MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 159 November 1977

# HIERARCHY IN KNOWLEDGE REPRESENTATIONS

by

Jon Doyle*

**Abstract:**

This paper discusses a number of problems faced in communicating expertise and common sense to a computer, and the approaches taken by several current knowledge representation languages towards solving these problems. The main topic discussed is hierarchy. The importance of hierarchy is almost universally recognized. Hierarchy forms the backbone of many existing representation languages. We discuss several technical problems raised in constructing hierarchical and almost hierarchical systems as criteria and open problems.

* Fannie and John Hertz Foundation Fellow

This research was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0643.

## INTRODUCTION

This paper discusses a number of problems faced in communicating expertise and common sense to a computer, and the approaches taken by several current knowledge representation languages towards solving these problems. The main topic discussed is hierarchy. The importance of hierarchy is almost universally recognized. Hierarchy forms the backbone of many existing representation languages. We discuss several technical problems raised in constructing hierarchical and almost hierarchical systems as criteria and open problems.

The issues raised in the study of hierarchy are used in examining the abilities and inabilities of several current languages for knowledge representation. The systems discussed include FRL [Goldstein and Roberts 1977, Roberts and Goldstein 1977a, 1977b], OWL [Szolovits, Hawkinson and Martin 1977], KRL [Bobrow and Winograd 1977a, 1977b], and NETL [Fahlman 1977]. Each of these systems has been designed with differing goals, styles and ambitions in mind. The goals of FRL are modest. FRL not a programming language or a set of strong commitments, but rather a set of conventions for using a data structure generalizing the traditionally useful concept of property lists. OWL is strongly directed by a desire to remain close to English in expression, and is evolving as an expert problem solving and language understanding system. KRL is an ambitious project with few concrete commitments and a desire for extreme flexibility. NETL attempts to be relatively complete in its ability to express several forms of knowledge, and is structured to allow a radically parallel implementation.

## HIERARCHY

Hierarchy is an important concept. It allows economy of description, economy of storage and manipulation of descriptions, economy of recognition, efficient planning strategies, and modularity in design. It has had an important place in many theories of engineering, problem solving, and computer science. It has also been important in the design of many knowledge representation languages. There are, however, numerous problems associated with the representation and meaning of hierarchy, particularly in its interaction with problems of equality. In this section we discuss the problem of the individual/prototype distinction, two types of multiple descriptions, and almost hierarchical descriptions. In Appendix 1 we discuss and efficient representation of these forms of descriptions.

### Individuals

The distinction between individual and prototype is a slippery problem. This section will not present any solutions or definite conclusions, but is intended to indicate that problems do exist and should be considered carefully by representation theorists and language designers.

In many representational systems, individuals are considered to be instances of prototypes, but cannot themselves be prototypes. There are cases, however, when this rigid separation causes problems. What is an individual and what is a prototype may change depending on what world the object is viewed from. An example is that within a design of a radio, resistor R-23 is an individual, distinct from all other resistors in the circuit. In the real world, R-23 is a prototype for a resistor - each individual radio built will have its own instance of R-23, and each of these instances will be distinct from other resistors.

For another example, consider a description of a person named Fred to be a description of an individual in reference to other people. We may also wish to regard that description as a prototype when considering Fred in different hypothetical situations, or on different

days, in which the more detailed descriptions of Fred are considered to be individuals in the set of descriptions of possible Freds. One might also use an individual as a prototype in other ways. Consider a description like "He's a real Fred." or "Half my students are Napoleons.".

As an extreme example, one can imagine that there exist infinite chains of individual/prototypes derived from an infinite process of further specification. [Moore and Newell 1973] Consider the pedigree expert, to whom a change in ancestry means a change in nature. Since each person has parents, and they in turn are persons and so have parents, the person with no ancestry serves as a prototype for the person with known parents, who serves as a prototype for the person with known parents and grandparents, and so on. At each level, there is a prototype describing the person with a known set of ancestors, which serves as a template for describing the individuals that would result from differing possibilities for the next layer of ancestors. In more abstract terms, one can always turn an individual into a prototype by adjoining a set of properties, and splitting the individual into the set of individuals such that each of these is distinguished by one of the set of properties.

The reason for distinguishing individuals is so that questions of equality can be reasoned about. Presumably only individuals can be decided to be equal or unequal. This means that questions of individuality might be tied to what questions of equality are being asked. For example, one common type of entity used in reasoning is that of the anonymous or unidentified individual. Such an entity is known to be an individual, but may be determined to be the same individual as some other individual. Suppose we wish to reason about some unidentified bird. Our prototype for a bird might be a duck. In this case, our anonymous bird would be thought of as a duck until proven otherwise. This adds an additional complication to the distinction between prototypes and individuals. In general, if the individuality of a description depends on some factor, there must be some way of describing the determiner and the dependence. Some studies aimed at elucidating these distinctions and problems would be very valuable.

Multiple Descriptions

The use of multiple descriptions of an object has been a popular idea. There have been two forms of multiple descriptions studied. The first is that used in the representation languages, where multiple descriptions are used as a method for factoring descriptions into several smaller, more generally useful sets of features. This allows more sharing of descriptions in the data base, and allows investigation of properties based on relevance criteria. For instance, if the topic involves the hair color of a person, a procedural system might choose to investigate only descriptions of physical appearances, rather than also searching through descriptions of the person as a professor or as a pianist.

