THE DESIGN OF A SYSTEM FOR DESIGNING KNOWLEDGE REPRESEMTATION SYSTEMS

James L. Welner
Computer Science Department
University of New Hampshire

Merthe Palmer
Computer end Information Science Department
Univerelty of Pennsylvania

ABSTRACT

The use of ebstrect dete types as a basis for designing experimental knowledge representation systems is discussed. Abstrect dete types ere shown to heve features in common with severel diverse representation formalisms (e.g. semantic networks, frames and KLONE). For example, ebstrect data types have notions analogous to concept, subconcept end inheritance. The relatively small conceptual distence between abstrect data types and knowledge representation formalisms make them en ideal vehicle for implementing such formalisms.

I INTRODUCTION

Representation of knowledge is of primary importance to programs dealing with real world situations. There have been several attempts to derive an appropriate formalism that would help to minimize the problems in this difficult area. There heve been just as many resulting methodologies, offering widely different solutions: Frames [Minsky, 1975], Semantic Networks [Qulllian, 1968], KRL [Bobrow and Wlnograd, 1977], First-Order Logic [Hayes, 1977], and KLONE [Brachman, 1979]. These different methodologies ere based on fundamentally different assumptions about the neture of knowledge, each with convincing claims for its precedence. In order to compare these assumptions it would be necessary to find a common basis of representation. Ue contend that the different formalisms can all be expressed in logic combined with abstract data types, and that this proves to be a useful and Informative tool for designing knowledge representation systems.

Logic with abstract data types (ADTs) offers a solid foundation that already Includes many of the facilities built into knowledge representation languages. It also allows a system to be built that combines features from different formalisms, providing an ideal testing ground for purposes of comparison. There are also inherent advantages in the choice of logic with ADTs. By writing programs in logic, especially clausal logic [Kowalskl, 1980], we have both a mathematical semantics or specification and an operational semantics that allows us to execute thet specificetion, ADT's have the same advantage, whether defined algrebraically [Zllles, 1975, Goguen, Thatcher and Wegner, 1977] or model-theoretically [Nourani, 1980, Van Emden and Maibaum, 1980]. This allows the builder of the knowledge base to concentrate on defining concepts independently of how they might be implemented or executed, and thus allows the designer of the system to concentrate on issues of knowledge representetlon, rether thet programming. ADTs have the advantege of providing a powerful tool for structuring the masses of knowledge required in modelling even the most trivial situation.

The rest of this paper demonstretes the usefulness of this approach. Examples are given using an extension of PROLOG [Warren, 1977] that supports ADTs. This extension is based on the languages HOPE [Burstall, MacQueen and Sannella, 1980] and OBJ [Goguen and Tardo, 1979] with most of the syntax taken from HOPE, A similar extension was proposed by Van Emden and Maibaum (1980).

II SEMANTIC NETWORKS

We view semantic networks as a combination of logic and ADTs. This view differs slightly from e view held by many, notably Hayes (1977), which states that semantics networks are equivalent to e set of assertions in First-Order Logic and that the only velue they heve is as syntactic sugar for those essertions. In this view, the network in Figure 2-la would be equivalent to the essertions in Figure 2-lb and the network in Figure 2-lc would be equivalent to the assertions in Figure 2-ld.

Although this view sheds important light on the status of semantic networks as a formalism, it ignores their velue as an important structuring technique. That is, one can also view Semantic Networks as both e structure and a set of rules

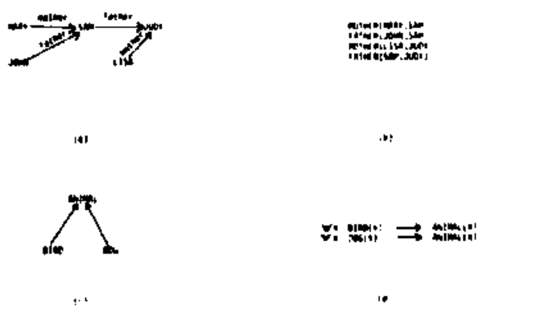

Fig. 2-1: Semantic Networks as Assertions

for Interpreting that structure Independent of the specific nodes or arcs an instance of a network contains.

