\, \bar{u}_{\texttt{bird}} \,\middle|\, \bar{u}_{\texttt{canary}} \,\middle|\, \bar{u}_{\texttt{dog}} \right) = \left(\begin{array}{c|c|c|c} 0.03 & 0.29 & 0.05 & -0.30 \\ 0.27 & 0.04 & 0.01 & 0.35 \\ 0.31 & 0.33 & 0.49 & 0.04 \end{array} \right).$$

This tells us that, for example, "bird" assumedly has high weight on the property of "flying", and — because the canary is the only exemplar of a bird — also on "yellow". The above world model can be visualized schematically as shown in Fig. 3, where it is shown how concepts, having numerical rather than symbolic content, can be used as attributes for other concepts.

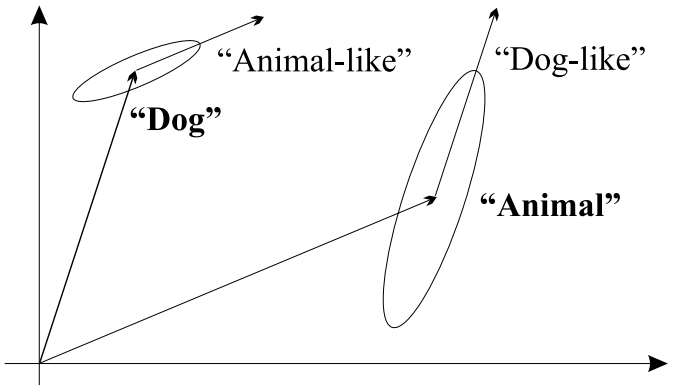

Figure 3: Visualization of the interchangeable roles of categories and attributes. The class having the most emphasis is regarded as being the category center

4. FOCUS ON *COMBINATION*

The above totally linear starting point is good for analysis, but there are well-known limitations plaguing linear approaches. The perhaps simplest way to eliminate the restrictions is to restrain to *non-negative* membership values, that is, there holds $x \geq 0$. Unfortunately this nonlinearity ruins the above explicit formulations, and the fixed state of x has to be iterated:

$$x(k+1) = f\left(A \cdot x(k) + B \cdot u\right),$$

where the function $f(\cdot)$ cuts the non-positive vector elements; the iteration hopefully converges to some fixed state $x(\infty)$. In Fig. 4, the final results are shown as the iteration is carried out for the above example system.

Comparing the above expression to (1), one can see the essential similarity: in both cases the same kind of iteration is performed, and the nonlinearity has the same form. There is one difference, though: now there is an exogenous input rather than non-zero initial state that determines the faith of the iteration process. However, it can be shown that the computational power of the new formulation is (at least) as high as it was in the previous case; this comes with the expence of increased dimensionality.

When starting from a purely static model it may seem strange that any computation can also be realized in that framework. The key is in the iterative implementation of the mechanism. On the other hand, "hidden" categories are needed that have no role as

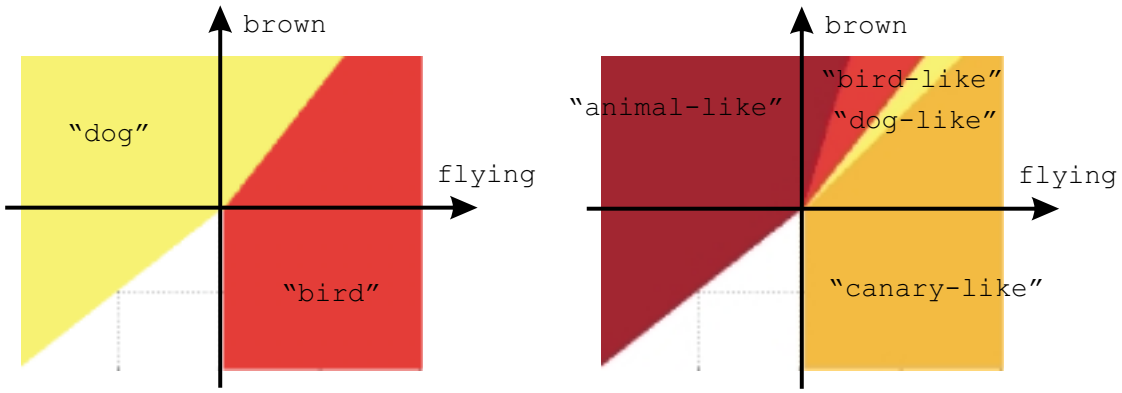

Figure 4: Categorization of the inputs: the two entries u_1 and u_2 (features "flying" and "brown", respectively) are varied between -1 and 1, and the resulting main category is shown on the left; on the right, the corresponding main attributes, or the next most significant categories are shown

explicit categories but are necessary to implement the computation steps. The qualitative enhancements in representational power are essentially based on the high dimension of the state space: remember "Flatlands", where the two-dimensional creatures living in the two-dimensional world experience wonderful phenomena when tree-dimensional objects hit their world — similarly, the computational capacity of high-dimensional systems amazes us who are slaves of our three-dimensional world.

The above construction shows that at least *some* formalisms based on no computation but representation only can have unlimited computational capacity *if implemented appropriately.* Perhaps the quest for Turing computability has been exaggerated in the cognitive science community — if it turns out that even the simplest (static) approaches can fulfill the computational requirements, why not concentrate on more relevant issues.

REFERENCES

1. Åström, K.J. and Wittenmark, B.: *Computer-Controlled Systems—Theory and Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1997 (3rd edition).

2. Davis, M. and Weyuker, E.: *Computability, Complexity, and Languages — Fundamentals of Theoretical Computer Science.* Academic Press, New York, 1983.

3. Hyötyniemi, H.: On Unsolvability of Nonlinear System Stability. In *Proceedings of the European Control Conference (ECC'97),* Brussels, Belgium, July 1-5, 1997 (CD-ROM format).

4. Hyötyniemi, H.: On Mental Images and 'Computational Semantics'. In *Proceedings of the 8th Finnish Artificial Intelligence Conference STeP'98* (eds. Koikkalainen, P. and Puuronen, S.), Finnish Artificial Intelligence Society, Helsinki, Finland, 1998, pp. 199–208.

5. Hyötyniemi, H.: From Knowledge to "Virtual Data". *Arpakannus* 1/1999, Finnish Artificial Intelligence Society, pp. 31–34.