A quite different sort of multiple description occurs when is described as a shared part of other objects, rather than as a common instance. In this form, the description is specified in terms of other descriptions, but the other descriptions share structure. In the simpler form of multiple descriptions, each inherited property of a description is usually derived from just one of its descriptions (or from a common ancestor of two of its descriptions if the property can be inherited from either one.) For example, Bobrow and Winograd [1977a] present a use of KRL in describing a particular event as both a "Visit" and a "Travel". In contrast, the other form of multiple description uses coincidences between structure inherited from distinct descriptions to derive new information and constraints. An example of this might be a node in a circuit which is both the input of one stage of the circuit and the output of another stage of the circuit. This imposes some important demands on the representation system, but allows an important problem solving advantage.

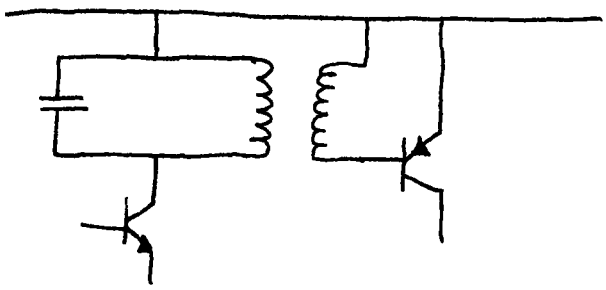
Multiple descriptions which share structure can be used to reduce the complexity of the tools needed to solve problems. In contrast to the simpler form of multiple descriptions, which by factoring descriptions may lead to reduction in the amount of simple inheritance computations needed (a change in quantity but not quality of the computation), multiple descriptions sharing structure can transform problems from ones requiring arbitrary search, perhaps even

problem solving in an uncomputable domain, into problems with straightforward or algorithmic solution. This is exemplified by the SLICES form of multiple description developed by Sussman [1977]. In his example, the use of multiple descriptions for a circuit allows the avoidance of the tool of full-fledged algebraic manipulation in a domain of multivariate rational functions. The multiple descriptions allow the problem to be solved instead with simple linear and numerical algebra, a domain in which straightforward algorithms exist. A crucial aspect of this effect is that the multiple descriptions interact. Information in one description produces constraints on other descriptions. This then derives new information by resolving the new constraint with the previously existing constraints. [Stallman and Sussman 1977] If all the descriptions are simple in structure themselves, this frequently allows solution of problems by simple tools.

The most important problem caused by this form of multiple description is that it leads to almost hierarchical descriptions, as described in the next section.

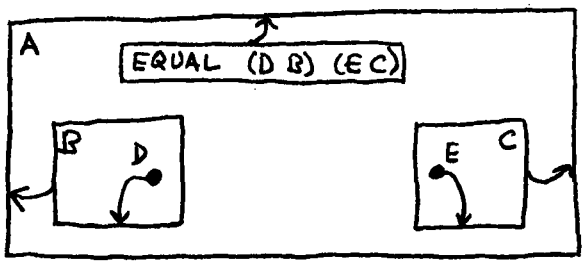
## Almost Hierarchical Descriptions

In many cases, a strictly hierarchical decomposition of the structure of an object cannot be made. Particularly when components of a decomposition must be connected in some sense, modules must share subcomponents, thus destroying the pure hierarchy. This means that what appears hierarchical at one level may not be at the next level down. Consider Sussman's favorite example, the following circuit diagram:



Here the hierarchical description of the circuit may mention one stage (containing the capacitor and left inductor) connected to another stage (containing the right inductor), but at the same time, the description may refer to the fact that the left and right inductors are actually two windings of a transformer. This does not exhaust the description, as there are many more views of these components in terms of DC bias paths, AC signal paths, and other parts of the circuit.

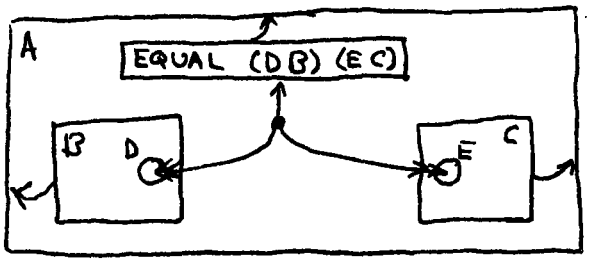
While such forms of descriptions can be precisely indicated (as in [Sussman 1977]), they cause problems in implementing hierarchical memory systems. For instance, one can try to use a purely hierarchical system to describe the almost hierarchical system by using EQUAL statements to denote equality, as in the following diagram:




The arrows denote the parents of each described object, that is, the reason for each object's existence. In this diagram, both the two sub-boxes and the EQUAL statement have the outer box as their parent, while the identified objects have their enclosing sub-box as their parents. This means of identification necessitates some form of pattern

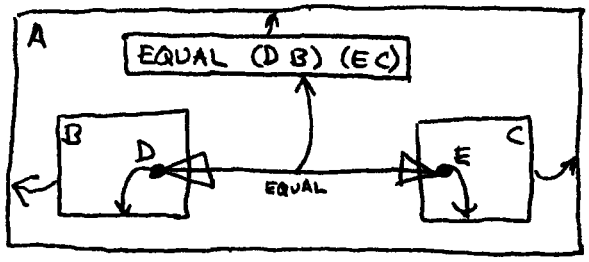
matching, for answering a question about an object requires that all known descriptions (of the form "the X of the Y of the ...") of the object be computed and compared against the set of relevant EQUAL statements. This process is extremely painful. This process is, as we shall see, essentially the only one possible in several representation systems.