In our view, a Semantic Network can most clearly be represented as an ADT. The ADT defines the structure of the network and defines operations which interpret that structure.* To understand how one sight go about defining a Sematic Network using ADTs, consider the following:

```
(1)  data semantic_net(alpha,beta) ==
        mkksn (set node(alpha)#set arc(alpha,beta))
(2)  data node (alpha) == mknode (alpha)
(3)  data arc (alpha,beta) ==
        mkarc (node(alpha)#node(alpha)#beta)
```

A Semantic Network is defined in (1) as consisting of a set of nodes (defined in (2)) of soma type alpha and a set of arcs (defined in (3)) of type alpha and beta (which are the type of the node and of the label respectively). Alpha and beta are variables ranging over types. Data definitions such as (1) are used to introduce a new data type along with the constructors which create elements of that type, mksn ("make semantic network") is a function** which takee~"se"ts"~of nodes and sets of arcs between these nodes and constructs a semantic network from them. A definition of a parameterized type, such as (1), is not really a type, but a type constructor. It constructs different sorts of semantic nets, depending on the type (or types) supplied as a parameter. Por example, if we define mary. jam, judy. John and llea as persons:

```
(4) data person == mary ++ sam ++ judy ++
                              john ++ lisa
```

and mother and father as relations between members of a family:

```
(5)data family_relationships == mother ++ father
```

we can then define a family network as in:

```
(6)data family_net ==
    mkfn(semantic_net(person,
         family_relationships))
```

which defines a type of semantic network whose nodes are of the type person (where Judy, sam etc. are constants of that type) and whose arcs are labelled by family relationships (where mother and father are constants of that type).

Now, since we defined a semantic net as really a constructor of semantic nets, we can use it to define a simple form of partitioned network iHendrix, 1979]. To do this, we simply view a partitioned semantic net as a network whose nodes are themselves networks. For example, if we define:

```
 (7) data neighbor_relation == neighbor
and then
 (8) data neighbor_net ==
     mksn(semantic_net(family_net,
          neighbor_relation))
```

We have a network whose arcs are labelled by the neighbor relation, and whose nodes are networks of families. This could be used to answer a query like "Who is the youngest person in John's neigh boring family."

To actually complete the definition we would have to define operations to interpret the network. These, of course, are lengthy and will not be given here.

In the definition given above, we have tried to be consistent with the way semantic networks are traditionally defined. A slightly different formulation would be to distinguish different sorts of arcs, not by their labels, but by their type. To do this, we need to Introduce a corre-spondence between notions used to describe ADTs and notions used to describe semantic networks. Corresponding to concept is, of course type, to subconcept subtype, and to Inheritance the notion of coercion: a function that takes an object of one type and returns an object of another type. Using these notions, father_arc and mother_arc can be defined as subtypes of arc as follows:

```
(9)   data arc == mkarc(father_arc) ++
                  mkarc(mother_arc)
```

assuming mother arc and father arc were previously defined.

Using this formulation, any operation on arcs will apply to both father arcs and mother arcs, by having the system automatically coerce an object of type mother or father arc to type arc.

We can also get rid of arcs altogether by treating them as operations whose source type is the type of the node at the arc's tail, and whose tsrget type is the type of the node at the arc's head (we would also have to include some environ-ment). Thus, Instead of incorporating knowledge about mother and father through the arcs and the rules which Interpret them, we Incorporate it through the semantics of the corresponding opera-tion. For example, we could define general opera-tions corresponding to mother and father as:

```
father: person, environment → person
mother: person, environment → person
```

along with a set of clauses which define how, given a person, his/her mother or father could be determined.

What we get by doing this is essentially a Frame-like representation. In the next section we will see how Frames can also be represented by ADTs.

*Glven this view, one could, in turn, argue that an ADT is also just syntactic sugar for a set of as-sertions, but this becomes like arguing that one should not prograa in a high level language. Since it is equivalent to a Turing machine.
**Our extended PROLOG provides a functional inter-face.

III FRAMES

In this section, we will show how ADTs can be used to implement Frames, by comparing a definition of a date frame, written in KRL, a frame-based language, with a definition of a date ADT written in our extended PROLOG.

First, consider the definition of date in KRL (taken from [Bobrow and Winograd, 1977])

```
[date
     month name (when filled reset-day)
     day        (bounded integer 1-31)
     year Integer (to fill assume 1975)
]
```

A date consists of three slots, month, day, and year which are to be filled by a name, an integer between 1 and 31 and an integer respectively. Attached to the month and year slots are procedures which are to be automatically activated when their associated conditions are satisfied. These procedures must contain a great deal of task specific information such as the number of months in a year, days in a month etc. The procedure attached to the month slot will reset the day slot back to 1 whenever the month slot is filled, and the procedure attached to the year slot will return the Integer 1975 whenever the slot is referenced before it is filled.

Now consider the definition of a date ADT. First, we need to define a year, a month-date and a day ADT.