The pattern matching can be avoided by the following method, but at the expense of losing the ability to assign meaningful reasons for the existence of objects. This approach involves creating the shared object at the level of the equivalence, and then passing this object down the hierarchy to the descriptions sharing the object. This approach is used in one of a series of hierarchical descriptive systems investigated by Sussman and Steele.




The use of this technique has several implications. First, the shared object has more than one parent in the hierarchy (the two sub-boxes), but these parents are not the reason for the object's existence - the reason is the EQUAL statement - and so the objects which share the object cannot be considered in isolation. Second, all connections between objects must be known at the outset, for one cannot identify two components of already existing structure. This makes the use of multiple descriptions extremely difficult, as one cannot then add newly discovered descriptions to existing ones.

Another method, used in Fahlman's NETL system and in another of the Sussman-Steele systems, is that of equivalence links. In this method, distinct objects are used, as in the first scheme above. These then are directly connected by a bi-directional EQUAL link.




This scheme allows each hierarchical module to be considered as an entity in itself, since each sub-box has its own representation of the equivalenced object. This also allows reasonable explanations to be constructed, since the justifications for information transmitted across the EQUAL link can add the link to the justification. This makes sense since the parent of the EQUAL link is the module at which the statement of equality is being made, as above.

A crucial requirement for implementing this method is that the representation of parts of objects be first class entities themselves. That is, such equivalences refer to the intensional objects of descriptions, and not to the denotations of these objects. If parts of descriptions are not themselves objects of description, then equivalences cannot even be stated. For example, one might wish to say that the SON of the SON of a PERSON is the same as the GRANDSON of a PERSON, with the effect that information can be stored and retrieved under either description. This is not possible in FRL, KRL and other systems. In these, one can only reference the denotation of a slot in a frame or unit, but one cannot refer to the slot itself. The only possibilities are either to use canonical representations (a method with well-known difficulties) or to use interface functions which keep pairs of equivalenced descriptions and handle the search for requests and insertions. This is the pattern matching solution described above.

Even when all substructures of entities are themselves entities for discussion, as in NETL and OWL, other problems must be considered. For instance, an object might be described in several distinct ways, but not all these descriptions can be held at one time. The Necker

cube is a drawing that has two distinct views. Contradictions arise if both descriptions are held simultaneously. Fahlman calls this problem the copy-confusion problem.

The copy-confusion problem takes several subtle forms. One form is the conflicting view example of the Necker cube. Another form is the recursive description problem, in which an object (like an amplifier) is described in terms of other objects of the same type. The problem here is that in separating the properties of one copy of a description from another (for instance, distinguishing between the gain of the amplifier and the gains of its stages). Fahlman [1977] presents algorithms for accessing multiple descriptions which take care to avoid these problems.

## Other Considerations

Once a system represents substructures as first-class entities in their own right, as is necessary for representing the almost hierarchical descriptions above, it becomes possible to be very economical in allocating storage for descriptions. That is, it becomes possible to represent knowledge in a way which uses no more storage than is logically necessary. Appendix 1 discusses such methods, in the context of the algorithms and techniques of NETL.

## LANGUAGES FOR KNOWLEDGE REPRESENTATION

One of the primary requirements for a knowledge representation system is that the language have a clean semantics. [Hayes 1974, 1977] Without a well-defined semantics for the language primitives, it becomes difficult to decide how to encode knowledge, and to decide what a given representation means. Thus one major concern of designers of knowledge representation languages should be to provide a well-defined semantics.

One endeavor I feel worth a good bit of effort is the study of representation semantics. Brian Smith [1977] has begun work in this area, and I would urge that this field receive more attention. The benefits of such a theory are numerous. Smith makes a particularly nice statement of the practical use of such a theory:

> "To put this point in a more practical light, suppose that there are 8 representation systems, named A through H. In order to evaluate them, it is necessary to know what they are saying. It is not the role of a theory of semantics to say that one of them, say system G, is empirically correct. Nor should it say that three of them are better than the other five. Instead, it should identify whether each of them is coherent and internally consistent, and it should set forth just what the empirically-testable claims are that each of them makes. Given this semantic basis, one is then in a position to evaluate their theoretical content and their empirical validity." [Smith 1977, p. 19]

In the rest of this section, I will discuss aspects of each of the following representation languages: FRL [Goldstein and Roberts 1977], OWL [Szolovits, Hawkinson and Martin 1977], and KRL [Bobrow and Winograd 1977b]. The NETL system [Fahlman 1977] will be discussed in Appendix 2. One of the major difficulties in understanding these systems is the lack of published description. FRL and NETL are outstanding in the precise, detailed nature of the specification of their operation, and, in FRL's case, an easy to use public version of the system with which to experiment. KRL and OWL have, unfortunately,

few published details with which to understand their systems. There is no publicly usable version of NETL or (as far as I know) KRL-0. The only existing implementation of OWL is based on a representation with far less descriptive power than that described here.

FRL

FRL is a language for manipulating data structures called "frames". FRL frames are a generalization of property lists. A frame is a structured entity that can have an arbitrary collection of properties called "slots". Each slot has several "facets" describing various types of information. Each facet contains several "datums", and each datum can be annotated with a list of "comments".

FRL has three primary mechanisms with a system imposed meaning, and several other features whose meanings are up to the user. The most important mechanism is that of inheritance. Any frame can have a special slot, the "AKO" slot, which contains a list of other frames from which the frame inherits slot information. If a request for some slot information is made of a frame, all of the AKO pointers will be followed recursively, and all information in these ancestor frames which matches the request will be returned. That is, if the WEIGHT slot interrogated but does not exist in the CLYDE frame, the AKO pointers will be traced upwards through ELEPHANT, MAMMAL, ANIMAL, and PHYSOB (or some such hierarchy) until the desired information is found. Inheritance operates by keyword matching. Additional procedural or indirection information must be used if a slot at one level means the same as a differently named slot at another level. For instance, the frame definitions must specify special information if requests of the TRUNK of CLYDE are to inherit from the NOSE of MAMMAL.

Thus FRL inheritance could act as a virtual copy mechanism (see Appendix 1) for the frame data type. Unfortunately, the FRL authors do not explain its use as such. Instead, they offer it as a uniform mechanism for all types of inheritance. For example, [Goldstein and Roberts 1977] presents an AKO hierarchy in which GROUP AKO PEOPLE (which is fine, as both GROUP and PEOPLE denote sets of persons), and PERSON AKO PEOPLE, which is clearly wrong. And, although they acknowledge that problems exist, the FRL Primer [Roberts and Goldstein 1977a] suggests using AKO relationships to represent part/whole and other types of relationships.

FRL has another mechanism for default values of slots. Any

slot can have a set of default values associated with it. Under the normal inheritance procedure, if the value (a particular facet of a slot called the "$VALUE" facet) is not to be found in the frame or its ancestors, a similar search is run on the frame and its ancestors for default value specifications. If any defaults are found, they are used as the answer.

FRL has a mechanism for procedural attachment. Any slot can have "$IF-ADDED", "$IF-REMOVED", and "$IF-NEEDED" facets which are run whenever a value is added to, removed from, or requested of the $VALUE facet.

There are several conventions in FRL which do not yet have a system defined meaning. These include the "$REQUIRE" and"$PREFER" facets and the standard "CLASSIFICATION" slot of frames. The $REQUIRE and $PREFER facets are used to specify requirements on and preferences about values for frame slots. However, these only have meaning to a matching procedure. As yet, FRL has only a rudimentary matcher which has not been documented. (A forthcoming paper by Rosenberg and Roberts is promised.) Thus any value may be specified for a frame slot. It is up to the user to call the predicates and functions in these facets and to interpret the results. This means that there is really no way to create a defined concept (that is, a frame defined by certain properties) and have the system treat the new concept as such.

The CLASSIFICATION slot is used to distinguish, somewhat confusingly, between frames representing individuals and frames representing prototypes or sets. Presumably, this classification would be of interest to the matcher also. Currently, however, nothing seems to use this distinction: one can even create an instance of an individual (which by inheritance is itself an individual). In fact, one can have a generic frame as an instance of an individual. This may be useful: the point is that the system does not care.

Just as FRL has no system defined meaning for individuals, there is also no explicit way of splitting the subframes of a frame into distinct classes. If one wishes to represent the fact that mammals and reptiles are distinct types of animals, the best that can

be done is to give some slot of mammal (such as bloodedness) a requirement that will be true for mammals and false for reptiles (such as WARM-BLOODED?).

There is also no system convention for distinguishing between the typical member of a class and the class itself. This is necessary for reasoning that the size of the typical person is distinct from the size of the set of persons. In fact, the confusion of the AKO hierarchy presented in [Goldstein and Roberts 1977] mentioned above (PERSON inherits from PEOPLE) can probably be traced to this missing convention and to the use of the GENERIC classification to distinguish sets from individuals.

Possibly the major extension called for in FRL would be the ability to describe the frame concepts by name rather than by having to write programs to effect these descriptions. Several of the limitations of FRL pointed out by its authors can be traced to this lack. For example, one cannot attach procedures to arbitrary forms, but only to values: similarly, one cannot attach comments to a subset of the slots of a frame. A related situation is that FRL has no context mechanism. This also require making explicit all entities which might be context dependent. Most of the problems I have mentioned are also due to the second-class nature of frame slots. For example, one frequently useful statement to make about a slot is the type of its filler. In FRL, this must be done by saying ($REQUIRE (AKO? :value <type>)). This is embedding a clearly declarative type of information about the slot itself within a (possibly obscure) LISP function.

Another possible extension would be to make FRL a programming language. At present, it is strictly a collection of functions for manipulating the frame data structures. Although one can attach three standard forms of procedures to slots, there is no easy way to control their application. All real control must be written in LISP, and cannot be described and used in terms of frames themselves. I understand that experiments in using frames as rules or productions have been undertaken, but naturally such mechanisms require substantial development of the frame matching system.

OWL

OWL is a formalism for representing knowledge which is intended to be as close to English as is convenient. The basic entity in OWL is called a "concept". Concepts are constructed by "specialization" of other concepts. Thus each concept is of the form C = (genus specializer), where "genus" is the parent concept being specialized, an the "specializer" is a symbol or a concept which distinguishes C from other specializations of the genus. Specializations of a genus are intended to be subtypes of the genus, so that a concept usually inherits properties from its genus. There are several flavors of specialization, each conveying a different sense and a different set of rules for inheritance.