(10) data month__name == January 4+ febuary ++ march ++ april ++ may ++ June ++ July 4+ august ++ September ++ October ++ november ++ december

(11) data year — mkyear(num)
(12) data day == mkday(num) ++ out-of-bounds

The function mkday which takes a number and returns a day is a hidden function; it is not accessible to the user. Instead, the user interfaces to the type day thru a function day declared as:

day:num --> day

and defined by the following clauses:

```
day(n) < = mkday(n) if n =<31, n >= 1
day(n) < = out-of-bounds if n > 31
day(n) < = out-of-bounds if n >  1
```

which restrict the user to defining a day numbered between 1 and 31. More specifically, these clauses say that if the number passed to the function day is between 1 and 31 then a day corresponding to that number is returned (mkday(n)) otherwise out-of-bounds is returned, which indicates the appropriate error.

Using these definitions, we can define date as:

(13) data date — mkdate(month_name#day#year)

To create a date, the user uses a function date declared as:

date: month_name --> date

to take a month_name and return a date. The other functions which update and access a date are:

```
get-month: date → month_name
put-month: month_name, date → date
get-year: date → year
put-year: year,date → date
get-day: date → day
put-day: day,date → date
```

The semantics of date are given by the following rules or clauses:

```
(14) date(mn) <= mkdate(mn,mkday(1),mkyear(1975))
(15) get-day(mkdate(dm,dd,dy)) <= dd
(16) put-day(d,mkdate(dm,dd,dy)) <=
            mkdate(dm,d,dy)
(17) get-month(mkdate(dm,dd,dy)) <= dm
(18) put-month(mn,mkdate(dm,dd,dy)) <=
            mkdate(mn,mkday(1),dy)
(19) get-year(mkdate(dm,dd,dy)) <= dy
(20) put-year(y,mkdate(dm,dd,dy)) <=
            mkdate(dm,dd,y)
```

Rule 14 creates a new date, rules 15, 17 and 19 select from a date the day, month_name and year respectively, and rules 16, 18 and 20 update the day, month__name and year respectively and return a new date with those parts updated.

The work done by the attached procedures in the KRL definition is done by rules 14 and 18, Rule 14 will cause the year 1975 to be returned if the date is not updated before it is referenced. For example, get-year(mkdate (July))) will return 1975 but get-year(put-year(mkyear(1977), mkdate (July))) will return 1977. Rules 14 and 18 will reset the day when the month_name is filled. An advantage of the ADT definition is that we can represent knowledge about defaults, such as that the default year is 1975, and knowledge about what is implied by an action, such as that the day is to be reset to 1 when the month directly with the clauses without having to resort to an additional mechanism, as is the case in KRL.

In this section we have shown how Frames can be represented by ADTs. This should not be seen as detracting from the importance of Frames or KRL, since their importance is not as a system, but as a theory of knowledge representation which emphasizes ideas like prototypes and multiple descriptions.

In the next section we will consider the need for a methodology for building up a knowledge base, and whether one developed for knowledge representation can be used for ADTs and vice versa.

IV METHODOLOGY

Regardless of the formalism used, a knowledge representation system must embody a methodology for adding to its knowledge base. If not, the system will eventually be overcome with the complexlty of the knowledge needed to model the situation at hand. One system whose main contribution is such a methodology, which it calls an epistemology because of the broader context it is working in, is KLONE.

The KLONE methodology is based on concepts and their interrelationships. Concepts consist of roles (parts of the concept) and structural descriptions (relationships between the parts). The essence of the KLONE methodology is the ability to build up knowledge by generalisation and specialization. For example, once we have defined course with roles instructor and students, we can specialise it to the subtype service course which results in specialising, in turn, student and teacher fillers of those roles. One could, of course, go in the other direction, and first define service course and then course. Differentiation is another type of specialisation. First we define Instructor and then differentiate it into two subroles: lab instructor and lecturer.

In this section we will attempt to separate out the KLONE methodology from its implementation by showing how we can apply the methodology to the design of ADTs. In some ways, we can view KLONE as a user's interface to the design of ADTs where KLONE is the source language and ADTs are the target language of some translator.

The interrelationships we are concerned with here are

1. Subordination between concepts such that one concept is a subconcept (or superconcept) of another.

2. Individuation of a concept by another. For example, the concept Babe Ruth is an individual concept from the set represented by the generic concept baseball star.

3. Restriction on the type of object that can fill a role of another.

4. Differentiation of a role into subroles.

5. Satisfaction, or the relation between a role of a generic concept and one of its Individual concepts such that the filling of one is the same as filling of the other. For example, a toll may require 50 cents, but this requirement may be satisfied, Instead, by 3 quarts of chop suey.