Information is also expressible as "attachments". The attachment of concept A to concept B is notated as [A B]. Attachments are something like universally accepted statements, with meanings including "B is the value of A", "B is a characterization of A", and "B is an attribute of A". Each concept has an associated "reference list", which contains pointers to

[1] all specializations of the concept directly under the concept in the hierarchy,

[2] some concepts whose specializer is the concept in question (these are called the "indexed aspects" of the concept), and

[3] to all other concept to which the given concept is linked by attachment.

As mentioned above, there are several discriminations in the forms of specialization. These are indicated by one of seven meta-attributes as (F*<meta-attribute> G). In such a concept the genus is the entire combination F*<meta-attribute>, and not just F alone. The forms of specialization are as follows:

[1] *R - Restriction. (F*R G) is an F required to have property G.

[2] *T - Stereotype. (F*T G) is a type of F, but does not necessarily derive any properties from G.

[3] *S - Species. (F*S G) represents a subspecies of F which is disjoint from the species represented by (F*S H) for any H distinct

from G.

[4] *I - Instance. (F*I G) is an instance of F, which is distinct from all other instances of F.

[5] *A - Aspect. (F*A G), in other terminology, is a slot of type F in the description of concept G.

[6] *X - Inflection. (F*X G) is used to specify grammatical or interpretive modifications of F, where properties are derived from both F and G.

[7] *P - Partitive. (F*P G) denotes semantic inflection. Its properties derive from either F or G depending on the context of interpretation.

Two other forms of expressing information in OWL are predication, in which a concept represents the assertion of a property of another concept, and naming, in which one concept is declared to be the name of other concepts.

It is my feeling that there is an interpretation of all of these mechanisms which is semantically clear and useful, or nearly so. Unfortunately, the existing discussions of OWL do not discuss any of the details necessary to understanding the meaning of most of these mechanisms. For example, concepts normally inherit properties from their genus. The precise cases for this need to be spelled out. But even before that is done, what does property inheritance mean in OWL? Are the values, characterizations, and attributes kept on reference lists inherited? Are their reference lists inherited as well? Are names and predications inherited? How is naming implemented? What are the allowable combinations of specializations? Can an instance itself be instanced? I am told that there are fairly clear ideas about several of these questions in the minds of the designers, but also that they are still evolving.

For example, it would seem that one can describe almost hierarchical structures in OWL, since aspects of concepts are themselves concepts. However, to implement the links stating an equality between two concepts requires some definite mechanism whereby any property discovered for one will be inherited by the other. Perhaps mutual naming or characterization can be used to implement

this, but since the exact nature of these mechanisms has not been defined, I cannot say how this would be done.

One mechanism which has been described, at least in terms of the above mechanisms, is that of the inheritance of slots of concepts. This is done by a mechanism called "derivative subclassification" in which specializations of concepts are organized to reflect the hierarchy organizing the specializers. By this means, if both PIG and DOG have the genus ANIMAL, then (TAIL PIG) and (TAIL DOG) are arranged under (TAIL ANIMAL). Thus if (TAIL ANIMAL) exists and has some property, then both PIG and DOG will inherit this aspect, and (TAIL PIG) and (TAIL DOG) will have the property by inheritance as well. This mechanism is also important in the function calling method used in the OWL-I interpreter.

Another aspect of OWL (which presumably would occur in other systems as well) is that two distinct hierarchies are needed, one for semantic concepts and one for grammatical concepts. This is because OWL, as a language understanding system, needs one classification in terms of grammatical characteristics of text symbols (which would contain the word "dogs"), and another classification in terms of the ideas they denote (which would contain "a set of dogs" but not "dogs"). There are also many interwoven hierarchies in the world. Perhaps these can be represented using the reference list mechanisms, but as I stated above, I cannot tell. This distinction would seem useful in aiding the implementation of a Weyhrauch-like meta-circular description of OWL.

One of the most exciting aspects of OWL is that it can interpret its own representations as programs. Unfortunately, the existing interpreter is based on a far less expressive theory than that mentioned above. Once the existing theory is defined precisely, I feel it would be valuable to have a working interpreter. I would suggest, as a source of problems to solve in working out the semantics of the language, that an attempt be made to write an OWL interpreter in OWL itself. In many cases, this dramatically highlights any deficiencies existing in the expressiveness of the representation or in the definitions of the semantics of the representation.

KRL

KRL-0 (hereafter KRL) is similar in a number of ways to FRL, but semantically much less clear and practically more complex to program. A useful way of understanding KRL is as FRL in a baroque setting of features. KRL "units" are essentially the same as FRL frames. KRL provides further specification and defaults corresponding to FRL AKO relationships and defaults, although Bobrow and Winograd [1977b] state that the basic KRL inheritance mechanism does not effect inheritance of slots. (I am unable to tell what it does do.) As in FRL, slots are second-class objects, so that multiple descriptions cannot have identified substructures. KRL also lacks a context mechanism. KRL units come in several flavors. Since KRL has lots of matchers, these distinctions generally mean something, although what they mean is up to the particular matchers being used. The categories of units are:

[1] Basic, which describe distinct prototypes,
[2] Specializations, which are non-exclusive refinements of Basics,
[3] Abstract, which are untyped prototypes, as opposed to Basics,
[4] Individuals, which are distinct from each other,
[5] Manifestations, which are "Ghosts" or instances of Individuals,
[6] Relations, which are prototype statements of relationships, and
[7] Propositions, which are instances of Relations.

KRL includes an elaborate matching framework, and several unspecified standard matchers. One of the assumptions of the KRL authors is that the matching process is the basis for most of the processing in a KRL-based system. This is reflected in the fact that most of the complexity of the matching framework is to allow arbitrary forms of control to be written into the matcher. This allows matches ranging from simple syntactic comparisons to full-fledged multiprocessing backtracking recognition procedures. However, no language is provided for controlling the matching process. To implement the match control structure, one uses either the underlying INTERLISP or the KRL agenda. This lack of coercion by KRL on the structure of the matching process means that KRL really has nothing to say about the matching process - all meaning and actual content is up

to the procedures written by the user.

KRL also includes a control structure based on closures and queues. Like FRL, KRL has IF-ADDED and IF-NEEDED (but not IF-REMOVED) procedural attachments. The basic control primitive available is the insertion of a process into the priority queue. A global scheduler is responsible for actually running the processes on the queue. The processes are specified by the procedure names in addition to a named environment. The mechanisms for handling these funargs are called "procedure directories" and "signal tables". For any really complex control, the KRL must resort to INTERLISP, since the language for controlling the agenda consists simply of the process priorities.

## Conclusion

There are several difficult problems to be faced in constructing hierarchical systems for representing knowledge. The distinction between individuals and prototypes is not sharp. The quality of being an individual often makes reference to some particular set of questions about the equality of the individual with other individuals. This is frequently a question that depends upon the goals of the descriptive and reasoning processes. To represent almost hierarchical systems in which multiple descriptions share structure requires that parts of structures be full-fledged objects of discussion themselves. This is necessary in order to state equivalences and other relationships between pieces of hierarchical structures.

Frame theories provide more descriptive power than many logic-based systems because they provide explicit handles on descriptions. This permits one to manipulate the descriptions, rather than just interrogating descriptions as occurs in logic-based systems. Explicit handles allow one to state how to use a description. This advantage does not only apply to the main objects of descriptive structure, the frames, but also to their parts, the slots. To forego handles on the internals of descriptive objects is to revert to a logic-like limitation on the use of these descriptions.

## Acknowledgements

I wish to thank Candy Bullwinkle, Johan de Kleer, Scott Fahlman, Lowell Hawkinson, Beth Levin, Bill Long, Marilyn Matz, Bruce Roberts, Steve Rosenberg, Brian Smith, Guy Steele, Gerry Sussman, and Peter Szolovits for many helpful discussions. I have been supported during this research by a fellowship of the Fannie and John Hertz Foundation.

References

[Bobrow and Winograd 1977a]
Daniel G. Bobrow and Terry Winograd, "An Overview of KRL, a Knowledge Representation Language," Cognitive Science, Vol. 1, No. 1, 1977.

[Bobrow and Winograd 1977b]
Daniel G. Bobrow and Terry Winograd, "Experience With KRL-0, One Cycle of a Knowledge Representation Language," IJCAI-5, pp. 213-222, August 1977.

[Fahlman 1977]
Scott E. Fahlman, "A System for Representing and Using Real World Knowledge," MIT Ph.D. Thesis, September 1977.

[Goldstein and Roberts 1977]
Ira P. Goldstein and R. Bruce Roberts, "NUDGE, A Knowledge-Based Scheduling Program," MIT AI Lab AI Memo 405, February 1977.

[Hayes 1974]
Patrick J. Hayes, "Some Problems and Non-Problems in Representation Theory," Proceedings of the AISB Summer Conference, 1974, pp. 63-79.

[Hayes 1977]
Patrick J. Hayes, "In Defense of Logic," IJCAI-5, August 1977, pp. 559-565.

[Moore and Newell 1973]
J. Moore and A. Newell, "How Can MERLIN Understand?," Department of Computer Science, Carnegie-Mellon University, November 1973.

[Roberts and Goldstein 1977a]
R. Bruce Roberts and Ira P. Goldstein, "The FRL Primer," MIT AI Lab AI Memo 408, July 1977.

[Roberts and Goldstein 1977b]
R. Bruce Roberts and Ira P. Goldstein, "The FRL Manual," MIT AI Lab AI Memo 409, September 1977.

[Smith 1977]
Brian C. Smith, "Levels, Layers, and Planes: The Framework of a Theory of Knowledge Representation Semantics," forthcoming MIT Masters Thesis, 1977.

[Stallman and Sussman 1977]
Richard M. Stallman and Gerald Jay Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," Artificial Intelligence, Vol. 9, No. 2, (October 1977), pp. 135-196.

[Sussman 1977]
Gerald Jay Sussman, "SLICES: At the Boundary Between Analysis and Synthesis," MIT AI Lab, Memo 433, July 1977.

[Szolovits, Hawkinson and Martin 1977]
Peter Szolovits, Lowell B. Hawkinson, and William A. Martin, "An Overview of OWL, a Language for Knowledge Representation," MIT LCS, TM-86, June 1977.

[Woods 1975]
William A. Woods, "Whats in a Link: Foundations for Semantic Networks," in Bobrow and Collins, editors, Representation and Understanding, pp. 35-82.