The arcs have the following interpretation — arcs of the form A superc £ indicate that B is superconcept of A, arcs of~the form A ind * Indicate that A Is an individual concept of generic B, arcs of the for A V/R B indicate that the filler of role A must be ol type~B, arcs of the form A satisfies B Indicate that role A satisfies role B, arcs of the form A diff B indicate that A is a eubrole of B, and, finally, arcs of the form A role B indicate that B is a role of A.

Consider the KLONE description in Figure 4-1

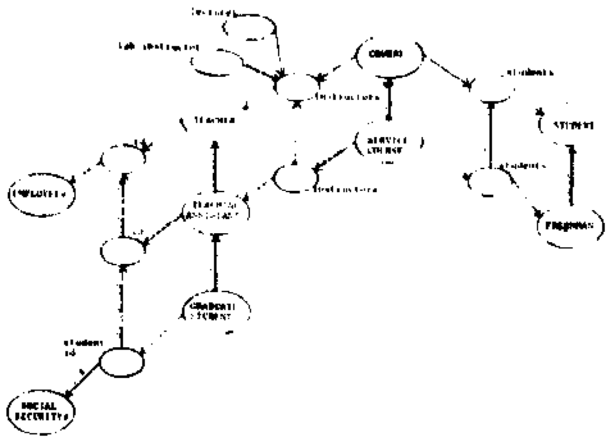

Figure 4-1: KLONE description*

Now let's define some of the ADTs that correspond to the above KLONE description. Starting with course, we see that it has roles instructor *nd students, and that instructor is differentiated into roles lab instructor and lecturer. Now to define a corresponding ADT for course, we need to know how to construct it. That is, what parts come together to form a course. These parts are, of course, the different roles. Therefore, one way to define the ADT course is as:

```
          data course — mkcourse(teacher!teacher!
                                    list student)
```

where one teacher is the lab instructor and the other is the lecturer. Roles also indicate that we need corresponding operations which access and update them.

The differentiation links indicate that we need operations which allow you to select out only those teachers which are lecturers and those which are lab instructors. One way to differentiate between these objects is to type them (or in this case subtype them). We can define instructor as:

```
(21) data instructor == mkinst(lab_instructor)
                        ++ mkinst(lecturer)
(22) data lab_instructor == mklab(teacher)
(23) data lecturer == mklec(teacher)
```

which says that instructors are either lab instructors or lecturers.

The above definition may appear odd, since the fact that instructors consist of teachers is specified twice, once in the definition of lab instructors snd once in the definition of lecturer, rather than specifying it only once in the definition of instructor and letting "inheritance" pass

♦Note that this network would only let a single graduate student be s teaching aasistant. Although clearly contrived, this allowed us to include a satifaction link without over complicating the network.

it down. The problem is that the data definition not only specifies knowledge at the conceptual level, but, at the Implementation level, specifies how to construct it. For which, like any function in any programming language, we need to know what its parameters are and their order. This information, although necessary at some level, is probably not necessary at the level in which the user Interacts with the system; this gives another reason for needing some interface, whether or not it is KLONE. Now we can change the definition of course appropriately.

```
(24)data course
    mkcourse(instructor#instructor//list  student)
```

The KLONE description above actually implies a different definition, because courses (in our very limited description) consist only of service courses. Therefore, the ADT should be defined as:

```
(25)data course == mkcourse(service_course)
(26)data service_course ==
     mkcourse(teaching_assistant#
     teachlng_assistantant# freshman)
```

Of course, the operations on course can still be applied to service course, because the system will automatically coerce a service course into a course* For example, assume we have an operation to select out the lecturer from a course, declared as:

```
get_lecturer:  course --> instructor
```

When it is applied to an object of type service course it will return an instructor who is a lecturer who is, in turn, a teaching assistant.

Nothing has to be added to handle restriction links if the value of the restricted role is just a subtype of the value of the unrestricted role. If not, all that has to be done is for additional clauses to be added to the definition of the operations corresponding to the role. A similar situation occurs for satisfaction links.

The additional clauses to implement a satisfaction link define a mapping between objects of one role and objects of another. Given operations that select the id[from a teacher and the student id from a graduate student, declared as:

```
get-id: teacher --> employees
get-student-id: graduate_student -->
                   social_security#
```

We can then add the following clause to define the mapping from the role student id to id:

```
get-id(grad) <■ ss#toemp0(get-student-id(grad))
```

where grad Is a graduate student which is coerced t0 teacher before being applied to get-id and ssfltoempd* is a function which changes social security numbers to employee numbers.