## Appendix 1: Storage Efficiency

One important benefit possible in systems with first-class entities for entity substructures is an efficient representation of storage. The basic idea is that if one can indicate and use the equivalence of descriptions, then one can always store new information in the form that requires the least amount of new structure to be added.

Scott Fahlman [1977] has developed a system which uses such a space economizing representation. He describes a method for representing hierarchical data structures which uses no more storage than is logically necessary, and a set of algorithms for manipulating this storage representation. In addition, the representation can be extended to cover several other types of knowledge representation of interest to artificial intelligence research. These extensions will be discussed later. As an extra bonus, most of these algorithms can be implemented in a radically parallel fashion which makes many previously expensive computational steps cheap and widely usable.

The basis of this theory is the concept of virtual copies of data structures. The virtual copy concept captures the essence of what is commonly called inheritance and makes the nature of this operation clear. Several previous attempts at defining inheritance have been semantically unclear, leading to inexpressible concepts or erroneous operation. These problems are discussed at length by Woods [1975] and by Fahlman. The following is Fahlman's own description of the virtual copy concept:

> "What we really want is to create virtual copies of entire descriptions. These descriptions can be arbitrarily large and complex pieces of semantic network. When we learn that Clyde is an elephant, we want to create a single VC link from CLYDE to TYPICAL-ELEPHANT and let it go at that, but we want the effect of this action to be identical to the effect of actually copying the entire elephant description, with the CLYDE node taking the place of TYPICAL-ELEPHANT. It must be possible to augment or alter the information in this imaginary description without harming the

original. It must be possible to climb around on the imaginary copy-structure and to access any part of it in about the same amount of time (speaking in orders of magnitude) that would be required if the copy had actually been made. But we want all of this for free. We just cannot afford the time or memory-space necessary to actually copy such large structures whenever we want to make an instance or a sub-type of some type-node, especially since the type-node's description may itself be a virtual copy of some other description, and so on up many levels. A description may also contain other descriptions -- the parts of an object, for example -- that are themselves expressed as virtual copies. We want all of this structure to be virtually, but not physically, present." [Fahlman 1977, pp. 32-33]

The virtual copy concept is a data structure concept, and is not a set-theory or predicate-calculus concept (at least in current formulations of these theories).

[a] VC is not MEMBER. Here I am interpreting MEMBER in set-membership terms; that is, if B is a set, known to be such that each of its elements has certain properties, then the assertion A MEMBER B means that A shares all of these properties. That is, exceptions cannot be made.

[b] VC is not PART-OF. Many early semantic net schemas are incorrect due to transmission of properties across a variety of relationships. Although on some occasions properties of the whole can be "inherited" in some sense by the parts (for example, the wings of a yellow canary are themselves yellow), this is not true in general. In particular, it is usually not true that the parts are the same sort of object as the whole. Systems which use the prototype/instance inheritance mechanism to also transmit properties from wholes to parts are simply looking for trouble and courting confusing complexity. For example, one might find that the weight of a tree is transferred to its arboreal inhabitants. ("What is yellow, weighs 4500 pounds, and goes "cheep, cheep"?") Such a use of an inheritance mechanism must work by forcing the explicit inhibition of most properties, rather than explicitly

indicating the few actually inherited properties.

[c] VC is not INSTANCE. Here I am interpreting INSTANCE in a predicate-calculus sense; that is, if A INSTANCE B, then A can be thought of as a term derived from B by substituting another term in for some variable occurring free in B. Like the MEMBER interpretation above, this does not allow exceptions to be made.

Instead, one can view the meaning of A VC B as saying that during any query, a question about A can be asked by making an actual copy of the structure described by B, using side effects to make any modifications local to A, and then asking the question of this clobbered instance. The extensibility comes from the ability to add parts to B's structure, and allowing the instance to acquire any modifications made to the parent.

Fahlman presents algorithms for searching through data structures defined by VC links, modifiers, and substructure representations. These algorithms allow a minimal amount of space to be used to represent the structures, but at the cost of introducing some search into the algorithms. This element of search is invisible in the parallel algorithms, but makes a difference in serial simulations. I here describe the problem, and point out how the search can be avoided by using more space than is logically necessary. I couch my language in data structure terminology, although Fahlman data structures are somewhat different from the normal concepts.

Suppose there is a prototype record called PERSON. One can view this prototype as a generalized CONS function which creates individual PERSON records on request, each of which will act as though it has all the default fields for PERSON filled with their default values. Actually, the instance will have no storage allocated for the fields unless necessary to fill one with a non-default value. One of the fields a PERSON record can have is a SON field, whose value is another instance of PERSON. Since this other instance of PERSON has itself a SON field, each PERSON record has implicit in it a field for the SON of the SON. We can make this an explicit field, called GRANDSON, of the PERSON record prototype. We then create a SON

substructure for the SON field of PERSON and equivalence this to the GRANDSON field.

The algorithms developed by Fahlman allow the following to be done. Let FRED be an instance of the PERSON record. We may learn that the SON of FRED's SON is a world-conquerer called ALEXANDER. Since we have nothing to say about FRED's SON explicitly, we can skip a level and get by with allocating a GRANDSON field for FRED and filling it with ALEXANDER. This avoids having to create an instance of PERSON for FRED's SON. The algorithms presented by Fahlman would still allow the access of information about FRED's SON, namely that the SON of the SON of FRED is ALEXANDER. If we later wish to say that FRED's SON is PHILIP, who also has other properties ("from Macedonia"), we can allocate the SON field for FRED and fill it with PHILIP. Note that the location of ALEXANDER is unchanged. We still access the SON of PHILIP by accessing the GRANDSON of PHILIP's father. Thus, while the algorithms allow us to skip unimportant levels in describing data structures, and so conserve space, a certain amount of search enters the system. To find out properties of the SON of PHILIP, we need to do a search to find out exactly where this field is stored - as the SON of PHILIP, or as the GRANDSON of FRED, or as some other field of some other record containing an implicit field for ALEXANDER, such as the GREAT-GRANDSON of his great grandfather.

Note that the search for the location of specific fields can be avoided simply by always allocating records in linear order using anonymous individuals if necessary to fill in the unspecified levels. This means that implicit fields are never allocated, that fields are never stored under collapsed specifications like GRANDSON. Thus in the example above, we can avoid that specific type of search by allocating a record for PHILIP when we want to say something about ALEXANDER. This allocates a record which may never have any fields filled, but means that we can always locate a field described as "the A of the B of the C of the ... of Z" simply by a series of trivial lookups at records beginning with record Z. This added burden of space may be too expensive, particularly in domains like electronics in which hierarchies for specific devices are many levels deep. With added depth of the hierarchy, the amount of wasted space becomes enormous,

leading to practical difficulties in implementing programs.

Appendix 2: NETL

NETL [Fahlman 1977] is a system for representing several types of knowledge, organized in such a way that a structure sharing scheme with clear semantics is implemented (assuming parallel hardware) with extraordinary efficiency. I have already presented a discussion of the storage representation concepts in NETL in Appendix 1. I will not discuss the parallel aspects of NETL.

NETL includes a mechanism for defining concepts in terms of others, a mechanism for procedural attachment, and a matching system based on the parallel architecture driving these functions. In addition, all nodes and links are themselves objects of discourse, so reasoning about the system's knowledge is possible. Fahlman also presents mechanisms for describing several important concepts including standard relationships within structured objects, space, time, existence, and quantification.

There are no second-class entities in NETL. (Actually, one can make some structures second-class for efficiency if it will never be necessary to discuss them as objects.) The basic system consists of several types of nodes and links. The most important node types are as follows:

[1] *INDV - An individual node. These represent individual entities (not prototypes) within some universe or roles (subcomponents, slots) within some structure (frame, unit).

[2] *TYPE - A type node. These represent the description of the typical member of some set, and serve as prototypes for creating *INDV nodes.

[3] *MAP - A map node. These are used to selectively copy roles of some prototype description into a specialization of that prototype.

[4] *IST - An individual statement node. This is basically an individual node which is an instance of some statement schema description (which centers around a *TYPE node).

[5] *EVERY - A class definition node. This is a type node used to represent an intensional definition of a class. The definition is accomplished by marking some of the properties of the node as

required properties. The node also serves as the prototype of the typical member of the class.

These nodes are used in conjunction with several link types, the most important of which are:

[1] *VC - A virtual copy link. Connects a copy to its prototype.
[2] *EQ - An equality link. Connects two compatible views of the same entity.
[3] *CANCEL - A cancellation link. Used to modify virtually copied descriptions.
[4] *SPLIT - A split link. Used to create a set of non-intersecting classes for clash detection purposes.
[5] *EXFOR - An existence link. Used to link a role to the description it modifies.
[6] *EXIN - An existence link. Used to link a role to the description in which it exists.
[7] *SCOPE - A scope link. Used to indicate the context (the area of validity) of a statement.

These links can be modified by several types of flags. These flags control the particular behavior of virtual copying and activation in different circumstances. I will not go into them here, as they are described in detail in Fahlman's thesis.

The basic operations in NETL are creating, copying and modifying descriptions. These use *EVERY nodes, *TYPE nodes, *INDV, *VC and *IST links in fairly straightforward fashions. The system performs type checking by examining conflicting *SPLIT nodes, and context maintenance by using the *EXIN and *SCOPE links. Context creation is very easy in NETL, since new contexts can be pushed onto old ones by means of a single *VC link. The system provides unsurprising means for representing space and time relationships, adapted to the parallel architecture. These are used by the context system, among other things. These temporal and spatial representations are yet to be fully developed and tested for expressiveness and usefulness.

NETL makes a clear distinction between a set and the typical member of the set, although it makes no commitment about what the description of the typical member should be. It also makes a clear distinction between prototypes and individuals. It allows several levels of splitting among prototypes, as opposed to KRL, which apparently requires that all type splits be specified at one level as Basic units. But this means that one cannot make a virtual copy of an individual in NETL. However, individuals can be mapped into sets of hypothetical contexts, creating the effect of an individual acting as a prototype.

It is possible to make procedural attachments in NETL by defining the triggers of demons as statements and using the matching procedures to find all of the demons with triggers matching a certain statement. It is also possible to use *EVERY nodes as guides for IF-NEEDED procedures, since when looked at properly, these represent problem-reduction methods. However, NETL is not a procedural system. Although Fahlman discusses representations for states, actions and events, there are no NETL procedures as such. All programs must be externally supplied, and can operate by using the standard insertion, matching and retrieval mechanisms, or by using any serial technique desired for examining the knowledge base.