ADTs also come with their own methodology for building up knowledge which allows existing types to be combined and enriched to produce new types ([Goguen, Thatcher and Wagner, 1977, and Nouranl, 1980]). Although this methodology is not as specific and directed as the KLONE methodology, it is Important, because it guarantees that the new types are well-defined if the old ones were.

We have tried to show that the methodology provided by KLONE can be accommodated by using ADTs and therefore one can view KLONE simply as an interface to such an ADT system. This has the advantage that we can decouple the concerns of the methodology from its implementation.

V SUMMARY AND CONCLUSIONS

In this paper we have tried to demonstrate the usefulness of developing a system for knowledge representation that is based on a foundation of ADTs and logic. This foundation is strong enough to allow other systems supporting various formalisms for knowledge representation to be built upon it.

The advantages of this approach are many, including: easing the problem of implementing a system by providing a base, allowing several formalisms to be supported by one system, and, finally, providing a common language with which to compare representations in the different formalisms.

To demonstrate the usefulness of our approach, we have shown that two competing formalisms (Frames and Semantic Networks) can be supported by a system such as we propose, and that our foundation is consistent with a methodology for adding to a knowledge base which is already in use.

Several objects and situations have been modelled in a system designed along the proposed lines of this paper and run in our extended PROLOG. These include, a blocks world, a company world, and the largest one; an ATN and a lexicon for that ATN (following [Woods, 1979]).

ACKNOWLEDGEMENTS

We are indebted to Joe Goguen for sharing his insights into the nature of Abstract Data Types. We would also like to thank Rod Burstall and Farshid Nouranl for discussions about issues related to this paper, Gabrial Barta, Eugene Freuder and Chris Melllsh for comments on the structure of this paper, and Barbara Layne for preparation of this paper.

REFERENCES

[1] Bobrow, D.G., and Winograd, T. An overview of KRL, a knowledge representation language. Cognitive Science. 1977, 1(1), 3-46.

[2] Brachman, R.J. On the Epistomological Status of Semantic Networks. In N.V. Finlder (Fd.), Associative Networks Representation and use in

knowledge by Computers,: Academic Press, 1979.

[3] Burstall, R.M., D. MacQueen and D. Sannella. HOPE: An Experimental Applicative Language. In Proceedings of the 1980 LISP Conference.: The LISP Company, 1980.

[A] Goguen, J,A. and J.J. Tardo. An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications. In Proceedings of the Conference on Specifications of Reliable Software. Cambridge IEEE, 1979.

[5] Goguen, J.A., J.V. Thatcher, E.G. Wagner. An Initial Algebra Approach to the Specification of Abstract Data Types. In R. Yeh (Ed.), Current Trends in Programming Methodology, New York: Prentice-Hall, 1977.

[6] Hayes, Pat J. In Defence of Logic. In Proceedings of the 5th International Joint Conference on Artificial Intelligence,: IJCAI, 1977.

[7] Hendrlx, G. Encoding Knowledge in Partitioned Networks. In N.V. Findler (Ed.) Associative Networks: Representation and use in knowledge by Computers.: Academic Press, 1979.

[8] Kowalski, R. Logic for Problem Solving. Amsterdam: North-Holland 1980.

[9] Minsky, M.A. Framework for Representing Knowledge. In Patrick H. Winston (Ed.), The Psychology of Computer Vision, New York: McGraw-Hill, 1975.

[10] Nourani, F.A. Model-Theoretic Approach to Specification, Extension, and Implementation. In B. Robinet (Ed.), Proceedings of the Fourth International Symposium on Programming.: Springer-Verlag, 1980. Lecture Notes in Computer Science vol 83.

[11] Quilllan, M.R. Semantic Memory. In M. Minsky (Ed.), Semantic Information Processing. Cambridge: M.I.T. Press, 1968.

[12] Van Emden, M and T. Malbaum. Clauses versus Equations in the Specifications of Abstract Data Typea. In J.L. Minker and H. Gallaire (Ed.), Advances in Data Base Theory.: Plenum Press, 1980.

[13] Warren, D.H.D., L.M. Perelra, F.C.N. Perelra. PROLOG — The Language and Its Implementation Compared with LISP. In Proceedings of the Symposium on Artificial Intelligence and Programming Languages Rochester. N.Y.: ACM, 1977.

[14] Woods, W.A. Research in Natural Language Understanding: Annual Progress Report, 1 September 1978 - 31 August 1979. Technical Report 4279, BBN, 1979.

[15] Zilles, S. Algebraic Specification of Abstract Progress Report XI,: Laboratory of Computer Science, MIT, 1975.